[2014.4新版uboot启动流程分析](http://blog.csdn.net/skyflying2012/article/details/25804209)

分类： [bootloader](http://blog.csdn.net/skyflying2012/article/category/2264565)2014-05-14 16:10 4615人阅读 [评论](http://blog.csdn.net/skyflying2012/article/details/25804209#comments)(11) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/skyflying2012/article/details/25804209#report)

[新版uboot](http://www.csdn.net/tag/%e6%96%b0%e7%89%88uboot)[relocate](http://www.csdn.net/tag/relocate)[启动](http://www.csdn.net/tag/%e5%90%af%e5%8a%a8)[initialization](http://www.csdn.net/tag/initialization)

最近开始接触uboot，现在需要将2014.4版本uboot移植到公司armv7开发板。

**在网上搜索讲uboot启动过程的文章，大多都是比较老版本的uboot，于是决定将新版uboot启动过程记录下来，和大家共享。**

**辛苦之作，大家共享，转载还请注明出处！**

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. #
2. # (C) Copyright 2000-2013
3. # Wolfgang Denk, DENX Software Engineering, wd@denx.de.
4. #
5. # SPDX-License-Identifier:  GPL-2.0+
6. #
8. VERSION = 2014
9. PATCHLEVEL = 04
10. SUBLEVEL =
11. EXTRAVERSION =
12. NAME =

到我写这篇文章之时，这个版本的uboot是最新版本。

2014.4版本uboot启动至命令行几个重要函数为：**\_start，\_main，board\_init\_f，relocate\_code，board\_init\_r**。

**一 \_start**

对于任何程序，入口函数是在链接时决定的，uboot的入口是由链接脚本决定的。uboot下armv7链接脚本默认目录为arch/arm/cpu/u-boot.lds。这个可以在配置文件中与CONFIG\_SYS\_LDSCRIPT来指定。

入口地址也是由连接器决定的，在配置文件中可以由CONFIG\_SYS\_TEXT\_BASE指定。这个会在编译时加在ld连接器的选项-Ttext中

uboot的配置编译原理也非常值得学习，我想在另外写一篇文章来记录，这里不详细说了。

查看u-boot.lds

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">OUTPUT\_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
2. OUTPUT\_ARCH(arm)
3. ENTRY(\_start)
4. SECTIONS
5. {
6. . = 0x00000000;
8. . = ALIGN(4);
9. .text :
10. {
11. \*(.\_\_image\_copy\_start)
12. CPUDIR/start.o (.text\*)
13. \*(.text\*)
14. }
16. . = ALIGN(4);
17. .rodata : { \*(SORT\_BY\_ALIGNMENT(SORT\_BY\_NAME(.rodata\*))) }
19. . = ALIGN(4);
20. .data : {
21. \*(.data\*)
22. }
23. </span>

链接脚本中这些宏的定义在linkage.h中，看字面意思也明白，程序的入口是在\_start.，后面是text段，data段等。

\_start在arch/arm/cpu/armv7/start.S中，一段一段的分析，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">.globl \_start
2. \_start: b   reset
3. ldr pc, \_undefined\_instruction
4. ldr pc, \_software\_interrupt
5. ldr pc, \_prefetch\_abort
6. ldr pc, \_data\_abort
7. ldr pc, \_not\_used
8. ldr pc, \_irq
9. ldr pc, \_fiq
10. #ifdef CONFIG\_SPL\_BUILD
11. \_undefined\_instruction: .word \_undefined\_instruction
12. \_software\_interrupt:    .word \_software\_interrupt
13. \_prefetch\_abort:    .word \_prefetch\_abort
14. \_data\_abort:        .word \_data\_abort
15. \_not\_used:      .word \_not\_used
16. \_irq:           .word \_irq
17. \_fiq:           .word \_fiq
18. \_pad:           .word 0x12345678 /\* now 16\*4=64 \*/
19. #else
20. .globl \_undefined\_instruction
21. \_undefined\_instruction: .word undefined\_instruction
22. .globl \_software\_interrupt
23. \_software\_interrupt:    .word software\_interrupt
24. .globl \_prefetch\_abort
25. \_prefetch\_abort:    .word prefetch\_abort
26. .globl \_data\_abort
27. \_data\_abort:        .word data\_abort
28. .globl \_not\_used
29. \_not\_used:      .word not\_used
30. .globl \_irq
31. \_irq:           .word irq
32. .globl \_fiq
33. \_fiq:           .word fiq
34. \_pad:           .word 0x12345678 /\* now 16\*4=64 \*/</span>

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#endif  /\* CONFIG\_SPL\_BUILD \*/
3. .global \_end\_vect
4. \_end\_vect:
6. .balignl 16,0xdeadbeef
7. </span>

.global声明\_start为全局符号，\_start就会被连接器链接到，也就是链接脚本中的入口地址了。

以上代码是设置arm的异常向量表，arm异常向量表如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 地址 | 异常 | 进入模式 | 描述 |
| 0x00000000 | 复位 | 管理模式 | 复位电平有效时，产生复位异常，程序跳转到复位处理程序处执行 |
| 0x00000004 | 未定义指令 | 未定义模式 | 遇到不能处理的指令时，产生未定义指令异常 |
| 0x00000008 | 软件中断 | 管理模式 | 执行SWI指令产生，用于用户模式下的程序调用特权操作指令 |
| 0x0000000c | 预存指令 | 中止模式 | 处理器预取指令的地址不存在，或该地址不允许当前指令访问，产生指令预取中止异常 |
| 0x00000010 | 数据操作 | 中止模式 | 处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常 |
| 0x00000014 | 未使用 | 未使用 | 未使用 |
| 0x00000018 | IRQ | IRQ | 外部中断请求有效，且CPSR中的I位为0时，产生IRQ异常 |
| 0x0000001c | FIQ | FIQ | 快速中断请求引脚有效，且CPSR中的F位为0时，产生FIQ异常 |

8种异常分别占用4个字节，因此每种异常入口处都填写一条跳转指令，直接跳转到相应的异常处理函数中，reset异常是直接跳转到reset函数，其他7种异常是用ldr将处理函数入口地址加载到pc中。

后面汇编是定义了7种异常的入口函数，这里没有定义CONFIG\_SPL\_BUILD，所以走后面一个。

接下来定义的\_end\_vect中用.balignl来指定接下来的代码要16字节对齐，空缺的用0xdeadbeef，方便更加高效的访问内存。接着分析下面一段代码

