

目录

第一部分 ARM 体系结构

DAY01—存储器、工作模式和寄存器.....	11
一、ARM 体系结构.....	11
1. 处理器型号.....	11
2. ARM 处理器性能.....	11
3. 基本流水线.....	11
二、地址总线、数据总线、控制总线和 I/O.....	12
1. 地址总线 AB (AddressBus)	12
2. 数据总线 DB (DataBus)	12
3. 控制总线 CB (ControlBus)	12
三、Flash、RAM 和 Flash 的区别.....	12
1. 随机存取存储器 RAM.....	12
2. 只读存储器 ROM.....	13
3. 闪存 Flash.....	13
四、ARM 处理器的 7 种工作模式.....	13
五、ARM 状态下的寄存器.....	14
DAY02—ARM 指令.....	16
1. 指令格式.....	16
2. 分支跳转指令.....	16
3. 数据传送指令.....	16
4. 加法指令.....	16
5. 减法指令.....	16
5. 逻辑运算指令.....	17
6. 比较指令.....	17
7. 第二操作数移位模式.....	17
8. 单寄存器寻址模式.....	17
9. 单寄存器加载指令.....	18
10. 多寄存器加载指令.....	18
11. 乘法指令.....	19
12. 交换指令.....	19
13. *字节交换指令.....	19
14. 状态寄存器操作指令.....	19
15. 常用条件.....	20
DAY03—ARM 汇编程序设计.....	21
DAY04—烧写程序、点亮 LED 灯.....	22
一、烧写初始化代码.....	22
二、uboot 设置语句.....	22
三、从 tftp 服务器下载文件.....	22
四、烧写 uboot 到 NandFlash.....	22
五、GPIO 口设置 (GPC1_3 为例)	22

六、点亮 led 灯（gpc1_3 为低电平 led 亮）	23
DAY05—伪指令、混合编程.....	25
一、伪指令.....	25
1. 符号声明伪操作.....	25
2. 数据定义伪操作.....	25
3. 汇编与反汇编代码控制伪操作.....	26
4. 预定义控制伪操作.....	27
二、汇编、连接.....	27
三、C 语言和汇编混合编程.....	28
1. C 语言内嵌汇编程序.....	28
2. 汇编调用 C 语言函数.....	28
3. C 语言调用汇编函数.....	29
4. 在汇编中使用 C 定义的全局变量.....	29
5. ATPCS（arm 程序调用规范）	30
DAY06—串口编程.....	31
一、串口原理.....	31
二、串口寄存器.....	31
1. ULCON:.....	31
4. UCON:.....	31
5. UBRDIV:.....	31
4. *UDIVSLOT:微调波特率.....	32
5. UTRSTAT: 状态寄存器.....	32
6. UTXH:发送寄存器.....	32
7. URXH:接受寄存器.....	32
三、串口编程.....	32
DAY07—NandFlash.....	35
一、NandFlash 结构.....	35
二、K9F2G08R0A 存储分析.....	35
三、NandFlash 控制器.....	35
1. NandFlash 控制器用.....	35
2. NFCONF: 配置寄存器.....	35
3. NFCONT: 控制寄存器.....	36
4. 写命令寄存器.....	36
5. 写地址控制器.....	36
四、读 NandFlash ID.....	37
DAY08—读 NandFlash、加载 Linux 内核.....	39
一、读 NandFlash.....	39
1. NFSTA.....	39
2. 步骤.....	39
二、引导 Linux 内核.....	40
DAY09—ARM 初始化.....	41
一、处理器初始化.....	41
1. 硬件初始化.....	41
2. 软件初始化.....	41

二、ARM 的启动方式.....	41
1. ARM 默认从 0 地址启动.....	41
三、时钟配置.....	42
四、软件初始化.....	42
DAY10—异常处理.....	43
一、异常分类.....	43
二、异常发生时的状态转移.....	43
1. 硬件.....	43
2. 软件.....	43
3. 异常处理步骤.....	43
三、搭建异常向量表.....	44
四、中断.....	44
1. 中断处理函数.....	44
2. 软中断.....	45
五、MMU (Memory Manager Unit)	45

第二部分 Linux 系统移植

DAY11—TFTP 原理、NFS 原理.....	46
一、嵌入式回顾.....	46
1. 嵌入式系统组成.....	46
2. 嵌入式硬件最小系统.....	46
3. ARM 处理器的选择.....	46
二、嵌入式 Linux 在开发板上运行过程.....	47
三、Linux 介绍.....	47
1. Linux 主要特性.....	47
2. Linux 系统架构.....	47
四、搭建 Linux 系统移植开发环境.....	48
1. 安装交叉编译工具.....	48
2. 串口工具软件.....	48
3. TFTP 服务器.....	48
4. NFS 服务器.....	48
五、烧写系统.....	49
1. 烧写 uboot.....	49
2. 烧写内核 zImage.....	49
3. 烧写跟文件系统.....	49
4. 设置 Nandflash 启动的环境变量.....	49
六、网络挂载.....	50
1. tftp 挂载 Linux 镜像 (zImage)	50
2. nfs 挂载根文件系统.....	50
3. nfs 挂载程序.....	50
DAY12—uboot 移植.....	51
一、bootloader 介绍.....	51
1. bootloader 两大功能.....	51
2. bootloader 工作.....	51

3. 什么是 uboot.....	51
二、uboot 目录.....	51
1. 跟硬件相关目录.....	51
2. 跟体系机构无关目录.....	52
3. 可能需要修改目录.....	52
三、编译 uboot.....	52
1. 配置.....	52
2. 编译.....	53
3. 移植需要做的事情.....	53
DAY13—uboot 代码.....	54
一、S5PV210 的地址空间.....	54
二、IROM 操作顺序.....	54
三、IROM (BL0) 启动顺序.....	55
四、u-boot 代码分析.....	55
DAY14—Linux 内核移植.....	57
一、Linux 内核.....	57
1. Linux 子系统.....	57
2. Linux 内存空间.....	57
3. Linux 内核目录.....	57
二、配置、编译 Linux 内核.....	57
1. 修改顶层 Makefile.....	57
2. 找一个参考配置.....	58
3. 配置内核.....	58
4. 编译内核.....	58
5. 编译内核、模块.....	58
三、Kconfig 和 Makefile 语法.....	58
四、内核 C 代码从哪开始.....	59
五、其他命令.....	59
六、内核配置.....	59
七、make zImage/uImage 的过程.....	61
DAY15—Linux 文件系统移植.....	62
一、文件系统.....	62
1. 什么是文件系统.....	62
2. 文件系统格式.....	62
3. 文件系统存放位置.....	62
4. 根文件系统.....	62
二、文件系统.....	62
1. 创建相关目录.....	62
2. 移植 busybox.....	63
3. 创建必要设备节点.....	64
4. 创建动态链接库.....	64
5. 修改 ubuntu 下的/etc/exports 文件.....	64
6. nfs 挂载根文件系统.....	64
7. 打包.....	65

三、拓展.....	65
1. 系统启动自动加载程序.....	65
2. 系统启动自动挂载文件.....	65
3. 自动创建设备节点.....	66
4. 手动挂载设备.....	66
5. 自动挂载设备.....	66

第三部分 Linux 驱动

DAY15—Linux 开发环境搭建.....	67
一、学习课程.....	67
二、GUN C 与标准 C 的区别.....	67
1. GUN C 支持 0 长度数组.....	67
2. case 语句.....	67
3. 宏定义.....	68
二、驱动相关的内核源码目录.....	68
三、编译模块.....	68
四、加载、卸载模块.....	69
五、驱动模板.....	69
DAY16—内核模块、内存管理.....	70
一、内核导出符号.....	70
二、printf.....	70
1. printk 优先级(按递减顺序).....	70
2. 查看/修改 printk 打印级别.....	70
三、模块参数.....	70
1. 模块参数声明.....	71
2. 模块数组参数声明.....	71
3. 模块参数支持的数据类型.....	71
4. 模块参数用户操作权限.....	71
5. 例子.....	72
四、内存管理.....	72
1. 地址.....	72
2. CPU.....	72
3. 段式管理模式.....	72
4. 页式管理模式.....	72
五、内核内存分配.....	73
1. kmalloc/kfree.....	73
2. * __get_free_pages/free_pages.....	73
3. vmalloc/vfree.....	73
六、GDB 调试.....	74
DAY17—内核链表、定时器.....	75
一、内核链表.....	75
1. 链表结构体.....	75
2. 链表操作函数.....	75
二、内核定时器.....	76

1. 常见名称.....	76
2. 内核定时器.....	76
3. 定时器函数.....	76
三、系统调用.....	77
1. 空间访问.....	77
2. 系统调用原理.....	77
3. 向内核中添加系统调用.....	77
DAY18—字符设备驱动.....	79
一、设备驱动分类.....	79
二、字符设备驱动框架.....	79
1. struct cdev.....	79
2. struct file_operations.....	80
3. struct innod.....	80
4. struct file.....	80
三、操作流程.....	81
1. 自动分配设备号.....	81
2. 初始化 cdev 结构体.....	81
3. 注册 cdev 设备进内核.....	81
4. 添加设备节点.....	81
5. 编写 file_operations 函数.....	82
6. module_exit 需要做的工作.....	82
DAY19—错误处理.....	83
1. 用户空间.....	83
2. 内核调试：.....	83
DAY20—点亮 LED 灯、竞态与并发.....	88
一、原理图.....	88
二、控制 LED 灯.....	88
1. 直接操作寄存器地址对应的虚拟地址.....	88
2. 调用内核提供的 GPIO 操作函数.....	88
三、竞态与并发.....	89
1. 产生原因.....	89
2. 内核中的解决方式.....	89
四、原子操作.....	89
1. 位原子操作.....	89
2. 整形原子操作.....	89
3. 原子操作和普通操作比较.....	89
五、自旋锁.....	90
1. 定义.....	90
2. 使用自旋锁.....	90
六、信号量.....	90
1. 定义.....	90
2. 使用信号量.....	91
七、信号量和自旋锁的区别.....	91
DAY21—等待队列、多路监听、地址映射.....	93

一、等待队列.....	93
1. 等待队列的数据结构.....	93
2. 等待队列的操作函数.....	93
3. 使用步骤.....	94
二、多路监听侦测.....	95
1. 用户空间操作.....	95
2. 内核空间操作.....	95
三、IO 与内存.....	96
1. X86 汇编中.....	96
2. ARM/powerpc/MIPS 汇编中.....	96
3. 虚拟地址使用过程.....	96
DAY22—中断、mmap.....	97
一、中断.....	97
1. 中断定义.....	97
2. 按键中断.....	97
3. tasklet 底半部中断.....	98
4. 工作队列.....	98
二、虚拟地址和物理地址转换.....	98
三、mmap.....	98
1. 用户空间.....	99
2. 内核空间.....	99
DAY24—字符设备编程架构.....	100
一、input 子系统.....	100
1. 什么是 input 子系统.....	100
2. 驱动实现步骤.....	100
3. 查看设备节点.....	101
4. 测试程序实现步骤.....	101
二、Linux 中的 input 子系统架构.....	101
DAY24—平台设备驱动.....	103
二、总线.....	103
1. 总线数据结构.....	103
2. 总线操作函数.....	103
三、设备.....	103
1. 设备数据结构.....	103
2. 设备操作函数.....	104
四、驱动.....	104
1. 驱动数据结构.....	104
2. 驱动操作函数.....	104
五、平台设备总线.....	105
1. 内核平台设备总线的注册.....	105
2. 平台设备.....	105
3. 设备驱动.....	106
4. match.....	106
5. probe 函数传递的值.....	106

DAY25—网卡驱动.....	108
一、Linux 网络子系统的框架结构.....	108
1. 网络协议线.....	108
2. Linux 网络设备驱动框架.....	108
3. 核心数据结构 struct net_device.....	109
4. net_device 操作函数.....	109
5. 网卡接收数据的相关函数.....	109
6. struct sk_buff,	110
二、DM9000AEP 网卡驱动分析及其移植.....	110
1. 从 datasheet 得到的信息.....	110
2. 从硬件电路图得到的信息.....	111
3. 内核中自带的网卡驱动程序.....	112
DAY26—I2C 设备驱动.....	114
一、I2C 介绍.....	114
二、通信协议.....	114
1. 信号.....	114
2. 数据传输.....	114
四、AT42C02.....	114
1. 地址.....	114
2. 写数据.....	115
3. 读数据.....	115
五、Linux I2C 框架.....	116
1. 向内核添加 client.....	116
2. 编写 I2C driver.....	119
DAY27—I2C 控制器原理和块设备.....	121
一、I2C 控制器的驱动.....	121
1. I2C 控制寄存器.....	121
2. 主设备发送数据.....	121
3. 主设备接收数据.....	121
4. S5PV210 的 I2C 控制器驱动程序.....	121
二、块设备.....	122
1. 块设备介绍.....	122
2. 内核中块设备驱动的框架机制.....	122
DAY28—LCD 驱动.....	124
一、LCD 概念.....	124
1. RGB.....	124
2. 电子枪.....	124
3. framebuffer.....	124
二、用户空间测试程序.....	124
三、驱动程序.....	124
1. linux 中 framebuffer 框架.....	124
2. 驱动程序，如何写？	125
四、LCD 驱动移植.....	126
1. LCD 屏幕管脚.....	126

2. s5pv210 的 LCD 控制器.....	127
3. LCD 控制器的可配置参数.....	127
4. 内核源码分析.....	127
5. 移植 LCD 驱动时应该注意哪些问题.....	128

第四部分 智能家居

DAY30—单总线驱动.....	129
一、DS18B20 芯片资料.....	129
1. 特性.....	129
二、DS18B20 操作流程.....	129
1. 第一步：初始化操作.....	129
2. 第二步：发送 ROM COMMANDS.....	129
3. 第三步：function COMMANDS.....	130
三、读写一个字节的时序.....	130
二、混杂设备.....	131
DAY31—ADC.....	132
一、210 ADC.....	132
1. diagram of A/D converter and Touch Screen Interface.....	132
2. A/D CONVERSION TIME.....	132
3. over view.....	132
4. ADC INTERFACE INPUT CLOCK DIAGRAM.....	133
5. 关注的寄存器.....	133
二、驱动.....	133
1. 注册一个混杂设备.....	133
2. 硬件初始化.....	133
DAY31—网络通信.....	135
一、按键驱动的方法.....	135
二、input 子系统.....	135
三、服务器端 UDP 编程.....	135
DAY32—QT 编程及其移植.....	136
一、设计图形界面.....	136
1. 创建带图形界面的项目.....	136
2. 注册槽函数.....	136
3. 信号和槽的连接函数.....	136
二、QT 移植.....	136
1. 注意事项.....	136
2. 交叉编译 tslib.....	137
3. 交叉编译 qt-embedded-linux.....	137
4. 增加环境变量.....	137
5. 编译、测试应用程序.....	138
6. 按键支持.....	138
7. 触摸屏支持.....	138
8. 添加 wenquanyi 字库.....	139
DAY33—串口编程和 I/O 多路监听.....	140

一、遵循 posix system 串口编程模型.....	140
1. 打开串口设备.....	140
2. 设置串口的工作模式和属性（struct termios）.....	140
3. 读写.....	140
二、I/O 多路监听.....	140
1. 读取串口和网口数据的方法.....	140
2. 测试程序.....	140
DAY33—项目总结(1).....	142
一、分区规划.....	142
二、uboot.....	142
1. 通过工具去烧写.....	142
2. 设置启动参数.....	142
三、文件系统.....	142
1. 创建目录.....	142
2. 移植 busybox.....	142
3. 移植 lib 库.....	143
4. 添加系统启动的配置文件.....	143
四、测试.....	145
1. NFS 进行测试！.....	145
2. 采用 ramdisk 或者 initramfs 进行测试！.....	145
DAY33—项目总结(2).....	146
一、常用文件系统.....	146
二、ramdisk 文件系统制作.....	146
1. 分区规划.....	146
2. 内核支持文件系统.....	147
3. 通过工具制作镜像.....	148
4. 通过工具烧写.....	148
5. 测试.....	149
三、ramdisk 文件系统制作.....	149
三、cramfs 文件系统制作.....	149
四、yaffs2 文件系统.....	150
五、rootfs.img 和 userdata.img 关联起来.....	150
附录一 遗忘点.....	151

DAY01—存储器、工作模式和寄存器

一. ARM 体系结构

1. 处理器型号

ARMv4:arm920T;

ARMv5:arm9ej;

ARMv6:ARM11;

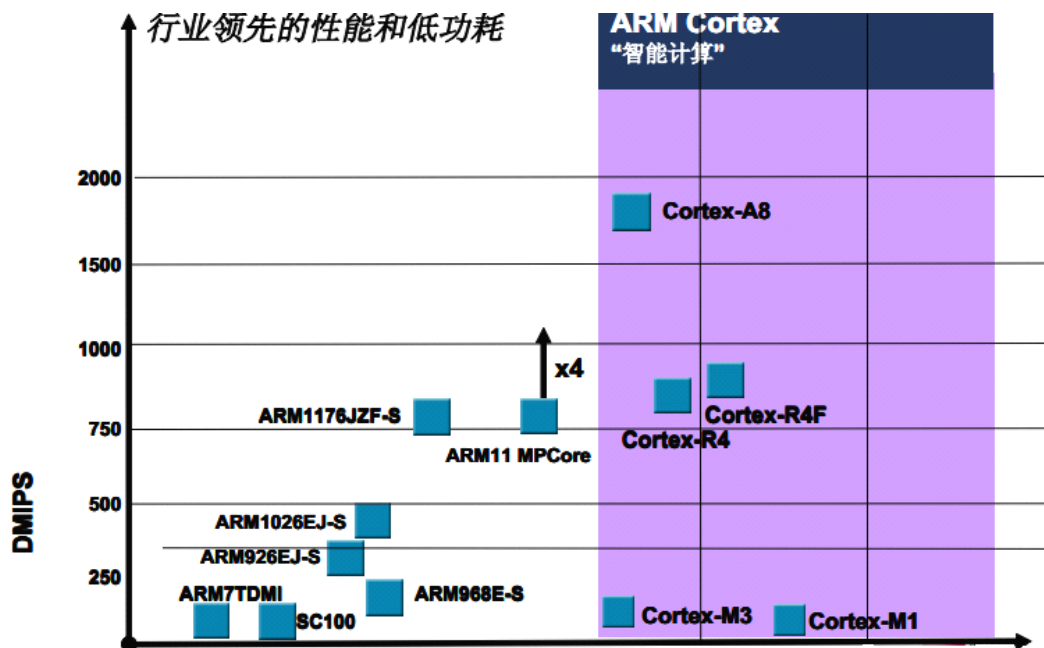
ARMv7:

cortex-A:1500~2000Hz, 解决高端问题

cortex-R:750 ~1000Hz, 解决实时性的问题

cortex-M: 低端产品

2. ARM 处理器性能



3. 基本流水线

基本 3 级流水线:取址、译码、执行

二、地址总线、数据总线、控制总线和 I/O

1. 地址总线 AB (AddressBus)

地址总线是专门用来传送地址的，由于地址只能从 CPU 传向外部存储器或 I/O 端口，所以地址总线总是单向三态的，这与数据总线不同。地址总线的位数决定了 CPU 可直接寻址的内存空间大小。比如 8 位微机的地址总线为 16 位，则其最大可寻址空间为 $2^{16}=64\text{KB}$ ，一个 32 位地址总线可以寻址到 4 GB 的地址。一般来说，若地址总线为 n 位，则可寻址空间为 2^n 字节。地址总线的宽度，随可寻址的内存元件大小而变，决定有多少的内存可以被存取。

2. 数据总线 DB (DataBus)

数据总线是双向三态形式的总线，即它既可以把 CPU 的数据传送到存储器或输入输出接口等其它部件，也可以将其它部件的数据传送到 CPU。数据总线的位数是微型计算机的一个重要指标，通常与微处理的字长相一致。例如 Intel8086 微处理器字长 16 位，其数据总线宽度也是 16 位。需要指出的是，数据的含义是广义的，它可以是真正的数据，也可以是指令代码或状态信息，有时甚至是一个控制信息，因此，在实际工作中，数据总线上传送的并不一定仅仅是真正意义上的数据。

3. 控制总线 CB (ControlBus)

控制总线主要用来传送控制信号和时序信号。控制信号中，有的是微处理器送往存储器和输入输出设备接口电路的，如读/写信号，片选信号、中断响应信号等；也有是其它部件反馈给 CPU 的，比如：中断申请信号、复位信号、总线请求信号、限备就绪信号等。因此，控制总线的传送方向由具体控制信号而定，一般是双向的，控制总线的位数要根据系统的实际控制需要而定。实际上控制总线的具体情况主要取决于 CPU。

cpu 在给内存读写数据时，需要指定数据总线和地址总线,数据总线确定数据的大小，地址总线确定数据的位置；

三、Flash、RAM 和 Flash 的区别

1. 随机存取存储器 RAM

RAM 也称读/写存储器，即 CPU 在运行过程中能随时进行数据的读出和写入，**相当于内存**。RAM 中存放的信息在关闭电源时会全部丢失，所以，RAM 是易失性存储器，只能用来存放暂时性的输入/输出数据、中间运算结果和用户程序，也常用它来与外存交换信息或用作堆栈。通常人们所说的微机内存容量就是指 RAM 存储器的容量。

按照 RAM 存储器存储信息电路原理的不同，RAM 可分为静态 RAM 和动态 RAM 两种。

①静态 RAM (Static RAM) 简称 SRAM，只要不断电，所存信息就不会丢失。因此，SRAM 工作速度快、稳定可靠，不需要外加刷新电路，使用方便，但是集成度不易做得很高，功耗

也较大。一般 SRAM 常用作微型系统的高速缓冲存储器（Cache）。

②动态 RAM（Dynamic RAM）简称 DRAM。为维持 DRAM 所存信息不变，需要定时地对 DRAM 进行刷新（Refresh），即对电容补充电荷。因此，集成度可以做得很高，成本低、功耗少，但它需外加刷新电路。DRAM 的工作速度比 SRAM 慢得多，一般微型机系统中的内存存储器多采用 DRAM。

DRAM 又主要分为 DDR（DDR SDRAM）和 SDR（SDRAM），只要记住 DDR 是 SDR 的加强版，性能比 SDR 好就行了

③对于嵌入式，在处理器内部一般集成了一块很小（大约 4K）的静态 RAM，外部就基本上在外接一块动态 RAM 就行了。

2. 只读存储器 ROM

ROM 是一种一旦写入信息之后，在程序运行中只能读出而不能写入的固定存储器，**相当于硬盘**。断电后，ROM 中存储的信息仍保留不变，所以，ROM 是非易失性存储器。因此，微型系统中常用 ROM 存放固定的程序和数据，如监控程序、操作系统中的 BIOS（基本输入/输出系统）、BASIC 解释程序或用户需要固化的程序。

3. 闪存 Flash

FLASH 存储器又称闪存，它结合 ROM 和 RAM 的长处，不仅具备电子可擦除可编程（EEPROM）的性能，还不会断电丢失数据同时可以快速读取数据（NVRAM 的优势），U 盘和 MP3 里用的就是这种存储器。在过去的 20 年里，嵌入式系统一直使用 ROM（EPROM）作为它们的存储设备，然而近年来 Flash 全面代替了 ROM（EPROM）在嵌入式系统中的地位，用作存储 Bootloader 以及操作系统或者程序代码或者直接当硬盘使用（U 盘）。

目前 Flash 主要有两种 NOR Flash 和 NAND Flash。NOR Flash 的读取和我们常见的 SDRAM 的读取是一样，用户可以直接运行装载在 NOR FLASH 里面的代码，这样可以减少 SRAM 的容量从而节约了成本。NAND Flash 没有采取内存的随机读取技术，它的读取是以一次读取一块的形式来进行的，通常是一次读取 512 个字节，采用这种技术的 Flash 比较廉价。用户不能直接运行 NAND Flash 上的代码，因此好多使用 NAND Flash 的开发板除了使用 NAND Flash 以外，还作上了一块小的 NOR Flash 来运行启动代码。

四、ARM 处理器的 7 种工作模式

管理模式（SVC）： 处理器复位或者软中断之后以进入该模式；

快速中断模式（FIQ）： 发生高级优先中断时加入该模式

中断模式（IRQ）： 发生低优先级中断时进入该模式

终止模式（Abort）： 用于处理非正常访问存储器(访问内存异常)

未定义模式（Undef）： 用于处理未定义指令

系统模式（System）： 专门运行操作系统自生

用户模式（User）： 多数应用程序和系统任务在该模式运行

除用户模式外，其它模式都是特权模式（privilege），特权模式可以访问所有通用寄存器；


除用户模式和系统模式外，其他 5 种都是异常模式；

五、ARM 状态下的寄存器

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

ARM状态下的程序状态寄存器

CPSR	CPSR SPSR_fiq	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_irq	CPSR SPSR_und
------	------------------	------------------	------------------	------------------	------------------

 = 分组寄存器

- R0~R15: 通用寄存器，参与程序运算，保存中间结果和最后结果
- CPSR 和 SPSR: 状态寄存器(PSR program status register)，用于保存运算结果的状态（如进位或借位、负数、溢出、零）
- R0~R12: 几乎所有模式公有(对于 FIQ 模式，R8~R12 为私有)
- R13(sp): 堆栈指针(stack pointer),把数据放入自己堆栈，防止其它模式访问，所有模式私有
- R14(lr): 连接寄存器(link register)，所有模式私有
- R15(pc):程序计数器(program counter)，所有模式共有，永远指向正在取址的指令对 R15 写地址，表示进行直接跳转
- CPSR(current program status register): 当前程序运行状态寄存器，所有模式共享
- SPSR(save program status register): 从非异常模式切换到异常模式时，把 CPSR 复制到 SPSR，这样就可以随意修改 CPSR，记录异常模式的状态；从异常模式恢复带非异常模式时，把 SPSR 拷贝到 CPSR，恢复非异常模式的状态记录

标志位	含义
N	当用两个补码表示的带符号数进行运算时，N=1 表示运算的结果为负数；N=0 表示运算的结果为正数或零；
Z	Z=1 表示运算的结果为零；Z=0 表示运算的结果为非零；
C	可以有 4 种方法设置 C 的值： — 加法运算（包括比较指令 CMN）：当运算结果产生了进位时（无符号数溢出），C=1，否则 C=0。 — 减法运算（包括比较指令 CMP）：当运算时产生了借位（无符号数溢出），C=0，否则 C=1。 — 对于包含移位操作的非加/减运算指令，C 为移出值的最后一位。 — 对于其他的非加/减运算指令，C 的值通常不改变。

DAY02—ARM 指令

1. 指令格式

助记符 + (条件) + (标志位 S) + 通用寄存器 + 通用寄存器 + (可选操作数)

2. 分支跳转指令

注意：在执行跳转指令的时候，会把跳转指令的下一条指令的地址复制到 R14

① B 指令：相对跳转，跳转过去之后不会返回，依次按顺序执行

语法：B + label

② BL 指令：相对跳转，跳转完成之后会返回，类似于函数调用。

在跳转程序末尾加上：mov pc R14，就可以返回跳转语句的下一条语句

3. 数据传送指令

① MOV 指令：MOV + (条件) + (标志位) + 目的寄存器 + 寄存器/立即数/寄存器位移

mov r0,#0 //立即数

moveq r1,r2 //寄存器

mov r0,r1,ls1 #2 //寄存器位移

注意：如果寄存器是 R15，表示直接跳转

② MOV 指令：取反再赋值

4. 加法指令

注意：相加/相减的 2 个参数中，第一个参数必须为寄存器，不能为常数

mov r1, #1, #2 //错误

① ADD 指令：ADD+(条件)+(标志位)+目的寄存器+寄存器 1+寄存器 2/立即数/寄存器位移

② ADC 指令：带进位的加法指令。2 个参数加之后，还会再加进位位

5. 减法指令

① SUB 指令：ADD+(条件)+(标志位)，目的寄存器+寄存器 1+寄存器 2/立即数/寄存器位移
目的寄存器=寄存器 1-寄存器 2/立即数/寄存器位移

② SBC 指令：带借位的减

目的寄存器=寄存器 1-寄存器 2/立即数/寄存器位移-借位位

③ RSB 指令：反减，交换减数和被减数的位置

目的寄存器=寄存器 2/立即数/寄存器位移-借位位-寄存器 1

5. 逻辑运算指令

- ① ADD 指令: ADD {条件} {S} 目的寄存器, **寄存器 1**, 寄存器 2/立即数/寄存器位移
寄存器 1 和寄存器 2/立即数/寄存器位移 **按位相与**, 在存放到目的寄存器
- ② ORR 指令: ORR {条件} {S} 目的寄存器, **寄存器 1**, 寄存器 2/立即数/寄存器位移
寄存器 1 和寄存器 2/立即数/寄存器位移 **按位相或**, 在存放到目的寄存器
- ③ EOR 指令: EOR {条件} {S} 目的寄存器, **寄存器 1**, 寄存器 2/立即数/寄存器位移
寄存器 1 和寄存器 2/立即数/寄存器位移 **按位异或**, 在存放到目的寄存器
- ④ BIC 指令: BIC {条件} {S} 目的寄存器, **寄存器 1**, 寄存器 2/立即数/寄存器位移
eg: bic r3, r2, #2 //把 r2 的**第 2 位**置 0, 保存到 r3 中

7 6 5 4 3 2 1 0



6. 比较指令

- ① CMP {条件}, **寄存器 1**, 寄存器 2/立即数/寄存器位移
注意: 比较指令没有目的寄存器, 结果直接修改标志位 S, 如果相等, Z=1, 否则 Z=0
- ① TEQ {条件}, **寄存器 1**, 寄存器 2/立即数/寄存器位移
测试 **寄存器 1** 和寄存器 2/立即数/寄存器位移是否相等, 或符号是否相同, 影响标志位

7. 第二操作数移位模式

<code><Rm>, LSL #<shift_imm></code>	//Rm 寄存器逻辑左移 imm 位, 不足补 0
<code><Rm>, LSR #<shift_imm></code>	//Rm 寄存器逻辑右移 imm 位, 不足补 0
<code><Rm>, ROR #<shift_imm></code>	//Rm 寄存器逻辑循环右移 imm 位, 最高位移动到最低位, 最低位移动到最高位
<code><Rm>, ASL #<shift_imm></code>	//算术左移
<code><Rm>, ASR #<shift_imm></code>	//算术右移

算数移位和逻辑移位的区别在于逻辑移位不考虑正负号, 算数移位要考虑正负号; 算数左移补 0, 算数右移看符号位 (最前面是 1 补 1, 否则补 0); 逻辑左移补 0, 右移也补 0; 真正的 *2、/2 是算数移位

