

嵌入式操作系统

5 BootLoader和bootloader举例

陈香兰 (xlanchen@ustc.edu.cn)

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

December 4, 2014

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

Outline

1 BootLoader简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader的主要任务和典型结构框架
- 部分开源的Boot Loader

2 u-boot

3 RedBoot

4 小结和作业

- 本节从以下四个方面来讨论嵌入式系统的 Boot Loader，包括：

- ① Boot Loader 的概念
- ② Boot Loader 的安装
- ③ Boot Loader 的启动过程
- ④ Boot Loader 的主要任务和典型框架结构

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
 - 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。

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 将内核映像从硬盘上读到 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

- 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。

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

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。

- ① 初始化硬件设备

- ② 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。

- ③ 加载操作系统内核映像到RAM中，并将系统的控制权传递给它

- ★ 例如：Linux

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。

支持的 CPU 和嵌入式板

Boot Loader依赖于

① CPU 的体系结构

- ① 不同的CPU体系结构都有不同的Boot Loader
- ② 有些 Boot Loader 也支持多种CPU体系结构
 - ★ 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

- ▶ 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

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

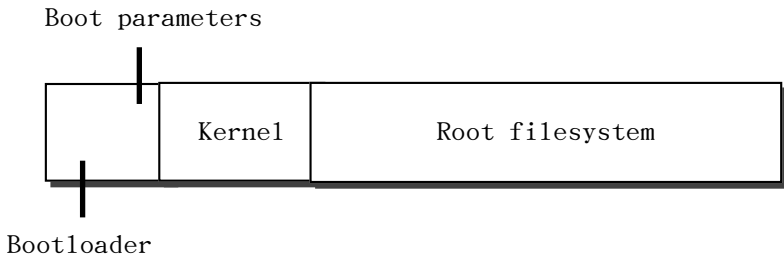
4 小结和作业

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。

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



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

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 在执行时也常通过串口来进行 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

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- **Boot Loader 的启动过程和操作模式**
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

Boot Loader 的启动过程

Boot Loader的启动过程可以是

❶ 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

❷ 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
 - ① 启动加载 (Boot loading) 模式和
 - ② 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

Boot Loader 的操作模式

④ 启动加载 (Boot loading) 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

Boot Loader 的操作模式

❷ 下载 (Downloading) 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

- 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。

Boot Loader 的操作模式

- 与boot loader两种模式相关的问题

- ▶ μ Clinux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像，一般是内核和文件系统的复合体
- ▶ 常见有
 - ① image.ram (常称为ram版内核) 和
 - ② 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

Boot Loader 的操作模式

④ 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，然后用boot loader跳转过去就把 μ clinux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

Boot Loader 的操作模式

② 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像解压释放到SDRAM指定地址，接着自动引导内核，启动 μ Clinux
- ▶ 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

- 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

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

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。

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

① 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

② 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

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

- ① 硬件设备初始化
- ② 为加载 Boot Loader的stage2准备RAM空间拷贝
- ③ 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

Stage1：硬件初始化 I

- 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

❶ 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

❷ 设置 CPU 的速度和时钟频率

Stage1：硬件初始化 II

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 I

- 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 足够
- 具体的地址范围可以任意安排
 - ▶ 比如，blob 将它的 stage2 可执行映像安排到系统的 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

Stage1 : 为 stage2 准备 RAM 空间 II

- 假设

空间大小 : `stage2_size` (字节)

起止地址分别为 : `stage2_start` 和 `stage2_end` (均与4字节对齐)

则有 :

$$\text{stage2_end} = \text{stage2_start} + \text{stage2_size}$$

- 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` :

- ▶ 以内存页为被测单位，测试每个页面头两个字是否可读写

Stage1 : 为stage2 准备 RAM 空间 III

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1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1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。

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 空间范围清零。

Stage1 : 拷贝 stage2 到 RAM 中

- 拷贝时要确定：

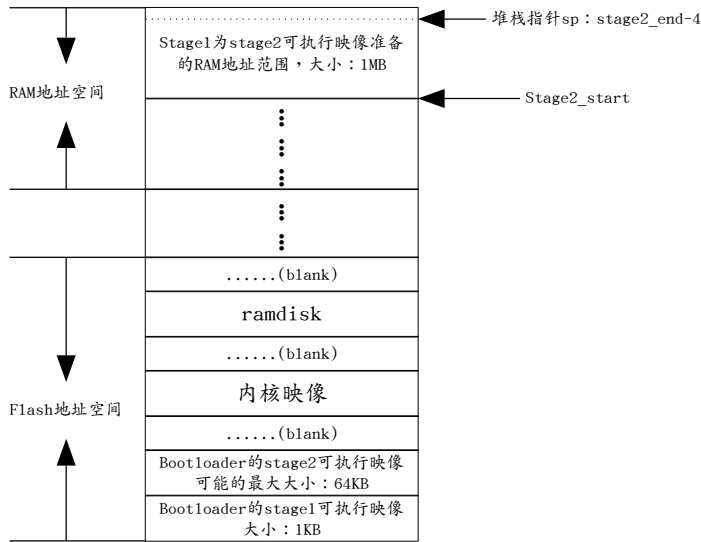
- ▶ Stage2的可执行映象在固态存储设备的存放起始地址和终止地址
- ▶ RAM 空间的起始地址

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。

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

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：
 - ❶ 无法通过`main()` 函数传递函数参数；
 - ❷ 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
 - ① 用 CPU 跳转指令跳入 main() 函数中去执行；
 - ② 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)。

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。

trampoline(弹簧床)的概念

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

- 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

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。

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

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。

Stage2：检测系统内存映射

- 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。

Stage2：检测系统内存映射

● 内存映射的描述

- ▶ 如下数据结构用来描述 RAM 地址空间中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。

Stage2：检测系统内存映射

- 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
    },
};
```

Stage2：检测系统内存映射

- 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

- 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。

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($\text{MEM_START} + 0\text{x}8000$) 开始约1MB的内存范围内
 - ★ 嵌入式 Linux 的内核一般都不超过 1MB。
 - ★ 为什么要把从 MEM_START 到 $\text{MEM_START} + 0\text{x}8000$ 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。
- ▶ 对根文件系统映像，一般从 $\text{MEM_START} + 0\text{x}0010,0000$ 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。

Stage2：加载映像

● 加载映像：从 Flash 上拷贝

- ▶ 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
- ▶ 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 */  
};
```

