Bootloader简介

联航精英训练营

嵌入式Linux系统的四个层次

1. **引导加载程序**：包括固化在固件(firmware)中的 boot 代码(可选)，和 Bootloader 两大部分。
2. **Linux 内核**：特定于嵌入式板子的定制内核以及内核的启动参数。
3. **文件系统**：包括根文件系统和建立于 Flash 内存设备之上文件系统。通常用 ramdisk 来作为 rootfs。
4. **用户应用程序**：特定于用户的应用程序。有时在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户界面。

PC机的启动

- PC 机中的引导加载程序由 BIOS(其本质就是一段固件程序)和位于硬盘 MBR 中的 OS Boot Loader（比如，LILO 和 GRUB 等）一起组成。
- BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 Boot Loader 读到系统的 RAM 中，然后将控制权交给 OS Boot Loader。
- Boot Loader 的主要运行任务就是将内核映象从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，也即开始启动操作系统。

嵌入式启动

- 在嵌入式系统中，通常并没有像 BIOS 那样的固件程序（注，有的嵌入式 CPU 也会内嵌一段短小的启动程序），因此整个系统的加载启动任务就完全由 Boot Loader 来完成。比如在一个基于 ARM920T Core 的嵌入式系统中，系统在上电或复位时通常都从地址 0x00000000 处开始执行，而在这个地址处安排的通常就是系统的 Boot Loader 程序。

Boot Loader 的概念

- Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

Boot Loader 所支持的 CPU 和嵌入式板

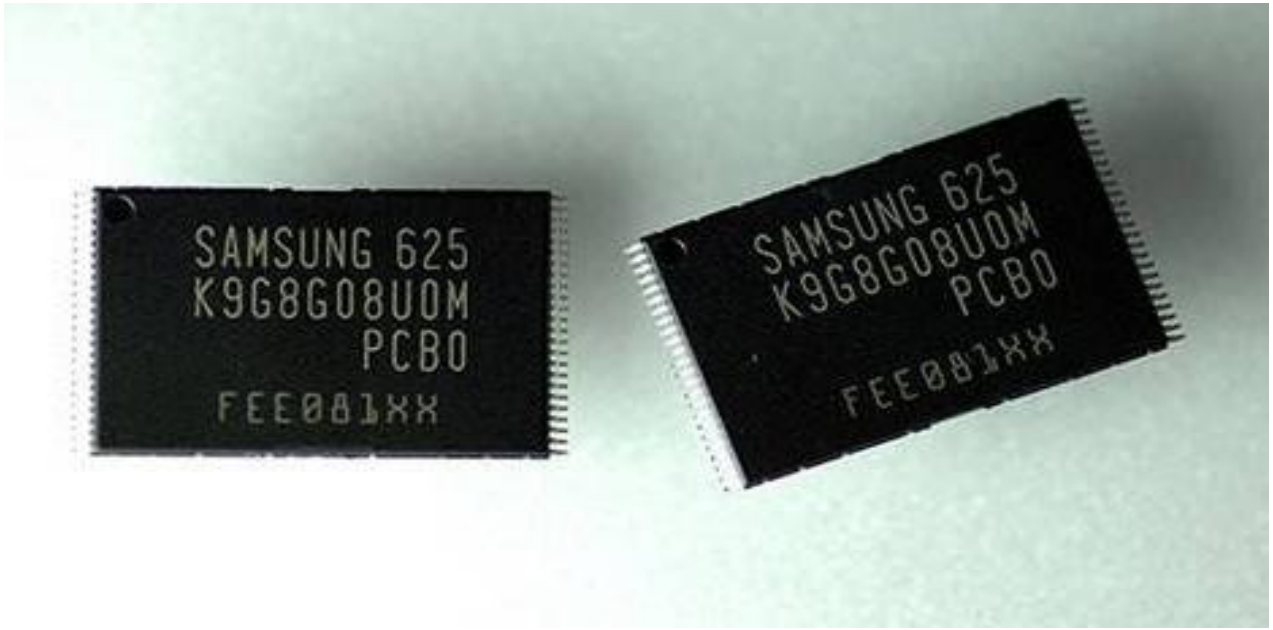
- 每种不同的 CPU 体系结构都有不同的 Boot Loader。有些 Boot Loader 也支持多种体系结构的 CPU，比如 U-Boot 就同时支持 ARM 体系结构和MIPS 体系结构。除了依赖于 CPU 的体系结构外，Boot Loader 实际上也依赖于具体的嵌入式板级设备的配置。这也就是说，对于两块不同的嵌入式板而言，即使它们是基于同一种 CPU 而构建的，要想让运行在一块板子上的 Boot Loader 程序也能运行在另一块板子上，通常也都需要修改 Boot Loader 的源程序。

