DMA相关:

1. 防止终端传输过半需要在HAL\_UARTEx\_ReiceiveToIdle\_DMA后面一行添加: \_\_HAL\_DMA\_DISABLE\_IT(&huart,DMA\_IT\_HT)
2. HAL\_UARTEx\_ReiceiveToIdle\_DMA的回调函数是RxEventCallback(&huart,size) 此处size是缓冲区最大容量

GPIO:

HAL\_GPIO\_WritePin(端口, 引脚号, 电平状态); // 写引脚电平

HAL\_GPIO\_ReadPin(端口, 引脚号); // 读引脚电平

HAL\_GPIO\_TogglePin(端口, 引脚号); // 翻转引脚电平

UART:

HAL\_UART\_Transmit(&串口句柄, 数据指针, 数据长度, 超时时间); // 串口发送（阻塞方式）

HAL\_UART\_Receive(&串口句柄, 数据指针, 数据长度, 超时时间); // 串口接收（阻塞方式）

HAL\_UART\_Transmit\_IT(&串口句柄, 数据指针, 数据长度); // 串口发送（中断方式）

HAL\_UART\_Receive\_IT(&串口句柄, 数据指针, 数据长度); // 串口接收（中断方式）

HAL\_UART\_Transmit\_DMA(&串口句柄, 数据指针, 数据长度); // 串口发送（DMA方式）

HAL\_UART\_Receive\_DMA(&串口句柄, 数据指针, 数据长度); // 串口接收（DMA方式）

IIC:

HAL\_I2C\_Master\_Transmit(&I2C句柄, 设备地址, 数据指针, 数据长度, 超时时间); // 主机发送数据（阻塞方式）

HAL\_I2C\_Master\_Receive(&I2C句柄, 设备地址, 数据指针, 数据长度, 超时时间); // 主机接收数据（阻塞方式）

HAL\_I2C\_Master\_Transmit\_IT(&I2C句柄, 设备地址, 数据指针, 数据长度); // 主机发送数据（中断方式）

HAL\_I2C\_Master\_Receive\_IT(&I2C句柄, 设备地址, 数据指针, 数据长度); // 主机接收数据（中断方式）

HAL\_I2C\_Master\_Transmit\_DMA(&I2C句柄, 设备地址, 数据指针, 数据长度); // 主机发送数据（DMA方式）

HAL\_I2C\_Master\_Receive\_DMA(&I2C句柄, 设备地址, 数据指针, 数据长度); // 主机接收数据（DMA方式）

SPI:

HAL\_SPI\_Transmit(&SPI句柄, 数据指针, 数据长度, 超时时间); // 发送数据（阻塞方式）

HAL\_SPI\_Receive(&SPI句柄, 数据指针, 数据长度, 超时时间); // 接收数据（阻塞方式）

HAL\_SPI\_TransmitReceive(&SPI句柄, 发送指针, 接收指针, 数据长度, 超时时间); // 同时收发（阻塞方式）

HAL\_SPI\_Transmit\_IT(&SPI句柄, 数据指针, 数据长度); // 发送数据（中断方式）

HAL\_SPI\_Receive\_IT(&SPI句柄, 数据指针, 数据长度); // 接收数据（中断方式）

HAL\_SPI\_TransmitReceive\_IT(&SPI句柄, 发送指针, 接收指针, 数据长度); // 同时收发（中断方式）

HAL\_SPI\_Transmit\_DMA(&SPI句柄, 数据指针, 数据长度); // 发送数据（DMA方式）

HAL\_SPI\_Receive\_DMA(&SPI句柄, 数据指针, 数据长度); // 接收数据（DMA方式）

HAL\_SPI\_TransmitReceive\_DMA(&SPI句柄, 发送指针, 接收指针, 数据长度); // 同时收发（DMA方式）

定时器:

HAL\_TIM\_Base\_Start(&定时器句柄); // 启动基本定时器（不带中断）

HAL\_TIM\_Base\_Start\_IT(&定时器句柄); // 启动基本定时器（中断方式）

HAL\_TIM\_Base\_Stop(&定时器句柄); // 停止基本定时器（不带中断）

HAL\_TIM\_Base\_Stop\_IT(&定时器句柄); // 停止基本定时器（中断方式）

HAL\_TIM\_PWM\_Start(&定时器句柄, 通道); // 启动PWM输出

HAL\_TIM\_PWM\_Stop(&定时器句柄, 通道); // 停止PWM输出

HAL\_TIM\_IC\_Start(&定时器句柄, 通道); // 启动输入捕获

HAL\_TIM\_IC\_Stop(&定时器句柄, 通道); // 停止输入捕获

HAL\_TIM\_Encoder\_Start(&定时器句柄, 通道组合); // 启动编码器模式

HAL\_TIM\_Encoder\_Stop(&定时器句柄, 通道组合); // 停止编码器模式

ADC:

HAL\_ADC\_Start(&ADC句柄); // 启动ADC转换（阻塞方式）

HAL\_ADC\_Stop(&ADC句柄); // 停止ADC转换

HAL\_ADC\_Start\_IT(&ADC句柄); // 启动ADC转换（中断方式）

HAL\_ADC\_Stop\_IT(&ADC句柄); // 停止ADC转换（中断方式）

HAL\_ADC\_Start\_DMA(&ADC句柄, 数据指针, 长度); // 启动ADC转换（DMA方式）

HAL\_ADC\_Stop\_DMA(&ADC句柄); // 停止ADC转换（DMA方式）

HAL\_ADC\_PollForConversion(&ADC句柄, 超时时间); // 等待ADC转换完成（阻塞轮询）

HAL\_ADC\_GetValue(&ADC句柄); // 获取ADC采样值

中断/事件回调函数:

//启用dma传输完成中断:

\_\_HAL\_DMA\_ENABLE\_IT(&hdma,DMA\_IT\_TC);

//启用dma传输

HAL\_DMA\_Start\_IT(&hdma,src\_addr,dest\_addr,data\_len);

// 使能 UART 接收中断（RXNE 中断） 一次一字节 HAL\_UART\_Receive\_IT(&huart1, rx\_buf, 1);

// 使能定时器更新中断

HAL\_TIM\_Base\_Start\_IT(&htim2);

// 使能 ADC 转换完成中断

HAL\_ADC\_Start\_IT(&hadc1);

HAL\_GPIO\_EXTI\_Callback(uint16\_t 引脚号); // 外部中断回调（GPIO口触发）

HAL\_UART\_TxCpltCallback(UART\_HandleTypeDef \*串口句柄); // UART发送完成回调（中断/DMA）

HAL\_UART\_RxCpltCallback(UART\_HandleTypeDef \*串口句柄); // UART接收完成回调（中断/DMA）

HAL\_UARTEx\_RxEventCallback(UART\_HandleTypeDef \*串口句柄, uint16\_t 数据长度); // UART接收事件回调（如IDLE触发接收长度）

HAL\_I2C\_MasterTxCpltCallback(I2C\_HandleTypeDef \*I2C句柄); // I2C主机发送完成回调（中断/DMA）

HAL\_I2C\_MasterRxCpltCallback(I2C\_HandleTypeDef \*I2C句柄); // I2C主机接收完成回调（中断/DMA）

