U-BOOT启动过程

本文描述的是开发板上电到uboot启动内核的过程。以socfpga_cyclone5为例，主要分为三个阶段：**rom固件、spl、uboot**，不同的开发板启动流程会稍微有些差别，但是大体上是类似的。

一、第一阶段：rom固件

1、rom固件概述

对于arm处理器来说，cpu上电时会从0x0地址处开始执行，因此通常soc厂商开始会将rom映射到0x0地址，里面放着相应的部分初始化代码，运行完这段代码将控制权交给下一阶段的代码后，由下一阶段将0x0地址处开始的地址重新映射到别的设备，一般是片上sram。

2、rom固件启动流程

2.1 一般rom固件大体流程

在系统上电后，rom固件首先获得控制权，此时需要做一些初始化才能进行接下来的工作。对于普通开发板，可以通过上电之前在板子上的相应开关上进行设置来决定初始化的参数，主要为时钟启动源的设置。固件代码会读取相应的硬件引脚或者开关设置来获得这些信息，然后配置相应的时钟和I/O引脚，和相应的闪存控制器。如果选择从sd卡启动则配置的是sd卡相关的I/O和初始化sd卡控制器，如果是从flash启动则配置的是flash相关的I/O和初始化flash控制器。在做完相关初始化后可能还会将某些启动参数传给下一阶段的spl代码，此时双方会遵守相应的规范在指定的地方按照指定的格式存放，一般是片上sram某块指定区域。最后从闪存(sd卡或flash)中指定的位置将下一阶段的spl镜像加载到片上sram中，然后验证镜像，最后跳转到spl第一条代码处运行spl，之后控制权就到spl手上了。**需要注意的是：**在启动前要把spl镜像烧写到闪存中指定的地址，这些地址在soc手册中描述。

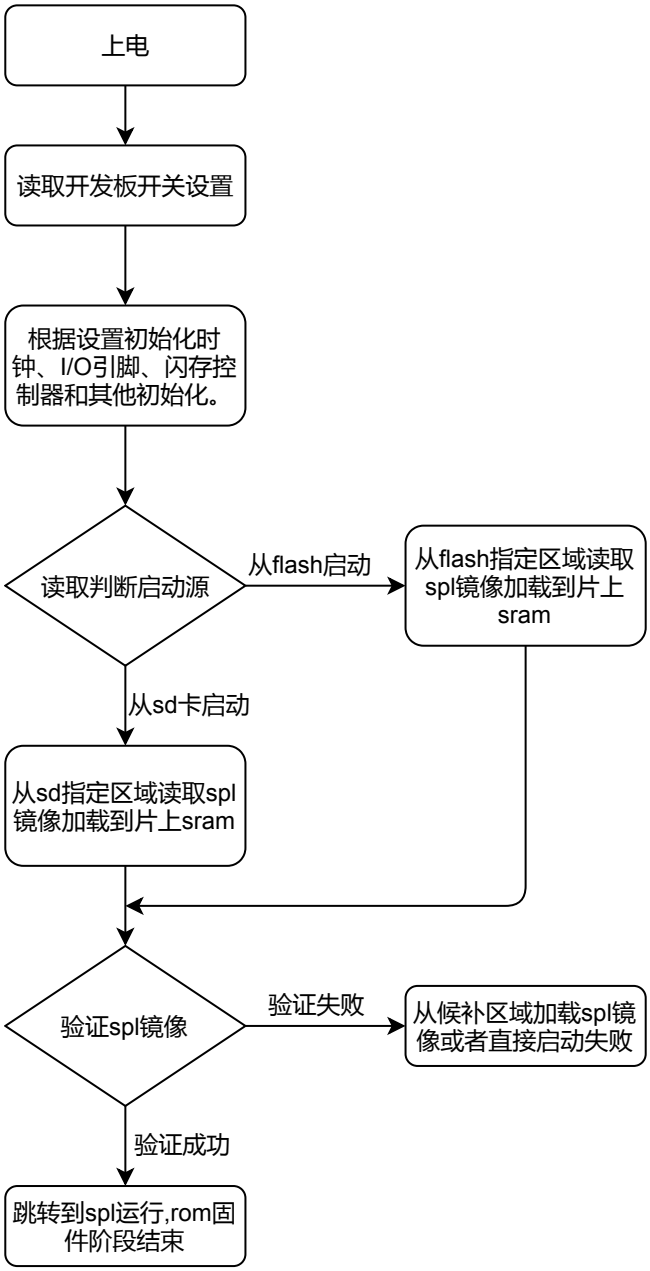

图1 一般的rom固件启动流程

2.2 socfpga_cyclone5的rom固件

socfpga_cyclone5的ROM 代码可以执行片上RAM 的热引导、器件的FPGA 部分的冷引导或闪存的冷引导。

器件的FPGA 部分的冷引导期间，引导ROM 代码等待直到FPGA 准备就绪，然后试图通过HPS- to- FPGA 桥接在地址0x0 直接执行。例如，引导软件可能会由器件的FPGA 部分中的地址0x0 的初始化的片上RAM 提供。

闪存的冷引导期间，引导ROM 代码试图将第一个预加载器镜像从闪存加载到片上RAM 并且将控制传递到预加载器。如果该镜像无效，那么引导ROM 代码试图从闪存加载接下来的3 个镜像。如果在接下来的加载中仍然没有有效镜像，那么引导ROM 代码检查器件的FPGA 部分是否存在一个备用镜像。

片上RAM 的热引导期间，引导ROM代码读取系统管理器中romcodegrp组的预加载器状态寄存器(initswstate)，以便决定片上RAM中是否具有有效预加载器镜像。如果片上RAM中具有有效预加载器镜像，那么引导ROM代码跳过从闪存加载预加载器镜像，而是将控制传递到片上RAM中的预加载器。如果片上RAM 中没有有效预加载器镜像，那么引导ROM 代码试图加载从闪存加载的最后一个有效预加载器镜像(由系统管理器romcodegrp组的初始软件最后镜像

加载寄存器(initswlastld) 的index 域识别)。如果镜像无效，那么引导ROM 代码试图从闪存加载接下来的3 个镜像。如果片上RAM 或闪存中不存在有效预加载器镜像，那么引导ROM代码检查器件的FPGA 部分是否有备用镜像。

复位后片上RAM（总共64KB）顶部4KB为引导ROM代码而保留，该区域包括共享内存，引导ROM代码将共享内存的位置传递到SPL的寄存器r0

引导ROM 执行以下操作来初始化HPS：

- 使能指令缓存、分支预测器、浮点单元和NEON 矢量单元
- 设置level 4 (l4) 看门狗0 计时器
- 根据CSEL 值配置主PLL 和外设PLL
- 根据BSEL 值配置I/O 单元和管脚复用
- 以默认设置初始化闪存控制器

当引导ROM 代码准备将控制传递到预加载器时，处理器(CPU0) 处于以下状态：

- 指令缓存被使能
- 分支预测器被使能
- 数据缓存被禁用
- **MMU被禁用**
- 浮点单元被使能
- NEON矢量单元被使能
- 处理器处于ARM安全监督模式
- **r0包含共享内存模块的指针，可用于将信息从引导ROM代码传递到SPL,共享内存模块位于片上RAM的顶部4KB**
- r1包含共享内存的长度
- **引导ROM仍被映射到地址0x0**
- L4 看门狗0 计时器有效

二、第二阶段：spl

1、spl概述

spl其实是uboot第一阶段，之所以要把uboot拆成spl和uboot第二阶段的原因是rom固件还没有初始化通用内存ddr，rom空间有限因此要将这一任务交给spl。此时只有一个几十kb的片上sram，无法将一个上百kb的uboot整个加载到片上sram上运行，因此rom固件先将一个片上sram可以容纳的spl镜像加载到片上sram，然后运行spl，最后由spl初始化了ddr之后再将uboot第二阶段加载到ddr中运行。

2、spl主要工作

由于不同硬件平台的spl的工作稍有区别，但大体类似。这里以socfpga_cyclone的spl为例描述spl的主要工作：

- 初始化内存控制器和DDR内存芯片
- 将片上RAM映射到地址0x0，以便SPL处理异常情况（片上RAM在地址0xFFFF0000也是可访问的。地址0x0是一个别名）
- 通过扫描管理器配置HPS I/O（在引导ROM期间根据BSEL有一个预先配置）
- 通过系统管理器配置管脚复用（在引导ROM期间根据BSEL有一个预先配置）
- 通过时钟管理器配置HPS时钟（在引导ROM期间根据CSEL有一个主PLL和外设PLL预先配置）
- 初始化包含下一个阶段引导软件uboot的闪存控制器（在引导ROM期间有一个默认值的预先配置）
- 将下一个阶段引导软件uboot加载到DDR 并且将控制权传递到uboot

