Sitara AM335x Bootload 的流程分析

平台：AM335x

Linux SDK ： PROCESSOR-SDK-LINUX-AM335X（v03.01）

此 SDK 下载网址: http://www.ti.com/tool/processor-sdk-am335x

文章面向对象：ARM 的初学者 或 SDK 新版本初级使用者

文章 base 操作：在 Linux 平台成功安装上述的 Linux SDK 以及 uboot 成功编译。

若在安装或者编译有问题，请参考以下关于文章基础操作的教程 link：

http://processors.wiki.ti.com/index.php/Processor_SDK_Linux_Getting_Started_Guide

http://processors.wiki.ti.com/index.php/Processor_SDK_Linux_U-Boot

文章内容：第一部分概括移植 Linux 的启动整体步骤，接着第二部分会描述出厂固化在 am335x 的 ROM code 的作用、引导模式和执行流程，然后第三部分描述 SPL 和 Uboot 的运行的位置和一下基础性知识。最后就是重点：SPL、Uboot 的流程图，以及 DDR 的内容分布。

由于 SPL、Uboot 的代码冗长和某些函数实现的功能复杂，所以分析 SPL、Uboot 的流程图，我主要是以”全而简”（全过程+简单分析）的角度进行介绍，这样做的目的主要是让大家比较清晰地知道 bootload 在哪里在哪个文件在哪个函数实现什么功能，可以方便地让大家随时随地定位你想要了解的功能的位置。

若在细节上有任何疑问，可以根据我画的流程图方便查找和定位程序所在地，若仍然解决不了，欢迎大家可以在论坛上积极地提出来。谢谢！

好，现在开始讲述文章内容。

一．Bootloader 的整体运行流程为：

1. 芯片时序性上电或复位
2. rom_code 运行，从外设或外部存储器中加载二级 bootloader(SPL)到内存中运行
3. SPL 做 CPU 和外设的初始化(主要是 DDR)并将 u-boot 加载到内存运行
4. u-boot 继续做其它板载或外设的初始化并加载 Linux 内核
5. linux 内核开始启动

二．rom_code

am335x 系列出厂时在芯片内部的 ROM 中固化了一段代码，称为 rom_code，在芯片上电或复位后首先执行芯片内部固化的这段代码，这段代码的作用是：引导二级 bootloader（SPL）的镜像文件到内部 sram 中运行。

三．am335x 引导模式

am335x 的引导模式其实就是二级 bootloader（SPL）镜像的读取方式，刚才提到 rom_code 的作用是将二级 bootloader 加载到内部的 SRAM 运行，那么 rom_code 从什么地方以及通过什么样的方式获取 SPL 的可执行镜像文件呢？这就决定于 am335x 的引导模式了。

am335x 支持两大类的引导模式，分别为 memory 模式及外设模式。memory 模式可以从 SD 卡、NAND flash、NOR flash 及 EMMC 中读取 SPL 镜像文件并加载到内部 SRAM 中运行。外设模式是指 rom_code 可以通过芯片的一些通信接口（比如以太网、串口、SPI、USB）获取 SPL 镜像文件并加载到内部 SRAM 中运行。

am335x 的引导模式取决于上电或复位时芯片一组引脚的状态，这组引脚称作 SYSBOOT[15:0]，一共 16 位。其中 SYSBOOT[4:0]决定了芯片的引导顺序。

举个例子：下图一中可以看出当 SYSBOOT[4:0]为 00010 时，芯片会首先从 UART0 开始引导，如果失败会尝试 SPI0、NAND、NANDI2C。

在这里顺便提一下 SYSBOOT[15:14]这组引脚，它们决定了外部晶振频率，注意在设计时，重点关注这几个关于外部晶振频率的引脚，根据外部所接的晶振频率拉低或拉高相关引脚，如图一所示。晶振频率可选：19.2 24 25 26M 四种常见类型。

SYSBOOT[4:0]	Boot Sequence			
CONTROL_STATUS[4:0]				
	1st	2nd	3rd	4th
00000b	Reserved			
00001b	UART0	XIP w/ WAIT[1] (MUX2)[2]	MMC0	SPI0
00010b	UART0	SPI0	NAND	NANDI2C
00011b	UART0	SPI0	XIP (MUX2)[2]	MMC0

