清華大學
Tsinghua University

# 《嵌入式系统》

## 4-1 Boot Loader

# 本章提纲

- 嵌入式系统启动流程
- BootLoader概念
- BootLoader架构
- *典型BootLoader介绍

# 本章提纲

- 嵌入式系统启动流程
- BootLoader概念
- BootLoader架构
- *典型BootLoader介绍

# 嵌入式系统启动流程

- 硬件加电
- 引导加载程序
  - Boot代码、Bootloader等
- 操作系统内核，如Linux 内核
  - 根据特定的目标嵌入式硬件系统，定制的内核及启动参数
- 加载文件系统
  - 包括根文件系统以及建立于Flash内存设备上的文件系统
- 运行用户程序
  - 用户编写的完成特定功能的程序
  - 一些用户程序运行在一个嵌入式图形用户界面（GUI）上，常用的嵌入式GUI包括：MicroWindows 和MiniGUI等

# Boot Loader

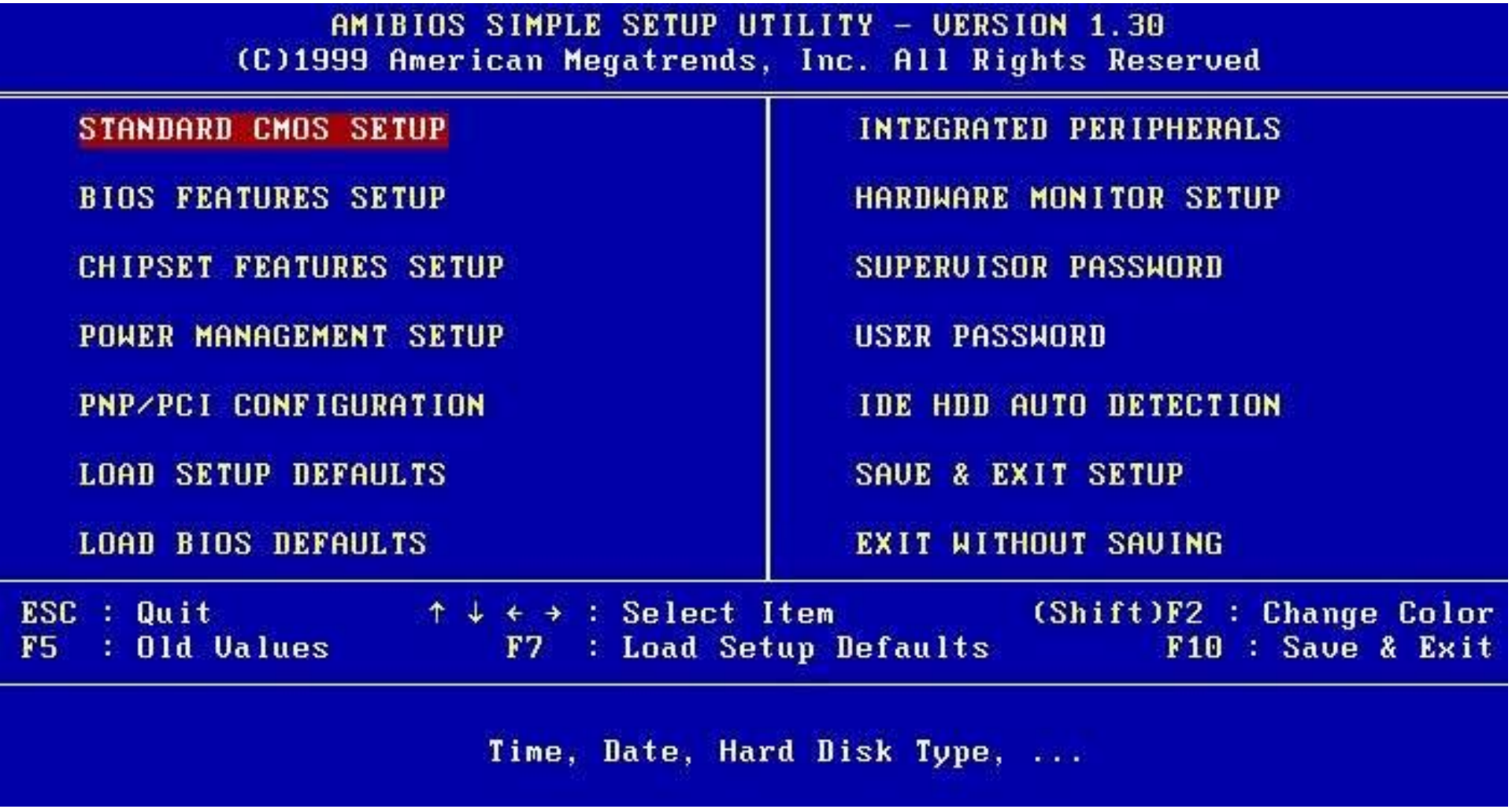
- 嵌入式系统中的 OS 启动加载程序
- 引导加载程序
  - 包括固化在固件(firmware)中的 boot 代码(可选)，和 Boot Loader两大部分
  - 是系统加电后运行的第一段软件代码
  - 对 OS 内核而言，Boot Loader发挥了部分硬件抽象层的作用。

# PC 机中的引导加载程序

- BIOS: 基本输入输出系统
- 基本功能
  - 为存储在其它介质中的软件程序做准备工作
  - 使它们能够正常地装载，执行并接管计算机的控制权
  - 这个过程被称为启动
- 两部分组成
  - BIOS (其本质就是一段固件程序)
  - 位于硬盘主引导记录(Master Boot Record)中的 OS Boot Loader，如LILO (Linux Loader) 和 GNU GRUB 等

# PC 机中的引导加载程序

- BIOS ：硬件管
  - 系统BIOS
    - 用来管理设备的基本子程序
    - 一般都装在主板的ROM中
  - 适配器上的BIOS
    - 硬盘、网卡、显卡等的BIOS都在自己的控制卡上
- OS Boot Loader：操作系统管