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

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；

BootLoader的工作到此为止

- 从此操作系统接管所有的工作

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

开源的Boot Loader

- ARMboot
- PPCBoot
- u-Boot

ARMboot已经和PPCBoot合并到U-Boot中

- Red Boot
- blob
- OpenBIOS
- FreeBIOS
- LinuxBIOS

Outline

1 BootLoader简介

2 u-boot

- u-boot简介
- 编译u-boot
- 简单分析u-boot源码

3 RedBoot

4 小结和作业

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

u-boot (Universal Boot)

- uboot是在ppcboot以及armboot的基础上发展而来的
- 支持很多处理器，比如PowerPC、ARM、MIPS、x86

最新的主页

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

u-boot的使用手册，参见

The DENX U-Boot and Linux Guide (DULG) for canyonlands

- 1 安装交叉开发环境ELDK：Embedded Linux Development Kit
 - 2 通过串口/网络连接到目标端
 - 3 配置、编译并安装u-boot
 - 4 配置、编译并安装Linux
- 该手册使用SELF：
Simple Embedded Linux Framework

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

编译u-boot I

- 下载源码u-boot-1.3.0.tar.bz2
- 解压缩

```
tar -jvxf u-boot-1.3.0.tar.bz2
```

- 阅读源代码根目录下的README文件
- 编译u-boot-1.3.0

- ▶ 使用交叉编译器：arm-linux-tools-20061213.tar.gz
(gcc版本为3.4.4)

```
make ep7312_config  
make all
```

编译u-boot II

- 可以看到链接命令：

```
arm-linux-ld -Bstatic -T /home/xlanchen/workspace/u-boot-1.3.0/board/ep7312/u-boot.lds  
-Ttext 0xc0f80000 $UNDEF_SYM cpu/arm720t/start.o \ --start-group  
lib_generic/libgeneric.a board/ep7312/libep7312.a cpu/arm720t/libarm720t.a  
lib_arm/libarm.a fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a  
fs/jffs2/libjffs2.a fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a net/libnet.a  
disk/libdisk.a rtc/librtc.a dtb/libdtb.a drivers/libdrivers.a  
drivers/bios_emulator/libatibiosemu.a drivers/nand/libnand.a  
drivers/nand_legacy/libnand_legacy.a drivers/onenand/libonenand.a drivers/net/libnet.a  
drivers/serial/libserial.a drivers/sk98lin/libsk98lin.a post/libpost.a  
post/drivers/libpostdrivers.a common/libcommon.a libfdt/libfdt.a --end-group -L  
/usr/local/lib/gcc/arm-linux/3.4.4/soft-float -lgcc \ -Map u-boot.map -o u-boot
```

- 和最后objcopy命令

```
arm-linux-objcopy --gap-fill=0xff -O srec u-boot u-boot.srec  
arm-linux-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
```

编译u-boot III

● 目录中的文件

arm_config.mk	include	mips_config.mk
avr32_config.mk	lib_arm	mkconfig
blackfin_config.mk	lib_avr32	nand_spl
board	lib_blackfin	net
CHANGELOG	libfdt	nios2_config.mk
CHANGELOG-before-U-Boot-1.1.5	lib_generic	nios_config.mk
common	lib_i386	post
config.mk	lib_m68k	ppc_config.mk
COPYING	lib_microblaze	README
cpu	lib_mips	rtc
CREDITS	lib_nios	rules.mk
disk	lib_nios2	System.map
doc	lib_ppc	tools
drivers	m68k_config.mk	u-boot
dtb	MAINTAINERS	u-boot.bin
examples	MAKEALL	u-boot.map
fs	Makefile	u-boot.srec
i386_config.mk	microblaze_config.mk	

● 在skyeeye上运行u-boot

编译u-boot IV

- ▶ 拷贝skyeye-testsuite-1.2.5/u-boot/ep7312/skyeye.conf
- ▶ 运行

```
skyeye -c skyeye.conf -e u-boot  
或者  
skyeye -e u-boot
```

编译u-boot V

skyeye -e u-boot

```
Your elf file is little endian.  
arch: arm  
cpu info: armv4, arm720t, 41807200, fffffff0, 1  
mach info: name ep7312, mach_init addr 0x806bef0  
dbct info: turn on dbct!  
uart_mod:0, desc_in:, desc_out:, converter:  
SKYEYE: use arm7100 mmu ops  
start addr is set to 0xc0f80000 by exec file.
```

U-Boot 1.3.0 (Nov 28 2014 - 17:20:51)

```
DRAM: 16 MB  
Flash: 16 MB  
*** Warning - bad CRC, using default environment
```

```
In: serial  
Out: serial  
Err: serial  
Hit any key to stop autoboot: 0  
EP7312 #
```

编译u-boot VI

- ▶ 另一种运行方式：在skyeye.conf中

```
cpu: arm720t
mach: ep7312
mem_bank: map=M, type=RW, addr=0x00000000, size=0x00400000, file=u-boot.bin
mem_bank: map=I, type=RW, addr=0x80000000, size=0x00010000
mem_bank: map=M, type=RW, addr=0xc0000000, size=0x00200000
mem_bank: map=M, type=RW, addr=0xc0200000, size=0x00600000
mem_bank: map=M, type=RW, addr=0xc0800000, size=0x00800000
#lcd:type=ep7312,mod=gtk
#dbct:state=on
```

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

u-boot源代码 I

● 阅读源码根目录下的README

- ▶ 了解目录结构和各目录下的内容
- ▶ 了解u-boot的配置和编译的命令
- ▶ 知道在什么地方可以找到配置文件中的各个配置信息的含义

● 阅读根目录下的Makefile

- ▶ 了解配置和编译过程
- ▶ 了解ARCH、CPU、BOARD三个层次
 - ★ 例如：arm、arm720t、ep7312
- ▶ 了解编译的产物有哪些
- ▶ 了解目标映像的结构
- ▶ 知道u-boot的启动文件：start.S

u-boot源代码 II

● make ep7312_config

```
#####  
## ARM720T Systems  
#####  
...  
ep7312_config : unconfig  
    @$(MKCONFIG) $(@:_config=) arm arm720t ep7312
```

```
unconfig:  
    @rm -f $(obj)include/config.h $(obj)include/config.mk \  
        $(obj)board/*/config.tmp $(obj)board/*/*/config.tmp
```

```
MKCONFIG := $(SRCTREE)/mkconfig  
export MKCONFIG
```

