u-boot-2012

图示

描述已自动生成

根据上图，系统启动的大概顺序：

iROM在SOC内部，是一个64KB的ROM，初始化一些系统启动必须的功能。比如：时钟、栈。

iROM负责从特殊的启动外设加载BL1的image到soc内部的256KB的SRAM中。启动的外设由操作按钮来决定的。根据不同按键的值，iROM将会对bl1 的image做不同的校验。

BL1初始化系统时钟和DRAM控制器，然后从启动外设加载OS image到DRAM中。根据启动按钮的值的不同，BL1会对OS做不同的校验。

启动完成之后，BL1跳转到操作系·统（kernel）。

iROM会根据OM 引脚的不同选择不同的启动设备，对应的OM寄存器需要提供对应的启动信息。

AM335x通常的启动分为三个阶段，ROM Code、SPL、u-boot，最终加载内核。

ROM Code:

顾名思义，这块的代码是TI内置的。上电后，硬件去加载该代码。这块的代码作用，是根据启动选项来从不同的启动源启动。目前支持的启动源有MMC、Nand、Uart、USB、net等方式。如，我们的项目用的Nand，ROM Code运行后，会初始化Nand，然后在Nand的0x0的位置加载下一阶段的代码，即SPL的代码。把SPL的代码，加载到内部存储器中运行。

SPL:

SPL和MLO有什么区别，可以认为是没有区别。具体的区别是SPL代码中包含了调试等描述信息，MLO作为裁剪后的代码，烧写进Nand中。最大的疑问是，为什么需要SPL阶段，直接ROM Code启动u-boot不是更快吗？因为硬件，目前，ROM Code只能把代码加载到内部存储器中运行，而内部存储器资源非常有限，运行不了正常的u-boot，所以先运行精简版的u-boot，再去加载正常的u-boot。  
所以，SPL阶段的使命很明确，初始化外存，加载u-boot到外存。

u-boot:

不必多言了，这就是正常的u-boot，一是提供交互界面，供用户进行配置；二是初始化硬件环境加载内核运行。当然，核心功能上面也说了，是加载内核。

另外，目前AM335x也支持Falcon模式，在该模式下，SPL直接加载内核启动，后面会有详细的介绍。

# **1:am335x的cpu上电后，会跳到哪个地址去执行？**

芯片到uboot启动流程 ：ROM→ MLO(SPL)→uboot.img

AM335x 中bootloader被分成了 3 个部分：

**第一级 bootloader（ROM）：**引导加载程序，板子上电后会自动执行这些代码，如选择哪种方式启动（NAND，SDcard，UART。。。），然后跳转转到第二级 bootloader（mlo/spl）。这些代码应该是存放在 176KB 的 ROM 中。

**第二级 bootloader（MLO/SPL**）**：**用以硬件初始化：关闭看门狗，关闭中断，设置 CPU 时钟频率、速度等操作。然后会跳转到第三级bootloader（u-boot）。MLO文件应该会被映射到 Internal SRAM 中。

**第三级 bootloader（u-boot）：**C代码的入口。

其中第一级 bootloader 是板子固化的，第二级和第三级是通过编译 uboot 所得的。

表格

描述已自动生成

表格

描述已自动生成

表格

描述已自动生成

## **第二级 bootloader：MLO（SPL）做了哪些事情？**

**A:MLO(SPL)内存分布如下（arch\arm\cpu\armv7\sunxi\u-boot-spl.lds）：**

图片包含 文本

描述已自动生成

### 0x402f0400

### 第一步：**arch/arm/cpu/armv7/start.S**

(arch/arm/cpu/armv7/start.S).globl\_start

----→reset:

----→(arch/arm/cpu/armv7/ti81xx/lowlevel\_init.S)save\_boot\_params:

---→(arch/arm/cpu/armv7/start.S)cpu\_init\_crit(cpu的初始化):

---→(arch/arm/cpu/armv7/ti81xx/lowlevel\_init.S)lowlevel\_init

----→s\_init\_start

----->(board/forlinx/ok335x/evm.c)s\_init()

--→(arch/arm/cpu/armv7/omap-common/spl.c)board\_init\_f

---→(arch/arm/cpu/armv7/omap-common/spl.c)relocate\_code()

---->(arch/arm/cpu/armv7/omap-common/spl.c)board\_init\_r()

-→cpu\_init\_crit---->(arch/arm/cpu/armv7/lowlevel\_init.S)-------->lowlevel\_init---->s\_init

/\*

\* 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.

