源码开放学ARM

@亚嵌李明老师[1]

2012-12-04

前言

学习目标

ARM 阶段的学习，构成了嵌入式软件开发工程师知识体系中不可缺少的一个内容。在这个阶段我们努力培养学员具备以下的素质和能力：

* 掌握ARM体系结构和汇编语言。
* 能够初步学会阅读硬件原理图和芯片数据手册。
* 具备为SoC芯片常见外设如 UART，NandFlash，Timer 等编写驱动程序的能力。
* 能够完成 Bootloader 项目的程序编写和移植工作。

适合对象

本阶段对于学员学习基础的要求如下：

* 掌握C语言，熟悉指针的用法。
* 学过计算机组成原理和数字电路等课程。
* 具备一定的英文阅读能力。
* 对计算机底层的运行机制和软硬件协同工作具有浓厚的兴趣。

如何写作本书的

对于有志于参与本书编写的学员，可以通过学习以下内容来进一步了解关于如何写书的相关知识。

如何安装 GIT

* http://progit.org/book/zh/ch1-4.html

如何使用 GitHub

* http://www.worldhello.net/gotgithub/index.html

如何用 markdown 写书

* http://www.slideshare.net/larrycai/write-book-in-markdown

如何生成 pdf 版本

* http://github.com/larrycai/kaiyuanbook/blob/master/BUILD.md

简介轻量级标记语言 Markdown

* http://www.worldhello.net/gotgithub/appendix/markups.html

Github 偏爱的 Markdown

* http://github.github.com/github-flavored-markdown/

在线的 Markdown 编辑浏览器

* http://dillinger.io/

致谢

在本书的编写过程中，得到了很多热心人士，同时也是技术高手的帮助。

Peter Wang (@happycasts)　开源电子书 LGCB 的作者，从他这儿我学会了用 github 搭建这本书的写作方法，Peter 还有很多精心录制的学习开源技术的视频，大家一定不要错过。

chunzi　Pro Git 的中文译者，正是这本书教会了我如何使用Git，启发了我想要通过协作迭代来写书，后来我发现协作本身比写作更有乐趣。虽然后来没有采用Pro Git的框架，但还是非常感谢chunzi及时回复了我的邮件，相信这本书还会帮助到更多的人。

Larry Cai (@larrycaiyu)　上海爱立信研发中心的软件开发高级专家，从微博上加入“中文开源技术书”之后，身为管理员的larry就一直默默主动帮我解决从 markdown 到生成 pdf 格式的各种细节问题，他所维护的 mkbok 这里强烈推荐噢，以后一定会成为 GitHub 上写书必备之利器。

tonghuix　倡导自由开源生活方式的 @爱开源未来，正是他的提醒，我最后将本书的名字定为《源码开放学ARM》，以此表示对开源社区文化的尊重和支持，等有机会我会争取在开源硬件上写一本真正开源学习的书。

所有来自这些朋友的热情帮助和技术支持，让我常有如拨云雾而见青天之顿悟，虽然你们中的大多数未有机会得以见面，但对技术传播和分享的共同热爱使我们心灵相通。在此希望能够一并致谢。

参与

你可以通过关注新浪微博 @亚嵌李明老师 联系作者或者发邮件给 limingth AT gmail.com 告知你希望出现在书中的内容和想要解决的问题。

如果你愿意参与本书的编写，可以通过 fork 《源码开放学ARM》 作出贡献。

目录

第 1 章

开发环境搭建

1.1 硬件平台

本课程采用 广州友善之臂 的 Tiny210 开发板 作为实验开发平台。 关于这个硬件开发板的详细描述和介绍，可以参考阅读下面这个链接的内容。 http://arm9.net/tiny210.asp

图 1.1: Tiny210广州友善之臂开发板

请通过阅读上述材料之后，回答以下有关开发板硬件平台的问题：

1）开发板采用的主芯片是什么型号，基于什么 ARM 内核？
2）开发板运行程序的主频是多少？ 使用什么内存？ 内存有多大？
3）开发板上面能够运行哪几种操作系统？ 它们有什么差别？
4）什么叫 BSP ，开发板的 BSP 支持哪些外设？

1.1.1 芯片识别

```
board:        tiny210
CPU:        S5PV210 (封装/FBGA)
MEM:        K4T1G08
```

```
FLASH:        K9F2G08
NET:        DM9000
AUDIO:        WM8960
UART:         MAX3232
```

1.1.2 外设识别

```
reset键
key键/home/back/menu
串口
网口
USB口
SD卡
CAMERA接口
MIC/HEARPHONE
AV口
启动跳线：NAND-SDboot
可变电阻：ADC
LCD接口：外接LCD
```

1.1.3 准备工作

需要参加课程的学员提前准备好以下环境：

```
1）电源线 （5v）
2）串口线 （双母头）
3）开发板 （已经烧写了 u-boot 或者用 SD 卡可以启动到 u-boot 下）
4）超级终端 （hypertrm，115200，无硬件流控，连接开发板有输出）
5）如果是用笔记本，通常没有串口，需要自备一根 USB转RS232串口的线，并安装相应驱动。
   推荐使用 Z-Tek 力特，驱动比较好装： http://www.360buy.com/product/134961.html
```

如果开发板是刚拿到的，则一般都没有烧写 u-boot，可以自己烧写 u-boot 到开发板SD卡上，这需要准备以下条件：

```
1）SD卡（自备，2G或者4G都可以）
2）SD卡读卡器（自备，连接PC后格式化为 FAT32 分区，可以显示盘符）
烧写SD卡的步骤可以参考随开发板附带的《用户手册》，需要用到 SD-Flasher.exe 这个工具。
```

1.1.4 硬件平台的验证

在开始学习后继内容之前，通常需要对硬件平台进行以下3个方面的正确性验证：

```
1）主芯片 （通常使用 jtag 工具，能够读取到 cpu 的 ID）
2）串口输出 （通常使用 超级终端，设置好波特率和流控制，能够和开发板bootloader进行交互）
3）下载和烧写 （一般通过串口或者usb进行，下载是到 SDRAM，烧写是到 NandFlash）
```

串口连接开发板，验证bootloader命令的输入输出：

```
1）连接开发板
2）启动超级终端 hypertrm(.exe)
3）选择 COM1（pc）
4）修改波特率为 115200
5）修改流控制为 无（none）（否则影响输入）
6）连接之后，重启开发板，及时按回车键，输入 help
   （如果发现有输出但无法输入，留意 Scroll Lock 是否被误按）
```

嵌入式开发的硬件平台，是以后我们学习ARM开发课程的实验平台。目前常见的硬件开发板从 ARM7、ARM9、ARM11内核一直到 Cortex A/R/M 系列发展很快，大部分都是采用核心板+底板的结构，如何进一步了解硬件设计的原理，弄懂主芯片和外设之间的作用关系，就需要我们了解硬件设计原理图方面的知识了。

1.2 硬件原理图

通常提供硬件平台的厂家，都会随开发板光盘附带硬件原理图。硬件原理图一般都是以PDF格式提供，这和硬件设计人员通过使用 Protel/OrCAD Capture 这样的硬件原理图设计工具产生的文件不同，但PDF的文档方便查阅，易于打开，对于嵌入式软件开发工程师用于开发已经足够了。

Tiny210-core-board.pdf

Tiny210-mother-board.pdf

推荐使用 FoxReader 福昕阅读器来打开 PDF 文档

```
1. Ctrl + 放大  Ctrl - 缩小  Ctrl 滚轮也可以
2. Ctrl + F 查找
3. Ctrl + Shift + N 跳转
4. Ctrl + TAB 切换文档
```

1.2.1 原理图包含的信息

核心板的原理图和底板的原理图，通常分为两个文件存放。通过查看原理图，能够知道哪些信息?

有哪些芯片
芯片有哪些管脚
芯片之间的连接关系
芯片与外设本身之间的连接关系
得到有些芯片管脚的默认连接（接地/接电源/NC（No Connection）不连接的管脚）
电阻电容等分立元件（模拟电路）

1.2.2　核心板

核心板一般包含了最小系统，也就是主芯片SoC，内存SDRAM，闪存FLASH，复位电路Reset，调试接口JTAG，时钟CLOCK和电源Power。请在原理图上分别找到这些器件，并初步熟悉了解它们的芯片型号。

```
主芯片          SoC:        S5PV210 -> 584pin              [S5PV210_UM_REV1.1.pdf]
内存            Mem:       K4T1G -> 1Gb=128MB              [K4T1G164QE_rev11.pdf]
闪存            Flash:       K9F2G08 -> 2Gbx8bit=256MB     [K9F2G08.pdf]
复位电路        Reset:       Max811                            [MAX811T.pdf]
调试电路        Jtag:       TCK/TDI/TDO/TMS/TRST
时钟           CLOCK:       24Mhz (X1) XXTI/XXTO
发光二极管      LED:        LED1-4
高清接口        HDMI:        mini HDMI(CON8)
```

1.2.3　底板

底板一般包含了常见外设，例如通用GPIO，串口UART，网卡Net，液晶屏接口LCD，音频接口Audio等。请在原理图上找到这些器件，并初步熟悉了解它们的芯片型号。

```
CONN-AB:        30*2 * 2个 = 120pin 引出 (例如串口TxD/RxD)
CONN-C:               15*2 = 30pin 引出 (例如I2C, CAM)
Power-On/Off:        S1
NET :               DM9000AEP (Ethernet) HR91105A          [DM9000.pdf]
Audio:               WM8960 (DA/AD convert)                [WM8960_Rev40.pdf]
Buzzer:              PWM Timer (beep)
RTC:               Real-Time Clock
I2C-Eeprom:       MAC address (6 bytes)
Buttons:        K1-K8 (XEINT16-27)
SD-CARD:        Data(4pin) + (4pin)
UART:                串口(通用异步收发器)                  [MAX3232.pdf]
LCD :                40pin / 45pin (DATA-24pin)          [H43-HSD043I9W1.pdf]
```

以上所有芯片的数据手册，都可以在开发板附带光盘中找到，也可以从 https://github.com/limingth/ARM-Resources 链接下载得到。

1.3　开发工具链

在ARM开发领域，有两大类开发工具可以选择，一类是基于 Windows 平台的 SDT，ADS，RealView MDK，DS-5 系列，一类是基于 Linux 平台的 GNU Cross-Toolchain 。

考虑到从简单到复杂的学习路线，我们先介绍 Windows 平台上的工具链。一旦我们对工具链背后的开发思路比较了解之后，再来学习 Linux 上的工具就会比较容易上手。

有关 ADS 工具和 MDK 工具的介绍，可以参考阅读百度百科的介绍。

ADS简介

MDK简介

DS-5 (ARM Development Studio 5) 是 ARM 公司最新推出的开发工具套件，采用 Eclipse IDE，支持对于裸机，RTOS 和 Linux 系统的调试。

DS-5下载页面

总体说来，不同的开发工具套件虽然在界面上做了很大改动，但后台使用的命令行工具链基本是一样的。下面以 ADS 安装为例，对命令行工具链做一个简单说明。

1.3.1 ADS 安装使用说明

工具下载 http://limingth.github.com/ARM-Tools

ADS1.2.zip 解压之后运行 setup.exe 安装

```
注意：          1) Full 安装，不是 Typical
        2) Install Licence，在解压后的 crack\licence.dat
```

安装完成之后

```
安装目录： C:\Program Files\ARM\ADSv1_2\Bin
```

图形开发环境

```
IDE.exe -         ADS IDE
axd.exe         -         AXD debugger
```

命令行开发工具链 启动命令行方式

```
开始 -> 运行 -> cmd 命令打开一个窗口

C:>path
是否有 C:\Program Files\ARM\ADSv1_2\bin

如果是 path 问题，需要添加 路径到 path 环境变量
我的电脑 -> 鼠标右键属性 -> 高级 -> 环境变量 -> 系统变量下面添加
        （注意用分号间隔；原来的不要删除掉，把 C:\Program Files\ARM\ADSv1_2\bin 添加到最后）

C:>armcc
是否有这个命令，这一点最重要，如果成功则会输出
ARM C Compiler, ADS1.2 [Build 805]

Usage:          armcc [options] file1 file2 ... filen
Main options:
xxxxx

armcc.exe -         C compiler (armcpp.exe) /(gcc)
armasm.exe -         ASM Assembler /(as)
armlink.exe -         Linker         /(ld)
fromelf.exe -         Bin-Utils /(objdump/objcopy)
```

```
armcc 常用编译参数
        -c 只编译, 不连接
        -D (定义)条件编译 (-DDEBUG)
        -U (不定义)条件编译 (-DDEBUG)
        -g 增加调试信息
        -I 指定 include 路径(自己的)
        -On 编译优化级别
        -S 生成汇编
        -o 指定生成文件名

思考问题： arm-linux-gcc 交叉编译器的库 和 armcc 链接的库是否一样？

用法举例：
        armcc hello.c
                默认会生成  __image.axf (a.out), 其中 axf 文件是 ELF 格式的可执行文件

        armcc -c hello.c
                默认会生成  hello.o , 此时还需要 link 之后才能生成 axf 可执行文件

特殊用法：
        如果一个C程序, 没有 main 函数, 编译会怎么样？ (有警告warning, 无错误error, 能生成可执行文件
axf)
```

C 编译器

```
armasm 通常只生成 .o 的目标文件

用法举例：
        armasm start.s
                默认会生成  start.o, 此时还需要 link 之后才能生成 axf 可执行文件
```

asm 汇编器

```
armlink 常用链接参数
        -ro-base 指定可执行代码的位置 (代码段执行地址)
        -rw-base 数据段执行地址
        -first  指定 .o 文件放在链接的最开始处
        -entry  指定 axd 调试工具加载 axf 文件的入口地址 (转成bin之后就丢失了)
        -scatter file 指定链接脚本 (.scf文件/在linux下.lds文件)

用法举例：
        armlink hello.o -o hello.axf
        armlink -first start.o -ro-base 0x0 -entry begin start.o main.o -o hello4.axf
```

obj 链接器

```
fromelf 常用转换参数
        -bin 生成bin文件，最终烧写到开发板上
        -c 生成txt文本文件，反汇编文件
        -d 打印数据段内容
        -s 打印符号表
        -t 打印字符串表

用法举例：
        fromelf -bin hello.axf -o hello.bin
        fromelf -c -s -d hello.axf -o hello.txt
```

bin-utils 二进制转换工具

```
单独汇编程序的编译链接
        armasm start.s
        armlink start.o -o demo.axf

单独C程序的编译链接
        armcc -c hello.c
        armlink hello.o -o hello.axf

汇编和C程序的混合链接
        armasm start.s
        armcc -c main.c
        armlink -first start.o -ro-base 0x0 -entry begin start.o main.o -o demo.axf
```

综合应用

```
DDI0100E_ARM_ARM.pdf                - ARM体系结构知识，侧重于内核
ADS_CompilerGuide_D.pdf         - 编译器使用
ADS_AssemblerGuide_B.pdf        - 汇编器使用
ADS_LinkerGuide_A.pdf               - 链接器使用
ADS_DebugTargetGuide_D.pdf        - 调试器使用
```

ARM Docs 开发文档

1.3.2 Linux 工具链安装使用说明

安装 arm-linux-gcc 工具下载： arm-linux-gcc-4.5.1-v6-vfp-20120301.tgz

解压安装：

```
sudo tar zxvf arm-linux-gcc-4.5.1-v6-vfp-20120301.tgz  -C /
ls /opt/FriendlyARM/toolschain/4.5.1/bin/
        -> arm-linux-gcc
```

```
vi ~/.bashrc
        -> export PATH=$PATH:/opt/FriendlyARM/toolschain/4.5.1/bin/
source ~/.bashrc
echo $PATH
        -> /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/opt/FriendlyARM/
toolschain/4.5.1/bin/
$ arm-linux-gcc
arm-linux-gcc: no input files

以上步骤完成之后，可以在任意目录下使用 arm-linux-gcc 来编译程序。在输入 arm-linux- 之后使用 tab 键，
还可以看到 arm-linux-as, arm-linux-ld, arm-linux-objcopy, arm-linux-objdump 等工具也都一并安装完成。
```

```
sudo apt-get install minicom
sudo minicom -s
        -> Serial port setup : /dev/ttyS0 (for USB Serial Port: /dev/ttyUSB0)
        -> Hardware Flow Control : No
        -> Save setup as dfl
        -> Exit
minicom

此时重启开发板，可以看到 minicom 中能够显示从开发板串口输出的字符。
```

安装 minicom

1.4　基本开发流程

1.4.1　Windows 平台开发

ADS 汇编语言格式要点　TAB 开头

ARM 汇编语言格式要求除了 label 标号之外，其他包含伪操作和指令的行都应该以一个 TAB 开头。

AREA 名称

AREA 表示定义代码段/数据段区域的开始，后面跟的名称会进入符号表参与链接。

ENTRY 入口

表示指定汇编程序的入口，最多只能写一个，也可以不写。

END 结束

表示汇编程序的结束，必须要写。

; 注释

分号表示注释，用 // 或者 /* */ 都不是注释。

label 定义符号

用于程序跳转时的标号，必须顶格写。

import symbol

用于链接外部的符号名，例如跳转到C语言的主函数。

export symbol

用于对外输出汇编程序内部的符号名，汇编文件内部的symbol默认对外是不公开的，隐藏的，类似C程序的static修饰。

C 程序注意要点 main函数的负面影响

链接器会引入很多外部代码，带来跟踪调试的很多问题，同时造成可执行bin文件的体积明显增大。

** “main的用法”**

默认main函数真正的入口是”main”，因此直接用 “__main” 来定义C语言的主函数是比较便利的做法。

替换return 0

如果主程序没有无限循环，那么最后执行 return 0 的结果是不可预期的，应该用while(1); 无限循环来替代。

volatile 修饰符

编译器对 volatile 修饰的变量或者指针，都会避免优化，强制访存。这对于嵌入式寄存器的读写操作是非常必要的。

unsigned 修饰符

对于特殊功能寄存器的访问，通常不需要进行算术运算，将它们声明为 unsigned 无符号整型是通常的做法。

编写 Makefile all 目标

第一个默认的目标，一般都起名为 all，make执行时不需要跟目标名，会自动执行默认目标。

clean 目标

清除中间产生的不必要的文件，例如 .o .bak 等。

宏变量

引入 PRJ，SRCS，OBJS ，增强项目的可移植性。

匹配替换技巧

$(SRC:.c=.o) 的用法，灵活替换源文件到目标文件。

依赖关系的传递

用依赖关系，对于文件数量较多，编译时间较长的项目非常有好处，可以做到增量编译。

隐含规则

%o:%c 的用法，如果需要修改编译参数，则需要重新实现该条隐含规则。

```
make clean        清除原有临时文件
make              重新生成所有文件
```

make 编译项目

```
# loadb 输入命令
        超级终端菜单 -> 传送 -> 发送文件 -> 选择文件 + Kermit协议 -> 点击发送

# go 0x21000000
        之后观察 LED1 灯亮的现象
```

测试可执行文件

1.4.2 Linux 平台开发

以下代码可以从 https://github.com/limingth/ARM-Codes/tree/master/tiny210-linux-codes/1-led-s 下载

相关工具可以从 https://github.com/limingth/ARM-Tools/tree/gh-pages/utils 下载

```
.global _start
_start:
        ldr r0, =0x1
        ldr r1, =0xe0200280
        str r0, [r1]

loop:
        ldr r0, =0x0
        ldr r1, =0xe0200284
        str r0, [r1]
        bl delay

        ldr r0, =0x1
        ldr r1, =0xe0200284
        str r0, [r1]
        bl delay

        b loop

delay:
        ldr r0, =0x100000
go_on:
        sub r0, r0, #1
        cmp r0, #0
        bne go_on
        mov pc, lr
```

led.s 汇编程序

```
all:
        arm-linux-as led.s -o led.o
        arm-linux-ld led.o -o led.elf
        arm-linux-objcopy -O binary led.elf led.bin
        arm-linux-objdump -d led.elf > led.lst
        ./mktiny210spl.exe led.bin sd-led.bin
```

Makefile

```
make

-> ls -l *.bin
   -rwxr-xr-x 1 limingth limingth   76 2012-05-11 18:21 led.bin
   -rw-r-r-- 1 limingth limingth 8192 2012-05-11 18:21 sd-led.bin
```

make

```
insert sd card
dmesg | grep sdb
        -> /dev/sdb should be found
sudo dd iflag=dsync oflag=dsync if=sd-led.bin of=/dev/sdb seek=1
        -> 记录了16+0 的读入
           记录了16+0 的写出
           8192字节(8.2 kB)已复制, 0.209805 秒, 39.0 kB/秒
```

烧写 sd-led.bin 到 SD-Card

```
1. fdisk SD card to a Win95-FAT32 partition

if you just buy a new SD card, here we do it from the very beginning
sudo fdisk /dev/sdb
Command (m for help): m (help)
Command (m for help): p (print the partition table)
Command (m for help): d (delete the old partition)
Command (m for help): n (add a new partition)
Command action
   e   extended
   p   primary partition (1-4)
p (for primary) ->
Partition number (1-4): 1
First cylinder (1-121, default 1): 1
Last cylinder, +cylinders or +size{K,M,G} (1-121, default 121):  (Enter)
Using default value 121

Command (m for help): t
Selected partition 1
Hex code (type L to list codes): L

Hex code (type L to list codes): b
Changed system type of partition 1 to b (W95 FAT32)

Command (m for help): p

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1         121      971901    b  W95 FAT32

Command (m for help): w
```

```
The partition table has been altered!

Syncing disks.
format FAT32 (This is VERY important for later use! However, it is not a must for led blink)

2. sudo mkfs.vfat /dev/sdb1
now you can test if SD-boot can blink led1

3. insert SD card to slot (CON10 of tiny210)
>> switch to SDBOOT of S2
reset and see if it works!
```

补充：如何格式化 SD-Card

第 2 章

芯片手册导读

2.1 内部结构框图

如何阅读芯片手册，是很多初学者需要面临和解决的问题，但大部分人都会感到难以入手不知所措。除了数据手册一般都全部采用英文书写所带来的困难之外，更多的困难还在于不知道如何去阅读相关章节，不知道哪些是真正重要的或者哪些是无关紧要的内容。

我们以 S5PV210 芯片数据手册为例，给大家讲解一下应该如何阅读芯片手册。这本手册大约有2000多页，如果能够把握住其中的内容，则阅读其他手册也应该没有障碍。

2.1.1 S5PV210 芯片数据手册目录结构

```
一、Overview
        1. overview
                block diagram
        2. memory map
                System Memory Map
                SFRs (Special Function Register)
                        0xE0000000 - 0xFB6FFFFF (max-512M)
                        真实的存储容量：寄存器的个数 * 4bytes
                        以串口为例： 15个 * 4组 = 60个 * 50 = 3K * 4 = 12K
        3. ball map
                pin assignment
                signal description
                        UART-RxD0/TxD0 CTS0/RTS0
二、System
        1. GPIO
        2. Clock
        3. Power
        4. boot sequence
三、Bus
        1. AXI/AHB
四、Interrupt
        1. Vectored Interrupt Controller
五、Memory
        1. DRAM => DDR memeory
        2. SROM => SRAM + ROM(Nor Flash)
        3. OneNand
        4. Nand
```

```
        5. Compact Flash
六、DMA
        1. DMA Controller
七、Timer
        1. PWM Timer
        2. System Timer
        3. Watchdog Timer
        4. RTC
八、Connectivity
        1. UART
        2. IIC-bus
        3. SPI (serial peripheral interface)
        4. USB Host
        5. USB OTG
        6. SD/MMC
九、Multimedia
        1. LCD
        2. CAM
        3. G3D
        4. CODEC
        5. TVOUT
        6. VIDEO
        7. MIXER
        8. IMAGE ROTATOR
        9. JPEG
        10. G2D
十、AUDIO
        1. IIS
        2. AC97
        3. PCM
        4. ADC (TS)
        5. KeyPad
十一、Security
```

2.1.2　S5PV210 Block Diagram 芯片内部结构框图

```
PRODUCT OVERVIEW
Block Diagram
        CPU（运算）（控制）
        BUS（地址/数据/控制）
        Controllers（特殊功能寄存器）
Pin Assignment Diagram 管脚定义
SIGNAL DESCRIPTIONS 信号描述
SPECIAL REGISTERS (SFRs)
SYSTEM MANAGER
        System Memory Map（系统内存映射）
                0x4000000 = 0x40M = 64M
        System Manager Registers (10+)
SFR
        Name + Address + R/W + desc. + Reset Value
        Bit-Field 位域
Timing 时序图
Controller 控制器
```

```
        OVERVIEW 综述
        Block Diagram 框图
        SPECIAL REGISTERS 特殊功能寄存器
        Timing 时序图
```

2.1.3 芯片手册的一般结构

```
CORE - Cortex-A8
        ALU（运算器）
        Regs（通用寄存器）
        MMU
        CACHE
BUS
        片内 - 地址线32bit
        片外 - 地址线取决于BANK的大小
iRAM（SRAM）
iROM
Peripheral Controllers
        Memory cont.
        GPIO cont.
        UART cont.
        USB cont.
        IIC cont.
        SD cont.
        SPI cont.
        DMA cont.
        Interrupt cont.
        G3D, G2D
        HDMI
        CAMERA
        TVOUT
        IMAGE ROTATOR
        JPEG
```

2.1.4 芯片手册应该怎样阅读？

```
Overview
        CHIP Block Diagram, MemoryMap,
        Pin Assignment, Signal Description
        SFRs（how many, address range）
        Controllers（how many）
System
        Clock（BUS）, Power
        Boot Sequence（启动/引导流程）
        GPIO
Device
        UART
        NandFlash
        Memory（DDR）
```

```
        DMA
        Interrupt
        PWM Timer
        LCD + TS
        AUDIO
        DM9000
        SD
        Security
```

2.2　存储管理和地址映射

2.2.1　存储器件

对于一个SoC芯片，最重要的认识莫过于了解它所能支持的存储器件和地址空间。这对于任何一个SoC芯片而言，都应该放在精读内容的首位。

存储器件主要包括以下这些，了解和掌握它们的接口和存储特性，是做嵌入式底层开发人员所必须掌握的理论知识。

```
1) 片内的 RAM
        iRAM - SRAM
2) 片外的 RAM
        SRAM
        SDRAM
        DDR SDRAM
3) 片内的 ROM
        iROM
4) 片外的 ROM
        Nor Flash
        Nand Flash
        OneNand Flash
```

2.2.2　地址空间

地址空间主要是指32位地址线 0-4G 所对应（也可称为映射）的存储器件和访问方法（如何读写）。

```
0 地址          - 加电后运行的第一条指令
片内RAM         - 可以无需初始化直接使用的存储
SFR         - 特殊功能寄存器的地址范围
SDRAM           - 程序通过bootloader命令可以下载到的地址
虚地址          - 使能MMU之后的虚拟地址
```

2.2.3　Memory Map 存储映射

存储映射这个概念是本节最重要的知识，以 S5PV210 芯片为例，了解并掌握它的映射情况。

```
Boot Area: 0x0 - 0x20000000 =512M
        Mirrored region depending on boot mode

DRAM0:          0x20000000 - 0x3FFFFFFF: 2^29 = 512M
DRAM1:  0x40000000 - 0x7FFFFFFF: =1G
SFR:        0xE0000000 - 0xFFFFFFFF: =512M
iROM:        0xD0000000 - 0xD0010000: =64K
iRAM:         0xD0020000 - 0xD0038000: =96K (0x18=24*0x1000)

思考问题： 根据上述地址映射，对不同地址进行的访问，会引发底层硬件产生何种响应？
        *(int *)0x00000000 : it depends
        *(int *)0x21000000 : DRAM
        *(int *)0xE0200280 : GPIO SFR
        *(int *)0xE2900000 : UART SFR
```

2.3 特殊功能寄存器

特殊功能寄存器(Special Function Register) 是在 SoC 芯片内部的一个重要组成部分，区别于 ARM 内核的 R0—R15 这样的寄存器，这些寄存器本质上和片内的 SRAM 一样，按地址访问，掉电以后值会丢失，无需初始化直接可以访问这样一些特性。

从逻辑结构上看，所有对特殊功能寄存器(以下简称 SFRs) 的访问操作的实现，都对应到不同的外设控制器上。访问指令本质上和对外部存储器的读写一样，都是通过LDR和STR汇编指令来实现的。

2.3.1 关于内核，控制器，总线和外设之间的关系

```
内核：Cortex-A8 (CORE)
        寄存器 Regs (R0-R15, CPSR)
        指令集 Ins (add/sub, ldr/str)
        总线 (访存指令)
地址概念：
        cpu：32bit        ldr r0, [r1]         (r1:addr)
                1)SRAM:
                2)SFR:
                --------------------------
                3)DDR:
                4)Device internal memory:
总线概念：
        片内总线
                地址线-32bit        由 地址寄存器的字节数 决定
        片外总线
                地址线-29bit        由 设计cpu时，外接的单个存储器件的最大容量 决定
        内存控制器
                通过把 32bit 的最大访存范围，分割为若干个 bank 来进行统一访问
外设控制器概念：
        外设 = 可见(接插件/关联芯片/连接线) + 不可见(外设控制器)
        外设控制器 = 寄存器接口 <---- 黑匣子(类似函数库.o) ---> 外设工作的时序图
总结关系：
        都是涉及到裸板驱动（无操作系统/无MMU参与）
```

```
本质上除了运算之外，都是变成为对地址的读写
C语言的指针，在本阶段，占一个很重要的角色。
```

2.3.2 特殊功能寄存器的设置

所有我们设置的SFR，都应该和外设控制器内部的工作逻辑有关。因此了解Controller的工作方式和内在逻辑，是我们能够得以正确配置这些寄存器的正确路径。

每一个 controllers 都有一批自己的寄存器，通过读写操作就可以用来进行软件编程和控制。

```
全大写，未来用来宏定义的，前面部分是这个Controller的缩写，后面部分就是它的功能）
CON - control 控制
STAT - status 状态
DAT - data 数据
MOD - mode 模式
FIFO - fifo 缓冲寄存器
CFG - config 配置
CNT - counter 计数
TXH - transmit Holder 发送缓冲
RXH - receive Holder 接收缓冲
BRDIV - baud divisor 波特率分频因子
```

Register Name

```
这个地址，是在写代码的时候，所对应操作的寄存器的唯一标识。
名字只是用来帮助记忆的，不是内部标识，也不是用来给编译器的。
```

Register Address

```
#define PRO_ID          (*(volatile unsigned int *)0xE0000000)

volatile 关键词的作用
        http://learn.akae.cn/media/index.html
        http://learn.akae.cn/media/ch19s06.html

写法要点：
        强制类型转换，1个括号
        防止优先级结合问题，1个括号
        unsigned 无符号类型，防止右移问题
        int 类型对应4个字节，依据手册决定是否换为 char 类型
```

定义举例

2.4 时序图

2.4.1 基本概念

时序图的基本概念

时序图是芯片与芯片之间进行数据通信所需要遵循的一种协议。通过时序图能够直观的看出，不同的芯片引脚在时间轴上的不同时刻所呈现出来的高低电平的变换，这些引脚中，有的代表地址信息，有的代表数据信息，有的代表控制信息，它们出现的先后顺序是有严格的时间参数规定的，这称为 Timing 。

芯片手册里的时序图一般都会列出其中关键的时间参数名称，在手册中都可以查到这些时间参数的具体要求。

```
min        - 最小值
max        - 最大值
typical        - 典型值
```

[Timing Table]

第 3 章

GPIO 控制器

3.1 控制器内部结构

```
GPIO Controller
        how many pin
        pin number, pin name, pin mux functional
        device <-> GPIO (LED1 -> GPJ2_0) 原理图
GPIO SFRs
        Set mux function
                控制寄存器 GPXCON
        Set pin value
                数据寄存器 GPXDAT
        Set interrupt function
                中断寄存器 ...
```

