**QT210烧写UBOOT到SD卡原理以及UBOOT启动**

2014年12月19日 11:27:23 [mjl007](https://me.csdn.net/MJL007) 阅读数：566

世界早已进入cortex-a8了，我也得跟进一下所以买了QT210的开发板。

[长话短说](https://www.baidu.com/s?wd=%E9%95%BF%E8%AF%9D%E7%9F%AD%E8%AF%B4&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)开始搞SD卡烧写UBOOT，从SD启动UBOOT。

        从s5pv210\_irom\_applicationnote\_preliminary\_20091126.pdf知道，s5pv210启动分BL0、BL1、BL2阶段。BL0是s5pv210内部IROM固化的代码，这段代码根据OM引脚配置状态来选择从哪个外部存储设备加载BL1段代码（实际上BL1代码就是我们编写的UBOOT的前8K代码，这段代码要包含完整的将后半部UBOOT代码复制和清bss段的功能，当然我们要从SD卡启动烧写在上面的UBOOT，OM引脚就必须配置为从SD卡启动配置）。

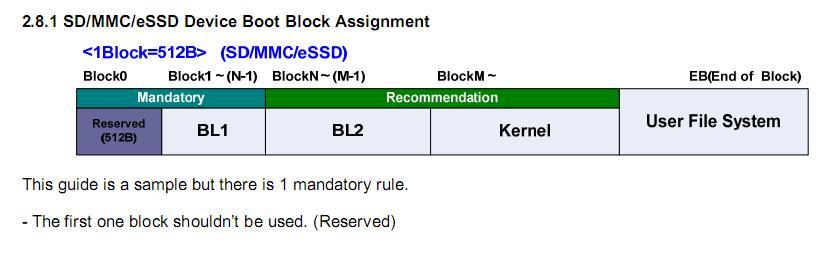


            图1

从上图可知，从sd启动的时候BL0加载的代码是从第512个字节处开始加载代码，为什么要这样做呢？由于以后功能扩展的需要[三星](https://www.baidu.com/s?wd=%E4%B8%89%E6%98%9F&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的软件工程师写的固化到IROM中的BL0代码是从SD卡的512字节处加载BL1的，他就是这样写的，我们对应UBOOT放置在SD卡中的位置就要往后移动512字节，后面有介绍怎么指定把uboot写到sd卡指定的位置的命令。

       还有一定要注意如下所示的地方：

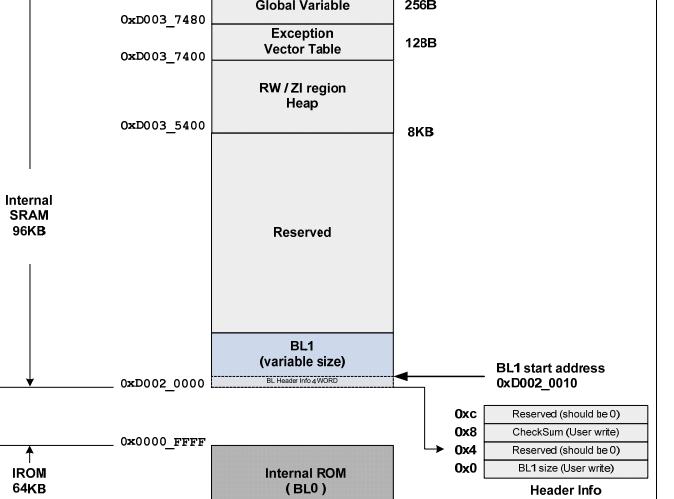


                                                                                          图2

在BL1之前要加16个字节的头部信息。也就是在真正的UBOOT第一条指令之前要加16个字节的头部信息，于是就有我们所看到的uboot代码如下的用宏定义的一段：

**[cpp]** [view plaincopy](http://blog.csdn.net/shushi0123/article/details/8018998)

1. #if defined(CONFIG\_EVT1) && !defined(CONFIG\_FUSED)
2. .word 0x2000
3. .word 0x0
4. .word 0x0
5. .word 0x0
6. #endif
8. .globl \_start
9. \_start: b   reset
10. ldr pc, \_undefined\_instruction
11. ldr pc, \_software\_interrupt
12. ldr pc, \_prefetch\_abort

其中的.word 0x2000  代表BL1size（8K长度），.word 0x0 为保留字节 .word 0x0为checksum（后续会通过一个mkbl1工具来计算bl1的checksum并填写这个位置），最后一个.word 0x0也为保留字节。