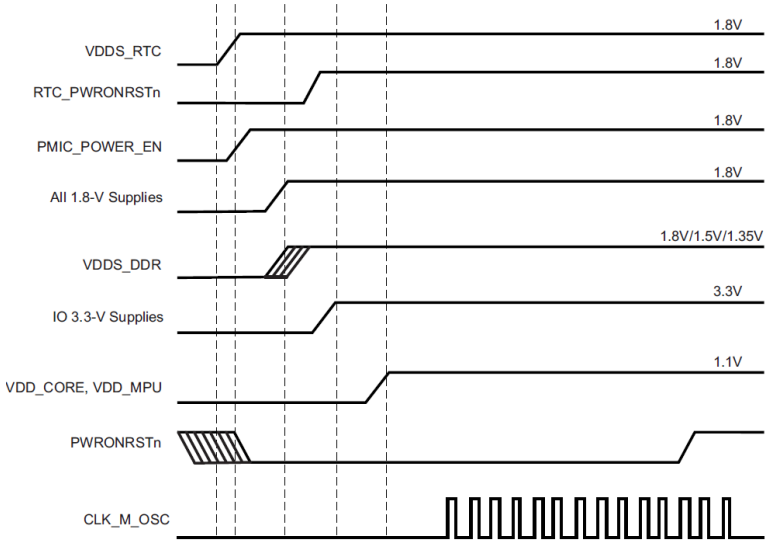
图一

SYSBOOT[15:14]	SYSBOOT[13:12]
For all boot modes: Crystal Frequency	For all boot modes: Set to 00b for normal operation
CONTROL_STATUS[23:22]	CONTROL_STATUS[21:20]
00b = 19.2MHz 01b = 24MHz 10b = 25MHz 11b = 26MHz	00b (all other values reserved)
00b = 19.2MHz 01b = 24MHz 10b = 25MHz 11b = 26MHz	00b (all other values reserved)
00b = 19.2MHz 01b = 24MHz 10b = 25MHz 11b = 26MHz	00b (all other values reserved)
00b = 19.2MHz 01b = 24MHz 10b = 25MHz 11b = 26MHz	00b (all other values reserved)

图二

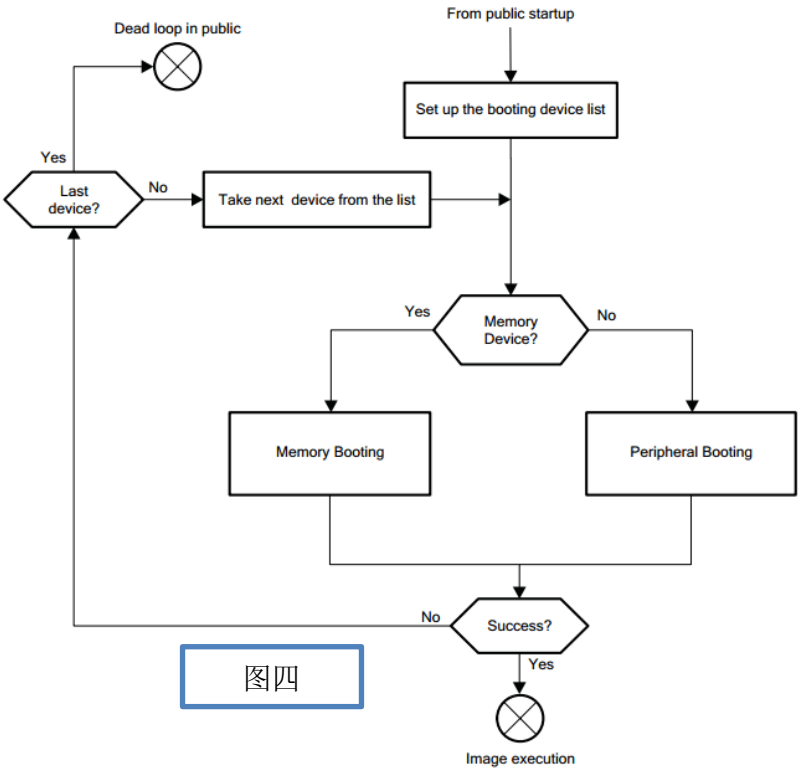
四 程

根据 am335x 的芯片技术手册，获取图三，此图为芯片上电时序要求。满足该要求方法有：可以选用特定的 PMIC，也可以使用分离电源芯片，但必须严格要求上电时序。而 PMIC 在设计上简单，可参考 TI 的电源芯片设计。



图三

Figure 26-2. Public ROM Code Boot Procedure



图四

从图四分析 rom_code 的执行流程：

芯片符合上电时序后或复位，PC 指针首先跳转到 0x20000 地址开始执行。此时出厂固化在内部 ROM 的 rom_code 进行芯片的简单初始化，如看门狗和 NAND、NOR、SD 卡、UART、SPI 和 USB 等符合引导类型的外设。rom_code 主要功能是：