3.2 GPIO 输出引脚

General Purpose Input/Output (p92-p352)

```
1) 237 multi-functional input/output port pins
        34 general port groups
2) GPIO <---> peripheral controller (signal mux)
3) GPIO Block Diagram
        APB bus (addr + data) *(int *)0xE0000000 = 0x1234;
        Register File (GPIO SFRs)
        Mux Control (functional)
        Interrupt Control (Interrupt cont.)
4) Pin Mux Description
        pin name -> GPIO name
        default function
```

3.3 GPIO 特殊功能寄存器

```
GPIO REGISTER DESCRIPTION
        addr range: 0xE020_0000 - 0xE020_0F80
        Reg1: GPA0CON: GPA0 + CON (control)
                bit-field name: GPA0CON[7] -> GPA0_7
                field width: [31:28]
                GPA0_7 pin setting:
                        input: 0000
                        output: 0001
                        UART_1: 0010
                        INT: 1111

        Reg2: GPA0DAT
                bit-field: [7:0]
                when input:        the pin state(high-level:1 low-level:0)
                when output: set bit 1, output high-level
                when functional: leave this pin to peripheral controller

        Reg3: GPA0PUD
                Pull-up
                Pull-down
                        00 = Pull-up/down disabled
                        01 = Pull-down enabled
                        10 = Pull-up enabled
                        11 = Reserved

        Reg4: GPA0_INT_CON
                Sets the signaling method
                        000 = Low level
                        001 = High level
                        010 = Falling edge triggered
                        011 = Rising edge triggered
                        100 = Both edge triggered
                        101 ~ 111 = Reserved

        Reg5: GPA0_INT_MASK
                Enables Interrupt / Masked
                        0 = Enables Interrupt
                        1 = Masked

        Reg6: GPA0_INT_PEND
                Interrupt occur status
                        0 = Not occur
                        1 = Occur interrupt
```

3.4 GPIO 驱动代码实现

3.4.1 汇编程序

```
TAB
AREA led
CODE, DATA
READONLY
label
instruction...
END
```

3.4.2 ARM汇编的延时函数

```
        主程序中
        bl delay
        ...

delay
        ldr r0, =0x10000000
go_on
        sub r0, r0, #1
        cmp r0, #0
        bne go_on

        mov pc, lr
```

3.4.3 立即数的表示

```
mov 操作，#后面跟着的数字是立即数，会写入指令中
有效位不超过8位，同时通过循环右移偶数位得到
1 0000 0010 = 0x102
10 0000 0100 = 0x204
```

3.4.4 连接开发板

```
启动超级终端 hypertrm        (.exe)
选择 COM1（pc）
修改波特率为 115200
修改流控制为 无（none）（影响输入）
连接之后，输入 help
如果发现有输出但无法输入，留意 Scroll Lock 是否被误按

# loadb 输入命令
超级终端菜单 -> 传送 -> 发送文件
-> 选择文件 + Kermit协议 -> 点击发送
# go 0x21000000
之后观察 LED1 灯亮啦
```

3.4.5 led.c 参考代码实现

```
// led.c
#define GPJ2CON                (*(volatile unsigned int *)0xE0200280)
#define GPJ2DAT                (*(volatile unsigned int *)0xE0200284)

void led_init(void)
{
        // LED1-4: GPJ2_0, ... GPJ2_3
        // 0001 0001 0001 0001 = output
        GPJ2CON &= ~0xFFFF;
        GPJ2CON |= 0x1111;

        return;
}

void led_on(void)
{
        // set bit 0 -> led on
        GPJ2DAT &= ~0xF;

        return;
}

void led_off(void)
{
        // set bit 1 -> led off
        GPJ2DAT |= 0xF;

        return;
}
```

3.4.6 button.c 参考代码实现

```
// button.c
#define GPH2CON                (*(volatile unsigned int *)0xE0200C40)
#define GPH2DAT                (*(volatile unsigned int *)0xE0200C44)

void button_init(void)
{
        // K1 - K4        (see mother board)
        // GPH2_0 - GPH2_3
        // GPH2CON[0]   [3:0]   0000 = Input
        // ...
        // GPH2CON[3]   [15:12]   0000 = Input
        GPH2CON &= ~0xFFFF;

        return;
}

int button_is_down(int which)
{
```

```
        // button down -> DAT = 0
        int index = which - 1;

        if ((GPH2DAT & (1<<index)) == 0)
                return 1;

        return 0;
}
```

3.4.7 buzzer.c 参考代码实现

```
// buzzer.c
#define GPD0CON                (*(volatile unsigned int *)0xE02000A0)
#define GPD0DAT                (*(volatile unsigned int *)0xE02000A4)

void buzzer_init(void)
{
        // buzzer GPD0CON
        // [0] SET 1
        GPD0CON |= 1<<0;

        return;
}

void buzzer_on(void)
{
        // set bit 1 -> buzzer on
        GPD0DAT |= 1<<0;

        return;
}

void buzzer_off(void)
{
        // set bit 0 -> buzzer off
        GPD0DAT &= ~(1<<0);

        return;
}
```

第 4 章

CLOCK 时钟管理

4.1 时钟发生器

4.1.1 时钟源的周期换算关系

```
        24Mhz 输入时钟
        10^(-3)         毫秒         10^(-6)         微秒         10^(-9)         纳秒
        1Khz 10^3                  1Mhz 10^6                  1Ghz 10^9
        1Ghz -> 1纳秒
        100Mhz -> 10纳秒
```

4.1.2 Clock Controller 时钟控制器 (p353-p417)

```
CMU: Clock Management Unit
        总线频率
        MSYS: Main  (200Mhz-100Mhz)
                Cortex A8 Core
                DRAM controller
        DSYS: Display (166Mhz-83Mhz)
                JPEG
        PSYS: Peripheral (133Mhz-66Mhz)
                UART
                AC97
                PWM Timer
                GPIO

Clock Generator 时钟发生器
        S5PV210 Top-Level Clocks
                XXTI - 24Mhz --> (4 PLLS's input)
                XrtcXTI - 32.768Khz
        4 PLLs (APLL, MPLL, VPLL, EPLL)
                Phase Locked Loop 锁相环（倍频）
```

4.2 时钟输出频率

4.2.1 时钟输出

```
MSYS clock domain (H->HighPerformance P->Peripheral)
        AHB / APB
        freq(ARMCLK)  = 1000Mhz                1000M/1
        freq(HCLK_MSYS) = 200Mhz        ARMCLK/5
        freq(PCLK_MSYS) = 100Mhz        HCLK_M/2
        freq(HCLK_IMEM) = 100Mhz        HCLK_M/2

DSYS clock domain
        freq(HCLK_DSYS) = 166Mhz
        freq(PCLK_DSYS) = 83Mhz                HCLK_D/2

PSYS clock domain
        freq(HCLK_PSYS) = 133Mhz
        freq(PCLK_PSYS) = 66Mhz                HCKL_P/2
        freq(SCLK_ONENAND) = 133M/166Mhz

PLL PMS setting
```

4.3 锁相环和分频器

4.3.1 锁相环 PLL

```
PLL: Fin -> Fout 倍频                XPLL_CON
MUX: 0/1 选择器 SRC选择源        CLK_SRC
DIV: /2-16 分频器 DIV                CLK_DIV

REGISTER DESCRIPTION
OM[0]: cpu pin OM[0]=0, XXTI=24M
Fin_PLL: 24Mhz
APLL: APLL_CON: e0100100: 0xa07d0301 => 1000Mhz
```

4.3.2 分频器 Divider

```
Mux_APLL: CLK_SRC0, 0xe0100200: 10001111 [0]=> 1
Mux_MSYS: CLK_SRC0, 0xe0100200: 10001111 [16]=> 0
DIV_APLL: CLK_DIV0, 0xE0100300: 14131440 [2:0]=> 0 = /1
DIV_APLL: CLK_DIV0, 0xE0100300: 14131440 [10:8]=> 100 = /5
```

4.3.3 寄存器配置

```
[FriendlyLEG-TINY210]# md 0xe0100100
e0100100: a07d0301 00000000 a29b0c01 00000000    ..}.............
```

```
e0100110: a8500303 00000000 00000000 00000000    ..P.............
e0100120: a06c0603 00000000 00000000 00000000    ..l.............
e0100130: 00000000 00000000 00000000 00000000    ................
e0100140: 00000000 00000000 00000000 00000000    ................
e0100150: 00000000 00000000 00000000 00000000    ................
e0100160: 00000000 00000000 00000000 00000000    ................
e0100170: 00000000 00000000 00000000 00000000    ................
e0100180: 00000000 00000000 00000000 00000000    ................
e0100190: 00000000 00000000 00000000 00000000    ................
e01001a0: 00000000 00000000 00000000 00000000    ................
e01001b0: 00000000 00000000 00000000 00000000    ................
e01001c0: 00000000 00000000 00000000 00000000    ................
e01001d0: 00000000 00000000 00000000 00000000    ................
e01001e0: 00000000 00000000 00000000 00000000    ................
e01001f0: 00000000 00000000 00000000 00000000    ................

[FriendlyLEG-TINY210]# md 0xe0100200
e0100200: 10001111 00000000 00000000 00000000    ................
e0100210: 66667777 00000000 00000000 00000000    wwff............
e0100220: 00000000 00000000 00000000 00000000    ................
e0100230: 00000000 00000000 00000000 00000000    ................
e0100240: 00000000 00000000 00000000 00000000    ................
e0100250: 00000000 00000000 00000000 00000000    ................
e0100260: 00000000 00000000 00000000 00000000    ................
e0100270: 00000000 00000000 00000000 00000000    ................
e0100280: ffffffff ffffffff 00000000 00000000    ................
e0100290: 00000000 00000000 00000000 00000000    ................
e01002a0: 00000000 00000000 00000000 00000000    ................
e01002b0: 00000000 00000000 00000000 00000000    ................
e01002c0: 00000000 00000000 00000000 00000000    ................
e01002d0: 00000000 00000000 00000000 00000000    ................
e01002e0: 00000000 00000000 00000000 00000000    ................
e01002f0: 00000000 00000000 00000000 00000000    ................

[FriendlyLEG-TINY210]# md 0xe0100300
e0100300: 14131440 00000400 00000000 00000000    @...............
e0100310: 99990000 00000000 00070000 00000000    ................
e0100320: 00000000 00000000 00000000 00000000    ................
e0100330: 00000000 00000000 00000000 00000000    ................
e0100340: 00000000 00000000 00000000 00000000    ................
e0100350: 00000000 00000000 00000000 00000000    ................
e0100360: 00000000 00000000 00000000 00000000    ................
e0100370: 00000000 00000000 00000000 00000000    ................
e0100380: 00000000 00000000 00000000 00000000    ................
e0100390: 00000000 00000000 00000000 00000000    ................
e01003a0: 00000000 00000000 00000000 00000000    ................
e01003b0: 00000000 00000000 00000000 00000000    ................
e01003c0: 00000000 00000000 00000000 00000000    ................
e01003d0: 00000000 00000000 00000000 00000000    ................
e01003e0: 00000000 00000000 00000000 00000000    ................
e01003f0: 00000000 00000000 00000000 00000000    ................
```

4.3.4　分析 ARMCLK 的产生

```
Q1: OM[0] = 0, SRC->XXTI (24Mhz)        由硬件连线决定
Q2: APLLCON (24M->1000M)        0xE0100100 => 0xa07d0301
        Fout = Fin * MDIV / (PDIV * 2 ^ SDIV-1)
                        = 24M * 0x7d / (3 * 2^0)
                        = 24M * 125 / (3*1) = 1000M
Q3: CLK_SRC0        0xE0100200 => 0x10001111
        bit[0]=1          MUXPLL = 1, Fout_APLL
Q4: CLK_SRC0        0xE0100200 => 0x10001111
        bit[16]=0         Fout_APLL = 1G
Q5: CLK_DIV0        Address = 0xE0100300 => 0x14131440
        APLL_RATIO  [2:0]         n=0+1
        ARMCLK = Fout_APLL/n = 1G/1 = 1Ghz
```

4.3.5　举例: UART 串口时钟 PCLK_PSYS 的生成过程

```
Mux_PSYS: CLK_SRC0, 0xe0100200: 10001111 [24]=>0  Fout_mpll

MPLL: MPLL_CON: 0xa29b0c01 => 667Mhz (p358)
        Equation to calculate the output frequency:
        FOUT = MDIV X FIN / (PDIV X 2^SDIV)
        Fout = (0x29b)*24M/(12 * 2^1) = 667Mhz
Mux_MPLL: CLK_SRC0, 0xe0100200: 10001111 [4]=> 1  Fout_mpll

DIV_HCLKP: CLK_DIV0, 0xE0100300: 14131440 [27:24]=> 0100 = /5 -> 667/5 = 133Mhz
DIV_PCLKP: CLK_DIV0, 0xE0100300: 14131440 [30:28]=> 001 = /2 -> 133/2 = 66Mhz
```

4.4　时钟驱动代码实现

4.4.1　Clock 时钟管理知识点总结

```
时钟管理单元 CMU
        MSYS Domain
                MSYS (Cortex A8, DRAM)
        DSYS Domain
                DSYS (JPEG, IIC_HDMI)
        PSYS Domain
                PSYS (JTAG, NandFlash, USB, IIS, AC97, IIC, PWM Timer, RTC)

        几点结论：
                ARMCLK 进行分频可以得到 HCLK_MSYS, PCLK_MSYS
                不同 domain 域之间, 输出频率的关系 PCLK = HCLK / 2

时钟发生器 Clock Generator
        锁相环 PLL (Phase-Locked Loop)
                APLL, MPLL, EPLL, VPLL
```

```
                倍频公式 Fout = Fin * M/(P * 2^S)
                Fout_mpll / Fin_mpll

        分频器 Divider
                1-2-4-8-16 分频 divider
                Div_out = Div_in / (DIVN + 1)
                DIV_PCLKP        DIV_HCLKP

        二路选通器 Mux
                OM 时钟源选择

输入时钟
        X1: XXTI/XXTO 24Mhz
        X2: XusbXTI 24Mhz
        X3: XhdmiXTI 24Mhz
        X4: XrtcXTI 32.768khz
        24Mhz        12Mhz        常用输入频率

输出时钟
        ARMCLK                          (1Ghz)
        HCLKM / PCLKM        (200Mhz / 100Mhz)
        HCLKD / PCLKD        (166Mhz / 83Mhz)
        HCLKP / PCLKP        (133Mhz / 66Mhz)
        记住 100Mhz = 10ns

Clock 时钟特殊功能寄存器
        PLL_CON - APLL_CON
        CLK_SRC
        CLK_DIVn

经验总结
        PLL Sel 决定用或者不用 PLL 锁相环的输出时钟
        串口的时钟输入，采用 MPLL 的输出，但是也可以用 APLL 的输出
        时钟输出的图表，可以从后往前分析
        24Mhz 时钟的选择 是因为 iROM 是采用 24Mhz 时钟 (P354)

        搜索芯片手册的时候，通过在图标的名字，DIV_HCKLP  DIVHCLKP , HCLKP
        时钟的输出都可以通过软件来控制，输出主频越高，耗电量越高
```

4.4.2 代码举例:

```
1) 设置 ARMCLK 分频因子 从 1 到 8，把 1Ghz 调整为 128Mhz

2) 设置 DIV_PCLKP 分频因子 从 2 到 4，把 PCLK 从 66M 调整为 33M
        修改超级终端把 波特率 改为 57600 来测试 PCLK 的设置是否成功
```

第 5 章

UART 控制器

5.1 串口的硬件连接 (硬件原理图)

```
COM0 接口 DB9 九针公头
pin2: RSRXD0
pin3: RSTXD0
pin5: GND
通过 TxD 发送字符（以字节为单位 5-8bits）
通过 RxD 接收字符（以字节为单位 5-8bits）

RS232 电平：-15v->+15v（+15v-逻辑0, -15v-逻辑1）
TTL 电平： 0->+5v（0-逻辑0, 5v-逻辑1）
逻辑电平的转换： MAX3232（美信芯片）

查看 MAX3232 芯片+核心板原理图可得
        RSTXD0 <- XuTxD0 -- TINY1B B7 -- XuTXD0/GPA0_1
        RSRXD0 -> XuRxD0 -- TINY1B B8 -- XuRXD0/GPA0_0

结论： GPA0 管理了 UART 的 Txd/Rxd 两个引脚
```

5.2 串口的管脚功能复用

参考 S5PV210芯片手册

```
GPA0 Mux Function
查看 GPA0[0] GPA0[1], 确认了 UART 的复用功能
查看 GPA0CON 寄存器，了解如何设置（0010）
GPA0CON ： 地址 0xe0200000

[FriendlyLEG-TINY210]# md 0xe0200000
e0200000: 22222222 000000ab 00005555 00000000    """"....UU......
e0200010: 00000000 00000000 00005555 00000000    ........UU......

[FriendlyLEG-TINY210]# mw 0xe0200000 0x22222202

结论：RXD 影响接收， TXD 影响发送
```

RTS和CTS对发送和接收暂时无影响（无流控制）

5.3 串口时序图

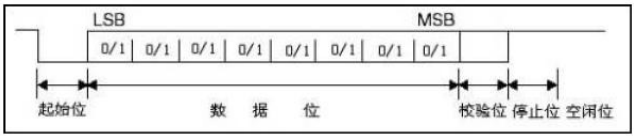

图 5.1: UART Timing

```
Timing:
        空闲状态 high-level
        起始位 start bit - 1 bit
        数据位 data bit - 8bit
                奇偶校验 odd/even Parity（无）
        停止位 stop bit - 1 bit
```

5.4 串口控制器结构

5.4.1 串口控制器功能

UART Controller (p853-p882)

```
类似是一个函数，需要了解它的->输入，输出，如何实现
        1. 输出：Timing（串行通信实现时序图）
                Serial I/O Frame Timing Diagram (Normal UART)  ->p860
        2. 输入：SFR（串口控制器的寄存器）
                REGISTER DESCRIPTION  ->p864
        3. 实现：Block Diagram（结构框图）
                Block Diagram of UART  ->p854
```

5.4.2 串口控制器框图

Block Diagram:

```
        Peripheral Bus 外设总线
                *(int *)SFR_ADDR = value;
        Controll Unit 控制单元
                Control Regs（数据位，停止位，奇偶校验位，时钟源选择，工作模式等）
        Baud-Rate Gengenrator 波特率发生器
                Clock Source 时钟源（PCLK=66M）
```

```
        Transmitter 发送器
                Transmit shifter 发送移位器
                Transmit buffer 发送队列FIFO缓冲器
        Receiver 接收器
                Receiver shifter 接收移位器
                Receiver buffer 接收队列FIFO缓冲器
```

5.5 串口寄存器配置

5.5.1 串口寄存器分类 SFR:

```
15 Regs         Register  Address
控制类          通常是可读可写，属性 R/W        6个
        ULCON0  0xE290_0000
        UCON0  0xE290_0004
        UFCON0  0xE290_0008
        UMCON0  0xE290_000C
        UBRDIV0  0xE290_0028
        UDIVSLOT0  0xE290_002C

状态类          通常是只读，属性 R              4个
        UTRSTAT0  0xE290_0010
        UERSTAT0  0xE290_0014
        UFSTAT0  0xE290_0018
        UMSTAT0  0xE290_001C

数据类          通常是可读可写，属性 R/W        2个
        UTXH0  0xE290_0020
        URXH0  0xE290_0024

中断类          通常是可读可写，属性 R/W        3个
        UINTP0  0xE290_0030
        UINTSP0  0xE290_0034
        UINTM0  0xE290_0038
```

5.5.2 查看 uboot 对串口寄存器的设置

```
[FriendlyLEG-TINY210]# md 0xe2900000
e2900000: 00000003 00000245 00000000 00000000    ....E...........
e2900010: 00000000 00000000 00010000 00000010    ................
e2900020: 00000000 0000000d 00000023 00000808    ........#.......
e2900030: 00000005 00000005 00000000 00000000    ................
e2900040: 00000000 00000000 00000000 00000000    ................
e2900050: 00000000 00000000 00000000 00000000    ................
e2900060: 00000000 00000000 00000000 00000000    ................
e2900070: 00000000 00000000 00000000 00000000    ................
e2900080: 00000000 00000000 00000000 00000000    ................
e2900090: 00000000 00000000 00000000 00000000    ................
```

```
ULCON0          0xE290_0000         00000003
UCON0                  0xE290_0004         00000245
UFCON0          0xE290_0008         0
UMCON0          0xE290_000C         0
UBRDIV0          0xE290_0028         00000023
UDIVSLOT0          0xE290_002C         00000808

其中 FIFO control & Modem control 可以不用设置
Timing setting
        ULCON0        0x3
                data bit: 8bit
                stop bit: 1bit
                parity: none
        UCON0        0x245
                enable Transmit & Receive MODE: 01 01        (INT&Polling)
                interrupt: disable
                dma: disable

CLOCK setting
        UCON0        00:  PCLK (66M)
                --> 115200 bps (bit/second)
                --> 波特率并不是串口控制器的工作频率，串口控制器在接收采样时，是波特率的16倍

        66Mhz = 66000000hz

        分频因子 = 66000000 / (115200*16) - 1
        (分频因子 + 1) = PCLK/(bps*16)

        UBRDIV0:  0x23 = (66000000)/(115200*16)-1= 35
        UDIVSLOT0:
```

5.6　串口驱动代码实现

5.6.1　uart.c 参考代码实现

```
// uart.c
#define ULCON0          (*(volatile unsigned int *)0xE2900000)
#define UCON0                  (*(volatile unsigned int *)0xE2900004)
#define UTRSTAT0          (*(volatile unsigned int *)0xE2900010)
#define UTXH0                  (*(volatile unsigned char *)0xE2900020)
#define URXH0                  (*(volatile unsigned char *)0xE2900024)
#define UBRDIV0         (*(volatile unsigned int *)0xE2900028)
#define UDIVSLOT0          (*(volatile unsigned int *)0xE290002C)

void uart_init(void)
{
        // 66Mhz / (115200*16) - 1 = 0x23
        // 66Mhz / (19200*16) - 1 = 0xD5
        //UBRDIV0 = 0xD5;
        return;
}
```

```
char uart_getchar(void)
{
        char c;
        // polling receive status: if buffer is full
        //while ((UTRSTAT0 & (1<<0)) == 0)
        while (!(UTRSTAT0 & (1<<0)))
                ;

        c = URXH0;

        return c;
}

void uart_putchar(char c)
{
        // polling transmit status: if buffer is empty
        //while ((UTRSTAT0 & (1<<2)) == 0)
        while (!(UTRSTAT0 & (1<<2)))
                ;

        UTXH0 = c;

        return;
}
```

5.6.2 uart.h 参考代码实现

```
// uart.h
void uart_init(void);

char uart_getchar(void);

void uart_putchar(char c);
```

第 6 章

SDRAM 控制器

6.1 SDRAM 硬件连接

6.1.1 SDRAM 引脚描述

硬件原理图

```
1）从芯片角度
        地址线        A0-A13         14根
                BA0, BA1, BA2
        数据线        DQ0-DQ7 * 4 = 32bit data bus (8bit * 4chips)
        控制线        nCS, nRAS, nCAS

2）从处理器角度
        地址线        Xm1ADDR0-Xm1ADDR13
                Xm1BA0, Xm1BA1, Xm1CSn1/BA2
        数据线        Xm1DATA0-Xm1DATA31
        控制线        Xm1CSn0, Xm1RASn, Xm1CASn
```

6.1.2 SDRAM 内部结构

芯片手册

```
1Gbit = 128MByte

内部分 8 bank, 每个bank = 128M/8 = 16M（24根地址线）

24根地址线的信号分为行地址和列地址
        行地址：14根
        列地址：10根
```

6.1.3 从SoC芯片到SDRAM芯片

```
128M * 4 chips = 512M 存储容量 (512M = 2^29)

29根地址线 (A0-A28)
        A28,A27,A26 : BA2,BA1,BA0
        A25-A12: 行地址 14根
        A11-A2: 列地址 10根
        A1, A0, - GND

如何访问 0x0 地址
        A31,A30,A29 : 000        -> nCS 片选无效
        A28,A27,A26 : 0 00
        A25-A12: 00 0000 0000 0000
        A11-A2: 0000 0000 00
        A1, A0: 00

如何访问 0x20000000 地址
        A31,A30,A29 : 001        -> nCS 片选有效
        A28,A27,A26 : 0 00
        A25-A12: 00 0000 0000 0000
        A11-A2: 0000 0000 00
        A1, A0: 00

如何访问 0x21000000 地址
        A31,A30,A29 : 001        -> nCS 片选有效
        A28,A27,A26 : 0 00
        A25-A12: 01 0000 0000 0000
        A11-A2: 0000 0000 00
        A1, A0: 00

如何访问 0x3FFFFFFF 地址
        A31,A30,A29 : 001        -> nCS 片选有效
        A28,A27,A26 : 1 11
        A25-A12: 11 1111 1111 1111
        A11-A2: 1111 1111 11
        A1, A0: 11

如何访问 0x40000000 地址
        A31,A30,A29 : 010        -> nCS 片选无效
        A28,A27,A26 : 0 00
        A25-A12: 00 0000 0000 0000
        A11-A2: 0000 0000 00
        A1, A0: 00
```

6.2 SDRAM 管脚功能复用

搜索芯片数据手册，发现所有管脚均无功能复用。

```
举例
Xm1ADDR[0]
```

```
Xm1DATA[0]
Xm1SCLK
Xm1RASn         Xm1CASn
Xm1CSn[0]
```

6.3 SDRAM 时序图

Timing:

http://www.cnblogs.com/iqstudy/articles/2034422.html

http://www.cnblogs.com/adamite/archive/2010/05/22/1422792.html

http://blog.sina.com.cn/wangdxlove

http://blog.sina.com.cn/s/blog_5755d4b70100b3o0.html

http://blog.chinaunix.net/uid-9012903-id-3056317.html

http://blog.163.com/hanozi@126/blog/static/1865756200897105453/

6.4 SDRAM 控制器结构

http://www.doc88.com/p-47549556518.html

6.5 SDRAM 寄存器配置

```
[FriendlyLEG-TINY210]# md 0xF0000000
f0000000: 0ff02330 00202400 20e00323 00e00323    0#...$ .#.. #...
f0000010: 00110400 ff000000 79101003 00000086    ...........y....
f0000020: 00000000 00000000 ffff00ff 00000000    ................
f0000030: 00000618 2b34438a 24240000 0bdc0343    .....C4+..$$C...
f0000040: 00001db7 00000000 60000000 00000000    ...........`....
f0000050: 000005b2 00000000 00000000 00000000    ................
f0000060: 00000000 00000000 00000000 00000000    ................
f0000070: 00000000 00000000 00000000 00000000    ................
f0000080: 00000000 00000000 00000000 00000000    ................
f0000090: 00000000 00000000 00000000 00000000    ................
f00000a0: 00000000 00000000 00000000 00000000    ................
f00000b0: 00000000 00000000 00000000 00000000    ................

DDR 寄存器配置参数
        data width: 32bit
        row address bit: 14bit
        col address bit: 10bit
        memory type: DDR2
        number of banks: 8banks (DDR chip)
        Average Periodic Refresh Interval: 0x618
                7.8us * 200Mhz = 1560 = 0x618
                7.8us : 刷新周期
        Timing 时间参数
                ...
```