Boot Loader 的安装媒介

- 系统加电或复位后，所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。比如，基于 ARM920T Core 的 CPU 在复位时通常都从地址 0x00000000 取它的第一条指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备(比如：ROM、EEPROM 或 FLASH 等)被映射到这个预先安排的地址上。因此在系统加电后，CPU 将首先执行 Boot Loader 程序。

Flash

- Flash Memory中文名字叫闪存，是一种长寿命的非易失性（在断电情况下仍能保持所存储的数据信息）的存储器。

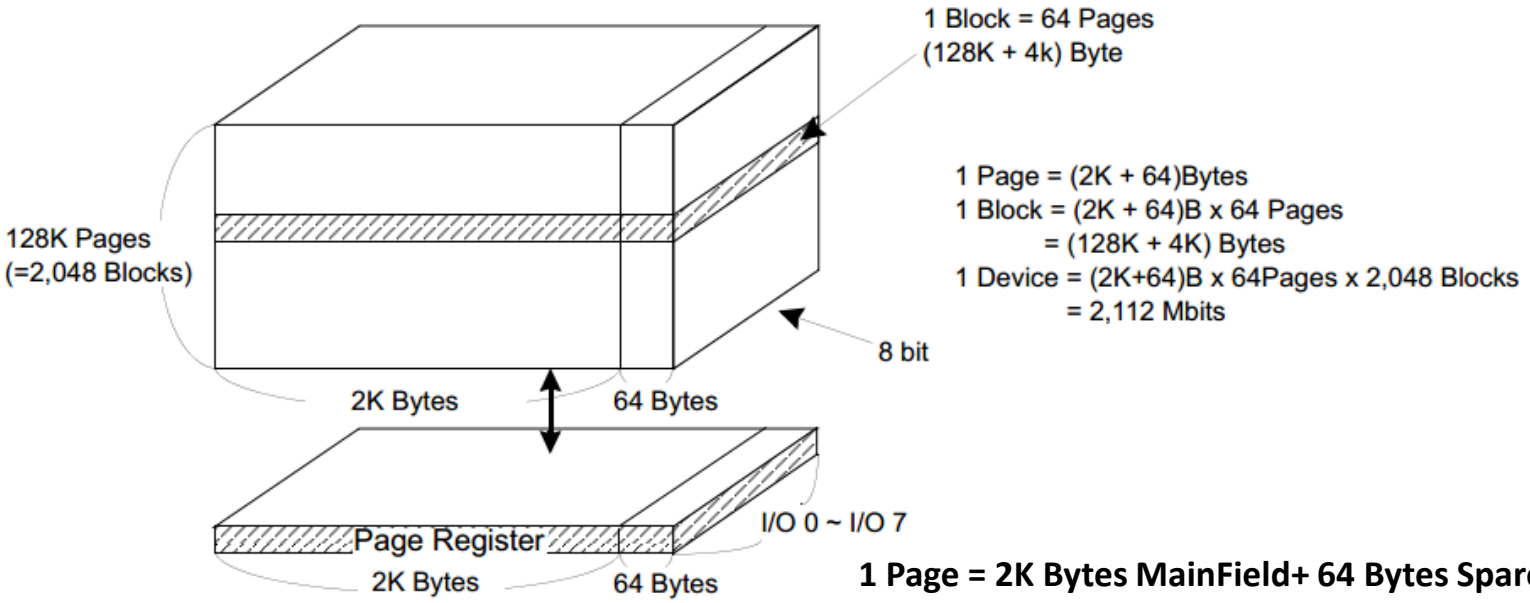


Nand Flash与Nor Flash

- Nor Flash和Nand Flash是现在市场上两种主要的非易失闪存技术。Intel于1988年首先开发出Nor Flash技术，彻底改变了原先由EPROM和EEPROM一统天下的局面。紧接着，1989年，东芝公司发表了Nand Flash结构，强调降低每比特的成本，更高的性能，并且象磁盘一样可以通过接口轻松升级。
