嵌入式操作系统

陈香兰
xlanchen@ustc.edu.cn
Fall 2009

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Boot Loader

- 本章从以下四个方面来讨论嵌入式系统的 Boot Loader，包括:
  - Boot Loader 的概念
  - Boot Loader 的主要任务
  - Boot Loader 的框架结构
  - Boot Loader 的安装
  - 部分开源的boot loader

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

嵌入式Linux的软件层次

- 在专用的嵌入式板子上运行 GNU/Linux 系统已变得越来越流行。
- 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及内核的启动参数

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

嵌入式Linux的软件层次（cont'd）

3. 文件系统。
   包括根文件系统和建立于 **Flash** 内存设备之上的文件系统
   通常用**RAM-Disk**来作为根文件系统
4. 用户应用程序。
   特定于用户的应用程序

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

嵌入式GUI

- 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（GUI）。
- 常用的嵌入式 GUI 有:
  - MicroWindows 和
  - MiniGUI等。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

引导加载程序

- 引导加载程序是系统加电后运行的第一段软件代码
- 例如PC机的引导加载程序，包括
  1. BIOS(其本质就是一段固件程序)
  2. 位于硬盘 MBR 中的 OS Boot Loader
     - 比如LILO、GRUB 等。

BIOS的主要任务是

1. 进行硬件检测和资源分配
2. 将MBR中的OS Boot Loader读到系统的 RAM 中
3. 将控制权交给 OS Boot Loader

Boot Loader的主要运行任务是

1. 将内核映象从硬盘上读到 RAM 中
2. 跳转到内核的入口点去运行，也即启动操作系统。

引导加载程序（cont'd）

- 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
  - 整个系统的加载启动任务完全由 **Boot Loader** 完成
    - 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。

Boot Loader 的概念

- **Boot Loader** 是在操作系统内核运行之前运行的第一段小程序。
  - 初始化硬件设备
  - 建立内存空间的映射图
    - 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。
  - 加载操作系统内核映象到**RAM**中，并将系统的控制权传递给它
    - 例如：**Linux**

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

通用的Boot Loader

- 在嵌入式世界里建立一个通用的 **Boot Loader** 几乎是不可能的
  - **Boot Loader**对硬件的依赖性非常强，特别是在嵌入式系统世界中
- 尽管如此，仍可对 **Boot Loader** 归纳出一些通用的概念，以指导用户特定的 **Boot Loader**设计与实现。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

支持的 CPU 和嵌入式板

Boot Loader依赖于

1. CPU 的体系结构
   - Ø 不同的CPU体系结构都有不同的Boot Loader
   - Ø 有些 Boot Loader 也支持多种CPU体系结构
     - 1 例如U-Boot同时支持ARM和MIPS体系结构
2. 具体的嵌入式板级设备的配置
   - Ø 对于两块不同的嵌入式板，即使它们基于同一种CPU，要想让运行在一块板子上的 Boot Loader也能运行在另一块板子上，通常也都需要修改 Boot Loader源程序

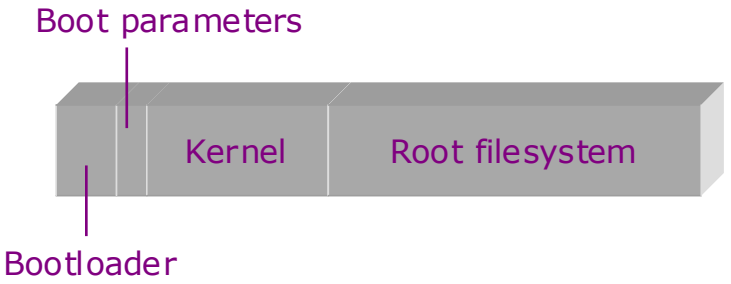
Boot Loader 的安装媒介

- 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 ARM7TDMI core 的 CPU 在复位时通常都从地址 0x00000000 取它的第一条指令。
- 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被映射到该预先安排的地址上。
  - 比如：ROM、EEPROM 或 FLASH 等
- 因此：**在系统加电后，CPU 将首先执行 Boot Loader 程序**。

