基础8-嵌入式系统 Boot Loader 技术内幕

1. 引言

在专用的嵌入式板子运行 GNU/Linux 系统已经变得越来越流行。一个嵌入式 Linux 系统从软件的角度看通常可以分为四个层次：

1. **引导加载程序**。包括固化在固件(firmware)中的 boot 代码(可选)，和 Boot Loader 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 ram disk 来作为 root fs。

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。常用的嵌入式 GUI 有：MicroWindows 和 MiniGUI 懂。

引导加载程序是系统加电后运行的第一段软件代码。回忆一下 PC 的体系结构我们可以知道，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。

而在嵌入式系统中，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 ARM7TDMI 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。

本文将从 Boot Loader 的概念、Boot Loader 的主要任务、Boot Loader 的框架结构以及 Boot Loader 的安装等四个方面来讨论嵌入式系统的 Boot Loader。

2. Boot Loader 的概念

简单地说，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。

通常，Boot Loader 是严重地依赖于硬件而实现的，特别是在嵌入式世界。因此，在嵌入式世界里建立一个通用的 Boot Loader 几乎是不可能的。尽管如此，我们仍然可以对 Boot Loader 归纳出一些通用的概念来，以指导用户特定的 Boot Loader 设计与实现。

1. 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。

2. Boot Loader 的安装媒介（Installation Medium）

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 ARM7TDMI 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。

下图1就是一个同时装有 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。

图1 固态存储设备的典型空间分配结构

图1 固态存储设备的典型空间分配结构

3. 用来控制 Boot Loader 的设备或机制

主机和目标机之间一般通过串口建立连接，Boot Loader 软件在执行时通常会通过串口来进行 I/O，比如：输出打印信息到串口，从串口读取用户控制字符等。

4. Boot Loader 的启动过程是单阶段（Single Stage）还是多阶段（Multi-Stage）

通常多阶段的 Boot Loader 能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的 Boot Loader 大多都是 2 阶段的启动过程，也即启动过程可以分为 stage 1 和 stage 2 两部分。而至于在 stage 1 和 stage 2 具体完成哪些任务将在下面讨论。

5. Boot Loader 的操作模式 (Operation Mode)

大多数 Boot Loader 都包含两种不同的操作模式："启动加载"模式和"下载"模式，这种区别仅对于开发人员才有意义。但从最终用户的角度看，Boot Loader 的作用就是用来加载操作系统，而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载（Boot loading）模式：这种模式也称为"自主"（Autonomous）模式。也即 Boot Loader 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。这种模式是 Boot Loader 的正常工作模式，因此在嵌入式产品发布的时侯，Boot Loader 显然必须工作在这种模式下。

下载（Downloading）模式：在这种模式下，目标机上的 Boot Loader 将通过串口连接或网络连接等通信手段从主机（Host）下载文件，比如：下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Boot Loader 保存到目标机的 RAM 中，然后再被 Boot Loader 写到目标机上的FLASH 类固态存储设备中。Boot Loader 的这种模式通常在第一次安装内核与根文件系统时被使用；此外，以后的系统更新也会使用 Boot Loader 的这种工作模式。工作于这种模式下的 Boot Loader 通常都会向它的终端用户提供一个简单的命令行接口。

像 Blob 或 U-Boot 等这样功能强大的 Boot Loader 通常同时支持这两种工作模式，而且允许用户在这两种工作模式之间进行切换。比如，Blob 在启动时处于正常的启动加载模式，但是它会延时 10 秒等待终端用户按下任意键而将 blob 切换到下载模式。如果在 10 秒内没有用户按键，则 blob 继续启动 Linux 内核。

6. BootLoader 与主机之间进行文件传输所用的通信设备及协议

最常见的情况就是，目标机上的 Boot Loader 通过串口与主机之间进行文件传输，传输协议通常是 xmodem / ymodem / zmodem 协议中的一种。但是，串口传输的速度是有限的，因此通过以太网连接并借助 TFTP 协议来下载文件是个更好的选择。