AMIBIOS SIMPLE SETUP UTILITY - VERSION 1.30
(C)1999 American Megatrends, Inc. All Rights Reserved
STANDARD CMOS SETUP
BIOS FEATURES SETUP
CHIPSET FEATURES SETUP
POWER MANAGEMENT SETUP
PNP/PCI CONFIGURATION
LOAD SETUP DEFAULTS
LOAD BIOS DEFAULTS
INTEGRATED PERIPHERALS
HARDWARE MONITOR SETUP
SUPERVISOR PASSWORD
USER PASSWORD
IDE HDD AUTO DETECTION
SAVE & EXIT SETUP
EXIT WITHOUT SAVING
ESC : Quit
↑ ↓ ← → : Select Item
(Shift)F2 : Change Color
F5 : Old Values
F7 : Load Setup Defaults
F10 : Save & Exit
Time, Date, Hard Disk Type, ...


CMOS Setup Utility - Copyright (C) 1984-2008 Award Software
MB Intelligent Tweaker(M.I.T.)
CPU Clock Ratio [x12]
CPU NorthBridge Freq. [x 5]
CPU Host Clock Control [Manual]
CPU Frequency(MHz) [260]
PCIE Clock(MHz) [Auto]
HT Link Frequency [600 MHz]
Set Memory Clock [Auto]
x Memory Clock x4.00 1040Mhz
EPP Mode [Disabled]
x EPP Voltage Control Normal
▶ DRAM Configuration [Press Enter]
System Voltage Control [Manual]
Normal CPU Vcore 1.3500V
DDR2 Voltage Control [+0.500V] 2.300V
NorthBridge Volt Control [+0.3V] 1.400V
CPU NB VID Control [+0.500V]
CPU Voltage Control [+0.300V]
Item Help
Menu Level ▶
Select the internal clock multiplier of the processor
FSB x Ratio = CPU operating freq
[Auto] BIOS to set the most optimized ratio.
[Other Values] Manually set the CPU clock ratio.
↑↓→←:Move Enter:Select +/-/PU/PD:Value F10:Save ESC:Exit F1:General Help
F5:Previous Values F6:Fail-Safe Defaults F7:Optinized Defaults


Phoenix - AwardBIOS CMOS Setup Utility
AwardBIOS设置主界面
technic & fun
SoftMenu Setup 软超频设置选项
Standard CMOS Features 标准COMS选项
Advanced BIOS Features 高级BIOS功能
Advanced Chipset Features 高级芯片组设置
Integrated Peripherals 集成设备管理
Power Management Setup 电源管理
PnP/PCI Configurations PnP/PCI配置
PC Health Status PC健康状态
Load Fail-Safe Defaults
加载默认设置
Load Optimized Defaults
加载最佳默认设置
Set Password
密码设置
Save & Exit Setup
保存并退出
Exit Without Saving
不保存退出
Change CPU's Clock & Voltage

# PC 机中的引导加载程序

- 流程
  - 系统加电启动时，系统BIOS负责检测并启动加载控制卡上的BIOS
  - BIOS 在完成硬件检测和资源分配后，将硬盘 MBR（Master Boot Record，主引导记录）中的 Boot Loader 读到系统的 RAM 中，然后将控制权交给 OS Boot Loader
  - Boot Loader 的主要任务是将内核映象从硬盘读到 RAM 中，然后跳转到内核的入口点去运行，即开始启动操作系统。
- BIOS常驻系统内存的高端(C0000-FFFFF)

# 嵌入式系统中引导加载程序

- 没 BIOS 那样的固件程序
  - 有的嵌入式 CPU 也会内嵌一段短小的启动程序
- 系统的加载启动任务就完全由 Boot Loader 来完成
  - 如ARM7TDMI中，系统在上电或复位时从地址 0x00000000 处开始执行
  - 这个地址是Boot Loader 程序

# 本章提纲

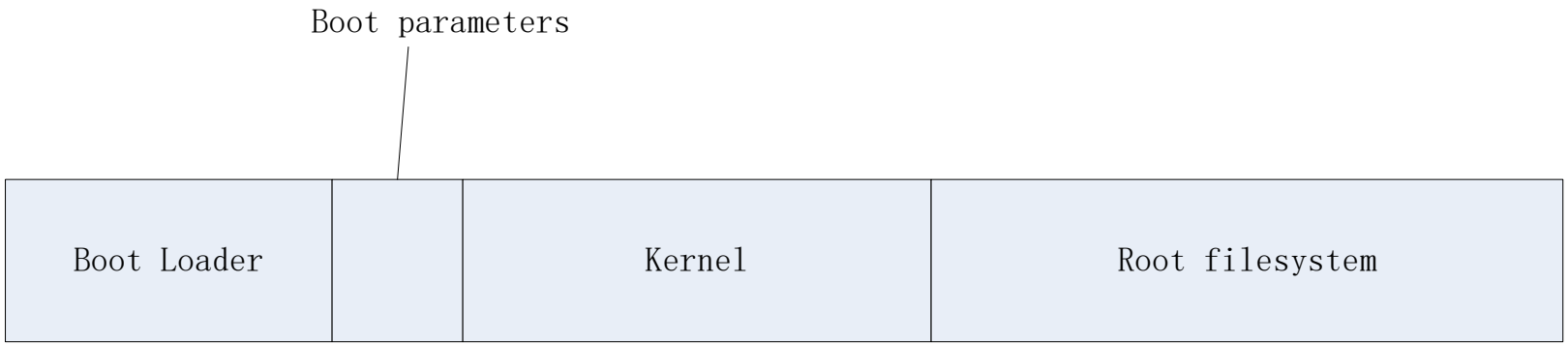
- 嵌入式系统启动流程
- BootLoader概念
- BootLoader架构
- *典型BootLoader介绍