值得注意的是上述的spl的工作其中有一部分在rom固件中有过预先配置，但是rom固件的配置较为简陋，比如初始化不完全或者配置参数不是spl以后阶段想要的，因此spl会重新和补充初始化。

3、spl流程分析

3.1 spl入口

在分析spl流程之前需要大概了解spl的布局和找到spl的入口，链接脚本是找到这些信息的关键。以下是spl的链接脚本，完整路径名是 UBOOT根目录/spl/u-boot-spl.lds。

```
 MEMORY { .sram : ORIGIN = 0xFFFF0000, LENGTH = (64 * 1024) }
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 .text :
 {
  arch/arm/cpu/armv7/start.o (.text)
  *(.text*)
 } >.sram
 . = ALIGN(4);
 .rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) } >.sram
 . = ALIGN(4);
 .data : { *(SORT_BY_ALIGNMENT(.data*)) } >.sram
 __ecc_padding_start = .;
 . = ALIGN(4);
 __image_copy_end = .;
 _end = .;
 .bss : {
  . = ALIGN(4);
  __bss_start = .;
  *(.bss*)
  . = ALIGN(4);
  __bss_end__ = .;
 } >.sram
 . = ALIGN(8);
 .malloc :
 {
  __malloc_start = .;
  . = . + (5 * 1024);
  __malloc_end = .;
 } >.sram
 .stack :
 {
  . = . + (4 * 1024);
  . = ALIGN(8);
  __stack_start = .;
  . = . + 0x8;
  __ecc_padding_end = .;
 } >.sram
}
```

以下是编译spl过程中的部分打印信息。

```
cd /home/hn/TEST/u-boot-altera-2012.10/spl/ && arm-linux-gnueabihf-ld.bfd  -T
/home/hn/TEST/u-boot-altera-2012.10/spl/u-boot-spl.lds --gc-sections -Bstatic -Ttext
0xFFFF0000 arch/arm/cpu/armv7/start.o...
```

从连接脚本spl/u-boot-spl.lds和编译输出信息中的-Ttext 0xFFFF0000中可以得到SPL入口在_start,运行时应该在地址0xFFFF0000（片上RAM的起始地址总会映射到这里，还有可能有另外映射地址）处。

3.2 spl流程路线

spl的整体流程：
_start --> reset --> save_boot_params(什么都没做) --> 设置管理模式，禁止IRQ和FIQ中断 --> cpu_init_cp15 --> cpu_init_crits --> lowlevel_init --> s_init --> call_board_init_f --> board_init_f --> board_init_r

各阶段详细分析：

3.2.1 _start --> reset --> save_boot_params(什么都没做) --> 设置管理模式，禁止IRQ和FIQ中断 --> cpu_init_cp15 --> cpu_init_crit(arch\arm\cpu\armv7\start.S)

```
.globl _start
_start: b   reset

.....................................................

#ifdef CONFIG_SPL_BUILD
.globl  save_timer_value
save_timer_value:
    .word  0xabcdabcd
#endif

reset:
/* 使用OSC1计时器1测试SPL到U-BOOT的时间  */
#ifdef CONFIG_SPL_BUILD
    bl osc_timer_reset_release  @ 将OSC1定时器1从复位释放
    bl osc_timer_config         @ 将0xffffffff写入计数器中，屏蔽OSC1定时器1中断，自由模式，使能OSC1
定时器1（复位后是disabled状态）
    bl osc_timer_get            @ 获取osc1timer1当前值读入r0
    adr r1, save_timer_value    @ 将save_timer_value标号的地址读到r1
    str r0, [r1]                @ 将得到的osc1timer1当前值写到save_timer_value处
#endif


bl  save_boot_params
/* 设置为svc管理模式，禁止IRQ和FIQ中断 */
mrs r0, cpsr        @ 将cpsr寄存器读到r0中
bic r0, r0, #0x1f   @ 将读到r0中的cpsr副本前5位清零
orr r0, r0, #0xd3   @ 打开r0中的cpsr副本第0，1，4，6，7位
msr cpsr,r0         @ 将处理好后的值写入cpsr寄存器中
```

```
.....................................................

/* SPL会执行，U-BOOT阶段不执行 */
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl  cpu_init_cp15
    bl  cpu_init_crit
#endif

.....................................................

ENTRY(cpu_init_cp15)
    /*
     * Invalidate L1 I/D
     */
    mov r0, #0          @ 设置MCR协处理器
    mcr p15, 0, r0, c8, c7, 0   @ 失效 TLBs
    mcr p15, 0, r0, c7, c5, 0   @ 失效 icache
    mcr p15, 0, r0, c7, c5, 6   @ 失效 分支预测
    mcr p15, 0, r0, c7, c10, 4  @ DSB，数据同步屏障，内存和缓存，tlb等维护操作完成才进行后面指令的操
作
    mcr p15, 0, r0, c7, c5, 4   @ ISB，指令同步屏障，清空流水线

    /*
     * disable MMU stuff and caches
     */
    mrc p15, 0, r0, c1, c0, 0   @ 将协处理器p15的系统控制寄存器读到r0中
    bic r0, r0, #0x00002000 @ clear bits 13 (--V-)  @ 异常向量表映射到0x00000000
    bic r0, r0, #0x00000007 @ clear bits 2:0 (-CAM) @ 关闭mmu，关闭对齐检查，关闭数据和统一缓存
    orr r0, r0, #0x00000002 @ set bit 1 (--A-) Align        @ 打开对齐检查
    orr r0, r0, #0x00000800 @ set bit 11 (Z---) BTB     @ 打开分支预测

    orr r0, r0, #0x00001000 @ set bit 12 (I) I-cache        @ 使能指令缓存

    mcr p15, 0, r0, c1, c0, 0   @写入协处理器p15的系统控制寄存器中
    mov pc, lr          @ back to my caller
ENDPROC(cpu_init_cp15)

ENTRY(cpu_init_crit)
    /*
     * Jump to board specific initialization...
     * The Mask ROM will have already initialized
     * basic memory. Go here to bump up clock rate and handle
     * wake up conditions.
     */
    b   lowlevel_init       @ 设置pll,mux,sdram
ENDPROC(cpu_init_crit)
```

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

```
ENTRY(lowlevel_init)
    /*
     * 设置一个临时栈
     */
    ldr sp, =CONFIG_SYS_INIT_SP_ADDR    @ 0xFFFF0000 + 0x10000 - 0x100 -
GENERATED_GBL_DATA_SIZE(一定要在片上ram中)
    bic sp, sp, #7      @ 将栈指针8字节对齐
    push    {ip, lr}    @ 将老的lr和当前lr保存在临时栈中，当前lr指向call_board_init_f，是bl
cpu_init_crit时设置的
    bl  s_init          @ 去设置pl,mux,sdram
    pop {ip, pc}        @ 返回到cpu_init_crit下一条指令执行（call_board_init_f）
ENDPROC(lowlevel_init)
```

3.2.3 s_init(arch\arm\cpu\armv7\socfpga\s_init.c)

```
/* 第一个c函数 */
void s_init(void)
{
    unsigned long reg;
    /*
     * First C code to run. Clear fake OCRAM ECC first as SBE
     * and DBE might triggered during power on
     */
    reg = readl(CONFIG_SYSMGR_ECC_OCRAM);   //On-chip RAM ECC Enable Register
    if (reg & SYSMGR_ECC_OCRAM_SERR)    //如果不为0则代表有单个位的On-chip RAM ECC错误中断，可纠
正，由硬件设置，但需要写1清除中断标志
        writel(SYSMGR_ECC_OCRAM_SERR | SYSMGR_ECC_OCRAM_EN,
            CONFIG_SYSMGR_ECC_OCRAM);   //使能On-chip RAM ECC功能，并清除single bit ecc错误中断
    if (reg & SYSMGR_ECC_OCRAM_DERR)    //如果不为0则代表有双位的On-chip RAM ECC错误中断，不可纠
正，由硬件设置，需要写1清除中断标志
        writel(SYSMGR_ECC_OCRAM_DERR  | SYSMGR_ECC_OCRAM_EN,
            CONFIG_SYSMGR_ECC_OCRAM);   //使能On-chip RAM ECC功能，并清除double bit ecc错误中断

    /*
     * 配置remap(L3 NIC-301 GPV)然后on-chip RAM会映射到0x0，取消ROM映射0x0
     * 同时使能 HPS2FPGA and LWHPS2FPGA
     */
    writel(0x19,SOCFPGA_L3REGS_ADDRESS);

    /* re-setup watchdog */
    DEBUG_MEMORY
    if (!(is_wdt_in_reset())) {     //如果不在复位状态则执行
        /*
         * only disabled if wdt not in reset state
         * disable the watchdog prior PLL reconfiguration
         */
        DEBUG_MEMORY
        watchdog_disable();         //先关闭osc1wd0（l4wd0）后打开
    }

#ifdef CONFIG_HW_WATCHDOG
    /* release osc1 watchdog timer 0 from reset */
```

```
    DEBUG_MEMORY
    reset_deassert_osc1wd0();   //将osc1wd0（l4wd0）从复位状态释放

    /* reconfigure and enable the watchdog */
    DEBUG_MEMORY
    hw_watchdog_init();     //在drivers\watchdog\designware_wdt.c，使能l4wd0，刷新l4wd0的计数器重新计数

    WATCHDOG_RESET();       //再次刷新l4wd0的计数器从新计数
#endif /* CONFIG_HW_WATCHDOG */

    DEBUG_MEMORY
    /* 在连接脚本中定义，用来数据对齐设置 */
    if (&__ecc_padding_start < &__ecc_padding_end) {
        memset(&__ecc_padding_start, 0,
            &__ecc_padding_end - &__ecc_padding_start);
    }
}
```