此外，在论及这个话题时，主机方所用的软件也要考虑。比如，在通过以太网连接和 TFTP 协议来下载文件时，主机方必须有一个软件用来的提供 TFTP 服务。

在讨论了 BootLoader 的上述概念后，下面我们来具体看看 BootLoader 的应该完成哪些任务。

3. Boot Loader 的主要任务与典型结构框架

在继续本节的讨论之前，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。

另外，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。

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

- 硬件设备初始化。
- 为加载 Boot Loader 的 stage2 准备 RAM 空间。
- 拷贝 Boot Loader 的 stage2 到 RAM 空间中。

- 设置好堆栈。
- 跳转到 stage2 的 C 入口点。

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

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

3.1 Boot Loader 的 stage1

3.1.1 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。**

3.1.2 为加载 stage2 准备 RAM 空间

为了获得更快的执行速度，通常把 stage2 加载到 RAM 空间中来执行，因此必须为加载 Boot Loader 的 stage2 准备好一段可用的 RAM 空间范围。

由于 stage2 通常是 C 语言执行代码，因此在考虑空间大小时，除了 stage2 可执行映象的大小外，还必须把堆栈空间也考虑进来。此外，空间大小最好是 memory page 大小(通常是 4KB)的倍数。一般而言，1M 的 RAM 空间已经足够了。具体的地址范围可以任意安排，比如 blob 就将它的 stage2 可执行映像安排到从系统 RAM 起始地址 0xc0200000 开始的 1M 空间内执行。但是，将 stage2 安排到整个 RAM 空间的最顶 1MB(也即(RamEnd-1MB) - RamEnd)是一种值得推荐的方法。

为了后面的叙述方便，这里把所安排的 RAM 空间范围的大小记为：stage2_size(字节)，把起始地址和终止地址分别记为：stage2_start 和 stage2_end(这两个地址均以 4 字节边界对齐)。因此：

```
1 stage2_end=stage2_start+stage2_size
```

另外，还必须确保所安排的地址范围的的确确是可读写的 RAM 空间，因此，必须对你所安排的地址范围进行测试。具体的测试方法可以采用类似于 blob 的方法，也即：以 memory page 为被测试单位，测试每个 memory page 开始的两个字是否是可读写的。为了后面叙述的方便，我们记这个检测算法为：test_mempage，其具体步骤如下：

1. 先保存 memory page 一开始两个字的内容。

2. 向这两个字中写入任意的数字。比如：向第一个字写入 0x55，第 2 个字写入 0xaa。

3. 然后，立即将这两个字的内容读回。显然，我们读到的内容应该分别是 0x55 和 0xaa。如果不是，则说明这个 memory page 所占据的地址范围不是一段有效的 RAM 空间。

4. 再向这两个字中写入任意的数字。比如：向第一个字写入 0xaa，第 2 个字中写入 0x55。

5. 然后，立即将这两个字的内容立即读回。显然，我们读到的内容应该分别是 0xaa 和 0x55。如果不是，则说明这个 memory page 所占据的地址范围不是一段有效的 RAM 空间。

6. 恢复这两个字的原始内容。测试完毕。

为了得到一段干净的 RAM 空间范围，我们也可以将所安排的 RAM 空间范围进行清零操作。

3.1.3 拷贝 stage2 到 RAM 中

拷贝时要确定两点：(1) stage2 的可执行映象在固态存储设备的存放起始地址和终止地址；(2) RAM 空间的起始地址。

3.1.4 设置堆栈指针 sp

堆栈指针的设置是为了执行 C 语言代码作好准备。通常我们可以把 sp 的值设置为(stage2_end-4)，也即在 3.1.2 节所安排的那个 1MB 的 RAM 空间的最顶端(堆栈向下生长)。

