目 录

实验二十一：USB 接口实验

一．实验目的

通过本实验，使学生了解 usb 的工作原理。

二．实验原理和说明

1. USB 开发的基础步骤

(1) 详细阅读英文的 USB 标准。

(2) 找合适的芯片。根据对应用的分析，找到能满足性能并且有最方便外围接口的片子，但可能不会一次就能搞定，随着开发的深入，会有更换的想法，没什么，基调已经定了，浪费的是几个样片而已。

(3) 购买一套开发工具。一开始摸不着头脑的一大主原就是没有技术资料，没有文档，没有例子，不能直观地实际地体验，现在有了开发工具，就有了一切，没有什么值得发愁的了。

(4) 文档要看完，看透彻。

(5) 我看到很多说 bulk 传输速率如何如何慢的问题。在没有另外的 USB 的设备的理想情况下，即使一祯（1ms）里只安排一个 bulk 传输事务，1s 也能传输 64KB。

2. USB 设备的枚举过程

主机对 USB 设备的识别过程叫做枚举，一个完整的 Windows 对 USB 设备枚举的过程为（以本系统为例，其它设备可能稍微有些不同）：

(1) Get Device Descriptor。主机的第一个命令要求得到设备描述符，此 SETUP 包为 8 个字节数据（80，06，00，01，00，00，40，00）。“40”表示返回数据长度最大为 40H 个字节。实际上，只返回一个包，即数组 DEV_DESC[ ]中的前 8 个字节，用于说明设备的描述符的真实长度和设备的类型。

(2) Set Address。接着是设置设备地址处理事件，主机发送一个含有指定地址的数据包（00，05，02，00，00，00，00，00），在主机只有一个 USB 设备的时候，这个地址一般会是 2，最大地址 127，USB 协议中可以连接 127 个设备。设置地址事件处理结束后，设备进入地址状态，主机以后会在新的指定地址处访问设备。

(3) Get Device Descriptor。主机再次发送请求得到设备描述符的数据包（80，06，00，01，00，00，12，00），与上次不同的是，要求的数据的长度是实际的数据长度。因为 D12 设备每次只能发送 16 个字节，因此它会分两次完成此要求（“12”指 12H，十进制值为 18）。

(4) Get Configuration Descriptor。接着主机要求得到设备的配置描述符（80，06，00，02，00，00，09，00），包中数据“09”指定设备发送 9 个字节，这正是设备配置描述符的长度。

(5) 读取全部 Configuration Descriptor。接着主机要求得到设备全部的配置描述符、接口描述符和节点描述符（80，06，00，02，00，00，FF，00），由于主机不知道设备描述符的真实长度，因此它要求得到 256 个字节，实际上本系统中的 D12 发送 46 个字节就完成了此任务。

(6) 如果以上步骤都正确，主机将找到新设备，提示安装驱动程序；否则找到未知设备，不可用。 如果驱动程序安装成功，主机会再次以描述符的实际长度要求设备重新发送设备描述符和配置描述符；接着主机发送设置设备配置 SETUP 包，设备处理此事件，将允许所有节点进入工作状态；最后主机请求得到设备和接口的配置，如果设备成功应答，枚举

过程结束。此后 D12 状态灯应该一直亮。

3. 什么是前导包？

集线器通过禁止低速端口的方法来阻止高速事务处理在低速数据线上进行处理。在广播一个低速信息包之前，必须广播一个前导包，目的是通知所有的集线器在这个前导包后面跟随一个低速的事务处理。集线器必须对这个前导包作出响应，激活低速端口。

前导包格式： 由一个同步序列和一个包的 ID 组成。该报是全速传输。

4. USB 传输速度

USB 的速度 1.5Mb/s 是理想速度，USB 所谓的速率只是理论上的值，在实际上根本达不到。

因为有 10％的总线资源留给控制传输使用，另外还有一些必要的总线开销，例如位填充等等。

另外还有一个重要因素，就是 NAK 的存在会使传输速率下降。

对 USB 的传输速率有影响的因素包括：

(1) usb 的主机系统程序和客户驱动程序以及应用程序的处理速度；
(2) usb 系统的物理因素,如连线的电磁特性,usb 设备的类型(全速或低速)等；
(3) 采用何种传输模式,等时(同步),中断,控制,块(批)；
(4) 驱动程序中的端点描述符,接口描述符,设备描述符等的相关参数的设定情况；
(5) usb 设备的供电情况；
(6) 系统带宽的使用情况：

USB 的传输值达到 1M 的方法：修改 D12 的 CLKOUT 的输出，让它输出 24MHz 的时钟，在 BULK 传输的时候每次都直接让它发送 64 个字节。经过同时优化 windows device driver 和 firmware，结果传输 16MByte 的数据用了 33 秒！计算一下达到了 496KByte/s。如果使用可以运行 50MHz 的 MCU，速度会加倍，达到 992KByte/s。

当然，如果要实际的每次给 D12 写 64byte 的数据，速度就会慢许多，基本上会减半。D12 因为有 double buffering 的功能,可以每次向它写 128bytes。而且,在 driver 里面直接把 16Mbyte 的数据分成 64K，发给 usbd，全部完成以后再一次性的返回给 app，这样就得到了上面说的数据了。

因此，只要有合适的 MCU，再加上时序的调整，driver 和 app 的配合，很有希望得到一个 800~900k 的传输速率。

另外，由于 usb 的传输和 usbd.sys 的速度也有很大关系，因此用一个速度快的 PC 和可以得到更快的传输。

(7) Data0/Data1

DATA0 ,DATA1 主要是用于数据的检错，它是数据报的前导字段，在数据传输过程中，依次是“DATA0,DATA1...”。

在 EZ-USB 系列芯片中,其数据触发的工作大部分由 EZ-USB 核心完成,只有 Set_feature，Set_configuration，Set_interface 由 8051 完成。

DATA0 和 DATA1 用于数据的错误恢复，ISO 仅仅使用 DATA0， 其它传输将交错的使用 DATA0 和 DATA1。

对于主机的 DRIVER 而言，无需考虑，STACK 会自动处理的。

对于设备，如果 HARDWARE 不能自动处理，参见 USB 规范 1.1 第 8.6 节。

(8) pdiusbd12 的 int#总是低的 BUG

中断脚不应该悬空，因为 D12 的 INT，GOODLINK，SUSPEND 脚都是集电极开路输出的，必须上拉，需要在上电以后，把所有的中断寄存器都读一下，这样它就会保证变高了。这个问题很早以前就在 usb.org 的 webboard 上被提出来过。

(9) Interrupt Transfer

USB 和 PC 的通讯都是由 PC 发起的（不管是向 USB 里面送数据还是送 USB 取数据）使用 Interrupt Transfer，使 USB 处理完后主动向 PC 发信息，使 PC 端可以检测 USB 的端点中存在待发送的数据了。

这些是 usb 内核会自动处理的,你只要把数放到缓冲区中就行。

USB 的中断传输并不是真正意义上的中断，而是类似于 BULK 类型的一种传输方式，只不过是由主机端不停的进行查询而已，至于查询的频率是在设备描述符的一个参数中进行设置。

具体参数在就在配置描述符里面。应该对每个 EndPoint 都有一个描述符，那里面有相应的设置，其中最后一项用来设置间隔时间。

(10) Endpoint & Pipe

端点是硬件设备中的一组缓冲区，用来与 PC 机交换数据用的，在 PC 机中，为了与这几个所谓的端点通信，就把它封装成一个管道（其中包含端点的地址，端点缓冲区的长度等），这样 PC 机与之通信的时候就只要用这个管道的句柄就可以方便的与之交换数据了。

端点（ENDPOINT）：每一个 USB 设备在主机看来就是一个端点的结合，主机只能通过端点与设备进行通信，以使用设备的功能；每一个端点实际上就是一个一定大小的数据缓冲区。

管道（PIPE）：一个 USB 管道是驱动程序的一个数据区缓冲与一个外设端点的连接，它代表了一种在两者之间移动数据的能力。一旦设备被配置，管道就存在了。

pipe （管道）并不是一个实际存在的物理实质，只是逻辑上的一个东西，比如 d12 芯片有三个端点，那它在被配置完之后就会有三个管道和主机通信。在通信时并不需要指明哪个通道，只要把数据写入一个端点，那个端点自然会用它自己与主机之间的管道传输数据。而你说的 bulk pipe，实际上说 bulk endpoint 我觉得更确切，因为在上电初期的协议传输阶段，应该是把每一个管道的传输方式定义成四种中的一个(实际上并不是都能使任意是四种之一，管道零一定是控制传输)，所以 bulk(批量)应该是管道的传输方式。

(11)Suspend & Resume：关于 D12 的 SUSPEND 引脚问题

D12 的 suspend 引脚是双向的，也就是说，即作输出也做输入。

比如说 d12 上电时，suspend 为低，如果 d12 发现总线空闲，就会将 suspend 拉高，这种情况下的 suspend 便起到输出作用。

当 PDIUSBD12 处于挂起模式时，内部寄存器不能被访问，如果需要对设备进行访问，将 PDIUSBD12 的挂起脚拉低唤醒设备然后进行访问，这种情况 suspend 作为输入用。最关键的是要对 d12 操作前，suspend 一定要拉低。

(12)关于 AN2131

firmware 中的 EP Pair 把两个 buffer 连在一起使用，可以提高传输速度。例如 OUTEP2 和 OUTEP3 做 PAIR EP PAIR 后向 OUTEP2 发数据，原 OUTEP2 的缓冲区装入数据，在 OUT2BC 没有清零前，只要原 OUTEP3 的缓冲区没有使用就可以继续向 OUTEP2 发数据，AN2131 会自动将其放入原 OUTEP3 的缓冲区里，这样只要数据处理速度够快就可以交错使用两个 BUF，

不断发送数据。表达能力不好，具体请仔细阅读 CYPRESS 的文档。

经过设置 pair 设置后，把两个 buffer 当一个用，只用一个 ISR 和 Busy 标志，首先在 ISR 中，装载 64 个数，然后 IN2BC = 64，然后再判断是否 busy，如果 busy = 0，Buf2 进行赋值当进行大块数据读的时候，比如 HOST 端一次要求读 4096BYTE，速度就有区别了。

(13) s3c2410 的 usb host 不稳定的解决方法

可能是 s3c2410 芯片本身的问题, 导致启动时无法正确初始化 UPLLCON 寄存器，使之给 USB 输出 48.00MHz 的时钟频率，于是修改了下 driver/usb/usb.c 的代码：

```
/*************************/
--- drivers/usb/usb.c 2004-07-27 16:59:28.000000000 +0800
+++ ../kerne-1version/drivers/usb/usb.c 2004-07-03 15:15:32.000000000 +0800
@@ -38,13 +38,6 @@
#endif
#include <linux/usb.h>
#include <asm/hardware.h>
#include <asm/irq.h>
#include <asm/io.h>
#include <asm/pci.h>
#define OHCI_HW_DRIVER 1
#include "usb-ohci.h"

static const int usb_bandwidth_option =

#ifdef CONFIG_USB_BANDWIDTH 1;
@@ -2268,15 +2261,6 @@
dev->epmaxpacketout[0] = 8;
err = usb_set_address(dev);
if( err < 0 )
{
    CLKCON &= ~CLKCON_USBH;
    UPLLCON = FInsrt(0x78, fPLL_MDIV) | FInsrt(0x02, fPLL_PDIV) | FInsrt(0x03, fPLL_SDIV);
    CLKCON |= CLKCON_USBH;
    err = usb_set_address(dev);
    printk("first usb_set_address error\n");
}
if (err < 0) {
    err("USB device not accepting new address=%d (error=%d)", dev->devnum, err);
}
/*************************/
```

使得再 usb_set_address 失败的时候，再设置一次 UPLLCON,这样就不会有问题了， 另外我的 driver/usb/usb-ohci-s3c2410.c 用的是 mizi.com 上的 2.4.19 版本。

dmesg 内容如下:

hub.c: 2 ports detected
hub.c: USB new device connect on bus1/1, assigned device number 2
first usb_set_address error ---- 这句话表明重新设置 UPLLCON 了
usb.c: USB device 2 (vend/prod 0xdd8/0x3) is not claimed by any active driver.
Initializing USB Mass Storage driver...
usb.c: registered new driver usb-storage
scsi0 : SCSI emulation for USB Mass Storage devices
Vendor: Model: Rev:
Type: Direct-Access ANSI SCSI revision: 02
Attached scsi removable disk sda at scsi0, channel 0, id 0, lun 0
可以看见， u 盘被正确地找到了。 如果有人有更好的方法请指出，谢谢。

5. USB 入门介绍

现在电脑系统连接外围设备的接口并无统一的标准，如键盘用 PS/2 接口，连接打印机要用 25 针的并行接口，鼠标则要用串行或 PS/2 接口。USB 则将这些不同的接口统一起来，使用一个 4 针插头作为标准插头。通过这个标准插头，采用菊花链形式可以把所有的外设连接起来，并且不会损失带宽。

USB 规范中将 USB 分为五个部份：控制器、控制器驱动程序、 USB 芯片驱动程序、USB 设备以及针对不同 USB 设备的客户驱动程序。

根据设备对系统资源需求的不同，在 USB 规范中规定了四种不同的数据传输方式：等时传输方式(Isochronous)、中断传输方式(Interrupt)、控制传输方式(Control)和批(Bulk)传输方式，这些传输方式各有特点，分别用于不同的场所。

USB 需要主机硬件、操作系统和外设三个方面的支持才能工作。目前主板一般都采用支持 USB 功能的控制芯片组，而且也安装了 USB 接口插座。Windows98 操作系统内置了对 USB 功能的支持（但 WindowsNT 尚不支持 USB）。目前已经有数字照相机、数字音箱、数字游戏杆、打印机、扫描仪、键盘、鼠标等很多 USB 外设问世。

随着大量的支持 USB 的个人电脑的普及以及 Windows98 的广泛应用，USB 逐步成为 PC 机的一个标准接口已经是大势所趋。最新推出的 PC 机几乎 100％支持 USB，另一方面使用 USB 接口的设备也在以惊人的速度发展。

USB 是英文 Universal Serial Bus 的缩写，中文含义是“通用串行总线”。它不是一种新的总线标准，而是应用在 PC 领域的新型接口技术。早在 1995 年，就已经有 PC 带有 USB 接口了，但由于缺乏软件及硬件设备的支持，这些 PC 机的 USB 口都是闲置未用的。1997 年，微软在 WIN95OSR2（WIN97）中开始以外挂模块的形式提供对 USB 的支持， 1998 年后随着微软在 Windows98 中内置了对 USB 接口的支持模块，加上 USB 设备的日渐增多， USB 逐步走进了实用阶段。

(1) USB 的历史及发展

在谈论 USB 技术之前，不妨让我们来看看外设接口技术的发展历程。多年来个人计算机的串口与并口的功能和结构并没有什么变化。串口的出现是在 1980 年前后，数据传输率是 115kbps～230kbps，串口一般用来连接鼠标和外置 Modem；并口的数据传输率比串口快 8 倍，标准并口的数据传输率为 1Mbps，一般用来连接打印机、扫描仪等。原则上每一个外设必须插在一个接口上，如果所有的接口均被用上了就只能通过添加插卡来追加接口了。串并口不仅速度有限，而且在使用上很不方便。

1994 年，Intel、Compaq、 Digital、IBM、Microsoft、NEC、Northern Telecom 等七家世

界著名的计算机和通讯公司成立了 USB 论坛，花了近两年的时间形成了统一的意见，于 1995 年 11 月正式制定了 USB0．9 通用串行总线（Universal Serial Bus）规范，1997 年开始有真正符合 USB 技术标准的外设出现。USB1．1 是目前推出的在支持 USB 的计算机与外设上普遍采用的标准。1999 年初在 Intel 的开发者论坛大会上，与会者介绍了 USB2．0 规范，该规范的支持者除了原有的 Compaq、Intel、Microsoft 和 NEC 四个成员外，还有惠普、朗讯和飞利浦三个新成员。USB2．0 向下兼容 USB1．1，数据的传输率将达到 120Mbps～ 240Mbps，还支持宽带宽数字摄像设备及下一代扫描仪、打印机及存储设备。

目前普遍采用的 USB1．1 主要应用在中低速外部设备上，它提供的传输速度有低速 1．5Mbps 和全速 12Mbps 两种，低速的 USB 带宽（1．5Mbps）支持低速设备，例如显示器、调制解调器、键盘、鼠标、扫描仪、打印机、光驱、磁带机、软驱等。全速的 USB 带宽（12Mbps）将支持大范围的多媒体设备。

继 USB 之后，另一种称为 FIREWIRE(即 IEEE 1394)的接口技术正在从实验室步入市场领域，这种新型的接口比 USB 功能更为强大而且性能稳定。

IEEE 1394 也是一种高效的串行接口标准。IEEE 1394 可以在一个端口上连接多达 63 个设备，设备间采用树形或菊花链拓扑结构。IEEE 1394 标准定义了两种总线模式，即：Backplane 模式和 Cable 模式。其中 Backplane 模式支持 12.5、25、50Mbps 的传输速率；Cable 模式支持 100、200、400Mbps 的传输速率。目前正在开发 1Gbps 的版本。

现在，支持 USB 的 PC 及外设越来越多，在软件上 USB 也已成为 Windows98 的一个关键部件，并很快在 WindowsCE 和 Windows2000 中得到支持。Apple 的操作平台早已提供对 USB 的支持，预计今后 Sun 和 Digital 的平台也将会提供对这一技术的支持。

(2) USB 的特点

USB 之所以能得到广泛支持和快速普及，是因为它具备下列的很多特点:

(a) 使用方便

使用 USB 接口可以连接多个不同的设备，支持热插拔，在软件方面，为 USB 设计的驱动程序和应用软件可以自动启动，无需用户干预。 USB 设备也不涉及 IRQ 冲突等问题，它单独使用自己的保留中断，不会同其它设备争用 PC 机有限的资源，为用户省去了硬件配置的烦恼。USB 设备能真正做到“即插即用”。

(b) 速度加快

快速性能是 USB 技术的突出特点之一。USB 接口的最高传输率目前可达 12Mb / s，比串口快了整整 100 倍，比并口也快了十多倍。今后 USB 的速度还将会提高到 100Mb / s 以上。

(c) 连接灵活

USB 接口支持多个不同设备的串列连接，一个 USB 口理论上可以连接 127 个 USB 设备。连接的方式也十分灵活，既可以使用串行连接，也可以使用中枢转接头 (Hub)，把多个设备连接在一起，再同 PC 机的 USB 口相接。在 USB 方式下，所有的外设都在机箱外连接，不必打开机箱；允许外设热插拔，而不必关闭主机电源。USB 采用“级联”方式，即每个 USB 设备用一个 USB 插头连接到一个外设的 USB 插座上，而其本身又提供一个 USB 插座供下一个 USB 外设连接用。通过这种类似菊花链式的连接，一个 USB 控制器可以连接多达 127 个外设，而每个外设间距离(线缆长度)可达 5 米。USB 还能智能识别 USB 链上外围设备的接入或拆卸。

(d) 独立供电

普通使用串口、并口的设备都需要单独的供电系统，而 USB 设备则不需要，因为 USB 接口提供了内置电源。USB 电源能向低压设备提供 5V 的电源，因此新的设备就不需要

专门的交流电源了,从而降低了这些设备的成本并提高了性价比。

(e) 支持多媒体

USB 提供了对电话的两路数据支持，USB 可支持异步以及等时数据传输,使电话可与 PC 集成,共享语音邮件及其它特性。USB 还具有高保真音频。由于 USB 音频信息生成于计算机外，因而减少了电子噪音干扰声音质量的机会,从而使音频系统具有更高的保真度。

(f) USB 存在的问题

尽管在理论上，USB 可以实现高达 127 个设备的串列连接，但是在实际应用中，也许串联 3 到 4 个设备就可能导致一些设备失效。而且大多数 USB 产品，只有一个输入口，根本无法再连接下一个 USB 设备。另外，尽管 USB 本身可以提供 500mA 的电流，但一旦碰到高电耗的设备，就会导致供电不足。解决这些问题的办法是使用 USBHub，但 Hub 的价格目前还太贵了点。

(3) USB 的应用

到目前为止，USB 已经在 PC 机的多种外设上得到应用，包括扫描仪、数码相机、数码摄像机、音频系统、显示器、输入设备等等。

扫描仪和数码相机、数码摄像机是从 USB 中最早获益的产品。传统的扫描仪，在执行扫描操作之前，用户必须先启动图像处理软件和扫描驱动软件，然后通过软件操作扫描仪。而 USB 扫描仪则不同，用户只需放好要扫描的图文，按一下扫描仪的按钮，屏幕上会自动弹出扫描仪驱动软件和图像处理软件，并实时监视扫描的过程。USB 数码相机、摄像机更得益于 USB 的高速数据传输能力，使大容量的图像文件传输在短时间内即可完成。

USB 在音频系统应用的代表产品是微软公司推出的 Microsoft DigitalSound System80(微软数字声音系统 80)。使用这个系统，可以把数字音频信号传送到音箱，不再需要声卡进行数模转换，音质也较以前有一定的提高。USB 技术在输入设备上的应用很成功，USB 键盘、鼠标器以及游戏杆都表现得极为稳定，很少出现问题。

早在 1997 年，市场上就已经出现了具备 USB 接口的显示器，为 PC 机提供附加的 USB 口。这主要是因为大多数的 PC 机外设都是桌面设备，同显示器连接要比同主机连接更方便、简单。目前市场上出现的 USB 设备还有 USB Modem、Iomega 的 USB ZIP 驱动器以及 eTek 的 USB PC 网卡等等。

对于笔记本电脑来说，使用 USB 接口的意义更加重大，通用的 USB 接口不仅使笔记本电脑对外的连接变得方便，更可以使笔记本电脑生产厂商不再需要为不同配件在主板上安置不同的接口，这使主板的线路、组件的数量以及复杂程度都有不同程度的削减，从而使系统运行中的散热问题得到了改善。也将促进更高主频的处理器可以迅速应用在移动计算机中，使笔记本电脑与桌面 PC 的差距进一步缩小。

USB 的应用会越来越广泛，一些业界人士甚至预测，未来的 PC 将是一个密封设备，所有外设都将通过 USB 或其他外部接口连接。

(4) uu12288 USB

uu12288 USB(Universal Serial Bus)的中文名叫“通用串行总线”。这是近两年在 PC 领域广为应用的新型接口技术。理论上讲，USB 技术由三部份组成：具备 USB 接口的 PC 系统、能够支持 USB 的系统软件和使用 USB 接口的设备。1997 年，微软推出 Win95、Win97 后，USB 进入实用阶段，但由于这个版本对 USB 的支持属于外挂式模块，因此直到 Win 98 推出后，USB 接口的支持模块才真正日趋成熟。因此，从某种意义上讲，Win98 成了 USB 技术发展和应用的“催生婆”。由于安装简单，使用方便，据 Dataquest 公司统计结果显示，仅去年全球已有 1000 万台 USB 设备售出，预计这个数字到今年会增加到 5000 万台，而到 2000 年

这个数字可达近 1 亿台。

(a) USB HUB 的连接方法

uu12288 目前新型的主板一般只有两个内建的 USB 接口(图 1)，要连接 4 个以上的 USB 设备就必须加装 USB HUB(图 2)，通过 USB HUB 来扩充 USB 接口数量。大家对 HUB 并不陌生，前边加上个 USB，功能与传统的产品完全一样，只不过属于某个领域内的固有集线器了。USB HUB 可以连接 USB 设备，同时也可以串接另外一个 USB HUB。但是 USB HUB 连续串接时不能超过三个。从规格上来分析，通过 USB 最多可以连接 127 个设备，这个数字其实也只是理论上的，谁又会在一台电脑上同时连接超过一百个以上周边装置（又能装些什么呢）。因此，USB HUB 的安装也比较“傻瓜式”。第一步要做的就是启动主板上的 USB 接口。检查在 CMOS SETUP 中的 USB 选项，如果被 Disabled 掉了，请将此选项改成 Enabled，存盘后重新进入 Windows 便可找到 USB 控制器。一般的 HUB 有一对二、一对四、一对五的三种类型。所谓一对二，就是通过原来的一个 USB 接口，扩充出两个 USB 接口。说是一对二，但由于会占用原先的一个 USB 口，因此虽然扩充出两个接口，但实质上只多出一个 USB 接口(见图 3)。依此类推，一对四便可多出三个 USB 接口 ,而一对五则可多出四个 USB 接口（接口越多 HUB 的价格当然也就越高，相对的耗电量也会增加）。以一对四的 USB HUB 安装为例，这种 USB HUB 有 1 个输入接头和 4 个输出接头(见图 4)。输出接头与输入接头的样子不太一样，比较容易区分。同时，随 HUB 一般都会提供一条连接 USB 装置的导线，在导线一端的接头是用来连接 USB 装置(或 USB HUB)的输入端；相对的，在导线另一端的接头则是用来与 USB HUB 输出端连接的部份，依次对接安装就可以了。值得注意的是，现在许多 USB 设备本身其实已经具有 USB HUB 的功能。笔者曾经见过这样一台新型的显示器（17 英寸），其机壳背面有四个 USB 输出接头(当然，还有一个是 USB 输入接头)，很显然，这台显示器也可以担当一个 USB HUB 的责任了。还有一点就是电源，一对二的 USB HUB 通常没有外接电源，而一对四的 USB HUB 则大部份附有电源适配器，不过一对四的 USB HUB 不接电源也是可以工作的，只是每个接口只能供电约 100mA 左右，而一旦接上电源适配器，则可提升至 500mA 左右。接不接电源主要取决于用户自身，我建议你在具体安装时，将耗电量高者接于电脑上的 USB 接口，将不需要电源仍可正常运作的低耗电量 USB 设备，如鼠标、Modem 等接往一对四的 USB HUB。

(b) 升级

uu12288 对于那些电脑主板没有内置 USB 接口的用户，并不等于你就不能使用 USB。用户可以通过购买 USB 接口卡来升级自己的电脑，使之具备 USB 功能。现在市场上常常可见的 USB 卡都是 PCI 卡，自带两个 USB 端口，你只要把这块卡插入主板的 PCI 插槽中，安装上随卡附送的驱动软件就万事 OK 了。具体安装很简单，不用任何电源线和信号线，像安装显卡、声卡一样，只要插上即可。目前，市面上提供 USB 卡的著名厂商大体有两家，一家是 Entrega，另一家是 Belkin，质量都还可以。

(c) 安装实例

uu12288 到目前为止，USB 已经在 PC 机的多种外设上得到应用，输出设备方面，包括扫描仪、数码相机、数码摄像机、音频系统、显示器等等。扫描仪、数码相机和数码摄像机是最早在 USB 技术上显山露水的高手。这几种产品的迅速跟进主要还是得益于 USB“一触即发”的高速数据传输能力，使大容量的图像传输在短时间内即可完成。输入设备方面，USB 键盘、鼠标器以及游戏杆都表现得极为稳定，很少出现问题。另外，还有 DSL 的 USB“猫”、Iomega 的 USB ZIP 驱动器以及 eTek 的 USB PC 网卡等等。特别是，如今越来越多的笔记本

电脑也武装上了 USB 接口，这并不是说笔记本电脑可以从 USB 接口中获得多大的好处，关键在于那些经常在台式机和笔记本电脑之间传输数据的用户，可以使用 USB 来节省大量的时间。下面以麦宝 USB 声卡为例介绍其连接安装方法。打开包装，可以见到这块 USB 声卡的外观颇有些像鼠标。只不过与鼠标不同，它屁股底下有两根连线，一根用来连接 USB 插座，而另一根则用来连接音箱。与所有的 USB 设备一样，这种 USB 声卡的安装非常简单，只要将这块卡接入电脑上的 USB 接口，重新启动电脑后，Win98 就自动检测到了该设备，并自动安装了 Win98 中自带的驱动程序。由于是 USB 设备，该卡同样也支持热插拔，不必关机便可进行安装。uu12288 经试听，在 PII300/64M/ PC WORK4.1 上，总体感觉音质和音效都还不错。但在一些支持 A3D 的游戏中，与 DIAMOND MX300 等高档声卡相比较则明显不足，显然没有对此进行优化。同时，这块 USB 声卡功放较低，也使其在整体档次上稍显不足。

(d) uu12288 存在的问题

以上介绍了 USB 的诸多妙处，当然 USB 也非尽善尽美。

(i) 连接方面的问题

当你购买了一款 USB 外设后，应该试着用一用主板上所有的 USB 接口。按以往的经验，有时将一个 USB 外设插入到从未使用过的 USB 接口时会导致系统崩溃。另外，USB 标准规定可以以菊花链的形式将外部设备连接在一起，这就要求所有产品都应该有两个 USB 接口，但现在销售的许多 USB 外设只有一个接口，使得其他 USB 产品无法串联到 USB 总线上。正如我们前面已经讲到的，现在有些 USB 的显示器可以提供 4 个端口的 USB HUB 功能，而且市场上现在也能够看到具有 USB HUB 功能的键盘可以提供两个 USB 接口。不过即使如此，我还是建议你节约使用 USB 接口，如果让键盘、鼠标、手柄统统占据了本来为数不多的 USB 接口，以后数字照相机、扫描仪、摄像头等真正需要 USB 的设备就无从立足了，寄全部希望于 HUB 也不是上策，毕竟要花不少银子哟。

(ii) USB 产品的供电问题

尽管 USB 总线能将我们从纠缠不清的电缆线、信号线的桎梏中摆脱出来，但 PC 电脑上的 USB 接口往往不能提供 4 台以上 USB 外设的电源供应。因此，当系统同时存在多个 USB 产品时，有必要考虑外加一个可供电的 USB 集线器。

(iii) USB 接口问题

USB(Universal Serial Bus)接口(外观如图 3)的提出是基于采用通用连接技术，实现外设的简单快速连接，达到方 便用户、降低成本、扩展 PC 机连接外设的范围的目的。目前 PC 中似乎每个设备都有它自己的一套连接设备。外设接口的规格不一、有限的接口数量，已无法满足众多外设连接的迫切需要。解决这一问题的关键是，提供设备的共享接口来解决个人计算机与周边设备 的通用连接。

uu12288 USB 技术应用是计算机外设连接技术的重大变革。现在 USB 接口标准属于中低速的界面传输，面向家庭与小型办公领域的中低速设备。比如键盘、鼠标、游戏杆、 显示器、数字音箱、数字相机以及 Modem 等，目的是在统一的 USB 接口上实现中低速外设的通用连接。PC 主机上只需要一个 USB 端口，其他的连接可以通过 USB 接口和 USB 集线器在桌面上完成。USB 系统由 USB 主机（HOST）、集线器（HUB）、连接电缆、USB 外设组成。下一代的 USB 接口， 数据传输率将提高到 120Mbps～240Mbps，并支持宽带宽数字摄像设备及新型扫描仪、打印机及存储设备。 虽然早在 1997 年，Intel 440 LX 芯片组就提供了 USB 功能，但由于当时支持 USB 的设备极少，这一技术并没有引起用户过多的关注。时至今天，支持 USB 的设备数量已经相当可观， USB 接口的种种优势正凸显出来，在 PC99 规范中，USB 已经是个人电脑的标准设备之一。可以预见，以后的主板上将没有 PS／2、COM 等规格不一

的烦人的外设接口，取而代之的是数个 USB 接口，所有的外设都通过这一接口连接。事实上，今年风靡一时的 iMac 已经先行一步，它的键盘、鼠标都使用 USB 接口，非常前卫，不过，PC 机用户也不用眼红，现在有大量的 USB 设备可供选择，现在我们就来看看这些时代前沿的产品吧。

(5) 前沿产品

(a) USB 音箱

uu12288 现在具有 USB 音箱生产能力的厂家众多，较为有名的有 J－S 淇誉电子的“爵士” 音箱、漫步者、麦蓝。而目前市场上较为流行的有 J－S 的 J6502 和漫步者的 USB1900T / USB1000TC。 uu12288 漫步者的 R1900 / R1000 是典型的木质音箱，提供两路模拟信号输入接口，一个 USB 数字信号输入接口。除多了一个 USB 接口外，其它方面没有任何改变。在音箱后座多了一块 USB 控制电路板，从形式上来看，就是把一块 USB 声卡从外部移到了音箱内部。 uu12288 淇誉电子的 J6502 也是采用 USB 接口的塑料音箱，带有模拟信号输入接口。谈到塑料音箱，普通用户有一种误解，认为木质音箱就一定比塑料音箱声音质量更高。以前这种想法也许正确，但随着塑料材料的不断改进，其密度可以设计到合乎声学要求的程度，在一些不是很专业的场合，塑料音箱已开始崭露头角。最重要的是，塑料音箱可以做出各种各样具有个性的外观（大家很少看到外形多样的木质音箱吧），与整个计算机及外设达到和谐的统一，国外的多媒体音箱大多采用塑料音箱也正是这个道理。从声音表现来看，J6502 的中高音细致逼真，低音效果尚可，但其震撼力度不够，不过，对于放在电脑桌上的多媒体音箱来说，其音质表现完全能够满足用户要求。

(b) USB 声卡

uu12288 现在流行的 A3D 和 EAX 环境音效技术及 64 位 DLS 波表让用户非常满意，而厂商针对声卡发挥想象的空间却越来越小了。不过，USB 的流行让一些挖空心思找卖点的厂商嗅到了吸引用户的香味。USB 声卡的出现就是这些厂商的杰作。 uu12288 USB 声卡可满足那些已经买了传统的模拟音箱但又想体味数字音频技术的用户的味口。由于采用 USB 接口，USB 声卡就成为外置式设备。USB 声卡的发声原理很简单：从 USB 接口获取数字音频，经过 USB 声卡的 D / A（数 / 模）转换输出到传统音箱上。

uu12288 与传统声卡相比，USB 声卡具有以下优点：

(i) 信号的来源不同。传统声卡的信号来源于数字音频文件、CD－ROM 音频或其它模拟音频信号，而 USB 声卡信号则来源于 USB 接口的数字音频信号。

(ii) 输出到音箱的模拟信号受到的电磁干扰更小。 USB 声卡外置于计算机机箱外，声卡和音箱的距离更近，模拟音频信号受到的电磁干扰更小，当然它的音质会更好。

(iii) USB 声卡安装更为方便简单，不必再打开机箱。

说到底， USB 声卡与传统声卡相比没有本质上的区别，只是接口形式有所变化，这种变化只不过是为了顺应 USB 设备的大潮流，性能上并未有突破。uu12288 笔者有一个麦蓝公司的 USB 声卡，外形看起来像一个 USB 鼠标，只是多了一条接到音箱的音频线。不过，要让 USB 声卡正常工作，Windows 里“控制面板” / “多媒体” / “CD 音乐”的“数字音频”必须设置成工作状态，否则，放 CD / VCD 时可能不会发音。打开 USB 声卡的外壳后，可以看到 USB 声卡布局是非常简洁的，只有一颗 D / A(数/模转换)芯片，由大名鼎鼎的 Philips 生产，其它的都是一些电容、电阻。现在有几家公司可以生产这种 D / A 音频芯片，而 Philips 的几款芯片占据了市场的主流，分别是 UDA1321、 UDA1325、UDA1331、UDA1335，其芯片的详细资料可到 http://www_eu2.semiconductors.com/handbook/chapter_1955.htm 网站上看到。 uu12288 笔者认为 USB 声卡由于安装简单，将从传统声卡的市场份额中分得一杯羹，

但传统的声卡技术成熟，接口亦较为丰富，如 MIC，JOYSTICK 接口等，估计还将继续统治声卡市场较长时间。

(c) USB 打印机

uu12288 带有 USB 接口的打印机最大的优点就是可以方便地连接 PC 机、Macintosh 和 iMac，而不必担心接口类型问题。uu12288 EPSON(爱普生)公司是老牌的打印机生产商，USB 技术的推广也不落人后， USB 接口几乎成为该公司打印机的标准配置。EPSON STYLUS PHOTO 750 就是一款带 USB 接口的优秀的喷墨打印机。PHOTO 750 采用 EPSON 独有的微压电打印技术（Micro Piezo），可以实现最小为 6 微微升墨滴（1 微微升=1 10－12 升）打印，这时墨滴的直径仅仅为头发的 1／4，肉眼几乎无法分别。为了保障打印速度，PHOTO 750 使用了智能墨滴变换技术，自动调整墨滴大小，在高浓度区采用大墨滴，同时还采用双向打印和增加喷嘴提高打印速度。 uu12288 由于打印机的速度瓶颈并不在电脑向打印机传输数据部份，所以使用 USB 接口给它带来的速度提升并不大。它的 USB 接口应该说是锦上添花的一项功能。

(d) USB 扫描仪

uu12288 USB 相对于其它传统接口来说，其快速数据传输能力和安装简单的和谐平衡在扫描仪上体现得最为充份。传统 SCSI 接口的扫描仪安装极为麻烦，并且对电脑的要求苛刻，必须带有 SCSI 接口，并行口传输数据又显得实在太慢。USB 接口可以从速度及安装的简易性两方面满足用户要求。现在的主流扫描仪厂商都开始生产 USB 扫描仪，如 HP 的 Scanjet 5200C，Acer 的 Acerscan 310U,TARGA 的 TS12MU。 uu12288 TARGA 推出的这款 USB 扫描仪具有透明外壳，它可扫描的最大尺寸是 A4 尺寸，光学解析度是 6001200dpi。这款机器的主要优点在于以下方面：

(i) USB 接口可以有效地提升数据传输速度。
(ii) 无外置电源设计，由于采用了 CIS 技术，耗电量只有 2W，适合出差人士。
(iii) 采用了超薄超轻设计，绿色透明式外形设计，外形悦目。

uu12288 TS12MU 扫描仪充份体现了未来扫描仪的发展方向。

(e) USB 摄像头

uu12288 目前，摄像头的成像质量还不能同数码相机相提并论，最主要的是，图像传输速度较慢，这也是摄像头迟迟不能得到广泛应用的原因。早期的摄像头是通过并行口或串行口与计算机相联，数据传输速度极慢，最糟糕的是它还得从键盘或鼠标口获取电源（＋5V）（笔者曾经使用过一款手写笔,连接＋5V 电源的复杂程度让我这个动手能力并不差的 DIY 迷也大动肝火）。录像情况自然是惨不忍睹，丢帧现象严重，好像总要对主人的动作来个不合时宜的定格。 uu12288 摄像头厂商都注意到了这个致命的缺点，纷纷投入研发力量改进自己的产品。不过，在 USB 接口成为流行总线以前，他们取得的成绩并不大，“不合时宜的定格”还是在几乎所有的摄像头产品中存在。1998 年以后，Creative、Logitech、NEC 等厂商推出的摄像头产品都是 USB 接口，同传统的其它接口摄像头相比，USB 接口的摄像头优势明显：支持热插拔，传输数据的速度更快，因为传输速率的提高，图像的分辨率也可以做得更高一些，图像质量的提高也成为可能。 uu12288 现在市场上有一款“精通二号”的摄像头，成像质量不错，还带有数码相机拍照功能，具体的性能测试情况可见《电脑报》上的“谁的慧眼更亮”一文。

(f) USB 键盘、鼠标

uu12288 键盘和鼠标算得上是从 USB 得益最少的设备，目前也没有什么必要使用 USB 接口，因为现在的主板都提供了键盘和鼠标专用的 PS／2 口。不过 USB 终会取代 PS／2 口，抢先感受一下新产品也不错。 uu12288 微软的新款人体工程学键盘就采用了 USB 接口，兼容 PC 和 iMac，无论外型、手感都无可挑剔，不过在 DOS 系统下就不能使用了。由于 USB 接口的键盘没有什么出奇的优势，所以也有公司为它增加了一点功能，Cherry 制造的 USB 键盘内建4个USB接口的Hub，可以连接4个低功率的USB设备，如鼠标，飞行摇杆等。 uu12288 罗技应该算是鼠标厂商中的老大了。使用 USB 接口的银貂和旋貂是罗技公司的骄傲，这两款鼠标都提供了一个 USB 转 PS／2 的转接头，主板不支持 USB 的用户也可以使用。深灰色的旋貂显得比较纤细，线条优美，双按键加一个滚轮，左手型用户也适用。采用人体工程学设计的银貂适合右手型用户，多提供了一个按键，位于大拇指下，比普通三键鼠标更合理。在使用了罗技鼠标后，笔者觉得手感、灵敏度、精确度都很好，唯一的不足是银貂可能是按西方人的手型设计，对笔者来说，稍微大了一点。这两款鼠标的所有按键都可以自定义，每个按键有超过 50 种功能可以选择，通过罗技的 MouseWare 控制中心软件可以对鼠标属性进行调整，功能强大。USB 鼠标和 PS／2 鼠标并没有太大的区别，热插拔的机会也比较少，但罗技银貂和旋貂同时提供 USB 和 PS／2 口，比单一接口有更多选择。

(g) USB 显示器

uu12288 现在的 USB 显示器并不像 USB 音箱那样真正通过 USB 总线传输数字信号，它还是要使用显卡，所以它只能算作“带 USB 功能的显示器”。 uu12288 LG795FT Plus 就是这样一款显示器，通过 USB 接口，用户可以使用软件方式调整一些显示器属性，包括屏幕大小、屏幕位置、失真、色彩等，而不必使用 OSD 菜单。使用时，只要选中显示属性的 USB 显示器项，就可以用鼠标拖动滑动条，对大多数原本需要通过 OSD 菜单调整的功能进行设置，非常简单。更为重要的是它还内建了一台 4 口的 USB Hub(集成器)，可以为你节约一笔开支。笔者认为现阶段的 USB 显示器还没有什么实际上的意义，795FT Plus 内建的 Hub 显然比提供的 USB 功能更诱人。

(h) USB Modem

uu12288 简单地说，Modem 的作用就是将电脑的数字信号调制为能在电话线上传输的模拟信号，USB Modem 也是如此，但它的结构要比普通外置式 Modem 简单得多。 uu12288 笔者曾试用过联宝的“小黑猫”56K USB Modem，从外型上看，这款 Modem 通体黑色，非常小巧，只有一只鼠标大小，和常见的外置式 56K Modem 完全两样。“小黑猫” 不需要外接电源，Modem 的两端分别有一个 USB 接口和电话线接口。使用中，USB Modem 的表现还不错，连接速度一直稳定在 52000bps，而且很少出现掉线的情况。但是它的 CPU 占用率要比普通 56K Modem 高一些，特别是在打开网页的时候，CPU 占用率约在 50％左右。这款 USB Modem 没有 Photo 接口，不能接电话，也就是说如果用户不是使用一条专门的电话线上网，就要将电话线头在电话和Modem之间换来换去，没有普通Modem方便。 uu12288 因为USB Modem 的价格和普通 Modem 相比要便宜，性价比较高，应该是目前意义比较大的 USB 外设之一。

(i) USB 软驱

uu12288 3.5 英寸 USB 软驱是国内较少见的外设之一，实用性也并不强，毕竟绝大多数电脑都带有一个内置的软驱。VSTTech 的 3.5 英寸软盘驱动器最初是为 iMac 设计的，但它完全兼容 PC 机的 3.5 英寸盘，只要你的 PC 支持 USB，就可以将它当作普通软驱一样使用。这款软驱使用了和 iMac 一样的半透明外型，非常漂亮，并且传输率提高到了 500KB／s。 uu12288 ZIP 驱动器在国内比较少见，但在欧美比较普及，它使用专门的 100MB 或 250MB ZIP 磁盘，

容量、速度和可靠性都与 3.5 英寸软驱不可同日而语。内置 ZIP 驱动器使用 IDE 接口，显然比 USB 接口更有优势，但它要求机箱中有一个 3.5 英寸驱动器的空位，而多数品牌机并不具备这一条件，所以 ZIP 驱动器以外置式为主，接口包括并行口、Ultra SCSI 和 USB。持续数据传输率分别为 0.8MB／s 、2.4MB／s 和 1.2MB／s，我们可以看到 SCSI 接口的驱动器速度最快，但一块 SCSI 接口卡价值不菲，从性价比来说，USB 接口的 ZIP 驱动器显然是最好的选择。新款 Iomega ZIP USB 驱动器的平均寻道时间为 29ms，容量 100MB，在数据移动方面作用极大。

(j) USB 网卡

uu12288 USB 网卡的功能还比不上我们常用的 PCI 网卡，价格也不便宜，不过它也是 USB 家族的一员。USB 网卡目前只能提供 10Mbps 的局域网连接，而较大型的局域网现在都使用 100Mbps 连接，所以它比较适用于家庭或小型办公室。目前 USB 网卡的价格都很高，甚至比 100Mb PCI 网卡更贵，对国内普通用户来说没有太大的实用价值，但对带有 USB 接口的笔记本电脑用户来说则很有用处。另外，还有一种 USB to USB 对联器，它可以让两台计算机通过 USB 接口互相通信，比串口对联提供更高的连接速度，和 USB 网卡比起来，笔者觉得它是一种更为实用的设备。

(k) USB 转接设备

uu12288 USB 转接设备的种类很多，主要是提供 USB 接口与其它接口的转换功能，包括 USB to ADB、USB to PS／2、USB to SCSI、USB to Serial、USB type A to B 等等。这些转接设备可以使其它非 USB 接口的外设接到 USB 口上使用，这些设备在今天看来还没有太多用处，但有朝一日主板上只提供 USB 接口时，它们就有了用武之地了。现在比较实用的 USB 转接设备主要有两种，其中之一是 USB to PCI 接口卡，它的功能是为不支持 USB 但支持 PCI 的主板（如 Intel 430TX 等）提供两个 USB 接口；或者为只支持两个 USB 接口的主板（如 Intel 440BX 等）提供额外的两个 USB 接口，不过要占用一个 PCI 插槽。Keyspan 的产品采用 OPTi 的芯片，CableMAX 的产品采用 VIA 的芯片，它们都支持 USB 高速和低速连接，无跳线设计，自动设置 IRQ，用户只需要将卡插到 PCI 插槽并安装驱动软件就可以了。另一种实用的 USB 转接设备是 USB 设备延长电缆。普通 USB 设备的联线距离被限制在 1.8 米之内，超出这一距离，USB 设备可能不能正常工作，而通过 USB 设备延长电缆可以把这一距离延长到 5 米。这种电缆内置有 ASIC 芯片，用来缓冲输入和输出的 USB 数字信号，它就像一个单一端口的 USB Hub 一样。这种电缆还可以集联工作，最多支持 5 条电缆串联，将 USB 设备和计算机之间的距离延长到 25 米。

(6) USB 应用技术

通用串行总线(Universal Serial Bus，简称 USB)从诞生后发展到今天，已将近十年。伴随着计算机技术的迅猛发展，USB 协议从 1.1 过渡到 2.0，作为其最重要指标的设备传输速度也从 1.5Mb/s 的低速和 12Mb/s 的全速提高到如今的 480Mb/s 的高速。USB 作为过去几年里计算机和嵌入式系统领域中的热点，推动了计算机外设的飞速发展。毫无疑问的是，USB 已经占领了 PC 和外设的市场；而在未来，USB 又将以 OTG 再次引领计算机外设产业的发展方向，同时也将把计算机和嵌入式领域的学术研究带入更为深入的层次。

图 1　带有 USB 接口的 PC 外设 USB 设备开发技术

1994 年 11 月，以 Intel 为首的 7 家公司推出了 USB 协议规范的第一个草案。自从 1996 年 2 月 USB 版本 1.0 发布后短短几年内，USB 不光成为了 PC 主板上的标准接口，而且成为了所有 PC 外部设备如键盘、鼠标、显示器、打印机、数码相机、扫描仪和游戏手柄等与 PC 相连的标准协议之一，迅速占领了计算机中低速外部设备的市场，大有取代串口和并口之势。图 1 展示了几款带有 USB 接口的 PC 外设、数码家电和通信产品。

首先，我们总结出 USB 设备开发的基本内容。USB 设备作为一个完整的硬件设备，是由硬件和固件两部分来组成的。其中固件中包括了有关系统配置和 CPU 的一些设置模块、USB 协议栈模块等几部分。USB 总线上的信息有两种：一种是差模数据线上的包；另一种则是有特殊定义的数据线的信号，比如复位信号、远程唤醒信号等等。因此，设备的 USB 栈就要能够识别并处理这些不同的信息内容。同时，在上层，这些信息又要被组成各种传输的类型来加以处理。所以，整个协议栈的内容是非常庞大的。

一般来说，USB 设备在硬件上要由 USB 的芯片来实现。这个芯片的作用有：管理和实现 USB 物理层差模信号；提供给连接的端口；电源管理(主要指提供 3.3V 的电源)；以寄存器的形式提供各种端点；提供各种配置和存储寄存器。因此，固件就是以这些硬件资源为基础来实现 USB 的功能。一般的 USB 芯片都会提供几个标准的端点，每个端点都支持单一的总线传输方式。其中端点 0 必须支持控制传输，而其他的端点则可以支持同步传输、批量传输或中断传输中的任意一种传输方式。管理和使用这些端点，就需要通过相应的控制寄存器、状态寄存器、中断寄存器和数据寄存器来实现。其中，控制寄存器用于设置端点的工作模式、启用端点的功能等；状态寄存器用于查询端点的当前状态；中断寄存器则用于设置端点的中断触发和响应功能；数据寄存器则是设备与主机交换数据用的缓冲区。合理和有效地使用这些寄存器，是编好 USB 协议栈的关键。

简而言之，USB 的协议栈以设备端点的使用和管理作为基础和核心。而在端点的这些寄存器中，对中断寄存器的管理尤其重要。也因此，编写 USB 的中断服务程序是整个设备端 USB 固件编写的主要内容。

(7) USB 主机嵌入式化的必要性

随着 USB 应用领域的逐渐扩大，人们希望 USB 能应用在各种计算机领域中，尤其是在移动数据交换等没有 PC 的领域中。 非 PC 应用领域？这正是 USB 一个致命的弱点。USB 的拓扑结构中居于核心地位的是主机(Host)，任何一次 USB 的数据传输都必须由主机来发起和控制，所有的 USB 设备都只能和主机建立连接，任何两个外设之间或是两个主机之间无法直

接通信。而目前，大量的扮演主机角色的是个人电脑(PC)。因此，我们目前所买到和使用的 USB 移动设备，都是 USB 的设备，比如 USB 的移动硬盘、USB 接口的数码相机等等。所有这些设备都只能在 PC 上使用，只能通过 PC 来进行相互的文件和数据交换。没有了 PC，这些设备就“失灵”了(就数据交换的功能而言)。
因此，“如何将 USB 应用到嵌入式领域？如何实现 USB 点对点的通信？”等问题，开始进入了 USB 开发者的讨论议程。正是在这种新的需求之下，USB 主机的嵌入式应用成了 USB 领域新的兴奋点。

(8) PC 上 USB 主机的功能与工作原理

USB 主机完成的主要功能包括以下 5 个方面：检测 USB 设备的连接和断开、管理主机和设备之间的标准控制管道、管理主机和设备之间的数据流、收集设备的状态和统计总线的活动、控制和管理主机控制器与设备之间的电气接口。

剖析 PC 上 USB 主机部分的结构，可以看到，PC 主板上一般都有两个 USB 端口，并由一个 USB 主机接口芯片控制；这个 USB 主机接口芯片又通过 PCI 总线，与 CPU 进行通信；此外，芯片附近还有一些电源管理的部分，用于对 USB 外设进行电源的供给和管理。这是其硬件部分。软件部分，很显然，就是 PC 的操作系统所能够提供的各种驱动程序和应用程序支持，具体来说，包括三部分：USB 主控制器驱动程序，其负责 CPU 与 USB 主机接口芯片的通讯，处理底层 USB 包的发送与接收；USB 核心驱动程序，这部分是 USB 底层与用户程序之间的桥梁，负责解释用户程序中对 USB 的各种操作命令，并解码后发送给底层驱动；USB 用户程序和类协议驱动程序,这部分就是上层的应用层,主要包括操作系统提供给用户的 API、以及用户自己定义的对 USB 设备的各种操作，比如读取 USB 设备某几个特定的数据等等。

(9) 移动 USB——USB OTG 的发展

USB On-The-Go，顾名思义，是 USB 应用在便携式移动设备领域中，因此，我们姑且将其翻译为“便携式 USB”(或者“移动 USB”)，简记成 USB OTG。OTG 1.0 作为 USB 2.0 的补充协议，基本上符合 USB 2.0 规范。但是，有所不同的是符合 USB OTG 的设备完全抛开了 PC，既可以作为主机，也可以作为外设，而与另一个 OTG 设备直接实现点对点(Pear to Pear)通讯。因此，这类 OTG 设备也被称为是双角色设备(Dual-Role Device，简称为 DRD)，并能够根据接入设备的特性和数据传输过程中的情况，自动切换为主机或是设备。需要注意的是，USB OTG 设备保留了作为普通 USB 2.0 设备的功能，可以作为外设直接连接到 PC 的 USB 主机上。

三．实验内容和步骤

1. 板子上电系统正常启动以后，执行 cat /proc/device，显示：

```
Character devices:
  1 mem
  2 pty/m%d
  3 pty/s%d
  4 vc/0
  5 ptmx
  7 vcs
 10 misc
 14 sound
```

```
 29 fb
 90 mtd
108 ppp
128 ptm
136 pts/%d
162 raw
180 usb
204 ttyS%d
205 cua%d
220 gpiotest
254 s3c2410-ts

Block devices:
  1 ramdisk
  7 loop
 30 mtdblock
 43 nbd
```

2. 插入 U 盘，若显示

```
hub.c: USB new device connect on bus1/1, assigned device number 2
Manufacturer: USB
Product: Solid state disk
SerialNumber: 7777777777777777
scsi0 : SCSI emulation for USB Mass Storage devices
Vendor: OTi          Model: Ultra Floppy        Rev: 1.11
Type:   Direct-Access                           ANSI SCSI revision: 02
Attached scsi removable disk sda at scsi0, channel 0, id 0, lun 0
SCSI device sda: 59904 512-byte hdwr sectors (31 MB)
sda: Write Protect is off
Partition check:
/dev/scsi/host0/bus0/target0/lun0: p1
```

则表示系统成功检测到 USB 设备。

3. 执行 cat /proc/device 观察该目录的信息

注意其中 ock devices 增加了检测到的 sd 设备。

```
Block devices:
  1 ramdisk
  7 loop
  8 sd
 30 mtdblock
 43 nbd
 65 sd
 66 sd
```

4. 执行 mount -t vfat /dev/sda1 /mnt/，来将一个 fat32 文件系统的 u 盘 mount 到/mnt 目录上，观察/mnt 目录中数据的变化，通过读写/mnt 目录来读写 U 盘。

四．思考题

1. 如果用户在 mount U 盘的时候，不指定-t vfat 会出现什么现象，试解释该现象出现的原因。

实验二十二： IDE 硬盘接口实验

一．实验目的

通过本实验，使学生了解 IDE 的驱动原理，以及硬盘的管脚，片选，地址空间，读写方式等等。

二．实验原理和说明

1. IDE 接口是通用硬盘接口，它的管角描述见下表：

Description	Host	Dir	Dev	Acronym
Cable select	(See note 1)			CSEL
Chip select 0	→			CS0-
Chip select 1	→			CS1-
Data bus bit 0	↔			DD0
Data bus bit 1	↔			DD1
Data bus bit 2	↔			DD2
Data bus bit 3	↔			DD3
Data bus bit 4	↔			DD4
Data bus bit 5	↔			DD5
Data bus bit 6	↔			DD6
Data bus bit 7	↔			DD7
Data bus bit 8	↔			DD8
Data bus bit 9	↔			DD9
Data bus bit 10	↔			DD10
Data bus bit 11	↔			DD11
Data bus bit 12	↔			DD12
Data bus bit 13	↔			DD13
Data bus bit 14	↔			DD14
Data bus bit 15	↔			DD15
Device active or slave (device 1) present	(See note 1)			DASP-
Device address bit 0	→			DA0
Device address bit 1	→			DA1
Device address bit 2	→			DA2
DMA acknowledge	→			DMACK-
DMA request	←			DMARQ
Interrupt request	←			INTRQ
I/O read	→			DIOR-
I/O ready	←			IORDY

Description	Host Dir Dev	Acronym
I/O write	→	DIOW-
Passed diagnostics	(See note 1)	PDIAG-
Reset	→	RESET-
Note: 1 See signal descriptions for information on source of these signals		

2． Hard Disks（硬盘） Linux 内核分析

硬盘把数据存放在转动的磁碟上，提供了一个更永久存储数据的方式。为了写入数据，微小的磁头把磁碟表面的一个微小的点磁化。通过磁头可以探测指定的微粒是否被磁化，从而可以读出数据。

一个磁盘驱动器由一个或多个磁碟组成，每一个都用相当光滑的玻璃或者陶瓷制成，并覆盖上一层精细的金属氧化物。磁碟放在一个中心轴上面，并按照稳定的速度转动。转动速度根据型号不同从 3000 到 1000RPM（转/每分钟）。磁盘的读/写磁头负责读写数据，每一个磁碟有一对，每一面一个。读/写磁头和磁碟表面并没有物理的接触，而是在一个很薄的空气垫（十万分之一英寸）上面漂浮。读写磁头通过一个驱动器在磁碟表面移动。所有的磁头都粘在一起，一起在磁碟表面移动。

每一个磁碟的表面都分成多个狭窄的同心环，叫做磁道（track）。磁道 0 是最外面的磁道，最高编号的磁道是最接近中心轴的磁道。一个柱面（cylinder）是相同编号磁道的组合。所以每一个磁碟的每一面的所有的第 5 磁道就是第 5 柱面。因为柱面数和磁道数相同，所以磁盘的尺寸常用柱面来描述。每一个磁道分成扇区。一个扇区是可以从硬盘读写的最小数据单元，也就是磁盘的块大小。通常扇区大小是 512 字节，扇区大小通常是在制造磁盘的时候进行格式化的时候设定的。

磁盘通常用它的尺寸（geometry）描述：柱面数、磁头数和扇区数。例如，启动的时候 Linux 这样描述我的 IDE 磁盘：

hdb: Conner Peripherals 540MB - CFS540A, 516MB w/64kB Cache, CHS=1050/16/63

这意味着它由 1050 柱面（磁道），16 头（8 个磁碟）和 63 个扇区/磁道。对于 512 字节的扇区或块大小，磁盘的容量是 529200K 字节。这和磁盘声明的 516M 的存储能力不符合，因为一些扇区用作存储磁盘的分区信息。一些磁盘可以自动找出坏的扇区，对其进行重新索引。

硬盘可以再分为分区。一个分区是分配用于特定目的的一大组扇区。对磁盘分区允许磁盘用于几个操作系统或多个目的。大多数单个磁盘的 Linux 系统都由 3 个分区：一个包含 DOS 文件系统，另一个是 EXT2 文件系统，第三个是交换分区。硬盘的分区用分区表描述，每一个条目用磁头、扇区和柱面号描述分区的起止位置。对于用 fdisk 格式化的 DOS 磁盘，可以有 4 个主磁盘分区。不是分区表所有的 4 个条目都必须用到。Fdisk 支持三种类型的分区：主分区、扩展分区和逻辑分区。扩展分区不是真正的分区，它可以包括任意数目的逻辑分区。发明扩展分区和逻辑分区是为了突破 4 个主分区的限制。下面是一个包括 2 个主分区的磁盘的 的输出：

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders
Units = cylinders of 2048 * 512 bytes
Device Boot Begin Start End Blocks Id System
/dev/sda1 1 1 478 489456 83 Linux native
/dev/sda2 479 479 510 32768 82 Linux swap
Expert command (m for help): p
Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Nr AF Hd Sec Cyl Hd Sec Cyl Start Size ID
1 00 1 1 0 63 32 477 32 978912 83
2 00 0 1 478 63 32 509 978944 65536 82
3 00 0 0 0 0 0 0 0 0 00
4 00 0 0 0 0 0 0 0 0 00

它显示了第一个分区开始于柱面或磁道 0，磁头 1 和扇区 1，直到柱面 477，扇区 32 和磁头 63。因为一个磁道由 32 个扇区和 64 个读写磁头，这个分区的柱面都是完全包括的。Fdisk 缺省把分区对齐在柱面的边界。它从最外面的柱面（0）开始向内，朝向中心轴，扩展 478 个柱面。第 2 个分区，交换分区，开始于下一个柱面（478）并扩展到磁盘最里面的柱面。

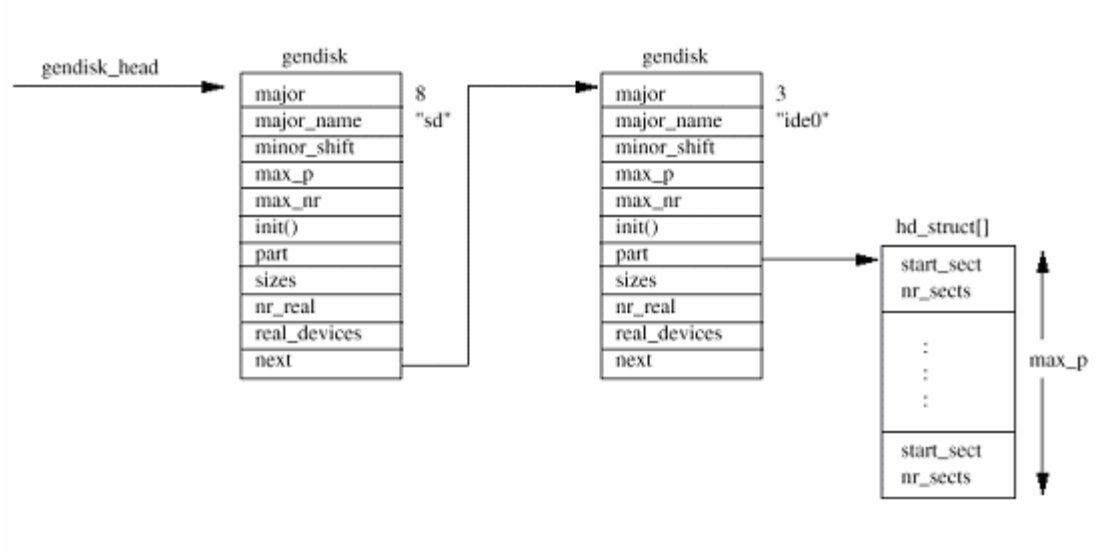


图 1 已经连接的硬盘设备列表

在初始化的时候 Linux 映射系统中的硬盘的拓扑结构。它找出系统中有多少个硬盘以及硬盘的类型。Linux 还找出每一个磁盘如何分区。这些都是由 gendisk_head 指针列表指向的一组 gendisk 数据结构的列表表达。对于每一个磁盘子系统，例如 IDE，初始化的时候生成 gendisk 数据结构表示它找到的磁盘。这个过程和它登记的文件操作和在 blk_dev 数据结构中增加它的条目发生在同一时间。每一个 gendisk 数据结构都由一个唯一的主设备号，和块特殊设备的相同。例如，SCSI 磁盘子系统会创建一个独立的 gendisk 条目（“sd”），主设备号是 8（所有 SCSI 磁盘设备的主设备号）。图 8.3 显示了两个 gendisk 条目，第一个是 SCSI 磁盘子系统，第二个是 IDE 磁盘控制器。这里是 ide0，主 IDE 控制器。

虽然磁盘子系统在初始化的时候会建立相应的 gendisk 条目，Linux 只是在进行分区检查的时候才用到。每一个磁盘子系统必须维护自己的数据结构，让它自己可以把设备的主设备号和次设备号映射到物理磁盘的分区上。不管什么时候读写块设备，不管是通过 buffer cache 或者文件操作，核心都根据它在块特殊设备文件（例如/dev/sda2）中找到的主设备号和次设备号把操作定向到合适的设备。是每一个设备驱动程序或子系统把次设备号映射到真正的物理设备上。

3． IDE Disks（IDE 磁盘）

今天 Linux 系统中最常用的磁盘是 IDE 磁盘（Integrated Disk Electronic）。IDE 和 SCSI 一样是一个磁盘接口而不是一个 I/O 总线。每一个 IDE 控制器可以支持最多 2 个磁盘，一个是 master，另一个是 slave。Master 和 slave 通常用磁盘上的跳线设置。系统中的第一个 IDE

控制器叫做主 IDE 控制器，下一个叫从属控制器等等。IDE 可以从/向磁盘进行 3.3M/秒的传输，IDE 磁盘的最大尺寸是 538M 字节。扩展 IDE 或 EIDE 把最大磁盘尺寸增加到 8.6G 字节，数据传输速率高达 16.6M/秒。IDE 和 EIDE 磁盘比 SCSI 磁盘便宜，大多数现代 PC 都有一个或更多的主板上的 IDE 控制器。

Linux 按照它发现的控制器的顺序命名 IDE 磁盘。主控制器上的主磁盘是/dev/had，slave 磁盘是/dev/hdb。/dev/hdc 是次 IDE 控制器上的 master 磁盘。IDE 子系统向 Linux 登记 IDE<> 控制器而不是磁盘。主 IDE 控制器的主标识符是 3，次 IDE 控制器的标识符是 22。这意味着如果一个系统有两个 IDE 控制器，那么在 blk_dev 和 blkdevs 向量表中在索引 3 和 22 会有 IDE 子系统的条目。IDE 磁盘的块特殊文件反映了这种编号：磁盘/dev/had 和/dev/hdb，都连接在主 IDE 控制器上，主设备号都是 3。核心使用主设备标识符作为索引，对于这些块特殊文件的 IDE 子系统进行的所有的文件或者 buffer cache 操作都被定向到相应的 IDE 子系统。当执行一个请求的时候，IDE 子系统负责判断这个请求是针对哪一个 IDE 磁盘。为此，IDE 子系统使用设备特殊文件中的次设备号，这些信息允许它把请求定向到正确的磁盘的正确的分区。/dev/hdb，主 IDE 控制器上的 slave IDE 磁盘的设备标识符是（3，64）。它的第一个分区（/dev/hdb1）的设备标识符是（3，65）。

4． Initializing the IDE Subsystem（初始化 IDE 子系统）

IBM PC 的大部分历史中都有 IDE 磁盘。这期间这些设备的接口发生了变化。这让 IDE 子系统的初始化过程比它第一次出现的时候更加复杂。

Linux 可以支持的最大 IDE 控制器数目是 4。每一个控制器都用一个 ide_hwifs 向量表中的一个 ide_hwif_t 数据结构表示。每一个 ide_hwif_t 数据结构包含两个 ide_drive_t 数据结构，分别表示可能支持的 master 和 slave IDE 驱动器。在 IDE 子系统初始化期间，Linux 首先查看在系统的 CMOS 内存中记录的磁盘的信息。这种用电池做后备的内存在 PC 关机的时候不会丢失它的内容。这个 CMOS 内存实际上在系统的实时时钟设备里面，不管你的 PC 开或者关，它都在运行。CMOS 内存的位置由系统的 BIOS 设置，同时告诉 Linux 系统中找到了什么 IDE 控制器和驱动器。Linux 从 BIOS 中获取找到的磁盘的尺寸（geometry），用这些信息设置这个驱动器的 ide_hwif_t 的数据结构。大多数现代 PC 使用 PCI 芯片组例如 Intel 的 82430 VX 芯片组，包括了一个 PCI EIDE 控制器。IDE 子系统使用 PCI BIOS 回调（callback）定位系统中的 PCI （E）IDE 控制器。然后调用这些芯片组的询问例程。

一旦发现一个 IDE 接口或者控制器，就设置它的 ide_hwif_t 来反映这个控制器和上面的磁盘。操作过程中 IDE 驱动程序向 I/O 内存空间的 IDE 命令寄存器写命令。主 IDE 控制器的控制和状态寄存器的缺省的 I/O 地址是 0x1F0-0x1F7。这些地址是早期的 IBM PC 约定下来的。IDE 驱动程序向 Linux 的 buffer cache 和 VFS 登记每一个控制器，分别把它加到 blk_dev 和 blkdevs 向量表中。IDE 驱动程序也请求控制适当的中断。同样，这些中断也有约定，主 IDE 控制器是 14，次 IDE 控制器是 15。但是，象所有的 IDE 细节一样，这些都可以用核心的命令行选项改变。IDE 驱动程序在启动的时候也为每一个找到的 IDE 控制器在 gendisk 列表中增加一个 gendisk 条目。这个列表稍后用于查看启动时找到的所有的硬盘的分区表。分区检查代码明白每一个 IDE 控制器可以控制两个 IDE 磁盘。

5． SCSI Disks（SCSI 磁盘）

SCSI（Small Computer System Interface 小型计算机系统接口）总线是一种有效的点对点的数据总线，每个总线支持多达 8 个设备，每个主机可以有一或者多个。每一个设备都必须由一个唯一的标识符，通常用磁盘上的跳线设置。数据可以在总线上的任意两个设备之间同步或者异步传输，可以用 32 位宽的数据传输，速度可能高达 40M/秒。SCSI 总线可以在设备

之间传输数据和状态信息，发起者（initiator）和目标（target）之间的事务会涉及多达 8 个不同的阶段。你可以通过 SCSI 总线上的 5 种信号判断出当前的阶段。这 8 个阶段是：

(1) BUS FREE 没有设备有总线的控制权，当前没有发生任何事务；
(2) ARBITRATION （仲裁）一个 SCSI 设备试图得到 SCSI 总线的控制权，它在地址管脚上声明（assert）它的 SCSI 标识符。最高编号的 SCSI 标识符成功；
(3) SELECTION 一个设备通过仲裁成功地得到了 SCSI 总线的控制权，现在它必须向它要发送命令的 SCSI 目标发送信号。它在地址管脚上声明目标的 SCSI 标识符；
(4) RESELECTION SCSI 设备在处理请求的过程中可能断线，目标会重新选择发起者。并非所有的 SCSI 设备都支持这一阶段；
(5) COMMAND 6、10 或者 12 字节的命令可以从发起者发送到目标；
(6) DATA IN，DATA OUT 在这一阶段，数据在发起者和目标之间传输；
(7) STATUS 在完成了所有的命令，进入这一阶段。允许目标向发起者发送一个状态字节，表示成功或失败；
(8) MESSAGE IN，MESSAGE OUT 在发起者和目标之间传递的附加信息。

Linux SCSI 子系统由两个基本元素组成，每一个都用数据结构表示：

Host 一个 SCSI host 是一个物理的硬件，一个 SCSI 控制器。NCR810 PCI SCSI 控制器是一个 SCSI host 的例子。如果一个 Linux 系统有多于一个同类型的 SCSI 控制器，每一个实例都分别用一个 SCSI host 表示。这意味着一个 SCSI 设备驱动程序可能控制多于一个控制器的实例。SCSI host 通常总是 SCSI 命令的发起者（initiator）。

Device SCSI 设备通常是磁盘，但是 SCSI 标准支持多种类型：磁带、CD-ROM 和通用（generic）SCSI 设备。SCSI 设备通常都是 SCSI 命令的目标。这些设备必须不同地对待。例如可移动介质如 CD-ROM 或磁带，Linux 需要探测介质是否取出。不同的磁盘类型有不同的主设备编号，允许 Linux 把块设备请求定向到合适的 SCSI 类型。

6. Initializing the SCSI Subsystem（初始化 SCSI 子系统）

初始化 SCSI 子系统相当复杂，反映出 SCSI 总线和设备的动态的实质。Linux 在启动的时候初始化 SCSI 子系统：它查找系统中的 SCSI 控制器（SCSI host），并探测每一个 SCSI 总线，查找每一个设备。然后初始化这些设备，让 Linux 核心的其余部分可以通过普通的文件和 buffer cache 块设备操作访问它们。这个初始化过程有四个阶段：

首先，Linux 找出核心建立的时候建立到核心的哪一个 SCSI host 适配器或控制器有可以控制的硬件。每一个内建的 SCSI host 在 buildin_scsi_hosts 向量表中都有一个 Scsi_Host_Template 的条目。这个 Scsi_Host_Template 数据结构包括例程的指针，这些例程可以执行和 SCSI host 相关的动作例如探测这个 SCSI host 上粘附了什么 SCSI 设备。这些例程在 SCSI 子系统配置期间被调用，是支持这种 host 类型的 SCSI 设备驱动程序的一部分。每一个查到的 SCSI 控制器（有真实的 SCSI 设备粘附），它的 Scsi_Host_Template 数据结构都加到 scsi_hosts 列表中，表示有效的 SCSI host。每一个探测到的 host 类型的每一个实例都用 scsi_hostlist 列表中的一个 Scsi_Host 数据结构表示。例如一个系统有两个 NCR810 PCI SCSI 控制器，在这个列表中会有两个 Scsi_Host 条目，每一个控制器一个。每一个 Scsi_Host 指向的 Scsi_Host_Template 表示它的设备驱动程序。