6.6 SDRAM 驱动代码实现

```
/*
 *  armboot - Memory Initialize Code for S5PV210/ARM-Cortex CPU-core
 *
 *  Copyright (c) 2009  Samsung Electronics
 *
 *
 * See file CREDITS for list of people who contributed to this
 * project.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 *
 * Base codes by scsuh  (sc.suh)
 * Modified By JhoonKim (jhoon_kim@nate.com), aESOP Embedded Forum(http://www.aesop.or.kr)
 * 10.08.15 - To Supported for SEC K4T1G164QX DDR2 Memory for aESOP S5PV210
 */

/* #include <config.h>
*/
/*#include "s5pc110.h"
#include "tiny210.h"
*/


/*
 * SDRAM Controller
 */
#define APB_DMC_0_BASE                        0xF0000000
#define APB_DMC_1_BASE                        0xF1400000
#define ASYNC_MSYS_DMC0_BASE                0xF1E00000

#define ELFIN_GPIO_BASE                        0xE0200000


#define DMC0_MEMCONTROL           0x00202400     // MemControl     BL=4, 1Chip, DDR2 Type, dynamic self refresh, force precharge
#define DMC0_MEMCONFIG_0     0x20E00323     // MemConfig0     256MB config, 8 banks,Mapping Method[12:15]0:linear, 1:linterlea
#define DMC0_MEMCONFIG_1        0x00E00323        // MemConfig1

#define      DMC1_MEMCONTROL           0x00202400     // MemControl     BL=4, 2 chip, DDR2 type, dynamic self refresh, force pre
#define DMC1_MEMCONFIG_0     0x40F00313     // MemConfig0     512MB config, 8 banks,Mapping Method[12:15]0:linear, 1:linterlea
```

```
#define DMC1_MEMCONFIG_1        0x00F00313        // MemConfig1

#define DMC0_TIMINGA_REF     0x00000618    // TimingAref  7.8us*133MHz=1038(0x40E), 100MHz=780(0x30C), 20MHz=156(0x9C), 10MHz=
#define DMC0_TIMING_ROW         0x2B34438A      // TimingRow    for @200MHz
#define DMC0_TIMING_DATA        0x24240000      // TimingData   CL=3
#define DMC0_TIMING_PWR         0x0BDC0343      // TimingPower

#define DMC1_TIMINGA_REF     0x00000618    // TimingAref  7.8us*133MHz=1038(0x40E), 100MHz=780(0x30C), 20MHz=156(0x9C), 10MHz=
#define DMC1_TIMING_ROW         0x2B34438A      // TimingRow    for @200MHz
#define DMC1_TIMING_DATA        0x24240000      // TimingData   CL=3
#define DMC1_TIMING_PWR         0x0BDC0343      // TimingPower

        .globl mem_ctrl_asm_init
mem_ctrl_asm_init:

#ifndef CONFIG_EVT1

        ldr     r0, =ASYNC_MSYS_DMC0_BASE

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

        /* This register is removed at EVT1 of C110. */
        ldr     r1, =0x0
        str     r1, [r0, #0xC]

#endif

/* #ifdef CONFIG_MCP_SINGLE */
#if 1

#define MP1_0DRV_SR_OFFSET                0x3CC
#define MP1_1DRV_SR_OFFSET                0x3EC
#define MP1_2DRV_SR_OFFSET                0x40C
#define MP1_3DRV_SR_OFFSET                0x42C
#define MP1_4DRV_SR_OFFSET                0x44C
#define MP1_5DRV_SR_OFFSET                0x46C
#define MP1_6DRV_SR_OFFSET                0x48C
#define MP1_7DRV_SR_OFFSET                0x4AC
#define MP1_8DRV_SR_OFFSET                0x4CC


#define MP2_0DRV_SR_OFFSET                0x4EC
#define MP2_1DRV_SR_OFFSET                0x50C
#define MP2_2DRV_SR_OFFSET                0x52C
#define MP2_3DRV_SR_OFFSET                0x54C
#define MP2_4DRV_SR_OFFSET                0x56C
#define MP2_5DRV_SR_OFFSET                0x58C
#define MP2_6DRV_SR_OFFSET                0x5AC
#define MP2_7DRV_SR_OFFSET                0x5CC
#define MP2_8DRV_SR_OFFSET                0x5EC


        /* DMC0 Drive Strength (Setting 2X) */
        ldr        r0, =ELFIN_GPIO_BASE
```

```
        ldr         r1, =0x0000AAAA
        str         r1, [r0, #0x3cc]
        str         r1, [r0, #MP]
        str         r1, [r0, #MP1_0DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP1_1DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP1_2DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP1_3DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP1_4DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP1_5DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP1_6DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP1_7DRV_SR_OFFSET]

        ldr         r1, =0x00002AAA
        str         r1, [r0, #MP1_8DRV_SR_OFFSET]


        /* DMC1 Drive Strength (Setting 2X) */

        ldr         r0, =ELFIN_GPIO_BASE

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP2_0DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP2_1DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP2_2DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP2_3DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP2_4DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP2_5DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
        str         r1, [r0, #MP2_6DRV_SR_OFFSET]

        ldr         r1, =0x0000AAAA
```

```
        str        r1, [r0, #MP2_7DRV_SR_OFFSET]

        ldr        r1, =0x00002AAA
        str        r1, [r0, #MP2_8DRV_SR_OFFSET]


#define DMC_CONCONTROL                        0x00
#define DMC_MEMCONTROL                        0x04
#define DMC_MEMCONFIG0                        0x08
#define DMC_MEMCONFIG1                        0x0C
#define DMC_DIRECTCMD                        0x10
#define DMC_PRECHCONFIG                0x14
#define DMC_PHYCONTROL0                0x18
#define DMC_PHYCONTROL1                0x1C
#define DMC_RESERVED                     0x20
#define DMC_PWRDNCONFIG                0x28
#define DMC_TIMINGAREF                        0x30
#define DMC_TIMINGROW                        0x34
#define DMC_TIMINGDATA                        0x38
#define DMC_TIMINGPOWER                0x3C
#define DMC_PHYSTATUS                       0x40
#define DMC_CHIP0STATUS                0x48
#define DMC_CHIP1STATUS                0x4C
#define DMC_AREFSTATUS                        0x50
#define DMC_MRSTATUS                     0x54
#define DMC_PHYTEST0                     0x58
#define DMC_PHYTEST1                     0x5C
#define DMC_QOSCONTROL0                0x60
#define DMC_QOSCONFIG0                        0x64
#define DMC_QOSCONTROL1                0x68
#define DMC_QOSCONFIG1                        0x6C
#define DMC_QOSCONTROL2                0x70
#define DMC_QOSCONFIG2                        0x74
#define DMC_QOSCONTROL3                0x78
#define DMC_QOSCONFIG3                        0x7C
#define DMC_QOSCONTROL4                0x80
#define DMC_QOSCONFIG4                        0x84
#define DMC_QOSCONTROL5                0x88
#define DMC_QOSCONFIG5                        0x8C
#define DMC_QOSCONTROL6                0x90
#define DMC_QOSCONFIG6                        0x94
#define DMC_QOSCONTROL7                0x98
#define DMC_QOSCONFIG7                        0x9C
#define DMC_QOSCONTROL8                0xA0
#define DMC_QOSCONFIG8                        0xA4
#define DMC_QOSCONTROL9                0xA8
#define DMC_QOSCONFIG9                        0xAC
#define DMC_QOSCONTROL10                0xB0
#define DMC_QOSCONFIG10                0xB4
#define DMC_QOSCONTROL11                0xB8
#define DMC_QOSCONFIG11                0xBC
#define DMC_QOSCONTROL12                0xC0
#define DMC_QOSCONFIG12                0xC4
#define DMC_QOSCONTROL13                0xC8
#define DMC_QOSCONFIG13                0xCC
#define DMC_QOSCONTROL14                0xD0
```

```
#define DMC_QOSCONFIG14                0xD4
#define DMC_QOSCONTROL15                0xD8
#define DMC_QOSCONFIG15                0xDC


/*
 * SDRAM Controller
 */
#define APB_DMC_0_BASE                      0xF0000000
#define APB_DMC_1_BASE                      0xF1400000
#define ASYNC_MSYS_DMC0_BASE               0xF1E00000

#define DMC_CONCONTROL                       0x00
#define DMC_MEMCONTROL                       0x04
#define DMC_MEMCONFIG0                       0x08
#define DMC_MEMCONFIG1                       0x0C
#define DMC_DIRECTCMD                       0x10
#define DMC_PRECHCONFIG                0x14
#define DMC_PHYCONTROL0                0x18
#define DMC_PHYCONTROL1                0x1C
#define DMC_RESERVED                     0x20
#define DMC_PWRDNCONFIG                0x28
#define DMC_TIMINGAREF                       0x30
#define DMC_TIMINGROW                       0x34
#define DMC_TIMINGDATA                       0x38
#define DMC_TIMINGPOWER                0x3C
#define DMC_PHYSTATUS                       0x40
#define DMC_CHIP0STATUS                0x48
#define DMC_CHIP1STATUS                0x4C
#define DMC_AREFSTATUS                       0x50
#define DMC_MRSTATUS                     0x54
#define DMC_PHYTEST0                     0x58
#define DMC_PHYTEST1                     0x5C
#define DMC_QOSCONTROL0                0x60
#define DMC_QOSCONFIG0                       0x64
#define DMC_QOSCONTROL1                0x68
#define DMC_QOSCONFIG1                       0x6C
#define DMC_QOSCONTROL2                0x70
#define DMC_QOSCONFIG2                       0x74
#define DMC_QOSCONTROL3                0x78
#define DMC_QOSCONFIG3                       0x7C
#define DMC_QOSCONTROL4                0x80
#define DMC_QOSCONFIG4                       0x84
#define DMC_QOSCONTROL5                0x88
#define DMC_QOSCONFIG5                       0x8C
#define DMC_QOSCONTROL6                0x90
#define DMC_QOSCONFIG6                       0x94
#define DMC_QOSCONTROL7                0x98
#define DMC_QOSCONFIG7                       0x9C
#define DMC_QOSCONTROL8                0xA0
#define DMC_QOSCONFIG8                       0xA4
#define DMC_QOSCONTROL9                0xA8
#define DMC_QOSCONFIG9                       0xAC
#define DMC_QOSCONTROL10                0xB0
#define DMC_QOSCONFIG10                0xB4
#define DMC_QOSCONTROL11                0xB8
```

```
#define DMC_QOSCONFIG11                0xBC
#define DMC_QOSCONTROL12                0xC0
#define DMC_QOSCONFIG12                0xC4
#define DMC_QOSCONTROL13                0xC8
#define DMC_QOSCONFIG13                0xCC
#define DMC_QOSCONTROL14                0xD0
#define DMC_QOSCONFIG14                0xD4
#define DMC_QOSCONTROL15                0xD8
#define DMC_QOSCONFIG15                0xDC

        /* DMC0 initialization at single Type*/
        ldr        r0, =APB_DMC_0_BASE

     ldr     r1, =0x00101000                    @PhyControl0 DLL parameter setting, manual 0x00101000
        str        r1, [r0, #DMC_PHYCONTROL0]

       ldr       r1, =0x00000086                          @PhyControl1 DLL parameter setting, LPDDR/
LPDDR2 Case
        str        r1, [r0, #DMC_PHYCONTROL1]

        ldr        r1, =0x00101002                             @PhyControl0 DLL on
        str        r1, [r0, #DMC_PHYCONTROL0]

        ldr        r1, =0x00101003                             @PhyControl0 DLL start
        str        r1, [r0, #DMC_PHYCONTROL0]

find_lock_val:
        ldr        r1, [r0, #DMC_PHYSTATUS]               @Load Phystatus register value
        and        r2, r1, #0x7
        cmp        r2, #0x7                               @Loop until DLL is locked
        bne        find_lock_val

        and        r1, #0x3fc0
        mov        r2, r1, LSL #18
        orr        r2, r2, #0x100000
        orr        r2 ,r2, #0x1000

        orr        r1, r2, #0x3                               @Force Value locking
        str        r1, [r0, #DMC_PHYCONTROL0]

#if 0        /* Memory margin test 10.01.05 */
        orr        r1, r2, #0x1                               @DLL off
        str        r1, [r0, #DMC_PHYCONTROL0]
#endif
        /* setting DDR2 */
        ldr        r1, =0x0FFF2010                             @ConControl auto refresh off
        str        r1, [r0, #DMC_CONCONTROL]

     ldr     r1, =DMC0_MEMCONTROL               @MemControl BL=4, 1 chip, DDR2 type, dynamic self refresh, force precharge, dyn
        str        r1, [r0, #DMC_MEMCONTROL]

     ldr     r1, =DMC0_MEMCONFIG_0               @MemConfig0 256MB config, 8 banks,Mapping Method[12:15]0:linear, 1:linterlea
        str        r1, [r0, #DMC_MEMCONFIG0]

        ldr        r1, =DMC0_MEMCONFIG_1                       @MemConfig1
        str        r1, [r0, #DMC_MEMCONFIG1]
```

```
    ldr        r1, =0xFF000000                              @PrechConfig
    str        r1, [r0, #DMC_PRECHCONFIG]

ldr     r1, =DMC0_TIMINGA_REF              @TimingAref     7.8us*133MHz=1038(0x40E), 100MHz=780(0x30C), 20MHz=156(0x9C
    str        r1, [r0, #DMC_TIMINGAREF]

    ldr        r1, =DMC0_TIMING_ROW                      @TimingRow        for @200MHz
    str        r1, [r0, #DMC_TIMINGROW]

    ldr        r1, =DMC0_TIMING_DATA                      @TimingData        CL=4
    str        r1, [r0, #DMC_TIMINGDATA]

    ldr        r1, =DMC0_TIMING_PWR                      @TimingPower
    str        r1, [r0, #DMC_TIMINGPOWER]

    ldr        r1, =0x07000000                              @DirectCmd        chip0 Deselect
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x01000000                              @DirectCmd        chip0 PALL
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x00020000                              @DirectCmd        chip0 EMRS2
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x00030000                              @DirectCmd        chip0 EMRS3
    str        r1, [r0, #DMC_DIRECTCMD]

ldr     r1, =0x00010400                    @DirectCmd     chip0 EMRS1 (MEM DLL on, DQS# disable)
    str        r1, [r0, #DMC_DIRECTCMD]

ldr     r1, =0x00000542                     @DirectCmd     chip0 MRS (MEM DLL reset) CL=4, BL=4
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x01000000                              @DirectCmd        chip0 PALL
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x05000000                              @DirectCmd        chip0 REFA
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x05000000                              @DirectCmd        chip0 REFA
    str        r1, [r0, #DMC_DIRECTCMD]

 ldr       r1, =0x00000442                          @DirectCmd       chip0 MRS (MEM DLL unreset)
    str        r1, [r0, #DMC_DIRECTCMD]

 ldr       r1, =0x00010780                           @DirectCmd       chip0 EMRS1 (OCD default)
    str        r1, [r0, #DMC_DIRECTCMD]

  ldr       r1, =0x00010400                             @DirectCmd       chip0 EMRS1 (OCD exit)
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x07100000                              @DirectCmd        chip1 Deselect
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x01100000                              @DirectCmd        chip1 PALL
```

```
        str        r1, [r0, #DMC_DIRECTCMD]

        ldr        r1, =0x00120000                              @DirectCmd        chip1 EMRS2
        str        r1, [r0, #DMC_DIRECTCMD]

        ldr        r1, =0x00130000                              @DirectCmd        chip1 EMRS3
        str        r1, [r0, #DMC_DIRECTCMD]

      ldr      r1, =0x00110400                    @DirectCmd      chip1 EMRS1 (MEM DLL on, DQS# disable)
        str        r1, [r0, #DMC_DIRECTCMD]

      ldr      r1, =0x00100542                     @DirectCmd      chip1 MRS (MEM DLL reset) CL=4, BL=4
        str        r1, [r0, #DMC_DIRECTCMD]

        ldr        r1, =0x01100000                              @DirectCmd        chip1 PALL
        str        r1, [r0, #DMC_DIRECTCMD]

        ldr        r1, =0x05100000                              @DirectCmd        chip1 REFA
        str        r1, [r0, #DMC_DIRECTCMD]

        ldr        r1, =0x05100000                              @DirectCmd        chip1 REFA
        str        r1, [r0, #DMC_DIRECTCMD]

       ldr       r1, =0x00100442                          @DirectCmd       chip1 MRS (MEM DLL unreset)
        str        r1, [r0, #DMC_DIRECTCMD]

       ldr       r1, =0x00110780                           @DirectCmd       chip1 EMRS1 (OCD default)
        str        r1, [r0, #DMC_DIRECTCMD]

       ldr       r1, =0x00110400                             @DirectCmd       chip1 EMRS1 (OCD exit)
        str        r1, [r0, #DMC_DIRECTCMD]

        ldr        r1, =0x0FF02030                              @ConControl        auto refresh on
        str        r1, [r0, #DMC_CONCONTROL]

        ldr        r1, =0xFFFF00FF                              @PwrdnConfig
        str        r1, [r0, #DMC_PWRDNCONFIG]

      ldr      r1, =0x00202400                    @MemControl      BL=4, 1 chip, DDR2 type, dynamic self refresh, force precharge
        str        r1, [r0, #DMC_MEMCONTROL]

        /* DMC1 initialization */
        ldr        r0, =APB_DMC_1_BASE

        ldr        r1, =0x00101000                              @Phycontrol0 DLL parameter setting
        str        r1, [r0, #DMC_PHYCONTROL0]

        ldr        r1, =0x00000086                              @Phycontrol1 DLL parameter setting
        str        r1, [r0, #DMC_PHYCONTROL1]

        ldr        r1, =0x00101002                              @PhyControl0 DLL on
        str        r1, [r0, #DMC_PHYCONTROL0]

        ldr        r1, =0x00101003                              @PhyControl0 DLL start
        str        r1, [r0, #DMC_PHYCONTROL0]
find_lock_val1:
```

```
        ldr        r1, [r0, #DMC_PHYSTATUS]               @Load Phystatus register value
        and        r2, r1, #0x7
        cmp        r2, #0x7                               @Loop until DLL is locked
        bne        find_lock_val1

        and        r1, #0x3fc0
        mov        r2, r1, LSL #18
        orr        r2, r2, #0x100000
        orr        r2, r2, #0x1000

        orr        r1, r2, #0x3                               @Force Value locking
        str        r1, [r0, #DMC_PHYCONTROL0]

#if 0        /* Memory margin test 10.01.05 */
        orr        r1, r2, #0x1                               @DLL off
        str        r1, [r0, #DMC_PHYCONTROL0]
#endif

        /* settinf fot DDR2 */
        ldr        r0, =APB_DMC_1_BASE

        ldr        r1, =0x0FFF2010                              @auto refresh off
        str        r1, [r0, #DMC_CONCONTROL]

     ldr     r1, =DMC1_MEMCONTROL               @MemControl     BL=4, 2 chip, DDR2 type, dynamic self refresh, force precharge
        str        r1, [r0, #DMC_MEMCONTROL]

     ldr     r1, =DMC1_MEMCONFIG_0               @MemConfig0     512MB config, 8 banks,Mapping Method[12:15]0:linear, 1:linte
        str        r1, [r0, #DMC_MEMCONFIG0]

        ldr        r1, =DMC1_MEMCONFIG_1                       @MemConfig1
        str        r1, [r0, #DMC_MEMCONFIG1]

        ldr        r1, =0xFF000000
        str        r1, [r0, #DMC_PRECHCONFIG]

     ldr     r1, =DMC1_TIMINGA_REF               @TimingAref     7.8us*133MHz=1038(0x40E), 100MHz=780(0x30C), 20MHz=156(0x9C
        str        r1, [r0, #DMC_TIMINGAREF]

        ldr        r1, =DMC1_TIMING_ROW                        @TimingRow        for @200MHz
        str        r1, [r0, #DMC_TIMINGROW]

        ldr        r1, =DMC1_TIMING_DATA                       @TimingData        CL=3
        str        r1, [r0, #DMC_TIMINGDATA]

        ldr        r1, =DMC1_TIMING_PWR                        @TimingPower
        str        r1, [r0, #DMC_TIMINGPOWER]


        ldr        r1, =0x07000000                              @DirectCmd        chip0 Deselect
        str        r1, [r0, #DMC_DIRECTCMD]

        ldr        r1, =0x01000000                              @DirectCmd        chip0 PALL
        str        r1, [r0, #DMC_DIRECTCMD]

        ldr        r1, =0x00020000                              @DirectCmd        chip0 EMRS2
```

```
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x00030000                              @DirectCmd        chip0 EMRS3
    str        r1, [r0, #DMC_DIRECTCMD]

ldr     r1, =0x00010400                    @DirectCmd     chip0 EMRS1 (MEM DLL on, DQS# disable)
    str        r1, [r0, #DMC_DIRECTCMD]

ldr     r1, =0x00000542                     @DirectCmd     chip0 MRS (MEM DLL reset) CL=4, BL=4
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x01000000                              @DirectCmd        chip0 PALL
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x05000000                              @DirectCmd        chip0 REFA
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x05000000                              @DirectCmd        chip0 REFA
    str        r1, [r0, #DMC_DIRECTCMD]

 ldr       r1, =0x00000442                          @DirectCmd       chip0 MRS (MEM DLL unreset)
    str        r1, [r0, #DMC_DIRECTCMD]

 ldr       r1, =0x00010780                           @DirectCmd       chip0 EMRS1 (OCD default)
    str        r1, [r0, #DMC_DIRECTCMD]

  ldr       r1, =0x00010400                             @DirectCmd       chip0 EMRS1 (OCD exit)
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x07100000                              @DirectCmd        chip1 Deselect
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x01100000                              @DirectCmd        chip1 PALL
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x00120000                              @DirectCmd        chip1 EMRS2
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x00130000                              @DirectCmd        chip1 EMRS3
    str        r1, [r0, #DMC_DIRECTCMD]

ldr     r1, =0x00110440                    @DirectCmd     chip1 EMRS1 (MEM DLL on, DQS# disable)
    str        r1, [r0, #DMC_DIRECTCMD]

ldr     r1, =0x00100542                     @DirectCmd     chip1 MRS (MEM DLL reset) CL=4, BL=4
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x01100000                              @DirectCmd        chip1 PALL
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x05100000                              @DirectCmd        chip1 REFA
    str        r1, [r0, #DMC_DIRECTCMD]

    ldr        r1, =0x05100000                              @DirectCmd        chip1 REFA
    str        r1, [r0, #DMC_DIRECTCMD]
```

```
       ldr       r1, =0x00100442                          @DirectCmd       chip1 MRS (MEM DLL unreset)
        str         r1, [r0, #DMC_DIRECTCMD]

       ldr       r1, =0x00110780                           @DirectCmd       chip1 EMRS1 (OCD default)
        str         r1, [r0, #DMC_DIRECTCMD]

       ldr       r1, =0x00110400                           @DirectCmd       chip1 EMRS1 (OCD exit)
        str         r1, [r0, #DMC_DIRECTCMD]

        ldr         r1, =0x0FF02030                                @ConControl        auto refresh on
        str         r1, [r0, #DMC_CONCONTROL]

        ldr         r1, =0xFFFF00FF                                @PwrdnConfig
        str         r1, [r0, #DMC_PWRDNCONFIG]

     ldr      r1, =DMC1_MEMCONTROL                @MemControl      BL=4, 2 chip, DDR2 type, dynamic self refresh, force precharge
        str         r1, [r0, #DMC_MEMCONTROL]

#else        /* CONFIG_MCP_SINGLE */


#endif        /* CONFIG_MCP_AC / CONFIG_MCP_H / CONFIG_MCP_B / CONFIG_MCP_D */

        mov         pc, lr
```

6.6.1　课堂修改作业

```
1) 修改 uart_init, 把波特率改为 19200
2) 修改 时钟发生器, 把 PCLK 输出改为 33M, 波特率重新计算 115200
3) 实现 puts("hello, world");  输出 "hello, world"
        实现 实现 putchar_hex('a') 十六进制, 输出 0x61
目的：了解 \n换行  \r回车  之间的差别

4) 修改 DRAM Controller,
        把 col address 的宽度改为 9bit
        把 data bit 的32bit 改为 8bit
        用上述 putchar_hex 输出接口, 观察读取DDR内存单元的数值变化
```

第 7 章

NandFlash 控制器

7.1 K9F2G08 芯片

```
48-pin TSOP1 封装
NC - No Connect 不连接，留待扩展（25pin NC）

控制信号：
CLE：Command Latch Enable 命令锁存使能
ALE: Address Latch Enable 地址锁存使能
nCE: Chip Enable 芯片使能（片选）
nRE: Read Enable 读使能
nWE: Write Enable 写使能
nWP: Write Protect 写保护
R/nB: Ready/not Busy 闲/不忙
```

7.2 NandFlash 管脚功能复用

```
Signal Desc.
        IO[7:0]: 无地址线，也无数据线，只有IO线（IO0-IO7）
        CLE: 命令锁存使能
        ALE: 地址锁存使能
        nCE: 芯片使能（片选）
        nRE: 读使能
        nWE: 写使能
        nWP: 写保护
        R/nB: 就绪/忙 信号
```

7.3 Nand Flash Timing 时序

```
READ ID Timing (p31)
        CLE 命令周期 写1次 90h
```

```
        ALE 地址周期 写1次 00h
        DAT 数据周期 读5次 id = 0x EC DA 10 95 44

READ page Timing (p23)
        CLE 命令周期 写1次 00h
        ALE 地址周期 写5次
        CLE 命令周期 写1次 30h
        忙等待周期 R/nB 高有效
        数据周期 读2048次data + 64次ECC
```

7.4 NandFlash 控制器结构

7.5 NandFlash 寄存器配置

7.5.1 NandFlash 寄存器分类

```
SFR 特殊功能寄存器（部分）
控制类
        NFCONF  0xB0E00000
        NFCONT  0xB0E00004
        NFCMMD  0xB0E00008          [7:0] 命令
        NFADDR  0xB0E0000C          [7:0] 地址
数据类
        NFDATA  0xB0E00010          [31:0] 数据
状态类
        NFSTAT  0xB0E00028
```

7.5.2 初始化配置

```
        mw 0xe0200320 0x22222222
        mw 0xb0e00000 0x00006552
        mw 0xb0e00004 0x00c100c5
        mw 0xb0e00008 0x90
        mw 0xb0e0000c 0x00
        md 0xb0e00010

        GPIO 功能复用设置为 NF signal
                ALE: MP0_3[1]
                CLE: MP0_3[0]
                MP0_3CON         Address = 0xE020_0320
                mw 0xe0200320 0x22222222

        NFCONF        0x00001000        -> 0x00006552
                NAND clock = 133M (p363-NFCON) -> 7.5ns (1 clock)
                Nand Timing (k9f2g08.pdf - p13)

                mw 0xb0e00000 0x00006552
```

```
        NFCONT         0x00c100c6  -> 0x00c100c5
                mw 0xb0e00004 0x00c100c5

        NFCMMD:
                mw 0xb0e00008 0x90

        NFADDR:
                mw 0xb0e0000c 0x00

        NFDATA:
                md 0xb0e00010
```

7.6 NandFlash 驱动代码实现

7.6.1 nand.c 参考代码实现

```
// nand.c
#define        NFCONF  (*(volatile unsigned int *)0xB0E00000)
#define        NFCONT  (*(volatile unsigned int *)0xB0E00004)
#define        NFCMMD  (*(volatile unsigned char *)0xB0E00008)
#define        NFADDR  (*(volatile unsigned char *)0xB0E0000C)
#define        NFDATA  (*(volatile unsigned char *)0xB0E00010)
#define        NFSTAT  (*(volatile unsigned int *)0xB0E00028)

#define        MP0_3CON  (*(volatile unsigned int *)0xE0200320)

#define PAGE_SIZE        2048

void nand_init(void)
{
        // [15:12] TACLS = 1 -> (1)         1/133Mhz = 7.5ns
        // [11:8] TWRPH0 = 1 -> (1+1)        7.5ns * 2 = 15ns
        // [7:4] TWRPH1 = 1 -> (1+1)        7.5ns * 2 = 15ns
        NFCONF |= 1<<12 | 1<<8 | 1<<4;

        // AddrCycle  [1]         1 = 5 address cycle
        NFCONF |= 1<<1;

        // MODE  [0]  NAND Flash controller operating mode
        //        0 = Disable NAND Flash Controller
        //        *1 = Enable NAND Flash Controller
        NFCONT |= 1<<0;

        // Reg_nCE0  [1]  NAND Flash Memory nRCS[0] signal control
        //        *0 = Force nRCS[0] to low (Enable chip select)
        //        1 = Force nRCS[0] to High (Disable chip select)
        NFCONT &= ~(1<<1);

        // GPIO functional mux setting
        //                0010 = NF_xxx
        MP0_3CON = 0x22222222;
```

```
        return;
}

void nand_read_id(char id[])
{
        int i;

        // write read_id cmd 90h
        NFCMMD = 0x90;

        // write address 00h
        NFADDR = 0x00;

        for (i = 0; i < 5; i++)
                id[i] = NFDATA;

        return;
}

void nand_read_page(int addr, char buf[])
{
        int i;
        char tmp;

        // write read_page cmd 00h
        NFCMMD = 0x00;

        // write 5 address
        NFADDR = (addr>>0) & 0xFF;
        NFADDR = (addr>>8) & 0x7;
        NFADDR = (addr>>11) & 0xFF;
        NFADDR = (addr>>19) & 0xFF;
        NFADDR = (addr>>27) & 0x1;

        // write read_page cmd 30h
        NFCMMD = 0x30;

        // wait for R/nB -> Ready
        while ((NFSTAT & (1<<0)) == 0)
                ;

        // read data 2048 bytes
        for (i = 0; i < PAGE_SIZE; i++)
                buf[i] = NFDATA;

        for (i = 0; i < 64; i++)
                tmp = NFDATA;

        return;
}

void nand_read(int nand_addr, char * sdram_addr, int size)
{
        int pages = (size - 1)/PAGE_SIZE + 1;
        int i;
```

```
        for (i = 0; i < pages; i++)
                nand_read_page(nand_addr + i*PAGE_SIZE, sdram_addr + i*PAGE_SIZE);

        return;
}
```

7.6.2 nand.h 参考代码实现

```
// nand.h
void nand_init(void);

void nand_read_id(char id[]);

void nand_read_page(int addr, char buf[]);

void nand_read(int nand_addr, char * sdram_addr, int size);
```

7.6.3 思考问题：如何访问 Flash 0地址

```
1. 软件上访问 应该是 计算出 0 地址所在的 page + block
        NFCMMD = 0x00;
        NFADDR = 0x0;
        NFADDR = 0x0;
        NFADDR = 0x0;
        NFADDR = 0x0;
        NFADDR = 0x0;
        NFCMMD = 0x30;
        wait R/nB;
        read NFDATA 2048;

2. 硬件上
        NFCMMD = 0x00;        -> CLE 使能/ALE 禁止, IO[7:0] = 0x00;
        NFADDR = 0x0;        -> ALE 使能/CLE 禁止, IO[7:0] = 0x00;
        NFADDR = 0x0;        -> ALE 使能/CLE 禁止, IO[7:0] = 0x00;
        NFADDR = 0x0;        -> ALE 使能/CLE 禁止, IO[7:0] = 0x00;
        NFADDR = 0x0;        -> ALE 使能/CLE 禁止, IO[7:0] = 0x00;
        NFCMMD = 0x00;        -> CLE 使能/ALE 禁止, IO[7:0] = 0x30;
        read NFSTAT;        -> R/nB 线会设置 STAT 寄存器 0 位
        read NFDATA;        -> nRE 使能, CLE/ALE 禁止, IO[7:0] => NFDATA 中
```

第 8 章

Exception 异常处理

8.1 异常相关基本概念

8.1.1 ARM 的工作模式有几种？各是哪些？

```
7种
        USR: helloworld（非特权->MSR不允许执行）
        SYS: kernel        （2种非异常模式）
        --------------------
        SVC: reset加电     （5种异常模式）
        IRQ: EINT0 key
        FIQ: EINT0 key + MODE(REG)
        ABT: Memory Fault
        UND: execute undefine ins.
```

8.1.2 ARM 的寄存器有多少？ 各是哪些？

```
37个 = 31 通用 + 6 状态
        R0-R7: 未分组（8个）
        R8-R12: 分组（10个=5*2组 FIQ/~FIQ）
        R13-R14: 分组（12个=2*6组 5异常+1非异常）
                R13: SP（stack pointer）压栈/出栈
                R14: LR（Link Register）BL/异常发生后保存PC
        R15: 程序计数器（1个）
                R15: PC（Program Counter）流水线 +8
        -----------------------
        CPSR: 程序状态寄存器 （1个）
        SPSR: 备份的状态寄存器（5个-5种异常）
```

8.1.3 ARM 的异常有几种？各是哪些？

```
7种
```

```
复位异常
数据访问中止
快速中断
快速中断
普通中断
指令预取中止
软件中断： SWI
未定义指令异常： 什么样的指令是未定义？
```

8.2 异常向量表的实现

8.2.1 ARM 的异常向量表是指什么? 有什么特点？

```
异常发生后的入口地址
        从0开始的32字节，7种异常都有入口，0x14保留
        向量一般情况下是指地址，但是异常向量表里面存放是指令
        所有ARM内核的处理器都按上述方式工作
        0x0: reset
        0x8: swi
        0x18: IRQ
        0x1C: FIQ（放在最后，那么好处是可以省一条跳转）
里面存放的是跳转指令
        跳转指令可以是 B（相对，有限）/ BL(不能)
        还可以是 LDR（任意跳转）
                B:              0xEA000000 + offset
                LDR:        0xE59ff000 + offset
```

8.3 异常处理流程

8.3.1 ARM 的软中断异常发生后，硬件做何响应？

```
硬件要做6件事情，以发生SWI异常为例：
        1. 保存（PC-4）-> LR_svc        （PC 是当前执行指令地址+8）
        2. 保存 CPSR -> SPSR_svc
        3. 修改 CPSR -> SVC mode
        4. 修改 CPSR I-bit -> disable IRQ
        5. 映射相应(USR ->) SVC 模式的寄存器
        6. 修改 PC -> 0x8
```

8.3.2 cpu 内核跳转到 0x8 之后，软件需要做哪些工作？

PC 跳转到 0x8 之后，第一个问题是如何返回？ 包括两个返回：

```
1. 地址的返回          mov pc, lr
2. 模式的返回        movs pc, lr
```

```
mov pc, lr        [0xe1a0f00e]
movs pc, lr        [0xe1b0f00e]
```

直接返回没有实际意义，因此第二个问题是如何跳转到 swi_handler ？

```
b   跳转        b    swi_handler
        0xEA000000 | offset        offset计算公式 ( swi_handler - (0x8+8) ) / 4
        offset：代表从 PC 到 目标地址 之间相差的指令数

ldr 跳转        ldr  pc, [pc, offset]
        0xE59FF000 | offset        offset计算公式 ( data_addr - (0x8+8) )
        offset：代表从 PC 到 数据存储地址 之间相差的字节数
       data_addr: 代表 存储目标地址的内存单元的地址，这个地址被看成为一个数据，用ldr从内存中读出来。
```

跳转到 handler 之后， handler 需要做哪些工作?

```
swi_handler 要做以下工作：
        1. 保存现场： r0-r12, r14 压栈
                STMFD r13!, {r0-r12, r14}
        2. 进入异常处理:
                BL        C_swi_handler                ; 用C实现
        3. 恢复现场： r0-r12, pc 出栈
                A) LDMFD r13!, {r0-r12, pc}^
                B) LDMFD r13!, {r0-r12, r14}
                   movs pc, lr
                将原来保存的 spsr 恢复给 CPSR
```

8.4 软中断异常代码实现

8.4.1 New Project

```
start.s -> add to project
main.c         -> add to project
Setting -> ARM Linker -> 1. robase 0x21000000
                         2. layout start.o
                         3. PostLinker fromELF

start.s
        1. 切换模式到 USR = 0xD0
        2. 跳转到 C

main.c
        1. 在C程序中调用软中断的方法
                int __swi(0x1) sys_add(int a, int b, int c);
                ret = sys_add(1, 2, 3);
```

```
        上述代码会编译生成 4 条指令
        [0xe3a02003]   mov      r2,#3
        [0xe3a01002]   mov      r1,#2
        [0xe3a00001]   mov      r0,#1
        [0xef000001]   swi      0x1

2. 在C程序中, 注册 SWI 异常的处理函数
        因为要跳转的地址是在 0x21000000+ 区域,
        因此只能在 0x8 注册一条 LDR 跳转指令, 而不是B指令
        实现办法是：          0x8:  写入一条 0xE59FFxxx (xxx=>020)
                        0x30: 写入 handler 的地址

3. 实现一个真正的在汇编中能够返回模式和地址的 asm_swi_handler
        需要修改 0x30: (int)asm_swi_handler
        需要在 start.s 里面加入一个 asm_swi_handler 的函数
        它的实现应该为：
        asm_swi_handler
                stmfd r13!, {r0-r12, r14}

                bl C_swi_handler

                ldmfd r13!, {r0-r12, r14}
                movs pc, lr

        需要          import C_swi_handler
                export asm_swi_handler

4. 修改 C_swi_handler 使得它能够传递用户参数
int C_swi_handler(int usera, int userb, int userc)
{
        return add(usera, userb, userc);
}

5. 修改 asm_swi_handler        使得它能够漏过返回值给用户
asm_swi_handler
        stmfd r13!, {r1-r12, r14}

        bl C_swi_handler

        ldmfd r13!, {r1-r12, r14}

6. 最后测试时, 把 0x8 处的断点去掉,
从用户看来, 系统调用 sys_add 本质就是一条 swi 指令, 而不是一个函数。
```