此外，在设置堆栈指针 sp 之前，也可以关闭 led 灯，以提示用户我们准备跳转到 stage2。

经过上述这些执行步骤后，系统的物理内存布局应该如下图2所示。

3.1.5 跳转到 stage2 的 C 入口点

在上述一切都就绪后，就可以跳转到 Boot Loader 的 stage2 去执行了。比如，在 ARM 系统中，这可以通过修改 PC 寄存器为合适的地址来实现。

图2 bootloader 的 stage2 可执行映象刚被拷贝到 RAM 空间时的系统内存布局

图2 bootloader 的 stage2 可执行映象刚被拷贝到 RAM 空间时的系统内存布局

3.2 Boot Loader 的 stage2

正如前面所说，stage2 的代码通常用 C 语言来实现，以便于实现更复杂的功能和取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是，在编译和链接 boot loader 这样的程序时，我们不能使用 glibc 库中的任何支持函数。其原因是显而易见的。这就给我们带来一个问题，那就是从那里跳转进 main() 函数呢？直接把 main() 函数的起始地址作为整个 stage2 执行映像的入口点或许是最直接的想法。但是这样做有两个缺点：1)无法通过main() 函数传递函数参数；2)无法处理 main() 函数返回的情况。一种更为巧妙的方法是利用 trampoline(弹簧床)的概念。也即，用汇编语言写一段trampoline 小程序，并将这段 trampoline 小程序来作为 stage2 可执行映象的执行入口点。然后我们可以在 trampoline 汇编小程序中用 CPU 跳转指令跳入 main() 函数中去执行；而当 main() 函数返回时，CPU 执行路径显然再次回到我们的 trampoline 程序。简而言之，这种方法的思想就是：用这段 trampoline 小程序来作为 main() 函数的外部包裹(external wrapper)。

下面给出一个简单的 trampoline 程序示例(来自blob)：

```
1  .text
2  .globl _trampoline
3  _trampoline:
4      bl  main
5      /* if main ever returns we just call it again */
6      b   _trampoline
```

可以看出，当 main() 函数返回后，我们又用一条跳转指令重新执行 trampoline 程序——当然也就重新执行 main() 函数，这也就是 trampoline(弹簧床)一词的意思所在。

3.2.1初始化本阶段要使用到的硬件设备

这通常包括：（1）初始化至少一个串口，以便和终端用户进行 I/O 输出信息；（2）初始化计时器等。

在初始化这些设备之前，也可以重新把 LED 灯点亮，以表明我们已经进入 main() 函数执行。

设备初始化完成后，可以输出一些打印信息，程序名字字符串、版本号等。

3.2.2 检测系统的内存映射（memory map）

所谓内存映射就是指在整个 4GB 物理地址空间中有哪些地址范围被分配用来寻址系统的 RAM 单元。比如，在 SA-1100 CPU 中，从 0xC000,0000 开始的 512M 地址空间被用作系统的 RAM 地址空间，而在 Samsung S3C44B0X CPU 中，从 0x0c00,0000 到 0x1000,0000 之间的 64M 地址空间被用作系统的 RAM 地址空间。虽然 CPU 通常预留出一大段足够的地址空间给系统 RAM，但是在搭建具体的嵌入式系统时却不一定会实现 CPU 预留的全部 RAM 地址空间。也就是说，具体的嵌入式系统往往只把 CPU 预留的全部 RAM 地址空间中的一部分映射到 RAM 单元上，而让剩下的那部分预留 RAM 地址空间处于未使用状态。 **由于上述这个事实，因此 Boot Loader 的 stage2 必须在它想干点什么 (比如，将存储在 flash 上的内核映像读到 RAM 空间中) 之前检测整个系统的内存映射情况，也即它必须知道 CPU 预留的全部 RAM 地址空间中的哪些被真正映射到 RAM 地址单元，哪些是处于 "unused" 状态的。**

(1) 内存映射的描述

可以用如下数据结构来描述 RAM 地址空间中的一段连续(continuous)的地址范围：

```
typedef struct memory_area_struct {
    u32 start; /* the base address of the memory region */
    u32 size; /* the byte number of the memory region */
    int used;
} memory_area_t;
```