u-boot源代码 III

● make all

```
ALL += $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(U_BOOT_NAND)
```

```
all: $(ALL)
```

```
$(obj)u-boot.srec: $(obj)u-boot  
    $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
```

```
$(obj)u-boot.bin: $(obj)u-boot  
    $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
```

```
$(obj)u-boot: depend version $(SUBDIRS) $(OBJJS) $(LIBS) $(LDSCRIPT)  
    UNDEF_SYM= '$(OBJDUMP) -x $(LIBS) |sed -n -e  
' s/.*\(__u_boot_cmd_.*\)/-u\1/p' |sort|uniq ';'\  
    cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF_SYM $(__OBJJS) \  
        --start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \  
        -Map u-boot.map -o u-boot
```

u-boot源代码 IV

```
#####  
# U-Boot objects....order is important (i.e. start must be first)  
  
OBJS = cpu/$(CPU)/start.o  
ifeq ($(CPU),i386)  
OBJS += cpu/$(CPU)/startl6.o  
OBJS += cpu/$(CPU)/reset.o  
endif  
ifeq ($(CPU),ppc4xx)  
OBJS += cpu/$(CPU)/resetvec.o  
endif  
ifeq ($(CPU),mpc85xx)  
OBJS += cpu/$(CPU)/resetvec.o  
endif  
ifeq ($(CPU),bf533)  
OBJS += cpu/$(CPU)/startl.o cpu/$(CPU)/interrupt.o cpu/$(CPU)/cache.o  
OBJS += cpu/$(CPU)/flush.o cpu/$(CPU)/init_sdram.o  
endif  
ifeq ($(CPU),bf537)  
OBJS += cpu/$(CPU)/startl.o cpu/$(CPU)/interrupt.o cpu/$(CPU)/cache.o  
OBJS += cpu/$(CPU)/flush.o cpu/$(CPU)/init_sdram.o  
endif  
ifeq ($(CPU),bf561)  
OBJS += cpu/$(CPU)/startl.o cpu/$(CPU)/interrupt.o cpu/$(CPU)/cache.o  
OBJS += cpu/$(CPU)/flush.o cpu/$(CPU)/init_sdram.o
```


cpu/arm720t/start.S I

- 阅读board/ep7312/u-boot.lds，了解start.S在u-boot中的位置

```
...
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;

    . = ALIGN(4);
    .text :
    {
        cpu/arm720t/start.o (.text)
        *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }

    . = ALIGN(4);
    .data : { *(.data) }
```

cpu/arm720t/start.S II

```
. = ALIGN(4);  
.got : { *(.got) }  
  
. = .;  
__u_boot_cmd_start = .;  
.u_boot_cmd : { *(.u_boot_cmd) }  
__u_boot_cmd_end = .;  
  
. = ALIGN(4);  
__bss_start = .;  
.bss : { *(.bss) }  
_end = .;  
}
```

- 了解start.S中的向量表的位置和含义

```
.globl _start
_start: b reset
        ldr pc, _undefined_instruction
        ldr pc, _software_interrupt
        ldr pc, _prefetch_abort
        ldr pc, _data_abort
#ifdef CONFIG_LPC2292
        .word 0xB4405F76 /* 2' s complement of the checksum of the vectors */
#else
        ldr pc, _not_used
#endif
        ldr pc, _irq
        ldr pc, _fiq
```

cpu/arm720t/start.S IV

```
_undefined_instruction: .word undefined_instruction
_software_interrupt:   .word software_interrupt
_prefetch_abort:      .word prefetch_abort
_data_abort:          .word data_abort
_not_used:             .word not_used
_irq:                  .word irq
_fiq:                  .word fiq

.balignl 16,0xdeadbeef
```

● 了解arm的启动方式

▶ reset

● 阅读reset

- ① set the cpu to SVC32 mode
- ② relocate U-Boot to RAM
- ③ Set up the stack
- ④ clear bss
- ⑤ 跳转到start_armboot (lib_arm/board.c)

lib_arm/board.c::start_armboot

- 阅读start_armboot()函数

- ▶ 各种初始化
- ▶ main_loop()死循环 (common/main.c)

Outline

1 BootLoader简介

2 u-boot

3 RedBoot

- RedBoot简介
- RedBoot的下载、编译和运行
- RedBoot的简单分析

4 小结和作业

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot


- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

RedBoot简介

- RedHat Embedded Debug and Bootstrap

- ▶ 是RedHat公司的一个标准嵌入式系统引导和调试环境

- ▶ 基于eCos操作系统 

- ★ eCos 是个可配置的操作系统 (<http://ecos.sourceware.org/>)

- ★ RedBoot是从eCos配置而来，RedBoot User's Guide

- ▶ 支持ARM系列、MIPS系列、PPC系列等平台

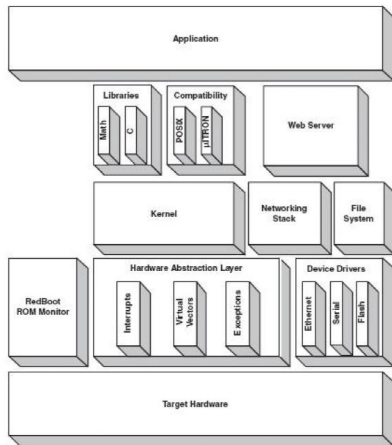
- ▶ 具有较高的可移植性

- ▶ 移植的重点是其体系结构平台相关层

- 目前最新版本为3.0

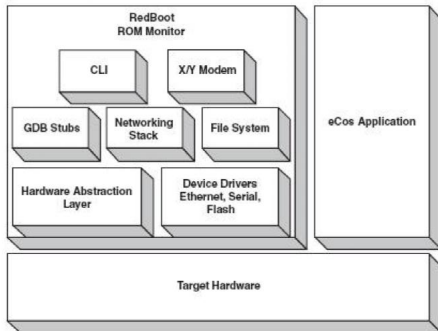
- ▶ 文档链接：<http://ecos.sourceware.org/docs-3.0/>

eCos 架构



<http://lh6.ggpht.com/rolle.xu/SDaAqa9yypI/AAAAAAAAAEw/jUYE6htfVNY/s800/ecos.jpg>

RedBoot架构



<http://lh5.ggpht.com/rolle.xu/SDaAqK9yyoI/AAAAAAAAAEo/eNK6Vz1u0eY/s800/redboot.jpg>

常用命令

- alias — Manipulate command line aliases
- baudrate — Set the baud rate for the system serial console
- cache — Control hardware caches
- channel — Select the system console channel
- cksum — Compute POSIX checksums
- disks — List available disk partitions.
- dump — Display memory.
- help — Display help on available commands
- iopeek — Read I/O location
- iopoke — Write I/O location
- gunzip — Uncompress GZIP compressed data
- ip_address — Set IP addresses
- load — Download programs or data to the RedBoot platform
- mcmp — Compare two segments of memory
- mcopy — Copy memory
- mfill — Fill RAM with a specified pattern
- ping — Verify network connectivity
- reset — Reset the device
- version — Display RedBoot version information

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

下载、建立开发环境

- 访问ecos主页：ecos.sourceforge.org，按照主页上的安装要求和安装方法进行
- 下载ecos-install.tcl并运行

```
wget --passive-ftp  
ftp://ecos.sourceforge.org/pub/ecos/ecos-install.tcl  
sh ecos-install.tcl
```

- 按照提示操作，给出你的安装目录或者使用缺省目录，编译器选择1,2,3,q。（时间有点长。）

下载、建立开发环境

Available prebuilt GNU tools:

```
[1] arm-eabi
[2] arm-elf (old)
[3] i386-elf
[4] m68k-elf
[5] mipsisa32-elf
[6] powerpc-eabi
[7] sh-elf
[q] Finish selecting GNU tools
```

("*" indicates tools already selected)

Please select GNU tools to download and install: 1

...