8. 单寄存器寻址模式

注意: []内部的寄存器所存的值表示为地址

- (1) 前变址: [Rn, +/-shift_operand], 表示 Rn 的值加/减 shift_operand 这个常量

Eg: LDR R0, [R1, #4]

//R0 ← [R1+4]; 将寄存器 R1 的内容加上 4 形成操作数的有效地址, 从而取得操作数存入寄存器 R0 中。

```
Ldr r1, [r2, r3]
Ldr r1, [r2, #-4]
str r5, [r3, r5, asr, #5]
```

STR R0, [R1, #8] //将 R0 中的字数据写入以 **R1+8 为地址** 的存储器中。

(2) 后变址: [Rn], +/-shift_operand

```
Eg: Ldr r0, [r2], #4      //先将 r2 的值作为地址所对的寄存器的数据放到 r0 中,
                           //r2 的值再自加 4 (地址在加 4)
Ldr r2, [r3], r4          //将 r3 的值作为地址所对的寄存器的数据放到 r2 中,
                           //r3 的值=r3 的值+r4 的值
STR R0, [R1], #8          //将 R0 中的数据写入以 R1 的值为地址的存储器中, 并
                           //将新地址 R1+8 写入 R1。
```

(3) 回写前变址: [Rn, +/-shift_operand]!

```
Eg: Ldr r2, [r1, r3]!     //R2 ← [R1+R3], r1=r1+r3
str r2, [r1, r3]!         //r2 → [r1+r3], r1=r1+r3
LDRR0, [R1, #4]!          //R0 ← [R1+4], R1 ← R1+4。将寄存器 R1 的内容加上 4 形成操作数的有效地址,
                           //从而取得操作数存入寄存器 R0 中, 然后, R1 的内容自增 4 个字节。
```

9. 单寄存器加载指令

① LDR: 从内存中读取一个字节或字节数据存入寄存器中

语法: LDR (条件), 寄存器, 寻址模式

Eg: LDR R0, [R1] //R0 ← [R1], 以 R1 的值为地址的存储器中的数据传送到 R0 中

STR R0, [R1] // [R1] ← R0, R0 的值传送到以 R1 的值为地址的存储器中

注意: [] 中的数据是作为地址, 而不是作为值

② STR: 从寄存器中存储一个字节或字节数据存入内存中

STR (条件), 寄存器, 寻址模式

Eg: **mov r2 0**

Str r2, [r1], #4 //循环执行这两条语句, 可以清空内存

10. 多寄存器加载指令

① LDM/STM

STM (条件) (多寄存器寻址方式) Rb (!), {寄存器列表} {^}

LDM (条件) (多寄存器寻址方式) Rb (!), {寄存器列表} {^}

Rb(!): !表示是否跟新基地址

{寄存器列表}: 可以写 r2, r3 也可以写成 r4-r9

{^}: 用于模式切换, 如异常处理

② *从内存角度来看

I(Increment)表示向高地址方向读数据, D(Decrement)表示向低地址方向读数据

B(Before)表示先跟新地址, A(After)表示后跟新地址

LDMIA/STMIA Increment After (先操作, 后增加)

LDMIB/STMIB Increment Before (先增加, 后操作)

LDMDA/ STMDA	Decrement After	(先操作, 后递减)
LDMDB/ STMDB	Decrement Before	(先递减, 后操作)

③ 从堆栈角度去看

FD (Full Descending) //满减堆栈
ED (Empty Descending) //空减堆栈
FA (Full Ascending) //满加堆栈
EA (Empty Ascending) //空加堆栈

//内存方法基本不用, 堆栈方式更常用, 其中的 FD 方式最常用

Eg: STMFD R13!, {R0, R4-R12, LR}
//将寄存器列表中的寄存器 (R0, R4 到 R12, LR) 存入堆栈。
LDMFD R13!, {R0, R4-R12, PC}
//将堆栈内容恢复到寄存器 (R0, R4 到 R12, LR)。

11. 乘法指令

MUL 32 位乘法指令
MLA 32 位乘加指令
MULL 64 位有符号数乘法指令
SMLAL 64 位有符号数乘加指令
UMULL 64 位无符号数乘法指令
UMLAL 64 位无符号数乘加指令

Eg: MLA R0, R1, R2, R3 //R0 = R1 × R2 + R3
 SMULL R0, R1, R2, R3 //R0 = (R2 × R3) 的低 32 位
 R1 = (R2 × R3) 的高 32 位
 SMLAL R0, R1, R2, R3 //R0 = (R2 × R3) 的低 32 位 + R0
 //R1 = (R2 × R3) 的高 32 位 + R1

12. 交换指令

指令格式: SWP{cond} <Rd>, <Rm>, [<Rn>]; 其中 [<Rn>] 为地址

13. *字节交换指令

SWP 指令将一个内存字单元的内容读取到一个寄存器, 同时将另一个寄存器的内容写入该内存单元。当上述两个寄存器为同一个寄存器时, 指令交换该寄存器和内存单元的内容。

14. 状态寄存器操作指令

MSR(move status register): 将普通寄存器的值读到状态寄存器里
MRS(move register status): 将状态寄存器的值读到普通寄存器里
mrs rd/immediate_8, cpsr/spsr_<fields>
msr cpsr/spsr_<fields>, rd/immediate_8

<fields>可以为以下字母（必须小写）的一个或者组合

c: 控制域屏蔽字节(psr[7..0])

x: 扩展域屏蔽字节(psr[15..8])

s: 状态域屏蔽字节(psr[23..16])

f: 标志域屏蔽字节(psr[31..24])

immediate_8: 8 位图立即数

eg: MSR CPSR_c, #0xd2 //传送 0xd2 到 SPSR，但仅仅修改 CPSR 中的控制位域

15. 常用条件

条件助记符	标 志	含 义
EQ	Z=1	相等
NE	Z=0	不相等
CS/HS	C=1	无符号数大于或等于
CC/LO	C=0	无符号数小于
MI	N=1	负数
PL	N=0	正数或零
VS	V=1	溢出
VC	V=0	没有溢出
HI	C=1,Z=0	无符号数大于
LS	C=0,Z=1	无符号数小于或等于

DAY03—ARM 汇编程序设计

查看 arm-linux-gcc 的版本: arm-linux-gcc-**v**:

新增环境变量: export PATH=\$PATH:新增环境变量;

然后输入 source /etc/profile 永久修改环境变量

① arm-linux-gcc-**c***.s: 编译成.o 目标文件

② arm-linux-gcc -nostdlib -nostartfiles -e start test.o -o test

这个 start 是 test.S 中的入口 labe

③ arm-linux-objcopy -O binary test test.bin: 把 test 这个可执行文件的头尾去掉

DAY04—烧写程序、点亮 LED 灯

一、烧写初始化代码

1. 通过 usb 下载 usb.bin 到 0xd0020010 //因为 irom 就往这跳转
2. 通过 usb 下载 uboot.bin 到 0x23e00000

二、uboot 设置语句

1. print:列出 uboot 支持的环境变量
ipaddr: 开发板的 ip 地址
serverip: 代表服务器的 ip 地址
1. setenv:设置环境变量
setenv 环境变量名 环境变量值
setenv ipaddr 192.168.1.3
setenv serverip 192.168.1.2
2. 保存环境变量:saveenv
3. 删除环境变量: setenv 环境变了名 (后面什么都不加)

三、从 tftp 服务器下载文件

1. 下载程序:
tftp 内存中的地址 文件名
eg: tftp 0x20008000 led.bin: 0x20008000 不是固定的,有一定的范围
2. 执行程序
go 内存地址
go 0x20008000

四、烧写 uboot 到 NandFlash

```
nand erase 0x0 0x0100000
nand write 0x20008000 0x0 0x0100000
```

五、GPIO 口设置 (GPC1_3 为例)

1. GPC1CON:W/R,Address=0xE020_0080
GPC1CON[3] [15:12]
0000 Input

- | | |
|-------------|-------------|
| 0001 | Output |
| 0010 | PCM_2_SIN |
| 0011 | Reserved |
| 0100 | I2S_2_SDI |
| 0101 ~ 1110 | Reserved |
| 1111 | GPC1_INT[3] |
- GPC1DAT[4:0] [4:0]
When the port is configured as input port, the corresponding bit is the pin state. When the port is configured as output port, the pin state is the same as the corresponding bit. When the port is configured as functional pin, the undefined value will be read.
 - Port Group GPC1 Control Register (GPC1PUDPDN, R/W, Address = 0xE020_0094)可以不用

GPC1PUDPDN	Bit	Description
		00 = Pull-up/ down disabled
		01 = Pull-down enabled
GPC1[n]	[2n+1:2n]	10 = Pull-up enabled
	n=0~4	11 = Reserved

六、点亮 led 灯（gpc1_3 为低电平 led 亮）

- 程序


```

start:
@配置寄存器
@配置 GPC1CON,(0XE0200080),[15:12]=0B0001
ldr r0,=0xe0200080      //伪指令，相当于 mov r0 0xe0200084，但是由于立即数太
                        //大，所以采用伪指令方式

ldr r1,[r0]              //r0 的值就是 0xe0200084
bic r1,r1,#0xf<<12      //或： 12~15 位清零
orr r1,r1,#1<<12        //第 12 为置为 1
str r1,[r0]

@配置 GPC1PUD,(0XE0200088),[7:6]=0B00
ldr r0,=0xe0200088
ldr r1,[r0]
bic r1,r1,#1<<6
bic r1,r1,#1<<7
str r1,[r0]

@循环亮灭
@亮灯 GPC1DAT(0XE0200084),[3]=0B1
loop:
    ldr r0,=0xe0200084
    mov r1,#0x1<<3
    str r1,[r0]
```

@延迟

```
mov r0,#0xf00000
```

delay1:

```
subs r0,r0,#1
```

```
bne delay1
```

@灭灯 GPC1DAT(0xE0200084),[3]=0B0

```
ldr r0,=0xe0200084
```

```
mov r1,#0x0<<3
```

```
str r1,[r0]
```

@延迟

```
mov r0,#0xf00000
```

delay2:

```
subs r0,r0,#1
```

```
bne delay2
```

```
b loop
```

2.操作

```
arm-linux-gcc -c led.S
```

```
arm-linux-gcc -Ttext 0x20008000 -nostdlib -nostartfiles -e start led.o -o led
```

```
arm-linux-objcopy -O binary led led.bin
```


DAY05—伪指令、混合编程

一、伪指令

1. 符号声明伪操作

- (1) `.equ` 符号 , 值 //把值定义为某个符号,相当于 `define`
eg: `.equ GPC1CON,0X20008000` //把 `GPC1CON` 定义为 `0XFFFFFFF`
- (2) `.global` 或者 `.globl` 标记符 //声明全局常量
eg: `.global start`
- (3) `.extern` 标记符 //声明外部函数, 可要可不要

2. 数据定义伪操作

- (1). 字节定义`.byte`
语法格式: `.byte expr {, expr }...` //expr:数字表达式或程序中的标号。
- (2). 半字定义`.hword` 或`.short`
语法格式:
`hword expr {, expr }...`
`short expr {, expr }...`
- (3). 字定义`.word` 或`.int` 或`.long`
语法格式:
`.word expr {, expr }...`
`.int expr {, expr }...`
`.long expr {, expr }...`
- (4). 字符串定义`.ascii` 和`.asciz` 或`.string`
语法格式:
`.ascii expr {, expr }...`
`.asciz expr {, expr }...`
`.string expr {, expr }...`
- (5). 单精度浮点数定义`.float` 或`.single`
语法格式:
`.float expr {, expr }...`
`.single expr {, expr }...`
- (6). 双精度浮点数定义`.double`
语法格式: `.double expr {, expr }...`
- (7). *重复内存单元定义`.fill`
语法格式:
`.fill repeat {, size}{, value}`

repeat :重复填充的次数;

size :每次所填充的字节数;

value :所填充的数据。

eg: .fill 2048,1,0xff //在接下来 2048 个*1 字节空间, 并且填充的数据为 1, 相当于
// 2048*8 位

hexdump + 二进制文件: 查看存二进制文件

(8). *声明数据缓冲池

语法格式: .ltorg

在安全的地方加数据缓冲池:例如跳转语句的后面

c 语言中定义一个 int 变量:int a;

汇编中定义一个 int 变量: a:

.word //a 是一个 label

c 语言中定义一个 int 变量并赋值: int b=5;

汇编中定义一个 int 变量并赋值: b:

.word 5 //a 是一个 label

c 语言中定义一个 char 变量:char *c="hello";

汇编中定义一个 char 变量: c:

.ascii "hello"

3. 汇编与反汇编代码控制伪操作

(1). GNU ARM 预定义的段

.text :代码段

.data :数据段, 保存有初值的全局变量

.bss :堆栈段, 保存未初始化的全局变量

(2). 段定义伪操作,

语法格式如下: section <section_name> {," <flags>" }

section_name: 段名称, 可以是自己定义的名称, 也可以是预定义的段名.text、.data、.bss
中的一个

flags: ELF 文件格式的标志, <Flag> 可以是: -a 可加载段, -w 可写段, -x 可执行段

(3). *对齐方式设置伪操作.align 或.balign

语法格式:

.align {alignment} {, fill}

.balign {alignment} {, fill}

alignment: 是一个数值表达式, 用于指定对齐方式, 其取值在 0~15 范围内;

fill: 用来指定进行填充的数据。

eg: .short 5//short 占 2 个字节

.align 2 //4 字节对其

mov r0,r4

4. 预定义控制伪操作

(1). 条件编译伪操作.if

语法格式:

```
.if    logical_expression
程序代码段 A
{.else
程序代码段 B
}
.endif
```

(2). 宏定义

不带参数

```
.macro  ADD1
add r0, r1, r2
.endm
```

带参数

```
定义:  .macro  ADD2  R1, R2
        add r0, \r1, \r2
        .    endm
```

应用: ADD2 r3, r4

(3). 文件包含伪操作.include

.include 伪操作用于将一个源文件包含到当前的源文件中, 所包含的文件在.include 指令的位置处进行汇编处理。

语法格式: .include "file_name"

二、汇编、连接

1. 准备 3 个.S 文件, 用 arm-linux-gcc -c 编译 3 个.S 文件, 生成 3 个.o 文件

2. 编写 a.lds 脚本连接文件 (连接多个.o 文件)

```
ENTRY(start)
```

```
SECTIONS{
```

```
    . = 0x20008000 //代码段运行地址
```

```
    .text : {
```

```
        a.o(.text) //先执行 a.o 文件中的代码段, 然后是 b.o 文件中的代码段, 最后是
                //c.o 文件中的代码段
```

```
        b.o(.text)
```

```
        c.o(.text)
```

```
    }
```

```
    .data : {
```

```
        *.o(.data)
```

```
    }
```

```
    .bss : {
```

```
        *(.bss)
```

```

    }
}
3. arm-linux-gcc -nostdlib -nostartfiles -T a.lds a.o b.o c.o -o a

```

三、C 语言和汇编混合编程

1. C 语言内嵌汇编程序

```

eg: int add(int a,int b)
{
    int c;
    int d=0xffff;
    asm volatile(
        "mov r0,%2\n"
        "add %0,%1,%2\n"
        : "=r"(c)           //第一句必须为输出值，与这个函数
                           //包括 main 函数）是否有返回值无关
        : "r"(a),"r"(b),"r"(d) //后面的才是输入值
        : "r4"
    );
    return c;
}

```

2. 汇编调用 C 语言函数

.S 汇编文件

```

.extern x      //c 文件中的全局变量
.extern add0   //c 语言中的全局函数
start:

```

```

    ldr r3,=x //使用全局变量
    @int add(int a,int b);
    mov r0,#1
    mov r1,#2
    bl add0

```

.c 文件

```

int x;           //定义全局变量
int add0(int a,int b)
{
    return a+b;
}

```

3. C 语言调用汇编函数

S 汇编文件

```
@int addby2(int a,int b);
addby2:
    mov r1,r1,lsl #1
    add r0,r0,r1
    mov pc,lr
```

.c 文件

```
extern int addby2(int a,int b);
int main()
{
    int a=5;
    int b=5;
    int c=addby2(2,2);
}
```

4. 在汇编中使用 C 定义的全局变量

//C 语言文件*.c

```
#include <stdio.h>
int gVar=12;
extern asmDouble(void);
int main(){
    printf("original value of gVar is: %d", gVar_1);
    asmDouble();
    printf(" modified value of gVar is: %d", gVar_1);
    return 0;
}
```

//汇编语言文件*.S

```
AREA asmfile, CODE, READONLY EXPORT asmDouble
IMPORT gVar
asmDouble
    ldr r0, =gVar
    ldr r1, [r0]
    mov r2, #2
    mul r3, r1, r2
    str r3, [r0]
    mov pc, lr
END
```

在此例中，汇编文件与 C 文件之间相互传递了全局变量 `gVar` 和函数 `asmDouble`，留意声明的关键字 `extern` 和 `IMPORT`

5. ATPCS (arm 程序调用规范)

(1). 规定寄存器使用

(2). 规定函数传参方式

.开始四个字大小的参数直接使用寄存器的 R0-R3 来传递(快速且高效的),如果需要更多的参数,将使用堆栈。(需要额外的指令和慢速的存储器操作)

所以通常限制参数的个数,使它为 4 或更少;如果不可避免,把常用的参数放在前 4 个

r0~r3 可记为 a0~a3, r4~r11 可记为 v1~v8,但最好不要这么写;

r12 为连接寄存器,用于保存堆栈指针 sp,当子程序返回时使用该寄存器出栈

r14 为链接寄存器,用于保存子程序的返回地址

参数自动从 R0 开始排,返回值为 32 则自动保存在 R0 里面,返回值为 64 位则自动保存在 r0 和 r1 里面

(3).规定栈使用方法

ARM 使用满减栈的方法

DAY06—串口编程

一、串口原理

- 1.在 TXD 和 RXD 上逻辑 1 为-3~-15V，逻辑 0 表示+3~+15V
- 2.波特率表示每秒传输的二进制位，在环境比较差或者传输距离比较远的时候，最好降低一下波特率
- 3.表示：8n1，115200：8 位，不校验，1 位停止位，波特率为 115200
n：不校验，e (even)：偶校验；o (odd)：奇校验
- 4.rs232 电平和 ttl 电平转换：开发板为 rs232 电平，电脑一般为 ttl 电平

二、串口寄存器

1. ULCON:

[31:7]:保留
[6]:是否红外模式
[5:3]:校验位
[2]:停止位
[1:0]:位宽
一般配置为 0x3

4. UCON:

[31:20]:保留
[20:11]:跟 DMA 有关
[10]:串口模块的时钟输入源，有 2 个时钟源信号，默认为 PCLK
[9:6]:跟中断有关，先设置为 0
[5]:回路模式，自己发给自己，用于测试程序
[4]:发送端发送暂停或者终止信号
[3:2]:传输模式，一般设置为 01（查询中断模式）
[1:0]:接受模式，一般设置为 01（查询中断模式）

5. UBRDIV:

[31:16]:保留
[15:0]:DIV_VAL
$$\text{DIV_VAL} = (\text{PCLK} / (\text{bps} \times 16)) - 1 \quad \text{or} \quad \text{DIV_VAL} = (\text{SCLK_UART} / (\text{bps} \times 16)) - 1$$

eg: $\text{DIV_VAL} = (40000000 / (115200 \times 16)) - 1 = 21.7 - 1 = 20.7$

4. *UDIVSLOT:微调波特率

5. UTRSTAT: 状态寄存器

[1]:0,发送 buffer 有数据; 1, 接受 buffer 为空

[0]:0,接收 buffer 为空; 1, 接受 buffer 有数据

6. UTXH:发送寄存器

7. URXH:接受寄存器

三、串口编程

@配置端口为串口模式

@关闭端口上下拉电阻

```
#define GPA0CON (*(volatile unsigned long*)0xe0200000)
#define GPA0PUD (*(volatile unsigned long*)0xe0200008)
#define ULCON0 (*(volatile unsigned long*)0xe2900000)
#define UCON0 (*(volatile unsigned long*)0xe2900004)
#define UFCON0 (*(volatile unsigned long*)0xe2900008)
#define UMCON0 (*(volatile unsigned long*)0xe290000C)
#define UBRDIV0 (*(volatile unsigned long*)0xe2900028)
#define UTRSTAT0 (*(volatile unsigned long*)0xe2900010)
#define URXH0 (*(volatile unsigned char*)0xe2900024)
#define UTXH0 (*(volatile unsigned char*)0xe2900020)
```

```
char c[2048];
```

```
void uart_config()
```

```
{
    //1.GPA0_0,GPA0_1
    //1.1:GPA0CIN:[7:0]
    GPA0CON=GPA0CON|0x22;
    //1.2:GPA0PUD:[3:0]
    GPA0PUD=GPA0PUD&0x00;
    //2:UART
    //2.1:ULCON
    ULCON0=0x03;
    //2.2:UCON
```



```

    UCON0=0x05;
    //2.3:UFCON,UMCON=0
    UFCON0=0x00;
    UMCON0=0x00;
    //2.4:UDIVBRD
    UBRDIV0=(66500000)/(115200*16)-1;
    //2.5:
}
//-----
void put_c(unsigned char c)
{
    //循环读取 UTRSTAT[1],为 1 退出循环
    while(!((UTRSTAT0)&0x02));
    //2:UTXH0
    UTXH0=c;
}
//-----
void put_s(char *s)
{
    int i=0;
    while(s[i]){
        put_c(s[i++]);
    }
    put_c('\n');
    put_c('\r');
}
//-----
char get_c(void){
    while(!((UTRSTAT0)&0x01));

    return URXH0;
}
//-----
char* get_s(void){
    char ch;
    int i=0;
    while((ch=get_c())!='\r')
        c[i++]=ch;
    c[i]='\0';
    return c;
}
//-----
void uart_run()

```

```
{  
    uart_config();  
    int i=10;  
    while(i)  
    {  
        put_c('a');  
        i--;  
    }  
    put_c('\n');  
    put_c('\r');  
    put_s("hello tarena");  
    char *text=get_s();  
    put_s(text);  
}
```

DAY07—NandFlash

一、NandFlash 结构

- I/O_{0~7}: 数据地址总线
- CLE: 控制锁存使能, 不用自己操作
- ALE: 地址锁存使能, 不用自己操作
- CE: 片选
- RE/WE: 读写使能, 不用自己操作

二、K9F2G08R0A 存储分析

- 1. NandFlash 以页为单位操作
- 2. 通过发送命令的方式对 NandFlash 操作
- 3. 控制命令和地址公用一条总线
- 4. NandFlash 上半页为实际存储数据大小(2048Byte), 下半页为校验数据的地方(64Byte), 所以一页的大小为 (2k+64) Byte
- 5. 每一页都有自己的偏移量
- 6. NandFlash 有行地址和列地址 (如果对整页操作, 列地址就是 0)

三、NandFlash 控制器

1. NandFlash 控制器用

来处理一些片选型号、使能信号、指令类型等, CPU 只是通过对控制器 NandFlash 控制器的寄存器发送命令来对 NandFlash 操作

2. NCONF: 配置寄存器

[1]AddrCycle: 地址周期, NandFlash 提供

	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7	Column Address
2nd Cycle	A8	A9	A10	A11	*L	*L	*L	*L	Column Address
3rd Cycle	A12	A13	A14	A15	A16	A17	A18	A19	Row Address
4th Cycle	A20	A21	A22	A23	A24	A25	A26	A27	Row Address
5th Cycle	A28	*L	*L	*L	*L	*L	*L	*L	Row Address

[2]PageSize: 每页大小

[3]MLCFlash: 设计工艺

[7:4]TWRPH1: 写使能保持时间 NandFlash 提供

[11:8]TWRPH0: 写使能脉冲宽度 NandFlash 提供

[15: 12]TACLAS: 地址/命令锁存的建立时间, NandFlash 提供; 只要比 NandFlash 提供的时间大就行, 为了方便, 这里我们选最大值

AC Timing Characteristics for Command / Address / Data Input

Parameter	Symbol	Min		Max		Unit
		1.8V	3.3V	1.8V	3.3V	
CLE Setup Time	tCLS ⁽¹⁾	21	12	-	-	ns
CLE Hold Time	tCLH	5	5	-	-	ns
$\overline{\text{CE}}$ Setup Time	tCS ⁽¹⁾	25	20	-	-	ns
$\overline{\text{CE}}$ Hold Time	tCH	5	5	-	-	ns
$\overline{\text{WE}}$ Pulse Width	tWP	21	12	-	-	ns
ALE Setup Time	tALS ⁽¹⁾	21	12	-	-	ns
ALE Hold Time	tALH	5	5	-	-	ns
Data Setup Time	tDS ⁽¹⁾	20	12	-	-	ns
Data Hold Time	tdH	5	5	-	-	ns
Write Cycle Time	tWC	42	25	-	-	ns
$\overline{\text{WE}}$ High Hold Time	tWH	15	10	-	-	ns
Address to Data Loading Time	tADL ⁽²⁾	100	100	-	-	ns

3. NFCONT: 控制寄存器

[22]、[23]、[2]、[1]: Reg_nCE1~4, 片选型号

[18]MLCEccDirection: 校验位

[0]MODE: 总开关, 用的时候置一

4. 写命令寄存器

NFCMMD	Bit	Description	Initial State
Reserved	[31:8]	Reserved	0x000000
REG_CMMD	[7:0]	NAND Flash memory command value	0x00

5. 写地址控制器

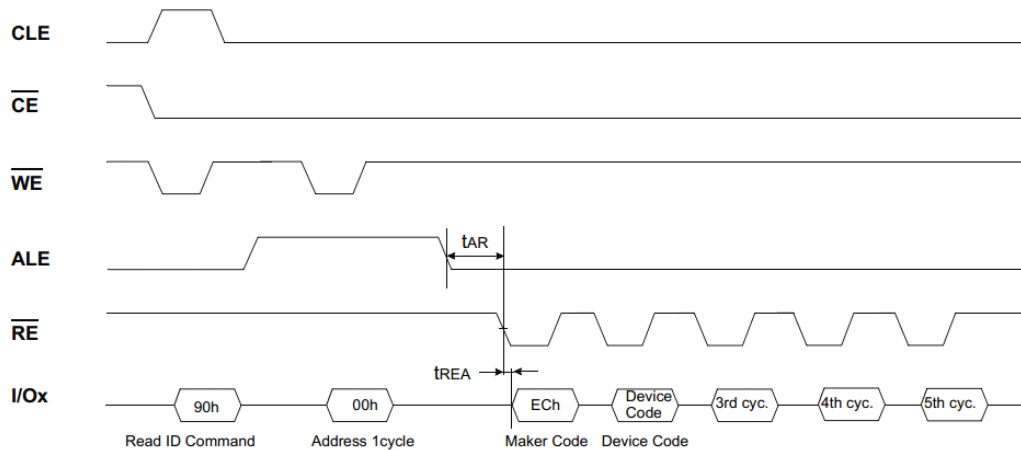
NFADDR	Bit	Description	Initial State
Reserved	[31:8]	Reserved	0x000000
REG_ADDR	[7:0]	NAND Flash memory address value	0x00

6. 读数据寄存器

NFDATA	Bit	Description	Initial State
NFDATA	[31:0]	NAND Flash read/ program data value for I/O Note: For more information, refer to 4.3.1 Data Register Configuration in page 4-4 .	0x00000000

四、读 NandFlash ID

Read ID Operation



char* itoa2(char *buf,unsigned int a) //整数到字符串

```
{
    int i;
    unsigned int c;
    buf[0]='0';
    buf[1]='x';
    i=9;
    while(a)
    {
        c=a%16;
        if(c>=10)
            buf[i]=c-10+'A';
        else
            buf[i]=c+'0';

        a=a/16;
        i--;
    }
    while(i>=2)
    {
        buf[i--]='0';
    }
    buf[10]=0;
```

```
}
//-----
void nand_init()
```

```

{
    //NFCNF
    NFCNF=0xff2;
    //NFCNT
    NFCNT=0xf1;
}
//-----
void nand_id()          //打印 ID
{
    //NFCMMD<-0X90
    NFCMMD=0x90;
    //NFADDR<-0X0
    NFADDR=0;
    //int id1<-NFDATA
    int id1=NFDATA;
    //int id2<-NFDATA
    int id2=NFDATA;
    //itoa
    //put_s();
    itoa2(value,id1);
    put_s(value);
    itoa2(value,id2);
    put_s(value);
}

```

DAY08—读 NandFlash、加载 Linux 内核

一、读 NandFlash

1. NFSTA

[4]nB_TransDetect: 0 = RnB transition is not detected, 表示数据补存在
 1=RnB transition is detected, 表示数据存在, 可以进行读数据
 需要手动写 1, 清楚这个位

2. 步骤

step1: clear_stat 清楚 nB_TransDetect[4]状态位
step2: NFCMMD=0;
step3: NFADDR=0;
 NFADDR=0;
 page_no (第几页) = 页地址/页大小 (对于 K9F2G08, 每页大小为 2K)
 NFADDR=(page_no&0xff);
 NFADDR=(page_no>>8&0xff);
 NFADDR=(page_no>>16&0xff);
step4: NFCMMD=0X30
step5: 判断(NFSTAT&(1<<4))是否是 1
step6: 循环读取
void nand_read_page(char *mem,int naddr)
{

```
    int i=0;
    int pag_no=naddr/2048;
    //NFCMMD=0X0
    NFCMMD=0X0;
    //NFADDR=0X0
    NFADDR=0X0;
    //NFADDR=0X0
    NFADDR=0X0;
    //NFADDR<-low 8bit
    NFADDR=(pag_no)&0xff;
    //NFADDR<-mid 8bit
    NFADDR=(pag_no>>8)&0xff;
    //NFADDR<-high 8bit
    NFADDR=(pag_no>>16)&0xff;
```

```

//NFCMMD=0X30
NFCMMD=0X30;
//WAIT RnB SIG
while(!(NFSTAT&(1<<4)))
{}
//循环 READ NFDATA,保存到 mem 基地址
int *ptr=(int*)mem;
for(i=0;i<512;i++)
{
    *(ptr+i)=NFDATA;
}
for(i=0;i<256;i++)
{
    itoa2(value,ptr[i]);
    put_s(value);
    put_s("\n");
}
}

```

二、引导 **Linux** 内核

- Tftp 下载的东西是放在内存里面的，没有放在 NandFlash 里面，断电消失
- tftp 下载的 uboot.bin 是放在内存里面的，要通过 nand write 写到 NandFlash，下载到 NandFlash 的 0x0 处，当选择从 NandFlash 启动的时候，就自动加载 NandFlash
- 把 Linux 镜像文件 image 通过 tftp 下载到 0x20008000 处，再通过 nand write 写到随便那个地方（以 0x60000 为例），把 Linux 引导程序通过 tftp 下到 0x20004000 处然后执行这个程序，NandFlash 中的 image 文件从 0x60000 读到 0x20008000 处执行。