HAL\_SPI\_TxCpltCallback(SPI\_HandleTypeDef \*SPI句柄); // SPI发送完成回调（中断/DMA）

HAL\_SPI\_RxCpltCallback(SPI\_HandleTypeDef \*SPI句柄); // SPI接收完成回调（中断/DMA）

HAL\_SPI\_TxRxCpltCallback(SPI\_HandleTypeDef \*SPI句柄); // SPI同时收发完成回调（中断/DMA）

HAL\_ADC\_ConvCpltCallback(ADC\_HandleTypeDef \*ADC句柄); // ADC转换完成回调（阻塞/中断/DMA）

HAL\_TIM\_PeriodElapsedCallback(TIM\_HandleTypeDef \*定时器句柄); // 定时器周期溢出回调

HAL\_TIM\_IC\_CaptureCallback(TIM\_HandleTypeDef \*定时器句柄); // 定时器输入捕获完成回调

HAL\_TIM\_PWM\_PulseFinishedCallback(TIM\_HandleTypeDef \*定时器句柄); // PWM脉冲完成回调

启用freertos功能的宏定义

队列:

#define configUSE\_QUEUES 1 启用队列功能

#define configQUEUE\_REGITRY\_SIZE 10 队列注册表大小

互斥锁:

#define configUSE\_MUTEXES 1 启用互斥锁

#define configUSE\_RECURSIVE\_MUTEXES 1 启用递归锁

#define configUSE\_PRIORITY\_INHERITANCE 1 启用优先级 继承

任务通知:

#define configUSE\_TASK\_NOTIFICATION 1 启用任务通知

#define configTASK\_NOTIFICATION\_ARRAY\_ENTRIES 1 每个 任务的通知组大小

事件组:

#define configUSE\_EVENT\_GROUPS 1 启用事件组

软件通知:  
 #define configUSE\_TIMERS 1 创建定时器服务服务

内存分配:

#define configSUPPORT\_DYNAMIC\_ALLOCATION 1 动态(默 认1)

#define configSUPPORT\_STATIC\_ALLOCATION 1 启用静态 (默认1)

配置堆栈大小

#define configTOTAL\_HEAP\_SIZE (10 \* 1024) 系统堆大小

#define configMINIMAL\_STACK\_SIZE 64 空闲任务

#define configMINIMAL\_STACK\_SIZE 64 最小堆栈

CAN:

HAL\_CAN\_ActivateNotification的第二个参数可用宏定义:

CAN\_IT\_RX\_FIFO0\_MSG\_PENDING:fifo接收到新消息触发

CAN\_IT\_RX\_FIFO0\_FULL:fifo中消息达到最大容量

CAN\_IT\_RX\_FIFO0\_OVERRUN:fifo已满仍有新消息,导致数据丢 失时触发

CAN\_IT\_TX\_MAILBOX\_EMPTY:邮箱发送完成时触发

任务优先级宏:

osPriorityNormal : 中等优先级

临界区函数:

UBaseType\_t uxStatus;

uxStatus = taskENTER\_CRITICAL(); // 进入临界区，禁用任务切换

taskEXIT\_CRITICAL(uxStatus); // 退出临界区，恢复任务切换

//可以不用参数

中断临界区函数:

UBaseType\_t uxSavedInterruptStatus; //创建变量

uxSavedInterruptStatus = taskENTER\_CRITICAL\_FROM\_ISR();

// 临界区代码

taskEXIT\_CRITICAL\_FROM\_ISR(uxSavedInterruptStatus);

//调度挂起:

vTaskSuspendAll(); // 挂起调度器

// 临界区代码

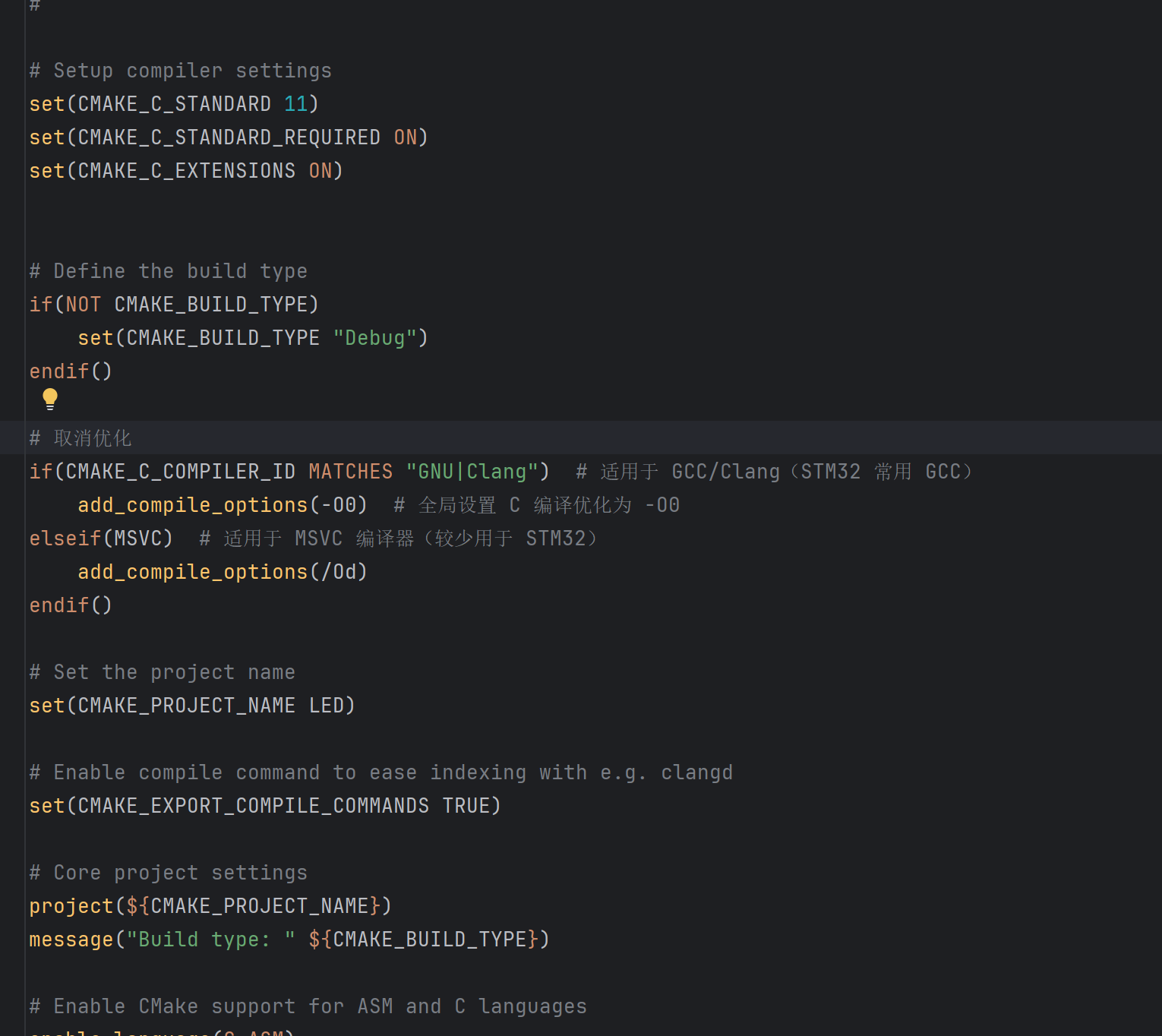
counter++;