  - Nor的读速度比Nand稍快一些
  - Nand的写入速度比Nor快很多
  - Nand的擦除速度远比Nor的快
  - 大多数写入操作需要先进行擦除操作
  - Nand的擦除单元更小，相应的擦除电路更少

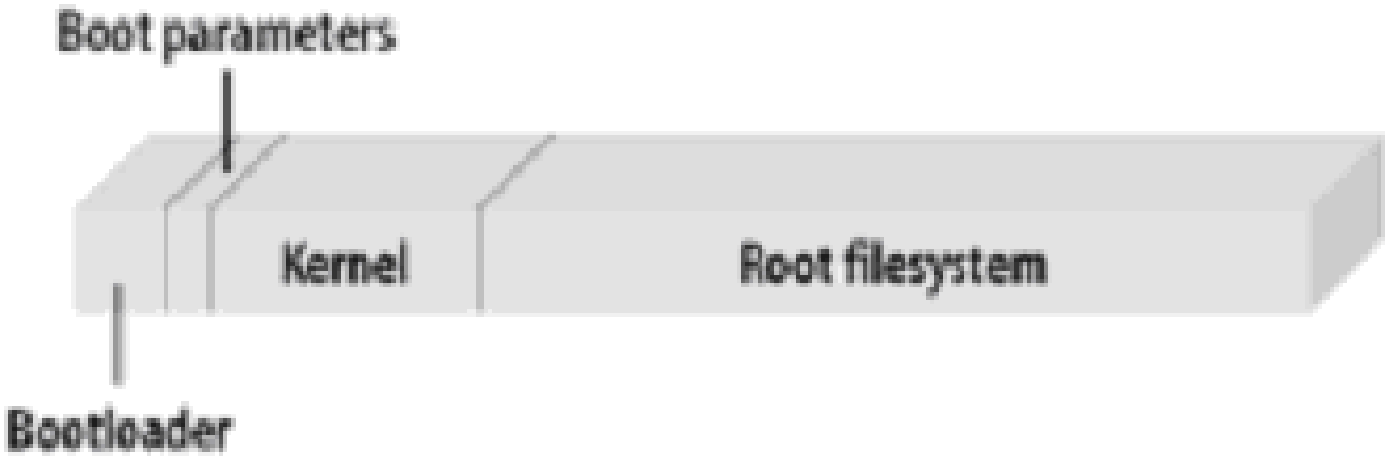
Nand 和Nor Flash的接口区别

- NOR flash带有SRAM接口，线性寻址，可以很容易地存取其内部的每一个字节
- NAND flash使用复用接口和控制IO多次寻址存取数据
- NAND读和写操作采用不同大小（512、2K字节）的块，这一点有点像硬盘管理此类操作易于取代硬盘等类似的块设备



1 Page = 2K Bytes MainField+ 64 Bytes Spare Field

固态存储设备的典型空间分配结构



同时装有 **Boot Loader**、**内核的启动参数**、**内核映像**和**根文件系统映像**的固态存储设备的典型空间分配结构图

Boot Loader 的工作模式

启动加载模式：

在这种模式下，Bootloader从目标机上的某个固态存储设备上将操作系统加载到RAM中运行，整个过程并没有用户的介入。通常当产品最终定型后，我们会采这种工作模式。