DAY09—ARM 初始化

一、处理器初始化

1. 硬件初始化

时钟配置、(内存初始化)、(关闭中断控制器)、(初始化看门狗定时器)、(串口)

- 多数处理器在启动的时候，由于 CPU 还没有准备好，需要关闭中断控制器，不处理任何中断
- 这里的定时器主要是指 watchdog (看门狗)，看门狗能够保证程序不会死锁，需要隔一段时间就初始化看门狗，否则一到设定的时间，看门狗就会硬件初始化 CPU

2. 软件初始化

管理模式下的栈初始化、BSS 段的清空、(切换到管理模式)、(处理 CP15 寄存器)、

- 栈用来保存已经初始化的变量，如果不初始化栈，则只能用汇编语言编程，而且不能调用函数，如何初始化栈：给 SP 一个高内存地址 (因为 ARM 堆栈模式为满减栈)，栈的大小需要人为划分
- BSS 段用来保存未初始化、或者初始化为 0 的全局变量 (BSS 段在文件大小中不体现，只有在程序运行时，在内存中才会体现)

二、ARM 的启动方式

1. ARM 默认从 0 地址启动

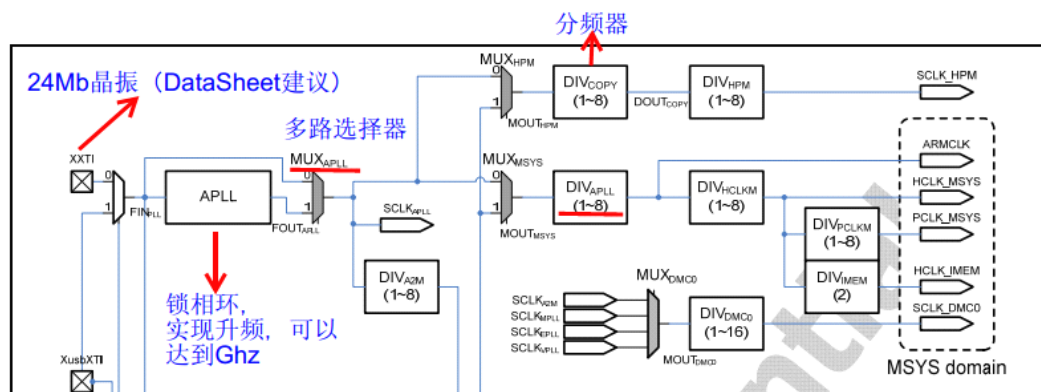
◇对于一般 ARM 芯片：

- ARM 有 4G 的地址空间，ARM 是从 0 地址空间启动，而板子的外接动态内存一般不会接在 0 地址处，对一般 ARM，0 地址没有固定，可以进行地址映射。
- 假设 NorFlash 的地址为 0x40000000，那么可以把 0 地址映射到 0x40000000，访问 0 地址和 0x40000000 地址都是访问同一内存，这样就可以通过 NorFlash 进行启动
- 对一般 ARM，还可以把 0 地址映射到芯片内部集成的 SRAM (内部静态内存)，把 NandFlash 的前 4k 程序搬运 SRAM，运行 SRAM 的程序
- 对一般 ARM，还支持 IROM (内置 ROM) 启动，IROM 固化的程序会初始化串口、USB 接口、SD 卡、NandFlash 等

◇对于 S5PV210，支持 NorFlash 启动和 IROM 启动

- 外接动态内存大概在 0x2000000，内部静态内存大概在 0xd000000。

三、时钟配置



1. 关闭多路选择器
CLK_SRC0 (0xE010_0200) = 0
2. 关闭锁相环
APLL_CON0=0;
MPLL_CON=0;
3. 配置系统时钟分频器 CLK_DIV0
4. 配置锁相环
5. 打开多路选择器

四、软件初始化

栈初始化: MOV SP , 0XD0036000

BSS 段清除: ldr r0 , bss_start
ldr r1 , bss_end
mov r2 , #0x00000000
loop:
str r2 , [r0]
add r0 , r0 , #4
cmp r0 , r1
ble loop

电源管理芯片: 把 GPJ2_5 设置为输出模式, 并且输出为高电平, GPJ2_5 连接电源管理芯片
内存初始化: 必须放在时钟初始化之后

DAY10—异常处理

一、异常分类

复位异常：reset

未定义指令异常（译码时发生）

软中断：swi，出发系统调用时产生，它使得在 User 模式下运行的程序能够请求在 Supervisor 模式运行的特权

预取址中止：在处理器试图执行一个未取址时发生，取址出错（内存出错）

数据中止：发生在数据运行的时候（ldr、str、ldm、stm 这 4 个指令）

IRQ：在处理器外部中断申请管教有效（低电平）且 CPRS 中的 I 位清除时发生

FIQ：在处理器外部中断申请管教有效（低电平）且 CPRS 中的 F 位清除时发生

二、异常发生时的状态转移

1. 硬件

- （1）当中断到来的时候，首先发生 PC 跳转
- （2）返回地址保存到异常模式的 lr 里面
- （3）将处理器的当前状态寄存器（cpsr）复制到异常模式的程序状态备份寄存器（spsr）

2. 软件

- （1）搭建异常向量表
- （2）写中断处理函数
 - 执行完中断处理程序之后，执行 pc 跳转
 - 模式切换（将异常模式的 spsr 赋值给 cpsr）
- （3）irq 模式下的栈（SP_IRQ）初始化

3. 异常处理步骤

在系统初始化的时候，先执行软件部分，然后等待中断的到来；当中断到来的时候，执行硬件部分

三、搭建异常向量表

0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

四、中断

1. 中断处理函数

Exception or entry	Return instruction	Previous state		Notes
		ARM r14_x	Thumb r14_x	
SVC	MOVS PC, R14_svc	PC + 4	PC+2	Where the PC is the address of the SVC, SMC, or Undefined instruction
SMC	MOVS PC, R14_mon	PC + 4	-	
UNDEF	MOVS PC, R14_und	PC + 4	PC+2	
PABT	SUBS PC, R14_abt, #4	PC + 4	PC+4	Where the PC is the address of instruction that had the prefetch abort
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC+4	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC+4	
DABT	SUBS PC, R14_abt, #8	PC + 8	PC+8	Where the PC is the address of the load or store instruction that generated the data abort
RESET	-	-	-	The value saved in r14_svc on reset is Unpredictable
BKPT	SUBS PC, R14_abt, #4	PC + 4	PC+4	Software breakpoint

中断函数模版

Func:

```
sub    r14, r14, #4
stmfd  sp!, {r0-r3, r14}
.....
ldmfd  sp!, {r0-r3, r14}^
```

```
//#4 是根据上面的表来更新
//把参数寄存器和程序寄存器压栈
//中断处理函数
//出栈、恢复 pc、^表示把 spsr 跟新到 cpsr
```

2. 软中断

swi 软中断号

软中断号 = (R14-4) & 0FFFFFFF

五、MMU (Memory Manager Unit)

1. 物理地址

前面操作的地址全部是物理地址

2. 虚拟地址空间

可以通过访问虚拟地址空间的内存来间接访问物理地址

3. MMU 的最大用处就是能够把虚拟地址的空间，随便映射到物理地址的空间；当一个程序多次运行的时候，MMU 会把这个程序的物理地址空间映射到各个虚拟地址空间；MMU 映射的基本单元是页；可以通过向协处理器激活 MMU

4. MMU 一旦被激活，就不能再访问物理地址了，只能访问虚拟地址；如果还想访问物理地址，只能建立相等映射，

DAY11—TFTP 原理、NFS 原理

一、嵌入式回顾

1. 嵌入式系统组成



2. 嵌入式硬件最小系统

- 内存
- 电源电路、复位电路
- 处理器（ARM）
- Flash
- 晶振、时钟电路

3. ARM 处理器的选择

- 处理器性能
- 成本
- 功耗
- 稳定性、可靠性：看芯片公司的规模、市场占有率
- 芯片的资源是否丰富（有很多资料可以查阅）
- 研发人员的技术是否到位

二、嵌入式 Linux 在开发板上运行过程

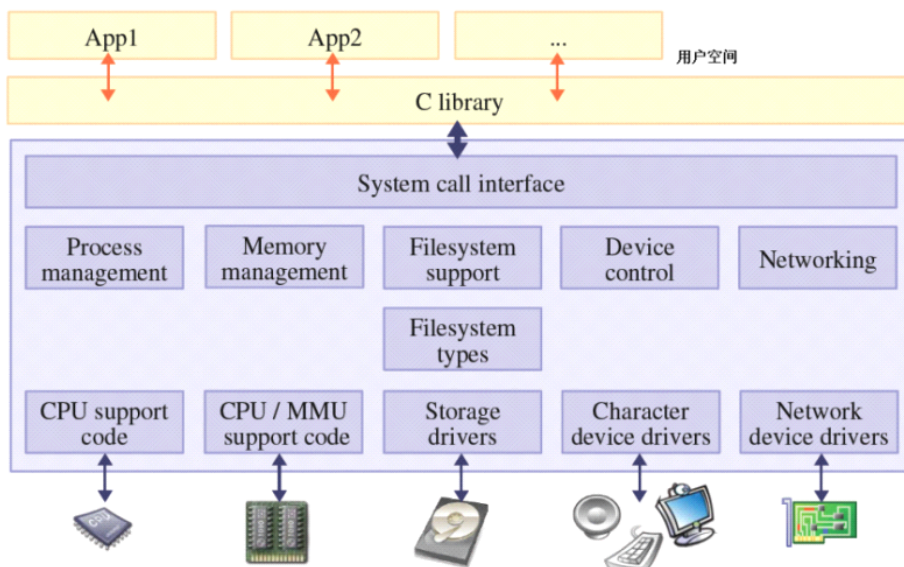
1. 开发板一开机就发生复位异常，跳转到异常向量表的复位异常处执行；一般来说，复位异常的程序在 0 地址处，而对于 S5PV210，0 地址处是内置 I Rom；
2. 一旦引导程序加载了 Linux 内核，引导程序的任务就结束了；
3. 内核放在 NandFlash 的什么地方是由用户决定，只要 UBoot 能够找到内核就行；
4. 可以由 Bootloader 向内核传送参数，这个参数可以决定内核到哪去执行第一个应用程序
5. 编译 uboot 和内核的时候不能使用任何 C 库，因为那个时候 C 库还没有移植；
6. 在测试程序阶段，内核可以通过网络（NFS）的方式访问文件系统，减少 NandFlash 的烧写次数

三、Linux 介绍

1. Linux 主要特性

- 自由、免费、开源
- 符合 IEEE POSIX 标准——编写程序更容易，软件兼容性好
- 稳定、可靠——完全运行于保护模式
- 支持硬件广泛、可移植性好
- 高可配置性，系统伸缩性强——模块化设计
- 强大的网络功能
- 真正的多任务、多用户系统
- 支持的文件系统格式丰富
- 安全性能好，少受病毒困扰

2. Linux 系统架构



四、搭建 **Linux** 系统移植开发环境

1. 安装交叉编译工具

- a) 取决于硬件环境（CPU 架构）
- b) 编译器从哪儿来：
 - 从原厂要
 - 从专业网上下: sourcery_g++_lite
- c) 解压、安装

2. 串口工具软件

- a) DNW
- b) SecureCRT
- c) kermit (kermit)
- d) 超级终端

3. TFTP 服务器

- a) 安装 tftp-hpa tftpd-hpa
`sudo apt-get install tftp-hpa tftpd-hpa`
- b) 配置 tftp
`vim /etc/default/tftpd-hpa`
`tftp-directory="/tftpboot"`
- c) 确保/tftpboot 目录存在，并且大家都权限
`chmod 777 /etc/default/tftp-hpa`
- d) 重启 tftp 服务
`sudo /etc/init.d/tftpd-hpa restart`

4. NFS 服务器

- a) 安装 nfs
`sudo apt-get install nfs-kernel-server`
- b) 配置 nfs
`sudo vim /etc/exports`
修改红色部分：
`/home/tarena/dlc/nfs *(rw, sync, no_root_squash)` //红色部分为共享目录
- c) 重启 nfs 服务
`sudo /etc/init.d/nfs-kernel-server restart`
- d) 挂载 nfs
`sudo mount -t nfs localhost:/home/tarena/workdir/rootfs/rootfs /mnt`

localhost:对方的 ip 地址

/home/tarena/workdir/rootfs/rootfs: 服务器的目录

/mnt: 挂载到自己的目录

e) 卸载 nfs

sudo umount /mnt

//mnt 为挂载的目录

五、烧写系统

1. 烧写 uboot

```
tftp 0x20008000 u-boot.bin
```

//0x20008000 不是固定的

```
nand erase 0x0 0x0100000
```

//从 0 地址开始, 清除 1M 的空间

```
nand write 0x20008000 0x0 0x0100000
```

//把在内存 0x20008000 中的数据写到 0 地址处, 大小为 1M

2. 烧写内核 zImage

```
tftp 0xc0008000 zImage
```

```
nand erase 0x500000 0x500000
```

//从 0x500000 地址开始, 清除 5M 的空间

```
nand write 0xc0008000 0x500000 0x500000
```

//把在内存 0xc0008000 中的数据写到 0x500000 地址处, 大小为 5M

3. 烧写跟文件系统

```
tftp 0xc0008000 rootfs.cramfs
```

```
nand erase 0xa00000 0x1000000
```

//从 0x500000 地址开始, 清除 16M 的空间

```
nand write 0xc0008000 0xa00000 0x1000000
```

//把在内存 0xc0008000 中的数据写到 0xa00000 地址处, 大小为 16M

4. 设置 Nandflash 启动的环境变量

```
setenv bootcmd nand read 0xc0008000 0x500000 0x500000 \; bootm 0xc0008000
```

```
setenv bootargs root=/dev/mtdblock3 init=/linuxrc console=ttySAC0,115200 rootfstype=cramfs
```

//-----

bootcmd: u-boot 启动以后自动执行的命令, 在没有人为干预的情况下执行命令列表

nand read 0xc0008000 0x500000 0x500000: 把 NandFlash 地址为 0x500000, 大小为 5M 的数据读到 0xc0008000

\; : 表示分开 2 个命令

bootm 0xc0008000: 引导内核的命令, 内核在内存地址 0xc0008000

bootargs: 指明了 uboot 给内核的参数

root=/dev/mtdblock3:告诉内核, 跟文件系统在那个设备上

init=/linuxrc: 告诉内核, 执行的第一个 init 进程 (调用的第一个应用程序是谁)

console=ttySAC0, 115200: 指定内核 console 终端使用的设备, 指定串口

nfsroot: 指明 nfs 在哪 192.168.1.8:/home....

IP: 本机地址:服务器地址:网关:子网掩码

rootfstype: 根文件系统类型 (yaffs2、fat32、cramfs 等), 文件系统就是数据在磁盘上格式

六、网络挂载

1. tftp 挂载 Linux 镜像 (zImage)

```
setenv bootcmd tftp c0008000 zImage \; bootm c0008000
saveenv
```

2. nfs 挂载根文件系统

```
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/home/tarena/dlc/tarena/my_rootfs
init=/linuxrc console=ttySAC0,115200
ip=192.168.1.6:192.168.1.8:192.168.1.1:255.255.255.0::eth0:on
```

3. nfs 挂载程序

```
mount -o nolock 192.168.1.8:/home/tarena/dlc/nfs /mnt/
```

DAY12—uboot 移植

一、bootloader 介绍

1. bootloder 两大功能

- 引导操作系统
- 调试、下载、烧写

2. bootloder 工作

bootloder 是上电以后，第一个自己写的移植的程序，执行很多跟硬件相关工作

- 关闭 CPU 中断
- 初始化 CPU 时钟
- 初始化内存
- 有部分代码是汇编语言写的
- 移植性比较差
- 短小精悍

3. 什么是 uboot

- uboot 是一个通用引导程序，支持硬件硬件平台多
- 支持引导多种操作系统
- 具有丰富功能

二、uboot 目录

1. 跟硬件相关目录

/cpu: 各种 CPU
/board: 各种开发板
/drivers: 通用驱动程序
/lib_XXX: 各种库
 lib_arm
 lib_mips
 lib_i386

2. 跟体系机构无关目录

/common: 通用程序（命令）

/api: 接口目录

/fs: 文件系统目录

/include: 头文件目录（与体系结构相关、无关的都有）

3. 可能需要修改目录

●/cpu

●/board

●/include

●/Makefile

三、编译 uboot

1. 配置

make CW210_config

在 Makefile 里相关内容:

```
CW210_config :   unconfig
```

```
@$(MKCONFIG) $(@:_config=) arm s5pv210 CW210 CONCENWIT s5pv210
```

```
//上一句相当于: @mkconfig CW210 arm s5pv210 CW210 CONCENWIT s5pv210
```

```
@echo "TEXT_BASE = 0xc3e00000" > $(obj)board/CONCENWIT/CW210/config.mk
```

1) 在 include 创建了一个 asm 符号连接: 在/asm/arch

连接到相关的体系结构目录: asm-arm

2) 在 include 目录下创建了 config.h

```
ARCH    = arm
```

```
CPU     = s5pv210
```

```
BOARD   = CW210
```

```
VENDOR  = CONCENWIT
```

```
SOC     = s5pv210
```

3) 在 linux 目录下创建了 config.h, 内容类似于#include <configs/xxx.h>:

```
#include <configs/CW210.h>
```

4) 为什么要符号连接符

```
ln -s asm-arm asm -> asm-arm/uart.h
```

```
ln -s asm-i386 asm- > asm-i386/uart.h
```

主要是为了编程方便

2. 编译

make

- 1) 根据 include/config.mk 确定使用那个 start.S 把公用和体系结构相关的 *.c/*.S 生成相应库, 并和 start.o 一起连接生成 u-boot
- 2) arm-linux-objcopy -O binary u-boot u-boot.bin 生成 u-boot.bin

3. 移植需要做的事情

1. 在 board 目录下创建自己的开发板目录
例如 smdk2440, 包含自己开发板的代码
找一个体系结构相近的过来, 修改
2. 在 cpu 目录下创建自己的 cpu 相关目录
找一个体系结构相近的 copy 过来, 修改
3. 在 include/configs/ 增加自己开发板相关的配置文件, 例如 smdk2440.h
例如 sam-yyy
4. 修改顶层 Makefile 文件, 增加一个 xxxx_config: 目标, 用于配置开发板 u-boot 生成配置
5. make XXX_config
6. make
7. 下载调试

注意:

- u-boot.bin 需要下载到内存什么位置,
在 u-boot_CW210_1.3.4/board/CONCENWIT/CW210/config.mk 中, 不是 u-boot.lds 中的 0x00000000
- u-boot 从哪启动
在 u-boot_CW210_1.3.4/board/CONCENWIT/CW210/u-boot.lds 中

DAY13—uboot 代码

一、S5PV210 的地址空间

Address		Size	Description	Note
0x0000_0000	0x1FFF_FFFF	512MB	Boot area	Mirrored region depending on the boot mode.
0x2000_0000	0x3FFF_FFFF	512MB	DRAM 0	外接内存1
0x4000_0000	0x7FFF_FFFF	1024MB	DRAM 1	外接内存2
0x8000_0000	0x87FF_FFFF	128MB	SROM Bank 0	
0x8800_0000	0x8FFF_FFFF	128MB	SROM Bank 1	程序运行的地方 uboot、zImage、rootfs 都在这执行
0x9000_0000	0x97FF_FFFF	128MB	SROM Bank 2	
0x9800_0000	0x9FFF_FFFF	128MB	SROM Bank 3	
0xA000_0000	0xA7FF_FFFF	128MB	SROM Bank 4	
0xA800_0000	0xAFFF_FFFF	128MB	SROM Bank 5	
0xB000_0000	0xBFFF_FFFF	256MB	OneNAND/NAND Controller and SFR	
0xC000_0000	0xCFFF_FFFF	256MB	MP3_SRAM output buffer	
0xD000_0000	0xD000_FFFF	64KB	IROM	
0xD001_0000	0xD001_FFFF	64KB	Reserved	
0xD002_0000	0xD003_7FFF	96KB	IRAM	usb.bin就是下到这里的，初始化IRAM
0xD800_0000	0xDFFF_FFFF	128MB	DMZ ROM	
0xE000_0000	0xFFFF_FFFF	512MB	SFR region	

二、IROM 操作顺序

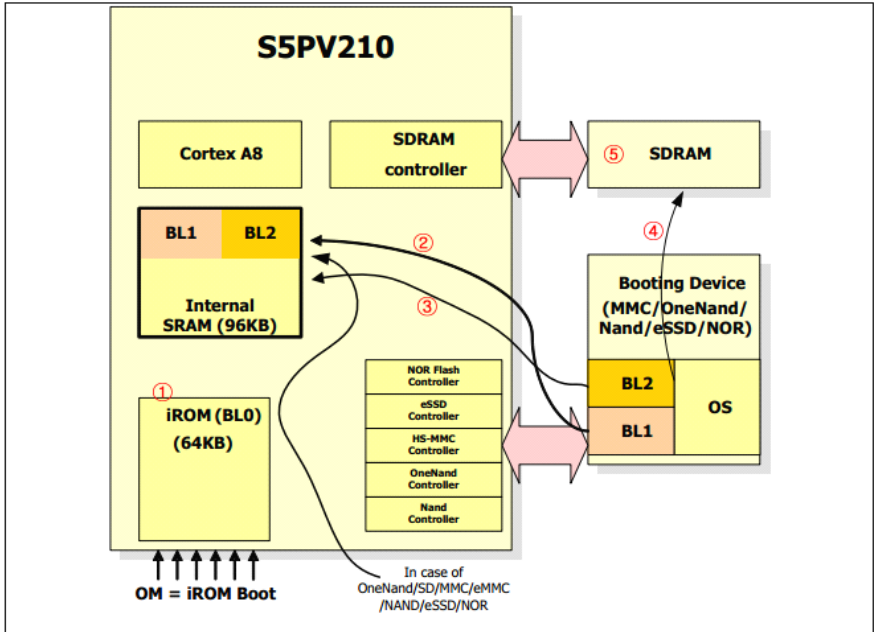


Figure 1. Overall boot-up diagram

BL1 / BL2 : It can be variable size copied from boot device to internal SRAM area.
BL1 max. size is 16KB. BL2 max. size is 80KB.

1. iROM can do initial boot up : initialize system clock, device specific controller and booting device.
2. iROM boot codes can load boot-loader to SRAM. The boot-loader is called BL1.
then iROM verify integrity of BL1 in case of secure boot mode.
3. BL1 will be executed: BL1 will load remained boot loader which is called BL2 on the SRAM
then BL1 verify integrity of BL2 in case of secure boot mode.
4. BL2 will be executed : BL2 initialize DRAM controller then load OS data to SDRAM.
5. Finally, jump to start address of OS. That will make good environment to use system.

三、IROM (BL0) 启动顺序

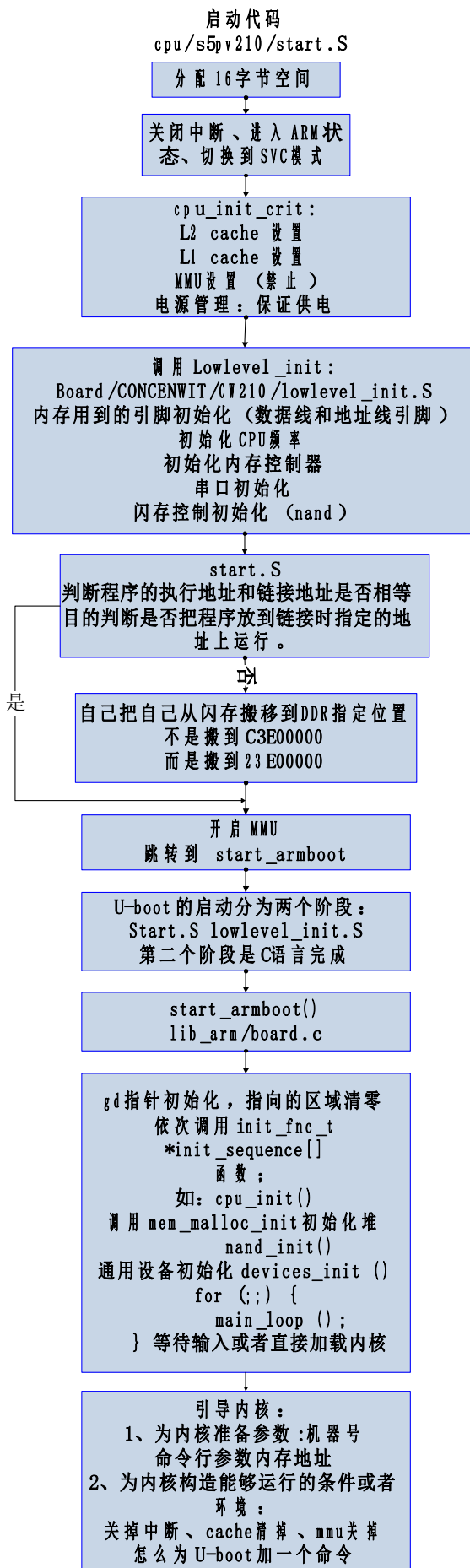
1. Disable the Watch-Dog Timer
2. Initialize the instruction cache
3. Initialize the stack region
4. Initialize the heap region.
5. Initialize the Block Device Copy Function.
6. Initialize the PLL and Set system clock.
7. Copy the BL1 to the internal SRAM region
8. Verify the checksum of BL1.
If checksum fails, iROM will try the second boot up. (SD/MMC channel 2)
9. Check if it is secure-boot mode or not.
If the security key value is written in S5PV210, It's secure-boot mode.
If it is secure-boot mode, verify the integrity of BL1.
10. Jump to the start address of BL1

四、u-boot 代码分析

加载内核的 id

在 common\cmd_bootm.c 中

```
#define LINUX_ZIMAGE_MAGIC 0x016f2818
```



DAY14—Linux 内核移植

一、Linux 内核

1. Linux 子系统

- 进程管理
- 内存管理
- 文件系统
- 网络协议
- 设备管理

2. Linux 内存空间

- 内核空间：3G~4G
- 用户空间：0G~3G
- 在用户空间和内核空间之间有一个系统调用层，系统调用层在内核空间

3. Linux 内核目录

arch/
 configs: 放的是相应平台编译的默认配置
 include: arm 体系结构的头文件目录
 kernel: 目录存放的进程管理和体系结构相关的代码
 mm: 内存管理部分和体系结构相关代码
 mach-***: 这个就是不同 CPU 和开发板的相关代码

二、配置、编译 Linux 内核

1. 修改顶层 Makefile

在 Makefile 190~194 行

```
CROSS_COMPILE ?= $(CONFIG_CROSS_COMPILE:"%"=%)    修改为:  
CROSS_COMPILE =arm-linux-  
ARCH           ?= arm          修改为:  ARCH = arm
```

2. 找一个参考配置

```
cp arch/arm/configs/smdkv210_android_defconfig .config
```

3. 配置内核

```
make menuconfig
```

内核配置主要是通过主 Kconfig 和每一级目录下的 Kconfig 这两个文件来决定的
Kconfig 是用来生成菜单的需要根据开发板的情况选择不同配置

4. 编译内核

```
make zImage 或者 make uImage
```

uImage 是 u-boot 的专用镜像文件，他是在 zImage 之前加上一个长度为 64 字节的头
在 arch/arm/boot/uImage (u-boot 加载内核的标准格式)
先拷贝 u-boot 的 /tools/mkimage 到 user/local/bin 目录下
然后 make uImage，在 arch/arm/boot 目录下生成 uImage 文件

5. 编译内核、模块

```
make modules
```

```
make modules_install
```

三、Kconfig 和 Makefile 语法

●在 Makefile 中

obj-y +=***.o: obj-y 包含的文件会被编译到 zImage

obj-m +=***.o: obj-m 包含的文件会被编译到模块，生成***.ko

●Kconfig 语法：可以在 Documentation/kbuild\$/ kconfig-language.txt 查询

```
menu "kernel_test"
    config kernel_test
        bool "aaa"
    config kernel_test2
        tristate "bbb"
    config kernel_test3
        string "ccc"
    config kernel_test4
        hex "ddd"
    config kernel_test5
        int "eee"
endmenu
```

- 修改 Kconfig 和 Makefile（以在 drivers/char 里面增加一个字符驱动设备为例）

修改 drivers/char/Kconfig

修改 driver/char/Makefile

增加类似: obj-\$(CONFIG_KERNEL_TEST) += yyy.o

- Eg: 在 driver/char 目录下修改 Kconfig

config MY_LEDS

tristate "MY LEDS' driver"

depends on MACH_CW210 || CPU_S5PV210

default y

在 driver/char 目录下修改 Makefile

obj-\$(CONFIG_MY_LEDS) += led_drv.o //()内部为“CONFIG_”+红色部分

四、内核 C 代码从哪开始

init//

在 arch/arm/目录下有很多关于平台相关的目录: mach-***、plat-***

五、其他命令

make mrproper: 清除所有配置文件和目标文件

insmod 模块名（绝对路径或者相对路径，自己根据情况判断）: 加载模块（驱动）

rmmod 模块名: 删除模块

lsmod: 查看内核里面有哪些模块已经加载

modprobe: 加载/下载内核模块，这个加载命令需要模块的依赖关系。

mknod: 创建设备节点

mknod 设备节点名 c/b 主设备号 次设备号

c 为字符设备，b 为块设备

mknod console c 5 1

mknod null c 1 3

六、内核配置

- General setup

通用的配置选项:

Cross-compile 前缀, arm-linux-

Init RAM disk

- Enable loadable module support

内核支持动态加载模块机制

- Enable block layer

块设备的属性

- System Type

MMU 使能

ARM system type ARM 处理器

输出信息 (console), 使用的串口, u-boot console=ttySAC0

●Kernel Feature

有内核空间和用户空间区域的划分, 使用默认配置即可。

●Boot options

default kernel commong string

内核启动参数的缺省配置,

bootloader 没有给内核传参数, 如果缺省参数配置配置了, 将使用这个配置。

●CPU Power Management

处理器的电源管理支持。

●Userspace binary formats

通常选 ELF 格式

●Networking support

网络配置、红外、CAN、蓝牙

●Device Drivers

设备驱动程序

Memory Technology Device (MTD) support

闪存的配置, 分区, 闪存驱动 (处理器部分)

