

目 录

第一章 ARM 体系结构和启动代码分析.....	1
1.1 ARM 介绍.....	1
1.2 ARM 处理器系列.....	2
1.2.1 ARM7 系列.....	2
1.2.2 ARM9 系列.....	2
1.2.3 ARM9E 微处理器系列.....	3
1.2.4 ARM10E 系列.....	3
1.2.5 SecurCore 系列.....	4
1.2.6 StrongARM 系列.....	4
1.2.7 Xscale.....	4
1.3 ARM 编程模型.....	4
1.3.1 工作状态及切换.....	5
1.3.2 存储器格式.....	5
1.3.3 处理器模式.....	6
1.3.4 寄存器.....	6
1.3.5 异常.....	9
1.4 Armlinux 启动分析.....	11
1.4.1 概述.....	11
1.4.2 ppcboot.....	11
1.4.3 head.S 和 head-s3c2410.S 分析.....	18
1.4.4 head-armv.S 分析.....	32
第二章 S3C2410 处理器特点和 HHARM9-EDU 软硬件系统描述.....	42
2.1 快速上手指南.....	42
2.2 目标板上的目录结构简单介绍.....	46
2.3 开发环境.....	47
2.3.1 操作系统的选择和安装.....	47
2.3.2 NFS 和 TFTP 服务器的配置.....	48
2.3.3 安装开发环境软件包.....	49
2.3.4 光盘目录介绍.....	49
2.3.5 HHARM9-EDU 目录结构介绍.....	50
2.3.6 内核编译.....	51
2.4 软件下载烧写说明.....	54
2.4.1 烧写过程.....	54
2.4.2 内核下载至 RAM 中直接启动.....	62
2.5 HHARM9-EDU 平台硬件系统.....	63
2.5.1 核心板功能模块结构图.....	64
2.5.2 内存部分的构成.....	66
2.5.3 片选.....	66
2.5.4 中断.....	67
2.5.5 GPIO.....	69
2.5.6 总线.....	69
2.5.7 外设接口图.....	70
2.5.8 接口管脚说明.....	71
第三章 实验指导.....	77
实验一：嵌入式应用程序开发入门.....	77
实验二：WEB SERVER/CGI 实验.....	86
实验三：应用程序移植实验.....	99
实验四：框架型驱动实验.....	120
实验五：串口通信实验.....	129
实验六：中断实验.....	137

第一章 ARM 体系结构和启动代码分析

1.1 ARM 介绍

ARM 即 Advanced RISC Machines 的缩写，既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司，作为知识产权供应商，本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片进入市场。目前，全世界有几十家大的半导体公司都使用 ARM 公司的授权，因此既使得 ARM 技术获得更多的第三方工具、制造、软件的支持，又使整个系统成本降低，使产品更容易进入市场被消费者所接受，更具有竞争力。

采用 ARM 技术知识产权（IP）核的微处理器，即我们通常所说的 ARM 微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 75% 以上的市场份额，ARM 技术正在逐步渗入到我们生活的各个方面。采用 RISC 架构的 ARM 微处理器一般具有三大特点：

- (1) 小体积，低功耗，低成本，高性能
- (2) 支持 Thumb（16 位）/ARM（32 位）双指令集
- (3) 全球众多的合作伙伴

传统的 CISC（Complex Instruction Set Computer，复杂指令集计算机）结构有其固有的缺点，随着计算机技术的发展而不断引入新的复杂的指令集，为支持这些新增的指令，计算机的体系结构会越来越复杂，然而，在 CISC 指令集的各种指令中，其使用频率却相差悬殊，大约有 20% 的指令会被反复使用，占整个程序代码的 80%。而余下的 80% 的指令却不经常使用，在程序设计中只占 20%，显然，这种结构是不太合理的。

基于以上的不合理性，1979 年美国加州大学伯克利分校提出了 RISC（Reduced Instruction Set Computer，精简指令集计算机）的概念，RISC 并非只是简单地去减少指令，而是把着眼点放在了如何使计算机的结构更加简单合理地提高运算速度上。RISC 结构优先选取使用频最高的简单指令，避免复杂指令；将指令长度固定，指令格式和寻址方式种类减少；以控制逻辑为主，不用或少用微码控制等措施来达到上述目的。

到目前为止，RISC 体系结构也还没有严格的定义，一般认为，RISC 体系结构应具有如下特点：

- (1) 采用固定长度的指令格式，指令归整、简单、基本寻址方式有 2~3 种。
- (2) 使用单周期指令，便于流水线操作执行。
- (3) 大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/存储指令可以访问存储器，以提高指令的执行效率。

当然，和 CISC 架构相比较，尽管 RISC 架构有上述的优点，但决不能认为 RISC 架构就可以取代 CISC 架构，事实上，RISC 和 CISC 各有优势，而且界限并不那么明显。现代的 CPU 往往采用 CISC 的外围，内部加入了 RISC 的特性，如超长指令集 CPU 就是融合了 RISC 和 CISC 的优势。

ARM 体系结构继承了 RISC 结构。只有加载和存储指令可以访问存储器，数据处理指令只对寄存器的内容进行操作。同时 ARM 体系舍弃了 RISC 的寄存器窗口，延迟转移和所有单指令周期。ARM 体系结构目前被公认为是业界领先的 32 位嵌入式 RISC 微处理器结构。所有 ARM 处理器共享这一体系结构。因而确保了开发者转向更高性能的 ARM 处理器时，在软件开发上可以得到最大的回报。

1.2 ARM 处理器系列

ARM 处理器目前包括以下几个系列：ARM7 系列，ARM9 系列，ARM9E 系列，ARM10E 系列，SecurCore 系列，Intel 的 Xscale 和 StrongARM 等等。每一个系列的 ARM 微处理器都有各自的特点和应用领域。所有系列中，ARM7、ARM9、ARM9E 和 ARM10E 为 4 个通用处理器系列，每一个系列提供一套相对独特的性能来满足不同应用领域的需求。SecurCore 系列专门为安全要求较高的应用而设计。Xscale 系列性能高达 1200MIPS，功耗测量为 $\mu\text{W}/\text{MHz}$ ，并且所有体系结构兼容。

下面就详细介绍一下各种处理器的特点及应用领域。

1.2.1 ARM7 系列

ARM7 系列微处理器为低功耗的 32 位 RISC 处理器，最适合用于对价位和功耗要求较高的消费类应用。ARM7 微处理器系列具有如下特点：

- (1) 具有嵌入式 ICE—RT 逻辑，调试开发方便
- (2) 非常低的功耗，适合对功耗要求较高的应用
- (3) 能够提供 0.9MIPS/MHz 的三级流水线结构
- (4) 代码密度高并兼容 16 位的 Thumb 指令集。

ARM7 系列微处理器的主要应用领域为：工业控制、Internet 设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。

ARM7 系列微处理器包括如下几种类型的核：ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ。其中，ARM7TMDI 是目前使用最广泛的 32 位嵌入式 RISC 处理器，是目前用于低端的 ARM 处理器核，且应用范围很广。ARM7TDMI-S 最适合于可移植性和灵活性为关键的设计。ARM720T 是全性能的 MMU，最适合低功耗和体积为关键的应用。ARM7EJ 是 Jazelle 和 DSP 指令集的最小及最低功耗的实现。

TDMI 的基本含义为：

- T：支持 16 位压缩指令集 Thumb；
 - D：支持片上 Debug；
 - M：增强型内嵌硬件乘法器（Multiplier）
 - I：嵌入式 ICE 硬件支持片上断点和调试点；
- Samsung 公司的 S3C4510B 即属于该系列的处理器。

1.2.2 ARM9 系列

ARM9 系列微处理器在高性能和低功耗特性方面提供最佳的性能。具有以下特点：

- (1) 5 级整数流水线，指令执行效率更高。
- (2) 提供 1.1MIPS/MHz 的哈佛结构。
- (3) 支持 32 位的高速 AMBA 总线接口。
- (5) 全性能的 MMU，支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。
- (6) MPU 支持实时操作系统。
- (7) 支持数据 Cache 和指令 Cache，具有更高的指令和数据处理能力。

ARM9 系列微处理器主要应用于无线设备、仪器仪表、安全系统、机顶盒、高端打印机、数字

照相机和数字摄像机等。

ARM9 系列微处理器包含 ARM920T, ARM922T 和 ARM940T, 以适用于不同的应用场合。ARM920T 和 ARM922T 采用双 8KB 的 cache, 支持全性能的 MMU。ARM940T 采用双 4KB 的 cache, 内置保护单元。它们为要求虚拟存储管理和复杂内存保护提供了一个高性能的处理器方案, 可以用于高性能无线应用, 网络, 图像, 音视频编解码等方面。

实验平台的 Samsung 公司的 S3C2410 即属于该系列的处理器。

1.2.3 ARM9E 微处理器系列

ARM9E 系列微处理器为可综合处理器, 使用单一的处理器内核提供了微控制器、DSP、Java 应用系统的解决方案, 极大的减少了芯片的面积和系统的复杂程度。

ARM9E 系列微处理器的主要特点如下:

- (1) 支持 DSP 指令集, 适合于需要高速数字信号处理的场合。
- (2) 5 级整数流水线, 指令执行效率更高。
- (3) 支持 32 位的高速 AMBA 总线接口。
- (4) 全性能的 MMU, 支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。
- (5) MPU 支持实时操作系统。
- (6) 支持数据 Cache 和指令 Cache, 具有更高的指令和数据处理能力。
- (7) 支持 VFP9 浮点处理协处理器。

ARM9E 系列广泛用于硬盘驱动和 DVD 播放器等海量存储设备, 语音编码器, 调制解调器, 语音识别和合成以及自动控制解决方案。

ARM9E 系列包含 ARM926EJ-S、ARM946E-S 和 ARM966E-S 三种类型, 以适用于不同的应用场合。

1.2.4 ARM10E 系列

ARM10E 系列微处理器采用了新的体系结构, 与同等的 ARM9 器件相比较, 在同样的时钟速度下, 性能提高了近 50%, 同时, ARM10E 系列微处理器采用了两种先进的节能方式, 使其功耗极低。

ARM10E 系列微处理器的主要特点如下:

- (1) 支持 DSP 指令集, 适合于需要高速数字信号处理的场合。
- (2) 6 级整数流水线。
- (3) 支持 32 位的高速 AMBA 总线接口。
- (4) 支持 VFP10 浮点处理协处理器。
- (5) 全性能的 MMU, 支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。
- (6) 支持数据 Cache 和指令 Cache, 具有更高的指令和数据处理能力。
- (7) 内嵌并行读/写操作部件。

ARM10E 系列微处理器专为数字机顶盒, 智能电话等高效手提设备而设计, 并为复杂的视频游戏机和高性能打印机提供高效的整数和浮点运算能力。

ARM10E 系列微处理器包含 ARM1020E、ARM1022E 和 ARM1026EJ-S 三种类型。

1.2.5 SecurCore 系列

SecurCore 系列微处理器专为安全需要而设计，具有特定的抗篡改和反工程特性。它除了具有 ARM 体系结构各种主要特点外，还在系统安全方面具有如下的特点：

- (1) 带有灵活的保护单元，以确保操作系统和应用数据的安全。
- (2) 采用软内核技术，防止外部对其进行扫描探测。
- (3) 可集成用户自己的安全特性和其他协处理器。

SecurCore 系列微处理器主要应用于对安全性要求较高的应用。包含 SecurCore SC100、SecurCore SC110、SecurCore SC200 和 SecurCore SC210 四种类型。

1.2.6 StrongARM 系列

Inter StrongARM SA-1100 处理器是采用 ARM 体系结构高度集成的 32 位 RISC 微处理器。它融合了 Inter 公司的设计和处理技术以及 ARM 体系结构的电源效率，采用在软件上兼容 ARMv4 体系结构、同时采用具有 Intel 技术优点的体系结构。

Intel StrongARM 处理器是便携式通讯产品和消费类电子产品的理想选择。

1.2.7 Xscale

Xscale 处理器是一款全性能、高性价比、低功耗的处理器。它支持 16 位的 Thumb 指令和 DSP 扩充，适合于数字移动电话、个人数字助理和网络产品等场合。

ARM7 处理器曾得到了最广泛的应用，采用 ARM7 处理器作为内核生产芯片的公司很多。但是 ARM7 没有集成 USB，在 USB 越来越重要的今天，这是一个很大的缺陷。而 ARM9 系列是高性能和低功耗方面最佳的硬宏单元，越来越多的公司和客户转向了 ARM9，ARM9 已经代替了 ARM7，成为了主流芯片。我们的实验平台的 S3C2410 使用的是 ARM920T 处理器。因此下面主要以 ARM920T 处理器为例，对 ARM 的体系结构进行介绍。

1.3 ARM 编程模型

首先对字，半字和字节的概念作一个说明：

- (1) 字 (Word)：在 ARM 体系结构中，字的长度为 32 位，而在 8 位/16 位处理器体系结构中，字的长度一般为 16 位，请注意区分。字必须 4 字节边界对齐。
- (2) 半字 (Half-Word)：在 ARM 体系结构中，半字的长度为 16 位，与 8 位/16 位处理器体系结构中字的长度一致。半字必须 2 字节边界对齐。
- (3) 字节 (Byte)：在 ARM 体系结构和 8 位/16 位处理器体系结构中，字节的长度均为 8 位。

目前 ARM 体系结构支持所有的这三种数据类型。当这些数据类型的任意一种被说明成 unsigned 类型时，N 位数据值的表示范围是 $0 \sim 2^N - 1$ 的非负整数，使用通常的二进制格式；当这些数据类型的任意一种被说明成 signed 类型时，N 位数据值的表示范围是 $-2^{N-1} \sim 2^{N-1} - 1$ 的整数，使用二进制的补码格式。

1.3.1 工作状态及切换

从编程的角度看，ARM 微处理器的工作状态一般有两种，并可在两种状态之间切换：

- (1) ARM 状态，此时处理器执行 32 位的字对齐的 ARM 指令；
- (2) Thumb 状态，此时处理器执行 16 位的、半字对齐的 Thumb 指令。

ARM 处理器可以在这两种工作状态之间切换。ARM 指令集和 Thumb 指令集均有切换处理器状态的指令，并可在两种工作状态之间切换，但 ARM 微处理器在开始执行代码时，应该处于 ARM 状态。状态切换方法为：

- (1) 进入 Thumb 状态：当操作数寄存器的状态位（位[0]）为 1 时，执行 BX 指令就可以进入 Thumb 状态。如果处理器处于 Thumb 状态时发生异常，那么当异常处理（IRQ,FIQ,UNDEF,ABORT,SWI 等）返回时，自动切换回 Thumb 状态。
- (2) 进入 ARM 状态：当操作数寄存器的状态位（位[0]）为 0 时，执行 BX 指令就可以进入 ARM 状态。处理器在进行异常处理（IRQ,FIQ,UNDEF,ABORT,SWI 等）时，把 PC 放入异常模式链接寄存器中，从异常地址开始执行也可以进入 ARM 状态。

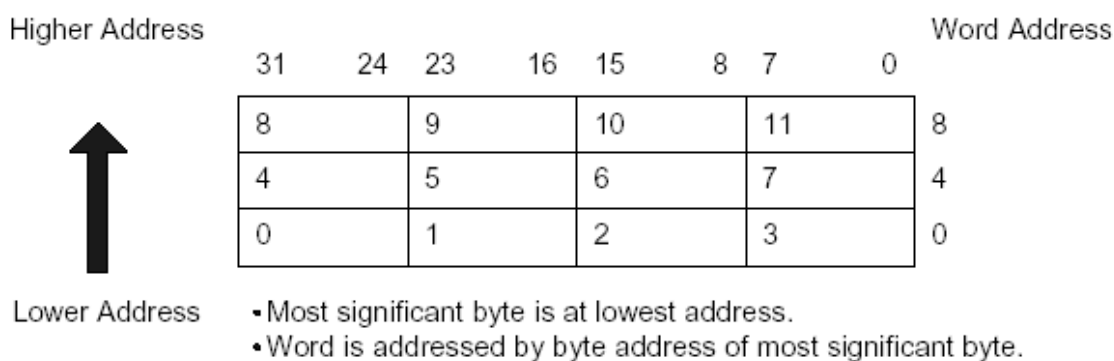
注意的是，两种状态的切换并不影响处理器模式和寄存器内容。

1.3.2 存储器格式

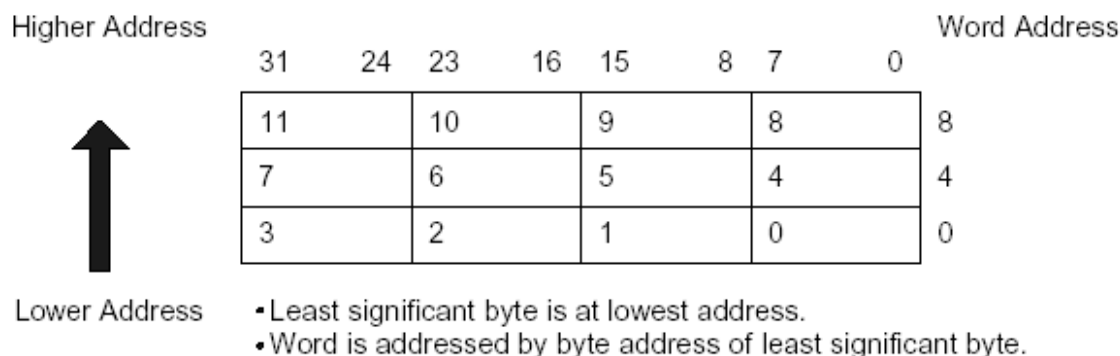
ARM 体系结构将存储器看作是从零地址开始的字节的线性组合，使用 2^{32} 个 8 位字节(4GB)的单一线性地址空间。将字节地址作为无符号数看待。将地址空间看作是 2^{30} 个 32 位的字组成。每个字的地址是字对准的，字对准地址是 A 的字由地址是 A，A+1，A+2，A+3 的 4 字节组成。当地址空间看作是 2^{31} 个 16 位的半字组成。每个半字的地址是半字对准的，半字对准地址是 A 的半字由地址是 A，A+1 的 2 字节组成。

ARM 体系结构可以用两种方法存储字数据，称之为大端格式和小端格式。

- (1) 大端格式：字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中。



- (2) 小端格式：与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节。



小端格式通常是 ARM 处理器的缺省格式。

1.3.3 处理器模式

ARM 微处理器支持 7 种运行模式，分别为：

- (1) 用户模式（usr）：ARM 处理器正常的程序执行状态
- (2) 快速中断模式（fiq）：用于高速数据传输或通道处理
- (3) 外部中断模式（irq）：用于通用的中断处理
- (4) 管理模式（svc）：操作系统使用的保护模式，系统复位后的缺省模式
- (5) 指令终止模式（abt）：当指令预取终止时进入该模式。
- (6) 数据访问终止模式（abt）：当数据访问终止时进入该模式，可用于虚拟存储及存储保护。
- (7) 系统模式（sys）：运行具有特权的操作系统任务。

在软件控制下可以改变运行模式，外部中断或异常处理也可以引起模式发生改变。

大多数应用程序都在用户模式下执行。当处理器工作在用户模式时，正在执行的程序不能访问某些被保护的资源，也不能改变模式，除非异常发生。

除用户模式的其他 6 种模式称为特权模式或非用户模式。它们可以自由的访问系统资源和改变模式。其中除去用户模式和系统模式以外的 5 种又称为异常模式，即 FIQ，IRQ，管理，终止，数据访问终止模式。

1.3.4 寄存器

ARM 总共有 37 个 32 位的寄存器：

- (1) 31 个通用寄存器，包括程序计数器(PC)，这些寄存器是 32 位的
- (2) 6 个状态寄存器，这些寄存器也是 32 位的，但是只使用其中的 12 位

但是这些寄存器并不是同时可用的，哪些对程序员可见是依赖于处理器状态和操作模式的。但在任何时候，通用寄存器 R14~R0、程序计数器 PC、一个或两个状态寄存器都是可访问的。

(1) ARM 状态的寄存器组织

16 个通用寄存器和 1~2 个状态寄存器。通用寄存器包括 R0~R15，可以分为三类：


- (a) 未分组(Unbanked)寄存器 R0~R7；
- (b) 分组(Banked)寄存器 R8~R14
- (c) 程序计数器 PC(R15)

ARM状态下的通用寄存器与程序计数器

System & User	FIQ	Supervisor	About	IRG	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM状态下的程序状态寄存器

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

 = 分组寄存器

(2) Thumb 状态的寄存器组织


Thumb 状态的可见寄存器集合是 ARM 状态下寄存器集合的子集。程序可以直接访问 8 个通用寄存器(R0~R7), PC, SP, LR 和 CPSR。此状态下寄存器 R8~R15(高寄存器)并不是标准寄存器的一部分,但是可以将其用作快速暂存存储器。

Thumb状态下的通用寄存器与程序计数器

System & User	FIQ	Supervisor	About	IRG	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
SP	SP_fiq	SP_svg	SP_abt	SP_irq	SP_und
LR	LR_fiq	LR_svc	LR_abt	LR_irq	LR_und
PC	PC	PC	PC	PC	PC

Thumb状态下的程序状态寄存器

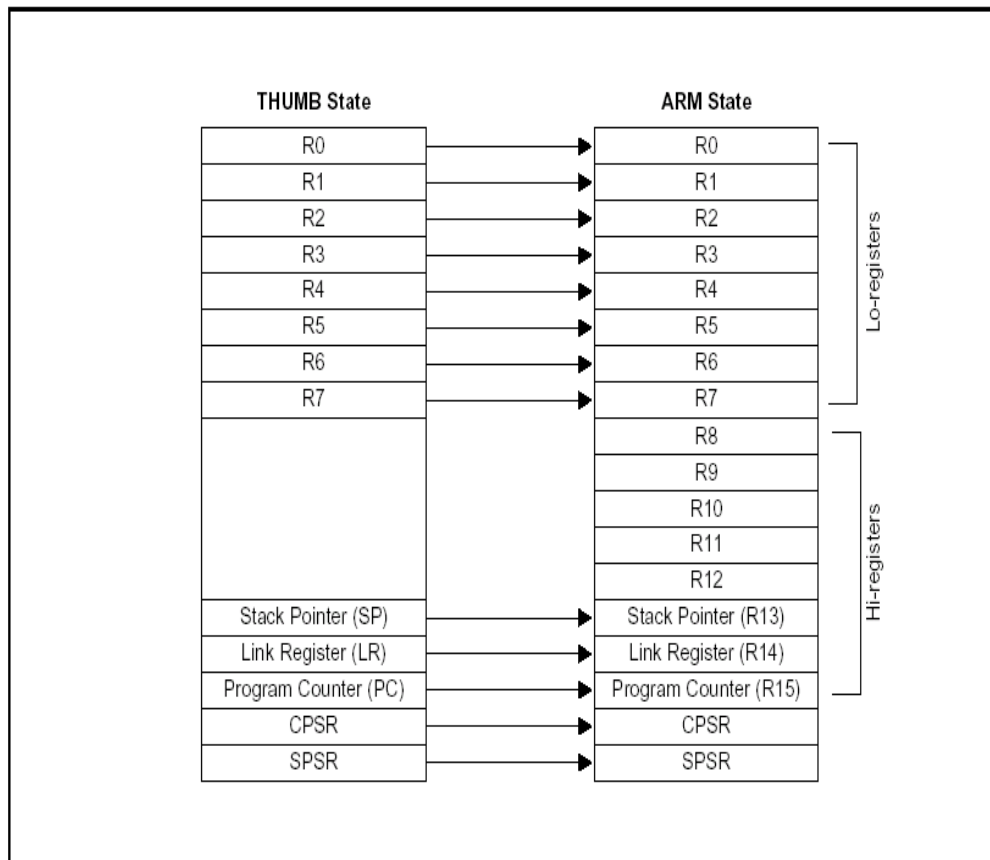
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

 = 分组寄存器

ARM 和 Thumb 寄存器间的关系

(a) Thumb 状态的 R0~R7 与 ARM 状态的 R0~R7 是一致的

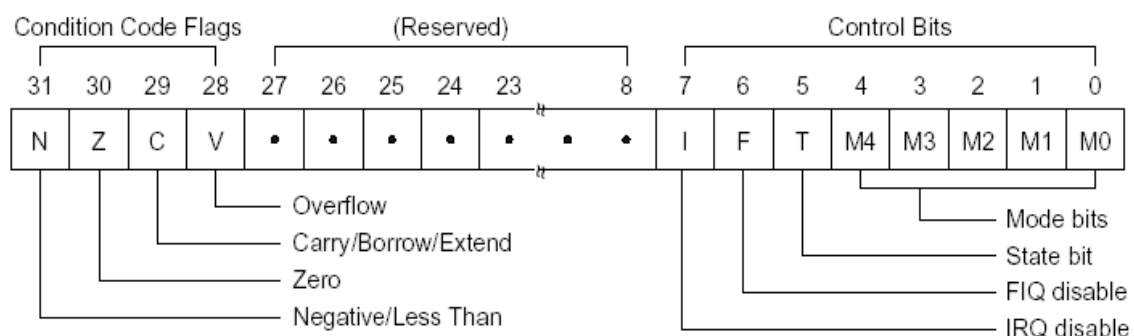
- (b) Thumb 状态的 CPSR 和 SPSR 与 ARM 状态的 CPSR 和 SPSR 是一致的
- (c) Thumb 状态的 SP 映射到 ARM 状态的 R13
- (d) Thumb 状态的 LR 映射到 ARM 状态的 R14
- (e) Thumb 状态的 PC 映射到 ARM 状态的 PC(R15)



(3) 程序状态寄存器

ARM 含有 CPSR 和 5 个 SPSR 状态寄存器。CPSR(Current Program Status Register, 当前程序状态寄存器), CPSR 可在任何运行模式下被访问。每一种异常模式下又都有一个专用的物理状态寄存器, 称为 SPSR (Saved Program Status Register, 备份的程序状态寄存器)。当异常出现时, SPSR 用于保留 CPSR 的状态。

程序状态寄存器的格式如下:



在ARM状态下, 绝大多数的指令都是有条件执行的。在Thumb状态下, 仅有分支指令是有条件执行的。条件码标志如下:

- (1) 条件码标志：N，Z，C，V 和 Q。
- (2) 控制位：最低 8 位 I，F，T，M[4:0]用作控制位。当异常出现时改变。
- (3) 其他位：保留，用作以后的扩展。

标志位	含 义
N	当用两个补码表示的带符号数进行运算时，N=1 表示运算的结果为负数；N=0 表示运算的结果为正数或零；
Z	Z=1 表示运算的结果为零；Z=0 表示运算的结果为非零；
C	可以有 4 种方法设置 C 的值： (1) 加法运算（包括比较指令 CMN）：当运算结果产生了进位时（无符号数溢出），C=1，否则 C=0。 (2) 减法运算（包括比较指令 CMP）：当运算时产生了借位（无符号数溢出），C=0，否则 C=1。 (3) 对于包含移位操作的非加/减运算指令，C 为移出值的最后一位。 (4) 对于其他的非加/减运算指令，C 的值通常不改变。
V	可以有 2 种方法设置 V 的值： — 对于加/减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1 表示符号位溢出。 — 对于其他的非加/减运算指令，C 的值通常不改变。
Q	在 ARM v5 及以上版本的 E 系列处理器中，用 Q 标志位指示增强的 DSP 运算指令是否发生了溢出。在其他版本的处理器中，Q 标志位无定义。

1.3.5 异常

当正常的程序执行流程发生暂时的停止或改变时，称之为异常，例如处理一个外部的中断请求。处理异常前，处理器状态必须保留，以便在异常处理程序完成后，原来的程序能够重新执行。同一时刻可以出现多个异常。

(1) ARM 支持七种类型的异常：

- (a) 复位：当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处理程序处执行。
- (b) 未定义指令：当 ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常。采用这种机制，可以通过软件仿真扩展 ARM 或 Thumb 指令集。
- (c) 软件中断：该异常由执行 SWI 指令产生，可用于用户模式下的程序调用特权操作指令。可使用该异常机制实现系统功能调用。
- (d) 指令预取中止：若处理器预取指令的地址不存在，或该地址不允许当前指令访问，存储器会向处理器发出中止信号，预取的指令被记为无效，但只有当处理器试图执行无效指令时，指令预取中止异常才会发生，如果指令未被执行，例如在指令流水线中发生了跳转，则预取指令中止不会发生。
- (e) 数据中止：若处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常。若数据中止发生，系统的响应与指令的类型有关。
- (f) IRQ（外部中断请求）：当处理器的外部中断请求引脚有效，且 CPSR 中的 I 位为 0 时，产生 IRQ 异常。系统的外设可通过该异常请求中断服务。
- (g) FIQ（快速中断请求）：当处理器的快速中断请求引脚有效，且 CPSR 中的 F 位为 0 时，产生 FIQ 异常。FIQ 异常是为了支持数据传输或者通道处理而设计的。在 ARM 状态下，系统有足够的私有寄存器，从而可以避免对寄存器保存的需求，并减小了系统上下文切换的开销。若将

CPSR 的 F 位置为 1，则会禁止 FIQ 中断，若将 CPSR 的 F 位清零，处理器会在指令执行时检查 FIQ 的输入。注意只有在特权模式下才能改变 F 位的状态。

(2) 异常的进入

- (a) 将下一条指令的地址存入相应连接寄存器 LR，以便程序在处理异常返回时能从正确的位置重新开始执行。
- (b) 将 CPSR 复制到相应的 SPSR 中。
- (c) 根据异常类型，强制设置 CPSR 的运行模式位。
- (d) 强制 PC 从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序。也可以设置中断禁止位来阻止其他无法处理的异常嵌套。

如果在异常发生时处理器是在 Thumb 状态下，那么当中断异常向量地址加载 PC 时，自动切换进入 ARM 状态。

(3) 从异常返回

异常处理完毕之后，ARM 微处理器会执行以下几步操作从异常返回：

- (a) 将连接寄存器 LR 的值减去相应的偏移量后送到 PC 中。
- (b) 将 SPSR 复制回 CPSR 中。
- (c) 如果进入时设置了中断禁止位，那么清除该标志。

可以认为应用程序总是从复位异常处理程序开始执行的，因此复位异常处理程序不需要返回。

(4) 异常向量

地址	异常	进入模式
0x0000,0000	复位	管理模式
0x0000,0004	未定义指令	未定义模式
0x0000,0008	软件中断	管理模式
0x0000,000C	终止（预取指令）	终止模式
0x0000,0010	终止（数据）	终止模式
0x0000,0014	保留	保留模式
0x0000,0018	IRQ	IRQ 模式
0x0000,001C	FIQ	FIQ 模式

(5) 异常优先级

当同时出现多个中断时，系统按照固定优先级处理它们。异常优先级由高到低的排列次序如下表：

优先级	异 常
1（最高）	复位
2	数据中止
3	FIQ
4	IRQ
5	预取指令中止
6（最低）	未定义指令、SWI

1.4 Armlinux 启动分析

1.4.1 概述

在内核运行之前需要系统引导程序(ppcboot)完成加载内核和一些辅助性的工作，然后跳转到内核代码的起始地址并执行。本文先分析了 ppcboot 的初始化工作，接着从内核镜像的起始地址进行分析。整个 arm linux 内核的启动可分为三个阶段：第一阶段主要是进行 cpu 和体系结构的检查、cpu 本身的初始化以及页表的建立等；第二阶段主要是对系统中的一些基础设施进行初始化；最后则是更高层次的初始化，如根设备和外部设备的初始化。

1.4.2 ppcboot

1.4.2.1 简介

本处介绍主要来自内核源代码下的/HHARM9-EDU/ppcboot-2.0.0/文件，适合于 arm linux 2.4.18-rmk6 及以上版本。

ppcboot 主要作用是初始化一些必要的设备，然后调用内核，同时传递参数给内核。主要完成如下工作：

- (1) 建立和初始化 RAM。
- (2) 初始化一个串口。
- (3) 检测机器的系统结构。
- (4) 建立内核的 tagged list。
- (5) 调用内核镜像。

1.4.2.2 功能详细介绍

- (1) 建立和初始化 RAM。
要求：必须
功能：探测所有的 RAM 位置和大小，并对 RAM 进行初始化。
- (2) 初始化一个串口。
要求：可选，建议
功能：Ppcboot 应该初始化并启动一个串口。这可以让内核的串口驱动自动探测哪个串口作为内核的控制台。另外也可以通过给内核传递“console=”参数完成此工作。
- (3) 检测机器的系统结构。
要求：必须
功能：Ppcboot 应该通过某种方法探测机器类型，最后传递给内核一个 MACH_TYPE_xxx 值，这些值参看 HHARM9-EDU/kernel/arch/arm/tools/mach-types。
- (4) 建立内核的 tagged list。
要求：必须
功能：Ppcboot 必须创建和初始化内核的 tagged list。一个合法的 tagged list 开始于

ATAG_CORE 并结束于 ATAG_NONE。ATAG_CORE tag 可以为空。一个空的 ATAG_CORE tag 的 size 字段设为“2”(0x00000002)。ATAG_NONE 的 size 字段必须设为“0”。tagged list 可以有任意多的 tag。Ppcboot 必须至少传递系统内存的大小和位置，以及根文件系统的位置，一个最小化的 tagged list 应该像如下：

```

+-----+
base -> | ATAG_CORE | |
+-----+ |
| ATAG_MEM | | increasing address
+-----+ |
| ATAG_NONE | |
+-----+ v

```

tagged list 应该放在内核解压时和 initrd 的“bootp”程序都不会覆盖的内存区域。建议放在 RAM 的起始的 16K 大小的地方。

(5) 调用内核镜像。

要求：必须

功能：可以从 flash 调用内核，也可以从系统 RAM 中调用内核。对于后者需要注意，内核使用内核镜像以下的 16K 内存作为页表，建议把内核起始放在 RAM 的 32K 处。无论是哪种方法，如下条件必须满足：

- CPU register settings

r0 = 0,

r1 = machine type number discovered in (3) above.

r2 = physical address of tagged list in system RAM.

- CPU mode

All forms of interrupts must be disabled (IRQs and FIQs)

The CPU must be in SVC mode. (A special exception exists for Angel)

- Caches, MMUs

The MMU must be off.

Instruction cache may be on or off.

Data cache must be off.

- The boot loader is expected to call the kernel image by jumping directly to the first instruction of the kernel image.

1.4.2.3 Start.S 代码分析

Ppcboot 烧在 INTEL 28J128A NOR FLASH 的最前面，所以 S3C2410 上电就从 ppcboot 的最开始的代码开始执行，这个代码就是 /HHARM9-EDU/ppcboot-2.0.0/cpu/arm920t/Start.S。

```

-----
.globl _start
_start:  b      reset      //上电复位向量，PC 指针自动被填写为 reset 的
                           //地址，这个地址从反汇编代码可以看到它的绝
                           //对地址 33f00054 <reset>

ldr  pc, _undefined_instruction
-----

```

```

ldr pc, _software_interrupt
ldr pc, _prefetch_abort
ldr pc, _data_abort
ldr pc, _not_used
ldr pc, _irq
ldr pc, _fiq

_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
.....

```

//下面才是真正的上电复位的启动代码:

```

/*
 * the actual reset code
 */
reset:

/* turn off the watchdog */
#ifdef CONFIG_S3C2400
#define pWTCON 0x15300000
/* Interrupt-Controller base addresses */
#define INTMSK 0x14400008
/* clock divisor register */
#define CLKDIVN 0x14800014
#elif defined(CONFIG_S3C2410)
#define pWTCON 0x53000000
/* Interrupt-Controller base addresses */
#define INTMSK 0x4A000008
#define INTSUBMSK 0x4A00001C
/* clock divisor register */
#define CLKDIVN 0x4C000014
#define MPLLCON 0x4C000004
#define CLK_CTL_BASE 0x4C000000
#endif

ldr r0, =pWTCON
mov r1, #0x0
str r1, [r0]

```

```

/*
 * mask all IRQs by setting all bits in the INTMR - default
 */
mov r1, #0xffffffff
ldr    r0, =INTMSK
str    r1, [r0]
#if defined(CONFIG_S3C2410)
ldr    r1, =0x7ff
ldr    r0, =INTSUBMSK
str    r1, [r0]
#endif

@ initialise system clocks
mov    r1, #CLK_CTL_BASE
mvn    r2, #0xff000000
str    r2, [r1, #0x0] /*oLOCKTIME*/

@ldr r2, mpll_50mhz
@str    r2, [r1, #0x4] /*oMPLLCON*/

mov    r1, #CLK_CTL_BASE

mov    r2, #0x3
str    r2, [r1, #0x14] /*oCLKDIVN*/
mrc    p15, 0, r1, c1, c0, 0      @ read ctrl register
orr    r1, r1, #0xc0000000        @ Asynchronous
mcr    p15, 0, r1, c1, c0, 0      @ write ctrl register

@ now, CPU clock is 200 Mhz
mov    r1, #CLK_CTL_BASE
ldr    r2, mpll_200mhz
str    r2, [r1, #0x4] /*oMPLLCON*/

/*
 * we do sys-critical inits only at reboot,
 * not when booting from ram!
 */
#ifdef CONFIG_INIT_CRITICAL
bl    cpu_init_crit    //这里调用 bl memsetup 初始化 SDRAM，具体实现
                        //在/HHARM9-EDU/pcboot-2.0.0/board/smdk-2410/ memsetup.S
#endif

relocate:
/*

```

```

* relocate armboot to RAM 下面代码将 ppcboot 全部复制到 SDRAM 中去
*/
adr r0, _start    /* r0 <- current position of code */ /*0x33f00000*/
ldr r2, _armboot_start
ldr r3, _armboot_end    //在反汇编中可以看到这两个地址里面保存的是
                        //两个地址:
                        //33f00044 <_armboot_start>:
                        //33f00044:33f00000 mvnccs r0, #0 ; 0x0
                        //33f00048 <_armboot_end_data>:
                        //33f00048:33f15294 mvnccs r5, #1073741833 ; 0x40000009
sub r2, r3, r2    /* r2 <- size of armboot */
ldr r1, _TEXT_BASE    /* r1 <- destination address */
add r2, r0, r2    /* r2 <- source end address */

/*
* r0 = source address
* r1 = target address
* r2 = source end address
*/
copy_loop:                //循环复制
ldmia r0!, {r3-r10}
stmia r1!, {r3-r10}
cmp r0, r2
ble copy_loop

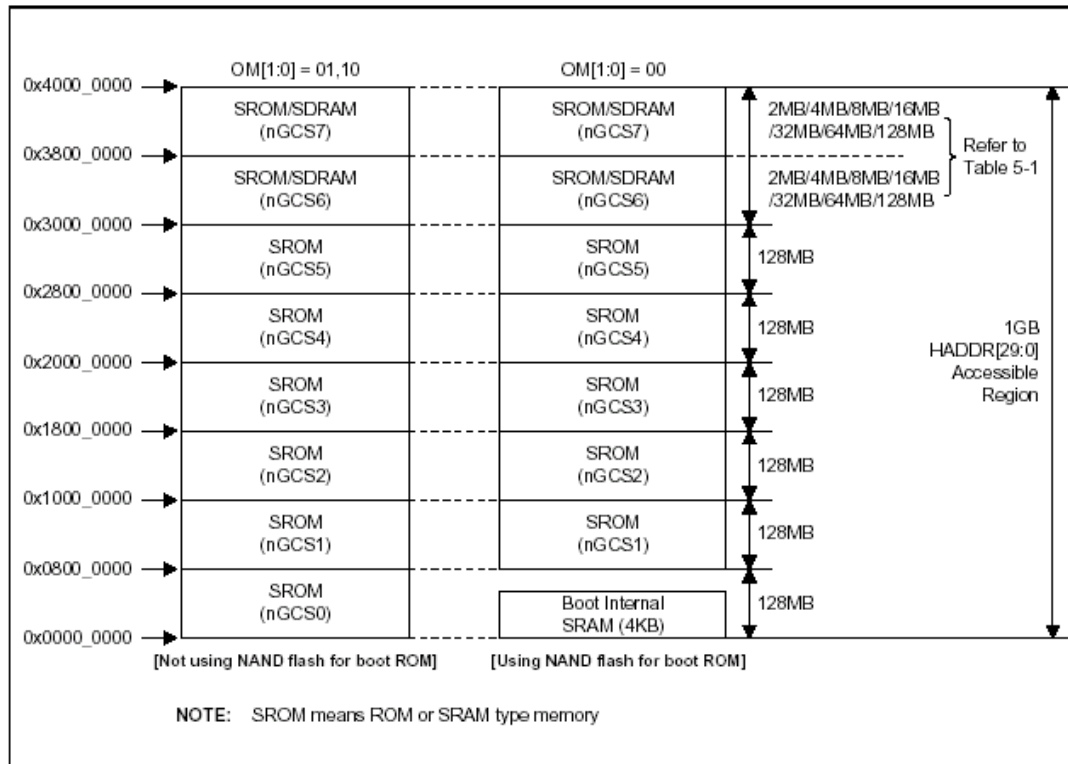
/* set up the stack 设置堆栈指针到 ppcboot 复制到内存的尾端 */
ldr r0, _armboot_end
add r0, r0, #CONFIG_STACKSIZE
sub sp, r0, #12    /* leave 3 words for abort-stack */

ldr pc, _start_armboot //跳转到 ppcboot-2.0.0/lib_arm/board.c 里面执行
                        //void start_armboot (void), 就不再返回了, 这里
                        //第一次用到了绝对地址的跳转, 因为这个时候才刚刚初始化
                        //好 SDRAM
_start_armboot: .word start_armboot

```

1.4.2.4 地址空间

S3C2410 提供 8 路片选, nGCSn[0~7], 每个片选都指定了固定的地址, 每个片选固定间隔为 128M 字节。具体参见 CPU 手册 P5-2。



HHARM9-EDU 开发板内存由两片 16M ×16 位数据宽度的 SDRAM 构成，两片拼成 32 位模式，公用 nGCS6。共 64M RAM。起始物理实地址：0x30000000，经 MMU 映射后地址为 0xc0000000。

这个实地址 TO 虚地址的转换是在：

HHARM9-EDU/kernel/include/asm/arch/memory.h 中实现：

```
/*
 * Page offset: 3GB
 */
#define PAGE_OFFSET (0xc0000000UL)
#define PHYS_OFFSET (0x30000000UL)

/*
 * We take advantage of the fact that physical and virtual address can be the
 * same. This NUMA code is handling the large holes that might exist between
 * all memory banks.
 */
#define __virt_to_phys__is_a_macro
#define __phys_to_virt__is_a_macro
#define __virt_to_phys(x) ((x) - PAGE_OFFSET + PHYS_OFFSET)
#define __phys_to_virt(x) ((x) - PHYS_OFFSET + PAGE_OFFSET)
```

中完成的。

nGCS0 接的是一片 8M ×16 位数据宽度的 INTEL E28F128 FLASH。起始地址：0x10000000。按照 S3C2410 处理器手册，NOR FLASH 安装在 BANK0，地址应该为 0，但由于 2410 中地址是循环映射的，0x01000000 就是 0 地址，也就是 0x10000000。其中内核 zImage 烧写在地址 0x01040000 开始处，根文件系统 RAMDISK 烧在 0x01140000 地址处。

系统启动后，由 ppcboot 负责解压 linux 的内核到物理地址：0x30008000 处，RAMDISK 加载

到 0x30800000 地址处。

具体实现参见下面两处：

/HHARM9-EDU/ppcboot-2.0.0/common/main.c

bootm:

```

    if(c == 'y' || c == 'Y'){
        strcpy(lastcommand, "bootm 30008000 30800000\r");
        flag = 0;
        rc = run_command (lastcommand, flag);
        if (rc <= 0) {
            /* invalid command or not repeatable, forget it */
            lastcommand[0] = 0;
        }
    }
    else{
        printf("\n\n");
    }
}

```

下面是 bootm 的实现代码：

HHARM9-EDU/ppcboot-2.0.0/lib_arm/ armlinux.c

```

void do_bootm_linux(cmd_tbl_t *cmdtp, int flag, int argc, char *argv[],
                    ulong addr, ulong *len_ptr, int verify)

