

## ArmLinux BOOTLOADER 全程详解

作者: 董磊塋

### 1.几个重要的概念

#### COMPRESSED KERNEL and DECOMPRESSED KERNEL

#### Jffs2 File System

#### RAMDISK

启动参数(摘自 IBM developer)

### 2.开发环境和开发板配置:

### 3.启动方式:

### 4.代码分析

网上关于 Linux 的 BOOTLOADER 文章不少了,但是大都是 vivi,blob 等比较庞大的程序,读起来不太方便,编译出的文件也比较大,而且更多的是面向开发用的引导代码,做成产品时还要裁减,这一定程度影响了开发速度,对初学者学习开销也比较大,在此分析一种简单的 BOOTLOADER,是在三星公司提供的 2410 BOOTLOADER 上稍微修改后的结果,编译出来的文件大小不超过 4k,希望对大家有所帮助.

### 1.几个重要的概念

#### COMPRESSED KERNEL and DECOMPRESSED KERNEL

压缩后的 KERNEL,按照文档资料,现在不提倡使用 DECOMPRESSED KERNEL,而要使用 COMPRESSED KERNEL,它包括了解压器.因此要在 ram 分配时给压缩和解压的 KERNEL 提供足够空间,这样它们不会相互覆盖.

当执行指令跳转到 COMPRESSED KERNEL 后,解压器就开始工作,如果解压器探测到解压的代码会覆盖掉 COMPRESSED KERNEL,那它会直接跳到 COMPRESSED KERNEL 后存放数据,并且重新定位 KERNEL,所以如果没有足够空间,就会出错.

#### Jffs2 File System

可以使 armlinux 应用中产生的数据保存在 FLASH 上,我的板子还没用到这个.

#### RAMDISK

使用 RAMDISK 可以使 ROOT FILE SYSTEM 在没有其他设备的情况下启动.一般有两种加载方式,我就介绍最常用的吧,把 COMPRESSED RAMDISK IMAGE 放到指定地址,然后由 BOOTLOADER 把这个地址通过启动参数的方式 ATAG\_INITRD2 传递给 KERNEL.具体看代码分析.

启动参数(摘自 IBM developer)

在调用内核之前,应该作一步准备工作,即:设置 Linux 内核的启动参数. Linux 2.4.x 以后的内核都期望以标记列表(tagged list)的形式来传递启动参数.启动参数标记列表以标记 ATAG\_CORE 开始,以标记 ATAG\_NONE 结束.每个标记由标识被传递参数的 tag\_header 结构以及随后的参数值数据结构来组成.数据结构 tag 和 tag\_header 定义在 Linux 内核源码的 include/asm/setup.h 头文件中.

在嵌入式 Linux 系统中, 通常需由 **BOOTLOADER** 设置的常见启动参数有: **ATAG\_CORE**、**ATAG\_MEM**、**ATAG\_CMDLINE**、**ATAG\_RAMDISK**、**ATAG\_INITRD** 等。

(注)参数也可以用 **COMMANDLINE** 来设定,在我的 **BOOTLOADER** 里,我两种都用了。

## 2.开发环境和开发板配置:

**CPU:S3C2410**,**BANK6** 上有 **64M** 的 **SDRAM**(两块),**BANK0** 上有 **32M NOR FLASH**,串口当然是逃不掉的. 这样,按照数据手册,地址分配如下:

**0x4000\_0000** 开始是 **4k** 的片内 **DRAM**.

**0x0000\_0000** 开始是 **32M FLASH 16bit** 宽度

**0x3000\_0000** 开始是 **64M SDRAM 32bit** 宽度

注意:控制寄存器中的 **BANK6** 和 **BANK7** 部分必须相同.

**0x4000\_0000**(片内 **DRAM**)存放 **4k** 以内的 **BOOTLOADER IMAGE**

**0x3000\_0100** 开始存放启动参数

**0x3120\_0000** 存放 **COMPRESSED KERNEL IMAGE**

**0x3200\_0000** 存放 **COMPRESSED RAMDISK**

**0x3000\_8000** 指定为 **DECOMPRESSED KERNEL IMAGE ADDRESS**

**0x3040\_0000** 指定为 **DECOMPRESSED RAMDISK IMAGE ADDRESS**

开发环境:**Redhat Linux**,**armgcc toolchain**, **armlinux KERNEL**

如何建立 **armgcc** 的编译环境:建议使用 **toolchain**,而不要自己去编译 **armgcc**,偶试过好多次,都以失败告终.

