新版uboot启动

一个嵌入式的存储设备通过通常包括四个分区：第一分区：存放的当然是u-boot第二个分区：存放着u-boot要传给系统内核的参数，第三个分区：是系统内核（kernel）第四个分区：则是根文件系统

U-Boot启动内核的过程可以分为两个阶段，两个阶段的功能如下：

（1）第一阶段的功能

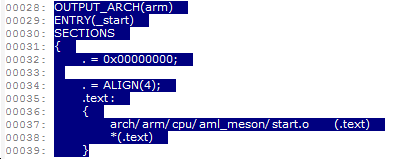
Ø  硬件设备初始化Ø  加载U-Boot第二阶段代码到RAM空间Ø  设置好栈Ø  跳转到第二阶段代码入口

（2）第二阶段的功能

Ø  初始化本阶段使用的硬件设备Ø  检测系统内存映射Ø  将内核从Flash读取到RAM中Ø  为内核设置启动参数Ø  调用内核

1第一阶段

嵌入式系统中，系统在上电或复位时通常都从地址0x00000000处开始执行，而在这个地址处安排的通常就是系统的Boot Loader程序。一个可执行的image必须有一个入口点，并且只能有一个**全局入口点**，所以要通知编译器这个入口在哪里。由此我们可以找到程序的入口点是在**u-boot.lds**中指定的，其中ENTRY(\_start)说明程序从\_start开始运行，而他指向的是start.o文件。



u-boot.lds找到start.S的入口，整个程序的入口取决于连接脚本中ENTRY声明的地方。ENTRY(\_start)因此 \_start符号是整个程序的入口，\_start所在的代码就是整个程序的起始代码。

1.1头文件包含

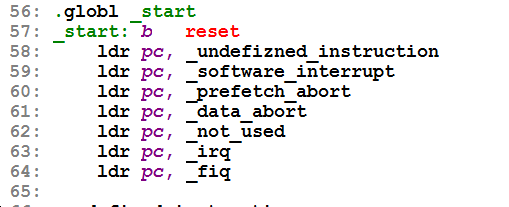
#include <config.h> config.h是在include文件下，这个文件不是源码中的文件，是配置时自动生成的文件。详见mkconfig脚本，在mkcofig脚本最后 发现

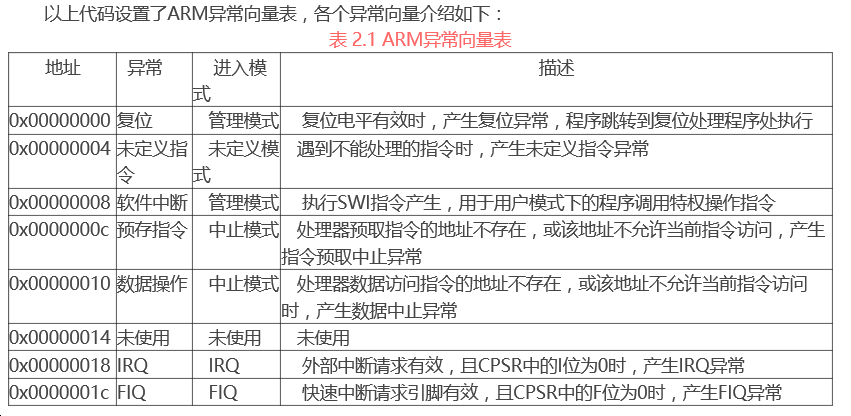
# 

/build/include/config.h /include/linux/config.h /arch/arm/cpu/aml\_meson/m8b/firmware/arc\_power/config.h arch/arm/include/asm/config.h cpu.h #include <version.h>

1.2 start.s分析

1.2.1异常向量表构建



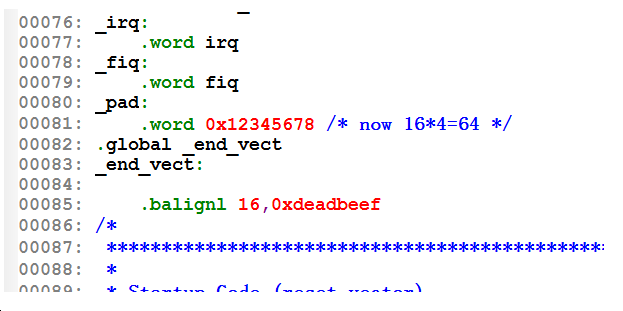


这个异常表顺序是CPU设计时决定的，是硬件决定的。这些异常应该被处理，如果不处理这些这些异常，程序会跑飞。

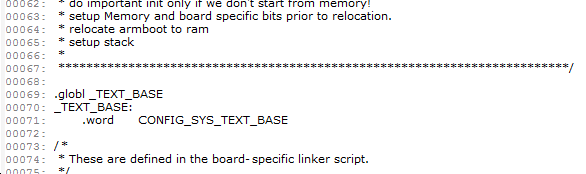
　　当一个异常产生时，CPU根据异常号在异常向量表中找到对应的异常向量，然后执行异常向量处的跳转指令，CPU就跳转到对应的异常处理程序执行。