# Boot Loader 概念

- 在操作系统内核运行之前运行的一段小程序
- 功能
  - 初始化硬件设备
  - 建立内存空间的映射图
  - 调整系统的软硬件环境，以便操作系统内核启动
- 一般不通用
  - 依赖于处理器架构
    - CPU体系结构：ARM、MIPS、DSP、x86 etc
  - 依赖于具体的板级配置
    - 板级设备的配置：不同厂家的芯片、不同的内存空间
  - 不同的 CPU有不同的Boot Loader
  - 有一些 Boot Loader 也能支持多种CPU: U-Boot 就同时支持 ARM 体系结构和MIPS 体系结构

# Boot Loader 存储

- 嵌入式系统没有BIOS，系统上电或复位后从0x00000000开始执行
- 嵌入式系统通常有固态存储设备(比如：ROM、EEPROM 或 FLASH 等)被映射到0地址上，Boot Loader存储在0地址
- 系统加电后，CPU 将首先执行 Boot Loader 程序
- 下面是固态存储设备的典型空间分配结构图
  - Boot Loader，内核启动参数，内核映像，根文件系统映像。



# Boot Loader 的输入输出

- 可用来控制 Boot Loader 的设备或机制
- 调试方法
  - 在Boot loader阶段，显示设备不可用
  - 因此字符方式与用户进行交互
  - 主机和目标机之间一般通过串口建立连接
  - 通常需要与Host主机相连，Host作为TTY(Teletype)终端
  - 输出打印信息到串口，从串口读取用户控制命令

# Boot Loader 操作模式 (1/3)

- Boot Loader有两种操作模式：启动加载模式、下载模式
- 启动加载模式(Boot Loading)
  - 自主(Autonomous)模式: Boot Loader 的正常工作模式
- 启动加载模式的一般流程
  - 从目标机某个固态存储设备上将OS加载到 RAM
  - 准备好内核运行所需的环境和参数
  - 在RAM运行操作系统内核

# Boot Loader 操作模式（2/3）

- 下载模式(Downloading)
  - 用户干预进入下载模式，在控制台打印提示信息，等待用户输入
    - 如用户不干预，则进入正常启动模式，即调用操作系统内核
  - 可通过串口连接或网络连接等通信手段从主机（Host）下载文件
    - 可以下载内核映像、根文件系统映像、Boot loader自身
  - 通常在第一次安装内核与根文件系统时被使用
  - 系统更新也会使用 Boot Loader 的这种工作模式
  - 流程
    - 从主机下载的文件首先被 Boot Loader 保存到目标机的 RAM 中
    - 被 Boot Loader 写到目标机上的固态存储设备中，或者直接在RAM中运行

# Boot Loader 操作模式（3/3）

- 通用boot loader一般同时支持两种工作模式
  - 如Blob (Boot Loader Object) 或 U-Boot
  - 允许用户在这两种工作模式之间进行切换

- Blob（面向ARM Linux）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10秒内没有用户按键，则 blob 继续启动 Linux 内核

# Boot Loader 的文件传输设备及协议

- 控制信息传输
  - 目标机Boot Loader 与主机之间：串口
  - 传输协议：xmodem/ymodem/zmodem 协议 （传输效率递增）
  - 简单、通用，易于设置；速度慢
- 大型文件，如image
  - 串口传输的速度是有限的
  - 以太网，TFTP协议
    - 主机提供 TFTP 服务
    - 通过以太网连接并借助 TFTP 协议来下载文件
    - 通用，易用，速度快；编程略复杂
  - USB
    - 简单、易用、速度快、适于下载内核时采用

# 本章提纲

- 嵌入式系统启动流程
- BootLoader概念
- BootLoader架构
- *典型BootLoader介绍

# Boot Loader 的基本架构

- Boot Loader 的启动过程
  - 单阶段（Single-Stage）或多阶段（Multi-Stage）
- 多阶段的 Boot Loader
  - 提供更为复杂的功能，以及更好的可移植性
- Boot Loader的生命周期
  1. 初始化硬件,如设置UART(至少设置一个),检测存储器等
  2. 设置启动参数,告诉内核硬件的信息,如用哪个启动界面,波特率.
  3. 跳转到操作系统的首地址.
  4. 消亡
- 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
  - 启动过程可以分为 stage 1 和 stage 2 两部分

# 两阶段 Boot Loader

- stage 1
  - 汇编, 短小精悍
  - 简单的硬件初始化
- stage 2
  - C语言，便于移植，功能比Stage 1复杂
  - 复制数据
  - 设置启动参数
  - 串口通信等功能

# Boot Loader 的主要任务

- 假定
  - 内核映像与根文件系统映像都被加载到 RAM 中运行（也存在不用加载，就地执行的技术: XIP, 直接在 ROM 或 Flash 这样的固态存储设备中直接运行 ）
- Stage 1 直接运行在固态存储上，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 入口点

# Boot Loader 的主要任务

- Boot Loader 的 stage 2 通常包括以下步骤
  - 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)
  - 将内核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

# Stage 1，基本的硬件初始化

- 目的
  - 为 stage 2 的执行以及随后的 kernel 的执行准备好一些基本的硬件环境
- 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，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（如 ARM 的 CPSR 寄存器）来完成
- 设置 CPU 的速度和时钟频率。

# Stage 1，基本的硬件初始化

- 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。
- 初始化 LED
  - 通过 GPIO (General Purpose Input Output 通用输入/输出) 来驱动 LED，其目的是表明系统的状是 OK 还是 Error
  - 如板子上没有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
- 关闭 CPU 内部指令 / 数据 cache
  - 避免CPU执行不必要的指令或者访问不存在的cache数据，带来不确定性。

