目　录

第 2 章 STM32 的学习

在你阅读这个部分的时候，我希望你对 STM32 的寄存器及库函数的开发方式有所了解。而这一部分的主要内容则是基于 HAL 库和 Cube MX 的开发方式。本来我有计划写一个部分介绍 STC51 单片机，但是这两者重复部分太多，所以最终仅保留 STM32 部分，并对其做更细致的探讨。

如果你学过 STC51，你一定知道 STC51 操作是极其方便的。如果你学过 STM32 的库函数，你一定知道 STM32 操作是极其繁琐的。传统的库函数开发方式，将太多时间花费在各种东西的初始化上。同时，如果你学过 STM32F1、STM32F3、STM32F4 的话，你会发现对于不同型号的 STM32 在使用库函数的开发方式下，他的初始化流程也是不一样的，这也是传统开发方式的一种弊端。**而 Cube MX + HAL 库开发的方式，则是省去了初始化的部分，让开发人员将更多的精力放在业务的处理！**但是寄存器及库函数的开发方式也是有必要学习的，因为 Cube MX 也可能存在 Bug, 如果你对寄存器及库函数不了解那你会很被动。

如有错误，欢迎指正，如有帮助，不胜荣幸！

2.1 GPIO

GPIO（英语：General-purpose input/output），通用型之输入输出的简称，其接脚可以供使用者由程控自由使用，PIN 脚依现实考量可作为通用输入（GPI）或通用输出（GPO）或通用输入与输出（GPIO）

2.1.1 GPIO 8 种工作模式

GPIO_Mode_AIN 模拟输入
GPIO_Mode_IN_FLOATING 浮空输入
GPIO_Mode_IPD 下拉输入
GPIO_Mode_IPU 上拉输入
GPIO_Mode_Out_OD 开漏输出
GPIO_Mode_Out_PP 推挽输出
GPIO_Mode_AF_OD 复用开漏输出
GPIO_Mode_AF_PP 复用推挽输出

2.1.2 应用总结

1、上拉输入、下拉输入可以用来检测外部信号；例如，按键等；

2、浮空输入模式，由于输入阻抗较大，一般把这种模式用于标准通信协议的 I2C、USART 的接收端；

3、普通推挽输出模式一般应用在输出电平为 0 和 3.3V 的场合。而普通开漏输出模式一般应用在电平不匹配的场合，如需要输出 5V 的高电平，就需要在外部一个上拉电阻，电源为 5V，把 GPIO 设置为开漏模式，当输出高阻态时，由上拉电阻和电源向外输出 5V 电平。

4、对于相应的复用模式（复用输出来源片上外设），则是根据 GPIO 的复用功能来选择，如 GPIO 的引脚用作串口的输出（USART/SPI/CAN），则使用复用推挽输出模式。如果用在 I2C、SMBUS 这些需要线与功能的复用场合，就使用复用开漏模式。

5、在使用任何一种开漏模式时，都需要接上拉电阻。

2.1.3 Cube MX 相关配置

2.1.3.1 选择引脚类型

GPIO_Input-输入引脚 GPIO_Output-输出引脚

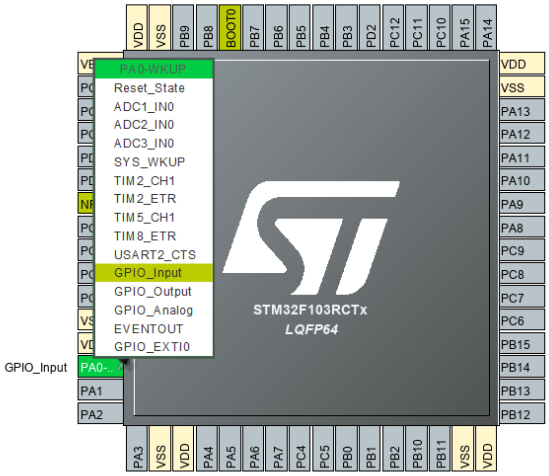

图 2.1: 选择引脚类型

2.1.3.2 配置引脚

对于输入引脚，可以配置的就是 GPIO Pull-up/Pull-down。这分别对应的就是 Pull-up（输入上拉）与 Pull-down （输入下拉）。

Pull-up: 输入上拉就是把电位拉高，比如拉到 Vcc。上拉就是将不确定的信号通过一个电阻嵌位在高电平。电阻同时起到限流的作用。弱强只是上拉电阻的阻值不同，没有什么严格区分。

Pull-down: 输入下拉就是把电压拉低，拉到 GND。与上拉原理相似。

简单的说，如果你希望你的引脚平时处于高电平用于检测低电平，你就使用 Pull-up。如果你希望你的引脚平时处于低电平用于检测高电平，你就使用 Pull-down。

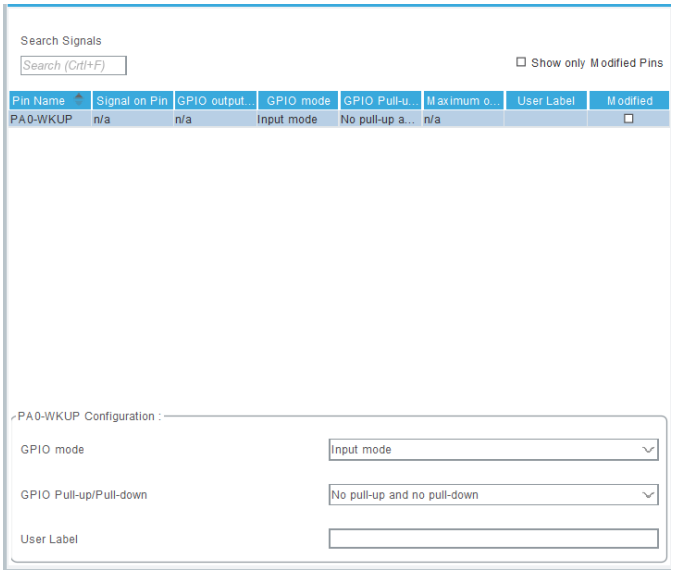

图 2.2: 配置输入引脚

对于输出引脚，比输入多了更多的配置：

GPIO output level -> 初始化输出电平

GPIO mode -> 输出方式 -> 开漏或推挽输出

GPIO Pull-up/Pull-down -> 上拉或下拉输出

Maximum output speed 选中 GPIO 管脚的速率

选中 *GPIO* 管脚的速率

I/O 口的输出模式下，有 3 种输出速度可选 (Low - 2MHz、Medium - 10MHz、High - 50MHz)，这个速度是指 I/O 口驱动电路的响应速度而不是输出信号的速度，输出信号的速度与程序有关（芯片内部在 I/O 口的输出部分安排了多个响应速度不同的输出驱动电路，用户可以根据自己的需要选择合适的驱动电路）。通过选择速度来选择不同的输出驱动模块，达到最佳的噪声控制和降低功耗的目的。高频的驱动电路，噪声也高，当不需要高的输出频率时，请选用低频驱动电路，这样非常有利于提高系统的 EMI 性能。当然如果要输出较高频率的信号，但却选用了较低频率的驱动模块，很可能会得到失真的输出信号。

举个栗子：

1、USART 串口，若最大波特率只需 115.2k，那用 2M 的速度就够了，既省电也噪声小。

2、I2C 接口，若使用 400k 波特率，若想把余量留大些，可以选用 10M 的 GPIO 引脚速度。

3、SPI 接口，若使用 18M 或 9M 波特率，需要选用 50M 的 GPIO 的引脚速度。

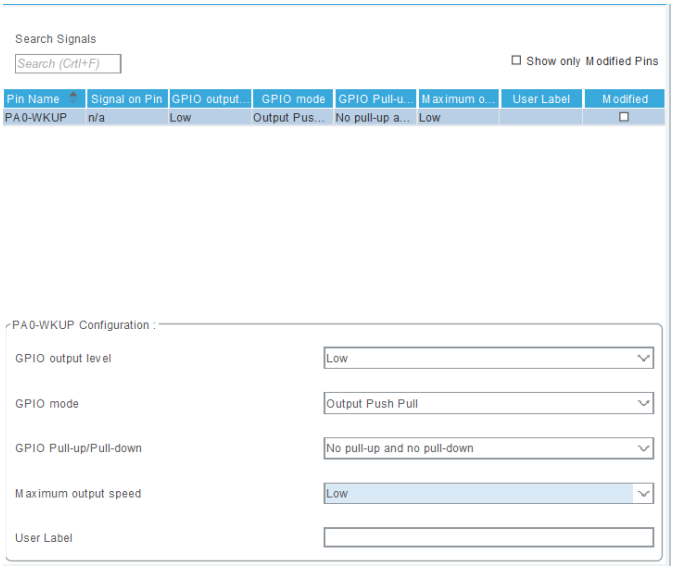

图 2.3: 配置输出引脚

2.1.4 编写业务代码

2.1.4.1 初始化及重置相关

```
//初始化引脚
void HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init);
 //重置引脚
void HAL_GPIO_DeInit(GPIO_TypeDef *GPIOx, uint32_t GPIO_Pin);
```

2.1.4.2 IO 口操作相关

```
//读取电平状态
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin);
//设置引脚状态
void HAL_GPIO_WritePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin, GPIO_PinState
    PinState);
//转换引脚状态
void HAL_GPIO_TogglePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin);
//锁定引脚状态
HAL_StatusTypeDef HAL_GPIO_LockPin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin);
```

同时 HAL 库帮我定义好了 GPIO_PIN_RESET 与 GPIO_PIN_SET，代表着 1（高电平）、0（低电平）。

2.1.5 User Label

对于任意引脚，它都有这么一个选项。我想告诉你这个选项特别特别好用！这个选项简单的说就是它帮你在 main.h 中生成 define 语句。但是对于 HAL 库编程，main.h 会

被用户的每个模块调用，也就是这些 define 语句的作用域几乎是全局。

```
/* Private defines -----------------------------------------------------------*/
#define R3_Pin GPIO_PIN_2
#define R3_GPIO_Port GPIOE
#define R2_Pin GPIO_PIN_3
#define R2_GPIO_Port GPIOE
#define R1_Pin GPIO_PIN_4
#define R1_GPIO_Port GPIOE
#define C1_Pin GPIO_PIN_5
#define C1_GPIO_Port GPIOE
#define C2_Pin GPIO_PIN_6
#define C2_GPIO_Port GPIOE
#define PWM_Pin GPIO_PIN_0
#define PWM_GPIO_Port GPIOA
#define PWM_EXTI_IRQn EXTI0_IRQn
#define BEER_Pin GPIO_PIN_2
#define BEER_GPIO_Port GPIOA
#define C3_Pin GPIO_PIN_7
#define C3_GPIO_Port GPIOE
#define C4_Pin GPIO_PIN_8
#define C4_GPIO_Port GPIOE
#define key_Pin GPIO_PIN_8
#define key_GPIO_Port GPIOA
#define R4_Pin GPIO_PIN_1
#define R4_GPIO_Port GPIOE
```

图 2.4: Cube MX 生成的 define 语句

举个例子让你感受一下，在一次开发中，我使用 PA0 来作为输出引脚。如果随着开发的继续 PA0 被迫要用于其他功能，那么你该怎么办？那你必须使用另外一个引脚（假设是 PB1）来替代它。

如果你没有配置 User Label 选项，那你的代码中可能大量的充斥着

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_RESET);//将PA0引脚状态改为低电平
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_SET);//将PA0引脚状态改为高电平
```

然后你又需要用 PB1 来代替 PA0，那你就需要将整个代码中有关 PA0 的 GPIOA 改成 GPIOB，将 GPIO_PIN_0 改成 GPIO_PIN_1。这会导致巨大的工作量，并且容易出错。

那么我们来看看使用了 User Label 会带来什么变化，使用 User Label 把他取名 R1。那你的代码中充斥着的不在是 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_RESET)，而是 HAL_GPIO_WritePin(R1_GPIO_Port, R1_Pin, GPIO_PIN_RESET)。当遇到 PA0 被迫要用于其他功能，你只需要把 PB1 的 User Label 取名为 R1 后，代码不需要做丝毫改变。

在我的开发中，这个应用最典型的两个例子就是“矩阵键盘”和“ADS1256”的开发。用矩阵键盘来举例，需要用到 8 个引脚。

Pin Name	Signal on Pin	GPIO output...	GPIO mode	GPIO Pull-up...	Maximum o...	User Label	Modified
PA0-WKUP	n/a	n/a	External Inte...	Pull-down	n/a	PWM	☑
PA2	n/a	Low	Output Push...	Pull-down	Low	BEER	☑
PA8	n/a	n/a	Input mode	Pull-up	n/a	key	☑
PE1	n/a	Low	Output Push...	Pull-up	Low	R4	☑
PE2	n/a	Low	Output Push...	Pull-up	Low	R3	☑
PE3	n/a	Low	Output Push...	Pull-up	Low	R2	☑
PE4	n/a	Low	Output Push...	Pull-up	Low	R1	☑
PE5	n/a	n/a	Input mode	Pull-down	n/a	C1	☑
PE6	n/a	n/a	Input mode	Pull-down	n/a	C2	☑
PE7	n/a	n/a	Input mode	Pull-down	n/a	C3	☑
PE8	n/a	n/a	Input mode	Pull-down	n/a	C4	☑

图 2.5: Cube MX 的配置

我的矩阵键盘中的代码全是由 R1-R4、C1-C4 组成，所以在各这个代码的复用性极

其强，无论是换引脚还是换单片机型号，我只需要在 Cube MX 中配置一下，就可以马上投入使用。

```
uint8_t key_scan() {

    uint8_t key_value = 0;
    HAL_GPIO_WritePin(R1_GPIO_Port, R1_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(R2_GPIO_Port, R2_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(R3_GPIO_Port, R3_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(R4_GPIO_Port, R4_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(C1_GPIO_Port, C1_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(C2_GPIO_Port, C2_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(C3_GPIO_Port, C3_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(C4_GPIO_Port, C4_Pin, GPIO_PIN_RESET);

