弄清的问题uboot启动到内核启动的流程

1.1嵌入式系统启动流程

Boot🡪Booloader🡪linux 内核-🡪挂接根文件系统🡪应用程序

1.2 uboot打补丁

所谓补丁就对uboot新修改的部分。打补丁就是将补丁添加入uboot。

1.2.1如何打补丁？

$ tar xjf u-boot-1.1.6 解压官方uboot

$ cd u-boot-1.1.6/ 进入uboot目录

$ patch -p1 < ../uboot-1.1.6.jz2440.patch 打补丁

p1代表去掉补丁里面路径信息第一个/前面的部分

补丁文件里面的diff指示同一个文件更改前后之间的差异，也指示了补丁打到哪里去。

uboot 补丁文件：

diff -urN u-boot-1.1.6/board/100ask24x0/boot\_init.c u-boot-1.1.6\_jz2440/board/100ask24x0/boot\_init.c

--- u-boot-1.1.6/board/100ask24x0/boot\_init.c 1970-01-01 07:00:00.000000000 +0700

+++ u-boot-1.1.6\_jz2440/board/100ask24x0/boot\_init.c 2010-11-26 12:54:37.042080086 +0800

-34，6 +34 8

---表示修改前的文件，+++表示的是修改后的文件

diff显示着两个文件的差别

-34，6表示是原来的代码从34行开始共6行

+34， 8 表示修改后的代码也是从34行开始共8行

-代表原来的代码，需要去掉，+代表添加的代码

- ulong offset = 0;

+ ulong offset = 0;

1.3 uboot 编译

1、解压uboot

2、打补丁

3、配置 make 100ask24x0\_config

4、编译 make

1.4 uboot命令

启动uboot ,输入help得到uboot支持的命令，输入？cmd 就可以得到某个命令的使用方法， ？md就得到了md命令的使用方法。

1.5设置uboot 环境变量

命令行输入Print 打印所有的可设置环境变量

1、Set bootdealy 10

2、save 保存设置

3、reset 重新启动uboot

1.6 uboot 实现功能

1.5.1为启动内核实现功能

1、关看门狗

2、设置时钟

3、初始化sdram

4、从flash读出内核

5、启动内核

1.5.2为开发方便需要实现的功能

1、读写flash

2、网卡

3、Usb

4、串口通信

2.1 uboot 配置

配置命令：make smdk2410\_config

根目录下Makefile中执行命令

smdk2410\_config : unconfig

@$(MKCONFIG) $(@:\_config=) arm arm920t smdk2410 NULL s3c24x0

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

其中“@”的作用是执行该命令时不在shell显示

SRCTREE := $(CURDIR) //把当前源码所在目录 $(CURDIR) 赋给SRCTREE

MKCONFIG := $(SRCTREE)/mkconfig

//MKCONFIG指向源码所在目录(U-BOOT顶层目录)下的mkconfig配置文件

MKCONFIG等同于 ./mkconfig

$(@:\_config=)的作用就是将smdk2410\_config中的\_config去掉，得到smdk2410。

arm 表示CPU的构架是基于ARM体系的,arm920t 表示CPU的类型是arm920t,smdk2410 表示开发板的名字，NULL 表示开发商或者经销商的名称，这里为空，s3c24x0 表示基于S3C2410的片上系统。

$(MKCONFIG)在上面指定为“$(SRCTREE)/mkconfig”。$(@:\_config=)为将传进来的所有参数中的\_config替换为空（其中“@”**指规则的目标文件名**，在这里就是“smdk2410\_config ”。$(text:patternA=patternB)，这样的语法表示把text变量每一个元素中结尾的patternA的文本替换为patternB，然后输出）。因此$(@:\_config=)的作用就是将smdk2410\_config中的\_config去掉，得到smdk2410。

因此“@$(MKCONFIG) $(@:\_config=) smdk2410 arm arm920t smdk2410 samsung s3c24x0”实际上就是执行了如下命