第 9 章

Interrupt 控制器

9.1 中断相关基本概念

9.1.1 异常和中断的概念区分

```
异常指的都是内核里面(Cortex-A8)发生的事情
        例如：执行 swi 指令
中断指的都是板子上面(tiny210)发生的事情
        例如：用户按下 K1 按键，
                        定时器Timer(不在板子上，但在芯片里)
联系在于？都有模式的切换，中断处理需要包含异常处理
        异常模式包含(中断)异常模式
```

9.1.2 中断处理的相关概念

```
属于 Controller (Samsung) + Board (tiny210)
```

中断源 Interrupt Source

```
VIC PL192 手册下载 - 属于 Controller (Samsung)
        http://infocenter.arm.com/help/topic/com.arm.doc.ddi0273a/DDI0273.pdf
```

中断控制器 Interrupt Controller

```
中断模式的响应和恢复，属于 Core (ARM)
Cortex-A8 Core (ARM) 手册下载 -
        http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf
```

内核 ARM Core

```
S3C4510 - ARM7TDMI
S3C2440 - ARM920T
S3C6410 - ARM1176
S5PV210 - Cortex A8
```

常见的 Soc - ARM Core

9.2 中断处理流程

9.2.1 哪些事情硬件做，哪些事情软件做？

```
1. 中断触发（用户/用户设置/外部数据）
        中断触发条件的初始化操作--软件有关
2. 中断响应（从当前位置跳转到 0x18）
        保存原来的模式和返回的地址--硬件有关
3. 中断发生后的(老的)现场保存和恢复
        压栈和出栈--软件有关
4. 中断返回
        恢复原来的模式和地址--软件有关
5. 中断标志位 SFR Pending bit
        pending bit 的设置--硬件有关（控制器寄存器PND）
        pending bit 的清除--软件有关（控制器寄存器PND）
        作用：识别中断源/调用相应的handler--软件有关(可以交给VIC实现)
6. 中断允许位 CPSR I-bit
        加电之后，I-bit 的初始状态是 关闭/禁止 的（0xD3->0x53）
        MSR/MRS 可以允许使能--软件有关（内核寄存器CPSR）
7. IRQ发生 -> Pending bit -> I-bit        enable -> 跳转
        (跳转之前，硬件会完成 disable I-bit，
                因此不再响应后继任何中断，直到软件恢复CPSR)
8. 在软件恢复CPSR(重新允许中断)之前，软件需要清除之前的pending bit(VIC+ADDR寄存器=0)

内核和处理器升级的过程，就是不断把原来软件的工作交给硬件来做。
中断的优先级
中断的处理程序 ISR -> 中断的向量表
```

9.2.2 如何跳转

```
        IC                              VIC
-----------------------------------------------------------------
        b irq_handler0                      VectADDR -> PC
        LDR pc, =irq_handler0               VIC interface(A31-A0)
-----------------------------------------------------------------
```

9.3 中断寄存器配置

9.3.1 中断相关寄存器的设计演变

```
               IC                      IC                      Vectored IC ->

               ARM7(4510)              ARM9(2440)              ARM11(6410) & A8(210)
-------------------------------------------------------------------------------------------
               CPSR I-bit              CPSR I-bit              CPSR I-bit
内核                                                           VIC Port(Enable)
(Core)                                                           VIC interface(PC<->A0-A31)
-------------------------------------------------------------------------------------------
                                    INTOFFSET                    VectADDRESS(32bit->A0-A31)
                                                          Vectors(handlers)
                                    INTPRI                       Priority
               INTMOD                     INTPND                      (IRQ/FIQ)STATUS(PND)
中断               INTPND                      INTMOD                      SELECT(MOD) IRQ/FIQ
控制器               INTMSK                      INTMSK                      ENABLE(MSK)
(IC)                                  SRCPND                     RAWINTR(SRC)
-------------------------------------------------------------------------------------------
                                                          INTMSK
                                                          INTPND(clear)
               EINTCON                    EINTCON                     EINTCON
中断源               (F/R/L)                    (F/R/L)                     (F/R/L)
控制器               GPXCON                     GPXCON                      GPXCON
(GPIO)               (EINT)                     (EINT)                      (EINT)
-------------------------------------------------------------------------------------------
硬件层                Key/UART/USB/Timer
```

9.3.2 S5PV210 中断相关寄存器

```
中断源 GPIO Controller
        GPH2CON[0]   [3:0]   Set the pin mux function as EXT_INT
                0000 = Input
                0001 = Output
                0010 = Reserved
                0011 = KP_COL[0]
                0011 ~ 1110 = Reserved
        *        1111 = EXT_INT[16]

        EXT_INT_2_CON[0]  [2:0]  Sets the signaling method of EXT_INT[16]
                000 = Low level
                001 = High level
        *        010 = Falling edge triggered
                011 = Rising edge triggered
                100 = Both edge triggered
                101 ~ 111 = Reserved

        EXT_INT_2_MASK[0]  [0]
        *        0 = Enables Interrupt
                1 = Masked

        EXT_INT_2_PEND[0]  [0]  (R)
```

```
                0 = Not occur
                1 = Occur interrupt

向量中断控制器 Vectored Interrupt Controller
        VIC0RAWINTR  0xF200_0008 R
                Specifies the Raw Interrupt Status Register
                RawInterrupt  [31:0]  Shows the status of the FIQ interrupts before masking by the
                        VICINTENABLE and VICINTSELECT Registers:
                        0 = Interrupt is inactive before masking
                        1 = Interrupt is active before masking

        VIC0INTENABLE  0xF200_0010 R/W
                Specifies the Interrupt Enable Register
                IntEnable  [31:0]  Enables the interrupt request lines
                        Write:
                        0 = No effect
        *                1 = Enables Interrupt.

        VICINTENCLEAR
                IntEnable Clear
                        Write:
                        0 = No effect
        *                1 = Disables Interrupt in VICINTENABLE Register.

        VIC0IRQSTATUS  0xF200_0000 R
                Specifies the IRQ Status Register
                IRQStatus  [31:0]  Shows the status of the interrupts after masking by the
                        VICINTENABLE and VICINTSELECT Registers:
                        0 = Interrupt is inactive
                        1 = Interrupt is active.

        VIC0INTSELECT  0xF200_000C R/W
                Specifies the Interrupt Select Register
                IntSelect  [31:0]  Selects interrupt type for interrupt request:
        *                0 = IRQ interrupt
                        1 = FIQ interrupt

        VIC0VECTADDR0 0xF200_0100 R/W
                Specifies the Vector Address 0 Register
                VICVECTADDR[0-31]  Bit  Description
                        VectorAddr 0-31  [31:0]  Contains ISR vector addresses.

        VIC0VECTPRIORITY0  0xF200_0204 R/W
                Specifies the Vector Priority 1 Register
                VectPriority  [3:0]  Selects vectored interrupt priority level. You can select
                        any of the 16 vectored interrupt priority levels by
                        programming the register with the hexadecimal value of
                        the priority level required, from 0-15.

        VIC0ADDRESS  0xF200_0F00 R/W
                Specifies the Vector Address Register
                VectAddr  [31:0]
                Contains the address of the currently active ISR

内核
        CPSR I-bit
```

```
        __asm
        {
                mov r0, #0x53
                msr CPSR_cxsf, r0
        }

VIC Enable (p15)
        __asm
        {
                mrc p15, 0, r0, c1, c0, 0
                orr r0, r0, #(1<<24)
                mcr p15, 0, r0, c1, c0, 0
        }
```

9.4 硬件中断异常代码实现

9.4.1 实验验证结论：

```
第一阶段，观察 GPIO 控制器里面的 pending bit 设置情况

        VIC0(IRQ/FIQ)STATUS        标识中断是否通过(IRQ/FIQ)
        VIC0INTSELECT                选择IRQ还是FIQ
        VIC0INTENABLE                使能中断通过
        VIC0RAWINTR                通过 EXT_INT_2_MASK 之后的情况
        --------------------
        EXT_INT_2_MASK                中断 Mask bit
        EXT_INT_2_PEND                中断 Pending bit
        EXT_INT_2_CON                下升沿触发 Falling Edge
        GPH2CON                        管脚复用 EXT_INT[16]

第二阶段，如何清除 pending bit ？
        写1清除 pending bit
        写0无效

        3种清0的写法，只有最后一种是正确的清除。
        PEND |= 1<<0;                (not good)
        PEND = 0xFFFFFFFF;        (not good)
        PEND = 1<<0;                (Good!)

        VIC0RAWINTR 寄存器取决于 EXT_INT_2_PEND 的值，如 PEND 是 1 ，则通过 MASK 之后它也是 1；
        如果 PEND 做了清除，则相应的 VIC0RAWINTR 中的位，也随之清除；无需手工清除该寄存器的位。

第三阶段，观察中断控制器中的使能Enable和状态Status标识寄存器
        使能 Enable 寄存器在 RAW 和 STATUS 之间
        STATUS 寄存器的标识位无需软件清除，只要清除 PENDing 位，该状态位自动清 0
        SELECT 寄存器决定该中断是 IRQ 还是 FIQ，从而硬件会设置不同的 STATUS 寄存器

第四阶段，如何打开 CPSR I-bit ？
        通过 汇编 MSR         (SVC: 0xD3        1101->0101 0x53->CPSR)
        __asm
        {
                mov         r0, #0x53
```

```
                msr        cpsr, r0
        }

第五阶段，中断发生了之后怎么办？
        接下来有2种处理办法：
        A）简单的办法就是使用 VIC 向量中断控制器的功能
                1. 跳转的地址向量要提前设置好
                2. 通知内核，启用 VIC Port 功能
                紧接着的问题是，如何在执行完 beep 之后返回主程序？
                        原因： beep 程序不能作为 IRQ_handler
                        真正的 IRQ_handler 应该要完成
                        1）保存cpu 现场 STMFD
                        2）清除掉 Pending bit，调用 beep
                        3）恢复cpu 现场 LDMFD

                修改 start.s , 实现 IRQ_handler
                        1）IRQ 模式下的 sp 指针需要初始化
                        2）除了清除pending bit 之外，还需要清除 VIC0ADDRESS = 0;
                        3）返回地址 lr 寄存器需要 减4 即 sub lr, lr, #4
                        (lr-4)->PC          SPSR->CPSR

        B）复杂的办法就是不用 VIC , 自己实现全过程
                1. 当 IRQ 异常发生的时候，cpu 跳转到 0x18
                2. 背景知识：reset 0 地址被映射 map 到 iROM（内容不可修改）
                        0 地址 在 iROM 中（0xD0000000）
                                iRAM（0xD0020000）-> 0x20000 （iROM 被映射到了 0x20000）
                        通过 md 命令，查看相关内存单元值，发现 0x18: 0xEA000018
                        经过一系列分析，最终在 iROM 中的跳转指令会加载从 0xD0037400 地址开始的值，
                        作为异常发生后要跳转的地址+offset , 因此只需要修改 0xD0037418 的向量即可。
                3. (int)IRQ_handler -> 0xD0037400 + 0x18
                        如果是 SWI 软件中断，则在 0xD0037408 处填写swi_handler的地址
                4. 参考代码： https://github.com/limingth/ARM-Codes/tree/master/tiny210-codes/A-INT-
demo
```

第 10 章

PWM Timer 定时器

10.1 定时器工作原理

10.1.1 定时器功能

* 计时
* 中断
* PWM Timer（驱动PWM信号的设备）

10.1.2 原理

* 类似于以前的 “沙漏”
* 沙漏的计时原理：沙子量，漏沙的速度，到时的反转(连续)
  沙子量： Counter的初值
  漏沙速度：counter （自动完成，并且依据Clock=PCLK=66Mhz）
  反转：reload 操作（InitValue -> Counter）
  装沙子：Manual update
* 真正的硬件设计是怎样的？ TCNTBn - 用来装沙子的量筒，可以修改 TCNTn - 真正用来做 counter ，不可修改，不可见 TCNTOn - 可以用来观察 TCNTn 的值的变化

10.1.3 课堂讨论

* 控制器的工作时钟是怎么产生的?
  - PCLK source
  - level1: 8-bit prescaler
  - level2: 2/4/8/16 divider
* 定时中断是怎么产生的?
  - If the down-counter reaches zero,
  - Timer Count Buffer register (TCNTBn)

- down-counter
- set the timer enable bit of TCON
- set interrrupt enable bit

* PWM 信号是怎么产生的?

- value of the TCMPBn register
- down-counter value matches the value of the compare register
- reload function

10.2 定时器寄存器配置

```
Register  Address
TCFG0         0xE250_0000         65
        PCLK = 66M          66000000 -> 分频
        Timer Input Clock Frequency = PCLK / ( {prescaler value + 1} ) / {divider value}
                {prescaler value} = 1~255
                {divider value} = 1, 2, 4, 8, 16, TCLK
        66M/66(65+1)         = 1M = 1000000

TCFG1          0xE250_0004         0100=0x4
        1M/16 = 62500

TCON          0xE250_0008
        *[3] Timer 0 Auto Reload on/off
                        0 = One-Shot
                        1 = Interval Mode(Auto-Reload)
        [2]         Timer 0 Output Inverter on/off
                        0 = Inverter Off
                        1 = TOUT_0 Inverter-On
        *[1]         Timer 0 Manual Update
                        0 = No Operation
                        1 = Update TCNTB0,TCMPB0
        *[0] Timer 0 Start/Stop
                        0 = Stop
                        1 = Start Timer 0

TCNTB0  0xE250_000C
        val = 62500 (=1s)

TCMPB0  0xE250_0010

TCNTO0  0xE250_0014
        read value -> TCNT0's value

TINT_CSTAT, R/W, Address = 0xE250_0044)
        Interrupt Control and Status Register
        Timer 0 Interrupt Status
```

```
                [5]  Timer 0 Interrupt Status Bit.
                Clears by writing '1' on this bit.

        Timer 0 interrupt Enable
                [0]   Enables Timer 0 Interrupt.
                        1 = Enables
                        0 = Disabled
```

10.3 定时器驱动代码实现

```
// init Timer
// step 0: setup clock TCFG0+TCFG1
// step 1: TCNTBn <- init value
// setp 2: Set the manual update bit
//                 and clear only manual update bit
//                enable auto-reload bit
// step 3: Set the start bit
// step 4: Enable interrupt

while (1)
{
        // polling TCNTO0, putint_hex()

        // polling TINT_CSTAT bit[5] == 1 ?
                beep();
}

// init VIC
Timer0 SRC = bit[21] belong to VIC0

Timer0 中断传递相关寄存器
VIC0(IRQ/FIQ)STATUS        标识中断是否通过(IRQ/FIQ)
VIC0INTSELECT                选择IRQ还是FIQ: bit21
VIC0INTENABLE                使能中断通过: bit21
VIC0RAWINTR                        通过Timer0之后的情况 bit21
---------------------
TINT_CSTAT                        中断(MSK) Enable bit [0]
TINT_CSTAT                        中断(PND) Status bit [5]
TCNTB0                                触发方式 1 sec interrupt
        (TCFG0 + TCFG1) clock init
-GPD0CON                        管脚复用 TOUT0
```

第 11 章

Linux 驱动开发基础

11.1 驱动基本概念

11.1.1 设备

计算机最基本的三个组成部分：CPU、内存及其输入输出(I/O)设备。我们说的设备驱动中的设备就是输入输出设备。

常见的设备有：键盘、鼠标、串口、声卡、显卡、网卡、SD、flash、IDE、USB、PCI….

CPU与这些设备的接口就是输入/输出。CPU从这些设备上获取数据叫做输入，CPU将数据写入到设备上就是输出。例如对硬盘的读写。

以上的设备中键盘、鼠标是我们常见的数据设备。用来接受用户的输入。串口是一个数据传输设备，工作原理相对简单而且工作稳定。

声卡、显卡是常见的多媒体设备,主要是输出信息给用户。

网卡是我们常见的网络设备，也是计算机里非常重要的设备之一。它的作用主要用来在计算机之间的通讯。

SD卡、flash、IDE都是存储设备，具有容量较大、断电数据不丢失等特点。

USB、PCI都是总线协议驱动，他们的驱动不针对详细设备。注意：这里USB总线不包含USB设备（例如:USB鼠标驱动就是一个设备，而不是一个总线驱动）。

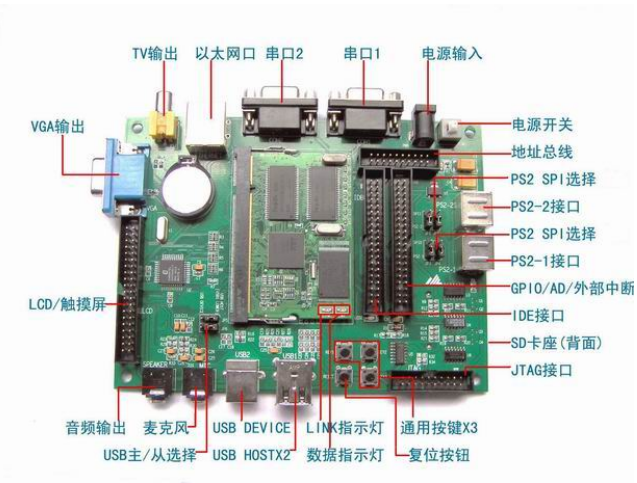

图 11.1: MC2410E开发板1

设备驱动的一般结构：

```
SoC（主芯片 -> 设备控制器 -> 外设引脚
--------------------------------------
设备相关芯片（DM9000/WM8960/MAX3232）
--------------------------------------
设备接插件（RJ45/Speaker/DB9）
```

11.1.2 设备驱动

我们可以写个简单的驱动来访问设备。可以不在任何具有操作系统的裸机上，也可以在bootloader里。例如：串口，网卡等。 但常见的驱动运行在Linux,Windows操作系统上。我们这节课要研究的就是Linux操作系统上的驱动。

Linux操作系统的驱动与裸机（或者bootloader） 上的驱动有很多的不同。

* 分层
  要考虑与应用层的接口，例如：应用程序获取键盘输入；
* 并发
  考虑多用户，例如：几个程序都在访问串口；
* 协议
  考虑其他的协议，例如：网络协议；

等等还有很多细节区别。关于这个区别的细节后面章节将会有描述。

设备、驱动和操作系统三者之间的关系是：

* 驱动是提供操作系统访问硬件的接口；
* 设备可以通过产生中断通知操作系统有数据到来或者发送；
* 驱动是操作系统内核和硬件之间的一个中间接口和媒介；
* 内核通过驱动来最终控制硬件；
* 操作系统中的驱动和设备的关系是一一对应的；
* 应用和驱动是一对多关系；

从图中我们可以看出应用并不是直接和设备驱动进行交互，而是通过抽象层统一的系统调用接口和驱动交互。

操作系统中的驱动的任务

* 具有一般驱动的操作功能：初始化设备，读写设备；
* 将设备的数据分配给应用；例如：网卡驱动，控制台驱动；
* 将用户数据分配给设备；例如：读写硬盘上的文件；

用户程序和操作系统，驱动程序之间是如何实现关联的？ 这就需要了解设备文件这个概念。

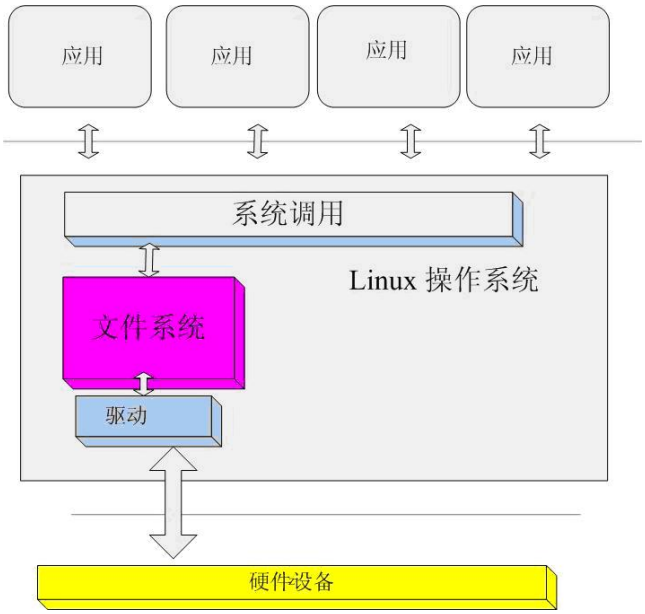

图 11.2: Linux设备驱动层次结构

11.1.3 设备文件

Unix（Linux是类Unix系统）操作系统从一开始就将设备看作文件，通过操作文件的接口统一操作设备。 Linux上大部分设备都有对应的设备文件；应用程序可以通过设备文件访问设备。

Linux通过设备驱动程序为应用程序提供了统一抽象的接口，从而隐藏了大量不同设备之间的区别和细节。 在Linux中对硬件设备的操作和通常的文件一样，利用标准的文件操作可以对设备上进行打开、关闭、读取或者写入操作。 系统中的每个设备由“设备特殊文件”来代表。 通过/dev访问驱动程序，/dev目录下的文件可用来访问驱动程序

```
ls /dev/
```

```
/dev/hda        系统中的第一个IDE硬盘
/dev/hdc        光驱
/dev/sdb        系统中的u盘
/dev/mmcblk       系统中的sd卡
/dev/ttyS0       PC 串口设备
/dev/ttySAC0       板子上的串口设备
/dev/tty0-6       虚拟控制台
/dev/zero        软件设备
/dev/null        空设备
/dev/urandom       随机数字
/dev/fb0        LCD显示屏
/dev/dsp        音频声卡
```

常用设备文件名

常用设备文件操作方法 重定向符 > 写设备

```
向串口写入数据：echo data > /dev/ttyS0
向LCD写入数据：cat sunflower.bmp > /dev/fb0
向声卡写入数据：cat ringing.wav > /dev/dsp
```

cat 命令 读设备

```
从串口读取数据： cat /dev/ttyS0 （PC Linux）
从串口读取数据： cat /dev/ttySAC0 （板子 Linux）
```

dd 命令 读写设备

```
将文件写入串口中
dd if=1.txt of=/dev/ttyS0

写入0x00到2.txt中，每次读写的数据量是512个字节，写入2次
dd if=/dev/zero        of=2.txt bs=512 count=2

随机显示
dd if=/dev/uramdom of=/dev/fb0

播放 1.wav 文件
dd if=1.wav of=/dev/dsp

查找 wav 文件及其大小的命令
find . -name *.wav -exec ls -l {} \;
```

11.2 硬件基础知识

11.2.1 处理器

内核

* 主流处理器体系结构
  ARM
  MIPS
  PowerPC
  68K/ColdFire
  X86
* 冯·诺伊曼结构和哈佛结构
  冯·诺伊曼结构
  程序指令存储器和数据存储器合并在一起的存储器结构。
  程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置，
  因此程序指令和数据的宽度相同。

哈佛结构

将程序指令和数据分开存储，指令和数据可以有不同的数据宽度。

独立的程序总线和数据总线，分别作为 CPU 与每个存储器之间的专用通信路径，具有较高的执行效率。

* RISC vs CISC

RISC

精简指令集计算机，减少指令条数、指令单周期执行。

CISC

复杂指令集计算机，指令复杂，指令周期长。

控制器和特殊功能寄存器

* 常用控制器

内存控制器 MemC

中断控制器 IntC

定时器 Timer

DMA控制器 DMAC

串口控制器 UARTC

* 特殊功能寄存器

控制类 CON，CFG，DIV，MSK

数据类 DAT，TXH/RXH，CNTO

状态类 STAT，PND

11.2.2 存储

MMU 虚拟地址和物理地址

* MMU

Memory Management Unit 存储管理单元

* CP15

ARM 协处理器

* TLB

Translation Lookaside Buffer，即转换旁路缓存，是转换表的Cache，因此也经常被称为“快表” 。

ref: http://blog.sina.com.cn/s/blog_5d6836440100bfgg.html
http://www.cnblogs.com/timkyle/archive/2012/03/09/2388384.html

* MMU/Core/SoC/SDRAM 之间的关系

ProcessCore = ALU + Regsister(PC，R0-R15，CPSR)

CORE = ProcessCore + CP15 + MMU + Cache

SoC = CORE + SFR + iROM + iRAM

SoC < bus > SDRAM

* 重要结论

无论在用户应用程序，还是在内核模块中，打印出的变量（全局和局部），函数名，代表的都是虚拟地址。

裸板编程中用到的地址，从数据手册中得到的地址，都是没有启用MMU的，代表的都是物理地址。

LDR/STR 命令中，涉及到的内存地址，从本质上说，都是虚拟地址；
当MMU没有启用的情况下，上面的虚拟地址就等于物理地址；如果启动MMU，这些虚拟地址就会被映射为不同的物理地址。

内核空间和用户空间

* X86 Linux 内存设计
  0 – 3G 用户空间 (0x0 – 0xC0000000)
  3G – 4G 系统空间 (0xC0000000 – 0xFFFFFFFF)
* ARM Linux 内存设计
  0 – (3G–16M) 用户空间 (0x0 – 0xBF000000)
  (3G–16M) – 3G Kernel Modules (0xBF000000 – 0xC0000000)
  3G – 4G 系统空间 (0xC0000000 – 0xFF000000)

ref: http://blog.csdn.net/hzpeterchen/article/details/5363518

11.2.3 常见接口和总线

* 串口
* I2C总线
* USB总线
* ISA总线
* PCI总线
* 以太网接口

计算机接口大全
http://www.technibble.com/articlecontent/2009/07/computer-hardware-chart1.jpg

11.3 开发环境搭建

11.3.1 准备工作

* 获得参考资料
  http://github.com/limingth/ARM-Lessons/tree/master/CortexA8-s5pv210-20120901
* 驱动源码
  http://github.com/limingth/ARM-Lessons/tree/master/CortexA8-s5pv210-20120901/tiny210/drivers
* 应用程序
  http://github.com/limingth/ARM-Lessons/tree/master/CortexA8-s5pv210-20120901/tiny210/examples
* 相关书籍
  - 《linux设备驱动程序》（第3版） http://oss.org.cn/kernel-book/ldd3/index.html
  - 《深入分析Linux内核源码》 http://oss.org.cn/kernel-book/index.htm

  - 《1inux设备驱动开发详解》（第2版）

  - 《1inux内核设计与实现》（第2版）

* 常用网站

  - http://lxr.free-electrons.com/ 在线阅读Linux内核源码，查找内核符号的定义和引用

  - http://kernelbook.sourceforge.net/kernel-api.html/ 查找哪些函数可以在模块中使用。

11.3.2 安装交叉编译器

http://github.com/limingth/ARM-Tools/tree/gh-pages/dev

```
tar zxvf arm-linux-gcc-4.5.1-v6-vfp-20120301.tgz -C /usr/bin
export PATH=$PATH:your-install-dir
vi ~/.bashrc        -> 修改 PATH 环境变量
which arm-linux-gcc
```

```
/* led.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
        int on;
        int led_no;
        int fd;

        if (argc != 3 || sscanf(argv[1], "%d", &led_no) != 1 || sscanf(argv[2],"%d", &on) != 1 ||
                        on < 0 || on > 1 || led_no < 0 || led_no > 3) {
                fprintf(stderr, "Usage: leds led_no 0|1\n");
                exit(1);
        }

        fd = open("/dev/leds0", 0);
        if (fd < 0) {
                fd = open("/dev/leds", 0);
        }
        if (fd < 0) {
                perror("open device leds");
                exit(1);
        }
```

```
        ioctl(fd, on, led_no);
        close(fd);

        return 0;
}
```

编写应用程序

```
arm-linux-gcc led.c -o led
file led
```

编译生成可执行程序

11.3.3 搭建测试环境

```
$ rx xxx        (通过xmodem协议)
xmodem -> 2 spaces -> 1 space + enter
```

方法1 串口上传

方法2 ftp 上传　配置网络连通

```
1. 有ip吗？没有则用ifconfig；
        sudo /etc/init.d/networking restart
2. 能ping通自己ip吗？
3. 能ping通网关ip吗？        不能则换网线试试；
4. 能ping通 8.8.8.8 吗？
        不能则sudo route add default gw 192.168.x.x 设置默认网关；
5. 能ping通www.google.com 吗？
        不能则修改 /etc/resolv.conf 配置dns , 添加nameserver 8.8.8.8；
```

设置开发板 ip 地址

```
vi /etc/eth0-setting
IP = 192.168.0.201
Gateway = 192.168.0.1
DNS = 192.168.0.1
```

设置主机 ip 地址

```
sudo ifconfig eth0 192.168.0.200
```

ftp 上传

```
$ ftp 192.168.0.201
Name: root
Password: root
ftp> binary
ftp> put led
ftp> quit
```

ftp 脚本

vi ftp.sh

```
#!/bin/sh
DIR=$1
FILE=$2
ftp -i -in <<!
open 192.168.0.201 21
user username password
cd /home
lcd $DIR
binary
put $FILE
bye
!
ls -l $DIR/$FILE
```

test ftp.sh

```
chmod 777 ftp.sh
./ftp.sh leds led
        (leds is dir name, led is file name)
./ftp.sh . led
        (if ftp.sh is in same diretory as led file)
```

11.4 开发调试流程

11.4.1 基本流程

编译应用程序 http://github.com/limingth/ARM-Lessons/blob/master/CortexA8-s5pv210-20120901/tiny210/examples/leds/led.c

http://github.com/limingth/ARM-Lessons/blob/master/CortexA8-s5pv210-20120901/tiny210/examples/buttons/buttons_test.c

http://github.com/limingth/ARM-Lessons/blob/master/CortexA8-s5pv210-20120901/tiny210/examples/pwm/pwm_test.c

```
$ make led
$ make buttons_test
$ make pwm_test
```

```
/* mini210_leds.c */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/moduleparam.h>
#include <linux/slab.h>
#include <linux/ioctl.h>
#include <linux/cdev.h>
#include <linux/delay.h>

#include <mach/gpio.h>
#include <mach/regs-gpio.h>
#include <plat/gpio-cfg.h>


#define DEVICE_NAME "leds"

static int led_gpios[] = {
        S5PV210_GPJ2(0),
        S5PV210_GPJ2(1),
        S5PV210_GPJ2(2),
        S5PV210_GPJ2(3),
};

#define LED_NUM                ARRAY_SIZE(led_gpios)


static long mini210_leds_ioctl(struct file *filp, unsigned int cmd,
                unsigned long arg)
{
        switch(cmd) {
                case 0:
                case 1:
                        if (arg > LED_NUM) {
                                return -EINVAL;
                        }

                        gpio_set_value(led_gpios[arg], !cmd);
                        //printk(DEVICE_NAME": %d %d\n", arg, cmd);
                        break;

                default:
```

```
                        return -EINVAL;
        }

        return 0;
}

static struct file_operations mini210_led_dev_fops = {
        .owner                        = THIS_MODULE,
        .unlocked_ioctl        = mini210_leds_ioctl,
};

static struct miscdevice mini210_led_dev = {
        .minor                        = MISC_DYNAMIC_MINOR,
        .name                        = DEVICE_NAME,
        .fops                        = &mini210_led_dev_fops,
};

static int __init mini210_led_dev_init(void) {
        int ret;
        int i;

        for (i = 0; i < LED_NUM; i++) {
                ret = gpio_request(led_gpios[i], "LED");
                if (ret) {
                        printk("%s: request GPIO %d for LED failed, ret = %d\n", DEVICE_NAME,
                                        led_gpios[i], ret);
                        return ret;
                }

                s3c_gpio_cfgpin(led_gpios[i], S3C_GPIO_OUTPUT);
                gpio_set_value(led_gpios[i], 1);
        }

        ret = misc_register(&mini210_led_dev);

        printk(DEVICE_NAME"\tinitialized\n");

        return ret;
}

static void __exit mini210_led_dev_exit(void) {
        int i;

        for (i = 0; i < LED_NUM; i++) {
                gpio_free(led_gpios[i]);
        }

        misc_deregister(&mini210_led_dev);
}

module_init(mini210_led_dev_init);
module_exit(mini210_led_dev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("FriendlyARM Inc.");
```

