**Uboot是如何启动内核的**

**1.uboot启动内核的代码缩减如下：**

Uboot 1.16/lib\_arm/board.c中start\_armboot()函数调用/common/main.c中main\_loop()函数，在main\_loop()中有uboot启动内核的代码：

s = getenv ("bootcmd");
debug ("### main\_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");
if (bootdelay >= 0 && s && !abortboot (bootdelay))
{
                run\_command (s, 0);
}

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

<1> nand read.jffs2 0x30007FC0 kernel

从nand读出内核：

从哪里读？  ： kernel分区

读到哪里去？：0x30007FC0

何为分区？

简单的说就是将nand划分为几个区域，一般如下：

**bootloader->params->kernel->root**

这些分区划分在**/include/configs/100ask24x0.h**中写死的：

#define MTDPARTS\_DEFAULT "mtdparts=nandflash0:256k@0(bootloader)," \
           "128k(params)," \
           "2m(kernel)," \
           "-(root)"

进入uboot执行mtd ,可以查看已有分区:

**#        name                     大小                在nand上的起始地址**

**0        bootloader         0x00040000                0x00000000**

**1        params              0x00020000               0x00040000**

**2        kernel                0x00200000                0x00060000**

**3        root                0xfda00000                0x00260000**

上面的nand read.jffs2 0x30007FC0 kernel等价于：

nand read.jffs2 0x30007FC0 0x00060000 0x00200000

注：read.jffs2并不是指定特定的格式， 仅表示不需要块/页对齐,所以kernel的分区大小可以随意定。

<2> bootm 0x30007FC0

关键函数do\_bootm()

flash上存的内核：uImage

**uImage = 头部+真正的内核**

头部的定义如下：

typedef struct image\_header {
        uint32\_t        ih\_magic;

        uint32\_t        ih\_hcrc;

        uint32\_t        ih\_time;

        uint32\_t        ih\_size;

        uint32\_t        ih\_load;

        uint32\_t        ih\_ep;

        uint32\_t        ih\_dcrc;

        uint8\_t                ih\_os;

        uint8\_t                ih\_arch;

        uint8\_t                ih\_type;

        uint8\_t                ih\_comp;

        uint8\_t                ih\_name[IH\_NMLEN];

 } image\_header\_t;

我们需要关心：

        uint32\_t        ih\_load;

        uint32\_t        ih\_ep;

ih\_load是加载地址，即内核运行是应该位于的地方

ih\_ep是入口地址，即内核的入口地址

这与uboot类似，uboot的加载地址是TEXT\_BASE = 0x33F80000；入口地址是start.S中的\_start。

从nand读出来的内核可以放在ram中的任意地方，如0x31000000，0x32000000等等，只要它不破坏uboot所占用的内存空间就可以

既然设定好了加载地址和入口地址, 为什么内核还能随意放？

因为uImage有一个头部！头部里有加载地址和入口地址，当我们用bootm xxx时，

do\_bootm先去读uImage的头部以获取该uImage的加载地址和入口地址，当发现该uImage目前所处的内存地址不等于它的加载地址时，会将uImage移动到它的加载地址上，代码中体现如下：

uboot 1.16/common/cmd\_bootm.c中的bootm\_load\_os()函数

case IH\_COMP\_NONE:：
if (load != image\_start)
{
                memmove\_wd ((void \*)load, (void \*)image\_start, image\_len, CHUNKSZ);
}

另外，当内核正好处于头部指定的加载地址，便不用uboot的do\_bootm函数来帮我们搬运内核了，可以缩短启动时间。这就是为什么我们一般都下载uImage到0x30007FC0的原因。

内核加载地址是0x30008000，而头部的大小64个字节，将内核拷贝到0x30007FC0，加上头部的64个字节，内核正好位于0x30008000处。

总结**bootm做了什么：**

**1. 读取头部**

**2. 将内核移动到加载地址**

**3. 启动内核**

具体如何启动内核？

使用在/lib\_arm/bootm.c定义的do\_bootm\_linux()，我们已经知道入口地址，只需跳到入口地址就可以启动linux内核了，在这之前需要做一件事———— uboot传递参数(启动参数)给内核。

启动代码在do\_bootm\_linux()函数：

void (\*theKernel)(int zero, int arch, uint params); //定义函数指针theKernel

        **theKernel = (void (\*)(int, int, uint))images->ep; //先是将入口地址赋值给theKernel**

**theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params); //然后是调用thekernel**，以0，**bd->bi\_arch\_number**，bd->bi\_boot\_params为参数

下面分析这三个参数：

1. 0—相当于mov ，ro #0

2. bd->bi\_arch\_number ：uboot机器码，这个在/board/100ask24x0.c设置: gd->bd->bi\_arch\_number = MACH\_TYPE\_S3C2440，MACH\_TYPE\_S3C2440在/arch/arm/asm/mach-types.h定义：362, 内核机器码和uboot机器码必须一致才能启动内核

2. **bd->bi\_boot\_parmas**--- 启动参数地址

也是在在/board/100ask24x0.c设置: gd->bd->bi\_boot\_params = 0x30000100;

**启动参数(tag)在哪里设置？**

在lib\_arm/armlinux.c设置：

setup\_start\_tag (bd);
setup\_revision\_tag (parmas);
setup\_memory\_tags (bd);
setup\_commandline\_tag (bd, commandline);
setup\_initrd\_tag (bd, images->rd\_start, images->rd\_end);
setup\_videolfb\_tag ((gd\_t \*) gd);
setup\_end\_tag (bd);

### 每一个启动参数对应一个tag结构体，所谓的设置传递参数其实就是初始化这些tag的值，想了解这个结构体以及这些tag的值是如何设置的请看[嵌入式Linux应用开发完全手册](http://www.baidu.com/link?url=C44fCOU_eW_knVMqDMaGplh7olrkErU7iK_06hI8t9U0_V6TWPoEqanOC8goWQw2)关于uboot章节

我们来看setup\_start\_tag(bd)函数：

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);
}

再看setup\_commandline\_tag (bd , commandline)：

static void setup\_commandline\_tag (bd\_t \*bd, char \*commandline)
{
 // commandline就是我们的bootargs
        char \*p;
        if (!commandline)
                return;
        for (p = commandline; \*p == ' '; p++);
        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);
}

内核启动时会读取这些tag(参数)并跳转启动。

更多uboot启动内核的细节请点击**阅读原文**观看毕业班视频自己写uboot。