./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0

2.2 执行./mkconfig 文件

mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0

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

BOARD\_NAME=""

[ "${BOARD\_NAME}" ] || BOARD\_NAME="$1" // BOARD\_NAME= smdk2410

/\*若将目标文件设定为输出到源文件所在目录，则以下代码在include目录下建立了到

asm-arm目录的符号链接asm。其中的ln -s asm-$2 asm即ln -s asm-arm asm

asm 指向asm-arm ,将源文件中的asm替换为asm-arm。为何这样做，比如源码#include<asm/type.h>就会翻译成#include<asm-arm/type.h>,才能找到正确的头文件\*/

cd ./include //下面目录都为include

rm -f asm

ln -s asm-$2 asm

/\*

rm -f asm-arm/arch

LNPREFIX没有定义，为空

ln -s ${LNPREFIX}arch-$6 asm-$2/arch相当于 ln –s arch-s3c24x0 asm-arm/arch

\*/

rm -f asm-$2/arch

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

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

rm -f asm-$2/proc //rm –f asm-arm/proc

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

/\* >创建文件config.mk,>>追加内容。Config.mk文件最终内容

ARCH =arm

CPU = arm-920t

BOARD = smdk2410

SOC = s3c24x0

\*/

echo "ARCH = $2" > config.mk

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

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

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

/\*

创建文件config.h内容如下

/\* Automatically generated - do not edit \*/

#include <configs/smdk2410.h>

\*/

> config.h # Create new config file

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

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

2.2 uboot 编译过程

include $(OBJTREE)/include/config.mk //mkconfig生成的文件

ifeq ($(ARCH),arm)

CROSS\_COMPILE = arm-linux-

OBJS = cpu/$(CPU)/start.o //CPU=arm920t 起始文件是cpu/arm920t/start.s

LIBS = lib\_generic/libgeneric.a

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

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

/\*

执行make命令就是执行make all

obj := $(OBJTREE)/ //义变量obj，让其等于目标存放目录

\*/

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

all: $(ALL)

$(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

执行make命令编译uboot，uboot执行编译命令如下

cd /work/tools/u-boot-1.1.6 && arm-linux-ld -Bstatic -T /work/tools/u-boot-1.1.6/board/100ask24x0/u-boot.lds -Ttext 0x33F80000 $UNDEF\_SYM cpu/arm920t/start.o \

--start-group lib\_generic/libgeneric.a board/100ask24x0/lib100ask24x0.a cpu/arm920t/libarm920t.a cpu/arm920t/s3c24x0/libs3c24x0.a lib\_arm/libarm.a…

以上命令知道链接地址是0x33f80000,也是Uboot的运行地址。链接脚本文件的路径：

/work/tools/u-boot-1.1.6/board/100ask24x0/u-boot.lds

链接脚本u-boot.lds分析

ENTRY(\_start)

SECTIONS

{

. = 0x00000000; //代码存储地址

. = 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 = .;

TEXT\_BASE在研发板相关的目录中的\board\smdk2410\config.mk文档中定义, 他定义了

代码在运行时所在的地址, TEXT\_BASE = 0x33F80000 因为内存只有64M空间，把uboot

放在了内存最上面的512k空间内。内核则是从内存起始空间0x30000000开始运行，内核

启动后uboot就不再起作用。

3.1 uboot源码流程分析

3.1.1 uboot第一阶段代码

1、设为SVC模式

2、关看门狗

3、屏蔽中断

4、时钟初始化

5、关闭cache和Mmu

6、初始化sdram

7、重定位，即拷贝uboot代码到sdram

8、设置栈

9、清bss段

10、跳到第二阶段代码

代码执行流程：

1、b reset //设为SVC模式，关看门狗，屏蔽中断，初始化时钟

2、bl cpu\_init\_crit //关闭cahe和Mmu

3、bl lowlevel\_init //设置内存

4、relocate:

5、stack\_setup:

6、clear\_bss:

7、ldr pc, \_start\_armboot //跳转到第二阶段

3.1.2 uboot第二阶段代码

1、start\_armboot (void)函数内部

1.1、调用 init\_sequence[]函数数组里面的函数

board\_init (void)

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410 //传递给uboot的机器id和参

gd->bd->bi\_boot\_params = 0x30000100; //传递给uboot的参数存放的地址

1.2、flash\_init (); //对norflash初始化

1.3、nand\_init(); //对nandflash初始化

1.4、env\_relocate ();//环境变量初始化，例如Ip地址，串口波特率等。

代码中默认环境变量和Flash上保存的环境变量。如果flash上存在该环境变量，就使用flash中保存的环境变量，否则使用默认环境变量。

1.5、调用 main\_loop ()函数

2、main\_loop()函数内部

功能循环解析用户输入的命令，根据命令执行代码

2.1、启动内核

s = getenv ("bootcmd"); //获得启动内核命令环境变量

//如果没有按下空格键，执行启动内核命令

if (bootdelay >= 0 && s && !abortboot (bootdelay))

run\_command (s, 0);

bootcmd是在uboot运行时使用命令行设置的,可以使用print命令查看

bootcmd=”nand read jffs 0x30007fc0 kernel;bootm 0x30007fc0”

2.2 如果按下空格，启动菜单

run\_command(“menu”,0”) //官方Uboot没有该句，韦东山添加

2.3、等待输入命令，执行

for (;;)

readline (CFG\_PROMPT);

rc = run\_command (lastcommand, flag);

3.1 启动内核

s = getenv ("bootcmd");run\_command (s, 0);这两个函数实际执行的指令如下： bootcmd=”nand read.jffs2 0x30007fc0 kernel;bootm 0x30007fc0”

为什么地址是0x30007fc0，这个地址是什么地址？

因为kernel的起始地址是0x300080000,uimage前面包含了64B的数据头，因此uimage实际地址是0x30007fc0。该地址是下载地址，内核下载或者拷贝到内存的存储地址。

Uimage=header(64B)+Kernel

image\_header\_t header;

typedef struct image\_header { //头部结构体

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 \*/

/\* 加载地址，内存应该存放的起始地址，如果内核在内存中的地址（下载地址）和该地址不一致，那么执行bootm命令时，会先将内核拷贝到该地址\*/

uint32\_t ih\_load; /\*加载地址\*/

/\*运行地址，内核从该地址开始执行，一般和加载地址设为相同，也可以不同\*/

uint32\_t ih\_ep;/\*运行地址\*/

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;

加载地址和入口地址在哪里指定?

是在Uimage头部存储的，是在make image内核时确定的

//拷贝内核到加载地址

memmove ((void \*) ntohl(hdr->**ih\_load**), (uchar \*)data, len);

//data是内核所在地址，ih\_load是内核运行时应该存放的地址

为什么地址是jffs2？

Jffs2和yaffs2是一种文件系统。但是这里和文件系统没有关系。用该命令地址可以不用页对齐。

Flash分为四个区：

引导启动分区，环境变量，内核分区，文件系统分区

Boot env kernel rootfs

Include/configs/100ask24x0.h中定义了flash分区：

#define MTDPARTS\_DEFAULT "mtdparts=nandflash0:256k@0(bootloader)," \

"128k(params)," \

"2m(kernel)," \

"-(root)"

Uboot命令行中可以使用mtd命令来查看分区

nand read.jffs2 0x30007fc0 kernel可以替换为

nand read.jffs2 0x30007fc0 0x00060000 0x200000,意思是从0x00060000地址开始读2M

空间数据，kernel是这一段分区的别名，关联了起始地址和长度。

3.2 nand命令

调用do\_nand（）函数，在文件cmd\_nand.c中。有两个do\_nand()函数，使用第一个版本，该版本支持nand\_dump 命令。

do\_nand()

if (strncmp(cmd, "read", 4) == 0 || strncmp(cmd, "write", 5) == 0)

nand\_read(nand, off, &size, (u\_char \*)addr);