3.2.4 call_board_init_f(arch\arm\cpu\armv7\start.S)

```
call_board_init_f:
    ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
    bic sp, sp, #7 /* 8-byte alignment for ABI compliance */
    ldr r0,=0x00000000
    bl  board_init_f
```

3.2.5 board_init_f(arch\arm\lib\spl.c)

```
void __weak board_init_f(ulong dummy)
{
    /* Set the stack pointer. */
    asm volatile("mov sp, %0\n" : : "r"(CONFIG_SPL_STACK));     //在连接脚本中__stack_start

    /* Clear the BSS. */
    memset(__bss_start, 0, __bss_end__ - __bss_start);  //清bss段

    /* Set global data pointer. */
    gd = &gdata;        //gd是存在r8寄存器当中的，通过宏DECLARE_GLOBAL_DATA_PTR

    saved_cpu0_global_data_ptr = (u32)gd;   //将gd写在start.S中的saved_cpu0_global_data_ptr地址处
    asm volatile("dsb");

    board_init_r(NULL, 0);
}
```

3.2.6 board_init_r(common\spl\spl.c)

该函数是spl最后部分，也是工作量最大的部分，主线是spl_board_init -> spl_mmc_probe -> spl_mmc_load_image -> jump_to_image_no_args。按照主线分为三个阶段：

(1) **spl_board_init**的板级初始化阶段，主要有初始化时钟，引脚复用和物理特性配置，串口初始化，sdram控制器初始化和ddr内存芯片初始化，中断向量设置。

(2) **spl_mmc_probe()和spl_mmc_load_image()**的初始化sd/mmc和加载uboot提取uboot头部信息。

(3) **jump_to_image_no_args()**的spl最后一个阶段，跳转到u-boot启动运行

```
void board_init_r(gd_t *dummy1, ulong dummy2)
{
    u32 boot_device;
    debug(">>spl:board_init_r()\n");

    mem_malloc_init(CONFIG_SYS_SPL_MALLOC_START,
            CONFIG_SYS_SPL_MALLOC_SIZE);    //将__malloc_start和__malloc_end之间的内存清零

    timer_init();   //加载0xffffffff新计数，用户计数模式，使能osc1timer1

    spl_board_init();   //初始化时钟，引脚复用，串口，sdram，中断向量

    boot_device = spl_boot_device();    //获取boot device类型(BOOT_DEVICE_MMC1)
    debug("boot device - %d\n", boot_device);

    switch (boot_device) {  //加载启动bm

    case BOOT_DEVICE_MMC1:
    case BOOT_DEVICE_MMC2:
    case BOOT_DEVICE_MMC2_2:
        amp_share_param_init();         //0x1E100000,u-boot, bare metal, linux的共享内存区域
地址

        load_bm_start();                //获取加载bm.bin的开始时间

        spl_mmc_probe();                //初始化sd/mmc控制器和设备

        spl_mmc_load_bm_image_mbr();    //将bm.bin从sd/mmc加载到sdram中

        load_bm_end();                  //获取加载bm.bin的结束时间

        spl_mmc_save_func();            //保存spl_mmc_probe等函数到共享内存给bm使用

        boot_bm_on_cpu1();              //从cpu1启动bm

        cpu0_wait_cpu1_load_rbf();      //等待cpu1中运行的bm加载rbf完成

        spl_mmc_load_image();           //将u-boot镜像加载到ddr中

        break;

    default:
        debug("SPL: Un-supported Boot Device\n");
        hang();
    }

    switch (spl_image.os) {     //spl_image在spl_mmc_load_image()里填充
```

```
    case IH_OS_U_BOOT:
        debug("Jumping to U-Boot\n");
        break;

    default:
        debug("Unsupported OS image.. Jumping nevertheless..\n");
    }

    jump_to_image_no_args();        //跳转到u-boot启动运行
}
```

