# uboot配置和编译流程分析 2016-09-13

精简过的u-boot代码下载:<https://github.com/xfqjklw/uboot_lite_jz2440.git>

uboot编译过程分析主要从顶层的makefile开始下手，uboot编译主要分为两个步骤:配置和编译。

## 1.uboot配置流程

### 1.1配置方法

配置命令：make 100ask24x0\_config O=obj

100ask24x0\_config代表配置名称。

O=obj表示编译的中间文件和结果都放在obj目录下，obj目录会自动创建。

顶层makefile中和配置命令相关代码和注释如下

ifdef O

ifeq ("$(origin O)", "command line")

BUILD\_DIR **:=** $(O)

endif

endif

#如果make的时候有O=obj选项，则$(origin O)结果为command line,这个时候BUILD\_DIR的值为obj

OBJTREE **:=** $(if $(BUILD\_DIR),$(BUILD\_DIR),$(CURDIR))

SRCTREE **:=** $(CURDIR)

MKCONFIG **:=** $(SRCTREE)/mkconfig

unconfig**:**

@rm -f $(obj)include/config.h $(obj)include/config.mk \

$(obj)board/\*/config.tmp $(obj)board/\*/\*/config.tmp

#配置之前先删除上次配置产生的文件。

100ask24x0\_config **:** unconfig

@$(MKCONFIG) $(@:\_config=) arm arm920t 100ask24x0 NULL s3c24x0

#$(MKCONFIG)为根目录下的mkconfig脚本

#$(@:\_config=)表示将100ask24x0\_config中的\_config去掉

#等效为./mkconfig 100ask24x0 arm arm920t 100ask24x0 NULL s3c24x0

### 1.2 mkconfig脚本分析

uboot配置的核心脚本为顶层目录下的mkconfig脚本,mkconfig脚本的作用如下：创建平台/开发板相关头文件的链接

创建config.mk和config.h配置文件

mkconfig相关代码和注释如下

APPEND**=**no # Default: Create new config file

BOARD\_NAME**=**"" # Name to print in make output

**while** **[** **$#** **-**gt 0 **]** **;** **do**

**case** "$1" **in**

**--)** **shift** **;** **break** **;;**

-a**)** **shift** **;** APPEND**=**yes **;;**

-n**)** **shift** **;** BOARD\_NAME**=**"${1%%\_config}" **;** **shift** **;;**

**\*)** **break** **;;**

**esac**

**done**

# $#代表mkconfig执行时参数个数

# $1代表mkconfig第一个参数

# shift：删除掉第一个参数，将第二参数变成第一个

# -gt表示大于

# 这段代码意思 检查参数中，有无-a -n --这样的参数，没有的话直接break

# 其中

# -a表示 Append to existing config file，默认是Create new config file

# -n代表取得类似xxxxx\_config参数的xxxxx内容

**[** "${BOARD\_NAME}" **]** **||** BOARD\_NAME**=**"$1"

#["${BOARD\_NAME}"]表示一个条件，BOARD\_NAME != NULL, ||表示如果 条件为假 则运行后面的程序。

#BOARD\_NAME变量从声明开始一直是空，到这里给他赋值$1,即 BOARD\_NAME = 100ask24x0.

**[** **$#** **-**lt 4 **]** **&&** **exit** 1

**[** **$#** **-**gt 6 **]** **&&** **exit** 1

**echo** "Configuring for ${BOARD\_NAME} board..."

#

# Create link to architecture specific headers

#

**if** **[** "$SRCTREE" **!=** "$OBJTREE" **]** **;** **then**

**mkdir** -p ${OBJTREE}**/**include

**mkdir** -p ${OBJTREE}**/**include2

**cd** ${OBJTREE}**/**include2

**rm** -f asm

**ln** -s ${SRCTREE}**/**include**/**asm-**$2** asm

LNPREFIX**=**"../../include2/asm/"