编译设备驱动

```
ftp.sh        网络传输
rx 串口传送
```

下载至开发板上

```
insmod led.ko
printk 输出调试信息
```

加载驱动

```
调用 ioctl 点亮 led 灯
./led

编译 buttons_test.c , 运行按键测试程序
./buttons_test

编译 pwm_test.c , 运行蜂鸣器测试程序
./pwm_test
```

测试应用

```
$ tar zxvf linux-2.6.35.7-20120829.tar.gz
$ cd linux-2.6.35.7/
$ ls
        Process: kernel/arch
        Memery: mm/
        File: fs/
        Device: drivers/
        Net: net/
$ du -sh

$ find . -name *.c | wc -l

$ make menuconfig
$ sudo aptitude search ncurses

$ sudo apt-get install ncurses-dev

$ make
```

编译驱动内核模块

11.4.2 课堂小练习

完成一个 跑马灯 的应用程序。读取用户按键编号 n，作为启动 跑马灯 之后的循环次数。
要求每次按键按下时，蜂鸣器播放一个对应频率的按键音，循环结束后，蜂鸣器播放一个结束音。

第 12 章

Linux 内核模块

12.1 用户空间编写驱动程序

12.1.1 错误写法

```
/* main.c */

#define UTXH        *(volatile unsigned int *)0xe2900020

int main(void)
{
        while (1)
                UTXH = 'a';

        return 0;
}
```

Makefile limingth@ubuntu:~/led-drv-on-linux$ cat Makefile

```
all:
        arm-linux-gcc main.c -o main

clean:
        -rm *.obj *.o app
```

```
$ ./main
$ Segmentation Fault
```

执行会出段错误

12.1.2　正确写法

```
/* main.c */

#include <stdio.h>
#include <sys/mman.h>                // mmap()
#include <fcntl.h>                // O_RDWR

int global = 100;

int main(void)
{
        int * vmem = (int *)0x0;
        int uart = 0xe2900000;        // 0x20 offset
        int fd;

        fd = open("/dev/mem", O_RDWR);

        printf("fd = %d\n", fd);

        vmem = mmap(0, 1, PROT_READ|PROT_WRITE, MAP_SHARED, fd, uart);

        printf("vmem = %p\n", vmem);

        printf(".text = %p\n", main);
        printf(".data = %p\n", &global);
        printf(".stack = %p\n", &uart);

        sleep(1);

        while (1)
                *(vmem + 0x20/4) = 'a';

        close(fd);

        return 0;
}
```

```
[root@FriendlyARM /home]# ./main
fd = 3
vmem = 0x40020000
.text = 0x8440
.data = 0x11034
.stack = 0xbe9fbce4
aaaaa....
```

输出结果

```
arm-linux-ld -verbose
```

如何链接

```
arm-linux-readelf -a main
```

查看 Section 地址

12.2 内核模块程序结构

12.2.1 内核

在计算机科学中，内核是操作系统最基本的部分。 它是为众多应用程序提供对计算机硬件的安全访问的一部分软件，这种访问是有限的，并且内核决定一个程序在什么时候对某部分硬件操作多长时间。直接对硬件操作是非常复杂的，所以内核通常提供一种硬件抽象的方法来完成这些操作。硬件抽象隐藏了复杂性，为应用软件和硬件提供了一套简洁，统一的接口，使程序设计更为简单。

宏内核结构在硬件之上定义了一个高阶的抽象界面，应用一组原语(或者叫系统调用)来实现操作系统的功能，例如进程管理，文件系统，和存储管理等等，这些功能由多个运行在核心态的模块来完成。 尽管每一个模块都是单独地服务这些操作，内核代码是高度集成的，而且难以编写正确。因为所有的模块都在同一个内核空间上运行，一个很小的bug都会使整个系统崩溃。然而，如果开发顺利，单内核结构就可以从运行效率上得到好处。

宏内核结构的例子:

* 传统的UNIX内核，例如伯克利大学发行的版本
* Linux内核
* MS-DOS，Windows 9x (Windows 95，98，Me)

微内核结构由一个非常简单的硬件抽象层和一组比较关键的原语或系统调用组成，这些原语仅仅包括了建立一个系统必需的几个部分，如 线程管理，地址空间和进程间通信等。 微核的目标是将系统服务的实现和系统的基本操作规则分离开来。例如，进程的输入/输出锁定服务可以由运行在微核之外的一个服务组件来提供。这些非常模块化的用户态服务器用于完成操作系统中比较高级的操作，这样的设计使内核中最核心的部分的设计更简单。一个服务组件的失效并不会导致整个系统的崩溃，内核需要做的，仅仅是重新启动这个组件，而不必影响其它的部分。

微内核结构的例子:

* Mach，用于GNU Hurd和MacOS X
* Minix
* QNX
* VxWorks
* NEXTSTEP http://zh.wikipedia.org/wiki/NEXTSTEP

微内核和宏内核

12.2.2　模块的由来

一般Linux kernel编译后生成一个vmlinux(非压缩格式)或者zImage（压缩格式）文件。 我们统称为kernel image。一般kernel image包含很多驱动，例如：键盘、鼠标、网卡等等。 当我们往kernel里添加一个新的驱动时，例如：声卡驱动。在调试阶段，我们会经常编译整个kernel,然后重启机器来调试驱动。

以上操作很耗时，很繁琐。因此Linux kernel开发者采用了一种的机制，将kernel分成静态的kernel image部分和可动态加载部分。 一般我们调试新的驱动就采用动态加载机制，这样我们无须编译整个kernel也无须重启机器。这就是模块。

```
$ make menuconfig
```

12.2.3　模块的定义

模块是Linux kernel 的一部分（通常是设备驱动）。它能被静态编译到kernel image里面，也可以动态加载。按需动态装入模块使内核空间达到最小。一旦模块被装入到Linux内核，那么它就像任何标准的内核代码一样成为内核的一部分，具有相同的权限和职责。

关于模块的几个描述如下：

* 运行在kernel空间；
* 不能使用printf等用户库的函数；
* 可以动态加载和卸载。

我们来看一个模块的例子：

```
obj-m := hello.o

KDIR := /home/limingth/tiny210/src/linux-2.6.35.7

all:
	make -C $(KDIR)	SUBDIRS=$(PWD)	modules

clean:
	rm -rf *.o *.ko *.mod.* *.cmd
	rm -rf .*
```

Makefile 文件

```
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_AUTHOR("AKAEDU");
MODULE_DESCRIPTION("module example ");
MODULE_LICENSE("GPL");
```

```
int __init akae_init (void)
{

        printk ("Hello, akaedu\n");
        return 0;
}

void __exit akae_exit (void)
{
        printk ("module exit\n");
        return ;
}

module_init(akae_init);
module_exit(akae_exit);
```

hello.c 文件

12.2.4 运行在kernel空间

因为模块是kernel的一部分，而不是一个普通的应用程序，所以它在kernel空间运行。

12.2.5 不能使用库函数

看上面例子中打印”hello，akaedu”。使用的是printk，而不是printf。因为库函数提供的接口是给应用程序使用的，并且运行空间在应用空间。所以类似于printf这样的库函数不能在模块中使用。

关于kernel 的API，我们可以访问 http://kernelbook.sf.net 我们可以通过这个网站查找哪些函数可以在模块中使用。当然最直接的还是通过查看kernel源代码，来获取这些函数。

12.2.6 模块与驱动的关系

模块并不全是驱动，例如我们看的第一个例子，它就是显示信息而已。但大多数模块都是驱动。 后面的课程我们会详细描述驱动模块的写法。

12.2.7 模块的基本元素

```
模块需要有自己的入口函数和退出函数。如下表所示：
int init_module(void)
{
        return 0;
}

void cleanup_module(void)
{
        return;
}
```

函数名　注意函数名称的写法。 特别注意：return 0；不可缺少！！！ 但我们经常会看到kernel里的入口函数如下：

```
/* see linux/init.h */

module_init(akae_init);
module_exit(akae_exit);
```

并没有我们上面描述的函数名称，这是为什么呢？它怎么实现的呢？ 答案很简单：其实 module_init 是一个宏定义。将akae_init函数别名为init_module。 见kernel代码的include/linux/init.h 文件（此处不再重点讲解）。

详见 http://bbs.chinaunix.net/thread-1350305-1-1.html

在linux下的驱动函数都有module__init（x）和module_exit(x)在linux内核里都定义成一个宏例如：

```
#define module_init(x) \
        int init_module(void) __attribute__((alias(#x))); \
        extern inline __init_module_func_t __init_module_inline(void) \
        { return x; }
```

头文件　一般来说Linux kernel的模块都有如下的头文件：

```
#include <linux/module.h>
#include <linux/kernel.h>
```

函数修饰 __init __exit　这2个修饰符起到什么作用，如果不写对内核模块的插入和卸载有影响吗？
（ 提示： cat /proc/kallsyms | grep akae ）

12.3　模块加载和卸载

12.3.1　模块的操作命令

插入模块 insmod　模块编译后需要通过特定的工具加载，这个工具就是insmod. 与其相关的命令还有几个，我们来看看详细的描述：

```
insmod
bash# insmod hello.ko

vi include/generated/utsrelease.h
#define UTS_RELEASE "2.6.35.7-FriendlyARM"
```

查看模块 lsmod 查看当前有多少模块。 lsmod

```
$ lsmod
Module                  Size  Used by
hello                   2304   0
Used by表明有没有用户使用这个模块。如果使用了就不能rmmod它。

cat /proc/modules 查看已经加载的模块信息。
```

删除模块 rmmod 删除已经有的模块。

```
rmmod
$ rmmod hello
```

```
$ modinfo hello.ko
modinfo 命令可以获得模块作者，说明，参数和许可协议等信息。
```

查看模块 modinfo

```
$ modprobe helllo.ko
modprobe，如同 insmod，加载一个模块到内核.
它的不同在于它会查看要加载的模块，看是否它引用了当前内核没有定义的符号.
如果发现有，modprobe 在定义相关符号的当前模块搜索路径中寻找其他模块.
当 modprobe 找到这些模块( 要加载模块需要的 )，它也把它们加载到内核.
如果你在这种情况下使用 insmod 代替，命令会失败，在系统日志文件中留下一条 " unresolved symbols "消息.
```

modprobe 工具

12.3.2 许可协议

内核认识的特定许可有，“GPL”（ 适用 GNU 通用公共许可的任何版本 ），“GPL v2”（ 只适用 GPL 版本 2 ），“GPL and additional rights”，“Dual BSD/GPL”，“Dual MPL/GPL”，和 “Proprietary”. 除非你的模块明确标识是在内核认识的一个自由许可下，否则就假定它是私有的，内核在模块加载时被“弄污浊”了. 象我们在第1章 “许可条款”中提到的，内核开发者不会热心帮助在加载了私有模块后遇到问题的用户.

可以在模块中包含的其他描述性定义有

```
MODULE_AUTHOR ( 声明谁编写了模块 ),
MODULE_DESCRIPION( 一个人可读的关于模块做什么的声明 ),
MODULE_VERSION ( 一个代码修订版本号; 看 <linux/module.h> 的注释以便知道创建版本字串使用的惯例),
MODULE_ALIAS ( 模块为人所知的另一个名子 ),
MODULE_DEVICE_TABLE ( 来告知用户空间，模块支持那些设备 ).
```

12.3.3　命名空间和符号导出

```
cat /proc/kallsyms 查看内核的符号表。

EXPORT_SYMBOL(symbol_name)
```

12.3.4　给模块传参数

Linux2.6允许用户insmod的时候往内核模块里面传递参数，它主要使用module_param宏定义来实现这一功能。

module_param的定义可以在include/linux/moduleparam.h文件里面查看到，它的原型为：

```
module_param(name, type, perm);

name是在模块中定义的变量名称，type是变量的类型，perm是权限掩码，一般填0就可以。

static int howmany = 1;
module_param(howmany, int, 0);
```

12.4　内核模块驱动代码实现

12.4.1　write led.c

```
#include <linux/module.h>
#include <asm/io.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("limingth");
MODULE_DESCRIPTION("led module driver");
MODULE_VERSION("version1.0");
MODULE_ALIAS("myled test");

static int * vmem;
static int led_no = 0;
module_param(led_no, int, 0);

static __init int led_init(void)
{
        printk("led init ok!\n");

//        *(int *)0xe0200284 = 0x0;
        vmem = ioremap(0xe0200284, 4);
```

```
        printk("vmem = %p\n", vmem);
//        *vmem = 0x0;
        *vmem &= ~(1 << led_no);

        return 0;
}

static __exit void led_exit(void)
{
        printk("led exit ok!\n");

//        *(int *)0xe0200284 = 0xf;
        *vmem = 0xf;

        return;
}

module_init(led_init);
module_exit(led_exit);
```

12.4.2 write a Makefile to compile

```
obj-m := led.o

KDIR := /home/akaedu/teacher_li/linux-2.6.35.7

all:
        make modules -C $(KDIR) SUBDIRS=$(PWD)

clean:
        -rm *.ko *.o *.order *.mod.c *.symvers
```

第 1 行
定义生成模块的名称。没有特殊约定时，hello.c 将会成为编译成 hello.c 的源代码文件。

第 3 行
指定内核源代码的位置

第 4 行
指定编译对象模块源代码所在位置的当前目录。

第 7 行
指定编译模块的命令。

第 8 行
利用编译结果清除所有的生成文件。

编译后，会生成许多文件，2.6 内核中生成的模块实际名称为 hello.ko ， 也可以使用 hello.o 。

12.4.3 compile and test it

```
make
make V=1 (verbose)
insmod led.ko led_no=3
rmmod led
```

12.4.4 GPIO

```
/* gpio_drv.c */
#include <linux/module.h>
#include <asm/io.h>

MODULE_LICENSE("GPL");

static int * vmem_led;
static int * vmem_buzzer_con;
static int * vmem_buzzer;
static int * vmem_btn;

void delay(void)
{
        /* tell compiler: do not optimize */
        volatile int i;

        for (i = 0; i < 10000000; i++)
                ;
}

void led_blink(int n)
{
        int i;

        for (i = 0; i < n; i++)
        {
                *vmem_led = 0xF;
                delay();

                *vmem_led = 0x0;
                delay();
        }
}

void buzzer_beep(int n)
{
        int i;

        for (i = 0; i < n; i++)
        {
                *vmem_buzzer = 0x1;
                delay();
```

```
                *vmem_buzzer = 0x0;
                delay();
        }
}

int btn_get_id(void)
{
        int i;

        for (i = 0; i < 4; i++)
        {
                if ((*vmem_btn & (1<<i)) == 0)
                        return i+1;
        }

        return 0;
}

void init_all(void)
{
        vmem_led = ioremap(0xe0200284, 4);
        vmem_buzzer_con = ioremap(0xe02000A0, 4);
        vmem_buzzer = ioremap(0xe02000A4, 4);
        vmem_btn = ioremap(0xe0200C44, 4);

        /* set GPIO CON = output 0001 */
        *vmem_buzzer_con = 0x1;
}

EXPORT_SYMBOL(init_all);
EXPORT_SYMBOL(led_blink);
EXPORT_SYMBOL(buzzer_beep);
EXPORT_SYMBOL(btn_get_id);

/* gpio_test.c */
#include <linux/module.h>
#include <asm/io.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("limingth");
MODULE_DESCRIPTION("led module driver");
MODULE_VERSION("version1.0");
MODULE_ALIAS("myled test");

#if 1
extern void led_blink(int n);
extern void buzzer_beep(int n);
extern int btn_get_id(void);
extern void init_all(void);
#endif

static __init int gpio_init(void)
{
        int btn_id;

        printk("gpio init ok!\n");
```

```
        init_all();

        while (1)
        {
                btn_id = btn_get_id();

                led_blink(btn_id);

                buzzer_beep(btn_id);

                if (btn_id == 4)
                        break;
        }

        printk("finished!\n");

        return 0;
}

static __exit void gpio_exit(void)
{
        printk("gpio exit ok!\n");

        return;
}

module_init(gpio_init);
module_exit(gpio_exit);
```

12.4.5 compile

```
$ make obj-m=gpio_test.o
$ make obj-m=gpio_drv.o
```

12.4.6 Exercise

```
/* uart.c */

#include <linux/module.h>
#include <asm/io.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("limingth");
MODULE_DESCRIPTION("uart module driver");
MODULE_VERSION("version1.0");
MODULE_ALIAS("myuart test");

struct uart_sfr
{
```

```
        int ulcon;
        int ucon;
        int ufcon;
        int umcon;
        int utrstat;
        int uerstat;
        int ufstat;
        int umstat;
        int utxh;
        int urxh;
        int ubrdiv;
        int udivslot;
};

typedef struct uart_sfr USFR;

static volatile USFR *puart;

int uart_putchar(char c)
{
        while ((puart->utrstat & (1<<2)) == 0)
                ;

        puart->utxh = c;

        return 0;
}

char uart_getchar(void)
{
        char c;

        while ((puart->utrstat & (1<<0)) == 0)
                ;

        c = puart->urxh;

        return c;
}

void uart_init(void)
{
        puart = ioremap(0xe2900c00, sizeof(USFR));
        puart->ulcon = 0x3;
        puart->ucon = 0x7c5;
        puart->ubrdiv = 0x23;
        puart->udivslot = 0x808;
}

static __init int uart_mod_init(void)
{
        printk("uart module init ok!\n");

        uart_init();

        while (1)
```

```
        {
                char ch;

                ch = uart_getchar();

                uart_putchar(ch);

                if (ch == 'q')
                        break;
        }

        printk("finished!\n");

        return 0;
}

static __exit void uart_mod_exit(void)
{
        printk("uart module exit ok!\n");

        return;
}

module_init(uart_mod_init);
module_exit(uart_mod_exit);
```

第 13 章

Linux 字符设备驱动

13.1 字符设备驱动结构

13.1.1 设备分类

字符设备 一个字符(char) 设备是一种可以当作一个字节流来访问存取的设备(如同一个文件); 一个字符驱动负责实现这种行为. 这样的驱动常常至少实现 open, close, read, 和 write 系统调用. 文本控制台(/dev/console)和串口(/dev/ttyS0 及其类似的设备)就是两个字符设备的例子, 因为它们很好地展现了流的抽象. 字符设备通过文件系统结点来存取, 例如 /dev/tty1 和 /dev/lp0. 在一个字符设备和一个普通文件之间唯一有关的不同就是, 你经常可以在普通文件中移来移去, 但是大部分字符设备仅仅是数据通道, 你只能顺序存取.然而, 存在看起来象数据区的字符设备, 你可以在里面移来移去. 例如, frame grabber 经常这样, 应用程序可以使用 mmap 或者 lseek 存取整个要求的图像.

块设备 如同字符设备, 块设备通过位于 /dev 目录的文件系统结点来存取. 一个块设备(例如一个磁盘)应该是可以驻有一个文件系统的. 在大部分的 Unix 系统, 一个块设备只能处理这样的 I/O 操作, 传送一个或多个长度经常是 512 字节(或一个更大的 2 的幂的数)的整块. Linux, 相反, 允许应用程序读写一个块设备象一个字符设备一样 它允许一次传送任意数目的字节. 结果就是, 块和字符设备的区别仅仅在内核在内部管理数据的方式上, 并且因此在内核/驱动的软件接口上不同. 如同一个字符设备, 每个块设备都通过一个文件系统结点被存取的, 它们之间的区别对用户是透明的. 块驱动和字符驱动相比, 与内核的接口完全不同.

网络接口 任何网络事务都通过一个接口来进行, 就是说, 一个能够与其他主机交换数据的设备. 通常, 一个接口是一个硬件设备, 但是它也可能是一个纯粹的软件设备, 比如环回接口. 一个网络接口负责发送和接收数据报文, 在内核网络子系统的驱动下, 不必知道单个事务是如何映射到实际的被发送的报文上的. 很多网络连接(特别那些使用 TCP 的)是面向流的, 但是网络设备却常常设计成处理报文的发送和接收. 一个网络驱动对单个连接一无所知; 它只处理报文.

13.1.2 字符设备驱动结构

字符设备是指发送或者接受数据按照字符方式进行, 一般应用程序都通过设备文件来访问字符设备。

字符设备的驱动一般在 kernel-src/drivers/char 目录下。常见的字符设备有：鼠标，控制台，声卡，显示设备，touch panel，串口，并口等等。

我们可以在Linux源代码目录通过 make menuconfig来看到字符设备。

字符设备管理　我们都知道应用程序是通过设备文件来访问字符设备的。那么设备文件通过什么标示来对应相关的驱动呢？这就是我们前面提到的设备号。

因为应用程序要与字符设备进行数据交互（read，write）。那么驱动还要提供读写函数。 并且要求将读写函数与设备号连接起来。这就是我们下面要讲的应用和驱动的关联。

应用和驱动关联　在前面的课程我们谈到了设备文件，给出了设备文件和设备号的概念。这节课我们先看一个例子：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define filename "/dev/akae_c"

int main (void)
{
        int fd;
        char buf[10];
        int ret;

        fd = open (filename,O_RDWR);

        if (fd < 0)
        {
            printf ("Open %s file error,please use sudo !",filename);
            return -1;
        }

        ret = read (fd,buf,10);
        if (ret != 10)
        {
            printf ("NOTICE: read %s file,get %d Bytes\n",filename,ret);
        }

        ret = write (fd,buf,10);
        if (ret != 10)
        {
            printf ("NOTICE: write %s file,get %d Bytes\n",filename,ret);
        }

        close (fd);

        return 0;
}

/* mychar_test.c */

#include <stdio.h>
```

```
#include <fcntl.h>

#include <sys/types.h>
#include <unistd.h>

int main(void)
{
        int fd;
        int local = 100;

        printf("hello, test char drv\n");
        printf("<main> &local = %p\n", &local);

        printf("current pid = %d\n", getpid());

        fd = open("mychar", O_RDWR);
        printf("fd = %d\n", fd);

        if (fd < 0)
        {
                perror("open mychar failed!\n");
                return -1;
        }

        return 0;
}

/* mychar_drv.c */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/sched.h>

MODULE_LICENSE("GPL");

int mychar_open(struct inode * in, struct file * filp)
{
        int local = 100;

        printk("mychar open called!\n");

        printk("current pid = %d\n", current->pid);
        printk("current parent pid = %d\n", current->parent->pid);
        printk("current process name = %s\n", current->comm);
        printk("current open function = %p\n", mychar_open);

        printk("<driver> &local = %p\n", &local);
        printk("task struct size = %d\n", sizeof(*current));

//        while (1);

        printk("mychar open called finished!\n");

        return 0;
}
```

```
struct file_operations mychar_fops =
{
        .open = mychar_open,
};

static __init int mychar_init(void)
{
        int rc;

        printk("mychar init\n");

        // register mychar_drv
        rc = register_chrdev(240, "this is my first char drv", &mychar_fops);

        printk("rc = %d\n", rc);

        return 0;
}


static __exit void mychar_exit(void)
{
        printk("mychar exit\n");

        // unregister mychar_drv
        unregister_chrdev(240, "this is my first char drv");

        return;
}

module_init(mychar_init);
module_exit(mychar_exit);


/* Makefile */
CC = arm-linux-gcc

obj-m := mychar_drv.o
KDIR := /home/akaedu/teacher_li/linux-2.6.35.7/

all:
        make mychar_test
        make -C $(KDIR)        SUBDIRS=$(PWD)         modules
        ls -l *.ko mychar_test

clean:
        -rm *.ko *.o *.order *.mod.c *.symvers
        -rm mychar_test
```

从上例中我们可以看出：

```
* 应用程序可以通过打开设备文件访问设备；
* 可以通过read, write函数访问设备驱动；
```

```
* 设备号是设备的唯一重要标示；
* 多个设备文件可以对应到一个设备号，反之则不行；
* 设备文件名可以根据需要命名，没有硬性规定；
* 设备驱动和设备文件之间，通过设备节点的设备号(id)关联起来，和取名(name)无关，和存放位置(/dev)无关；
```

13.2 主设备号和次设备号

13.2.1 主次设备号

字符设备通过文件系统中的名字来存取. 那些名字称为文件系统的特殊文件，或者设备文件，或者文件系统的简单结点；惯例上它们位于 /dev 目录. 字符驱动的特殊文件由使用 ls -l 的输出的第一列的 “c” 标识. 块设备也出现在 /dev 中，但是它们由 “b” 标识. 本章集中在字符设备，但是下面的很多信息也适用于块设备.

如果你发出 ls -l 命令，你会看到在设备文件项中有 2 个数(由一个逗号分隔)在最后修改日期前面，这里通常是文件长度出现的地方. 这些数字是给特殊设备的主次设备编号. 下面的列表显示了一个典型系统上出现的几个设备. 它们的主编号是 1，4，7，和 10，而次编号是 1，3，5，64，65，和 129.

```
crw-rw-rw- 1 root  root  1,  3 Apr 11  2002 null
crw------- 1 root  root  10, 1 Apr 11  2002 psaux
crw------- 1 root  root  4,  1 Oct 28 03:04 tty1
crw-rw-rw- 1 root  tty   4, 64 Apr 11  2002 ttys0
crw-rw---- 1 root  uucp  4, 65 Apr 11  2002 ttyS1
crw-w---- 1 vcsa  tty   7,  1 Apr 11  2002 vcs1
crw-w---- 1 vcsa  tty   7,129 Apr 11  2002 vcsa1
crw-rw-rw- 1 root  root  1,  5 Apr 11  2002 zero
```

传统上，主编号标识设备相连的驱动. 例如，/dev/null 和 /dev/zero 都由驱动 1 来管理，而虚拟控制台和串口终端都由驱动 4 管理；同样，vcs1 和 vcsa1 设备都由驱动 7 管理. 现代 Linux 内核允许多个驱动共享主编号，但是你看到的大部分设备仍然按照一个主编号一个驱动的原则来组织.

次编号被内核用来决定引用哪个设备. 依据你的驱动是如何编写的(如同我们下面见到的)，你可以从内核得到一个你的设备的直接指针，或者可以自己使用次编号作为本地设备数组的索引. 不论哪个方法，内核自己几乎不知道次编号的任何事情，除了它们指向你的驱动实现的设备.

主设备号是由 include/linux/major.h 定义的。 设备号和设备名可在内核源代码的 Documentation/devices.txt 里查到， mknod 可为这些指定的设备创建节点，当然节点的位置不是一定要在/dev下，但是为了便于管理一般都是指定/dev。

mknod 命令 mknod - make block or character special files

```
mknod [OPTION]... NAME TYPE [MAJOR MINOR]
    option 有用的就是 -m 了
    name   自定义
```

```
type   有 b 和 c 还有 p
主设备号
次设备号
    b        表示特殊文件是面向块的设备（磁盘、软盘或磁带）。
    c        表示特殊文件是面向字符的设备（其他设备）。
    p        创建 FIFO（已命名的管道）。
```

举例

$ mknod mylcd c 29 0 （创建一个自己的lcd设备，主设备号为 29，等价于 /dev/fb0 ）

实验一下，如果用 cp 命令和 mv 命令分别对一个设备文件进行操作，会有什么结果？（结论是 cp 不可用，mv 可以）

13.2.2 注册设备

注册一个字符设备的经典方法是使用：

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
```

这里，major 是感兴趣的主编号，name 是驱动的名字(出现在 /proc/devices)，fops 是缺省的 file_operations 结构. 一个对 register_chrdev 的调用为给定的主编号，并且为每一个建立一个缺省的 cdev 结构.

如果你使用 register_chrdev，从系统中去除你的设备的正确的函数是：

```
void unregister_chrdev(unsigned int major, const char *name);
```

major 和 name 必须和传递给 register_chrdev 的相同，否则调用会失败.

这两个通用方法的声明在 include/linux/fs.h 中，它们的实现主要体现在 fs.h 文件的内联函数和 fs/char_dev.c 中.

13.2.3 调用范例

```
struct file_operations xxx_fops =
{
        .owner = THIS_MODULE,
        .open = xxx_open,
        .read = xxx_read,
        .write = xxx_write,
        .release = xxx_release,
};

xxx_init() 中注册设备
rc = register_chrdev(XXX_MAJOR, "akae", &xxx_fops);

xxx_release() 中卸载设备
unregister_chrdev(XXX_MAJOR, "akae");
```

当前已经注册使用的设备可以通过 cat /proc/devices 文件得到：

```
Character devices:
 1 mem
 2 pty
 3 ttyp
 4 ttyS
 6 lp
 7 vcs
 10 misc
 13 input
 14 sound
 21 sg
 180 usb

Block devices:
 2 fd
 8 sd
 11 sr
 65 sd
 66 sd
```

13.2.4 设备编号的内部表示

在内核中，dev_t 类型(在 linux/types.h 中定义)用来持有设备编号 主次部分都包括. 对于 2.6.0 内核，dev_t 是 32 位的量，12 位用作主编号，20 位用作次编号. 你的代码应当，当然，对于设备编号的内部组织从不做任何假设；相反，应当利用在 linux/kdev_t.h 中的一套宏定义. 为获得一个 dev_t 的主或者次编号，使用：

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

相反，如果你有主次编号，需要将其转换为一个 dev_t，使用：

```
MKDEV(int major, int minor);


/* linux/fs.h */
struct inode {
        ...
        dev_t                   i_rdev;
        ...
};

/* linux/types.h */
typedef __u32 __kernel_dev_t;
typedef __kernel_dev_t                dev_t;

/* linux/kdev_t.h *?
```

```
#define MINORBITS       20
#define MINORMASK       ((1U << MINORBITS) - 1)

#define MAJOR(dev)       ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)       ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)       (((ma) << MINORBITS) | (mi))
```

13.2.5　cdev结构体

在 Linux 2.6 内核中使用 cdev结构体描述字符设备， cdev 结构体的定义在 include/linux/cdev.h 中，内容如下：

```
struct cdev
{
        struct kobject kobj;   /*所属模块*/
        struct module *owner;
        struct file_operations  /*文件操作结构体*/
        struct list_head list;
        dev_t dev;           /*设备号*/
        unsigned int count;
};
```

Linux 2.6 内核提供了一组函数用于操作 cdev 结构体，它们的实现在 fs/char_dev.c 中，声明如下所示：

```
void cdev_init(struct cdev *, struct file_operations *);
struct cdev *cdev_alloc(void);
void cdev_put(struct cdev *p);
int cdev_add(struct cdev *, dev_t, unsigned);
void cdev_del(struct cdev *);
```

cdev_init()函数用于初始化 cdev 的成员，并建立 cdev 和 file_operations 之间的连接。

cdev_init()函数内部实现

```
void cdev_init(struct cdev *cdev, struct file_operations *fops)
{
        memset(cdev, 0, sizeof *cdev);
        INIT_LIST_HEAD(&cdev->list);
        cdev->kobj.ktype = &ktype_cdev_default;
        kobject_init(&cdev->kobj);
        cdev->ops = fops;    /*将传入的文件操作结构体指针赋值给cdev的ops*/
}
```

cdev_alloc()函数用于动态申请一个cdev内存。

```
struct cdev *cdev_alloc(void)
{
        struct  cdev  *p=kmalloc(sizeof(struct  cdev),GFP_KERNEL);  /*分配cdev的内存*/
        if (p) {
                memset(p, 0, sizeof(struct cdev));
                p->kobj.ktype = &ktype_cdev_dynamic;
                INIT_LIST_HEAD(&p->list);
                kobject_init(&p->kobj);
        }
        return p;
}
```

13.2.6 分配和释放设备号

在调用 cdev_add() 函数向系统注册字符设备之前，应首先调用 register_chrdev_region() 函数向系统申请设备号。

相反地，在调用 cdev_del() 函数从系统注销字符设备之后，应该调用 unregister_chrdev_region() 以释放原先申请的设备号，

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
{
        p->dev = dev;
        p->count = count;
        return kobj_map(cdev_map, dev, count, NULL, exact_match, exact_lock, p);
}

void cdev_del(struct cdev *p)
{
        cdev_unmap(p->dev, p->count);
        kobject_put(&p->kobj);
}
```

其中用到的申请区域的两个函数的原型如下，它们的实现也在 fs/char_dev.c 中：

```
int  register_chrdev_region(dev_t  from,  unsigned  count,  const  char *name);
void unregister_chrdev_region(dev_t from, unsigned count);
```

当 register_chrdev_region 返回值为0，则表示申请成功。
如果 from 参数是0，代表要求系统自动分配一个可用的 major 并返回这个 major 。

```
struct cdev uart_cdev;
dev_t uart_dev_no;

int uart_drv_init(void)
```

```
{
        printk("uart_drv init ok \n");

        //register_chrdev(UART_MAJOR, "myttyS3", &uart_drv_fops);

        // use cdev
        uart_dev_no = MKDEV(UART_MAJOR, 3);
        register_chrdev_region(uart_dev_no, 1, "myttyS3");
        cdev_init(&uart_cdev, &uart_drv_fops);
        cdev_add(&uart_cdev, uart_dev_no, 1);

        return 0;
}

void uart_drv_exit(void)
{
        printk("uart_drv exit ok \n");

        //unregister_chrdev(UART_MAJOR, "myttyS3");

        // use cdev
        unregister_chrdev_region(uart_dev_no, 1);
        cdev_del(&uart_cdev);

        return;
}
```

Examples

13.3 文件操作和file结构

13.3.1 struct file_operations

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
```

```
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
      unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
       ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **);
};
```

这个结构，定义在 linux/fs.h，是一个函数指针的集合． 每个打开文件(内部用一个 file 结构来代表，稍后我们会查看)与它自身的函数集合相关连(通过包含一个称为 f_op 的成员，它指向一个 file_operations 结构)． 这些操作大部分负责实现系统调用，因此，命名为 open，read，等等． 我们可以认为文件是一个“对象”并且其上的函数操作称为它的“方法”，使用面向对象编程的术语来表示一个对象声明的用来操作对象的动作．

13.3.2 基本元素

file_operations的主要域:

```
struct module *owner：指向模块自身。
open：打开设备。
release：关闭设备。
read：从设备上读数据。
write：向设备上写数据。
ioctl：操作设备函数。
mmap：映射设备空间到进程的地址空间。
```

13.3.3 接口含义

```
struct module *owner
```

第一个 file_operations 成员根本不是一个操作；它是一个指向拥有这个结构的模块的指针． 这个成员用来在它的操作还在被使用时阻止模块被卸载． 几乎所有时间中，它被简单初始化为 THIS_MODULE，一个在 linux/module.h 中定义的宏．

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

用来从设备中获取数据． 在这个位置的一个空指针导致 read 系统调用以 -EINVAL(“Invalid argument”) 失败． 一个非负返回值代表了成功读取的字节数(返回值是一个 “signed size” 类型，常常是目标平台本地的整数类型)．

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

发送数据给设备. 如果 NULL，-EINVAL 返回给调用 write 系统调用的程序. 如果非负，返回值代表成功写的字节数.

