Luffy Li (李壮)

嵌入式学习笔记（一）

ARM 裸机全集

基于S5PV210

[一、 ARM体系结构与汇编指令 1](#_Toc26340253)

[1.1可编程器件的编程原理 1](#_Toc26340254)

[1.1.1电子器件的历史背景 1](#_Toc26340255)

[1.1.2可编程器件的特点 1](#_Toc26340256)

[1.1.3从源代码到CPU的执行过程 1](#_Toc26340257)

[1.2指令集对CPU的意义 1](#_Toc26340258)

[1.3 RISC和CISC的区别 2](#_Toc26340259)

[1.4统一编址和独立编址&冯诺依曼结构和哈佛结构 2](#_Toc26340260)

[1.4.1编址 2](#_Toc26340261)

[1.4.2 IO与内存统一编址 2](#_Toc26340262)

[1.4.3 IO与内存独立编址 2](#_Toc26340263)

[1.4.4两种编址方式的对比 2](#_Toc26340264)

[1.4.5代码和数据 2](#_Toc26340265)

[1.4.6冯诺依曼结构和哈佛结构 2](#_Toc26340266)

[1.4.7两种结构的对比 3](#_Toc26340267)

[1.5软件编程控制硬件的关键-寄存器 3](#_Toc26340268)

[1.5.1寄存器 3](#_Toc26340269)

[1.5.2两类寄存器：通用寄存器和SFR 3](#_Toc26340270)

[1.6 ARM体系结构要点总结 3](#_Toc26340271)

[1.6.1 ARM是RISC架构的 3](#_Toc26340272)

[1.6.2 ARM是IO与内存统一编址的 3](#_Toc26340273)

[1.6.3常见的ARM是哈佛结构的（ARM7除外） 4](#_Toc26340274)

[1.7 S5PV210的内存地址映射详解 4](#_Toc26340275)

[1.7.1内存地址映射 4](#_Toc26340276)

[1.7.2一些专业术语 4](#_Toc26340277)

[1.7.3地址映射注意点 5](#_Toc26340278)

[1.8 CPU和外部存储器的接口 5](#_Toc26340279)

[1.8.1 SoC常用外部存储器 5](#_Toc26340280)

[1.9 S5PV210的启动过程详解 6](#_Toc26340281)

[1.9.1内存 6](#_Toc26340282)

[1.9.2外存 6](#_Toc26340283)

[1.9.3 S5PV210的启动过程 6](#_Toc26340284)

[1.10 ARM的编程模式和7种工作模式 7](#_Toc26340285)

[1.10.1 ARM提供的指令集 7](#_Toc26340286)

[1.10.2 ARM的7种工作模式 8](#_Toc26340287)

[1.11 ARM的37个寄存器详解 8](#_Toc26340288)

[1.11.1 CPSR寄存器 9](#_Toc26340289)

[1.11.2 PC寄存器 10](#_Toc26340290)

[1.12 ARM的异常处理方式简单介绍 10](#_Toc26340291)

[1.12.1什么是异常 10](#_Toc26340292)

[1.12.2异常向量表 10](#_Toc26340293)

[1.12.3 ARM的异常处理机制 11](#_Toc26340294)

[1.13 ARM汇编指令1-ARM汇编特点 12](#_Toc26340295)

[1.13.1指令与伪指令 12](#_Toc26340296)

[1.13.2 ARM汇编特点1:LDR/STR架构 12](#_Toc26340297)

[1.13.3 ARM汇编特点2:8种寻址方式 12](#_Toc26340298)

[1.13.4 ARM汇编特点3: 指令后缀 B H S 12](#_Toc26340299)

[1.13.5 ARM汇编特点4：条件后缀 12](#_Toc26340300)

[1.14 ARM汇编指令2-常用指令 13](#_Toc26340301)

[1.14.1常用ARM指令1：数据处理指令 13](#_Toc26340302)

[1.14.2常用ARM指令2：CPSR访问指令 15](#_Toc26340303)

[1.14.3常用ARM指令3：跳转（分支）指令 15](#_Toc26340304)

[1.14.4常用ARM指令4：访存指令 15](#_Toc26340305)

[1.14.5常用ARM指令5 15](#_Toc26340306)

[1.14.6 ARM汇编中的立即数 15](#_Toc26340307)

[1.15 ARM汇编指令3-CP15协处理器 15](#_Toc26340308)

[1.15.1协处理器 15](#_Toc26340309)

[1.15.2协处理器CP15操作指令 16](#_Toc26340310)

[1.15.3 mrc & mcr的使用方法 16](#_Toc26340311)

[1.16 ARM汇编指令4-多寄存器指令 16](#_Toc26340312)

[1.16.1多寄存器访问指令 16](#_Toc26340313)

[1.16.2 18种后缀：stmia stmfd 16](#_Toc26340314)

[1.16.3四种栈 17](#_Toc26340315)

[1.16.4 !的作用 17](#_Toc26340316)

[1.16.5 ^的作用 17](#_Toc26340317)

[1.17 ARM汇编伪指令 17](#_Toc26340318)

[1.17.1伪指令的意义 17](#_Toc26340319)

[1.17.2 gnu汇编中的一些符号 17](#_Toc26340320)

[1.17.3常用gnu伪指令 17](#_Toc26340321)

[1.17.4偶尔会用到的gnu伪指令 18](#_Toc26340322)

[1.17.5最重要的几个伪指令 18](#_Toc26340323)

[二、GPIO和LED 18](#_Toc26340324)

[2.1 Makefile 18](#_Toc26340325)

[2.1.1 Makefile的一些基本概念 18](#_Toc26340326)

[2.1.2 Makefile的基本工作原理 19](#_Toc26340327)

[2.1.3 make的依赖性 19](#_Toc26340328)

[2.2 mkv210\_image.c详解 19](#_Toc26340329)

[2.2.1 Makefile文件 19](#_Toc26340330)

[2.2.2 mkv210\_image.c程序工作流分析 20](#_Toc26340331)

[2.2.3 mkv210\_image.c代码详解 20](#_Toc26340332)

[2.3一步步点亮LED 21](#_Toc26340333)

[2.3.1写程序的一般步骤 21](#_Toc26340334)

[2.3.2 GPIO:(General Purpose Input Output)通用输入输出 22](#_Toc26340335)

[2.3.3点亮LED相关寄存器 22](#_Toc26340336)

[2.3.4汇编编写及调用函数的方式 22](#_Toc26340337)

[2.4反汇编工具objdump的使用简介 22](#_Toc26340338)

[2.4.1反汇编的原理&为什么要用反汇编 22](#_Toc26340339)

[2.4.2反汇编文件的格式和看法 22](#_Toc26340340)

[2.4.3初识指令地址 22](#_Toc26340341)

[2.4.4展望：反汇编工具帮助我们分析链接脚本 23](#_Toc26340342)

[三、SDRAM和重定位relocate 23](#_Toc26340343)

[3.1汇编写启动代码之关看门狗 23](#_Toc26340344)

[3.1.1什么是看门狗？ 23](#_Toc26340345)

[3.1.2分析硬件物理特性、原理图、数据手册 23](#_Toc26340346)

[3.1.3找到关键性操作SFR 23](#_Toc26340347)

[3.1.4总结210中看门狗特性（iROM中已经关看门狗） 23](#_Toc26340348)

[3.2汇编写启动代码之设置栈和调用C语言 24](#_Toc26340349)

[3.2.1 C语言运行时需求和栈的意义 24](#_Toc26340350)

[3.2.2 CPU的7种工作模式和各种模式下的栈 24](#_Toc26340351)

[3.2.3查阅文档并设置栈指针至合法位置 24](#_Toc26340352)

[3.2.4汇编程序和C程序的互相调用 25](#_Toc26340353)

[3.2.5 C函数的编写和被汇编调用 25](#_Toc26340354)

[3.2.6使用C语言来访问寄存器的方法 25](#_Toc26340355)

[3.2.7神奇的volatile 25](#_Toc26340356)

[3.2.8总结 26](#_Toc26340357)

[3.3汇编写启动代码之开关iCache 26](#_Toc26340358)

[3.3.1什么是cache，有什么用 26](#_Toc26340359)

[3.3.2 iROM中BL0对cache的操作 26](#_Toc26340360)

[3.3.3汇编代码读写cp15以开关icache 26](#_Toc26340361)

[3.3.4实验验证 26](#_Toc26340362)

[3.4重定位引入和链接脚本 27](#_Toc26340363)

[3.4.1一个事实：大部分指令是位置有关编码 27](#_Toc26340364)

[3.4.2再接S5PV210的启动过程：三星推荐和uboot的实现是不同的 27](#_Toc26340365)

[3.4.3为什么要有重定位? 28](#_Toc26340366)

[3.4.4运行时地址由什么决定？ 28](#_Toc26340367)

[3.4.5链接地址由什么决定？ 28](#_Toc26340368)

[3.4.6从源码到可执行程序的步骤：预编译、编译、链接、（strip、objcopy）等 28](#_Toc26340369)

[3.4.7程序段的概念：代码段、数据段、bss段（ZI段）、自定义段 29](#_Toc26340370)

[3.4.8链接脚本究竟要做什么？ 29](#_Toc26340371)

[3.5代码重定位实战 30](#_Toc26340372)

[3.5.1任务：在iSRAM中将代码从0xd0020010重定位到0xd0024000 30](#_Toc26340373)

[3.5.2思路 30](#_Toc26340374)

[3.5.3 adr和ldr伪指令的区别 31](#_Toc26340375)

[3.5.4重定位（代码拷贝） 31](#_Toc26340376)

[3.5.5清bss段 31](#_Toc26340377)

[3.5.6长跳转 31](#_Toc26340378)

[3.6 SDRAM引入 33](#_Toc26340379)

[3.6.1 SDRAM的特性（容量大、价格低、掉电易失性、随机读写、总线式访问） 33](#_Toc26340380)

[3.6.2 SDRAM数据手册K4T1G164QQ 33](#_Toc26340381)

[3.6.3 SDRAM原理图 34](#_Toc26340382)

[3.6.4 SDRAM数据手册《NT5TU64M16GG-DDR2-1G-G-R18-Consumer》 36](#_Toc26340383)

[3.7汇编初始化SDRAM详解 36](#_Toc26340384)

[3.7.1初始化代码框架介绍（函数调用和返回、步骤等） 37](#_Toc26340385)

[3.7.2 27步初始化DDR2 37](#_Toc26340386)

[3.7.3设置IO端口驱动强度 37](#_Toc26340387)

[3.7.4 DRAM port时钟设置 37](#_Toc26340388)

[3.7.5 DMC0\_MEMCONTROL 38](#_Toc26340389)

[3.7.6 DMC0\_MEMCONFIG\_0 38](#_Toc26340390)

[3.7.7 DMC0\_MEMCONFIG\_1 38](#_Toc26340391)

[3.7.8 DMC\_DIRECTCMD 38](#_Toc26340392)

[3.7.9重定位代码到SDRAM中 38](#_Toc26340393)

[四、时钟系统 38](#_Toc26340394)

[4.1 SoC时钟系统简介 38](#_Toc26340395)

[4.1.1什么是时钟，SoC为什么需要时钟 38](#_Toc26340396)

[4.1.2时钟一般如何获得？ 39](#_Toc26340397)

[4.1.3为什么S5PV210使用第三种时钟？ 39](#_Toc26340398)

[4.1.4时钟和系统性能的关系、超频、稳定性 39](#_Toc26340399)

[4.1.5时钟和外设编程的关联 39](#_Toc26340400)

[4.1.6时钟和功耗控制的关系 39](#_Toc26340401)

[4.2 S5PV210的时钟系统简介 39](#_Toc26340402)

[4.2.1时钟域：MSYS、DSYS、PSYS 39](#_Toc26340403)

[4.2.2时钟来源：外部晶振+内部时钟发生器+PLL+分频电路 40](#_Toc26340404)

[4.2.3 PLL：APLL、MPLL、EPLL、VPLL 40](#_Toc26340405)

[4.3 S5PV210的时钟域详解 40](#_Toc26340406)

[4.3.1 MSYS域 40](#_Toc26340407)

[4.3.2 DSYS域 40](#_Toc26340408)

[4.3.3 PSYS域 40](#_Toc26340409)

[4.3.4各时钟的典型值（默认值，iROM中设置的值） 40](#_Toc26340410)

[4.4 S5PV210时钟体系框图详解 41](#_Toc26340411)

[4.5时钟设置的关键寄存器 41](#_Toc26340412)

[4.6汇编实现时钟设置代码详解 42](#_Toc26340413)

[4.6.1时钟设置的步骤分析 42](#_Toc26340414)

[4.6.2 CLK\_SRC0寄存器的配置 42](#_Toc26340415)

[4.6.3CLK\_LOCK寄存器的设置（PLL\_LOCK） 42](#_Toc26340416)

[4.6.4 CLK\_DIV0寄存器的设置 42](#_Toc26340417)

[4.6.5 PLL倍频的相关计算 42](#_Toc26340418)

[五、通信的基础概念 43](#_Toc26340419)

[5.1通信的基础概念 43](#_Toc26340420)

[5.1.1通信的发展历史 43](#_Toc26340421)

[5.1.2电子通信的概念1：同步通信和异步通信 43](#_Toc26340422)

[5.1.3电子通信概念3：电平信号和差分信号 43](#_Toc26340423)

[5.1.4电子通信概念3：并行接口和串行接口 44](#_Toc26340424)

[5.2串口通信的基本概念 44](#_Toc26340425)

[5.2.1串口通信的特点：异步、电平信号、串行 44](#_Toc26340426)

[5.2.2电平信号中的RS232电平和TTL电平 44](#_Toc26340427)

[5.2.3波特率 45](#_Toc26340428)

[5.2.4起始位、数据位、奇偶校验位、停止位 45](#_Toc26340429)

[5.2.5单工通信和双工通信 45](#_Toc26340430)

[5.3串口通信的基本原理 45](#_Toc26340431)

[5.3.1三根通信线：Tx Rx GND 45](#_Toc26340432)

[5.3.2收发双方事先规定好通信参数（波特率、数据位、奇偶校验位、停止位等） 46](#_Toc26340433)

[5.3.3信息以二进制流的方式在信道上传输 46](#_Toc26340434)

[5.3.4 DB9接口介绍 46](#_Toc26340435)

[5.4 S5PV210串行通信接口详解 46](#_Toc26340436)

[5.4.1 S5PV210的串口控制器工作原理框图 46](#_Toc26340437)

[5.4.2自动流控（AFC:Auto flow control） 47](#_Toc26340438)

[5.4.3串口的高级功能 47](#_Toc26340439)

[5.4.4串行通信与中断的关系 48](#_Toc26340440)

[5.4.5 210串行通信接口的时钟设计 48](#_Toc26340441)

[5.5S5PV210串行通信编程实战 49](#_Toc26340442)

[5.5.1整个流程分析 49](#_Toc26340443)

[5.5.2串口初始化关键步骤 49](#_Toc26340444)

[5.5.3主要的几个寄存器 49](#_Toc26340445)

[5.5.4在C源文件中定义访问寄存器的宏 49](#_Toc26340446)

[5.5.5串口Tx、Rx对应的GPIO的初始化 49](#_Toc26340447)

[5.5.6 UCON、ULCON、UMCON、UFCON等主要控制寄存器赋值 49](#_Toc26340448)

[5.5.7波特率的计算和设置 49](#_Toc26340449)

[5.5.8串口发送和接收函数的编写 49](#_Toc26340450)

[5.5.9综合调试：注意Makefile的修改 50](#_Toc26340451)

[5.6 uart stdio的移植 50](#_Toc26340452)

[5.6.1什么是stdio？ 50](#_Toc26340453)

[5.6.1printf函数的工作原理 50](#_Toc26340454)

[5.6.1移植printf函数的三种思路 50](#_Toc26340455)

[5.6.1printf移植 50](#_Toc26340456)

[5.6.1gcc可变参数及va\_arg介绍 50](#_Toc26340457)

[5.6.1vsprintf函数详解： 50](#_Toc26340458)

[5.6.1USB下载bin最多96KB，SD卡下载bin文件最多16KB怎么办？ 51](#_Toc26340459)

[六、按键和CPU的中断系统 51](#_Toc26340460)

[6.1什么是按键 51](#_Toc26340461)

[6.1.1按键的物理特性 51](#_Toc26340462)

[6.1.2按键的电学原理（结合原理图分析） 51](#_Toc26340463)

[6.1.3按键属于输入类设备 51](#_Toc26340464)

[6.1.4按键的2种响应方法 52](#_Toc26340465)

[6.2轮询方式处理按键 52](#_Toc26340466)

[6.2.1 X210开发板的按键接法 52](#_Toc26340467)

[6.2.2按键对应的GPIO模式设置 52](#_Toc26340468)

[6.2.3轮询方式处理按键的程序流程 52](#_Toc26340469)

[6.3串口输出与按键消抖 52](#_Toc26340470)

[6.3.1基于串口标准输出的按键调试 52](#_Toc26340471)

[6.3.2什么是消抖 52](#_Toc26340472)

[6.4 S5PV210的中断体系介绍 53](#_Toc26340473)

[6.4.1什么是中断？ 53](#_Toc26340474)

[6.4.2 SoC对中断的实现机制：异常向量表 53](#_Toc26340475)

[6.4.3 S5PV210的异常向量表 53](#_Toc26340476)

[6.4.4异常和中断的区别和联系 54](#_Toc26340477)

[6.5异常向量表的编程处理 54](#_Toc26340478)

[6.5.1像内存一样去访问异常向量表 54](#_Toc26340479)

[6.5.2函数名的实质就是函数的首地址 54](#_Toc26340480)

[6.5.3为什么中断处理要先在汇编中进行 54](#_Toc26340481)

[6.5.4汇编保存现场和恢复现场 55](#_Toc26340482)

[6.6 S5PV210的向量中断控制器 55](#_Toc26340483)

[6.6.1异常处理的2个阶段 55](#_Toc26340484)

[6.6.2回顾：中断处理的第一个阶段（异常向量表跳转阶段）处理 55](#_Toc26340485)

[6.6.3 S3C2440的第二阶段处理过程 55](#_Toc26340486)

[6.6.4 S5PV210的第二阶段处理过程 55](#_Toc26340487)

[6.6.5总结：第一个阶段都相同，第二个阶段各不同 56](#_Toc26340488)

[6.7 S5PV210中断处理的主要寄存器 56](#_Toc26340489)

[6.8 S5PV210中断处理的编程实战 57](#_Toc26340490)

[6.8.1中断控制器初始化 57](#_Toc26340491)

[6.8.2中断的使能与禁止 57](#_Toc26340492)

[6.8.3绑定自己实现的isr到VICnVECTADDRx 58](#_Toc26340493)

[6.8.4真正的中断处理程序如何获取isr 58](#_Toc26340494)

[6.9外部中断 58](#_Toc26340495)

[6.9.1什么是外部中断 58](#_Toc26340496)

[6.9.2电平触发和边沿触发 59](#_Toc26340497)

[6.9.3关键寄存器：CON、PEND、MASK 59](#_Toc26340498)

[6.10中断方式处理按键编程实战 59](#_Toc26340499)

[七、定时器、看门狗和RTC 59](#_Toc26340500)

[7.1什么是定时器 59](#_Toc26340501)

[7.1.1定时器是SoC中常见外设 59](#_Toc26340502)

[7.1.2定时器有什么用 60](#_Toc26340503)

[7.1.3定时器的原理 60](#_Toc26340504)

[7.1.4定时器和看门狗、RTC、蜂鸣器的关系 60](#_Toc26340505)

[7.2 S5PV210中的定时器 60](#_Toc26340506)

[7.2.1 PWM定时器(Pulse Width Modulation Timer) 60](#_Toc26340507)

[7.2.2系统定时器 60](#_Toc26340508)

[7.2.3看门狗定时器 61](#_Toc26340509)

[7.2.4实时时钟RTC(Real Time Clock) 61](#_Toc26340510)

[7.3 S5PV210的PWM定时器 61](#_Toc26340511)

[7.3.1为什么叫PWM定时器 61](#_Toc26340512)

[7.3.2 PWM定时器介绍 61](#_Toc26340513)

[7.3.3 S5PV210的PWM定时器框图简介 61](#_Toc26340514)

[7.3.4预分频器和分频器 61](#_Toc26340515)

[7.3.5 TCNT&TCMP、TCNTB&TCMPB、TCNTO 62](#_Toc26340516)

[7.3.6自动重载和双缓冲（auto-reload and double buffering） 62](#_Toc26340517)

[7.3.7什么是PWM 62](#_Toc26340518)

[7.3.8 PWM波形的生产原理 62](#_Toc26340519)

[7.3.9输出电平翻转器 63](#_Toc26340520)

[7.3.10死区生成器 63](#_Toc26340521)

[7.4蜂鸣器和PWM定时器编程实践 63](#_Toc26340522)

[7.4.1蜂鸣器的工作原理 63](#_Toc26340523)

[7.4.2原理图和硬件信息 64](#_Toc26340524)

[7.4.3 PWM定时器的主要寄存器 64](#_Toc26340525)

[7.5看门狗定时器 64](#_Toc26340526)

[7.5.1什么是看门狗、有何用 64](#_Toc26340527)

[7.5.2 S5PV210看门狗定时器的结构框图 64](#_Toc26340528)

[7.5.3看门狗定时器的主要寄存器 65](#_Toc26340529)

[7.6实时时钟RTC 65](#_Toc26340530)

[7.6.1何为实时时钟 65](#_Toc26340531)

[7.6.2 S5PV210实时时钟的结构框图 65](#_Toc26340532)

[7.6.3闹钟发生器 65](#_Toc26340533)

[7.6.4 S5PV210实时时钟的主要寄存器 65](#_Toc26340534)

[7.6.5 BCD码 65](#_Toc26340535)

[7.7 RTC编程实战 66](#_Toc26340536)

[7.7.1设置时间与读取显示时间 66](#_Toc26340537)

[7.7.2闹钟实验 66](#_Toc26340538)

[八、SD卡启动详解 66](#_Toc26340539)

[8.1主流的外存设备介绍 66](#_Toc26340540)

[8.1.1软盘、硬盘、光盘、CD、磁带 66](#_Toc26340541)

[8.1.2纯粹的Flash：NandFlash、NorFlash 66](#_Toc26340542)

[8.1.3 SD卡、MMC卡、MicroSD、TF卡 67](#_Toc26340543)

[8.1.4 iNand、MoviNand、eSSD 67](#_Toc26340544)

[8.1.5 SSD（固态硬盘） 67](#_Toc26340545)

[8.2 SD卡的特点和背景知识 67](#_Toc26340546)

[8.2.1 SD卡和MMC卡的关系 67](#_Toc26340547)

[8.2.2 SD卡和Nand、Nor等Flash芯片差异 67](#_Toc26340548)

[8.2.3 SD卡与MicroSD的区别 67](#_Toc26340549)

[8.2.4 SD卡与TF卡的区别 67](#_Toc26340550)

[8.3 SD卡的编程接口 67](#_Toc26340551)

[8.3.1 SD卡的物理接口 67](#_Toc26340552)

[8.3.2 SD协议与SPI协议 68](#_Toc26340553)

[8.3.3 SPI协议的特点（低速、接口操作时序简单、适合单片机） 68](#_Toc26340554)

[8.3.4 SD协议的特点（高速、接口时序复杂，适合有SDIO接口的SoC） 68](#_Toc26340555)

[8.3.5 S5PV210的SD/MMC控制器 68](#_Toc26340556)

[8.4 S5PV210的SD卡启动详解 68](#_Toc26340557)

[8.4.1 SoC为何要支持SD卡启动 68](#_Toc26340558)

[8.4.2 SD卡启动的难点在哪里（SRAM、DDR、SDcard） 68](#_Toc26340559)

[8.4.3 S5PV210的启动过程回顾 69](#_Toc26340560)

[8.4.4 SD卡启动流程（bin文件小于16KB时和大于16KB时） 69](#_Toc26340561)

[8.4.5最重要的但是却隐含未讲的东西 69](#_Toc26340562)

[8.4.6 SoC支持SD卡启动的秘密（iROM代码） 69](#_Toc26340563)

[8.4.7扇区和块的概念 69](#_Toc26340564)

[8.4.8用函数指针方式调用device copy function 70](#_Toc26340565)

[8.5 S5PV210的SD卡启动实战 70](#_Toc26340566)

[8.5.1任务：大于16KB的bin文件使用SD卡启动 70](#_Toc26340567)

[8.5.2代码划分为2部分（BL1和BL2） 70](#_Toc26340568)

[8.5.3 uboot中的做法 71](#_Toc26340569)

[九、NandFlash和iNand 71](#_Toc26340570)

[9.1 NandFlash的接口 71](#_Toc26340571)

[9.1.1 Nand的型号与命名 71](#_Toc26340572)

[9.1.2 Nand的数据位 72](#_Toc26340573)

[9.1.3 Nand的功能框图 72](#_Toc26340574)

[9.2 NandFlash的结构 72](#_Toc26340575)

[9.2.1 Nand的单元组织：block与page（大页Nand与小页Nand） 73](#_Toc26340576)

[9.2.2带内数据和带外数据（ECC与坏块标记） 73](#_Toc26340577)

[9.2.3 Nand的地址时序 73](#_Toc26340578)

[9.2.4 Nand的命令码 73](#_Toc26340579)

[9.3 Nand的常见操作及流程分析 74](#_Toc26340580)

[9.3.1坏块检查 74](#_Toc26340581)

[9.3.2页写（program）操作 74](#_Toc26340582)

[9.3.3擦除（erase）操作 74](#_Toc26340583)

[9.3.4页读(read)操作 75](#_Toc26340584)

[9.4 S5PV210的NandFlash控制器 75](#_Toc26340585)

[9.4.1 SoC的Nand控制器的作用 75](#_Toc26340586)

[9.4.2结构框图分析 75](#_Toc26340587)

[9.4.3 S5PV210的Nand控制器的主要寄存器 75](#_Toc26340588)

[9.5 Nand操作代码解析 75](#_Toc26340589)

[9.5.1擦除函数 75](#_Toc26340590)

[9.5.2页读取函数 75](#_Toc26340591)

[9.5.3页写入函数 75](#_Toc26340592)

[9.6 iNand介绍 75](#_Toc26340593)

[9.6.1 iNand/eMMC/SDCard/MMCCard的关联 75](#_Toc26340594)

[9.6.2 iNand/eMMC的结构框图 75](#_Toc26340595)

[9.6.3 iNand/eMMC的物理接口和SD卡物理接口的对比 75](#_Toc26340596)

[9.6.4结论：iNand/eMMC其实就是芯片化的SD/MMC卡，软件操作和SD卡相同 75](#_Toc26340597)

[十、I2C通信详解 75](#_Toc26340598)

[10.1什么是I2C通信 75](#_Toc26340599)

[10.1.2通信特征：串行、同步、非差分、低速 75](#_Toc26340600)

[10.1.3突出特征1：主设备 + 从设备 76](#_Toc26340601)

[10.1.4突出特征2：可以多个设备挂在一条总线上（从设备地址） 76](#_Toc26340602)

[10.2由I2C学通信时序 76](#_Toc26340603)

[10.2.1什么是时序？ 76](#_Toc26340604)

[10.2.2 I2C总线空闲状态、起始位、结束位 76](#_Toc26340605)

[10.2.3 I2C数据传输格式（数据位&ACK） 77](#_Toc26340606)

[10.2.4数据在总线上的传输协议 77](#_Toc26340607)

[10.3 S5PV210的I2C控制器 77](#_Toc26340608)

[10.3.1结构框图 77](#_Toc26340609)

[10.3.2系统分析I2C的时钟 77](#_Toc26340610)

[10.3.3主要寄存器I2CCON、I2CSTAT、I2CADD、I2CDS 78](#_Toc26340611)

[10.4 X210板载gsensor介绍 78](#_Toc26340612)

[10.4.1原理图查询 78](#_Toc26340613)

[10.4.2重力加速度传感器 78](#_Toc26340614)

[10.4.3 I2C从设备的设备地址 78](#_Toc26340615)

[10.4.4 I2C从设备的通信速率 78](#_Toc26340616)

[10.5 I2C总线的通信流程 79](#_Toc26340617)

[10.5.1 S5PV210的主发送流程图 79](#_Toc26340618)

[10.5.2 S5PV210的主接收流程图 79](#_Toc26340619)

[10.5.3 gsensor的写寄存器流程图 79](#_Toc26340620)

[10.5.4 gsensor的读寄存器流程图 79](#_Toc26340621)

[10. 6 I2C通信代码分析 79](#_Toc26340622)

[10. 6.1 I2C控制器初始化：s3c24xx\_i2c\_init 79](#_Toc26340623)

[10. 6.2 I2C控制器主模式开始一次读写：s3c24xx\_i2c\_message\_start 79](#_Toc26340624)

[10. 6.3 I2C控制器主模式结束一次读写：s3c24xx\_i2c\_stop 79](#_Toc26340625)

[10. 6.4 gsensor写寄存器：gsensor\_i2c\_write\_reg 79](#_Toc26340626)

[10. 6.5 gsensor读寄存器：gsensor\_i2c\_read\_reg 79](#_Toc26340627)

[10. 6.6 gsensor编程：gsensor\_initial等 79](#_Toc26340628)

[十一、ADC 79](#_Toc26340629)

[11.1 ADC的引入 79](#_Toc26340630)

[11.1.1什么是ADC 80](#_Toc26340631)

[11.1.2为什么需要ADC 80](#_Toc26340632)

[11.1.3关于模拟量和数字量 80](#_Toc26340633)

[11.1.4有AD自然就有DA 80](#_Toc26340634)

[11.2 ADC的主要相关概念 80](#_Toc26340635)

[11.2.1量程（模拟量输入范围） 80](#_Toc26340636)

[11.2.2精度（分辨率resolution） 80](#_Toc26340637)

[11.2.3转换速率（MSPS与conventor clock的不同） 81](#_Toc26340638)

[11.2.4通道数 81](#_Toc26340639)

[11.3 S5PV210的ADC控制器 81](#_Toc26340640)

[11.3.1 ADC和（电阻式）触摸屏的关系 81](#_Toc26340641)

[11.3.2 ADC的工作时钟框图 81](#_Toc26340642)

[11.3.3 210的10个ADC通道（注意ADC引脚和GPIO的区别） 81](#_Toc26340643)

[11.3.4 ADC控制器的主要寄存器 81](#_Toc26340644)

[11.4 AD转换的编程实践 82](#_Toc26340645)

[11.4.1 AD控制器初始化 82](#_Toc26340646)

[11.4.2循环进行AD采样 82](#_Toc26340647)

[11.4.3编译运行调试 82](#_Toc26340648)

[11.4.5 start by read模式介绍 82](#_Toc26340649)

[11.4.6 DAC的应用简介 82](#_Toc26340650)

[十二、LCD显示器 82](#_Toc26340651)

[12.1 LCD简介 82](#_Toc26340652)

[12.1.1什么是LCD 82](#_Toc26340653)

[12.1.2其他主流显示设备（LED、CRT、等离子、OLED） 82](#_Toc26340654)