xTaskResumeAll(); // 恢复调度器

## 注意临界区和调度器的区别:

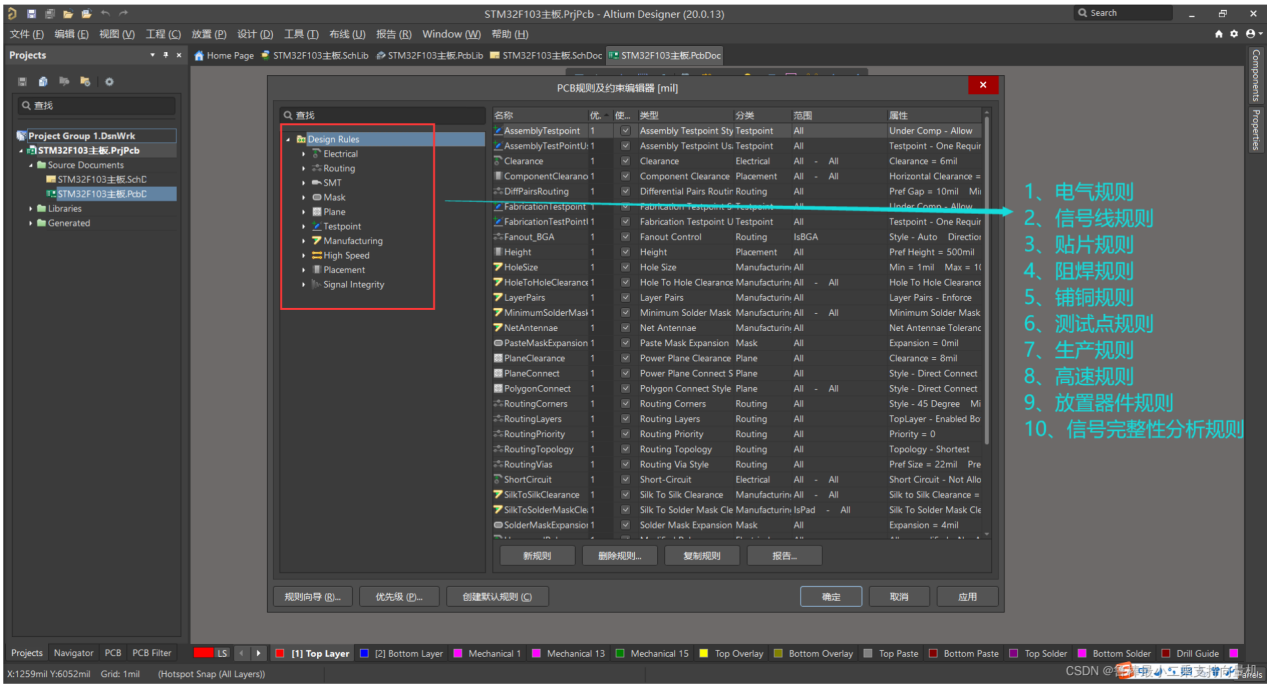
临界区会禁止中断,调度器不会

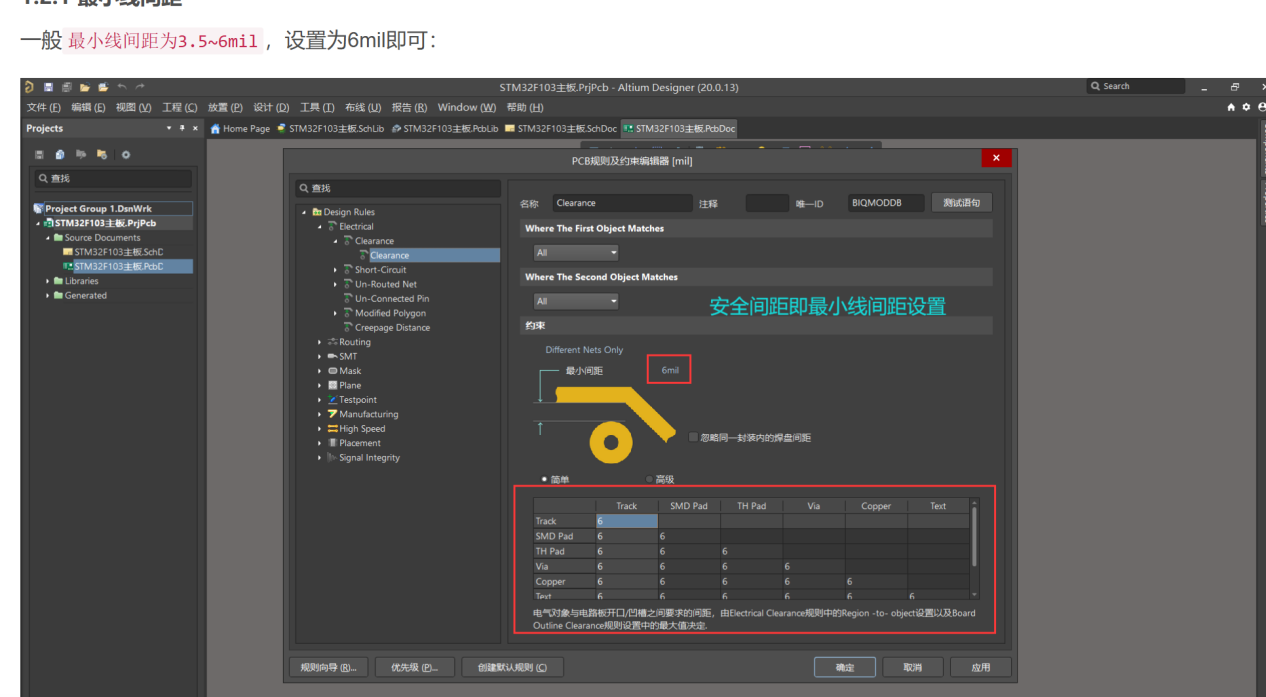
在CLion中取消优化

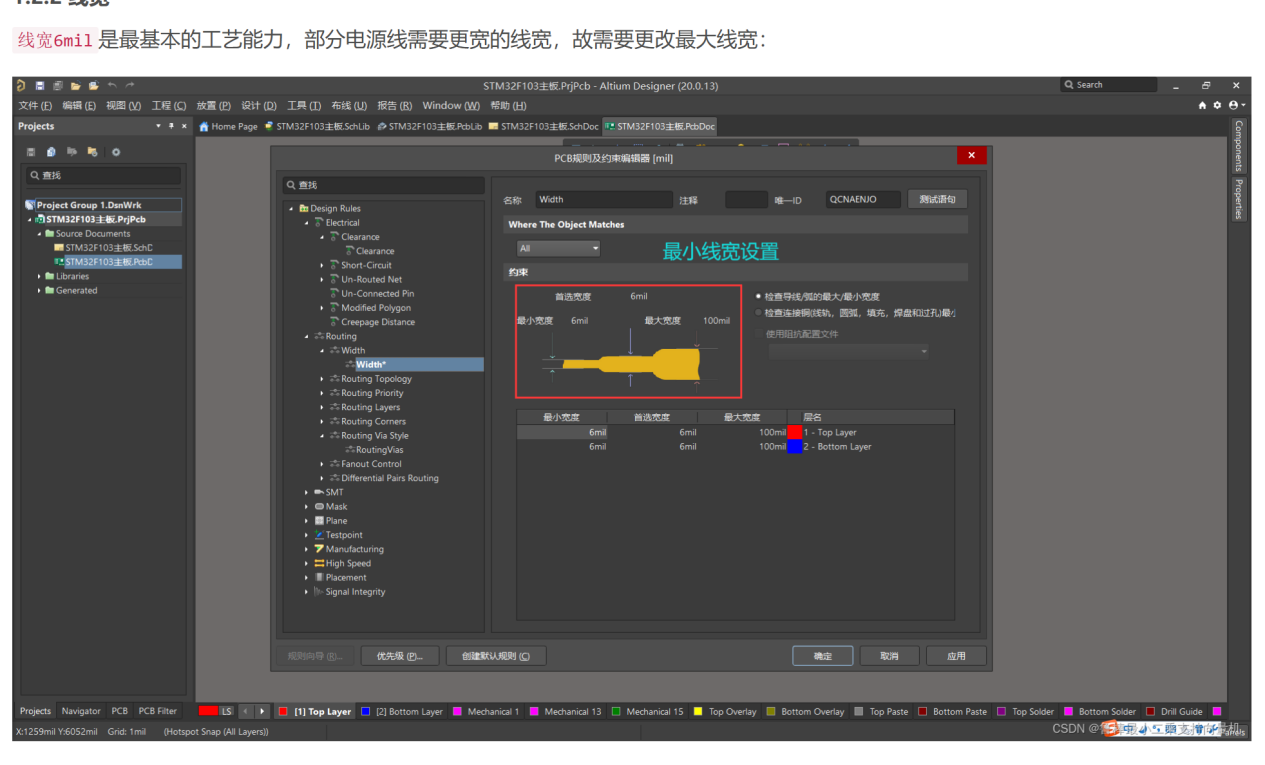


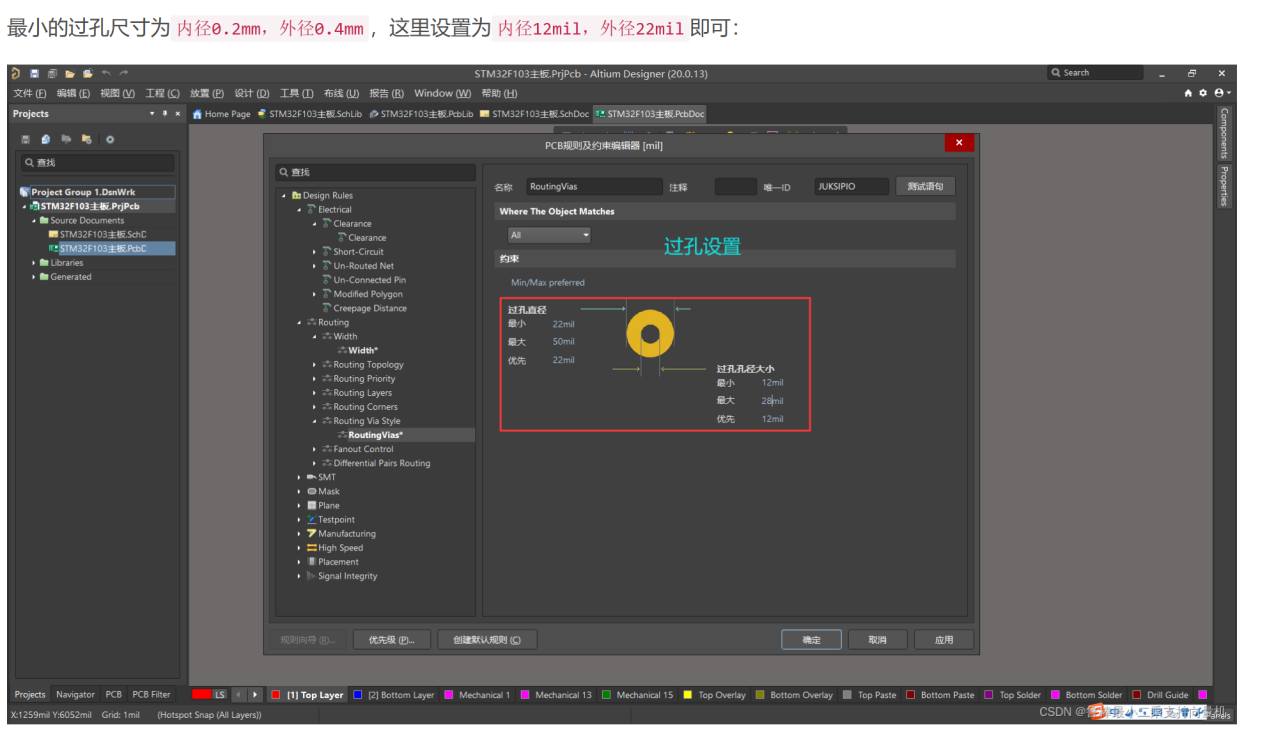
