

# 目录

第一章 stm32 .....	7
1.1 引言 .....	7
1.1.1 STM32 简介 .....	7
1.1.2 STM32 与 Linux 系统的比较 .....	7
1.2 STM32F103ZET6 开发概述 .....	7
1.2.1 STM32 系列微控制器的命名方式 .....	8
1.2.2 STM32F103ZET6 开发板介绍 .....	9
1.2.3 STM32F103ZET6 参考文档 .....	10
1.2.4 STM32F103ZET6 开发方式 .....	10
1.2.5 STM32 固件库介绍 .....	11
1.3 开发环境搭建 .....	12
1.3.1 keil 软件安装 .....	12
1.3.2 CubeMX 软件安装 .....	19
1.3.3 驱动文件安装 .....	24
1.4 STM32 启动过程 .....	26
1.4.1 系统架构 .....	26
1.4.2 存储器的组织架构 .....	27
1.4.3 stm32 启动方式 .....	29
1.4.4 启动文件 .....	29
1.4.5 启动文件分析 .....	30
1.5 STM32 系统时钟树 .....	31
1.5.1 系统时钟的时钟源 .....	31
1.5.2 时钟树 .....	31
1.5.3 系统时钟分析 .....	33

1.6 STM32 通用输入输出 .....	35
1.6.1 GPIO 框图 .....	36
1.6.2 GPIO 的八种工作模式 .....	36
1.7 使用 CubeMX 配置使用 GPIO.....	41
1.7.1 新建工程 .....	41
1.7.2 配置时钟树 .....	43
1.7.3 配置 GPIO .....	43
1.7.4 配置工程管理.....	47
1.7.5 生成工程.....	48
1.7.6 keil5 打开工程文件 .....	48
1.7.7 MX_GPIO_Init 函数解析.....	48
1.7.8 将文件烧录至 STM32ZET6 开发板 .....	49
1.7.9 实验现象.....	52
1.8 工程详解 .....	52
1.8.1 工程目录 .....	52
1.8.2 MDK-ARM 工程文件介绍.....	53
1.8.3 主程序结构详解.....	54
1.9 GPIO 相关 API .....	57
1.9.1 相关 API .....	57
1.9.2 hal 库实现跑马灯 .....	60
1.9.3 作业：使用 GPIO 驱动蜂鸣器.....	61
1.10 stm32 嵌套向量中断控制器（NVIC） .....	61
1.10.1 NVIC 的介绍 .....	61
1.10.2 NVIC 的优先级 .....	61
1.10.3 Cubemx 配置中断优先级 .....	63

1.10.4 中断的具体行为 .....	63
1.10.5 外部中断的介绍 .....	64
1.10.6 中断的主要特性 .....	65
1.10.7 外部中断/事件线路映像 .....	65
1.10.8 中断相关 API 及数据结构 .....	66
1.11 按键中断 .....	69
1.11.1 CubeMX 基础配置 .....	69
1.11.2 中断模式配置 .....	70
1.11.3 工程代码分析 .....	71
1.11.4 回调程序设计 .....	74
1.12 定时器 .....	74
1.12.1 定时器介绍 .....	74
1.12.2 定时器计时公式 .....	75
1.12.3 定时器结构体 .....	75
1.13 定时器时间基准模式（Time Base） .....	78
1.13.1 时间基准模式概述 .....	78
1.13.2 时间基准模式相关配置参数 .....	78
1.13.3 时间基准模式相关 API .....	80
1.13.4 hal 工程函数详解 .....	83
1.13.5 定时器使灯光闪烁 .....	87
1.14 定时器输出比较模式（Output Compare） .....	88
1.14.1 输出比较模式概述 .....	88
1.14.2 输出比较模式模式相关参数 .....	88
1.14.3 输出比较模式模式相关 API .....	89
1.14.4 PWM 实现呼吸灯 .....	92

1.14.5 SG90 伺服电机基本原理 .....	97
1.14.6 PWM 驱动舵机 .....	98
1.15 定时器输入捕获模式（Input Capture） .....	101
1.15.1 输入捕获模式概述 .....	101
1.15.2 输入捕获模式模式相关配置参数 .....	101
1.15.3 输入捕获模式模式相关 API .....	101
1.15.4 电容触摸按键原理 .....	104
1.15.5 电容按键控制 LED 亮灭 .....	105
1.15.6 作业 .....	116
1.16 串口通信 .....	116
1.16.1 串口通信概述 .....	116
1.16.2 串口通信时序 .....	117
1.16.3 串口通信结构体详解 .....	118
1.16.4 串口通信基础参数 .....	121
1.16.5 串口 IO 相关 API .....	123
1.16.6 串口轮询模式实现回显 .....	131
1.16.7 串口中断模式控制 LED .....	135
1.16.8 串口 DMA 模式控制舵机 .....	138
1.16.9 作业 .....	140
1.17.1 单总线通信 .....	140
1.17.1 单总线通信概述 .....	140
1.17.2 DHT11 概述 .....	141
1.17.3 DHT11 时序 .....	141
1.17.4 读取 DHT11 数据并串口显示 .....	142
1.18 I2C 通信 .....	146

1.18.1 I2C 通信概述 .....	146
1.18.2 ssd1306 屏幕概述 .....	147
1.18.3 ssd1306 I2C 通信分析 .....	147
1.18.4 ssd1306 I2C 写入程序的实现 .....	148
1.18.5 ssd1306 I2C 初始化程序 .....	151
1.18.6 ssd1306 内存寻址模式 .....	152
1.18.7 ssd1306 显示程序分析 .....	155
1.18.8 显示“千锋欢迎你” .....	156
1.18.9 作业 .....	160
1.19 SPI 通信 .....	160
1.19.1 SPI 概述 .....	160
1.19.2 W25Q16 存储芯片概述 .....	162
1.19.3 操作示例时序图 .....	166
1.19.4 W25Q16 读写程序分析 .....	169
1.19.5 循环读写实验 .....	180
1.20 看门狗 .....	181
1.20.1 看门狗概述 .....	181
1.20.2 独立看门狗 .....	181
1.20.3 独立看门狗案例 .....	185
1.20.4 窗口看门狗 .....	187
1.20.5 窗口看门狗案例 .....	193
1.21 模数转换 ADC .....	197
1.21.1 ADC 概述 .....	197
1.21.2 ADC 结构体详解 .....	197
1.21.3 ADC 相关 API .....	199

1.21.4 ADC 测量电压.....	206
1.21.5 ADC 使用光敏/热敏测量光照/温度.....	210
1.21.6 STM32 大作业.....	212

# 第一章 stm32

## 1.1 引言

### 1.1.1 STM32 简介

STM32 是 STMicroelectronics 推出的一系列 32 位微控制器，基于高效的 ARM Cortex-M 系列处理器。这些处理器采用 32 位架构，提供出色的计算能力和响应速度，适用于多种嵌入式应用。

STM32 微控制器的主要特点包括其多样的处理器核心选择，如 Cortex-M0, M3, M4, M7 等，满足不同的性能和功耗需求。其低功耗设计特别适合电池供电和能源敏感的应用，通过多种节能模式降低待机时的能耗。

此外，STM32 提供丰富的外设和接口支持，包括多种通信接口（如 UART, SPI, I2C），模拟接口（如 ADC, DAC），以及复杂的定时器和 PWM 输出等。这些特性使其能够处理复杂的多任务操作和高级控制逻辑。

内存和存储方面，STM32 系列提供不同的配置选项，包括 RAM 和闪存，同时支持外部存储设备。这些灵活的内存选项使 STM32 能够适应从简单的控制任务到需要大量数据处理的复杂应用。

STM32 的另一个优势是其集成的开发环境和工具支持，包括多种 IDE 选项和 STM32CubeMX 配置工具，简化了开发过程。结合这些特点，STM32 成为了嵌入式系统设计中的一种受欢迎的选择，适用于从工业自动化到消费电子等多个领域。

### 1.1.2 STM32 与 Linux 系统的比较

STM32 作为一个微控制器，与基于 Linux 的系统在设计和应用方面存在明显差异。STM32 主要用于直接控制硬件和执行特定任务的嵌入式应用，而 Linux 系统通常用于更复杂的计算任务，需要较大的处理能力和内存。

在操作系统方面，STM32 通常不运行传统的操作系统，而是执行裸机代码或实时操作系统（RTOS）。这使得 STM32 在响应时间和资源占用上更高效，适合实时性要求高的应用。相比之下，Linux 是一个完整的操作系统，提供了丰富的功能和服务，但也因此需要更多的资源，如处理器性能和内存。

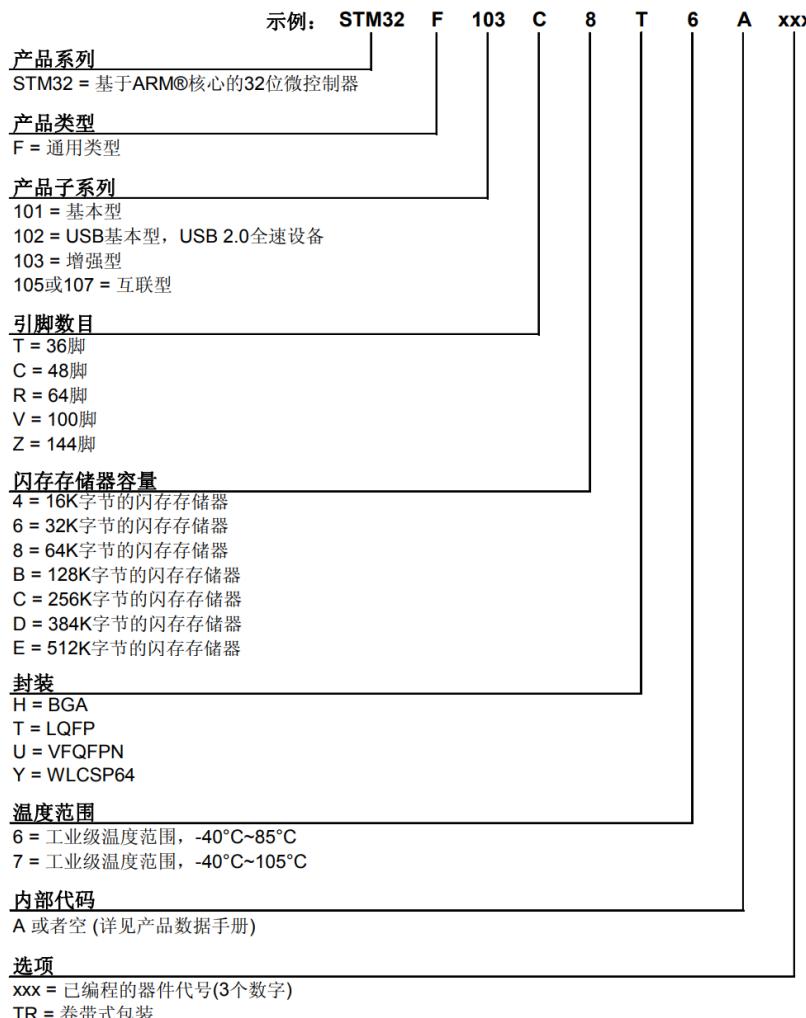
从应用角度来看，STM32 适用于需要精确时间控制和资源约束的场合，如传感器数据采集、电机控制等。而 Linux 系统更适合需要复杂数据处理、网络通信和用户交互的应用，如服务器、桌面计算等。

## 1.2 STM32F103ZET6 开发概述

STM32F103ZET6 作为 STM32 系列中的高性能微控制器，其开发具有显著的优势。凭借其强大的 ARM Cortex-M3 内核，这款微控制器提供了出色的处理速度和计算能力，非常适合要求高性能和实时处理的应用。同时，它还具备丰富的外设接口和

灵活的内存配置，使得开发者可以轻松实现复杂的多功能设计。此外，STM32F103ZET6 的低功耗特性和成本效益使其成为一个理想的选择，适用于各种规模的项目，从简单的家用设备到复杂的工业应用。

### 1.2.1 STM32 系列微控制器的命名方式



STM32F103ZET6

"STM32": STMicroelectronics 公司的产品，基于 ARM Cortex-M 内核的 32 位微控制器。

"F": 表示该微控制器属于 STM32 的 F (Foundation) 系列。

"103": 表示该微控制器属于 103 系列，这一系列的微控制器具有相对较高的主频，丰富的内部外设以及适中的内存容量。

"Z": 表示该微控制器 144 引脚。

"E": 512K 闪存。

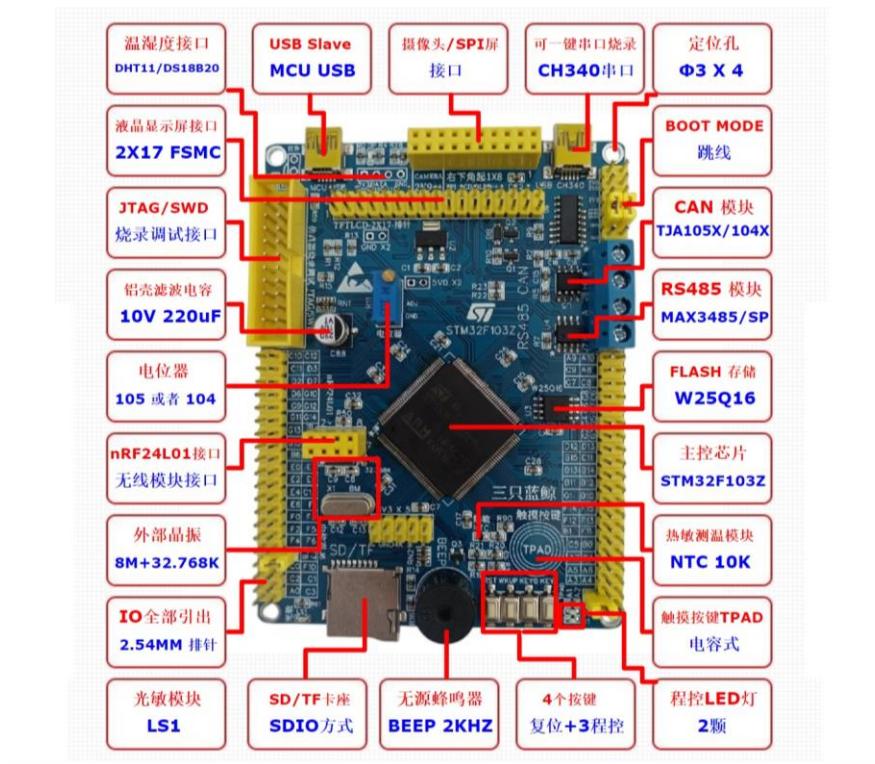
"T": 表示该微控制器的封装类型是 LQFP。

"6": 表示该微控制器的工作温度范围是工业级别，即-40°C 到 85°C。

### 1.2.2 STM32F103ZET6 开发板介绍

开发板的核心由 STM32F103ZET6 微控制器构成，它被各类接口和模块环绕，旨在提供一站式的开发体验。板上不仅包括 USB 从设接口和 CH340G USB 转串口芯片等标准通信接口，还装备了 CAN 和 RS485 总线，以及外部 W25Q16 FLASH 存储器，为高级数据处理和通信提供了坚实的基础。

此外，开发板还融入了 SD 卡接口、IO 扩展口、蜂鸣器等元素，以便实现存储扩展和音频反馈。JTAG/SWD 接口的集成确保了高效的程序下载与调试流程。电源管理方面，通过集成的电压稳压模块，开发板可以保障稳定的电源供应，确保系统的可靠性。



### 1.2.3 STM32F103ZET6 参考文档

Cortex-M3 技术参考手册.pdf	2023/8/3 9:41	Microsoft Edge ...	1,801 KB
Cortex-M3权威指南CnR2 (电子书) .pdf	2023/8/3 9:40	Microsoft Edge ...	6,608 KB
STM8和STM32产品选型手册.pdf	2023/8/3 9:40	Microsoft Edge ...	1,340 KB
STM32CUBEMX官方文档.pdf	2023/10/7 14:09	Microsoft Edge ...	22,422 KB
STM32F10xx固件库中文使用手册v3.5.0...	2023/8/3 9:41	Microsoft Edge ...	5,493 KB
STM32F10x微控制器参考手册(2009年1...	2023/8/3 9:40	Microsoft Edge ...	13,972 KB
stm32F103寄存器整理列表.xls	2023/8/3 9:41	Microsoft Excel ...	557 KB
STM32中文参考手册2010.pdf	2023/8/3 9:40	Microsoft Edge ...	16,555 KB
STM32103ZE中文数据手册.pdf	2023/10/7 11:04	Microsoft Edge ...	2,910 KB
ucGUI使用手册.pdf	2023/8/3 9:40	Microsoft Edge ...	2,365 KB
功能图_硬件资源分布图.JPG	2023/7/25 11:06	JPG 文件	658 KB
数据手册_STM32F103_C_D_E.PDF	2023/7/25 11:06	Microsoft Edge ...	3,150 KB
原理图_STM32F103ZX_V520.pdf	2023/7/25 11:06	Microsoft Edge ...	47 KB

原理图：对于软件开发工程师来讲，查看的目的是找外接设备与 STM32 的电气连接关系，为编程做准备。

中文参考：软件软件开发需要参考的文档

数据手册：硬件资源介绍

### 1.2.4 STM32F103ZET6 开发方式

在开发 STM32 单片机项目时，开发者通常使用汇编语言和 C 语言这两种主要编程语言。汇编语言允许开发者直接与硬件进行交互，而 C 语言则提供了更高级、更容易管理的编程接口。

实际的编程过程中，有两种主要的方法来操作和控制 STM32 单片机：

1. 直接配置和操作微控制器的功能模块寄存器：

这种方法涉及直接访问和修改微控制器内部的寄存器，以配置其硬件功能。

它需要对 STM32 的硬件结构和寄存器映射有深入的了解，适合对性能和资源使用有特别要求的场合。

2. 使用 ST 官方提供的固件库和驱动进行操作：

ST 公司为 STM32 提供了官方的固件库，这些库封装了对硬件的操作，简化了编程过程。

通过使用这些固件库，开发者可以更容易地实现对微控制器功能的控制，而不必深入到底层的硬件细节。

### 1.2.5 STM32 固件库介绍

在 STM32 微控制器的开发领域中，有多种固件库可供选择，每种库都有其独特的特点和优势。这些库包括最初的 Standard Peripheral Library (SPL)、更为现代化的 STM32Cube（包括 HAL 和 LL 库），以及广泛适用于 ARM Cortex-M 微控制器的 CMSIS。

#### 1. Standard Peripheral Library (SPL)

Standard Peripheral Library (SPL): 这是 ST Microelectronics 最初为其 STM32 微控制器系列发布的固件库。此库包含了一些方便的 C 函数，可以直接控制 STM32 的各种外设，通常称为标准库。

#### 2. STM32Cube

STM32Cube: ST Microelectronics 自 2015 年以来开始推广的一种新的固件库。

STM32Cube 包括一个嵌入式软件平台和一个独立的集成开发环境。嵌入式软件平台包括一个硬件抽象层(HAL)，该层为 STM32 的各种外设提供通用的 API，并且还包含一些中间件组件（如 FreeRTOS，USB 库，TCP/IP 库等）。STM32Cube 的集成开发环境（STM32CubeIDE）则包含了代码生成器，它可以生成基于 STM32Cube HAL 的初始化代码。

#### 3. LL (Low Layer) Drivers

LL (Low Layer) Drivers: LL 库是 STM32Cube 库的一部分，为高级用户提供了一个硬件抽象层的替代方案。LL 库提供了一组低级 API，可以让用户直接访问 STM32 外设的寄存器。这些 API 比 HAL 更加高效，但是需要更深入的硬件知识。

#### 4. CMSIS (Cortex Microcontroller Software Interface Standard)

CMSIS (Cortex Microcontroller Software Interface Standard): CMSIS 并不是一个 STM32 特定的固件库，而是 ARM 公司为 Cortex-M 微控制器定义的一组接口。许多 STM32 固件库（包括 SPL 和 STM32Cube）都使用 CMSIS 作为底层的硬件抽象。

STM32Cube HAL 是 STMicroelectronics 为了替代 SPL 而开发的更现代、更全面的固件库。它不仅提供了广泛的硬件支持，还包含了多种中间件组件，如 FreeRTOS、USB 库、TCP/IP 库等。HAL 库通过提供通用的 API，使得操作 STM32 的各种外设变得更加简单。这种抽象层降低了学习曲线，尤其适合初学者和那些希望快速实现项目原型的开发者。STM32CubeIDE 集成了代码生成器，可以自动生成基于 HAL 的初始化代码。这极大地简化了项目的初始配置过程，使开发者能够更快地进入实际的应用开发阶段。

## 1.3 开发环境搭建

在 STM32F103ZET6 的开发过程中，搭建合适的开发环境是至关重要的第一步。这通常包括安装和配置 Keil MDK-ARM，这是一个功能强大的集成开发环境，专门用于 ARM 架构的微控制器开发。Keil 提供了丰富的调试工具和易于使用的界面，使得编码、调试和测试变得更加高效。此外，结合 STM32CubeMX 工具，可以进一步优化开发流程。STM32CubeMX 是一款图形化配置工具，能够自动生成初始化代码，帮助开发者快速配置微控制器的外设和中间件，从而大大简化了项目的初始搭建工作。

### 1.3.1 keil 软件安装

#### 安装步骤

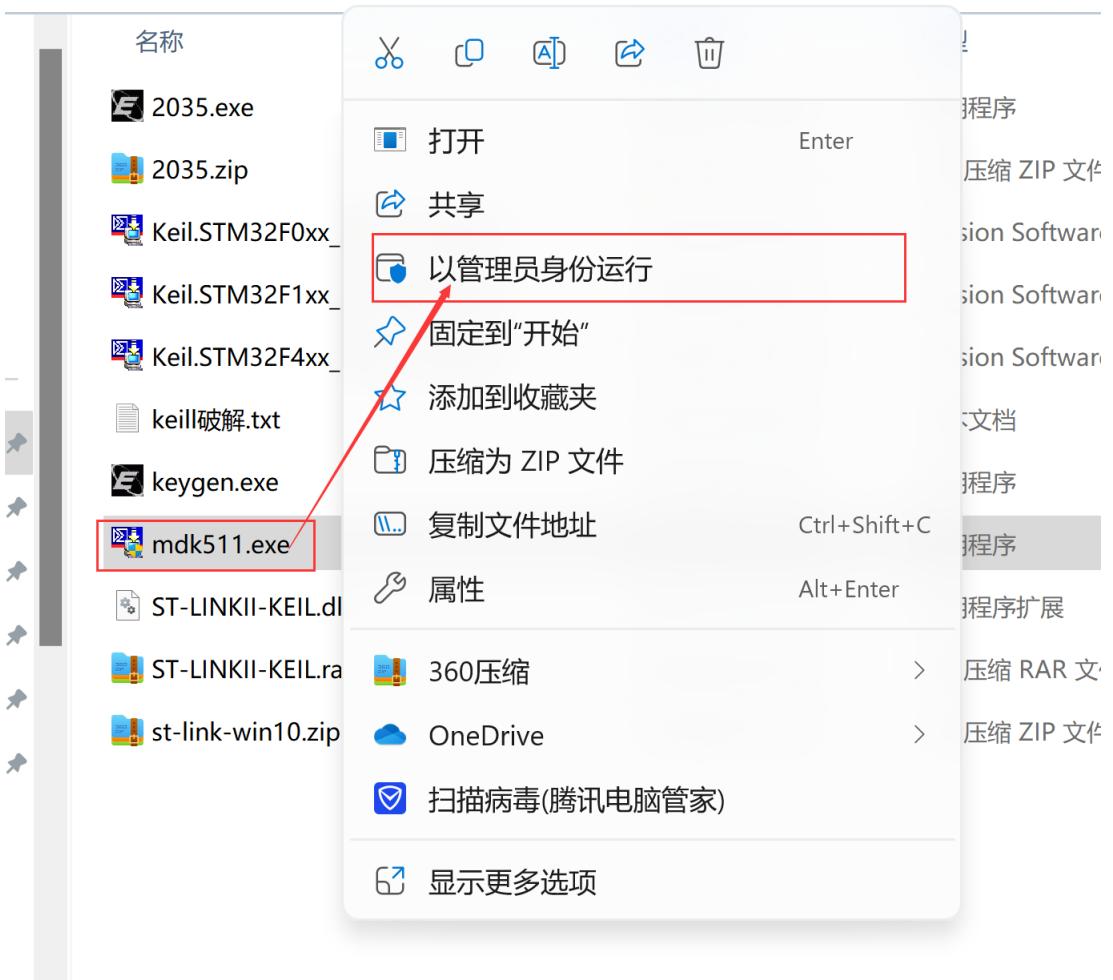
##### 1：找到并解压 keil 压缩包

 FlyMcuConfig.ini	2023/11/23 8:56	配置设置	2 KB
 LED.hex	2023/8/3 16:35	HEX 文件	6 KB
 MDK511.zip	2023/8/3 16:12	360压缩 ZIP 文件	418,432 KB
 SetupSTM32CubeMX-6.9.1-Win.exe	2023/7/28 19:12	应用程序	547,501 KB
 st-link v2 驱动(适用Win7和XP).zip	2023/10/7 15:29	360压缩 ZIP 文件	10,182 KB

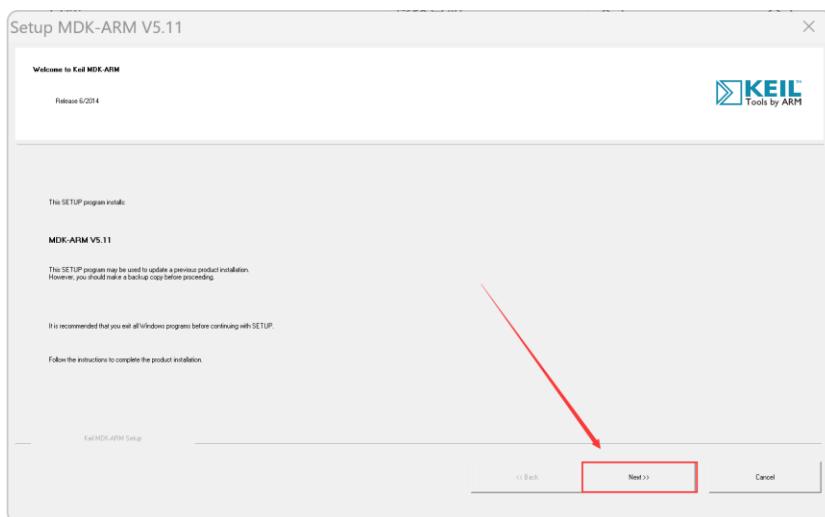
#### 解压后目录

 2035.exe	2020/3/5 18:04	应用程序	498 KB
 2035.zip	2021/3/26 11:53	360压缩 ZIP 文件	23 KB
 Keil.STM32F0xx_DFP.1.4.0.pack	2018/8/20 20:03	uVision Software...	22,256 KB
 Keil.STM32F1xx_DFP.2.0.0.pack	2018/8/20 20:04	uVision Software...	49,841 KB
 Keil.STM32F4xx_DFP.1.0.8.pack	2014/9/16 18:06	uVision Software...	35,613 KB
 keill破解.txt	2018/8/20 20:01	文本文档	3 KB
 keygen.exe	2018/8/20 20:01	应用程序	18 KB
 mdk511.exe	2018/8/20 20:12	应用程序	308,252 KB
 ST-LINKII-KEIL.dll	2023/11/27 11:24	应用程序扩展	388 KB
 ST-LINKII-KEIL.rar	2018/8/20 20:01	360压缩 RAR 文件	130 KB
 st-link-win10.zip	2018/8/20 20:02	360压缩 ZIP 文件	5,206 KB

