嵌入式往年试卷核心考点总结

一、 改错题 (判断对错并改正)

这类题目主要考察基础概念的准确性。

1. **C标准库使用 (2019)**
    - **原题:** 使用C语言标准库可以加快开发进度, 但在嵌入式系统开发中要尽量避免使用标准库。
    - **答案:** √ (正确)。
    - **理由:** 嵌入式系统资源有限（ROM/RAM），且对实时性、确定性要求高。标准库可能体积大、执行时间不确定、非可重入，因此常避免使用或使用裁剪过的嵌入式专用库。
2. **ADC/DAC 功能 (2019)**
    - **原题:** 将数字信号转换成模拟信号的电路是 ADC 转换器。
    - **答案:** × (错误)。
    - **改正:** 将**数字**信号转换成**模拟**信号的电路是 **DAC** (Digital-to-Analog Converter) 转换器。ADC (Analog-to-Digital Converter) 是将模拟信号转换为数字信号。
3. **Thumb 指令集 (2019)**
    - **原题:** Thumb 指令集是复杂指令集。
    - **答案:** × (错误)。
    - **改正:** Thumb 指令集是 **RISC** (精简指令集计算) 架构的一种 **16位** 指令集，用于提高 ARM 处理器的代码密度。
4. **NAND 与 NOR Flash 成本 (2019)**
    - **原题:** Nand Flash 比 Nor Flash 成本高。
    - **答案:** × (错误)。
    - **改正:** 通常 **NAND Flash** 的**单位比特成本更低**，适合大容量存储。NOR Flash 成本较高，但支持 XIP (Execute-In-Place，代码原地执行)。
5. **ARM TrustZone 技术范围 (2019, 2016)**
    - **原题:** ARM TrustZone 技术是系统范围的安全方法。
    - **答案:** √ (正确)。
    - **理由:** TrustZone 是基于硬件的安全扩展，贯穿处理器核、内存系统和外设，划分安全世界和普通世界，提供系统级的隔离。
6. **RTOS 性能评价 (2020, 2021)**
    - **原题:** 实时操作系统内核的性能可以以每秒钟能做多少次任务切换来评价。
    - **答案:** × (错误)。
    - **改正:** 评价 RTOS 性能的关键指标是**延迟 (Latency)** 和**确定性 (Determinism)**，例如**最大任务切换时间**、**最大中断延迟**和系统响应时间，而不是单纯的切换次数（吞吐量）。
7. **ARM 复位类型 (2020, 2015)**

- **原题:** 在 ARM 中复位属于异常。
- **答案:** √ (正确)。
- **理由:** 复位 (Reset) 是 ARM 架构中优先级最高的异常，用于将处理器初始化到已知状态。

8. **字节序 (Endianness) 存储 (2020, 2015, 2016 下)**
   - **原题 (大端 2020):** 在大端存储模式下存储 32 位数 0x12345678 到 1000H-1003H 四字节单元中, 1000H 单元值为 0x12。
   - **答案:** √ (正确)。 **理由:** 大端模式将最高有效字节 (MSB) 存放在最低地址。(1000H: 12, 1001H: 34, 1002H: 56, 1003H: 78)
   - **原题 (小端 2021):** 在小端存储模式下存储 32 位数 0x1234567 到 2000H-2003H 四字节单元中, 2003H 单元的数值为 0x67。 (注意题目数字是 0x1234567，应为 0x01234567)
   - **答案:** × (错误)。
   - **改正:** 小端模式将最低有效字节 (LSB) 存放在最低地址。(2000H: 67, 2001H: 45, 2002H: 23, 2003H: 01)。所以 2003H 单元的值应为 **0x01**。
   - **原题 (小端 2016下):** 在小端存储模式下存储 32 位数 1234567H 到 2000:200 四字节单元中, 200 单元的数值为 0x70。 (注意数字是 1234567H，应为 0x1234567)
   - **答案:** × (错误)。
   - **改正:** 小端模式。(200H: 67, 201H: 45, 202H: 23, 203H: 01)。所以 200H 单元的值应为 **0x67**。
9. **I2C 总线引脚配置 (2020)**
   - **原题:** I2C 总线中, SDA 需配置成 OD 结构, SCL 不必配置成 OD 结构。
   - **答案:** × (错误)。
   - **改正:** I2C 总线的 **SDA 和 SCL 都需要配置成 OD (Open Drain, 开漏) 或 OC (Open Collector, 开集)** 结构，配合外部上拉电阻，实现 "线与" 逻辑和多主机仲裁。
10. **操作系统内核类型 (2020, 2015, 2016下)**
    - **原题:** 商业操作系统几乎都是不可剥夺型内核。
    - **答案:** × (错误)。
    - **改正:** 大多数商业实时操作系统 (RTOS) 都是**可剥夺型 (Preemptive)** 内核，以保证高优先级任务能及时响应。通用操作系统 (如 Windows, Linux 桌面版) 也是可剥夺型的。早期的或非常简单的内核可能是非剥夺型的。
11. **I2C/SPI 总线类型 (2015, 2016下, 2021)**
    - **原题 (I2C):** I2C 总线是异步通信总线。
    - **答案:** × (错误)。 **改正:** I2C 是**同步**通信总线，由 SCL 时钟线控制数据传输。
    - **原题 (SPI):** SPI 总线是异步通信总线。
    - **答案:** × (错误)。 **改正:** SPI 是**同步**通信总线，由 SCK 时钟线同步数据。
    - **原题 (JART):** JART 总线是同步通信总线。 (JART 可能是 UART 的笔误)
    - **答案 (假设是 UART):** × (错误)。 **改正:** UART 是**异步**通信总线，没有专门的时钟线，依靠起始位、停止位和约定波特率同步。
12. **旁道攻击定义 (2016)**

- **原题:** 旁道攻击是一种针对密码系统实现上的物理攻击方式。但它既没有系统的攻击方式，也没有系统的解决方法。
- **答案:** × (错误) 或 √ (部分正确，看侧重点)。
- **改正/说明:** 旁道攻击确实是针对物理实现的攻击。但是，它**有系统的攻击方法**（如 DPA, SPA, TA 等）和**成体系的防御对策**（如掩蔽、隐藏、随机化等）。说它"没有系统的解决方法"是错误的。