  复位异常处理：复位异常的代码是 b reset ，因此在CPU复位后真正去执行的有效代码是reset处的代码，因此reset符号处，才是真正有意义代码的开始处。

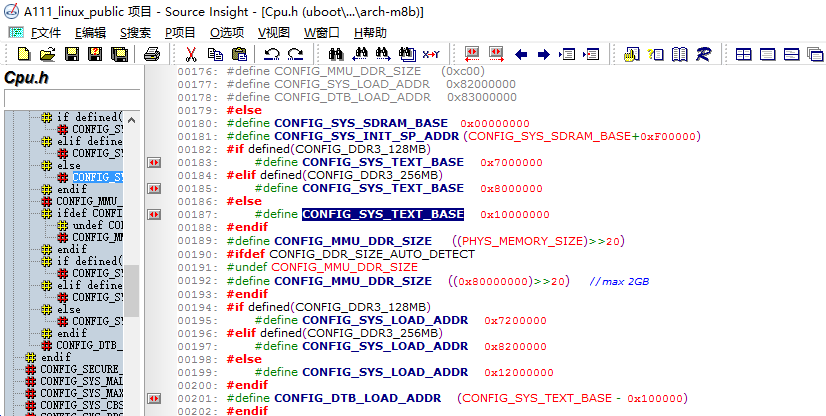
    .balignl 16,0xdeadbeef  .balignl 16 是以16字节对齐，如果没有对齐，用0xdeadbeef这个数字来填充，这个填充没有什么特别的意义，

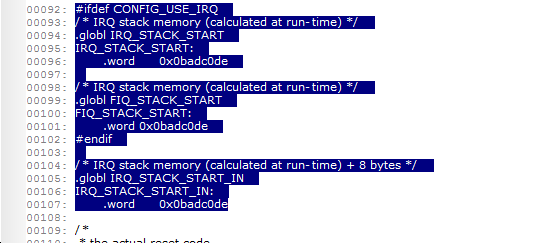


　TEXT\_BASE



TEXT\_BASE在整个代码里都是被引用，它是在makefile里配置阶段时的TEXT\_BASE，其实就是我们链接时，指定的程序的链接地址

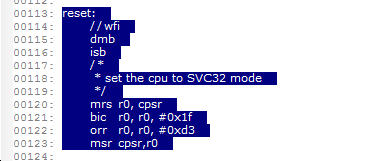




如果uboot中使用中断，这里声明中断处理函数栈起始地址，这里给出的值是0x0badc0de，是一个非法值，注释也说明了，这个值会在运行时重新计算，我查找了一下代码是在interrupt\_init中

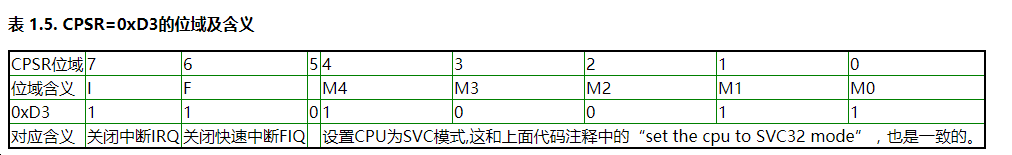
1.2.2 **CPU进入SVC模式和关看门口**

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。start.S中真正的复位代码：



修改cpsr寄存器，设置处理器进入svc模式，并且关掉irq和fiq。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。

其实我们arm CPU默认的就是SVC模式的，但是这里还是用软件将其置为SVC模式，整个uboot也一直工作在SVC模式。用软件来设置只是为了不管硬件的预设。

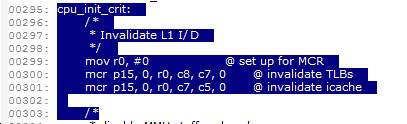


**看门狗：**watchdog一般是一个硬件模块，其作用是，在嵌入式操作系统中，很多应用情况是系统长期运行且无人看守，所以难免或者怕万一出现系统死机，这时，watchdog就会自动帮你重启系统。watchdog硬件的逻辑就是，其硬件上有个记录超时功能，然后要求用户需要每隔一段时间（此时间可以根据自己需求而配置）去对其进行一定操作，比如往里面写一些固定的值，俗称“喂狗”，那么我发现超时了，即过了这么长时间你还不给偶喂食，那么偶就认为你系统是死机了，出问题了，偶就帮你重启系统。此处只是去用uboot初始化必要的硬件资源和系统资源而已，完全用不到这个watchdog的机制。需要用到，那也是你linux内核跑起来了，是你系统关心的事情，和我uboot没啥关系的，所以肯定此处要去关闭watchdog（的reset功能）了。**说白了**，就是你运行的代码如果超出喂狗时间，而你不关狗，就会导致，你代码还没运行完又得去喂狗，就这样反复得重启CPU，那你代码永远也运行不完，所以，得先关看门狗得原因，就是这样。

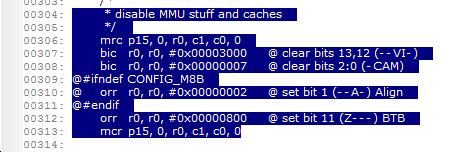
接下来如果没有定义宏CONFIG\_SKIP\_LOWLEVEL\_INIT，则会跳转执行cpu\_init\_crit。



刷新L1的icache和dcache



1.2.3关闭MMU



**刷新L1 cache的icache和dcache。关闭MMU**  
总结：和CPU的cache和mmu有关的，不用去细看，大概知道即可。

**关catch**

       catch和MMU是通过CP15管理的，刚上电的时候，CPU还不能管理他们

       上电的时候MMU必须关闭，指令catch可关闭，可不关闭，但数据catch一定要关闭

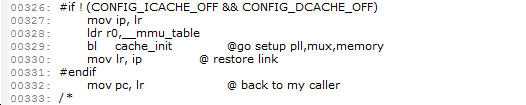
       否则可能导致刚开始的代码里面，去取数据的时候，从catch里面取，而这时候RAM中数据还没有catch过来，导致数据预取异常