**cd** **../**include

**rm** **-**rf asm-**$2**

**rm** -f asm

**mkdir** asm-**$2**

**ln** -s asm-**$2** asm

**else**

**cd** **./**include

**rm** -f asm

**ln** -s asm-**$2** asm

**fi**

# 当使用了O=obj选项时，SRCTREE！= OBJTREE。

# 在obj目录下先创建include和include2文件夹。

# 在obj/include2下建立include/asm-arm目录的链接asm，

# 在obj/include下建立asm-arm目录，并将asm目录链接到asm-arm上

# 如果未使用O=obj选项，则在include目录下建立asm-arm的链接asm.

**rm** -f asm-**$2/**arch

**if** **[** **-z** "$6" **-o** "$6" **=** "NULL" **]** **;** **then**

**ln** -s ${LNPREFIX}arch-**$3** asm-**$2/**arch

**else**

**ln** -s ${LNPREFIX}arch-**$6** asm-**$2/**arch

**fi**

**if** **[** "$2" **=** "arm" **]** **;** **then**

**rm** -f asm-**$2/**proc

**ln** -s ${LNPREFIX}proc-armv asm-**$2/**proc

**fi**

# $6为s3c24x0,所以创建obj/include2/asm/arch-s3c24x0的链接asm-arm/arch

# $2为arm,所以创建obj/include2/asm/pro-armv的链接asm-arm/proc

#

# Create include file for Make

#

**echo** "ARCH = $2" **>** config.mk

**echo** "CPU = $3" **>>** config.mk

**echo** "BOARD = $4" **>>** config.mk

**[** "$5" **]** **&&** **[** "$5" **!=** "NULL" **]** **&&** **echo** "VENDOR = $5" **>>** config.mk

**[** "$6" **]** **&&** **[** "$6" **!=** "NULL" **]** **&&** **echo** "SOC = $6" **>>** config.mk

#创建config.mk文件,内容如下

# ARCH = arm

# CPU = arm920t

# BOARD = 100ask24x0

# SOC = s3c24x0

#

# Create board specific header file

#

**if** **[** "$APPEND" **=** "yes" **]** # Append to existing config file

**then**

**echo** **>>** config.h

**else**

**>** config.h # Create new config file

**fi**

**echo** "/\* Automatically generated - do not edit \*/" **>>**config.h

**echo** "#include <configs/$1.h>" **>>**config.h

**exit** 0

# 创建config.h文件，内容为

# #include <configs/100ask24x0.h>

# 其中configs/100ask24x0.h为配置文件，定义了各种功能的开关，以此文件作为裁剪。

# 在common.h的开头便包含了include <config.h>也就是include <configs/100ask24x0.h>

## 2.uboot编译流程

### 2.1 相关变量初始化

相关代码和分析如下

VERSION **=** 1

PATCHLEVEL **=** 1

SUBLEVEL **=** 6

EXTRAVERSION **=**

U\_BOOT\_VERSION **=** $(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)

VERSION\_FILE **=** $(obj)include/version\_autogenerated.h

# 声明u-boot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/macppc/ppc/)

HOSTOS **:=** *$(shell uname -s | tr '[:upper:]' '[:lower:]' | \*

sed -e 's/\(cygwin\).\*/cygwin/')

export HOSTARCH HOSTOS

# HOSTARCH为编译机器的架构，我的电脑为i686

# HOSTOS为编译机器的操作系统，我的电脑为linux

ifdef O

ifeq ("$(origin O)", "command line")

BUILD\_DIR **:=** $(O)

endif

endif

ifneq ($(BUILD\_DIR),)

saved-output **:=** $(BUILD\_DIR)

# Attempt to create a output directory.

$(shell [ -d ${BUILD\_DIR} ] || mkdir -p ${BUILD\_DIR})

# 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),)

# make的时候是否增加了O=xxx选项，如果有，表明编译的中间文件将生成在xxx目录下

OBJTREE **:=** $(if $(BUILD\_DIR),$(BUILD\_DIR),$(CURDIR))

SRCTREE **:=** $(CURDIR)

TOPDIR **:=** $(SRCTREE)

LNDIR **:=** $(OBJTREE)

export TOPDIR SRCTREE OBJTREE

ifneq ($(OBJTREE),$(SRCTREE))

REMOTE\_BUILD **:=** 1

export REMOTE\_BUILD

endif

#如果OBJTREE不等于SRCTREE，REMOTE\_BUILD为1

ifneq ($(OBJTREE),$(SRCTREE))

obj **:=** $(OBJTREE)/

src **:=** $(SRCTREE)/

else

obj **:=**

src **:=**

endif

export obj src

#定义obj和src目录，如果有O=xxx选项，obj将会是xxx

ifeq ($(OBJTREE)/include/config.mk,$(wildcard $(OBJTREE)/include/config.mk))

include $(OBJTREE)/include/config.mk

export ARCH CPU BOARD VENDOR SOC

#如果存在obj/include/config.mk文件，则include该文件

ifeq ($(ARCH),arm)