3.2.6.1 spl_board_init (arch\arm\cpu\armv7\socfpga\spl.c)

```
void spl_board_init(void)
{
    cm_config_t cm_default_cfg = {      //main,Peripheral和sdram pll时钟组设置数据
        /* main group */
        MAIN_VCO_BASE,                          //M=63+1,N=0+1 , main_vco_clk = 64eosc1_clk
        CLKMGR_MAINPLLGRP_MPUCLK_CNT_SET(
            CONFIG_HPS_MAINPLLGRP_MPUCLK_CNT),  //mpu_clk = main_vco_clk/2 = 32eosc1_clk
        CLKMGR_MAINPLLGRP_MAINCLK_CNT_SET(
            CONFIG_HPS_MAINPLLGRP_MAINCLK_CNT), //main_clk = l3_main_clk = l4_main_clk =
main_vco_clk/4 = 16eosc1_clk
        CLKMGR_MAINPLLGRP_DBGATCLK_CNT_SET(
            CONFIG_HPS_MAINPLLGRP_DBGATCLK_CNT),    //dbg_base_clk = 16eosc1_clk
        CLKMGR_MAINPLLGRP_MAINQSPICLK_CNT_SET(
            CONFIG_HPS_MAINPLLGRP_MAINQSPICLK_CNT), //main_qspi_clk = main_vco_clk/4 =
16eosc1_clk
        CLKMGR_PERPLLGRP_PERNANDSDMMCCLK_CNT_SET(
            CONFIG_HPS_MAINPLLGRP_MAINNANDSDMMCCLK_CNT),    //main_nand_sdmmc_clk =
main_vco_clk/4 = 16eosc1_clk
        CLKMGR_MAINPLLGRP_CFGS2FUSER0CLK_CNT_SET(
            CONFIG_HPS_MAINPLLGRP_CFGS2FUSER0CLK_CNT),  //cfg_s2f_user0_clk =
main_vco_clk/16 = 4eosc1_clk
        CLKMGR_MAINPLLGRP_MAINDIV_L3MPCLK_SET(
            CONFIG_HPS_MAINPLLGRP_MAINDIV_L3MPCLK) |    //l3_mp_clk = l3_main_clk/(value+1)
= l3_main_clk/2 = 8eosc1_clk
        CLKMGR_MAINPLLGRP_MAINDIV_L3SPCLK_SET(
            CONFIG_HPS_MAINPLLGRP_MAINDIV_L3SPCLK) |    //l3_sp_clk = l3_mp_clk/(value+1) =
l3_mp_clk/2 = 4eosc1_clk
        CLKMGR_MAINPLLGRP_MAINDIV_L4MPCLK_SET(
            CONFIG_HPS_MAINPLLGRP_MAINDIV_L4MPCLK) |    //l4_mp_clk = periph_base_clk/2
        CLKMGR_MAINPLLGRP_MAINDIV_L4SPCLK_SET(
            CONFIG_HPS_MAINPLLGRP_MAINDIV_L4SPCLK),     //l4_sp_clk = periph_base_clk/2
        CLKMGR_MAINPLLGRP_DBGDIV_DBGATCLK_SET(
            CONFIG_HPS_MAINPLLGRP_DBGDIV_DBGATCLK) |    //dbg_at_clk = dbg_base_clk
        CLKMGR_MAINPLLGRP_DBGDIV_DBGCLK_SET(
            CONFIG_HPS_MAINPLLGRP_DBGDIV_DBGCLK),       //dbg_clk = dbg_at_clk/2 =
dbg_base_clk/2
        CLKMGR_MAINPLLGRP_TRACEDIV_TRACECLK_SET(
            CONFIG_HPS_MAINPLLGRP_TRACEDIV_TRACECLK),   //dbg_trace_clk = dbg_base_clk
        CLKMGR_MAINPLLGRP_L4SRC_L4MP_SET(
            CONFIG_HPS_MAINPLLGRP_L4SRC_L4MP) |     //设置l4_mp_clk的时钟源为periph_base_clk
```

```
        CLKMGR_MAINPLLGRP_L4SRC_L4SP_SET(
            CONFIG_HPS_MAINPLLGRP_L4SRC_L4SP),      //设置l4_sp_clk的时钟源为periph_base_clk

        /* peripheral group */
        PERI_VCO_BASE,          //设置peripheral PLL组的时钟源为eosc1_clk，peri_vco_clk=
(79+1)eosc1_clk/(1+1) = 40eosc1_clk
        CLKMGR_PERPLLGRP_EMAC0CLK_CNT_SET(
            CONFIG_HPS_PERPLLGRP_EMAC0CLK_CNT),
//emac0_clk=emac0_base_clk=peri_vco_clk/(3+1)=peri_vco_clk/4 = 10eosc1_clk
        CLKMGR_PERPLLGRP_EMAC1CLK_CNT_SET(

CONFIG_HPS_PERPLLGRP_EMAC1CLK_CNT),//emac1_clk=emac0_base_clk=peri_vco_clk/(3+1)=peri_vco_cl
k/4 = 10eosc1_clk
        CLKMGR_PERPLLGRP_PERQSPICLK_CNT_SET(
            CONFIG_HPS_PERPLLGRP_PERQSPICLK_CNT),//periph_qspi_base_clk=peri_vco_clk/2 =
20eosc1_clk
        CLKMGR_PERPLLGRP_PERNANDSDMMCCLK_CNT_SET(
            CONFIG_HPS_PERPLLGRP_PERNANDSDMMCCLK_CNT),//periph_nand_sdmmc_clk=peri_vco_clk/5
= 8eosc1_clk
        CLKMGR_PERPLLGRP_PERBASECLK_CNT_SET(
            CONFIG_HPS_PERPLLGRP_PERBASECLK_CNT),//periph_base_clk=peri_vco_clk/5 =
8eosc1_clk
        CLKMGR_PERPLLGRP_S2FUSER1CLK_CNT_SET(
            CONFIG_HPS_PERPLLGRP_S2FUSER1CLK_CNT),//h2f_user1_base_clk=peri_vco_clk/10 =
4eosc1_clk
        CLKMGR_PERPLLGRP_DIV_USBCLK_SET(
            CONFIG_HPS_PERPLLGRP_DIV_USBCLK) |//usb_mp_clk=periph_base_clk=peri_vco_clk/5 =
8eosc1_clk
        CLKMGR_PERPLLGRP_DIV_SPIMCLK_SET(
            CONFIG_HPS_PERPLLGRP_DIV_SPIMCLK) |//spi_m_clk=periph_base_clk=peri_vco_clk/5 =
8eosc1_clk
        CLKMGR_PERPLLGRP_DIV_CAN0CLK_SET(
            CONFIG_HPS_PERPLLGRP_DIV_CAN0CLK) |//can0_clk=periph_base_clk/2=peri_vco_clk/10
= 4eosc1_clk
        CLKMGR_PERPLLGRP_DIV_CAN1CLK_SET(
            CONFIG_HPS_PERPLLGRP_DIV_CAN1CLK),//can1_clk=periph_base_clk/2=peri_vco_clk/10 =
4eosc1_clk
        CLKMGR_PERPLLGRP_GPIODIV_GPIODBCLK_SET(

CONFIG_HPS_PERPLLGRP_GPIODIV_GPIODBCLK),//gpio_db_clk=periph_base_clk/6250=peri_vco_clk/3125
0 = 4eosc1_clk/3125
        CLKMGR_PERPLLGRP_SRC_QSPI_SET(
            CONFIG_HPS_PERPLLGRP_SRC_QSPI) |//设置qspi_clk的时钟源为main_qspi_clk
        CLKMGR_PERPLLGRP_SRC_NAND_SET(
            CONFIG_HPS_PERPLLGRP_SRC_NAND) |//设置nand_clk的时钟源为periph_nand_sdmmc_clk
        CLKMGR_PERPLLGRP_SRC_SDMMC_SET(
            CONFIG_HPS_PERPLLGRP_SRC_SDMMC),//设置sdmmc_clk的时钟源为periph_nand_sdmmc_clk

        /* sdram pll group */
        SDR_VCO_BASE,   //设置sdram PLL组的时钟源为eosc1_clk，sdr_vco_clk=
(63+1)eosc1_clk/(1+1)=32eosc1_clk
        CLKMGR_SDRPLLGRP_DDRDQSCLK_PHASE_SET(
            CONFIG_HPS_SDRPLLGRP_DDRDQSCLK_PHASE) | //设置ddr_dqs_base_clk和sdr_voc_clk同一相位
```

```
        CLKMGR_SDRPLLGRP_DDRDQSCLK_CNT_SET(
            CONFIG_HPS_SDRPLLGRP_DDRDQSCLK_CNT),    //ddr_dqs_clk=sdr_vco_clk/2 =
16eosc1_clk
        CLKMGR_SDRPLLGRP_DDR2XDQSCLK_PHASE_SET(
            CONFIG_HPS_SDRPLLGRP_DDR2XDQSCLK_PHASE) |   //设置ddr_2x_dqs_clk和sdr_vco_clk同一
相位
        CLKMGR_SDRPLLGRP_DDR2XDQSCLK_CNT_SET(
            CONFIG_HPS_SDRPLLGRP_DDR2XDQSCLK_CNT),  //ddr_2x_dqs_clk = sdr_vco_clk/2 =
16eosc1_clk
        CLKMGR_SDRPLLGRP_DDRDQCLK_PHASE_SET(
            CONFIG_HPS_SDRPLLGRP_DDRDQCLK_PHASE) |  //设置ddr_dq_clk的相位比sdr_voc_clk增加
4*45°
        CLKMGR_SDRPLLGRP_DDRDQCLK_CNT_SET(
            CONFIG_HPS_SDRPLLGRP_DDRDQCLK_CNT),     //ddr_dq_clk = sdr_vco_clk/2 =
16eosc1_clk
        CLKMGR_SDRPLLGRP_S2FUSER2CLK_PHASE_SET(
            CONFIG_HPS_SDRPLLGRP_S2FUSER2CLK_PHASE) |   //设置s2f_user2_clk和sdr_vco_clk同一相
位
        CLKMGR_SDRPLLGRP_S2FUSER2CLK_CNT_SET(
            CONFIG_HPS_SDRPLLGRP_S2FUSER2CLK_CNT),      ////s2f_user2_clk = sdr_vco_clk/2 =
16eosc1_clk
    };

    WATCHDOG_RESET();       //使能l4wd0，刷新计数

    debug("Freezing all I/O banks\n");
    /* freeze all IO banks */
    sys_mgr_frzctrl_freeze_req(FREEZE_CHANNEL_0,
        FREEZE_CONTROLLER_FSM_SW);  //冻结通道0的bank7D和bank7E
    sys_mgr_frzctrl_freeze_req(FREEZE_CHANNEL_1,
        FREEZE_CONTROLLER_FSM_SW);  //冻结通道1的bank7B和bank7C
    sys_mgr_frzctrl_freeze_req(FREEZE_CHANNEL_2,
        FREEZE_CONTROLLER_FSM_SW);  //冻结通道2的bank7A
    sys_mgr_frzctrl_freeze_req(FREEZE_CHANNEL_3,
        FREEZE_CONTROLLER_FSM_SW);  //冻结通道3的bank6

    WATCHDOG_RESET();   //使能l4wd0，刷新计数

    debug("Asserting reset to all except L4WD and SDRAM\n");
    reset_assert_all_peripherals_except_l4wd0_osc1timer1(); //除l4wd0和osc1timer1外的外设复位置
1，使pll重配置期间没有跳变
    reset_assert_all_bridges();     //三个桥复位置1



    debug("Deassert reset for OSC1 Timer\n");
    /*
     * deassert reset for osc1timer0. We need this for delay
     * function that required during PLL re-configuration
     */
    reset_deassert_osc1timer0();    //osc1timer0复位置0

    debug("Init timer\n");
```

```
    /* init timer for enabling delay function */
    timer_init();   //加载0xffffffff新计数，用户计数模式，使能osc1timer1


#ifndef CONFIG_SOCFPGA_VIRTUAL_TARGET

    WATCHDOG_RESET();   //使能l4wd0，刷新计数

    debug("Reconfigure Clock Manager\n");
    cm_basic_init(&cm_default_cfg);     //利用上文设置好的数据配置main pll,Peripheral PLL,和SDRAM
PLL三个时钟组的时钟

    WATCHDOG_RESET();   //使能l4wd0，刷新计数

    /* Skip configuration is warm reset happen and WARMRSTCFGIO set */

    if (((readl(CONFIG_SYSMGR_ROMCODEGRP_CTRL) &
        SYSMGR_ROMCODEGRP_CTRL_WARMRSTCFGIO) == 0) ||
        ((rst_mgr_status & RSTMGR_WARMRST_MASK) == 0)) {    //如果warmrstcfgio置1则跳过配置IO

        /* Enable handshake bit with BootROM */
        setbits_le32(CONFIG_SYSMGR_ROMCODEGRP_CTRL,
            SYSMGR_ROMCODEGRP_CTRL_WARMRSTCFGIO);   //warmrstcfgio置1,引导rom将会在热复位后配置IO

        debug("Configure IOCSR\n");

        /* 启动启动扫描管理器对HPS I/O配置位移寄存器进行配置 */
        scan_mgr_io_scan_chain_prg(
            IO_SCAN_CHAIN_0,
            CONFIG_HPS_IOCSR_SCANCHAIN0_LENGTH,
            iocsr_scan_chain0_table);
        scan_mgr_io_scan_chain_prg(
            IO_SCAN_CHAIN_1,
            CONFIG_HPS_IOCSR_SCANCHAIN1_LENGTH,
            iocsr_scan_chain1_table);
        scan_mgr_io_scan_chain_prg(
            IO_SCAN_CHAIN_2,
            CONFIG_HPS_IOCSR_SCANCHAIN2_LENGTH,
            iocsr_scan_chain2_table);
        scan_mgr_io_scan_chain_prg(
            IO_SCAN_CHAIN_3,
            CONFIG_HPS_IOCSR_SCANCHAIN3_LENGTH,
            iocsr_scan_chain3_table);

        /* Clear handshake bit with BootROM */
        DEBUG_MEMORY
        clrbits_le32(CONFIG_SYSMGR_ROMCODEGRP_CTRL,
            SYSMGR_ROMCODEGRP_CTRL_WARMRSTCFGIO);   //warmrstcfgio置0,引导rom不会在热复位后配置IO

    }

    WATCHDOG_RESET();   //使能l4wd0，刷新计数
```

```
    /* Skip configuration is warm reset happen and WARMRSTCFGPINMUX set */
    if (((readl(CONFIG_SYSMGR_ROMCODEGRP_CTRL) &
        SYSMGR_ROMCODEGRP_CTRL_WARMRSTCFGPINMUX) == 0) ||
        ((rst_mgr_status & RSTMGR_WARMRST_MASK) == 0)) {    //如果warmrstcfgpinmux置1则跳过配置
pinmux



        /* Enable handshake bit with BootROM */
        setbits_le32(CONFIG_SYSMGR_ROMCODEGRP_CTRL,
            SYSMGR_ROMCODEGRP_CTRL_WARMRSTCFGPINMUX);       //warmrstcfgpinmux置1,引导rom将会
在热复位后配置引脚复用

        debug("Configure PINMUX\n");
        sysmgr_pinmux_init();       //通过系统管理器配置管脚复用

        /* Clear handshake bit with BootROM */
        DEBUG_MEMORY
        clrbits_le32(CONFIG_SYSMGR_ROMCODEGRP_CTRL,
            SYSMGR_ROMCODEGRP_CTRL_WARMRSTCFGPINMUX);   //warmrstcfgpinmux置0,引导rom不会在热复
位后配置引脚复用

    }


    WATCHDOG_RESET();   //使能l4wd0，刷新计数

    debug("Deasserting resets\n");
    /* de-assert reset for peripherals and bridges based on handoff */

    reset_deassert_peripherals_handoff();   //peripherals复位置1？不是置0码？（在boot_bm_on_cpu1
中释放）暖复位握手fpga,etr,sdram

    reset_deassert_bridges_handoff();       //bridges复位置1？不是置0码？（在boot_bm_on_cpu1中释
放）


    WATCHDOG_RESET();       //使能l4wd0，刷新计数

    debug("Unfreezing/Thaw all I/O banks\n");
    /* unfreeze / thaw all IO banks */
    sys_mgr_frzctrl_thaw_req(FREEZE_CHANNEL_0,
        FREEZE_CONTROLLER_FSM_SW);  //解冻通道0的bank7D和bank7E
    sys_mgr_frzctrl_thaw_req(FREEZE_CHANNEL_1,
        FREEZE_CONTROLLER_FSM_SW);  //解冻通道1的bank7B和bank7C
    sys_mgr_frzctrl_thaw_req(FREEZE_CHANNEL_2,
        FREEZE_CONTROLLER_FSM_SW);  //解冻通道2的bank7A
    sys_mgr_frzctrl_thaw_req(FREEZE_CHANNEL_3,
        FREEZE_CONTROLLER_FSM_SW);  //解冻通道3的bank6


    WATCHDOG_RESET();   //使能l4wd0，刷新计数

    /* enable console uart printing */
```

```
    preloader_console_init();    //common\spl\spl.c,初始化串口控制器，波特率57600，然后可以用串口在spl阶段调试


    WATCHDOG_RESET();    //使能l4wd0，刷新计数

    puts("SDRAM : Initializing MMR registers\n");
    /* SDRAM MMR initialization */
    if (sdram_mmr_init_full() != 0)      //初始化SDRAM控制器
        hang();


    WATCHDOG_RESET();    //使能l4wd0，刷新计数


    puts("SDRAM : Calibrationg PHY\n");
    /* SDRAM calibration */
    if (sdram_calibration_full() == 0)  //校准sdram phy
        hang();

    WATCHDOG_RESET();    //使能l4wd0，刷新计数

    /* configure the interconnect NIC-301 security */
    nic301_slave_ns();  //arch\arm\cpu\armv7\socfpga\nic301.c
                           设置只有非安全主机才能访问LWHPS2FPGA从机，HPS2FPGA从机，MPU ACP从机，ROM从机,ON CHIP RAM从机，SDRAM DATA从机

    WATCHDOG_RESET();    //使能l4wd0，刷新计数

    init_boot_params();      //boot_params_ptr = 0,片上ram的开始地址
}
```

