





嵌入式系统

An Introduction to Embedded System

第四课 嵌入式系统的 BootLoader技术

课程大纲

-  Bootloader程序的基本概念
-  Bootloader典型框架结构
-  S3C2410 Bootloader代码分析
-  嵌入式软件开发环境建立实验

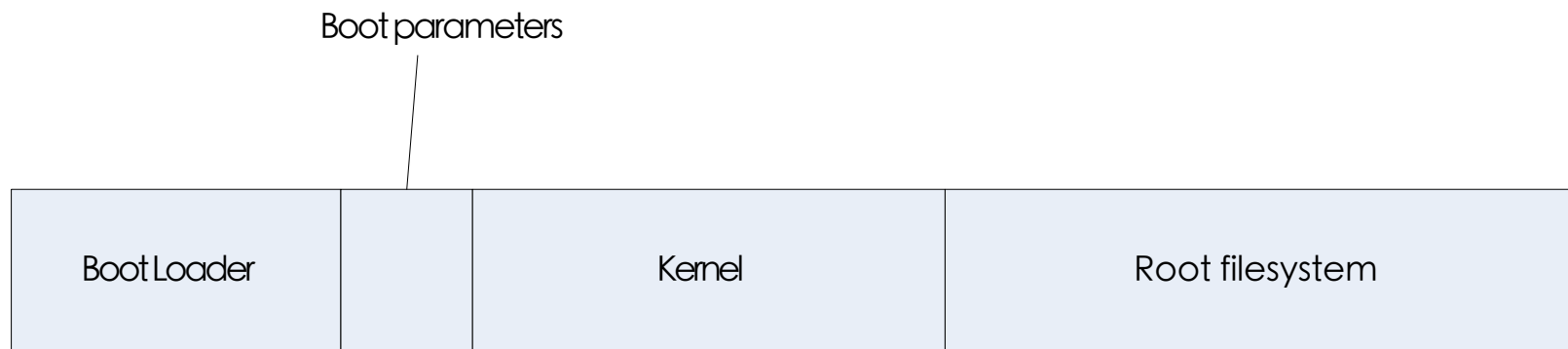
什么是bootloader?

- 上电后的第一段代码可能是：
 - 程序本身：小规模单片机程序
 - bootloader：
 - BIOS：PC
- 采用bootloader是为了方便程序的下载、启动和最初的调试

Boot Loader程序基本概念

- Boot Loader是在系统启动时激活，在操作系统内核运行之前运行的一段程序
 - 初始化硬件设备和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

固态存储设备的典型空间分配结构



固态存储设备指ROM、EEPROM或FLASH等

如何把程序放到嵌入式机器里？

- 预先烧录：
 - ROM
 - flash
 - 将整个MCU当作flash来烧录
- 专用硬件接口：
 - JTAG
 - SWI
- ISP：
 - 芯片厂家实现的bootloader

bootloader

- 芯片内的ISP程序
 - 不占用地址空间（特殊空间）
 - 二进制协议
 - 只能下载烧录程序
- 第三方bootloader
 - 在ROM/flash地址空间中
 - 需要由其他方式预先烧录
 - 文本协议
 - 可以做简单调试

ISP or ICP?

- 不同厂家的不同名词而已
- 有ISP的好处是不需要特殊的编程（烧录）硬件，有串口就可以
- IAP = ?

程序可以在

- ROM
- flash
- SDIO
- IDE
- SATA
- 但是只有ROM或flash的可以启动时直接访问

程序可以运行在

- ROM
- flash
- SRAM
- DRAM
- 上电时的第一条指令只能在ROM或flash中
- DRAM需要初始化控制器之后才能访问

RESET

- x86: 0x0FFF FFF0, 16位实模式
- ARM: 0x0000 0000或0xFFFF 0000
- PPC: 0x0000 0100或0xFFFF 0100
- 启动地址是启动代码的地址
- 启动地址是中断向量表, 所以reset是一种中断

- Boot Loader所支持的硬件环境
 - 每种不同的CPU体系结构都有不同的Boot Loader。
 - 也依赖于具体的嵌入式板级设备的配置。
- Boot Loader相关的设备和机制
 - 主机和目标机之间通过串口建立连接，通过串口进行I/O。
- Boot Loader的启动过程
 - 启动过程可以分为阶段1和阶段2两部分。

- Boot Loader的操作模式

- 启动加载
- 下载模式

- Boot Loader

- 串口协议
- 网络协议：TFTP

1、xmodem是最早的协议之一，一种由几乎所有通讯程序支持的文件传送协议, 传送128个字节信息块；
ymodem和zmodem都是它的改进协议，
2、ymodem传送1024字节长的信息块，快于xmodem并且可送多个文件；
3、zmodem速度快于ymodem和xmodem，且可以更好地在断开后恢复传输。