1.根据 SYSBOOT 的配置生成引导设备列表

2.查看当前的引导类型是 memory 模式还是外设模式，并通过初始化相应的接口或地址读取 SPL，可能性如下：

若 SPL 读取成功后，执行 SPL 镜像，SPL 开始运行

若 SPL 读取失败，尝试启动列表中的下一个外设或接口

若所有外设都没有正确读取到 SPL，则进入死循环（dead loop）等待看门狗复位

四．关于内存运行的位置

首先先看一下 am335x 内部 176KB 的 rom 和内部 64 KB ram 的结构如下：

Figure 26-3. ROM Memory Map

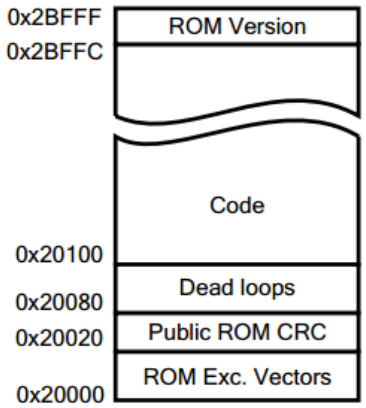
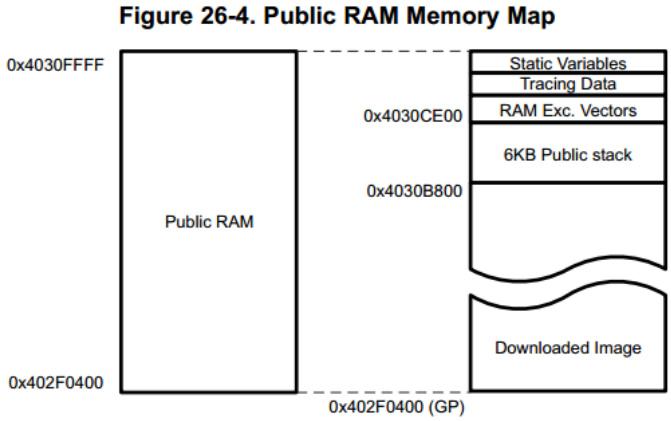


Figure 26-4. Public RAM Memory Map



ROM_core 就是运行在左上图的位置，而右上图标志的 Dowloaded Image 区域：是用来保存 MLO（SPL） 镜像文件的，其最大可达到 **109 KB。**

关于 bootload 每个阶段的起始位置如下表所示：

addr	说明
0x20000	rom_code 的运行起始地址
0x402f0400	SPL 镜像在内部 SRAM 中的加载地址，也是内部 ram 的起始地址
0x80000000	U-bool 镜像在 DDR 中的加载地址，也是内部 DDR 的起始地址

当 SPL 已经加载到芯片的内部 SRAM 中了，那么它是从哪儿开始运行的呢？这需要查看 spl 在被链接的时候所需的链接文件 **u-boot-spl.lds**，该文件决定了 spl 和 uboot 从哪里开始运行。查看该文件可以发现其中有一句话：**CPUDIR/start.o (.text*)**，表示了 start.s 作为该起始函数。一开始会设置 CUP 模式和中断，操作的寄存器是 CPSW，这个是一个很重要的寄存器，以下是对 CPSW 的寄存器内容进行描述。最重要的是低四位 M0-M4，其决定 CPU 八种模式。

31	30	29	28	27 ~ 8	7	6	5	4	3	2	1	0
N	Z	C	V	保留	I	F	T	M4	M3	M2	M1	M0

N	Negative/Less Than	I	IRQ disable
Z	Zero	F	FIQ disable
C	Carry/Borrow/Extend	T	State bit
V	Overflow	M0~4	Mode bits

M[4：0]	处理器模式	ARM模式可访问的寄存器	THUMB模式可访问的寄存器
0b10000	用户模式	PC,CPSR,R0~R14	PC,CPSR,R0~R7,LR,SP
0b10001	FIQ模式	PC,CPSR,SPSR_fiq,R14_fiq~R8_fiq,R0~R7	PC,CPSR,SPSR_fiq,LR_fiq,SP_fiq,R0~R7
0b10010	IRQ模式	PC,CPSR,SPSR_irq,R14_irq~R13_irq,R0~R12	PC,CPSR,SPSR_irq,LR_irq,SP_irq,R0~R7
0b10011	管理模式	PC,CPSR,SPSR_svc,R14_svc~R13_svc,R0~R12	PC,CPSR,SPSR_svc,LR_svc,SP_svc,R0~R7
0b10111	中止模式	PC,CPSR,SPSR_abt,R14_abt~R13_abt,R0~R12	PC,CPSR,SPSR_abt,LR_abt,SP_abt,R0~R7
0b11011	未定义模式	PC,CPSR,SPSR_und,R14_und~R13_und,R0~R12	PC,CPSR,SPSR_und,LR_und,SP_und,R0~R7
0b11111	系统模式	PC,CPSR,R0~R14	PC,CPSR,LR,SP,R0~R74

五．以下是 spl 和 Uboot 的流程图，格式：左边是函数执行顺序，中间是分析函数内容：右边是进一步分析。

程序流程

函数解析

进一步解析

SPL 开始

Start.s
(arch\arm\cpu\armv7)

