Hi3536 uboot引导内核全过程

本文讲述uboot引导内核启动的全部过程，uboot版本为2010.06

# 1、arch/arm/cpu/hi3536/u-boot.lds

连接文件，不同平台不一样。

OUTPUT\_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm") //elf格式 32位，小端

OUTPUT\_ARCH(arm) //arm架构

ENTRY(\_start) //代码进入点arch/arm/cpu/hi3536/start.S中定义，一切从这里开始

SECTIONS

{

. = 0x00000000; //可执行文件入口地址

. = ALIGN(4);

.text : //代码段

{

\_\_text\_start = .;

arch/arm/cpu/hi3536/start.o (.text)

drivers/ddr/ddr\_training\_impl.o (.text)

drivers/ddr/ddr\_training\_ctl.o (.text)

drivers/ddr/ddr\_training\_boot.o (.text)

drivers/ddr/ddr\_training\_custom.o (.text)

\_\_init\_end = .;

ASSERT(((\_\_init\_end - \_\_text\_start) < 0x16000), "init sections too big!");

\*(.text)

}

. = ALIGN(4);

.rodata : { \*(SORT\_BY\_ALIGNMENT(SORT\_BY\_NAME(.rodata\*))) } //只读数据段

. = ALIGN(4);

.data : { \*(.data) } //数据段

. = ALIGN(4);

.got : { \*(.got) }

\_\_u\_boot\_cmd\_start = .; //存放uboot命令

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

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

. = ALIGN(4);

\_\_bss\_start = .; //bss段

.bss : { \*(.bss) }

\_end = .;

}

# 2、arch/arm/cpu/hi3536/start.S

Arm在uboot下的入口文件，纯汇编，由于代码太多，这里描述基本过程：

（1）定义入口

.globl \_start

\_start: b reset //跳转到复位

//以下是各种异常处理入口定义

ldr pc, \_undefined\_instruction

ldr pc, \_software\_interrupt

ldr pc, \_prefetch\_abort

ldr pc, \_data\_abort

ldr pc, \_not\_used

ldr pc, \_irq

ldr pc, \_fiq

（2）进入reset第一步

A. set the cpu to SVC32 mode

B. Invalidate L1 I/D

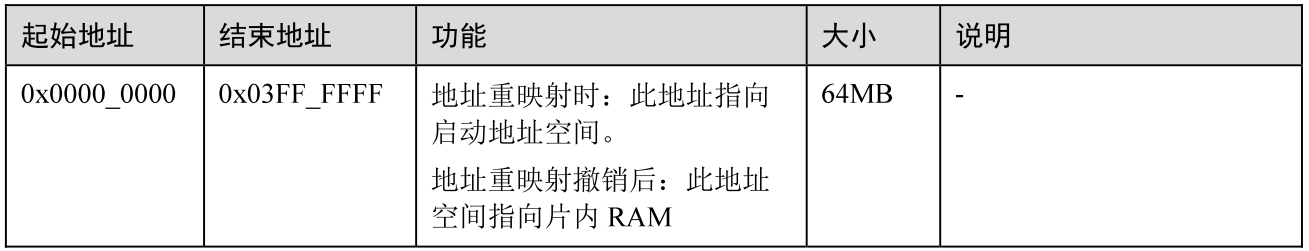
C. Invalidate L1 D-cache

D. disable MMU stuff and caches

（3）第二步

判断boot方式，从bootrom / spi /nand中的一种启动，撤销地址空间映射。

假设启动方式为spi nor flash启动，这时起始地址空间映射到spi nor flash地址空间，但是spi nor flash不能直接执行代码，这里猜测是：映射后硬件自己从spi nor flash中将部分代码搬运到内部开始执行。



（4）第三步

进行pll /flash/ddr等的初始化

（5）第4步

拷贝flash数据到DDR空间。在这步之前只是搬运了初始化用的最基本的部分到内部，但是整个uboot并未搬运到DDR，这里则进行完整的搬运：r1-DDR地址0x40c00000(src/board/hi3536/config.mk中定义)，r0-flash地址为spi地址，r2大小为\_bss\_start - \_armboot\_start（\_start）,memcpy（r1, r0, r2）。内存布局见第8节