[12.1.3 LCD的显示原理和特点(液晶分子透光+背光) 83](#_Toc26340655)

[12.1.4 LCD的发展史和种类(TN/STN/TFT) 83](#_Toc26340656)

[12.2 LCD的接口技术 83](#_Toc26340657)

[12.2.1从电平角度讲本质上都是TTL信号 83](#_Toc26340658)

[12.2.2各种接口（TTL、LVDS、EDP、MIPI）在传输速率、距离、适配性方面不同 83](#_Toc26340659)

[12.2.3 RGB接口详情 83](#_Toc26340660)

[12.4 LCD如何显示图像 84](#_Toc26340661)

[12.4.1像素(pixel) 84](#_Toc26340662)

[12.4.2扫描 84](#_Toc26340663)

[12.4.3驱动器&控制器 85](#_Toc26340664)

[12.4.4显示内存（显存） 85](#_Toc26340665)

# 一、 ARM体系结构与汇编指令

## 1.1可编程器件的编程原理

### 1.1.1电子器件的历史背景

(1)模拟器件 > 数字器件

模拟器件：输入输出都是模拟信号（值随时间连续变化的信号）。

数字器件：输入模拟或数字信号，输出数字信号（值为1或0的信号）。

P.S：数字器件集成度高，模拟器件集成度低。

(2) ASIC > 可编程器件

ASIC: application specific integrated circuit 专用集成电路

根据功能需要，专门定做的芯片，内部程序固化，不能修改。

### 1.1.2可编程器件的特点

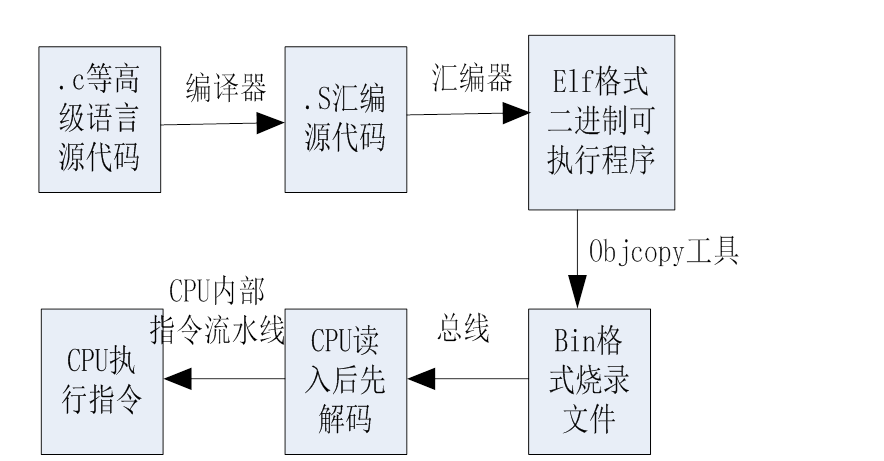
(1)CPU在固定的时钟频率控制下节奏运行。

(2)CPU可以通过总线读取外部存储设备的二进制指令集，进行解码执行。

(3)可被CPU读取、解码和执行的二进制指令集是一款CPU的特征。

P.S : 程序即指令集。

### 1.1.3从源代码到CPU的执行过程



## 1.2指令集对CPU的意义

(1)指令集是一款CPU的特征。它是由CPU的设计者定义的。 CPU内部电路设计就是为了实现指令的功能。

(2)不同CPU的指令集不同，因此汇编程序不能在不同CPU之间互相移植。

(3)指令集类似于CPU的API接口。

(4)汇编指令集是机器指令集的助记符。

(5)使用汇编可以充分发挥CPU的设计特点，因此在操作系统内核效率极其重要的地方都需要使用汇编处理。

充分发挥CPU的设计特点：

虽然C语言到最后也会被编译为汇编语言，但是被编译器加工处理后的程序并没有直接写出来的汇编程序简洁高效。

## 1.3 RISC和CISC的区别

(1) CISC: Complex instruction set computer 复杂指令集 intel

CISC的设计理念：用最少的指令完成任务，例如乘加指令，把乘指令+加指令做成一个指令。(所以指令繁多，300左右) 这需要CPU的逻辑电路设计更复杂，工艺要求更高。编译器的设计简单了。

(2)RISC: Reduced instuction set computer 精简指令集 arm

RISC的设计理念：只让CPU提供基本的功能指令（所以指令较少，30左右），让编程人员通过程序来实现更复杂的功能。 CPU的设计和工艺相对简单。编译器的设计复杂了。

(3)发展趋势：没有纯粹的RISC和CISC，发展方向是二者结合，介于二者之间的一种CPU。

## 1.4统一编址和独立编址&冯诺依曼结构和哈佛结构

### 1.4.1编址

CPU通过地址总线访问内存和外设。地址总线的位数是由CPU的设计者决定的。n位的CPU的可以访问的最大地址空间为2expn。所以CPU的寻址空间是一种有限资源。

### **1.4.2 IO与内存统一编址**

（IO只是不太严谨的说法，其实指的是所有的外设。）CPU有限的地址寻址空间给内存一部分，其他给外设。CPU可以用访问内存的指令来访问外设，即把外设的寄存器地址当做内存的地址来访问。所以叫IO与内存统一编址。

### **1.4.3 IO与内存独立编址**

不通过地址总线来访问外设，而是使用专用的CPU指令访问外设。

### **1.4.4两种编址方式的对比**

IO与内存统一编址的优点是编程方便。缺点是占用了CPU的寻址空间；

IO与内存独立编址的优点是不占用CPU的寻址空间。缺点是CPU的设计更复杂。

### **1.4.5代码和数据**

(1)程序运行的两大要素：代码+数据

(2)代码即我们编写的源代码经过编译、汇编生成的机器码，它可以被CPU读取、解释和执行，CPU不会改写代码，所以代码是只读的。

(3)数据即程序运行过程中定义和产生的变量的值，程序运行实际就是为了改变数据的值。

### **1.4.6**冯诺依曼结构和哈佛结构

**哈佛结构**：哈佛结构就是将程序的代码和数据分开存放的一种结构，而他们存放的位置可以是相同的也可以是不同的（ROM&RAM或者RAM），总之只要是分成两个部分单独访问的结构都可以叫哈佛结构。（例如：51的程序运行时，代码放在ROM（NorFlash）中原地运行，而数据则存放在RAM中随代码动作而变动；而S5PV210程序运行时，代码和数据都在DRAM中运行，但是DRAM中又划分了代码段和数据段，二者互不干扰。）哈佛结构的特点就是代码和数据单独存放，使之不会互相干扰，进而当程序出BUG时，最多只会修改数据的值（因为代码部分是只读的，不可改写），而不会修改程序的执行顺序。因此，这种结构大量应用在嵌入式编程。

**冯诺依曼结构**：冯诺依曼结构是将代码和数据统一都放在RAM中，他们之间一般是按照程序的执行顺序依次存储。这样就会导致一个问题，如果程序出BUG，由于程序没有对代码段的读写限定，因此，它将拥有和数据一样的读写操作权限。于是就会很容易的死机，一旦代码执行出现一点改变就会出现非常严重的错误。但是，冯诺依曼结构的好处是可以充分利用有限的内存空间，并使CPU对程序的执行十分的方便，不用来回跑。

### 1.4.7两种结构的对比

冯诺依曼结构的优点是CPU处理起来简单，缺点是程序可改写，安全性和稳定性较差。

哈佛结构的优点是安全性和稳定性较好，缺点是编程较为复杂，需要统一规划链接地址等。

## 1.5软件编程控制硬件的关键-寄存器

### 1.5.1寄存器

(1)CPU可以像访问内存一样访问寄存器。（当然，前提是IO与内存统一编址）

(2)寄存器是CPU硬件设计者制定的，目的是留作外设被编程控制的活动开关。（正如汇编指令集像是CPU的API一样，寄存器就像是外设硬件的软件编程的API，使用软件编程硬件，其实就是使用软件读写该硬件的寄存器）。

### 1.5.2两类寄存器：通用寄存器和SFR

(1)通用寄存器（ARM中37个）是CPU的组成部分，CPU的很多活动都需通过通用寄存器的支持和参与。

(2)SFR(Special function register)不在CPU中，而存在于SoC的内部外设中，我们通过访问外设的SFR来编程控制该外设，这就是硬件被编程控制的方法。

## 1.6 ARM体系结构要点总结

### 1.6.1 ARM是RISC架构的

(1)ARM的汇编指令集只有30条左右。

(2)ARM的CPU是低功耗的（相对来说CISC的CPU电路设计复杂，模块多，功耗大）

(3)ARM非常适合应用于单片机、嵌入式，尤其是物联网领域。而服务器等高性能领域目前还是Intel为主导。

### 1.6.2 ARM是IO与内存统一编址的

(1)大部分ARM是32位架构的（M3/M4/M7/M0/ARM9/ARM11/A8/A9等）

(2)32位ARM CPU支持的内存小于4G，通过CPU地址总线来访问。

(3)SoC中的各种内部外设通过各自的SFR编程访问，这些SFR的访问方式类似于访问普通内存，这叫做IO与内存统一编址。

### 1.6.3常见的ARM是哈佛结构的（ARM7除外）

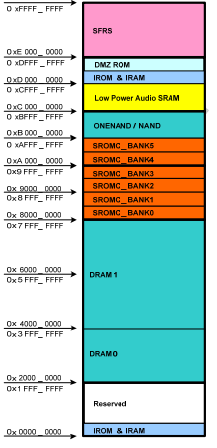
(1)哈佛结构保证了ARM CPU运行的稳定性和安全性，因此ARM适用于嵌入式领域。

(2)哈佛结构也决定了ARM裸机程序（使用实际地址即物理地址）的链接比较麻烦，必须使用复杂的链接脚本告知链接器如何组织程序；对于OS之上的应用（工作在虚拟地址之中）则不需要考虑这么多。

## 1.7 S5PV210的内存地址映射详解

### 1.7.1内存地址映射

ARM32位CPU的4G寻址范围映射。因为210是IO与内存统一编址，所以CPU可寻址空间除了RAM外还有其他IO寄存器地址等。



### **1.7.2一些专业术语**

ROM:只读存储器

RAM:随机访问存储器 （掉电即消失）

IROM:集成在SoC内部的ROM

IRAM:集成在SoC内部的RAM

DRAM:(Dynamic RAM)动态RAM，DRAM只能将数据保存很短的时间，为保持数据，DRAM使用电容存储，所以必须隔一段时间refresh一次，如果存储单元没有被刷新，存储的信息就会丢失。（掉电即消失）

SRAM:(Static RAM)静态RAM，SRAM只要通电，里边存储的数据就可以恒常保持。（掉电即消失）

SDRAM:(Synchronous Dynamic Access Memory)同步动态随机访问存储器。同步指的是内存工作需要同步时钟。

SROM：包括ROM和SRAM

NOR FLASH:非易失闪存技术，intel1988年开发。

NAND FLASH：非易失闪存技术，东芝1989年开发。

ONENAND：三星推出的NAND

iNand：SanDisk推出的NAND

### **1.7.3地址映射注意点**

6个128M的SROM(Bank0~Bank5)，用于总线型设备寻址的。

64KB iROM(BL0)和96KB 片内iRAM(BL1 BL2)，一般是启动时用到的。

## 1.8 CPU和外部存储器的接口

CPU访问内存和访问外存的方式不同，内存需要直接地址访问。外存是通过CPU的外存接口来访问（好处是不占用CPU的寻址空间，当然，它没有访问内存的总线式快，它的访问时序较复杂）

### 1.8.1 SoC常用外部存储器

NorFlash：1988年intel开发的非易失性存储技术，可以总线式访问，接到SROM bank，一般用于启动。

NandFlash：1989年东芝开发的非易失性存储技术（分为SLC和MLC。SLC容量不大，稳定性好，价格高；MLC容量很大，容易坏块，价格低，必要ECC校验）

Nor和Nand的区别：

(1)NorFlash的特点是芯片内执行（XIP,eXecute In Place）,用户可以直接运行装载NorFlash里边的代码。这样可以减少SRAM的容量从而节约了成本。Nor的传输速率很高，在1~4MB的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

(2)NandFlash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。NandFlash内部采用非线性宏单元模式，为固态大容量Memery的实现提供了廉价有效的解决方案。

(3)CPU不能直接运行NandFlash上的代码，因此好多使用NandFlash的开发板除了使用NandFlash以外，还作上了一块小的NorFlash来运行启动代码。

eMMC/iNand/moviNand：

eMMC(embeded MMC 嵌入式MMC卡)

iNand是SanDisk出的eMMC,

moviNand是三星出的eMMC

oneNand：oneNand是三星出的Nand

SD卡/TF卡/MMC卡：当成一种来处理。

eSSD

SATA硬盘（机械式访问、磁存储原理、SATA接口）

## 1.9 S5PV210的启动过程详解

### 1.9.1内存

SRAM 特点是容量小，价格高，优点是不需要软件初始化直接上电就能用

DRAM 特点是容量大，价格低，缺点是上电后不能直接使用，需要软件初始化

### 1.9.2外存

NorFlash：特点是容量小，价格高，优点是可以和CPU总线式相连，CPU上电后可以直接读取，所以用作启动介质。

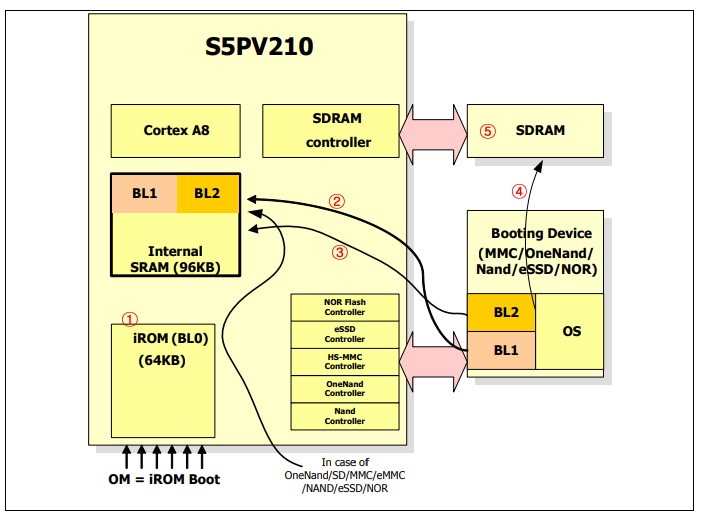
NandFlash：特点是容量大，价格低，缺点是CPU不能总线式访问，需要需要被初始化后通过时序接口读写。

一般PC机:很小容量的BIOS(NorFlash) + 很大容量的硬盘(类似于NandFlash) + 大容量的DRAM

一般单片机：很小容量的NorFlash + 很小容量的SRAM （所以不需要做初始化，上电都能用）

一般嵌入式（210）：因为NorFlash很贵，所以现在嵌入式系统倾向于不使用NorFLash,而是直接用：Nand + DRAM + iRAM(96KB SRAM) + iROM(64KB NorFlash)

### 1.9.3 S5PV210的启动过程



厂商内置iROM中的初始化程序BL0 、BL1 + BL2 （启动代码u-boot = BL1 + BL2）

第一步：CPU上电后，iROM中运行预先设置的代码（BL0），BL0负责关看门狗，初始化cache、设置堆栈指针、初始化内部的块设备拷贝函数（用于快速从nand、nor、sd卡等外部存储设备中把代码拷贝到iRAM中）、将引导设备的前16KB（BL1）拷贝到iRAM中、确认引导设备中的代码是否有效（检查校验字节）等。

第二步：iRAM中运行BL1，BL1负责初始化板子的各个部件等，然后加载BL2到iRAM。

第三步：iRAM中运行BL2，BL2负责初始化SDRAM，将内核拷贝到SDRAM，中然后引导内核的运行。

（以上启动过程为uboot小于96KB，是210推荐的启动方式，实际上我们uboot的启动过程和此处的启动过程是有些区别的）

**BL0做了什么**(SoC内部初始化，读取外部代码BL1)

关watch dog

初始化指令Cache

初始化栈

初始化堆 (为什么要初始化栈和堆？ 因为不初始化栈和堆，C语言没法跑，这也就是以上~程序要汇编来写的原因。这之后才可以用C语言来写)

初始化块设备复制函数(Device Copy Function) （正是因为有了多种Device Copy Funtion，所以它可以支持外部的多种启动方式）

初始化PLL和设置SoC时钟系统

复制BL1到内部的iRAM(16KB)

检查BL1的校验和

跳转到BL1地址开始执行BL1

当拨码开关OM拨向UART/USB启动时，就会从UART/USB启动（从这里启动不会做BL1头校验，这也是为什么我们使用DNW下载x210\_usb.bin下载到0xd0020010的原因），而不是SD卡启动。当拨向SD卡启动后，默认是从SD通道0板载eMMC卡启动，当eMMC卡BL1第一个扇区被破坏（BL1头校验不通过）才会从SD卡通道2外部SD卡启动。

## 1.10 ARM的编程模式和7种工作模式

### 1.10.1 ARM提供的指令集

ARM态-ARM指令集（32-bit）

Thumb态-Thumb指令集（16-bit）

Thumb2态-Thumb2指令集（16 & 32 bit）

Thumb指令集是对ARM指令集的一个子集重新编码得到的，指令长度为16位。通常在处理器执行ARM程式时，称处理器处于ARM状态；当处理器执行Thumb程式时，称处理器处于Thumb状态。Thumb指令集中的数据处理指令的操作数仍然为32位，指令寻址地址也是32位的。（在处理中断程序中只能使用ARM态）。

Thumb2指令集是Thumb指令集的一个扩充，既包含了16位也包含了32位，几乎可以提供与ARM相同的功能，但是指令密度（单位内存所存放的指令数）与Thumb指令集相似。有了Thumb2指令集就不再需要在ARM与Thumb两者之间切换了，因为Thumb2就够了!

### 1.10.2 ARM的7种工作模式

-User：非特权模式，大部分任务执行在这种模式

-FIQ(快中断)：当一个高优先级（fast）中断产生时将会进入这种模式

-IRQ(中断)：当一个低优先级（normal）中断产生时将会进入这种模式

-Supervisor(svc)：当复位或者软中断指令执行时将会进入这种模式

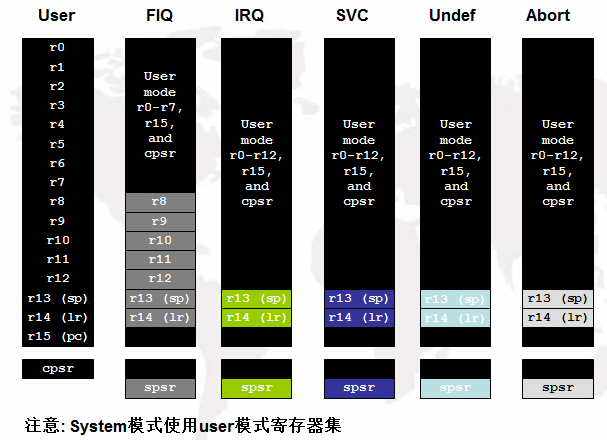
-Abort(中止abt)：当存取异常时将会进入这种模式

-Undef(未定义und)：当执行未定义指令时会进入这种模式

-System：使用和User模式相同寄存器集的特权模式

ARM有7种工作模式，除User为非特权模式（普通模式）外，其他6中工作模式为特权模式（Privilege），特权模式中除系统模式外，其他5种异常模式，其中Abort、Undef是异常中的异常。CPU同时只能在一种模式下工作，模式的切换，可以是程序员通过代码主动切换（通过写CPSR寄存器）；也可以是CPU在某些情况下自动切换（例如按键或触摸屏产生中断，CPU就会自动进入中断异常模式）。各种模式下权限和可以访问的寄存器不同。

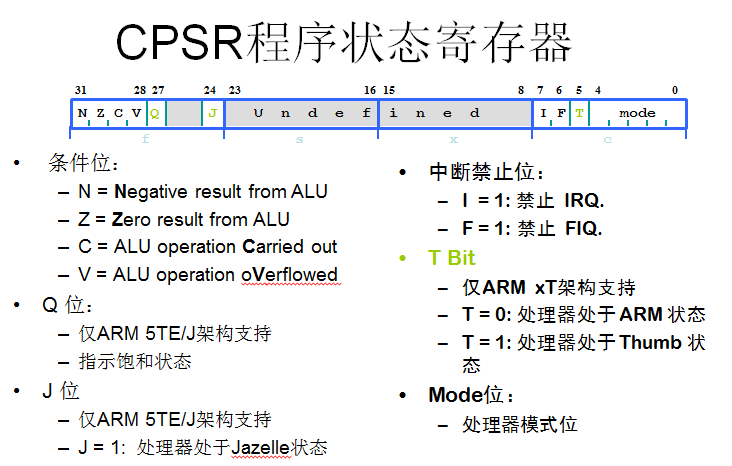
## 1.11 ARM的37个寄存器详解



* ARM中寄存器包括SFR和37个通用寄存器，通用寄存器是CPU(运算器+控制器+通用寄存器)的组成部分。
* 37个通用寄存器是搭配7种工作模式来学习的。因为每种工作模式下，可见的通用寄存器都不相同，每种模式下最多只能看到18个寄存器，部分寄存器虽然名字相同，但是在当前模式下不可见。
* 对于R13这个名字来说，在ARM中共有6个名叫R13（又叫sp）的寄存器，但是在每种特定工作模式下，只有当前模式的R13是可见的，其他r13必须切换到其他对应模式下才能看到。这种设计叫影子寄存（banked register）。
* 用户模式与系统模式共用一组寄存器，5种异常模式有各自的R13、R14和SPSR。
* R13用作栈指针，故也叫sp；由于处理器的每种运行模式均有自己独立的物理寄存器R13，在用户应用程序的初始化部分，一般都要初始化每种模式下的R13，使其指向该运行模式的栈空间。这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入R13所指向的栈，而当程序从异常模式返回时，则从对应的栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。
* R14用作保存返回地址，故也叫链接寄存器lr；当执行子程序调用指令(BL)时,R14可得到R15(程序计数器PC)的备份。在每一种运行模式下，都可用R14保存子程序的返回地址，当用BL或BLX指令调用子程序时，将PC的当前值复制给R14，执行完子程序后，又将R14的值复制回PC，即可完成子程序的调用返回。
* r15用作程序控制寄存器，故也叫PC；在ARM状态下,位[1:0]为0,位[31:2]用于保存PC,在Thumb状态下,位[0]为0,位[31:1]用于保存PC。由于ARM体系结构采用了多级流水线技术，对于ARM指令集而言，PC总是指向当前指令的下两条指令的地址,即PC的值为当前指令的地址值加8个字节。
* CPSR程序状态寄存器。记录当前CPU的状态；CPSR可在任何运行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。
* SPSR是CPSR的复制。每一种运行模式下又都有一个专用的物理状态寄存器，称为SPSR(Saved Program Status Register，备份的程序状态寄存器)，当异常发生时，SPSR用于保存CPSR的当前值，从异常退出时则可由SPSR来恢复CPSR。

由于用户模式和系统模式不属于异常模式，它们没有SPSR，当在这两种模式下访问SPSR，结果是未知的。

### 1.11.1 CPSR寄存器



(1)CPSR中各个bit位表明了CPU的某些状态信息，这些信息非常重要，和后边学到的汇编指令息息相关（譬如BLE指令中的E就和CPSR中Z标志位有关）

(2)CPSR中的I、F位和开中断、关中断有关

(3)CPSR中mode位（bit4~bit0）决定了CPU的工作模式，在uboot代码中会使用汇编进行设置。

(4)CPSR和SPSR的区别和联系：CPSR是程序状态寄存器，整个SoC中只有1个，而SPSR有5个，分别在5种异常模式下，作用是当从普通模式进入异常模式时，用来保存CPSR的，以返回普通模式时恢复原来的CPSR。

### 1.11.2 PC寄存器

程序控制寄存器，为程序指针，PC指向哪里，CPU就会执行哪条指令（所以程序跳转时就是把目标代码地址放到PC中）。

## 1.12 ARM的异常处理方式简单介绍

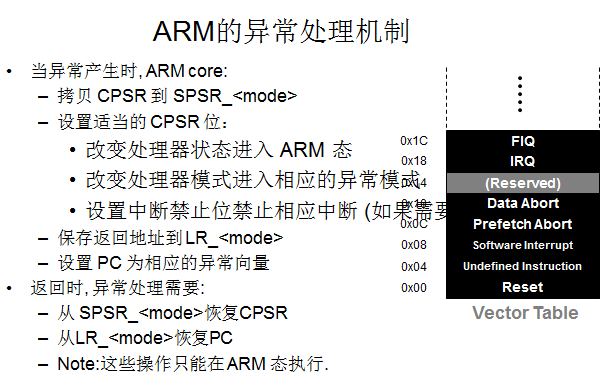
### 1.12.1什么是异常

正常工作之外的流程都叫异常

异常会打断正在执行的工作，并且一般我们希望异常处理完后继续回来执行原工作

中断是异常的一种

### 1.12.2异常向量表



所有的CPU都有异常向量表，这是CPU设计时就设定好的，是硬件决定的。

当异常发生时，CPU会自动动作（PC跳转到异常向量处处理异常，有时伴有一些辅助动作）

异常向量表是硬件向软件提供的处理异常的支持。

### 1.12.3 ARM的异常处理机制

当异常产生时, ARM core:

(1)将CPSR的值保存到将要执行的异常中断对应的SPSR中，以实现对处理器当前状态、中断屏蔽及各标志位的保护。

(2)设置当前状态寄存器CPSR的相应位。设置CPSR中的M4~M0的5位，进入相应工作模式，设置I=1禁止IRQ中断，如果进入复位模式或FIQ模式，还要设置F=1以禁止FIQ中断。

(3)将引起异常指令的下一条地址（断点地址）保存到 LR（R14）中，使异常处理程序执行完后正确返回原来程序处继续向下执行。

(4)给程序计数器PC强制赋值，转入向量地址，以便执行相应的处理程序。

每种中断异常模式对应两个寄存器SP和LR。

从中断返回。如果是复位异常，系统自动从0x00000000开始重新执行程序，无需返回

(1)首先恢复原来被保护的用户寄存器。

(2)将SPSR寄存器复制到CPSR中，使得原来CPSR状态从相应的SPSR中恢复，——恢复被中断的程序状态。

(3)根据异常类型将PC值恢复成断点地址，以继续执行用户原来运行着的程序。

(4)清除CPSR中的中断禁止标志I和F，开放外部中断和快速中断。

注意：(1)程序状态寄存器及断点地址的恢复必须同时进行。

(2)由于异常随机发生，所以要对异常向量进行初始化，即在异常向量的地址处放置一条跳转指令，跳转到异常处理程序。

## 1.13 ARM汇编指令1-ARM汇编特点

### 1.13.1指令与伪指令

(1)（汇编）指令是CPU机器指令的助记符，经过汇编后会得到一串1、0组成的机器码，可以由CPU读取执行。

(2)（汇编）伪指令本质上不是指令（只是和指令一起写在代码中），它是汇编器环境提供的，目的是用来指导汇编的过程，经过汇编后伪指令最终不会生成机器码。

### 1.13.2 ARM汇编特点1:LDR/STR架构

(1)ARM采用RISC架构，CPU本身不能直接读取内存，而需要先将内存中内容加载入CPU的通用寄存器中才能被CPU处理。

(2)ldr（load register）指令将内存内容加载入通用寄存器。

(3)str（store register）指令将寄存器内容存入内存空间中。

(4)ldr/str组合用来实现ARM CPU和内存数据交换。

1.13.3 ARM汇编特点2:8种寻址方式 https://blog.csdn.net/RJ\_Cheng/article/details/72858469

(1)寄存器寻址 mov r0, r1

(2)立即寻址 mov r0, #0xff00

(3)寄存器移位寻址 mov r0, r1, lsl#3 lsl(逻辑左移，左移一位相当于乘以2，左移三位相当于乘以8)

(4)寄存器间接寻址 ldr r1, [r2] (ld出现了，表示从内存到寄存器加载值)

(5)基址变址寻址 ldr r1, [r2, #4]

(6)多寄存器寻址 ldmia r1!, {r2-r7, r12}

(7)堆栈寻址 stmfd sp!, {r2-r7, lr}

(8)相对寻址 beq flag

flag:

### 1.13.4 ARM汇编特点3: 指令后缀 B H S

同一指令经常附带不同后缀，变成不同的指令。经常使用的后缀有：

B（byte）功能不变，操作长度变为8位

H（half word）功能不变，长度变为16位

S（signed）功能不变，操作数变为有符号 如 ldr ldrb ldrh ldrsb ldrsh

S（S标志）功能不变，影响CPSR标志位 如 mov和movs movs r0, #0

### 1.13.5 ARM汇编特点4：条件后缀

EQ/NE:等于/不等于(equal / not equal)

HS/LO:无符号数高于或等于/无符号数小于(higher or same/lower)

HI/LS:无符号数高于/无符号数低于或等于(higher/lower or same)

GE/LT:有符号数大于或等于/有符号数小于(greater or equal/less than)

GT/LE:有符号数大于/有符号数小于或等于(greater than/less or equal)

MI/PL:负/非负

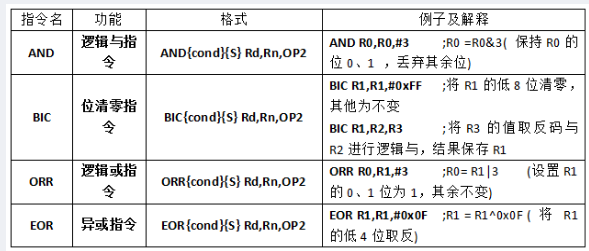
VS/VC:溢出/不溢出(overflow set / overflow clear)

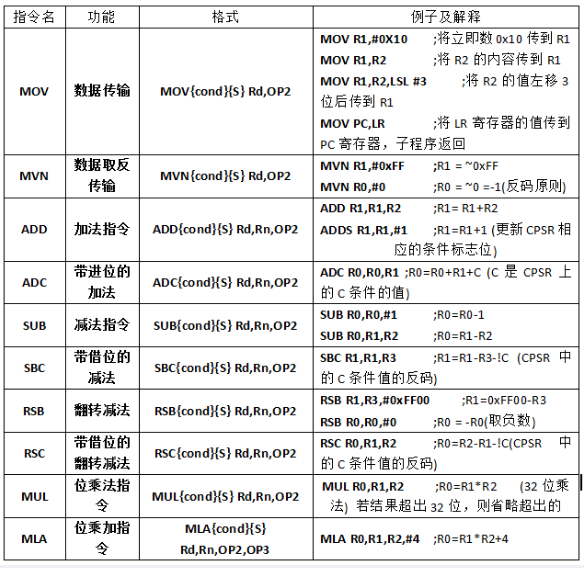
CS/CC:进位/无进位(carry set / carry clear)