● Network device support

网卡驱动。

● Input device support

输入设备, 鼠标, 键盘, 游戏手柄, 触摸屏

● Graphics support

显示设备的支持, VGA, LCD 等

我们的系统移植里面选三星的 lcd 接口设备。

● Sound card support

声卡支持: 有两种类型的驱动

1、ALSA 2、OSS

HID Devices

usb 主机接口设备

USB support

USB 支持, 包括 u 盘等。

MMC/SD/SDIO card support

MMC/SD 卡设备支持。

●File systems

ext2、ext3、ext4

yaffs2

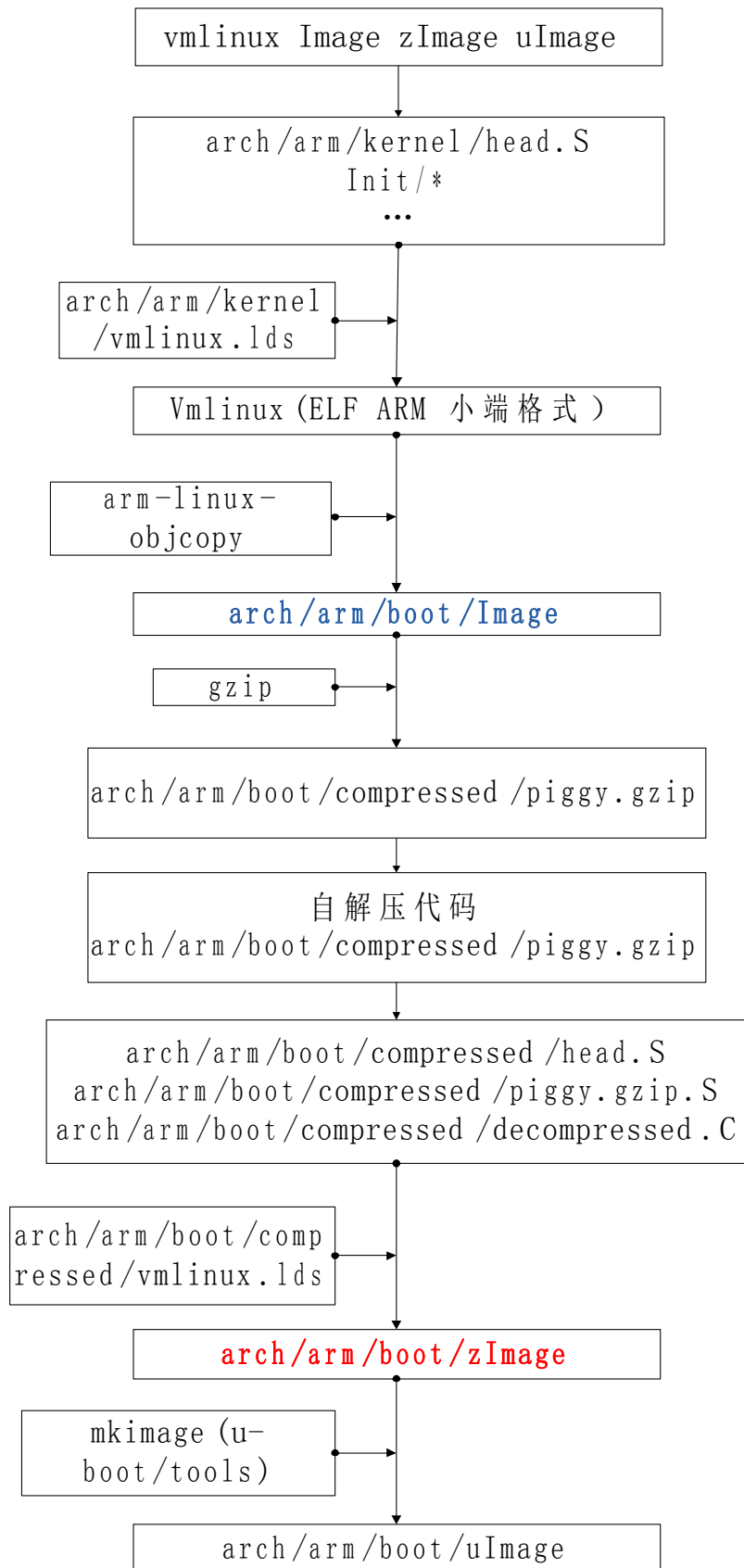
cramfs (默认配置是没有选上的)

NFS 文件系统

Enable loadable module support

本地语言支持

七、make zImage/uImage 的过程



DAY15—Linux 文件系统移植

一、文件系统

1. 什么是文件系统

文件系统是在存储设备上管理、组织数据的格式和实现机制

2. 文件系统格式

win: fat32、ntfs

linux: ext2、ext3、ext4、cramfs、yaffs2、ramfs、nfs、fat、ntfs、proc、sysfs

3. 文件系统存放位置

文件系统要放在设备上，如：flash、磁盘、光盘、u 盘、sd 卡、ram

4. 根文件系统

内核启动以后，第一个挂载点是“/”的文件系统，那么这个系统就是根文件系统

根文件系统存放

程序	/bin	/sbin	/usr/bin	/usr/sbin
	/usr/local/bin		/usr/local/sbin	

设备（设备节点/设备文件）	/dev
---------------	------

库（动态库和静态库）	/lib
------------	------

配置文件	/etc
------	------

/proc: proc 文件系统通常挂载到/proc，proc 文件系统为虚拟文件系统，实现用户空间程序和内核交互信息的

/sys: sysfs 挂载到/sys，sysf 虚拟文件系统，只存在内存中，和设备管理相关

二、文件系统

1. 创建相关目录

mkdir bin sbin dev etc proc sys tmp usr var mnt lib

2. 移植 busybox

busybox 用来把一些常用的命令做成一个软件包

●配置 busybox

make menuconfig

(1)Linux Module Utilities ->

```
[*] modinfo
[ ] Simplified modutils
[*] insmod
[*] rmmod
[*] lsmod
[*] Pretty output
[*] modprobe
[*] Blacklist support
[*] depmod
```

取消掉 simplified modutils, 把 insmod、rmmod、lsmod 选上

(2)Networking Utilities ->添加 ifconfig

```
[*] ifconfig
[*] Enable status reporting output (+7k)
[*] Enable slip-specific options "keepalive" and "outfill"
[*] Enable options "mem_start", "io_addr", and "irq"
[*] Enable option "hw" (ether only)
[*] Set the broadcast automatically
```

(3)Linux System Utilities ->支持 mdev, mount, umount

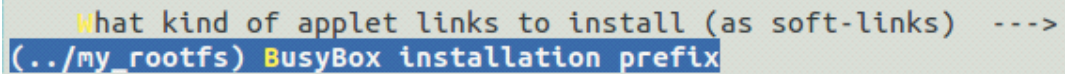
```
[*] Support option -f
[*] Support option -v
[ ] Support mount helpers
[*] Support specifying devices by label or UUID
[*] Support mounting NFS file systems
[*] Support mounting CIFS/SMB file systems
[*] Support lots of -o flags in mount
[*] Support /etc/fstab and -a
```

```
[*] mdev
[*] Support /etc/mdev.conf
[*] Support subdirs/symlinks
[*] Support regular expressions substitutions when renaming
[*] Support command execution at device addition/removal
[*] Support loading of firmwares
```

(4)Busybox Setting -> Build Option -> 支持大文件、修改交叉编译工具

```
[ ] Build BusyBox as a static binary (no shared libs) (NEW)
[ ] Build BusyBox as a position independent executable (NEW)
[ ] Force NOMMU build (NEW)
[ ] Build shared libbusybox (NEW)
[*] Build with Large File Support (for accessing files > 2 GB) (NEW)
(arm-linux-) Cross Compiler prefix
[ ] Additional CFLAGS (NEW)
```

(5)Busybox Setting -> Installaction Option -> 修改 busybox 安装的地方



```
What kind of applet links to install (as soft-links) --->
(../my_rootfs) BusyBox installation prefix
```

(6)Miscellaneous Utilities

```
->  [*] nandwrite
      [*] flashcp
      [*] flash_eraseall
```

●编译 busybox

```
make ARCH=arm CROSS_COMPILE=arm-linux-
```

●安装 busybox

```
make install
```

3. 创建必要设备节点

```
mknod console c 5 1
mknod null c 1 3
```

4. 创建动态链接库

把交叉编译工具/lib/目下的东西全部拷贝到 rootfs/lib 目录下

```
rm -rf *.o
```

```
rm -rf *.a: 删除所有的静态库
```

```
arm-linux-strip *.so: 删除调试信息
```

5. 修改 ubuntu 下的/etc/exports 文件

增加一行

```
/home/tarena/dlc/tarena/my_rootfs *(rw,sync,no_root_squash)
```

重启 nfs 服务器

```
sudo /etc/init.d/nfs-kernel-server restart
```

6. nfs 挂载根文件系统

修改 env

```
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/home/tarena/dlc/tarena/my_rootfs
```

```
init=/linuxrc console=ttySAC0,115200
```

```
ip=192.168.1.6:192.168.1.8:192.168.1.1:255.255.255.0::eth0:on
```

保存 env

saveenv

7. 打包

mkfs.cramfs my_rootfs rootfs.cramfs

三、拓展

1. 系统启动自动加载程序

(1)参考 example/inittab

在/etc/inittab 添加

<id>:<runlevels>:<action>:<process>

<id>: WARNING: This field has a non-traditional meaning for BusyBox init!

<runlevels>: The runlevels field is completely ignored.

<action>: Valid actions include: sysinit, respawn, askfirst, wait, once, restart, ctrlaltdel, and shutdown.

<process>: Specifies the process to be executed and it's command line.

eg:

```
::sysinit:/etc/init.d/rcS           //开机初始化时候执行
::shutdown:/bin/umount -a -r       //关机的时候执行
::restart:/sbin/init               //重启的时候执行
::askfirst:-/bin/ash               //终端按键后，执行 shall
::respawn:-/bin/ash                //永远执行 shall，无法在终端输入“exit”退出
```

(2)在/etc/init.d/创建 rcS，rcS 是我们写的第一个脚本，这个脚本是自动执行的程序

```
eg: #!/bin/sh                       //打开 shall
    echo "hello world"              //打印“hello world”
    bin/mount -a                    //挂载 fstab 文件中的设备
    bin/mount -t proc /proc         //直接在 rcS 挂载，也可以在 fstab 挂载
    echo /sbin/mdev > /proc/sys/kernel/hotplug //后面会讲
    mdev -s                         //后面会讲
```

(3)chmod 777 rcS：修改权限为可读可、写可、执行

2. 系统启动自动挂载文件

参考 docm/mdev

方法一：在 rcS 里面用 mount 挂载

方法二：在/etc 目录下创建 fstab 文件，将自动挂载的文件系统填入这个文件，具体格式参考 tts 文档，在 rcS 里面执行/bin/mount -a

```
eg:  # device      mount-point  type  options      dump  fsck order
      proc         /proc       proc  defaults     0     0
```

tmpfs	/tmp	tmpfs	defaults	0	0
sysfs	/sys	sysfs	defaults	0	0
tmpfs	/dev	tmpfs	defaults	0	0

3. 自动创建设备节点

```
[0] mount -t proc proc /proc
[1] mount -t sysfs sysfs /sys
[2] echo /sbin/mdev > /proc/sys/kernel/hotplug
[3] mdev -s
```

4. 手动挂载设备

查看设备节点信息

```
cat /sys/block/***/dev
mknod sda b 主设备号 次设备号
mount -t vfat /dev/sda /mnt: 挂载 U 盘
```

5. 自动挂载设备

在 dev 目录下创建 mdev.config

写上:

```
sd[a-z] 0:0 666 @/bin/mount -t vfat /dev/$MDEV /mnt
sd[a-z] 0:0 666 @/bin/umount /mnt
```

DAY15—Linux 开发环境搭建

一、学习课程

- 环境搭建

 - nfs

 - tftp

- 编译内核

- 内核相关知识

- 字符设备驱动

 - 按键驱动程序

 - input 子系统

 - platform 设备

 - i2c 总线驱动（EEPROM）

 - LCD 驱动

- 块设备驱动

 - 虚拟块设备驱动

- 网络设备启动

 - DM9000 驱动

二、GUN C 与标准 C 的区别

1. GUN C 支持 0 长度数组

```
struct var_char
{
    char data[0];
};
```

2. case 语句

标 C

```
case 0:
```

```
case 2:
```

```
.....
```

GUN C

```
case 2 3...2000
```

3. 宏定义

标准 C:

```
#define MINX((x),(y)) ((x)<(y)?(x):(y))  
int x = 10;  
int y = 20;  
int z = MIN(++x, ++y);
```

GUN C:

```
#define MIN(type, x,y) {type _x=x;type _y=y; _x<_y?_x:_y}  
int z = MIN(int,++x, ++y);  
{int _x=++x;int _y=++y;_x<_y?_x:_y}
```

typeof----->typedef

typeof 获取变量的类型

__attribute__((...)) 属性 ipares

防止编译器非法报警

二、驱动相关的内核源码目录

1. arch

->arm

->mach-xxx: 与开发板相关的目录

->plat-xxx: 平台相关目录

2. driver

三、编译模块

方法一：将.c 文件放入内核源码目录，让内核编译模块，需要修改 3 个目录：

Kconfig: 提供了菜单选择

Makefile: 决定了当前目录下的文件（夹）是编译进内核还是模块

.config: make menuconfig 退出保存时，是以变量的显示记录到文件中

●在 driver/char 目录下修改 Kconfig

config MY_LEDS

tristate "MY LEDS' driver"

depends on MACH_CW210 || CPU_S5PV210

default y

●在 driver/char 目录下修改 Makefile

obj-\$(CONFIG_MY_LEDS) += led_drv.o

//()内部为“CONFIG_”+红色部分，led_drv.o 为 driver/char 目录下的 led_drv.c

●make menuconfig

方法二：.c 文件不再网内核放，单独建文件夹编译

●单独写 Makefile

obj-m += ***, o

●在终端输入

```
make -C 内核源码目录/ M=$(pwd) modules
// -C: 指定内核源码目录
// M: 指定 Makefile
// modules: 编译成模块
```

方法三：全部直接写到 Makefile 里面去，直接 make

●vim Makefile

```
KERNELDIR ?= /home/tarena/dlc/tarena/driver/kernel
```

```
obj-m += hello.o
```

```
default:
```

```
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

●在终端直接输入：make

四、加载、卸载模块

```
insmod 模块名
```

```
rmmod 模块名
```

五、驱动模板

```
#include<linux/init.h>
```

```
#include<module.h>
```

```
static int __init hello_init(void) // __init 是将 hello_init 放到固定的段去
{
```

```
    return 0;
```

```
}
```

```
static void __exit hello_exit(void) // __exit 是将 hello_exit 放到固定的段去
{
```

```
}
```

```
module_init(hello_init);
```

```
module_exit(hello_exit);
```

```
MODULE_LICENSE("GPL");
```

// __ini 和 __exit 使用完函数后就释放内存，不再使用该函数

DAY16—内核模块、内存管理

一、内核导出符号

方法一：

a.c 中定义一个函数

```
int max(int,int)
```

b.c 中使用 max(x,y)

```
extern int max(int,int)
```

方法二：

a.h 头文件中声明

```
extern int max(int,int)
```

b.c 中包含头文件

```
#include "a.h"
```

二、printf

1. printk 优先级(按递减顺序)

KERN_EMERG "<0>" 用于处理紧急消息，通常是系统崩溃前的消息

KERN_ALERT "<1>" 需要立即处理的消息

KERN_CRIT "<2>" 严重情况

KERN_ERR "<3>" 错误情况

KERN_WARNING "<4>" 有问题的情况

KERN_NOTICE "<5>" 正常情况，但是仍然需要注意

KERN_INFO "<6>" 信息型消息

KERN_DEBUG "<7>" 用作调试信息

2. 查看/修改 printk 打印级别

```
cat /proc/sys/kernel/printk
```

```
7      4      1      7
```

第一个值：定义了可以打印到终端的最小优先级，printk 中使用的优先级小于这个值才能显示出来；修改第一个值：echo 8 >/proc/sys/kernel/printk

第二个值：printk 的默认优先级，如果不定义，默认为 4

三、模块参数

模块参数能够让用户空间修改内核空间的参数

1. 模块参数声明

- `module_param(name, type, perm);`
 - `name`: 参数名称
 - `type`: 参数数据类型
 - `perm`: 用户对模块参数的操作权限

2. 模块数组参数声明

- `module_param_array(name, type, nump, perm);`
 - `name`: 数组参数名称
 - `type`: 数组参数数据类型
 - `nump`: 如果数组参数在加载时设置，该值为加载时设置的数据个数，不允许传递比数组允许个数更多的值
 - `perm`: 用户对模块参数的操作权限

3. 模块参数支持的数据类型

- `bool` 布尔型
- `invbool` 布尔型反值
- `charp` 字符指针
- `short` 短整型
- `ushort` 无符号短整型
- `int` 整型
- `uint` 无符号整型
- `long` 长整型
- `ulong` 无符号长整型

4. 模块参数用户操作权限

- 应该使用<linux/stat.h>中定义的权限
 - `S_IRWXUSR`: 用户读、写、执行权限
 - `S_IRWXU`: 用户读、写、执行权限
 - `S_IRWXGRP`: 同组用户读、写、执行权限
 - `S_IRWXG`: 同组用户读、写、执行权限
 - `S_IRWXOTH`: 其他用户读、写、执行权限
 - `S_IRWXO`: 其他用户读、写、执行权限
 - `0`: 无任何权限

5. 例子

- 加载完模块之后 `cd /sys/module/modulepara/parameters`，可以查看模块的参数的值
通过在终端输入：`echo 222 > mpshort` 修改 `mpshort` 参数的值
- 在加载模块时也可以不使用原先的参数，在加载模块的时候再赋值
`insmod modulepara.ko mpshort=20 mpint=30 mpchar="tarena" mpparray=50,60`

四、内存管理

1. 地址

- 逻辑地址：汇编程序中出现的地址
 - 线性地址（虚拟地址）：32bit 表示的空间地址 0~4G
UC 开发时，使用的地址是 0~3G；内核开发时，使用的地址为 3~4G
 - 物理地址：实际出现在地址总线上的值
- 段式管理单元

逻辑地址 —————> 线性地址

页式管理单元

线性地址 —————> 物理地址

2. CPU

- 16 位 CPU，内部地址总线为 20 位，寻址空间为 1M，内部寄存器是 16 位，用 2 个寄存器来存放一个地址
 段基址寄存器：DS CS SS ES
 段内偏移寄存器（逻辑地址）：IP SP BX
 线性地址=段基址<4 + 段偏移地址
- 32 位 cpu，内部地址总线为 32 位，寻址空间为 4G，内部寄存器 32 位

3. 段式管理模式

- 实模式：类似与 16bit 段管理模式
- 保护模式：CPU 绝大部分时间工作于保护模式
- Linux 内核中没有（有限的）使用了段式管理模式，Linux 始终将段基地址寄存器赋值为 0，内核中逻辑地址=虚拟基地址

4. 页式管理模式

- Linux 内核中充分使用了页式管理机制使用的是 4 级页表管理结构

五、内核内存分配

1. kmalloc/kfree

●static __always_inline void *kmalloc(size_t size, gfp_t flags){...}

//参数一：要申请的字节大小

//参数二：标志

GFP_KERNEL: 分配内存，分配过程中可能导致睡眠(例如没有足够内存分配)

GFP_ATOMIC: 在中断中分配内存，失败就立即返回，不会导致睡眠

__GFP_DMA: 申请到的物理内存位于物理地址的 0~16M 空间，用于 DMA

__GFP_HIGHMEM: 用于分配高端内存（物理地址 869M 以上）的情况

●kfree(变量)

●eg:

```
static unsigned char* x=kmalloc(200, GFP_KERNEL);
```

```
kfree(x);
```

2. *__get_free_pages/free_pages

●unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order){...}

//参数一：分配标志，用于控制__get_free_pages 行为

//参数二：请求或释放的页数的 2 的幂，例如：操作 1 页该值为 0，操作 16 页该值为 4。

如果该值太大会导致分配失败，该值允许的最大值依赖于体系结构

//返回：指向分配到的连续内存区第一个字节的指针，失败返回 NULL

●void free_pages(unsigned long addr, unsigned int order)

//参数一：变量名

//参数二：要释放几页

●eg:

```
static unsigned char* y=(unsigned char *)__get_free_pages(GFP_KERNEL, 4);
```

```
free_pages((unsigned long)y, 4);
```

3. vmalloc/vfree

●void *vmalloc(unsigned long size);

// size: 待分配的内存大小，按字节计算，自动按页对齐

//返回：分配到的内核虚拟地址，失败返回 NULL

●void vfree(const void *addr);

// addr: 由 vmalloc 返回的内核虚拟地址

●eg:

```
static unsigned char* z=vmalloc(2048);
```

```
vfree(z);
```

六、GDB 调试

`gcc -g`

`gdb xxx`

`l`: 列出代码

`b` 行号: 在某一行加断点

`r`: 运行程序

`n/s`: 单步执行 (n 会加入函数)

`p` 变量: 打印变量

`q`: 退出调试

DAY17—内核链表、定时器

一、内核链表

1. 链表结构体

```
#include </linux/list.h>
struct my_list
{
    ....
    struct list_head *next, *prev;
}
```

2. 链表操作函数

●初始化头结点

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

●插入节点

```
static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

或者

```
static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
    __list_add(new, head->prev, head);
}
```

●删除节点

```
static inline void list_del(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}
```

●遍历链表

```
define list_for_each(pos, head) \
for (pos = (head)->next; prefetch(pos->next), pos != (head); pos = pos->next)
```

或者

```
#define list_for_each_safe(pos, n, head) \
for (pos = (head)->next, n = pos->next; pos != (head); pos = n, n = pos->next)
```

注意：list_for_each_safe 不会造成内核崩溃

●提取数据结构

```
define list_entry(ptr, type, member) container_of(ptr, type, member)
```

通过结构体某个成员变量的地址找到该结构体的地址

注意：list_entry(ptr, type, member)的功能和 container_of(ptr, type, member)一样，如果把结构体成员 struct list_head *next, *prev 放在结构体的第一个位置，那么就可以不要 list_entry 这个函数，因为 list_head 的地址就是结构体的地址

二、内核定时器

1. 常见名称

时钟中断

HZ：常数，决定了时钟中断发生的频率（一秒钟发生时钟中断的次数）

jiffies：核心变数，该变量记录了从开始发生的时钟中断次数

tick：发生时钟中断的时间间隔，tick=1/HZ

2. 内核定时器

在<linux/timer.h>中定义

```
struct timer_list
{
    struct list_head entry; // 内核使用
    unsigned long expires; // 超时时候 jiffies 的值
    void (*function)(unsigned long); // 超时处理函数
    unsigned long data; // 内核调用超时处理函数时传递给它的参数
    .....
};
```

3. 定时器函数

●初始化定时器函数：init_timer(timer)

●添加定时器函数：add_timer(struct timer_list * timer)

●删除定时器函数：del_timer(struct timer_list * timer)

●修改定时器函数：mod_timer(struct timer_list * timer, unsigned long expires)

●Linux 内核中只产生一次函数调用，如果要不断的产生定时器

- . function=void fun(unsigned long data)

三、系统调用

1. 空间访问

用户空间的程序是不能直接访问内核空间的资源，只有通过中断和系统调用才能从用户态进入内核态

2. 系统调用原理

- 应用程序首先用适当的值填充寄存器，然后调用一个特殊的指令，跳转到内核某个固定的位置；内核根据应用程序所填充的适当的值，来找到相应的函数执行
- 适当的值：/arch/arm/include/asm/unistd.h，2.6.35 支持的系统调用为 364 个
- 特殊的命令：arm->SWI(软中断)
- 固定的位置：在 arm 体系结构中，程序跳转到的位置在 entry-common.S 文件中的 vector_swi，在执行的过程中会用到 sys_call_table(call.S)
- 用户空间编程：fd=open(" a.txt", 0_RDWR)
5->寄存器
swi->vector_swi
sys_call_tables[5]
sys_open(open.c)
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)
{
}
} //把 open 拼接成 sys_open

3. 向内核中添加系统调用

(1)添加新的内核函数

```
vi arch/arm/kernel/sys_arm.c
asmlinkage long sys_add(int x, int y)
{
    printk("enter sys_add!\n");
    return x+y;
}
```

(2)更新头文件

```
arch/arm/include/asm/unistd.h
#define __NR_add      (__NR_SYSCALL_BASE+366)
```

(3)更新系统调用表

```
arch/arm/kernel/calls.S
/* 365 */      CALL(sys_recvmmsg)
```

CALL(sys_add)

(4)重新编译内核，使用新编译的内核启动开发板

DAY18—字符设备驱动

一、设备驱动分类

1. 字符设备驱动:
 顺序读写，不带缓冲区
2. 块设备驱动
 读写顺序不固定，不带缓冲区
3. 网络设备驱动

二、字符设备驱动框架

```
//#include <linux/cdev.h>
```

1. struct cdev

●cdev 对应了某个字符设备

●struct cdev

```
{  
    struct kobject kobj;  
    struct module *owner;  
    const struct file_operations *ops;           //设备操作函数命  
    struct list_head list;  
    dev_t dev;                                   //设备号  
    unsigned int count;  
};
```

●dev (32bits) =主设备号 (高 12bits) +次设备号 (低 20 位)

主设备号: 用于标识设备类型, 内核代码依据该号码对应设备文件和对应的设备驱动程序

次设备号: 用于标识同类型的不同设备个体, 驱动程序依据该号码辨删具体操作的是哪个设备个体

●设备号的选取:

(1)静态分配设备号

—查看设备号

cat /proc/ devices

或者查看 Linux 内核文档 document/devices

—注册设备号函数

register_chrdev_region(dev_t from, unsigned count, const char * name)

//from: 起始设备号

//count: 连续注册的设备号个数

//name: 注册设备号的名称

```

    // 返回 0 表示成功，非 0 表示失败
    register_chrdev      //不建议使用
(2)动态分配设备号
    alloc_chrdev_region(dev_t * dev, unsigned baseminor, unsigned count, const char * name)
    // baseminor: 基本设备号
    //count: 连续注册的设备号个数，一般为 1
    //name: 注册设备号的名称
(3)注销设备号函数
    unregister_chrdev_region(dev_t from, unsigned count)
    //from: 起始设备号
    //count: 连续注销的设备号个数，一般为 1

```

2. struct file_operations

file_operations 对应了字符设备的所有启动

```

struct file_operations
{
    .....
}

```

3. struct innod

```

struct innod
{
    struct cdev      *i_cdev      //设备
    dev_t            i_rdev;      //次设备号
}

struct chardevicedriver_cdev *pcdevp = NULL;
pcdevp = container_of(inode->i_cdev, struct chardevicedriver_cdev, cdev);

```

4. struct file

```

struct file
{
    loff_t            f_pos;      // 文件读写位置
    void              *private_data; // 文件私有数据
}

```


三、操作流程

1. 自动分配设备号

- 方法一：手动分配

```
dev_t my_dev=MKDEV(my_device_major,my_device_minor);
register_chrdev_region(dev_t from, unsigned count, const char * name)
```

- 方法二：自动分配

```
alloc_chrdev_region(dev_t * dev, unsigned baseminor, unsigned count, const char * name)
```

2. 初始化 **cdev** 结构体

```
cdev_init(struct cdev * cdev, const struct file_operations * fops)
```

// cdev: 设备指针

// fops: 设备操作

3. 注册 **cdev** 设备进内核

```
cdev_add(struct cdev * p, dev_t dev, unsigned count)
```

// cdev: 设备指针

// dev: 设备号（主设备号和次设备号结合）

//count: 注册个数，一般为 1

4. 添加设备节点

- 方法一：手动添加节点

在 dev 目录下 `mknod /dev/设备名称 c 主设备号 次设备号`

- 方法二：自动添加设备节点

第一步:注册设备类

```
class_create(owner, name)
```

//在 sys/class 目录下生成一个 **name** 的目录，

//owner:THIS_MODULE

//name:设备名字

eg: `dev_class=class_create(THIS_MODULE,"my_class");`

第二步:创建设备

```
device_create(struct class * cls, struct device * parent, dev_t devt, void * drvdata, const char *
fmt,...)
```

//在/sys/**name** 目录下生成 `fmt+...`的文件夹

// cls: class_create 所创建的设备类

// parent: 父设备，一般为 NULL

// devt: 设备

```
// drvdata: 一般为 NULL
//fmt: 设备名称
eg: dev_device=device_create(dev_class,NULL,my_dev_t,NULL,"mydriver%d",3);
```

●注意:

"mydriver%d",3: 表示"mydriver3"也可以写成还有"mydriver%d",MAJOR(my_dev_t)

mdev : 监视/**sys** 该目录下, 根据目录的变化, 动态创建/**dev** 下的设备节点文件

procfs 文件系统-->/proc: 内存中的文件系统, 在内核执行的时候导出的信息

sysfs 文件系统-->/sys: 内存中的文件系统, 在 2.6 的内核才出现, 设备

5. 编写 file_operations 函数

6. module_exit 需要做的工作

●从内核中删除设备

```
void device_destroy(struct class *class, dev_t devt)
```

●从内核中删除设备类

```
void class_destroy(struct class *cls)
```

●从内核中删除 c_dev 设备

```
void cdev_del(struct cdev *p)
```

●注销准备

```
void unregister_chrdev_region(dev_t from, unsigned count)
```

DAY19—错误处理

1. 用户空间

段错误: core 调试方法

(1)ulimit -c unlimited(取消掉 core 大小限制)

(2)./test : 产生 core 文件 an

(3)gdb test -c core

(4)->bt

2. 内核调试:

(1)内核空间出错分析

/* 出错原因*/

Unable to handle kernel NULL pointer dereference at virtual address 00000000

pgd = f34e8000

[00000000] *pgd=534ba031, *pte=00000000, *ppte=00000000

Internal error: Oops: 817 [#1] PREEMPT

last sysfs file: /sys/devices/virtual/chardevicedriver_class/chardevicedriver0/dev

Modules linked in: chardevicedriver

CPU: 0 Not tainted (2.6.35.7-Concenwit #3)

PC is at chardevicedriver_open+0x1c/0x44 [chardevicedriver]

LR is at chrdev_open+0x260/0x288

内核崩溃前 CPU 内部寄存器的值

pc : [<bf0000b8>] lr : [<c00eabb4>] psr: a0000013

sp : f34dbdb0 ip : f34dbdd0 fp : f34dbdcc

r10: f3474180 r9 : c00eaac8 r8 : 00000000

r7 : 00000001 r6 : f34e0540 r5 : f34e0540 r4 : 00000000

r3 : 00000064 r2 : 00000000 r1 : f3474180 r0 : bf000396

Flags: NzCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user

Control: 10c5387d Table: 534e8019 DAC: 00000015

LR: 0xc00eab34:

ab34 e2422001 e5832004 e5933000 e3130002 0a000000 eb11e614 e3a07001 e1a0200d

ab54 e3c23d7f e3c3303f e5932004 e2422001 e5832004 e5933000 e3130002 0a000000

ab74 eb11e609 e3570000 0a000001 e5953028 ea000000 e3a03000 e3530000 e58a3010

ab94 03e04005 0a000007 e5933030 e3530000 0a000006 e1a00006 e1a0100a e12fff33

abb4 e2504000 0a000001 e1a00005 ebffff59 e1a00004 e24bd028 e89daff0 c0841a10

abd4 c07e8c80 c00eaac8 e1a0c00d e92dd830 e24cb004 e59f301c e1a04000 e5901034

abf4 e5942038 e5930000 eb0742b7 e1a00004 eb05bc28 e89da830 c0841a10 e1a0c00d
ac14 e92dd800 e24cb004 e24dd010 e59fe030 e1a0c000 e5801034 e3a03000 e5802038