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#ifdef CONFIG\_USE\_IRQ
2. /\* IRQ stack memory (calculated at run-time) \*/
3. .globl IRQ\_STACK\_START
4. IRQ\_STACK\_START:
5. .word   0x0badc0de
7. /\* IRQ stack memory (calculated at run-time) \*/
8. .globl FIQ\_STACK\_START
9. FIQ\_STACK\_START:
10. .word 0x0badc0de
11. #endif
13. /\* IRQ stack memory (calculated at run-time) + 8 bytes \*/
14. .globl IRQ\_STACK\_START\_IN
15. IRQ\_STACK\_START\_IN:
16. .word   0x0badc0de
17. </span>

如果uboot中使用中断，这里声明中断处理函数栈起始地址，这里给出的值是0x0badc0de，是一个非法值，注释也说明了，这个值会在运行时重新计算，我查找了一下代码是在interrupt\_init中。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">reset:
2. bl  save\_boot\_params
3. /\*
4. \* disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,
5. \* except if in HYP mode already
6. \*/
7. mrs r0, cpsr
8. and r1, r0, #0x1f       @ mask mode bits
9. teq r1, #0x1a       @ test **for** HYP mode
10. bicne   r0, r0, #0x1f       @ clear all mode bits
11. orrne   r0, r0, #0x13       @ set SVC mode
12. orr r0, r0, #0xc0       @ disable FIQ and IRQ
13. msr cpsr,r0
14. </span>

在上电或者重启后，处理器取得第一条指令就是b reset，所以会直接跳转到reset函数处。reset首先是跳转到save\_boot\_params中，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
2. \*
3. \* void save\_boot\_params(u32 r0, u32 r1, u32 r2, u32 r3)
4. \*  \_\_attribute\_\_((weak));
5. \*
6. \* Stack pointer is not yet initialized at this moment
7. \* Don't save anything to stack even if compiled with -O0
8. \*
9. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
10. ENTRY(save\_boot\_params)
11. bx  lr          @ back to my caller
12. ENDPROC(save\_boot\_params)
13. .weak   save\_boot\_params
14. </span>

这里save\_boot\_params函数中没做什么直接跳回，注释也说明了，栈没有初始化，最好不要再函数中做操作。

这里值得注意的是.weak关键字，在网上找了到的解释，我的理解是.weak相当于声明一个函数，如果该函数在其他地方没有定义，则为空函数，有定义则调用该定义的函数。

具体解释可以看这位大神的详解：

http://blog.csdn.net/norains/article/details/5954459

接下来reset执行7条指令，修改cpsr寄存器，设置处理器进入svc模式，并且关掉irq和fiq。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">/\*
2. \* Setup vector:
3. \* (OMAP4 spl TEXT\_BASE is not 32 byte aligned.
4. \* Continue to use ROM code vector only in OMAP4 spl)
5. \*/
6. #if !(defined(CONFIG\_OMAP44XX) && defined(CONFIG\_SPL\_BUILD))
7. /\* Set V=0 in CP15 SCTRL register - for VBAR to point to vector \*/
8. mrc p15, 0, r0, c1, c0, 0   @ Read CP15 SCTRL Register
9. bic r0, #CR\_V       @ V = 0
10. mcr p15, 0, r0, c1, c0, 0   @ Write CP15 SCTRL Register
12. /\* Set vector address in CP15 VBAR register \*/
13. ldr r0, =\_start
14. mcr p15, 0, r0, c12, c0, 0  @Set VBAR
15. #endif
17. /\* the mask ROM code should have PLL and others stable \*/
18. #ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT
19. bl  cpu\_init\_cp15
20. bl  cpu\_init\_crit
21. #endif
23. bl  \_main
24. </span>

前面6条汇编指令是对协处理器cp15进行操作，设置了处理器的异常向量入口地址为\_start，

这里需要注意，ARM默认的异常向量表入口在0x0地址，uboot的运行介质（norflash nandflash sram等）映射地址可能不在0x0起始的地址，所以需要修改异常向量表入口。

但是我在网上没有找到cp15协处理器的c12寄存器的说明，可能是armv7新添加的

协处理器cp15的说明可以看我转的一篇文章：

http://blog.csdn.net/skyflying2012/article/details/25823967

接下来如果没有定义宏CONFIG\_SKIP\_LOWLEVEL\_INIT，则会分别跳转执行cpu\_init\_cp15以及cpu\_init\_crit。

在分析这2个函数之前先总结一下上面分析的这一段\_start中汇编的作用：

1 初始化异常向量表   2 设置cpu svc模式，关中断    3 配置cp15，设置异常向量入口

都是跟异常有关的部分。

接下来先分析cpu\_init\_cp15

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
2. \*
3. \* cpu\_init\_cp15
4. \*
5. \* Setup CP15 registers (cache, MMU, TLBs). The I-cache is turned on unless
6. \* CONFIG\_SYS\_ICACHE\_OFF is defined.
7. \*
8. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
9. ENTRY(cpu\_init\_cp15)
10. /\*
11. \* Invalidate L1 I/D
12. \*/
13. mov r0, #0          @ set up **for** MCR
14. mcr p15, 0, r0, c8, c7, 0   @ invalidate TLBs
15. mcr p15, 0, r0, c7, c5, 0   @ invalidate icache
16. mcr p15, 0, r0, c7, c5, 6   @ invalidate BP array
17. mcr     p15, 0, r0, c7, c10, 4  @ DSB
18. mcr     p15, 0, r0, c7, c5, 4   @ ISB
20. /\*
21. \* disable MMU stuff and caches
22. \*/
23. mrc p15, 0, r0, c1, c0, 0
24. bic r0, r0, #0x00002000 @ clear bits 13 (--V-)
25. bic r0, r0, #0x00000007 @ clear bits 2:0 (-CAM)
26. orr r0, r0, #0x00000002 @ set bit 1 (--A-) Align
27. orr r0, r0, #0x00000800 @ set bit 11 (Z---) BTB
28. #ifdef CONFIG\_SYS\_ICACHE\_OFF
29. bic r0, r0, #0x00001000 @ clear bit 12 (I) I-cache
30. #else
31. orr r0, r0, #0x00001000 @ set bit 12 (I) I-cache
32. #endif
33. mcr p15, 0, r0, c1, c0, 0
34. </span>