## 1.14 ARM汇编指令2-常用指令

### 1.14.1常用ARM指令1：数据处理指令





数据传输指令 mov mvn

算术指令 add sub rsb adc sbc rsc

逻辑指令 and orr eor bic

比较指令 cmp cmn tst teq

乘法指令 mvl mla umull umlal smull smlal

前导零计数 clz

数据传输指令：

mov r1, r0 @两个寄存器之间数据传递

mov r1, #0xff @将立即数赋值给r1

mvn和mov用法是一样的，区别是mvn是按位取反后传递。

逻辑指令：

and:逻辑与

orr:逻辑或

eor:逻辑异或

bic:位清除指令 bic r0, r1, #0x1f @将r1中的数的bit0~bit4清零后赋值给r0

比较指令：比较指令用来比较2个寄存器中的数。

（注意：比较指令不用加s后缀就可以影响cpsr中的标志位，之所以这样做的原因是重在过程，而不像mov那种注重结果。例如cmp r0, r1 这句执行之后，下句就可以通过cpsr中的z位为1来知道上句相等。）

cmp: cmp r0, r1 等价于 sub r2, r0, r1 (r2 = r0 - r1)

cmn: cmn r0, r1 等价于 add r0, r1

tst: tst r0, #0xf @测试r0的bit0~bit3是否全为0

teq:

### 1.14.2常用ARM指令2：CPSR访问指令

mrs & msr

mrs用来读cpsr/spsr，msr用来写cpsr/spsr

CPSR寄存器比较特殊，需要专门的指令访问，这就是mrs和msr。

### 1.14.3常用ARM指令3：跳转（分支）指令

b & bl & bx

b 直接跳转（就没打开算返回）

bl branch and link，跳转前把返回地址放入lr中，以便返回，以便用于函数调用

bx跳转同时切换到ARM模式，一般用于异常处理的跳转。

### 1.14.4常用ARM指令4：访存指令

ldr/str & ldm/stm & swp(内存和寄存器互换指令)

单个字/半字/字节访问 ldr/str

多字批量访问 ldm/stm m(multiple 多重的)

swp r1, r2, [r0] 把r0作为地址里边的值读取到r1里边，把r2里边的值写回r0里边。

swp r1, r1, [r0] 把r0作为地址里边的值读取到r1里边，把r1原来的值写回r0里边。

### 1.14.5常用ARM指令5

swi（software interrupt）

软中断指令用来实现OS中系统调用

### **1.14.6 ARM汇编中的立即数**

ARM指令都是32位的，除了指令标记和操作标记外，本身只能附带很少位数的立即数。因此立即数有合法和非法之分。

合法立即数：经过任意位数的移位后非0部分可以用8位表示的即为合法立即数。

合法立即数：0x000000ff 0x00ff0000 0xf000000f

非法立即数：0x000001ff

## 1.15 ARM汇编指令3-CP15协处理器

### 1.15.1协处理器

SoC内部另一处理核心，协助主CPU实现某些功能，被主CPU调用执行一定任务。

ARM设计上支持多达16个协处理器，但是SoC只实现其中的CP15。(coprocesser)

协处理器和MMU、cache、TLB等处理有关，功能上和操作系统的虚拟地址映射、cache管理有关。

### 1.15.2协处理器CP15操作指令

mrc & mcr

mrc用来读取CP15中的寄存器，mcr用来写入CP15中的寄存器

### 1.15.3 mrc & mcr的使用方法

mcr{<cond>} p15, <opcode\_1>, <Rd>, <Crn>, <Crm>, {<opcode\_2>}

opcode\_1：对于cp15永远为0

Rd：ARM的普通寄存器

Crn：cp15的寄存器，合法值是c0～c15

Crm：cp15的寄存器，一般均设为c0

opcode\_2：一般省略或为0

mrc p15, 0, r0, c1, c0, 0

orr r0, r0, #1

mcr p15, 0, r0, c1, c0, 0

功能：将CP15协处理器中的c1寄存器值读取到r0中，将bit0置1，然后再写入到c1中。

## 1.16 ARM汇编指令4-多寄存器指令

### 1.16.1多寄存器访问指令

ldr/str每周期只能访问4字节内存，如果需要批量读取、写入内存的话太慢，解决方案就是ldm/stm，ldm(load register multiple)，stm(store register multiple)

举例：

stmia sp, {r0 - r12}

将r0存入sp指向的内存处（假设为0x30001000）；然后地址+4（即指向0x30001004），将r1存入该地址；然后地址再+4（指向0x30001008），将r2存入该地址······直到r12内容放入（0x3001030），指令完成。

一个访存周期同时完成13个寄存器的读写

### 1.16.2 18种后缀：stmia stmfd

ia（increment after） 先传输，再地址+4 相当于空递增堆栈 ea

ib（increment before） 先地址+4，再传输 相当于满递增堆栈 fa

da（decrement after） 先传输，再地址-4 相当于空递减堆栈 ed

db（decrement before） 先地址-4，再传输 相当于满递减堆栈 fd

fd（full descending） 满递减堆栈

ed（empty descending） 空递减堆栈

fa（full ascending） 满递增堆栈

ea（empty ascending） 空递增堆栈

### **1.16.3四种栈**

空栈：栈指针指向空位，每次存入时可以直接存入然后栈指针移动一格；而取出时需要先移动一格才能取出

满栈：栈指针指向栈中最后一格数据，每次存入时需要先移动栈指针一格再存入；取出时可以直接取出，然后再移动栈指针

增栈：栈指针移动时向地址增加的方向移动的栈

减栈：栈指针移动时向地址减小的方向移动的栈

### 1.16.4 !的作用

ldmia r0, {r2 - r3}

ldmia r0！, {r2 - r3}

!的作用就是r0的值在ldm过程中发生的增加或者减少最后写回到r0去，也就是说ldm时的感叹号会改变r0的值。

### 1.16.5 ^的作用

ldmfd sp!, {r0 - r6, pc}

ldmfd sp!, {r0 - r6, pc}^

^的作用：在目标寄存器中有pc时，会同时将spsr写入到cpsr，一般用于从异常模式返回。

* 谨记：操作栈时使用相同的后缀就不会出错，不管是满栈还是空栈、增栈还是减栈

## 1.17 ARM汇编伪指令

### 1.17.1伪指令的意义

伪指令不是指令，伪指令和指令的根本区别是经过汇编后不会生成机器码。

伪指令的意义在于指导汇编过程。

伪指令是和具体的汇编器有关的，我们使用gnu工具链，因此学习gnu下的汇编伪指令

### 1.17.2 gnu汇编中的一些符号

@ 用来做注释。

: 以冒号结尾的是标号

. 点号在gnu汇编中表示当前指令的地址 （b . 这句表示死循环）

# 立即数前要加#或$，表示这是个立即数

### 1.17.3常用gnu伪指令

.global\_start @给\_start外部链接属性，意思是在其他文件可以调用\_start。

.section.text @指定当前段为代码段

.ascii .byte .word 这三个最常用的 相当于声明变量为char byte int

.short .long .quad .float .string @定义数据

IRQ\_STACK\_START:

.word 0x0badc0de

等价于 unsigned int IRQ\_STACK\_START = 0x0badc0de;

.align4 @以16字节对齐 （2exp4）

.balignl 16 0x1234567f @16字节对齐填充

b表示位填充；align表示要对齐；l表示long，以4字节为单位填充；16表示16字节对齐；0xdeadbeef是用来填充的原料。

0x00000008: .balignl 16, 0xdeadbeef

0x0000000c 0xdeadbeef

0x00000010: 下一条指令

### 1.17.4偶尔会用到的gnu伪指令

.end @标识文件结束

.include @头文件包含

.arm / .code32 @声明以下为arm指令

.thumb / .code16 @声明以下为thumb指令

### 1.17.5最重要的几个伪指令

ldr 大范围的地址加载指令

adr 小范围的地址加载指令

adrl 中等范围的地址加载指令

nop 空操作

ARM中有一个ldr指令，还有一个ldr伪指令

一般都使用ldr伪指令而不用ldr指令（因为使用ldr指令还要考虑立即数的合法问题，而使用ldr伪指令会在汇编过程中帮忙处理）

ldr指令: ldr r0, #0xff

ldr伪指令: ldr r0, =0xfff1 @涉及到合法/非法立即数，涉及到ARM文字池

adr和ldr的差别：ldr加载的地址在链接时确定，而adr加载的地址在运行时确定；所以可以通过adr和ldr加载的地址比较来判断当前程序是否在链接时指定的地址运行。

# 二、GPIO和LED

## 2.1 Makefile

### 2.1.1 Makefile的一些基本概念

目标 : 依赖

命令

### 2.1.2 Makefile的基本工作原理

（1）当我们make xx时，Makefile会自动执行xx这个目标下的命令语句。

（2）当我们make xx时，是否执行命令是取决于依赖的，依赖如果成立才可以执行。

（3）当直接make时，和make 第一个目标 效果是一样的。第一个目标实际就是默认目标。

### 2.1.3 make的依赖性

make会一层又一层的去找文件的依赖关系。直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么make就会退出，并报错，而对于所定义的命令的错误或是编译不成功，make根本不care。make只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后边的文件还是不在，那么对不起，我就不工作啦。

* 具体学习参考《跟我一起写Makeflie》陈晧

## 2.2 mkv210\_image.c详解

### 2.2.1 Makefile文件

led.bin: start.o

arm-linux-ld -Ttext 0x0 -o led.elf $^

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

arm-linux-objdump -D led.elf > led\_elf.dis

gcc mkv210\_image.c -o mkx210

./mkx210 led.bin 210.bin

%.o : %.S

arm-linux-gcc -o $@ $< -c

%.o : %.c

arm-linux-gcc -o $@ $< -c

clean:

rm \*.o \*.elf \*.bin \*.dis mkx210 -f

解释：真正项目找的Makefile是把程序的编译和链接过程分开的。（我们平常gcc a.c -o exe这种方式来编译时，实际上把编译和链接过程一步完成了。在内部实际上编译和链接永远是分开独立进行的，编译要使用编译器gcc，链接要使用链接器ld。）

链接器得到的led.elf其实就是我们的可执行程序，（如果在OS下，这个led.elf就可以执行了）但是在嵌入式裸机中，我们需要的是可以烧写的文件（可烧写的文件就叫镜像image），因此我们需要用这个led.elf作为原材料来只做镜像，制作工具就是交叉编译工具链中的arm-linux-objcopy

我们使用arm-linux-objdump工具进行反编译（反汇编），反汇编其实就是把编译后的elf格式的可执行程序给反过来得到对应的汇编程序，得到它的汇编源代码。

* mkv210\_image.c这个程序其实最终不是在开发板上执行，而是在主机Linux中执行的，因此编译mkv210\_image.c时用gcc而不是用arm-linux-gcc。mkv210\_image.c编译链接后得到可执行文件，这个可执行文件可以将usb启动的可执行的烧录文件（led.bin）加工为SD卡启动的可执行的烧录文件(210.bin)。
* 当210作为裸机来使用时，所写的程序即相当于BL1（SD卡启动的话包含BL1头文件（16字节），USB启动的话不包含BL1头文件（16字节））。

210带OS启动过程回顾：SoC片上iROM运行已固化的BL0程序，BL0初始化FLASH，从FLASH中读取BL1加载到SoC片上iRAM运行BL1，BL1从FLASH加载BL2到iRAM运行，BL2初始化DRAM，从FLASH加载OS到DRAM运行。

作为裸机实验：

细节：BL0运行，通过OMpin判断启动通道（USB or SD），如果是SD卡启动默认从板载eMMC卡读取BL1，如果板载eMMC卡BL1的第一个扇区被破坏则从SD通道2的外部SD卡读取BL1。读取到BL1后会做Checksum（需要校验的内存区域16KB-16Bytes中，所有内存中的内容按照字节为单位来进行相加，最终相加的和极为校验和，mkv210\_image.c将校验和放在BL1头文件第3个字（即0xd0020008--0xd002000c）中,BL0也会计算校验和，通过BL0计算的校验和和BL1头文件中的校验和对比即可得知BL1是否正确），判断BL1是否正确，如果正确则加载BL1到iRAM运行BL1程序（这里的BL1程序指的是16个字节的BL1头文件+我们写的程序+空0）；如果是从USB启动，则读取到BL1程序后不做Checksum，所以不care 16个字节的BL1头文件，所以我们的程序就不需要包含BL1头文件。我们把所写的程序放到iRAM 0xd0020010运行即可。之所以要使用 mkv210\_image.c，就是要把我们写的程序加上16个字节的BL1头文件烧录到SD卡来启动。从SD卡启动方式对应BL1程序有16KB(16字节BL1头文件+led.bin+空0）；从USB启动方式对应BL1程序只有led.bin。

### 2.2.2 mkv210\_image.c程序工作流分析

整个程序中首先申请一个16KB的buffer，然后把所有内容按照各自的位置填充进去，最终把填好的buffer写入到一个文件（名叫210.bin）就形成了我们想要的镜像。

### 2.2.3 mkv210\_image.c代码详解

<mkv210_image.c>

第1步：检验用户传参是不是3个。

第2步：分配16KB buffer并且填充为0.

第3步：打开源bin（led.bin），判断bin长度是否不大于16KB-16Bytes

第4步：以16个字符串填充0~15这16个Bytes

第5步：将源bin从第17个Bytes开始填充，直到整个内容为16KB

第6步：从17个Bytes开始以字节为单位统计校验和

第7步：将校验和保存到16个字节的8~12个字节中

第8步：以二进制写的方式创建一个新的二进制文件（210.bin）

第9步：将上方创建的16KB整个内容完全填充到新bin（210.bin）中

main函数两个形参的作用:

main函数接收2个形参：argc和argv。

argc是用户（通过命令行来）执行这个程序时，实际传递的参数个数。注意这个个数是包含程序执行本身的

argv是一个字符串数组，这个数组中存储的字符串就是一个个的传参。

譬如我们执行程序时使用./mkx210 led.bin 210.bin

则argc = 3

则argv[0] = "./mkx210" argv[1] = led.bin argv[2] = 210.bin

glibc读写文件接口:

linux中要读取一个文件，可以使用fopen打开文件，fread读取文件，读完之后fclose关闭文件。要写文件用fwrite来写。这些函数是glibc的库函数，在linux中用man 3 可以查找。如果你本身就知道这些函数的用法，只是记不起来可以man查找；如果你本身根本就不会用这些接口，建议先去baidu。

mkv210\_image.c的疑问：

S5PV210启动时会从外扩存储器（nand /sd/usb）中拷贝16K代码到iRAM（0xd002\_0000，其中0xd002\_0010之前的16个字节储存的BL1的校验信息和BL1尺寸）中执行。其中用户手册中注明16字节头部信息排列格式如下：

0x0地址：BL1 size

0x4地址：必须设置为0 (是规定)

0x8地址：CheckSum

0xc地址：必须设置为0 (是规定)

但是，uboot中给出的参考代码中，首先将该16字节填充为"S5PC110 HEADER "，然后替换0x8地址的内容，并没有和用户手册给出的16字节头部信息对应，这是为什么呢？

## 2.3一步步点亮LED

* 软件编程控制硬件的方式就是通过硬件的寄存器来控制。

### 2.3.1写程序的一般步骤

(1)了解硬件原理

(2)查询电路原理图，找到相应硬件

(3)查询数据手册，找到相应寄存器

(4)据工程需求，通过代码配置相关寄存器

### 2.3.2 GPIO:(General Purpose Input Output)通用输入输出

GPIO就是芯片的引脚（芯片上的引脚有些不是GPIO，只有一部分是），作为GPIO的这类引脚，他的功能和特点是可以被编程控制它的工作模式，也可以编程控制他的电压高低等。

### 2.3.3点亮LED相关寄存器

GPJ0CON/GPJ0DTA、GPD0CON/GPD0DTA，我们要做的就是设置寄存器的值，控制相关引脚的工作模式和数据内容（电平高低）。

### 2.3.4汇编编写及调用函数的方式

汇编中整个汇编的主程序是一个死循环，这个死循环是我们汇编程序的主体，类似于C中的main函数。其他函数必须写在这个主死循环程序的外边，不然会出错。

汇编编写delay延时函数时，要注意函数的初始化和循环体的位置，不能把初始化写在了循环体内。

汇编中调用函数用bl指令（这句指令的功能是跳转之前先将当前pc的值写入到lr寄存器保存，以便函数返回时，继续从lr保存的地址处开始执行原程序。），子函数中最后用mov pc, lr来返回。（bl经常用回函数调用）

## 2.4反汇编工具objdump的使用简介

### 2.4.1反汇编的原理&为什么要用反汇编

arm-linux-objdump -D led.elf > led\_elf.dis

objdump是gcc工具链中的反汇编工具，作用是由编译链接好的elf格式的可执行程序反过来得到汇编源代码

-D表示反汇编

> 左边的是elf可执行程序（反汇编的源）

> 右边的是反汇编生成的反汇编程序

反汇编的原因有以下：

1）逆向破解。

2）调试程序时，反汇编可以帮助我们理解，尤其是在理解链接脚本、链接地址等概念时。

3）把C语言源代码编译链接生成的可执行程序反汇编后得到的反汇编代码可以帮助我们理解C语言和汇编语言之间的对应关系。非常有助于深入理解C语言。

### 2.4.2反汇编文件的格式和看法

（汇编 assemnbly 反汇编 dissembly）

标号地址、标号名字、指令地址、机器码、指令机器码及汇编到的指令

扩展：ARM汇编中用地址池方式来实现非法立即数

### 2.4.3初识指令地址

下载烧录执行的bin文件，内部其实就是一条一条的指令机器码。这些指令每一条都有一个指令地址，这个地址时链接的时候ld给指定的（ld根据我们写的链接脚本来指定）

### 2.4.4展望：反汇编工具帮助我们分析链接脚本

反汇编的时候得到的指令地址是链接器考虑了链接之后得到的地址，而我们写代码时通过指定的链接脚本来让链接器给我们链接合适的地址。但是有时候我们写的链接脚本有误（或者我们根本不知道这个链接脚本会怎么样），这时候可以通过反汇编文件来分析这个链接脚本的效果。看是不是我们想要的，如果不是，再改再看。

# 三、SDRAM和重定位relocate

## 3.1汇编写启动代码之关看门狗

### 3.1.1什么是看门狗？

看门狗（watch dog timer 看门狗定时器）。大家想象这样一个场景：家门口有一只狗，这个狗定时会饿（譬如说2小时一饿），够饿了会胡乱咬死人。人进进出出要想保证安全必须提前喂狗（必须在上次喂过后的2小时内喂狗才行）。如果超时没喂狗就会被咬死，如果提前喂狗没关系，但是本次喂狗时间就会从这里开始计算。

现实中因为一些外部因素，电子设备经常会跑飞或者死机（譬如极端炎热、极端寒冷、工业复杂场合）。在这种情况下我们希望设备自动复位而不需要人工干预（无人值守）。看门狗用来完成这个工作。看门狗其实是我们SoC内部的一个定时器（类似于闹钟，类似于门口的狗），定好时间之后看门狗定时器会去计时，时间到之前（狗饿了之前）必须去重新置位看门狗定时器（喂狗），如果没有喂狗则系统会被强制复位。

系统在正常工作时，系统软件会自己去喂狗，所以看门狗定时器不会复位。但是系统一旦故障跑飞啥的，看门狗就没人喂了，然后下一个周期就会自动复位，达到我们期望的效果。

### 3.1.2分析硬件物理特性、原理图、数据手册

物理特性上看门狗其实是个定时器（跟现实中的闹钟类似），硬件上就是SoC内部的一个内部外设。

原理图：看门狗不用分析原理图，因为看门狗属于内部外设，且没有外部相关的原件与他有关，所以不需要原理图分析，原理图上根本找不到和看门狗有关的地方。

数据手册：在数据手册的Section7.3，大家可以详细来看。如果直接看不懂数据手册，可以百度看门狗，然后看别人的博客来学习。

### 3.1.3找到关键性操作SFR

WTCON（0xE2700000），其中bit5是看门狗的开关：0代表关，1代表开

### 3.1.4总结210中看门狗特性（iROM中已经关看门狗）

为什么要关看门狗？

一般CPU设计，在CPU启动后看门狗默认是工作的（为什么默认不关闭而要工作？我猜测是因为怕你的程序在启动代码前端就死机了或者跑飞了没人管），好处就是没有空当和漏洞，坏处就是在启动代码段我们不方便去喂狗（或者说懒得去喂狗）时看门狗会复位，所以为了偷懒我们就在启动代码前端先去关闭看门狗，然后在后面系统启动起来之后再根据需要决定是否要打开看门狗（一旦打开就必须同时提供喂狗）。

在S5PV210内部的iROM代码（BL0）中，其实已经关过看门狗了。所以我们的启动代码实际上是不用去关也没事的，也就是说今天写的关闭看门狗的代码运行后没有任何现象（没有现象就是正常现象）.

很多CPU内部是没有BL0的，因此也没人给你关看门狗，都要在启动代码前段自己写代码关看门狗，所以这部分学习的内容也是有价值的。

## 3.2汇编写启动代码之设置栈和调用C语言

### 3.2.1 C语言运行时需求和栈的意义

“C语言运行时（runtime）”需要一定的条件，这些条件由汇编来提供。C语言运行时主要是需要栈。

C语言和栈的关系：C语言中的局部变量都是用栈来实现的。如果我们汇编部分没有给C部分预先设置合理合法的栈地址，那么C代码中定义的局部变量就会落空，整个程序就会死掉了。（C语言需要汇编设置栈空间来实现局部变量）

我们编写单片机程序时（譬如51单片机）或者编写应用程序时并没有设置栈，但C程序还是可以运行。原因是：在单片机中由于硬件初始化时提供了一个默认可用的栈；在应用程序中我们编写的C程序其实并不是全部，编译器（gcc）在链接时会帮我们自动添加一个头，这个头就是一段引导我们C程序能够执行的一段汇编实现的代码，这个代码中就帮我们的C程序设置了栈及其他的运行时需要。

### 3.2.2 CPU的7种工作模式和各种模式下的栈

在ARM的7种工作模式下，每种工作模式都有自己独立的SP寄存器（r13）。这种设计的原因是：

如果各个模式都使用同一个sp，那么就意味着整个程序（操作系统内核程序、用户自己编写的应用程序）都是同一个栈。你的应用程序如果一旦出错（譬如栈溢出），就会连累操作系统的栈也损坏，整个OS的程序就会崩溃。这样的OS设计是非常脆弱的，不合理的。

解决方案就是各种模式下用不同的栈。我们的操作系统内核使用自己的栈，应用程序也使用自己独立的栈，这个各是各的，互不影响。

我们现在要设置栈，要先找到自己的模式，然后设置自己的模式下的栈到合理合法的位置即可。

注意：系统在复位后默认是进入SVC模式的

我们如何访问SVC模式下的SP呢？先把模式设置为SVC，再直接操作SP。但因为系统复位后就是SVC模式，所以直接设置SP即可。

### 3.2.3查阅文档并设置栈指针至合法位置

栈必须是当前一段可用内存（可用的意思就是这个地方必须有被初始化过可用访问的内存，而且这个内存只会被我们用作栈，不会被其他程序征用）

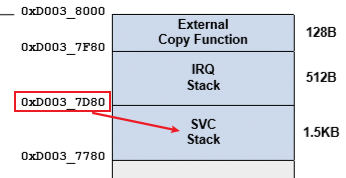
当CPU刚复位或启动，外部DRAM尚未初始化，目前可用的内存只有iSRAM（因为它不需要初始化即可使用）。因此我们只能在iSRAM中找一段内存来作为SVC的栈。

在ARM中，ATPCS（ARM关于程序应该怎么实现的一个规范）要求使用满减栈，所以基本都是满减栈。。

满减栈：

进栈：先向下移动指针再存放数据 出栈：先出数据再向上移动指针

结合iROM\_application\_note中的iRAM的memery map，可知SVC栈应该设置为0xD0037D80



### 3.2.4汇编程序和C程序的互相调用

bl cfunction

### 3.2.5 C函数的编写和被汇编调用

在工程中新建并添加一个C源文件（led.c），注意添加时要修改Makefile

在汇编启动代码中设置好栈后用bl xxx的方式来调用C中的函数xxx

### 3.2.6使用C语言来访问寄存器的方法

寄存器的地址类似于内存地址（I/O与内存统一编址的），所以这里的问题是用C语言读写寄存器，就是C语言来读写内存地址。用C语言来访问内存，就要用到指针。

unsigned int \*p = (unsigned int \*)0xE0200240;

\*p = 0x11111111;

简化为：

\*((unsigned int \*)0xE0200240) = 0x11111111;

### 3.2.7神奇的volatile

volatile的作用是让程序在编译时，编译器不对程序做优化。优化有时候是OK的，但有时候是自作聪明。如果你的一个变量是易变的，不希望编译器帮忙优化，就在这个变量定义时加volatile。

### 3.2.8总结

C和汇编函数的互相调用（函数名和汇编标号的真实意义）

C语法对内存访问的封装方式（使用指针来访问内存的技巧）

汇编的意义（起始代码&效率关键部位）

## 3.3汇编写启动代码之开关iCache

### 3.3.1什么是cache，有什么用

cache是一种内存，叫高速缓存。

从容量来说：CPU < 寄存器 < cache < DDR

从速度来说：CPU > 寄存器 > cache > DDR

cache的存在，是因为寄存器和DDR之间速度差异太大了，DDR的速度远不能满足寄存器的需要（不能满足CPU的需要，所以没有cache会拉低整个系统的整体速度）

整个系统CPU的供应链由：寄存器+cache+DDR+固态硬盘/硬盘/Flash四阶组成，这是综合考虑了性能、成本后得到的妥协的结果。

210内部有32KB icache和 32KB dcache。icache是用来缓存指令的；dcache是用来缓存数据的。

cache的意义：指令平时放在硬盘/Flash中，运行时读到DDR中，再从DDR中读给寄存器，再由寄存器送给CPU。但是DDR的速度和寄存器（代表的CPU）相差太大，如果CPU运行完一句再去DDR读取下一句，那CPU的速度就完全被DDR拖累了。解决方案就是icache。

icache工作时，会把我们CPU正在运行的指令的旁边几句指令实现给读取到icache中（CPU设计有一个基本原理：代码执行时，下一句执行当前一句代码旁边的代码几率较高），当下一句CPU要指令时，icache首先会检查自己事先准备的缓存指令中有无这句，如果有直接给CPU，如果没有则需要从DDR中重新读取拿给CPU，并同时做一系列动作：清缓存、重新缓存。

### 3.3.2 iROM中BL0对cache的操作

首先，icache的一切动作都是自动的，不需人为干预。我们所要做的就是打开/关闭icache。

其次，在210的iROM的BL0已经打开了icache。所以之前看到的现象都是icache打开时的现象。

### 3.3.3汇编代码读写cp15以开关icache

mrc p15,0,r0,c1,c0,0; // 读出cp15的c1到r0中

bic r0, r0, #(1<<12) // bit12 置0 关icache

orr r0, r0, #(1<<12) // bit12 置1 开icache

mcr p15,0,r0,c1,c0,0;

### 3.3.4实验验证

我们来看三种情况下的实验现象：

(1)直接使用BL0中对icache的操作

(2)关icache

(3)开icache

实验结果分析：

结论1：irom中确实是打开了icache的。

结论2：icache关闭确实比icache打开时led闪烁变慢，说明指令执行速度变慢。

## 3.4重定位引入和链接脚本

### 3.4.1一个事实：大部分指令是位置有关编码

位置无关编码（PIC,position independent code）：汇编源文件被编码成二进制可执行程序时编码方式与位置（内存地址）无关。在我们写程序时，必须给链接器指定地址。将来的程序被执行时必须放在当时链接时给定的地址才能运行。

位置有关编码：汇编源码编码成二进制可执行程序后和内存地址是有关的。但是也有一种特别的指令他可以跟指定的链接地址没有关系，这些代码不管放在哪里都可以正常运行。

分析：

Ⅰ.之前的裸机程序中，makefile中用-Ttext 0x0来指定链接地址是0x0;这意味着我们认为将来这个程序会被放在0x0这个地址去运行。但是实际上我们在DNW中，把程序下载到0xd0020010（在s5pv210中，由于在内部做了映射，所以0xd002\_0010和0x0000\_0010是一样的）。那么为什么放在0x0而不是0x0000\_0010依旧可以呢?原因就这个是位置无关编码。

☆链接地址：链接时指定的地址（指定方式：makefile中用T-text,或者链接脚本）

☆运行地址：程序实际运行的地址（指定方式：由实际运行时被加载到内存那个位置说了算）

Ⅱ.在linux中的应用程序：gcc hello.c -o hello

☞ 这时候默认的链接地址就是0x0，所以链接在0地址，因为应用程序运行在操作系统的一个进程中，这个进程独享了4G的内存空间，所以应用程序可以链接到0地址，因为每个进程都是从0地址开始的。

☞ 210中的裸机程序运行地址由我们下载时确定，下载时下载到0xd0020010，所以就从这里开始运行。（这个下载地址是iROM中的BL0加载BL1时实现指定好的地址，这个是由CPU设计时候决定的）。所以理论上我们编译链接时应该指定到0xd0020010,但是实际上我们之前的裸机程序都是使用位置无关码PIC，所以链接地址可以是0（随意）

对比：位置无关编码链接地址和运行时地址可以不同，并不影响程序的正常运行；位置有关编码的链接地址和运行时地址必须相同，也就是程序必须放在我们所指定的运行地址处运行。

### 3.4.2再接S5PV210的启动过程：三星推荐和uboot的实现是不同的

***三星推荐方式***：

bootloader必须大于16KB小于96KB。假定bootloader为74KB，启动过程：开机，BL0运行，BLO加载外部启动设备中的bootloader前16KB（BL1）到SRAM中运行，BL1运行时会加载BL2(bootloader中74-16=58KB)到SRAM中（从SRAM的16KB处开始用）去运行；BL2运行时会初始化DRAM并且将OS搬运到DRAM中去执行OS，启动完成

***uboot方式***：

uboot大小随意，假定为200KB，启动过程：开机，BL0运行，BL0加载外部启动设备中的bootloader（uboot）的前16KB（BL1）到SRAM中运行，BL1运行时会初始化DRAM，然后将整个bootloader（uboot，200KB）搬运到DRAM中，然后用一句长跳转指令从iSRAM中直接跳转到DRAM中继续执行bootloader（uboot）直到bootloader(uboot)完全启动。bootloader（uboot）启动后在bootloader（uboot）命令行中去启动OS。

✌题外话：三星为什么推荐BL1+BL2的方式而不使用uboot方式？

因为实际项目中，bootloader的大小并不会像uboot那么大，因为uboot中做了很多没有用的工作，实际的项目启动代码只需要满足定制化需求即可，其大小是在96KB范围之内的，因此使用BL1+BL2方式也省的使用代码重定位了。

### 3.4.3为什么要有重定位?

**原因：**链接地址和运行时地址有时候必须不同，而且还不能全部用位置无关码，这时候只能重定位。（运行地址空间（96KB）不够uboot（200KB）的使用，因此链接地址和运行时地址必须不同，解决方案就是使用长跳转指令将程序运行从iSRAM中解脱出来，放在更大的空间DRAM中去运行。这就是所谓的代码重定位。）