SP: 0xf34dbd30:

bd30 f346fa80 f34c1100 f34dbd5c f34dbd48 c0259e78 c025b088 ffffffff f34dbd9c
bd50 f34e0540 00000001 f34dbdcc f34dbd68 c0036a6c c00362ac bf000396 f3474180
bd70 00000000 00000064 00000000 f34e0540 f34e0540 00000001 00000000 c00eaac8
bd90 f3474180 f34dbdcc f34dbdd0 f34dbdb0 c00eabb4 bf0000b8 a0000013 ffffffff
bdb0 00000000 f346fa80 f34e0540 00000001 f34dbe04 f34dbdd0 c00eabb4 bf0000a8
bdd0 f3991380 00000000 f34dbdf4 f3474180 f3447d00 f34e0540 00000000 00000000
bdf0 f3991380 c00ea954 f34dbe3c f34dbe08 c00e59dc c00ea960 f34dbe24 f34dbe18
be10 c0221d6c f3474180 00000000 00000002 00000026 00000000 f3868380 00000000

IP: 0xf34dbd50:

bd50 f34e0540 00000001 f34dbdcc f34dbd68 c0036a6c c00362ac bf000396 f3474180
bd70 00000000 00000064 00000000 f34e0540 f34e0540 00000001 00000000 c00eaac8
bd90 f3474180 f34dbdcc f34dbdd0 f34dbdb0 c00eabb4 bf0000b8 a0000013 ffffffff
bdb0 00000000 f346fa80 f34e0540 00000001 f34dbe04 f34dbdd0 c00eabb4 bf0000a8
bdd0 f3991380 00000000 f34dbdf4 f3474180 f3447d00 f34e0540 00000000 00000000
bdf0 f3991380 c00ea954 f34dbe3c f34dbe08 c00e59dc c00ea960 f34dbe24 f34dbe18
be10 c0221d6c f3474180 00000000 00000002 00000026 00000000 f3868380 00000000
be30 f34dbe5c f34dbe40 c00e5bd8 c00e57cc f3474700 00000000 00000002 f34dbec8

FP: 0xf34dbd4c:

bd4c f34dbd9c f34e0540 00000001 f34dbdcc f34dbd68 c0036a6c c00362ac bf000396
bd6c f3474180 00000000 00000064 00000000 f34e0540 f34e0540 00000001 00000000
bd8c c00eaac8 f3474180 f34dbdcc f34dbdd0 f34dbdb0 c00eabb4 bf0000b8 a0000013
bdac ffffffff 00000000 f346fa80 f34e0540 00000001 f34dbe04 f34dbdd0 c00eabb4
bdcc bf0000a8 f3991380 00000000 f34dbdf4 f3474180 f3447d00 f34e0540 00000000
bdec 00000000 f3991380 c00ea954 f34dbe3c f34dbe08 c00e59dc c00ea960 f34dbe24
be0c f34dbe18 c0221d6c f3474180 00000000 00000002 00000026 00000000 f3868380
be2c 00000000 f34dbe5c f34dbe40 c00e5bd8 c00e57cc f3474700 00000000 00000002

R1: 0xf3474100:

4100 f3c02d10 f3c02d10 f3447d00 f3868600 f3855280 f3cf8000 f3474118 f3474118
4120 f3447d18 f3447d18 00000020 f3476740 f3c00808 f3447d30 f3474138 f3474138
4140 f3474140 f3474140 f3474148 f3474148 f3474150 f3474150 00000000 f3c00800
4160 00000013 00000000 00000001 00000000 00000000 00000000 00000000 00000000
4180 f34b7e80 f34b6074 f3447d00 f3991380 bf000514 00000001 00000002 0000001f
41a0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 f3474700
41c0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
41e0 00000000 00000000 00000000 00000000 f34741f0 f34741f0 f34e05e0 00000000

R5: 0xf34e04c0:

```
04c0  00000000 00000000 00000000 00000000 00000000 00000000 f34e04e0 ffffffff
04e0  00200000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0500  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0520  00000000 00000000 00000000 00000000 00000000 f34e0534 f34e0534 00000000
0540  00000000 00000000 f3992148 f399ca80 f34e03b0 f34b6068 f39913c0 f39913c0
0560  000003c6 00000001 00000001 00000000 00000000 0f800000 0000000c 00000000
0580  00000000 00000000 00000000 00000000 50ddcb08 36b7176e 50ddcb08 36b7176e
05a0  50ddcb08 36b7176e 00000000 00000000 21b00000 00000001 f34e05b8 f34e05b8
```

R6: 0xf34e04c0:

```
04c0  00000000 00000000 00000000 00000000 00000000 00000000 f34e04e0 ffffffff
04e0  00200000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0500  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0520  00000000 00000000 00000000 00000000 00000000 f34e0534 f34e0534 00000000
0540  00000000 00000000 f3992148 f399ca80 f34e03b0 f34b6068 f39913c0 f39913c0
0560  000003c6 00000001 00000001 00000000 00000000 0f800000 0000000c 00000000
0580  00000000 00000000 00000000 00000000 50ddcb08 36b7176e 50ddcb08 36b7176e
05a0  50ddcb08 36b7176e 00000000 00000000 21b00000 00000001 f34e05b8 f34e05b8
```

R9: 0xc00eaa48:

```
aa48  e3a04000 e1a07004 e1a0200d e3c23d7f e3c3303f e5932004 e2422001 e5832004
aa68  e5933000 e3130002 0a000000 eb11e649 e1a00007 ebffffaa e3540000 1a00004e
aa88  e5953028 e3530000 0a00003c e5937000 e3570000 0a000037 e1a0200d e3c23d7f
aaa8  e3c3303f e5932004 e2822001 e5832004 e5972000 e3520002 01a07004 0a000021
aac8  e5971148 e59f2100 e59f9100 e5910000 e2800001 e5810000 e5921004 e3510000
aae8  0a000017 e5931004 e2811001 e5831004 e5928010 e3580000 0a000007 e5983000
ab08  e1a01007 e5980004 e1a02009 e12fff33 e5b83008 e3530000 eaffffff e1a0200d
ab28  e3c23d7f e3c3303f e5932004 e2422001 e5832004 e5933000 e3130002 0a000000
```

R10: 0xf3474100:

```
4100  f3c02d10 f3c02d10 f3447d00 f3868600 f3855280 f3cf8000 f3474118 f3474118
4120  f3447d18 f3447d18 00000020 f3476740 f3c00808 f3447d30 f3474138 f3474138
4140  f3474140 f3474140 f3474148 f3474148 f3474150 f3474150 00000000 f3c00800
4160  00000013 00000000 00000001 00000000 00000000 00000000 00000000 00000000
4180  f34b7e80 f34b6074 f3447d00 f3991380 bf000514 00000001 00000002 0000001f
41a0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 f3474700
41c0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
41e0  00000000 00000000 00000000 00000000 f34741f0 f34741f0 f34e05e0 00000000
```

Process test (pid: 73, stack limit = 0xf34da2f0)

内核崩溃时堆栈中的数据

Stack: (0xf34dbdb0 to 0xf34dc000)

```
bda0: 00000000 f346fa80 f34e0540 00000001
bdc0: f34dbe04 f34dbdd0 c00eabb4 bf0000a8 f3991380 00000000 f34dbdf4 f3474180
```

```

bde0: f3447d00 f34e0540 00000000 00000000 f3991380 c00ea954 f34dbe3c f34dbe08
be00: c00e59dc c00ea960 f34dbe24 f34dbe18 c0221d6c f3474180 00000000 00000002
be20: 00000026 00000000 f3868380 00000000 f34dbe5c f34dbe40 c00e5bd8 c00e57cc
be40: f3474700 00000000 00000002 f34dbec8 f34dbe94 f34dbe60 c00f2d4c c00e5b9c
be60: f34dbe94 f34dbe70 c00e8944 f34dbec8 00000002 f34da000 00000000 00000026
be80: f34dbf28 00000026 f34dbf5c f34dbe98 c00f4a40 c00f2898 00000000 f34f2000
bea0: f34dbf20 f34dbedc fffff9c f34dbedc 00000000 00000000 c0080c40 f34f2000
bec0: 00000003 00000000 f3447d00 f3991380 273a8f10 00000011 f34f2005 f346c080
bee0: f3861e80 00000101 00000000 00000000 00000000 40025000 f34dbfac f34dbf08
bf00: c00362dc c003d8d8 f346f908 f346f900 00000000 00000003 00000000 f3474180
bf20: f34dbf5c f34dbf30 f3447d00 f3991380 00000000 00000000 00000003 00000002
bf40: becd1ecc f34f2000 f34da000 00000000 f34dbf94 f34dbf60 c00e56b0 c00f48c0
bf60: 00000000 c025de10 f34dbfa4 00008758 00000000 000084e8 00000005 c0037168
bf80: f34da000 00000000 f34dbfa4 f34dbf98 c00e57a0 c00e5658 00000000 f34dbfa8
bfa0: c0036fc0 c00e5788 00008758 00000000 000087cc 00000002 becd1ecc 00008594
bfc0: 00008758 00000000 000084e8 00000005 00000000 00000000 40025000 becd1d74
bfe0: 00000000 becd1d68 000085b0 400db88c 60000010 000087cc 210d1021 210d1421

```

系统崩溃前函数调用关系

Backtrace:

```

[<bf00009c>] (chardevicedriver_open+0x0/0x44 [chardevicedriver]) from [<c00eabb4>]
(chrdev_open+0x260/0x288)

```

```

r7:00000001 r6:f34e0540 r5:f346fa80 r4:00000000

```

```

[<c00ea954>] (chrdev_open+0x0/0x288) from [<c00e59dc>] (__dentry_open+0x21c/0x340)

```

```

[<c00e57c0>] (__dentry_open+0x0/0x340) from [<c00e5bd8>] (nameidata_to_filp+0x48/0x60)

```

```

[<c00e5b90>] (nameidata_to_filp+0x0/0x60) from [<c00f2d4c>] (do_last+0x4c0/0x620)

```

```

r4:f34dbec8

```

```

[<c00f288c>] (do_last+0x0/0x620) from [<c00f4a40>] (do_filp_open+0x18c/0x4e8)

```

```

[<c00f48b4>] (do_filp_open+0x0/0x4e8) from [<c00e56b0>] (do_sys_open+0x64/0x11c)

```

```

[<c00e564c>] (do_sys_open+0x0/0x11c) from [<c00e57a0>] (sys_open+0x24/0x28)

```

```

[<c00e577c>] (sys_open+0x0/0x28) from [<c0036fc0>] (ret_fast_syscall+0x0/0x30)

```

```

Code: e3a04000 e1a05000 e3a03064 e59f0020 (e5843000)

```

```

---[ end trace 38e7723cb4bda835 ]---

```

Segmentation fault

由于是堆栈，最先调用的函数在下面，最后调用的函数在上面

(2)内核崩溃调试方法：

PC 指针 bf0000b8

0. 通过出错信息，得到是在那个函数中出的错

1. 通过/proc/kallsyms 该文件结合 PC 指针，确定内核崩溃指针在那个 xxx.ko 文件中及其函数在程序中运行时的地址。

/proc/kallsyms 列出了内核用到的所有函数以及函数的实际调用地址，

2. arm-linux-objdump -D xxx.ko > 2.txt

确定该函数在程序中的偏移地址，例如下图中的 000001cc

```
...  
000001cc <buttons_timer_function>:  
1cc: e1a0c00d      mov     ip, sp  
1d0: e92dd8f0      push   {r4, r5, r6, r7, fp, ip, lr, pc}  
1d4: e24cb004      sub     fp, ip, #4
```

3. 函数的实际调用地址-函数在代码中的偏移地址=函数代码在程序中地址和实际运行地址之间的差值
4. PC 指针-差值=出错语句在程序中的偏移地址，把这个地址和反汇编代码一比较，就能够找到该语句对应的汇编指令

DAY20 一点亮 LED 灯、竞态与并发

一、原理图

- 点亮灯需要相应的管脚输出高电平
 - LED1 连接 CPU 的管脚为 GPC1_32. 内核中提供了一套 GPIO 操作的函数，操作相应的寄存器
2. CPU datasheet
 - CONF: GPC1CON->0xe0200080
 - PUD: GPC1DAT -> GPC1DAT +4
 - DATA: GPC1DAT -> GPC1CON +4

二、控制 LED 灯

1. 直接操作寄存器地址对应的虚拟地址

内核编程用到的都是虚拟地址，如果要物理地址，需要通过映射，区地址赋值，在操作相应的寄存器

2. 调用内核提供的 GPIO 操作函数

- 内核中将 GPIO 管脚统一做了编号，使用时如同 S5PV210_GPC1(3)

```
#define S5PV210_GPC1(_nr)    (S5PV210_GPIO_C1_START + (_nr))  
S5PV210_GPIO_C1_START = S5PV210_GPIO_NEXT(S5PV210_GPIO_C0),  
#define S5PV210_GPIO_NEXT(__gpio) \  
    ((__gpio##_START) + (__gpio##_NR) + CONFIG_S3C_GPIO_SPACE + 1)  
上一句相当于->#define S5PV210_GPIO_NEXT(S5PV210_GPIO_C0) \  
    (S5PV210_GPIO_CO_START)+( S5PV210_GPIO_CO_NR)+ CONFIG_S3C_GPIO_SPACE + 1
```
- 管脚申请
可以在**_init 的地方告诉内核，也可以在**_open 的地方告诉内核
- 设置相应的 GPIO 管脚为输出管脚，并初始化

```
gpio_direction_output(S5PV210_GPC1(3),0);
```
- 禁止内部上拉

```
s3c_gpio_setpull(S5PV210_GPC1(3), S3C_GPIO_PULL_NONE);
```
- 设置输出的电平

```
gpio_set_value(S5PV210_GPC1(3),1);    //输出为高电平  
gpio_set_value(S5PV210_GPC1(3),0);    //输出为低电平
```


三、竞态与并发

1. 产生原因

(1)多 CPU 之间竞争操作

(2)单 CPU 操作系统支持进程抢占

Linux 不是实时操作系统，Linux 靠的是**时间片轮转+部分优先级**这种方式进行进程调度的
实时操作系统有 Ucos-II、Vxworks

(3)中断和进程之间抢断资源

(4)中断和中断之间抢占资源

2. 内核中的解决方式

(1)中断屏蔽（对单 CPU 而言，不建议编程时使用，因为 Linux 的进程调度是靠中断实现的）

(2)原子操作

(3)自旋锁

(4)信号量

为了使一个文件只能被一个用户打开，可以通过原子操作、自旋锁、信号量的例子实现

四、原子操作

1. 位原子操作

设置位

```
set_bit(nr, void *addr);           //将 addr 地址内数据的第 nr 位设置为 1
```

清除位

```
clear_bit(nr, void *addr);         //将 addr 地址内数据的第 nr 位设置为 0
```

2. 整形原子操作

●定义在 arch/arm/include/asm/atomic.h 中

```
typedef struct
{
    volatile int counter;
} atomic_t;
```

3. 原子操作和普通操作比较

原子操作能够保证一个操作要么全执行，要么全不执行，不会被打断

五、自旋锁

1. 定义

- 自旋锁最多只能被一个执行单元持有
- 自旋锁不会引起进程睡眠，如果一个执行进程试图获取一个已经被持有的自旋锁，那举进程就会一直进行忙循环，一直等待下去，等在返里看是否该自旋锁的持有者已经释放了锁，“自旋”近似于在“原地转圈”的意思

- 定义在 linux/spinlock_types.h

```
typedef struct
{
    raw_spinlock_t raw_lock;
    ...
} spinlock_t;
typedef struct
{
    volatile unsigned int lock;
} raw_spinlock_t;
```

2. 使用自旋锁

- 定义自旋锁

```
spinlock_t lock;
```

- 初始化自旋锁

```
spin_lock_init(&lock)
```

- 获取自旋锁

```
spin_lock(spinlock_t * lock) //获取锁成功立即返回，不成功就自旋等待
```

```
spin_trylock(spinlock_t * lock) //获取锁成功立即返回，不成功立刻返回一个错误值
```

- 释放自旋锁

```
spin_unlock(spinlock_t * lock)
```

注意：spin_lock 和 spin_unlock 是成对出现的，如果连续释放两次，可能会导致内核崩溃，spin_trylock 和 spin_unlock 不一定是成对出现的，如果锁住了，就要 spin_unlock

六、信号量

1. 定义

- 信号量(semaphore)是用于保护临界区的一种内核的信号量在概念和原理上和用户态的信号量是一样的，但是它不能在内核之外使用，内核信号量实际上是一种睡眠锁
- 如果有一个任务想要获得已经被占用的信号量时，信号量会将返个进程放入一个等待队

列，然后让其睡眠当持有信号量的进程将信号释放后，处于等待队列的进程将被唤醒，并让其获得信号量

●定义在<linux/semaphore.h>中

```
struct semaphore
{
    spinlock_t lock;
    unsigned int count;
    struct list_head wait_list;
};
```

2. 使用信号量

●定义自旋锁

```
struct semaphore sem
```

●初始化信号量

—`sema_init(struct semaphore *sem, int val);`

用于初始化信号量，并设置信号量 `sem` 的值为 `val`

—`init_MUTEX(struct semaphore *sem);`

用于初始化信号量，并将信号量 `sem` 的值设置为 1

—`init_MUTEX_LOCKED(struct semaphore *sem);`

用于初始化信号量，并将信号量 `sem` 的值设置 0；也就是在创建时就处于已锁状态

—`DECLARE_MUTEX(name);`

定义一个名称为 `name` 的信号量，并将信号量初始化为 1

●获取信号量

—`void down(struct semaphore *sem);`

获取信号量 `sem`；该函数可能寻致进程睡眠，因此不能在中断上下文中使用

—`int down_interruptible(struct semaphore *sem);`

获取信号量 `sem`；如果信号量不可用，进程将被设置为 `TASK_INTERRUPTIBLE` 类型的睡眠状态,该函数由返回值来区分正常返回还是被信号中断返回；如果返回 0，代表获取信号量正常返回；如果返回非 0，代表被信号打断

—`int down_killable(struct semaphore *sem);`

获取信号量 `sem`；如果信号量不可用，进程将被设置为 `TASK_KILLABLE` 类型的睡眠状态

—`int down_trylock(struct semaphore *sem);`

该函数尝试获取信号量 `sem`；如果能够立即获得，它就获得信号量并返回 0；否则，返回非 0 值；它不会寻致调用者睡眠，可以在中断上下文中使用

●释放信号量

—`void up(struct semaphore *sem);`

该函数释放信号量 `sem`；实质上是把 `sem` 的值加 1，如果 `sem` 的值为非正数，表明有任务等待该信号量，因此需要唤醒等待者

七、信号量和自旋锁的区别

●信号量的实现依赖于自旋锁

- 信号量可以有多个持有，自旋锁只允许一个持有
- 自旋锁包含代码区（临界资源）通常非常短，信号量的保护区域时间通常比较长
- 自旋锁原地自旋，消耗大量的 CPU 资源，信号量使进程睡眠，当能获取信号量时唤醒睡眠的进程

DAY21—等待队列、多路监听、地址映射

一、等待队列

1. 等待队列的数据结构

```
●/include/linux/wait.h
●struct __wait_queue_head
{
    spinlock_t lock;
    struct list_head task_list;
};
●struct __wait_queue
{
    unsigned int flags;
#define WQ_FLAG_EXCLUSIVE    0x01
    void *private;
    wait_queue_func_t func;
    struct list_head task_list;
};
```

2. 等待队列的操作函数

```
●include/linux/wait.h
●定义和初始化等待队列头
wait_queue_head_t wqh;           //定义等待队列头
init_waitqueue_head(q);          //初始化等待队列头
DECLARE_WAIT_QUEUE_HEAD(name)    //定义并初始化等待队列头
DECLARE_WAITQUEUE(wq, current)   //定义并初始化等待队列
//current 标志 CPU 正在执行的进程，使等待队列和当前进程关联起来

●添加、移除等待队列
add_wait_queue();
remove_wait_queue();

●等待事件
- wait_event(wq, condition)
    当 condition 为真时，立即返回，否则进程进入 TASK_INTERRUPTIBLE，并挂载 queue 指定的等待队列上
- wait_event_interruptible(wq, condition)
```

—wait_event_interruptible_timeout(wq, condition, timeout)
—wait_event_killable(wq, condition)
—wait_event_timeout(wq, condition, timeout)
●唤醒队列（唤醒睡眠的进程）
—wake_up(.....)

3. 使用步骤

方法一：

●睡眠过程：

- (1)定义并初始化等待队列
- (2)定义并初始化等待队列头
- (3)添加队列头到指向的等待队列链表中去
- (4)如果不满足条件，随着该进程睡眠，并且主动放弃 CPU，让 CPU 调度其他进程
- (5)唤醒之后可以从等待队列中删除等待队列头，也可以不删除
- (6)设置该进程为运行模式

●其它程序唤醒某个进程的过程：

通过 wake_up 方式在其他进程中唤醒睡眠的进程

●等待队列头其实就是该进程的标志，可以通过这个头对某个进程进行操作

●eg:

```
/*定义等待队列头*/
wait_queue_head_t wqh;
/*初始化等待队列头*/
init_waitqueue_head(&pcdevp->wqh);
/*定义等待队列，使等待队列和当前进程关联起来*/
DECLARE_WAITQUEUE(wq, current);
/*添加队列头 wqh，到指向的等待队列链表中去*/
add_wait_queue(&(pcdevp->wqh), &wq);
/*假设 led 为 0，表示没有数据供读*/
if(pcdevp->led==0)
{
    printk("no data for reading\n");
    /*设置当前进程的状态为中断可打断的睡眠*/
    set_current_state(TASK_INTERRUPTIBLE);
    /*让出 CPU，重新进行调度*/
    schedule();
    printk("have data for reading\n");
}
/*唤醒后从指定链表中删除等待队列*/
remove_wait_queue(&(pcdevp->wqh), &wq);
/*设置当前状态为正在执行状态*/
set_current_state(TASK_RUNNING);
//-----
/*唤醒等待队列头指向链表中的等待队列*/
```

```
wake_up_interruptible(&pcdevp->wqh);
```

方法二：

●睡眠过程：

(1)定义并初始化等待队列头

```
DECLARE_WAIT_QUEUE_HEAD(name)
```

(2)调用 `wait_event_interruptible(wq, condition)` 函数,它能够自动完成方法一的 3~6 步。

//满足条件就返回,不满足条件就睡眠

eg: `wait_event_interruptible(pcdevp->wqh, pcdevp->led!=0);`

●其它程序唤醒某个进程的过程：

通过 `wake_up` 方式在其他进程中唤醒睡眠的进程

●在调用 `wake_up_interruptible()`之前,一定要满足 `condition` 条件,才能唤醒进程

二、多路监听侦测

1. 用户空间操作

●监听函数定义

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

//nfds: 监听文件描述符的最大值加 1

//readfds: 要监听的读文件描述符集合

//writefds: 要监听的写文件描述符集合

//exceptfds: 要监听的异常文件描述符集合

//timeout: 监听超时时间

//成功返回监听的个数 (大于 0), 失败返回 `errno` (小于 0), 超时返回 0

只要有一个文件描述符发数据了 (读、写), 该函数就能正常返回

●对文件描述符的操作

`FD_SET()`: respectively add a given file descriptor from a set

`FD_ISSET()`: tests to see if a file descriptor is part of the set; this is useful after `select()` returns

`FD_CLR()`: respectively remove a given file descriptor from a set

`FD_ZERO()`: clears a set

2. 内核空间操作

●用户空间对应的 `select` 对应的是内核空间的 `poll`

```
static unsigned int chardevicedriver_poll(struct file *filp, struct poll_table_struct *wait)
```

```
{
```

```
    struct chardevicedriver_cdev *pcdevp = filp->private_data;
```

```
    unsigned int mask = 0;
```

```
    /*第一步*/
```

```
    poll_wait(filp, &pcdevp->wqh, wait);
```

```
    /*第二步判断是否可读*/
```

```
    if(pcdevp->led)
```

```

        mask |= POLLIN | POLLRDNORM;
    return mask;
}

```

三、IO 与内存

1. X86 汇编中

- MOV: 内存空间数据访问, IN/OUT: 对外设操作
- 从 0x0 到 0xFFFFFFFF, 既有可能是内存, 也有可能是外设, 通过 MOV 和 IN/OUT 区别
- X86 中, 内存和 IO 独立编程, 存在两套地址, 对地址访问时要靠指令进程区分

2. ARM/powerpc/MIPS 汇编中

- 内存与 IO 统一编址, 只存在一套地址
- Linux 编程中使用到的都是虚拟地址, 驱动开发时, 从芯片手册得到的都是物理地址, 需要转换成虚拟地址再使用

3. 虚拟地址使用过程

(1)申请 IO 内存 (建议)

```

request_mem_region(start, n, name)
//start: 待申请的起始地址 (物理地址)
//n: 从起始地址开始的字节数
//name: 名字

```

(2)映射

```

ioremap(unsigned long physaddr, unsigned long size)
//physaddr: 物理地址
//size: 映射的直接数
//成功返回映射后的虚拟地址, 假设放到 vir_addr 变量里面
eg: volatile long __iomem *vir_addr= (volatile unsigned long *)ioremap(0xe0200080, 4);

```

(3)IO 内存操作函数

方法一: 调用内核函数

```

ioread8/16/32
iowrite8/16/32

```

方法二: 直接赋值

```

*vir_addr=0x20;

```

(4)取消映射

```

iounmap(vir_addr)

```

(5)释放内存

```

release_mem_region(start, n)

```


DAY22—中断、mmap

一、中断

1. 中断定义

- 中断是指 CPU 在执行过程中，出现了某些突变事件时，CPU 必须暂停执行当前的程序，转去处理突变事件，处理完毕后 CPU 又返回原程序被中断的位置并继续执行。
- 中断服务程序要尽量短，Linux 内核在处理中断事件的时候可以分两步：紧急事件放在顶部处理；不紧急或者耗时比较长的放在底半部处理
- 中断处理程序的上下文中一般不能出现睡眠
- 底半部执行的机制：底半部是系统自动执行
 - (1)软中断
 - (2)tasklet
 - (3)工作队列

2. 按键中断

- 中断注册函数：request_irq(unsigned int irq, irq_handler_t handler, unsigned long irqflags, const char * devname, void * dev_id)

//irq: 待注册的中断号，在/* linux/arch/arm/mach-s5pv210/include/mach/irqs.h 中

//handler: 顶部处理函数->注册的中断处理程序，紧急事件放到该函数中

eg: irqreturn_t fun(int irq, void * data)

//irqflags: 中断标志

IRQF_TRIGGER_NONE	//
IRQF_TRIGGER_RISING	//上升沿触发中断
IRQF_TRIGGER_FALLING	//下降沿触发中断
IRQF_TRIGGER_HIGH	//高电平触发中断
IRQF_TRIGGER_LOW	//低电平触发中断
IRQF_SHARED	//表示多个设备共享中断（例如一个管脚接多个设备）

//devname: 中断设备名称

//dev_id: 调用 handler 时传递的参数

//返回 0 表示成功，大于 0 表示失败

eg: request_irq(IRQ_EINT0, chardevicedriver_irq,IRQF_TRIGGER_RISING|IRQF_TRIGGER_FALLING, "irq_led", chardevicedriver_cdevp);

- 中断注销函数：free_irq(unsigned int irq, void * dev_id)

// dev_id: 这个参数一定要和对应的 request_irq 中的 dev_id 一样

- 实验时，取消掉内核自带的 led 启动

-> Device Drivers

-> Input device support
-> Keyboards
-> s3c gpio keyboard support

- 查看系统中断：cat /proc/interrupts
第一排是中断号 第二排是中断数字

3. tasklet 底半部中断

- tasklet 用到的是软中断机制，工作于中断上下文
- 定义 tasklet
DECLARE_TASKLET(name, func, data)
//name: 变量名
//func: 对应的处理函数，耗时且不太紧要的工作放到该函数执行
eg: void demo_func(unsigned long data)
//data: 调用 fun 函数传递的参数
- 登记底半部
tasklet_schedule(name)
//这个 name 就是 DECLARE_TASKLET 中的 name，要一一对应

4. 工作队列

- 工作队列是 Linux 内核中将工作推后执行的一种机制；返种机制和 Tasklet 不同之处在于工作队列是把推后的工作交由一个内核线程去执行，它工作于内核上下文，因此工作队列的优势就在于它允许重新调度甚至睡眠，内核链表在轮询到这个 work 自动时会调用
- 定义一个 work
struct work_struct my_work;
- 初始化工作
INIT_WORK(_work, _func)
func 原型：void (*work_func_t)(struct work_struct *work);
- 将初始化后的工作添加到 keventd_wq 链表中去
schedule_work(*name)
//name: 队列名的地址

二、虚拟地址和物理地址转换

1. virt_to_phys
2. phys_to_virt

三、mmap

1. 用户空间

```
void *p= void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

//addr: 指定的虚拟地址，一般为 NULL

//length: 映射的空间大小，eg: 0x1000

//prot: PROT_READ|PROT_WRITE

//flags: MAP_SHARED

//fd: 文件描述符

//offset: 一般为 0

用户空间得到的地址是 0~3G

2. 内核空间

```
#define IOMEM_GPIO_BASE 0xE0200000
```

```
#define IOMEM_GPIO_SIZE 0x00001000
```

```
int chardevicedriver_mmap(struct file *filp, struct vm_area_struct *vma)
```

```
{
```

```
    int ret = 0;
```

```
//-----
```

```
    //vm_start:映射到进程空间的起始地址
```

```
    //vm_end: 映射到进程空间的结束地址
```

```
    unsigned long vmasize = vma->vm_end - vma->vm_start;
```

```
//-----
```

```
    //用户空间 mmap 传入的 prot 参数
```

```
    //禁止缓存
```

```
    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
```

```
//-----
```

```
    //vm_flag 来源于用户空间 mmap 出入的参数 flags
```

```
    //VM_IO:标志该区域为 I/O 内存
```

```
    //VM_RESERVED:将此虚拟区域锁住，防止系统将该区域交换出去
```

```
    vma->vm_flags |= VM_IO | VM_RESERVED;
```

```
//-----
```

```
    //作用:一次性创建页表
```

```
    //addr,起始的虚拟地址
```

```
    //pfn,对应物理地址所在的页号
```

```
    ret =remap_pfn_range(vma, vma->vm_start, (IOMEM_GPIO_BASE>>PAGE_SHIFT),  
                        IOMEM_GPIO_SIZE, vma->vm_page_prot);
```