# 为加载 stage 2 准备 RAM 空间

- 通常把 stage 2 加载到 RAM 空间中来执行（更快）
- Stage 2 通常是 C 语言执行代码，所以需要同时考虑为其安排堆栈空间
- 准备的RAM空间大小最好是 memory page 大小(通常是 4KB)的倍数
- 一般1M RAM 空间已经足够，地址范围可以任意安排
  - 如 blob 就将 stage 2 可执行映像从系统 RAM 起始地址 0xc0200000 开始的 1M 空间内执行
  - 将 stage 2 安排到 RAM 空间的最顶 1MB也是一种值得推荐的方法
  - Stage 2_end = stage 2_start + stage 2_size

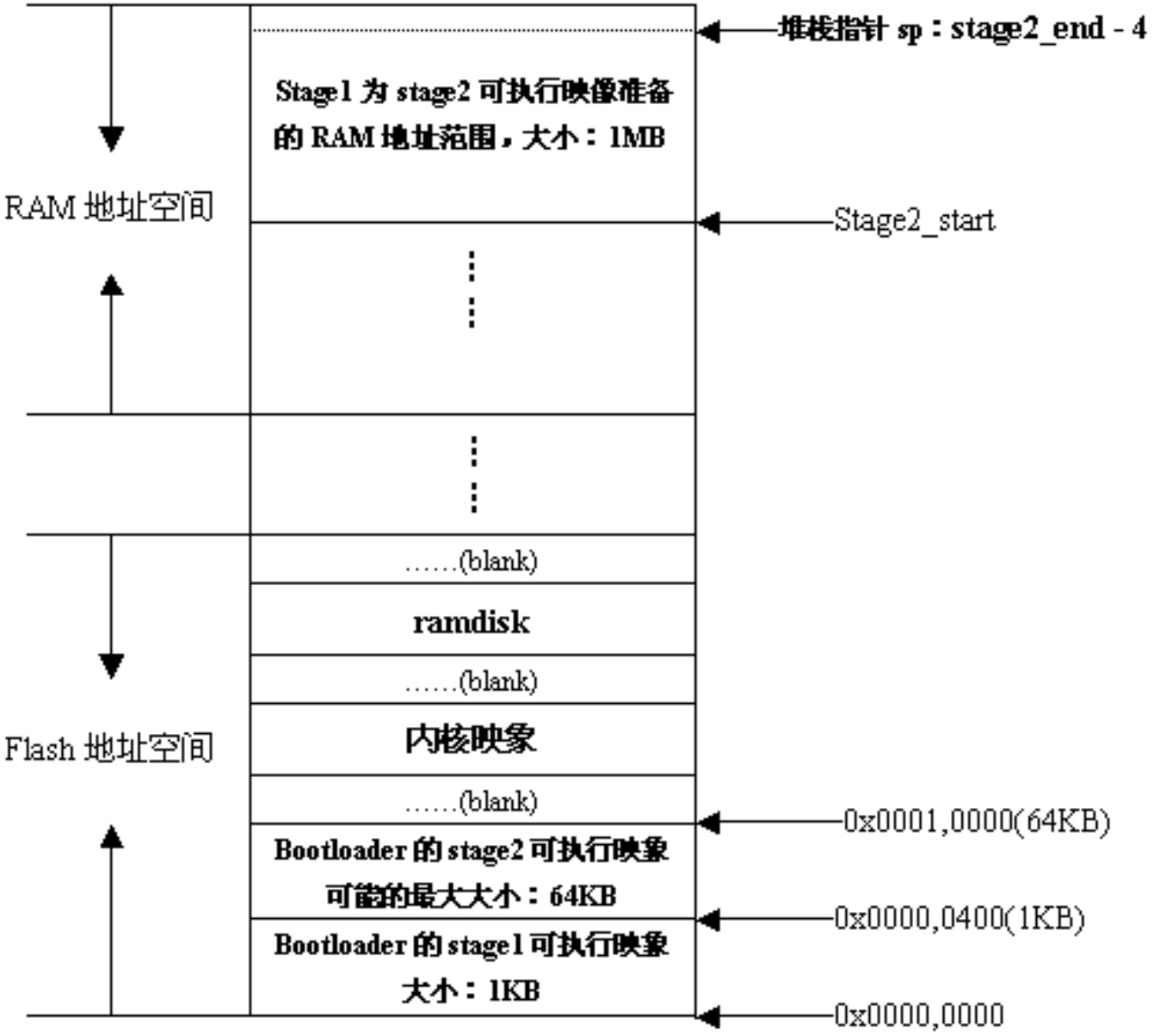
# 为加载 stage 2 准备 RAM 空间

- 对所安排的地址范围进行测试（为什么要测试？）
  - 必须确保所安排的地址范围对应可读写的 RAM 空间
  - 测试方法（blob 的方法）
    - 以 **memory page 为被测试单位**，测试**每个 page** 开始的**两个字**是否是可读写的：具体而言，就是先往每个page的前两个字里各写一个特别的数，然后再读出来，如果写与读的一致，则认为可用。否则不可用。

# 拷贝 stage 2 到 RAM 中

- 拷贝时要确定两点
  - stage 2 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（由此知道哪一段是可执行的）
  - RAM 空间的起始地址（由此知道其他可用空间的范围）

# Stage 2可执行映像刚被复制到RAM空间时的系统内存布局



# 设置堆栈指针sp

- 通常把 sp 的值设置为(stage 2_end-4)
  - 1MB 的 RAM 空间的最顶端(**堆栈向下生长**)

- 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 stage 2

# 跳转到 stage 2 的 C 入口点

- 可以跳转到Boot Loader的 stage 2 去执行

- 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

# Stage 2

- Stage 2 的代码通常用 C 语言来实现：代码可读性和可移植性
- 不能使用 glibc 库中的任何支持函数
  - Q: Why ?