现在每一个 SCSI host 都找到了，SCSI 子系统必须找到每一个 host 总线上的所有的 SCSI 设备。SCSI 设备编号从 0 到 7，每一个设备编号或者 SCSI 标识符在它所粘附的 SCSI 总线上都是唯一的。SCSI 标识符通常用设备上的跳线设置。SCSI 初始化代码通过向每一个设备发送 TEST_UNIT_READY 命令来查找一个 SCSI 总线上的每一个 SCSI 设备。当一个设备回应，再向它发送一个 ENQUIRY 命令来完成它的判别。这向 Linux 给出 Vendor 的名称和设备的型号

和修订号。SCSI 命令用一个 Scsi_Cmnd 数据结构来表示，这些命令通过调用这个 SCSI host 的 Scsi_Host_Template 数据结构中的设备驱动程序例程传递给设备驱动程序。每一个找到的 SCSI 设备用一个 Scsi_Device 数据结构表示，每一个都指向它的父 Scsi_Host。所有的 Scsi_Device 数据结构都加到 scsi_devices 列表中。图 8.4 显示了主要的数据结构和其他数据结构的关系。

有四种 SCSI 设备类型：磁盘、磁带、CD 和通用（generic）。每一种 SCSI 类型都分别向核心登记，有不同的主块设备类型。但是，它们只有在一个或多个给定的 SCSI 设备类型的设备找到的时候才登记自己。每一个 SCSI 类型，例如 SCSI 磁盘，维护它自己的设备表。它用这些表把核心的块操作（文件或 buffer cache）定向到正确的设备驱动程序或 SCSI host。每一个 SCSI 类型都用一个 Scsi_Type_Template 数据结构表示。它包括这种类型的 SCSI 设备的信息和执行多种任务的例程的地址。SCSI 子系统使用这些模板调用每一种 SCSI 设备类型的 SCSI 类型例程。换句话说，如果 SCSI 子系统希望粘附一个 SCSI 磁盘设备，它会调用 SCSI 磁盘类型的例程。如果探测到某类型的一个或多个 SCSI 设备，它的 Scsi_Type_Templates 的数据结构就加到了 scsi_devicelist 列表中。

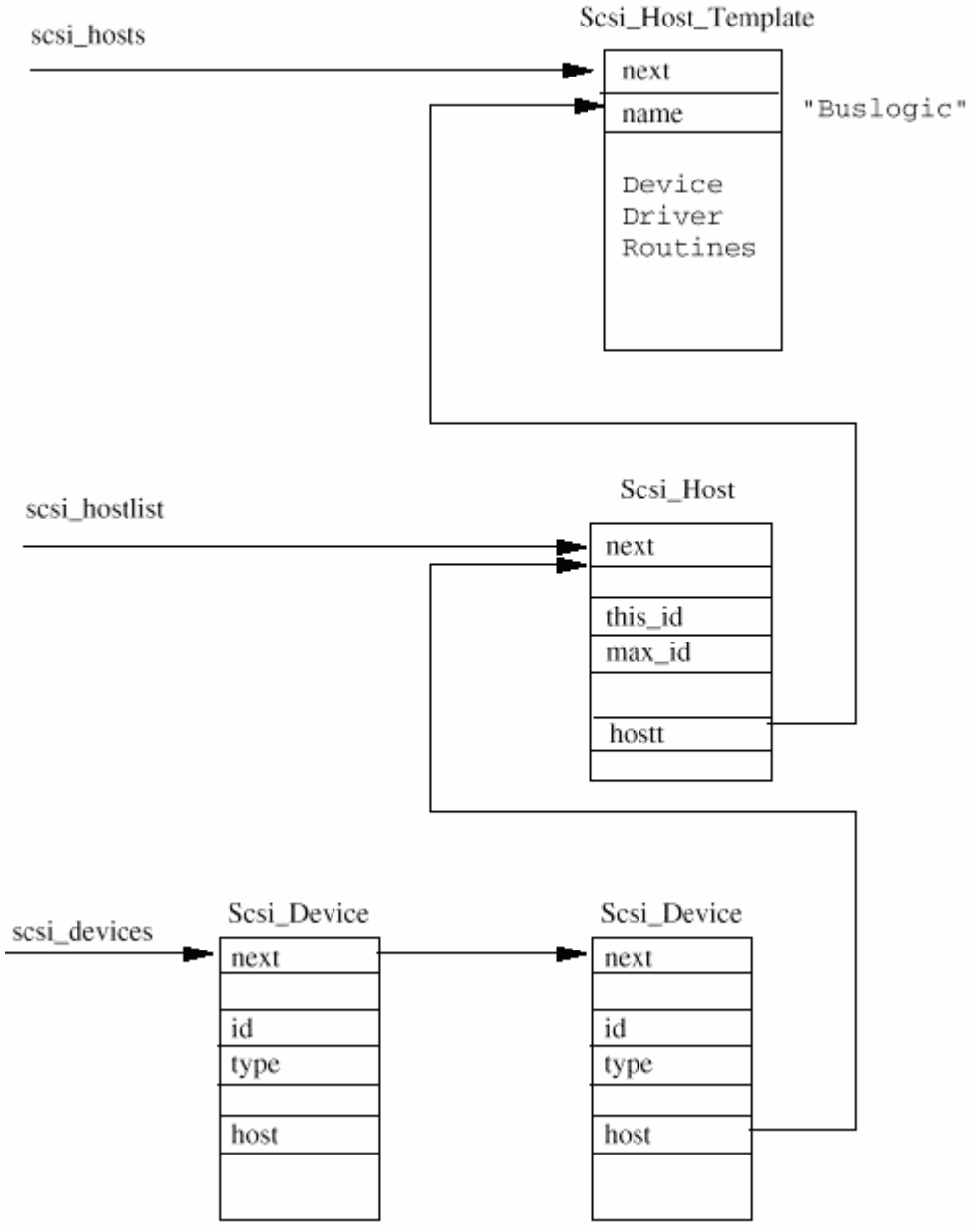


图 2： scsi 数据结构

SCSI 子系统初始化的最后阶段是调用每一个登记的 Scsi_Device_Template 的完成函数。对于 SCSI 磁盘类型让所有的 SCSI 磁盘转动起来并记录它们的磁盘尺寸。它也把表示所有 SCSI 磁盘的 gendisk 数据结构增脚的磁盘的链接列表中，如图 2。

Delivering Block Device Requests（传递块设备请求）

一旦 Linux 初始化了 SCSI 子系统，就可以使用 SCSI 设备了。每一个有效的 SCSI 设备类型都在核心中登记自己，所以 Linux 可以把块设备请求定向到它那里。这些请求可能是通过 blk_dev 的 buffer cache 请求或者是通过 blkdevs 向量表中都有一个 EXT2 文件系统分区的 SCSI 磁盘驱动器为例，当它的 EXT2 分区安装上的时候核心的缓冲区请求是如何定向到正确

的 SCSI 磁盘呢？

每一个向/从一个 SCSI 磁盘分区读/写一块数据的请求都会在 blk_dev 向量表中这个 SCSI 磁盘的 current_request 列表中加入一个新的 request 数据结构。如果这个 request 列表正在处理，那么 buffer cache 不需要做什么。否则它必须让 SCSI 磁盘子系统处理它的请求队列。系统中的每一个 SCSI 磁盘用一个 Scsi_Disk 数据结构表示。它们保存在 rscsi_disks 向量表中，用 SCSI 磁盘分区的次设备号的一部分作为索引。例如，/dev/sdb1 主设备号 8，次设备号 17，它的所以是 1。每一个 Scsi_Disk 的数据结构包括一个指向表示这个设备的 Scsi_Device 数据结构的指针。Scsi_Device 又指向一个“拥有它”的 Scsi_Host 数据结构。Buffer cache 中的 request 数据结构转换成为描述需要发送到 SCSI 设备的 SCSI 命令的 Scsi_Cmd 数据结构中，并在表示这个设备的 Scsi_Host 数据结构中排队。一旦适当的数据块读/写之后，会由各自的 SCSI 设备驱动程序处理。

7. 硬盘驱动代码在 s3c2410 上得移植：

简介：在 RUIJIARM9-EDU 上增加 IDE 接口的驱动程序，占用的硬件资源是片选 nGCS2、中断 4 、外部中断 EINT6、IDE 接口的地址空间的基地址是 0x10000000，16 位的 I/O 读写方式（寄存器读写时只用到低 8 位）。软件使用的是 linux（Ver：2.4.0）系统自带的标准 IDE 驱动程序。

在 RUIJIARM9-EDU 系统中加入 IDE 接口驱动程序的简单说明：

(1) 硬件设计是通过可编程逻辑芯片直接把 IDE 接口的各信号线连接到系统总线上， IDE 寄存器读写和 IDE 数据读写都采用 16 位的读写方式（由片选决定的，但对寄存器的读写只有低 8 位数据是有效的），IDE 各寄存器端口地址由下表给出。

1000 0038	切换状态/设备控制
1000 0040	读写数据
1000 0044	查错误/写特征
1000 0048	扇区计数
1000 004C	扇区号
1000 0050	柱面低
1000 0054	柱面高
1000 0058	磁头号
1000 005C	读状态/写命令

RJARM-2410 硬盘各寄存器及地址列表

(2) 因为是标准的 IDE 驱动程序，所以我们的主要工作是修改 IDE 驱动的 I/O 地址值和 I/O 地址在 ARM9 的 MMU 下的映射以及读写方式等。

(3) 在 RUIJIARM9-EDU 系统中加入 IDE 接口驱动程序的具体过程：

(a) 在 linux.2.4.19 pre /drivers/ide 目录下添加 cpci405ide.c 文件，该文件提供了地址映射（ioremap）和 IDE 寄存器端口地址的设置（ide_setup_ports 函数），且在 IDE 驱动的核心文件 ide.c 中增加对 cpci405ide.c 文件的调用，从而我们只要把 cpci405ide.c 文件加入到 IDE 驱动中编译就可以了。

(b) 在驱动中加入对 cpci405ide.c 文件的编译：IDE 驱动目录下的 MakeFile 文件中加入对 cpci405ide.c 的支持“ide-obj-$(CONFIG_BLK_DEV_CPCI405_IDE)　+= cpci405ide.o”，修改 IDE 驱动目录下的 Config.in 文件，在“bool '　Generic 4 drives/port support' CONFIG_BLK_DEV_4DRIVES”此行之前加入 “bool '　CPCI-405 IDE interface support' CONFIG_BLK_DEV_CPCI405_IDE” 这行语句，

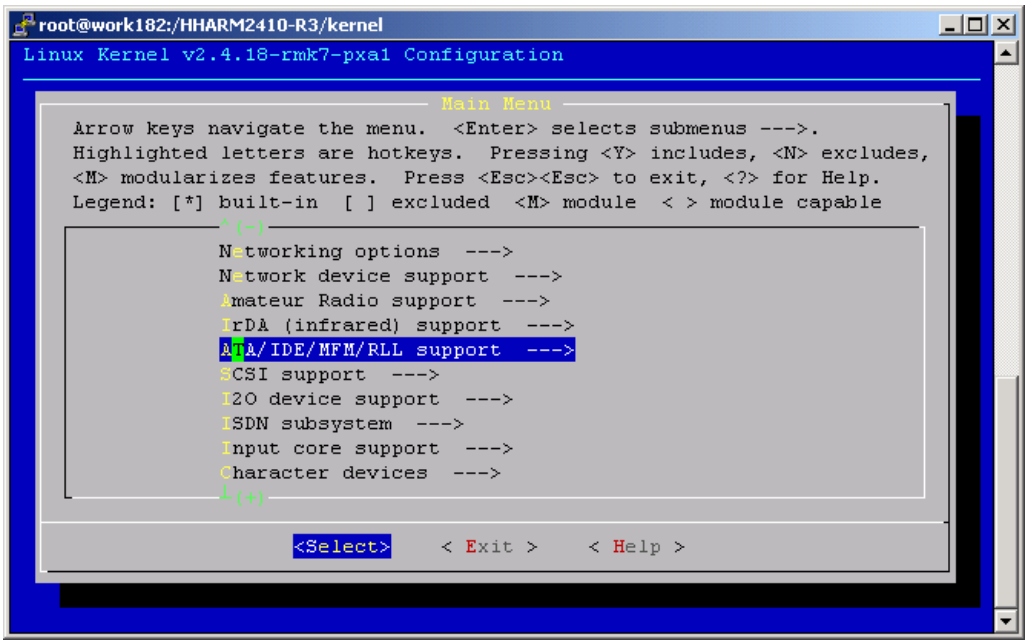
在”if ["$CONFIG_Q40" = "y"]; then”此行之前加上

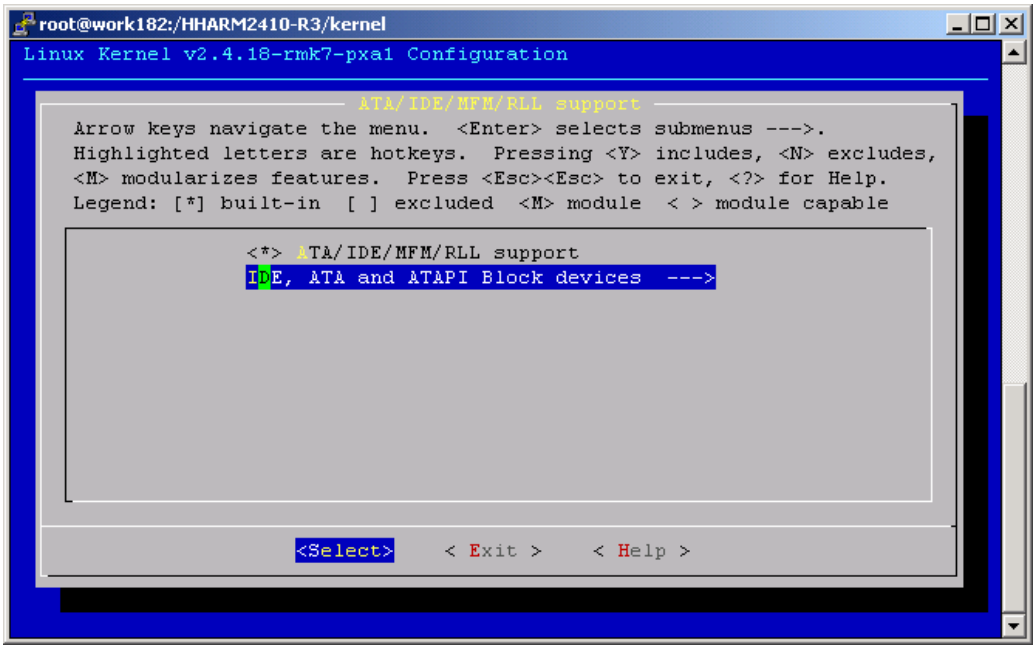
if ["$CONFIG_CPCI405" = "y"];

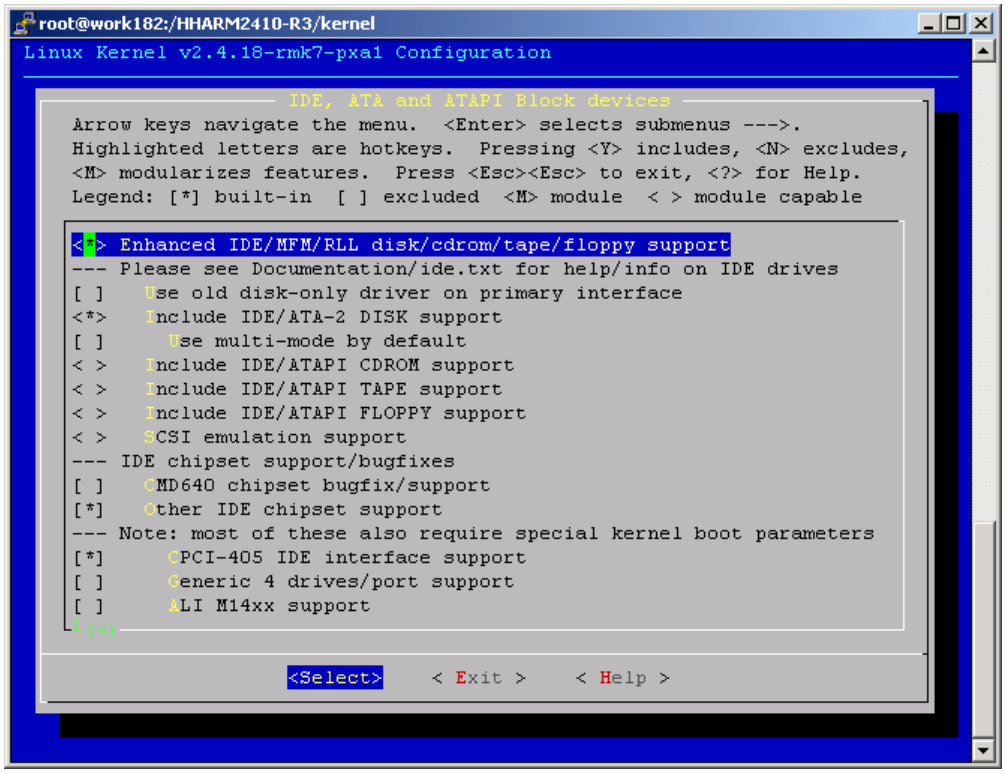
then dep_bool '　CPCI-405 IDE interface support' CONFIG_BLK_DEV_CPCI405_IDE $CONFIG_CPCI405

fi

三行语句，这样在配置菜单中就可以选择了。







(c) 修改 linux.2.4.19 pre /drivers/ide/ cpci405ide.c 文件中 IDE 端口寄存器基地址和各寄存器偏移量的定义，修改后的值如下：

```
#define CPCI405_HD_BASE 0x10000000
#define CPCI405_HD_DATA          0x40
#define CPCI405_HD_ERROR         0x44
#define CPCI405_HD_NSECTOR       0x48
#define CPCI405_HD_SECTOR        0x4C
#define CPCI405_HD_LCYL          0x50
#define CPCI405_HD_HCYL          0x54
#define CPCI405_HD_SELECT        0x58
#define CPCI405_HD_STATUS        0x5C
#define CPCI405_HD_CONTROL       0x38
#define CPCI405_IRQ_IDE          4    //EINT6
```

(d) 在 linux.2.4.19 pre /drivers/ide/ cpci405ide.c 文件的函数 void __init cpci405ide_init(void)中加入片选代码：

```
BWSCON &= 0xFFFFF0FF;
WSCON |= 0x500;       //DW2=16 位、WS2=1、ST2=0
BANKCON2 = 0x550;
```

加入中断使能代码：

```
set_external_irq(IRQ_EINT6,EXT_FALLING_EDGE, GPIO_PULLUP_DIS);
disable_irq(IRQ_EINT6);
enable_irq(IRQ_EINT6);
```

(e) 在 linux.2.4.19 pre /drivers/ide/ ide-probe.c 文件的函数 void __init hwif_check_regions (void) 中加入区域检测代码：

```
//RJTECH
{
    extern int cpci405ide_check_region(ide_ioreg_t start, unsigned int len);
    #undef ide_check_region
    #define  ide_check_region(from,extent)          cpci405ide_check_region((from), (extent))
}
```

(f) 在 linux.2.4.19 pre /drivers/ide/ ide.c 文件的函数 void ide_intr (int irq, void *dev_id, struct pt_regs *regs)增加清中断状态寄存器的代码：

```
SRCPND &= 0xFFFFFFEF;
INTPND = INTPND;
EINTPEND &= 0xFFFFFFBF;
```

(g) 修改 linux.2.4.19 pre /include/linux/ide.h 文件，该文件定义了读写 IDE 寄存器的宏 IN_BYTE 和 OUT_BYTE，但不符合要求。在这里我们的解决方法是把该文件中的宏定义给注释掉，而是在 IDE 的驱动文件 linux.2.4.19pre /drivers/ide/ide.c 中定义，下面给出在 linux.2.4.19pre /drivers/ide/ide.c 文件中读写 IDE 寄存器的具体定义：

```
void OUT_BYTE(n,p)
{
    *(volatile unsigned short *)(p)=(unsigned short)n;
}

unsigned char IN_BYTE(p)
{
    return (((*(volatile unsigned short *)(p))) &0x00ff);
}
```

(h) 修改 linux/drivers/ide/ide-probe.c 文件，该文件中的 do_probe 函数在检测 IDE 设备时，因为板子从加电启动到驱动启动的过程间隔的时间非常短，所以可能 IDE 设备还没有准备好，这时候检测 IDE 设备时将会得到不正确的设备状态，当然也就导致检测不到的可能。故在检测 IDE 不成功时不要立即退出，要增加检测的次数。在 do_probe 函数中所增加的检测代码如下：

在 SELECT_DRIVE(hwif,drive); ide_delay_50ms();这两条语句之后增加：

```
if (IN_BYTE(IDE_SELECT_REG) != drive->select.all)
{
    unsigned char select,i;
    printk("IDE: waiting for drives to settle...\n");
    i=0;
    while(1)
    {
        SELECT_DRIVE(hwif,drive);
```

```
            ide_delay_50ms();
            select = IN_BYTE(IDE_SELECT_REG);
            if (select == drive->select.all || i >=18)     //检测次数最多 18 次
                break;
            else
                i++;
            ide_delay_50ms();
        }
    }
```

(i) 把 IDE 驱动加入到 linux 内核，在 make menuconfig 中选中“ATA/IDE/MFM/RLL support”，然后选择“IDE, ATA and ATAPI Block devices”下的如下选项：

(i) Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support (NEW)

(ii) Include IDE/ATA-2 DISK support (NEW)

(iii) Other IDE chipset support

(iv) CPCI-405 IDE interface support

8. IDE 驱动启动的过程简要分析：整个驱动加载的整体过程是由 ide.c 文件来控制的。

首先 ide.c 初始化一些必要的数据结构如 IDE 端口寄存器的地址，但是如果不是标准的情况如 MMU 下需要 ioremap 这时我们可通过外扩的驱动文件来实现如以上所提到的 cpci405ide.c 文件（IDE 驱动的结构化使得我们很容易在驱动的基础上增加自己的模块）。

其次 ide.c 调用 ide-probe.c 文件来检测和读取磁盘信息以及显示磁盘信息。检测的具体过程是：驱动向磁头号寄存器中写入一个特定的值 0xa0，然后读出磁头号寄存器中的值和 0xa0 比较，若不相等表示没有 IDE 设备则驱动结束，若相等则会执行读命令从磁盘中读取 512 个字节的磁盘 ID 信息。ID 信息的显示分两步：第一步是显示磁盘的厂商和型号等，在 ide-probe.c 文件中完成；第二步是显示磁盘的容量、柱面数、磁头数、扇区数，在 ide-disk.c 文件中完成。

申请中断，挂接 IDE 设备的中断处理程序。

磁盘分区检测：由文件系统通过调用 IDE 驱动的中断服务程序读取 IDE 设备的分区信息来识别的。

硬盘读写速度的提升：用硬盘数据线实现数据的高低位交换来替代软件上的交换，减小 IDE 片选的时间。

9. IDE 硬盘驱动代码分析

入口函数是 ide.c 文件包含的 int __init ide_init (void)函数。

```
int __init ide_init (void)
{
    static char banner_printed;
    int i;
    if (!banner_printed) {                                          //打印信息；
        ide_devfs_handle = devfs_mk_dir (NULL, "ide", NULL);
        //调用 devfs_mk_dir 建立 ide 设备文件目录，赋值给全局变量 ide_devfs_handle。
        system_bus_speed = ide_system_bus_speed();                  //设置总线速度
        banner_printed = 1;
    }
```

```
    init_ide_data ();                       //初始化 ide 数据结构
    initializing = 1;
    ide_init_builtin_drivers();             //初始化 ide 驱动
    initializing = 0;

    for (i = 0; i < MAX_HWIFS; ++i) {
        ide_hwif_t  *hwif = &ide_hwifs[i];
        if (hwif->present)
            ide_geninit(hwif);              //初始化分区结构
    }
    return 0;
}

#define MAGIC_COOKIE 0x12345678
static void __init init_ide_data (void)         //ide.c 文件包含
{
    unsigned int index;
    static unsigned long magic_cookie = MAGIC_COOKIE;
    if (magic_cookie != MAGIC_COOKIE)
        return;         /* already initialized */
    magic_cookie = 0;

    /* Initialise all interface structures */
    // MAX_HWIFS 值为 4，表示 cpu 可以有 4 个 ide 接口，每个 ide 接口可以拥有 2 个 ide
设备
    for (index = 0; index < MAX_HWIFS; ++index)
        init_hwif_data(index);

    /* Add default hw interfaces */
    ide_init_default_hwifs();
    idebus_parameter = 0;
    system_bus_speed = 0;
}

static void init_hwif_data (unsigned int index)     //ide.c 文件包含
{
    unsigned int unit;
    ide_hwif_t *hwif = &ide_hwifs[index];

    /* bulk initialize hwif & drive info with zeros */
    memset(hwif, 0, sizeof(ide_hwif_t));

    /* fill in any non-zero initial values */
```

```
        hwif->index         = index;
        hwif->noprobe       = 1;                        //还没有 probe
        hwif->major    = ide_hwif_to_major[index];
        hwif->name[0]       = 'i';
        hwif->name[1]       = 'd';
        hwif->name[2]       = 'e';
        hwif->name[3]       = '0' + index;              // 该 ide 接口的命名，依次为
ide0,ide1,ide2,ide3
        hwif->bus_state = BUSSTATE_ON;
        for (unit = 0; unit < MAX_DRIVES; ++unit) {
            // MAX_DRIVES 为 2 ，每个接口 2 个 ide 设备
            ide_drive_t *drive = &hwif->drives[unit];

            drive->media              = ide_disk;       //ide 设备为硬盘
            drive->select.all         = (unit<<4)|0xa0; //参看 select 数据结构
            drive->hwif               = hwif;           //指向本设备所属的 ide 接口
            drive->ctl          = 0x08;                 //控制命令
            drive->ready_stat         = READY_STAT;
            drive->bad_wstat          = BAD_W_STAT;
            drive->special.b.recalibrate    = 1;
            drive->special.b.set_geometry       = 1;
            drive->name[0]                  = 'h';
            drive->name[1]                  = 'd';
            drive->name[2]                  = 'a' + (index * MAX_DRIVES) + unit;
                                              //设备命名，依次为 had,hdb,…hdh
            drive->max_failures             = IDE_DEFAULT_MAX_FAILURES;
                                              //设备命令失败最大次数
            init_waitqueue_head(&drive->wqueue);
                                              //初始化设备等待队列
        }
}

void __init ide_init_builtin_drivers (void)         // ide.c 文件包含
{
        /*Probe for special PCI and other "known" interface chipsets*/
        probe_for_hwifs ();       //初始化 ide 接口

#ifdef CONFIG_BLK_DEV_IDE         //为 1
#if defined(__mc68000__) || defined(CONFIG_APUS)      //为 0
        if (ide_hwifs[0].io_ports[IDE_DATA_OFFSET]) {
            ide_get_lock(&ide_lock, NULL, NULL);    /* for atari only */
            disable_irq(ide_hwifs[0].irq); /* disable_irq_nosync ?? */
        }
#endif /* __mc68000__ || CONFIG_APUS */
```

```
    (void) ideprobe_init();                //

#if defined(__mc68000__) || defined(CONFIG_APUS)      //为 0
    if (ide_hwifs[0].io_ports[IDE_DATA_OFFSET]) {
        enable_irq(ide_hwifs[0].irq);
        ide_release_lock(&ide_lock);/* for atari only */
    }
#endif /* __mc68000__ || CONFIG_APUS */
#endif /* CONFIG_BLK_DEV_IDE */

#ifdef CONFIG_PROC_FS
    proc_ide_create();                     //建立 ide 的 proc 文件系统
#endif

    /*Attempt to match drivers for the available drives*/
#ifdef CONFIG_BLK_DEV_IDEDISK//为 1
    (void) idedisk_init();                 //初始化 ide 硬盘
#endif /* CONFIG_BLK_DEV_IDEDISK */
#ifdef CONFIG_BLK_DEV_IDECD
    (void) ide_cdrom_init();
#endif /* CONFIG_BLK_DEV_IDECD */
#ifdef CONFIG_BLK_DEV_IDETAPE
    (void) idetape_init();
#endif /* CONFIG_BLK_DEV_IDETAPE */
#ifdef CONFIG_BLK_DEV_IDEFLOPPY
    (void) idefloppy_init();
#endif /* CONFIG_BLK_DEV_IDEFLOPPY */
#ifdef CONFIG_BLK_DEV_IDESCSI
#ifdef CONFIG_SCSI
    (void) idescsi_init();
#else
#warning ide scsi-emulation selected but no SCSI-subsystem in kernel
#endif
#endif /* CONFIG_BLK_DEV_IDESCSI */
}

static void __init probe_for_hwifs (void)      // ide.c 文件包含
{
    根据配置，选择相应的
}
void __init cpci405ide_init(void)   // cpci405ide.c 文件包含
{
    hw_regs_t hw;
```

```
    int index = -1;
    unsigned long vaddr;

    //这是按照 s3c2410 的 cpu 手册设置 ide 片选
    BWSCON &= 0xFFFFF0FF;
    BWSCON |= 0x500;
    BANKCON2 = 0x550;

    //按照 s3c2410 的 cpu 手册设置 ide 中断信号脚
    set_external_irq(IRQ_EINT6,EXT_FALLING_EDGE, GPIO_PULLUP_DIS);
    disable_irq(IRQ_EINT6);
    enable_irq(IRQ_EINT6);

    cpci405_ide_base_mapped = ioremap(CPCI405_HD_BASE, 0x200);
        // CPCI405_HD_BASE 是按照 s3c2410 的 cpu 手册设置的 ide 接口的基地址

    ide_setup_ports(&hw, (ide_ioreg_t)cpci405_ide_base_mapped,
                                cpci405ide_offsets, 0, 0, NULL, CPCI405_IRQ_IDE);
        // cpci405ide_offsets 就是各个寄存器对于基址的 offset 值的数组

    index = ide_register_hw(&hw, NULL);
    printk("ioremap:%x\n",cpci405_ide_base_mapped);
    if (index != -1)
        printk("ide%x: CPCI405 IDE interface\n", index);
}

static int do_probe (ide_drive_t *drive, byte cmd)// ide-probe.c 文件包含
{
    int rc;
    ide_hwif_t *hwif = HWIF(drive);
    if (drive->present) {    /* avoid waiting for inappropriate probes */
        if ((drive->media != ide_disk) && (cmd == WIN_IDENTIFY))
            return 4;
    }
#ifdef DEBUG
    printk("probing for %s: present=%d, media=%d, probetype=%s\n",
        drive->name, drive->present, drive->media,
        (cmd == WIN_IDENTIFY) ? "ATA" : "ATAPI");
#endif
    ide_delay_50ms();  /* needed for some systems (e.g. crw9624 as drive0 with disk as slave) */
    SELECT_DRIVE(hwif,drive);          //发给 ide 设备 select 命令
    ide_delay_50ms();                  //等待硬盘响应

    if (IN_BYTE(IDE_SELECT_REG) != drive->select.all){
```

```
                //硬盘加电后，响应较慢，多发几次 select 命令
        unsigned char i, select;
        printk("IDE: waiting for drives to settle...\n");
        i = 0;
        while(1){
            SELECT_DRIVE(hwif,drive);
            ide_delay_50ms();
            select = IN_BYTE(IDE_SELECT_REG);
            if((select == drive->select.all) || i > 18)
                break;
            else
                i++;
            ide_delay_50ms();
        }
    }

    if (IN_BYTE(IDE_SELECT_REG) != drive->select.all && !drive->present) {
        //读入 select 寄存器的值看是否和写入的值相同，并且没有证明设备存在
        if (drive->select.b.unit != 0) {        //如果是检测第二个设备
            SELECT_DRIVE(hwif,&hwif->drives[0]);  /* exit with drive0 selected */
            ide_delay_50ms();         /* allow BUSY_STAT to assert & clear */
        }
        return 3;       /* no i/f present: mmm.. this should be a 4 -ml */
    }

    if (OK_STAT(GET_STAT(),READY_STAT,BUSY_STAT)
        || drive->present || cmd == WIN_PIDENTIFY)
        //如果是硬盘设备准备就绪，或者设备已证明存在，或者是第二次探测此设备
    {
        if ((rc = try_to_identify(drive,cmd)))      /* send cmd and wait */
            rc = try_to_identify(drive,cmd); /* failed: try again */
                                    //可以进行两次识别
        if (rc == 1 && cmd == WIN_PIDENTIFY && drive->autotune != 2) {
                                    //如果没有成功，并且是第二此探测，可以 reset ide 设备，
等
                                    //ide 设备 reset 成功以后，再次识别
            unsigned long timeout;
            printk("%s: no response (status = 0x%02x), resetting drive\n", drive->name,
GET_STAT());
            ide_delay_50ms();
            OUT_BYTE (drive->select.all, IDE_SELECT_REG);
            ide_delay_50ms();
            OUT_BYTE(WIN_SRST, IDE_COMMAND_REG);
            timeout = jiffies;
```

```
            while ((GET_STAT() & BUSY_STAT) && time_before(jiffies, timeout + WAIT_WORSTCASE))
                ide_delay_50ms();
            rc = try_to_identify(drive, cmd);
        }
        if (rc == 1)                                //失败
            printk("%s: no response (status = 0x%02x)\n", drive->name, GET_STAT());
        (void) GET_STAT();          /* ensure drive irq is clear */
    } else {
        rc = 3;                     /* not present or maybe ATAPI */
    }
    if (drive->select.b.unit != 0) {
        SELECT_DRIVE(hwif,&hwif->drives[0]);  /* exit with drive0 selected */
        ide_delay_50ms();
        (void) GET_STAT();          /* ensure drive irq is clear */
    }
    return rc;
}

static int hwif_init (ide_hwif_t *hwif)
{
    if (!hwif->present)         //如果不存在
        return 0;
    if (!hwif->irq) {                   //如果没有 irq 号
        if (!(hwif->irq = ide_default_irq(hwif->io_ports[IDE_DATA_OFFSET])))
        {
            printk("%s: DISABLED, NO IRQ\n", hwif->name);
            return (hwif->present = 0);
        }
    }
#ifdef CONFIG_BLK_DEV_HD        //没定义这个宏
    if (hwif->irq == HD_IRQ && hwif->io_ports[IDE_DATA_OFFSET] != HD_DATA) {
        printk("%s: CANNOT SHARE IRQ WITH OLD HARDDISK DRIVER (hd.c)\n", hwif->name);
        return (hwif->present = 0);
    }
#endif /* CONFIG_BLK_DEV_HD */

    hwif->present = 0; /* we set it back to 1 if all is ok below */

    //注册块设备
    if (devfs_register_blkdev (hwif->major, hwif->name, ide_fops)) {
        printk("%s: UNABLE TO GET MAJOR NUMBER %d\n", hwif->name, hwif->major);
        return (hwif->present = 0);
```

```
    }

    if (init_irq(hwif)) { //初始化中断
        int i = hwif->irq;
        /*
         *  It failed to initialise. Find the default IRQ for
         *  this port and try that.
         */
        if (!(hwif->irq = ide_default_irq(hwif->io_ports[IDE_DATA_OFFSET]))) {
                                            //没有缺省的 irq 号
            printk("%s: Disabled unable to get IRQ %d.\n", hwif->name, i);
            (void) unregister_blkdev (hwif->major, hwif->name);
            return (hwif->present = 0);
        }
        if (init_irq(hwif)) {                   //再次初始化中断
            printk("%s: probed IRQ %d and default IRQ %d failed.\n",
                hwif->name, i, hwif->irq);
            (void) unregister_blkdev (hwif->major, hwif->name);
                                            //失败，取消注册
            return (hwif->present = 0);
        }
        printk("%s: probed IRQ %d failed, using default.\n",
            hwif->name, hwif->irq);
    }

    init_gendisk(hwif);                     //初始化通用硬盘
    blk_dev[hwif->major].data = hwif;       //request 数据结构赋值
    blk_dev[hwif->major].queue = ide_get_queue;         //
    read_ahead[hwif->major] = 8;       /* (4kB) */              //缓冲大小
    hwif->present = 1;  /* success */

#if (DEBUG_SPINLOCK > 0)                    //无效
{
    static int done = 0;
    if (!done++)
        printk("io_request_lock is %p\n", &io_request_lock);       /* FIXME */
}
#endif
    return hwif->present;
}
```

三．实验内容和步骤

1．　Mount IDE 盘，读取数据等，使用 cat /proc/devices 查看当前系统设备的变化，使用 cat

/proc/partitions 查看当前分区的变化等等。测量 IDE 盘的读写速度等。

(1) 初步调试：向磁头号寄存器中写入 0xa0，再读出该寄存器的值，读出的值和写入的值应该是相等的。在 ppcboot 提示符调试如下：

```
mw.w    7c00000c    a0
md.w     7c00000c    1
```

(2) 进一步调试：这一步我们要从磁盘上读取磁盘 ID 信息，读取的方法是向读状态/写命令寄存器中写入读磁盘 ID 信息的命令字 0xec，写入命令之后连续读读写数据寄存器，所读取的信息就是磁盘 ID 信息了（磁盘 ID 信息上面有描述）。在 ppcboot 提示符下实现如下：

```
mw.w    7c00000e    ec
md.w    7c000000     1
md.w    7c000000     1
     .          .          <－可根据需要来决定读的次数
.md.w    7c000000     1
```

四．思考题

1． 如何优化 IDE 读写代码以进一步提升 IDE 读写性能？

实验二十三： PS2 键盘试验

一．实验目的

通过本实验，使学生掌握 PS/2 键盘体系结构，了解 Linux 下的 PS/2 键盘驱动程序的流程。

二．实验原理和说明

1． 键盘硬件体系：

(1) 键盘发展的历史：

键盘发展经历了三代历程：PC/XT Keyboard，AT Keyboard，PS/2 Keyboard。近些年随着 USB 接口的出现，键盘接口有逐渐转向 USB 的趋势。

IBM PC/XT Keyboard (1981):

83 keys

5-pin DIN connector

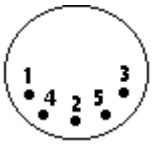
Simple uni-directional serial protocol

Uses what we now refer to as scan code set 1

No host-to-keyboard commands

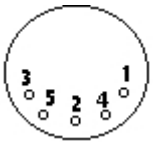
IBM AT Keyboard (1984) - Not backward compatible with XT systems.

Male



(Plug)

Female



(Socket)

5-pin DIN (AT/XT):

1 - Clock

2 - Data

3 - Not Implemented

4 - Ground

5 - Vcc (+5V)

84 -101 keys

5-pin DIN connector

Bi-directional serial protocol

Uses what we now refer to as scan code set 2

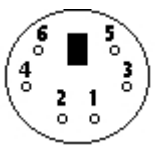
Eight host-to-keyboard commands

IBM PS/2 Keyboard (1987) - Compatible with AT systems, not compatible with XT systems.

Male

(Plug)

Female



(Socket)

6-pin Mini-DIN (PS/2):

1 - Data

2 - Not Implemented

3 - Ground

4 - Vcc (+5V)

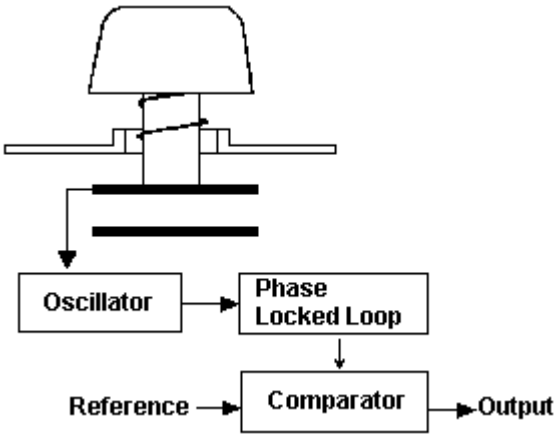
5 - Clock

6 - Not Implemented

84 - 101 keys
6-pin mini-DIN connector
Bi-direction serial protocol
Offers optional scan code set 3
17 host-to-keyboard commands

(2) 键盘按键的工作原理：

当键盘按下时会发送键盘扫描码（scancode），如果按住这个键不放，键盘会以一定的频率持续的发送同样的扫描码直到释放该键，当释放没个按键时，键盘还会发一种叫 break code 的键盘扫描码，以向主机报告该键已经释放。



图一

(3) 键盘扫描码

键盘扫描码一共有三套，对于 PS/2 键盘，它发送出来的键盘扫描码对应的是 Keyboard Scan Codes: Set 1，但是在 PC 机上操作系统处理中的键盘驱动程序处理的扫描码却并非 Keyboard Scan Codes: Set 1，而是 Keyboard Scan Codes: Set 2，可能大家都感觉到比较奇怪，有兴趣的，可以用示波器测试一下。为什么会这样呢？那么到底 Keyboard Scan Codes: Set 1 到 Keyboard Scan Codes: Set 2 的转换在哪儿做的呢？这个问题答案很简单，经过思考应该不难得到。

(a) Keyboard Scan Codes: Set 1

*All values are in hexadecimal

101-, 102-, and 104-key keyboards:

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1E	9E	9	0A	8A	[	1A	9A
B	30	B0	`	29	89	INSERT	E0,52	E0,D2
C	2E	AE	-	0C	8C	HOME	E0,47	E0,97
D	20	A0	=	0D	8D	PG UP	E0,49	E0,C9
E	12	92	\	2B	AB	DELETE	E0,53	E0,D3

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
F	21	A1	BKSP	0E	8E	END	E0,4F	E0,CF
G	22	A2	SPACE	39	B9	PG DN	E0,51	E0,D1
H	23	A3	TAB	0F	8F	U ARROW	E0,48	E0,C8
I	17	97	CAPS	3A	BA	L ARROW	E0,4B	E0,CB
J	24	A4	L SHFT	2A	AA	D ARROW	E0,50	E0,D0
K	25	A5	L CTRL	1D	9D	R ARROW	E0,4D	E0,CD
L	26	A6	L GUI	E0,5B	E0,DB	NUM	45	C5
M	32	B2	L ALT	38	B8	KP /	E0,35	E0,B5
N	31	B1	R SHFT	36	B6	KP *	37	B7
O	18	98	R CTRL	E0,1D	E0,9D	KP -	4A	CA
P	19	99	R GUI	E0,5C	E0,DC	KP +	4E	CE
Q	10	19	R ALT	E0,38	E0,B8	KP EN	E0,1C	E0,9C
R	13	93	APPS	E0,5D	E0,DD	KP .	53	D3
S	1F	9F	ENTER	1C	9C	KP 0	52	D2
T	14	94	ESC	01	81	KP 1	4F	CF
U	16	96	F1	3B	BB	KP 2	50	D0
V	2F	AF	F2	3C	BC	KP 3	51	D1
W	11	91	F3	3D	BD	KP 4	4B	CB
X	2D	AD	F4	3E	BE	KP 5	4C	CC
Y	15	95	F5	3F	BF	KP 6	4D	CD
Z	2C	AC	F6	40	C0	KP 7	47	C7
0	0B	8B	F7	41	C1	KP 8	48	C8
1	02	82	F8	42	C2	KP 9	49	C9
2	03	83	F9	43	C3	]	1B	9B
3	04	84	F10	44	C4	;	27	A7
4	05	85	F11	57	D7	'	28	A8
5	06	86	F12	58	D8	,	33	B3
6	07	87	PRNT SCRN	E0,2A, E0,37	E0,B7, E0,AA	.	34	B4
7	08	88	SCROLL	46	C6	/	35	B5
8	09	89	PAUSE	E1,1D,45 E1,9D,C5	-NONE-			

ACPI Scan Codes:

Key	Make Code	Break Code
Power	E0, 5E	E0, DE
Sleep	E0, 5F	E0, DF
Wake	E0, 63	E0, E3

Windows Multimedia Scan Codes:

Key	Make Code	Break Code
Next Track	E0, 19	E0, 99
Previous Track	E0, 10	E0, 90
Stop	E0, 24	E0, A4
Play/Pause	E0, 22	E0, A2
Mute	E0, 20	E0, A0
Volume Up	E0, 30	E0, B0
Volume Down	E0, 2E	E0, AE
Media Select	E0, 6D	E0, ED
E-Mail	E0, 6C	E0, EC
Calculator	E0, 21	E0, A1
My Computer	E0, 6B	E0, EB
WWW Search	E0, 65	E0, E5
WWW Home	E0, 32	E0, B2
WWW Back	E0, 6A	E0, EA
WWW Forward	E0, 69	E0, E9
WWW Stop	E0, 68	E0, E8
WWW Refresh	E0, 67	E0, E7
WWW Favorites	E0, 66	E0, E6

(b) Keyboard Scan Codes: Set 2

*All values are in hexadecimal

101-, 102-, and 104-key keyboards:

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1C	F0,1C	9	46	F0,46	[	54	F0,54
B	32	F0,32	`	0E	F0,0E	INSERT	E0,70	E0,F0,70
C	21	F0,21	-	4E	F0,4E	HOME	E0,6C	E0,F0,6C

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
D	23	F0,23	=	55	F0,55	PG UP	E0,7D	E0,F0,7D
E	24	F0,24	\	5D	F0,5D	DELETE	E0,71	E0,F0,71
F	2B	F0,2B	BKSP	66	F0,66	END	E0,69	E0,F0,69
G	34	F0,34	SPACE	29	F0,29	PG DN	E0,7A	E0,F0,7A
H	33	F0,33	TAB	0D	F0,0D	U ARROW	E0,75	E0,F0,75
I	43	F0,43	CAPS	58	F0,58	L ARROW	E0,6B	E0,F0,6B
J	3B	F0,3B	L SHFT	12	F0,12	D ARROW	E0,72	E0,F0,72
K	42	F0,42	L CTRL	14	F0,14	R ARROW	E0,74	E0,F0,74
L	4B	F0,4B	L GUI	E0,1F	E0,F0,1F	NUM	77	F0,77
M	3A	F0,3A	L ALT	11	F0,11	KP /	E0,4A	E0,F0,4A
N	31	F0,31	R SHFT	59	F0,59	KP *	7C	F0,7C
O	44	F0,44	R CTRL	E0,14	E0,F0,14	KP -	7B	F0,7B
P	4D	F0,4D	R GUI	E0,27	E0,F0,27	KP +	79	F0,79
Q	15	F0,15	R ALT	E0,11	E0,F0,11	KP EN	E0,5A	E0,F0,5A
R	2D	F0,2D	APPS	E0,2F	E0,F0,2F	KP .	71	F0,71
S	1B	F0,1B	ENTER	5A	F0,5A	KP 0	70	F0,70
T	2C	F0,2C	ESC	76	F0,76	KP 1	69	F0,69
U	3C	F0,3C	F1	05	F0,05	KP 2	72	F0,72
V	2A	F0,2A	F2	06	F0,06	KP 3	7A	F0,7A
W	1D	F0,1D	F3	04	F0,04	KP 4	6B	F0,6B
X	22	F0,22	F4	0C	F0,0C	KP 5	73	F0,73
Y	35	F0,35	F5	03	F0,03	KP 6	74	F0,74
Z	1A	F0,1A	F6	0B	F0,0B	KP 7	6C	F0,6C
0	45	F0,45	F7	83	F0,83	KP 8	75	F0,75
1	16	F0,16	F8	0A	F0,0A	KP 9	7D	F0,7D
2	1E	F0,1E	F9	01	F0,01	]	5B	F0,5B
3	26	F0,26	F10	09	F0,09	;	4C	F0,4C
4	25	F0,25	F11	78	F0,78	'	52	F0,52
5	2E	F0,2E	F12	07	F0,07	,	41	F0,41
6	36	F0,36	PRNT SCRN	E0,12, E0,7C	E0,F0, 7C,E0, F0,12	.	49	F0,49
7	3D	F0,3D	SCROLL	7E	F0,7E	/	4A	F0,4A
8	3E	F0,3E	PAUSE	E1,14,77, E1,F0,14, F0,77	-NONE-			

ACPI Scan Codes:

Key	Make Code	Break Code
Power	E0, 37	E0, F0, 37
Sleep	E0, 3F	E0, F0, 3F
Wake	E0, 5E	E0, F0, 5E

Windows Multimedia Scan Codes:

Key	Make Code	Break Code
Next Track	E0, 4D	E0, F0, 4D
Previous Track	E0, 15	E0, F0, 15
Stop	E0, 3B	E0, F0, 3B
Play/Pause	E0, 34	E0, F0, 34
Mute	E0, 23	E0, F0, 23
Volume Up	E0, 32	E0, F0, 32
Volume Down	E0, 21	E0, F0, 21
Media Select	E0, 50	E0, F0, 50
E-Mail	E0, 48	E0, F0, 48
Calculator	E0, 2B	E0, F0, 2B
My Computer	E0, 40	E0, F0, 40
WWW Search	E0, 10	E0, F0, 10
WWW Home	E0, 3A	E0, F0, 3A
WWW Back	E0, 38	E0, F0, 38
WWW Forward	E0, 30	E0, F0, 30
WWW Stop	E0, 28	E0, F0, 28
WWW Refresh	E0, 20	E0, F0, 20
WWW Favorites	E0, 18	E0, F0, 18

(c) AT Keyboard Scan Codes (Set 3)

*All values are in hexadecimal

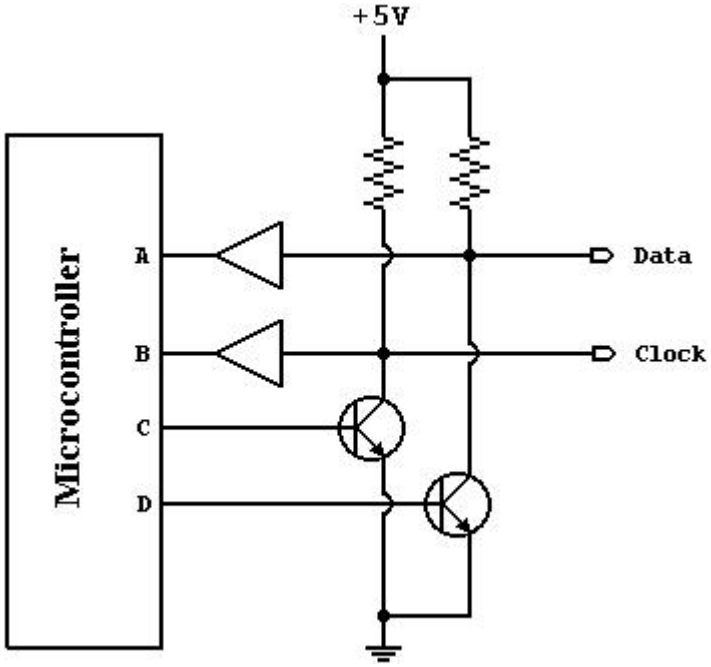
KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1C	F0,1C	9	46	F0,46	[	54	F0,54

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
B	32	F0,32	`	0E	F0,0E	INSERT	67	F0,67
C	21	F0,21	-	4E	F0,4E	HOME	6E	F0,6E
D	23	F0,23	=	55	F0,55	PG UP	6F	F0,6F
E	24	F0,24	\	5C	F0,5C	DELETE	64	F0,64
F	2B	F0,2B	BKSP	66	F0,66	END	65	F0,65
G	34	F0,34	SPACE	29	F0,29	PG DN	6D	F0,6D
H	33	F0,33	TAB	0D	F0,0D	U ARROW	63	F0,63
I	43	F0,48	CAPS	14	F0,14	L ARROW	61	F0,61
J	3B	F0,3B	L SHFT	12	F0,12	D ARROW	60	F0,60
K	42	F0,42	L CTRL	11	F0,11	R ARROW	6A	F0,6A
L	4B	F0,4B	L WIN	8B	F0,8B	NUM	76	F0,76
M	3A	F0,3A	L ALT	19	F0,19	KP /	4A	F0,4A
N	31	F0,31	R SHFT	59	F0,59	KP *	7E	F0,7E
O	44	F0,44	R CTRL	58	F0,58	KP -	4E	F0,4E
P	4D	F0,4D	R WIN	8C	F0,8C	KP +	7C	F0,7C
Q	15	F0,15	R ALT	39	F0,39	KP EN	79	F0,79
R	2D	F0,2D	APPS	8D	F0,8D	KP .	71	F0,71
S	1B	F0,1B	ENTER	5A	F0,5A	KP 0	70	F0,70
T	2C	F0,2C	ESC	08	F0,08	KP 1	69	F0,69
U	3C	F0,3C	F1	07	F0,07	KP 2	72	F0,72
V	2A	F0,2A	F2	0F	F0,0F	KP 3	7A	F0,7A
W	1D	F0,1D	F3	17	F0,17	KP 4	6B	F0,6B
X	22	F0,22	F4	1F	F0,1F	KP 5	73	F0,73
Y	35	F0,35	F5	27	F0,27	KP 6	74	F0,74
Z	1A	F0,1A	F6	2F	F0,2F	KP 7	6C	F0,6C
0	45	F0,45	F7	37	F0,37	KP 8	75	F0,75
1	16	F0,16	F8	3F	F0,3F	KP 9	7D	F0,7D
2	1E	F0,1E	F9	47	F0,47	]	5B	F0,5B
3	26	F0,26	F10	4F	F0,4F	;	4C	F0,4C
4	25	F0,25	F11	56	F0,56	'	52	F0,52
5	2E	F0,2E	F12	5E	F0,5E	,	41	F0,41
KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
6	36	F0,36	PRNT SCRN	57	F0,57	.	49	F0,49
7	3D	F0,3D	SCROLL	5F	F0,5F	/	4A	F0,4A

8	3E	F0,3E	PAUSE	62	F0,62			

(4) PS/2 接口原理：

图二是一个通用外设接口模型。数据线和时钟线分别连在微控制器的管脚 A 和管脚 B。两根线都通过上拉电阻接到＋5V 电源上，当 C 管脚和 D 管脚输入为逻辑“1”时，数据线和电源线的电位为 0。从上面的原理可以看出，数据线的电位反相于 D 的输出，时钟线的电位反相于 C 的输出。



图二

(a) 交互: 设备到主机

PS/2 接口的数据线和时钟线都是基于上面的通用外设接口模型。数据线和时钟线都通过上拉电阻接到＋5V 电源上，所以两根线的空闲状态都是高电位。当键盘和鼠标要发送数据时，会首先确认时钟线在高电位，也就是逻辑“1”。如果不是逻辑“1”，主机将禁止与设备通信。设备这是只能把数据暂存在 buffer 中，等待主机时钟线空闲。而且只有时钟线空闲达到 50 毫秒，设备才能发送数据。

根据上面所说的，键盘，鼠标和主机通信用的是 11 个 bit 位一帧的串行通信协议。11bit 的含义如下：

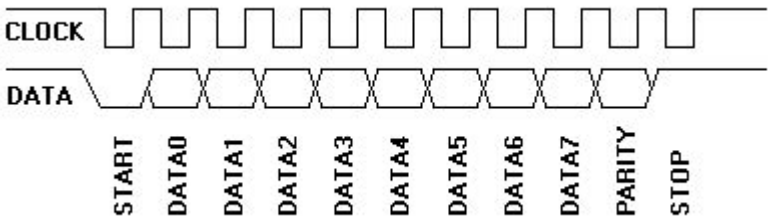
1 start bit.　This is always 0.

8 data bits, least significant bit first.

1 parity bit (odd parity).

1 stop bit.　This is always 1.

键盘和鼠标在时钟为高时发送 1bit 数据，主机则在时钟为低时接收键盘和鼠标发送的数据。参照图二和图三的说明：



图三：设备到主机。
数据线在时钟为高时改变状态，否则为无效数据。

图四：在键盘向计算机发送数据时用示波器截取的波形图。

A 通道是时钟信号；B 通道是数据信号。传送的是键盘“Q”的扫描码（15H）。时钟频率是 10－16。7kHZ。从时钟的上升沿到传送数据至少要间隔 5 毫秒。从传送数据到时钟的下降沿也是至少要 5 毫秒，而且不能大于 25 毫秒。

主机可以在任何时候通过把数据线下来到低电位来禁止设备和主机的通信。如果在一帧的第十一位传送之前，主机禁止了传送，那么设备必须中断当前的通信，等待主机释放信号线时重新传送数据。要重新传送的数据可能并不只是没有被成功传输的 11bit，比如说，当键盘传送 break code 的第二个字节时受到主机的中断，那么键盘下次必须重新传输真个 break code，而不是仅仅被打断的那一个 byte。

在开始传输第一个 bit 位之前以及传输完最后一个 bit 位之后，主机设置数据线电位为低，并不需要键盘或者鼠标重新传输数据，不过，如果这个时候新的数据已经产生并且等待传输，这些数据必须要被缓存在键盘的 buffer 中，直到主机重新释放数据线。键盘有一个 16byte 的传输区专门来缓冲等待传输的数据，如果要缓冲的数据大于 16byte 时，多余的数据将被丢弃。对鼠标来讲，它只在 buffer 中缓存鼠标当前的移动。

(b)　主机到设备的通信：

主机到设备的通信和设备到主机的通信有些不同，相对来说，主机到设备的通信更复杂一下。PS/2 设备在整个传输过程中总是不断的产生时钟信号，当主机要发送数据时，主机必须把自己的时钟线和数据线调整到“发送请求”状态：

首先下拉时钟信号到低电位，持续 100 微秒以上来禁止通信；置数据信号为低，然后释放

时钟线，发起“发送请求”。设备将每隔一段时间检测“发送请求”状态，当检测到该状态时设备会开始不断的产生时钟信号。之后主机将发送一个开始位，八个数据位，一个奇偶校验位，一个停止位。

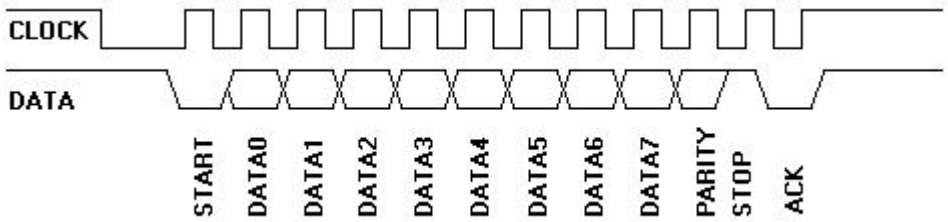
在接收到停止位后，设备会把时钟线下来到低电位，来告诉主机已经接收到了一 byte 的数据。如果主机在传输完第 11bit 后并没有释放时钟线，那么设备会持续下拉时钟线为低电位，直到主机释放时钟线（之后，设备会产生一个错误）。

主机可以在任何时候通过设置时钟线电位为低来中止传输。

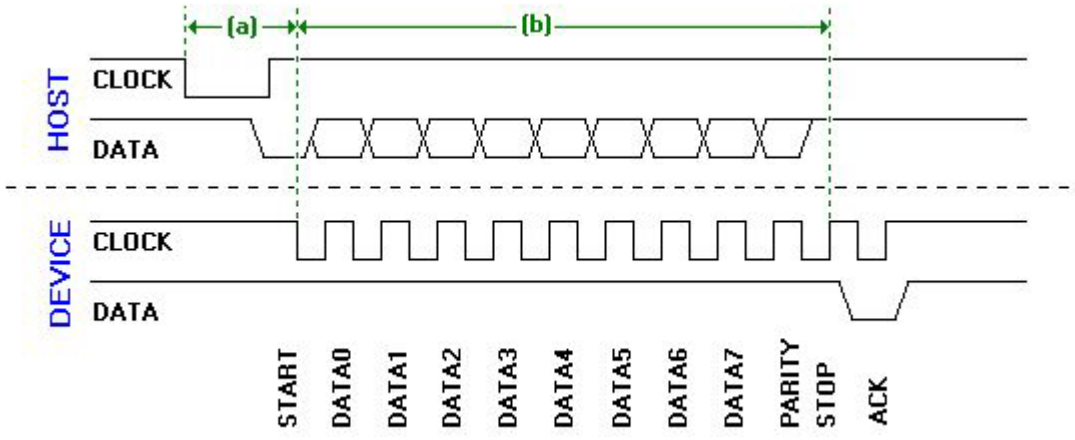
下面是主机发送信息到 PS/2 设备的步骤，可以帮助理解这一过程：

1) Bring the Clock line low for at least 100 microseconds.
2) Bring the Data line low.
3) Release the Clock line.
4) Wait for the device to bring the Clock line low.
5) Set/reset the Data line to send the first data bit
6) Wait for the device to bring Clock high.
7) Wait for the device to bring Clock low.
8) Repeat steps 5-7 for the other seven data bits and the parity bit
9) Release the Data line.
10) Wait for the device to bring Data low.
11) Wait for the device to bring Clock low.
12) Wait for the device to release Data and Clock

图五显示了上面流程所对应的时序图。图六详细的指示了时序图中哪个信号是由主机产生，哪个信号是由 PS/2 设备产生。注意时序中的“ack”位：它是在时钟线为高时产生的（和其他的 11bit 是不一样的）。



图五：主机到设备的通信



图六：主机到设备通信的详细时序

参考图六，主机会查询两个状态：

(i) 在主机下拉时钟信号为低后，主机会查询设备是否开始产生时钟信号，这个过程不能超过 15ms；

(ii) 主机也会检查数据是否被发送，这个过程不能超过 2ms。

这两种状态限制都必须得到满足，否则主机将报告一个错误。当主机发送数据到设备时，主机一接收到“ack”位就必须下拉时钟信号为低来禁止继续通讯。如果主机发出的命令需要得到回复，主机必须在释放时钟线 20ms 后接受到回复。如果没有得到回复，主机也会报告一个错误。

2． 键盘驱动程序代码

(1) 键盘模式

键盘模式有 4 种， 在 Linux 下你可以用 kbd_mode -参数 来设置和显示你的模式：

(a) (-s) Scancode mode （raw ）raw 模式：将键盘端口上读出的扫描码放入缓冲区；

(b) (-k) Keycode mode (mediumraw) mediumraw 模式：将扫描码过滤为键盘码放入缓冲区；

(c) (-a) ASCII mode (XLATE) XLATE 模式：识别各种键盘码的组合，转换为 TTY 终端代码放入缓冲区；

(d) (-u) UTF-8 MODE (UNICODE) Unicode 模式：UNICODE 模式基本上与 XLATE 相同，只不过可以通过数字小键盘间接输入 UNICODE 代码。

(2) 键盘模块的上层漫游：

键盘模块的最上层严格的说应该是 TTY 设备，在这里不加描述。对于使用者而言，keyboard.c 是我们的最上层。

在 keyboard.c 中，不涉及底层操作，也不涉及到任何体系结构，他主要负责：键盘初始化、键盘 tasklet 的挂入、按键盘后的处理、keymap 的装入、scancode 的转化、与 TTY 设备的通信。也就是不论什么体系结构，keyboard.c 都是相同的，但是和和体系结构有关的代码各个体系结构就有些不同，比如说 x86 体系结构它用到的低层操作就是 pc_keyb.c。

Keyboard.c 是一个大杂烩，包含了大多数 keyboard 的 key 的处理，因此代码才变的庞大和复杂，我们可以修改它，让他变的很小（但这样做并不会使空间节省，因为 keyboard.c 的许多代码是动态加载的）。

Keyboard.c 中的 int __init kbd_init(void) 函数是键盘代码执行的开始点，但 keyboard.c 并非是一定执行的，你必须先定义 CONFIG_VT（Character devices -> Virtual terminal）才有效，为什么？因为 kbd_init（）是在 tty_io.c 的 tty_init（）中被调用的。

Kbd_init（）在进行一些基本初始化后，执行 kbd_init_hw（），此函数可以看做是一个统一的上层界面，对于不同的体系或相同体系下不同的 board，他们的 kbd_init_hw（）的实现代码是不同的(同过 CONFIG_XXX_XXX 来实现)，他的实现代码就是操作硬件。kbd_init_hw（）会：检查硬件及有效性、分配资源和中断、操作寄存器的实现函数调用。

初始化一完成，就把 keyboard_tasklet 放到 CPU tasklet 中（注意到 keyboard_tasklet 是开放出来的，等下可以看到怎么使用它）。

现在就等你按键了，当有按键时，会调用键盘中断处理函数，一般都是体系下的 keyboard_interrupt()，此函数会调用到 keyboard.c 中的 handle_scancode() 。

handle_scancode()就是我们的重点，这个函数最终决定了我们按下的 key 如何处理，在下面会分析他的流程，他主要做：与 TTY 的通信、keymap 的装入、按键的处理。
handle_scancode()处理的结果就是把你按下的 key 发到不同的处理函数中，一般的，这些函数离最终的结果已经很近了。这其中，大多数的函数都会调用 put_queue() 函数。

put_queue()的工作原理就是利用 TTY, 它将经过转换的键盘功能码用 tty_insert_flip_char()放入到当前 TTY 的 flip buffer 中，然后将缓冲区输出任务函数(flush_to_ldisc)添加到控制台任务队列(con_task_queue)并激活控制台软中断执行该任务函数. flush_to_ldisc()翻转读写缓冲区,将缓冲区接收数据传递给 tty 终端规程的 n_tty_receive_buf()接收函数,n_tty_receive_buf()处理输入字符,将输出字符缓冲在终端的循环缓冲区(read_buf)之中.用户通过 tty 规程的 read_chan()读取 read_buf 中的字符.当多个进程同时读取同一终端时, 使用 tty->atomic_read 信号灯来竞争读取权.

```
int __init kbd_init(void)
{
    int i;
    /*  定义一个 kbd_struct, 他用于保存当前键盘 LED 灯状态、缺省 keymap 表、键盘复合锁定状态、一些功能灯的定义、键盘模式定义、及 modeflags 模式*/
    struct kbd_struct kbd0;

     /* console_driver 这个全局变量是很重要的,他维护着庞大的 TTY/Console 对象,承当 TTY 对外的输入和输出  */
     extern struct tty_driver console_driver;

     /*  缺省不亮灯  */
    kbd0.ledflagstate = kbd0.default_ledflagstate = KBD_DEFLEDS;
    /*  缺省，用于显示 flags */
    kbd0.ledmode = LED_SHOW_FLAGS;
    /*  缺省，表示用 key_map 的第一个表，即没有 lock 键*/
    kbd0.lockstate = KBD_DEFLOCK;
    /*  没有粘键  */
    kbd0.slockstate = 0;
    /* modeflags=0 */
    kbd0.modeflags = KBD_DEFMODE;
    /* ASCII 模式  */
    kbd0.kbdmode = VC_XLATE;

    /*  为每个控制台分配一个 KBD 的结构  *
    for (i = 0 ; i < MAX_NR_CONSOLES ; i++)
          kbd_table[i] = kbd0;

     /* ttytab 是一个很重要的指针，他维护着当前各个控制台的 tty_struct 表（即相当于一个 2 维表），tty_struct 可看成/dev/tty*的输入设备，只有在/dev/tty*打开时它才接收输入数据*/
     ttytab = console_driver.table;
```

```
    /* 以下就是涉及到硬件的操作，由于我们没有 KBD 硬件，因此我们可能希望屏蔽所产
    生的 error 信息*/
#ifdef CONFIG_SA1100_ROSETTA
    button_setup();      /* initialize button hardware */
#else
    kbd_init_hw();  /*下面会分析他*/
#endif

    /* 我们希望把 keyboard_tasklet 挂到 CPU 的运行队列中去，下面就是 */
    tasklet_enable(&keyboard_tasklet);
    tasklet_schedule(&keyboard_tasklet);

    /* 下面 register 一个电源管理的 KBD 设备？操作函数为 NULL? */
    pm_kbd = pm_register(PM_SYS_DEV, PM_SYS_KBC, NULL);

    return 0;
}
```

kbd_init_hw（）函数：

我们已经说了，kbd_init_hw() 相当与一个统一的界面，不同硬件操作代码是不同的，因为是涉及硬件的操作。对 arm 体系而言，目前的 kbd_init_hw() 提供了 4 种硬件代码，象如果是 ADS 板，就要提供 ADS 的相应代码。

我们下面选择分析 SA1100 体系的 kbd_init_hw()。SA1100 体系的此函数放在文件 include/asm-arm/arch-sa1100/keyboard.h 中。大家可以看到在这个文件中除了 kbd_init_hw() 外，还有 kbd_setkeycode（） / kbd_getkeycode（）/ kbd_translate（）/ ……等函数都是函数名相同，处理代码不同。而这些函数都会在 keyboard.c 中被调用到。

SA1100 定义了“CONFIG_SA1100_BRUTUS / CONFIG_SA1100_GRAPHICSCLIENT/ CONFIG_SA1111 / other board”4 种已知硬件的操作代码，对于 Assabet 就会调用 other board 的代码，也就是无论什么函数都为 NULL（因为没有这个硬件呀！），CONFIG_SA1100_GRAPHICSCLIENT 也就是 ADS 板的处理代码（SmartIO 芯片决定的），CONFIG_SA1111 也有自己的一块 KBD 芯片，而且功能和 PC 的差不多。我们来挑一个最复杂的来分析把。那就是 SA1111 了。我们从 kbd_init_hw() 进入：

```
void __init sa1111_init_hw(void)
{
    /* 分配 kbd 端口，其实根本就不用分配了，对 PC 来说，0x60~0x68 就是 KBD 的 IO 口，
    对 sa1100 更不用分配了，所以此函数为 NULL*/
    kbd_request_region();

    SKPCR |= (1<<6); /* 即 0x4000,0000+0x0200 处的 register 操作 */

    /* 以下初始化 KBD 的 4 个 register，以便开始下面的 test 工作 */
    KBDCLKDIV = 0;
    KBDPRECNT = 127;
```

```
    KBDCR = 0x08;
    mdelay(50);
    KBDDATA = 0xff;
    mdelay(50);

    /* Flush any pending input. */
    kbd_clear_input();

    if (kbd_startup_reset) { /* int kbd_startup_reset =1，所以肯定执行 */
        /* initialize_kbd() 真正开始 init 硬件，出错会 return 错误 message,显示在系统启动的
        时候 */
        char *msg = initialize_kbd();
        if (msg)
            printk(KERN_WARNING"initialize_kbd: %s\n", msg);
    }

#if defined CONFIG_PSMOUSE
    psaux_init();
#endif

    /* allocate the IRQ, keyboard_interrupt 是处理函数 */
    kbd_request_irq(keyboard_interrupt);

#if 0 /*@@@*/
    for (;;) {
        if (KBDSTAT & KBD_STAT_RXF)
            printk("K_%.2x ", KBDDATA & 0xff);
        if (MSESTAT & MSE_STAT_RXF)
            printk("M_%.2x ", MSEDATA & 0xff);
    }
#endif

#if 0 /*@@@*/
    timer_table[COMTROL_TIMER].fn = pckbd_timer;
    timer_table[COMTROL_TIMER].expires = jiffies + 2*HZ/100;
    timer_active |= 1 << COMTROL_TIMER;
#endif
}
```

initialize_kbd()函数：

这个函数如果逐层展开，会比较庞大，我们就分析最上层的把。

首先，要 reset 键盘，即对键盘的 KBDDATA 寄存器写 KBD_CMD_RESET 数据，然后读状态位,假如超时,那么可能是没有 AT 键盘挂上。这里有一个很重要的问题就是你在对 register 读的时候，先收到 KBD_REPLY_ACK 数据标志，然后再受到数据，因此你需要读 2 次。另，