通过对 CPSR 操作
禁止 interrupts (FIQ and IRQ)
设置 cpu 于 SVC32 mode，即管理模式，上图有相关描述

CPSR:程序状态寄存器(当前程序状态寄存器)，在任何处理器模式下被访问。它包含了条件标志位、中断禁止位、当前处理器模式标志以及其他的一些控制和状态位。上图已有描述。

Start.s
运行 cpu_init_cp15

初始化 L1 的 I/D caches
禁止 MMU

Start.s
运行 cpu_init_crit

跳转到 lowlevel_init

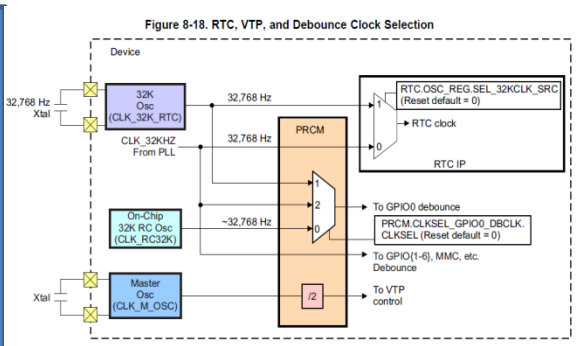
lowlevel_init.S
(arch\arm\cpu\armv7)

设置暂时的堆栈（为了调用 C 函数）
SP 指向 CONFIG_SYS_INIT_SP_ADDR
确保 sp 是 8 字节对齐
保存当前 ip 指针，跳到 s_init，

CONFIG_SYS_INIT_SP_ADDR 在 Ti_armv7_common.h (include\configs)定义，值为 0x40310000-224=0x4030ff20，224 是指 global data 的大小

s_init
Board.c(arch\arm\cpu\armv7\am33xx)

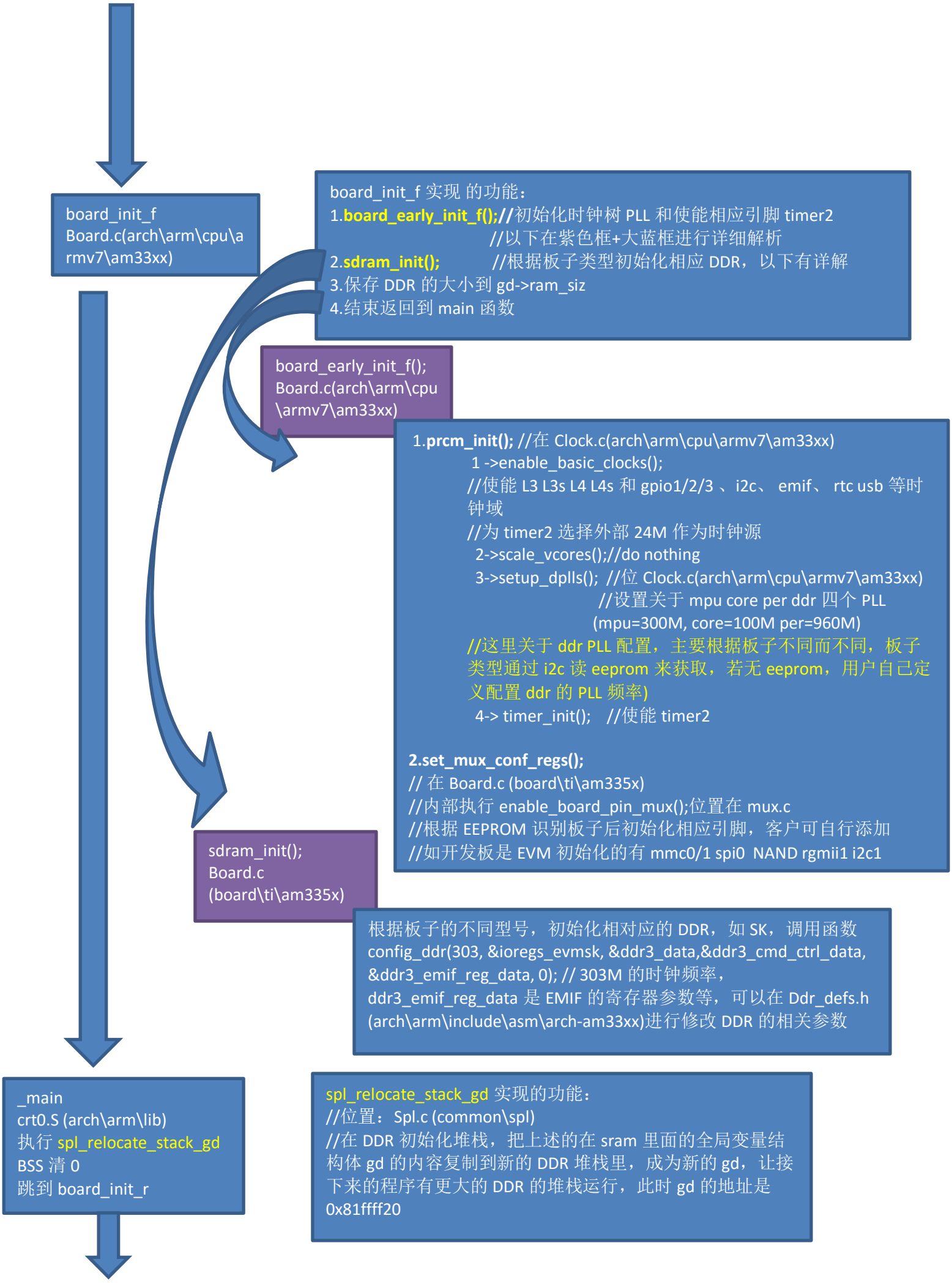
S_ini 实现的功能：
1.禁止看门狗
2.使能 UART0 相关引脚（可通过 define 来改其他 UART1/2/3/4/5）
3.使能与 UART 相关时钟树和功能模块
4.软件复位 UART
5.选择 rtc32k OSC（设置 RTC.OSC_REG.SEL=1 选择 32k 外部时钟源，如右图所示。并允许时钟源进入 RTC 模块）