根据uboot.map可以看到连接收地址发生了变化从0x40c00000开始，修改这个的地方由底层的config.mk完成：

LDFLAGS += -Bstatic -T $(obj)u-boot.lds $(PLATFORM\_LDFLAGS)

ifneq ($(TEXT\_BASE),)

LDFLAGS += -Ttext $(TEXT\_BASE)

endif

连接地址替换成TEXT\_BASE（src/board/hi3536/config.mk定义）

（6）第5步

跳转到DDR，进入第一个C语言入口

clbss\_l:

str r2, [r0] @ clear BSS location

cmp r0, r1 @ are we at the end yet

add r0, r0, #4 @ increment clear index pointer

bne clbss\_l @ keep clearing till at end

ldr pc, \_start\_armboot @ jump to C code

\_start\_armboot: .word start\_armboot（0x40c04fdc start\_armboot连接地址）

# 3、src/arch/arm/lib/board.c

init\_fnc\_t \*init\_sequence[] = { //初始化函数列表

timer\_init, /\* initialize timer before usb init \*/

board\_init, /\* basic board dependent setup \*/

env\_init, /\* initialize environment \*/

init\_baudrate, /\* initialze baudrate settings \*/

serial\_init, /\* serial communications setup \*/

console\_init\_f, /\* stage 1 init of console \*/

display\_banner, /\* say that we are here \*/

dram\_init, /\* configure available RAM banks \*/

NULL,

};

void start\_armboot (void)

{

/\* Pointer is writable since we allocated a register for it \*/

gd = (gd\_t\*)(\_armboot\_start - CONFIG\_SYS\_MALLOC\_LEN - sizeof(gd\_t));

\_armboot\_start = 0x40c00000（内存起始地址0x40000000，预留了12M）

CONFIG\_SYS\_MALLOC\_LEN = 4 \* 1024 \* 1024（自己修改）

…

//运行所有初始化函数

for (init\_fnc\_ptr = init\_sequence; \*init\_fnc\_ptr; ++init\_fnc\_ptr) {

if ((\*init\_fnc\_ptr)() != 0) {

hang ();

}

}

…

#ifdef CONFIG\_CMD\_SF

spi\_flash\_probe(0, 0, 0, 0); //初始化spi flash

#endif

env\_relocate (); //初始化uboot环境变量

…

#if defined(CONFIG\_CMD\_NET)

eth\_initialize(gd->bd); //网络初始化

#endif

#ifdef START\_UBOOT\_NO\_NET\_OPERATION

int ret = 0;

ret = eth\_init(gd->bd); //网络驱动初始化，该部分自己添加

if(!ret)

eth\_halt();

#endif

…

for (;;) {

main\_loop (); //进入main主函数

}

}

# 4、src/common/main.c

void main\_loop (void)

{

…

#if defined(CONFIG\_BOOTDELAY) && (CONFIG\_BOOTDELAY >= 0)

s = getenv ("bootdelay"); //获取bootdelay次数（即秒数）

bootdelay = s ? (int)simple\_strtol(s, NULL, 10) : CONFIG\_BOOTDELAY;

debug ("### main\_loop entered: bootdelay=%d\n\n", bootdelay);

#ifdef CONFIG\_HI3536\_A7

//modify by hlb

//s = getenv("slave\_cmd");

s = getenv("bootcmd");

#else

s = getenv("bootcmd"); //获取bootcmd命令内容

/\* bootcmd=usbupdate; tftpupdate; fsload; bootm

usbupdate: 扫描U盘，升级U盘文件，自己添加不进行说明

tftpupdate: tftp自动化升级，自己添加不进行说明

fsload: 加载内核文件, 自己修改，后续章节说明

bootm：引导内核，后续章节说明

\*/

#endif

debug ("### main\_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");

//在bootdelay时间内连续输入3个字符\*(自己修改为\*)，则不执行bootcmd命令，进入uboot命令行，否则执行bootcmd命令

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

# ifndef CONFIG\_SYS\_HUSH\_PARSER

run\_command (s, 0); //执行bootcmd命令

# else

parse\_string\_outer(s, FLAG\_PARSE\_SEMICOLON |

FLAG\_EXIT\_FROM\_LOOP);