# Stage 2

- Stage 2 的代码通常用 C 语言来实现：代码可读性和可移植性
- 不能使用 glibc 库中的任何支持函数
  - OS内核尚未运行，glibc会调用的系统调用接口尚未就绪
- Q: 怎么进入main( )函数？

# Stage 2

- Stage 2 的代码通常用 C 语言来实现：代码可读性和可移植性
- 不能使用 glibc 库中的任何支持函数
  - OS内核尚未运行，glibc会调用的系统调用接口尚未就绪
- Q: 怎么进入main( )函数？
  ——————将main( )函数的起始地址作为stage 2的入口点，是否是最合理的方法？

# Stage 2

- Stage 2 的代码通常用 C 语言来实现：代码可读性和可移植性
- 不能使用 glibc 库中的任何支持函数
  - OS内核尚未运行，glibc会调用的系统调用接口尚未就绪
- Q: 怎么进入main( )函数？
  - 将main( )函数的起始地址作为stage 2的入口点是最直接的想法，但很少采用，因为：
  - 1)无法传递函数参数；2)无法处理函数返回

# Stage 2

- 更为巧妙的方式：trampoline(弹簧床)
  - 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
  - 在 trampoline 汇编小程序中用 CPU 跳转指令跳入 main() 函数中去执行（为什么这样做？）

# Stage 2

- 更为巧妙的方式：trampoline(弹簧床)
  - 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
  - 在 trampoline 汇编小程序中用 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我们的 trampoline 程序
  - 核心思想：用 上述trampoline 程序来作为 main() 函数的外部包裹(external wrapper)

# blob 的 trampoline 程序示例

- .text
- .globl _trampoline
- _trampoline:
- bl main
- /* 上面一行的作用是什么？*/
- /* if main ever returns we just call it again */
- b _trampoline

# Stage 2：初始化本阶段要使用的硬件设备

- 初始化至少一个串口，以便终端用户进行 I/O 输出信息
  - 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
- 初始化计时器等

# 检测系统的内存映射（memory map）

- 在 4GB 物理地址空间中哪些地址范围被分配用来寻址系统的 RAM 单元
  - 如 SA-1100 中，从 0xC0000000 开始的512M空间被用作系统的 RAM 空间
  - 在Samsung S3C44B0X 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
- 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
- Boot Loader 的 stage 2 必须检测整个系统的内存映射情况
  - 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