3.2.6.2 amp_share_param_init到spl_mmc_load_image的初始化sd/mmc和加载镜像阶段详细分析见另一个文档bm_load_boot.pdf。

这个阶段主要关注的是`spl_mmc_probe()`和`spl_mmc_load_image()`,其他的是裸机程序和fpga相关的，是定制方案相关的，不是uboot启动主线。其中`spl_mmc_probe()`初始化sd/mmc控制器和sd/mmc设备，`spl_mmc_load_image()`将uboot第二阶段的镜像加载到ddr中。我们重点关注`spl_mmc_load_image()`内部做了什么：

```
void spl_mmc_load_image(void)
{
    int err;
    u32 boot_mode;

    boot_mode = spl_boot_mode();    //MMCSD_MODE_MBR
    if (boot_mode == MMCSD_MODE_RAW) {
        debug("boot mode - RAW\n");
        mmc_load_image_raw(dw_mmc);
    } else if (boot_mode == MMCSD_MODE_MBR) {
        debug("boot mode - MBR\n");
        mmc_load_image_mbr(dw_mmc);     //从sd卡加载u-boot镜像到ddr中，并分析头部
```

```
    } else {
        puts("spl: wrong MMC boot mode\n");
        hang();
    }
}
```

`spl_mmc_load_image()`中再调用`mmc_load_image_mbr()`来真正的加载uboot镜像到ddr中，并分析头部提取信息。需要注意的是其中头部大小为64字节，以16进制表示为0x40，包含头部的uboot被加载到0x01000000地址上，因此真正的uboot在地址0x01000040上。所以加载地址为0x1000000，入口地址为0x01000040，即uboot第一条执行的代码地址。

```
static void mmc_load_image_mbr(struct mmc *mmc)
{
    s32 err;
    u32 image_size_sectors;
    const struct image_header *header;

    header = (struct image_header *)(CONFIG_SYS_TEXT_BASE -
                        sizeof(struct image_header));   //0x0100 0040 - 0x40 = 0x0100 0000

    /* read image header to find the image size & load address */
    err = mmc->block_dev.block_read(0,
            (offset + CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR), 1,
            (void *)header);        //将sd卡中的u-boot镜像第一个扇区读到0x0100 0000地址中去

    if (err <= 0)
        goto end;

    spl_parse_image_header(header);     //解析u-boot镜像头部，主要获取加载地址，入口地址和镜像长度写入
spl_image中
                                        //加载地址：头部的起始地址0x1000000
                                    //入口地址：payload的起始地址0x1000040（从头部后的第一个节）
                                       // 镜像长度：头部加payload的长度0x3b124字节
    /* convert size to sectors - round up */
    image_size_sectors = (spl_image.size + mmc->read_bl_len - 1) /
                mmc->read_bl_len;   //将镜像长度从字节单位转换为扇区（512字节）单位0x1d9

    /* Read the header too to avoid extra memcpy */
    err = mmc->block_dev.block_read(0,
            (offset + CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR),
            image_size_sectors, (void *)spl_image.load_addr);       //将u-boot镜像从sd卡读到
sdram加载地址上

end:
    if (err <= 0) {
        printf("spl: mmc blk read err - %d\n", err);
        hang();
    }
}
```

