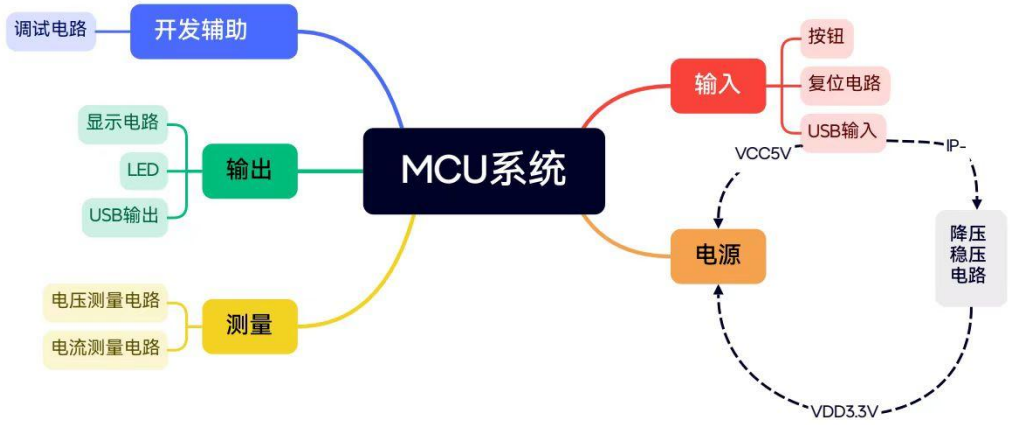
Question 1.1

1. 电路单元思维导图：



2. 分条称述见下;

MCU 系统

功能：作为整个系统的核心控制器，负责运行主程序、管理外设、处理数据、与上位机或其他模块通信等。

工作原理 ：STM8S003 负责核心逻辑控制，连接各功能模块。通过 VDD 和地引脚 VSS 供能；时钟电路可使用内部 RC 或外部晶振/有源晶振；复位、BOOT、SWD 等引脚用于系统启动模式选择与在线调试。

设计理由 ：高级 STM8 内核，具有低成本优势、低功耗设计，开发便捷，适用于资源需求有限、预算敏感且无需复杂运算的场景；MCU 集成设计有利于合理安排外设复用，以减少 PCB 体积、简化外围电路。

USB 输入输出模块

功能:从 USB 接口获取 5V 电源，为整个系统提供电源输入。

工作原理：USB 口的 5V 通过 100nF 电容器滤波输送到后级电源管理模块。

设计理由：USB 5V 供电常见且方便获取；100nF 电容器滤波可以提升系统稳定性与抗干扰能力。

降压稳压电路

功能: 调整输出电压 VDD 使其保持在设定值附近，从而抵消由于输入电压或负载变化产生的波动，来实现电路稳定的功能

工作原理： AMS1117-3.3 提供 3.3V 稳定电压，滤波电容抑制噪声。

设计理由：需要为 MCU 系统、显示电路、复位电路、LED 电路、按钮电路提供 3.3V 稳定工作电压，因此设计该部分电路。

电流测量电路

功能: 检测电流大小，以供后续计算功率之用。

工作原理：使用 CC6920BSO-5A 电流传感器以及后级的运算放大器组成的负反馈电路检测电流大小。

设计理由：CC6920 是一款高性能霍尔效应电流传感器，精度高、线性度好、性能稳定，可支持直流或交流电流的测量，适用于本项目电流检测单元。

电压测量电路

功能: 检测电压大小，以供后续计算功率之用。

工作原理：通过分压电路采集 1/2VCC 的大小，通过 UMES 端口输入 MCU 单元计算；使用 100nF 滤波电容抑制噪声。

设计理由：电路简单又能实现电压检测功能，节省 PCB 板空间

显示电路

功能: 输出功率等信息，实现人机交互。

工作原理：MCU 单元通过 SCL、SDA、RES、DC、CS 端口连接 OLED -096IN 屏幕实现数据传输与显示

设计理由 ：便于直观观察功率等信息，以实现人机交互功能。

LED

功能:用于扩展、检测、输出等场合

工作原理：VDD 供电，LED1\LED2 的下端与 MCU 的 PC6\PC7 端口相连，当端口输出低电平时 LED 导通，输出高电平时 LED 熄灭

设计理由：便于输出并直观观察 MCU 工作性能与状态（如 MOSI、MISO 等），可与按键等电路配合形成丰富的扩展功能。

按钮

功能: 检测是否被按下，按下时输入低电平信号。

工作原理： 当按钮 SW1、SW2 未被按下时，BUT1、BUT2 端口通过 R19 与 VDD 相连，向 MCU 的 PC4、PC5 端口输出高电平信号；当按钮 SW1、SW2 被按下时，BUT1、BUT2 端口接地，向 MCU 的 PC4、PC5 端口输出低电平信号。