回到文件 Start.s
运行_main
crt0.S (arch\arm\lib)

_main 功能:
1.重新对 SP 赋值
SP 指向 CONFIG_SYS_INIT_SP_ADDR
确保 sp 是 8 字节对齐
2.运行 board_init_f_alloc_reserve
3.运行 board_init_f_init_reserve
4.跳到 board_init_f

board_init_f_alloc_reserve 主要是为 u-boot 的全局变量结构体 global data（struct global_data）分配空间，起始地址是 0x4030fa40
运行 board_init_f_init_reserve 主要将 global data 清 0 操作
这两函数位置都在文件 Board_init.c (common\init)


board_init_f
Board.c(arch\arm\cpu\armv7\am33xx)
board_init_f 实现 的功能：
1.board_early_init_f();//初始化时钟树 PLL 和使能相应引脚 timer2
//以下在紫色框+大蓝框进行详细解析
2.sdram_init(); //根据板子类型初始化相应 DDR，以下有详解
3.保存 DDR 的大小到 gd->ram_siz
4.结束返回到 main 函数
board_early_init_f();
Board.c(arch\arm\cpu\armv7\am33xx)
1.prcm_init(); //在 Clock.c(arch\arm\cpu\armv7\am33xx)
1 ->enable_basic_clocks();
//使能 L3 L3s L4 L4s 和 gpio1/2/3 、i2c、 emif、 rtc usb 等时钟域
//为 timer2 选择外部 24M 作为时钟源
2->scale_vcores();//do nothing
3->setup_dplls(); //位 Clock.c(arch\arm\cpu\armv7\am33xx)
//设置关于 mpu core per ddr 四个 PLL
(mpu=300M, core=100M per=960M)
//这里关于 ddr PLL 配置，主要根据板子不同而不同，板子类型通过 i2c 读 eeprom 来获取，若无 eeprom，用户自己定义配置 ddr 的 PLL 频率)
4-> timer_init(); //使能 timer2
2.set_mux_conf_regs();
// 在 Board.c (board\ti\am335x)
//内部执行 enable_board_pin_mux();位置在 mux.c
//根据 EEPROM 识别板子后初始化相应引脚，客户可自行添加
//如开发板是 EVM 初始化的有 mmc0/1 spi0 NAND rgmii1 i2c1
sdram_init();
Board.c
(board\ti\am335x)
根据板子的不同型号，初始化相对应的 DDR，如 SK，调用函数 config_ddr(303, &ioregs_evmsk, &ddr3_data,&ddr3_cmd_ctrl_data, &ddr3_emif_reg_data, 0); // 303M 的时钟频率，ddr3_emif_reg_data 是 EMIF 的寄存器参数等，可以在 Ddr_defs.h (arch\arm\include\asm\arch-am33xx)进行修改 DDR 的相关参数
_main
crt0.S (arch\arm\lib)
执行 spl_relocate_stack_gd
BSS 清 0
跳到 board_init_r
spl_relocate_stack_gd 实现的功能：
//位置：Spl.c (common\spl)
//在 DDR 初始化堆栈，把上述的在 sram 里面的全局变量结构体 gd 的内容复制到新的 DDR 堆栈里，成为新的 gd，让接下来的程序有更大的 DDR 的堆栈运行，此时 gd 的地址是 0x81ffff20

void board_init_r
(gd_t *dummy1,
ulong dummy2)
// Spl.c (common\spl)