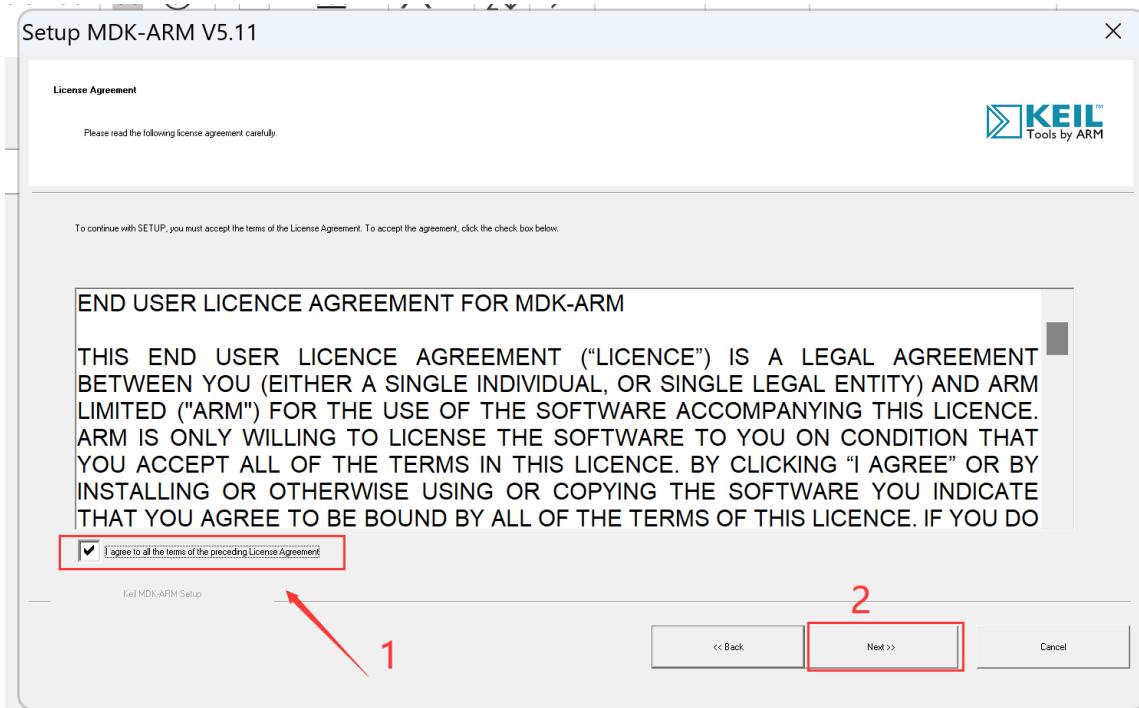
##### 2：右键管理员权限运行 keil 安装包



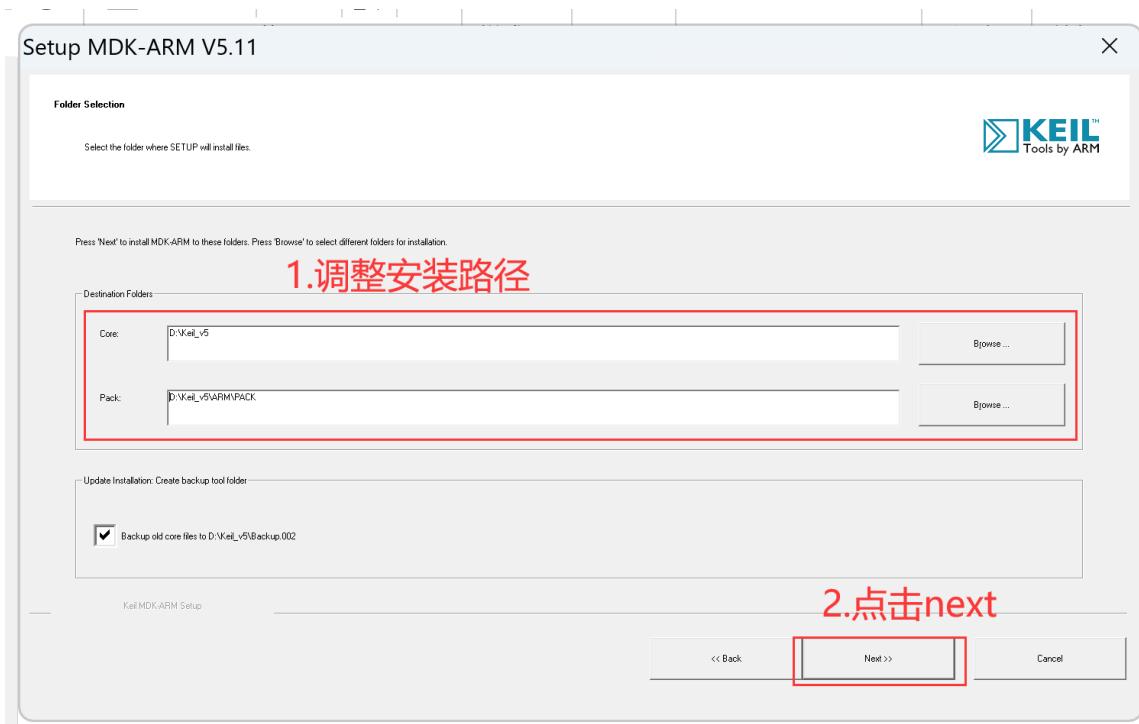
3: 单击 next



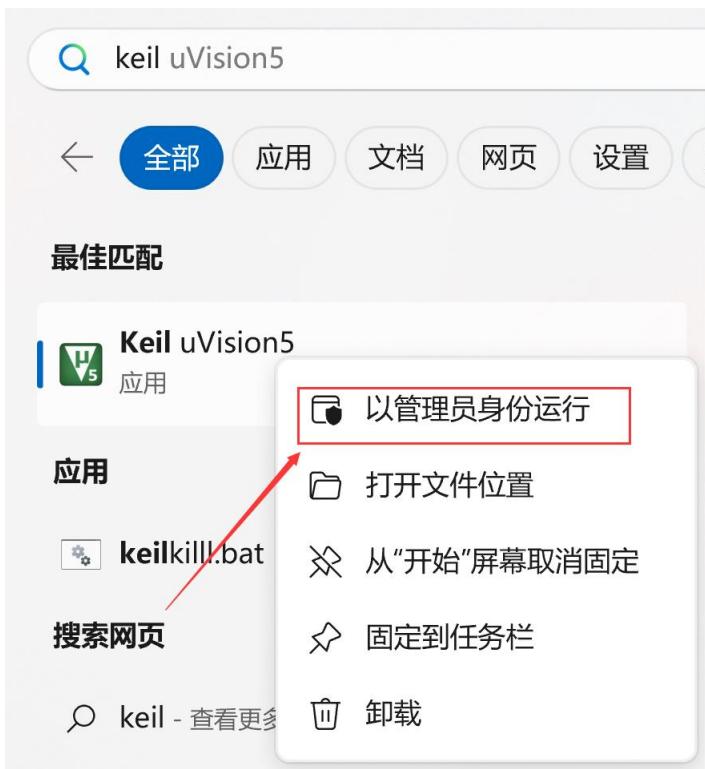
4: 勾选同意，点击 next



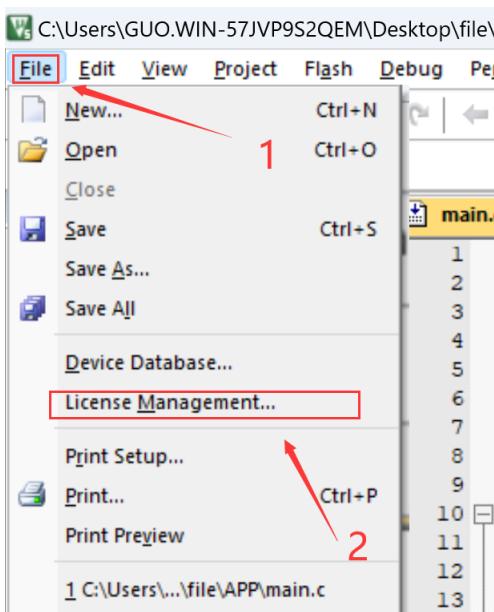
5: 调整安装路径，不能有中文，包括用户名不能为中文，点击 next



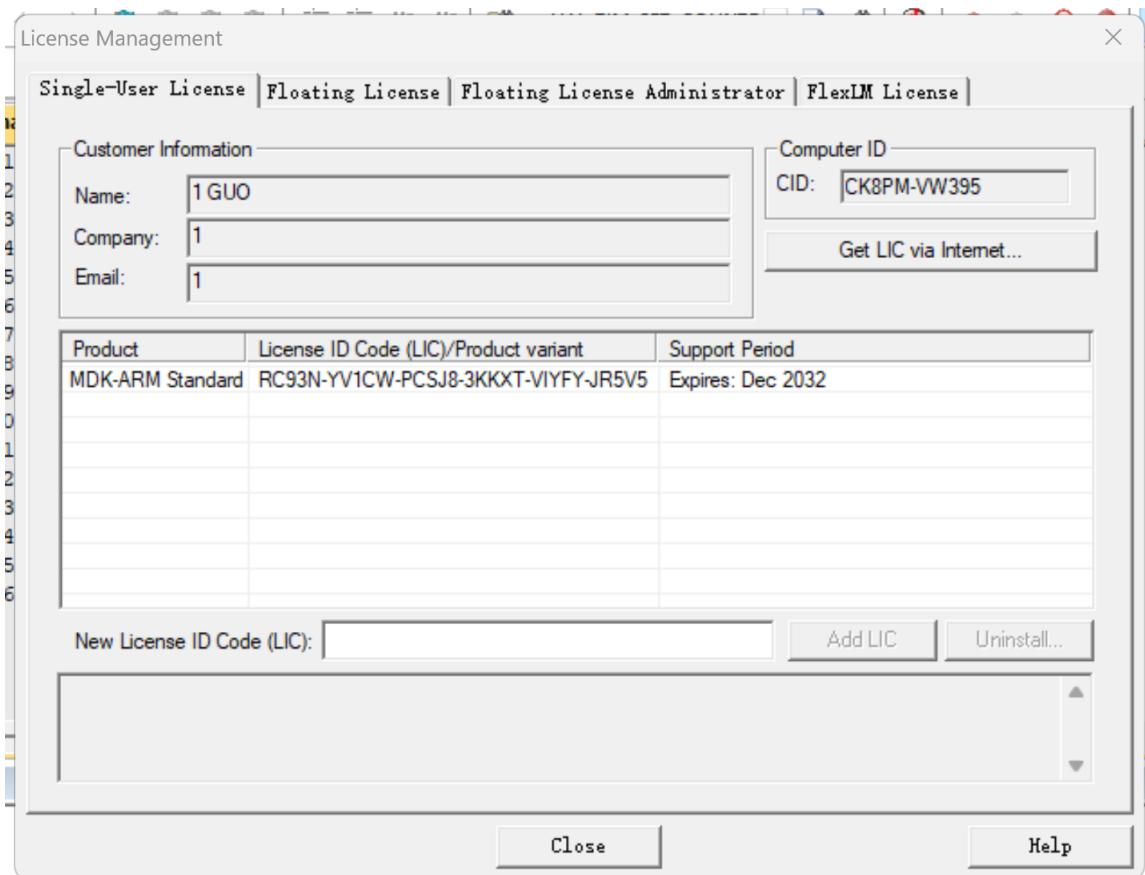
6: 安装完成后，在开始菜单搜索 keil，同时管理员身份运行



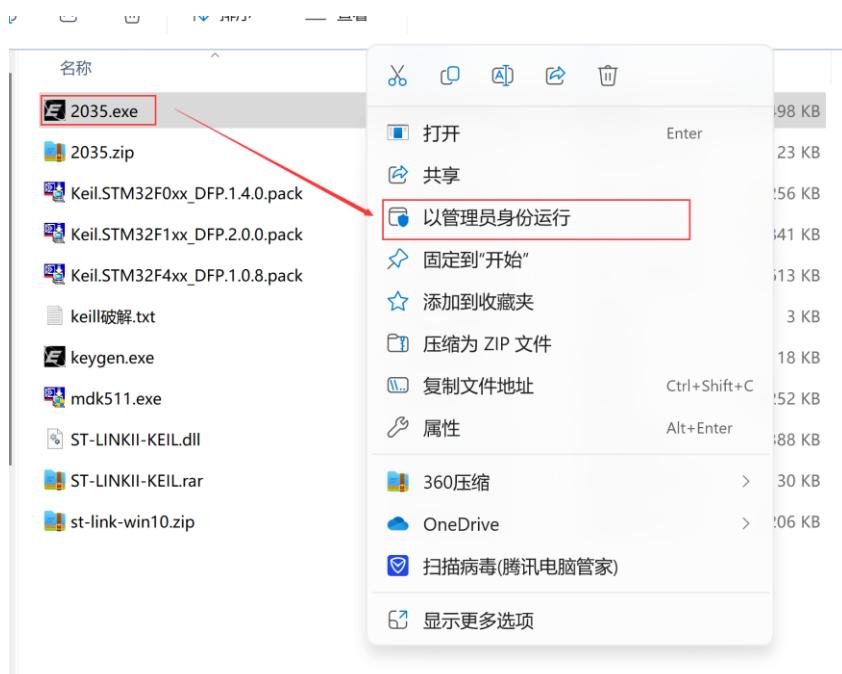
7: 在软件界面，点击 file，如下图操作打开许可证管理界面



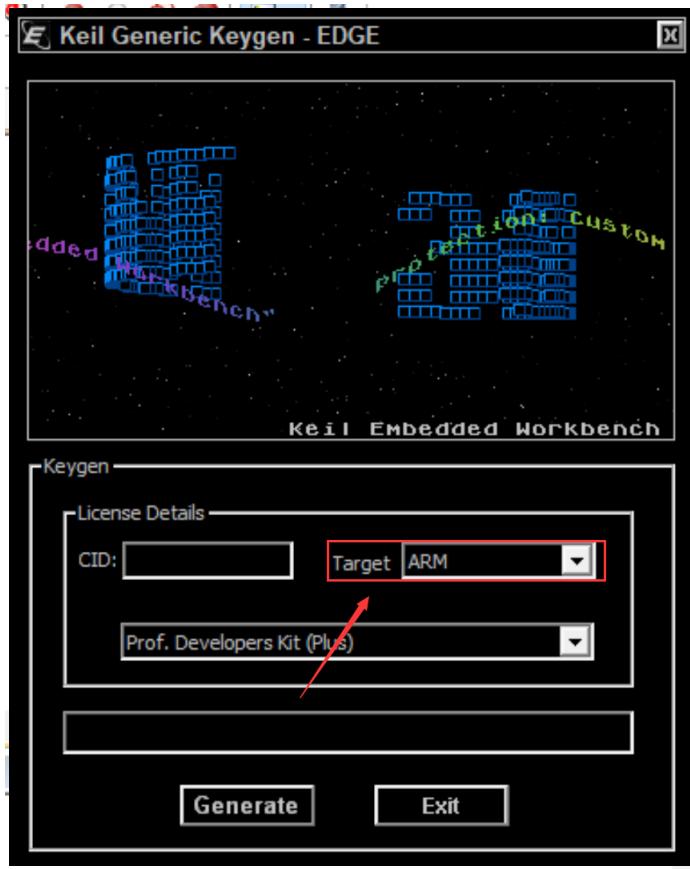
许可证管理界面



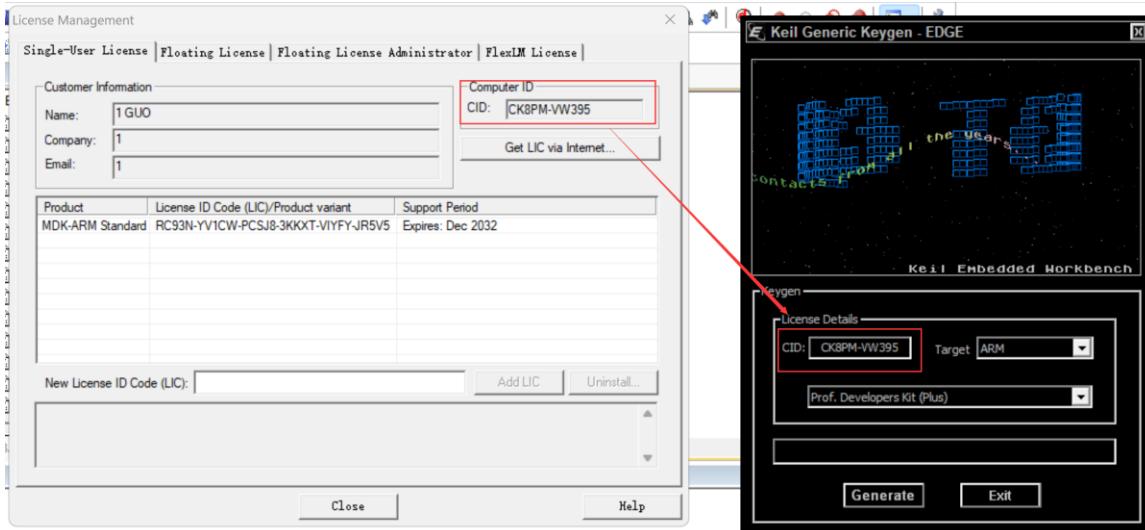
## 8: 管理员身份运行注册机



## 9: 修改参数为 ARM



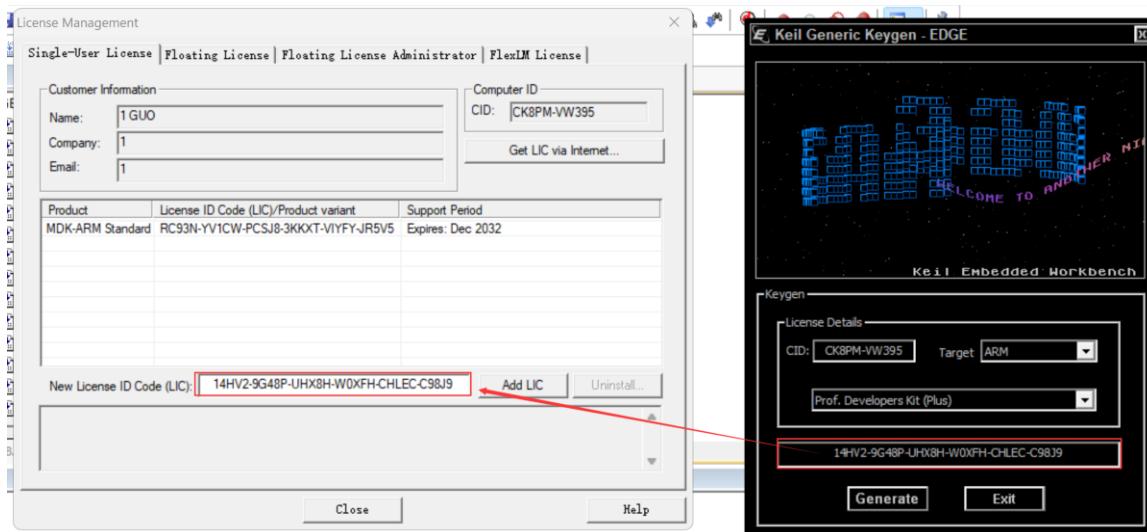
10：复制 CID 到注册机



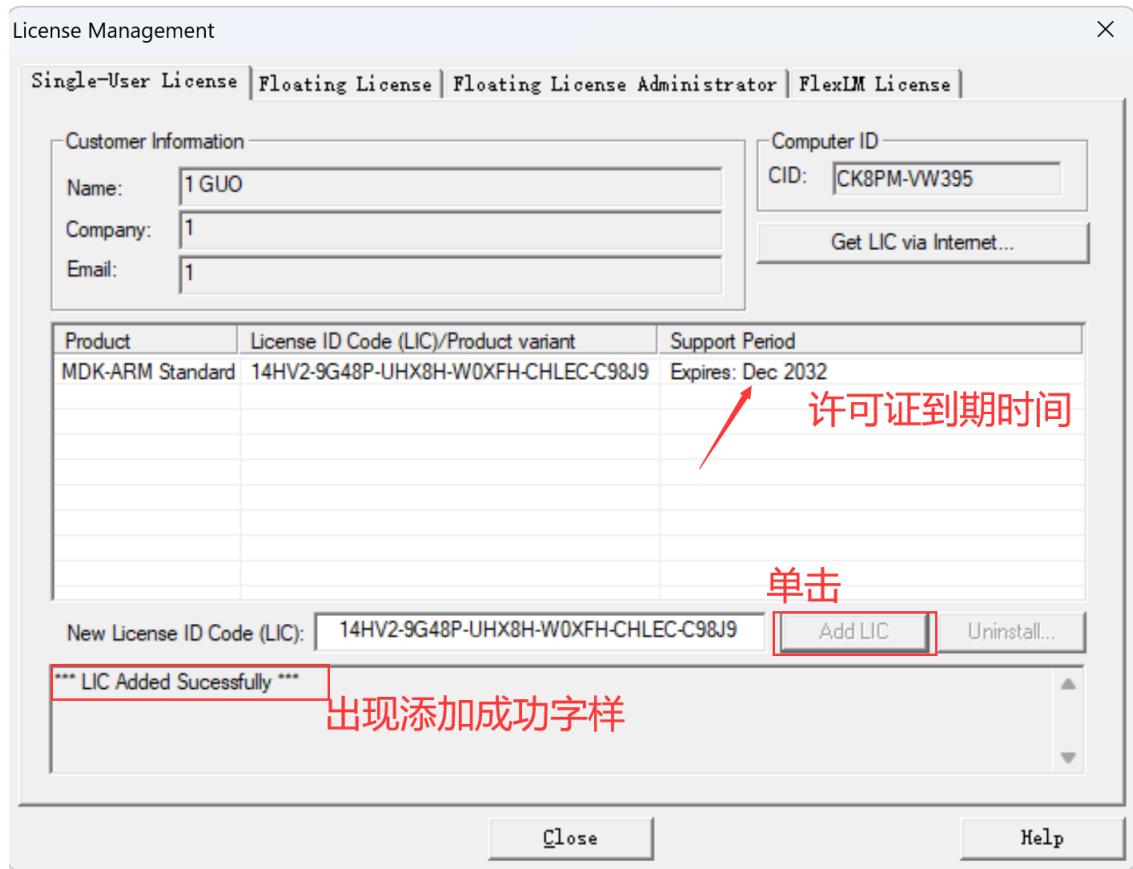
11：点击生成按钮



## 12: 将许可证代码复制到 keil



## 13: 单击 Add lic, 许可证安装成功, 2032 年到期

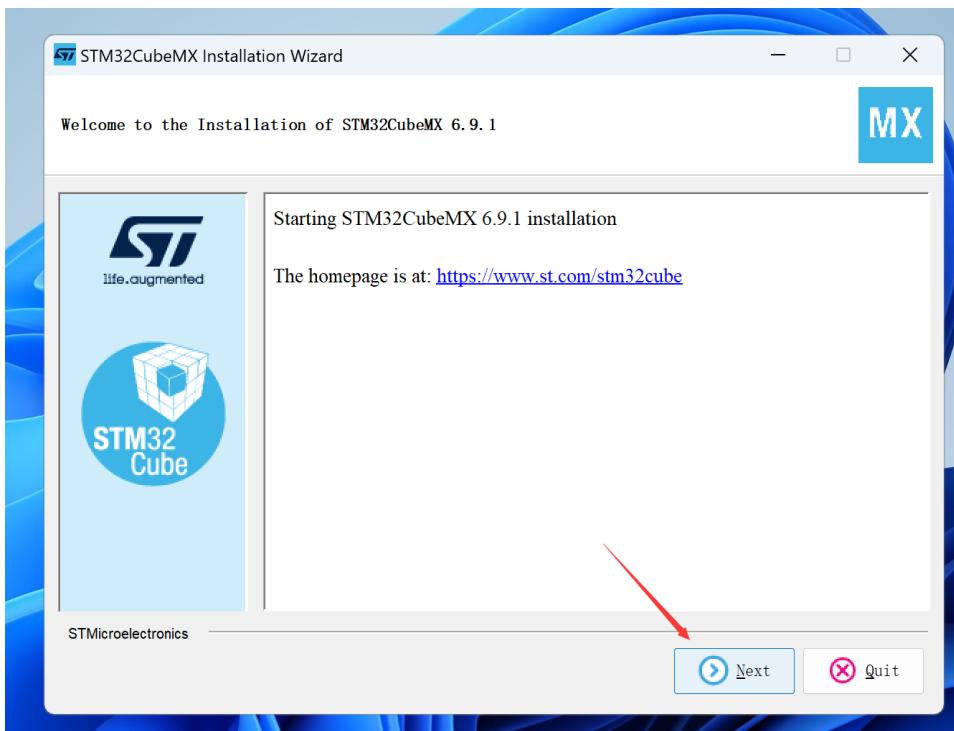


### 1.3.2 CubeMX 软件安装

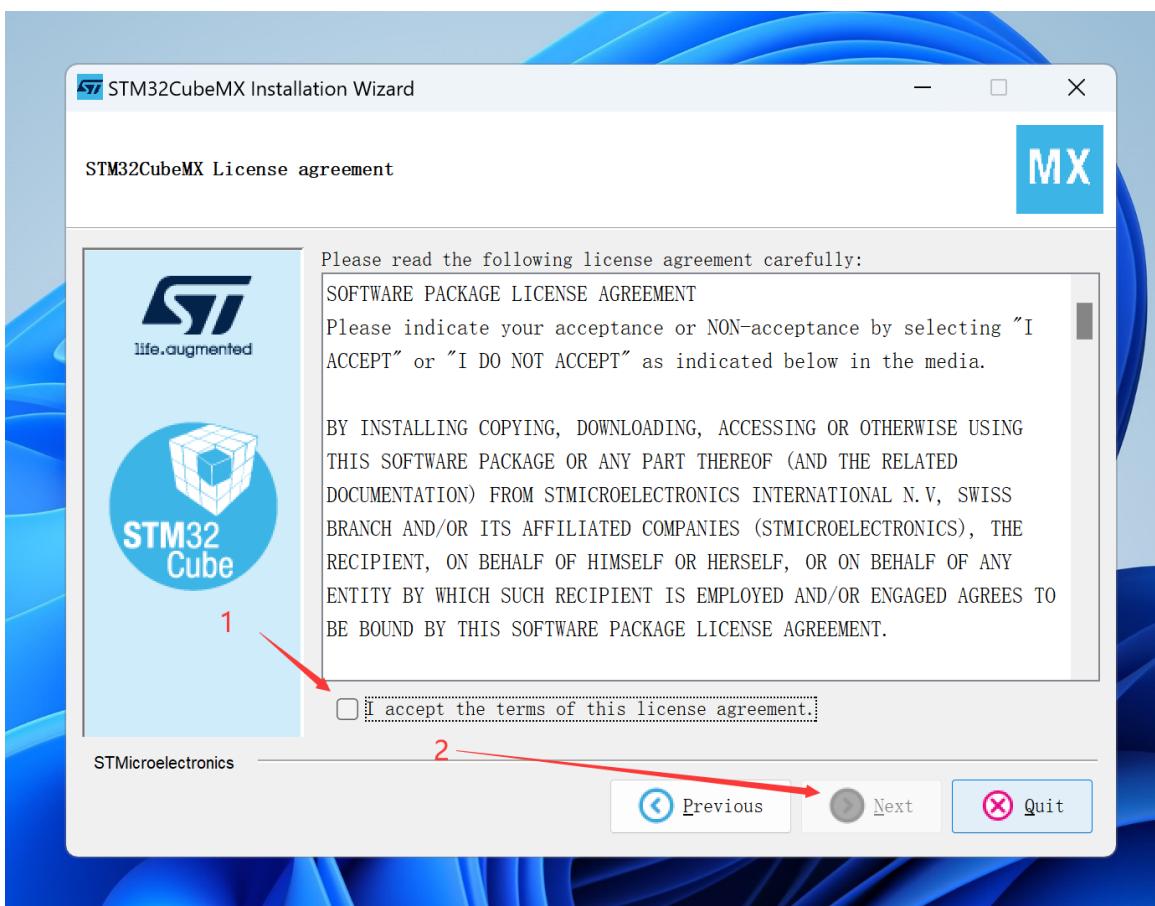
#### 1.右键安装包运行



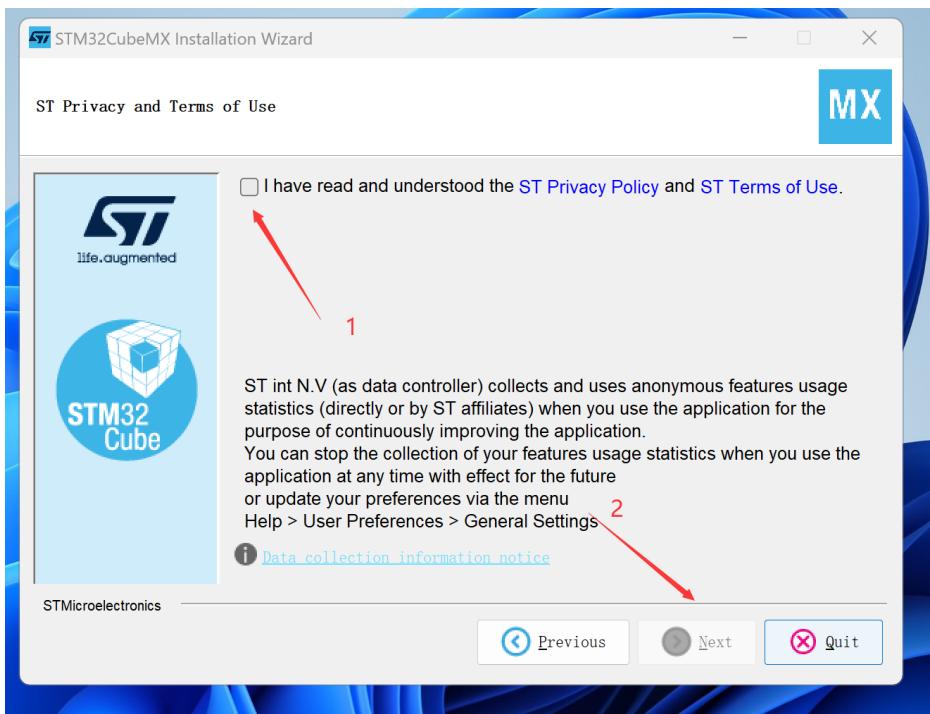
#### 2.点击 NEXT



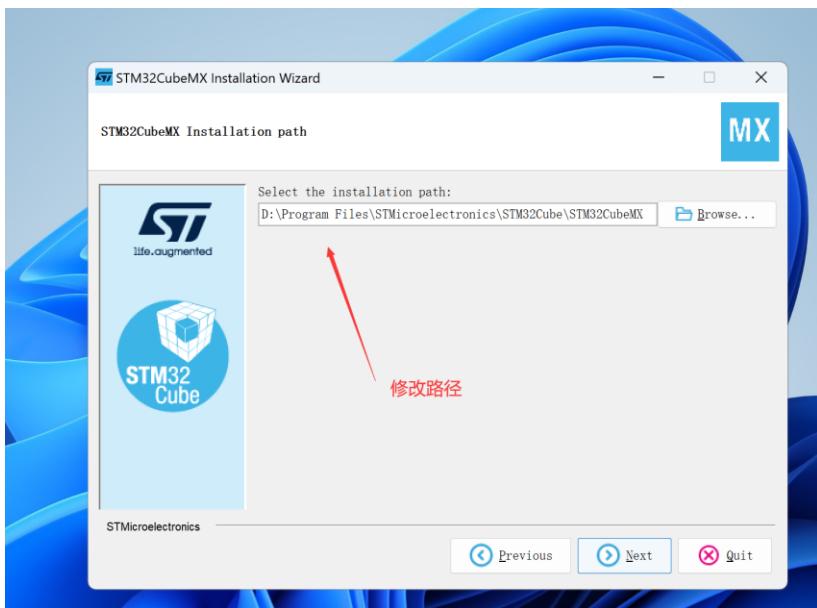
3.点击接收，之后点击 NEXT



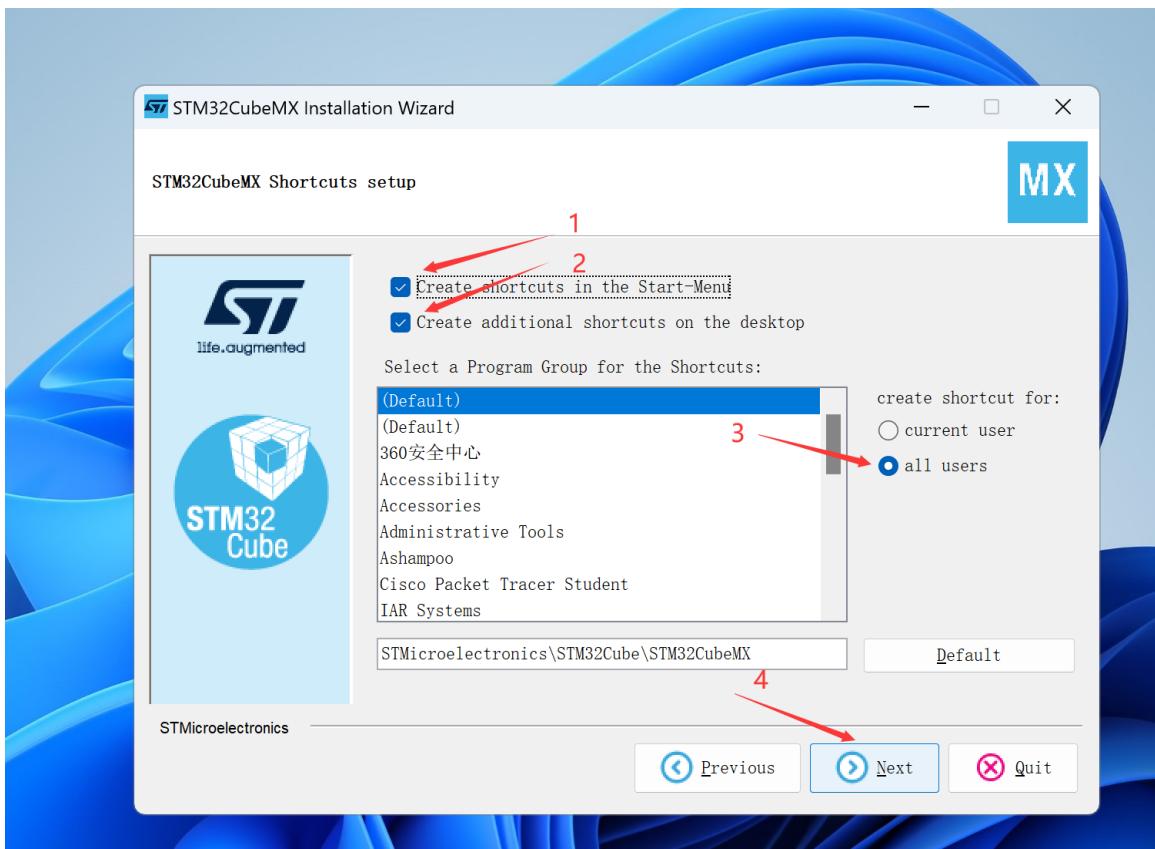
#### 4.如图设置



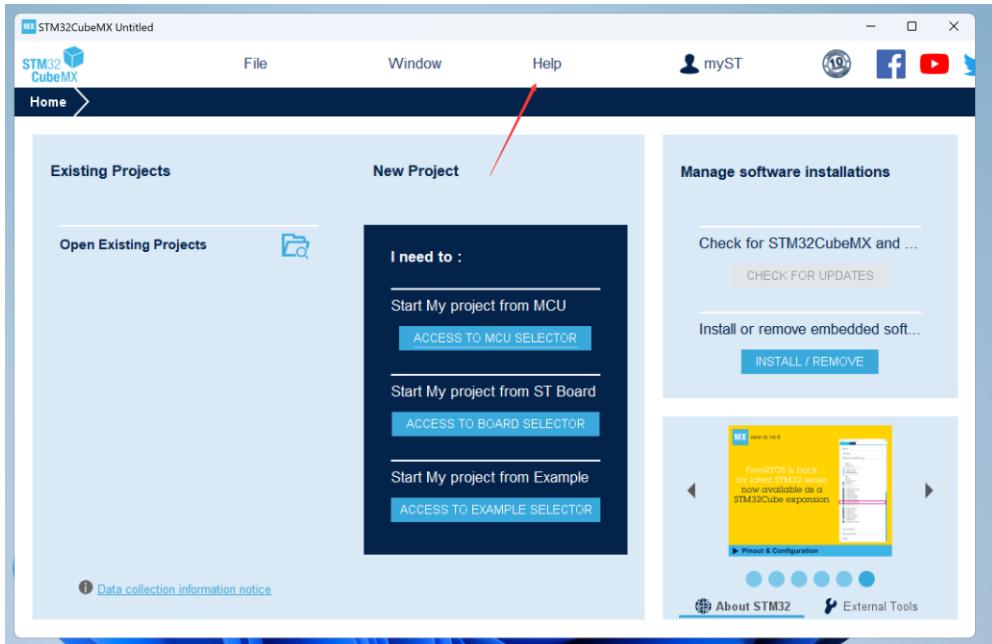
#### 5.修改路径（无中文目录）



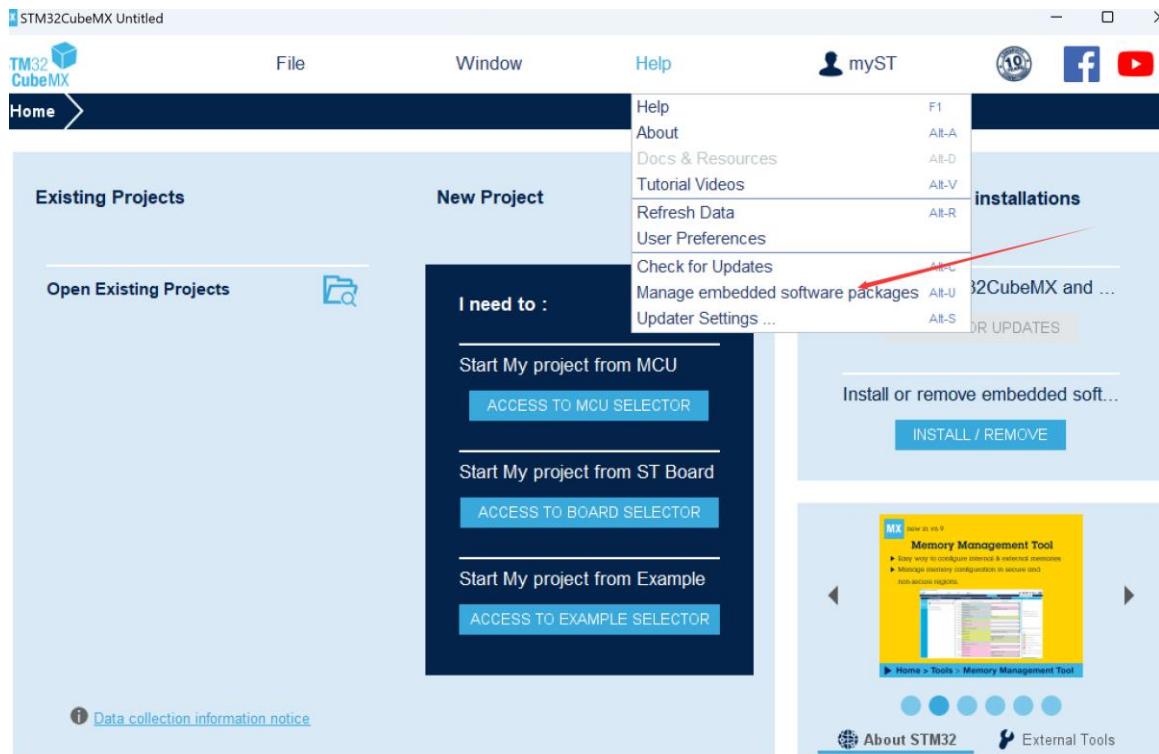
#### 6.如图设置



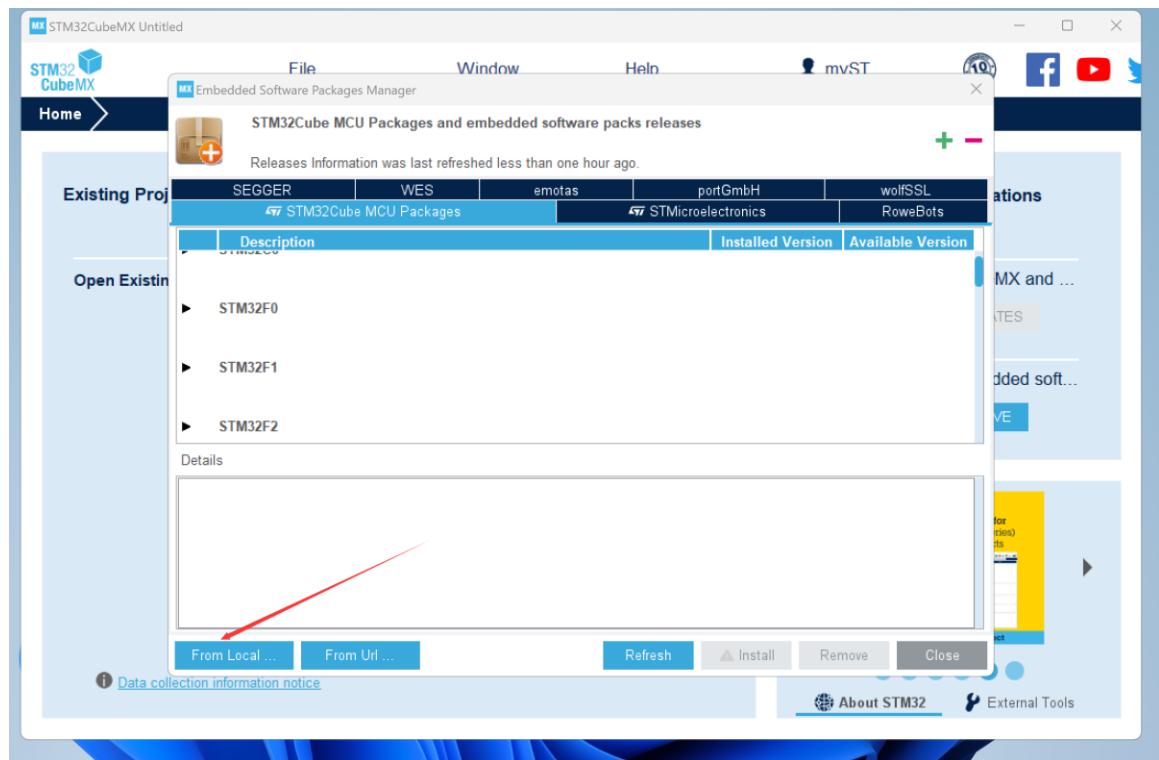
## 7. 注册后，打开 Cubemx，点击 Help

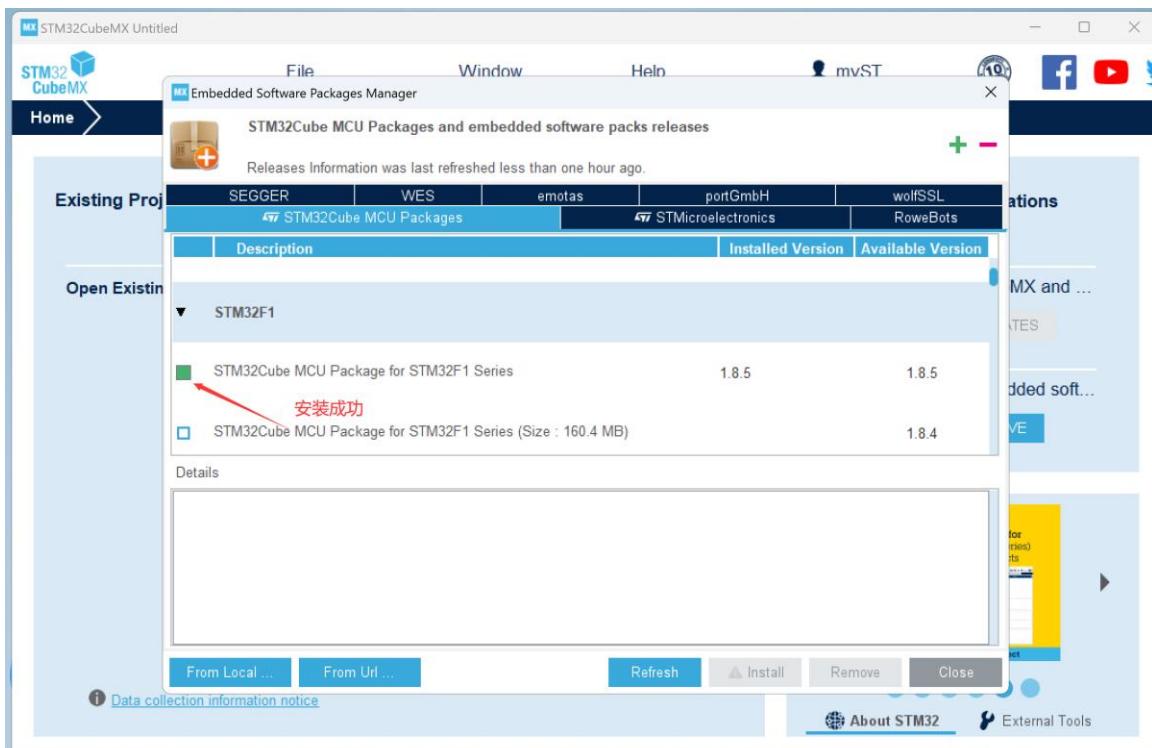