先下载 **arm-gcc 3.3.2 toolchain**

将 **arm-linux-gcc-3.3.2.tar.bz2** 解压到 **/toolchain**

```
# tar jxvf arm-linux-gcc-3.3.2.tar.bz2
```

```
# mv /usr/local/arm/3.3.2 /toolchain
```

在 **makefile** 中在把 **arch=arm CROSS\_COMPILE** 设置成 **toolchain** 的路径

还有就是 **INCLUDE = -I ../include -I /root/my/usr/local/arm/3.3.2/include.**,否则库函数就不能用了

### 3.启动方式:

可以放在 **FLASH** 里启动,或者用 **Jtag** 仿真器.由于使用 **NOR FLASH**,根据 **2410** 的手册,片内的 **4K DRAM** 在不需要设置便可以直接使用,而其他存储器必须先初始化,比如告诉 **memory controller**,**BANK6** 里有两块 **SDRAM**,数据宽度是 **32bit**,= =.否则 **memory control** 会按照复位后的默认值来处理存储器.这样读写就会产生错误.

所以第一步,通过仿真器把执行代码放到 **0x4000\_0000**, (在编译的时候,设定 **TEXT\_BASE=0x40000000**)

第二步,通过 **AxD** 把 **linux KERNEL IMAGE** 放到目标地址(**SDRAM**)中,等待调用

第三步,执行 **BOOTLOADER** 代码,从串口得到调试数据,引导 **armlinux**

### 4.代码分析

讲了那么多执行的步骤,是想让大家对启动有个大概印象,接着就是 **BOOTLOADER** 内部的代码分析了,**BOOTLOADER** 文章内容网上很多,我这里精简了下,删除了不必要的功能.

**BOOTLOADER** 一般分为 **2** 部分,汇编部分和 **c** 语言部分,汇编部分执行简单的硬件初始化,**C** 部分负责复制数据,设置启动参数,串口通信等功能.

**BOOTLOADER** 的生命周期:

1. 初始化硬件,比如设置 **UART**(至少设置一个),检测存储器 = =.
2. 设置启动参数,这是为了告诉内核硬件的信息,比如用哪个启动界面,波特率 = =.
3. 跳转到 **Linux KERNEL** 的首地址.
4. 消亡

当然,在引导阶段,象 **vivi** 等,都用虚地址,如果你嫌烦的话,就用实地址,都一样.

我们来看代码:

**2410init.s**

**.global \_start**//开始执行处

**\_start:**

//下面是中断向量

**b reset @ Supervisor Mode**//重新启动后的跳转

.....

.....

**reset:**

**ldr r0,=WTCON** /WTCON 地址为 53000000,watchdog 的控制寄存器 \*/

**ldr r1,=0x0** /\*关 watchdog\*/

**str r1,[r0]**

**ldr r0,=INTMSK**

**ldr r1,=0xffffffff** /\*屏蔽所有中断\*/

**str r1,[r0]**

**ldr r0,=INTSUBMSK**

**ldr r1,=0x3ff** /\*子中断也一样\*/

**str r1,[r0]**

**/\* Initialize Ports...for display LED.\*/**

**ldr r0,=GPFCON**

**ldr r1,=0x55aa**

**str r1, [r0]**

**ldr r0,=GPFUP**

**ldr r1,=0xff**

**str r1, [r0]**

**ldr r0,=GPFDAT**

```

ldr r1,=POWEROFFLED1

str r1,[r0]

/* Setup clock Divider control register

* you must configure CLKDIVN before LOCKTIME or MPLL UPLL

* because default CLKDIVN 1,1,1 set the SDMRAM Timing Conflict

nop

* FCLK:HCLK:PCLK = 1:2:4 in this case

*/

ldr r0,=CLKDIVN

ldr r1,=0x3

str r1,[r0]


/*To reduce PLL lock time, adjust the LOCKTIME register. */

ldr r0,=LOCKTIME

ldr r1,=0xffffffff

str r1,[r0]

/*Configure MPLL */

ldr r0,=MPLLCON

ldr r1,=((M_MDIV<<12)+(M_PDIV<<4)+M_SDIV) //Fin=12MHz,Fout=203MHz

str r1,[r0]

ldr r1,=GSTATUS2

ldr r10,[r1]

tst r10,#OFFRST