从一个最小的“操作系统”说起

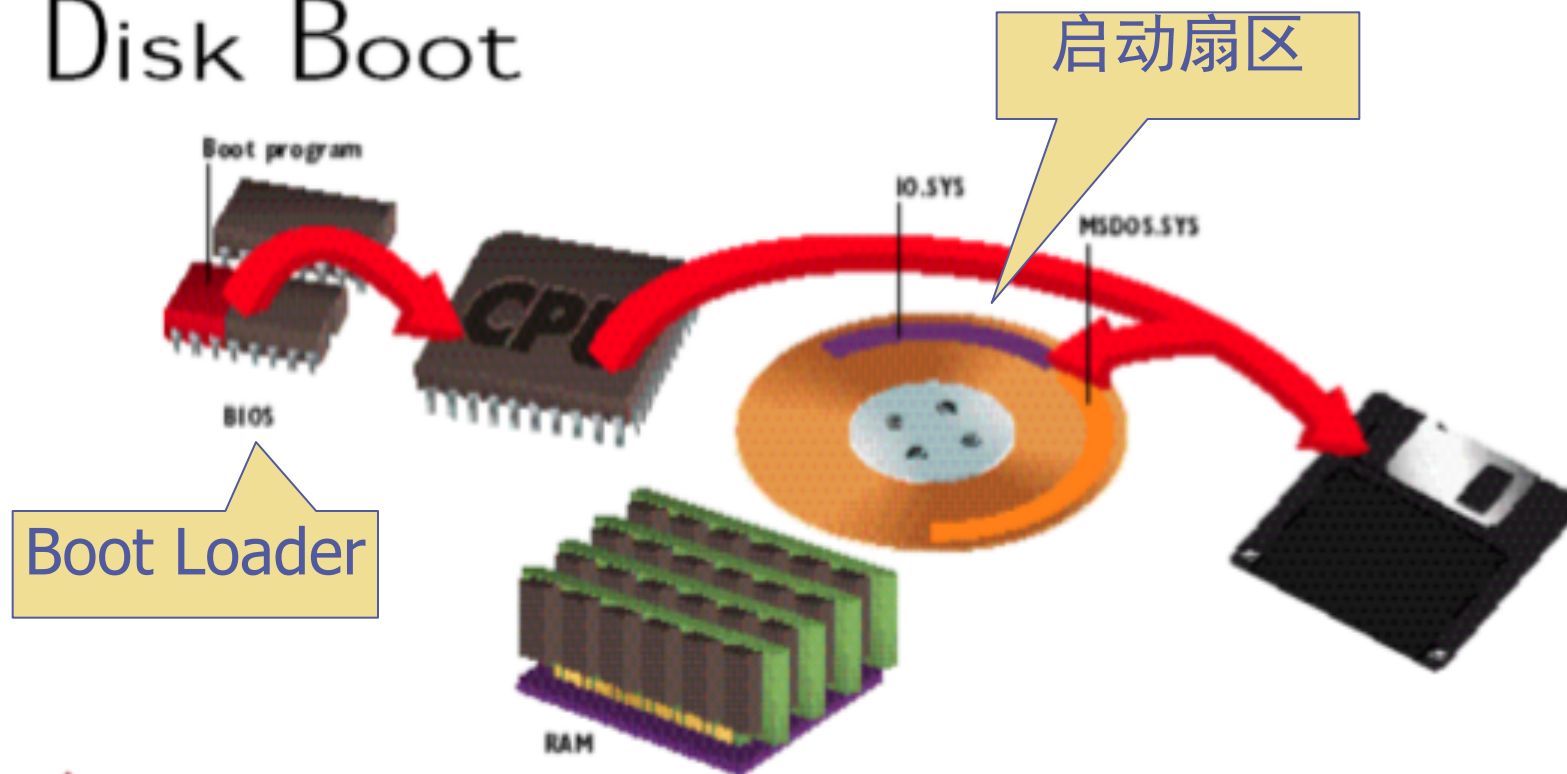
- 摘自《自己动手写操作系统》

```
1  org 07c00h      ; 告诉编译器程序加载到7c00处
2  mov ax, cs
3  mov ds, ax
4  mov es, ax
5  call DispStr     ; 调用显示字符串例程
6  jmp $           ; 无限循环
7  DispStr:
8  mov ax, BootMessage
9  mov bp, ax       ; ES:BP = 串地址
10 mov cx, 16       ; CX = 串长度
11 mov ax, 01301h   ; AH = 13, AL = 01h
12 mov bx, 000ch    ; 页号为0 (BH = 0) 黑底红字 (BL = 0Ch, 高亮)
13 mov dl, 0
14 int 10h         ; 10h 号中断
15 ret
16 BootMessage:     db "Hello, OS world!"
17 times 510-($-$$) db 0 ; 填充剩下的空间, 使生成的二进制代码恰好为512字节
18 dw 0xaa55       ; 结束标志
```



PC机的引导装载简介 (1/2)

Disk Boot



PC机的引导装载简介（2/2）

PC机是通过BIOS来启动机器:

- 1、当PC机加电之后，BIOS启动相应的程序完成机器自检POST (Power-On Self Test) ；
- 2、寻找可以引导的驱动器，即启动盘IPL (Initial Program Load Device) ；
- 3、若找到合法的引导扇区（以0xAA55结束），那么就会将引导扇区的内容（共512字节）装载到内存0x0000:7C00处。
- 4、BIOS把控制权限交给系统引导扇区的程序。