13. **嵌入式操作系统必要性 (2016, 2021)**
    - **原题:** 在嵌入式系统设计中嵌入式操作系统是必需的。
    - **答案:** × (错误)。
    - **改正:** 嵌入式操作系统**不是必需的**。对于简单的应用（裸机程序）或有特殊性能要求的场景，可以不使用操作系统，直接编写前后台系统或状态机。
14. **SPI 时钟时序 (2016)**
    - **原题:** SPI 总线对于不同的串行接口外围芯片, 它们的时钟时序是不同的。
    - **答案:** √ (正确)。
    - **理由:** SPI 协议定义了四种时钟极性 (CPOL) 和相位 (CPHA) 组合模式，不同的外设可能工作在不同的模式下，主机需要配置匹配的时序。
15. **COS APDU 命令头 (2016, 2016下)**
    - **原题:** COS 命令 APDU 包含一个必备的命令头。其中关于命令是否密文传输由 INS 说明。
    - **答案:** × (错误)。
    - **改正:** 命令头确实是必备的。但关于命令是否涉及安全传输（如加密、MAC）通常由 **CLA (Class byte)** 字节指示安全报文格式。**INS (Instruction byte)** 指示具体的命令类型。*(P2 在某些规范下可能与安全相关，但 CLA 更根本)*
16. **YAFFS 文件系统 (2016下)**
    - **原题:** YAFFS 文件系统是为 NOR FLASH 量身定做的。
    - **答案:** × (错误)。
    - **改正:** YAFFS (Yet Another Flash File System) 是专门为 **NAND Flash** 设计的文件系统，考虑了 NAND 的擦除块、坏块管理和写入特性。
17. **嵌入式软件时间相关性 (2021)**
    - **原题:** 嵌入式软件中时间相关性很强的操作是靠中断保证的。
    - **答案:** √ (正确)。
    - **理由:** 中断提供了对外部事件或定时器事件的快速、异步响应能力，是实现精确时间控制和处理时间敏感操作的关键机制。

二、 简答题

1. **嵌入式文件系统写均衡 (Wear Leveling) (2019)**
    - **答案:** 写均衡是指通过特定算法，将擦写操作均匀地分布在 Flash 存储器的各个块上，避免某些块因过度擦写而提前失效，从而延长整个 Flash 寿命的技术。
2. **评估 RTOS 性能的三个重要时间 (2019, 2020, 2021, 2016)**

- **答案:**
  - **任务切换时间 (Task Switching Time):** 从一个任务切换到另一个任务所需的时间。
  - **中断延迟时间 (Interrupt Latency):** 从硬件中断信号产生到中断服务程序 (ISR) 开始执行第一条指令的时间。
  - **系统响应时间 (System Response Time):** 从外部事件发生到系统对其做出响应（通常是相关任务开始执行）的时间。

3. **可重入函数 (Reentrant Function) 如何编写 (2019)**
   - **答案:** 编写可重入函数需要：
     - **不使用静态变量或全局变量进行数据存储:** 或者使用临界区保护对它们的访问。
     - **仅使用传入的参数或在函数栈上分配的局部变量。**
     - **不调用非可重入的函数。**
     - **避免依赖共享资源（或通过互斥锁等机制保护访问）。**
4. **电磁兼容性 (EMC) 包含的两个要素 (2019, 2020, 2016)**
   - **答案:**
     - **EMI (Electro Magnetic Interference, 电磁干扰):** 指设备自身产生的电磁辐射不应超过一定限制，以免干扰其他设备。
     - **EMS (Electro Magnetic Susceptibility, 电磁抗扰度):** 指设备在一定的电磁干扰环境下应能保持正常工作的能力。(简单说：不干扰别人，且能抵抗别人的干扰)
5. **提高循环语句效率的方法 (2019)**
   - **答案:**
     - 在多重循环中，应将最长的循环放在最内层，最短的循环放在最外层.这样可以减少CPU跨切循环的次数；
     - 充分分解小的循环；
     - 公共表达式放在循环外；
     - 使用指针，减少使用数组；`*P++`比`a[i++]`快
     - 判断条件用0；（例如自减延时函数）；
     - while与do while: do•while编译后的代码的长度短于while；while （1） 与for（；I；）：for（；；）编译后的代码的长度短于while（1）；
     - 循环展开；
     - 相关循环放到一个循环里。
6. **串口异步通信为何必须使用外部振荡器 (2019, 2016)**
   - **答案:** UART 异步通信没有时钟线，收发双方依靠**精确约定的波特率 (Baud Rate)** 来同步数据位的采样。内部 RC 振荡器精度和稳定性通常较差（受温度、电压影响大），难以保证在各种工况下双方波特率误差足够小，会导致通信错误。外部晶体振荡器提供高精度、高稳定的时钟源，是保证可靠异步通信的基础。
7. **数据单元 (Data Unit) 组成及为何使用 (2019)**
   - **答案:**