下载模式：

在这种模式下，目标机上的Bootloader将通过串口或网络等通信手段从开发主机（Host）上下载内核映像和根文件系统映像等到RAM中。然后可以再被Bootloader写到目标机上的固态存储媒质中，或者直接进行系统的引导。通常这种方式适用于开发人员开发过程中使用。

BOOT LOADER 的主要任务与典型结构框架

假设

- 首先我们做一个假定，那就是：**假定内核映像与根文件系统映像都被加载到 RAM 中运行**。之所以提出这样一个假设前提是因为，在嵌入式系统中内核映像与根文件系统映像也可以直接在 ROM 或 Flash 这样的固态存储设备中直接运行。但这种做法无疑是以运行速度的牺牲为代价的。
- 从操作系统的角度看，Boot Loader 的**总目标**就是正确地调用内核来执行。

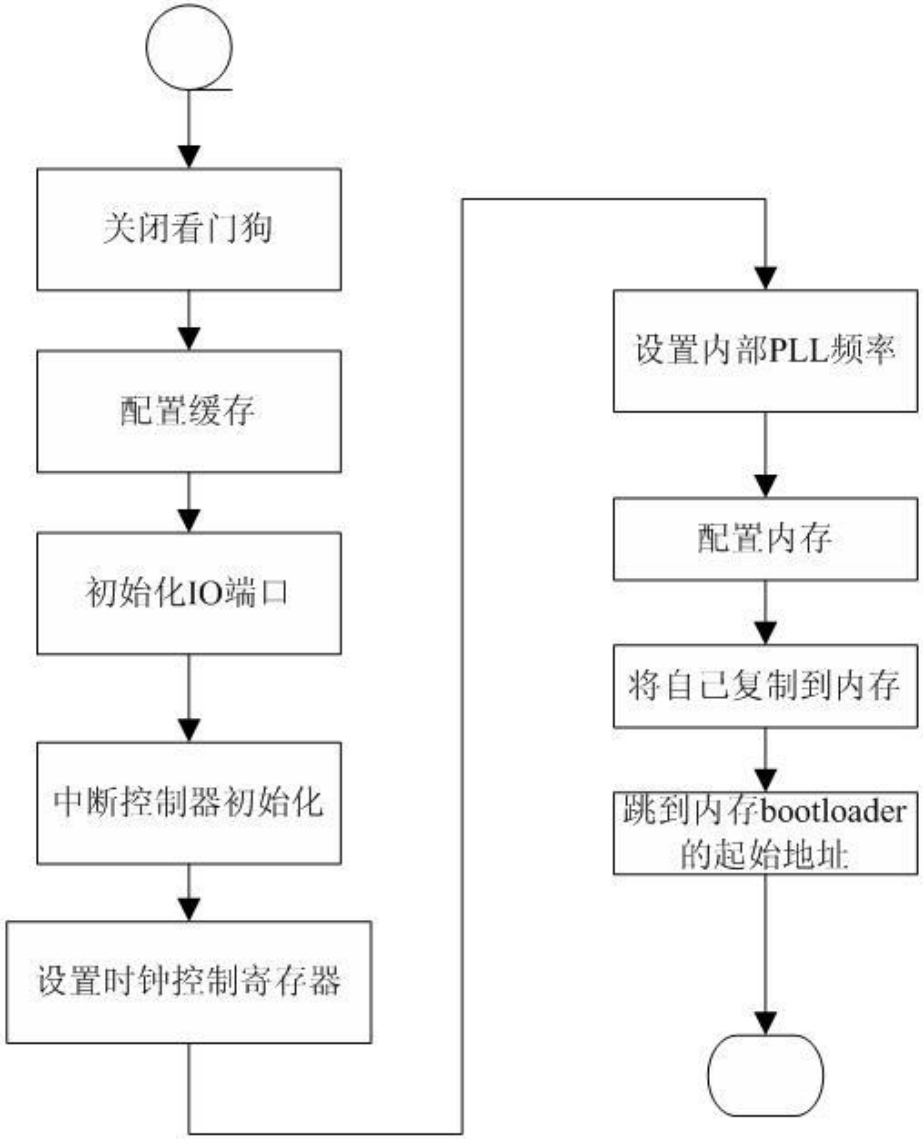
Bootloader的结构

- 由于 Boot Loader 的实现依赖于 CPU 的体系结构，因此大多数 Boot Loader 都分为 **stage1** 和 **stage2** 两大部分。
- 依赖于 CPU 体系结构的代码，比如设备初始化代码等，通常都放在 **stage1** 中，而且通常都用汇编语言来实现，以达到短小精悍的目的。
- 而 **stage2** 则通常用C语言来实现，这样可以实现给复杂的功能，而且代码会具有更好的可读性和可移植性。

Stage1

Boot Loader 的 stage1 通常包括以下步骤(以执行的先后顺序)：

1. 硬件设备初始化。
2. 为加载 Boot Loader 的 stage2 准备 RAM 空间。
3. 拷贝 Boot Loader 的 stage2 到 RAM 空间中。
4. 设置好堆栈。
5. 跳转到 stage2 的 C 入口点。


关闭看门狗
配置缓存
初始化IO端口
中断控制器初始化
设置时钟控制寄存器
设置内部PLL频率
配置内存