设计理由：检测用户的输入行为，可与其他电路配合形成丰富的扩展功能。

复位电路

功能: 通过按钮实现 MCU 复位，切换 MCU 的工作状态。

工作原理： 当按钮 SW3 未被按下时，NRST 端口通过 R22 与 VDD 相连，向 MCU 输出高电平信号；当按钮 SW3 被按下时，NRST 端口直接接地，向 MCU 输出低电平信号。

设计理由 ：用于方便地切换 MCU 的工作状态。

调试电路

功能: 用于调试电路的工作状态，切换输入模式（SWIM）。

工作原理： PZ254V-11-04P 通过 VCC 供电，连接 SWIM 口与 MCU 的 NRST 端口，实现 STLINK 连接。

设计理由：STLINK 连接成本低廉，支持广泛，传输速度较快，调试功能强大，可以用于单步调试、断点调试、变量查看等功能，方便开发人员进行程序调试。

3．主控引脚复用模式：
PA1/PA2：复用为 UART 的 TX/RX，用于串口通信。
PB4/PB5：复用为 I2C 的 SCL/SDA，驱动 OLED 屏幕。
PC3/PC4：通用 GPIO，控制 LED 和按键输入。

Question 1.2
1. 两版工程差异**
1、第一版电源 USB 接口使用外界晶振 X2，第二版中删去，使用内部晶振；
2、第一版电源中有三个按钮,第二版中删去了 SW3,将调试电路原先闲置的的 RES 用于连接 MCU 的 PC3 端口，使电路调节更方便
3、第二版电流测量电路删去 R27 及其支路，将比较器+输入端口通过负载 R29、R31 接地，并将 R28 换成两个 33kΩ的负载；

2. 改进原因
1、项目对频率要求不高，不涉及串口通信和精确定时，用内部晶振能节省钱与 PCB 空间；

2、优化端口分配，让调试电路可以通过 STLINK 与 MCU 的 RES 端口连接，更方便调试
3、第一版中的电流测量电路通过 R27 形成电压并联反馈，与 R28 至-端口的电压串联反馈相互影响，会导致 R23 右端比较器输入电位不稳定，删去后将只剩下电压串联反馈，反馈电位稳定。

Question 1.3

第二版可改进点：

PCB 设计上：

1、PCB 布局中可以删去原理图中已经不存在的 SW3 ，节省空间；
2、PCB 布局中模拟信号走线应远离数字信号，减少串扰。
3、电流采样电阻附近可添加散热焊盘，避免长时间工作温漂。

电路上：

1、仅使用单级电容滤波，可以增加二阶 LC 滤波电路降低高频噪声；
2、USB 接口易受静电损坏，可以加入 TVS 二极管保护 USB 接口，提升可靠性；

Part 2：原理图与 PCB 设计

Question 2.1

1.改进方案

供电电路：替换 AMS1117 为 TPS7A4700（低噪声 LDO），添加 10μF 钽电容与 100nF 陶瓷电容组合滤波。

采集电路：使用 INA219 高精度电流传感器，集成 PGA 和 12 位 ADC，减少外部噪声。

二阶低通滤波电路：截止频率 **二阶低通滤波**：截止频率 fc =50Hz，选用 OPA2188 运放

2.选型依据

TPS7A4700：噪声密度 4.7μV RMS，优于 AMS1117（约 30μV RMS），能提供更干净的电源。

INA219：集成度高，支持 I2C 输出，减少布线复杂度。

3.成本估算(以每批 100 个计)

TPS7A4700：约 8 元/个

INA219：约 12 元/个

OPA2188：约 6 元/个

其他滤波电容、分压电阻、PCB、Type-C 接口、PD3.0 协议芯片及附加模块：约 11 元/个

总计：约 26 + 11 ≈ 37 元/个

Part 3

图 1: 焊接好的板子正反面

由于上过电子工程训练，所以对焊接并不太陌生，但是这次使用的刀型焊枪使用不太习惯。

焊接芯片时使用了拖焊的方式，有时会发生连焊，加上助焊剂会好很多。

Part 4

3.1 Hello,world

首先我们焊接之后得到单片机，通过 ST-Link 将 Vcc，Swim，GND，Nrst 与芯片的对应针脚用杜邦线连接，就可以在 iar 软件中进行烧录。

但我们在烧录的过程中出现了报错，我们的 iar 无法检测到 swim 端口，导致代码无法烧录。后来，通过查阅 PCB 线路图，发现 Vcd 为 3.3V，而 Vcc 为 5V，故重新将板子以 5V 电压供电，并将 iar 工程中的芯片型号，debug 工具设置为 stm8s003F3 与 stlink，并将 led 管脚设置为 pc6，终于将板子点亮。