```

```
bne 1000f
```

//以上这段,我没动,就用三星写的了,下面是主要要改的地方

```
/* MEMORY CONTROLLER(MC)设置*/
```

```
add r0,pc,#MCDATA - (.+8)// r0 指向 MCDATA 地址,那里存放着 MC 初始化要用到的数据
```

```
ldr r1,=BWSCON // r1 指向 MC 控制器寄存器的首地址
```

```
add r2,r0,#52 // 复制次数,偏移 52 字
```

```
1: //按照偏移量进行循环复制
```

```
ldr r3,[r0],#4
```

```
str r3,[r1],#4
```

```
cmp r2,r0
```

```
bne 1b
```

```
.align 2
```

```
MCDATA:
```

```
.word
```

```
(0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<24)+(B7_BWSCON<<28))
```

上面这行就是 **BWSCON** 的数据,具体参数意义如下:

需要更改设置 **DW6** 和 **DW7** 都设置成 **10**,即 **32bit**,**DW0** 设置成 **01**,即 **16bit**

下面都是每个 **BANK** 的控制器数据,大都是时钟相关,可以用默认值,设置完 **MC** 后,就跳到调用 **main** 函数的部分

```
.word
```

```
((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tacp<
```

<2)+(B0\_PMC))

.word

((B1\_Tacs<<13)+(B1\_Tcos<<11)+(B1\_Tacc<<8)+(B1\_Tcoh<<6)+(B1\_Tah<<4)+(B1\_Tacp<<2)+(B1\_PMC))

.word

((B2\_Tacs<<13)+(B2\_Tcos<<11)+(B2\_Tacc<<8)+(B2\_Tcoh<<6)+(B2\_Tah<<4)+(B2\_Tacp<<2)+(B2\_PMC))

.word

((B3\_Tacs<<13)+(B3\_Tcos<<11)+(B3\_Tacc<<8)+(B3\_Tcoh<<6)+(B3\_Tah<<4)+(B3\_Tacp<<2)+(B3\_PMC))

.word

((B4\_Tacs<<13)+(B4\_Tcos<<11)+(B4\_Tacc<<8)+(B4\_Tcoh<<6)+(B4\_Tah<<4)+(B4\_Tacp<<2)+(B4\_PMC))

.word

((B5\_Tacs<<13)+(B5\_Tcos<<11)+(B5\_Tacc<<8)+(B5\_Tcoh<<6)+(B5\_Tah<<4)+(B5\_Tacp<<2)+(B5\_PMC))

.word ((B6\_MT<<15)+(B6\_Trtd<<2)+(B6\_SCAN))

.word ((B7\_MT<<15)+(B7\_Trtd<<2)+(B7\_SCAN))

.word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)

.word 0xB2 /\* REFRESH Control Register \*/

.word 0x30 /\* BANKSIZE Register : Burst Mode \*/

.word 0x30 /\* SDRAM Mode Register \*/

.align 2

.global call\_main //调用 main 函数,函数参数都为 0

call\_main:

ldr sp,STACK\_START

mov fp,#0 /\* no previous frame, so fp=0\*/

```
mov a1, #0 /* set argc to 0*/
```

```
mov a2, #0 /* set argv to NUL*/
```

```
bl main /* call main*/
```

```
STACK_START:
```

```
.word STACK_BASE
```

```
undefined_instruction:
```

```
software_interrupt:
```

```
prefetch_abort:
```

```
data_abort:
```

```
not_used:
```

```
irq:
```

```
fiq:
```

```
/*以上是主要的汇编部分,实现了时钟设置,串口设置 watchdog 关闭,中断关闭功能(如果有需要还可以降频使用),然后转入 main*/
```

```
2410init.c file
```

```
int main(int argc,char **argv)
```

```
{
```

```
u32 test = 0;
```

```
void (*theKERNEL)(int zero, int arch, unsigned long params_addr) = (void (*)(int, int, unsigned long))RAM_COMPRESSED_KERNEL_BASE; //压缩后的 IMAGE 地址
```

```
int i,k=0;
```

```
// downPt=(RAM_COMPRESSED_KERNEL_BASE);
```

```
chkBs=(_RAM_STARTADDRESS); //SDRAM 开始的地方
```

```
// fromPt=(FLASH_LINUXKERNEL);
```



```

MMU_EnableICache();

ChangeClockDivider(1,1); // 1:2:4

ChangeMPIIValue(M_MDIV,M_PDIV,M_SDIV); //Fin=12MHz FCLK=200MHz

Port_Init();//设置 I/O 端口,在使用 com 口前,必须调用这个函数,否则通信芯片根本得不到数据

Uart_Init(PCLK, 115200);//PCLK 使用默认的 200000,波特率 115200

/******(检查 ram 空间)*****/

Uart_SendString("\n\tLinux S3C2410 Nor BOOTLOADER\n");

Uart_SendString("\n\tChecking SDRAM 2410loader.c...\n");

for(;chkBs<0x33FA0140;chkBs=chkBs+0x4,test++)//

//根据我的经验,最好以一个字节为递增,我们的板子,在 256byte 递增检测的时候是没问题的,但是

//以 1byte 递增就出错了,第 13 跟数据线随几的会冒"1",检测出来是硬件问题,现象如下

//用仿真器下代码测试 SDRAM, 开始没贴 28F128A3J FLASH 片子, 测试结果很好, 但在上了 FLASH 片子
//之后, 测试数据 (data) 为 0x00000400 连续成批写入读出时, 操作大约 1k 左右内存空间就会出错, //
而且随机。那个出错数据总是变为 0x00002400, 数据总线 10 位和 13 位又没短路发生。用其他数据//测试
比如 0x00000200; 0x00000800 没这问题。dx 帮忙。

//至今没有解决,所以我用不了 Flash.

{

chkPt1 = chkBs;

*(u32 *)chkPt1 = test;//写数据

if(*(u32 *)chkPt1==1024)//读数据和写入的是否一样?

{

chkPt1 += 4;

Led_Display(1);

Led_Display(2);

```

```

Led_Display(3);

Led_Display(4);

}

else

goto error;

}

Uart_SendString("\n\tSDRAM Check Successful!\n\tMemory Maping...");

get_memory_map();

//获得可用 memory 信息,做成列表,后面会作为启动参数传给 KERNEL

//所谓内存映射就是指在 4GB 物理地址空间中有哪些地址范围被分配用来寻址系统的 RAM 单元。

Uart_SendString("\n\tMemory Map Successful!\n");

//我用仿真器把 KERNEL,RAMDISK 直接放在 SDRAM 上,所以下面这段是不需要的,但是如果
KERNEL,RAMDISK 在 FLASH 里,那就需要。

/*****(copy linux KERNEL)*****/

Uart_SendString("\tLoading KERNEL IMAGE from FLASH... \n ");

Uart_SendString("\tand copy KERNEL IMAGE to SDRAM at 0x31000000\n");

Uart_SendString("\t\tby LEIJUN DONG dongleijun4000@hotmail.com \n");

for(k = 0;k < 196608;k++,downPt += 1,fromPt += 1)//3*1024*1024/32linux KERNEL
des,src,length=3M

* (u32 *)downPt = * (u32 *)fromPt;

/***** (load RAMDISK) *****/

Uart_SendString("\t\tloading COMPRESSED RAMDISK...\n");

downPt=(RAM_COMPRESSED_RAMDISK_BASE);

fromPt=(FLASH_RAMDISK_BASE);

```

```
for(k = 0;k < 196608;k++,downPt += 1,fromPt += 1)//3*1024*1024/32linux KERNEL  
des,src,length=3M
```

```
* (u32 *)downPt = * (u32 *)fromPt;
```

```
/******jffs2 文件系统,在开发中如果用不到 FLASH,这段也可以不要******/
```

```
Uart_SendString("\t\tloading jffs2...\n");
```

```
downPt=(RAM_JFFS2);
```

```
fromPt=(FLASH_JFFS2);
```

```
for(k = 0;k < (1024*1024/32);k++,downPt += 1,fromPt += 1)
```

```
* (u32 *)downPt = * (u32 *)fromPt;
```

```
Uart_SendString( "Load Success...Run...\n ");
```

```
/******(setup param)******/
```

```
setup_start_tag();//开始设置启动参数
```

```
setup_memory_tags();//内存印象
```

```
setup_commandline_tag("console=ttyS0,115200n8");//启动命令行
```

```
setup_initrd2_tag();//root device
```

```
setup_RAMDISK_tag();//ramdisk image
```

```
setup_end_tag();
```

```
/*关 I-cache */
```

```
asm ("mrc p15, 0, %0, c1, c0, 0": "=r" (i));
```

```
i &= ~0x1000;
```

```
asm ("mcr p15, 0, %0, c1, c0, 0": : "r" (i));
```

```
/* flush I-cache */
```

```
asm ("mcr p15, 0, %0, c7, c5, 0": : "r" (i));
```

```

//下面这行就跳到了 COMPRESSED KERNEL 的首地址

theKERNEL(0, ARCH_NUMBER, (unsigned long *) (RAM_BOOT_PARAMS));

//启动 kernel 时候,I-cache 可以开也可以关,r0 必须是 0,r1 必须是 CPU 型号
(可以从 linux/arch/arm/tools/mach-types 中找到),r2 必须是参数的物理开始地址

/*****END*****/

error:

Uart_SendString("\n\nPanic SDRAM check error!\n");

return 0;

}

static void setup_start_tag(void)

{

params = (struct tag *)RAM_BOOT_PARAMS;//启动参数开始的地址

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(void)