**扩展：**分散加载：把uboot分成2部分（BL1和整个uboot），两部分分别指定不同的链接地址。启动时将两部分加载到不同的地址（BL1加载到SRAM，整个uboot加载到DDR），这时候不用重定位也能启动。

**评价：**分散加载其实相当于手工重定位。重定位是用代码来进行重定位，分散加载是手工操作重定位的。

### 3.4.4运行时地址由什么决定？

运行时地址是由运行时决定的（编译链接时是无法绝对确定运行时地址的）

### 3.4.5链接地址由什么决定？

链接地址是由程序员在（编译）链接的过程中，通过Makefile中-Ttext xxx或者在链接脚本中指定的。程序员事先会预知自己的程序的执行要求，并有一个期望的运行地址，并会用这个地址来作为链接地址。

### 3.4.6从源码到可执行程序的步骤：预编译、编译、链接、（strip、objcopy）等

预编译：预编译器执行，譬如C语言的宏定义、注释等。

编译：编译器执行，包括编译和汇编。把源码.c .S编成机器码.o文件。

链接：链接器执行，把.o文件中的各函数（段）按照一定的规则（链接脚本指定）累积在一起，形成可执行程序。

strip：strip是把可执行程序中的符号信息（函数名等）拿掉，以节省空间。（Debug版本和Release版本）。

objcopy：由可执行elf文件生成可以烧录的镜像bin文件。

### 3.4.7程序段的概念：代码段、数据段、bss段（ZI段）、自定义段

段就是程序的一部分，我们把整个程序的所有东西都分成了一个一个的段，给每个段都起个名字，然后再链接时就可以使用这个名字指示这些段。

段分为两种：一种是编译器链接器内部定好的，先天性段名；一种是人为指定的、自定义段名。

**先天性段名**：

• 代码段：（.text），又叫文本段，代码段其实就是函数编译后生成的东西

• 数据段：（.data），数据段就是C语言中有显示初始化为非0的全局变量

• bss段：（.bss），又叫ZI(zero initial)段，就是零初始化段，就是C语言中初始化为0的全局变量。

**后天性段名**：

段名、段的属性和特征都是由程序员自定义的。

分析一些问题，跟这里结合，然后试图明白一些本质：

(1)C语言中全局变量如果未显式初始化，值是0。本质就是C语言把这类全局变量放在了bss段，从而保证了为0。

(2)C运行时环境如何保证显式初始化为非0的全局变量的值在main之前就被赋值了？就是因为它把这类变量放在了.data段中，而.data段会在main执行之前被处理（初始化）。

### 3.4.8链接脚本究竟要做什么？

链接脚本其实是个规则文件，它是程序员用来指挥链接器工作的。链接器会参考链接脚本，并且使用其中规定的规则来处理.o文件中的那些段（编译后生成.o文件，相当与炒菜的准备阶段，即准备好菜、葱姜蒜、油、调料等），将其链接成一个可执行程序（相当于炒菜阶段，先放油、再放葱姜蒜、再放菜、再放调料等）。

链接脚本的关键内容有2部分：段名 + 地址（程序员指定的运行地址（链接地址））

链接脚本的理解：

SECTION {} 这个是整个链接脚本

. 号在链接脚本中代表当前位置

= 等号代表赋值

SECTIONS

{

. = 0xd0024000;

.text : {

start.o

\*(.text)

}

.data : {

\*(.data)

}

bss\_start = .;

.bss : {

\*(.bss)

}

bss\_end = .;

}

## 3.5代码重定位实战

### 3.5.1任务：在iSRAM中将代码从0xd0020010重定位到0xd0024000

注解：本练习对代码本身运行无实际意义，我们做这个重定位纯粹是为了练习重定位技能。但是某些情况重定位就是必须的，譬如在uboot中。

### 3.5.2思路

（1）通过链接脚本将代码链接到0xd0024000(链接地址)

（2）使用dnw下载时将bin文件下载到0xd0020010(运行时地址)

当代码链接地址设置为0xd0024000时，隐含意思就是我这个代码将来必须在0xd0024000位置才能正确执行。如果运行时地址不是这个地址就会出问题（除非是PIC代码）。重定位就是：在PIC执行完之前（在代码中第一句位置有关码执行之前）必须将整个代码搬移到0xd0024000位置去执行。

（3）代码执行时通过代码前段的少量PIC将整个代码搬移到0xd0024000。

（4）使用一个长跳转指令跳转到0xd0024000处的代码继续执行，重定位完成。

长跳转：跳转指令通过给PC(r15)赋一个新值来完成代码段的跳转执行。长跳转指的是跳转到的地址和当前地址差异比较大，跳转的范围比较广。

当我们执行完代码重定位后，实际上在SRAM中有2份代码的镜像（一份是我们下载到0xd0020010处开头的，另一份是重定位代码复制到0xd0024000处开头的），这两份内容完全相同，仅仅地址不同。重定位之后使用ldr pc, =led\_blink这句长跳转直接从0xd0020010处代码跳转到0xd0024000开头的那一份代码的led\_blink函数处去执行。（实际上此时在SRAM中有2个led\_blink函数镜像，两个都能执行，如果短跳转bl led\_blink则执行的就是0xd0020010开头的这一份，如果长跳转ldr pc, =led\_blink则执行的是0xd0024000开头处的这一份）。这就是短跳转和长跳转的区别。

当链接地址和运行时地址相同时，短跳转和长跳转实际效果是一样的；但是当链接地址不等于运行时地址时，短跳转和长跳转就有差异了。这时候短跳转实际执行的是运行时地址处的那一份，而长跳转执行的是链接地址处那一份。

总结：重定位实际就是在运行地址处执行一段位置无关码PIC，让这段PIC（也就是重定位代码）从运行地址处把整个程序镜像拷贝一份到链接地址处，完了之后使用一句长跳转指令从运行地址处直接跳转到链接地址处去执行同一个函数（led\_blink），这样就实现了重定位之后的无缝连接。

### 3.5.3 adr和ldr伪指令的区别

ldr和adr都是伪指令，区别是ldr是长加载、adr是短加载。

adr指令加载的是运行时地址；ldr指令加载的是链接地址。

（通过反汇编文件可以深入分析adr和ldr的区别）

### 3.5.4重定位（代码拷贝）

重定位就是汇编代码中的copy\_loop函数，代码的作用是使用循环结构来逐句复制代码到链接地址。

复制的源地址是iSRAM的0xd0020010，目标地址是iSRAM的0xd0024000，复制长度是bss\_start减去\_start。

复制的长度就是真个重定位需要重定位的长度，也就是整个程序代码段+数据段的长度。

bss段（0初始化的全局变量）不需要重定位。

### 3.5.5清bss段

清除bss段是为了满足C语言运行时要求（C语言要求显示初始化为0的全局变量或者未显示初始化的全局变量值为0，实际上C语言编译器就是通过清bss段来实现C语言的这个特性的）。一般情况下我们的程序是不需要清bss段的（C语言编译器会自动帮程序添加一段头程序，这段程序会在main函数之前运行，负责清除bss）。但是在代码重定位了之后，因为编译器帮我们附加的代码只是帮我们清除了运行时地址那一份代码中的bss，而未清除重定位地址处开头的那一份代码中的bss，所以重定位之后需要自己去清除bss。

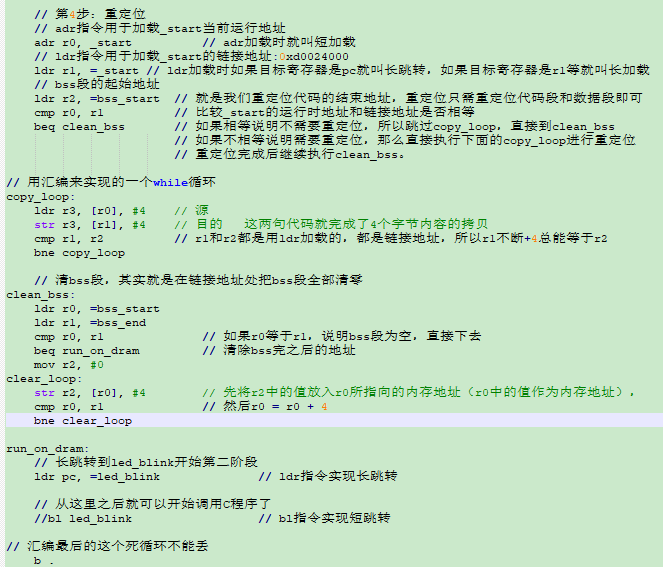
### 3.5.6长跳转

清理完bss段后重定位就结束了，当前的情况就是：

(1)当前运行地址还是0xd0020010开头的（重定位之前的）那一份代码中运行着

(2)此时iRAM中已经有了2份代码，1份在0xd0020010开头，另一份在0xd0024000开头的位置。

然后就要长跳转了。



//重定位

// adr指令用于加载\_start当前运行地址

adr r0, \_start // adr加载时就叫短加载

// ldr指令用于加载\_start的链接地址:0xd0024000

ldr r1, =\_start // ldr加载时如果目标寄存器是pc就叫长跳转，如果目标寄存器是r1等就叫长加载

// bss段的起始地址

ldr r2, =bss\_start // 就是我们重定位代码的结束地址，重定位只需重定位代码段和数据段即可

cmp r0, r1 // 比较\_start的运行时地址和链接地址是否相等

beq clean\_bss // 如果相等说明不需要重定位，所以跳过copy\_loop，直接到clean\_bss

// 如果不相等说明需要重定位，那么直接执行下面的copy\_loop进行重定位

// 重定位完成后继续执行clean\_bss。

// 用汇编来实现的一个while循环

copy\_loop:

ldr r3, [r0], #4 // 源

str r3, [r1], #4 // 目的 这两句代码就完成了4个字节内容的拷贝

cmp r1, r2 // r1和r2都是用ldr加载的，都是链接地址，所以r1不断+4总能等于r2

bne copy\_loop

// 清bss段，其实就是在链接地址处把bss段全部清零

clean\_bss:

ldr r0, =bss\_start

ldr r1, =bss\_end

cmp r0, r1 // 如果r0等于r1，说明bss段为空，直接下去

beq run\_on\_dram // 清除bss完之后的地址

mov r2, #0

clear\_loop:

str r2, [r0], #4 // 先将r2中的值放入r0所指向的内存地址（r0中的值作为内存地址），

cmp r0, r1 // 然后r0 = r0 + 4

bne clear\_loop

run\_on\_dram:

// 长跳转到led\_blink开始第二阶段

ldr pc, =led\_blink // ldr指令实现长跳转

// 从这里之后就可以开始调用C程序了

//bl led\_blink // bl指令实现短跳转

// 汇编最后的这个死循环不能丢

b .

## 3.6 SDRAM引入

### 3.6.1 SDRAM的特性（容量大、价格低、掉电易失性、随机读写、总线式访问）

SDRAM/DDR都属于动态内存（相对于静态内存SRAM），都需要先运行一段初始化代码来初始化才能使用，不像SRAM开机上电后就可以直接运行。类似于SDRAM和SRAM的区别的，还有NorFlash和NandFlash（硬盘）这两个。

正是因为硬件本身特性有限制，所以才导致启动代码比较怪异、比较复杂。而我们研究裸机是为了研究uboot，在uboot中就充分利用了硬件的各种特性，处理了硬件复杂性。

### 3.6.2 SDRAM数据手册K4T1G164QQ

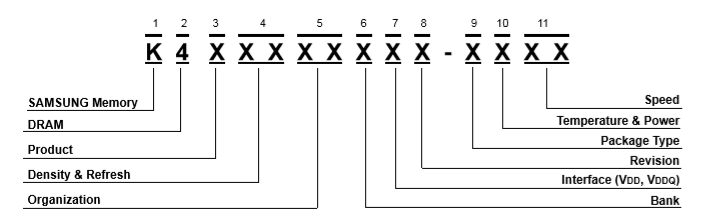
SDRAM在系统中属于SoC外接设备（外部外设。以前说过随着半导体技术发展，很多东西都逐渐集成到SoC内部去了。现在还长期在外部的一般有：Flash、SDRAM/DDR、网卡芯片如DM9000、音频Codec。现在有一些高集成度的芯片也试图把这几个集成进去，做成真正的单芯片解决方案。）

SDRAM通过地址总线和数据总线接口（总线接口）与SoC通信。

K4T1G164QE：

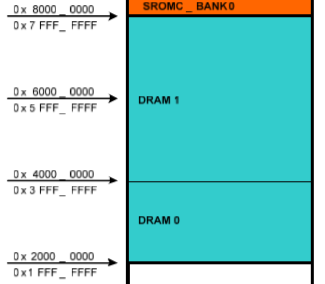
K表示三星产品，4表示是DRAM，T表示产品号码，1G表示容量（1Gb，等于128MB，我们开发板X210上一共用了4片相同的内存，所以总容量是128×4=512MB）16表示单芯片是16位宽的，4表示是8bank

三星官方的数据手册上其实没有芯片相关的参数设置信心，都是芯片选型与外观封装方面的信息，选型是给产品经理来看的，封装和电压等信息是给硬件工程师看的。软件工程师最关注的是工作参数信息，但是数据手册没有。



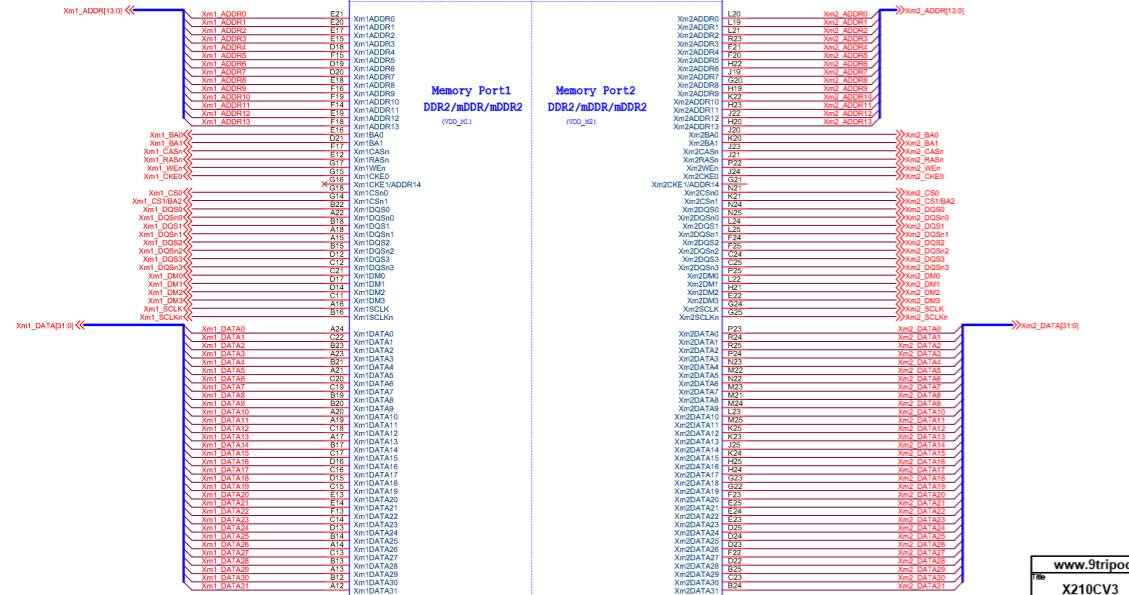
### 3.6.3 SDRAM原理图

S5PV210共有2个内存端口（就好像有2个内存插槽）。再结合查阅数据手册中内存映射部分，可知：两个内存端口分别叫DRAM0和DRAM1：



DRAM0：内存寻址范围：0x20000000~0x3fffffff(512MB)，对应引脚是Xm1xxxx

DRAM1：内存寻址范围：0x40000000~0x7fffffff(1024MB) ，对应引脚是Xm2xxxx



结论：

(1)整个210最多支持内存为1.5GB，如果给210更多的内存CPU就无法识别。

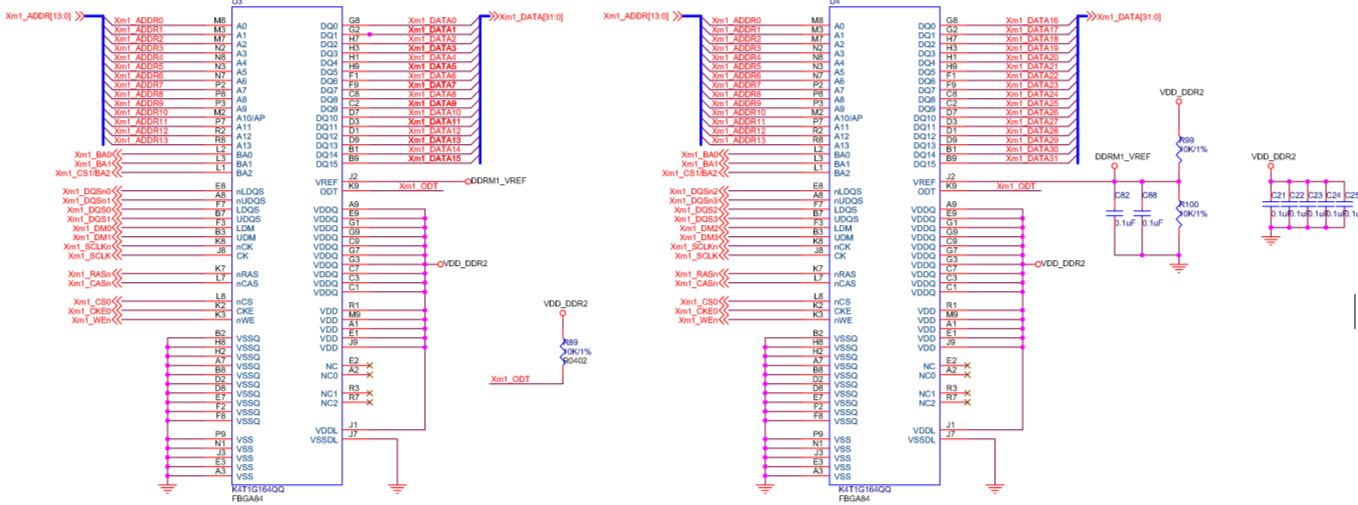
(2)我们所使用的X210开发板只有512MB内存，连接方法是在DRAM0端口分布256MB，在DRAM1端口分布了256MB。

(3)由(2)可知，X210开发板上合法地址是：0x20000000~0x2fffffff(256MB)+0x40000000~0x4fffffff(256MB)。当板子上DDR初始化完成之后，这些地址都是可以使用的；如果使用了其他地址譬如0x30004000就是死路一条。

原理图中每个DDR端口都是由3类总线构成地址总线（Xmn\_ADDR0~XMnADDR13共14根地址总线） + 控制总线（中间部分，自己看原理图） + 数据总线（Xmn\_DATA0~XMnDATA31共32根数据线）

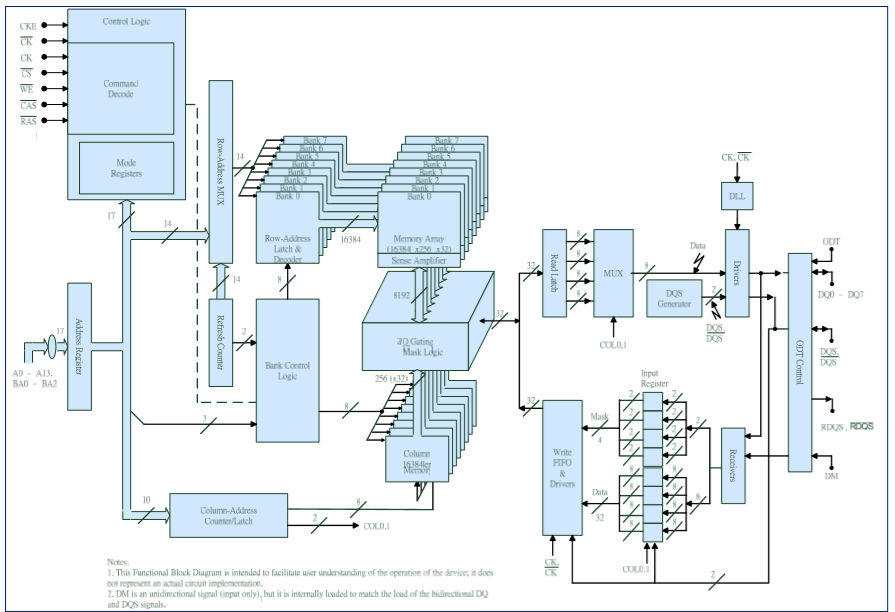
分析：从数据总线的位数可以看出，我们用的是32位的（物理）内存。

原理图中画出4片内存芯片的一页，可以看出：X210开发板共使用了4片内存（每片1Gb=128MB，共512MB），每片内存的数据总线都是16位的（单芯片是16位内存）。如何由16位内存得到32位内存呢？可以使用并联方法。在原理图上横向的2颗内存芯片就是并联连接的。并联时地址总线接法一样，但是数据总线要加起来。这样连接相当于在逻辑上可以把这2颗内存芯片看成是一个（这一个芯片是32位的，接在Xm1端口上）。



### 3.6.4 SDRAM数据手册《NT5TU64M16GG-DDR2-1G-G-R18-Consumer》

第10页的block diagram。这个框图是128Bb×8结构的，这里的8指的是8bank，每bank128Mbit。



210的DDR端口信号中有BA0~BA2，接在内存芯片的BA0~BA2上，这些引脚就是用来选择bank的。

每个bank内部有128Mb，通过row address（14位） + column address（10位）的方式来综合寻址。

一共能寻址的范围是：2的14次方+2的10次方 = 2的24次方。对应16MB（128Mbit）内存。

## 3.7汇编初始化SDRAM详解

### 3.7.1初始化代码框架介绍（函数调用和返回、步骤等）

SDRAM初始化使用一个函数sdram\_asm\_init，函数在sdram\_init.S文件中实现，是一个汇编函数。

强调：汇编实现的函数在返回时需要明确使用返回指令（mov pc, lr）

### 3.7.2 27步初始化DDR2

(1)首先，DDR初始化和SoC（准确说是和SoC中的DDR控制器）有关，也和开发板使用的DDR芯片有关，和开发板设计时DDR的连接方式也有关。

(2)S5PV210的DDR初始化步骤在SoC数据手册：Section 5 1.2.1.3 DDR2这个章节。可知初始化DDR共需27个步骤。

(3)之前分析过X210的内存连接方式是：在DRAM0上连接256MB，在DRAM1上连接了256MB。所以初始化DRAM时分为2部分，第一部分初始化DRAM0，第二部分初始化DRAM1.

(4)我们的代码不是自己写的，这个代码来自于：第一，九鼎官方的uboot中；第二，参考了九鼎的裸机教程中对DDR的初始化；第三，有些参数是我根据自己理解修改过的。

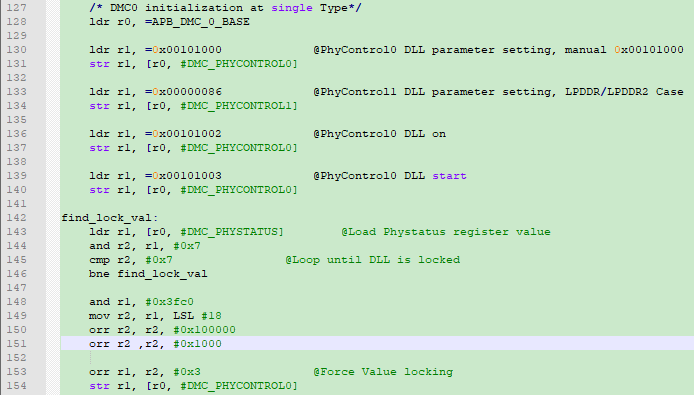
### 3.7.3设置IO端口驱动强度

因为DDR芯片和S5PV210之间是通过很多总线连接的，总线的物理表现就是很多个引脚，也就是说DDR芯片和S5PV210芯片是通过一些引脚连接的。DDR芯片工作时需要一定的驱动信号，这个驱动信号需要一定的电平水平才能抗干扰，所以需要设置这些引脚的驱动能力，使DDR正常工作。

DRAM控制器对应的引脚设置为驱动强度2X（我也不知道为什么是2X，什么时候设置成3X 4X?，这东西只能问DDR芯片厂商或者SoC厂商，我们一般是参考原厂给的代码）

### 3.7.4 DRAM port时钟设置

从代码第128行到154行。主要是开启DLL（dram pll）然后等待锁存。这段代码对应27步中的第2到第4步。



### 3.7.5 DMC0\_MEMCONTROL

bust length = 4， 1chip，... 对应值是0x00202400



### 3.7.6 DMC0\_MEMCONFIG\_0

DRAM0通道中memory chip0的参数设置寄存器



### 3.7.7 DMC0\_MEMCONFIG\_1

DRAM0通道中memory chip1的参数设置寄存器



推论：三星设置DRAM0通道，允许我们接2片256MB的内存，分别叫memory chip0和memory chip1，分别用这两个寄存器来设置它的参数。按照三星的设计，chip0的地址应该是0x20000000到0x2FFFFFFF，然后chip1的地址应该是0x30000000~0x3FFFFFFF.各自256MB。

但是我们X210开发板实际在DRAM0端口只接了256MB的内存，所以只用了chip0，没有使用chip1.（我们虽然是2片芯片，然后这两片是并联形成32位内存的，逻辑上只能算1片）。按照这个推论，DMC0\_MEMCONFIG\_0有用，而DMC0\_MEMCONFIG\_1无用，所以我直接给他了默认值。

### 3.7.8 DMC\_DIRECTCMD

这个寄存器是个命令寄存器，我们210通过这个寄存器写值来向DDR芯片发送命令（通过命令总线），这些命令应该都是用来配置DDR的芯片的工作参数。

总结：DDR配置过程比较复杂，基本上都是按照DDR控制器的时序要求来做的，其中很多参数要结合DDR芯片本身的参数来定，还有些参数是时序参数，要去详细计算。所以DDR配置非常繁琐、细致、专业。所以我们对DDR初始化的态度就是：学会这种思路和方法，结合文档和代码能看懂，会算一些常见的参数即可。

### 3.7.9重定位代码到SDRAM中

DRAM初始化之后，实际上重定位代码过程和之前重定位到SRAM中完全相同。

实际上就是把链接地址由SRAM中重定位实验时的链接地址0x0xd0024000修改为DRAM中初始化部分的地址处，如0x23e00000。

# 四、时钟系统

## 4.1 SoC时钟系统简介

### 4.1.1什么是时钟，SoC为什么需要时钟

（1）时钟同步工作系统的同步节拍

（2）SoC内部有很多器件，譬如CPU、DRAM控制器、串口、GPIO等内部外设，这些东西要彼此协调工作，需要一个同步的时钟系统来指挥。这个就是SoC的时钟系统。

### 4.1.2时钟一般如何获得？

（1）外部直接输入时钟信号，SoC有个引脚用来输入外部时钟信号：用的很少(有源晶振)

（2）外部晶振+内部时钟发生器产生时钟：大部分单片机（无源晶振）

（3）外部晶振+内部时钟发生器产生时钟+内部PLL倍频+内部分频器分频得到各种频率的时钟：S5PV210

### 4.1.3为什么S5PV210使用第三种时钟？

（1）为什么不直接使用外部高频晶振产生高频信号给CPU？

主要因为芯片外部电路不适宜使用高频率，因为传导辐射比较难控制；高频晶振太贵了

（2）为什么要内部先高频然后再分频？

主要因为SoC内部有很多部件都需要时钟，而且各自需要的时钟频率是不同的，没办法统一供应。因此思路是PLL后先得到一个最高的频率（1GHz、1.2GHz）,然后各外设都由自己的分频器来分频得到自己想要的时钟频率。

### 4.1.4时钟和系统性能的关系、超频、稳定性

（1）一般SoC时钟频率都是人为编程控制的，频率的高低对系统性能的有很大的影响。

（2）S5PV210建议工作频率为800MHz~1.2GHz，一般我们都设置到1GHz主频。如果你设置到1.2GHz就叫超频。超频的时候系统性能会提升，但是发热也会增大，因此会影响系统稳定性。

### 4.1.5时钟和外设编程的关联

每个外设工作都需要一定的时钟频率，这些时钟都是由时钟系统提供的。时钟系统可以通过编程控制工作模式，因此可以人为为每个外设指定时钟来源、时钟分频系统、从而制定这个外设的工作时钟。

### 4.1.6时钟和功耗控制的关系

（1）SoC中各种设备工作时，时钟频率越高功耗越大，发热越大，越容易不稳定，需要外部的散热条件越苛刻。

（2）SoC内部有很多外设，这些外设不用的时候最好关掉（不关掉会一定程度浪费电），开关外设不是通过外设，而是通过时钟。。也就是说我们给某个外设断掉时钟，这个外设就不工作了。

## 4.2 S5PV210的时钟系统简介

### 4.2.1时钟域：MSYS、DSYS、PSYS

因S5PV210的时钟体系比较复杂，内部外设模块太多，因此把整个内部的时钟划分为3大块，叫做3个域。之所以分为3个域，是因为210内部的这些模块彼此工作时钟频率差异太大了，所以又必要把高速的放一起，相对低速的放在一起。

（1）MSYS：CPU（Cortex-A8内核）、DRAM控制器（DMC0和DMC1）、iRAM和iROM等

（2）DSYS：都是和视频显示、编解码有关的模块

（3）PSYS：都是和内部的各种外设有关，譬如串口、SD接口、I2C、AC97、USB等

### 4.2.2时钟来源：外部晶振+内部时钟发生器+PLL+分频电路

S5PV210外部有4个晶振接口，设计板子硬件时可以根据需要来决定在哪里接晶振。接了晶振之后上电，相应的模块就能产生振荡，产生原始时钟。原始时钟再经过一系列筛选开关进入相应的PLL电路生成倍频后的高频时钟。高频时钟再经过分频到达芯片内部各模块上。（有些模块，譬如串口内部还有进一步的分频器进行再次分频使用）

### 4.2.3 PLL：APLL、MPLL、EPLL、VPLL

APLL:Cortex-A8内核 MSYS域

MPLL&EPLL:DSYS PSYS

VPLL:video视频相关模块

## 4.3 S5PV210的时钟域详解

### 4.3.1 MSYS域

ARMCLK：给cpu内核工作的时钟，也就是所谓的主频

HCLK\_MSYS：MSYS域的高频时钟，给DMC0和DMC1使用

PCLK\_MSYS：MSYS域的低频时钟

HCLK\_IMEM：给iROM和iRAM（合称iMEM）使用

### 4.3.2 DSYS域

HCLK\_DSYS：DSYS域的高频时钟

PCLK\_DSYS：DSYS域的低频时钟

### 4.3.3 PSYS域

HCLK\_PSYS：PSYS域的高频时钟

PCLK\_PSYS：PSYS域的低频时钟

SCLK\_ONENAND

总结：210内部各个外设都是接在总线（内部AMBA总线）上面的，AMBA总线有1条高频分支叫AHB，有一条低频分支叫APB。上边各个域都有各自对应的HCLK\_XXXX和PCLK\_XXXX，其中HCLK\_XXXX就是XXXX这个域中AHB总线的工作频率；PCLK\_XXXX就是XXXX这个域中APB总线的工作频率。