\*/

--→(arch/arm/lib/crt0.S)\_main

->(arch/arm/cpu/armv7/omap-common/spl.c)board\_init\_f(①、初始化一系列外设，比如串口、定时器，或者打印一些消息等。

②、初始化 gd 的各个成员变量， uboot 会将自己重定位到 DRAM 最后面的地址区域，也就

是将自己拷贝到 DRAM 最后面的内存区域中。这么做的目的是给 Linux 腾出空间，防止 Linux

kernel 覆盖掉 uboot，将 DRAM 前面的区域完整的空出来。在拷贝之前肯定要给 uboot 各部分

分配好内存位置和大小，比如 gd 应该存放到哪个位置， malloc 内存池应该存放到哪个位置等

等。这些信息都保存在 gd 的成员变量中，因此要对 gd 的这些成员变量做初始化。最终形成一

个完整的内存“分配图”，在后面重定位 uboot 的时候就会用到这个内存“分配图”。)

->(**arch/arm/lib/crt0.S**)relocate\_code

->(**arch/arm/lib/relocate.S**)relocate\_vectors

->(**common/board\_r.c**)board\_init\_r(board\_init\_f 函数，在此函数里面会调用一系列的函数来初始化一些外设和 gd 的成员变量。但是 board\_init\_f 并没有初始化所有的外设，还需要做一些后续工作。

)

---> initr\_nand（）:就是初始化nand终端输出nand：256mb

----> initr\_mmc（）:初始化mmc终端输出mmc：0

---->(board/ti/am335x/board.c)board\_late\_init（）

--→(common/board\_r.c)run\_main\_loop():如果倒计时结束以后都没有按下回车键，那么就会自 动启动 Linux 内核 ， 这 个 功 能 就 是 由 run\_main\_loop 函 数 来 完 成 的

--→(common/main.c)main\_loop():

--->cli\_init():跟命令初始化有关，初始化 hushshell 相关的变量

--->run\_preboot\_environment\_command():获取环境变量 perboot 的内容， perboot是一些预启动命令，一般不使用这个环境变量

----→bootdelay\_process():此函数会读取环境变量 bootdelay 和 bootcmd 的内容，

然后将 bootdelay 的值赋值给全局变量 stored\_bootdelay，返回值为环境变量 bootcmd 的值

---→(common/autoboot.c)autoboot\_command():此函数就是检查倒计时是否结束？ 倒计时结束之前有没有被打断？

---->run\_command\_list

---→cli\_loop():

[如果倒计时自然结束那么就执行函数run\_command\_list，此函数会执行参数 s 指定的一系列命令， 也就是环境变量 bootcmd 的命令，bootcmd 里面保存着默认的启动命令，因此 linux 内核启动！这 个就是 uboot 中倒计时结束以后自动启动 linux 内核的原理。如果倒计时结束之前按下了键盘上的按 键，那么 run\_command\_list函数就不会执行，相当于 autoboot\_command 是个空函数。

回到“遥远”的 main\_loop 函数中，如果倒计时结束之前按下按键，那么就会执行第 74 行的 cli\_loop 函数，这个就是命令处理函数，负责接收好处理输入的命令]

\_main:

/\*

\* This file handles the target-independent stages of the U-Boot

\* start-up where a C runtime environment is needed. Its entry point

\* is \_main and is branched into from the target's start.S file.