将自己复制到内存
跳到内存bootloader
的起始地址

基本的硬件初始化

这是 Boot Loader 一开始就执行的操作，其目的是为 stage2 的执行以及随后的 kernel 的执行准备好一些基本的硬件环境。它通常包括以下步骤（以执行的先后顺序）：

1．**屏蔽所有的中断。**为中断提供服务通常是 OS 设备驱动程序的责任，因此在 Boot Loader 的执行全过程中可以不必响应任何中断。中断屏蔽可以通过写 CPU 的中断屏蔽寄存器或状态寄存器（比如 ARM 的 CPSR 寄存器）来完成。

2．**设置 CPU 的速度和时钟频率。**

3．**RAM 初始化。**包括正确地设置系统的内存控制器的功能寄存器以及各内存库控制寄存器等。

4．**初始化 LED。**典型地，通过 GPIO 来驱动 LED，其目的是表明系统的状态是 OK 还是 Error。如果板子上没有 LED，那么也可以通过初始化 UART 向串口打印 Boot Loader 的 Logo 字符信息来完成这一点。

5．**关闭 CPU 内部指令 / 数据 cache。**

Stage2

Boot Loader 的 stage2 通常包括以下步骤(以执行的先后顺序)：

1. 初始化本阶段要使用到的硬件设备。
2. 检测系统内存映射(memory map)。
3. 将 kernel 映像和根文件系统映像从 flash 上读到 RAM 空间中。
4. 为内核设置启动参数。
5. 调用内核。

Bootloader家族

Bootloader	介　绍	X86	ARM	PowerPC
GRUB	LILO的升级程序，常用于引导基于X86体系结构的Linux操作系统	是	否	否
BLOB	LART等硬件平台的引导程序	否	是	否
RedBoot	基于eCos的引导程序	否	是	否
U-Boot	目前最常用的bootloader	否	是	否
Vivi	韩国的MIZI公司开发的支持S3C24X0的bootloader	否	是	否