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

ioctl 系统调用提供了发出设备特定命令的方法(例如格式化软盘的一个磁道，这不是读也不是写). 另外，几个 ioctl 命令被内核识别而不必引用 fops 表. 如果设备不提供 ioctl 方法，对于任何未事先定义的请求(-ENOTTY, “设备无这样的 ioctl”)，系统调用返回一个错误.

```
int (*mmap) (struct file *, struct vm_area_struct *);
```

mmap 用来请求将设备内存映射到进程的地址空间. 如果这个方法是 NULL，mmap 系统调用返回 -ENODEV.

```
int (*open) (struct inode *, struct file *);
```

尽管这常常是对设备文件进行的第一个操作，不要求驱动声明一个对应的方法. 如果这个项是 NULL，设备打开一直成功，但是你的驱动不会得到通知.

```
int (*release) (struct inode *, struct file *);
```

在文件结构被释放时引用这个操作. 如同 open，release 可以为 NULL.

13.3.4 struct inode_operations

```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
        struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
```

```
        int (*readlink) (struct dentry *, char __user *,int);
        void * (*follow_link) (struct dentry *, struct nameidata *);
        void (*put_link) (struct dentry *, struct nameidata *, void *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*check_acl)(struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
        void (*truncate_range)(struct inode *, loff_t, loff_t);
        long (*fallocate)(struct inode *inode, int mode, loff_t offset,
                          loff_t len);
        int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
                      u64 len);
};
```

13.3.5 用户空间和kernel空间的数据互传

模块在很多时候需要和用户程序交互，其中包括数据传输。在模块里做传输的时候使用memcpy往往会出错。

请看例子：

```
在用户空间里定义：
char user_buffer[100];

在内核模块里定义：
char kernel_buffer[100];

使用 memcpy(kernel_buffer,user_buffer,100);
往往不能成功；原因在于用户空间的user_buffer，有可能是缺页状态。这时候kernel会出现异常。

我们需要使用更安全的函数 copy_to_user，copy_from_user。 它们的定义在 arch/arm/include/asm/uaccess.h 文件中。

copy_to_user(user_buffer,kernel_buffer,100);

copy_from_user(kernel_buffer,user_buffer,100);

这里还有一个函数：
put_user(k,u),get_user(k,u);        它们适合于每次访问char,int等单个数据类型；
```

13.4 GPIO/UART 驱动代码实现

13.4.1 led 驱动

```
#include <linux/module.h>        // module_init
#include <asm/io.h>                // ioremap
#include <linux/fs.h>                // file_operations
#include <asm/uaccess.h>        // copy_from_user

MODULE_LICENSE("GPL");

#define MA        240

volatile int * pled;

int led_drv_open(struct inode *inode, struct file *filp)
{
        int major, minor;

        major = MAJOR(inode->i_rdev);
        minor = MINOR(inode->i_rdev);

        printk("led drv open: major %d, minor %d\n", major, minor);

        return 0;
}

ssize_t led_drv_write(struct file *filp, const char __user * buf, size_t count, loff_t *f_pos)
{
        char kbuf[128];

        //buf[count] = '\0';
        printk("led drv write %d\n", count);

//        printk("buf = %s\n", buf);
//        printk("buf at %p\n", buf);
        printk("count = %d\n", count);

//        copy_from_user(kbuf, buf, count);
        *pled = buf[0];

//        printk("kbuf = %s\n", kbuf);
//        printk("kbuf at %p\n", kbuf);

        return count;
}

int led_drv_release(struct inode *inode, struct file *filp)
{
        printk("led drv release ok!\n");

        return 0;
}

struct file_operations led_fops =
{
        .owner = THIS_MODULE,
```

```
        .open = led_drv_open,
        .write = led_drv_write,
        .release = led_drv_release,
};

static int led_drv_init(void)
{
        int rc;

        printk("led init \n");

        pled = ioremap(0xE0200284, 4);

        //*pled = 0;
        rc = register_chrdev(MA, "akae", &led_fops);
        if (rc < 0)
        {
                printk("register failed\n");
                return -1;
        }

        printk("register char ok %d!\n", rc);

        return 0;
}

static void led_drv_exit(void)
{
        printk("led exit \n");

        //*pled = 0xF;
        unregister_chrdev(MA, "akae");
        printk("unregister char ok!\n");

        return;
}

module_init(led_drv_init);
module_exit(led_drv_exit);
```

13.4.2 课堂练习: 串口设备驱动 (char device driver)

```
Makefile for kernel module
基本的内核模块功能, uart_drv_init, uart_drv_exit, #include, License
串口的驱动接口:  uart_init, uart_putchar, uart_getchar (ioremap)
insmod uart_drv.ko
        uart_drv_init -> write "hello" -> for + uart_putchar
        uart_drv_init -> read char -> write char -> echo
```

1 基本驱动功能实现

```
/* uart_drv.c */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/io.h>

MODULE_LICENSE("GPL");

struct uart_sfr
{
        int ulcon;
        int ucon;
        int ufcon;
        int umcon;
        int utrstat;
        int uerstat;
        int ufstat;
        int umstat;
        int utxh;
        int urxh;
        int ubrdiv;
        int udivslot;
};

typedef struct uart_sfr USFR;

static volatile USFR *puart;

void uart_init(void)
{
#if 0
        // see how linux set UART0 regs
        puart = ioremap(0xe2900000, sizeof(USFR));
        printk("reg ulcon = %x\n", puart->ulcon);
        printk("reg ucon = %x\n", puart->ucon);
        printk("reg ubrdiv = %x\n", puart->ubrdiv);
        printk("reg udivslot = %x\n", puart->udivslot);
#endif

        puart = ioremap(0xe2900c00, sizeof(USFR));
        puart->ulcon = 0x3;
        puart->ucon = 0x7c5;
        puart->ubrdiv = 0x23;
        puart->udivslot = 0x808;

        return;
}

int uart_putchar(char c)
{
        while ((puart->utrstat & (1<<2)) == 0)
                ;

        puart->utxh = c;
```

```
        return 0;
}

int uart_drv_init(void)
{
        printk("uart_drv init ok \n");

        uart_init();

        uart_putchar('h');
        uart_putchar('e');
        uart_putchar('l');
        uart_putchar('l');
        uart_putchar('o');
        uart_putchar('\n');

        return 0;
}

void uart_drv_exit(void)
{
        printk("uart_drv exit ok \n");

        return;
}

module_init(uart_drv_init);
module_exit(uart_drv_exit);
```

example code

2 加入字符设备驱动的接口 加入对于串口 UART3 的 open, release, read, write

```
        open - 115200, 8N1 (init)
        read - return 1 (getchar())
        write - uart_putchar() 每次写入1个字节
        release - null
```

加入注册字符设备和注销字符设备

```
struct file_operations fops =
{
        ...
};
init -> register_chrdev(245, "notmyttyS3", &f_ops)
exit -> unregister_chrdev(245, "notmyttyS3")
```

创建设备文件的节点

```
mknod myttyS3 c 245 3
```

测试字符设备驱动

```
test write:                echo "hello" > myttyS3
test read:                cat myttyS3
```

```
/* uart_drv.c */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/io.h>
#include <linux/cdev.h>

MODULE_LICENSE("GPL");

#define UART_MAJOR        245

struct uart_sfr
{
        int ulcon;
        int ucon;
        int ufcon;
        int umcon;
        int utrstat;
        int uerstat;
        int ufstat;
        int umstat;
        int utxh;
        int urxh;
        int ubrdiv;
        int udivslot;
};

typedef struct uart_sfr USFR;

static volatile USFR *puart;

#define ULCON0                (puart->ulcon)
#define UCON0                (puart->ucon)
#define UBRDIV0                (puart->ubrdiv)
#define UDIVSLOT0        (puart->udivslot)
#define UTRSTAT0        (puart->utrstat)
#define UTXH0                (puart->utxh)
#define URXH0                (puart->urxh)

void uart_init(void)
{
        // set UART SFRs
```

```
        ULCON0 = 0x3;
        UCON0 = 0x245;

        // 66Mhz / (115200*16) - 1 = 0x23
        // 66Mhz / (19200*16) - 1 = 0xD5
        UBRDIV0 = 0x23;
        UDIVSLOT0 = 0x808;

        return;
}

char uart_getchar(void)
{
        char c;

        // polling receive status: if buffer is full
        while ((UTRSTAT0 & (1<<0)) == 0)
                ;

        c = URXH0;

        return c;
}

void uart_putchar(char c)
{
        // polling transmit status: if buffer is empty
        while ((UTRSTAT0 & (1<<2)) == 0)
                ;

        UTXH0 = c;

        return;
}

int uart_drv_open(struct inode * inode, struct file * filp)
{
        printk("uart open\n");

        uart_init();

        uart_putchar('o');
        uart_putchar('p');
        uart_putchar('e');
        uart_putchar('n');
        uart_putchar('\n');

        return 0;
}

int uart_drv_release(struct inode * inode, struct file * filp)
{
        printk("uart release\n");

        uart_putchar('c');
        uart_putchar('l');
```

```
        uart_putchar('o');
        uart_putchar('s');
        uart_putchar('e');
        uart_putchar('\n');

        return 0;
}

int uart_drv_write(struct file * filp, const char __user * buf, size_t count, loff_t *f_pos)
{
        char c;

        printk("uart write %d bytes\n", count);

        c = *buf;

        uart_putchar(c);

        return 1;
}

int uart_drv_read(struct file * filp, char __user * buf, size_t count, loff_t *f_pos)
{
        char c;

        printk("uart read %d bytes\n", count);

        c = uart_getchar();

        *buf = c;

        return 1;
}

struct file_operations uart_drv_fops =
{
    .owner = THIS_MODULE,
    .open = uart_drv_open,
    .release = uart_drv_release,
    .write = uart_drv_write,
    .read = uart_drv_read,
};

struct cdev uart_cdev;
dev_t uart_dev_no;

int uart_drv_init(void)
{
        printk("uart_drv init ok \n");

        //register_chrdev(UART_MAJOR, "myttyS3", &uart_drv_fops);

        // use cdev
        uart_dev_no = MKDEV(UART_MAJOR, 3);
        register_chrdev_region(uart_dev_no, 1, "myttyS3");
        cdev_init(&uart_cdev, &uart_drv_fops);
```

```
        cdev_add(&uart_cdev, uart_dev_no, 1);

        return 0;
}

void uart_drv_exit(void)
{
        printk("uart_drv exit ok \n");

        //unregister_chrdev(UART_MAJOR, "myttyS3");

        // use cdev
        unregister_chrdev_region(uart_dev_no, 1);
        cdev_del(&uart_cdev);

        return;
}

module_init(uart_drv_init);
module_exit(uart_drv_exit);
```

example code

```
引入系统调用, open, read, write, close
在 read 和 write 的基础上, 实现简单的 shell 功能
```

3 加入应用程序

```
/* test.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <stdio.h>
#include <unistd.h>

int fd;

char mygetchar(void)
{
        char c;

        read(fd, &c, 1);

        // echo
        write(fd, &c, 1);

        return c;
```

```
}

void myputchar(char c)
{
        write(fd, &c, 1);

        return;
}

void myputs(char * s)
{
        while (*s)
                myputchar(*s++);

        return;
}

void mygets(char * s)
{
        char c;

        while ((c = mygetchar()) != '\r')
                *s++ = c;

        *s = '\0';
        myputs("\r\n");

        return;
}

int main(void)
{
        fd = open("myttyS3", O_RDWR);

        while (1)
        {
                char buf[512];

                myputs("akaedu $ ");
                mygets(buf);
                myputs(buf);
                myputs("\r\n");

        #if 0
                char c;
                c = mygetchar();
                myputchar(c);
        #endif
        }

        close(fd);

        return 0;
}
```

example code

13.4.3 uart 驱动

```
#include <linux/module.h>
#include <linux/fs.h>        // register_chrdev
#include <asm/io.h>
//#include <asm/uaccess.h>

#define MA        243

MODULE_LICENSE("GPL");

struct uart_sfr
{
        int ulcon;
        int ucon;
        int ufcon;
        int umcon;
        int utrstat;
        int uerstat;
        int ufstat;
        int umstat;
        int utxh;
        int urxh;
        int ubrdiv;
        int udivslot,
};

typedef struct uart_sfr USFR;

static volatile USFR *puart;

//#define printk                noprintk

int noprintk(char * fmt, ...)
{
        return 0;
}

#define PRINT(x)        printk(#x " = 0x%x\n", x);

int uart_open(struct inode * inode, struct file * filp)
{
        int major = MAJOR(inode->i_rdev);
        int minor = MINOR(inode->i_rdev);
        int * p;
        int i;

        printk("uart open: major %d, minor %d\n", major, minor);

        puart = ioremap(0xe2900000, sizeof(USFR));
        p = (int *)puart;

        PRINT((int)p);
```

```
        for (i = 0; i < sizeof(USFR)/4; i++)
        {
                PRINT(*p++);
        }

        puart->ufcon = 0;

        return 0;
}

int uart_release(struct inode * inode, struct file * filp)
{
        printk("uart release\n");

        return 0;
}

int uart_putchar(char c)
{
        while ((puart->utrstat & (1<<2)) == 0)
                ;

        puart->utxh = c;

        return 0;
}

int myputchar(char c)
{
        if (c == '\n')
                uart_putchar('\r');

        uart_putchar(c);

        return 0;
}

int uart_write(struct file * filp, const char __user * buf, size_t count, loff_t *f_pos)
{
        int i;

        printk("uart write\n");

        printk("buf = %c\n", buf[0]);
        printk("count = %d\n", count);


        for (i = 0; i < count; i++)
        {
                myputchar(buf[i]);
        }


        return count;
}
```

```
int uart_drv_ioctl(struct inode * inode, struct file * filp, unsigned int cmd, unsigned long arg)
{
        #define PRINTD(x)        printk(#x " = %d, <%s>\n", x, __FUNCTION__);

        PRINTD(cmd);
        PRINTD((int)arg);

#define UART_SET_BUAD_RATE  0
#define UART_ENABLE_FIFO    1

        switch (cmd)
        {
        case UART_SET_BUAD_RATE:
                uart_set_baudrate(arg);
                break;
        }

        return 0;
}


struct file_operations uart_fops =
{
        .owner = THIS_MODULE,
        .open = uart_open,
        .release = uart_release,
        .write = uart_write,
        .ioctl = uart_ioctl,
};

int uart_init(void)
{
        int rc;

        rc = register_chrdev(MA, "myuart", &uart_fops);

        if (rc < 0)
        {
                printk("register chrdev failed! %d\n", rc);
                return -1;
        }

        printk("uart init ok \n");

        return 0;
}

void uart_exit(void)
{
        printk("uart exit ok \n");

        unregister_chrdev(MA, "myuart");

        return;
```

```
}


module_init(uart_init);
module_exit(uart_exit);
```

第 14 章

中断的概念

14.1 中断的概念

所谓中断，是一个过程。一般CPU在执行程序的时候，遇到外部紧急事件需要处理。CPU会立即暂停正在执行的程序，马上处理紧急事件后，再接着执行被暂停的程序。这个概念叫做中断。

中断可分为同步（synchronous）中断和异步（asynchronous）中断： 1. 同步中断是当指令执行时由 CPU 控制单元产生，之所以称为同步，是因为只有在一条指令执行完毕后 CPU 才会发出中断，而不是发生在代码指令执行期间，比如系统调用。 2. 异步中断是指由其他硬件设备依照 CPU 时钟信号随机产生，即意味着中断能够在指令之间发生，例如键盘中断，串口中断，网卡中断。

根据中断的来源，中断可分为内部中断和外部中断，内部中断的中断源来自CPU内部（软件中断指令、溢出、除法错误等，例如，操作系统从用户态切换到内核态需借助CPU内部的软件中断），外部中断的中断源来自CPU外部，由外设提出请求。根据是否可以屏蔽中断分为可屏蔽中断与不屏蔽中断（NMI），可屏蔽中断可以通过屏蔽字被屏蔽，屏蔽后，该中断不再得到响应，而不屏蔽中断不能被屏蔽。

根据中断入口跳转方法的不同，中断分为向量中断和非向量中断。采用向量中断的 CPU 通常为不同的中断分配不同的中断号，当检测到某中断号的中断到来后，就自动跳转到与该中断号对应的地址执行。

不同中断号的中断有不同的入口地址。非向量中断的多个中断共享一个入口地址，进入该入口地址后再通过软件判断中断标志来识别具体是哪个中断。也就是说，向量中断由硬件提供中断服务程序入口地址，非向量中断由软件提供中断服务程序入口地址。

一般我们将第一类归为异常;第二类归为外部中断，也就是我们常说的中断（不包括异常）。今天我们要描述的就是外部中断。

14.1.1 中断与轮询/DMA的关系

轮询 轮询是早期计算机对I/O设备访问的一种管理方式;定时对各种设备轮流询问，看是否有需要处理的要求。如果设备需要处理，我们就加以处理，否则返回继续执行。

中断 当设备希望CPU处理此设备的数据时，设备就会主动向CPU发送处理请求。CPU会暂停当前任务来执行这个请求。这种处理方式叫做中断。

DMA　DMA（Direct Memory Access)直接内存访问，原本是指：设备不通过CPU将数据传输到内存的意思。目前很多系统都有DMA控制器，支持DMA访问。

14.1.2　中断轮询优缺点

轮询的缺点：　在我们使用的桌面系统中经常需要获取键盘，鼠标的输入信息。这些信息的输入是不定时的。 如果采取轮询方式去读取数据，就给设计者带来麻烦。有些应用需要不停的敲打键盘，例如打字员。有些应用很少敲打键盘，例如：媒体播放。如何给定一个合理的轮询时间间隔是非常重要的。

再如我们的网络应用，有时候我们几个小时不需要接收网络数据，有些应用需要大量网络数据，例如：网络服务器。

当这些设备都采取轮询方式去工作的时候，几乎很难设计一个完善的操作系统。这就是轮询的最大缺点。而中断恰好解决了这个问题。

中断的缺点：　但是中断方式处理起来非常复杂;当中断产生后，系统需要保存当前执行的任务信息，并且中断执行完成后,需要返回到正确的执行任务。这是一个复杂的事情。而像Bootloader这样的系统里去实现这样的程序是没有必要的。因此大部分的Bootloader里是没有中断处理的。 当一个设备的中断非常频繁的时候，会给系统带来非常大的开销，这个时候如果采用轮询的方式，避免大量的中断进程与系统进程的切换，可以提高系统的执行效率。

14.1.3　DMA与轮询

轮询的方式读取数据会消耗很多CPU的时间，而DMA的方式就会减少很多CPU的时间，从而大大提高的了系统的效率。

看一个例子分别说明：　当我们从硬盘上读取一个扇区（512字节）内容。

★ 轮询方式：

设置硬盘寄存器，告诉硬盘我们需要读取第315扇区内容；从硬盘的寄存器读取512次（如果每次一个字节）。

★ DMA方式：

设置硬盘寄存器，告诉硬盘我们需要读取第315扇区内容；DMA控制器将512个字节的内容存放到内存后通知CPU。

因此DMA方式省去了CPU去访问512次硬盘的时间。

14.1.4　DMA与中断

上面我们两次提到了轮询，我们是否能够发现这两次轮询的作用并不一样。 也就是中断取代了轮询的通知方式，DMA取代了轮询的读写数据方式。

那么DMA和中断是什么关系呢？其实在现代的操作系统中，DMA和中断是并存的。 他们的着重点不相同。他们是互相配合的关系。

14.2　中断相关数据结构

14.2.1　Linux中断数据结构

上面我们已经讲述了 SoC 中断控制器可以支持多个设备。当一个设备产生中断后，CPU进入中断状态，Linux处理流程如下：

```
保护现场；
通过中断控制寄存器获取产生中断的设备；
找到设备的处理程序；
调用设备处理程序；
返回中断
```

那么我们就需要有一个表来存放设备的中断处理程序。这个表在Linux里叫做irq_desc[]。

每个IRQ中断线，Linux都用一个irq_desc_t数据结构来描述，我们把它叫做 IRQ描述符， NR_IRQS 个IRQ形成一个全局数组 irq_desc[]，其定义在 include/linux/irq.h 中：

数据结构irq_desc里有一项叫做 irqaction。它里面就包含着 handler() 函数。

irq_desc_t 的 action 成员实际是一个 irqaction 结构的链表，因为多个设备可能共享一个中断源，每个设备都必须提供一个 irqction 结构挂入 action 链表。

内核中的定义：

```
/**
 * struct irq_desc - interrupt descriptor
 *
 * @handle_irq:                highlevel irq-events handler [if NULL, __do_IRQ()]
 * @chip:               low level interrupt hardware access
 * @msi_desc:               MSI descriptor
 * @handler_data:       per-IRQ data for the irq_chip methods
 * @chip_data:               platform-specific per-chip private data for the chip
 *                       methods, to allow shared chip implementations
 * @action:              the irq action chain
 * @status:              status information
 * @depth:               disable-depth, for nested irq_disable() calls
 * @wake_depth:               enable depth, for multiple set_irq_wake() callers
 * @irq_count:               stats field to detect stalled irqs
 * @irqs_unhandled:       stats field for spurious unhandled interrupts
 * @last_unhandled:       aging timer for unhandled count
 * @lock:               locking for SMP
 * @affinity:               IRQ affinity on SMP
 * @cpu:               cpu index useful for balancing
 * @pending_mask:       pending rebalanced interrupts
 * @dir:               /proc/irq/ procfs entry
 * @affinity_entry:       /proc/irq/smp_affinity procfs entry on SMP
 * @name:               flow handler name for /proc/interrupts output
 */
struct irq_desc {
        irq_flow_handler_t       handle_irq;
        struct irq_chip              *chip;
        struct msi_desc              *msi_desc;
        void                        *handler_data;
        void                        *chip_data;
        struct irqaction       *action;       /* IRQ action list */
        unsigned int               status;              /* IRQ status */

        unsigned int               depth;               /* nested irq disables */
        unsigned int               wake_depth;       /* nested wake enables */
        unsigned int               irq_count;       /* For detecting broken IRQs */
        unsigned int               irqs_unhandled;
```

```
        unsigned long               last_unhandled;       /* Aging timer for unhandled count */
        spinlock_t                lock;
#ifdef CONFIG_SMP
        cpumask_t               affinity;
        unsigned int              cpu;
#endif
#if defined(CONFIG_GENERIC_PENDING_IRQ) || defined(CONFIG_IRQBALANCE)
        cpumask_t               pending_mask;
#endif
#ifdef CONFIG_PROC_FS
        struct proc_dir_entry       *dir;
#endif
        const char               *name;
} ____cacheline_internodealigned_in_smp;

extern struct irq_desc irq_desc[NR_IRQS];

/*
 * Migration helpers for obsolete names, they will go away:
 */
#define hw_interrupt_type       irq_chip
typedef struct irq_chip               hw_irq_controller;
#define no_irq_type               no_irq_chip
typedef struct irq_desc               irq_desc_t;
```

hw_interrupt_type 中断控制器相关的操作。

```
此结构体用来描述中断控制器，它是一个抽象的中断控制器，其成员是一系列指向函数的指针。
typename：给相应的控制器起一个便于理解的名字。
startup：允许从给定的控制器的IRQ所产生的事件。（基本上与enable相同）
shutdown：禁止从给定的控制器的IRQ所产生的事件。（基本上与disable相同）
```

struct irqaction 结构记录当中断发生时具体的处理函数，定义在 include/linux/interrupt.h 文件中。

```
struct irqaction {
        irq_handler_t           handler;              /* 具体的处理函数 */
        void                    *dev_id;
        void __percpu           *percpu_dev_id;
        struct irqaction        *next;                    /* 利用该成员形成链表 */
        irq_handler_t           thread_fn;
        struct task_struct      *thread;
        unsigned int            irq;
        unsigned int            flags;
        unsigned long           thread_flags;
        unsigned long           thread_mask;
        const char              *name;
        struct proc_dir_entry   *dir;
};
```

其中 irqreturn_t 是一个函数指针类型，通过 typedef 来进行了新类型声明。 typedef irqreturn_t (*irq_handler_t)(int, void *);

irqreturn_t 是一个枚举类型，定义在 include/linux/irqreturn.h 中。正常中断处理返回 IRQ_HANDLED 。

```
/**
* enum irqreturn
* @IRQ_NONE            interrupt was not from this device
* @IRQ_HANDLED         interrupt was handled by this device
* @IRQ_WAKE_THREAD     handler requests to wake the handler thread
*/
enum irqreturn {
        IRQ_NONE                = (0 << 0),
        IRQ_HANDLED             = (1 << 0),
        IRQ_WAKE_THREAD         = (1 << 1),
};

typedef enum irqreturn irqreturn_t;
```

14.2.2 中断号 irq 的确定

```
$ cat /proc/interrupt | grep uart
```

查看 /proc/interrupt 文件

```
$ grep -rns "IRQ" arch/arm/* | grep "UART" | grep "16"

$ vi arch/arm/plat-s5p/include/plat/irqs.h

我们可以获取如下信息：

/* UART interrupts, each UART has 4 intterupts per channel so
 * use the space between the ISA and S3C main interrupts. Note, these
 * are not in the same order as the S3C24XX series! */

#define IRQ_S5P_UART_BASE0        (16)
#define IRQ_S5P_UART_BASE1        (20)
#define IRQ_S5P_UART_BASE2        (24)
#define IRQ_S5P_UART_BASE3        (28)

#define UART_IRQ_RXD                (0)
#define UART_IRQ_ERR                (1)
#define UART_IRQ_TXD                (2)

#define IRQ_S5P_UART_RX0        (IRQ_S5P_UART_BASE0 + UART_IRQ_RXD)
#define IRQ_S5P_UART_TX0        (IRQ_S5P_UART_BASE0 + UART_IRQ_TXD)
#define IRQ_S5P_UART_ERR0        (IRQ_S5P_UART_BASE0 + UART_IRQ_ERR)
```

```
#define IRQ_S5P_UART_RX1        (IRQ_S5P_UART_BASE1 + UART_IRQ_RXD)
#define IRQ_S5P_UART_TX1        (IRQ_S5P_UART_BASE1 + UART_IRQ_TXD)
#define IRQ_S5P_UART_ERR1        (IRQ_S5P_UART_BASE1 + UART_IRQ_ERR)

#define IRQ_S5P_UART_RX2        (IRQ_S5P_UART_BASE2 + UART_IRQ_RXD)

...


#define IRQ_EINT(x)               ((x) < 16 ? ((x) + S5P_EINT_BASE1) \
                                       : ((x) - 16 + S5P_EINT_BASE2))

#define EINT_OFFSET(irq)        ((irq) < S5P_EINT_BASE2 ? \
                                              ((irq) - S5P_EINT_BASE1) : \
                                              ((irq) + 16 - S5P_EINT_BASE2))

#define IRQ_EINT_BIT(x)               EINT_OFFSET(x)
```

查看内核源码定义

14.3　中断处理程序

14.3.1　Linux 注册函数

设备驱动要将回调函数 handler 注册到 irq_desc 的表项中。这就是我们要讲的 request_irq 。

request_irq()/free_irq() 的声明在 linux/interrupt.h 中，实现在 kernel/irq/manager.c 中，如下：

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
        return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}

extern void free_irq(unsigned int, void *);
```

request_irq 函数的参数如下：

```
unsigned int irq                // 请求的中断号
irqreturn_t (*handler)                 // 安装的处理函数指针. 我们在本章后面讨论给这个函数的参数以及
它的返回值.
unsigned long flags                // 与中断管理相关的选项的位掩码(后面描述).
const char *dev_name                //这个传递给 request_irq 的字串用在 /proc/interrupts 来显示中断
的拥有者(下一节看到)
void *dev_id                        // 用作共享中断线的指针. 它是一个独特的标识, 用在当释放中断线时
以及可能还被驱动用来指向它自己的私有数据区(来标识哪个设备在中断). 如果中断没有被共享, dev_id 可以设
```

```
置为 NULL，但是使用这个项指向设备结构不管如何是个好主意．我们将在"实现一个处理"一节中看到 dev_id 的一个实际应用．
```

flags 中可以设置的位如下：

```
SA_INTERRUPT
        当置位了，这表示一个"快速"中断处理．快速处理在当前处理器上禁止中断来执行(这个主题在"快速和慢速处理"一节涉及)．
SA_SHIRQ
        这个位表示中断可以在设备间共享．共享的概念在"中断共享"一节中略述．
SA_SAMPLE_RANDOM
        这个位表示产生的中断能够有贡献给 /dev/random 和 /dev/urandom 使用的加密池．这些设备在读取时返回真正的随机数并且设计来帮助应用程序软件为加密选择安全钥．这样的随机数从一个由各种随机事件贡献的加密池中提取的．如果你的设备以真正随机的时间产生中断，你应当设置这个标志．如果，另一方面，你的中断是可预测的( 例如，一个帧抓取器的场消隐)，这个标志不值得设置 -- 它无论如何不会对系统加密有贡献．可能被攻击者影响的设备不应当设置这个标志；例如，网络驱动易遭受从外部计时的可预测报文并且不应当对加密池有贡献．更多信息看 drivers/char/random.c 的注释．