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#ifdef CONFIG\_ARM\_ERRATA\_716044
2. mrc p15, 0, r0, c1, c0, 0   @ read system control **register**
3. orr r0, r0, #1 << 11    @ set bit #11
4. mcr p15, 0, r0, c1, c0, 0   @ write system control **register**
5. #endif
7. #if (defined(CONFIG\_ARM\_ERRATA\_742230) || defined(CONFIG\_ARM\_ERRATA\_794072))
8. mrc p15, 0, r0, c15, c0, 1  @ read diagnostic **register**
9. orr r0, r0, #1 << 4     @ set bit #4
10. mcr p15, 0, r0, c15, c0, 1  @ write diagnostic **register**
11. #endif
13. #ifdef CONFIG\_ARM\_ERRATA\_743622
14. mrc p15, 0, r0, c15, c0, 1  @ read diagnostic **register**
15. orr r0, r0, #1 << 6     @ set bit #6
16. mcr p15, 0, r0, c15, c0, 1  @ write diagnostic **register**
17. #endif
19. #ifdef CONFIG\_ARM\_ERRATA\_751472
20. mrc p15, 0, r0, c15, c0, 1  @ read diagnostic **register**
21. orr r0, r0, #1 << 11    @ set bit #11
22. mcr p15, 0, r0, c15, c0, 1  @ write diagnostic **register**
23. #endif
24. #ifdef CONFIG\_ARM\_ERRATA\_761320
25. mrc p15, 0, r0, c15, c0, 1  @ read diagnostic **register**
26. orr r0, r0, #1 << 21    @ set bit #21
27. mcr p15, 0, r0, c15, c0, 1  @ write diagnostic **register**
28. #endif
30. mov pc, lr          @ back to my caller
31. ENDPROC(cpu\_init\_cp15)
32. </span>

cpu\_init\_cp15函数是配置cp15协处理器相关寄存器来设置处理器的MMU，cache以及tlb。如果没有定义CONFIG\_SYS\_ICACHE\_OFF则会打开icache。关掉mmu以及tlb。  
具体配置过程可以对照cp15寄存器来看，这里不详细说了

接下来看cpu\_init\_crit

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
2. \*
3. \* CPU\_init\_critical registers
4. \*
5. \* setup important registers
6. \* setup memory timing
7. \*
8. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
9. ENTRY(cpu\_init\_crit)
10. /\*
11. \* Jump to board specific initialization...
12. \* The Mask ROM will have already initialized
13. \* basic memory. Go here to bump up clock rate and handle
14. \* wake up conditions.
15. \*/
16. b   lowlevel\_init       @ go setup pll,mux,memory
17. ENDPROC(cpu\_init\_crit)
18. </span>

看注释可以明白，cpu\_init\_crit调用的lowlevel\_init函数是与特定开发板相关的初始化函数，在这个函数里会做一些pll初始化，如果不是从mem启动，则会做memory初始化，方便后续拷贝到mem中运行。

lowlevel\_init函数则是需要移植来实现，做clk初始化以及ddr初始化

从cpu\_init\_crit返回后，\_start的工作就完成了，接下来就要调用\_main，总结一下\_start工作：

1 前面总结过的部分，初始化异常向量表，设置svc模式，关中断

2 配置cp15，初始化mmu cache tlb

3 板级初始化，pll memory初始化

**二 \_main**

\_main函数在arch/arm/lib/crt0.S中，mian函数的作用在注释中有详细的说明，我们分段来分析一下

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. ENTRY(\_main)
3. /\*
4. \* Set up initial C runtime environment and call board\_init\_f(0).
5. \*/
7. #if defined(CONFIG\_SPL\_BUILD) && defined(CONFIG\_SPL\_STACK)
8. ldr sp, =(CONFIG\_SPL\_STACK)
9. #else
10. ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)
11. #endif
12. bic sp, sp, #7  /\* 8-byte alignment for ABI compliance \*/
13. sub sp, sp, #GD\_SIZE    /\* allocate one GD above SP \*/
14. bic sp, sp, #7  /\* 8-byte alignment for ABI compliance \*/
15. mov r9, sp      /\* GD is above SP \*/
16. mov r0, #0
17. bl  board\_init\_f

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-family: Arial, Helvetica, sans-serif; background-color: rgb(255, 255, 255);">首先</span><span style="font-family: Arial, Helvetica, sans-serif; background-color: rgb(255, 255, 255);">将CONFIG\_SYS\_INIT\_SP\_ADDR定义的值加载到栈指针sp中，这个宏定义在配置头文件中指定。</span>

这段代码是为board\_init\_f C函数调用提供环境，也就是栈指针sp初始化

8字节对齐，然后减掉GD\_SIZE,这个宏定义是指的全局结构体gd的大小，是160字节在此处，这个结构体用来保存uboot一些全局信息，需要一块单独的内存。

最后将sp保存在r9寄存器中。因此r9寄存器中的地址就是gd结构体的首地址。

在后面所有code中如果要使用gd结构体，必须在文件中加入DECLARE\_GLOBAL\_DATA\_PTR宏定义，定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. #define DECLARE\_GLOBAL\_DATA\_PTR     register volatile gd\_t \*gd asm ("r9")

gd结构体首地址就是r9中的值。

C语言函数栈是向下生长，这里sp为malloc空间顶端减去gd bd空间开始的，起初很纳闷，sp设在这里，以后的C函数调用不都会在malloc空间了吗，堆栈空间不就重合了嘛，不用急，看完board\_init\_f就明白了。

接着说\_main上面一段代码，接着r0赋为0，也就是参数0为0，调用board\_init\_f

**三 board\_init\_f**

移植uboot先做一个最精简版本，很多配置选项都没有打开，比如fb mmc等硬件都默认不打开，只配置基本的ddr serial，这样先保证uboot能正常启动进入命令行，然后再去添加其他。

我们这里分析就是按最精简版本来，这样可以更加简洁的说明uboot的启动流程。

board\_init\_f函数主要是根据配置对全局信息结构体gd进行初始化。

gd结构体中有个别成员意义我也不是很理解，这里我只说我理解并且在后面起到作用的成员。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. gd->mon\_len = (ulong)&\_\_bss\_end - (ulong)\_start;

初始化mon\_len，代表uboot code的大小。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. **for** (init\_fnc\_ptr = init\_sequence; \*init\_fnc\_ptr; ++init\_fnc\_ptr) {
2. **if** ((\*init\_fnc\_ptr)() != 0) {
3. hang ();
4. }
5. }

