第一章 ESP32 的启动过程

1. 目标

- 了解 app_main 之前 ESP32 干了什么
- 了解 ESP32 复位原因

2. ESP32 启动总流程（原文）

ESP32 开发程序中有且只能有一个 app_main 函数，该函数是用户程序的入口，相当于其它系统中的 main 函数。但在 app_main 之前，系统还有一段初始化的过程，其大致可以分为以下三个过程：

1. ROM 中的第一级引导加载程序将闪存偏移 0x1000 的第二级引导加载程序映像加载到 RAM（IRAM 和 DRAM）；
2. 第二级引导程序从闪存加载分区表和主应用程序映像，主应用程序包含 RAM 段和通过闪存缓存映射的只读段；
3. 主应用程序执行，此时可以启动第二个 CPU 和 RTOS 调度程序。

3. ESP32 启动详细过程（可以不看了）

3.1. 第一阶段

系统 first-stage bootload 启动，对于系统的 first-stage bootloader，其主要任务是负责从 Flash 的地址 0X1000 开始加载 bootloader 镜像到 RAM 中。此工程源码在 esp-idf 的 component 目录下 bootloader/subproject/main/bootloader_start.c。

在 SoC 复位后，PRO CPU 将立即开始运行，执行复位向量代码，而 APP CPU 将保持复位。在启动过程中，PRO CPU 执行所有初始化。call_start_cpu0 应用程序启动代码功能中的 APP CPU 复位被取消置位。复位向量代码位于 ESP32 芯片掩码 ROM 中的地址 0x40000400，不能修改。

从复位向量调用的启动代码通过检查 GPIO_STRAP_REG(gpio_reg.h 定义的)引导引脚[GPIO0，GPIO2，GPIO4，MTDO，GPIO5]状态的寄存器来确定引导模式。

- 如果 GPIO0 和 GPIO2 同时为低电平，则会进入下载模式，等待串口通信信息。
- 如果 GPIO0 为高电平，则会进入 Flash 运行模式，启动 SPI 驱动，并加载 Flash 中的程序段。

本次启动 ESP32 可知道上次复位的原因，有以下几种可能。

从深度睡眠复位

如果 RTC_CNTL_STORE6_REG 值为非零，并且 RTC 存储器的 CRC 值 RTC_CNTL_STORE7_REG 有效，RTC_CNTL_STORE6_REG 则将其用作入口点地址并立即跳转。如果 RTC_CNTL_STORE6_REG 为零，或 RTC_CNTL_STORE7_REG 包含无效的 CRC，或者一旦调用通过 RTC_CNTL_STORE6_REG 返回的代码，继续进行启动，就好像是上电复位一样。注意，此时运行自定义代码，提供了一个深度睡眠存根机制。

对于上电复位，软件 SOC 复位和看门狗 SOC 复位

GPIO_STRAP_REG 如果要求 UART 或 SDIO 下载模式，请检查寄存器。如果是这种情况，请配置 UART 或 SDIO，并等待下载代码。否则，继续进行启动，就好像是由于软件 CPU 复位。

对于软件 CPU 复位和看门狗 CPU 复位

根据 EFUSE 值配置 SPI 闪存，并尝试从闪存加载代码。如果从闪存加载代码失败，将 BASIC 解释器解压缩到 RAM 中并启动它。当发生这种情况时，RTC 看门狗仍然使能，因此除非解释器接收到任何输入，否则看门狗将在几百毫秒内重置 SOC，重复整个过程。如果解释器从 UART 接收到任何输入，它将禁用看门狗。

ESP32 复位源

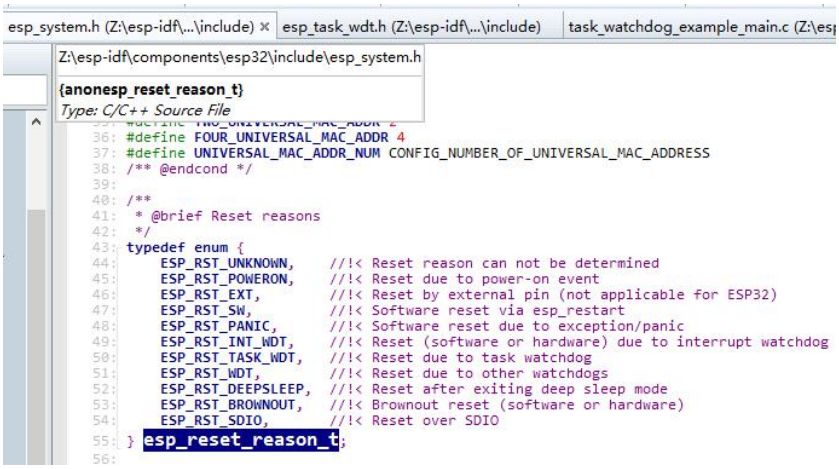
3.1.2 复位源

大多数情况下，APP_CPU 和 PRO_CPU 将被立刻复位，有些复位源只能复位其中一个。APP_CPU 和 PRO_CPU 的复位原因也各自不同：当系统复位起来之后，PRO_CPU 可以通过读取寄存器 RTC_CNTL_RESET_CAUSE_PROCPU 来获取复位源，APP_CPU 则可以通过读取寄存器 APP_CNTL_RESET_CAUSE_PROCPU 来获取复位源。