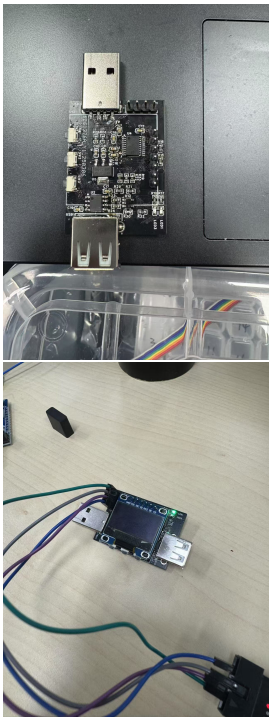


图 2: 焊接好的板子以及点亮的样子

后来我们放弃使用芯片专有的库来进行寄存器级别的开发，转而使用 stm8s.h 来进行开发，但是编译过程中一直无法找到库文件，后面查看了标程后把标准库文件下载后放入项目文件夹，并添加了预编译路径，勉强可以使用 stm8s.h 进行开发。

在串口通信方面，我使用 pip 进行串口通信的读取，pyserial，但是 python 无法读到端口，到结题时也无法解决。但我们把理论上能够实现的上位机 python 实现完成了。

但是我们成功实现了用按钮控制灯的亮灭

```c
/* Includes ------------------------------------------------------------------*/
#include "stm8s.h"

/* Private defines -----------------------------------------------------------*/
#define LED_G_GPIO_PORT GPIOC
#define LED_G_GPIO_PIN GPIO_PIN_6
#define BUTTON1_GPIO_PORT GPIOC
#define BUTTON1_GPIO_PIN GPIO_PIN_4
#define LED_R_GPIO_PORT GPIOC
#define LED_R_GPIO_PIN GPIO_PIN_7
#define BUTTON2_GPIO_PORT GPIOC
#define BUTTON2_GPIO_PIN GPIO_PIN_5

/* 按钮状态检测宏 */
#define BUTTON1_PRESSED() (GPIO_ReadInputPin(BUTTON1_GPIO_PORT, BUTTON1_GPIO_PIN) == RESET)
#define BUTTON2_PRESSED() (GPIO_ReadInputPin(BUTTON2_GPIO_PORT, BUTTON2_GPIO_PIN) == RESET)
#define DEBOUNCE_DELAY 1000 // 单位取决于Delay函数实现

void Delay(uint16_t nCount) {
  while(nCount--);
}

void main(void) {
  /* 初始化LED引脚: 推挽输出 */
  GPIO_Init(LED_G_GPIO_PORT, LED_G_GPIO_PIN, GPIO_MODE_OUT_PP_LOW_FAST);

  GPIO_Init(LED_R_GPIO_PORT, LED_R_GPIO_PIN, GPIO_MODE_OUT_PP_LOW_FAST);
  /* 初始化按钮引脚: 上拉输入（按钮按下接地） */
  GPIO_Init(BUTTON1_GPIO_PORT, BUTTON1_GPIO_PIN, GPIO_MODE_IN_PU_IT);
  GPIO_Init(BUTTON2_GPIO_PORT, BUTTON2_GPIO_PIN, GPIO_MODE_IN_PU_IT);
  while(1) {
    if(BUTTON1_PRESSED()) { // 检测按钮按下
      Delay(DEBOUNCE_DELAY); // 简单消抖
      if(BUTTON1_PRESSED()) { // 确认有效按下
        GPIO_WriteReverse(LED_G_GPIO_PORT, LED_G_GPIO_PIN); // 翻转LED状态
        while(BUTTON1_PRESSED()); // 等待按钮释放
      }
    }
    if(BUTTON2_PRESSED()) { // 检测按钮按下
      Delay(DEBOUNCE_DELAY); // 简单消抖
      if(BUTTON2_PRESSED()) { // 确认有效按下
        GPIO_WriteReverse(LED_R_GPIO_PORT, LED_R_GPIO_PIN); // 翻转LED状态
        while(BUTTON2_PRESSED()); // 等待按钮释放
      }
    }
  }
}
```

```
#ifdef USE_FULL_ASSERT

/**
  * @brief Reports the name of the source file and the source line number
  * where the assert_param error has occurred.
  * @param file: pointer to the source file name
  * @param line: assert_param error line source number
  * @retval : None
  */
void assert_failed(u8* file, u32 line)
{
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */

  /* Infinite loop */
  while (1)
  {
  }
}
#endif
```

其中推挽输出可能带动耗电量较大的原件，如继电器，发光二极管。

通过 GPIO 初始化管脚，其中管脚的地址都在 PCB 的原理图中可以找到。

需要设置按钮为上拉，此时按钮按下就会接地，这个信号可以被检测到用于控制灯。

然后通过在灯的脚上输出反转的电平，就可以实现灯的亮灭。

串口通信的思考理解：

芯片通过 uart 协议与上位机进行通信，stm8 的 tx 管脚在 PD5，rx 在 PD6。

```
port_name = "COM3"
ser = serial.Serial(
    port=port_name,
    baudrate=115200,
    bytesize=serial.EIGHTBITS,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    timeout=1
)
```

通过这块可以打开 COM3 串口并且读取文件。

3.2 串口实现

上位机数据处理与可视化: 思路:

1. 初始化阶段

加载配置文件，比如端口的参数、显示的样式等；初始化串口连接，将单片机与上位机进行串口的实现；

2. 数据采集阶段

   STM8 实时采集 USB 的电压与电流数据；

3. 数据处理阶段

   主要在 Python 中采用了 serial 和 Matplotlib 实时绘制折线图。

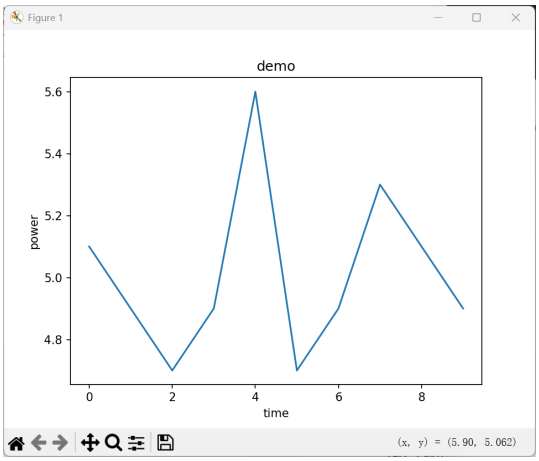
   计算功率值：$P = V \times I$

4. 界面展现阶段

   (a) 自动调整坐标轴范围保证数据可见，显示图形时横轴表示时间（最近 10 秒），纵轴显示三条曲线：电压、电流及功率，并定义了横轴和纵轴的名字

   (b) 更新状态栏实时数值显示

初始化阶段问题：最终只能展现一条线，无法呈现三条线



Bonus

Question 5.1：读取、写入 CUID 卡

一、学习路径：

1、书籍：

（1）《STM8 单片机自学笔记》

（2）《深入浅出 STM8 单片机入门、进阶与应用实例》

2、视频教程：

（1）youtube

SPI: The serial peripheral interface

PROTOCOLS: UART - I2C - SPI - Serial communications

（2）bilibili

深入理解 SPi 通讯协议，5 分钟看懂！（爱上半导体）

85，SPI-手把手写 SPI 配置源码（向量道科技）

二、题目理解和思路

1、硬件接口：STM8 和 PN5180 模块的连接

（1）STM8S105K4T6 SPI 引脚分配

SPI 信号	STM8 引脚	引脚名称	复用功能	GPIO 模式配置
SCK	PC5	SPI1_SCK	复用推挽输出	GPIO_MODE_OUT_PP_HIGH_FAST
MOSI	PC6	SPI1_MOSI	复用推挽输出	GPIO_MODE_OUT_PP_HIGH_FAST
MISO	PC7	SPI1_MISO	复用上拉输入	GPIO_MODE_IN_PU_NO_IT
NSS	PD3	SPI1_NSS	软件控制（普通 GPIO）	GPIO_MODE_OUT_PP_HIGH_FAST

（2）PN5180 模块连接

PN5180 引脚	STM8S105K4T6 引脚	功能说明
SCK	PC5	SPI 时钟信号
MOSI	PC6	主机输出从机输入
MISO	PC7	主机输入从机输出
NSS	PD3	片选信号（低电平有效）
RST	PD4	PN5180 复位引脚
IRQ	PD2	PN5180 中断引脚（可选）
VCC	3.3V	电源（需与 STM8 共地）
GND	GND	地线

2、编程实现：

（1）文件引入：Stm8 库文件的导入（stm8s.h、stm8s_clk.h、stm8s_conf.h、stm8s_spi.h 等）

（2）宏定义：区块数据长度、扇区默认设置等

（3）SPI 初始化：设置 SPI 的各种参数，如数据传输顺序、波特率、工作模式等。

（4）扇区和区块进行认证操作

指定密钥类型 keytype、区块编号 block 和 DEFAULT_KEY 构造认证指令，再通过 for 循环把 cmd 中的每个字节通过 spi 接口发送给 Mifare 卡，返回值为 0 时表示认证成功。

（5）数据读写操作

1、SPI_Transfer 函数用以实现实现 SPI 单字节数据的发送与接收，当 SPI_FLAG_TXE 标志置位时，即 SPI 发送缓冲区没数据时，发送数据；当 SPI_FLAG_RXNE 标志置位时，即接收缓冲区有数据时，最后接收并返回数据；ReadBlock 函数用以实现读取，每次调用 SPI_Transfer 函数发送读取命令和区块号，然后再循环 16 次，利用 SPI_Transfer 把读取的数据存放到 buffer 数组里面；WriteBlock 函数用以写入数据，每次发送写入命令和区块号，再 for 循环 16 次写入数据到 data。

2、功能一函数 ReadCUID：读取 CUID 卡的区块数据，通过调用 ReadBlock 来实现数据点读入，这里我添加了一个 for 循环来验证区块 1 是否为按位取反备份，如果不是的话，就输出 Error，结束程序；

3、功能二函数 WriteNamesToCUID：写入姓名数据，创建 blockData 数组，初始化为 0，用来匹配单个区块的存储大小；第一个姓名用 memcpy 复制写入前 8 个字节，多余的用 for 循环补 0，最后再用 WriteBlock 依次写入三个区块的数据。