\* 此文件处理 U-Boot 的目标独立阶段在需要 C 运行时环境的地方启动。它的入口点

\* 是\_main的，从目标的开头分支到。S 文件。

\*

\* \_main execution sequence is:

\*

\* 1. Set up initial environment for calling board\_init\_f().

\* This environment only provides a stack and a place to store

\* the GD ('global data') structure, both located in some readily

\* available RAM (SRAM, locked cache...). In this context, VARIABLE

\* global data, initialized or not (BSS), are UNAVAILABLE; only

\* CONSTANT initialized data are available. GD should be zeroed

\* before board\_init\_f() is called.

\*1.设置用于调用 board\_init\_f（） 的初始环境。此环境仅提供堆栈和存储位置GD（“全局数据”）结构，

\* 两者都位于一些容易可用的内存（SRAM，锁定的缓存...在这种情况下，变量全局数据，无论是否初始化

\*（BSS）都不可用;只常量初始化数据可用。GD 应归零

\* 在调用 board\_init\_f（） 之前。

\* 2. Call board\_init\_f(). This function prepares the hardware for

\* execution from system RAM (DRAM, DDR...) As system RAM may not

\* be available yet, , board\_init\_f() must use the current GD to

\* store any data which must be passed on to later stages. These

\* data include the relocation destination, the future stack, and

\* the future GD location.

\* 2.调用 board\_init\_f（）。此函数为从系统RAM执行（DRAM，DDR...）由于系统内存可能不会

\* 可用，board\_init\_f（） 必须使用当前的 GD 来存储必须传递到后期阶段的任何数据。这些

\* 数据包括搬迁目的地、未来堆栈和未来的GD位置。

\* 3. Set up intermediate environment where the stack and GD are the

\* ones allocated by board\_init\_f() in system RAM, but BSS and

\* initialized non-const data are still not available.

\* 3. 设置堆栈和 GD 所在的中间环境由系统 RAM 中的 board\_init\_f（） 分配，但 BSS 和

\* 初始化的非常量数据仍然不可用。

\* 4a.For U-Boot proper (not SPL), call relocate\_code(). This function

\* relocates U-Boot from its current location into the relocation

\* destination computed by board\_init\_f().

\* 4a.对于 U-Boot 正确（不是 SPL），请调用 relocate\_code（）。此函数将 U-Boot 从其当前位置重新

\* 定位到重新定位目标由 board\_init\_f（） 计算。

\* 4b.For SPL, board\_init\_f() just returns (to crt0). There is no

\* code relocation in SPL.

\* 4b.对于 SPL，board\_init\_f（） 只返回 （到 crt0）。没有 在 SPL 中重新定位代码。

\* 5. Set up final environment for calling board\_init\_r(). This

\* environment has BSS (initialized to 0), initialized non-const

\* data (initialized to their intended value), and stack in system

\* RAM (for SPL moving the stack and GD into RAM is optional - see

\* CONFIG\_SPL\_STACK\_R). GD has retained values set by board\_init\_f().

\* 5.设置用于调用 board\_init\_r（） 的最终环境。这环境有 BSS（初始化为 0），初始化为非常量

\* 数据（初始化为预期值），并在系统中堆叠RAM（对于将堆栈和 GD 移动到 RAM 中的 SPL 是可选的 – 请参

\* 阅 CONFIG\_SPL\_STACK\_R）。GD 保留了由 board\_init\_f（） 设置的值。

\* 6. For U-Boot proper (not SPL), some CPUs have some work left to do

\* at this point regarding memory, so call c\_runtime\_cpu\_setup.

\* 6.对于U-Boot正确（不是SPL），某些CPU还有一些工作要做此时关于内存，请调用

\* c\_runtime\_cpu\_setup。

\* 7. Branch to board\_init\_r().

\* 7.分支到 board\_init\_r（）

\* For more information see 'Board Initialisation Flow in README.

\*/

/\*

\* entry point of crt0 sequence

\*/