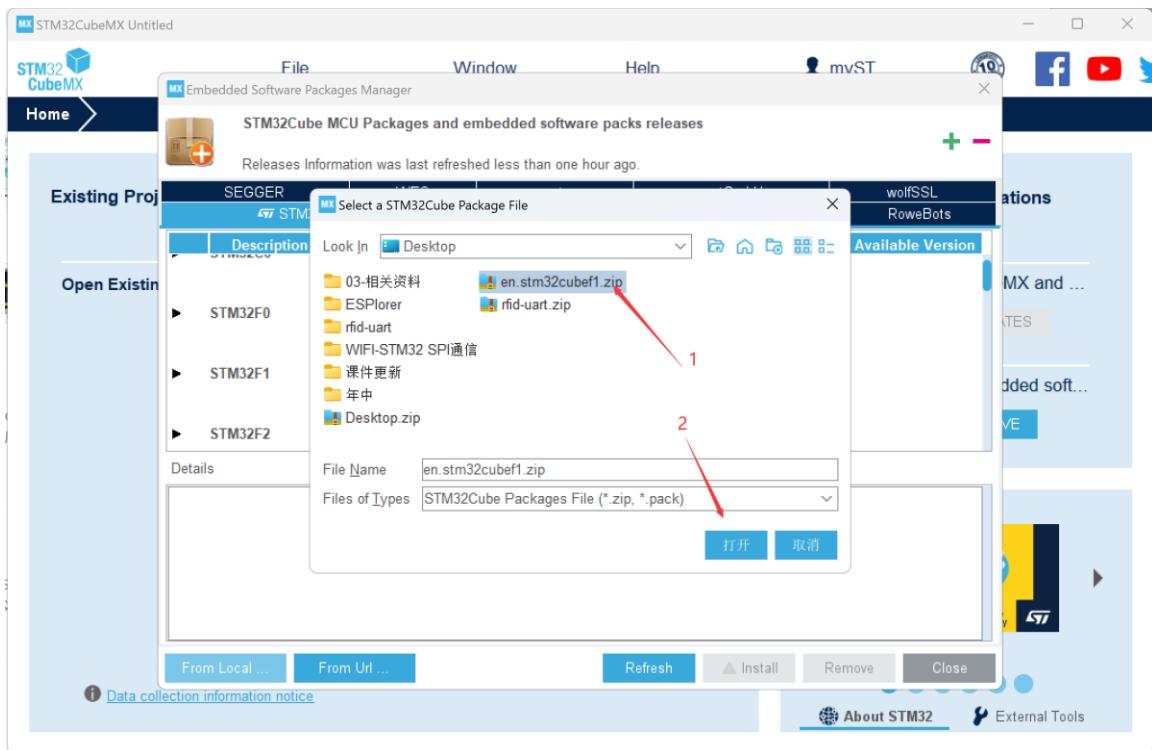


## 8. 选择包管理



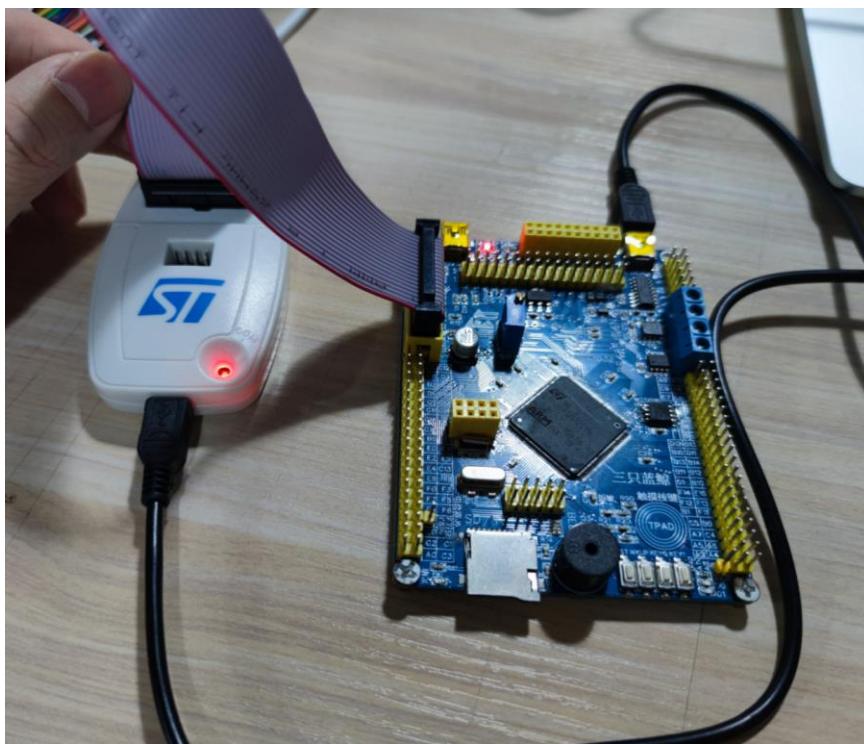
## 9.从本地添加 F103 相关





### 1.3.3 驱动文件安装

1.如图将 ST-LINK 与串口线分别连接至 2 个电脑 USB 口



2.连接成功后，安装 CH340 驱动



右键运行，点击安装

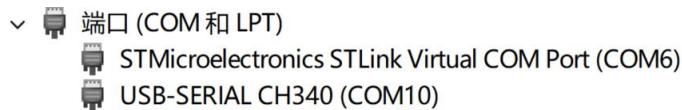


3.连接成功后，安装 ST-LINK 驱动

	st-link v2 usbdriver.exe	2011/5/9 17:44	应用程
	st-link v2 驱动(适用Win7和XP).zip	2023/10/7 15:29	360压

右键管理员运行后安装

4.在电脑打开设备管理器中，如下显示为安装成功



## 1.4 STM32 启动过程

探究 STM32F103ZET6 的启动过程是理解其系统架构和工作原理的关键一环。这个过程涉及到微控制器从上电到运行用户程序的各个阶段。首先，微控制器的内部架构决定了其启动行为，包括重要的硬件组件如中断系统、时钟管理和存储器。在上电或复位后，STM32 首先执行其内置的启动代码，该代码负责初始化硬件并决定后续的启动方式，例如是直接从主存储器启动还是通过引导加载器进行。此外，对启动文件的分析有助于理解微控制器如何加载和执行用户程序，这对于深入了解 STM32 的工作机制至关重要。了解这些启动过程的细节不仅有助于高效的故障排查，也为高级功能的实现提供了必要的背景知识。

### 1.4.1 系统架构

总线矩阵（Bus Matrix）：

总线矩阵主要用于在多个主设备（如 CPU、DMA 等）和从设备（内存和外设）之间进行数据的路由和管理。通过总线矩阵，主设备可以同时（并行）地访问从设备。

DMA 总线（DMA Bus）：

DMA，全称 Direct Memory Access，是一种让某些硬件子系统在内存和外设之间直接传输数据，而无需处理器参与的技术。在 STM32 中，DMA 总线主要用于 DMA 控制器和内存以及需要使用 DMA 服务的外设（如 USART、SPI、ADC 等）之间的数据传输。

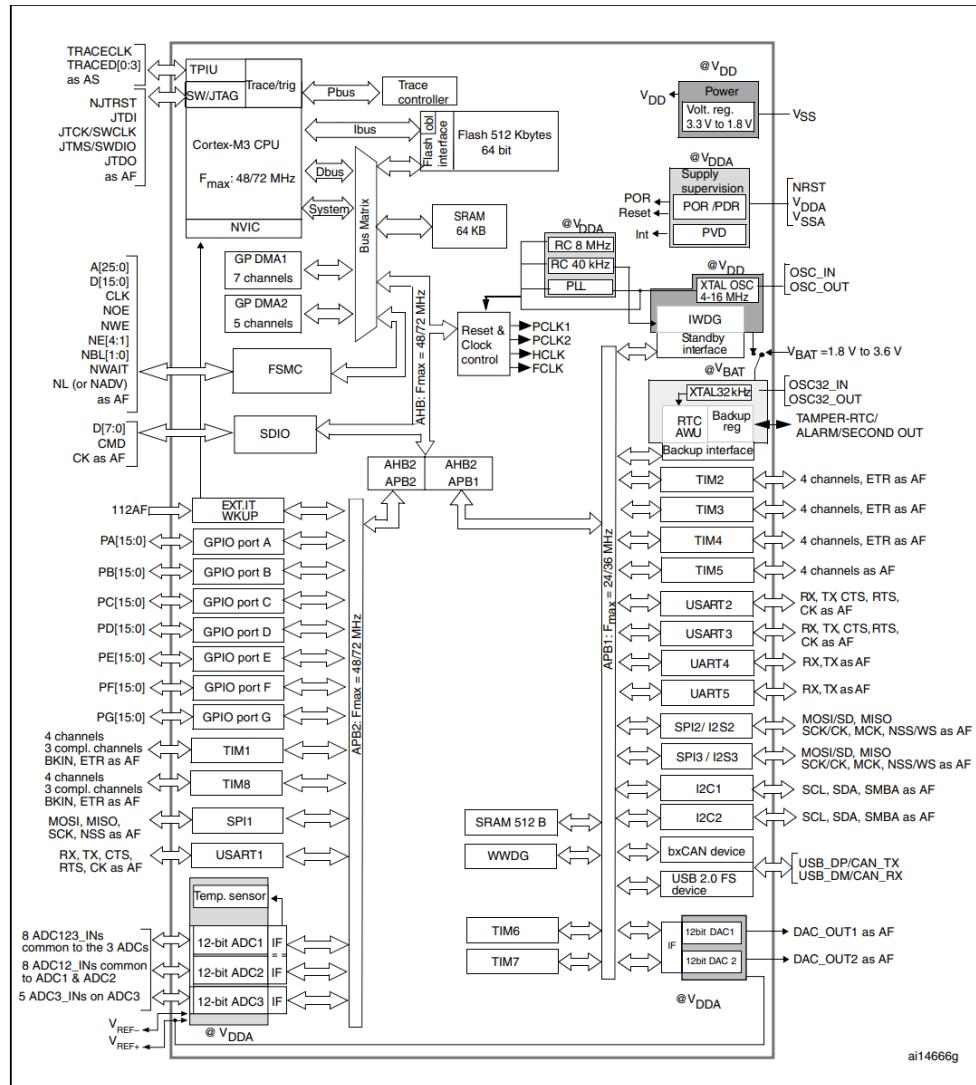
ICode 总线（ICode Bus）：

ICode 总线是一个专门用于 CPU 从 Flash 内存中取指令的总线。这条总线专门优化了指令的取得，以提高 CPU 的性能。专门用于从 Flash 内存或者其他指令存储器中获取指令，支持 CPU 以最优化的方式取指。该总线用于把指令从存储器传输到处理器核心，以便执行，主要面向读操作。