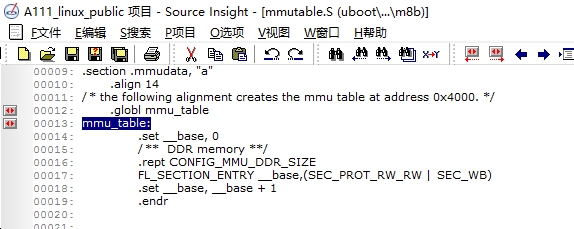
**关MMU**

      因为MMU是;把虚拟地址转化为物理地址得作用

      而目的是设置控制寄存器，而控制寄存器本来就是实地址（物理地址），再使能MMU，不就是多此一举了吗？







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。

从cpu\_init\_crit返回后，\_start的工作就完成了，总结一下\_start工作：

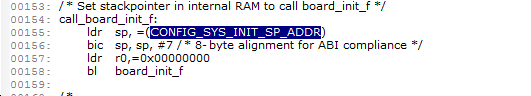
1 前面总结过的部分，初始化异常向量表，设置svc模式，关中断, 关闭MMU

2 配置cp15，初始化mmu cache tlb

3 板级初始化，pll memory初始化

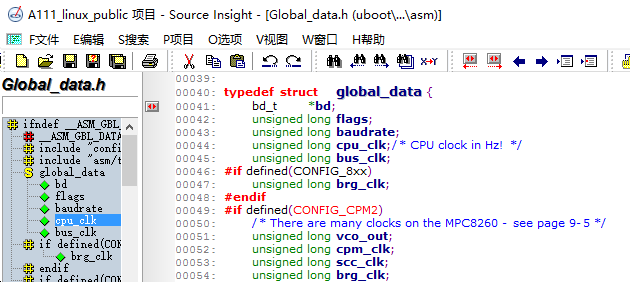
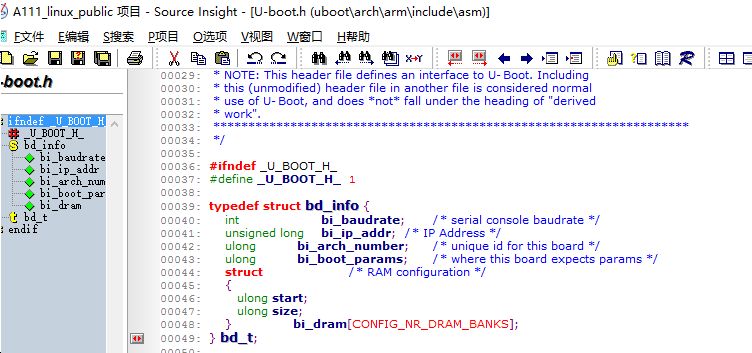
当cpu\_init\_crit 初始化完毕后，会顺序执行到call\_board\_init\_f的地方。

1. /\*
2. \* Set up initial C runtime environment and call board\_init\_f(0).
3. \*/



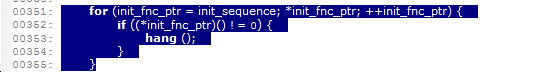
首先将CONFIG\_SYS\_INIT\_SP\_ADDR定义的值加载到栈指针sp中，这个宏定义在配置头文件中指定。这段代码是为board\_init\_f （uboot/arch/arm/lib/board.c）C函数调用提供环境，也就是栈指针sp初始化。必须要明白的一点是，当代码从存储介质(nand flash，SD，norflash，onenand等)中搬运到了DRAM中后随即会跳转到内存中运行u-boot，接着会有一个重定位(relocate\_code)的过程，relocate\_code子函数在start.S中，当判断到当前u-boot在内存的低地址处，那么relocate\_code就会工作，把u-boot代码从低地址处再搬运到内存地址的顶端，然后跳转到新的位置去继续运行u-boot。而搬运的目标地址是在board\_init\_f()函数中计算出来的. 如果u-boot一旦被搬运到内存中运行，那么是必须要跳过时钟和DRAM的初始化的，因为这在搬运之前已经做过了。并且如果代码在内存中运行的时侯你却去初始化DRAM，那必然导致崩溃！

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。



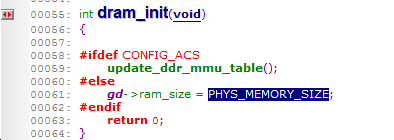
初始化mon\_len ,gd->mon\_len = \_uboot\_end\_ofs; u-boot code, data & bss段大小总和

遍历调用init\_sequence所有函数，



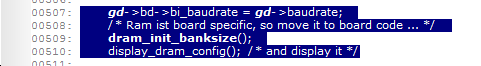


dram\_init对gd->ram\_size初始化，以便board\_init\_f后面代码对dram空间进行规划。



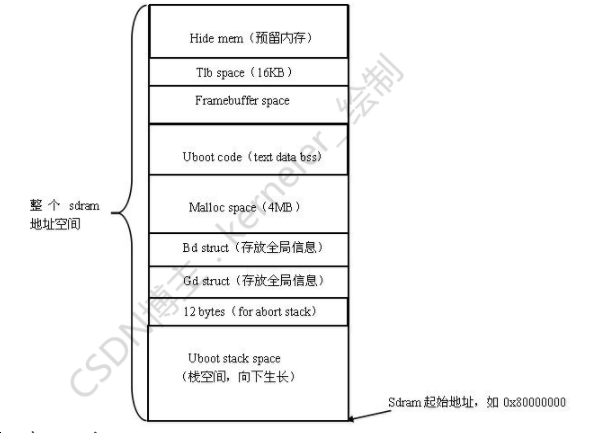
addr = CONFIG\_SYS\_SDRAM\_BASE + gd->ram\_size;

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



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]。



最后进入

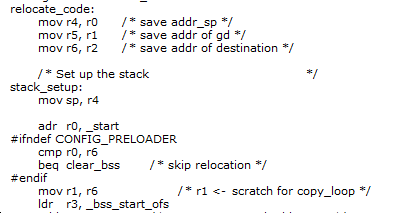
relocate\_code(addr\_sp, id, addr);这个将addr\_sp，id ，addr 传入参数，跳转到了start.S