CROSS\_COMPILE **=** arm-linux-

endif

export CROSS\_COMPILE

#如果ARCH是arm架构的，设置交叉编译器前缀

include $(TOPDIR)/config.mk

# 顶层目录的下的config.mk文件包含最终编译OBJ的规则，此外，包含gcc下的各种编译选项，链接选项，定义头文件包含的目录，库文件的路径等，是Makefile的核心

OBJS **=** cpu/$(CPU)/start.o

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

# 增加编译内容 cpu/arm920t/start.o，如果make的时候有O=xxx选项，增加$(obj)前缀

LIBS **=** lib\_generic/libgeneric.a

LIBS +**=** board/$(BOARDDIR)/lib$(BOARD).a

LIBS +**=** cpu/$(CPU)/lib$(CPU).a

ifdef SOC

LIBS +**=** cpu/$(CPU)/$(SOC)/lib$(SOC).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 +**=** rtc/librtc.a

LIBS +**=** drivers/libdrivers.a

LIBS +**=** drivers/nand/libnand.a

LIBS +**=** drivers/nand\_legacy/libnand\_legacy.a

LIBS +**=** drivers/usb/libusb.a

LIBS +**=** common/libcommon.a

LIBS +**=** $(BOARDLIBS)

LIBS **:=** $(addprefix $(obj),$(LIBS))

.PHONY **:** $(LIBS)

#编译的时候增加上述静态库，如果make的时候有O=xxx选项，增加$(obj)前缀

PLATFORM\_LIBS +**=** -L $(shell dirname `$(CC) $(CFLAGS) -print-libgcc-file-name`) -lgcc

# 获取arm-linux-gcc libgcc的安装路径

# 其中CC在底层的config.mk中有定义

SUBDIRS **=** tools \

examples \

post \

post/cpu

.PHONY **:** $(SUBDIRS)

#定义一些子目录

ifeq ($(CONFIG\_NAND\_U\_BOOT),y)

NAND\_SPL **=** nand\_spl

U\_BOOT\_NAND **=** $(obj)u-boot-nand.bin

endif

#如果$(obj)/include/config.mk中定义了CONFIG\_NAND\_U\_BOOT =y,则需要单独编译nand\_spl模块

\_\_OBJS **:=** $(subst $(obj),,$(OBJS))

\_\_LIBS **:=** $(subst $(obj),,$(LIBS))

#subst表示将$(OBJS)和$(LIBS)中的 $(obj)替换成空

### 2.2 rules.mk脚本分析

rules.mk用于生成依赖关系，make depend会依赖该脚本。

相关代码和分析如下：

\_depend**:** **$(obj).**depend

**$(obj).**depend**:** **$(src)**Makefile **$(TOPDIR)/**config.mk **$(SRCS)**

**@**rm -f **$@**

**@**for f in **$(SRCS);** **do** **\**

g**=`basename $$f | sed -e 's/\(.\*\)\.\w/\1.o/'`;** **\**

**$(CC)** -M **$(HOST\_CFLAGS)** **$(CPPFLAGS)** **-**MQ **$(obj)$$g** **$$f** **>>** **$@** **;** **\**

**done**

#.\*表示任意多个字符；

# \(.\*\)是用来做后向引用的，\1就是它了；

# \.是一个点

# \w用于匹配字母，数字或下划线字符；

# 作用：把文件扩展名的首个字符替换成o

# 比如有一个文件叫做test.log，文件名会被替换成test.oog

# g=`basename $$f | sed -e 's/(.\*).w/1.o/'`; 表示将源文件(\*.c,\*.S)转换为\*.o

# $(CC) -M $(HOST\_CFLAGS) $(CPPFLAGS) -MQ $(obj)$$g $$f >> $@，-M表示生成依赖关系，-MQ表示生成依赖列表的文件格式，后面的$(obj)$$g，即为格式，显示绝对路径

make depend的作用

C源码的开头经常有一系列被包含的头文件，例如 stdio.h。有一些是系统级的头文件，有一些是你正在写的项目的头文件：

#include <stdio.h>#include "foo.h"int main(....

要确定在你的 foo.h 被改变之后，这个文件也会被重新编译，就要在你的 Makefile 这样写：foo: foo.c foo.h

当你的项目变得越来越大，你自己的头文件越来越多的时候，要追踪所有这些头文件和所有依赖它的文件会是一件痛苦的事情。如果你改变了其中一个头文件，却忘了重新编译所有依赖它的源文件，结果会是很失望的。