        - **一般组成: （1）数据部分（Data Payload）**：存储实际需要处理或传输的有效数据内容（如传感器数据、计算结果等）。**（2）控制/元数据部分（Control/Metadata）**：包含描述数据或控制数据的附加信息（如地址、长度、校验码、状态标志、时间戳等）。
        - **为何使用:**
            - **结构化管理**：通过固定或可变结构组织数据，便于高效存储、访问和处理。
            - **确保完整性**：元数据中的校验码（如CRC）可检测数据错误，防止篡改。
            - **提升效率**：元数据支持快速寻址、并行处理和资源优化（如内存管理、DMA传输）。
            - **状态与优先级控制**：通过状态标志和优先级字段实现任务调度和流程管理。
            - **兼容性与标准化**：适配通信协议（如数据包头）和硬件接口需求。
8. **WDT 看门狗如何提高系统可靠性 (原理与作用) (2019, 2020, 2021, 2016)**
    - **答案:**
        - **原理:** WDT 是一个定时器电路。程序正常运行时，需要在 WDT 定时器溢出前周期性地对其进行复位（称为“喂狗”）。
        - **作用/提高可靠性:** 如果程序因干扰或错误陷入死循环或“跑飞”，无法按时“喂狗”，WDT 定时器就会溢出，产生一个复位信号，强制系统重启。这能让系统从异常状态中恢复过来，避免长时间宕机，从而提高系统的可靠性和可用性。
9. **提高嵌入式系统可靠性，如何配置未使用的引脚 (2019, 2015 答案)**
    - **答案:**
        - **参考芯片手册:** 官方手册通常有推荐配置。
        - **输出引脚:** 可以配置为**输出低电平**或**悬空**（不连接）。一般不建议配置为输出高电平悬空，以降低功耗和潜在干扰。
        - **输入引脚: 绝对不能悬空！** 悬空的输入引脚电平不定，易受干扰导致逻辑错误，并可能增加功耗。应配置为：
            - **启用内部上拉/下拉电阻** (如果芯片支持)。
            - **连接外部上拉电阻到 VCC 或下拉电阻到 GND**。
            - **配置为输出模式** (如果该引脚可用作 GPIO 输出)。
10. **什么是能量攻击 (Power Analysis Attack) (2019)**
    - **答案:** 能量攻击（或称功耗分析攻击）是一种**侧信道攻击**。它通过**精确测量**密码设备在运行加密算法时**功耗**的变化，分析功耗轨迹与密钥或中间数据之间的相关性，从而推断出密钥信息。例如，简单功耗分析 (SPA) 直接观察功耗波形，差分功耗分析 (DPA) 则利用统计方法分析大量功耗曲线。
11. **什么是嵌入式系统？由哪几部分组成？ (2020, 2015, 2016)**
    - **答案:**
        - **定义:** 嵌入式系统是以**应用为中心**，以**计算机技术为基础**，**软硬件可裁剪**，适用于应用系统对功能、可靠性、成本、体积、功耗有严格要求的**专用计算机系统**。
        - **组成:** 通常由四部分组成：
            - **嵌入式微处理器 (核心):** 如 MCU, EMPU, DSP, SoC。
            - **外围硬件电路:** 包括存储器、电源、时钟、定时器、I/O 接口、传感器、驱动器等。

        - **嵌入式操作系统 (OS) 或调度器:** (可选) 如 RTOS 或简单的任务调度机制。
        - **嵌入式应用软件:** 实现特定功能的应用程序。
12. **ARM 处理器对异常中断的响应过程 (2020)**
    - **答案:**
        1. **保存现场:** 将当前处理器状态寄存器 (CPSR) 保存到对应异常模式的 SPSR (Saved Program Status Register)。
        2. **更新 CPSR:** 改变处理器模式到对应的异常模式，根据异常类型可能屏蔽 FIQ 或 IRQ。设置 T (Thumb) 位为 0 (进入 ARM 状态)。
        3. **保存返回地址:** 将当前指令的下一条指令地址保存到对应异常模式的链接寄存器 (LR, R14)。
        4. **跳转:** 将程序计数器 (PC) 设置为向量表中对应异常的入口地址，开始执行异常处理程序。
13. **嵌入式微处理器体系结构分类及特点 (2020, 2015 答案)**
    - **答案:** 主要分为两类：
        - **冯·诺依曼 (Von Neumann) / 普林斯顿 (Princeton) 结构:**
            - **特点:** 程序存储器和数据存储器**统一编址**，共用一套地址和数据总线。
            - **优点:** 结构简单，地址空间统一，易于实现代码和数据的混合处理（如自修改代码）。
            - **缺点:** 指令和数据访问不能同时进行，存在**冯·诺依曼瓶颈**，影响速度。
        - **哈佛 (Harvard) 结构:**
            - **特点:** 程序存储器和数据存储器**独立编址**，拥有**各自独立**的地址和数据总线。
            - **优点:** 可以**同时读取指令和操作数**，提高了执行速度和数据吞吐率。
            - **缺点:** 结构相对复杂，存储空间分配不如冯诺依曼灵活。 (现代很多处理器采用**改进型哈佛结构**，允许一定程度的数据和指令总线交互)。
14. **TEE 是什么？有哪两种安全级别？ (2020)**
    - **答案:**
        - **TEE (Trusted Execution Environment, 可信执行环境):** 是计算设备主处理器内的一个**安全区域**，它保证加载到其中的代码和数据的**机密性 (Confidentiality)** 和**完整性 (Integrity)** 受到保护。TEE 与富操作系统 (REE, Rich Execution Environment) 并存但相互隔离。
        - **安全级别 (通常指 TEE 解决方案的安全认证级别，或 GlobalPlatform 定义的):**
            - **安全级别 1 (Profile 1 / Security Level 1):** 主要依赖于 **软件隔离** 机制。第一个安全级别（Profile 1）目标是应对软件攻击。
            - **安全级别 2 (Profile 2 / Security Level 2):** 依赖于 **硬件隔离** 机制 (如 ARM TrustZone)，提供更强的保护。 第二个安全级别（Profile 2）目标是同时应对软件和硬件攻击。
15. **解释 Cortex M3 的 NVIC 尾链 (Tail-chaining) (2020)**
    - **答案:** NVIC (Nested Vectored Interrupt Controller) 尾链是 Cortex-M3/M4 中断处理的一种**优化机制**。当一个中断服务程序 (ISR) 即将结束时，如果此时有另一个挂起的中断请求且其优先级**高于**当前运行的其他任务（包括被中断的任务），NVIC **可以跳过**常规的出栈 (restoring context) 和入栈 (saving context) 过程，直接**链接**到下一个 ISR 的入口。这**显著减少了**连续处理中断时的**延迟和开销**。
16. **举例说明什么是时间攻击 (Timing Attack) (2020, 2015 答案)**