Please select GNU tools to download and install: 2

...

Please select GNU tools to download and install: 3

```
[*] arm-eabi
[*] arm-elf (old)
[*] i386-elf
[4] m68k-elf
[5] mipsisa32-elf
[6] powerpc-eabi
[7] sh-elf
[q] Finish selecting GNU tools
```

("*" indicates tools already selected)

Please select GNU tools to download and install: q

下载、建立开发环境

```
Retrieving GNU tools for arm-eabi
```

```
*****
```

```
Retrieving GNU tools for arm-elf (old)
```

```
*****
```

```
Retrieving GNU tools for i386-elf
```

```
*****
```

```
Retrieving eCos version 3.0
```

```
*****
```

```
Downloads complete.
```

```
If you wish to disconnect from the internet you may do so now.
```

```
Unpacking ecos-gnutools-arm-eabi-20120623.i386linux.tar.bz2...
```

```
Unpacking ecoscentric-gnutools-arm-elf-1.4-2.i386linux.tar.bz2...
```

```
Unpacking ecoscentric-gnutools-i386-elf-20081107-sw.i386linux.tar.bz2...
```

```
Unpacking ecos-3.0.i386linux.tar.bz2...
```

```
Generating /media/salmon/store/work/5教学/2014FallEmbeddedOS/ecos/ecosenv.sh
```

```
Generating /media/salmon/store/work/5教学/2014FallEmbeddedOS/ecos/ecosenv.csh
```

```
-----
```

```
In future, to establish the correct environment for eCos,
```

```
run one of the following commands:
```

```
. /media/salmon/store/work/5教学/2014FallEmbeddedOS/ecos/ecosenv.sh (for sh/bash  
users); or
```

```
source /media/salmon/store/work/5教学/2014FallEmbeddedOS/ecos/ecosenv.csh (for  
csh/tcsh users)
```

```
It is recommended you append these commands to the end of your shell startup files  
such as $HOME/.profile or $HOME/.login
```

```
-----
```

```
Installation complete!
```

下载、建立开发环境

- 查看ecos的目录以及相关目录

```
tree ecos/ -L 1
```

```
ecos/  
├── ecos-3.0  
├── ecosenv.csh  
├── ecosenv.sh  
├── ecos-install.tcl  
└── gnutools
```

2 directories, 3 files

其中，gnutools目录中就是我们选择下载的交叉编译器

```
tree ecos/gnutools/ -L 1
```

```
ecos/gnutools/ ls  
├── arm-eabi  
├── arm-elf  
└── i386-elf
```

3 directories, 0 files

下载、建立开发环境

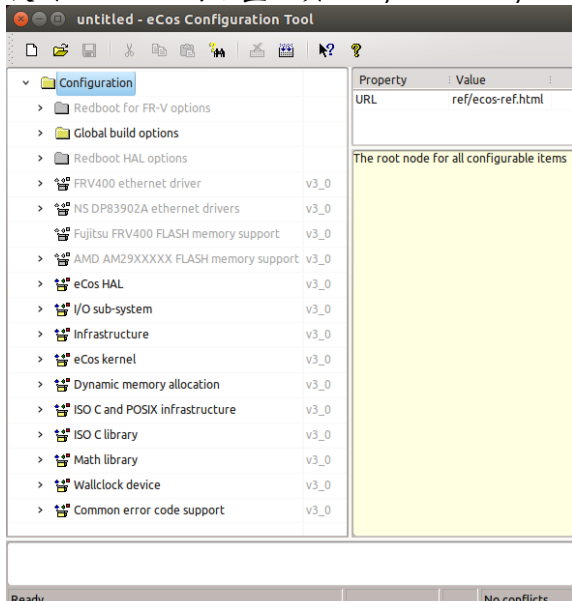
- 运行`ecosenv.sh`，建立开发环境
 - `. ecosenv.sh` 或者 `source ecosenv.sh`
- 查看`ecosenv.sh`是否运行成功，

```
echo $PATH
```

```
/PATH_TO_ECOS/gnutools/i386-elf/bin:/PATH_TO_ECOS/gnutools/arm-elf/bin:/PATH_TO_ECOS/gnutools/arm-eabi/bin:/PATH_TO_ECOS/ecos-3.0/tools/bin:...
```

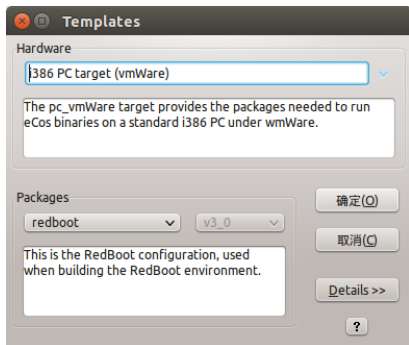
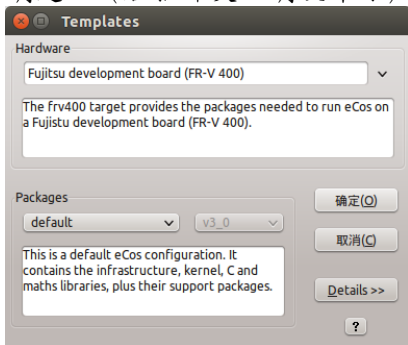
为pc配置、编译redboot