DCode 总线（DCode Bus）：

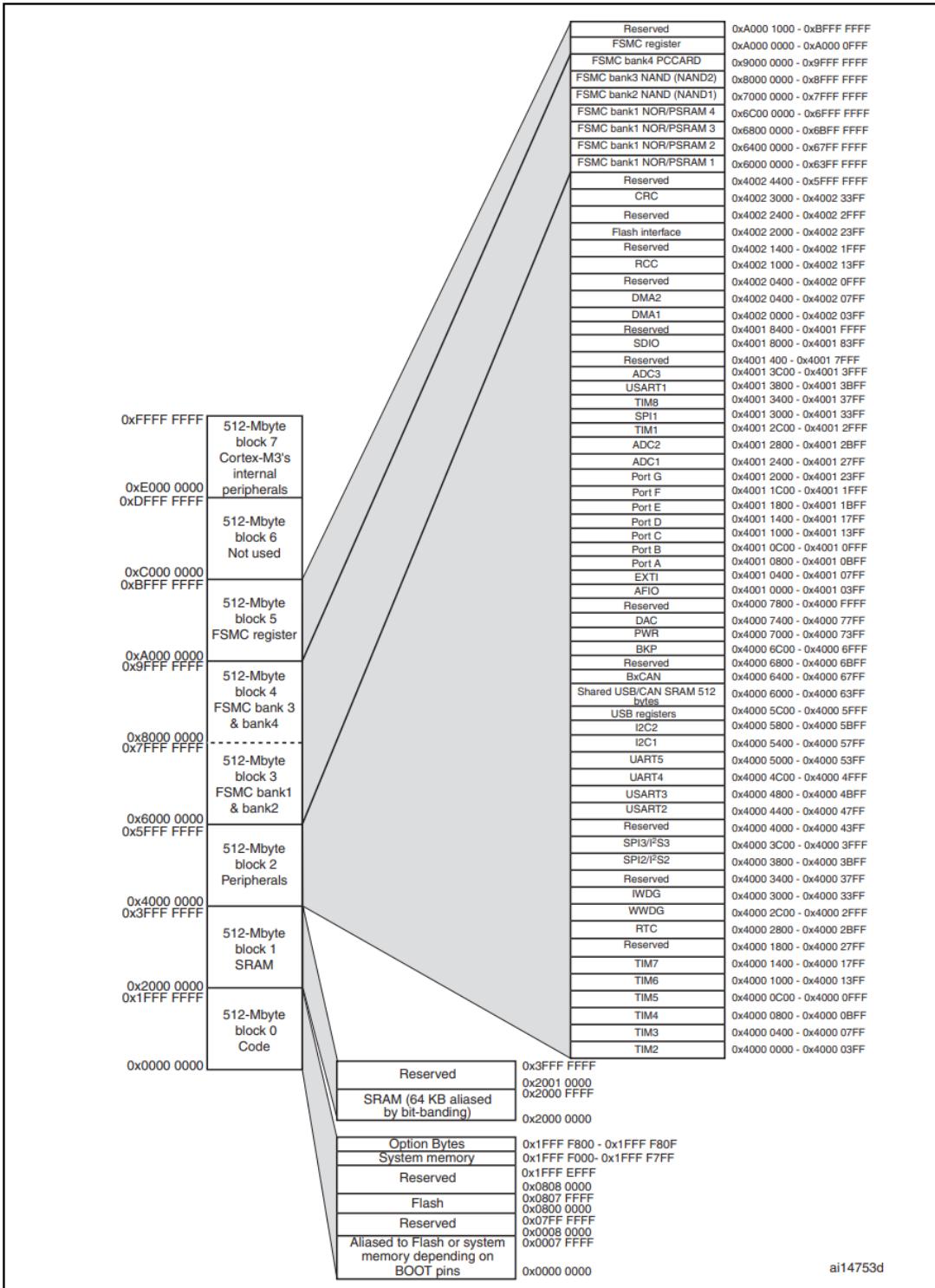
DCode 总线是一个专门用于 CPU 访问数据的总线，可以从 Flash 内存或系统内存中读取数据。用于 CPU 核心的数据访问，包括对 Flash 内存和 RAM 的读写。使数据可以并行地（与指令访问操作并行）从存储器中读取或写入。

主系统由以下部分构成：数据手册\_STM32F103\_C\_D\_E 第 12 页



#### 1.4.2 存储器的组织架构

程序存储器、数据存储器、寄存器和输入输出端口被组织在同一个 4GB 的线性地址空间内。可访问的存储器空间被分成 8 个主要块，每个块为 512MB。存储器映像：数据手册\_STM32F103\_C\_D\_E 第 40 页



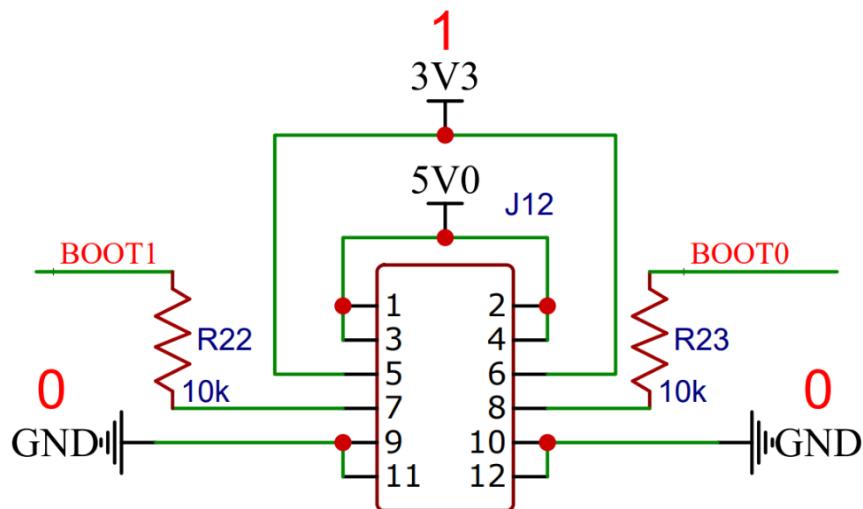
### 1.4.3 stm32 启动方式

在STM32F10xxx里，可以通过BOOT[1:0]引脚选择三种不同启动模式。

表6 启动模式

启动模式选择引脚		启动模式	说明
BOOT1	BOOT0		
X	0	主闪存存储器	主闪存存储器被选为启动区域
0	1	系统存储器	系统存储器被选为启动区域
1	1	内置SRAM	内置SRAM被选为启动区域

### 启动模式选择跳线



### 1.4.4 启动文件

! > 参考资料 > TEST > TEST > Drivers > CMSIS > Device > ST > STM32F1xx > Source > Templates > iar			
名称	修改日期	类型	大小
linker	2023/8/3 15:43	文件夹	
startup_stm32f100xb.s	2023/7/4 0:07	S 文件	13 KB
startup_stm32f100xe.s	2023/7/4 0:07	S 文件	15 KB
startup_stm32f101x6.s	2023/7/4 0:07	S 文件	11 KB
startup_stm32f101xb.s	2023/7/4 0:07	S 文件	11 KB
startup_stm32f101xe.s	2023/7/4 0:07	S 文件	14 KB
startup_stm32f101xg.s	2023/7/4 0:07	S 文件	15 KB
startup_stm32f102x6.s	2023/7/4 0:07	S 文件	12 KB
startup_stm32f102xb.s	2023/7/4 0:07	S 文件	12 KB
startup_stm32f103x6.s	2023/7/4 0:07	S 文件	13 KB
startup_stm32f103xb.s	2023/7/4 0:07	S 文件	13 KB
startup_stm32f103xe.s	2023/7/4 0:07	S 文件	16 KB

使用的启动文件为 startup\_stm32f103xe.s



The screenshot shows the STM32CubeMX software interface. On the left, the 'Project' tree view shows a project named 'test' with a folder 'Application/MDK-ARM' containing 'startup\_stm32f103xe.s'. This file is highlighted with a red border. Other folders like 'CMSIS', 'Application/User/Core', 'Drivers/STM32F1xx\_HAL\_Driver', and 'Drivers/CMSIS' are also listed. On the right, the code editor displays the assembly file 'startup\_stm32f103xe.s'. The code is as follows:

```
1 ;***** (C) COPYRIGHT 2017 STMicroelectronics *****
2 /* File Name : startup_stm32f103xe.s
3  * Author : MCD Application Team
4  * Description : STM32F103xE Device vector table for MDK-ARM toolchain.
5  */
6 /*
7  * This module performs:
8  * - Set the initial SP
9  * - Set the initial PC == Reset_Handler
10 * - Set the vector table entries with the exceptions ISR address
11 * - Configure the clock system
12 * - Branches to _main in the C library (which eventually calls main()).
13 */
14 After Reset the Cortex-M3 processor is in Thread mode,
15 priority is Privileged, and the Stack is set to Main.
16
17 ;* Copyright (c) 2017-2021 STMicroelectronics.
18 ;* All rights reserved.
19 */

; Reset handler
Reset_Handler PROC
    EXPORT Reset_Handler [WEAK]
    IMPORT __main
    IMPORT SystemInit
    LDR R0, =SystemInit
    BLX R0
    LDR R0,=__main
    BX R0
ENDP
```

#### 1.4.5 启动文件分析

##### 1. 复位处理函数

```
; Reset handler
Reset_Handler PROC
    EXPORT Reset_Handler [WEAK]
    IMPORT __main
    IMPORT SystemInit
    LDR R0, =SystemInit
    BLX R0
    LDR R0,=__main
    BX R0
ENDP
```

**Reset\_Handler PROC:** 这一行定义一个程序例程（或函数）Reset\_Handler。

PROC 是 ARM 汇编语言的一个关键字，用于标记一个程序例程或函数的开始。

**EXPORT Reset\_Handler [WEAK]:** 这一行将 Reset\_Handler 这个标签（函数）导出，使其可以在其他文件中被引用或调用。[WEAK]标记表示这是一个弱符号，如果在其他地方定义了同名的强符号（不带[WEAK]标记的符号），则该弱符号会被替换。

**IMPORT \_\_main 和 IMPORT SystemInit:** 这两行导入了两个外部函数\_\_main 和 SystemInit，它们在其他文件中定义。

**LDR R0, =SystemInit:** 这一行将 SystemInit 函数的地址加载到寄存器 R0 中。

**BLX R0:** 这一行执行一个分支指令，并将程序的控制权交给 R0 寄存器中的地址，也就是 SystemInit 函数。BLX 指令同时也会保存返回地址到 LR（链接寄存器）中。

**LDR R0,=\_\_main:** 这一行将\_\_main 函数的地址加载到寄存器 R0 中。

**BX R0:** 这一行执行一个分支指令，将程序的控制权交给 R0 寄存器中的地址，也就是\_\_main 函数。

**ENDP:** 这是一个汇编指令，表示程序例程或函数的结束。

上电，复位后调用的函数。上电，复位后首先调用 SystemInit 函数，之后调用 \_main 函数。

## 2.SystemInit 函数

```
175 void SystemInit (void)
176 {
177     #if defined(STM32F100xE) || defined(STM32F101xE) || defined(STM32F101xG) || defined(STM32F103xE) || defined(STM32F103xG)
178     #ifdef DATA_IN_ExtSRAM
179         SystemInit_ExtMemCtl();
180     #endif /* DATA_IN_ExtSRAM */
181     #endif
182
183     /* Configure the Vector Table location -----*/
184     #if defined(USER_VECT_TAB_ADDRESS)
185         SCB->VTOR = VECT_TAB_BASE_ADDRESS | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal SRAM. */
186     #endif /* USER_VECT_TAB_ADDRESS */
187 }
```

SystemInit 的主要目的是进行系统级别的初始化，包括配置内部硬件和设置系统的运行环境。

## 3.\_main 函数

负责完成初始化 C 环境、设置堆栈、清零全局和静态变量等，并最终调用用户定义的 main 函数。

# 1.5 STM32 系统时钟树

系统时钟树是微控制器内部运作的心脏，它决定了处理器和外设的工作频率。这一架构包括多种时钟源，如内部 RC 振荡器、外部晶振，以及更复杂的相位锁定环（PLL）系统。这些时钟源通过精心设计的时钟树分配给微控制器的不同部分，包括核心处理器、各种外设以及总线系统。理解系统时钟的配置和分配是优化性能和功耗的关键，同时也对保证系统稳定运行至关重要。对 STM32 的系统时钟树进行分析，可以帮助开发者更好地把握时钟管理的细节，确保硬件组件在正确的频率下高效运行。此外，时钟树的配置也直接影响到系统的响应速度和处理能力，是微控制器编程中不可忽视的重要组成部分。

## 1.5.1 系统时钟的时钟源

三种不同的时钟源可被用来驱动系统时钟(SYSCLK): HSI 振荡器时钟（高速系统内部时钟） HSE 振荡器时钟（高速系统外部时钟） PLL 时钟（锁相环时钟）

这些设备有以下 2 种二级时钟源: 40kHz 低速内部 RC，可以用于驱动独立看门狗和通过程序选择驱动 RTC。RTC 用于从停机/待机模式下自动唤醒系统。

32.768kHz 低速外部晶体也可用来通过程序选择驱动 RTC(RTCCLK)。当不被使用时，任一个时钟源都可被独立地启动或关闭，由此优化系统功耗

## 1.5.2 时钟树

三种不同的时钟源可被用来驱动系统时钟(SYSCLK): HSI 振荡器时钟（高速系统内部时钟） HSE 振荡器

时钟（高速系统外部时钟） PLL 时钟（锁相环时钟）

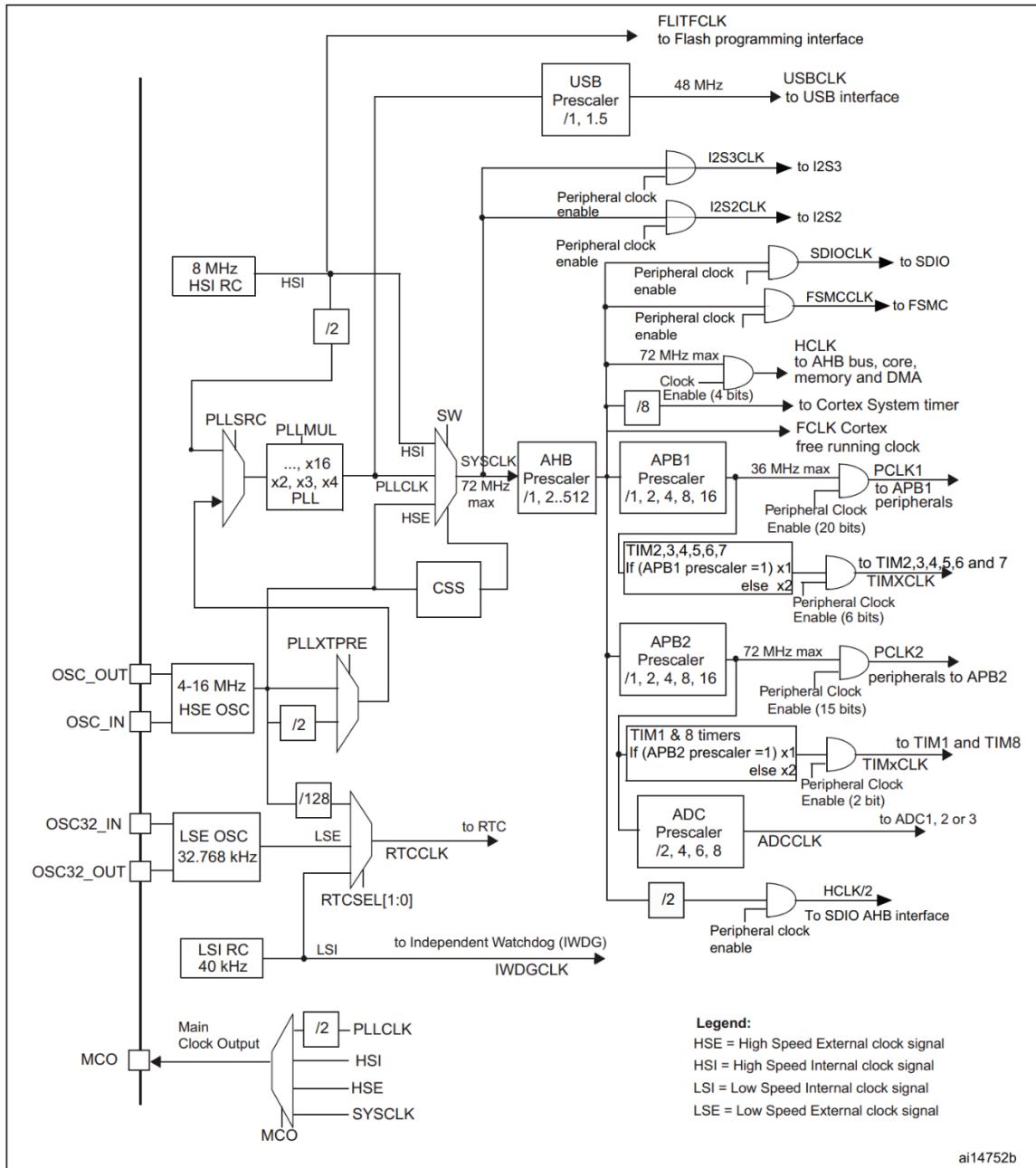
这些设备有以下 2 种二级时钟源： 40kHz 低速内部 RC，可以用于驱动独立看门狗和通过程序选择驱动

RTC。 RTC 用于从停机/待机模式下自动唤醒系统。 32.768kHz 低速外部晶体也可用来通过程序选择驱动

RTC(RTCCLK)。 当不被使用时，任一个时钟源都可被独立地启动或关闭，由此优化系统功耗

PLL 是 Phase-Locked Loop（相位锁定环）的简称，是一种控制系统的结构，常用于电子通信设备中。在微控制器和嵌入式系统中，PLL 被用作时钟系统的一部分，能够生成频率比输入时钟（通常是内部的低速时钟或者外部的晶体振荡器提供的时钟）更高的时钟信号。

时钟树：数据手册\_STM32F103\_C\_D\_E 第 13 页



- When the HSI is used as a PLL clock input, the maximum system clock frequency that can be achieved is 64 MHz.

### 1.5.3 系统时钟分析

系统时钟设置在 main.c 中的 `SystemClock_Config` 函数中完成

```
void SystemClock_Config(void)
{
    // 初始化两个类型结构体，这两个结构体会被用来存储时钟配置的参数
}
```

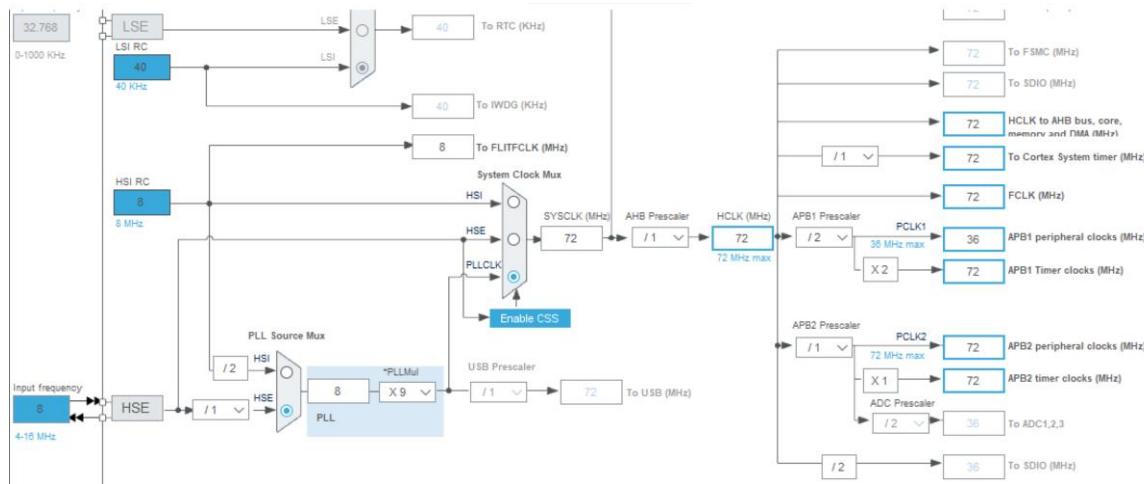
```
RCC_OscInitTypeDef RCC_OscInitStruct = {0};  
RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};  
  
// 初始化 RCC_OscInitTypeDef 结构体，这个结构体中的参数用来配置振荡器  
RCC_OscInitStruct.OscillatorType = RCC OSCILLATORTYPE_HSE; // 选择外部高速  
振荡器 (HSE) 作为时钟源  
  
RCC_OscInitStruct.HSEState = RCC_HSE_ON; // 打开 HSE  
  
RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1; // 设置 HSE 预分频  
器的值为 1  
  
RCC_OscInitStruct.HSISite = RCC_HSI_ON; // 打开 HSI  
  
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON; // 打开 PLL  
  
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE; // 设置 PLL 源为 HSE  
  
RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9; // 设置 PLL 倍频器的值为 9  
  
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)  
{  
    Error_Handler(); // 如果配置失败，则调用错误处理函数  
}  
  
// 初始化 RCC_ClkInitTypeDef 结构体，这个结构体中的参数用来配置时钟  
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK  
|RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2; // 设置时钟类  
型为 HCLK, SYSCLK, PCLK1, PCLK2  
  
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK; // 设置系统时钟  
源为 PLL 时钟  
  
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1; // 设置 AHB 时钟分频器  
的值为 1  
  
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2; // 设置 APB1 时钟分频器  
的值为 2  
  
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1; // 设置 APB2 时钟分频器  
的值为 1  
  
if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
```

```

{
    Error_Handler(); // 如果配置失败，则调用错误处理函数
}
}

```

时钟设置如图所示

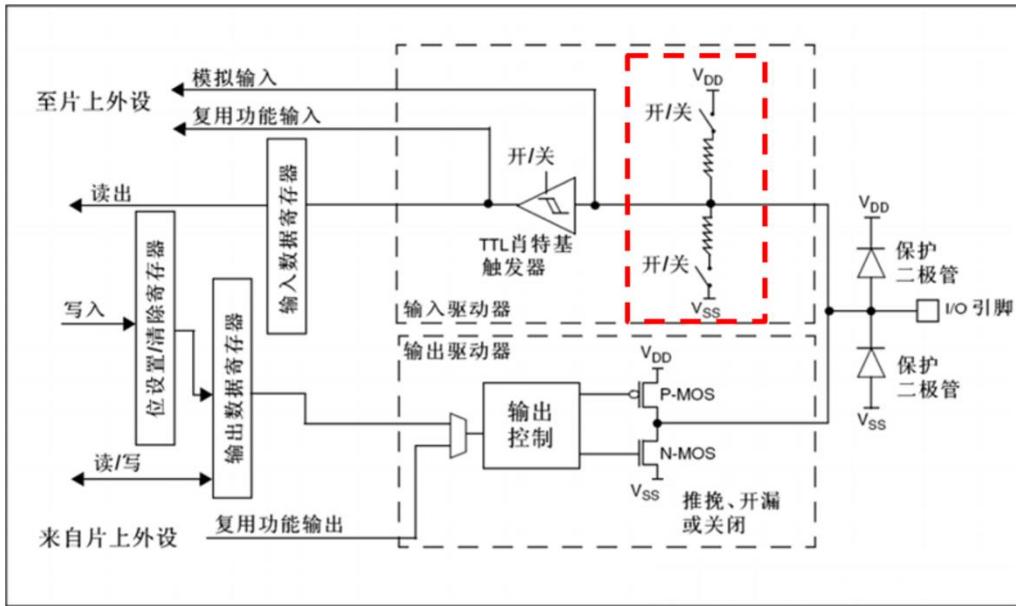


## 1.6 STM32 通用输入输出

GPIO(General purpose input/output, 通用型输入输出), 一个引脚可以用于输入、输出或其他特殊功能, PIN 脚依现实需要可作为通用输入 (GPI) 或通用输出 (GPO) 或通用输入与输出 (GPIO) , 并多个 GPIO 一起组合可实现诸如 UART、SPI 等外设功能。

GPIO 是通过寄存器操作来实现电平信号输入输出的, 对于输入, 通过读取输入数据寄存器 (IDR, Input Data Register) 来确定引脚电位的高低; 对于输出, 通过写入输出数据寄存器 (ODR, Output Data Register) 来让这个引脚输出高电位或者低电位; 对于其他特殊功能, 则有另外的寄存器来控制它们。GPIO 的输出通常可以输出 0/1 信号, 用来实现如 LED 灯、继电器、蜂鸣器等控制, 而 GPIO 的输入可识别 0/1 信号, 用来实现开关、按键等等动作或状态判定。

### 1.6.1 GPIO 框图



保护二极管：

IO 引脚上下两边两个二极管用于防止引脚外部过高、过低的电压输入，当引脚电压高于  $V_{DD\_FT}$  时，上方的二极管导通，当引脚电压低于  $V_{SS}$  时，下方的二极管导通，防止不正常电压引入芯片导致芯片烧毁

上拉、下拉电阻：

控制引脚默认状态的电压，开启上拉的时候引脚默认电压为高电平，开启下拉的时候引脚默认电压为低电平

TTL 施密特触发器：

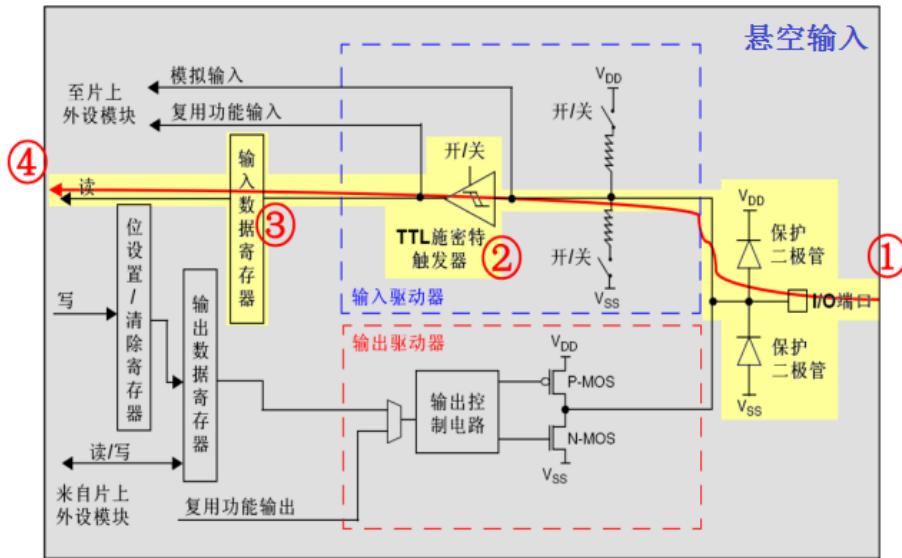
基本原理是当输入电压高于正向阈值电压，输出为高；当输入电压低于负向阈值电压，输出为低；IO 口信号经过触发器后，模拟信号转化为 0 和 1 的数字信号。