SoC内部的各个外设都是挂在总线上工作的，也就是说这个外设的时钟来自于他挂在的总线，譬如串口UART挂在PSYS域下的APB总线上，因此串口的时钟来源是PCLK\_PSYS。我们可以通过记住和分析上边的时钟域和总线值，来确定我们各个外设的具体时钟频率。

### 4.3.4各时钟的典型值（默认值，iROM中设置的值）

（1）当210刚上电时，默认是外部晶振+内部时钟发生器 产生24MHz频率的时钟直接接给ARMCLK的，这时系统的主频就是24MHz，运行非常慢。

（2）iROM代码执行的第6步中初始化了时钟系统，这时给了系统一个默认推荐运行频率。这个时钟频率是三星推荐的210工作性能和稳定性最佳的频率。

（3）各时钟的典型值：

freq(ARMCLK) = 1000 MHz

freq(HCLK\_MSYS) = 200 MHz

freq(HCLK\_IMEM) = 100 MHz

freq(PCLK\_MSYS) = 100 MHz

freq(HCLK\_DSYS) = 166 MHz

freq(PCLK\_DSYS) = 83 MHz

freq(HCLK\_PSYS) = 133 MHz

freq(PCLK\_PSYS) = 66 MHz

freq(SCLK\_ONENAND) = 133 MHz, 166 MHz

## 4.4 S5PV210时钟体系框图详解

* 两张图之间是渐进的关系。第一张图从左到右依次完成了原始时钟生成->PLL倍频得到高频时钟->初次分频得到各总线时钟；第二张图是从各中间时钟（第一张图中某个步骤生成的时钟）到各外设自己使用的时钟（实际就是个别外设自己再额外分频的设置）。可见，第一张图是理解整个时钟体系的关键，第二种图是进一步分析各外设时钟来源的关键。
* 看懂时序框图，2个符号很重要：一个是MUX开关，另一个是DIV分频器。
* MUX开关就是个或门，实际对应某个寄存器的某几个bit位的设置，设置值决定了哪条通道通的，分析这个可以知道右边的时钟是从左边哪条路来的，从而知道右边的时钟是多少。
* DIV分频器，是一个硬件设备，可以对左边的频率进行n分频，分频后的低频时钟输出到右边。分频器在编程时实际对应某个寄存器中的某几个bit位，我们可以通过设置这个寄存器的这些对应bit位来设置分频器的分频系数（譬如左边进来的时钟是80MHz，分频系数设置为8，则分频器右边输出的时钟就是10MHz）。
* 寄存器中的clock source x寄存器就是在设置MUX开关；clock divider control寄存器就是在设置分频器的分频系数。

## 4.5时钟设置的关键寄存器

(1)xPLL\_LOCK

xPLL\_LOCK寄存器主要控制PLL锁存周期的。(从24MHz倍频到1GHz是需要一个过程的)

(2)xPLL\_CON/xPLL\_CON0/xPLL\_CON1

PLL\_CON寄存器主要用来打开/关闭PLL电路，设置PLL的倍频参数，查看PLL锁存状态等。

(3)CLK\_SRCn(n: 0~6)

CLK\_SRC寄存器主要用来设置时钟来源的，对应时钟框图的MUX开关。

(4)CLK\_SRC\_MASKn

CLK\_SRC\_MASK决定MUX开关n选1后是否能继续通过。默认的时钟都是打开的，好处是不会因为某个模块的时钟关闭而导致莫名其妙的问题，坏处是功耗控制不精细，功耗高。

(5)CLK\_DIVn

各模块的分频器参数配置。

(6)CLK\_GATE\_x

类似于CLK\_SRC\_MASK，对时钟进行开关控制。

(7)CLK\_DIV\_STATn 和 CLK\_MUX\_STATn

这两类寄存器，用来查看DIV和MUX的状态是否已经完成还是在进行中

总结：其中最终要的寄存器有3类：CON、SRC、DIV。其中CON决定PLL倍频到多少，SRC决定走那一条路，DIV决定分频多少。

## 4.6汇编实现时钟设置代码详解

### 4.6.1时钟设置的步骤分析

第1步：先选择不使用PLL。让外部24MHz原始时钟直接过去，绕过APLL那条路。

第2步：设置锁定时间（PLL\_LOCK）。默认值是0x0FFF，保险起见我们设置0xFFFF

第3步：设置分频系统，决定由PLL出来的最高时钟如何分频到各个分时钟

第4步：设置PLL，主要是设置PLL倍频系统，决定由输入端24MHz的原始时钟可以得到多大的输出时钟频率。我们按照默认值设置输出为ARMCLK为1GHz。

第5步：打开PLL。前面4步已经设置好 所有开关和分频系数，本步骤打开PLL后PLL开始工作，锁定时钟频率后输出，然后分频得到各个频率。

### 4.6.2 CLK\_SRC0寄存器的配置

CLK\_SRC0寄存器其实是用来设置MUX开关的。在这里先将该寄存器设置为全0，主要是bit0和bit4设置为0，表示APLL和MPLL暂时都不启用。

### 4.6.3CLK\_LOCK寄存器的设置（PLL\_LOCK）

设置PLL锁定延时的。官方推荐值是0xFFF,我们设置为0xFFFF。

### 4.6.4 CLK\_DIV0寄存器的设置

0x14131440这个值的含义分析：

PCLK\_PSYS = HCLK\_PSYS / 2 1

HCLK\_PSYS = MOUT\_PSYS / 5 4

PCLK\_DSYS = HCLK\_DSYS / 2 1

HCLK\_DSYS = MOUT\_DSYS / 4 3

PCLK\_MSYS = HCLK\_MSYS / 2 1

HCLK\_MSYS = ARMCLK / 5 4

SCLKA2M = SCLKAPLL / 5 4

ARMCLK = MOUT\_MSYS / 1 0

### 4.6.5 PLL倍频的相关计算

（1）我们设置了APLL和MPLL，其他两个EPLL和VPLL没管。

（2）APLL和MPLL设置的关键都是M、P、S三个值，这三个值都来自数据手册推荐值。

（3）M、P、S的设置依赖位运算技术

总结：结合寄存器、时钟框图和代码三者综合分析S5PV210的时钟系统，分析时记得在图上做标记（把MUX开关选哪个和DIV分频多少都标出来）就清楚了。

# 五、通信的基础概念

## 5.1通信的基础概念

### 5.1.1通信的发展历史

（1）烽火台、狼烟；信件；电子通信（电报、电话、网络信号）

（2）通信中最重要的两个方面：信息表示、解析方法 + 信息的传输方法

（3）通信双方事先需要约定好信息的表示方法和解析方法，要做到一致才能实现有效传递。

（4）信息的传输方法是指经过编码后的信息如何让在传输介质上传输的过程。

总结：通信的过程分为3步，发送方将可以有效信息按照编码方式进行编码，然后在传输介质上进行传输，接收方接收到被编码的信息后进行解码，解码后得到有效信息。（首先发送方按照信息编码方式对有效信息进行编码（编成可以在通信线路上传输的号形态），然后编码后的信息在传输介质上进行传输，输送给接收方；最后接收方接收到编码信息后进行解码，解码后得到可以理解的有效信息。）

### 5.1.2电子通信的概念1：同步通信和异步通信

（1）同步和异步的区别：发送方和接收方按照同一时钟节拍工作就叫做同步；发送方和接收方没有统一的时钟节拍，而按照自己的时钟节拍工作就叫异步。

（2）同步通信中，通信双发按照统一节拍工作，所以配合很好；一般需要发送方给接收方发送信息时同时发送时钟信号，接收方根据发送方给它的时钟信号来安排自己的节奏。同步通信主要用在通信双方信息交换频率固定或者通信双方经常通信时。

（3）异步通信中，接收方不必一直在意发送方，发送方给接收方发送信息前需要先给接收方发送一个信息开始的起始信号，接收方接收到起始信号后就认为后边紧跟着的就是有效信息，才会开始注意接收信息，知道发送方发来结束标志。

### 5.1.3电子通信概念3：电平信号和差分信号

（1）电平信号和差分信号是用来描述通信线路传输方式的。也就是如何在通信线路上表达1和0。

（2）电平信号的传输线中一般有一个参考电平线（一般是GND），信号值是由信号线上的电平和参考电平线（一般是GND）的电压差决定的。

（3）差分信号的传输线中没有参考电平，所有传输线都是信号线。1和0的表示由信号线之间的电压差决定的。

总结：电平信号的2跟通信线之间的电平容易受到干扰，传输容易失败；差分信号不容易受到干扰，因此传输质量比较稳定，而且差分信号快（1个发送周期更短），现代通信一般都使用差分信号，电平信号几乎没有了。

### 5.1.4电子通信概念3：并行接口和串行接口

（1）串行、并行主要是考虑通信线的根数，就是发送方和接收方同时可以传递的信息量的多少。

（2）譬如在电平信号下的串行，1根参考电平线+1根信号线可以传递1位二进制数；在电平信号下的并行，如果有3根线（1根参考电平线+2根信号线）就可以同时发送2位二进制数；如果想同时发送8位二进制数就需要9根线。

（3）譬如在差分信号下的串行，2根线（彼此差分）可以同时发送1位二进制数；在差分信号下的并行，如果想同时发送8位二进制数就需要16根线。

总结：听起来似乎并行接口比串行接口要快（串行接口一次只能发送1位二进制数，而并行接口一次可以发送多位二进制数），但是实际上还是串行接口用的广。因为更省信号线，而且对传输线的要求更低，成本更低，而且串行时可以通过通信速度的提高来提升总体通信性能，不一定非要并行。

总结：最终胜出的是：异步、差分、串行，譬如USB和网络通信。

## 5.2串口通信的基本概念

* 串口：universal asynchronous reciver and transmitter，通用异步收发器

### 5.2.1串口通信的特点：异步、电平信号、串行

（1）异步：串口通信的发送方和接收方之间没有统一的时钟信号。

（2）电平信号：串口通信出现的时间比较早，速率较低，传输的距离较近，所以干扰还不太明显，因此当时使用了电平信号传输。后期出现的传输协议都改成差分信号传输了。

（3）串行通信：串口通信每次同时只能传输1个二进制位。

### 5.2.2电平信号中的RS232电平和TTL电平

（1）电平信号是用信号线减去参考线电平得到的电压差，这个电压差决定了传输值是1还是0。

（2）在电平信号时多少V代表1，多少V代表0不是固定的，取决于电平标准。譬如RS232电平中-3V~-15V表示1，+3V~+15V表示0；TTL电平则是+5V表示1，0V表示0。

（3）RS232还是TTL都是为了在传输线上表示1和0。区别在于适用的环境和条件不同。RS232电平电压定义范围比较大，适合抗干扰大，距离远的情况；TTL电平电压定义范围比较小，适合距离近且干扰小的情况。

（4）工业上用串口时都用RS232，传输距离小于15米，TTL电平一般用在电路板内部两个芯片之间。

（5）把TTL电平和RS232电平接混是不可以的

### 5.2.3波特率

（1）波特率（bandrate），指的是串口通信的速率，即串口通信时每秒可以传输多少个二进制位。（譬如每秒钟可以传输9600个二进制位（传输一个二进制位需要的时间是1/9600秒，也就是104us），波特率就是9600。）单位是bps。

（2）串口通信的波特率不能随意设定，而应该在一些值中去选择。一般最常见的波特率是9600或者115200（低端单片机如51常用9600，高端单片机和嵌入式SoC一般用115200）.为什么波特率不可以随便指定？主要是因为：第一，通信双方必须事先设定相同的波特率这样才能成功通信，如果发送方和接收方按照不同的波特率通信则根本收不到，因此波特率最好是大家熟知的而不是随意指定的。第二，常用的波特率经过长久发展，就形成了共识，大家常用就是9600或者115200。

### 5.2.4起始位、数据位、奇偶校验位、停止位

（1）串口通信时，收发是一个周期一个周期进行的，每周期传输n个二进制位。这一个周期就叫做一个通信单元，一个通信单元由：起始位+数据位+奇偶校验位+停止位组成的。

（2）起始位是发送方开始发送一个通信单元的标志；数据位是一个通信单元中发送的有效信息；奇偶校验位是用来校验数据位的，以防止数据位出错；停止位是发送方用来表示本通信单元结束的标志。

（3）起始位的定义是串口通信标准事先指定的，是由通信线上的电平变化来反应的。

（4）数据位是本次通信真正要发送的有效数据，串口通信一次发送多少位有效数据是可以设定的（一般可选的有6、7、8、9，99%情况下我们都是选择8位数据位。因为我们一般通过串口发送的文字信息都是ASCII码编码的，而ASCII码中一个字符刚好编码为8位。）

（5）奇偶校验位是用来给数据位进行奇偶校验的（把待校验的有效数据逐个位的加起来，总和为奇数奇偶校验位就为1，总和为偶数奇偶校验位就为0）的，可以在一定程度上防止位反转。

（6）停止位的定义是串口通信标准事先指定的，是由通信线上的电平变化来反映的。常见的有1位停止位，1.5位停止位，2位停止位等。99%情况下都是用1位停止位。

总结：串口通信时因为是异步通信，所以通信双方必须事先约定好通信参数，这些通信参数包括：波特率、数据位、奇偶校验位、停止位（串口通信中起始位定义是唯一的，所以一般不用选择）

### 5.2.5单工通信和双工通信

如果只能A发B收则单工，A发B收或者B发A收（两个方向不能同时）叫半双工，A发B收同时B发A收叫全双工。

## 5.3串口通信的基本原理

### 5.3.1三根通信线：Tx Rx GND

（1）任何通信都要有信息作为传输载体，或者有线的或则无线的。

（2）串口通信时有线通信，是通过串口线来通信的。

（3）串口通信最少需要2根（GND和信号线），可以实现单工通信，也可以使用3根通信线（Tx Rx GND）来实现全双工通信。

（4）一般开发板都会引出SoC上串口引脚直接输出的TTL电平的接口（x210开发板没有哦），插座用插针式插座，每个串口引出的都有3根线（Tx Rx GND），可以用这些插座直接连接外部的TTL电平的串口设备。

### 5.3.2收发双方事先规定好通信参数（波特率、数据位、奇偶校验位、停止位等）

(1)串口通信属于基层基本性的通信规约，它自己本身不会去协商通信参数，需要通信前通信双方事先约定好通信参数（一般4个最重要的）

(2)串口通信的任何一个关键参数设置错误，都会导致通信失败。譬如波特率调错了，发送方发送没问题，接收方也能接收，但是接收到全是乱码。

### 5.3.3信息以二进制流的方式在信道上传输

（1）串口通信的发送方每隔一定时间（时间固定为1/波特率，单位是秒）将有效信息（1或者0）放到通信线上去，逐个二进制位的进行发送。

（2）接收方通过定时（起始时间由读到起始位标志开始，间隔时间由波特率决定）读取通信线上的电平高低来区分发送给我的是1还是0。依次读取数据位、奇偶校验位、停止位，停止位就表示这一个通信单元（帧）结束，然后中间是不定长短的非通信时间（发送方有可能紧接着就发送第二帧，也可能半天都不发第二帧，这就叫异步通信），下来就是第二帧·····

总结：第一，波特率非常重要，波特率错了整个通信就乱套了；数据位、奇偶校验位、停止位也很重要，否则可能认不清数据。第三，通过串口不管发数字、还是文本还是命令还是什么，都要先对发送内容进行编码，编码成二进制再进行逐个位的发送。

（3）串口发送的一般都是字符，一般都是ASCII码编码后的字符，所以一般设置数据位都是8，方便刚好一帧发送1个字符。

### 5.3.4 DB9接口介绍

(1)DB9接口是串口通信早期比较常用的一种规范化接口。

(2)串行通信在早期是计算机与外界通信的主要手段，那时候的计算机都有标准配置的串口以实现和外部通信。那时候就定义了一套标准的串口规约，DB9接口就是标准接口。

(3)DB9接口中有9根通信线，其中3根很重要，为GND、Tx、Rx，必不可少；剩余6根都是和流控有关的，现代我们使用串口都是用来做调试一般都禁用流控，所以这6根没用。

(4)现在一般使用串口时要记得把流控禁止掉，不然可能发生意想不到的问题。

## 5.4 S5PV210串行通信接口详解

### 5.4.1 S5PV210的串口控制器工作原理框图

（1）整个串口控制器包含transmitter和receiver两部分，两部分功能彼此独立，transmitter负责210向外部发送信息，receiver负责从外部接收信息到210内部。

（2）总线角度讲，串口控制器是接在APB总线上的。对编程的影响是：将来计算串口控制器的源时钟时是以APB总线来计算的。

（3）transmitter由发送缓冲区和发送移位器构成。我们要发送信息时，首先将信息进行编码（一般用ASCII码）成二进制流，然后将一帧数据（一般是8位）写入发送缓冲区（从这里以后程序就不用管了，剩下的发送部分就是硬件自动的），发送移位器会自动从发送缓冲区中读取一帧数据，然后自动移位（移位的目的是将一帧数据的各个位分别拿出来）将其发送到Tx通信线上。

（4）receiver由接收缓冲区和接收移位器构成。当有人通过串口线向210发送信息时，信息通过Rx通信线进入210接收移位器，然后接收移位器自动移位将该二进制位保存到接收缓冲区，接受完一帧数据后receiver会产生一个中断给CPU，CPU接收到中断后即可知道receiver接收满了一帧数据，就会来读取这一帧数据。

总结：发送缓冲区和接收缓冲区是关键。发送移位器和接收移位器的工作都是硬件自动的，不用软件控制。所以代码就是：首先初始化（初始化的实质是读写寄存器）好串口控制器（包括发送控制器和接收控制器），然后要发送信息时直接写入发送缓冲区，要接收信息时直接去接受缓冲区去读即可。可见，串口底层的工作（譬如怎么移位的，譬如起始位怎么定义的，譬如TTL电平还是RS232电平等）都是软件无需关注的。软件对串口操作的接口就是发送/接收缓冲区（实质就是寄存器，操作方式就是读写内存）

（5）串口控制器有一个波特率发生器，作用是产生串口发送/接收的节拍时钟。波特率发生器其实就是个时钟分频器，它的工作需要源时钟（APB总线来），然后内部将源时钟进行分频（软件设置寄存器来配置）得到目标时钟，然后再用这个目标时钟产生波特率（硬件自动的）。

### 5.4.2自动流控（AFC:Auto flow control）

（1）为什么需要流控？流控的目的是让串口通信非常可靠，在发送发速率比接收方快的时候可以保证发送给接收方的信息不回漏掉。

（2）现在为什么不用流控？现在计算机之间有更好更高级（usb、internet）的通讯方式，串口已经基本被废弃了。现在串口的用途更多是SoC用来输出调试信息的。由于调试信息不是关键性信息、而且由于硬件发展串口本身速度已经相对慢的要死了，所以硬件都能协调发送和接收速率，因此流控已经失去意义了，所以现在基本都废弃了。

### 5.4.3串口的高级功能

* FIFO模式及其作用

(1)典型的串口设计，发送/接收缓冲区只有1字节，每次发送/接收只能处理1帧数据。这样在单片机中没什么问题，但是到复杂SoC中（一般有操作系统的）就会有问题，会导致效率低下，因为CPU需要不断切换上下文。

(2)解决方案就是想办法扩展串口控制器的发送/接收缓冲区，譬如将发送/接收缓冲器设置为64字节，CPU一次过来直接给发送缓冲区64字节的待发送数据，然后transmitter慢慢发，发完再找CPU再要64字节。但是串口控制器本来的发送/接收缓冲区是固定的1字节长度的，所以做了个变相的扩展，就是FIFO。

(3)FIFO就是first in first out，先进先出。fifo其实是一种数据结构，这里这个大的缓冲区叫FIFO是因为这个缓冲区的工作方式类似于FIFO这种数据结构。

* DMA模式及其作用

(1)DMA direct memory access，直接内存访问。DMA本来是DSP中的一种技术，DMA技术的核心就是在交换数据时不需要CPU参与，模块可以自己完成。

(2)DMA模式要解决的问题和上面FIFO模式是同一个问题，就是串口发送/接收要频繁的折腾CPU造成CPU反复切换上下文导致系统效率低下。

(3)传统的串口工作方式（无FIFO无DMA）效率是最低的，适合低端单片机；高端单片机上CPU事物繁忙所以都需要串口能够自己完成大量数据发送/接收。这时候就需要FIFO或者DMA模式。FIFO模式是一种轻量级的解决方案，DMA模式适合大量数据迸发式的发送/接收时。

* IrDA模式及其用法

(1)IrDA其实就是红外，红外就是红外线通信（电视机、空调遥控器就是红外通信的）。

(2)红外通信的原理是发送方固定间隔时间向接收方发送红外信号（表示1或0）或者不发送红外信号（表示0或者1），接收方每隔固定时间去判断有无红外线信号来接收1和0.

(3)分析可知，红外通信和串口通信非常像，都是每隔固定时间发送1或者0（判断1或0的物理方式不同）给接收方来通信。因此210就利用串口通信来实现了红外发送和接收。

(4)210的某个串口支持IrDA模式，开启红外模式后，我们只需要向串口写数据，这些数据就会以红外光的方式向外发射出去（当然是需要一些外部硬件支持的），然后接收方接收这些红外数据即可解码得到我们的发送信息。

### 5.4.4串行通信与中断的关系

（1）串口通信分为发送/接收两个部分。发送方一般不需要（也可以使用）中断即可完成发送，接收方必须（一般来说必须，也可以轮询方式接收）使用中断来接收。

（2）发送方可以选择使用中断，也可以不使用中断。使用中断的工作情景是：发送方事先设置好中断并绑定一个中断处理程序，然后发送发丢一帧数据给transmitter，transmitter发送耗费一段时间来发送这一帧数据，这段时间内发送方CPU可以去做别的事情，等transmitter发送发送完成后会产生一个TXD中断，该中断会导致事先绑定的中断处理程序的执行，在中断处理程序中CPU会切换回来继续给transmitter放一帧数据，然后CPU切换离开；不使用中断的工作情景是：发送方事先禁止TxD中断（当然也不需要给相应的中断处理程序了），发送方CPU给一帧数据给transmitter，然后transmitter耗费一段时间来发送这帧数据，这段时间CPU在这等着（CPU没有切换去做别的事情），待发送方发送完成后CPU再给它一帧数据继续发送直到所有数据发完。CPU是怎么直到transmitter已经发送完了？原来是有个状态寄存器，状态寄存器中有一个位叫做发送缓冲区空标志，transmitter发送完成了（发送缓冲区空了）就会给这个标志位，CPU就是通过不断查询这个标志位为1还是0来知道发送是否已经完成的。

（3）因为串口通信是异步的，异步的意思就是说发送方占主导权。也就是发送方随时想发就发，而接收方只有时刻等待才不会丢失数据。所以这个差异就导致发送方可以不用中断，而接收方不得不使用中断模式。

### 5.4.5 210串行通信接口的时钟设计

（1）串口通信为什么需要时钟？因为串口通信需要一个固定的波特率，所以transmitter和receiver都需要一个时钟信号。

（2）时钟信号从哪来？源时钟信号是外部APB总线（PCLK\_PSYS,66MHz）所提供给串口模块的（这就是为什么我们说串口是挂在APB总线上的），然后进到串口控制器内部后给波特率发生器（实质上是一个分频器），在波特率发生器中进行分频，分频后得到一个低频时钟，这个时钟就是给transmitter和receiver使用的。

（3）串口通信中时钟的设置主要看寄存器的设置。重点有：寄存器源设置（为串口控制器选择源时钟，一般选择PCLK\_PSYS，也可以是SCLK\_UART），还有波特率发生器的2个寄存器。

（4）波特率发生器中有2个重要寄存器：UBRDIVn和UDIVSLOTn，其中UBRDIVn是主要的设置波特率的寄存器，UDIVSLOTn是用来辅助设置的，目的是为了校准波特率的。

## 5.5S5PV210串行通信编程实战

### 5.5.1整个流程分析

整个串口通信相关程序包含2部分：uart\_init负责初始化串口，uart\_putc负责发送一个字节

### 5.5.2串口初始化关键步骤

（1）初始化串口的Tx和Rx引脚所对应的GPIO(查原理图可知Tx和Rx分别对应GPA0\_1和GPA0\_0)

（2）GPA0CON（0xE0200000），bit[3:0] = 0b0010 bit[7:4] = 0b0010

（3）初始化这几个关键寄存器UCON0 ULCONO UMCON0 UFCON0 UBRDIV0 UDIVSLOT0

### 5.5.3主要的几个寄存器

（1）ULCON0 = 0x3 //0校验位、8数据位、1停止位

（2）UCON = 0x5 //发送和接收都是polling mode

（3）UMCON0 = 0x0 //禁止modem、afc

（4）UFCON0 = 0x0 //禁止FIFO模式

（5）UBRDIV0和UDIVSLOT0和波特率有关，要根据公式去算

### 5.5.4在C源文件中定义访问寄存器的宏

定义好了访问寄存器的宏之后，将来写代码时直接使用即可。

### 5.5.5串口Tx、Rx对应的GPIO的初始化

给GPA0CON的相应bit位赋值为相应值，用C语言位操作来完成。

### 5.5.6 UCON、ULCON、UMCON、UFCON等主要控制寄存器赋值

### 5.5.7波特率的计算和设置

（1）第一步，用PCLK\_PSYS和目标波特率去计算DIV\_VAL

DIV\_VAL = (PCLK / (bps x 16)) - 1

（2）第二步，UBRDIV0寄存器中写入DIV\_VAL的整数部分

（3）第三步，用小数部分\*16得到1的个数，查表得UBDIVSLOT0寄存器的设置值

### 5.5.8串口发送和接收函数的编写

写发送函数，主要发送前要用while循环等待发送缓冲区为空才能发送。

### 5.5.9综合调试：注意Makefile的修改

# 5.6 uart stdio的移植

### 5.6.1什么是stdio？

（1）#include <stdio.h>

（2）stdio:standard input output

（3）stdio是os定义的默认的输入和输出通道。一般在PC机的情况下，标准输入指的是键盘，标准输出指的是屏幕。

（4）printf函数和scanf函数可以和底层输入、输出函数绑定，然后这两个函数就可以和stdio绑定起来。也就是说我们直接调用printf函数输出，内容就会被从标准输出输出出去。

（5）在本节中，标准输出不是屏幕，而是串口；标准输入也不是键盘，而是串口。

### 5.6.1printf函数的工作原理

printf函数工作时内部实际调用了2个关键函数：一个是vsprintf函数（主要功能是格式化打印信息，最终得到纯字符串格式的打印信息等待输出），另一个就是真正的输出函数putc（操控标准输出的硬件，将信息发送出去）

### 5.6.1移植printf函数的三种思路

（1）我们希望在我们开发板上使用printf函数进行（串口）输出，使用scanf函数进行（串口）输入，就像在PC机上用键盘和屏幕进行输入和输出一样。因此需要移植printf函数/scanf函数。

（2）我们说的移植而不是编写，我们不希望自己完全重新编写，而是尽量借用已有的代码。（移植）

（3）一般printf函数可以有3个途径获取printf的实现源码：最原始的原本的来源就是linux内核中的printk。难度较大且麻烦；稍微简单些的方法是从uboot中移植printf；更简单的方法是直接使用别人移植好的。

### 5.6.1printf移植

（1）修改Makefile

（2）Makefile及gcc的库文件介绍

（3）多文件夹裸机工程的结构解析

（4）编译运行及测试

（5）在移植后的uart stdio项目中添加link.lds链接脚本，指定链接地址为0xd0020010

### 5.6.1gcc可变参数及va\_arg介绍

（1）printf函数中首先使用了C语言的可变参数va\_start/va\_arg/va\_end;

（2）建议先去百度“C语言可变参数”，然后按照别人的教程、博客实际写几个简单的变参的使用示例，先明白可变参数怎么工作，然后再来分析这里。

### 5.6.1vsprintf函数详解：

printf

vsprintf

number

vsprintf函数的作用是按照我们的printf传进去的格式化标本，对变参进行处理，然后将之格式化后缓存在一个事先分配好的缓冲区中。

printf后半段调用putc函数将缓冲区中格式化好的字符串直接输出到标准输出。

### 5.6.1USB下载bin最多96KB，SD卡下载bin文件最多16KB怎么办？

通过USB下载最多也只能下载96KB的bin，如果bin大于96KB肯定SRAM放不下会出错。如果SD卡启动，那么mkv210\_image.c决定了bin文件最大不能超过16KB。

解决方法：

第一：在USB下载时，可以先下载一个x210\_usb.bin，然后再将裸机程序链接到0x23e00000，然后修改dnw下载地址，将裸机代码下载到0x23e00000运行。（不需要重定位）

第二：在SD卡启动时，将整个裸机工程分为2部分；第一部分大小16KB以内，第二部分放剩下的（放在SD卡后面的某个扇区开始的位置，譬如放在第50个扇区开始的位置），然后在裸机代码中进行重定位（SD卡中重定位）。

# 六、按键和CPU的中断系统

## 6.1什么是按键

### 6.1.1按键的物理特性

(1)、平时没人按的时候，弹簧把按键按钮弹开。此时内部断开的。

(2)、有人按下的时候，手的力量克服弹簧的弹力，将按钮按下，此时内部保持接通（闭合）状态；如果手拿开，则弹簧作用下按钮又弹开，同时内部又断开。

(3)、一般的按键都有4个引脚，这4个引脚成2对：其中一对是常开触点（像上面描述的不按则断开，按下则闭合）；一对是常闭触点（平时不按时是闭合的，按下后是断开的）

### 6.1.2按键的电学原理（结合原理图分析）

(1)硬件接法： SW5:GPH0\_2 SW6:GPH0\_3 SW78910:GPH2\_0123

(2)按键的电路连接分析：平时按钮没有按下时，按钮内部断开，GPIO引脚处电压为高电平；当有人按下按钮时，按钮内部导通，外部VDD经过电阻和按钮连接到地，形成回路，此时GPIO引脚处电压就变成了低电平。此时VDD电压全部分压在了电阻上（这个电阻就叫分压电阻，这个电阻不能太小，因为电阻的功率是U\*U/R）

(3)总结：按键的工作方法：其实就是按键的按下与弹开，分别对应GPIO的两种电平状态（按下则GPIO为低电平，弹开则GPIO为高电平）。此时SoC内部可以通过检测这个GPIO的电平高低来判断按键有没有被按下，这个判断结果即可作为SoC的输入信号。

### 6.1.3按键属于输入类设备

(1)按键一般用来做输入设备（由人向SoC发送信息的设备，叫输入设备），由人向SoC发送按键信号（按键信号有2种：按下信号和弹开信号）。

(2)有些设备就是单纯的输入设备，譬如按键、触摸屏等；有些设备就是单纯的输出设备，譬如LCD；还有一些设备是既能输入又能输出的，叫输入输出设备（IO），譬如串口。