这就是depend 的作用，如果不使用这个功能， 则当你修改一个头文件时， 必须重新把所有的源文件都编译一次， 使用depend ,就可只编译包含此头文件的源文件，可大大节省时间。

gcc 有一个选项可以分析你的源文件然后产生一个头文件的列表和它的依赖关系： -MM。

如果你把下面的内容加到你的 Makefile 里面：

depend:    gcc -E -MM \*.c > .depend

然后运行 make depend，就会产生一个 .depend，里面包含了目标文件，C 文件和头文件的列表：foo.o: foo.c foo.h

如果你改变了 foo.h，下一次运行 make 的时候，所有依赖 foo.h 的文件就会被重新编译。每一次你增加一个头文件的时候，别忘了运行一次 make depend。

### 2.3 config.mk脚本分析

config.mk文件用于源文件的编译，将.c和.s文件生成.o文件

相关代码和分析如下，以底层文件的post目录编译为例

ifneq ($(OBJTREE),$(SRCTREE))

ifeq ($(CURDIR),$(SRCTREE))

dir **:=**

else

dir **:=** $(subst $(SRCTREE)/,,$(CURDIR))

endif

obj **:=** $(if $(dir),$(OBJTREE)/$(dir)/,$(OBJTREE)/)

src **:=** $(if $(dir),$(SRCTREE)/$(dir)/,$(SRCTREE)/)

$(shell mkdir -p $(obj))

else

obj **:=**

src **:=**

endif

# clean the slate ...

PLATFORM\_RELFLAGS **=**

PLATFORM\_CPPFLAGS **=**

PLATFORM\_LDFLAGS **=**

# 如果make的时候定义了O=xxx选项，那么$(OBJTREE)为./xxx $(SRCTREE)为顶层目录

# 现在满足$(OBJTREE)！=$(SRCTREE) 条件，如果在post目录下include该脚本，

# $(CURDIR)也不等于$(SRCTREE)，所以dir的值为post所在目录的路径去除掉顶层目录路径前缀

# 也就是post所在目录的相对于顶层目录的相对路径。也就是post.

# obj设置为xxx/post的绝对路径,src设置为post目录的绝对路径

ifdef ARCH

sinclude $(TOPDIR)/$(ARCH)\_config.mk # include architecture dependend rules

endif

ifdef CPU

sinclude $(TOPDIR)/cpu/$(CPU)/config.mk # include CPU specific rules

endif

ifdef SOC

sinclude $(TOPDIR)/cpu/$(CPU)/$(SOC)/config.mk # include SoC specific rules

endif

ifdef VENDOR

BOARDDIR **=** $(VENDOR)/$(BOARD)

else

BOARDDIR **=** $(BOARD)

endif

ifdef BOARD

sinclude $(TOPDIR)/board/$(BOARDDIR)/config.mk # include board specific rules

endif

# ARCH = arm

# CPU = arm920t

# BOARD = 100ask24x0

# SOC = s3c24x0

# 所以需要include如下文件

# arm\_config.mk arm920t/config.mk arm920t/s3c24x0/config.mk 100ask24x0.config.mk

# 但是并不会所有的config.mk都会存在，所以使用了sinclude选项，即使不存在也没关系

CONFIG\_SHELL **:=** *$(shell if [ -x "$$BASH" ]; then echo $$BASH; \*

else if [ -x /bin/bash ]; then echo /bin/bash; \

else echo sh; fi ; fi)

ifeq ($(HOSTOS)-$(HOSTARCH),darwin-ppc)

HOSTCC **=** cc

else

HOSTCC **=** gcc

endif

HOSTCFLAGS **=** -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer

HOSTSTRIP **=** strip

# CONFIG\_SHELL设置为bash

# HOSTCC设置为gcc

cc-option **=** $(shell if $(CC) $(CFLAGS) *$(1) -S -o /dev/null -xc /dev/null \*

> /dev/null 2>&1; then echo "$(1)"; else echo "$(2)"; fi ;)

# 其他地方会经常用call函数来调用它，测试编译选项

# 这条语句等效为

# if $(CC) $(CFLAGS) $(1) -S -o /dev/null -xc /dev/null > /dev/null 2>&1; then

# echo "$(1)";

# else

# echo "$(2)";

# fi

# 函数cc-option用于检查编译器CC是否支持某选项。将2个选项作为参数传递给cc-option函数，

# 该函数调用CC编译器检查参数1是否支持，若支持则函数返回参数1，否则返回参数2

#（因此CC编译器必须支持参数1或参数2，若两个都不支持则会编译出错）.