# endif

}

…

//进入uboot命令行，响应用户输入的命令

for (;;) {

len = readline (CONFIG\_SYS\_PROMPT);

flag = 0; /\* assume no special flags for now \*/

if (len > 0)

strcpy (lastcommand, console\_buffer);

else if (len == 0)

flag |= CMD\_FLAG\_REPEAT;

if (len == -1)

puts ("<INTERRUPT>\n");

else

run\_command(lastcommand, flag);

/\* invalid command or not repeatable, forget it \*/

lastcommand[0] = 0;

}

}

# 5、src/common/cmd\_fsload.c

bootcmd环境变量中fsload命令，实际运行函数就为该文件的do\_fsload函数。

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

{

char \*fsname;

char \*filename = "/boot/uImage"; //读取内核完整路径名

int size = 0;

struct part\_info \*part;

ulong offset = CONFIG\_SYS\_LOAD\_ADDR; //=0x42000000，内核加载的内存地址

part = spi\_cramfs\_init(); //读取内核所在根文件系统信息

if (part ==0)

{

return 1;

}

printf("### squashfs loading '%s' to 0x%lx\n", filename, offset);

if(squashfs\_check(part)) //这里使用自己添加的squashfs文件系统，默认是cramfs。

{

size = squashfs\_fload((char \*) offset, part, filename); //加载内核到上述内存地址

}

if (size > 0) { //加载成功

char buf[10];

printf("### %s load complete: %d bytes loaded to 0x%lx\n",

fsname, size, offset);

sprintf(buf, "%x", size);

setenv("filesize", buf);

} else { //加载失败

printf("### %s LOAD ERROR<%x> for %s!\n", fsname, size, filename);

}

return !(size > 0);

}

# 6、src/common/cmd\_bootm.c

bootcmd环境变量中bootm命令，实际运行函数就为该文件的do\_bootm函数。

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

{

…

if (bootm\_start(cmdtp, flag, argc, argv)) //该函数解析内核image头，获取内核信息

return 1;

/\*

images.os.image\_start = 0x42000040 (0x40 64字节头大小)

images.os.image\_len = 0x25fa88

images.os.type = IH\_TYPE\_KERNEL

images.os.comp = IH\_COMP\_NONE

images.os.os = IH\_OS\_LINUX

images.os.start= 0x42000000

images.os.end = 0x4225fac8

images.os.load = 0x40008000 真正的内核会加载到该地方

\*/

…

ret = bootm\_load\_os(images.os, &load\_end, 1);

/\*

将内核从os.image\_start搬运到os.load地址

load\_end = 0x40267a88(0x40008000+0x25fa88)

\*/

…

boot\_fn = boot\_os[images.os.os]; //根据操作系统类型获取对应操作系统入口函数，这里为do\_bootm\_linux

…

boot\_fn(0, argc, argv, &images); //运行do\_bootm\_linux函数，成功则会进入内核

…

do\_reset (cmdtp, flag, argc, argv); //如果失败则重启uboot

}

# 7、src/arch/arm/lib/bootm.c

int do\_bootm\_linux(int flag, int argc, char \*argv[], bootm\_headers\_t \*images)

{

int machid = bd->bi\_arch\_number; //传递给内核的机器码hi3536 0x8000

void (\*theKernel)(int zero, int arch, uint params); //进入内核函数指针

theKernel = (void (\*)(int, int, uint))images->ep; //ep = 0x40008000

…

#if defined (CONFIG\_SETUP\_MEMORY\_TAGS) || \

defined (CONFIG\_CMDLINE\_TAG) || \

defined (CONFIG\_INITRD\_TAG) || \

defined (CONFIG\_SERIAL\_TAG) || \

defined (CONFIG\_REVISION\_TAG)

setup\_start\_tag (bd);

#ifdef CONFIG\_SETUP\_MEMORY\_TAGS

setup\_memory\_tags (bd);

#endif

#ifdef CONFIG\_CMDLINE\_TAG

setup\_commandline\_tag (bd, commandline);

#endif

….

setup\_end\_tag (bd);

#endif

/\* 需要传递给内核的各个TAG，放入参数列表 \*/

…

theKernel (0, machid, bd->bi\_boot\_params); //进入内核，无返回

/\* 第2个参数为机器码，第二个参数为传递给内核参数列表的首地址=0x40000100 \*/

}

# 8、uboot阶段的内存布局