P-MOS 管和 N-MOS 管：

信号由 P-MOS 管和 N-MOS 管，依据两个 MOS 管的工作方式，使得 GPIO 具有“推挽输出”和“开漏输出”的模式。P-MOS 管低电平导通，高电平关闭，下方的 N-MOS 高电平导通，低电平关闭

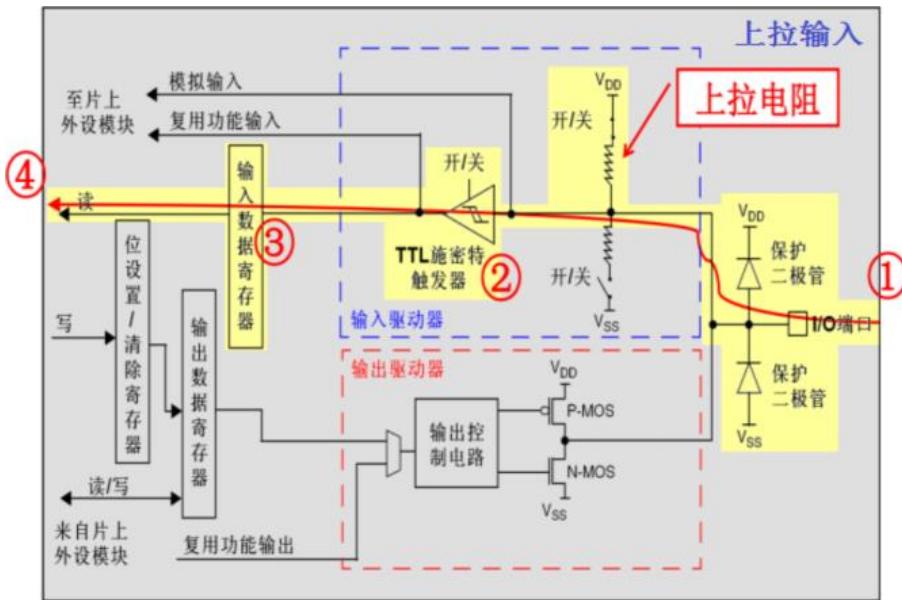
### 1.6.2 GPIO 的八种工作模式

#### 1. 浮空输入 GPIO\_Mode\_IN\_FLOATING



浮空输入模式下，I/O 端口的电平信号直接进入输入数据寄存器。MCU 直接读取 I/O 口电平，I/O 的电平状态是不确定的，完全由外部输入决定；如果在该引脚悬空（在无信号输入）的情况下，读取该端口的电平是不确定的，低功耗。

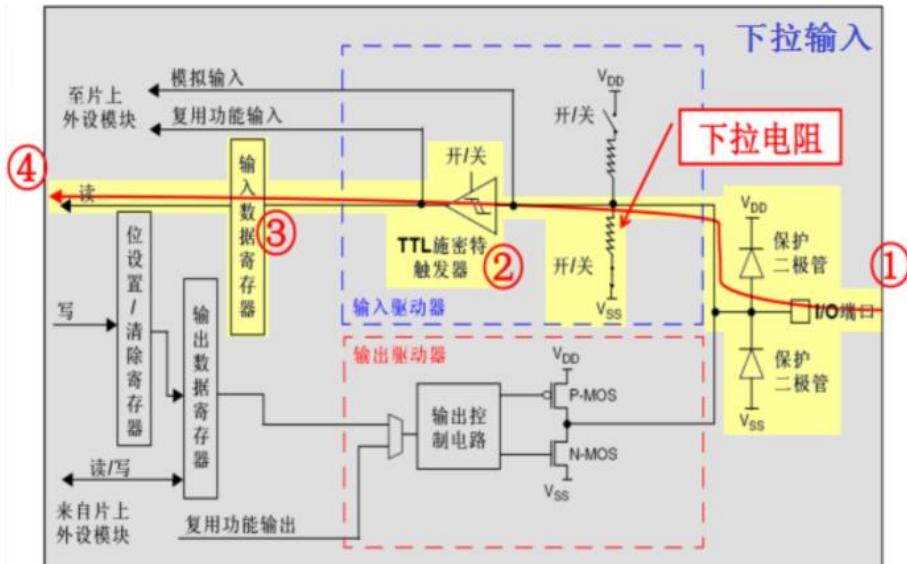
## 2. 上拉输入 GPIO\_Mode\_IPU (In Pull Up)



I/O 内部接上拉电阻，此时如果 I/O 口外部没有信号输入或者引脚悬空，I/O 口默认为高电平。如果 I/O 口输入低电平，那么引脚就为低电平，MCU 读取到的就是低电平。

钳位电平、增强驱动能力、抗干扰。可以用来检测外部信号；例如，按键等；

## 3. 下拉输入 GPIO\_Mode\_IPD (In Pull Down)

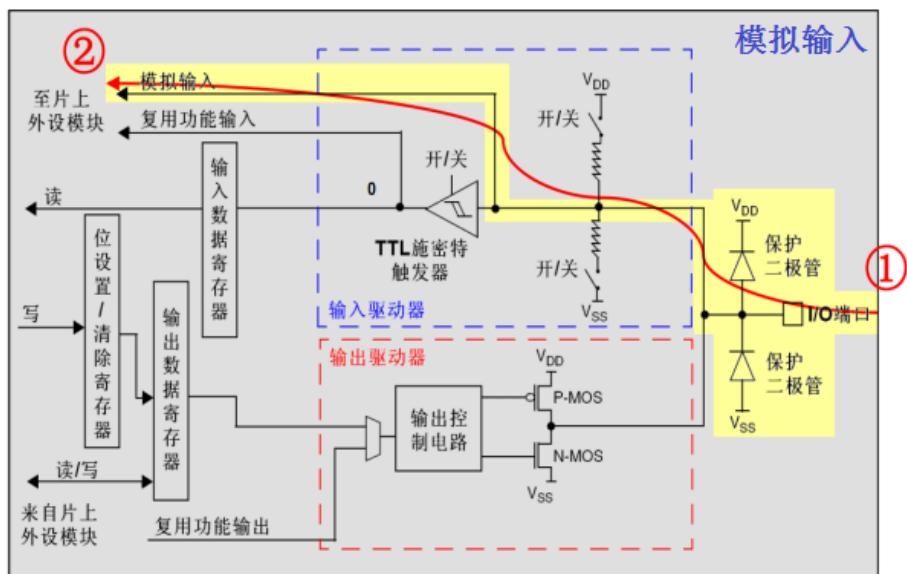


IO 内部接下拉电阻，此时如果 IO 口外部没有信号输入或者引脚悬空，IO 口默认为低电平。如果 I/O 口输入高电平，那么引脚就为高电平，MCU 读取到的就是高电平，可以用来检测外部信号；例如，按键等；

#### 4. 模拟输入 GPIO\_Mode\_AIN (Analog Input)

当 GPIO 引脚用于 ADC 采集电压的输入通道时，用作“模拟输入”功能，此时信号不经过施密特触发器，直接进入 ADC 模块，并且输入数据寄存器为空，CPU 不能在输入数据寄存器上读到引脚状态

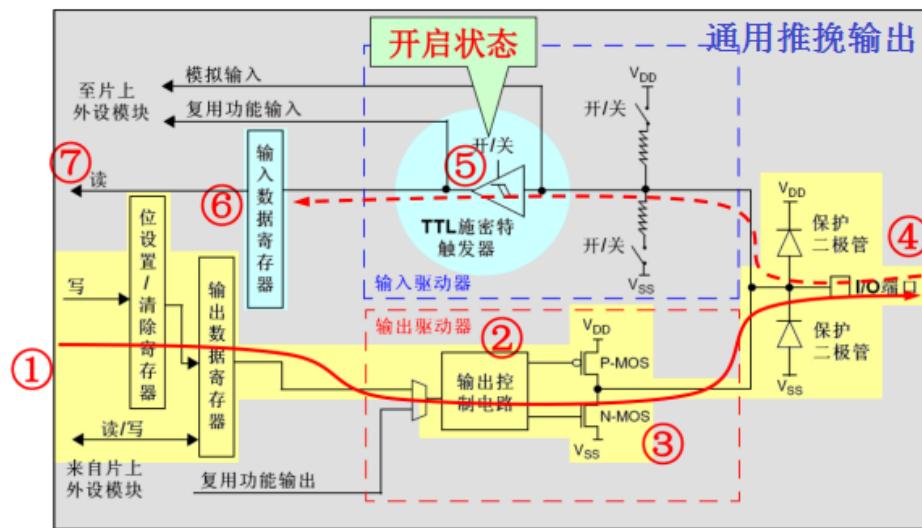
当 GPIO 用于模拟功能时，引脚的上、下拉电阻是不起作用的，这个时候即使配置了上拉或下拉模式，也不会影响到模拟信号的输入输出除了 ADC 和 DAC 要将 IO 配置为模拟通道之外其他外设功能一律要配置为复用功能模式，应用 ADC 模拟输入，或者低功耗下省电。



## 5.推挽输出 GPIO\_Mode\_Out\_PP (out push-pull)

在推挽输出模式时，N-MOS 管和 P-MOS 管都工作，如果我们控制 IO 输出为 0，低电平，则 P-MOS 管关闭，N-MOS 管导通，使输出低电平，I/O 端口的电平就是低电平，若控制输出为 1 高电平，则 P-MOS 管导通 N-MOS 管关闭，输出高电平，I/O 端口的电平就是高电平，外部上拉和下拉的作用是控制在没有输出时 IO 口电平

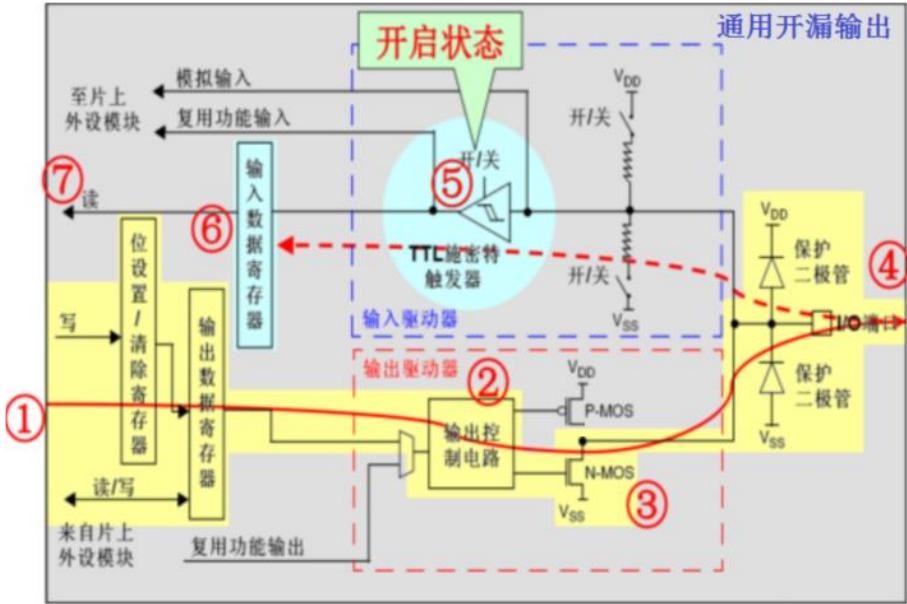
此时施密特触发器是打开的，即输入可用，通过输入数据寄存器 GPIOx\_IDR 可读取 I/O 的实际状态。I/O 口的电平一定是输出的电平，一般应用在输出电平为 0 和 3.3 伏而且需要高速切换开关状态的场合。



## 6.开漏输出 GPIO\_Mode\_Out\_OD (out open drain)

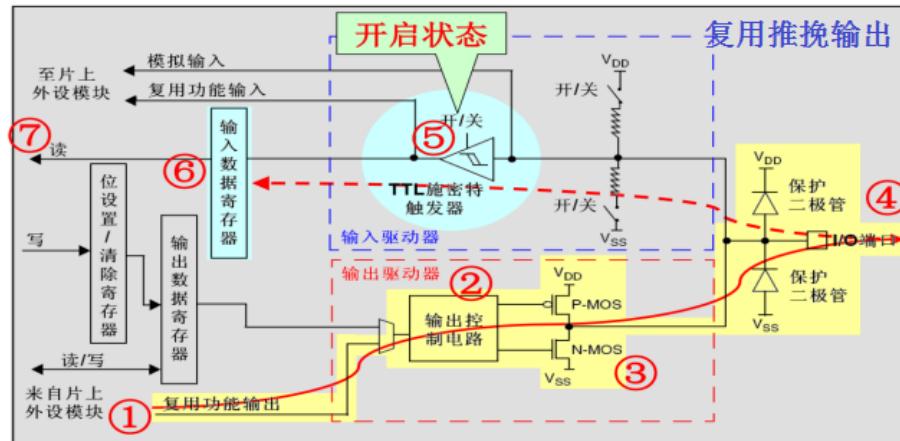
在开漏输出模式时，只有 N-MOS 管工作，如果我们控制输出为 0，低电平，则 P-MOS 管关闭，N-MOS 管导通，使输出低电平，I/O 端口的电平就是低电平，若控制输出为 1 时，高电平，则 P-MOS 管和 N-MOS 管都关闭，输出指令就不会起到作用，此时 I/O 端口的电平就不会由输出的高电平决定，而是由 I/O 端口外部的上拉或者下拉决定 如果没有上拉或者下拉 IO 口就处于悬空状态。

并且此时施密特触发器是打开的，即输入可用，通过输入数据寄存器 GPIOx\_IDR 可读取 I/O 的实际状态。I/O 口的电平不一定是输出的电平。一般应用在 I2C、SMBUS 通讯等需要"线与"功能的总线电路中。



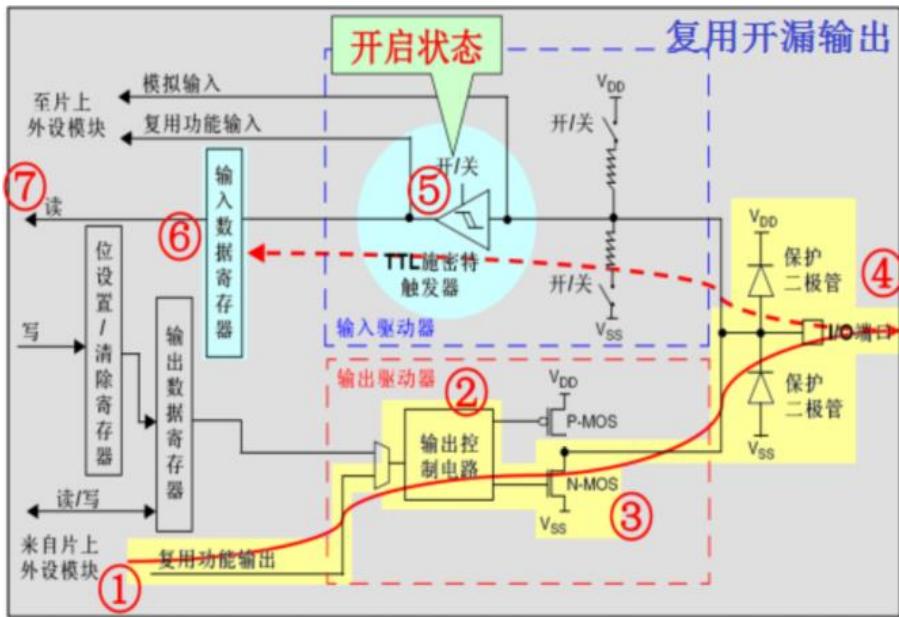
### 7. 复用推挽输出 GPIO\_AF\_PP (alternate function open push—pull)

GPIO 复用为其他外设(如 I2C)，输出数据寄存器 GPIOx\_ODR 无效；输出的高低电平的来源于其它外设，施密特触发器打开，输入可用，通过输入数据寄存器可获取 I/O 实际状态，除了输出信号的来源改变，其他与推挽输出功能相同。应用于片内外设功能（I2C 的 SCL,SDA）等。



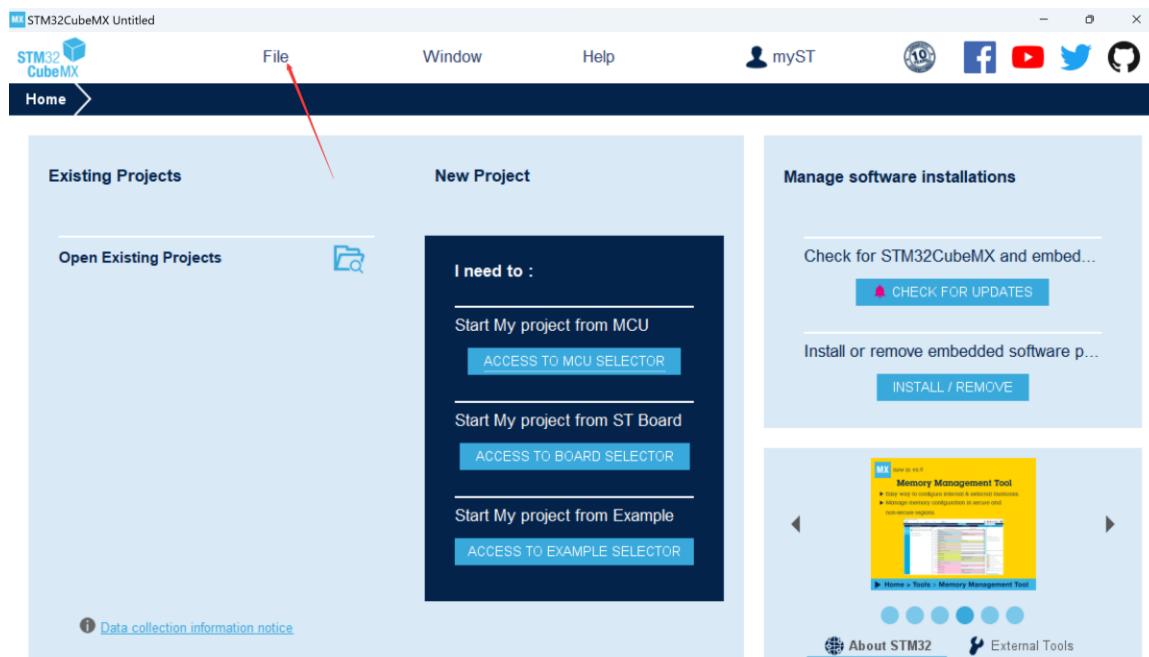
### 8. 复用开漏输出 GPIO\_AF\_OD (alternate function open drain)

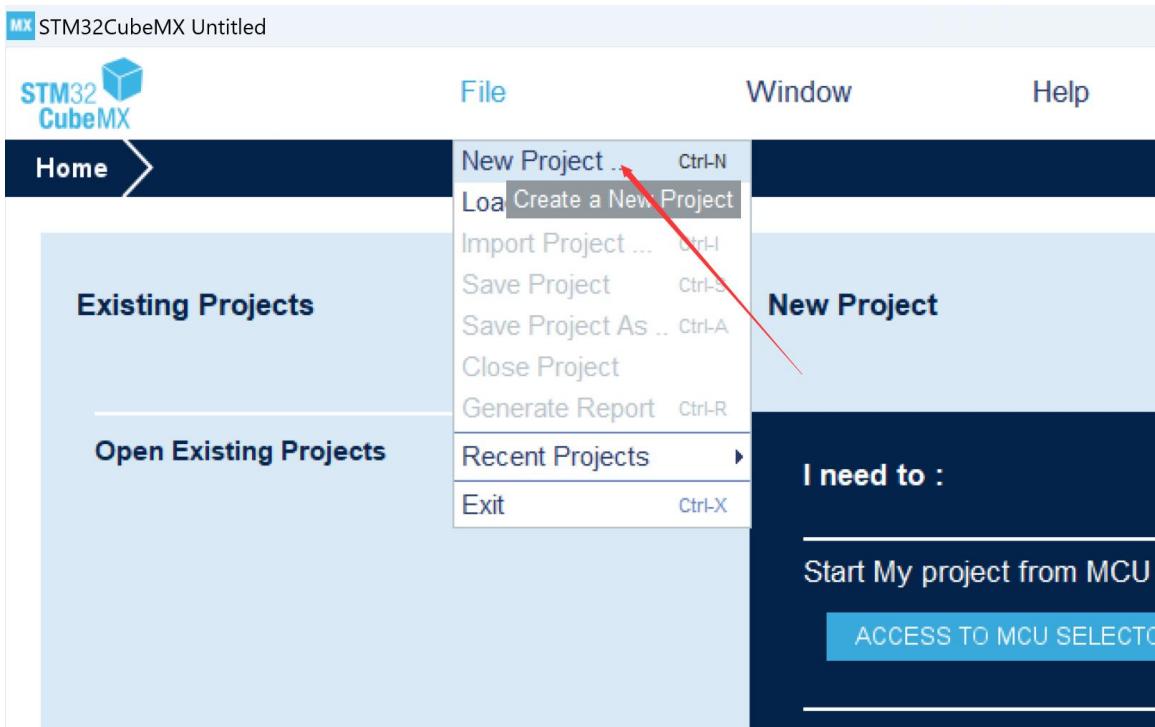
GPIO 复用为其他外设，输出数据寄存器 GPIOx\_ODR 无效；输出的高低电平的来源于其它外设，施密特触发器打开，输入可用，通过输入数据寄存器可获取 I/O 实际状态，除了输出信号的来源改变其他与开漏输出功能相同。应用于片内外设功能（TX1,MOSI,MISO,SCK,SS）等。



## 1.7 使用 CubeMX 配置使用 GPIO

### 1.7.1 新建工程





The screenshot shows the "New Project" dialog. The "MCU/MPU Selector" tab is active. The "Commercial Part Number" dropdown is set to "STM32F103ZET6". A red arrow points from the text "1.输入型号查找" (1. Enter model number search) to the dropdown. Another red arrow points from the text "2.点击型号" (2. Click the model) to the dropdown. A third red arrow points from the text "3.选择新建工程" (3. Select New Project) to the "Start P..." button.

MCU/MPU Selector | Board Selector | Example Selector | Cross Selector

MCU/MPU Filters

Commercial Part Number: STM32F103ZET6

1. 输入型号查找

Start P...

3. 选择新建工程

STM32F1 Series

**STM32F103ZET6**

Mainstream Performance line, Arm Cortex-M3 MCU with 512 Kbytes of Flash memory, 72 MHz CPU, motor control, USB and CAN

ACTIVE Product is in mass production

Unit Price for 10kU (US\$): 6.2857

LQFP 144 20x20x1.4 mm

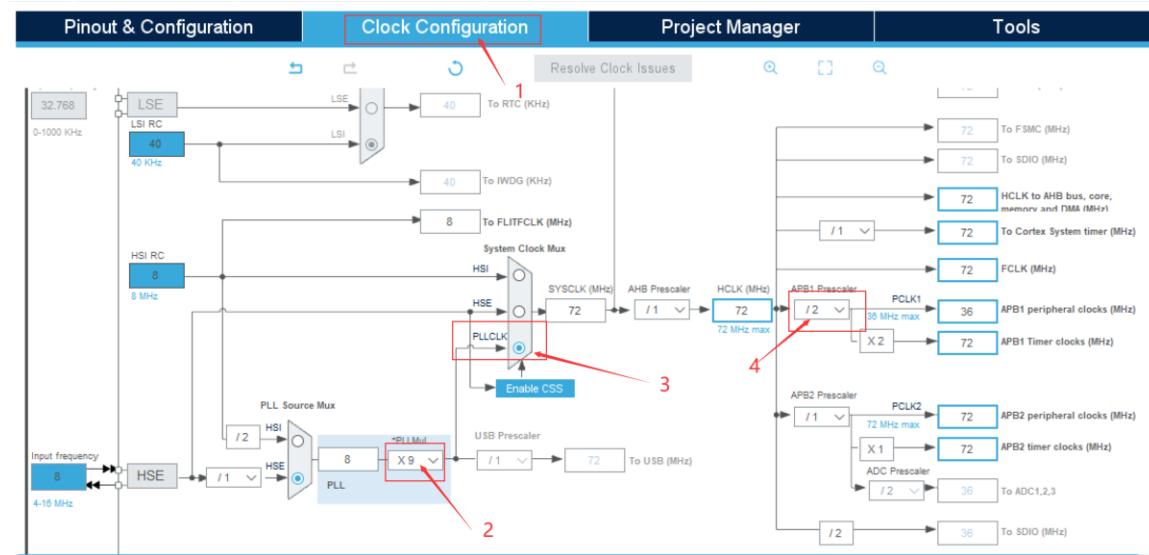
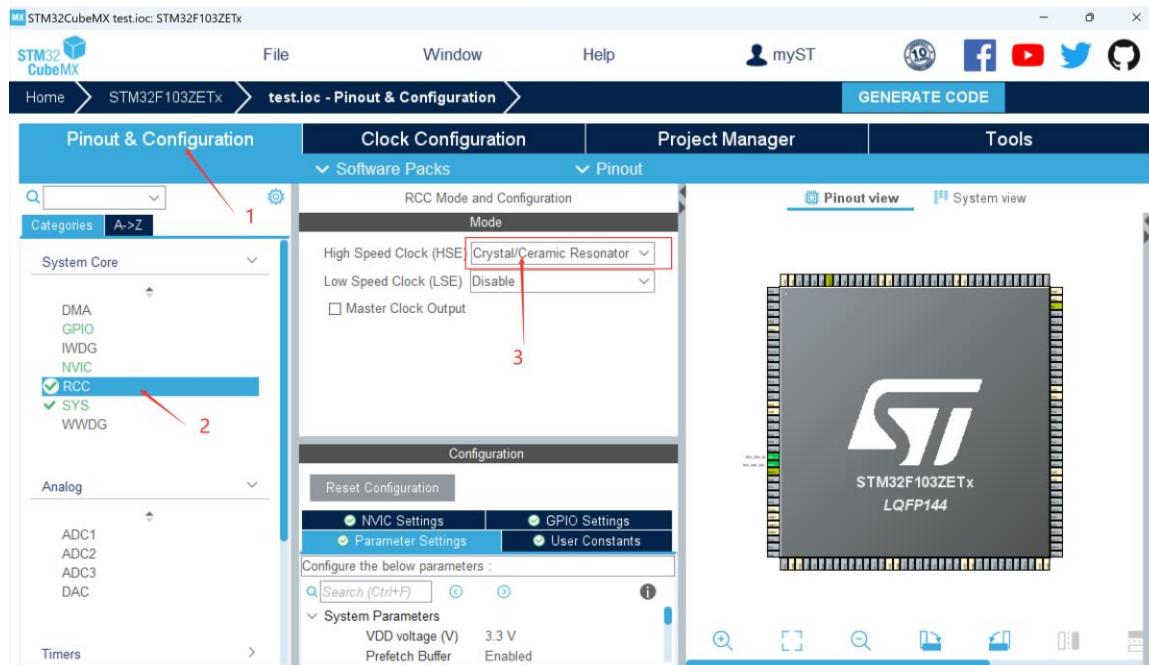
The STM32F103xC, STM32F103xD and STM32F103xE performance line family incorporates the high-performance ARM® Cortex®-M3 32-bit RISC core operating at a 72 MHz frequency, high-speed embedded memories (Flash memory up to 512 Kbytes and SRAM up to 64 Kbytes), and an

MCUs/MPUs List: 2 items

Commercial Pa...	Par...	Reference	Marketin...	Unit Price f...	B...	Package	Flash	RAM	Freq...
STM32F103ZET6	STM32F1...	Active	6.2857		LQFP 144 20x20x...	512 k...	64 kB...	112 72 M...	
STM32F103ZET6...	STM32F1...	Active	6.2857		LQFP 144 20x20x...	512 k...	64 kB...	112 72 M...	