- 使用ecos-3.0的配置工具ecos/ecos-3.0/tools/bin/configtool



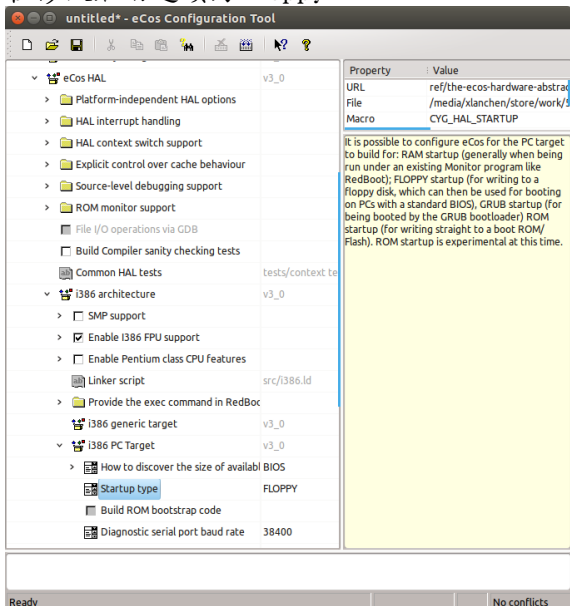
为pc配置、编译redboot

- 在build菜单中选择Templates得到如下左边的界面。
其中，Hardware选择“i386 PC target (vmWare)”；
Packages选择“redboot”。
确定。（若报冲突，确认即可）



为pc配置、编译redboot

● 在修改启动选项为floppy



为pc配置、编译redboot

- 创建一个目录virtualbox-i386，将配置结果保存到这个目录下，命名为“i386.ecc”
- 进入virtualbox-i386目录，可以看到：

```
tree virtualbox-i386/ -L 1
```

```
virtualbox-i386/  
├── i386_build  
├── i386.ecc  
└── i386_install
```

```
2 directories, 1 file
```

其中，ecc是eCos Configuration的缩写，表示配置文件

- 进入i386_build，运行make。（这个挺快的）

为pc配置、编译redboot

- make结束后，到i386_install目录下，可以看到生成了redboot.bin文件和redboot.elf文件

```
tree virtualbox-i386/i386_install/bin/
```

```
virtualbox-i386/i386_install/bin/  
├── redboot.bin  
└── redboot.elf
```

```
0 directories, 2 files
```

在virtualbox上运行redboot

- 安装virtualbox

```
sudo apt-get install virtualbox
```

- 为redboot.bin生成软盘镜像floppy_redboot.img

```
dd conv=sync if=redboot.bin of=floppy_redboot.img bs=1440k
```

```
dd conv=sync if=bin/redboot.bin of=floppy_redboot.img bs=1440k
```

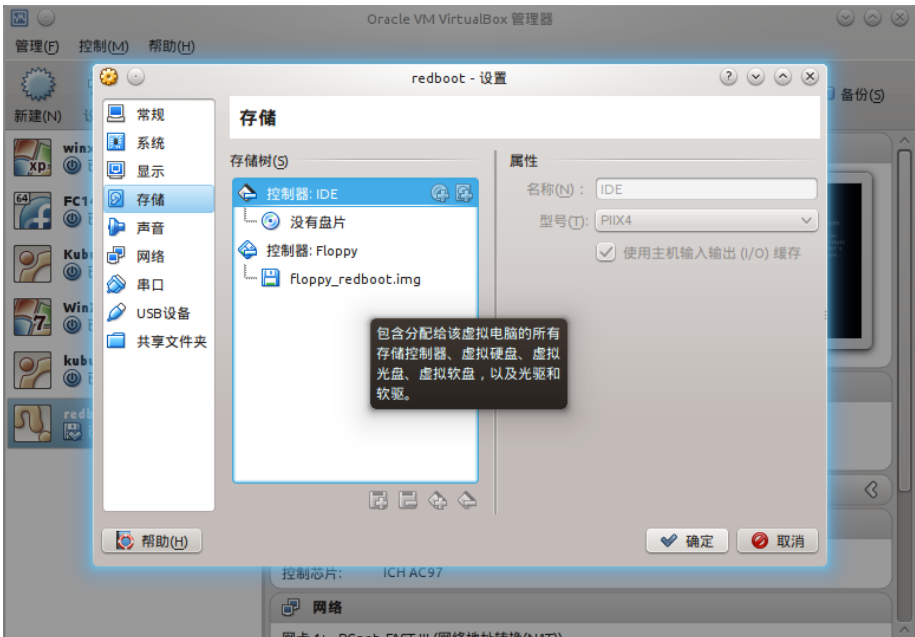
记录了0+1 的读入

记录了1+0 的写出

1474560字节(1.5 MB)已复制，0.00415758 秒，355 MB/秒

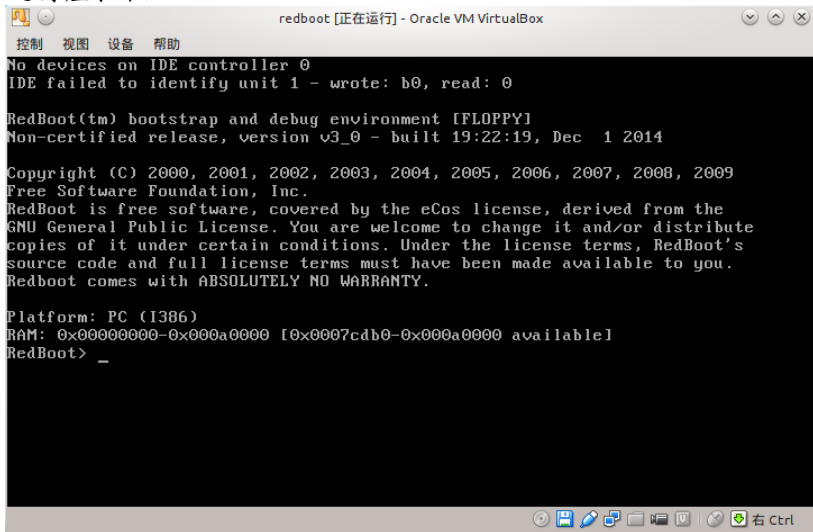
- 运行virtualbox，创建一个虚拟机redboot，在存储中添加一个软盘控制器，使用上面生成的floppy_redboot.img

在virtualbox上运行redboot



在virtualbox上运行redboot

● 运行虚拟机redboot



```
redboot [正在运行] - Oracle VM VirtualBox
控制 视图 设备 帮助
No devices on IDE controller 0
IDE failed to identify unit 1 - wrote: b0, read: 0

RedBoot(tm) bootstrap and debug environment [FLOPPY]
Non-certified release, version v3_0 - built 19:22:19, Dec 1 2014

Copyright (C) 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009
Free Software Foundation, Inc.
RedBoot is free software, covered by the eCos license, derived from the
GNU General Public License. You are welcome to change it and/or distribute
copies of it under certain conditions. Under the license terms, RedBoot's
source code and full license terms must have been made available to you.
Redboot comes with ABSOLUTELY NO WARRANTY.

Platform: PC (I386)
RAM: 0x00000000-0x000a0000 [0x0007c0b0-0x000a0000 available]
RedBoot> _
```

