打补丁：

Patch 打到的位置 < 补丁的文件名

比如下面的补丁

--- u-boot-1.1.6/board/100ask24x0/100ask24x0.c 1970-01-01 08:00:00.000000000 +0800

+++ u-boot-1.1.6\_jz2440\_20171103/board/100ask24x0/100ask24x0.c 2017-11-03

当你进入u-boot-1.1.6目录下 使用patch –p1 < ../xxxx.patch /\* –p1表示忽略第一个/之前的所有内容\*/

一般打补丁的步骤：

1. 解压
2. 打补丁
3. 配置 （让uboot其支持很多的单板）在这次学习板中使用make 100ask24x0\_config
4. 编译 make

将生成的u-boot.b可以烧写进板子中去，这样就能通过usb下载文件

启动后在倒计时前空格进入命令行：

q推出后Help 命令可以看有什莫命令 menu 回到菜单

print 可以查看环境变量：

例如将bootdelay设置为5

Set bootdelay 5

save

bootloader的作用启动内核：首先内核一般放在flash中，从flash读出放在sdram中。启动内核

那么uboot要实现启动内核，他因该有如下的功能：

启动内核(核心功能)

读flash + 写flash

初始化sdram

例如还有初始化时钟（起来默认时12M但是太慢2440最大可以400m）

初始化 串口 网卡 usb

分析配置：(基于u-boot-1.1.6\_jz2440.patch)

1.执行 mke 100ask24x0\_config

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

//MKCONFIG

3.MKCONFIG := $(SRCTREE)/mkconfig /\*原目标树下有mkconfig这个文件\*/

$(@ 是目标的意思也就是100ask24x0\_config

所以：2 等效于

4.mkconfig 100ask24x0\_config arm arm920t 100ask24x0 NULL s3c24x0

在mkconfig文件中具体分析4：

判断参数是否有 -- -a -n

while [ $# -gt 0 ] ; do

case "$1" in

--) shift ; break ;;

-a) shift ; APPEND=yes ;;

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

\*) break ;;

esac

done

mkconfig 100ask24x0\_config arm arm920t 100ask24x0 NULL s3c24x0

$0 $1 $2 $3 $4 $5 $6

二：

[ "${BOARD\_NAME}" ] || BOARD\_NAME="$1" /\*有定义BOARD\_NAME 就不执行后者，在Linux shell中 $0 $1 $2 代表参数\*/

三：

[ $# -lt 4 ] && exit 1

[ $# -gt 6 ] && exit 1 /\*参数个数判断\*/

四：

echo "Configuring for ${BOARD\_NAME} board..."/\*打印此句话\*/

五：

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 //执行else

cd ./include

rm -f asm

ln -s asm-$2 asm //建立软链接文件给asm-arm ln -s asm-arm asm 在包含头文件方便

fi

makefile 中可以看出 OBJTREE与 SRCTREE相等，因为没有定义BUILD\_DIR。

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

SRCTREE := $(CURDIR)

六：

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

七：

if [ -z "$6" -o "$6" = "NULL" ] ; then // $6 不等于空 或者NULL

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

else //执行else

ln -s ${LNPREFIX}arch-$6 asm-$2/arch // ln -s arch-s3c24x0 asm-arm/arch(没有定义LNPREFIX) asm-arm/arch为链接文件指向arch-s3c24x0

八：

if [ "$2" = "arm" ] ; then // 成立

rm -f asm-$2/proc

ln -s ${LNPREFIX}proc-armv asm-$2/proc // ln -s proc-armv asm-arm/proc 同样的链接文件

九：

echo "ARCH = $2" > config.mk // > 表示创建一个文件ARCH＝arm

echo "CPU = $3" >> config.mk // >> 表示追加 CPU =arm920t

echo "BOARD = $4" >> config.mk // BOARD =100ask240x

执行了上面的内容，相当与创建了一个config.mk文件他的内容是：

ARCH＝arm

CPU =arm920t

BOARD =100ask240x

十：

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

[ "$6" ] && [ "$6" != "NULL" ] && echo "SOC = $6" >> config.mk //第六个参数存在并且不为null SOC = s3c2440

十一：

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 //追加到config.h中

echo "#include <configs/$1.h>" >>config.h //追加同样的方式

/\*config的文件\*/

编译：

Make 假如不加参数的话意思就是要生成all

OBJS = cpu/$(CPU)/start.o //cpu/$(arm920t)/start.o

LIBS += board/$(BOARDDIR)/lib$(BOARD).a //board/100ask24x0/lib$(BOARD).a

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

假如不懂:

$(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

可以执行make之后查看：

cd /home/book/work/u-boot/u-boot-1.1.6 && arm-linux-ld -Bstatic -T /home

/book/work/u-boot/u-boot-1.1.6/board/100ask24x0/u-boot.lds -Ttext 0x33F80000 $U

NDEF\_SYM cpu/arm920t/start.o

--start-group lib\_generic/libgeneric.a board/100ask24x0/lib100as k24x0.a cpu/arm920t/libarm920t.a cpu/arm920t/s3c24x0/libs3c24x0.a lib\_arm/libarm