# 以像下面这样调用cc-option函数，并将支持的选项添加到FLAGS中：

# FLAGS +=$(call cc-option,option1,option2)

# 我们来做个实现，当执行如下命令

# arm-linux-gcc -tx -S -o /dev/null -xc /dev/null >/dev/null

# -tx为不存在的gcc选项，所以会以标准错误的方式输出

# arm-linux-gcc: unrecognized option `-tx'

# 后面的2>&1选项表示将错误输出重定向到标准输出中。所以在终端执行

# arm-linux-gcc -tx -S -o /dev/null -xc /dev/null >/dev/null 2>&1不会有任何输出，

# 因为标准输出又重定向到/dev/null中了,该做法是为了让cc-option不会有错误信息打印到终端

AS **=** $(CROSS\_COMPILE)as

LD **=** $(CROSS\_COMPILE)ld

CC **=** $(CROSS\_COMPILE)gcc

CPP **=** $(CC) -E

AR **=** $(CROSS\_COMPILE)ar

NM **=** $(CROSS\_COMPILE)nm

STRIP **=** $(CROSS\_COMPILE)strip

OBJCOPY **=** $(CROSS\_COMPILE)objcopy

OBJDUMP **=** $(CROSS\_COMPILE)objdump

RANLIB **=** $(CROSS\_COMPILE)RANLIB

#定义交叉编译相关工具

ifneq (,$(findstring s,$(MAKEFLAGS)))

ARFLAGS **=** cr

else

ARFLAGS **=** crv

endif

# 如果执行了make -s 选项，则MAKEFLAGS的值为s，则ARFLAGS =crv，

# ARFLAGS为为ar命令的参数

# ar命令是用来将若干 .o 文件打包成（静态）库文件。 按照习惯，库文件后缀名都是 .a

# crv 分别是 ar 命令的三个参数，cr的意思是创建指定.a文件（如果不存在），并将.o文件加入到这个.a文件中。

# v 的意思是让 ar 命令在执行的时候打印更多的提示信息。

# ar crv abc.a $(OBJ) 就是将 abc.o 打包到库文件 abc.a 中的意思。

# make -s的s选项表示slient,在编译的时候不会有太多的log打印

RELFLAGS**=** $(PLATFORM\_RELFLAGS)

DBGFLAGS**=** -g # -DDEBUG

OPTFLAGS**=** -Os #-fomit-frame-pointer

# 定义了一些编译时候的选项

ifndef LDSCRIPT

#LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds.debug

ifeq ($(CONFIG\_NAND\_U\_BOOT),y)

LDSCRIPT **:=** $(TOPDIR)/board/$(BOARDDIR)/u-boot-nand.lds

else

LDSCRIPT **:=** $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds

endif

endif

# 设置链接脚本

# LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds

OBJCFLAGS +**=** --gap-fill=0xff

# OBJCFLAGS为objcopy时使用的参数.

# --gap-fill=0xff 指明了用0xff来填充section之间的空隙。

gccincdir **:=** $(shell $(CC) -print-file-name=include)

# 获取arm-linux-gcc编译器所使用到的头文件路径

# 我的电脑为/opt/JZ2440\_TOOLS/gcc-3.4.5-glibc-2.3.6/bin/../lib/gcc/arm-linux/3.4.5/include

CPPFLAGS **:=** $(DBGFLAGS) $(OPTFLAGS) $(RELFLAGS) \

-D\_\_KERNEL\_\_ -DTEXT\_BASE=$(TEXT\_BASE) \

ifneq ($(OBJTREE),$(SRCTREE))

CPPFLAGS +**=** -I$(OBJTREE)/include2 -I$(OBJTREE)/include

endif

CPPFLAGS +**=** -I$(TOPDIR)/include

CPPFLAGS +**=** -fno-builtin -ffreestanding -nostdinc \

-isystem $(gccincdir) -pipe $(PLATFORM\_CPPFLAGS)

ifdef BUILD\_TAG

CFLAGS **:=** $(CPPFLAGS) -Wall -Wstrict-prototypes \

-DBUILD\_TAG='"$(BUILD\_TAG)"'

else

CFLAGS **:=** $(CPPFLAGS) -Wall -Wstrict-prototypes

endif

#设置gcc的C文件编译使用的选项

AFLAGS\_DEBUG **:=**

AFLAGS **:=** $(AFLAGS\_DEBUG) -D\_\_ASSEMBLY\_\_ $(CPPFLAGS)

#设置gcc的S文件编译使用的选项

LDFLAGS +**=** -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT\_BASE) $(PLATFORM\_LDFLAGS)