这段 RAM 地址空间中的连续地址范围可以处于两种状态之一：(1)used=1，则说明这段连续的地址范围已被实现，也即真正地被映射到 RAM 单元上。(2)used=0，则说明这段连续的地址范围并未被系统所实现，而是处于未使用状态。

基于上述 memory_area_t 数据结构，整个 CPU 预留的 RAM 地址空间可以用一个 memory_area_t 类型的数组来表示，如下所示：

```
memory_area_t memory_map[NUM_MEM_AREAS] = {
    [0 ... (NUM_MEM_AREAS - 1)] = {
        .start = 0,
        .size = 0,
        .used = 0
    },
};
```

(2) 内存映射的检测

下面我们给出一个可用来检测整个 RAM 地址空间内存映射情况的简单而有效的算法：

```
/* 数组初始化 */
for(i = 0; i < NUM_MEM_AREAS; i++)
    memory_map[i].used = 0;
/* first write a 0 to all memory locations */
for(addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE)
    * (u32 *)addr = 0;
for(i = 0, addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE) {
     /*
      * 检测从基地址 MEM_START+i*PAGE_SIZE 开始,大小为
* PAGE_SIZE 的地址空间是否是有效的RAM地址空间。
      */
     调用3.1.2节中的算法test_mempage();
     if ( current memory page isnot a valid ram page) {
        /* no RAM here */
        if(memory_map[i].used )
            i++;
        continue;
    }

    /*
     * 当前页已经是一个被映射到 RAM 的有效地址范围
     * 但是还要看看当前页是否只是 4GB 地址空间中某个地址页的别名?
     */
    if(* (u32 *)addr != 0) { /* alias? */
        /* 这个内存页是 4GB 地址空间中某个地址页的别名 */
        if ( memory_map[i].used )
            i++;
        continue;
    }

    /*
     * 当前页已经是一个被映射到 RAM 的有效地址范围
     * 而且它也不是 4GB 地址空间中某个地址页的别名。
     */
    if (memory_map[i].used == 0) {
        memory_map[i].start = addr;
        memory_map[i].size = PAGE_SIZE;
        memory_map[i].used = 1;
    } else {
        memory_map[i].size += PAGE_SIZE;
    }
} /* end of for (…) */
```

在用上述算法检测完系统的内存映射情况后，Boot Loader 也可以将内存映射的详细信息打印到串口。

3.2.3 加载内核映像和根文件系统映像

(1) 规划内存占用的布局

这里包括两个方面：(1)内核映像所占用的内存范围；（2）根文件系统所占用的内存范围。在规划内存占用的布

局时，主要考虑基地址和映像的大小两个方面。

对于内核映像，一般将其拷贝到从(MEM_START＋0x8000) 这个基地址开始的大约1MB大小的内存范围内(嵌入式Linux 的内核一般都不操过 1MB)。为什么要把从 MEM_START 到 MEM_START＋0x8000 这段 32KB 大小的内存空出来呢？这是因为 Linux 内核要在这段内存中放置一些全局数据结构，如：启动参数和内核页表等信息。

而对于根文件系统映像，则一般将其拷贝到 MEM_START+0x0010,0000 开始的地方。如果用 Ramdisk 作为根文件系统映像，则其解压后的大小一般是1MB。

（2）从 Flash 上拷贝

由于像 ARM 这样的嵌入式 CPU 通常都是在统一的内存地址空间中寻址 Flash 等固态存储设备的，因此从 Flash 上读取数据与从 RAM 单元中读取数据并没有什么不同。用一个简单的循环就可以完成从 Flash 设备上拷贝映像的工作：

```
while(count) {
    *dest++ = *src++; /* they are all aligned with word boundary */
    count -= 4; /* byte number */
};
```

3.2.4 设置内核的启动参数

应该说，在将内核映像和根文件系统映像拷贝到 RAM 空间中后，就可以准备启动 Linux 内核了。但是在调用内核之前，应该作一步准备工作，即：设置 Linux 内核的启动参数。