```

```

{

```

```

int i;

for(i = 0; i < NUM_MEM_AREAS; i++) {

if(memory_map[i].used) {

params->hdr.tag = ATAG_MEM;

params->hdr.size = tag_size(tag_mem32);

params->u.mem.start = memory_map[i].start;

params->u.mem.size = memory_map[i].len;

params = tag_next(params);

}

}

}

```

```

static void setup_commandline_tag(char *commandline)

{

int i = 0;

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

* use its default command line.

*/

params->hdr.tag = ATAG_CMDLINE;

params->hdr.size = 8;

//console=ttyS0,115200n8

```

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

```
params = tag_next(params);
```

```
}
```

```
static void setup_initrd2_tag(void)
```

```
{
```

```
/* an ATAG_INITRD node tells the kernel where the compressed
```

```
* ramdisk can be found. ATAG_RDIMG is a better name, actually.
```

```
*/
```

```
params->hdr.tag = ATAG_INITRD2;
```

```
params->hdr.size = tag_size(tag_initrd);
```

```
params->u.initrd.start = RAM_COMPRESSED_RAMDISK_BASE;
```

```
params->u.initrd.size = 2047; //k byte
```

```
params = tag_next(params);
```

```
}
```

```
static void setup_ramdisk_tag(void)
```

```
{
```

```
/* an ATAG_RAMDISK node tells the kernel how large the
```

```
* decompressed ramdisk will become.
```

```
*/
```

```

params->hdr.tag = ATAG_RAMDISK;

params->hdr.size = tag_size(tag_ramdisk);

params->u.ramdisk.start = RAM_DECOMPRESSED_RAMDISK_BASE;

params->u.ramdisk.size = 7.8*1024; //k byte

params->u.ramdisk.flags = 1; // automatically load ramdisk

params = tag_next(params);

}

```

```

static void setup_end_tag(void)

{

params->hdr.tag = ATAG_NONE;

params->hdr.size = 0;

} void Uart_Init(int pclk,int baud)//串口是很重要的

{

int i;

if(pclk == 0)

pclk = PCLK;

rUFCON0 = 0x0; //UART channel 0 FIFO control register, FIFO disable

rUMCON0 = 0x0; //UART chaneel 0 MODEM control register, AFC disable


//UART0

rULCON0 = 0x3; //Line control register : Normal,No parity,1 stop,8 bits

```

下面这段 **samsung** 好象写的不太对,但是我按照 **Normal, No parity, 1 stop, 8 bits** 算出来的确是 **0x245**

```
// [10] [9] [8] [7] [6] [5] [4] [3:2] [1:0]

// Clock Sel, Tx Int, Rx Int, Rx Time Out, Rx err, Loop-back, Send break, Transmit Mode, Receive
Mode

// 0 1 0 , 0 1 0 0 , 01 01

// PCLK Level Pulse Disable Generate Normal Normal Interrupt or Polling

rUCON0 = 0x245; // Control register

rUBRDIV0=( (int)(PCLK/16./ baud) -1 ); //Baud rate divisor register 0

delay(10);

}
```

经过以上的折腾,接下来就是 **kernel** 的活了.能不能启动 **kernel**,得看你编译 **kernel** 的水平了.

这个 **BOOTLOADER** 不象 **blob** 那样需要交互信息,使用虚拟地址,总的来说非常简洁明了.