# 禁用自动优化 - 添加的配置

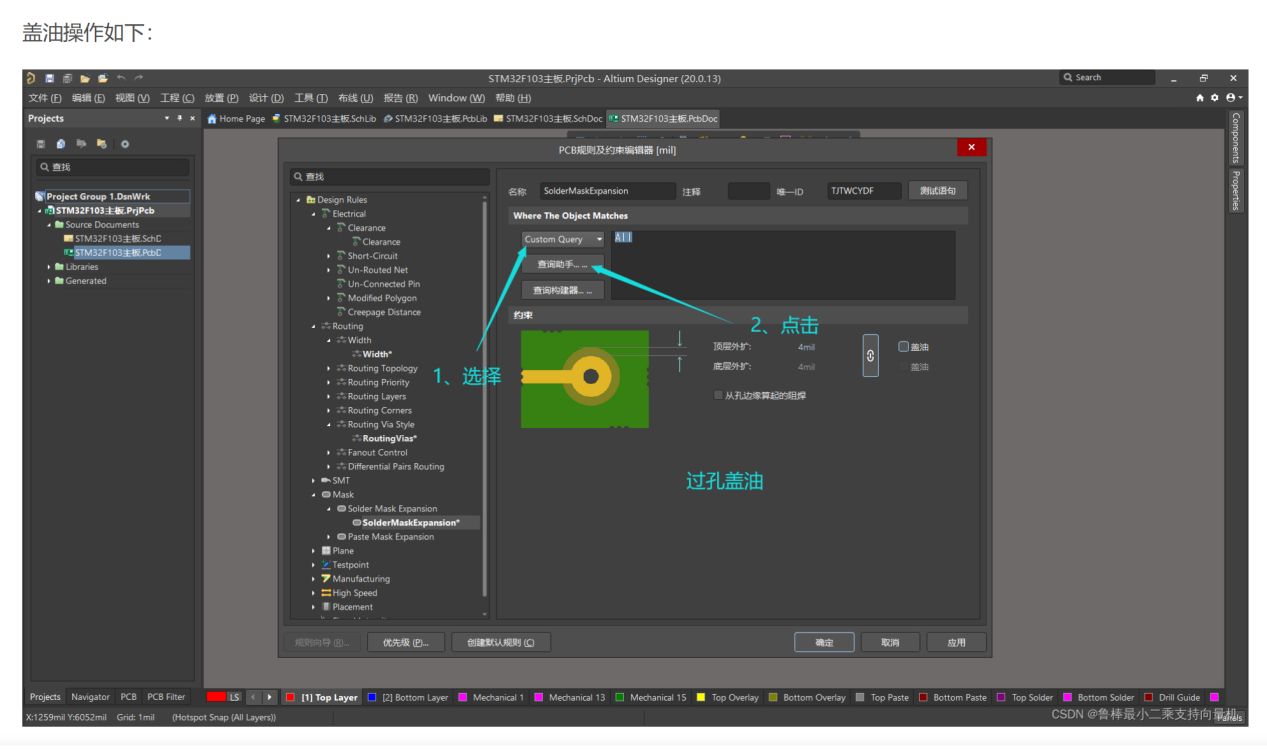
# 取消优化  
if(CMAKE\_C\_COMPILER\_ID MATCHES "GNU|Clang") # 适用于 GCC/Clang（STM32 常用 GCC）  
 add\_compile\_options(-O0) # 全局设置 C 编译优化为 -O0  
elseif(MSVC) # 适用于 MSVC 编译器（较少用于 STM32）  
 add\_compile\_options(/Od)  
endif()