    HAL_GPIO_WritePin(R1_GPIO_Port, R1_Pin, GPIO_PIN_SET);
    if(HAL_GPIO_ReadPin(C1_GPIO_Port, C1_Pin) == GPIO_PIN_SET) {
        HAL_Delay(5);
        if(HAL_GPIO_ReadPin(C1_GPIO_Port, C1_Pin) == GPIO_PIN_SET) {
            while(HAL_GPIO_ReadPin(C1_GPIO_Port, C1_Pin) == GPIO_PIN_SET) {}
            return 1;
        }
    }
    if(HAL_GPIO_ReadPin(C2_GPIO_Port, C2_Pin) == GPIO_PIN_SET) {
        HAL_Delay(5);
        if(HAL_GPIO_ReadPin(C2_GPIO_Port, C2_Pin) == GPIO_PIN_SET) {
            while(HAL_GPIO_ReadPin(C2_GPIO_Port, C2_Pin) == GPIO_PIN_SET) {}
            return 2;
        }
    }
    if(HAL_GPIO_ReadPin(C3_GPIO_Port, C3_Pin) == GPIO_PIN_SET) {
        HAL_Delay(5);
```

图 2.6: 矩阵键盘代码截图

2.2 串口通信

串口通信（Serial Communications）的概念非常简单，串口按位（bit）发送和接收字节。

2.2.1 UART 与 USART

UART: 通用异步收发传输器（Universal Asynchronous Receiver/Transmitter)，通常称作 UART。它将要传输的资料在串行通信与并行通信之间加以转换。作为把并行输入信号转成串行输出信号的芯片，UART 通常被集成于其他通讯接口的连结上。

USART:(Universal Synchronous/Asynchronous Receiver/Transmitter) 通用同步/异步串行接收/发送器,USART 是一个全双工通用同步/异步串行收发模块，该接口是一个高度灵活的串行通信设备。

2.2.2 Cube MX 相关配置

2.2.2.1 初始化引脚

Mode :

Asynchronous : 异步, 整个过程，不会阻碍发送者的工作。

Synchronous : 同步, 同步信息一旦发送，发送者必须等到应答，才能继续后续的行为。

Single Wire : 单总线, 半双工。

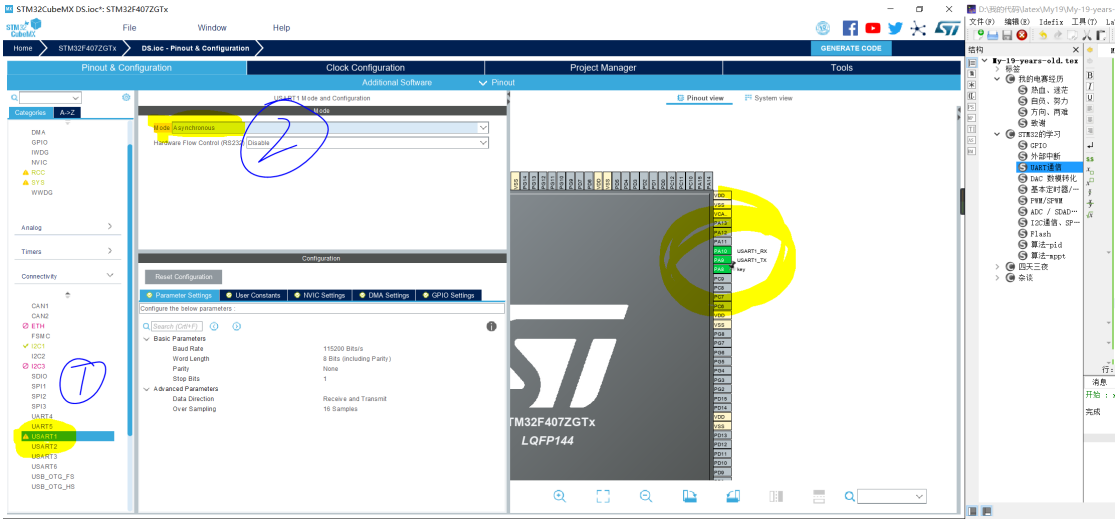

图 2.7: 使能引脚

2.2.2.2 配置引脚

Baud Rate: 波特率, 波特率表示每秒钟传送的码元符号的个数，是衡量数据传送速率的指标，它用单位时间内载波调制状态改变的次数来表示。对于串口最重要的就是波特率, 常用的波特率为 115200 与 9600。

Wrod Length : 数据长

Parity : 奇偶校验 -> 无、奇校验、偶校验

Stop : 停止位

以上的配置与需要通信双方完全配对

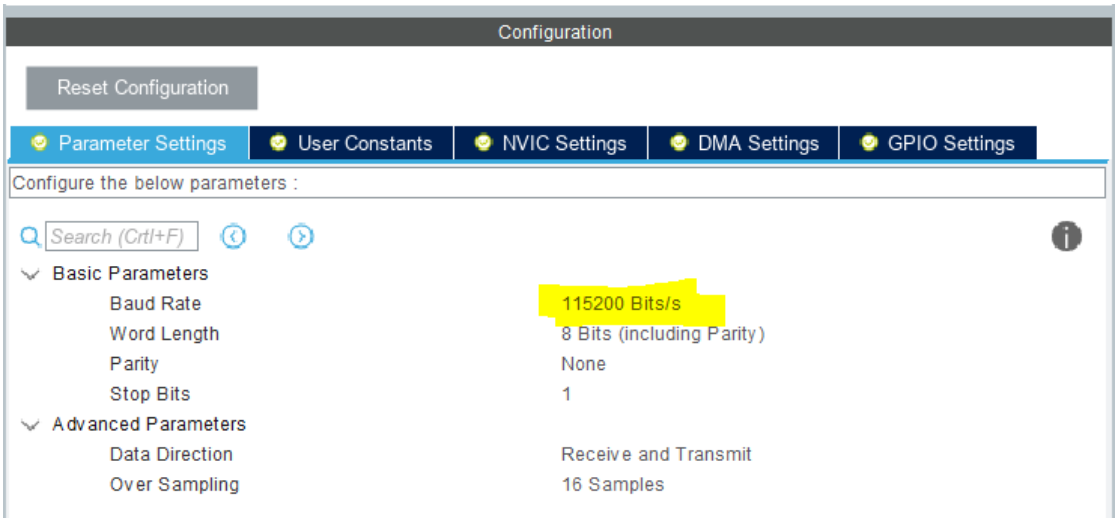

图 2.8: 配置引脚

2.2.3 编写逻辑代码

```
//发送数据
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData,
    uint16_t Size, uint32_t Timeout);
//接收数据
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t *pData,
    uint16_t Size, uint32_t Timeout);
//发送中断
HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart, uint8_t *
    pData, uint16_t Size);
//接收中断
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData
    , uint16_t Size);
//使用DMA发送
HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart, uint8_t *
    pData, uint16_t Size);
//使用DMA接收
HAL_StatusTypeDef HAL_UART_Receive_DMA(UART_HandleTypeDef *huart, uint8_t *
    pData, uint16_t Size);
//DMA暂停
HAL_StatusTypeDef HAL_UART_DMAPause(UART_HandleTypeDef *huart);
//DMA恢复
HAL_StatusTypeDef HAL_UART_DMAResume(UART_HandleTypeDef *huart);
//DMA停止
HAL_StatusTypeDef HAL_UART_DMAStop(UART_HandleTypeDef *huart);
```

就我目前的学习来看 HAL 并没有对同步通信的方式做拓展，所以上述都是关于 UART 的函数。

2.2.4 printf 重定向

在 Private includes 中引入：

```
#include <stdio.h>
```

在 USER CODE BEGIN 0 添加：

```
int fputc(int ch, FILE *f){
  uint8_t temp[1] = {ch};
  HAL_UART_Transmit(&huart1, temp, 1, 2);//huart1需要根据你的配置修改
  return ch;
}
```

然后你就可以在任意地方使用 printf 语句方便的输出你想要的内容。

2.2.5 Log 信息格式

2.2.5.1 格式 1

参考目前主流嵌入式、安卓等输出方式：

```
[日志级别] 文件名 ： 日志信息
//例: [info] main.c : init ok!
//例: [debug] adc.c : adc_getvalue -> 3.3v
```

2.2.5.2 格式 2

参考 Java 日志框架的输出方式：

```
[        文件名] 日志级别 ： 日志信息
//例: [      main] info : init ok!
//例: [      adc] debug : adc_getvalue -> 3.3v
```

2.2.6 条件编译

说到这里我还想向大家介绍一下条件编译。因为在进行单片机开发的过程中，会需要大量的 Log 信息，但是在开发结束时，你又不想它一直打印（这会拖慢单片机的速度）。所以我提出我的办法：

在头文件中添加：

```
#define Log 1 // 打印Log信息，不想打印时改为0即可
```

再把.c 文件中将所有的 printf 包裹上 #if Log 与 #endif:

```
#if Log
printf("[info]main.c:init!\r\n");
#endif
```

下面截选 mppt 算法中条件编译的使用：

```
int mppt_po(double u, double i, int pwm) {

    double power = (u * i) < 0 ? 0 : u * i ;

    if(mppt.state) {

#if Log
        printf("[Log] mppt_po : 当前电流: %f ,当前电压: %f ,当前功率: %f \r\n", i, u, power);
 #endif

        if(power < mppt.l_power  || pwm == mppt.pwm_max || pwm == mppt.pwm_min) {
            mppt.updown ^= 1;
            mppt.time ++;

            if(mppt.time > 5 && power < mppt.l_power )  mppt.l_pwm[mppt.count++] = pwm;
#if Log
            printf("[Log] mppt_po : 当前功率: %f , 小于此前功率: %f \r\n", power, mppt.l_power);
 #endif
```

图 2.9: 条件编译在代码中的使用

2.2.7 可变参数宏

关于这个内容，是我在阅读国内某云物联网模块源码是发现并学习的。

```
#ifdef USER_MQTT_DEBUG
    #define user_mqtt_info(format, ...)  mprintf( format "\r\n", ##__VA_ARGS__)
    #define user_mqtt_error(format, ...) mprintf( "[error]%s() %d " format "\r\n", /*__FILE__,*/ __FUNCTION__, __LINE__, ##__VA_ARGS__)
#else
    #define user_mqtt_info(format, ...)
    #define user_mqtt_error(format, ...)
#endif
```

图 2.10: 源码学习

我觉得这个解决方案比之前提到的条件编译强 100 倍，甚至让我感觉到以前的做法多么的愚蠢。这种方法不仅达到了代码的格式化，同时也完成了条件编译。

在此分享我的设计：

```
#ifdef USER_MAIN_DEBUG
#define user_main_printf(format, ...) printf( format "\r\n", ##__VA_ARGS__)
#define user_main_info(format, ...) printf("[\tmain]info:" format "\r\n", ##
    __VA_ARGS__)
#define user_main_debug(format, ...) printf("[\tmain]debug:" format "\r\n", ##
    __VA_ARGS__)
#define user_main_error(format, ...) printf("[\tmain]error:" format "\r\n",##
    __VA_ARGS__)
#else
#define user_main_printf(format, ...)
#define user_main_info(format, ...)
#define user_main_debug(format, ...)
#define user_main_error(format, ...)
#endif
```

当我需要打印串口信息的时候，define 一个 USER_MAIN_DEBUG, 在我不需要时将其注释。

2.2.8 个性化输出

1、借助下面的网站设计自己的字符

http://patorjk.com/software/taag/

图 2.11: 字符组成的 reyunn

2、编写代码

先逐行复制输入

```
/* USER CODE BEGIN 2 */
user_main_printf("  _____                                  ");
user_main_printf(" |  __ \                                 ");
user_main_printf(" | |__) |___ _   _ _   _ _ __  _ __     ");
user_main_printf(" |  _  // _ \ | | | | | | '_ \| '_ \    ");
user_main_printf(" | | \ \  __/ |_| | |_| | | | | | | |   ");
user_main_printf(" |_|  \_\___|\__, |\__,_|_| |_|_| |_|   ");
user_main_printf("               __/ |                    ");
user_main_printf("              |___/                     ");
/* USER CODE END 2 */
```

图 2.12: 逐行复制输入

再用转义字符修补错误

```
MX_USART1_UART_Init();
/* USER CODE BEGIN 2 */
user_main_printf("  _____                                  ");
user_main_printf(" |  __ \\                                 ");
user_main_printf(" | |__) |___ _   _ _   _ _ __  _ __     ");
user_main_printf(" |  _  // _ \\ | | | | | | '_ \\| '_ \\    ");
user_main_printf(" | | \\ \\  __/ |_| | |_| | | | | | | |   ");
user_main_printf(" |_|  \\_\\___|\\__, |\\__,_|_| |_|_| |_|   ");
user_main_printf("               __/ |                    ");
user_main_printf("              |___/                     ");
/* USER CODE END 2 */
```

图 2.13: 转义字符修补错误

3、串口助手看效果

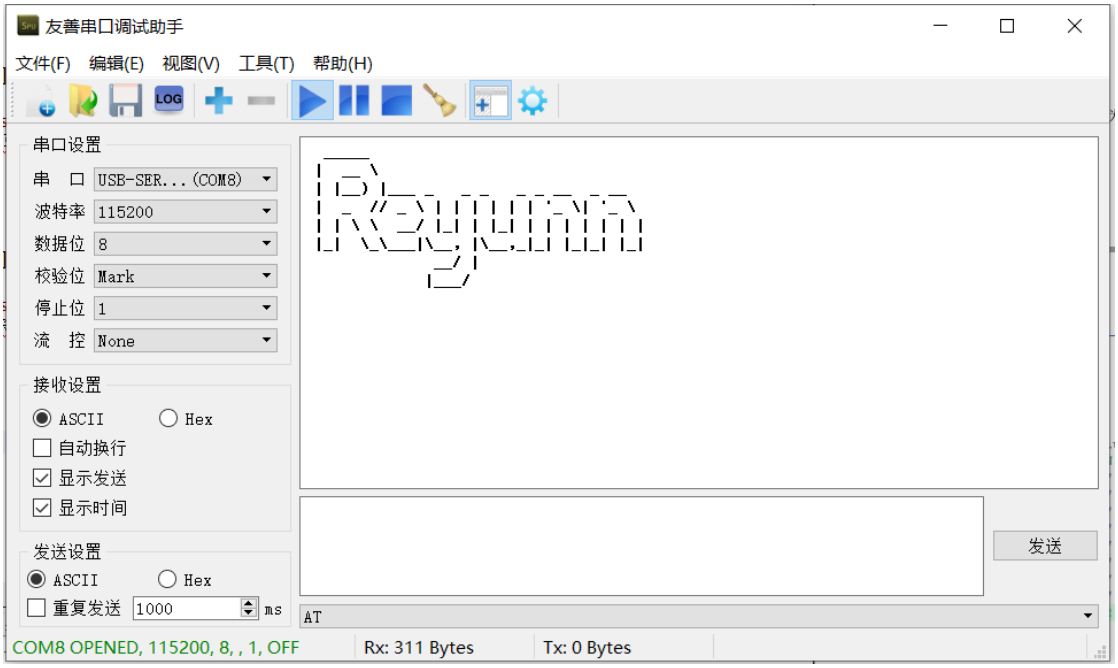

图 2.14: 打印信息效果

2.2.9 串口中断

1、Cube MX 中开启中断

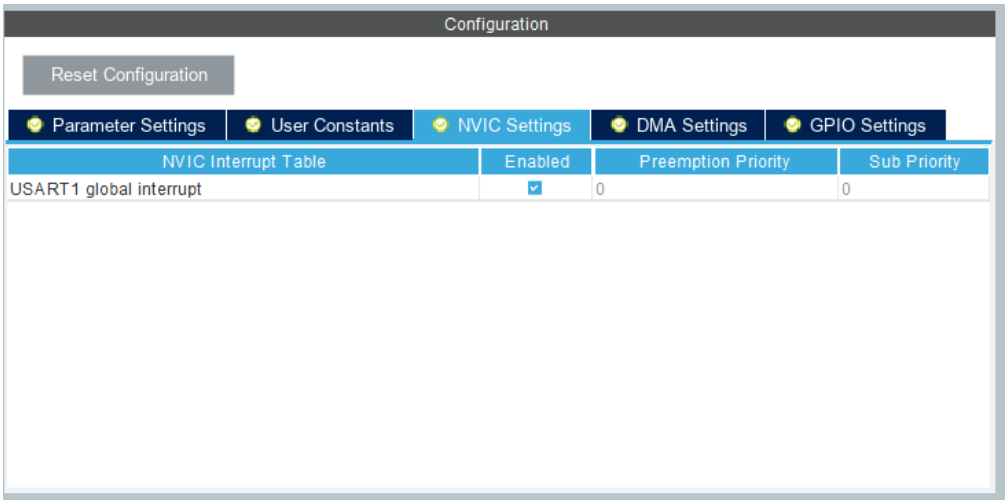

图 2.15: 开启中断

2、在 USER CODE BEGIN 2 中打开串口中断