- 可以用如下数据结构来描述 RAM 地址空间中的一段连续的地址范围：

```
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：

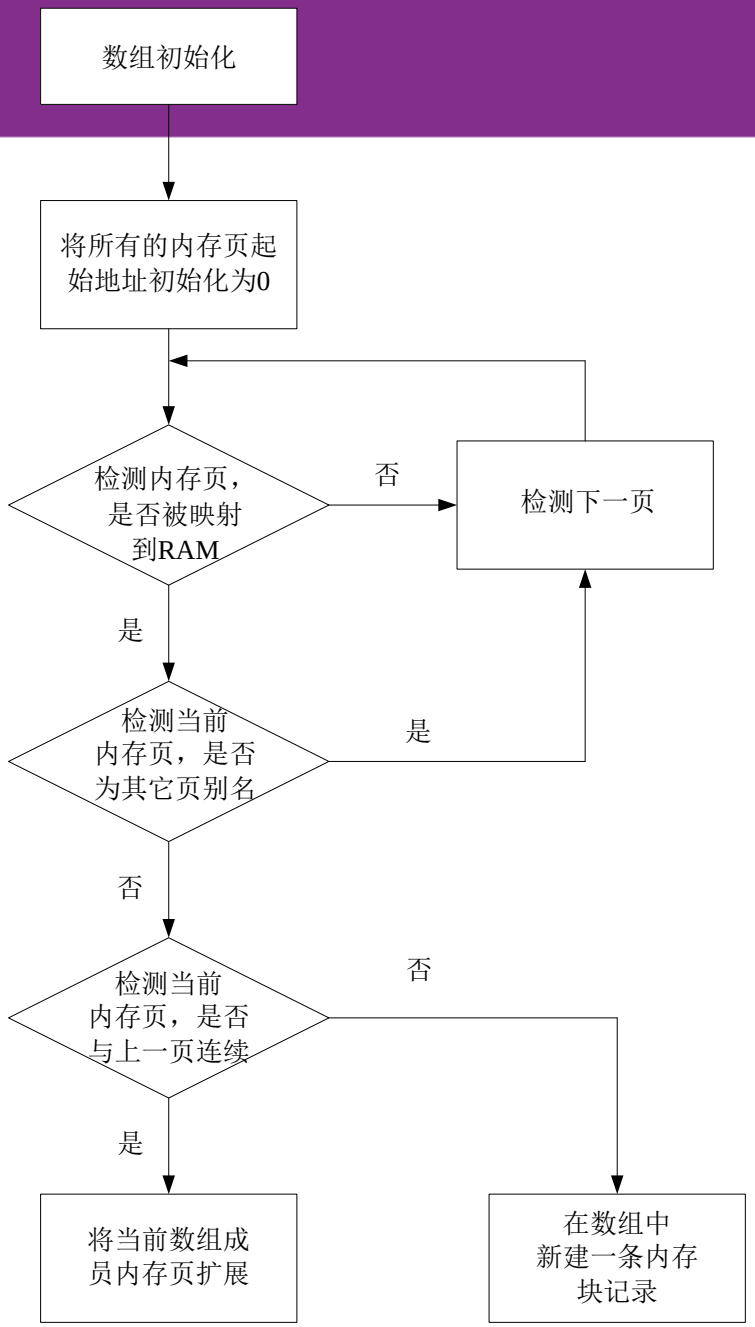
  (1)used=1，则说明这段连续的地址范围已被实现，也即真正地被映射到 RAM 单元上。

  (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。

# 检测流程

检测流程输出的是整个地址空间的映射信息，反映整个地址空间中哪些地址范围被分配用来寻址系统的RAM 单元。



# 加载内核映像和根文件系统映像

- 规划内存占用的布局（以ARM为例）
  - 内核映像所占用的内存范围
    - 一般将其复制到从MEM_START+0x8000这个基地址开始的大约1MB大小的内存范围内(内核大小一般不超过1MB)
    - MEM_START到MEM_START+0x8000这段32KB大小的内存一般放置一些全局数据结构，如启动参数和内核页表等。
  - 根文件系统所占用的内存范围
    - 一般将其复制到从MEM_START+0x00100000这个基地址开始的大约1MB大小的内存范围。
- 从 Flash 上拷贝内核映像到 RAM 上

# 设置内核的启动参数

- 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
- 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

# 启动参数中的可用TAG

| Tag name | Value | Size | Description |
|---|---|---|---|
| ATAG_NONE | 0x00000000 | 2 | Empty tag used to end list |
| ATAG_CORE | 0x54410001 | 5 (2 if empty) | First tag used to start list |
| ATAG_MEM | 0x54410002 | 4 | Describes a physical area of memory |
| ATAG_VIDEOTEXT | 0x54410003 | 5 | Describes a VGA text display |
| ATAG_RAMDISK | 0x54410004 | 5 | Describes how the ramdisk will be used in kernel |
| ATAG_INITRD2 | 0x54420005 | 4 | Describes where the compressed ramdisk image is placed in memory |
| ATAG_SERIAL | 0x54410006 | 4 | 64 bit board serial number |
| ATAG_REVISION | 0x54410007 | 3 | 32 bit board revision number |
| ATAG_VIDEOLFB | 0x54410008 | 8 | Initial values for vesafb-type framebuffers |
| ATAG_CMDLINE | 0x54410009 | 2 + ((length_of_cmdline + 3) / 4) | Command line to pass to kernel |

http://blog.chinaunix.net/uid-20321537-id-1966888.html

# 调用内核

- 直接跳转到内核的第一条指令处
- 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；@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（管理模式，系统复位时进入）；
  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
- theKernel()函数永远不返回，如果这个调用返回，则说明出错了

# 串口终端的调试作用

- 调试手段: 打印信息到串口终端

- 串口终端显示乱码或根本没有显示
  - boot loader 对串口的初始化设置不正确
  - 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

# 比特率与波特率

- 比特率在数字信道中，比特率是数字信号的传输速率，它用单位时间内传输的二进制代码的有效位(bit)数来表示。
- 波特率指数据信号对载波的调制速率，它用单位时间内载波调制状态改变次数来表示，其单位为波特(Baud)。
- 比特率=波特率X单个调制状态对应的二进制位数。