（6）main 函数执行

一开始先是 SPI 初始化操作以及时钟配置，再调用 ReadCUID 和 WriteNamesToCUID 实现读取和写入 CUID 卡的两个功能。

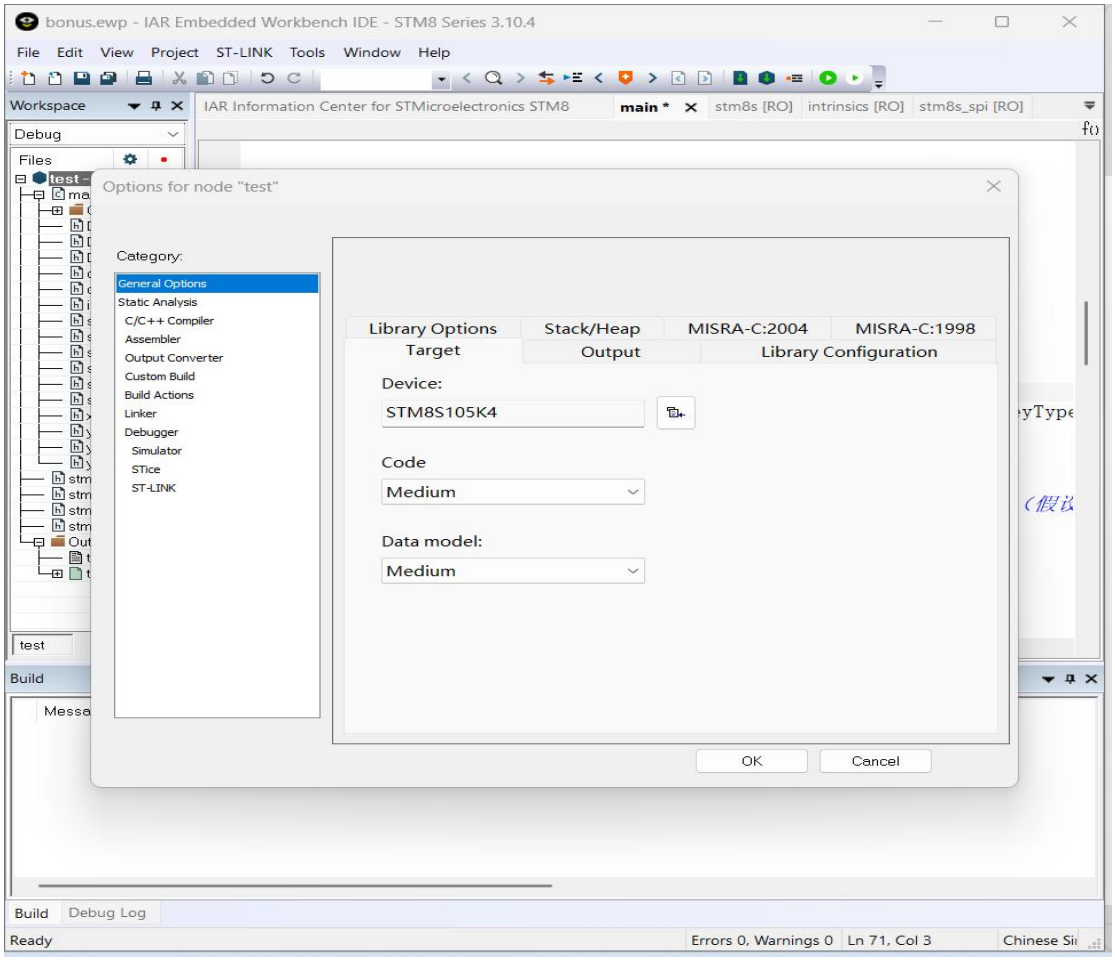
三、遇到的困难及解决策略

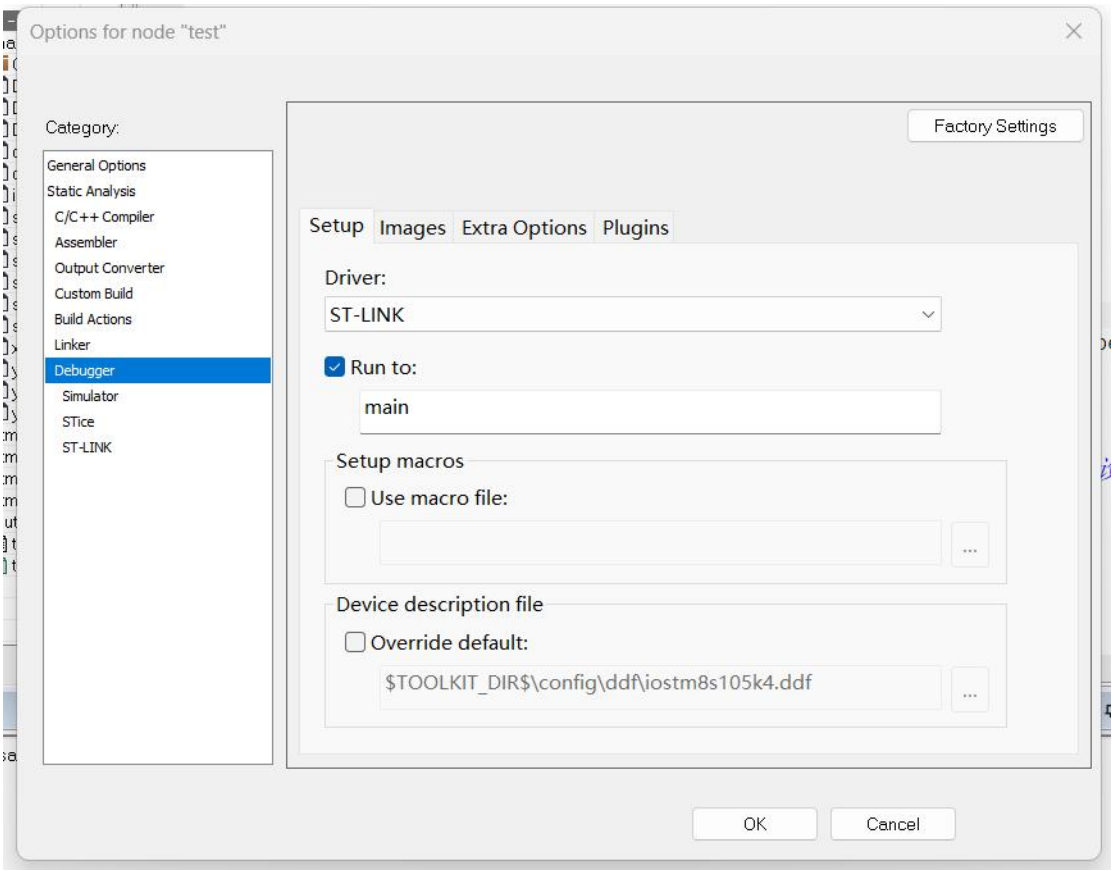
1、IAR 的操作

主要是前置操作：比如芯片型号的配置、编译时出现的各种错误（以头文件部分为主）

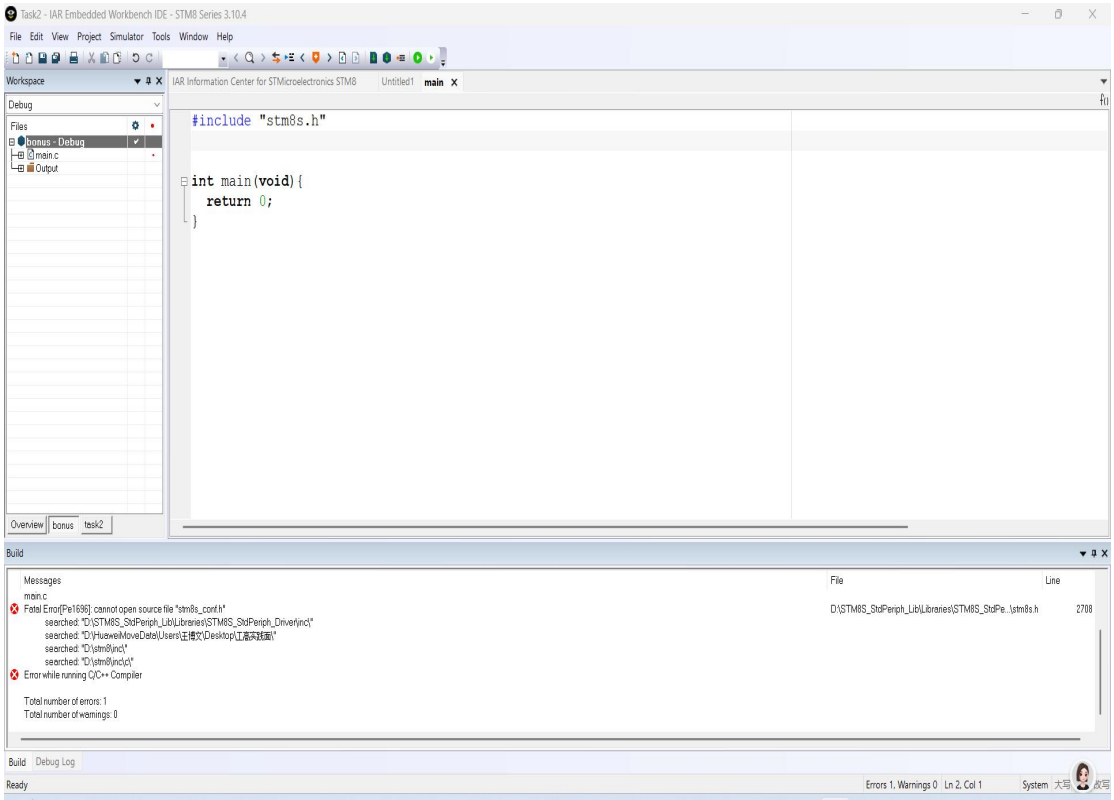
解决策略：主要是 CSDN、附件和大模型以及《STM8 单片机自学笔记》