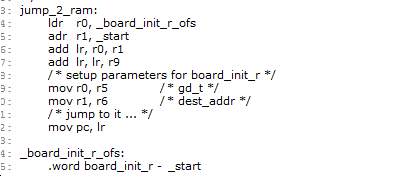
（uboot最重要的一个功能就是relocate，重定向。重定向最通俗的说，就是把NAND Flash上程序搬到内存中。重定向涉及到三个地址，链接地址、加载地址和运行地址。加载地址是uboot重定向时，拷贝源码的目的地址。通俗地讲，就是把NAND Flash里的uboot代码拷贝到内存的哪个地址上。

运行地址是指uboot在内存中运行的地址。uboot运行时的一些寻址操作，就跟运行地址有关系。

由于uboot的二进制文件直接或间接地包括了内存的绝对物理地址，所以以上三个地址一般是一样的。通俗地讲，uboot把内存的绝对物理地址固定在了二进制文件里，所以uboot应该被拷贝到内存哪个地方，下一条指令应该跳转到哪个物理地址都是在编译的时候就已经固定的。新版本的uboot有两个重定向，第一个重定向跟老版本是一模一样的，目的就是从NAND Flash或者NOR Flash这些存储设备中把代码拷贝到内存地址(destA)中。还有第二个重定向，这个重定向函数就是relocate\_code。这部分代码是uboot官方做好的，我们并不需要修改。它的作用就是把uboot从当前的位置(destA)拷贝到内存的高端地址(destB)上。Linux内核一般用的低端的内存地址，所以uboot选了一个高端地址。但至于为什么还得有第二次的重定向，原因我并不清楚。即使把第二次的重定向屏蔽，uboot还是可以正常运行的

）





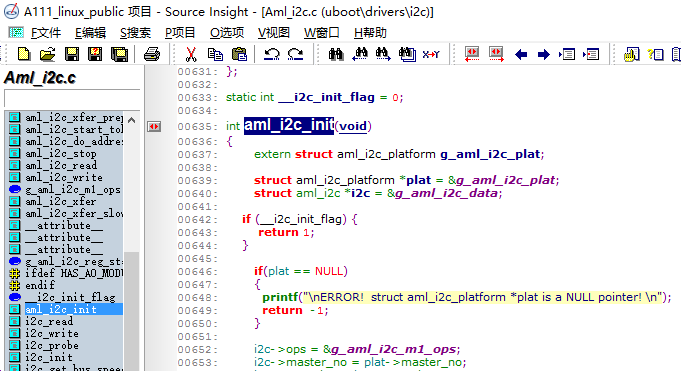
这中间将mmc上的代码拷贝到ram中后，跳转到board\_init\_r函数中Board.c (arch\arm\lib)

**2、第二阶段**

board\_init\_r()是进入定制板目录的入口：

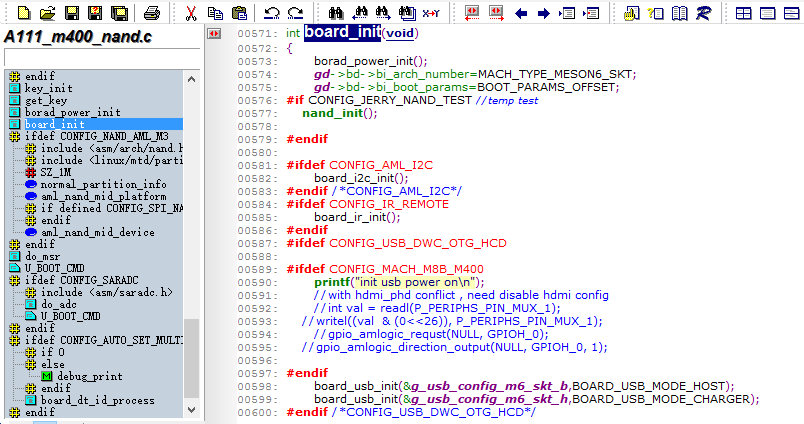


对于最精简能正常启动的uboot，serial和ddr是必须正常工作的。所有串口驱动都会实现一个xxxx\_serial\_initialize函数，并且添加到serial\_initialize中，xxxx\_serial\_initialize函数中是将所有需要的串口（用结构体struct serial\_device表示，其中实现了基本的收 发 配置）调用serial\_register注册总结一下，serial\_initialize工作是将所有serial驱动中所有串口注册到serial\_devices链表中，然后找到指定的默认串口。



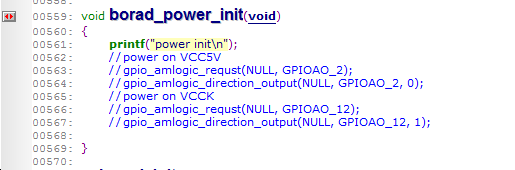
.**关键!!!这个就是我们苦苦寻找的板级定制文件的入口函数！！！**

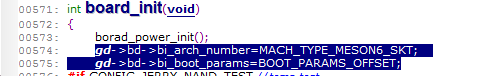
board\_init(); /\* Setup chipselects \*/





Vcck电源

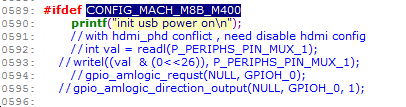




**gd->bd->bi\_arch\_number**  
(1)bi\_arch\_number是board\_info中的一个元素，含义是：开发板的机器码。所谓机器码就是uboot给这个开发板定义的一个唯一编号  
(2)机器码的主要作用就是在uboot和linux内核之间进行比对和适配。  
(3)嵌入式设备中每一个设备的硬件都是定制化的，不能通用。嵌入式设备的高度定制化导致硬件和软件不能随便适配使用。这就告诉我们：这个开发板移植的内核镜像绝对不能下载到另一个开发板去，否则也不能启动，就算启动也不能正常工作，有很多隐患。因此linux做了个设置：给每个开发板做个唯一编号（机器码），然后在uboot、linux内核中都有一个软件维护的机器码编号。然后开发板、uboot、linux三者去比对机器码，如果机器码对上了就启动，否则就不启动（因为软件认为我和这个硬件不适配）。  
**gd->bd->bi\_boot\_params**