### 6.1.4按键的2种响应方法

（1）SoC处理按键有两种思路：轮询方式和中断方式

（2）轮询方式，就是SoC主动的每隔一段时间去读取（按键所对应的）GPIO的电平高低，以此获得按键信息；缺点在于CPU要一直注意按键事件，会影响CPU做其他事情。

（3）中断方式，就是SoC事先设定好GPIO触发的中断所对应的中断处理程序ISR，当外部按键按下或弹开时会自动触发GPIO对应的外部中断，导致ISR执行，从而自动处理按键信息。

## 6.2轮询方式处理按键

### 6.2.1 X210开发板的按键接法

（1）查原理图，找到按键对应的GPIO：SW5:GPH0\_2 SW6:GPH0\_3 SW78910:GPH2\_0123

（2）原理图上可看出：按下时是低电平，弹起时是高电平。

### 6.2.2按键对应的GPIO模式设置

（1）按键接到GPIO上，按键按下还是弹起，决定外部电路的接通与否，从而决定这个GPIO引脚的电压是高还是低；这个电压可以作为这个GPIO引脚的输入信号，此时GPIO配置为输入模式，即可从SoC内部读取该引脚的电平为1还是0（1对应高电平，0对应低电平）。

（2）GPH0CON(0xE0200C00) GPH2DAT(0xE0200C04) GPH2CON(0xE0200C40) GPH2DAT(0xE0200C44)

（3）设置CON寄存器为输入模式，然后读取DAT寄存器。（读取到相应位的值为1表示外部是高电平（对应按键弹起），读取到的位值为0表示外部是低电平（按键按下））

### 6.2.3轮询方式处理按键的程序流程

（1）第一步，先初始化GPIO为input模式

（2）第二部，循环读取GPIO的电平值，然后判断有无按键按下

## 6.3串口输出与按键消抖

### 6.3.1基于串口标准输出的按键调试

（1）以之前的串口stdio的工程为基础来移植添加轮询方式的按键处理。

（2）注意USB下载方式可能有错误（有可能不下载，也有可能下载了执行不对），解决方案是用SD卡启动来替代。

### 6.3.2什么是消抖

（1）按键这种物理器件本身会由抖动信号，抖动信号指的是在电平由高到低（也就是按键按下时）或者电平由低到高（也就是按键弹起时）过程中，电平的边画画不是立刻变化，而是经过了一段时间的不稳定期才完成变化，这个不稳定期间电平可能会时高时低反复变化，这个不稳定期就叫抖动（抖动期间获取按键信息是不可靠的，要想办法消抖）。

（2）什么叫消抖？消抖就是用硬件或软件的方法尽量减小抖动期对按键获取的影响。硬件：消抖思路就是尽量减小抖动时间，方法是通过硬件添加电容等元件来减小抖动；软件：消抖思路是发现一次按键按下/弹起事件后，不立即处理按键，而是延时一段时间（一般是10ms~20ms）后再次获取按键值，如果此次获取和上次一样按下/弹起，那就认为真的按下/弹起了。

（3）一般比较精密需要的时候，需要硬件消抖和软件消抖一起配合。

## 6.4 S5PV210的中断体系介绍

### 6.4.1什么是中断？

（1）中断的发明是用来解决宏观上的并行需要的。宏观就是从整体来看，并行就是多件事情都完成了。

（2）微观上的并行，就是指的真正的并行，就是精确到每一秒甚至每一刻，多个事情都在同时进行的。宏观上面的并行不等于微观的并行，有时候宏观上是并行的，微观上是串行的。

（3）例子中一个人在看电影，快递来了暂停电影跑去收快递，收完快递继续回来看电影，这个例子就是宏观上的并行和微观上的串行。例子中一个人等同于SoC中1个CPU（也就是单核CPU），这个CPU看电影就不能收快递，收快递就不能看电影（也就是说不能真正的并行）。单核心CPU在微观角度是串行的，但是因为CPU很快，所以在宏观看来可以并行。

（4）上例中大部分时间在看电影，中间少量时间去收快递，那么类比于CPU来说，看电影就应该是CPU的常规任务，而收快递则应该是中断例程。也就是说CPU平时一直在进行看电影任务，等快递来了（中断发生了）快递员（类似于中断源）会打电话叫人去收快递（中断源会触发中断通知CPU去处理中断），人收到电话（CPU收到中断信号）后会暂定电影（CPU保存常规任务的现场）跑去收快递（CPU去执行中断处理程序ISR处理中断），收完快递（执行完ISR）回来继续看电影（CPU恢复常规任务的现场，继续执行常规任务）

（5）为什么需要中断？因为单核CPU实际无法并行的，但是通过中断机制，可以实现假并行（宏观上的并行，微观上实际还是串行的）。

### 6.4.2 SoC对中断的实现机制：异常向量表

（1）异常向量表是CPU中某些特定地址的特定定义。当中断发生时，中断要想通知CPU去处理中断，怎么做到？这就要靠异常向量表。

（2）在CPU设计时，就事先定义了CPU中一些特定地址作为特定异常的入口地址（譬如定义0x00000000地址为复位异常地址，则发生复位异常时CPU会自动跳转到0x00000000地址去执行指令。又譬如外部中断对应的异常向量地址是0x30000008，则发生外部中断后，CPU会硬件自动跳转到0x30000008地址去执行指令。）

（3）以上讲的是CPU硬件设计时对异常向量表的支持，下来就需要软件支持了。硬件已经决定了发生什么异常CPU自动跳转PC到哪个地址去执行，软件需要做的就是把处理这个异常的代码的首地址填入这个异常向量地址。

### 6.4.3 S5PV210的异常向量表

（1）异常向量表中各个向量的相对位置是固定的，但是他们的起始地址是不固定的，各种SoC可以不一样，而且复杂ARM中还可以让用户来软件设置这个异常向量表的基地址。

（2）扩展到所有架构的CPU中：所有架构（譬如51单片机、PIC单片机）的CPU实现中断都是通过异常向量表来实现的，这个机制是不变的；但是不同CPU的异常向量表的构造和位置是不同的。

### 6.4.4异常和中断的区别和联系

（1）针对SoC来说，发生复位、软中断、中断、快速中断、取指令异常、数据异常等，我们都统一叫异常。所以说：中断其实是异常的一种。

（2）异常的定义就是突发事件，打断了CPU的正常常规业务，CPU不得不跳转到异常向量表中去执行异常处理程序；中断是异常的一种，一般特指SoC内的内部外设产生的打断SoC常规业务，或者外部中断（SoC的GPIO引脚传回来的中断）。

## 6.5异常向量表的编程处理

### 6.5.1像内存一样去访问异常向量表

（1）S5PV210的异常向量表可以改变（在CP15协处理器中），以适应操作系统的需求。但是目前系统刚启动，此时DRAM尚未初始化，程序哦都市在iSRAM中运行。210在iSRAM中设置了异常向量表，供暂时性使用。

（2）查210的iROM application note文档中iRAM的地址分配，可知，iRAM中的异常向量表起始地址为0xd0037400。知道了异常向量表的起始地址后，各个异常的对应地址的入口就知道了。

### 6.5.2函数名的实质就是函数的首地址

（1）函数名在C语言中的理解方法和变量名其实没有区别。编译器会把这个函数的函数名对应的代码段和这个函数的函数名（实质是符号）对应起来，等我们在使用这个函数名符号时，编译器会将函数的函数体实际上做替换。因为函数体都不只4个字节，而函数名这个符号只能对应1个地址，所以实际对应的是函数体那一个代码段的首地址。

（2）拿C语言中的语法来讲，函数名就是这个函数的函数指针。

总结：当我们将异常处理程序的首地址和异常向量表绑定后，异常处理初步阶段就完成了。到目前可以保证相应异常发生后，硬件自动跳转到对应异常向量表入口去执行时，可以执行我们事先绑定的函数。

### 6.5.3为什么中断处理要先在汇编中进行

（1）中断处理要注意保护现场（中断从SVC模式来，则保存SVC模式下的必要寄存器的值）和恢复现场（中断处理完成后，准备返回SVC模式前，要将保存的SVC模式下的必要寄存器的值恢复回去，不然到了SVC模式后寄存器的值乱了，SVC模式下原来正在进行的常规任务就被你搞坏了）

（2）保存现场包括：第一：设置IRQ栈；第二，保存LR；第三，保存R0~R12

（3）为什么要保存LR寄存器？要考虑中断返回的问题。中断ISR执行完成后如何返回SVC模式下去接着执行原来的代码。中断返回其实取决于我们进入中断时如何保存现场。中断返回时关键的2个寄存器就是PC和CPSR。所以我们在进入IRQ模式时，应该将SVC模式下的下一句指令的地址（中断返回地址）和CPSR保存起来，将来恢复时才可以将中断返回地址给PC，将保存的CPSR给CPSR。

（4）中断返回地址就保存在LR中，而CPSR（自动）保存在（IRQ模式下的）SPSR中。

### 6.5.4汇编保存现场和恢复现场

（1）保护现场关键是保存：中断处理程序的返回地址，r0-r12（cpsr是自动保存的）

（2）恢复现场主要是恢复：r0-r12,pc,cpsr

## 6.6 S5PV210的向量中断控制器

### 6.6.1异常处理的2个阶段

可以将异常处理分为2个阶段来理解。第一个阶段是异常向量表跳转；第二个阶段是进入了真正的异常处理程序irq\_handler之后的部分。

### 6.6.2回顾：中断处理的第一个阶段（异常向量表跳转阶段）处理

（1）第一个阶段之所以能够进行，主要依赖于CPU设计时提供的异常向量表机制。第一个阶段的主要任务是从异常发生到响应异常并且保存/恢复现场、跳转到真正的异常处理程序处。

（2）第二个阶段的目的是识别多个中断源中究竟哪一个发生了中断，然后调用相应的中断处理程序来处理这个中断。

### 6.6.3 S3C2440的第二阶段处理过程

（1）第一个问题，怎么找到具体是哪个中断:S3C2440的中断控制器中有一个寄存器（32位的），寄存器的每一个位对应一个中断源（为了解决支持更多中断源，2440又设计了一个子中断机制。在一级中断寄存器中有一些中断是共用的一个bit位，譬如AC97和WDT。对于共用中断，用子中断来区分究竟是哪一个发生了中断）

（2）第二个问题，怎么找到对应isr的问题：首先给每个中断做了个编号，进入irq\_handler之后先通过查阅中断源寄存器和子中断寄存器（中哪一位为1）确定中断的编号，然后用这个编号去isr数组（isr数组是中断初始化时事先设定好的，就是把各个中断的isr的函数名组成的一个数组，用中断对应的编号作为索引老查询这个数组）中查阅得到isr地址。

评价：2440的中断处理设计不是很优秀。第一个过程中使用子中断搞成2级的很麻烦；第二个过程中计算中断编号是个麻烦事，很耗费时间。而中断处理的时间是很宝贵的（系统有一个性能指标，叫实时性。实时性就是中断发生到响应的时间，这个时间越短越好。）

### 6.6.4 S5PV210的第二阶段处理过程

（1）第一个过程，怎么找到具体是哪个中断：S5PV210中因为支持的中断源很多，所以直接设计了4个中断寄存器，每个32位，每位对应一个中断源。（理论上210最多可以支持128个中断源，实际支持不足128个，有些位是空的）；210没有子中断寄存器，每个中断源都是并列的。当中断发生时，在irq\_handler中依次去查询4个中断源寄存器，看哪一个的哪一位被置1，则这个位对应的寄存器就发生中断，即找到了中断编号。

（2）第二个问题，怎么找到对应的isr的问题：210中支持的中断源多了很多，如果还是用2440那一套来寻找isr地址就太慢了，太影响实时性了。于是210开拓了一种全新的寻找isr的机制。210提供了很多寄存器来解决每个中断源对应isr的寻找问题，具体寻找过程和建立过程见下节，实现的效果是当发生相应中断时，硬件会自动将相应isr推入一定的寄存器中，我们软件只要去这个寄存器中执行函数就行了。

### 6.6.5总结：第一个阶段都相同，第二个阶段各不同

（1）第一极端（异常向量表阶段）2440和210几乎是完全相同的。实际上几乎所有的CPU在第一阶段都是相同的。

（2）第二阶段就彼此不同了。各个SoC更具自己对实时性的要求，和支持中断源的多少，各自发明了各自处理中断，找到中断编号，进一步找到对应isr地址的方式。

* 自我总结：

210的异常向量表中有8种异常模式（8层楼），每种异常模式下又会有多种中断源。异常处理分为2个阶段来理解，第一个阶段是异常向量表跳转，也就是前往哪个楼层的问题。（相当于去办事大厅办某件事，前台告知在7楼）这一阶段要考虑保护和恢复现场。第二阶段是进入到了真正的异常处理程序irq\_handler。（进入了7楼，但是7楼有4\*32个房间，需要找到是哪个房间，再进房间办事）包括找到是哪个中断源和进入对应的isr。找到是哪个中断源：4个32位中断寄存器，每一位代表一种中断源，依次读取4个中断寄存器，哪个位被置一，则这一位对应的中断源发生中断，即找到了中断编号。进入对应的isr：硬件会自动的将相应isr推入一定的寄存器中（210通过一堆寄存器来实现），我们软件只要去这个寄存器中执行函数即可。

## 6.7 S5PV210中断处理的主要寄存器

1.VICnINTENABLE和VICnINTENCLEAR

(1)VICnINTENABLE是指interrupt enable，VICnINTENCLEAR是指interrupt enable clear

(2)INTENABLE是使能中断，INTENCLEAR是禁止中断。

(3)当我们想使能（意思是启用这个中断，意思就是当硬件产生中断时CPU能接受的到）某个中断时，只要在这个中断编号对应的VICnINTENABLE的相应bit位写1即可（注意这个位写1其他位写0对其他位没有影响）；如果我们想禁止某个中断源时，只要向VICnINTENCLEAR中相应的位写1即可。

注：这里的设计可以有2种方式：有些CPU是中断使能和禁止共用寄存器位，写1使能写0禁止（或者写0使能写1禁止），这样的中断使能/禁止设计就要非常小心，要使用读改写来操作；210是中断使能和中断禁止分为2个寄存器，要使能就写使能寄存器，要禁止就写禁止寄存器。这样的好处是我们使能/禁止操作时不需要读改写，直接写即可。互不影响。

2.VICnINTSELECT

(1)设置各个中断的模式时irq还是fiq。一般都是irq。

(2)IRQ和FIQ的区别：irq是普通中断，fiq是快速中断。fiq提供一种更快相应处理的中断通道，用于对实时性要求很高的中断源。fiq在CPU设计时预先提供了一些机制保证fiq可以被快速处理，从而保证实时性。fiq的限制就是只能有一个中断源被设置为fiq，其他都是irq。

(3)CPU如何保证fiq比irq快？有2个原因：其一，fiq模式有专用的r8~r12，因此在fiq的isr中可以直接使用r8~r12而不用保存，这就能节省时间；其二，异常向量表中fiq是最后一个异常向量入口，因此fiq的isr不需要跳转，可以直接写在原地，这样就比其他异常少跳转一次，节省了时间。

3.VICnIRQSTATUS和VICnFIQSTATUS

中断状态寄存器，是只读的。当发生了中断时，硬件会自动将该寄存器的对应位置1，表示中断发生了。软件在处理中断第二阶段的第一部分找到中断源，就是靠查询这个寄存器来得到中断编号的。

4.VICnVECTPRIORITY0~VICnVECTPRIORITY31

中断优先级设置寄存器，设置多个中断同时发生时优先处理谁后处理谁的问题。一般来说高优先级的中断可以打断低优先级的中断，从而嵌套处理中断。当然，优先硬件/软件可以设置不支持中断嵌套。

5.VICnVECTADDR0~VICnVECTADDR31、VICnADDR

(1)这些个寄存器和210中断处理的第二阶段的第二部分进入对应isr有关。

(2)VICnVECTADDR 0到31这32个寄存器分别用来存放真正的各个中断对应的isr的函数地址。相当于每个中断源都有一个VECTADDR寄存器，程序员在设置中断的时候，把这个中断的isr地址直接放入这个中断对应的VECTADDR寄存器即可。

(3)VICnADDR这个寄存器是只需要去读的，它里边的内容是硬件自动设置的。当发生了相应的中断时，硬件会自动识别中断编号，并且会自动找到这个中断的VECTADDR寄存器，然后将其读出复制到VICnADDR中，供我们使用。这样的设计避免了软件查找中断源和isr，节省了时间，提高了210的中断响应速度。

## 6.8 S5PV210中断处理的编程实战

### 6.8.1中断控制器初始化

主要工作有：第一阶段绑定异常处理程序到异常向量表；禁止所有中断源；选择所有中断类型为IRQ；清理VICnADDR寄存器为0。

为什么要禁止所有中断源：

因为中断一旦打开，因为外部或硬件自己的原因产生中断后一定就会寻找isr，而我们可能认为自己用不到这个中断就没有提供isr，这时它自动拿到的就是乱码，程序可能跑飞，所以不用的中断一定要禁掉。一般的做法是先全部禁止，然后逐一打开自己感兴趣的中断，一旦打开就必须给这个中断提供相应的isr并绑定好。

### 6.8.2中断的使能与禁止

思路是先根据中断号intnum判断这个中断属于VIC几，然后在用中断号减去这个VIC的偏移量，得到这个中断号在本VIC的偏移量，然后1<<x位，写入相应的VIC的INTENABLE/INTENCLEAR寄存器即可。

### 6.8.3绑定自己实现的isr到VICnVECTADDRx

(1)先要搞清楚2个寄存器的区别：VICnVECTADDRx和VICnADDR。

VICnADDR是将真正产生中断的VICnVECTADDRx中isr的地址复制到VICnADDR中。

(2)VICVECTADDR寄存器一共有4\*32个，每个中断源都有一个VECTADDR寄存器，我们应该将自己为这个中断源写的isr地址丢到这个中断源对应的VECTADDR寄存器中即可。

### 6.8.4真正的中断处理程序如何获取isr

当发生中断时，硬件会自动把相应中断源的isr的地址从VICnVECTADDR寄存器中推入到VICnADDR寄存器中，所以我们第二阶段的第二步irq\_handler中，只需要到对应的VICnADDR寄存器中去拿出isr地址，调用执行即可。

总结：第3步绑定isr到VICnVECTADDR和第4步中断发生时第二阶段的第二部分如何获取isr地址，这两步是相关的。这两个的结合技术，就是我们一直在说的210的硬件自动寻找isr的机制。

整个中断的流程梳理：

整个中断的工作分为2部分：

第一部分是我们为中断响应而做的预备工作：

1.初始化中断控制器

2.绑定写好的isr到中断控制器

3.相应中断的所有条件使能

第二部分是当硬件产生中断后如何自动执行isr：

1.第一步，经过异常向量表跳转入IRQ/FIQ的入口

2.第二步，做中断现场保护（在start.S中），然后跳入irq\_handler

3.第三步，在irq\_handler中先搞清楚是哪个VIC中断了，然后直接去这个VIC的ADDR寄存器中取isr来执行即可。

4.第四步，isr执行完，中断现场恢复。直接返回继续做常规任务。

## 6.9外部中断

### 6.9.1什么是外部中断

(1)内部中断就是指中断源来自于SoC内部（一般是内部外设），譬如串口、定时器等部件产生的中断；外部中断是SoC外部的设备，通过外部中断对应的GPIO引脚产生的中断。

(2)按键在SoC中就使用了外部中断来实现。具体实现方法是：将按键电路接在外部中断的GPIO上，然后将GPIO配置为外部中断模式。此时人通过按按键改变按键电路的高低，这个电压高低会出发GPIO对应的外部中断，通过引脚传进去给CPU处理。

(3)外部中断相关的介绍和寄存器都在2.2.6章节（属于GPIO部分）

### 6.9.2电平触发和边沿触发

(1)外部中断的触发方式有2种：电平触发和边沿触发

(2)电平触发就是说GPIO上的电平只要满足条件，就会不停的触发中断。电平触发分为高电平触发和低电平触发。

(3)边沿触发分为上升沿触发、下降沿触发和双边沿触发三种。边沿触发不关心电平的常规状态，只关心电平变化的瞬间（边沿触发不关心电平本身是高还是低，只关心变化是从高到低还是从低到高这个过程）。

分析按键的工作：如果我们关注的是按键按下和弹起这两个事件本身，那么应该用边沿触发来处理按键；如果我们关心的是按键按下/弹起的那一段时间，那么应该用电平触发。

### 6.9.3关键寄存器：CON、PEND、MASK

(1)外部中断的主要配置寄存器有3个：EXT\_CON、EXT\_PEND、EXT\_MASK

(2)EXT\_CON配置外部中断的触发方式。

(3)EXT\_PEND是中断挂起寄存器。这个寄存器中每一位对应一个外部中断，平时没有中断时值为0。当发生了中断后，硬件会自动将这个寄存器中该中断对应的位置置1，我们去处理完这个中断后应该手工将该位置0。这个PEND寄存器的位就相当于是一个标志，如果发生了中断但是我们暂时忙来不及去处理时，这个位就一直是1（就是挂起），直到我有空了去处理了这个中断才会手工清除（写代码清除）这个挂起位表示这个中断被我处理了。

(4)EXT\_MASK寄存器就是各个外部中断的使能/禁止开关。

分析210开发板的按键对应的EINT编号：

EINT2、EINT3、EINT16、EIN717、EINT18、EINT19

## 6.10中断方式处理按键编程实战

.外部中断对应的GPIO模式设置

.中断触发模式设置

.中断允许、清挂起

.中断处理程序isr的编写

.总结对比：轮询方式处理按键和中断方式的差异

# 七、定时器、看门狗和RTC

## 7.1什么是定时器

### 7.1.1定时器是SoC中常见外设

(1)定时器与计数器。计数器是用来计数的（每隔一个固定时间会计一个数）；因为计数器的计数时间周期是固定的，因此到了一定时间只要用计数值\*技术实践周期，就能得到一个时间段，这个时间段就是我们定的时间（这就是定时器了）

(2)定时器/计数器作为SoC外设，主要是用来实现定时执行代码的功能。定时器相对于SoC来说，就好像闹钟对于人来说的意义一样。

### 7.1.2定时器有什么用

(1)定时器可以让SoC在执行主程序的同时，可以（通过定时器）具有计时功能，到了一定时间（计时结束）后，定时器会产生中断提醒CPU，CPU会去处理中断并执行定时器中断的ISR，从而去执行预先设定好的事件。

(2)定时器就好像是CPU的秘书一样，这个秘书专门管帮CPU来计时，并到时间后提醒CPU要做某件事情 。所以CPU有了定时器之后，只需预先把自己xx时间之后必须要做的事情绑定到定时器中断ISR即可，到了时间之后定时器就会以中断的方式来提醒CPU来处理这个事情。

### 7.1.3定时器的原理

(1)定时器计时其实是通过计数来实现的。定时器内部有一个计数器，这个计数器根据一个时钟（这个时钟来自于ARM的APB总线，然后经过时钟模块内部的分频器来分频得到）来工作。每隔一个时钟周期，计数器就计数一次，定时器的时间就是计数器计数值\*时钟周期。

(2)定时器内部有1个寄存器TCNT，计时开始时，我们会把一个总的计数值（譬如说300）放入TCNT寄存器中，然后每隔一个时钟周期（假设是1ms）TCNT中的值就会自动减1（硬件自动完成，不需要CPU软件区干预），直到TCNT中减为0的时候，TCNT就会触发定时器中断。

(3)定时时间由2个东西共同决定的：一个是TCNT中的计数值，一个是时钟周期。譬如上例中，定时周期就为300\*1ms = 300ms。

### 7.1.4定时器和看门狗、RTC、蜂鸣器的关系

(1)这几个东西都是和时间有关的部件。

(2)看门狗其实就是一个定时器，只不过定时时间到了之后不只是中断，还可以复位CPU。（看门狗其实定一个计数时间100，当时间到了之前如果没有喂狗（重置计数值为100），那么在到时间时会复位CPU）

(3)RTC是实时时钟，它和定时器的差别就好像闹钟（定时器）和钟表（RTC）的差别一样。

(4)蜂鸣器是一个发声设备，在ARM里蜂鸣器是用定时器模块来驱动的。

## 7.2 S5PV210中的定时器

* 在S5PV210内部，一共有4类定时器件。这4类定时器件的功能、特征是不同的。

### 7.2.1 PWM定时器(Pulse Width Modulation Timer)

(1)这种是最常用的，平时所说的定时器一般指的是这个。像简单单片机（譬如51单片机）中的定时器也是这类。

(2)为什么叫PWM定时器，因为一般SoC中产生PWM信号都是靠这个定时器模块的。

### 7.2.2系统定时器

(1)系统（指的是操作系统）定时器，系统定时器也是用来产生固定时间间隔（TCNT\*时钟周期）信号的，称为systick，这个systick用来给操作系统提供tick信号。

(2)产生systick作为操作系统的时间片（time slice）的。

(3)一般做操作系统移植的时候，这里不会由我们自己来做，一般原厂提供的基础移植部分就已经包含了。

### 7.2.3看门狗定时器

(1)看门狗定时器本质也是一个定时器，和上面2个没有任何本质区别。

(2)看门狗定时器可以设置在时间到了的时候产生中断，也可以选择发出复位信号复位CPU。

(3)看门狗定时器在实践中应用很多，尤其是工业领域（环境复杂、干扰多）机器容易出问题，而且出问题后果很严重，此时一般都会用看门狗定时器来进行系统复位。

### 7.2.4实时时钟RTC(Real Time Clock)

略

## 7.3 S5PV210的PWM定时器

### 7.3.1为什么叫PWM定时器

(1)叫定时器说明它本质上的原理是定时器。

(2)叫PWM定时器，是因为这个定时器天然是用来产生PWM波形的。

### 7.3.2 PWM定时器介绍

(1)S5PV210有5个PWM定时器。其中0、1、2、3各自对应一个外部GPIO，可以通过这些对应的GPIO产生PWM波形信号并输出；timer4没有对应的外部GPIO（因此不是为了产生PWM波形，而是为了产生内部定时器中断而生的）。

(2)S5PV210的5个PWM定时器的时钟源为PCLK\_PSYS,timer0和timer1共同使用一个预分频器，timer2、3、4共同使用一个预分频器；每个timer有一个专用的独立的分频器；预分频器和分频器构成了2级分频系统，将PCLK\_PSYS两级分频后生成的时钟供给timer模块作为时钟周期。

### 7.3.3 S5PV210的PWM定时器框图简介

关键点：时钟源、预分频器、分频器、TCMPB & TCNTB、dead zone

### 7.3.4预分频器和分频器

(1)两级分频是串联（级联）的，所以两级分频的分频数是相乘的。

(2)两级分频的分频系数分别在TCFG0和TCFG1两个寄存器中设置。

(3)预分频器有2个，prescaler0为timer0和timer1共用；prescaler1为timer2、3、4共用；两个prescaler都是8bit的，因此prescaler value范围为0~255；所以预分频器的分频值范围为1~256（注意实际分频值为prescaler value + 1）。

(4)分频器实质上是一个MUX开关，多选一开关决定了走哪个分频系数路线。可以选择的有1/1、1/2、1/4、1/8、1/16等。

(5)计算一下，两级分频下来，分频最小为1/2，最大分频为1/(256\*16)

(6)在PCLK\_PSYS为66MHz的情况下（默认时钟设置就是66MHz的），此时两级分频后的时钟周期范围为0.03us到62.061us；再结合TCNTB的值的设置（范围为1~2exp32），可知能定出来的时间最长为266548.27s（74h多）

### 7.3.5 TCNT&TCMP、TCNTB&TCMPB、TCNTO

(1)TCNT和TCNTB是相对应的，TCNTB是有地址的寄存器，供程序员操作；TCNT在内部和TCNTB相对应，它没有寄存器地址，程序员不能编程访问这个寄存器。

(2)TCNT寄存器功能就是用来减1的，它是内部的不能读写；我们想TCNT中写要通过TCNTB往进写；读取TCNT寄存器中的值要通过读取相对应的TCNTO寄存器。

(3)工作流程就是：我们事先算好TCNT寄存器中开始减的那个数（譬如300），然后将之写入TCNTB寄存器中，在启动timer前，将TCNTB中的值刷到TCNT寄存器中（有一位寄存器专门用来操作刷数据过去的），刷过去后就可以启动定时器开始计时；在计时过程中如果想知道TCNT寄存器中的值减到多少了，可以读取相应的TCNT0寄存器来得知。

(4)定时功能只需要TCNT、TCNTB两个即可;TCNTO寄存器用来做一些捕获计时；TCMPB用来生成PWM波形。

### 7.3.6自动重载和双缓冲（auto-reload and double buffering）

(1)定时器工作的时候，一次定时算是一个工作循环。定时器默认是单个循环工作的，也就是说定时一次，计时一次，到期中断一次就完了。下次如果还要再定时中断，需要另外设置。

(2)但是现实中用定时器来做的时候往往是循环的，最简单的方法就是写代码反复重置定时器寄存器的值（在每次中断处理的isr中再次给TCNTB中赋值，再次刷到TCNT中再次启动定时器），早期的单片机定时器就是这样的；但是现在高级SoC中的定时器已经默认内置了这种循环定时工作模式，就叫做自动装载（auto-reload）机制。

(3)自动装载机制就是当定时器初始化好开始计时后再不用管了，它一个周期到后，会自己从TCNTB中再次装载值到TCNT中，再次启动定时器开始下一个循环。

### 7.3.7什么是PWM

(1)PWM（pulse width modulation 脉宽调制）

(2)PWM波形是一个周期性波形，周期为T，在每个周期内波形是完全相同的。每个周期由一个高电平和低电平组成。

(3)PWM波形有2个重要参数：一个是周期T，另一个是占空比duty（占空比就是一个周期内高电平的时间除以周期时间的商）。

(4)对于一个PWM波形，知道了T和占空比duty，就可以算出这个波形的所有细节。譬如高电平时间为T\*duty，低电平时间为T\*(1-duty)。

(5)PWM波形有很多用处，譬如通信上用PWM来进行脉宽调制对基波进行载波调制；

在发光二极管LED照明领域可以用PWM波形来调制电流进行调光；用来驱动蜂鸣器等设备。

### 7.3.8 PWM波形的生产原理

(1)PWM波形其实就是用时间来控制电平高低，所以用定时器来实现PWM波形是天经地义的。

(2)早期的简单单片机里（譬如51单片机）是没有专用的PWM定时器的，那时候我们需要自己结合GPIO和定时器模块来手工生产PWM波形（流程是这样：先将GPIO引脚电平拉高、同时启动定时器定T\*duty时间，时间到了在isr中将电平拉低，然后定时T\*(1-duty)后再次启动定时器，然后时间到了后在isr中将电平拉高，然后再定时T\*duty时间再次启动定时器......如此循环即可得到周期为T，占空比为duty的PWM波形）。

(3)后来因为定时器经常和PWM产生纠结在一起，所以设计SoC的时候就直接把定时器和一个GPIO引脚内部绑定起来了，然后在定时器内部给我们设置了PWM产生的机制，可以更方便的利用定时器产生PWM波形。此时我们利用PWM定时器来产生PWM波形再不用中断了。绑定了之后坏处就是GPIO引脚是固定的、不能随便换的，好处是不用进入中断isr中，直接可以生产PWM。