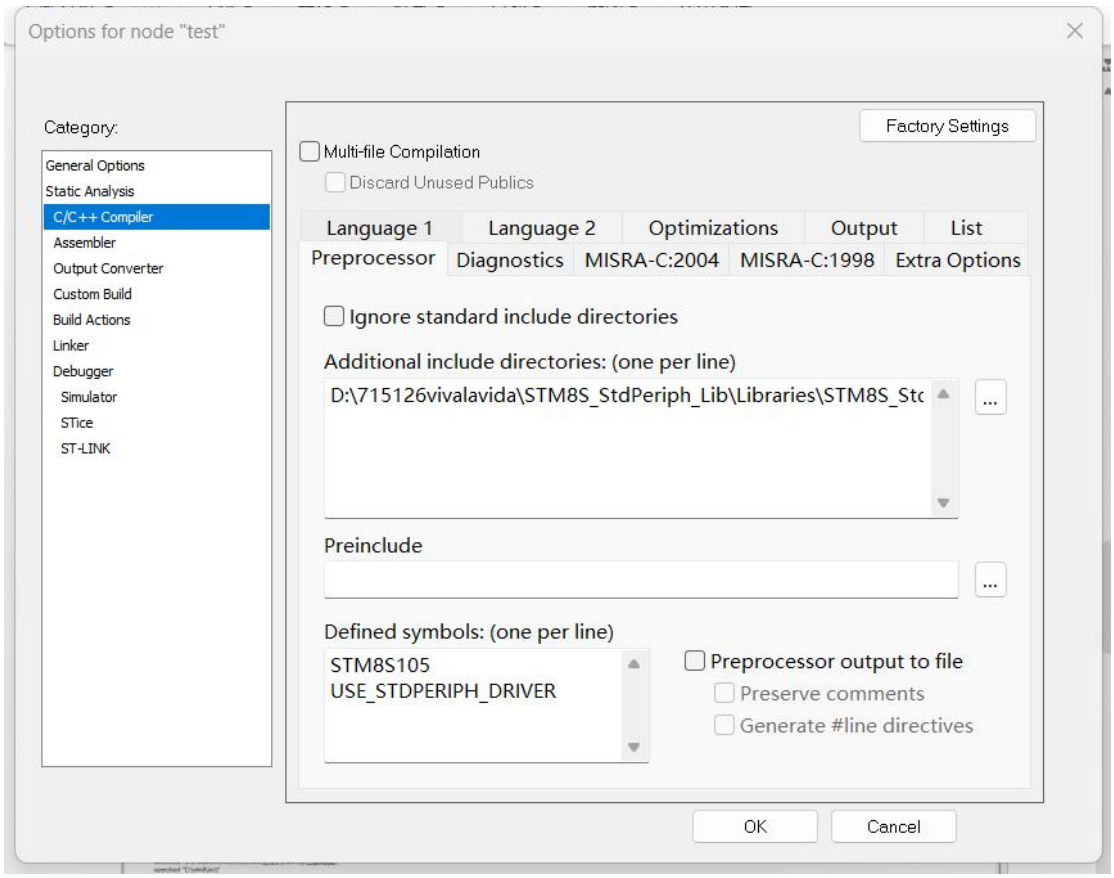
提供的信息：前期主要是都在问大模型 IAR 中的一些操作：比如 options 中芯片型号的确定、debugger 中 Driver 的确定；





对于编译时出现的头文件错误：这里主要是“no such file”或“重定义”的问题，但在大模型和搜索引擎获取信息后，发现是自己没有添加 stm8 的库文件，后面去官网下载了整个库文件，在 IAR 的 C/C++ Compiler 的 Preprocessor 里面添加了文件路径和芯片型号后，总算在试验代码（只有一个空的 main 函数）中解决了问题。





2、代码的实现：一开始先在 SPI 通讯协议上花了很多时间，主要是为了搞清楚具体的工作模式，这里获取的信息主要参

考 b 站和附件，关于代码中具体的函数主要是借助 AI，比如具体的发送和接受字节的函数、宏定义以及 SPI 初始化的函数等；在 IAR 中编译时出现的代码错误也以借助 AI 为主，主要是代码的一些细节上的书写问题为主。

我现在要解决B题的bonus的Question 5.1：读取、写入 CUID 卡
本题要求编写一个适用于 STM8S105K4T6 最小系统板的 IAR 工程，通过 SPI 与 PN5180 模块通信，使用给定的密钥与数据 CUID 卡进行交互。那么根据 PN5180 实际命令，读取与写入命令的代码：#define READ_CMD 0x30
#define WRITE_CMD 0xA0
这样有问题么

