北京邮电大学嵌入式系统协会
E-mail: bupt.embedded.system@gmail.com

U-boot的Makefile分析

ccliu

2008-11-2

u-boot的Makefile分析

U－BOOT是一个LINUX下的工程，在编译之前必须已经安装对应体系结构的交叉编译环境，这里只针对ARM，编译器系列软件为 ppc_6xx- 。U－BOOT的下载地址：http://sourceforge.net/projects/u-boot
我下载的是1.1.5版本

要了解一个LINUX工程的结构必须看懂Makefile，尤其是顶层的，没办法，UNIX世界就是这么无奈，什么东西都用文档去管理、配置。首先在这方面我是个新手，时间所限只粗浅地看了一些Makefile规则,请各位多多指教。以cpci5200为例:

几个重要的文件分析：

u-boot-1.1.5/Makefile

u-boot-1.1.5/mkconfig

u-boot-1.1.5/config.mk

u-boot根目录下的Makefile文件（*u-boot-1.1.5/Makefile*）它负责配置u-boot的编译方式，具体说来包括：使用何种指令集，需包含哪些接口驱动、库等。Makefile的内容从上到下分别是：分定义编译环境：使用何种编译器、编译方式、目标文件的生成及它们最终镜像中的链接次序等。Mkconfig和config.mk在接下来的分析中会涉及到。

在编译U－BOOT之前，先要执行

```
# make cpci5200_config
```

cpci5200_config 是 Makefile 的一个目标,定义如下：

```
cpci5200_config:    unconfig
    @$(MKCONFIG) -a cpci5200  ppc mpc5xxx cpci5200 esd
```

其中，unconfig 定义如下：

```
unconfig:
    @rm -f $(obj)include/config.h $(obj)include/config.mk \
        $(obj)board/*/config.tmp $(obj)board/*/*/config.tmp
```

显然，执行# make cpci5200_config 时，先执行 unconfig 目标，注意不指定输出目标时，obj，src 变量均为空，unconfig 下面的命令清理上一 次执行 make *_config 时生成的头文件和 makefile 的包含文件。主要是 include/config.h 和 include/config.mk 文件。

然后才执行命令

```
@$(MKCONFIG) -a cpci5200  ppc mpc5xxx cpci5200 esd
```

$(MKCONFIG)就会被替换成$(SRCTREE)/mkconfig 文件路径， MKCONFIG 是顶层目录下的 mkcofig 脚本文件（*u-boot-1.1.5/mkconfig*），后面五个是传入的参数。Mkconfig 文件的详细分析见文档末尾。

注意一下 u-boot 对板卡的分类方法：
Target：宿主机平台
Architecture：定义芯片架构（如 MIPS、POWERPC、ARM 等）
CPU：定义芯片指令集版本（如 ARM7、ARM9、ARM11 等）
Board：芯片厂商，它细分为两类
[VENDOR]：按厂商划分（如 AT9200、S3C44B0 等）
[SOC]：按 SOC 类型（如 S3C2440、S3C2410 等）

1)对于 cpci5200_config 而言，mkconfig 主要做三件事：

在 include 文件夹下建立相应的文件（夹）软(符号)连接，

```
#如果是 PPC 体系将执行以下操作：
#ln -s      asm-ppc          asm
#ln -s arch-mpc5xxx     asm-ppc/arch
#ln -s    proc-armv        asm-ppc/proc
```

生成 Makefile 包含文件 include/config.mk，内容很简单，定义了四个变量：

```
ARCH    = ppc
CPU     = mpc5xxx
BOARD = cpci5200
VENDOR     = esd
```

这些变量可以供其它的 makefile 使用，作为一个基本配置.

生成 include/config.h 头文件，只有一行：

```
echo “/* Automatically generated - do not edit */” >>config.h
echo “#include <config/$1.h>” >>config.h
```

这两行代码生成一个 include/config.h 文件，这个文件很简单，只有一句话：
#include <$1.h> 当然这里的$1 时要被替换成不同的 board 名字的.这个取决于我们在主 Makefile 中 xxxx_config 目标中的 xxxx 是什么.

mkconfig 脚本文件的执行至此结束

2)分析 config.mk 的内容：

u-boot 根目录下自带一个 config.mk 文件（*u-boot-1.1.5/config.mk*），应 该说这才是真正的 Makefile，以上介绍的两个脚本 Makefile 和 mkconfig 完成了环境配置之后，在该文件中才定义具体的编译规则，所以你会发现在各个子模块（board、 cpu、lib_xxx、net、disk...）目录中的 Makefile 第一句就是：include $(TOPDIR)/config.mk。文件内容分析如下：