(4)在S5PV210中，PWM波形产生有2个寄存器很重要，一个是TCNTB，一个是TCMPB。其中，TCNTB决定了PWM波形的周期，TCMPB决定了PWM波形的占空比。

(5)最终生产的PWM波形的周期是：TCNTB\*时钟周期（PCLK\_PSYS经过二级分频后得到的时钟周期）。注意这个周期是PWM中高电平+低电平的总时间，不是其中之一

(6)最终生成的PWM波形的占空比是TCMPB/TCNTB

### 7.3.9输出电平翻转器

(1)PWM定时器可以规定：当TCNT>TCMPB时为高电平，当TCNT<TCMPB时为低电平；也可以规定：当TCNT<TCMPB时为低电平，当TCNT>TCMPB时为高电平。在这两种规定下，计算时TCMPB寄存器的值会变化。

(2)基于上面讲的，当duty从30%变到70%时，我们TCMPB寄存器中的值就要改（譬如TCNTB

中是300时，TCMPB就要从210变化到90）。这样的改变可以满足需要，但是计算有点麻烦，于是x210的PWM定时器帮我们提供了一个友好的工具叫做电平翻转器。

(3)电平翻转器在电路上实质就是一个电平取反的部件，在编程上反映为一个寄存器位。写0就关闭输出电平翻转，写1就开启输出电平翻转。开启后和开启前输出电平则刚好高低翻转。（输出电平一翻转30%的duty就变成了70%）

### 7.3.10死区生成器

(1)PWM有一个应用就是用在功率电路中来对交流电压进行整流。整流时2路整流分别在正电平和负电平时导通工作，不能同时导通（同时导通会直接短路，瞬间的同时导通会导致电路被烧毁）。大功率的开关电源、逆变器等设备广泛使用了整流技术。特别是逆变器，用SoC的GPIO输出的PWM波形来分别驱动2路整流的IGBT。

(2)PWM波形用来做整流时要求不能同时高或低，因为会短路。但是实际电路是不理想的，不可能同时上升/下降沿，所以比较安全的做法是留死区。

(3)死区这东西离不了也多不了。死区少了容易短路，死区多了控制精度低了不利于产品性能的提升。

(4)S5PV210给我们提供了再带的死区生成器，只要开启死区生成器，生产出来的PWM波形就自带了死区控制功能，用户不用再自己去操心死区的问题。

(5)大部分人工作时用不到这个的，直接关掉死区生成器即可。

## 7.4蜂鸣器和PWM定时器编程实践

### 7.4.1蜂鸣器的工作原理

(1)蜂鸣器里边有2个金属片，离得很近但没挨着。没电的时候两个金属片在弹簧本身的张力作用下分开彼此平行，有电的时候两边分别充电，在异性电荷的吸力作用下两个片挨着。

(2)我们只要以快速的频率给蜂鸣器的正负极供电、断电，进行这样的循环，蜂鸣器的两个弹簧片就会挨着分开挨着分开···形成敲击，发出声音。

(3)人耳能听见的声音频率是有限的（20Hz~20000Hz），我们实验时一般给个2KHz的频率。

(4)频率高低会影响声音的音频，一般音频越低声音听起来越低沉、音频越高听起来越尖锐。

(5)据以上分析，只要用PWM波形的电压信号来驱动蜂鸣器，把PWM波形的周期T设置为要发出的声音信号的1/频率即可；PWM的占空比只要确保能驱动蜂鸣器即可（驱动能力问题，一般引脚驱动能力都不够，所以蜂鸣器会额外用三极管来放大来供电）。

### 7.4.2原理图和硬件信息

(1)查原理图知，开发板底板上的蜂鸣器通过GPD0\_2(XpwmTOUT2)引脚链接在SoC上。

(2)GPD0\_2引脚通过限流电阻接在三极管基极上，引脚有电蜂鸣器就会有电（三极管导通）；引脚没电蜂鸣器就会没电（三极管关闭）。我们软件只要控制GPD0\_2引脚的电平产生PWM波形即可。

(3)GPD0CON(0xE02000A0)，要把bit8~bit11设置为0b0010（功能选择为TOUT\_2,就是把这个引脚设置为PWM输出功能）。

(4)从GPD0\_2引脚可以反推出使用的是timer2这个PWM定时器。

### 7.4.3 PWM定时器的主要寄存器

TCFG0、TCFG1、CON、TCNTB2、TCMPB2、TCNTO2

注：PWM定时器产生PWM波形是不需要中断干预的

## 7.5看门狗定时器

### 7.5.1什么是看门狗、有何用

(1)看门狗定时器和普通定时器并无本质区别。定时器可以设定一个时间，在这个时间完成之前定时器不断计时，时间到的时候定时器会复位CPU（重启系统）。

(2)系统正常工作的时候当然不希望被重启，但是系统受到干扰、极端环境等可能会产生异常工作或者不工作，这种状态可能会造成不良影响（至少是不工作），此时解决方案就是重启系统。

(3)普通设备重启不是问题，但是有些设备人工重启存在困难。这时我们希望系统能够自己检验自己是否已经跑飞，并且在意识到自己跑飞的时候，可以很快的（几个ms或者更短）自我重启。这个功能就要靠看门狗定时器来实现。

(4)典型应用情景是：我们在应用程序中打开看门狗设备，初始化好给它一个时间，然后应用程序使用一个线程来喂狗，这个线程的执行时间安全短于看门狗的复位时间。当系统（或者应用程序）异常后，喂狗线程自然就不工作了，然后到时候看门狗就会复位。

(5)补充：实战中有时候为了绝对可靠，我们并不会使用SoC中自带的看门狗，而是使用专门的外置的看门狗芯片来实现看门狗。

### 7.5.2 S5PV210看门狗定时器的结构框图

(1)PCLK\_PSYS经过两级分频后生成WDT(watchdog timer)的时钟周期，然后把要定的时间写到WTDAT寄存器中，刷到WTCNT寄存器中去减1，减到0时（定时时间到）产生复位信号或中断信号。

(2)典型应用中是配置为产生复位信号，我们应该在WTCNT寄存器减到0之前给WTDAT寄存器中重新写值以喂狗。

### 7.5.3看门狗定时器的主要寄存器

WTCON WTDAT WTCNT WTCLRINT

## 7.6实时时钟RTC

### 7.6.1何为实时时钟

(1)real time clock，真实时间，就是所谓的xx年x月x日x时x分x秒星期x

(2)RTC是SoC中一个内部外设，RTC有自己独立的晶振提供RTC时钟源（32.768KHz），内部有一些寄存器用来记录时间（年月日时分秒星期）。一般情况下为了在系统关机时时间仍然在走，还会给RTC提供一个电池供电。

### 7.6.2 S5PV210实时时钟的结构框图

(1)时间寄存器7个

(2)闹钟发生器

### 7.6.3闹钟发生器

(1)可以定闹钟时间，到时间会产生RTC alarm interrupt，通知系统闹钟定时到了。

(2)闹钟定时是定的时间点，而timer定时是定的时间段。

### 7.6.4 S5PV210实时时钟的主要寄存器

(1)INTP 中断挂起寄存器

(2)RTCCON RTC控制寄存器

(3)RTCALM ALMxxx 闹钟功能有关的寄存器

(4)BCDxxx 时间寄存器

### 7.6.5 BCD码

(1)RTC中所有的时间（年月日时分秒星期，包括闹钟）都是用BCD码编码的。

(2)BCD码本质上是对数字的一种编码。用来解决这种问题：由56得到0x56（或者反过来）。也就是说我们希望十进制的56可以被编码成56（这里的56不是十进制56，而是两个数字5和6）。

(3)BCD码的作用在于可以将十进制数拆成组成这个十进制数的各个数字的编码，变成编码后就没有位数的限制了。譬如我有很大的书123456789123456789，如果这个数纯粹当数字肯定超出了int的范围，计算机无法直接处理。要想让计算机处理这个数，计算机首先能表达这个数，表达的方式就是先把这个数转成对应的BCD码（123456789123456789）

(4)BCD码在计算机中可以用十六进制的形式来表示。也就是说十进制的56转成BCD码后是 56，在计算机中用0x56来表示（暂时存储与运算）。

(5)需要写2个函数，一个bcd转十进制，一个是十进制转bcs。当我们要设置时间时（譬如设置为23分），我们需要将这个23转成0x23然后再赋值给相应的寄存器BCDMIN；当我们从寄存器BCDMIN中读取一个时间时（譬如读取到的是0x59），需要将之当做BCD码转成十进制再去显示（0x59当做BCD码就是59，转成十进制的59，所以显示就是59分）。

## 7.7 RTC编程实战

### 7.7.1设置时间与读取显示时间

(1)为了安全，默认情况下RTC读写是禁止的，此时读写RTC的时间都是不允许的，当我们要更改RTC时间时，应该先打开RTC的读写开关，然后再进行读写操作，操作完了后立即关闭读写开关。

(2)读写RTC寄存器时，一定要注意BCD码和十进制之间的转换。

(3)年的问题。S5PV210中做了个设定，BCDYEAR寄存器存的并不是完整的年数（譬如2019年），而是基于2000年的偏移量来存储的，譬如2019年实际存的就是19（2019-2000）。还有些RTC芯片是以1970年为基点来记录的。

### 7.7.2闹钟实验

# 八、SD卡启动详解

## 8.1主流的外存设备介绍

内存和外存的区别：一般是把这种RAM(random access memory,随机访问存储器，特点是任意字节读写，掉电丢失)叫内存，把ROM（read only memory，只读存储器，类似于Flash SD卡之类的，用来存储东西，掉电不丢失，不能随机地址访问，只能以块为单位来访问）叫外存。

### 8.1.1软盘、硬盘、光盘、CD、磁带

(1)存储原理大部分为磁存储，缺点是读写速度慢、可靠性差等。优点是技术成熟、价格便宜。广泛使用在桌面电脑中，在嵌入式设备中几乎无使用。

(2)现代存储的发展方向是Flash存储，闪存技术是利用电学原理来存储1和0，从而制成存储设备。所以闪存设备没有物理运动（硬盘中的磁头），所以读写速度可以很快，且无物理损耗。

### 8.1.2纯粹的Flash：NandFlash、NorFlash

(1)这些是最早出现的、最原始的Flash颗粒组成芯片。也就是说NandFlash、NorFlash芯片中只是对存储单元做了最基本的读写接口，然后要求外部的SoC来提供Flash读写的控制器以和Flash进行读写时序。

(2)缺陷：1、读写接口时序比较复杂。2、内部无坏块处理机制，需要SoC自己来管理Flash的坏块；3、各家厂家的Flash接口不一致，甚至同一个厂家的不同型号、系列的Flash接口都不一致，这就造成产品升级时很麻烦。

(3)NandFlash分MLC和SLC两种。SLC技术比较早，可靠性高，缺点是容量做不大（或者说容量大了太贵，一般SLC Nand都是512MB以下）；MLC技术比较新，不成熟，可靠性差，优点是容量可以做很大很便宜，现在基本都在发展MLC技术。

### 8.1.3 SD卡、MMC卡、MicroSD、TF卡

(1)这些卡其实内部就是Flash存储颗粒，比直接的Nand芯片多了统一的外部封装和接口。

(2)卡都有统一的标准，譬如SD卡都是遵照SD规范来发布的。这些规范规定了SD卡的读写速度、读写接口时序、读写命令集、卡大小尺寸、引脚个数及定义。这样做的好处就是不同厂家的SD卡可以通用。

### 8.1.4 iNand、MoviNand、eSSD

(1)电子产品如手机、相机等，前些年趋势是用SD卡/TF卡等扩展存储容量；但是近年来的趋势是直接内置大容量Flash芯片而不是外部扩展卡。

(2)外部扩展卡时间长了卡槽可能会接触不良导致不可靠。

(3)现在主流的发展方向是使用iNand、MoviNand、eSSD（还有别的一些名字）来做电子产品的存储芯片。这些东西的本质还是NandFlash，内部由Nand的存储颗粒构成，再集成了块设备管理单元，综合了SD卡为代表的各种卡的优势和原始的NandFlash芯片的优势。

(4)优势：1、向SD卡学习，有统一的接口标准（包括引脚定义、物理封装、接口时序）。2、向原始的Nand学习，以芯片的方式来发布而不是以卡的方式；3、内部内置了Flash管理模块，提供了诸如坏块管理等功能，让Nand的管理容易了起来。

### 8.1.5 SSD（固态硬盘）

## 8.2 SD卡的特点和背景知识

### 8.2.1 SD卡和MMC卡的关系

(1)MMC标准比SD标准早，SD标准兼容MMC标准。

(2)MMC卡可以被SD读卡器读写，而SD卡不可以被MMC读卡器读写。

### 8.2.2 SD卡和Nand、Nor等Flash芯片差异

(1)SD卡/MMC卡等卡类有统一的接口标准，而Nand芯片没有统一的标准（各家产品会有差异）

### 8.2.3 SD卡与MicroSD的区别

体积大小区别而已，传输与原理完全相同。

### 8.2.4 SD卡与TF卡的区别

(1)外观上，SD卡大而TF卡小；用途上，SD卡用于数码相机等而TF卡广泛用于手机、GPS等；

(2)时间上，SD卡1999年推出，TF卡于2004年推出；SD卡由日本松下、东芝与美国SanDisk共同推出，而TF卡由Motorola与SanDisk共同推出。

(3)SD卡有写保护而TF卡没有，TF卡可以通过卡套转成SD卡使用。

## 8.3 SD卡的编程接口

### 8.3.1 SD卡的物理接口

SD卡由9个针脚与外界进行物理连接，这9个脚中有2个地，1个电源，6个信号线。

### 8.3.2 SD协议与SPI协议

(1)SD卡与SRAM/DDR/SROM之类的东西的不同：SRAM/DDR/SROM之类的存储芯片是总线式的，只要连接上初始化好之后就可以由SoC直接以地址方式来访问，但是SD卡不能直接通过接口给地址来访问， 它的访问需要按照一定的接口协议（时序）来访问。

(2)SD卡虽然只有一种物理接口，但是却支持两种读写协议：SD协议和SPI协议。

### 8.3.3 SPI协议的特点（低速、接口操作时序简单、适合单片机）

(1)SPI协议是单片机中广泛使用的一种通信协议，并不是为SD卡专门发明的。

(2)SPI协议相对SD协议来说速度比较低。

(3)SD卡支持SPI协议，就是为了单片机方便使用。

### 8.3.4 SD协议的特点（高速、接口时序复杂，适合有SDIO接口的SoC）

(1)SD协议是专门用来和SD卡通信的。

(2)SD协议要求SoC中有SD控制器，运行在高速率下，要求SoC的主频不能太低。

### 8.3.5 S5PV210的SD/MMC控制器

(1)数据手册Section8.7，为SD/MMC控制器介绍、

(2)SD卡内部出了存储单元Flash外，还有SD卡管理模块，我们SoC和SD卡通信时，通过9针引脚以SD协议/SPI协议向SD卡管理模块发送命令、时钟、数据等信息，然后从SD卡返回信息给SoC来交互。工作时每一个任务（譬如初始化SD卡、、譬如读一个块、譬如写、譬如擦除......）都需要一定的时序来完成（所谓时序就是先向SD卡发送xx命令，SD卡回xx消息，然后再向SD卡发送xx命令......）。

## 8.4 S5PV210的SD卡启动详解

### 8.4.1 SoC为何要支持SD卡启动

(1)一个普遍性的原则就是：SoC支持的启动方式越多，将来使用时就越方便，用户的可选择性就越大，SoC的适用面就越广。

(2)SD卡有一些好处：譬如可以在不借用专用烧录工具（类似Jlink）的情况下对SD卡进行刷机，然后刷机后的SD卡插入卡槽，SoC即可启动；譬如可以用SD卡启动进行量产刷机（量产卡）。像我们x210开发板，板子贴片好的时候，内部inand是空的，此时直接启动无启动，因为此时inand是空的，所以第一启动失败，会转入第二启动，就从外部SD2通道的SD卡启动了。启动后会执行刷机操作对iNand进行刷机，刷机完成后自动重启（这回重启时iNand中已经有image可，所以可以启动了）。刷机完成后SD量产卡拔掉，烧机48小时，无死机即可装箱待发货。

### 8.4.2 SD卡启动的难点在哪里（SRAM、DDR、SDcard）

(1)SRAM、DDR都是总线式访问的，SRAM不需要初始化即可直接使用，而DDR需要初始化后才能使用，但是总之CPU可以直接和SRAM和DDR打交道；而SD卡需要时序访问，CPU不能直接和SD卡打交道；NorFlash读取时可以总线式访问，所以NorFlash启动非常简单，可以直接启动，但是SD/NandFlash不行。

(2)以前只有NorFlash可以作为启动介质，台式机笔记本的BIOS就是NorFlash做的介质。后来三星2440中使用了SteppingStone的技术，让NandFlash也可以作为启动介质。SteppingStone（启动基石）技术就是在SoC内部内置4KB的SRAM，然后开机时SoC根据OMPin判断用户设置的启动方式，如果是NandFlash启动，则Soc的启动部分的硬件直接从外部NandFlash中读取开头的4KB到内部SRAM作为启动内容。

(3)启动基石技术进一步发展，在6410芯片中得到完善，在210芯片时已经完全成熟。210中有96KB的SRAM，并且有一段iROM代码作为BL0，BL0再去启动BL1（210中的BL0做的事情在2440中也有，只不过那时候是硬件自动完成的，而且体系没有210中这么详细）。

### 8.4.3 S5PV210的启动过程回顾

(1)210启动首先执行内部的iROM（也就是BL0），BL0会判断OMPin来决定从哪个设备启动，如果启动设备是SD卡，则BL0会从SD卡读取前16KB到iSRAM中去启动执行（这部分就是BL1，这就是SteppingStone技术）

(2)BL1执行之后剩下的就是软件的事情了，SoC就不用再去操心了。

### 8.4.4 SD卡启动流程（bin文件小于16KB时和大于16KB时）

(1)启动的第一种情况是整个镜像大小小于16KB。这时候相当于我的整个镜像作为BL1被SteppingStone直接硬件加载执行了而已。

(2)启动的第二种情况就是整个镜像大小大于16KB。（只要大于16KB，哪怕是17KB，或者是700MB都是一样的）这时候就要把整个镜像分为2部分：第一部分16KB大小，第二部分是剩下的大小。然后第一部分作为启动介质，负责去初始化DRAM并且将第二部分加载到DRAM中去执行（uboot就是这样做的）。

### 8.4.5最重要的但是却隐含未讲的东西

(1)问题：iROM究竟是怎样读取SD卡/NandFlash的？

(2)三星在iROM中事先内置了一些代码去初始化外部SD卡/NandFlash，并且内置了读取各种SD卡/NandFlash的代码在iROM中。BL0执行时就是通过调用这些device copy function来读取外部SD卡/NandFlash中的BL1的。

### 8.4.6 SoC支持SD卡启动的秘密（iROM代码）

三星系列SoC支持SD卡/NandFlash启动，主要是依靠SteppingStone技术，具体在S5PV210中支持SteppingStone技术的是内部的iROM代码。

iROM application note：block device copy function

### 8.4.7扇区和块的概念

(1)早期的块设备就是软盘硬盘这类磁存储设备，这种设备的存储单元不是以字节为单位，而是以扇区为单位。磁存储设备读写的最小单元就是扇区，不能只读取或写部分扇区。这个限制是磁存储设备本身物理方面的原因造成的，也成为了我们编程时必须遵守的规律。

(2)一个扇区有好多个字节（一般是512个字节）。早期的磁盘扇区是512字节，实际上后来的磁盘扇区可以做的比较大（譬如1024字节，譬如2048字节，譬如4096字节），但是因为原来最早是512字节，很多的软件（包括操作系统和文件系统）已经默认了512这个数字，因此后来的硬件虽然物理上可能支持更大的扇区，但是实际上一般还是兼容512字节扇区这种操作方法。

(3)一个扇区可以看成是一个块block（块的概念就是：不是一个字节，是多个字节组成一个共同的操作单元块），所以就把这一类的设备称为块设备。常见的块设备有：磁存储设备硬盘、软盘、DVD和Flash设备（U盘、SSD、SD卡、NandFlash、Norflash、eMMC、iNand）

(4)linux里有个mtd驱动，就是用来管理这类块设备的。

(5)磁盘和Flash以块为单位来读写，就决定了我们启动时device copy function只能以整块为单位来读取SD卡。

### 8.4.8用函数指针方式调用device copy function

(1)第一种方法：宏定义方式来调用。好处是简单方便，坏处是编译器不能帮我们做参数的静态类型检查。

(2)第二种方法：用函数指针方式来调用。

## 8.5 S5PV210的SD卡启动实战

### 8.5.1任务：大于16KB的bin文件使用SD卡启动

(1)总体思路：将我们的代码分为2部分，第一部分BL1小于等于16KB，第二部分为任意大小，iROM代码执行完成后从SD卡启动会自动读取BL1到iRAM中执行；BL1执行时负责初始化DDR，然后手动将BL2从SD卡copy到DDR中正确位置，然后BL1远跳转到BL2中执行BL2。

(2)细节1：程序怎么安排？程序分为2个文件夹BL1和BL2，各自管理各自的项目。

(3)细节2：BL1中要完成：关看门狗、设置栈、开iCache、初始化DDR、从SD卡复制BL2到DDR中特定位置，跳转执行BL2。

(4)细节3：BL1在SD卡中必须从Block1开始（Block0不能用，这是三星官方规定的），长度为16KB内，我们就定位16KB（也就是32个block）；BL2理论上可以从33扇区开始，但是实际上为了安全都会留一些空扇区作为隔离，譬如可以从45扇区开始，长度由自己定（实际根据自己的BL2大小来分配，我们实验时BL2非常小，因此我们定义BL2长度为16KB，也就是32扇区）。

(5)细节4：DDR初始化好后，整个DDR都可以使用了，这时在其中选择一段长度足够BL2的DDR空间即可。我们选0x23E00000（因为我们BL1中只初始化了DDR1，地址空间范围是0x20000000~0x2fffffff）。

### 8.5.2代码划分为2部分（BL1和BL2）

BL1中重定位

BL2远跳转

(1)因为我们BL1和BL2其实2个独立的程序，链接时也是独立分开链接的，所以不能像以前一样使用ldr pc, =main这种方式来通过链接地址实现远跳转到BL2。

(2)我们的解决方案是使用地址进行强制跳转，因为我们知道BL2在内存地址0x23E00000处，所以直接去执行这个地址即可。

总结:代码分为2部分，这种技术叫做分散加载。这种分散加载的方法可以解决问题，但是比较麻烦。它的缺陷有：第一，代码完全分2部分，完全独立，代码编写和组织上麻烦；第二，无法让工程兼容SD卡启动和Nand启动、NorFlash启动等各种启动方式。

### 8.5.3 uboot中的做法

(1)第二种思路：程序代码仍然包括BL1和BL2两部分，但是组织形式上部分为2部分，而是作为一个整体来组织。它的实现方式是：iROM启动，然后从SD卡的扇区1开始读取16KB的BL1然后去执行BL1，BL1负责初始化DDR，然后从SD卡中读取整个程序(BL1+BL2)到DDR中，然后从DDR中执行（利用ldr pc, =main这种方式以远跳转到iRAM中运行的BL1跳转到DDR中运行BL2）。

(2)细节1：uboot编译好后有200多KB，超出了16KB。uboot的组织方式就是前面16KB为BL1，剩下的部分为BL2。

(3)细节2：uboot在烧录到SD卡的时候，先截取uboot.bin的前16KB(实际脚本截取的是8KB)烧录到SD卡的block1~block32，然后将整个uboot烧录到SD卡的某个扇区中（譬如49扇区）

(4)细节3：实际uboot从SD卡启动时是这样的：iROM先执行，根据OMpin判断出启动设备是SD卡，然后从SD卡的block1开始读取16KB(8KB)到iRAM中执行BL1，BL1执行时负责初始化DDR，并且从SD卡的49扇区开始复制整个uboot到DDR中指定位置（0x23E00000)去备用；然后BL1继续执行直到ldr pc, =main时BL1跳转到DDR上的BL2中接着执行uboot的第二阶段。

总结：uboot中的这种启动方式比分散加载的好处在于：能够兼容各种启动方式。

我的总结：假设启动程序为100KB，分散加载将程序分为2部分，一部分BL1为16KB，另一部分BL2为84KB（100-16）。这两部分各有各的链接地址，其中BL1一般是链接在0xd0020010，而BL2链接在0x23E00000。BL1负责初始化DDR，程序运行中将BL2复制到DDR中的相应位置，然后跳转到0x23E00000位置继续执行BL2。

uboot方式假设启动程序为100KB，将程序分为2部分，一部分BL1为16KB，一部分为BL1+BL2 100KB。这两部分链接在同一个地址0xd0020010。BL1负责初始化DDR和将BL1+BL2 copy到DDR中，然后在执行过程中使用ldr pc, =main 语句跳转到DDR中继续执行BL2。

# 九、NandFlash和iNand

## 9.1 NandFlash的接口

### 9.1.1 Nand的型号与命名

(1)K9F2G08：K9F表示是三星公司的NandFlash系列。2G表示Nand的大小是2Gbit(256MB)。08表示Nand是8位的（数据线有8根）。

(2)Nand命名中可以看出，厂家、系列型号、容量大小、数据位数

### 9.1.2 Nand的数据位

(1)Nand有8位数据位的，也有16位数据位的。做电路时/写软件时应该根据自己实际采购的Nand的位数来设计电路/写软件。

(2)有8位数据位，说明Nand是并行接口的(8/16位)。

(3)Nand的数据线上传递的不一定全部都是数据，也有可能有命令、地址等。

### 9.1.3 Nand的功能框图

(1)Nand的结构可以看成是一个矩阵式存储器。其中被分成一个一个的小块，每一小块可以存储一个bit位，然后彼此以一定单位组合成整个Nand。

(2)Nand中可以被单次访问的最小单元（就是说对Nand进行一次读写至少要读写这么多，或者是这么多的整数倍）叫做Page(页)，在K9F2G08芯片中，Page的大小为2KB+64B。也就是说我们要读写K9F2G08，每次至少要读写2KB或者n \* 2KB，即使我们只是想要读写其中的一个字节。这就是我们说的典型的块设备（现在有些块设备为了方便一种，提供了random read模式，是可以只读写1个字节）。

(3)页往上还有一个Block（块）的概念。1个块等于若干个页（譬如在啊K9F2G08中1个块等于64页）。

(4)块往上就是整个Nand芯片了，叫做Device。一个Device是若干个Block，譬如K9F2G08一个Device有2048个Block。所以整个Device的大小就是2048\*63\*2K = 256MB。

(5)块设备分page、分block的有什么意义？首先要明白，块设备不能完全按字节访问而必须块访问是物理上的限制，而不是人为设置的障碍。其次，Page和Block各有各的意义。譬如Nand中：Page是读写Nand的最小单位；Block是擦除Nand的最小单位。（这些规则都是Nand的物理原理和限制要求的，不是谁想要这样的，所以对软件来说只能想办法适应硬件，而不是超越硬件）。

(6)Nand芯片中主要包含2部分：Nand存储颗粒Nand接口电路。存储颗粒就是村翠的Nand原理的存储单元，类似于仓库；Nand接口电路是用来管理存储颗粒，并且给外界提供一个统一的Nand接口规格的访问接口的。

(7)Nand中有多个存储单元，每个单元都有自己的地址（地址是精确到字节的）。所以Nand是地址编排精确到字节，但是实际读写却只能精确到页（所以Nand的很多操作要求给的地址是页对齐的，譬如2K、4K、512K等这样的地址，不能给3K这样的地址）。Nand读写时地址传递是通过IO线发送的，因为地址有30位而IO只有8位，所以需要多个cycle才能发送完毕。一般的Nand都是4cycle或者5cycle发送地址（从这里把Nand分为了4cycle Nand和5cycle Nand）。

总结：Nand芯片内部有存储空间，并且有电路来管理这些存储空间，向外部提供统一的Nand接口的访问规则，然后外部的SoC可以使用Nand接口时序来读写这个Nand存储芯片。Nand接口是一种公用接口，是一种标准，理论上来说外部SoC可以直接模拟Nand接口来读写Nand芯片，但是实际上因为Nand接口对时序要求非常严格，而且时序很复杂，所以一般的SoC都是通过专用的硬件的Nand控制器（这些控制器一般是作为SoC内部外设来存在的）来操控Nand芯片的。

## 9.2 NandFlash的结构

### 9.2.1 Nand的单元组织：block与page（大页Nand与小页Nand）

(1)Nand的页和以前讲过的块设备（尤其是硬盘）的扇区是类似的。扇区最早在磁盘中是512字节，后来也有些高级硬盘扇区不是512字节，而是1024字节/2048字节/4096字节等。Nand也是一样，不同的Nand的页的大小是不同的，也有512字节/1024字节/2048字节/4096字节等。

(2)一个block等于多少page也是不定的，不同的Nand也不同。一个Nand芯片有多少个block也是不定的，不同的Nand芯片也不同。

总结：Nand的组织架构挺乱的，接口时序也不同，造成结果就是不同厂家的Nand芯片，或者是同一个厂家的不同系列型号存储容量的Nand接口也不一样。所以Nand有一个很大的问题就是一旦升级容量或者换芯片系列则硬件要重新做、软件要重新移植。

### 9.2.2带内数据和带外数据（ECC与坏块标记）

(1)Nand的每个页由2部分组成，这2部分各自都有一定的存储空间。譬如K9F2G08中2K+64B。其中2K字节数据带内数据，是我们真正的存储空间，将来存储在Nand中的有效数据就是存在这2K范围内的（我们平时计算Nand的容量时也是只考虑这2KB）；64字节的带外数据不能用来存储有效数据，是作为别的附加用途的（譬如用来存储ECC数据、用来存储坏块标志等····）

(2)什么是ECC：（error correction code，错误校验码）。因为Nand存储本身出错（位反转）概率高（Nand较Nor最大的缺点就是稳定性），所以当我们将有效信息存储到Nand中时都会同时按照一定算法计算一个ECC信息（譬如CRC16等校验算法），将ECC信息同时存储到Nand这个页的带外数据区。然后等将来读取数据时，对数据用同样的算法再计算一次ECC，并且和从带外数据区读出的ECC进行校验。如果校验通过则证明Nand的有效数据可信，如果校验不通过则证明这个数据已经被破坏（只能丢弃或者尝试修复）。

(3)坏块标志：Nand芯片用一段时间后，可能某些块会坏掉（这些块无法擦除了或者无法读写了），Nand的坏块非常类似于硬盘的坏道。坏块是不可避免的，而且

随着Nand的使用，坏块会越来越多。当坏块还不算太多时这个Nand都是可以用的，除非坏块太多了不划算使用了才会换新的。所以我们为了管理Nand发明了一种坏块标志机制。Nand的每个页的64字节的带外数据中，我们（一般是文件系统）