Outline

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

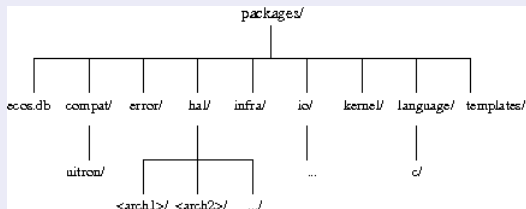
4 小结和作业

目录结构分析

- 参见

The eCos Component Writer's Guide

packages 目录层次结构



目录结构分析

```
tree ecos-3.0/packages/  
-L 1
```

```
ecos-3.0/packages/  
├── ChangeLog  
├── compat  
├── cygmon  
├── devs  
├── ecosadmin.tcl  
├── ecos.db  
├── error  
├── fs  
├── hal  
├── infra  
├── io  
├── isoinfra  
├── kernel  
├── language  
├── net  
├── NEWS  
├── pkgconf  
├── redboot  
├── services  
└── templates
```

16 directories, 4 files

- hal目录是体系结构相关部分
- io目录中是一些驱动
- language目录中提供一些语言支持库，例如C库
- template是指为某种目的build而建立的相应模板配置，比如选取一些包，配置一些选项等等
- **ecos.db**是一个数据库文件，配置工具从这个文件中获得各种信息

观察ecos.db

目录结构分析

● 观察hal目录

```
tree ecos-3.0/packages/hal/ -L 1
```

```
ecos-3.0/packages/hal/
```

```
|— arm  
|— calmrisc16  
|— calmrisc32  
|— common  
|— cortexm  
|— fr30  
|— frv  
|— h8300  
|— i386  
|— m68k  
|— mips  
|— mn10300  
|— powerpc  
|— sh  
|— sparc  
|— sparclite  
|— synth  
|— v85x
```

```
18 directories, 0 files
```

目录结构分析

- 观察ha1/i386目录

```
tree ecos-3.0/packages/ha1/i386/ -L 1
```

```
ecos-3.0/packages/ha1/i386/
```

```
|— arch  
|— generic  
|— pc  
|— pcmb
```

```
4 directories, 0 files
```

目录结构分析

● 观察ha1/arm目录

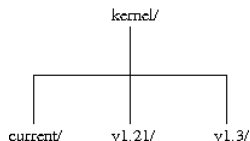
```
tree ecos-3.0/packages/ha1/arm/ -L 1
```

```
ecos-3.0/packages/ha1/arm/
```

```
|—— aeb  
|—— aim711  
|—— arch  
|—— arm9  
|—— at91  
|—— cma230  
|—— e7t  
|—— ebsa285  
|—— edb7xxx  
|—— gps4020  
|—— integrator  
|—— lpc24xx  
|—— lpc2xxx  
|—— mac7100  
|—— pid  
|—— sallx0  
|—— snds  
|—— xscale
```

```
18 directories, 0 files
```

Package的版本

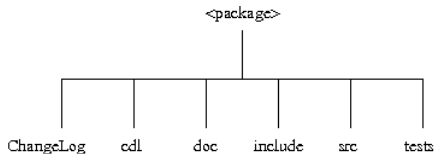


- 一般current最新，多用于ecos开发者
- 我们所使用的版本

```
tree ecos-3.0/packages/kernel/ -L 1
```

```
ecos-3.0/packages/kernel/  
└── v3_0
```


版本下的通用目录结构



- 对比kernel/v3_0的目录结构

```
tree ecos-3.0/packages/kernel/ -L 2
```

```
ecos-3.0/packages/kernel/
├── v3_0
│   ├── cdl
│   ├── ChangeLog
│   ├── doc
│   ├── host
│   ├── include
│   ├── src
│   └── tests
```

7 directories, 1 file

源文件

- 库源文件
- 链接描述文件 (*.ld)
- 外部头文件
- 文档 (doc目录)
- 测试用例 (tests目录)
- CDL脚本
 - (组件定义语言CDL, Component Definition Language)
 - ▶ CDL是eCos组件框架的一个关键部分。
 - ★ eCos中所有的包都必须具有至少一个CDL脚本对其进行描述。
 - ★ CDL脚本包含了该包中所有配置选项的详细信息，并提供了如何对该包进行编译的信息。
- ChangeLog

i386相关

```
tree ecos-3.0/packages/ha1/i386/ -L 1
```

```
ecos-3.0/packages/ha1/i386/
```

```
|— arch  
|— generic  
|— pc  
|— pcmb
```

```
4 directories, 0 files
```

- arch：为i386处理器体系结构提供支持
- generic：为普通IA32处理器提供支持，从80386→最新的Pentium/Athlon。
- pcmb中的mb是指MotherBoard，即提供对PC主板的支持
- pc：提供标准的pc目标机的支持

i386相关

```
tree ecos-3.0/packages/hal/i386/arch/
```

```
ecos-3.0/packages/hal/i386/arch/  
├── v3_0  
├── cdl  
│   └── hal_i386.cdl  
├── ChangeLog  
├── include  
│   ├── arch.inc  
│   ├── basetype.h  
│   ├── hal_arch.h  
│   ├── hal_cache.h  
│   ├── hal_intr.h  
│   ├── hal_io.h  
│   ├── hal_smp.h  
│   ├── i386.inc  
│   └── i386_stub.h  
└── src  
    ├── context.S  
    ├── hal_misc.c  
    ├── hal_syscall.c  
    ├── i386.ld  
    ├── i386_stub.c  
    ├── redboot_linux_exec.c  
    └── vectors.S
```