这个文件的功能就是给各个在编译过程中的变量赋值,包括编译执行的函数与编译的时候所带的参数等等。还会根据是否定义了 ARCH，CPU，SOC，VENDOR，BOARD 来决定是否包含相应位置的 config.mk 文件，这些个文件里，也是定义了相应的平台在编译的时候应该加的参数。所以如果你为你自己的开发板取了别的名字了，那么就要检查一下这个文件，看一下相应的位置上的 config.mk 文件有没有，内容是否为你要想的。

- 包含体系，开发板，CPU 特定的规则文件：

```
#指定预编译体系结构选项
ifdef  ARCH
sinclude $(TOPDIR)/$(ARCH)_config.mk  # include architecture dependend rules
endif
#定义编译时对齐，浮点等选项
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
#指定特定板子的镜像连接时的内存基地址，重要！
ifdef  BOARD
sinclude $(TOPDIR)/board/$(BOARDDIR)/config.mk   # include board specific rules
endif
```

- 定义交叉编译链工具

```
# Include the make variables (CC, etc...)
#
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 AR 选项 ARFLAGS，调试选项 DBGFLAGS，优化选项 OPTFLAGS　预处理选项 CPPFLAGS，C 编译器选项 CFLAGS，连接选项 LDFLAGS

```
#制定了在编译的时候告诉编译器生成的代码的基地址是TEXT_BASE
CPPFLAGS := $(DBGFLAGS) $(OPTFLAGS) $(RELFLAGS)      \
    -D__KERNEL__ -DTEXT_BASE=$(TEXT_BASE)      \

AFLAGS := $(AFLAGS_DEBUG) -D__ASSEMBLY__ $(CPPFLAGS)

#指定了起始地址TEXT_BASE
LDFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT_BASE) $(PLATFORM_LDFLAGS)
```

- 指定编译规则

```
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
```

3）分析最关键的 u-boot ELF 文件镜像的生成

Makefile 文件中的各个依赖目标分析如下：

- 依赖目标 depend ：生成各个子目录的.depend 文件，.depend 列出每个目标文件的依赖文件。生成方法，调用每个子目录的 make _depend。

```
depend dep:
   for dir in $(SUBDIRS) ; do $(MAKE) -C $$dir _depend ; done
```

- 依赖目标 version：生成版本信息到版本文件 VERSION_FILE 中。

```
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)
```

- 伪目标 SUBDIRS：执行 tools ,examples ,post,post\cpu 子目录下面的 make 文件。

```
SUBDIRS = tools \
   examples \
   post \
   post/cpu
.PHONY : $(SUBDIRS)

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

- 依赖目标$(OBJS)，即 cpu/start.o

```
$(OBJS):
   $(MAKE) -C cpu/$(CPU) $(if $(REMOTE_BUILD),$@,$(notdir $@))
```

- 依赖目标$(LIBS)，这个目标太多，都是每个子目录的库文件*.a ，通过执行相应子目录下的 make 来完成：

```
$(LIBS):
   $(MAKE) -C $(dir $(subst $(obj),,$@))
```

- 依赖目标$(LDSCRIPT)：

```
LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/u-boot.lds
LDFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT_BASE) $(PLATFORM_LDFLAGS)
```