(1)bd\_info中另一个主要元素，bi\_boot\_params表示uboot给linux kernel启动时的传参的内存地址。也就是说uboot给linux内核传参的时候是这么传的：uboot事先将准备好的传参（字符串，就是bootargs）放在内存的一个地址处（就是bi\_boot\_params），然后uboot就启动了内核（uboot在启动内核时真正是通过寄存器r0 r1 r2来直接传递参数的，其中有一个寄存器中就是bi\_boot\_params）。内核启动后从寄存器中读取bi\_boot\_params就知道了uboot给我传递的参数到底在内存的哪里。然后自己去内存的那个地方去找bootargs。

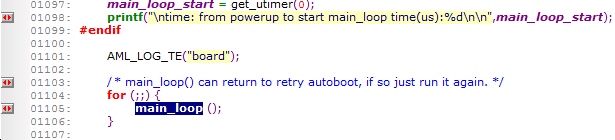
Usb电源



在board\_init\_r()中，







       main\_loop函数在common/main.c中定义。

**U-Boot启动Linux过程**

       U-Boot使用标记列表（tagged list）的方式向Linux传递参数。标记的数据结构式是tag，在U-Boot源代码目录include/asm-arm/setup.h中定义如下：

struct tag\_header {

       u32 size;       /\* 表示tag数据结构的联合u实质存放的数据的大小\*/

       u32 tag;        /\* 表示标记的类型 \*/

};

U-Boot使用命令bootm来启动已经加载到内存中的内核。而bootm命令实际上调用的是do\_bootm函数。对于Linux内核，do\_bootm函数会调用do\_bootm\_linux函数来设置标记列表和启动内核。do\_bootm\_linux函数在lib\_arm/bootm.c

kernel\_entry = (void (\*)(int, int, uint))images->ep; /\* 获取内核入口地址 \*/

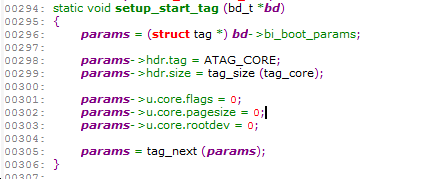
setup\_start\_tag (bd);             /\* 设置ATAG\_CORE标志 \*/

  setup\_memory\_tags (bd);                             /\* 设置内存标记 \*/

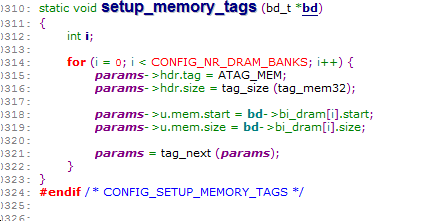
   setup\_commandline\_tag (bd, commandline);      /\* 设置命令行标记 \*/

  setup\_end\_tag (bd);                               /\* 设置ATAG\_NONE标志 \*/

kernel\_entry(0, machid, bd->bi\_boot\_params); /\* 调用内核 \*/



标记列表必须以ATAG\_CORE开始，setup\_start\_tag函数在内核的参数的开始地址设置了一个ATAG\_CORE标记。



  setup\_memory\_tags函数设置了一个ATAG\_MEM标记，该标记包含内存起始地址，内存大小这两个参数。

static void setup\_end\_tag (bd\_t \*bd)

{

       params->hdr.tag = ATAG\_NONE;

       params->hdr.size = 0;