```
//-----
```

```
    if(ret)
```

```
        return -EAGAIN;
```

```
    return 0;
```

```
}
```

DAY24—字符设备编程架构

一、input 子系统

1. 什么是 input 子系统

- 输入设备（如按键、键盘、触摸屏、鼠标等）是典型的字符设备，一般的工作机制是：首先在按键、触摸等动作变生时产生一个中断（或驱动通过 timer 定时查询），然后 CPU 通过特定的接口（如 SPI、I2C、USB 等总线）读取键值、坐标等数据，放入一个由字符设备驱动管理的缓冲区，驱动的 read()接口让用户可以读取键值、坐标等数据。
- 上述设备中重复的代码在 Linux 内核中已经实现好了框架，这些实现好的框架就称之为 input 子系统，input 子系统只有输入，没有输出，硬件设备驱动都可以归于 input 驱动。
- 在 Linux 系统中，input 输入子系统由输入子系统设备驱动局(Device Driver)、输入子系统核心局(Input Core)、输入子系统事件处理局(Event Handler)组成。

2. 驱动实现步骤

(1)分配一个输入设备空间

```
struct input_dev *input_allocate_device(void)
struct input_dev
{
    .evbit[]: 该设备会产生哪些事件
}
eg: struct input_dev *buttons_dev= input_allocate_device();
```

(2)设置分配得到的输入设备

①设置该设备能产生哪几类事件

#define EV_SYN	0x00 // 同步事件
#define EV_KEY	0x01 // 按键（键盘或按钮）
#define EV_REL	0x02 // 相对坐标（鼠标）
#define EV_ABS	0x03 // 绝对坐标（触摸屏等）
#define EV_MSC	0x04 // 其它
#define EV_SW	0x05
#define EV_LED	0x11 // LED 等指示设备
#define EV_SND	0x12 // 声音（如：蜂鸣器）
#define EV_REP	0x14 // 重复
#define EV_FF	0x15 // 力反馈
#define EV_PWR	0x16
#define EV_FF_STATUS	0x17 // 力反馈状态
#define EV_MAX	0x1f

eg:

● 按键事件 (EV_KEY): `set_bit(EV_KEY, buttons_dev->evbit);`

● 同步事件 (EV_SYN): `set_bit(EV_SYN, buttons_dev->evbit);`

② 设置按键时 (EV_KEY), 产生的键值

`set_bit(KEY_L, buttons_dev->keybit);`

③ 设置设备名称

`buttons_dev->name="buttons_test"`

`//memcpy(buttons_dev->name,"buttons_test");`写法错误, 因为现在的指针为空指针,

不能 `memcpy`; 如果执行 `buttons_dev->name="buttons_test"` 后再执行 `memcpy()`, 也是错的, 因为现在指针指向的是代码段, 不能赋值

(3) 注册输入设备

`int input_register_device(struct input_dev *dev)`

(4) 硬件相关的操作 (例如中断设置、管脚设置、定时器设置等)

(5) 报告事件

`void input_event(struct input_dev *dev,
 unsigned int type, unsigned int code, int value)`

`// dev:` 那个输入设备

`// type:` 报告的事件类型

`// code:` 报告的事件编码值

`// value:` 按键值, 0 表示释放, 1 表示按下, 可以通过 `gpio_get_val` 的返回值进行判断

注意: 报告事件需要报告 2 样以上, 分别是按键事件 (或者触摸屏事件) 和同步事件, 同步事件可以理解为报告完毕的意思, 对于同步事件, `code` 和 `value` 为 0。

(6) 注销设备

`input_unregister_device(struct input_dev *dev)`

(7) 释放输入设备

`void input_free_device(struct input_dev *dev)`

(8) 注意: 同步事件即需要注册, 也需做报告事件

3. 查看设备节点

`hexdump /dev/event3`

	秒	毫秒	type	code	value	
00000000	0015 0000	e6f4 0005	0001	0026	0001 0000	// EV_KEY
00000010	0015 0000	e6fe 0005	0000	0000	0000 0000	//EV_SYN
00000020	0015 0000	67d8 0008	0001	0026	0000 0000	// EV_KEY
00000030	0015 0000	67dd 0008	0000	0000	0000 0000	//EV_SYN

4. 测试程序实现步骤

二、Linux 中的 input 子系统架构

1. input core 对应 input.c 文件

input_register_device

- >ist_add_tail(&dev->node, &input_dev_list): 把设备加入 input 设备链表
- >list_for_each_entry(handler, &input_handler_list, node): //遍历内核已经定义的 input_handler 链表, 在这链表中存在所有的操作(按键、触摸屏等), 内核注册 input_handler 的函数是 input_register_handler
- >input_attach_handler(dev, handler)
 - >input_match_device //比较注册的 device 事件和内核(evdev.c)中注册的事件是否相同, 相同执行下一步
 - >handler->connect(handler, dev, id): 会初始化、创建设备, 建立设备节点

2. 当用户空间 open("/dev/event*", "RDWR")

->sys_open

->input_fops.open

- >handler = input_table[iminor(inode) >> 5]: evdev.c 中的 handler
- >new_fops=fops_get(handler->fops): evdev.中 handler 对应的函数操作集合 evdev_fops
- >file->f_op = new_fops: 替换设备操作函数集合, 由 input_fops 替换为 evdev_fops
- >new_fops->open(inode, file): evdev_fops.open

3. 当用户空间 read 时, 实际调用的是 handler->file_operations->read, 阻塞 xxx_read 唤醒 xxx_event 的执行过程

input_event (...)

- >input_handle_event(...)
 - >input_pass_event(...)
 - >handler->event(...)
 - =>evdev_handler.event
 - =>evdev_event
 - {
 - wake_up_interruptible}

DAY24—平台设备驱动

一、设备挂载到 CPU 上去的形式

- GPIO ->按键、LED
- ram-like(使用地址总线、数据总线和 CPU 连接) ->nand
- 协议类接口 ->串口、I2C、SPI、CAN、PCI、RS485、USB

二、总线

1. 总线数据结构

```
struct bus_type                                //定义在<linux/device.h>中
{
    const char          *name;                // 总线名称
    int (*match)(struct device *dev, struct device_driver *drv); //匹配和该总线关联的设备和设备驱动
    struct bus_attribute *bus_attrs;           // 总线属性
    struct device_attribute * dev_attrs;       // 设备属性
    struct driver_attribute * drv_attrs;       // 驱动属性
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env); //该函数允许总线在内核为用户空间产生热插拔事件之前为系统添加环境变量
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    .....
};
```

2. 总线操作函数

- 实现在内核源代码 drivers/base/bus.c
- 总线注册: bus_register(struct bus_type * bus)
- 总线注销: bus_unregister(struct bus_type * bus)
- 系统总线目录: /sys/bus/总线; 进入总线后可以看见该总线挂载的设备

三、设备

1. 设备数据结构

- struct device //定义在<linux/device.h>中

```

{
    struct bus_type *bus;           //该设备挂载到那种类型的总线上去
    struct device_driver *driver;    //管理该设备的驱动
    char bus_id[BUS_ID_SIZE];       //总线上唯一标识该设备的字符串
    void *platform_data;            //Platform specific data, device core doesn't touch it
    void (*release)(struct device *dev);
}

```

- 每个 struct device 对应一个具体的硬件设备(芯片)
- 软件上表示硬件上的设备(芯片)和具体的总线连接
device.bus=bus_type("i2c/spi/usb/...")

2. 设备操作函数

- 实现在内核源代码 drivers/base/core.c 中
- 设备初始化: void device_initialize(struct device *dev);
- 设备的注册: device_register(struct device * dev)
- 设备的注销: device_unregister(struct device * dev)

四、驱动

1. 驱动数据结构

```

struct device_driver           //定义在<linux/device.h>中
{
    const char *name;          //驱动名称
    struct bus_type *bus;      //驱动所属总线
    struct module *owner;      //驱动模块拥有者
    int (*probe)(struct device *dev); //设备侦测函数
    int (*remove)(struct device *dev); //设备移除函数
    int (*shutdown)(struct device *dev); // 设备关闭函数
    int (*suspend)(struct device *dev, pm_message_t state); //设备挂起函数，用来降低功耗
    int (*resume)(struct device *dev); //设备恢复函数
    .....
}

```

2. 驱动操作函数

- 实现在内核源代码 drivers/base/driver.c 中
- 驱动注册: int driver_register(struct device_driver *drv);
- 驱动注销: void driver_unregister(struct device_driver *drv);

五、平台设备总线

平台总线是一条虚拟总线，内核已经完成总线的注册，开发人员只需要写设备和驱动的程序

1. 内核平台设备总线的注册

`device_register(&platform_bus)` //存在于/driver/base/platform.c

2. 平台设备

(1)平台设备数据结构

```
struct platform_device
{
    const char* name;    //设备的名称，需要和 device_driver 中的 name 进程对比
    int id;              //设备 id，一般为-1
    struct device dev
    {
        .platform_data    //设备的数据结构
    }
    u32 num_resources;    //资源个数，一般为 ARRAY_SIZE(resource)
    struct resource* resource; //该元素存入了最为重要的设备资源信息，定义在
                               kernel/include/linux/ioport.h
    .....
};
```

(2)设备资源数据结构

```
struct resource
{
    resource_size_t start;    //资源的开始值
    resource_size_t end;     //资源结束值
    unsigned long flags;     //flags 长为 IORESOURCE_IRQ 和 IORESOURCE_MEM
    .....
};
```

如当 flags 为 IORESOURCE_MEM 时，start、end 分别表示该 platform_device 占据的内存的开始地址和结束地址；当 flags 为 IORESOURCE_IRQ 时，start、end 分别表示该 platform_device 使用的中断号的开始值和结束值，如果只使用了 1 个中断号，开始和结束值相同

(3)设备和平台总线的连接函数

`platform_device_register(struct platform_device * pdev)`

这个函数的主要功能：

- 初始化设备
- 把设备添加到平台设备总线的设备链表中去

3. 设备驱动

(1)设备驱动数据结构

```
struct platform_driver
{
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);    //一般为__devexit_p(函数)
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver
    {
        .name        //需要和 platform_device 中的 name 进行匹配
        .owner        //一般为 THIS_MODULE
    }
    const struct platform_device_id *id_table;
};
```

(2)设备驱动和平台总线的连接函数

`int platform_driver_register(struct platform_driver *drv)`

这个函数的主要功能:

- 把驱动添加到平台设备总线的驱动链表中去
- 注册驱动以后，设备在哪儿了，查看 `platform_driver` 中的 `name`，这个 `name` 就是设备内核自带的按键驱动 `device_driver` 在 `s3c-gpio-key.c` 中，`device` 在 `mach-cw210.c`

4. match

在添加设备的时候，`platform` 会让设备和平台总线中的驱动链表中的节点进行比较（调用到 `platform_match` 这个函数，他会比较设备的名字和驱动中的名字是否相同），比较成功的化会调用 `driver` 中的 `prob` 函数

5. probe 函数传递的值

(1)从 `platform_device_register` 角度看

`platform_device_register`

```
->device_initialize
->platform_device_add
    ->device_add
        ->bus_probe_device
            ->device_attach
                ->bus_for_each_drv
                    ->__device_attach
                        ->platform_drv_probe
                            ->button_probe
```

(2)从 platform_driver_register 角度看

platform_driver_register

->driver_register

->bus_add_driver

->driver_attach

->bus_for_each_dev

->__driver_attach

->driver_probe_device

->platform_drv_probe

->button_probe

DAY25—网卡驱动

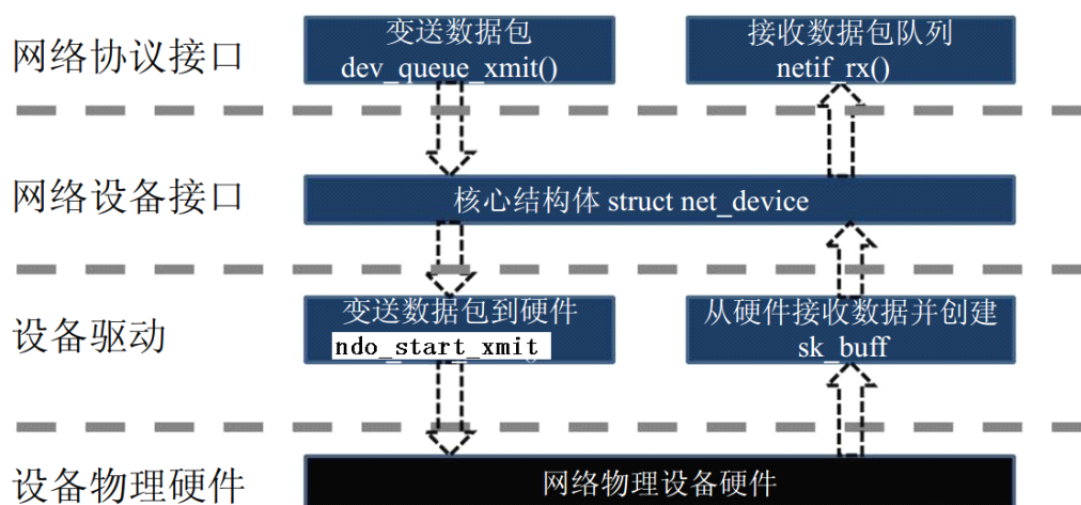
一、Linux 网络子系统的框架结构

1. 网络协议栈

OSI七层网络模型	Linux四层概念模型	对应网络协议
应用层	应用层	TFTP, FTP, NFS, WAIS
表示层		Telnet, Rlogin, SNMP, Gopher
会话层		SMTP, DNS
传输层	传输层	TCP, UDP
网络层	网际层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层	网络接口	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层		IEEE 802.1A/802.2

我们自己写的网卡驱动工作于网络接口层

2. Linux 网络设备驱动框架



3. 核心数据结构 **struct net_device**

```
●struct net_device //定义在 linux/netdevice.h
{
    char name[IFNAMSIZ]; //网络设备名称
    state; //网卡的状态
    unsigned long base_addr; //设备 I/O 基地址
    unsigned int irq; //设备占用中断号，通过中断通知网卡
    const struct net_device_ops *netdev_ops; //网卡的操作函数集合
    {
        int (*init)(struct net_device *dev); //设备初始化
        int (*open)(struct net_device *dev); //设备打开
        int (*stop)(struct net_device *dev); //设备关闭
        /*网络设备驱动开发所要实现的发送函数*/
        netdev_tx_t (*ndo_start_xmit) (struct sk_buff *skb, struct net_device *dev);
        // skb: 数据包
        //dev: 指定网卡设备
        .....
    }
    .....
}
```

●有一个网卡设备，内核中就应该有一个 `net_device` 与之对应

4. **net_device** 操作函数

●申请 `net_device` 设备:

`alloc_netdev(sizeof_priv, name, setup)`

// `sizeof_priv`: 申请空间时，出去 `net_device` 额外的空间

或者: `alloc_etherdev(sizeof_priv)`

●注册网络设备: `register_netdev(struct net_device * dev)`

●注销网络设备: `unregister_netdev(struct net_device * dev)`

●释放 `net_device` 设备: `void free_netdev(struct net_device *dev)`

5. 网卡接收数据的相关函数

●把接收到的数据发送到协议层: `int netif_rx(struct sk_buff * skb)`

This function receives a packet from a device driver and queues it for the upper (protocol) levels to process. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

●网卡接受数据是驱动工程师要完成的工作，将网卡芯片接受到的数据拷贝出来，形成 `sk_buff`，然后直接调用 `netif_rx`，到此为止驱动层的接收数据就完成了。

●对于 DM9000，首先是将 DM9000 中 RAM 空间的数据拷贝到开发板去，生成 `sk_buff`，然

后通过 `netif_rx(sk_buff)`，将数据交给设备无关接口。

6. struct sk_buff,

- `sk_buff` 是驱动层接受数据、发送数据的基本数据结构它包含了协议、实际发送接受的数据、数据头等
- C 编程时，动态申请空间的原则：谁申请谁释放；但是对于 `sk_buff` 编程时，不适合这条原则。
- 发送上层传送过来数据完成后，要释放 `sk_buff`，接收网络传来数据后，`kmalloc` 生成 `sk_buff`，传递给上层
- 申请 `sk_buff` 函数：`struct sk_buff *alloc_skb(unsigned int size, gfp_t priority);`
- 释放 `sk_buff` 函数：`struct sk_buff *dev_alloc_skb(unsigned int length);`

二、DM9000AEP 网卡驱动分析及其移植

1. 从 datasheet 得到的信息

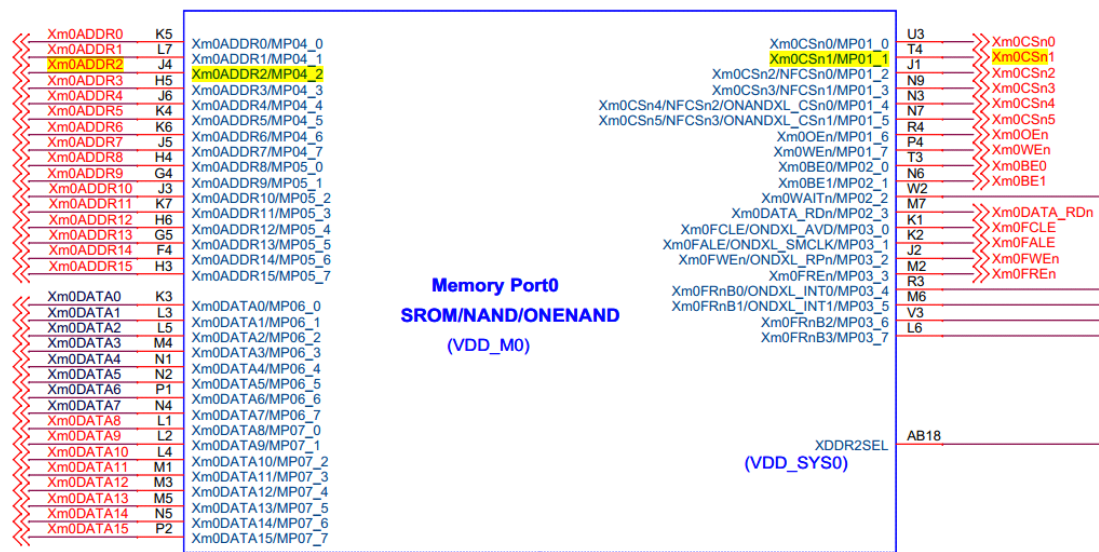
- 10/100M
- 8/16bits
- CMD 管脚：高电平为数据，低电平为索引
- CS 管脚：低电平有效
- EECK 管脚：高电平时，INT 管脚低电平有效；当它为低电平时，INT 管脚高电平有效
- EECS 管脚：高电平时，数据总线为 8bits；低电平时，数据总线为 16bits
- 内部有 16K 的 SRAM，0~3K 为发送 SRAM，3~16K 为接收 SRAM
- 如何访问 DM9000 内部寄存器？？？（第九章）
- DM9000 数据收发流程？？？（第九章）

发送：

Firstly write data to the TX SRAM using the DMA port and then write the byte count to byte_count register at index register 0fch and 0fdh. Set the bit 1 of control register. The DM9000A starts to transmit the index I packet.

接收：

2. 从硬件电路图得到的信息



- (1)CMD 连接到了地址总线 xm0ADDR2
- (2)SD0~SD15 连接到了 cpu 的数据总线是 Xm0DATA[0.. 15]
- (3)CS 连接到了 xm0csn1,也就连接到 CPU 的 bank1

nGCS[5:0]	Output	Bank selection signal	Xm0CSn[5:0]	muxed
-----------	--------	-----------------------	-------------	-------

结合 210 芯片手册，如果要选中 DM9000，那么地址总线就应该是 0x88000000 ~08FFFFFF 范围内的任意一个值，通常情况下将 0x88000000 称作 DM9000 的基地址

- (4)外部中断 IRQ_EINT10
- (5)EECS 为低电平，说明 dm9000 工作于 16bit 模式
- (6)EECK 为低电平，说明中断为高电平有效
- (7)写数据到数据总线上

- 第一步：写 TCR 寄存器(index) 一写数据 0x02 到数据总线上

地址总线的选择原则是：地址在 0x88000000 ~08FFFFFF 之间，片选信号(地址总线的第 2 为)为低电平，保证发送的是 index 而不是数据。因此我们选择的 0x88000000。

假设可以直接用物理地址操作的话，可以直接((volatile char *)0x88000000) = 0x02 ； 如果不行，就要进行物理地址和虚拟地址之间映射，然后在赋值

- 第二步：写数据(data) 一写数据 100 到数据总线上

地址总线的选择原则是：地址在 0x88000000 ~08FFFFFF 之间，片选信号(地址总线的第 2 为)为高电平，保证发送的是数据而不是 index。因此我们选择的 0x88000004。其实只要满足上面的两个要求，其它数据都可以

假设可以直接用物理地址操作的话，可以直接((volatile char *)0x88000004) = 100 ； 如果不行，就要进行物理地址和虚拟地址之间映射，然后在赋值

- (8)从数据总线上读数据

```

ior(board_info_t * db, int reg)
{
    writeb(reg, db->io_addr);
    //((volatile char *)0x88000000) = reg;
}

```

```

        return readb(db->io_data);
        //return ((volatile char *)0x8800004)
    }

```

3. 内核中自带的网卡驱动程序

(1)找到对应的驱动 DM9000.c

(2)找对应的 device

搜索 driver 的 name 字段 “dm9000”，过滤搜索结果。过滤原则：看目录是否与平台和开发板相关，例如：arm、plat-xxx(plat-s5p)、mach-xxx(mach-s5pv210)，最后找到

arch/arm/plat-s5p/devs.c

```

static struct resource s5p_dm9000_resources[] =
{
    /*向网卡发送 index(寄存器)时的地址*/
    [0] = {
        .start = S5P_PA_DM9000,
        .end   = S5P_PA_DM9000,
        .flags = IORESOURCE_MEM,
    },
    /*向网卡发送数据时的地址*/
    [1] = {
        .start = S5P_PA_DM9000 + 4,
        .end   = S5P_PA_DM9000 + 4,
        .flags = IORESOURCE_MEM,
    },
    /*DM9000 的中断管脚*/
    [2] = {
        .start = IRQ_EINT10,
        .end   = IRQ_EINT10,
        .flags = IORESOURCE_IRQ | IORESOURCE_IRQ_HIGHLEVEL,
    }
};

```

(3)发送数据

```

ndev->netdev_ops = &dm9000_netdev_ops;
dm9000_start_xmit
{
    /* Move data to DM9000 TX RAM */
    writeb(DM9000_MWCMD, db->io_addr);
    (db->outblk)(db->io_data, skb->data, skb->len);
    /*开始发送数据*/
    dm9000_send_packet(dev, skb->ip_summed, skb->len);
    {
        /* Set TX length to DM9000 */
        iow(dm, DM9000_TXPLL, pkt_len);
    }
}

```



```

        iow(dm, DM9000_TXPLH, pkt_len >> 8);
        /*开始发送指令*/
        iow(dm, DM9000_TCR, TCR_TXREQ);
    }
}

-----
iow(dm, DM9000_TCR, TCR_TXREQ);
{
    writeb(reg, db->io_addr);
    {
        *(volatile unsigned char __force *) addr = reg;
        /*((volatile char *)0x8800000) = 0x02 ;
    }
    writeb(value, db->io_data);
    {
        *(volatile unsigned char __force *) addr = reg;
        /*((volatile char *)0x8800004) = 100 ;
    }
}

(4)数据的接收
dm9000_interrupt
{
    if (int_status & ISR_PRS)
        dm9000_rx(dev);
}
dm9000_rx
{
    /* Move data from DM9000 */
    将数据由 DM9000 内部 SRAM 3K~16k 读到开发板的 SDRAM
    dev_alloc_skb
    填充 skb
    post skb
    netif_rx(skb);
}

```

平台设备架构的好处：驱动程序便于移植。只要不是和最小系统相关的硬件设备，都可以按平台设备架构完成相应的驱动程序。

DAY26—I2C 设备驱动

一、I2C 介绍

- I2C(Inter—Integrated Circuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。
- I2C 是两线式串行总线
 - SDA: 数据线，默认为高电平
 - SCL: 时钟线(提供时钟信号)
- I2C 总线最主要的优点是其简单性和有效性。由于接口直接在组件之上，因此 I2C 总线占用的空间非常小，减少了电路板的空间和芯片管脚的数量，降低了互联成本

二、通信协议

1. 信号

- start 信号: SCL 处于高电平时，SDA 产生一个下降沿
- stop 信号: SCL 处于高电平时，SDA 产生一个上升沿
- ASK 信号: SCL 处于高电平时，SDA 编程低电平状态

2. 数据传输

- 信号的读写都是站在 CPU 的角度来说的
- 在传送数据时，SDA 线上的高低电平表示传送数据的 0/1
- SCL 为周期性时钟，当 SCL 为低电平时，SDA 可以发生高低电平变化；当 SCL 为高电平时，读 SDA 上的电平；注意：数据的变换只能发生在 SCL 为低电平的时候
- I2C 通信时，设备之间是有主从关系的，通信是有主设备（CPU）发起的，从设备不会自动发起数据传输
- 当一个字节（8 位）的数据发送完之后，保持一个周期的低电平，表示 ASK 信号（收到数据）；如果不再发送数据，那么在 SCL 处于高电平时，SDA 产生一个上升沿；如果想继续发送数据，那么就在 SCL 处于低电平时，继续发送数据。

四、AT42C02

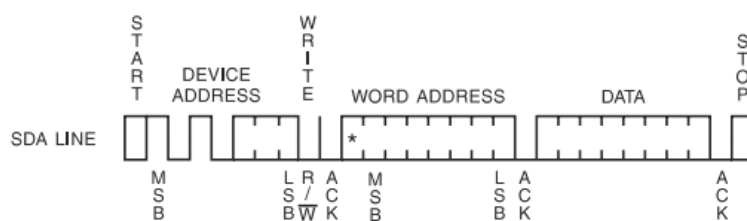
1. 地址

- A read operation is initiated if this bit is high and a write operation is initiated if this bit is low.

- The next 3 bits are the A2, A1 and A0 device address bits for the 1K/2K EEPROM. These 3 bits must compare to their corresponding hard-wired input pins

2. 写数据

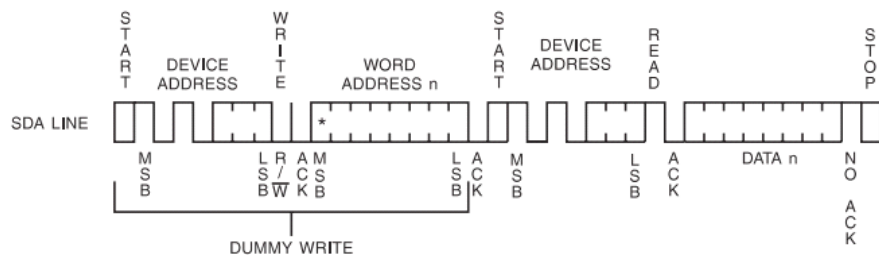
A write operation requires an 8-bit data word address following the device address word and acknowledgment. Upon receipt of this address, the EEPROM will again respond with a zero and then clock in the first 8-bit data word. Following receipt of the 8-bit data word, the EEPROM will output a zero and the addressing device must terminate the write sequence with a stop condition. All inputs are disabled during this write cycle and the EEPROM will not respond until the write is complete.



- 产生一个启动信号
- 发送从设备地址+写信号 (10100000)
- 等待 ACK
- 发送即将写入从设备内的偏移地址 (0~0xff)
- 等待 ACK
- 发送即将写入从设备指定偏移地址处的数据
- 等待 ACK
- 如果数据没有发送完，继续发送数据，等待 ACK，直到数据发完
- 发送停止信号

3. 读数据

Once the device address word and data word address are clocked in and acknowledged by the EEPROM, the microcontroller must generate another start condition. The microcontroller now initiates a current address read by sending a device address with the read/write select bit high. The EEPROM acknowledges the device address and serially clocks out the data word. The microcontroller does not respond with a zero but does generate a following stop condition.



(* = DON'T CARE bit for 1K)

- 产生一个 start 信号
- 发送从设备地址+写信号
- 等待 ACK
- 发送从设备读取设备的偏移地址（0~0xff）
- 等待 ACK
- 产生一个 start 信号
- 发送从设备地址+读信号（10100001）
- 等待 ACK
- 接收 SDA 上的数据
- 如果还需要继续接收数据，则 CPU 向从设备发送 ACK；如果不再需要读数据，则向从设备发送停在信号

五、Linux I2C 框架

- I2C 有一个核心模块 i2c-core.c
- 每一个 adapter 对应一组 I2C 接口，adapter 内部有相应的读写函数
- s5pv210 一共有 3 条 I2C 总线，AT42C02 接到了 s5pv210 的第一条 I2C 总线上，相应的 adapter 就是 adapter0
- 我们需要做的就是写 I2c_client 和 I2c_driver，系统会自动挂载这两个链表到 i2c-core.c 上

1. 向内核添加 client

```
vi arch/arm/mach-s5pv210/mach-cw210.c
static struct i2c_board_info i2c_at24cxx[] __initdata =
{
    {
        I2C_BOARD_INFO("at24cxx", 0x50)
        //参数一: client 的名称
        //参数二: client 对应的从设备地址
    },
    .platform_data = &pca9555_data,
};
i2c_register_board_info(0, i2c_at24cxx, ARRAY_SIZE(i2c_at24cxx));
```

(1) `struct i2c_board_info`

```
{
    char    type[I2C_NAME_SIZE];    // chip type, to initialize i2c_client.name
    unsigned short    flags;        // to initialize i2c_client.flags
    unsigned short    addr;        // stored in i2c_client.addr
    void    *platform_data;        // stored in i2c_client.dev.platform_data
    struct dev_archdata    *archdata;    // copied into i2c_client.dev.archdata
    int    irq;                    // stored in i2c_client.irq
};
```

I2C doesn't actually support hardware probing, although controllers and devices may be able to use I2C_SMBUS_QUICK to tell whether or not there's a device at a given address. Drivers commonly need more information than that, such as chip type, configuration, associated IRQ, and so on.

`i2c_board_info` is used to build tables of information listing I2C devices

- * that are present. This information is used to grow the driver model tree.

- * For mainboards this is done statically using `i2c_register_board_info()`;

- * bus numbers identify adapters that aren't yet available. For add-on boards,

- * `i2c_new_device()` does this dynamically with the adapter already known.

(2) `define I2C_BOARD_INFO(dev_type, dev_addr) \`