嵌入式系统实验室
Embedded System Laboratory
Suzhou Institute for Advanced Study of USTC

固态存储设备的典型空间分配结构图

- 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



嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY

Boot Loader 的安装

- 烧写boot loader程序
  - 一般通过jtag烧写
  - 需要jtag连接器和PC端的烧写程序

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

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 在执行时常通过串口来进行 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。
  - 最常用的串口通信软件
    - **Linux**：**minicom**
    - **Windows**：附件中的超级终端

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Boot Loader 的启动过程

- **Boot Loader**的启动过程可以是
  - 单阶段（**Single Stage**）或
    - 一些只需完成很简单功能的**boot loader**可能是单阶段的
  - 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 **stage1** 和 **stage2** 两部分

Boot Loader 的操作模式

- 大多数 Boot Loader 包含两种不同的操作模式
  - 启动加载（Boot loading）模式和
  - 下载（Downloading）模式
- 这种区别仅对于开发人员才有意义
  - 从最终用户的角度看，Boot Loader 的作用就是加载操作系统，并不存在上述两种模式的区别

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

启动加载模式

- 也称为自主（Autonomous）模式
- 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（开发人员）的介入。
- 这种模式是 Boot Loader 的正常工作模式
  - 在嵌入式产品发布时，Boot Loader必须工作在该模式下

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

下载模式

- 目标机的 **Boot Loader**通过串口或网络等通信手段从主机（**Host**）下载文件
  - 比如内核映像和根文件系统映像
  - **Host target ram target FLASH**
- 该模式的使用时机
  - 通常在第一次安装内核与根文件系统时被使用
  - 也用于此后的系统更新
- 工作于该模式下的 **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

Boot Loader 的操作模式（cont'd）

- 一些功能强大的 Boot Loader 通常
  1. 同时支持这两种工作模式
     - 如Blob和U-Boot
  2. 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。

与boot loader两种模式相关的问题

- uClinux包编译好后，可根据需要编译出各种镜像文件
  - 也就是按照板子内存预定位置生成的二进制映象，一般是内核和文件系统的复合体
- 常见有
  - image.ram（常称为ram版内核）和
  - image.rom（常称为rom版内核）
- 通过在make时指定的不同编译选项生成

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

ram版内核

- 一般不压缩，通过boot loader加载到目标板内存的指定位置，然后可用boot loader跳转过去就把uclinux引导启动了
- Boot loader + ram版内核
  - 内核/驱动相关调试期间常用方式

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

rom版内核

- 不严格的理解可以说是把boot loader + ram版烧写到flash内
- 上电或reset后首先执行boot loader初始化硬件功能，然后把压缩的内核映象解压释放到SDRAM指定地址，接着自动引导内核，启动uClinux
- 调试应用软件常用rom版镜像。

BootLoader 与主机之间进行文件传输所用的通信设备及协议

- 最常见通信设备是串口
  - 传输协议通常是 **xmodem**、**ymodem**、**zmodem**之一。
  - 但串口传输的速度有限
- 更好的选择是以太网
  - 使用**TFTP**协议
  - 主机方必须有一个软件提供 **TFTP** 服务

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Boot Loader的主要任务

- 系统假设：内核映像与根文件系统映像都被加载到 RAM 中运行。
  - 尽管在嵌入式系统中它们也可直接运行在 ROM 或 Flash这样的固态存储设备中。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加载并调用内核来执行。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Boot Loader的典型结构框架

- 由于 Boot Loader 的实现依赖于 CPU 体系结构，大多数 Boot Loader 都分为 stage1 和 stage2 两大部分
  - **Stage1**
    - 依赖于 CPU 体系结构，如设备初始化代码
    - 通常用汇编语言实现，短小精悍
  - **Stage2**
    - 通常用C语言
    - 可以实现复杂功能
    - 代码具有较好的可读性和可移植性