- 两相调制(单个调制状态对应1个二进制位)的比特率等于波特率；四相调制(单个调制状态对应2个二进制位)的比特率为波特率的两倍；八相调制(单个调制状态对应3个二进制位)的比特率为波特率的三倍；依次类推。

# 串口终端的调试作用（续）

- Bootloader的运行过程中可以正确向串口输出信息，但bootloader启动内核后无法看到内核启动输出信息
  - 确认是否在编译内核时配置了对串口的支持
  - 确认bootloader对串口的设置与内核对串口的设置一样
  - 确认bootloader所用的内核基地址与内核映像在编译时所用的运行基地址一致

# 本章提纲

- 嵌入式系统启动流程
- BootLoader概念
- BootLoader架构
- *典型BootLoader介绍

# WinCE 专用 Boot Loader

- 2009年2月16日Windows Mobile 6.5
  - 内核windows CE 5.x
  - 以后不用Windows Mobile ，用Windows phones

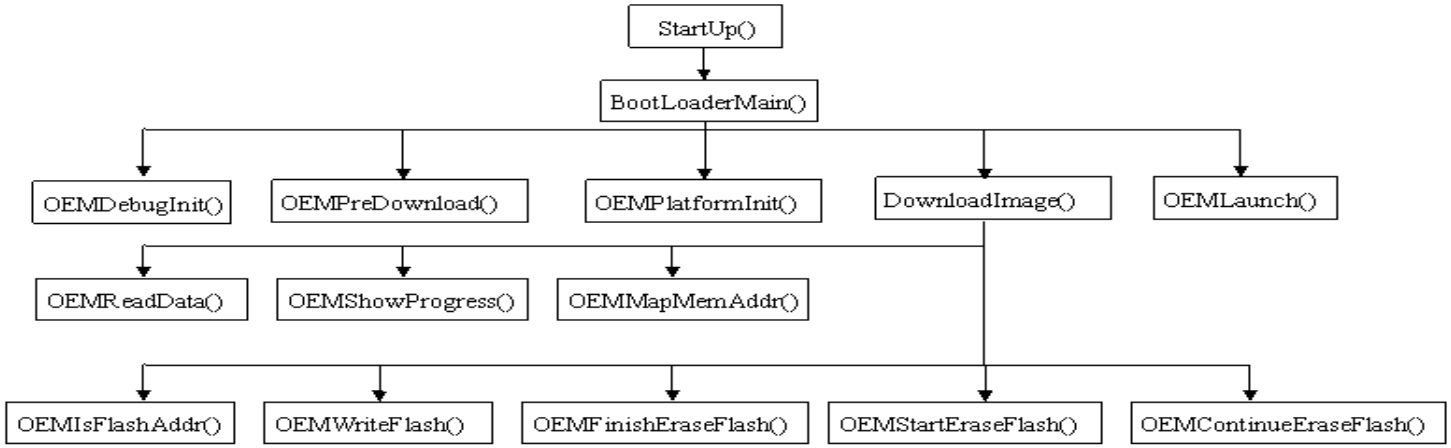
# WinCE 专用 Boot Loader

- X86的ROM Boot Loader
  - 又叫Rom Boot
  - 存放在Flash/EEPROM中，也就是原来BIOS的位置
  - 上电后CPU到固定地址执行代码，也就是执行了Rom Boot包含的代码
  - 对整个硬件系统进行初始化和检测
  - 支持通过网卡从远程机器上下载nk.bin或者从本地IDE/ATA硬盘的活动分区中寻找nk.bin文件加载
  - 优点：引导并且加载速度快，而且它自身完成了所有的操作，这样就不用BIOS、MSDOS，更不用Loadcepc了
  - 缺点：需要CE开发者读懂它的源码并修改
  - CE提供了Rom Boot的所有源码

- X86的BIOS Boot Loader
  - BIOS Boot Loader只是不需要MSDOS操作系统，它仍然需要BIOS和FAT文件系统。
  - 系统上电后BIOS执行完硬件初始化和配置后，BIOS检查引导设备的启动顺序，如果引导设备是硬盘、CF卡、DOC（Disk-On-Chip）一类的存储设备，那么就加载这些存储器上的主引导扇区（Master Boot Sector）中的实模式代码到内存，然后执行这些代码。

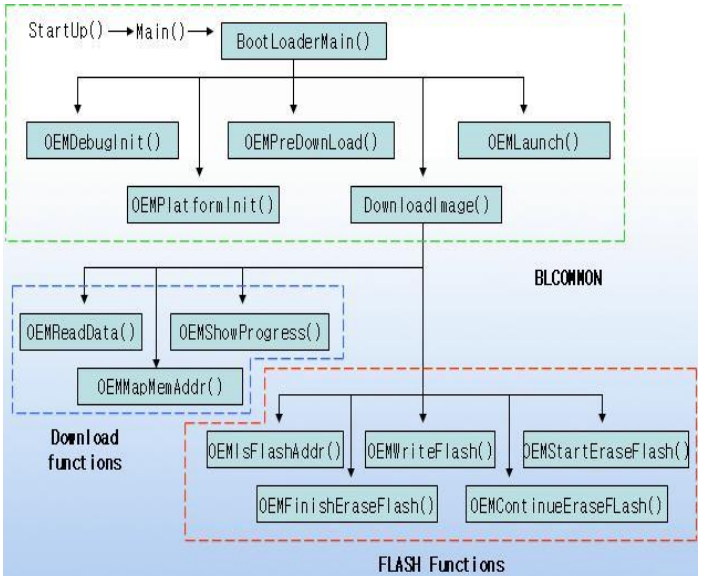
- X86的BIOS Boot Loader
  - 代码被称为主引导记录（MBR）
  - MBR首先在分区表（同样位于主引导扇区）中寻找活动分区，如果存在活动分区，那么加载位于这个活动分区的第一个扇区上的代码到内存，然后执行这些代码。这里提到的活动分区的第一个扇区被称为引导扇区（Boot Sector）。
  - 引导扇区上的代码的功能是找到并且加载BIOS Boot Loader，BIOS Boot Loader再加载nk.bin。
  - 对于BIOS Boot Loader，CE提供了Setupdisk.144和Bootdisk.144两个文件。

- MSDOS + Loadcepc
  - 非常简单
  - BIOS Boot Loader和MSDOS + Loadcepc两种方式差不多
  - 在MSDOS启动后再执行loadcepc.exe，让loadcepc加载nk.bin到内存后再把CPU控制权交给CE内核程序



- 非X86的EBOOT
  - Ethernet Boot loader，基于网络的Bootloader
  - Platform Builder集成开发环境提供了对Eboot开发的支持库
  - 可以加载WinCE image，还可以通过网络配合Platform Builder下载WinCE image进行调试
  - 组成
    - BLCOMMON、OEM 代码、Eboot、存储管理和网络驱动

- Eboot流程
- 初始化MCU
  - 包括初始化MCU的相关寄存器、中断、看门狗、系统时钟、内存和MMU
- 调用BootloaderMain
  - 依次调用以下几个函数，OEMDebugInit、OEMPlatformInit、OEMPreDownload、OEMLaunch
  - 这些函数必须由EBOOT的代码来实现。
- 最终跳转到OAL.exe的StartUp处
  - 启动WinCE操作系统



# Boot Strap

- 一段固化在处理器内部ROM之中的一段代码
  - 用户可以通过片外的管脚配置是否启动Boot Strap
  - 如果用户选择启动Boot Strap
    - 在系统上电复位时，固化在片上ROM中的代码将获得控制权
    - 这段代码将初始化片内的一些硬件设备，比如外部存储器接口控制器，通信接口等等
    - 完成这些初始化后，Boot Strap将通过这个通信接口等待调试主机发送来的命令，这些命令包括从调试主机下载一段映象到外部存储器的某个特定地址，或者是将外部存储器的内容通过通信接口上传到调试主机，从外部存储器的某个地址开始运行等等