对于每一种开发板来说,LDSCRIPT 即连接脚本文件，例如 board/esd/cpci5200/u-boot.lds，定义了连接时各个目标文件是如何组织的。内容如下：

```
OUTPUT_ARCH(powerpc)
SEARCH_DIR(/lib); SEARCH_DIR(/usr/lib); SEARCH_DIR(/usr/local/lib);
SEARCH_DIR(/usr/local/powerpc-any-elf/lib);
/* Do we need any of these for elf?
   __DYNAMIC = 0;    */
SECTIONS
```

```
{
  /* Read-only sections, merged into text segment: */
  . = + SIZEOF_HEADERS;
  .interp : { *(.interp) }
  .hash           : { *(.hash)      }
  .dynsym         : { *(.dynsym)    }
  .dynstr         : { *(.dynstr)    }
  .rel.text       : { *(.rel.text)      }
  .rela.text      : { *(.rela.text)     }
  .rel.data       : { *(.rel.data)      }
  .rela.data      : { *(.rela.data)     }
  .rel.rodata     : { *(.rel.rodata)    }
  .rela.rodata    : { *(.rela.rodata)   }
  .rel.got        : { *(.rel.got)       }
  .rela.got       : { *(.rela.got)      }
  .rel.ctors      : { *(.rel.ctors) }
  .rela.ctors     : { *(.rela.ctors)    }
  .rel.dtors      : { *(.rel.dtors) }
  .rela.dtors     : { *(.rela.dtors)    }
  .rel.bss        : { *(.rel.bss)       }
  .rela.bss       : { *(.rela.bss)      }
  .rel.plt        : { *(.rel.plt)       }
  .rela.plt       : { *(.rela.plt)      }
  .init           : { *(.init)   }
  .plt : { *(.plt) }
#.text的基地址由LDFLAGS中-Ttext $(TEXT_BASE)指定
  .text      :
  {
#start.o为首
    cpu/mpc5xxx/start.o (.text)
    *(.text)
    *(.fixup)
    *(.got1)
    . = ALIGN(16);
    *(.rodata)
    *(.rodata1)
    *(.rodata.str1.4)
    *(.eh_frame)
  }
  .fini      : { *(.fini)    } =0
  .ctors     : { *(.ctors)   }
  .dtors     : { *(.dtors)   }
```

```
/* Read-write section, merged into data segment: */
. = (. + 0x0FFF) & 0xFFFFF000;
_erotext = .;
PROVIDE (erotext = .);
.reloc   :
{
  *(.got)
  _GOT2_TABLE_ = .;
  *(.got2)
  _FIXUP_TABLE_ = .;
  *(.fixup)
}
__got2_entries = (_FIXUP_TABLE_ - _GOT2_TABLE_) >> 2;
__fixup_entries = (. - _FIXUP_TABLE_) >> 2;

.data    :
{
  *(.data)
  *(.data1)
  *(.sdata)
  *(.sdata2)
  *(.dynamic)
  CONSTRUCTORS
}
_edata  =  .;
PROVIDE (edata = .);

. = .;
__u_boot_cmd_start = .;
.u_boot_cmd : { *(.u_boot_cmd) }
__u_boot_cmd_end = .;


. = .;
__start___ex_table = .;
__ex_table : { *(__ex_table) }
__stop___ex_table = .;

. = ALIGN(4096);
__init_begin = .;
.text.init : { *(.text.init) }
.data.init : { *(.data.init) }
. = ALIGN(4096);
```

```
  __init_end = .;

  __bss_start = .;
  .bss       :
  {
   *(.sbss) *(.scommon)
   *(.dynbss)
   *(.bss)
   *(COMMON)
  }
  _end = . ;
  PROVIDE (end = .);
}
```

整个 makefile 剩下的内容全部是各种不同的开发板的*_config:目标的定义了。

对于各子目录的 makefile 文件，主要是生成*.o 文件然后执行 AR 生成对应的库文件。

概括起来，工程的编译流程也就 是通过执行执行一个 make *_config 传入 ARCH，CPU，BOARD，SOC 参数，mkconfig 根据参数将 include 头文件夹相应的头文件夹连接好，生成 config.h。然后执行 make 分别调用各子目录的 makefile 生成所有的 obj 文件和 obj 库文件 *.a. 最后连接所有目标文件，生成镜像。不同格式的镜像都是调用相应工具由 elf 镜像直接或者间接生成的。