遍历调用init\_sequence所有函数，init\_sequence定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. init\_fnc\_t \*init\_sequence[] = {
2. arch\_cpu\_init,      /\* basic arch cpu dependent setup \*/
3. mark\_bootstage,
4. #ifdef CONFIG\_OF\_CONTROL
5. fdtdec\_check\_fdt,
6. #endif
7. #if defined(CONFIG\_BOARD\_EARLY\_INIT\_F)
8. board\_early\_init\_f,
9. #endif
10. timer\_init,     /\* initialize timer \*/
11. #ifdef CONFIG\_BOARD\_POSTCLK\_INIT
12. board\_postclk\_init,
13. #endif
14. #ifdef CONFIG\_FSL\_ESDHC
15. get\_clocks,
16. #endif
17. env\_init,       /\* initialize environment \*/
18. init\_baudrate,      /\* initialze baudrate settings \*/
19. serial\_init,        /\* serial communications setup \*/
20. console\_init\_f,     /\* stage 1 init of console \*/
21. display\_banner,     /\* say that we are here \*/
22. print\_cpuinfo,      /\* display cpu info (and speed) \*/
23. #if defined(CONFIG\_DISPLAY\_BOARDINFO)
24. checkboard,     /\* display board info \*/
25. #endif
26. #if defined(CONFIG\_HARD\_I2C) || defined(CONFIG\_SYS\_I2C)
27. init\_func\_i2c,
28. #endif
29. dram\_init,      /\* configure available RAM banks \*/
30. NULL,
31. };

arch\_cpu\_init需要实现，要先启动uboot这里可以先写一个空函数。

timer\_init在lib/time.c中有实现，也是空函数，但是有\_\_WEAK关键字，如果自己实现，则会调用自己实现的这个函数

对最精简uboot，需要做好就是ddr和serial，所以我们最关心是serial\_init,console\_init\_f以及dram\_init.

先看serial\_init

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**int** serial\_init(**void**)
2. {
3. **return** get\_current()->start();
4. }
5. </span>

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**static** **struct** serial\_device \*get\_current(**void**)
2. {
3. **struct** serial\_device \*dev;
5. **if** (!(gd->flags & GD\_FLG\_RELOC))
6. dev = default\_serial\_console();
7. **else** **if** (!serial\_current)
8. dev = default\_serial\_console();
9. **else**
10. dev = serial\_current;
12. /\* We must have a console device \*/
13. **if** (!dev) {
14. #ifdef CONFIG\_SPL\_BUILD
15. puts("Cannot find console\n");
16. hang();
17. #else
18. panic("Cannot find console\n");
19. #endif
20. }
22. **return** dev;
23. }
24. </span>

gd->flags还没做初始化，serial\_current用来存放我们当前要使用的serial，这里也还没做初始化，所以最终serial\_device就是default\_serial\_console(),这个在serial驱动中有实现，来返回一个默认的调试串口。

serial\_device结构体代表了一个串口设备，其中的成员都需要在自己的serial驱动中实现。

这样在serial\_init中get\_current获取就是串口驱动中给出的默认调试串口结构体，执行start，做一些特定串口初始化。

console\_init\_f将gd中have\_console置1，这个函数不详细说了。

display\_banner,print\_cpuinfo利用现在的调试串口打印了uboot的信息。

接下来就是dram\_init。

dram\_init对gd->ram\_size初始化，以便board\_init\_f后面代码对dram空间进行规划。

dram\_init实现可以通过配置文件定义宏定义来实现，也可以通过对ddrc控制器读获取dram信息。

继续分析board\_init\_f，剩余代码将会对sdram空间进行规划！

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#**if** defined(CONFIG\_SYS\_MEM\_TOP\_HIDE)
2. /\*
3. \* Subtract specified amount of memory to hide so that it won't
4. \* get "touched" at all by U-Boot. By fixing up gd->ram\_size
5. \* the Linux kernel should now get passed the now "corrected"
6. \* memory size and won't touch it either. This should work
7. \* for arch/ppc and arch/powerpc. Only Linux board ports in
8. \* arch/powerpc with bootwrapper support, that recalculate the
9. \* memory size from the SDRAM controller setup will have to
10. \* get fixed.
11. \*/
12. gd->ram\_size -= CONFIG\_SYS\_MEM\_TOP\_HIDE;
13. #endif
15. addr = CONFIG\_SYS\_SDRAM\_BASE + get\_effective\_memsize();
16. </span>

CONFIG\_SYS\_MEM\_TOP\_HIDE宏定义是将一部分内存空间隐藏，注释说明对于ppc处理器在内核中有接口来实现使用uboot提供的值，这里咱们不考虑。

addr的值由CONFIG\_SYS\_SDRAM\_BASE加上ram\_size。也就是到了可用sdram的顶端。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#**if** !(defined(CONFIG\_SYS\_ICACHE\_OFF) && defined(CONFIG\_SYS\_DCACHE\_OFF))
2. /\* reserve TLB table \*/
3. gd->arch.tlb\_size = PGTABLE\_SIZE;
4. addr -= gd->arch.tlb\_size;
6. /\* round down to next 64 kB limit \*/
7. addr &= ~(0x10000 - 1);
9. gd->arch.tlb\_addr = addr;
10. debug("TLB table from %08lx to %08lx\n", addr, addr + gd->arch.tlb\_size);
11. #endif
13. /\* round down to next 4 kB limit \*/
14. addr &= ~(4096 - 1);
15. debug("Top of RAM usable for U-Boot at: %08lx\n", addr);
16. </span>

如果打开了icache以及dcache，则预留出PATABLE\_SIZE大小的tlb空间，tlb存放首地址赋值给gd->arch.tlb\_addr。

最后addr此时值就是tlb的地址，4kB对齐。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#ifdef CONFIG\_LCD
2. #ifdef CONFIG\_FB\_ADDR
3. gd->fb\_base = CONFIG\_FB\_ADDR;
4. #else
5. /\* reserve memory for LCD display (always full pages) \*/
6. addr = lcd\_setmem(addr);
7. gd->fb\_base = addr;
8. #endif /\* CONFIG\_FB\_ADDR \*/
9. #endif /\* CONFIG\_LCD \*/
11. /\*
12. \* reserve memory for U-Boot code, data & bss
13. \* round down to next 4 kB limit
14. \*/
15. addr -= gd->mon\_len;
16. addr &= ~(4096 - 1);
18. debug("Reserving %ldk for U-Boot at: %08lx\n", gd->mon\_len >> 10, addr);
19. </span>