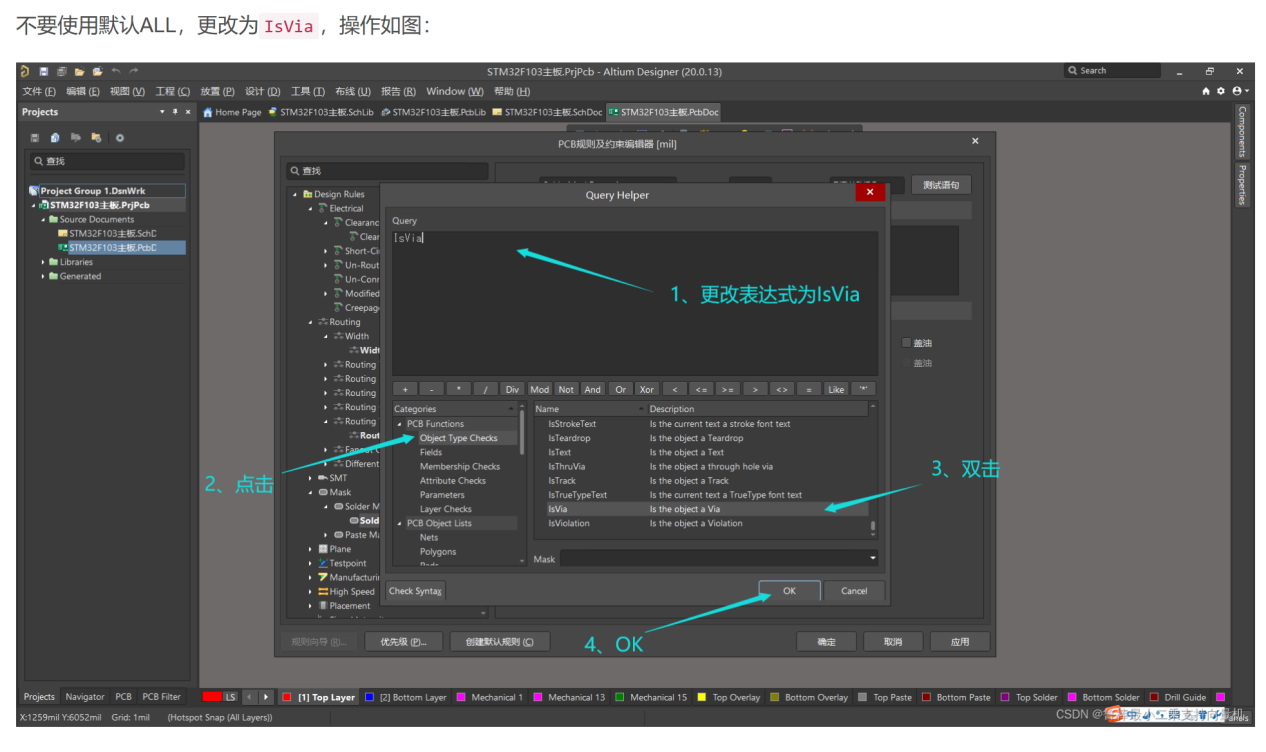


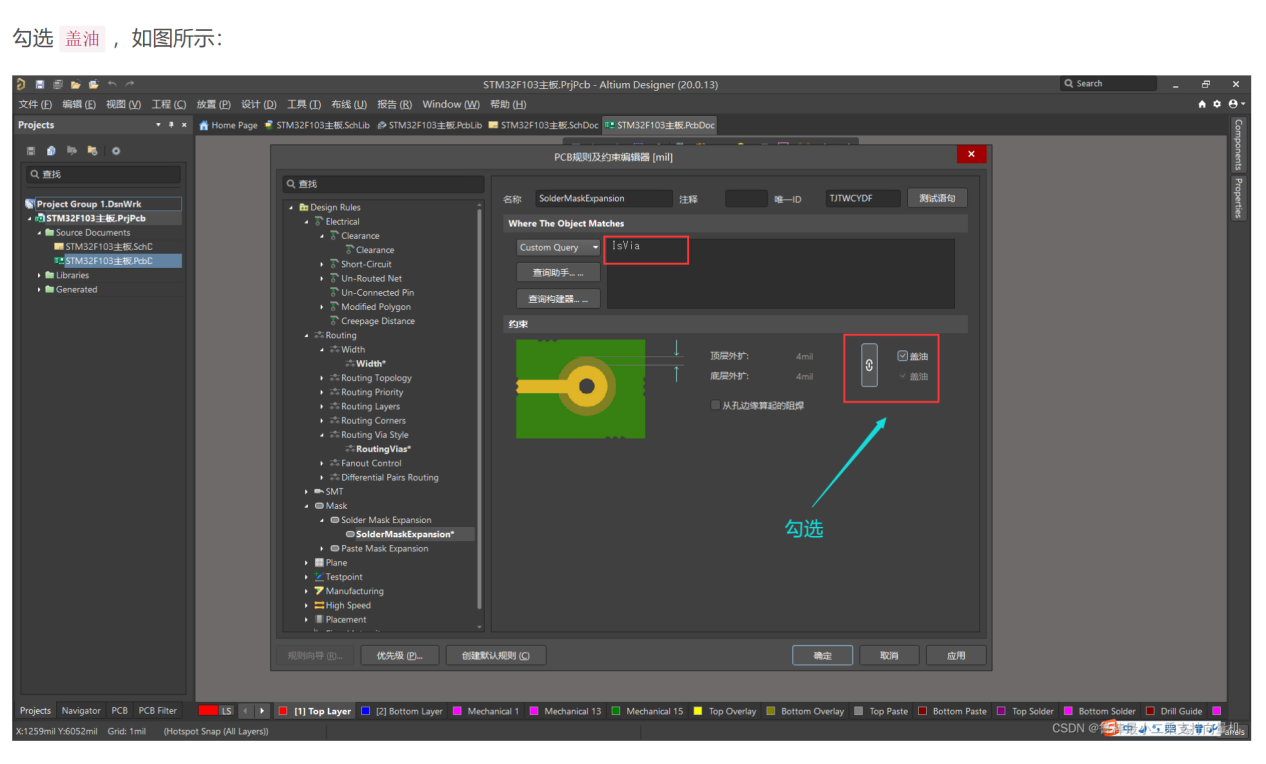


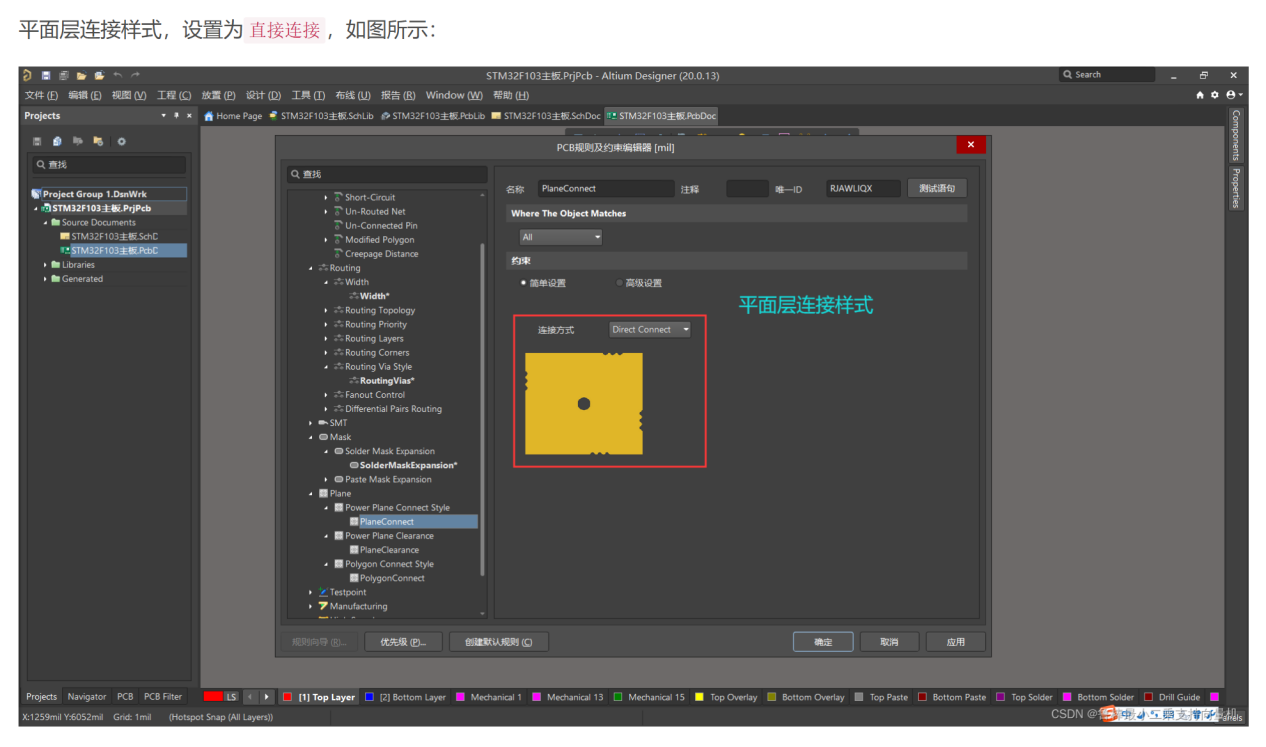


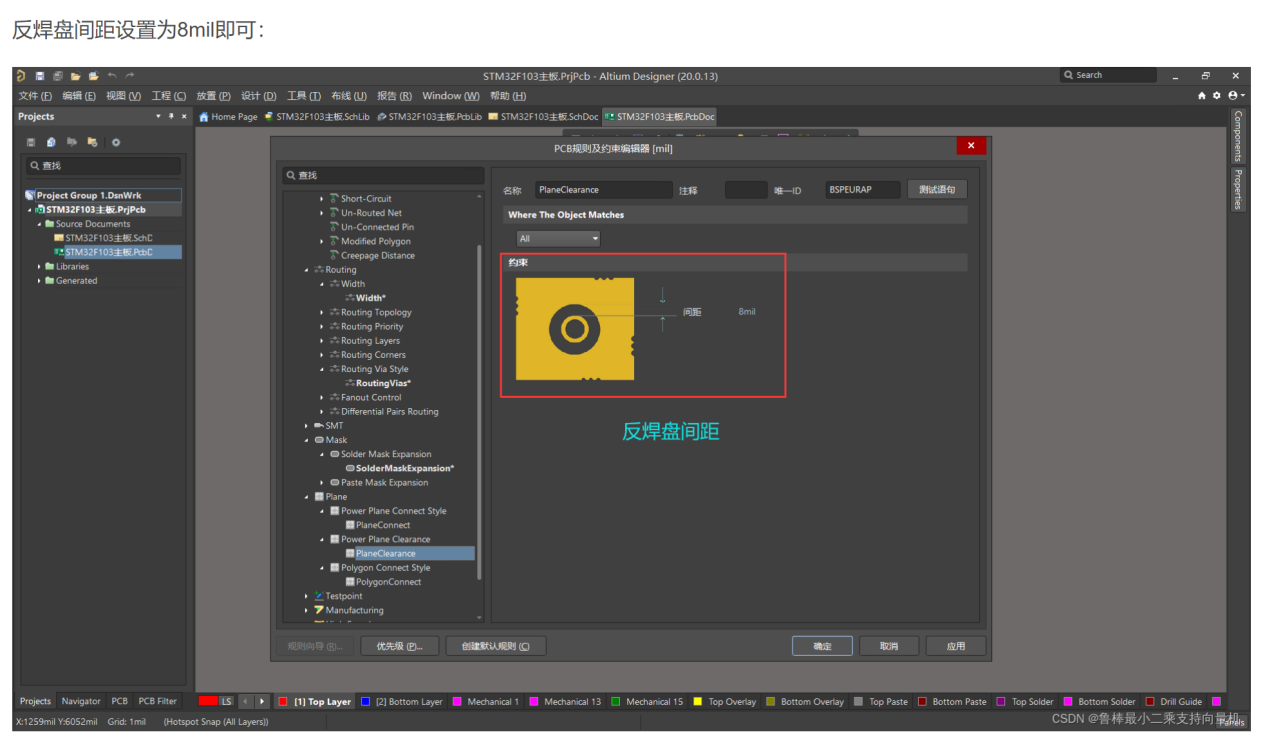


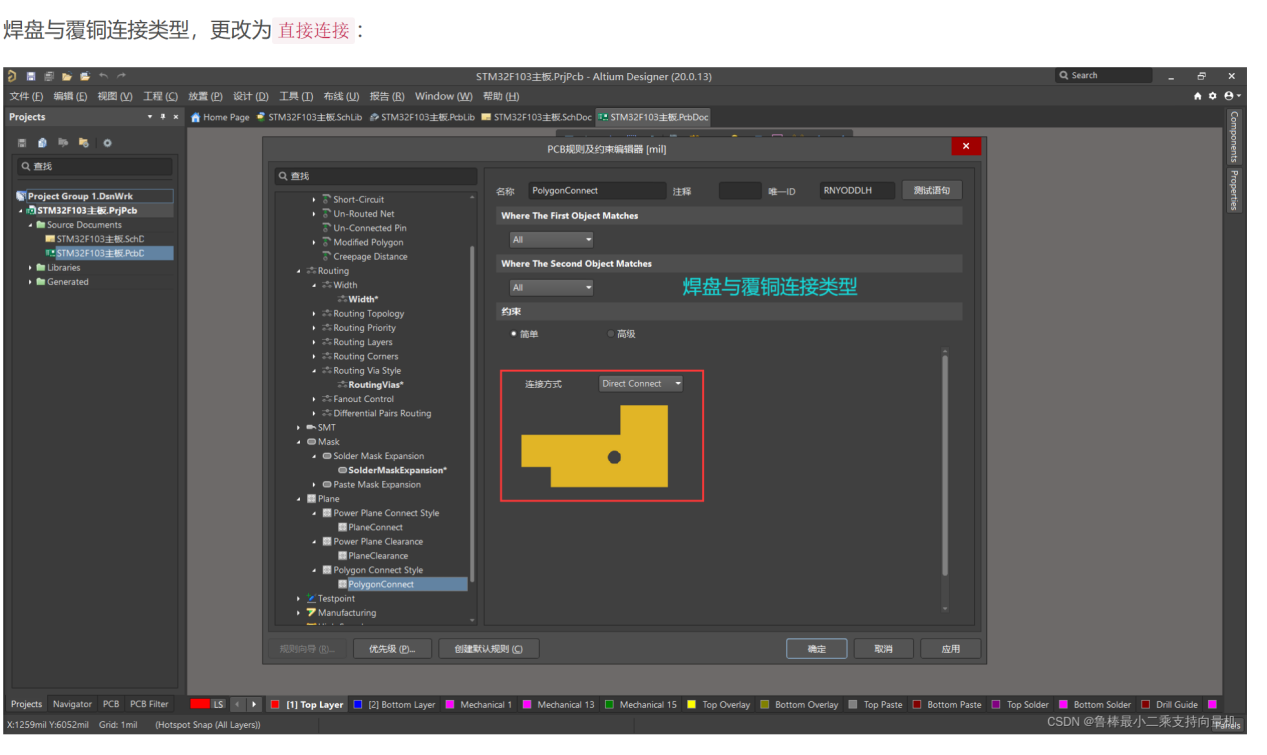


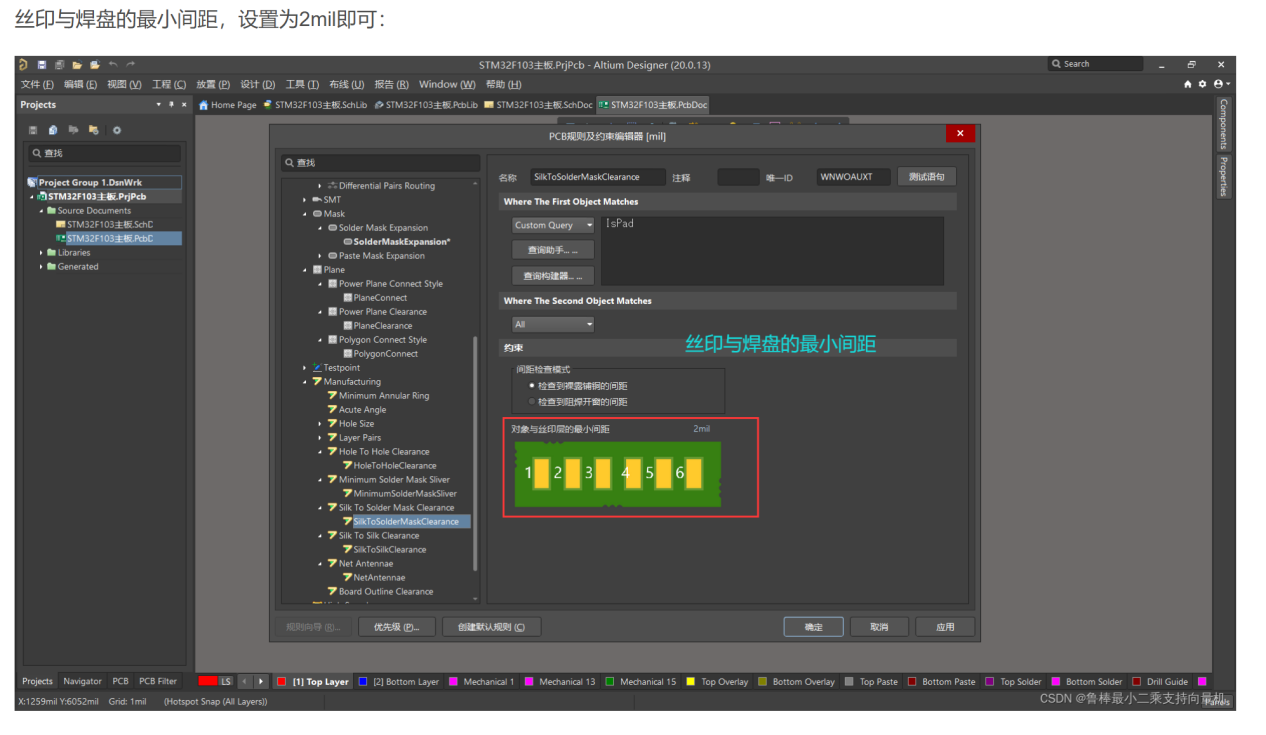


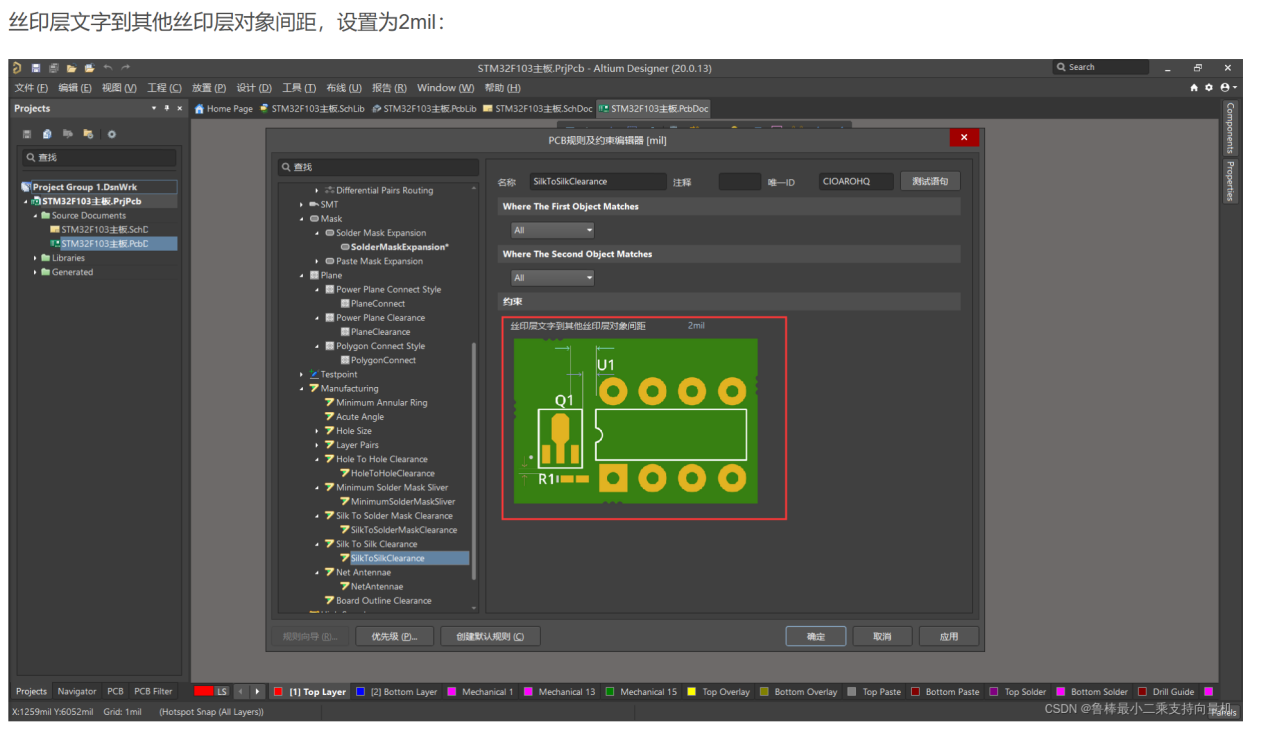


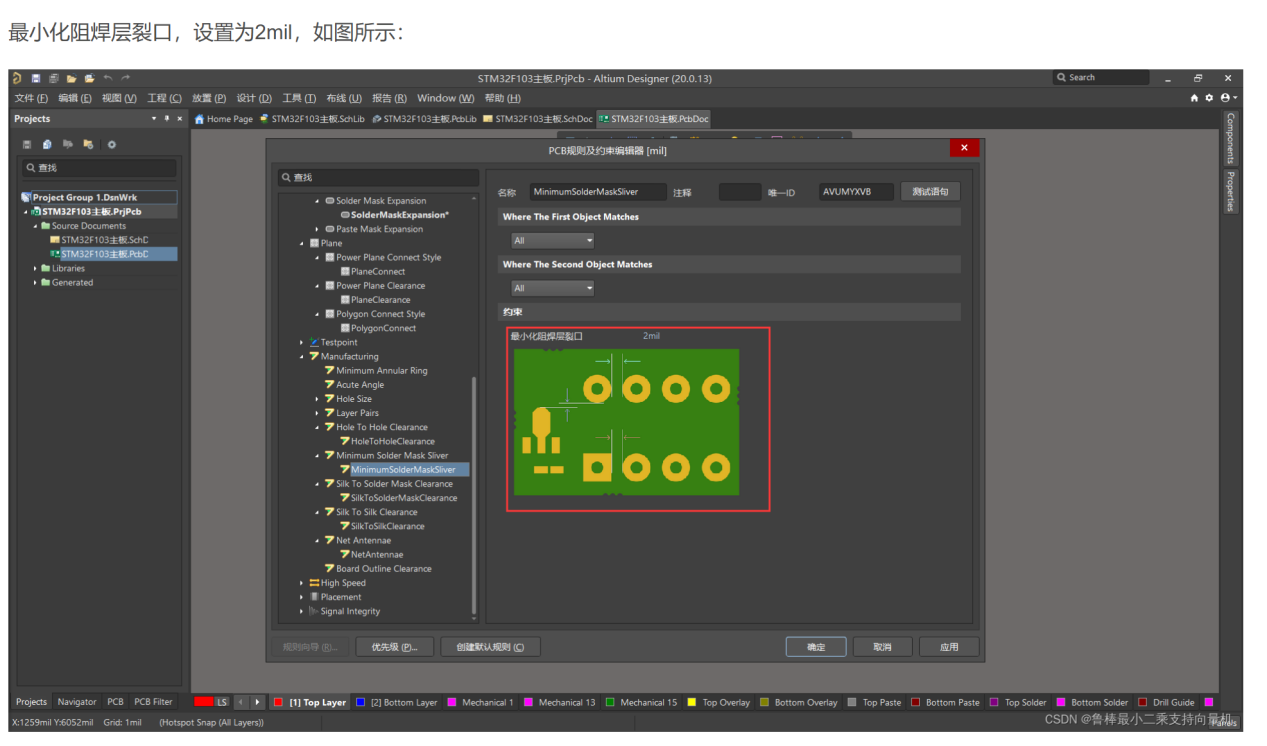












PID算法示例代码:

#include <stdint.h>

typedef struct {

// PID参数

float Kp;

float Ki;

float Kd;

// 误差变量

float error;

float last\_error;

float integral;

float derivative;

// 输出限制

float output\_max;

float output\_min;

// 控制输出

