uboot armv7m启动流程_uboot启动流程详细分析（二）_weixin_39634022的博客-CSDN博客

从start_armboot开始，在startup.c中所包含#include

在config.h中，

```
#define CONFIG_BOARDDIR board/hisilicon/hi3559av100#include #include #include #include #include
```

在hi3559av100.h中，

```
#define CONFIG_SYS_TEXT_BASE 0x40800000
```

在hi3559av100.h中，看到了CONFIG_SYS_TEXT_BASE的宏

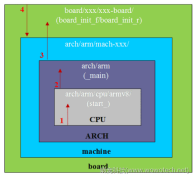
```
#define CONFIG_SYS_TEXT_BASE=0x40800000
```

查看u-boot.map在这里又来到了上一层的start.S中来，所以可以知道这两个是由两个文件组成的，一个是u-boot.bin和img_info.bin，就是说两个不同的start.S的流程来合成一个最终的u-boot-hi3559av100.bin

这个便是正常的流程了。

本文将结合u-boot的"board—>machine—>arch—>cpu"框架，介绍u-boot中平台相关部分的启动流程，并通过对启动流程的简单分析，掌握u-boot移植的基本方法。

这些问题的本质，是软件工程中的抽象和封装，以最简洁、最高效的方式，实现尽可能多的功能。u-boot作为一个跨平台、跨设备的bootloader，同样会面临这些问题，它的解决方案，就是"board—>machine—>arch—>cpu"框架，如下：



基于图片1的架构，u-boot和平台有关的初始化流程，显得比较直观、清晰。

1)u-boot启动后，会先执行CPU(位于arm)的初始化代码。

2)CPU相关的代码，会调用ARCH的公共代码(位于arch/arm)。

3)ARCH的公共代码，在适当的时候，调用board有关的接口。u-boot的功能逻辑，大多是由common代码实现，部分和平台有关的部分，则由公共代码声明，由board代码实现。

4)board代码在需要的时候，会调用machine(arch/arm/mach-xxx)提供的接口，实现特定的功能。因此machine的定位是提供一些基础的代码支持，不会直接参与到u-boot的功能逻辑中。

本文先不涉及u-boot和平台相关的Kconfig/Makefile部分，以ARM64为例，假定u-boot开先从"arch/arm/cpu/armv8/start.S"的_start接口开始执行，因此我们从_start开始分析。

3.1 _start

_start是u-boot启动后的第一个执行地址，对armv8来说，它只是简单的跳转到reset处执行，如下：

```
.globl _start_start:b\reset
```

3.2 reset

```
reset:bsave_boot_params.globlesave_boot_params_retsave_boot_params_ret:#ifdef CONFIG_SYS_RESET_SCTRLbl reset_sctr\#endifadrv0, vectorsswitch_el x1, 3f, 2f, 1f3:msr\vbar_el3, x0msrx0, scr_el1lsrx0, x0, #0xfmsrscr_el3, x0msrcptr_el3, xzrmt\#def COUNTER_FREQUENCY\ldrw0, =COUNTER_FREQUENCY\msrcntfrq_el0, x0\#endif\#f2:msr\vbar_el2, x0msrx0, #0x33ffmsrcptr_el2, x0\#f1:msr\vbar_el1, x0msrx0, #3 << 20msrcpacr_el1, x0b:blapply_core_erratablloaelevel_init
```

1)reset SCTRL寄存器

具体可参考reset_sctrl函数，由CONFIG_SYS_RESET_SCTRL控制，一般不需要打开，该配置项的解释如下：

> Reset the SCTRL register at the very beginning of execution to avoid interference from stale mappings set up by early firmware/loaders/etc.
>
> http://lists.denx.de/pipermail/u-boot/2015-April/211147.html

2)根据当前的EL级别，配置中断向量、MMU、Endian、i/d Cache等。

3)配置ARM的勘误表

具体可参考apply_core_errata函数，由CONFIG_ARM_ERRATA_XXX控制，在项目的初期，可以不打开，后续根据实际情况打开。

> 就是ARM有一些bug，但可以通过软件的方法绕过去，由u-boot的代码注释可知，应该只有Cortex-A57才有，具体什么bug，我也没有去研究

4)调用lowlevel_init的功能解释如下(具体可参考u-boot的readme文档)。

- purpose: essential init to permit execution to reach board_init_f()

```
- no global_data or BSS - there is no stack (ARMv7 may have one but it will soon be removed) - must not set up SDRAM or use console - must only do the bare minimum to allow execution to continue to board_init_f() - this is almost never needed - return normally from this function
```