- **答案:** 时间攻击是一种**侧信道攻击**。它通过**测量**密码算法或操作**执行所需的时间**来推断敏感信息（如密钥）。
- **例子:**
  - **RSA 模幂运算:** 如果 RSA 实现中，模幂运算 `m^d mod n` 的执行时间依赖于私钥 `d` 的汉明权重（即 `d` 中比特 '1' 的数量）或具体比特值，攻击者可以通过精确测量大量加密或签名操作的时间，统计分析出 `d` 的信息。
  - **比较操作:** 在某些认证协议中，如果字符串比较函数在遇到第一个不匹配字符时就提前返回，那么比较的时间就暴露了匹配字符的数量，攻击者可以逐位猜测密码。例如，比较 `password` 和 `guess`，如果 `guess="pa"` 比 `guess="px"` 执行时间稍长一点点，就暗示前两位匹配了。

17. **Global Platform 定义的应用提供商安全责任 (2020)**
    - **答案:** Global Platform (GP) 规范定义了 TEE 等安全环境的标准。应用提供商 (Application Provider) 在此框架下通常承担以下安全责任：
      - 生成,或者从可信第三方获得密钥自己的安全域的密钥,
      - 提供符合发行商安全策略与标准的应用,
      - 与发行商安全域一起创建和初始化安全域,
      - 根据应用提供商的安全策略提供应用代码的签名,
      - 从发行商获得下载、安装、让渡的预授权,
      - 根据发行商的管理策略，返回下载安装删除和让渡的收条。
18. **嵌入式系统特点 (2016)**
    - **答案:**
      - **专用性强:** 面向特定应用设计。
      - **软硬件可裁剪:** 根据需求配置资源。
      - **资源受限:** 对成本、功耗、体积要求严格。
      - **高可靠性、高实时性:** 许多应用场景要求稳定可靠、快速响应。
      - **通常不具备自主开发能力:** 需要交叉开发环境和工具。
      - **生命周期长:** 相对于通用计算设备。
19. **软实时与硬实时系统区别 (2016)**
    - **答案:**
      - **硬实时系统 (Hard Real-time):** 要求任务**必须**在规定的截止时间 (Deadline) 内完成，任何超时都可能导致系统失败或灾难性后果（如飞行控制、工业控制）。
      - **软实时系统 (Soft Real-time):** **期望**任务尽可能在截止时间内完成，但偶尔的超时**可以容忍**，只会导致系统性能下降或用户体验变差（如网络音视频传输、数据采集）。关键在于结果的及时性而非绝对的精确性。
20. **提高输入输出信号可靠性的软件方法 (2016)**
    - **答案:**
      - **输入:**
        - **软件滤波:** 如滑动平均、中值滤波、去抖动延时判断等，消除噪声和干扰。

        - **周期性重复配置:** 不同内容放在不同 block，防止配置因干扰丢失。
        - **输入范围检查:** 判断输入值是否在合理范围内。
        - **冗余输入:** 使用多个传感器/输入源并进行表决或校验。
        - **超时处理:** 对预期输入设置等待超时。
        - 输入部件寿命管理（例如按键次数）
    - **输出:**
        - **输出回读与校验:** (如果硬件支持) 读取输出状态确认是否设置成功。
        - **周期性刷新输出:** 内存可靠性高于端口，利用端口数据备份周期性刷新内容与配置（注意：内容在前）
        - **冗余输出:** (驱动能力允许时) 使用多个引脚驱动同一负载。
        - **输出保护:** 软件层面限制输出频率、占空比等防止损坏外部器件。
        - **备份:** 将关键输出状态备份到非易失性存储器。
        - **输出部件寿命管理**（例如 LED、马达、喇叭）
21. **从应用角度嵌入式微处理器分类 (2016 下, 2021)**
    - **答案:**
        - **嵌入式微处理器 (EMPU, Embedded Microprocessor Unit):** 功能接近通用 CPU，通常具有较高性能，不集成或集成少量外设，需要外扩 RAM/ROM 和外设芯片构成系统。
        - **嵌入式微控制器 (MCU, Microcontroller Unit): 单片微型计算机**，将 CPU 核、RAM、ROM/Flash、定时器、中断控制器、ADC/DAC、通信接口 (UART, I2C, SPI) 等常用外设集成在**单一芯片**上，构成最小系统。是目前应用最广的类型。
        - **数字信号处理器 (DSP, Digital Signal Processor):** 专门用于**高速数字信号处理**，具有特殊的硬件结构（如哈佛结构、硬件乘加器、多总线）和指令集，适合音视频、通信等计算密集型应用。
        - **片上系统 (SoC, System on Chip):** 将整个电子系统（包括处理器核、存储器、各种 IP 核、模拟电路、接口等）集成在**单一芯片**上，通常面向特定复杂应用（如手机、路由器）。
22. **CISC 与 RISC 指令系统主要差异 (2016 下, 2021)**
    - **答案:**
        - **指令数量与复杂度:** CISC 指令数量多、功能复杂、长度可变；RISC 指令数量少、功能简单、长度固定。
        - **寻址方式:** CISC 支持多种复杂寻址方式；RISC 支持的寻址方式较少且简单。
        - **内存访问:** CISC 很多指令可以直接访问内存；RISC 只有 Load/Store 指令访问内存，其他操作在寄存器间进行。
        - **寄存器:** CISC 寄存器数量较少；RISC 通常有大量通用寄存器。
        - **实现:** CISC 控制器多用微程序实现；RISC 多用硬布线逻辑实现，易于实现流水线。
        - **编译器:** CISC 对编译器优化要求相对较低；RISC 非常依赖编译器的优化。
        - **功耗与面积:** RISC 通常功耗更低、芯片面积更小。
23. **嵌入式系统硬件通常包含哪些电路模块 (2016 下)**
    - **答案:**

- **核心:** 嵌入式微处理器/微控制器。
- **存储器:** RAM (SRAM/DRAM), ROM/Flash。
- **时钟电路:** 提供系统工作时钟 (晶振、RC 振荡器)。
- **复位电路:** 系统上电复位和手动/看门狗复位。
- **电源管理:** 电压转换 (LDO, DC-DC), 功耗管理。
- **输入/输出 (I/O) 接口:** GPIO, ADC, DAC。
- **通信接口:** UART, I2C, SPI, CAN, Ethernet, USB 等。
- **定时器/计数器。**
- **中断控制器。**
- **调试接口:** JTAG, SWD。
- **(可选) 特定外设:** LCD 控制器, 传感器接口, 驱动电路等。