3.2.6.3 jump_to_image_no_args(common\spl\spl.c)。spl最后一个阶段，跳转到u-boot启动运行

```
static void __noreturn jump_to_image_no_args(void)
{
    typedef void __noreturn (*image_entry_noargs_t)(u32 *);
    image_entry_noargs_t image_entry =
            (image_entry_noargs_t) spl_image.entry_point;   //获取u-boot入口点地址（u-boot第一条指令的地址）

    WATCHDOG_RESET();       //使能l4wd0，刷新计数

    u32 calculated_crc;
    if (spl_image.crc_size != 0) {
        debug("Verifying Checksum ... ");
        calculated_crc = crc32_wd(0,
            (unsigned char *)spl_image.entry_point,
            spl_image.crc_size, CHUNKSZ_CRC32);
        if (calculated_crc != spl_image.crc) {
            puts("Bad image with mismatched CRC\n");
            debug("CRC calculate from 0x%08x "
                "with length 0x%08x\n",
                spl_image.entry_point, spl_image.size);
            debug("CRC Result : Expected 0x%08x "
                "Calculated 0x%08x\n",
                spl_image.crc, calculated_crc);
            hang();
        } else
            debug("OK\n");
    }

    debug("image entry point: 0x%X\n", spl_image.entry_point);

    /* to indicate a successful run */
    writel(CONFIG_PRELOADER_STATE_VALID, CONFIG_PRELOADER_STATE_REG);   //将魔数写入initswstate寄存器中，代表preloder完成加载下一阶段镜像到sdram的任务

    u32 boot_params_ptr_addr = (u32)&boot_params_ptr;   //boot_params_ptr=0

    WATCHDOG_RESET();   //使能l4wd0，刷新计数

    image_entry((u32 *)boot_params_ptr_addr);   //跳转到u-boot入口点地址启动u-boot，spl结束

}
```

三、第三阶段：uboot

1、uboot概述

大部分的硬件初始化已经在spl阶段做了，uboot要做的是剩下的少部分硬件初始化以及一些软件层面的工作。这些硬件初始化主要是因为调试的需要的串口和网口的初始化，而软件层面的工作主要是内存划分，重定位，设备注册以及最后加载内核和启动内核。

2、uboot入口

从链接脚本u-boot.lds的开始片段中可以得到U-BOOT入口在_start,运行时应该在地址0x00000000，但是经过makefile的设置，从编译输出打印信息可以看到真正的连接地址是0x01000040（在SPL阶段将片上RAM映射到0x00000000，另外片上ram还会映射到0xffff0000）。

```
链接脚本
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 . = 0x00000000;
 . = ALIGN(4);
 .text :
 {
  __image_copy_start = .;
  arch/arm/cpu/armv7/start.o (.text)
  *(.text)
 }
 . = ALIGN(4);
 .rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }
 . = ALIGN(4);
 .data : {
  *(.data)
 }
 . = ALIGN(4);
 . = .;
 __u_boot_cmd_start = .;
 .u_boot_cmd : { *(.u_boot_cmd) }
 __u_boot_cmd_end = .;
 . = ALIGN(4);
 __image_copy_end = .;
 .rel.dyn : {
  __rel_dyn_start = .;
  *(.rel*)
  __rel_dyn_end = .;
 }
 .dynsym : {
  __dynsym_start = .;
  *(.dynsym)
 }
 _end = .;
 . = ALIGN(4096);
 .mmutable : {
  *(.mmutable)
 }
 .bss __rel_dyn_start (OVERLAY) : {
  __bss_start = .;
  *(.bss)
   . = ALIGN(4);
  __bss_end__ = .;
 }
```

```
 /DISCARD/ : { *(.dynstr*) }
 /DISCARD/ : { *(.dynamic*) }
 /DISCARD/ : { *(.plt*) }
 /DISCARD/ : { *(.interp*) }
 /DISCARD/ : { *(.gnu*) }
}
```

编译打印信息

```
cd /home/hn/TEST/u-boot-altera-2012.10 && arm-linux-gnueabihf-ld.bfd  -pie -T u-boot.lds -Bstatic -Ttext 0x01000040 $UNDEF_SYM arch/arm/cpu/armv7/start.o ......
```

2.1 uboot流程路线

uboot的整体流程：

_start-->reset-->save_boot_params(什么都没做)-->设置管理模式，禁止IRQ和FIQ中断-->board_init_f-->relocate_code -->board_init_r-->main_loop()

各阶段详细分析：

2.2.1 _start --> reset --> save_boot_params(什么都没做) --> 设置管理模式，禁止IRQ和FIQ中断

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

```
.globl _start
_start: b   reset

.....................................................

reset:
    bl  save_boot_params
    /* 设置为svc管理模式，禁止IRQ和FIQ中断 */
    mrs r0, cpsr        @ 将cpsr寄存器读到r0中
    bic r0, r0, #0x1f   @ 将读到r0中的cpsr副本前5位清零
    orr r0, r0, #0xd3   @ 打开r0中的cpsr副本第0，1，4，6，7位
    msr cpsr,r0         @ 将处理好后的值写入cpsr寄存器中
```

2.2.2 board_init_f(0)(arch\arm\lib\board.c)

```
void board_init_f(ulong bootflag)
{
    bd_t *bd;
    init_fnc_t **init_fnc_ptr;
    gd_t *id;
    ulong addr, addr_sp;



    bootstage_mark_name(BOOTSTAGE_ID_START_UBOOT_F, "board_init_f");

    /* Pointer is writable since we allocated a register for it */
    gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) & ~0x07);  //gd指向的区域起始地址，0xffff0000到该地址前用作临时栈
```

```
    /* compiler optimization barrier needed for GCC >= 3.4 */
    __asm__ __volatile__("": : :"memory");

    memset((void *)gd, 0, sizeof(gd_t));        //初始化gd区域

    gd->mon_len = _bss_end_ofs;     //__bss_end__ - _start

    /* Allow the early environment to override the fdt address */
    gd->fdt_blob = (void *)getenv_ulong("fdtcontroladdr", 16,
                        (uintptr_t)gd->fdt_blob);       //从环境变量中得到设备树地址

    for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {     //大部分初始化工作
在这里
        if ((*init_fnc_ptr)() != 0) {
            hang ();
        }
    }

    /*
    init_fnc_t *init_sequence[] = {
        arch_cpu_init,      //什么都没做，返回0
        board_early_init_f, //关闭l4wd0看门狗
        timer_init,         //加载新计数，使能osc1timer1
        env_init,           //gd->env_addr = (ulong)&default_environment[0]; gd->env_valid =
1;(common\env_mmc.c)
        init_baudrate,      //gd->baudrate = getenv_ulong("baudrate", 10, CONFIG_BAUDRATE) =
57600
        serial_init,        //根据gd的波特率设置串口0的控制器(drivers\serial\serial.c)
        console_init_f,     //gd->have_console = 1;可以用printf,puts等打印函数了
        display_banner,     //打印点东西
        print_cpuinfo,      //打印CPU型号信息
        checkboard,         //打印板型号信息
        dram_init,          //gd->ram_size = get_ram_size(0x0, 0x40000000); 1G
        NULL,
    };
    */

    debug("monitor len: %08lX\n", gd->mon_len);
    /*
     * Ram is setup, size stored in gd !!
     */
    debug("ramsize: %08lX\n", gd->ram_size);

    addr = CONFIG_SYS_SDRAM_BASE + gd->ram_size;        //0x0 + gd->ram_size

    /* reserve TLB table */
    addr -= (4096 * 4);

    /* round down to next 64 kB limit */
    addr &= ~(0x10000 - 1);

    gd->tlb_addr = addr;        //保留给tlb用
    debug("TLB table at: %08lx\n", addr);
```

```
/* round down to next 4 kB limit */
addr &= ~(4096 - 1);
debug("Top of RAM usable for U-Boot at: %08lx\n", addr);

/*
 * reserve memory for U-Boot code, data & bss
 * round down to next 4 kB limit
 */
addr -= gd->mon_len;        //保留给u-boot的代码，数据，bss段
addr &= ~(4096 - 1);

debug("Reserving %ldk for U-Boot at: %08lx\n", gd->mon_len >> 10, addr);

/*
 * reserve memory for malloc() arena
 */
addr_sp = addr - TOTAL_MALLOC_LEN;
debug("Reserving %dk for malloc() at: %08lx\n",
        TOTAL_MALLOC_LEN >> 10, addr_sp);
/*
 * (permanently) allocate a Board Info struct
 * and a permanent copy of the "global" data
 */
addr_sp -= sizeof (bd_t);
bd = (bd_t *) addr_sp;      //保留给bd结构
gd->bd = bd;
debug("Reserving %zu Bytes for Board Info at: %08lx\n",
        sizeof (bd_t), addr_sp);



addr_sp -= sizeof (gd_t);
id = (gd_t *) addr_sp;      //保留给gd结构
debug("Reserving %zu Bytes for Global Data at: %08lx\n",
        sizeof (gd_t), addr_sp);

/* setup stackpointer for exeptions */
gd->irq_sp = addr_sp;       //gd->irq_sp = id

/* leave 3 words for abort-stack    */
addr_sp -= 12;

/* 8-byte alignment for ABI compliance */
addr_sp &= ~0x07;



debug("New Stack Pointer is: %08lx\n", addr_sp);



gd->bd->bi_baudrate = gd->baudrate;     //bd波特率与gd同步
```

```
    /* Ram ist board specific, so move it to board code ... */
    dram_init_banksize();    //gd->bd->bi_dram[0].start = 0x0; gd->bd->bi_dram[0].size = gd-
>ram_size
    display_dram_config();  /* and display it */

    gd->relocaddr = addr;           //重地位地址
    gd->start_addr_sp = addr_sp;         //新的栈指针
    gd->reloc_off = addr - _TEXT_BASE;  //重定位地址到_TEXT_BASE的重定位偏移
    debug("relocation Offset is: %08lx\n", gd->reloc_off);
    memcpy(id, (void *)gd, sizeof(gd_t));   //将保存到r8寄存器的gd指针指向的数据写入sdram对应地址中
传送给linux

    relocate_code(addr_sp, id, addr);       //重地位u-boot代码

    /* NOTREACHED - relocate_code() does not return */
}
```