- 某些厂商推出的嵌入式微处理器中（如飞思卡尔公司推出的龙珠328系列、新龙珠系列）提供了Boot Strap功能

- Boot Strap模式
  - Bootloader在Bootstrap模式下被加载到Flash上
- Boot strap模式
  - 容许开发者通过UART对系统进行初始化
  - 容许开发者通过UART将程序下载到系统RAM中
  - Bootstrap可以接收命令，并且运行事先储存在系统内存上的程序
  - Bootstrap 支持对内存和寄存器的读写操作

- Boot Strap 操作流程
  - 将目标板上的Boot Strap pin 打开
  - 上电后，Bootstrap模式将UART1和UART2设置为自动波特率模式
  - 同时将UART1和UART2设为8bit长、无校验位、1个停止位
  - 等待用户输入a或A，一旦接收到上述字符，则可以设定波特率并返回冒号
  - 通过Bootstrap record语句将CPU芯片内部寄存器，初始化为目标寄存器

# 通用 Boot Loader：U-boot

- 德国DENX软件工程中心的Wolfgang Denk
- 全称Universal Boot Loader
  - 遵循GPL条款的开放源码项目
  - http://sourceforge.net/projects/U-Boot
  - 从FADSROM、8xxROM、PPCBOOT逐步发展演化而来
  - 其源码目录、编译形式与Linux内核很相似
  - 源码就是相应Linux内核源程序的简化，尤其是设备驱动程序
- 支持
  - 嵌入式Linux/NetBSD/VxWorks/QNX/RTEMS/ARTOS/LynxOS
  - PowerPC、MIPS、x86、ARM 、Nios、XScale等处理器
- 对PowerPC系列处理器/对Linux的支持最完善

# 功能

| | |
|---|---|
| 系统引导 | 支持NFS挂载、RAMDISK 系统引导 (压缩或非压缩)形式的根文件系统 |
| | 支持NFS挂载，从Flash中引导压缩或非压缩系统内核 |
| 基本辅助 | 强大的操作系统接口功能，可灵活设置、传递多个关键参数给操作系统， 适合系统在不同开发阶段的调试要求与产品发布，尤其对Linux支持最为功能强劲 |
| | 支持目标板环境参数的多种存储方式，如Flash、NVRAM、EEPROM |
| | CRC32校验，可校验Flash中内核、RAMDISK镜像文件是否完好 |
| 设备驱动 | 串口、SDRAM、Flash、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI、RTC等驱动支持 |
| 上电自检功能 | SDRAM、Flash大小自动检测；SDRAM 故障检测；CPU型号 |
| 特殊功能 | XIP内核引导 |

# 移植方法

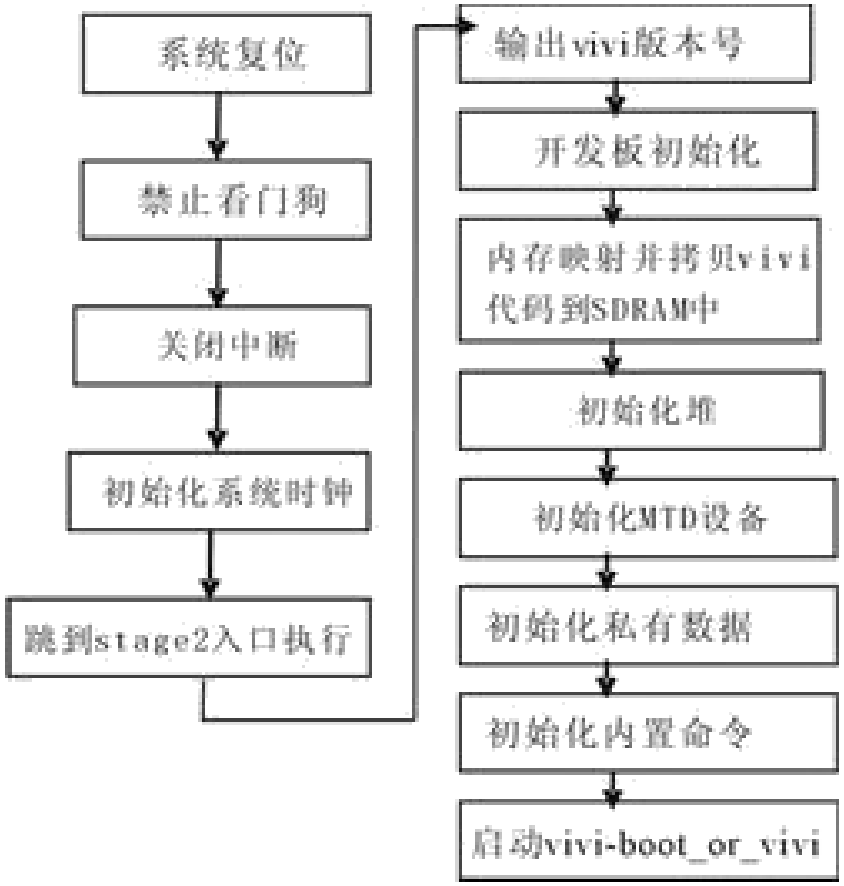
- 先用硬件调试器 BDI2000创建目标板初始运行环境，将U-Boot镜像文件u-boot．bin下载到目标板RAM中的指定位置，然后，用BDI2000进行跟踪调试
  - 好处:不用将Uboot镜像文件烧写到Flash中去
  - 弊端:对移植开发人员的移植调试技能要求较高，BDI2000的配置文件较为复杂
- 用BDI2000先将U—Boot镜像文件烧写到Flash中去，然后用GDB和BDI2000调试
  - 所用的BDI2000配置文件较为简单，调试过程与U-Boot移植后运行过程相吻合
  - U—Boot先从Flash中运行，再重载至RAM 中相应位置，并从那里正式投入运行
  - 需要不断烧写Flash

# 通用 Boot Loader：Redboot

- Redhat公司随eCos发布的一个BOOT方案
- 开源项目
- 支持
  - ARM，MIPS，MN10300，PowerPC， Renesas SHx，v850，x86
- 使用X-modem或Y-modem协议经由串口下载，也可以经由以太网口通过BOOTP/DHCP服务获得IP参数，使用TFTP方式下载程序映像文件，常用于调试支持和系统初始化（Flash下载更新和网络启动）
- Redboot可以通过串口和以太网口与GDB进行通信，调试应用程序，甚至能中断被GDB运行的应用程序
- Redboot为管理FLASH映像，映像下载，Redboot配置以及其他如串口、以太网口提供了一个交互式命令行接口，自动启动后，REDBOOT用来从TFTP服务器或者从Flash下载映像文件加载系统的引导脚本文件保存在Flash上

# 通用 Boot Loader：vivi

- vivi是由韩国Mizi公司开发的一种Bootloader
- 适合于ARM9处理器
- 源代码可以在 http://www.mizi.com网站下载




Thank you!