#设置链接时候的选项

ifndef REMOTE\_BUILD

%.s**:** %.S

$(CPP) $(AFLAGS) -o $@ $<

%.o**:** %.S

$(CC) $(AFLAGS) -c -o $@ $<

%.o**:** %.c

$(CC) $(CFLAGS) -c -o $@ $<

else

$(obj)%.s**:** %.S

$(CPP) $(AFLAGS) -o $@ $<

$(obj)%.o**:** %.S

$(CC) $(AFLAGS) -c -o $@ $<

$(obj)%.o**:** %.c

$(CC) $(CFLAGS) -c -o $@ $<

endif

#定义编译规则，当定义了O=xxx时，编译的.o文件存入到xxx目录中

### 2.4 正式编译流程

当执行make命令时，需要从makefile的all处开始分析

相关代码和分析如下

ALL **=** $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(U\_BOOT\_NAND)

all**:** $(ALL)

$(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**:** depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)

UNDEF\_SYM=`$(OBJDUMP) -x $(LIBS) |sed -n -e 's/.\*\(\_\_u\_boot\_cmd\_.\*\)/-u\1/p'|sort|uniq`;\

cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF\_SYM $(\_\_OBJS) \

--start-group $(\_\_LIBS) --end-group $(PLATFORM\_LIBS) \

-Map u-boot.map -o u-boot

从ALL中的描述可以看出依赖关系为：$(obj)u-boot.srec**🡪**$(obj)u-boot🡪 depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)

下面将分析这些依赖条件。

#### 2.4.1 depend依赖

depend依赖相关代码

depend dep**:**

for dir in $(SUBDIRS) ; do $(MAKE) -C $$dir \_depend ; done

# SUBDIRS的值为tools examples post post/cpu，

# 通过for循环，在$(SUBDIRS)目录下执行make \_depend.

在这里我们分析一下post目录下的make \_depend.

post目录下的makefile相关代码

SUBDIRS **=** cpu

LIB **=** libpost.a

AOBJS **=** cache\_8xx.o

COBJS **=** cache.o codec.o cpu.o dsp.o ether.o

COBJS +**=** i2c.o memory.o post.o rtc.o

COBJS +**=** spr.o sysmon.o tests.o uart.o

COBJS +**=** usb.o watchdog.o

include $(TOPDIR)/post/rules.mk

怎么没有找到\_depend字眼，原来还有一处include $(TOPDIR)/post/rules.mk

$(topdir)/post/rules.mk相关代码分析

include $(TOPDIR)/config.mk

SRCS **:=** $(AOBJS:.o=.S) $(COBJS:.o=.c)

OBJS **:=** $(addprefix $(obj),$(AOBJS) $(COBJS))

LIB **:=** $(obj)$(LIB)

CPPFLAGS +**=** -I$(TOPDIR)

all**:** $(LIB)

$(LIB)**:** $(obj).depend $(OBJS)

$(AR) $(ARFLAGS) $@ $(OBJS)

include $(SRCTREE)/rules.mk

sinclude $(obj).depend

其中rules.mk中包含\_depend依赖，rules.mk文件内容见2.2节，rules.mk脚本会遍历SRCS下的文件，并在对应的目录生成依赖关系文件。依赖文件包含每个.o文件会依赖哪些头文件和源文件。

.depend文件内容如下

home/jay/Code/JZ2440/u-boot\_lite\_jz2440/obj/post/cache\_8xx.o**:** \

cache\_8xx.S \

/home/jay/Code/JZ2440/u-boot\_lite\_jz2440/obj/include/config.h \

/home/jay/Code/JZ2440/u-boot\_lite\_jz2440/include/configs/100ask24x0.h \

/home/jay/Code/JZ2440/u-boot\_lite\_jz2440/include/cmd\_confdefs.h

#### 2.4.2 version依赖

生成版本号文件，路径为

VERSION\_FILE **=** $(obj)include/version\_autogenerated.h

文件内容大致为：

#define U\_BOOT\_VERSION "U-Boot 1.1.6-gc667fa10-dirty"

version依赖相关代码

version**:**

@echo -n "#define U\_BOOT\_VERSION \"U-Boot " > $(VERSION\_FILE); \

echo -n "$(U\_BOOT\_VERSION)" >> $(VERSION\_FILE); \

echo -n $(shell $(CONFIG\_SHELL) *$(TOPDIR)/tools/setlocalversion \*

$(TOPDIR)) >> $(VERSION\_FILE); \

echo "\"" >> $(VERSION\_FILE)

#### 2.4.3 SUBDIRS依赖

SUBDIRS依赖会对相对应目录下的源码进行编译。

subdirs依赖相关代码

SUBDIRS **=** tools \

examples \

post \

post/cpu

.PHONY **:** $(SUBDIRS)