float output;

} PID\_Controller;

// PID初始化

void PID\_Init(PID\_Controller \*pid, float Kp, float Ki, float Kd,

float output\_min, float output\_max) {

pid->Kp = Kp;

pid->Ki = Ki;

pid->Kd = Kd;

pid->error = 0;

pid->last\_error = 0;

pid->integral = 0;

pid->derivative = 0;

pid->output\_min = output\_min;

pid->output\_max = output\_max;

pid->output = 0;

}

// PID计算（位置式）

float PID\_Calculate(PID\_Controller \*pid, float setpoint, float feedback) {

// 计算当前误差

pid->error = setpoint - feedback;

// 计算积分项

pid->integral += pid->error;

// 计算微分项

pid->derivative = pid->error - pid->last\_error;

// PID输出计算

pid->output = pid->Kp \* pid->error +

pid->Ki \* pid->integral +

pid->Kd \* pid->derivative;

// 输出限幅

if (pid->output > pid->output\_max) {

pid->output = pid->output\_max;

} else if (pid->output < pid->output\_min) {

pid->output = pid->output\_min;

}

// 保存当前误差

pid->last\_error = pid->error;

return pid->output;

}

常用库函数:

int snprintf(char \*str, size\_t size, const char \*format, ...);

Str:缓冲区

Size:缓冲区最大容量

Format:格式字符

...:%xxx

返回值:若格式化后的字符长度<=n-1,返回实际写入的字符

若大于,返回完整字符串需要的总长度

int sprintf(char \*str, const char \*format, ...);

Str:目标字符串缓冲区,存储格式化后的结果

Format:格式化字符串(%d)

...:%d的具体值

返回值:字符总数(不包括\0),失败返回-1

char \*strncpy(char \*dest, const char \*src, size\_t n);

从src复制最多n个字符到dest

如果src长度小于n,会用\0填充剩余部分

如果大于n,仅复制前n个字符,且不会自动添加终止符”\0”

int strncmp(const char \*s1, const char \*s2, size\_t n);

比较s1与s2前n个字符

若s1<s2,前n个字符中存在不相等字符且s1的字符ascii码更小返回负数

若s1 == s2 返回0

int atoi(const char \*str);

将str转换为对应的int,可识别+或-,若字符串无法有效转换,则返回0

void \*memset(void \*ptr, int value, size\_t num);

将ptr指向的内存地址开始的num个字节,每个字节都设置为value,返回 指向ptr的指针

void \*memcpy(void \*dest, const void \*src, size\_t n);

从src指向的内存块中复制n个字节的数据到dest指向的内存块

复制过程按字节逐个进行