BIOS Boot Specification

□ BIOS Boot Specification

- ✓ 用于描述**BIOS**识别各类**IPL(Initial Program Load)**设备，并进行引导设备优先级排序、启动加载等功能。

□ **BIOS**启动时采用实模式（**real mode**）寻址，寻址空间为**16**位。

□ **BIOS**启动程序采用汇编语言编写，这使得编写**BIOS**程序变得比较复杂。



BIOS—内存分布图

□ BIOS内存分布图

■ BIOS Entry

✓ 0xFFFF0h

■ BIOS Boot Block

✓ 0xFE000h ~ 0xFFFFFh

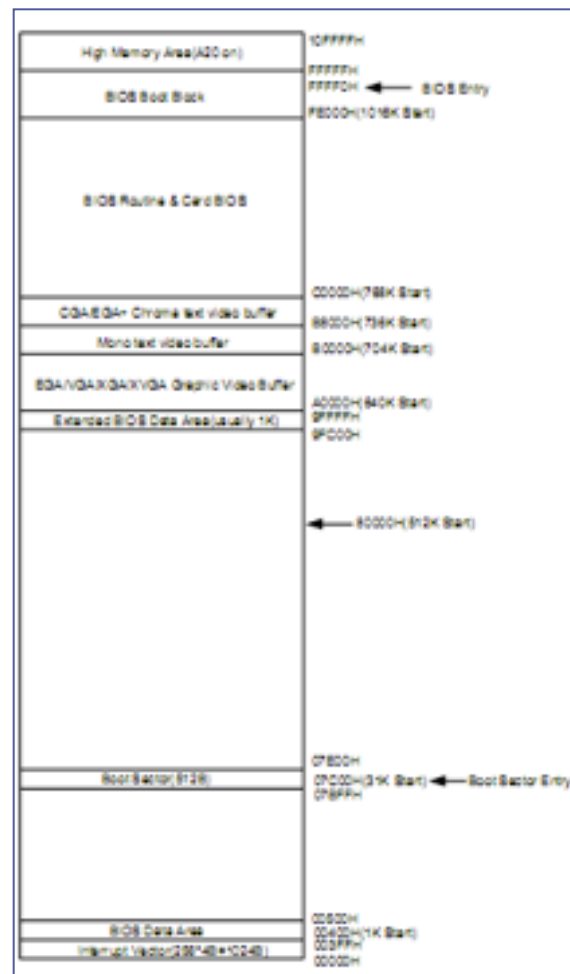
■ (Extended) BIOS Data Area

■ Boot Sector

✓ 0x7c00h

■ Interrupt Vector

✓ 0x0000h ~ 0x3ffh



BIOS—构成分析

BIOS映像文件

✓ BIOS-bochs-latest

源代码

✓ C程序: `rombios.c`、
`rombios32.c`

✓ 汇编代码:
`rombios32start.S`

Makefile

BIOS映像生成工具

✓ `biossums.c`

名称	大小	类型
acpi-dsdt.dsl	18 KB	DSL 文件
acpi-dsdt.hex	20 KB	HEX 文件
apmbios.S	7 KB	Assembler Source
bios_usage	1 KB	文件
BIOS-bochs-latest	128 KB	文件
BIOS-bochs-legacy	64 KB	文件
biossums.c	14 KB	C Source
Makefile	4 KB	文件
Makefile.in	4 KB	IN 文件
nakesym.perl	1 KB	PERL 文件
notes	2 KB	文件
rombios32.c	41 KB	C Source
rombios32.ld	1 KB	LD 文件
rombios32start.S	3 KB	Assembler Source
rombios.c	299 KB	C Source
rombios.h	3 KB	C/C++ Header
usage.cpp	3 KB	C++ Source
VGABIOS-elpin-2.40	32 KB	40 文件
VGABIOS-elpin-LICENSE	1 KB	文件
VGABIOS-lgpl-latest	38 KB	文件
VGABIOS-lgpl-latest-cirrus	35 KB	文件
VGABIOS-lgpl-latest-cirrus-debug	35 KB	文件
VGABIOS-lgpl-latest-debug	39 KB	文件
VGABIOS-lgpl-README	8 KB	文件

BIOS—映像文件分析

□ BIOS-bochs-latest

The image shows two windows side-by-side. The left window is UltraEdit, displaying a hex dump of a BIOS image. The right window is the Bochs for Windows - Console, showing the output of the Bochs emulator.

UltraEdit - [E:\temp\bochs-2.3.6\bios\BIOS-bochs-latest]

File (F) Edit (E) Search (S) Insert (I) Project (P) View (V) Format (T) List (L) Macro (M)
Advanced (A) Window (W) Help (H)

事务处理.txt 想法.txt bochsrc.bsrc Makefile BIOS-bochs-latest

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0001ff10h:	65	53	6F	66	74	20	53	2E	41	2E	20	57	72	69	74	74
0001ff20h:	65	6E	20	62	79	20	4B	65	76	69	6E	20	4C	61	77	74
0001ff30h:	6F	6E	20	26	20	74	68	65	20	42	6F	63	68	73	20	74
0001ff40h:	65	61	6D	2E	00	00	00	00	00	00	00	00	00	00	00	00
0001ff50h:	00	00	00	CF	BA	00	04	B8	AC	2A	EF	CF	00	00	00	00
0001ff60h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001ff70h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001ff80h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001ff90h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001ffa0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001ffb0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001ffc0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001ffd0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001ffe0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0001fff0h:	EA	5B	E0	00	F0	31	32	2F	32	30	2F	30	37	00	FC	47

Bochs for Windows - Console

```
E:\temp\bochs-2.3.6\obj-debug>bochs.exe -q -f bochsrc.bsrc
000000000000i(APIC?) local apic in initializing
*****
Bochs x86 Emulator 2.3.6
Build from CVS snapshot, on December 24, 2007
*****
000000000000i(      ) reading configuration from bochsrc.bsrc
000000000000i(      ) installing win32 module as the Bochs GUI
000000000000i(      ) using log file bochsout.txt
Next at t=0
(0) context not implemented because we HAVE HASH MAP=0
(0xffffffff) f000:ffff (unk. ctxt): jmp far f000:e05b      ; ea5be000f0
(bochs:1) > n
Next at t=1
(0) [0x000fe05b] f000:ffff (unk. ctxt): jmp far f000:e05b      ; 31c0
(bochs:2)
```

上电起始运行代码

BIOS模拟—rombios.c分析

□ 16位实模式程序

- ✓ Power-up Entry Point（系统上电入口）：0xffff0
- ✓ POST Entry Point（系统检测入口）：0xe05b
- ✓ INT 19H（系统引导入口）：0xe6f2 -> int19_relocated
-> int19_function
- ✓ INT 18H: int18_handler

嵌入式系统的Boot Loader程序（1/2）

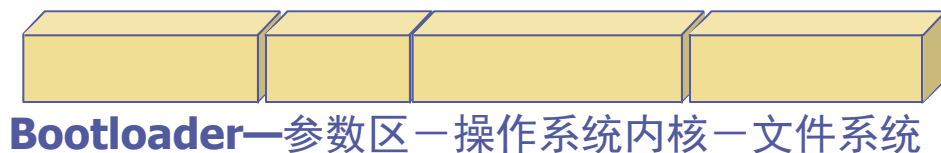
- 在嵌入式系统中没有BIOS那样的固件程序，因此，整个系统的加载启动任务完全由BootLoader来完成。
- 基于ARM内核的嵌入式系统，加电或复位的地址为0x00000000，Boot Loader程序的入口就安排在该地址上。

嵌入式系统的Boot Loader程序（2/2）

- Boot Loader的实现依赖于硬件环境
 - CPU体系结构：ARM、PPC、x86、MIPS
 - 板级设备：时钟、FLASH、通讯端口
 - Boot Loader的入口地址
 - Boot Loader与开发机的通讯机制
 - 串口
 - 网络
- 专用Boot Loader
- 通用Boot Loader：U-boot、Redboot

由Boot Loader启动操作系统的方式

- Flash启动方式



- 硬盘启动方式

- 在硬盘主引导区放置bootloader
- 从文件系统中引导操作系统

- 网络启动方式

- Bootloader放置在EPROM或Flash中
- 通过以太网远程下载操作系统内核或文件系统
- 开发板不需配置大的存储介质

Boot Loader的功能种类

- 简单Bootloader
 - 只具有系统引导功能
- 具有监控功能（Monitor）的Bootloader
 - 调试支持
 - 内存读写
 - Flash烧写
 - 网络下载
 - 环境变量配置

开放源码的Boot Loader程序

名称	支持体系结构	是否Monitor
LILO	X86	否
GRUB	X86	否
BLOB	ARM	否
Etherboot	X86	否
U—boot	X86、ARM、PPC	是
RedBoot	X86、ARM、PPC	是
VIVI	ARM	是

NP2108的例子

- <http://fm.zju.edu.cn/~wengkai/projects/2108/>
- Cirrus Logic 的EP7312为CPU，片外具有4MB的flash，16MB的SDRAM，一个10M的以太网口
- 2108上电以后，flash里的bootloader启动，把flash里的linux的kernel的压缩包解开释放到SDRAM中，再把flash里的ram disk的包解开到SDRAM中，最后启动Linux。整个过程大约需要90秒

电源5V, 内正

RESET按钮

网络灯

以太网

电话叉簧

接串口

状态灯

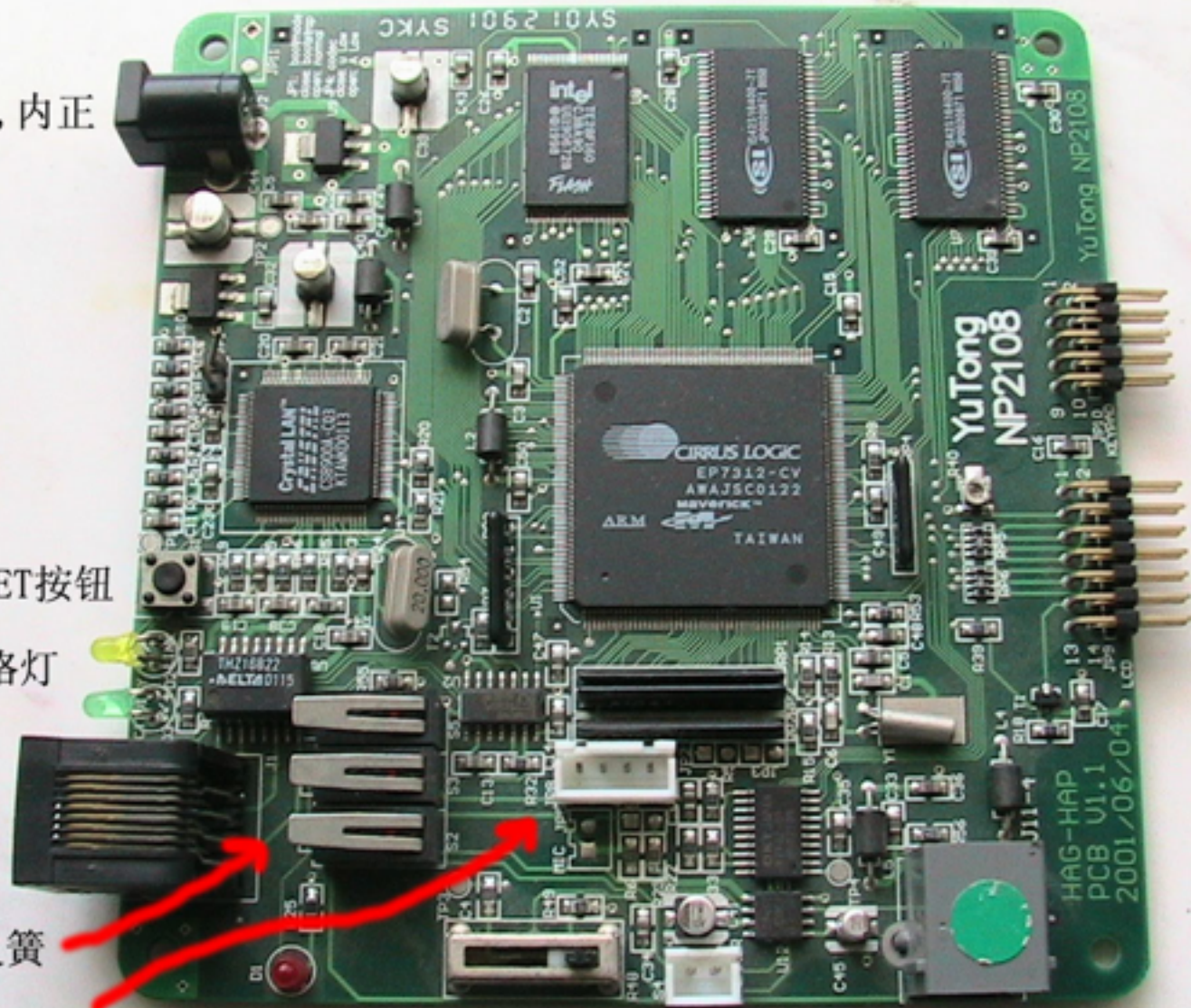
输出音量调节

接扬声器

接电话手柄

接键盘

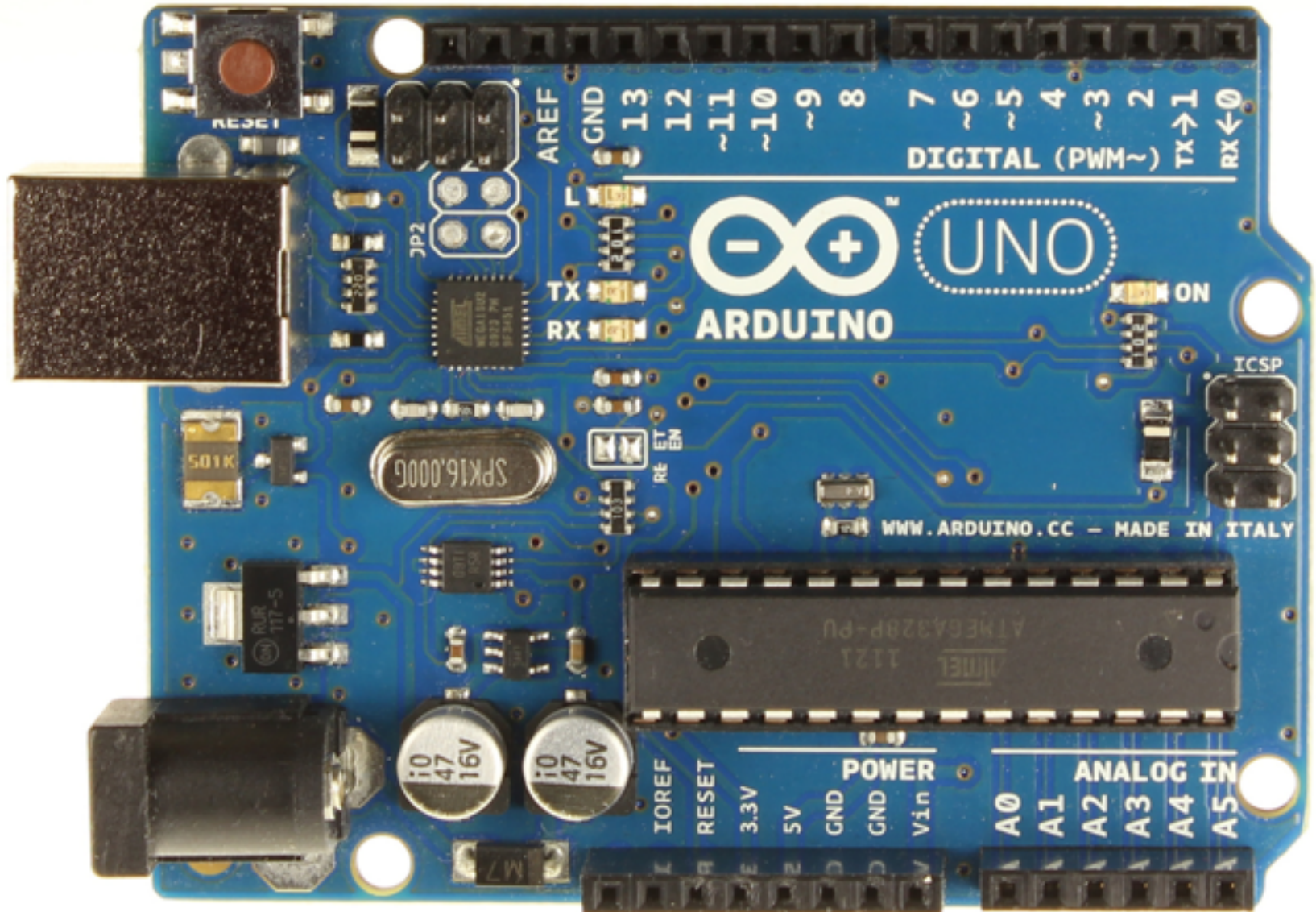
接LCD



EP7312

- 具有片内bootloader，通过跳线选择
- 这个bootloader通过串口接受一个很小的程序ramloader，放在片内SRAM执行
- 这个片内SRAM的程序初始化片外SDRAM，通过串口接受较大的程序（hermit），放在SDRAM执行
- hermit具有shell界面，可以查看/修改内存，接收程序烧录flash
- 由hermit烧录一个bootloader到flash，下次启动时跳线选择进flash
- 为了直接启动Linux，有两个版本的bootloader

Arduino



Arduino

- AVR单片机，8位，16MHz，片内flash/SRAM
- 不具有片内bootloader，需用ICP方式烧录程序
- Arduino bootloader是第三方bootloader，采用ICP烧录后，上电首先进入这个bootloader
- bootloader在串口等待2s，如果没有收到特殊字符，就转入用户程序，否则进入bootloader

Arduino

- 必须由上位机程序负责重启MCU，否则来不及进入bootloader
 - 串口的DTR信号经过一个电容接到MCU的reset引脚，DTR的变化形成一次reset
- 由于bootloader占据用户程序空间，链接时必须指定特殊的程序起始地址

课程大纲

 Bootloader程序的基本概念

 Bootloader典型框架结构

 S3C2410 Bootloader代码分析

 嵌入式软件开发环境建立实验

Boot Loader的典型框架结构

- Boot Loader的启动过程通常是多阶段的
 - 提供复杂的功能：突破引导扇区512字节限制
 - 提高代码移植性：高阶段代码采用高级语言
 - 提高运行速度：高阶段代码在内存中执行
- 大多数Boot Loader可分为阶段1和阶段2两大部分
 - 阶段1：实现依赖于CPU体系结构的代码
 - 阶段2：实现一些复杂的功能

Boot Loader阶段1介绍（1/4）

- Boot Loader的阶段1通常包括以下步骤：
- 1) 硬件设备初始化
 - 屏蔽所有的中断
 - 设置CPU的速度和时钟频率
 - RAM初始化
 - 初始化LED
 - 关闭CPU内部指令／数据Cache

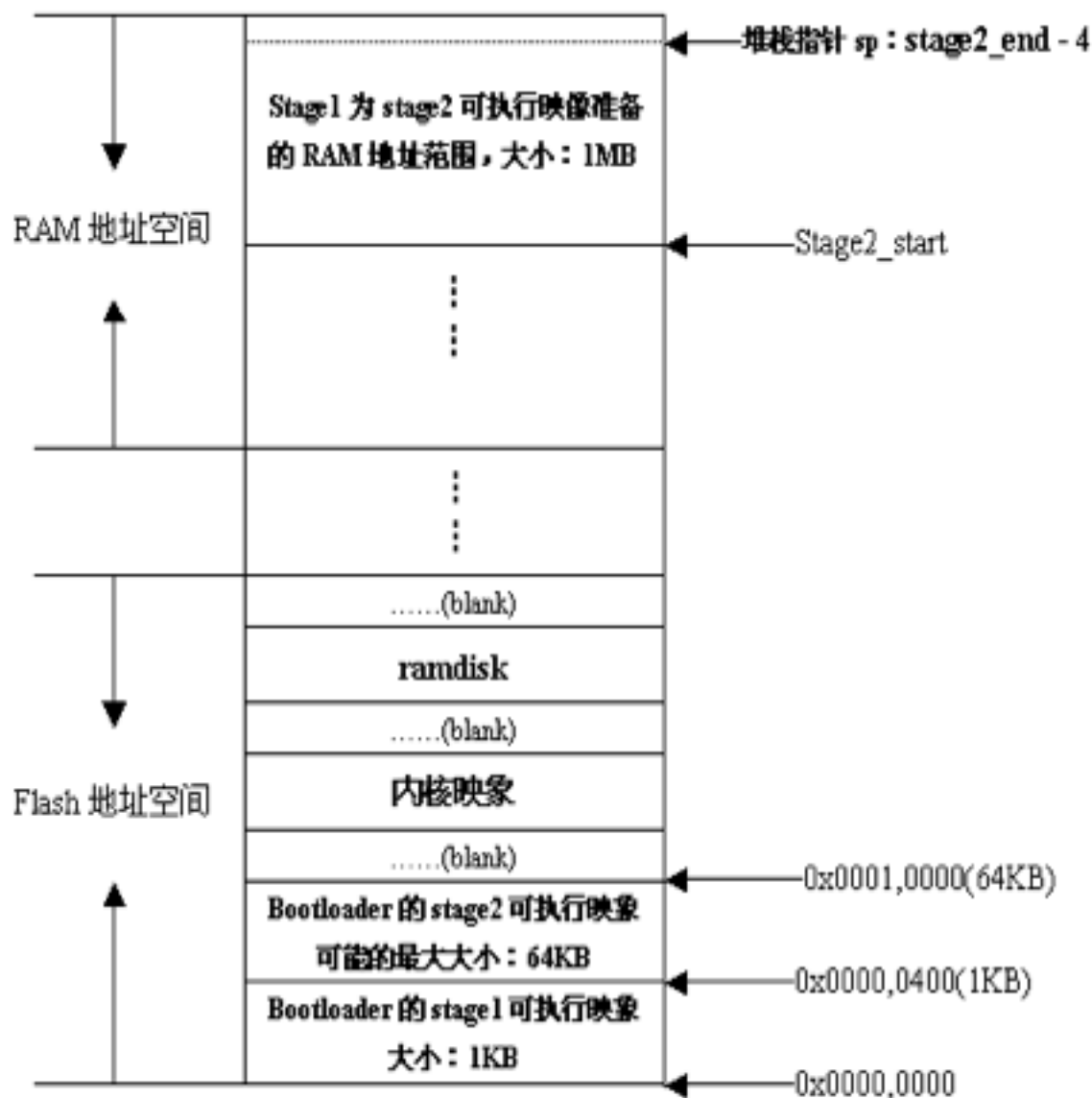
Boot Loader阶段1介绍 (2/4)

- 2) 为加载阶段2准备RAM空间
 - 除了阶段2可执行映象的大小外，还必须把堆栈空间也考虑进来
 - 必须确保所安排的地址范围的确是可读写的RAM空间
 - 内存区域有效性检测方法
 - 保存指定内存区域
 - 写入预定数据
 - 读入数据并比较
 - 恢复内存数据

Boot Loader阶段1介绍 (3/4)

- 3) 拷贝阶段2代码到RAM中
- 4) 设置堆栈指针sp
- 5) 跳转到阶段2的C语言入口点
- Boot Loader 的阶段1执行完成后的RAM 空间布局，如下图：

Boot Loader阶段1介绍 (4/4)



Boot Loader阶段2介绍（1/8）

- 1) 初始化本阶段要使用到的硬件设备
 - 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

Boot Loader阶段2介绍（2/8）

- 2) 检测系统的内存映射
 - 内存映射的描述
 - 可以用如下数据结构来描述RAM地址空间中一段连续的地址范围：

```
typedef struct memory_area_struct {  
    u32 start; /* 内存空间的基址 */  
    u32 size;  /* 内存空间的大小 */  
    int used;  
} memory_area_t;
```

- 内存映射的检测

Boot Loader阶段2介绍 (3/8)

- 连续内存区域探测——X86系统内存探测算法举例

```
for (p = (char *)0x100000; (int)p < 0x40000000; p += delta)
{
    for (ix = 0; ix < N_TIMES; ix++) /* 保存内存原有信息 */
    {
        temp[ix] = *((int *)p + ix);
        *((int *)p + ix) = TEST_PATTERN; /*TEST_PATTERN    0x12345678*/
    }
    cacheFlush (DATA_CACHE, p, 4 * sizeof(int));
    if (*(int *)p != TEST_PATTERN)/* 测试内存单元有效性 */
        p -= delta;
    for (ix = 0; ix < N_TIMES; ix++)/* 恢复内存原有信息 */
        *((int *)p + ix) = temp[ix];
}
```

Boot Loader阶段2介绍（4/8）

- 非连续内存区域探测——具有空洞内存的探测

from 内存区域低端 to 内存区域最高端，以页为单位进行内存探测

{

 对当前内存进行检测；

 if (当前内存页状态 与 前一内存页状态一致)

 合并内存状态情况；

 else

 记录当前内存区域状态。

}

Boot Loader阶段2介绍 (5/8)

- 3) 加载内核映像和根文件系统映像
 - 规划内存占用的布局
 - 内核映像所占用的内存范围
 - $\text{MEM_START} + 0\text{X}8000$
 - 根文件系统所占用的内存范围
 - $\text{MEM_START} + 0\text{X}00100000$
 - 从Flash上拷贝
 - While循环

Boot Loader阶段2介绍（6/8）

- 4) 设置内核的启动参数

- 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
- 嵌入式Linux系统中，通常需要由Boot Loader设置。常见启动参数有：ATAG_CORE、ATAG_MEM、ATAG_CMDLINE、ATAG_RAMDISK、ATAG_INITRD，以ATAG_NONE结束。

Boot Loader阶段2介绍（7/8）

- 例：设置ATAG_MEM的代码如下：

```
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);
```

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，计算出当前标记的下一个标记的起始地址。

Boot Loader阶段2介绍（8/8）

- 5) 调用内核
 - CPU寄存器的设置：
 - $R0=0$;
 - $R1$ = 机器类型ID; 关于机器类型号, 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;
 - CPU 模式:
 - 必须禁止中断 (IRQs和FIQs) ;
 - CPU必须SVC模式;
 - 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。

关于串口终端

- 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
- 如果碰到串口终端显示乱码或根本没有显示的问题，可能是因为：
 - 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

- Boot Loader 启动内核后却无法看到内核的启动输出信息：
 - 确认内核在编译时是否配置了对串口终端的支持，并配置了正确的串口驱动程序
 - Boot Loader 对串口的初始化设置是否和内核对串口的初始化设置一致
 - 还要确认 Boot Loader 所用的内核基地址必须和内核映像编译时所用的运行基地址一致

U-Boot

- <http://cnlearn.linksprite.com/?p=3074>



U-Boot介绍

- U-Boot全称Universal Boot Loader，遵循GPL协议，是由德国的工程师Wolfgang Denk从8XXROM代码发展而来的。
- 不仅支持Linux系统的引导，还支持NetBSD、VxWorks、QNX、RTEMS以及LynxOS嵌入式操作系统。
- 它支持很多处理器，比如PowerPC、ARM、MIPS和x86。

U-Boot特点（1）

- 支持SCC/FEC以太网、OFTP/TFTP引导、IP和MAC的预置功能。
- 在线读写Flash、DOC、IDE、IIC、EEROM、RTC。
- 支持串行口kermit和S-record下载代码。工具可以把ELF32格式的可执行文件转换为S-record格式，直接从串口下载并执行。
- 识别二进制、ELF32、ulmage格式的Image，对Linux引导有特别的支持。

U-Boot特点（2）

- 单任务软件运行环境。U-Boot可以动态加载和运行独立的应用程序。
- 监控命令集：读写I/O、内存、寄存器、内存、外设测试功能等。
- 脚本语言支持（类似bash脚本）。
- 支持WatchDog、LCD logo和状态指示功能等。
- 支持MTD和文件系统。
- 支持中断。
- 详细的开发文档。

代码结构分析（1）

- Board：和一些已有开发板相关的文件，比如 Makefile 和 u-boot.lds 等都和具体硬件有关。
- common：与体系结构无关的文件，实现各种命令的 C 文件。
- cpu：CPU 相关文件，其子目录都是以所支持的 CPU 为名，比如 arm926ejs、mips、mpc8260 和 nios 等，每个子目录中都包括 cpu.c 和 interrupt.c，start.S。
- disk：disk 驱动的分区处理代码。
- doc：文档。
- drivers：通用设备驱动程序。

代码结构分析（2）

- dtb: 数字温度测量器或传感器的驱动。
- examples: 一些独立运行的应用程序例子。
- fs: 支持文件系统的文件, U-Boot现在支持cramfs、fat、fdos、jffs2和registerfs。
- include: 头文件, 还有对各种硬件平台支持的汇编文件, 系统的配置文件和对文件系统支持的文件。
- net: 与网络有关的代码, BOOTP协议、TFTP协议、RARP协议和NFS文件系统的实现。
- lib_XXX: 与处理器体系结构相关的文件, 如lib_mips目录与MIPS体系结构相关, lib_arm目录与ARM相关。
- tools: 创建S-Record格式文件 和U-Bootimages的工具。