如果需要使用frambuffer，使用配置fb首地址CONFIG\_FB\_ADDR或者调用lcd\_setmem获取fb大小，这里面有板级相关函数需要实现，不过为了先能启动uboot，没有打开fb选项。addr值就是fb首地址。

gd->fb\_base保存fb首地址。

接着-gd->mon\_len为uboot的code留出空间，到这里addr的值就确定，addr作为uboot relocate的目标addr。

到这里，可以看出uboot现在空间划分是从顶端往下进行的。

先总结一下addr之上sdram空间的划分：

由高到低 ： top-->hide mem-->tlb space(16K)-->framebuffer space-->uboot code space-->addr

接下来要确定addr\_sp的值。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#ifndef CONFIG\_SPL\_BUILD
2. /\*
3. \* reserve memory for malloc() arena
4. \*/
5. addr\_sp = addr - TOTAL\_MALLOC\_LEN;
6. debug("Reserving %dk for malloc() at: %08lx\n",
7. TOTAL\_MALLOC\_LEN >> 10, addr\_sp);
8. /\*
9. \* (permanently) allocate a Board Info struct
10. \* and a permanent copy of the "global" data
11. \*/
12. addr\_sp -= **sizeof** (bd\_t);
13. bd = (bd\_t \*) addr\_sp;
14. gd->bd = bd;
15. debug("Reserving %zu Bytes for Board Info at: %08lx\n",
16. **sizeof** (bd\_t), addr\_sp);
17. </span>

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#ifdef CONFIG\_MACH\_TYPE
2. gd->bd->bi\_arch\_number = CONFIG\_MACH\_TYPE; /\* board id for Linux \*/
3. #endif
5. addr\_sp -= **sizeof** (gd\_t);
6. id = (gd\_t \*) addr\_sp;
7. debug("Reserving %zu Bytes for Global Data at: %08lx\n",
8. **sizeof** (gd\_t), addr\_sp);</span>

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#ifndef CONFIG\_ARM64
2. /\* setup stackpointer for exeptions \*/
3. gd->irq\_sp = addr\_sp;
4. #ifdef CONFIG\_USE\_IRQ
5. addr\_sp -= (CONFIG\_STACKSIZE\_IRQ+CONFIG\_STACKSIZE\_FIQ);
6. debug("Reserving %zu Bytes for IRQ stack at: %08lx\n",
7. CONFIG\_STACKSIZE\_IRQ+CONFIG\_STACKSIZE\_FIQ, addr\_sp);
8. #endif
9. /\* leave 3 words for abort-stack    \*/
10. addr\_sp -= 12;
12. /\* 8-byte alignment for ABI compliance \*/
13. addr\_sp &= ~0x07;
14. #else   /\* CONFIG\_ARM64 \*/
15. /\* 16-byte alignment for ABI compliance \*/
16. addr\_sp &= ~0x0f;
17. #endif  /\* CONFIG\_ARM64 \*/</span>

首先预留malloc len，这里我定义的是0x400000.

注释中说明，为bd，gd做一个永久的copy。

留出了全局信息bd\_t结构体的空间，首地址存在gd->bd。

留出gd\_t结构体的空间。首地址存在id中。  
将此地址保存在gd->irq\_sp中作为异常栈指针。uboot中我们没有用到中断。

最后留出12字节，for abort stack，这个没看懂。

到这里addr\_sp值确定，总结一下addr\_sp之上空间分配。

由高到低 ： addr-->malloc len(0x400000)-->bd len-->gd len-->12 byte-->addr\_sp（栈往下增长，addr\_sp之下空间作为栈空间）

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">    gd->bd->bi\_baudrate = gd->baudrate;
2. /\* Ram ist board specific, so move it to board code ... \*/
3. dram\_init\_banksize();
4. display\_dram\_config();  /\* and display it \*/
6. gd->relocaddr = addr;
7. gd->start\_addr\_sp = addr\_sp;
8. gd->reloc\_off = addr - (ulong)&\_start;
9. debug("relocation Offset is: %08lx\n", gd->reloc\_off);
10. **if** (new\_fdt) {
11. memcpy(new\_fdt, gd->fdt\_blob, fdt\_size);
12. gd->fdt\_blob = new\_fdt;
13. }
14. memcpy(id, (**void** \*)gd, **sizeof**(gd\_t));
15. </span>

给bd->bi\_baudrate赋值gd->baudrate，gd->baudrate是在前面baudrate\_init中初始化。

dram\_init\_banksize()是需要实现的板级函数。根据板上ddrc获取ddr的bank信息。填充在gd->bd->bi\_dram[CONFIG\_NR\_DRAM\_BANKS]。

gd->relocaaddr为目标addr，gd->start\_addr\_sp为目标addr\_sp,gd->reloc\_off为目标addr和现在实际code起始地址的偏移。reloc\_off非常重要，会作为后面relocate\_code函数的参数，来实现code的拷贝。

最后将gd结构体的数据拷贝到新的地址id上。

board\_init\_f函数将sdram空间重新进行了划分，可以看出栈空间和堆空间是分开的，就不存在\_main调用board\_init\_f之前的那个问题啦。

并且在重新规划空间完成之前并没有出现初始化堆，以及使用堆空间的问题，比如malloc函数，所以之前的堆栈空间重合的问题是过虑了。

至此，board\_init\_f结束，回到\_main

**四 \_main**

board\_init\_f结束后，代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#**if** ! defined(CONFIG\_SPL\_BUILD)
3. /\*
4. \* Set up intermediate environment (new sp and gd) and call
5. \* relocate\_code(addr\_moni). Trick here is that we'll return
6. \* 'here' but relocated.
7. \*/
9. ldr sp, [r9, #GD\_START\_ADDR\_SP] /\* sp = gd->start\_addr\_sp \*/
10. bic sp, sp, #7  /\* 8-byte alignment for ABI compliance \*/
11. ldr r9, [r9, #GD\_BD]        /\* r9 = gd->bd \*/
12. sub r9, r9, #GD\_SIZE        /\* new GD is below bd \*/
14. adr lr, here
15. ldr r0, [r9, #GD\_RELOC\_OFF]     /\* r0 = gd->reloc\_off \*/
16. add lr, lr, r0
17. ldr r0, [r9, #GD\_RELOCADDR]     /\* r0 = gd->relocaddr \*/
18. b   relocate\_code
19. here:
21. </span>