2.2.3 relocate_code(addr_sp, gd, addr)(arch\arm\cpu\armv7\start.S)

这部分代码的作用主要是将uboot重定位到ddr顶部，为内核腾出空间，同时也防止后来加载的内核将uboot覆盖。还有一点要注意的是这个函数最后不像平常函数那样返回，而是直接跳到board_init_r去执行。

```
ENTRY(relocate_code)
    mov r4, r0  /* save addr_sp */                  @ r4=addr_sp
    mov r5, r1  /* save addr of gd */               @ r5=gd  (gd地址)
    mov r6, r2  /* save addr of destination */      @ r6=addr(重定位地址)

    /* Set up the stack */
stack_setup:
    mov sp, r4

    adr r0, _start                                                  @将_start标号处地址给
r0(0x01000040)
    cmp r0, r6
    moveq   r9, #0      /* no relocation. relocation offset(r9) = 0 */  @如果r0==r6则r9=0
    beq clear_bss       /* skip relocation */                       @如果r0==r6则不需要重定
位，直接跳到后面
    mov r1, r6          /* r1 <- scratch for copy_loop */           @如果r0!=r6则将重定位目
标地址给r1
    ldr r3, _image_copy_end_ofs                                     @r3=将.rel.dyn之前的镜
像长度
    add r2, r0, r3      /* r2 <- source end address     */          @r2=r3+r0

copy_loop:
    ldmia   r0!, {r9-r10}       /* copy from source address [r0]    */  @r9=[r0],r10=
[r0+4],r0=r0+4+4
    stmia   r1!, {r9-r10}       /* copy to   target address [r1]    */  @[r1]=r9,
[r1+4]=r10,r1=r1+4+4
    cmp r0, r2          /* until source end address [r2]    */          @r0-r2
    blo copy_loop                                                   @如果r0-r2<0继续
copy_loop
                                                                    @@将_image_copy_start
和_image_copy_end之间的数据复制到gd->relocaddr地址处
```

```
    /*
     * fix .rel.dyn relocations
     */
    ldr r0, _TEXT_BASE      /* r0 <- Text base */                       @r0=0x01000040
    sub r9, r6, r0      /* r9 <- relocation offset */                   @r9=r6-r0=gd-
>reloc_off
    ldr r10, _dynsym_start_ofs  /* r10 <- sym table ofs */              @r10 =
__dynsym_start - _start
    add r10, r10, r0        /* r10 <- sym table in FLASH */             @r10 = .dynsym段开头的
加载地址
    ldr r2, _rel_dyn_start_ofs  /* r2 <- rel dyn start ofs */           @r2 =
__rel_dyn_start - _start
    add r2, r2, r0      /* r2 <- rel dyn start in FLASH */              @r2 = .rel.dyn段开头的
加载地址
    ldr r3, _rel_dyn_end_ofs    /* r3 <- rel dyn end ofs */             @r3 = __rel_dyn_end
- _start
    add r3, r3, r0      /* r3 <- rel dyn end in FLASH */                    @r3 = .rel.dyn段
末尾的加载地址
fixloop:
    ldr r0, [r2]        /* r0 <- location to fix up, IN FLASH! */
    add r0, r0, r9      /* r0 <- location to fix up in RAM */
    ldr r1, [r2, #4]
    and r7, r1, #0xff
    cmp r7, #23         /* relative fixup? */
    beq fixrel
    cmp r7, #2          /* absolute fixup? */
    beq fixabs
    /* ignore unknown type of fixup */
    b   fixnext
fixabs:
    /* absolute fix: set location to (offset) symbol value */
    mov r1, r1, LSR #4      /* r1 <- symbol index in .dynsym */
    add r1, r10, r1     /* r1 <- address of symbol in table */
    ldr r1, [r1, #4]        /* r1 <- symbol value */
    add r1, r1, r9      /* r1 <- relocated sym addr */
    b   fixnext
fixrel:
    /* relative fix: increase location by offset */
    ldr r1, [r0]
    add r1, r1, r9
fixnext:
    str r1, [r0]
    add r2, r2, #8      /* each rel.dyn entry is 8 bytes */
    cmp r2, r3
    blo fixloop
    b   clear_bss
_rel_dyn_start_ofs:
    .word __rel_dyn_start - _start
_rel_dyn_end_ofs:
    .word __rel_dyn_end - _start
_dynsym_start_ofs:
    .word __dynsym_start - _start
```

```
clear_bss:
    ldr r0, _bss_start_ofs                      @r0 = __bss_start - _start
    ldr r1, _bss_end_ofs                        @r1 = __bss_end - _start
    mov r4, r6          /* reloc addr */        @r4 = addr
    add r0, r0, r4                              @r0 = __bss_start - _start + addr(新的
__bss_start)
    add r1, r1, r4                              @r1 = __bss_end - _start + addr(新的
__bss_end)
    mov r2, #0x00000000     /* clear */         @r2 = #0x00000000

clbss_l:
    cmp r0, r1          /* clear loop... */                 @r0-r1
    bhs clbss_e         /* if reached end of bss, exit */   @r0大于或等于r1则跳转到clbss_e
    str r2, [r0]                                            @清零r0指向的一个4字节的内存
    add r0, r0, #4                                          @r0+=4
    b   clbss_l                                             @循环继续清0
clbss_e:

/*
 * We are done. Do not return, instead branch to second part of board
 * initialization, now running from RAM.
 */
jump_2_ram:
/*
 * If I-cache is enabled invalidate it
 */
#ifndef CONFIG_SYS_ICACHE_OFF
    mcr p15, 0, r0, c7, c5, 0   @ invalidate icache     @ 开icache
    mcr     p15, 0, r0, c7, c10, 4  @ DSB               @ DSB，数据同步屏障，内存操作完成才进行后面
指令的操作
    mcr     p15, 0, r0, c7, c5, 4   @ ISB               @ ISB，指令同步屏障，清空流水线
#endif
/*
 * Move vector table
 */
#if !defined(CONFIG_TEGRA20)
    /* Set vector address in CP15 VBAR register */
    ldr     r0, =_start                         @0x01000040
    add     r0, r0, r9                          @重定位后的_start地址
    mcr     p15, 0, r0, c12, c0, 0  @Set VBAR   @将异常向量地址设定为重定位后的_start地址处
#endif /* !Tegra20 */

    ldr r0, _board_init_r_ofs           @ r0 = board_init_r - _start
    adr r1, _start                      @ r1 = _start
    add lr, r0, r1                      @ lr = board_init_r
    add lr, lr, r9                      @ 重定位后的board_init_r地址lr = board_init_r + gd-
>reloc_off
    /* setup parameters for board_init_r */
    mov r0, r5      /* gd_t */          @ 设置board_init_r第一个参数id
    mov r1, r6      /* dest_addr */     @ 设置board_init_r第二个参数addr
    /* jump to it ... */
    mov pc, lr                          @ 跳转到重定位后的board_init_r地址去运行，拜拜，不回来了
```

```
_board_init_r_ofs:
    .word board_init_r - _start
ENDPROC(relocate_code)
```