.a fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/libjffs2.a f

s/reiserfs/libreiserfs.a fs/ext2/libext2fs.a net/libnet.a disk/libdisk.a rtc/lib

rtc.a dtt/libdtt.a drivers/libdrivers.a drivers/nand/libnand.a drivers/nand\_lega

cy/libnand\_legacy.a drivers/usb/libusb.a drivers/sk98lin/libsk98lin.a common/lib

common.a --end-group \

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

一般的单片机实验：

初始化:1 关闭开门狗 2 初始化时钟 3 初始化sdram

如果程序太大的话 将程序从nand =》 sdram

设置 堆栈（sp）

Uboot 是高级的单片机程序

也同样是进行硬件相关的初始化

设置sp 也就是将sp指向一段内存 ，就可以进行调用c函数

从uboot 的makefile 来看首先是 cpu/arm920t/start.s

start.s:

第一阶段：

设置管理模式

关闭看门狗

屏蔽中断

初始化SDRAM

设置栈

时钟

重定位（将代码放入到RAM）

清除BSS段

调用 start.armboot（c的实现）

第二阶段：

网卡， usb

1 Uboot 的终究目标从flash 读出内核然后 ===》 读写 flash flash\_init nand\_init 初始化

2启动

start\_armboot函数

flash\_init

nand\_init

main\_loop

s= getenv(“bootcmd”)

环境变量 bootcmd 有两个作用从flash 读出内核 启动内核

Nand read.jffe2 0x30007FC0 kernel (分区) 从flash 读到 0x3000 地址上

Bootm 0x30007FC0 启动

Uboot 控制界面

Readline(读入串口数据)

Uboot 命令：

通过输入字符串 ---动作（也就是函数）

#define Struct\_Section \_\_attribute\_\_ ((unused,section (".u\_boot\_cmd")))

bootcmd=nand read.jffs2 0x30007FC0 kernel; bootm 0x30007FC0

U\_BOOT\_CMD(

bootm, CFG\_MAXARGS, 1, do\_bootm,

"bootm - boot application image from memory\n",

"[addr [arg ...]]\n - boot application image stored in memory\n"

"\tpassing arguments 'arg ...'; when booting a Linux kernel,\n"

"\t'arg' can be the address of an initrd image\n"

#ifdef CONFIG\_OF\_FLAT\_TREE

"\tWhen booting a Linux kernel which requires a flat device-tree\n"

"\ta third argument is required which is the address of the of the\n"

"\tdevice-tree blob. To boot that kernel without an initrd image,\n"

"\tuse a '-' for the second argument. If you do not pass a third\n"

"\ta bd\_info struct will be passed instead\n"

#endif

);

宏定义如下：

#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}

则上边的宏可以展开如下：

cmd\_tbl\_t \_\_u\_boot\_cmd\_ bootm \_\_attribute\_\_ ((unused,section (".u\_boot\_cmd"))) =

{ bootm CFG\_MAXARGS 1 do\_bootm usage help(后面两个分别对应的后面的字符串)

}

s = getenv ("bootcmd"); //获取环境变量

1. 读出内核

bootcmd=Nand read.jffe2 0x30007FC0 kernel //从kernel 分区读出内核 放到0x30007FC0地址

其实也就是这样 bootcmd=Nand read.jffe2 0x30007FC0 0x00060000 0x00200000

kernel是分区名字

pc上每个硬盘上都有分区表，但是在嵌入式系统中没有分区表。

所以我们想象构造分区表：

|  |  |  |  |
| --- | --- | --- | --- |
| Uboot | env | Kernel | rootfile |

在源码写死分区的地址

从flash 读到 0x3000 地址上

读nand 根据do\_nand函数 调用 nand\_erase\_opts

1. 启动内核

Bootm 0x30007FC0 启动 do\_bootm 读取头部 移动内核到合适的地址

Flash 存贮的是uimage:

包括 头和真正的uimage

typedef struct image\_header {

uint32\_t ih\_magic; /\* Image Header Magic Number \*/

uint32\_t ih\_hcrc; /\* Image Header CRC Checksum \*/

uint32\_t ih\_time; /\* Image Creation Timestamp \*/

uint32\_t ih\_size; /\* Image Data Size \*/

uint32\_t ih\_load; /\* Data Load Address \*/

uint32\_t ih\_ep; /\* Entry Point Address \*/

uint32\_t ih\_dcrc; /\* Image Data CRC Checksum \*/

uint8\_t ih\_os; /\* Operating System \*/

uint8\_t ih\_arch; /\* CPU architecture \*/

uint8\_t ih\_type; /\* Image Type \*/

uint8\_t ih\_comp; /\* Compression Type \*/

uint8\_t ih\_name[IH\_NMLEN]; /\* Image Name \*/

} image\_header\_t;

do\_bootm\_linux 启动内核

1. uboot告诉内核启动参数 ===》设置启动参数 在某个地址（0x3000100）按某种格式(TAG)保存数据

2 . 跳到入口函数处

static void setup\_start\_tag (bd\_t \*bd)

{

params = (struct tag \*) bd->bi\_boot\_params;

params->hdr.tag = ATAG\_CORE;

params->hdr.size = tag\_size (tag\_core);

params->u.core.flags = 0;

params->u.core.pagesize = 0;

params->u.core.rootdev = 0;

params = tag\_next (params);

}

static void setup\_memory\_tags (bd\_t \*bd)

{

int i;

for (i = 0; i < CONFIG\_NR\_DRAM\_BANKS; i++) {

params->hdr.tag = ATAG\_MEM;

params->hdr.size = tag\_size (tag\_mem32);

params->u.mem.start = bd->bi\_dram[i].start;

params->u.mem.size = bd->bi\_dram[i].size;

params = tag\_next (params);

}

}

setup\_commandline\_tag