这段汇编很有意思，前4条汇编实现了新gd结构体的更新。

首先更新sp，并且将sp 8字节对齐，方便后面函数开辟栈能对齐，

然后获取gd->bd地址到r9中，需要注意，在board\_init\_f中gd->bd已经更新为新分配的bd了，下一条汇编将r9减掉bd的size，这样就获取到了board\_init\_f中新分配的gd了！

后面汇编则是为relocate\_code做准备，首先加载here地址，然后加上新地址偏移量给lr，则是code relocate后的新here了，relocate\_code返回条转到lr，则是新位置的here！

最后在r0中保存code的新地址，跳转到relocate\_code

**五 relocate\_code**

relocate\_code函数在arch/arm/lib/relocate.S中，这个函数实现了将uboot code拷贝到relocaddr。

这部分算是整个uboot中最核心也是最难理解的代码，我单独写了一篇文章来介绍这一部分的工作原理，感兴趣的朋友可以看下面这个链接

http://blog.csdn.net/skyflying2012/article/details/37660265

这里就不再详说了。

到这里需要总结一下，经过上面的分析可以看出，

新版uboot在sdram空间分配上，是自顶向下，

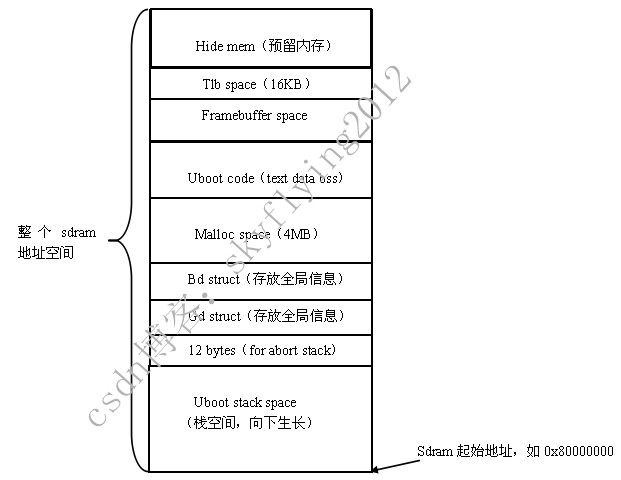
不管uboot是从哪里启动，spiflash，nandflash，sram等跑到这里code都会被从新定位到sdram上部的一个位置，继续运行。

我找了一个2010.6版本的uboot大体看了一下启动代码，是通过判断\_start和TEXT\_BASE（链接地址）是否相等来确定是否需要relocate。如果uboot是从sdram启动则不需要relocate。

新版uboot在这方面还是有较大变动。

这样变动我考虑好处可能有二，一是不用考虑启动方式，all relocate code。二是不用考虑uboot链接地址，因为都要重新relocate。

uboot sdram空间规划图：



**六 \_main**

从relocate\_code回到\_main中，接下来是main最后一段代码，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">/\* Set up final (full) environment \*/
3. bl  c\_runtime\_cpu\_setup /\* we still call old routine here \*/
5. ldr r0, =\_\_bss\_start    /\* this is auto-relocated! \*/
6. ldr r1, =\_\_bss\_end      /\* this is auto-relocated! \*/
8. mov r2, #0x00000000     /\* prepare zero to clear BSS \*/
10. clbss\_l:cmp r0, r1          /\* while not at end of BSS \*/
11. strlo   r2, [r0]        /\* clear 32-bit BSS word \*/
12. addlo   r0, r0, #4      /\* move to next \*/
13. blo clbss\_l
15. bl coloured\_LED\_init
16. bl red\_led\_on
18. /\* call board\_init\_r(gd\_t \*id, ulong dest\_addr) \*/
19. mov     r0, r9                  /\* gd\_t \*/
20. ldr r1, [r9, #GD\_RELOCADDR] /\* dest\_addr \*/
21. /\* call board\_init\_r \*/
22. ldr pc, =board\_init\_r   /\* this is auto-relocated! \*/
24. /\* we should not return here. \*/
26. </span>

首先跳转到c\_runtime\_cpu\_setup，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">ENTRY(c\_runtime\_cpu\_setup)
2. /\*
3. \* If I-cache is enabled invalidate it
4. \*/
5. #ifndef CONFIG\_SYS\_ICACHE\_OFF
6. mcr p15, 0, r0, c7, c5, 0   @ invalidate icache
7. mcr     p15, 0, r0, c7, c10, 4  @ DSB
8. mcr     p15, 0, r0, c7, c5, 4   @ ISB
9. #endif
10. /\*
11. \* Move vector table
12. \*/
13. /\* Set vector address in CP15 VBAR register \*/
14. ldr     r0, =\_start
15. mcr     p15, 0, r0, c12, c0, 0  @Set VBAR
17. bx  lr
19. ENDPROC(c\_runtime\_cpu\_setup)
20. </span>

如果icache是enable，则无效掉icache，保证从sdram中更新指令到cache中。

接着更新异常向量表首地址，因为code被relocate，所以异常向量表也被relocate。

从c\_runtime\_cpu\_setup返回，下面一段汇编是将bss段清空。

接下来分别调用了coloured\_LED\_init以及red\_led\_on,很多开发板都会有led指示灯，这里可以实现上电指示灯亮，有调试作用。

最后r0赋值gd指针，r1赋值relocaddr，进入最后的board\_init\_r !

**七 board\_init\_r**

参数1是新gd指针，参数2是relocate addr，也就是新code地址

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">    gd->flags |= GD\_FLG\_RELOC;  /\* tell others: relocation done \*/
2. bootstage\_mark\_name(BOOTSTAGE\_ID\_START\_UBOOT\_R, "board\_init\_r");
4. monitor\_flash\_len = (ulong)&\_\_rel\_dyn\_end - (ulong)\_start;
6. /\* Enable caches \*/
7. enable\_caches();
9. debug("monitor flash len: %08lX\n", monitor\_flash\_len);
10. board\_init();   /\* Setup chipselects \*/
11. </span>

置位gd->flags，标志已经relocate。monitor\_flash\_len这个变量作用没看懂。使能cache，最后board\_init是需要实现的板级支持函数。做开发板的基本初始化。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">#ifdef CONFIG\_CLOCKS
2. set\_cpu\_clk\_info(); /\* Setup clock information \*/
3. #endif
5. serial\_initialize();
7. debug("Now running in RAM - U-Boot at: %08lx\n", dest\_addr);
8. </span>