$(SUBDIRS)**:**

$(MAKE) -C $@ all

还是以post目录为例，下面继续贴出post目录makefile相关代码

post目录下的makefile相关代码

SUBDIRS **=** cpu

LIB **=** libpost.a

AOBJS **=** cache\_8xx.o

COBJS **=** cache.o codec.o cpu.o dsp.o ether.o

COBJS +**=** i2c.o memory.o post.o rtc.o

COBJS +**=** spr.o sysmon.o tests.o uart.o

COBJS +**=** usb.o watchdog.o

include $(TOPDIR)/post/rules.mk

$(topdir)/post/rules.mk相关代码分析

include $(TOPDIR)/config.mk

SRCS **:=** $(AOBJS:.o=.S) $(COBJS:.o=.c)

OBJS **:=** $(addprefix $(obj),$(AOBJS) $(COBJS))

LIB **:=** $(obj)$(LIB)

CPPFLAGS +**=** -I$(TOPDIR)

all**:** $(LIB)

$(LIB)**:** $(obj).depend $(OBJS)

$(AR) $(ARFLAGS) $@ $(OBJS)

include $(SRCTREE)/rules.mk

sinclude $(obj).depend

post目录编译的核心思想是利用2.3中介绍的config.mk脚本先将OBJS中指定的汇编和c源文件编译成.o文件，最终生成静态库。

config.mk生成.o文件的关键代码

ifndef REMOTE\_BUILD

**%.**s**:** **%.**S

**$(CPP)** **$(AFLAGS)** -o **$@** **$<**

**%.**o**:** **%.**S

**$(CC)** **$(AFLAGS)** -c -o **$@** **$<**

**%.**o**:** **%.**c

**$(CC)** **$(CFLAGS)** -c -o **$@** **$<**

**else**

**$(obj)%.**s**:** **%.**S

**$(CPP)** **$(AFLAGS)** -o **$@** **$<**

**$(obj)%.**o**:** **%.**S

**$(CC)** **$(AFLAGS)** -c -o **$@** **$<**

**$(obj)%.**o**:** **%.**c

**$(CC)** **$(CFLAGS)** -c -o **$@** **$<**

endif

#### 2.4.4 OBJS依赖

将cpu/arm920t cpu/arm920t/start.S编译成.o文件

objs依赖相关代码

OBJS **=** cpu/$(CPU)/start.o

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

$(OBJS)**:**

echo $(OBJS)

$(MAKE) -C cpu/$(CPU) $(if $(REMOTE\_BUILD),$@,$(notdir $@))

#上述语句等效为make -C cpu/arm920t obj/cpu/arm920t/start.o,

在cpu/arm920t的makefile中，可以找到对应的依赖START

cpu/arm920t makefile相关代码

include $(TOPDIR)/config.mk

LIB **=** $(obj)lib$(CPU).a

START **=** start.o

COBJS **=** cpu.o interrupts.o

SRCS **:=** $(START:.o=.S) $(SOBJS:.o=.S) $(COBJS:.o=.c)

OBJS **:=** $(addprefix $(obj),$(COBJS) $(SOBJS))

START **:=** $(addprefix $(obj),$(START))

all**:** $(obj).depend $(START) $(LIB)

$(LIB)**:** $(OBJS)

$(AR) $(ARFLAGS) $@ $(OBJS)

#### 2.4.5 LIBS依赖

LIBS依赖用于生成对应的静态库。

LIBS **=** lib\_generic/libgeneric.a

LIBS +**=** board/$(BOARDDIR)/lib$(BOARD).a

LIBS +**=** cpu/$(CPU)/lib$(CPU).a

ifdef SOC

LIBS +**=** cpu/$(CPU)/$(SOC)/lib$(SOC).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 +**=** rtc/librtc.a

LIBS +**=** drivers/libdrivers.a

LIBS +**=** drivers/nand/libnand.a

LIBS +**=** drivers/nand\_legacy/libnand\_legacy.a

LIBS +**=** drivers/usb/libusb.a

LIBS +**=** common/libcommon.a

LIBS +**=** $(BOARDLIBS)

LIBS **:=** $(addprefix $(obj),$(LIBS))

.PHONY **:** $(LIBS)

$(LIBS)**:**

$(MAKE) -C $(dir $(subst $(obj),,$@))

# $(MAKE) -C $(dir $(subst $(obj),,$@))取得目录地址，例如lib\_generic等

# lib的编译和上面的subdir编译流程是一样的

#### 2.4.5 LDSCRIPT依赖

生成uboot elf文件需要依赖lds文件

LDSCRIPT **:=** $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds

#### 2.4.6 生成uboot elf文件

生成uboot elf文件相关代码

$(obj)u-boot**:** depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)