2. 点击型号

## 1.7.2 配置时钟树

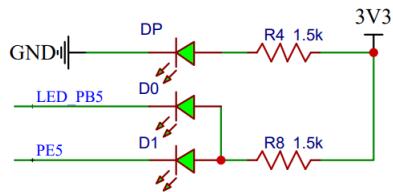


## 1.7.3 配置 GPIO

### 1. 读电路图

通过读电路图，配置 PB5 引脚为低电平，LED0 发光

## LED | 电源灯程控灯



## 2. Pinout view 选择 PB5 引脚的输出

Pinout & Configuration      Clock Configuration      Project Manager      Tools

Categories A-Z

System Core

- DMA
- GPIO**
- IWDG
- NVIC
- RCC
- SYS
- WWDG

Analog

- ADC1
- ADC2
- ADC3
- DAC

GPIO Mode and Configuration

Configuration

Group By Peripherals

RCC

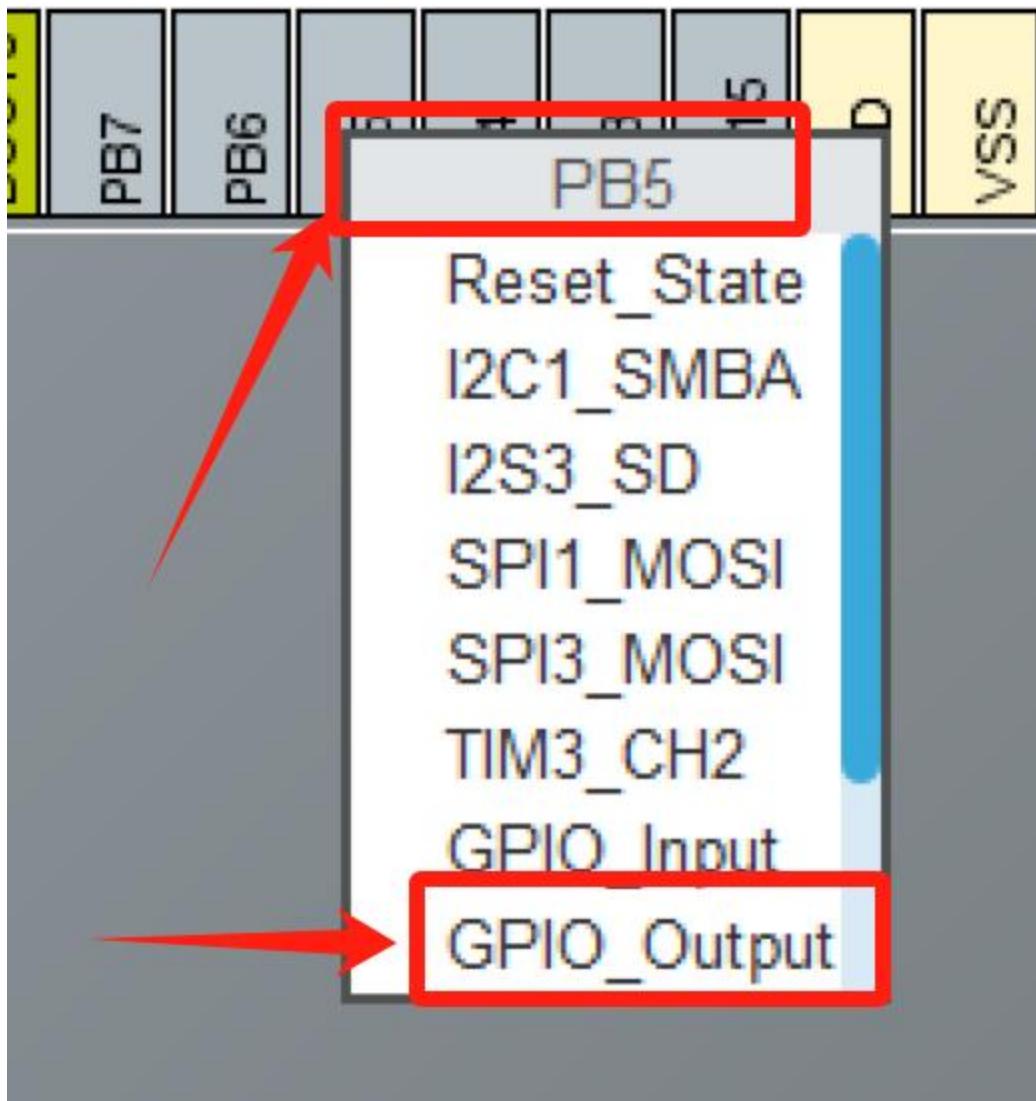
Search Signals

OS...	RC...	n/a	n/a	n/a	n/a	Mo...
OS...	RC...	n/a	n/a	n/a	n/a	

Select Pins from table to configure them. Multiple

Pinout view      System view

STM32F103ZETx  
LQFP144



### 3. GPIO 配置模块配置 PB5 参数

A screenshot of the Pinout & Configuration tool. The left sidebar shows categories like System Core (DMA, GPIO, IWDG, NVIC, RCC, SYS, WWDG), Analog (ADC1, ADC2, ADC3, DAC), and Timers. A red box highlights the 'GPIO' category under 'System Core'. A red arrow labeled '1' points to this box. The main area shows a table for 'GPIO Mode and Configuration' with a single row for 'PB5'. The table columns include Pin N., Signal on, GPIO out, GPIO mode, GPIO Pul., Maximum, User Label, and Modified. The values are: PB5, n/a, Low, Output Push Pull, No pull-up..., Low, and an unchecked checkbox. A red arrow labeled '2' points to the 'PB5' row in the table. Below the table, there are configuration options for 'PB5 Configuration': GPIO output level (Low), GPIO mode (Output Push Pull), and GPIO Pull-up/Pull-down (No pull-up and no pull-down). On the right, a pinout diagram shows pins PB1 through PB8 and VDD/VSS, with a red arrow labeled 'GPIO\_Output' pointing to the PB5 pin.

## PB5GPIO 配置

PB5 Configuration :

GPIO output level	Low
GPIO mode	Output Push Pull
GPIO Pull-up/Pull-down	No pull-up and no pull-down
Maximum output speed	Low
User Label	

Output level 参数 (输出电压) :

输出高电平: Low

输出低电平: High

GPIO output level	Low
GPIO mode	Low High

mode 参数 (输出模式) :

推挽输出: Output Push Pull

开漏输出: Output open drain

GPIO mode	Output Push Pull
GPIO Pull-up/Pull-down	Output Push Pull Output Open Drain

Pull-up/ Pull-updown 参数 (上下拉电阻) :

GPIO Pull-up/Pull-down	No pull-up and no pull-down
Maximum output speed	No pull-up and no pull-down Pull-up 上拉电阻 Pull-down 下拉电阻

Maximum output speed 参数 (输出速率) :

Maximum output speed	<input type="text" value="Low"/>
User Label	<input type="text" value="Low"/>
	<input type="text" value="Medium"/>
	<input type="text" value="High"/>

### User Label 参数 (别名)

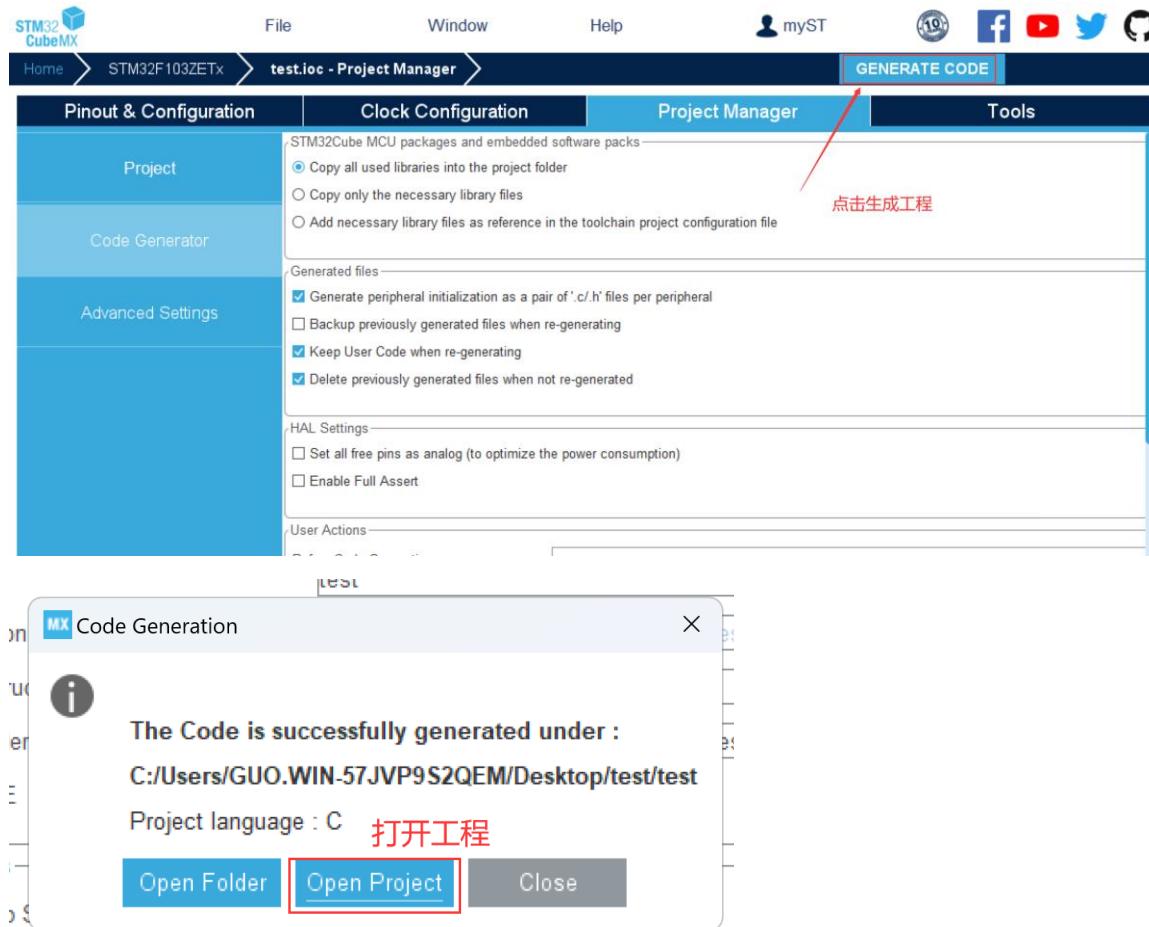
User Label

## 1.7.4 配置工程管理

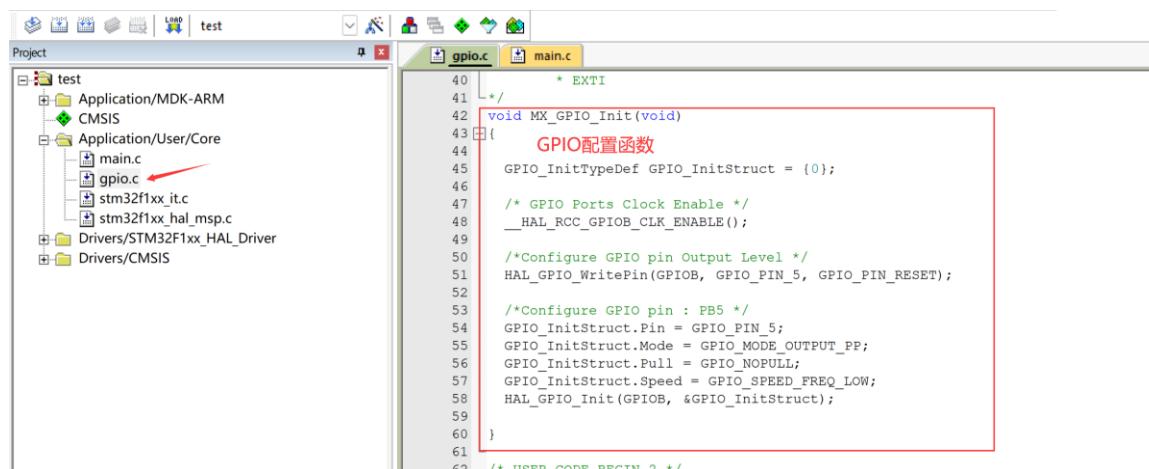
The screenshot shows the STM32CubeMX Project Manager interface with several annotations:

- Project Manager Tab:** Shows Project Settings with Project Name set to "工程名" (1).
- Code Generator Tab:** Shows Toolchain / IDE set to "MDK-ARM" (2). A red arrow points to the "Min Version" field set to "V5" with the note "注意IDE的版本" (Note IDE version).
- Advanced Settings Tab:**
  - Toolchain Folder Location:** Set to "C:\Users\GUO.WIN-57JVP9S2QEM\Desktop\test\test".
  - Toolchain / IDE:** Set to "MDK-ARM".
  - Linker Settings:** Minimum Heap Size is "0x200" and Minimum Stack Size is "0x400".
  - Thread-safe Settings:** Cortex-M3NS.
  - Firmware Package Name and Version:** Set to "STM32Cube\_FW\_F1\_V1.8.0". A red box highlights this field with the note "选择已经安装的1.8.0包" (Select the installed 1.8.0 package) and a checkbox labeled "Use latest" (取消勾选 - Uncheck).
- Project Tab:**
  - Toolchain Folder Location:** Set to "C:\Users\GUO.WIN-57JVP9S2QEM\Desktop\test\test".
  - Toolchain / IDE:** Set to "MDK-ARM".
  - Linker Settings:** Minimum Heap Size is "0x200" and Minimum Stack Size is "0x400".
  - Thread-safe Settings:** Cortex-M3NS.
  - Mcu and Firmware Package:** Mcu Reference is "STM32F103ZETx".
  - Generated files:**
    - Copy all used libraries into the project folder:** "复制所有的.c和.h文件到工程" (Copy all .c and .h files to the project folder) is selected.
    - Copy only the necessary library files:** "仅复制需要的" (Copy only what is needed) is unselected.
    - Add necessary library files as reference in the toolchain project configuration file:** "不复制, 从储存路径引用" (Do not copy, reference from storage path) is unselected.
    - Generate peripheral initialization as a pair of '.c/.h' files per peripheral:** "为每个外设成立独立的.c和.h文件" (Generate independent .c and .h files for each peripheral) is selected.
    - Backup previously generated files when re-generating:** "生成时备份以前的文档" (Backup previous files when regenerating) is unselected.
    - Keep User Code when re-generating:** "生成时保留用户代码" (Keep user code when regenerating) is selected.
    - Delete previously generated files when not re-generated:** "生成时删除以前的文件" (Delete previous files when not regenerating) is selected.

## 1.7.5 生成工程



## 1.7.6 keil5 打开工程文件



## 1.7.7 MX\_GPIO\_Init 函数解析

```

void MX_GPIO_Init(void)
{
    // 定义一个 GPIO_InitTypeDef 结构体变量，用于 GPIO 初始化配置，初始值为 0。
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    // 启用 GPIOB 端口的时钟。
    // 在配置任何 GPIO 端口之前，需要先使能其时钟。
    __HAL_RCC_GPIOB_CLK_ENABLE();

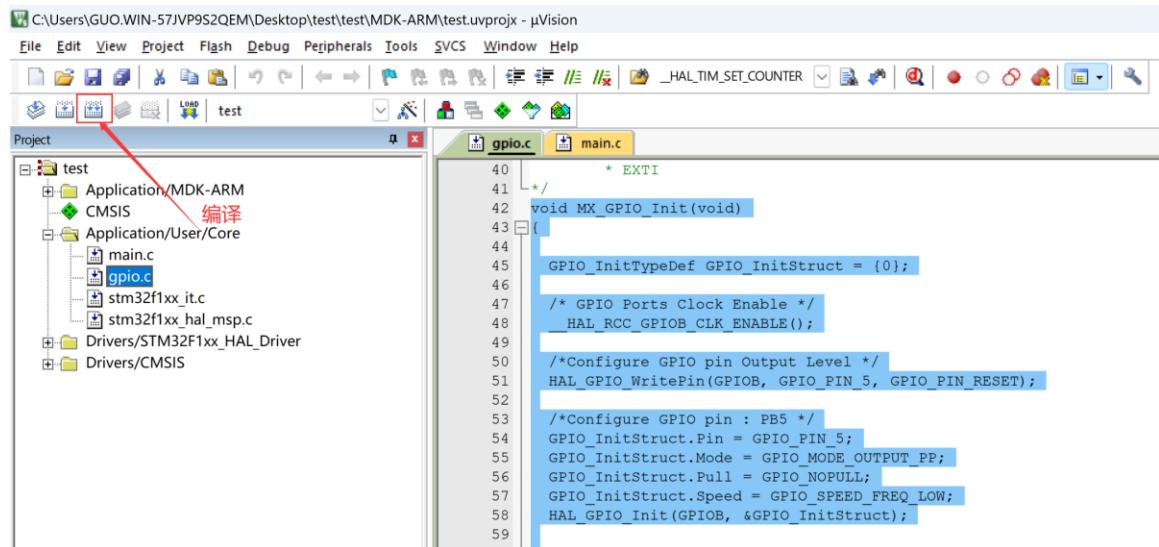
    // 配置 GPIOB 端口的第 5 个引脚为低电平。
    // 这是一个初始化步骤，确保引脚在配置前处于一个安全的状态。
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_RESET);

    // 开始配置 GPIOB 的第 5 个引脚。
    // 设置引脚号为 GPIO_PIN_5。
    GPIO_InitStruct.Pin = GPIO_PIN_5;
    // 设置引脚模式为推挽输出模式 (GPIO_MODE_OUTPUT_PP)。
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    // 设置引脚不使用内部上拉或下拉电阻 (GPIO_NOPULL)。
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    // 设置引脚的输出速度为低速 (GPIO_SPEED_FREQ_LOW)。
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    // 使用以上配置参数初始化 GPIOB 端口的第 5 个引脚。
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

}

```

### 1.7.8 将文件烧录至 STM32ZET6 开发板

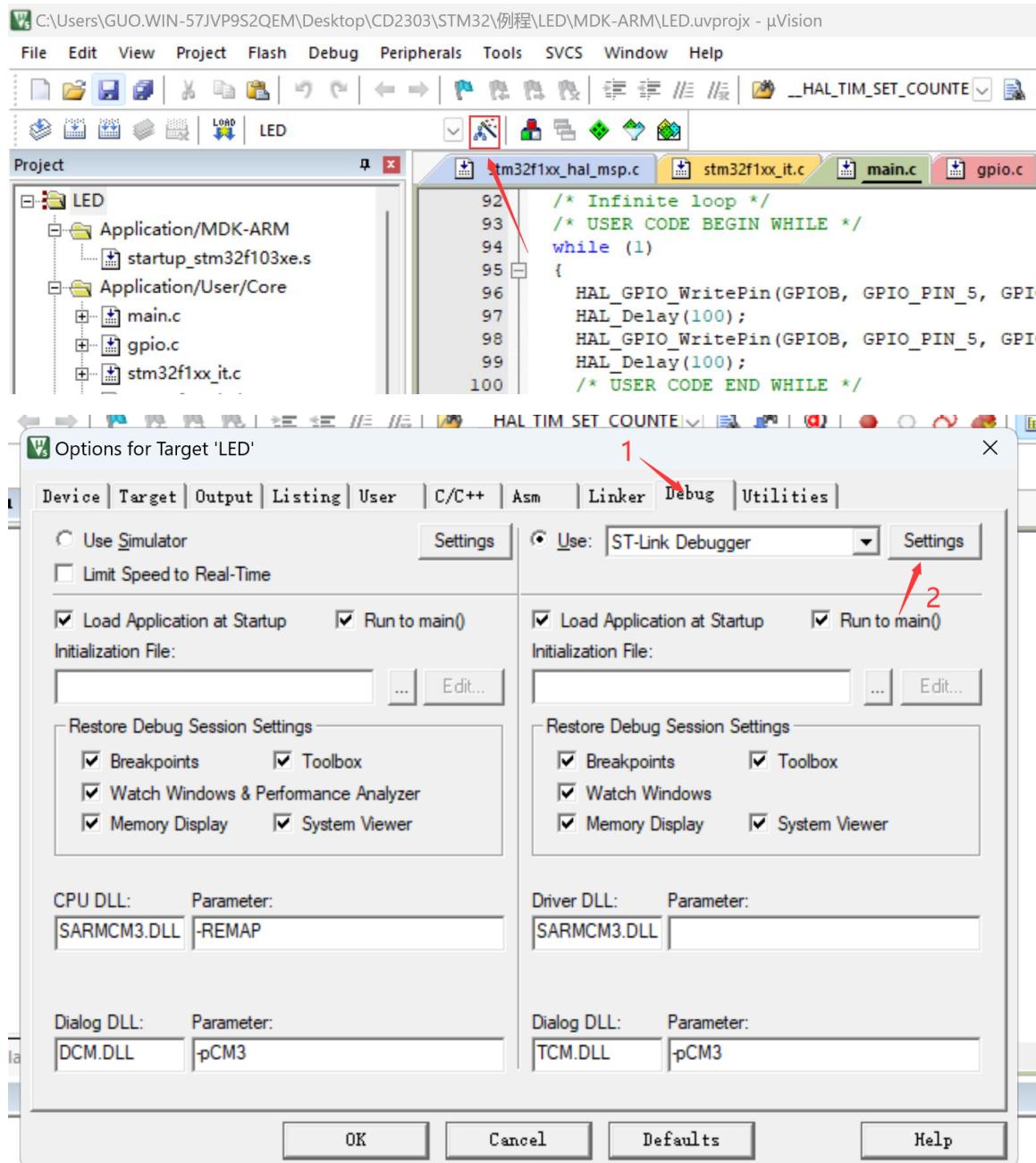


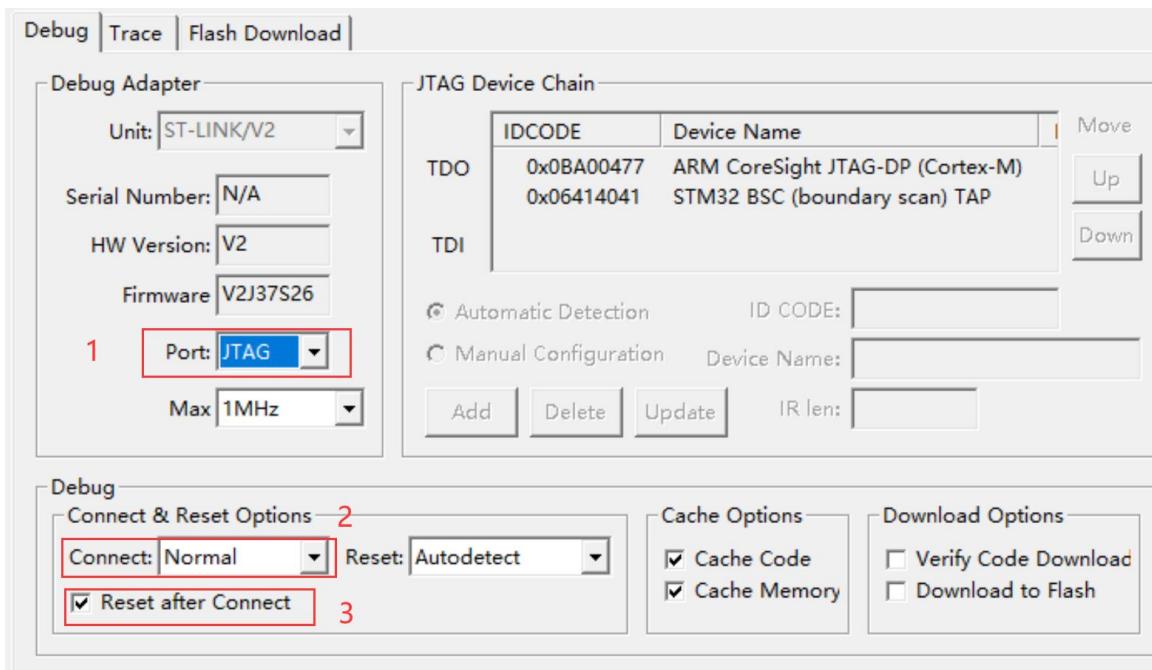
编译成功

Build Output

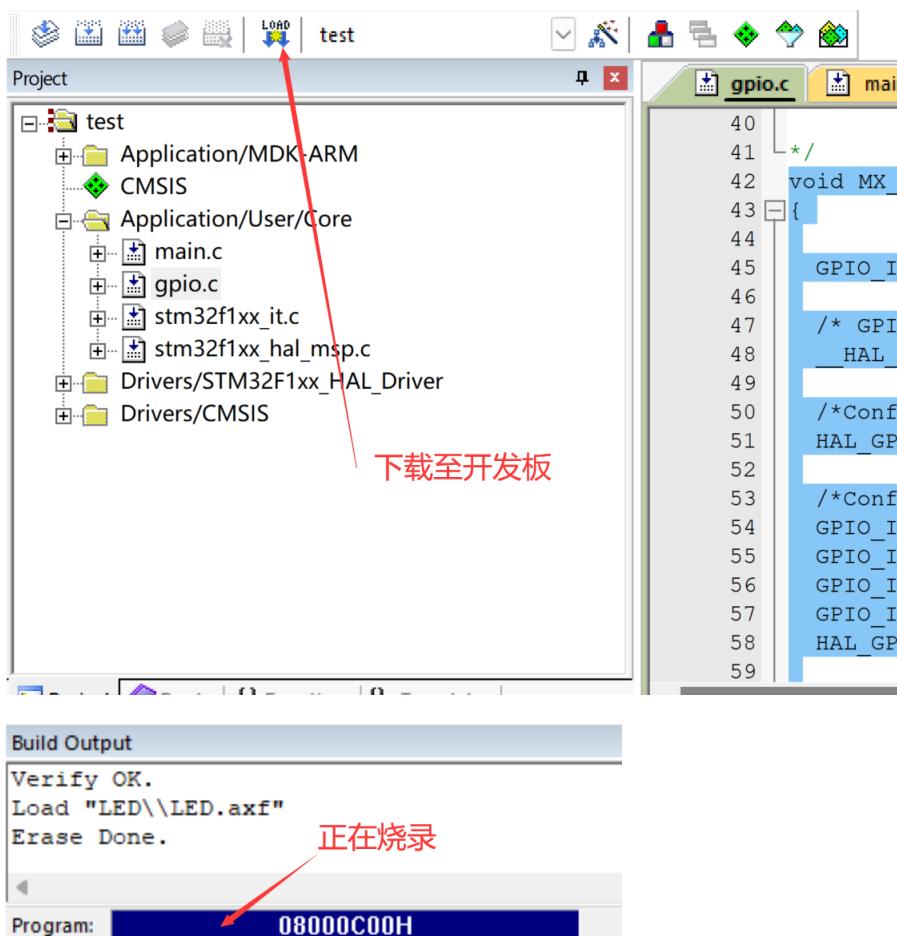
```
linking...
Program Size: Code=2696 RO-data=352 RW-data=16 ZI-data=1632
FromELF: creating hex file...
"test\test.axf" - 0 Error(s), 0 Warning(s).
```

## 配置烧录器





配置完成后点击确定



烧录完成



### 1.7.9 实验现象

LED 灯成功点亮

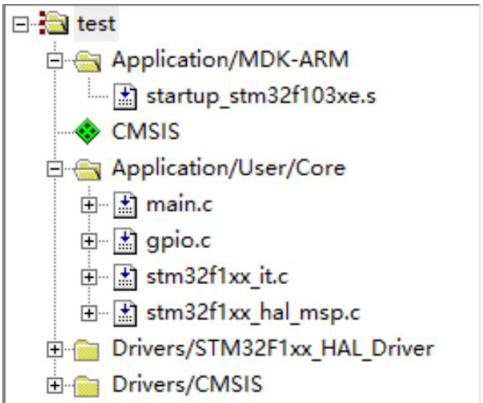


## 1.8 工程详解

### 1.8.1 工程目录

📁 Core	源文件与头文件	2023/11/27 14:50	文件夹
📁 Drivers	驱动相关	2023/11/27 14:50	文件夹
📁 MDK-ARM	MDK工程文件	2023/11/27 17:20	文件夹
📄 .mxproject		2023/11/27 16:53	MXPROJECT 文件 7 KB
MX test.ioc		2023/11/27 16:53	STM32CubeMX 5 KB