2.2.4 board_init_r(id, addr)(arch\arm\lib\board.c)

```
void board_init_r(gd_t *id, ulong dest_addr)
{
    ulong malloc_start;

    gd = id;

    gd->flags |= GD_FLG_RELOC;  /* tell others: relocation done */
    bootstage_mark_name(BOOTSTAGE_ID_START_UBOOT_R, "board_init_r");

    monitor_flash_len = _end_ofs;       //__image_copy_end - _start

    /* Enable caches */
    enable_caches();                    //使能D-cache. I-cache已经在start.S使能了，并开mmu

    debug("monitor flash len: %08lX\n", monitor_flash_len);
    board_init();   /* Setup chipselects */ //gd->bd->bi_boot_params = 0x00000100; tag或者设备
树地址
    /*
     * TODO: printing of the clock inforamtion of the board is now
     * implemented as part of bdinfo command. Currently only support for
     * davinci SOC's is added. Remove this check once all the board
     * implement this.
     */

    debug("Now running in RAM - U-Boot at: %08lx\n", dest_addr);


    /* The Malloc area is immediately below the monitor copy in DRAM */
    malloc_start = dest_addr - TOTAL_MALLOC_LEN;            //malloc堆的起始地址
    mem_malloc_init (malloc_start, TOTAL_MALLOC_LEN);   //初始化malloc堆

    puts("MMC:   ");
    mmc_initialize(gd->bd);                                 //初始化&mmc_devices链表，初始化一个mmc结
构体，注册mmc设备（将mmc添加到&mmc_devices链表中）

    /* initialize environment */
    env_relocate();     //通过sd卡初始化环境变量,如果没有或者其他问题则使用默认环境变量，存在env_htab中
(env_mmc.c)
    /* get the devices list going. */
    stdio_init();       //初始化devs.list链表，注册一个stdio_dev设备到该链表中，设备名
为“serial”(stdio.c)

    jumptable_init();   //分配内存给跳转表并赋值，模拟系统调用

    console_init_r();   //gd->flags |= GD_FLG_DEVINIT;设置stdio_devices[]三个设备
stdio,stdout,stderr都为stdio_init()中注册的名为“serial”的设备
```

```
    /* miscellaneous platform dependent initialisations */
    misc_init_r();

     /* set up exceptions */
    interrupt_init();           //没有使用
    /* enable exceptions */
    enable_interrupts();        //没有使用

    /* Initialize from environment */
    load_addr = getenv_ulong("loadaddr", 16, load_addr);        //获取环境变量loadaddr

    board_late_init();

    puts("Net:   ");
    eth_initialize(gd->bd);

    /* main_loop() can return to retry autoboot, if so just run it again. */
    for (;;) {
        main_loop();            //进入最后加载启动linux阶段
    }

    /* NOTREACHED - no way out of command loop except booting */
}
```

2.2.5 main_loop()(common\main.c)

该阶段是uboot生命最后阶段，主要工作是加载启动linux内核，还可以进入命令行模式进行人工调试和启动。下面提取出启动内核的主分支代码，其他零碎的去掉。

```
    ...

    s = getenv ("bootdelay");
    bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;

    ...

    s = getenv ("bootcmd");

    ...

    // 如果在bootdelay延时期间有按键则跳到下面的for循环进入uboot命令行人工调试和启动模式
    if (bootdelay != -1 && s && !abortboot(bootdelay)) {
        run_command_list(s, -1, 0);     //运行环境变量bootcmd中的命令自动启动
    }

    ...

    for (;;) {

        len = readline (CONFIG_SYS_PROMPT);     //读取命令行输入的命令

        flag = 0;   /* assume no special flags for now */
```

```
        if (len > 0)
            strcpy (lastcommand, console_buffer);
        else if (len == 0)
            flag |= CMD_FLAG_REPEAT;

            return;     /* retry autoboot */

        if (len == -1)
            puts ("<INTERRUPT>\n");
        else
            rc = run_command(lastcommand, flag);        //运行命令行输入的命令

        if (rc <= 0) {
            /* invalid command or not repeatable, forget it */
            lastcommand[0] = 0;
        }
    }
```

2.2.6 由环境变量bootcmd分析最后的加载启动linux内核

```
SOCFPGA_CYCLONE5 # printenv
ECC_SDRAM=0
ECC_SDRAM_DBE=0
ECC_SDRAM_SBE=0
baudrate=57600
bootargs=console=ttyS0,57600
bootcmd=run mmcload; run mmcboot
bootdelay=4
bootimage=uImage
bootimagesize=0x600000
ethact=mii0
fdtaddr=0x00000100
fdtimage=socfpga.dtb
fdtimagesize=0x2000
fpga=0
fpgadata=0x2000000
fpgadatasize=0x700000
loadaddr=0x7fc0
mmcboot=setenv bootargs console=ttyS0,57600 root=${mmcroot} rw rootwait;bootm ${loadaddr} -
${fdtaddr}
mmcload=mmc rescan;${mmcloadcmd} mmc 0:${mmcloadpart} ${loadaddr} ${bootimage};${mmcloadcmd}
mmc 0:${mmcloadpart} ${fdtaddr} ${fdtimage}
mmcloadcmd=fatload
mmcloadpart=1
mmcroot=/dev/mmcblk0p2
netboot=dhcp ${bootimage} ; tftp ${fdtaddr} ${fdtimage} ; run ramboot
qspiboot=setenv bootargs console=ttyS0,57600 root=${qspiroot} rw
rootfstype=${qspirootfstype};bootm ${loadaddr} - ${fdtaddr}
qspibootimageaddr=0xa0000
qspifdtaddr=0x50000
qspiload=sf probe ${qspiloadcs};sf read ${loadaddr} ${qspibootimageaddr} ${bootimagesize};sf
read ${fdtaddr} ${qspifdtaddr} ${fdtimagesize};
qspiloadcs=0
```

```
qspiroot=/dev/mtdblock1
qspirootfstype=jffs2
ramboot=setenv bootargs console=ttyS0,57600;bootm ${loadaddr} - ${fdtaddr}
stderr=serial
stdin=serial
stdout=serial
verify=n
```

从以上输出的环境变量可以看到bootcmd=run mmcload; run mmcboot，然后再分别展开mmcload和mmcboot

```
mmcload = mmc rescan;${mmcloadcmd} mmc 0:${mmcloadpart} ${loadaddr}
${bootimage};${mmcloadcmd} mmc 0:${mmcloadpart} ${fdtaddr} ${fdtimage}
        = mmc rescan;fatload mmc 0:1 0x7fc0 uImage ; fatload mmc 0:1 0x00000100 socfpga.dtb
```

```
mmcboot = setenv bootargs console=ttyS0,57600 root=${mmcroot} rw rootwait;bootm ${loadaddr}
- ${fdtaddr}
        = setenv bootargs console=ttyS0,57600 root=/dev/mmcblk0p2 rw rootwait;bootm 0x7fc0 -
0x00000100
```

因此可以总结出bootcmd的启动流程：将内核镜像uImage加载到内存地址0x7fc0处，将设备树文件socfpga.dtb加载到内存地址0x00000100处；设置环境变量bootargs为console=ttyS0,57600 root=/dev/mmcblk0p2 rw rootwait，bootargs是uboot启动内核时要传给内核的启动参数；然后通过bootm 0x7fc0 - 0x00000100命令启动内核，其中0x7fc0是内核加载到内存的地址，0x00000100是设备树加载到内存的地址。由此可见bootm是uboot最后一条命令，我们重点关注bootm做了什么。其中bootm命令会调用do_bootm函数，do_bootm函数首先通过bootm_start利用uImage镜像头部信息和命令行bootm命令的参数填充images结构体，然后通过boot_fn = boot_os[images.os.os]获得do_bootm_linux最后调用它。

```
int do_bootm_linux(int flag, int argc, char *argv[], bootm_headers_t *images)
{
    /* No need for those on ARM */
    if (flag & BOOTM_STATE_OS_BD_T || flag & BOOTM_STATE_OS_CMDLINE)
        return -1;

    if (flag & BOOTM_STATE_OS_PREP) {
        boot_prep_linux(images);
        return 0;
    }

    if (flag & BOOTM_STATE_OS_GO) {
        boot_jump_linux(images);
        return 0;
    }

    boot_prep_linux(images);
    boot_jump_linux(images);
    return 0;
}
```

首先在boot_prep_linux -> create_fdt -> fdt_chosen 中设置设备数中的bootargs属性。这段代码先通过getenv获取上文mmcboot命令中设置的bootargs环境变量，然后将它设置在设备数的bootargs属性中。

```
int fdt_chosen(void *fdt, int force)
{
    ...

    str = getenv("bootargs");
    if (str != NULL) {
        path = fdt_getprop(fdt, nodeoffset, "bootargs", NULL);
        if ((path == NULL) || force) {
            err = fdt_setprop(fdt, nodeoffset,
                "bootargs", str, strlen(str)+1);
            if (err < 0)
                printf("WARNING: could not set bootargs %s.\n",
                    fdt_strerror(err));
        }
    }

    ...

}
```

最后是boot_jump_linux(images)的最后部分。因为socfpga_cyclone5使用设备树启动，因此CONFIG_OF_LIBFDT定义，所以r2是设备树地址，最后通过kernel_entry(0, machid, r2)跳转到内核入口地址运行。

```
/* Subcommand: GO */
static void boot_jump_linux(bootm_headers_t *images)
{

...

#ifdef CONFIG_OF_LIBFDT
    if (images->ft_len)
        r2 = (unsigned long)images->ft_addr;
    else
#endif
        r2 = gd->bd->bi_boot_params;

    kernel_entry(0, machid, r2);
}
```