GNU C家族扩展

见参考资料《C语言家族扩展》

GNU 工具链

- as　GNU的汇编器是 GNU Binutils工具集中最重要的工具之一。as 工具主要用来将汇编语言编写的源程 序转换成二进制形式的目标代码。
- ld　GNU的链接器同as一样，ld也是GNU Binutils工具集中重要的工具，Linux 使用 ld作为标准的链接程 序，由汇编器产生的目标代码是不能直接在计算机上运行的，它必须经过链接器的处理才能 生成可执行代码，链接是创建一个可执行程序的最后一个步骤，ld可以将多个目标文件链接 成为可执行程序，同时指定了程序在运行时是如何执行的。
- add2line　将地址转换成文件名和行号以及所对应的函数
- ar 从文件中创建、修改档案文件（例如*.a的静态库文件），以及在档案文件中抽取文件（例如从.a文件中抽取出.o文件）
- nm 用于列出目标文件、库或者可执行文件中代码符号及代码符号所对应的程序开始地址
- objcopy　可以把目标文件的内容从一种文件格式 复制到另一种格式的目标文件中

- objdump 显示目标文件信息，可以反编译二进制文件，也可以对对象文件进行反汇编，并查看机器代码。
- readelf 显示elf文件信息，可以显示符号、段信息、二进制文件格式的信息等，这在分析编译器如何从源代码创建二进制文件时非常有用。
- ranlib 生成索引以加快对归档文件的访问，并将其保存到这个归档文件中。
- size 列出目标模块或文件的段信息
- strings 用于显示程序文件中可显示的字符串
- strip 用于去除程序文件中的符号信息，以减少程序文件的大小