设备通过函数request_irq（）注册一个IRQ号，并提供相应的处理函数。
```

从 request_irq 返回给请求函数的返回值或者是 0 指示成功，或者是一个负的错误码，如同平常． 函数返回 -EBUSY 来指示另一个驱动已经使用请求的中断线是不寻常的．

request_threaded_irq() 的实现在 kernel/irq/manage.c 文件中，如下：

```
__setup_irq(irq, desc, action);
```

下面是键盘驱动程中注册中断的代码：

```
request_irq(IRQ_EINT0, key1_irq_isr, SA_INTERRUPT, "key2345irq", NULL);
```

IRQ_EINT0是IRQ号，key1_irq_isr （）是处理函数。如果该函数成功的话，irq_desc[IRQ_EINT0]的action成员将会指向一个新分配的irqaction结构，该结构的handler指向key1_irq_isr （）。

14.3.2 /proc 接口

无论何时一个硬件中断到达处理器，一个内部的计数器递增，提供了一个方法来检查设备是否如希望地工作． 报告的中断显示在 /proc/interrupts． 下面的快照取自一个双处理器 Pentium 系统：

```
root@montalcino:/bike/corbet/write/ldd3/src/short# cat /proc/interrupts
        CPU0     CPU1
 0:  4848108       34   IO-APIC-edge  timer
 2:        0        0         XT-PIC  cascade
```

```
 8:        3        1   IO-APIC-edge  rtc
 10:    4335        1  IO-APIC-level  aic7xxx
 11:    8903        0  IO-APIC-level  uhci_hcd
 12:      49        1   IO-APIC-edge  i8042
NMI:       0        0
LOC: 4848187  4848186
ERR:       0
MIS:       0
```

第一项是中断的号，第二项是中断的次数，第三项是中断的触发方式，第四项是中断名称。

14.3.3 中断的处理过程

* 中断信号由外部设备发送到中断控制器，中断控制器根据IRQ号转换成相应的中断向量号传给CPU 。
* CPU接收中断后，保存现场，根据中断向量号到IDT中查找相应的处理函数。对于IRQ n的中断，它的处理函数IRQn_interrutp()。
* 调用do_IRQ()函数。该函数完成对中断控制器确认、设置中断源状态等动作， 接着它会根据IRQ号找到描述中断具体动作的irqaction结构变量action，执行如下代码：

处理 irq 事件的函数，主要实现在 kernel/irq/handle.c 中，其中 res = action->handler(irq, action->dev_id);
执行了调用用户注册的 handler() 函数的功能。

```
182 irqreturn_t handle_irq_event(struct irq_desc *desc)
183 {
184         struct irqaction *action = desc->action;
185         irqreturn_t ret;
186
187         desc->istate &= ~IRQS_PENDING;
188         irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
189         raw_spin_unlock(&desc->lock);
190
191         ret = handle_irq_event_percpu(desc, action);
192
193         raw_spin_lock(&desc->lock);
194         irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
195         return ret;
196 }

132 irqreturn_t
133 handle_irq_event_percpu(struct irq_desc *desc, struct irqaction *action)
134 {
135         irqreturn_t retval = IRQ_NONE;
136         unsigned int flags = 0, irq = desc->irq_data.irq;
137
138         do {
139                 irqreturn_t res;
140
141                 trace_irq_handler_entry(irq, action);
```

```
142                 res = action->handler(irq, action->dev_id);
143                 trace_irq_handler_exit(irq, action, res);
144
145                 if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pF enabled interrupts\n",
146                               irq, action->handler))
147                         local_irq_disable();
148
149                 switch (res) {
150                 case IRQ_WAKE_THREAD:
151                         /*
152                          * Catch drivers which return WAKE_THREAD but
153                          * did not set up a thread function
154                          */
155                         if (unlikely(!action->thread_fn)) {
156                                 warn_no_thread(irq, action);
157                                 break;
158                         }
159
160                         irq_wake_thread(desc, action);
161
162                         /* Fall through to add to randomness */
163                 case IRQ_HANDLED:
164                         flags |= action->flags;
165                         break;
166
167                 default:
168                         break;
169                 }
170
171                 retval |= res;
172                 action = action->next;
173         } while (action);
174
175         add_interrupt_randomness(irq, flags);
176
177         if (!noirqdebug)
178                 note_interrupt(irq, desc, retval);
179         return retval;
180 }
```

* 最后do_IRQ()函数要检查是否有软中断，如有则调用do_softirq()执行软中断。
* 跳转到 ret_from_intr退出，恢复中断前的现场。

14.3.4 Examples

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/interrupt.h>
#include <linux/irqreturn.h>
//#include <linux/irq.h>

MODULE_LICENSE("GPL");
```

```
#define DPRINT(x)        printk("<%s> " #x " = 0x%x\n", __func__, (int)x);

irqreturn_t myhandler(int irq, void * p)
{
//        printk("irqmyhandler called!\n");
        DPRINT(irq);

        return IRQ_HANDLED;
}

static __init int interrupt_init(void)
{
        int ri;
        int btn_irq;

        DPRINT(interrupt_init);

        btn_irq = 160;

//        ri = request_irq(17, myhandler, 0, "my test interrupt", NULL);
//        ri = request_irq(17, 1, 0, "my test interrupt", NULL);
        ri = request_irq(btn_irq, myhandler, IRQF_TRIGGER_FALLING, "btn K1 interrupt", NULL);
        ri = request_irq(btn_irq+1, myhandler, IRQF_TRIGGER_FALLING, "btn K2 interrupt", NULL);

        DPRINT(ri);

        return 0;
}

static __exit void interrupt_exit(void)
{
        DPRINT(interrupt_exit);

        free_irq(160, NULL);
        free_irq(161, NULL);

        return;
}

module_init(interrupt_init);
module_exit(interrupt_exit);
```

实际使用过程中，中断的信号常常是用来作为唤醒等待队列里面睡眠的进程的，这就需要了解有关等待队列的知识。

14.4　中断的下半部处理

14.4.1　Tasklet（小任务）

Tasklet 机制是中断处理下半部分最常用的一种方法，其使用也是非常简单的。

一个使用tasklet的中断程序首先会通过执行中断处理程序来快速完成上半部分的工作，接着通过调度 tasklet 使得下半部分的工作得以完成。但是下半部分何时执行则属于

内核的工作。

Tasklet 定义在 linux/include/interrupt.h 实现在 kernel/softirq.c

```
struct tasklet_struct
{
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
};
```

使用举例

```
#include <linux/module.h>        // module_init
#include <asm/io.h>                // ioremap
#include <linux/fs.h>                // file_operations
#include <asm/uaccess.h>        // copy_from_user
#include <linux/sched.h>        // wait_queue

#include <mach/gpio.h>                // S5PV210_GPH2
#include <mach/regs-gpio.h>
#include <linux/interrupt.h>        // requst_irq
#include <linux/irq.h>                // IRQ_TYPE_EDGE_BOTH
#include <asm/thread_info.h>                // IRQ_TYPE_EDGE_BOTH


MODULE_LICENSE("GPL");

struct tasklet_struct mytasklet;

void do_sth(unsigned long data);

irqreturn_t btn_irq_handler(int irq, void * dev_id)
{
        int local = 100;

        printk("<irq> irq = %d \n", irq);

        printk("<irq> &dev = %p \n", &dev_id);
        printk("<irq> &irq = %p \n", &irq);
        printk("<irq> &local = %p \n", &local);

        printk("current = %p \n", current);
        printk("current = %s \n", current->comm);
        printk("current pid = %d \n", current->pid);

        tasklet_schedule(&mytasklet);
        tasklet_schedule(&mytasklet);
//        do_sth(100);

        printk("<irq> = %d \n", irq);
```

```
        return IRQ_HANDLED;
}

void do_sth(unsigned long data)
{
        volatile int i;
        int local = 100;

        printk("\tbegin do_sth...\n");

        printk("\t<do> &data = %p \n", &data);
        printk("\t<do> &i = %p \n", &i );
        printk("\t<do_sth> &local = %p \n", &local);

        printk("\t<do> current = %p \n", current);
        printk("\t<do> current = %s \n", current->comm);
        printk("\t<do> current pid = %d \n", current->pid);

        for (i = 0; i < 1000000000; i++)
                ;

        printk("\tend do_sth...\n");

        return;
}

static int btn_drv_init(void)
{
        int err;
        unsigned long data = 100;

        int * p = ((int)&data) & 0xFFFFE000;

        struct thread_info * pt = (struct thread_info *)p;

        printk("btn init \n");

        printk("\t<do> &data = %p \n", &data);
        printk("p = %p\n", p);

        printk("pt -> current = %p\n", pt->task);

        printk("\t<do> current = %p \n", current);


        tasklet_init(&mytasklet, do_sth, data);

        // see include/linux/irq.h for more types
        //err = request_irq(irq, btn_irq_handler, IRQ_TYPE_EDGE_BOTH,
        err = request_irq(160, btn_irq_handler, IRQ_TYPE_EDGE_FALLING,
                        "btn0", NULL);
        if (err)
        {
                printk("request irq 0 failed!\n");
        }
```

```
        err = request_irq(161, btn_irq_handler, IRQ_TYPE_EDGE_FALLING,
                        "btn1", NULL);
        if (err)
        {
                printk("request irq 1 failed!\n");
        }


        printk("request irq ok! err = %d\n", err);

        return 0;
}

static void btn_drv_exit(void)
{
        printk("btn exit \n");

        free_irq(160, NULL);
        free_irq(161, NULL);

        return;
}

module_init(btn_drv_init);
module_exit(btn_drv_exit);
```

14.4.2 工作队列

工作队列（work queue） 是另外一种将工作推后执行的形式，它和前面讨论的tasklet有所不同。工作队列可以把工作推后，交由一个内核线程去执行，也就是说，这个下半部分可以 在进程上下文中执行。这样，通过工作队列执行的代码能占尽进程上下文的所有优势。最重要的就是工作队列允许被重新调度甚至是睡眠。 那么，什么情况下使用工作队列，什么情况下使用tasklet。如果推后执行的任务需要睡眠，那么就选择工作队列。如果推后执行的任务不需要睡眠，那么就选择tasklet。另外，如果需要用一个可以重新调度的实体来执行你的下半部处理，也应该使用工作队列。它是唯一能在进程上下文运行的下半部实现的机制，也只有它才可以睡眠。这意味着在需要获得大量的内存时、在需要获取信号量时，在需要执行阻塞式的I/O操作时，它都会非常有用。如果不需要用一个内核线程来推后执行工作，那么就考虑使用tasklet。 工作、工作队列和工作者线程 如前所述，我们把推后执行的任务叫做工作（work），描述它的数据结构为work_struct，这些工作以队列结构组织成工作队列（workqueue），其数据结构为workqueue_struct，而工作线程就是负责执行工作队列中的工作。系统默认的工作者线程为events,自己也可以创建自己的工作者线程。

* 表示工作的数据结构

work_struct 定义在 linux/workqueue.h 中，实现在 kernel/workqueue.c 中。

work_struct 结构表示如下：

```
struct work_struct{
```

```
        unsigned long pending; /* 这个工作正在等待处理吗？*/
        struct list_head entry; /* 连接所有工作的链表 */
        void (*func) (void *); /* 要执行的函数 */
        void *data; /* 传递给函数的参数 */
        void *wq_data; /* 内部使用 */
        struct timer_list timer; /* 延迟的工作队列所用到的定时器 */
};
```

这些结构被连接成链表。当一个工作者线程被唤醒时，它会执行它的链表上的所有工作。工作被执行完毕，它就将相应的work_struct对象从链表上移去。当链表上不再有对象的时候，它就会继续休眠。

* 创建推后的工作

要使用工作队列，首先要做的是创建一些需要推后完成的工作。可以通过DECLARE_WORK在编译时静态地建该结构：

```
DECLARE_WORK(name, void (*func) (void *), void *data);
```

这样就会静态地创建一个名为name，待执行函数为func，参数为data的work_struct结构。同样，也可以在运行时通过指针创建一个工作：

```
INIT_WORK(struct work_struct *work, woid(*func) (void *), void *data);
```

这会动态地初始化一个由work指向的工作。

* 工作队列中待执行的函数

工作队列待执行的函数原型是：

```
void work_handler(void *data)
```

这个函数会由一个工作者线程执行，因此，函数会运行在进程上下文中。默认情况下，允许响应中断，并且不持有任何锁。如果需要，函数可以睡眠。需要注意的是， 尽管该函数运行在进程上下文中，但它不能访问用户空间，因为内核线程在用户空间没有相关的内存映射。通常在系统调用发生时，内核会代表用户空间的进程运 行，此时它才能访问用户空间，也只有在此时它才会映射用户空间的内存。

* 对工作进行调度

现在工作已经被创建，我们可以调度它了。想要把给定工作的待处理函数提交给缺省的events工作线程，只需调用

```
schedule_work(&work);
```

work马上就会被调度，一旦其所在的处理器上的工作者线程被唤醒，它就会被执行。有时候并不希望工作马上就被执行，而是希望它经过一段延迟以后再执行。在这种情况下，可以调度它在指定的时间执行：

```
schedule_delayed_work(&work, delay);
```

这时，&work指向的work_struct直到delay指定的时钟节拍用完以后才会执行。

以上实现在 kernel/workqueue.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/workqueue.h>

static struct workqueue_struct *queue = NULL;
static struct work_struct work;

static void work_handler(struct work_struct *data)
{
        printk(KERN_ALERT "work handler function.\n");
}

static int __init test_init(void)
{
        queue = create_singlethread_workqueue("helloworld"); /*创建一个单线程的工作队列*/
        if (!queue)
                goto err;

        INIT_WORK(&work, work_handler);
        schedule_work(&work);

        return 0;
err:
        return -1;
}

static void __exit test_exit(void)
{
        destroy_workqueue(queue);
}
MODULE_LICENSE("GPL");
module_init(test_init);
module_exit(test_exit);
```

工作队列的简单应用

14.4.3 内核定时器

timer_list 定义在 linux/timer.h 中，实现在 kernel/timer.c 中，如下：

```
12 struct timer_list {
13         /*
14          * All fields that change during normal runtime grouped to the
15          * same cacheline
16          */
17         struct list_head entry;
18         unsigned long expires;
19         struct tvec_base *base;
20
21         void (*function)(unsigned long);
22         unsigned long data;
23
24         int slack;
25
26 #ifdef CONFIG_TIMER_STATS
27         int start_pid;
28         void *start_site;
29         char start_comm[16];
30 #endif
31 #ifdef CONFIG_LOCKDEP
32         struct lockdep_map lockdep_map;
33 #endif
34 };
```

定义在 linux/include/timer.h 实现在 kernel/timer.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <asm/current.h>
#include <linux/sched.h>

MODULE_LICENSE("GPL");

extern struct task_struct * current;

struct timer_list mytimer;

void timer_handler(unsigned long arg)
{
        printk("timer arg = %d\n", arg);

        // 启动下一次 timer
        mod_timer(&mytimer, jiffies + HZ * 5);

        return;
}

static int __init akae_init(void)
{
        int local = 0;
        printk("module name is %s\n", KBUILD_MODNAME);
        printk("akae_init at %p\n", akae_init);
```

```
        printk("local at %p\n", &local);
        printk("jiffies at %p\n", &jiffies);
        printk("jiffies is %ld\n", jiffies);
        printk("The process is \"%s\" (pid %i)\n", current->comm, current->pid);

        init_timer(&mytimer);

        printk("HZ = %d\n", HZ);
        mytimer.expires = jiffies + HZ * 5;
        mytimer.function = timer_handler;
        mytimer.data = 100;

        add_timer(&mytimer);

        printk("timer ok!\n");

        return 0;
}

static void __exit akae_exit(void)
{
        printk("module liming exit\n");
        printk("akae_exit at %p\n", akae_exit);

        del_timer_sync(&mytimer);

        return;
}

module_init(akae_init);
module_exit(akae_exit);
```

14.5 等待队列

14.5.1 声明和调用接口

在 Linux 中，一个等待队列由一个“等待队列头”来管理，一个 wait_queue_head_t 类型的结构，定义在 linux/wait.h 中，实现在 kernel/wait.c 。

一个等待队列头可被定义和初始化，使用静态初始化方法：

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

或者动态地，如下：

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

当一个进程睡眠，它这样做以期望某些条件在以后会成真． 如我们之前注意到的，任

何睡眠的进程必须在它再次 醒来时检查来确保它在等待的条件真正为真. Linux 内核中睡眠的最简单方式是一个宏定义，称为 wait_event(有几个变体)；它结合了处理睡眠的细节和进程在等待的条件的检查.

wait_event 的形式是:

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
```

基本的唤醒睡眠进程的函数称为 wake_up. 它有几个形式(但是我们现在只看其中 2 个):

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

wake_up 唤醒所有的在给定队列上等待的进程(尽管这个情形比那个要复杂一些，如同我们之后将见到的). 其他 的形式(wake_up_interruptible)限制它自己到处理一个可中断的睡眠. 通常，这 2 个是不用区分的(如果你使用可 中断的睡眠)；

通常如果你使用 wait_event ，则用 wake_up 唤醒。

如果你使用 wait_event_interruptible, 则用 wake_up_interruptible 唤醒。

14.5.2 范例代码

实现一个有简单行为的设备:任何试图从这个设备读取的进程都被置为睡眠. 无论何时一个进程写这个设备，所有的睡眠进程被唤醒. 这个行为由下面的 read 和 write 方法实现:

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;

ssize_t sleepy_read (struct file *filp, char __user *buf, size_t count, loff_t *pos)
{
        printk(KERN_DEBUG "process %i (%s) going to sleep\n",
               current->pid, current->comm);

        wait_event_interruptible(wq, flag != 0);
        flag = 0;
        printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);

        return 0; /* EOF */
}

ssize_t sleepy_write (struct file *filp, const char __user *buf, size_t count, loff_t *pos)
{
        printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
               current->pid, current->comm);

        flag = 1;
```

```
        wake_up_interruptible(&wq);

        return count; /* succeed, to avoid retrial */
}
```

注意这个例子里 flag 变量的使用. 因为 wait_event_interruptible 检查一个必须变为真的条件，我们使用 flag 来创建那个条件.

14.6 GPIO/UART 中断代码实现

14.6.1 相关文件

14.6.2 中断框架代码

```
//#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/fs.h>
//#include <asm/uaccess.h>

int flag = 0;
#define MA        134

wait_queue_head_t myqueue;

ssize_t myread(struct file *fl, char * buf, size_t size, loff_t * loff)
{
        printk("myread() begin!\n");
        printk("myread() &flag = %p\n", &flag);

        printk("myread() wait begin!\n");
        wait_event_interruptible(myqueue, flag != 0);
        flag = 0;
        printk("myread() wait end!\n");

        //return sizeof(int);
        return 0;
}

int mywrite(struct file *fl, const char * buf, size_t size, loff_t * loff)
{
        printk("mywrite() begin!\n");
        printk("mywrite() &flag = %p\n", &flag);

        printk("mywrite() wake up begin!\n");
        wake_up_interruptible(&myqueue);
        flag = 1;
        printk("mywrite() wake up end!\n");

        return 1;
}
```

```
struct file_operations fops =
{
        .read = myread,
        .write = mywrite,
};


int int_init(void)
{
        int rc;

        printk("int init\n");

        rc = register_chrdev(MA, "mychar", &fops);

        if (rc < 0)
        {
                printk("register failed\n");
                return -1;
        }

        init_waitqueue_head(&myqueue);

        return 0;
}

void int_exit(void)
{
        printk("int exit\n");

        unregister_chrdev(MA, "mychar");

        return;
}

module_init(int_init);
module_exit(int_exit);
```

14.6.3　GPIO 中断代码

```
#include <linux/module.h>        // module_init
#include <asm/io.h>                // ioremap
#include <linux/fs.h>                // file_operations
#include <asm/uaccess.h>        // copy_from_user
#include <linux/sched.h>        // wait_queue

#include <mach/gpio.h>                // S5PV210_GPH2
#include <mach/regs-gpio.h>
#include <linux/interrupt.h>        // requst_irq
#include <linux/irq.h>                // IRQ_TYPE_EDGE_BOTH
```

```
MODULE_LICENSE("GPL");

#define MA        240

volatile int * pbtn;
static int ev_press = 0;

wait_queue_head_t  btn_waitq;

static int irq;

struct btn_desc
{
        int gpio;
        int number;
        char * name;
};

static struct btn_desc btns[] =
{
        { S5PV210_GPH2(0), 0, "KEY0" },
        { S5PV210_GPH2(1), 1, "KEY0" },
        { S5PV210_GPH2(2), 2, "KEY0" },
};


static irqreturn_t btn_irq_handler(int irq, void * dev_id)
{
        printk("btn irq handler !!!\n");

        ev_press = 1;
        wake_up_interruptible(&btn_waitq);

        return IRQ_HANDLED;
}

int btn_drv_open(struct inode *inode, struct file *filp)
{
        int major, minor;
        int err;

        major = MAJOR(inode->i_rdev);
        minor = MINOR(inode->i_rdev);

        printk("btn drv open: major %d, minor %d\n", major, minor);

        irq = gpio_to_irq(btns[0].gpio);
        printk("irq = %d\n", irq);

        // see include/linux/irq.h for more types
        //err = request_irq(irq, btn_irq_handler, IRQ_TYPE_EDGE_BOTH,
        err = request_irq(irq, btn_irq_handler, IRQ_TYPE_EDGE_FALLING,
                        "btn0", NULL);
        if (err)
        {
                printk("request irq failed!\n");
```

```
        }

        printk("request irq ok! err = %d\n", err);

        init_waitqueue_head(&btn_waitq);

        return 0;
}

ssize_t btn_drv_read(struct file *filp, char __user * buf, size_t count, loff_t *f_pos)
{
        printk("btn read!\n");

        wait_event_interruptible(btn_waitq, ev_press);
        ev_press = 0;

        return 0;
}

ssize_t btn_drv_write(struct file *filp, const char __user * buf, size_t count, loff_t *f_pos)
{
        //char kbuf[128];

        //buf[count] = '\0';
        printk("btn drv write %d\n", count);

//        printk("buf = %s\n", buf);
//        printk("buf at %p\n", buf);
        printk("count = %d\n", count);

//        copy_from_user(kbuf, buf, count);
        *pbtn = buf[0];

//        printk("kbuf = %s\n", kbuf);
//        printk("kbuf at %p\n", kbuf);

        return count;
}

int btn_drv_release(struct inode *inode, struct file *filp)
{
        printk("btn drv release ok!\n");

        free_irq(irq, NULL);

        return 0;
}

struct file_operations btn_fops =
{
        .owner = THIS_MODULE,
        .open = btn_drv_open,
        .read = btn_drv_read,
        .write = btn_drv_write,
        .release = btn_drv_release,
};
```

```
static int btn_drv_init(void)
{
        int rc;

        printk("btn init \n");

        pbtn = ioremap(0xE0200284, 4);

        //*pbtn = 0;
        rc = register_chrdev(MA, "akae", &btn_fops);
        if (rc < 0)
        {
                printk("register failed\n");
                return -1;
        }

        printk("register char ok %d!\n", rc);

        return 0;
}

static void btn_drv_exit(void)
{
        printk("btn exit \n");

        //*pbtn = 0xF;
        unregister_chrdev(MA, "akae");

        printk("unregister char ok!\n");

        return;
}

module_init(btn_drv_init);
module_exit(btn_drv_exit);
```

参考实现 arch/arm/mach-s5pv210/include/mach/gpio.h:175:

```
#define S5PV210_GPH2(_nr)       (S5PV210_GPIO_H2_START + (_nr))
```

arch/arm/mach-s5pv210/include/mach/gpio.h

```
        S5PV210_GPIO_H2_START       = S5PV210_GPIO_NEXT(S5PV210_GPIO_H1),

#define S5PV210_GPIO_NEXT(__gpio) \
        ((__gpio##_START) + (__gpio##_NR) + CONFIG_S3C_GPIO_SPACE + 1)

#define S5PV210_GPIO_H2_NR      (8)
```

如何从 gpio 号获得中断号 irq　查看以下两个文件：

arch/arm/mach-s5pv210/include/mach/irqs.h
arch/arm/plat-samsung/include/plat/irqs.h

也可以通过 gpio_to_irq 得到中断号：

```
drivers/gpio/gpiolib.c:1575: * __gpio_to_irq() - return the IRQ corresponding to a GPIO
drivers/gpio/gpiolib.c:1583:int __gpio_to_irq(unsigned gpio)

/**
 * __gpio_to_irq() - return the IRQ corresponding to a GPIO
 * @gpio: gpio whose IRQ will be returned (already requested)
 * Context: any
 *
 * This is used directly or indirectly to implement gpio_to_irq().
 * It returns the number of the IRQ signaled by this (input) GPIO,
 * or a negative errno.
 */
int __gpio_to_irq(unsigned gpio)
{
        struct gpio_chip        *chip;

        chip = gpio_to_chip(gpio);
        return chip->to_irq ? chip->to_irq(chip, gpio - chip->base) : -ENXIO;
}
EXPORT_SYMBOL_GPL(__gpio_to_irq);
```

arch/arm/mach-s5pv210/mach-mini210.c
arch/arm/mach-s5pv210/mach-mini210.c:2128: .init_irq = s5pv210_init_irq,
arch/arm/mach-s5pv210/cpu.c:150:void __init s5pv210_init_irq(void)
arch/arm/plat-samsung/include/plat/cpu.h:51:extern void s5p_init_irq(u32 *vic, u32 num_vic);
arch/arm/plat-s5p/irq-eint.c:217:int __init s5p_init_irq_eint(void)

14.6.4　UART 中断代码

```
#include <linux/module.h>
#include <linux/fs.h>        // register_chrdev
#include <asm/io.h>
#include <plat/map-base.h>
#include <plat/regs-serial.h>
//#include <asm/uaccess.h>

#define MA        243

MODULE_LICENSE("GPL");

struct uart_sfr
{
        int ulcon;
        int ucon;
```

```
        int ufcon;
        int umcon;
        int utrstat;
        int uerstat;
        int ufstat;
        int umstat;
        int utxh;
        int urxh;
        int ubrdiv;
        int udivslot;
};

typedef struct uart_sfr USFR;

static volatile USFR *puart;
static volatile USFR *puart2;

//#define printk                noprintk

int noprintk(char * fmt, ...)
{
        return 0;
}

#define PRINT(x)        printk(#x " = 0x%x\n", x);

#include <linux/irq.h>
#include <linux/interrupt.h>                // request_irq

// see it in arch/arm/plat-s5p/include/plat/irqs.h
#define UART_RX                         (28 + 0)
#define UART_ERR                (28 + 1)
#define UART_TX                         (28 + 2)

int flag = 0;
wait_queue_head_t  uart_waitq;

int myputchar(char);

char outputc;

static irqreturn_t uart_read_irq_handler(int irq, void * dev_id)
{
        char c;

        printk("in read irq!!!\n");

        flag = 1;
        wake_up_interruptible(&uart_waitq);

        c = puart2->urxh;
        myputchar(c);

        return IRQ_HANDLED;
}
```

```
static irqreturn_t uart_write_irq_handler(int irq, void * dev_id)
{
        static int i = 0;

        printk("in write irq!!! %d \n", i++);

        puart2->utxh = outputc;

        disable_irq_nosync(UART_TX);
        //disable_irq(UART_TX);

        return IRQ_HANDLED;
}

int uart_open(struct inode * inode, struct file * filp)
{
        int major = MAJOR(inode->i_rdev);
        int minor = MINOR(inode->i_rdev);
        int * p;
        int i;
        int ri;

        printk("uart open: major %d, minor %d\n", major, minor);

//        puart = ioremap(0xe2900000, sizeof(USFR));        // uart0
//        puart = ioremap(0xe2900000, 0x10000);        // uart0
        puart = S3C24XX_VA_UART0;

        puart2 = ioremap(0xe2900c00, sizeof(USFR));        // uart3
//        puart2 = S3C24XX_VA_UART1;
        p = (int *)puart;
        PRINT((int)p);
        for (i = 0; i < sizeof(USFR)/4; i++)
        {
                PRINT(*p++);
        }
//        puart = (USFR *)(((int)puart) + 0xc000);
//        puart->ufcon = 0;                // fifo is important!
        p = (int *)puart2;
        PRINT((int)p);
        for (i = 0; i < sizeof(USFR)/4; i++)
        {
                PRINT(*p++);
        }

        puart2->ucon = 0x7c5;
        puart2->ubrdiv = 0x23;                // 115200
        puart2->udivslot = 0x808;        // 115200

        p = (int *)puart2;
        PRINT((int)p);
        for (i = 0; i < sizeof(USFR)/4; i++)
        { PRINT(*p++);
        }
#if 0
        puart->ulcon = 3;
```

```
        puart->ucon = 0x7c5;
//        puart->ufcon = 0x111;               // fifo is important!
        puart->umcon = 0;
        puart->ubrdiv = 0x23;               // 115200
        puart->udivslot = 0x808;        // 115200
#endif

        ri = request_irq(UART_RX, uart_read_irq_handler, 0, "read_irq", NULL);
        if (ri)
                printk("request irq %d failed! ri = %d\n", UART_RX, ri);
        else
                printk("request irq %d ok! ri = %d\n", UART_RX, ri);

        init_waitqueue_head(&uart_waitq);

        ri = request_irq(UART_TX, uart_write_irq_handler, 0, "write_irq", NULL);
        if (ri)
                printk("request irq %d failed! ri = %d\n", UART_TX, ri);
        else
                printk("request irq %d ok! ri = %d\n", UART_TX, ri);

        printk("uart open finished!\n");

        return 0;
}

int uart_release(struct inode * inode, struct file * filp)
{
        printk("uart release\n");

        free_irq(UART_RX, NULL);
        free_irq(UART_TX, NULL);
        printk("free irq %d ok! \n", UART_RX);
        printk("free irq %d ok! \n", UART_TX);

        return 0;
}

#if 0
int uart_putchar(char c)
{
        while ((puart->utrstat & (1<<2)) == 0)
                ;

        puart->utxh = c;
#if 1
        while ((puart2->utrstat & (1<<2)) == 0)
                ;

        puart2->utxh = c;
#endif

        return 0;
}
#else
```

```
int uart_putchar(char c)
{
        // uart0 putchar using polling
#if 0
        while ((puart->utrstat & (1<<2)) == 0)
                ;

        puart->utxh = c;
#endif
        printk("uart putchar %c(%d)\n", c, c);

        // uart3 putchar using irq
        outputc = c;
        enable_irq(UART_TX);

        return 1;
}

#endif

int myputchar(char c)
{
        if (c == '\n')
                uart_putchar('\r');

        uart_putchar(c);

        return 0;
}


int uart_read(struct file * filp, char __user * buf, size_t count, loff_t *f_pos)
{
        printk("uart read\n");

        wait_event_interruptible(uart_waitq, flag != 0);
        flag = 0;

        return 0;
}

int uart_write(struct file * filp, const char __user * buf, size_t count, loff_t *f_pos)
{
        int i;

        printk("uart write\n");

        printk("buf = %c\n", buf[0]);
        printk("count = %d\n", count);

        for (i = 0; i < count; i++)
        {
                //myputchar(buf[i]);
                uart_putchar(buf[i]);
        }
```

```
//        myputchar(buf[0]);

        return count;
}

int uart_ioctl(struct inode * inode, struct file * filp, unsigned int cmd, unsigned long arg)
{
        printk("uart ioctl\n");

        printk("cmd = %d\n", cmd);

        if (cmd == 19200)
                puart->ubrdiv = 0xd5;

        if (cmd == 115200)
                puart->ubrdiv = 0x23;

        return 0;
}

struct file_operations uart_fops =
{
        .owner = THIS_MODULE,
        .open = uart_open,
        .release = uart_release,
        .write = uart_write,
        .read = uart_read,
        //.ioctl = uart_ioctl,
};


int uart_init(void)
{
        int rc;

        rc = register_chrdev(MA, "myuart", &uart_fops);

        if (rc < 0)
        {
                printk("register chrdev failed! %d\n", rc);
                return -1;
        }

        printk("uart init ok \n");

        return 0;
}

void uart_exit(void)
{
        printk("uart exit ok \n");

        unregister_chrdev(MA, "myuart");


        return;
```

```
}


module_init(uart_init);
module_exit(uart_exit);
```