}

       标记列表必须以标记ATAG\_NONE结束，setup\_end\_tag函数设置了一个ATAG\_NONE标记，表示标记列表的结束。

       U-Boot设置好标记列表后就要调用内核了。但调用内核前，CPU必须满足下面的条件：

（1）    CPU寄存器的设置

Ø  r0=0

Ø  r1=机器码

Ø  r2=内核参数标记列表在RAM中的起始地址

（2）    CPU工作模式

Ø  禁止IRQ与FIQ中断

Ø  CPU为SVC模式

（3）    使数据Cache与指令Cache失效

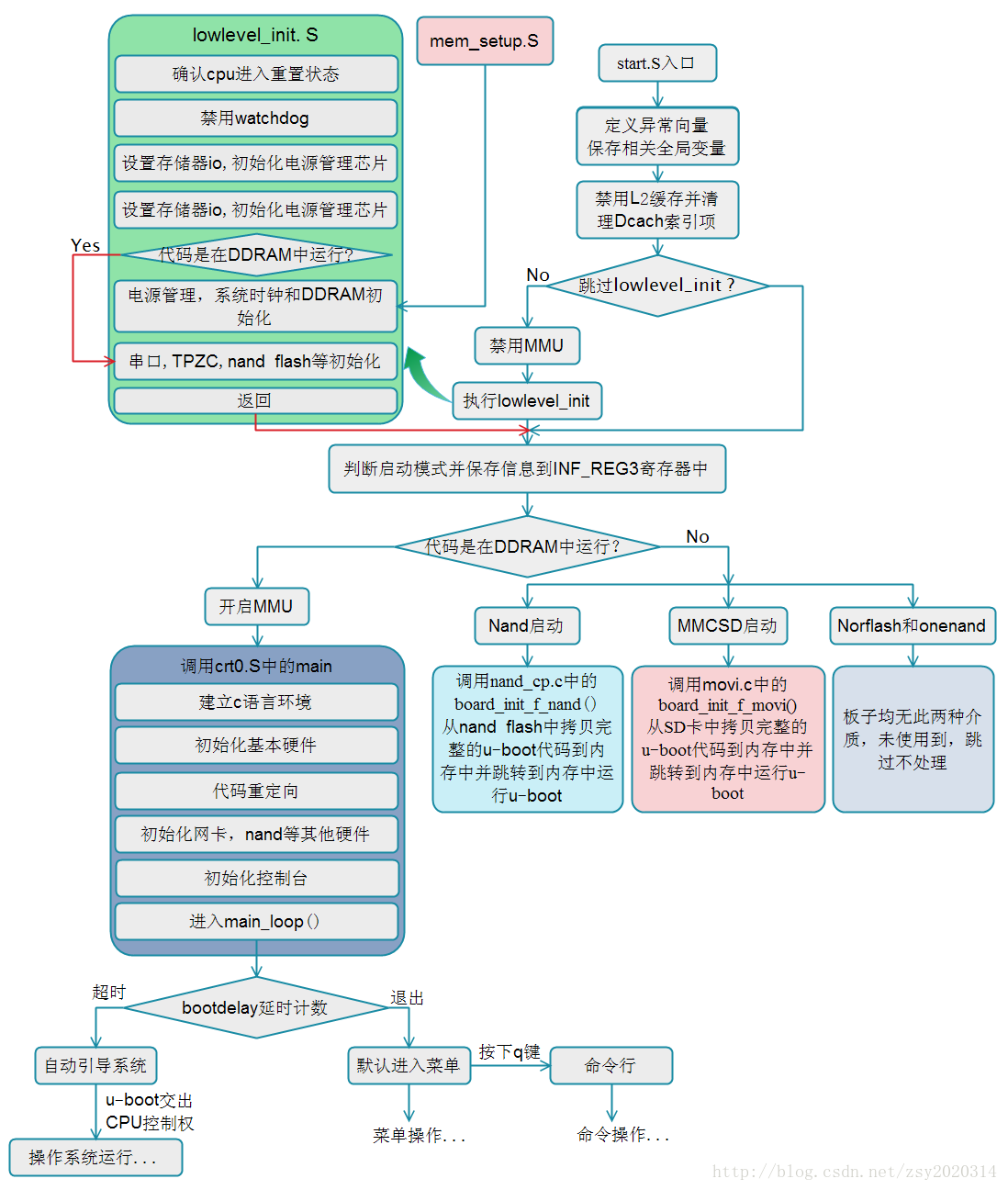
73       theKernel = (void (\*)(int, int, uint))images->ep;

… …

128      theKernel (0, machid, bd->bi\_boot\_params);

       第73行代码将内核的入口地址“images->ep”强制类型转换为函数指针。根据ATPCS规则，函数的参数个数不超过4个时，使用r0~r3这4个寄存器来传递参数。因此第128行的函数调用则会将0放入r0，机器码machid放入r1，内核参数地址bd->bi\_boot\_params放入r2，从而完成了寄存器的设置，最后转到内核的入口地址。

       到这里，U-Boot的工作就结束了，系统跳转到Linux内核代码执行。



启动代码相关的几个文件在u-boot中的路径

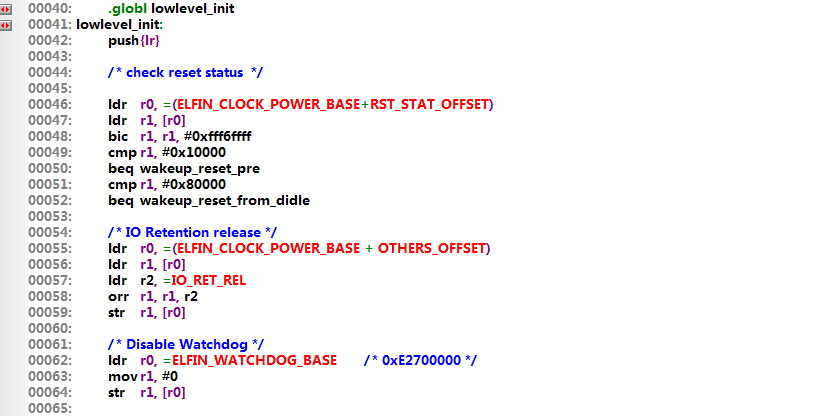
    start.S：            /arch/arm/cpu/armv7/start.S (需要自己修改)

    lowlevel\_init.S：/board/samsung/zsy210/ lowlevel\_init.S  (需要自己修改)

    mem\_setup.S:   /board/samsung/zsy210/ mem\_setup.S   (u-boot没有，需要自己添加)

    ctr0.S：            /arch/arm/lib/ctr0.S  (u-boot自带,一般不需要修改)

这里是第一次设置栈，这次设置栈是在SRAM中设置的，因为当前整个代码还在SRAM中运行，此时DDR还未初始化还不能用，这是只有内部SRAM可以用。



 一检查复位状态

（1）复杂CPU允许多种复位情况，譬如直接冷上电，热启动，睡眠状态下的唤醒，这些情况都属于复位，所以我们在复位代码中要去检测复位状态，来判断到底是哪种情况。

（2）判断哪种复位的意义在于：冷上电时，DDR是需要初始化，而热启动和睡眠状态的唤醒时不需要初始化DDR的。

/\* IO Retention release \*/

ldr r0, =(ELFIN\_CLOCK\_POWER\_BASE + OTHERS\_OFFSET)

ldr r1, [r0]

ldr r2, =IO\_RET\_REL

orr r1, r1, r2

str r1, [r0]

二：IO状态恢复

IO状态复位和检查复位状态和主线启动代码无关，

/\* Disable Watchdog \*/

ldr r0, =ELFIN\_WATCHDOG\_BASE /\* 0xE2700000 \*/

mov r1, #0

str r1, [r0]

关看门狗

三：来判定当前运行在SRAM（不相等）中还是DDR（相等）中，然后来决定是否跳过是时钟初始化部分和DDR 初始化

/\* when we already run in ram, we dont need to relocate U-Boot.