这些硬件代码实在是太多了，我想，如果大家有什么不懂来问我把，我吃不削写了。这里涉及到很多的 键盘命令/键盘标志，你可以参考 pc_keyb.h 文件，它包括了特殊的键盘命令操作，你可能需要对键盘的原理有所了解才看的懂。

```
static char * __init initialize_kbd(void)
{
	int status;

#if 0 /*@@@*/
	/*
	* Test the keyboard interface.
	* This seems to be the only way to get it going.
	* If the test is successful a x55 is placed in the input buffer.
	*/
	kbd_write_command_w(KBD_CCMD_SELF_TEST);
	if (kbd_wait_for_input() != 0x55)
		return "Keyboard failed self test";

	/*
	* Perform a keyboard interface test.	This causes the controller
	* to test the keyboard clock and data lines.	The results of the
	* test are placed in the input buffer.
	*/
	kbd_write_command_w(KBD_CCMD_KBD_TEST);
	if (kbd_wait_for_input() != 0x00)
		return "Keyboard interface failed self test";

	/*
	* Enable the keyboard by allowing the keyboard clock to run.
	*/
	kbd_write_command_w(KBD_CCMD_KBD_ENABLE);
#endif

	/*
	* Reset keyboard. If the read times out
	* then the assumption is that no keyboard is
	* plugged into the machine.
	* This defaults the keyboard to scan-code set 2.
	*
	* Set up to try again if the keyboard asks for RESEND.
	*/
	do {
		kbd_write_output_w(KBD_CMD_RESET);
		status = kbd_wait_for_input();
```

```
        if (status == KBD_REPLY_ACK)
            break;
        if (status != KBD_REPLY_RESEND)
            return "Keyboard reset failed, no ACK";
    } while (1);

    if (kbd_wait_for_input() != KBD_REPLY_POR)
        return "Keyboard reset failed, no POR";

    /*
    * Set keyboard controller mode. During this, the keyboard should be
    * in the disabled state.
    *
    * Set up to try again if the keyboard asks for RESEND.
    */
    do {
        kbd_write_output_w(KBD_CMD_DISABLE);
        status = kbd_wait_for_input();
        if (status == KBD_REPLY_ACK)
            break;
        if (status != KBD_REPLY_RESEND)
            return "Disable keyboard: no ACK";
    } while (1);

#if 0 /*@@@*/
    kbd_write_command_w(KBD_CCMD_WRITE_MODE);
    kbd_write_output_w(KBD_MODE_KBD_INT
                       | KBD_MODE_SYS
                       | KBD_MODE_DISABLE_MOUSE
                       | KBD_MODE_KCC);
    /* ibm powerpc portables need this to use scan-code set 1 -- Cort */
    kbd_write_command_w(KBD_CCMD_READ_MODE);
    if (!(kbd_wait_for_input() & KBD_MODE_KCC)) {
        /*
        * If the controller does not support conversion,
        * Set the keyboard to scan-code set 1.
        */
        kbd_write_output_w(0xF0);
        kbd_wait_for_input();
        kbd_write_output_w(0x01);
        kbd_wait_for_input();
    }
#else
    kbd_write_output_w(0xf0);
```

```
    kbd_wait_for_input();
    kbd_write_output_w(0x01);
    kbd_wait_for_input();
#endif

    kbd_write_output_w(KBD_CMD_ENABLE);
    if (kbd_wait_for_input() != KBD_REPLY_ACK)
        return "Enable keyboard: no ACK";

    /*
    * Finally, set the typematic rate to maximum.
    */
    kbd_write_output_w(KBD_CMD_SET_RATE);
    if (kbd_wait_for_input() != KBD_REPLY_ACK)
        return "Set rate: no ACK";
    kbd_write_output_w(0x00);
    if (kbd_wait_for_input() != KBD_REPLY_ACK)
        return "Set rate: no ACK";

    return NULL;
}
```

键盘中断：

在做完 init 后，接下来就等待 key press 事件了，如果产生了 key press 事件，就会调用 keyboard_interrupt（）来处理，在 sa1111 中就是调用 handle_kbd_event（）函数。

handle_kbd_event（）函数：首先去读 status port，看是不是有按键事件，然后判断是键盘还是鼠标的，如果是键盘的，读出 scancode，然后判断 status register 的第 8 位是否为 1，如果是 1（表示正确，即 key press 时状态），那么调用 handle_keyboard_event(scancode)。

```
static inline void handle_keyboard_event(unsigned char scancode)
{
#ifdef CONFIG_VT
    kbd_exists = 1;

    /* 我们希望读出的不是键盘的 CMD 数据，正确，返回 1，否则 0 */
    if (do_acknowledge(scancode))
        /* 这就是我们要注意的最重要的函数，我们看到 scancode & 0x80，为什么要这样，
        因为 KBDDATA register 最高位是判断键是 press 还是 release，如果是 1 就是 release。
        后面的 7 位才是我们要的数据。 */
        handle_scancode(scancode, !(scancode & 0x80));
#endif
    /* 一不做，二不修，我们再一次把 keyboard_tasklet 放到 tasklet 中，确保 bottom half 的
    执行。*/
    tasklet_schedule(&keyboard_tasklet);
```

}

handle_scancode（）函数：

进入 handle_scancode()， 你准备好了吗？，首先我们要看到此函数是 kernel EXPORT 出去的函数，也就是说我们可以在 user-level 程序中调用他，不过，如果把这段代码改写一下，在 user-level 中调用才有意义的多。

Handle_scancode()主要是：判断 key 是按下还是释放，然后把当前的 tty_driver 赋给变量 tty，tty_driver 可看成/dev/tty*的输出设备，不要和 tty_struct 混乱起来。

```
void handle_scancode(unsigned char scancode, int down)
{
    unsigned char keycode;
    /* 判断，如果是 press 那么 up_flag=0，否则 up_flag=0200 */
    char up_flag = down ? 0 : 0200;
    char raw_mode;   /* 键盘模式 */

    pm_access(pm_kbd);

    add_keyboard_randomness(scancode | up_flag);

    /* 把当前的 TTY 参数传给变量 tty, 由 tty 来控制操作 */
    tty = ttytab? ttytab[fg_console]: NULL;
    /* 我们如果直接操作 tty 是危险的，因为我们不知道 tty 是否被打开了，所以我们通过 tty->driver_data 来判断，tty 打开的时候，tty->driver_data 是被设置的，tty 关闭的时候，他被 clear */
    if (tty && (!tty->driver_data)) {
    /*
    * We touch the tty structure via the ttytab array
    * without knowing whether or not tty is open, which
    * is inherently dangerous.   We currently rely on that
    * fact that console_open sets tty->driver_data when
    * it opens it, and clears it when it closes it.
        */
        tty = NULL;
    }
    /*kbd_table 是 kbd[]的指针，所以 kbd 就是当前 tty 的 kbd_struct 的指针 */
    kbd = kbd_table + fg_console;

    /* 我们判断当前的 kbd 模式是否为 VC_RAW，我们知道如果是的话，我们就可以把 scancode 直接放到 tty_flip_buffer 中。 */
    if ((raw_mode = (kbd->kbdmode == VC_RAW))) {
        /* we do not return yet, because we want to maintain the key_down array, so that we have
        the correct values when finishing RAW mode or when changing VT's */
        put_queue(scancode | up_flag);
```

```
    }

    /*
    * 把 scancode 转化为 keycode。 kbd_translate（）函数，
    * 我们分析 sa1111 的。分析在下面
    */
    if (!kbd_translate(scancode, &keycode, raw_mode))
        return;

    if (0) printk(__FUNCTION__ ": scancode=%d keycode=%d raw_mode=%d\n", scancode, keycode, raw_mode);

    /*
    * 变量 keycode 包含的是键控代码（u-char）
    * 注意到 keycode 不允许为 0 (on m68k 0 是允许的).
    * 我们明了了键的 up/down 状态，我们要把状态传进去
    * 并且如果是 MEDIUMRAW 模式，我们要返回 keycode 的值
    */

    kbd_processkeycode(keycode, up_flag, 0);
}
```

sa1111_translate（）函数：

因为 scancode 转化成 keycode 是这样的重要，所以我才不的不把他拿出来单独讲解了。我们只有知道 scancode 和 keycode 的区别，我们才可以做下面的事情。

我们知道我们从键盘的 IO 口读出的是 scancode，那我们怎么知道每个键的 scancode 呀？通过:Linux 下的 showkey –s 我们就可以知道每个键的 scancode， 如果想知道相应的 keycode，你重要用 showkey 就可以了（后面的代码我写了一个模仿 showkey 的程序）。通过测试，我们知道 1~88 键按下的 scancode 是“0x**”，而 89~128 是以 “0xe0(或 0xe1) 0x**”表示的。如果是释放，只要加上个 128 就可以了。然后我们来读以下的代码：

```
int sa1111_translate(unsigned char scancode, unsigned char *keycode, char raw_mode)
{
    static int prev_scancode = 0;

    /* special prefix scancodes.. */
    if (scancode == 0xe0 || scancode == 0xe1) {
        prev_scancode = scancode;
        return 0;
    }

    /* 0xFF is sent by a few keyboards, ignore it. 0x00 is error */
    if (scancode == 0x00 || scancode == 0xff) {
        prev_scancode = 0;
```

```
        return 0;
    }

    scancode &= 0x7f;

    if (prev_scancode) {
        /*
        * usually it will be 0xe0, but a Pause key generates
        * e1 1d 45 e1 9d c5 when pressed, and nothing when released
        */
        if (prev_scancode != 0xe0) {
            if (prev_scancode == 0xe1 && scancode == 0x1d) {
                prev_scancode = 0x100;
                return 0;
            } else if (prev_scancode == 0x100 && scancode == 0x45) {
                *keycode = E1_PAUSE;
                prev_scancode = 0;
            } else {
#ifdef KBD_REPORT_UNKN
                if (!raw_mode)
                    printk(KERN_INFO "keyboard: unknown e1 escape sequence\n");
#endif
                prev_scancode = 0;
                return 0;
            }
        } else {
            prev_scancode = 0;
            /*
            *   The keyboard maintains its own internal caps lock and
            *   num lock statuses. In caps lock mode E0 AA precedes make
            *   code and E0 2A follows break code. In num lock mode,
            *   E0 2A precedes make code and E0 AA follows break code.
            *   We do our own book-keeping, so we will just ignore these.
            */
            /*
            *   For my keyboard there is no caps lock mode, but there are
            *   both Shift-L and Shift-R modes. The former mode generates
            *   E0 2A / E0 AA pairs, the latter E0 B6 / E0 36 pairs.
            *   So, we should also ignore the latter. - aeb@cwi.nl
            */
            if (scancode == 0x2a || scancode == 0x36)
                return 0;

            if (e0_keys[scancode])
```

```
                *keycode = e0_keys[scancode];
            else {
#ifdef KBD_REPORT_UNKN
                if (!raw_mode)
                    printk(KERN_INFO "keyboard: unknown scancode e0 %02x\n",
                        scancode);
#endif
                return 0;
            }
        }
    } else if (scancode >= SC_LIM) {
        /* This happens with the FOCUS 9000 keyboard.Its keys PF1..PF12 are reported to
        generate 55 73 77 78 79 7a 7b 7c 74 7e 6d 6f.Moreover, unless repeated, they do not
        generate key-down events, so we have to zero up_flag below */
        /* Also, Japanese 86/106 keyboards are reported to generate 0x73 and 0x7d for \ - and \ |
        respectively. */
        /* Also, some Brazilian keyboard is reported to produce 0x73 and 0x7e for \ ? and KP-dot,
        respectively. */

        *keycode = high_keys[scancode - SC_LIM];

        if (!*keycode) {
            if (!raw_mode) {
#ifdef KBD_REPORT_UNKN
                printk(KERN_INFO"keyboard: unrecognized scancode (%02x)"
                    " - ignored\n", scancode);
#endif
            }
            return 0;
        }
    } else
            *keycode = scancode;
    return 1;
}
```

因为涉及到 high_keys[] 和 e0_keys[]，因此我们把代码拿进来读一读：

```
static unsigned char high_keys[128 - SC_LIM] = {
  RGN1, RGN2, RGN3, RGN4, 0, 0, 0,              /* 0x59-0x5f */
  0, 0, 0, 0, 0, 0, 0, 0,                       /* 0x60-0x67 */
  0, 0, 0, 0, 0, FOCUS_PF11, 0, FOCUS_PF12,     /* 0x68-0x6f */
  0, 0, 0, FOCUS_PF2, FOCUS_PF9, 0, 0, FOCUS_PF3,
  /* 0x70-0x77 */
  FOCUS_PF4, FOCUS_PF5, FOCUS_PF6, FOCUS_PF7,
        /* 0x78-0x7b */
```

```
    FOCUS_PF8, JAP_86, FOCUS_PF10, 0            /* 0x7c-0x7f */
};

static unsigned char e0_keys[128] = {
    0, 0, 0, 0, 0, 0, 0, 0,                     /* 0x00-0x07 */
    0, 0, 0, 0, 0, 0, 0, 0,                     /* 0x08-0x0f */
    0, 0, 0, 0, 0, 0, 0, 0,                     /* 0x10-0x17 */
    0, 0, 0, 0, E0_KPENTER, E0_RCTRL, 0, 0,
/* 0x18-0x1f */
    0, 0, 0, 0, 0, 0, 0, 0,                   /* 0x20-0x27 */
    0, 0, 0, 0, 0, 0, 0, 0,                     /* 0x28-0x2f */
    0, 0, 0, 0, 0, E0_KPSLASH, 0, E0_PRSCR,
/* 0x30-0x37 */
    E0_RALT, 0, 0, 0, 0, E0_F13, E0_F14, E0_HELP,
/* 0x38-0x3f */
    E0_DO, E0_F17, 0, 0, 0, 0, E0_BREAK, E0_HOME,
/* 0x40-0x47 */
    E0_UP, E0_PGUP, 0, E0_LEFT, E0_OK, E0_RIGHT, E0_KPMINPLUS, E0_END,
                                    /* 0x48-0x4f */
    E0_DOWN, E0_PGDN, E0_INS, E0_DEL, 0, 0, 0, 0,
/* 0x50-0x57 */
    0, 0, 0, E0_MSLW, E0_MSRW, E0_MSTM, 0, 0,
/* 0x58-0x5f */
    0, 0, 0, 0, 0, 0, 0, 0,                     /* 0x60-0x67 */
    0, 0, 0, 0, 0, 0, 0, E0_MACRO,              /* 0x68-0x6f */
    //0, 0, 0, 0, 0, 0, 0, 0,                   /* 0x70-0x77 */
    0, 0, 0, 0, 0, E0_BACKSLASH, 0, 0, /* 0x70-0x77 */
    0, 0, 0, E0_YEN, 0, 0, 0, 0                 /* 0x78-0x7f */
};
```

我们可以知道 scancode 到 keycode 的转化了。

对于 1~88 的 scancode，就等于 keycode， 比如 ’a’ 的 scancode 是 0x1e 和 0x9e,对应的 keycode 就是 30 和 158。对于 89-128 的 scancode 有*keycode = e0_keys[scancode]来转化，我们可以举列，比如小键盘的回车：scancode = 0xe0 0x1c，（大家别奇怪怎么发送，它会先发送 0xe0 然后发送 0x1c）,根据上面代码，流程应为： prev_scancode = 0xe0 --> scancode = 0x1c --> prev_scancode = 0 --> 执行 *keycode = e0_keys[scancode]; --> 找到 e0_keys[]的 0x1c 个元素，即第 28 个元素，即 E0_KPENTER，即为 96（#define E0_KPENTER 96）。另对于键是按下还是放开，keycode 是相同的。

kbd_processkeycode（）函数：

这才是我们真正要仔细考虑和分析的函数，keyboard.c 文件的庞大就是由于它的存在，kbd_processkeycode（）最主要的工作就是根据 keycode 和 keymap 来决定最终的处理函数。在这个函数里有几个很重要的变量： 一个就是 tty，它的值就是当前 tty 设备的环境；u_short

keysym 是我们真正要用的表示一个键的唯一的值，它是一个 32 位的数，高 16 位表示 type，type 的基数是 0xf0;低 16 位表示键的值;现在你就知道了为什么我们要不费其烦的把 scancode 转化成 keycode，就是因为我们要用 keycode 来找到其对应的 key_map[]中的元素。u_char type 就是 keysym 的高 16 位值，这个值一定大于 0xf0。我们引入他是因为我们需要对不同的见类型做不同的处理代码，比如方向键，小键盘的键，功能键等的处理代码都是不同的。int shift_final key_maps[]是一个 2 维表，那我们怎么知道我们要用哪张表呀，答案就是这个变量，他对 kbd_struct 中相应的变量进行读取，得到这个值，然后就用此值来决定使用哪张表，见下面代码。

```
ushort *key_maps[MAX_NR_KEYMAPS] =
{
    plain_map, shift_map, altgr_map, 0,
    ctrl_map, shift_ctrl_map, 0, 0,
    alt_map, 0, 0, 0,
    ctrl_alt_map,   0
};
```

这个变量维护着所有的可能出现的表，每个表都有自己的 128 个元素。ushort *key_map key_map 就指向 key_maps[]中具体的一张表了。见下面的代码

```
static void
kbd_processkeycode(unsigned char keycode, char up_flag, int autorepeat)
{
    char raw_mode = (kbd->kbdmode == VC_RAW); /*  键盘模式  */

    if (up_flag) { /*  放开键  */
        rep = 0;
        if(!test_and_clear_bit(keycode, key_down))
            up_flag = kbd_unexpected_up(keycode);
    } else { /*  按下键  */
        rep = test_and_set_bit(keycode, key_down);
        /* If the keyboard autorepeated for us, ignore it.
        * We do our own autorepeat processing.
        */
        if (rep && !autorepeat)
            return;
    }
    /*  以下是关与重发的处理，我们暂时忽略他  */

    if (kbd_repeatkeycode == keycode || !up_flag || raw_mode) {
        kbd_repeatkeycode = -1;
        del_timer(&key_autorepeat_timer);
    }
    do_poke_blanked_console = 1;
    tasklet_schedule(&console_tasklet);

#ifdef CONFIG_MAGIC_SYSRQ       /* Handle the SysRq Hack */
```

```
    if (keycode == SYSRQ_KEY) {
        sysrq_pressed = !up_flag;
        return;
    } else if (sysrq_pressed) {
        if (!up_flag) {
            handle_sysrq(kbd_sysrq_xlate[keycode], kbd_pt_regs, kbd, tty);
            return;
        }
    }
#endif

    /*
    * Calculate the next time when we have to do some autorepeat
    * processing.    Note that we do not do autorepeat processing
    * while in raw mode but we do do autorepeat processing in
    * medium raw mode.
    */
    if (!up_flag && !raw_mode) {
        kbd_repeatkeycode = keycode;
        if (vc_kbd_mode(kbd, VC_REPEAT)) {
            if (rep)
                key_autorepeat_timer.expires = jiffies + kbd_repeatinterval;
            else
                key_autorepeat_timer.expires = jiffies + kbd_repeattimeout;
            add_timer(&key_autorepeat_timer);
        }
    }

    if (kbd->kbdmode == VC_MEDIUMRAW) {
        /* soon keycodes will require more than one byte */
        put_queue(keycode + up_flag);
        raw_mode = 1;       /* Most key classes will be ignored */
    }

    /*   以下就是我们的重要的部分了，下面的代码必须被执行，由于代码展开的话比较琐
    碎，我把重要的地方用蓝色来表示  */
    /*
    * Small change in philosophy: earlier we defined repetition by
    *        rep = keycode == prev_keycode;
    *        prev_keycode = keycode;
    * but now by the fact that the depressed key was down already.
    * Does this ever make a difference? Yes.
    */
```

```
/*
*    Repeat a key only if the input buffers are empty or the
*    characters get echoed locally. This makes key repeat usable
*    with slow applications and under heavy loads.
*/
if (!rep ||
    (vc_kbd_mode(kbd,VC_REPEAT) && tty &&
    (L_ECHO(tty) || (tty->driver.chars_in_buffer(tty) == 0))))
{
    u_short keysym;
    u_char type;

    /* the XOR below used to be an OR */
    /* 我们要计算出当前的键盘状态，键盘状态你别告诉我你不知道呀，就是有没有按住“shift / ctrl / alt / shift+alt /……”呀，当然还有正常状态，也就是什么都 没有按住（=0） */
    int shift_final = (shift_state | kbd->slockstate) ^
        kbd->lockstate;
    /* key_maps[shift_final] 的定义在 defkeymap.c 中，他就是大名鼎鼎的 key 表，下面会详细的介绍，那么这个时候，key_map 就指向具体的一张表了。 */
    ushort *key_map = key_maps[shift_final];

    if (key_map != NULL) {
        /* 我们已经把 scancode 转化成了 keycode,很多人问为什么，就是要在这里用呀，根据 keycode 找到一个相对应的 keysym 的值，keysym 才是我们真正要用的值呀！ */
        keysym = key_map[keycode];
        /* 分析一下那些 key_maps 表，你就知道了，type 要取出这些 32 位数的高 16 位，为什么？因为，高 16 位是表示键类型用的，那什么叫“键类型”，键类型就在 include/linux/keyboard.h 中定义，如 KT_LATIN 表示一般键，KT_CUR 表示方向键（就 4 个，你别说不知道哪 4 个呀！），……. */
        type = KTYP(keysym);
        if (0) printk(__FUNCTION__ ": keysym=%#x type=%d shift_final=%d\n", keysym, type, shift_final);

        if (type >= 0xf0)   /* 一定会执行了 */
        {
            /* 我们要减掉一个基数 0xf0，当然了，你也可以不减，那就直接判断呀。 */
            type -= 0xf0;

            if (raw_mode && ! (TYPES_ALLOWED_IN_RAW_MODE & (1 << type)))
                return;
            if (type == KT_LETTER) /* 大写键比较特殊，再次转化以后在执行 */
```

```
                {
                    type = KT_LATIN; /*  一般的键的 type */
                    if (vc_kbd_led(kbd, VC_CAPSLOCK))
                    {
                        key_map = key_maps[shift_final ^ (1<<KG_SHIFT)];
                        if (key_map)
                            keysym = key_map[keycode];
                    }
                }
               if (0) printk(__FUNCTION__ ": key_handler=%p keysym=%#x up_flag=%d\n",
key_handler[type], keysym, up_flag);

                /*  这就是重点中的重点了，针对不同的键，有不同的操作。比如’a’和方向
                键的操作就是不同的，在 keyboard.c 中我们定义了很多的针对不同的键的
                不同处理函数，并且放在 static k_hand key_handler[16]中，根据 type 我们选
                择相应的处理函数,这些处理函数都带 2 个参数,第一个就是 32 位数 keysym
                的低 16 位，另一个就是 up_flag 标志。这句代码会去调用诸如：do_cur（）
                即方向键的处理代码，do_shift（）即按住 shift 键时的处理代码。 */
                (*key_handler[type])(keysym & 0xff, up_flag);
                if (type != KT_SLOCK)
                    kbd->slockstate = 0;
            } else {
                /* maybe only if (kbd->kbdmode == VC_UNICODE) ? */
                if (!up_flag && !raw_mode)
                    to_utf8(keysym);
            }
        } else {
            /* maybe beep? */
            /* we have at least to update shift_state */
#if 1       /* how? two almost equivalent choices follow */
            compute_shiftstate();
            kbd->slockstate = 0; /* play it safe */
#else
            keysym = U(plain_map[keycode]);
            type = KTYP(keysym);
            if (type == KT_SHIFT)
                (*key_handler[type])(keysym & 0xff, up_flag);
#endif
        }
    }
    rep = 0;
}
```

各种不同的键类型：

```
#define KT_LATIN      0    /* we depend on this being zero */
#define KT_LETTER     11   /* symbol that can be acted upon by CapsLock */
#define KT_FN         1    /*  功能键  */
#define KT_SPEC       2    /*  专用键  */
#define KT_PAD        3    /*  小键盘上的键  */
#define KT_DEAD       4    /*  没有使用过  */
#define KT_CONS       5    /*  没有使用过*/
#define KT_CUR        6    /*  方向键  */
#define KT_SHIFT 7         /* shift 按下时？ */
#define KT_META       8
#define KT_ASCII 9
#define KT_LOCK       10   /*    shift, alt, ctrl  是否被锁住  */
#define KT_SLOCK      12

ushort *key_maps[MAX_NR_KEYMAPS] = {
      plain_map, shift_map, altgr_map, 0,
      ctrl_map, shift_ctrl_map, 0, 0,
      alt_map, 0, 0, 0,
      ctrl_alt_map,   0
};
```

处理函数：

由于处理函数比较多，而且都涉及到 tty 的操作，因此我把处理函数的列表如下：

处理函数	Keysym	Keycode	函数值	对应事件
Do_self	0xf01b	1	0x1b（27）	单击 Esc
	0xf031	2	0x31(49)	单击 1
do_fn	0xf100	59	'\033', '[', '[', 'A', 0,	单击 F1
	0xf101	60	'\033', '[', '[', 'B', 0,	单击 F2
处理函数	Keysym	Keycode	下一处理函数	对应事件
Do_spec	0xf200	0,89-95,120-127	NULL	没有定义
	0xf201	28	Enter()	单击 CR(回车)
	0xf203	70 shift_map 中	show_mem（）	Ctl+scrollloc k
do_pad	0xf300	82	do_fn(KVAL(K_INSERT), 0)	单击 kp_0 numlock 开、关分别处理
			pad_chars[0]=0	
	0xf305	76	applkey('G', vc_kbd_mode(kbd, VC_APPLIC));	单击 kp_5 numlock 开、关分别处理

			pad_chars[5]=5	

三．实验内容和步骤

1. 移植 PS/2 键盘驱动：

我们在以上的键盘代码分析中之所以分析的是 sa1100 体系结构的键盘驱动代码而不完全是我们教学平台所对应的键盘驱动代码，目的就是要大家通过学习后能够自己真正的理解，并达到应用的效果。本试验的内容就是要让大家自己写一个 arm2410 的键盘驱动代码（不参考我们提供的源码，根据上面的文档）。

成功编写这个驱动程序的关键是要认识到和 sa1100 体系结构相比，我们的键盘驱动到底有哪些差别，通过这个试验的实践，以及实践后的认真体会，不但能够真切的理解键盘体系结构，而且更能够感受到嵌入式驱动开发的模式和方法。

键盘驱动和一般的设备驱动不同，它并不提供文件系统的接口，可以说，键盘在 Linux 中与一般的设备管理方法有些差别。一般的设备在 Linux 上都是作为设备文件来管理的，但键盘并不需要做这样的管理，所以在键盘驱动中并没有 register_chrdev。

这里并不需要大家编写应用层的测试程序，有兴趣的同学可以试着编写一个 Linux 下键盘的钩子程序，检测所有的按键事件。

四．思考题

1. 添加对键盘的控制，比如控制 PS/2 键盘的指示灯。
2. 请比较 PS/2 键盘和 USB 键盘的区别。
3. 把键盘输出重定向到 tty

实验二十四： 并口实验

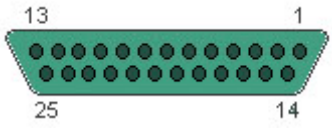
一．实验目的

通过本实验，使同学了解并口的驱动原理，管脚分配，以及作为打印机使用的转发机制。

二．实验原理和说明

1. 基础知识

并口引脚定义



1 ----> #STROBE 数据选通
2 <---> D0 数据位 0
3 <---> D1 数据位 1
4 <---> D2 数据位 2
5 <---> D3 数据位 3
6 <---> D4 数据位 4
7 <---> D5 数据位 5
8 <---> D6 数据位 6
9 <---> D7 数据位 7
10 <---- #ACK 应答脉冲
11 <---- BUSY 忙
12 <---- PE 打印纸尽
13 <---- SLCT 选择
14 ----> #AUTOFD 自动进纸
15 <---- #ERROR 错误
16 ----> #INIT 初始化
17 ----> #SELIN 打印机选择输入
18 ----- GND 信号地
19 ----- GND 信号地
20 ----- GND 信号地
21 ----- GND 信号地
22 ----- GND 信号地
23 ----- GND 信号地
24 ----- GND 信号地
25 ----- GND 信号地

(1) 并口寄存器

并行接口中有 3 个可访问的寄存器:数据端口、状态端口和控制端口。

(a) 数据端口寄存器

CPU 通过这个寄存器与外部设备传送并行数据。寄存器数据在系统初始化过程中被清除 。当 CPU 对该寄存器进行写访问时，该寄存器在 IOW#信号的上升沿处锁存 CPU 的写数据,然后把锁存的写数据输出到 D[0:7]数据线上。 当 CPU 对该寄存器进行读访问时，D[O:7]数据线上的内容经并行接口缓冲(不被锁存)后送入 CPU。

(b) 状态端口寄存器

CPU 通过这个只读寄存器输入外部设备的状态信息，当 CPU 对该寄存器进行读访问时，各对应状态信号线上的现行状态信息锁存于这个寄存器中并送至 CPU。

状态寄存器各位如下所示。

7	6	5	4	3	2	1	0
BUSY#	ACK#	E	SLCT	ERROR#	0	0	0

(i) 位 7 锁存的是 Busy 输入引脚电平的反码，该位为 0 表示打印机为忙状态不能接受新的字符数据；为 1，表示打印机已准备好接受下一字符数据。

(ii) 位 6 锁存的是 ACK#输入引脚的状态,该位为 0 意思是打印机已经收到个字符数据并且可以接受下一个数据了;为 1 意思是打印机还正在处理上一个字符数据或尚未收到数据。

(iii) 位 5 锁存的是 PE 输入引脚的状态， 该位为 1 表示打印纸已用完:为 0 表示还有打印纸。

(iv) 位 4 锁存的是 SLCT 输入引脚的状态， 该位为 1 表示打印机已经联机;为 0 表示打印机未被主机选择。

(v) 位 3 锁存的是 ERROR#输入引脚的状态，该位为 0 表示一个打印机错误已被检测到:为 0 表示没有检测到错误。

(c) 控制端口寄存器

并行接口对打印机输出的各控制信号是通过 CPU 写该寄存器来形成的，即由软件实现控制。当 CPU 对该寄存器进行 I/O 写访问时，该寄存器在 IOW#信号的上升沿处锁存 CPU 的写数据,然后把锁存的写数据输出到打印机的控制信号线上。该寄存器亦可读，故可作为输入外部设备命令信号的端口。并行接口的系统复位输入信号清除该寄存。

控制寄存器的各位如下：

7	6	5	4	3	2	1	0
0	0	CD	IRQE	SLCTIN#	INIT#	AUTOFD	STROBE#

(i) 位 0 写入的逻辑值经取反后输出到 STR0BE#信号线上，向该位写 1，使 STRORE#信号为低电平，它是 D[0:7]的选通信号，把 D[O:7]上的数据 输入外部设备里的数据输入寄存器;向该位写 0，则取消 STROBE#信号。

(ii) 位 1 写入的逻辑值经取反后输出到 AUTOFD#信号线上，向该位写 1，使 AUTOFD#信号线维持低电平，使打印机每打印一行自动走纸一行;向该位写 0,取消自动 走纸。

(iii) 位 2 写入的逻辑值不取反送到 INIT#信号线上， INIT#是打印机要求的初始化信号,打印机收到这个信号后清除打印缓冲区，把内部电路置为初态。向该位写 0，产生 INIT#信号;写 1，取消 INIT#。

(iv) 位 3 写入的逻辑值经取反后输出到 SLCTIN#信号线上，向该位写 1，使 SLCTIN#信号线维持低电平，表示主系统选中打印机，允许打印机工作;向该位写 0，表示不选中打印机。

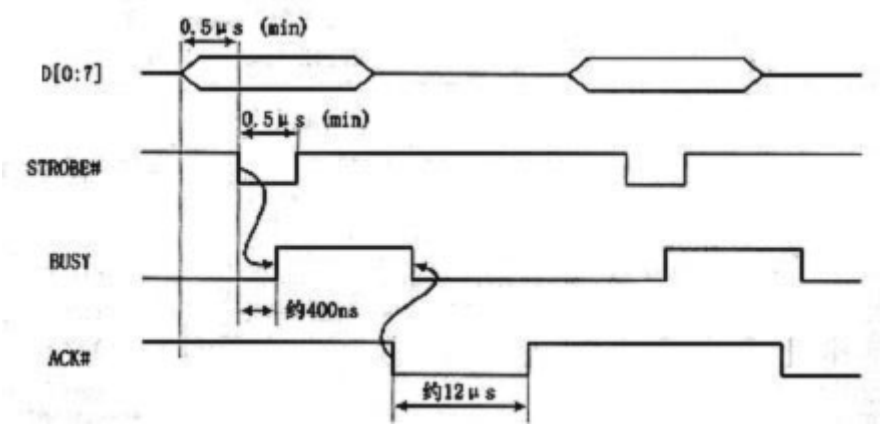
(v) 位 4 是中断请求使能位，用于并行接口内部。 并行接 口的中断请求信号是由 ACK#

输入信号经接口里的非门反相后形成的，即由 ACK#的前沿提出中断请求。该中断请求能否向系统提出由本位控制，本位置为 1， 便能并行接口的中断请求信号。在 ISA 兼容的系统中，并行接口送往主系统的中断请求设置为 IRQ7。当本位编程为 0 时，中断请求被禁止。

(vi) 位 5 是 PCD 位(Parallel Control Direction)， 用于并行接口的内部，是并行接口方向控制位.在打印机方式里，此位无效，因为在打印机方式下不管此位设置为何值方向总是输出的。在双向方式里，本位为 0，端口处于输出模式,写入 为 1 表示端口处于输入模式。

(2) 并行接口信号的时序

在并行接口信号中,有时序关系的信号为 D[O:7],STROBE#、BUSY 和 ACK#，其余为静态的控制或状态信号。从以上的叙述可知,并行接口中除了中断请求信号直接由硬件实现以外，输入/输出数 据、发出选通、检测状态都由程序实现， 可根据所连接的外部设备的实际需要设计时序。下面的时序图是并行接口与打印机之间的信号时序。



2. 扩展应用

2.1 PLIP -- Linux 并口网络解决方案

大家知道，在 DOS 环境下，我们可以用并口或串口将两台 PC 连接起来，一台充当服务器，另一台充当客户，但充当服务器的机器不能做其它操作，只能为 Client 服务。虽然在方便上和速度均不如网卡，但它提供了一个“穷人”的解决方案。如果仅拷贝少量数据，它还是可以满足一般人的需求。并口的速度要远远比串口快。

在 Linux 内核中，网络设备中有一个叫 PLIP (Parallel Line Internet Protocol). 它提供了并口的网络支持，并将并口映射成网络设备。它支持标准并口，扩展并口的支持。传送速度依赖于并口线的质量和机器的配置。

下面，我将系统的配置作简要介绍。

(1) 在内核中支持 PLIP

```
cd /usr/src/linux
make menuconfig
# select Network device support
select PLIP as modules
```

(2) 编译内核

```
cd /usr/src/linux
make dep; make bzImage; make modules; make modules_install;
```

(3) 将新内核配置到 Lilo 中去。
(4) 重新启动
(5) 打开 Ipforward
echo 1 > /proc/sys/net/ipv4/ip_forward
(6) 运行网络配置工具，配置 PLIP
turbonetcfg
add new interface
select PLIP
add the ipaddress and mask to that
save & exit
(7) 启动 PLIP
modprobe plip
if it does not work
echo 7 > /proc/parport/0/irq
modprobe plip
ifup plip0
(8) 配置网关。
eg. Machine A:
PLIP0 -- 10.0.0.1
B -- gateway
PLIP0 -- 10.0.0.2
eth0 -- 172.16.69.12
in B Machine, we setup ipchains
/sbin/ipchains -A forward -j MASQ -s 10.0.0.0/255.255.255.0 -d 0.0.0.0/0

通过我们的试验，使用 WWW, TELNET 与普通网卡没有区别，但当使用 FTP 时，速度稍慢，大约 35K/s 左右，并且在 FTP 的同时，我们也发现系统偶尔出现 Timeout, 并且当你作其它事情时，感到系统很慢。说 明 PLIP 的驱动程序还需要改进。

参考资料：/usr/src/linux/Documentation/networking/PLIP.txt

2.2 配置并口实现 windows 端到 linux 的网络打印
(1) windows 端的程序流程
(a) 产生一个 DC，画一个设备到 DC 上，等待打印程序调用该 DC。
(b) GDI 询问打印设备，询问如何描绘要打印的东西。
(c) 打印机反馈给程序，如何打印。
(d) Win95 为 32 位打印，调用 GDI32。
(e) GDI32 调用 spooler 进程 SPOOL32.EXE。
(f) 调用 Router 决定，往哪里发送该 print job。（通常为本地）
(g) 调用 LocalPrint Provider 执行该 Print job。
(h) 将要打印的东西写入一个文件.
(i) 后台执行一个进程在合适的时候，将该文件发送到打印机。
(j) 主线程改变监听器的状态，在适当的时候调用 startdoc 函数来开始打印。
(k) 唤醒端口，用 readprinter 读出打印内容
(l) writeport 向物理端口写入数据。

(m) 将数据送入打印机。

(n) 判断是网络打印还是本地打印，决定怎样传送。我们的应用程序就是在这一步进行处理，截取要传输的数据，然后直接传送到 linux 服务器当中，进行网络打印。

注意：

我们的应用程序进行处理的就是在数据通过端口要传送到打印机的时候，截取要传输的数据，然后直接传送到 linux 服务器当中，进行网络打印。

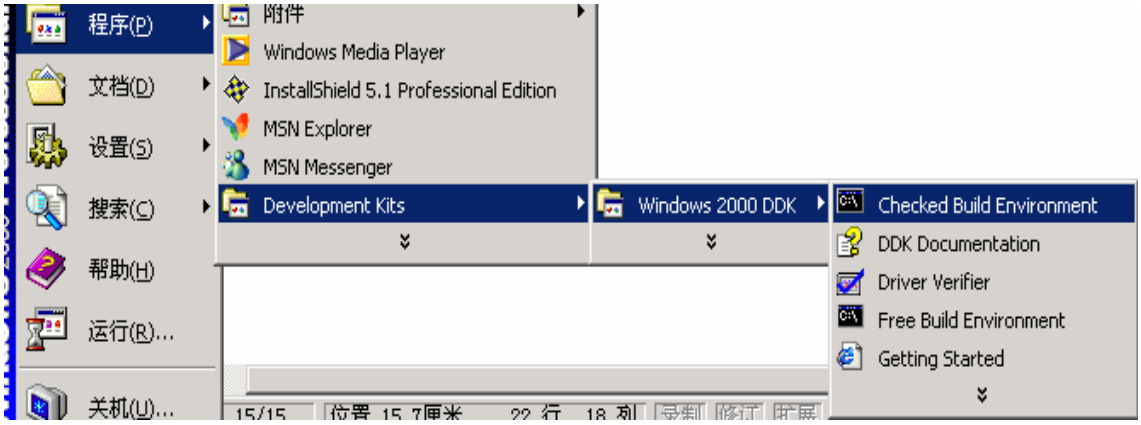
以上步骤是 windows 打印机的工作原理，我们的打印机 windows 端的工作就是在打印机驱动程序就要将数据写入端口的时候，截取这些数据，然后通过网络传送给 linux 服务器；Linux 端将数据送入和并口相连的打印机中就可，我们截取的数据是送入端口时的数据，是已经转换成打印码的数据，我们只需要将这些数据直接送到打印机即可，不需要再进行打印码的转化。

鉴于支持 lpd 协议的端口驱动比较难做，而且有些不支持一些特殊的打印机，我们没有采用 lpd 协议，只是把要打印的数据直接送到 linux 端打印机中，避免了 lpd 通信中可能造成的信息交换错误。而采用软件检测打印数据是否已经完整。(通过检测 socket 的断开来实现)。程序代码的修改主要集中在 localmon.c 和 dialogs.c 中，非说明者均在 localmon.c 中。

(2) WINDOWS 端软件的编译问题：

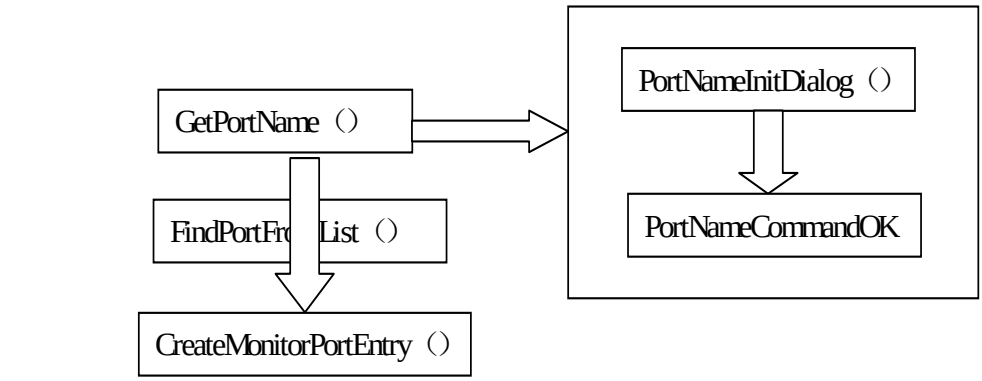
(a) 要安装 VC6＋对应 OS 的 DDK。

(b) 要设置 DDK 环境并编译 DDK，即选择“checked Build Environment”，会自动弹出一个 DOS COMMAND 窗口（该窗口见下图 2），它设置环境变量，但仅在此 COMMAND 窗口内有效，退出该窗口就失效，就必须再重来一次。这时要 cd bin，然后敲入 build 命令，它编译 DDK。然后才能编译自己的代码。进入自己的目录，执行 build 即可进行编译。



(3) 流程简介：

添加端口: LocalmonAddPort（）：



打印预处理 LocalmonStartDocPort（）：

socker()

PortNameCommandOK

FindPortFromList（）

CreateMonitorPortEntry（）

Connect()

传送数据：LocalmonWritePort（）：

Send（）

打印完成时 LocalmonEndDocPort（）：

Closesocket（）

(a) 详细说明：

(i) 添加端口时，调用 LocalmonAddPort（），其中 GetPortName（）函数获取用户输入的端口名称和服务器的 IP。

(ii) GetPortName（）函数，调用 DialogBoxParam（）函数产生对话框来让用户数据需要的数据，该对话框的资源在 localmon.rc 中，可以利用 vc 进行编辑，很方便。

① 对话框初始化时，调用 PortNameInitDialog（），（这部分在 dialogs.c 中），保存传进来的参数（数据指针），对对话框进行初始化的限制（文本框长度限制）。

② 当用户确定输入后，调用 PortNameCommandOK（）函数，获取传进来的参数（指针），然后获取窗口值，（端口名称，服务器 IP 地址），校正后，返回用户的输入。

(iii) 校验是否该端口名称已经存在 FindPortFromList（），存在返回错误退出。否则继续。

(iv) 产生这个端口 CreateMonitorPortEntry（），程序从打印机驱动程序获取已经被翻译成打印码的数据，然后发送到网络中即可。

(v) 有打印任务时，系统自动调用 LocalmonStartDocPort(….),在该部分的代码为：IsFileName：以后的部分，该部分的代码的功能：初始化 socket，建立本地 socket，输入服务器的参数，建立网络连接，准备打印。

(vi) 当打印打印数据开始传送时，调用 LocalmonWritePort（…），该部分代码的作用，接受到打印机驱动器传来的数据后，用 LocalmonStartDocPort（）中建立的 socket，将数据传送到网络服务器中。

(vii) 当打印任务完成后，调用 LocalmonEndDocPort（），关闭 LocalmonStartDocPort（）所建立的 socket。退出打印。

(b) 问题与解决

(i) IP 的存储问题

① 因为是网络打印，所以需要知道服务器的 IP，端口为 515，按照端口的一般写法，（参照其他的端口协议），把 IP 写在名称的后面，以‘@’区分。例如：ruiji@192.168.2.1，所以端口名中不能出现@符号。若名称中出现‘@’时，用‘_’代替。

② 将用该端口的 hFile 保存生成的 socket，写数据的时候，用该 hFile 作为 socket 来传送数据。

(ii) 未知的本地端口问题

① 生成的 dll 文件运行后，添加的端口的端口类别显示为未知的本地端口。

② 在程序中 LocalmonAddPort（）添加端口时，调用 CreateMonitorPortEntry（）来保存端口，其中第二个参数为端口的类别，默认为“unknown local port”，（系统的 rc 文件中），将其改为“RJtech Port”即可。

(iii) 系统重新启动后，上次生成的端口即不可用

① 系统默认的端口放置位置为 win.ini 文件的 ports 节中，感觉放在这个节中的端口系统会接手管理，而不调用生成的 dll 文件中的函数来管理，这样就不会调用 CreateMonitorPortEntry（）来生成端口，这样系统默认该端口为本地端口，并按文件打印处理。

② 我后来将所有的端口放入 RJtechPorts 节中，系统不认识这个节，就会在 dll 中查找，于是在 wininiport 中就会生成所需要的端口，和系统中的端口对应起来。打印时可以找到。

(iv) 容错问题

① windows 端：由于我们的程序是在打印机驱动程序的最接近端口的地方，驱动的错误通常不会到达这里。Startdoc 和 Writeport 以及 enddoc 中返回值为 BOOL,如果返

回 FALSE，程序会自动 弹出出错的提示信息。Socket 中的 connect 函数执行 tcpip 协议，会自动连接，直到数据传送成功为止，若网络出错，程序返回 FALSE。

② Linux 端：程序从网络接收数据，然后送到打印驱动当中，当连接存在的时候，就会不停的接收数据来打印。当一个打印任务完成的时候，windows 端会自动关闭 socket，此时 linux 端的 read socket 超时后返回－1，退出数据接收，完成本次打印。超时是用 alarm 做的，设定超时时间为 10 秒，可以更改全局定义变量 READTIMEOUT 来更改超时的时间。当用户取消打印或者网络出错的时候，退出接收数据，完成本次打印。

(v) 未知的遗留问题

① 设有 A，B 两台机器，A 机器上打一个大文件 1，正在打印时，B 机器上打印一个大文件 2，在队列中等待，这时，A 机器再打印一个文件 3（文件 1 正在打印，没有打印完），这时，取消打印文件 1，然后取消打印文件 2，这时，在打印文件 3 的时候，会显示写入打印文件出错。

② 考虑到该 bug 并不影响打印，出错后，你继续打印就可以了，而且这个无法定位是什么错误，就没有解决。（现在已经解决，不再出现了。）

(vi) Windows 重新启动后端口不可用

与 98 的解决方案差不多，将标志端口的全局变量 szPorts 的值由“Ports”改为“RjtechPorts”即可，这样系统存储该端口的时候，就不会与其它的本地端口存储到一起，也就不会将其认为是本地端口，同时我做的另一处修改就是将判断端口是否有效的函数 PortIsValid（）返回值始终定为 TURE，这样就不会因为端口校验不合格而使端口不可用。

当做完同 WinNt 同样的修改之后，安装端口的时候，提示 Monitor 安装不上，同时提示版本不对。真正的 bug 在于端口初始化的时候在函数 LocalMonInitializePrintMonitor2（）当中，对端口进行初始化，调用 GetProfileString（）函数找出已经存在的端口，并向 windows 添加端口，这段代码如下：

```
rc = GetProfileString(szPorts, NULL, szNULL, pPorts, dwCharCount);//初始化
        if ( !rc || dwCharCount >= 1024*1024 ) {
            DBGMSG(DBG_ERROR,
                    ("GetProfilesString failed with %d\n", GetLastError()));
            goto Fail;
        }
```

可以看出，这段代码认为在系统中没有端口存在是错误的，goto fail,fail 中的代码有：return NULL；系统认为没有端口就初始化失败，于是提示上述出错信息。我修改了上述代码，初始化时增加一个默认的端口 Rjtech@192.168.2.1，然后再查找端口，这样至少能找到该端口，就不会出错，而继续执行，修改后的程序如下：

```
WriteProfileString(szPorts, L"RJtech@192.168.2.1", L"");
rc = GetProfileString(szPorts, NULL, szNULL, pPorts, dwCharCount);
if ( !rc ) {
    DBGMSG(DBG_ERROR,("GetProfilesString failed with %d\n", GetLastError()));
    FreeSplMem((LPVOID)pPorts);
    return FALSE;
}
```

可以成功执行。

(vii) 打印出错后，占用资源太多

置变量，当出错后，睡眠的时间定为原来的两倍，效果好一些了。这段代码如下：

```
int printerror;
printerror = 0;//打印一个字符不成功以后，初始化该值
//出错以后，设置该变量的值
//举例代码如下
if (!(status & LP_PSELECD)) {
    printerror = 1;
    printk(KERN_INFO "lp%d off-line,cnt=%d,sts=%02x\n",
minor,bytes_written,status);
    if (LP_F(minor) & LP_ABORT)
        return rc?rc:-EIO;
}
//sleep 时判断是否出错，若出错，增加 sleep 的时间
current->timeout = jiffies + LP_TIMEOUT_INTERRUPT;
if(printerror){
    current->timeout += (0.3 * HZ);
}
sti();
interruptible_sleep_on(&lp->lp_wait_q);
```

(3) Linux 端

(a) 建立 socket，填入连接端口。

(b) 打开打印设备/dev/lp0。

(c) listen，进行监听。

(d) 当网络有数据来时，进行打印。For 循环。

(i) 初始化打印机。user_lp_reset()；

(ii) 建立连接。Select() ，accept()；

(iii) 从网络接受数据，送入打印设备。Read()， Write()。

(e) 程序执行结束后，关闭打印设备。

(4) 中断方式与轮循方式

(a) 采用中断方式，中断号为 2，而网络的中断号为 3，这样，打印的中断将优先于网络的中断，网络较慢，而且，打印的时候，中断非常频繁，大量的中断将击溃系统，导致网络几乎死掉。

(b) 解决方案：屏蔽中断，因为中断太频繁，直接采用轮循，完全占用内核，这样可以打印，而且网络采用中断方式，可以中断内核的执行，这样打印的数据可以通过网络传输过来，可以完成打印。屏蔽中断的代码在 lp.c 中如下所示，注释掉下列部分即可。

```
LP_STAT(minor).sleeps++;
cli();
outb_p(0x05, (LP_C(minor)));
status = LP_S(minor);

if (((status & LP_PBUSY))&& LP_CAREFUL_READY(minor, status)) {
    outb_p(0x05, (LP_C(minor)));
    sti();
```

```
}

lp_table[minor].runchars=0;
current->timeout = jiffies + LP_TIMEOUT_INTERRUPT;
sti();
interruptible_sleep_on(&lp->lp_wait_q);
```

(c) 修补方案：因为打印在内核态完成，所以这时候用户态的响应很慢，导致用户的程序几乎瘫痪。我们采用灵活的轮循方式，当很长时间都忙时，就主动放弃一段 cpu 时间，这样回到用户态，其它的用户程序就可以执行了。

这种方案不需要注释上面的部分，只需要更改 lp.h 中中断 sleep 的时间即可。

```
#define LP_TIMEOUT_INTERRUPT       (0.2 * HZ)
```

1HZ 表示 1 秒，0.2*HZ 表示 0.2 秒，你可以更改这个值。

如果将来需要改成中断方式，需要恢复 static int lp_open(struct inode * inode, struct file * file) 函数中注释掉的部分 //ret = request_irq(irq, lp_interrupt, SA_INTERRUPT, "printer",NULL);

```
/*if (ret) {
        kfree_s(lp_table[minor].lp_buffer, LP_BUFFER_SIZE);
        lp_table[minor].lp_buffer = NULL;
        printk("lp%d unable to use interrupt %d, error %d\n", minor, irq, ret);
        MOD_DEC_USE_COUNT;
        return ret;
}
else{
        //*(volatile unsigned long *)(0x10000020) |= 0x0f000000;
        //udelay(1000);
        //*(volatile unsigned long *)(0x10000034) |= 0x40000000;
        //udelay(1000);
        //printk("irq2 init ok\n");
}*/
```

同时在中断到来的以后关中断即可。

static void lp_interrupt(int irq, void *dev_id, struct pt_regs *regs)函数中

```
//disable irq
*(volatile unsigned long*)(0x10000020) = 0x08000000;
*(volatile unsigned long*)(0x10000020) &= 0xf0ffffff;
```

3. 内核代码：（LINUX 端驱动部分）

(1) 变量说明

```
#define LP_INIT_CHAR 1000 //该变量表示每次打印，若打印机忙或者其他原因不能打印需要等待的循环次数
#define LP_INIT_WAIT 1 //表示时序的间隔周期
#define LP_TIMEOUT_INTERRUPT       (1 * HZ)     //表示每次打印机忙时，sleep 的最大时间，默认值为 60hz,可以进行调整
#define LP_S(minor)        ((*(volatile unsigned char*)(LP_B((minor))+7) )^0x01)     //取状态位的信息，因为 busy 为反相输入，则忙位取反 0x70000b
```

#define LP_C(minor)　　(lp_table[(minor)].base+7)　　//控制位的地址

#define LP_PBUSY　　0x01　/* inverted input, active high */　　不忙为 0
#define LP_PACK　　0x40　/* unchanged input, active low *///该位其实没有用到，因为是延触发，该值不可能实时取得
#define LP_POUTPA　　0x02　/* unchanged input, active high */　　有纸为 0
#define LP_PSELECD　　0x04　/* unchanged input, active high */　　连线状态为 1
#define LP_PERRORP　　0x08　/* unchanged input, active low */　　无错为 1
//上述为状态位，16 位地址 0x200000a 的低 8 位
(a)　状态线（用读该地址方式获得）
16位数据总线的倒数第1位(最低位)为打印口：BUSY信号;
16位数据总线的倒数第2位为打印口：Paper end信号;
16位数据总线的倒数第3位为打印口：Select信号;
16位数据总线的倒数第4位为打印口：/Error信号;

#define LP_PINTEN　　0x10　/* high to read data in or-ed with data out */ //没有用到
#define LP_PSELECP　　0x08　/* inverted output, active low */　　连线为 0
#define LP_PINITP　　0x04　/* unchanged output, active low */　　正常为 1；用 0 初始化
#define LP_PAUTOLF　　0x02　/* inverted output, active low */ 怀疑没有用，0 和 1 都可以正常打印
#define LP_PSTROBE　　0x01　/* short high output on raising edge */
1 0 1 的时序使打印机字节输入
(b)　控制线（用写入该地址方式达得）
注明：写入后各个BIT将保留写入时的状态直到下次写入；
16位数据总线的倒数第1位(最低位)为打印口：/STROBE信号;
16位数据总线的倒数第2位为打印口：/Auto-Linefeed
16位数据总线的倒数第3位为打印口：/Initialize
16 位数据总线的倒数第 4 位为打印口：/Select-In

struct lp_struct lp_table[] = {{ 0x700004,66, 0, LP_INIT_CHAR, LP_INIT_TIME, LP_INIT_WAIT, NULL, NULL,0, 0, 0, {0} },
};打印机的基地址为 0x2000004,中断号 2
打印数据输出：　读取或写入该片选的基地址加0100 B之地址；
16位数据总线的高字节的倒数第1位(最低位)为打印口：DATA0;
16位数据总线的高字节的倒数第2位(最低位)为打印口：DATA1;
16位数据总线的高字节的倒数第3位(最低位)为打印口：DATA2;
16位数据总线的高字节的倒数第4位(最低位)为打印口：DATA3;
16位数据总线的高字节的倒数第5位(最低位)为打印口：DATA4;
16位数据总线的高字节的倒数第6位(最低位)为打印口：DATA5;
16位数据总线的高字节的倒数第7位(最低位)为打印口：DATA6;
16位数据总线的高字节的倒数第8位(最低位)为打印口：DATA7;

(2)　程序流程：

lp_init(),设置片选

(volatile unsigned short)(0x10000060) = 0x0200;

(volatile unsigned short)(0x10000062) = 0x0201;

//udelay(0);

(volatile unsigned short)(0x100000644) = 0xffff;

(volatile unsigned short)(0x10000066) = 0xf078;

此处若启动失败，可以在前后 udelay(1000),成功的范例是：前面 1，后 80 或者前 3 后 10 个 udelay(1000),前后各一个 udelay(1000)也成功过。Lp_probe 中调用 lp_reset 初始化（选中和初始化）。当有字符送来打印：调用 lp_write()➔lp_write_interrupt()➔lp_char_interrupt()。若成功则下一个，不成功，出错退出，否则，sleep 等待中断来到，或者时间到了，就继续打印。如此循环，直到打印完成。

输出一个字节打印数据到端口：

(a) Write the byte to the Data Port.

(b) Check to see is the printer is busy. If the printer is busy, it will not accept any data, thus any data which is written will be lost.

(c) Take the Strobe (Pin 1) low. This tells the printer that there is the correct data on the data lines.

(d) Put the strobe high again after waiting approximately 5 microseconds after putting the strobe low. (Step 3)

(e) Once the printer has accepted data, it will acknowledge the byte by a negative pulse about 5uS on the /Ack line.

(3) 管脚说明：硬件的 data2 和 data3 脚反了

(a) 控制位：　通常状态：0101

需要初始化时：0001

打印一个字符：0100　然后　0101

(b) 状态位：　通常 1101(busy 被取反了)

忙：1100

出错：0101

无纸：0111

4. 硬件测试指南

(1) get 10000040 80 启动后，使用该命令，你应该看到数据第三行为 0200 0201 ffff f078，如果你看到这些，就说明片选没有问题！

(2) Put 700004 0000 该命令使打印机数据位全部清零，可以简单测试硬件是否通！可用示波器检查该结果是否被正确传到并口。

(3) Put 2000004 ffff 该命令使打印机数据位全部为 1。

(4) Put 2000004 0100 data0 付值为 1 ，这以下的 16 条命令是精确检查每一位，是否能准确工作，比如是否两线颠倒等等！敲入这些命令后，需要用示波器检查相应的数据位是否有相应的变化。

(5) Put 2000004 feff　　data0 赋值为 0

(6) Put 2000004 0200　　data1 赋值为 1

(7) Put 2000004 fdff　　data1 赋值为 0

(8) Put 2000004 0400　　data2 赋值为 1

(9) Put 2000004 fbff data2 赋值为 0
(10) Put 2000004 0800 data3 赋值为 1
(11) Put 2000004 f7ff data3 赋值为 0
(12) Put 2000004 1000 data4 赋值为 1
(13) Put 2000004 efff data4 赋值为 0
(14) Put 2000004 2000 data5 赋值为 1
(15) Put 2000004 dfff data5 赋值为 0
(16) Put 2000004 4000 data6 赋值为 1
(17) Put 2000004 bfff data6 赋值为 0
(18) Put 2000004 8000 data7 赋值为 1
(19) Put 2000004 7fff data7 赋值为 0
(20) Put 200000a 0001 strore 位置 1
(21) Put 200000a fffe strore 位置 0
(22)Put 200000a 0002 auto_linefed 位置 1
(23)Put 200000a fffd auto_linefed 位置 0
(24) Put 200000a 0004 Initlised 位置 1
(25) Put 200000a fffb Initlised 位置 0
(26) Put 200000a 0008 select_in 位置 1
(27)Put 200000a fff7 select_in 位置 0

(28)Get 200000a 2 状态位无法输出，只能输入，通常该命令得到的结果是 50 8d,其中的 d 包含四位状态位的信息，从高位到低位依次是：error,select,paper_end,busy,你可以用 5.0v 电压抬高这些位或者用地来降低这些位，然后观察 Get 70000a 2 得到的信息，看看你是否得到正确的输入！

(29)Put 200000a 1;put 200000a 5;get 200000a 2 当已经连接打印机（加电）后，执行这三条命令后，你应该得到的是 50 8c ，如果你没有得到该信息，那么可能是硬件错，或者你没有把打印机的线插好！

(30) 综合测试

无法支持 HP LaserJet 6L Pro PCL5e 型激光打印，起初是打印时有响应，只不过打出来的没有东西，是白纸。但现在的现象是打印后打印机根本不响应，没有任何动作。根据现象应该是：打印数据并没有传送到打印机端口。最后证明是硬件并口电平不够，现在是 3V，而并口规范要求 5V，改正后一切正常。

三．实验内容和步骤

1. 低级操作，实现读写并口。
2. 高级操作，手工输入，使打印机打印。

四．思考题

1．分析打印机打印时出现乱码可能的原因。

实验二十五： 音频输入输出实验

一．实验目的

通过本实验，使学生了解音频控制芯片 UDA1380，了解 S3C2410 音频驱动的原理，了解 S3C2410 放音和录音的工作原理，并对 S3C2410 音频驱动加以改进。

二．实验原理和说明

S3C2410 通过 IIS（Inter-IC Sound）总线与音频控制芯片 UDA1380 进行通信。放音时发送数据到 UDA1380 的 DATAI 管脚，录音时从 UDA1380 的 DATAO 管脚接收数据，其数据传输方式为 DMA 方式。

1．S3C2410 音频驱动 ：

S3C2410 的音频驱动 2410audio.o 由 2410audio.c 来实现，放音时调用函数 smdk2410_audio_write 来对外发送数据，录音时通过函数 smdk2410_audio_read 接收数据。

在 S3C2410 音频驱动加载时，需要：

(1) 设置 GPI/O 模式，并初始化 IIS 总线；
(2) 通过 I2C 对 UDA1380 的各个寄存器进行初始化，使 UDA1380 可正确工作；
(3) 设置数据输入和输出的 DMA 通道，并发出 DMA 请求；
(4) 申请音频设备。

在驱动卸载时，必须：

(1) 将申请的音频设备注销；
(2) 申请的 DMA 通道释放。

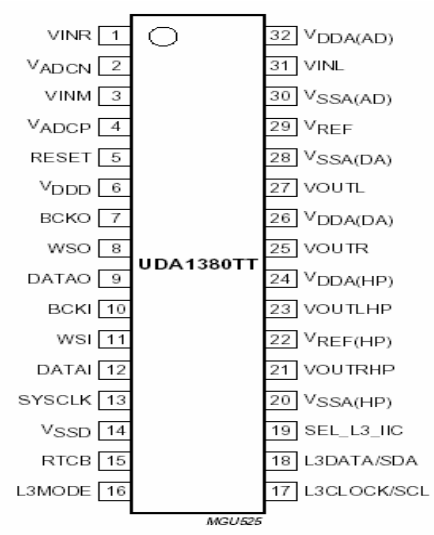
驱动加载、卸载代码分析（放音驱动）：

```
int init_module(void)
{
    unsigned long flags;
    int cold = !audio_active;

    local_irq_save(flags);

    /*以下开始初始化GPIO*/
    /* GPE 3: I2SSDI */
    set_gpio_ctrl(GPIO_E3|GPIO_PULLUP_EN|GPIO_MODE_I2SSDI);

    /* GPE 0: I2SLRCK */
    set_gpio_ctrl(GPIO_E0|GPIO_PULLUP_EN|GPIO_MODE_I2SSDI);

    /* GPE 1: I2SSCLK */
    set_gpio_ctrl(GPIO_E1|GPIO_PULLUP_EN|GPIO_MODE_I2SSCLK);

    /* GPE 2: CDCLK */
```

```
    set_gpio_ctrl(GPIO_E2|GPIO_PULLUP_EN|GPIO_MODE_CDCLK);

    /* GPE 4: I2SSDO */
    set_gpio_ctrl(GPIO_E4|GPIO_PULLUP_EN|GPIO_MODE_I2SSDO);

    local_irq_restore(flags);

    uda1380_init();                          //初始化uda1380

    init_s3c2410_iis_bus();                  //初始化IIS

   output_stream.dma_ch = DMA_CH2;     //设置放音DMA通道为DMA_CH2

    /*下面申请DMA，这是通过audio_init_dma函数实现*/
    if (audio_init_dma(&output_stream, "UDA1380 out")) {
        audio_clear_dma(&output_stream);
       printk( KERN_WARNING AUDIO_NAME_VERBOSE": unable to get DMA
              channels\n" );
        return –EBUSY;
    }
    /*申请放音设备：/dev/sound/dsp*/
    audio_dev_dsp = register_sound_dsp(&smdk2410_audio_fops, -1);
    printk("Audio play initialized.\n");          //打印信息
    /*初始化放音频率等*/
    if (cold) {
        audio_rate = AUDIO_RATE_DEFAULT;
        audio_channels = AUDIO_CHANNELS_DEFAULT;
        audio_fragsize = AUDIO_FRAGSIZE_DEFAULT;
        audio_nbfrags = AUDIO_NBFRAGS_DEFAULT;
        audio_clear_buf(&output_stream);
   }
     return 0;
}

void cleanup_module(void)
{
      unregister_sound_dsp(audio_dev_dsp);     //注销放音设备
      audio_clear_dma(&output_stream);         //释放DMA通道
      printk("Audio play unloaded.\n");        //打印信息
}
```

2．UDA1380：

(1) UDA1380 的管脚配置 ：



Pin configuration UDA1380TT

(2) UDA1380 寄存器：

寄存器地址	功能
System settings（runninng at the IIS-bus or L3-bus clock itself）	
00H	Evaluation modes，WSPLL settings，clock divider and clock selectors
01H	IIS-bus I/O settings
02H	Power control settings
03H	Analog mixer settings
04H	Reserved
Interpolation filter（running at 128fs interpolator clock）	
10H	Master volume control
11H	Mixer volume control
12H	Mode selection，left and right bass boost，and treble settings
13H	Master mute，channel1 and channel2 de-emphasis and channel mute
14H	Mixer，silence dector and interpolation filter oversampling settings
Decimator（running at 128fs decimator clock）	
20H	Decimator volume control
21H	PGA settings and mute
22H	ADC settings
23H	AGC settings

Register map of control settings

(3) UDA1380 各寄存器的具体设置：

(a) 赋值模式以及时钟设置：

BIT	15	14	13	12	11	10	9	8
Symbol	EV2	EV1	EV0	—	EN_ADC	EN_DEC	EN_DAC	EN_INT

Default	0	0	0	0	0	1	0	1
BIT	7	6	5	4	3	2	1	0
Symbol	—	—	ADC_CLK	DAC_CLK	Sys_div1	Sys_div0	PLL1	PLL0
Default	0	0	0	0	0	0	1	0

Register address：00H

BIT	SYMBOL	描述
15 – 13	EV[2:0]	Evaluation bits。Default value 000。
12	—	Default value 0
11	EN_ADC	ADC clock enable。The bit is logic 0：clock to ADC disable、the bit is logic 1：clock to ADC running。Default value 0。
10	EN_DEC	Decimator clock enable。The bit is logic 0：clock to decimator disable、the bit is logic 1：clock to decimator running。Default value 1。
9	EN_DAC	FSDAC clock enable。The bit is logic 0：clock to FSDAC disable、the bit is logic 1：clock to FSDAC running。Default value 0。
8	EN_INT	Interpolator clock enable。The bit is logic 0：clock to interpolator disable、the bit is logic 1：clock to interpolator running。Default value 1。
7 – 6	—	Default value 00。
5	ADC_CLK	ADC clock select。The bit is logic 0：SYSCLK is used、the bit is logic 1：WSPLL is used。Default value 0。
4	DAC_CLK	DAC clock select。The bit is logic 0：SYSCLK is used、the bit is logic 1：WSPLL is used。Default value 0。
3 – 2		Divider for system clock input。Default value 00。
1 – 0		WSPLL settings。Default value 10。

寄存器详细描述

Sys_div1	Sys_div0	Input clock on pin sysclk
0	0	256fs（default）
1	1	384fs
1	0	512fs
1	1	768fs

Dividers for system clock input

PLL1	PLL0	Inter frequency range(kHz) on pin wsi
0	0	6.25 to 12.5
0	1	12.5 to 25
1	0	25 to 50（default）
1	1	50 to 100

WSPLL settings

(b) IIS 总线输入、输出设置：

BIT	15	14	13	12	11	10	9	8
Symbol	—	—	—	—	—	SFORI2	SFORI1	SFORI0

Default	0	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	—	SEL_SOURCE	—	SIM	—	SFORI2	SFORI1	SFORI0
Default	0	0	0	0	0	0	0	0

Register address：01H

BIT	SYMBOL	描述
15 – 11	—	Default value 00000。
10 – 8	SFORI[2:0]	Digital data input formats。 Default value 000。
7	—	Defaulr value 0。
6	SEL_SOURCE	Digital output interface mode settings。The bit is logic 0：source digital output interface mode set to decimator、the bit is logic 1：source digital output interface mode set to digital mixer output。 Default value 0。
5	—	Default value 0。
4	SIM	Digital output interface mode settings。The bit is logic 0：mode of digital output interface is set to slave、the bit is logic 1：mode of digital output interface is set to master。Default value 0。
3	—	Default value 0。
2 – 0	SFORI[2:0]	Digital data output fotmats。Default value 000。

寄存器详细描述

SFORI2	SFORI1	SFORI0	Serial_format_DAI
0	0	0	IIS-bus（default）
0	0	1	LSB-justified，16 bits
0	1	0	LSB-justified，18 bits
0	1	1	LSB-justified，20 bits
1	0	1	MSB-justified
Others			Not used：mapped to IIS-bus。

Digital data input formats

SFORI2	SFORI1	SFORI0	Serial_format_DAO
0	0	0	IIS-bus（default）
0	0	1	LSB-justified，16 bits
0	1	0	LSB-justified，18 bits
0	1	1	LSB-justified，20 bits
SFORI2	SFORI1	SFORI0	Serial_format_DAO
1	0	0	LSB-justified，24 bits
1	0	1	MSB-justified
Others			Not used：mapped to IIS-bus。

Digital data output formats

(c) 电源控制设置：

BIT	15	14	13	12	11	10	9	8
Symbol	PON_PLL	—	PON_HP	—	—	PON_DAC	—	PON_BIAS
Default	0	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	EN_AVC	PON_AVC	—	PON_LNA	PON_PGAL	PON_ADCL	PON_PGAR	PON_ADCR
Default	0	0	0	0	0	0	0	0

Register address：02H

BIT	SYMBOL	描述
15	PON_PLL	Power-on WSPLL。The bit is logic 0：power-off、the bit is logic 1：power-on。Default value 0。
14	—	Default value 0。
13	PON_HP	Power-on headphone driver。The bit is logic 0：headphone driver is power-off、the bit is logic 1：headphone driver is power-on。Default value 0。
12 – 11	—	Default value 00。
10	PON_ADC	Power-on DAC。The bit is logic 0：DAC is power-off、the bit is logic 1：DAC is power-on。Default value 0。
9	—	Default value 0。
8	PON_BIAS	Power-on BIAS。The bit is logic 0：ADC、AVC、FSDAC bias circuits are power-off、the bit is logic 1：power-on bias for ADC、AVC、FSDAC。Default value 0。
7	EN_AVC	Enable control AVC。The bit is logic 0：analog mixer is disabled、the bit is logic 1：analog mixer is enabled。Default value 0。
6	PON_AVC	Power-on AVC。The bit is logic 0：analog mixer power-off、the bit is logic 1：analog mixer power-on。Default value 0。
5	—	Default value 0。
4	PON_LNA	Power-on LNA。The bit is logic 0：LAN and SDC are power-off、the bit is logic 1：LAN and SDC are power-on。Default value 0。
3	PON_PGAL	Power-on PGAL。The bit is logic 0：left PGA is power-off、the bit is logic 1：left PGA is power-on。Default value 0。
2	PON_ADCL	Power-on ADCL。The bit is logic 0：left ADC is power-off、the bit is logic 1：left ADC is power-on。Default value 0。
1	PON_PGAR	Power-on PGAR。The bit is logic 0：right PGA is power-off、the bit is logic 1：right PGA is power-on。Default value 0。
0	PON_ADCR	Power-on ADCR。The bit is logic 0：right ADC is power-off、the bit is logic 1：right ADC is power-on。Default value 0。

寄存器详细描述

(d) 模拟混音器设置：

BIT	15	14	13	12	11	10	9	8
Symbol	—	—	AVCL5	AVCL4	AVCL3	AVCL2	AVLC1	AVCL0
Default	0	0	1	1	1	1	1	1
BIT	7	6	5	4	3	2	1	0
Symbol	—	—	AVCR5	AVCR4	AVCR3	AVCR2	AVCR1	AVCR0
Default	0	0	1	1	1	1	1	1

Register address：03H

BIT	SYMBOL	描述
15 – 14	—	Default value 00。
13 – 8	AVCL[5:0]	Analog volume control。Default value 111111。
7 – 6	—	Default value 00。
5 – 0	AVCR[5:0]	Analog volume control。Default value 111111。

寄存器详细描述

AVCL5 AVCR5	AVCL4 AVCR4	AVCL3 AVCR3	AVCL2 AVCR2	AVCL1 AVCR1	AVCL0 AVCR0	VOLUME（dB）
0	0	0	0	0	0	16.5
0	0	0	0	0	1	15
0	0	0	0	1	0	13.5
0	0	0	0	1	1	12
0	0	0	1	0	0	10.5
……	……	……	……	……	……	……
1	0	1	0	1	1	-48
1	0	1	1	0	0	-∞
……	……	……	……	……	……	……
1	1	1	1	1	1	-∞（default）

Analog volume control

(e) 保留位设置：

BIT	15	14	13	12	11	10	9	8
Symbol	—	—	—	—	—	RSV12	RSV11	RSV10
Default	—	—	—	—	—	0	1	0
BIT	7	6	5	4	3	2	1	0
Symbol	—	—	—	—	—	RSV02	RSV01	RSV00
Default	—	—	—	—	—	0	1	0

Register address：04H

BIT	SYMBOL	描述
15 – 11	—	Not used
10	RSV12	Reserved bit。Default value 0。
9	RSV11	Reserved bit。Default value 1。