表 10 列出了从这些寄存器中可能读出的复位源。

表 10: PRO_CPU 和 APP_CPU 复位源

PRO	APP	源	复位方式	注释
0x01	0x01	芯片上电复位	系统复位	-
0x10	0x10	RWDT 系统复位	系统复位	详见 WDT 章节
0x0F	0x0F	欠压复位	系统复位	详见 Power Management 章节
0x03	0x03	软件系统复位	内核复位	配置 RTC_CNTL_SW_SYS_RST 寄存器
0x05	0x05	Deep Sleep Reset	内核复位	详见 Power Management 章节
0x07	0x07	MWDT0 全局复位	内核复位	详见 WDT 章节



第一阶段总结：可以看出，第一阶段主要是为了第二阶段做铺垫，应用程序二进制从地址 0x1000 开始从闪存加载。第一个 4kB 闪存扇区用于存储安全引导 IV 和应用程序映像的签名。

3.2. 第二阶段

在 ESP-IDF 中，闪存中位于 0x1000 位置的二进制映像是第二级引导加载程序。ESP-IDF 的组件 bootloader 目录中提供了第二阶段引导加载程序源代码。这种安排并不是 ESP32 芯片中唯一的可能，也可以编写一个功能齐全的应用程序，当闪存到 0x1000 时，该应用程序将工作，ESP-IDF 中使用第二阶段引导加载程序来增加闪存布局的灵活性（使用分区表），并允许发生与闪存加密，安全引导和空中更新（OTA）相关的各种流程。

当第一阶段引导加载程序完成检查和加载第二阶段引导加载程序时，它跳转到二进制映像头中找到的第二阶段引导加载程序入口点。

第二阶段引导程序读取在偏移 0x8000 处找到的分区表。引导加载程序找到工厂和 OTA 分区，并根据在 OTA 信息分区中找到的数据来决定哪一个进行引导。

对于所选分区，第二级引导加载程序将映射到 IRAM 和 DRAM 的数据和代码段复制到其加载地址。对于在 DROM 和 IROM 区域中具有加载地址的部分，Flash MMU 配置为提供正确的映射。第二阶段引导加载程序为 PRO 和 APP CPU 配置闪存 MMU，但只能为 PRO CPU 启用闪存

MMU。这样做的原因是第二阶段引导程序代码被加载到 APP CPU 缓存使用的内存区域中。启用 APP CPU 的缓存的功能被传递给应用程序。一旦加载了代码并且设置了闪存 MMU，则第二级引导加载程序将跳转到二进制映像头中的应用程序入口点。

目前，官方并不支持加载程序添加应用程序定义来自己定义应用程序分区选择逻辑。

3.3. 第三阶段

主函数镜像开始执行（即 main_task，应用程序入口点是 call_start_cpu0，可在 components/esp32/cpu_start.c 中找到），这个功能的两个主要作用是启用堆分配器并使 APP CPU 跳到其入口点 call_start_cpu1。PRO CPU 上的代码设置 APP CPU 的入口点，取消置位 APP CPU 复位，并等待由 APP CPU 上运行的代码设置的全局标志，表示已启动。一旦完成，PRO CPU 跳转到 start_cpu0 功能，并且 APP CPU 将跳转到 start_cpu1 功能。

start_cpu0 和 start_cpu 的功能并不是不可修改的，start_cpu0 根据所做的选择启用或初始化组件默认实现，可以通过查看 components/esp32/cpu_start.c 观察最新的执行步骤列表，不过值得注意的是，此阶段将调用应用程序中存在的所有 C++全局构造函数。一旦所有基本组件都被初始化，则创建主任务，并启动 FreeRTOS 调度程序。esp32 是一个双核 cpu，在这个过程中，当 PRO CPU 在 start_cpu0 功能中进行初始化时，APP CPU 会自动 start_cpu1 运行功能，等待在 PRO CPU 上启动调度程序。一旦在 PRO CPU 上启动了调度程序，APP CPU 上的代码也启动了调度程序。

main_task 的任务是可以配置主任务堆栈大小和优先级，当然我们可以使用此任务进行初始的应用程序特定设置，例如启动其它任务。应用程序还可以使用事件循环和其它通用活动的主要任务。但是需要注意的是，如果 app_main 函数返回，main_task 将被删除。

4. ESP32 启动总结

- 再不需要修改 boot 结构的情况下就不用看了，了解有好几个阶段即可。
- 如果实在想搞清楚启动流程，就把源码好好阅读一遍。反正我没看。
- 源码：esp-idf/component/bootloader/subproject/main/bootloader_start.c