\* and actually, memory controller must be configured before U-Boot

\* is running in ram.

\*/

ldr r0, =0xff000fff

bic r1, pc, r0 /\* r0 <- current base addr of code \*/

ldr r2, \_TEXT\_BASE /\* r1 <- original base addr in ram \*/

bic r2, r2, r0 /\* r0 <- current base addr of code \*/

cmp r1, r2 /\* compare r0, r1 \*/

beq 1f /\* r0 == r1 then skip sdram init \*/

/\* init system clock \*/

bl system\_clock\_init

/\* Memory initialize \*/

bl mem\_ctrl\_asm\_init

1:

/\* for UART \*/

bl uart\_asm\_init

（1） 判断当前代码执行的位置，110-115行，判断当前的代码是否需要重定位，判断当前代码是在SRAM中还是在DDR中，

为什么需要这个判断？

　　　　原因1：BL1（uboot的前一部分）是在SRAM中有一份，在DDR中也有一份，因此如果是冷启动，当前的代码应该是在SRAM中运行的BL1，如果是低功耗状态下的复位这时候 应该就是在DDR中运行的。

　　　　原因2：我们判定当前运行代码的地址是有用的，可以指导后面代码的运行。譬如在lowlevel\_init.S中，判定当前代码的运行地址，就是为了确定要不要执行时钟初始化和DDR初始化的代码。如果当前代码在SRAM中，说明是冷启动，那么时钟和DDR初始化都要执行，如果是DDR中运行，说明是热启动，就不需要执行时钟和DDR的初始化。

（2）bic r1, pc, r0   /\* r0 <- current base addr of code \*/

将pc的值中的某些比特位清零，剩下一些特殊的比特位赋值给r1（r0中为1的那些位清零）相当于 r1 = pc & ~(ff000fff)

(3) ldr r2, \_TEXT\_BASE /\* r1 <- original base addr in ram \*/  链接地址加载到r2，然后将r2的相应位清零，剩下特定位。

（4）最后比较r1和r2

总结：读取当前运行地址和连接地址，然后处理两个地址后对比是否相等，来判定当前运行在SRAM（不相等）中还是DDR（相等）中，然后来决定是否跳过是时钟初始化部分和DDR 初始化

四：system\_clock\_init 初始化系统时钟

五：mem\_ctrl\_asm\_init 初始化内存

总结：lowlevel\_init.S总共做了哪些事情?

lowlevel\_init.S做了哪些事情：

检查复位状态，IO恢复，关看门狗，开发板供电锁存，判断代码运行地址，时钟初始化，DDR初始化， 串口初始化

mlinux：这个就是原始的未经任何处理加工的原版内核elf文件，他一般加载在

PC机这种的硬盘中，因为未加工（比较大），所以PC机有大硬盘不怕放不下，

所以PC机的上电后直接加载vmlinux内核镜像就可以运行了。比如（Ubuntu）。

Image：vmlinux经过objcopy去掉一些不需要的东西之后得到的。

zImage：Image经过压缩得到的。原则上，Image就可以直接运行了，但是，嵌

入式系统要求精简所以把Image压缩在加上压缩算法放到真正的压缩镜像前面就

得到了zImage。如图可知zImage变小了。

uImage：uImage是经过加64字节的头部信息放到zImage的前面合成得到的。

uImage是用于uboot引导启动时提供的内核镜像，所以uboot在编译成功后，

在根目录下tools/下回提供一个mkimage的工具，将该工具拷贝到系统目录下，

在内核根目录执行make uImage就可以提供zImage生成uImage。

**总结：uboot启动内核过程：**

**1 不惜任何手段将kernel镜像加载到DDR**

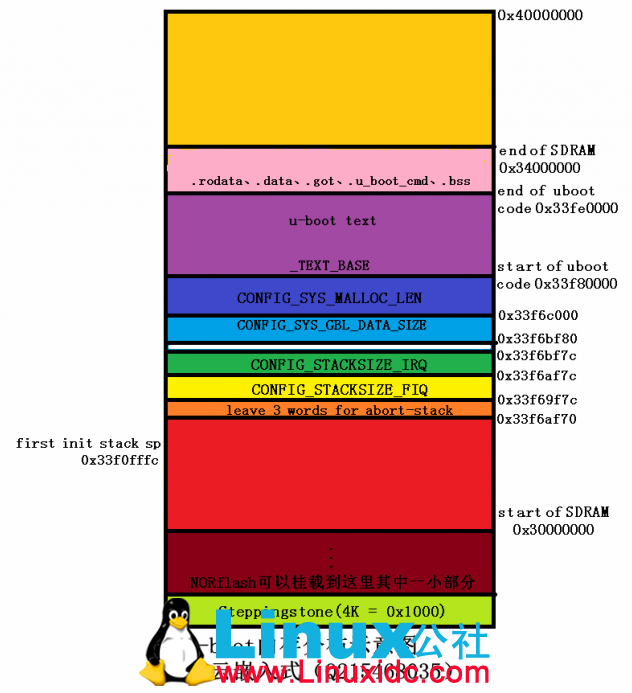
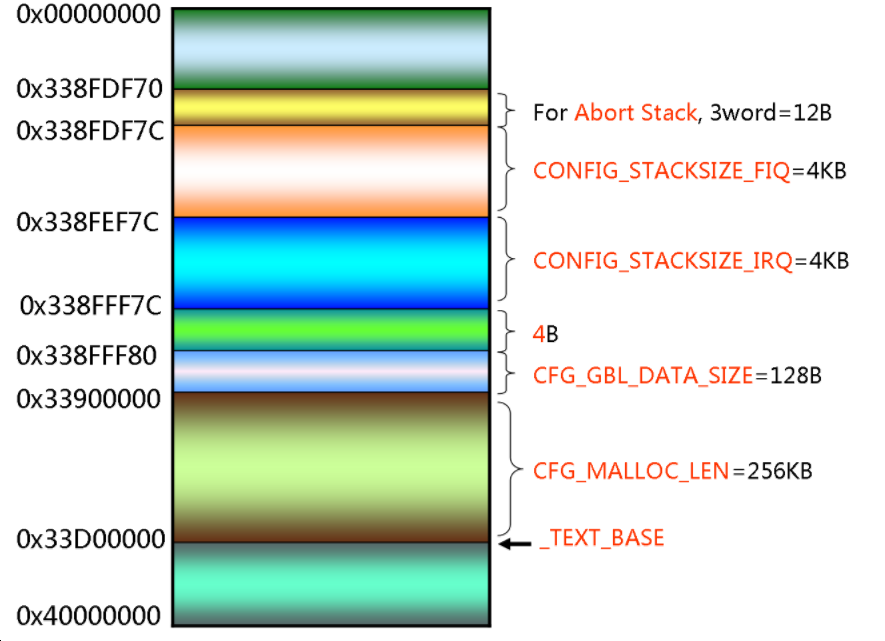
**2 检查是何种镜像**