如果打开CONFIG\_CLOCKS,set\_cpu\_clk\_info也是需要实现的板级支持函数。

重点来说一些serial\_initialize，对于最精简能正常启动的uboot，serial和ddr是必须正常工作的。

实现在drivers/serial/serial.c中，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**void** serial\_initialize(**void**)
2. {
3. mpc8xx\_serial\_initialize();
4. ns16550\_serial\_initialize();
5. pxa\_serial\_initialize();
6. s3c24xx\_serial\_initialize();
7. s5p\_serial\_initialize();
8. mpc512x\_serial\_initialize();。。。。
9. mxs\_auart\_initialize();
10. arc\_serial\_initialize();
11. vc0718\_serial\_initialize();
13. serial\_assign(default\_serial\_console()->name);
14. }</span>

所有串口驱动都会实现一个xxxx\_serial\_initialize函数，并且添加到serial\_initialize中，

xxxx\_serial\_initialize函数中是将所有需要的串口（用结构体struct serial\_device表示，其中实现了基本的收 发 配置）调用serial\_register注册，serial\_register如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**void** serial\_register(**struct** serial\_device \*dev)
2. {
3. #ifdef CONFIG\_NEEDS\_MANUAL\_RELOC
4. **if** (dev->start)
5. dev->start += gd->reloc\_off;
6. **if** (dev->stop)
7. dev->stop += gd->reloc\_off;
8. **if** (dev->setbrg)
9. dev->setbrg += gd->reloc\_off;
10. **if** (dev->getc)
11. dev->getc += gd->reloc\_off;
12. **if** (dev->tstc)
13. dev->tstc += gd->reloc\_off;
14. **if** (dev->putc)
15. dev->putc += gd->reloc\_off;
16. **if** (dev->puts)
17. dev->puts += gd->reloc\_off;
18. #endif
20. dev->next = serial\_devices;
21. serial\_devices = dev;
22. }</span>

就是将你的serial\_dev加到全局链表serial\_devices中。  
  
可以想象，如果你有4个串口，则再你的串口驱动中分别定义4个serial device，并实现对应的收发配置，然后serial\_register注册者4个串口。  
回到serial-initialize,最后调用serial\_assign，default\_serial\_console我们之前说过，就是你在串口驱动给出一个默认调试串口，serial\_assign如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**int** serial\_assign(**const** **char** \*name)
2. {
3. **struct** serial\_device \*s;
5. **for** (s = serial\_devices; s; s = s->next) {
6. **if** (strcmp(s->name, name))
7. **continue**;
8. serial\_current = s;
9. **return** 0;
10. }
12. **return** -EINVAL;
13. }</span>

serial\_assign就是从serial\_devices链表中找到指定的默认调试串口，条件就是串口的name，最后serial\_current就是当前的默认串口了。  
  
总结一下，serial\_initialize工作是将所有serial驱动中所有串口注册到serial\_devices链表中，然后找到指定的默认串口。

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">  /\* The Malloc area is immediately below the monitor copy in DRAM \*/
2. malloc\_start = dest\_addr - TOTAL\_MALLOC\_LEN;
3. mem\_malloc\_init (malloc\_start, TOTAL\_MALLOC\_LEN);</span>

根据咱们之前board\_init\_f中的分析，relocate addr之下的部分就是malloc的预留空间了。这里获取malloc首地址malloc\_start.

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**void** mem\_malloc\_init(ulong start, ulong size)
2. {
3. mem\_malloc\_start = start;
4. mem\_malloc\_end = start + size;
5. mem\_malloc\_brk = start;
7. memset((**void** \*)mem\_malloc\_start, 0, size);
9. malloc\_bin\_reloc();
10. }</span>

mem\_malloc\_init中就是对malloc预留的空间初始化，起始地址，结束地址，清空。咱们已经relocate，malloc\_bin\_reloc中无操作了。  
  
board\_init\_r接下来的代码是做一些外设的初始化，比如mmc flash eth，环境变量的设置，还有中断的使能等，这里需要说一下是关于串口的2个函数，stdio\_init和console\_init\_r.  
看stdio\_init代码，我们只定义了serial，会调到serial\_stdio\_init，如下:

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**void** serial\_stdio\_init(**void**)
2. {
3. **struct** stdio\_dev dev;
4. **struct** serial\_device \*s = serial\_devices;
6. **while** (s) {
7. memset(&dev, 0, **sizeof**(dev));
9. strcpy(dev.name, s->name);
10. dev.flags = DEV\_FLAGS\_OUTPUT | DEV\_FLAGS\_INPUT;
12. dev.start = s->start;
13. dev.stop = s->stop;
14. dev.putc = s->putc;
15. dev.puts = s->puts;
16. dev.getc = s->getc;
17. dev.tstc = s->tstc;
19. stdio\_register(&dev);
21. s = s->next;
22. }
23. }</span>

将serial\_devices链表上所有serial device同样初始化一个stdio\_dev，flag为output input，调用stdio-register，将stdio\_dev添加到全局devs链表中。  
  
可以想象，serial\_stdio\_init是在drivers/serial/serial.c中实现，uboot在这里是利用的内核分层思想，drivers/serial下是特定serial驱动，分别调用serial\_register注册到serial\_devices中，这可以说是通用的serial驱动层，  
  
通用serial层调用serial-stdio-init将所有serial注册到stdio device中，这就是通用的stdio层。  
  
看来分层思想还是非常重要的！  
  
board\_init\_r中调用完stdio\_init后又调用了console\_init\_r，如下

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**int** console\_init\_r(**void**)
2. {
3. **struct** stdio\_dev \*inputdev = NULL, \*outputdev = NULL;
4. **int** i;
5. **struct** list\_head \*list = stdio\_get\_list();
6. **struct** list\_head \*pos;
7. **struct** stdio\_dev \*dev;
9. /\* Scan devices looking for input and output devices \*/
10. list\_for\_each(pos, list) {
11. dev = list\_entry(pos, **struct** stdio\_dev, list);
13. **if** ((dev->flags & DEV\_FLAGS\_INPUT) && (inputdev == NULL)) {
14. inputdev = dev;
15. }
16. **if** ((dev->flags & DEV\_FLAGS\_OUTPUT) && (outputdev == NULL)) {
17. outputdev = dev;
18. }
19. **if**(inputdev && outputdev)
20. **break**;
21. }
23. **if** (outputdev != NULL) {
24. console\_setfile(stdout, outputdev);
25. console\_setfile(stderr, outputdev);
26. }
28. /\* Initializes input console \*/
29. **if** (inputdev != NULL) {
30. console\_setfile(stdin, inputdev);
31. }
33. #ifndef CONFIG\_SYS\_CONSOLE\_INFO\_QUIET
34. stdio\_print\_current\_devices();
35. #endif /\* CONFIG\_SYS\_CONSOLE\_INFO\_QUIET \*/
37. /\* Setting environment variables \*/
38. **for** (i = 0; i < 3; i++) {
39. setenv(stdio\_names[i], stdio\_devices[i]->name);
40. }
42. gd->flags |= GD\_FLG\_DEVINIT;    /\* device initialization completed \*/
44. **return** 0;
45. }</span>

console\_init\_r前半部分很清楚了，从devs.list链表中查找flag为output或者input的dev，如果只有serial之前注册了stdio\_dev，则outputdev inputdev都是咱们注册的第一个serial。  
  