1.对 gd 结构体里的参数进行各种赋值，保存 SDRAM 大小到 gd
2.gd 的 TLB 映射(TLB table from bfffc000 to c0000000)
3.使能 CP15 的 D-caches
4.设置大小为 0x1000000 的堆栈，从 0x80a80000 开始
5.执行 void spl_board_init(void) //函数解析如下方所示
6.board_boot_order(spl_boot_list);//将 gt 的 boot-device 赋值给 spl_boot_list[0]
7.announce_boot_device(spl_boot_list[i]);//终端输出启动 device 的名字
8.执行 spl_load_image(spl_boot_list[i])//从相关 device 载入 image 到 DDR 内
//函数解析如下方所示
9. cleanup_before_linux();
10.jump_to_image_no_args(&spl_image);
位置：Boot-common.c (arch\arm\cpu\armv7\omap-common)
->image_entry((u32 *)boot_params); 跳转到 uboot 的入口地址
spl_image->entry_point 执行，结束了 SPL 的过程。

void spl_board_init(void)

Boot-common.c (arch\arm\cpu\armv7\omap-common)

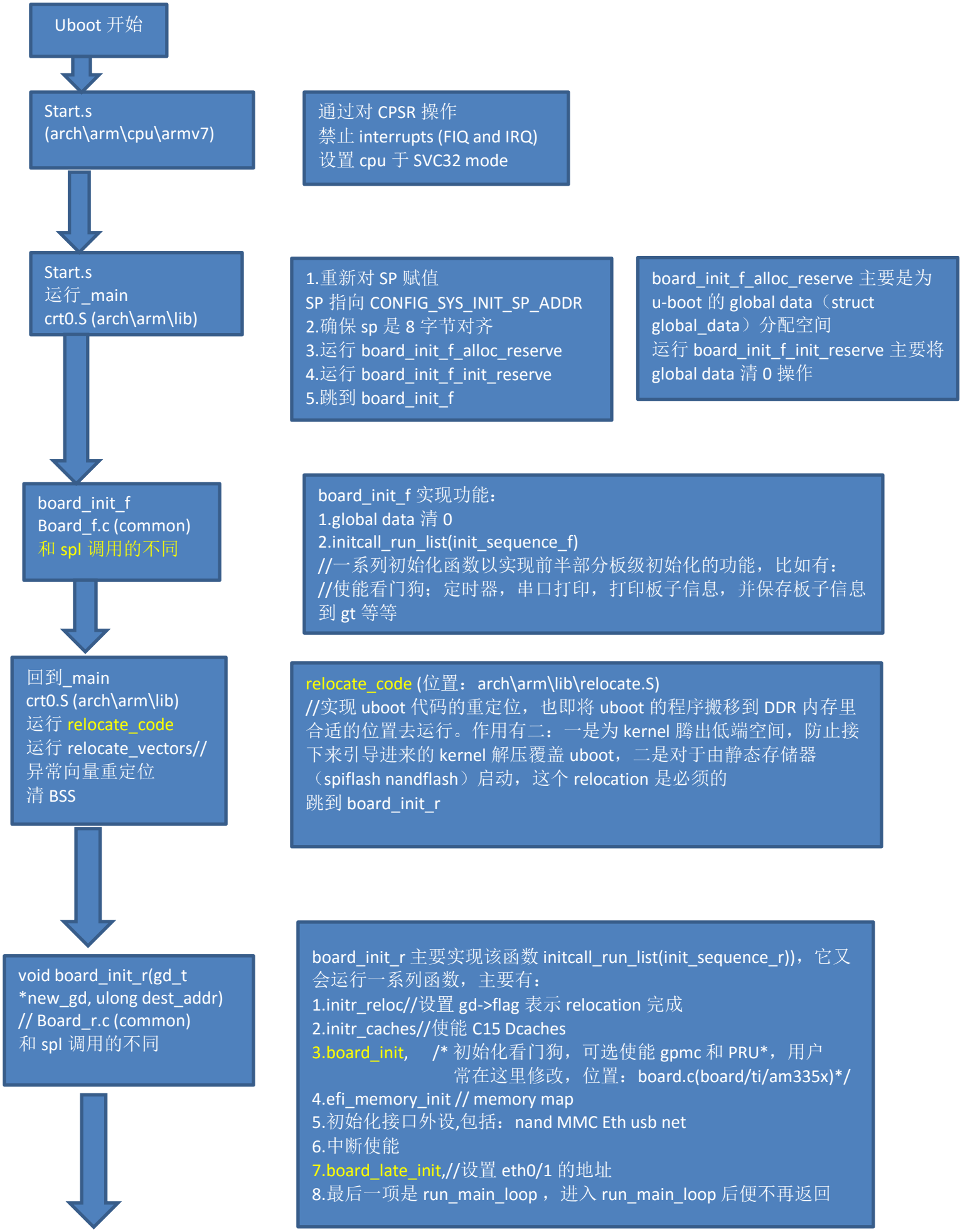
Table 9-59. efuse_sma Register Field Descriptions