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Boot Loader的stage1

- Stage1直接运行在固态存储设备上，通常包括以下步骤
  - 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

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage1：硬件初始化

- 这是 Boot Loader 一开始就执行的操作
- 目的：为 stage2及kernel的执行准备好基本硬件环境
- 通常包括

1. 屏蔽所有的中断
   - 为中断提供服务通常是 OS或设备驱动程序的责任，在 Boot Loader阶段不必响应任何中断
   - 中断屏蔽可以通过写 CPU 的中断屏蔽寄存器或状态寄存器来完成
     - 比如 ARM 的 CPSR 寄存器

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage1：硬件初始化（cont‘d）

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中内存控制器的功能寄存器以及各CPU外的内存（Memory Bank）的控制寄存器等。
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。若板子上无LED，也可通过初始化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

Stage1： 为stage2 准备 RAM 空间

- 为获得更快的执行速度，通常**stage2**被加载到 **RAM** 中执行
- 因此必须为加载 **stage2** 准备好一段可用的 **RAM** 空间
  - 空间大小，应考虑
    - **stage2** 可执行映象的大小+堆栈空间
      - 因为**stage2** 通常是 **C** 语言代码
    - 此外，最好对齐到**memory page**大小(通常是 **4KB)**
    - 一般而言**1MB**已足够

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage1： 为stage2 准备 RAM 空间

- 具体的地址范围可以任意安排
  - 比如 **blob** 将它的 **stage2** 可执行映像安排系统的**RAM**中**0xc0200000**开始的**1M** 空间内
  - 值得推荐的是
    可以将 **stage2** 安排到整个**RAM**空间的最顶**1MB**
    也即**(RamEnd-1MB)**开始处
- 假设
  空间大小： **stage2_size**（字节）
  起始和终止地址分别为： **stage2_start** 和 **stage2_end**（均与**4** 字节对齐）
- 则有： **stage2_end = stage2_start + stage2_size**

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage1: 为stage2准备RAM空间

- 必须确保所安排的地址范围的确为可读写的 **RAM** 空间，即必须进行有效性测试
- **Blob**的内存有效性测试方法:
  记为**test_mempage**:
  1 以内存页为被测单位，测试每个页面头两个字是否可读写

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

test_mempage

1. 保存被测页面头两个字的内容。
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. 立即将这两个字的内容读回。应当与写入的内容一致，否则此页面地址范围不是一段有效的 **RAM** 空间
4. 再次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. 立即将这两个字的内容读回。判断依据同**3**
6. 恢复这两个字的原始内容。