24. **嵌入式操作系统内核主要工作 (2016 下, 2021)**
    - **答案:**
        - **任务管理 (Task Management):** 创建、删除、挂起、恢复任务，管理任务状态。
        - **任务调度 (Task Scheduling):** 按照特定调度算法（如优先级抢占、时间片轮转）决定哪个任务获得 CPU 使用权。
        - **时间管理 (Time Management):** 提供延时、定时器、系统时钟等服务。
        - **内存管理 (Memory Management):** (在较复杂的 RTOS 中) 管理内存的分配和回收（如内存池、堆管理）。
        - **任务间通信与同步 (Inter-Task Communication & Synchronization):** 提供信号量、互斥锁、消息队列、事件标志组等机制，协调任务间的协作和资源共享。
        - **中断管理 (Interrupt Management):** 处理硬件中断，将其与任务关联起来。
25. **推挽输出 (Push-Pull) 与开漏输出 (Open-Drain) 差别及对可靠性影响 (2016 下)**
    - **答案:**
        - **差别:**
            - **推挽输出:** 内部由一个上拉晶体管和一个下拉晶体管组成。输出高电平时上拉管导通，输出低电平时下拉管导通。可以**强力驱动**高低两种电平。
            - **开漏 (OD) / 开集 (OC) 输出:** 内部只有下拉晶体管。输出低电平时下拉管导通将引脚拉到地。输出高电平时，下拉管关闭，引脚处于**高阻态 (Hi-Z)**，需要**外部上拉电阻**才能输出高电平。
        - **对可靠性影响:**
            - **驱动能力:** 推挽输出驱动能力强，速度快，适合驱动单个负载或板内信号。开漏输出驱动能力相对较弱（取决于上拉电阻），速度受 RC 延时影响。
            - **线与逻辑:** 开漏输出可以方便地实现多个输出**并联**在一条总线上，实现 "线与" 逻辑（如 I2C 总线），任何一个输出低电平则总线为低电平。推挽输出直接并联会造成电源和地的短路，**不允许直接并联**。
            - **电平转换:** 开漏输出可以连接到不同于芯片 VCC 电压的上拉电源，方便实现电平转换。
            - **总线应用:** 开漏结构是多主机总线（如 I2C）能够进行仲裁的基础。

        - **功耗:** 开漏输出低电平时有电流流过上拉电阻，可能比推挽输出低电平功耗稍高（取决于负载）。

26. **嵌入式软件为何区分冷热启动 (2021)**
    - **答案:**
        - **冷启动 (Cold Boot):** 指系统完全断电后的首次启动或复位后的启动。需要进行完整的硬件初始化、内存检查（可选）、加载操作系统和应用程序、初始化所有数据结构到默认状态。
        - **热启动 (Warm Boot):** 指系统在未完全断电的情况下进行的重启（如软件复位、看门狗复位）。此时某些硬件状态或内存中的数据可能仍然有效。热启动过程可以**跳过**某些耗时的初始化步骤（如内存自检），并可能尝试**恢复**之前的运行状态或数据，从而实现**更快的启动**。区分冷热启动是为了优化启动时间和根据不同情况执行必要的初始化或恢复逻辑。

27. **嵌入式软件为何需要代码执行序列检查 (2021)**
    - **答案:** 代码执行序列检查是为了**增强系统的安全性和可靠性**：
        - **防止控制流劫持:** 检测程序是否按照预期的顺序执行，防止攻击者通过缓冲区溢出、返回导向编程 (ROP) 等手段篡改程序执行流程。
        - **检测故障:** 程序因干扰或错误（如内存损坏、CPU 故障）可能导致执行流程异常跳转，序列检查可以及时发现这种异常。
        - **保证关键操作顺序:** 某些操作（如解锁、授权、写入关键数据）必须按照特定顺序执行，序列检查确保这些顺序不被打乱。
        - **符合安全标准:** 某些安全认证（如功能安全标准 ISO 26262）要求进行控制流监控。

28. **嵌入式部件配置为何需要刷新 (2021)**
    - **答案:** 定期或在特定条件下刷新嵌入式部件（如外设寄存器）的配置是为了：
        - **对抗干扰:** 瞬时干扰（如电磁脉冲、电源波动）可能导致寄存器的值被意外改变，刷新可以恢复正确的配置。
        - **防止意外修改:** 程序错误或意外的内存写入可能无意中修改了配置寄存器，定期刷新可以纠正错误。
        - **确保一致性:** 在复杂系统中，多个模块可能访问同一外设，或者运行模式切换后，刷新配置可以确保外设处于预期的状态。
        - **提高鲁棒性:** 这是防御性编程的一种体现，增加系统在恶劣环境下稳定运行的能力。

29. **嵌入式系统软件分层 (哪4层) (2021)**
    - **答案:** 一个典型的嵌入式系统软件可以分为：
        1. **硬件抽象层**（Hardware Abstraction Layer, HAL）
            - 提供对硬件的统一接口，屏蔽硬件差异，便于软件移植和开发。
        2. **驱动程序层**（Device Drivers）
            - 直接控制硬件设备（如传感器、外设），实现与硬件的交互。
        3. **操作系统层**（Operating System, OS）
            - 管理硬件资源（如内存、CPU）、进程调度和任务管理（如实时操作系统RTOS或Linux）。
        4. **应用层**（Application Layer）

        - 实现具体功能，如控制逻辑、数据处理或用户交互，直接面向最终用户需求。

30. **嵌入式系统的三要素是什么 (2021)**
    - **答案:** 嵌入式系统的三要素通常指：
        - **嵌入性 (Embedded):** 嵌入到对象体系中，有对象环境要求。
        - **专用性 (Dedicated):** 软、硬件按对象要求裁减。
        - **计算机性 (Computer):** 实现对象的智能化功能。