```

```

{
    DECLARE_GLOBAL_DATA_PTR;

    void *ret;
    ulong len = 0, checksum;
    ulong initrd_start, initrd_end;
    ulong data;
    void (*theKernel)(int zero, int arch);
    image_header_t *hdr = &header;
    bd_t *bd = gd->bd;
    #ifdef CONFIG_CMDLINE_TAG
    char *commandline = getenv("bootargs");
    #endif

    /*copy kernel and ramdisk from FLASH to SDRAM add by HHTECH*/
    ret = memcpy((void *)0x30008000, (void *)0x40000, 0x100000);
    if (ret != (void *)0x30008000)
        printf("copy kernel failed\n");
    else
        printf("copy kernel done\n");

    ret = memcpy((void *)0x30800000, (void *)0x140000, 0x440000);

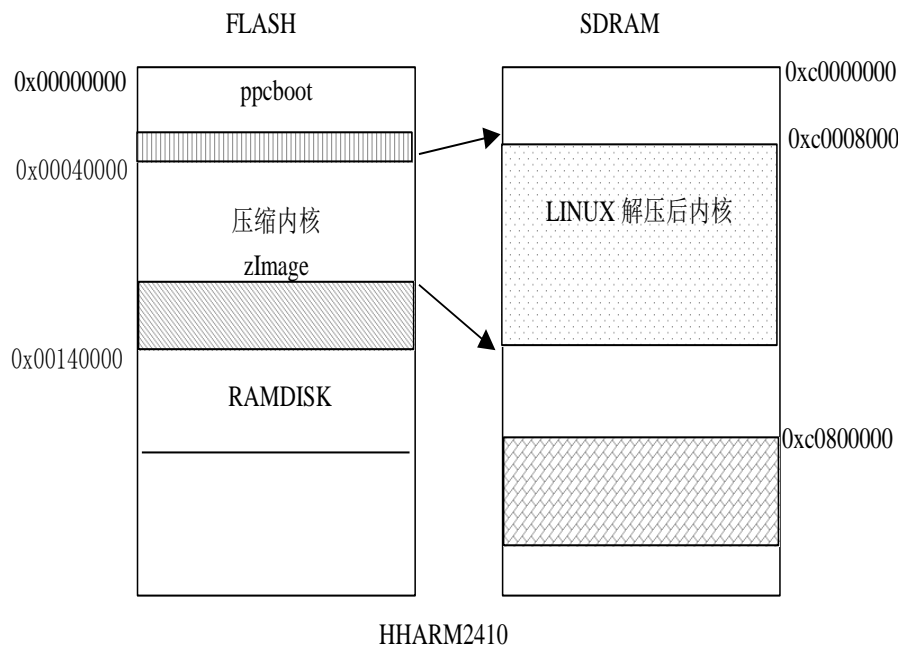
```

```

if (ret != (void *)0x30800000)
    printf("haha failed\n");
else
    printf("copy ramdisk done\n");

```

下面给出板上的地址空间分布：MEMORY MAP



INTEL E28F128J3A-150 FLASH 的单片 16M 字节，共 128 个扇区，每个扇区都是 128K 字节大小，均匀分布。

1.4.3 head.S 和 head-s3c2410.S 分析

1.4.3.1 简介

/HHARM9-EDU/kernel/arch/arm/boot/compressed/head.S、head-s3c2410.S 这两个文件，用汇编代码完成，是启动时最先执行的文件。其主要作用，是分配堆栈、内核、文件系统的空间，并完成对内核文件 zImage 的解压缩工作，同时使 cpu 状态满足 head-armv.S 的执行条件。

1.4.3.2 S3C2410 内核文件 zImage 的生成

zImage 文件是通过内核编译：make zImage 生成的，其生成过程如下：

```

/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -D__KERNEL__ -I/HHARM9-EDU/kernel/include
-Wall -Wstrict-prototypes -Wno-trigraphs -Os -mapcs -fno-strict-aliasing -fno-common
-fno-common -pipe -mapcs-32 -march=armv4 -mtune=arm9tdmi -mshort-load-bytes -msoft-float
-DKBUILD_BASENAME=main -c -o init/main.o init/main.c
. scripts/mkversion > .tmpversion

```

```

/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -D__KERNEL__ -I/HHARM9-EDU/kernel/include
-Wall -Wstrict-prototypes -Wno-trigraphs -Os -mapcs -fno-strict-aliasing -fno-common
-fno-common -pipe -mapcs-32 -march=armv4 -mtune=arm9tdmi -mshort-load-bytes -msoft-float
-DUTS_MACHINE="arm" -DKBUILD_BASENAME=version -c -o init/version.o init/version.c
make[1]: Entering directory `/HHARM9-EDU/kernel/arch/arm/tools'
make CFLAGS="-D__KERNEL__ -I/HHARM9-EDU/kernel/include -Wall -Wstrict-prototypes
-Wno-trigraphs -Os -mapcs -fno-strict-aliasing -fno-common -fno-common -pipe -mapcs-32
-march=armv4 -mtune=arm9tdmi -mshort-load-bytes -msoft-float " -C kernel
make[1]: Entering directory `/HHARM9-EDU/kernel/kernel'
make all_targets
make[2]: Entering directory `/HHARM9-EDU/kernel/kernel'
.....
.....
.....
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -D__ASSEMBLY__ -D__KERNEL__
-I/HHARM9-EDU/kernel/include -mapcs-32 -march=armv4 -mno-fpu -msoft-float -c -o entry.o
entry.S
/opt/host/armv4l/bin/armv4l-unknown-linux-ld -r -o nwfpe.o fpa11.o fpa11_cpdo.o fpa11_cpdt.o
fpa11_cpdt.o fpmodule.o fpopcode.o softfloat.o single_cpdo.o double_cpdo.o extended_cpdo.o
entry.o
rm -f math-emu.o
/opt/host/armv4l/bin/armv4l-unknown-linux-ld -r -o math-emu.o nwfpe.o
make[2]: Leaving directory `/HHARM9-EDU/kernel/arch/arm/nwfpe'
make[1]: Leaving directory `/HHARM9-EDU/kernel/arch/arm/nwfpe'
make CFLAGS="-D__KERNEL__ -I/HHARM9-EDU/kernel/include -Wall -Wstrict-prototypes
-Wno-trigraphs -Os -mapcs -fno-strict-aliasing -fno-common -fno-common -pipe -mapcs-32
-march=armv4 -mtune=arm9tdmi -mshort-load-bytes -msoft-float " -C arch/arm/fastfpe
make[1]: Entering directory `/HHARM9-EDU/kernel/arch/arm/fastfpe'
make all_targets
make[2]: Entering directory `/HHARM9-EDU/kernel/arch/arm/fastfpe'
rm -f fast-math-emu.o
/opt/host/armv4l/bin/armv4l-unknown-linux-ar rcs fast-math-emu.o
make[2]: Leaving directory `/HHARM9-EDU/kernel/arch/arm/fastfpe'
make[1]: Leaving directory `/HHARM9-EDU/kernel/arch/arm/fastfpe'
/opt/host/armv4l/bin/armv4l-unknown-linux-ld -p -X -T arch/arm/vmlinux.lds
arch/arm/kernel/head-armv.o arch/arm/kernel/init_task.o init/main.o init/version.o \
--start-group \
arch/arm/kernel/kernel.o arch/arm/mm/mm.o arch/arm/mach-s3c2410/s3c2410.o
kernel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o \
drivers/serial/serial.o drivers/char/char.o drivers/block/block.o drivers/misc/misc.o
drivers/net/net.o drivers/media/media.o drivers/scsi/scsidrv.o drivers/sound/sounddrivers.o
drivers/mtd/mtdlink.o drivers/video/video.o drivers/usb/usbdrv.o \
net/network.o \
arch/arm/nwfpe/math-emu.o arch/arm/lib/lib.a /HHARM9-EDU/kernel/lib/lib.a \

```

```

--end-group \
-o vmlinux
/opt/host/armv4l/bin/armv4l-unknown-linux-nm vmlinux | grep -v '\(compiled\)\(\.o$\)\([aUw]\)\(\.ng$\)\(LASH[RL]DI\)' | sort > System.map
//到此生成了内核文件 vmlinux
make[1]: Entering directory `/HHARM9-EDU/kernel/arch/arm/boot'
make[2]: Entering directory `/HHARM9-EDU/kernel/arch/arm/boot/compressed'
/opt/host/armv4l/bin/armv4l-unknown-linux-objcopy -O binary -R .note -R .comment -S
/HHARM9-EDU/kernel/vmlinux piggy
gzip -9 < piggy > piggy.gz
/opt/host/armv4l/bin/armv4l-unknown-linux-ld -r -o piggy.o -b binary piggy.gz
rm -f piggy piggy.gz
//将内核 vmlinux 压缩成文件 piggy.o
/opt/host/armv4l/bin/armv4l-unknown-linux-ld -p -X -T vmlinux.lds head.o misc.o head-s3c2410.o
piggy.o /opt/host/armv4l/lib/gcc-lib/armv4l-unknown-linux/2.95.2/soft-float/libgcc.a -o vmlinux
make[2]: Leaving directory `/HHARM9-EDU/kernel/arch/arm/boot/compressed'
/opt/host/armv4l/bin/armv4l-unknown-linux-objcopy -O binary -R .note -R .comment -S
compressed/vmlinux zImage
cp -f zImage /tftpboot/
make[1]: Leaving directory `/HHARM9-EDU/kernel/arch/arm/boot'
//由 head.o、head-s3c2410.o、misc.o、piggy.o 生成最终//的内核文件
zImage，并将 zImage 拷贝到/tftpboot/目录中

```

1.4.3.3 head.S 代码分析

（略去一些条件编译的代码）

```

/*
 * linux/arch/arm/boot/compressed/head.S
 * Copyright (C) 1996-2002 Russell King
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
#include <linux/config.h>
#include <linux/linkage.h>

    .macro kputc,val
    movr0, \val
    bl putc
    .endm

```

```

        .macro   kphex,val,len
        movr0, \val
        movr1, #\len
        bl   phex
        .endm

        .section ".start", #alloc, #execinstr
/*
* sort out different calling conventions
*/
        .align
start:
        .type    start,#function
        .rept    8
        movr0, r0
        .endr

        b   1f
        .word 0x016f2818      @ Magic numbers to help the loader
        .word start          @ absolute load/run zImage address
        .word _edata         @ zImage end address
1:      movr7, r1              @ save architecture ID
                                   //保存 architecture ID
        movr8, #0            @ save r0
                                   //保存 r0

#ifdef __ARM_ARCH_2__
/*
* Booting from Angel - need to enter SVC mode and disable
* FIQs/IRQs (numeric definitions from angel arm.h source).
* We only do this if we were in user mode on entry.
*/
        mrs r2, cpsr          @ get current mode
        tst r2, #3            @ not user?
        bne not_angel
        movr0, #0x17          @ angel_SWIreason_EnterSVC
        swi 0x123456          @ angel_SWI_ARM
not_angel:
        mrs r2, cpsr          @ turn off interrupts to
        orr r2, r2, #0xc0      @ prevent angel from running
        msr cpsr_c, r2
#else
        teqppc, #0x0c000003    @ turn off interrupts

```

```
#endif

                                //关闭所有的中断

/*
 * Note that some cache flushing and other stuff may
 * be needed here - is there an Angel SWI call for this?
 */

/*
 * some architecture specific code can be inserted
 * by the linker here, but it should preserve r7 and r8.
 */

.text
adr r0, LC0
ldmia r0, {r1, r2, r3, r4, r5, r6, ip, sp}
subs r0, r0, r1 @ calculate the delta offset
                                //计算偏移地址

teq r0, #0 @ if delta is zero, we're
beq not_relocated @ running at the address we were linked at.

/*
 * We're running at a different address. We need to fix
 * up various pointers:
 * r5 - zImage base address
 * r6 - GOT start
 * ip - GOT end
 */
add r5, r5, r0
add r6, r6, r0
add ip, ip, r0

#ifdef CONFIG_ZBOOT_ROM
/*
 * If we're running fully PIC == CONFIG_ZBOOT_ROM = n,
 * we need to fix up pointers into the BSS region.
 * r2 - BSS start
 * r3 - BSS end
 * sp - stack pointer
 */
add r2, r2, r0
add r3, r3, r0
add sp, sp, r0
```

```

/*
 * Relocate all entries in the GOT table.
 */
1:    ldr r1, [r6, #0]
      add r1, r1, r0
      str r1, [r6], #4
      cmpr6, ip
      blo 1b

#else

/*
 * Relocate entries in the GOT table. We only relocate
 * the entries that are outside the (relocated) BSS region.
 */
1:    ldr r1, [r6, #0]
      cmpr1, r2                @ entry < bss_start ||
      cmphs r3, r1             @ _end < entry
      addlo r1, r1, r0
      str r1, [r6], #4
      cmpr6, ip
      blo 1b

#endif

not_relocated:    movr0, #0
1:    str r0, [r2], #4          @ clear bss
      str r0, [r2], #4
      str r0, [r2], #4
      str r0, [r2], #4
      cmpr2, r3
      blo 1b

/*
 * The C runtime environment should now be setup
 * sufficiently. Turn the cache on, set up some
 * pointers, and start decompressing.
 */
bl cache_on

movr1, sp                @ malloc space above stack
add r2, sp, #0x10000      @ 64k max
                                //分配堆栈空间: 最大 64k

/*

```


* Check to see if we will overwrite ourselves.

* r4 = final kernel address

* r5 = start of this image

* r2 = end of malloc space (and therefore this image)

* We basically want:

* r4 >= r2 -> OK

* r4 + image length <= r5 -> OK

*/

cmpr4, r2

bhs wont_overwrite

add r0, r4, #4096*1024 @ 4MB largest kernel size

//分配内核空间：最大 4M

cmpr0, r5

bls wont_overwrite

movr5, r2 @ decompress after malloc space

movr0, r5

movr3, r7

bl decompress_kernel

//执行 decompress_kernel（在 misc.c 中），解压缩

//zImage 解压后的内核从 r5 地址开始

add r0, r0, #127

bic r0, r0, #127 @ align the kernel length

/*

* r0 = decompressed kernel length

* r1-r3 = unused

* r4 = kernel execution address

* r5 = decompressed kernel start

* r6 = processor ID

* r7 = architecture ID

* r8-r14 = unused

*/

add r1, r5, r0 @ end of decompressed kernel

adr r2, reloc_start

ldr r3, LC1

add r3, r2, r3

1: ldmia r2!, {r8 - r13} @ copy relocation code

stmia r1!, {r8 - r13}

ldmia r2!, {r8 - r13}

stmia r1!, {r8 - r13}

cmpr2, r3

blo 1b

bl cache_clean_flush

```

        add pc, r5, r0          @ call relocation code

/*
 * We're not in danger of overwriting ourselves.  Do this the simple way.
 *
 * r4      = kernel execution address
 * r7      = architecture ID
 */
wont_overwrite:  mov r0, r4
                 mov r3, r7
                 bl  decompress_kernel
                 b   call_kernel

                 .type    LC0, #object
LC0:             .word    LC0          @ r1
                 .word    __bss_start @ r2
                 .word    _end        @ r3
                 .word    _load_addr  @ r4
                 .word    _start      @ r5
                 .word    _got_start  @ r6
                 .word    _got_end    @ ip
                 .word    user_stack+4096 @ sp
LC1:             .word    reloc_end - reloc_start
                 .size    LC0, . - LC0

/*
 * Turn on the cache.  We need to setup some page tables so that we
 * can have both the I and D caches on.
 *
 * We place the page tables 16k down from the kernel execution address,
 * and we hope that nothing else is using it.  If we're using it, we
 * will go pop!
 *
 * On entry,
 * r4 = kernel execution address
 * r6 = processor ID
 * r7 = architecture number
 * r8 = run-time address of "start"
 * On exit,
 * r1, r2, r3, r8, r9, r12 corrupted
 * This routine must preserve:
 * r4, r5, r6, r7
 */
        .align    5

```

```

cache_on:  movr3, #8          @ cache_on function
           b    call_cache_fn

__setup_mmu:

           sub r3, r4, #16384    @ Page directory size
           bic r3, r3, #0xff     @ Align the pointer
           bic r3, r3, #0x3f00

/*
 * Initialise the page tables, turning on the cacheable and bufferable
 * bits for the RAM area only.
 */
           movr0, r3
           movr8, r0, lsr #18
           movr8, r8, lsl #18 @ start of RAM
                                   //RAM 的起始地址
           add r9, r8, #0x10000000 @ a reasonable RAM size
                                   //RAM 的可能大小: 256M

           movr1, #0x12
           orr r1, r1, #3 << 10
           add r2, r3, #16384
1:         cmp r1, r8            @ if virt > start of RAM

           orr r1, r1, #0x0c     @ set cacheable, bufferable

           cmp r1, r9            @ if virt > end of RAM
           bichs r1, r1, #0x0c   @ clear cacheable, bufferable
           str r1, [r0], #4      @ 1:1 mapping
           add r1, r1, #1048576
           teq r0, r2
           bne 1b

/*
 * If ever we are running from Flash, then we surely want the cache
 * to be enabled also for our execution instance... We map 2MB of it
 * so there is no map overlap problem for up to 1 MB compressed kernel.
 * If the execution is in RAM then we would only be duplicating the above.
 */
           movr1, #0x1e
           orr r1, r1, #3 << 10
           movr2, pc, lsr #20
           orr r1, r1, r2, lsl #20
           add r0, r3, r2, lsl #2
           str r1, [r0], #4
           add r1, r1, #1048576

```

```

    str r1, [r0]
    movpc, lr

__armv4_cache_on:
    movr12, lr
    bl __setup_mmu
    movr0, #0
    mcr p15, 0, r0, c7, c10, 4 @ drain write buffer
    mcr p15, 0, r0, c8, c7 @ flush I,D TLBs
    mcr p15, 0, r3, c2, c0 @ load page table pointer
    movr0, #-1
    mcr p15, 0, r0, c3, c0 @ load domain access register
    mrc p15, 0, r0, c1, c0
    orr r0, r0, #0x1000 @ I-cache enable

    orr r0, r0, #0x003d @ Write buffer, mmu

    mcr p15, 0, r0, c1, c0
    movpc, r12

/*
 * All code following this line is relocatable. It is relocated by
 * the above code to the end of the decompressed kernel image and
 * executed there. During this time, we have no stacks.
 *
 * r0 = decompressed kernel length
 * r1-r3 = unused
 * r4 = kernel execution address
 * r5 = decompressed kernel start
 * r6 = processor ID
 * r7 = architecture ID
 * r8-r14 = unused
 */
    .align 5
reloc_start: add r8, r5, r0
    debug_reloc_start
    movr1, r4
1:
    .rept 4
    ldmia r5!, {r0, r2, r3, r9 - r13} @ relocate kernel
    stmia r1!, {r0, r2, r3, r9 - r13}
    .endr

    cmp r5, r8

```

```

        blo 1b
        debug_reloc_end

call_kernel: bl  cache_clean_flush
            bl  cache_off
            movr0, #0
            movr1, r7                @ restore architecture number
            movpc, r4                @ call kernel

```

```

/*
 * Here follow the relocatable cache support functions for the
 * various processors. This is a generic hook for locating an
 * entry and jumping to an instruction at the specified offset
 * from the start of the block. Please note this is all position
 * independent code.

```

```

 *

```

```

 * r1 = corrupted
 * r2 = corrupted
 * r3 = block offset
 * r6 = corrupted
 * r12 = corrupted

```

```

 */

```

```

call_cache_fn:  adr r12, proc_types
                mrc p15, 0, r6, c0, c0    @ get processor ID
1:             ldr  r1, [r12, #0]          @ get value
                ldr  r2, [r12, #4]          @ get mask
                eor  r1, r1, r6            @ (real ^ match)
                tst  r1, r2                @      & mask
                addeq pc, r12, r3          @ call cache function
                add r12, r12, #4*5
                b    1b

```

```

/*

```

```

 * Table for cache operations. This is basically:

```

```

 * - CPU ID match
 * - CPU ID mask
 * - 'cache on' method instruction
 * - 'cache off' method instruction
 * - 'cache flush' method instruction
 *

```

```

 * We match an entry using: ((real_id ^ match) & mask) == 0

```

```

 *

```

```

 * Writethrough caches generally only need 'on' and 'off'

```

```

* methods. Writeback caches _must_ have the flush method
* defined.
*/

```

```

.type    proc_types,#object

```

```

//确定 cpu 类型

```

```

proc_types:

```

```

.word    0x41560600    @ ARM6/610

```

```

.word    0xffffffffe0

```

```

b    __arm6_cache_off    @ works, but slow

```

```

b    __arm6_cache_off

```

```

movpc, lr

```

```

.word    0x41007000    @ ARM7/710

```

```

.word    0xff8fe00

```

```

b    __arm7_cache_off

```

```

b    __arm7_cache_off

```

```

movpc, lr

```

```

.word    0x41807200    @ ARM720T (writethrough)

```

```

.word    0xfffff00

```

```

b    __armv4_cache_on

```

```

b    __armv4_cache_off

```

```

movpc, lr

```

```

.word    0x41129200    @ ARM920T

```

```

.word    0xff00fff0

```

```

b    __armv4_cache_on    //cache on

```

```

b    __armv4_cache_off    //cache off

```

```

b    __armv4_cache_flush    //cache flush

```

```

//S3C2410 属于 ARM920T

```

```

.word    0x4401a100    @ sa110 / sa1100

```

```

.word    0xffffffffe0

```

```

b    __armv4_cache_on

```

```

b    __armv4_cache_off

```

```

b    __armv4_cache_flush

```

```

.word    0x6901b110    @ sa1110

```

```

.word    0xffffffff0

```

```

b    __armv4_cache_on

```

```

b    __armv4_cache_off

```

```

b    __armv4_cache_flush

```

```

.word    0x69050000    @ xscale

```

```

.word    0xffff0000

```

```

b    __armv4_cache_on
b    __armv4_cache_off
b    __armv4_cache_flush

.word    0                @ unrecognised type
.word    0
movpc, lr
movpc, lr
movpc, lr

.size    proc_types, . - proc_types

/*
* Turn off the Cache and MMU.  ARMv3 does not support
* reading the control register, but ARMv4 does.
*
* On entry,  r6 = processor ID
* On exit,   r0, r1, r2, r3, r12 corrupted
* This routine must preserve: r4, r6, r7
*/
    .align    5
cache_off:  movr3, #12      @ cache_off function
            b    call_cache_fn

__armv4_cache_off:
    mrc p15, 0, r0, c1, c0
    bic r0, r0, #0x000d
    mcr p15, 0, r0, c1, c0    @ turn MMU and cache off
    movr0, #0
    mcr p15, 0, r0, c7, c7    @ invalidate whole cache v4
    mcr p15, 0, r0, c8, c7    @ invalidate whole TLB v4
    movpc, lr

/*
* Clean and flush the cache to maintain consistency.
*
* On entry,
* r6 = processor ID
* On exit,
* r1, r2, r3, r12 corrupted
* This routine must preserve:
* r0, r4, r5, r6, r7
*/
    .align    5

```

```

cache_clean_flush:
    movr3, #16
    b    call_cache_fn

__armv4_cache_flush:
    bic r1, pc, #31
    add r2, r1, #65536          @ 2x the largest dcache size
1:    ldr r12, [r1], #32        @ s/w flush D cache
    teq r1, r2
    bne 1b

    mcr p15, 0, r1, c7, c7, 0   @ flush I cache
    mcr p15, 0, r1, c7, c10, 4  @ drain WB
    movpc, lr

reloc_end:

    .align
    .section ".stack", "aw"
user_stack: .space 4096        //可使用堆栈大小: 4096

```

1.4.3.4 head-s3c2410.S 代码分析

实验所使用的 cpu 是 s3c2410，因此当执行完 head.S 后执行 head-s3c2410.S

```

-----
/*
 * linux/arch/arm/boot/compressed/head-s3c2410.S
 * Copyright (C) 2001 MIZI Research, Inc.
 * s3c2410 specific tweaks. This is merged into head.S by the linker.
 */

#include <linux/config.h>
#include <linux/linkage.h>
#include <asm/mach-types.h>

.section ".start", #alloc, #execinstr

__S3C2410_start:

    @ Preserve r8/r7 i.e. kernel entry values
    @ What is it?
    @ Nandy

```


@ Data cache, Instruction cache, MMU might be active.
 @ Be sure to flush kernel binary out of the cache,
 @ whatever state it is, before it is turned off.
 @ This is done by fetching through currently executed
 @ memory to be sure we hit the same cache

```

bic r2, pc, #0x1f
add r3, r2, #0x4000      @ 16 kb is quite enough...
1: ldr r0, [r2], #32
   teq r2, r3
   bne 1b
   mcr p15, 0, r0, c7, c10, 4 @ drain WB
   mcr p15, 0, r0, c7, c7, 0  @ flush I & D caches
/*
  * Pause for a short time so that we give enough time
  * for the host to start a terminal up.
  */
mov r0, #0x00200000
1: subs    r0, r0, #1
   bne 1b

```

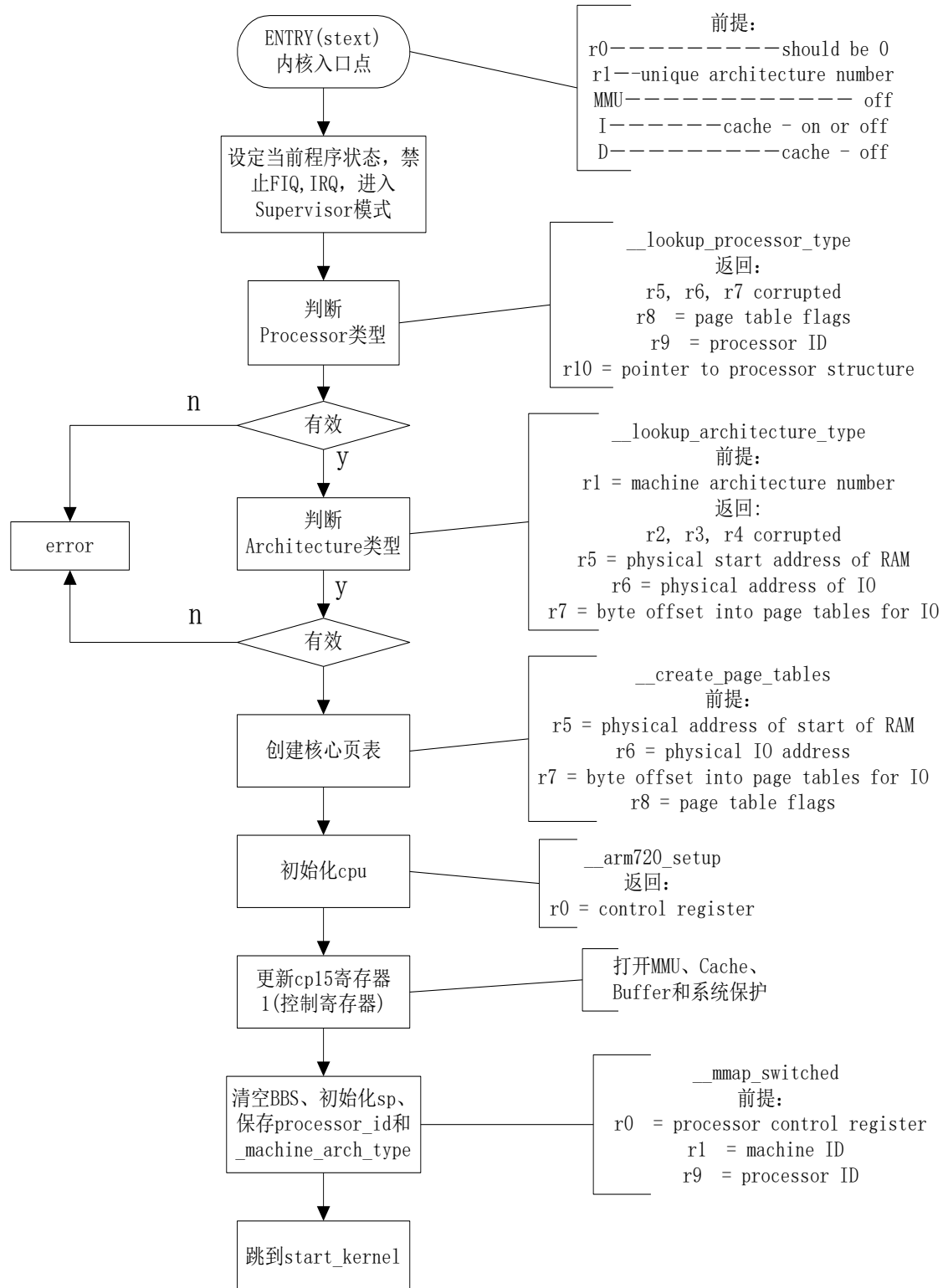
1.4.4 head-armv.S 分析

1.4.4.1 说明

这个文件是/HHARM9-EDU/kernelarch/arm/kernel/head-armv.S，用汇编代码完成，是内核最先执行的一个文件。这一段汇编代码的主要作用，是检查 cpu id，architecture number，初始化页表、cpu、bbs 等操作，并跳到 start_kernel 函数。它在执行前，处理器的状态应满足：

r0 - should be 0
 r1 - unique architecture number
 MMU - off
 I-cache - on or off
 D-cache - off

1.4.4.2 流程



1.4.4.3 代码详细注释

```
/*
 * linux/arch/arm/kernel/head-armv.S
 * Copyright (C) 1994-1999 Russell King
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 * 32-bit kernel startup code for all architectures
 */
#include <linux/config.h>
#include <linux/linkage.h>
#include <asm/asm.h>
#include <asm/mach-types.h>
#include <asm/mach/arch.h>
#define K(a,b,c) ((a) << 24 | (b) << 12 | (c))
/*
 * We place the page tables 16K below TEXTADDR. Therefore, we must make sure
 * that TEXTADDR is correctly set. Currently, we expect the least significant
 * "short" to be 0x8000, but we could probably relax this restriction to
 * TEXTADDR > PAGE_OFFSET + 0x4000
 * Note that swapper_pg_dir is the virtual address of the page tables, and
 * pgtbl gives us a position-independent reference to these tables. We can
 * do this because stext == TEXTADDR
 * swapper_pg_dir, pgtbl and krnladr are all closely related.
 */
#if (TEXTADDR & 0xffff) != 0x8000
#error TEXTADDR must start at 0xFFFF8000
#endif

.globl SYMBOL_NAME(swapper_pg_dir)
.equ SYMBOL_NAME(swapper_pg_dir), TEXTADDR - 0x4000
.macro pgtbl, reg, rambase
adr \reg, stext
sub \reg, \reg, #0x4000
.endm

/*
 * Since the page table is closely related to the kernel start address, we
 * can convert the page table base address to the base address of the section
 * containing both.
 */

.macro krnladr, rd, pgtable, rambase
bic \rd, \pgtable, #0x000ff000
.endm
```

```

/*
 * Kernel startup entry point.
 * The rules are:
 * r0      - should be 0
 * r1      - unique architecture number
 * MMU     - off
 * I-cache - on or off
 * D-cache - off
 * See linux/arch/arm/tools/mach-types for the complete list of numbers
 * for r1.
 */

.section ".text.init",#alloc,#execinstr
.type    stext, #function
ENTRY(stext)                                //内核入口点
    movr12, r0                               //r0=0, r12=0

    mov r0, #F_BIT | I_BIT | MODE_SVC@ make sure svc mode
                                           //程序状态，禁止 FIQ、IRQ，设定
                                           // Supervisor 模式。0b11010011

    msr cpsr_c, r0                          @ and all irqs disabled
                                           //置当前程序状态寄存器

    bl  __lookup_processor_type              //跳转到判断 cpu 类型，查找运行的
                                           //cpu 的 id 值，和此 linux 编译支持的
                                           //id 值，是否有相等

    teq r10, #0                             @ invalid processor?
                                           //没有则跳到__error

    moveq r0, #'p'                          @ yes, error 'p'
    beq __error

    bl  __lookup_architecture_type           //跳转到判断体系类型，看 r1 寄存器
                                           //的 architecture number 值是否支持。

    teq r7, #0                             @ invalid architecture?
                                           //不支持，跳到出错

    moveq r0, #'a'                          @ yes, error 'a'
    beq __error

    bl  __create_page_tables                 //创建核心页表

    adr lr, __ret                           @ return address //lr=0xc0028054
    add pc, r10, #12                       @ initialise processor
                                           @ (return control reg)
                                           //r10: pointer to processor structure
                                           // (__arm720_proc_info)
                                           //r10+12: __arm720_setup; 见
                                           //__arm720_proc_info (proc-arm720.S)

```

(proc-arm720.S的相应代码:

```

__arm720_setup: movr0, #0

```

```

mcr p15, 0, r0, c7, c7, 0      @ invalidate caches
mcr p15, 0, r0, c8, c7, 0      @ flush TLB (v4)
mcr p15, 0, r4, c2, c0          @ load page table pointer
                                //cp15寄存器1(ttb)=0xc0024000
movr0, #0x1f                    @ Domains 0, 1 = client
mcr p15, 0, r0, c3, c0          @ load domain access register
mrc p15, 0, r0, c1, c0          @ get control register
                                //r0=0x70
bic r0, r0, #0x0e00             @ ..V. ..RS BLDP WCAM
                                //bit[11:9]=0 r0=0x00000070
orr r0, r0, #0x2100             @ .... .... .111 .... (old)
                                //r0=0x00002170
orr r0, r0, #0x003d             @ ..1. ..01 ..11 1101 (new)
                                //r0=0x0000217d其中S LDPWC
                                //M位置1。（详见cp15寄存器1说明）

movpc, lr                       @ __ret (head-armv.S)
)

.type __switch_data, %object
__switch_data: .long __mmap_switched
               .long SYMBOL_NAME(__bss_start)
               .long SYMBOL_NAME(_end)
               .long SYMBOL_NAME(processor_id)
               .long SYMBOL_NAME(__machine_arch_type)
               .long SYMBOL_NAME(cr_alignment)
               .long SYMBOL_NAME(init_task_union)+8192

.type __ret, %function
__ret: ldr lr, __switch_data
mcr p15, 0, r0, c1, c0 //更新控制寄存器cp15寄存器1=0x0000217d
movr0, r0
movr0, r0
movr0, r0
movpc, lr //__switch_data

/*
 * This code follows on after the page
 * table switch and jump above.
 *
 * r0 = processor control register
 * r1 = machine ID
 * r9 = processor ID
 */
.align 5

```

__mmap_switched:

```

//把 sp 指针指向 init_task_union+8192
// (include/linux/sched.h) 处，即第一
//个进程的 task_struct 和系统堆栈的
//地址；清空 BSS 段；保存 processor ID
//和 machine type,到全局变量
//processor_id 和 __machine_arch_type
//这些值以后要用到；r0 为"A"置位的
//control register 值，r2 为"A"清空的
//control register 值，即对齐检查
//（Alignment fault checking）位，并
//保存到 cr_alignment，和 cr_no_alig-
//nment（在文件 entry-armv.S 中）。最
//后跳转到 start_kernel（init/main.c）

adr r3, __switch_data + 4          // __bss_start
ldmia r3, {r4, r5, r6, r7, r8, sp}@ r2 = compat
                                   //r2=0xc0000000
                                   @ sp = stack pointer
                                   //r4=0xc00c04e0; __bss_start
                                   //r5=0xc00e02a8; _end
                                   //r6=0xc00c0934; processor_id
                                   //r7=0xc00c0930; __machine_arch_type
                                   //r8=0xc00bcb88; cr_alignment
                                   //sp=0xc00bc000;(init_task_union)+8192

movfp, #0                          @ Clear BSS (and zero fp)
1: cmp r4, r5
   strcc fp, [r4],#4
   bcc 1b

   str r9, [r6]                     @ Save processor ID
   str r1, [r7]                     @ Save machine type
#ifdef CONFIG_ALIGNMENT_TRAP
   orr r0, r0, #2                   @ .....A.
#endif
   bic r2, r0, #2                   @ Clear 'A' bit
                                   //r0=0x217d
   stmia r8, {r0, r2}              @ Save control register values
   b SYMBOL_NAME(start_kernel) //跳转到 start_kernel
/*
* Setup the initial page tables. We only setup the barest
* amount which are required to get the kernel running, which
* generally means mapping in the kernel code.
*

```

* We only map in 4MB of RAM, which should be sufficient in

* all cases.

*

* r5 = physical address of start of RAM

* r6 = physical IO address

* r7 = byte offset into page tables for IO

* r8 = page table flags

*/

__create_page_tables:

pgtbl r4, r5 @ page table address

//调用宏 pgtbl, r4=0xc0024000: 页

//表基址

/*

* Clear the 16K level 1 swapper page table

*/

movr0, r4//r0=0xc0024000

movr3, #0

add r2, r0, #0x4000//r2=0xc0028000

1: str r3, [r0], #4

str r3, [r0], #4

str r3, [r0], #4

str r3, [r0], #4

teq r0, r2

bne 1b

//将地址 0xc0024000~0xc0028000 清 0

/*

* Create identity mapping for first MB of kernel to

* cater for the MMU enable. This identity mapping

* will be removed by paging_init()

*/

krnladr r2, r4, r5 @ start of kernel

//r2=0xc0000000; r4=0xc0024000

add r3, r8, r2 @ flags + kernel base

//flags, r8=0xc1e; r3=0xc0000c1e

str r3, [r4, r2, lsr #18] @ identity mapping

//addr: 0xc0027000;value: 0xc0000c1e

/*

* Now setup the pagetables for our kernel direct

* mapped region. We round TEXTADDR down to the

* nearest megabyte boundary.

*/

add r0, r4, #(TEXTADDR & 0xff000000) >> 18 @ start of kernel

//r0=0xc0027000

bic r2, r3, #0x00f00000

//r2=0xc0000c1e

str r2, [r0] @ PAGE_OFFSET + 0MB

```

add r0, r0, #(TEXTADDR & 0x00f00000) >> 18
str  r3, [r0], #4          @ KERNEL + 0MB
add r3, r3, #1 << 20
str  r3, [r0], #4          @ KERNEL + 1MB
add r3, r3, #1 << 20
str  r3, [r0], #4          @ KERNEL + 2MB
add r3, r3, #1 << 20
str  r3, [r0], #4          @ KERNEL + 3MB

```

//核心页表:

//addr

一级描述符值

//0xc0027000

0xc0000c1e

//0xc0027004

0xc0100c1e

//0xc0027008

0xc0200c1e

//0xc002700c

0xc0300c1e

//r0 =

0xc0027010

/*

* Ensure that the first section of RAM is present.

* we assume that:

* 1. the RAM is aligned to a 32MB boundary

* 2. the kernel is executing in the same 32MB chunk

* as the start of RAM.

*/

```

bic  r0, r0, #0x01f00000 >> 18 @ round down

```

//r0=0xc0027000

```

and  r2, r5, #0xfe000000 @ round down

```

//r2=0xc0000000

```

add  r3, r8, r2 @ flags + rambase

```

//r3=0xc0000c1e

```

str  r3, [r0]

```

```

bic  r8, r8, #0x0c @ turn off cacheable

```

@ and bufferable bits

//r8=0xc12

```

movpc, lr

```

/*

* Exception handling. Something went wrong and we can't

* proceed. We ought to tell the user, but since we

* don't have any guarantee that we're even running on

* the right architecture, we do virtually nothing.

* r0 = ascii error character:

* a = invalid architecture

* p = invalid processor

* i = invalid calling convention

*

* Generally, only serious errors cause this.


```

*/
__error:
1:      movr0, r0
        b    1b
/*
* Read processor ID register (CP#15, CR0), and look up in the linker-built
* supported processor list. Note that we can't use the absolute addresses
* for the __proc_info lists since we aren't running with the MMU on
* (and therefore, we are not in the correct address space). We have to
* calculate the offset.
* 返回值:
*   r5, r6, r7 corrupted
*   r8  = page table flags
*   r9  = processor ID
*   r10 = pointer to processor structure
*/
__lookup_processor_type:                                //判断 cpu 类型
        adr r5, 2f                                       //取标号 2 的地址
        ldmia r5, {r7, r9, r10}                          //r7: __proc_info_end;
                                                         //r9: __proc_info_begin; r10: r5
        sub r5, r5, r10                                  @ convert addresses
                                                         //r5=0? ?
        add r7, r7, r5                                    @ to our address space
        add r10, r9, r5                                  //r10: __proc_info_begin
        mrc p15, 0, r9, c0, c0                          @ get processor id
                                                         //读取 cp15 寄存器 0 中 cpu id 至 r9。//在此版本
                                                         //是 0x41807200
1:      ldmia r10, {r5, r6, r8}                          @ value, mask, mmuflags
                                                         //读取 arm linux 中 cpu 信息
                                                         // r5, id: 0x41807200; r6, mask:
                                                         //0xffffffff00; r8, mmuflags 即一级描
                                                         //述符: 0xc1e
        and r6, r6, r9                                  @ mask wanted bits
                                                         //屏蔽 cpu id 的低 8 位
        teq r5, r6                                       //寄存器 0 的 cpu id 与 arm linux 中 cpu
                                                         //id 比较
        moveq pc, lr                                     //相同则返回
        add r10, r10, #36                                @ sizeof(proc_info_list)
                                                         //否则寻找下一块 proc_info
        cmpr10, r7
        blt 1b
        movr10, #0                                       @ unknown processor
                                                         //没有匹配信息, r10=0
        movpc, lr

```

```

/*
 * Look in include/asm-arm/procinfo.h and arch/arm/kernel/arch.[ch] for
 * more information about the __proc_info and __arch_info structures.
 */
2:      .long    __proc_info_end
        .long    __proc_info_begin
        .long    2b
        .long    __arch_info_begin
        .long    __arch_info_end

/*
 * Lookup machine architecture in the linker-build list of architectures.
 * Note that we can't use the absolute addresses for the __arch_info
 * lists since we aren't running with the MMU on (and therefore, we are
 * not in the correct address space). We have to calculate the offset.
 * r1 = machine architecture number
 * 返回值为:
 * r2, r3, r4 corrupted
 * r5 = physical start address of RAM
 * r6 = physical address of IO
 * r7 = byte offset into page tables for IO
 */
__lookup_architecture_type:
        adr r4, 2b                                //判断体系类型
        ldmia r4, {r2, r3, r5, r6, r7}           //取上面标号 2 的地址
                                                @ throw away r2, r3
                                                //r5:r4; r6: __arch_info_begin;
                                                //r7: __arch_info_end

        sub r5, r4, r5                            @ convert addresses
                                                //r5=0

        add r4, r6, r5                            @ to our address space
                                                //r4: __arch_info_begin;

        add r7, r7, r5

1:      ldr r5, [r4]                                @ get machine type
        teq r5, r1                                //r1 是 machine type 号, 此为 91
        beq 2f
        add r4, r4, #SIZEOF_MACHINE_DESC //不匹配, 查找下一个 arch_info
        cmpr4, r7
        blt 1b
        mov r7, #0                                @ unknown architecture
        mov pc, lr

2:      ldmiwb r4, {r5, r6, r7}                    @ found, get results
                                                //r5, ram 物理起始地址: 0xc0000000; //r6, io 地
                                                址: 0x80000000;
                                                //r7, io 在页表的偏移: 0x3fc0

        mov pc, lr                                //返回

```

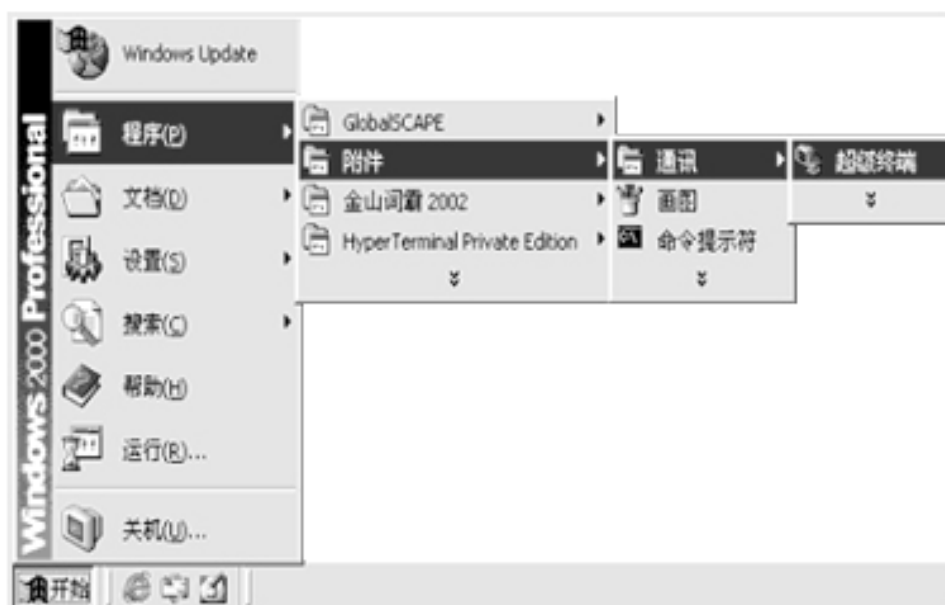
第二章 S3C2410 处理器特点和 HHARM9-EDU 软硬件系统描述

2.1 快速上手指南

在本章节中您可以对 **HHARM9-EDU** 教学实验平台有个初步的认识,对开发板进行简单检测及建立开发环境。

Windows98 和 Windows2000 安装盘中都附有超级终端应用程序。如果您的系统没有安装超级终端,您可以在控制面板“添加/删除程序”一栏选择安装超级终端应用程序。

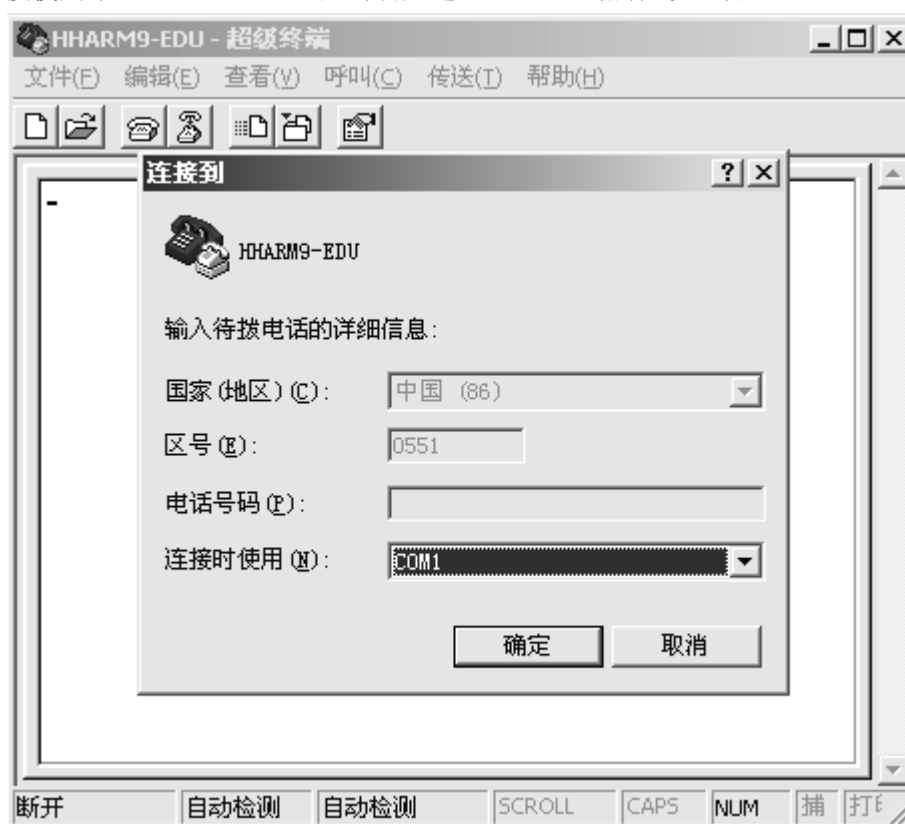
首先我们在 Windows2000 环境下启动超级终端。



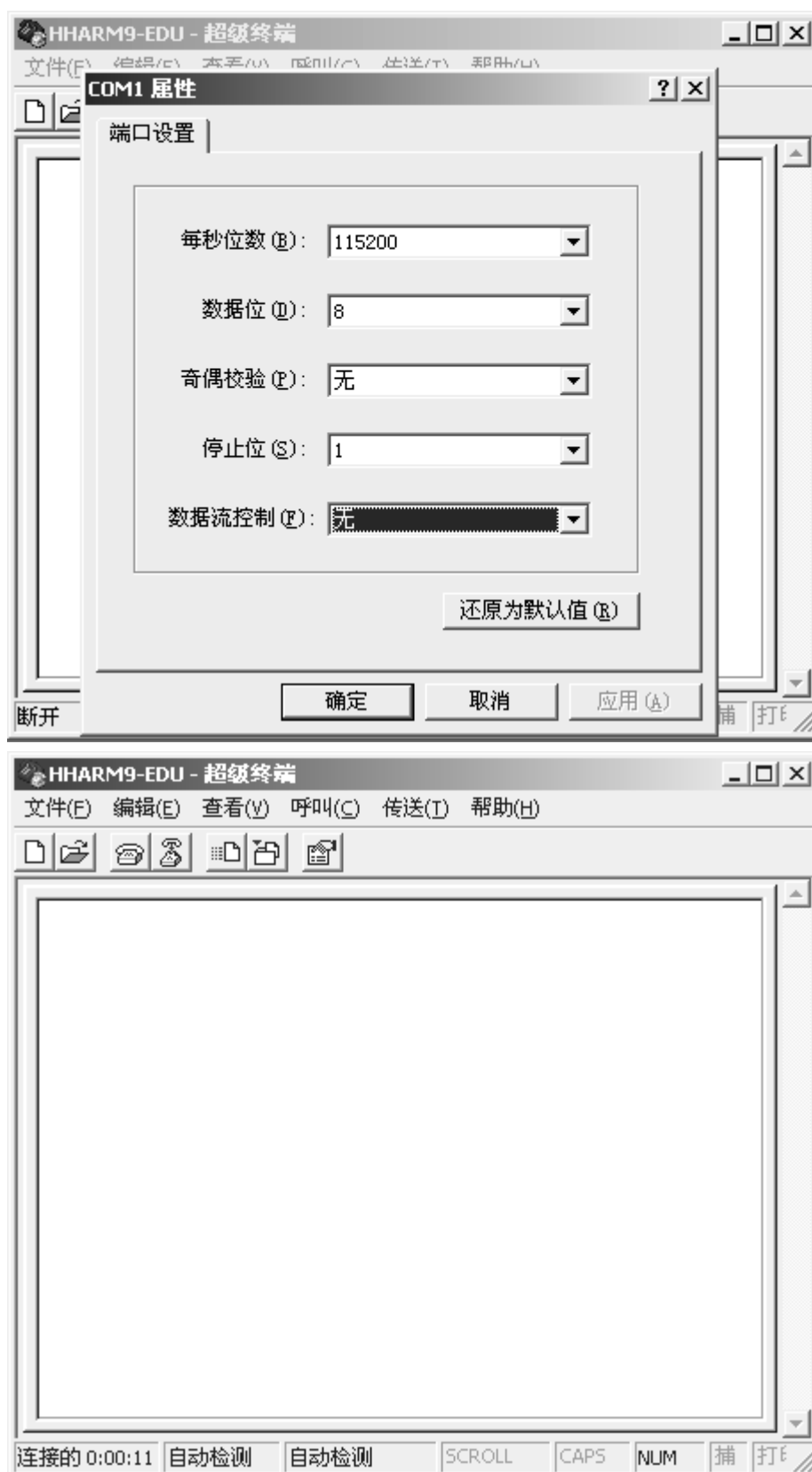
进入超级终端。



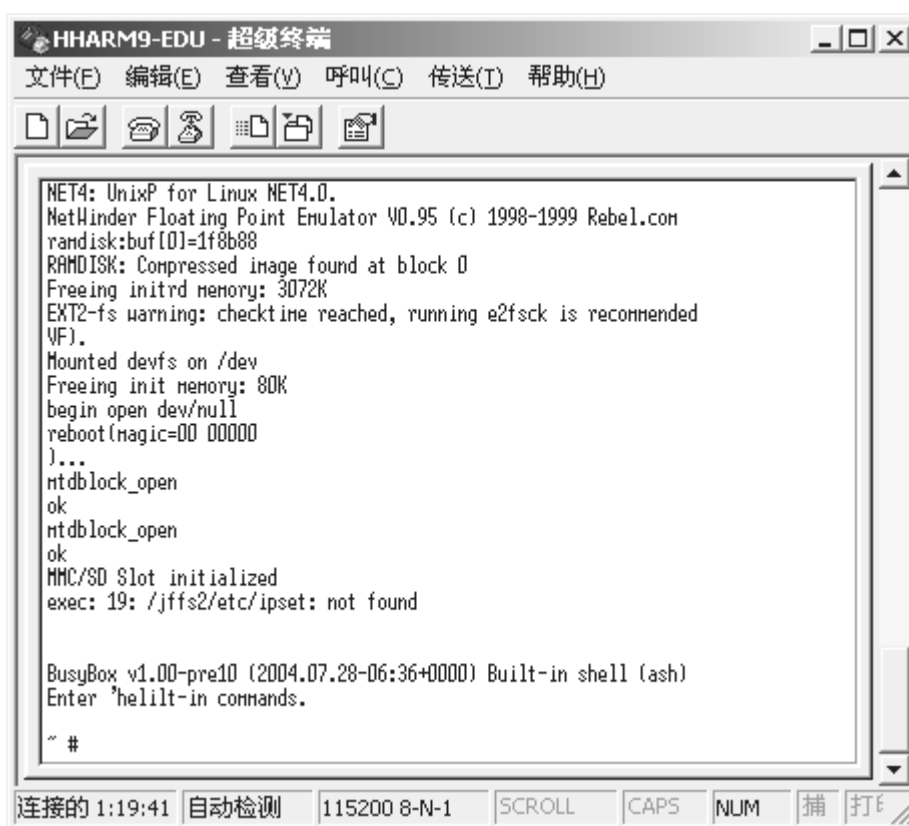
输入连接名称，这个名称可以随便输入，您可以输入“HHARM9-EDU”，然后点击确定按钮。设置连接使用串口 1（COM1），点击确定进入 COM1 的属性设置窗口。



设置串口每秒位数为 115200，数据位 8，奇偶校验无，停止位 1，数据流控无。确定后进入超级终端窗口。



请您用产品附带的串口对接线将开发板与 PC 机的串口 1 连接。接通开发板 12V 电源后超级终端中将会显示开发板的启动信息，并最终进入 Linux 命令提示符。



```
HHARM9-EDU - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

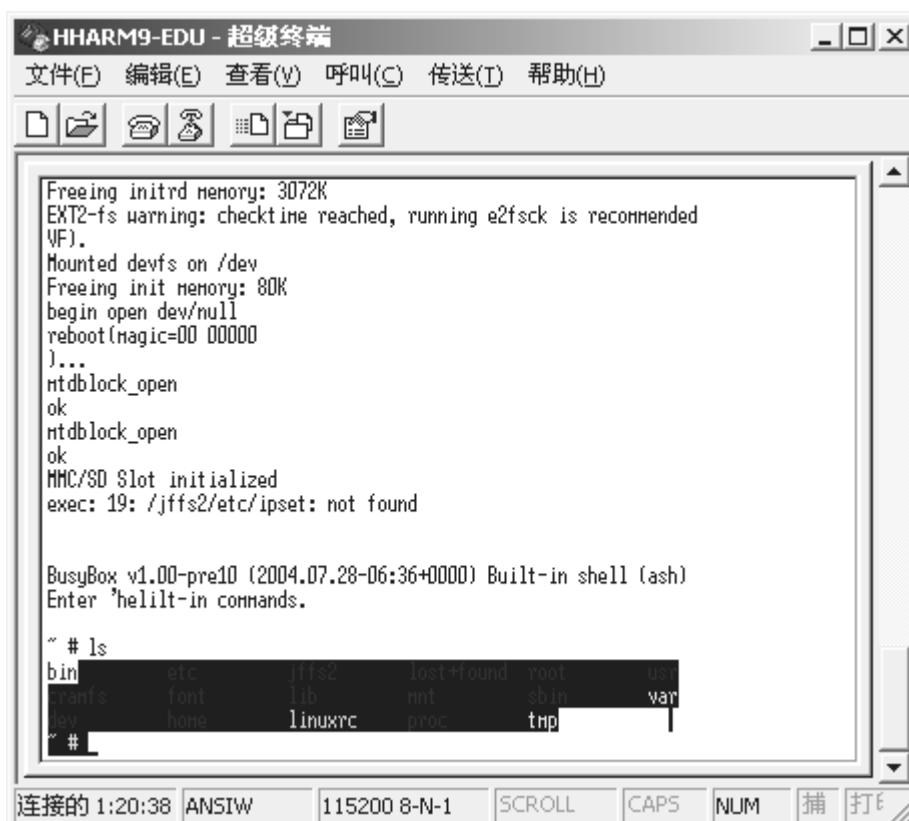
NET4: UnixP for Linux NET4.0.
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
ramdisk:buf[0]=1f8b88
RAMDISK: Compressed image found at block 0
Freeing initrd memory: 3072K
EXT2-fs warning: checktime reached, running e2fsck is recommended
(VF).
Mounted devfs on /dev
Freeing init memory: 80K
begin open dev/null
reboot(magic=00 00000
)...
ntdblock_open
ok
ntdblock_open
ok
MMC/SD Slot initialized
exec: 19: /jffs2/etc/ipset: not found

BusyBox v1.00-pre10 (2004.07.28-06:36+0000) Built-in shell (ash)
Enter 'helilt-in commands.

~ #
```

连接的 1:19:41 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打

如果您在命令提示符下键入 `ls`（回车），就可以看到开发板出厂时烧写的文件目录。



```
HHARM9-EDU - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

Freeing initrd memory: 3072K
EXT2-fs warning: checktime reached, running e2fsck is recommended
(VF).
Mounted devfs on /dev
Freeing init memory: 80K
begin open dev/null
reboot(magic=00 00000
)...
ntdblock_open
ok
ntdblock_open
ok
MMC/SD Slot initialized
exec: 19: /jffs2/etc/ipset: not found

BusyBox v1.00-pre10 (2004.07.28-06:36+0000) Built-in shell (ash)
Enter 'helilt-in commands.

~ # ls
bin      etc      jffs2    lost+found  root    us
+parts  font     lib      mnt        shun    var
+ps     home    linuxrc  proc       tmp

~ #
```

连接的 1:20:38 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打

若按以上操作步骤启动时开发板工作不正常，请立即拨打以下电话与华恒公司联系：0551—5325173 或 0551—5325323，或到我们的网站查询联系方式：

www.hhcn.com

www.hhcn.org

hharm-support@hhcn.com ：技术支持信箱。

我们将为您更换开发板。

【请注意】

请您绝对不要带电拔插串口对接线和 JTAG 烧写器！

2.2 目标板上的目录结构简单介绍

/bin, /sbin

这里分别放着启动的时候所需要的普通程序和系统程序。很多程序在启动以后也很有用，它们放在这个目录下是因为它们会经常被其他程序所调用。

/lib

启动的时候所需要用到的库文件都放在这个目录下。

/dev

这个目录下保存着所有的设备文件。

/home

一般用户的主目录都会放在这个目录下。在开发板启动以后，会存放一些 http 相关的信息，我们做的 boa 在运行时就会用到此目录。

/etc

这里保存着绝大部分的系统配置文件。下面将会列举一下重要的子目录：

/etc/init.d

对于 Debian 来说，这个目录保存着启动描述文件，包括各种模块和服务的加载描述，FTP 和 Telnet 的配置，miniGUI 的配置文件等等，如果不清楚的话，最好不要随便删这里的東西。这里的文件都是对系统进行配置的。

/etc/config

放置了 PPP 拨号的相关脚本。

/usr

存入用户的一些库及其它文件。/usr/local/lib，就是 miniGUI 用到的一些库文件。

/tmp

一般只有启动的时候产生的临时文件才会放在这个地方。我们自己的那些临时文件都放在 /var/tmp。

/mnt

该目录下放着一些用来 mount 其他设备的子目录，作为设备的挂载点。

/proc

里面的文件都是关于当前的系统状态的，包括正在运行的进程，硬件状态，内存使用的多少等等。

/var

存放一些记录信息，例如，PPP 拨号时，可以用 cat /var/log/messages 查看是否建立连接。

/jffs2

JFFS2 文件系统的挂载点。Ramdisk 解开以后只是一个空目录，但是在板子启动以后，可以在此目录中读写文件。

/cramfs

cramfs 文件系统的挂载点，放置了很多大的可执行文件文件，充分利用了 flash 的空间。

【说明】

相关目录的设置以及脚本文件的具体配置，请见后面的实验文档。

2.3 开发环境

2.3.1 操作系统的选择和安装

我们建议您完全安装的 Redhat9.0 操作系统，安装 Redhat9.0 Linux 时，刚开始安装不久，安装向导会弹出对话框询问您安装服务器或工作站等，请选择自定义(**Custom**)；安装过程中可以指定 PC 机上网卡的 IP 地址，由于我们的开发套件在烧写时默认的 IP 为 192.168.2.122，所以建议您的 PC 机也在此网段(192.168.2.X)，IP 地址可以在安装时指定，也可以在 PC 机安装好以后指定网卡；在配置防火墙(**Firewall**)时，选择不安装防火墙(**No Firewall**)，在选择软件 Package 时选择最后一项：**Everything**，即完全安装。完全安装完以后，大概占用 4.8GB 的硬盘空间。

最后会让您选择 Linux 启动以后进入 X 模式还是字符模式，根据自己的爱好决定，进入字符模式时启动花费的时间少一些，即使进入 X windows，也可以按 Ctrl + Alt + Fx(x 在默认情况可以是 1, 2, ……6)，输入安装 RedHat 时输入的用户名（root 或已经建立的其它用户）和密码，即可

进入操作系统的 Shell 提示符，例如像：`[root@localhost root]#`

2.3.2 NFS 和 TFTP 服务器的配置

(1) NFS 的配置：

首先在 REDHAT LINUX PC 机上 shell 提示符`[root@....]#`执行 `setup`，弹出菜单界面后，选中：**System services**，回车进入系统服务选项菜单，在其中选中 `[*]nfs`，然后退出 `setup` 界面返回到命令提示符下。

```
vim /etc/exports
```

将这个默认的空文件修改为只有如下一行内容：

```
/ (rw) //即根目录可读写，/和(rw)之间要要留空格
```

然后保存退出（`:wq`），然后执行如下命令：

```
/etc/rc.d/init.d/nfs restart
```

```
Shutting down NFS mountd: [ OK ]
Shutting down NFS daemon: [ OK ]
Shutting down NFS quotas: [ OK ]
Shutting down NFS services: [ OK ]
Starting NFS services: [ OK ]
Starting NFS quotas: [ OK ]
Starting NFS daemon: [ OK ]
Starting NFS mountd: [ OK ]
```

这样就一切 OK 了！

【注意】

默认情况下 Linux 启动时并不启动 NFS 服务，为了避免每次都要执行以下这一句：

```
/etc/rc.d/init.d/nfs restart
```

可以把此句写入 PC 机的脚本文件`/etc/rc.d/rc.local`中，PC 机启动时会执行此文件，不用每次执行上面的那条命令来启动 NFS。配置完成后，可用如下办法简单测试一下 NFS 是否配置好了：

PC 机自己 `mount` 自己，看是否成功就可以判断 NFS 是否配好了。例如在 PC 机的根目录下执行：（假定 PC 机的 IP 是 192.168.2.32）

```
mount 192.168.2.32:/ /mnt
```

然后到`/mnt/`目录下看是否可以列出所指定的 IP 的机器（可以是本机，当然可以测试其它机器是否可以被 `mount`）根目录（`/`）下的所有文件和目录，可以则说明 `mount` 成功，NFS 配置成功。

(2) TFTP 服务的配置：

TFTP 服务只在第一次使用时需要配置，以后其开机自己运行。

在 PC 机上执行 `setup`，选择 **System services**，将其中的 `tftp` 一项选中（出现 `[*]`表示选中），并去掉 `ipchains` 和 `iptables` 两项服务（即去掉它们前面的*号）。然后还要选择 **Firewall configuration**，选中 **No firewall**。最后，退出 `setup`，执行如下命令以启动 TFTP 服务：

```
service xinetd restart
```

配置完成后，建议简单测试一下 TFTP 服务器是否可用，即自己 `tftp` 自己，例如在 PC 机上执行：

```
cd /
```

```
cp /etc/inittab /tftpboot/ /*拷贝一个文件到/tftpboot 目录下，因为使用 tftp 服务下载时默认的情况下从 PC 机的/tftpboot 目录下载已经存在的文件。如果文件不存在，会提示您没有找到相关
```

文件。*/

```
ftp 192.168.2.23
```

```
ftp> get inittab
```

若出现如下信息：

```
Received 741512 bytes in 0.7 seconds
```

就表示 TFTP 服务器配置成功了。在根目录下就会在刚才下载的 inittab 文件存在了；若弹出信息说：Timed out，则表明未成功，或者用如下命令查看 tftp 服务是否开通：

```
netstat -a|grep tftp
```

若 TFTP 服务器没有配置成功，需要按照上述步骤重新检查一遍。

2.3.3 安装开发环境软件包

请您启动 PC 上的 Redhat9.0 操作系统，并将产品附带的光盘插入光盘驱动器，然后执行以下命令：

```
mount /dev/cdrom /mnt //挂载光盘
```

```
cd /mnt
```

```
./arminst //执行安装程序
```

在安装提示信息显示后，敲入 y，回车。

安装的过程中会显示一些提示信息。通常情况下，您只需按下 y 键后，按回车键即可完成整个开发环境的安装。

安装完光盘提供的源代文件和交叉编译环境以后，执行

```
umount /mnt //卸载光盘
```

现在可以取出光盘了。

2.3.4 光盘目录介绍

一般情况下，我们提供的光盘会有如下目录：

(1) hharm9-edu-Rx.tgz（Rx 表示版本号）：整个软件源代码和编译器的压缩包。

(2) HHARM9-EDU...PDF：开发套件的手册。

(3) Circuit.....：开发套件的底板的 PCB 图和原理图，其它的电路图。

(4) Linuxconf-1.25r7-3.i386.rpm：linuxconf 的 RPM 包，因为在 Redhat 9.0 上面不能在 shell 提示符下执行 linuxconf 命令，在 Redhat7.2 可以执行 linuxconf，所在在我们以前的手册中写的文档是基于 Redhat7.2 版本的，若现有 Redhat 9.0 如果仍想用 linuxconf 命令执行相关的配置，就要安装此 rpm 包能使用，不过现在的很多配置已经可以不用 linuxconf 也可以配置了，例如：NFS 和防火墙的配置，前面已经有配置方法，如果您仍不知道怎么配置，请注意我们网站上的“常见问题解答”：

<http://www.hhcn.com/chinese/embedlinux-res.html>

<http://www.hhcn.com/chinese/hharmfaq.html>

(5) cce.rpm:在字符模式下显示中文的 rpm 包,安装后相当在在 DOS 下面装了 UCDOS，有时我们的安装文件会写一些中文，当执行文件时，若要显示的是中文信息，则看到的是一堆乱码。所以使用了此包，可以看见中文信息。

(6) arminst ：安装脚本文件，进入光盘目录以后，键入./arminst,就会提示您安装上面提到的 tgz

扩展名的文件,并且安装相应的编译器和设置 minicom,此文件可以在 shell 提示符下键入 vi 打开来看看。

2.3.5 HHARM9-EDU 目录结构介绍

安装过我们提供的光盘以下,会在您的 PC 机上建立一个 HHARM9-EDU 的目录。在 shell 提示符下执行 ls 命令,可以显示整个 PC 上的目录结构:

```
[root@.... root]# cd /
[root@.... /]# ls
HHARM9-EDU  boot  lost+found  opt  sbin  usr  dev  home          proc  tftpboot
var  initrd  misc  root  tmp  bin  etc  lib  mnt
```

其中在 PC 机(宿主机)的根目录下安装了 HHARM9-EDU 的目录和 opt 目录,其中 HHARM9-EDU 是开发套件的源代码、驱动、以及相应的应用程序。opt 是 ARM 的编译器存放的目录。进入 HHARM9-EDU 看看。

```
[root@..... /]# cd / HHARM9-EDU
[root@HHARM9-EDU /]# ls
Images  applications  kernel      opt.tgz    gprs-ppp    minirc.dfl  ppcboot-2.0.0
record-image  SJF
```

下面对以上目录作简单介绍:

(1) /HHARM9-EDU/SJF/

JTAG 烧写工具源码目录,在该目录下执行 make,即可生成 JTAG 烧写工具 SJF2410,它就是我们通过 JTAG 烧写 ppcboot 要用到的文件。

(2) /HHARM9-EDU/ppcboot/

bootloader 源码目录,在该目录下简单的 make 即可生成 HHARM9-EDU 的 bootloader — ppcboot.bin,可以通过修改这些源码来修改 bootloader。

【说明】在嵌入式系统中,我们把引导系统的初始化部分的代码统称为 bootloader,相当于 PC 机的 BIOS。但在我们提供的很多套件中,有的引导代码用的是 ppcboot,有的是 u-boot,有的是 bootloader 等等,但实际烧写到 flash 中的文件一般为 ppcboot.bin、u-boot.bin、bootloader.bin 等二进制代码文件。

(3)/HHARM9-EDU/kernel/

Linux 内核源代码目录,以后如果改动内核后,若重新生成新的内核镜像文件 zImage,就可以在此目录下执行 make zImage。

(4) /HHARM9-EDU/application/

应用程序目录,用户可参考此目录下的其它目录,其中添加自己的应用程序。

(5) /HHARM9-EDU/Images/

其下是编译好的映像文件或者可执行文件,其中: zImage 是编译好的内核文件, SJF2410 是编译好的 JTAG 烧写工具, ppcboot 是编译好的引导程序, ramdisk.image.gz 是 ramdisk 文件系统映像, cramfs.img 是 cramfs 文件系统的镜像文件, jffs2.img 是 jffs2 文件系统的镜像文件。如果是用 tftp 下载相关的文件,请把相关文件放到/tftpboot 目录下。

2.3.6 内核编译

如果想编译可以在非 X86 系统的 CPU 中运行的可执行的程序，必须安装适合于此 CPU 体系结构的编译工具，也就是说不同的 CPU 体系结构用的编译器是不一样的，HHARM9-EDU 使用的交叉编译的工具被放置到/opt/host/armv4l 目录下。

GNU 工具集		
armv4l-unknown-linux-gcc	armv4l-unknown-linux-objcopy	Armv4l-unknown-linux-gdb
armv4l-unknown-linux-as	armv4l-unknown-linux-objdump	Armv4l-unknown-linux-gasp
armv4l-unknown-linux-ld	armv4l-unknown-linux-strip	Armv4l-unknown-linux-size
armv4l-unknown-linux-g++	armv4l-unknown-linux-nm	Armv4l-unknown-linux-addr2line
armv4l-unknown-linux-cc1	armv4l-unknown-linux-ar	
armv4l-unknown-linux-cpp	armv4l-unknown-linux-ranlib	
armv4l-unknown-linux-cc1plus	armv4l-unknown-linux-strings	

```
cd /HHARM9-EDU/kernel
```

```
make menuconfig
```

则出现如下界面，可逐项对内核和驱动模块进行选择 and 配置：

可见内核版本为：Linux Kernel v2.4.18-rmk7-pxa1



一些关键的设置:

```
System Type --->
(S3C2410-based) ARM system type
--- S3C2410 Implementation
[*] SMDK (MERI TECH BOARD)
[*] change AIJl
< > S3C2410 USB function support
--- Processor Type
[*] ARM920T CPU idle
[*] ARM920T I-Cache on
[*] ARM920T D-Cache on
[] Force write through caches on ARM920T
[] Support Thumb instructions (experimental)
```

完成自己的设置后,退出,保存配置,然后执行 `make zImage` 即可编译生成自己定制的内核映像文件 `zImage`,此文件会被复制到 `/tftpboot/` 目录下以供烧写。

整个编译过程可如下观察:

```
make zImage &>log
vim log
```

即可看到完整的编译过程,摘录如下:

```
/opt/host/armv4l/bin/armv4l-unknown-linux-ld -p -X -T arch/arm/vmlinux.lds
arch/arm/kernel/head-armv.o arch/arm/kernel/init_task.o init/main.o init/version.o \
--start-group \
arch/arm/kernel/kernel.o arch/arm/mm/mm.o arch/arm/mach-s3c2410/s3c2410.o
kernel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o \
drivers/serial/serial.o drivers/char/char.o drivers/block/block.o drivers/misc/misc.o
drivers/net/net.o drivers/media/media.o drivers/scsi/scsidrv.o drivers/video/video.o
drivers/usb/usbdrv.o \
net/network.o \
arch/arm/nwfpe/math-emu.o arch/arm/lib/lib.a /HHARM9-EDU/kernel/lib/lib.a \
--end-group \
-o vmlinux
```

这里生成的就是 `kernel` 目录下的 `vmlinux` 文件。

看看它内部的段结构和地址:

```
[root@localhost kernel]# /opt/host/armv4l/bin/armv4l-unknown-linux-objdump --header vmlinux
vmlinux: file format elf32-littlearm
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.init	00014000	c0008000	c0008000	00008000	2**5
	CONTENTS, ALLOC, LOAD, CODE					
1	.text	0016f41c	c001c000	c001c000	0001c000	2**5
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
2	.kstrtab	00004ea3	c018b41c	c018b41c	0018b41c	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
3	__ex_table	000009a0	c01902c0	c01902c0	001902c0	2**3

```

CONTENTS, ALLOC, LOAD, DATA
4 __ksymtab    00002538 c0190c60 c0190c60 00190c60 2**2
CONTENTS, ALLOC, LOAD, DATA
5 .data       00011cd3 c0194000 c0194000 00194000 2**5
CONTENTS, ALLOC, LOAD, DATA
6 .bss        000287bc c01a5ce0 c01a5ce0 001a5ce0 2**5
ALLOC
7 .comment    00004718 00000000 00000000 001a5ce0 2**0
CONTENTS, READONLY

```

它的地址分布就是由/HARM9-EDU/kernel/arch/arm/vmlinux.lds 文件指定的:

OUTPUT_ARCH(arm)

ENTRY(stext)

SECTIONS

```

{
    . = 0xC0008000;
    .init : {                /* Init code and data */
        _stext = .;
        __init_begin = .;
            *(.text.init)
        __proc_info_begin = .;
            *(.proc.info)
        __proc_info_end = .;
        __arch_info_begin = .;
            *(.arch.info)
        __arch_info_end = .;
        __tagtable_begin = .;
            *(.taglist)
        __tagtable_end = .;
            *(.data.init)
        . = ALIGN(16);
        __setup_start = .;
            *(.setup.init)
        __setup_end = .;
        __initcall_start = .;
            *(.initcall.init)
        __initcall_end = .;
        . = ALIGN(4096);
        __init_end = .;
        .....
    }
}

```

紧接着的这句话说明了 System.map 文件是如何生成的。

```

/opt/host/armv4l/bin/armv4l-unknown-linux-nm vmlinux | grep -v '\(compiled\)\|(\.o$)\|([aUw]
)\|(\.ng$)\|(LASH[RL]DI)\|' | sort > System.map

```

上面是真正运行的 LINUX 内核，下面则是将这个内核用 gzip 进行压缩后，与 head.S 这个解

压缩代码部分整合到一起生成 zImage:

```
make[1]: Entering directory /HHARM9-EDU/kernel/arch/arm/boot'
```

```
make[2]: Entering directory /HHARM9-EDU/kernel/arch/arm/boot/compressed'
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc      -D__ASSEMBLY__      -D__KERNEL__  
-I/HHARM9-EDU/kernel/include -mapcs-32 -march=armv4 -mno-fpu -msoft-float -traditional -c  
head.S
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -D__KERNEL__ -I/HHARM9-EDU/kernel/include  
-O2 -DSTDC_HEADERS -mapcs-32 -march=armv4 -mtune=arm9tdmi -mshort-load-bytes  
-msoft-float -fpic -D__KERNEL__ -I/HHARM9-EDU/kernel/include -c -o misc.o misc.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc      -D__ASSEMBLY__      -D__KERNEL__  
-I/HHARM9-EDU/kernel/include -mapcs-32 -march=armv4 -mno-fpu -msoft-float      -c -o  
head-s3c2410.o head-s3c2410.S
```

//请注意下面的几句话:

//先将 ELF 格式的 vmlinux 转换为二进制格式的, 尺寸大大缩水, 由原来

//的 2M 多变为 700~800K 字节, 并用 piggy 作为临时文件名

```
/opt/host/armv4l/bin/armv4l-unknown-linux-objcopy -O binary -R .note -R .comment -S  
/HHARM9-EDU/kernel/vmlinux piggy
```

//对内核进行 gzip 压缩

```
gzip -9 < piggy > piggy.gz
```

//再转换为.o 格式的文件

```
/opt/host/armv4l/bin/armv4l-unknown-linux-ld -r -o piggy.o -b binary piggy.gz
```

```
rm -f piggy piggy.gz
```

//将解压缩部分代码和压缩内核链接为一个新的 vmlinux

```
/opt/host/armv4l/bin/armv4l-unknown-linux-ld -p -X -T vmlinux.lds head.o misc.o head-s3c2410.o  
piggy.o /opt/host/armv4l/lib/gcc-lib/armv4l-unknown-linux/2.95.2/soft-float/libgcc.a -o vmlinux
```

```
make[2]: Leaving directory `/HHARM9-EDU/kernel/arch/arm/boot/compressed'
```

//再次将解压缩代码部分的 ELF 格式转换为二进制文件格式, 即最终的

//zImage 文件, 并复制到tftpboot 目录下以供 TFTP 下载烧写。

```
/opt/host/armv4l/bin/armv4l-unknown-linux-objcopy -O binary -R .note -R .comment -S  
compressed/vmlinux zImage
```

```
cp -f zImage /tftpboot/
```

2.4 软件下载烧写说明

2.4.1 烧写过程

- (1) 烧写 ppcboot.bin (在/HHARM9-EDU/Images/目录下);
- (2) 按复位键重启板子, 在 minicom 中应该有启动信息;
- (3) 烧写内核 zImage;
- (4) 烧写文件系统 ramdisk.image.gz;
- (5) 烧写 cramfs 文件系统和 JFFS2 文件系统;

【说明】和烧写 ramdisk.image.gz 一样，先下载，再烧写，只是烧写到 flash 中的位置不一样。所以，后面没有列出 cramfs 和 JFFS2 文件系统烧写的相关信息。

【解释】minicom: linux 下的一个终端程序，就像 windows 中的超级终端一样，只需要在 Shell 提示下键入 minicom 就可以了，更改相应的设置可以按 Ctrl+O 操作。

在第 1 步：当执行 ./SJF2410 /f:ppcboot.bin 时若出现以下信息

```
+-----+
```

```
| SEC JTAG FLASH(SJF) v 0.3 |
```

```
| (S3C2410X & SMDK2410 B/D) |
```

```
+-----+
```

```
Usage: SJF /f:<filename> /d=<delay>
```

```
> S3C2410X(ID=0x0032409d) is detected.
```

```
[SJF Main Menu]
```

```
0:K9S1208 program 1:28F128J3A program 2:Memory Rd/Wr 3:Exit
```

```
Select the function to test:
```

则正常，尤其是其中这一句> S3C2410X(ID=0x0032409d) is detected.可以判断 jtag 是否已经加上，若此句变成 ERROR: No CPU is detected(ID=0xffffffff).则有可能是 jtag 没有插好，或是插反了，或是 jtag 坏了。

【注意】jtag 线有凸起的那边朝向核心板的外面插到核心板的 jtag 插针上。如果使用了 20 针->10 针的转接头，当插上 JTAG 到核心板上，留出 PCB 板多的一方朝向核心板的外面。其实如果插反了，也不要紧，只是提示您没有找到 CPU，**请不要带电拔插 JTAG。**

建议有问题时，先试试烧写刚安装好的光盘上提供给你的 ppcboot.bin 这个文件，接下来按照手册上烧写,大致是重新启动开发板后，在 minicom 中分步骤键入以下命令：

【说明】如果把开发板和 PC 机的以太网口直接相连，要有交叉对接网线,如果要用普通的网线，则要经过 HUB，建议用对接网线。

```
tftp 0x30008000 zImage //从指定的 TFTP 服务器上下载 zImage
```

```
fl 0x1040000 0x30008000 0xc0000 //烧写刚下载的文件到指定的位置。
```

```
tftp 30800000 ramdisk.image.gz
```

```
fl 1140000 30800000 0x1a0000
```

```
tftp 0x30008000 cramfs.img
```

```
fl 1540000 30008000 200000
```

```
tftp 0x30008000 jffs2.img
```

```
fl 1940000 30008000 200000
```

其中的 0xc0000 和 0x1a0000 等 fl 命令的最后一个参数，指的是烧写时留给相应文件的空间大小，只要比所烧写的文件大且是后面有 4 个 0 的整数即可,当用 tftp 命令下载相关文件时，会提示您此文件的十六进制大小。

烧写的过程出现的信息如下：

```
PPCBoot 2.0.0 (Dec 15 2003 - 09:41:17)
```

```
PPCBoot code: 33F00000 -> 33F15118 BSS: -> 33F18318
```

```
DRAM Configuration:
```

```
Bank #0: 30000000 64 MB
```

```
Flash: 16 MB
```



```

start linux now(y/n):
SMDK2410 # tftp 0x30008000 zImage
<DM9000> I/O: 8000300, VID: 90000a46
NetOurIP =c0a80278
NetServerIP = c0a8027a      //【注意】这里 c0a8027a 就是 192.168.2.122，即要求 LINUX PC
                             的 IP 地址必须为 192.168.2.122，这个地址是可以修改修改的，参见下
                             面介绍。
NetOurGatewayIP = c0a80201
NetOurSubnetMask = fffff00
ARP broadcast 1
ARP broadcast 2
TFTP from server 192.168.2.122; our IP address is 192.168.2.120
Filename 'zImage'.
Load address: 0x30008000
Loading:
#####
#####
done
Bytes transferred = 753148 (b7dfc hex) //文件的大小为 b7dfc，十六进制值
SMDK2410 # fl 0x1040000 0x30008000 0xc0000 //0xc0000 比 b7dfc 大
start_sect=0x2,end_sect=0x7
*****erase sector 0x2*****
*****erase sector 0x3*****
*****erase sector 0x4*****
*****erase sector 0x5*****
*****erase sector 0x6*****
*****erase sector 0x7*****
-----program sector 0x2-----
-----program sector 0x3-----
-----program sector 0x4-----
-----program sector 0x5-----
-----program sector 0x6-----
-----program sector 0x7-----
SMDK2410 # tftp 30800000 ramdisk.image.gz
<DM9000> I/O: 8000300, VID: 90000a46
NetOurIP =c0a80278
NetServerIP = c0a8027a 【注意】这里 c0a8027a 就是 192.168.2.122
NetOurGatewayIP = c0a80201
NetOurSubnetMask = fffff00
ARP broadcast 1
ARP broadcast 2
TFTP from server 192.168.2.122; our IP address is 192.168.2.120
Filename 'ramdisk.image.gz'.
Load address: 0x30800000

```

Loading:

```
#####
#####
#####
#####
#####
```

done

Bytes transferred = 1672929 (1986e1 hex)

SMDK2410 # fl 1140000 30800000 1a0000

start_sect=0xa,end_sect=0x16

```
*****erase sector 0xa*****
*****erase sector 0xb*****
*****erase sector 0xc*****
*****erase sector 0xd*****
*****erase sector 0xe*****
*****erase sector 0xf*****
*****erase sector 0x10*****
*****erase sector 0x11*****
*****erase sector 0x12*****
*****erase sector 0x13*****
*****erase sector 0x14*****
*****erase sector 0x15*****
*****erase sector 0x16*****
```

```
-----program sector 0xa-----
-----program sector 0xb-----
-----program sector 0xc-----
-----program sector 0xd-----
-----program sector 0xe-----
-----program sector 0xf-----
-----program sector 0x10-----
-----program sector 0x11-----
-----program sector 0x12-----
-----program sector 0x13-----
-----program sector 0x14-----
-----program sector 0x15-----
-----program sector 0x16-----
```

SMDK2410 #

【说明】 用户可以修改 ppcboot 中指定的 TFTP 服务器和开发板的 IP 地址。

vim ppcboot-2.0.0/include/configs/smdk2410.h

修改如下行:

```
#define CONFIG_SERVERIP 192.168.2.122 //122.2.168.192
```

然后重新编译 ppcboot，重新烧写新的 ppcboot.bin，就可从用户指定的 IP 下载烧写了。

当所要烧写的文件烧写完了以后，按一下复位键就可以看一下启动以后的效果了，启动信息如下：

PPCBoot 2.0.0 (Jan 7 2004 - 17:21:13)

PPCBoot code: 33F00000 -> 33F15294 BSS: -> 33F18494

DRAM Configuration:

Bank #0: 30000000 64 MB

Flash: 16 MB

start linux now(y/n):## Booting image at 30008000 ...

copy kernel done

copy ramdisk done

Setup linux parameters at 0x30000100

cpu_arm920_cache_clean_invalidate_all finished

before call_linux

Uncompressing Linux.....Linux version 2.4.18-rmk7-pxa1
(root@work182) (gcc version 2.95.2 20000516 (rel4CPU: ARM/CIRRUS Arm920Tsid(wb) revision
0

Machine: Samsung-SMDK2410

On node 0 totalpages: 16384

zone(0): 16384 pages.

zone(1): 0 pages.

zone(2): 0 pages.

Kernel command line: initrd=0x30800000,0x440000 root=/dev/ram init=/linuxrc co0DEBUG:
timer count 15844

Console: colour dummy device 80x30

Calibrating delay loop... 101.17 BogoMIPS

Memory: 64MB = 64MB total

Memory: 62252KB available (1782K code, 352K data, 80K init)

Dentry-cache hash table entries: 8192 (order: 4, 65536 bytes)

Inode-cache hash table entries: 4096 (order: 3, 32768 bytes)

Mount-cache hash table entries: 1024 (order: 1, 8192 bytes)

Buffer-cache hash table entries: 4096 (order: 2, 16384 bytes)

Page-cache hash table entries: 16384 (order: 4, 65536 bytes)

POSIX conformance testing by UNIFIX

Linux NET4.0 for Linux 2.4

Based upon Swansea University Computer Society NET3.039

Initializing RT netlink socket

CPU clock = 202.800000 Mhz, HCLK = 101.400000 Mhz, PCLK = 50.700000 Mhz

Initializing S3C2410 buffer pool for DMA workaround

Starting kswapd

Journalled Block Device driver loaded

devfs: v1.10 (20020120) Richard Gooch (rgooch@atnf.csiro.au)

devfs: boot_options: 0x1

JFFS2 version 2.1. (C) 2001 Red Hat, Inc., designed by Axis Communications AB.

ttyS%d0 at I/O 0x50000000 (irq = 52) is a S3C2410

ttyS%d1 at I/O 0x50004000 (irq = 55) is a S3C2410

ttys%d2 at I/O 0x50008000 (irq = 58) is a S3C2410
Console: switching to colour frame buffer device 30x20
Installed S3C2410 frame buffer
pty: 256 Unix98 ptys configured
TouchPanel: digi_init start...
TouchPanel:set interrupt
TouchPanel:spi init ok
X1227 RTC driver installed OK
s3c2410-ts initialized
block: 128 slots per queue, batch=32
RAMDISK driver initialized: 16 RAM disks of 12288K size 1024 blocksize
Uniform Multi-Platform E-IDE driver Revision: 6.31
ide: Assuming 50MHz system bus speed for PIO modes; override with idebus=xx
ioremap:c4910000
ide0: CPCI405 IDE interface
IDE: waiting for drives to settle...
IDE: waiting for drives to settle...
IDE: waiting for drives to settle...
IDE: waiting for drives to settle...
HHTech DM9000 eth0 I/O: c4920300,VID: 90000a46,MAC: 00:13:F6:6C:87:89:
SLIP: version 0.8.4-NET3.019-NEWTTY (dynamic channels, max=256).
CSLIP: code copyright 1989 Regents of the University of California.
SLIP linefill/keepalive option.
loop: loaded (max 8 devices)
PPP generic driver version 2.4.1
SCSI subsystem driver Revision: 1.00
request_module[scsi_hostadapter]: Root fs not mounted
begin init_physmap 0x1140000
physmap flash device: 9c0000 at 540000
Using buffer write method
mtd: Giving out device 0 to Physically mapped flash
s3c2410 flash device: 1600000 at 1000000
Search for id:(89 18) interleave(1) type(2)
Search for id:(89 18) interleave(1) type(2)
Search for id:(89 18) interleave(1) type(2)
Search for id:(89 18) interleave(2) type(1)
Search for id:(89 18) interleave(2) type(1)
Search for id:(89 18) interleave(2) type(1)
Search for id:(89 00) interleave(2) type(2)
Search for id:(89 00) interleave(2) type(2)
Search for id:(89 00) interleave(2) type(2)
JEDEC: Found no s3c2410 flash device device at location zero
Using buffer write method
Creating 5 MTD partitions on "s3c2410 flash device":

0x00000000-0x00040000 : "reserved for bootloader"
mtd: Giving out device 1 to reserved for bootloader
0x00040000-0x00140000 : "reserved for kernel"
mtd: Giving out device 2 to reserved for kernel
0x00140000-0x00540000 : "reserved for ramdisk"
mtd: Giving out device 3 to reserved for ramdisk
0x00800000-0x00f00000 : "jffs2(8M)"
mtd: Giving out device 4 to jffs2(8M)
0x00540000-0x00800000 : "cramfs(2.75M)"
mtd: Giving out device 5 to cramfs(2.75M)
usb.c: registered new driver usbdevfs
usb.c: registered new driver hub
usb-ohci.c: USB OHCI at membase 0xe9000000, IRQ 26
usb.c: new USB bus registered, assigned bus number 1
Product: USB OHCI Root Hub
SerialNumber: e9000000
hub.c: USB hub found
hub.c: 2 ports detected
Initializing USB Mass Storage driver...
usb.c: registered new driver usb-storage
USB Mass Storage support registered.
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 4096 bind 4096)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
ramdisk:buf[0]=1f8b88
RAMDISK: Compressed image found at block 0
Freeing initrd memory: 3072K
EXT2-fs warning: checktime reached, running e2fsck is recommended
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
NetWinder Floating Point Emulator V0.95 (c) 1998-1999 Rebel.com
ramdisk:buf[0]=1f8b88
RAMDISK: Compressed image found at block 0
Freeing initrd memory: 3072K
EXT2-fs warning: checktime reached, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem).
Mounted devfs on /dev
Freeing init memory: 80K
begin open dev/null
reboot(magic=00000000
)...
mtdblock_open

```
ok
mtddblock_open
ok
MMC/SD Slot initialized
exec: 19: /jffs2/etc/ipset: not found
BusyBox v1.00-pre10 (2004.07.28-06:36+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

```
~ #
```

JFFS2 目录信息

```
~ # ls
```

```
bin          etc          jffs2        lost+found   root         usr
cramfs       font         lib          mnt          sbin         var
dev          home         linuxrc      proc         tmp
```

```
~ # cd jffs2/
```

```
/jffs2 # ls
```

```
asp_test  hhcn      hn          home         ipset        myapp
```

```
/jffs2 # cd home/httpd/
```

```
home/httpd/cgi-bin/  home/httpd/html/    home/httpd/log/
```

```
/jffs2 # cd home/httpd/
```

```
/jffs2/home/httpd #
```

Cramfs 目录信息

```
~ # ls
```

```
bin          etc          jffs2        lost+found   root         usr
cramfs       font         lib          mnt          sbin         var
dev          home         linuxrc      proc         tmp
```

```
~ # cd jffs2/
```

```
/jffs2 # ls
```

```
asp_test  hhcn      hn          home         ipset        myapp    wpq
```

```
/jffs2 # cd /cramfs/
```

```
/cramfs # ls
```

```
etc      modules  sbin
```

```
/cramfs # cd sbin/
```

```
/cramfs/sbin # ls
```

```
boa          ifuser          progress.png    server
chat          in.telnetd      progressbar     tip
depmod        inetd           pump            treeview
diald         lnx_init        pump.sh         ttytest
fuser         pppd            pure-ftpd
ifport        progress-bk.png recorder
```

```
/cramfs/etc # cd ../modules/
```

```
/cramfs/modules # ls
```

```
2410audio.o  digi.o          mmcsd_core.o  mmcsd_disk.o  mmcsd_slot.o
/cramfs/modules #
```

其它信息情况

~ # ps

PID	Uid	VmSize	Stat	Command
1	root	520	S	init
2	root		SW	[keventd]
3	root		SWN	[ksoftirqd_CPU0]
4	root		SW	[kswapd]
5	root		SW	[bdflush]
6	root		SW	[kupdated]
7	root		SW	[mtdblockd]
8	root		SW	[khubd]
31	root		SWN	[jffs2_gcd_mtd4]
39	root	600	S	/cramfs/sbin/boa
41	root	504	S	syslogd
43	root	528	S	/cramfs/sbin/inetd
55	root	664	S	-sh
57	root	584	R	ps

~ # cat /proc/interrupts

0:	1	DM9000 device
1:	0	digi
13:	0	DMA timer
14:	18700	timer
21:	0	SDI
26:	0	usb-ohci
46:	0	SD CD
52:	182	serial_s3c2410_rx
53:	650	serial_s3c2410_tx
54:	0	serial_s3c2410_err
Err:	0	

~ #

2.4.2 内核下载至 RAM 中直接启动

ppcboot 支持直接在 ram 中启动内核，这一功能可以极大的方便内核调试。内核和 ramdisk 映像下载到内存中后，可以使用 go 命令直接启动刚下载的内核，而不必烧写到 Flash 中。其使用步骤如下：

```
SDMK2410# tftp 30008000 zImage
SMDK2410# 30800000 ramdisk.image.gz
SMDK2410# go 30008000
```

即可直接启动刚下载的内核。

2.5 HHARM9-EDU 平台硬件系统

HHARM9-EDU 平台由核心板和底板（外设板或称基本板）组成，核心板上集成 Samsung S3C2410 处理器，16M 的 FLASH 和 64M SDRAM，为您的应用研发提供了足够的空间。Samsung's S3C2410 16/32-bit RISC 是一款高性价比，低功耗，体积小，高性能，高集成度的微处理器，采用 203MHz 的 ARM920T 内核。集成了 16KB 指令缓存和 16KB 数据缓存，利用 MMU 实现对虚拟内存的管理，LCD 控制器支持 STN 屏或 FTT 屏，支持 NAND flash。还拥有以下特性。

- (1) ARM920T 嵌入式处理器内核，主频可达 203MHz；
- (2) 扩展总线最大频率 100MHz；
- (3) 32 位数据,27 位外部地址线；
- (4) 完全静态设计(0-203M)；
- (5) 存储控制器(八个存储体)；
- (6) 包含 RAM(SDRAM)控制器,NAND 控制器；
- (7) 复位时引导芯片选择(8-, 16-比特存储或 NAND 可供选择)；
- (8) 3 个 UART，Supports IrDA 1.0；\
- (9) 4 个带有 PWM 的 16 位定时器；
- (10) 4 个 DMA 通道；（支持外设 DMA）；
- (11) 多达 55 个中断源的中断控制器；
- (12) 8 通道，500KSPS，10-bit ADC；
- (13) 看门狗；
- (14) IIS 音频接口；
- (15) 两个 USB 口；
- (16) IIC-Bus 接口；
- (17) 两个串行外围接口电路（SPI）
- (18) SD 卡接口；
- (19) RTC；

底板上则提供以下外设接口：

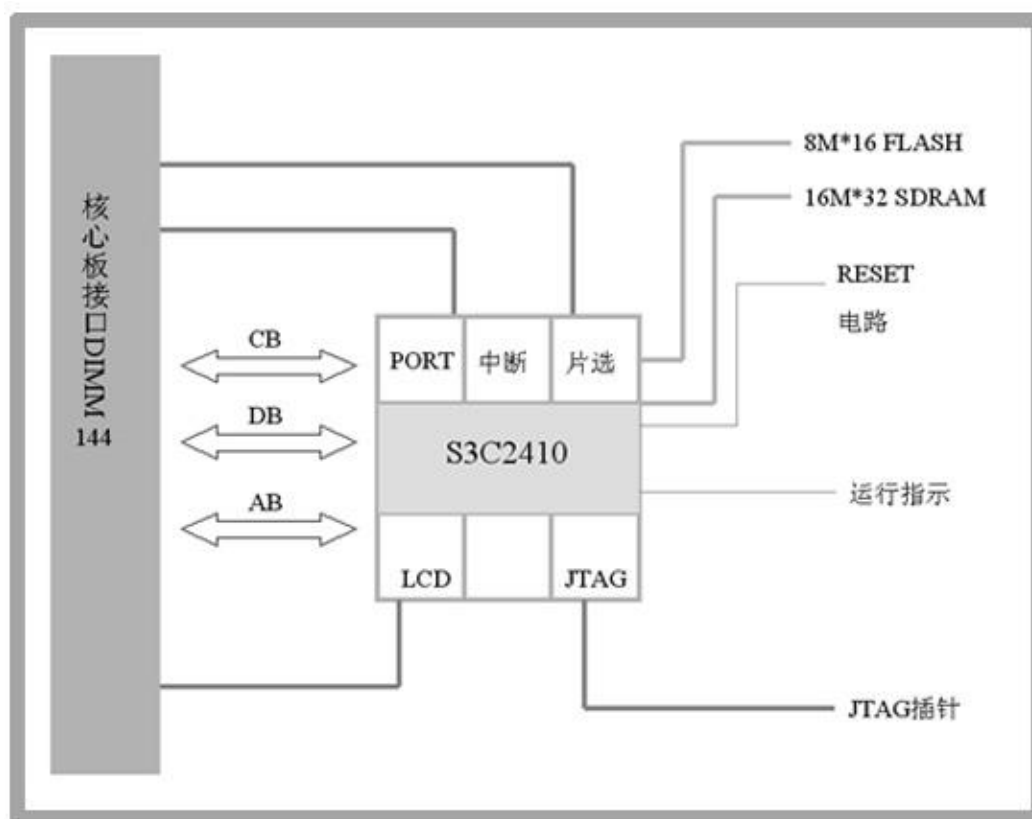
- (1) 10M/100M 自适应以太网接口一个；
- (2) 四线 RS-232 串口(COM1)一个；
- (3) 四线 RS232/RS485/GPRS 串口一个（复用）；
- (4) IDE/CF 卡接口；
- (5) SD/MMC 卡接口；
- (6) USB HOST 接口一个；
- (7) USB Device 接口一个；
- (8) TFT LCD 接口；
- (9) 触摸屏接口；
- (10) 音频输入输出接口，麦克风接口；
- (11) A/D，D/A 接口；
- (12) PS/2 接口；
- (13) 并口（待定）；
- (14) CAN 总线接口等。

核心板和底板配合即构成一个完整的应用系统。系统具有体积小、耗电低、处理能力强等特点，能够装载和运行嵌入式 Linux 操作系统。用户可以在这个系统平台上进行自主软件开发。套件中提

供底板硬件电路图及硬件设计文档，极大的方便了用户进行硬件扩展开发。

HHARM9-EDU 套件提供完备的嵌入式 Linux 开发环境及丰富的开发调试工具软件。

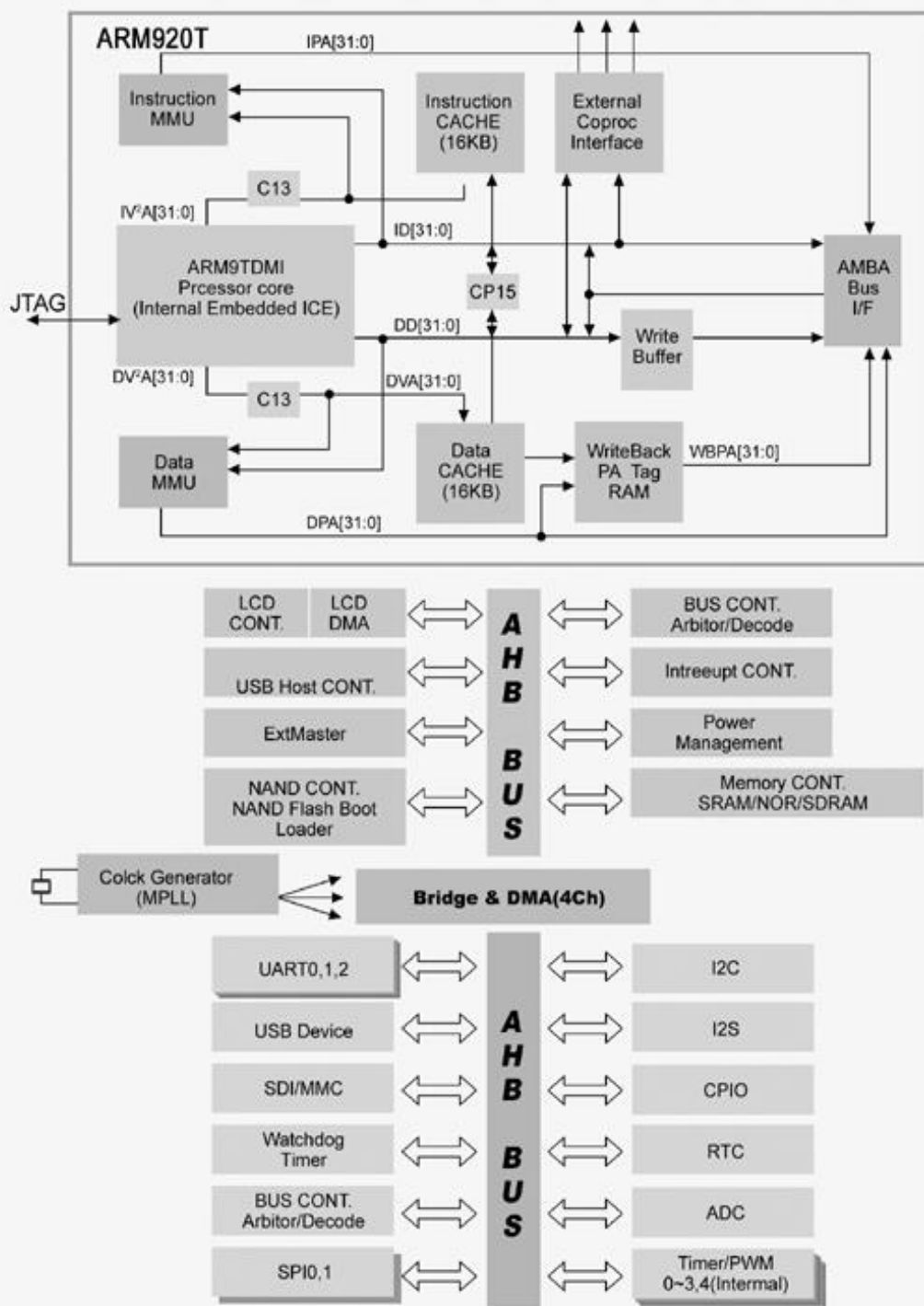
2.5.1 核心板功能模块结构图



S3C2410 功能模块:

HH Tech.
华恒科技

S3C2410 体系结构

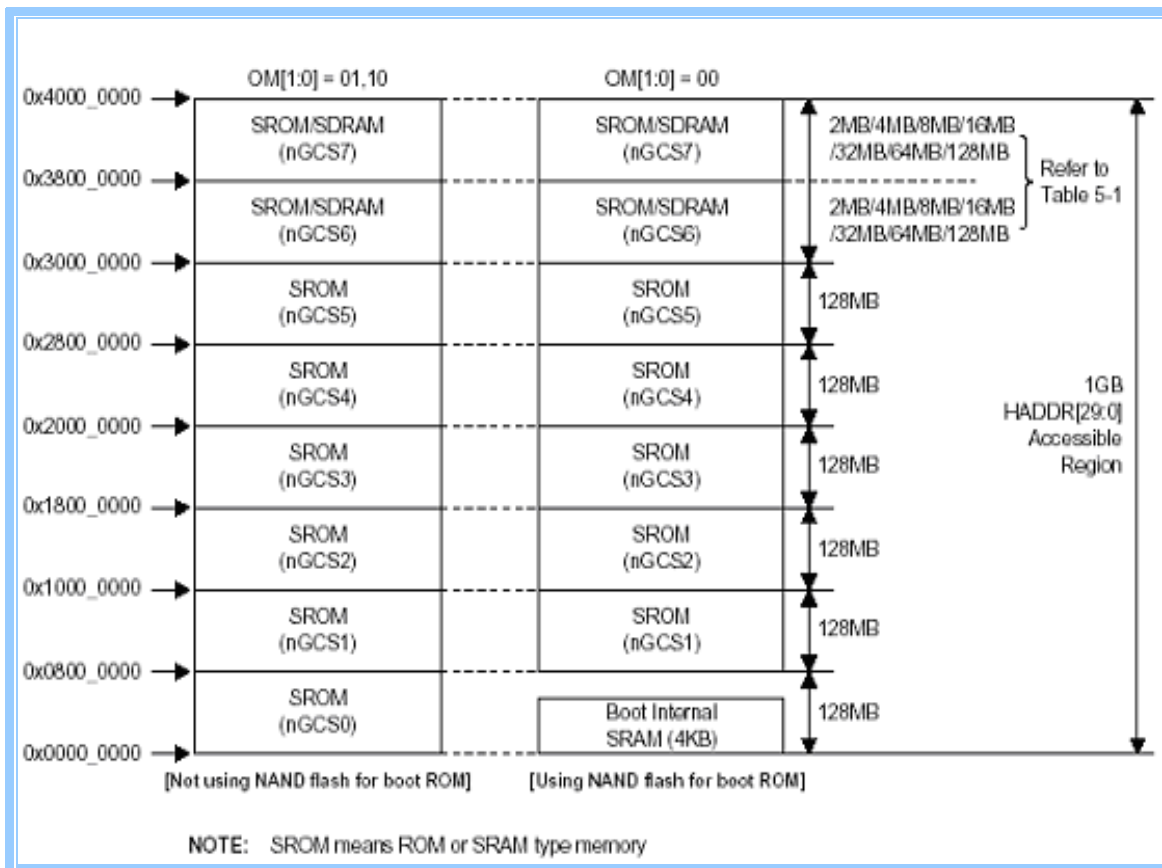


2.5.2 内存部分的构成

1 片 8 M×16 位数据宽度的 FLASH，共 16M 字节 Flash（intel 28F128J3C，如有不同型号，则是完全兼容的器件），速度 150ns； 两片 16M×16 位数据宽度的 SDRAM（HY57V561620B T，如有不同型号，则是完全兼容的器件）构成，共 64M SDRAM。

2.5.3 片选

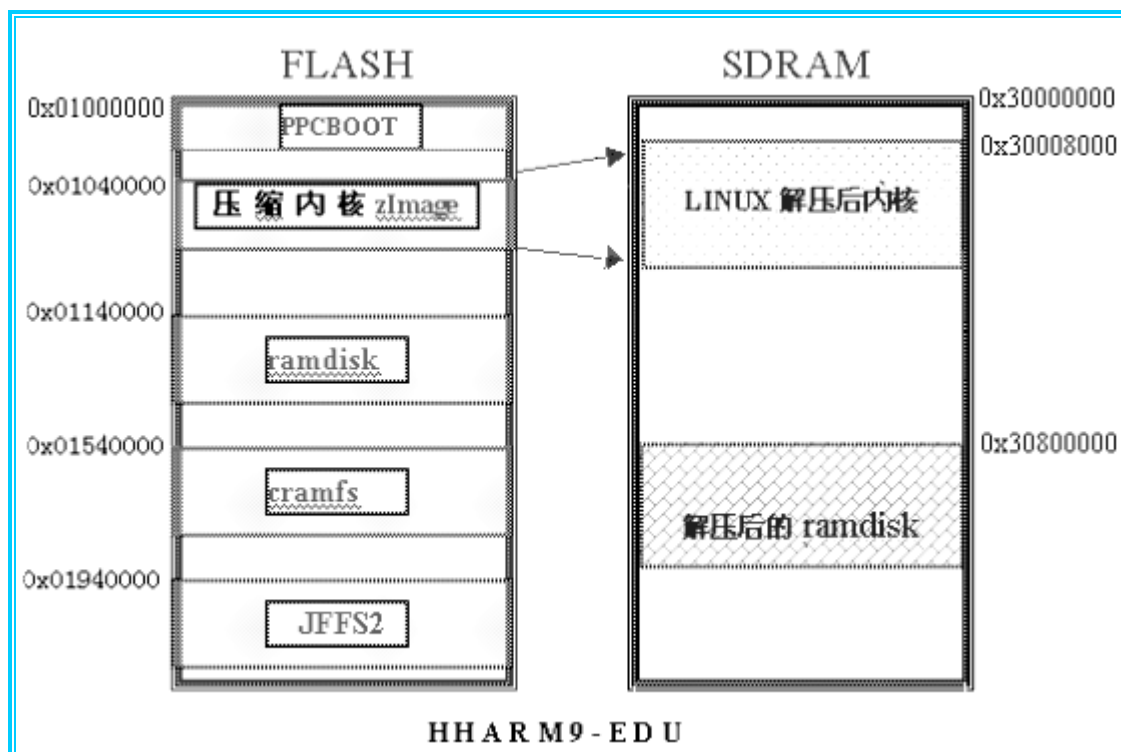
S3C2410 提供 8 路片选，nGCSn[0~7]，每个片选都指定了固定的地址，每个片选固定间隔为 128M 字节。具体参见 CPU 手册 P5-2。



HHARM9-EDU 开发板内存由两片 16M ×16 位数据宽度的 SDRAM 构成，两片拼成 32 位模式，公用 nGCS6。共 64M RAM。起始地址：0x30000000。

nGCS0 接的是一片 8M ×16 位数据宽度的 INTEL E28F128 FLASH。起始地址：0x10000000。按照 S3C2410 处理器手册，NOR FLASH 安装在 BANK0，地址应该为 0，但由于 2410 中地址是循环映射的，0x01000000 就是 0 地址，也就是 0x10000000。其中内核 zImage 烧写在地址 0x01040000 开始处，根文件系统 RAMDISK 烧在 0x01140000 地址处。

下面给出开发板上的地址空间分布：MEMORY MAP



INTEL E28F128J3A-150 FLASH 的单片 16M 字节，共 128 个扇区，每个扇区都是 128K 字节大小，均匀分布。

2.5.4 中断

S3C2410 可处理 56 路中断，其中 24 路为外部中断 EINTn，具体可参见 PDF 手册 P14-3。

Sources	Descriptions	Arbiter Group
INT_ADC	ADC EOC and Touch interrupt (INT_ADC/INT_TC)	ARB5
INT_RTC	RTC alarm interrupt	ARB5
INT_SPI1	SPI1 interrupt	ARB5
INT_UART0	UART0 Interrupt (ERR, RXD, and TXD)	ARB5
INT_IIC	IIC interrupt	ARB4
INT_USBH	USB Host interrupt	ARB4
INT_USBD	USB Device interrupt	ARB4
Reserved	Reserved	ARB4
INT_UART1	UART1 Interrupt (ERR, RXD, and TXD)	ARB4
INT_SPI0	SPI0 interrupt	ARB4
INT_SDI	SD1 interrupt	ARB 3
INT_DMA3	DMA channel 3 interrupt	ARB3
INT_DMA2	DMA channel 2 interrupt	ARB3
INT_DMA1	DMA channel 1 interrupt	ARB3
INT_DMA0	DMA channel 0 interrupt	ARB3
INT_LCD	LCD interrupt (INT_FrSyn and INT_FiCnt)	ARB3
INT_UART2	UART2 Interrupt (ERR, RXD, and TXD)	ARB2
INT_TIMER4	Timer4 interrupt	ARB2
INT_TIMER3	Timer3 interrupt	ARB2
INT_TIMER2	Timer2 interrupt	ARB2
INT_TIMER1	Timer1 interrupt	ARB 2
INT_TIMER0	Timer0 interrupt	ARB2
INT_WDT	Watch-Dog timer interrupt	ARB1
INT_TICK	RTC Time tick interrupt	ARB1
nBATT_FLT	Battery Fault interrupt	ARB1
Reserved	Reserved	ARB1
EINT8_23	External interrupt 8 – 23	ARB1
EINT4_7	External interrupt 4 – 7	ARB1
EINT3	External interrupt 3	ARB0
EINT2	External interrupt 2	ARB0
EINT1	External interrupt 1	ARB0
EINT0	External interrupt 0	ARB0

板上扩展的外设接口占用片选、中断情况：

接口	占用片选	占用中断
DM9000	nGCS1	EINT0
PS/2 键盘	nGCS4	EINT4
IDE 硬盘	nGCS3	EINT3
CAN Bus	GPG14	EINT7
LED	nGCS2	
A/D、D/A	nGCS2	
步进电机	nGCS2	
Esc 按键		EINT5

在板子的 minicom 终端可以通过如下命令查看板上的中断信息：

```
# cat proc/interrupts
```

0:	1	DM9000 device
13:	0	DMA timer
14:	1486	timer
26:	36	usb-ohci
52:	31	serial_s3c2410_rx
53:	100	serial_s3c2410_tx
54:	0	serial_s3c2410_err
Err:	0	

2.5.5 GPIO

S3C2410 提供 117 路复用的 IO 口线，分为如下端口进行管理：

- Port A (GPA): 23路输出口线
- Port B (GPB): 11路输入/输出口线
- Port C (GPC): 16路输入/输出口线
- Port D (GPD): 16路输入/输出口线
- Port E (GPE): 16路输入/输出口线
- Port F (GPF): 8路输入/输出口线
- Port G (GPG): 16路输入/输出口线
- Port H (GPH): 11路输入/输出口线

核心板上将这些复用的信号引脚中未被占用的全部引到底板上，具体请参见产品光盘中的 PROTEL 格式的电路原理图和 PCB。

2.5.6 总线

- (1) S3C2410 是内部 32 位地址，外部 27 位地址，数据总线宽度 32 位。203M 的主频，100M 的总线速度。
- (2) 若外接 8 位或 16 位数据宽度的外设芯片，与 CPU 相接时，HHARM9-EDU 的数据总线宽度是可配置的，可分别配为 32 位、16 位或 8 位模式。设置是在 BWSCON 中的 DW 位（参见 S3C2410X User's Manual 的 Memory Controller）实现的。在给外设分配片选时，设置好它的 BWSCON 中的这两位，在访问它的地址时就可以改变数据宽度。

16 位数据宽度时，是低 16 位数据线有效；

8 位模式时，是最低 8 位数据线有效。

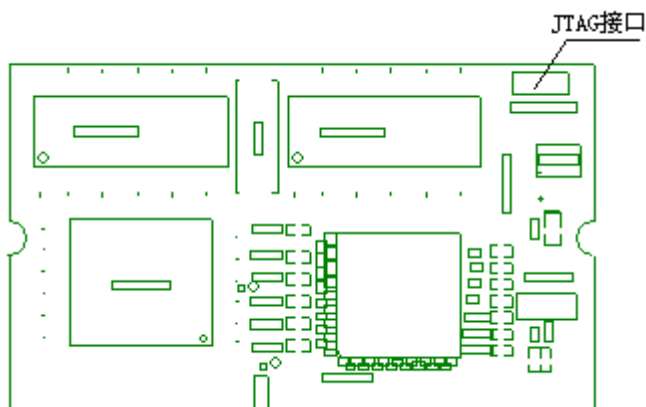
【注意】

启动时这个对 CS0 是无效的，因为 CS0 是接存放启动代码的存储器片选，一般都是 FLASH，在 CPU 刚加电时，这时的数据宽度就无法用 BWSCON 来设置了，就只有硬件实现了：由复位后硬件配置决定数据的宽度，复位默认为 0x00000000。

两片 SDRAM 为 32 位寻址，但两片的数据总线分别接 HHARM9-EDU 高 16 位和低 16 位数据总线，这样拼成 32 位 SDRAM 使用，所以两片 SDRAM 共享一个 CS。而一片 FLASH 则固定为 16 位数据读写访问模式，它就只接 HHARM9-EDU 的低 16 位数据总线。

2.5.7 外设接口图

(1) 核心板正面俯视图:

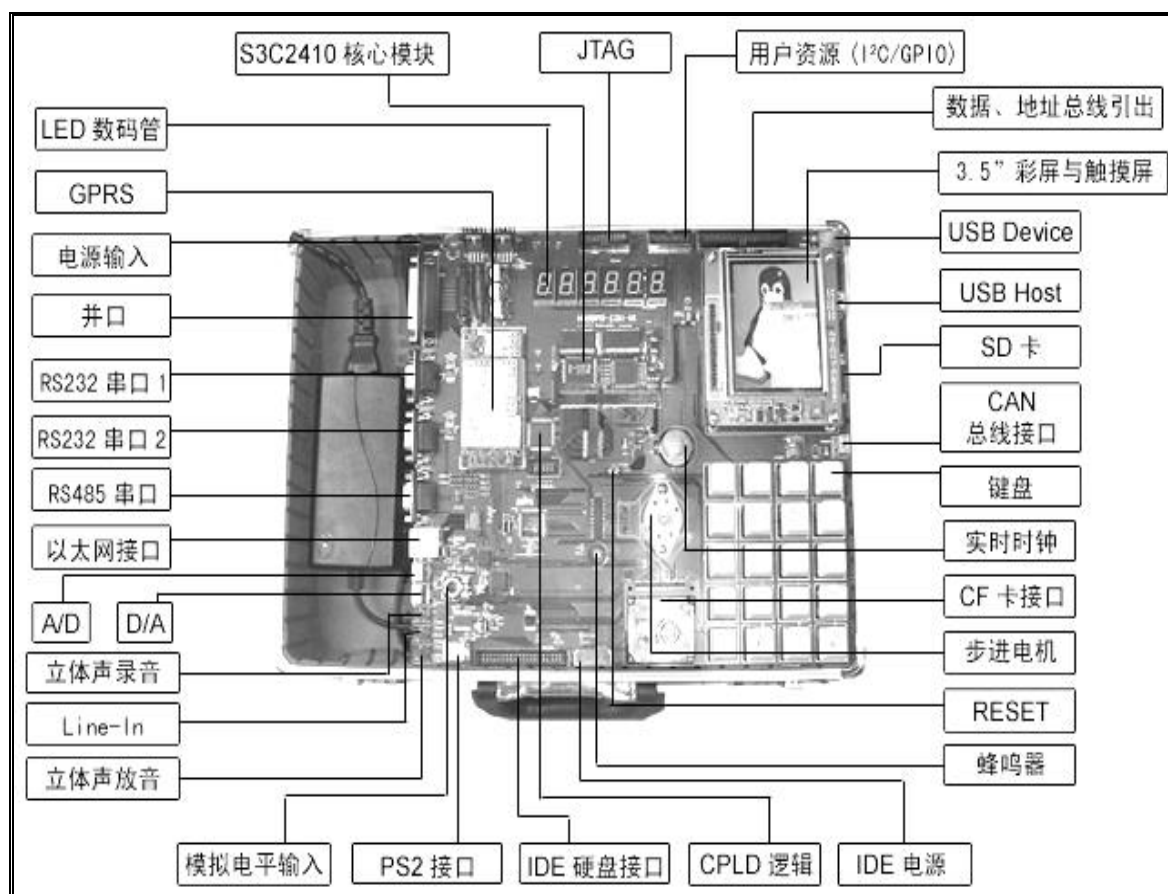


其中核心板上 JTAG 管脚分布如下图示:

2	4	6	8	10
1	3	5	7	9

其对应的管脚定义请参见下面 4.6 节。JTAG 头接 PC 并口一端的有标注的一边为 1 脚。

(2) 外设接口底板图:



HHARM9-EDU 基本底板的主要接口和器件排布参见上图。

2.5.8 接口管脚说明

(1) LED 指示灯

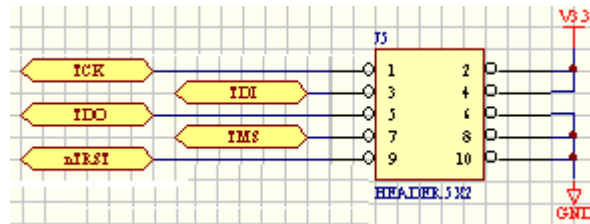
核心板：

状态指示部分：由 LED 提供 3.3V 电源/复位指示。

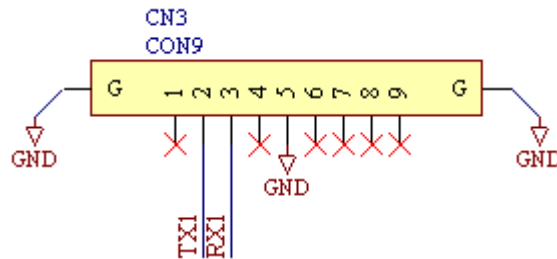
底板（基本板）：

状态指示部分：由 LED 提供电源指示。

(2) JTAG 接口



(3) RS232 COM1 接口：



管脚说明：

TX1 串行口 1 发送线

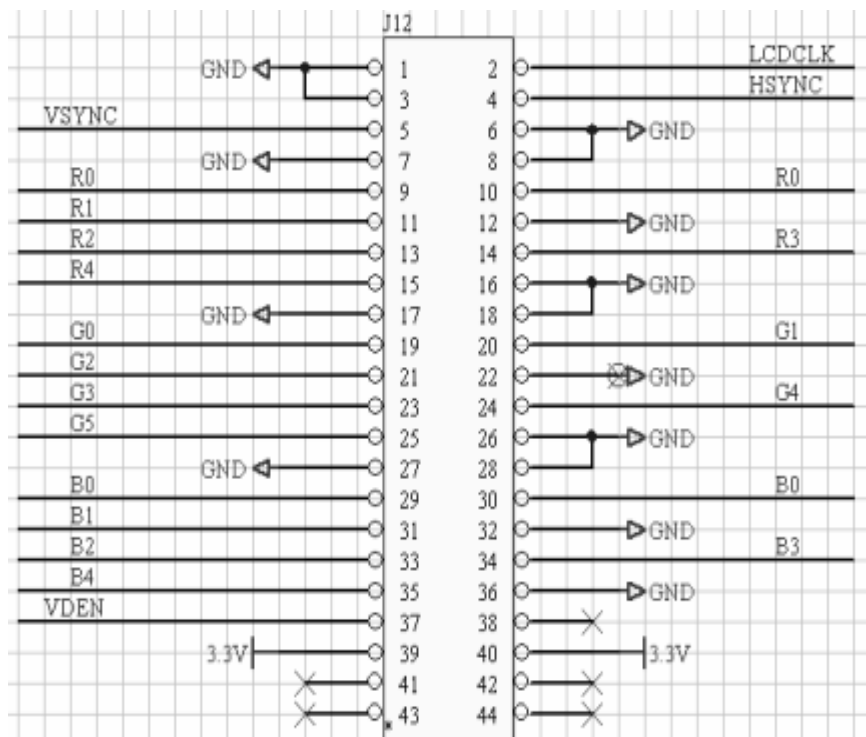
RX1 串行口 1 接受线

(4) DIMM144

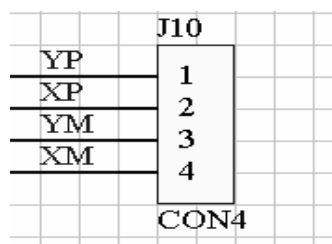
Pin#	Front Side	Pin#	Back Side	Pin#	Front Side	Pin#	Back Side
1	VSS	2	VSS	71	nXDREQ0	72	nXDACK0
3	MD0	4	MA0	73	nRST	74	nRST_IN
5	MD1	6	MA1	75	VSS	76	VSS
7	MD2	8	MA2	77	VD3	78	VD13
9	MD3	10	MA3	79	VD4	80	VD14
11	MD4	12	MA4	81	VD5	82	VD15
13	MD5	14	MA5	83	VD6	84	VD19
15	MD6	16	MA6	85	VD7	86	VD20
17	MD7	18	MA7	87	VD10	88	VD21
19	MD8	20	MA8	89	VD11	90	VD22
21	MD9	22	MA9	91	VD12	92	VD23
23	MD10	24	MA10	93	VFRAME	94	VLINE
25	MD11	26	MA11	95	VCLK	96	VM
27	MD12	28	MA12	97	LEND	98	LCD_PWR

29	MD13	30	MA13	99	VD0	100	VD9
31	MD14	32	MA14	101	VD1	102	VD16
33	MD15	34	MA15	103	VD2	104	VD17
35	nGCS1	36	MA16	105	VD8	106	VD18
37	nGCS2	38	MA17	107	LCDVF0	108	LCDVF1
39	nGCS3	40	MA18	109	LCDVF2	110	IIS_SDO
41	nGCS4	42	MA19	111	IIS_SDI	112	CDCLK
43	nRD	44	MA20	113	IIS_SCLK	114	IIS_LRCK
45	nOE	46	MA21	115	SDCLK	116	SDCMD
47	nWE	48	MA22	117	SDDATA0	118	SDDATA1
49	nWAIT	50	MA23	119	SDDATA2	120	SDDATA3
51	DQM0	52	DQM1	121	DP0	122	DN0
53	EINT0	54	EINT1	123	DP1	124	DN1
55	EINT2	56	EINT3	125	nSS0	126	SPI_CLK
57	EINT4	58	EINT5	127	SPI_MOS I	128	SPI_MISO
59	EINT6	60	EINT7	129	nYPON	130	YMON
				131	nXPON	132	XMON
				133	AIN0	134	AIN1
61	TXD0	62	nRTS0	135	AIN2	136	AIN3
63	RXD0	64	nCTS0	137	Vref	138	RTC_VDD
65	TXD1	66	nRTS1	139	VSS	140	VSS
67	RXD1	68	nCTS1	141	IIC_SDA	142	IIC_SCL
69	TOUT0	70	TOUT1	143	VDD	144	VDD

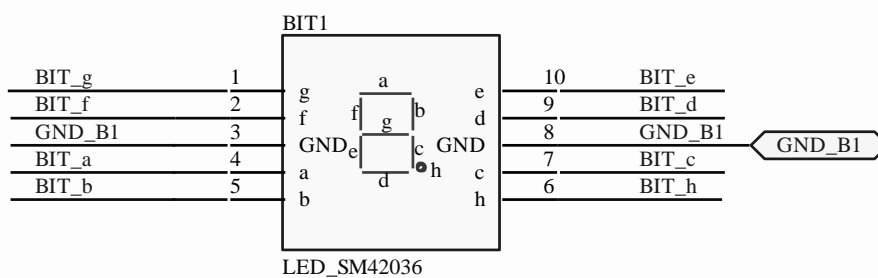
(5) LCD 接口



(6) 触摸屏接口

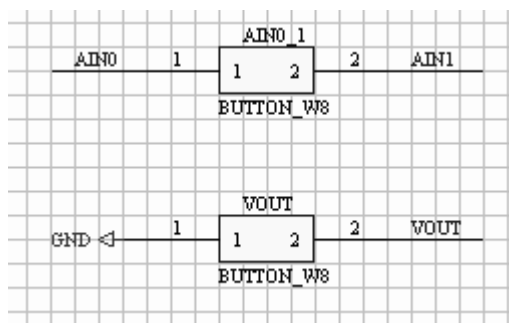


(7) 六个 7 段 LED 数码管



连接到逻辑芯片上，共阴极，可以通过逻辑控制 BIT_a,BIT_b...置成高电平后，点亮相应的那段数码管。

(8) A/D、D/A 模块:

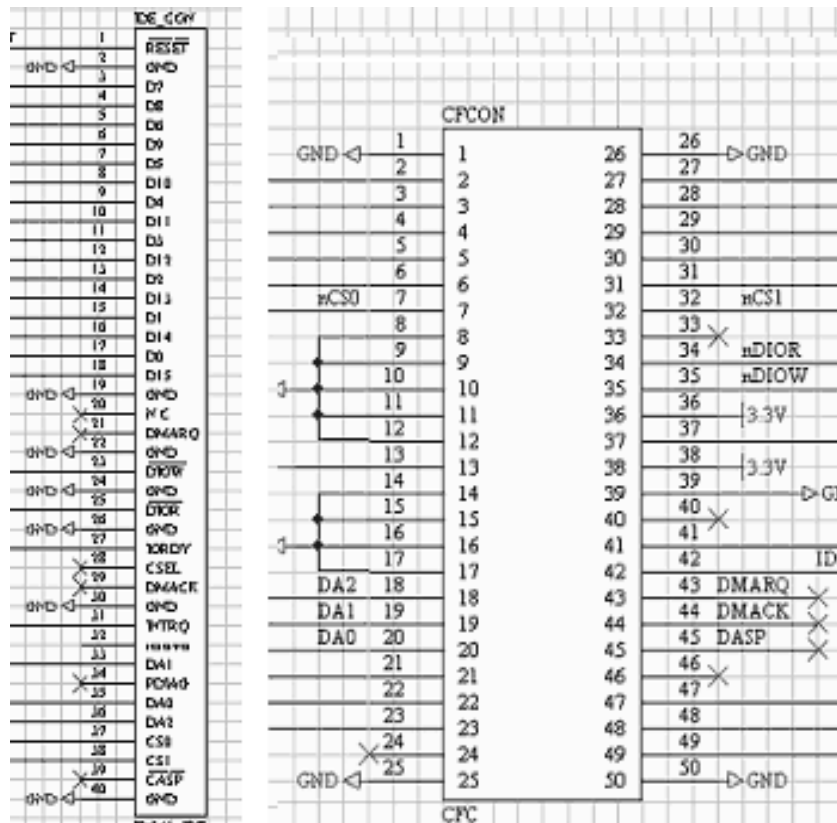


AIN0、AIN1：模拟输入信号；

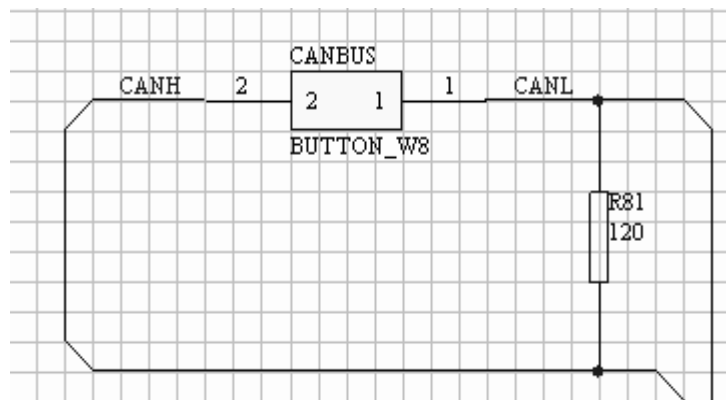
WOUT：数据输出信号。

(9) IDE/CF 卡接口

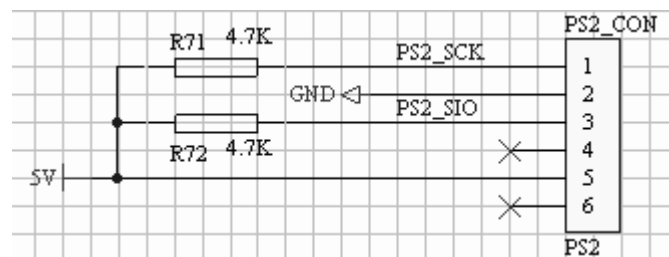
外扩 CPLD，支持硬盘/CF 卡，硬盘和 CF 卡两路信号复用，只能使用其中一个设备。



(10) CAN BUS（CAN 总线）接口



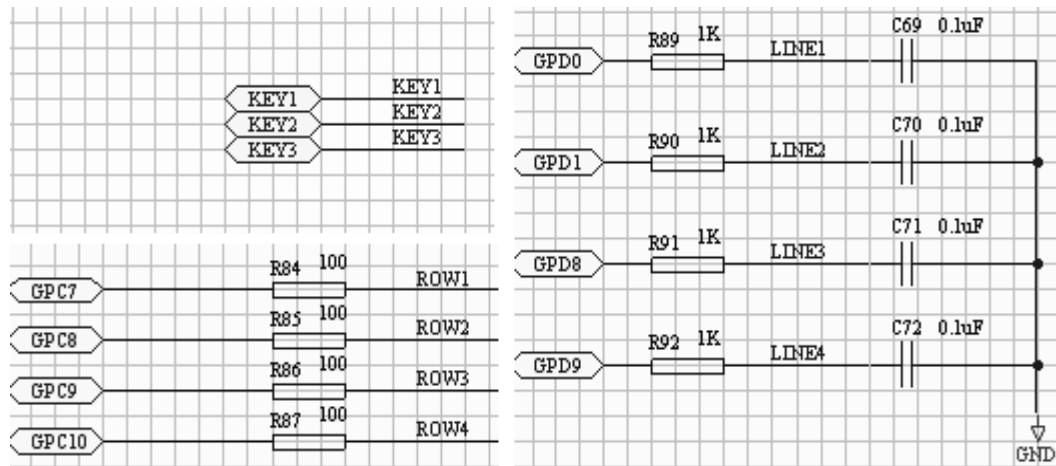
(11) PS/2 键盘接口



PS2_SCK: 键盘时钟。

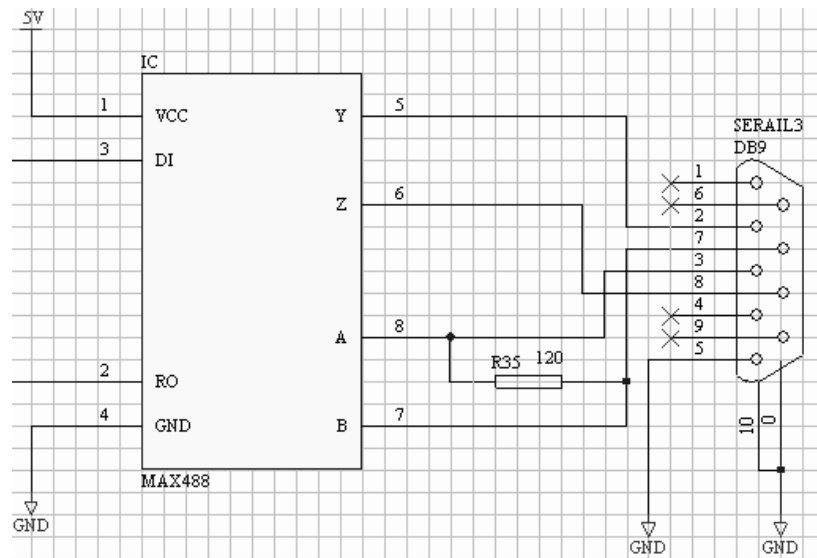
(12) 5*4 按键

其中有 4x4 GPIO 控制键盘，测试 CPLD 编程的三个按键和 EINT5 按键，构成 5x4 键盘。



(13) RS485 串口

与第二个 RS232 标准串口复用。

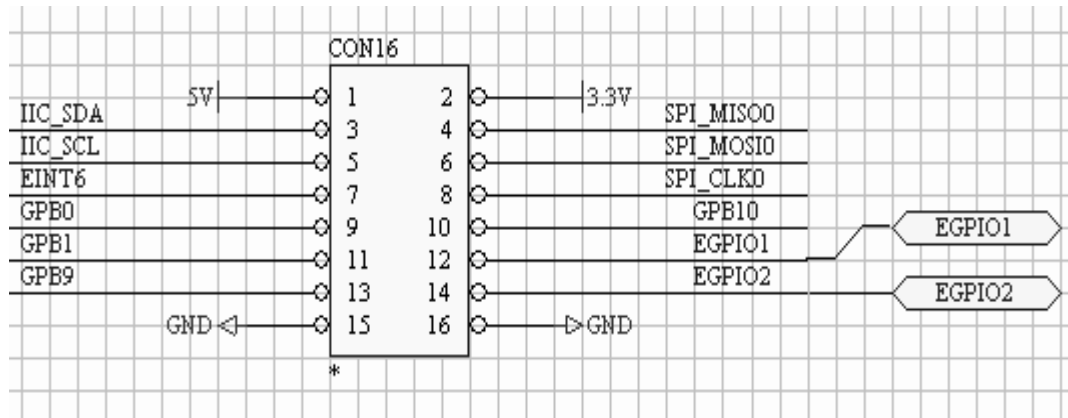


DI: 接 TXD

RO: 接 RXD

(14) 扩展接口

CON16（留给用户使用）



IIC_SDA: IIC 总线数据线

IIC_SCL: IIC 总线时钟线;

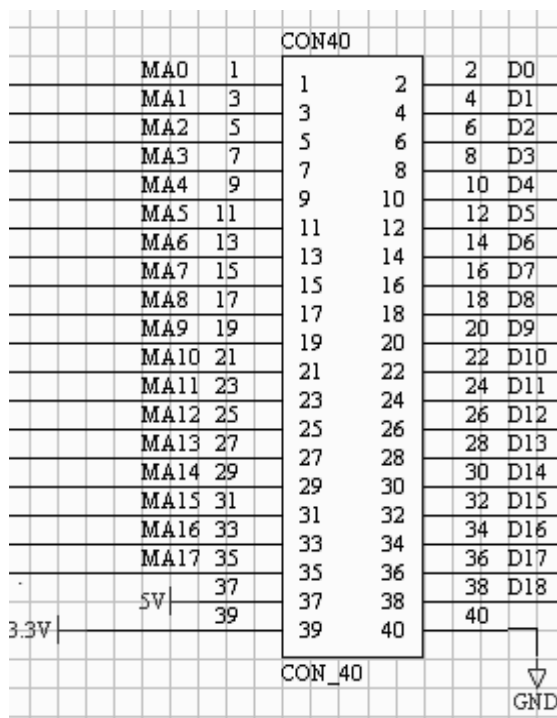
SPI 总线: SPIMISO0、SPIMOSI0、SPI_CLK0, 可以选择任一 GPIO 作为片选控制接到此 SPI 总线设备;

GPIO: GPB0、GPB1、GPB9、GPB10

EGPIO1、EGPIO2: 编程逻辑控制器 (CPLD) 的通用输入输出口。

(15) 扩展接口

CON40 (留给用户使用)



MA0, MA1, ……MA17: 18 根地址线;

D0, D1, ……D18: 19 根数据丝。

(16) 跳线说明:

- (a) 选择 RS232 串口或 RS485 串口: RXD1_j;TXD1_j,nRTS1_j,nCTS1_j 必须跳到离 PCB 板边缘近的一边。

选择 RS232:nRXD2_j;nTXD2_j 跳到离 PCB 板边缘近的一边,此两个跳线全部跳到远离 PCB 边缘则选择 RS485 串口。

- (b) 选择 GPRS 模块: 把 RXD1_j;TXD1_j,nRTS1_j,nCTS1_j 跳到远离 PCB 板边缘的一边。

第三章 实验指导

实验一：嵌入式应用程序开发入门

一. 实验目的

通过一个最简单，最基本的嵌入式应用程序，熟悉基本的嵌入式应用程序开发环境和工具，理解基本的嵌入式程序设计方法和流程。学会使用 **Make** 和 **Makefile**。

二. 实验原理和说明

1. 开发环境

绝大多数的 **Linux** 软件开发都是以 **native** 方式进行的，即本机（**HOST**）开发、调试，本机运行的方式。这种方式通常不适合于嵌入式系统的软件开发，因为对于嵌入式系统的开发，没有足够的资源在本机（即板子上系统）运行开发工具和调试工具。通常的嵌入式系统的软件开发采用一种交叉编译调试的方式。交叉编译调试环境建立在宿主机（即一台 **PC** 机）上，对应的开发板叫做目标板。

开发时使用宿主机上的交叉编译、汇编及连接工具形成可执行的二进制代码，（这种可执行代码并不能在宿主机上执行，而只能在目标板上执行。）然后把可执行文件下载到目标机上运行。调试时的方法很多，可以使用串口，以太网口等，具体使用哪种调试方法可以根据目标机处理器所提供的支持作出选择。宿主机和目标板的处理器一般都不相同，宿主机为 **Intel** 处理器，而目标板如 **HHARM9-EDU** 为 **SAMSUNG S3C2410**，**GNU** 编译器提供这样的功能，在编译编译器时可以选择开发所需的宿主机和目标机从而建立开发环境。所以在进行嵌入式开发前第一步的工作就是要安装一台装有指定操作系统的 **PC** 机作宿主开发机，对于嵌入式 **Linux**，宿主机上的操作系统一般要求为 **Redhat Linux**，在此，华恒推荐使用 **Redhat 9.0** 作为本套开发系统的宿主机 **PC** 操作系统。

在宿主机上我们要建立交叉编译调试的开发环境。环境的建立需要许多的软件模块协同工作，这将是一个比较繁杂的工作，但现在已完全由套件中光盘的安装而自动完成了。

这种宿主机（**HOST**）——目标板（**TARGET**）的开发模式下的系统连接图如图 2-1 所示：



2. 开发工具

宿主机上交叉编译开发环境由 **GNU** 开发工具集构成，它们在安装华恒软件光盘时被复制到宿主机的 **/opt/host/armv4l** 目录下。

GNU 工具集

armv4l-unknown-linux-gcc	armv4l-unknown-linux-objcopy	Armv4l-unknown-linux-gdb
armv4l-unknown-linux-as	armv4l-unknown-linux-objdump	Armv4l-unknown-linux-gas
armv4l-unknown-linux-ld	armv4l-unknown-linux-strip	Armv4l-unknown-linux-size
armv4l-unknown-linux-g++	armv4l-unknown-linux-nm	Armv4l-unknown-linux-add r2line
armv4l-unknown-linux-cc1	armv4l-unknown-linux-ar	
armv4l-unknown-linux-cpp	armv4l-unknown-linux-ranlib	
armv4l-unknown-linux-cc1plus	armv4l-unknown-linux-strings	

其中最主要的开发工具有：

- (1) 源代码编辑：vi (m)。vi (m) 是 LINUX 世界中最好用，使用最为广泛的编辑器，其具体用法介绍可以参见附录 B。
- (2) 编译：armv4l-unknown-linux-as(汇编代码)/armv4l-unknown-linux-gcc(C 代码)/armv4l-unknown-linux-g++(C++代码)。LINUX 内核及应用程序采用 GNU gcc 作为 C 编译器。gcc 已经成功的移植到不同的处理器平台上。
- (3) 链接：armv4l-unknown-linux-ld。
- (4) 辅助工具：armv4l-unknown-linux-objcopy / armv4l-unknown-linux -ar/
armv4l-unknown-linux -nm。

3. 应用程序开发流程

应用程序的开发有间接和直接在目标板上开发两种模式。

3.1 间接方式

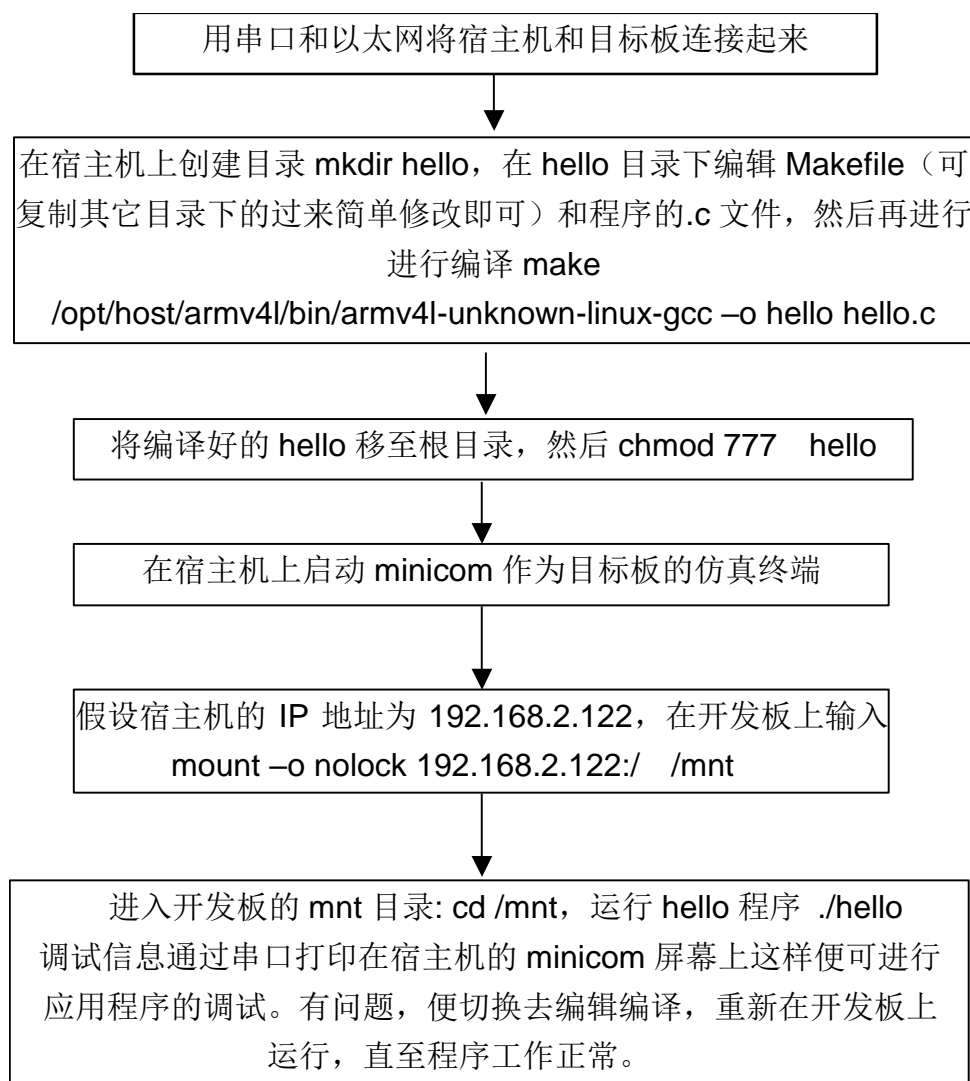
先在宿主机（Intel CPU）上调试通过后，再移植到目标板上。移植的工作包括两个方面：

- (1) 函数库的问题。在程序移植时可能会有函数未定义的问题。对于这种问题，一般要求开发者自己编制这些要用到却又未定义的函数。
- (2) 要修改 Makefile 以选择适合目标板的编译工具。

3.2 通用开发模式

直接在目标板上进行开发（通用开发模式，建议采用该模式）。将宿主机和目标板通过以太网连接，在宿主 PC 机上运行 minicom 作为目标板的显示终端，在目标板上通过 NFS (网络文件系统) 来 mount 宿主机硬盘，让应用程序直接运行在目标板上进行调试。下面给出这种直接 TARGET 开发模式下的开发流程图。

假设/HHARM9-EDU/application/hello/hello 是我们要调试的应用。



下面介绍一下 U 盘调试法：（需要有 USB host 支持）

- (1) 取消 U 盘的写保护，连接到宿主机上，执行

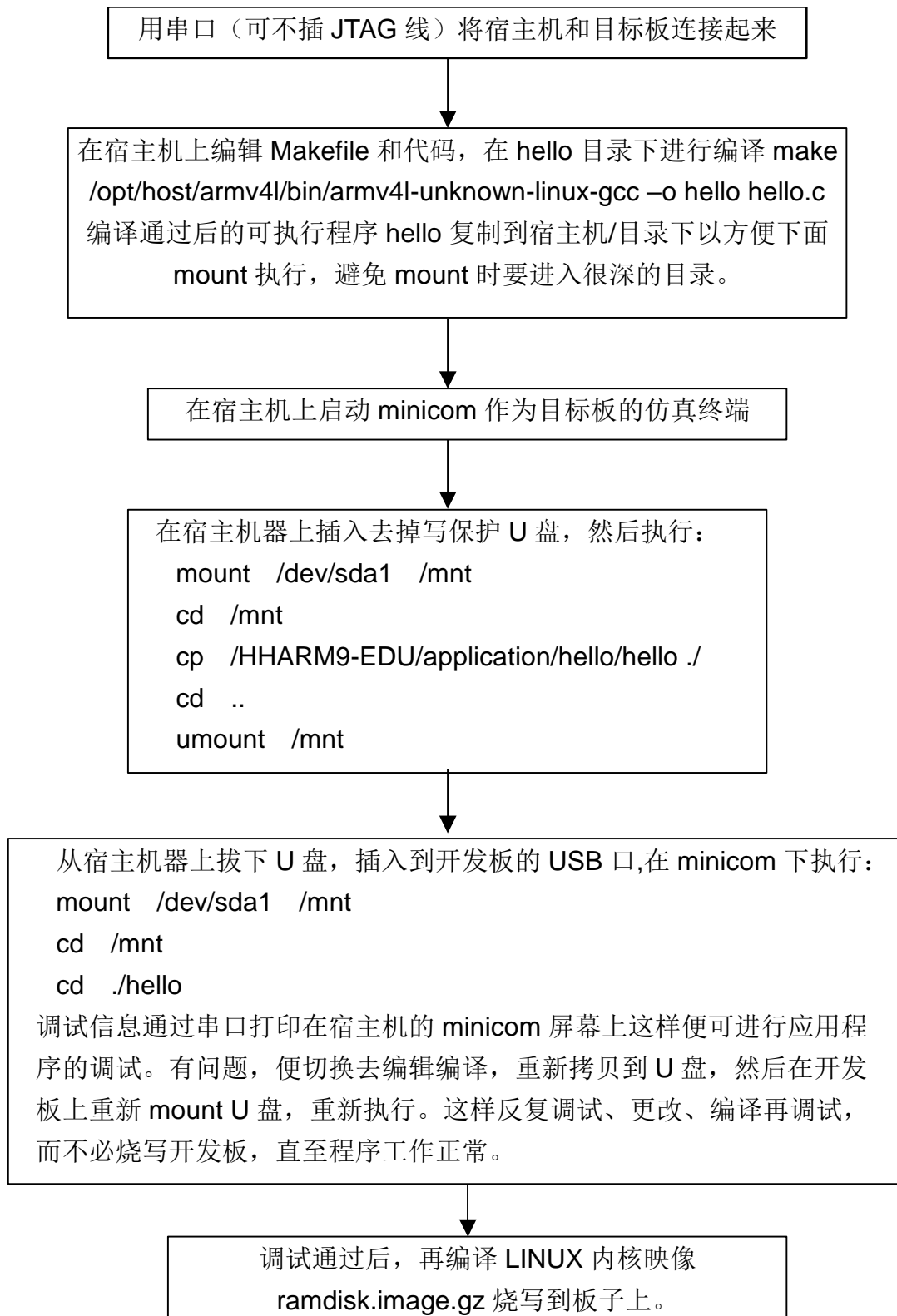
```
mount /dev/sda1 /mnt
cd /mnt
cp /HHARM9-EDU/application/hello/hello ./
cd ..
umount /mnt
```

- (2) 取下 U 盘，插入开发板的 USB 接口，执行

```
mount /dev/sda1 /mnt
cd /mnt
./hello
```

就可以看到程序在开发板上运行的情况。调试成功以后，把应用程序添加到 `ramdisk` 文件系统映像中，就可以下载，烧写新的 `ramdisk` 映像文件。

下面给出这种通过 U 盘直接 `TARGET` 开发模式下的开发流程：



4. Make 和 Makefile 文件

在大型的开发项目中，通常有几十到上百个的源文件，如果每次都手工键入 gcc 命令进行编译的话，则会非常不方便。因此，人们通常利用 make 工具来自动完成编译工作。这些工作包括：

如果仅修改了某几个源文件，则只重新编译这几个源文件；如果某个头文件被修改了，则重新编译所有包含该头文件的源文件。利用这种自动编译可以大大简化开发工作，避免不必要的重新编译。

实际上，**make** 工具通过一个称为 **makefile** 的文件来完成并自动维护编译工作。**makefile** 需要按照某种语法进行编写，其中说明了如何编译各个源文件并连接生成可执行文件，并定义了源文件之间的依赖关系。当修改了其中某个源文件时，如果其他源文件依赖于该文件，则也要重新编译所有依赖该文件的源文件。

makefile 中一般包含如下内容：

- (1) 需要由 **make** 工具创建的项目，通常是目标文件和可执行文件。通常使用“目标（**target**）”一词来表示要创建的项目。
- (2) 要创建的项目依赖于哪些文件。
- (3) 创建每个项目时需要运行的命令。

例如，假设你现在有一个 **C++** 源文件 **test.C**，该源文件包含有自定义的头文件 **test.h**，则目标文件 **test.o** 明确依赖于两个源文件：**test.c** 和 **test.h**。另外，你可能只希望利用 **g++** 命令来生成 **test.o** 目标文件。这时，就可以利用如下的 **makefile** 来定义 **test.o** 的创建规则：

```
# This makefile just is a example.
# The following lines indicate how test.o depends
# test.C and test.h, and how to create test.o
test.o: test.c test.h
    g++ -c -g test.C
```

从上面的例子注意到，第一个字符为 **#** 的行为注释行。第一个非注释行指定 **test.o** 为目标，并且依赖于 **test.c** 和 **test.h** 文件。随后的行指定了如何从目标所依赖的文件建立目标。当 **test.c** 或 **test.h** 文件在编译之后又被修改，则 **make** 工具可自动重新编译 **test.o**，如果在前后两次编译之间，**test.C** 和 **test.h** 均没有被修改，而且 **test.o** 还存在的话，就没有必要重新编译。这种依赖关系在多源文件的程序编译中尤其重要。通过这种依赖关系的定义，**make** 工具可避免许多不必要的编译工作。当然，利用 **Shell** 脚本也可以达到自动编译的效果，但是，**Shell** 脚本将全部编译任何源文件，包括哪些不必要重新编译的源文件，而 **make** 工具则可根据目标上一次编译的时间和目标所依赖的源文件的更新时间而自动判断应当编译哪个源文件。

一个 **makefile** 文件中可定义多个目标，利用 **make target** 命令可指定要编译的目标，如果不指定目标，则使用第一个目标。通常，**makefile** 中定义有 **clean** 目标，可用来清除编译过程中的中间文件，例如：

clean:

```
rm -f *.o
```

运行 **make clean** 时，将执行 **rm -f *.o** 命令，最终删除所有编译过程中产生的所有中间文件。

除了一般的目标之外，**make** 也允许指定伪目标。称它们为伪目标是因为它们并不对应实际的文件。比如上面的 **clean** 目标就是伪目标。但是由于 **clean** 不需要任何相关文件，它指定的命令不会自动执行。这是由 **make** 的工作机制决定的：在执行目标时 **make** 先检查相关文件是否存在。因为 **clean** 没有相关文件，**make** 就认为该目标已经是最新的了。此时，必须键入 **make clean** 来执行这个目标文件。然而如果正好有一个名为 **clean** 的文件存在。**Make** 就会发现它，并认为它是最新的。因此不会执行任何动作。这种情况下就需要使用特殊的目标 **.PHONY**。**.PHONY** 目标的相关文件的含义和通常一样，但是 **make** 将不检查是否存在这些文件而直接执行与之相关的命令。格式如下：

```
.PHONY: clean
```

GNU 的 `make` 工具除提供有建立目标的基本功能之外，还有许多便于表达依赖性关系以及建立目标的命令的特色。其中之一就是变量或宏的定义能力。如果你要以相同的编译选项同时编译十几个 C 源文件，而为每个目标的编译指定冗长的编译选项的话，将是非常乏味的。但利用简单的变量定义，可避免这种乏味的工作：

```
# Define macros for name of compiler
CC = gcc
# Define a macro for the CC flags
CCFLAGS = -D_DEBUG -g -m486
# A rule for building a object file
test.o: test.c test.h
    $(CC) -c $(CCFLAGS) test.c
```

在上面的例子中，`CC` 和 `CCFLAGS` 就是 `make` 的变量。GNU `make` 通常称之为变量，而其他 UNIX 的 `make` 工具称之为宏，实际是同一个东西。在 `makefile` 中引用变量的值时，只需变量名之前添加 `$` 符号，如上面的 `$(CC)` 和 `$(CCFLAGS)`。

GNU `make` 还有许多预定义的变量。下面给出一些主要的预定义变量，除这些变量外，GNU `make` 还将所有的环境变量作为自己的预定义变量。

<code>\$*</code>	不包含扩展名的目标文件名称。
<code>\$+</code>	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件。
<code>\$<</code>	第一个依赖文件的名称。
<code>\$?</code>	所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚。
<code>\$@</code>	目标的完整名称。
<code>^</code>	所有的依赖文件，以空格分开，不包含重复的依赖文件。
<code>%</code>	如果目标是归档成员，则该变量表示目标的归档成员名称。例如，如果目标名称为 <code>mytarget.so(image.o)</code> ，则 <code>\$@</code> 为 <code>mytarget.so</code> ，而 <code>%</code> 为 <code>image.o</code> 。

<code>AR</code>	归档维护程序的名称，默认值为 <code>ar</code> 。
<code>ARFLAGS</code>	归档维护程序的选项。
<code>AS</code>	汇编程序的名称，默认值为 <code>as</code> 。
<code>ASFLAGS</code>	汇编程序的选项。
<code>CC</code>	C 编译器的名称，默认值为 <code>cc</code> 。
<code>CCFLAGS</code>	C 编译器的选项。
<code>CPP</code>	C 预编译器的名称，默认值为 <code>\$(CC) -E</code> 。
<code>CPPFLAGS</code>	C 预编译的选项。
<code>CXX</code>	C++ 编译器的名称，默认值为 <code>g++</code> 。
<code>CXXFLAGS</code>	C++ 编译器的选项。
<code>FC</code>	FORTTRAN 编译器的名称，默认值为 <code>f77</code> 。
<code>FFLAGS</code>	FORTTRAN 编译器的选项。

GNU `make` 包含有一些内置的或隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标。两种类型的隐含规则：

(1) 后缀规则（Suffix Rule）。后缀规则是定义隐含规则的老风格方法。后缀规则定义了将一个具有

某个后缀的文件（例如，.c 文件）转换为具有另外一种后缀的文件（例如，.o 文件）的方法。每个后缀规则以两个成对出现的后缀名定义，例如，将 .c 文件转换为 .o 文件的后缀规则可定义为：

.c.o:

`$(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<`

- (2) 模式规则（pattern rules）。这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖性规则。模式规则看起来非常类似于正则规则，但在目标名称的前面多了一个 % 号，同时可用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个 X.c 文件转换为 X.o 文件：

`%c.o`

`$(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<`

同大多数 GNU 程序一样，make 也有丰富的命令行选项。下面列出最常见的部分：

<code>-C DIR</code>	在读取 makefile 之前改变到指定的目录 DIR。
<code>-f FILE</code>	以指定的 FILE 文件作为 makefile。
<code>-h</code>	显示所有的 make 选项。
<code>-i</code>	忽略所有的命令执行错误。
<code>-I DIR</code>	当包含其他 makefile 文件时，可利用该选项指定搜索目录。
<code>-n</code>	只打印要执行的命令，但不执行这些命令。
<code>-p</code>	显示 make 变量数据库和隐含规则。
<code>-s</code>	在执行命令时不显示命令。
<code>-r</code>	禁止使用所有 make 的内置规则
<code>-d</code>	打印调试信息
<code>-w</code>	在处理 makefile 之前和之后，显示工作目录。
<code>-W FILE</code>	假定文件 FILE 已经被修改。

如果在使用 make 时遇到问题，可以使用 -d 选项打印出大量的额外调试信息。此时因为需要显示 make 内部所做的每一件事以及为什么做这些事的调试信息，将会产生大量输出。其中包括如下信息：

- (1) 在重新编译时，make 需要检查的文件。
- (2) 被比较的文件以及比较的结果。
- (3) 需要被重新生成的文件。
- (4) make 想要使用的隐含规则。
- (5) make 实际使用的隐含规则以及所执行的命令。

下面列出使用 make 时可能遇到的最常见的出错信息。完整的文档请参见 make 使用手册。

- (1) No rule to make target 'target'. Stop: makefile 中没有包含创建指定的 target 所需要的规则，而且也没有合适的缺省规则可用。
- (2) 'target' is up to date: 指定 target 的相关文件没有变化，target 已是最新的。
- (3) command:command not found: make 找不到 command。一般是因为命令拼写错误或不在路径 \$PATH.s 下。
- (4) Target 'target' not remade because of errors: 在编译 target 时出错，这一消息仅在使用 make 的 -k 选项时才出现。
- (5) Illegal option-option: 在调用 make 时包含了不能被 make 识别的选项。

三. 实验内容和步骤

熟悉开发环境和工具，掌握并编写 Makefile。在操作系统的基础上，实现向串口输出“Hello World”字符串。

1. 在根目录下创建名为 **hello** 的目录。

2. 编写对应的 **Makefile** 文件。

注意：编译器要用/opt/host/armv4l/bin/armv4l-unknown-Linux-gcc。Makefile 中每行的缩进不允许用多个空格，而必须用 TAB 键，否则编译就会有莫名其妙的错误。应用程序默认的栈的大小为 8K。可以在 Makefile 为其指定堆栈大小，例如要增大该应用程序的堆栈容量，就在其 Makefile 中增加如下一句：FLTLFLAGS = -s 65536

3. 编写 **hello.c**，通过 **printf** 函数向终端上输出字符串。

4. 执行 **make**，生成可执行文件 **hello**。

注意：一定要学会看 **make** 打印的错误。一般的，在有許多错误的情况下，一定要查看第一个 **error**，**warning** 是不需要理会的。

5. 运行并检查结果。

用串口线将 PC 与板子的第一个串口连接起来。在宿主机上启动 **minicom** 作为仿真终端，在目标板上 **mount** 上宿主机根目录，切换到 **hello** 所在目录执行 **hello**。如果有问题，便切换回宿主机上编辑编译，重新 **mount** 和执行，直到正确为止。正确结果是 **minicom** 上将显示“Hello World!”字符串。

为了防止进入很深的目录层次，可以将可执行文件 **hello** 复制到根目录下。那么 **mount** 后进入一层目录就可以执行了。

6. 将生成的可执行文件加入 Linux 文件系统中去，重新制作 **ramdisk** 文件系统映像并烧写 **FLASH**。

/HHARM9-EDU/Images/ramdisk.image.gz 为 LINUX 的文件系统映像压缩文件。用户可以在文件系统中加入自己的应用，例如可以将 **ramdisk.image.gz** 拷贝到根目录下，新建一个 **ramdisk** 目录并解开 **ramdisk.image.gz**：

```
cp ramdisk.image.gz /
cd /
mkdir ramdisk
gunzip /ramdisk.image.gz
```

此时根目录下会生成 **ramdisk.image**。**ramdisk.image** 为解开后的 Linux 的文件系统映像文件；再将 **ramdisk.image** 文件系统映像文件 **mount** 到新建目录 **ramdisk** 中：

```
mount -o loop ramdisk.image ramdisk/
```

此时用户可以加入自己的应用程序 **hello**：

```
cd /ramdisk
mkdir application      (新建目录名可以自己定义)
cd application
mkdir hello
cd hello
cp /hello .
cd /
umount /ramdisk
```

现在压缩新生成的 **ramdisk.image** 文件系统映像文件：

```
gzip /ramdisk.image /ramdisk.image.gz
cp /ramdisk.image.gz /HHARM9-EDU/Images/
cp /ramdisk.image.gz /tftpboot/
```

此时就可以下载和烧写 `ramdisk.image.gz` 到 Flash 中。启动开发板，询问时输入 `n`。就进入了命令行模式。输入命令：

```
SMDK2410# tftp 30800000 ramdisk.image.gz
```

```
SMDK2410# fl 1140000 30800000 300000
```

参数的意义为：`0x1140000` 为内核烧写到 Flash 的地址，当 `ppcbboot` 启动后它会从此地址加载 Linux 内核。`0x30008000` 为使用的内核下载到内存中的地址。`0x200000` 为内核的大小，`0x200000` 可被替换为大于内核大小且是 `0x20000` 倍数的最小整数。您可以根据自己编译的内核大小来确定此值。

烧些完毕后重启目标板，可以看到文件系统中出现了 `application` 目录，在 `hello` 目录中出现了可执行文件 `hello`，就可以运行文件了。

```
cd application
```

```
cd hello
```

```
./hello
```

实验二：WEB SERVER/CGI 实验

一. 实验目的

学习 Web 服务器(boa)的配置方法, 掌握 CGI 控制板子的方法。学习通过 CGI 添加自己的服务, 定制自己的管理软件的方法。

二. 实验原理和说明

1. 软件组成

HHARM9-EDU 的 WEB 管理软件由 WEB SERVER (boa) +CGI 代码两部分构成。其中 boa 为一个标准的 WEB SERVER, 用户可扩展部分就是 CGI 代码部分, 通过修改 CGI 代码, 用户可定制自己的管理软件。

Boa 服务器的所有代码在 /HHARM9-EDU/application/boa/。源代码在 /HHARM9-EDU/application/boa/src/下。

CGI 开发的所有代码均在/home/httpd/cgi-bin/目录下。其中主要的为三个文件:

- (1) config.c: 为入口代码 (main 函数所在) 及命令 (flag) 分派代码。因为 CGI 的命令调用格式如下: 例如更改 IP 的 URL 为:

http://192.168.2.111/cgi-bin/config.cgi?flag=49

给它分配的命令码为 49, 在 config.c 代码中就是根据这些命令码来调用相应的处理响应函数, 它全部是由判断语句构成的, 是个总领性的文件。

- (2) cgihead.c: 这是 CGI 的主体代码, 它完成了所有的处理响应函数代码。

- (3) myhdr.c: 提供了一些辅助函数, 例如主体代码中常用的字符串处理等公用函数代码。

注意: 这里的 CGI 代码全部是用 C 代码写成, 不支持 perl, PHP 等脚本。

2. WEB 服务器编译流程

Boa 是一个单任务的 http 服务器, 源代码开放、性能高。实现 Boa 的过程比较简单。

- (1) 下载 boa 源代码包, 并解压。
- (2) 编译 boa 源代码, 生成可执行文件 boa。这需要在/boa/src/configure 文件中加入 CC 和 CPP 的说明, 同时修改 Makefile 里的
CC = /opt/host/armv4l/bin/armv4l-unknown-linux-gcc,
然后 make 就可以了。
- (3) 建目录。由于默认的根本文件系统 ramdisk 是只读的, 不能用 mkdir 等命令来新建目录, 故应在编译内核前先建好要用到的目录。
- (4) 需要对 Boa 做一些配置和修改。这主要通过对 boa.conf 和 mime.types 文件进行修改来实现, 需要改动的配置有以下几项。

- 1) 指定 Web 服务器的根目录路径 (SERVER_ROOT)

进入/boa/src/目录, 通过修改 defines.h 文件中修改

#define SERVER_ROOT "/home/httpd"

语句来指定 SERVER_ROOT。

另外, 还可以通过命令行来指定, 例如: `boa -c /home/httpd&`。而且命令行指定的 SERVER_ROOT 可以覆盖 defines.h 文件所指定的。

- 2) 修改 boa.conf 文件和 MIME.types 文件

修改方法参见后面 boa.conf 文件和 MIME.types 文件的说明。一般来说, 只要修改

boa.conf 文件，MIME.types 文件不需要修改。

- (5) 在 ramdisk 中的/bin/下加入 boa 可执行文件。并把修改后的 bao.conf 和 mime.types 拷贝到 Web 服务器根目录/home/httpd/下。并将一些静态页面放在由 boa.conf 指定的目录下。
- (6) 配置过程后，重新编译内核。把编译好的内核下载到开发板，启动 Boa Web Server，然后就可以通过 IE 访问你的网页了。如果想板子启动时自动启动 Boa Web Server，可以修改 rc 文件，在运行脚本 rc 中增加行：

boa 或者 boa -c /home/httpd/&

3. boa.conf 文件

该文件由一些规则组成，用于配置 boa 服务器，指定相应端口，服务器名称，一些相关文件的路径等等。Boa 服务器要想正确运行，必须保证该文件是正确配置的，而且该文件和某些静态网页，CGI 可执行程序等都放于某特定目录下，如服务器根目录下。

Boa 配置文件由 lex/yacc 或者由 flex/bison 产生的分割器识别分割各个配置项。如果分割时遇到错误，那么将给出错误所在的行号。每隔规则的语法很简单，而且出现顺序可以任意。

因为 boa.conf 文件如此重要，所以下面给出它的内容，并且详细说明每个规则。

Port: The port Boa runs on. The default port for http servers is 80.

If it is less than 1024, the server must be started as root.

Port 80

Listen: the Internet address to bind(2) to. If you leave it out,

it takes the behavior before 0.93.17.2, which is to bind to all

addresses (INADDR_ANY). You only get one "Listen" directive,

if you want service on multiple IP addresses, you have three choices:

1. Run boa without a "Listen" directive

a. All addresses are treated the same; makes sense if the

addresses are localhost, ppp, and eth0.

b. Use the VirtualHost directive below to point requests to different

files. Should be good for a very large number of addresses

(web hosting clients).

2. Run one copy of boa per IP address, each has its own configuration

with a "Listen" directive. No big deal up to a few tens of

addresses.Nice separation between clients.

The name you provide gets run through inet_aton(3), so you have to use

dotted quad notation. This configuration is too important to trust some

DNS.

#Listen 192.68.0.5

User: The name or UID the server should run as.

Group: The group name or GID the server should run as.

User 0

Group 0

ServerAdmin: The email address where server problems should be sent.


```
# Note: this is not currently used, except as an environment variable
# for CGIs.
#ServerAdmin root@localhost

# ErrorLog: The location of the error log file. If this does not start
# with /, it is considered relative to the server root.
# Set to /dev/null if you don't want errors logged.
# If unset, defaults to /dev/stderr
ErrorLog /home/httpd/log/boa/error_log

# Please NOTE: Sending the logs to a pipe ('|'), as shown below,
# is somewhat experimental and might fail under heavy load.
# "Usual libc implementations of printf will stall the whole
# process if the receiving end of a pipe stops reading."
#ErrorLog "|/usr/sbin/cronolog --symlink=/var/log/boa/error_log
#/var/log/boa/error-%Y%m%d.log"

# AccessLog: The location of the access log file. If this does not
# start with /, it is considered relative to the server root.
# Comment out or set to /dev/null (less effective) to disable
# Access logging.

AccessLog /home/httpd/log/boa/access_log
# Please NOTE: Sending the logs to a pipe ('|'), as shown below,
# is somewhat experimental and might fail under heavy load.
# "Usual libc implementations of printf will stall the whole
# process if the receiving end of a pipe stops reading."
#AccessLog "|/usr/sbin/cronolog --symlink=/var/log/boa/access_log
#/var/log/boa/access-%Y%m%d.log"

# UseLocaltime: Logical switch. Uncomment to use localtime
# instead of UTC time
#UseLocaltime

# VerboseCGILogs: this is just a logical switch.
# It simply notes the start and stop times of cgis in the error log
# Comment out to disable.
#VerboseCGILogs

# ServerName: the name of this server that should be sent back to
# clients if different than that returned by gethostname + gethostbyname
#ServerName www.your.org.here

# VirtualHost: a logical switch.
```

```
# Comment out to disable.
# Given DocumentRoot /var/www, requests on interface 'A' or IP 'IP-A'
# become /var/www/IP-A.
# Example: http://localhost/ becomes /var/www/127.0.0.1
#
# Not used until version 0.93.17.2. This "feature" also breaks commonlog
# output rules, it prepends the interface number to each access_log line.
# You are expected to fix that problem with a postprocessing script.
#VirtualHost

# DocumentRoot: The root directory of the HTML documents.
# Comment out to disable server non user files.
DocumentRoot /home/httpd/html

# UserDir: The name of the directory which is appended onto a user's home
# directory if a ~user request is recieved.
UserDir public_html

# DirectoryIndex: Name of the file to use as a pre-written HTML
# directory index. Please MAKE AND USE THESE FILES. On the
# fly creation of directory indexes can be _slow_.
# Comment out to always use DirectoryMaker
DirectoryIndex index.html

# DirectoryMaker: Name of program used to create a directory listing.
# Comment out to disable directory listings. If both this and
# DirectoryIndex are commented out, accessing a directory will give
# an error (though accessing files in the directory are still ok).
DirectoryMaker /usr/lib/boa/boa_indexer

# DirectoryCache: If DirectoryIndex doesn't exist, and DirectoryMaker
# has been commented out, the the on-the-fly indexing of Boa can be used
# to generate indexes of directories. Be warned that the output is
# extremely minimal and can cause delays when slow disks are used.
# Note: The DirectoryCache must be writable by the same user/group that
# Boa runs as.
# DirectoryCache /var/spool/boa/dircache

# KeepAliveMax: Number of KeepAlive requests to allow per connection
# Comment out, or set to 0 to disable keepalive processing
KeepAliveMax 1000

# KeepAliveTimeout: seconds to wait before keepalive connection times out
KeepAliveTimeout 10
```

```
# MimeTypes: This is the file that is used to generate mime type pairs
# and Content-Type fields for boia.
# Set to /dev/null if you do not want to load a mime types file.
# Do *not* comment out (better use AddType!)
MimeTypes /home/httpd/mime.types

# DefaultType: MIME type used if the file extension is unknown, or there
# is no file extension.
DefaultType text/plain

# AddType: adds types without editing mime.types
# Example: AddType type extension [extension ...]
# Uncomment the next line if you want .cgi files to execute from anywhere
#AddType application/x-httpd-cgi cgi

# Redirect, Alias, and ScriptAlias all have the same semantics -- they
# match the beginning of a request and take appropriate action. Use
# Redirect for other servers, Alias for the same server, and ScriptAlias
# to enable directories for script execution.
# Redirect allows you to tell clients about documents which used to exist in
# your server's namespace, but do not anymore. This allows you to tell the
# clients where to look for the relocated document.
# Example: Redirect /bar http://elsewhere/feh/bar

# Aliases: Aliases one path to another.
# Example: Alias /path1/bar /path2/foo
#Alias /doc /usr/doc

# ScriptAlias: Maps a virtual path to a directory for serving scripts
# Example: ScriptAlias /htbin/ /www/htbin/
ScriptAlias /cgi-bin/ /home/httpd/cgi-bin/
ScriptAlias /index.html/ /home/httpd/html/index.html
```

请特别注意上面加重的规则行。如果这些行指定的端口或者文件路径错误的话，服务器就无法正确运行。必须保证指定路径下确实存在那些文件或目录。你也可以把相关文件放于你想要的任何目录下，只要在该文件中修改相应的规则，指出在哪可以找到这些文件即可。

要注意的是，“ServerRoot”并不是在这个文件中指定，而是在 `boa/src/defines.h` 中指定。或者通过带参数 `-c` 的命令行

```
boa -c /home/httpd
```

指定。

4. MIME 类型

是一个标准的，大小写不敏感的串，用于标识数据类型。它普遍用于 Internet，用于多种目的。开始是通用数据类型（如 `text,image,audio`），然后是斜杠，以特定数据类型（如 `html,gif,jpeg`）结束。HTML 文件标识为 `text/html`，GIFs 和 JPEGs 标识为 `image/gif` 和 `image/jpeg`。

MIME 类型由配置文件 `mime.types` 给出。`Mime.types` 文件控制发送给客户的给定文件扩展名的网络媒介的类型。发送正确的媒介类型给客户是很重要的。他们可以凭借这个类型来决定如何处理接收到的文件。可以在该文件中添加新的附加类型，或者可以在 `config` 文件中使用 `AddTypes` 指示。

下面给出默认的配置文件的 `mime.types` 的一部分内容，以让大家了解其格式：

```
# MIME type      Extension

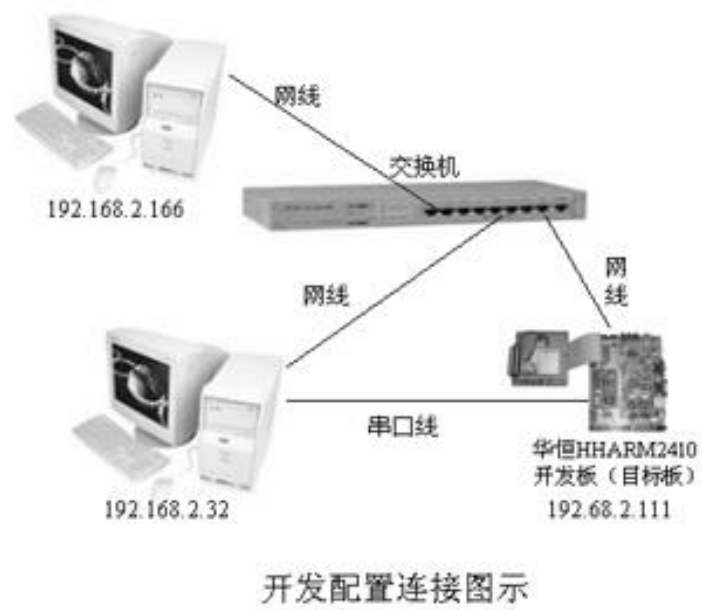
application/EDI-Consent
application/EDI-X12
.....
application/xml
application/zip          zip
audio/32kadpcm
audio/basic              au snd
audio/midi               mid midi kar
audio/mpeg               mpga mp2 mp3
.....
audio/x-pn-realaudio     ram rm
audio/x-realaudio        ra
audio/x-wav              wav
chemical/x-pdb            pdb xyz
image/cgm
image/g3fax
image/gif                gif
image/ief                ief
image/jpeg               jpeg jpg jpe
.....
image/x-xpixmap           xpm
image/x-xwindowdump       xwd
message/delivery-status
message/disposition-notification
message/external-body
message/http
message/news
message/partial
message/rfc822
model/iges                igs iges
model/mesh                msh mesh silo
model/vnd.dwf
model/vrml                wrl vrml
multipart/alternative
```

```
multipart/appldouble
.....
multipart/related
multipart/report
multipart/signed
multipart/voice-message
text/css          css
text/directory
text/enriched
text/plain        asc txt
.....
text/xml          xml
video/mpeg        mpeg mpg mpe
video/quicktime   qt mov
video/vnd.motorola.video
video/vnd.motorola.videop
video/vnd.vivo
video/x-msvideo   avi
video/x-sgi-movie movie
x-conference/x-cooltalk ice
text/html         html htm
```

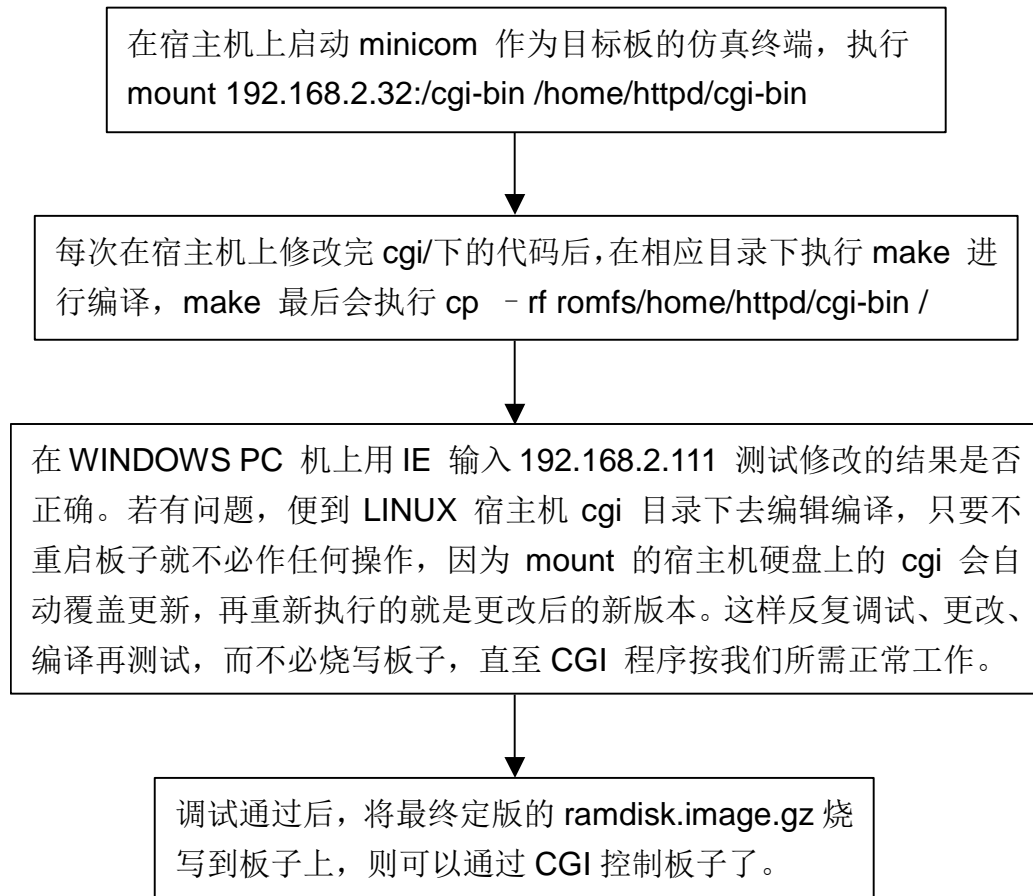
5. 关于 WEB/CGI 管理软件的扩展开发:

开发 WEB/CGI 管理软件除了需要一台安装 REDHAT LINUX 的 PC 机外, 还额外的需要一台 WINDOWS 的 PC 机, 用来通过 IE 测试板子上的 WEB 管理软件。

开发时系统连接图及其 IP 分配示例如下:

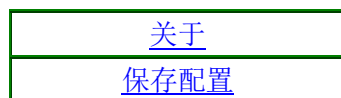


下面给出 WEB/CGI 管理软件开发的流程图：



系统管理界面是一个左右分割的界面，左边是管理功能菜单，右边是配置页面及配置结果显示页面。左边的管理菜单如下图示：

基本功能
设置 IP
系统时间设置
GPRS 拨号
MAC 地址设置
MINIGUI 演示
触摸屏演示
步进电机演示
录音
播放 MTV
高级设置
后台服务
系统工具
修改管理员密码
系统重启
恢复出厂配置



HHARM9-EDU 的 WEB 管理软件只是给出一个 WEB 管理的模板，其中提供了几个简单的功能演示。下面说明一下基本功能的实现：

- (1) 设置板子的 IP 地址：当写入新的 IP 地址后，将新的 IP 地址保存到/jffs2/etc/ipset 脚本，在下次启动时执行该脚本，完成 IP 设置功能。
- (2) 系统时间设置：通过调用 date 命令，来设置时间，修改 busybox 中的 date.c 使其直接从 RTC 中读取时间，并在设置时间时直接更改 RTC 时间。
- (3) GPRS 拨号：通过调用 pppd 拨号程序来实现此功能。
- (4) MAC 地址设置：以太网驱动中 MAC 地址是从 EEPROM 中获得，在此通过程序 setmac，向 EEPROM 中固定位置处，写入新的 MAC 地址，在下次启动时，获取的就是新的 MAC 地址。
- (5) MINIGUI 演示：通过/cramfs/sbin/treeview 程序，来演示 MINIGUI。
- (6) 触摸屏演示：通过执行/jffs2/mytest/test 脚本来演示，test 脚本是执行/jffs2/mytest/main 程序（直接执行 main 会出错）。
- (7) 步进电机演示：通过执行/cramfs/sbin/moto_test 程序，启动步进电机，在执行‘停止’命令时，将 moto_test 进程 kill，达到停止步进电机的目的。
- (8) 录音：通过执行/cramfs/sbin/recorder 程序来录音，录音时间在页面中输入，录音文件保存为/tmp/out.wav，按‘放音’键时，执行‘cp /tmp/out.wav /dev/sound/dsp’将录音文件播放。
- (9) 播放 MTV：首先将 U 盘 mount 到/tmp 目录，然后通过执行/jffs2/sbin/play 程序，播放位于 U 盘的 v120x160.avi 文件，程序 play 中将标准输出（STDOUT_FILENO）和标准出错（STDERR_FILENO）重定向到/dev/null（用 dup2（）函数），然后执行/jffs2/sbin/ffplay 程序，播放 MTV。

下面再以播放 MTV 为例，解析一下代码实现过程。

在 config.c 中：

```
else if(strcmp(string.stringout[0],"74")==0)
    showplaymtv();
else if(strcmp(string.stringout[0],"75")==0)
    playmtv()
```

这几行代码表示：当/cgi-bin/config.cgi?flag=74 时调用函数 showplaymtv（），当/cgi-bin/config.cgi?flag=75 时调用函数 playmtv（），其中 showplaymtv（）和 playmtv（）函数在 cgihead.c 中，并在 cgihead.h 中 extern；

```
int showplaymtv()
{
    printf("Content-type:text/html\n\n");    //下面的打印内容为 html 的内容
    printf("
    <html>
    <head>
    <meta http-equiv=pragma content=no-cache><meta http-equiv=expire content=now>
    <title>播放 MTV</title>
    </head>
    <body bgcolor=#F1F1E4 text=black>
    <form method=get action=/cgi-bin/config.cgi>
```



```

<br><br><br>
<div align=center>
<center>
<input type=hidden name=flag value=75 checked>/* 按 键 时 将 调 用
        /cgi-bin/config.cgi?flag=75, 即调用函数 playmtv ( ) */
<br>
欢迎您 MTV 播放演示功能!                //在页面上打印欢迎信息
<br>
<br>
<input type=submit value=确认><br><br><br>    //按键上显示 ‘确认’
</center>
</form>
</body>
</html>
");

return 0;
}

int playmtv()
{
    mount("/dev/sda1","/tmp","vfat",0xc0ed0000,NULL); /*调用 mount() 函数，将 U 盘
        (/dev/sda1)mount 到/tmp 目录，文件格式为 ‘vfat’ */
    sleep(3);                //等待 3 秒
    /* 将 标 准 输 入 STDIN_FILENO、标 准 输 出 STD-OUT_FILENO、标 准 出 错
STDERR_FILENO,三个文件描述符在子进程中关闭*/
    fcntl(STDOUT_FILENO,F_SETFD,FD_CLOEXEC);
    fcntl(STDIN_FILENO,F_SETFD,FD_CLOEXEC);
    fcntl(STDERR_FILENO,F_SETFD,FD_CLOEXEC);
    /*函数 vfork ( ) 复制一个子进程，返回两个值，在子进程中返回值为 0，在父进程中返回
子进程 id 号*/
    if(vfork() == 0)
    {
        //在子进程中执行程序/jffs2/sbin/play
        execl("/jffs2/sbin/play","/jffs2/sbin/play",NULL);
        exit(255);
    }
    save();
    showplaymtv();
}

```

程序 play 是由 playmtv.c 文件生成：

```

int main()
{

```

```

int devnull = open("/dev/null",0);    //打开设备/dev/null
/*用 dup2()函数将标准输出 STDOUT_FILENO 标准出错 STDERR_FILENO, 重定向到描述符 devnull */
dup2(devnull, STDOUT_FILENO);
dup2(devnull, STDERR_FILENO);

if(vfork() == 0){
    /*在子进程中执行程序 ffplay, 由于子进程完全复制父进程, 所以在子进程中, 标准输出和标准出错也将打印到描述符 devnull*/
    execl("/jffs2/sbin/ffplay","/jffs2/sbin/ffplay","-x","240","-y","320","/tmp/v120x160.avi",
        NULL);/
    exit(255);
}
return 0;
}

```

6. CGI 脚本的结构

通用网关接口 CGI 是一个外部应用和信息服务器如 HTTP 或 Web 服务器的标准接口。CGI 程序是一个实时可执行程序，可以动态的输出信息。CGI 脚本结构：一般包括下面三个：

6.1 读取用户的输入表单：

当用户提交表单时，脚本将接收到“名字=值”对形式的表单数据。名字是 INPUT 标签（或者 SELECT，TEXTAREA 标签），值是用户所键入或选择的数据。用户也可以提交文件，但是这里不详细讲述这种情况。

这种名字=值对以长字符串的形式给出，必须分解，但是很简单。这种串的格式是：

"name1=value1&name2=value2&name3=value3" 或者

"name1=value1;name2=value2;name3=value3"

所以只要在&或;，=处作划分就可以了。然后，将所有的“+”转换成空格，将所有的“%xx”序列转换成单个字符，如将“%3d”转换成“=”。因为原始的长串是 URL 编码的，以允许用户输入中含有等号等等。

得到该长字符串的变量要依赖于提交时的 HTML 方法。有两种向服务器提交表单的方法 Get 和 Post。Get 一般用于文件或其他资源，有参数确切指明需要什么数据，用于数据较少的情况。Post 用于向服务器发送大量数据，包括文件，而且每提交一次就调用一次脚本。如果是 GET 提交的，那么长字符串在环境变量 QUERY_STRING 中；若是 POST 提交的，那么读入字节的提取号在环境变量 CONTENT_LENGTH 中。

6.2 处理数据

这是用户需要对数据进行处理的数据。用户可以根据自己不同应用按需编写。

6.3 向标准输出写 HTML 应答：

如果向返回 HTML 数据，那么首先向标准输出写行：Content-type: text/html ，再加一个空行。然后向输出写 HTML 应答页。当脚本被执行时，就会发送给用户。

实际上可以返回任何数据。只需要在“Content-type:”行使用正确的 MIME 类型，并紧跟一个空白行，然后是想返回的原始数据资源。当返回 HTML 文件时，原始数据是 HTML 文本；返回图像，音频，视频时，是原始二进制数据。

如果想深入学习 cgi 程序，可以参考网页：

<http://www.jmarshall.com/easy/cgi/>

7. CGI 的扩展开发步骤

其实很简单，实际上就是在 config.c 为该扩展功能增加一个命令码 flag，并指定它的处理响应函数；然后在 cgihead.c 中完成这个处理响应函数的代码即可。当然也要在存放网页的目录下找到 left.html，加入该扩展功能的连接按钮。

下面就以设置板子的 IP 地址为例，说明如何增加一项管理功能：

- (1) 首先在左边主菜单栏中增加一项：“设置 IP”。这是要修改一个静态页面，即：

uClinux/romfs/home/httpd/html/left.html，增加一项：

```
<tr><td align=center> <a href=/cgi-bin/config.cgi?flag=49 target=main> 设置 IP</a></td></tr>
```

在这里用户可自己为其分配一个命令码 flag=49

- (2) 在 config.c 中为该命令码 49 指定处理函数，即增加如下代码：

```
else if(strcmp(string.stringout[0],"49")==0)
    showsetip();
```

- (3) 在 cgihead.c 中完成 showsetip 函数编码：

这个命令处理函数就是动态生成一个设置 IP 的页面，真正的 IP 设置是用户在这个页面上点击“确认”后才处理。用户在这个动态生成的页面上点击确认后，会调用另一个 CGI 命令处理函数，它是在 showsetip 函数中由如下一行指定的：

```
<input type=hidden name=flag value=48 checked>
```

它指定了用户点击确认后的处理命令码为 48。

- (4) 在 config.c 中为该命令码 49 指定处理函数，即增加如下代码：

```
else if(strcmp(string.stringout[0],"48")==0)
    setip();
```

- (5) 在 cgihead.c 中完成 setip 函数编码：

这个处理代码中用 vfork+execl 执行 ifconfig/route 命令，完成 IP 设置工作，并返回显示操作结果，提供用户重新启动。重新启动后板子的 IP 地址就是更改后的 IP 地址了。

三. 实验内容和步骤

通过 IE 来访问板子，并通过 cgi 来控制板子。掌握如何扩展 CGI 代码，定制自己的 Web 管理软件。

1. 打开 boa.conf 文件，查看里面各个规则项。并验证指定路径下是不是可以找到相应的文件。加深对 boa 服务器配置的理解。
2. 用网线连接一台装有 Windows 的机器。在 IE 窗口输入板子的 IP，访问板子。
在管理界面上，点击相应功能演示菜单，如修改 IP，设置系统时间，MAC 地址设置，MINIGUI 演示等等。
3. 学习扩展 CGI 代码的方法。仔细查看如何实现设置板子的 IP 地址的步骤及代码。对照原理 7 小节中的步骤，找到相应代码如 showsetip()，setip()等，并仔细分析，掌握扩展 CGI 功能的一般方法。

四. 思考题

1. 扩展 CGI 代码，实现其他的功能，如增加用户，定制自己的服务。

实验三：应用程序移植实验

一. 实验目的

通过移植 FTP SERVER 和拨号程序 PPP 这样一些常见的应用程序到 HHARM9-EDU 嵌入式实验平台,让同学们进一步深入理解 LINUX 下开放源代码的工作模式,加深对 make 过程和 Makefile 的理解和熟悉。

二. 实验原理和说明

1、源代码搜索和下载:

打开 IE 浏览器, 进入 www.google.com

键入 pure-ftp tar 这样的搜索关键字, 则可搜索到许多的页面, 其实, 大家熟悉了的话, 就可以记住这样一个开放源代码 PROJECTS 的收集站点, 它就是 www.sourceforge.org, 若搜索到它的话, 就选它即可。当然, 对于 pure-ftpd 这个开源项目的软件, 第一个搜索到的是如下链接, 这也是一个开源项目的站点:

<http://freshmeat.net/projects/pure-ftpd/>

我们下载最新的版本: [1.0.18](#)

2、永远保留一份网上下载的原始 tar 包文件, 用作编译过程的对比之用。

3、刚下载解压后的应用程序, 一般是没有 Makefile 的, 要通过执行一个 ./configure 来自动生成 Makefile。这个 ./configure 程序做的工作主要就是检查编译环境, 例如编译器是否在搜索路径下, 是否提供所有会用到的.h 头文件等, 这是 LINUX 下应用程序最常见通用的做法。下面记录看看它所做的工作:

```
. checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for a BSD-compatible install... /usr/bin/install -c
checking for ranlib... ranlib
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for AIX... no
```

```
checking for library containing strerror... none required
checking for perl... /usr/bin/perl
checking for python... /usr/bin/python
checking for ANSI C header files... yes
checking whether stat file-mode macros are broken... no
checking whether time.h and sys/time.h may both be included... yes
checking for dirent.h that defines DIR... yes
checking for library containing opendir... none required
checking for sys/wait.h that is POSIX.1 compatible... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for unistd.h... (cached) yes
checking for string.h... (cached) yes
checking for strings.h... (cached) yes
checking sys/param.h usability... yes
checking sys/param.h presence... yes
```

.....

它检查 N 多内容，检查没有问题后，生成 Makefile:

```
checking for utimes... yes
checking for mknod... yes
checking for mkfifo... yes
checking for random... yes
checking for srandomdev... no
checking for closefrom... no
checking whether statvfs64() is defined... yes
checking whether snprintf is C99 conformant... done
checking whether getgroups 0 is sane... yes
checking whether realpath likes unreadable directories... yes
checking whether you already have a standard MD5 implementation... no
checking whether you already have a standard SHA1 implementation... no
checking whether we are inside a Virtuozzo virtual host... no
checking default TCP send buffer size... 16384
checking default TCP receive buffer size... 87380
configure: You have /dev/urandom - Great
configure: You have /dev/random - Great
configure: creating ./config.status
config.status: creating Makefile
```

```

config.status: creating src/Makefile
config.status: creating pam/Makefile
config.status: creating man/Makefile
config.status: creating gui/Makefile
config.status: creating configuration-file/Makefile
config.status: creating contrib/Makefile
config.status: creating m4/Makefile
config.status: creating configuration-file/pure-ftpd.conf
config.status: creating configuration-file/pure-config.pl
config.status: creating configuration-file/pure-config.py
config.status: creating puredb/Makefile
config.status: creating puredb/src/Makefile
config.status: creating pure-ftpd.spec
config.status: creating config.h
config.status: executing depfiles commands

```

生成 Makefile 只后，检查一下是否可以在 REDHAT LINUX 上编译通过，若连 REDHAT LINUX 都无法通过编译，那就不要到嵌入式上尝试了；若可以通过编译，则可以着手来修改 Makefile 了：

首先，最基本的就是修改编译器 CC、AR、RANLIB 等为交叉编译工具，这个在上面一节的基本应用开发中都已经有所介绍了：

```
CROSS =/opt/host/armv4l/bin/armv4l-unknown-linux-
```

```
CC = $(CROSS)gcc
```

```
AR = $(CROSS)ar
```

修改 pure-ftpd-1.0.18/src/Makefile，这是 LINUX 下应用软件的常见的构架方式，一个软件名称的目录下，通常会用到三个目录：

- 1) src
- 2) include
- 3) lib

呵呵，感觉就和一个 LINUX 一样了。源代码都放在 src 目录下，所有的.h 头文件都放到 include 目录下，自己用到的私有库函数就放在 lib 下面。

编译一般都是先编译 lib 目录下的.c 代码，然后编译 src 目录下的.c 代码，这样对于编译 src 下代码时的编译参数，必须包含的就是 -I ../include 和 -L ../lib 和链接时的 -lxxx。

对于我们这里的 pureftpd，它没有用到 include 目录和 lib 目录，可能因为相对简单的缘故吧，它所有的.h 头文件都放在 src 下面了，和.c 放在一起。也没有构架私有库函数，所以就用不到 lib 目录了。但它用到一个配置文件 config.h，每个 src 下的.c 都会 include 这个 config.h，所以编译参数都要加入 -I ...。

编译出错了：

首先，在这个文件众多的软件中，make 后信息满屏飞，我们必须先确认是在编译哪个.c 文件的时候出错了。

下面红色字体标出的就是一条编译语句的框架：

```

/opt/host/armv4l/bin/armv4l-unknown-linux-gcc      -I.      -D_GNU_SOURCE=1
      -l/opt/host/armv4l/armv4l-unknown-linux/include      -DCONFDIR="/etc"      -c      -o
      pure_ftpd-caps.o caps.c

```

这个和我们在 REDHAT LINUX PC 上编译一个.c 文件的格式完全一样：

```
gcc -c -o hello.o hello.c
```

```
In file included from ftpd.h:117,
```

```
        from caps.c:8:
```

```
ipv4stack.h:47: warning: `s6_addr' redefined
```

【上面的 warning 不必理会】

```
/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the
previous definition
```

```
In file included from caps.c:10:
```

```
caps_p.h:6: sys/capability.h: No such file or directory
```

这才是错误所在，而这个 sys/capability.h 是指哪个目录下没有呢？这个 sys 又是什么呢？
这样来看：

因为这是 INCLUDE 的问题，所以我们看 Makefile 里面的 -I 参数，其实从上面贴出的编译过程就可以看出：

```
-I. -I.. -I/opt/host/armv4l/armv4l-unknown-linux/include
```

.是当前目录，这个目录下压根没有 sys 目录；

..是 pure-ftp-1.0.18 目录，这个目录下也没有 sys 目录；

所以必然是 /opt/host/armv4l/armv4l-unknown-linux/include 目录下的 sys 目录了。而这个目录下的确没有这个文件。

这时，有两种解决办法，一是去找一个这 capability.h 文件，复制到 sys 目录下；另一种办法就是想法跳过这个文件。

先看看第一种办法：

我们到 /opt/host/armv4l/ 目录下搜索：

```
[/opt/host/armv4l]# find -name capability.h
```

```
./src/linux/include/linux/capability.h
```

呵呵，果然还真有，不过，在 linux 目录下，因为可以看到 /opt/host/armv4l/armv4l-unknown-linux/include 目录下有个 linux 目录，不过是个虚链接：

```
[/opt/host/armv4l/armv4l-unknown-linux/include]# ll linux
```

```
lrwxrwxrwx    1 500      jason          29 Jun 17 14:17 linux -> ../../src/linux/include/linux
```

把这个 .src/linux/include/linux/capability.h 复制到 /opt/host/armv4l/armv4l-unknown-linux/include/sys 目录下，再编译：

呵呵，问题更大了：

```
[hn@Server src]$ make
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc      -I.      -I..      -D_GNU_SOURCE=1
-I/opt/host/armv4l/armv4l-unknown-linux/include  -DCONFDIR="/etc" -c -o pure_ftp-caps.o
caps.c
```

```
In file included from ftpd.h:117,
```

```
        from caps.c:8:
```

```
ipv4stack.h:47: warning: `s6_addr' redefined
```

```
/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the
previous definition
```

```
In file included from /opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:12,
```

```
        from /opt/host/armv4l/armv4l-unknown-linux/include/sys/capability.h:17,
```

```
        from caps_p.h:6,
```

```
        from caps.c:10:
```

/opt/host/armv4l/armv4l-unknown-linux/include/linux/wait.h:4: warning: `WNOHANG' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/bits/waitflags.h:26: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/wait.h:5: warning: `WUNTRACED' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/bits/waitflags.h:27: warning: this is the location of the previous definition
In file included from /opt/host/armv4l/armv4l-unknown-linux/include/sys/capability.h:17,
from caps_p.h:6,
from caps.c:10:
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:47: warning: `BLOCK_SIZE' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:28: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:107: warning: `MS_RDONLY' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:37: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:108: warning: `MS_NOSUID' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:39: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:109: warning: `MS_NODEV' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:41: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:110: warning: `MS_NOEXEC' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:43: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:111: warning: `MS_SYNCHRONOUS' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:45: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:112: warning: `MS_REMOUNT' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:47: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:113: warning: `MS_MANDLOCK' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:49: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:114: warning: `MS_NOATIME' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:57: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:115: warning: `MS_NODIRATIME' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:59: warning: this is the location of the previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:116: warning: `MS_BIND' redefined


```
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:61: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:122: warning: `MS_RMT_MASK'
redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:65: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:127: warning: `MS_MGC_VAL'
redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:70: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:135: warning: `S_APPEND' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:53: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:136: warning: `S_IMMUTABLE'
redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:55: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:172: warning: `BLKROSET' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:78: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:173: warning: `BLKROGET' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:79: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:174: warning: `BLKRRPART' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:80: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:175: warning: `BLKGETSIZE' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:81: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:176: warning: `BLKFLSBUF' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:82: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:177: warning: `BLKRASET' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:83: warning: this is the location of the
previous definition
/opt/host/armv4l/armv4l-unknown-linux/include/linux/fs.h:178: warning: `BLKRASET' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/sys/mount.h:84: warning: this is the location of the
previous definition
In file included from caps.c:10:
caps_p.h:9: parse error before `cap_keep_startup'
caps_p.h:18: warning: data definition has no type or storage class
caps_p.h:20: parse error before `cap_keep_login'
caps_p.h:28: warning: data definition has no type or storage class
caps.c:16: parse error before `*'
```

```
caps.c: In function `apply_caps':
caps.c:20: `cap_t' undeclared (first use in this function)
caps.c:20: (Each undeclared identifier is reported only once
caps.c:20: for each function it appears in.)
caps.c:20: parse error before `caps'
caps.c:25: `caps' undeclared (first use in this function)
caps.c:25: parse error before `0'
caps.c: In function `drop_login_caps':
caps.c:42: `cap_value_t' undeclared (first use in this function)
caps.c: In function `set_initial_caps':
caps.c:48: `cap_value_t' undeclared (first use in this function)
make: *** [pure_ftpd-caps.o] Error 1
```

不要理会前面长篇大论的 warning，也不要看后面的这些错误，看编译错误一定要看第一个，就是上面黑体标出的。这种错误就是说这个变量的类型没有定义，而这原来是 OK 的，这就是由于引入那个 `capability.h` 所引起的，这样的错误就不好解决了，我们暂且放下了。

再来看第二种方法：

我们找到罪魁祸首，就是那个

```
caps_p.h:6: sys/capability.h: No such file or directory
```

这个 `caps_p.h` 就在 `pure-ftpd-1.0.18/src` 目录下。

```
[hn@Server src]$ vim caps_p.h
```

```
#ifndef USE_CAPABILITIES
# ifdef HAVE_SYS_CAPABILITY_H
#  include <sys/capability.h>
# endif
```

这是个典型的 LINUX 下应用程序的配置问题，一般的，稍微复杂的应用软件都有自己的配置文件，就类似 LINUX 内核的 `.config/autoconf.h`，WINDOWS 下的注册表和 `.ini` 文件等。例如 `microwin` 的 `src/config` 文件，`busybox` 的 `.config` 文件，对于 `pure-ftpd` 而言，就是 `config.h` 文件。对于这些配置的宏定义，都在这些文件里面给出，用户要修改配置，就直接修改这些文件即可。其实，若您不知道这个文件的存在，可以直接查找这个宏在哪里有定义即可：

```
grep HAVE_SYS_CAPABILITY_H * -r
```

也可以发现它的定义之处 `pure-ftpd-1.0.18/config.h`。

```
vim config.h
```

```
/* Define to 1 if you have the <sys/capability.h> header file. */
```

```
#define HAVE_SYS_CAPABILITY_H 1
```

正如我们的编译器没有提供 `<sys/capability.h>`，则我们需要注释掉这行：

```
//#define HAVE_SYS_CAPABILITY_H 1
```

改完之后，再 `make`，发现：

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -I. -I.. -D_GNU_SOURCE=1
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o pure_ftpd-caps.o
caps.c
```

```
In file included from ftpd.h:117,
```

```
from caps.c:8:
```

```
ipv4stack.h:47: warning: `s6_addr' redefined
```

/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the previous definition

In file included from caps.c:10:

caps_p.h:9: parse error before `cap_keep_startup'

caps_p.h:10: `CAP_SETGID' undeclared here (not in a function)

caps_p.h:10: initializer element is not constant

caps_p.h:10: (near initialization for `cap_keep_startup[0]')

caps_p.h:11: `CAP_SETUID' undeclared here (not in a function)

caps_p.h:11: initializer element is not constant

caps_p.h:11: (near initialization for `cap_keep_startup[1]')

caps_p.h:12: `CAP_CHOWN' undeclared here (not in a function)

caps_p.h:12: initializer element is not constant

caps_p.h:12: (near initialization for `cap_keep_startup[2]')

caps_p.h:13: `CAP_NET_BIND_SERVICE' undeclared here (not in a function)

caps_p.h:13: initializer element is not constant

caps_p.h:13: (near initialization for `cap_keep_startup[3]')

caps_p.h:14: `CAP_SYS_CHROOT' undeclared here (not in a function)

caps_p.h:14: initializer element is not constant

caps_p.h:14: (near initialization for `cap_keep_startup[4]')

caps_p.h:15: `CAP_SYS_NICE' undeclared here (not in a function)

caps_p.h:15: initializer element is not constant

caps_p.h:15: (near initialization for `cap_keep_startup[5]')

caps_p.h:16: `CAP_NET_ADMIN' undeclared here (not in a function)

caps_p.h:16: initializer element is not constant

caps_p.h:16: (near initialization for `cap_keep_startup[6]')

caps_p.h:18: `CAP_DAC_READ_SEARCH' undeclared here (not in a function)

caps_p.h:18: initializer element is not constant

caps_p.h:18: (near initialization for `cap_keep_startup[7]')

caps_p.h:18: warning: data definition has no type or storage class

caps_p.h:20: parse error before `cap_keep_login'

caps_p.h:25: `CAP_NET_BIND_SERVICE' undeclared here (not in a function)

caps_p.h:25: initializer element is not constant

caps_p.h:25: (near initialization for `cap_keep_login[0]')

caps_p.h:28: `CAP_NET_ADMIN' undeclared here (not in a function)

caps_p.h:28: initializer element is not constant

caps_p.h:28: (near initialization for `cap_keep_login[1]')

caps_p.h:28: warning: data definition has no type or storage class

caps.c:16: parse error before `*'

caps.c: In function `apply_caps':

caps.c:20: `cap_t' undeclared (first use in this function)

caps.c:20: (Each undeclared identifier is reported only once

caps.c:20: for each function it appears in.)

caps.c:20: parse error before `caps'

caps.c:25: `caps' undeclared (first use in this function)

```
caps.c:25: parse error before `0'
caps.c: In function `drop_login_caps':
caps.c:42: `cap_value_t' undeclared (first use in this function)
caps.c: In function `set_initial_caps':
caps.c:48: `cap_value_t' undeclared (first use in this function)
make: *** [pure_ftpdcaps.o] Error 1
```

看来还是不行，那就只有整个的将"caps_p.h"文件里面的内容去掉了，即：

```
vim config.h
```

```
/* use capabilities HHTECH*/
```

```
##define USE_CAPABILITIES
```

然后再 make，就发现

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -I. -I.. -D_GNU_SOURCE=1
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o pure_ftpdcaps.o
caps.c
```

编译通过了。但这样修改是否会有后遗症，还要看后面编译的结果。

```
if /opt/host/armv4l/bin/armv4l-unknown-linux-gcc -DHAVE_CONFIG_H -I. -I.. -I..
-D_GNU_SOURCE=1 -I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc"
-DINCLUDE_IO_WRAPPERS=1 -g -O2 -MT pure_ftpdcaps.o -MD -MP -MF
".deps/pure_ftpdcaps.o" -c -o pure_ftpdcaps.o `test -f 'ftp_parser.c' || echo
'./'ftp_parser.c; \
then mv -f ".deps/pure_ftpdcaps.o" ".deps/pure_ftpdcaps.Po"; else rm -f
".deps/pure_ftpdcaps.o"; exit 1; fi
armv4l-unknown-linux-gcc: cannot specify -o with -c or -S and multiple compilations
make: *** [pure_ftpdcaps.o] Error 1
```

对于这种错误，就是明显的 Makefile 的参数问题所致，我们看看原来的 Makefile 的写法：

```
if $(CC) $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)
$(pure_ftpdcflags) $(CFLAGS) -MT pure_ftpdcaps.o -MD -MP -MF
"$(DEPDIR)/pure_ftpdcaps.o" -c -o pure_ftpdcaps.o `test -f 'ftp_parser.c' || echo
'$(srcdir)/'ftp_parser.c; \
then mv -f "$(DEPDIR)/pure_ftpdcaps.o" "$(DEPDIR)/pure_ftpdcaps.Po";
else rm -f "$(DEPDIR)/pure_ftpdcaps.o"; exit 1; fi
```

这是现在很常见的一种 Makefile 的写法，很多应用都采用了这种形式，常见的还有 madplay 等。它为每个.c 建立了对.h 的依赖关系，并在编译器前检查是否有缺失这个.c 文件，其实这些都是没有什么实际的用处，而且这种写法对于我们目前使用交叉编译器而言有问题，所以我们必须将其改写。我们将其简化改为：

```
$(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftpdcaps.o ftp_parser.c
```

即可通过编译。

但可以看出，这个 Makefile 是对每个.c 都写了这样一行编译的，所以最烦杂的就是要逐行修改这为数众多的 Makefile 内容。

最后改为：

```
PUREFTPD_OBJS = pure_ftpdcaps.o \
    pure_ftpdcaps.o pure_ftpdcaps.o \
    pure_ftpdcaps.o pure_ftpdcaps.o \
```

```

pure_ftp-log_pam.o pure_ftp-log_ldap.o \
pure_ftp-log_puredb.o pure_ftp-log_extauth.o \
pure_ftp-ls.o pure_ftp-parser.o \
pure_ftp-bsd-glob.o pure_ftp-fakesnprintf.o \
pure_ftp-bsd-realpath.o \
pure_ftp-mysnprintf.o pure_ftp-caps.o \
pure_ftp-ftp_parser.o pure_ftp-dynamic.o \
pure_ftp-ftpwho-update.o \
pure_ftp-bsd-getopt_long.o \
pure_ftp-upload-pipe.o pure_ftp-ipv4stack.o \
pure_ftp-altlog.o pure_ftp-crypto.o \
pure_ftp-crypto-md5.o pure_ftp-crypto-sha1.o \
pure_ftp-quotas.o pure_ftp-fakechroot.o \
pure_ftp-diraliases.o pure_ftp-ftpwho-read.o \
pure_ftp-getloadavg.o pure_ftp-privsep.o \
pure_ftp-tls.o pure_ftp-osx-extensions.o

```

all: pure-ftp

pure-ftp: \$(PUREFTPDOBJS)

\$(CC) \$(CFLAGS) \$(LDFLAGS) -o pure-ftp \$(PUREFTPDOBJS) \$(LIBS)

```

#####
#####

```

pure_ftp-daemons.o: daemons.c

\$(CC) -I. -c -o pure_ftp-daemons.o daemons.c

pure_ftp-ftp.o: ftpd.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-ftp.o ftpd.c

pure_ftp-log_unix.o: log_unix.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-log_unix.o log_unix.c

pure_ftp-log_mysql.o: log_mysql.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-log_mysql.o log_mysql.c

pure_ftp-log_pgsql.o: log_pgsql.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-log_pgsql.o log_pgsql.c

pure_ftp-log_pam.o: log_pam.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-log_pam.o log_pam.c

pure_ftp-log_ldap.o: log_ldap.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-log_ldap.o log_ldap.c

pure_ftp-log_puredb.o: log_puredb.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-log_puredb.o log_puredb.c

pure_ftp-log_extauth.o: log_extauth.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-log_extauth.o log_extauth.c

pure_ftp-ls.o: ls.c

\$(CC) -I. -I.. \$(CPPFLAGS) -c -o pure_ftp-ls.o ls.c

```
pure_ftp-parser.o: parser.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-parser.o parser.c
pure_ftp-bsd-glob.o: bsd-glob.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-bsd-glob.o bsd-glob.c
pure_ftp-fakesnprintf.o: fakesnprintf.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-fakesnprintf.o fakesnprintf.c
pure_ftp-bsd-realpath.o: bsd-realpath.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-bsd-realpath.o bsd-realpath.c
pure_ftp-mysnprintf.o: mysnprintf.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-mysnprintf.o mysnprintf.c
pure_ftp-caps.o: caps.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-caps.o caps.c
pure_ftp-ftp_parser.o: ftp_parser.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-ftp_parser.o ftp_parser.c
pure_ftp-dynamic.o: dynamic.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-dynamic.o dynamic.c
pure_ftp-ftpwho-update.o: ftpwho-update.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-ftpwho-update.o ftpwho-update.c
pure_ftp-bsd-getopt_long.o: bsd-getopt_long.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-bsd-getopt_long.o bsd-getopt_long.c
pure_ftp-upload-pipe.o: upload-pipe.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-upload-pipe.o upload-pipe.c
pure_ftp-ipv4stack.o: ipv4stack.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-ipv4stack.o ipv4stack.c
pure_ftp-altlog.o: altlog.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-altlog.o altlog.c
pure_ftp-crypto.o: crypto.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-crypto.o crypto.c
pure_ftp-crypto-md5.o: crypto-md5.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-crypto-md5.o crypto-md5.c
pure_ftp-crypto-sha1.o: crypto-sha1.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-crypto-sha1.o crypto-sha1.c
pure_ftp-quotas.o: quotas.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-quotas.o quotas.c
pure_ftp-fakechroot.o: fakechroot.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-fakechroot.o fakechroot.c
pure_ftp-diraliases.o: diraliases.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-diraliases.o diraliases.c
pure_ftp-ftpwho-read.o: ftpwho-read.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-ftpwho-read.o ftpwho-read.c
pure_ftp-getloadavg.o: getloadavg.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-getloadavg.o getloadavg.c
pure_ftp-privsep.o: privsep.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftp-privsep.o privsep.c
```

```

pure_ftpd-tls.o: tls.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftpd-tls.o tls.c
pure_ftpd-osx-extensions.o: osx-extensions.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_ftpd-osx-extensions.o osx-extensions.c
pure_mrtginfo-daemons.o: daemons.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_mrtginfo-daemons.o daemons.c
pure_mrtginfo-pure-mrtginfo.o: pure-mrtginfo.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_mrtginfo-pure-mrtginfo.o pure-mrtginfo.c
pure_mrtginfo-fakesnprintf.o: fakesnprintf.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_mrtginfo-fakesnprintf.o fakesnprintf.c
pure_mrtginfo-mysnprintf.o: mysnprintf.c
    $(CC) -I. -I.. $(CPPFLAGS) -c -o pure_mrtginfo-mysnprintf.o mysnprintf.c

```

clean:

```
rm -f $(PPDOBJ) pure-ftpd *.o *~ #* core
```

则一次编译通过。

不过，可以看出，上面的改法非常的愚笨，而之所以造成这种困境的原因就在于我们的要求生成的.o 和源文件.c 名字不一致，从而造成 gcc 编译的隐性规则无法应用。这是该项目软件人员的设计的代码命名管理方式，在每个.o 文件名前面加了 pure_ftpd-的前缀，但这对我们简化的嵌入式应用没有必要，所以为了省去这每个.o 生成的规则设置，我们还是采用.o 和.c 同名的方式，使得隐性规则生效。

修改如下：

```
CPPFLAGS = -D_GNU_SOURCE=1 -I. -I.. -I/opt/host/armv4l/armv4l-unknown-linux/include
```

```

PUREFTPDOBJ = daemons.o \
    ftpd.o log_unix.o \
    log_mysql.o log_pgsql.o \
    log_pam.o log_ldap.o \
    log_puredb.o log_extauth.o \
    ls.o parser.o \
    bsd-glob.o fakesnprintf.o \
    bsd-realpath.o \
    mysnprintf.o caps.o \
    ftp_parser.o dynamic.o \
    ftpwho-update.o \
    bsd-getopt_long.o \
    upload-pipe.o ipv4stack.o \
    altlog.o crypto.o \
    crypto-md5.o crypto-sha1.o \
    quotas.o fakechroot.o \
    diraliases.o ftpwho-read.o \
    getloadavg.o privsep.o \
    tls.o osx-extensions.o

```

all: pure-ftpd

pure-ftpd: \$(PUREFTPDOBJS)

\$(CC) \$(CFLAGS) \$(LDFLAGS) -o pure-ftpd \$(PUREFTPDOBJS) \$(LIBS)

整个的编译过程如下：

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o daemons.o
daemons.c
```

/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the previous definition

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o log_mysql.o
log_mysql.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o log_pgsq.o
log_pgsq.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o log_pam.o
log_pam.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o log_ldap.o
log_ldap.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o log_puredb.o
log_puredb.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o log_extauth.o
log_extauth.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o ls.o ls.c
```

/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the previous definition

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o parser.o parser.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o bsd-glob.o
bsd-glob.c
```

In file included from ftpd.h:117,

from bsd-glob.c:60:

ipv4stack.h:47: warning: `s6_addr' redefined

/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the previous definition


```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..  
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o fakesnprintf.o  
fakesnprintf.c
```

In file included from ftpd.h:117,

from fakesnprintf.c:31:

ipv4stack.h:47: warning: `s6_addr' redefined

/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the
previous definition

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..  
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o bsd-realpath.o  
bsd-realpath.c
```

In file included from ftpd.h:117,

from bsd-realpath.c:37:

ipv4stack.h:47: warning: `s6_addr' redefined

/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the
previous definition

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..  
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o mysnpprintf.o  
mysnpprintf.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..  
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o caps.o caps.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..  
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o ftp_parser.o  
ftp_parser.c
```

In file included from ftpd.h:117,

from ftp_parser.c:3:

ipv4stack.h:47: warning: `s6_addr' redefined

/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the
previous definition

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..  
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o dynamic.o  
dynamic.c
```

In file included from ftpd.h:117,

from dynamic.c:13:

ipv4stack.h:47: warning: `s6_addr' redefined

/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the
previous definition

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..  
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o ftpwho-update.o  
ftpwho-update.c
```

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..  
-I/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o bsd-getopt_long.o  
bsd-getopt_long.c
```

In file included from ftpd.h:117,

```

from bsd-getopt_long.c:59:
ipv4stack.h:47: warning: `s6_addr' redefined
/opt/host/armv4l/armv4l-unknown-linux/include/netinet/in.h:175: warning: this is the location of the
previous definition
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o upload-pipe.o
upload-pipe.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o ipv4stack.o
ipv4stack.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o altlog.o altlog.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o crypto.o crypto.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o crypto-md5.o
crypto-md5.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o crypto-sha1.o
crypto-sha1.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o quotas.o quotas.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o fakechroot.o
fakechroot.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o diraliases.o
diraliases.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o ftpwho-read.o
ftpwho-read.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o getloadavg.o
getloadavg.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o privsep.o privsep.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o tls.o tls.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I..
-l/opt/host/armv4l/armv4l-unknown-linux/include -DCONFDIR="/etc" -c -o osx-extensions.o
osx-extensions.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2
-L/opt/host/armv4l/armv4l-unknown-linux/lib -o pure-ftpd daemons.o ftpd.o log_unix.o
log_mysql.o log_pgsq.o log_pam.o log_ldap.o log_puredb.o log_extauth.o ls.o parser.o

```

```
bsd-glob.o fakesnprintf.o bsd-realpath.o mysnprintf.o caps.o ftp_parser.o dynamic.o
ftpwho-update.o bsd-getopt_long.o upload-pipe.o ipv4stack.o altlog.o crypto.o crypto-md5.o
crypto-sha1.o quotas.o fakechroot.o diraliases.o ftpwho-read.o getloadavg.o privsep.o tls.o
osx-extensions.o -lcrypt
```

通过编译！

【注】

1、**-lcrypt** 就表明在链接时链入

-L/opt/host/armv4l/armv4l-unknown-linux/lib/libcrypt.a 库文件。看看：

```
[/HHARM9-EDU/applications]# ll /opt/host/armv4l/armv4l-unknown-linux/lib/libcrypt.a
```

```
-rw-r--r--      1  500                jason          25014  Feb  10   2001
```

```
/opt/host/armv4l/armv4l-unknown-linux/lib/libcrypt.a
```

2、其中，上面编译过程中，可不必指定

-I/opt/host/armv4l/armv4l-unknown-linux/include 和

-L/opt/host/armv4l/armv4l-unknown-linux/lib

因为这是在 **/opt/host/armv4l/bin/armv4l-unknown-linux-gcc** 编译器里面指定死的 **INCLUDE** 路径和 **LIB** 路径，不必再认为指定了，只有不同于这个的路径才需要显式的指定，例如 **-I. -I..** 等。同学们可以试试从里面去掉这些参数，看是否可以通过编译。

至此，大家可以从总体上对一个多文件的程序的编译有个了解，很多从网上下载下来的应用程序的 **Makefile** 写得非常复杂，实际上，编译的过程用下面两行就可以描述的深入骨髓：

```
gcc -c -o -I. hello.o hello.c
```

```
gcc -o hello hello1.o hello2.o
```

把网上下载的复杂的 **Makefile** 扒皮剔肉后，剩下的骨架就是上面的形式，就是

编译+链接两个步骤而已，这个无论是 **DOS** 下的 **TURBO C** 还是 **WINDOWS** 下的 **VC++** 都是一样的道理。

所以，不要为表面的复杂所迷惑，拿到 **Makefile** 首先从 **all:** 入手看起（它就相当于 **C** 代码的 **main** 函数的地位），看出它要编译几个 **.o**，然后链接起来即可。编译就用隐性规则即可，只要写一句链接的规则即可。其中要注意的就是 **INCLUDE** 路径和库的路径的设置。

下面再来看看 **pppd** 的编译

从网上下载稳定的版本：**ppp-2.4.1/**

看看它的目录结构：

```
[hn@Server ppp-2.4.1]$ ls
Changes-2.3      makelog          README.sol2      ttt
FAQ              PLUGINS          README.cbcp      ttt
configure       Makefile         README.linux     SETUP
                README.MSCHAP80
```

它主要的目录就是 **pppd** 和 **include** 目录，其中 **pppd** 就相当于 **src** 目录。

修改了 **pppd/Makefile** 的 **CC** 等参数，**pppd** 的 **Makefile** 清晰明了，非常好改，直接编译，过程如下：

```
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -O2 -pipe -Wall -g -D_linux_=1
-DHAVE_PATHS_H -DIPX_CHANGE -DHAVE_MULTILINK -DHAVE_MMAP -I../include
```

```

-DCHAPMS=1 -DUSE_CRYPT=1 -DHAVE_CRYPT_H=1 -DHAS_SHADOW -DPLUGIN -c -o
main.o main.c
main.c: In function `device_script':
main.c:1349: warning: argument `in' might be clobbered by `longjmp' or `vfork'
main.c:1349: warning: argument `out' might be clobbered by `longjmp' or `vfork'
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -O2 -pipe -Wall -g -D_linux_=1
-DHAVE_PATHS_H -DIPX_CHANGE -DHAVE_MULTILINK -DHAVE_MMAP -I../include
-DCHAPMS=1 -DUSE_CRYPT=1 -DHAVE_CRYPT_H=1 -DHAS_SHADOW -DPLUGIN -c -o
magic.o magic.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -O2 -pipe -Wall -g -D_linux_=1
-DHAVE_PATHS_H -DIPX_CHANGE -DHAVE_MULTILINK -DHAVE_MMAP -I../include
-DCHAPMS=1 -DUSE_CRYPT=1 -DHAVE_CRYPT_H=1 -DHAS_SHADOW -DPLUGIN -c -o
fsm.o fsm.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -O2 -pipe -Wall -g -D_linux_=1
-DHAVE_PATHS_H -DIPX_CHANGE -DHAVE_MULTILINK -DHAVE_MMAP -I../include
-DCHAPMS=1 -DUSE_CRYPT=1 -DHAVE_CRYPT_H=1 -DHAS_SHADOW -DPLUGIN -c -o
lcp.o lcp.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -O2 -pipe -Wall -g -D_linux_=1
-DHAVE_PATHS_H -DIPX_CHANGE -DHAVE_MULTILINK -DHAVE_MMAP -I../include
-DCHAPMS=1 -DUSE_CRYPT=1 -DHAVE_CRYPT_H=1 -DHAS_SHADOW -DPLUGIN -c -o
ipcp.o ipcp.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -O2 -pipe -Wall -g -D_linux_=1
-DHAVE_PATHS_H -DIPX_CHANGE -DHAVE_MULTILINK -DHAVE_MMAP -I../include
-DCHAPMS=1 -DUSE_CRYPT=1 -DHAVE_CRYPT_H=1 -DHAS_SHADOW -DPLUGIN -c -o
upap.o upap.c
.....
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -O2 -pipe -Wall -g -D_linux_=1
-DHAVE_PATHS_H -DIPX_CHANGE -DHAVE_MULTILINK -DHAVE_MMAP -I../include
-DCHAPMS=1 -DUSE_CRYPT=1 -DHAVE_CRYPT_H=1 -DHAS_SHADOW -DPLUGIN -c -o
md4.o md4.c
/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -O2 -pipe -Wall -g -D_linux_=1
-DHAVE_PATHS_H -DIPX_CHANGE -DHAVE_MULTILINK -DHAVE_MMAP -I../include
-DCHAPMS=1 -DUSE_CRYPT=1 -DHAVE_CRYPT_H=1 -DHAS_SHADOW -DPLUGIN --static
-Wl,-E -o pppd main.o magic.o fsm.o lcp.o ipcp.o upap.o chap.o md5.o ccp.o auth.o options.o
demand.o utils.o sys-linux.o ipxcp.o multilink.o tdb.o tty.o md4.o chap_ms.o -lcrypt -ldl
结果在链接时，出错啦：
/opt/host/armv4l/armv4l-unknown-linux/lib/libc.a(nsswitch.o)(.data+0x64): undefined reference to
`_nss_files_getaliasent_r'
/opt/host/armv4l/armv4l-unknown-linux/lib/libc.a(nsswitch.o)(.data+0x6c): undefined reference to
`_nss_files_endaliasent'
/opt/host/armv4l/armv4l-unknown-linux/lib/libc.a(nsswitch.o)(.data+0x74): undefined reference to
`_nss_files_setaliasent'
/opt/host/armv4l/armv4l-unknown-linux/lib/libc.a(nsswitch.o)(.data+0x84): undefined reference to
`_nss_files_getetherent_r'

```

```
/opt/host/armv4l/armv4l-unknown-linux/lib/libc.a(nsswitch.o)(.data+0x8c): undefined reference to
`_nss_files_endetherent'
/opt/host/armv4l/armv4l-unknown-linux/lib/libc.a(nsswitch.o)(.data+0x94): undefined
.....
```

从上面信息可以清晰的看出，是 S3C2410 自带的编译器出问题了，它提供的 libc 库 /opt/host/armv4l/armv4l-unknown-linux/lib/libc.a 有问题，里面缺了很多函数的实现。遇到这种问题，除非我们去重新编译生成一套 FOR S3C2410 的 arm-linux-gcc，否则很难解决。不过，可以有个变通的办法，就是我们去找一个 FOR ARM9 的 libc.a 文件来替换。

我们看到：

```
[/opt/host/armv4l]# ll ./armv4l-unknown-linux/lib/libc.a
```

```
-rw-r--r-- 1 500 jason 2770984 Feb 10 2001 ./armv4l-unknown-linux/lib/libc.a
```

这个只有 2.7MB 大小，而我们看看 AT91RM9200 带的编译器里面的 libc.a 和 MXL 的 libc.a，都有 20MB 以上的大小。可见，S3C2410 带的这个编译器里面的库是非常节俭的，缺失了很多库函数。我们试用一下 AT91RM9200 的 libc.a，覆盖

```
/opt/host/armv4l/armv4l-unknown-linux/lib/libc.a,
```

然后重新回来 pppd 目录编译：

发生了新的错误：

```
/opt/host/armv4l/armv4l-unknown-linux/bin/ld: truncate.o: compiled for a big endian system and
target is little endian
```

这又是大小头不匹配的问题，还是不行！。

仔细研究一下，原来是 2410 发行的 libc.a 里做了拆分，将一些函数的实现分割到其它几个.a 库文件（例如 libnss_dns.a）里面去了，导致 libc.a 里面的库函数不全，所以我们需要做的就是将这些分割出去的.a 再合并回到 libc.a 里面即可。

具体命令就是：

```
/opt/host/armv4l/bin/armv4l-unknown-linux-ar ru libc.a xxx xxx
```

还有一种办法，我们看到上面的编译参数里面有 **--static**，这说明我们采用的是静态编译，我们可换用动态编译方式，去掉 Makefile 里面的 **--static** 即可顺利的通过编译。

```
#LDFLAGS += --static
```

最后，我们回头来看看交叉编译器的问题。

这些交叉编译器都是在 REDHAT 上用 gcc 编译生成的，编译生成交叉编译器，需要用到的源代码有：gcc 的源代码（设置了 CPU ARCHITECTURE）、libc 的源代码、LINUX 内核源代码（主要用于给编译器提供头文件）等。可以看出，这些交叉编译器是内置了 libc 库的，这些库不同，就可以有不同的编译器生成，可选的 libc 库有 glibc 和 uClibc。

下面是一篇非常好的文章，介绍如何用源代码自己编译生成 FOR ARM 的 arm-linux-gcc 交叉编译器。

<http://www.princeton.edu/~wqin/build.htm>

【注】

uClibc 作为一种高度精简的 libc 库，和 uClinux 并没有任何关系，它可以应用于任何的处理器平台。用 uClibc 构架编译出来的交叉编译器，可使得生成的代码更小。当然了，小是要付出代价的，例如很多复杂的应用程序就无法用这种 uClibc 的编译器编译过去，因为它的库太简单了，无法对一些复杂的应用程序提供支持。

还有，同一个处理器平台，可以支持用多种编译器编译出来的应用程序，例如对于 S3C2410 平台，用它自己带的 /opt/host/armv4l/bin/armv4l-unknown-linux-gcc 编译出来程序可以运行，而用 AT91RM9200 带的 /usr/local/arm/2.95.3/bin/arm-linux-gcc 编译出来的应用程序也可以在 S3C2410

平台上执行。同理，用 MXL9328 带的 arm-linux-gcc 静态编译的应用程序也可以在 S3C2410 平台上运行。

【注】

这里可以跨平台互相执行的应用程序必须是静态编译的独立运行的程序。

三. 实验内容和步骤

1、测试 gcc 的隐式规则

【注】

关于 gcc 编译的隐性规则的说明：

除用户定义变量外，make 也允许使用环境变量、自动变量和预定义变量。使用环境变量非常简单。在启动的时，make 读取已定义的环境变量，并且创建与之同名同值的变量。但是，如果 makefile 中有同名的变量，则这个变量将取代与之相应的环境变量，所以应当注意这一点。此外，make 也提供一长串预定义变量和自动变量，但是它们看起来有些神秘。表 1 给出了部分自动变量：

表 1 自动变量

变量	说明
\$@	规则的目标所对应的文件名
\$<	规则中的第一个相关文件名
\$^	规则中所有相关文件的列表，以空格为分隔符
\$?	规则中日期新于目标的所有相关文件的列表，以空格为分隔符
\$(@D)	目标文件的目录部分（如果目标在子目录中）
\$(@F)	目标文件的文件名部分（如果目标在子目录中）

除了表 1 中所列的自动变量之外，make 预定义了许多其他变量，用于定义程序名及传给这些程序的参数和标志，见表 2。

表 2 用于程序名和标志的预定义变量

变量	说明
AR	归档维护程序，缺省值=ar
AS	汇编程序，缺省值=ar
CC	C 编译程序，缺省值=cc
CPP	C 预处理程序，缺省值=cpp
RM	文件删除程序，缺省值=“rm-f”
ARFLAGS	传给汇编程序的标志，没有缺省值
CFLAGS	传给 C 编译器的标志，没有缺省值
CPPFLAGS	传给 C 预处理器的标志，没有缺省值
LDLFLAGS	传给链接程序（ld）的标志，没有缺省值

如果需要，可以在 makefile 中重新定义这些变量，但是在大多数情况下，这些缺省值都是合理的。

隐式规则

除在 Makefile 中显式定义的规则（显式规则）外，make 还有一系列的隐式规则（或称为预定义规则）集。其中的大部分出于特殊的目的或用途有限，所以在这里只介绍最常用的那些隐式规则。隐式规则可以简化 Makefile 的维护。

假设有下面这样的一个 Makefile:

```
1 OBJS=editor.o screen.o keyboard.o
```

```

2 editor : $(OBSJ)
3   cc-o editor $ (OBSJ)
4
5 .PHONY:clean
6
7 clean
8   rm ediot $ (OBSJ)

```

缺省目标 **editor** 所对应的命令提及了 **editor.o,screen.o**,但是 **Makefile** 中没有怎样编译生成这些目标的规则。此时, **make** 就使用所谓的隐式规则,实际上,对每一个名为 **somefile.o** 目标(object)文件, **make** 首先找到与之相应的源代码 **somefile.c**,并且用 **gcc somefile.c-o somefile.o** 编译生成这个目标文件。所以,在本例中 **make** 先查找名为 **editor.c,screen.c** 和 **keyboard.c** 的文件并将它们编译为目标文件 (**editor.o, screen.o** 和 **keyboard**),然后,编译生成缺省目标 **editor**。

实际的机制比这里所描述的要全面。目标文件(, o)可以从 **C,Pascal, Fortran** 等源代码中生成,所以 **make** 也应去查找符合实际情况的相关文件。所以,如果在工作目录下有 **editor.p,screen.p** 和 **keyboard.p** 三个 **Pascal** 文件(.p 通常被认为式 **Pascal** 源代码的扩展名),**make** 就会激活 **Pascal** 编译器来编译它们。因此,如果出于某种原因而在项目中需要使用多种语言时,就不能相关隐式规则,因为此时使用规则所得到的结果可能会与期望的有所不同。

模式规则

通过定义用户自己的隐式规则,模式规则则提供了扩展 **make** 的隐式规则的一种方法。模式规则类似于普通规则,但是它的目标必定含有符号%,这个符号可以与任何非空字符串匹配:为与目标中的%匹配,这个规则的相关文件部分也必须使用%例如,下面的规则:

```
%.o:%.c
```

告诉 **make** 所有形为 **somefile.o** 的目标(object)文件都应从 **somefile.c** 编译而来。与隐式规则一样, **make** 预定义了一些模式规则:

```
%.o:%.c
```

```
$ (CC)-c $ (CFLAGS) $ (CPPFLAGS) $< -o $@
```

make 定义了一条规则,即任何 **x.o** 文件都从 **x.c** 编译而来,每次使用该规则时,该规则用自动变量 **\$<**和 **\$@**来代替第一个相关名文件和目标,此外,变量 **\$(CC),\$(CFLAGS)**和 **(CPPFLAGS)**的缺省值如表 4.2 所示。

现在来做实验,即改变默认的预定义的隐式规则:

Makefile 还是用那种名字不匹配的.o 写法:

```

PUREFTPDOBJS = pure_ftp-daemons.o \
    pure_ftp-ftp.o pure_ftp-log_unix.o \
    pure_ftp-log_mysql.o pure_ftp-log_pgsql.o \
    pure_ftp-log_pam.o pure_ftp-log_ldap.o \
    pure_ftp-log_puredb.o pure_ftp-log_extauth.o \
    pure_ftp-ls.o pure_ftp-parser.o \
    pure_ftp-bsd-glob.o pure_ftp-fakesnprintf.o \
    pure_ftp-bsd-realpath.o \
    pure_ftp-mysnprintf.o pure_ftp-caps.o \
    pure_ftp-ftp_parser.o pure_ftp-dynamic.o \
    pure_ftp-ftpwho-update.o \
    pure_ftp-bsd-getopt_long.o \
    pure_ftp-upload-pipe.o pure_ftp-ipv4stack.o \

```

```

pure_ftp-altlog.o pure_ftp-crypto.o \
pure_ftp-crypto-md5.o pure_ftp-crypto-sha1.o \
pure_ftp-quotas.o pure_ftp-fakechroot.o \
pure_ftp-diraliases.o pure_ftp-ftpwho-read.o \
pure_ftp-getloadavg.o pure_ftp-privsep.o \
pure_ftp-tls.o pure_ftp-osx-extensions.o

```

CFLAGS += -D_GNU_SOURCE=1

all: pure-ftp

#下面就是我们自定义的模式规则，给出了上面这种 pure_ftp-xxx.o 的编译#规则：

```

pure_ftp-%.o : %.c
    $(CC) $(CFLAGS) -I. -I.. -c -o $@ $<

```

pure-ftp: \$(PUREFTPDOBJS)

```

    $(CC) $(CFLAGS) $(LDFLAGS) -o pure-ftp $(PUREFTPDOBJS) $(LIBS)

```

clean:

编译过程如下：

```

/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -I. -I.. -c -o
pure_ftp-daemons.o daemons.c

```

.....

```

/opt/host/armv4l/bin/armv4l-unknown-linux-gcc -g -O2 -D_GNU_SOURCE=1 -o pure-ftp
pure_ftp-daemons.o pure_ftp-ftp.o pure_ftp-log_unix.o pure_ftp-log_mysql.o
pure_ftp-log_pgsq.o pure_ftp-log_pam.o pure_ftp-log_ldap.o pure_ftp-log_puredb.o
pure_ftp-log_extauth.o pure_ftp-ls.o pure_ftp-parser.o pure_ftp-bsd-glob.o
pure_ftp-fakesnprintf.o pure_ftp-bsd-realpath.o pure_ftp-mysnprintf.o pure_ftp-caps.o
pure_ftp-ftp_parser.o pure_ftp-dynamic.o pure_ftp-ftpwho-update.o
pure_ftp-bsd-getopt_long.o pure_ftp-upload-pipe.o pure_ftp-ipv4stack.o pure_ftp-altlog.o
pure_ftp-crypto.o pure_ftp-crypto-md5.o pure_ftp-crypto-sha1.o pure_ftp-quotas.o
pure_ftp-fakechroot.o pure_ftp-diraliases.o pure_ftp-ftpwho-read.o pure_ftp-getloadavg.o
pure_ftp-privsep.o pure_ftp-tls.o pure_ftp-osx-extensions.o -lcrypt

```

一次通过！！

四. 思考题

1. 整理对 Linux 内核、LIBC 库、编译器和应用程序的关系的理解。

实验四：框架型驱动实验

一. 实验目的

通过本实验，使学生掌握 Linux 基本的驱动程序框架结构，了解编写驱动程序的基本步骤和方法。

二. 实验原理和说明

1. Linux 驱动程序原理

在 Linux 系统里，对用户程序而言，设备驱动程序隐藏了设备的具体细节，对各种不同设备提供了一致的接口，一般来说是把设备映射为一个特殊的设备文件（也有设备不作这样的映射），用户程序可以象对其它文件一样对此设备文件进行操作。Linux 对硬件设备支持两个标准接口：块特别设备文件和字符特别设备文件，通过块（字符）特别设备文件存取的设备称为块（字符）设备或具有块（字符）设备接口。块设备接口仅支持面向块的 I/O 操作，所有 I/O 操作都通过在内核地址空间中的 I/O 缓冲区进行，它可以支持几乎任意长度和任意位置上的 I/O 请求，即提供随机存取的功能。

字符设备接口支持面向字符的 I/O 操作，它不经过系统的快速缓存，所以它们负责管理自己的缓冲区结构。字符设备接口只支持顺序存取的功能，一般不能进行任意长度的 I/O 请求，而是限制 I/O 请求的长度必须是设备要求的基本块长的倍数。显然，本程序所驱动的串行卡只能提供顺序存取的功能，属于是字符设备，因此后面的讨论在两种设备有所区别时都只涉及字符型设备接口。设备由一个主设备号和一个次设备号标识。主设备号唯一标识了设备类型，即设备驱动程序类型，它是块设备表或字符设备表中设备表项的索引。次设备号仅由设备驱动程序解释，一般用于识别在若干可能的硬件设备中，I/O 请求所涉及到的那个设备。

设备驱动程序可以分为三个主要组成部分：

- (1) 自动配置和初始化子程序，负责检测所要驱动的硬件设备是否存在和是否能正常工作。如果该设备正常，则对这个设备及其相关的、设备驱动程序需要的软件状态进行初始化。这部分驱动程序仅在初始化的时候被调用一次。
- (2) 服务于 I/O 请求的子程序，又称为驱动程序的上半部分。调用这部分是由于系统调用的结果。这部分程序在执行的时候，系统仍认为是和进行调用的进程属于同一个进程，只是由用户态变成了核心态，具有进行此系统调用的用户程序的运行环境，因此可以在其中调用 `sleep()` 等与进程运行环境有关的函数。
- (3) 中断服务子程序，又称为驱动程序的下半部分。在 UNIX 系统中，并不是直接从中断向量表中调用设备驱动程序的中断服务子程序，而是由 UNIX 系统来接收硬件中断，再由系统调用中断服务子程序。中断可以产生在任何一个进程运行的时候，因此在中断服务程序被调用的时候，不能依赖于任何进程的状态，也就不能调用任何与进程运行环境有关的函数。因为设备驱动程序一般支持同一类型的若干设备，所以一般在系统调用中断服务子程序的时候，都带有一个或多个参数，以唯一标识请求服务的设备。

在系统内部，I/O 设备的存取通过一组固定的入口点来进行，这组入口点是由每个设备的设备驱动程序提供的。

一般来说，字符型设备驱动程序能够提供如下几个入口点：

- (1) `open` 入口点。打开设备准备 I/O 操作。对字符特别设备文件进行打开操作，都会调用设备的 `open` 入口点。`open` 子程序必须对将要进行的 I/O 操作做好必要的准备工作，如清除缓冲区等。如果设备是独占的，即同一时刻只能有一个程序访问此设备，则 `open` 子程序必须设置一些标志以

表示设备处于忙状态。

- (2) **close** 入口点。关闭一个设备。当最后一次使用设备终结后，调用 **close** 子程序。独占设备必须标记设备可再次使用。
- (3) **read** 入口点。从设备上读数据。对于有缓冲区的 I/O 操作，一般是从缓冲区里读数据。对字符特别设备文件进行读操作将调用 **read** 子程序。
- (4) **write** 入口点。往设备上写数据。对于有缓冲区的 I/O 操作，一般是把数据写入缓冲区里。对字符特别设备文件进行写操作将调用 **write** 子程序。
- (5) **ioctl** 入口点。执行读、写之外的操作，实现对设备的控制。
- (6) **select** 入口点。检查设备，看数据是否可读或设备是否可用于写数据。**select** 系统调用在检查与设备特别文件相关的文件描述符时使用 **select** 入口点。

如果设备驱动程序没有提供上述入口点中的某一个，系统会用缺省的子程序来代替。对于不同的系统，也还有一些其它的入口点。

1.2 LINUX 系统下的设备驱动程序

下面的 **file_operations** 结构体是定义在 **include/linux/fs.h** 中，驱动程序很大一部分工作就是要“填写”结构体中定义的函数（根据需要,实现部分函数或全部）：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
        void __user *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
            long, unsigned long, unsigned long);
    long (*fcntl)(int fd, unsigned int cmd, unsigned long arg, struct file *filp);
};
```

其中，`struct inode` 提供了关于特别设备文件`/dev/driver`（假设此设备名为 `driver`）的信息，它的定义为：

```
#include <linux/fs.h>
struct inode {
    dev_t i_dev;
    unsigned long i_ino; /* Inode number */
    umode_t i_mode; /* Mode of the file */
    nlink_t i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    dev_t i_rdev; /* Device major and minor numbers*/
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
    time_t i_ctime;
    unsigned long i_blksize;
    unsigned long i_blocks;
    struct inode_operations * i_op;
    struct super_block * i_sb;
    struct wait_queue * i_wait;
    struct file_lock * i_flock;
    struct vm_area_struct * i_mmap;
    struct inode * i_next, * i_prev;
    struct inode * i_hash_next, * i_hash_prev;
    struct inode * i_bound_to, * i_bound_by;
    unsigned short i_count;
    unsigned short i_flags; /* Mount flags (see fs.h) */
    unsigned char i_lock;
    unsigned char i_dirt;
    unsigned char i_pipe;
    unsigned char i_mount;
    unsigned char i_seek;
    unsigned char i_update;
    union {
        struct pipe_inode_info pipe_i;
        struct minix_inode_info minix_i;
        struct ext_inode_info ext_i;
        struct msdos_inode_info msdos_i;
        struct iso_inode_info isofs_i;
        struct nfs_inode_info nfs_i;
    } u;
};
```

`struct file` 主要用于与文件系统对应的设备驱动程序使用。当然，其它设备驱动程序也可以使用它。它提供关于被打开的文件的信息，定义为：

```
#include <linux/fs.h>
struct file {
    mode_t f_mode;
    dev_t f_rdev; /* needed for /dev/tty */
    off_t f_pos; /* Curr. posn in file */
    unsigned short f_flags; /* The flags arg passed to open */
    unsigned short f_count; /* Number of opens on this file */
    unsigned short f_reada;
    struct inode *f_inode; /* pointer to the inode struct */
    struct file_operations *f_op; /* pointer to the fops struct */
};
```

在结构 `file_operations` 里，指出了设备驱动程序所提供的入口点位置，分别是：

- (1) `lseek`，移动文件指针的位置，显然只能用于可以随机存取的设备。
- (2) `read`，进行读操作，参数 `buf` 为存放读取结果的缓冲区，`count` 为所要读取的数据长度。返回值为负表示读取操作发生错误，否则返回实际读取的字节数。对于字符型，要求读取的字节数和返回的实际读取字节数都必须是 `inode->i_blksize` 的的倍数。
- (3) `write`，进行写操作，与 `read` 类似。
- (4) `readdir`，取得下一个目录入口点，只有与文件系统相关的设备驱动程序才使用。
- (5) `select`，进行选择操作，如果驱动程序没有提供 `select` 入口，`select` 操作将会认为设备已经准备好进行任何的 I/O 操作。
- (6) `ioctl`，进行读、写以外的其它操作，参数 `cmd` 为自定义的命令。
- (7) `mmap`，用于把设备的内容映射到地址空间，一般只有块设备驱动程序使用。
- (8) `open`，打开设备准备进行 I/O 操作。返回 0 表示打开成功，返回负数表示失败。如果驱动程序没有提供 `open` 入口，则只要 `/dev/driver` 文件存在就认为打开成功。
- (9) `release`，即 `close` 操作。

设备驱动程序所提供的入口点，在设备驱动程序初始化的时候向系统进行登记，以便系统在适当的时候调用。Linux 系统里，通过调用 `register_chrdev` 向系统注册字符型设备驱动程序。`register_chrdev` 定义为：

```
#include <linux/fs.h>
#include <linux/errno.h>
int register_chrdev(unsigned int major, /*主设备号*/
                    const char name, /*设备名*/
                    struct file_operations fops); /*文件系统调用入口点*/
```

其中，`major` 是为设备驱动程序向系统申请的主设备号，如果为 0 则系统为此驱动程序动态地分配一个主设备号。`name` 是设备名。`fops` 就是前面所说的对各个调用的入口点的说明。此函数返回 0 表示成功。返回 `-EINVAL` 表示申请的主设备号非法，一般来说是主设备号大于系统所允许的最大设备号。返回 `-EBUSY` 表示所申请的主设备号正在被其它设备驱动程序使用。如果是动态分配主设备号成功，此函数将返回所分配的主设备号。如果 `register_chrdev` 操作成功，设备名就会出现在 `/proc/devices` 文件里。

初始化部分一般还负责给设备驱动程序申请系统资源，包括内存、中断、时钟、I/O 端口等，这些资源也可以在 `open` 子程序或别的地方申请。在这些资源不用的时候，应该释放它们，以利于资源的共享。在 UNIX 系统里，对中断的处理是属于系统核心的部分，因此如果设备与系统之间以中断方式进行数据交换的话，就必须把该设备的驱动程序作为系统核心的一部分。设备驱动程序通过调用 `request_irq` 函数来申请中断，通过 `free_irq` 来释放中断。它们的定义为：

```
#include <linux/sched.h>
int request_irq(unsigned int irq,
                void (*handler)(int irq,void dev_id,struct pt_regs *regs),
                unsigned long flags,
                const char *device,
                void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

参数说明：

参数 `irq` 表示所要申请的硬件中断号。`handler` 为向系统登记的中断处理子程序，中断产生时由系统来调用，调用时所带参数 `irq` 为中断号，`dev_id` 为申请时告诉系统的设备标识，`regs` 为中断发生时寄存器内容。`device` 为设备名，将会出现在 `/proc/interrupts` 文件里。`flag` 是申请时的选项，它决定中断处理程序的一些特性，其中最重要的是中断处理程序是快速处理程序（`flag` 里设置了 `SA_INTERRUPT`）还是慢速处理程序（不设置 `SA_INTERRUPT`），快速处理程序运行时，所有中断都被屏蔽，而慢速处理程序运行时，除了正在处理的中断外，其它中断都没有被屏蔽。在 LINUX 系统中，中断可以被不同的中断处理程序共享，这要求每一个共享此中断的处理程序在申请中断时在 `flags` 里设置 `SA_SHIRQ`，这些处理程序之间以 `dev_id` 来区分。如果中断由某个处理程序独占，则 `dev_id` 可以为 `NULL`。`request_irq` 返回 0 表示成功，返回 `-EINVAL` 表示 `irq>15` 或 `handler==NULL`，返回 `-EBUSY` 表示中断已经被占用且不能共享。作为系统核心的一部分，设备驱动程序在申请和释放内存时不是调用 `malloc` 和 `free`，而代之以调用 `kmalloc` 和 `kfree`，它们被定义为：

```
#include <linux/kernel.h>
void * kmalloc(unsigned int len, int priority);
void kfree(void * obj);
```

参数 `len` 为希望申请的字节数，`obj` 为要释放的内存指针。`priority` 为分配内存操作的优先级，即在没有足够空闲内存时如何操作，一般用 `GFP_KERNEL`。

与中断和内存不同，使用一个没有申请的 I/O 端口不会使 CPU 产生异常，也就不会导致诸如“segmentation fault”一类的错误发生。任何进程都可以访问任何一个 I/O 端口。此时系统无法保证对 I/O 端口的操作不会发生冲突，甚至会因此而使系统崩溃。因此，在使用 I/O 端口前，也应该检查此 I/O 端口是否已有别的程序在使用，若没有，再把此端口标记为正在使用，在使用完以后释放它。这样需要用到如下几个函数：

```
int check_region(unsigned int from, unsigned int extent);
void request_region(unsigned int from, unsigned int extent,
                    const char *name);
void release_region(unsigned int from, unsigned int extent);

int check_mem_region(unsigned int from, unsigned int extent);
void request_mem_region(unsigned int from, unsigned int extent,
                        const char *name);
void release_mem_region(unsigned int from, unsigned int extent);
```

调用这些函数时的参数为：`from` 表示所申请的 I/O 端口的起始地址；`extent` 为所要申请的从 `from`

开始的端口数；**name** 为设备名，将会出现在 `/proc/ioports` 文件里。`check_region` 返回 0 表示 I/O 端口空闲，否则为正在被使用。

在申请了 I/O 端口之后，就可以如下几个函数来访问 I/O 端口：

```
#include <asm/io.h>
inline unsigned int inb(unsigned short port);
inline unsigned int inb_p(unsigned short port);
inline void outb(char value, unsigned short port);
inline void outb_p(char value, unsigned short port);
```

上面的函数用于访问 8 位端口。**Port** 参数在一些平台上被定义为 `unsigned long`，而在另一些平台上被定义为 `unsigned short`。不同平台上 `inb` 返回值类型也不同。

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

这些函数用于访问 16 位端口（字宽度）。M68K 或 S390 平台上不提供这些函数，因为这些平台只支持字节宽度的 I/O 操作。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

这些函数用于访问 32 位端口。根据平台的不同，`longword` 参数被定义为 `unsigned long` 类型或 `unsigned int` 类型。和字宽度 I/O 一样，“长字” I/O 在 M68k 和 S390 平台上也不能用。

其中 `inb_p` 和 `outb_p` 插入了一定的延时以适应某些慢的 I/O 端口。在设备驱动程序里，一般都需要用到计时机制。在 LINUX 系统中，时钟是由系统接管，设备驱动程序可以向系统申请时钟。与时钟有关的系统调用有：

```
#include <asm/param.h>
#include <linux/timer.h>
void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
inline void init_timer(struct timer_list * timer);
```

`struct timer_list` 的定义为：

```
struct timer_list {
    struct timer_list *next;
    struct timer_list *prev;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long d);
};
```

其中 `expires` 是要执行 `function` 的时间。系统核心有一个全局变量 `JIFFIES` 表示当前时间，一般在调用 `add_timer` 时 `jiffies=JIFFIES+num`，表示在 `num` 个系统最小时间间隔后执行 `function`。系统最小时间间隔与所用的硬件平台有关，在核心里定义了常数 `HZ` 表示一秒内最小时间间隔的数目，则 `num*HZ` 表示 `num` 秒。系统计时到预定时间就调用 `function`，并把此子程序从定时队列里删除，因此如果想要每隔一定时间间隔执行一次的话，就必须在 `function` 里再一次调用 `add_timer`。`function` 的参数 `d` 即为 `timer` 里面的 `data` 项。

在设备驱动程序里，还可能会用到如下的一些系统函数：

```
#include <asm/system.h>
```

```
#define cli() __asm__ __volatile__ ("cli::")
```

```
#define sti() __asm__ __volatile__ ("sti::")
```

这两个函数负责打开和关闭中断允许。

```
#include <asm/segment.h>
```

```
void memcpy_fromfs(void * to,const void * from,unsigned long n);
```

```
void memcpy_tofs(void * to,const void * from,unsigned long n);
```

在用户程序调用 `read`、`write` 时，因为进程的运行状态由用户态变为核心态，地址空间也变为核心地址空间。而 `read`、`write` 中参数 `buf` 是指向用户程序的私有地址空间的，所以不能直接访问，必须通过上述两个系统函数来访问用户程序的私有地址空间。`memcpy_fromfs` 由用户程序地址空间往核心地址空间复制，`memcpy_tofs` 则反之。参数 `to` 为复制的目的指针，`from` 为源指针，`n` 为要复制的字节数。

在设备驱动程序里，可以调用 `printk` 来打印一些调试信息，用法与 `printf` 类似。`printk` 打印的信息不仅出现在屏幕上，同时还记录在文件 `syslog` 里。

1.3 LINUX 系统下的具体实现

在 LINUX 里，除了直接修改系统核心的源代码，把设备驱动程序加进核心里以外，还可以把设备驱动程序作为可加载的模块，由系统管理员动态地加载它，使之成为核心地一部分。也可以由系统管理员把已加载地模块动态地卸载下来。LINUX 中，模块可以用 C 语言编写，用 `gcc` 编译成目标文件（不进行链接，作为*.o 文件存在），为此需要在 `gcc` 命令行里加上 `-c` 的参数。在编译时，还应该在 `gcc` 的命令行里加上这样的参数：`-D__KERNEL__ -DMODULE`。由于在不链接时，`gcc` 只允许一个输入文件，因此一个模块的所有部分都必须在一个文件里实现。编译好的模块*.o 放在 `/lib/modules/xxxx/misc` 下（xxxx 表示核心版本，如在核心版本为 2.0.30 时应该为 `/lib/modules/2.0.30/misc`），然后用 `depmod -a` 使此模块成为可加载模块。模块用 `insmod` 命令加载，用 `rmmod` 命令来卸载，并可以用 `lsmod` 命令来查看所有已加载的模块的状态。

编写模块程序的时候，必须提供两个函数，一个是 `int init_module(void)`，供 `insmod` 在加载此模块的时候自动调用，负责进行设备驱动程序的初始化工作。`init_module` 返回 0 以表示初始化成功，返回负数表示失败。另一个函数是 `void cleanup_module (void)`，在模块被卸载时调用，负责进行设备驱动程序的清除工作。

在成功的向系统注册了设备驱动程序后（调用 `register_chrdev` 成功后），就可以用 `mknod` 命令来把设备映射为一个特别文件，其它程序使用这个设备的时候，只要对此特别文件进行操作就行了。

内核升级到 2.4 后，系统提供了两个新的函数：`devfs_register`，`devfs_unregister`。

三．实验内容和步骤

本试验通过操作内核中的一个数组元素，来介绍驱动的整体结构。

1. 编写框架驱动程序：

本试验的目标是要编写一个完整的虚拟字符设备驱动程序，之所以称之为虚拟驱动程序是因为它并不于任何的硬件设备打交道。我们将通过开发这个字符设备驱动程序来演示驱动编程的方法，步骤。

我们编写的框架驱动程序：`skeleton`，就是框架的意思，`skeleton` 是一个操作数组的简单字符设备驱动，我们把这个数组当作一个设备。这样处理的好处是可以避免与复杂的硬件设备打交道，对于初学者，很容易的就能够通过 `skeleton` 这个字符设备驱动掌握在 Linux 下编写驱动的一般方法，在后面的各个章节中，大家将要操作大量的外部设备，但不管多么复杂的驱动程序，它的框架和我们的 `skeleton` 也都是大同小异的。

在这里，我要提醒大家，驱动程序的代码量通常都不大，即使一个复杂的 `usb` 驱动程序它的代

码量也不过 2, 3 千行, 当你掌握了驱动程序的框架和驱动程序编程的一般方法后, 编写驱动的主要工作量就是熟悉硬件设备的工作机制, 协议等。

Skeleton 向大家展示了内核和字符设备驱动程序之间的接口, 并让用户运行某些测试程序, 通过编写 skeleton 字符设备驱动和这些测试程序, 大家应该能够清楚, 在应用程序中如何使用驱动程序提供的接口以及在驱动程序中如何建立这样的接口。

skeleton 作为驱动程序有着众多缺点:

(1) Skeleton 不支持代码重入。

(2) Skeleton 不支持多设备。

尽管 skeleton 有这些缺点, 但作为一个学习驱动程序的入门例子, 它已经能够胜任了。

Skeleton 的设计:

Skeleton 要完成的任务很简单, skeleton.c 中定义了一个全局数组 `char skeleton_drvinfo[100]`; 我们在整个框架驱动程序中操作的都是 `skeleton_drvinfo`, 也就是说我们通过 `open`, `read`, `write`, `ioctl` 这些接口函数操作 `skeleton_drvinfo` 结构。

2. 编写驱动测试程序:

LINUX 下对硬件的控制是通过对 `/dev` 目录下设备文件节点的操作, 驱动程序通常会在 `/dev` 目录下建立设备文件节点, 应用程序可以通过驱动提供的接口函数来操作设备文件节点, 从而控制设备。

```
#include <stdio.h>
```

```
#include <fcntl.h>
```

```
#include <stdlib.h>
```

```
char * device = "/dev/skeleton";
```

```
int skeleton_fd;
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    int i, count;
```

```
    char cBuffer[100];
```

```
    skeleton_fd = open(device, O_RDONLY);
```

```
    if (skeleton_fd == -1) {
```

```
        printf("Unable to open skeleton device");
```

```
        exit(0);
```

```
    }
```

```
    count = read(skeleton_fd, cBuffer, 100);
```

```
    close(skeleton_fd);
```

```
}
```

上面这段代码可以做为应用程序调用驱动程序接口函数的参考, 首先通过 `open` 函数打开 `/dev/` 目录下相应的文件节点, `open()` 调用在正常情况下返回一个文件描述符, 在发生错误时返回 -1 并把 `errno` 置为相应的错误代码。Open() 函数的第二个参数是代开设备的附加标志, 可参见下表:

标志	说明
O_RDONLY	只读访问打开文件
O_WRONLY	只写访问打开文件
O_RDWR	读和写访问打开文件

O_CREATE	如果文件不存在则建立文件
O_EXCL	如果文件存在，则强制 open（）失败
O_NOCTTY	在打开 tty 时如果进程没有控制 tty 则不控制终端
O_TRUNC	如果文件存在，则将文件的长度截至 0
O_APPEND	把向文件添加内容的指针蛇设到文件的结束处
O_NONBLOCK	如果其操作不能无延迟地完成，则在操作完成之前返回
O_NODELAY	同 O_NONBLOCK
O_SYNC	在数据被物理地写入磁盘或其他设备之后操作才返回

应用程序中 open 后就可以通过驱动提供的其他接口函数操作设备，在对设备控制完毕后，千万不要忘了通过 close 调用关闭设备。

四．思考题

1. 请思考字符设备和块设备本质区别是什么！
2. 完成 ioctl 接口，试着对 skeleton 设备进行某些控制
3. 对 Linux 下驱动程序开发有兴趣的读者，请仔细阅读《LINUX DEVICE DRIVERS》，这是一本指导开发 Linux 下设备驱动的必读书籍。

实验五：串口通信实验

一. 实验目的

了解 linux 下串行程序设计方法，了解 RS_232 和 RS_485 标准。掌握终端的主要属性和设置方法，熟悉终端 IO 函数的使用。掌握 CPU 利用串口进行通讯的方法。了解特殊字符 0x0d0a 的处理方法。

二. 实验原理和说明

1. 基本原理

串行端口的本质功能是作为 CPU 和串行设备间的编码转换器。当数据从 CPU 经过串行端口发送出去时，字节数据转换为串行的位。在接收数据时，串行的位被转换为字节数据。串口是系统资源的一部分，应用程序要使用串口进行通信，必须在使用之前向操作系统提出资源申请要求（打开串口），通信完成后必须释放资源（关闭串口）。

2. 串口通信的基本任务

- (1) 实现数据格式化：因为来自 CPU 的是普通的并行数据，所以，接口电路应具有实现不同串行通信方式下的数据格式化的任务。在异步通信方式下，接口自动生成起止式的帧数据格式。在面向字符的同步方式下，接口要在待传送的数据块前加上同步字符。
- (2) 进行串—并转换：串行传送，数据是一位一位串行传送的，而计算机处理数据是并行数据。所以当数据由计算机送至数据发送器时，首先把串行数据转换为并行数才能送入计算机处理。因此串并转换是串行接口电路的重要任务。
- (3) 控制数据传输速率：串行通信接口电路应具有对数据传输速率——波特率进行选择和控制的能力。
- (4) 进行错误检测：在发送时接口电路对传送的字符数据自动生成奇偶校验位或其他校验码。在接收时，接口电路检查字符的奇偶校验或其他校验码，确定是否发生传送错误。
- (5) 进行 TTL 与 EIA 电平转换：CPU 和终端均采用 TTL 电平及正逻辑，它们与 EIA 采用的电平及负逻辑不兼容，需在接口电路中进行转换。

串行通信可以分为两种类型：同步通信、异步通信。微机中大量使用异步通信，下面就详细介绍异步串行 I/O。

3. 异步串行 I/O 原理

异步串行方式是将传输数据的每个字符一位接一位(例如先低位、后高位)地传送。数据的各不同位可以分时使用同一传输通道，因此串行 I/O 可以减少信号连线，最少用一对线即可进行。接收方对于同一根线上一连串的数字信号，首先要分割成位，再按位组成字符。为了恢复发送的信息，双方必须协调工作。在微型计算机中大量使用异步串行 I/O 方式，双方使用各自的时钟信号，而且允许时钟频率有一定误差，因此实现较容易。但是由于每个字符都要独立确定起始和结束(即每个字符都要重新同步)，字符和字符间还可能有长度不定的空闲时间，因此效率较低。



上图给出异步串行通信中一个字符的传送格式。开始前，线路处于空闲状态，送出连续“1”。传送开始时首先发一个“0”作为起始位，然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可以约定为5位、6位、7位或8位，一般采用ASCII编码。后面是奇偶校验位，根据约定，用奇偶校验位将所传字符中为“1”的位数凑成奇数个或偶数个。也可以约定不要奇偶校验，这样就取消奇偶校验位。最后是表示停止位的“1”信号，这个停止位可以约定持续1位、1.5位或2位的时间宽度。至此一个字符传送完毕，线路又进入空闲，持续为“1”。经过一段随机的时间后，下一个字符开始传送才又发出起始位。每一个数据位的宽度等于传送波特率的倒数。微机异步串行通信中，常用的波特率为300, 600, 1200, 2400, 4800, 9600, 19200, 57600, 115200等。

接收方按约定的格式接收数据，并进行检查，可以查出以下三种错误：

- (1) 奇偶错：在约定奇偶检查的情况下，接收到的字符奇偶状态和约定不符。
- (2) 帧格式错：一个字符从起始位到停止位的总位数不对。
- (3) 溢出错：若先接收的字符尚未被微机读取，后面的字符又传送过来，则产生溢出错。

每一种错误都会给出相应的出错信息，提示用户处理。

4. 串口终端函数

4.1 串口编程要用到的头文件

```
#include <stdio.h>      /*标准输入输出定义*/
#include <unistd.h>      /*linux标准函数定义*/
#include <assert.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>       /*文件控制定义*/
#include <termios.h>     /*POSIX 终端控制定义*/
```

4.2 打开串口

在Linux下串口文件是位于/dev下，串口一为/dev/ttyS0，串口二为/dev/ttyS1，打开串口是通过使用标准的文件打开函数操作：

```
int fd;
fd = open("/dev/ttyS0", O_RDWR);
if (fd == -1 )
{
    perror("提示错误！");
}
```

4.3 设置串口

最基本的设置串口包括波特率设置，校验位和停止位设置。串口的设置主要是设置 `struct termios` 结构体的各成员值。

```
struct termios
{
    unsigned short c_iflag;    /* 输入模式标志*/
    unsigned short c_oflag;    /* 输出模式标志*/
    unsigned short c_cflag;    /* 控制模式标志*/
    unsigned short c_lflag;    /* 本地模式标志 */
    unsigned char c_line;      /* 线性规程*/
    unsigned char c_cc[NCC];   /* 控制字符 */
}
```

设置这个结构体很复杂，可以参考 `man` 手册。这里就只考虑常见的一些设置：

(1) 波特率设置

```
struct termios Opt;
tcgetattr(fd, &Opt);
cfsetispeed(&Opt,B19200); /*设置为 19200Bps*/
cfsetospeed(&Opt,B19200);
tcsetattr(fd,TCANOW,&Opt);
```

(2) 校验位设置

(a) 无校验 8 位：

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS8;
```

(b) 奇效验(Odd) 7 位：

```
Option.c_cflag |= ~PARENB;
Option.c_cflag &= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
```

(c) 偶效验(Even) 7 位：

```
Option.c_cflag &= ~PARENB;
Option.c_cflag |= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
```

(d) Space 校验 7 位：

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= &~CSIZE;
Option.c_cflag |= CS8;
```

(3) 停止位设置

(a) 1 位:

```
options.c_cflag &= ~CSTOPB;
```

(b) 2 位:

```
options.c_cflag |= CSTOPB;
```

需要注意的是,如果不是开发终端之类的,只是串口传输数据,而不需要串口来处理,那么使用原始模式(Raw Mode)方式来通讯,设置方式如下:

```
options.c_iflag &= ~(ICANON | ECHO | ECHOE | ISIG); /*Input*/
```

```
options.c_oflag &= ~OPOST; /*Output*/
```

试验示例代码中是在函数 OpenComConfig 里设置的:

```
int OpenComConfig(int port, const char deviceName[], long baudRate, int parity, int dataBits,
                  int stopBits, int iqSize, int oqSize)
```

```
{
    .....
    newtio.c_cflag = CS8 | CLOCAL | CREAD ;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = ~OPOST;
    newtio.c_iflag = ~(ICANON | ECHO | ECHOE | ISIG);
    newtio.c_cc[VINTR]    = 0;
    newtio.c_cc[VQUIT]    = 0;
    newtio.c_cc[VERASE]   = 0;
    newtio.c_cc[VKILL]    = 0;
    newtio.c_cc[VEOF]     = 4;
    newtio.c_cc[VTIME]    = 0;
    newtio.c_cc[VMIN]     = 1;
    newtio.c_cc[VSWTC]    = 0;
    newtio.c_cc[VSTART]   = 0;
    newtio.c_cc[VSTOP]    = 0;
    newtio.c_cc[VSUSP]    = 0;
    newtio.c_cc[VEOL]     = 0;
    newtio.c_cc[VREPRINT] = 0;
    newtio.c_cc[VDISCARD] = 0;
    newtio.c_cc[VWERASE]  = 0;
    newtio.c_cc[VLNEXT]   = 0;
    newtio.c_cc[VEOL2]    = 0;
    cfsetospeed(&newtio, GetBaudRate(baudRate));
    cfsetispeed(&newtio, GetBaudRate(baudRate));
    tcsetattr(ports[port].handle, TCSANOW, &newtio);
    .....
}
```

4.4 读写串口

设置好串口之后,读写串口就很容易了,把串口当作文件读写就可以了。

(1) 发送数据:

```
char buffer[1024];
int Length=1024;
int nByte;
nByte = write(fd, buffer ,Length);
```

(2) 读取串口数据:

使用文件操作 `read` 函数读取, 如果设置为原始模式(Raw Mode)传输数据, 那么 `read` 函数返回的字符数是实际串口收到的字符数。

```
char buff[1024];
int Len=1024;
int readByte = read(fd, buff, Len);
```

也可以使用操作文件的函数来实现异步读取, 如 `fcntl`, 或者 `select` 等来操作。

```
fd_set rfd;
struct timeval tv;
int retval;
```

/*下面几行设置要监视进行读写操作的文件集*/

```
FD_ZERO(&rfd); //文件集清零
FD_SET(ports[portNo].handle, &rfd); //向集合中添加一个文件句柄
tv.tv_sec = Timeout/1000; //设置等待的时间
tv.tv_usec = (Timeout%1000)*1000;
```

```
retval = select(16, &rfd, NULL, NULL, &tv); //等待所监视的文件集准备好。
if (retval) //文件集中有文件在等待时间内准备好了。
{
    actualRead = read(ports[portNo].handle, buf, maxCnt); //读取数据
}
```

4.5 关闭串口

关闭串口就是关闭文件。

```
close(fd);
```

5. 串行接口的物理层标准

通用的串行 I/O 接口有许多种, 现仅就最常见的两种标准作简单介绍。

5.1 EIA RS—232C

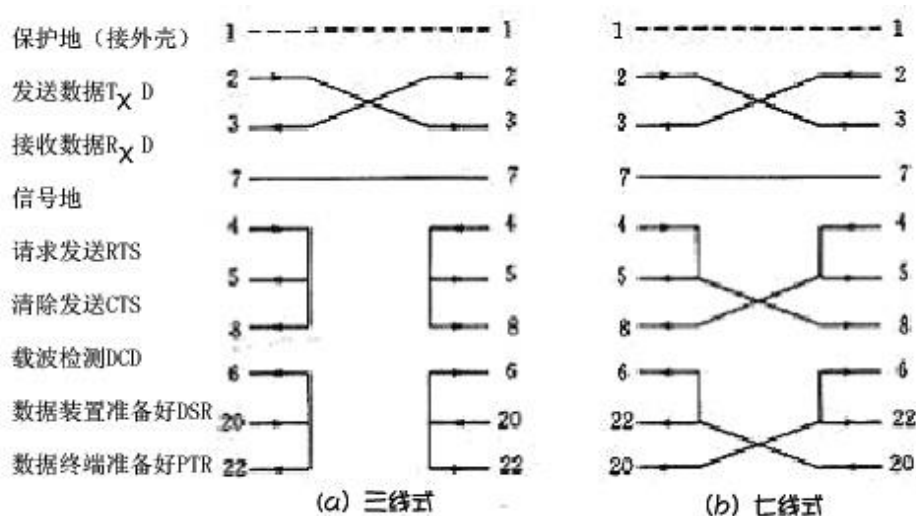
这是美国电子工业协会推荐的一种标准(Electronic industries Association Recoil-mended Standard)。它在一种 25 针接插件(DB—25)上定义了串行通信的有关信号。这个标准后来被世界各国所接受并使用到计算机的 I/O 接口中。

5.1.1 信号连线

在实际异步串行通信中, 并不要求用全部的 RS—232C 信号, 许多 PC/XT 兼容机仅用 15 针接插件(DB—15)来引出其异步串行 I/O 信号, 而 PC 中更是大量采用 9 针接插件(DB—9)来担当此任, 因此这里也不打算就 RS—232C 的全部信号作详细解释。图 3-2 给出两台微机利用 RS—232C 接口通信的连线(无 MODEM), 我们按 DB—25 的引脚号标注各个信号。

下面对下图中几个主要信号作简要说明。

- (1) 保护地：通信线两端所接设备的金属外壳通过此线相联。当通信电缆使用屏蔽线时，常利用其外皮金属屏蔽网来实现。由于各设备往往已通过电源线接通保护地，因此，通信线中不必重复接此地线(图中用虚线表示)。例如使用 9 针插头(DB-9)的异步串行 I/O 接口就没有引出保护地信号。
- (2) TXD/RXD：是一对数据线，TXD 称发送数据输出，RXD 称接收数据输入。当两台微机以全双工方式直接通信(无 MODEM 方式)时，双方的这两根线应交叉联接(扭接)。
- (3) 信号地：所有的信号都要通过信号地线构成耦合回路。通信线有以上三条(TXD、RXD 和信号地)就能工作了。其余信号主要用于双方设备通信过程中的联络(握手信号)，而且有些信号仅用于和 MODEM 的联络。若采取微型机对微型机直接通信，且双方可直接对异步串行通信电路芯片编程，若设置成不要任何联络信号，则其它线都可可不接。有时在通信线的同一端将相关信号短接以“自握手”方式满足联络要求。这就是下图所示的情况。



- (4) RTS/CTS：请求发送值号 RTS 是发送器输出的准备好信号。接收方准备好后送回清除发送信号 CTS 后，发送数据开始进行，在同一端将这两个信号短接就意味着只要发送器准备好即可发送。
- (5) DCD：载波检测(又称接收线路信号检测)。本意是 MODEM 检测到线路中的载波信号后，通知终端准备接收数据的信号，在没有接 MODEM 的情况下，也可以和 RTS、CTS 短接。相对于 MODEM 而言，微型机和终端机一样被称为数据终端 DTE(Data Terminal Equipment)而 MODEM 被称为数据通信装置 DCE(Data Communications Equipment)，DTE 和 DCE 之间的连接不能像上图中有“扭接”现象，而应该是按接插件芯号，同名端对应相接。此处介绍的 RS-232C 的信号名称及信号流向都是对 DTE 而言的。
- (6) DTR/DSR：数据终端准备好时发 DTR 信号，在收到数据通信装置准备好 DSR 信号后，方可通信。上图 a 中将这一对信号以“自握手”方式短接。
- (7) R1：原意是在 MODEM 接收到电话交换机有效的拨号时，使 RI 有效，通知数据终端准备传送。在无 MODEM 时也可和 DTR 相接。

上图 b 给出了无 MODEM 情况下，DTE 对 DTE 异步串行通信线路的完整连接，它不仅适用于微型机和微型机之间的通信，还适用于微型机和异步串行外部设备(如终端机、绘图仪、数字化仪等)的连接。

5.1.2 接口的电气特性

在 RS-232-C 中任何一条信号线的电压均为负逻辑关系。即：逻辑“1”，-5~-15V；逻辑“0”+5~+15V。噪声容限为 2V。即要求接收器能识别低至+3V 的信号作为逻辑“0”，高到-3V 的信号作为逻辑“1”。

5.1.3 接口的物理结构

RS-232-C 接口连接器一般使用型号为 DB-25 的 25 芯插头座,通常插头在 DCE 端,插座在 DTE 端. 一些设备与 PC 机连接的 RS-232-C 接口,因为不使用对方的传送控制信号,只需三条接口线,即“发送数据”、“接收数据”和“信号地”。所以采用 DB-9 的 9 芯插头座,传输线采用屏蔽双绞线。

5.2 RS—485C

由于 RS-232-C 接口标准出现较早,难免有不足之处,主要有以下四点:

- (1) 接口的信号电平值较高,易损坏接口电路的芯片,又因为与 TTL 电平不兼容故需使用电平转换电路方能与 TTL 电路连接。
- (2) 传输速率较低,在异步传输时,波特率为 20Kbps。
- (3) 接口使用一根信号线和一根信号返回线而构成共地的传输形式,这种共地传输容易产生共模干扰,所以抗噪声干扰性弱。
- (4) 传输距离有限,最大传输距离标准值为 50 英尺,实际上也只能用在 50 米左右

针对 RS-232-C 的不足,于是就不断出现了一些新的接口标准,RS-485 就是其中之一,它具有以下特点:

- (1) RS-485 的电气特性:逻辑“1”以两线间的电压差为+ (2~6) V 表示;逻辑“0”以两线间的电压差为- (2~6) V 表示。接口信号电平比 RS-232-C 降低了,就不易损坏接口电路的芯片,且该电平与 TTL 电平兼容,可方便与 TTL 电路连接。
- (2) RS-485 的数据最高传输速率为 10Mbps。
- (3) RS-485 接口是采用平衡驱动器和差分接收器的组合,抗共模干扰力增强,即抗噪声干扰性好。
- (4) RS-485 接口的最大传输距离标准值为 4000 英尺,实际上可达 3000 米,另外 RS-232-C 接口在总线上只允许连接 1 个收发器,即单站能力。而 RS-485 接口在总线上是允许连接多达 128 个收发器。即具有多站能力,这样用户可以利用单一的 RS-485 接口方便地建立起设备网络。

Rs485 驱动器使用平衡差分信号。由驱动器产生的电压显示在一对号线上,仅仅传输一个信号。平衡线性驱动器将通过它的 A 和 B 输出线产生一个 2~6V 的电压,同时也连接一个信号线。尽管信号地的连接重要,但是它并不用来决定数据线的逻辑状态。

线路的两个信号状态定义如下:

- (1) 当驱动器的 A 端相对于 B 端为负时,线路处于状态 1 (MARK 或者 OFF)
- (2) 当驱动器的 A 端相对于 B 端为正时,线路处于状态 0 (SPACE 或者 ON)

一般 A, B 端分别对应于很多装置的一, + 端。

Rs485 驱动器必须有一个“Enable”的输入控制信号,用于连接到输出端 A 和 B。如果该信号处于 OFF,那么驱动器从线路断开,一般认为是驱动器除了 1, 0 信号状态外的第三状态。有两种方法使得驱动器处于三态:一种是使用控制线,一般是 RTS 握手线。RTS 线连接驱动器的使能端,使得当 RTS 设置成高(逻辑 1)时,有效 RS485 驱动器;设置 RTS 为低时,使驱动器处于三态,这时候实际上从总线上断开了驱动器,从而允许其他节点可以使用同一传输线。当使用 RTS 时,必须确保发送数据前将 RTS 设置成高,在发送完数据的最后一位后,将 RTS 线设成低。。另一种可选方法是自动发送数据控制。这种方法要求特殊的电路,当数据传输时自动使能或无效驱动器。它减少了软件开销和程序员的潜在错误。

驱动器和接收器可以接受的正常模式电压为-7 ~ +12V。注意的是,传输线在线的两端有终端,但是在线中间的交点处并没有终端。应该只在高数据传输率和传输线很长时才用终端。推荐 RS485 中用信号地,保证接收器必须接收的电压保持在-7 ~ +12V。

RS485 标准允许工作在一对驱动/接收线或者多节点共享模式，最多可达 32 对。因为 RS485 接口组成的半双工网络，一般只需二根连线，所以 RS485 接口均采用屏蔽双绞线传输。RS485 接口连接器采用 DB-9 的 9 芯插头座，与智能终端 RS485 接口采用 DB-9（孔），与键盘连接的键盘接口 RS485 采用 DB-9（针）。

RS485 网络也可以连接成四线模式。注意的是，四线连接中，使用四个数据线和一個附加的信号线。同时必须有一个主节点，其他全部是从节点。主节点可以和所有从节点通信，而从节点只可以与主节点通信。从节点没有收到主节点的请求时不会传输数据，也不和其他从节点通信。每个从节点必须有一个单独的地址，以独立于其他节点。

三. 实验内容和步骤

实现开发板和 PC 机通过 minicom 通信。按下 PC 键盘通过 minicom 发送给串口，开发板监视串口，将串口中接收到的数据再发送给串口，显示在 minicom 上。

1. 编写 rs232.c，函数包括打开串口 OpenCom，发送数据 ComWrt，接收数据 ComRd。特别注意 tcsetattr 函数是如何设置终端属性的。
2. 在主函数 ttytest.c 中实现将从串口 0 接收到的字符再发送给串口 0。
3. 编译并在板子上运行。并在 minicom 上查看结果。
4. 修改主函数中 OpenCom 调用参数的波特率，重复上述步骤，实现不同波特率通信。
5. 对特殊字符 0x0d0a 的处理问题。在宿主机上编写并运行测试程序 ttytest_char，仔细观察在发送数据中如果遇到 0x0A, 0x0D, 接收数据端会发生什么情况？如下修改 c_lflag 和 c_oflag:

```
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
```

```
options.c_oflag &= ~OPOST;
```

即去掉这两行注释，修改成

```
options.c_lflag = 0;
```

```
options.c_oflag = 0;
```

再次运行该程序，还会发生上述情况吗？

四. 思考题

1. 用 RS485 标准编程实现 PC 和板子的串口通讯，和 RS232 标准有什么不同？需要修改应用程序吗？
2. 用两台运行 minicom 的 PC，或者一台运行两个 minicom 的 PC，实现串口 0 和串口 1 同时收发消息。

实验六：中断实验

一．实验目的

了解中断的原理，加深对中断向量表的理解。理解系统是如何响应外部中断的。学习设置中断，注册中断以及清除中断等函数的使用。

二．实验原理和说明

1. 中断概念

1.1 基本定义

计算机系统的“中断”是指中央处理器 CPU 正在处理某件事情的时候，发生了异常事件(如定时溢出等)，产生一个中断请求信号，请求 CPU 迅速去处理。CPU 暂时中断当前的工作，转入处理所发生的事件，处理完以后，再回到原来被中断的地方继续原来的工作，这样的过程称为中断，实现这种功能的部件称为中断系统，产生中断的部件或设备称为中断源。

一个计算机系统一般有多个中断请求源。当多个中断源同时向 CPU 请求中断时，就存在 CPU 优先响应哪一个中断请求源的问题。一般根据中断源(所发生的事件)的轻重缓急，规定中断源的优先级，CPU 优先响应中断优先级高的中断源请求。

当 CPU 正在处理一个中断请求时，又发生了另外的中断请求，如果 CPU 能暂时中止对原中断的处理，转去处理优先级更高的中断请求，待处理完以后，再继续处理原来的中断事件，这样的过程称为中断嵌套。这样的中断系统称为多级中断系统。而没有中断嵌套功能的系统称为单级中断系统。

1.2 中断向量

每个中断都可以用一个无符号整数来标识，称之为“中断向量 (Interrupt Vector)”。所有的 ARM 系统都有一张中断向量表，当出现中断需要处理时，必须调用向量表。向量表一般要位于 0 地址处。

地址	异常
0x0000,0000	复位
0x0000,0004	未定义指令
0x0000,0008	软件中断
0x0000,000C	终止（预取指令）
0x0000,0010	终止（数据）
0x0000,0014	保留
0x0000,0018	IRQ
0x0000,001C	FIQ

- (a) 复位：当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处理程序处执行。
- (b) 未定义指令：当 ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常。采用这种机制，可以通过软件仿真扩展 ARM 或 Thumb 指令集。
- (c) 软件中断：该异常由执行 SWI 指令产生，可用于用户模式下的程序调用特权操作指令。可使用该异常机制实现系统功能调用。
- (d) 指令预取中止：若处理器预取指令的地址不存在，或该地址不允许当前指令访问，存储器会向处理器发出中止信号，但当预取的指令被执行时，才会产生指令预取中止异常。
- (e) 数据中止：若处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据

中止异常。

- (f) **IRQ**（外部中断请求）：当处理器的外部中断请求引脚有效，且 **CPSR** 中的 **I** 位为 0 时，产生 **IRQ** 异常。系统的外设可通过该异常请求中断服务。
- (g) **FIQ**（快速中断请求）：当处理器的快速中断请求引脚有效，且 **CPSR** 中的 **F** 位为 0 时，产生 **FIQ** 异常。

1.3 ARM 的中断过程

1.3.1 中断的进入

- (6) 将下一条指令的地址存入相应连接寄存器 **LR**，以便程序在处理异常返回时能从正确的位置重新开始执行。
- (7) 将 **CPSR** 复制到相应的 **SPSR** 中。
- (8) 根据异常类型，强制设置 **CPSR** 的运行模式位。
- (9) 强制 **PC** 从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序。也可以设置中断禁止位来阻止其他无法处理的异常嵌套。

1.3.2 从中断返回

- (1) 将连接寄存器 **LR** 的值减去相应的偏移量后送到 **PC** 中。
- (2) 将 **SPSR** 复制回 **CPSR** 中。
- (3) 如果进入时设置了中断禁止位，那么清除该标志。

1.3.3 标准中断过程

下面详细说明 **IRQ** 中断的过程。

- (1) **AIC** 已经正确编程，**AIC_SVR** 也已经写入正确的中断服务程序的入口地址。且中断已经使能。
- (2) 地址 **0x18**（**IRQ** 的中断向量地址，参见上面的向量表）的指令为

LDR PC, [PC, #&F20]

当 **NIRQ** 到来，且 **CPSR** 的 **I** 位为 0 时，步骤如下：

- (a) **CPSR** 被拷贝到 **SPSR_irq**，当前程序计数器 **PC** 的值被保存到 **IRQ** 链接寄存器(**R14_irq**)，同时 **PC** (**R15**) 自身也被赋予了新值 **0x18**。在接下来的时钟里（处理器向 **0x1C** 取指令），**ARM** 核使 **R14_irq** 减 4。
- (b) **ARM** 内核进入 **IRQ** 模式。
- (c) 当指令 **LDR PC, [PC, #&F20]** 得到执行（**ARM** 为流水线结构，当前 **PC** 之前还有两条指令）后，**PC** 被赋予了 **AIC_IVR** 的内容。读取 **AIC_IVR** 具有如下作用：
 - (i) 将当前中断设置为被挂起的最高优先级中断，并把它作为最高优先级的中断；当前中断级别则设置为此中断的优先级。
 - (ii) 将 **NIRQ** 的信号撤消（即使系统没有用到向量功能，也必须去读 **AIC_IVR**，以便将 **NIRQ** 撤消）。
 - (iii) 如果中断为边沿触发，则读取 **AIC_IVR** 会自动将中断清除。
 - (iv) 将当前的中断的优先级推入堆栈。
 - (v) 返回当前中断的 **AIC_SVR** 的值。
- (d) 上述步骤将程序跳到了对应的中断服务程序。接下来的第一步是保存链接寄存器 **LR**（**R14_irq**）和 **SPSR**（**SPSR_irq**）。如果需要在中断返回时，把 **LR** 的值直接赋给程序计数器，则 **LR** 首先要减去 4 才能保存。否则在中断返回时，**LR** 要首先减去 4 之后才能拷贝给 **PC**。
- (e) 清零 **CPSR** 的位 **I** 就可以使其他中断不被屏蔽，再施加的 **NIRQ** 可以被内核接受。只要发

生的中断的 优先级高于当前中断的优先级，嵌套中断就会发生。

- (f) 接着中断例程可以保存相应的寄存器以保护现场。如果此时有高优先级中断发生，则处理器将重复执行从步骤(1)开始的动作。要注意的是，如果中断是电平敏感的，那么在中断结束前要清除中断源。
- (g) 在退出中断前要首先置位 CPSR 的位 I，以便屏蔽其他中断，保证多个中断有序地完成。
- (h) 在结束中断之前还必须执行一次对 AIC_EOICR 的写操作，向 AIC 表明中断已经完成。存放于堆栈的前一个当前中断优先级将被弹出并作为当前中断优先级。如果此时系统又有一个挂起的中断，其优先级比刚才结束的中断的优先级低（或相等）、但又高于从堆栈弹出来的中断的优先级，则将重新施加 NIRQ；但是，中断步骤不会立即开始，因为此时 CPSR 的 I 位是置位的。
- (i) SPSR (SPSR_irq) 被恢复。最后是链接寄存器 LR 恢复到 PC。程序返回到中断发生前之处。SPSR 也恢复为 CPSR，中断屏蔽状态恢复为 SPSR 所指明的状态。

注意：SPSR 的位 I 是很重要的。如果在 SPSR 恢复之后为置位状态，则表明 ARM 核正要屏蔽中断，在执行屏蔽指令时被中断。因此，SPSR 恢复后，屏蔽指令得以完成，亦即 I 被置位，因而 IRQ 被屏蔽。

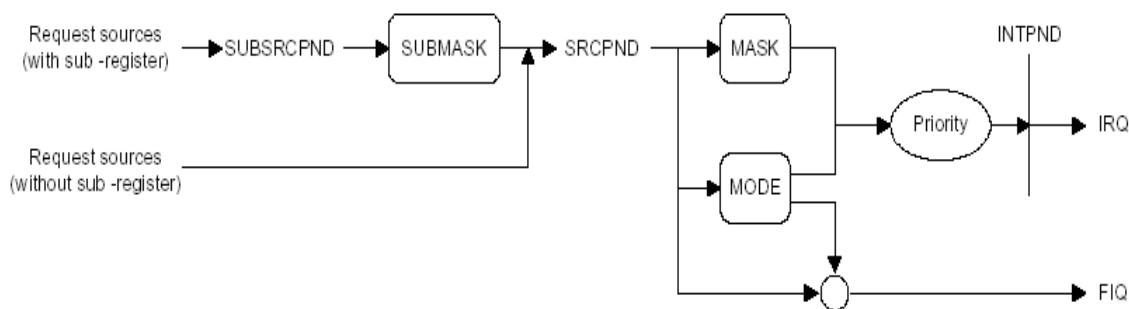
从上面可以看出 ARM 中断的通用过程。需要注意的是，在给相应 IRQ 脚中断信号前，必须先打开该中断的使能寄存器和正确设置对应的屏蔽寄存器，否则将不会有任何反应。当这两个寄存器都设置正确了，中断产生了，CPU 保存当前程序运行环境，跳到中断入口，ARM 芯片一般是 0x18 地址处。如果你没有设置中断向量，即 0x18 处不是你的代码，程序就会飞掉，当然也可能正常运行，这种情况一般发生在正好飞到正常代码处。设置好中断向量了，中断向量一般是个跳转语句，跳到你的正式的中断处理过程，在这里你可以关闭所有中断，清中断，处理等等，然后退出。记住某些处理器一定要清中断，否则下次再给中断信号时就没有反应了。

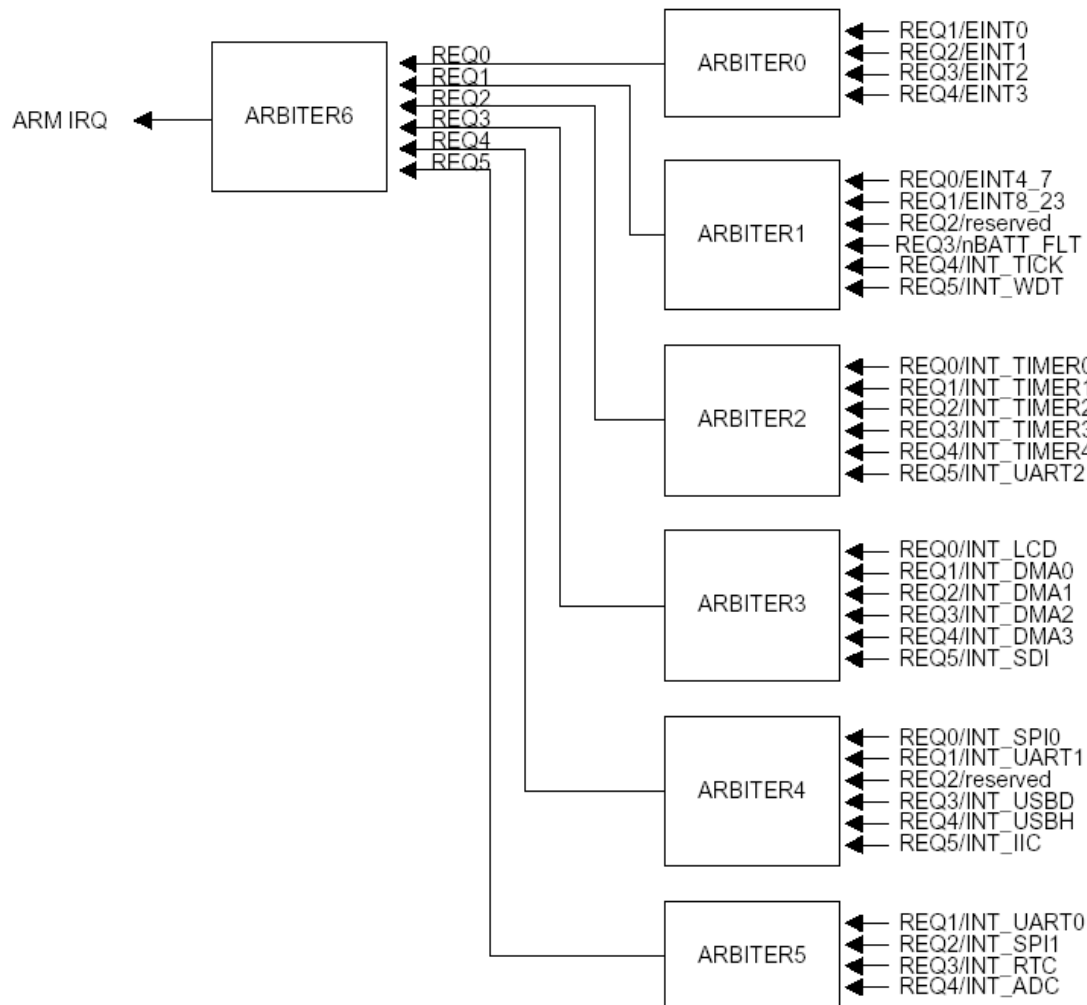
2. S3C2410 的中断系统

S3C2410 的中断控制器可以接收 56 路中断源的输入。这些中断源由如 DMA 控制器，UART，IIC 或其他内部外围设备提供的。它支持两种中断模式：FIQ 和 IRQ。每个中断源都可以决定中断请求时使用哪种模式。

当从内部外围设备或外部中断引脚接收到多个中断请求时，在经过中断裁决后，中断控制器就向 ARM920T 内核请求 FIQ 或者 IRQ 中断。仲裁过程依赖于硬件优先级逻辑，其结果写入中断待决寄存器(interrupt pending register)，通过查询它用户可以知道在各路中断源中产生的是哪路中断。

下图说明了中断控制器处理的过程和优先级裁决逻辑。





下面介绍一下相关的 5 个中断控制寄存器：源待决寄存器 **Source Pending Register**，中断模式寄存器 **Interrupt Mode Register**，中断掩码寄存器 **Interrupt Mask Register**，优先级寄存器 **Priority Register**，中断待决寄存器 **Interrupt Pending Register**。

中断源发出的所有的中断请求首先都要在源待决寄存器中注册。他们基于中断模式寄存器为两组：**FIQ** 和 **IRQ**。多个 **IRQ** 优先级的裁决依赖于优先级寄存器。

(1) 源待决寄存器 **SRCPND**

SRCPND 寄存器是 32 位的。如果中断源产生了中断请求，那么相应位设为 1，等待中断服务。注意的是，**SRCPND** 由中断源自动设置，而不管掩码位。而且它的值也不受优先级逻辑的影响。在中断服务例程中需要清除相应位(设为 0)。否则系统将认为是该中断源又产生了一次请求。

SRCPND	Bit	Description	Initial State
INT_ADC	[31]	0 = Not requested, 1 = Requested	0
INT_RTC	[30]	0 = Not requested, 1 = Requested	0
INT_SPI1	[29]	0 = Not requested, 1 = Requested	0
INT_UART0	[28]	0 = Not requested, 1 = Requested	0
INT_IIC	[27]	0 = Not requested, 1 = Requested	0
INT_USBH	[26]	0 = Not requested, 1 = Requested	0
INT_USBD	[25]	0 = Not requested, 1 = Requested	0
Reserved	[24]	Not used	0
INT_UART1	[23]	0 = Not requested, 1 = Requested	0
INT_SPI0	[22]	0 = Not requested, 1 = Requested	0
INT_SDI	[21]	0 = Not requested, 1 = Requested	0
INT_DMA3	[20]	0 = Not requested, 1 = Requested	0
INT_DMA2	[19]	0 = Not requested, 1 = Requested	0
INT_DMA1	[18]	0 = Not requested, 1 = Requested	0
INT_DMA0	[17]	0 = Not requested, 1 = Requested	0
INT_LCD	[16]	0 = Not requested, 1 = Requested	0
INT_UART2	[15]	0 = Not requested, 1 = Requested	0
INT_TIMER4	[14]	0 = Not requested, 1 = Requested	0
INT_TIMER3	[13]	0 = Not requested, 1 = Requested	0
INT_TIMER2	[12]	0 = Not requested, 1 = Requested	0
INT_TIMER1	[11]	0 = Not requested, 1 = Requested	0
INT_TIMER0	[10]	0 = Not requested, 1 = Requested	0
INT_WDT	[9]	0 = Not requested, 1 = Requested	0
INT_TICK	[8]	0 = Not requested, 1 = Requested	0
nBATT_FLT	[7]	0 = Not requested, 1 = Requested	0
Reserved	[6]	Not used	0
EINT8_23	[5]	0 = Not requested, 1 = Requested	0
EINT4_7	[4]	0 = Not requested, 1 = Requested	0
EINT3	[3]	0 = Not requested, 1 = Requested	0
EINT2	[2]	0 = Not requested, 1 = Requested	0
EINT1	[1]	0 = Not requested, 1 = Requested	0
EINT0	[0]	0 = Not requested, 1 = Requested	0

(2) 中断模式寄存器 INTMOD

INTMOD 寄存器也是 32 位的。如果某位设成 1，那么相应的中断源就以 FIQ 方式处理。否则以 IRQ 模式处理。

注意，控制器中仅仅一个中断源可以以 FIQ 方式处理。因此 INTMOD 只能有 1 位设成 1。

Register	Address	R/W	Description	Reset Value
INTMOD	0X4A000004	R/W	Interrupt mode register. 0 = IRQ mode 1 = FIQ mode	0x00000000

INTMOD	Bit	Description	Initial State
INT_ADC	[31]	0 = IRQ, 1 = FIQ	0
INT_RTC	[30]	0 = IRQ, 1 = FIQ	0
INT_SPI1	[29]	0 = IRQ, 1 = FIQ	0
INT_UART0	[28]	0 = IRQ, 1 = FIQ	0
INT_IIC	[27]	0 = IRQ, 1 = FIQ	0
INT_USBH	[26]	0 = IRQ, 1 = FIQ	0
INT_USBD	[25]	0 = IRQ, 1 = FIQ	0
Reserved	[24]	Not used	0
INT_URRT1	[23]	0 = IRQ, 1 = FIQ	0
INT_SPI0	[22]	0 = IRQ, 1 = FIQ	0
INT_SDI	[21]	0 = IRQ, 1 = FIQ	0
INT_DMA3	[20]	0 = IRQ, 1 = FIQ	0
INT_DMA2	[19]	0 = IRQ, 1 = FIQ	0
INT_DMA1	[18]	0 = IRQ, 1 = FIQ	0
INT_DMA0	[17]	0 = IRQ, 1 = FIQ	0
INT_LCD	[16]	0 = IRQ, 1 = FIQ	0
INT_UART2	[15]	0 = IRQ, 1 = FIQ	0
INT_TIMER4	[14]	0 = IRQ, 1 = FIQ	0
INT_TIMER3	[13]	0 = IRQ, 1 = FIQ	0
INT_TIMER2	[12]	0 = IRQ, 1 = FIQ	0
INT_TIMER1	[11]	0 = IRQ, 1 = FIQ	0
INT_TIMER0	[10]	0 = IRQ, 1 = FIQ	0
INT_WDT	[9]	0 = IRQ, 1 = FIQ	0
INT_TICK	[8]	0 = IRQ, 1 = FIQ	0
nBATT_FLT	[7]	0 = IRQ, 1 = FIQ	0
Reserved	[6]	Not used	0
EINT8_23	[5]	0 = IRQ, 1 = FIQ	0
EINT4_7	[4]	0 = IRQ, 1 = FIQ	0
EINT3	[3]	0 = IRQ, 1 = FIQ	0
EINT2	[2]	0 = IRQ, 1 = FIQ	0
EINT1	[1]	0 = IRQ, 1 = FIQ	0
EINT0	[0]	0 = IRQ, 1 = FIQ	0

(3) 中断掩码寄存器 INTMASK

INTMASK 同样是 32 位的，位模式和 SRCPND 相同。当某一位设置为 1 时，CPU 不会处理相应位的中断源请求，即使此时 SRCPND 中相应位是 1。

Register	Address	R/W	Description	Reset Value
INTMSK	0X4A000008	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced. 0 = Interrupt service is available. 1 = Interrupt service is masked.	0xFFFFFFFF

(4) 优先级寄存器 PRIORITY

PRIORITY 是 20 位的。它是 IRQ 的优先级控制寄存器。

Register	Address	R/W	Description	Reset Value
PRIORITY	0x4A00000C	R/W	IRQ priority control register	0x7F

(5) 中断待决寄存器 INTPND

INTPND 是 32 位的。每一位表明相应的中断请求(没有掩码, 正等待中断服务)是否有最高优先级。它在优先级逻辑之后所以只能有 1 位被设置, 该位向 CPU 产生 IRQ 请求。

Register	Address	R/W	Description	Reset Value
INTPND	0X4A000010	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

和 SRCPND 一样, 在中断服务例程中也要清除该位。注意的是, 不能向值为 1 的 INTPND 中写 '0'! 因为可能使得 INTPND 和 INTOFFSET 寄存器产生异常。最简捷的方法是用 INTPND 中已有的值来写 INTPND! 可以参见示例代码。

在 S3C2410 中, GPIO 可以很容易的配置成中断模式。因此下面也介绍一下与外部中断有关的几个寄存器。

(1) I/O 端口 F 的控制寄存器(GPFCON, GPFDAT, GPFPU)

在 S3C2410 系统中, 大多数引脚都可以复用, 因此需要控制寄存器(PnCON)来决定每个引脚用作什么功能。当端口配置成输入/输出模式时, 数据寄存器(PnDAT)可以读写引脚数据。拉高寄存器可以使能/无效每个引脚, 当相应位设成 0 时, 可以使能该引脚。

GPF 可以复用为 EINT0~EINT7。

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configure the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undefined
GPFUP	0x56000058	R/W	Pull-up disable register for port F	0x0
Reserved	0x5600005C	—	Reserved	Undefined

GPFCON	Bit	Description	
GPF7	[15:14]	00 = Input 10 = EINT7	01 = Output 11 = Reserved
GPF6	[13:12]	00 = Input 10 = EINT6	01 = Output 11 = Reserved
GPF5	[11:10]	00 = Input 10 = EINT5	01 = Output 11 = Reserved
GPF4	[9:8]	00 = Input 10 = EINT4	01 = Output 11 = Reserved
GPF3	[7:6]	00 = Input 10 = EINT3	01 = Output 11 = Reserved
GPF2	[5:4]	00 = Input 10 = EINT2	01 = Output 11 = Reserved
GPF1	[3:2]	00 = Input 10 = EINT1	01 = Output 11 = Reserved
GPF0	[1:0]	00 = Input 10 = EINT0	01 = Output 11 = Reserved

- (2) I/O 端口 G 的控制寄存器(GPGCON, GPGDAT, GPGUP)
和 GPF 相似，不同的是它可以复用为 EINT8~EINT23。

Register	Address	R/W	Description	Reset Value
GPGCON	0x56000060	R/W	Configure the pins of port G	0x0
GPGDAT	0x56000064	R/W	The data register for port G	Undefined
GPGUP	0x56000068	R/W	Pull-up disable register for port G	0xF800
Reserved	0x5600006C	–	Reserved	Undefined

GPGCON	Bit	Description	
GPG15	[31:30]	00 = Input 10 = EINT23	01 = Output 11 = nYPON
GPG14	[29:28]	00 = Input 10 = EINT22	01 = Output 11 = YMON
GPG13	[27:26]	00 = Input 10 = EINT21	01 = Output 11 = nXPON
GPG12	[25:24]	00 = Input 10 = EINT20	01 = Output 11 = XMON
GPG11	[23:22]	00 = Input 10 = EINT19	01 = Output 11 = TCLK1
GPG10 (5V Tolerant Input)	[21:20]	00 = Input 10 = EINT18	01 = Output 11 = Reserved
GPG9 (5V Tolerant Input)	[19:18]	00 = Input 10 = EINT17	01 = Output 11 = Reserved
GPG8 (5V Tolerant Input)	[17:16]	00 = Input 10 = EINT16	01 = Output 11 = Reserved
GPG7	[15:14]	00 = Input 10 = EINT15	01 = Output 11 = SPICLK1
GPG6	[13:12]	00 = Input 10 = EINT14	01 = Output 11 = SPIMOSI1
GPG5	[11:10]	00 = Input 10 = EINT13	01 = Output 11 = SPIMISO1
GPG4	[9:8]	00 = Input 10 = EINT12	01 = Output 11 = LCD_PWREN
GPG3	[7:6]	00 = Input 10 = EINT11	01 = Output 11 = nSS1
GPG2	[5:4]	00 = Input 10 = EINT10	01 = Output 11 = nSS0
GPG1	[3:2]	00 = Input 10 = EINT9	01 = Output 11 = Reserved
GPG0	[1:0]	00 = Input 10 = EINT8	01 = Output 11 = Reserved

- (3) 外部中断控制寄存器(EXTINTn)

24 路外部中断可以由各种信号模式触发。EXTINTn 寄存器配置信号触发方式。N 的值可以是 0~2。EXTINT0 配置 EINT0~7，EXTINT1 配置 EINT8~15，EXTINT2 配置 EINT16~23。
每个寄存器的每隔 3 位保留 1 位，剩余位中，每个外部中断占三位，000 表示低电平触发，

001 表示高电平触发，01X 表示下降沿触发，10X 表示上升沿触发，11X 表示两个边沿都触发。

Register	Address	R/W	Description	Reset Value
EXTINT0	0x56000088	R/W	External interrupt control register 0	0x0
EXTINT1	0x5600008C	R/W	External interrupt control register 1	0x0
EXTINT2	0x56000090	R/W	External interrupt control register 2	0x0

EXTINT0	Bit	Description
EINT7	[30:28]	Set the signaling method of the EINT7. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT6	[26:24]	Set the signaling method of the EINT6. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT5	[22:20]	Set the signaling method of the EINT5. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT4	[18:16]	Set the signaling method of the EINT4. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT3	[14:12]	Set the signaling method of the EINT3. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT2	[10:8]	Set the signaling method of the EINT2. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT1	[6:4]	Set the signaling method of the EINT1. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT0	[2:0]	Set the signaling method of the EINT0. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered

EXTINT1	Bit	Description
Reserved	[31]	Reserved
EINT15	[30:28]	Set the signaling method of the EINT15. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
Reserved	[27]	Reserved
EINT14	[26:24]	Set the signaling method of the EINT14. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
Reserved	[23]	Reserved
EINT13	[22:20]	Set the signaling method of the EINT13. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
Reserved	[19]	Reserved
EINT12	[18:16]	Set the signaling method of the EINT12. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
Reserved	[15]	Reserved
EINT11	[14:12]	Set the signaling method of the EINT11. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
Reserved	[11]	Reserved
EINT10	[10:8]	Set the signaling method of the EINT10. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
Reserved	[7]	Reserved
EINT9	[6:4]	Set the signaling method of the EINT9. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
Reserved	[3]	Reserved
EINT8	[2:0]	Set the signaling method of the EINT8. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered

EXTINT2	Bit	Description
FLTEN23	[31]	Filter Enable for EINT23 0 = Disable 1= Enable
EINT23	[30:28]	Set the signaling method of the EINT23. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN22	[27]	Filter Enable for EINT22 0 = Disable 1= Enable
EINT22	[26:24]	Set the signaling method of the EINT22. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN21	[23]	Filter Enable for EINT21 0 = Disable 1= Enable
EINT21	[22:20]	Set the signaling method of the EINT21. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN20	[19]	Filter Enable for EINT20 0 = Disable 1= Enable
EINT20	[18:16]	Set the signaling method of the EINT20. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN19	[15]	Filter Enable for EINT19 0 = Disable 1= Enable
EINT19	[14:12]	Set the signaling method of the EINT19. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN18	[11]	Filter Enable for EINT18 0 = Disable 1= Enable
EINT18	[10:8]	Set the signaling method of the EINT18. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN17	[7]	Filter Enable for EINT17 0 = Disable 1= Enable
EINT17	[6:4]	Set the signaling method of the EINT17. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN16	[3]	Filter Enable for EINT16 0 = Disable 1= Enable
EINT16	[2:0]	Set the signaling method of the EINT16. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered

(4) 外部中断过滤寄存器(EINTFLTn)

EINTFLTn 控制 EINT[23:16]共 8 路外部中断的过滤长度。n 可以为 0~3。不过 EINTFLT0 和 EINTFLT1 保留以备将来扩展用。

(5) 外部中断掩码寄存器(EINTMASK)

外部中断 EINT[23:4]的掩码寄存器。

(6) 外部中断待决寄存器(EINTPNDn)

外部中断 EINT[23:4]的待决寄存器。可以通过写该寄存器相应位为 0 可以清掉指定位。

Register	Address	R/W	Description	Reset Value
EINTPEND	0x560000A8	R/W	External interrupt pending register	0x0

EINTPEND	Bit	Description	
EINT23	[23]	0 = Not requested	1= Requested
EINT22	[22]	0 = Not requested	1= Requested
EINT21	[21]	0 = Not requested	1= Requested
EINT20	[20]	0 = Not requested	1= Requested
EINT19	[19]	0 = Not requested	1= Requested
EINT18	[18]	0 = Not requested	1= Requested
EINT17	[17]	0 = Not requested	1= Requested
EINT16	[16]	0 = Not requested	1= Requested
EINT15	[15]	0 = Not requested	1= Requested
EINT14	[14]	0 = Not requested	1= Requested
EINT13	[13]	0 = Not requested	1= Requested
EINT12	[12]	0 = Not requested	1= Requested
EINT11	[11]	0 = Not requested	1= Requested
EINT10	[10]	0 = Not requested	1= Requested
EINT9	[9]	0 = Not requested	1= Requested
EINT8	[8]	0 = Not requested	1= Requested
EINT7	[7]	0 = Not requested	1= Requested
EINT6	[6]	0 = Not requested	1= Requested
EINT5	[5]	0 = Not requested	1= Requested
EINT4	[4]	0 = Not requested	1= Requested
Reserved	[3:0]	0	

3. 代码需要使用的中断函数说明

(1) set_external_irq 函数

```
int set_external_irq(int irq, int edge, int pullup);
```

参数说明:

(a) irq: 需要设置的中断线号

(b) edge: 中断触发方式

(c) pullup: 是否拉高标志

该函数设置中断线相应的属性。在 **s3c2410** 中中断不必再像以前一样需要程序员手工设置各个相关寄存器了。该函数就可以完成注册中断前的各个设置工作，既简单又直观。

参数 **irq** 就是需要使用中断号。可以是自定义数值，也可以直接使用 **/HHARM9-EDU/kernel/arch/arm/kernel/irqs.h** 中预定义的值，例如本次实验使用的是外部中断 **5**，就可以直接使用 **IRQ_EINT5**。

S3c2410 支持各种触发方式，包括低电平触发，高电平触发，下降沿触发，上升沿触发，上升下降沿都触发方式。只需要将 **edge** 分别设为 **EXT_LOWLEVEL**，**EXT_HIGHLEVEL**，**EXT_FALLING_EDGE**，**EXT_RISING_EDGE**，**EXT_BOTH_EDGES** 就可以了。

拉高标志可以设为 **GPIO_PULLUP_EN** 和 **GPIO_PULLUP_DIS** 方式。当设为 **GPIO_PULLUP_EN** 时，在 **GPIO** 没有输入输出时，线路保持高电平。否则始终保持低电平。一般需要设置成拉高状态。

(2) request_irq 函数

```
int request_irq(unsigned int irq, void (*handler)(int,void *,struct pt_regs *),
               unsigned long irq_flags,const char * devname, void * dev_id);
```

该函数有 5 个参数：

(a) irq: 外设所使用的 IRQ 线号。

(b) handler 函数指针：设备驱动程序所实现的 ISR 函数。

(c) irqflags: 设备驱动程序指定的中断请求类型标志，它可以是下列三个值的“或”：SA_SHIRQ、SA_INTERRUPT 和 SA_SAMPLE_RANDOM。

(d) devname 指针：设备名字字符串。

(e) dev_id: 指向全局唯一的设备标识 ID，这是一个 void 类型的指针，可供设备驱动程序自行解释。

该调用定位中断源，使能中断线和 IRQ 处理函数。从该点处理函数才可能被触发。因为中断处理函数必须清除板子发出的任何中断，因此必须仔细的初始化硬件，并设置正确的中断处理函数。该调用当返回值为 0 表示成功，非 0 表示失败。

Dev_id 必须全局唯一。一般装置数据结构的地址作为 cookie 使用。因为处理函数接收该值。如果你的中断是共享的，那么在释放该中断的时候，必须传送一个非空 dev_id。

标志如下：

SA_SHIRQ: 中断是共享的。

SA_INTERRUPT: 当处理中断时，其他局部中断不可用。

SA_SAMPLE_RANDOM: 中断可以作为随机计量单位熵使用。

注意，在指定了中断共享标志 IRQ_SHIRQ 标志时，参数 dev_id 必须有效，不能为 NULL；IRQ 线号参数 irq 不能大于 NR_IRQS；且 handler 指针不能为 NULL。否则将出现错误号为-22 的参数无效错，无法注册中断服务程序。

(3) enable_irq 函数

```
void enable_irq(unsigned int irq);
```

该函数使能所选的中断线。参数 irq 就是 set_external_irq 中设置的中断线号。

(4) disable_irq 函数

```
void disable_irq(unsigned int irq);
```

该函数使得所选择的中断线无效。该调用一般和 enable_irq 配对使用，实现相反功能。

(5) free_irq 函数

```
void free_irq(unsigned int irq, void *dev_id);
```

该函数释放和中断号绑定的中断处理函数。如果没有其他装置使用该中断线，它将被无效。在共享中断时，调用者必须确保再调用该函数前中断线是无效的。

注意的是，该调用必须和 register_irq()一起使用，它的参数必须和 register_irq()里注册的参数一致，irq 是 register_irq()中声明的外设所使用的 IRQ 线号，dev_id 是 register_irq()中声明的设备标识 ID，如果 register_irq()里是 NULL，这里也必须为 NULL。特别注意，dev_id 不能是中断服务程序地址，否则在执行 rmmod 删除该模块时将出现错误，提示不能释放程序所使用的外部中断，必须重启系统才能再次 insmod 该模块。

如果想深入研究这些函数，请参见相关的源代码。`set_external_irq` 位于 `/HHARM9-EDU/kernel/arch/arm/mach-s3c2410/irq.c` 中，其他四个函数位于 `/HHARM9-EDU/kernel/arch/arm/kernel/irq.c` 中。

4. 中断编程中应该注意的问题

- (1) 中断程序的调试不能用单步执行，因为中断发生很快，无法跟踪。
- (2) 中断例程中尽量不要用 C 库函数，因为中断处理中所有的任务都被挂起。
- (3) 在中断调试过程中轮询方式简明，易于调试。在实时多任务程序中肯定用中断方式，不能用轮询。

中断出错时调试检查的方法（以 ARM IRQ 中断为例）：

- (1) 确定一下是否中断发生，在中断发生时，查中断标志寄存器 `SRCPND` 相应的 IRQ 位是否置 1，或用示波器量该 IRQ 管脚。有中断产生且中断 `enable`，就应该执行相应的 `ISR`。
- (2) 如上面没有问题，仍不执行，再查一下是否有比该 IRQ 等级更高的中断持续发生屏蔽了该 IRQ 中断。

特别要注意一些 `FIQ` 中断，在 ARM 中 `FIQ` 中断永远高于 `IRQ`，可能会导致 `IRQ` 中断不被执行。在中断屏蔽寄存器 `INTMASK` 中屏蔽所有优先级高于该 `IRQ` 的 `FIQ`，会 `disable` 这些 `FIQ`。

三. 实验内容和步骤

写一个中断服务程序，初始化时注册 `GPF5` 对应的中断（`EINT5`）。当 `GPF` 接收到按键产生的一次中断时，就进入中断服务处理程序打印出一个响应信息。注意程序必须写成驱动的形式，以进入内核代码流程。

1. 编写 `testirq_init` 函数。设置，使能和注册中断。需要注意的是，`register_irq` 里的 `irq` 参数必须是 `set_external_irq` 里已经设置了的 `irq`。否则将出现错误号为 -22 的 `EINVAL` 错误，提示参数无效。
2. 编写中断服务函数 `testirq_interrupt()`。注意，必须在该函数里清除中断，否则系统会将本次中断认为是又有了新的中断进来了。示例程序中打印出一个响应信息。我们可以在该函数中添加代码实现其他功能。例如 `GPF5` 对应 `EINT5`，我们可以打印和使用它的值。
3. 编译，用 `insmod` 方式运行。检查是否每按下一次键盘上的 `ESC` 键，就进入一次中断。除了查看打印出来的消息外，也可以用 `cat /proc/interrupts` 查看是否进入了中断。

四. 思考题

1. `S3c2410` 有多少个中断向量？它的中断向量表和微机的中断向量表相同吗？
2. 如果不使用系统提供的中断函数 `set_external_irq()`，那么需要手工设置哪些寄存器？
3. 如何清中断？注意清 `INTPND` 的特殊方法。