Bit	Field	Type	Reset	Description
31-18	Reserved	R		These bits are undefined and contents can vary from device to device.
17-16	package_type	R	Package-dependent	Designates the Package type of the device (PG2.x only). 00b - Undefined 01b - ZCZ Package 10b - ZCE Package 11b - Reserved
15-13	Reserved	R		These bits are undefined and contents can vary from device to device.
12-0	arm_mpu_max_freq	R	Device-dependent	Designates the ARM MPU Maximum Frequency supported by the device (PG2.x only). There are also voltage requirements that accompany each frequency (OPPs). See the device specific data manual for this information and for information on device variants. 0x1FEF - 300 MHz ARM MPU Maximum (ZCZ Package only) 0x1FAF - 600 MHz ARM MPU Maximum (ZCZ Package only) 0x1F2F - 720 MHz ARM MPU Maximum (ZCZ Package only) 0x1E2F - 800 MHz ARM MPU Maximum (ZCZ Package only) 0x1C2F - 1 GHz ARM MPU Maximum (ZCZ Package only) 0x1FDF - 300 MHz ARM MPU Maximum (ZCE Package only) 0x1F9F - 600 MHz ARM MPU Maximum (ZCE Package only) All other values are reserved.

1.save_omap_boot_params(); //保存启动的 device 和 mode
2.preloader_console_init();//保存波特率于 gd，开启串口功能
//此函数可以在 s_init 后随意调用，前提是 gd 有效
3.gpmc_init(); //若有 define 则初始化 gpmc
4.i2c_init(100k, 1); //若有 define 则初始化 i2c 100k 速率
5.arch_misc_init(); //若有 define 则初始化 usb
6.hw_watchdog_init(); //若 define 则初始化看门狗
7.am33xx_spl_board_init();//在 board.c(board\ti\am335x),
//通过函数 am335x_get_efuse_mpu_max_freq(cdev)
//来读取 efuse_sma 寄存器的低 13 位来判别芯片 MPU 的最大频率 Fmax（如左图所示），这点很重要。接着初始化 PMIC，使得芯片工作电压满足最高频率 。 最后设置 Core 频率=1G 和 Mpu 最大频率 Fmax

spl_load_image(spl_boot_list[i]) 这里选择 MMC 加载 image:，即执行函数
spl_mmc_load_image(boot_device) //位于在 Spl_mmc.c (common\spl)

spl_mmc_load_image 里判断 MMC 是否支持，获取 mode=MMCSD_MODE_FS，执行以下:
-> spl_mmc_do_fs_boot(mmc)
-> if (!spl_start_uboot()) 执行
-> spl_load_image_fat_os(&mmc->block_dev, CONFIG_SYS_MMCSD_FS_BOOT_PARTITION);
-> file_fat_read(CONFIG_SPL_FS_LOAD_ARGS_NAME, (void *)CONFIG_SYS_SPL_ARGS_ADDR, 0);
（在这个函数中会试图读取 “args” 文件到地址 CONFIG_SYS_SPL_ARGS_ADDR(=0x80F80000)中，如果这个文件不存在，则退出这个函数，继续执行 spl_mmc_do_fs_boot(mmc)
-> spl_load_image_fat(&mmc->block_dev,CONFIG_SYS_MMCSD_FS_BOOT_PARTITION, //=1
CONFIG_SPL_FS_LOAD_PAYLOAD_NAME); //=u-boot.image
-> file_fat_read(filename, header, sizeof(struct image_header));
（在这个函数中会试图读取 “u-boot.image” header 文件到 0x807fffc0 中，大小 64）
->spl_load_simple_fit(&load, 0, header);
(会调用三次 spl_fit_read 函数，此函数调用 fat_read_file(filename, buf, file_offset, size, &actread)去读 “u-boot.image” 文件偏移量不同的地方)