三、 论述题

1. **阐述串口 (UART) 通信在应用时的注意事项 (2019, 2021)**
    - **答案:**
        - **物理层匹配:**
            - **电平标准:** 确保收发双方电平匹配（如 TTL/CMOS 3.3V/5V, RS-232, RS-485/422）。RS-232 需要电平转换芯片，RS-485/422 注意差分信号和终端匹配电阻。
            - **接线:** TXD 接 RXD，RXD 接 TXD，GND 必须共地。
        - **协议参数一致:**
            - **波特率 (Baud Rate):** 双方必须设置完全一致的波特率，且时钟源精度要足够高（常用外部晶振保证）以减小累积误差。
            - **数据位 (Data Bits):** 通常为 7 或 8 位，双方需一致。
            - **停止位 (Stop Bits):** 通常为 1, 1.5 或 2 位，双方需一致。
            - **校验位 (Parity Bit):** 无校验 (None), 奇校验 (Odd), 偶校验 (Even)。双方需一致。
        - **数据处理:**
            - **数据帧格式:** 理解起始位、数据位、校验位、停止位的结构。
            - **缓冲区管理:** 使用接收缓冲区 (FIFO 或 Ring Buffer) 处理数据，防止数据丢失。注意缓冲区溢出问题。
            - **数据解析:** 定义好应用层的数据协议，进行分包、粘包处理、校验和验证等。
        - **流控 (Flow Control):** (可选) 对于高速或易拥塞的通信，可使用硬件流控 (RTS/CTS) 或软件流控 (XON/XOFF) 防止接收缓冲溢出。
        - **错误处理:** 检测校验错误、帧错误、溢出错误，并进行相应处理（如重传请求、丢弃数据）。
        - **中断方式:** 推荐使用中断方式接收数据，提高效率，避免 CPU 轮询等待。
        - **可靠性与安全:** 长距离或干扰环境下考虑使用 RS-485/422 差分传输；对于敏感数据考虑加密传输；注意物理端口的保护，防止静电或浪涌。
2. **什么是实现攻击？抵御实现攻击的软件方法有哪些？ (2019, 2016)**
    - **答案:**
        - **实现攻击 (Implementation Attack):** 指不直接攻击密码算法本身的数学逻辑，而是利用密码算法在特定硬件或软件上**物理实现**过程中**泄露**的**旁路信息 (Side Channel Information)** 或通过**注入故障 (Fault Injection)** 来获取密钥或敏感信息的攻击方式。
        - **分类:** 主要分为**被动式攻击 (旁路攻击/侧信道攻击 SCA)** 和**主动式攻击 (故障注入攻击 FIA)**。

        - **旁路攻击:** 通过测量和分析物理泄露（如功耗、时间、电磁辐射、声音）进行攻击。
        - **故障注入攻击:** 通过改变环境（电压、时钟、温度）或使用激光、电磁脉冲等手段在计算过程中引入错误，分析错误输出来获取信息。
    - **抵御实现攻击的软件方法 (主要针对侧信道攻击，部分也对故障攻击有效):**
        - **数据层面 - 隐藏 (Hiding) / 掩蔽 (Masking):**
            - **随机化/盲化:** 在计算前对敏感数据（如密钥、明文、中间值）进行随机化处理（如乘以随机数、加随机掩码），计算后再去除随机性。使得泄露的物理量与真实数据间的关联变得复杂。
        - **操作层面 - 隐藏 (Hiding):**
            - **时间均衡 (Constant Time Execution):** 确保操作的执行时间不依赖于操作数的值。例如，避免使用执行时间依赖于数据的分支判断，使用位运算代替条件转移。
            - **插入伪操作/随机延时:** 在指令序列中随机插入不影响结果的伪操作或随机延时，扰乱时间或功耗特征。
        - **流程层面 - 隐藏 (Hiding):**
            - **指令序列随机化:** 对于等价的计算步骤，随机选择不同的执行顺序。
            - **控制流随机化:** 随机化函数调用顺序或基本块执行顺序。
        - **特定算法选择:** 选择本身具有一定抗侧信道攻击特性的算法或实现方式。
        - **软件陷阱/检测:** 设置一些检测机制，判断是否发生异常（可能是故障注入）。
        - **SLEEP 指令:** 在敏感操作间隙插入 SLEEP 指令，可能降低功耗分析的信噪比（效果有限）。
        - **注意:** 软件方法通常需要与硬件配合才能达到最佳效果，且会带来性能开销。

3. **什么是侧信道攻击？抵御侧信道攻击的软件方法有哪些？ (2020, 2021)**
    - **答案:**
        - **侧信道攻击 (Side-Channel Attack, SCA):** 见上一题关于实现攻击的定义，侧信道攻击是实现攻击中的**被动式攻击**。它利用密码设备在运行时无意中泄露的**物理信息**（如运行时间、功耗、电磁辐射、声音等），通过分析这些信息与内部敏感数据（如密钥）之间的相关性来破解密码系统。常见的侧信道攻击有时间攻击 (TA)、功耗分析攻击 (PA, 包括 SPA 和 DPA)、电磁分析攻击 (EMA)。
        - **抵御侧信道攻击的软件方法:** (与抵御实现攻击的软件方法基本相同，重点在于破坏物理泄露与敏感数据的关联性)
            - **掩蔽 (Masking) / 盲化 (Blinding):** 对参与运算的敏感数据加入随机掩码，运算后再去除。这是**最常用和有效**的方法之一。
            - **隐藏 (Hiding):**
                - **时间均衡 / 恒定时间执行 (Constant Time):** 使代码执行路径和时间不依赖于秘密数据。
                - **伪操作 / 随机延时:** 插入无用操作或随机延时扰乱时序。
                - **指令/代码/控制流随机化:** 改变执行顺序。
                - **Shuffling:** 对循环或查表操作的顺序进行随机化。
            - **协议层面:** 设计安全性不依赖于具体实现细节的密码协议。
            - **算法层面:** 选择对侧信道攻击有天然抵抗力的算法变体。