3.3 bootm命令

Cmd\_bootm.c中定义了启动命令bootm,bootm调用do\_bootm函数启动内核do\_bootm调用do\_bootm\_linux（）函数，该函数内部需要先设置启动参数，然后再启动内核

do\_bootm（）

do\_bootm\_linux (cmdtp, flag, argc, argv,addr, len\_ptr, verify);

theKernel = (void (\*)(int, int, uint))ntohl(hdr->ih\_ep);//入口地址

setup\_start\_tag (bd);

setup\_memory\_tags (bd);//设置内存启动参数

theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params);//跳到入口地址执行

传递参数bi\_arch\_number被称为机器Id,机器id在哪里设置？

调用 init\_sequence[]函数数组里面的函数

board\_init (void)

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410

内核根据机器Id确定内核是否支持该单板

4.1 设置启动参数

如何传递参数

Uboot把参数放置在约定的地址，内核从该地址读取参数数据。

参数地址在何处设置？

start\_armboot (void)函数内部

调用 init\_sequence[]函数数组里面的函数

board\_init (void)

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410 //传递给uboot的机器id和参

gd->bd->bi\_boot\_params = 0x30000100; //传递给uboot的参数存放的地址

struct tag { //参数称为tag标记

struct tag\_header hdr;

union {

struct tag\_core core;

struct tag\_mem32 mem;

struct tag\_videotext videotext;

struct tag\_ramdisk ramdisk;

struct tag\_initrd initrd;

struct tag\_serialnr serialnr;

struct tag\_revision revision;

struct tag\_videolfb videolfb;

struct tag\_cmdline cmdline;

} u;

};

setup\_start\_tag (bd);//设置tag头

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);//指向下一个参数的地址

}

setup\_memory\_tags (bd); //设置内存

init\_sequence[]函数数组中对dram的初始化设置了内存起始地址和大小

int dram\_init (void)

{

gd->bd->bi\_dram[0].start = PHYS\_SDRAM\_1;

gd->bd->bi\_dram[0].size = PHYS\_SDRAM\_1\_SIZE;

return 0;

}

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 (bd, commandline);//设置命令行

char \*commandline = getenv ("bootargs");命令行来源于bootargs参数

bootargs可以在uboot命令行中设置。Print查看。下面是nfs启动的设置

bootargs=noinitrd console=ttySAC0 root=/dev/nfs nfsroot=192.168.0.111:/work/nfsroot/firstfs ip=192.168.0.112: 192.168.0.111:192.168.0.1:255.255.255.0::eth0:off

static void setup\_commandline\_tag (bd\_t \*bd, char \*commandline)

{

char \*p;

if (!commandline)

return;

/\* eat leading white space \*/

for (p = commandline; \*p == ' '; p++);

/\* skip non-existent command lines so the kernel will still

\* use its default command line.

\*/

if (\*p == '\0')

return;

params->hdr.tag = ATAG\_CMDLINE;

params->hdr.size =

(sizeof (struct tag\_header) + strlen (p) + 1 + 4) >> 2;

strcpy (params->u.cmdline.cmdline, p);

params = tag\_next (params);

}

setup\_end\_tag (bd);//结束参数设置标记

//拷内存贝函数

void \* memmove(void \* dest,const void \*src,size\_t count)

{

char \*tmp, \*s;

if (dest <= src) {

tmp = (char \*) dest;

s = (char \*) src;

while (count--)

\*tmp++ = \*s++;

}

else {

tmp = (char \*) dest + count;

s = (char \*) src + count;

while (count--)

\*--tmp = \*--s;

}

return dest;