Uboot 开始
Start.s
(arch\arm\cpu\armv7)
通过对 CPSR 操作
禁止 interrupts (FIQ and IRQ)
设置 cpu 于 SVC32 mode
Start.s
运行_main
crt0.S (arch\arm\lib)
1.重新对 SP 赋值
SP 指向 CONFIG_SYS_INIT_SP_ADDR
2.确保 sp 是 8 字节对齐
3.运行 board_init_f_alloc_reserve
4.运行 board_init_f_init_reserve
5.跳到 board_init_f
board_init_f_alloc_reserve 主要是为 u-boot 的 global data（struct global_data）分配空间
运行 board_init_f_init_reserve 主要将 global data 清 0 操作
board_init_f
Board_f.c (common)
和 spl 调用的不同
board_init_f 实现功能：
1.global data 清 0
2.initcall_run_list(init_sequence_f)
//一系列初始化函数以实现前半部分板级初始化的功能，比如有：
//使能看门狗；定时器，串口打印，打印板子信息，并保存板子信息到 gt 等等
回到_main
crt0.S (arch\arm\lib)
运行 relocate_code
运行 relocate_vectors//
异常向量重定位
清 BSS
relocate_code (位置：arch\arm\lib\relocate.S)
//实现 uboot 代码的重定位，也即将 uboot 的程序搬移到 DDR 内存里合适的位置去运行。作用有二：一是为 kernel 腾出低端空间，防止接下来引导进来的 kernel 解压覆盖 uboot，二是对于由静态存储器（spiflash nandflash）启动，这个 relocation 是必须的
跳到 board_init_r
void board_init_r(gd_t *new_gd, ulong dest_addr)
// Board_r.c (common)
和 spl 调用的不同
board_init_r 主要实现该函数 initcall_run_list(init_sequence_r))，它又会运行一系列函数，主要有：
1.initr_reloc//设置 gd->flag 表示 relocation 完成
2.initr_caches//使能 C15 Dcaches
3.board_init, /* 初始化看门狗，可选使能 gpmc 和 PRU*，用户常在这里修改，位置：board.c(board/ti/am335x)*/
4.efi_memory_init // memory map
5.初始化接口外设,包括：nand MMC Eth usb net
6.中断使能
7.board_late_init,//设置 eth0/1 的地址
8.最后一项是 run_main_loop，进入 run_main_loop 后便不再返回

run_main_loop // Board_r.c (common)

for (;;)
main_loop();

main_loop()
//位置 Main.c (common)
//分析如右框所示：

bootstage_mark_name//调用了 show_boot_progress，利用它显示启动进程(progress)，此处为空函数。

cli_init();//调用 u_boot_hush_start();使用 hush shell 来作为执行器。hush shell 是一种轻量型的 shell。cli_init 用来初始化 hush shell 使用的一些变量

run_preboot_environment_command 该函数如果定义 CONFIG_PREBOOT(默认不定义)则从环境变量中获取"preboot"的内容，然后使用 run_command_list 启动该命令

s = bootdelay_process();从环境变量中取出"bootdelay"和"bootcmd"的配置值，将取出的"bootdelay"配置值转换成整数，赋值给全局变量 stored_bootdelay，最后返回"bootcmd"的配置值，赋值给 s。

bootdelay 为 u-boot 的启动延时计数值，计数期间内若无用户按键输入干预，那么将执行"bootcmd"配置中的命令。

由于没有定义 CONFIG_OF_CONTROL，函数 cli_process_fdt 返回 false，接下来执行 autoboot_command(s);该函数会先等待 bootdelay 这么长的时间，若无用户干预，则函数执行 run_command_list(s, -1, 0); 该 s 就是上述的"bootcmd"，

run_command_list 中调用了 hush shell 的命令解释器(parse_stream_outer 函数)，解释 bootcmd 中的启动命令。此处的环境变量 bootcmd 中的启动命令为：
bootcmd="run findfdt; run init_console; run envboot; run distro_bootcmd, 该命令主要选出 fdt 类型，保存串口打印配置，也即用来设置 linux 必要的启动环境，最后一条命令是轮询的从 MMC NAND 等外设去启动 kernel。u-boot 启动 linux 内核后，将控制权交给 linux 内核，至此不再返回。

若在 bootdelay 期间有用户输入时，则 autoboot_command(s)不执行 run_command_list(s, -1, 0);直接跳出函数

接着运行 cli_loop(); 该函数运行 int parse_file_outer(void)；

->执行：parse_stream_outer(&input, FLAG_PARSE_SEMICOLON);

->执行：do-while 会循环命令解析器的"命令输入解析--执行"运行模式

循环的读取用户输入的命令，执行其中的函数 run_list 执行如下的函数调用流程：
run_list-->run_list_real-->run_pipe_real
最后在函数 run_pipe_real 中有：
return cmd_process(...);//完成 u-boot 命令的定位和执行,具体如下：
函数 cmd_process 调用 find_cmd 查找到命令名对应的 cmd_tbl_t 结构体变量后，cmd_process 接下来将调用函数 cmd_call 执行 cmd_tbl_t 中的命令，如果命令执行的返回值为 CMD_RET_USAGE，代表命令执行出错，且置标 CMD_RET_USAGE，那么将调用 cmd_usage，输出简短的命令使用帮助信息

以上就是 SPL 和 U-boot 的流程分析，还是得再次强调一次，这次分析主要是以”全而简”（全过程+简单分析）的角度进行介绍，**若在细节上有任何疑问，可以根据我画的流程图方便查找和定位，若仍然解决不了，欢迎大家可以在论坛上随时随地提出来。谢谢！**

以下第六章额外附加两幅关于内存分布的图，希望能够帮助大家理解关键时刻的 DDR 的内存分布

六. 以下框图表示不同时间段 DDR 内存里的内容分布