Linux 2.4.x 以后的内核都期望以标记列表(tagged list)的形式来传递启动参数。启动参数标记列表以标记 ATAG_CORE 开始，以标记 ATAG_NONE 结束。每个标记由标识被传递参数的 tag_header 结构以及随后的参数值数据结构来组成。数据结构 tag 和 tag_header 定义在 Linux 内核源码的include/asm/setup.h 头文件中：

```
/* The list ends with an ATAG_NONE node. */
#define ATAG_NONE   0x00000000
struct tag_header {
    u32 size; /* 注意，这里size是字数为单位的 */
    u32 tag;
};
……
struct tag {
    struct tag_header hdr;
    union {
        struct tag_core     core;
        struct tag_mem32    mem;
        struct tag_videotext    videotext;
        struct tag_ramdisk  ramdisk;
        struct tag_initrd   initrd;
        struct tag_serialnr serialnr;
        struct tag_revision revision;
        struct tag_videolfb videolfb;
        struct tag_cmdline  cmdline;
        /*
         * Acorn specific
         */
        struct tag_acorn    acorn;
        /*
         * DC21285 specific
         */
        struct tag_memclk   memclk;
    } u;
};
```

在嵌入式 Linux 系统中，通常需要由 Boot Loader 设置的常见启动参数有：ATAG_CORE、ATAG_MEM、ATAG_CMDLINE、ATAG_RAMDISK、ATAG_INITRD等。

比如，设置 ATAG_CORE 的代码如下：

```
params = (struct tag *)BOOT_PARAMS;
    params->hdr.tag = ATAG_CORE;
    params->hdr.size = tag_size(tag_core);
    params->u.core.flags = 0;
    params->u.core.pagesize = 0;
    params->u.core.rootdev = 0;
    params = tag_next(params);
```

其中，BOOT_PARAMS 表示内核启动参数在内存中的起始基地址，指针 params 是一个 struct tag 类型的指针。宏 tag_next() 将以指向当前标记的指针为参数，计算紧临当前标记的下一个标记的起始地址。注意，内核的根文件系

统所在的设备ID就是在这里设置的。

下面是设置内存映射情况的示例代码：

```
for(i = 0; i < NUM_MEM_AREAS; i++) {
        if(memory_map[i].used) {
            params->hdr.tag = ATAG_MEM;
            params->hdr.size = tag_size(tag_mem32);
            params->u.mem.start = memory_map[i].start;
            params->u.mem.size = memory_map[i].size;

            params = tag_next(params);
        }
}
```

可以看出，在 memory_map [ ] 数组中，每一个有效的内存段都对应一个 ATAG_MEM 参数标记。

Linux 内核在启动时可以以命令行参数的形式来接收信息，利用这一点我们可以向内核提供那些内核不能自己检测的硬件参数信息，或者重载(override)内核自己检测到的信息。比如，我们用这样一个命令行参数字符串"console=ttyS0,115200n8"来通知内核以 ttyS0 作为控制台，且串口采用 "115200bps、无奇偶校验、8位数据位"这样的设置。下面是一段设置调用内核命令行参数字符串的示例代码：

```
char *p;
    /* eat leading white space */
    for(p = commandline; *p == ' '; p++)
        ;
    /* skip non-existent command lines so the kernel will still
    * use its default command line.
     */
    if(*p == '\0')
        return;
    params->hdr.tag = ATAG_CMDLINE;
    params->hdr.size = (sizeof(struct tag_header) + strlen(p) + 1 + 4) >> 2;
    strcpy(params->u.cmdline.cmdline, p);
    params = tag_next(params);
```

请注意在上述代码中，设置 tag_header 的大小时，必须包括字符串的终止符'\0'，此外还要将字节数向上圆整4个字节，因为 tag_header 结构中的size 成员表示的是字数。