## 1.8.2 MDK-ARM 工程文件介绍



### **startup\_stm32f103xe.s**

针对 STM32F103xE 系列微控制器的启动文件。它包含了微控制器启动时执行的低级初始化代码，比如设置向量表和初始化数据。

### **Application/User/Core:**

**用户代码目录**

#### **main.c:**

这是程序的入口点，包含 **main** 函数。主要实现程序逻辑

#### **gpio.c:**

管理和实现与通用输入/输出（GPIO）引脚相关的功能。

#### **stm32f1xx\_it.c:**

包含中断服务例程（ISR）。在这里处理所有的中断请求。

#### **stm32f1xx\_hal\_msp.c:**

MSP 代表 “MCU Specific Package” 。这个文件包含了所有 HAL 库要求用户定义的回调函数，如用于初始化硬件（如 GPIO、DMA、ADC 等）的低级初始化函数。

### **Drivers/STM32F1xx\_HAL\_Driver:**

这个目录包含了 STM32F1 系列的 HAL（硬件抽象层）驱动代码，用于简化硬件接口的编程。

### **Drivers/CMSIS:**

CMSIS 代表 “Cortex Microcontroller Software Interface Standard” , 是 ARM 提供的一个硬件抽象层，旨在保持与 ARM Cortex 处理器兼容的代码的一致性。这个目录包含了 CMSIS 相关的核心文件。

后续主要在 **Application/User/Core** 文件中进行操作

### 1.8.3 主程序结构详解

```
/* USER CODE BEGIN Header */
/**
 * @file          : main.c
 * @brief         : Main program body
 ****
 * @attention
 *
 * Copyright (c) 2023 STMicroelectronics.
 * All rights reserved.
 *
 * This software is licensed under terms that can be found in the LICENSE
 * file
 * in the root directory of this software component.
 * If no LICENSE file comes with this software, it is provided AS-IS.
 *
 ****
 */
/* USER CODE END Header */

/* Includes -----
---*/
#include "main.h"
#include "gpio.h"

/* Private includes -----
---*/
/* USER CODE BEGIN Includes */
// 这里添加的私有包含
/* USER CODE END Includes */

/* Private typedef -----
---*/
/* USER CODE BEGIN PTD */
// 这里添加私有类型定义
/* USER CODE END PTD */
```

```
/* Private define -----
---*/
/* USER CODE BEGIN PD */
// 这里添加私有宏定义
/* USER CODE END PD */

/* Private macro -----
---*/
/* USER CODE BEGIN PM */
// 这里添加私有宏
/* USER CODE END PM */

/* Private variables -----
---*/
/* USER CODE BEGIN PV */
// 这里添加私有变量
/* USER CODE END PV */

/* Private function prototypes -----
---*/
void SystemClock_Config(void);
/* USER CODE BEGIN PFP */
// 这里添加私有函数原型
/* USER CODE END PFP */

/* USER CODE BEGIN 0 */
// 这里添加自定义代码
/* USER CODE END 0 */

/**
 * @brief  The application entry point.
 * @retval int
 */
int main(void)
{
    /* USER CODE BEGIN 1 */
    // 初始化代码
    /* USER CODE END 1 */

    /* MCU Configuration-----*/
---*/

    /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
    HAL_Init();
}
```

```
/* USER CODE BEGIN Init */
// 用户自定义的初始化
/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */
// 系统初始化代码
/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();

/* USER CODE BEGIN 2 */
// 这里添加程序代码
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE BEGIN 3 */
    // 主循环
}
/* USER CODE END 3 */
}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
    // 系统时钟配置代码
}

/* USER CODE BEGIN 4 */
// 这里添加更多的用户代码
/* USER CODE END 4 */

/**
 * @brief This function is executed in case of error occurrence.
 * @retval None

```

```

/*
void Error_Handler(void)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    // 错误处理代码
    /* USER CODE END Error_Handler_Debug */
}

#ifndef 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(uint8_t *file, uint32_t line)
{
    /* USER CODE BEGIN 6 */
    // 断言失败处理
    /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

```

后续修改时，需要在对应位置修改

## 1.9 GPIO 相关 API

### 1.9.1 相关 API

#### 1. HAL\_GPIO\_Init

```

/**
 * 函数功能：初始化 GPIOx 引脚按照 GPIO_Init 指定的参数。
 *
 * void HAL_GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_Init)
 *
 * 参数：
 *     GPIOx - GPIO 端口（如 GPIOA, GPIOB 等）。
 *     GPIO_Init - 指向 GPIO_InitTypeDef 结构的指针，包含 GPIO 引脚的配置信息。
 * 返回值：无。
 */

```

#### 2. HAL\_GPIO\_DeInit

```

/**
 * 函数功能：将 GPIOx 引脚重置为默认状态。
 *

```

```
* void HAL_GPIO_DeInit(GPIO_TypeDef* GPIOx, uint32_t GPIO_Pin)
*
* 参数:
*   GPIOx - GPIO 端口 (如 GPIOA, GPIOB 等)。
*   GPIO_Pin - 指定要重置的 GPIO 引脚。这个值可以是 GPIO_PIN_x (x 可以是 0 到
15)。
* 返回值: 无。
*/
```

### 3. HAL\_GPIO\_ReadPin

```
/**
* 函数功能: 读取指定 GPIO 端口的引脚状态。
*
* GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
*
* 参数:
*   GPIOx - GPIO 端口 (如 GPIOA, GPIOB 等)。
*   GPIO_Pin - 指定要读取的 GPIO 引脚。这个值可以是 GPIO_PIN_x (x 可以是 0 到
15)。
* 返回值: GPIO_PinState - 引脚的当前状态 (GPIO_PIN_SET 或 GPIO_PIN_RESET)。
*/

```

### 4. HAL\_GPIO\_WritePin

```
/**
* 函数功能: 设置或清除选定的 GPIO 端口引脚。
*
* void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
GPIO_PinState PinState)
*
* 参数:
*   GPIOx - GPIO 端口 (如 GPIOA, GPIOB 等)。
*   GPIO_Pin - 指定要写入的 GPIO 引脚。这个值可以是 GPIO_PIN_x (x 可以是 0 到
15)。
*   PinState - 要写入引脚的状态 (GPIO_PIN_SET 或 GPIO_PIN_RESET)。
* 返回值: 无。
*/

```

### 5. HAL\_GPIO\_TogglePin

```
/**
* 函数功能: 切换指定 GPIO 端口的引脚状态。
*
* void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
*
* 参数:

```

```
*      GPIOx - GPIO 端口（如 GPIOA, GPIOB 等）。
*      GPIO_Pin - 指定要切换的 GPIO 引脚。这个值可以是 GPIO_PIN_x (x 可以是 0 到
15)。
* 返回值: 无。
*/
```

## 6. HAL\_GPIO\_LockPin

```
/**
 * 函数功能: 锁定 GPIO 引脚的配置, 直到下一次复位。
*
* HAL_StatusTypeDef HAL_GPIO_LockPin(GPIO_TypeDef* GPIOx, uint16_t
GPIO_Pin)
*
* 参数:
*      GPIOx - GPIO 端口（如 GPIOA, GPIOB 等）。
*      GPIO_Pin - 指定要锁定的 GPIO 引脚。这个值可以是 GPIO_PIN_x 的任意组合 (x
可以是 0 到 15)。
* 返回值: HAL_StatusTypeDef - 操作的状态 (HAL_OK 或 HAL_ERROR)。
*/

```

## 7. HAL\_GPIO\_EXTI\_IRQHandler

```
/**
 * 函数功能: 处理外部中断/事件请求。
*
* void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
*
* 参数:
*      GPIO_Pin - 指定连接到 EXTI 线的引脚。
* 返回值: 无。
*/

```

## 8. HAL\_GPIO\_EXTI\_Callback

```
/**
 * 函数功能: 外部中断/事件线的回调函数。
*
* void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
*
* 参数:
*      GPIO_Pin - 指定连接到 EXTI 线的引脚。
* 返回值: 无。
*/

```

## 9. HAL\_Delay

```
/**
```

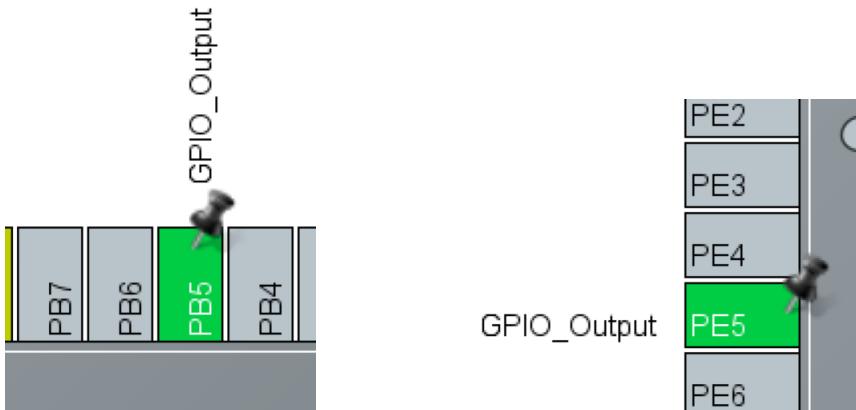
```

* 函数功能：在指定时间内阻塞。
*
* void HAL_Delay(uint32_t Delay)
*
* 参数：
*     Delay - 延时长度，以毫秒为单位。
* 返回值：无。
*/

```

### 1.9.2 hal 库实现跑马灯

1. 使用 cubemx 生成基础代码，时钟配置，工程配置等与 1.7 相同，配置 PB5 与 PE5 为 OUTPUT



2. 在 main 函数的中

在用户函数 2 处添加

```

/* USER CODE BEGIN 2 */
    /*设置 E5 初始为高电平*/
    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_5,GPIO_PIN_SET);
    /*设置 B5 初始为低电平*/
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5,GPIO_PIN_RESET);
/* USER CODE END 2 */

```

在主循环处添加

```

while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    /*切换 GPIO 状态*/
}

```

```

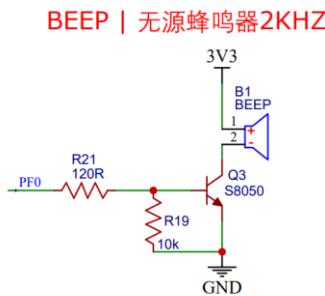
    HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5);
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_5);

    /* 延迟 1 秒 */
    HAL_Delay(1000);
}

```

### 1.9.3 作业：使用 GPIO 驱动蜂鸣器

使用 GPIO 驱动蜂鸣器，相关电路图：



## 1.10 stm32 嵌套向量中断控制器（NVIC）

### 1.10.1 NVIC 的介绍

NVIC (Nest Vector Interrupt Controller)，嵌套向量中断控制器，作用是管理中断嵌套，核心任务是管理中断优先级。特点：

1. 68 个可屏蔽中断通道(不包含 16 个 Cortex-M3 的中断线)
2. 16 个可编程的优先等级(使用了 4 位中断优先级)
3. 低延迟的异常和中断处理
4. 电源管理控制
5. 系统控制寄存器的实现

嵌套向量中断控制器(NVIC)和处理器核的接口紧密相连，可以实现中断的低延迟处理和高效地处理晚到的中断。

### 1.10.2 NVIC 的优先级

NVIC 给每个中断赋予抢占优先级和响应优先级。关系如下：

1. 拥有较高抢占优先级的中断可以打断抢占优先级较低的中断
2. 若两个抢占优先级的中断同时挂起，则优先执行响应优先级较高的中断

3. 若两个挂起的中断优先级都一致，则优先执行位于中断向量表中位置较高的中断
  4. 响应优先级不会造成中断嵌套，也就是说中断嵌套是由抢占优先级决定的
- 每个中断源都需要被指定这两种优先级，Cortex-M3 核定义了 8 个 bit 用于设置中断源的优先级。但是 Cortex-M3 允许具有较少中断源时使用较少的寄存器位指定中断源的优先级，因此 STM32 中断优先级的寄存器位只用到 AIRCR 高四位。

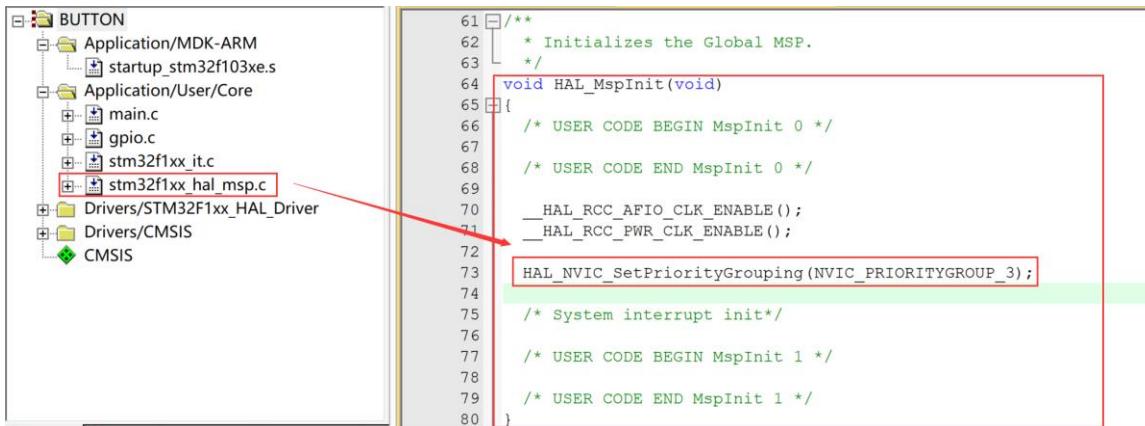
相关 NVIC 设置函数如下

```
/**
 * 函数功能：设置 NVIC 优先级分组，配置抢占优先级和子优先级的位长度。
 *
 * void HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup)
 *
 * 参数：
 *     PriorityGroup - 定义 NVIC 优先级分组。这个参数可以是以下值之一：
 *         @arg NVIC_PRIORITYGROUP_0: 抢占优先级 0 位，子优先级 4 位
 *         @arg NVIC_PRIORITYGROUP_1: 抢占优先级 1 位，子优先级 3 位
 *         @arg NVIC_PRIORITYGROUP_2: 抢占优先级 2 位，子优先级 2 位
 *         @arg NVIC_PRIORITYGROUP_3: 抢占优先级 3 位，子优先级 1 位
 *         @arg NVIC_PRIORITYGROUP_4: 抢占优先级 4 位，子优先级 0 位
 *
 * 注意：当选择 NVIC_PRIORITYGROUP_0 时，中断抢占将不再可能。挂起的中断优先级将仅由子优先级管理。
 * 返回值：无。
 */

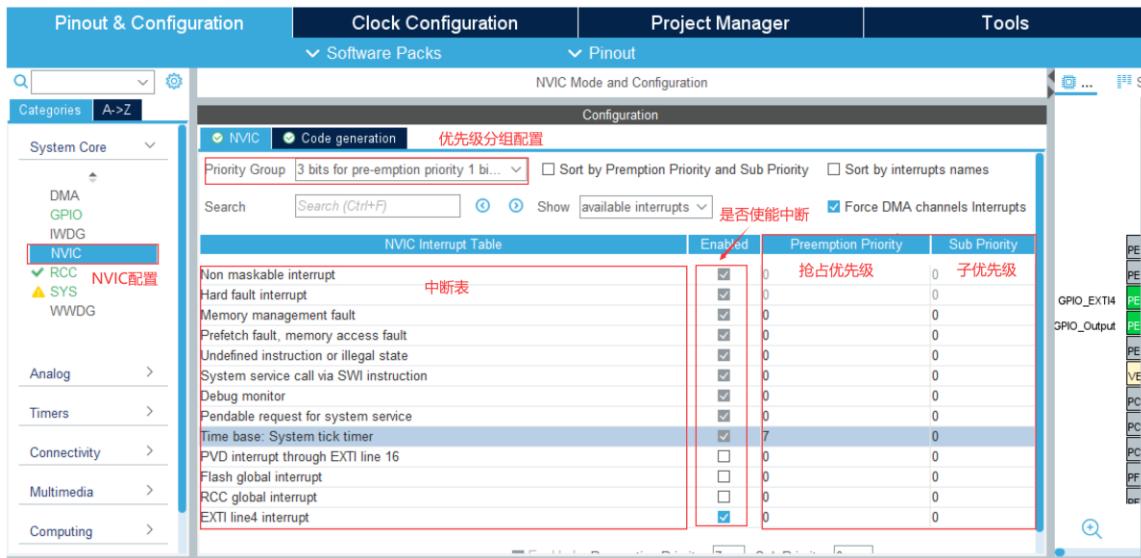
```

在 Cubemx 默认设置中断优先级分组为 `NVIC_PRIORITYGROUP_4`: 抢占优先级 4 位，子优先级 0 位。

如果需要修改中断优先级分组，则在如下位置进行修改。



### 1.10.3 Cubemx 配置中断优先级



### 1.10.4 中断的具体行为

参考手册：Cortex-M3 权威指南 CnR2（电子书）.pdf 第 9 章

当 CM3 开始响应一个中断时，会做如下动作：

1. 入栈：把 8 个寄存器的值压入栈
2. 取向量：从向量表中找出对应的服务程序入口地址
3. 选择堆栈指针 MSP(主堆栈)/PSP(进程堆栈)，更新堆栈指针 SP，更新链接寄存器 LR，更新程序计数器 PC

入栈 响应异常的第一个动作，就是自动保存现场，依次把 xPSR、PC, LR, R12 以及 R3-R0 由硬件寄存器自动压入适当的堆栈中。取向量 数据总线（系统总线）在执行入栈的时候，指令总线从向量表中找出正确的异常向量，然后在服务程序的入口处预取指。（由此可以看到各自都有专用总线的好处：入栈与取指这两个工作能同时进行）更新寄存器 在入栈和取向量操作完成之后，执行服务例程之前，还要更新一系列的寄存器。

- **SP：**在入栈后会把堆栈指针更新到新的位置。在执行服务例程时，将由 MSP 负责对堆栈的访问。
- **PSR：**更新 IPSR 位段的值为新响应的异常编号。
- **PC：**在取向量完成后，PC 将指向服务例程的入口地址。
- **LR：**在出入 ISR（Interrupt Service Routines）中断服务程序的时候，LR 的值将得到更新

(在异常进入时由系统计算并赋给 LR，并在异常返回时使用它)

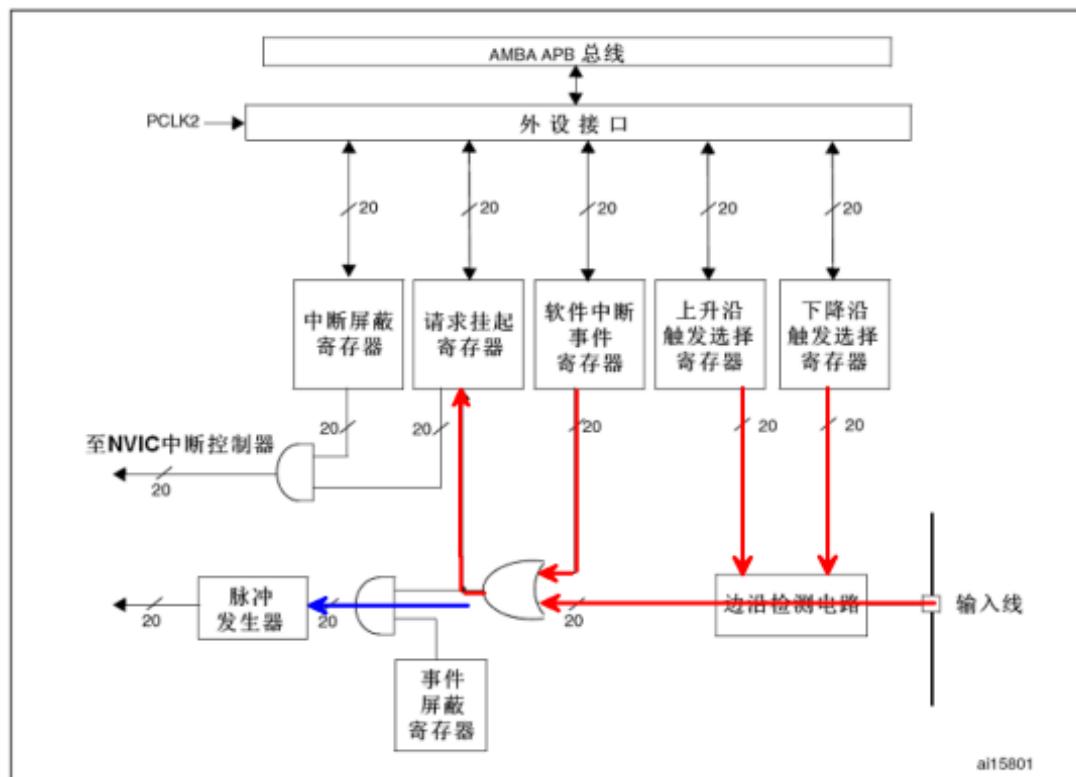
## 异常/中断返回

当异常服务例程执行完毕后，需要恢复先前的系统状态，才能使被中断的程序得以继续执行。

异常/中断处理完成后，执行如下处理： 出栈：恢复先前压入栈中的寄存器，堆栈指针的值也改回先前的值 更新 NVIC 寄存器：伴随着异常的返回，它的活动位也被硬件清除

### 1.10.5 外部中断的介绍

对于互联型产品，外部中断/事件控制器由 20 个产生事件/中断请求的边沿检测器组成，对于其它产品，则有 19 个能产生事件/中断请求的边沿检测器。每个输入线可以独立地配置输入类型(脉冲或挂起)和对应的触发事件(上升沿或下降沿或者双边沿都触发)。每个输入线都可以独立地被屏蔽。挂起寄存器保持着状态线的中断请求。



EXTI 可分为两大部分功能，一个是产生中断，另一个是产生事件

中断：

信号从输入线输入，经过边沿检测电路来控制信号触发（根据上升沿下降沿触发选择寄存器的设置来控制），如果检测到有效信号后，将该有效信号输出到或门电路，由红色箭头方向经过请求挂起寄存器和中断屏蔽寄存器，到达与门电路，条件满足送至 NVIC。

事件：

信号从输入线输入，经过边沿检测电路来控制信号触发（根据上升沿下降沿触发选择寄存器的设置来控制），如果检测到有效信号后，将该有效信号输出到或门电路，由蓝色箭头方向经过与门电路，送至买脉冲发生器，产生脉冲。这个脉冲信号可以给其他外设电路使用，比如定时器 TIM、模拟数字转换器 ADC 等等，这样的脉冲信号一般用来触发 TIM 或者 ADC 开始转换。

事件和中断的区别：

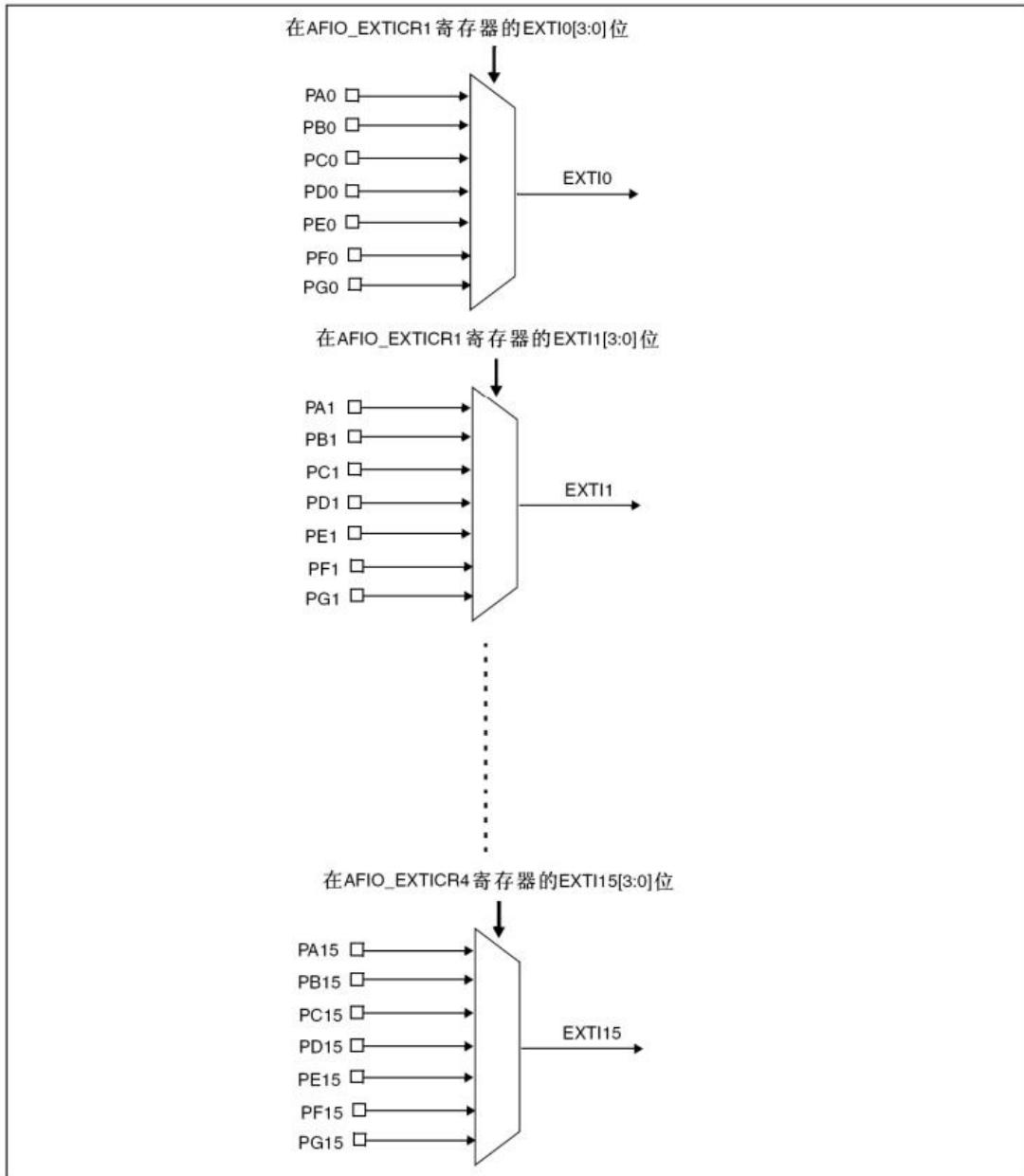
产生中断线路目的是把输入信号输入到 NVIC，进一步会运行中断服务函数，实现功能，这样是软件级的。而产生事件线路目的就是传输一个脉冲信号给其他外设使用，并且是电路级别的信号传输，属于硬件级的。

#### 1.10.6 中断的主要特性

- 每个中断/事件都有独立的触发和屏蔽
- 每个中断线都有专用的状态位
- 支持多达 20 个软件的中断/事件请求
- 检测脉冲宽度低于 APB2 时钟宽度的外部信号

#### 1.10.7 外部中断/事件线路映像

通用 I/O 端口以下图的方式连接到 16 个外部中断/事件线上。



1. 通过AFIO\_EXTICRx配置GPIO线上的外部中断/事件，必须先使能AFIO时钟。对于小容量、中容量和大容量的产品，参见6.3.7节；对于互联型产品，参见7.3.7节。

另外四个EXTI线的连接方式如下：

- EXTI线16连接到PWD输出
- EXTI线17连接到RTC闹钟事件
- EXTI线18连接到USB唤醒事件
- EXTI线19连接到以太网唤醒事件(只适用于互联型产品)