> 海思的相应生about代码的start.S其实就是增加以下内容

```
-------
        bl      gic_wait_for_interrupt
#endif

        /*
         * All slaves will enter EL2 and optionally EL1.
         */
        adr     x3, lowlevel_in_el2
        ldr     x4, =ES_TO_AARCH64
        bl      armv8_switch_to_el2

lowlevel_in_el2:
#ifdef CONFIG_ARMV8_SWITCH_TO_EL1
        adr     x3, lowlevel_in_el1
        ldr     x4, =ES_TO_AARCH64
        bl      armv8_switch_to_el1

lowlevel_in_el1:
#endif

#endif /* CONFIG_ARMV8_MULTIENTRY */
```

```
-------
        bl      gic_wait_for_interrupt
#endif

        /*
         * All slaves will enter EL2 and optionally EL1.
         */

        bl      armv8_switch_to_el2

#ifdef CONFIG_ARMV8_SWITCH_TO_EL1

        bl      armv8_switch_to_el1

#endif

#endif /* CONFIG_ARMV8_MULTIENTRY */
```

5)如果是多CPU的场景，处理其它的CPU的boot

多CPU功能由CONFIG_ARMV8_MULTIENTRY控制，不需要打开。

6)跳转到arm公共的_main中执行

ARM64平台的_main位于crt0_64.S文件中，具体请参考下面的描述。

3.3 _main

crt0是C-runtime Startup Code的简称，意思就是进行C代码之前的准备工作，关于_main函数，crt0_64.S中有非常详细的注释(这一点要感谢u-boot点100个赞！)，大家可以参考，该函数的定义如下：

```
ENTRY(_main)#if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)ldrx0, =(CONFIG_SPL_STACK)#elseldrx0, =(CONFIG_SYS_INIT_SP_ADDR)#endifbicsp, x0, #0xfmovx0, sp\bl board_init_f_alloc_reservemovsp, x0movx18, x0\bl board_init_f_init_reservemovx0, #0\bl board_init_f#if !defined(CONFIG_SPL_BUILD)ldrx0, [x18, #GD_START_ADDR_SP]bicsp, x0, #0xfldrx30, [x18, #GD_BD]subx30, x30, #GD_SIZEadr\r, relocation_return\ldrx9, [x18, #GD_RELOC_OFF]add\r, lr, x9ldrx0, [x18, #GD_RELOCADD#
```

功能可总结为大部分翻译自crt0_64.S中的注释)。

1)设置C代码的运行环境，为调用board_init_f接口做准备，包括：

a)设置堆栈(C代码的函数调用，堆栈是必需的)，如果当前的编译是SPL(由CONFIG_SPL_BUILD定义)，可参数定义堆栈指定(CONFIG_SPL_STACK)，否则，通过CONFIG_SYS_INIT_SP_ADDR定义堆栈指定；b)调用board_init_f_alloc_reserve接口，从堆栈开始的地方，为u-boot中大共愚蠢的GD（'global data'）数据结构，分配空间；c)调用board_init_f_init_reserve接口，对GD进行初始化。

2)调用board_init_f函数，完成一些前期的初始化工作，例如：

a)点亮一个Debug用的LED灯，表示u-boot已经活了；b)初始化DRAM，DDR(System总线的RAM)等；c)计算后续代码需要使用的一些参数，包括relocation destination、the future stack、the future GD location等。

注5：关于u-boot的relocation操作，后续会有专门的文章介绍。

3)如果当前是SPL(由CONFIG_SPL_BUILD控制)，则_main函数结束，直接返回；如果是正常的u-boot，则继续执行后续的动作。

4)根据board_init_f指定的参数，执行u-boot的relocation操作。

5)清除BBS段。

6)调用board_init_r函数，执行后续的初始化操作(已经不再本文的讨论范围了，具体请参考后续的分析文章)。

4.1 SPL功能

SPL是Secondary Program Loader的简称，之所以称作secondary，是相对于ROM code来说的，SPL是u-boot中独立的一个代码分支，由CONFIG_SPL_BUILD配置项控制，是为了在正常的u-boot image之外，提供一个独立的、小size的SPL image，通常用于那些SRAM比较小(或者其它限制)，无法直接装载并运行整个u-boot的平台。

如果使用了SPL功能，u-boot的启动流程通常是：

ROM code→BSP→SPL→正式的u-boot

ROM code和BSP先运行，SPL进行必要的初始化之后，加载u-boot并运行，u-boot进行后续的操作。

因此，如果使用SPL功能，需要尽可能的减小SPL的代码量，以减小它的size。

4.2 配置项总结

经过第1章的流程分析，我们可以总结出armv8平台相关部分的启动流程"有关的配置项，记录如下：

CONFIG_SYS_RESET_SCTRL：控制是否在启动的时候reset SCTRL寄存器，一般不需要打开；CONFIG_ARM_ERRATA_XXX：控制ARM core的勘误信息，一般不需要打开；CONFIG_GICV2：CONFIG_GICV3：控制GIC的版本，同时的时钟声说明；CONFIG_ARMV8_MULTIENTRY：控制是否在u-boot中使用多CPU，一般不需要；CONFIG_SPL_BUILD：是否是在SPL的编译，需要的话可以打开；CONFIG_SPL_STACK：如果配置了CONFIG_SPL_BUILD，是否为SPL image配置单独的stack(SP指定)，如果需要，通过调配置项配置，如果不需要，则使用CONFIG_SYS_INIT_SP_ADDR。