}

6.1 Uboot命令

run\_command（name）根据名字进行命令的匹配，如果找到该命令就调用相应的函数。

parse\_line (finaltoken, argv)//解析输入命令，例子：md.w 0会被分成argv[0]=”md.w” //argv[1]=0

cmdtp = find\_cmd(argv[0])//查找uboot是否有匹配的命令，有返回命令结构体

typedef struct cmd\_tbl\_s cmd\_tbl\_t;

struct cmd\_tbl\_s {

char \*name; /\* Command Name \*/

int maxargs; /\* maximum number of arguments \*/

int repeatable; /\* autorepeat allowed? 命令是否可重复，enter重复上次命令，1命令可重复 \*/

/\* Implementation function \*/

int (\*cmd)(struct cmd\_tbl\_s \*, int, int, char \* const []);//命令函数

char \*usage; /\* Usage message (short) 短的帮助信息\*/

#ifdef CONFIG\_SYS\_LONGHELP

char \*help; /\* Help message (long)长的帮助信息 \*/

#endif

#ifdef CONFIG\_AUTO\_COMPLETE

/\* do auto completion on the arguments \*/

int (\*complete)(int argc, char \* const argv[], char last\_char, int maxv, char \*cmdv[]);

#endif};

cmd\_tbl\_t \*find\_cmd (const char \*cmd)

{

cmd\_tbl\_t \*cmdtp;

cmd\_tbl\_t \*cmdtp\_temp = &\_\_u\_boot\_cmd\_start; /\*Init value \*/

const char \*p;

int len;

int n\_found = 0;

/\*

\* Some commands allow length modifiers (like "cp.b");

\* compare command name only until first dot.

\*/

len = ((p = strchr(cmd, '.')) == NULL) ? strlen (cmd) : (p - cmd);

//链接脚本里面定义\_\_u\_boot\_cmd\_start。在这段空间内搜索命令

//名字相同标识搜索到

for (cmdtp = &\_\_u\_boot\_cmd\_start;

cmdtp != &\_\_u\_boot\_cmd\_end;

cmdtp++) {

if (strncmp (cmd, cmdtp->name, len) == 0) {

if (len == strlen (cmdtp->name))

return cmdtp; /\* full match \*/

cmdtp\_temp = cmdtp; /\* abbreviated command ? \*/

n\_found++;

}

}

if (n\_found == 1) { /\* exactly one match \*/

return cmdtp\_temp;

}

return NULL; /\* not found or ambiguous command \*/

}

\_\_u\_boot\_cmd\_start = .;/\* 赋值为当前位置，起始位置\*/

.u\_boot\_cmd : { \*(.u\_boot\_cmd) }/\*指定u\_boot\_cmd段, uboot把所有的uboot命令放在该段.\*/

\_\_u\_boot\_cmd\_end = .;/\*把\_\_u\_boot\_cmd\_end赋值为当前位置,即结束位置\*/

以启动内核命令为例子分析bootm 0x30007fc0，搜索bootm命令：

U\_BOOT\_CMD(

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

"bootm - boot application image from memory\n",//项与项之间逗号隔开

"[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

);

搜索U\_BOOT\_CMD

#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}

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

展开命令

cmd\_tbl\_t \_\_u\_boot\_cmd\_##name \_\_attribute\_\_ ((unused,section (".u\_boot\_cmd"))) = {bootm, CFG\_MAXARGS, 1, do\_bootm, usage, help}

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

自己添加一个命令 cmd\_hello.c

#include <common.h>

#include <watchdog.h>

#include <command.h>

#include <image.h>

#include <malloc.h>

#include <u-boot/zlib.h>

#include <bzlib.h>

#include <environment.h>

#include <lmb.h>

#include <linux/ctype.h>

#include <asm/byteorder.h>

#include <linux/compiler.h>

int do\_hello(cmd\_tbl\_t \*cmdtp, int flag, int argc, char \* const argv[])

{

printf("hello world\n");

return 0;

}

U\_BOOT\_CMD(

hello,CFG\_MAXARGS,1,do\_hello,

"hello -just for test\n",

"hello,this is long help for command hello...\n"

)

命令源文件都在common文件夹下，把代码保存到common目录下

修改common文件夹下的Makefile

COBJS = main.o xxx.o yyy.o cmd\_hello.o

Make重新编译Uboot