第 15 章

Linux 驱动的并发控制

15.1 并发与竞态

15.1.1 什么是并发与竞态

并发（concurrency）是指多个执行单元同时、并行的被执行，而并发执行单元对共享资源（硬件、全局变量、静态变量）的访问则很容易导致竞态（race condition）

15.1.2 并发与竞态发生的条件

```
1、对称多处理器（SMP）的多个CPU。
2、单CPU内进程与强占它的进程。
3、中断（硬件中断，软中断，tasklet）与进程之间。
```

15.1.3 解决并发与竞态的途径

解决竞态的最根本的途径是对共享资源的互斥访问，访问共享资源的代码区域成为临界区（critical sections），对临界区的代码需要以某种互斥机制加以保护。

常用的互斥机制有：中断屏蔽 、原子操作、自旋锁、信号量。

15.2 中断屏蔽

15.2.1 中断屏蔽

在单CPU范围内避免竞态的一个简单的办法就是屏蔽中断。 这种办法可以保证正在执行的内核路径不会被中断处理程序强占，防止中断与进程之间竞态条件的发生。 另外由于linux内核的进程调度和异步IO等操作都是依赖中断来实现的，所以中断屏蔽也可以避免内核强占进程之间的竞态发生。

15.2.2 中断屏蔽使用方法

定义在 linux/irqflags.h ，实现在 asm/irqflags.h 中

```
local_irq_disable()
……
critical section //临界区
……
local_irq_enable()
```

15.2.3　中断屏蔽的注意事项

1、local_irq_disable()只能禁止本地CPU的中断，所以不能解决SMP多CPU引发的竞态。

2、中断对系统正常运行很重要，所以长时间屏蔽中断很危险，有可能会造成数据丢失或系统崩溃，所以要求在中断屏蔽之后当前的内核执行路径尽可能快的执行完毕。

15.2.4　中断屏蔽方法

```
local_irq_disable(void)
local_irq_enable(void)

local_irq_save(flags)
local_irq_restore(flags)

local_bh_disable(void)
local_bh_enable(void)
```

15.3　原子操作

原子操作指的是在执行操作的过程中不会被别的代码路径所中断。

原子操作分类：整形原子操作、位原子操作。

特点：

1、任何情况下操作都是原子的。

2、都依赖底层CPU的原子操作来实现，所以和CPU架构密切相关。

15.3.1　整形原子操作

定义和实现都在 asm/atomic.h 中。

```
void atomic_set(atomic_t *v,int i);
设置原子变量的值为i。
atomic_t v=ATOMIC_INIT(0);
定义原子变量v,并初始化为0。
```

设置原子变量的值

```
atomic_read(atomic_t *v);
返回原子变量的值
```

获取原子变量的值

```
void atomic_add(int i,atomic_t *v);
原子变量加i
void atomic_sub(int i,atomic_t *v);
原子变量减i
```

原子变量加减

```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
```

原子变量自增自减

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(atomic_t *v);
操作后测试是否为0，为0返回ture，否则返回false
```

操作并测试

```
int atomic_add_return(int I,atomic_t *v);
int atomic_sub_return(int I,atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
操作后返回新的值。
```

操作并返回

15.3.2 位原子操作

定义和实现都在 asm/bitops.h 中。

```
void set_bit(nr,void *addr)
设置addr指针指向的值的nr位为1。
```

设置位

```
void clear_bit(nr, void *addr)
清除addr地址的值nr位为0。
```

清除位

```
void change_bit(nr,void *addr)
对addr地址的值第nr位取反。
```

改变位

```
test_bit(nr,void *addr)
返回addr地址的值第nr位的值。
```

测试位

```
int test_and_set_bit(nr,void *addr)
int test_and_clear_bit(nr,void *addr)
int test_and_change_bit(nr,void *addr)
```

测试并操作位 上述操作等同于先执行test_bit(nr,voidaddr)然后在执行xxx_bit(nr,voidaddr)

原子操作案例 设想在不同CPU运行的两个进程都获取了某个共享资源，并对该共享资源计数都加了一， 现在不 想使用该共享资源了，就需要递减该计数值（共 享资源），可能发生的情况是:

```
1. CPU A（上的进程，以下同）从内存单元把 当前计数值（2）装载进它的寄存器中；
2. CPU B从内存单元把当前计数值（2）装载进它的寄存器中。
3. CPU A在它的寄存器中将计数值递减为1；
4. CPU B在它的寄存器中将计数值递减为1；
5. CPU A把修改后的计数值（1）写回内存单元。
6. CPU B把修改后的计数值（1）写回内存单元。
```

我们期望内存里的计数值应该是0，然而它却是1。如果该计数值是一个共享资源的引用计数，每个进程都在递减后把该值与0进行比较，从而确定是否需要释放该共享资源。 这时，两个进程都去掉了对该共享资源的引用，但没有一个进程能够释放它 两个进程都推断出：计数值是1，共享资源仍然在被使用。

15.4 自旋锁

15.4.1 自旋锁的定义

自旋锁是一种对临界资源进行互斥访问的典型手段，其名来源于它的工作方式， 通俗的讲，自旋锁就是一个变量，该变量把一个临界区标记为“我当前在运行，请等待”或者标记为“我当前不在运行，可以被使用”， 如果A执行单元首先获得锁，那么当B进入同一个例程时将获知自旋锁已被持有，需等待A释放后才能进入，所以B只好原地打转（自旋）。

15.4.2 自旋锁的特点

1、自旋锁主要针对SMP或单CPU且内核可 抢占的情况，对于单CPU且内核不可抢占的系统自旋锁退化为空操作。

2、在单CPU且内核可抢占的系统中，自旋锁持有期间内核的抢占被禁止。

3、尽管自旋锁可以保证临界区不受别的 CPU和本CPU内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候还可 能受到中断和底半部影响，此时应该使用 自旋锁的衍生操作。

15.4.3 自旋锁使用的注意事项

1、自旋锁实际上是忙等待锁，当锁不可用时，CPU一直循环的执行“测试并设置”该锁，直到可用而取得该锁，CPU在等待自旋锁时不做任何有用的工作，仅仅是等待。 因此只有占用锁的时间极短的情况下使用自旋锁才是合理的。如果临界区很大并且有共享的设备时使用自旋锁会降低系统的性能。

2、自旋锁有可能导致死锁。

A、同一CPU或进程递归使用自旋锁。

B、获得自旋锁后发生阻塞，一般引起阻塞的函数有 copy_from_user(),copy_to_user(),kmalloc()等，所以在自旋锁的占用期间内不能调用这些函数。

15.4.4 自旋锁的操作函数

定义在 linux/spinlock_types.h

```
1、定义自旋锁
spinlock_t lock;

2、初始化自旋锁
spin_lock_init(&lock);

3、获得自旋锁
spin_lock(&lock);
如不能获得，原地打转。
spin_trylock(&lock);
```

```
尝试获得，如能获得返回真，不能获得返回假，不在原地打转。

4、释放自旋锁
spin_unlock(&lock);
与spin_lock()和spin_trylock()配对使用。

5、自旋锁衍生操作
spin_lock_irq()
spin_unlock_irq()
spin_lock_irqsave()
spin_unlock_irqrestore()
spin_lock_bh()
spin_unlock_bh()
```

15.4.5　使用范例

```
spinlock_t lock;

spin_lock_init(&lock);

spin_lock(&lock)
…..(临界区)
spin_unlock(&lock)
```

15.4.6　自旋锁的衍生（略）

1、读写自旋锁

读写自旋锁（rwlock）是一种比自旋锁粒度更小的自旋锁机制，它保留了“自旋”的概念，但是在写操作方面，只能最多有一个写进程，在读方面，同时可拥有多个执行单元，当然读和写也不能同时进行。

2、顺序锁

顺序锁（seqlock）是对读写锁的一种优化，若使用顺序锁，读执行单元不会被写执行单元阻塞，写执行单元也不需要等待所有的读执行单元完成读操作才去进行写操作。 但写执行单元和写执行单元之间是互斥的。如果在读执行单元读的过程中写执行单元发生了操作，那么读执行单元必须重新读取数据，以确保得到的数据是完整的。

15.5　信号量

15.5.1　信号量的操作

定义在 linux/semaphore.h 中，实现在 ipc/sem.c 和 kernel/semaphore.c 中

```
1、定义信号量
struct semaphore sem;

2、初始化一个信号量
```

```
void sema_init(struct semaphore *sem,int val);
初始化该信号sem，并把它的值设为val。
void init_MUTEX(struct semaphore *sem);
初始化一个信号量，并把信号量的值设为1。
void init_MUTEX_LOCKED(struct semaphore);
初始化一个信号量，并把该值设为0。

3、定义并初始化的快捷方式
DECLARE_MUTEX(name)；
DECLARE_MUTEX_LOCKED(name)；

4、获得信号量
void down(struct semaphore *sem);
用于获得信号量，他会导致睡眠，所以不能在中断上下文中使用，进入睡眠后不能用信号打断。
int down_interruptible(struct semaphore *sem);
进入睡眠后可被信号中断，此时返回值为非0，一般如果得到非0的值就立即返回-ERESTARTSYS；
int down_trylock(struct semaphore *sem);
尝试获得信号量sem，如果能够立刻获得该信号量就获得该信号量，并返回0，否则返回非0，它不会导致调用者睡眠，可以在中断上下文中使用。

5、释放信号量
void up(struct semaphore *sem)

6、信号量使用方法：
DECLARE_MUTEX(mount_sem);
down(&mount_sem);
……//临界区
up(&mount_sem);
```

15.5.2 信号量的衍生（略）

读写信号量 读写信号量与信号量的关系和读写自旋锁与自旋锁的关系类似，读写信号量可能引起进程阻塞，但它可以允许多个读执行单元同时访问共享资源，而最多只有一个写执行单元。

15.6 互斥体

互斥体完成的功能和信号量很相似。

15.6.1 互斥体的操作

定义在 linux/mutex.h 中，实现在 kernel/mutex.c 中。

```
struct mutex my_mutex;
mutex_init(&my_mutex);
```

定义并初时化互斥体

```
void fastcall mutex_lock(struct mutex *lock);
int fastcall mutex_lock_interruptiable(struct mutex *lock);
int fastcall mytex_trylock(struct mutex *lock);
```

获取互斥体

```
void fastcall mutex_unlock(struct mutex        *lock);
```

释放互斥体

```
struct mutex my_mutex;
mutex_init(&my_mutex);
mutex_lock(&my_mutex);
……
mutex_unlock(&my_mutex);
```

互斥体使用模板

15.6.2　总结

自旋锁 VS 信号量

1、信号量是进程级的，用于多个进程之间对资源的互斥，虽然也是在内核中，但是该内核执行路径是以进程的身份。 如果竞争失败，会发生进程上下文切换，当前进程进入休眠状态，CPU运行其他进程。 因为进程上下文切换开销很大，所以只有当进程占用资源时间较长时，用信号量才是较好的选择。

2、自旋锁得不到锁时就会在原处自旋一直到获得锁，它节省了上下文切换的时间，所以一般用于要保护的临界区访问时间比较短的时候，否则会降低系统效率。

3、自旋锁和信号量选用的原则：

```
A、临界区执行时间比较小时，采用自旋锁，否则使用信号量。

B、信号量所保护的临界区包含可能引起阻塞的代码，而自旋锁要绝对避免在临界区中使用该类型的代码。

C、信号量存在于进程上下文，因此如果被保护的共享资源要在中断或软中断情况下使用，则应该选用自旋锁，
如果一定要使用信号量，则只能通过 down_trylock()的方式进行，不能获取就立即返回，避免阻塞。
```

第 16 章

Linux 网络设备驱动

16.1 驱动硬件基础

16.1.1 DM9000 的物理连接

* RJ45 接口 8 根线
* 重要的4根线被引出
  pin1 pin3 分别是 TX+/RX+
  pin2 pin6 分别是 TX-/RX-
  能够区分对连和直连（交叉和级联）1< >3 2< >6

16.1.2 DM9000 与 SoC 芯片的连接

* 使能的问题
  XmOCSn1 – Bank 片选
  0x8800_0000 0x8FFF_FFFF 128MB SROM Bank 1
* 控制的问题
  CMD – 1：数据周期 0：地址周期
  Control & Status register
  如何能够被 SoC 芯片访问到 40多个内部寄存器
* 数据的问题
  DATA0 – DATA15

16.1.3 两个端口-地址口和数据口

```
#define DM_ADDR_PORT (*((volatile unsigned short *) 0x88000000)) //地址口
#define DM_DATA_PORT (*((volatile unsigned short *) 0x88000004)) //数据口
```

向地址端口写的数据，作为选择寄存器的编号
向数据端口写的数据，作为配置上面那个寄存器的配置值

16.1.4 端口的读写接口

```
//写DM9000寄存器
void __inline dm_reg_write(unsigned char reg, unsigned char data)
{
        DM_ADDR_PORT = reg;            //将寄存器地址写到地址端口
        DM_DATA_PORT = data;            //将数据写到数据端口
}

//读DM9000寄存器
unsigned char __inline dm_reg_read(unsigned char reg)
{
        DM_ADDR_PORT = reg;
        return DM_DATA_PORT;             //将数据从数据端口读出
}
```

16.1.5 通过读写接口获取芯片ID

```
void dm_read_id(char id[])
{
        id[0] = dm_reg_read(DM9000_VIDL);
        id[1] = dm_reg_read(DM9000_VIDH);
        id[2] = dm_reg_read(DM9000_PIDL);
        id[3] = dm_reg_read(DM9000_PIDH);

        return;
}
```

16.2 网络数据封装过程

16.2.1 TCP/IP协议栈结构

```
1. 应用层
2. 运输层
3. 网络层
4. 链路层
```

网络协议栈层次图

```
1. FTP客户端-服务器
2. TCP协议
3. IP协议
4. 以太网协议
```

传输过程

```
1. User APP
2. TCP/UDP
3. ICMP/IP/IGMP
4. ARP/RARP
```

协议分层

```
1. 用户数据 -> 应用数据
2. TCP头 + 应用数据
3. IP头 + TCP头 + 应用数据
4. 以太网帧头 + IP头 + TCP头 + 应用数据 + 以太网帧尾
```

协议封装过程

16.2.2 驱动和协议层的关系

网络协议接口层

1、网络协议接口层向网络层协议提供提供统一的数据包收发接口，不论上层协议为ARP还是IP，都通过dev_queue_xmit()函数发送数据，并通过netif_rx()函数接受数据。这一层的存在使得上层协议独立于具体的设备。

2、网络设备接口层向协议接口层提供统一的用于描述具体网络设备属性和操作的结构体net_device，该结构体是设备驱动功能层中各函数的容器。实际上，网络设备接口层从宏观上规划了具体操作硬件的设备驱动功能层的结构。

3、设备驱动功能层各函数是网络设备接口层net_device数据结构的具体成员，是驱使网络设备硬件完成相应动作的程序，他通过hard_start_xmit()函数启动发送操作，并通过网络设备上的中断触发接受操作。

4、网络设备与媒介层是完成数据包发送和接受的物理实体，包括网络适配器和具体的传输媒介，网络适配器被驱动功能层中的函数物理上驱动。对于Linux系统而言，网络设备和媒介都可以是虚拟的。

16.3 网络设备驱动程序框架

16.3.1 网络设备驱动程序框架

上层协议发送数据包时 dev_queue_xmit(struct sk_buf *skb)

上层协议接受数据包时 int netif_rx(struct sk_buff *skb)

struct sk_buff 套接字缓冲区，用于在linux网络子系统中各层之间传递数据，是linux网络子系统数据传输的中枢神经。

struct sk_buff (linux/skbuff.h) 中的关键成员：

各层协议头　A 传输层协议头：

```
union {
        struct tcphdr *th;
        struct udphdr *uh;
        struct icmphdr *icmph;
        struct igmphdr *igmph;
        struct iphdr *ipiph;
        struct ipv6hdr *ipv6h;
        unsigned char *raw;//数据链路层头部
} h;
```

B 网络层协议头

```
union {
        struct iphdr *iph;
        struct ipv6hdr      *ipv6h;
        struct arphdr *arph;
        unsigned char     *raw;
} nh;
```

C 链路层协议头

```
union {
        unsigned char  *raw;
} mac;
```

```
unsigned char *head :
指向内存中已经分配的用于承载网络数据的缓冲 区起始地址，sk_buff和相关数据块在分配后该指针的值就被固定了。

unsigned char *data :
指向对应当前协议层有效数据的起始地址，每层协议有效数据含义不同。

unsigned char *tail :
指向当前协议层有效数据负载的结尾地址。

unsigned char *end :
指向分配的数据缓冲区的结尾，分配完即固定。
```

数据缓存区指针：　　注意：网络报文在内存中不一定是联系存储的，同一网络报文有可能分成若干片存储在内存的不同位置：

长度信息

```
unsigned int data_len:
记录在frags和frag_list中网络报文的长度。

unsigend int len :
记录网络报文的总长度。

unsigned int truesize :
记录head所指的存储区的大小。
```

套接字缓冲区操作 1、分配:

```
struct sk_buff *alloc_skb(unsigned int len,int priority);
分配一个套接字缓冲区和一个数据缓冲区，len为数据缓冲区大小，16字节对齐。

struct sk_buff *dev_alloc_skb(unsigned int)
以GFP_ATOMIC方式调用上面接口，并保留head和data之间的16个字节。
```

2、释放

```
void kfree_skb(struct sk_buff *skb)
内核内部使用。

void dev_kfree_skb(struct sk_buff *skb)
驱动中使用，用于非中断上下文。

void dev_kfree_skb_irq(kfree_skb(struct sk_buff *skb)
驱动中使用，用于中断上下文。

void dev_kfree_skb_any(kfree_skb(struct        sk_buff *skb)
驱动中使用，中断和非中断上下文都可以。
```

指针移动 A、put操作:

```
将tail指针下移，增加sk_buff的len值，并返回skb->tail的当前值，用于在缓冲区尾部添加数据。
unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
```

B、push操作

```
将data指针上移，增加len的值，在存储空间的头部增加一段可以存储网络数据的空间，主要用于在数据包发送时添加头部。
unsigned char *skb_push(struct sk_buff *skb, unsigned int len)
```

C、pull操作:

```
将data指针下移，并减小skb的len的值，一般用于下层协议向上层协议移交数据包，使data指向上一层协议的协议头。
unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)
```

D、reserve操作：

```
将data指针和tail指针同时下移，主要用于在存储空间的头部预留len长度的空隙。
unsigned char *skb_reserve(struct sk_buff *skb, unsigned int len)
```

UDP包接收上传至应用程序实例

数据链路层：dev_alloc_skb(),skb_pull()

网络层： skb_pull()

传输层：

应用层：

调用recv()接收数据时，从skb->data+sizeof(struct udphdr)的位置开始复制数据到应用层缓冲区，由此可见UDP头部始终没有被剥离。

驱动需要实现的功能：

```
1、网卡的初始化
2、数据的发送
3、数据的接收（中断）
4、其它（流量控制，错误统计，超时处理）
```

* 模块初始化函数。注册（register_netdev）网络设备的net_device变量。
* 模块销毁函数。注销（unregister_netdev）网络设备的net_device变量 。
* 声明net_device变量，并实现如下函数：
  - 实现init函数，完成设备的初始化和net_device结构体变量自身的初始化。
  - 实现打开设备(open)和关闭设备(stop)
  - 实现发送函数(hard_start_transmit)
  - 实现中断处理函数，处理中断。
  - 实现数据接收函数。
  - 实现其他的函数get_stats,do_ioctl

net_device 结构细节

* Linux内核源代码include/linux/netdevice.h
* 重要的域
  - name：接口名称。
  - dev_addr：链路层地址，可以认为是MAC地址。
  - priv：私有数据指针。
  - init()：初始化函数。

  - open()：接口打开。
  - stop()：接口关闭。
  - hard_start_xmit ()：发送数据。

注册驱动程序 register_netdev(struct net_device *dev)：

内核以dev_base为头指针的设备链表来集体管理所有网络设备，该设备链表中的每个元素代表一个网络设备接口。register_netdev把注册的dev插入到dev_base链表中。

注销驱动程序 unregister_netdev(struct net_device *dev)：

从dev_base链表移除dev设备

模块初始化：主要完成注册（register_netdev）网络设备的工作。 网络设备初始化：既可以放在init_module函数中完成，也可由net_device数据结构中的init函数指针所指的初始化函数来完成的。 工作包括：

* 首先检测网络物理设备是否存在，这是通过检测物理设备的硬件特征来完成，然后再对设备进行资源配置。
* 构造并初始化设备的net_device数据结构，并把检测到的数值来对net_device中的变量初始化，比如mac地址等。

功能：

```
- 初始化硬件，准备发送数据。
- 注册中断处理函数（可以在net_device的init函数中注册）。
- 启动接口的传输队列（允许接口接受传输数据 包）。内核提供的如下函数可启动该队列 netif_start_queue。
- 调用open，通过如下方法：
        ifconfig eth0 up
        ifconfig eth0 192.168.x.x
```

内核要传输一个数据包，会调用 hard_start_xmit 方法将数据放入发送队列。内核处理后的每个数据包位于一个套接字缓冲区结构（struct sk_buff）里。 这个缓冲区包含了物理数据包（以它在介质上的格式），并拥有完整的传输层数据包头。接口无需修改要传输的数据。

* hard_start_xmit函数的处理过程：
  - 得到要发送数据的起始地址和长度。skb->data 指向要传输的数据包，而 skb->len 是以字节为单位的长度。
  - 把数据发送到网卡的物理缓冲区中。可以直接把数据写入缓冲区寄存器中；或者通过DMA的方式发送数据。
  - 网卡把得到的数据发送到物理媒介中去。这一步一般由网卡自动完成。
* 处理接收数据包的方式：一般通过中断方式
  - 中断方式：当有新数据到来时，中断处理器操作，并调用中断处理程序接收新数据。
  - 轮询方式：每隔一定间隔访问网卡寄存器，判断是否有新数据到来。如果有则接收数据。

两种方式都可以。但中断方式效率高，占用系统资源少。

* 大多数网卡都可以通过中断处理程序来控制。中断可能产生的原因：

- 新数据包到达
- 外发数据包的传输已经完成
- 数据发送或接收错误。

* 处理流程:
  - 读取中断寄存器，判断中断发生的原因。
  - 如果中断是新数据到来，进入数据接收函数。
  - 如果中断是数据包发送完成，或者产生传输错误，进行数据统计。

16.4 DM9000 驱动代码实现

16.5 Linux 网卡驱动实验

16.5.1 标准Board Linux启动 & uBuntu Linux 连接

* pc linux (192.168.0.200)
* board linux (192.168.0.201)

  ifconfig sudo ifconfig eth0 192.168.0.xxx

测试网络连接 对ping可以连通

16.5.2 uboot 启动 & uBuntu Linux 连接

* 开发板上的跳线 S2 拨到 SDBOOT 那边，通过 SD 卡启动
* 启动之后进入 uboot 选项
* [FriendlyLEG-TINY210]# printenv

  [FriendlyLEG-TINY210]# printenv baudrate=115200 bootdelay=3 ethact=dm9000 ethaddr=00:01:02:03:04:05 gatewayip=192.168.0.1 ipaddr=192.168.0.201 netmask=255.255.255.0 serverip=192.168.0.200 stderr=serial stdin=serial stdout=serial

  [FriendlyLEG-TINY210]# setenv ipaddr 192.168.0.201 [FriendlyLEG-TINY210]# setenv serverip 192.168.0.200 [FriendlyLEG-TINY210]# saveenv Saving Environment to NAND… Erasing Nand… Erasing at 0x40000 100% complete. Writing to Nand… done [FriendlyLEG-TINY210]#
* 重启开发板，验证是否设置成功? 看到 is alive 表示成功

  [FriendlyLEG-TINY210]# ping 192.168.0.200 dm9000 i/o: 0x88001000, id: 0x90000a46

  DM9000: running in 16 bit mode

  MAC: 00:01:02:03:04:05

  operating at 100M full duplex mode

  Using dm9000 device

  host 192.168.0.200 is alive

  [FriendlyLEG-TINY210]#

16.5.3 uboot 和 uBuntu 之间的 tftp 连接

ubuntu下搭建tftp server

* Install tftpd and related packages.

  $ sudo apt-get install xinetd tftpd tftp

* Create /etc/xinetd.d/tftp and put this entry:
  service tftp { protocol = udp port = 69 socket_type = dgram wait = yes user = nobody server = /usr/sbin/in.tftpd server_args = /tftpboot disable = no }
* Make /tftpboot directory
  $ sudo mkdir /tftpboot $ sudo chmod -R 777 /tftpboot $ sudo chown -R nobody /tftpboot
* Start tftpd through xinetd
  $ sudo /etc/init.d/xinetd restart
* test tftp in localhost
  $ cp something /tftpboot
  $ tftp localhost
  $ get something Received 4315348 bytes in 0.4 seconds
  $ tftp> quit $ ls something
* test tftp in board $ tftp 21000000 zImage

16.5.4 uboot 和 uBuntu 之间的 uImage 内核下载测试

```
# setenv bootargs root=/dev/mtdblock4 console=ttySAC0,115200 init=/linuxrc lcd=S70
# setenv ipaddr 192.168.0.201
# setenv serverip 192.168.0.200
# saveenv
# printenv
```

设置 uboot 参数

```
# tftp 21000000 uImage
# bootm 21000000
```

制作 uImage 内核文件

获得 mkimage

16.5.5 DM9000 驱动代码实现

平台设备的注册 platform_device　arch/arm/mach-s5pv210/mach-mini210.c

平台驱动的注册 platform_driver　drivers/net/dm9000.c

头文件的包含　drivers/net/dm9000.h include/linux/dm9000.h

几个重要的文件

* .config
  make menuconfig 的配置选项保存在这个文件中
  CONFIG_DM9000 = m

* include/generated/autoconf.h
  make 执行时会通过脚本分析 .config 文件，生成 autoconf.h 参与内核编译
  #define CONFIG_DM9000 1
* driver/net/makefile
  obj-$(CONFIG_DM9000) = dm9000.o

```
obj-m := dm9000.o

KDIR := /home/limingth/tiny210/src/linux-2.6.35.7

modules:
        make -C $(KDIR)        SUBDIRS=$(PWD)  $@
        ls -l dm9000.ko

clean:
        -rm *.o *.ko
```

Makefile

16.5.6 step by step

* install tftpd
  sudo apt-get install tftpd
* install inetd
  sudo apt-get install openbsd-inetd
  server dir /srv/tftp
  sudo vi /etc/inetd.conf

```
  #:BOOT: TFTP service is provided primarily for booting.  Most sites
  #       run this only on machines acting as "boot servers."
  #tftp                dgram        udp        wait        nobody        /usr/sbin/tcpd        /
  usr/sbin/in.tftpd /srv/tftp
  tftp                dgram        udp        wait        nobody        /usr/sbin/tcpd        /
  usr/sbin/in.tftpd /home/limingth/tiny210
```

  sudo /etc/init.d/openbsd-inetd restart
* download Image
  limingth@ubuntu:~$ tftp 127.0.0.1
  tftp> get zImage2
  tftp> quit
* change u-boot bootargs

操作步骤

```
[FriendlyLEG- TINY210]#  setenv  bootargs  root=/ dev/ mtdblock4  console=ttySAC0, 115200  init=/
linuxrc lcd=S70
```

```
[FriendlyLEG-TINY210]# printenv
baudrate=115200
bootargs=root=/dev/mtdblock4 console=ttySAC0,115200 init=/linuxrc lcd=S70
bootdelay=3
ethact=dm9000
ethaddr=08:0a:0a:0a:0a:0a
gatewayip=192.168.0.1
ip=192.168.0.201
ipaddr=192.168.0.201
netmask=255.255.255.0
serverip=192.168.0.200
stderr=serial
stdin=serial
stdout=serial

Environment size: 315/16380 bytes
```

* 启动命令

  tftp 21000000 uImage bootm 21000000

修改内核源码 按以下方法修改内核以适应uboot，详参见liukun321的文章 http://blog.csdn.net/liukun321/article/details/7383669， 忽略此步骤将会出现加载kernel时卡死在 Uncompressing Linux… done, booting the kernel….

```
arch/arm/mach-s5pv210/include/mach/memory.h 文件26,27行内容,
将Maximum of 256MiB in one bank的限制改为Maximum of 512MiB in one bank 作如下修改：

#defineSECTION_SIZE_BITS   29

#defineNODE_MEM_SIZE_BITS   29
```

```
limingth@ubuntu:~/tiny210/src$ ./mkimage
Usage: ./mkimage -l image
          -l ==> list image header information
     ./mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d data_file[:data_file...] image
          -A ==> set architecture to 'arch'
          -O ==> set operating system to 'os'
          -T ==> set image type to 'type'
          -C ==> set compression type 'comp'
          -a ==> set load address to 'addr' (hex)
          -e ==> set entry point to 'ep' (hex)
          -n ==> set image name to 'name'
          -d ==> use image data from 'datafile'
          -x ==> set XIP (execute in place)
       ./mkimage [-D dtc_options] -f fit-image.its fit-image
       ./mkimage -V ==> print version information and exit
```

```
limingth@ubuntu:~/tiny210/src$ ./mkimage -A arm -O linux -T kernel -C none -a 21000000 -e 21000040 -
n "linux-2.6.28" -d zImage2  myuImage
Image Name:   linux-2.6.28
Created:      Wed Aug 29 13:32:34 2012
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    4288476 Bytes = 4187.96 kB = 4.09 MB
Load Address: 21000000
Entry Point:  21000040
```

生成 uImage　/include/generated/mach-types.h
./arch/arm/include/asm/mach-types.h

```
limingth@ubuntu:~/tiny210/src/linux-2.6.35.7$ vi arch/arm/include/asm/mach-types.h
limingth@ubuntu:~/tiny210/src/linux-2.6.35.7$ vi include/generated/mach-types.h

in line 2950
        #define MACH_TYPE_MINI210              3466
```

16.6　参考速查

16.6.1　头文件和函数

```
#include <linux/kernel.h>        // printk()
#include <linux/module.h>        // module_init()
#include <asm/io.h>                        // ioremap()
#include <linux/fs.h>                 // register_chrdev()
#include <linux/cdev.h>                 // cdev()
#include <linux/interrupt.h>                 // request_irq()/free_irq()
#include <linux/wait.h>                // wait_event/wake_up()

fs/char_dev.c                cdev_init()

#include <asm/uaccess.h>                 // copy_to_user()
```

16.6.2　关联操作汇总

```
open        close
read        write

init        exit

module_init()
module_exit()

led_init()
```

```
led_exit()

printf()        - user
printk()        - kernel

mmap()                - user
ioremap()        - kernel

register_chrdev()
unregister_chrdev()

open()
release()

request_irq()
free_irq()

// wait queue
wait_queue_head_t wq;
init_waitqueue_head(&wq);
wait_event()/wait_event_interruptible()
wake_up()/wake_up_interruptible()
```

16.6.3 驱动的几种模型

```
                user space                        kernel space
-------------------------------------------------------------------
"hello"                printf                                printk
                <stdio.h>                       <linux/kernel.h>

-------------------------------------------------------------------
ledon/off        1.                                1.
                open("/dev/leds");                cdev
                ioctl(fd, No, on);                f_ops

                2.                                2.
                open("/dev/mem");                ioremap(addr, size);
```