UNDEF\_SYM=`$(OBJDUMP) -x $(LIBS) |sed -n -e 's/.\*\(\_\_u\_boot\_cmd\_.\*\)/-u\1/p'|sort|uniq`;\

cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF\_SYM $(\_\_OBJS) \

--start-group $(\_\_LIBS) --end-group $(PLATFORM\_LIBS) \

-Map u-boot.map -o u-boot

# UNDEF\_SYM:这个变量实际上就是把中间这一群的静态库里面\_\_u\_boot\_cmd\_开头的symbol都提取出来，

# 并在每个的开头加上了-u，以备后用。用处是Force symbol to be entered in the output file as an undefined symbol。

# arm-linux-objdump -x 显示所有可用的头信息，包括符号表、重定位入口

# LNDIR := $(OBJTREE)

# $(LDFLAGS)为LDFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT\_BASE) $(PLATFORM\_LDFLAGS)

# TEXT\_BASE = 0x33F80000

# \_\_OBJS = cpu/$(CPU)/start.o

# LDSCRIPT为$(TOPDIR)/board/$(BOARDDIR)/u-boot.lds

# --start-group --end-group 之间列出了所有的archives. 同一般的直接写archive不同在于，当search了一遍仍然有undefined reference的时候，会继续搜索。

# -Map  link map

其中，u-boot.lds文件分析如下：

OUTPUT\_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")

/\*OUTPUT\_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")\*/

OUTPUT\_ARCH(arm)

ENTRY(\_start)

SECTIONS

{

. = 0x00000000;

. = ALIGN(4);

.text :

{

cpu/arm920t/start.o (.text)

board/100ask24x0/boot\_init.o (.text)

\*(.text)

}

. = ALIGN(4);

.rodata : { \*(.rodata) }

. = ALIGN(4);

.data : { \*(.data) }

. = ALIGN(4);

.got : { \*(.got) }

. = .;

\_\_u\_boot\_cmd\_start = .;

.u\_boot\_cmd : { \*(.u\_boot\_cmd) }

\_\_u\_boot\_cmd\_end = .;

. = ALIGN(4);

\_\_bss\_start = .;

.bss : { \*(.bss) }

\_end = .;

}

uboot中有一个特殊的段（.u\_boot\_cmd），用于存储uboot中可以使用的命令。

\_\_u\_boot\_cmd\_start为该段的第一个符号；

\_\_u\_boot\_cmd\_end为该段的最后一个符号。

如果要把一个变量放入.u\_boot\_cmd段，需要通过宏U\_BOOOT\_CMD实现。

#define U\_BOOT\_CMD(name,maxargs,rep,cmd,usage,help) \

cmd\_tbl\_t \_\_u\_boot\_cmd\_##name Struct\_Section = {#name, maxargs, rep, cmd, usage, help}

例如定义bootm命令

U\_BOOT\_CMD**(**bootm**,** 16**,** 1**,** do\_bootm**,** “bootm**--**usage”**,** “help”**)**

展开后为：

cmd\_tbl\_t  \_\_u\_boot\_cmd\_ bootm  Struct\_Section **={**“bootm”**,** 16**,** 1**,** do\_bootm**,** “bootm**--**usage”**}**

如果把Struct\_Section也展开，则为：

cmd\_tbl\_t  \_\_u\_boot\_cmd\_ bootm  \_\_attribute\_\_ **((**unused**,**section **(**".u\_boot\_cmd"**)))** **=** **{**“bootm”**,** 16**,** 1**,** do\_bootm**,** “bootm**--**usage”**}**

\_\_attribute\_\_((section(“bar”)))命令用于把某个符号放到特定段中，因此在这里符号\_\_u\_boot\_cmd\_ bootm会被放入段.u\_boot\_cmd中。

uboot中所有命令都是存放在这个特殊段中的。当用户输入一个命令时（需要与定义时的name变量同名），会在函数find\_cmd中从\_\_u\_boot\_cmd\_start开始循环匹配，匹配到后就执行对应变量中的cmd项；如果直到\_\_u\_boot\_cmd\_end也没有发现，则匹配失败。

#### 2.4.7 生成uboot bin文件

elf文件是不能执行的，只能用于调试，所以最后需使用objcopy转成二进制可执行文件。

$(obj)u-boot.bin**:** $(obj)u-boot

$(OBJCOPY) ${OBJCFLAGS} -O binary $< $@