已深度思考（用时 29 秒）

根据PN5180的文档及Mifare Classic协议，读取和写入命令的定义需要结合具体操作流程进行调整：

问题分析：

1. **读命令0x30**：符合Mifare Classic标准读指令（对应读取单个区块），但需注意：
   - 需先通过`PN5180`发送认证命令（如`MIFARE_Authent`），使用密钥验证目标扇区。
   - 读命令格式应为：`0x30 + 区块地址`，并需附加CRC校验（部分库可能自动处理）。
2. **写命令0xA0**：需修正为**0xA0或0xA2**：
   - `0xA0`用于写入已认证的区块（需配合认证流程）。
   - `0xA2`用于未认证的区块（部分场景可能无效）。
   - 实际写入需确保认证成功，且控制位允许写入。

关键代码修正建议：

c 复制

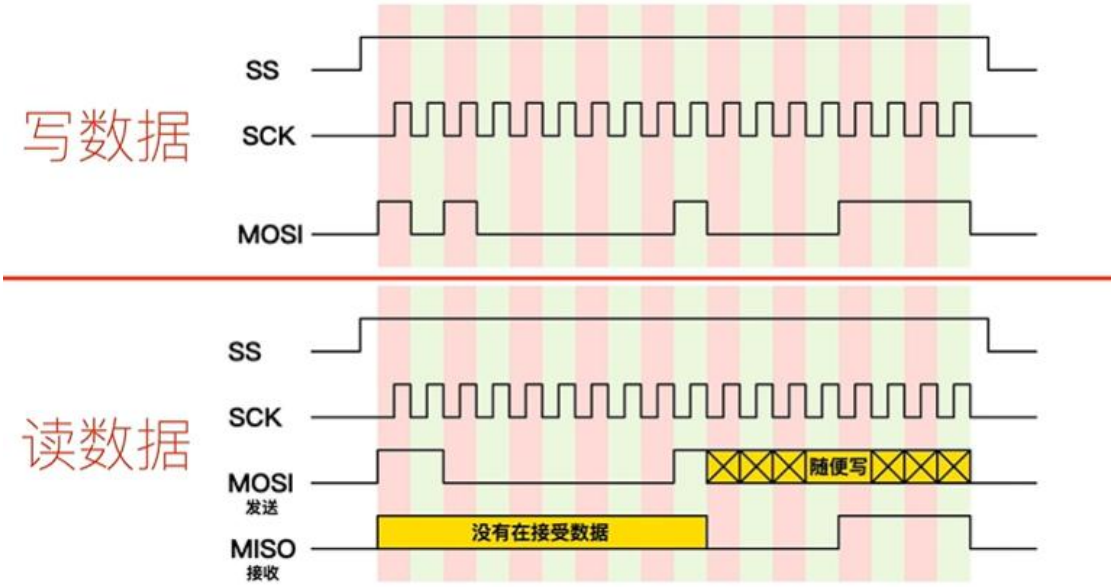
```c
// 认证命令（假设密钥已加载到Key Slot 0）
uint8_t auth_cmd[] = {0x40, 0x01, 0x00, 0x00}; // 使用密钥槽0认证区块0
send_spi_command(auth_cmd, sizeof(auth_cmd));

// 读命令（区块地址需转为1字节，如区块0：0x00）
#define READ_CMD  0x30
uint8_t read_block0[] = {READ_CMD, 0x00};
send_spi_command(read_block0, sizeof(read_block0));

// 写命令（需认证后）
#define WRITE_CMD 0xA0
uint8_t write_block0[] = {WRITE_CMD, 0x00, 0x11, 0x22, ..., 0xFF}; // 数据+CRC
send_spi_command(write_block0, sizeof(write_block0));
```

注意事项：

- **密钥加载**：需先通过`LoadKey`命令将默认密钥`FF FF FF FF FF FF`加载到PN5180的密钥存储区。
- **SPI时序**：确保STM8的SPI时钟速率与PN5180兼容（建议≤1MHz初始调试）。
- **CRC校验**：部分操作需手动添加CRC-A校验（可调用PN5180内置CRC功能简化流程）。
- **错误重试**：加入超时机制和重试逻辑，避免通信失败导致死锁。



b 站中 SPI 通信协议教程的截图

3、未尽的事业：唯一很可惜的就是一直想把代码烧录进单片机，但是苦于编译时出现的问题（主要是头文件报错），过程中一直在搜索信息（主要是问 ai 报错的原因，但得到的结果一直是细枝末节的“非根本性解决办法”，比如验证宏定义是否生效、在编译器中定义芯片型号宏等），由于这块耗时过长，只能先优化代码，先解决其他问题了，现在回想起来还是很可惜。

四、心得感悟

做 bonus 这块其实耗时花了很久，由于 SPI 的通信从未接触过，所以一开始背景知识花了很久时间，但是整体上收获特别多，主要的困难都是头文件引用部分，现在想想感觉还是特别可惜，总体上而言虽然最终没有获取卡里的数据，但是

还算形成较为完整的思路，等之后有时间了再来一战吧！