Stage1: 为stage2准备RAM空间

- 测试结束后，为了得到一段干净的 **RAM** 空间范围，可以将所安排的 **RAM** 空间范围进行清零操作。

Stage1：拷贝 stage2 到 RAM 中

- 拷贝时要确定：
  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

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

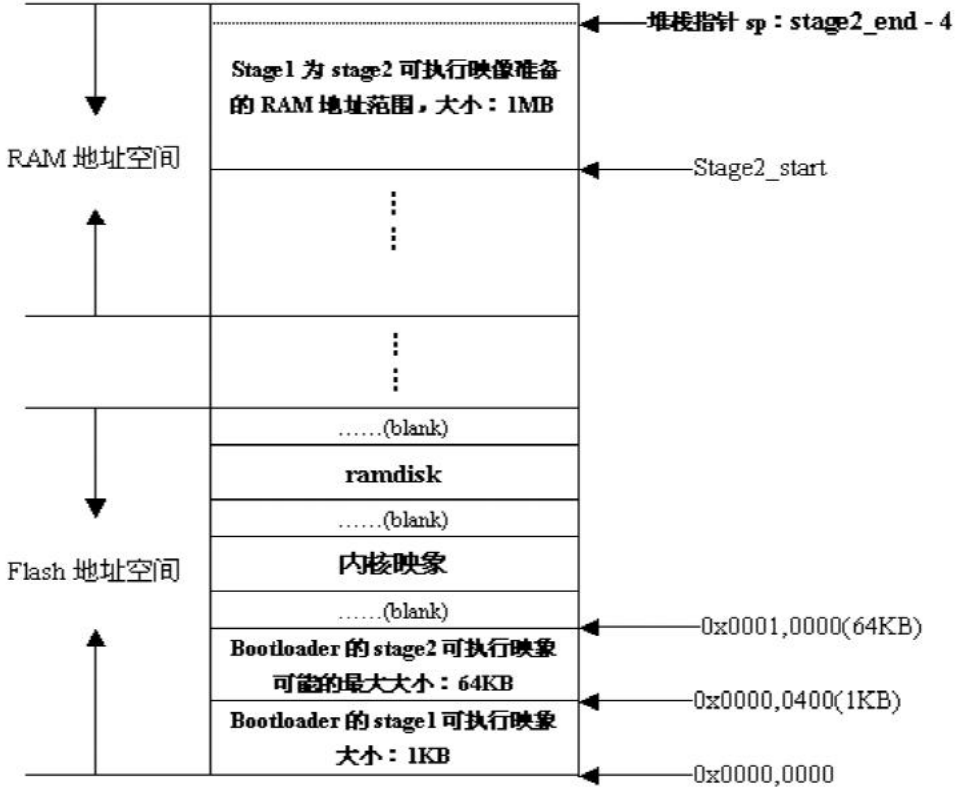
Stage1：设置堆栈指针 sp

- 对C语言编写的程序应当准备运行堆栈
- 通常设置在上述1MB RAM空间的最顶端
  - sp=(stage2_end-4)
  - 注：堆栈是向下生长的
- 此外，在设置堆栈指针前，也可关闭 led 灯，以提示用户即将跳转到 stage2。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

系统的物理内存布局

经过上述步骤后，系统的物理内存布局应该如下图所示



Stage1: 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

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

关于C入口点的疑惑

- **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，不能使用 **glibc** 库中的任何支持函数。
- 其原因是显而易见的。？？？
- 那么从哪里跳转进 **main()** 函数呢？

直接使用main函数的起始地址

- 最直接的想法就是
  直接把 **main()** 函数的起始地址作为整个 **stage2** 执行映像的入口点？
  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。

trampoline(弹簧床)的概念

- 一种更为巧妙的方法是利用 **trampoline(弹簧床)**的概念。用汇编语言写一段**trampoline** 小程序，并将它来作为 **stage2** 可执行映象的执行入口点。
  在 **trampoline**中用 **CPU** 跳转指令跳入 **main()** 函数中去执行;
  当 **main()** 函数返回时，**CPU** 执行路径显然再次回到 **trampoline** 程序。

  简而言之：用这段 **trampoline** 小程序作为 **main()** 函数的外部包裹**(external wrapper)**。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

一个简单的 trampoline 程序示例(来自 blob bootloader):

```
.text

.globl _trampoline
_trampoline:
   bl  main
   /* if main ever returns we just call it again
   */
   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。

Boot Loader的stage2

- 通常包括以下步骤
  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

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage2：初始化要用的硬件设备

- 这通常包括：
  - 初始化至少一个串口，以便和终端用户进行 I/O 输出信息；
  - 初始化计时器等。
- 在初始化这些设备之前，也可重新把 LED 灯点亮，以表明已进入 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。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage2：检测系统内存映射

- 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，
    - **SA-1100 CPU** 中，从 **0xC000,0000** 开始的 **512M** 被用作系统的 **RAM**地址空间
    - **Samsung S3C44B0X CPU** 中，从 **0x0c00,0000** 到 **0x1000,0000** 间的 **64M**被用作系统的 **RAM**地址空间

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

CPU预留的地址空间 VS. 实际使用的地址空间

- 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。
- 也即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。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

- 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。

内存映射的描述

- 如下数据结构用来描述 **RAM** 地址空间中的一段连续的地址范围：