之后调用console\_setfile，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**static** **int** console\_setfile(**int** file, **struct** stdio\_dev \* dev)
2. {
3. **int** error = 0;
5. **if** (dev == NULL)
6. **return** -1;
8. **switch** (file) {
9. **case** stdin:
10. **case** stdout:
11. **case** stderr:
12. /\* Start new device \*/
13. **if** (dev->start) {
14. error = dev->start();
15. /\* If it's not started dont use it \*/
16. **if** (error < 0)
17. **break**;
18. }
20. /\* Assign the new device (leaving the existing one started) \*/
21. stdio\_devices[file] = dev;
23. /\*
24. \* Update monitor functions
25. \* (to use the console stuff by other applications)
26. \*/
27. **switch** (file) {
28. **case** stdin:
29. gd->jt[XF\_getc] = dev->getc;
30. gd->jt[XF\_tstc] = dev->tstc;
31. **break**;
32. **case** stdout:
33. gd->jt[XF\_putc] = dev->putc;
34. gd->jt[XF\_puts] = dev->puts;
35. gd->jt[XF\_printf] = printf;
36. **break**;
37. }
39. **break**;
41. **default**:        /\* Invalid file ID \*/
42. error = -1;
43. }
44. **return** error;
45. }</span>

首先运行设备的start，就是特定serial实现的start函数。然后将stdio\_device放到stdio\_devices全局数组中，这个数组3个成员，stdout，stderr，stdin。最后还会在gd中设一下操作函数。  
  
在console\_init\_r中最后会改变gd中flag状态，为GD\_FLG\_DEVINIT。表示设备初始化完成。  
  
board\_init\_r进行完板级初始化后最后进入死循环，打印命令行，等待命令输入和解析。到这里uboot的启动过程就全部结束了！  
  
上面用很大篇幅自下往上解释uboot下serial到console的架构，那来看一下实际使用时由printf到最后serial输出这个自上到下的流程吧。  
  
首先来看printf，实现在common/console.c中如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**int** printf(**const** **char** \*fmt, ...)
2. {
3. **va\_list** args;
4. uint i;
5. **char** printbuffer[CONFIG\_SYS\_PBSIZE];
7. #if !defined(CONFIG\_SANDBOX) && !defined(CONFIG\_PRE\_CONSOLE\_BUFFER)
8. **if** (!gd->have\_console)
9. **return** 0;
10. #endif
12. va\_start(args, fmt);
14. /\* For this to work, printbuffer must be larger than
15. \* anything we ever want to print.
16. \*/
17. i = vscnprintf(printbuffer, **sizeof**(printbuffer), fmt, args);
18. va\_end(args);
20. /\* Print the string \*/
21. puts(printbuffer);
22. **return** i;
23. }</span>

字符串的拼接跟一般printf实现一样，最后调用puts，puts实现也在console.c中，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**void** puts(**const** **char** \*s)
2. {
3. #ifdef CONFIG\_SANDBOX
4. **if** (!gd) {
5. os\_puts(s);
6. **return**;
7. }
8. #endif
10. #ifdef CONFIG\_SILENT\_CONSOLE
11. **if** (gd->flags & GD\_FLG\_SILENT)
12. **return**;
13. #endif
15. #ifdef CONFIG\_DISABLE\_CONSOLE
16. **if** (gd->flags & GD\_FLG\_DISABLE\_CONSOLE)
17. **return**;
18. #endif
20. **if** (!gd->have\_console)
21. **return** pre\_console\_puts(s);
23. **if** (gd->flags & GD\_FLG\_DEVINIT) {
24. /\* Send to the standard output \*/
25. fputs(stdout, s);
26. } **else** {
27. /\* Send directly to the handler \*/
28. serial\_puts(s);
29. }
30. }</span>

gd->have\_console在board\_init\_f的console\_init\_f中置位，flag的GD\_FLG\_DEVINIT则是在刚才board\_init\_r中console\_init\_r最后置位。

如果GD\_FLG\_DEVINIT没有置位，表明console没有注册，是在board\_init\_f之后，board\_init\_r执行完成之前，这时调用serial\_puts,如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**void** serial\_puts(**const** **char** \*s)
2. {
3. get\_current()->puts(s);
4. }       </span>

直接调到serial.c中的函数，完全符合board\_init\_f中serial\_init的配置，仅仅找到一个默认串口来使用，其他串口暂且不管。  
如果GD\_FLG\_DEVINIT置位，表明console注册完成。调用fputs，如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/skyflying2012/article/details/25804209)[在CODE上查看代码片](https://code.csdn.net/snippets/344954)

1. <span style="font-size:14px;">**void** fputs(**int** file, **const** **char** \*s)
2. {
3. **if** (file < MAX\_FILES)
4. console\_puts(file, s);
5. }
7. **static** **inline** **void** console\_puts(**int** file, **const** **char** \*s)
8. {
9. stdio\_devices[file]->puts(s);
10. }</span>

fputs调console\_puts从全局stdio\_devices中找到对应stdout对应的成员stdio\_device，调用puts，最终也是会调用到特定serial的puts函数。  
  
  
分析后总结一下：  
  
  
可以看出，对于serial，uboot实现了一个2级初始化：  
  
  
stage 1，仅初始化default console serial，printf到puts后会直接调用特定串口的puts函数，实现打印  
  
  
stage 2，将所有serial注册为stdio\_device，并挑出指定调试串口作为stdio\_devices的stdout stdin stderr。printf到puts后再到全局stdio\_devices中找到对应stdio\_device，调用stdio-device的puts，最终调用特定serial的puts，实现打印。  
  
  
区分这2个stage，是利用gd的flag，GD\_FLG\_DEVINIT。