ARM体系与架构

硬件基础

NAND FLASH 和NOR FLASH异同？

不同点

类别	NOR	NAND
读	快 像访问SRAM一样，可以随机访问任意地址的数据；如：unsighed short *pwAddr = (unsighed short *)0x02;unisignded short wVal;wVal = *pwAddr	快,有严格的时序要求，需要通过一个函数才能读取数据，先发送读命令->发送地址->判断nandflash是否就绪->读取一页数据读命令、发送地址、判断状态、读数据都是通过操作寄存器实现的,如数据寄存器NFDATA
写	慢，写之前需要擦除，因为写只能是1->0,擦除可以使0->1	快，写之前需要擦除，因为写只能是1->0,擦除可以使0->1
擦除	非常慢（5S）	快（3ms）
XIP	代码可以直接在NOR FLASH上运行	NO
可靠性	比较高，位反转的比例小于NAND FLASH的10%	比较低，位反转比较常见，必须有校验措施
接口	与RAM接口相同，地址和数据总线分开	I/O接口
可擦除次数	10000~100000	100000~1000000
容量	小，1MB~32MB	大，16MB~512MB
主要用途	常用于保存代码和关键数据	用于保存数据
价格	高	低

注意：nandflash和norflash的0地址是不冲突的，norflash占用BANK地址，而nandflash不占用BANK地址，它的0地址是内部的。

相同点

1	**写之前都要先擦除，因为写操作只能使1->0，而擦除动作是为了把所有位都变1**
2	擦除单元都以块为单位

CPU,MPU,MCU,SOC,SOPC联系与差别？

1.CPU（Central Processing Unit）,是一台计算机的**运算核心和控制核心**。CPU由运算器、控制器和寄存器及实现它们之间联系的数据、控制及状态的总线构成。差不多所有的CPU的运作原理可分为四个阶段：提取（Fetch）、解码（Decode）、执行（Execute）和写回（Writeback）。CPU从存储器或高速缓冲存储器中取出指令，放入指令寄存器，并对指令译码，并执行指令。所谓的计算机的可编程性主要是指对CPU的编程。

2.MPU (Micro Processor Unit)，叫**微处理器**（不是微控制器），通常代表一个**功能强大的CPU**（暂且理解为**增强版的CPU**吧）,但不是为任何已有的特定计算目的而设计的芯片。这种芯片往往是个人计算机和高端工作站的核心CPU。最常见的微处理器是Motorola的68K系列和Intel的X86系列。

3.MCU(Micro Control Unit)，叫微控制器，是指随着大规模集成电路的出现及其发展，**将计算机的CPU、RAM、ROM、定时计数器和多种I/O接口集成在一片芯片上**，形成芯片级的芯片，**比如51，avr这些芯片，内部除了CPU外还有RAM,ROM,可以直接加简单的外围器件（电阻，电容）就可以运行代码了**，而MPU如x86，arm这些就不能直接放代码了，它只不过是增强版的CPU，所以得**添加RAM,ROM**。

MCU MPU 最主要的区别就睡能否直接运行代码。MCU有内部的RAM ROM，而MPU是增强版的CPU，需要添加外部RAM ROM才可以运行代码。

4.SOC（System on Chip），指的是片上系统，MCU只是**芯片级**的芯片，而**SOC是系统级的芯片**，它既MCU（51，avr）那样有内置RAM,ROM同时又像MPU（arm）那样强大的，不单单是放简单的代码，可以放系统级的代码，也就是说**可以运行操作系统**（将就认为是MCU集成化与MPU强处理力各优点二合一）。

5.SOPC（System On a Programmable Chip）可编程片上系统（FPGA就是其中一种），上面4点的硬件配置是固化的，就是说51单片机就是51单片机，不能变为avr，而avr就是avr不是51单片机，他们的硬件是一次性掩膜成型的，能改的就是软件配置，说白点就是改代码，本来是跑流水灯的，改下代码，变成数码管，而SOPC则是**硬件配置，软件配置都可以修改**，软件配置跟上面一样，没什么好说的，至于硬件，是可以自己构建的也就是说这个芯片是自己构造出来的，**这颗芯片我们叫"白片"**，什么芯片都不是，把**硬件配置信息下载进去了，他就是相应的芯片了**，可以让他变成51，也可以是avr，甚至arm，同时SOPC是在SOC基础上来的，所以他也是系统级的芯片，所以记得当把他变成arm时还得加外围ROM，RAM之类的，不然就是MPU了。

什么是交叉编译？

在一种计算机环境中运行的编译程序，能编译出在**另外一种环境**下运行的代码，我们就称这种编译器**支持交叉编译**。这个**编译过程**就叫交叉编译。简单地说，就是**在一个平台上生成另一个平台上的可执行代码**。

这里需要注意的是所谓平台，实际上包含两个概念：体系结构（Architecture）、操作系统（OperatingSystem）。同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。举例来说，我们常说的x86 Linux平台实际上是Intel x86体系结构和Linux for x86操作系统的统称；而x86 WinNT平台实际上是Intel x86体系结构和Windows NT for x86操作系统的简称。

为什么需要交叉编译？

有时是因为目的平台上不允许或不能够安装我们所需要的编译器，而我们又需要这个编译器的某些特征；有时是因为目的平台上的资源贫乏，无法运行我们所需要编译器；有时又是因为目的平台还没有建立，连操作系统都没有，根本谈不上运行什么编译器。

描述一下嵌入式基于ROM的运行方式和基于RAM的运行方式有什么区别？

基于RAM

1. 需要把硬盘和其他介质的代码先加载到ram中，加载过程中一般有重定位的操作。
2. 速度比基于ROM的快，可用RAM比基于ROM的少，因为所有的代码，数据都必须存放在RAM中。

基于ROM

1. 速度较基于RAM的慢，因为会有一个把变量，部分代码等从存储器（硬盘，flash）搬移到RAM的过程。
2. 可用RAM资源比基于RAM的多；

ARM处理器

什么是哈佛结构和冯诺依曼结构？

定义

冯诺依曼结构采用指令和数据**统一编址**，使用**同条总线**传输，CPU读取指令和数据的操作**无法重叠**。

哈佛结构采用指令和数据**独立编址**，使用**两条独立的总线**传输，CPU读取指令和数据的操作**可以重叠**。

利弊

冯诺依曼结构主要用于通用计算机领域，需要对存储器中的代码和数据频繁的进行修改，统一编址有利于**节约资源**。

哈佛结构主要用于嵌入式计算机，程序固化在硬件中，有较高的可靠性、运算速度和较大的吞吐。

什么是ARM流水线技术？

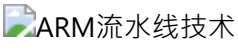
流水线技术通 过**多个功能部件并行工作**来缩短程序执行时间，提高处理器核的效率和吞吐率，从而成为微处理器设计中最为重要的技术之一。**ARM7**处理器核使用了典型**三级流水线的冯·诺伊曼结构**，**ARM9**系列则采用了基于**五级流水线的哈佛结构**。通过增加流水线级数简化了流水线各级的逻辑，进一步提高了处理器的性能。

PC代表程序计数器，流水线使用三个阶段，因此指令分为三个阶段执行：1.取指（从存储器装载一条指令）；2.译码（识别将要被执行的指令）；3.执行（处理指令并将结果写回寄存器）。而R15（PC）总是指向“正在取指”的指令，而不是指向“正在执行”的指令或正在“译码”的指令。一般来说，人们习惯性约定将“**正在执行的指令作为参考点**”，称之为当前第一条指令，因此PC总是指向第三条指令。当ARM状态时，每条指令为4字节长，所以PC始终指向该指令地址加8字节的地址，即：**PC值=当前程序执行位置+8**；

ARM指令是三级流水线，取指，译指，执行，同时执行的，现在**PC指向的是正在取指的地址（下一条指令）**，那么cpu正在译指的指令地址是PC-4（假设在ARM状态下，一个指令占4个字节），**cpu正在执行的指令地址是PC-8**，也就是说PC所指向的地址和现在所执行的指令地址相差8。

当突然发生中断的时候，保存的是PC的地址（PC-8+4 = PC-4 下一条指令的地址）

这样你就知道了，如果返回的时候返回PC，那么中间就有一个指令没有执行，所以用`SUB pc lr-irq #4`。


ARM流水线技术

ARM有几种工作模式？

1. **用户模式(USR)**

   用户模式是用户程序的工作模式，它运行在操作系统的用户态，它没有权限去操作其它硬件资源，只能执行处理自己的数据，也**不能切换到其它模式下**，要想

   访问硬件资源或切换到其它模式只能通过软中断或产生异常。

2. **系统模式(SYS)**

   系统模式是特权模式，不受用户模式的限制。用户模式和系统模式共用一套寄存器，操作系统在该模式下可以方便的访问用户模式的寄存器，而且操作系统的

   一些特权任务可以使用这个模式访问一些受控的资源。

   > 说明：用户模式与系统模式两者使用相同的寄存器，都没有SPSR（Saved Program Statement Register，已保存程序状态寄存器），但系统模式比用户模式有更高的权限，可以访问所有系统资源。

3. **一般中断模式(IRQ)**

   一般中断模式也叫普通中断模式，用于处理一般的中断请求，通常在硬件产生中断信号之后自动进入该模式，该模式为特权模式，可以自由访问系统硬件资源。

4. **快速中断模式(FIQ)** 快速中断模式是相对一般中断模式而言的，它是用来处理对时间要求比较紧急的中断请求，主要用于高速数据传输及通道处理中。（快中断有许多（R8~R14）自己的专用寄存器，发生中断时，使用自己的寄存器就避免了保存和恢复某些寄存器。如果异常中断处理程序中使用它自己的物理寄存器之外的其他寄存器，异常中断处理程序必须保存和恢复这些寄存器）

5. **管理模式（SVC）** 管理模式是**CPU上电后默认模式**，因此，在该模式下主要用来做**系统的初始化**，软中断处理也在该模式下。当用户模式下的用户程序请求使用硬件资源时，通过软件中断进入该模式。

   > 说明：系统复位或开机、软中断时进入到SVC模式下。

6. **终止模式(ABT)：** 中止模式用于支持虚拟内存或存储器保护，当用户程序访问**非法地址，没有权限读取的内存地址时**，会进入该模式，linux下编程时经常出现的segment fault通常都是在该模式下抛出返回的。

7. **未定义模式(UND)：** 未定义模式用于支持硬件协处理器的软件仿真，CPU在指令的译码阶段不能识别该指令操作时，会进入未定义模式。

> 1. 除了用户模式外，其它6种模式称为特权模式。所谓特权模式，即具有如下权利：
>
> a. MRS（把状态寄存器的内容放到通用寄存器）；
>
> b. MSR（把通用寄存器的内容放到状态寄存器中）。
>
> 由于状态寄存器中的内容不能够改变，因此，要先把内容复制到通用寄存器中，然后修改通用寄存器中的内容，再把通用寄存器中的内容复制给状态寄存器中，即可完成“修改状态寄存器”的任务。
>
> 2. 剩下的六种模式中除去系统模式外，统称为异常模式。

Arm有多少32位寄存器？

ARM处理器共有**37**个寄存器。它包含**31**个通用寄存器和**6**个状态寄存器。

Arm2440和6410有什么区别？

1. 主频不同。2440是400M的。6410是533/667M的；
2. 处理器版本不一样：2440是arm920T内核，6410是arm1176ZJF内核；
3. 6410在视频处理方面比2440要强很多。内部视频解码器，包括MPEG4等视频格式；
4. 6410支持WMV9、xvid、mpeg4、h264等格式的硬解码和编码；
5. 6410多和很多扩展接口比如：tv-out、CF卡和S-Video输出等；
6. spi、串口、sd接口也比那两个要丰富；
7. 6410采用的是DDR内存控制器；2440采用的是SDRam内存控制器；
8. 6410为双总线架构，一路用于内存总线、一路用于Flash总线；
9. 6410的启动方式更加灵活：主要包括SD、Nand Flash、Nor Flash和OneFlash等设备启动；
10. 6410的Nand Flash支持SLC和MLC两种架构，从而大大扩大存储空间；
11. 6410为双总线架构，一路用于内存总线、一路用于Flash总线；
12. 6410具备8路DMA通道，包括LCD、UART、Camera等专用DMA通道；
13. 6410还支持2D和3D的图形加速；

ARM指令集分为几类？

2类，分别为Thumb指令集，ARM指令集。ARM指令长度为32位，Thumb指令长度为16位。这种特点使得ARM既能执行16位指令，又能执行32位指令，从而增强了ARM内核的功能。

通用寄存器包括R0～R15，可以分为具体哪三类？

通用寄存器包括R0-R15，可以分为3类：

1. 未分组寄存器R0-R7

   在所有运行模式下，未分组寄存器都指向**同一个物理寄存器**，他们未被系统用作特殊的用途。因此在中断或异常处理进行**异常模式转换**时，由于不同的处理器运行模式均使用相同的物理寄存器，所以可能造成**寄存器中数据的破坏**。

2. 分组寄存器R8-R14

   对于分组寄存器，他们每次所访问的物理寄存器都**与当前的处理器运行模式相关**。

   R13常用作存放堆栈指针，用户也可以使用其他寄存器存放堆栈指针，但在Thumb指令集下，某些指令强制要求使用R13存放堆栈指针。

   R14称为链接寄存器（LR，Link Register），当执行子程序时，R14可得到R15（PC）的备份，执行完子程序后，又将R14的值复制回PC，即使用R14保存返回地址。

3. 程序计数器PC（R15）

   寄存器R15用作程序计数器（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。

Arm处理器有几种工作状态？

从编程的角度来看，ARM微处理器的工作状态一般ARM和Thumb有两种，并可在两种状态之间切换。

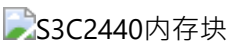
1. ARM状态：此时处理器执行32位的字对齐ARM指令，绝大部分工作在此状态。
2. Thumb状态：此时处理器执行16位的半字对齐的Thumb指令。

ARM系统中，在函数调用的时候，参数是通过哪种方式传递的？

当参数小于等于4的时候是通过**r0-r3寄存器**来进行传递的，当参数大于4的时候是通过**压栈**的方式进行传递。

为什么2440的内存起始地址是0x30000000？

S3C2440处理器有八个固定的内存块，只有两个是可以作为ROM,SRAM和SDRAM等存储器bank。具体如下图所示。

 S3C2440内存块

ARM协处理器指令包括哪3类，请描述它们的功能。

ARM协处理器指令包括以下3类：

1. 用于ARM处理器初始化ARM协处理器的数据处理操作。
2. 用于ARM处理器的寄存器和ARM协处理器的寄存器间的数据传送操作。
3. 用于在ARM协处理器的寄存器和内存单元之间传送数据。

什么是PLL（锁相环）？

简单来说，输入时钟的存在是作为“参考源”。锁相环不是为了单纯产生同频同相信号，而是一般集成进某种“频率综合电路”，产生一个不同频，但锁相的信号。

有点绕，打个比方：某参考晶振10Mhz，频率综合器A使用该参考源产生了900Mhz时钟，而频率综合器B产生了1Ghz时钟。虽然两路频率不同，但**由于使用的通一个参考源，他们俩仍然是同源信号**。相反，如果不同源，那么即便同频他们也不可能一致，因为世界上没有两个钟能做到完全一样，总有微弱的频差，导致相位飘移。在很多现实应用中有要求同源时钟的场合，所以，锁相环被广泛应用。锁相环的另外一项衍生应用是相干解调，可以自己查查相关资料。

中断与异常

中断与异常有何区别？

中断是指**外部硬件**产生的一个电信号从CPU的中断引脚进入，打断CPU的运行。

异常是指软件运行过程中发生了一些**必须作出处理**的事件，CPU自动产生一个陷入来打断CPU的运行。异常在处理的时候必须考虑与处理器的**时钟同步**，实际上异常也称为**同步中断**，在处理器执行到因编译错误而导致的

错误指令时，或者在执行期间出现特殊错误，必须靠内核处理的时候，处理器就会产生一个异常。

中断与DMA有何区别？

DMA：是一种**无须CPU**的参与，就可以让**外设与系统内存**之间进行**双向数据传输**的硬件机制，使用DMA可以使系统CPU从实际的I/O数据传输过程中摆脱出来，从而大大提高系统的吞吐率。

中断：是指CPU在执行程序的过程中，出现了某些**突发事件**时，CPU必须**暂停执行**当前的程序，转去**处理突发事件**，处理完毕后CPU又**返回**源程序被中断的位置并继续执行。

所以中断和DMA的区别就是：**DMA不需CPU参与，而中断是需要CPU参与的。**

中断能不能睡眠，为什么？下半部能不能睡眠？

1. 中断处理的时候,不应该发生进程切换。因为在中断上下文中，**唯一能打断当前中断handler的只有更高优先级的中断**，它**不会被进程打断**。如果在中断上下文中休眠，则**没有办法唤醒它**，因为所有的**wake_up_xxx都是针对某个进程而言的**，而在中断上下文中，没有进程的概念，没有一个task_struct（这点对于softirq和tasklet一样）。因此真的休眠了，比如调用了会导致阻塞的例程，内核几乎肯定会死。

2. schedule()在切换进程时，**保存当前的进程上下文**（CPU寄存器的值、进程的状态以及堆栈中的内容），以便以后恢复此进程运行。中断发生后，内核会**先保存当前被中断的进程上下文**（在调用中断处理程序后恢复）。

   但在中断处理程序里，CPU寄存器的值肯定已经变化了（最重要的程序计数器PC、堆栈SP等）。如果此时因为睡眠或阻塞操作调用了schedule()，则保存的进程上下文就不是当前的进程上下文了。所以，不可以在中断处理程序中调用schedule()。

3. 2.4内核中schedule()函数本身在进来的时候**判断是否处于中断上下文**:

```
if(unlikely(in_interrupt()))
BUG();
```

因此，强行调用schedule()的结果就是内核BUG，但看2.6.18的内核schedule()的实现却没有这句，改掉了。

4. 中断handler会使用被中断的进程内核堆栈，但不会对它有任何影响，因为handler使用完后会完全清除它使用的那部分堆栈，恢复被中断前的原貌。

5. 处于**中断上下文**时候，**内核是不可抢占的**。因此，如果休眠，则内核一定挂起。

中断的响应执行流程是什么？

中断的响应流程：cpu接受中断->保存中断上下文跳转到中断处理历程->执行中断上半部->执行中断下半部->恢复中断上下文。

当一个异常出现以后，ARM微处理器会执行哪几步操作？

1. 将下一条指令的地址存入相应连接寄存器LR，以便程序在处理异常返回时能从正确的位置重新开始执行。若异常是从**ARM状态**进入，则LR寄存器中保存的是**下一条指令的地址**（当前PC＋4或PC＋8，与异常的类型有关）；若异常是从**Thumb状态**进入，则在LR寄存器中保存**当前PC的偏移量**，这样，异常处理

程序就不需要确定异常是从何种状态进入的。例如：在软件中断异常SWI，指令 MOV PC，R14_svc总是返回到下一条指令，不管SWI是在ARM状态执行，还是在Thumb状态执行。
2. 将CPSR复制到相应的SPSR中。
3. 根据异常类型，强制设置CPSR的运行模式位。
4. 强制PC从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处。

写一个中断服务需要注意哪些？如果中断产生之后要做比较多的事情你是怎么做的？

1. 写一个中断服务程序要注意**快进快出**，在中断服务程序里面尽量**快速采集信息**，包括硬件信息，然后退出中断，要做其它事情可以使用**工作队列**或者**tasklet**方式。也就是中断上半部和下半部。

2. 中断服务程序中**不能有阻塞操作**。应为中断期间是完全占用CPU的（即不存在内核调度），中断被阻塞住，其他进程将无法操作。

3. 中断服务程序**注意返回值**，要用操作系统定义的宏做为返回值，而不是自己定义的。

4. 如果要做的事情较多，应将这些任务放在后半段(tasklet，等待队列等)处理。

为什么FIQ比IRQ要快？

1. ARM的FIQ模式提供了**更多的banked寄存器，r8到r14还有SPSR**，而IRQ模式就没有那么多，R8,R9,R10,R11,R12对应的banked的寄存器就没有，这就意味着在ARM的IRQ模式下，中断处理程序**自己要保存R8到R12这几个寄存器**，然后**退出中断处理时程序要恢复这几个寄存器**，而FIQ模式由于这几个寄存器都有banked寄存器，模式**切换时CPU自动保存这些值到banked寄存器**，退出FIQ模式时自动恢复，所以这个过程FIQ比IRQ快.不要小看这几个寄存器，ARM在编译的时候，如果你FIQ中断处理程序足够用这几个独立的寄存器来运作，它就**不会进行通用寄存器的压栈**，这样也省了一些时间。

2. FIQ比IRQ有**更高优先级**，如果FIQ和IRQ同时产生，那么FIQ先处理。

3. 在symbian系统里，当CPU处于FIQ模式处理FIQ中断的过程中，预取指令异常，未定义指令异常，软件中断全被禁止，所有的中断被屏蔽。所以FIQ就会很快执行，不会被其他异常或者中断打断，所以它又比IRQ快了。而IRQ不一样，当ARM处理IRQ模式处理IRQ中断时，如果来了一个FIQ中断请求，那正在执行的IRQ中断处理程序会被抢断，ARM切换到FIQ模式去执行这个FIQ，所以FIQ比IRQ快多了。

4. 另外FIQ的入口地址是0x1c,IRQ的入口地址是0x18。写过完整汇编系统的都比较明白这点的差别，18只能放一条指令，为了不与1C处的FIQ冲突，**这个地方只能跳转**，而FIQ不一样，1C以后没有任何中断向量表了，这样**可以直接在1C处放FIQ的中断处理程序**，由于跳转的范围限制，至少少了一条跳转指令。

中断和轮询哪个效率高？怎样决定是采用中断方式还是采用轮询方式去实现驱动？

中断是CPU处于**被动状态**下来接受设备的信号，而轮询是**CPU主动去查询**该设备是否有请求。

凡事都是两面性，所以，看效率不能简单的说那个效率高。如果是请求设备是一个**频繁请求**cpu的设备，或者有**大量数据**请求的网络设备，那么**轮询**的效率是比中断高。如果是一般设备，并且该设备请求cpu的**频率比较低**，则用**中断**效率要高一些。主要是看请求频率。

通信协议

什么是异步传输和同步传输？

异步传输：是一种典型的基于字节的输入输出，数据按每次一个字节进行传输，其传输速度低。

同步传输：需要外界的时钟信号进行通信，是把数据字节组合起来一起发送，这种组合称之为帧，其传输速度比异步传输快。

RS232和RS485通讯接口有什么区别？

1. **传输方式不同。**RS232采取不平衡传输方式，即所谓**单端通讯。**而RS485则采用平衡传输，即**差分传输方式。**
2. **传输距离不同。**RS232适合本地设备之间的通信，传输距离一般**不超过20m**。而RS485的传输距离为**几十米到上千米。**
3. **设备数量。**RS232 只允许**一对一通信**，而RS485 接口在总线上是允许连接**多达128个收发器。**
4. **连接方式。**RS232，规定**用电平表示数据**，因此线路就是**单线路的，用两根线**才能达到**全双工**的目的；而RS485，　使用**差分电平表示数据**，因此，必须用**两根线**才能达到传输数据的基本要求，要实现**全双工，**必需用**4根线。**

总结：从某种意义上，可以说，线路上存在的仅仅是电流，**RS232/RS485规定了这些电流在什么样的线路上流动和流动的样式。**

SPI协议

SPI的应用

SPI(Serial Peripheral Interface)协议是由摩托罗拉公司提出的通讯协议，即**串行外围设备接口**，是一种**高速全双工**的通信总线。SPI总线系统是一种同步串行外设接口，它可以使MCU与各种外围设备以串行方式进行通信以交换信息。SPI总线可直接与各个厂家生产的多种标准外围器件相连，包括FLASH、RAM、网络控制器、LCD显示驱动器、A/D转换器和MCU等。

接口

1. **MOSI (Master Output,　Slave Input)**

   主设备输出/从设备输入引脚。主机的数据从这条信号线输出，从机由这条信号线读入主机发送的数据，即这条线上数据的方向为主机到从机。

2. **MISO(Master Input,,　Slave Output)**

   主设备输入/从设备输出引脚。主机从这条信号线读入数据，从机的数据由这条信号线输出到主机，即在这条线上数据的方向为从机到主机。

3. **SCLK (Serial Clock)**

   时钟信号线，用于通讯数据同步。它由通讯主机产生，决定了通讯的速率，不同的设备支持的最高时钟频率不一样，如 STM32 的 SPI 时钟频率最大为fpclk/2，两个设备之间通讯时，通讯速率受限于低速设备。

4. **SS(Slave Select)**

   从设备选择信号线，常称为片选信号线，也称为 NSS、CS，以下用 NSS 表示。 当有多个 SPI 从设备与 SPI 主机相连时，设备的其它信号线 SCK、MOSI 及 MISO 同时并联到相同的 SPI 总线上，即无论有多少

个从设备，都共同只使用这 3 条总线；而每个从设备都有独立的这一条 NSS 信号线，本信号线独占主机的一个引脚，即有多少个从设备，就有多少条片选信号线。

I2C 协议中通过设备地址来寻址、选中总线上的某个设备并与其进行通讯；而 SPI 协议中没有设备地址，它使用 NSS 信号线来寻址，当主机要选择从设备时，把该从设备的 NSS 信号线设置为低电平，该从设备即被选中，即片选有效，接着主机开始与被选中的从设备进行 SPI 通讯。所以SPI 通讯以 NSS 线置低电平为开始信号，以 NSS 线被拉高作为结束信号。

协议层

SPI 通讯设备之间的常用连接方式见下图：

 SPI连接方式

SPI 通讯的通讯时序，见下图：

 SPI时序图

1. 通讯的起始和停止信号

   在图中的标号1处，NSS 信号线由高变低，是 SPI 通讯的起始信号。 NSS 是每个从机各自独占的信号线，当从机检在自己的 NSS 线检测到起始信号后，就知道自己被主机选中了，开始准备与主机通讯。在图中的标号□处， NSS 信号由低变高，是 SPI 通讯的停止信号，表示本次通讯结束，从机的选中状态被取消。

2. 数据有效性

   SPI 使用 MOSI 及 MISO 信号线来传输数据，使用 SCK 信号线进行数据同步。 MOSI及 MISO 数据线在 SCK 的每个时钟周期传输一位数据，且数据输入输出是同时进行的。数据传输时， MSB 先行（高位先行）或 LSB（低位先行）先行并没有作硬性规定，但要保证两个 SPI 通讯设备之间使用同样的协定，一般都会采用上图中的 MSB 先行（高位先行）模式。

   观察图中的2345标号处， MOSI 及 MISO 的数据在 SCK 的上升沿期间变化输出，在 SCK 的下降沿时被采样。即在 SCK 的下降沿时刻， MOSI 及 MISO 的数据有效，高电平时表示数据"1"，为低电平时表示数据"0"。在其它时刻，数据无效， MOSI 及 MISO为下一次表示数据做准备。

   SPI 每次数据传输可以 8 位或 16 位为单位，每次传输的单位数不受限制。

3. CPOL（时钟极性）/CPHA（时钟相位）及通讯模式

   上面讲述的图中的时序只是 SPI 中的其中一种通讯模式， SPI 一共有四种通讯模式，它们的主要区别是：总线空闲时 SCK 的时钟状态以及数据采样时刻。为方便说明，在此引入"时钟极性CPOL"和"时钟相位 CPHA"的概念。

   > 时钟极性 CPOL 是指 SPI 通讯设备处于空闲状态时， SCK 信号线的电平信号(即 SPI 通讯开始前、NSS 线为高电平时 SCK 的状态)。CPOL=0 时， SCK 在空闲状态时为低电平，CPOL=1 时，则相反。
   >
   > 时钟相位 CPHA 是指数据的采样的时刻，当 CPHA=0 时， MOSI 或 MISO 数据线上的信号将会在 SCK 时钟线的"奇数边沿" 被采样。当 CPHA=1 时，数据线在 SCK 的"偶数边沿" 采样。

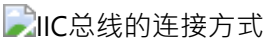
IIC协议

简介

IIC协议是由数据线SDA和时钟SCL构成的串行总线，可发送和接收数据,是一个多主机的**半双工通信方式**

每个挂接在总线上的器件都有个**唯一的地址**。位速在标准模式下可达 100kbit/s,在快速模式下可达400kbit/s，在高速模式下可待3.4Mbit/s。

I2C总线系统结构,如下所示:

 IIC总线的连接方式

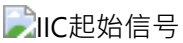
I2C时序介绍

1. 空闲状态

当总线上的**SDA和SCL两条信号线同时处于高电平**,便是**空闲状态**,如上面的硬件图所示,当我们不传输数据时,SDA和SCL被上拉电阻拉高,即进入空闲状态

2. 起始信号

当**SCL为高**期间，**SDA由高到低的跳变；**便是总线的**启动信号**,**只能由主机发起**,且在空闲状态下才能启动该信号,如下图所示：

 IIC起始信号

3. 停止信号

当**SCL为高期间，SDA由低到高的跳变；**便是总线的**停止信号,**表示数据已传输完成,如下图所示：

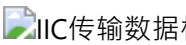
 IIC停止信号

4. 传输数据格式 当发了起始信号后,就开始传输数据,传输的数据格式如下图所示：

当**SCL为高电平时**,便会获取SDA数据值,其中**SDA数据必须是稳定的**(若SDA不稳定就会变成起始/停止信号)。

当**SCL为低电平时**,便是**SDA的电平变化状态。**

若主从机在传输数据期间,需要完成其它功能(例如一个中断),可以主动**拉低SCL**,使I2C进入**等待状态**,直到**处理结束再释放SCL,数据传输会继续**

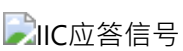
 IIC传输数据格式

5. 应答信号ACK

I2C总线上的数据都是以**8位数据(字节)进行的，当发送了8个数据后，发送方会在第9个**时钟脉冲期间**释放SDA数据**，当接收方接收该字节成功，便会输出一个ACK应答信号，当SDA为高电平,表示为非应答信号NACK，当SDA为低电平，表示为**有效应答信号ACK**

PS:当主机为接收方时,收到最后一个字节后,主机可以不发送ACK,直接发送停止信号来结束传输。

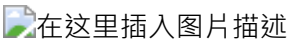
当从机为接收方时，没有发送ACK，则表示从机可能在忙其它事、或者不匹配地址信号和不支持多主机发送，主机可以发送停止信号，再次发送起始信号启动新的传输。

 IIC应答信号

6. 完整的数据传输

如下图所示, **发送起始信号**后,便发送一个**8位的设备地址**,其中**第8位是对设备的读写标志**,后面紧跟着的就是**数据**了,直到发送停止信号**终止**。

PS:当我们第一次是读操作，然后想换成写操作时，可以再次发送一个起始信号，然后发送读的设备地址，不需要停止信号便能实现不同的地址转换。

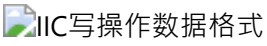
 在这里插入图片描述

IIC传输数据的格式

1.写操作

刚开始主芯片要发出一个**start信号**，然后发出一个(用来确定是往哪一个芯片写数据)，**方向**(读/写，0表示写，1表示读)。**回应**(用来确定这个设备是否存在)，然后就可以**传输数据**，传输数据之后，要有一个**回应信号**（确定数据是否接受完成)，然后再传输**下一个数据**。每传输一个数据，接受方都会有一个回应信号，数据**发送完**之后，主芯片就会发送一个**停止信号**。

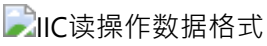
白色背景：主→从。灰色背景：从→主。

 IIC写操作数据格式

2.读操作

刚开始主芯片要发出一个**start信号**，然后发出一个**设备地址**(用来确定是从哪一个芯片读取数据)，**方向**(读/写，0表示写，1表示读)。**回应**(用来确定这个设备是否存在)，然后就可以**传输数据**，传输数据之后，要有一个回应信号（确定数据是否接受完成)，然后在传输**下一个数据**。每传输一个数据，接受方都会有一个**回应信号**，数据发送完之后，主芯片就会发送一个**停止信号**。

白色背景：主→从。灰色背景：从→主

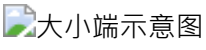
 IIC读操作数据格式

编程

嵌人式编程中，什么是大端？什么是小端？

大端模式：低位字节存在高地址上，高位字节存在低地址上。

小端模式：高位字节存在高地址上，低位字节存在低地址上。

 大小端示意图

STM32属于小端模式，简单的说，比如u32 temp=0X12345678；假设temp地址在0X2000 0010。那么在内存里面,存放就变成了：

```
地址          |            HEX      |
0X2000 0010   |  78   56   43  12   |
```

因为是16进制的，一个数为0.5字节，所以 12 代表一个字节 34 代表一个字节。

采用小端模式的CPU对操作数的存放方式是从低字节到高字节，而大端模式对操作数的存放方式是从高字节到低字节。例如，16位宽的数0x1234在小端模式CPU内存中的存放方式（假设从地址0x4000开始存放）见表1，而在大端模式CPU内存中的存放方式见表2。

表1　0x1234在小端CPU内存中的存放方式

内存地址	存放内容
0x4000	0x34
0x4001	0x12

表2　0x1234在大端CPU内存中的存放方式

内存地址	存放内容
0x4000	0x12
0x4001	0x34

32位宽的数0x12345678在小端模式CPU内存中的存放方式（假设从地址0x4000开始存放）见表3，而在大端模式CPU内存中的存放方式见表4。

表3　0x12345678在小端CPU内存中的存放方式

内存地址	存放内容
0x4000	0x78
0x4001	0x56
0x4002	0x34
0x4003	0x12

表4　0x12345678在大端CPU内存中的存放方式

内存地址	存放内容
0x4000	0x12
0x4001	0x34
0x4002	0x56
0x4003	0x78

以下程序为例：

```
#include <stdio.h>
struct mybitfields
{
    unsigned short a:4;
    unsigned short b:5;
```

```
    unsigned short c:7;
}test;
int main()
{
    int i;
    test.a = 2;
    test.b = 3;
    test.c = 0;
    i =*((short*)&test);
    printf("%d\n",i);
    return 0;
}
```

程序的输出结果为 50。

上例中， sizeof（ test）=2，上例的声明方式是把一个 short（也就是一块16位内存）分成3部分，各部分的大小分别是4位、5位、7位，赋值语句i*（ short*）&test）就是把上面的16位内存转换成 short类型进行解释。

变量a的二进制表示为0000000000000010，取其低四位是0010.变量b的二进制表示为0000000000000011，取其低五位是00011。变量c的二进制表示为0000000000000000，取其低七位是0000000。

80x86机是小端（修改分区表时要注意）模式，单片机一般为大端模式。小端一般是低位字节在高位字节的前面，也就是低位在内存地址低的一端，可以这样记（小端→低位→在前→与正常逻辑顺序相反），所以合成后得到0000000000110010，即十进制的50。

下面给出另外一个例子

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
    unsigned int uiVal_1 = 0x12345678;
    unsigned int uiVal_2 = 0;
    unsigned char aucVal[4] = {0x12,0x34,0x56,0x78};
    unsigned short usVal_1 = 0;
    unsigned short usVal_2 = 0;
    memcpy(&uiVal_2,aucVal,sizeof(uiVal_2));
    usVal_1 = (unsigned short)uiVal_1;//在这里截断，都取得的是低位
    usVal_2 = (unsigned short)uiVal_2;//在这里截断
    printf("usVal_1:%x\n",usVal_1);//在这里又转化回来
    printf("usVal_2:%x\n",usVal_2);//在这里又转化回来
    return 0;

}
```

小端模式是低地址存放低字节，高地址存放高字节，结构如下所示

```
78//低地址
56
34
12//高地址
```

在内存里面测试机是小端，地址由小到大。

```
val1:78563412
riVal2:12345678
```

结果如下：

```
5678
3412
```

如何判断计算机处理器是大端，还是小端？

```
#include <stdio.h>
int checkCPU()
{
    {
        union w
        {
            int a;
            char b;
        }c;
        c.a =1;
        return(c.b == 1);
    }
}
int main()
{
    if(checkCPU())
        printf("小端\n");
    else
        printf("大端\n");
    return 0;
}
```

编者的处理器为ntel处理器，因为 Intel处理器一般都是小端模式，所以此时程序的输出结果为：小端

上述代码中，如果处理器是大端，则返回0；如果处理器是小端，则返回1.联合体 union的存放顺序是所有成员都从低地址开始存放，如果能够通过改代码知道CPU对内存是采用小端模式读写，还是采用大端模式读写，一定会令面试官刮目相看。

还可以通过指针地址来判断，由于在32位计算机系统中， short占两个字节，char占1个字节，所以可以采用如下做法实现该判断。

```
#include <stdio.h>
int checkCPU()
{
    unsigned short usData = 0x1122;
    unsigned char*pucData = (unsigned char*)&usData;
    return (*pucData == 0x22);
}
int main()
{
    if(checkCPU())
        printf("小端\n");
    else
        printf("大端\n");
    return 0;
}
```

程序输出的结果为：小端

如何进行大小端的转换？

```
int swapInt32(int intValue){

    int temp = 0;

    temp = ((intValue & 0x000000FF) <<24)|

           ((intValue & 0x0000FF00) <<8) |

           ((intValue & 0x00FF0000) >>8) |

           ((intValue & 0xFF000000) >>24);
    return temp;
}
/*short型：*/
unsigned short swapShort16(unsigned short shortValue){



return ((shortValue & 0x00FF ) <<8) | ((shortValue & 0xFF00)>>8);

}
/*float型:*/
float swapFloat32(float floatValue){

         typedef union SWAP_UNION{

         float unionFloat;
```

```
        int   unionInt;

        }SWAP_UNION;

        SWAP_UNION swapUnion;
        swapUnion.unionFloat = floatValue;
        swapUnion.unionInt = swapInt32(  swapUnion.unionInt);

       return     swapUnion.unionFloat;
}
/*double型换一种写法，用一下指针，不然移位移死了……*/
void swapDouble64(unsigned char *pIn, unsigned char *pOut){

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

pOut[7-i] = pIn[i];

}

int main()
{
    int x = 0x12345678;
    int y = swapInt32(x);
    printf("%x\r\n",y);
    return 0;
}
```

如何对绝对地址0x100000赋值？

```
*(unsigned int*)0x100000 = 1234;
```

那么要是想让程序跳转到绝对地址是0x100000去执行，应该怎么做？

```
*((void (*)( ))0x100000 ) ( );
```

首先要将0x100000强制转换成函数指针,即：

```
(void (*)())0x100000
```

然后再调用它：

```
*((void (*)())0x100000)();·
```

用typedef可以看得更直观些：

```
typedef void(*)() voidFuncPtr;
*((voidFuncPtr)0x100000)();
```

结语

资料中，难免会有一些错误，有任何问题，都可以在github向我提交issues。文中的勘误，我都会更新在github和公众号中。