目录

[1.请描述冯·诺依曼结构与哈佛结构 2](#_Toc169982076)

[2.请描述ARM处理器对异常中断 2](#_Toc169982077)

[中断 2](#_Toc169982078)

[中断源 2](#_Toc169982079)

[中断/异常优先级 2](#_Toc169982080)

[异常和中断的区别 3](#_Toc169982081)

[中断的状态 3](#_Toc169982082)

[ARM-7种异常类型 3](#_Toc169982083)

[ARM-异常中断的响应流程(四大步三小步) 4](#_Toc169982084)

[3.请描述Thumb指令集与ARM指令集 6](#_Toc169982085)

[4.安全防拆一般有哪些技术要求： 7](#_Toc169982086)

[5.请描述12C总线的仲裁方式。 8](#_Toc169982087)

[6.嵌入式软件安全的防错误注入 10](#_Toc169982088)

[7.请描述嵌入式系统主控芯片的典 11](#_Toc169982089)

[8.Cortex M.3存储器管理故障的三 11](#_Toc169982090)

[9.综合比较OTP、ROM、NAND 13](#_Toc169982091)

[10.与SAR ADC相比，积分型AD 14](#_Toc169982092)

[11.Cortex M3中断处理中的咬尾 15](#_Toc169982093)

[12.描述Cortex M3合法的操作模式 17](#_Toc169982094)

[13.解释TEE的概念与原理，它有 19](#_Toc169982095)

[14.嵌入式软件的安全审计应具备 20](#_Toc169982096)

[15.请描述UICC和外部设备的安全 20](#_Toc169982097)

[16.Global Platform所定义的嵌入 21](#_Toc169982098)

[17.引起Cortex M3的用法故障的 21](#_Toc169982099)

[18.请描述CAN总线的仲裁方式。 23](#_Toc169982100)

[19.Global Platform的安全域应承 24](#_Toc169982101)

[20.解释高速串行通信中采用8B/ 25](#_Toc169982102)

[21.与推挽输出相比，开漏输出有 25](#_Toc169982103)

[22.阐述嵌入式系统中使用DEP和 27](#_Toc169982104)

[23.中断、异常及陷阱这三个概念 28](#_Toc169982105)

[24.在有cache的单片机系统中写 28](#_Toc169982106)

[25.将以下分支程序代码使用805 29](#_Toc169982107)

[26.将以下分支程序代码使用ARM 29](#_Toc169982108)

[27.嵌入式系统中浮空输入电路有 29](#_Toc169982109)

[28.MCU芯片为什么通常采用CM 29](#_Toc169982110)

[29.编程实现安全比较 30](#_Toc169982111)

[30.编程实现安全拷贝 32](#_Toc169982112)

# 1.请描述冯·诺依曼结构与哈佛结构

冯·诺伊曼(普林斯顿)体系结构

程序存储器和数据存储器公用一个存储空间，统一编址采用统一的地址及数据总线，指令和数据的宽度相同使用灵活(例如代码远程更新OTA)

哈佛体系结构

程序存储器和数据存储器是独立编址的两个存储空间

分离的程序总线和数据总线可允许一个机器周期内同时获取指令码(程序存储器)和操作数(数据存储器)，从而提高执行速度和数据的吞吐率，可靠性高(大部分程序存储器是只读存储器)

# 2.请描述ARM处理器对异常中断

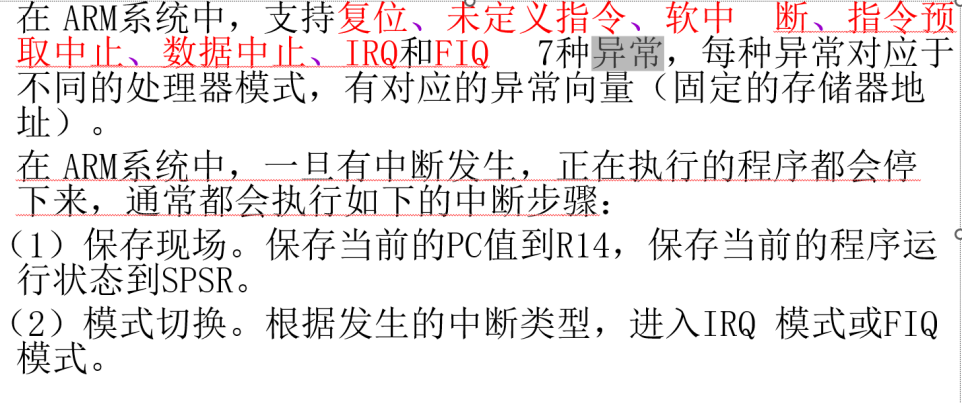
ARM 异常中断处理过程为：

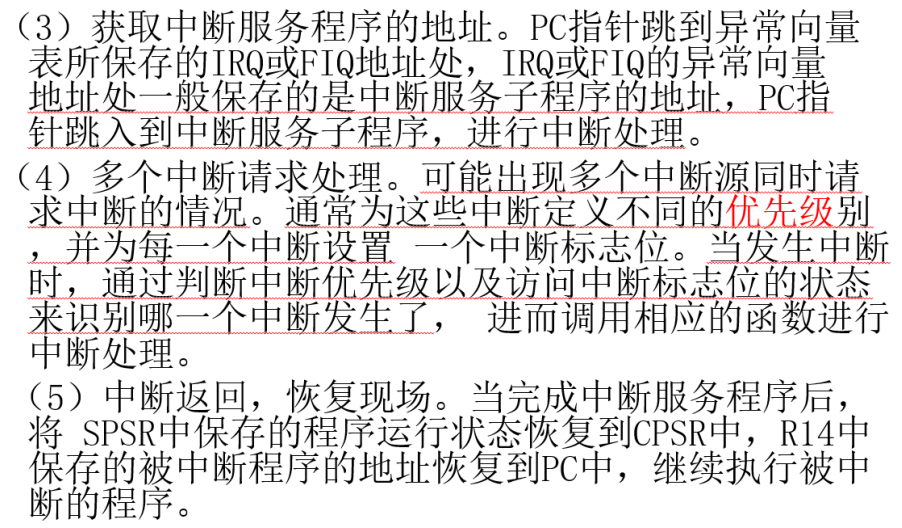
1）保存处理器当前状态、中断屏蔽位以及各条件标志位；

2）设置当前程序状态寄存器 CPSR 中的相应位；

3）将寄存器 lr\_mode 设置成返回地址

4）将程序计数器值 PC，设置成该异常中断的中断向量地址，跳转到相应异常中断处继续执行。





## 中断

1） 硬中断

在处理器中，中断是一个过程。即CPU在正常执行程序的过程中，遇到外部或内部的紧急事件需要处理，暂时中断(中止)当前程序的执行，而转去完成事件服务程序，待事件完毕后，再返回到暂停处(断点)继续执行原来的程序。事件服务程序又称中断处理程序或中断服务程序。严格意义上来说，上面的描述是针对硬件中断而言的。

2） 软中断

用软件方法引起的中断，即事先在程序中安排特殊的指令，CPU执行到该类指令的时候，会跳转去执行相应的一段预先安排好的程序，待程序执行完，再返回原来程序处。这种通过软件方法实现的中断叫软中断。

3） 软、硬中断和信号的区别

硬中断的出现过程是不可预测的，是随机的；而软中断是预先安排好的指令中断方式。

“硬中断是外部设备对CPU的中断”，“软中断通常是硬中断服务程序对内核的中断”，“信号则是由内核（或其他进程）对某个进程的中断”。

## 中断源

把引起中断的原因，或者能够发出中断请求信号的来源统称为中断源。

## 中断/异常优先级

ARM处理器五种模式对应的7种异常源优先级顺序(从高到低)：

1 复位异常(reset)

2 数据异常(Data Abord)

3 快速中断异常(FIQ)

4 外部中断异常(IRQ)

5 预取异常(prefetch Abord)

6 软中断异常(SWI)

7 未定义指令异常(Undefined)

## 异常和中断的区别

异常主要是从处理器被动接受异常的角度来考虑；中断带有主动请求处理器处理中断的意味。

共同点：都是打断当前程序执行，进入特定的程序处理事件的一种机制。

## 中断的状态

挂起态：中断已发生，但是中断没有被处理执行

激活态：中断发生，正在执行对应的中断处理函数，但是还没有执行结束

未激活态：中断没有发生。

## ARM-7种异常类型

1） 复位异常(reset)

两种情况引起异常：系统上电/系统复位

复位异常中断处理程序完成的功能：

设置异常向量表

初始化栈和寄存器

初始化MMU，Cache

初始化关键的I/O设备

使能中断

切换到SVC模式

初始化C程序中的变量，跳转到相应程序的执行

2）数据异常(Data Abord)

存储器访问指令load/store访问外部数据时，出现地址不存在或不允许访问等时产生：取数据中止异常

3）快速中断异常(FIQ)

4）外部中断异常(IRQ)

5）预取异常(prefetch Abord)

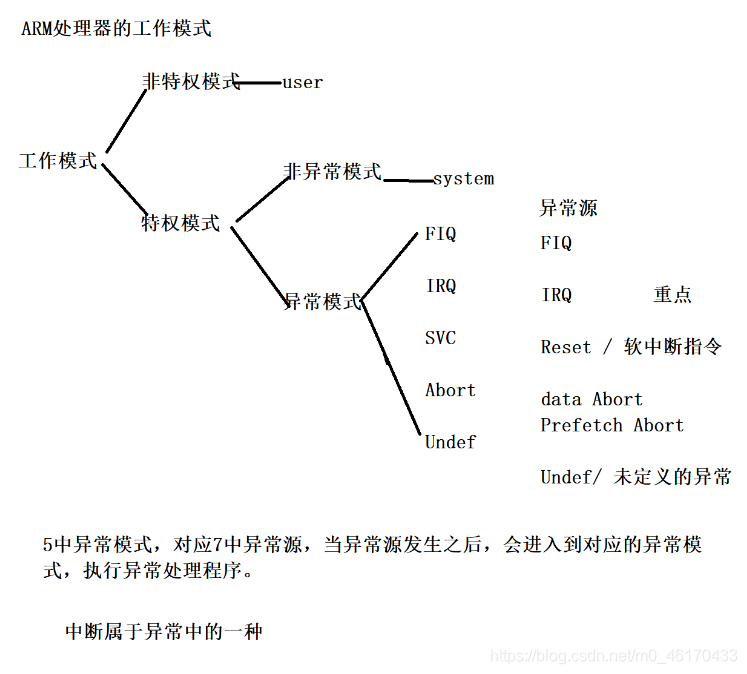
取一条被标记无效的指令时：取指令中止异常

6）软中断异常(SWI)

软中断发生，系统进入SVC模式，跳转到处理程序

7）未定义指令异常(Undefined)

：未定义指令的异常，指令取出后通过译码器后未定义指令导致译码出错



## ARM-异常中断的响应流程(四大步三小步)

注意：

User和System模式是不可通过异常进入的两种模式，即要进入这两种模式，必须要编程设置CPSR的模式状态。

ARM处理器中cortex-A不支持中断嵌套，因为ARM处理器的处理中断处理速度较快；

cortex-M系列支持中断嵌套，eg：stm32。

ARM异常处理：处理器对特定的异常事件进行的处理流程（CPU指导硬件自动完成：四大步三小步）。

一、保存现场(四大步)：

1）保存CPSR到SPSR\_mode

2）适当设置 CPSR 对应功能位（三小步）：

1、切换处理器进入ARM状态：T[5]

2、根据需要，禁止中断位：F[6] / I[7]

3、根据异常切换到对应的异常模式：M[4:0]

3）保存返回地址：把当前 PC 保存到 lr\_mode

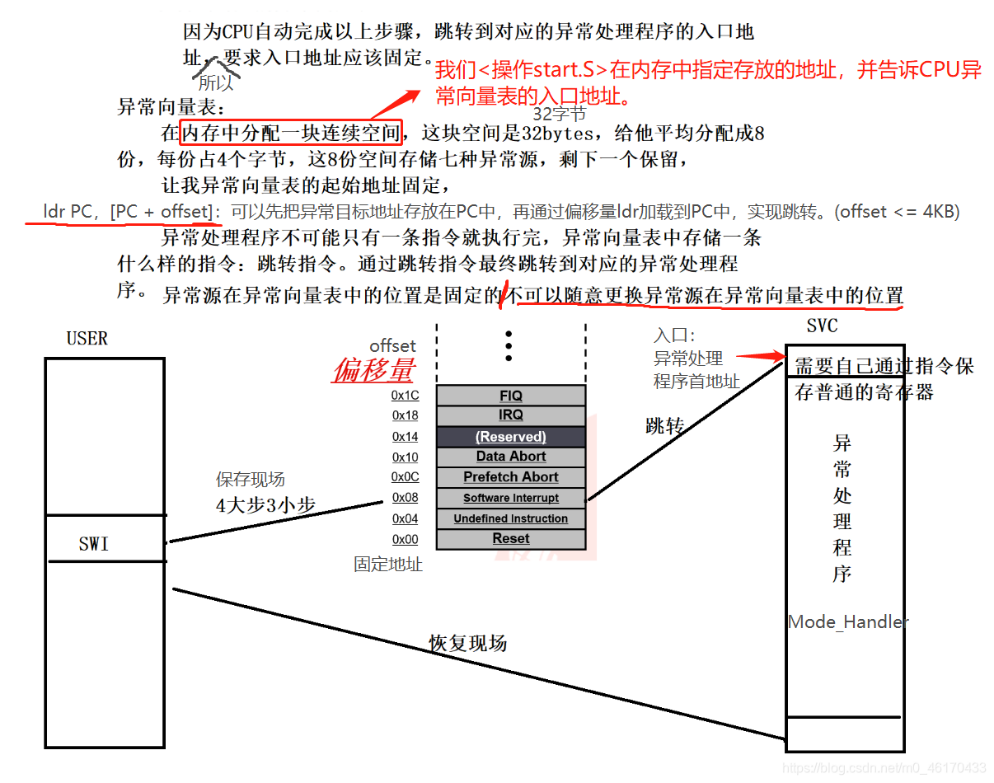
4）设置PC = 存放跳转到对应的异常向量表的固定首地址。

二、恢复现场（必须手动恢复）：

1）恢复 SPSR\_mode 到 CPSR

2）恢复 lr\_mode 到 PC

3）恢复普通寄存器：R0-R7，操作sp将栈中将数据恢复。



ARM处理器对异常中断的处理遵循一套规定的异常处理机制。当发生异常（例如中断或故障）时，ARM处理器会按照特定的优先级和规则来处理。

ARM处理器定义了一组异常级别，包括：

1. 用户级别（User Level）：最低特权级别，用于执行普通应用程序。
2. 特权级别（Privileged Level）：具有更高特权的级别，用于执行操作系统内核代码和特权任务。
3. 异常级别（Exception Level）：用于处理异常事件的特殊级别，例如中断和故障。

当发生异常时，ARM处理器会根据异常的类型和优先级来确定处理方式。以下是一般的异常处理过程：

1. 保存寄存器状态：ARM处理器会自动保存当前执行的指令位置和相关寄存器的状态，以便后续恢复执行时使用。
2. 切换到异常模式：ARM处理器会切换到特定的异常处理模式，例如中断模式或故障模式。在这些模式下，处理器会有更高的特权级别，可以执行特权操作。
3. 执行异常处理程序：ARM处理器会跳转到预定义的异常处理程序，该程序负责处理特定类型的异常。这些异常处理程序通常由操作系统或软件提供。
4. 处理异常：在异常处理程序中，可以进行一系列操作，包括保存其他寄存器状态、处理异常事件、执行必要的操作以响应异常，并可能触发相应的异常处理机制。
5. 恢复寄存器状态并返回：在异常处理程序完成后，ARM处理器将恢复先前保存的寄存器状态，并返回到原始的执行上下文中，继续执行被中断的指令。

ARM处理器提供了灵活的异常处理机制，可以根据具体的应用和需求进行配置和扩展。这使得ARM处理器在嵌入式系统和移动设备等领域广泛应用，并能够有效地处理各种异常情况。

# 3.请描述Thumb指令集与ARM指令集

课本p26

ARM指令集

编代码全部是 32bits 的，每条指令能承载更多的信息，因此使用最少的指令完成功能， 所以在相同频率下运行速度也是最快的， 但也因为每条指令是32bits 的而占用了最多的程序空间

Thumb指令集

编代码全部是 16bits 的，每条指令所能承载的信息少，因此它需要使用更多的指令才能完成功能， 因此运行速度慢， 但它也占用了最少的程序空间

Thumb-2指令集

在前面两者之间取了一个平衡， 兼有二者的优势， 当一个 操作可以使用一条 32bits指令完成时就使用 32bits 的指令， 加快运行速度， 而当一次操作只需要一条16bits 指令完成时就使用16bits 的指令，节约存储空间

消灭了状态切换的额外开销，节省了执行时间和指令空间。不再需要把源代码文件分成按 ARM 编译的和按Thumb 编译的，软件开发的管理大大减负。无需再反复地求证和测试：究竟该在何时何地切换到何种状态下，我的程序才最有效率。开发软件容易多了。

Thumb指令集和ARM指令集都是由ARM架构定义的指令集体系。

**ARM指令集：**

ARM指令集是ARM架构最初定义的指令集。它使用32位的指令长度，提供了丰富的指令集和功能，包括整数操作、浮点运算、数据传输、逻辑运算、分支跳转等。ARM指令集的指令长度固定，每条指令都占用32位存储空间。ARM指令集的特点是指令丰富、灵活，适用于复杂的应用和高性能的计算需求。

**Thumb指令集：**

Thumb指令集是ARM架构中的一种压缩指令集。它采用16位的指令长度，相比于ARM指令集的32位指令，Thumb指令集可以节省存储空间，并提供了更高的代码密度。Thumb指令集主要用于嵌入式系统和资源受限设备，如移动设备和传感器节点等。Thumb指令集包含了一部分ARM指令集的常用指令，并且提供了一些特殊的压缩指令，以实现更高的代码密度和节能效果。

在ARM架构的处理器中，可以同时支持ARM指令集和Thumb指令集。这意味着处理器可以根据需要在ARM指令集和Thumb指令集之间进行切换。这种切换可以通过特殊的指令或者程序状态寄存器（PSR）中的标志位来实现。

**对比**

在Thumb指令集中，指令的操作码和寄存器操作的位数都是16位，并且每条指令只能访问16位的寄存器。因此，Thumb指令集相对于ARM指令集来说，具有更小的寻址范围和寄存器数量，以及一些指令和功能的限制。但由于Thumb指令集的指令长度更短，可以在同样的存储空间内存放更多的指令，从而提高了代码密度和缓存命中率，有利于嵌入式系统中的资源优化和功耗控制。

需要注意的是，ARM指令集和Thumb指令集是兼容的，可以在同一程序中同时使用这两种指令集。这种混合使用可以结合ARM指令集的灵活性和Thumb指令集的代码密度，以实现更好的性能和资源优化。

# 4.安全防拆一般有哪些技术要求：

**防拆的定义**

终端输入口连接外部监测电路，一旦监测电路的电平发生变化，中断响应，程序执行相应的操作。

监测电路一般采用机械开关或防拆触点结合逻辑门电路实现，可在结构上安装导电颗粒，利用结构锁定后的压力，一旦终端被拆开，防拆触点断开，或门的输入端被上拉电阻拉为高电平，或门输出高电平，产生中断信号给CPU。

使用或门，使得防拆触点只要有一个出现断开，即可产生中断信号；

防拆触点的个数与其在PCB上的布局由终端安全要求决定

上述方式在终端不工作时不能防拆。此时，一般采用低功耗的、内部带有SRAM存储和快速中断响应口的器件进行设计；

快速中断响应接口线仍连接上述的监测电路进行防拆监测，监测电路采用纽扣电池对器件和监测电路进行供电，再在器件内部的SRAM中写入预定的标志位；

当终端受到拆卸攻击，监测电路的电平可以快速的产生中断,再擦除SRAM中的标志位；

终端重新上电、识别到错误的标志位，将删除重要的数据和程序。

由于终端此时由纽扣电池供电，运行在低功耗模式下，运行速度相对较慢，使得标志位的擦除需要一定时间，存在风险。

**要求**

国内的银联卡受理终端PIN输入设备安全认证（PIN PAD认证）和国际的支付终端认证（PCI-DSS安全认证Payment Card Industry Data Security Standard）对支付终端的安全防拆机制提出了全面、细致的要求；

对于所有涉及金融交易数据软件上要求层层加密，复杂和重要的密钥不由终端存储，简单的密钥存储在终端的高安全器件中，同时所有安全相关的器件、电气走线都应处于硬件上的防拆保护网所围成的结构空间（又称安全区）内部。

• 专用监测RAM。

• 低功耗，在生命期内持续监测。

• 拆除自毁。

防拆保护网的在PCB板上形成网状走线，不同网络的网状走线之间可相 互交叉保护；这样的网状走线必须出现在所有安全器件的空间周围。

• 假设安全器件位于A板上的BOTTOM层，则其上方必须还有一个带有保护网状走线的B板盖住A板上的安全器件，而在A、B板之间的缝隙中还必须有若干个小的PCB板围成的围栏板， 使得安全器件完全处于一个密闭的、带有保护网状走线的结构空间内，此时A板的TOP层（通常设计为按键区）也必须有网状走线的保护，一般采用FPC板（柔性印制线路板）；这样的设计，使所有带有敏感数据（密钥、防拆信息、交易程序等）的器件在物理上受到了层层保护。

安全防拆通常需要满足以下技术要求：

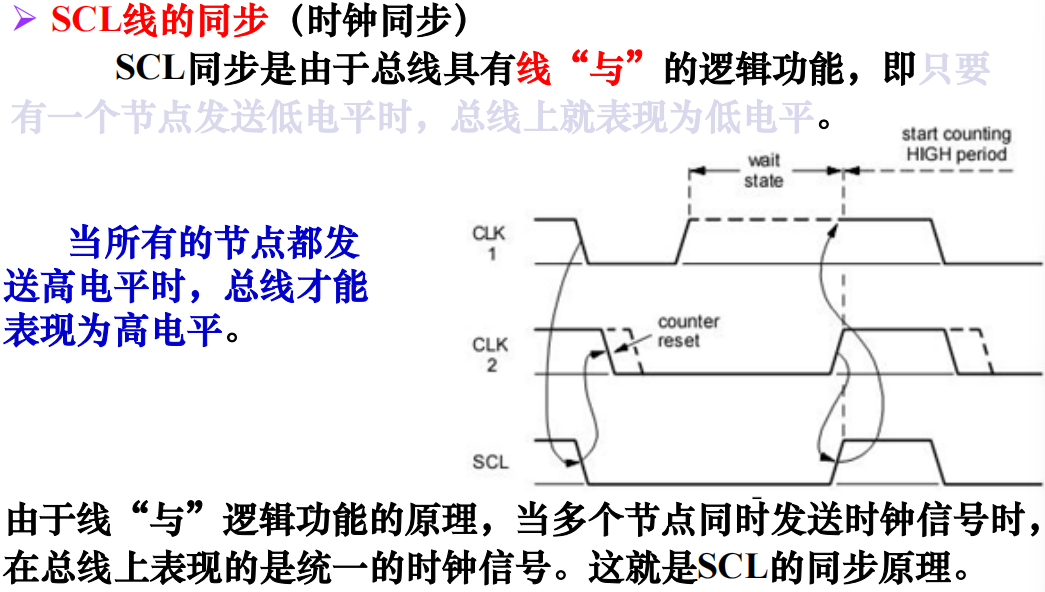
1. 物理防护：确保设备的物理安全，防止非授权人员进行拆解或物理攻击。这可以包括使用防拆标签、密封胶、封装材料等，以及设计坚固的外壳和机械结构，增加拆解的难度。
2. 防拆检测：设备应该具备防拆检测机制，能够监测和识别拆解行为。这可以包括使用振动传感器、加速度计、光电传感器等技术，用于检测设备外壳的破坏或移动。
3. 密封保护：对设备内部重要组件、电路和存储器进行密封保护，防止非授权访问和修改。这可以包括使用加密技术、密钥保护、硬件安全模块（HSM）等措施，确保数据的机密性和完整性。
4. 防撬设计：在设备的外壳和连接部件中采用防撬设计，使得拆解过程更加困难。例如使用特殊螺丝、粘合剂、防撬结构等。
5. 软件安全：通过软件层面的安全机制来保护设备免受拆解攻击。这可以包括使用加密算法、数字签名、访问控制、安全协议等，确保设备的软件和固件不受篡改。
6. 远程监控和反馈：设备应具备远程监控和反馈功能，能够向授权的监控系统发送拆解警报或状态信息。这可以通过网络连接、传感器和通信模块实现，及时通知相关人员并采取行动。
7. 防拆报警：设备应该能够触发警报或报警系统，以吸引注意并阻止拆解行为。这可以包括使用声音警报、闪光灯、无线通信等技术。

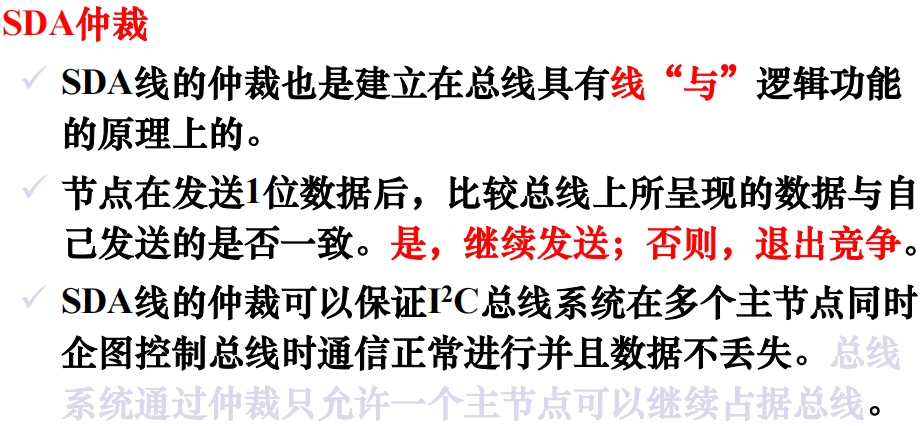
总体来说，安全防拆需要综合考虑物理、电子和软件等多个方面的技术要求，以保护设备的完整性、机密性和可用性。具体的要求和措施将根据不同的应用场景和安全级别而有所差异。

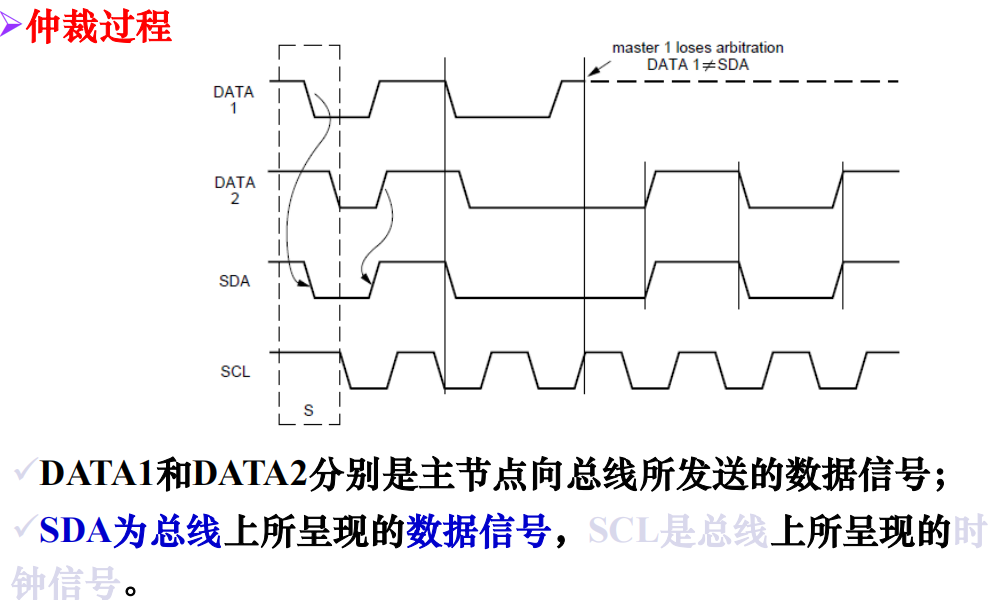
# 5.请描述12C总线的仲裁方式。

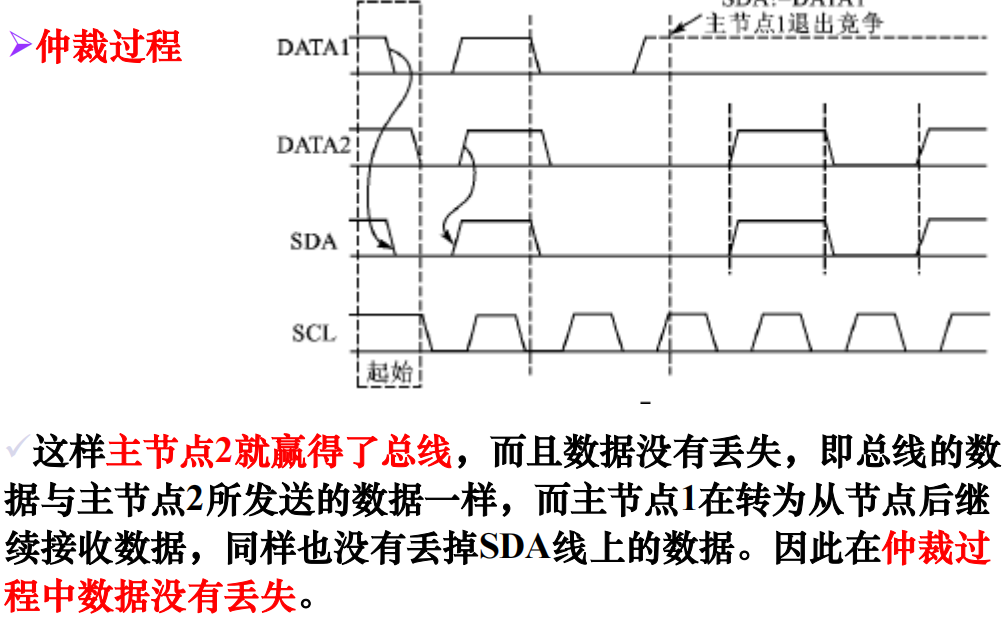
如果有两个或两个以上的节点都向总线上发送启动信号并开始传送数据，这样就形成了冲突。要解决这种冲突，就要进行仲裁的判决，这就是I2C总线上的仲裁。

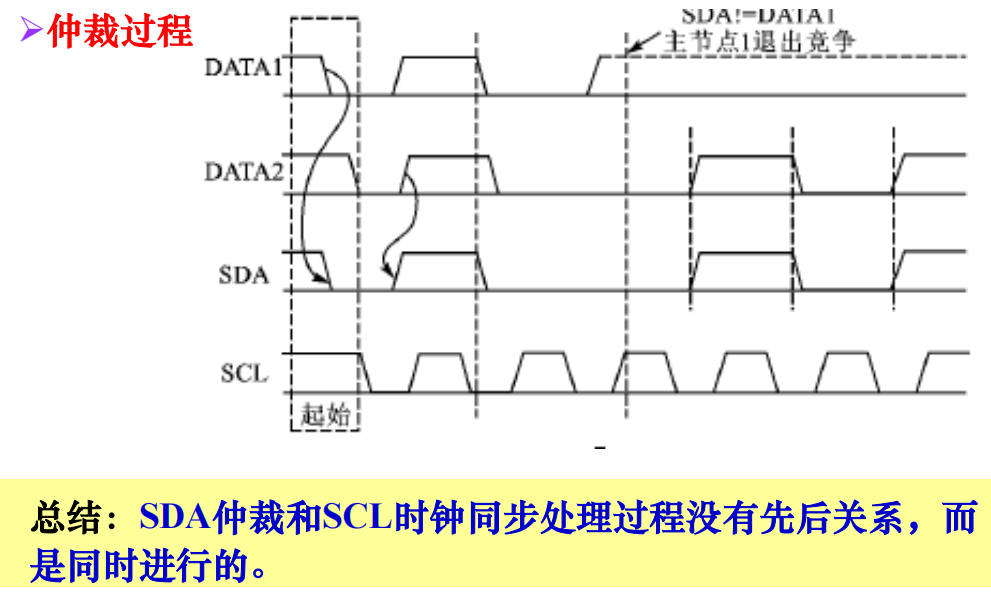
I2C总线上的仲裁分两部分：SCL线的同步和SDA线的仲裁。











I2C（Inter-Integrated Circuit）总线是一种常用的串行通信协议，它使用两根信号线（SDA和SCL）进行数据传输。在I2C总线中，仲裁是指多个设备之间竞争总线控制权的过程，以确定哪个设备可以在总线上发送数据。

以下是I2C总线的仲裁方式的基本描述：

1. 总线主设备发送起始条件：当总线上无设备正在发送数据时，总线主设备（通常是主控芯片）可以发送起始条件。起始条件是将SDA线从高电平拉低，然后将SCL线从高电平拉低的过程。
2. 同时发送地址：在起始条件之后，所有连接到I2C总线上的设备都可以同时发送自己的地址。地址是一组位，用于识别设备。
3. 竞争和仲裁：当多个设备同时发送地址时，会发生竞争。每个设备在发送地址期间都会监测SDA线的电平，如果发现自己发送的地址与总线上的实际地址不匹配，则会放弃发送数据的机会。
4. 仲裁冲突检测：设备在发送地址的同时，会不断检测SDA线上的电平状态。如果设备发送的数据位与SDA线上的电平不一致，则说明发生了仲裁冲突。在发生冲突后，设备会立即停止发送数据，并监听总线上的数据传输。
5. 仲裁冲突解决：在检测到仲裁冲突后，所有参与冲突的设备会停止发送数据，并释放SDA和SCL线的控制权。然后，这些设备会等待一段时间，再次尝试发送数据。
6. 重试和优先级：设备在发生仲裁冲突后会等待一段随机时间再次尝试发送数据。为了确保公平性，一些I2C设备实现了优先级机制，使得某些设备比其他设备更具优先级，可以更早地重新发送数据。

通过这种仲裁方式，I2C总线能够有效地处理多个设备同时发送数据的情况，并避免冲突。仲裁机制保证了每个设备都有机会发送数据，并确保总线上的通信顺利进行。

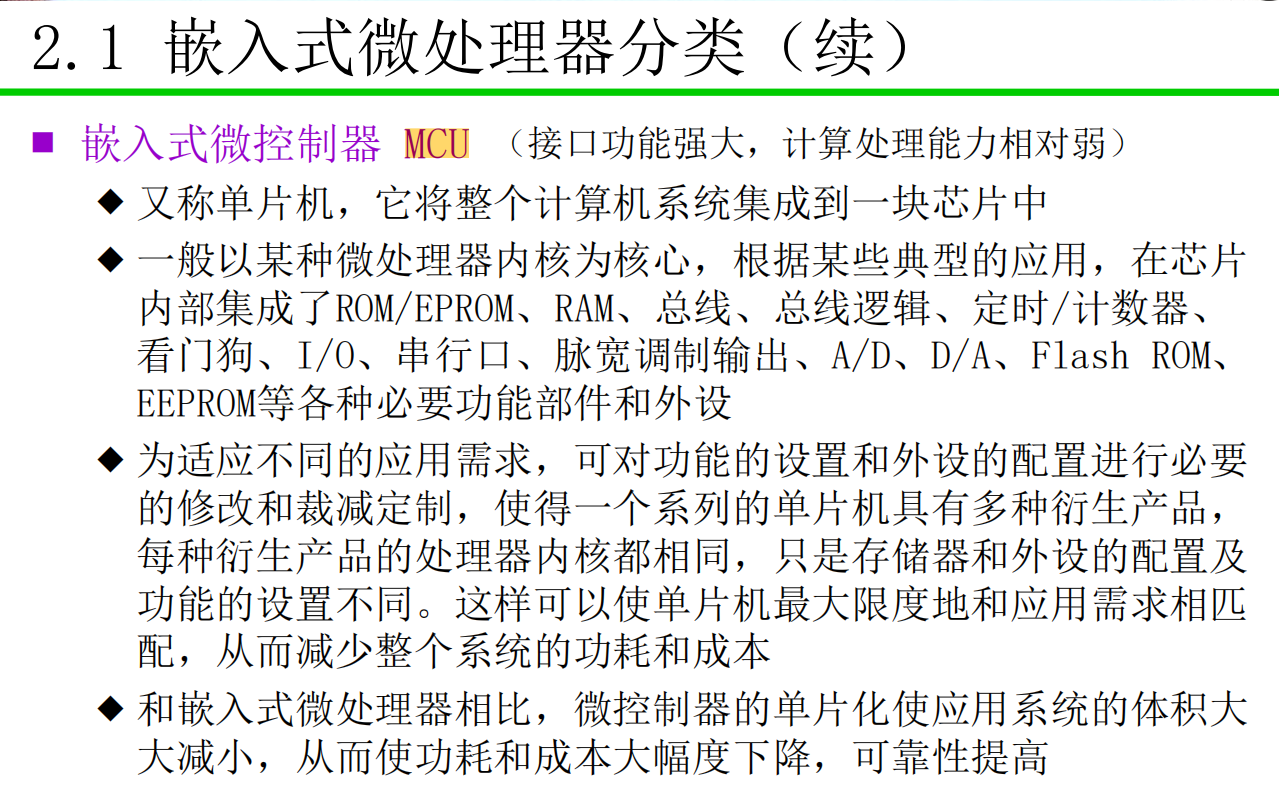
## 6.嵌入式软件安全的防错误注入

嵌入式软件安全的防错误注入是指通过采取措施防止恶意用户或攻击者在嵌入式软件中注入错误、恶意代码或漏洞。下面是一些常见的防错误注入技术：

1. 输入验证和过滤：对于所有的输入数据，包括用户输入和外部输入，进行输入验证和过滤。这包括检查输入的边界、长度和格式，过滤特殊字符、非法命令和恶意代码等。通过有效的输入验证，可以防止恶意注入、缓冲区溢出和其他常见的注入攻击。
2. 安全编码实践：采用安全的编码实践来编写嵌入式软件，避免常见的安全漏洞和注入点。这包括避免使用不安全的函数和API、正确处理输入和输出、适当地使用加密算法和哈希函数，以及避免硬编码敏感信息等。
3. 内存保护机制：使用内存保护机制，如堆栈保护、缓冲区溢出检测、地址空间布局随机化（ASLR）等。这些机制可以防止恶意注入攻击利用缓冲区溢出漏洞或覆盖关键数据。
4. 输入参数验证：对于接收到的输入参数进行验证，确保其符合预期的范围和规则。这包括对输入参数进行类型检查、范围检查和边界检查，以避免错误数据的注入。
5. 安全更新和固件验证：定期更新嵌入式软件和固件，修复已知的安全漏洞和弱点。同时，确保固件的完整性和真实性，使用数字签名或哈希校验等机制验证固件的合法性，防止未经授权的固件注入。
6. 访问控制和权限管理：使用访问控制和权限管理机制，限制对嵌入式系统的访问和操作。只允许授权用户或合法设备进行操作，并根据需要进行身份验证和授权。
7. 日志记录和审计：记录嵌入式软件的运行日志，并进行定期审计，发现异常行为和潜在的注入攻击。通过监控和分析日志，可以及时发现并应对安全事件。

以上是一些常见的防错误注入技术，但需要根据具体的嵌入式系统和应用场景选择适合的安全措施。同时，定期进行安全评估和漏洞扫描，及时修复和更新系统中的安全问题，也是确保嵌入式软件安全的重要步骤。

## 7.请描述嵌入式系统主控芯片的典





常嵌入式系统主控芯片的典型特征和功能包括：

1. 处理器核心：主控芯片通常搭载一或多个处理器核心，如ARM Cortex系列、MIPS、x86等，用于执行嵌入式软件和处理系统任务。
2. 存储器：主控芯片集成了存储器单元，包括闪存（Flash）用于存储程序代码和数据，以及静态随机存取存储器（SRAM）用于临时数据存储。
3. 外设接口：主控芯片提供多种外设接口，用于连接和控制外部设备，如通用串行总线（USB）、以太网、SPI（Serial Peripheral Interface）、I2C等。
4. 定时器和计数器：主控芯片通常包含内置的定时器和计数器模块，用于精确计时、周期性任务调度和事件触发。
5. 中断控制器：主控芯片具备中断控制器，用于管理和响应外部设备的中断请求，实现实时响应和任务切换。
6. PWM控制器：主控芯片可能集成脉冲宽度调制（PWM）控制器，用于生成和控制模拟信号，如用于电机控制、LED调光等应用。
7. ADC和DAC接口：某些主控芯片提供模拟数字转换器（ADC）和数字模拟转换器（DAC）接口，用于采集和输出模拟信号。
8. 安全功能：为了增强嵌入式系统的安全性，主控芯片可能包含安全功能，如硬件加密引擎、随机数生成器、安全存储等，用于保护敏感数据和执行加密算法。
9. 低功耗模式：嵌入式系统通常要求低功耗设计，主控芯片可能支持多种低功耗模式，如睡眠模式、待机模式等，以延长电池寿命或节能。
10. 调试和调试接口：主控芯片提供调试接口，如JTAG（Joint Test Action Group）或SWD（Serial Wire Debug），用于软件开发、调试和故障排查。
11. 温度和电压监测：主控芯片可能集成温度和电压监测电路，用于检测芯片的工作状态和环境条件。

这些是主控芯片的一些典型特征和功能，具体的主控芯片选择将根据嵌入式系统的需求、性能要求和应用场景进行。不同的芯片厂商和系列产品会提供各种不同的功能和定制选项，以满足不同的嵌入式系统设计需求。

# 8.Cortex M.3存储器管理故障的三

存储器管理faults 多与MPU有关，其诱因常常是某次访问触犯了MPU设置的保护策略。另外，某些非法访问，例如，在不可执行的存储器区域试图取指，也会触发一MemManage fault，而且即使没有MPU 也会触发。MemManage faults 的常见诱因如下所示：

访问了 MPU 设置区域覆盖范围之外的地址

往只读 region 写数据

用户级下访问了只允许在特权级下访问的地址

在MemManage fault 发生后，如果其服务例程是使能的，则执行服务例程。

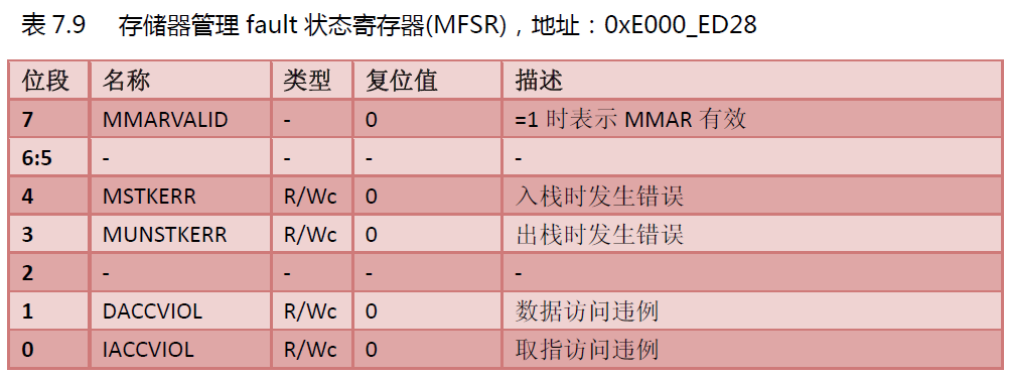
◦ 如果同时还发生了其它高优先级异常，则优先处理这些高优先级的异常，MemManage 异常被悬起。

◦ 如果此时处理器已经在处理同级或高优先级异常，或者MemManage fault 服务例程被除能，则和

总线fault 一样：上访成硬fault，最终执行的是硬fault 的服务例程。

◦ 如果硬fault 服务例程或NMI 服务例程的执行也导致了MemManage fault，那就不可救药了——

内核将被锁定



Cortex-M3处理器是一款广泛用于嵌入式系统的32位处理器核心，具有内置的存储器管理单元（Memory Management Unit，MMU）。MMU负责管理处理器核心的存储器访问和保护，以提高系统的可靠性和安全性。以下是Cortex-M3存储器管理故障的三种常见情况：

1. 存储器访问错误（Memory Access Violation）：当程序试图访问未分配或未授权的存储器区域时，会发生存储器访问错误。这可能是由于指针错误、越界访问、权限不足或未初始化的存储器等原因引起的。MMU会检测到这种错误，并触发相应的异常处理程序，如HardFault异常。在异常处理过程中，可以对错误进行诊断和处理，以防止系统崩溃或数据损坏。
2. 存储器保护错误（Memory Protection Fault）：Cortex-M3的MMU支持存储器保护功能，可以通过设置存储器区域的权限来限制对存储器的访问。如果程序试图访问受保护的存储器区域，将会触发存储器保护错误。这种错误可能是由于未授权的存储器访问、写保护区域的写操作或特权级别不匹配等引起的。MMU会捕获这种错误并引发相应的异常处理程序，以执行必要的错误处理和恢复操作。
3. 存储器映射错误（Memory Mapping Fault）：Cortex-M3的MMU负责将虚拟地址映射到物理地址，实现存储器的分段和映射。在使用MMU时，如果发生存储器映射错误，表示虚拟地址无法正确映射到物理地址。这可能是由于映射表配置错误、无效的映射关系或存储器空间不足等原因引起的。MMU会检测到这种错误并引发相应的异常处理程序，以进行错误的诊断和修复。

在处理这些存储器管理故障时，可以通过使用调试工具、异常处理程序和错误日志等手段来定位和解决问题。此外，合理的存储器管理和访问权限配置、正确的指针操作和存储器分配策略，以及对存储器错误进行适当的异常处理和恢复措施，都有助于提高系统的稳定性和可靠性。

## 9.综合比较OTP、ROM、NAND



OTP (One-Time Programmable)、ROM (Read-Only Memory) 和 NAND (Not AND) 是三种常见的储存器类型，它们在应用和特性上有所不同。下面是对它们进行的综合比较：

1. 功能：
   * OTP：OTP是一种只可编程一次的存储器，一旦被编程，数据就不能被修改或擦除。它通常用于存储固定的配置信息、密钥或唯一的序列号等。
   * ROM：ROM是一种只读存储器，其中的数据在制造过程中被编程，用户无法对其进行修改。它通常用于存储固化的程序代码和不会变化的数据。
   * NAND：NAND是一种闪存存储器类型，它可以被多次擦写和编程。它通常用于存储大容量的数据，如操作系统、应用程序、媒体文件等。
2. 可擦写性：
   * OTP：OTP存储器是一次性编程的，编程后无法擦除或修改已存储的数据。
   * ROM：ROM存储器的数据在制造过程中被编程，用户无法对其进行擦除或修改。
   * NAND：NAND闪存是可擦写的，可以通过特定的擦除操作将数据擦除，并用新数据进行编程。
3. 容量：
   * OTP：OTP存储器的容量通常较小，适用于存储少量的固定配置信息或密钥。
   * ROM：ROM存储器的容量可以较大，可用于存储较大的程序代码和不会变化的数据。
   * NAND：NAND闪存存储器的容量通常较大，可以存储大量的数据。
4. 速度：
   * OTP：OTP存储器的读取速度通常较快，因为数据是固化在芯片中的。
   * ROM：ROM存储器的读取速度也较快，因为数据是在制造过程中编程的。
   * NAND：NAND闪存的读取速度相对较慢，但写入速度较快。
5. 成本：
   * OTP：OTP存储器通常具有较低的成本，适用于存储小容量的固定数据。
   * ROM：ROM存储器的成本适中，适用于存储程序代码和固定数据。
   * NAND：NAND闪存的成本相对较高，但随着技术的进步，其成本正在逐渐降低。

综上所述，OTP、ROM和NAND是不同类型的存储器，适用于不同的应用场景。OTP适用于存储少量的固定配置信息或密钥，ROM适用于存储固化的程序代码和不会变化的数据，而NAND闪存适用于存储大容量的可擦写数据。选择适合的存储器类型取决于应用需求、数据的可变性和成本等因素。

## 10.与SAR ADC相比，积分型ADC

将输入电压转换成时间(脉冲宽度信号)或频率(脉冲频率)，然后由定时器/计数器获得数字值

优点：

用简单电路就能获得高分辨率

抗干扰能力强(假设一个对于零点正负的白噪声干扰，通过积分可滤掉该噪声)

缺点：

由于转换精度依赖于积分时间，因此转换速率极低。

初期的单片AD转换器大多采用积分型，现在逐次比较型已逐步成为主流

与SAR（Successive Approximation Register）ADC相比，积分型ADC（Integrating ADC）是一种不同的模数转换器。下面是对它们的比较：

1. 原理：
   * SAR ADC：SAR ADC使用逐次逼近法进行模数转换。它通过逐位逼近来确定输入信号的数字表示，从而实现模数转换。SAR ADC将待转换的模拟信号与参考电压进行比较，并通过逼近逐渐逼近输入信号的数字表示。
   * 积分型ADC：积分型ADC基于积分的原理进行模数转换。它通过对输入信号进行积分，然后测量积分结果的大小来获取模拟信号的数字表示。积分型ADC在一段时间内对输入信号进行积分，并通过比较积分结果与参考电压来进行模数转换。
2. 转换速度：
   * SAR ADC：SAR ADC通常具有较快的转换速度，因为它可以在较短的时间内逼近输入信号的数字表示。SAR ADC的转换速度通常以几百千赫兹（ksps）或更高的速度进行。
   * 积分型ADC：积分型ADC的转换速度相对较慢，因为它需要在一段时间内对输入信号进行积分。积分型ADC的转换速度通常以几十千赫兹（ksps）为单位。
3. 分辨率：
   * SAR ADC：SAR ADC通常具有高分辨率，可以实现高精度的模数转换。常见的SAR ADC可以提供12位、14位或更高的分辨率。
   * 积分型ADC：积分型ADC的分辨率通常较低，一般为8位或10位。由于积分过程中的噪声和非线性等因素，积分型ADC的精度相对较低。
4. 抗噪声性能：
   * SAR ADC：SAR ADC通常具有较好的抗噪声性能，可以在较高的转换精度下工作。它具有较低的噪声和较好的线性度，适用于对信号质量要求较高的应用。
   * 积分型ADC：积分型ADC受到积分过程中的噪声影响较大，因此抗噪声性能相对较差。在高噪声环境下，积分型ADC可能需要额外的滤波措施来提高信号质量。
5. 复杂度和功耗：
   * SAR ADC：SAR ADC通常具有较低的复杂度和功耗，适用于功耗敏感的应用。它只需要一个逐次逼近寄存器和一组比较器，因此电路结构相对简单。
   * 积分型ADC：积分型ADC的电路结构较为复杂，需要积分电路和积分放大器等组件。由于需要在一段时间内积分输入信号，因此功耗较高。

综上所述，SAR ADC和积分型ADC在工作原理、转换速度、分辨率、抗噪声性能、复杂度和功耗等方面有所不同。选择适合的ADC取决于应用的要求，例如需要高速转换、高分辨率和较低功耗的场景更适合选择SAR ADC，而对抗噪声要求较高的应用可能更适合选择积分型ADC。

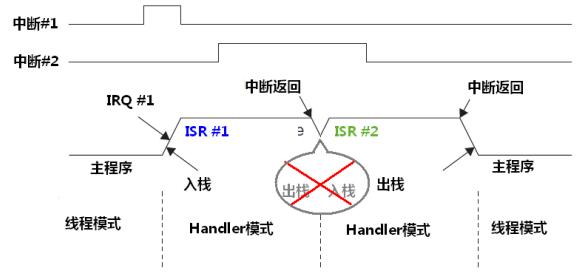
# 11.Cortex M3中断处理中的咬尾

咬尾中断：

在处理器在响应某些异常时，如果又发生其他异常，但它们优先级不够高，则它们会被阻塞。

那么，在当前的异常执行返回后，系统处理悬起的异常时，倘若还是先POP，然后又把POP处理的内容PUSH回去，那么就白白浪费CPU时间了。因此，Cortex-M3不会再POP这些寄存器，而是继续使用上一个异常已经PUSH好的结果，消除POP和PUSH操作的耗时。

这么一来，看上去好像后一个异常把前一个的尾巴要掉了，前前后后只执行了一次PUSH/POP操作。于是，这两个异常之间的“时间沟”就变窄了很多，如图所示：



和常规中断处理（ARM7）的比较：

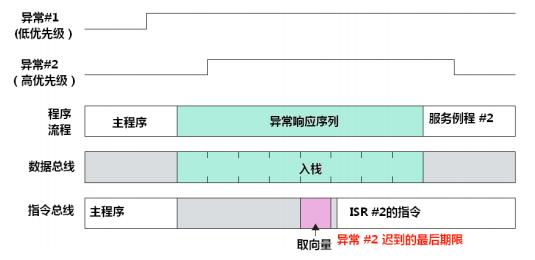


晚到中断

Cortex-M3的中断处理还有另一个机制，它强调了优先级的作用，这就是“晚到的异常处理”。

当Cortex-M3对某异常的响应序列还处在早期：入栈的阶段，尚未执行其他服务程序时。如果此时收到了更高优先级异常的请求，则本次入栈就成了高优先级中断做的了。入栈后，将执行高优先级的异常服务程序。可见，高优先级的异常虽然来晚了，却因为优先级高使得服务程序可以被先处理，低优先级异常的入栈操作则变成了为高优先级异常的入栈。

比如，若在响应某低优先级异常#1的早起，检测到了高优先级异常#2，则只要#2没有太晚，就能以“晚到中断”的方式处理，在入栈完毕后执行ISR#2。如图所示：



如果异常#2来得太晚，以至于已经执行了ISR#1的指令，则按普通的抢占处理，这会需要更多的处理器时间和额外32字节的堆栈空间。在ISR#2执行完毕后，则以“咬尾中断”的方式来启动ISR#1的执行。

内建的嵌套向量中断控制器

◦ 可嵌套中断支持

◦ 可嵌套中断支持的作用范围很广，覆盖了所有的外部中断和绝大多数系统异常。外在表现是，这些异常都可以被赋予不同的优先级。当前优先级

被存储在xPSR 的专用字段中。当一个异常发生时，硬件会自动比较该异常的优先级是否比当前的异常优先级更高。如果发现来了更高优先级的

异常，处理器就会中断当前的中断服务例程（或者是普通程序），而服务新来的异常——即立即抢占。

◦ 向量中断支持

◦ 当开始响应一个中断后，CM3 会自动定位一张向量表，并且根据中断号从表中找出ISR 的入口地址，然后跳转过去执行。不需要像以前的ARM 那

样，由软件来分辨到底是哪个中断发生了，也无需半导体厂商提供私有的中断控制器来完成这种工作。这么一来，中断延迟时间大为缩短。

◦ 动态优先级调整支持

◦ 软件可以在运行时期更改中断的优先级。如果在某ISR 中修改了自己所对应中断的优先级，而且这个中断又有新的实例处于悬起中（pending），

也不会自己打断自己，从而没有重入(reentry)［译注7］风险。

◦ 中断延迟大大缩短

◦ Cortex‐M3 为了缩短中断延迟，引入“咬尾中断”和“晚到中断”等新特性。包括自动的现场保护和恢复，以及其它的措施，用于缩短中断嵌

套时的ISR 间延迟。

◦ 中断可屏蔽

◦ 既可以屏蔽优先级低于某个阈值的中断/异常 (设置BASEPRI 寄存器)，也可以全体封杀(设置PRIMASK 和FAULTMASK 寄存器)。这是为了让时间关

键（time‐critical）的任务能在死线(deadline，或曰最后期限)到来前完成，而不被干扰。

在Cortex-M3处理器中，"咬尾"（Tail-Chaining）是一种优化技术，用于提高中断处理的效率。当一个中断发生时，处理器会立即响应该中断，并跳转到中断服务例程（ISR）执行相应的操作。在传统的中断处理中，当ISR执行完成后，处理器会返回到中断之前的执行点，然后再检查是否有其他挂起的中断需要处理。

而当启用了咬尾优化时，当一个中断的ISR执行完成后，处理器会立即检查是否有其他挂起的中断，并直接跳转到下一个挂起中断的ISR，而不需要返回到中断之前的执行点。这种优化技术可以减少中断处理的延迟，并提高系统的响应速度。

咬尾优化的实现依赖于中断向量表（Interrupt Vector Table）的设置和中断控制器的支持。中断向量表是一个存储中断处理程序地址的表格，每个中断都有一个唯一的中断向量。当中断发生时，处理器会根据中断号查找中断向量表，并跳转到对应中断的ISR。

对于支持咬尾优化的中断控制器，当一个中断的ISR执行完成后，中断控制器会自动检查是否有其他挂起的中断，并根据优先级决定是否触发下一个中断。这样，中断处理器可以连续地执行多个中断的ISR，从而减少了额外的中断处理开销和延迟。

需要注意的是，咬尾优化并非适用于所有情况。在某些特定的应用场景下，由于中断之间的依赖关系或资源竞争等问题，可能需要禁用咬尾优化。因此，在使用咬尾优化时，应仔细考虑系统的需求和中断处理的特点，确保其在特定应用中的适用性和正确性。

# 12.描述Cortex M3合法的操作模式

两种操作模式为：

◦ 处理者模式(handler mode，异常服务例程的代码，包括中断服务例程的代码)

◦ 线程模式(thread mode，普通应用程序的代码)。

支持两级特权操作

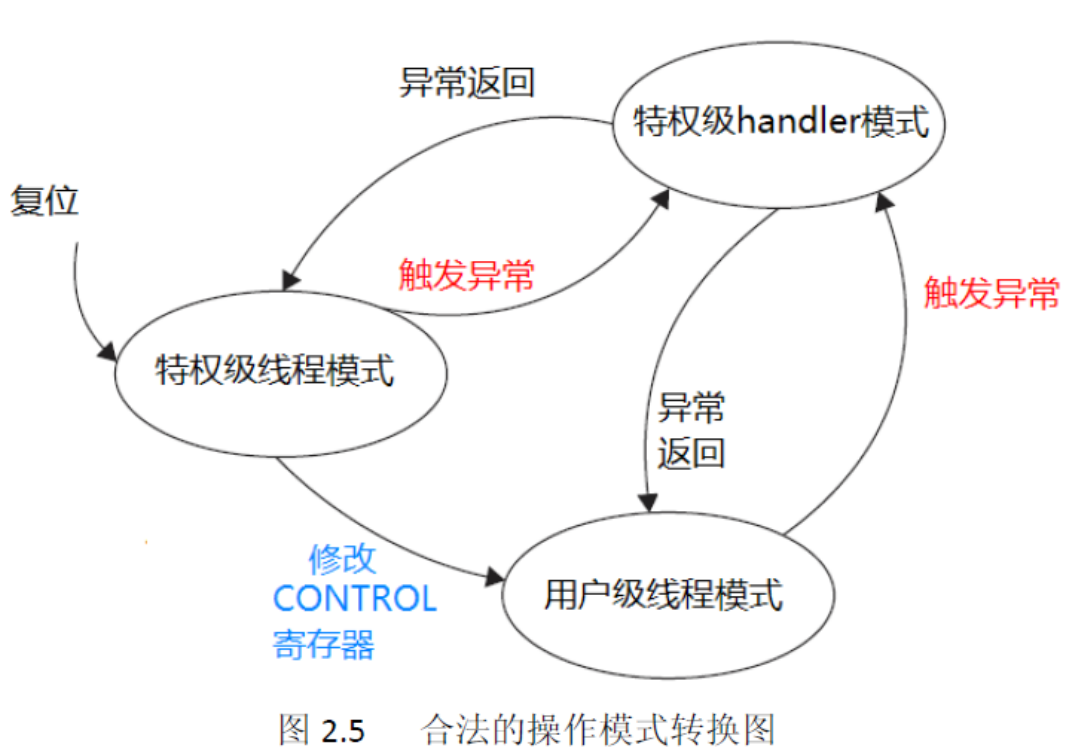
特权：特权级和用户级可以提供一种存储器访问的保护机制，使得普通的用户程序代码不能意外地，甚至是恶意地执行涉及到要害的操作。处理器支持两种特权级，这也是一个基本的安全模型。

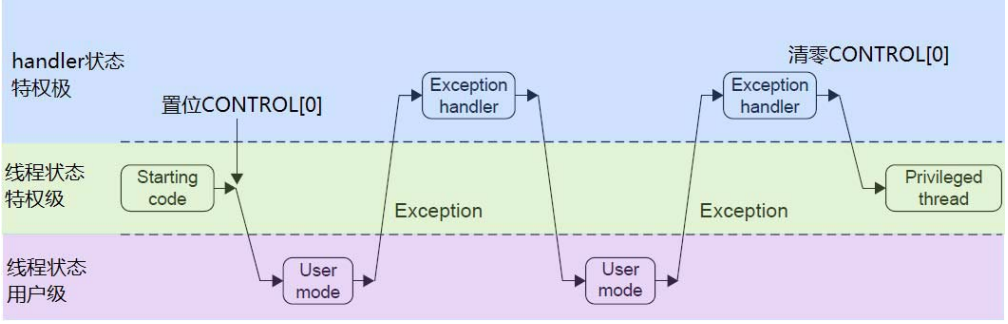
运行线程模式下的普通应用程序时，既可以使用特权级，也可以使用用户级；

但是异常服务例程，也就是处理者模式必须在特权级下执行。

复位后，处理器默认进入线程模式，特权极访问。在特权级下，程序可以访问所有范围的存储器(如果有 MPU，还要 在MPU规定的禁地之外)，并且可以执行所有指令。

在特权级下的程序可以为所欲为，但也可能会把自己给玩进去——切换到用户级。一旦进入用户级，再想回来就得走“法律程序”了——用户级的程序不能简简单单地试图改写 CONTROL寄存器就回到特权级，它必须先“申诉”：执行一条系统调用指令(SVC)。这会触发SVC异常，然后由异常服务例程(通常是操作系统的一部分)接管，如果批准了进入，则异常服务例程修改 CONTROL寄存器，才能在用户级的线程模式下重新进入特权级。 事实上，从用户级到特权级的唯一途径就是异常：如果在程序执行过程中触发了一个异常，处理器总是先切换入特权级，并且在异常服务例程执行完毕退出时，返回先前的状态(也可以手工指定返回的状态)





Cortex-M3是一款基于ARM架构的32位微控制器核心，支持多种操作模式。下面是Cortex-M3的合法操作模式的描述：

1. Thread Mode（线程模式）：Thread Mode是Cortex-M3的默认操作模式，在该模式下，处理器可以执行所有的指令和操作。大多数应用程序在Thread Mode下运行，通过中断或异常切换到其他模式。
2. Handler Mode（处理器模式）：Handler Mode用于处理中断和异常。当发生中断或异常时，处理器会从Thread Mode切换到Handler Mode，执行相应的处理程序。在Handler Mode下，可以访问更多的特权指令和寄存器，以执行特殊操作。
3. Privileged Mode（特权模式）：特权模式是Cortex-M3的特殊操作模式，用于执行特权指令和访问特权寄存器。在特权模式下，可以执行特权级别较高的操作，如配置和管理系统硬件。
4. User Mode（用户模式）：用户模式是一种受限制的操作模式，用于运行用户应用程序。在用户模式下，只能执行一些受限的指令和操作，并且对特权级别较低的寄存器和系统资源有限制性访问。
5. Sleep Mode（睡眠模式）：睡眠模式是一种低功耗模式，用于将处理器置于休眠状态以节省能量。在睡眠模式下，处理器会停止执行指令，并且只有外部中断或复位可以唤醒处理器。

这些操作模式在Cortex-M3处理器中提供了灵活性和安全性。通过适当的模式切换，处理器能够在不同的工作环境和需求下运行，并提供适当的特权级别和资源访问权限。在编程中，可以使用相应的指令和配置来切换和管理这些操作模式。

## 13.解释TEE的概念与原理，它有

TEE（Trusted Execution Environment）是一种安全的执行环境，用于保护敏感数据和执行关键代码的安全性。它提供了一种隔离的执行环境，使得敏感应用程序可以在受到可信任的硬件和软件保护的条件下运行。

概念：  
TEE的概念基于可信计算的理念，旨在解决在不可信环境中执行关键任务时所面临的安全性和隐私性问题。TEE提供了一个受保护的执行环境，其中包含一个安全操作系统（Trusted Operating System）和安全处理器（Trusted Processor）。该环境与主操作系统（Normal World）相隔离，可以独立运行和管理受信任的应用程序（Trusted Applications）。

原理：  
TEE的原理基于硬件和软件的组合，主要包括以下关键技术：

1. 安全处理器：TEE依赖于专门的安全处理器或安全扩展，例如ARM TrustZone技术。安全处理器提供了硬件级别的隔离和保护，将TEE与主操作系统隔离开来。安全处理器通常提供两个执行环境：安全世界（Secure World）和普通世界（Normal World）。安全世界是TEE的执行环境，提供了安全的内存、处理器状态和执行环境等。
2. 安全操作系统：TEE需要一个安全操作系统来管理和控制安全世界的运行。安全操作系统负责安全世界的启动、内存管理、进程调度、安全服务的提供等任务。它提供了安全的API（Application Programming Interface），供开发者在安全世界中编写和运行受信任的应用程序。
3. 受信任的应用程序：TEE允许开发者在安全世界中编写和运行受信任的应用程序。这些应用程序通常处理敏感数据和关键代码，如数字版权管理、安全支付、加密通信等。TEE提供了安全的通信通道和API，使得这些应用程序可以在受到保护的环境中运行，防止恶意软件和攻击者的干扰。
4. 安全通信通道：TEE提供了安全的通信通道，确保安全世界与主操作系统之间的通信不受干扰。通过安全通信通道，受信任的应用程序可以与主操作系统进行安全通信，例如传递数据、请求服务等。

优势和应用：  
TEE的优势在于提供了更高的安全性和隐私性，适用于各种敏感应用场景，如安全支付、数字版权管理、安全存储、安全认证等。TEE的隔离性和安全性使得攻击者难以获取和篡改敏感数据，同时保护关键代码免受恶意软件和攻击的威胁。

需要指出的是，TEE并非绝对安全，仍然可能受到物理攻击、侧信道攻击和软件漏洞等威胁。因此，在设计和使用TEE时，需要综合考虑多种安全措施，并进行全面的安全评估和测试，以确保其满足特定应用场景的安全需求。

## 14.嵌入式软件的安全审计应具备

在进行嵌入式软件的安全审计时，应具备以下要素：

1. 审计目标明确：确定安全审计的具体目标和范围，明确要审计的软件组件、系统功能和安全要求，确保审计的重点和方向明确。
2. 审计计划和流程：制定详细的审计计划和流程，包括审计的时间安排、任务分工、审计方法和工具的选择等。确保审计工作有组织性和系统性，能够全面覆盖需要审计的内容。
3. 审计工具和技术：选择适当的审计工具和技术，用于分析和评估嵌入式软件的安全性。这可能包括静态代码分析工具、漏洞扫描工具、模糊测试工具等。同时，需要了解和应用相关的安全标准和最佳实践，以确保审计工作的有效性和准确性。
4. 审计人员的专业知识和经验：审计人员应具备相关的安全知识和经验，了解嵌入式系统和软件的特点，熟悉常见的安全漏洞和攻击方式。他们应具备分析和发现潜在安全问题的能力，并能提供相应的解决方案和建议。
5. 安全要求的验证和测试：审计过程中应进行安全要求的验证和测试，确保嵌入式软件满足相应的安全标准和规范。这可能包括功能测试、安全性能测试、安全漏洞测试等。通过验证和测试，可以发现潜在的安全问题，并及时采取措施进行修复和改进。
6. 审计报告和建议：完成安全审计后，应撰写详细的审计报告，包括审计过程、发现的安全问题、风险评估和建议等。报告应清晰、准确地呈现审计结果，为软件开发人员和系统设计者提供有针对性的改进意见和建议。
7. 持续改进和更新：安全审计是一个持续的过程，随着嵌入式软件的演化和安全威胁的变化，应持续进行安全审计和改进。定期评估和更新安全要求、审计计划和工具，以确保嵌入式软件的安全性得到持续的提升和保持。

这些要素共同确保了嵌入式软件安全审计的有效性和全面性，帮助发现和修复潜在的安全问题，提高嵌入式系统的安全性和可靠性。

## 15.请描述UICC和外部设备的安全

UICC（Universal Integrated Circuit Card）是一种智能卡，通常用于移动通信设备中的SIM卡。它包含一个嵌入式芯片，具有存储和处理能力，用于管理与移动网络的通信和身份验证。

关于UICC和外部设备的安全，以下是一些常见的安全考虑因素：

1. 认证和授权：UICC与移动设备之间的通信应该经过认证和授权。移动设备应验证UICC的身份，并确保只有经过授权的UICC可以与设备进行通信，防止未授权的访问和攻击。
2. 通信加密：UICC与移动设备之间的通信应使用加密协议和算法，以保护通信的机密性和完整性。例如，使用安全的通信协议（如TLS）来加密UICC与设备之间的数据传输，防止数据被窃听或篡改。
3. 安全应用程序：UICC上的应用程序应该受到安全保护，以防止恶意应用程序的运行或对敏感数据的访问。UICC应使用安全的操作系统和应用程序，采取适当的安全措施，如访问控制和数据加密，以保护存储在UICC中的敏感信息。
4. 物理安全：UICC应受到物理安全的保护，以防止物理攻击和非授权访问。例如，将UICC放置在安全的SIM卡托盘中，只允许经过授权的人员进行物理接触和更换。
5. 更新和安全性评估：UICC上的软件应及时更新，以修复已知的安全漏洞和缺陷。此外，UICC的安全性应定期进行评估和审计，以确保其满足最新的安全标准和要求。

这些安全考虑因素有助于确保UICC和外部设备之间的通信和数据处理安全。通过采取适当的安全措施，可以防止未经授权的访问、数据泄露和恶意攻击，提高整个移动通信系统的安全性。

## 16.Global Platform所定义的嵌入

GlobalPlatform是一个非营利性组织，专注于制定和推广安全芯片和嵌入式系统的规范和标准。它定义了嵌入式安全环境（Embedded Secure Element，简称eSE）的规范，eSE是指集成在智能卡、SIM卡、安全芯片等嵌入式设备中的安全芯片。

GlobalPlatform定义的嵌入式安全环境规范包括以下主要组件：

1. 安全应用容器（Secure Application Container）：提供了安全应用程序的运行环境，用于隔离和保护不同的安全应用程序。每个安全应用容器都有自己的运行空间和安全上下文，防止应用程序之间的相互干扰和攻击。
2. 安全元件（Secure Element）：包括硬件和软件组件，用于提供安全存储、加密运算和安全通信功能。安全元件包括密钥存储和管理单元、加密算法、随机数生成器等，用于支持安全应用程序的运行和安全操作的执行。
3. 安全域（Secure Domain）：是一组相关的安全应用程序和安全元件的集合，共享相同的安全环境和安全策略。不同的安全域之间是隔离的，彼此之间的安全性和资源访问是独立的。
4. 安全通信（Secure Communication）：提供了安全通信协议和机制，用于保护与嵌入式安全环境的通信。这包括加密通信、身份验证和完整性保护，以防止数据的泄露、篡改和未经授权的访问。

GlobalPlatform的规范和标准为嵌入式安全环境提供了统一的架构和接口定义，促进了不同厂商的产品互操作性和安全性。这使得各种应用场景中的安全芯片和嵌入式设备能够提供可信的安全功能，例如移动支付、身份认证、物联网安全等。

# 17.引起Cortex M3的用法故障的

用法faults 发生的场合可以是：

◦ 执行了未定义的指令

◦ 执行了协处理器指令（Cortex‐M3 不支持协处理器，但是可以通过fault 异常机制来使用软件模拟协处理器的功能，从而可以方便地在其它Cortex 处理器间移植）

◦ 尝试进入 ARM 状态（因为CM3 不支持ARM 状态，所以用法fault 会在切换时产生。软件可以利用此机制来测试某处理器是否支持ARM 状态）

◦ 无效的中断返回（LR 中包含了无效/错误的值）

◦ 使用多重加载/存储指令时，地址没有对齐。

◦ 另外，通过设置NVIC 的对应控制位，可以在下列场合下也产生用法fault：

◦ 除数为零

◦ 任何未对齐的访问

◦ 为了调查用法fault 的案发现场，NVIC 中有一个“用法fault 状态寄存（UFSR）”，它指出导致用法fault 的原因。

◦ 在服务例程中，导致用法fault 的指令地址被压入堆栈中。

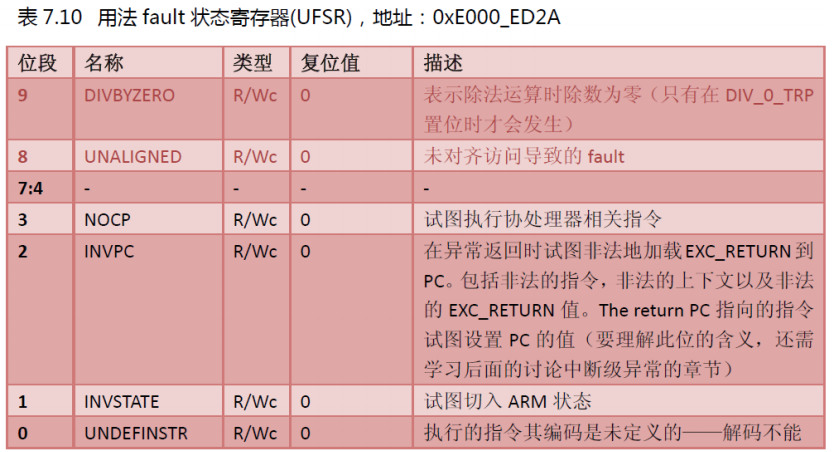
未支持的指令：

◦ BLX #im 在使用立即数做操作数时，BLX 总是要切入ARM 状态。因为

Cortex-M3 只在Thumb 态下运行，故以此指令为代表的，凡是试图切入

ARM 态的操作，都将引发一个用法fault。

为了捕捉被零除的非法操作，你以在NVIC 的配置控制寄存器中置位DIVBYZERO 位。这样，如果出现了被零除的情况，将会引发一个用法fault 异常。如果没有任何措施，Rd 将在除数为零时被清零



引起Cortex M3微控制器使用故障的一些常见原因和问题可能包括：

1. 硬件故障：硬件故障是可能引起Cortex M3使用故障的一个常见原因。这可能包括芯片损坏、电路连接错误、电源问题等。如果硬件出现故障，可能会导致微控制器无法正常工作或产生不可预测的行为。
2. 软件错误：软件错误是另一个常见的引起Cortex M3使用故障的原因。这可能包括编程错误、内存溢出、死循环、错误的中断处理等。这些错误可能导致程序崩溃、无限循环或产生不正确的结果。
3. 时钟和时序问题：Cortex M3微控制器的正常操作依赖于准确的时钟和正确的时序。时钟和时序问题可能导致微控制器的时序不稳定或不正确，从而影响其功能和性能。
4. 电源问题：不稳定或不足的电源供应也可能导致Cortex M3使用故障。如果微控制器没有足够的电源供应或电源不稳定，它可能无法正常工作或产生错误的结果。
5. 外部设备冲突：Cortex M3微控制器通常与外部设备（如传感器、存储器、通信接口等）进行交互。如果外部设备与微控制器之间存在冲突或不兼容，可能会导致微控制器无法正确读取或控制外部设备，从而引发使用故障。

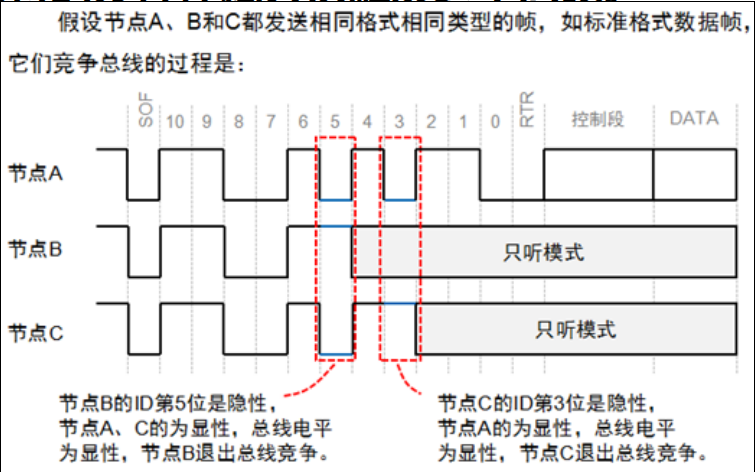
解决Cortex M3使用故障的方法包括：

1. 硬件检查：检查硬件连接、电源供应和时钟配置，确保它们符合规范和要求。
2. 软件调试：使用调试工具和技术，例如调试器、日志记录和追踪功能，来识别和修复软件错误。
3. 时钟和时序验证：验证时钟和时序配置是否正确，确保微控制器的时序满足规范和要求。
4. 电源供应稳定性：确保微控制器的电源供应稳定且符合规范要求，避免电源问题对其正常工作的影响。
5. 外部设备兼容性：确保外部设备与微控制器之间的兼容性，避免冲突和不兼容性导致的故障。

在解决使用故障时，仔细分析和排除可能导致问题的各种因素，使用适当的工具和技术进行调试和验证，以确保Cortex M3微控制器的正常运行。

# 18.请描述CAN总线的仲裁方式。

只要总线空闲，总线上任何节点都可以发送报文，那么就会存在总线访问冲突的可能。CAN使用了标识符的逐位仲裁方法解决这个问题



仲裁期间，每一个发送器都对发送的电平与被监控的总线电平进行比较。

如果电平相同，则这个单元可以继续发送。

如果发送的是一"隐性"电平而监视到的是一"显性"电平，那么这个节点失去了仲裁，必须退出发送状态。

帧ID越小，优先级越高。

由于数据帧的RTR位为显性电平，远程帧为隐性电平，所以帧格式和帧ID相同的情况下，数据帧优先于远程帧；

由于标准帧的IDE位为显性电平，扩展帧的IDE位为隐形电平，对于前11位ID相同的标准帧和扩展帧，标准帧优先级比扩展帧高。

CAN（Controller Area Network）总线是一种常用于车辆和工业控制系统等领域的串行通信协议。CAN总线的仲裁方式是一种用于解决多个节点同时发送消息时的冲突问题的机制。下面是CAN总线的仲裁方式的描述：

1. 仲裁位（Arbitration Bit）：CAN总线使用仲裁位进行消息冲突解决。仲裁位位于CAN帧的标识符字段中，用于标识消息的优先级。优先级高的消息将优先发送。
2. 基于标识符的仲裁：当多个节点同时准备发送消息时，它们会依据标识符进行仲裁。标识符的比特位按照反序进行比较，即从最高位开始比较。节点在发送过程中，会持续检测总线上的仲裁位，如果在发送的过程中检测到总线上的仲裁位与节点发送的仲裁位不一致，则节点会中止发送。
3. 非破坏性仲裁：CAN总线的仲裁方式是非破坏性的，即仲裁过程不会导致数据的丢失或损坏。在发生冲突时，较低优先级的节点会自动中止发送，让较高优先级的节点继续发送。这种方式确保了高优先级消息的传输，同时不会影响其他节点的正常通信。
4. 仲裁段和数据段：CAN总线的仲裁过程分为仲裁段和数据段。仲裁段用于节点之间的仲裁冲突解决，而数据段用于实际的数据传输。在仲裁段中，节点通过比较仲裁位来确定优先级，并在冲突发生时中止发送。在数据段中，优先级最高的节点将继续发送其完整的消息。

通过CAN总线的仲裁方式，多个节点能够根据优先级顺序发送消息，从而实现高效的通信。优先级高的消息将优先发送，而仲裁过程的非破坏性特性确保了消息的可靠传输。这使得CAN总线成为一种可靠的通信协议，适用于实时性要求较高的应用场景。

## 19.Global Platform的安全域应承

GlobalPlatform的安全域（Secure Domain）是一种用于实现安全应用程序隔离和保护的概念。安全域是一组相关的安全应用程序和安全元件的集合，它们共享相同的安全环境和安全策略，并且彼此之间是隔离的。

安全域应承（Secure Domain Assurance）是指对安全域的验证和认证过程，以确保安全域符合GlobalPlatform定义的安全要求和规范。安全域应承包括以下方面：

1. 安全域标识：安全域应承需要确保每个安全域都有唯一的标识符，以便进行识别和管理。
2. 安全域隔离：安全域应承验证安全域之间的隔离性，确保每个安全域在运行时都是相互独立的，防止应用程序之间的干扰和攻击。
3. 安全域安全策略：安全域应承评估安全域的安全策略，包括访问控制、数据保护和安全通信等方面，以确保安全域具有适当的安全措施来保护敏感数据和操作。
4. 安全域漏洞和弱点：安全域应承检测和评估安全域中的潜在漏洞和弱点，包括软件漏洞、设计缺陷和配置错误等，以确保安全域的健壮性和安全性。

通过安全域应承，GlobalPlatform确保安全域在实现安全应用程序时符合一致的安全标准，从而提供可信赖的安全环境。这有助于保护应用程序和敏感数据免受未经授权的访问和攻击，并确保各个安全域之间的互操作性和安全性。

## 20.解释高速串行通信中采用8B/

在高速串行通信中，8B/10B编码（8 bits per 10 bits）是一种常用的数据编码方案。它用于将8位的数据转换为10位的编码，以便在数据传输中实现可靠性和时钟恢复。

8B/10B编码的基本原理是将每个8位数据块映射为10位的编码块。编码过程中，原始数据被分为两个5位的子块，并分别进行编码。其中一个子块用于数据传输，另一个子块用于错误检测和时钟恢复。

在8B/10B编码中，编码表中定义了256个8位数据对应的10位编码。编码表的设计旨在满足一些重要的特性，包括数据平衡、时钟恢复和错误检测。具体来说，编码表的设计要求：

1. 数据平衡：编码表中的每个10位编码均具有平衡的0和1的数量，以便维持数据传输中的直流平衡，避免长时间连续的高或低电平。
2. 时钟恢复：编码表中的编码保证了时钟恢复的可靠性。由于编码块中包含了时钟信息，接收端可以通过解码过程中的状态转换来恢复发送端的时钟信号。
3. 错误检测：编码表中的编码具有一定的错误检测能力。通过检测和纠正编码块中的特定错误模式，8B/10B编码可以提高数据传输的可靠性和抗干扰性。

8B/10B编码广泛应用于许多高速串行通信标准，如PCI Express、USB 3.0和Gigabit Ethernet等。它通过将8位数据转换为10位编码，提供了一种有效的方式来实现高速数据传输的可靠性和时钟恢复。

# 21.与推挽输出相比，开漏输出有

推挽输出

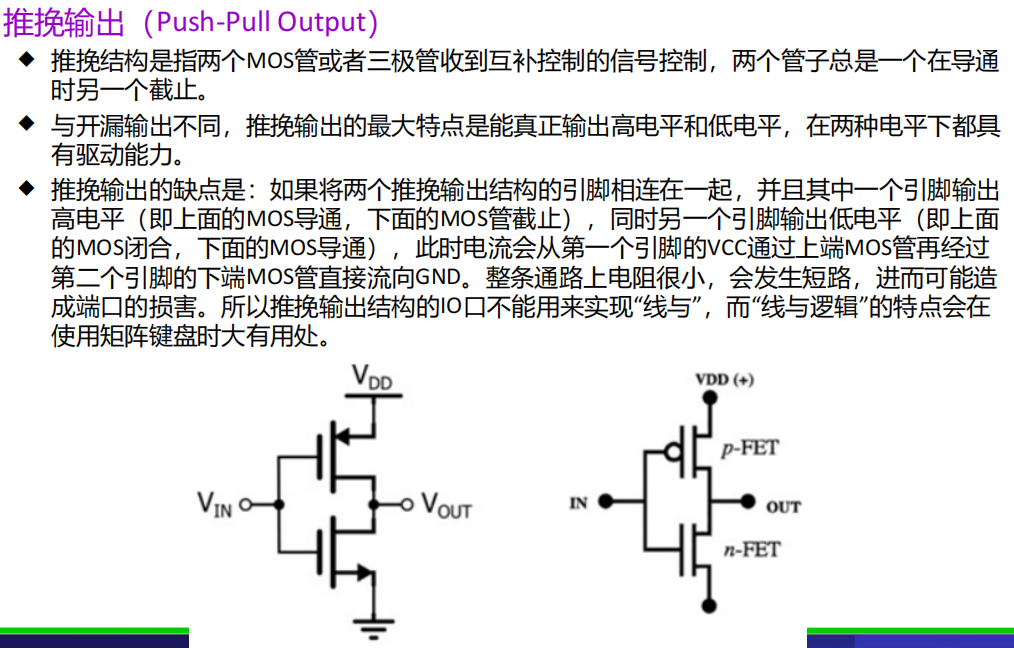
功耗大、驱动能力达、信号快

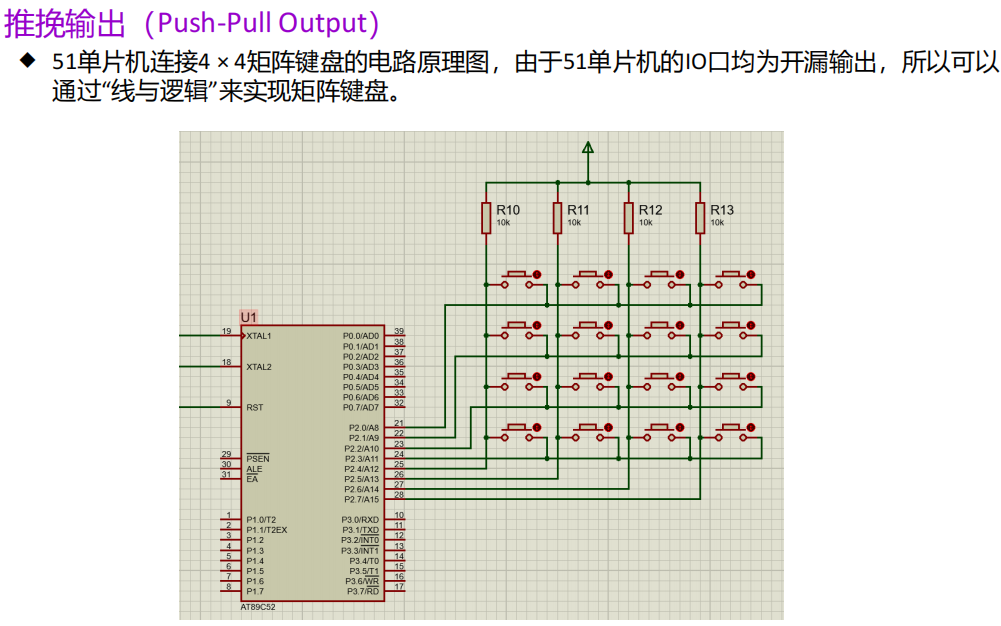
需要考虑集肤效应

导体中有交流电或者交变电磁场时，导体内部的电流分不均匀的一种现象，随着与导体表

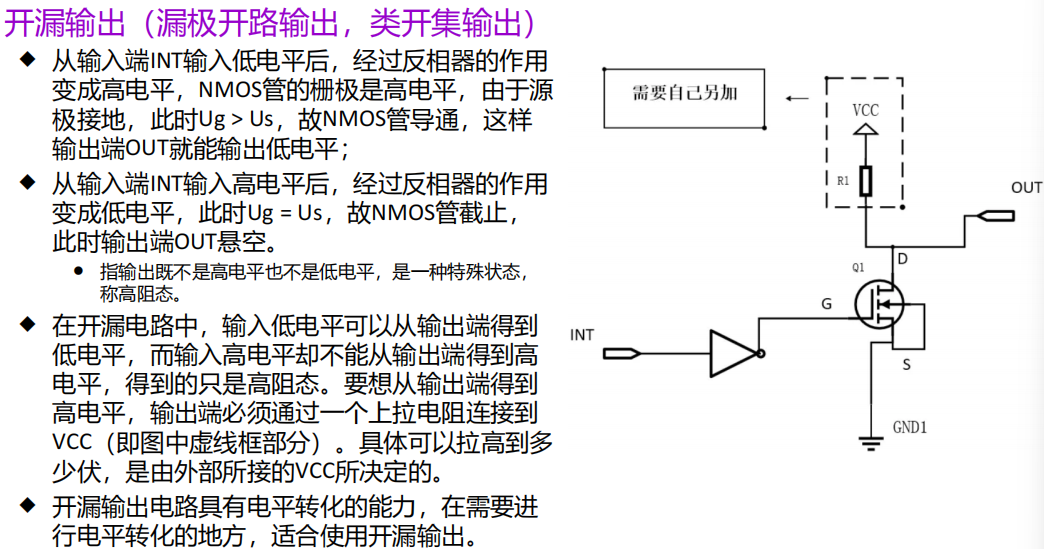
面的距离逐渐增加，导体内的电流密度呈指数衰减，即导体内的电流会集中在导体的表面

常用于板内，不容易形成短路





OC/OD开漏输出，常用于板间





与推挽（Push-Pull）输出相比，开漏（Open-Drain）输出有以下特点：

1. 单向输出：开漏输出只能进行单向的输出，通常是将输出引脚连接到低电平（通常是地）或不连接。而推挽输出可以提供双向的输出，能够驱动高电平和低电平。
2. 无驱动能力：开漏输出不具备直接驱动负载的能力，因此需要通过外部上拉电阻来提供驱动电流。推挽输出可以提供较高的驱动能力，无需外部电阻。
3. 电平转换：开漏输出适用于电平转换的场景。通过连接外部上拉电阻和外部电源电压，可以将开漏输出的低电平转换为高电平。这种电平转换在连接多个设备或总线时非常有用。
4. 多个开漏输出的连接：多个开漏输出可以连接到同一个总线线上，形成多主设备的共享总线。这样的连接方式允许多个设备共享同一条总线，并通过适当的协议实现冲突检测和解决。
5. 输入保护：开漏输出还可以用作输入保护。当开漏输出被配置为输入模式时，它可以有效地保护器件免受电压冲击和干扰。

总的来说，开漏输出适用于需要电平转换、多设备共享总线或输入保护的应用场景。它提供了一种灵活的输出方式，尽管无法直接驱动负载，但通过外部上拉电阻和适当的设计，可以实现各种应用需求。推挽输出则更适合直接驱动负载和提供双向输出的场合。

## 22.阐述嵌入式系统中使用DEP和

在嵌入式系统中，DEP（Data Execution Prevention）和ASLR（Address Space Layout Randomization）是两种常见的安全技术，用于提高系统的安全性和防御能力。

1. DEP（Data Execution Prevention）：DEP是一种硬件和软件的协同机制，用于防止恶意代码的执行和缓冲区溢出攻击。它通过将内存中的数据区域标记为不可执行，使得任何试图在这些区域执行的恶意代码都会触发异常并被终止。DEP可以有效地防止攻击者利用缓冲区溢出漏洞在系统中执行恶意代码。它通常与操作系统和处理器的硬件支持相结合，提供对内存的保护。
2. ASLR（Address Space Layout Randomization）：ASLR是一种内存地址随机化技术，用于增加系统的安全性和抵御攻击。它通过在每次系统启动时随机化程序和库的内存布局，使得攻击者很难准确地确定目标代码和数据的位置。通过随机化内存布局，ASLR能够有效地降低攻击者利用内存地址的成功率，使得攻击变得更加困难。ASLR通常由操作系统提供支持，并可以在编译和链接程序时启用。

综合使用DEP和ASLR可以提高嵌入式系统的安全性和防御能力。DEP防止恶意代码的执行，有效地防御缓冲区溢出攻击。ASLR通过随机化内存布局，增加了攻击者的难度，使得攻击变得更加复杂和困难。这两种技术相互补充，为嵌入式系统提供了一层强大的安全防护。然而，需要注意的是，DEP和ASLR只是安全措施之一，综合的安全设计和实施仍然是确保嵌入式系统安全性的关键。

## 23.中断、异常及陷阱这三个概念

在计算机系统中，中断（Interrupt）、异常（Exception）和陷阱（Trap）是三个重要的概念，它们都涉及到处理器的控制流程的改变和响应。

1. 中断（Interrupt）：中断是一种硬件或软件发起的事件，用于打断正常的程序执行流程，引起处理器的注意并进行相应的处理。中断可以来自外部设备（如键盘、鼠标、定时器等）或内部事件（如硬件错误、系统调用等）。当中断发生时，处理器会中断当前的执行，保存当前上下文，并转入中断处理程序来处理中断事件。处理完成后，处理器会恢复到原来的执行点。
2. 异常（Exception）：异常是指在程序的执行过程中出现的一些非正常的情况，例如除零错误、无效的内存访问等。异常通常是由指令执行期间的错误或特殊条件触发的。当异常发生时，处理器会暂停当前指令的执行，保存当前上下文，并转入异常处理程序来处理异常情况。处理完成后，处理器会根据异常的类型采取相应的处理措施，如继续执行、转到异常处理程序或终止程序执行。
3. 陷阱（Trap）：陷阱是一种有意引发的异常，用于实现系统调用或进行特定的调试操作。陷阱通常是由软件指令显式地触发的，用于切换到内核模式执行特权指令或进行特定的系统操作。当陷阱指令执行时，处理器会暂停当前指令的执行，保存当前上下文，并转入陷阱处理程序来执行相应的操作。处理完成后，处理器会返回到原来的执行点继续执行。

总的来说，中断是用于响应外部事件的机制，异常是用于处理程序执行过程中的非正常情况，而陷阱是一种有意引发的异常，用于特殊的程序控制和系统操作。这三个概念在计算机系统中扮演着重要的角色，对于实现系统的可靠性、安全性和灵活性至关重要。

## 24.在有cache的单片机系统中写

在具有缓存（Cache）的单片机系统中进行写操作时，需要考虑以下几个方面：

1. 写操作的一致性：由于缓存的存在，系统中可能存在缓存与主存之间的数据不一致性。当进行写操作时，需要确保写入的数据同时更新到缓存和主存中，以保持数据的一致性。这可以通过使用写回（Write-back）或写直达（Write-through）策略来实现。

* 写回策略：写回策略将写操作首先写入缓存，而不是立即写入主存。只有当缓存需要被替换出或读取操作需要访问被修改的数据时，才将数据写回主存。这样可以减少主存的访问次数，提高性能。但需要注意及时更新主存，以避免数据不一致性。
* 写直达策略：写直达策略将写操作同时写入缓存和主存，以确保数据的一致性。每次写操作都会导致主存的更新，但可能会增加总线的负载和访存延迟。

1. 写操作的缓存失效：写操作可能会导致相关的缓存行失效。当写操作涉及到某个缓存行时，该缓存行将被标记为无效，以确保下一次对该缓存行的读取能够获取最新的数据。这可能会引起额外的缓存访问延迟，因为需要从主存中加载新的数据。
2. 写操作的原子性：在多线程或多核的系统中，写操作涉及到共享数据时，需要考虑写操作的原子性。原子性保证了对共享数据的写操作在整个过程中不会被中断或干扰。可以使用原子操作或锁机制来实现写操作的原子性，以防止多线程环境下的数据竞争和不一致性。

需要注意的是，具体的单片机系统和缓存架构可能会有不同的特性和配置选项。因此，在进行写操作时，建议参考特定单片机的文档和手册，了解系统的具体缓存行为和配置选项，以确保正确处理写操作的一致性和性能。

## 25.将以下分支程序代码使用805

## 26.将以下分支程序代码使用ARM

## 27.嵌入式系统中浮空输入电路有

在嵌入式系统中，浮空输入电路（Floating Input Circuit）用于处理输入信号未连接或处于未定义状态时的情况。当输入信号未连接时，输入引脚处于高阻抗状态，没有明确的电平值。这可能导致不确定的结果或系统错误。为了避免这种情况，可以采用以下几种浮空输入电路设计：

1. 上拉电阻（Pull-up Resistor）: 在需要浮空输入的引脚上，可以连接一个上拉电阻。这个电阻将引脚连接到正电压电源（例如Vcc），使得当输入未连接时，引脚被拉高到高电平。这样可以确保输入信号具有一个明确的默认状态。
2. 下拉电阻（Pull-down Resistor）: 类似地，在需要浮空输入的引脚上，可以连接一个下拉电阻。这个电阻将引脚连接到地（GND），使得当输入未连接时，引脚被拉低到低电平。这样也可以确保输入信号具有一个明确的默认状态。
3. 外部信号源：在某些情况下，可以通过外部信号源来提供一个明确的输入信号。例如，可以连接一个外部电平转换器或信号源到输入引脚，以确保输入信号始终具有一个确定的电平值。

这些浮空输入电路设计可以根据具体的系统需求和引脚特性进行选择。需要根据电路的工作环境、信号的稳定性要求和噪声抑制等因素来决定使用哪种浮空输入电路设计。此外，还可以结合软件处理来进一步处理浮空输入，例如通过软件检测和处理未连接的输入信号，以避免不确定性和错误。

## 28.MCU芯片为什么通常采用CM

MCU芯片（Microcontroller Unit）通常采用CMOS技术（Complementary Metal-Oxide-Semiconductor）的原因有以下几点：

1. 低功耗：CMOS技术在静态功耗方面表现出色。由于CMOS电路只在切换时消耗功耗，因此在静态状态下几乎没有功耗。这对于嵌入式系统来说非常重要，因为它们通常需要长时间运行，例如电池供电的设备或低功耗应用。
2. 高集成度：CMOS技术允许在芯片上集成更多的功能和复杂的电路。它可以实现密集的集成电路设计，并且具有较小的晶体管尺寸，这使得在单个芯片上集成处理器核心、存储器、外设接口和其他逻辑电路成为可能。这种高度集成的设计有助于减小芯片尺寸、降低成本，并提供更好的性能。
3. 抗噪声和抗干扰能力：CMOS技术对电磁干扰和噪声具有很好的抵抗能力。由于CMOS电路的输出是由两个互补的晶体管组成，它们对来自外界的噪声和干扰信号具有较高的免疫能力。这是在嵌入式系统中至关重要的特性，因为它们通常需要在噪声环境和复杂的电磁干扰中正常工作。
4. 低电压操作：CMOS技术可以在较低的工作电压下正常运行。相比于其他集成电路技术（如TTL），它需要较低的电压供电。这有助于减小功耗，并且使得嵌入式系统在低电压条件下工作更加可靠。

综上所述，CMOS技术在功耗、集成度、抗干扰和低电压操作等方面具有优势，使其成为MCU芯片的首选技术。它能够满足嵌入式系统对低功耗、高性能和可靠性的需求，同时还具备较小的尺寸和较低的成本优势。

## 29.编程实现安全比较

在C语言中，实现安全比较（Secure Comparison）可以采用以下方法：

1. 使用memcmp\_s函数：C11标准引入了memcmp\_s函数，它提供了安全的内存比较操作。该函数需要包含头文件<string.h>，并且具有以下原型：

errno\_t memcmp\_s(const void \*s1, rsize\_t s1max, const void \*s2, rsize\_t s2max, int \*diff);

s1和s2是要比较的内存区域的指针，s1max和s2max分别是两个内存区域的大小，diff是一个指向int类型的变量，用于存储比较结果的差异。这个函数会在比较操作时检查内存区域的大小，防止越界访问。

以下是使用memcmp\_s函数进行安全比较的示例：

#include <string.h>

#include <stdio.h>

int main() {

char str1[] = "Hello";

char str2[] = "World";

int diff;

size\_t str1Len = sizeof(str1);

size\_t str2Len = sizeof(str2);

memcmp\_s(str1, str1Len, str2, str2Len, &diff);

if (diff == 0) {

printf("The strings are equal.\n");

} else if (diff > 0) {

printf("str1 is greater than str2.\n");

} else {

printf("str1 is less than str2.\n");

}

return 0;

}

1. 手动逐字节比较：如果你使用的是较早的C标准或不支持memcmp\_s函数，你可以手动逐字节地比较数据。在比较过程中，需要小心处理边界条件和长度检查，以避免越界访问。以下是一个示例：

#include <stdio.h>

int secureCompare(const char\* str1, const char\* str2) {

size\_t i = 0;

while (str1[i] == str2[i]) {

if (str1[i] == '\0') // 字符串相等，已达到末尾

return 0;

i++;

}

return (str1[i] > str2[i]) ? 1 : -1;

}

int main() {

char str1[] = "Hello";

char str2[] = "World";

int result = secureCompare(str1, str2);

if (result == 0) {

printf("The strings are equal.\n");

} else if (result > 0) {

printf("str1 is greater than str2.\n");

} else {

printf("str1 is less than str2.\n");

}

return 0;

}

无论使用哪种方法，都要确保比较的内存区域大小足够，以避免越界访问。此外，还应该注意检查错误码或返回值，以确保比较操作成功。

## 30.编程实现安全拷贝

在C语言中，可以使用以下方法来实现安全拷贝：

1. 使用memcpy\_s函数：C11标准引入了memcpy\_s函数，它提供了安全的内存拷贝操作。该函数需要包含头文件<string.h>，并且具有以下原型：

errno\_t memcpy\_s(void \*dest, rsize\_t destsz, const void \*src, rsize\_t count);

dest是目标内存的指针，destsz是目标内存的大小，src是源内存的指针，count是要拷贝的字节数。这个函数会在拷贝操作时检查目标内存的大小，防止缓冲区溢出。

以下是使用memcpy\_s函数进行安全拷贝的示例：

#include <string.h>

int main() {

char src[] = "Hello, World!";

char dest[20];

size\_t srcLen = strlen(src) + 1; // 获取源字符串的长度（包括终止符）

size\_t destSize = sizeof(dest); // 获取目标数组的大小

memcpy\_s(dest, destSize, src, srcLen);

// 在目标数组末尾添加终止符，以确保拷贝的字符串正确终止

if (srcLen <= destSize)

dest[srcLen - 1] = '\0';

printf("Copied String: %s\n", dest);

return 0;

}

1. 手动逐字节拷贝：如果你使用的是较早的C标准或不支持memcpy\_s函数，你可以手动逐字节地拷贝数据。在拷贝过程中，需要小心处理边界条件和长度检查，以避免缓冲区溢出。以下是一个示例：

#include <stdio.h>

void secureCopy(char\* dest, size\_t destSize, const char\* src) {

size\_t i = 0;

while (src[i] != '\0' && i < destSize - 1) {

dest[i] = src[i];

i++;

}

dest[i] = '\0'; // 在目标字符串末尾添加终止符

}

int main() {

char src[] = "Hello, World!";

char dest[20];

secureCopy(dest, sizeof(dest), src);

printf("Copied String: %s\n", dest);

return 0;

}

无论使用哪种方法，都要确保目标内存足够大，以容纳源数据并适当处理字符串的终止符（'\0'）。此外，还应该注意检查错误码或返回值，以确保拷贝操作成功。