4 directories, 18 files

i386相关

```
tree ecos-3.0/packages/ha1/i386/generic/
```

```
ecos-3.0/packages/ha1/i386/generic/
```

```
├── v3_0
├── cdl
│   └── hal_i386_generic.cdl
├── ChangeLog
├── include
│   ├── var_arch.h
│   ├── variant.inc
│   └── var_intr.h
└── src
    └── var_misc.c
```

```
4 directories, 6 files
```

i386相关

tree ecos-3.0/packages/hal/i386/pc/

```
ecos-3.0/packages/hal/i386/pc
├── v3_0
│   ├── cd1
│   │   └── hal_i386_pc.cd1
│   ├── Changelog
│   ├── doc
│   │   └── RELEASENOTES.txt
│   └── include
│       ├── hal_diag.h
│       ├── pkgconf
│       │   ├── mlt_i386_pc_floppy.h
│       │   ├── mlt_i386_pc_floppy.ldi
│       │   ├── mlt_i386_pc_floppy.mlt
│       │   ├── mlt_i386_pc_grub.h
│       │   ├── mlt_i386_pc_grub_hi.h
│       │   ├── mlt_i386_pc_grub_hi.ldi
│       │   ├── mlt_i386_pc_grub_hi.mlt
│       │   ├── mlt_i386_pc_grub.ldi
│       │   ├── mlt_i386_pc_grub.mlt
│       │   ├── mlt_i386_pc_ram.h
│       │   ├── mlt_i386_pc_ram_hi.h
│       │   ├── mlt_i386_pc_ram_hi.ldi
│       │   ├── mlt_i386_pc_ram_hi.mlt
│       │   ├── mlt_i386_pc_ram.ldi
│       │   ├── mlt_i386_pc_ram.mlt
│       │   ├── mlt_i386_pc_rom.h
│       │   └── mlt_i386_pc_rom.ldi
│       ├── platform.inc
│       ├── plf_arch.h
│       ├── plf_intr.h
│       ├── plf_io.h
│       ├── plf_misc.h
│       └── plf_stub.h
├── misc
│   ├── menu.lst
│   ├── redboot_FLOPPY_D850GB.ecm
│   ├── redboot_FLOPPY.ecm
│   ├── redboot_FLOPPY_SMP.ecm
│   ├── redboot_GRUB.ecm
│   └── redboot_ROM.ecm
└── src
    ├── hal_diag.c
    ├── PKGconf.mak
    ├── plf_misc.c
    ├── plf_stub.c
    ├── romboot.ld
    └── romboot.S
```

7 directories, 39 files

i386相关

```
tree ecos-3.0/packages/ha1/i386/pcmb/
```

```
ecos-3.0/packages/ha1/i386/pcmb/
├── v3_0
│   ├── cdl
│   │   └── ha1_i386_pcmb.cdl
│   ├── ChangeLog
│   ├── include
│   │   ├── pcmb.inc
│   │   ├── pcmb_intr.h
│   │   ├── pcmb_io.h
│   │   └── pcmb_serial.h
│   ├── src
│   │   ├── pcmb_misc.c
│   │   ├── pcmb_screen.c
│   │   ├── pcmb_serial.c
│   │   └── pcmb_smp.c
│   └── support
│       └── gfxmode.c
```

5 directories, 11 files

redboot.bin的生成

- 观察ecos-3.0/packages/ha1/i386/pc/v3_0/cdl/ha1_i386_pc.cdl , 寻找redboot.bin

```
cdl_component CYGBLD_BUILD_REDBOOT_BIN {
    display " Build RedBoot binary image"
    no_define
    default_value 1

    cdl_option CYGBLD_BUILD_REDBOOT_BIN_FLOPPY {
        display " Build Redboot FLOPPY binary image"
        active_if CYGBLD_BUILD_REDBOOT
        active_if { CYG_HAL_STARTUP == " FLOPPY" }
        calculated 1
        no_define
        description " This option enables the conversion of the Redboot
            ELF image to a binary image suitable for
            copying to a floppy disk."

        make -priority 325 {
            <PREFIX>/bin/redboot.bin : <PREFIX>/bin/redboot.elf
            $(OBJCOPY) -O binary $< $@
        }
    }
}
```


redboot.bin的生成

```
cdl_option CYGBLD_BUILD_REDBOOT_BIN_ROM {
    display " Build Redboot ROM binary image"
    active_if CYGBLD_BUILD_REDBOOT
    active_if { CYG_HAL_STARTUP == " ROM" }
    calculated 1
    no_define
    description " This option enables the conversion of the Redboot
                ELF image to a binary image suitable for ROM
                programming."

make -priority 325 {
    <PREFIX>/bin/redboot.bin : <PREFIX>/bin/redboot.elf
    $(OBJCOPY) -O binary $< $(@:.bin=.img)
    $(OBJCOPY) -O binary $(PREFIX)/bin/romboot.elf $(PREFIX)/bin/romboot.img
    dd if=/dev/zero of=$@ bs=1024 count=64 conv=sync
    dd if=$(@:.bin=.img) of=$@ bs=512 conv=notrunc, sync
    dd if=$(PREFIX)/bin/romboot.img of=$@ bs=256 count=1 seek=255 conv=notrunc
}
}
}
```

redboot.bin的生成

- 观察ecos-3.0/packages/redboot/v3_0/cd1/redboot.cd1，寻找redboot.elf

```
cd1_component CYGBLD_BUILD_REDBOOT {
    display " Build Redboot ROM ELF image"
    default_value 0
    requires CYGPKG_INFRA
    requires CYGPKG_ISOINFRA
    ...
    compile main.c
    compile misc_funs.c io.c parse.c ticks.c syscall.c alias.c
    compile -library=libextras.a load.c

    make -priority 320 {
        <PREFIX>/bin/redboot.elf : $(PREFIX)/lib/target.ld $(PREFIX)/lib/vectors.o
        $(PREFIX)/lib/libtarget.a $(PREFIX)/lib/libextras.a
        @sh -c " mkdir -p $(dir $@)"
        $(CC) -c $(INCLUDE_PATH) $(ACTUAL_CFLAGS) -o $(PREFIX)/lib/version.o
        $(REPOSITORY)/$(PACKAGE)/src/version.c
        $(CC) $(LDFLAGS) -L$(PREFIX)/lib -Ttarget.ld -o $@ $(PREFIX)/lib/version.o
    }
    ...
}
```

redboot.bin的生成

- 观察`ecos-3.0/packages/hal/i386/arch/v3_0/cdl/hal_i386.cdl` ,
寻找`vectors.o`

```
cdl_package CYGPKG_HAL_I386 {  
    display " i386 architecture"  
    parent CYGPKG_HAL  
    hardware  
    include_dir cyg/hal  
    define_header hal_i386.h  
    description "  
        The i386 architecture HAL package provides generic  
        support for this processor architecture. It is also  
        necessary to select a specific target platform HAL  
        package."  
  
    implements CYGINT_PROFILE_HAL_MCOUNT  
  
    compile hal_misc.c context.S i386_stub.c hal_syscall.c
```