8	RSV10	Reserved bit。Default value 0。
7 – 3	—	Not used
2	RSV02	Reserved bit。Default value 0。
1	RSV01	Reserved bit。Default value 1。
0	RSV00	Reserved bit。Default value 0。

寄存器详细描述

(f) 放音音量控制：

BIT	15	14	13	12	11	10	9	8
Symbol	MVCR_7	MVCR_6	MVCR_5	MVCR_4	MVCR_3	MVCR_2	MVCR_1	MVCR_0
Default	0	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	MVCL_7	MVCL_6	MVCL_5	MVCL_4	MVCL_3	MVCL_2	MVCL_1	MVCL_0
Default	0	0	0	0	0	0	0	0

Register address：10H

BIT	SYMBOL	描述
15 – 8	MVCR_[7:0]	Master volume control。Default value 00000000。
7 – 0	MVCL_[7:0]	Master volume control。Default value 00000000。

寄存器详细描述

MVCR_7 MVCL_7	MVCR_6 MVCL_6	MVCR_5 MVCL_5	MVCR_4 MVCL_4	MVCR_3 MVCL_3	MVCR_2 MVCL_2	MVCR_1 MVCL_1	MVCR_0 MVCL_0	Volume（dB）
0	0	0	0	0	0	0	0	0（default）
0	0	0	0	0	0	0	1	-0.25
0	0	0	0	0	0	1	0	-0.50
0	0	0	0	0	0	1	1	-0.75
0	0	0	0	0	1	0	0	-1
……	……	……	……	……	……	……	……	……
1	1	0	0	1	0	0	0	-50
1	1	0	0	1	1	0	0	-51
1	1	0	0	1	1	0	1	-51.25
1	1	0	0	1	1	1	0	-51.50
……	……	……	……	……	……	……	……	……
1	1	1	1	1	0	0	0	-78
1	1	1	1	1	1	0	0	-∞

Master volume control bits

(g) 录音音量控制：

BIT	15	14	13	12	11	10	9	8
Symbol	VC2_7	VC2_6	VC2_5	VC2_4	VC2_3	VC2_2	VC2_1	VC2_0
Default	1	1	1	1	1	1	1	1

BIT	7	6	5	4	3	2	1	0
Symbol	VC1_7	VC1_6	VC1_5	VC1_4	VC1_3	VC1_2	VC1_1	VC1_0
Deafult	0	0	0	0	0	0	0	0

Register address：11H

BIT	SYMBOL	描述
15 – 8	VC2_[7:0]	Digital mixer volume control。Default value for channel 2：00000000。
7 – 0	VC1_[7:0]	Digital mixer volume control。Default value for channel 1：11111111。

寄存器详细描述

VC2_7 VC1_7	VC2_6 VC1_6	VC2_5 VC1_5	VC2_4 VC1_4	VC2_3 VC1_3	VC2_2 VC1_2	VC2_1 VC1_1	VC2_0 VC1_0	Volume（dB）
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	-0.25
0	0	0	0	0	0	1	0	-0.50
0	0	0	0	0	0	1	1	-0.75
0	0	0	0	0	1	0	0	-1
……	……	……	……	……	……	……	……	……
1	0	1	1	0	1	0	0	-45
1	0	1	1	0	1	0	1	-45.25
1	0	1	1	0	1	1	0	-45.50
……	……	……	……	……	……	……	……	……
1	1	1	0	0	1	0	0	-∞
……	……	……	……	……	……	……	……	……
1	1	1	1	1	1	0	0	-∞

Digital mixer volume control

(h) 低音放大和高音模式：

BIT	15	14	13	12	11	10	9	8
Symbol	M1	M0	TRL1	TRL0	BBL3	BBL2	BBL1	BBL0
Default	0	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	—	—	TRR1	TRR0	BBR3	BBR2	BBR1	BBR0
Default	0	0	0	0	0	0	0	0

Register address：12H

BIT	SYMBOL	描述
15 – 14	M[1:0]	Flat/minimum/maximum settings。Default value 00。
13 – 12	TRL[1:0]	Treble setting left。Default value 00。
11 – 8	BBL[3:0]	Bass boost setting left。Default value 0000。
7 – 6	—	Default value 00。

5 – 4	TRR[1:0]	Treble setting right。Default value 00。
3 – 0	BBR[3:0]	Bass boost setting right。Default value 0000。

寄存器详细描述

M1	M0	模式
0	0	Flat（default）
0	1	Minimum
1	0	Minimum
1	1	Maximum

Flat/minimum/maximum setting bits

TRL1 TRR1	TRL0 TRR0	Flat set （dB）	Minimum set （dB）	Maximum set （dB）
0	0	0（default）	0（default）	0（default）
0	1	0	2	2
1	0	0	4	4
1	1	0	6	6

Treble setting bits

BBL3 BBR3	BBL2 BBR2	BBL1 BBR1	BBL0 BBR0	Flat set （dB）	Minimum set （dB）	Maximum set （dB）
0	0	0	0	0（default）	0（default）	0（default）
0	0	0	1	0	2	2
0	0	1	0	0	4	4
0	0	1	1	0	6	6
……	……	……	……	……	……	……
1	0	0	0	0	16	16
1	0	0	1	0	18	18
1	0	1	0	0	18	20
1	0	1	1	0	18	22
Others				0	18	24

Bass boost setting bits

(i) 放音静音设置：

BIT	15	14	13	12	11	10	9	8
Symbol	—	MTM	—	—	MT2	DE2_2	DE2_1	DE2_0
Default	0	1	0	0	1	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	—	—	—	—	MT1	DE1_2	DE1_1	DE1_0
Default	0	0	0	0	0	0	0	0

Register address：13H

BIT	SYMBOL	描述

15	—	Default value 0。
14	MTM	Master mute。The bit is logic 0：no soft mute of master、the bit is logic 1：soft mute of master。Default value 1。
13 – 12	—	Default value 00。
11	MT2	Channel 2 mute。The The bit is logic 0：no soft mute of channel 2、the bit is logic 1：soft mute of channel 2。Default value 1。
10 – 8	DE2_[2:0]	De-emphasis。Default value 000。
7 – 4	—	Default value 0000。
3	MT1	Channel 1 mute。The The bit is logic 0：no soft mute of channel 1、the bit is logic 1：soft mute of channel 1。Default value 0。
2 – 0	DE1_[2:0]	De-emphasis。Default value 000。

寄存器详细描述

DE2_2 DE1_2	DE2_1 DE1_1	DE2_0 DE2_1	功能
0	0	0	Off（default）
0	0	1	32kHz
0	1	0	44.1kHz
0	1	1	48kHz
1	0	0	96kHz

De-emphasis selection bits

(j) 录音静音设置：

BIT	15	14	13	12	11	10	9	8
Symbol	DA_POL_INV	SEL_NS	MIX_POS	MIX	—	—	—	—
Default	0	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	SILENCE	SDET_ON	SD_VALUE1	SD_VALUE0	—	—	OS1	OS0
Default	0	0	0	0	0	0	0	0

Register address：14H

BIT	SYMBOL	描述
15	DA_POL_INV	DAC polarity control。The bit is logic 0：DAC output not inverted、the bit is logic 1：DAC output inverted。Default value 0。
14	SEL_NS	Noise shaper order select。The bit is logic 0：select 3rd-order noise shaper、the bit is logic 1：select 5th-order noise shaper。Default value 0。
13	MIX_POS	Mixer signal control。Default value 00。
12	MIX	
11 – 8	—	Default value 0000。
7	SILENCE	Silence detector。The bit is logic 0：no overruling、the bit is logic 1：

		overruling。Default value 0。
6	SDET_ON	Silence detector enable。The bit is logic 0：silence detection circuit disable、the bit is logic 1：silence detection circuit enable。Default value 0。
5 – 4	SD_VALUE[1:0]	Silence detection settings。Default value 00。
3 – 2	—	Default value 00。
1 – 0	OS[1:0]	Oversampling input settings。Default value 00。

寄存器详细描述

MIX_POS	MIX	功能
0	0	No mixing（default）
1	0	Volume of channel 1 is forced to 0 dB and volume of channel 2 is forced to -∞ dB。
0	1	Mixing is done before the sound processing。
1	1	Mixing is done after the sound processing。

Mixer signal control setting bits

SD_VALUE1	SD_VALUE0	功能
0	0	3200 samples（default）
0	1	4800 samples
1	0	9600 samples
1	1	19200 samples

Silence detector setting bits

OS1	OS0	功能
0	0	Single-speed input is normal input；mixing possible；（default）
0	1	Duuble-speed input is after frist half-band；no mixing possible
1	0	Quad-speed input is in front of noise shaper；no mixing possible
1	1	Reserved

Oversampling input setting bits

(k) ADC 音量控制：

BIT	15	14	13	12	11	10	9	8
Symbol	ML_DEC7	ML_DEC6	ML_DEC5	ML_DEC4	ML_DEC3	ML_DEC2	ML_DEC1	ML_DEC0
Default	0	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	MR_DEC7	MR_DEC6	MR_DEC5	MR_DEC4	MR_DEC3	MR_DEC2	MR_DEC1	MR_DEC0
Default	0	0	0	0	0	0	0	0

Register address：20H

BIT	SYMBOL	描述
15 – 8	ML_DEC[7:0]	ADC volume control left。Default value 00000000。
7 – 0	MR_DEC[7:0]	ADC volume control right。Default value 00000000。

寄存器详细描述

ML_DEC7 MR_DEC7	ML_DEC6 MR_DEC6	ML_DEC5 MR_DEC5	ML_DEC4 MR_DEC4	ML_DEC3 MR_DEC3	ML_DEC2 MR_DEC2	ML_DEC1 MR_DEC1	ML_DEC0 MR_DEC0	GAIN（dB）
0	0	1	1	0	0	0	0	24
0	0	1	0	1	1	1	1	23.5
0	0	1	0	1	1	1	0	23
……	……	……	……	……	……	……	……	……
0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	0	1	0.5
0	0	0	0	0	0	0	0	0（default）
1	1	1	1	1	1	1	1	-0.5
……	……	……	……	……	……	……	……	……
1	0	0	0	0	0	1	0	-63
1	0	0	0	0	0	0	1	-63.5
1	0	0	0	0	0	0	0	-∞

ADC volume control setting bits

(l)　PGA 设置：

BIT	15	14	13	12	11	10	9	8
Symbol	MT_ADC	—	—	—	PGA_GAIN CTRLR3	PGA_GAIN CTRLR2	PGA_GAIN CTRLR1	PGA_GAIN CTRLR0
Default	1	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	—	—	—	—	PGA_GAIN CTRLL3	PGA_GAIN CTRLL2	PGA_GAIN CTRLL1	PGA_GAIN CTRLL0
Default	0	0	0	0	0	0	0	0

Register address：21H

BIT	SYMBOL	描述
15	MT_ADC	Decimator mute。The bit is logic 0：no muting、the bit is logic 1：muting。Default value 1。
14 – 12	—	Default value 000。
11 – 8	PGA_GAIN CTRLR[3:0]	ADC input amplifier right gain settings。Default value 0000。
7 – 4	—	Default value 0000。
3 – 0	PGA_GAIN CTRLL[3:0]	ADC input amplifier left gain settings。Default value 0000。

寄存器详细描述

PGA_GAIN CTRLR3 PGA_GAIN CTRLL3	PGA_GAIN CTRLR2 PGA_GAIN CTRLL2	PGA_GAIN CTRLR1 PGA_GAIN CTRLL1	PGA_GAIN CTRLR0 PGA_GAIN CTRLL0	PGA_GAIN（dB）
0	0	0	0	0（default）
0	0	0	1	3
0	0	1	0	6
……	……	……	……	……
0	1	1	1	21
1	X	X	X	24

ADC input amplifier PGA gain setting bits

(m) ADC 设置：

BIT	15	14	13	12	11	10	9	8
Symbol	–	–	–	ADCPOL_INV	VGA_CTRL3	VGA_CTRL2	VGA_CTRL1	VGA_CTRL0
Default	0	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	–	–	–	–	SEL_LNA	SEL_MIC	SKIP_DCFIL	EN_DCFIL
Default	0	0	0	0	0	0	1	0

Register address：22H

BIT	SYMBOL	描述
15 – 13	–	Default value 000。
12	ADCPOL_INV	ADC polarity control。The bit is logic 0：polarity of ADC not-inverting、the bit is logic 1：polarity of ADC inverting。Default value 0。
11 – 8	VGA_CTRL[3:0]	Microphone input VGA gain settings。Default value 0000。
7 – 4	–	Default value 0000。
3	SEL_LNA	Line input select。The bit is logic 0：select line input、the bit is logic 1：select LNA fot the left ADC。Default value 0。
2	SEL_MIC	Microphone input select。The bit is logic 0：select right channel ADC、the bit is logic 1：select left channel ADC。Default value 0。
1	SKIP_DCFIL	DC filter bypass。The bit is logic 0：DC filter enabled、the bit is logic 1：DC filter bypassed。Default value 1。
0	EN_DCFIL	DC filter enable。The bit is logic 0：DC filter disabled、the bit is logic 1：DC filter enabled。Default value 0。

寄存器详细描述

VGA_CTRL3	VGA_CTRL2	VGA_CTRL1	VGA_CTRL0	LNA GAIN（dB）
0	0	0	0	0（default）
0	0	0	1	2
0	0	1	0	4
……	……	……	……	……
1	1	1	1	30

Microphone input VGA gain setting bits

(n) AGC 设置：

BIT	15	14	13	12	11	10	9	8
Symbol	—	—	—	—	—	AGC_TIME2	AGC_TIME1	AGC_TIME0
Default	0	0	0	0	0	0	0	0
BIT	7	6	5	4	3	2	1	0
Symbol	—	—	—	—	AGC_LEVEL1	AGC_LEVEL0	—	AGC_EN
Default	0	0	0	0	0	0	0	0

Register address：23H

BIT	SYMBOL	描述
15 – 11	—	Default value 00000。
10 – 8	AGC_TIME[2:0]	AGC time constant settings。Default value 000。
7 – 4	—	Default value 0000。
3 – 2	AGC_LEVEL[1:0]	AGC target level settings。Default value 00。
1	—	Default value 0。
0	AGC_EN	AGC enable control。The bit is logic 0：AGC off、the bit is logic 1：AGC enabled。Default value 0。

寄存器详细描述

AGC_ TIME2	AGC_ TIME1	AGC_ TIME0	AGC setting			
			44.1kHz sampling		8kHz sampling	
			Attack time （ms）	Decay time （ms）	Attack time（ms）	Decay time （ms）
0	0	0	11	100	61	551（default）
0	0	1	16	100	88.2	551
0	1	0	11	200	61	1102
0	1	1	16	200	88.2	1102
1	0	1	21	200	116	1102
1	0	0	11	400	61	2205
1	1	0	16	400	88.2	2205
1	1	1	21	400	116	2205

AGC time constant setting bits

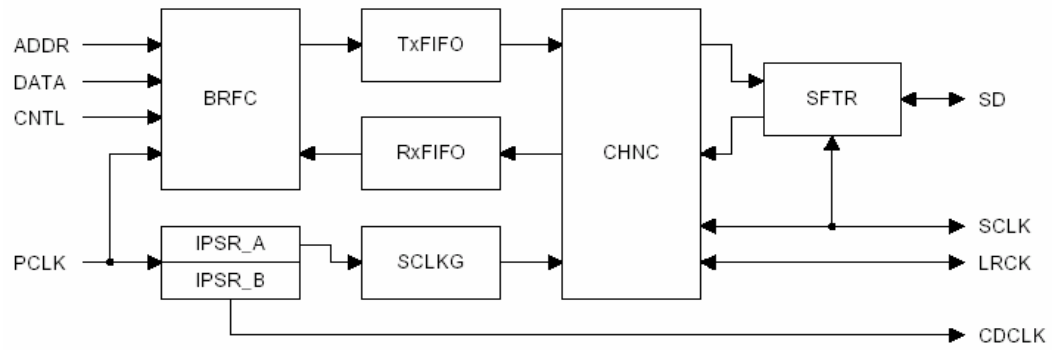
AGC_LEVEL1	AGC_LEVEL0	AGC target level value（dBFS）
0	0	-5.5（default）
0	1	-8
1	0	-11.5
1	1	-14

AGC target level setting bits

3．IIS：

S3C2410 的 IIS 总线用于实现外部 8/16 位立体声音频的多媒体数字信号编解码器，它支持 IIS 总线数据格式和 MSB-justified 数据格式。它可以同时发送和接收数据，也可以在发送

和接收数据中二者择一。



IIS-Bus Block Diagram

(1) IIS 的发送、接收模式：

(a) 普通传输方式：

IIS 控制寄存器中有发送和接收数据的起始标志位。当数据开始发送时，如果要发送的数据不为空，则 IIS 控制寄存器中发送数据的起始标志位被置 1。当数据发送完毕后，起始标志位被置 0。在接收数据时，当接收数据未满时，接收数据的起始标志位被置 1，它表示数据正准备被接收。当接收数据满时，接收数据的起始标志位被置 0。

(b) DMA 传输方式：

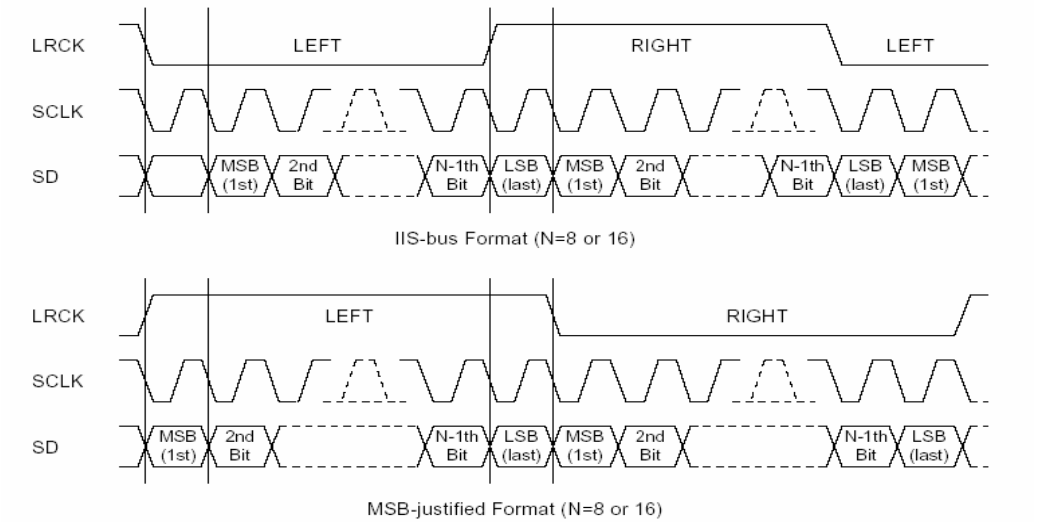
在这种方式下，发送和接收数据受到 DMA 控制器的影响。发送或接收数据的 DMA 服务请求被自动的执行。

(c) 发送、接收方式：

在这种方式下，IIS 总线可以同时地发送、接收数据。

(2) IIS 总线格式：

IIS 总线有四种序列：连续的数据输入（IISDI）、连续的数据输出（IISDO）、左/右声道选择（IISLRCK）、连续的时钟（IISCLK）。



IIS-Bus and MSB(left)-justified data interface formats

(3) 采样频率和主时钟：

主时钟频率（PCLK）可以从采样频率中选择。主时钟频率来自 IIS 的预定标器，其中预定标器的值以及主时钟频率的类型是确定的。连续数据传输的时钟频率类型可以通过主时钟

频率和连续数据每个通道来选择。

IISLRCK (fs)	8.000 kHz	11.025 kHz	16.000 kHz	22.050 kHz	32.000 kHz	44.100 kHz	48.000 kHz	64.000 kHz	88.200 kHz	96.000 kHz
CODECLK (MHz)	256fs									
	2.0480	2.8224	4.0960	5.6448	8.1920	11.2896	12.2880	16.3840	22.5792	24.5760
	384fs									
	3.0720	4.2336	6.1440	8.4672	12.2880	16.9344	18.4320	24.5760	33.8688	36.8640

CODEC clock（CODECLK ＝ 256 or 384fs）

(4) IIS 总线寄存器：

寄存器	地址	R/W	描述
IISCON	0x55000000（Li/HW，Li/W，Bi/W） 0x55000002（Bi/HW）	R/W	IIS control register
IISMOD	0x55000004（Li/HW，Li/W，Bi/W） 0x55000006（Bi/HW）	R/W	IIS mode register
IISPSR	0x55000008（Li/HW，Li/W，Bi/W） 0x5500000A（Bi/HW）	R/W	IIS prescaler register
IISFCON	0x5500000C（Li/HW，Li/W，Bi/W） 0x5500000E（Bi/HW）	R/W	IIS FIFO interface register
IISFIFO	0x55000010（Li/HW，Li/W，Bi/W） 0x55000012（Bi/HW）	R/W	IIS FIFO register

(5) IIS 总线寄存器的具体设置：

IISCON	Bit	描述	起始状态
Left/rigth channel index（read only）	8	0：left 1：right	1
Transmit FIFO ready flag（read only）	7	0：empty 1：not empty	0
Receive FIFO ready flag（read only）	6	0：full 1：not full	0
Transmit DMA service request	5	0：disable 1：enable	0
Receive DMA service request	4	0：disable 1：enable	0
Transmit channel idle command	3	0：not idle 1：idle	0
Receive channel idle command	2	0：not idle 1：idle	0
IIS prescaler	1	0：disable 1：enable	0
IIS interface	0	0：disable（stop） 1：enable（start）	0

IIS control register

IISMOD	Bit	描述	起始状态
Master/slave mode select	8	0：master mode 1：slave mode	0
Transmit/receive mode select	[7:6]	00：no transfer 01：receive mode 10：transmit mode 11：transmit and receive mode	00
Active level of left/right channel	5	0：low for left channel 1：high for left channel	0
Serial interface format	4	0：IIS compatible format 1：MSB（left）-justified format	0
Serial data bit per channel	3	0：8-bit 1：16-bit	0
Master clock frequency select	2	0：256fs 1：384fs	0
Serial bit clock frequency select	[1:0]	00：16fs 01：32fs 10：48fs 11：N/A	00

IIS mode register

IISPSR	Bit	描述	起始状态
Prescaler control A	[9:5]	Data value：0 － 31	00000
Prescaler control B	[4:0]	Data value：0 － 31	00000

IIS prescaler register

IISFCON	Bit	描述	起始状态
Transmit FIFO access mode select	15	0：normal 1：DMA	0
Receive FIFO access mode select	14	0：normal 1：DMA	0
Transmit FIFO	13	0：disable 1：enable	0
Receive FIFO	12	0：disable 1：enable	0
Transmit FIFO data count（read only）	[11:6]	Data count value ＝ 0-32	000000
Receive FIFO data count（read only）	[11:6]	Data count value ＝ 0-32	000000

IIS FIFO interface register

IISFIF	Bit	描述	起始状态
Fentry	[15:0]	Transmit/receive data for IIS	0x0

IIS FIFO register

4．DMA

(1) S3C2410 支持 4 路 DMA 通道，各通道 DMA 的申请源如下表所示 ：

	Source0	Source1	Sourec2	Source3	Source4
Ch-0	NXDREQ0	UART0	SDI	Timer	USB device EP1
Ch-1	NXDREQ1	UART1	I2SSDI	SPI0	USB device EP2
Ch-2	I2SSDO	I2SSDI	SDI	Timer	USB device EP3
Ch-3	UART2	SDI	SPI1	Timer	USB device EP4

Table ： DMA Request Sources for Each Channel

(2) DMA 的工作过程

DMA 使用 3 态 FSM（Finite State Machine）来控制它的运行，分为 3 个步骤 ：

(a) 等待一个 DMA 请求。在等待状态时，DMA ACK 和 INT REQ 两者都为 0；

(b) 有 DMA 请求时，DMA ACK 的值变为 1，并一直保持，直到它被清除。同时从 DCON 寄存器中取出计数器的值；

(c) 次 FSM 开始处理 DMA 的微操作：从源地址读取数据，然后将数据写到目的地址（数据的大小和传输的长度必须考虑）。这一过程被不断重复，直到计数器的值（由 CURR_TC 寄存器中获得）变为 0。当次 FSM 完成每次传输时，主 FSM 将倒计数 CURR_TC。当整个服务中的 CURR_TC 的值变为 0 时，主 FSM 发出中断申请，此时 DCON（DMA CONTROL REGISTER）中中断请求被设置为 1。DMA ACK 将在下列情况中被清除：

(i) 在整个服务中 CURR_TC 的值变为 0；

(ii) 在单一服务过程中微操作结束。

(3) DMA 寄存器：

(a) DMA 初始源寄存器（DISRC）：

寄存器	地址	R/W	描述	初始值
DISRC0	0x4B000000	R/W	DMA 通道 0 初始源寄存器	0x00000000
DISRC1	0x4B000040	R/W	DMA 通道 1 初始源寄存器	0x00000000
DISRC2	0x4B000080	R/W	DMA 通道 2 初始源寄存器	0x00000000
DISRC3	0x4B0000C0	R/W	DMA 通道 3 初始源寄存器	0x00000000

DISRCn	Bit	描述	起始状态
S_ADDR	[30:0]	发送数据源的起始地址	0x00000000

寄存器详细描述

(b) DMA 初始源控制寄存器（DISRCC）：

寄存器	地址	R/W	描述	初始值
DISRCC0	0x4B000004	R/W	DMA 通道 0 初始源控制寄存器	0x00000000
DISRCC1	0x4B000044	R/W	DMA 通道 1 初始源控制寄存器	0x00000000
DISRCC2	0x4B000084	R/W	DMA 通道 2 初始源控制寄存器	0x00000000

DISRCC3	0x4B0000C4	R/W	DMA 通道 3 初始源控制寄存器	0x00000000

DISRCCn	Bit	描述	起始状态
LOC	1	0：the source is in the system bus（AHB） 1：the source is in the peripheral bus（APB）	0
INC	0	0：increment 1：fixed	0

寄存器详细描述

(c) DMA 初始目的地址寄存器（DIDST）：

寄存器	地址	R/W	描述	初始值
DIDST0	0x4B000008	R/W	DMA 通道 0 初始目的地址寄存器	0x00000000
DIDST1	0x4B000048	R/W	DMA 通道 1 初始目的地址寄存器	0x00000000
DIDST2	0x4B000088	R/W	DMA 通道 2 初始目的地址寄存器	0x00000000
DIDST3	0x4B0000C8	R/W	DMA 通道 3 初始目的地址寄存器	0x00000000

DIDSTn	Bit	描述	起始状态
D_ADDR	[30:0]	数据传输的目的地的起始地址	0x00000000

寄存器详细描述

(d) DMA 初始目的地址控制寄存器（DIDSTC）：

寄存器	地址	R/W	描述	初始值
DIDSTC0	0x4B00000C	R/W	DMA 通道 0 初始目的地址控制寄存器	0x00000000
DIDSTC1	0x4B00004C	R/W	DMA 通道 1 初始目的地址控制寄存器	0x00000000
DIDSTC2	0x4B00008C	R/W	DMA 通道 2 初始目的地址控制寄存器	0x00000000
DIDSTC3	0x4B0000CC	R/W	DMA 通道 3 初始目的地址控制寄存器	0x00000000

DISRCCn	Bit	描述	起始状态
LOC	1	0：the source is in the system bus（AHB） 1：the source is in the peripheral bus（APB）	0
INC	0	0：increment 1：fixed	0

寄存器详细描述

(e) DMA 控制寄存器（DCON）：

寄存器	地址	R/W	描述	初始值

DCON0	0x4B000010	R/W	DMA 通道 0 控制寄存器	0x00000000
DCON1	0x4B000050	R/W	DMA 通道 1 控制寄存器	0x00000000
DCON2	0x4B000090	R/W	DMA 通道 2 控制寄存器	0x00000000
DCON3	0x4B0000D0	R/W	DMA 通道 3 控制寄存器	0x00000000

DISRCCn	Bit	描述	起始状态
DMD_HS	31	0：选择查询方式 1：选择握手方式	0
SYNC	30	0：DREQ 和 DACK 与 PCLK 同步（APB clock） 1：DREQ 和 DACK 与 HCLK 同步（AHB clock）	0
INT	29	0：CURR_TC 中断未打开 1：当所有传输完成时，发出中断请求	0
TSZ	28	0：一个数据的传输完成 1：一定长度的突发数据传输完成	0
SERVMODE	27	0：单一服务模式 1：完全服务模式	0
HWSRCSEL	[26:24]	选择各 DMA 通道的 DMA 申请源	00
SWHW_SEL	23	0：S/W request mode is selected 1：DMA source selected by bit[26:24] triggers the DMA operation	0
RELOAD	22	0：当传输完毕后自动再次执行 1：当传输完毕后 DMA 通道被关闭	0
DSZ	[21:20]	传输数据的长度 00：位 01：半个字节 10：一个字节 11：保留	00
TC	[19:0]	传输数据的总计数	0000

寄存器详细描述

(f) DMA 状态寄存器（DSTAT）：

寄存器	地址	R/W	描述	初始值
DSTAT0	0x4B000014	R	DMA 通道 0 状态寄存器	000000h
DSTAT1	0x4B000054	R	DMA 通道 1 状态寄存器	000000h
DSTAT2	0x4B000094	R	DMA 通道 2 状态寄存器	000000h
DSTAT3	0x4B0000D4	R	DMA 通道 3 状态寄存器	000000h

DSTATn	Bit	描述	起始状态
STAT	[21:20]	DMA 控制状态 00：表示已经为其他的 DMA 请求做好准备 01：表示 DMA 正在传输数据	00b
CURR_TC	[19:0]	传输数据的总计数	00000h

寄存器详细描述

(g) DMA 当前申请源地址寄存器（DCSRC）：

寄存器	地址	R	描述	初始值
DCSRC0	0x4B000018	R	DMA 通道 0 当前申请源地址寄存器	0x00000000
DCSRC1	0x4B000058	R	DMA 通道 1 当前申请源地址寄存器	0x00000000
DCSRC2	0x4B000098	R	DMA 通道 2 当前申请源地址寄存器	0x00000000
DCSRC3	0x4B0000D8	R	DMA 通道 3 当前申请源地址寄存器	0x00000000

DCSRCn	Bit	描述	起始状态
CURR_SRC	[30:0]	DMA 通道 n 当前使用的申请源地址	0x00000000

寄存器详细描述

(h) DMA 当前目的地地址寄存器（DCDST）：

寄存器	地址	R	描述	初始值
DCDST0	0x4B00001C	R	DMA 通道 0 当前目的地地址寄存器	0x00000000
DCDST1	0x4B00005C	R	DMA 通道 1 当前目的地地址寄存器	0x00000000
DCDST2	0x4B00009C	R	DMA 通道 2 当前目的地地址寄存器	0x00000000
DCDST3	0x4B0000DC	R	DMA 通道 3 当前目的地地址寄存器	0x00000000

DCDSTn	Bit	描述	起始状态
CURR_DST	[30:0]	DMA 通道 n 当前使用的目的地地址	0x00000000

寄存器详细描述

(i) DMA 屏蔽触发寄存器（DMASKTRIG）：

寄存器	地址	R/W	描述	初始值
DMASKTRIG0	0x4B000020	R/W	DMA 通道 0 屏蔽触发寄存器	000
DMASKTRIG1	0x4B000060	R/W	DMA 通道 1 屏蔽触发寄存器	000
DMASKTRIG2	0x4B0000A0	R/W	DMA 通道 2 屏蔽触发寄存器	000
DMASKTRIG3	0x4B0000E0	R/W	DMA 通道 3 屏蔽触发寄存器	000

DMASKTRIGn	Bit	描述	起始状态
STOP	2	停止 DMA 的操作 1：当传输的原子操作一结束就停止 DMA	0
ON_OFF	1	0：DMA 通道关闭 1：打开 DMA 通道，并处理 DMA 请求	0
SW_TRIG	0	在 S/W 请求模式下，触发 DMA 通道 1：请求一个 DMA 的服务	0

寄存器详细描述

(4) RUIJIARM9-EDU 中 DMA 的使用：

RUIJIARM9-EDU 的音频驱动使用的是 I2SSDO、I2SSDI 两路进行数据的输出和输入。因此只可以使用 DMA-Ch2，对数据进行输出，即实现放音的功能。而可以使用 DMA-Ch1 或 DMA-Ch2，对数据进行输入，即实现录音的功能。当录音使用 DMA-Ch1 时，录、放音使用了不同的 DMA 通道，这样就可以实现录音与放音的同步。当录音使用 DMA-Ch2 时，由于和

放音时申请的是同一个 DMA 通道，此时只能录音与放音分别进行，即必须加载相应的驱动模块。

三．实验内容和步骤

1．参考 UDA1380 各寄存器的设置，在 2410iic.c 中添加 command 数组，并测试寄存器是否设置正确：

(1) command 数组中每三个数一组，每组的第一个是 UDA1380 的寄存器的地址，后两个是往该寄存器里写入的值，注意后两个值的先后顺序；
(2) 根据前面章节介绍的驱动程序框架，读懂相应的驱动测试文件，test_uda1380.c；
(3) 读懂 Makefile_uda1380，进行编译，先 make clean，然后 make –f Makefile_uda1380；
(4) 把宿主机和开发板用串口线和对接线相连，在宿主机上输入命令 minicom；
(5) 进入命令行后，将宿主机 mount 到开发板的 mnt 目录，并用 lsmod 看已经加载的驱动；
(6) insmod test_uda1380.o，用音频对接线将宿主机的音乐输出口和开发板的 line in 口相连，并将耳机接到开发板的 phone 口。在宿主机播放音乐，如果 uda1380 初始化正确，就可以从耳机中听到音乐回放的声音；
(7) rmmod test_uda1380，卸载测试驱动。

2．参考 2410audio_play.c 中的 inti_module 和 clearup_module 函数，在 2410audio_rec.c 中添加相应的 inti_module 和 clearup_module 函数，并测试所写函数是否正确：

(1) 查看/RUIJIARM9-EUD/kernel/arch/arm/mach-s3c2410/dma.h，得到内核设定给录音所使用的 DMA 通道，代码为：

```
#define I2SSDI_CTL  (HS_MODE | SYNC_PCLK | INT_MODE | TSZ_UNIT |
                     SINGLE_SERVICE | HWSRC(CH2_I2SSDI) |
                     DMA_SRC_HWCLR_ATRELOAD |
                     DSZ(DSZ_HALFWORD) | TX_CNT(0))
```

初始状态下，录音使用的是：DMA_CH2；
(2) init_module 完成初始化总线、注册设备、申请 DMA 通道等功能，录音申请的 DMA 通道必须与内核设定的 DMA 通道相同，cleanup_module 完成注销设备、释放 DMA 通道的工作；
(3) 参考 Makefile_play，编写 Makefile_rec，然后 make –f Makefile_rec，生成录音驱动模块 2410audio_rec.o；
(4) lsmod 查看已加载的驱动，如果加载过 2410audio_play，则需要先将其卸载：rmmod 2410audio_play，然后 insmod 2410audio_rec.o，加载生成的录音驱动；
(5) 读懂 recorder.c，执行命令 make clean，然后 make，生成录音应用程序 recorder；
(6) 执行./recorder –n30，录音 30 秒，录音文件自动保存为/tmp/out.wav；
(7) 执行 make –f Makefile_play，生成放音驱动 2410audio_play.o，在开发板上，执行 rmmod 2410audio_rec 卸载录音驱动，insmod 2410audio_play.o 加载放音驱动，执行 cp /tmp/out.wav /dev/sound/dsp，播放刚才的录音文件；
(8) rmmod 2410audio_play，卸载放音驱动。

3．参考 2410audio_play.c、2410audio_rec.c，编写程序 2410audio_play_rec.c，使录、放音使用不同的 DMA 通道，做到录放音可以使用同一驱动：

(1) 修改 arch/arm/mach-s3c2410/dma.h，将其中设定的录音 DMA 通道由 DMA_CH2 改为

DMA_CH1， make zImage 生成新的内核，然后将新的内核烧写到 flash；

(2) 编写 2410audio_play_rec.c 时，申请的录音 DMA 通道为 DMA_CH1；

(3) 参考 Makefile_play，编写 Makefile_play_rec，然后 make –f Makefile_play_rec，生成录、放音驱动模块 2410audio_play_rec.o；

(4) insmod 2410audio_play_rec.o，加载新的驱动；

(5) 执行./recorder –n30，录音 30 秒，录音文件自动保存为/tmp/out.wav；

(6) 执行 cp /tmp/out.wav /dev/sound/dsp，播放刚才的录音文件；

(7) ./recorder –n30 &后台运行录音程序，cp song.wav /dev/sound/dsp 在前台播放音乐，测试同时录放音；

(8) rmmod 2410audio_play_rec，卸载驱动。

4．在录放音使用同一驱动的情况下，修改录音应用程序 recorder.c，做到录放音的同步，即在录音的同时，将所录的内容播放：

(1) insmod 2410audio_play_rec.o 加载音频驱动；

(2) recorder.c 在录音时，是将读到的数据直接写到录音文件/tmp/out.wav 中。如果将读到的数据直接播放，就可以实现录放音的同步；

(3) 执行命令 make clean，然后 make，生成新的录音应用程序；

(4) 执行./recorder –n30，录音 30 秒，此时可以直接从耳机中听到录音的内容；

(5) rmmod 2410audio_play_rec，卸载驱动。

四．思考题

1. 在做录放音同步实验时，录、放音时可以听到明显的声音滞后，为什么？如何修改才可以使声音的滞后减少？

实验二十六： MP3 解码实验

一．实验目的

通过本实验，使学生了解 MP3 解码播放的工作原理，了解 MP3 播放器的设计原理。

二．实验原理和说明：

1. 音频格式为什么要压缩

数字音频是对模拟声音信号每秒上千次的采样，然后把每个样值按一定的比特数量化，最后得到标准的数字音频的码流。对 CD 音质的信号来讲，每秒要 44100 次的采样，每个样值是 16 比特的量化，而立体声 CD 音质信号，它每秒的码流是 44.1K×16×2≈1.4Mbit/S。这样高的码流和容量，对于数字音频的存储、处理和传输提出了很高的要求。对音频的压缩理论，是从研究人耳的听感系统开始的，首先第一个特点是人耳对各频率的灵敏度是不同的，在 2K~4K 频段，很低的电平就能被人耳听到，其他频段时，相对要高一点的电平才能听到，这就是说在听觉阈值以下的电平可以去掉，相当于压缩了数据。第二个特点就是频率之间的掩蔽效应，其实就是指人耳接收信号时，不同频率之间的相互干扰。当电平高的频率点和电平相对来说较低的不同频率点同时出现时，电平低的频率点的声音将听不到。因为人耳的灵敏度不一样，所以不同频率点的掩蔽程度是不一样的。低于掩蔽阈值的信号将不编码，高于掩蔽阈值的信号将重新分配量化比特值，实施压缩，这是 MPEG 能得到较高的压缩比，又能保证音质的重要原因。第三个特点是指短暂掩蔽效应，指在一个强信号之前或之后的弱信号，也会被遮蔽掉。这样利用人耳的感觉特性，对数据流本身进行压缩，做到既能降低码流，又能通过科学的压缩方法提高码流的效率，而又不影响音质本身。完全了解了人耳的特性后，就会知道人耳实际上可看成一个多频段的听感分析器，在接收端的最后，它对瞬间的频谱功率进行了重新分配，这就为音频的数据压缩提供了依据。

2. MP3 历史

MP3 格式开始于 1980 年代中期(1987)，在德国 Erlangen 的 Fraunhofer 研究所开始的，研究致力于高质量、低数据率的声音编码。在 Dieter Seitzer-个德国大学教授的帮助下，1989 年，Fraunhofer 在德国被获准取得了 MP3 的专利权，几年后这项技术被提交到国际标准组织(ISO),整合进入了 MPEG-1 标准。

最早的播放器是 Frauenhofer 在 1990 年早期开发的,但他只是一个非常不知名的小程序，没有引起大家的重视。而被大家公认的第一个 Mp3 播放器是在 1997 年,由一个叫做 Tomislav Uzelac 的开发者开发的。他开发了 AMP MP3 播放引擎。当 AMP 引擎进入网络以后不久，几个大学生 Justin Frankel 和 Dmitry Boldyrev 拿到了 Amp 引擎，并且为他添加了一个 Windows 界面，最后他们把这个程序命名为"Winamp."

在 1998 年,当 Winamp 作为免费的音乐播放器在网络上传播的时候，Mp3 的狂潮开始了。许许多多的爱好者在网络上交换有版权的音乐 mp3。

MP3 格式是一个让音乐界产生巨大震动的一个声音格式。MP3 的全称是 Moving Picture Experts Group, Audio Layer III,它所使用的技术是在 VCD（MPEG-1）的音频压缩技术上发展出的第三代，而不是 MPEG-3。MP3 是一种音频压缩的国际技术标准。

MP3 格式可以使音乐文件在音乐质量做很小牺牲的情况下将文件大小缩小很多。MP3 文件能以不同的比率压缩，但是压缩的越多，声音质量下降的也越多。标准的 MP3 压缩比是

10：1，一个三分钟长的音乐文件压缩后大约是 4 MB。

MPEG 代表的是 MPEG 活动影音压缩标准，MPEG 音频文件指的是 MPEG 标准中的声音部分即 MPEG 音频层。MPEG 音频文件根据压缩质量和编码复杂程度的不同可分为三层（MPEG AUDIO LAYER 1/2/3 分别与 MP1，MP2 和 MP3 这三种声音文件相对应 MPEG 音频编码具有很高的压缩率，MP1 和 MP2 的压缩率分别为 4：1 和 6：1-8：1，而 MP3 的压缩率则高达 10：1-12：1，也就是说一分钟 CD 音质的音乐未经压缩需要 10MB 存储空间，而经过 MP3 压缩编码后只有 1MB 左右，同时其音质基本保持不失真。因此，目前 INTERNET 上的音乐格式以 MP3 最为常见。

3. MPEG-1 数字音频压缩原理

(1) MP3 及其基本原理

MP3 就是采用国际标准 MPEG 中的第三层音频压缩模式，对声音信号进行压缩的一种格式，中文也称“电脑网络音乐”。MPEG 中的第三层音频压缩模式比第一层和第二层编码要复杂得多，但音质最高，可与 CD 音质相比。为了更深入地理解，先把 MPEG－1 音频标准的特点作一些介绍。

(2) MPEG－1 音频标准的特点

MPEG－1 音频压缩标准是第一个高保真音频数据压缩标准。除了 AC－3 之外，其他的音频压缩算法只适用于语言(如码激励线性预测 CELP)或只有中等的压缩质量(如自适应差分脉冲编码调制 ADPCM)。MPEG－1 音频压缩标准虽然是 MPEG－1 标准的一部分，但它完全可独立应用。

为保证其普遍适用性，MPEG－1 音频压缩标准提供了以下压缩模式：

(a) 音频信号采样频率可以是 32kHz，44.1kHz 或 48kHz；

(b) 压缩后的比特流可按下列 4 种模式之一支持单声道或双声道：
提供给音频通道的单声道模式(monophonic mode)；
提供给两个独立的单音频通道的双－单声道模式(dual－monophonic mode)；
提供给立体声通道的立体声模式(stereomode),通道之间有比特共享；
联合立体声模式(joint－stereomode)，利用立体声之间的关联或通道之间的相位差的无关性，或者对两者同时利用；

(c) 压缩后的比特流具有预定的几种比特率之一。此外，MPEG－1　音频标准也支持用户使用预定的比特率之外的比特率；

(d) MPEG－1 音频标准提供三个独立的压缩层次，使用户可在复杂性和压缩质量之间权衡选择：
层 1 (Layer 1)最为简单，使用的比特率 384kb / s，主要用于数字小卡座 DCC(Digital Compact Cassette)。
层 2 (Layer 2)的复杂度属于中等，使用的比特率 192kb / s 左右。其应用包括：数字广播(Digital Audio Broadcasting)的音频编码，CD－ROM 上的音频信号以及交互式 CD－I(CD－zinteractive)。
层 3 (Layer 3)最为复杂，但音质最佳，使用的比特率为 64kb / s，尤其适用于综合业务数据网(ISDN)上的音频传输。

(e) 编码后的比特流支持循环冗余校验 CRC(Cyclic Redundancy Check)。

(f) MPEG－1 音频标准还支持在比特流中载带附加信息。

(3) MPEG－1 音频压缩标准基本原理描述

MPEG－1 音频标准是一个普遍适用的音频压缩标准，它对音频源没有任何要求。它利用人耳听觉系统的感知特性，压缩率的取得来自去掉人耳听不到的信息细节，即图 1 和图 2 曲线以下的部分。虽然压缩是有失真的，但对人耳来说这些失真是听不到的。也即对人耳而言，MPEG－1 音频压缩是不失真的。因此，MPEG－1 音频标准的应用非常广泛。

(4) MPEG－1 声音的一些性能指标

利用 MPEG－1 音频标准一般能达到的压缩率(以仍能达到 CD 音质为准)如表 1 所示。

1:4	Layer 1 (corresponds to 384 kbps for a stereo signal),
1:6...1:8	Layer 2 (corresponds to 256..192 kbps for a stereo signal),
1:10...1:12	Layer 3 (corresponds to 128..112 kbps for a stereo signal),

表 1

MPEG 层 3 在各种音质下的性能如表 2 所示。

音质	带宽	模式	压缩比率*（bitrate）	压缩率
普通电话音质	2.5 kHz	单声道	8 kbps	96:1
短波收音机音质	4.5 kHz	单声道	16 kbps	48:1
AM 调幅收音机音质	7.5 kHz	单声道	32 kbps	24:1
FM 调频收音机音质	11 kHz	立体声	56－64 kbps	26－24:1
接近 CD 唱盘音质	15 kHz	立体声	96 kbps	16:1
CD 唱盘音质	>15 kHz	立体声	112－128kbps	14－12:1

表 2

（*压缩比率（bitrate）：在音频压缩编/码过程中表示声音数据压缩大小的度量单位，指每秒声音数据在音频压缩文件中所占平均位数，单位是 kb/s 或者 kbps。）

从上述 MPEG－1 音频标准可看出：高压缩比是 MP3 的一个主要特性，其基本理论就是去除节目源中人耳听觉阈以外的所有信号，并将大信号掩盖下的小信号也加以除去，因为人耳具有掩盖效应，这种变化基本上觉察不出来，这样实际记录的信息量就比压缩前小得多，其压缩比可在 10：1～96：1(这次新科采用的是 CD 音质的 12：1)。这样，一张只能容纳十几首歌曲的光盘，就可记录 150 首以上的 MP3 格式歌曲。

(5) MP3 的优点

MP3 的突出优点是：压缩比高，音质较好，制作简单，交流方便。

压缩比高上面已述。音质是人们关心的一个焦点。虽然 MP3 对原始信号进行了高压缩比处理，但因为去除的大多是一些无关紧要的信号，因此单纯从听感上说，MP3 压缩几乎对音质没有影响。事实上，制作精良的 MP3 音乐碟，在专门的数字随身听(比如 MPMan)中播放，完全可以达到普通 CD 唱机播放 CD 唱片的音质水平。但最吸引人的还是 MP3 制作和交流上的方便。只要有一台电脑，就可将 CD 节目录入电脑硬盘，然后压制成 MP3 格式。也可直接从 Internet 网上下载 MP3 音乐，网上有取之不尽的 MP3 音乐。还可把你自己制作的 MP3 音乐上网交流。良好的音质和丰富的节目源将使 MP3 成为最佳的大众音乐媒体。

当然，MP3 的高压缩比是以牺牲细微的音质换来的，无疑会对音质产生一定的影响，对一般人和爱好者不会在意，但 MP3 是否能为 Hi－Fi 发烧友喜爱?!只有他们自己来回答。

(6) MP3 压缩原理

数字音频是对模拟声音信号每秒上千次的采样，然后把每个样值按一定的比特数量化，最后得到标准的数字音频的码流。对 CD 音质的信号来讲，每秒要 44100 次的采样，每个样值是 16 比特的量化，而立体声 CD 音质信号，它每秒的码流是 44.1K×16×2≈1.4Mbit/S。这样高的码流和容量，对于数字音频的存储、处理和传输提出了很高的要求。对音频的压缩理论，是从研究人耳的听感系统开始的，首先第一个特点是人耳对各频率的灵敏度是不同的，在 2K~4K 频段，很低的电平就能被人耳听到，其他频段时，相对要高一点的电平才能听到，这就是说在听觉阈值以下的电平可以去掉，相当于压缩了数据。第二个特点就是频率之间的掩蔽效应，其实就是指人耳接收信号时，不同频率之间的相互干扰。当电平高的频率点和电平相对来说较低的不同频率点同时出现时，电平低的频率点的声音将听不到。因为人耳的灵敏度不一样，所以不同频率点的掩蔽程度是不一样的。低于掩蔽阈值的信号将不编码，高于掩蔽阈值的信号将重新分配量化比特值，实施压缩，这是 MPEG 能得到较高的压缩比，又能保证音质的重要原因。第三个特点是指短暂掩蔽效应，指在一个强信号之前或之后的弱信号，也会被遮蔽掉。这样利用人耳的感觉特性，对数据流本身进行压缩，做到既能降低码流，又能通过科学的压缩方法提高码流的效率，而又不影响音质本身。完全了解了人耳的特性后，就会知道人耳实际上可看成一个多频段的听感分析器，在接收端的最后，它对瞬间的频谱功率进行了重新分配，这就为音频的数据压缩提供了依据。

在音频压缩标准化方面取得巨大成功的是 MPEG-1 数字音频压缩方案。我们介绍一下 MPEG-1 音频压缩的内容。在 MPEG-1 压缩中，按复杂程度规定了三种模式即层Ⅰ，层Ⅱ，层Ⅲ。目前广泛使用的 VCD 的音频压缩方案为层Ⅰ，它的典型的码流为每通道 192Kbit/S。层Ⅱ即称掩蔽模式通用子带集成编码与多路复用，典型的码流为每通道 128 Kbit/S，广泛应用于数字音频广播、数字演播室等数字音频专业的制作、交流、存储和传送。层Ⅲ是综合于层Ⅱ和 ASPEC 的优点提出的混合压缩技术，MP3 的复杂度相对较高，编码不利于实时，典型码流为 64 Kbit/S，在低码率下有高品质的音质，所以成为网上音源的宠儿。

MPEG-1 的压缩技术方案是子带压缩，子带分割的实现是通过时频映射，采用多相正交分解滤波器组将数字化的宽带音频信号分成 32 个子带；同时，信号通过 FFT 运算，对信号进行频谱分析；子带信号与频谱同步计算，得出对各子带的掩蔽特性，由于掩蔽特性的存在，减少了对量化比特率的要求，不同子带分配不同的量化比特数，但对于各子带而言，是线性量化。另上 CRC 校验码，得到标准的 MPEG 码流。在解码端，只要解帧，子带样值解码，最后进行频——时映射还原，最后输出标准 PCM 码流。其原理方框图如图 1 所示：

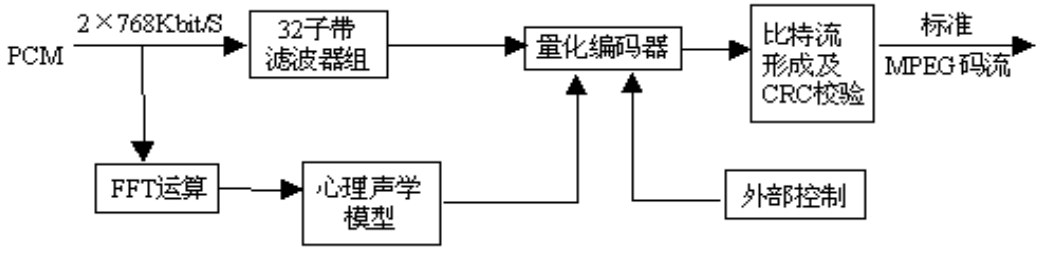


图 1

(a) 层Ⅰ：

时频映射：数字的多相正交滤波器组把信号分成 32 个子带信号，因为层Ⅰ是均匀的划分，所以每个子带频宽为 24K/32=625HZ。这种划分与关键频宽段的概念不一样，在低端只有一个子带 625HZ，这样对低频的量化比较简单，容易引起低频端的量化误差。

心理声学模型：使用 512 个点的 FFT 变换得到信号的短时频谱功率信息，输出的电平和时频映射的子带样值同步计算，得到每个子带的掩蔽阈值。最后将该子带的最大信号/掩蔽阈值率输入给量化器。

量化/编码：首先检测每个子带的样值，找到最大相对值，并且将它 6 比特量化，对该子带来讲叫比例因子。通过最小化噪音/掩蔽值，由比例因子决定动态量化比特数，将该子带值样值线性量化。量化比特数用一个 4 位码来描述，4 位码可以用来描述 24=16bit，最大 16bit 的量化。比例因子用 6 位码来描述，最大 26=64 的子带样值的系数。这样每个子带用的量化比特数和每个子带的最大样值都在 MPEG 的码流里，在接收端再按照这些信息还原原信号的幅值。

帧形成：每一帧的开始都有一个同步的信息，还有 CRC 的循环冗余纠错码。帧是 MPEG-1 处理的最小单元，一帧信号处理 384 个 PCM 的样值，因为要检测每个样值的大小后，才能开始处理，所以延时时间 384/48K=8ms。一帧相当于 8ms 的声音样本。一个子带所得码流的结构如表 3 所示：

同步头	CRC	比特分配信息	比例因子	样值数据
12 bit 同步 20bit 系统信息	16 bit	4 bit 线性描述	6bit 线性描述	一个子带样值对应 32 个 PCM 输入样值

表 3

(b) 层 II：

时频映射：和层Ⅰ类似，不同之处在于每个子带不是均匀频带宽，因为人耳低频时的灵敏度在 700HZ 以后急剧降低。与之相关的一个概念叫关键带宽，因为在同样的掩蔽值时，低频有窄的带宽，而高频端则有较宽的带宽。这样，在按关键带宽分割时，低频取的带宽窄，即意味着对低频有较高频率分辨率，在高频端时则相对有较低一点的分辨率。这样的分配，更符合人耳的灵敏度特性，可以改善对低频端压缩编码的失真。但这样做，需要较复杂一些的滤波器组。

心理声学模型：和层Ⅰ类似，但是使用的 FFT 精度高一些，是 1024 点的 FFT 运算方式，提高了频率的分辨率，得到原信号的更准确瞬间频谱特性。

量化/编码：和层Ⅰ类似。但是层Ⅱ的帧长度码流是层Ⅰ的三倍，所以层Ⅱ允许每个子带有三个连续的比例因子，但编码时用一、二个或者三个，由它们之间的差别来定。子带内有三个比例因子，这就意味着带内再进行动态比特分配，更增加了 MPEG1 的压缩率。

帧形成：和层Ⅰ不一样的是，描述比特分配的比特位数是不一样的。在低端子带用 4 位码来描述，相对低端子带量化比特数最大为 24=16bit，在中间子带用 3 位码描述，相对中间子带比特最大为 23=8bit，高端子带用 2 位码来描述，相对最大比特为 22=4bit，这种分频率不同而比特率不一样的做法，也是关键带宽的应用。层Ⅱ的帧码流是 1152 个 PCM 的样值，这样层Ⅱ处理的延时时间=1152/48K=24ms。这样对层Ⅱ的精确度为 24ms，而对于层Ⅰ来言，精确度为 8ms，如果用于编辑的话，层Ⅰ更精确。层Ⅱ标准码流信号如表 4 所示：

同步头	CRC	比特分配	比例因子	子带样值码
12bit 同步	16bit	低频端 4 位码描述	6 比特线性量化	三个子带样值
		中间频端 3 位码描述		

码 20bit 系统信息码		高频端 2 位码描述		对应 96 个 PCM 输入样值

表 4

(c) 层Ⅲ：

层Ⅲ比层Ⅱ更为复杂，它使用了多相正交滤波器组之外，还使用了 DCT 变换滤波器组，提高频率的分辨率，还应用了预测心理声学模式，使用更为复杂的量化和编码，允许不同的帧码流。这样对Ⅲ层而言，需要更快的数字信号处理器才能达到实时编码的要求。另外还使用了一些新的方案：

(i) 用冗余字节以作为缓冲用，因为有些音乐小节是无法将其按限定的码率来编码的，否则会损失其音质。为了保证其质量就需要用较高的码率。在 MP-3 的码流中就增加了一些冗余的字节，以便在需要的时候可以在短时间内提到较高的码率。

(ii) Ⅲ中，还采用了 Huffman 编码进行无损压缩，这就更进一步降低了数码率，提高了压缩比，据估计，采用 Huffman 编码以后，可以节省 20%的码率。

例如，经过分析，32 个子带的 16 个子带的最高样值电平如表 5 所示：

子带	1	2	3	4	5	6	7	8
电平（db）	0	8	12	10	6	2	10	60
子带	9	10	11	12	13	14	15	16
电平（db）	35	20	15	2	3	5	3	1

表 5

如表中第 8 个子带有一个强度为 60db 的 1KHZ 信号，通过心理声学模式计算出它的掩蔽阈值，在整个第八子带都有掩蔽效应，其门限为 35db，这样本子带的信噪比为 S/N=60-35=25db，因而 5bit 分辨率就足够了。同样，这种掩蔽也存在于邻近的子带中，在第 8~13，和第 5~7 子带都有掩蔽效应，只是信号频率离得越近，掩蔽效应就越严重，离得越远，效应越弱。由它决定的掩蔽值第七段是 12db，在第 9 段是 15db，这样，对第七段的电平 10db<12db，所以第 7 子带全部不编码，而第 9 段的信噪比 S/N 是 35-15（db）=20db，因而用 4bit 分辨率就足够了。这样第七段忽略不编码，第八段子带 5bit 线性量化，第九段子带 4bit 线性量化，这样就完成了码流的压缩。

三．实验内容和步骤

1． 移植 MADPLAY：在 S3C2410 平台上移植 MADPLAY。

(1) 在宿主机/RUIJIARM9-EUD/application/madplay/目录下，修改 Makefile 文件，将编译器路径替换为：/opt/host/armv4l/bin/armv4l-unknown-linux-，运行 make，重新编译，得到新的 madplay 程序，并将新的 madplay 拷贝到根目录下；

(2) 用对接线将开发板和宿主机相连，在开发板上运行命令：mount -o nolock [宿主机 IP]: / /tmp，将宿主机 mount 到开发板的 tmp 目录。在 tmp 目录下，运行./madplay [mp3 文件名]，播放 mp3 文件。

2. 通过计算解码时间，分析在不同解码频率下的解码效率：
计算纯解码时间的方法：

(1) 在 player.c 文件中，用函数 gettimeofday（），在播放前后各自取时间，两个时间相减，得到的就是播放时间。相关代码为：

```
static int play_all(struct player *player)
{
        ……
         struct timeval timerval1 , timeval2;
         struct timezone timezone1 , timezone2;
         long playtime;
        ……
         while(count&&(player->repeat==-1||player->repeat--)) {
              while(playlist->current<playlist->length) {
                   ……
                   gettimeofday(&timeval1 , &timezone1);   //播放前取时间
                   if(play_one(player)==-1) {
                        ……
                   }                                                    //播放 mp3
                   gettimeofday(&timeval2 , &timezone2);   //播放后再次取时间
                   ……
         playtime = (timeval2.tv_sec – timeval1.tv_sec)*1000000 + (timeval2.tv_usec –
    timeval1.tv_usec);
         printf(“The music play time is %ld us\n”,playtime);
        //计算 mp3 播放时间并打印出来，时间单位为：us
}
```

(2) 使用命令：madplay [mp3 文件名]，这样得到的时间就是 mp3 的播放时间；

(3) 使用命令：madplay -o /dev/null –R [解码频率] [mp3 文件名]，这样得到的时间就是纯解码时间；

3. 计算不同解码频率下的解码 CPU 占用率：
CPU 的占用率可以用如下公式计算：
CPU 占用率 ＝ 纯解码时间 / mp3 的播放时间；

四．思考题

1．根据实验结果，做不同解码频率下的解码 CPU 占用率表。

解码频率（Hz）	8K	16K	32K	64K	44.1K
解码时间（us）					
CPU 占用率表（％）					

不同解码频率下的解码 CPU 占用率表

2．还有什么方法可以测 CPU 的占用率？

实验二十七： 文件系统实验

一．实验目的

了解文件系统的机制以及同一操作系统中多种文件系统共存的实现。

二．实验原理和说明

1. 文件系统简介

象 Unix 一样，在 Linux 里，系统对独立的文件系统不是用设备标示符来存取（比如驱动器编号或驱动器名称），而是连接成为一个树型结构。Linux 在安装新的文件系统时，把它安装到指定的安装目录，比如/mnt/cdrom，从而合并到这个单一的文件系统树上。Linux 的一个重要特征是它支持多种不同的文件系统。这使它非常灵活而且可以和其他操作系统良好共存。Linux 最常用的文件系统是 EXT2，大多数 Linux 发布版都支持。

文件系统将存放在系统硬盘上的文件和目录用可以理解的统一的形式提供给用户，让用户不必考虑文件系统的类型或底层物理设备的特性。Linux 透明的支持多种文件系统（如 MS-DOS 和 EXT2），将所有安装的文件和文件系统集合成为一个虚拟的文件系统。所以，用户和进程通常不需要确切知道所使用的文件所在的文件系统的类型，用就是了。

块设备驱动程序掩盖了物理块设备类型的区别（如 IDE 和 SCSI）。对于文件系统来讲，物理设备就是线性的数据块的集合。不同设备的块大小可能不同，如软驱一般是 512 字节，而 IDE 设备通常是 1024 字节，同样，对于系统的用户，这些区别又被掩盖。EXT2 文件系统不管它用什么设备，看起来都是一样的。

文件系统指文件存在的物理空间，Linux 系统中每个分区都是一个文件系统，都有自己的目录层次结构。Linux 会将这些分属不同分区的、单独的文件系统按一定的方式形成一个系统的，总的目录层次结构。一个操作系统的运行离不开对文件的操作，因此必然要拥有并维护自己的文件系统。

Linux 文件系统使用索引节点来记录文件信息，作用像 Windows 的文件分配表。

索引节点是一个结构，它包含了一个文件的长度、创建及修改时间、许可权、所属关系、磁盘中的位置等信息。一个文件系统维护了一个索引节点的阵列，每个文件或目录都与索引节点阵列中的惟一一个元素对应。系统给每个索引节点分配了一个号码，也就是该节点在阵列中的索引号，称为索引节点号。

Linux 文件系统将文件索引节点号和文件名同时保存在目录中。所以，目录只是将文件的名称和它的索引节点号结合在一起的一张表，目录中每一对文件名称和索引节点号称为一个连接。

对于一个文件来说，有惟一的索引节点号与之对应，而对于一个索引节点号，却可以有多个文件名与之对应。因此，在磁盘上的同一个文件可以通过不同的路径去访问它。

可以用 ln 命令对一个已经存在的文件再创建一个新的连接，而不复制文件的内容。

连接有软连接和硬连接之分，软连接又叫符号连接。它们各自的特点如下。

(1) 硬连接：

(a) 原文件名和链接文件名都指向相同的物理位址。

(b) 目录不能有硬连接；硬连接不能跨越文件系统（不能跨越不同的分区）。

(c) 文件在磁盘中只有一个复制，以节省硬盘空间。

(d) 由于删除文件要在同一个索引节点属于惟一的连接时才能成功，因此可以防止不必要的误删除。

(2) 符号连接：

(a) 用 ln -s 命令创建文件的符号连接；

(b) 符号连接是 Linux 特殊文件的一种，作为一个文件，它的资料是它所连接的文件的路径名。类似于 Windows 下的快捷方式。

(c) 可以删除原有的文件而保存链接文件，没有防止误删除功能。

装载文件系统

由上一节知道，Linux 系统中每个分区都是一个文件系统，都有自己的目录层次结构。Linux 会将这些分属不同分区的、单独的文件系统按一定的方式形成一个系统的总的目录层次结构。这里所说的“按一定方式”就是指的装载。

将一个文件系统的顶层目录挂到另一个文件系统的子目录上，使它们成为一个整体，称为装载。把该子目录称为挂接点。

注意：挂接点必须是一个目录。

一个分区装载在一个已存在的目录上，这个目录可以不为空，但装载后这个目录下以前的内容将不可用。

对于其他操作系统创建的文件系统的装载也是这样。但是需要理解的是，光盘、软盘、其他操作系统使用的文件系统的格式与 Linux 使用的文件系统格式是不一样的。光盘是 ISO9660；软盘是 FAT16 或 ext2；Windows NT 是 FAT16, NTFS, Windows 98 是 FAT16, FAT32；Windows 2000 和 Windows XP 是 FAT16, FAT32, NTFS。装载前要了解 Linux 是否支持所要装载的文件系统格式。

装载时使用 mount 命令。格式：mount [-参数] [设备名称] [挂接点]

其中常用的参数有：

-t<文件系统类型> 指定设备的文件系统类型，常见的有：

minix Linux 最早使用的文件系统；

ext2 Linux 目前常用的文件系统；

msdos MS-DOS 的 FAT，就是 FAT16；

vfat Windows 98 常用的 FAT32；

nfs 网络文件系统；

iso9660 CD-ROM 光盘标准文件系统；

ntfs Windows NT/2000 的文件系统；

hpfs OS/2 文件系统；

auto 自动检测文件系统。

-o<选项> 指定装载文件系统时的选项，有些也可用在/etc/fstab 中。常用的有：

codepage=XXX 内码表；

iocharset=XXX 字符集；

ro 以只读方式装载；

rw 以读写方式装载；

nouser 使一般用户无法装载；

user 可以让一般用户装载设备。

提醒一下，mount 命令没有创建挂接点的功能，因此您应该确保执行 mount 命令时，挂接点已经存在。（即你要把文件系统装载到哪，首先要先建立这个目录。）

例：Windows 98 装在 hda1 分区，同时电脑上还有软盘和光盘需要装载。

```
# mk /mnt/winc
# mk /mnt/floppy
# mk /mnt/cdrom
# mount -t vfat /dev/hda1 /mnt/winc
# mount -t msdos /dev/fd0 /mnt/floppy
# mount -t iso9660 /dev/cdrom /mnt/cdrom
```

现在就可以进入/mnt/winc 等目录读写这些文件系统了。

要保证最后两行的命令不出错，就要确保软驱和光驱里有盘。

如果你的 Windows 98 目录里有中文文件名，使用上面的命令装载后，显示的是一堆乱码。这就要用到 -o 参数里的 codepage iocharset 选项。codepage 指定文件系统的内码表，简体中文代码是 936；iocharset 指定字符集，简体中文一般用 cp936 或 gb2312。

当装载的文件系统 Linux 不支持时，mount 一定会报错，如 Windows 2000 的 NTFS 文件系统。可以重新编译 Linux 内核以获得对该文件系统的支持。

2. ext2 和 ext3

Linux 中最基本的文件系统是 ext 文件系统，目前普遍使用的是 ext2 和 ext3 文件系统。

(1) Ext2: 是 GNU/Linux 系统中标准的文件系统，其特点为存取文件的性能极好，对于中小型的文件更显示出优势，这主要得利于其簇快取层的优良设计。其单一文件大小与文件系统本身的容量上限与文件系统本身的簇大小有关，在一般常见的 x86 电脑系统中，簇最大为 4KB, 则单一文件大小上限为 2048GB, 而文件系统的容量上限为 16384GB。但由于目前核心 2.4 所能使用的单一分割区最大只有 2048GB，因此实际上能使用的文件系统容量最多也只有 2048GB。

(2) Ext3: 顾名思义，它就是 ext2 的下一代，也就是在保有目前 ext2 的格式之下再加上日志功能。目前它离实用阶段还有一段距离，也许在下一版的核心就可以上路了。 ext3 是一种日志式文件系统。日志式文件系统的优越性在于：由于文件系统都有快取层参与运作，如不使用时必须将文件系统卸下，以便将快取层的资料写回磁盘中。因此每当系统要关机时，必须将其所有的文件系统全部卸下后才能进行关机。

如果在文件系统尚未卸下前就关机 (如停电) 时，下次重开机后会造成文件系统的资料不一致，故这时必须做文件系统的重整工作，将不一致与错误的地方修复。然而，此一重整的工作是相当耗时的，特别是容量大的文件系统，而且也不能百分之百保证所有的资料都不会流失。故这在大型的伺服器上可能会造成问题。

为了克服此问题，业界经长久的开发，而完成了所谓‘日志式文件系统 (Journal File System) ’。此类文件系统最大的特色是，它会将整个磁盘的写入动作完整记录在磁盘的某个区域上，以便有需要时可以回朔追踪。由于资料的写入动作包含许多的细节，像是改变文件标头资料、搜寻磁盘可写入空间、一个个写入资料区段等等，每一个细节进行到一半若被中断，就会造成文件系统的不一致，因而需要重整。然而，在日志式文件系统中，由于详细纪录了每个细节，故当在某个过程中被中断时，系统可以根据这些记录直接回朔并重整被中断的部分，而不必花时间去检查其他的部分，故重整的工作速度相当快，几乎不需要花时间。

另外 Linux 中还有一种专门用于交换分区的 swap 文件系统，Linux 使用整个分区来作为交换空间，而不象 Windows 使用交换文件。一般这个 SWAP 格式的交换分区是主内存的 2 倍。

3. 日志文件系统与非日志文件系统

众所周知，文件系统是操作系统最为重要的一部分。每种操作系统都有自己的文件系统。文件系统直接影响着操作系统的稳定性和可靠性。Linux 下的文件系统通常有两种，即日志文件系统和非日志文件系统，以下简单介绍两类文件系统。

3.1 非日志文件系统

非日志文件系统在工作时，不对文件系统的更改进行日志记录。

文件系统通过为文件分配文件块的方式把数据存储在磁盘上。每个文件在磁盘上都会占用一个以上的磁盘扇区，文件系统的工作就是维护文件在磁盘上的存放，记录文件占用了哪几个扇区。另外扇区的使用情况也要记录在磁盘上。文件系统在读写文件时，首先找到文件使用的扇区号，然后从中读出文件内容。如果要写文件，文件系统首先找到可用扇区，进行数据追加。同时更新文件扇区使用信息。不同的文件系统用不同的方法分配和读取文件块。例如，dos/windows 就使用 fat 文件系统，而 windows NT 则采用 NTFS 文件系统。

非日志文件系统能够工作得很稳定，但是，它存在不少问题。各位请看，对于一个普通的日志文件系统，如 Ext2 文件系统，如果系统刚将文件的磁盘分区占用信息(meta-data)写入到磁盘分区中，还没有来得及将文件内容写入磁盘，这时意外发生了：系统断电了，结果会造成：文件的内容仍然是老内容，而 meta-data 信息是新内容，二者不一致了。

让我们再看一下 Linux 系统中 fsck 是如何工作的：通常情况下，当 Linux 系统启动时，首先运行 fsck，由它扫描/etc/fstab 文件中列出的所有本地文件系统。fsck 的工作就是确保要装载的文件系统的元数据是处于可使用的状态。当系统关闭时，fsck 又把所有的缓冲区数据转送到磁盘，并确保文件系统被彻底卸载，以保证系统下次启动时能够正常使用。

然而意想不到掉电或者其它故障会导致系统死机、重启。出现这种情况时，操作系统来不及卸载文件系统。重启后，fsck 对磁盘进行彻底扫描，全面地检查元数据，竭尽全能修正检查过程中能找到的所有错误。对所有的元数据做彻底的一致性检查极其耗时。文件系统越大，完成彻底的扫描时间就越长。Fsck 也会碰到它无法修复的磁盘错误。碰到这种情况，就是简单地将文件删除或另存为一个文件。在高密度访问的数据中心，fsck 可能会造成极大的数据文件破坏。只有当 fsck 完成扫描、检查与修复工作后，Linux 系统才能开始使用。当然，如果有严重的文件或数据丢失的话，系统很可能无法重新启动了!