```
.type = dev_type,        //identifies the device type
.addr = (dev_addr)       //the device's address on the bus
```

- * This macro initializes essential fields of a `struct i2c_board_info`,

- * declaring what has been provided on a particular board. Optional

- * fields (such as associated irq, or device-specific `platform_data`)

- * are provided using conventional syntax.

(3) `i2c_register_board_info(`

```
int busnum,                //identifies the bus to which these devices belong
struct i2c_board_info const *info, //vector of i2c device descriptors
unsigned len)              //how many descriptors in the vector; may
                           //be zero to reserve the specified bus number.
```

Systems using the Linux I2C driver stack can **declare tables of board info while they initialize**. This should be done in board-specific init code near `arch_initcall()` time, or equivalent, before any I2C adapter driver is registered. For example, mainboard init code could define several devices, as could the init code for each daughtercard in a board stack.

The I2C devices will be created later, after the adapter for the relevant bus has been registered. After that moment, standard driver model tools are used to bind "new style" I2C drivers to the devices. The bus number for any device declared using this routine is not available for dynamic allocation.

The board info passed can safely be `__initdata`, but be careful of embedded pointers (for `platform_data`, functions, etc) since that won't be copied.

(4) 向内核注册 `i2c_client` 的过程


`i2c_register_board_info`

-> `list_add_tail(&devinfo->list, &__i2c_board_list);`

```

-> i2c_register_board_info
    -> i2c_new_device
        -> i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info)
            {
                client->adapter = adap;
                client->dev.platform_data = info->platform_data;
                client->dev.archdata = *info->archdata;
                client->flags = info->flags;
                client->addr = info->addr;
                client->irq = info->irq;
            }

```



(5)i2c_client 的生成过程

<-i2c_new_device: adapter(适配器)的由来

```

<-i2c_scan_static_board_info
    <-i2c_register_adapter
        <-i2c_add_numbered_adapter
            <-s3c24xx_i2c_probe
                {
                    platform_get_resource(pdev, IORESOURCE_MEM, 0)
                    request_mem_region(res->start, resource_size(res), pdev->name);
                    ioremap(res->start, resource_size(res));
                    i2c_add_numbered_adapter
                }
            <- platform_driver_register(&s3c24xx_i2c_driver);
                <- i2c-sc2410.c 已经被编译进内核，系统开机自动加载

```

(6)根据 s3c24xx_i2c_driver 中的 "s3c-i2c",可以找到 platform_device(在 plat-samsung/i2c-dev*)

```

static struct resource s3c_i2c_resource[] =
{
    [0] = {
        .start = S3C_PA_IIC,                (0xE1800000)
        .end   = S3C_PA_IIC + 0x00001000 - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = IRQ_IIC,
        .end   = IRQ_IIC,
        .flags = IORESOURCE_IRQ,
    },
};

```

```

struct platform_device s3c_device_i2c0 =
{
    .name      = "s3c2410-i2c",

```

```

        .id = 0,
        .num_resources = ARRAY_SIZE(s3c_i2c_resource),
        .resource = s3c_i2c_resource,
    };

```

2. 编写 I2C driver

(1)添加删除 I2C 驱动

```

i2c_add_driver(&at24cxx_driver);
i2c_del_driver(&at24cxx_driver);

```

(2)编写 i2c_driver 结构体

```

static struct i2c_driver at24cxx_driver=
{
    .driver= {
        .name="at24cxx",
        .owner=THIS_MODULE,
    },
    .probe=at24cxx_probe,
    .remove=__devexit_p(at24cxx_remove),
    /*I2Cmach 时会调用*/
    .id_table=at24cxx_id,
};

```

(3)编写 probe 函数

```

static int at24cxx_probe (struct i2c_client *client,const struct i2c_device_id
*id)
{
    int ret;
    /*动态注册设备号*/
    ret = alloc_chrdev_region(&dev,0,1,"at24cxx");
    /*申请 cdev 空间*/
    at24cxx_devp=kmalloc(sizeof(struct at24cxx_dev),GFP_KERNEL);
    memset(at24cxx_devp,0,sizeof(struct at24cxx_dev));
    /*配置*/
    at24cxx_devp->client =client;
    /*添加 cdev 到内核*/
    cdev_init(&at24cxx_devp->cdev, &at24cxx_fops);
    cdev_add(&at24cxx_devp->cdev, dev, 1);
    /*创建设备节点文件*/
    dev_class=class_create(THIS_MODULE,"at24cxx_class");
    dev_device=device_create(dev_class,NULL, dev, NULL,"at24cxx0" );
    return 0;
}

```

(4)编写 i2c_device_id 结构体

```

static const struct i2c_device_id at24cxx_id[] =

```

```

{
    {"at24cxx", 0},
    {},          //结束标志
};

```

(5)编写 file_operations 中的读写函数

```

static ssize_t at24cxx_read (struct file *filp, char __user *buf, size_t count, loff_t *offset)
{
    unsigned char address;
    unsigned char data;
    struct i2c_msg msg[2];
    int ret;
    ret=copy_from_user(&address,buf,1);
    //-----
    msg[0].addr=at24cxx_devp->client->addr;
    msg[0].buf=&address;
    msg[0].len=1;
    msg[0].flags=0;          //0 表示写入数据
    i2c_transfer(at24cxx_devp->client->adapter, &msg[0],1);
    //-----
    msg[1].addr=at24cxx_devp->client->addr;
    msg[1].buf=&data;
    msg[1].len=1;
    msg[1].flags=1;          //1 表示读出数据
    i2c_transfer(at24cxx_devp->client->adapter, &msg[1],1);
    //-----
    ret=copy_to_user(buf, &data, 1);
    return ret;
}

static ssize_t at24cxx_write (struct file *filp, const char __user* buf, size_t count, loff_t *offset)
{
    struct i2c_msg msg[1];
    unsigned char val[2];
    int ret=0;
    ret=copy_from_user(val, buf,2);
    printk("kernel print:write val[0]=%d,write val[1]=%d\n",val[0],val[1]);
    msg[0].addr=at24cxx_devp->client->addr;
    msg[0].buf=val;          //偏移地址+ 写入的值
    msg[0].len=2;            //写入数据长度
    msg[0].flags=0;          //0 表示写入设备
    /*使用 i2c 对应的控制器发送 msg*/
    ret=i2c_transfer(at24cxx_devp->client->adapter, msg,1);
    return ret;
}

```


DAY27—I2C 控制器原理和块设备

一、I2C 控制器的驱动

1. I2C 控制寄存器

- control register- I2CCON
- control/status register- I2CSTAT
- Tx/Rx data shift register- I2CDS
- address register- I2CADD

2. 主设备发送数据

- (1)配置寄存器为主设备模式
- (2)写从设备地址到 I2C ADD
- (3)写 0xf0 到 I2CSTAT
- (4)收到 ASK 会产生中断
- (5)写新数据到 I2CDS 继续发或者写 0xD0 到 I2CSTAT 停止

3. 主设备接收数据

4. S5PV210 的 I2C 控制器驱动程序

- make menuconfig
 - Location:
 - > Device Drivers
 - > I2C support (I2C [=y])
 - > I2C Hardware Bus support
- 变量: CONFIG_I2C_S3C2410
- 路径: drivers/i2c/busses/
- 内核中自带的 I2C 控制器驱动程序为 driver/i2c/busses/i2c-s3c2410.c
- 第一步 i2c_register_adapter 生成 adapter, 扫描列表, 看哪些设备挂接到该 adapter, 创建相应的 i2c_client, 同时让 i2c_client->adapter = adap;
 - s3c24xx_i2c_xfer
 - > s3c24xx_i2c_doxfer
 - {

```

s3c24xx_i2c_set_master      //设置为主端
s3c24xx_i2c_enable_irq      //使能中断
s3c24xx_i2c_message_start
{
    stat |= S3C2410_IICSTAT_TXRXEN;    //收发使能
    /*设置为主模式收或者发*/
    stat |= S3C2410_IICSTAT_MASTER_RX 或者 stat |= S3C2410_IICSTAT_MASTER_TX;
    s3c24xx_i2c_enable_ack(i2c);      //使能应答信号
}
}

```

●当 I2C 控制器接收到从设备发来的 ACK 会产生中断

注意：主机的 *i2c* 的读写速率要和外设的读写速率相适应，如果主机太快了，可以在每次读写完数据之后进行一会延迟

二、块设备

1. 块设备介绍

- 字符设备，采用字节流的方式，
- 块设备，每次读写通常一个整块的数据硬盘、SD 卡、flash
- linux 中块设备驱动最早是硬盘设备的驱动，以后出现的 sd、flash 驱动都是由硬盘设备驱动衍生出来的。
- 块设备的读写都是需要缓存机制的。
- 硬盘的容量的计算：磁盘容量 = 磁道数（柱面数） * 扇区数 * 512 * 磁头数

2. 内核中块设备驱动的框架机制

- 有一个硬盘在内核中就有一个 gendisk 结构与其对应，
- 有一个 gendisk 就有一个对应的 request_queue 结构
- 每一次 I/O 操作对应内核中一个 bio 结构
- 多个 bio 结构合并成一个 request，
- 合并后的 request 放入对应的 request_queue 中去
- 需要关注的数据结构

(1)struct gendisk //include/linux/genhd.h

```

{
    struct request_queue *queue;
}

```

(2)struct request_queue

```

{
    request_fn_prco  *request_fn;//请求处理函数
}

```

(3)struct request

```

{
    sector_t __sector;//读写设个起始扇区设备
}

```

(4)struct bio

●需要关注的操作函数

- | | |
|-------------------------|-----------------|
| (1)register_blkdev(...) | //注册块设备 |
| (2)alloc_disk(...) | //申请块设备空间 |
| (3)add_disk (...) | //向内核添加 gendisk |
| (4)set_capacity(...) | //设置磁盘容量 |
| (5)blk_rq_pos (req) | //获取该请求要操作的起始扇区 |

DAY28—LCD 驱动

一、LCD 概念

1. RGB

每个点在计算机领域由 RGB 三原色组成点的要素在屏幕上显示的每一个点就对应缓存中一个具体的数值，该数值是由 RGB 的值组成的。

2. 电子枪

在屏幕上打点，一幅图像，打出的点越多，图像越清晰

3. framebuffer

在内核分配了显存，将显存的首地址告诉了 LCD 控制器，LCD 控制器可以自动从显存读取数据，将读取到的数据按照分配好的时序传送给 LCD 屏，LCD 中的驱动器会驱动电子枪进行点灯，完成图像显示

LCD 控制器如果实现了 mmap，mmap 完成将显存地址映射到用户空间去，那么用户空间就可以直接操作显存

mmap 机制+GUI/QTE 开发嵌入式下的带图形界面的应用开发就变得非常简单了

二、用户空间测试程序

```
scream_size=fbvar.xres*fbvar.yres*(fbvar.bits_per_pixel/8);  
mmap(0,scream_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
```

三、驱动程序

1. linux 中 framebuffer 框架

make menuconfig

-> Device Drivers

-> Graphics support

-> Support for frame buffer devices

得到的变量: CONFIG_FB

路径: drivers/video

```
obj-$(CONFIG_FB) += fb.o
fb-y := fbmem.o fbmon.o fbcmap.o fbsysfs.o modedb.o fbcvt.o
```

得到的结论：fbmem.c 是 LCD framebuffer 驱动的核心代码
fbmem.c 文件中实现了注册设备，完成设备的操作函数

```
{
    .open
    .release
}
```

-> S3C Framebuffer support
得到的变量：CONFIG_FB_S3C
路径：drivers/video/samsung/
obj-y += s3c_fb.o

得到的结论：s3c_fb.c(驱动开发的主要工作)
s3c_fb_fimd6x.c(辅助文件，其中的函数供 s3c_fb.c 文件调用)

2. 驱动程序，如何写？

(1)分配一个 fb_info 结构体

```
s3c_fb_alloc_framebuffer(...)
struct fb_info /*描述控制器设备*/
{
    var
    fix
    fbops
}
struct fb_var_screeninfo
{
    xres
    yres
    bits_per_pixel
}
struct fb_fix_screeninfo
{
    smem_start /*分配到的显存的物理地址*/
    smem_len
}
```

(2)设置该结构体

```
s3c_fb_init_fbinfo
```

(3)初始化硬件

时序初始化

极性初始化

分辨率设置

.....

(4)注册 fb_info

```
s3cfb_register_framebuffer
{
    registered_fb[i] = fb_info;
}
```

●用户空间 open()

```
内核空间 fb_open (fbmem.c)
{
    int fbidx = iminor(inode);    //取得次设备号
    struct fb_info *info;
    info = registered_fb[fbidx];
    if (info->fbops->fb_open) {
        res = info->fbops->fb_open(info,1);
    }
}
```

用户空间首先调用 fbmem.c 中的相应操作函数（open），在该函数中判断 s3cfb.c 中是否实现了相应操作函数(open),如果实现了就调用 s3cfb.c 中相应的操作函数（open）,如果没有实现，则 执行 fbmem.c 中默认的操作。

四、LCD 驱动移植

1. LCD 屏幕管脚

VD0~VD23：数据管脚，传送 RGB 值

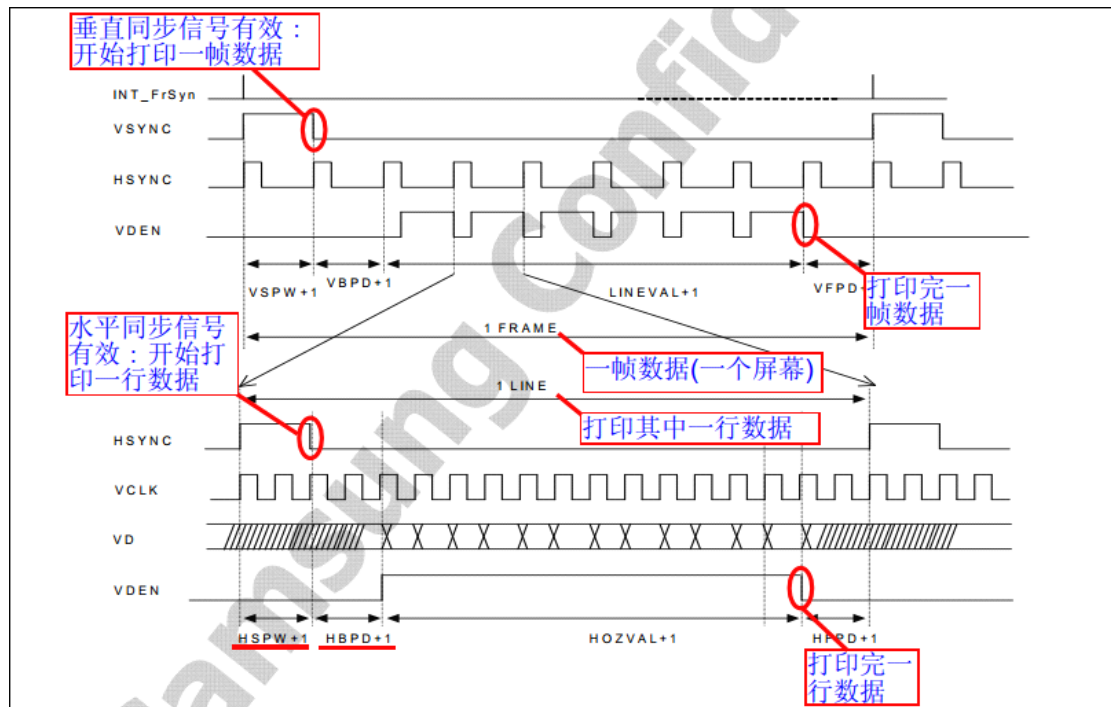
HSYNC：当这个管脚收到一个信号的时候，电子枪就由最右端调回最左端（水平同步信号）

VSYNC：这个管脚收到一个信号的时候，电子枪就由右下角调回左上脚（垂直同步信号）

VCLK：这个管脚收到一个信号的时候，由当前像素点跳向下一个像素点

VDEN：当 VDEN 有效时，才能从 VDO~VD23 取颜色（屏幕数据有效信号）

2. s5pv210 的 LCD 控制器



3. LCD 控制器的可配置参数

HSPW:水平同步脉冲宽度

HBPD: 表示从水平同步信号有效到下一行的有效数据开始的时间间隔

HFPD: 一行接收后, 到下一水平同步信号开始间隔的时间 (CLK)

VSPW: 垂直同步脉冲宽度 (HSYNC 周期)

VBPD:一帧结束后, 垂直同步信号以后的无效行数 (时间值)

VFPD:一帧结束后, 垂直同步信号以前的无效行数 (时间值)

4. 内核源码分析

```
//-----  
static struct resource s3cfb_resource[] = {  
    [0] = {  
        .start    = S5P_PA_LCD,          // (0xF8000000)  
        .end      = S5P_PA_LCD + 0x00100000 - 1,  
        .flags    = IORESOURCE_MEM,  
    },  
};  
//-----  
s3cfb_probe  
-> request_mem_region          //资源在 devs.c 中
```

```

-> ioremap
-> s3cfb_init_global
{
    -> s3cfb_set_output(ctrl);
    -> s3cfb_set_display_mode(ctrl);
    -> s3cfb_set_polarity(ctrl);        //设置极性，修改 VIDCON1 寄存器
    -> s3cfb_set_timing(ctrl);         //设置时序，修改 VIDTCON0 和 VIDTCON1 寄存器
    -> s3cfb_set_lcd_size(ctrl);       //设置分辨率(可以打多少行，一行多少点)，
    显示屏的所有参数在 mach-cw210 中
    #elif defined(CONFIG_FB_AT070TN92) / static struct s3cfb_lcd lte480wv ={.....}
}
-> s3cfb_alloc_framebuffer
-> s3cfb_init_fbinfo
-> s3cfb_map_video_memory
    -> fix->smem_start = pdata->pmem_start;
    -> fb->screen_base = ioremap_wc    //分配显存
-> s3cfb_register_framebuffer        //注册 framebuffer
-> register_framebuffer
    -> device_create(fb_class, fb_info->device, MKDEV(FB_MAJOR, i), NULL, "fb%d", i);
//-----
fbmem_init(void)
{
    proc_create("fb", 0, NULL, &fb_proc_fops);
    fb_class = class_create(THIS_MODULE, "graphics");
}

```

5. 移植 LCD 驱动时应该注意哪些问题

- (1)设置时序
- (2)设置极性
- (3)设置分辨率
- (4)分配显存：在用户空间 mmap

DAY30—单总线驱动

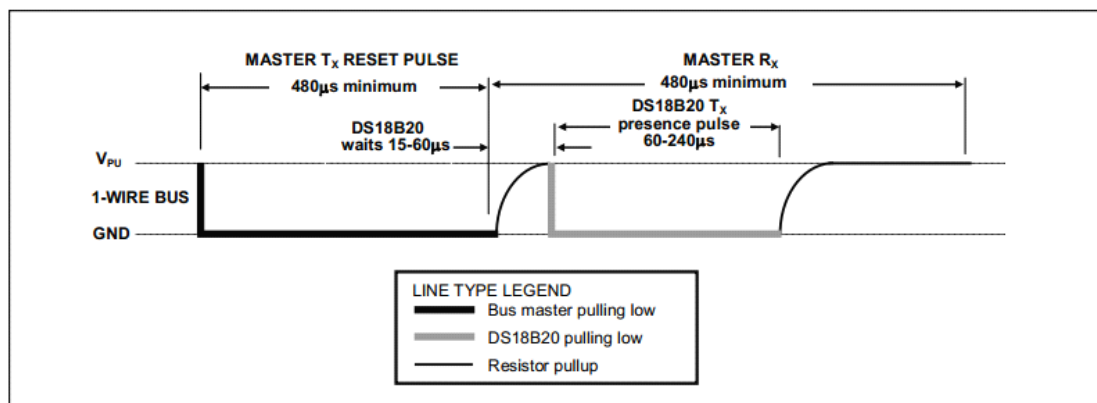
一、DS18B20 芯片资料

1. 特性

- Thermometer Resolution is User Selectable from 9 to 12 Bits.
- Each Device has a Unique 64-Bit Serial Code Stored in an On-Board ROM.
- Converts Temperature to 12-Bit Digital Word in 750ms (Max).
- The memory contains the 2-byte temperature register that stores the digital output from the temperature sensor.
- the scratchpad provides access to the 1-byte upper and lower **alarm trigger registers** (TH and TL).
- the scratchpad provides the **configuration register** allows the user to set the resolution of the temperature-to-digital conversion to 9, 10, 11, or 12 bits.

二、DS18B20 操作流程

1. 第一步：初始化操作



2. 第二步：发送 ROM COMMANDS

GPIO 设置为输出 0x44

```
0xf0 11110000
```

```
for(i=0;i<8;i++)
```

3. 第三步: function COMMANDS

读数据的时候, 先执行(1), 再执行(2)

(1) CONVERT T [44h]

This command initiates a single temperature conversion. Following the conversion, the resulting thermal data is stored in the 2-byte temperature register in the scratchpad memory and the DS18B20 returns to its low-power idle state.

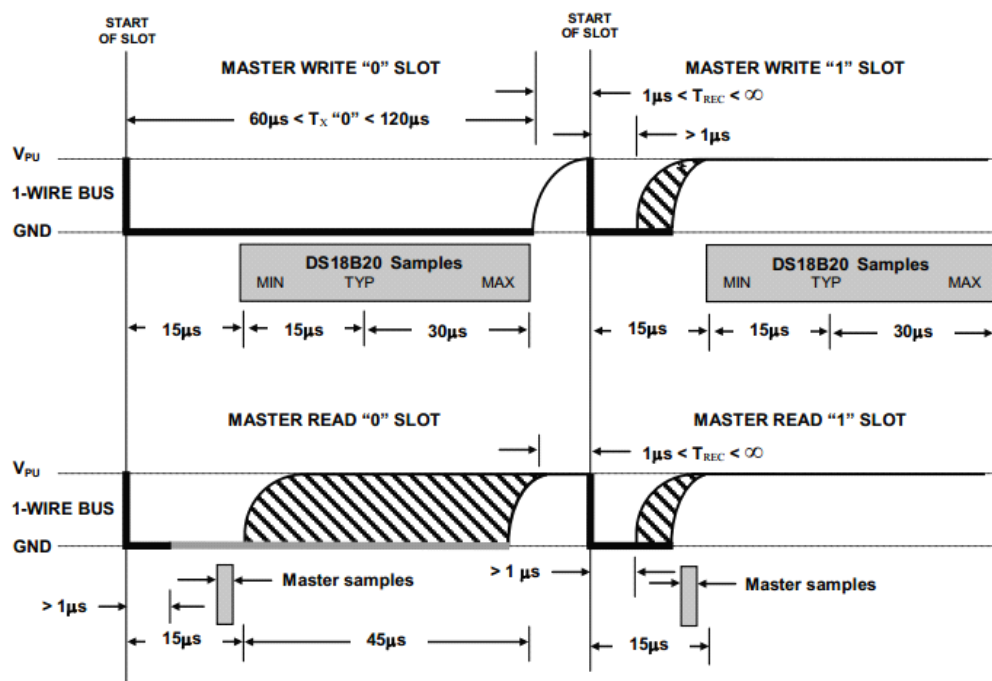
(2) READ SCRATCHPAD [BEh]

This command allows the master to read the contents of the scratchpad. The data transfer starts with the least significant bit of byte 0 and continues through the scratchpad until the 9th byte (byte 8 - CRC) is read.

(3) WRITE SCRATCHPAD [4Eh]

This command allows the master to write 3 bytes of data to the DS18B20's scratchpad. The first data byte is written into the TH register (byte 2 of the scratchpad), the second byte is written into the TL register (byte 3), and the third byte is written into the configuration register (byte 4). Data must be transmitted least significant bit first.

三、读写一个字节的时序



二、混杂设备

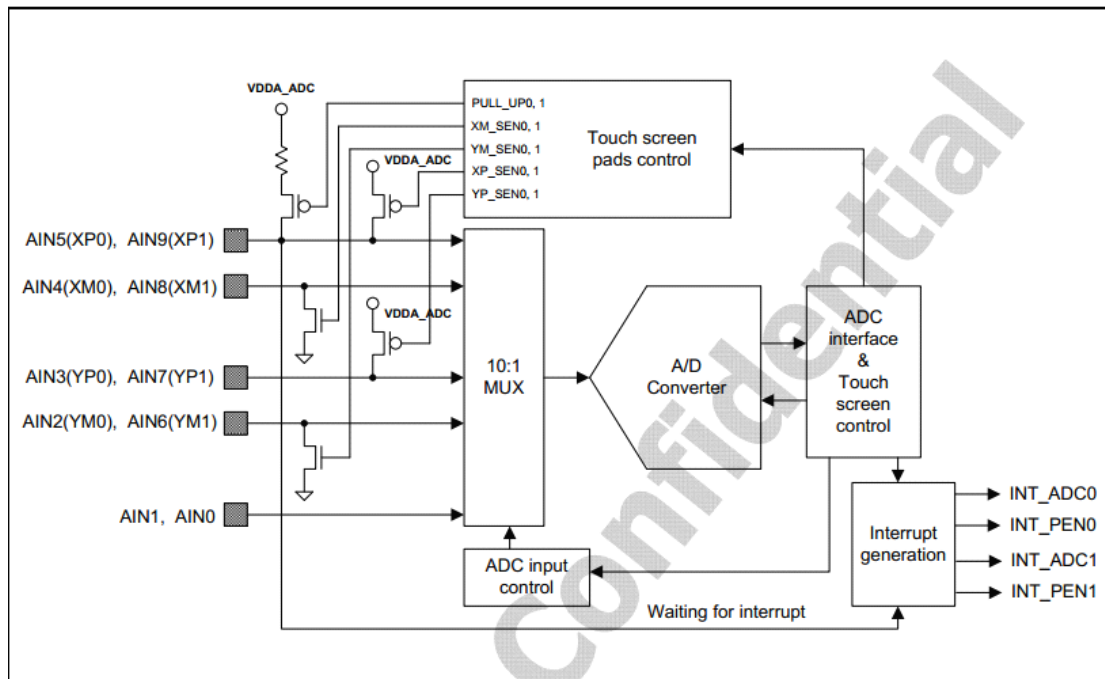
1.