redboot.bin的生成

```
make {  
    <PREFIX>/lib/vectors.o : <PACKAGE>/src/vectors.S  
    $(CC) -Wp,-MD,vectors.tmp $(INCLUDE_PATH) $(CFLAGS) -c -o $@ $<  
    @echo $@ " : \\" > $(notdir $@).deps  
    @tail -n +2 vectors.tmp >> $(notdir $@).deps  
    @echo >> $(notdir $@).deps  
    @rm vectors.tmp  
}  
  
make {  
    <PREFIX>/lib/target.ld: <PACKAGE>/src/i386.ld  
    $(CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $(INCLUDE_PATH) $(CFLAGS) -o $@ $<  
    @echo $@ " : \\" > $(notdir $@).deps  
    @tail -n +2 target.tmp >> $(notdir $@).deps  
    @echo >> $(notdir $@).deps  
    @rm target.tmp  
}
```

redboot.bin的生成

- 观察生成的virtualbox-i386/i386_build/hal/i386/arch/v3_0/vectors.o.deps文件

```
/PATH_TO_ECOS/virtualbox-i386/i386_install/lib/vectors.o : \  
/PATH_TO_ECOS/ecos-3.0/packages/hal/i386/arch/v3_0/src/vectors.S \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/pkgconf/system.h \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/pkgconf/hal.h \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/pkgconf/hal_i386.h \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/pkgconf/hal_i386_pc.h \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/pkgconf/hal_i386_pcmb.h \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/cyg/hal/arch.inc \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/cyg/hal/i386.inc \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/cyg/hal/variant.inc \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/cyg/hal/platform.inc \  
/PATH_TO_ECOS/virtualbox-i386/i386_install/include/cyg/hal/pcmb.inc
```

启动流程

- i386 PC (vmWare)
- ./ecos-3.0/packages/hal/i386/arch/v3_0/src/vectors.S
→cyg_start (阅读：开始→结束)

```
#=====
# Real startup code. We jump here from the various reset vectors to set up the
# world.

.text
.globl _start

_start:
    hal_cpu_init
    hal_smp_init
    hal_diag_init
    hal_mmu_init
    hal_memc_init
    hal_intc_init
    hal_cache_init
    hal_timer_init
```

启动流程

```
# Loading the stack pointer seems appropriate now.
# In SMP systems, this may not be the interrupt stack we
# actually need to use for this CPU. We fix that up later.

movl $__interrupt_stack, %esp
....
....
extern cyg_start
call cyg_start

# Hmm. Not expecting to return from cyg_start.
l:    hlt
      jmp 1b
```

- ./ecos-3.0/packages/hal/i386/pcmb/v3_0/include/pcmb.inc : hal_cpu_init
- ./ecos-3.0/packages/redboot/v3_0/src/main.c : cyg_start
- 关于hal_cpu_init，有三个文件提供对hal_cpu_init的定义，按照依赖的顺序

启动流程

❶ arch/v3_0/include/arch.inc

```
#ifndef CYGPKG_HAL_I386_CPU_INIT_DEFINED
    # Initialize CPU
    .macro hal_cpu_init
    .endm
#endif /* !CYGPKG_HAL_I386_CPU_INIT_DEFINED */
```


启动流程

② pc/v3_0/include/platform.inc ,

此文件内部首先包含了pcmb.inc。然后有定义：

```
##=====
## CPU initialization
#ifndef CYGPKG_HAL_I386_CPU_INIT_DEFINED
#define CYGPKG_HAL_I386_CPU_INIT_DEFINED
##=====
## ROM and GRUB startup
## ...

#if defined(CYG_HAL_STARTUP_ROM) || defined(CYG_HAL_STARTUP_GRUB)

    .macro hal_cpu_init
    ...
    .endm

#endif

##=====
## RAM startup
#ifdef CYG_HAL_STARTUP_RAM

    .macro hal_cpu_init
    .endm

#endif /* CYG_HAL_STARTUP_RAM */
#endif // CYGPKG_HAL_I386_CPU_INIT_DEFINED
```

启动流程

● pcmb/v3_0/include/pcmb.inc

```
##=====
## CPU initialization
#ifndef CYGPKG_HAL_I386_CPU_INIT_DEFINED
#ifdef CYG_HAL_STARTUP_FLOPPY
#define CYGPKG_HAL_I386_CPU_INIT_DEFINED
    .macro hal_cpu_init
        ...
    .endm /* hal_cpu_init */
#endif /* CYG_HAL_STARTUP_FLOPPY */
#endif // CYGPKG_HAL_I386_CPU_INIT_DEFINED
```

● 显然上述pcmb.inc中的定义将会起作用，阅读之。

- ▶ 准备栈
- ▶ 加载redboot
- ▶ 关中断
- ▶ 加载GDT表和IDT表
- ▶ 切换到保护模式
- ▶ 重置eflags寄存器为全0

Outline

- 1 BootLoader简介
- 2 u-boot
- 3 RedBoot
- 4 小结和作业**

小结

1 BootLoader 简介

- Boot Loader 的概念
- Boot Loader 的安装
- Boot Loader 的启动过程和操作模式
- Boot Loader 的主要任务和典型结构框架
- 部分开源的 Boot Loader

2 u-boot

- u-boot 简介
- 编译 u-boot
- 简单分析 u-boot 源码

3 RedBoot

- RedBoot 简介
- RedBoot 的下载、编译和运行
- RedBoot 的简单分析

4 小结和作业

作业：

- ① 嵌入式Linux的软件层次。
- ② 常用的嵌入式GUI。
- ③ BIOS
- ④ 什么是BootLaoder，其主要任务是什么？
有哪些工作（操作）模式。
- ⑤ 列举5个BootLoader。

Project4

- 安装dosbox，在dosbox中成功编译并运行 μ C/OS-II

- ▶ 通过阅读 μ C/OS-II的源代码，掌握 μ C/OS-II的编译过程和代码组成
- ▶ 了解各个目录下的代码在 μ C/OS-II中的作用
- ▶ 分析 μ C/OS-II是如何在dosbox中运行成功的

- 安装bochs，在bochs中成功引导运行 μ C/OS-II的EX1_x86L

- ▶ 提示：

- ★ 参考linux-

- 2.4.18中的启动代码，编写16位代码作为bootsect，加载 μ C/OS-II内核，进入保护模式，跳转到 μ C/OS-II内核入口运行

- ★ 改写 μ C/OS-II中的汇编代码

- ★ 设置好中断向量表

- ★

- 通过直接写VGA（0xB80000开始），25行 \times 80列，来输出信息(格式：双字符=ASCII码+颜色)

- ▶ 要求：

- ★ 添加的文件或者修改的文件使用独立的目录/子目录

- ★ 尽可能不要修改 μ COS-II目录下的文件，但可以拷贝出来修改

- ★ 提供使用手册

- ★ 以成功运行EX1_x86L为标准

- 给出详细的实验报告

Thanks !

The end.