4. **嵌入式软件的安全审计应具备哪些功能？ (2020)**
    - **答案:** 嵌入式软件安全审计旨在发现和评估软件中的安全风险和漏洞，确保其符合安全要求。应具备以下功能：
        1. **安全事件记录**
           能够详细记录与安全相关的事件，包括系统操作、配置变更、异常行为等，帮助管理者发现潜在攻击和错误配置导致的安全隐患。
        2. **历史操作回顾**
           提供管理者回顾和分析之前操作信息的能力，以判断系统是否遭受过攻击或异常行为。
        3. **早期攻击预警**
           通过审计数据识别潜在的系统攻击迹象，及时发出预警，支持快速响应和防护措施的部署。
        4. **详细数据记录**
           记录芯片和操作系统的详细历史信息，以及关键操作事件，如环境传感器的开启关闭、checksum校验失败、应用程序的装载和删除等，确保审计信息的完整性和准确性。
5. **工业控制系统 (ICS) 特点、安全理念、两种解决方案 (2016)**
    - **答案:**
        - **工业控制系统特点：**

          （1）实时性要求高，强调实时 I/O 能力；

          （2）可用性要求高，系统一旦上线，不能接受重新启动之类的响应，中断

          停机必须有计划（例检修）；

          （3）工控硬件要求寿命长、可靠性高，防电磁干扰，防爆，防尘等要求非

          常严格；

          （4）特有的工业控制协议通讯协议，不同厂商控制设备采用不同通信协议，很多协议不公开；

          （5）工控系统上线生产后，一般不会调整；

          （6）工控系统要求封闭性比较强
        - **实现工控系统安全理念：**白名单、层次化、边缘化、透明化
        - **国际上两种不同的工控系统信息安全解决方案：**
            - 主动隔离式解决方案：即相同功能和安全要求的设备放在同一区域内，区域间通信靠专有管道执行， 通过对管道的管理来阻挡非法通信， 保护网络区域及其中的设备。 其典型代表是加拿大 Byres Security 公司推出的 Tofino 工控系统信息安全解决方案。
            - 被动检测式解决方案：被动检测式解决方案延续了 IT 系统的网络安全防护策略。由于 IT 系统具有结构、程序、通信多变的特点，所以除了身份认证、数据加密等技术以外，多采用病毒查杀、入侵检测等黑名单匹配的方式确定非法身份，通过多层次的部署来加强网络信息安全。其典型代表是美国 Industrial De-fender 公司的工控系统信息安全解决方案。

四、 设计题

1. **用 C 语言设计一个去极值算术平均滤波算法的函数 (2015 答案)**
    - **说明:** 去掉 N 个采样值中的最大值和最小值，然后对其余 N-2 个值计算算术平均值。滑动指每次采样一个新值，去掉一个最旧的值，窗口向前滑动。

- **代码示例 (假设窗口大小为 N，去掉 1 个最大 1 个最小):**

```c
// https://blog.csdn.net/m0_37738838/article/details/84869388

#include <stdint.h>
#define N 18
uint8_t Data[N];

/*****************************************************************************
函数名：avg_filter
描述：去极值滑动算数平均滤波算法，待滤波数据存放于 uint8_t Data[18] 中
输入值：无
输出值：无
返回值：去极值滑动滤波的平均值
*****************************************************************************/
uint8_t avg_filter(void) {
    // 指针直接寻址快于数组索引
    uint8_t *p = Data;
    // 使用缓冲区，避免频繁进行数组和指针的运算
    uint8_t temp;
    // 存储数值滤波最大值
    uint8_t max = 0;
    // 存储数值滤波最小值
    uint8_t min = 0xff;
    // 累加和
    uint16_t sum = 0;

    for (; p < Data + 18; p++) {
        temp = *p;
        sum += temp;

        // 使用 < 和 >=，充分利用 PSW 标志位
        if (temp < min) min = temp;
        if (temp >= max) max = temp;
    }

    // 使用移位代替除法运算
    return (sum - min - max) >> 4;
}
```

2. 请使用C语言实现去极值**滑动平均滤波**（2019, 2020,）

```c
#include <Arduino.h>

#define FILTER_N 12

int filter_buf[FILTER_N];  // 滤波缓存，长度为N
int Filter_Value;

// 模拟获取ADC值的函数
int Get_AD() {
```

```
  return random(295, 305);
}

// 去极值滑动平均滤波函数
int Filter() {
  int i;
  int filter_sum = 0;
  int max_val = 0;
  int min_val = 0x7FFFFFFF;  // 初始化为很大值

  // 新数据加入队尾，队列左移一位，丢弃最旧数据
  for (i = 0; i < FILTER_N - 1; i++) {
    filter_buf[i] = filter_buf[i + 1];
  }
  filter_buf[FILTER_N - 1] = Get_AD();

  // 计算总和，找最大最小值
  for (i = 0; i < FILTER_N; i++) {
    int val = filter_buf[i];
    filter_sum += val;
    if (val > max_val) max_val = val;
    if (val < min_val) min_val = val;
  }

  // 去掉最大值和最小值后求平均
  filter_sum = filter_sum - max_val - min_val;
  return filter_sum / (FILTER_N - 2);
}

void setup() {
  Serial.begin(9600);
  randomSeed(analogRead(0));

  // 初始化滤波缓存，避免第一次滤波数据异常
  for (int i = 0; i < FILTER_N; i++) {
    filter_buf[i] = Get_AD();
  }
}

void loop() {
  Filter_Value = Filter();
  Serial.println(Filter_Value);
  delay(50);
}
```

3. **用 C 语言编写读、写多备份数据单元的函数 (2021)**

- **说明:** 设计函数来读取和写入一个数据单元，该单元有多个备份（例如，主+备1+备2）。写入时需要保证所有备份都写入成功（或尽可能写入），读取时能从有效的备份中读出数据。
- **代码示例 (假设有 1 个主数据区，2 个备份区):**

```c
#include <stdint.h> // 假设可用

#define DATA_UNIT_SIZE 64
#define NUM_BACKUPS 3 // 1 主 + 2 备 = 3

// 自定义布尔类型 (如果不能用 <stdbool.h>)
typedef enum { C_FALSE = 0, C_TRUE = 1 } c_bool;

typedef struct {
    uint8_t data[DATA_UNIT_SIZE];
    uint16_t checksum; // 使用 16 位校验和
    uint32_t version;  // 使用 32 位版本号
    // 可以增加一个魔数 (Magic Number) 来快速判断数据是否有效
    // uint32_t magic;
} DataUnit;

// --- 模拟底层存储和状态 ---
// 同样使用 static 变量
static DataUnit storage[NUM_BACKUPS];
// 使用一个状态字节，每位代表一个存储区是否有效 (例如 0x07 代表 3 个都有效)
static uint8_t storage_status = 0;

// --- 辅助函数 ---

// 计算简单累加和校验 (16位)
static uint16_t calculate_checksum16(const uint8_t* data, uint16_t size) {
    uint16_t sum = 0;
    uint16_t i;
    for (i = 0; i < size; ++i) {
        sum += data[i];
    }
    return sum;
}

// 手动实现内存复制 (替代 memcpy)
static void* memory_copy(void* dest, const void* src, uint16_t n) {
    uint8_t* d = (uint8_t*)dest;
    const uint8_t* s = (const uint8_t*)src;
    uint16_t i;
    for (i = 0; i < n; ++i) {
        d[i] = s[i];
    }
    return dest;
}

// --- 模拟底层读写 ---
// location_index: 0=主, 1=备1, 2=备2

static c_bool platform_write_sim(uint8_t location_index, const DataUnit* unit) {
    if (location_index >= NUM_BACKUPS) return C_FALSE;
    // 模拟写入
    memory_copy(&storage[location_index], unit, sizeof(DataUnit));
```

```
    // 更新状态位，标记该位置有效
    storage_status |= (1 << location_index);
    return C_TRUE;
}

static c_bool platform_read_sim(uint8_t location_index, DataUnit* unit) {
    if (location_index >= NUM_BACKUPS) return C_FALSE;
    // 检查状态位，看该位置是否有效
    if (!((storage_status >> location_index) & 1)) {
        return C_FALSE; // 无效数据
    }
    // 模拟读取
    memory_copy(unit, &storage[location_index], sizeof(DataUnit));
    // 校验数据完整性
    if (unit->checksum != calculate_checksum16(unit->data, DATA_UNIT_SIZE)) {
        // 校验失败，标记该位置无效
        storage_status &= ~(1 << location_index);
        return C_FALSE;
    }
    return C_TRUE;
}

// --- 核心读写函数 ---

/**
 * @brief 写入带备份的数据单元 (优化版)
 * @param data_to_write 指向要写入的数据 (大小需为 DATA_UNIT_SIZE)
 * @param current_version 当前数据的版本号
 * @return c_bool C_TRUE 表示写入成功 (至少主备份成功), C_FALSE 表示失败
 */
c_bool write_data_unit_optimized(const uint8_t* data_to_write, uint32_t
current_version) {
    DataUnit unit;
    uint8_t i;
    c_bool primary_success = C_FALSE;

    // 准备数据单元
    memory_copy(unit.data, data_to_write, DATA_UNIT_SIZE);
    unit.version = current_version;
    unit.checksum = calculate_checksum16(unit.data, DATA_UNIT_SIZE);
    // unit.magic = 0xDEADBEEF; // 设置魔数

    // 尝试写入所有备份
    for (i = 0; i < NUM_BACKUPS; ++i) {
        if (platform_write_sim(i, &unit)) {
            if (i == 0) { // 主区写入成功
                primary_success = C_TRUE;
            }
        } else {
            // 可以在这里记录写入失败的位置，进行错误处理
            // 如果主区写入失败，可能需要直接返回 C_FALSE
             if (i == 0) {
```

```
                return C_FALSE; // 主区写入失败则整体失败
            }
        }
    }
    return primary_success; // 只要主区写入成功就认为操作成功
}

/**
 * @brief 读取带备份的数据单元 (优化版)
 * @param buffer 用于存放读取到的数据的缓冲区 (大小需为 DATA_UNIT_SIZE)
 * @param read_version (可选) 用于返回读取到的数据版本号的指针
 * @return c_bool C_TRUE 表示成功读取到有效数据, C_FALSE 表示所有备份均无效或读取失败
 */
c_bool read_data_unit_optimized(uint8_t* buffer, uint32_t* read_version) {
    DataUnit unit;
    uint8_t i;
    uint32_t latest_version = 0; // 如果需要找最新版本
    c_bool found_latest = C_FALSE;
    int best_location = -1; // 记录最新有效数据的位置

    // 查找最新版本的有效数据
    for (i = 0; i < NUM_BACKUPS; ++i) {
        if (platform_read_sim(i, &unit)) { // 读取成功且校验通过
            // 如果需要找到最新版本的数据
            if (!found_latest || unit.version > latest_version) {
                latest_version = unit.version;
                best_location = i;
                found_latest = C_TRUE;
            }
            // 如果只需要找到任意一个有效的，可以在这里直接 break 或 return
            // memory_copy(buffer, unit.data, DATA_UNIT_SIZE);
            // if (read_version) *read_version = unit.version;
            // return C_TRUE;
        }
    }

    // 如果找到了最新有效数据
    if (best_location != -1) {
         // 需要重新读取一次，因为 unit 可能已被下一次循环覆盖
        if (platform_read_sim((uint8_t)best_location, &unit)) {
            memory_copy(buffer, unit.data, DATA_UNIT_SIZE);
            if (read_version) *read_version = unit.version;
             // (可选) 尝试修复其他无效或旧版本的备份区
             // for (i = 0; i < NUM_BACKUPS; ++i) {
             //     if (i != best_location) {
             //         // 可以尝试读取 i，如果无效或版本旧，则用 best_location 的数据覆盖
             //     }
             // }
            return C_TRUE;
        }
    }

```

```
    // 所有备份都无效
    return C_FALSE;
}
```