UBOOT的移植

如何获得源码包

- http://www.denx.de/wiki/U-Boot

- 下载合适版本
- tar -xzvf U-Boot-2010.3.tar.gz

U-Boot工程目录结构

- 在顶层目录下有18个子目录，分别存放和管理不同的源程序。这些目录中所要存放的文件有其规则，可以分为3类。

- 第1类目录是与处理器体系结构或者开发板硬件直接相关；
- 第2类目录是一些通用的函数或者驱动程序；
- 第3类目录是U-Boot的应用程序、工具或者文档。

Uboot的编译

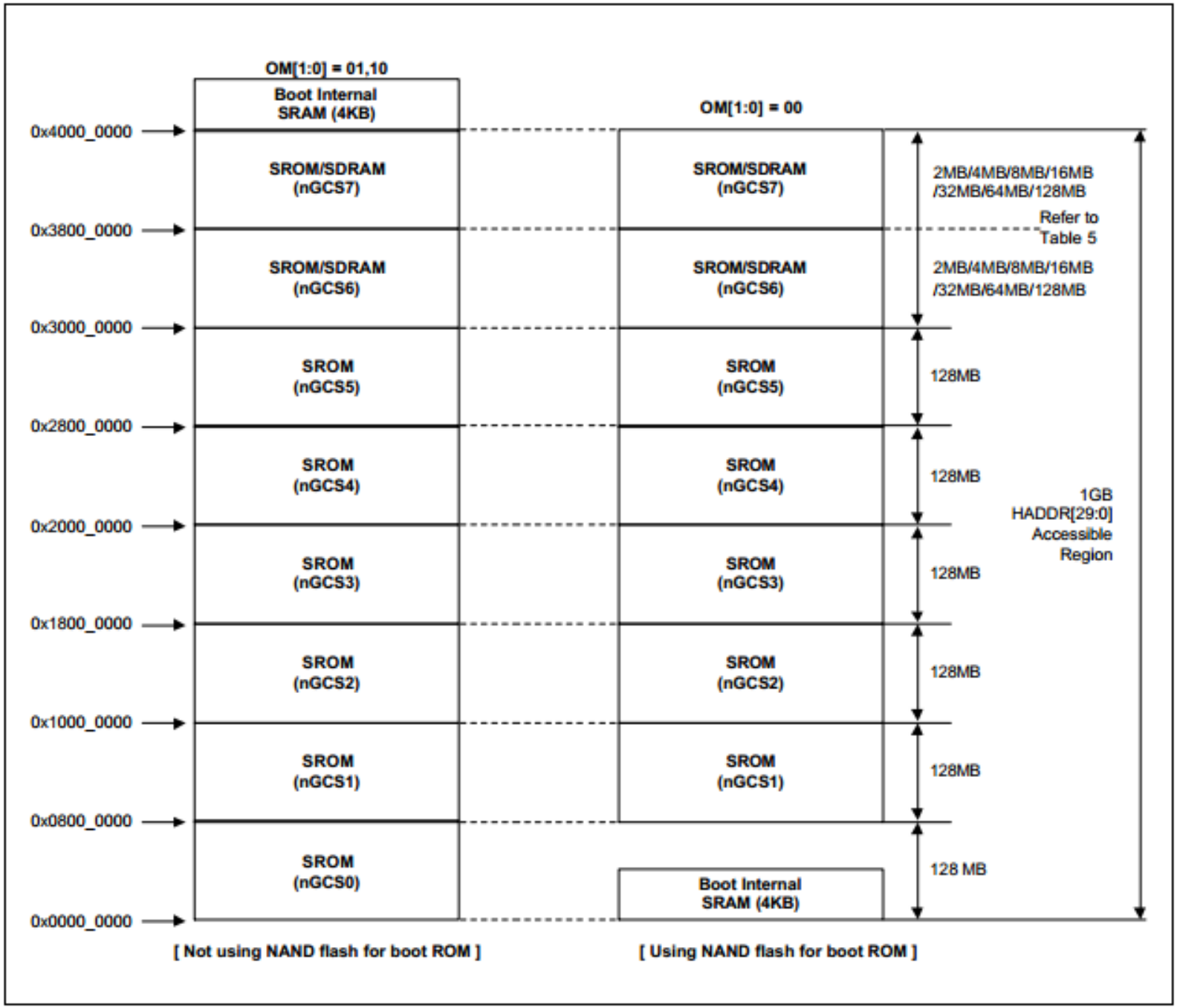
- 见移植手册

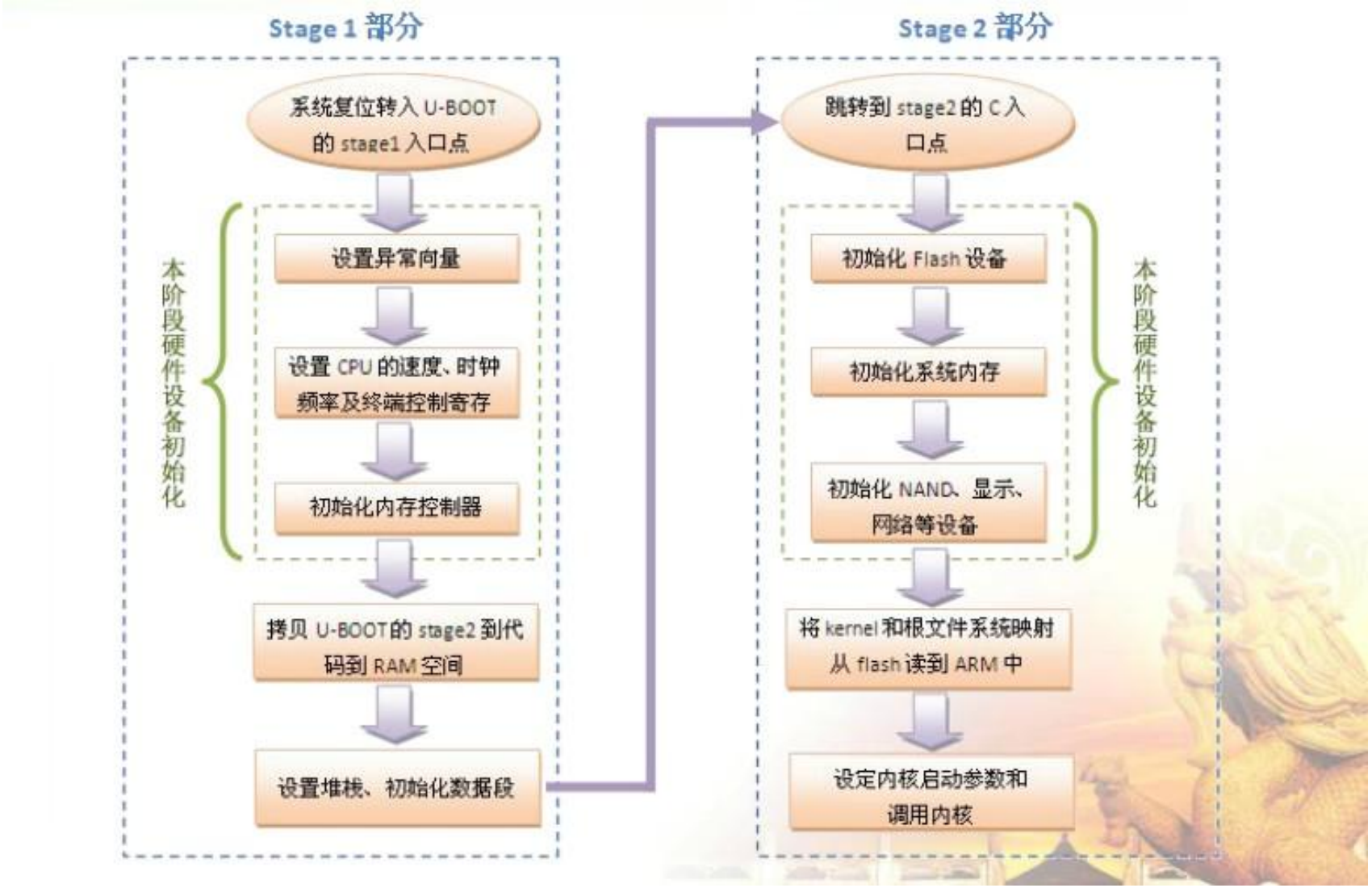
U-Boot Stage1

- U-Boot中Stage1阶段的入口点在cpu/xxxx/start.S文件中，针对本平台为cpu/arm920t/目录下的start.S文件。

- 在本阶段的移植中我们将进行两个方面的工作：

1. 通过点亮开发板中的LED指示灯来验证程序是否被执行
2. 修改CPU时钟、主频以及中断等硬件资源


OM[1:0] = 01,10
Boot Internal SRAM (4KB)
SROM/SDRAM (nGCS7)
SROM/SDRAM (nGCS6)
SROM (nGCS5)
SROM (nGCS4)
SROM (nGCS3)
SROM (nGCS2)
SROM (nGCS1)
SROM (nGCS0)
0x4000_0000
0x3800_0000
0x3000_0000