```
static struct miscdevice device_driver _miscdev = {  
    .minor = DEVICE_MINOR,  
    .name  = "device_driver",  
    .fops  = & device_driver_fops,  
};  
misc_register(&device_driver _miscdev);  
misc_deregister(&device_driver _miscdev)
```
2. 硬件初始化
 - (1)iorammap
 - (2)device_driver_fops
3. `cat /proc/misc`: 可以查看系统所有的混杂设备

DAY31—ADC

一、210 ADC

1. diagram of A/D converter and Touch Screen Interface



ADC0~1 是内部中断，当他把数据处理完之后，用中断通知 CPU 来取数据
PEN0~1 是外部中断，触摸屏就是通过个管脚来通知 CPU 的

2. A/D CONVERSION TIME

- When the PCLK frequency is 66MHz and the prescaler value is 65, total 12-bit conversion time is as follows.
- A/D converter freq. = $66\text{MHz}/(65+1) = 1\text{MHz}$
- Conversion time = $1/(1\text{MHz} / 5\text{cycles}) = 1/200\text{kHz} = 5\mu\text{s}$

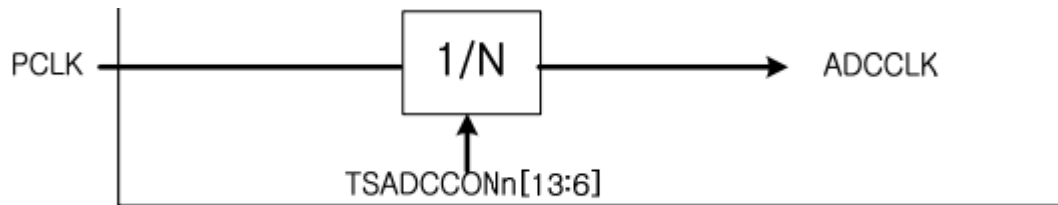
3. over view

The 10-bit or 12-bit CMOS Analog to Digital Converter (ADC) comprises of 10-channel analog inputs. It converts the analog input signal into 10-bit or 12-bit binary digital codes at a maximum conversion rate of 1MSPS with 5MHz A/D converter clock. A/D converter operates with on-chip sample-and-hold function. ADC supports low

power mode.

● Analog Input Range: 0 ~ 3.3V

4. ADC INTERFACE INPUT CLOCK DIAGRAM



5. 关注的寄存器

● TSADCCONn

如何找到 ADC 转换完成没有：中断和查询 TSADCCONn 第 15 位

● TSDATXn[14、11:0]

● CLRINTADCn：如果通过中断的方法通知 CPU，则关注这个寄存器

● ADCMUX

注意：对于触摸屏，需要注册 2 个中断，一个是笔的中断，一个是电压转中断

二、驱动

1. 注册一个混杂设备

read

打开 ADC 寄存器

读电压

使能 ADC

读数据（如果没有转完成，则睡眠，只有中断触发才能唤醒）[等待队列]

copy_to_user()

ioctl

配置分辨率

配置转换通道

配置参考电压

2. 硬件初始化

```
clk = clk_get(NULL, "adc");
```

```
//从时钟链表中,查找名字和"adc"匹配的  
时钟频率并返回 clk 结构体
```

```
clk_enable(clk);
```

```
//使能外设时钟源
```

ioremap

初始化分辨率

使能分频

设置分频系数

设置延迟时间 10000

选择 AINI

注册中断

DAY31—网络通信

一、按键驱动的方法

- 轮询
- 中断: cat /proc/interrupts 可以查看系统的中断阻塞、非阻塞
poll / select
- 异步通知、信号、signal、fcntl

二、input 子系统

- 在/dev 目录下有很多的 input, 如何确定按键驱动是哪一个了
方法一: hexdump /dev/event*, 一个一个测试, 当按下按键有显示时, 就是那个 event
方法二: cat /proc/bus/input/devices: 查看 input 子系统
- make menuconfig 中关于 input 的最重要配置
-> Device Drivers
-> Input device support
-> Event interface

三、服务器端 UDP 编程

- int inet_aton(const char *cp, struct in_addr *inp);
converts the Internet host address: cp from the IPv4(numbers-and-dots) notation into binary
把本地("*. *.*.*")的 ip 地址转为网络上传输的 ip 地址
- char *inet_ntoa(struct in_addr in); //把二进制网络地址转换为 "*. *.*.*"
- client_addr.sin_addr.s_addr=inet_addr("*. *.*.*");
- 接收到的消息类型
struct MSG_HEAD //定义的消息头, 接下来还有消息数据
{
 U16 msg_type; //消息类型
 U16 msg_length; //消息长度
 u8 trans_tpye; //传输类型
 u8 reseved[3]; //保留
};
char buf[100] = {};
recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr*)&fromaddr, &len);
struct MSG_HEAD msg_head;
memcpy(&msg_head, buf, sizeof(msg_head));

DAY32—QT 编程及其移植

一、设计图形界面

1. 创建带图形界面的项目

创建项目->QT 控件项目->QT gui 应用

2. 注册槽函数

在 mainwindow.h 中添加红色部分

```
class MainWindow : public QMainWindow{
.....
private slots:
    槽函数(声明);
.....
}
```

3. 信号和槽的连接函数

在 mainwindow.cpp 中添加 connect 函数

```
bool QObject::connect ( const QObject * sender, const char * signal, const char * method,
/*Qt::ConnectionType type = Qt::AutoConnection ) const
*sender: 那一个按键
*signal: 什么动作触发的槽函数
*method: 不懂，一般为 this
*AutoConnection: 槽函数
*/
eg: connect(ui->version_sure,SIGNAL(clicked()),this, SLOT(VersionFunction()));
```

二、QT 移植

1. 注意事项

- 搞定 LCD 驱动（framebuffer、触摸屏驱动，按键驱动）
- 触摸屏、按键驱动一定要遵循 input 子系统

2. 交叉编译 tslib

- tslib 是触摸屏的校准软件，如果需要使用触摸屏，就必须交叉编译 tslib。
- 交叉编译 tslib 的目的是获取头文件(include)和库(lib)
- 当交叉编译 QT 软件的时候会用到 tslib 中的 include 和 lib；因为在 QT 移植的时候选择了支持触摸屏，所以在编译 QT 的时候需要头文件和库
- 当 QT 软件在板卡上运行的时候，也就必须有 tslib 的动态库的支持
- 在 tslib 目录下编辑脚本文件：arm.cfg，arm.cfg 文件内容如下：

```
#!/bin/sh
export CC=arm-linux-gcc
echo "ac_cv_func_malloc_0_nonnull=yes" >arm.cache
./configure --host=arm-linux --prefix=/home/tarena/workdir/tslib/ --cache-file=arm.cache
```

3. 交叉编译 qt-embedded-linux

(1)配置

- 在 PC Linux 机上支持
./configure (--help)
- 在嵌入式 Linux 板卡上支持
./configure -help 可以查看所有编译选项的含义
- 配置的结果会产生 Makefile 和相应的头文件（config.h）
- ./configure 的一些选项说明
//--prefix 或者 CONFIG_PREFIX 都是指定的安装目录
//--depths：每一个像素占多少位，如果不知道，就写 all
//--qt-kbd-usb：支持外接 usb 键盘
//--no-feature-QWS_CURSOR：屏幕中间有个光标，不要这个选项在屏幕中间就没有光标了

(2)编译

make

(3)安装

make install

安装到指定的目录 build-qt(-prefix)，其中 include lib bin(工具 qmake)最重要

- bin/qmake:创建的工程和 Makefile 是针对 ARM 相关
- lib 目录：libQtCore.so.*、libQtGui.so.*、libQtNetwork.so.* 三个库必加上
fonts 字库一定要加上！
- 对于 include lib bin 在交叉编译 QT 的时候有用；在板卡运行的时候，只需要 lib 即可！
- 把 QT 需要的库设置到 LIBRARY_PATH 这个环境变量中

4. 增加环境变量

在板卡的/etc 目录下新建 profile 文件

export set V_ROOT=/home/tslib

export set QTDIR=/home/build-qt

```
export set QWS_DISPLAY="LinuxFB:/dev/fb0"
export set QWS_DISPLAY="LinuxFB:mmWidth130:mmHeight100:0"
export set QWS_MOUSE_PROTO="TSlib:/dev/event4 Intellimouse:/dev/mouse3"
export set QT_PLUGIN_PATH=$QTDIR/plugins/
export set QT_QWS_FONTDIR=$QTDIR/lib/fonts/
export set PATH=$QTDIR/bin:$PATH
export set LD_LIBRARY_PATH=$QTDIR/lib:$TSlibDIR/lib:$LD_LIBRARY_PATH
```

5. 编译、测试应用程序

●交叉编译你自己的 APP（以下所有的 **qmake** 都是交叉编译 **qt-embedded-linux** 生成的 **qmake**）

方法一：

(1)进入 QT 工程目录

(2)**qmake -project** (如果提示无“qmake”命令，则肯定是你的环境变量的路径设置不正确，或者 **source setenv-embedded.sh** 一下即可)

(3)**qmake** (生成 Makefile)

(4)**make clean**

(5)**make** (生成可执行程序)

方法二：

```
export PATH=/home/tarena/dlc/my_qt/myrootfs/home/build-qt/bin:$PATH
```

```
./qmake -makefile AppTest.pro
```

```
make
```

●运行应用程序

```
./app -qws
```

6. 按键支持

```
export QWS_KEYBOARD=USB:/dev/event1
```

7. 触摸屏支持

●一定有 **tslib** 的支持，移植参看文档！

●重点是 **qt-embedded-linux** 移植的时候，在 **configure** 中需要支持 **tslib** 这个选项：

```
-qt-mouse-tslib
```

```
-I /home/tarena/workdir/tslib/include
```

```
-L /home/tarena/workdir/tslib/lib
```

●添加与 **tslib** 有关的 6 个环境变量

```
echo "tslib setting"
```

```
export set TSlib_TSDEVICE=/dev/event4 //触摸屏色设备节点
```

```
export set TSlib_CONFFILE=$V_ROOT/etc/ts.conf
```

```
export set TSlib_CALIBFILE=/etc/pointercal
```

```
export set TSLIB_PLUGINDIR=$V_ROOT/lib/ts  
export set TSLIB_CONSOLEDEVICE=none  
export set TSLIB_FBDEVICE=/dev/fb0
```

8. 添加 **wenquanyi** 字库

DAY33—串口编程和 I/O 多路监听

一、遵循 **posix system** 串口编程模型

linux 串口编程详解

1. 打开串口设备

对于 Linux，设备节点是/dev/ttyS0，

对于三星处理器，设备节点是/dev/s3c2410_serial*

2. 设置串口的工作模式和属性（**struct termios**）

- 首先获取 termios 的信息
- 修改配置 termios 信息
- 回写 termios 信息

3. 读写

read/write(fd, buffer, size)

注意：

三星所有的处理器的串口设备节点都是 s3c2410_serial0~ s3c2410_serial3

opt.c_cc[VTIME] = 150; //超时时间：15 秒没有数据就返回

opt.c_cc[VMIN] = 0; //0 表示只要有数据就返回

二、I/O 多路监听

1. 读取串口和网口数据的方法

方法一：读串口，没有数据就读网口

方法二：开启两个任务分别检测网口和串口

方法三：poll/select（推荐看 unix 高级环境编程 CH14）

poll/select 机制首先要底层驱动支持

2. 测试程序

- FD_SET(): 添加一个文件描述符到集合
- FD_ISSET(): 判断文件描述符来自于那个文件

- FD_CLR(): 从集合中减去一个文件描述符
 - FD_ZERO(): 清空文件描述符
 - int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
- //nfd: 监听文件描述符的最大值加 1
- //readfds: 要监听的读文件描述符集合
- //writefds: 要监听的写文件描述符集合
- //exceptfds: 要监听的异常文件描述符集合
- //timeout: 监听超时时间
- //成功返回监听的个数 (大于 0), 失败返回 errno (小于 0), 超时返回 0

DAY33—项目总结(1)

项目部署、项目名称、硬件平台、软件平台、负责内容

一、分区规划

- 根据软件镜像的个数和大小来指定分区的个数和空间;
- 如果通过 uboot 来进行烧写,本身可以知道镜像文件的存放地址信息,这些地址本身就构造一个隐式分区;这个隐式分区一定要在 kernel 的 mtd 驱动中体现(struct mtd_partatino)只需修改 driver/mtd/nand/xxx.c 驱动即可!

二、uboot

1. 通过工具去烧写

- 方法一:通过 uboot 来烧写(包括 fastboot 协议)
- 方法二:通过命令来烧写(flash_eraseall, nandwrite...)方法二是在整个操作系统正常运行的情况进行!

2. 设置启动参数

- 启动内核: bootcmd

```
setenv bootcmd nand read 50008000 start size \; bootm 50008000
setenv bootcmd tftp c0008000 zImage \; bootm c0008000
```
- 挂接文件系统:
 - 方法一:通过 uboot 传递 bootargs
 - 方法二:通过 kernel 默认的 boot options

三、文件系统

1. 创建目录

```
mkdir dev proc sys etc tmp home mnt lib
```

2. 移植 busybox

- 配置 make menuconfig

-> Busybox Settings

-> Build Options

-> [] Build BusyBox as a static binary (no shared libs) //编译方式为静态([*])或动态([])

-> [*] Build with Large File Support (for accessing files > 2 GB)

(arm-linux-) Cross Compiler prefix

-> Busybox Library Tuning

-> Command line editing

-> Tab completion //允许 tab 按键

Linux Module Utilities

-> [*] modinfo

-> [] Simplified modutils //不需要精简的 module 命令，加上 insmod、rmmod、lsmod

-> [*] insmod

-> [*] rmmod

-> [*] lsmod

Miscellaneous Utilities

-> [*] nandwrite //支持 nandwrite 命令

-> [*] flash_eraseall //支持 flash_eraseall 命令，这两个命令会在烧写方式二中用到

●编译

如果在 Makefile 修改了 ARCH=arm CROSS_COMPILE=arm-linux-，就直接 make

或者：make ARCH=arm CROSS_COMPILE=arm-linux-

●安装

make install

3. 移植 lib 库

●方法一：先把交叉编译工具的/lib 目录下的所有库文件都拷贝过来

(1)rm -rf *.o //删除.o 文件

(2)rm -rf *.a //删除静态库

(3)arm-linux-strip /lib: //删除调试信息

●方法二：根据需求拷贝库

(1)在 busybox 目录下执行：arm-linux-readelf -a busybox | grep "Shared"

0x00000001 (NEEDED) Shared library: [libm.so.6]

0x00000001 (NEEDED) Shared library: [libc.so.6]

把交叉编译工具的/lib 下的 libm.so.6 和 libc.so.6 拷贝到跟文件系统的/lib 目录下

(2)拷贝把交叉编译工具的/lib 下的 ld-*, 到跟文件系统的/lib 目录下(ld-*是库的加载工具)

(3)在 rootfs 目下执行：arm-linux-strip /lib:

4. 添加系统启动的配置文件

u-boot->kernel->linuxrc->/sbin/init->解析

kernel_init

init_post

{

```

        run_init_process("/sbin/init");
        run_init_process("/etc/init");
        run_init_process("/bin/init");
        run_init_process("/bin/sh");
// run_init_process 如果执行成功，那么程序就有内核态转换到用户态，函数不会不会返回，
// 如果执行失败，就继续执行下面的 run_init_process
        panic("No init found. Try passing init= option to kernel. "
        "See Linux Documentation/init.txt for guidance.");
//如果打印这句话，有可能是库没找到，有可能没有找到 init
    }

```

(1)添加/etc/inittab 文件

id:runlevel:action:process

对嵌入式而言，runlevel 是不存在的

action: sysinit askfirst respawn once ... //不能随便该，除非修改 busybox

```

::sysinit:/etc/init.d/rcS //系统初始化，对应的 process 只执行一次
::askfirst: -/bin/sh // “-”表示可交互的，在高版本的 busybox 中,可加可不加
::respawn:-/bin/sh //一般用这个，不用 askfirst
::shutdown:/bin/umount -a -r //关机时调用
process: 程序

```

(2)添加/etc/init.d/rcS

```

#!/bin/sh
echo "hello world"
bin/mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
//-----
bin/mount -a: 解析/etc/fstab
mount -t devpts devpts /dev/pts: telnet 的时候会用
mdev -s: 启动守护进程
    class_create():/sys/class/leds
    device_create():/sys/class/leds/leds/dev(243:0)
    凡是向在/dev/创建设备，一般启动 mdev(嵌入式)或者 udev(PC 机)
echo /sbin/mdev > /proc/sys/kernel/hotplug: 所有的热插拔都用 mdev 来控制

```

(3)添加/etc/fstab

# device	mount-point	type	options	dump	fsck order
proc	/proc	proc	defaults	0	0
tmpfs	/tmp	tmpfs	defaults	0	0
sysfs	/sys	sysfs	defaults	0	0
tmpfs	/dev	tmpfs	defaults	0	0

四、测试

1. NFS 进行测试！

(1)一定要让 uboot 的 bootargs 或者 kernel 的默认配置启动参数要有作用！

```
setenv bootargs root=/dev/nfs nfsroot= serverip:/home/tarena/workdir/rootfs/rootfs
```

板子 ip:serverip:网关:255.255.255.0::eth0:on console=ttySAC0,115200

如果你使用 android 的 kernel，一定要把网络信息填充好！否则板子 ip=boardip:serverip:

网关:255.255.255.0::eth0:on， 可以简写 ip=boardip

针对 kernel 的默认配置选项：

```
make menuconfig
```

Boot options --->

(2)配置 NFS 网络服务！

```
/etc/exports
```

```
sudo /etc/init.d/nfs-kernel-server restart
```

```
sudo mount -t nfs -o nolock localhost/ip/lo:NFS server dir /mnt
```

(3)一定要让内核支持网络协议栈和 NFS 网络文件系统

网络协议栈的支持：

```
[*] Networking support --->
```

```
[*] Networking support --->
```

```
[*] IP: kernel level autoconfiguration
```

```
[*] IP: DHCP support
```

```
[*] IP: BOOTP support
```

```
[*] IP: RARP support
```

NFS 网络文件系统支持：

```
File systems --->
```

```
[*] Network File Systems --->
```

```
<*> NFS client support
```

```
<*> NFS server support
```

```
<*> NFS server support
```

2.采用 ramdisk 或者 initramfs 进行测试！

实现软件自启动功能！

只需在初始化脚本： /etc/init.d/rcS 添加相应的初始化过程即可@！

DAY33—项目总结(2)

一、常用文件系统

- 网络文件系统 NFS
- ramdisk: ext2, 存放在内存中, 压缩, 可读可写
- cramfs: 存放在 flash 中, 压缩, 只读
- initramfs: ext2, 运行环境在内存中, 压缩, 可读可写; initramfs 包含了 kernel+rootfs
- yaffs/yaffs2: 存放在 flash 中, 不压缩, 可读可写
 - 标准内核源码没有 yaffs2 文件系统, 我们用的内核都是三星打了补丁的, 如果下了标准的内核源代码, 需要从相关网站下载 yffs2 源码, 放置内核中, 修改 Kconfig 和 Makefile, 编译支持
 - Android 内核默认是支持 yaffs2
 - 谷歌手机早期用的文件系统都是 yaffs2 文件系统
- 最好是基本的跟文件系统做为 cramfs, 用户程序 randisk

二. ramdisk 文件系统制作

1. 分区规划

修改内核关于 flash 的驱动(/driver/mtd/nand/s3c_nand.c), 有些分区表在平台代码里

vim /include/generated/autoconfig.h	//所有需要要编译的.c 文件的配置信息
vim /include/config/auto.conf	//给 makefile 用的
vim /include/generated/mach-types.h	//程序运行以后与平台有关的所有的宏
	不能直接修改这个文件, 只能在
	arch/arm/tools/mach-tpye.h 中修改

以上 3 个文件 make zImage 时才会产生, 非常非常重要

```
//-----  
#if defined(CONFIG_ARCH_S5PV210) //这个文件可以在 autoconfig.h 中查看是否有  
struct mtd_partition s3c_partition_info[] = {  
#if 1  
    {  
        .name          = "bootloader",  
        .offset         = (0),          /* for bootloader */  
        .size           = (1024*SZ_1K),  
        .mask_flags     = MTD_CAP_NANDFLASH,  
    },  
}
```

```

{
    .name      = "param",
    .offset    = (SZ_1M),          /* for bootloader param */
    .size      = (1024*SZ_1K),
    .mask_flags = MTD_CAP_NANDFLASH,
},
{
    .name      = "kernel",
    .offset    = MTDPART_OFS_APPEND, /*继续追加*/
    .size      = (5*SZ_1M),
},
{
    .name      = "ramdisk",
    .offset    = MTDPART_OFS_APPEND,
    .size      = (5*SZ_1M),
},
{
    .name      = "system",
    .offset    = MTDPART_OFS_APPEND,
    .size      = MTDPART_SIZ_FULL, /*所有的剩余磁盘都给这个分区*/
}
#endif
};

```

2. 内核支持文件系统

第一步：内核支持该文件系统

General setup --->

☒ Initial RAM filesystem and RAM disk (initramfs/initrd) support

File systems --->

//支持 ex2、ex3 文件系统

☒ Second extended fs support

☐ Ext2 extended attributes (NEW)

☐ Ext2 execute in place support (NEW)

☒ Ext3 journalling file system support

Device Drivers --->

☒ Block devices --->

☒ RAM block device support //把文件系统做成块设备来访问，所以要支持块设备

(16) Default number of RAM disks

(8192) Default RAM disk size (kbytes)

第二步：make zImage

3. 通过工具制作镜像

(1)制作一个空的 randisk

```
dd if=/dev/zero of=randisk bs=1k count=8192
```

//dd: 输入输出命令,

//if: 输入文件, 从/dev/zero 中输入, 全部输入 0

//of: 输出文件名字叫做 initrd.img

//bs: block size

//count: 个数

//创建了一个大小为 8M 的 ramdisk, 并初始 ramdisk 化为 0

(2)格式化为 ext2 文件系统

```
sudo mkfs.ext2 -F randisk
```

(3)挂载

```
sudo mount -t ext2 -o loop randisk /mnt
```

(4)把跟文件系统拷贝到/mnt 下, 就相当于拷贝到 ramdisk

```
sudo cp rootfs/* /mnt/ -fr
```

(5)取消挂载

```
sudo umount /mnt
```

(6)压缩 ramdisk 数据块

```
gzip --best -c ramdisk > ramdisk.img
```

(7)查看

```
hexdump ramdisk.img
```

4. 通过工具烧写

●方法一: 通过 uboot 烧写

```
tftp 50008000 ramdisk.img
```

```
nand erase 700000 500000
```

```
nand write 50008000 700000 500000
```

●方法二: 在系统正常状态下, 进行烧写, 通过 flash_eraseall 和 nand_writes 烧写

```
flash_eraseall /dev/mtd3(自己定)
```

```
nandwrite -p /dev/mtd3 ramdisk.img
```

●设置 ramdisk 启动参数

```
setenv bootcmd nand read 50008000 200000 500000 \;
```

```
nand read 50800000 700000 500000\;
```

```
bootm 50008000
```

```
//-----
```

```
setenv bootargs root=/dev/ram rw init=/linuxrc
```

```
console=ttySAC0,115200 initrd=0x50800000,8M rootfstype=ext2
```

●参数说明

一在 bootcmd 中可以不加 0x, 因为这个是 uboot 传递给内核的, uboot 中默认的数字都是十六进制; 而在 bootargs 是 linux 传递给根文件系统的, 在 linux 中默认的数字不是十六进制, 所以在数字前要加上 0x

—rootfstype=ext2 可要可不要

5. 测试

●查看启动打印的分区信息

●ls /dev/mtd*: 查看是否挂载上

```
crw-rw---- 1 0 0 90, 0 /dev/mtd0
crw-rw---- 1 0 0 90, 1 /dev/mtd0ro
brw-rw---- 1 0 0 31, 0 /dev/mtdblock0
kernel 对 nandflash 的操作有字符和块操作！
```

●cat /proc/mtd: 查看分区

三. ramdisk 文件系统制作

1. 分区规划

同上，只需要把内核和跟文件系统合并为一个分区，其他不变

2. 内核支持文件系统

同上 (ramdisk)

General setup ---> //把文件系统编译进内核

```
[*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
(/home/tarena/workdir/rootfs/rootfs) Initramfs source file(s)
```

3. 通过工具制作镜像

make zImage(arch/arm/boot/zImage(kernel+rootfs))

4. 通过工具烧写

●方法一 uboot:

```
tftp 50008000 initramfs.img
nand erase 200000 a00000
nand write 50008000 200000 a00000
```

●方法二: 通过 flash_eraseall 、nandwrite 同上

●设置启动参数:

```
setenv bootcmd nand read 50008000 200000 a00000 \; bootm 50008000
```

```
setenv bootargs rdinit=/linuxrc console=ttySAC0,115200
```

问题: Warning: unable to open an initial console.

解决方法: sudo mknod rootfs/dev/console c 5 1

三. cramfs 文件系统制作

1. 分区规划

同上

2. 内核支持文件系统

标准 Linux 内核默认支持 cramfs, 标准 Android 内核不支持 cramfs

File systems --->

```
[*] Miscellaneous filesystems --->
```

<*> Compressed ROM file system support (cramfs)

3. 通过工具制作镜像

```
sudo mkfs.cramfs rootfs rootfs.cramfs
```

4. 通过工具烧写

```
setenv bootcmd nand read 50008000 200000 500000\; bootm 50008000
setenv bootargs root=/dev/mtdblock3(1f03) init=/linuxrc console=ttySAC0,115200
rootfstype=cramfs
```

四、yaffs2 文件系统

1. 分区规划

同上

2. 内核支持文件系统

标准 Linux 内核默认不支持 yaffs2，标准 Android 内核支持 yaffs2

3. 通过工具制作镜像

```
sudo mkyaffs2image myrootfs rootfs.yaffs2
```

4. 通过工具烧写

```
● tftp 50008000 rootfs.yaffs2
nand erase c00000
nand write.yaffs 50008000 c00000 $filesize
● setenv bootargs root=/dev/mtdblock3 init=/linuxrc
console=ttySAC0,115200 rootfstype=yaffs2
```

五、rootfs.img 和 userdata.img 关联起来

1. 分区规划

		根文件系统		userdata	
0-----1M	-----2M	-----7M	-----12	-----	剩余
u-boot	param	kernel	cramfs	yaffs2	

- 2.

- 3.

- 4.

5. 关联

```
mount -t yaffs2 /dev/mtdblock4 /home/
```

把 mtdblock4 挂载到 home 目录下，挂载的类型为 yaffs2 类型

6. 测试

md5sum 文件：校准文件

附录一 遗忘点

1. 一条汇编指令是 4 字节 (32 位)
 2. hexdump + 二进制文件: 查看存二进制文件
 3. arm-linux-objdump -d *.o:把.o 文件反汇编
 4. 对于处理器, 读取内存的时间远大于语句执行的时间
 5. 读静态内存几乎不用时间, 读动态内存时间比较长, 芯片内部集成的小内存是静态内存, 外部连接的 ddr 是动态内存
 6. 汇编操作的立即数最大为 4k(4096 字节:0x08)
 7. volatile 作用:
 - (1) 确保本指令不会因为编译器的优化而省略, 而且要求每次直接读值
 - (2) 对于不同体系的结构, 设备可能是端口映射, 也有可能是内存映射。如果系统支持独立的 IO 地址空间, 并且是端口映射, 就必须使用汇编语言程序完成实际对设备的控制, 因为 C 语言中没有提供真正的端口概念, 如果是内存映射, 那就方便多了
- ```
#define IOPIN (*(volatile unsigned long*)0xe00280000)
IOPIN=0x20;
```
- 注释: (volatile unsigned long\*)0xe00280000 表示把 0xe00280000 强制转换为(volatile unsigned long)型的指针, 暂时记为 p, define A \*p 就表示 A 为 p 所指向的内容; IOPIN 可以看为一个有了地址的常量
8. 回车为'\r', 换行为'\n', 在 c 语言中'\n'集成了'\r', 而串口则是分开的
  9. export PS1="##", 修改终端前面的符号
- 
10. nCE、CE、#CE、CE#: 都表示低电平有效
  11. 文档比较软件 beyond compare (windows)
  12. 文档生成软件 doxygen
  13. sudo /etc/init.d/networking restart  
auto eth0  
iface eth0 inet static  
address 192.168.1.8  
netmask 255.255.255.0  
gateway 219.218.122.254
  14. 生效环境变量  
source /etc/environment 或者  
source /etc/profile
  15. 解压命令:  
tar zxvf \*.tar.gz  
tar jxvf \*.tar.bz\*  
复制一个目录: cp -rf 文件夹
  16. fprintf 函数  
是 C/C++ 中的一个格式化写—库函数; 其作用是格式化输出到一个流/文件中;  
函数完整形式: int fprintf(FILE \*stream,char \*format,[argument])  
eg:

```
char name[20];
sprintf(name, "%s_Employee%d", "cqyh", i);
```

#### 17. sprintf 函数

```
int sprintf(char *buffer, const char *format, [argument] ...);
```

##### 参数列表:

buffer: char 型指针, 指向将要写入的字符串的缓冲区。

format: char 型指针, 指向的内存里面存放的将要格式字符串。

[argument]...: 可选参数, 可以是任何类型的数据。

返回值: 字符串长度 (strlen)

##### 参数说明

sprintf 格式的规格如下所示。[]中的部分是可选的。

%[指定参数][标识符][宽度][.精度]指示符

若想输出`%`本身时, 请这样`%%`处理。

1. 处理字符方向。负号时表示从后向前处理。
2. 填空字元。0 的话表示空格填 0; 空格是内定值, 表示空格就放着。
3. 字符总宽度。为最小宽度。
4. 精确度。指在小数点后的浮点数位数。

##### 转换字符

%% 印出百分比符号, 不转换。

%c 整数转成对应的 ASCII 字元。

%d 整数转成十进位。

%f 倍精确度数字转成浮点数。

%o 整数转成八进位。

%s 整数转成字符串。

%x 整数转成小写十六进位。

%X 整数转成大写十六进位。

##### 用法 1: 数字转换为字符串

```
$formatted = sprintf ("%06.2f", $money); // 此时变数 $ formatted 值为 "123.10"
```

```
$formatted = sprintf ("%08.2f", $money); // 此时变数 $ formatted 值为 "00123.10"
```

```
$formatted = sprintf ("% -08.2f", $money); //此时变数 $ formatted 值为 "123.1000"
```

```
$formatted = sprintf ("% .2f%%", 0.95 * 100); // 格式化为百分比
```

%08.2f 解释:

%开始符

0 是 "填空字元" 表示,如果长度不足时就用 0 来填满。

8 格式化后总长度

2f 小数位长度, 即 2 位

第 3 行值为"00123.10" 解释:

因为 2f 是(2 位)+小数点符号(1 位)+前面 123(3 位)=6 位, 总长度为 8 位,故前面用[填空字元]0 表示, 即 00123.10

第 4 行值为"123.1000" 解释:

-号为反向操作, 然后填空字元 0 添加在最后面了

##### 用法 2: 拼接字符串

```
char a1[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
```

```
char a2[] = {'H', 'I', 'J', 'K', 'L', 'M', 'N'};
```



```
sprintf(s, "%.4s%.4s", sizeof(a1), a1, sizeof(a2), a2); //产生: "ABCDEFGHJKLMN"
sprintf(version_rspr.s8Version,"%d%d",buffer[0],buffer[1]=0x50);
```

18. UC 开发时，所有的设备、文件都当文件操作，网络设备例外

19. 常量指针和指针常量

常量指针：

```
const int *p 或者 int const *p
```

const int \* pi=&a; "告诉编译，\*pi 是常量，不能将\*pi 作为左值进行操作。

\*p=...是错误的，只能 p=地址

常量指针是一个指向常量的指针变量，对象因为是一个常量，所以不能改变，但是可以改变指向的方向

指针常量

定义" int \*const pi=&a; "告诉编译，pi 是常量，不能作为左值进行操作，但是允许修改间接访问值，pi=地址是错误的，只能\*pi=值

20. 用户空间段错误

ulimit -c unlimited: 取消掉 core 文件的限制

```
gcc test.c -o test -g
```

```
gdb test -c core
```

```
bt
```

21. 内核空间奔溃

```
segment
```

在开发板上: cat /proc/kallsys >2.txt 列出了所有的函数，函数前面的地址为函数在开发板上的加载地址

```
arm-linu-objdump -D ***.ko > .txt
```

PC 寄存器的内容：

22. 指针-->成员变量

```
eg: (a+i)->dlc
```

数组.成员变量

```
eg: a[i].dlc
```

```
struct student *a=kmalloc(sizeof(student)*20,GPF_KERNEL)
```

那么去第 2 个结构体的地址是&(a[2])或者是 a+2

23. char a[200]={0}

memset(a,0,200) →对数组 a 清空

24. #if 0

```
....
```

```
#endif //注释
```

25. 某一位（或几位）值 1

```
x |= 0x1<<位数
```

```
x |= 0x1<<8 //第 8 位置 1
```

```
x |= 0x3<<8 //第 8、9 位置 1
```

```
x |= 0xf<<8 //8、9、10、11 位值 1
```

26. 某一位（或几位）值 0

```
x &=~(0x1<<位数)
```

```
x &=~(0x1<<8) //第 8 位置 0
```

```
x &=~(0x3<<8) //第 8、9 位置 0
```

- `x &=~(0xf <<8)` //8、9、10、11 位值 0
27. `ARRAY_SIZE(数组)`: 求数组元素的个数
28. `hexdump`: 查看二进制文件  
`hexdump /dev/节点`: 查看节点的二进制信息
29. `WR/C1`: 可读可写, 并写 1 清除这个位
30. `linux` 中 `container_of(ptr, type, member)` 宏的作用是传入结构体类型 `type` 的域 `member` 地址 `ptr`, 返回该结构体变量的首地址
31. `gcc -fpic`: 编译与地址无关
32. `Linux` 内核的结构体在应用程序都可以用
33. `(cd u-boot && make CW210_config && make all) > tmp/log/uboot.log 2>&1`  
`tmp/log/uboot.log`:  
`2>&1`: 2 为标准出错, 1 为标准输出, 0 为标准输入; 这句话表示把标准出错打印到标准输出上 (显示屏), 不保存到日志文件里面
32. `GCC -I -L`: 库、头文件位置  
`PATH`  
`LIBRARY_PATH`:  
eg: `arm-linux-gcc $(obj) -o main -I ../include/ -lpthread`
33. `cat /proc/device`: 查看设备号
34. `char buf[8]={0},buffer[8]={0};`  
`buffer[0]=96;`  
`buffer[1]=0x50;`  
`sprintf(buf,"%d%d",buffer[0],buffer[1]);` //buf 为 "9680"  
`sprintf(buf,"%c%c",buffer[0],buffer[1]);` //buf 为 "aP"
35. `find . -name *`