定义一个固定位置（譬如定位24字节）来标记这个块是好的还是坏的。文件系统在发现这个块已经坏了没法用了时会将这个块标记为坏块，以后访问Nand时直接跳过这个块即可。

### 9.2.3 Nand的地址时序

(1)Nand的地址由多位，分4/5周期通过IO引脚发送给Nand芯片来对Nand进行寻址。寻址的最小单位是字节，但是读写的最小单位是页。

(2)Nand的地址在写代码时要按照Nand要求的时序和顺序依次写入。

### 9.2.4 Nand的命令码

(1)外部SoC要想通过Nand控制器来访问Nand（实质就是通过Nand接口），就必须按照Nand接口给Nand发送命令、地址、数据等信息来读写Nand。

(2)Nand芯片内部的管理电路本身可以接受外部发送的命令，然后根据这些命令来读写Nand内容与外部SoC交互。所以我们队Nand进行的所有操作（擦除、读、写...）都要有命令、地址、数据的参与才能完成，而且必须按照Nand芯片规定的流程来做。

## 9.3 Nand的常见操作及流程分析

### 9.3.1坏块检查

(1)Flash使用之前一定要先统一擦除（擦除的单元是块）。Flash类设备擦除后里边全是1，所以擦干净之后读出来的值是0xFF

(2)检查坏块的思路就是：先擦除，然后将整个块读出来，依次检测各字节是否为0xFF，如果是则表明不是坏块，如果不是则表明是坏块。

### 9.3.2页写（program）操作

(1)写之前确保这个页是被擦干净的。如果不是擦干净的（而是脏的、用过的）页，写进去的值就是错的，不是你想要的结果。

(2)写操作(wright)在Flash的操作中就叫编程（program）

(3)SoC写Flash时通过命令线、IO线依次发送写命令、写页地址、写数据等进入NandFlash。

(4)写的过程：SoC通过Nand控制器和Nand芯片完成顺序对接，然后按照时序要求将一页数据发给Nand芯片内部的接口电路。接口电路先接收数据到自己的缓冲区，然后再集中写入Nand芯片的存储区域中。Nand接口电路将一页数据从缓冲区中写入Nand存储系统需要一定的时间，这段时间Nand芯片不能再响应SoC发过来的其他命令，所以SoC要等待Nand接口电路忙完。等待方法是SoC不断读取状态寄存器（这个状态寄存器有2种情况：一种是SoC的Nand控制器自带的，另一种是SoC通过发命令得到命令响应得到的），然后通过检查这个状态寄存器的状态位就能知道Nand接口电路刚才写的那一页数据写完了没、写好了没。直到SoC收到正确的状态寄存器响应才能认为刚才要写的那一页数据已经OK。（如果SoC收到的状态一直不对，可以考虑重写或认为这一页所在的块已经是坏块，或者整个Nand芯片已经坏了）。

(5)正常情况下到了（4）已经结束了。但是因为Nand的读写有不靠谱情况，因此，为了安全要去做ECC校验。ECC校验有硬件式校验和软件式校验2种。软件式校验可以采用的策略有很多，其中之一（Nand芯片手册上推荐的方式是）：将刚才写入的一页数据读出来，和写入的内容进行逐一对比。如果读出的和写入的完全一样说明刚才的写入过程正确完成；如果读出来的和写入的不完全一样那就说明刚才的写入有问题。

(6)硬件式ECC：SoC的Nand控制器可以提供硬件式ECC（这个也是比较普遍的情况）。硬件式ECC就是在Nand控制器中有个硬件模块专门做ECC操作。当我们操作Nand芯片时，只要按照SoC的要求按时打开ECC生成开关，则当我们写入Nand芯片时SoC的Nand控制器的ECC模块会自动生成ECC数据放在相应的寄存器中，然后我们只需将这生成的ECC数据写入Nand芯片的带外数据区即可；在将来读取这块Nand芯片时，同样要打开硬件ECC开关，然后开始读，在读的过程中硬件ECC会自动计算读进来的一页数据的ECC值并将之放到相应的寄存器中。然后我们再去读取带外数据区中原来写入时的ECC值，和我们刚才读的时候得到的ECC值进行校验。校验通过则说明读写正确，校验不通过则说明不正确（放弃数据或尝试修复）。

### 9.3.3擦除（erase）操作

(1)擦除时必须给块对齐的地址。如果给了不对齐的地址，结果是不可知的（有的Nand芯片没关系，它内部会自动将其对齐，而有些Nand会返回地址错误）。

### 9.3.4页读(read)操作

## 9.4 S5PV210的NandFlash控制器

### 9.4.1 SoC的Nand控制器的作用

### 9.4.2结构框图分析

### 9.4.3 S5PV210的Nand控制器的主要寄存器

## 9.5 Nand操作代码解析

### 9.5.1擦除函数

### 9.5.2页读取函数

### 9.5.3页写入函数

## 9.6 iNand介绍

### 9.6.1 iNand/eMMC/SDCard/MMCCard的关联

### 9.6.2 iNand/eMMC的结构框图

### 9.6.3 iNand/eMMC的物理接口和SD卡物理接口的对比

### 9.6.4结论：iNand/eMMC其实就是芯片化的SD/MMC卡，软件操作和SD卡相同

# 十、I2C通信详解

## 10.1什么是I2C通信

10.1.1物理接口：SCL + SDA

(1)SCL：时钟线，传输CLK，一般是I2C主设备向从设备提供时钟的通道。

(2)SDA：数据线，通信数据都通过SDA线传输。

### 10.1.2通信特征：串行、同步、非差分、低速

(1)I2C属于串行通信，所有数据以位为单位在SDA线上串行传输。

(2)同步通信就是通信双方工作在同一个时钟下，一般是通信的A方通过一根CLK信号线传输A自己的时钟给B，B工作在A传输的时钟下。所以同步通信的显著特征就是：通信线中有CLK。

(3)非差分：因为I2C通信速率不高，而且通信双方距离很近，所以使用电平信号通信。

(4)低速率：I2C一般是用在同一个板子上的2个IC之间的通信，而且用来传输的数据量不大，所以本身通信速率很低（一般几百KHz，不同的I2C芯片的通信速率可能不同，具体在编程时要看自己所使用的设备允许的I2C通信最高速率，不能超过这个速率）。

### 10.1.3突出特征1：主设备 + 从设备

(1)I2C通信的时候，通信双方地位是不对等的，而是分主设备和从设备。通信由主设备发起，由主设备主导，从设备只是按照I2C协议被动的接受主设备的通信，并及时响应。

(2)谁是主设备、谁是从设备是由通信双方来定的（I2C协议并无规定），一般来说一个芯片可以只能做主设备、也可以只能做从设备、也可以既能当主设备又能当从设备（软件配置）。

### 10.1.4突出特征2：可以多个设备挂在一条总线上（从设备地址）

(1)I2C通信可以一对一，也可以多对一。

(2)主设备来负责调度总线，决定某一时间和哪个从设备通信。注意；同一时间内，I2C总线上只能传输一对设备的通信信息，所以同一时间只能有一个从设备和主设备通信，其他设备处于“冬眠”状态，不能出来捣乱，否则通信就乱套了。

(3)每一个I2C从设备在通信中都有一个I2C从设备地址，这个设备地址是从设备本身固有的属性，然后通信时主设备需要知道自己将要通信的那个设备的地址，然后在通信中通过地址来甄别是不是自己要找的那个从设备。（这个地址是一个电路板上唯一的，不是全球唯一的）

10.1.5主要用途：SoC和周边外设之间的通信（典型的如EEPROM、电容触摸IC、各种sensor等）

## 10.2由I2C学通信时序

### 10.2.1什么是时序？

字面意思，时序就是时间顺序，实际上在通信中时序就是通信线上按照时间顺序发生的电平变化，以及这些变化对通信的意义就叫时序。

### 10.2.2 I2C总线空闲状态、起始位、结束位

(1)I2C总线上有1个主设备，n（n>=1）个从设备。I2C总线上有2种状态；空闲态（所有从设备都未和主设备通信，此时总线空闲）和忙态（其中一个从设备在和主设备通信，此时总线被这一对占用，其他从设备必须歇着）。

(2)整个通信分为一个周期一个周期的，两个相邻的通信周期是空闲态。每一个通信周期由一个起始位开始，一个结束位结束，中间是本周起的通信数据。

(3)起始位并不是一个时间点，而是一个时间段。在这段时间内总线状态变化情况是：SCL线维持高电平，同时SDA线发生一个从高到底的下降沿。

(4)与起始位相似，结束位也是一个时间段。在这段时间内总线的变化情况是：SCL维持高电平，同时SDA线发生一个从低到高的上升沿。

### 10.2.3 I2C数据传输格式（数据位&ACK）

(1)每一个通信周期的发起和结束都是由主设备来做的，从设备只有被动的相应主设备，没法自己自发的去做任何事情。

(2)主设备在每个通信周期会先发8位的从设备地址（其实8位中只有7位是从设备地址，还有1位表示主设备下面是要写入还是读出）到总线（主设备是以广播的形式发送的，只要是总线上的所有设备其实都能收到这个信息）。然后总线上的每个从设备都能收到这个地址，并且收到地址后和自己的设备地址比较看是否相等。如果相等说明主设备本次通信就是给我说话，如果不相等说明这次通信与我无关，不理会。

(3)发送方发送一段数据后，接收方需要回应一个ACK。这个响应本身只有1个bit位，不能携带有效信息，只能表示2个意思（要么表示收到数据，即有效响应；要么表示未收到数据，无效响应）

(4)在某一通信时刻，主设备和从设备只能有一个在发（占用总线，也就是向总线写），另一个在收（从总线读）。如果在某个时间主设备和从设备都试图向总线写就完蛋了。

### 10.2.4数据在总线上的传输协议

(1)I2C通信时的基本数据单位也是以字节为单位的，每次传输的有效数据都是1个字节（8位）。

(2)起始位及其后的8个clk中哦都市主设备在发送（主设备掌控总线），此时从设备只能读取总线，通过读总线来得知主设备发给从设备的信息；然后到了第9周期，按照协议规定从设备需要发送ACK给主设备，所以此时主设备必须释放总线（主设备把总线置为高电平然后不要动，其实就类似于总线空闲状态），同时从设备试图拉低总线发出ACK。如果从设备拉低总线失败，或者从设备根本就没有拉低总线，则主设备看到的现象就是总线在第9周期仍然一直保持高，这对主设备来说，意味着我没收到ACK，主设备就认为刚才给从设备发送的8位不对（接收失败）。

## 10.3 S5PV210的I2C控制器

通信双方本质上是通过时序在工作，但是时序会比较复杂不利于SoC软件完成，于是解决方案就是SoC内置了硬件控制器来产生通信时序。这样我们写软件时只需要向控制器的寄存器中写入配置值即可，控制器会产生适当的时序在通信线上和对方通信。

### 10.3.1结构框图

(1)时钟部分，时钟来源是PCLK\_PSYS，经过内部分频最终得到I2C控制器的CLK，通信中这个CLK会通过SCL线传给从设备。

(2)I2C总线控制逻辑（前台代表是I2CCON、I2CSTAT这两个寄存器），主要负责产生I2C通信时钟。实际编程中药发送起始位、停止位、接收ACK等都是通过这两个寄存器（背后所代表的电路模块）实现的。

(3)移位寄存器（shift register），将代码中要发送的字节数据，通过移位寄存器变成1个位一个位的丢给SDA线上去发送/接收。

(4)地址寄存器+比较器。本I2C控制器做从设备的时候用。

### 10.3.2系统分析I2C的时钟

(1)I2C时钟源来源于PCLK（PCLK\_PSYS，等于65MHz），经过了2级分频后得到的。

(2)第一级分频是I2CCON的bit6，可以得到一个中间时钟I2CCLK（等于PCLK/16或PCLK/512）

(3)第二级分频是得到最终I2C控制器工作的时钟，以I2CCLK这个中间时钟为来源，分频系数为[1,16]。

(4)最终要得到的时钟是2级分频后的时钟，譬如一个可用的设置是：65000KHz/512/4=31KHz

### 10.3.3主要寄存器I2CCON、I2CSTAT、I2CADD、I2CDS

(1)I2CCON + I2CSTAT：主要用来产生通信时序和I2C接口配置。

(2)I2CADD：用来写自己的slave address。

(3)I2CDS：发送/接收的数据都放在这里。

## 10.4 X210板载gsensor介绍

### 10.4.1原理图查询

(1)gsensor的供电由PWMTOUT3引脚控制。当PWMTOUT3输出低电平时gsensor无电不工作；当输出高电平时gsensor才会工作。

(2)gsensor的SDA和SCL接的是S5PV210的I2C端口0。

(3)将来编程时在gsensor init函数中要去初始化相关的GPIO。要把相应的GPIO设置为正确的模式和输入输出值。

### 10.4.2重力加速度传感器

(1)用在手机、平板、智能手表等设备上，用来感受人的手的移动，获取一些运动的方向性信息用来给系统作为输入参量。

(2)可以用来设计智能手表的计步器功能。

(3)重力加速度传感器、地磁传感器、陀螺仪等三个传感器结合起来，都是用来感测运动的速度、方位等信息的，所以现在有9轴传感器，就是把三者结合起来，并且用一定的算法进行综合得出结论，目的是更加准确。

(4)一般传感器的接口有2种：模拟接口和数字接口。模拟接口是用接口电平变化来作为输出的（譬如模拟接口的压力传感器，在压力不同时输出电平在0～3.3V范围内变化，每一个电压对应一个压力。），SoC需要用AD接口来对接这种传感器对它输出的数据进行AD转换，转换得到的数字电压值，再用数字电压值去校准得到的压力值；数字接口是后来发展出来的，数字机接口的sensor是在模拟接口的sensor基础上，内部集成了AD，直接（通过一定的总线接口协议，一般是I2C）输出一个数字值的参数，这样SoC直接通过总线接口初始化、读取传感器输出的参数即可（譬如gsensor、电容触摸屏IC）。

### 10.4.3 I2C从设备的设备地址

(1)KXTE9的I2C地址固定为0b0001111（0x0f）

(2)I2C从设备地址本身是7位的，但是在I2C通信中发送I2C从设备地址时实际发送的是8位，这8位中高7位（bit7~bit1）对应的I2C从设备的7位地址，最低一位（LSB）存放的是R/W信息（就是说下一个数据时主设备写从设备读（对应0），还是主设备读从设备写（对应1））

(3)基于以上，对于KXTE9来说，主设备（SoC）发给gsensor信息时，SAD应该是：0b00011110（0x1e）。如果是主设备读取gsensor信息时，SAD应该是：0b00011111（0x1f）

### 10.4.4 I2C从设备的通信速率

(1)I2C协议本身属于低速协议，通信速率不能太高。

(2)实际上通信的主设备和从设备本身都有最高的通信速率限制（属于各个芯片本身的参数），实际编程时怎么确定最终的通信速率？只要小于两个即可。

(3)一般来说只能做从设备的sensor芯片本身i2c通信速率偏低，像KXTE9最高支持400KHz的频率。

## 10.5 I2C总线的通信流程

### 10.5.1 S5PV210的主发送流程图

### 10.5.2 S5PV210的主接收流程图

### 10.5.3 gsensor的写寄存器流程图

### 10.5.4 gsensor的读寄存器流程图

## 10. 6 I2C通信代码分析

### 10. 6.1 I2C控制器初始化：s3c24xx\_i2c\_init

(1)初始化做的事情：初始化GPIO，设置IRQEN和ACKEN，初始化I2C时钟

### 10. 6.2 I2C控制器主模式开始一次读写：s3c24xx\_i2c\_message\_start

### 10. 6.3 I2C控制器主模式结束一次读写：s3c24xx\_i2c\_stop

框架分析：我们最终目的是通过读写gsensor芯片的内部寄存器来得到一些信息。为了完成这个目的，我们需要能够读写gsensor的寄存器，根据gsensor的规定我们需要按照一定的操作流程来读写gsensor的内部寄存器，这是一个层次（姑且叫做传输层、协议层、应用层）；我们要按照操作流程去读写寄存器，就需要考虑I2C接口协议（这就是所谓的物理层，本质就是那些时序）。此时主机SoC有或者没有控制器，有控制器时考虑控制器的寄存器，没控制器时要自己软件模拟时序。

协议层的代码主要取决于gsensor芯片；物理层代码主要取决于主机SoC。

### 10. 6.4 gsensor写寄存器：gsensor\_i2c\_write\_reg

### 10. 6.5 gsensor读寄存器：gsensor\_i2c\_read\_reg

### 10. 6.6 gsensor编程：gsensor\_initial等

# 十一、ADC

## 11.1 ADC的引入

### 11.1.1什么是ADC

(1)ADC:analog digital converter,AD转换，模数转换（也就是模拟转数字）

(2)CPU本身是数字的，而外部世界变量（如电压、温度、高度、压力···）都是模拟的，所以需要用CPU来处理这些外部的模拟变量的时候就需要做AD转换。

### 11.1.2为什么需要ADC

为了用数字技术来处理外部的模拟物理量

### 11.1.3关于模拟量和数字量

(1)模拟的就是连续的，现实生活当中的时间、电压、高度等都是模拟的（连续分布的，划分的话可以无限的更小划分）。模拟量反映在数学里面就是无限小数位（从0到1之间有无数个数）

(2)数字的就是离散的，离线的就是不连续的。这种离散处理实际上是从数学上对现实中的模拟量的一种有限精度的描述。数字化就是离散化，就是把连续分布的模拟量按照一定精度进行取点（采样）变成有限多个不连续分布的数字值，就叫数字量。

(3)数字化的意义就在于可以用（离散）数学来简化描述模拟量，这东西是计算机技术的基础。

(4)计算机处理参量的时候都是数字化的，计算机需要数字化的值来参与运算。如果系统输入参数中有模拟量，就需要外加AD转换器将模拟量转成数字量再给计算机。

### 11.1.4有AD自然就有DA

(1)AD是analog to digital，DA自然就是digital to analog，数字转模拟。

(2)纯粹用cpu是不可能实现数字转模拟，因为cpu本身就是数字的。使用一些（具有一些积分或微分效果的）物理器件就可实现数字转模拟。

(3)数字转模拟的作用。譬如可以用来做波形发生器。

## 11.2 ADC的主要相关概念

### 11.2.1量程（模拟量输入范围）

(1)AD转换器是一个电子器件，所以他只能输入电压信号。其他种类的模拟信号要先经过传感器（Sensor）的转换变成模拟的电压信号然后才能给AD。

(2)AD输入端的模拟电压要求有一个范围，一般是0～3.3V或0～5V或者是0～12V等等。模拟电压的范围是AD芯片本身的一个参数。实际工作时给AD的电压信号不能超过这个电压范围。

### 11.2.2精度（分辨率resolution）

(1)AD转换输出的数字值是有一定的位数的（譬如说10位，意思就是输出的数字值是用10个二进制位来表示的，这种就叫10位AD）。这个位数就表示了转换精度。

(2)10位AD就相当于把整个范围分成了1024个格子，每个格子之间的间隔就是电压的表示精度。假如AD芯片的量程是0～3.3V，则每个格子代表的电压值是3.3V/1024=0.0032265V。如果此时AD转换后得到的数字量是447，则这个数字量代表的模拟值是：447×0.0032265V=1.44V。

(3)AD的位数越多，则每个格子表示的电压值越小，将来算出来的模拟电压值就越精确。

(4)AD的模拟量程一样的情况下，AD精度位数越多精度越高，测出来的值越准。但是如果AD的量程不一样。譬如2个AD，A的量程是0～50V，B的量程是0～0.5V，A是12位的，B是10位的，可能B的精度比A的还要高。（A的精度：50/1024=0.04883，B的精度：0.5/4096=0.000122）

### 11.2.3转换速率（MSPS与conventor clock的不同）

(1)首先要明白：AD芯片进行AD转换是要耗费时间的。这个时间需要多久，不同的芯片是不一样的，同一颗芯片在配置不一样（譬如说精度配置为10位时时间比精度配置为12位时要小，譬如说有些AD可以配转换时钟，时钟频率高则转换时间短）时转换时间也不一样。

(2)详细的需要时间可以参考数据手册。一般数据手册中描述转换速率用的单位是MSPS（第一个M是兆，S是sample，就是采样；PS就是per second，总的意思就是兆样本每秒，每秒种转出来多少M个数字值）

(3)AD工作都需要一个时钟，这个时钟有一个范围，我们实际给他配置时不要超出这个范围就可以了。AD转换是在这个时钟下进行的，时钟的频率控制着AD转换的速率。注意：时钟频率和MSPS不是一回事，只是成正比不是完全相等。譬如S5PV210中的AD转换器，MSPS = 时钟频率/5

### 11.2.4通道数

AD芯片有多少路analog input通道，代表了将来可以同时进行多少路模拟信号的输入。

## 11.3 S5PV210的ADC控制器

### 11.3.1 ADC和（电阻式）触摸屏的关系

(1)ADC在210数据手册的section 10.7

(2)电阻式触摸屏本身工作时就依赖于AD转换，所以在210的SoC中电阻触摸屏接口本身和ADC接口是合二为一的。或者说电阻触摸屏接口使用了（复用了）ADC的接口。

### 11.3.2 ADC的工作时钟框图

ADCCLK是ADC控制器工作的时钟，也就是converter clock。从时钟框图可以看出，它是PCLK(PCLK\_PSYS)经过一次分频后得到的。所以将来初始化ADC控制器时一定有一个步骤是初始化这里的分频器。

### 11.3.3 210的10个ADC通道（注意ADC引脚和GPIO的区别）

(1)210一共支持10个ADC通道，分别叫AIN[0]~AIN[9]。理论上可以同时做10路AD转换。

(2)SoC的引脚至少分2种：digital数字引脚和analog模拟引脚。我们以前接触的GPIO都属于数字引脚，ADC Channel通道引脚属于模拟引脚。数字引脚和模拟引脚一般是不能混用的。

### 11.3.4 ADC控制器的主要寄存器

TSADCCON0

TSDATX0 TSDATY0 转出来的ADC值存在这里，我们读也是读这里

CLRINTADC0 清中断

ADCMUX 选择当前正在操作的AD通道

(1)等待触摸屏转换完毕的方法有2种：一种是检查标志位，第二种是中断。第一种方式下我们先开启一次转换，然后循环不停检查标志位直到标志位为1表示已经转换完可以去读了；第二种方式下就是设置好中断，写好中断isr来读取AD转换数据。然后开启中断后CPU就不用去管了，等AD转换完成后会生成一个中断信号给CPU，就会进入中断处理流程。第一种方法是同步的，第二种方式是异步的。

(2)AD转换都是需要反复进行的，那么转完一次一般要立即开启下一次转换，所以需要有一种机制能够在一次转完时自动开启下一次转换。这个机制就叫start by read，这个机制的工作方法是：当我们读取本次AD转换的AD值后，硬件自动开启下一次AD转换。

## 11.4 AD转换的编程实践

### 11.4.1 AD控制器初始化

### 11.4.2循环进行AD采样

### 11.4.3编译运行调试

### 11.4.5 start by read模式介绍

(1)应用方法：开启start by read模式，第一次先读一次丢掉，这次读就能开启下一次AD转换，然后以后就可以不停的读取AD值了。

### 11.4.6 DAC的应用简介

# 十二、LCD显示器

## 12.1 LCD简介

### 12.1.1什么是LCD

(1)Liquid Crystal Display，俗称液晶显示

(2)液晶是一种材料，液晶这种材料具有一种特点：可以在电信号的驱动下液晶分子进行旋转，旋转时会影响透光性，因此我们可以在整个液晶面板后面用白光照（称为背光），可以通过不同电信号让液晶分子进行选择性的透光，此时在液晶面板前面看到的就是各种各样不同的颜色，这就是LCD显示。

(3)被动发光和主动发光。有些显示器（譬如LED显示器、CRT显示器）自己本身会发光称为主动发光，有些（LCD）本身不会发光只会透光，需要背光的协助才能看起来是发光的，称为被动发光。

(4)液晶应用领域：电视机、电脑显示屏、手机显示屏、工业显示屏等····

### 12.1.2其他主流显示设备（LED、CRT、等离子、OLED）

(1)CRT：阴极摄像管显示器。

(2)等离子显示：未成为主流

(3)OLED：目前未成为主流，但是很有市场潜力

(4)LED：主要用在户外大屏幕

(5)LCD：目前是主流显示器

### 12.1.3 LCD的显示原理和特点(液晶分子透光+背光)

白光其实是由各种不同颜色的光组成的，所以白光被选择性透光之后可以产生各种不同颜色的光。

### 12.1.4 LCD的发展史和种类(TN/STN/TFT)

(1)TN最早。坏处是响应性不够好，有拖尾现象。

(2)STN是TN的升级版。有效解决拖尾现象，显示更清晰。

(3)TFT的最大特点就是超薄。

(4)TFT技术之上发展出来很多更新的技术。

(参考资料一http://blog.163.com/tao198352\_\_4232/blog/static/85020645201062285210682/)

(参考资料二：http://display.ofweek.com/2013-12/ART-8321301-8300-28763136.html)

## 12.2 LCD的接口技术

### 12.2.1从电平角度讲本质上都是TTL信号

(1)什么是TTL接口。+5V表示逻辑1，0V表示逻辑0。这种就叫TTL电平，和CMOS电平相对比。

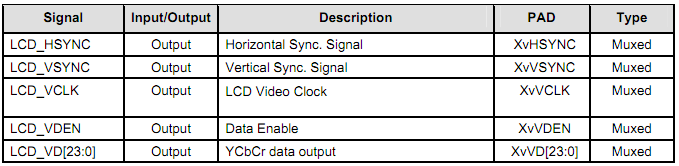
(2)SoC的LCD控制器硬件接口是TTL电平的，LCD这边硬件接口也是TTL电平的。所以他们俩本来是可以直接对接的，手机、平板、开发板都是这样直接对接的（一般用软排线连接）。

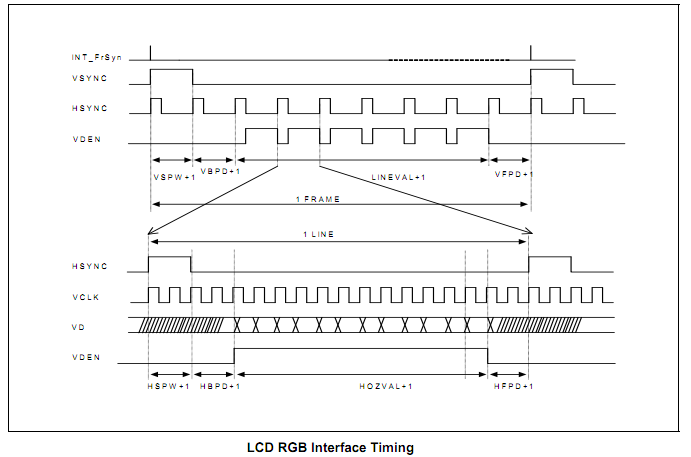
(3)TTL电平的缺陷就是不能传递太远，如果LCD屏幕和主机控制器太远（1米甚至更远）就不能直接TTL连接了，要进行转换。转换方式：主机SoC(TTL) ->VGA-> LCD屏幕(TTL)

### 12.2.2各种接口（TTL、LVDS、EDP、MIPI）在传输速率、距离、适配性方面不同

（参考资料：http://blog.csdn.net/wocao1226/article/details/23870149）

### 12.2.3 RGB接口详情





(1)VD[23:0]：24跟数据线，用来传输图像信息。可见LCD是并行接口，速率才够快。

(2)HSYNC(水平同步信号)

(3)VSYNV(垂直同步信号)：时序信号线，为了让LCD能够正常显示给的控制信号

(4)VCLK(像素时钟)：LCD工作时需要主板控制器给LCD模组一个工作时钟信号，就是VCLK。

(5)VDEN(数据有效标志位)：时序信号，和HSYNC、VSYNC结合使用。

(6)LEND(行结束标志，不是必须的)：时序信号，非必须，譬如x210接口就没有。

## 12.4 LCD如何显示图像

### 12.4.1像素(pixel)

(1)像素就是组成图像的最基本元素，或者说显示中可以被控制的最小单位，整个图像就是由很多个像素组成的。

(2)像素可以被单独控制，或控制其亮或不亮（单色屏）、或控制其亮度强弱（譬如亮50%，35%，这样叫灰度屏，以前的黑白电视机）、或控制其显示一定的颜色（这就是我们现在最常用的彩色显示屏）。

总结：像素很重要，整个显示图像是由一个个的像素组成的。我们要在显示器上显示一个图像，就是把这个图像离散化成一个一个的点，然后把各个点的颜色对应在显示器的像素上。

### 12.4.2扫描

(1)扫描是一个动作而不是一个名词，扫描就是依次将颜色数值放入屏幕中所有的像素的这个过程。

(2)扫描这个词是由最早的CRT显示器遗留下来的，到LCD显示器的年代本来已经失去意义了，但是我们还是延续着这么叫。

(3)显示器的扫描显示原理依赖于人眼的视觉暂留。只要显示器扫描频率大于人眼的发现频率，人眼看到的图像就是恒定的。如果扫描频率偏小人眼就会看到闪动。（扫描频率的概念就叫做刷新率）

### 12.4.3驱动器&控制器

(1)LCD驱动器一般和LCD显示面板集成在一起（本来是分开的，做面板的是只做面板的，譬如说三星、LG、台湾的友达、奇美都是做面板的；驱动器也由专门的IC厂商生产；集成厂商买来面板和驱动器之后集成在一起做成LCD屏幕），面板只负责里面的液晶分子旋转透光，面板需要一定的模拟电信号来控制液晶分子；LCD驱动器芯片负责给面板提供控制液晶分子的模拟电信号，驱动器的控制信号（数字信号）来自于自己的数字接口，这个接口就是LCD屏幕的外部接口（第二节中讲到的接口）。

(2)LCD控制器一般集成在SoC内部，它负责通过数字接口向远端的LCD驱动器提供控制像素显示的数字信号。LCD控制器的关键在于时序。它必须按照一定的时序和LCD驱动器通信；LCD控制器受SoC控制，SoC会从内存中拿像素数据给LCD控制器并最终传给LCD驱动器。

### 12.4.4显示内存（显存）

(1)SoC在内存中挑选一段内存（一般来说是程序员随便挑选的，但是挑选的时候必须符合一定规矩），然后通过配置将LCD控制器和这一段内存（以后统称显存）连接起来构成一个映射关系。一旦这个关系建立之后，LCD控制器就会自动从显存中读取像素数据传输给LCD驱动器。这个显示的过程就不需要CPU的参与了。

(2)显示体系建立起来后，CPU就不用再管LCD控制器、驱动器、面板这些东西了；以后CPU就只关心显存了，因为我们只要把显示的图像的像素数据丢到显存中，硬件就会自动响应（屏幕上就能自动看到显示的图像了）。

总结：LCD显示是分为2个阶段的：第一个阶段就是建立显示体系的过程，目的就是CPU初始化LCD控制器使其和显存联系起来构成映射；第二个阶段就是映射建立之后，此阶段主要任务是将要显示的图像丢到显存中去。