```
HAL_UART_Receive_IT(&huart1, temp, 1);
```

3、在 USER CODE BEGIN 4 中实现回调函数

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
  if(huart -> Instance == huart1.Instance ) {
    ...//业务代码
  }
}
```

2.3 外部中断

外部中断是单片机实时地处理外部事件的一种内部机制。当某种外部事件发生时，单片机的中断系统将迫使 CPU 暂停正在执行的程序，转而去进行中断事件的处理；中断处理完毕后．又返回被中断的程序处，继续执行下去。

2.3.1 Cube MX 相关配置

2.3.1.1 初始化引脚

如果你想使用 PA1 作为外部中断的接收引脚，那么你只需要点击 PA1，在点击它对应的 GPIO_EXTIx

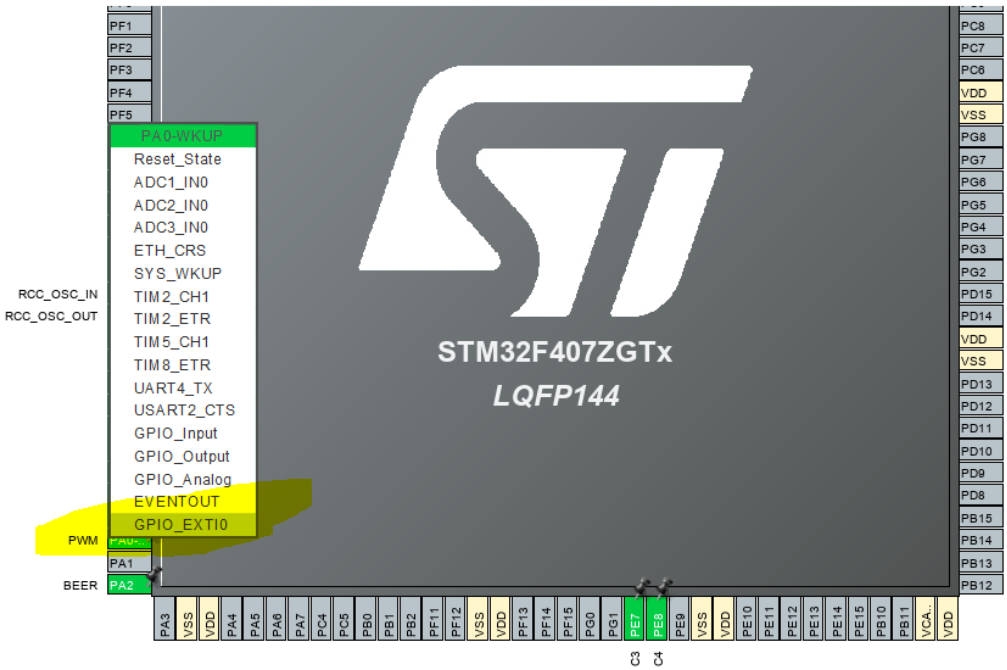

图 2.16: 使能引脚

2.3.1.2 使能中断

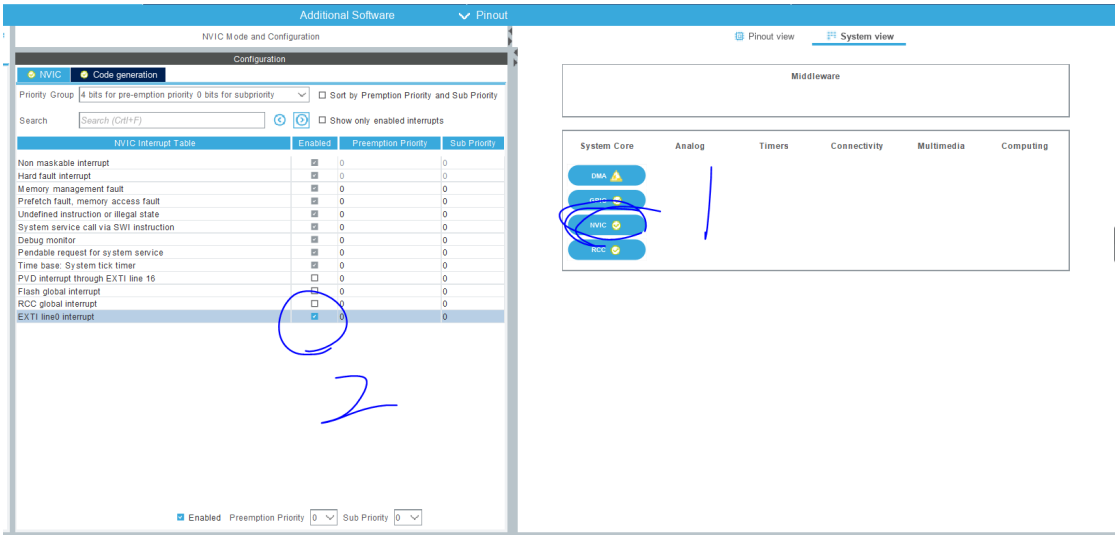

图 2.17: 使能中断

2.3.1.3 配置引脚

这个地方与此前不同的地方在于 GPIO mode。

External Interrupt Mode with Rising edge trigger detection//上升沿触发

External Interrupt Mode with Falling edge trigger detection//下降沿触发

External Interrupt Mode with Rising/Falling edge trigger detection//上升沿或下降沿触发

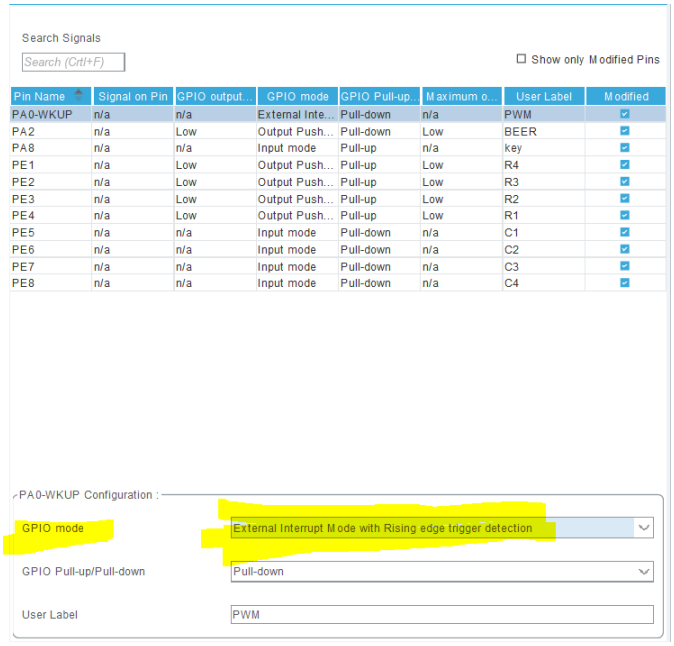

图 2.18: 配置引脚

2.3.2 编写逻辑代码

在 main.c 中的 USER CODE BEGIN 4 编程范围内添加外部中断的回调函数：

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if(GPIO_Pin == PWM_Pin) {
    ...//业务代码
  }
}
```

2.3.3 测量 pwm 频率

在我平时的学习中没有太多的使用外部中断，但是在最后的电赛中却巧妙的使用了它。

当时的情况是我们需要测量一个 PWM 的频率，我的解决办法是这样的：

当有上升沿的时候，就进入外部中断将 pwm_value 的值 +1。it is clear that "1s 钟上升沿的次数就是 pwm 的频率"。所以当我要用 pwm 的频率时，我就先将 pwm_value 置 0，再延时 1s，最后再使用 pwm_value。当然这并不是我最终的代码，因为你读到这里还有很多的内容没有学习, 往后的定时器章节将介绍它的滤波算法。

```
int pwm_value =0 ;

int main(){

  while (1){
    pwm_value = 0; // pwm_value置0
    HAL_Delay(1000); // 延时1s
    printf("[\tmain]info:pwm_value=%d\r\n",pwm_value); // 读取pwm_value
  }

}

/**
  * @brief 外部中断的回调函数
  * @param GPIO_Pin 触发中断的引脚
  * @retval None
  */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if(GPIO_Pin == PWM_Pin) { // 判断触发引脚是否是定义的引脚
    pwm_value++;
  }
}
```

2.4 时钟树

说到 STM32，必然逃不开时钟树。但是时钟树要展开讲的话会很麻烦，而且我也不一定讲的好。但是我想告诉你的是：通常我们会让单片机的频率（决定单片机的处理速度）提到最大，再进行其他分频操作。

原谅我技术有限，所以我想分享的关于时钟树的就是小小的一点：

2.4.1 使能外部时钟源

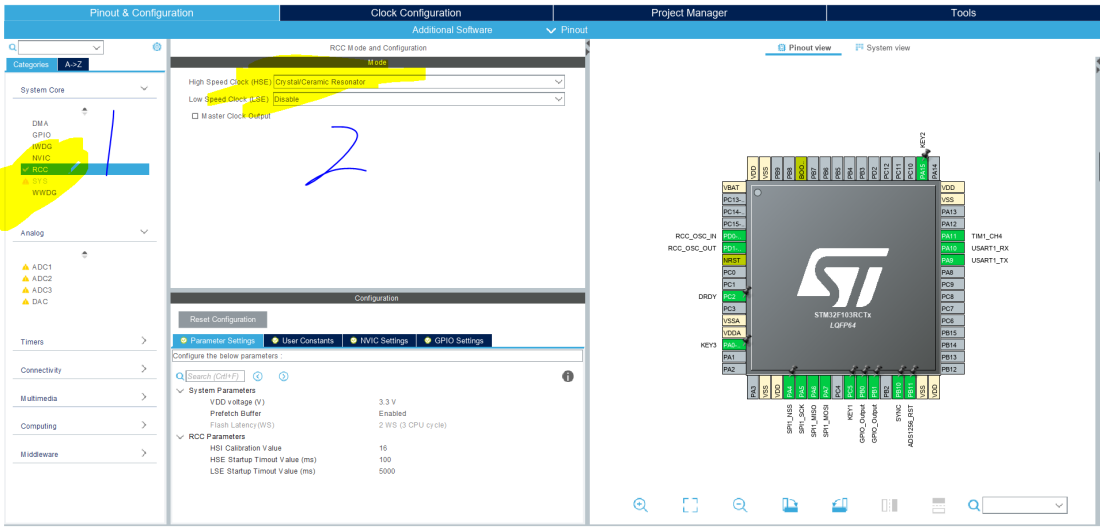

图 2.19: 使能外部时钟源

2.4.2 将频率调至最大

不同单片机的最大运行频率是不同的，例如 stm32f103 为 72M 而 stm32f407 为 84M。

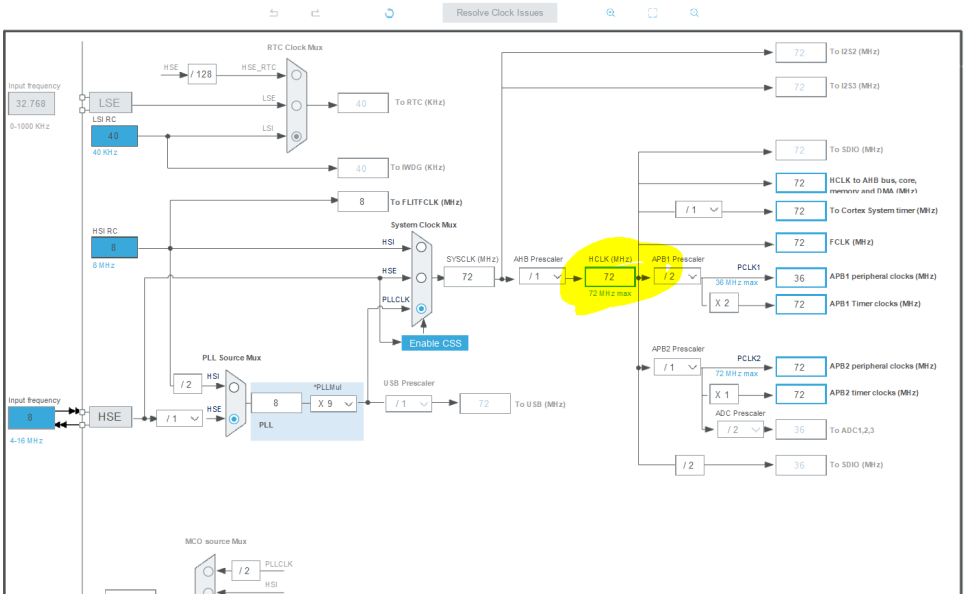

图 2.20: 将频率调至最大

2.4.3 按需分频

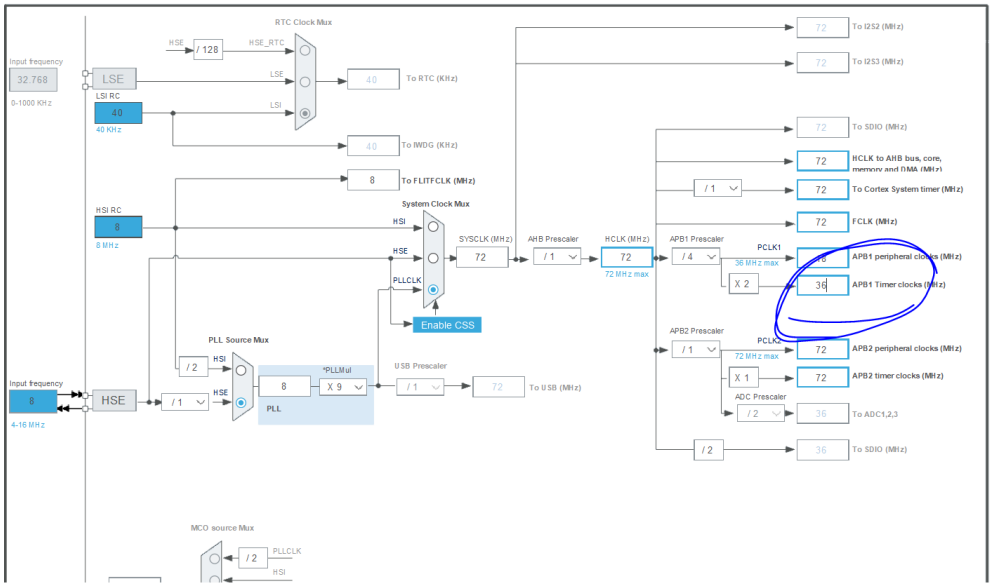

图 2.21: 按需分频

2.5 定时器

分享完时钟树的部分，接下来就是和它最紧密的定时器了。定时器最基本的内容就是定时产生中断了：

2.5.1 Cube MX 相关配置

2.5.1.1 配置定时器时钟

如之前所示，将定时器的时钟设为 72M。

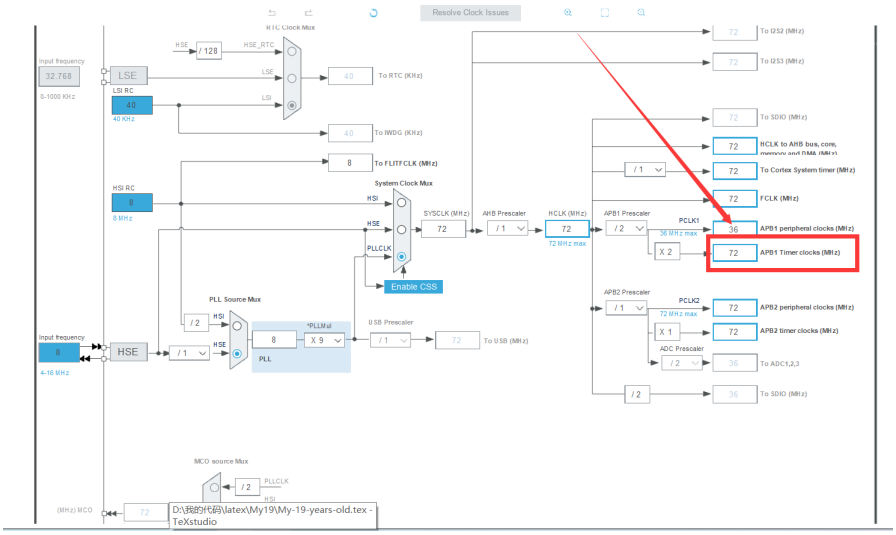

图 2.22: 配置定时器频率

2.5.1.2 选择时钟源

选择内部时钟

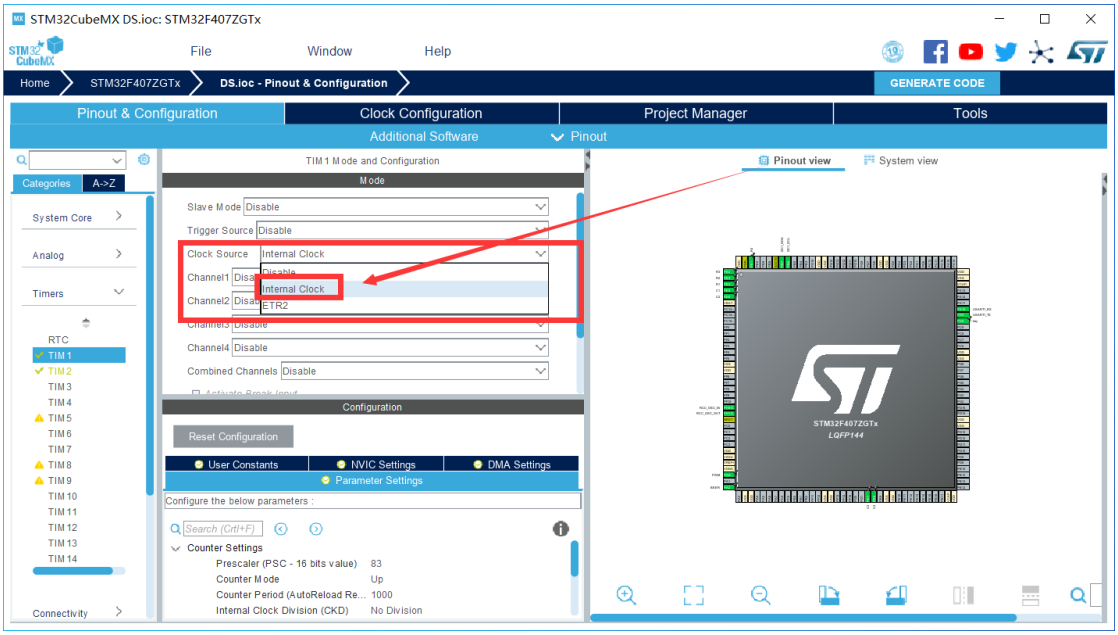

图 2.23: 选择时钟源

2.5.1.3 配置定时器

定时器的配置主要有两个：定时时间与是否重装定时器。

定时频率 = 定时器时钟 /（预分频 +1）/（计数值 + 1 ）Hz。

定时时间 = 1 / 定时频率 s。

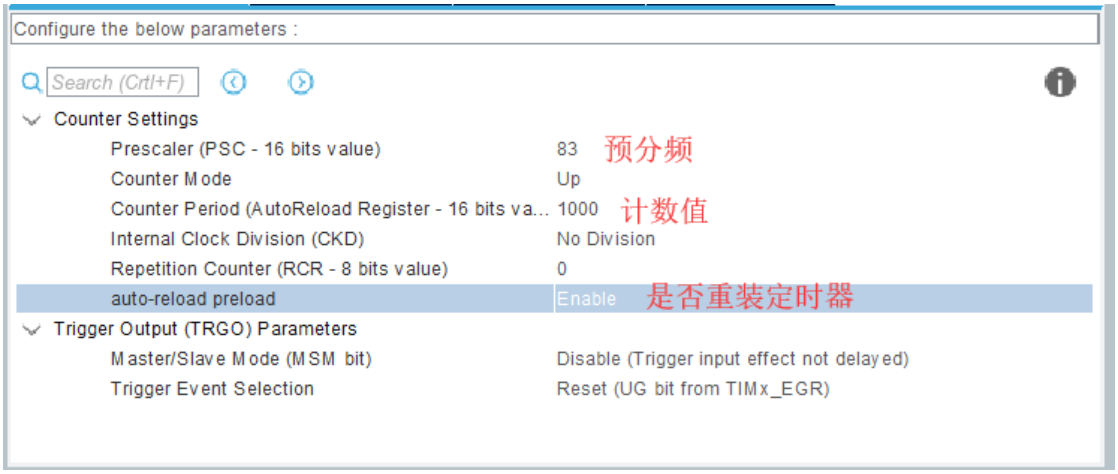

图 2.24: 配置定时器

2.5.1.4 开启中断 - 基本定时器

勾选 Enabled 框即可。

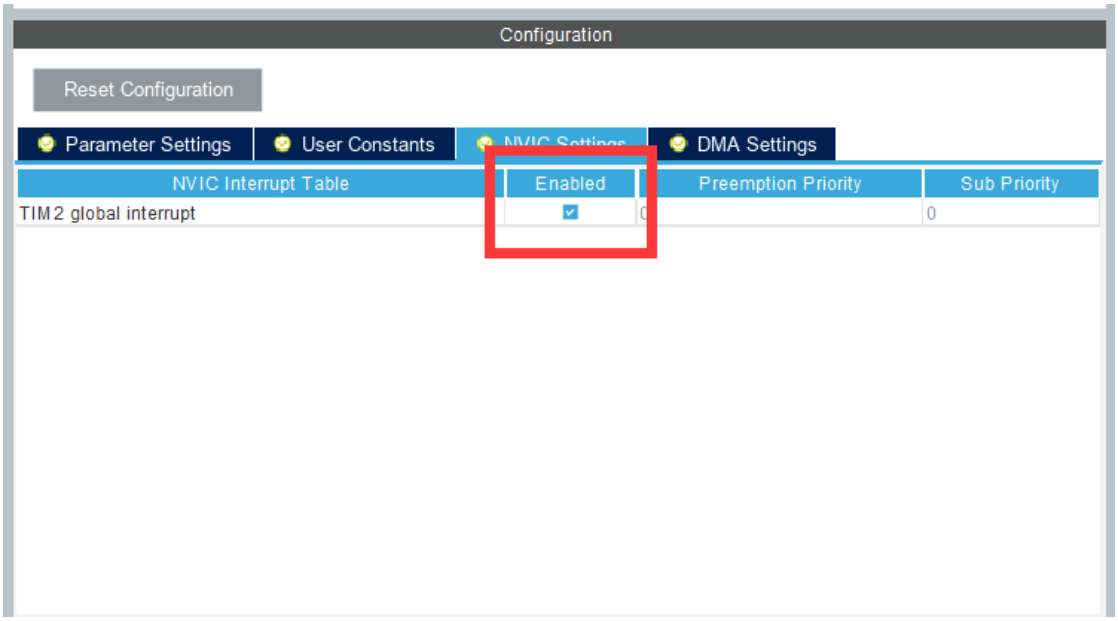

图 2.25: 开启中断

2.5.1.5 开启中断 - 高级定时器

勾选 TIM X update interrupt 后的 Enabled 框即可。

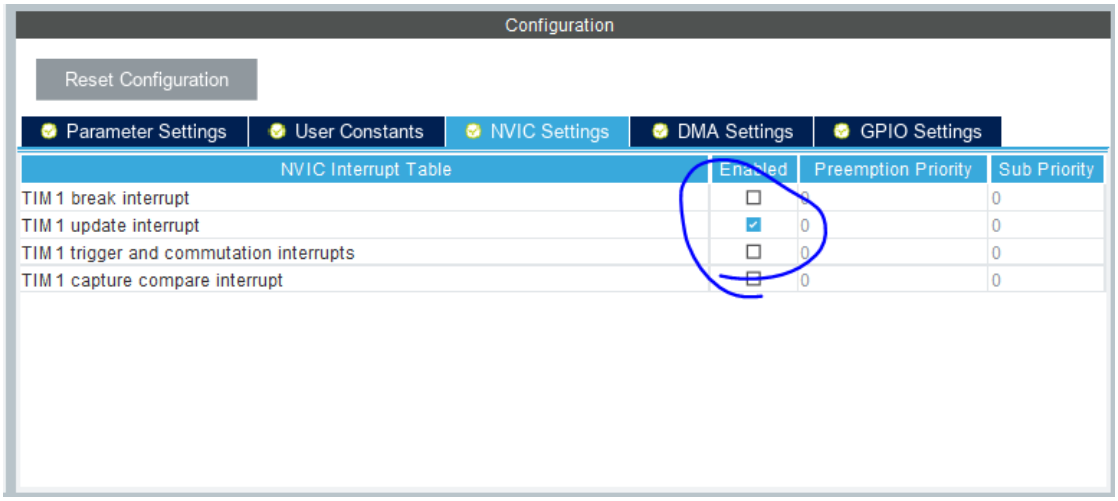

图 2.26: 开启中断

2.5.2 编写业务代码

```
int main(){

  HAL_TIM_Base_Start_IT(&htim1); //定时器1使能
  HAL_TIM_Base_Start_IT(&htim2); //定时器2使能
  ...

}
```

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {

  if (htim->Instance == htim1.Instance) {
    ...//定时器1中断业务
  }

  else if(htim-> Instance == htim2.Instance) {
     ...//定时器2中断业务
  }
  ...
}
```

2.5.3 平滑滤波

在这里我想在介绍定时器的另一种用法：平滑滤波。绝大部分人的滤波算法都是用的时候，多次采样再滤波。但是我希望让采样值在另一个“线程”一直滤波，而在我需要他的时候，直接取它的值即可。记得之前我描述过用外部中断实现的测量 pwm 波的频率，接下我想分享一下用定时器对其进行滤波。

```
/* 定时器2配置为0.1s触发一次中断 */
/**
  * @brief 定时器中断的回调函数
  * @param htim 触发中断的定时器
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  if(htim-> Instance == htim2.Instance) {
    pwm_sum += pwm_value * 10; //pwm_sum累加
    pwm_sum -= pwm_avg;        //pwm_sum减去上次的平均值
    pwm_avg = pwm_sum * 1.0 / 5; //更新pwm的平均值
    pwm_value_final = pwm_avg; //pwm_value_final的值即为当前pwm的频率
    pwm_value = 0;             //将pwm_value清空，重新计数
  }
}
/**
  * @brief 外部中断的回调函数
  * @param GPIO_Pin 触发中断的引脚
  * @retval None
  */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  if(GPIO_Pin == PWM_Pin) { // 判断触发引脚是否是定义的引脚
    pwm_value++;
  }
```

```
}
```

当我们在任意时刻需要使用 pwm 的频率时，只需要使用 pwm_value_final 的值即可。

2.6 PWM/SPWM

脉冲宽度调制（PWM），是英文“Pulse Width Modulation”的缩写，简称脉宽调试。是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。广泛应用在从测量、通信到功率控制与变换的许多领域中。

SPWM(Sinusoidal PWM) 法是一种比较成熟的、使用较广泛的 PWM 法。冲量相等而形状不同的窄脉冲加在具有惯性的环节上时，其效果基本相同。SPWM 法就是以该结论为理论基础，用脉冲宽度按正弦规律变化而和正弦波等效的 PWM 波形即 SPWM 波形控制逆变电路中开关器件的通断，使其输出的脉冲电压的面积与所希望输出的正弦波在相应区间内的面积相等，通过改变调制波的频率和幅值则可调节逆变电路输出电压的频率和幅值。

PWM 和 SPWM 在电源的备战中是很有必要的。基础的恒流源、恒压源需要使用 PWM 的占空比及频率来达到数控的作用，往后的逆变则需要用到 SPWM。那我就先从简单的 PWM 做分享,PWM 输出其实是定时器的一种应用。那么配置定时器时钟与选择时钟源我就不再赘述了。就从使能 PWM 通道开始讲起。

2.6.1 Cube MX 相关配置-PWM

2.6.1.1 使能 PWM 通道

在这里我将 TIM2 的 Channel1 设置为 PWM 输出通道 (PWM Generation CHx 正向、PWM Generation CHxN 反向、PWM Generation CHx CHxN 一对互补 pwm 输出)

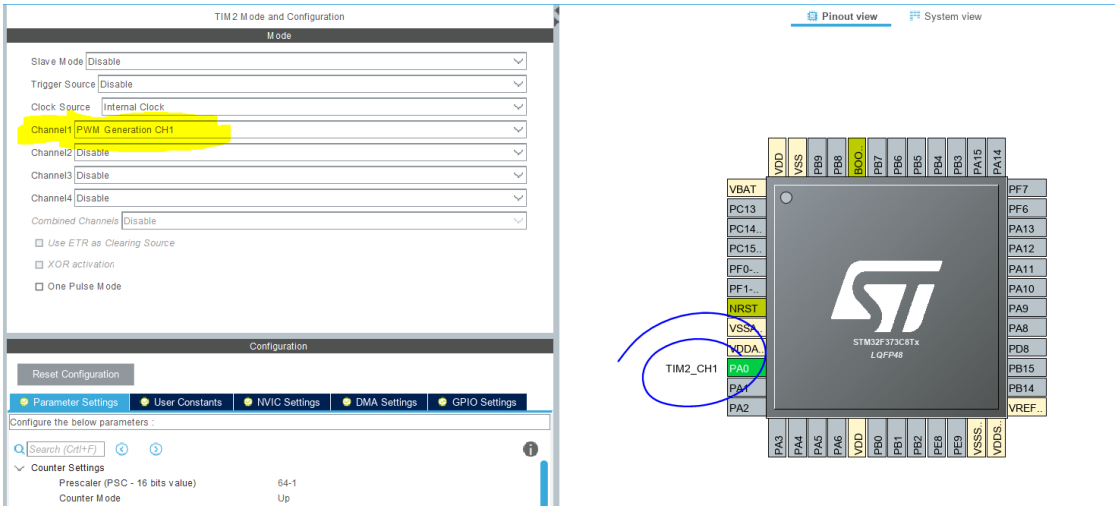

图 2.27: 使能 PWM 通道

2.6.1.2 配置频率及占空比

频率 = 定时器时钟 /（Prescaler 预分频 + 1）/（Counter Period 计数值 + 1）Hz

占空比 = Pulse (对比值) /（C ounter Period 计数值）%

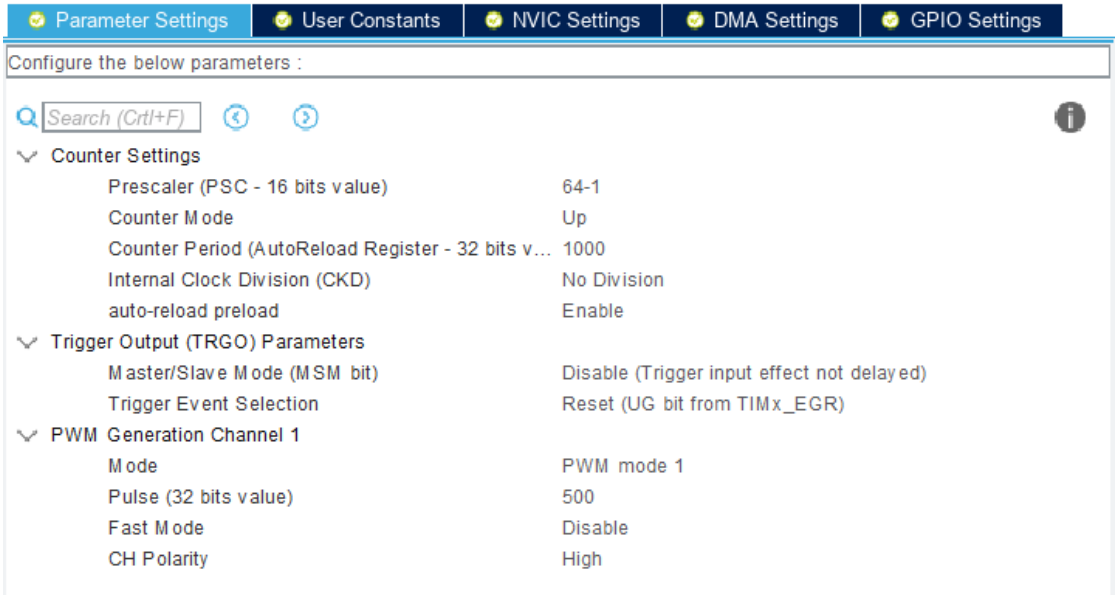

图 2.28: 配置频率及占空比

2.6.2 编写业务代码-PWM

```
// 使能timx的通道y
HAL_TIM_PWM_Start(&htimx,TIM_CHANNEL_y);
// 修改timx的通道y的pwm比较值为z，即修改占空比
__HAL_TIM_SET_COMPARE(&htimx, TIM_CHANNEL_y, z);
```

pwm 的输出是很简单的，但是因为定时器的频率是有上限的通常需要在频率和 pwm 的精细度两者之间做取舍。所以你想做电源，那么你可以了解一下 STM32F334 这款处理器，它拥有一个高分辨率定时器 (HRTIM)，能将定时器的频率倍频至 4.096G。那你在频率和 pwm 的精细度两者都可以兼得。

SPWM 其实就是在 PWM 的基础上，让 PWM 的占空比做正弦变化。

2.6.3 Cube MX 相关配置-SPWM

之前的 PWM 生成的操作不变，只需要开启一个新的定时器，配置完后需要开启定时器中断

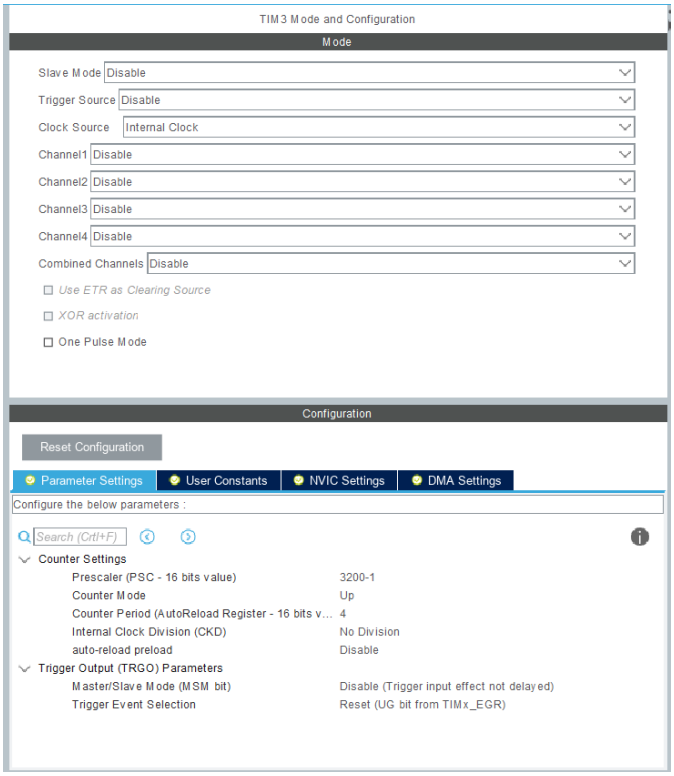

图 2.29: 开启一个新的定时器

2.6.4 使用软件生成正弦向量表-SPWM

SPWM 中值 = Pulse (对比值) /2

SPWM 幅值 = Pulse (对比值) /2

周内点数影响频率与正弦波精细度。周内点数越大，频率越小、正弦波精细度越高。

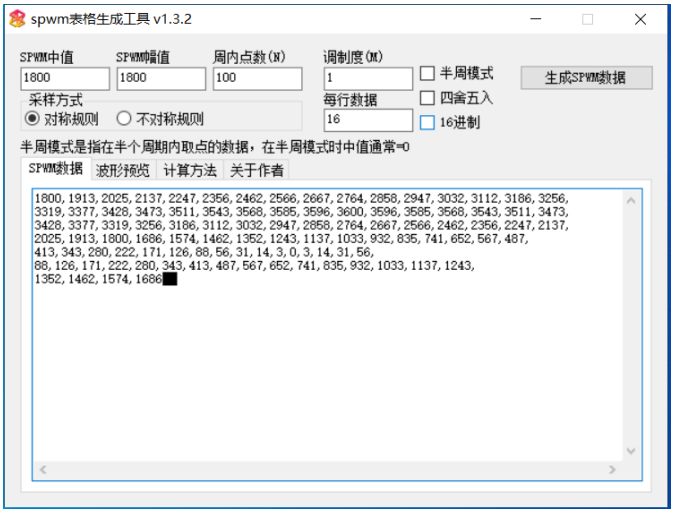

图 2.30: 使用软件生成正弦向量表

2.6.5 编写业务代码-SPWM

```
uint16_t sin[] = {
```

```
  1800,1913,2025,2137,2247,2356,2462,2566,2667,2764,
  2858,2947,3032,3112,3186,3256,3319,3377,3428,3473,
  3511,3543,3568,3585,3596,3600,3596,3585,3568,3543,
  3511,3473,3428,3377,3319,3256,3186,3112,3032,2947,
  2858,2764,2667,2566,2462,2356,2247,2137,2025,1913,
  1800,1686,1574,1462,1352,1243,1137,1033,932,835,
  741,652,567,487,413,343,280,222,171,126,
  88,56,31,14,3,0,3,14,31,56,
  88,126,171,222,280,343,413,487,567,652,
  741,835,932,1033,1137,1243,1352,1462,1574,1686
}

int main(){
  HAL_TIM_PWM_Start(&htimx,TIM_CHANNEL_y); // 开启pwm输出
  HAL_TIM_Base_Start_IT(&htimz); //使能刚刚配置的定时器z
  while(1){
  }
}
/**
  * @brief 定时器中断的回调函数
  * @param htim 触发中断的定时器
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
  static int i = 0;
  if(++i == size)i = 0;
  if (htim->Instance == htim3.Instance){
    __HAL_TIM_SET_COMPARE(&htimx, TIM_CHANNEL_y, sin[i]); //由向量表修改占空比
  }
}
```

对于做电源的同学来说，这个是必须要掌握的内容！

2.7 ADC / SDADC / ADS 模数转化

先介绍最简单的片上 ADC，通常是 12 位，精度则为 3.3/4096 v。

读取 ADC 的方式有很多：

1、轮询

2、中断

3、DMA

因为在实际开发中仅有轮询和 DMA 存在使用场景，所以在这里我仅介绍轮询和 DMA 的方式。

2.7.1 Cube MX 相关配置-轮询方式

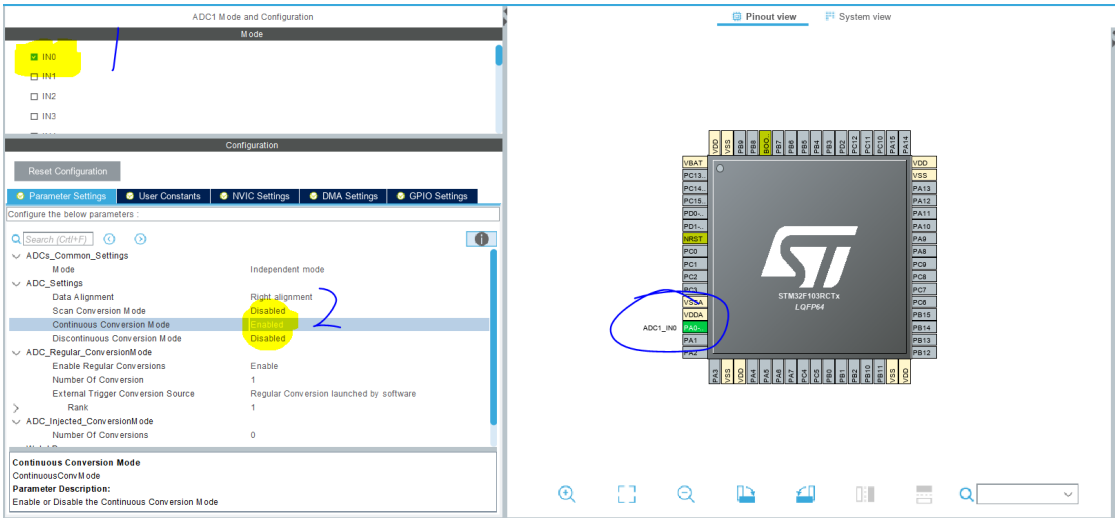

图 2.31: 使能 ADC 引脚

2.7.2 编写业务代码-轮询方式

```
while(1){
  HAL_ADC_Start(&hadc1);//启动ADC装换
  HAL_ADC_PollForConversion(&hadc1, 50);//等待转换完成，第二个参数表示超时时间，
      单位ms.
  if(HAL_IS_BIT_SET(HAL_ADC_GetState(&hadc1), HAL_ADC_STATE_REG_EOC)){
    AD_Value = HAL_ADC_GetValue(&hadc1);//读取ADC转换数据，数据为12位
    printf("[\tmain]info:v=%.1fmv\r\n",AD_Value*3300.0/4096);//打印日志
  }
}
```

前面介绍了通过 ADC 轮询的方式采集单通道的数据。现在介绍一下通过 DMA 方式采集多通道的数据。

2.7.3 Cube MX 相关配置-DMA 方式

2.7.3.1 初始化两个 ADC 通道

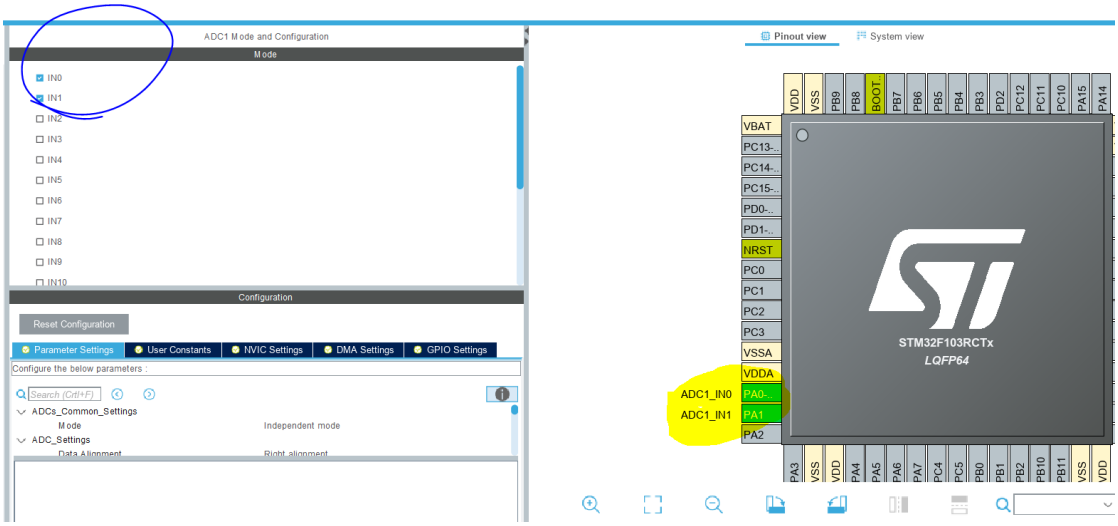

图 2.32: 初始化两个 ADC 通道

2.7.3.2 配置相关属性

step 1： 使能扫描转换模式 (Scan Conversion Mode), 使能连续转换模式 (Continuous Conversion Mode)。

step 2：ADC 规则组选择转换通道数为 2(Number Of Conversion)。

step 3：配置 Rank 的输入通道。

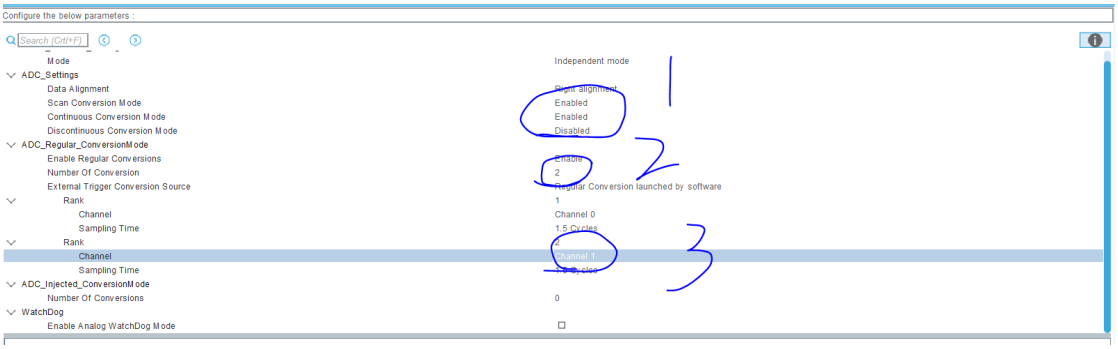

图 2.33: 配置相关属性

2.7.3.3 添加 DMA

添加 DMA 设置，设置为连续传输模式，数据长度为字。

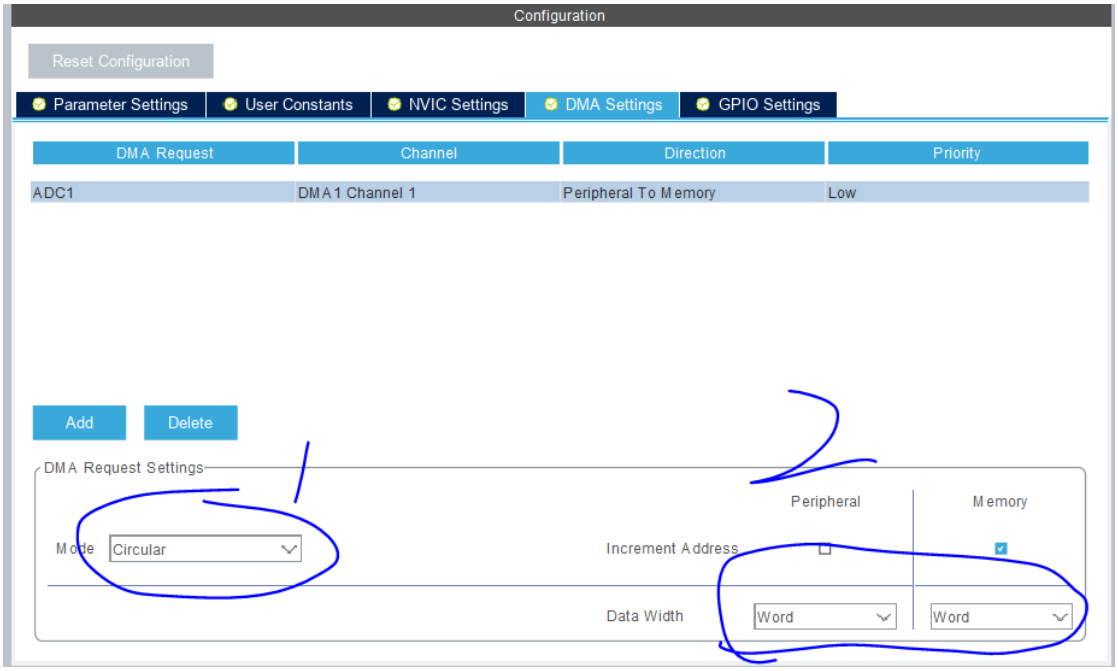

图 2.34: 添加 DMA

2.7.4 编写业务代码-DMA 方式

1、在 main 函数前面添加变量。其中 ADC_Value 作为转换数据缓存数组，ad1,ad2 存储 PA0(转换通道 0),PA1(转换通道 1) 的电压值。

```
/* USER CODE BEGIN PV */
/* Private variables */
uint32_t ADC_Value[100];
uint8_t i;
uint32_t ad1,ad2;
/* USER CODE END PV */
```

2、在 while(1) 前面以 DMA 方式开启 ADC 装换。HAL_ADC_Start_DMA() 函数第二个参数为数据存储起始地址，第三个参数为 DMA 传输数据的长度。

```
/* USER CODE BEGIN 2 */
HAL_ADC_Start_DMA(&hadc1, (uint32_t*)&ADC_Value, 100);
/* USER CODE END 2 */
```

由于 DMA 采用了连续传输的模式，ADC 采集到的数据会不断传到到存储器中（此处即为数组 ADC_Value）。ADC 采集的数据从 ADC_Value[0] 一直存储到 ADC_Value[99]，然后采集到的数据又重新存储到 ADC_Value[0]，一直到 ADC_Value[99]。所以 ADC_Value 数组里面的数据会不断被刷新。这个过程中是通过 DMA 控制的，不需要 CPU 参与。我们只需读取 ADC_Value 里面的数据即可得到 ADC 采集到的数据。其中 ADC_Value[0] 为通道 0(PA0) 采集的数据，ADC_Value[1] 为通道 1(PA1) 采集的数据，ADC_Value[2] 为通道 0 采集的数据，如此类推。数组偶数下标的数据为通道 0 采集数据，数组奇数下标的数据为通道 1 采集数据。

在 while(1) 循环中添加应用程序，将采集的数据装换为电压值并输出。

```
/* USER CODE BEGIN WHILE */
while (1){
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
  HAL_Delay(500);
  for(i = 0,ad1 =0,ad2=0; i < 100;){
    ad1 += ADC_Value[i++];
    ad2 += ADC_Value[i++];
  }
  ad1 /= 50;
  ad2 /= 50;
  printf("\r\n********ADC-DMA-Example********\r\n");
  printf("[\tmain]info:AD1_value=%1.3fV\r\n", ad1*3.3f/4096);
  printf("[\tmain]info:AD2_value=%1.3fV\r\n", ad2*3.3f/4096);
}
/* USER CODE END 3 */
```

程序中将数组偶数下标数据加起来求平均值，实现均值滤波的功能，再将数据装换为电压值，即为 PA0 管脚的电压值。同理对数组奇数下标数据处理得到 PA1 管脚的电压值。

同时 *ADC* 采样也可以采用我之前描述的采用定时器对其平滑滤波！

通常片上的 ADC 的精度往往达不到我们的要求，因为它的精度实在是太低了。有两个替代方案：

1、SDADC, 这个是 STM32F373 上特有的功能，16 位高速 ADC，支持差分输入。掌握难度较大，我也没有很好的掌握，所以就不在此展示了。

2、ADS, 就是外置 ADC。在我们比赛前，我们一直调教的是 ADS1256 这款芯片，能做到 0.01mV 的精度！这类芯片只需要进行 SPI 通信操作，便可以获取 ADC 数据。

2.8 DAC 数模转化

说实话，这两年的开发中，我还没有使用过 DAC 的功能。但是这个功能也十分简单，配置好引脚后，编写业务代码即可。

2.8.1 Cube MX 相关配置

勾选 DAC 中的 OUT Configuration, 其余配置为默认配置不需修改。

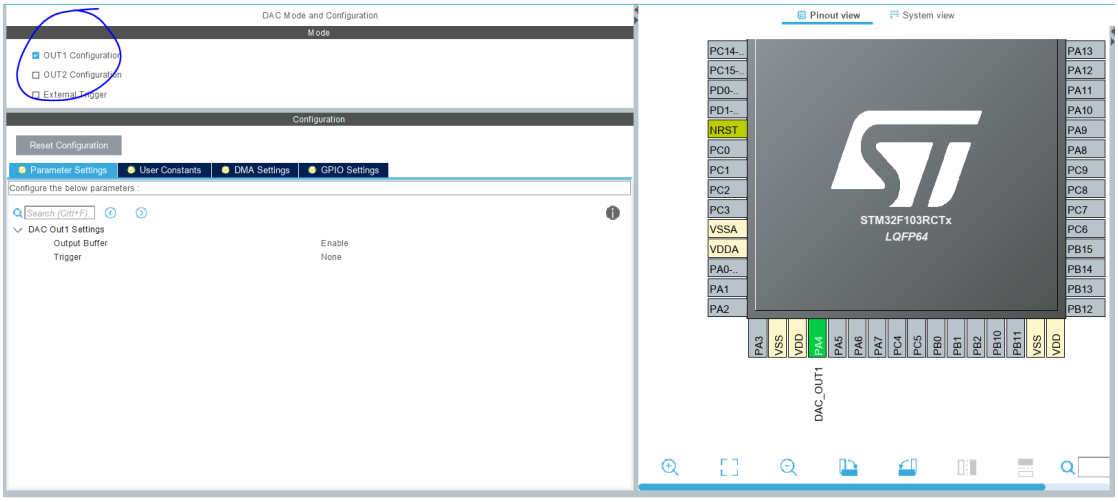

图 2.35: Cube MX 相关配置

2.8.2 编写业务代码

```
//开启DAC转换
HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
// 设置DAC的大小
HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, 2048);
```

编译程序并下载到开发板。如果没有出错用万用表测管脚的电压为 1.65V。

2.9 I2C/SPI

在开发中，使用到 I2C/SPI 的时候通常是与其他模块间的通信，例如：使用 I2C 与 OLED 通信，使用 SPI 与 ADS1256 通信。所以在此情况下，我们只需要在模块现有库函数的基础之上，做少量代码的移植即可。

2.9.1 Cube MX 相关配置-I2C

直接使能 I2C 即可。

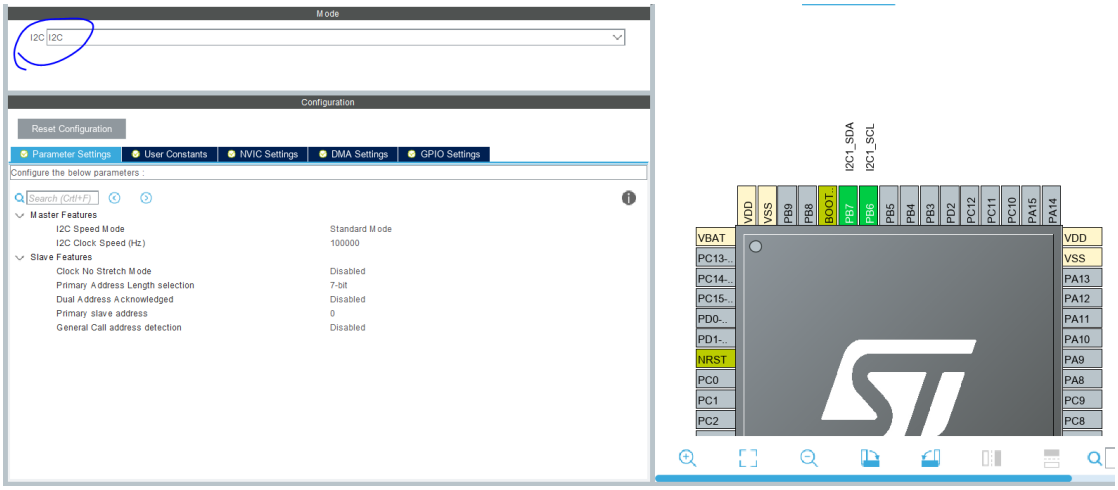

图 2.36: Cube MX 相关配置

2.9.2 编写业务代码-I2C

```
//主机的发送
HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t
    DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);
//主机的接收
HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t
    DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout);
//从机的发送
HAL_StatusTypeDef HAL_I2C_Slave_Transmit(I2C_HandleTypeDef *hi2c, uint8_t *
    pData, uint16_t Size, uint32_t Timeout);
//从机的接收
HAL_StatusTypeDef HAL_I2C_Slave_Receive(I2C_HandleTypeDef *hi2c, uint8_t *pData
    , uint16_t Size, uint32_t Timeout);
```

2.9.3 OLED 的移植-I2C

2.9.3.1 原有库函数的代码

```
...
void I2C_Configuration(void){

  I2C_InitTypeDef I2C_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1,ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB,ENABLE);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
```

```
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_OD;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  I2C_DeInit(I2C1);
  I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
  I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;
  I2C_InitStructure.I2C_OwnAddress1 = 0x30;
  I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
  I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
  I2C_InitStructure.I2C_ClockSpeed = 400000;

  I2C_Cmd(I2C1, ENABLE);
  I2C_Init(I2C1, &I2C_InitStructure);
}

void I2C_WriteByte(uint8_t addr,uint8_t data){

  while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));

  I2C_GenerateSTART(I2C1, ENABLE);
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));

  I2C_Send7bitAddress(I2C1, OLED_ADDRESS, I2C_Direction_Transmitter);
  while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));

  I2C_SendData(I2C1, addr);
  while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));

  I2C_SendData(I2C1, data);
  while (!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED));

  I2C_GenerateSTOP(I2C1, ENABLE);
}

void WriteCmd(unsigned char I2C_Command){
  I2C_WriteByte(0x00, I2C_Command);
}

void WriteDat(unsigned char I2C_Data){
  I2C_WriteByte(0x40, I2C_Data);
}

void OLED_Init(void){
```

```
  DelayMs(100);

...
```

2.9.3.2 移植代码分析

I2C_Configuration 其实就是 I2C 的初始化函数，Cube MX 会帮我们生成，所以直接删除

I2C_WriteByte 被接下来的两个函数依赖，但是 HAL 中有相应的函数，所以直接删除

WriteCmd 和 WriteDat 改写成 HAL 库的方式

DelayMs 改为 HAL 库中的函数

2.9.3.3 移植后的代码

```
...
#include "i2c.h"

void WriteCmd(unsigned char I2C_Command){
  HAL_I2C_Mem_Write(&hi2c1,OLED_ADDRESS,0x00,I2C_MEMADD_SIZE_8BIT,&I2C_Command
      ,1,100);
}

void WriteDat(unsigned char I2C_Data){
  HAL_I2C_Mem_Write(&hi2c1,OLED_ADDRESS,0x40,I2C_MEMADD_SIZE_8BIT,&I2C_Data
      ,1,100);
}

void OLED_Init(void){
  HAL_Delay(100);
...
```

2.9.4 Cube MX 相关配置-SPI

使能 SPI 后，但是需要根据设备的不同做分频处理。

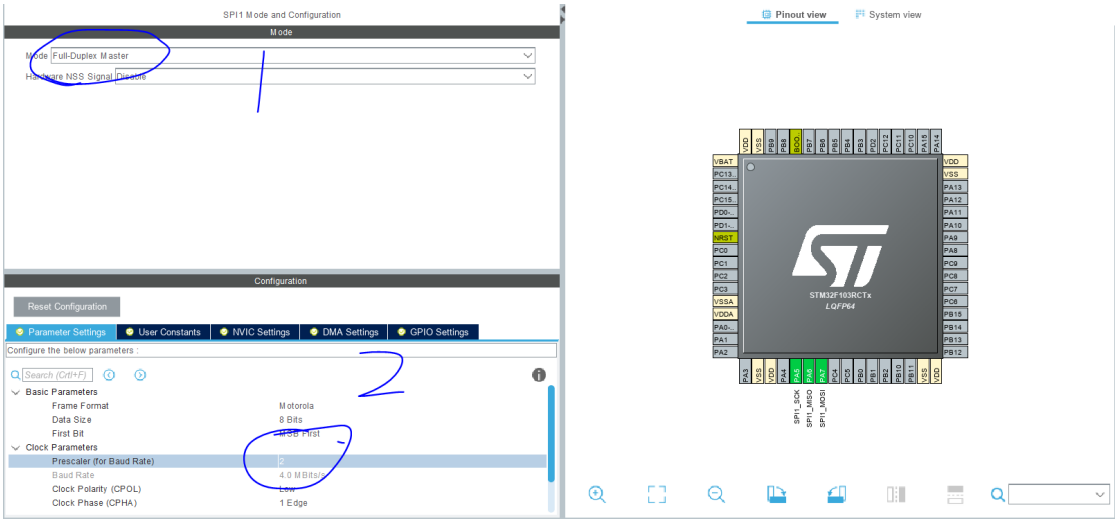

图 2.37: Cube MX 相关配置

2.9.5 编写业务代码-SPI

```
//SPI的发送
HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData,
    uint16_t Size, uint32_t Timeout);
//SPI的接收
HAL_StatusTypeDef HAL_SPI_Receive(SPI_HandleTypeDef *hspi, uint8_t *pData,
    uint16_t Size, uint32_t Timeout);
//SPI的发送和接收
HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t *
    pTxData, uint8_t *pRxData, uint16_t Size,
uint32_t Timeout);
```

2.9.6 ADS1256 的移植-SPI

SPI 的移植比 I2C 的移植要难并且复杂很多，基本上所有的函数都需要做大大小小的改动，建议大家尽量不要自己移植，最好是在网上找到相关的资源。

2.10 FLASH

FLASH 的操作，不需要使用 Cube MX 做任何的配置，只需要做编程操作即可。

2.10.1 相关定义

```
#define BaseAddress ((uint32_t)0x080E0000) // 操作FLAH基地址
//需根据自己单片机的型号进行修改

uint32_t paper_table[100] = {0} ;//需要写入FLAH中的第一张表
uint32_t pwm_table[100] = {0};//需要写入FLAH中的第二张表
```

```
uint32_t length_table = 0;
```

2.10.2 FLAH 的写入

注意:我接下来提供的例程是来自 *STM32F407*, 不同板子间的 *FLASH_EraseInitTypeDef* 可能不同。

```
HAL_FLASH_Unlock();

FLASH_EraseInitTypeDef f;

f.TypeErase = FLASH_TYPEERASE_SECTORS;
//F103中为FLASH_TYPEERASE_PAGES, 即页擦除
f.Sector = FLASH_SECTOR_11;
//F103中为 f.PageAddress = BaseAddress 即开始操作的地址为BaseAddress
f.NbSectors = 1;
//F103中为 f.NbPages = x, 即擦除x页



//设置PageError
uint32_t PageError = 0;
//调用擦除函数
HAL_FLASHEx_Erase(&f, &PageError);
//对FLASH烧写
for(int i = 0; i < 100; i++) {
  HAL_FLASH_Program(TYPEPROGRAM_WORD, (BaseAddress + 4 * i), paper_table[i]);
}
for(int i = 0; i < 100; i++) {
  HAL_FLASH_Program(TYPEPROGRAM_WORD, (BaseAddress + 400 + 4 * i), pwm_table[i
      ]);
}
//锁住FLASH
HAL_FLASH_Lock();
```

2.10.3 FLAH 的读取

FLAH 的读取十分简单，只需要读取相应地址上的值即可。

```
for(int i = 0; i < 100; i++) {
  paper_table[i] = *(__IO uint32_t*) (BaseAddress + 4 * i);
}

for(int i = 0; i < 100; i++) {
  pwm_table[i] = *(__IO uint32_t*) (BaseAddress + 400 + 4 * i);
```

```
}
while(paper_table[length_table] != 0) {
  length_table++;
}
```

2.11 算法-排序

这两个模板，是我用了很久的，通过长时间的测试，我向你保证它是绝对的可靠！

2.11.1 快速排序

```
void quick_sort(int q[], int l, int r){
  if (l >= r) return;

  int i = l - 1, j = r + 1, x = q[l];
  while (i < j){
  do i ++ ; while (q[i] < x);
  do j -- ; while (q[j] > x);
  if (i < j) {
    q[i] = q[i]^q[j];
    q[j] = q[i]^q[j];
    q[i] = q[i]^q[j];
  }
  else break;
  }
  quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
```

2.11.2 归并排序

```
void merge_sort(int q[], int l, int r){
  if (l >= r) return;

  int mid = l + r >> 1;
  merge_sort(q, l, mid);
  merge_sort(q, mid + 1, r);

  int k = 0, i = l, j = mid + 1;
  while (i <= mid && j <= r)
  if (q[i] < q[j]) tmp[k ++ ] = q[i ++ ];
  else tmp[k ++ ] = q[j ++ ];
```

```
  while (i <= mid) tmp[k ++ ] = q[i ++ ];
  while (j <= r) tmp[k ++ ] = q[j ++ ];

  for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

2.12 算法-MPPT

在做最大功率点追踪，这个算法是十分重要的。我在这里分享一下我是怎么对其优化的，首先我写了一个能实现功能的最基础的版本。

2.12.1 基础版本

```
#include "mppt.h"
#include "main.h"
#include "usart.h"

//上一次的功率
double l_power = 0.0 ;
//功率上升或下降
int updown = 0;
//步长
int MPPT_STEP = 160;

/* 扰动法计算
*
*/
int mppt_po(double u, double i, int pwm) {

  double power = (u * i) < 0 ? 0 : u * i ;

  printf("[Info]mppt_po:当前电流:%f,当前电压:%f,当前功率: %f\r\n", i, u, power);

  if(power < l_power || power == 0) {
    updown ^= 1;
    printf("[Info]mppt_po:当前功率:%f,小于此前功率: %f\r\n", power, l_power);
  }

  if(updown) {
    printf("[Info]mppt_po:PWM:%d调节为:%d\r\n", pwm, pwm + MPPT_STEP);
    pwm += MPPT_STEP;
```

```
  }
  else {
    printf("[Info]mppt_po:PWM:%d调节为: %d\r\n", pwm, pwm - MPPT_STEP);
    pwm -= MPPT_STEP;
  }

  printf("[Info]mppt_po:该次调节结束\r\n");

  pwm = pwm < 0 ? 0 : (pwm >= 1599 ? 1599 : pwm);

  l_power = power;

  return pwm;
}
```

2.12.2 算法的不足与解决方案

1、这个算法一直在调节，这很有可能造成能量的损耗。解决方案：采用标志位，判断是否稳定。

2、这个算法步长不变，从头到尾固定步长。如果步长太长不精细，如果步长太短整体调节较慢。解决方案：采用可变步长。

3、使用三目运算符取代大量的 if-else

4、Log 信息采用条件编译的方法（很早之前写的，没能采用串口的终极解决方案，有点遗憾）

5、抽离变量，写出结构体，配以初始化函数。

2.12.3 最终版本

```
/**
  * @file : mppt.c
  *
  * @brief: MPPT 最大功率点追踪
  *
  * @auther : Reyunn
  *
  */
#include "mppt.h"
#include "main.h"
#include "usart.h"

extern void quick_sort(int q[], int l, int r);
```

```
typedef struct {
  double l_power;//上一次的功率
  uint8_t updown;//功率上升或下降
  int max_step;//步长
  int min_step;//步长
  int pwm_max ; //pwm最大值
  int pwm_min ; //pwm最小值
  uint8_t count; //计算微调次数
  uint8_t state; //状态
  uint8_t time; //改变方向次数
  int l_pwm[10];
} MPPT;

MPPT mppt;

/**
* @brief mppt 初始化
* @param l_power : 上次测量功率
* @param updown : 上升或下降
* @param min_step : 最小步长
* @param max_step : 最大步长
* @param pwm_max : pwm最大值
* @param pwm_min : pwm最小值
* @retval None
*/
void mppt_init(double l_power, uint8_t updown, int min_step, int max_step, int
    pwm_max, int pwm_min) {
  mppt.l_power = l_power;
  mppt.updown = updown;
  mppt.max_step = max_step;
  mppt.min_step = min_step;
  mppt.pwm_max = pwm_max;
  mppt.pwm_min = pwm_min;
  mppt.count = 0;
  mppt.state = 1;
  mppt.time = 0;
}

/**
* @brief 扰动法计算
* @param u : 当前电压值
* @param i : 当前电流值
* @param pwm : 当前pwm值
```

```
* @retval 计算后的pwm值
*/
int mppt_po(double u, double i, int pwm) {

  double power = (u * i) < 0 ? 0 : u * i ;

  if(mppt.state) {

#if Log
  printf("[info]mppt_po:当前电流:%f,当前电压:%f,当前功率%f\r\n", i, u, power);
#endif

  if(power < mppt.l_power || pwm == mppt.pwm_max || pwm == mppt.pwm_min) {
    mppt.updown ^= 1;
    mppt.time ++;

    if(mppt.time > 5 && power < mppt.l_power ) mppt.l_pwm[mppt.count++] = pwm;
#if Log
    printf("[info]mppt_po:当前功率:%f,小于此前功率%f\r\n", power, mppt.l_power);
#endif
  }
  pwm = (mppt.updown == 1) ? ((mppt.count > 0) ? pwm + mppt.min_step : pwm +
      mppt.max_step) : ( (mppt.count > 0 ) ? pwm - mppt.min_step : pwm - mppt.
      max_step);

  pwm = (pwm < mppt.pwm_min ) ? mppt.pwm_min : ( (pwm >= mppt.pwm_max) ? mppt.
      pwm_max : pwm);

  mppt.l_power = power;

  if(mppt.count == 10) {
    mppt.state = 0;
    quick_sort(mppt.l_pwm, 0, 9);
    return ( mppt.l_pwm[5] + mppt.l_pwm[4] + mppt.l_pwm[3] + mppt.l_pwm[6] ) /
        4;
  }
  return pwm;
  } else {

    if( power - mppt.l_power > 1000 || power - mppt.l_power < -1000 ){
    mppt.count = 0;
    mppt.state = 1;
    mppt.time = 0;
```

```
#if Log
    printf("[info]mppt.c:进入调整模式\r\n");
#endif
  }
  return pwm;
  }

}
```

2.13 后记

这个 STM32 部分的内容，到目前（2019 年 8 月 26 日）为止共有两次改动。

v1.0.x：第一次版本。

v1.1.x：在第一次的基础上，有以下改动：

1、规范目录结构。

2、增多串口通信、ADC 内容。

3、增加 I2C/SPI 及 FLASH 的内容。

4、删除 Astyle 等内容。

同时这个部分的内容，也是作为信息学院 2019 年暑假集训的讲义，期间大家相互学习。在和电子科技协会全体会员的一起努力下，我们共同录制了一套 CubeMX + MDK5 + HAL 库 + 库函数一站式学习 STM32 的学习视频。

图 2.38: STM32 系列视频

如果你想仅仅学习 HAL 库就完全掌握 STM32 那是不可能的，在此之前你必须对寄存器、库函数这两种开发方式有一定的了解。

但是如果学习到此你就认为你完全掌握了 STM32 那也是不可能的，往后还有 DSP、FreeRTOS 等需要你学习。

第 4 章　杂谈

这个部分，则是对大学这两年避不开的话题谈谈自己的经历及看法。还是那句话，如有错误，欢迎指正，如有帮助，不胜荣幸！

4.1 读研

对于这个议题，我的看法是“不要人云亦云，有自己独立的思考”。我想我自己可能没有资格谈论这个话题，因为我也不是过来人。所以我摘录 laike9m 的一篇《所以，到底要不要读研?》希望对大家有帮助：

过去几年，我零散地表达过一些对读研的看法。鉴于最近有人感兴趣，我也正好写篇文章总结一下。本文所说的“读研”，特指在中国境内读三年制硕士研究生。这也符合大多数人的情况。本文所谈论的“读研”不包括：读博、读两年制项目、在职研究生、出国读书等等。

我对读研怎么看？用一句话概括就是：**如果你明确知道读研就是达到既定目标过程中所缺失的那个条件，那就去读。否则不要读。**

在展开讨论之前，我必须先讲一个致命的思维盲区：**权衡要不要做某件事的时候，只看到做这件事的收益，而完全不考虑因此导致无法做别的事所造成的损失**。以读研为例。有人说“读研提升了学历，增加了进大企业的机会，好处太多了，一定要读”。True，我承认有这些好处。问题是，你把这些好处放在天平的一端，另一端放什么？如果什么都不放，那最后的结论当然是读研好。但应该这样比吗？不该啊。要放在天平另一端作为比较对象的，不是三年前的你，而是没读研去做了其它事的那个你。一个人**要判断的，不是读研带来了多少收益，而是读研和其它选择哪个收益更大**。这才是正确的思考方式。

有了上面这些基础，我们来列举几种常见的值得读研的情形。

情形一，你想去的地方，对学历有明确要求。包括但不限于积分落户、某些事业编制、男/女朋友的家人要求你必须是硕士等。没什么好说的，去读就行了，毕竟你没法改变规则。

情形二，你想去的地方，对学校有明确要求。比如某岗位要求必须是 211 及以上，而你之前只是普通一本。这条看似和上一条类似，但其实不一样，后面会说。

情形三，你想转行。比如误入了天坑专业的学子，想完全依靠自己力量跳出去实在是太难了。用人单位看到你本科专业，可能直接就把简历扔了。去读个对口专业的研究生会好很多。

情形四，你想去的地方，拥有硕士学历能带来更好的发展。诸如政府机关、国企、军队，唯学历论依然盛行，本科生和硕士可能会被分配完全不同的就职岗位和培养路径。

当然，上面只是几个例子，无法涵盖全部情况。归纳起来还是最开始说的：如果你明确知道读研就是达到既定目标过程中所缺失的那个条件，那就去读。

然而，大部分人连目标都不确定，只不过是把读研当做一种改变命运的美好幻想罢了。幻想的形式包括但不限于:

1. 谁说计算机从业人员就只有程序员、进公司 code 这一条路了，选择读研读博继续深造的，可能是想在计算机前沿领域进行研究，以后仍然可以选择进公司或者出国留学然后留校留研究院搞科研学术。

> 读博我们这里不谈，但你跟我讲读研深造？我就想知道第一年上课最后一年写毕业论文找工作可能还有实习请问你要怎么搞研究？把学术界当儿戏以为成果能随便就能出？你想读研留校搞学术，我还想去二次元开后宫呢，好不好？

2. 因为你儿子 /女儿以后可以跟同学说我爸不是个纯 code 屌

> 笑尿了，读研就不是“纯 coder”，就高大上了。这种话简直不值一驳，能说出这话的人可以想象是处在怎样一种封闭无知的圈子里。

3. 控制变量，能力相等的情况下，有研究生学历的你，和没有研究生学历的你，哪个在别人眼中更优秀？

> 这个还是有必要说一下。对于能提升学校档次的考研，我完全支持，但首先你得确定这是你需要的，其次是你得去一个真正提升了档次的学校。你说我从xx理工学院城市学院考研到xx化工学院科亚学院，读不读又有什么区别呢？还不如去积累工作经验。

4. 看看今年研究生算法岗的神仙打架和工资....

> 不好意思，我就没见到哪家公司的岗位指定了要研究生，因为他们知道学位不等于能力，一个 NB 的本科生可以顶十个水货硕士。

5. 能考上研究生的，整体水平要比本科水平高。不要把公司当傻子，研究生这个门槛天生就隔绝了能力水平稍低的人。

> 不值一驳。

6. 因为太菜了，纯写代码竞争不过各位大佬。

> 所以你读完研就不菜了？读研不能让你脱胎换骨变成另一个人，醒醒吧。

7. 大部分人平庸到谈不上靠能力争取到话语权，所以只能尽力找一个块儿厚点的敲门砖。

> 这句话不能说没道理，但首先，你总得先选好想进的门吧？

8. 可能你眼中的程序员只是日常 CRUD 吧

> 嗯，研究生牛逼，做的都是高大上的工作，CRUD 什么的太没技术含量啦。

9. 样本虽然小，但是我身边确实就学历和经验来看，学历高的比经验足的后劲更大。前几个月刚毕业来公司的交大小伙，现在熟悉业务了代码写得嗖嗖的，又好又快，而且产品那边一些新需求他也能就着英文文档去啃出来。而另一位 3 年经验的专科同事，虽然解决问题挺熟练，但是文档几乎不会啃，全靠简书和 csdn 里面的博客

对着写，有些生僻点的玩意百度找不到，他那边就 gg 了，更不用说数据处理，除了日常的增删改查，其他的复杂的数据处理他那边就会僵住.

> 您厉害，这变量控制得真好。他厉害不是因为他是研究生，是因为他是交大的啊。

10. 作为一名研究生应届毕业生，来说说我对此问题的看法 1. 研究生选择更多。可以选择去做科研，做教师，工作职位可选择的也多，比如女生不想做研发可以做测试等等 2. 发展空间更大。个人认为学历还是很重要的，尤其是在以后职位的晋升上 3. 思维方式和学习更力。研究生期间，学习到更多的应该是看问题的方式，对行业的见解等等，而不仅仅是码代码 4. 起点更高。更容易去大公司，起薪也会更高

> 1 不说了。2 是一种很奇怪的误解，因为我发现老一辈人（比如我妈）真的会这么想。但实际上除了少数地方比如国企，大部分工作并不存在本科生发展空间受限一说，尤其当你讨论的还是 CS 专业的时候。3 有一定道理。4 我要重点反驳一下，这就是前面提到的思维盲区的典型。研究生的你，的确更有可能拿到比本科的你更好的 offer。然而，和工作了三年的你相比呢？三年时间，快的话可以涨薪两次或是跳槽两次了，在大公司的话升一级没问题，快的话可以升两级，更不要提工作经验的积累了。别忘了学校和公司可是完全不同的。

11. 程序员只是青春饭，你会发现上了年纪就不行了

> 读研之后你老了三岁，能吃这口饭的时间更少了。

12. 校招大厂来面试的大多数是研究生，足以说明问题

> 看来 Google 不是大厂了。再说进了公司你就会发现研究生和本科生干活真的没什么差别。

这些幻想要批判起来说一年也说不完。不过相信大家也发现了一些共性，就是这种盲目推崇读研的人，往往对很多问题都缺乏基本了解，看问题也非常片面。我暗自揣测，他们中大部分可能并没有真正读过研究生，却又把自己目前的不如意归咎于没有读研，并幻想出了所谓读研之后的美好生活。我在计算所的时候，周围很多同学都觉得读研浪费时间，也包括我在内。有个哥们实在受不了实验室安排的无聊工作直接退学然后面试进了头条，人家也没嫌他学历不够。总之呢，读研这件事好不好，各人有各人的情况。对想进互联网行业的同学，可能确实是浪费时间，但若是想去考公务员或者拿户口，可能又很必要。关键还是要清楚自己的目标，分析自身情况，再来判断读研到底是不是一个好的选择。

写这么多，想说的基本都说了。我试图保持客观，但也并不想掩饰对读研的负面看法。说真的，如果国内的研究生学制也是像国外那样是一年或者一年半，我断然是不会写这些的，因为读了也就读了。然而，它是三年，还是你人生中非常宝贵的三年。三年时间是真的不短啊。好好思考一下三年你能做什么，是否值得用这段时间去换一张文凭。我希望所有人都能做出令自己不后悔的选择。

以上就是文章的全部，既然我没有资格谈自己的看法，那我就讲讲自己的一个经历好了。我记的在大二的上学期，那时候是秋招的时间。思科公司来我们学校招聘，招聘

的教室就在我们实验室的旁边，思科的某位领导可能是来早了，到处逛逛，就进了我们的实验室。在闲谈中长者就提到了：“我们公司招聘不会把学历放在第一位，更看重个人能力，本科生中也有很多能力强的。在中国的学术环境的影响下，读研并没有起到应有的作用”。

图 4.1: 神农架之行

4.2 终生学习

好吧，还是要继续摘录其他人的文章了。先带大家看一篇 Summer 所写的《每一个编程从业者都应该是「终身编程者」》：

程序员是很棒的职业

世界是由软件构成的，而程序员是撰写软件的人。

在未来，很多职位会消失，这是因为计算机和软件可以取代它们。但是从另一个角度看，因为我们需要不断开发和维护这些程序，所以这么一想，程序员的前景还是很美好的。

即便你不把编程当成职业，也可以拿编程来解决生活中的问题，以工程师的思维来思考这个世界，并尝试去优化自己的生活。

编程是一辈子的

从决定把编程作为职业开始，我就告诉自己，编程应该搞一辈子。为啥？因为这个职业对经验和学习能力要求太高了，隔语言如隔行，得无时不刻地学习，没有经验还找

不到工作。如果决心不够坚定，自己肯定会很难混下去。所以只要入这个行，不管十年后是否从事编程的工作，编码都应该是一辈子的。

有了这个定位，就不怕自己学的东西太广太泛了，脑子里会想：「我这是在打地基」。遇到编程问题时，也会去寻找最佳实践，会带着长久发展的思维去思考。例如说不把自己的专业当成 PHP Web 开发，而是计算机科学，会主动去学习软件工程。

把编程当成终身职业时，工作也会变成学习进步的途径，而不是艰苦地讨生活。业余时间用来学习和编码，也会变得名正言顺。随着而来的好处是，同样几年过去了，你比身边的同事要经验丰富得多。

上面的文章是来自 learnku 社区站长的一篇文章。一入编程深似海，不断学习成必然。就笔者写作的年代来看，是处于技术不断革新，基本上每月都有新的技术、新的概念提出来，意味着程序员学习的东西不断的累加。但是就目前的趋势来看，现在的开发都讲究自适应，也就是一套代码在各大平台都能完美的运行。这样自然是大大减少了程序员的学习成本。也许未来能出现一统多端的语言，可能是 Dart 也可能是 JavaScript，也可能是其他的。

4.3 奖学金

首先努力之后获得成绩是值得肯定的。我们学校的奖学金的有很多，同时制度也相对健全，只要满足要求就能申请。我也通过自己的努力凭大一的成绩获得了奖学金。但是我持有以下观点“努力学习是为了追求卓越，奖学金只是在追求卓越路上的小小的奖励”。

4.4 论坛

我是一个喜欢逛论坛的人。刚上大学的时候接触到的两个论坛：知乎、CSDN。现在都不怎么逛了，首先知乎上通常是长篇大论读起来太费劲，其次自己没达到知乎的平均水平-动不动就是《月薪 5W 是一种怎样的体验》，最后知乎并不是技术性的论坛。对于 CSDN 我认为他的技术文章参差不齐，且种类繁多，所以我一种把它当作查资料的工具在用。

现在我逛的论坛有很多，下面给大家做推荐（软件学习相关）：

1. GitHub：https://github.com/
   全球最大的同性交友平台，优秀的代码托管平台。
2. 掘金：https://juejin.im/
   文章质量高！针对性性强。
3. V2EX：https://www.v2ex.com/
   这个论坛没有太多技术性的文章。喜欢它的原因有两个：“酷工作”板块十分活跃，在这个版块下可以看到公司的招聘需求，也可以看到并下载其他人的简历。其次它更像是程序员的休闲灌水群。

4. SegmentFault: https://segmentfault.com/

   有着中国的 StrakOverFlow 之称，是一个问答性质的平台。同时思否学院中的课程质量较高。

此外还有 learnKu、简书等优秀的平台。

图 4.2: 发际线

4.5 竞赛

大学里面的竞赛有很多，电子设计大赛只是其中之一。但是并不是每一个都适合你参加，所以择其善者而攻之。想参加比赛不要着急，首先你要有技术的积累，只有你自身实力够强，老师才愿意带你，同学才愿意找你。准备比赛的时间应从进校就开始，但是参加比赛的时间应该是大二下的寒假才开始。当你进入大二下，你就是学校参加比赛的主力选手了。以下我只介绍几个我所了解的：

那年寒假有“全国大学生创新创业训练计划”的申报，注意不是比赛，只要完成最后的答辩结题即可，且通过概率极高。大家一定要把握好这个机会。这个比赛能获得保研加分，但是不具有保研资格。想要获取保研资格还需通过学生手册上列举的比赛。

然后大二暑假就有“全国大学生电子设计大赛”，这个比赛在信息学院和机电学院参与的人数很多，比赛的方向也很多。需要找好队员，和选好参赛方向。电子设计大赛总的来说分为五类：“电源类”、“测控类”、“信号类”、“高频类”、“无人机”。其中参与人数最多的就是“电源类”与“信号类”。无论是那一类都不简单，都需要做好长期奋斗的准备。

在大三的要参加准备以久的“挑战杯”和“节能减排”了，这可是竞赛里面的两大巨头。做好了你就不用努力了，做不好你可能“人财两空”。之所以说“准备以久”，意味着这个项目一定是经过长时间的孵化，并且经历过许许多多的答辩历练。在我写下这篇文章的时候，这两个比赛都还在继续推进。我这一年的“挑战杯”是创业类的比赛，也

就是“小挑”。这对我们信息学院的学生并不是十分有利，但是我们都在尽力做，并且对自己有信心。

图 4.3: i 创杯

我的看法就是所有比赛的选题还是要有老师的指导比较好，最好是老师正在研究的课题。所以，在此再次感谢所有帮助过我的老师。

4.6 社团

最后一个话题，我选择留给协会。

此前我一直有个遗憾，就是自己没能加入一个小团体的社团，面试啥啥失败。但是，“电子科技协会会长”的身份就像一份礼物一样摆在了我的面前。

这段往事有点不堪，但是我还是想把它写出来。在我的上一届，我认为是做的不够好的，糟糕到换届的时候从头至尾找不到人竞选。我有个很强势的女同学，是其中的干部，也是原本的会长候选人。她在换届的前一天找到我，想让我担任副会长，我充满疑惑的接受了。也在同一天晚上，“协会会长”这份礼物就送到了我的面前，既然大家都不想干那我就干。

我当会长的第一年是从 2018 年暑假开始的。先自我评价一下这一年，“很烂，承诺的事情没做到”。毕竟是第一次当会长，既没有示范，也没有实习。当时我承诺的是一个学年将举办 4 次活动，也只举行了一次，会员大会啥的都没有。因为我陷入了一个巴掌拍不响的局面。

协会又混混噩噩到了换届的时候，此时我大可拍拍屁股走人，大不了被别人背后小声的骂几句。但是我是不甘心的，我有野心要把协会做好，要做到五星级社团。于是我

选择再干一年，如果干不好那就再干一年。此时，我不在是一个人奋斗，我有了一群志同道合的伙伴。那么这一年，我要为协会未来 5 年的发展奠定基础，没有的东西我们从头建设，已有的东西发扬传承。

这里我想说明一下，这是我的个人文章为什么要已协会的名义发表呢？我想我把电子科技协会当成了自己的家，家是付出而不是索取，同时我也想这个家能留下我的印记。

电子科技协会，在奔跑。