非日志文件系统的种类：

Linux 可以支持种类繁多的文件系统，几乎所有的 Linux 发行版都用 ext2 作为默认的文件系统。Ext2 文件系统就是一个非日志文件系统。此外，Linux 支持的其它非日志文件系统还有：FAT、VFAT、HPFS（OS/2）、NTFS（Windows NT）、Sun 的 UFS 等。

3.2 日志式文件系统

日志文件系统则是在非日志文件系统的基础上，加入了文件系统更改的日志记录。

日志文件的设计思想是：跟踪记录文件系统的变化，并将变化内容记录入日志。日志式文件系统的思想来自于大型数据库系统。数据库操作由多个相关的、相互依赖的子操作组成，任何一个子操作的失败都意味着整个操作的无效性，所以，对数据的任何修改都要求回复到操作以前的状态。日志式文件系统采用了类似的技术。

日志文件系统在磁盘分区中保存有日志记录，写操作首先是对记录文件进行操作，若整个写操作由于某种原因(如系统掉电)而中断，系统重启时，会根据日志记录来恢复中断前的写操作。这个过程只需要几秒钟到几分钟。

日志文件系统是如何工作的？

在日志文件系统中，所有的文件系统的变化、添加和改变都被记录到“日志”(即记录文件 metadata 信息的数据)中。每隔一定时间，文件系统会将更新后的文件 metadata 及文件内

容写入磁盘，之后删除这部分日志。重新开始新日志记录。

在对元数据做任何改变以前，文件系统驱动程序会向日志中写入一个条目，这个条目描述了它将要做些什么。然后，它继续并修改元数据。通过这种方法，日志文件系统就拥有了近期元数据被修改的历史记录，当检查到没有彻底卸载的文件系统的一致性问题时，只要根据数据的修改历史进行相应的检查即可了。也即日志文件系统除了存储数据和元数据（metadata）以外，它们还保存有一个日志，我们可以称之为元元数据（关于元数据的元数据）。

日志文件系统使得数据、文件变安全了，但是系统开销加大了。每一次更新和大多数的日志操作都需要写同步，这需要更多的磁盘 I/O 操作。从日志文件的原理出发，将那些需要经常写操作的分区上使用日志文件系统是一个好的主意。

Linux 系统中可以混合使用日志文件系统或非日志文件系统。日志增加了文件操作的时间，但是，从文件安全性角度出发，磁盘文件的安全性得到了重大的提高。笔者对日志文件系统进行了测试，日志文件系统的性能并不比 ext2 文件系统有太大的性能损失，有的日志文件系统由于采用 B+树算法，在操作一些大尺寸的文件时，性能反面比非日志文件系统的性能还要好。

使用日志文件系统有什么好处？

文件的安全提高了，文件被破坏的机率降低了，对磁盘的扫描时间缩短了，扫描次数减少了。当系统意外宕机后，不会再有文件内容的丢失，至少文件应该保持上一个版本的内容；采用日志文件系统，通常系统每重新启动 20－30 次后，才会对磁盘进行一次整体扫描，扫描次数减少了。

4. 高级文件系统管理

4.1 磁盘与文件结构

文件结构是文件存放在磁盘等存储设备上的组织方法，主要体现在对文件和目录的组织上。目录提供了管理文件的一个方便而有效的途径。我们能够从一个目录切换到另一个目录，而且可以设置目录和文件的许可权，设置文件的共用程度。

使用 Linux，用户可以设置目录和文件的许可权，以便允许或拒绝其他人对其进行访问。Linux 目录采用多级树形结构，用户可以浏览整个系统，可以进入任何一个已授权进入的目录，访问那里的文件。

文件结构的相互关联性使共用资料变得容易，几个用户可以访问同一个文件。 Linux 是一个多用户系统，操作系统本身的驻留程序存放在以根目录开始的专用目录中，有时被指定为系统目录。

内核、Shell 和文件结构一起形成了基本的操作系统结构。它们使得用户可以运行程序、管理文件以及使用系统。此外，Linux 操作系统还有许多被称为实用工具的程序，可以辅助用户完成一些特定的任务。

4.2 硬盘分区

4.2.1 MBR（主引导记录）、启动扇区和分区表

一个硬盘如何分区的信息存在它的第一个扇区（即第一面第一轨第一扇区）。这个第一扇区是硬盘的主引导记录（MBR）；这是电脑启动时 BIOS 读入和启动的扇区。主引导记录包括一段小程序，读入分区表，检查哪个分区是活动分区（即启动分区），并读入活动分区的第一个扇区，即该分区的启动扇区（MBR 也是启动扇区，只不过因为其特殊地位，所以使用特殊的名字）。这个启动扇区包括另一个小程序，读入这个分区（假设是可启动的）上操作系统的第一个部分，然后启动它。

这个分区方案不是内置于硬件和 BIOS 的，只是许多操作系统遵循的约定。但并非所有的操作系统都遵循这个约定。有些操作系统支持分区，但它们占领硬盘上的一个分区，然后使用它们自己的内部分区方法管理这个分区。较新的操作系统可以和其他操作系统和平共处（包括 Linux），而无需特殊的措施，但不支持分区的操作系统无法在同一硬盘上与其他操作系统共存。

为安全预防，最好先在纸上写下分区表，这样在错误发生时不会丢失你的文件。（可以使用 fdisk 修复坏的分区表）。相关信息可用 fdisk -l 命令给出:

```
$ fdisk -l /dev/hda
Disk /dev/hda: 15 heads, 57 sectors, 790 cylinders
Units = cylinders of 855 * 512 bytes
Device Boot Begin Start End Blocks Id System
/dev/hda1 1 1 24 10231+ 82 Linux swap
/dev/hda2 25 25 48 10260 83 Linux native
/dev/hda3 49 49 408 153900 83 Linux native
/dev/hda4 409 409 790 163305 5 Extended
/dev/hda5 409 409 744 143611+ 83 Linux native
/dev/hda6 745 745 790 19636+ 83 Linux native
```

4.2.2 扩展和逻辑分区

PC 硬盘的最初分区方案只允许 4 个分区。实际使用中这太少了，比如有人想装多于 4 个操作系统（Linux, MS-DOS, OS/2, Minix, FreeBSD, NetBSD, Windows/NT 等），或有时一个操作系统有多个分区更好，例如由于速度的原因，Linux 的对换区最好单独使用自己的分区，而不是在主 Linux 分区中（下文详述）。

为克服这个设计问题，发明了扩展分区。这个方法允许将基本分区分为若干子分区，因而被进行子分区的基本分区称为扩展分区，而子分区称为逻辑分区。它们的表现类似基本分割区，但产生方法不同。它们之间没有速度差别。

4.2.3 分区种类

分区表（MBR 和扩展分区里都有）中，对每个分区都有一个字节指出分区种类。这试图确定使用该分区的操作系统，或用于何操作系统。其目的是避免两个操作系统使用同一分区。可实际上，操作系统并不真的注意分区种类字节，例如，Linux 根本不管它是什么。较坏的情况是，有些操作系统错误地使用它，例如有些版本的 DR-DOS 忽略了它的最高位（MSB），而其他一些系统则不是。

没有一个标准化组织定义分区种类字节每个值的意义，但存在一些共同接受的值。相同的列表可以通过 Linux 的 fdisk 命令得到。

4.3 管理软驱和光驱

4.3.1 软盘

如同硬盘，一张软盘也分为磁道和扇区（软盘两面上的相同的磁道组成柱面），但数量要比硬盘少得多。

软驱通常可以使用几种不同的软盘，例如，一个 3.5 英寸软驱可以使用 720KB 和 1.44MB 的软盘。因为软驱操作有些不同，而操作系统必须知道软盘的容量，所以软驱有许多设备文件，每个都与软驱和软盘种类有关。因此，/dev/fd0H1440 是第一个软驱（fd0），必须是 3.5 英寸，使用 3.5 英寸高密度软盘（H），容量是 1440KB（1440），即普通的 3.5 英寸 HD 软盘。

软驱的名字是复杂的，因此 Linux 有一个特定的软驱设备类型，能自动检测软驱中软盘的种类。它使用不同的软盘类型，试图读取新插入的软盘的第一个扇区，直到找到正确的一个。这自然要求软盘是已经格式化过的。

Linux 除了能处理所有标准的软盘，还能处理许多非标准的软盘格式。这有时需要特殊的格式化程序。我们现在先跳过这些软盘格式，同时你可以查看/etc/fdprm 文件，它定义了 setfdprm 识别的设置。

4.3.2 格式化软盘

软盘格式化使用 fdformat 命令。软盘设备使用给定的参数，例如下面的命令是在第一个软驱中格式化一张高密度 3.5 英寸软盘：

```
$ fdformat /dev/fd0H1440
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
$
```

注意，如果想使用自动检测设备（如/dev/fd0），必须先用 setfdprm 设置参数。要得到与上面一样的结果，可以这样：

```
$ setfdprm /dev/fd0 1440/1440
$ fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
$
```

选择与软盘类型相符的正确的设备文件通常更方便。注意，比软盘设计格式化更多的信息容量是没有意义的。

4.3.3 CD-ROM

CD-ROM 驱动器使用一个光学可读的塑胶涂布的碟片。信息记录在盘表面的从中心到边沿的螺旋型小坑上。 驱动器发出一束激光来读盘。当激光射到小坑上，激光以一种方式反射；当它射到光滑表面上，它以另一种方式反射。这很容易地编码成 bit，组成信息。

CD-ROM 驱动器比硬盘慢。典型的硬盘的平均寻道（seek）时间小于 15ms，而快速的 CD-ROM 驱动器要花零点几秒。实际资料传输率则相当快，在数百 KB/s。速度慢使 CD-ROM 驱动器不能代替硬盘使用（有些 Linux distributions 提供“live”CD-ROM 文件系统，使之不必复制文件到硬盘，使安装简单并节约了许多硬盘空间），虽然是可能的。要安装新软件，CD-ROM 很好用，因为在安装时速度并非最重要的。

不同 UNIX 不能使用 ISO9660 文件系统，因此开发了对这个标准的一个增强，叫 Rock Ridge 增强。Rock Ridge 允许长文件名、符号连接和许多其他优点，使 CD-ROM 更像 UNIX 文件系统。同时，Rock Ridge 文件系统仍然是一个有效的 ISO9660 文件系统，使非 UNIX 用户一样可以使用。Linux 同时支持 ISO9660 和 Rock Ridge 增强，增强被自动识别和使用。

文件系统只是一部分，许多 CD-ROM 包含的资料需要特定的程序存取，而多数程序不能运行在 Linux 下（当然，可能运行在 Linux 的 MS-DOS 模拟器 dosemu 下）。

4.4 管理用户的磁盘空间

4.4.1 为普通用户和用户组加入磁盘配额限制

Linux 的 Quota 程序允许为系统上每一用户或用户组指定所能使用的磁盘配额。目前 Quota 仅能工作在 ext2 和 ext3 类型的文件系统上。使用 Quota 需要确定以下两点：

(1) 当前的系统内核支持 Quota。

(2) 系统已正确安装 Quota 套装程序。

如果你当前的系统内核不支持 Quota，请重新编译你的内核，当系统提示：

Quota support(CONFIG—QUOTA)[n]

回答 y，生成新的系统内核。

如果没有 Quota 套装程序，请到以下地址下载 Quota 的源程序并编译：

ftp://ftp.funet.fi/pub/l inux/PEOPLE/Linus/subsystems/Quota/all.tar.gz

一般 Linux 的发行版本的内核都默认包含了 Quota 支持，也附带了 Quota 套装程序，只需安装 Quota 并加以设置便可让 Quota 工作。

Quota 的具体设置步骤：

(1) 编辑系统初始脚本让它检查 Quota 并启动。

```
# Check Quota and then turn Quota on.
if [ －x /usr/sbin/Quotacheck ]
then
echo ″ Checking Quotas. This may take some time.″
/usr/sbin/Quotacheck －avug
echo ″ Done.″
fi
if [ －x /usr/sbin/Quotaon ]
then
    echo ″ Turning on Quota.″
    /usr/sbin/Quotaon －avug
fi
```

(2) 编辑/etc/fstab。你的 /etc/fstab 文件可能会是如下的形式：

```
/dev/hda1 / ext2 defaults 1 1
/dev/hda2 /home ext2 defaults 1 1
```

选择用户所在分区的行的第 4 个域，为用户加入 Quota 支持，如下：

```
/dev/hda1 / ext2 defaults 1 1
/dev/hda2 /home ext2 defaults,usrQuota 1 1
```

如为用户组加入 Quota 支持，可将 usrQuota 替换为 grpQuota。

如两者兼而有之，可将这两项一并写入，如下：

```
/dev/hda1 / ext2 defaults 1 1
/dev/hda2 /home ext2 defaults,usrQuota,grpQuota 1 1
```

(3) 创建 Quota 记录文件 Quota.user 和 Quota.group。进入用户所在分区根目录，如在上例中输入 cd /home 即可，按下面命令创建文件：

```
touch Quota.user
touch Quota.group
chmod 600 Quota.user Quota.group
```

完成上面几步以后，重新启动电脑以使设置生效。

(4) 为用户或用户组设置磁盘配额限制。假设在你的系统上有一名为 bob 的用户，现在想给他 10MB 的硬盘配额限制,他所拥有的最大文件数不得超过 100 个。执行 edQuota －u dquo，系统将进入编辑环境（具体编辑环境视 editor 变量设置而定），将如下 3 行

Quotas for user bob:
/dev/hda2: blocks in use: 14, limits (soft=0, hard=0)
inodes in use: 12, limits (soft=0, hard=0)
改为
Quotas for user bob:
/dev/hda2: blocks in use: 14, limits (soft=0, hard=10240)
inodes in use: 12, limits (soft=0, hard=100)
其中，
blocks in use：用户已使用块的大小，单位是 KB。
inodes in use：用户现有文件的大小。
这两项都是系统自动给出的，不必改动。

4.4.2 软限制（soft limits）

通常设置软限制为一个接近硬限制的值，超越此限制时，系统将警告用户将到达最大磁盘配额限制。软限制为 0 时没有软限制。结合宽限期使用时，只要用户超越了软限制，一过宽限期，任何对磁盘空间的额外需求将被立即拒绝。

4.4.3 硬限制（hard limits）

硬限制磁盘配额的绝对限制，设置了 Quota 的用户不能超越此限制。

4.4.4 宽限期（Grace Period）

宽限期是用户超越了软限制而没有到达硬限制时的一段放宽期，在这段时间内，用户可以在硬限制范围内自由地使用磁盘空间，超过这段时间，所有对磁盘空间的额外需求将被拒绝，即使用户还在硬限制之内。宽限期的单位可以是秒、分、时、天。执行 edQuota －t 命令可设置宽限期。执行该命令后，将系统提示中的两个 0 days 改成你认为合适的值即可。

有时想给一批用户加上同样的限制，比如，给系统上所有 100 个用户加上与 bob 同样的限制，可手工先给 bob 加上限制，然后执行下面命令：

edQuota -p bob ′ awk -F: ′ ＄3 499 {print ＄1}′ /etc/passwd′

给用户组设置磁盘配额限制与普通用户类似，假设有一用户组 game，执行 edQuota -g game 即可。

4.5 控制用户的登录地点

文件/etc/secruity/access.conf 可控制用户登录地点，为了使用 access.conf，必须在文件/etc/pam.d/login 中加入下面的行：

account required /lib/security/pam-access.so

access.conf 文件的格式：

permission : users : origins

其中，

permission：可以是+或-，表示允许或拒绝。

user：可以是用户名、用户组名，如果是 all 则表示所有用户 。

origins：登录地点。local 表示本地，all 表示所有地点，console 表示控制台。另外，origins 也可以是某一个网络。

后面两个域中加上 except 是“除了”的意思。例如，除了用户 wheel, shutdown, sync 禁止所有的控制台登录：

-:ALL EXCEPT wheel shutdown sync:console

root 账户的登录地点不在 access.conf 文件中控制，而是由/etc/securetty 文件控制。

4.6 限制用户每次所发邮件大小

Linux 系统使用 sendmail 发送邮件，配置文件是/etc/sendmail.cf，默认使用 TCP/IP 协定。我们的 Linux 机器上往往会有多个用户同时工作，或者干脆就用它作为邮件服务器，在同一时刻，系统可能要收发很多邮件，因此不能让某一用户过多地占用 sendmail 的时间。sendmail 的默认配置对每次收发邮件的大小没有限制，更改配置文件/etc/sendmail.cf，找到 O MaxMessageSize，去掉行首的＃号，并将其后的数值改为合适的数值，单位为字节。如：

MaxMessageSize = 1048576

意思为每次收发邮件最大为 1MB，任何超过这个值的邮件将被拒绝。

5. 分析 RAMFS 文件系统

RAMFS 是一个非常巧妙的，利用 VFS 自身结构而形成的内存文件系统.RAMFS 没有自己的文件存储结构，它的文件存储于 page cache 中，目录结构由 dentry 链表本身描述,文件则由 VFS 的 inode 结构本身描述。从 RAMFS 可看出,VFS 本质上可看成一种内存文件系统，它统一了文件在内核中的表示方式并对磁盘文件系统进行缓冲。

代码摘录分析如下:

```
; fs/ramfs/inode.c
static struct address_space_operations ramfs_aops = { 文件的低级页操作接口
    readpage:    ramfs_readpage,   读文件页块
    writepage:   ramfs_writepage,  写文件脏页入块设备
    prepare_write:    ramfs_prepare_write,  准备从用户写文件页块
    commit_write:     ramfs_commit_write     从用户写文件页块完成
};
static int ramfs_readpage(struct file *file, struct page * page)
{
    ; 将文件页块读入 page 所描述的页面

    if (!Page_Uptodate(page)) {
        ; 当 lseek()在文件中造成空洞时,会运行到这里
        memset(kmap(page), 0, PAGE_CACHE_SIZE); 页面清零
        kunmap(page);
        flush_dcache_page(page);
        SetPageUptodate(page); 标记为最新
    }
    ;现在的问题是什么情况下系统调用该函数并且 Page_Uptodate(page)为 TRUE.
    UnlockPage(page);
    return 0;
}
static int ramfs_writepage(struct page *page)
{
    ; 将 page cache 中脏页写入块设备.
    ;对于 RAMFS,这里将它重新标记为 DIRTY,防止它们被系统丢弃
```

```
    SetPageDirty(page);
    UnlockPage(page);
    return 0;
}
static int ramfs_prepare_write(struct file *file, struct page *page, unsigned offset,
unsigned to)
{
    ; 系统将用户数据拷贝到 page 之前调用此函数,
    ;offset 是文件指针在页内的起始位置,to 是在页内的终止位置
    ;表示系统将要在 page 从 offset 到 to 的位置上拷贝用户数据.

    void *addr = kmap(page); 取页面描述结构(page)所描述页面的地址
    if (!Page_Uptodate(page)) {
        ; 对于 RAMFS,当 lseek()在文件中产生空洞时,运行到这里.
        memset(addr, 0, PAGE_CACHE_SIZE);
        flush_dcache_page(page);
        SetPageUptodate(page);
    }
    SetPageDirty(page);
    return 0;
}

static int ramfs_commit_write(struct file *file, struct page *page, unsigned offset,
unsigned to)
{
    ; 系统将用户数据拷贝到 page 之后调用此函数
    struct inode *inode = page->mapping->host; //
取 page 所代表的文件,mapping 结构是 inode 的一部分,
    loff_t pos = ((loff_t)page->index << PAGE_CACHE_SHIFT) + to;
    // page->index 表示该 page 在文件中的页块号,to 是该页的终止偏移量
    kunmap(page);
    if (pos > inode->i_size) // inode->i_size 为文件的尺寸
        inode->i_size = pos; 如果文件的写入的终止位置大于文件原来的尺寸,则更新
i_size 的值
    return 0;
}

static struct file_operations ramfs_file_operations = {
    read:       generic_file_read, 数据文件的通用高级读函数
    write:      generic_file_write, 数据文件的通用高级写函数
    mmap:       generic_file_mmap,数据文件的通用高级内存映射函数
    fsync:      ramfs_sync_file,
};
static int ramfs_sync_file(struct file * file, struct dentry *dentry, int datasync)
```

```
{
    return 0;
}
static struct file_operations ramfs_dir_operations = {
    read:       generic_read_dir, 返回-EISDIR 错误的函数
    readdir:  dcache_readdir, 目录文件的高级读目录函数
    fsync:      ramfs_sync_file,
};
static struct inode_operations ramfs_dir_inode_operations = {
    create:     ramfs_create,
    lookup:     ramfs_lookup,
    link:    ramfs_link,
    unlink:     ramfs_unlink,
    symlink: ramfs_symlink,
    mkdir:      ramfs_mkdir,
    rmdir:      ramfs_rmdir,
    mknod:      ramfs_mknod,
    rename: ramfs_rename,
};
static int ramfs_create(struct inode *dir, struct dentry *dentry, int mode)
{
    ; 在目录 dir 内创建一个以 dentry 为目录项的普通文件
    return ramfs_mknod(dir, dentry, mode | S_IFREG, 0);
}
static struct dentry * ramfs_lookup(struct inode *dir, struct dentry *dentry)
{
    ;
对于 ramfs,所有的目录项都已经在 dentry 链表中,只有企图寻找 VFS 中不存在的项时,才会运行到这?br>?br>   d_add(dentry, NULL);
    ; 将 dentry 加入目录链表,置空的 inode,表示 No such file or directories,就是说生成一个"虚"的目录项,
    ;之所以这样,为了加速 create 过程
    return NULL;
}
static int ramfs_link(struct dentry *old_dentry, struct inode * dir, struct dentry * dentry)
{
    struct inode *inode = old_dentry->d_inode;

    ; 在目录 dir 内创建一个以 dentry 所表示目录项的链接,指向 old_entry 目录项所表示的文件

    if (S_ISDIR(inode->i_mode))
        return -EPERM;
```

```
    inode->i_nlink++; 文件的连接数,为 0 表示文件已删除
    atomic_inc(&inode->i_count);   /* New dentry reference */
    dget(dentry);       /* Extra pinning count for the created dentry */
    d_instantiate(dentry, inode);   使目录项 dentry 指向文件 inode
    return 0;
}
static int ramfs_unlink(struct inode * dir, struct dentry *dentry)
{
    int retval = -ENOTEMPTY;

    ; 在目录 dir 内删除 dentry 所表示的目录项

    if (ramfs_empty(dentry)) {  只能删除空目录
        struct inode *inode = dentry->d_inode;

        inode->i_nlink--;
        dput(dentry);           /* Undo the count from "create" - this does all the work
*/
        retval = 0;
    }
    return retval;
}
static int ramfs_symlink(struct inode * dir, struct dentry *dentry, const char *
symname)
{
    int error;
    ; 在目录 dir 中创建名称为 symname 的符号链接文件目录项
    ; 符号链接的名称作为符号链接文件的数据体存放在 page cache 中

    error = ramfs_mknod(dir, dentry, S_IFLNK | S_IRWXUGO, 0);
    if (!error) {
        int l = strlen(symname)+1;
        struct inode *inode = dentry->d_inode;
        error = block_symlink(inode, symname, l);
    }
    return error;
}
static int ramfs_mkdir(struct inode * dir, struct dentry * dentry, int mode)
{
    ; 在目录 dir 内创建一个以 dentry 为目录项的目录
    return ramfs_mknod(dir, dentry, mode | S_IFDIR, 0);
}
static int ramfs_rename(struct inode * old_dir, struct dentry *old_dentry, struct
```

```
inode * new_dir,struct dentry *new_dentry)
{
    int error = -ENOTEMPTY;
    ; 将目录 old_dir 内的目录项改名为 new_dir 目录中的 new_dentry 所描述的目录项

    if (ramfs_empty(new_dentry)) {
        struct inode *inode = new_dentry->d_inode;
        if (inode) {
            inode->i_nlink--;
            dput(new_dentry);
        }
        error = 0;
    }
    return error;
}
; 在 VFS 中创建一个节点,用来描述文件,目录或符号链接
static int ramfs_mknod(struct inode *dir, struct dentry *dentry, int mode, int dev)
{
    struct inode * inode = ramfs_get_inode(dir->i_sb, mode, dev);
    int error = -ENOSPC;
    if (inode) {
        d_instantiate(dentry, inode); 将目录项 dentry 与文件描述 inode 相关联
        dget(dentry);       /* Extra count - pin the dentry in core */
        error = 0;
    }
    return error;
}

static struct super_operations ramfs_ops = {
    statfs:         ramfs_statfs,
    put_inode:      force_delete,
};
static int ramfs_statfs(struct super_block *sb, struct statfs *buf)
{
    ; 取文件系统结构参数
    buf->f_type = RAMFS_MAGIC;
    buf->f_bsize = PAGE_CACHE_SIZE;
    buf->f_namelen = 255;
    return 0;
}

; 获取一个属于 ramfs 的自由的 inode.
struct inode *ramfs_get_inode(struct super_block *sb, int mode, int dev)
{
```

```
        struct inode * inode = new_inode(sb);

        if (inode) {
            inode->i_mode = mode;
            inode->i_uid = current->fsuid;
            inode->i_gid = current->fsgid;
            inode->i_blksize = PAGE_CACHE_SIZE;
            inode->i_blocks = 0;
            inode->i_rdev = to_kdev_t(dev);
            inode->i_mapping->a_ops = &ramfs_aops;
            inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
            switch (mode & S_IFMT) {
            default:
                init_special_inode(inode, mode, dev); 与设备文件相联的 inode 函数
                break;
            case S_IFREG: 普通文件的 inode 函数
                inode->i_fop = &ramfs_file_operations;
                break;
            case S_IFDIR: 目录文件的 inode 函数
                inode->i_op = &ramfs_dir_inode_operations;
                inode->i_fop = &ramfs_dir_operations;
                break;
            case S_IFLNK: 符号链接的 inode 函数
                inode->i_op = &page_symlink_inode_operations;
                break;
            }
        }
        return inode;
    }
    static int ramfs_empty(struct dentry *dentry) 判断目录是否为空
    {
        struct list_head *list;

        spin_lock(&dcache_lock);
        list = dentry->d_subdirs.next; 取子目录链

        while (list != &dentry->d_subdirs) {
            struct dentry *de = list_entry(list, struct dentry, d_child);

            if (ramfs_positive(de)) {
                spin_unlock(&dcache_lock);
                return 0; 如果 dentry 目录中存在一个实在的子目录项,则说明该目录非空.
            }
            list = list->next;
```

```
        }
        spin_unlock(&dcache_lock);
        return 1;
}
static inline int ramfs_positive(struct dentry *dentry)
{
        ; 判断 dentry 是否实在,即 dentry 指向某个文件并且被 dentry 的散列表索引
        return dentry->d_inode && !d_unhashed(dentry);
}
; fs/devices.c
void init_special_inode(struct inode *inode, umode_t mode, int rdev)
{
        inode->i_mode = mode;
        if (S_ISCHR(mode)) { 如果为字符设备文件,提供字符设备的 inode 函数
                inode->i_fop = &def_chr_fops;
                inode->i_rdev = to_kdev_t(rdev);
        } else if (S_ISBLK(mode)) { 如果为块设备文件,提供块设备的 inode 函数
                inode->i_fop = &def_blk_fops;
                inode->i_rdev = to_kdev_t(rdev);
                inode->i_bdev = bdget(rdev); 指向块设备描述结构
        } else if (S_ISFIFO(mode)) 如果为 FIFO 文件,提供 FIFO 的 inode 函数
                inode->i_fop = &def_fifo_fops;
        else if (S_ISSOCK(mode)) 如果为 SOCK 文件,提供 SOCK 的 inode 函数
                inode->i_fop = &bad_sock_fops;
        else
                printk(KERN_DEBUG "init_special_inode: bogus imode (%o)\n", mode);
}
; fs/buffer.c
int block_symlink(struct inode *inode, const char *symname, int len)
{
        struct address_space *mapping = inode->i_mapping;
        struct page *page = grab_cache_page(mapping, 0);
        int err = -ENOMEM;
        char *kaddr;
        ; 符号链接中的符号名存放在 page cache 中.
        if (!page)
                goto fail;
        err = mapping->a_ops->prepare_write(NULL, page, 0, len-1);
        if (err)
                goto fail_map;
        kaddr = page_address(page);
        memcpy(kaddr, symname, len-1);
        mapping->a_ops->commit_write(NULL, page, 0, len-1);
        /*
```

```
         * Notice that we are _not_ going to block here - end of page is
         * unmapped, so this will only try to map the rest of page, see
         * that it is unmapped (typically even will not look into inode -
         * ->i_size will be enough for everything) and zero it out.
         * OTOH it's obviously correct and should make the page up-to-date.
         */
        err = mapping->a_ops->readpage(NULL, page);
        wait_on_page(page);
        page_cache_release(page); 释放 grab 状态
        if (err < 0)
            goto fail;
        mark_inode_dirty(inode);
        return 0;
    fail_map:
        UnlockPage(page);
        page_cache_release(page);
    fail:
        return err;
    }
```

6. S3C2410 上 Jffs2 的移植

(1) 移植环境：

CPU：ARMS3C2410

Linux version:2.4.18

Flash：Intel E28F128

(2) 修改设备号

由于 ROM 设备和 MTDBlock 设备的主设备号（major）都是 31，所以如果你不想把 JFFS2 作为根文件系统的话，必须修改他们之一的 major。如果你要修改 JFFS2 的设备 major，在 uClinux-dist/linux-2.4.x/include/linux/mtd/mtd.h 中把

```
#define MTD_BLOCK_MAJOR    31
```

改成

```
#define MTD_BLOCK_MAJOR    30
```

(3) 编写 Maps 文件

添加在 flash 上的 map 文件。在 RUIJIARM2410-R3/kernel/drivers/mtd/maps 下添加 flash(e28f128j3a-150)的 map。

```
//以下是分区的内容，当然要根据你自己的需要确定了
static struct mtd_partition s3c2410_partitions[] = {
    {
            name: "reserved for bootloader",
            size: 0x040000,
            offset: 0x0,
        mask_flags: MTD_WRITEABLE,
    },
    {
```

```
                name: "reserved for kernel",
                size: 0x0100000,
                offset: 0x040000,
            mask_flags: MTD_WRITEABLE,
        },
        {
                name: "reserved for ramdisk",
                size: 0x400000,
                offset: 0x140000,
            mask_flags: MTD_WRITEABLE,

        },
    {
        name: "jffs2(8M)",
        size: 0x800000,
        offset: 0x800000,
    }
    };

    int __init init_s3c2410(void)
    {
        printk(KERN_NOTICE "s3c2410 flash device: %x at %x\n", WINDOW_SIZE,
WINDOW_ADDR);
    s3c2410_map.map_priv_1 = (unsigned long)ioremap(WINDOW_ADDR,
WINDOW_SIZE);
    //printk("0\n");
    if (!s3c2410_map.map_priv_1) {
        printk("Failed to ioremap/n");
        return -EIO;
    }
    //printk("1\n");
    mymtd = do_map_probe("jedec_probe", &s3c2410_map);
    if (!mymtd)
    mymtd = do_map_probe("cfi_probe", &s3c2410_map);
    //printk("2\n");
    if (mymtd) {
        mymtd->module = THIS_MODULE;
        mymtd->erasesize = 0x20000; //擦除的大小 INTEL E28F128J3A-150 是
128kb
        return add_mtd_partitions(mymtd, s3c2410_partitions,
sizeof(s3c2410_partitions) / sizeof(struct mtd_partition));
    }
    //printk("3\n");
    iounmap((void *)s3c2410_map.map_priv_1);
```

```
 return -ENXIO;
}

static void __exit cleanup_s3c2410(void)
{
 if (mymtd) {
       del_mtd_partitions(mymtd);
       map_destroy(mymtd);
 }
 if (s3c2410_map.map_priv_1) {
       iounmap((void *)s3c2410_map.map_priv_1);
       s3c2410_map.map_priv_1 = 0;
 }
}

module_init(init_s3c2410);
module_exit(cleanup_s3c2410);
```

至于其文件内容及语句的含义网上相关的文章也有不少，参考一下吧。

(4) 将配置加入 RUIJIARM2410-EDU/kernel/drivers/mtd/maps/Config.in

```
if [ "$CONFIG_ARM" = "y" ]; then
    dep_tristate '  CFI Flash device mapped on ARM Integrator/P720T' CONFIG_MTD_ARM_INTEGRATOR $CONFIG_MTD_CFI $CONFIG_ARCH_INTEGRATOR
    dep_tristate '  Cirrus CDB89712 evaluation board mappings' CONFIG_MTD_CDB89712 $CONFIG_MTD_CFI $CONFIG_ARCH_CDB89712
    dep_tristate '  CFI Flash device mapped on StrongARM SA11x0' CONFIG_MTD_SA1100 $CONFIG_MTD_CFI $CONFIG_ARCH_SA1100 $CONFIG_MTD_PARTITIONS
    dep_tristate '  CFI Flash device mapped on DC21285 Footbridge' CONFIG_MTD_DC21285 $CONFIG_MTD_CFI $CONFIG_ARCH_FOOTBRIDGE $CONFIG_MTD_PARTITIONS
    dep_tristate '  CFI Flash device mapped on Lubbock board' CONFIG_MTD_LUBBOCK $CONFIG_MTD_CFI $CONFIG_ARCH_LUBBOCK $CONFIG_MTD_PARTITIONS
    dep_tristate '  CFI Flash device mapped on the FortuNet board' CONFIG_MTD_FORTUNET $CONFIG_MTD_CFI $CONFIG_ARCH_FORTUNET $CONFIG_MTD_PARTITIONS
    dep_tristate '  CFI Flash device mapped on Epxa10db' CONFIG_MTD_EPXA10DB $CONFIG_MTD_CFI $CONFIG_MTD_PARTITIONS $CONFIG_ARCH_CAMELOT
    dep_tristate '  CFI Flash device mapped on PXA CerfBoard' CONFIG_MTD_PXA_CERF $CONFIG_MTD_CFI $CONFIG_ARCH_PXA_CERF $CONFIG_MTD_PARTITIONS
    dep_tristate '  NV-RAM mapping AUTCPU12 board'
```

CONFIG_MTD_AUTCPU12 $CONFIG_ARCH_AUTCPU12
#wpq add S3C2410 的 CFI 配置
dep_tristate ' CFI Flash device mapped on S3C2410' CONFIG_MTD_S3C2410 $CONFIG_MTD_CFI
fi

(5) 修改 Makefile 文件

在 RUIJIARM2410-R3/kernel/drivers/mtd/maps/ Makefile 文件中加入如下语句(当然要根据你的实际情况写啊？！)：

#wpq add
obj-$(CONFIG_MTD_S3C2410) += s3c2410_wpq.o

(6) 配置内核使其支持 jffs2

说明：

这里要特别注意 Memory Technology Devices (MTD)的选项支持及其子项
RAM/ROM/Flash chip drivers --->
Mapping drivers for chip access --->
的选项支持；
还有 File systems 下选项支持。

```
####################################
**********************
################################################
 Linux Kernel v2.4.18-rmk7-pxa1 Configuration

Linux Kernel v2.4.18-rmk7-pxa1 Configuration
 ------------------------------------------------------------------------------
  +--------------------- Memory Technology Devices (MTD) ---------------------+
  |   Arrow keys navigate the menu.   <Enter> selects submenus --->.            |
  |   Highlighted letters are hotkeys.   Pressing <Y> includes, <N> excludes,   |
  |   <M> modularizes features.   Press <Esc><Esc> to exit, <?> for Help.       |
  | Legend: [*] built-in   [ ] excluded   <M> module   < > module capable      |
  | +---------------------------------------------------------------------+ |
  | |     <*> Memory Technology Device (MTD) support                        | |
  |  | |           [*] Debugging                                            | |
  | |             (3)     Debugging verbosity (0 = quiet, 3 = noisy)        | |
  | |       <*>     MTD partitioning support                               | |
  | |   <*>     MTD concatenating support                                   | |
  | |     < >     RedBoot partition table parsing                          | |
  | |     < >     Command line partition table parsing                     | |
  | |     < >     ARM Firmware Suite partition parsing                     | |
  | |     --- User Modules And Translation Layers                          | |
  | |   <*>     Direct char device access to MTD devices                   | |
  | | <*>     Caching block device access to MTD devices                   | |
  |   < >     FTL (Flash Translation Layer) support                        | |
```

```
  | | < >     NFTL (NAND Flash Translation Layer) support                  | |
  | |       RAM/ROM/Flash chip drivers    --->                               | |
  | |         Mapping drivers for chip access    --->                       | |
  | |     Self-contained MTD device drivers    --->                         | |
  | |     NAND Flash Device Drivers    --->                                   | |
  | +-------------------------------------------------------------------+ |
  +------------------------
  | +---------v(+)-----------------------------------------------------+ |
  +-----------------------------------------------------------------------+
  |                     <Select>       < Exit >       < Help >                 |
  +-----------------------------------------------
```

```
 Linux Kernel v2.4.18-rmk7-pxa1 Configuration
 ------------------------------------------------------------------------------
  +---------------------- RAM/ROM/Flash chip drivers ----------------------+
  |   Arrow keys navigate the menu.    <Enter> selects submenus --->.          |
  |   Highlighted letters are hotkeys.    Pressing <Y> includes, <N> excludes, |
  |   <M> modularizes features.    Press <Esc><Esc> to exit, <?> for Help.     |
  | Legend: [*] built-in    [ ] excluded    <M> module    < > module capable    |
  | +---------------------------------------------------------------+ |
  | |<*>     Detect flash chips by Common Flash Interface (CFI) probe       | |
  | |<*> Detect JEDEC JESD21c compatible flash chips                        | |
  | |[ ]     Flash chip driver advanced configuration options               | |
  | |<*>     Support for Intel/Sharp flash chips                            | |
  | |< >     Support for AMD/Fujitsu flash chips                        | |
  | |< > Support for RAM chips in bus mapping                               | |
  | |< > Support for ROM chips in bus mapping                               | |
  | |< >     Support for absent chips in bus mapping                        | |
  | |[ ]     Older (theoretically obsoleted now) drivers for non-CFI chips  | |
   | |                                                                                |
   |
  | |
  | +---------------------------------------------------------------+ |
  +-----------------------------------------------------------------------+
  |                <Select>       < Exit >       < Help >                   |
  +-----------------------------------------------------------------------+
```

```
Linux Kernel v2.4.18-rmk7-pxa1 Configuration
 ------------------------------------------------------------------------------
  +-------------------- Mapping drivers for chip access --------------------+
  |   Arrow keys navigate the menu.    <Enter> selects submenus --->.          |
  |   Highlighted letters are hotkeys.    Pressing <Y> includes, <N> excludes, |
  |   <M> modularizes features.    Press <Esc><Esc> to exit, <?> for Help.     |
  |Legend: [*] built-in    [ ] excluded    <M> module    < > module capable    |
```

```
| +----------------------------------------------------------------------+ |
| |   <*>     CFI Flash device in physical memory map                    | |
| |          (800000)        Physical start address of flash mapping     | |
| |          (800000)        Physical  length  of  flash  mapping        |    | |
(2)        Bus width in octets                                           | |
| |    <*>     CFI Flash device mapped on S3C2410                        | |
| |                                                                      | |
| |                                                                      | |
| |                                                                      | |
| |
| |
| |                                                                      | |
| +----------------------------------------------------------------------+ |
+------------------------------------------------------------------------+
|                   <Select>       < Exit >       < Help >                |
+------------------------------------------------------------------------+
```

Linux Kernel v2.4.18-rmk7-pxa1 Configuration

```
------------------------------------------------------------------------------
 +---------------------------- File systems -----------------------------+
 |   Arrow keys navigate the menu.   <Enter> selects submenus --->.       |
 |   Highlighted letters are hotkeys.   Pressing <Y> includes, <N> excludes, |
 |   <M> modularizes features.   Press <Esc><Esc> to exit, <?> for Help.  |
 | Legend: [*] built-in   [ ] excluded   <M> module   < > module capable  |
 | +--------------------------------------------------------------------+ |
 || [ ]   Quota support                                                  | |
 | | < > Kernel automounter support                                     | |
 | |   < > Kernel automounter version 4 support (also supports v3)      | |
 | |   < > Reiserfs support                                             | |
 | |   < > ADFS file system support                                     | |
 || < > Amiga FFS file system support (EXPERIMENTAL)                    | |
 | |   < > Apple Macintosh file system support (EXPERIMENTAL)           | |
 | |   < > BFS file system support (EXPERIMENTAL)                       | |
 | |   <*> Ext3 journalling file system support (EXPERIMENTAL)          | |
 | |   [ ]    JBD (ext3) debugging support                              | |
 | |   <*> DOS FAT fs support                                           | |
|   < >     MSDOS fs support                                           | |
 | |   <*>     VFAT (Windows-95) fs support                             | |
 | |   < > EFS file system support (read only) (EXPERIMENTAL)           | |
 | |   < > Journalling Flash File System (JFFS) support                 | |
 | |   <*> Journalling Flash File System v2 (JFFS2) support             | |
 | |   (2) JFFS2 debugging verbosity (0 = quiet, 2 = noisy)             | |
 | |   <*> Compressed ROM file system support                           | |
 | |   [*] Virtual memory file system support (former shm fs)           | |
```

```
| |   <*> Simple RAM-based file system support                    | |
| |   < > ISO 9660 CDROM file system support                      | |
| |   < > Minix fs support
|   < > FreeVxFS file system support (VERITAS VxFS(TM) compatible)    | |
| |   < > NTFS file system support (read only)                      | |
| |   < > OS/2 HPFS file system support                           | |
| |   [*] /proc file system support                             | |
| |   [*] /dev file system support (EXPERIMENTAL)                 | |
| |   [*]     Automatically mount at boot                       | |
| |   [ ]     Debug devfs                                     | |
| |   [*] /dev/pts file system for Unix98 PTYs                    | |
| |   < > QNX4 file system support (read only) (EXPERIMENTAL)     | |
| |   < > ROM file system support                                 | |
| |   <*> Second extended fs support
|   [ ]     Debug devfs                                         | |
| |   [*] /dev/pts file system for Unix98 PTYs                    | |
| |   < > QNX4 file system support (read only) (EXPERIMENTAL)     | |
| |   < > ROM file system support                               | |
| |   <*> Second extended fs support                              | |
| |   < > System V/Xenix/V7/Coherent file system support          | |
| |   < > UDF file system support (read only)                     | |
| |   < > UFS file system support (read only)                     | |
| |   Network File Systems    --->                              | |
| |   Partition Types    --->                                   | |
| |   Native Language Support    --->
| +--v(+)-------------------------------------------------------+ |
+-----------------------------------------------------------------+
|                          <Select>       < Exit >       < Help >          |
+-----------------------------------------------------------------+
```

(7) 制作 jffs2 映象

首先取得 jffs2 的制作工具：mkfs.jffs2（可从网上取得）

执行如下命令即可生成所要的映象：

chmod 777 mkfs.jffs2 //取得 mkfs.jffs2 的执行权限，即 mkfs.jffs2 成为可执行文件

./mkfs.jffs2 －d jffs2/ －o jffs2.img //生成 jffs2 文件映象，其中目录 jffs2 可以是任意的目录，这里的 jffs2 是我新建的一个目录

(8) Jffs2 的应用

对于 ppcboot、zImage、ramdisk.image.gz 向 romfs 一样正常烧写；

以上三项烧写完之后，接着烧写 jffs2.img，具体烧写如下：

tftp 30800000 jffs2.img

fl 1800000 30800000 20000 （其中 20000 可根据 jffs2 的大小适当调整，理论上只要比 jffs2.img 略大即可，但要为 20000 的整数倍）

特别注意：要想使我们做的 jffs2 文件系统更加的人性化，我们还可以在 ramdisk.image.gz

的 mnt/etc/init.d/rc$文件中加入如下指令以便启动时自动挂载 jffs2 文件系统。

Mount －t jffs2 /dev/mtdblock/4 /mnt //其中的/dev/mtdblock/4 是 flash 上的 jffs2 分区。

(9) 以上配置烧写完成之后就可启动我们的系统，对 jffs2 分区尽情的添加和删除了，添加的东东再不会因断电而丢失了，呵呵，就到这了。

7. Cramfs 的移植

生成 image：

在 kernel/script/cramfs 有生成 image 的程序的源代码，编译之后生成 mkcramfs

然后，建立一个目录，将需要放到文件系统的文件 copy 到这个目录，mkcramfs 目录名 image 名，就可以生成一个 cramfs 文件系统的 image 文件

如果目录名为 mnt，则该语句为：mkcramfs mnt mnt.cramfs

内核支持：

在 kernel 目录 make menuconfig，配置内核需求

在选中 MTD（Memory Technology Device)设备

我选择直接编译到内核中去，这样就不用 insmod 了

选中其中的相关选项，可以根据自己的使用情况定制，

多选中一些只是使启动变得慢一些，没有什么其他坏处。

在其中配置文件烧写到 flash 中的起始地址和文件长度

在 physical mapping by chips(大概是这样的一个选项）

选中起始地址为：0x1800000,长度为 0x800000，buswidth 为 2

这个可以根据自己的实际情况进行选择，

flash 的起始地址为 0x1000000,

botloader:0x1000000-0x1040000

kernal:0x1040000-0x1140000

ramdisk:0x1140000-????

ramdisk 使用剩余的部分都可以用来做 cramfs 系统

一般 8M 的 ramdisk 压缩以后为 3M，这样 cramfs 可以从 0x1500000 开始烧写

安全起见，我们的设置方法为 flash 的后半部分（后面 8M：0x1800000-0x1ffffff)为 cramfs 所占用的区间。

其中因为 cramfs 文件系统不是作为系统的根文件系统，所以 mtdblock0 的主设备号要从 31 改为 30

这个修改在文件 kernel/include/linux/mtd/mtd.h 中

然后重新编译内核就可以了。

文件系统的烧写：

重新烧写编译后的系统内核 zImage

tftp 30008000 zImage;fl 1040000 30008000 e0000

重新启动以后继续烧写新的 cramfs 文件系统

如上面的文件名

tftp 30008000 mnt.cramfs;fl 1800000 30008000 xxxxxx

其中 xxxxxx 为 mnt.cramfs 文件的实际大小

系统启动以后，就可以 mount cramfs 文件系统了

在根目录下建立需要 mount 的目录

cd /
mkdir cramfs
mount -t cramfs /dev/mtdblock/0 cramfs
然后，你就可以 cd cramfs，看到你原来 copy 过来的东西了
cramfs 支持的文件系统大小为 256M，单个文件的大小限制在 16M，相信可以满足基本的需求了。
将 cramfs 做为根文件系统目前还有一定的困难。
Cramfs 需要在内核中的文件系统中增加对 compressed rom disk 的支持。

8. 将 Cramfs，jffs2，ramfs 移植到同一个文件系统中
文件系统对 jffs2 和 Cramfs 的支持都已经编入内核，用户只需要设置各个文件系统分配不同的 flash 的大小和区间就可以了。同时对应的将生成的 jffs2 和 cramfs 的 disk 烧写的 flash 中相应的位置就可以了。
代码中相应的更改如下，参见 kernel/drivers/mtd/maps/s3c2410_llg.c 文件第 76 行开始。

```
static struct mtd_partition s3c2410_partitions[] = {
{
        name: "reserved for bootloader",
        size: 0x040000,
        offset: 0x0,
        mask_flags: MTD_WRITEABLE,
},
{
        name: "reserved for kernel",
        size: 0x0100000,
        offset: 0x040000,
        mask_flags: MTD_WRITEABLE,
},
{
        name: "reserved for ramdisk",
        size: 0x400000,
        offset: 0x140000,
        mask_flags: MTD_WRITEABLE,
},
{
            name: "jffs2(8M)",
            size: 0x800000,
            offset: 0x800000,
 },
{
            name: "cramfs(2.75M)",
            size:0x2c0000,
            offset:0x540000,
}
};
```

前 3 个分区对应着系统的 bootloader、kernel、ramdisk，其实这部分并不通过 MTD 来访问，把他们写在这里主要是为了便于理解，为了代码的完整性，我们没有删除上面的三个分区。

我们主要注意第四个和第五个分区。

第四个分区是 jffs2 分区，其中 offset 表示该分区在 flash 中的相对位置，（flash 的起始地址为 0），size 表示该分区的大小。Jffs2 的配置表示 jffs2 分区位于 flash 上从 8M 开始 8M 大小的分区上。

我们是后来增加的对 cramfs 文件系统的支持，此时 flash 上只剩下从 0x540000~0x800000 的地址空间可用，所以我们就把 cramfs 增加在 flash 的这一部分。如上 cramfs 的配置表示 cramfs 分区位于 flash 上从 0x540000 开始，大小为 2c0000 的分区上，事实上，cramfs 是一个压缩的文件系统，这 2.7M 大小的分区大概可以支持 8M 的空间的 disk。

对应 jffs2 和 cramfs 的生成和烧写命令如下：

jffs2 disk 的生成：(在 pc 机上)

```
./mkfs.jffs2 －d　jffs2/　－o　jffs2.img　//jffs2 表示含该分区文件的目录
cp –f jffs2.imag /tftpboot
```

jffs2 disk 的烧写，在板子上 ppcboot 方式下：

```
tftp 30008000 jffs2.img
fl 1800000 30008000 ****** (******的大小要大于 jffs2.img 文件的大小)
```

cramfs disk 的生成：（在 pc 机上）

```
mkcramfs mnt mnt.cramfs (mnt 表示包含该文件系统的目录)
cp –f mnt.cramfs /tftpboot
```

cramfs disk 的烧写：在板子的 ppcboot 方式下

```
tftp 30008000 mnt.cramfs
fl 1540000 30008000 ******　(******的大小要大于 mnt.cramfs 文件的大小)
```

系统启动以后，用户可以 mount cramfs 文件系统

```
cd /
mkdir cramfs
mount –t cramfs /dev/mtdblock/5 cramfs
```

jffs2 文件系统也同样可以 mount

```
cd /
mkdir jffs2
mount －t　jffs2　/dev/mtdblock/4　/jffs2
```

当然，用户可以将这部分写入到 mnt/etc/init.d/rcS 文件中，这样系统启动以后就自动 mount 文件系统。

如果用户想重新进行分区，则只需要更改这一部分上述部分，比如用户现在想生成 4M 的 cramfs 分区，其余部分给 jffs2 分区，则上述代码应更改如下：

```
static struct mtd_partition s3c2410_partitions[] = {
    {
            name: "reserved for bootloader",
            size: 0x040000,
            offset: 0x0,
            mask_flags: MTD_WRITEABLE,
    },
```

```
    {
            name: "reserved for kernel",
            size: 0x0100000,
            offset: 0x040000,
            mask_flags: MTD_WRITEABLE,
    },
    {
            name: "reserved for ramdisk",
            size: 0x400000,
            offset: 0x140000,
            mask_flags: MTD_WRITEABLE,
    },
    {
                name: "jffs2(6.75M)",
                size: 0x6c0000,
                offset: 0x940000,
     },
    {
                name: "cramfs(4M)",
                size:0x400000,
                offset:0x540000,
    }
};
```

对应 jffs2 和 cramfs 的烧写命令如下：

jffs2 disk 的烧写，在板子上 ppcboot 方式下：

```
tftp 30008000 jffs2.img
fl 1940000 30008000 ****** (******的大小要大于 jffs2.img 文件的大小)
```

cramfs disk 的烧写：在板子的 ppcboot 方式下

```
tftp 30008000 mnt.cramfs
fl 1540000 30008000 ******   (******的大小要大于 mnt.cramfs 文件的大小)
```

9. 各种文件系统的选择

通常情况下，ROMfs 是使用最多的文件系统，它是一种简单、紧凑和只读的文件系统。ROMfs 顺序存储文件数据，并可以在操作系统支持的存储设备上直接运行文件系统，这样可以在系统运行时节省许多 RAM 空间。

Cramfs 是针对 Linux 内核 2.4 之后的版本所设计的一种新型文件系统，也是压缩和只读格式的。它主要的优点是将文件数据以压缩形式存储，在需要运行的时候进行解压缩。由于它存储的文件形式是压缩的格式，所以文件系统不能直接在 Flash 上运行。虽然这样可以节约很多 Flash 存储空间，但是文件系统运行需要将大量的数据拷贝进 RAM 中，消耗了 RAM 空间。

考虑到多数系统需要能够读/写的文件系统，可以使用 MTD driver 的 JFFS 和 JFFS2 日志式文件在 Flash 头部建立根文件系统（Root Filesystem）。日志式文件系统可以免受系统突然

掉电的危险，并且在下一次系统引导时不需要文件系统的检查。由于 JFFS 和 JFFS2 文件格式是特别为 Flash 存储器设计的，二者都具一种称为“损耗平衡”的特点，也就是说 Flash 的所有被擦写的单元都保持相同的擦写次数。利用这些特有保护措施，Flash 的使用周期得到相当大的提升。JFFS2 使用压缩的文件格式，为 Flash 节省了大量的存储空间，它更优于 JFFS 格式在系统中使用。值得注意的是，使用 JFFS2 格式可能带来少量的 Flash 空间的浪费，这主要是由于日志文件的过度开销和用于回收系统的无用存储单元，浪费的空间大小约是两个数据段。

如果使用 RAM disk，一般应选择 EXT2 文件格式，但 EXT2 并不是一块特别高效的文件存储空间。由于存在 RAM disk 上，所以任何改变在下一次启动后都会丢失。当然，也有许多人认为对嵌入式存储空间来讲，这是一种优势，因为每次系统启动都是从已知的文件系统状态开始的。

三．实验内容和步骤

1．cramfs：

制作 crafms 的电子盘，在 PC 机上创建一个目录，如 mnt，然后将 copy 一些目录、文件到这个目录中，这些文件是希望出现在 cramfs 只读文件系统的目录、文件。然后执行 mkcramfs mnt cramfs.img 生成一个 cramfs 的电子盘。将该电子盘复制到/tftpboot 目录中以备烧写只用。

在板子加电以后按回车，激活 ppcboot，执行 tftp 30008000 cramfs.img，将 cramfs 的电子盘读到内存的 0x30008000 处。执行 fl 1540000 30008000 20000，其中 20000 为电子盘的文件大小，如果 cramfs.img 文件大小超过 0x20000，则将增加该值，使之超过 cramfs.img 的文件大小。

重新启动板子，观看板子中/cramfs 目录的新文件是否为刚才烧写的新文件。

尝试写目录和其他目录，观察只读文件系统和非只读文件系统的区别。

2．jffs2

制作 jffs2 的电子盘，在 PC 机上创建一个目录，如 jffs2，然后将 copy 一些目录、文件到这个目录中，这些文件是希望出现在 jffs2 文件系统的目录、文件。然后执行./mkfs.jffs2 –d jffs2 –o jffs2.img 生成一个 jffs2 的电子盘。将该电子盘复制到/tftpboot 目录中以备烧写只用。

在板子加电以后按回车，激活 ppcboot，执行 tftp 30008000 jffs2.img，将 jffs2 的电子盘读到内存的 0x30008000 处。执行 fl 1800000 30008000 20000，其中 20000 为电子盘的文件大小，如果 jffs2.img 文件大小超过 0x20000，则将增加该值，使之超过 jffs2.img 的文件大小。

重新启动板子，观看板子中/jffs2 目录的新文件是否为刚才烧写的新文件。

在/jffs2 目录创建新的目录文件，重新加电之后观察这些目录文件是否存在。观察可读写文件系统和只读文件系统 cramfs 的区别。

3．直接读写 flash

准备好 save recover ppcflash.o 三个文件。

通常 ppcflash.o 在/RUIJIARM2410-R3/applications/save-to-flash/ppcflash_driver 目录，save 和 recover 在/RUIJIARM2410-R3/applications/save-to-flash/save_recover，在相应的目录下执行 make 可以重新生成这些文件。

Mount 这些文件到板子上，执行 insmod ppcflash.o; ./save 将"/etc/passwd","/etc/inetd.conf"两个文件烧写到板子的 0x01530000 开始的 4k 范围内。

重启板子，在 ppcboot 下面使用 md 1530000 100,来观察 flash 上内容的变化。

在板子上执行./recover，将上述两个文件回复到/tmp/hehe0,/tmp/hehe1 中。观察这两个文件的内容和"/etc/passwd","/etc/inetd.conf"是否相同，如果相同，则读写 flash 成功。

修改 save.c 的 char filename[SIZE][32]={"/etc/passwd","/etc/inetd.conf"};部分，这里保存了需要存到 flash 上的文件列表，将其改成其他文件，重复上述步骤，观察烧写到 flash 的 0x1530000 部分的内容的变化，执行./recover，观察回复出的/tmp/hehe0 和/tmp/hehe1 文件的变化。

四．思考题

1. 比较各种文件系统的优缺点。
2. 从网上查找资料，分别将 jffs2 和 cramfs 作为系统的根文件系统加载。

实验二十八： 2.6 内核移植实验

一．实验目的

通过本实验，使学生掌握 ARM 交叉编译环境的建立方法，内核移植步骤和方法。

二. 实验原理和说明

本实验通过在 S3C2410 上移植 linux2.6 内核，介绍在嵌入式处理器上移植 Linux 的基本

步骤和方法。介绍交叉编译环境的建立过程。

1．在 Host 机上如何建立 arm-linux-gcc 交叉编译环境：

下面我们介绍如何在 readhat7.2，8.0，9.0 上建立 arm-linux-gcc 的交叉编译环境，gcc 版本是 gcc-2.95.3。

采用的源代码版本和下载路径如下：

binutils-2.14.tar.gz ftp://ftp.gnu.org/gnu/binutils/binutils-2.14.tar.gz

gcc-core-2.95.3.tar.gz ftp://ftp.gnu.org/gnu/gcc/gcc-2.95.3/gcc-core-2.95.3.tar.gz

gcc-g++2.95.3.tar.gz ftp://ftp.gnu.org/gnu/gcc/gcc-2.95.3/gcc-g++-2.95.3.tar.gz

glibc-2.2.4.tar.gz ftp://ftp.gnu.org/gnu/glibc/glibc-2.2.4.tar.gz

glibc-linuxthreads-2.2.4.tar.gz ftp://ftp.gnu.org/gnu/glibc/glibc-linuxthreads-2.2.4.tar.gz

linux-2.4.21.tar.gz ftp://ftp.kernle.org/pub/linux/kernel/v2.4/linux-2.4.21.tar.gz

patch-2.4.21-rmk1.gz # linux kernel patch for arm ftp://ftp.arm.linux.org.uk/pub/linux/arm/kernel/v2.4/patch-2.4.21-rmk1.gz

我们在 bash 下工作，先设定一些变量：

```
$ export VBINUTILS=2.14
$ export VGCC=2.95.3
$ export VGLIBC=2.2.4
$ export VLINUX=2.4.21
$ export VLINUX_PATCH=rmk1
$
$ export PREFIX=/armtools
$ export TARGET=arm-linux
```

你可以把它们加到 .bashrc 中。如果你这么做了，你需要 logout 再 login 才能生效。否则在 bash 的命令行上输入它们并立即生效，但你 logout 再 login 时它就无效了。

我们的工作路径是：

```
...../ ----- ~ -- tars -------- SourceDir
.............|.................|---- BuildDir
.............|--- armtools
```

```
$ cd ~
$ mkdir -p tars/SourceDir
$ mkdir tars/BuildDir
$ mkdir arm_tools
$ su
# mv arm_tools $PREFIX
# exit
$
```

tars ---------- 在这里放我们的下载来的 .tar.gz 文件

SourceDir ------ 这个临时目录放我们解压缩后的源文件

BuildDir --------- 我们在这里编译

armtools -------- 把 arm-linux 交叉编译环境的安装在这里

1.1 安装 linux 的头文件

当你为不同类型的 ARM 编译 gcc，或编译不同版本的 kernel，或交叉编译 gcc 时都需要一套不同的 linux 头文件。

(1) 解压缩，打补丁

```
$ cd ~/tars/SourceDir
$ tar -zxf ../linux-$VLINUX.tar.gz
$ cd linux
$ zcat ../../patch- $VLINUX-$VLINUX_PATCH.gz | patch -p1
```

(2) 清理一下

```
$ make mrproper
```

(3) 修改 Makefile

将 Makefile 中 ARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -e s/arm.*/arm/ -e s/sa110/arm/) 这一行注释掉，并加一行 ARCH=arm。修改后象这样：

```
ARCH=arm
# ARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -e s/arm.*/arm/ -e s/sa110/arm/)
```

如果你的系统里的 sed 程序支持用 '\' 续行（通常都支持），你可以用这个 script 去修改 Makefile

```
#!/bin/sh
mv Makefile Makefile.orig
sed 's/ARCH := $(shell uname -m/ARCH=arm\
# ARCH := $(shell uname -m/' < Makefile.orig > Makefile
#end of script
```

注意：这个 script 里的 # ARCH := 是上一行的续行，不是 shell 里的注释，它也是要输入的。 如果你从浏览器(IE, netscape, etc)上 copy-paste 这个 script 到你的 bash console，它很有可能不工作； 但你在 bash console 里手工输入就可以工作。

因为有时 copy 过来后，是 'ARCH=arm\r\n'，而能工作的是 'ARCH=arm\n'。

(4) 建立连接

(a) 如果是 LART 板子

```
$ make lart_config
$ yes "" | make oldconfig
$ make include/linux/version.h
```

或:

```
$ make lart_config
$ make menuconfig 选择 <Exit> <Yes>
```

也有这样的用法：

```
$ make symlinks include/linux/version.h
```

那是不完全的。make symlinks 的作用相当于：

```
$ cd include/asm-arm
$ rm -f arch proc
$ ln -s arch-sa1100 arch
$ ln -s proc-armv proc
$ cd ../../
```

它并没有产生一个很重要的文件 include/linux/autoconf.h。 而 yes "" | make oldconfig 不仅是 make symlinks， 还产生了 include/linux/autoconf.h 。但它也没有产生 include/linux/version.h。

(b) 如果是 clps711x 的 CPU
连接应该为：

```
$ cd include/asm-arm
$ rm -f arch proc
$ ln -s arch-clps711x arch
$ ln -s proc-armv proc
$ cd ../../
```

(c) 如何是 s3c2410 的 CPU

```
$ make s3c2410_defconfig
$ make menuconfig 选择 <Exit> <Yes>
```

为你自己的系统定制:

```
$ make menuconfig
```

在这里你只需要选你使用的 CPU 或选则有你使用的 CPU 的板子即可，
当然进行更详细的配置更好。

注：

include/asm-arm/proc-armo 是 26 位 ARM
include/asm-arm/proc-armv 是 32 位 ARM

注：背景知识

在 ARM1 中实现 26 位地址空间，但没有被商用。
在 ARM2,2a 中还有 26 位地址空间，现在已经废弃。
在 ARM3 中扩展到 32 位地址空间，但是还反向兼容 26 位。
在 ARM4 中是 32 位地址空间，停止兼容 26 位地址空间。在 T 系列中加入 Thumb 指令。
在 ARM5 中是 32 位地址空间，在所有系列中均支持 16 位的 Thumb 指令。

(5) 拷贝头文件

```
$ mkdir -p $PREFIX/$TARGET/include
$ cp -dR include/linux $PREFIX/$TARGET/include
$ cp -dR include/asm-arm $PREFIX/$TARGET/include/asm
```

(6) 为 gcc 建立一个 linux kernel 头文件的连接。编译 gcc 时，它需要 linux kernel 的头文件，你可以用 --with-headers=$PREFIX/$TARGET/include 来指定头文件的位置，gcc 把它拷贝到 $PREFIX/$TARGET/sys-include。我们可以建立个 sys-include 连接，就不用 --with-headers 了。

```
$ cd $PREFIX/$TARGET
$ ln -s include sys-include
```

1.2 编译安装 binutils

这里用不到前面准备的 linux 头文件

(1) 解压缩

```
$ cd ~/tars/SourceDir
$ tar -zxf ../binutils-$VBINUTILS.tar.gz
```

(2) 编译

```
$ cd ~/tars/BuildDir
$ mkdir binutils
$ cd binutils
$ ../../SourceDir/binutils-$VBINUTILS/configure \
--target=$TARGET \
--prefix=$PREFIX
$ make all install
```

(3) 输出 binutils 的路径到环境变量中

你可以把它加到 .bashrc 中。如果你这么做了，你需要 logout 再 login 才能生效。否则在 bash 的命令行上输入它并立即生效，但你 logout 再 login 时它就无效了。

```
export PATH=$PREFIX/bin:$PATH
```

1.3 编译安装 gcc 的 c 编译器

(1) 解压缩

```
$ cd ~/tars/SourceDir
$ tar -zxf ../gcc-core-$VGCC.tar.gz
```

注意：为什么不用 all-in-one 的 gcc-$VGCC.tar.gz 呢?

all-in-one 的 gcc 包里面有 chill, fortran, java 等语言的编译器，虽然在下面 configure 时指定 -enable-languages=c，但编译时还是把所有的都编译一便，这不是我们需要的，而且它也总会有错误。因此我们只编译 C 语言的编译器。后面第二次编译的时候也是这个问题，我们只编译 C 和 C++ 的编译器。

(2) 修改 gcc 的 t-linux 文件

在 t-linux 文件中的 TARGET_LIBGCC2_CFLAGS 上加上 __gthr _ posix_h 和 inhibit_libc

```
$ cd gcc-$VGCC/gcc/config/arm
$ mv t-linux t-linux-orig
$ sed 's/TARGET_LIBGCC2_CFLAGS =/TARGET_LIBGCC2_CFLAGS =
-D__gthr_posix_h -Dinhibit_libc/' < t-linux-orig > t-linux-core
```

$ cp ./t-linux-core ./t-linux

(4) 编译

```
$ cd ~/tars/BuildDir
$ mkdir gcc-core
$ cd gcc-core
$ ../../SourceDir/gcc-$VGCC/configure \
--target=$TARGET \
--prefix=$PREFIX \
--enable-languages=c \
--disable-shared \
--disable-threads
$ make all install
```

1.4 编译安装 glibc

(1) 解压缩

```
$ cd ~/tars/SourceDir
$ tar -zxf ../glibc-$VGLIBC.tar.gz
$ cd glibc-$VGLIBC
$ tar -zxf ../../glibc-linuxthreads-$VGLIBC.tar.gz
```

(2) 编译

```
$ cd ~/tars/BuildDir
$ mkdir glibc
$ cd glibc
$ CC=$TARGET-gcc AR=$TARGET-ar RANLIB=$TARGET-ranlib \
../../SourceDir/glibc-$VGLIBC/configure \
$TARGET \
--prefix=$PREFIX/$TARGET \
--enable-add-ons
$ make all install
```

1.5.编译安装 gcc 的 c, c++ 编译器

(1) 恢复 t-linux 文件

```
$ cd ~/tars/SourceDir/gcc-$VGCC/gcc/config/arm/
$ cp t-linux-orig t-linux
```

(2) 解压缩 c++ 编译器

```
$ cd ~/tars/SourceDir/
$ tar -zxf ../gcc-g++-$VGCC.tar.gz
$
$ cd ~/tars/BuildDir
$ mkdir gcc
$ cd gcc
```

(3) 编译

```
$ ../../SourceDir/gcc-$VGCC/configure \
--target=$TARGET \
--prefix=$PREFIX \
--enable-languages=c,c++ \
--with-included-gettext
$ make all
$ make install
```

注：如果你下载的是 filename.tar.bz2，你可以用如下命令之一解压缩，第三种方式在任何系统中都好使。

```
$ tar -jxf filename.tar.bz2
$ tar -Ixf filename.tar.bz2
$ bzip2 -dc filename.tar.bz2 | tar xf -
```

如果你是第一次制作 arm-linux 交叉编译环境，强烈建议你用本文所使用的各个程序的版本。如果用其它版本，按照本文的方法可能会在编译的时候出问题，因为我没有时间去测试各个版本的组合。这里是源程序: crossarm.sh，它使用的是：

```
linux-2.4.21.tar.bz2
patch-2.4.21.bz2
binutils-2.14.tar.gz
gcc-core-2.95.3.tar.gz
gcc-g++-2.95.3.tar.gz
glibc-2.2.4.tar.gz
```

生成的 toolchain 大于 150 兆，用如下方法压缩：

```
$ cd ~
$ tar -cf armtools.tar /armtools
$ bzip2 -z armtools.tar
```

压缩后生成的 armtools.tar.bz2 大概有 30 几兆。

2．内核编译方法

(1) 下载新版本内核，解压(tar xzvf …)到相应的目录；
(2) 使用 make mrproper 清理一下内核；
(3) 配置内核：使用 make menuconfig 配置内核，对于移植内核来说，通常是尽量裁减内核；
(4) 生成新内核：make dep；make clean；make zImage。

三．实验内容和步骤

1．建立交叉编译环境

根据原理部分介绍的 arm-linux-gcc 交叉编译环境的建立方法，建立 gcc-3.2.0 以上版本的交叉编译器（编译 Linux2.6 内核需要 gcc-3.2 以上版本）。建议使用以下版本的源代码建立 gcc-3.4.0 版本的 arm-linux-gcc：

binutils-2.15.tar.gz
gcc-core-3.4.0.tar.gz
gcc-g++-3.4.0.tar.gz
glibc-2.3.tar.gz
glibc-linuxthreads-2.3.tar.gz
linux-2.6.7.tar.gz

2．配置 Linux2.6.7 内核

(1) 首先修改 Makefile，将 Makefile 中 SUBARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -e s/arm.*/arm/ -e s/sa110/arm/) 这一行注释掉，并加一行 SUBARCH=arm。修改后象这样：

SUBARCH=arm

SUBARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -e s/arm.*/arm/ -e s/sa110/arm/)

或者直接将 ARCH ?= $(SUBARCH)这行注释掉，并加一行 ARCH ?=arm 。

上面两种修改方法都是可以的，不管怎么做目的就是要是编译出来的目标文件（可执行文件）面向的是 arm 平台。

还要注释掉 CROSS_COMPILE ?= 这行，加一行 CROSS_COMPILE =../armbuild/tars/SourceDir/toolchain/bin/arm-linux-。修改这一行是为了在编译的过程中使用我们已经编译好的交叉编译器，路进指向你的交叉编译器的位置即可。

(2) 使用 make menuconfig（也可以使用 make config,make xconfig）来配置 Linux2.6.7 内核。可以参考下面的 config 文件。因为我们是为了移植而配置内核，应该尽量的裁减内核到最小程度（但必须要包含串口的驱动，我们需要串口获得内核引导时的输出信息）。