        再来看看uboot的下面的部分，如果bl0正常读取了bl1，代码就会到如下段：

**[cpp]** [view plaincopy](http://blog.csdn.net/shushi0123/article/details/8018998)

1. /\* Read booting information \*/
2. ldr r0, =PRO\_ID\_BASE
3. ldr r1, [r0,#OMR\_OFFSET]   //读OM引脚的配置状态
4. bic r2, r1, #0xffffffc1
6. #ifdef CONFIG\_VOGUES
7. /\* PS\_HOLD(GPH0\_0) set to output high \*/
8. ldr r0, =ELFIN\_GPIO\_BASE
9. ldr r1, =0x00000001
10. str r1, [r0, #GPH0CON\_OFFSET]
12. ldr r1, =0x5500
13. str r1, [r0, #GPH0PUD\_OFFSET]
15. ldr r1, =0x01
16. str r1, [r0, #GPH0DAT\_OFFSET]
17. #endif
19. /\* NAND BOOT \*/
20. cmp r2, #0x0        @ 512B 4-cycle
21. moveq   r3, #BOOT\_NAND          //根据OM引脚配置状态来给R3寄存器赋予代表系统是从何冲外部存储器启动的配置值。
23. cmp r2, #0x2        @ 2KB 5-cycle
24. moveq   r3, #BOOT\_NAND
26. cmp r2, #0x4        @ 4KB 5-cycle   8-bit ECC
27. moveq   r3, #BOOT\_NAND
29. cmp r2, #0x6        @ 4KB 5-cycle   16-bit ECC
30. moveq   r3, #BOOT\_NAND
32. cmp r2, #0x8        @ OneNAND Mux
33. moveq   r3, #BOOT\_ONENAND
35. /\* SD/MMC BOOT \*/
36. cmp     r2, #0xc<pre name="code" **class**="cpp">   ldr sp, \_TEXT\_PHY\_BASE  /\* setup temp stack pointer \*/
37. sub sp, sp, #12
38. mov fp, #0          /\* no previous frame, so fp=0 \*/
40. /\* when we already run in ram, we don't need to relocate U-Boot.
41. \* and actually, memory controller must be configured before U-Boot
42. \* is running in ram.
43. \*/
44. ldr r0, =0xff000fff
45. bic r1, pc, r0      /\* r0 <- current base addr of code \*/
46. ldr r2, \_TEXT\_BASE      /\* r1 <- original base addr in ram \*/
47. bic r2, r2, r0      /\* r0 <- current base addr of code \*/
48. cmp     r1, r2                  /\* compare r0, r1                  \*/
49. beq     after\_copy      /\* r0 == r1 then skip flash copy   \*/</pre><br>
50. moveq r3, #BOOT\_MMCSD/\* NOR BOOT \*/cmp r2, #0x14moveq r3, #BOOT\_NOR#**if** 0 /\* Android C110 BSP uses OneNAND booting! \*//\* For second device booting \*//\* OneNAND BOOTONG failed \*/cmp r2, #0x8moveq r3, #BOOT\_SEC\_DEV#endif/\* Uart BOOTONG failed \*/cmp r2, #(0x1<<4)moveq
51. r3, #BOOT\_SEC\_DEVldr r0, =INF\_REG\_BASEstr r3, [r0, #INF\_REG3\_OFFSET] //将配置值存入到用户使用的寄存器中，方便以后使用的时候读取此寄存器来了解此次启动时从何种外部存储设备启动
52. <pre></pre>
53. <p></p>
54. <pre></pre>
55. <pre name="code" **class**="cpp">   //ldr   sp, =0xd0036000 /\* end of sram dedicated to u-boot \*/
56. ldr sp, =0xd0035400  //BL1段的函数进行操作的堆栈位置，这里我修改了堆栈到了0xd0035400是由于图2中所示BL0代码进行了它自己代码堆栈的初始化，而RW/ZI region Heap的最低部就位于0xd0035400，为了不修改BL0堆栈和BL0代码已经写好的一些函数功能（会在下面用到），我将堆栈修改到了0xd0035400
57. sub sp, sp, #12 /\* set stack \*/
58. mov fp, #0
60. bl  lowlevel\_init   /\* go setup pll,mux,memory \*/ //这里会进行始终，内存，串口初始化</pre>
61. <p>之后运行到</p>
62. <p></p>
63. <pre name="code" **class**="cpp">   ldr sp, \_TEXT\_PHY\_BASE  /\* setup temp stack pointer \*/   //由于上面一步已经初始化了DRAM，所以在这里将以后的堆栈设置到了DRAM中的位置（以后BL2代码中函数都是基于此堆栈）
64. sub sp, sp, #12
65. mov fp, #0          /\* no previous frame, so fp=0 \*/
67. /\* when we already run in ram, we don't need to relocate U-Boot.
68. \* and actually, memory controller must be configured before U-Boot
69. \* is running in ram.
70. \*/
71. ldr r0, =0xff000fff
72. bic r1, pc, r0      /\* r0 <- current base addr of code \*/
73. ldr r2, \_TEXT\_BASE      /\* r1 <- original base addr in ram \*/  //加载程序要运行的编译地址，同现在程序运行的地址比较
74. bic r2, r2, r0      /\* r0 <- current base addr of code \*/
75. cmp     r1, r2                  /\* compare r0, r1                  \*/
76. beq     after\_copy      /\* r0 == r1 then skip flash copy   \*/  //如果两地址相等，代码程序已经再DRAM中运行不必再拷贝，不相等代表程序没有在编译地址运行，还要拷贝代码到编译地址去，最后跳到对应编译地址内运行代码。</pre><pre name="code" class="cpp">  ldr r0, =INF\_REG\_BASE
77. ldr r1, [r0, #INF\_REG3\_OFFSET]
78. cmp r1, #BOOT\_NAND      /\* 0x0 => boot device is nand \*/
79. beq nand\_boot
80. cmp r1, #BOOT\_ONENAND   /\* 0x1 => boot device is onenand \*/
81. beq onenand\_boot
82. cmp     r1, #BOOT\_MMCSD
83. beq     mmcsd\_boot
84. cmp     r1, #BOOT\_NOR
85. beq     nor\_boot
86. cmp     r1, #BOOT\_SEC\_DEV
87. beq     mmcsd\_boot      //代码判断要拷贝后，读取之前存入到用户使用寄存器中的值来判断此次启动从何种外部存储设备启动，这里为sd卡启动
88. ~~~~~省略若干代码~~~~~~~~
89. mmcsd\_boot:
90. #if DELETE
91. ldr     sp, \_TEXT\_PHY\_BASE
92. sub     sp, sp, #12
93. mov     fp, #0
94. #endif
95. bl      movi\_bl2\_copy   //最后BL1代码来到此处从sd卡拷贝剩余的代码
96. b       after\_copy</pre><br>
97. <pre></pre>
98. <pre name="code" **class**="cpp"></pre>