**3 设置给内核的传参**

**4 运行内核**

程序计数器R15（PC）总是指向取指的指令，而不是指向正在执行的指令或者正在译码的指令。一般情况下，人们总是习惯把正在执行的指令作为参考点，称之为当前第1条指令，因此PC总是指向第3条指令。

对于ARM状态下指令，PC值=当前程序执行位置+8；



**1、再次设置栈（DDR中的栈）**

(1)再次开发板供电锁存。第一，做2次是不会错的；第二，做2次则第2次无意义；做代码移植时有一个古怪谨慎保守策略就是尽量添加代码而不要删除代码。

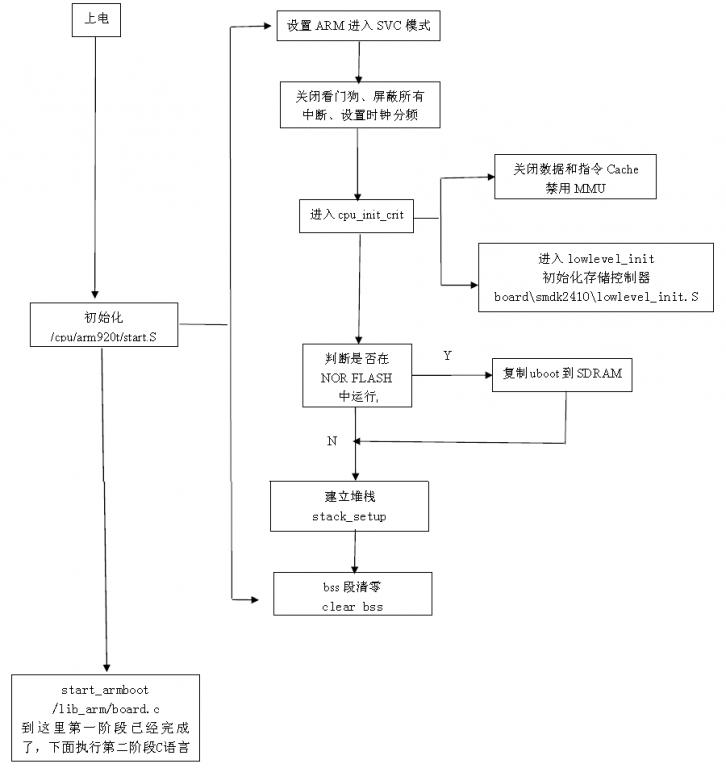
(2)之前在调用lowlevel\_init程序前设置过1次栈（start.S 284-287行），那时候因为DDR尚未初始化，因此程序执行都是在SRAM中，所以在SRAM中分配了一部分内存作为栈。本次因为DDR已经被初始化了，因此要把栈挪移到DDR中，所以要重新设置栈，这是第二次（start.S 297-299行）；这里实际设置的栈的地址是33E00000，刚好在uboot的代码段的下面紧挨着。

(3)为什么要再次设置栈？DDR已经初始化了，已经有大片内存可以用了，没必要再把栈放在SRAM中可怜兮兮的了；原来SRAM中内存大小空间有限，栈放在那里要注意不能使用过多的栈否则栈会溢出，我们及时将栈迁移到DDR中也是为了尽可能避免栈使用时候的小心翼翼。

**2、再次判断当前地址以决定是否重定位**

(1)再次用相同的代码判断运行地址是在SRAM中还是DDR中，不过本次判断的目的不同（上次判断是为了决定是否要执行初始化时钟和DDR的代码）这次判断是为了决定是否进行uboot的relocate。

(2)冷启动时当前情况是uboot的前一部分（16kb或者8kb）开机自动从SD卡加载到SRAM中正在运行，uboot的第二部分（其实第二部分是整个uboot）还躺在SD卡的某个扇区开头的N个扇区中。此时uboot的第一阶段已经即将结束了（第一阶段该做的事基本做完了），结束之前要把第二部分加载到DDR中链接地址处（33e00000），这个加载过程就叫重定位。



<https://blog.csdn.net/ce123_zhouwei/article/details/7304909>

https://blog.csdn.net/ce123\_zhouwei/article/details/7318851