在配置内核的过程中需要注意以下几点：

(a) 尽量裁减内核，可以舍弃的部分都先去掉；
(b) 在 SYSTEM TYPE 目录中选择正确的 CPU 型号，我们这里应该选择 Samsung S3C2410，在 S3C2410 Implementations 选项中选择 SMDK2410/A9M2410。
(c) 在 General setup 目录下 Default kernel command string 选项中填入正确的 command string，在这里设置正确的串口名和波特率。
(d) 在 Character devices 目录下的 Serial drivers 选项中一定要选择 Samsung S3C2410 Serial port support 选项。保证内核对串口的支持。

```
#
# Automatically generated make config: don't edit
#
CONFIG_ARM=y
CONFIG_MMU=y
CONFIG_UID16=y
CONFIG_RWSEM_GENERIC_SPINLOCK=y

#
# Code maturity level options
#
```

```
# CONFIG_EXPERIMENTAL is not set
CONFIG_CLEAN_COMPILE=y
CONFIG_STANDALONE=y
CONFIG_BROKEN_ON_SMP=y

#
# General setup
#
CONFIG_SWAP=y
# CONFIG_SYSVIPC is not set
# CONFIG_BSD_PROCESS_ACCT is not set
# CONFIG_SYSCTL is not set
# CONFIG_AUDIT is not set
CONFIG_LOG_BUF_SHIFT=14
# CONFIG_HOTPLUG is not set
# CONFIG_IKCONFIG is not set
# CONFIG_EMBEDDED is not set
CONFIG_KALLSYMS=y
CONFIG_FUTEX=y
CONFIG_EPOLL=y
CONFIG_IOSCHED_NOOP=y
CONFIG_IOSCHED_AS=y
CONFIG_IOSCHED_DEADLINE=y
CONFIG_IOSCHED_CFQ=y
CONFIG_CC_OPTIMIZE_FOR_SIZE=y

#
# Loadable module support
#
# CONFIG_MODULES is not set

#
# System Type
#
# CONFIG_ARCH_ADIFCC is not set
# CONFIG_ARCH_CLPS7500 is not set
# CONFIG_ARCH_CLPS711X is not set
# CONFIG_ARCH_CO285 is not set
# CONFIG_ARCH_EBSA110 is not set
# CONFIG_ARCH_CAMELOT is not set
# CONFIG_ARCH_FOOTBRIDGE is not set
# CONFIG_ARCH_INTEGRATOR is not set
# CONFIG_ARCH_IOP3XX is not set
# CONFIG_ARCH_IXP4XX is not set
```

```
# CONFIG_ARCH_L7200 is not set
# CONFIG_ARCH_PXA is not set
# CONFIG_ARCH_RPC is not set
# CONFIG_ARCH_SA1100 is not set
CONFIG_ARCH_S3C2410=y
# CONFIG_ARCH_SHARK is not set
# CONFIG_ARCH_LH7A40X is not set
# CONFIG_ARCH_OMAP is not set
# CONFIG_ARCH_VERSATILE_PB is not set

#
# S3C2410 Implementations
#
# CONFIG_ARCH_BAST is not set
# CONFIG_ARCH_H1940 is not set
CONFIG_ARCH_SMDK2410=y
# CONFIG_MACH_VR1000 is not set

#
# Processor Type
#
CONFIG_CPU_32=y
CONFIG_CPU_ARM920T=y
CONFIG_CPU_32v4=y
CONFIG_CPU_ABRT_EV4T=y
CONFIG_CPU_CACHE_V4WT=y
CONFIG_CPU_COPY_V4WB=y
CONFIG_CPU_TLB_V4WBI=y

#
# Processor Features
#
CONFIG_ARM_THUMB=y
# CONFIG_CPU_ICACHE_DISABLE is not set
# CONFIG_CPU_DCACHE_DISABLE is not set
# CONFIG_CPU_DCACHE_WRITETHROUGH is not set

#
# General setup
#
# CONFIG_ZBOOT_ROM is not set
CONFIG_ZBOOT_ROM_TEXT=0x0
CONFIG_ZBOOT_ROM_BSS=0x0
```

```
#
# At least one math emulation must be selected
#
# CONFIG_FPE_NWFPE is not set
CONFIG_BINFMT_ELF=y
CONFIG_BINFMT_AOUT=y
# CONFIG_BINFMT_MISC is not set

#
# Generic Driver Options
#
# CONFIG_PM is not set
# CONFIG_ARTHUR is not set
CONFIG_CMDLINE="root=/dev/mtdblock/3 console=ttyS0,115200 init=/sbin/init"
CONFIG_ALIGNMENT_TRAP=y

#
# Parallel port support
#
# CONFIG_PARPORT is not set

#
# Memory Technology Devices (MTD)
#
CONFIG_MTD=y
# CONFIG_MTD_DEBUG is not set
CONFIG_MTD_PARTITIONS=y
CONFIG_MTD_CONCAT=y
# CONFIG_MTD_REDBOOT_PARTS is not set
CONFIG_MTD_CMDLINE_PARTS=y
# CONFIG_MTD_AFS_PARTS is not set

#
# User Modules And Translation Layers
#
# CONFIG_MTD_CHAR is not set
# CONFIG_MTD_BLOCK is not set
# CONFIG_MTD_BLOCK_RO is not set
# CONFIG_FTL is not set
# CONFIG_NFTL is not set
CONFIG_INFTL=y

#
# RAM/ROM/Flash chip drivers
```

```
#
CONFIG_MTD_CFI=y
# CONFIG_MTD_JEDECPROBE is not set
CONFIG_MTD_GEN_PROBE=y
# CONFIG_MTD_CFI_ADV_OPTIONS is not set
CONFIG_MTD_CFI_INTELEXT=y
# CONFIG_MTD_CFI_AMDSTD is not set
# CONFIG_MTD_CFI_STAA is not set
CONFIG_MTD_RAM=y
CONFIG_MTD_ROM=y
# CONFIG_MTD_ABSENT is not set
# CONFIG_MTD_OBSOLETE_CHIPS is not set

#
# Mapping drivers for chip access
#
# CONFIG_MTD_COMPLEX_MAPPINGS is not set
CONFIG_MTD_PHYSMAP=y
CONFIG_MTD_PHYSMAP_START=0x800000
CONFIG_MTD_PHYSMAP_LEN=0x800000
CONFIG_MTD_PHYSMAP_BUSWIDTH=2
# CONFIG_MTD_ARM_INTEGRATOR is not set
# CONFIG_MTD_EDB7312 is not set

#
# Self-contained MTD device drivers
#
# CONFIG_MTD_SLRAM is not set
# CONFIG_MTD_MTDRAM is not set
# CONFIG_MTD_BLKMTD is not set

#
# Disk-On-Chip Device Drivers
#
# CONFIG_MTD_DOC2000 is not set
# CONFIG_MTD_DOC2001 is not set
# CONFIG_MTD_DOC2001PLUS is not set

#
# NAND Flash Device Drivers
#
# CONFIG_MTD_NAND is not set

#
```

```
# Plug and Play support
#

#
# Block devices
#
# CONFIG_BLK_DEV_FD is not set
# CONFIG_BLK_DEV_LOOP is not set
CONFIG_BLK_DEV_RAM=y
CONFIG_BLK_DEV_RAM_SIZE=4096
CONFIG_BLK_DEV_INITRD=y

#
# Multi-device support (RAID and LVM)
#
# CONFIG_MD is not set

#
# Networking support
#
# CONFIG_NET is not set
# CONFIG_NETPOLL is not set
# CONFIG_NET_POLL_CONTROLLER is not set

#
# ATA/ATAPI/MFM/RLL support
#
# CONFIG_IDE is not set

#
# SCSI device support
#
# CONFIG_SCSI is not set

#
# Fusion MPT device support
#

#
# IEEE 1394 (FireWire) support
#
# CONFIG_IEEE1394 is not set

#
```

```
# I2O device support
#

#
# ISDN subsystem
#

#
# Input device support
#
CONFIG_INPUT=y

#
# Userland interfaces
#
CONFIG_INPUT_MOUSEDEV=y
# CONFIG_INPUT_MOUSEDEV_PSAUX is not set
CONFIG_INPUT_MOUSEDEV_SCREEN_X=1024
CONFIG_INPUT_MOUSEDEV_SCREEN_Y=768
# CONFIG_INPUT_JOYDEV is not set
# CONFIG_INPUT_TSDEV is not set
# CONFIG_INPUT_EVDEV is not set
# CONFIG_INPUT_EVBUG is not set

#
# Input I/O drivers
#
# CONFIG_GAMEPORT is not set
CONFIG_SOUND_GAMEPORT=y
CONFIG_SERIO=y
# CONFIG_SERIO_I8042 is not set
# CONFIG_SERIO_SERPORT is not set
# CONFIG_SERIO_CT82C710 is not set

#
# Input Device Drivers
#
# CONFIG_INPUT_KEYBOARD is not set
# CONFIG_INPUT_MOUSE is not set
# CONFIG_INPUT_JOYSTICK is not set
# CONFIG_INPUT_TOUCHSCREEN is not set
# CONFIG_INPUT_MISC is not set

#
```

```
# Character devices
#
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
CONFIG_HW_CONSOLE=y
# CONFIG_SERIAL_NONSTANDARD is not set

#
# Serial drivers
#
# CONFIG_SERIAL_8250 is not set

#
# Non-8250 serial port support
#
CONFIG_SERIAL_S3C2410=y
CONFIG_SERIAL_S3C2410_CONSOLE=y
CONFIG_SERIAL_CORE=y
CONFIG_SERIAL_CORE_CONSOLE=y
CONFIG_UNIX98_PTYS=y
# CONFIG_LEGACY_PTYS is not set
# CONFIG_QIC02_TAPE is not set

#
# IPMI
#
# CONFIG_IPMI_HANDLER is not set

#
# Watchdog Cards
#
# CONFIG_WATCHDOG is not set
# CONFIG_NVRAM is not set
# CONFIG_RTC is not set
# CONFIG_GEN_RTC is not set
# CONFIG_DTLK is not set
# CONFIG_R3964 is not set
# CONFIG_APPLICOM is not set

#
# Ftape, the floppy tape device driver
#
# CONFIG_FTAPE is not set
# CONFIG_AGP is not set
```

```
# CONFIG_DRM is not set
# CONFIG_RAW_DRIVER is not set

#
# I2C support
#
# CONFIG_I2C is not set

#
# Multimedia devices
#
# CONFIG_VIDEO_DEV is not set

#
# Digital Video Broadcasting Devices
#

#
# File systems
#
CONFIG_EXT2_FS=y
# CONFIG_EXT2_FS_XATTR is not set
CONFIG_EXT3_FS=y
CONFIG_EXT3_FS_XATTR=y
# CONFIG_EXT3_FS_POSIX_ACL is not set
# CONFIG_EXT3_FS_SECURITY is not set
CONFIG_JBD=y
# CONFIG_JBD_DEBUG is not set
CONFIG_FS_MBCACHE=y
# CONFIG_REISERFS_FS is not set
# CONFIG_JFS_FS is not set
# CONFIG_XFS_FS is not set
# CONFIG_MINIX_FS is not set
CONFIG_ROMFS_FS=y
# CONFIG_QUOTA is not set
CONFIG_AUTOFS_FS=y
CONFIG_AUTOFS4_FS=y

#
# CD-ROM/DVD Filesystems
#
# CONFIG_ISO9660_FS is not set
# CONFIG_UDF_FS is not set
```

```
#
# DOS/FAT/NT Filesystems
#
# CONFIG_FAT_FS is not set
# CONFIG_NTFS_FS is not set

#
# Pseudo filesystems
#
# CONFIG_PROC_FS is not set
CONFIG_SYSFS=y
# CONFIG_DEVPTS_FS_XATTR is not set
# CONFIG_TMPFS is not set
# CONFIG_HUGETLB_PAGE is not set
CONFIG_RAMFS=y

#
# Miscellaneous filesystems
#
# CONFIG_HFSPLUS_FS is not set
# CONFIG_JFFS_FS is not set
# CONFIG_JFFS2_FS is not set
# CONFIG_CRAMFS is not set
# CONFIG_VXFS_FS is not set
# CONFIG_HPFS_FS is not set
# CONFIG_QNX4FS_FS is not set
# CONFIG_SYSV_FS is not set
# CONFIG_UFS_FS is not set

#
# Partition Types
#
# CONFIG_PARTITION_ADVANCED is not set

#
# Native Language Support
#
# CONFIG_NLS is not set

#
# Graphics support
#
# CONFIG_FB is not set
```

```
#
# Console display driver support
#
# CONFIG_VGA_CONSOLE is not set
# CONFIG_MDA_CONSOLE is not set
CONFIG_DUMMY_CONSOLE=y

#
# Sound
#
# CONFIG_SOUND is not set

#
# Misc devices
#

#
# USB support
#

#
# USB Gadget Support
#
# CONFIG_USB_GADGET is not set

#
# Kernel hacking
#
CONFIG_FRAME_POINTER=y
# CONFIG_DEBUG_USER is not set
# CONFIG_DEBUG_INFO is not set
# CONFIG_DEBUG_KERNEL is not set

#
# Security options
#
# CONFIG_SECURITY is not set

#
# Cryptographic options
#
# CONFIG_CRYPTO is not set
```

```
#
# Library routines
#
# CONFIG_CRC32 is not set
# CONFIG_LIBCRC32C is not set
```

3．编译内核

使用 make dep;make clean;make zImage 命令编译内核，得到内核压缩映象 zImage，把他拷贝到根目录的/tftpboot（我们的板子将从这个目录下载内核映像）目录下。

4．下载内核到开发板上

通过板子上的 bootloader(ppcboot)把内核映像 zImage 下载到 flash 中运行，查看运行结果，根据错误信息对内核代码进行相应的修改。

5．调试错误信息

起初代码下载到开发板上运行，ppcboot 显示完解压内核信息后就再也没有任何信息输出。也就是说内核运行后并没有信息输出到串口上。这种状态可能性比较大的原因是串口驱动有问题：修改/drivers/serial/s3c2410.c 的 38 行，改为：

#define SERIAL_S3C2410_NAME　　"ttyS"。

修改源码后重新编译内核，下载映像到开发板上，顺利看到串口控制台上得到内核的输出信息。

在嵌入式平台上移植 Linux 最重要的一步就是能够在串口上得到内核的打印信息。由于 Linux2.6.7 已经带有 S3C2410 平台的分支代码，所以我们的移植工作相对来说比较简单。

四．思考题

1．移植当前 Linux 内核最高版本到开发板上。

实验二十九： PPP/GPRS 拔号实验

一．实验目的

通过 GPRS 模块或 Modem 实现 PPP 拔号。

二. 实验原理和说明

PPP （点对点协议）是在串行线路上运行 IP （互联网协议）以及其它网络协议的一种机制，串行连结可以是直接的串行连接（使用 null-modem 电缆）或是使用调制解调器以及

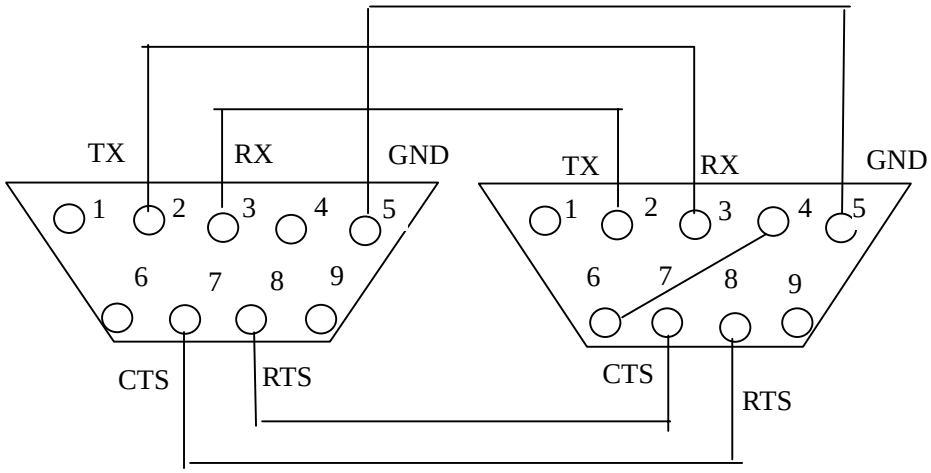
电话线路的连结 (当然也包括如 ISDN 的数字线路) ，使用 PPP, 你可以把你的开发板（PC 机）连接到一台 PPP 服务器上并存取该服务器所连接的网络资源，如同你是直接连接在该网络上一般．开发板作为一个独立的 Internet 主机，拥有自己的 IP 地址， 用户可以通过一定方式看到这个 IP 地址(ipconfig)， 比如 202.207.9.234， 这个地址一般是动态的,每次拨号上网都有可能不一样。

三．实验内容和步骤

1．串口线制作

因为板子的串口平时是和 PC 机相连，所以用同样的串口和 Modem 相连，则连接开发板和 Modem（GPRS 模块）的串口线需要 2，3；7，8 交换一下，用九针串口线将 RUIJIARM9-EDU 第二个串口(ttyS1)与 Modem 连接起来，线的两头都是针头。

串口线的制作方法见下图：



HHARM9-EDE 串口 2　　**串口线接 GPRS 模块（MODEM 端）**

PC 接 MODEM 的线就是标准串口线，即 2 对 2，3 对 3 顺接的线，内部不要交叉。

PC 机的串口定义：2（RX）、3（TX）、7（RTS）、8（CTS）

MODEM 的串口定义：2（TX）、3（RX）、7（CTS）、8（RTS）

RUIJIARM9-EDU 底板的串口接 MODEM 要 2、3 交换，7、8 交换。

2．可能由于 Modem 的要求，需要把 RUIJIARM9-EDU 第二个串口芯片的第十三脚 RTS（第 13 脚）与地相连，即低电平时 RTS（发送请求）信号才有效。

3．连接好开发板的硬件，等待和 GPRS 模块连接好后，再加电源。

由于可以通过让板子刚启动就拨号，无须用手工拨号，我们在开发板的/usr/etc/rc.local 文件中注释掉这样一句：

#/cramfs/sbin/pppd modem /dev/ttyS1 115200 crtscts connect 'chat -v -f /etc/config/chat.ttySx' debug noauth noipdefault defaultroute

如果打开此句，就会在开发板刚启动时，就会自己拨号。

4．如果怀疑是否已经运行了拨号脚本，可以在 minicom 中输入以下命令看看是否已经启动了相关脚本。显示的信息如下：

```
# ps
  PID   Uid      VmSize Stat Command
    1 root         504 S    init
    2 root             SW   [keventd]
    3 root             SWN  [ksoftirqd_CPU0]
    4 root             SW   [kswapd]
    5 root             SW   [bdflush]
    6 root             SW   [kupdated]
   11 root             SW   [mtdblockd]
   12 root             SW   [khubd]
   24 root         528 S    inetd
   43 root         480 S    syslogd
   45 root         744 S    -sh
  187 root         364 S    diald -f /etc/config/diald.ttySx
  261 root         492 S    sh -c /etc/config/connect
  262 root         508 S    /bin/sh /etc/config/connect
  263 root         196 S    chat -v -f /etc/config/chat.ttySx
  265 root         584 R    ps
```

5．可以使用如下命令查看拔号相关信息，默认输出文件为/var/log/messages,若中途有断线的情况发生，再此文件中也可以看见。

cat /var/log/messages　　显示信息如下：

```
Jan  1 00:02:31 (none) daemon.notice pppd[149]: pppd 2.4.1 started by (unknown)0Jan 1 00:02:32 (none) local2.info chat[151]: abort on (BUSY)
Jan  1 00:02:32 (none) local2.info chat[151]: abort on (NO CARRIER)
Jan  1 00:02:32 (none) local2.info chat[151]: send (ATZ\d^M)
Jan  1 00:02:33 (none) local2.info chat[151]: send (ATE1V1Q0^M)
Jan  1 00:02:34 (none) local2.info chat[151]: expect (OK)
Jan  1 00:02:34 (none) local2.info chat[151]: ATZ^M^M
Jan  1 00:02:34 (none) local2.info chat[151]: OK
Jan  1 00:02:34 (none) local2.info chat[151]:   -- got it
Jan  1 00:02:34 (none) local2.info chat[151]: send (ATDT16388^M)
Jan  1 00:02:34 (none) local2.info chat[151]: timeout set to 60 seconds
Jan  1 00:02:34 (none) local2.info chat[151]: expect (CONNECT)
Jan  1 00:02:34 (none) local2.info chat[151]: ^M
Jan  1 00:02:48 (none) local2.info chat[151]: ATDT16388^M^M
Jan  1 00:02:48 (none) local2.info chat[151]: CONNECT
Jan  1 00:02:48 (none) local2.info chat[151]:   -- got it
Jan  1 00:02:48 (none) daemon.info pppd[149]: Serial connection established.
Jan  1 00:02:48 (none) daemon.debug pppd[149]: using channel 1
Jan  1 00:02:48 (none) daemon.info pppd[149]: Using interface ppp0
Jan  1 00:02:48 (none) daemon.notice pppd[149]: Connect: ppp0 <--> /dev/ttyS1
Jan  1 00:02:50 (none) daemon.warn pppd[149]: Warning - secret file /etc/ppp/pasJan  1 00:02:50 (none) daemon.debug pppd[149]: sent [LCP ConfReq id=0x1 <asyncm]Jan  1
```

00:02:50 (none) daemon.debug pppd[149]: rcvd [LCP ConfReq id=0xb7 <mru 1]Jan 1
00:02:50 (none) daemon.debug pppd[149]: sent [LCP ConfRej id=0xb7 <mrru]Jan 1
00:02:50 (none) daemon.debug pppd[149]: rcvd [LCP ConfReq id=0xb8 <mru 1]Jan 1
00:02:50 (none) daemon.debug pppd[149]: sent [LCP ConfAck id=0xb8 <mru 1]Jan 1
00:02:50 (none) daemon.debug pppd[149]: rcvd [LCP ConfAck id=0x1 <asyncm]Jan 1
00:02:50 (none) daemon.warn pppd[149]: Warning - secret file /etc/ppp/pasJan 1
00:02:50 (none) daemon.debug pppd[149]: sent [PAP AuthReq id=0x1 user="1]Jan 1
00:02:50 (none) daemon.debug pppd[149]: rcvd [PAP AuthAck id=0x1 ""]
Jan 1 00:02:50 (none) daemon.debug pppd[149]: sent [IPCP ConfReq id=0x1 <addr]Jan
1 00:02:50 (none) daemon.err modprobe: modprobe: Can't open dependencies f)Jan 1
00:02:50 (none) daemon.err modprobe: modprobe: Can't open dependencies f)Jan 1
00:02:50 (none) daemon.debug pppd[149]: sent [CCP ConfReq id=0x1 <deflat]Jan 1
00:02:50 (none) daemon.debug pppd[149]: rcvd [IPCP ConfReq id=0x63 <addr]Jan 1
00:02:50 (none) daemon.debug pppd[149]: sent [IPCP ConfAck id=0x63 <addr]Jan 1
00:02:50 (none) daemon.debug pppd[149]: rcvd [IPCP ConfRej id=0x1 <compr]Jan 1
00:02:50 (none) daemon.debug pppd[149]: sent [IPCP ConfReq id=0x2 <addr]Jan 1
00:02:50 (none) daemon.debug pppd[149]: rcvd [IPCP ConfNak id=0x1 <addr]Jan 1
00:02:50 (none) daemon.debug pppd[149]: rcvd [LCP ProtRej id=0xb8 80 fd]Jan 1
00:02:50 (none) daemon.debug pppd[149]: rcvd [IPCP ConfNak id=0x2 <addr]Jan 1
00:02:50 (none) daemon.debug pppd[149]: sent [IPCP ConfReq id=0x3 <addr]Jan 1
00:02:51 (none) daemon.debug pppd[149]: rcvd [IPCP ConfAck id=0x3 <addr]Jan 1
00:02:51 (none) daemon.notice pppd[149]: local IP address 220.178.18.9
Jan 1 00:02:51 (none) daemon.notice pppd[149]: remote IP address 220.178.0.69
从显示信息的最后两句可以看出，自己动态生成的 IP 和远程的 IP 都已经获得。现在可以在开发上用 ifconfig 命令查看一下，可以发现在已经多了一个 ppp0 的网络设备。

6．当拨号成功时，可以使用命令 ping 来测试一下能否 ping 通一个远程的 IP 或域名。例如：
ping 202.38.64.1　　或 ping www.ruijitek.com　　显示信息大体上如下：

PING 202.38.64.1 (202.38.64.1): 56 data bytes
64 bytes from 202.38.64.1: icmp_seq=0 ttl=59 time=140.2 ms
64 bytes from 202.38.64.1: icmp_seq=1 ttl=59 time=150.1 ms
64 bytes from 202.38.64.1: icmp_seq=2 ttl=59 time=150.2 ms
如果能 ping 通一个指定的 IP 而 ping 一个域名是不行，则可能是 resolv.conf 中设置的域名服务器（DNS）有误或指定的域名服务器不存在，修改/etc/resolv.conf 文件（也即 ramdisk 解开以后的 mnt/etc/resolv.conf），此文件的内容一般上只有如下一行：
nameserver　　202.102.192.68

7．拔号脚本的配置的详细情况请见附件《PPP 拔号详解》，大部分拔号脚本放在板子启动以后的/etc/config 目录下。

四．思考题和作业

1．把已经连接上 internet 的开发板作为网关，实现局域网中其它计算上网，需要怎么做？

2．亲自动手，在没有带 GPRS 拨号脚本及相关拨号软件的 ramdisk 中加入 GPRS 拨号。
3．利用 Modem 来实现 PPP 拨号。

附录：PPP 拨号详解

1．简介

锐极公司 PPP 拨号软件包 FOR uClinux 提供在锐极 RUIJICO5272-R1 开发板上进行 PPP 拨号的软件技术。该软件包支持四种工作模式：

(1) 一开机就拨号，且永不挂断。

(2) 检测到有通信数据流量时才自动拨号，且可设定自动挂断时间。

(3) 一开机就拨号，且可以设定自动挂断时间。

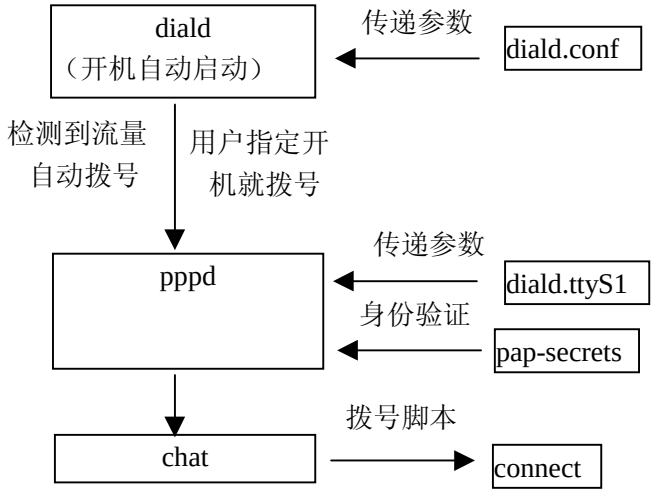
(4) 检测到有通信数据流量时才自动拨号，且永不挂断。

以上四种工作模式均通过修改一个脚本文件（diald.conf）的内容来控制实现。

2．拨号软件机制

PPP 拨号的软件实现机制简介如下：（详见附录的开发文档）

系统启动时自动启动 diald 后台进程，它启动时读取 diald.conf 中对它的配置，从而设置自己的工作模式。当用户指定开机就拨号或者检测到用户有访问外部网络的流量时，它就自动调用 pppd 进行拨号，启动 pppd 时，它还要通过脚本 diald.ttyS1 给 pppd 传递参数从而配置 pppd，并指示 pppd 调用 chat 和 connect 脚本进行对 MODEM 进行初始化并进行拨号。



3．软件配置及脚本修改

拨号软件的配置完全通过脚本修改来实现。

4．设定四种工作模式只需要修改 diald.conf

首先简单描述一下 diald.conf 脚本的大致功能：

该文件其实是个规则列表，规则主要有两个：ignore/accept。其中 ignore 表示这种协议的数据包不会引发拨号连接，而 accept 则反之，即这种协议的数据包将引发拨号连接。例如，设定 ping 操作时不进行拨号，则加入如下一句：

ignore icmp any

而规定对外的 HTTP 访问就会引发 PPP 拨号，就加入如下一句：

accept tcp 120 tcp.dest=tcp.www

其中 120 表示 120 秒的空闲等待时限。这里的时延都以秒为单位。

整个 diald.conf 就是一个由多条 accept 规则组成的大规则列表，若线路上在任何一条规则指定的时延内都没有出现其对应指定的协议数据包，则 diald 认为整个 PPP 线路处于空闲状态，且已经超出了用户设定的空闲时限，这时将执行 disconnect 脚本，从而自动挂断 PPP

线路。

在锐极提供的 PPP 软件包中我们已经给出了四个标准的配置脚本，分别对应四种工作模式。用户烧制板子后，通过 minicom 在板子上的目录/etc/config 下可以看到这四个文件，它们是：

startkeep －> 一开机就拨号，且永不挂断。

trafficidle －> 检测到有通信数据流量时才自动拨号，且可设定空闲自动挂断时间。

startidle －> 一开机就拨号，且可以设定空闲自动挂断时间。

traffickeep －> 检测到有通信数据流量时才自动拨号，且永不挂断。

出厂这四个配置脚本中除第四个外，其线路空闲时限均设置为 30 秒，用户若要修改这个时限，就必须先在宿主 LINUX PC 机上用 vi 修改脚本。

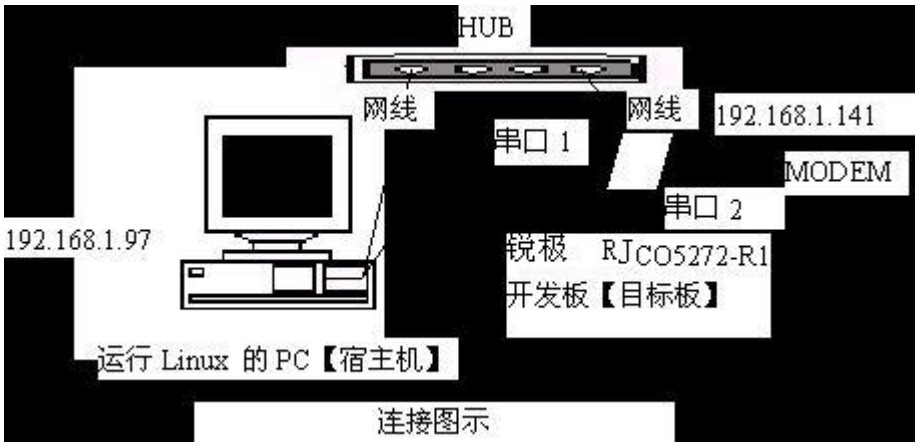
连接图示如下：更改空闲时间为 10 分钟即 600 秒。

在 PC 机上：

```
cp uClinux/romfs/etc/default/diald.conf /
vim diald.conf
```

在 vi 中执行字符串替换操作：

```
:%s/30/600/g                    //字符串替换操作
:wq
```



连接图示

下面分别介绍一下这四个配置脚本：

(1) startkeep

```
up                    【就一句话，该选项告知 diald 从启动就拨号并保持连接不断】
```

(2) trafficidle

```
ignore tcp tcp.dest=tcp.53
ignore tcp tcp.source=tcp.53
accept tcp 30 ip.tot_len=40,tcp.syn
ignore tcp ip.tot_len=40,tcp.live
accept tcp 30 tcp.dest=tcp.80
accept tcp 30 tcp.source=tcp.80
keepup tcp 30 !tcp.live
ignore tcp !tcp.live
accept tcp 30 tcp.dest=tcp.21
accept tcp 30 tcp.source=tcp.21
accept tcp 30 any
```

ignore udp udp.dest=udp.513
ignore udp udp.source=udp.513
ignore udp udp.dest=udp.520
ignore udp udp.source=udp.520
ignore udp udp.dest=udp.123
ignore udp udp.source=udp.123
ignore udp udp.dest=udp.525
ignore udp udp.source=udp.525
ignore udp udp.dest=udp.53,udp.source=udp.53
accept udp 30 udp.dest=udp.53
accept udp 30 udp.source=udp.53
ignore udp udp.source=udp.137,udp.dest=udp.137
accept any 30 any

(3) startidle

uprule 【该选项告知 diald 启动就拨号而且后续的规则有效，即空闲 30 秒自动挂断】
ignore tcp tcp.dest=tcp.53
ignore tcp tcp.source=tcp.53
accept tcp 30 ip.tot_len=40,tcp.syn
ignore tcp ip.tot_len=40,tcp.live
accept tcp 30 tcp.dest=tcp.80
accept tcp 30 tcp.source=tcp.80
keepup tcp 30 !tcp.live
ignore tcp !tcp.live
accept tcp 30 tcp.dest=tcp.21
accept tcp 30 tcp.source=tcp.21
accept tcp 30 any
ignore udp udp.dest=udp.513
ignore udp udp.source=udp.513
ignore udp udp.dest=udp.520
ignore udp udp.source=udp.520
ignore udp udp.dest=udp.123
ignore udp udp.source=udp.123
ignore udp udp.dest=udp.525
ignore udp udp.source=udp.525
ignore udp udp.dest=udp.53,udp.source=udp.53
accept udp 30 udp.dest=udp.53
accept udp 30 udp.source=udp.53
ignore udp udp.source=udp.137,udp.dest=udp.137
accept any 30 any

(4) traffickeep

ignore tcp tcp.dest=tcp.53
ignore tcp tcp.source=tcp.53

accept tcp 99999999 ip.tot_len=40,tcp.syn
ignore tcp ip.tot_len=40,tcp.live
accept tcp 99999999 tcp.dest=tcp.80
accept tcp 99999999 tcp.source=tcp.80
keepup tcp 99999999 !tcp.live
ignore tcp !tcp.live
accept tcp 99999999 tcp.dest=tcp.21
accept tcp 99999999 tcp.source=tcp.21
accept tcp 99999999 any
ignore udp udp.dest=udp.513
ignore udp udp.source=udp.513
ignore udp udp.dest=udp.520
ignore udp udp.source=udp.520
ignore udp udp.dest=udp.123
ignore udp udp.source=udp.123
ignore udp udp.dest=udp.525
ignore udp udp.source=udp.525
ignore udp udp.dest=udp.53,udp.source=udp.53
accept udp 99999999 udp.dest=udp.53
accept udp 99999999 udp.source=udp.53
ignore udp udp.source=udp.137,udp.dest=udp.137
accept any 99999999 any
keepon 【该选项告知 diald 前面的规则有效，但一旦连接就保持连接不断。】

5．更改拨号 ISP

要涉及 chat.ttyS1 脚本、pap-secrets 文件和 ttys1 脚本文件。

出厂默认拨的 ISP 为本地的 169（用户名 169，密码 169）。若要拨其它 ISP（例如：163，用户名 rjcn，密码 123456），则要改动：

(1) 修改 chat.ttyS1 文件中 ATDT169 为 ATDT163。
(2) 修改 pap-secrets 文件中的用户名和密码：
“rjcn” dialout “123456” *
dialout “rjcn” “123456” *
(3) 修改 ttys1 文件，将其中的 169 改为 rjcn。

6．更改拨号用端口

板子默认的拨号用端口为串口 2，即 ttyS1，若要更改端口，例如改为 ttyS0，则：

(1) 对应脚本文件更名：将 diald.ttyS1 更名为：diald.ttyS0，chat.ttyS1 改名为 chat.ttyS0,更改 ttys1 为 ttys0。
(2) 修改 diald.ttyS1 中 pppd-options file /etc/config/ttys1 的 ttys1 为 ttys0。

7．软件说明

软件组合：/bin/diald（0.16.5），/bin/pppd（2.3.8），/bin/chat（）

通过 diald 调用 pppd 进行拨号。pppd 使用 chat 脚本与串口通讯。

配置文件：（下面为板上路径，在 PC 机上为：romfs/etc/default/目录下）

/etc/config/diald.conf
/etc/config/diald.ttyS1
/etc/config/connect
/etc/config/chat.ttyS1
/etc/config/disconnect
/etc/config/options（虽然为空，但不可少）

启动方式：

/bin/diald –f　/etc/config/diald.ttyS1 （可写在文件 etc/config/inittab 中）

涉及的修改参数：

使用的串口设备：device /dev/ttySX （文件/etc/config/diald.ttyS1）
重拨时间：　redial-timeout　　（同上）
拨号号码：　chat "" ATDTXXX　　（文件/etc/config/connect）

密码文件：这两个文件名是系统默认指定的，编译到内核中的。ISP 指定用哪种验证方式，pppd 就会自动去读取对应的密码文件。

chap-secrets
pap-secrets
这两个文件由 UI 配置自动更改。

chap-secrets 文件【CHAP 验证方式】

```
"169" dialout "169" *
dialout "169" "169" *
```

pap-secrets 文件【PAP 验证方式】

```
"169" dialout "169" *
dialout "169" "169" *
```

diald.ttyS1 文件

```
modem
mode ppp
-pidstring
speed 38400
device /dev/ttyS1
crtscts
defaultroute
connect /etc/config/connect
disconnect /etc/config/disconnect
pppd-options file /etc/config/ttys1
redial-timeout 1
dynamic
```

local 0.0.0.0
remote 0.0.0.0
netmask 0.0.0.0

该文件中的参数都是将要由 diald 传给 pppd 使用的。

diald.conf 文件

```
ignore tcp tcp.dest=tcp.53
ignore tcp tcp.source=tcp.53
accept tcp 900 ip.tot_len=40,tcp.syn
ignore tcp ip.tot_len=40,tcp.live
accept tcp 900 tcp.dest=tcp.80
accept tcp 900 tcp.source=tcp.80
keepup tcp 900 !tcp.live
ignore tcp !tcp.live
accept tcp 900 tcp.dest=tcp.21
accept tcp 900 tcp.source=tcp.21
accept tcp 900 any
ignore udp udp.dest=udp.513
ignore udp udp.source=udp.513
ignore udp udp.dest=udp.520
ignore udp udp.source=udp.520
ignore udp udp.dest=udp.123
ignore udp udp.source=udp.123
ignore udp udp.dest=udp.525
ignore udp udp.source=udp.525
ignore udp udp.dest=udp.53,udp.source=udp.53
accept udp 900 udp.dest=udp.53
accept udp 900 udp.source=udp.53
ignore udp udp.source=udp.137,udp.dest=udp.137
accept any 180 any
```

connect 文件【拨号脚本，通过 UI 更改拨号属性后，就会改写该脚本文件】

```
#!/bin/sh                          这一句不可少，它通知 pppd 要用 shell 执行
chat -v    -f /etc/config/chat.ttyS1
```

chat.ttyS1 文件：

```
ABORT BUSY
ABORT 'NO CARRIER'
''
ATZ\d                      等待 MODEM 返回”后发出 ATZ 初始化 MODEM
''
ATE1V1Q0                   等待 MODEM 返回”后发出 ATE1V1Q0 命令
OK
ATDT169                        等待 MODEM 返回”后发出 ATDT169 命令，拨号 169
TIMEOUT 60
```

CONNECT

使用 chat 命令通过串口与 modem 进行通讯。

下面给出一个老版本的 connect 脚本，它也能拨，但不够通用。

chat "" ATZ0&K3

chat OK ""

chat "" ATDT169

chat CONNECT ""

chat : 169

chat word: 169

chat "" ppp　　【这句没有也行】

即原来的 connect 脚本中要进行用户名和密码验证，现在验证都不在 connect 中进行了，而是改到 pap-secrets 和 chap-secrets 中进行了，是由 pppd 读取这两个文件进行验证。

8．pppd 命令格式及主要参数说明

命令格式：　　pppd [tty_name] [speed] [options]

[tty_name]：用于实现通讯的设备命，例如：/dev/ttyS1

[speed] ：用于设定串口的波特率，该值为一个十进制的数

asyncmap：设置异步字符映射。Map 为 32bit 的十六进制数，每一 bit 为“1”表示从 0 到 31 的数值中对应的字符不能在串行线路上接收，pppd 将将要求对方以 2 字节的 escape 序列来发送这些字符。“0”表示没有任何字符要求用 escape 序列。

Auth：在网络包发送和接收之前，要求对方验证自己，如果系统有缺省路由的话，则 auth 为缺省选项。如果 auth 和 noauth 选项都没有指定，则 pppd 只允许对方使用系统不具有路由的地址。

Call name：从/etc/ppp/peers/name 文件中读入选项。

Connect script：通过脚本程序中的命令来设置串口。特别时使用 chat 命令与 modem 对话和启动远地 ppp 会话。

Crtscts：使用硬件流控 RTS/CTS 来控制数据在串行口中地流动。

Cdtrcts：使用非标准的硬件流控 DTR/CTS 来控制数据在串口中的流动。

Defaultroute：当 ipcp 谈判完成后，在系统路由表中增加一个缺省路由，使用对方作为网关。当 ppp 断开后，该记录将被删除。Defaultroute 优先于 nodefaultroute。

Disconnect　script：在 ppp 断开线路后，运行脚本程序中指定的命令。在不能使用 modem 硬件流控信号时，可以使用指定的命令挂起 modem。

Escape xx，yy……：指定传送中需要 escape 的字符，xx，yy 为 16 进制数。

File name：从指定的文件中读入选项。

Lock：指定 pppd 使用 uucp 方式的 lock 文件来确保串行设备的独占方式访问。

Mru　n：设置最大接收单元为 n。缺省为 1500，最小为 128，慢的线路可以使用 296

Mtu　n：设置最大传送单元为 n。

<本地 ip>:<远地 ip>：设置本地和/或远地 ip 地址。任何一个地址都可以省略。地址可以是主机名或十进制数字。除非指定 noipdefault 选项，缺省的本地 ip 地址是系统的第一个 ip 地址。远地地址如果没有指定可以从对方获得

debug：允许对连接进行 debug，pppd 将通过 syslog 记录所有发送和接受的控制包。

Login：使用系统 password 数据库用户用于对方 pap 验证，并在系统 wtmp 文件中记录用户。对方必须在/etc/ppp/pap-secrets 文件以及系统 password 数据库中含有记录。

Modem：使用 modem 控制线。这是缺省设置。当 pppd 打开串行设备时要等待来自 Modem 的 cd 信号（除非指定了一个连接脚本）；当一个连接要中断和在执行连接脚本之前，pppd 将暂短降低 dtr 信号。

Name name：设置用于身份验证的本地系统名字为 name。这是一个特殊的选项。Pppd 在验证对方过程中将在密码文件里含有第二个区域是 name 的行中查找要使用的密码。注意 pppd 并不把 domain 设置的域名加在 name 后面。

Noauth：不需要对方验证自己。这是特殊的选项。

Nocrtscts：禁止在串口上使用硬件流控，如 RTS/CTS。

Nodefaultroute：禁止缺省路由选项。系统管理员可以将该选项放在/etc/ppp/options 文件中防止用户通过 pppd 建立缺省路由。

Proxyarp：在本地系统的 arp 表中增加对方的 ip 地址和本系统的以太网地址记录。使对方在其他系统看起来是在本地以太网上。

Remotename name：为了身份验证，设置远地系统假定的名字为 name。

9．常见问题及其解决办法

(1) 拨号时 MODEM 吱里哇啦叫一通，但在 minicom 下 ifconfig 只能看到 sl0，看不到 ppp0，说明根本没有拨上。MODEM 有两个灯常亮，即：CD 和 OH，但收发灯都不闪。

这是 UI 配置生成的拨号脚本/etc/config/connect 有问题。

(2) 拨号已经通了，ping DNS 也通了（从板子或 LAN 上 WINDOWS PC ping 都通了），但 WINDOWS PC 就是无法上网，输入 IP 是可以的，这样似乎是 DNS 的问题，但 DNS 明明是通的。

测试结果表明：是 UI 设置 dhcpd.conf 时有错误，在 DNS 一项后多加了一个;号，导致 WINDOWS PC 机获取 IP 时 DNS 有误。

调测过程记录：

用 PC 连接 MODEM，通过超级终端手工拨 01095700 时，使用 AT 命令，信息如下：

首先输入

+++ATH0

ATZ 【初始化 MODEM】

反馈：

OK

ATM1L1

反馈

OK

ATDT01095700

ISP 返回 CONNECT 38400

并继续要求输入：

Username:

Password:

然后返回更多连接信息，整个过程如下所示：

AATTHH00

OK

++++++AATTHH00

OK
AATTZZ
OK
AATTMM11LL11
OK
AATTDDTT116699
CONNECT 38400
Welcome to HeFei 169 Dial Access Services
Username: 116699
~}-}*e}0~~Entering PPP mode.}-}*Async interface address is unnumbered (FastEther
net0/0/0)}-}*Your IP address is 202.111.194.72. MTU is 1500 b 瑙~~ytes}-}*}-}*恁~
~ }#?}!T} }4}"}&} }*} } }%}&灥@$}'}"}(}"]2~~ }#?}!U} }4}"}&} }*} } }%}&灥@$}'}
"}(}"}4　~ }#?}!V} }4}"}&} }*} } }%}&灥@$}'}"}(}"迄<~~ }#?}!W} }4}"}&} }*} } }
%}&灥@$}'}"}(}"棌~~ }#?}!X} }4}"}&} }*} } }%}&灥@$}'}"}(}"W 諂~ }#?}!Y} }4}"}&}
}*} } }%}&灥@$}'}"}(}">F~~ }#?}!Z} }4}"}&} }*} } }%}&灥@$}'}"}(}"喳~~ }#?}![
} }4}"}&} }*} } }%}&灥@$}'}"}(}"漢~~ }#?}!\} }4}"}&} }*} } }%}&灥@$}'}"}(}"Q 垀~
}#?}!]} }4}"}&} }*} } }%}&灥@$}'}"}(}"}8};~~ }#?}!^} }4}"}&} }*} } }%}&灥@$}'
}"}()"姚~
NO CARRIER

****
*　　Welcome to ChinaNet　　*
**********　　Zizhuyuan
please input username:169
please input password:***
Entering PPP mode.
Async interface address is unnumbered(Ethernet0)
Header compression will match your system.
Your IP address is: 211.149.85.66 MTU is 1500 bytes
~ }#?}!坍 }9}!}$}%貗"}&} }*} } }'}"}(}"}1}$}%貗 3}#}!砐~~ }#?}!坍 }9}!}$}%貗"}&
} }*} } }'}"}(}"}1}$}%貗 3}#}!砐~~ }#?}!坍 }9}!}$}%貗"}&} }*} } }'}"}(}"}1}$}%貗
3}#}!砐~~ }#?}!坍 }9}!}$}%貗"}&} }*} } }'}"}()")1)$)%貗 3)#)!砐~

KDE 下的 kppp 功能十分强大，完善。
尝试用 ppp-on　　的 C 代码拨号
　　connect
总是在输入用户名后，无法等到要求输入密码
看/var/log/messages，
看到总是：
expect (:)

问题解决的方法：
　　我们最后使用了 kde 桌面环境中 kppp 的拨号参数，这样可以不经过 chat 进行验证而直

接建立 ppp 连接。这一点可以通过在拨号的过程中查看/var/log/messages 文件得知（这是我们在标准 PC 上做实验得出的结果）。Kppp 拨号参数的获得：在 kppp 进行拨号时，监视启动 X 的终端的输出得到。

具体的参数参见 diald.ttyS1 文件和 ttys1 文件（这两个文件是在 5272 上的）。

diald.ttyS1：【这些都是由 diald 传给 pppd 的参数】

```
modem
mode ppp
-pidstring
speed 38400
device /dev/ttyS1
crtscts
defaultroute
connect /etc/config/connect
disconnect /etc/config/disconnect
pppd-options file /etc/config/ttys1
redial-timeout 1
dynamic
local 0.0.0.0
remote 0.0.0.0
netmask 0.0.0.0
```

ttys1：

```
name 169
remotename dialout
usepeerdns
debug
noauth
```

经过改进后，所有已知的 ISP 我们都拨通了（165 除外），真正实现了脚本的通用性。

关于 PAP 验证和 CHAP 验证：

大部分 ISP 都支持 PAP 验证，或同时支持多种验证方式。

我们缺省使用 PAP。

LINEO 的拨号 UI 中就有一项要用户指定验证方式，我们这里没有做，所以只能支持 PAP 方式。对于只支持 CHAP 方式的 ISP 就无法拨通。不过，幸运的是，现在大多数的 ISP 都支持 PAP 验证方式。

pppd 版本号为 2.3.8

在 minicom 终端下可以直接输入 pppd 命令行手工拨号，

记住不能直接用 chat 拨号，必须由 pppd 来调用 chat 执行。

调用 chat 可以写在文件中【已实现】，也可以完全由字符串组成。

pppd /dev/ttyS1 38400 crtscts connect /etc/config/connect name 169 debug noauth noipdefault defaultroute remotename dialout &

实验三十： VHDL 语言和 CPLD 实验

一、实验目的：

1、了解 VHDL 语言的发展过程和用它进行工程设计的优点。掌握 VHDL 的基本语法结构，熟悉 VHDL 代码的执行流程。

2、用 VHDL 编写代码，实现一些常用的逻辑功能，加深对 VHDL 的语法结构和代码执行流程的理解。

3、了解 CPLD 的特点，掌握使用下载电缆下载 CPLD 代码的方法。

4、熟悉用 XILINX 公司的 ISE 4.1I 软件设计逻辑电路的工作流程。掌握用 VHDL 语言编写简单的逻辑代码，并通过下载电缆下载到 CPLD 上实现。

二、实验原理和说明：

1、VHDL 的发展过程及其优点。

VHDL 的英文全名是 Very-High-Speed Integrated Circuit HardwareDescription Language,诞生于 1982 年。1987 年底，VHDL 被 IEEE 和美国国防部确认为标准硬件描述语言 。自 IEEE 公布了 VHDL 的标准版本，IEEE-1076（简称 87 版)之后，各 EDA 公司相继推出了自己的 VHDL 设计环境，或宣布自己的设计工具可以和 VHDL 接口。此后 VHDL 在电子设计领域得到了广泛的接受，并逐步取代了原有的非标准的硬件描述语言。1993 年，IEEE 对 VHDL 进行了修订，从更高的抽象层次和系统描述能力上扩展 VHDL 的内容，公布了新版本的 VHDL，即 IEEE 标准的 1076-1993 版本，（简称 93 版）。现在，VHDL 和 Verilog 作为 IEEE 的工业标准硬件描述语言，又得到众多 EDA 公司的支持，在电子工程领域，已成为事实上的通用硬件描述语言。

VHDL 主要用于描述数字系统的结构、行为、功能和接口。除了含有许多具有硬件特征的语句外，VHDL 的语言形式、描述风格和句法十分类似于一般的计算机高级语言。VHDL 的程序结构特点是将一项工程设计，或称设计实体（可以是一个元件，一个电路模块或一个系统）分成外部（或称可是部分,及端口)和内部（或称不可视部分），既涉及实体的内部功能和算法完成部分。在对一个设计实体定义了外部界面后，一旦其内部开发完成后，其他的设计就可以直接调用这个实体。这种将设计实体分成内外部分的概念是 VHDL 系统设计的基本点。应用 VHDL 进行工程设计的优点是多方面的：

（1）与其他的硬件描述语言相比，VHDL 具有更强的行为描述能力，从而决定了他成为系统设计领域最佳的硬件描述语言。强大的行为描述能力是避开具体的器件结构，从逻辑行为上描述和设计大规模电子系统的重要保证。

（2）VHDL 丰富的仿真语句和库函数，使得在任何大系统的设计早期就能查验设计系统的功能可行性，随时可对设计进行仿真模拟。

（3）VHDL 语句的行为描述能力和程序结构决定了他具有支持大规模设计的分解和已有设计的再利用功能。符合市场需求的大规模系统高效，高速的完成必须有多人甚至多个代发组共同并行工作才能实现。

（4）对于用 VHDL 完成的一个确定的设计，可以利用 EDA 工具进行逻辑综合和优化，并自动的把 VHDL 描述设计转变成门级网表。

（5）VHDL 对设计的描述具有相对独立性，设计者可以不懂硬件的结构，也不必管最终设计实现的目标器件是什么，而进行独立的设计。

2、VHDL 的基本语法结构。

2.1　概述。

一个完整的 VHDL 语言程序通常包含实体(Entity)、结构体（Architecture）、配置（Configuration）、包集合（Package）、库（Library）5 个部分。前 4 种是可分别编译的源设计单元。实体用于描述所设计的系统的外部接口信号；结构体用于描述系统内部的结构和行为；包集合存放各设计模块都能共享的数据类型、常数和子程序等。配置用于从库中选取所需单元来组成系统设计的不同版本；库存放已经编译的实体、构造体、包集合和配置。库可由用户生成或由 ASIC 芯片制造商提供，以便在设计中为大家所共享。

2.2　VHDL 格式。

电路基本结构都由实体说明（Entity Declaration）和构造体（Architecture Body）两部分构成。实体说明部分规定了设计单元的输入输出接口信号和引脚，而构造体部分定义了设计单元的具体构造和操作（行为）。具体结构如下所示：

2.2.1　实体说明(Entity)：

Entity 实体名 IS
[GENERIC （类型参数说明）；]
[PORT （端口说明）；]
END 实体名；

一个基本单元设计的实体说明以“Entity 实体名 IS”开始至“END 实体名”结束。在实体说明中应给出实体名，并描述实体的外部接口情况。此时，实体被视为“黑盒”，不管其内部结构功能如何，只描述它的输入输出信号。

类属参数说明语句的书写格式为：

GENERIC （常数名： 数据类型： = 设定值；
•
•
常数名： 数据类型： = 设定值）；

端口说明语句的书写格式为：

PORT （端口信号名，{端口信号名}： 端口模式 数据类型；）
•
•
端口信号名，{端口信号名}： 端口模式 数据类型）；

a、 端口信号名是赋给每个外部引脚的名称，通常用英文字母和数字来命名，端口信号名在实体中必须是唯一的。

b、 端口模式用来说明信号的方向，详细的端口方向说明如下：

IN ： 输入
OUT ： 输出（构造体内不能再用）
INOUT ： 双向（可以输入，也可以输出）
BUFFER ： 输出（构造体内可以再用），可以读写
LINKAGE ： 不指定方向，无论哪一个方向都可以连接

c、 数据类型是端口信号的取值类型，常见的有下面几种：

BIT ：位类型，取值为 0、1，由 STANDARD 程序包定义；
BIT _VECTOR ：位向量类型，是 BIT 的组合；
STD_LOGIC ： 工业标准的逻辑类型，取值为 0、1、X、Z，

由 STD_LOGIC_1164 程序包定义；

INTEGER ： 整数类型，可用作循环的指针或常数，通常不用作 I/O 信号；

STD_LOGIC_VECTOR ：工业标准的逻辑向量类型，是 STD_LOGIC 的组合；

BOOLEAN ： 布尔类型，取值 FALSE、TRUE。

2.2.2 构造体（Architecture）：

ARCHITECTURE 构造体名 OF 实体名 IS

[定义语句] 内部信号，常数，数据类型，函数等的定义；

BEGIN

[并行处理语句]；

END 构造体名；

一个构造体从“ARCHITECTURE 构造体名 OF 实体名 IS”开始至“END 构造体名”结束。一个实体中可以有一个或几个构造体。一个实体中如有几个构造体，则构造体名不能重复。构造体中的定义语句位于 ARCHITECTURE 和 BEGIN 之间，用于对构造体内部所使用的信号（SIGNAL）、常数（CONSTANT）、数据类型、元件（COMPONENT）和过程（PROCEDURE）等进行定义。

2.2.3 子结构描述语句：

VHDL 语言有 3 种形式的子结构描述语句：

BLOCK 语句结构

PROCESS 语句结构

SUBPROGRAM 语句结构

a、BLOCK（块）语句结构：

BLOCK 语句的书写格式为：

块结构名

BLOCK [块保护表达式]

[端口说明语句]

[类属参数说明语句]

BEGIN

•

•

END BLOCK 块结构名；

端口说明语句用来对 BLOCK 的端口设置以及对外界信号的连接情况进行说明，可以包含由关键词 PORT、GENERIC、PORT MAP 和 GENERIC MAP 引导的端口说明语句。块的说明语句只适用于当前的 BLOCK。块的说明部分可以定义的对象主要是 USE 语句、子程序、数据类型、子类型、常数、信号和元件。

b、PROCESS（进程）语句结构：

PROCESS 语句的书写格式为：

[进程名]：PROCESS（信号 1，信号 2，……）

[进程说明部分]

BEGIN

•

•

END PROCESS；

进程名是给进程命名，这并不是必须的。括号中的信号是进程的敏感信号，任意一个信号发生变化，进程中由顺序语句定义的行为就会从新执行一遍。进程说明部分对该进程所需的局部数据环境进行定义，主要是定义一些局部量，包括数据类型、常数、变量、属性、子程序等。位于 BEGIN 和 AND PROCESS 之间的是描述进程行为的顺序执行语句，可分为赋侄语句、进程启动语句、子程序调用语句、顺序描述语句和进程跳出语句等。进程行为的结果可以赋给信号，并通过信号被其他的 PROCESS 或 BLOCK 读取或赋值。一个构造体中可以有多个 PROCESS 结构，每个进程可以在任何时刻被激活或者启动。而所有被激活的进程都是并行运行的。

在进程设计时要注意以下几个方面的问题：

- 同一构造体中的进程是并行的，但同一进程中的逻辑描述是顺序执行的。
- 进程是由敏感信号的变化启动的，如果没有敏感信号，则在进程中必须有一

个显示的 WAIT 语句来激励。

- 构造体中多个进程之间的通信是通过信号和共享变量值来实现的。也就是说，对构造体而言，信号具有全局特性，是进程间进行并行联系的重要途径。因此，在任何一个进程的说明部分不允许定义信号和共享变量。

c、SUBPROGRAM（子程序）语句结构：

在 VHDL 中有两种类型的子程序：函数（Function）和过程（Procedure）。

- VHDL 语言中函数语句的格式为：

```
FUNCTION  函数名 （参数 1；参数 2；……）
        RETURN  数据类型名  IS
    [定义语句]
BEGIN
    [顺序处理语句]
    RETURN [返回变量值]
END  函数名；
```

在 VHDL 语言中，FUNCTION 语句只能计算数值，不能改变其参数值，所以其参数模式只能是 IN，通常省略不写。FUNCTION 的输入值由调用者复制到参数中，如没有特别指定，在 FUNCTION 语句中按常数处理。

- VHDL 语言中过程语句的格式为：

```
PROCEDURE  过程名 （参数 1；参数 2；……） IS
[定义语句]    （变量等定义）
BEGIN
[顺序处理语句]   （过程的语句）
END  过程名；
```

在 PROCEDURE 结构中，参数可以是 IN、OUT、INOUT 等多中模式。与 PROCESS 相同的是，过程结构中的语句也是顺序执行的。与 FUNCTION 一样，过程体中的参数说明部分只是局部的，其中的各种定义只能用于过程体内。

2.3　VHDL 的常用语句。

VHDL 的描述语句分为并行语句和顺序语句两种。

2.3.1 并行语句。

常用的并行语句有信号赋值语句、条件赋值语句和元件例化语句。

a、 信号赋值语句：

例如：　C1 <=　NOT（a　AND　b）;

　　C2 <=　b　AND　d;

表达式中的信号 a，b，c 是语句的敏感信号，当信号 a 发生变化时语句 1 被执行，当信号 d 发生变化时语句 2 被执行，当信号 b 发生变化时语句 1 和语句 2 同时被执行。这就并行语句之间的关系。

b、 条件赋值语句：

- WITH_SELECT_WHEN 语句：

下面是一个 4 选 1 电路的例子。

```
LIBRARY　IEEE;
  USE　IEEE.　STD_LOGIC_1164.ALL;
  USE　IEEE.　STD_LOGIC_UNSIGNED.ALL;
ENTITY　mun4　IS
   PORT(input :　IN　STD_LOGIC_VECTOR(3　DOWNTO　0);
         sel　:　IN　STD_LOGIC_VECTOR(1　DOWNTO　0);
         x　　:　OUT　STD_LOGIC);
END　mun4;
ARCHITECTURE　rt1　OF　IS
BEGIN
   WITH　SEL　SELECT
      x <= input(0)　when　sel = 0;
      x <= input(1)　when　sel = 1;
      x <= input(2)　when　sel = 2;
      x <= input(3)　when　sel = 3;
END　rt1;
```

注意：用 WITH_SELECT_WHEN 语句赋值，必须列出所有的输入取值。

- WHEN_ELSE 语句：

用此语句描述上面的 4 选 1 电路的构造体如下：

```
ARCHITECTURE　rt1　OF　mun4　IS
   BEGIN
      y <= input(0)　WHEN　sel = 0　ELSE
           input(1)　WHEN　sel = 1　ELSE
           input(2)　WHEN　sel = 2　ELSE
           input(3);
END rt1;
```

c、 元件例化语句。

元件例化语句用来调用库单元或低一级的实体，通过关联表将实际信号与定义断口对应联系起来。元件例化语句在后面的例子中可以看到，此处略去。

2.3.2 顺序语句。

a、 WAIT 语句：

进程在运行时总是处于两种状态：执行和挂起。当进程执行到 WAIT 语句时就被挂起，并设置好再次执行的条件。WAIT 语句有如下四种格式：

WAIT　　：　无限等待
WAIT　ON　　：　敏感信号变化
WAIT　UNTIL　：　条件满足
WAIT　FOR　　：　时间到

b、 断言语句：

断言语句主要用于程序防真、调试中的人机交互，它可以给出一个文字串作为警告和错误信息。ASSERT 语句的格式为：

ASSERT　条件　[REPORT　输出信息]　[SEVERITY　级别]

当执行 ASSERT 语句时，先判断条件，如条件为“真”则执行下面的语句，如条件为“假”则输出错误信息。SEVERITY 后面是错误严重的级别，根据错误严重程度的不同共分为四个级别：FAILURE、ERROR、WARNING、NOTE。

c、 信号带入语句：

例如：　　a <= b;　　把右边信号量表达式的值赋给左边信号量。

2.4　VHDL 语言的数据类型及运算符。

2.4.1　VHDL 语言的客体及其分类。

VHDL 语言共定义了三类客体：变量、常数和信号。

a、 常数（Constant）

常数说明的一般格式为：

CONSTANT　常数名：　常数类型：　= 表达式；

b、 变量：

变量说明的一般格式为：

VARIABLE　变量名：　数据类型　约束条件：　= 表达式；

注意：变量在赋值时不能产生附加延时。

c、 信号：

信号说明的一般格式为：

SIGNAL　信号名：　数据类型　约束条件　表达式；

注意：信号是一个全局变量，可以用于进程与进程之间通信，信号在赋值时可以产生附加延时。

2.4.2　VHDL 语言的数据类型。

VHDL 语言的数据类型可分为标准的数据类型和用户定义数据类型。

a、标准的数据类型有 10 种：

整数（Integer）：在 VHDL 中，整数的范围从-247483647 到+2147483646。对整数不能用逻辑操作符。

实数（Real）：实数的定义值范围从-1.0E+38 到+1.0E38 。

位（Bit）：在数字系统中，位值常用‘0’或‘1’来表示。

位矢量（Bit_Vector）：位矢量是用双引号括起来的一组位数据，如：“0011010”。

布尔量（Boolean）：布尔量有“真”或“假”两种状态。

字符（Character）：所定义的字符量通常为单引号括起来，如‘A’、‘5’。

字符串（String）：字符串是用双引号括起来的一组字符，如“ABCD”。

时间（Time）：时间数据包含整数和单位两部分，在整数和时间之间至少应留一个空格，如 10 ns，3 min 等。

错误等级（Severity Level）：错误等级共有 4 级：NOTE（注意）、WARNING（警告）、ERROR（错误）、FAILURE（失败）。

自然数（Natural）、正整数（Positive）。

b、用户定义的数据类型：

用户定义的数据类型说明的格式为：

TYPE　数据类型名　{，数据类型名}　数据类型定义；

用户定义的数据类型是利用其他已定义的说明所进行的假定义，是不能进行逻辑综合的。

2.4.3　VHDL 语言的运算符。

VHDL 语言共有 4 类运算符，可以分别进行逻辑运算、关系运算、算术运算和并置运算。要注意的是，被操作数所操作的的对象是操作数，且操作数的类型应该和操作符所要求的一致。下面表 1 列出了 VHDL 的运算符及其优先级次序。

优先级顺序	运算符类型	运算符	功能
低	逻辑运算符	AND	逻辑与
		OR	逻辑或
		NAND	逻辑与非
		NOR	逻辑或非
		XOR	逻辑异或
		NOT	取反
	关系运算符	=	等号
		/=	不等与
		<	小于
		>	大于
		<=	小于等于
		>=	大于等于
	算术运算符	+	加
		_	减
	并置运算符	&	并置
	算术运算符	+	正
		_	负
		*	乘
		/	除
		MOD	求模
		REM	求余
		**	指数
高		ABS	取绝对值

表 1　　VHDL 的运算符优先级次序表

2.5　　常用电路的 VHDL 的描述。

下面举例介绍三种常用电路的 VHDL 语言的描述。

2.5.1　寄存器（Register）

可利用信号的属性判断时钟沿：

例如：

```
PROCESS(clk, d)
BEGIN
IF (clk ‘EVENT　AND　clk = ‘1‘)　-- 判断 clk 的上升沿
    a <= b;                          -- 数据寄存
END IF;
END PROCESS;
```

2.5.2　三态输出锁存器

例如：

```
ENTITY　latch　IS
    PORT ( b:　IN STD_LOGIC_VECTOR(7 DOWNTO 0);
           c :　OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
           clk, oe :　IN STD_LOGIC );
END　latch;
ARCHITECTURE　mx1　OF　latch　IS
    SIGNAL　x1 :　STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
PROCESS(clk, d)
    BEGIN
        IF (clk ‘EVENT　AND　clk = ‘1’)　THEN
            x1 <= b;
        END　IF;
END　PROCESS;
c <= x1　WHEN (oe = ‘0’)
ELSE “ZZZZZZZZ”;
END　mxl;
```

2.5.3　时钟分频计数器

例如：把一个 10KHz 的时钟输入信号分频为 1KHz 输出信号可以这样写：

```
ENTITY　clkcounter　IS
    PORT (clock1 :　IN STD_LOGIC;
          clock2 :　OUT STD_LOGIC;);
END　clkcounter;
ARCHITECTURE　Behavioral　OF　clkcounter　IS

SIGNAL count :　STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
    PROCESS(clock1)
    BEGIN
        IF (count > "1010")　THEN
            count <= "0000";
        ELSIF (countad >= "0101")　THEN
            clock2 <= '0';
            count <= count + '1';
```

```
        ELSE
            clock2 <= '1';
            count <= count + '1';
        END    IF;
    END PROCESS;
END    Behavioral;
```

2.6　VHDL 语言小结

上面简单介绍了 VHDL 语言的基本语法结构，通过例子可以看出 VHDL 的语法接近高级语言，灵活易用，执行效率高，且被众多流行的 EDA 软件所支持，所以 VHDL 已经成为一种标准易学实用的硬件描述语言，很好的学习和运用 VHDL 语言将大大提高 EDA 设计的灵活性和工作效率，缩短研发周期，降低设计成本。

3、 CPLD 简介

复杂可编程逻辑器件（CPLD——Complex Programmable Logic Device）是 20 世纪 80 年代后期迅速发展起来的新一代可编程 ASIC（Application Specific Integrated Circuits），进入 20 世纪 90 年代后，CPLD 已经成为可编程 ASIC 的主流产品。CPLD 一般都具有可重编程特性，实现的工艺有 EPROM 技术、闪烁 EPROM 技术和 E^2PROM 技术。在互连特性上，CPLD 采用连续互连方式，即用固定长度的金属线实现逻辑单元之间的连接，这种互连结构可以方便的预测设计的时序，同时也保证了 CPLD 的高速性能。目前 CPLD 的集成度一般可达到几千门甚至几万门以上，能够实现较大规模的电路集成。目前设计和生产 CPLD 的公司有许多，下面将以 RJARM9-EDU 多功能教学实验系统中所使用的 Xilinx 公司的 CPLD（XC95144XL）为例，介绍 CPLD 的一些基本功能和硬件设计方法，以及如何使用 VHDL 语言编写代码并通过 CPLD 来实现一些常用的电路模块。

3.1　Xilinx 公司的 CPLD 的特点、烧写和测试方法简介。

Xilinx 公司的 CPLD 系列器件包括 XC9500 系列器件和 CoolRunner XPLA 系列器件。Xilinx CPLD 器件可使用 Foundation 开发软件进行开发设计，也可以使用专门针对 CPLD 器件的 Webpack 开发软件进行开发设计。

XC9500 系列 CPLD 器件采用快闪存储技术（FastFlash），与 E^2CMOS 工艺相比，功耗明显降低，可用门数多达 6400 个，系统时钟频率最高可达 200MHz。XC9500 系列 CPLD 符合 PCI 总先规范；含有 JTAG 测试接口电路，具有可测试性；具有在系统可编（ISP——In System Programmable）能力。XC9500 系列器件分为 XC9500 5V 器件、XC9500XL 3.3V 器件和 XC9500XV 2.5V 器件三种类型，RUIJIARM9-EDU 教学实验系统中使用的 XC95144XL 就是 XC9500XL 3.3V 器件中的一种。

XC9500 系列器件主要有以下几个特点：

（1） 高密度。XC9500 系列器件内有 36——288 的宏单元（每个宏单元内包含一个寄存器），800——6400 个等效门，封装引脚 44——352 个。

（2） 高性能。XC9500 系列器件所有信号的延时均相同，而与路径无关。XC9500XL CPLD 器件 t_{pd}（tpd 代表 pin-to-pin 时延）最快可达 4ns，相应的计数器频率 f_{CNT} 可达 200MHz。

（3） 在系统可编程。所有 XC9500 系列器件均含有 JTAG 测试接口电路，具有 5V 或 3.3V 在系统可编程 ISP 能力，且达到最小 1 万次编程/擦写次数。在系统编程通

过边界扫描测试引脚进行。

（4） 快闪存储技术。所有的 XC9500 器件都采用先进的快闪存储技术，比 E^2COMS 工艺的功耗明显低。

（5） 5V 和 3.3V 工作电压混合模式。XC9500 系列器件可在 5V 正常电压和 3.3V/2.5V 的低电压条件下安全工作。XC9500 CPLD 器件的输出可作为 XC9500 CPLD 器件的输入信号，而 XC9500 CPLD 器件的输出也可作为 XC9500XL CPLD 器件的输入信号。

（6） 保密和抗干扰。XC9500 器件包含先进的数据保密特性，可以完全保护编程数据不被非法读取和擦除。XC9500 器件的每个 I/O 都有一个可编程输出摆率控制位从而可减小系统噪声。

（7） 驱动负载能力强。XC9500 CPLD 每个输入/输出口的负载电流可达 24mA，可直接驱动 LED 显示，无须外加驱动电路。

（8） 增强引脚锁定功能。XC9500 系列器件的结构特性着重系统内编程的要求，增强的引脚锁定功能可以避免重做印刷电路板。

3.1.1 Xilinx CPLD 的边界扫描。

Xilinx CPLD 符合 IEEE 1149.1 边界扫描测试结构。如下面图 1 所示，每个支持边界扫描的器件均有一个移位寄存器。移位寄存器由边界扫描单元、一个 4 线或 5 线的 TAP 口和一个状态机（TAP 控制器）组成。当器件工作在 JTAG BST 模式时，使用 4 个 I/O 脚和一个可选引脚 TRST 作为 JTAG 引脚。几个 TAP 引脚分别为：

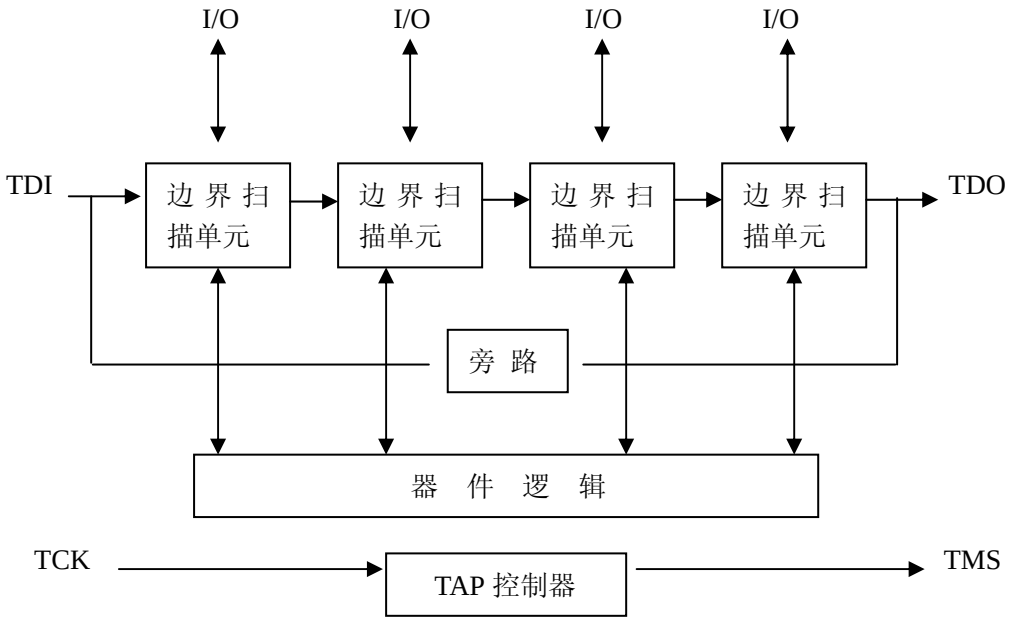


图 1　边界扫描原理图

（1）测试时钟 TCK（Test Clock）：用来同步内部边界扫描状态机（TAP 控制器）操作的时钟信号。

（2）测试模式选择 TMS（Test Mode Select）：内部状态机的模式选择信号，在 TCK 上升延到来时其电平高低决定了下一个状态机的状态。

（3）测试数据输入 TDI（Test Data In）：指令和测试编程的串行数据输入引脚，数据在 TCK

上升延输入。

（4）测试数据输出 TDO（Test Data Out）：指令和测试编程的串行数据输出引脚，数据在 TCK 的下降延输出。如果数据不在输出时，该脚为三态。

（5）测试复位输入 TRST（Test Reset）：异步复位端口，当为低电平时，内部状态机，立即跳至复位状态。由于此脚为可选引用脚，同时内部状态机的同步复位机制较好，因此，器件中通常没有此脚。

对于多个边界扫描器件，可以用菊花链式串联起来，所有器件共用 TCK 和 TMS 信号。前一个器件的 TDO 和后一个器件的 TDI 相连，这样从第一个 TDI 到最后一个 TDO 之间就像一个固定长度的独立的移位寄存器。

JTAG BST 需要以下几个寄存器：

- 指令寄存器：用来决定是否进行测试或访问数据寄存器操作。
- 旁路寄存器：该 1 路寄存器用来提供 TDI 和 TDO 的最小串行通道。
- 边界扫描寄存器：由器件引脚上的所有边界扫描单元构成。