```
type struct memory_area_struct
{
   u32 start;          //内存区域的起始地址
   u32 size;                    //内存区域的大小（字节数）
   int used;                    //内存区域的状态
} memory_area_t;
```

- **used=0|1**
  - **1**=这段地址范围已被实现，也即真正地被映射到 **RAM** 单元上
  - **0**=这段地址范围并未被系统所实现，处于未使用状态。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

内存映射的描述

- 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

```
memory_area_t memory_map[NUM_MEM_AREAS]=
{
    [0…(NUM_MEM_AREAS)]=
    {
            .start=0,
            .size=0,
            .used=0 //表示检测内存映射之前的初始状态
    },
};
```


嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

内存映射检测算法（代码）

1. 数组初始化，每个区域的**used**标志设为**0**
2. 将整个空间中所有页面的前**32**位（**4**个字节）写为**0**
3. 依次检测每个页面是否有效（使用**test_mempage**算法）
    1. 若当前页面无效
        1. 若当前区域已映射，则当前区域检测结束
    2. 若当前页面有效
        1. 判断该页面是否由其他页面映射而来，若是同**3.1**
        2. 否则若当前区域已映射，则增加有效页面到当前区域中
        3. 若当前区域为一个新的区域，则初始化该区域并增加当前页面到当前区域中

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

- 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。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage2：加载映像

- 规划内存占用的布局，包括
  - 内核映像所占用的内存范围；
  - 根文件系统所占用的内存范围。
- 主要考虑基地址和映像的大小，例如：
  - 对内核映像，一般考虑从(MEM_START + 0x8000) 开始约1MB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。

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

- Ø 对根文件系统映像，一般从 **MEM_START+0x0010,0000** 开始。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**。

v 加载映像：从 **Flash** 上拷贝

- Ø 像 **ARM** 这样的嵌入式 **CPU** 通常都在统一的内存地址空间中寻址 **Flash** 等固态存储设备
- Ø 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完成从 **Flash** 设备上拷贝映像的工作

从 Flash 上拷贝

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


嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage2：设置内核的启动参数

- 在嵌入式Linux系统中，需要由boot_loader设置的参数有：
  - 内核参数，如页面大小、根设备
  - 内存映射情况
  - 命令行参数
  - initrd映像参数
    - 起始地址，大小
  - Ramdisk参数
    - 解压后的大小

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

Stage2：调用内核

- 调用方法：
  - 直接跳转到内核的第一条指令处，也即RAM中内核被加载的地址处
- 对于ARM Linux系统，在跳转之前必须满足：
  1. CPU寄存器的设置：
     1. R0 = 0；
     2. R1 = 机器类型 ID；
     3. R2 = 传递给内核的启动参数起始地址；
  2. CPU模式：
     1. 必须禁止中断（IRQs和FIQs）；
     2. CPU必须处于SVC模式；
  3. Cache和MMU的设置：
     1. MMU必须关闭；
     2. 指令Cache可以打开也可以关闭；
     3. 数据Cache必须关闭；

嵌入式系统实验室
EMBEDDED SYSTEM LABORATORY
SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

BootLoader的工作到此为止

- 从此操作系统接管所有的工作

开源的Boot Loader

- ARMboot
- PPCBoot
- u-Boot
- Red Boot
- blob
- OpenBIOS
- FreeBIOS
- LinuxBIOS

目前，ARMboot已经和PPCBoot合并到U-Boot中

Thanks!

The end.

嵌入式系统实验室

EMBEDDED SYSTEM LABORATORY

SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC