第 1 题

1) 简述复位的原理与目的，比较上电复位、看门狗复位和软件复位的异同。本课程实验使用的开发板上的复位功能是如何设计的?

2) 本课程实验所用开发板是否使用了消抖设计？是否能用一个施密特触发器来修改相关设计?

答：

1.

原理：

嵌入式系统上电之后，首先要对必要的寄存器、I/O 接口等资源的值和进行状态进行初始化，这个过程称为复位。

目的：

初始化处理器/系统：在系统电源建立过程中，为 CPU 或某些接口电路提供一个几十毫秒至数百毫秒的复位脉冲（高或低）。利用这段时间（大于芯片最小复位时间要求），系统振荡器启动并稳定下来，CPU 复位内部寄存器及指针，为程序运行做准备；

在系统出现故障或异常时：通过复位操作，强制使系统回到一个已知的初始状态，以保证系统可以从一个稳定且预定义的状态开始重新工作。

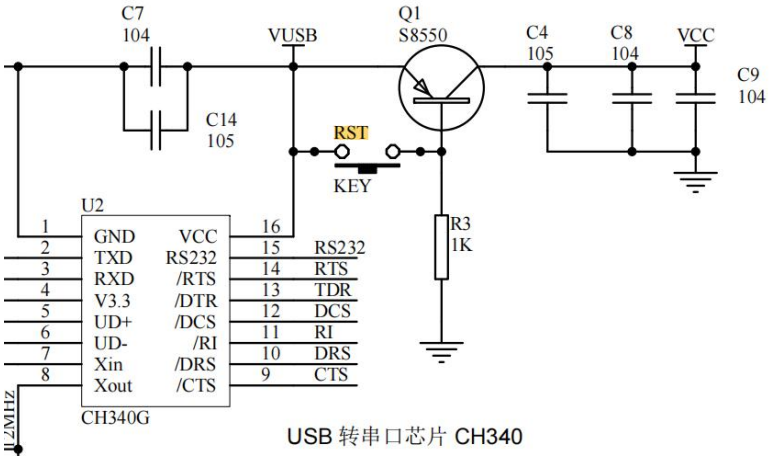
上电复位：在设备上电时自动触发，确保系统从已知状态启动。它是由电源管理电路实现的，通常涉及到一个延时电路，以确保在电源稳定后复位信号被释放。

看门狗复位：由看门狗定时器实现，如果系统在一定时间内没有向看门狗定时器发送“喂狗”信号，看门狗定时器会触发复位。这用于检测和恢复系统从长时间无响应或死锁状态。

软件复位：由软件控制，可以在检测到错误或需要重新初始化系统时触发。软件复位可以通过调用特定的系统函数或写入特定的控制寄存器来实现。

STC-B 中：

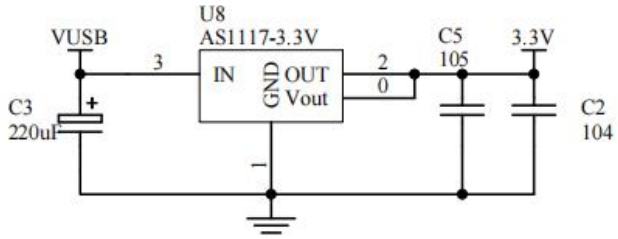
手动复位。

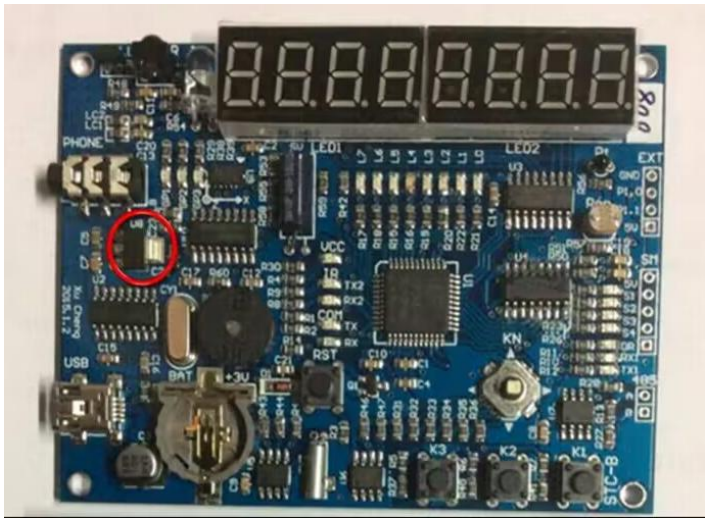


USB 转串口芯片 CH340

电源电路部分：

基于低压差稳压（LDO）芯片 AS1117-3.3V;当输入电压高于 4.5V 之后，输出电压稳定在 3.3V 左右；输入电压为 5V 时，可能出现负阻抗特性，即输出电流增加，输出电压上升。精心设计输入输出回路的滤波电容。

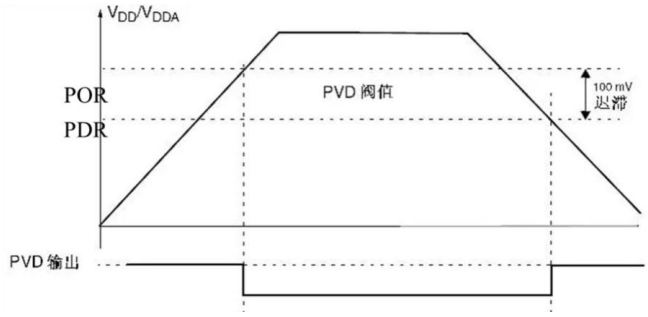




STM32 上电复位内部自带了一个可编程电压检测器（PVD）：

与 Vdd 电压比较，达到监控电压的目的。对 Vdd 的电压进行监控，可通过电

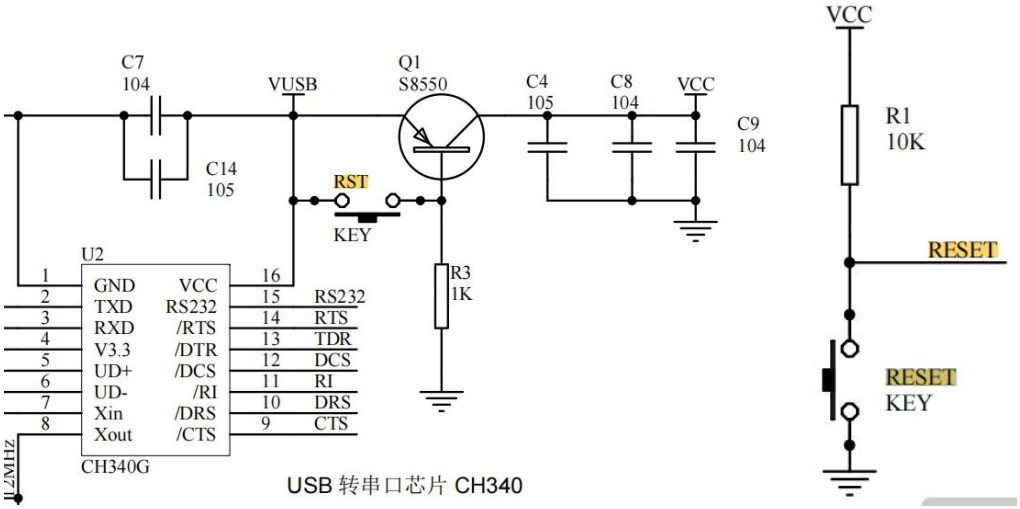
源控制寄存器 PLS[2:0]位来设置监控电压的阀值。当 Vdd 下降到 PVD 阀值以下或 Vdd 上升到 PVD 阀值之上时，通过外部中断线（连接到 PVD 输出）上升或下降边沿触发设置，产生 PVD 中断。



2.

STC-B 中和 STM32 中都没有提供消抖电路，但可以自行实现软件消抖，这个软件消抖的方法老师在课堂上有讲过，还提供了代码。

可以，将斯密特触发器连接在按键连接处，按键信号先经过整形再输入电路，可以减少软件消抖的负担。不过 STC-B 这边可能会有点问题，因为 STC-B 中使用的 RST 并不是传统的那种电阻加电容加开关的结构，它是用截断电源供电来实现的，传统的施密特触发器是利用它的迟滞特性来进行消抖的。



STM32 很容易完成。

第 2 题

1) 什么是最小系统？其蕴含了什么样的思想方法？最小系统有何价值？

2) 分层的理念，在本课程以及相关专业课程中均有出现，请举例，并阐释你至今所学中以分层思想解决的相关问题。

答：

1.

最小系统：

一个仅具有进入正确执行模式所需最少资源的系统。从硬件角度，最小钱入戏系统硬件包括了嵌入式处理器、片上/片外存储器以及电源供电、复位、时钟等外围辅助电路。

思想方法：

最小系统的概念蕴含了“简约性”和“核心功能优先”的思想方法。这意味着在设计阶段，工程师会专注于实现系统的核心功能，而将非核心功能暂时搁置，以便更快地开发和测试基础系统。

价值：

快速迭代：最小系统允许快速开发和测试，因为系统更简单，更容易理解和维护。

降低复杂性：通过剥离非核心功能，最小系统降低了整体复杂性，使得问题更容易诊断和解决。

资源效率：在资源受限的环境中，如嵌入式系统，最小系统可以确保资源得到最有效的利用。

2.

计算机网络课程中，整个课程的设置都是分层的，从上到下依次是应用层、运输层、网络层、链路层、物理层，这种分层方式分离了每一层的内容，使得上

层的服务都在下层调用的基础上，而下层的改动对上层的影响较小。

在本课程中提到的，**嵌入式计算机的基本硬件组成**也是分层的，并实现了模块化，这将耦合降低至最小，便于设计、修改和后期维护。

在**嵌入式软件结构**中也有分层服务，硬件抽象层（HAL），Boot Loader、操作系统层，以及应用层，每一层都为上一层提供必要的服务和接口，同时隐藏下层的复杂性。

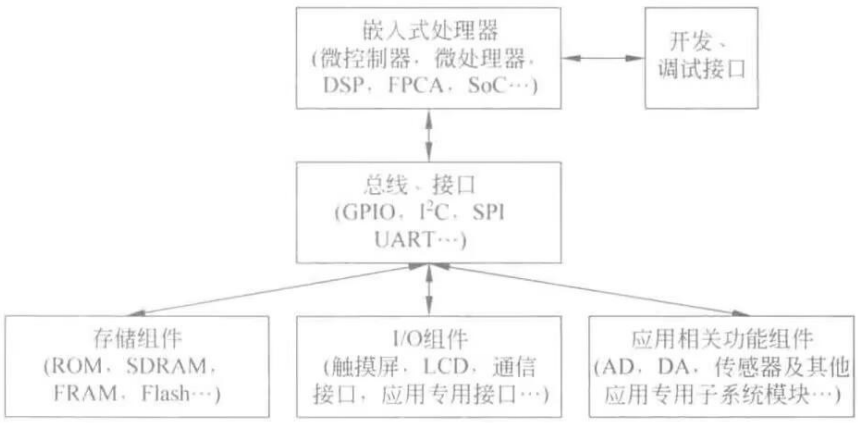


图 2.17 （单台）嵌入式计算机的基本硬件组成

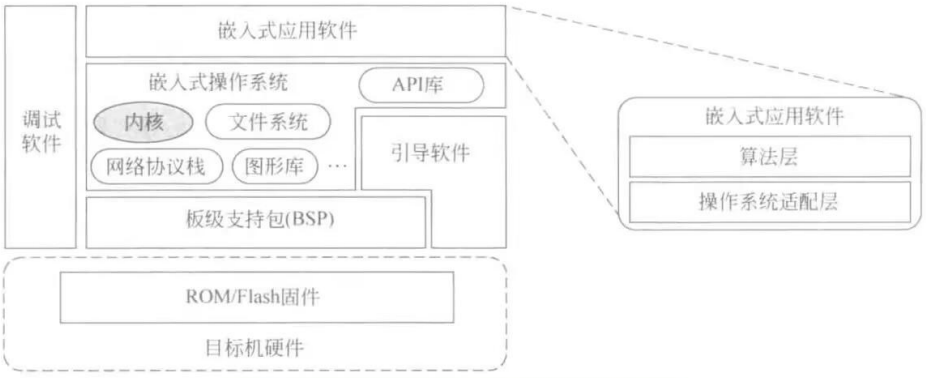


图 7.3 基于嵌入式操作系统的嵌入式软件结构

第 3 题

1) GPIO 与专用接口/引脚有何不同？GPIO 有什么特点？根据你的实验实践经验，如何使用 GPIO?

2) 在第 6 章 图 6.49（教材第一版 P251）描述的是车载 CAN 驱动总线电平，请搜索资料，分析“车载舒适总线”中的电平状态，并参照图 6.49，画出 CAN_H 和 CAN_L 的变化关系。

答：

1.

GPIO 是区别于其他专用 I/O 的、可适用于多种输入/输出要求的 I/O 接口，允许用户根据不同的功能需求进行动态配置和管理。

GPIO 的特点：通过操作相应的控制寄存器，GPIO 可以被配置为数字量模式或模拟量模式，同事也能配置为不同类型开关电路的控制输出接口或者数据总线接口，增强了通用性。它可以节省处理器的引脚、提高引脚利用率，并提升处理器接口功能的灵活性。使用简单，不需要复杂的协议。速度通常比专用接口慢，因为它们是通用的，没有针对特定任务优化。

（1）每次最多允许控制 16 个 I/O；
（2）允许推挽或“开漏＋上/下拉电阻”的输出；
（3）可以从数据输出/输入寄存器（GPIOx_ODR）或外设（复用功能输出/输入）输出/输入数据；
（4）为每个 I/O 设定不同的速度；
（5）支持浮空、上拉/下拉、模拟等输入状态；
（6）按位写输出寄存器（GPIOx_IDR）的置位和复位寄存器（GPIOx_BSRR）；
（7）I/O 类型的寄存器锁定机制（GPIOx_LCKR）；
（8）两个时钟周期的快速开关；
（9）高度灵活的引脚复用，允许将 I/O 引脚配置为 GPIO 或其他外设功能模式；复位后，复用功能未被激活，调试引脚 PA13、PB4 为上拉状态，PA14 为下拉状态，其他所有 I/O 端口被默认配置为输入浮动状态。

需要说明的是，不同的 GPIO 可能对应不同的电源域，那么在使用 GPIO 时就要考虑不同处理器模式下各电源域的供电情况。以待机模式为例，该模式中除了 3 个唤醒引脚之外的其他 GPIO 引脚都不工作。

GPIO 的使用：

要了解和掌握各个 I/O 端口的特性及其寄存器组，就输入输出而言，GPIO 的引脚有浮空、上拉输入（按键常用）、下拉输入、模拟输入（ADC）、开漏输出（电流大的时候用）、可配置的上拉推挽、下拉推挽输出。同时，该类接口至少会提供 GPIO 控制寄存器和 GPIO 数据寄存器，控制寄存器用于控制数据寄存器中各位为输入、输出或其他功能类型。

需要将引脚信号读入 MCU，则选择输入工作模式，反之选择输出工作模式。

普通 GPIO/内置外设输入：浮空，弱上/下拉输入，不使能复用模块。

普通模拟输入：配置为模拟输入模式，不使能复用模块。

普通 GPIO/内置外设 输出：根据需要配置该引脚为推挽或开漏输出，不使能/使能复用模块。

GPIO 工作在输出模式，既要输出高电平又要输出低电平，且输出速度快（比如 OLED 显示），应选择推挽输出。

GPIO 工作在输出模式，若要输输出电流大，或外部电平不匹配（例如 5V），则应选择开漏输出。

（1）输入端口模式。

在该模式下，需要禁止输出缓冲区，激活施密特触发器的输入，根据上拉/下拉电阻寄存器 GPIO*x*_PUPDR 的配置激活上拉或下拉电阻。在每个 AHB 时钟周期，该引脚上出现的数据将被采样到输入数据寄存器，而对输入数据寄存器的访问操作将返回一个 I/O 状态。

（2）输出端口模式。

该模式时需要使能输出缓冲区，激活施密特触发器，并根据 GPIO*x*_PUPDR 决定是否激活弱的上拉/下拉电阻。每个 AHB 时钟周期，该引脚的数据将被采样到输入数据寄存器，读输入数据寄存器的操作会返回引脚状态，而读输出数据寄存器可以获得最近写入的值。在开漏输出模式下，P-MOS 晶体管不工作，此时数据输出寄存器中的 0 将使得 N-MOS 导通，而 1 则使该引脚为高阻抗态。推挽输出模式时，两个场效应管都工作，0 使 N-MOS 导通漏电流，1 使 P-MOS 导通灌电流。

（3）复用功能端口模式。

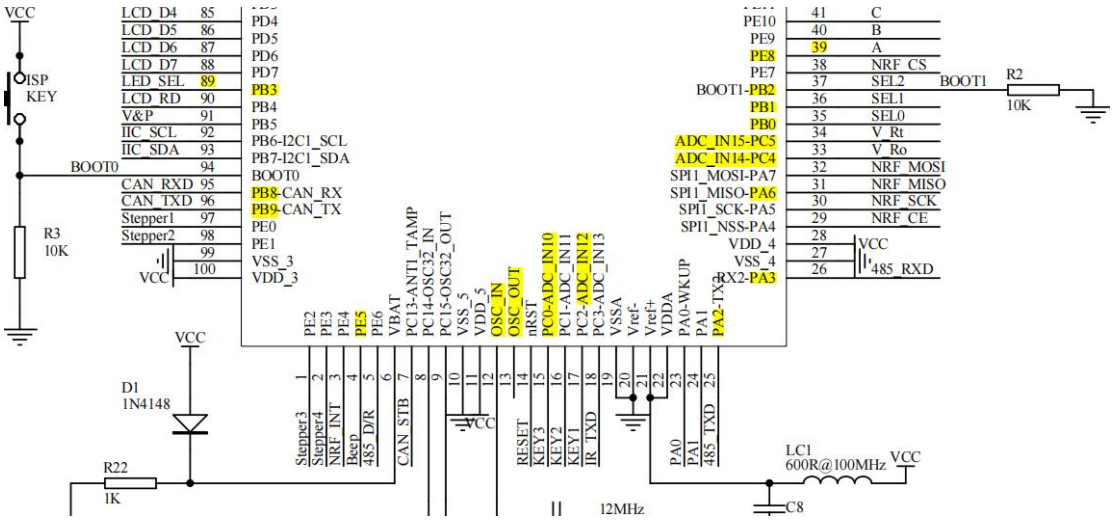
输出缓冲区可配置为开漏或推挽方式，并由来自外设的信号驱动。进而，激活施密特触发器，并根据 GPIO*x*_PUPDR 决定是否激活弱的上拉/下拉电阻。同样，每个 AHB 时钟周期该引脚的数据将被采样到输入数据寄存器，读输入数据寄存器时返回引脚状态。

（4）模拟端口模式。

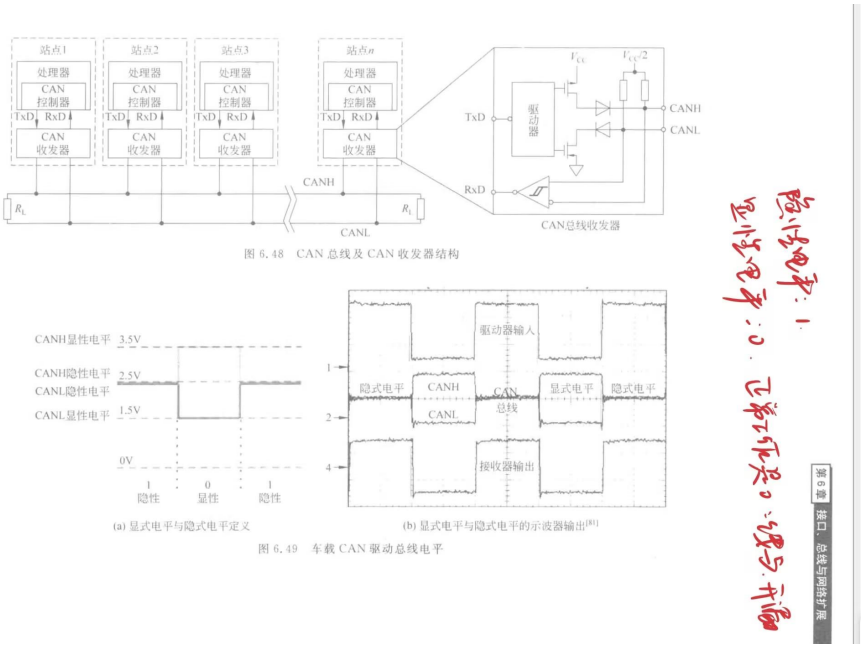
将输出缓冲区禁止、施密特触发器输入禁止且强制输出 0、禁止上拉和下拉电阻，就进入了模拟端口模式，此时读输入数据寄存器的操作返回 0。该模式下，模拟量从该端口的模拟量通道直接输入至芯片内部的 ADC 器件。

实际设计中，如果处理器提供的 GPIO 数量不够用，设计者还可以采用 MAX7319～MAX7329 等 8 端口/16 端口 GPIO 集成电路器件以及驱动器等对系统电路的 GPIO 接口进行扩展。另外，不同处理器的 GPIO 实现和特性存在差异，同一处理器的 GPIO 特性也可能不同，应用时要加以甄别。

在 STM32 中的使用：根据电路图中的连接，在 STM32CubeMX 中将引脚配置为推挽模式，并使能相应的中断。



2.

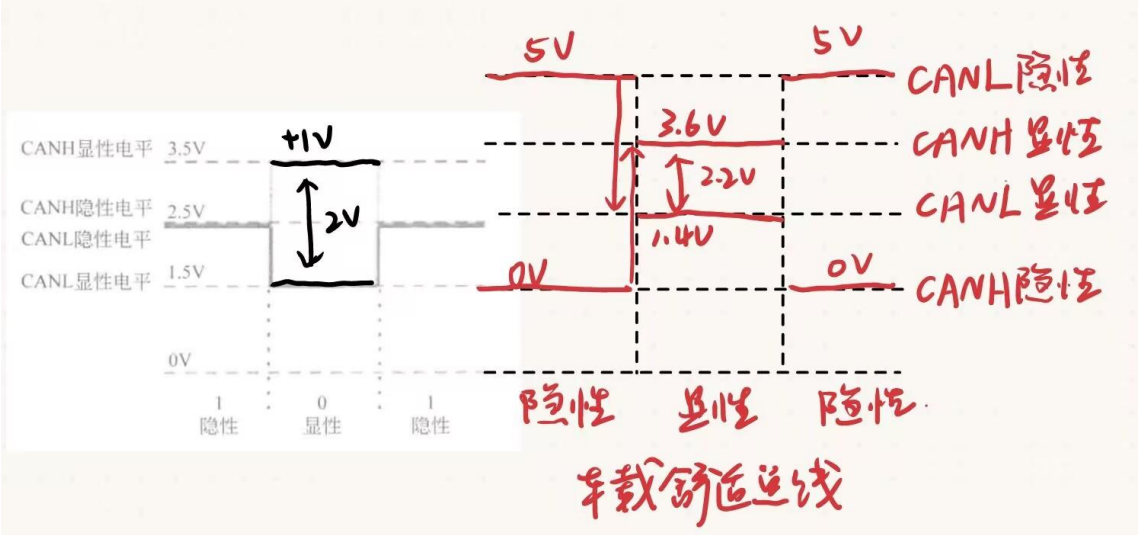


书中有提到车载舒适总线的电平状态，根据它来画：

隐性状态时，CANH 和 CANL 分别为预先设定的电压值，如车载 CAN 驱动总线中都为 2.5V，而车载 CAN 舒适总线中 CANH 为 0V、CANL 为 5V，$V_{CANH}-V_{CANL}$ 为 0V 或 −5V，表示 0。在显性状态时，CANH 上的电压升高一个预定值，如驱动总线中为 1V、舒适总线中为 3.6V，而 CANL 上的电压值会降低一个同样的值，此时（$V_{CANH}-V_{CANL}$）为 2V 或 2.2V，表示 1。某一时刻，总线电平要么处于显性电平，要么为隐性电平，驱动总线的电平变化如图 6.49 所示。如果输出传输信号中引入了噪声电压 V_N，那么接收器收到的电压将

分别为 $V_{CANH}-V_N$ 和 $V_{CANL}-V_N$，二者相减后两线的电压差值仍为 $V_{CANH}-V_{CANL}$。由此我们可以看出，差分传输会有效提高信号的抗干扰能力。

这是我的答案：



第 4 题

1) 结合 STM32 处理器，以及你的个人实验实践经验，分析裸机软件和嵌入式 OS 的启动过程。

2) 根据你使用 STC-B 学习板配套 BSP 的使用经验，简述该 BSP 的功能和基本组成；如果要为 STM32 开发板设计开发一个 BSP，你觉得需要对前述 BSP 进行哪些修改/增补设计？

3) ARM 的 CMSIS 和 STM32 的 HAL 有什么异同/关系？你在实验中是否使用过 TA（们），如果使用过，请简述；如果没有使用过，请描述如果要使用可以如何应用到你在本课程的实验实践中。

答：

1.

裸机软件启动：通常涉及初始化硬件，如时钟、内存和外设。启动过程可能包括设置堆栈指针、初始化全局变量和调用主函数。

嵌入式 OS 启动：除了裸机软件的初始化步骤外，还需要初始化操作系统内

核，如任务调度器、内存管理和中断处理。启动过程包括创建启动任务、初始化设备驱动和启动用户任务。

2.

STC-B：

系统：更改时钟频率。sys 调度等。

中断服务例程：1ms 10ms 100ms 1s 的事件中断，还有各种外设的中断，比如按键、红外、ADC、震动传感器等等。

各种模块的初始化代码：数码管显示，LED 指示灯显示，实时时钟，温度光照测量，音乐播放，FM 收音机，EXT 扩展接口（电子秤、超声波测距、旋转编码器、PWM 控制，4 选 1 工作），振动传感器，霍尔传感器，步进电机控制，串口 1 通信，串口 2 通信（485、EXT 扩展接口，2 选 1），红外遥控，红外收发通信，非易失性 NVM 存储。

对 STM32 的 BSP 增补：

寄存器定义：STM32 与 STC-B 的内核不同，需要重新对寄存器进行定义。

时钟配置：根据 STM32 开发板的时钟树需求，调整时钟源和分频器设置，以满足系统的时钟频率要求。

系统调度适配：实现调度算法，以适应多任务环境。

中断服务例程适配：为 STM32 开发板上的外设编写或修改中断服务例程，以处理来自这些外设的中断请求，并设置优先级。需要对 STM32 的三个输入口，COM、USB、SWD 以及通信的 CAN 模块作相应适配。

模块初始化代码适配：为 STM32 开发板上与 STC-B 不同的特定模块编写初始化代码，确保这些模块能够正确初始化并准备就绪。

3.

CMSIS 是 ARM 提供的一个标准软件接口，用于访问 Cortex-M 处理器的硬件特性。它包括以下几个部分：Core：提供与 Cortex-M 处理器核心相关的功能，如 NVIC（嵌套向量中断控制器）和系统滴答定时器。Device：包含特定微控制器

的外设寄存器定义和初始化代码。RTOS：提供与实时操作系统（RTOS）兼容的接口。DSP：提供数字信号处理（DSP）和数学函数库。Pack：用于管理设备驱动和软件组件的安装包。STM32 HAL：是 ST 公司为 STM32 微控制器提供的硬件抽象层，它建立在 CMSIS 之上，提供更高级的硬件访问接口和简化的编程模型。

HAL 依赖于 CMSIS，为 STM32 系列微控制器提供更丰富的功能和更易用的 API。相当于 HAL 的抽象程度更高一点，CMSIS 更接近底层。

我没有使用过 CMSIS 进行编程，不过我可以想象一下。我记得老师上课的时候好像提到过，可以使用它来配置端口寄存器，讲的可能是控制 LED 点亮的例子，也说有同学在实验中用到了，好像还提供了一点代码，不过我实在想不起来是在哪里讲的了，也没有找到 PPT 上的记录，可能是口述的。

使用 CMSIS 进行编程：

初始化：使用 CMSIS 提供的系统初始化函数，如 SystemInit()，来配置系统时钟和外设。

外设控制：利用 CMSIS 提供的外设寄存器定义来控制微控制器的外设。例如，使用 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;来使能 GPIOA 的时钟。

中断处理：使用 CMSIS 的中断向量表和 NVIC 函数来配置中断优先级和使能中断。例如，NVIC_SetPriority(EXTI0_IRQn, 0); NVIC_EnableIRQ(EXTI0_IRQn);。

应用 CMSIS 到实验：

RTOS 编程：CMSIS-RTOS 接口可以更容易地在 Cortex-M 微控制器上实现多任务编程。

性能优化：CMSIS 提供的 DSP 和数学库可以帮助优化算法性能，特别是在需要数字信号处理的应用中。

第 5 题

1) 简述微内核和宏内核结构，并分别举例，以分析二者各自的优缺点。
2) 在 RTOS 中的常用任务调度有哪些？请举两例，并佐以 C 代码或者伪码描述，比较两种调度方法的特点与差异。
3) 为何会产生优先级翻转问题？请给出至少两种不同解决方法，并比较这些方法的特点与异同。

答：

1.

微内核是一种提供最小化服务的操作系统核心，通常只包括进程管理、线程管理、地址空间和中断处理等基本功能。其他服务如文件系统、网络协议栈等作为用户态的进程运行。

举例：MINIX、L4 微内核。嵌入式操作系统的内核一般是微内核。

宏内核将操作系统的所有服务都集成在核心空间运行，包括进程调度、内存管理、文件系统和设备驱动等。

举例：Linux、Windows NT 内核。

微内核相比宏内核具有松耦合的特点，模块独立性更高，便于生产和维护，具有更高的安全性，但切换开销更大；宏内核是紧耦合，具有更好的性能，稳定性和可维护性交叉。

2.

基于优先级的抢占式调度：

```
void DPS()//动态优先级调度dynamic priority scheduling
{
    PCB* process;
    printf("\n动态优先级调度算法开始执行:\n");

    while (Equeue != NULL)
    {
        // 把优先级最大的进程调整到队头
        take_MaxPrioOfEqueue();
        displayState();

        // 取出优先级最高的进程进行调度
        process = dequeue(&Equeue);
        process->state = 'R';  // 标记为运行态
        printf("\n调度进程: %s\n", process->name);
        process->cputime++;  // 运行一个时间片
        process->needtime--;  // 剩余时间减少
        process->prio--;  // 运行进程的优先级降低，也可以自己更改此处优先级降低的具体方式
        //还可以添加就绪进程优先级增加的代码段

        // 记录当前执行的进程
        LastExecuted = process;

        if (process->needtime > 0)
        {
            // 进程未完成，重新加入就绪队列
            process->state = 'E';  // 标记为就绪态
            enqueue(&Equeue, process);
        }
        else
        {
            // 进程完成，加入完成队列
            process->state = 'C';  // 标记为完成态
            enqueue(&Cqueue, process);
            printf("*******进程 %s 已完成*******\n", process->name);
        }
    }
}
```

轮转调度：

```
void RR()//时间片轮转调度round robin scheduling
{
    PCB* process;
    printf("\n时间片轮转调度算法开始执行:\n");

    while (Equeue != NULL)
    {
        displayState();
        process = dequeue(&Equeue);  // 从就绪队列取出进程
        process->state = 'R';  // 标记为运行态
        printf("\n调度进程: %s\n", process->name);
        process->cputime += process->round;  // 运行一个时间片
        process->needtime -= process->round;  // 剩余时间减少
        if (process->needtime > 0)
        {
            // 进程未完成，重新加入就绪队列
            process->state = 'E';  // 标记为就绪态
            enqueue(&Equeue, process);
        }
        else
        {
            // 进程完成，加入完成队列
            process->state = 'C';  // 标记为完成态
            enqueue(&Cqueue, process);
            printf("*******进程 %s 已完成*******\n", process->name);
        }
    }
}
```

轮转调度简单公平，但不适合实时性要求高的场景；优先级调度适合实时性要求高的场景，但可能导致低优先级任务饥饿。

3.

优先级翻转：资源共享与互斥导致高优先级任务阻塞。

解决方法：

优先级天花板：

优先级天花板协议为系统中的每个资源分配一个固定的优先级，这个优先级通常设置为系统中所有任务中最高的优先级加一。当一个任务占用了某个资源，它的优先级将被提升到该资源的优先级天花板。这样，即使有更高优先级的任务存在，占用资源的任务也不会被抢占。

特点：简单易实现，因为每个资源的优先级是固定的。可以保证资源的有序访问，避免优先级翻转。可能导致系统中其他任务的执行延迟，因为占用资源的任务可能长时间保持高优先级。

优先级继承：

优先级继承协议允许一个低优先级任务在持有资源时，如果被更高优先级的任务等待，那么低优先级任务的优先级将临时提升到等待它的最高优先级任务的优先级。一旦低优先级任务释放资源，它的优先级将恢复到原来的值。

特点：动态调整任务优先级，更加灵活。可以减少因优先级提升导致的系统延迟。实现复杂度较高，需要跟踪任务之间的等待关系。可能导致系统不稳定，如果不当使用，可能会引起优先级反转或优先级链问题。

比较：

实现复杂度：优先级天花板的实现相对简单，因为它只需要为资源分配固定的优先级。而优先级继承需要动态调整任务的优先级，实现更为复杂。

系统性能：优先级天花板可能会导致其他任务的执行延迟，因为它会长时间保持高优先级。优先级继承则可以减少这种延迟，因为它只在必要时提升优先级。

实时性保证：两者都能有效地解决优先级翻转问题，但优先级继承可能提供

更好的实时性保证，因为它允许任务在必要时快速响应。

适用场景：优先级天花板适用于资源访问频繁且资源数量较少的场景。优先级继承适用于任务间依赖关系复杂，且需要频繁访问资源的场景。