由于下面的实验并没有涉及到许多边界扫描的问题，所以此处不详细介绍了！

3.1.2　Xilinx 器件下载。

对 Xilinx CPLD/FPGA 进行编程，可用 MultiLINX Cable、Parallel Cable 或 Xchecker Cable 三种下载电缆。

除了用 Xilinx 公司提供的上面三种下载电缆外，我们还可以使用并行下载电缆，并行下载电缆只在 PC 机上使用，它连接 PC 机的并口上，可以支持以下系列器件：XC4000XL/XV/EX/E、SPARTAN、SPARTAN-XL、XC5200、XC9500、XC9500XL、VIRTEX 和 VIRTEX-E。图 2 给出了并行下载电缆的电路原理图，与 Xilinx 公司提供的下载电缆相比，此下载电缆使用一片 GAL16V8C 完成了两片 74HC125 的功能，驱动能力更强。与目标板的接口也不是 9 芯线，而是 6 芯线，6 芯线的接口定义如下：

CPLD：1——VCC ， 2——GND ， 3——TCK ， 4——TDO ， 5——TDI ， 6——TMS

FPGA：1——VCC ， 2——GND ， 3——CCLK ， 4——D/P ， 5——DIN， 6——PROG

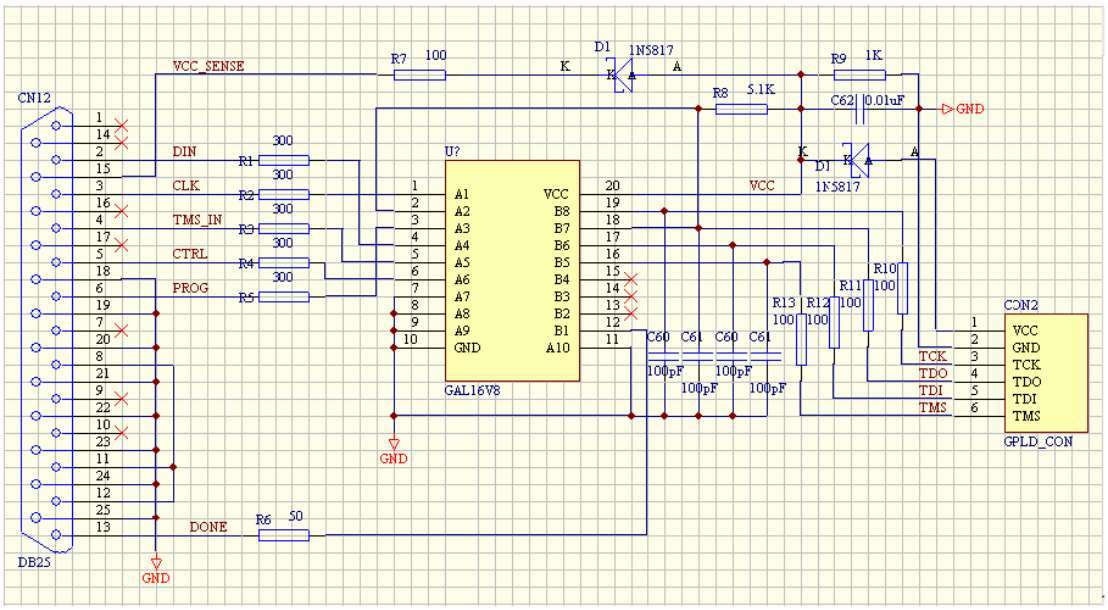


图 2

3.2　Xilinx ISE 集成软件环境的使用方法简介。

下面通过一个具体的小例子来介绍 Xilinx ISE 集成软件环境的使用方法。

在 PC 机上（Windows 操作系统）安装好 Xilinx ISE4.X（你也可以安装高与 4.X 的版本）和 ModelSim VHDL 后，可选择：开始——程序——Xilinx ISE4.X——Project Navigator 来启动软件。

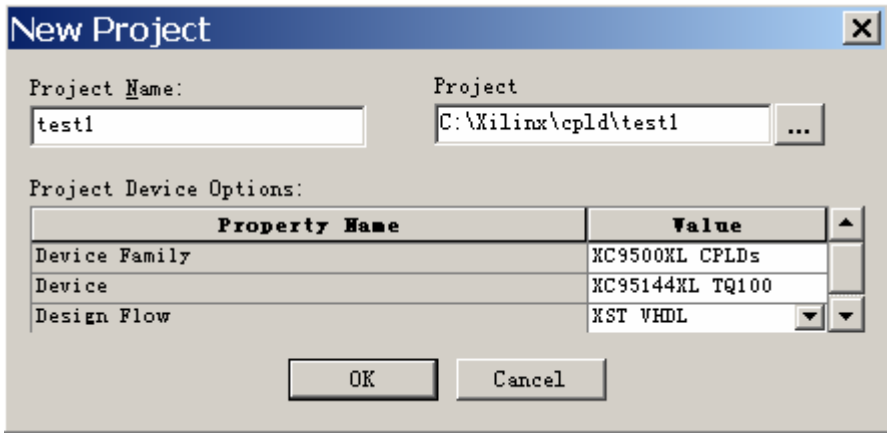
3.2.1 创建一个新工程。

按照以下步骤来创建一个新工程：

（1）选择 File——New Project，出现如图 3 所示的对话框；

（2）在 New Project 对话框中的 Project Location 下，指定新工程所存放的路径；

（3）在 Project Name 下键入新工程的名称 test1；



（4）使用 Value 处的下拉菜单可对每种属性进行选择。我们选择如下属性值：

图 3

Device Family（器件系列）　：　XC9500XL CPLDs

Device（器件）　：　XC95144XL TQ100

Design Flow（设计流程）　：　XST VHDL

点击 OK，ISE 会自动在工程项导航器中创建和显示新工程项。

3.2.2　创建一个计数器模块。

按照以下步骤创建一个计数器模块：

（1）选择 Project ——New Source；

（2）选择 VHDL 模块（Module）作为源程序类型；

（3）在文件名中键入“counter”；

（4）点击 Next;

（5）再点击 Next;

（6）点击 Finish 完成新源程序模块的创建。新源程序模块 counter.vhd 将会显示再编辑窗口中。

3.2.3　利用计数器模板修改你的计数器模块。

利用 ISE 的语言模板（ISE language Template）工具可以很方便的完成这个计数器模块。

（1）选择 Edit——Language Templates 来打开语言模板，或者通过点击工具栏中最右边的灯泡按钮来打开语言模板。

（2）在语言模板（Language Templates）窗口中，通过单击“+”符号展开 VHDL 下的综合模板（Synthesis Templetes）；

（3）从综合模板中选择计数器模板（Counter Template），并把它拖动或粘贴到源程序

counter.vhd 的 begin 和 end 之间，关闭语言模板窗口。

（4）把带有注释符号“--”的计数器端口定义语句剪切并粘贴到计数器的实体（entity）描述语句中去，并去掉其前面的注释符号和 COUNT 端口定义语句后的分号。计数器端口定义语句如下：

CLK: in STD_LOGIC;

RESET: in STD_LOGIC;

CE, LOAD, DIR: in STD_LOGIC;

DIN: in STD_LOGIC_VECTOR(3 downto 0);

COUNT: inout STD_LOGIC_VECTOR(3 downto 0)

（5）选择 File——Save，保存 counter.vhd 源程序。此时，你的 counter.vhd 源程序应和图 4 中的 VHDL 描述相同。

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity counter is
    Port (
     CLK: in STD_LOGIC;
     RESET: in STD_LOGIC;
     CE, LOAD, DIR: in STD_LOGIC;
     DIN: in STD_LOGIC_VECTOR(3 downto 0);
     COUNT: inout STD_LOGIC_VECTOR(3 downto 0)
    );
end counter;

architecture Behavioral of counter is
begin
process (CLK, RESET)
begin
   if RESET='1' then
      COUNT <= "0000";
   elsif CLK='1' and CLK'event then
      if LOAD='1' then
         COUNT <= DIN;
      else
         if CE='1' then
            if DIR='1' then
               COUNT <= COUNT + 1;
            else
               COUNT <= COUNT - 1;
            end if;
         end if;
      end if;
   end if;
end process;
end Behavioral;
```

图 4

3.2.4　编译源程序并下载编译成功后的代码到 CPLD。

（1）在 Sources in Project 窗口中选中 counter.vhd 模块；

（2）在 Processes for Current Source 窗口中双击”Generate Programming File”即开始编译源程序，（如果编译出错，会有出错提示显示）。

（3）编译成功后，即可为 CPLD 创建相应的引脚定义图。方法是：在 Sources in Project 窗口中选中 counter.vhd 模块，然后在 Processes for Current Source 窗口中双击 Design Entry Utilities 目录下的 User Constraints 子目录下的 Assign Pins 项，系统会自动生成一张相应 CPLD 的引脚定义图窗口，在窗口的左边部分列出了是源程序中所定义的所有引脚信号，窗口的右边部分是与你所指定的 CPLD 相对应的器件引脚图，先用鼠标左键单击窗口左面的信号名称，将鼠标移到窗口右边与信号相对应的器件引脚上再单击鼠标左键，这样就把你再源程序中定义的信号放置到相应的器件引脚上了，把所有定义的信号全部放置到相应的引脚上后，点击 File——Save 保存器件引脚定义图，这时的器件引脚定义图如图 5 所示，然后关闭器件引脚定义图窗口，这时会弹出一个对话框询问你是否从新定义器件引脚，点击 Reset 即可。

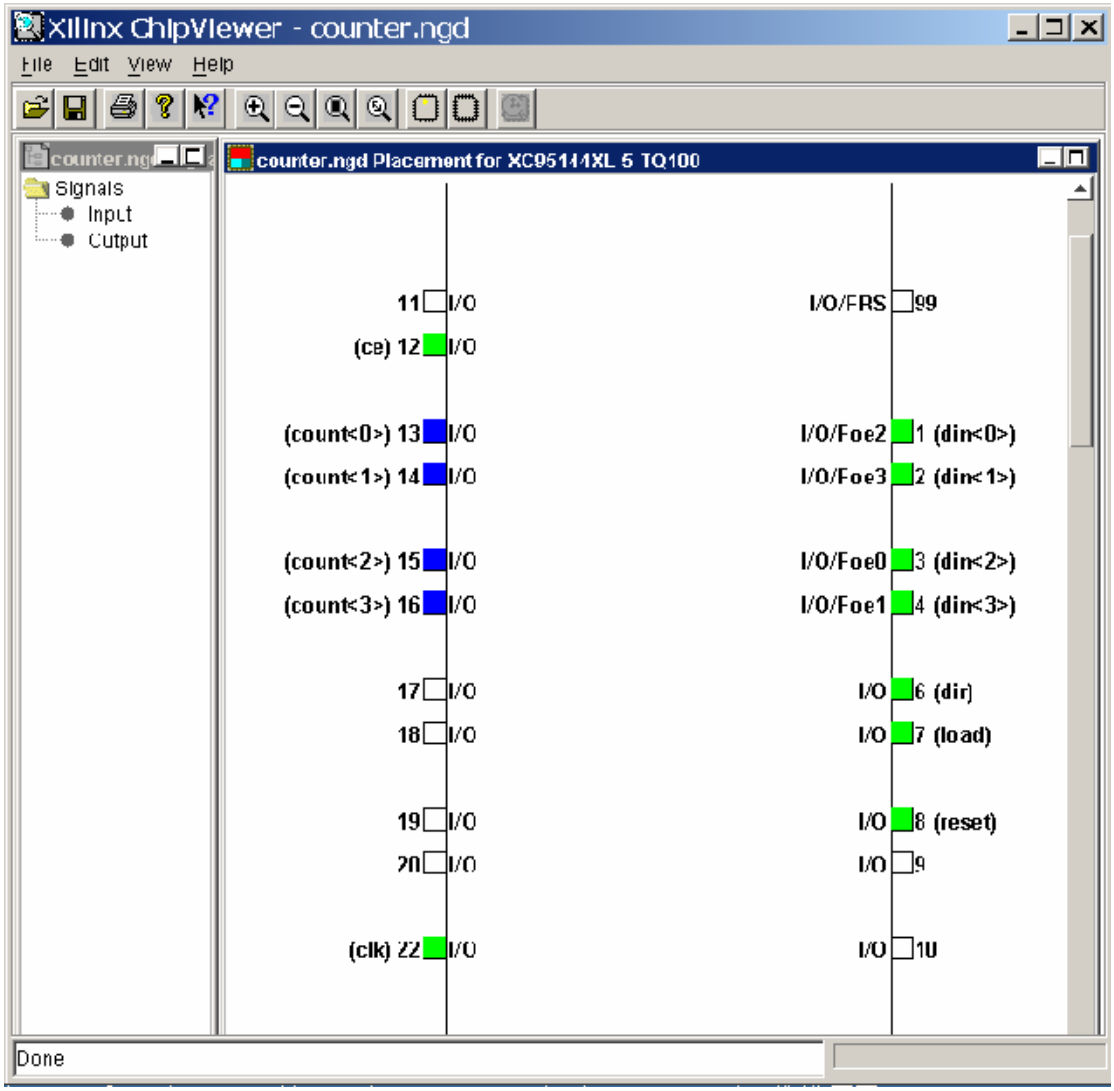


图 5

（4）创建好器件引脚图后，就可以准备把编译成功的代码下载到 CPLD 上了。先把并口下载线的并口连接到 PC 机的并口上，另一端连接到 CPLD 的 JTAG 接口上，然后在 Sources in Project 窗口中选中 counter.vhd 模块，再在 Processes for Current

Source 窗口中双击 Generate Programming File 目录下的“Configure Device”，此时开始依次进行编译源程序、检查下载线连接的工作，如果源程序编译成功且 PC 机并口也连接好了下载线，系统会自动弹出一个如图 6 所示的新窗口，生成可以下载到 CPLD 里并能被其识别的文件（后缀为“.jed”格式）。在图 6 窗口中用鼠标右键单击生成 CPLD 图标，在弹出的如图 7 所示菜单中点击第 1 项 Programme，即可把编译成功后生成的代码下载到 CPLD 中。下载成功后重新启动电路板 CPLD 即可工作了。

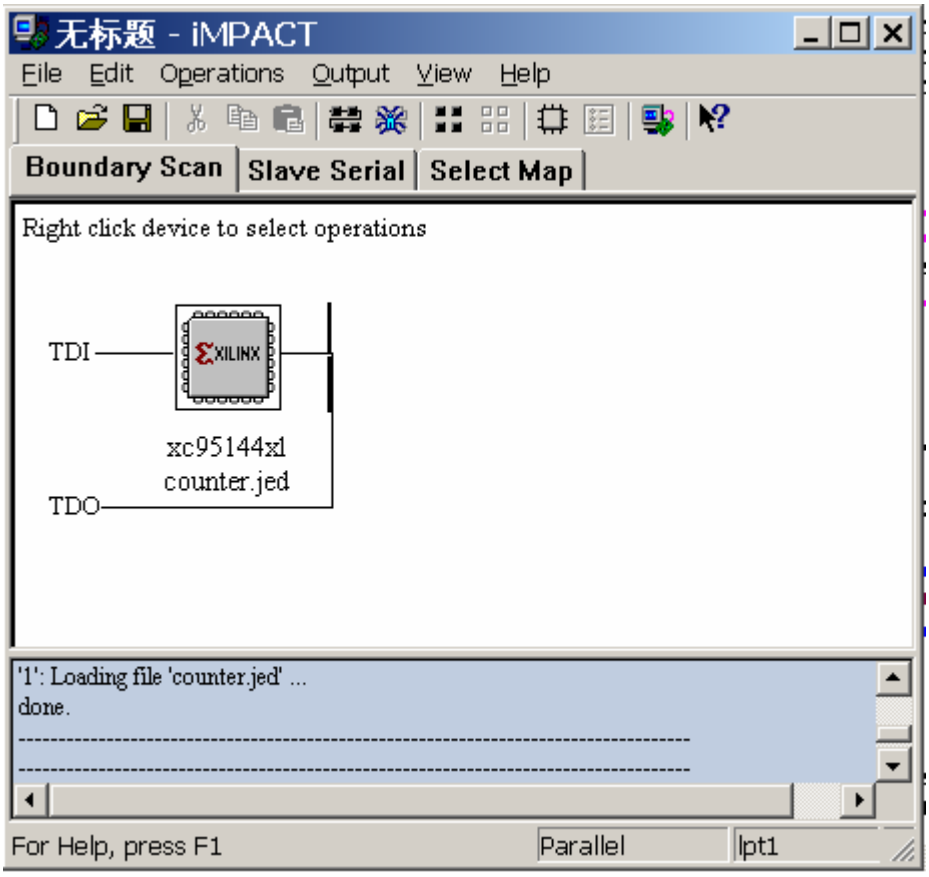


图 6

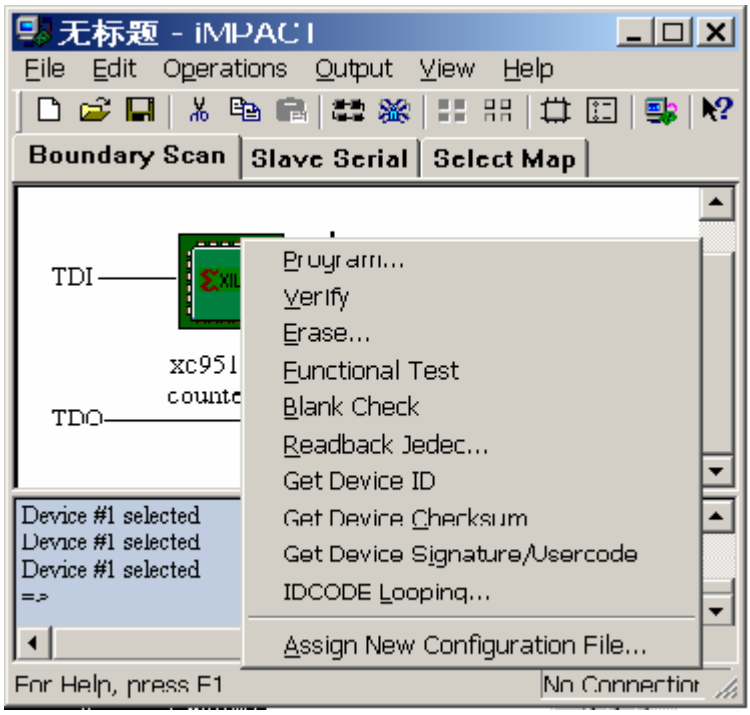


图 7

对于 Xilinx 的 CPLD 器件，下载操作中应注意：

- 下载前应检查下载线的连接是否正确，确保稳定、标准的编程电压。
- 下载结束后，请从启电路板，以便使 CPLD 能够正常工作。

上面简单介绍了 VHDL 和 CPLD 的基本知识，下面开始介绍 RUIJIARM9-EDU 嵌入式教学实验系统的 CPLD 部分。

4、 RUIJIARM9-EDU 嵌入式教学实验系统的 CPLD。

4.1　XC95144XL 简介及其引脚定义。

RUIJIARM9-EDU 教学实验系统选用的 CPLD 是 Xilinx 公司的 XC95144XL-TQ100，使用 3.3V 低电压供电，它提供了 81 个 I/O 脚，最多可提供 144 个宏单元，t_{pd} 最快达 5ns，最高系统时钟为 178MHz。XC95144 的输出电压为 3.3V 或 2.5V，其 I/O 引脚可接受 5V、3.3V 和 2.5V 电压输入，可安全的工作在混合电压系统中，其引脚定义如图 8 所示。

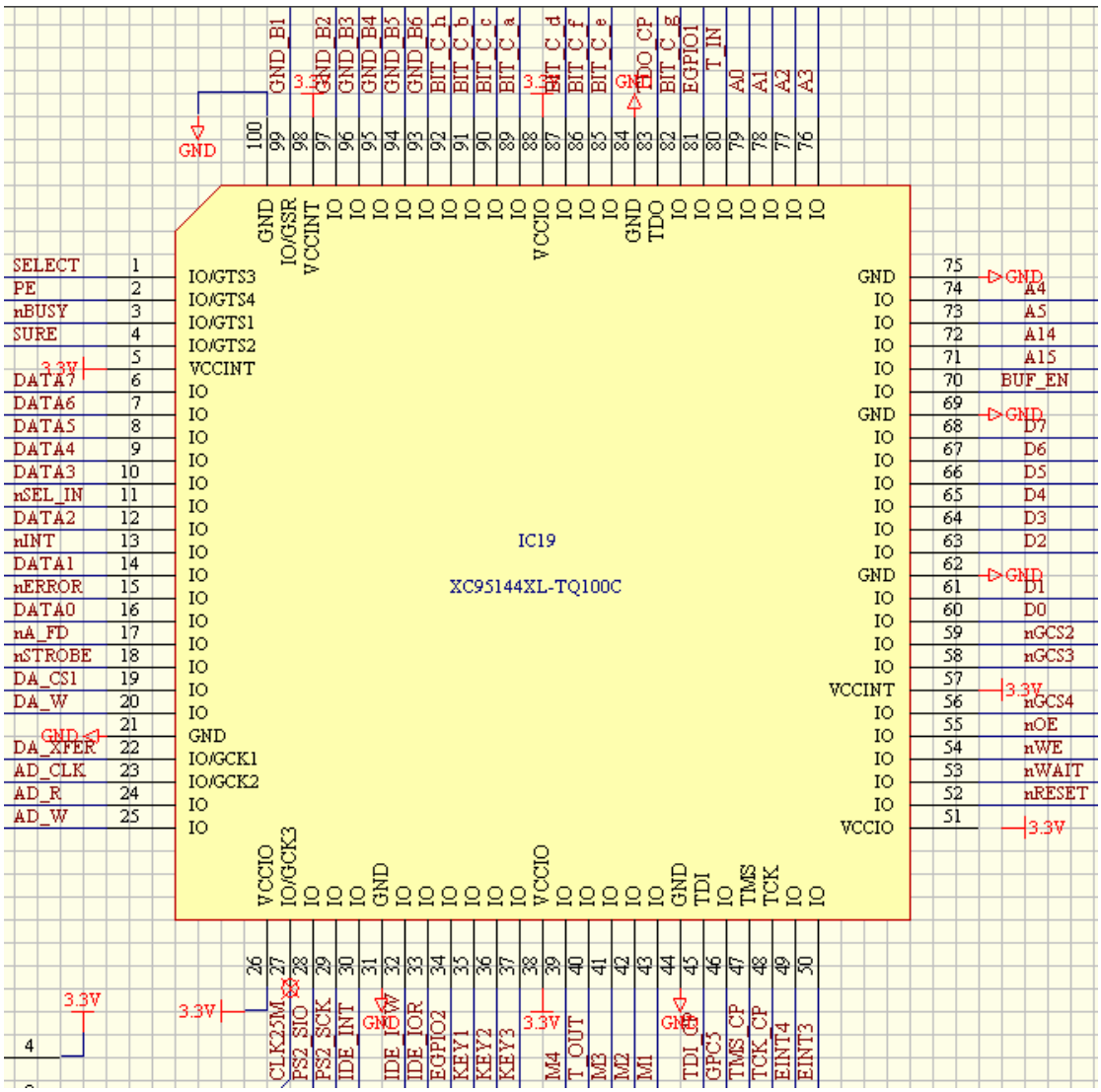


图 8

图 8 中所示的 XC95144XL 的引脚信号定义与 RJARM9-EDU-R2 嵌入式教学实验系统相应功能模块的信号对应如下：

1——4 脚和 6——18 脚对应的是并口的相应信号；

19、20 和 22 脚对应的是 D/A 转换的相应信号；

23、24 和 25 脚对应的是 A/D 转换的相应信号；

27 脚是外部提供给 XC95144XL 的系统时钟信号输入脚，频率为 25MHz；

28、29、49 和 56 脚对应的是 PS2 的相应信号；

30、32、33、50 和 58 脚对应的是 IDE 的相应信号；

34 和 81 脚对应的是扩展口 CON16 上留给用户自己使用的 CPLD 控制信号；

35、36 和 37 脚对应的是 5*4 键盘中的 F1、F2 和 F3 三个键，留给用户自己编程使用；

39、41、42 和 43 脚对应的是步进电机的四步驱动信号；

40 脚对应的是蜂鸣器的控制信号；

45、47、48 和 83 脚对应的是 XC95144XL 的 JTAG 接口信号；

46 脚对应的是从 CPU 引入的一个 GPIO 信号，留给用户编程使用；

52 脚对应的是 CPU 的复位信号；

53 脚对应的是 CPU 的等待信号；

54 脚对应的是 CPU 的写信号；
55 脚对应的是 CPU 的读信号；
59、85——87、89——97 和 99 脚对应的是 LED（带小数点的七段荧光数码管）相应信号；

4.2　RJARM9-EDU-R2 嵌入式教学实验系统中用 CPLD 实现的主要功能。

RJARM9-EDU-R2 嵌入式教学实验系统中用 CPLD 实现的主要功能有：

（1）通过 CPLD 编程实现 LED（七段荧光数码管）的显示控制；
（2）通过 CPLD 编程实现控制步进电机转动的方向和转速；
（3）通过 CPLD 编程实现 IDE 接口的的时钟及控制信号，访问硬盘；
（4）通过 CPLD 编程实现 PS2 接口的时钟及控制信号，外接 PS2 键盘；
（5）通过 CPLD 编程实现 A/D 和 D/A 转换的时钟及控制信号；
（6）通过 CPLD 编程实现控制蜂鸣器鸣叫的时间和鸣叫时声音的高低（由用户自己编程实现）。

下面以用 CPLD 编程实现 LED（七段荧光数码管）的显示控制为例具体介绍怎样用 CPLD 来编程实现所需功能。

4.2.1 用 CPLD 编程实现 LED（七段荧光数码管）的显示控制。

RUIJIARM9-EDU 嵌入式教学实验系统中所使用的荧光数码管是共阴极的，它的引脚定义如图 9 所示。当数码管的 3 和 8 脚（图 9 中的 GND_B 信号）均接低电平时,如其他脚有一个或多个为高电平则对应的荧光管就被点亮。我们把七段荧光数码管的 GND_B 以及其他信号均接到 CPLD 上，这样在用 CPLD 编程实现 LED 的显示控制时，我们所要做的主要是控制好 GND_B 信号变低的时间与其他引脚信号的按照所要求的变高时间配合好即可。

在 RUIJIARM9-EDU 嵌入式教学实验系统中共有 6 个七段荧光数码管，为了即能让 6 个数码管都能够在视觉上同时正常显示又节省 CPLD 的 I/O 资源，我们采用扫描方式来编程实现对 LED 的控制。如图 9 所示，我们使这 6 个七段荧光数码管的信号引脚（除去 GND_B 信号）共用 GPLD 的 8 个 I/O 引脚信号，另外每个七段荧光数码管的 GND_B 信号再分别使用一个 CPLD 的 I/O 信号，这样我们用一定的扫描频率不停的依次扫描这 6 个七段荧光数码管，就可以实现使 6 个七段荧光数码管都能在视觉上同时正常显示。

为了实现按一定的扫描频率对 6 个七段荧光数码管扫描，我们要对 CPLD 外部提供的

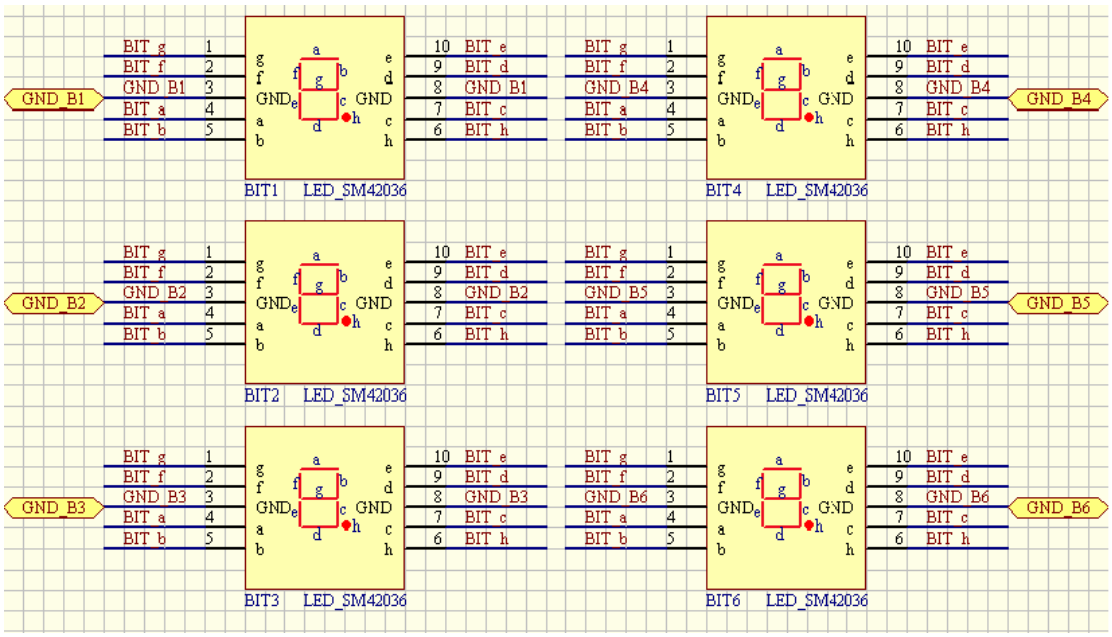


图 9

25MHz 时钟信号 CLK25M 进行分频处理，我们通过计数器计数来产生 6 个七段荧光数码管的扫描时钟，VHDL 代码可以写成如下形式：

```
if (clk25m'event and clk25m = '1') then          --在 clk25m 信号的上升沿开始计数
   if (countled > "10000010011111000" ) then
      countled <= "0000000000000000";        --计数器 countled 清‘0’
      countled2 <= countled2 + '1';          --启动计数器 countled2 开始计数
   else
      countled <= countled + '1';
   end if;
   case countled2 is
      when "000" =>
         ledgnd <= "111110";                 --第 1 个七段荧光数码管工作
      when "001" =>
         ledgnd <= "111101";                 --第 2 个七段荧光数码管工作
      when "010" =>
         ledgnd <= "111011";                 --第 3 个七段荧光数码管工作
      when "011" =>
         ledgnd <= "110111";                 --第 4 个七段荧光数码管工作
      when "100" =>
         ledgnd <= "101111";                 --第 5 个七段荧光数码管工作
      when "101" =>
         ledgnd <= "011111";                 --第 6 个七段荧光数码管工作
      when others =>
         ledgnd <= "111111";                 --当 countled2 的值大与 5 时，所有七段荧光
   end case;                                    数码管都工作
```

```
end if;
```

在上面的代码中我们在外部提供的时钟信号 clk25m 的上升沿使计数器 countled 开始计数。当计数器 countled 每计到“10000010011111000”时将计数器清 0 一次，同时使计数器 countled2 开始计数。每当计数器 countled2 值加 1 时，我们就相应的向 ledgnd 信号送一个值，并且每次所送的值中只有 1 位为‘0’，而 ledgnd 信号的每一位对应着 1 个七段荧光数码管的 GND_B 信号，所以当向 ledgnd 信号上送的值中哪一位为‘0’，其所对应的那个七段荧光数码管就开始工作，这样我们就可以实现按一定的扫描频率来扫描 6 个七段荧光数码管了。如果我们在每次向 ledgnd 信号送一个值的同时也相应的向 leddata 信号送一个值，而 leddata 信号的每一位也对应着每个七段荧光数码管除 3 脚和 8 脚以外的每一个引脚信号，所以当某个七段荧光数码管工作时就可以立即显示出相应的数码来。上面的代码可以修改为如下形式：

```
if (clk25m'event and clk25m = '1') then
   if (countled > "10000010011111000" ) then
      countled <= "0000000000000000";
      countled2 <= countled2 + '1';
   else
      countled <= countled + '1';
   end if;
   case countled2 is
      when "000" =>
         ledgnd <= "111110";
         leddata <= regled1;
      when "001" =>
         ledgnd <= "111101";
         leddata <= regled2;
      when "010" =>
         ledgnd <= "111011";
         leddata <= regled3;
      when "011" =>
         ledgnd <= "110111";
         leddata <= regled4;
      when "100" =>
         ledgnd <= "101111";
         leddata <= regled5;
      when "101" =>
         ledgnd <= "011111";
         leddata <= regled6;
      when others =>
         ledgnd <= "111111";
         leddata <= X"00";
   end case;
end if
```

为了使七段荧光数码管在工作时能够正确的显示出相应的数码来，我们要用 6 个 8 位寄存器 regled1——regled6 来分别存放每次从数据总线上送来的数据，这 6 个 8 位寄存器每一个

都分别对应着一个七段荧光数码管，并且根据 a0 、a1、a2 这 3 位地址线的值来判断每次从数据总线上送来的值应该存放到这 6 个 8 位寄存器中的哪一个里。为此，我们可以将上面的代码进一步修改为如下形式：

```
if (clk25m'event and clk25m = '1') then
   if (countled > "10000010011111000" ) then
      countled <= "00000000000000000";
      countled2 <= countled2 + '1';
   else
      countled <= countled + '1';
   end if;
   case countled2 is
      when "000" =>
         ledgnd <= "111110";
         leddata <= regled1;
      when "001" =>
         ledgnd <= "111101";
         leddata <= regled2;
      when "010" =>
         ledgnd <= "111011";
         leddata <= regled3;
      when "011" =>
         ledgnd <= "110111";
         leddata <= regled4;
      when "100" =>
         ledgnd <= "101111";
         leddata <= regled5;
      when "101" =>
         ledgnd <= "011111";
         leddata <= regled6;
      when others =>
         ledgnd <= "111111";
         leddata <= X"00";
   end case;
   if (ngcs2 = '0' and nwe = '0') then
      if(a(2 downto 0) = "000") then
          regled1(7 downto 0) <= d(7 downto 0);
      elsif (a(2 downto 0) = "001") then
          regled2(7 downto 0) <= d(7 downto 0);
      elsif (a(2 downto 0) = "010") then
          regled3(7 downto 0) <= d(7 downto 0);
      elsif (a(2 downto 0) = "011") then
          regled4(7 downto 0) <= d(7 downto 0);
      elsif (a(2 downto 0) = "100") then
```

```
            regled5(7 downto 0) <= d(7 downto 0);
        elsif (a(2 downto 0) = "101") then
            regled6(7 downto 0) <= d(7 downto 0);
        else
            regled6(7 downto 0) <= ”0000000”;
        end if;
    end if;
end if;
```

由于 CPU 的数据总线是通过数据总线 buffer 后连接到 CPLD 上的，并且数据总线 buffer 的使能信号 buf_en 也连接在 CPLD 的 I/O 引脚上，所以我们可以方便的通过编程控制数据总线 buffer 来配合上面代码的执行。我们用片选 nGCS2 的起始地址来作为 6 个 8 位寄存器 regled1——regled6 的基地址，用 a0、a1、a2 三位地址线来作为 6 个 8 位寄存器的偏移地址，这样我们就可以很方便的访问这 6 个寄存器了。因此，上面的代码又可以修改为如下形式：

```
if (ngcs2 = ‘0’) then          -- 当片选 nGCS2 有效时，数据总线 buffer 使能信号有效
    buf_en <= ‘0’;
else
    buf_en <= ‘1’;
end if;
if (clk25m'event and clk25m = '1') then
    if (countled > "10000010011111000" ) then
        countled <= "00000000000000000";
        countled2 <= countled2 + '1';
    else
        countled <= countled + '1';
    end if;
    case countled2 is
        when "000" =>
            ledgnd <= "111110";
            leddata <= regled1;
        when "001" =>
            ledgnd <= "111101";
            leddata <= regled2;
        when "010" =>
            ledgnd <= "111011";
            leddata <= regled3;
        when "011" =>
            ledgnd <= "110111";
            leddata <= regled4;
        when "100" =>
            ledgnd <= "101111";
            leddata <= regled5;
        when "101" =>
```

```
            ledgnd <= "011111";
            leddata <= regled6;
         when others =>
            ledgnd <= "111111";
            leddata <= X"00";
      end case;
      if (ngcs2 = '0' and nwe = '0') then
         if(a(2 downto 0) = "000") then
            regled1(7 downto 0) <= d(7 downto 0);
         elsif (a(2 downto 0) = "001") then
            regled2(7 downto 0) <= d(7 downto 0);
         elsif (a(2 downto 0) = "010") then
            regled3(7 downto 0) <= d(7 downto 0);
         elsif (a(2 downto 0) = "011") then
            regled4(7 downto 0) <= d(7 downto 0);
         elsif (a(2 downto 0) = "100") then
            regled5(7 downto 0) <= d(7 downto 0);
         elsif (a(2 downto 0) = "101") then
            regled6(7 downto 0) <= d(7 downto 0);
         else
            regled6(7 downto 0) <= ”0000000”;
         end if;
      end if;
   end if;
```

至此，用 CPLD 编程实现控制 6 个七段荧光数码管显示数码的 VHDL 代码基本完成了，在上面代码的前后再分别加上 CPLD 的引脚信号定义语句、寄存器定义语句以及进程的开始和结束语句后，这部分的 VHDL 代码就完全完成了。最终的 VHDL 代码可以写为如下形式：

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RJ2410edu6 is
      Port (                                  --CPLD 的引脚信号定义语句
            nreset          : in    std_logic;
            nwe             : in    std_logic;
            noe             : in    std_logic;
            nwait           : out    std_logic;
            buf_en          : out std_logic;
            clk25m          : in    std_logic;
            a               : in    std_logic_vector(5 downto 0);
            a14             : in    std_logic;
```

```
        a15             : in    std_logic;
        d               : inout    std_logic_vector(7 downto 0);
        egpio1          : in    std_logic;
        egpio2          : in    std_logic;
        gpc5            : in    std_logic;
        ngcs3           : in    std_logic;
        ide_iow         : out std_logic;
        ide_ior         : out std_logic;
        ide_int         : in    std_logic;
        eint3           : out    std_logic;
        ngcs4           : in    std_logic;
        eint4           : out std_logic;
        ps2_sck         : in    std_logic;
        ps2_sio         : in    std_logic;
        SELECT_P        : inout    std_logic;
        PE              : inout std_logic;
        nBUSY           : in    std_logic;
        data            : inout    std_logic_vector(7 downto 0);
        nSEL_IN         : out std_logic;
        nINT            : inout    std_logic;
        nERROR          : inout std_logic;
        nA_FD           : inout std_logic;
        nSTROBE         : inout std_logic;
        SURE            : in std_logic;
        ad_clk          : out std_logic;
        ad_r            : out std_logic;
        ad_w            : out std_logic;
        da_cs1          : out std_logic;
        da_w            : out std_logic;
        da_xfer         : out std_logic;
        ngcs2           : in    std_logic;
        ledgnd          : out std_logic_vector(5 downto 0);
        leddata         : out std_logic_vector(7 downto 0);
        key             : in    std_logic_vector(3 downto 1);
        t_in            : in    std_logic;
        t_out           : out std_logic;
        m               : out std_logic_vector(3 downto 0)
         );
end RJ2410edu6;

architecture Behavioral of RJ2410edu6 is

-- signal for led                                    --CPLD 的内部寄存器定义语句
    signal regled1     :    std_logic_vector(7 downto 0);
```

```
    signal regled2     :   std_logic_vector(7 downto 0);
    signal regled3     :   std_logic_vector(7 downto 0);
    signal regled4     :   std_logic_vector(7 downto 0);
    signal regled5     :   std_logic_vector(7 downto 0);
    signal regled6     :   std_logic_vector(7 downto 0);
    signal countled    :   std_logic_vector(16 downto 0);
    signal countled2 :   std_logic_vector(2 downto 0);

begin                                   --CPLD 的逻辑实现语句（首先应该给在进程中没有
    eint4 <= '1';                         用到的输出信号赋上相应的值，以防止与这些信号
    ad_clk <= 'Z';                        对应的引脚被拉死而导致对 CPLD 的功耗甚至对
    ad_r <= 'Z';                          整个系统的稳定产生影响）
    ad_w <= 'Z';
    da_cs1 <= 'Z';
    da_xfer <= 'Z';
    da_w <= 'Z';
    nSEL_IN <='Z';
    ide_ior <= ‘Z’;
    ide_iow <= ‘Z’;
    eint3 <= ‘1’;
--the process for led
    process(ngcs2,nwe)
    begin
      if (ngcs2 = '0' or ngcs3 = '0') then
        buf_en <= '0';
      else
        buf_en <= '1';
      end if;
      if (clk25m'event and clk25m = '1') then
        if (countled > "10000010011111000" ) then
            countled <= "00000000000000000";
            countled2 <= countled2 + '1';
        else
            countled <= countled + '1';
        end if;
        case countled2 is                   --LED 扫描语句
            when "000" =>
                ledgnd <= "111110";
                leddata <= regled1;
            when "001" =>
                ledgnd <= "111101";
                leddata <= regled2;
            when "010" =>
                ledgnd <= "111011";
```

```
                leddata <= regled3;
            when "011" =>
                ledgnd <= "110111";
                leddata <= regled4;
            when "100" =>
                ledgnd <= "101111";
                leddata <= regled5;
            when "101" =>
                ledgnd <= "011111";
                leddata <= regled6;
            when others =>
                ledgnd <= "111111";
                leddata <= X"00";
        end case;
        if (ngcs2 = '0' and nwe = '0') then                --CPLD 内部寄存器选择访问语句
            if(a(2 downto 0) = "000") then
                regled1(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "001") then
              regled2(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "010") then
              regled3(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "011") then
              regled4(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "100") then
              regled5(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "101") then
              regled6(7 downto 0) <= d(7 downto 0);
            else
              regled6(7 downto 0) <= ”0000000”;
            end if;
        end if;
    end if;
 end process;

end Behavioral;
```

我们再通过 CPLD 的并口下载线把上面的代码下载到 CPLD 中，然后从新启动整个系统即可。我们通过键盘驱动程序把所按下的键值通过编码后转换成与七段荧光数码管相对应的 8 位 2 进制码后通过数据总线传送给 CPLD，这样只要在键盘被按下的同时使片选 nGCS2 有效，并且通过 a0、a1、a2 三根地址线的不同状态来指定要读或写的是那 6 个 8 位寄存器中的哪一个，那么在七段荧光数码管上就会显示出相应的数码来。

经过上面对 VHDL 和 CPLD 相关基本知识的介绍，我们基本了解了用 CPLD 来实现一些常用功能的整个设计流程和设计方法。下面我们就开始做 VHDL 和 CPLD 的实验。

三、 实验内容和步骤：

3.1 熟悉 Xilinx ISE 软件集成环境、并行下载线的连接方法，掌握 CPLD 的 VHDL 代码的下载方法。

实验步骤：

1、 按照实验原理和说明中所介绍的 ISE 软件集成环境的使用方法，创建一个新工程。
2、 按照实验原理和说明中所介绍的方法，把用 CPLD 来控制 LED 显示的 VHDL 代码添加到所创建的新工程中，并执行编译，直至编译成功。
3、 按照实验原理和说明中所介绍的方法，在连接好 CPLD 的并行下载线后把上面用 CPLD 来控制 LED 显示的 VHDL 代码编译并下载到 CPLD 中。
4、 从新启动系统，按键盘来测试 LED 是否可以正确显示键盘中被按键所对应的数码。

3.2 熟悉 VHDL 语言的基本语法结构，用 VHDL 语言编写一些常用的功能模块。

实验步骤：

1、 按照实验原理和说明中所介绍的 VHDL 语言的基本语法结构，参照图 8 中给出的 CPLD 的硬件连接原理图，分别写出上面所介绍的用 CPLD 来控制 LED 显示的 VHDL 代码的引脚信号定义语句、内部寄存器定义语句、计数器实现语句、LED 扫描语句和 CPLD 的内部寄存器选择访问语句。
2、 修改计数器实现语句和 LED 扫描语句，以减慢 LED 的扫描时钟，使 LED 的显示在视觉上出现的闪烁。
3、 在上面介绍的控制 LED 显示的 VHDL 代码进程（process）之外，添加两个新进程。在一个新进程中编写一个分频器使的可以从 AD_CLK 引脚输出符合我们要求的时钟信号；在另一个新进程中编写能够响应键盘中 F1、F2、F3 三个按键的 VHDL 代码，使得按下 F1、F2、F3 三个键时，LED 能够显示相应的数码。

3.3 编写控制步进电机转动的 VHDL 代码。

实验步骤：

参照图 8 中给出的 CPLD 硬件连接图，修改上面介绍的控制 LED 显示的 VHDL 代码，并添加相应的新代码以使步进电机可以按照我们的要求转动。

下面给出一个修改好的可以控制步进电机转动的 VHDL 代码，供大家参考。

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RJ2410edu6 is
    Port (
        nreset      : in  std_logic;
        nwe         : in  std_logic;
        noe         : in  std_logic;
        nwait       : out  std_logic;
        buf_en      : out std_logic;
        clk25m      : in  std_logic;
        a           : in  std_logic_vector(5 downto 0);
        a14         : in  std_logic;
```

```
a15             : in    std_logic;
d               : inout    std_logic_vector(7 downto 0);
egpio1          : in    std_logic;
egpio2          : in    std_logic;
gpc5            : in    std_logic;

ngcs3           : in    std_logic;
ide_iow         : out std_logic;
ide_ior         : out std_logic;
ide_int         : in    std_logic;
eint3           : out    std_logic;

ngcs4           : in    std_logic;
eint4           : out std_logic;
ps2_sck         : in    std_logic;
ps2_sio         : in    std_logic;

SELECT_P        : inout    std_logic;
PE              : inout std_logic;
nBUSY           : in    std_logic;
data            : inout    std_logic_vector(7 downto 0);
nSEL_IN         : out std_logic;
nINT            : inout    std_logic;
nERROR          : inout std_logic;
nA_FD           : inout std_logic;
nSTROBE         : inout std_logic;
SURE            : in std_logic;

ad_clk          : out std_logic;
ad_r            : out std_logic;
ad_w            : out std_logic;
da_cs1          : out std_logic;
da_w            : out std_logic;
da_xfer         : out std_logic;

ngcs2           : in    std_logic;
ledgnd          : out std_logic_vector(5 downto 0);
leddata         : out std_logic_vector(7 downto 0);

key             : in    std_logic_vector(3 downto 1);
t_in            : in    std_logic;
t_out           : out std_logic;
m               : out std_logic_vector(3 downto 0)
   );
```

```
end RJ2410edu6;

architecture Behavioral of RJ2410edu6 is

-- signal for led
    signal regled1     :   std_logic_vector(7 downto 0);
    signal regled2     :   std_logic_vector(7 downto 0);
    signal regled3     :   std_logic_vector(7 downto 0);
    signal regled4     :   std_logic_vector(7 downto 0);
    signal regled5     :   std_logic_vector(7 downto 0);
    signal regled6     :   std_logic_vector(7 downto 0);
    signal countled    :   std_logic_vector(16 downto 0);
    signal countled2 :   std_logic_vector(2 downto 0);

-- signal for electromotor
    signal regbjdj     : std_logic_vector(3 downto 0);

begin
    eint4 <= '1';
    ad_clk <= 'Z';
    ad_r <= 'Z';
    ad_w <= 'Z';
    da_cs1 <= 'Z';
    da_xfer <= 'Z';
    da_w <= 'Z';
    nSEL_IN <='Z';

--the process for led and electromotor
    process(ngcs2,nwe)
    begin
      if (ngcs2 = '0' ) then
          buf_en <= '0';
      else
          buf_en <= '1';
      end if;
      if (clk25m'event and clk25m = '1') then
        if (countled > "10000010011111000" ) then
            countled <= "00000000000000000";
            countled2 <= countled2 + '1';
        else
            countled <= countled + '1';
        end if;
        case countled2 is
              when "000" =>
```

```
                ledgnd <= "111110";
                leddata <= regled1;
            when "001" =>
                ledgnd <= "111101";
                leddata <= regled2;
            when "010" =>
                ledgnd <= "111011";
                leddata <= regled3;
            when "011" =>
                ledgnd <= "110111";
                leddata <= regled4;
            when "100" =>
                ledgnd <= "101111";
                leddata <= regled5;
            when "101" =>
                ledgnd <= "011111";
                leddata <= regled6;
            when "110" =>
                m(3 downto 0) <= regbjdj(3 downto 0);      --for electromotor
            when others =>
                ledgnd <= "111111";
                leddata <= X"00";
        end case;
        if (ngcs2 = '0' and nwe = '0') then
            if(a(2 downto 0) = "000") then
                regled1(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "001") then
              regled2(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "010") then
              regled3(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "011") then
              regled4(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "100") then
              regled5(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "101") then
              regled6(7 downto 0) <= d(7 downto 0);
            elsif (a(2 downto 0) = "110") then
              regbjdj(3 downto 0) <= d(3 downto 0); -- electromotor
            else
              regled6(7 downto 0) <= ”0000000”;
            end if;
        end if;
    end if;
end process;
```

```
end Behavioral;
```

四、思考题：

1、 理解 VHDL 语言的并行执行的特点，找出上面介绍的一些 VHDL 代码中的可能并行执行的部分，在编写代码的过程中思考并总结出怎样能更好的避免在代码并行执行时发生冲突。

2、 思考并总结出用 VHDL 编程时要注意的一些常见问题。例如：给在 VHDL 代码中定义的引脚信号赋值时，一定要注意所要赋值的引脚信号被定义为输入信号还是输出信号或者是既可做输入信号也可做输出信号，只有做输出信号时我们才可以在 VHDL 代码中给其赋值。其他还有一些要注意的常见问题，请同学们在编程后好好总结一下。

一些常用工具：

使用工具

1)Max+Plus II Baseline

Altera Corporation

2)SpDE

Quicklogic Corporation

附：一个简单的 VHDL 的例子：(12 位寄存器）

```
--- VHDL Example
-- User-Defined Macrofunction
ENTITY reg12 IS
    PORT(
        d           : IN BIT_VECTOR(11 DOWNTO 0);
        clk         : IN BIT;
        q           : OUT BIT_VECTOR(11 DOWNTO 0));
END reg12;