下面是设置 ATAG_INITRD 的示例代码，它告诉内核在 RAM 中的什么地方可以找到 initrd 映象(压缩格式)以及它的大小：

```
params->hdr.tag = ATAG_INITRD2;
params->hdr.size = tag_size(tag_initrd);

params->u.initrd.start = RAMDISK_RAM_BASE;
params->u.initrd.size = INITRD_LEN;

params = tag_next(params);
```

下面是设置 ATAG_RAMDISK 的示例代码，它告诉内核解压后的 Ramdisk 有多大（单位是KB）：

```
params->hdr.tag = ATAG_RAMDISK;
params->hdr.size = tag_size(tag_ramdisk);

params->u.ramdisk.start = 0;
params->u.ramdisk.size = RAMDISK_SIZE; /* 请注意，单位是KB */
params->u.ramdisk.flags = 1; /* automatically load ramdisk */

params = tag_next(params);
```

最后，设置 ATAG_NONE 标记，结束整个启动参数列表：

```
static void setup_end_tag(void)
{
    params->hdr.tag = ATAG_NONE;
    params->hdr.size = 0;
}
```

3.2.5 调用内核

Boot Loader 调用 Linux 内核的方法是直接跳转到内核的第一条指令处，也即直接跳转到 MEM_START＋0x8000 地址处。在跳转时，下列条件要满足：

1. CPU 寄存器的设置：

- R0＝0；
- R1＝机器类型 ID；关于 Machine Type Number，可以参见 **linux/arch/arm/tools/mach-types**。
- R2＝启动参数标记列表在 RAM 中起始基地址；

2. CPU 模式：

- 必须禁止中断（IRQs和FIQs）；
- CPU 必须 SVC 模式；

3. Cache 和 MMU 的设置：

- MMU 必须关闭；
- 指令 Cache 可以打开也可以关闭；
- 数据 Cache 必须关闭；

如果用 C 语言，可以像下列示例代码这样来调用内核：

```
1  void (*theKernel)(int zero, int arch, u32 params_addr) =
2    (void (*)(int, int, u32))KERNEL_RAM_BASE;
3  ……
4  theKernel(0, ARCH_NUMBER, (u32) kernel_params_start);
```

注意，theKernel()函数调用应该永远不返回的。如果这个调用返回，则说明出错。

4. 关于串口终端

在 boot loader 程序的设计与实现中，没有什么能够比从串口终端正确地收到打印信息能更令人激动了。此外，向串口终端打印信息也是一个非常重要而又有效的调试手段。但是，我们经常会碰到串口终端显示乱码或根本没有显示的问题。造成这个问题主要有两种原因：(1) boot loader 对串口的初始化设置不正确。(2) 运行在 host 端的终端仿真程序对串口的设置不正确，这包括：波特率、奇偶校验、数据位和停止位等方面的设置。

此外，有时也会碰到这样的问题，那就是：在 boot loader 的运行过程中我们可以正确地向串口终端输出信息，但当 boot loader 启动内核后却无法看到内核的启动输出信息。对这一问题的原因可以从以下几个方面来考虑：

(1) 首先请确认你的内核在编译时配置了对串口终端的支持，并配置了正确的串口驱动程序。

(2) 你的 boot loader 对串口的初始化设置可能会和内核对串口的初始化设置不一致。此外，对于诸如 s3c44b0x 这样的 CPU，CPU 时钟频率的设置也会影响串口，因此如果 boot loader 和内核对其 CPU 时钟频率的设置不一致，也会使串口终端无法正确显示信息。

(3) 最后，还要确认 boot loader 所用的内核基地址必须和内核映像在编译时所用的运行基地址一致，尤其是对于 uClinux 而言。假设你的内核映像在编译时用的基地址是 0xc0008000，但你的 boot loader 却将它加载到 0xc0010000 处去执行，那么内核映像当然不能正确地执行了。

5. 结束语

Boot Loader 的设计与实现是一个非常复杂的过程。如果不能从串口收到那激动人心的"uncompressing linux..... done, booting the kernel……"内核启动信息，恐怕谁也不能说："嗨，我的 boot loader 已经成功地转起来了！"。