Mkconfig 源码注解

```
#下面这一行的内容，表示这个shell脚本的解释器是/bin/sh，给的解释器的参数为-e，这个
#参数的意思就是，当shell返回值为非零值的时候，shell马上退出执行。在shell脚本里也
#可以没有这一行，这一行不是必须的，如果没有这一行的话，那么shell脚本就会用当前运
#行环境下的默认的shell来执行。
#!/bin/sh -e

# Script to create header files and links to configure
# U-Boot for a specific board.
#
# Parameters:    Target    Architecture    CPU    Board [VENDOR] [SOC]
#
# (C) 2002-2006 DENX Software Engineering, Wolfgang Denk <wd@denx.de>
#

#APPEND变量与BOARD_NAME变量都设置了默认值，如果后面对变量值有修改就以后面的为准，
#没有就是默认值了。这里的APPEND的参数的意义，实际上就是用来标识是否产生一个新的配
```

```
#置文件，还是直接把生成的配置信息写到旧文件后面。
APPEND=no       # Default: Create new config file
BOARD_NAME=""  # Name to print in make output

#$#代表的是传入脚本的参数的个数，-gt表示是如果左边参数比右边参数大，则返回true，
#否则为false。$1代表的是传入的第一个参数的内容，$2表示传入的第二个参数的内容，以
#此类推。shift表示参数都左移一位，原来的$3变为$2，$2为$1，$1的内容则被丢弃。这个
#地方，实际上是处理了一些附加的参数的问题，如果是--，则仅丢弃不做其它处理；如果是
#-a，则把APPEND的值置为yes；如果是-n，则表示后面跟的是板子的名字。${a%%pattern}是
#shell中的一个替换语法，表示把变量a中的内容，从右至左，最大程度上把符合pattern样
#式的字符串删掉。这里可以看出，如果-n后面跟的是类似于smdk2410_config的参数的话，
#则最后会变成smdk2410；如果是其它的值，则退出循环，不执行了。在Makefile中的动作，
#其实并不会触发这里的处理逻辑，估计这里是作者为了调试方便，需要单独运行此脚本的时
#候，加的一些对参数的处理。
while [ $# -gt 0 ] ; do
    case "$1" in
    --) shift ; break ;;
    -a) shift ; APPEND=yes ;;
    -n) shift ; BOARD_NAME="${1%%_config}" ; shift ;;
    *)    break ;;
    esac
done

#第一行的语法表示如果BOARD_NAME没有被设置过，则把它置为变量1的值。然后进行参数个
#数的判断，-lt表示less than则返回true，也就是如果参数少于4个或是参数大于6个，则退
#出，不执行。如果没有退出执行的话，则输出Configuring for ${BOARD_NAME} board...这
#句话，其中${BOARD_NAME}会被替换成用户输入的板子的名字。
[ "${BOARD_NAME}" ] || BOARD_NAME="$1"

[ $# -lt 4 ] && exit 1
[ $# -gt 6 ] && exit 1

echo "Configuring for ${BOARD_NAME} board..."

#
# Create link to architecture specific headers
#
#当用户指定的$OBJTREE与$SRCTREE不一致的时候会做如下一些事情：
#在$OBJTREE下建立include文件夹
#在$OBJTREE下建立include2文件夹
#进入到include2文件夹，其实就是$OBJTREE文件夹下的include2
#删除asm，其实往后看就知道了，这是一个文件夹，是以link的方式建立的
#然后建立一个asm的文件夹，这个文件夹是指向${SRCTREE}/include/asm-$2的，$2这个参数
#多指CPU架构类型如：PPC,ARM。
#给变量LNPREFIX赋值。这个变量在以后的执行会用到，所以这里给的asm的位置，是相对于
#后来执行的时候，当前工作目录与asm之间的关系来定的。
```

```
#进入到$OBJTREE中的include文件夹（之前是在include2里）。
#删除掉asm-$2文件夹，其实就是asm-arm文件夹。
#删除掉asm文件夹
#建立一个新的asm-$2文件夹，其实就是asm-arm文件夹
#建立一个名为asm的link，这个link指向新建立的asm-arm文件夹。
#现在看一下整个目录大概的结构：
${OBJTREE}
Include
  asm-arm
  asm -> ./asm-arm
include2
  asm -> ${SRCTREE}/include/asm-arm
#可以看到，目前在include下的asm-arm与asm其实是同一个文件夹，并且内容为空，include2
#文件夹下的asm是指向了源码树中的${SRCTREE}/include/asm-arm文件夹。
#如果"$SRCTREE"与"$OBJTREE"是同一个文件（大多数情况下，咱们都是这种方式来编译的），
#那么就是仅仅在include文件夹下，建立一个名为asm的link，直接指向asm-$2，即asm-arm。
#最后删除asm-arm/arch文件夹。
if [ "$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

rm -f asm-$2/arch

#下面第一句中连接两个判断的-o，相当于or的意思，就是表示的“或”。意思就是如果第6
#个参数为空或是为NULL，则在include下建立一个asm-$2/arch的link，指向
#${LNPREFIX}arch-$3，否则就在include下建立一个asm-$2/arch的link，指向
#${LNPREFIX}arch-$6
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
```

```
#如果第二个参数是arm的话，则删除include/asm-$2/proc文件，实际上就是
#include/asm-arm/proc文件夹，建立一个新的link，在include/asm-$2/proc处，也即
#include/asm-arm/proc，指向${LNPREFIX}proc-armv文件夹
if [ "$2" = "arm" ] ; then
    rm -f asm-$2/proc
    ln -s ${LNPREFIX}proc-armv asm-$2/proc
fi

#
# Create include file for Make
#
#第一行，输出ARCH = $2，即ARCH = arm到config.mk，也即include/config.mk，注意这里
#用了一个>，它表示重新生成一个config.mk文件，如果有旧的，则覆盖
#第二行，同理，输出CPU = arm920t到config.mk中
#第三行，同理，输出BOARD = smdk2410到config.mk
#第四行，判断，如果$5不为空，且$5的值不为NULL，
#则把VENDOR = $5的值输出到config.mk中
#第五行，同理，把SOC输出到config.mk中
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

#
# Create board specific header file
#
#如果APPEND的值为yes，则写入一空行到已经存在的config.h中，也即include/config.h中，
#如果不是，则生成一个新的config.h，如果旧的文件存在，则覆盖。
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
```