ARCHITECTURE a OF reg12 IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL clk = '1';
        q <= d;
    END PROCESS;
END;
```

实验三十一： 仿真器实验

一、实验目的

熟悉 ADS 开发环境，学会 ARM_STAR 仿真器的使用。学习仿真器和 LINUX 下 GDB 连接下的程序调试过程。

二、实验原理和说明

1． 以 ARM_STAR 仿真器为例

1.1 功能介绍

ARM_Star 仿真器完全实现 ARM RDI 1.5 和 RDI 1.51 协议，与 ARM Multi-ICE™ 兼容，支持所有含有 Embedded-ICE Logic 的 ARM 内核 CPU, 在调试软件的控制下，ARM_Star 仿真器可以停止、启动 ARM CPU 的运行，用户通过 ARM_Star 仿真器察看、修改寄存器，存储器，设置断点、单步执行，下载烧写 Flash 程序等。

1.2 主要特点

(1) 程序下载速度快：与 PC 机的通信速度最高可达到 12M, 下载速度达到 100K---200Kbytes 每秒，单步速度可达 85 步每秒。
(2) 支持调试软件多：支持的调试软件有 ADS、SDT、GreenHill Multi2000、IAR EWARM 等，极大地方便了用户。
(3) 支持 CPU 种类丰富：支持几乎所有含 ARM 内核 CPU, 如 ARM7、ARM9、ARM10、Intel Xscale 等。
(4) JTAG 口频率可编程，支持各类不同性能目标板。
(5) 开放式接口：支持多个内核串联的 CPU 调试，极大的体现了 JTAG 调试器的优势。
(6) 无需外加电源：ARM_Star 仿真器功耗低，可以直接从目标板取电，和目标板共用一个电源系统，使用及外出携带方便。
(7) 免费升级：由于 ARM_Star 的 FirmWare 是每次上电时直接下载到仿真器的，所以，用户可以随时使用最新版本。

2．仿真器硬件使用方法

2.1 ARM_STAR 仿真器对 PC 机的配置要求

(1) 系统环境：Win95, Win98, Windows Me, Win2000, Windows XP，Windows NT 4.0 及以上版本。
(2) 硬件环境：Pentium 200MHz 以上 CPU，64M 内存，300M 硬盘空间，计算机并行口。

2.2 仿真器的功能框图

(1) DB25 并行接口；用于仿真器跟 PC 机的通信，连接线为普通并口延长线。

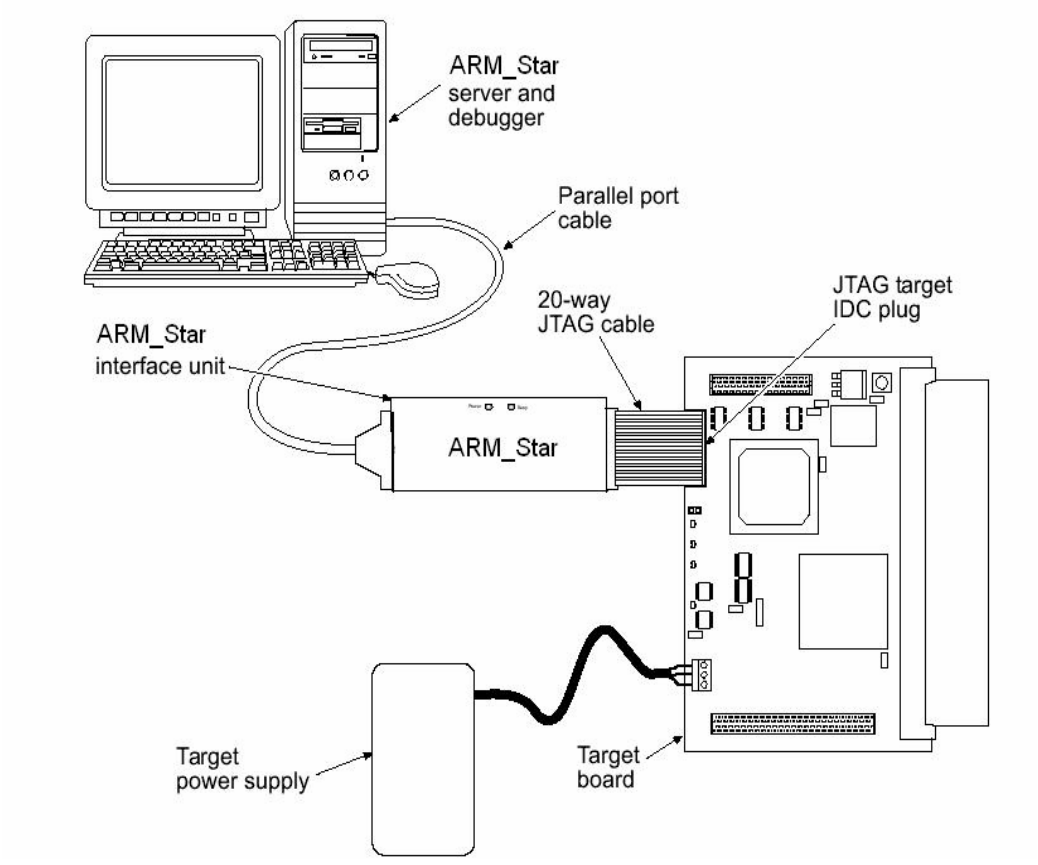
(2) 20 针 IDC20　JTAG 接口：用于连接仿真器和目标板的 JTAG 口。

(3) 电源指示灯：当打开目标板电源，并且仿真器与目标板已正确连接时，电源指示灯就会亮起。

(4) 运行指示灯：当仿真器工作时，如：走单步等，运行指示灯闪烁。

(5) JTAG 接口说明：ARM_STAR 默认的 JTAG 接口为 20Pin 标准接口，如果用户目标板的 JTAG 接口是 14Pin 标准，请用附件中的 JTAG 转换模块实现 20Pin 到 14Pin 的转换。

(6) ARM_STAR 与目标板连接示意图



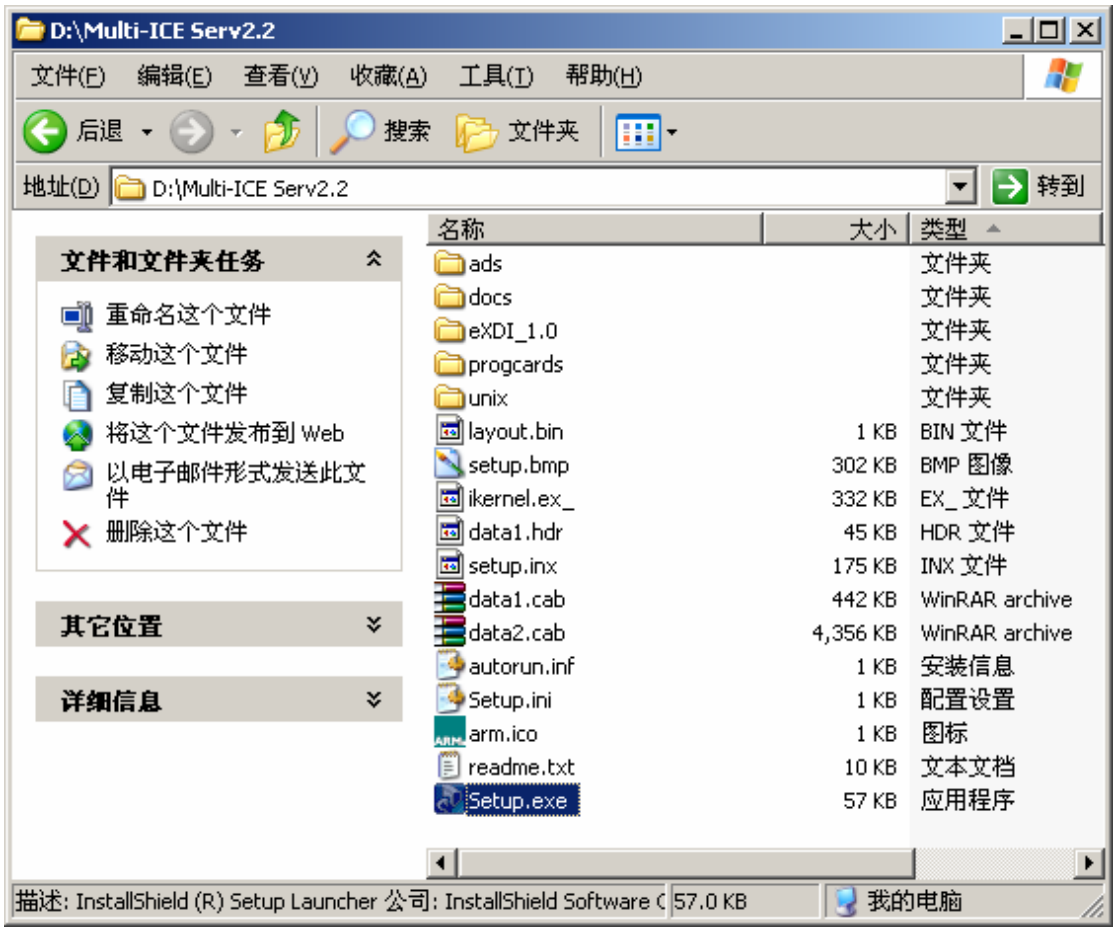
用户首次连接出现问题时，请进入计算机的 BIOS 中更改并口模式。

(7) 电源需求：ARM_Star 仿真器是直接由目标板经过 JTAG 口供电。仿真器 JTAG 口输出信号的电压能自动随目标板电源电压的变化而变化。

3、仿真器软件 Muti-ICE Server 的安装和使用方法

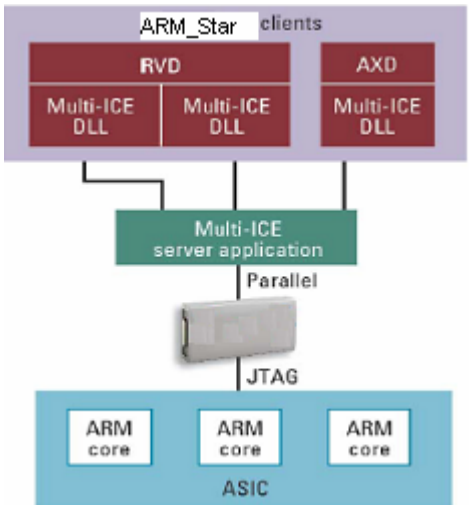
3.1 安装

点击 Setup.exe



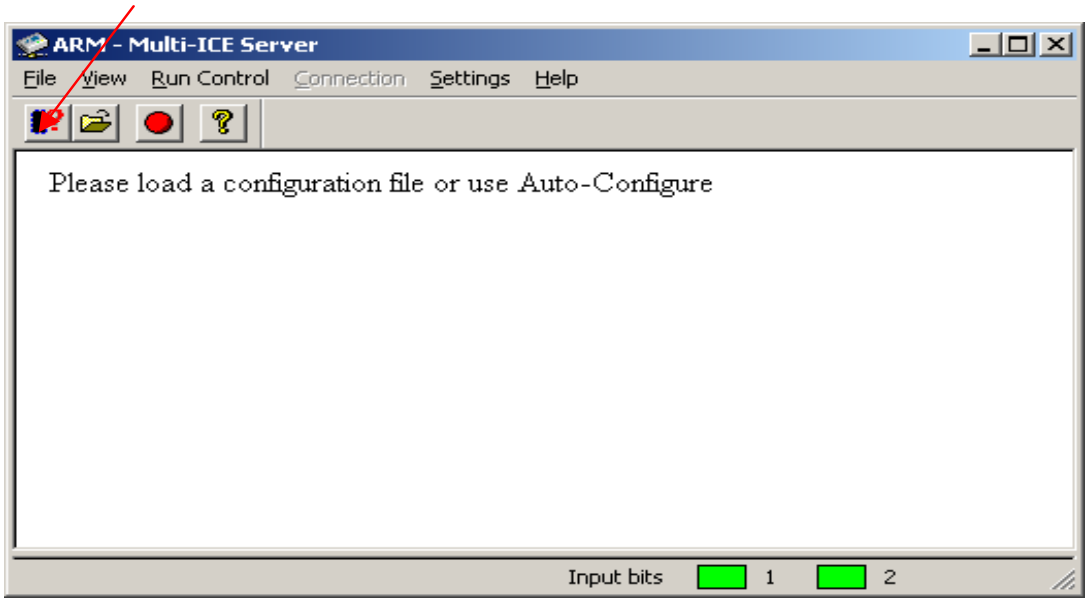
3.2 工作原理介绍

ARM_Star 仿真器是通过 Multi_ICE Server 来实现与其他遵从 RDI 接口标准的调试软件，如 ADS, SDT, GreenHills Multi2000、IAR EWARM 等的连接。



3.3 典型应用

确认仿真器连接正确，然后打开电源，启动 MultiICE-Server。在窗口中点击自动配置图标：



3.4 配置文件的使用方法

一般情况下，使用 Muiti-ICE 的自动配置功能，都可以自动识别目标 CPU 的种类，在有些特殊情况下，仿真器不能自动识别目标 CPU 时，可以使用手动配置的方法实现连接。

首先，编辑一个 rjc2410.cfg 文件，典型内容如下：

[TITLE]
My Cfg File　　；您的配置名称，任意写

[TAP0]　　；目标系统中包含TAP0 控制器
ARM7TDMI　　；　TAP0 控制器上连着一个ARM7TDMI

；如果您的CPU内核是ARM920T，那么您在这
；个位置上就写上ARM920T。

[Timing]　　；JTAG 口TCK信号的时序设置
High = 8　　；TCK 信号的高电平持续时间
Low = 8　　；TCK 信号的低电平持续时间
Adaptive = OFF　　；RTCK 功能开或关（ON or OFF）

[TAPINFO]　　；显示TAP的详细信息
YES

[Reset]　　；复位信号的选择
nTRST

在 Cfg 文件中，[TAP0]表示 CPU 内核的类型，常用的符合语法的内核名称有：

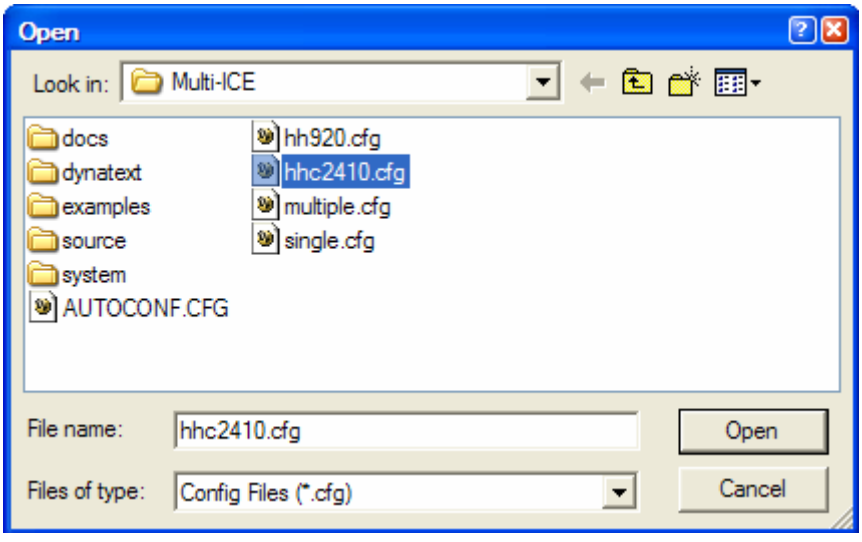
ARM7：ARM7TDMI ARM7TDMI-S ARM720T ARM710T ARM7DMI ARM740T ARM7TDI-S ARM7EJ-S … 等

ARM9：ARM9TDMI ARM920T ARM940T ARM922T ARM9E-S ARM946E-S ARM966E-S ARM926EJ-S …等

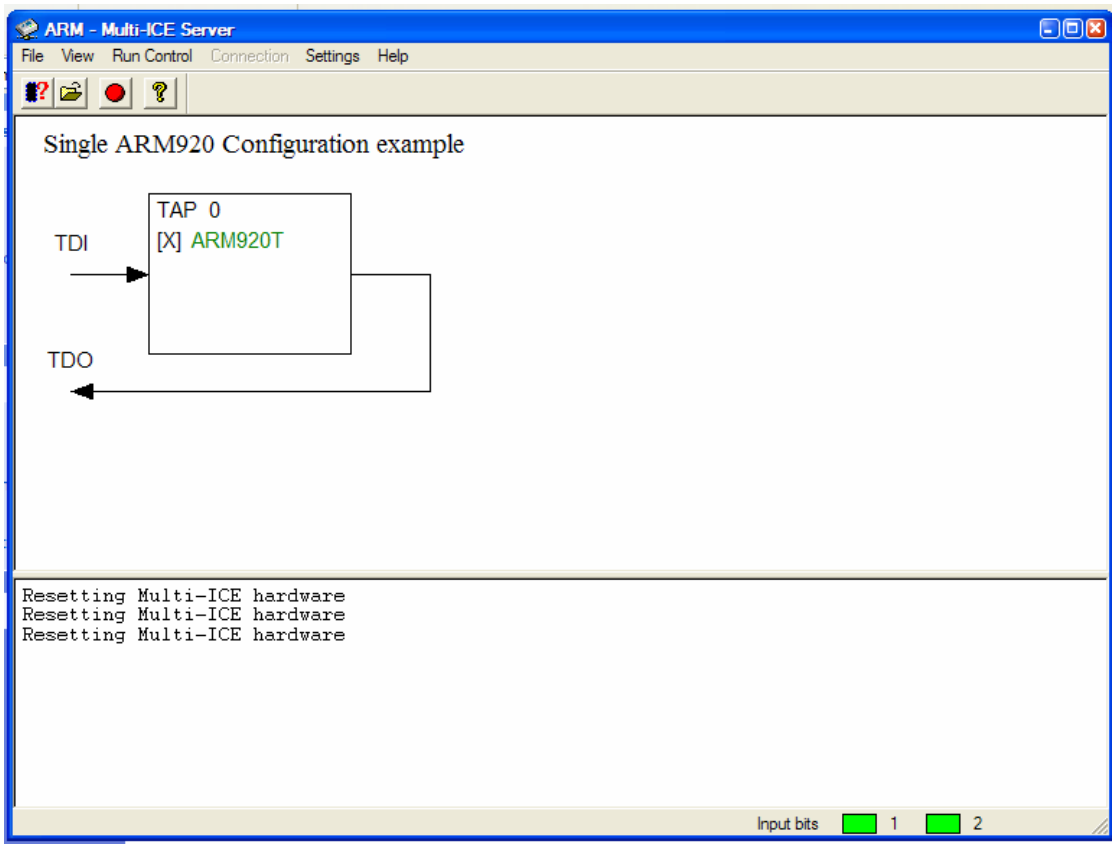
ARM10：ARM1020E ARM1022E ARM1020T … 等

Intel® XScale™ ：XScale-80321 XScale-PXA210 XScale-PXA250 XScale-80200 XScale-IXP425 XScale-IXP2400 XScale-IXP2800 … 等

在 File 菜单下选择 Load Configuration 按钮,会出现一个提示框,选择 RJC2410.CFG 配置文件：

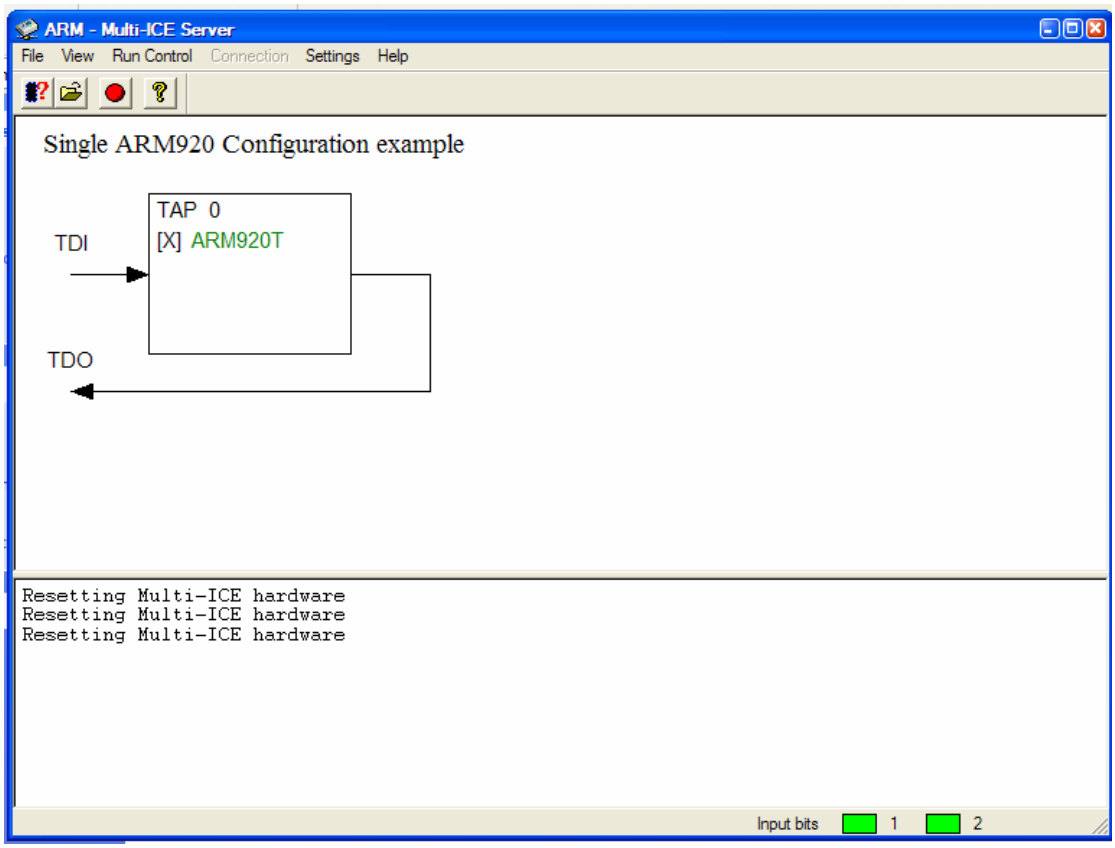


这样，根据用户要求定制的配置就生成了。

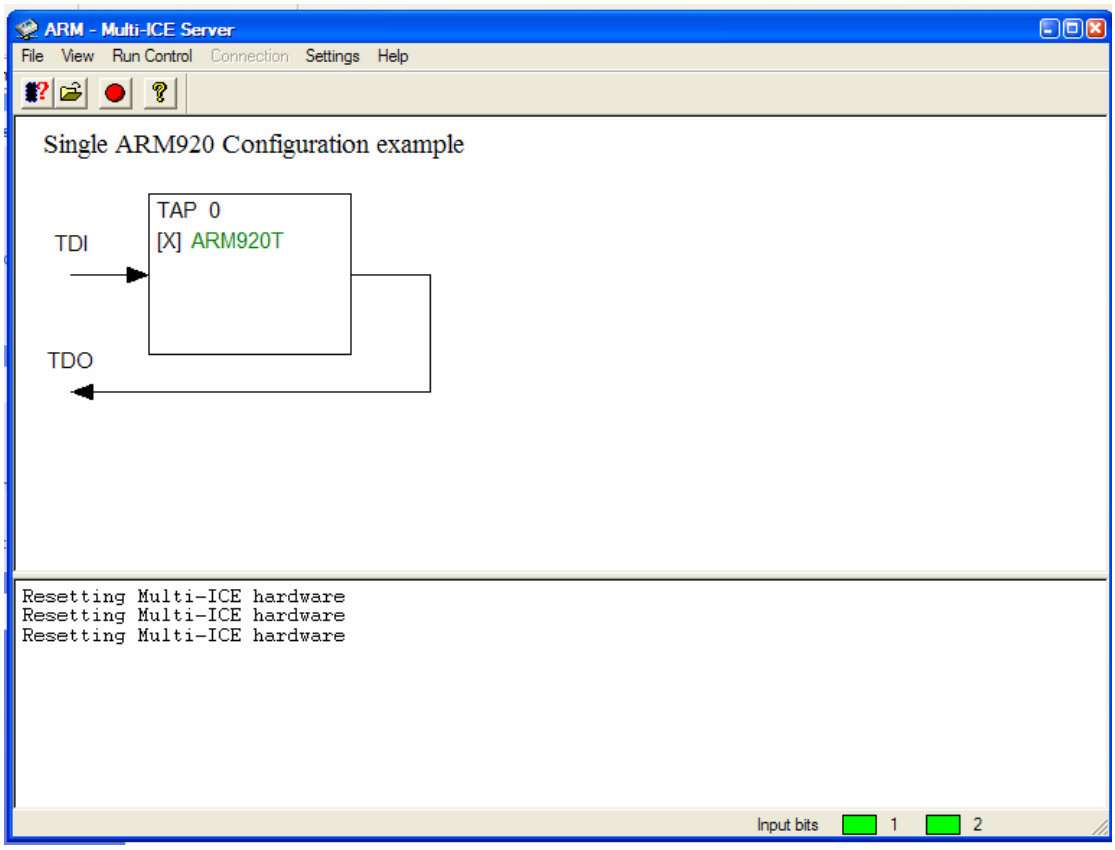


3.5 窗口显示信息

Multi-ICE Srver 的窗口分两部分：CPU 内核信息窗口和调试信息窗口。



在 CPU 内核信息窗口中显示的是目标 CPU 的内核种类、数量、连接状态等信息，如下图中：



自动检测显示目标是一个 CPU 是 ARM920T 的 CPU 的单内核系统，在 TAP 配置显示区中用图形的方法直观的把这一结果显示了出来。内核的类型名称 ARM920T 显示成绿色，并且前面加了一个字母 X。这表示 server 现在还没有连接到任何一个调试程序上去。

处理器类型名称前面的字母叫是状态位，共有四状态位显示：

[S] 处理器处于暂停状态

[R] 处理器忙（运行状态）

[D] 处理器处于下载状态

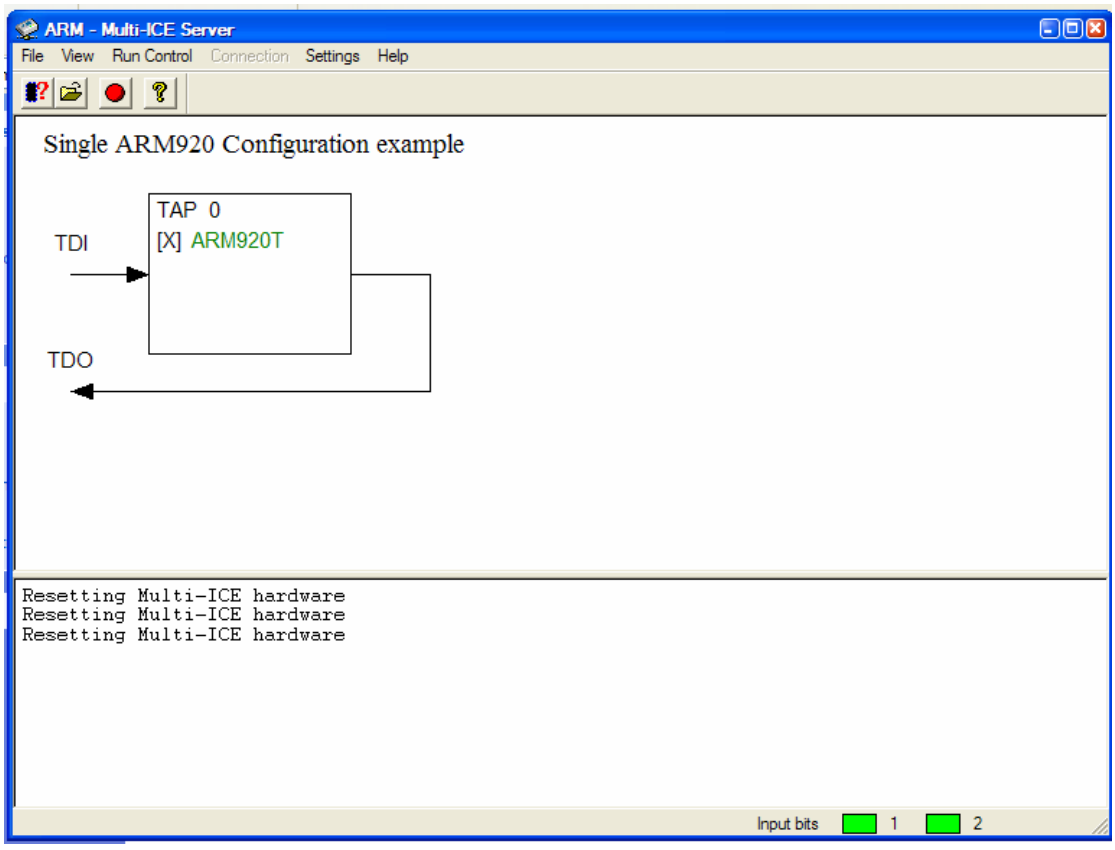
[X] 处理器类型未知或没有被调用

4．仿真器和其它调试环境的连接

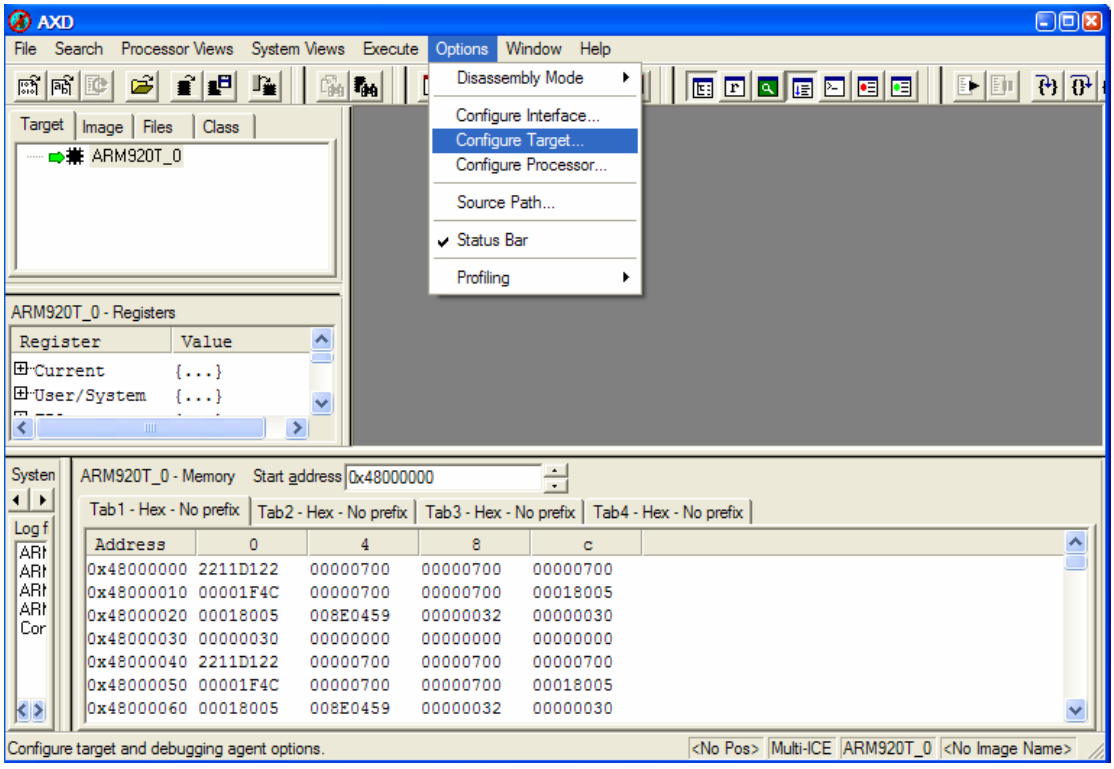
ARM_STAR 可以第三方的 IDE 调试环境配合使用，如 SDT2.5 、ADS1.2、 IAR、GDB 等等，这这里我们主要以 ADS1.2 和 GDB 为实验例子。

4.1 ARM_STAR 仿真器和 ADS1.2 的连接

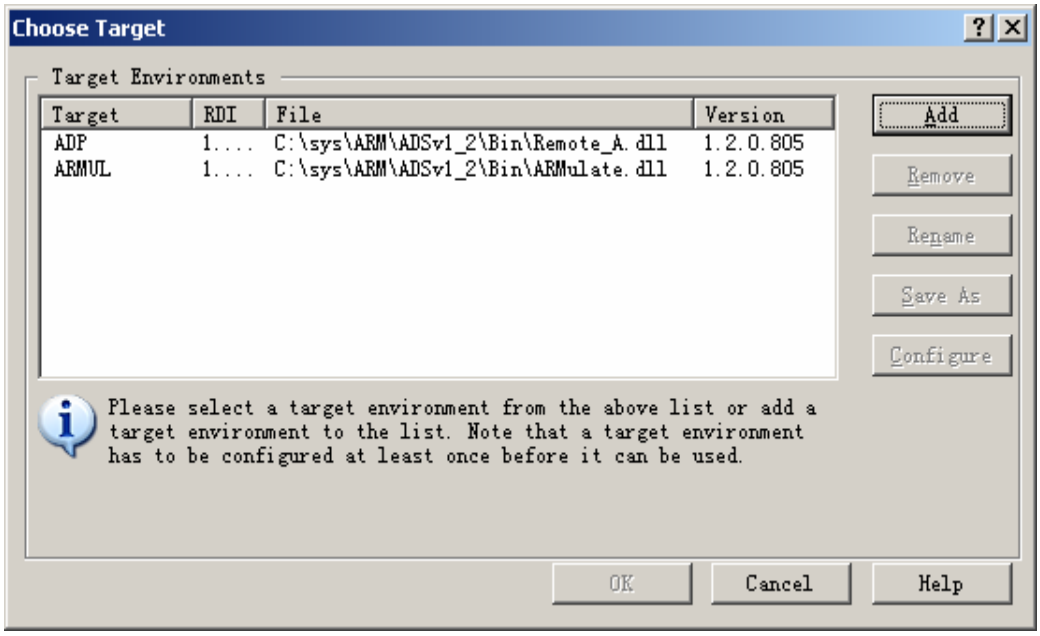
打开 Multi-ICE Server，连接正确后，出现以下画面，不要关闭。

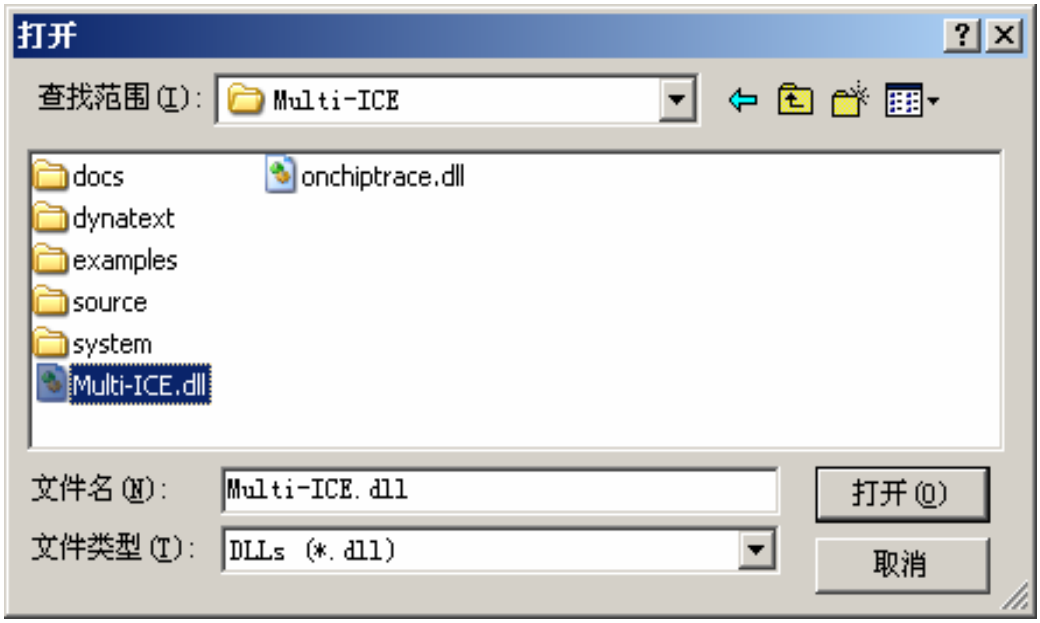


打开 AXD Debugger，在菜单中运行 Option->Config Target：

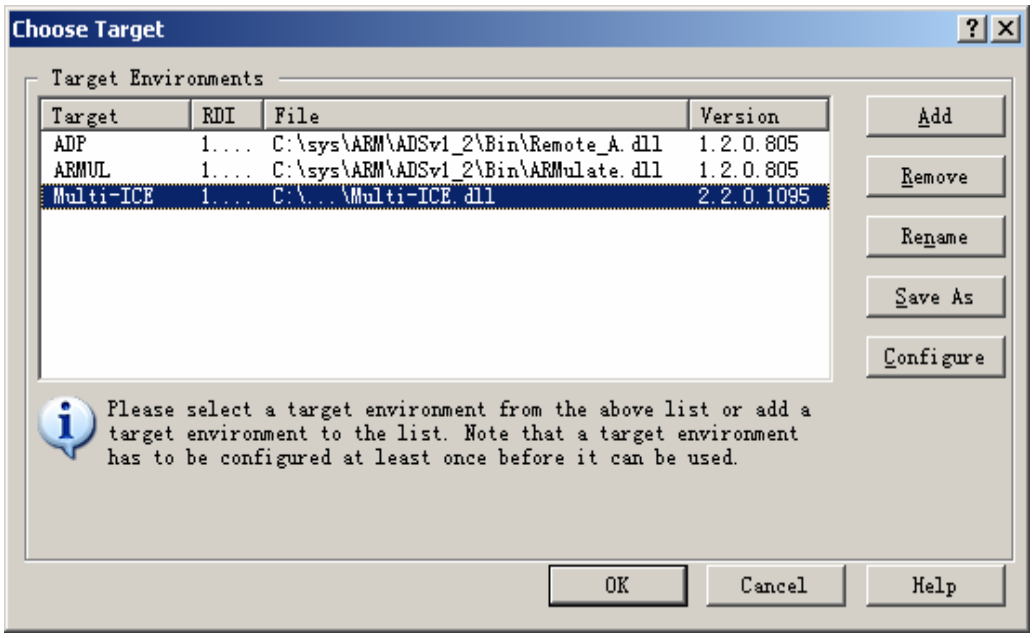


首次使用，需要添加 Multi-ICE.dll 文件。

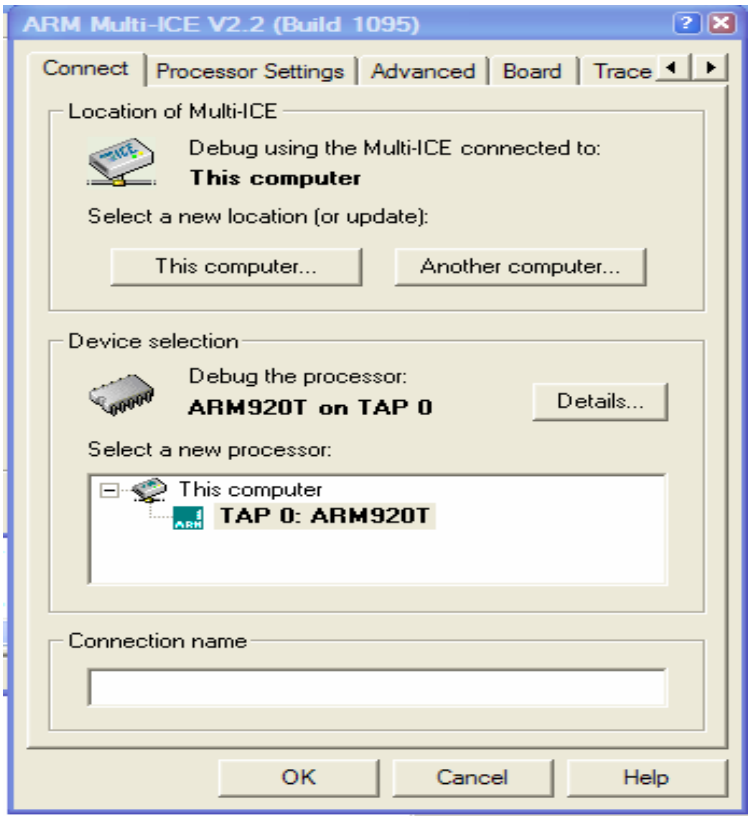




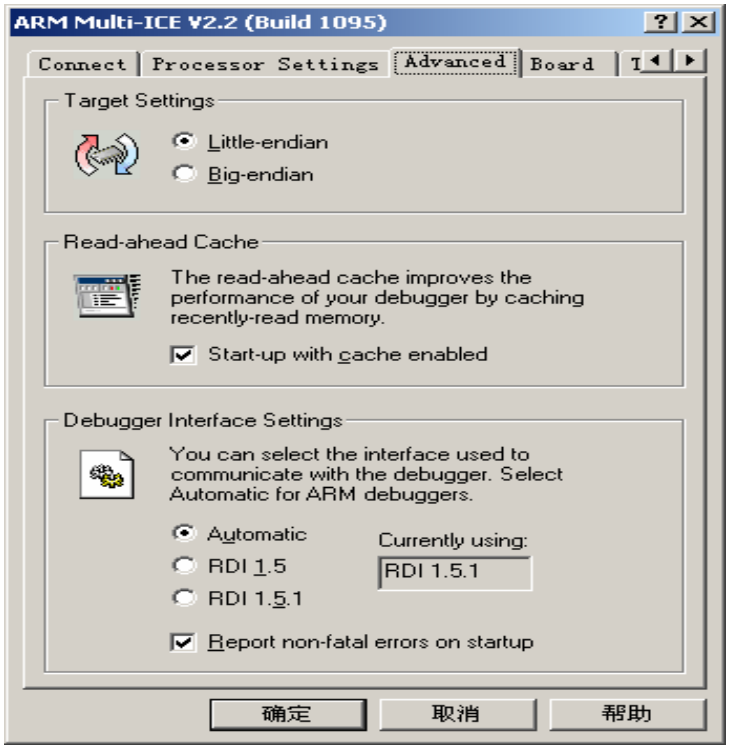
添加后，选择 Configure。



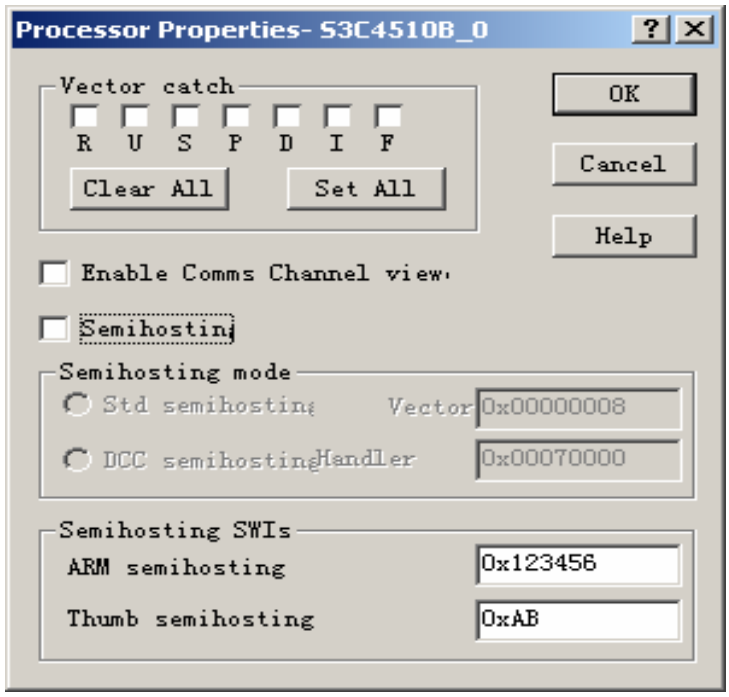
首次 Configure，出现以下画面。



设置存储器的 Big-Endian 或 Little-Endian 模式，在我们这个实验中选择 Little-Endian 模式。



在菜单中打开 Option->Configure Processor…，清除所有选项，以免在发生 Exception 异常时，AXD 打断用户调试。



在用仿真器调试时，需要把程序下载到目标板的 RAM 中，但是因为 ARM CPU 的 Memory 地址等参数需要初始化，所以在调入调试程序前需要运行初始化文件。方法是在 Command Line Interface 窗口中运行批处理文件，例如：obey d:\RJc2410.conf。RJc2410.conf 是 RJARM2410-EDU 实验箱初始化文件，文件的格式如下：

；给 0x48000000 地址写值 0x2211d122

fillmem 0x48000000 0x48000000 0x2211d122　　32

fillmem 0x48000004 0x48000004 0x00000700　32

fillmem 0x4800001c 0x4800001c 0x00018005　32

fillmem 0x48000020 0x48000020 0x00018005　32

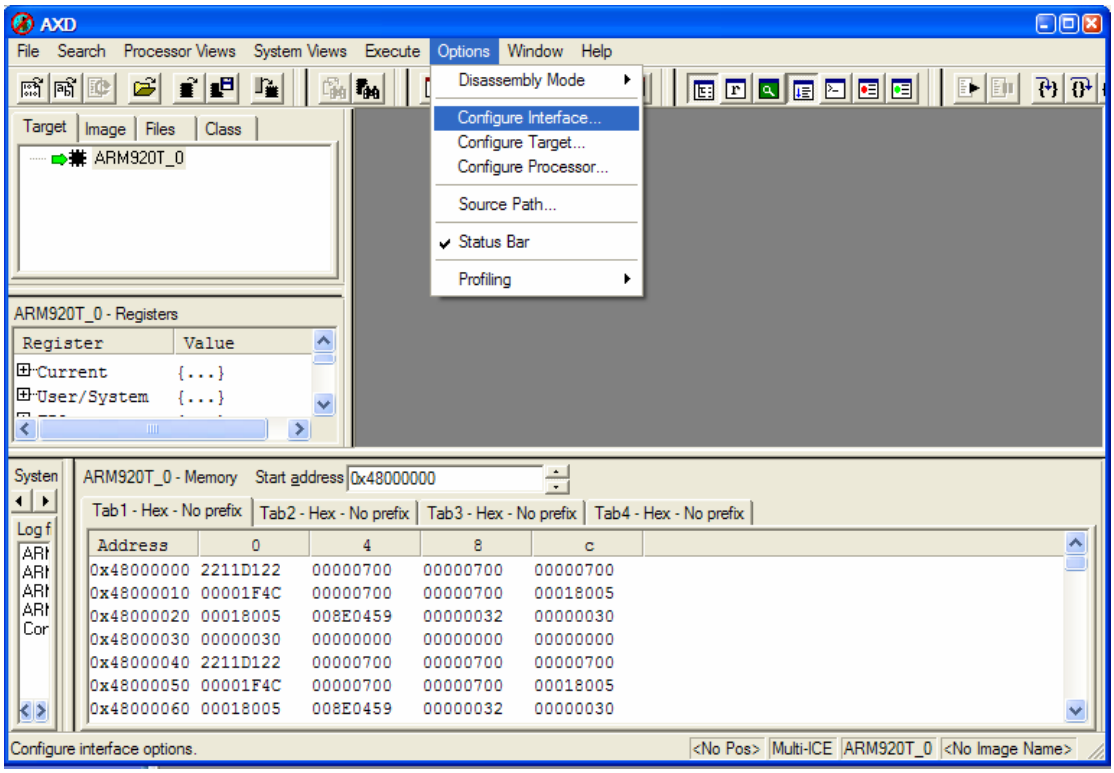
fillmem 0x48000024 0x48000024 0x008e0459　32

fillmem 0x48000028 0x48000028 0x00000032　32

fillmem 0x4800002c 0x4800002c 0x30　　32

fillmem 0x48000030 0x48000030 0x30　　32

在菜单中运行 Option->Command Line Interface：



在弹出的 Command Line Interface 窗口中运行初始化文件。最后，打开 Flie->Load Image… 装载调试文件，进行调试。

4.2 仿真器和 LINUX 下 GDB 的完美无缺的连接

本章节主要是基于 S3C2410 硬件设计上，理解 GDB 工具在嵌入式开发应用中的重要性，及学会在 GDB 调试环境下常用的 GDB 调试命令。

4.2.1 GDB 工具调试应用程序重要性

无论是多么资深的程序员在编写的程序时，都不大可能一次性就会成功，在程序运行时，会出现许许多多意想不到的错误，一味地只是查看程序用处不大，最有效的方法通过一些手段进入到程序内部进行调试。通常在调试程序的时候如果能够得到以下一些信息，对于开发

者找到错误所在是很有帮助的。

(1) 程序是运行到哪个语句或者表达式就发生了错误？

(2) 如果错误是在执行一个函数的时候出现的，那么是程序的哪一行包含了这个函数的调用语句，在调用该函数的时候传递的实参是什么？

(3) 在程序执行到某处时，所关心的某一个变量值为多少？

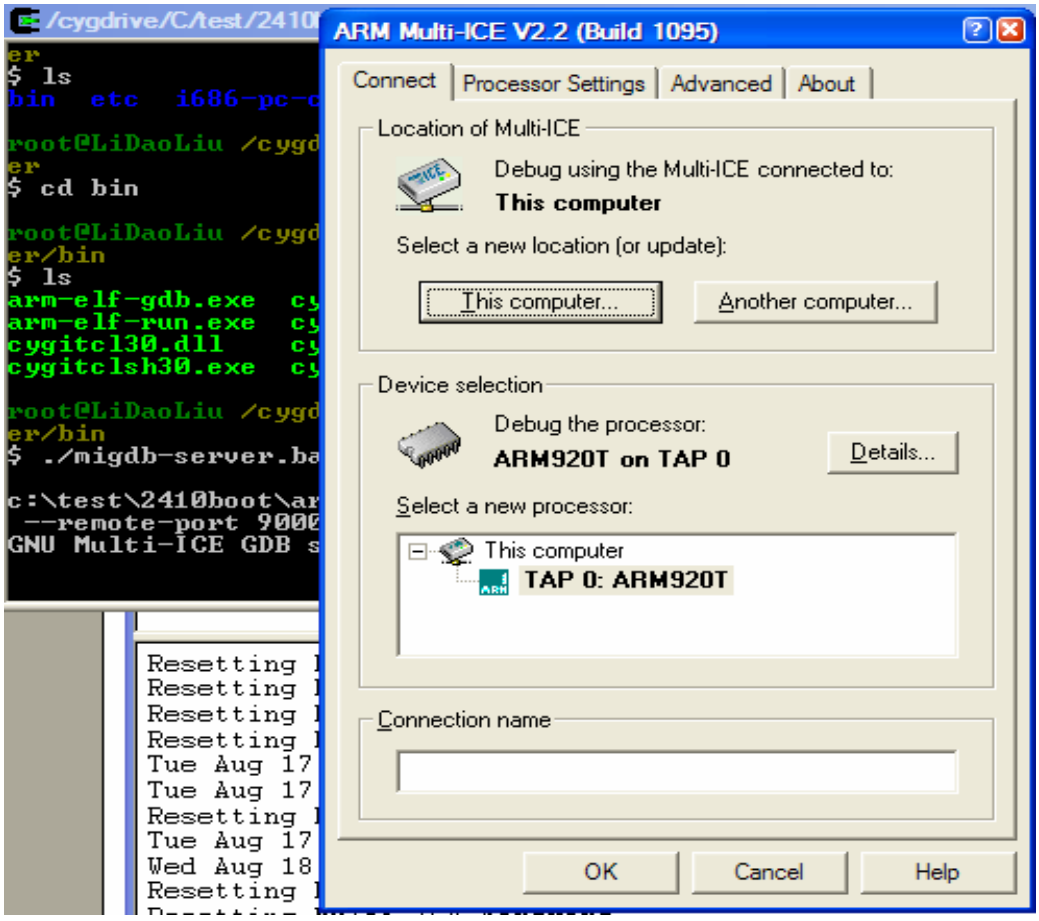
(4) 某个表达式最终运行的结果为何值？

调试器(更准确地说应该称为符号调试器)能够完成上述目标。它是一个能够运行其他程序的应用程序，它和普通意义上的程序的唯一不同之处在于，调试器能够进入到程序源码中，允许开发者进行逐行单步运行，了解程序代码执行顺序，和每条语句执行的结果，可以在程序运行的同时，查看甚至是改变任一变量值。在程序运行出错时，它为程序开发者提供程序运行时的详细细节，从而找到出错的原因。在 Linux 系统中，最常用到的就是 GDB(GNU Degugger)。GDB 是 GNU 自带的调试工具。

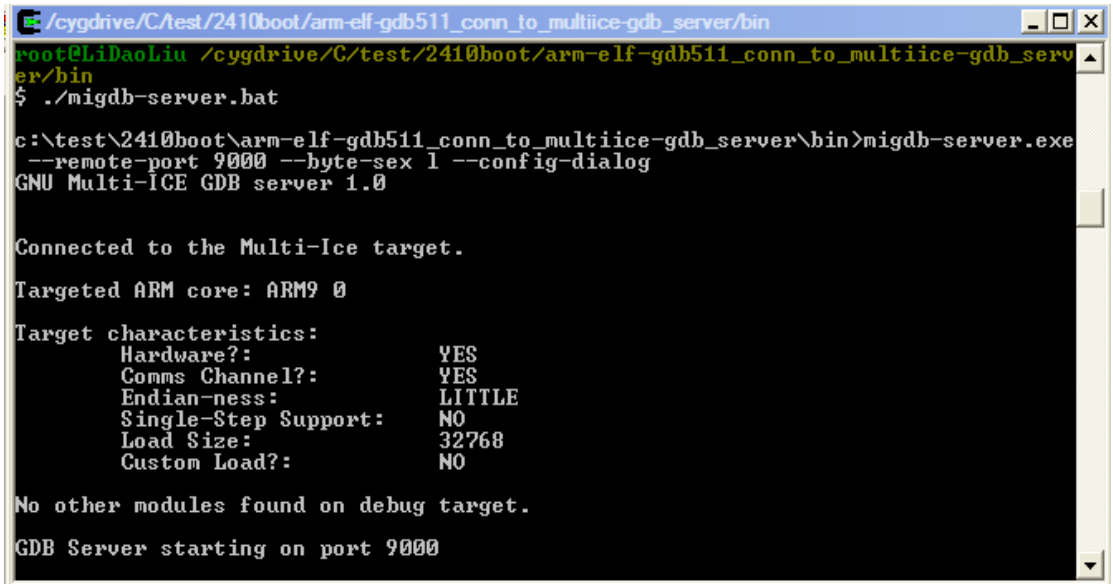
4.2.2 GDB 调试过程

下面以 ARM　STAR 方真器为例子，详细讲述 GDB 调试过程。

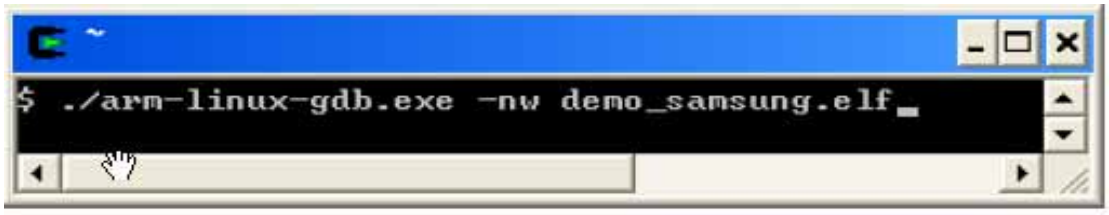
首先要在你的装好 WINDOWS 的 PC 机上装 CYGWIN 模拟 LINUX 的环境，确认仿真器连接正确，然后打开电源，启动 MultiICE-Server。在 CYGWIN 输入./migdb-server.bat 批处理，会出现配置 ARM　STAR 对话框。



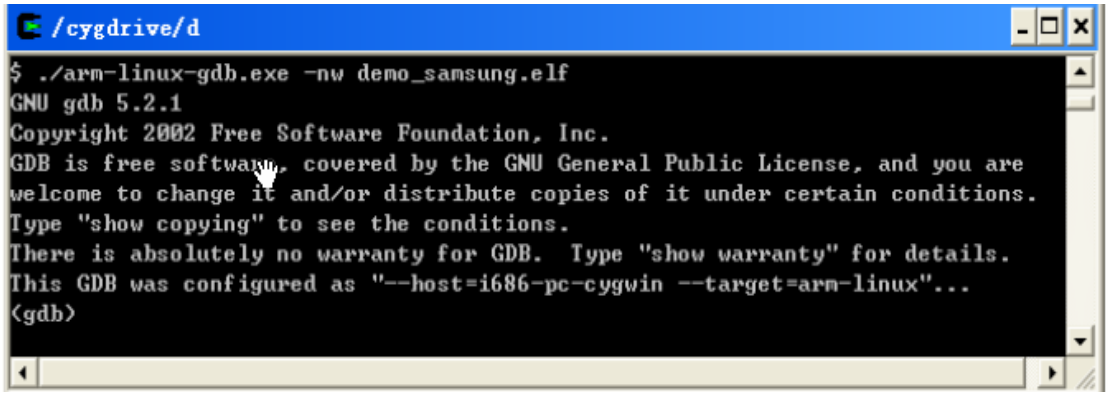
然后单击 OK 按钮，操作结果如下，至此 GDB SERVER 已经启动。



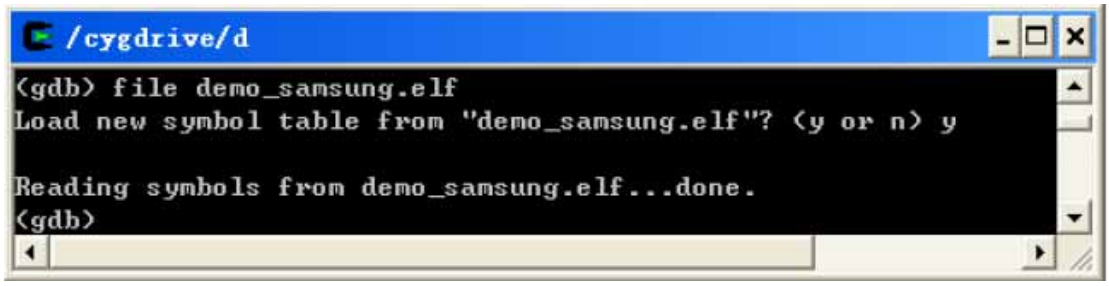
再开个 CYGWIN 窗口，启动客户端程序，其中 ARM-LINX-GDB 是 LINUX 下客户端程序，DEMO_SAMSUNG.ELF 是我们要调试的应用程序，命令格式如下：



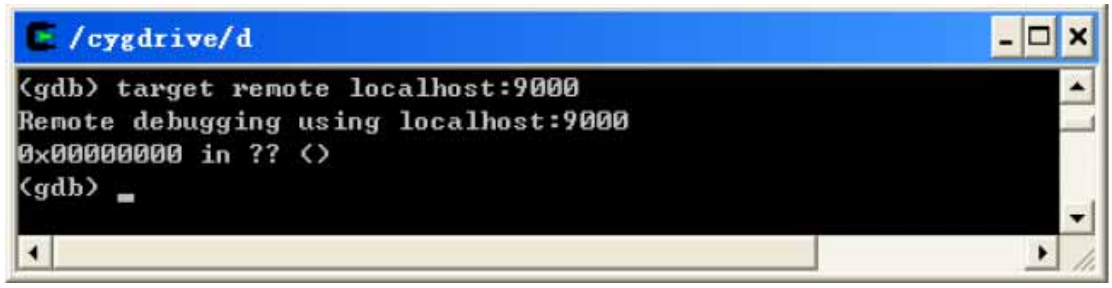
执行该命令的结果显示如下：



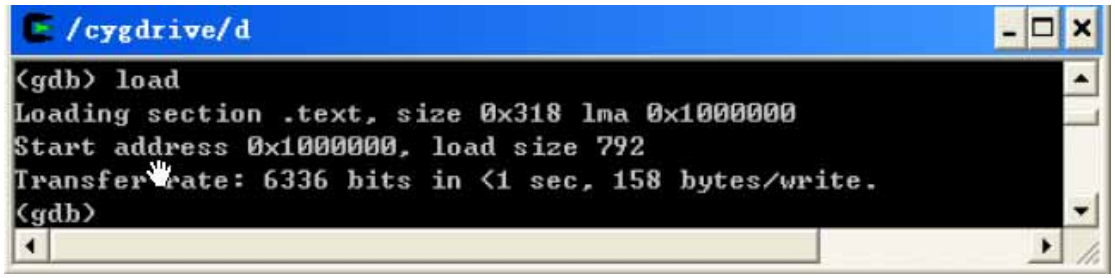
现在就可以用 GDB 提供的丰富多彩的命令啦！有时候使用者仅仅是在 linux 的 bash 提示符下输入命令 gdb 后，启动了 gdb 而已，此时，如果要加载可执行文件，需要在 gdb 提示符下键入命令：file filename(filename 为可执行文件名)，注意是可执行文件的名字而不是源文件名。



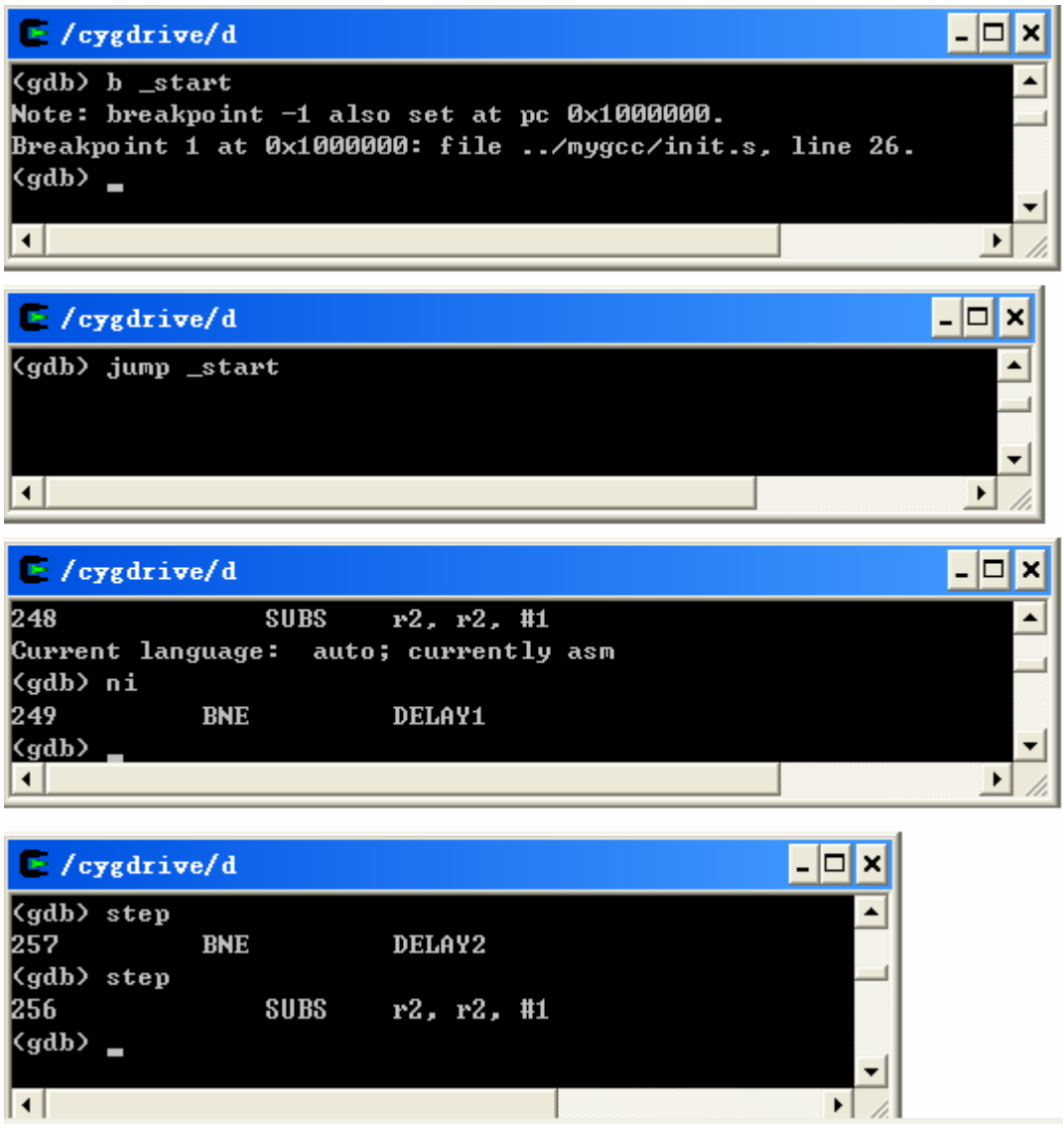
用下列命令联机到 MULTI SERVER ，其中 LOCALHOST 是本机器名字，9000 是 TCP 端口号。



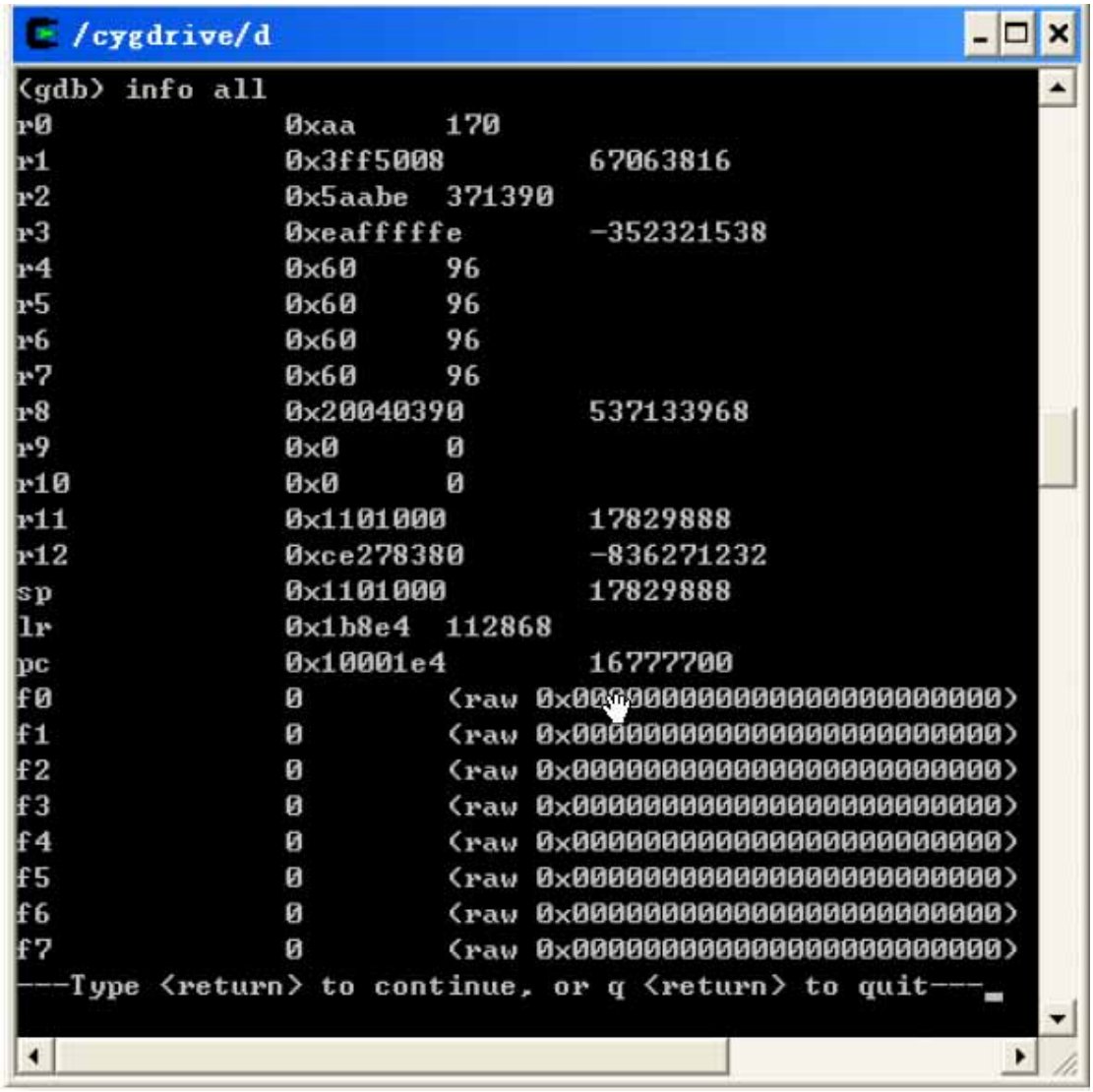
装载命令是对你要调试的文件进行符号下载，这样一来就可以访问内部寄存器。



下面几条指令分别是跳转 、单步指令执行情况：



以下命令是查看我们已经下载程序的一些符号信息。



至此我们一个简单的 GDB 的调试步骤基本完成。要想使用 gdb，必须在对源码进行编译的时候，使用-g 编译选项开关，来通知编译器，开发者希望进行程序调试。用了-g 选项后，程序在编译的时候就会包含调试信息，这些调试信息存在目标文件中，它描述了每个函数或变量的数据类型以及源码行号和可执行代码地址间对应关系，gdb 正是通过这些信息使源码和机器码相关联的，它实现了源码级的调试。

三、实验内容和步骤

1. 熟悉仿真器软件 Muti-ICE Server。按照原理 3 中的方法安装该软件，并按照自己的要求定制配置文件。
2. 熟悉 ADS1.2 环境。按照原理 4.1 节中的步骤连接和配置好仿真器和 ADS1.2。编写一段小程序，装载并调试该程序。
3. 学习 GDB 调试过程。连接仿真器和 GDB，用步骤 2 中的小程序练习联机，装载，跳转，单步执行等命令。步骤可参考原理 4.2.2 节。

附录 A： 常用 LINUX 命令

以下均以 REDHAT LINUX 为例说明。

1) 基本命令：

ls：显示当前目录下的所有文件和目录。

ls –a ：可以看到隐藏的文件，如以.开头的文件。

pwd：显示当前目录路径。

ps：列举当前 TTY 下所有进程

ps –A：列举所有

cd 目录名：进入目录

mkdir 目录名：创建目录

rmdir 目录名：删除空目录

rm –rf 目录名：强行删除整个目录内容（无法恢复），其中 f 表示强制不进行提示，r 表示目录递归。

2) LINUX 下的文件和目录是区分大小写的。

3) TAB 文件目录匹配搜索的使用

例如锐极软件安装的目录为：/RUIJIARM9-EDU，假设/目录下没有其它以 RUIJI 字符开头的其它目录和文件，则要进入这个目录，只需敲入：

cd /RUIJI

然后按下 TAB 键，则 SHELL 会自动匹配找到 RUIJIARM9-EDU 目录，这样就不必完全键入剩余的字符，这个功能在访问名字很长的文件和目录时非常有效，可以大大提供键盘输入的速度，极为方便。

4) ncftp 工具的使用：

ncftp 是 LINUX 下非常好的 FTP 工具软件，它除了支持 FTP 命令操作外，还支持 LINUX SHELL 下的命令用法，例如，它也支持 TAB 键用法，支持目录上传和下载（用-r 或-R 参数）。

ncftp 的用法，例如要 FTP 一台 IP 为 192.168.2.32 的 LINUX PC 机 A，命令如下：

ncftp –u hhcn 192.168.2.32

其中 rjcn 为 A 机器上的合法的用户，连接上之后会提示输入 rjcn 用户的密码，密码验证通过后，就进入 ncftp 命令提示符。

5) 编程时获取帮助 man（类似于 VC 编程中的 MSDN）

man，即 manunal，是 UNIX 系统手册的电子版本。根据习惯，UNIX 系统手册通常分为不同的部分（或小节，即 section），每个小节阐述不同的系统内容。目前的小节划分如下：

1. 命令：普通用户命令
2. 系统调用：内核接口
3. 函数库调用：普通函数库中的函数
4. 特殊文件：/dev 目录中的特殊文件
5. 文件格式和约定：/etc/passwd 等文件的格式
6. 游戏。
7. 杂项和约定：标准文件系统布局、手册页结构等杂项内容
8. 系统管理命令。
9. 内核例程：非标准的手册小节。

手册页一般保存在 /usr/man 目录下，其中每个子目录（如 man1, man2, ..., manl,

mann）包含不同的手册小节。使用 man 命令查看手册页。

常用 man 命令行：

man strtoul

6) 取消 root 密码：

vim /etc/shadow

可以看到第一行内容大致如下：

root:1dVVd5YVP$OgZG58TL/NRExTfcr6URH.:11829:0:99999:7:-1:-1:134539236

要取消 root 密码，只需将第一行 root 后第一对:之间的字符全部删除即可，删除后如下：

root::11829:0:99999:7:-1:-1:134539236

然后用:w!强行存盘（因为 shadow 文件是只读的）后用:q 退出 vi 则实现取消了 root 密码。

7) 修改 PC 机 IP 地址：

ifconfig eth0 192.168.2.32

8) 压缩/解压缩：

LINUX 的软件一般是以.gz 或.tar 或者.tar.gz 结尾的。前者是由 gzip 压缩的，后者是先用 tar 归档，在用 gzip 压缩而成的。

1、以.gz 结尾的为压缩文件，用命令：gzip -d filename 来解压，得到的文件在当前目录中，但已没有了.gz。

2、以.tar 结尾的为归档文件，用命令：tar -xvf filename 来展开，生成的文件与源文件在同一目录中，只是少了.tar。

3、以.tar.gz 结尾的文件最常见，可直接用命令：gzip -cd filename | tar xfv 来解开。

tar 的用法：

解压：x 参数表示解压

tar xzf hhco5249-r1.tgz

把一个目录 HHCO5249-R1 压缩成一个文件：hhco5249-r1.tgz

tar czf hhco5249-r1.tgz HHCO5249-R1

c 参数表示压缩。

9) 查找文件，如：main.c：

find –name main.c

或者：

locate shadow

注意：locate 为模糊匹配，它会递归的在当前目录下的所有子目录下搜索，并列出所有名字包含 shadow 字串的文件。

10) 在一个目录下（含子目录）的所有文件中查找含有某个字符串（如“Modified by hhcn”）的所有文件：

grep ‘Modified by hhcn’ * -r

11) vi(m)用法

vi 是 Linux/Unix 世界里极为普遍的全屏幕文本编辑器，几乎可以说任何一台 Linux/Unix 机器都会提供这个软件。

vi 有三种状态，即编辑方式、插入方式和命令方式。

- 在命令方式下，所有命令都要以:开始，所键入的字符系统均作命令来处理，如:q 代表退出，:w 表示存盘。

- 当你进入 vi 时，会首先进入命令方式（同时也是编辑方式）。按下 i 就进入插入方式，用户输入的可视字符都添加到文件中，显示在屏幕上。按下 ESC 就可以回到命令状态（同时也是编辑方式）。
- 编辑方式和命令方式类似，都是要输入命令，但它的命令不要以:开始，它直接接受键盘输入的单字符或组合字符命令，例如直接按下 u 就表示取消上一次对文件的修改，相当于 WINDOWS 下的 Undo 操作。编译方式下有一些命令是要以/开始的，例如查找字符串就是：/string 则在文件中匹配查找 string 字符串。在编辑模式下按下:就进入命令方式。

基本命令解释：

1．光标命令

k、j、h、l——上、下、左、右光标移动命令。虽然您可以在 Linux 中使用键盘右边的 4 个光标键，但是记住这 4 个命令还是非常有用的。这 4 个键正是右手在键盘上放置的基本位置。

nG——跳转命令。n 为行数，该命令立即使光标跳到指定行。

Ctrl+G——光标所在位置的行数和列数报告。

w、b——使光标向前或向后跳过一个单词。

2．编辑命令

i、a、r——在光标的前、后以及所在处插入字符命令(i=insert、a=append、r=replace)。

cw、dw——改变(置换)/删除光标所在处的单词的命令 (c=change、d=delete)。

x、d$、dd——删除一个字符、删除光标所在处到行尾的所有字符以及删除整行的命令。

3．查找命令

---- /string、?string——从光标所在处向后或向前查找相应的字符串的命令。

4．拷贝复制命令

---- yy、p——拷贝一行到剪贴板或取出剪贴板中内容的命令。

常用操作：

无论是开启新档或修改旧文件，都可以使用 vi，所需指令为：

$ vi filemane

如果文件是新的，就会在荧幕底部看到一个信息，告诉用户正在创建新文件。如果文件早已存在，vi 则会显示文件的首廿四行，　　用户可再用光标（cursor）上下移动。

～

～

上面是一个经 vi 开启的模拟文件，一行开始处的波折号（～）表示文件的结尾。

—指令 i 在光标处插入正文

—指令 I 在一行开始处插入正文

—指令 a 在光标後追加正文

—指令 A 在行尾追加正文

—指令 o 在光标下面新开一行

—指令 O 在光标上面新开一行

在插入方式下，不能打入指令，必需先按〈Esc〉键，返回命令方式。假若户不知身处何态，也可以按〈Esc〉键，不管处於何态，都会返回命令方式。

在修改文件时，如何存档及退出指定文件都非常重要。在 vi 内，行使存档或退出的指令时，要先按冒号（:），改变为命令方式，用户就可以看见在荧幕左下方，出现冒号（:），显示 vi 已经改为指　　令态，可以进行存档或退出等工作。

:q!放弃任何改动而退出 vi，也就是强行退出

:w 存档

:w! 对于只读文件强行存档

:wq 存档并退出 vi

:x 与 wq 的工作一样

:zz 与 wq 的工作一样删除正文

删除或修改正文都是利用编辑方式，故此，下面所提及的指令只需在编辑方式下，直接键入指令即行。

—x 删除光标处字符（Character）

—nx 删除光标处後 n 个字符

—nX 删除光标处前 n 个字符

—ndw 删除光标处下 n 个单词（word）

—dd 删除整行

—d$或 D 删除由光标至该行最末

—u 恢复前一次所做的删除

当使用 vi 修改正文，加减字符时，就会采用另一组在编辑方式下操作的指令。

—　r　char 由 char 代替光标处的字符

—Rtext〈Esc〉由 text 代替光标处的字符

—cwtext〈Esc〉由 text 取代光标处的单词

—Ctext〈Esc〉由 text 取代光标处至该行结尾处

—cc 使整行空白，但保留光标位置，让你开始打入

—如删除指令一样，在指令前打入的数，表示执行该指令多少次。

要检索文件，必需在编辑方式下进行。

—/str〈Return〉向前搜寻 str 直至文件结尾处

—?str〈Return〉往後搜寻 str 直至文件开首处

—n 同一方向上重复检索

—N 相反方向上重复检索

—vi 缠绕整个文件，不断检索，直至找到与模式相匹配的下一个出现。

全程替换命令：

:%s/string1/string2/g 在整个文件中替换“string1”成“string2”。

如果要替换文件中的路径：

使用命令“:%s#/usr/bin#/bin#g”可以把文件中所有路径/usr/bin 换成/bin。也可以使用命令“:%s/\/usr\/bin/\/bin/g”实现，其中“\”是转义字符，表明其后的“/”字符是具有实际意义的字符，不是分隔符。

同时编辑 2 个文件，拷贝一个文件中的文本并粘贴到另一个文件中：

命令如下：

---- vi file1 file2

---- yy 在文件 1 的光标处拷贝所在行

---- :n 切换到文件 2 (n=next) 或者按 ctrl+ww，就在两个文件间切换。

---- p 在文件 2 的光标所在处粘贴所拷贝的行

---- :n 切换回文件 1

将文件中的某一部分修改保存到临时文件，例如仅仅把第 20～59 行之间的内容存盘成文件/tmp/1，我们可以键入如下命令。

---- vi file

---- :20,59w /tmp/1

如果要在 vi 执行期间，转到 shell 执行，使用惊叹号（!）执行系统指令，例如在 vi 期间，列出当前目录内容，可以键入 ：

:!ls

另一方面，用户可以在主目录中创建.exrc 环境文件，用 set 打入选项，每次调用 vi 时，就会读入.exrc 中的指令与设置。下面是.exrc 环境文件的实例：

set wrapmarging＝8

set showmode

set autoindent

12) minicom 用法

minicom 是安装 REDHAT 时安装的软件，它使用配置文件/etc/minirc.dfl，锐极光盘安装时会提供这个文件。

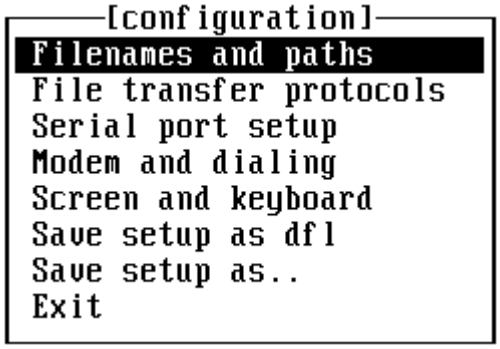
【注意】

minicom 占用串口，能且仅能启动一个 minicom，启动第二个时就会报错：**Device /dev/modem is locked**。其中/dev/modem 就是/dev/ttyS0，即 PC 机串口 1，它是在光盘安装时执行./ucinst 时创建的链接。查看 ucinst 文件，可以看到如下一行：

ln -sf /dev/ttyS0 /dev/modem

minicom 所有的操作都以 ctrl+A 开始，例如：退出为 ctrl+A，松手后再按下 Q，则弹出如下一个小框：选 Yes 即可退出 minicom。

minicom 中最重要的操作就是对其进行配置的修改。这个操作要先 ctrl+A,松手后按下 O，则弹出如下框：



选择第三项“Serial port setup”，则弹出下面框：

```
A -    Serial Device      : /dev/modem
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 19200 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No

   Change which setting?
```

键入 E 则弹出如下框，可改变波特率：

```
                [Comm Parameters]

Current: 19200 8N1

  Speed              Parity             Data

A: 300               L: None            S: 5
B: 1200              M: Even            T: 6
C: 2400              N: Odd             U: 7
D: 4800              O: Mark            V: 8
E: 9600              P: Space
F: 19200
G: 38400
H: 57600
I: 115200            Q: 8-N-1
J: 230400            R: 7-E-1


Choice, or <Enter> to exit?
```

若要使用 PC 机的串口 2 来接板子的串口 1 做监控，则要在串口配置框中选择 A，即“Serial Device”，则原来的配置框第一行进入编辑模式，将原来的/dev/modem 改为如下的：/dev/ttyS1，即串口 2。

```
A -    Serial Device      : /dev/ttyS1
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 19200 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No

   Change which setting?
```

退出配置框只需连续按 ESC 键即可返回。

13) 软、硬盘及光驱的使用

在 Linux 中对其他硬盘逻辑分区、软盘，光盘的使用与我们通常在 DOS 与 Windows 中的

使用方法是不一样的，不能直接访问，因为在 Linux 中它们都被视为文件，因此在访问使用前必须使用装载命令 mount 将它们装载到系统的/mnt 目录中来，使用结束，必须进行卸载。命令格式如下：

mount –t 文件系统类型 设备名 装载目录

文件类型常用的有：

msdos	dos 分区文件
ext2	Linux 的文件系统
swap	Linux swap 分区或 swap 文件
iso9660	安装 CD-ROM 的文件系统
vfat	支持长文件名的 dos 分区
hpfs	OS/2 分区文件系统

设备名是指要装载的设备的名称，如软盘、硬盘、光盘等，软盘一般为/dev/fd0 fd1，硬盘一般为/dev/hda hdb ，硬盘逻辑分区一般为期 hda1 hda2...等等，光盘一般为/dev/hdc。在装载前一般要在/dev/mnt 目录下建立一个空的目录，如软盘为 floppy，硬盘分区为其盘符如 c、d 等等，光盘为 cd-rom，使用命令：

mount –t msdos /dev/fd0 /mnt/floppy

装载一个 mddos 格式的软盘

mount –t ext2 /dev/fd0 /mnt/floppy

装载一个 Linux 格式的软盘

mount –t vfat /dev/hda1 /mnt/c

装载 Windows98 格式的硬盘分区

mount –t iso9660 /dev/hdc /mnt/cd-rom

装载一个光盘

装载完成之后便可对该目录进行操作，在使用新的软盘及光盘前必须退出该目录，使用卸载命令进行卸载，方可使用新的软盘及光盘，否则系统不会承认该软盘的，光盘在卸载前是不能用光驱面板前的弹出键退出的。

14) LILO 与 GRUB

LINUX 安装时一般都安装 bootloader，可以支持多操作系统并存。常见的 bootloader 有 LILO 和 GRUB，REDHAT6.x 使用 LILO，REDHAT7.2 同时支持 LILO 和 GRUB，但默认使用的是 GRUB。

15) diff 创建软件补丁，用 patch 打补丁

diff 是生成源码补丁的必备工具。其命令格式为：

diff [命令行选项] 原始文件 新文件

常用命令行选项如下：

-r 递归处理目录　　-u 输出统一格式(unified format)

-N patch 里包含新文件　　-a patch 里可以包含二进制文件

它的输出在 stdout 上，所以你可能需要把它重定向到一个文件。

输出格式保存了上下文（缺省是上下各三行，最少需要两行），这样，patch 的时候可以允许行号不精确匹配的情况出现。另外，在 patch 文件的开头明确地用---和+++标示出原始文件和当前文件，也方便阅读。

例如：

diff f1 f2 >r

则将文件 f1 和 f2 的区别之处都记录在 r 文件中。

通常，我们需要对整个软件包做修改，并生成一个 patch 文件，下面是典型的操作过程。

tar xzvf software.tar.gz # 展开原始软件包，其目录为 software

cp _a software software-orig # 做个修改前的备份

cd software

[修改，测试……]

cd ..

diff -ruNa software-orig software > software-my.patch

现在我们就可以保存 software-my.patch 做为这次修改的结果，至于原始软件包，可以不必保存。等到下次需要再修改的时候，可以用 patch 命令把这个补丁打进原始包，再继续工作。比如是在 linux kernel 上做的工作，就不必每次保存几十兆修改后的源码了。这是好处之一，好处之二是维护方便，由于 unified patch 格式有一定的模糊匹配能力，能减少原软件包升级带来的维护工作量。

patch

patch 程序根据补丁（patch）文件修改一个文件。补丁文件通常是使用 diff 程序建立的一个列表，这个列表包含如何修改原始文件的一些指令。由于节省时间和空间，Patch 经常用于源代码的修补。可以想象一个有 1MB 的程序包，这个程序包的下一个版本仅仅改变了前面一个版本的两个文件的内容，这个新版本可以完全以一个 1MB 的新版本进行发布或者以一个仅仅有 1KB 的补丁文件进行发布。这个补丁文件可以对第一个版本的进行更新，更新后的版本就和第二个版本完全一致了。因此，如果已经下载了第一个版本，那么为了下一个版本而进行的大数据量下载工作就可以有效地避免。

常用命令行选项：

patch [命令行选项] [待 patch 的文件[patch]]

-pn patch level(n 是数字) -b[后缀] 生成备份，缺省是.orig

为了说明什么是 patch level，这里看一个 patch 文件的头标记。

diff -ruNa xc.orig/config/cf/Imake.cf xc.bsd/config/cf/Imake.cf

--- xc.orig/config/cf/Imake.cf Fri Jul 30 12:45:47 1999

+++ xc.new/config/cf/Imake.cf Fri Jan 21 13:48:44 2000

这个 patch 如果直接应用，它会去找 xc.orig/config/cf 目录下的 Imake.cf 文件，假如你的源码树的根目录是缺省的 xc 而不是 xc.orig，除了 mv xc xc.orig 之外，有无简单的方法应用此 patch 呢？patch level 就是为此而设：patch 会把目标路径名砍去开头 patch level 个节(由/分开的部分)。在本例中，可以用下述命令：

cd xc; patch -p1 < /pathname/xxx.patch

完成操作。注意，由于没有指定 patch 文件，patch 程序默认从 stdin 读入，所以用了输入重定向。

又例如：

diff -r dir1 dir2 >patch20020523.patch

递归的比较目录 dir1 与 dir2 内，所有各文件之不同处，并将不同处记录到 patch20020523.patch 文件中。

patch -p1 < [patchfile]

-p1 选项代表 patchfile 中文件名左边目录的层数,顶层目录在不同的机器上有所不同。要使用这个选项，就要把你的 patch 放在要被打补丁的目录下，然后在这个目录中运行 path -p1 < [patchfile]。

16) LINUX 下的硬盘分区

对习惯于使用 Dos 或 Windows 的用户来说，有几个分区就有几个驱动器，并且每个分区都会获得一个字母标识符，然后就可以选用这个字母来指定在这个分区上的文件和目录，它们的文件结构都是独立的，非常好理解。但对 Red Hat Linux 用户来说无论有几个分区，分给哪一目录使用，它归根结底就只有一个根目录，一个独立且唯一的文件结构。Red Hat Linux 中每个分区都是用来组成整个文件系统的一部分，因为它采用了一种叫“载入”的处理方法，它的整个文件系统中包含了一整套的文件和目录，且将一个分区和一个目录联系起来。这时要载入的一个分区将使它的存储空间在一个目录下获得。

对于 IDE 硬盘，驱动器标识符为“hdx~”,其中“hd”表明分区所在设备的类型，这里是指 IDE 硬盘了。“x”为盘号（a 为基本盘，b 为基本从属盘，c 为辅助主盘，d 为辅助从属盘）,“~”代表分区，前四个分区用数字 1 到 4 表示，它们是主分区或扩展分区，从 5 开始就是逻辑分区。例，hda3 表示为第一个 IDE 硬盘上的第三个主分区或扩展分区,hdb2 表示为第二个 IDE 硬盘上的第二个主分区或扩展分区。对于 SCSI 硬盘则标识为“sdx~”，SCSI 硬盘是用“sd”来表示分区所在设备的类型的，其余则和 IDE 硬盘的表示方法一样。

从上面可以看到，Red Hat Linux 的分区是不同于其它操作系统分区的，它的分区格式只有 Ext2（3）和 Swap 两种，Ext2（3）用于存放系统文件，Swap 则作为 Red Hat Linux 的交换分区。Red Hat Linux 至少需要两个专门的分区（Linux Native 和 Linux Swap）况且不能将 Red Hat Linux 安装在 Dos/Windows 分区。一般来说将 Red Hat Linux 安装一个或多个类型为“Linux Native”的硬盘分区，但是在 Red Hat Linux 的每一个分区都必须要指定一个“Mount Point”（载入点），告诉 Red Hat Linux 在启动时，这个目录要给哪个目录使用。对“Swap”分区来说,一般定义一个且它不必要定义载入点。

SWAP 分区是 LINUX 暂时存储数据的交换分区，它主要是把主内存上暂时不用的数据存起来，在需要的时候再调进内存内，且作为 SWAP 使用的分区不用指定“Mout Point”（载入点），既然它作为交换分区，我们理所当然应给它指定大小，它至少要等于系统上实际内存的量，一般来说它的大小是内存的两倍，如果你是 16MB 的内存，那么 SWAP 分区的大小是 32MB 左右，以此类推。但必须还要注意一点，SWAP 分区不要大于 128MB，如果你是 64MB 的内存，那么 SWAP 分区最大也只能被定为 127MB，再大就是浪费空间了，因为系统不需要太大的交换分区。以此类推，如果你是 128MB 或更大的内存，SWAP 分区也只能最大被定为 127MB。可以创建和使用一个以上的交换分区，最多 16 个。

Linux Native 是存放系统文件的地方，它只能用 EXT2（3）的分区类型。对 Windows 用户来说，操作系统必须装在同一分区里，它是商业软件吗!所以你没有选择的余地！对 Red Hat Linux 来说，你有了较大的选择余地，你可以把系统文件分几个区来装（必须要说明载入点），也可以就装在同一个分区中（载入点是“/”）。

/boot 分区，它包含了操作系统的内核和在启动系统过程中所要用到的文件，建这个分区是有必要的，因为目前大多数的 PC 机要受到 BIOS 的限制,况且如果有了一个单独的/boot 启动分区，即使主要的根分区出现了问题，计算机依然能够启动。这个分区的大小约在 50MB－100MB 之间。但是如果想用 LILO 启动 Red Hat Linux 系统的话，含有/boot 的分区必须完全在柱面 1023 以下。又由于 8GB 后的数据 LILO 不能读取，所以 Red Hat Linux 要安装在 8GB 的区域以内。

/usr 分区，是 Red Hat Linux 系统存放软件的地方，如有可能应将最大空间分给它。

/home 分区，是用户的 home 目录所在地，这个分区的大小取决于有多少用户。如果是多用户共同使用一台电脑的话，这个分区是完全有必要的，况且根用户也可以很好地控制普通用户使用计算机，如对用户或者用户组实行硬盘限量使用，限制普通用户访问哪些文件等。

其实单用户也有建立这个分区的必要，因为没这个分区的话，那么你只能以 root 用户的身份登陆系统，这样做是危险的，因为 root 用户对系统有绝对的使用权，一旦对系统进行了误操作，就会导致系统崩溃。

/var/log 分区，是系统日志记录分区，如果设立了这一单独的分区，这样即使系统的日志文件出现了问题，它们也不会影响到操作系统的主分区。

/tmp 分区，用来存放临时文件。这对于多用户系统或者网络服务器来说是有必要的。这样即使程序运行时生成大量的临时文件，或者用户对系统进行了错误的操作，文件系统的其它部分仍然是安全的。因为文件系统的这一部分仍然还承受着读写操作，所以它通常会比其它的部分更快地发生问题。

/bin 分区，存放标准系统实用程序。

/dev 分区，存放设备文件。

/opt 分区，存放可选的安装的软件。

/sbin 分区，存放标准系统管理可执行文件，如 insmod，ifconfig 等。

上面介绍了几个常用的分区，一般来说需要一个 SWAP 分区，一个/boot 分区，一个/usr 分区，一个/home 分区，一个/var/log 分区。当然这没有什么规定，完全是依照个人来定的。但记住至少要有两个分区，一个 SWAP 分区，一个/分区。

用户可以使用两种分区工具：

1. Disk Druid：它是 Red Hat Linux 提供的硬盘管理工具，它最初是随 Red HatLinux5 一起发售的，它可以根据用户的要求创建和删除硬盘分区，另外还可以为每个分区管理载入点，这是一个不错的分区软件，建议使用。

2. Fdisk：它是传统的 Linux 硬盘分区工具，比 Disk Druid 更强大，使用更加灵活。但是 Fdisk 要求用户对硬盘分区有一定经验，并能够适应且读懂简单的文本界面。如果你是第一次对一个硬盘驱动器进行分区操作的话，最好还是避免 Fdisk 这样的程序，它虽然强大但用起来的感觉不是太好的。

附录 B：　参考书目

A)　《UNIX 环境高级编程》，W.Richard Stevens 著，机械工业出版社

B)　《LINUX 设备驱动程序》，ALESSANDRO RUBINI 著，LISOLEG 译，中国电力出版社

C)　《嵌入式 LINUX 设计与应用》（清华大学出版社 2002 年出版）

D)　S3C2410X 32-bit MICROPROCESSOR User’s Manual

E)　《ARM 体系结构与编程》杜春雷编著，清华大学出版社出版