0x2800_0000
0x2000_0000
0x1800_0000
0x1000_0000
0x0800_0000
0x0000_0000
[Not using NAND flash for boot ROM]
OM[1:0] = 00
SROM/SDRAM (nGCS7)
SROM/SDRAM (nGCS6)
SROM (nGCS5)
SROM (nGCS4)
SROM (nGCS3)
SROM (nGCS2)
SROM (nGCS1)
Boot Internal SRAM (4KB)
[Using NAND flash for boot ROM]
2MB/4MB/8MB/16MB /32MB/64MB/128MB
Refer to Table 5
2MB/4MB/8MB/16MB /32MB/64MB/128MB
128MB
128MB
128MB
128MB
128 MB
1GB HADDR[29:0] Accessible Region


Stage 1 部分
系统复位转入 U-BOOT 的 stage1 入口点
设置异常向量
设置 CPU 的速度、时钟频率及终端控制寄存
初始化内存控制器
拷贝 U-BOOT的 stage2 到代码到 RAM 空间
设置堆栈、初始化数据段
本阶段硬件设备初始化
Stage 2 部分
跳转到 stage2 的 C 入口点
初始化 Flash 设备
初始化系统内存
初始化 NAND、显示、网络等设备
将 kernel 和根文件系统映射从 flash 读到 ARM 中
设定内核启动参数和调用内核
本阶段硬件设备初始化

移植操作详见移植手册