### 1.10.8 中断相关 API 及数据结构

#### 1. HAL\_NVIC\_SetPriority

函数功能：配置中断优先级，数字越低，中断优先级越高

```
HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority);  
参数：  
typedef enum  
{  
    /****** Cortex-M3 Processor Exceptions Numbers  
*****/  
    NonMaskableInt_IRQn      = -14,    /*!< 2 不可屏蔽中断 */  
    HardFault_IRQn           = -13,    /*!< 3 Cortex-M3 硬故障中断 */  
    MemoryManagement_IRQn    = -12,    /*!< 4 Cortex-M3 内存管理中断 */  
    BusFault_IRQn            = -11,    /*!< 5 Cortex-M3 总线故障中断 */  
    UsageFault_IRQn          = -10,    /*!< 6 Cortex-M3 使用故障中断 */  
    SVCall_IRQn              = -5,     /*!< 11 Cortex-M3 SV 调用中断 */  
    DebugMonitor_IRQn         = -4,     /*!< 12 Cortex-M3 调试监视器中断 */  
    PendSV_IRQn              = -2,     /*!< 14 Cortex-M3 Pend SV 中断 */  
    SysTick_IRQn              = -1,     /*!< 15 Cortex-M3 系统滴答定时器中断 */  
}/  
  
    /****** STM32 specific Interrupt Numbers  
*****/  
    WWDG_IRQn                = 0,      /*!< 窗口看门狗中断 */  
    PVD_IRQn                 = 1,      /*!< PVD 通过 EXTI 线检测中断 */  
    TAMPER_IRQn               = 2,      /*!< 篡改中断 */  
    RTC_IRQn                  = 3,      /*!< RTC 全局中断 */  
    FLASH_IRQn                = 4,      /*!< FLASH 全局中断 */  
    RCC_IRQn                  = 5,      /*!< RCC 全局中断 */  
    EXTI0_IRQn                = 6,      /*!< EXTI 线 0 中断 */  
    EXTI1_IRQn                = 7,      /*!< EXTI 线 1 中断 */  
    EXTI2_IRQn                = 8,      /*!< EXTI 线 2 中断 */  
    EXTI3_IRQn                = 9,      /*!< EXTI 线 3 中断 */  
    EXTI4_IRQn                = 10,     /*!< EXTI 线 4 中断 */  
    DMA1_Channel1_IRQn         = 11,     /*!< DMA1 通道 1 全局中断 */  
    DMA1_Channel2_IRQn         = 12,     /*!< DMA1 通道 2 全局中断 */  
    DMA1_Channel3_IRQn         = 13,     /*!< DMA1 通道 3 全局中断 */  
    DMA1_Channel4_IRQn         = 14,     /*!< DMA1 通道 4 全局中断 */  
    DMA1_Channel5_IRQn         = 15,     /*!< DMA1 通道 5 全局中断 */  
    DMA1_Channel6_IRQn         = 16,     /*!< DMA1 通道 6 全局中断 */  
    DMA1_Channel7_IRQn         = 17,     /*!< DMA1 通道 7 全局中断 */  
    ADC1_2_IRQn                = 18,     /*!< ADC1 和 ADC2 全局中断 */  
    USB_HP_CAN1_TX_IRQn        = 19,     /*!< USB 设备高优先级或 CAN1 TX 中断 */  
}/  
    USB_LP_CAN1_RX0_IRQn       = 20,     /*!< USB 设备低优先级或 CAN1 RX0 中断 */  
}/  
    CAN1_RX1_IRQn              = 21,     /*!< CAN1 RX1 中断 */
```

```

CAN1_SCE_IRQHandler      = 22,      /*!< CAN1 SCE 中断 */
EXTI9_5_IRQHandler      = 23,      /*!< 外部线[9:5]中断 */
TIM1_BRK_IRQHandler     = 24,      /*!< TIM1 Break 中断 */
TIM1_UP_IRQHandler       = 25,      /*!< TIM1 Update 中断 */
TIM1_TRG_COM_IRQHandler = 26,      /*!< TIM1 Trigger and Commutation
中断 */
TIM1_CC_IRQHandler       = 27,      /*!< TIM1 Capture Compare 中断 */
TIM2_IRQHandler          = 28,      /*!< TIM2 全局中断 */
TIM3_IRQHandler          = 29,      /*!< TIM3 全局中断 */
TIM4_IRQHandler          = 30,      /*!< TIM4 全局中断 */
I2C1_EV_IRQHandler      = 31,      /*!< I2C1 事件中断 */
I2C1_ER_IRQHandler      = 32,      /*!< I2C1 错误中断 */
I2C2_EV_IRQHandler      = 33,      /*!< I2C2 事件中断 */
I2C2_ER_IRQHandler      = 34,      /*!< I2C2 错误中断 */
SPI1_IRQHandler          = 35,      /*!< SPI1 全局中断 */
SPI2_IRQHandler          = 36,      /*!< SPI2 全局中断 */
USART1_IRQHandler        = 37,      /*!< USART1 全局中断 */
USART2_IRQHandler        = 38,      /*!< USART2 全局中断 */
USART3_IRQHandler        = 39,      /*!< USART3 全局中断 */
EXTI15_10_IRQHandler    = 40,      /*!< 外部线[15:10]中断 */
RTC_Alarm_IRQHandler     = 41,      /*!< RTC 闹钟通过 EXTI 线中断 */
USBWakeUp_IRQHandler     = 42,      /*!< USB 设备从待机状态通过 EXTI 线
中断唤醒 */
TIM8_BRK_IRQHandler     = 43,      /*!< TIM8 Break 中断 */
TIM8_UP_IRQHandler       = 44,      /*!< TIM8 Update 中断 */
TIM8_TRG_COM_IRQHandler = 45,      /*!< TIM8 Trigger and Commutation
中断 */
TIM8_CC_IRQHandler       = 46,      /*!< TIM8 Capture Compare 中断 */
ADC3_IRQHandler          = 47,      /*!< ADC3 全局中断 */
FSMC_IRQHandler          = 48,      /*!< FSMC 全局中断 */
SDIO_IRQHandler          = 49,      /*!< SDIO 全局中断 */
TIM5_IRQHandler          = 50,      /*!< TIM5 全局中断 */
SPI3_IRQHandler          = 51,      /*!< SPI3 全局中断 */
UART4_IRQHandler         = 52,      /*!< UART4 全局中断 */
UART5_IRQHandler         = 53,      /*!< UART5 全局中断 */
TIM6_IRQHandler          = 54,      /*!< TIM6 全局中断 */
TIM7_IRQHandler          = 55,      /*!< TIM7 全局中断 */
DMA2_Channel1_IRQHandler = 56,      /*!< DMA2 通道 1 全局中断 */
DMA2_Channel2_IRQHandler = 57,      /*!< DMA2 通道 2 全局中断 */
DMA2_Channel3_IRQHandler = 58,      /*!< DMA2 通道 3 全局中断 */
DMA2_Channel4_5_IRQHandler= 59,      /*!< DMA2 通道 4 和通道 5 全局中断 */
} IRQn_Type;
uint32_t PreemptPriority 抢占优先级，根据优先级分组确定。
uint32_t SubPriority) 响应优先级，根据优先级分组确定。

```

## 2. HAL\_NVIC\_EnableIRQ

```
/**  
 * 函数功能：在 NVIC 中启用特定于设备的中断。  
 *  
 * void HAL_NVIC_EnableIRQ(IRQn_Type IRQn)  
 *  
 * 参数：  
 *     IRQn - 外部中断号。可以是 IRQn_Type 枚举的成员  
 *             (对于完整的 STM32 设备 IRQ 通道列表，请参阅相应的 CMSIS 设备文件，如  
 *             stm32f10xxx.h)。  
 * 返回值：无。  
 */
```

## 3. HAL\_NVIC\_DisableIRQ

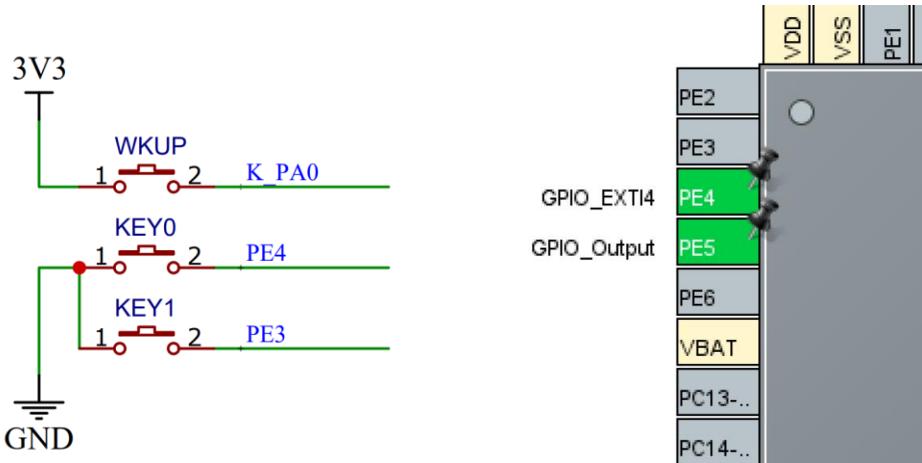
```
/**  
 * 函数功能：在 NVIC 中禁用特定于设备的中断。  
 *  
 * void HAL_NVIC_DisableIRQ(IRQn_Type IRQn)  
 *  
 * 参数：  
 *     IRQn - 外部中断号。可以是 IRQn_Type 枚举的成员  
 *             (对于完整的 STM32 设备 IRQ 通道列表，请参阅相应的 CMSIS 设备文件，如  
 *             stm32f10xxx.h)。  
 * 返回值：无。  
 */
```

## 1.11 按键中断

### 1.11.1 CubeMX 基础配置

时钟配置与工程管理配置与 1.7 相同

根据电路图，额外配置 PE5 为 Output（灯），PE4 为 EXTI（按键）。



## 1.11.2 中断模式配置

### External Interrupt Mode with Rising Edge Trigger Detection:

外部中断模式，上升沿触发检测：这种模式配置 GPIO 引脚在检测到上升沿时生成中断，即当引脚上的电压水平从低（0）变为高（1）。

### External Interrupt Mode with Falling Edge Trigger Detection:

外部中断模式，下降沿触发检测：在这种模式下，GPIO 引脚在检测到下降沿时生成中断，即电压水平从高变低。

### External Interrupt Mode with Rising/Falling Edge Trigger Detection:

外部中断模式，上升/下降沿触发检测：这种模式使 GPIO 引脚能够在上升沿和下降沿上都生成中断。

### External Event Mode with Rising Edge Trigger Detection:

外部事件模式，上升沿触发检测：类似于中断模式，这种模式检测上升沿，但不是生成中断，而是触发一个事件。

### External Event Mode with Falling Edge Trigger Detection:

外部事件模式，下降沿触发检测：这种设置使 GPIO 引脚在检测到下降沿时触发事件。

### External Event Mode with Rising/Falling Edge Trigger Detection:

外部事件模式，上升/下降沿触发检测：这种模式在上升沿和下降沿都会触发事件。

根据电路图：

选择 External Interrupt Mode with Falling Edge Trigger Detection

在 NVIC 选择中，开启 EXTI line4 中断，并调整优先级。

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
Non maskable interrupt	<input checked="" type="checkbox"/>	0	0
Hard fault interrupt	<input checked="" type="checkbox"/>	0	0
Memory management fault	<input checked="" type="checkbox"/>	0	0
Prefetch fault, memory acce	<input checked="" type="checkbox"/>	0	0
Undefined instruction or illegal state	<input checked="" type="checkbox"/>	0	0
System service call via SWI instruction	<input checked="" type="checkbox"/>	0	0
Debug monitor	<input checked="" type="checkbox"/>	0	0
Pendable request for system service	<input checked="" type="checkbox"/>	0	0
Time base: System tick timer	<input checked="" type="checkbox"/>	15	0
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0	0
Flash global interrupt	<input type="checkbox"/>	0	0
RCC global interrupt	<input type="checkbox"/>	0	0
EXTI line4 interrupt	<input checked="" type="checkbox"/>	0	0

### 1.11.3 工程代码分析

Gpio.c 文件中

```
void MX_GPIO_Init(void)
{
    // 定义一个 GPIO 初始化结构体变量
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    // 使能 GPIOE 端口的时钟
    __HAL_RCC_GPIOE_CLK_ENABLE();

    // 将 GPIOE 的第 5 引脚设置为低电平
    HAL_GPIO_WritePin(GPIOE, GPIO_PIN_5, GPIO_PIN_RESET);
```

```

// 配置 GPIOE 的第 4 引脚为下降沿触发的中断模式, 不使用内部上拉或下拉
GPIO_InitStruct.Pin = GPIO_PIN_4;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

// 配置 GPIOE 的第 5 引脚为推挽输出模式, 不使用内部上拉或下拉, 输出速度为低速
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

// 为 GPIOE 的第 4 引脚配置外部中断, 设置中断优先级
HAL_NVIC_SetPriority(EXTI4_IRQn, 0, 0);
// 使能 GPIOE 的第 4 引脚的中断
HAL_NVIC_EnableIRQ(EXTI4_IRQn);
}

```

与 1.7 中 gpio 相比, 增加了外部中断的设置

在 stm32f1xx\_it.c 文件中, 增加了 EXTI4 中断服务函数,

```

200
201 /**
202  * @brief This function handles EXTI line4 interrupt.
203 */
204 void EXTI4_IRQHandler(void)
205 {
206     /* USER CODE BEGIN EXTI4_IRQn 0 */
207
208     /* USER CODE END EXTI4_IRQn 0 */
209     HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_4);
210     /* USER CODE BEGIN EXTI4_IRQn 1 */
211
212     /* USER CODE END EXTI4_IRQn 1 */
213 }
214
215 /* USER CODE BEGIN 1 */
216
217 /* USER CODE END 1 */
218

```

其中 `HAL_GPIO_EXTI_IRQHandler()` 函数作用如下

```

/**
 * 函数功能: 处理 GPIO 外部中断请求。
 *
 * void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
 *
 * 参数:

```

```

*      GPIO_Pin - 指定连接到 EXTI 线的引脚。
*
* 返回值: 无。
*/
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
    // 检测到 EXTI 线路的中断
    if (__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != 0x00u)
    {
        // 清除该 GPIO 引脚上的 EXTI 中断标志
        __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
        // 调用回调函数来处理这个中断
        HAL_GPIO_EXTI_Callback(GPIO_Pin);
    }
}

```

其中 `HAL_GPIO_EXTI_Callback(GPIO_Pin)` 相关解释如下；

```

/**
 * 函数功能: EXTI 线路检测回调。
*
* __weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
*
* 参数:
*      GPIO_Pin - 指定连接到 EXTI 线的引脚。
*
* 返回值: 无。
*
* 注意: 此函数在默认状态下不应被修改。当需要使用回调时,
*       用户应在自己的文件中实现 HAL_GPIO_EXTI_Callback 函数。
*/
__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    /* 防止未使用参数引起的编译警告 */
    UNUSED(GPIO_Pin);
    /* 注意: 当需要回调时, 应在用户文件中实现此函数 */
}

```

该函数是一个弱定义函数，意味着用户可以在自己的代码中重定义这个函数以实现特定的功能。默认情况下，这个函数仅用于避免编译器因未使用的参数 `GPIO_Pin` 而发出警告。当需要处理特定 GPIO 引脚的外部中断时，可以在自己的代码中提供这个函数的具体实现，以响应和处理外部中断事件。

中断总体实现逻辑为：

当一个外部中断事件发生在与 EXTI Line 4 相关联的 GPIO 引脚上时，中断处理流程被触发。这一流程首先进入 EXTI4\_IRQHandler 函数，这是为 EXTI Line 4 专门配置的中断服务例程（ISR）。在 EXTI4\_IRQHandler 函数内部，调用 HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_4) 函数，传递与中断相关的 GPIO 引脚（在这种情况下为 GPIO\_PIN\_4）作为参数。

HAL\_GPIO\_EXTI\_IRQHandler 函数的主要职责是管理与该 GPIO 引脚相关的外部中断处理。它首先检查指定的 GPIO 引脚是否真的有未处理的中断事件。如果检测到中断，该函数将执行两个关键步骤：首先，清除该 GPIO 引脚上的中断标志位，这是防止因同一事件重复进入中断处理的重要步骤。其次，调用 HAL\_GPIO\_EXTI\_Callback(GPIO\_Pin) 函数。这个回调函数是用户定义的，允许在应用程序级别自定义对特定 GPIO 引脚上的中断事件的响应。

#### 1.11.4 回调程序设计

在 main.c 中添加

```
/* USER CODE BEGIN 0 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5);
}
/* USER CODE END 0 */
```

实验效果：按下按键 KEY0，灯切换状态

### 1.12 定时器

#### 1.12.1 定时器介绍

共有 8 个定时器：高级定时器（TIM1 和 TIM8）、通用定时器（TIM2——TIM5）、基本定时器（TIM6 和 TIM7）

每个定时器都是一个可编程预分频器驱动的 16 位自动装载计数器构成。每个定时器都是完全独立的，没有互相共享任何资源，它们可以一起同步操作。除了可以进行基本定时外，还可以输出 PWM，输入捕获，互补输出等。

高级定时器	定时器	计数器分辨率	计数器类型	预分频系数	产生DMA	捕获/比较通道	互补输出
	TIM1	16位	向上/向下	1~65535	可以	4 有	
通用定时器	TIM8	16位	向上/向下	1~65535	可以	4 有	
	TIM2	16位	向上/向下	1~65535	可以	4 没有	
	TIM3	16位	向上/向下	1~65535	可以	4 没有	
	TIM4	16位	向上/向下	1~65535	可以	4 没有	
基本定时器	TIM5	16位	向上/向下	1~65535	可以	4 没有	
	TIM6	16位	向上	1~65535	可以	0 没有	
	TIM7	16位	向上	1~65535	可以	0 没有	

### 1.12.2 定时器计时公式

定时时间计算公式

$$T_{out} = \frac{\text{TIM_Period} * (\text{TIM_Prescaler} + 1)}{72M} \text{ (s)}$$

TIM\_Period 自动转载的数

TIM\_Prescaler 预分频数

### 1.12.3 定时器结构体

定时器结构体 `TIM_HandleTypeDef` 包含了定时器基本信息

```
typedef struct
{
    TIM_TypeDef *Instance;                                // 定时器的寄存器地址
    TIM_Base_InitTypeDef Init;                            // 定时器基本初始化所需的参数
    HAL_TIM_ActiveChannel Channel;                      // 当前激活的定时器通道
    DMA_HandleTypeDef *hdma[7];                          // DMA 处理器的数组，用于定时
    器的 DMA 操作
    HAL_LockTypeDef Lock;                               // 用于锁定的对象，防止资源冲
    突
    __IO HAL_TIM_StateTypeDef State;                   // 定时器的当前操作状态
    __IO HAL_TIM_ChannelStateTypeDef ChannelState[4]; // 各个定时器通道的当前
    操作状态
    __IO HAL_TIM_ChannelStateTypeDef ChannelNState[4]; // 定时器互补通道的当前
    操作状态
    __IO HAL_TIM_DMABurstStateTypeDef DMABurstState;   // DMA 突发操作的当前状
    态

    // 如果定义了 USE_HAL_TIM_REGISTER_CALLBACKS，则包含以下回调函数
#if (USE_HAL_TIM_REGISTER_CALLBACKS == 1)
    typedef struct __TIM_HandleTypeDef

```

```

#else
typedef struct
#endif /* USE_HAL_TIM_REGISTER_CALLBACKS */
{
    TIM_TypeDef *Instance; // 定时器寄存器地址
    TIM_Base_InitTypeDef Init; // 定时器基础初始化参数
    HAL_TIM_ActiveChannel Channel; // 激活的通道
    DMA_HandleTypeDef *hdma[7]; // DMA 处理句柄数组
    HAL_LockTypeDef Lock; // 锁定对象
    __IO HAL_TIM_StateTypeDef State; // 定时器操作状态
    __IO HAL_TIM_ChannelStateTypeDef ChannelState[4]; // 定时器通道操作状态
    __IO HAL_TIM_ChannelStateTypeDef ChannelNState[4]; // 定时器补充通道操作状态
    __IO HAL_TIM_DMABurstStateTypeDef DMABurstState; // DMA 突发操作状态

#if (USE_HAL_TIM_REGISTER_CALLBACKS == 1)
    void (*Base_MspInitCallback)(struct __TIM_HandleTypeDef
*htim); // 基础模式 Msp 初始化回调
    void (*Base_MspDeInitCallback)(struct __TIM_HandleTypeDef
*htim); // 基础模式 Msp 去初始化回调
    void (*IC_MspInitCallback)(struct __TIM_HandleTypeDef
*htim); // 输入捕获 Msp 初始化回调
    void (*IC_MspDeInitCallback)(struct __TIM_HandleTypeDef
*htim); // 输入捕获 Msp 去初始化回调
    void (*OC_MspInitCallback)(struct __TIM_HandleTypeDef
*htim); // 输出比较 Msp 初始化回调
    void (*OC_MspDeInitCallback)(struct __TIM_HandleTypeDef
*htim); // 输出比较 Msp 去初始化回调
    void (*Pwm_MspInitCallback)(struct __TIM_HandleTypeDef
*htim); // PWM Msp 初始化回调
    void (*Pwm_MspDeInitCallback)(struct __TIM_HandleTypeDef
*htim); // PWM Msp 去初始化回调
    void (*OnePulse_MspInitCallback)(struct __TIM_HandleTypeDef
*htim); // 单脉冲 Msp 初始化回调
    void (*OnePulse_MspDeInitCallback)(struct __TIM_HandleTypeDef
*htim); // 单脉冲 Msp 去初始化回调
    void (*Encoder_MspInitCallback)(struct __TIM_HandleTypeDef
*htim); // 编码器 Msp 初始化回调
    void (*Encoder_MspDeInitCallback)(struct __TIM_HandleTypeDef
*htim); // 编码器 Msp 去初始化回调
    void (*HallSensor_MspInitCallback)(struct __TIM_HandleTypeDef
*htim); // 霍尔传感器 Msp 初始化回调
    void (*HallSensor_MspDeInitCallback)(struct __TIM_HandleTypeDef
*htim); // 霍尔传感器 Msp 去初始化回调

```

```

    void (* PeriodElapsedCallback)(struct __TIM_HandleTypeDef
*hitm); // 周期结束回调
    void (* PeriodElapsedHalfCpltCallback)(struct __TIM_HandleTypeDef
*hitm); // 周期结束半完成回调
    void (* TriggerCallback)(struct __TIM_HandleTypeDef
*hitm); // 触发回调
    void (* TriggerHalfCpltCallback)(struct __TIM_HandleTypeDef
*hitm); // 触发半完成回调
    void (* IC_CaptureCallback)(struct __TIM_HandleTypeDef
*hitm); // 输入捕获回调
    void (* IC_CaptureHalfCpltCallback)(struct __TIM_HandleTypeDef
*hitm); // 输入捕获半完成回调
    void (* OC_DelayElapsedCallback)(struct __TIM_HandleTypeDef
*hitm); // 输出比较延迟结束回调
    void (* PWM_PulseFinishedCallback)(struct __TIM_HandleTypeDef
*hitm); // PWM 脉冲完成回调
    void (* PWM_PulseFinishedHalfCpltCallback)(struct __TIM_HandleTypeDef
*hitm); // PWM 脉冲半完成回调
    void (* ErrorCallback)(struct __TIM_HandleTypeDef
*hitm); // 错误回调
    void (* CommutationCallback)(struct __TIM_HandleTypeDef
*hitm); // 换相回调
    void (* CommutationHalfCpltCallback)(struct __TIM_HandleTypeDef
*hitm); // 换相半完成回调
    void (* BreakCallback)(struct __TIM_HandleTypeDef
*hitm); // 断开(Break)事件时的回调函数

#endif
} TIM_HandleTypeDef;

```

其中结构体 `TIM_HandleTypeDef` 为寄存器相关

```

typedef struct
{
    __IO uint32_t CR1; // TIM 控制寄存器 1, 地址偏移: 0x00
    __IO uint32_t CR2; // TIM 控制寄存器 2, 地址偏移: 0x04
    __IO uint32_t SMCR; // TIM 从模式控制寄存器, 地址偏移: 0x08
    __IO uint32_t DIER; // TIM DMA/中断使能寄存器, 地址偏移: 0x0C
    __IO uint32_t SR; // TIM 状态寄存器, 地址偏移: 0x10
    __IO uint32_t EGR; // TIM 事件生成寄存器, 地址偏移: 0x14
    __IO uint32_t CCMR1; // TIM 捕获/比较模式寄存器 1, 地址偏移: 0x18
    __IO uint32_t CCMR2; // TIM 捕获/比较模式寄存器 2, 地址偏移: 0x1C
    __IO uint32_t CCER; // TIM 捕获/比较使能寄存器, 地址偏移: 0x20
    __IO uint32_t CNT; // TIM 计数器寄存器, 地址偏移: 0x24
    __IO uint32_t PSC; // TIM 预分频寄存器, 地址偏移: 0x28
    __IO uint32_t ARR; // TIM 自动重载寄存器, 地址偏移: 0x2C
}

```

```

    __IO uint32_t RCR;      // TIM 重复计数器寄存器，地址偏移: 0x30
    __IO uint32_t CCR1;     // TIM 捕获/比较寄存器 1, 地址偏移: 0x34
    __IO uint32_t CCR2;     // TIM 捕获/比较寄存器 2, 地址偏移: 0x38
    __IO uint32_t CCR3;     // TIM 捕获/比较寄存器 3, 地址偏移: 0x3C
    __IO uint32_t CCR4;     // TIM 捕获/比较寄存器 4, 地址偏移: 0x40
    __IO uint32_t BDTR;     // TIM 断开和死区时间寄存器, 地址偏移: 0x44
    __IO uint32_t DCR;      // TIM DMA 控制寄存器, 地址偏移: 0x48
    __IO uint32_t DMAR;     // TIM DMA 全传输地址寄存器, 地址偏移: 0x4C
    __IO uint32_t OR;       // TIM 选项寄存器, 地址偏移: 0x50
} TIM_TypeDef;

```

其中结构体 `TIM_Base_InitTypeDef` 为基础配置，与 1.13.2 对应。

```

typedef struct
{
    uint32_t Prescaler;          // 预分频器值, 用于分频 TIM 时钟。范围: Min_Data
= 0x0000, Max_Data = 0xFFFF
    uint32_t CounterMode;        // 计数器模式
    uint32_t Period;            // 装载到下一个更新事件的有效自动重载寄存器的周
期值。范围: Min_Data = 0x0000, Max_Data = 0xFFFF
    uint32_t ClockDivision;      // 时钟分频。
    uint32_t RepetitionCounter; // 重复计数器值。
    uint32_t AutoReloadPreload; // 自动重载预装载。
} TIM_Base_InitTypeDef;

```

## 1.13 定时器时间基准模式（Time Base）

### 1.13.1 时间基准模式概述

时间基准模式是微控制器中定时器的一种基本工作方式，它为各种定时和计数操作提供了基础。在 STM32 微控制器系列中，时间基准模式通过定时器（TIM）实现，其核心功能是生成准确的时间延迟或计算经过的时间。

时间基准模式的工作原理是基于内部或外部时钟源。定时器内部包含一个计数器，该计数器会根据时钟源的脉冲信号递增或递减。定时器的预分频器（Prescaler）允许调整时钟脉冲的频率，从而控制计数器的计数速度。此外，定时器的自动重装载寄存器（ARR）定义了计数器的上限值，一旦计数器的值达到这个上限，它可以自动重置为零，并可选择性地触发中断或事件。

### 1.13.2 时间基准模式相关配置参数

#### 基本参数

Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits...)	65535
Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - 8 bits value)	0
auto-reload preload	Disable
Trigger Output (TRGO) Parameters	
Master/Slave Mode (MSM bit)	Disable (Trigger input effect not delayed)
Trigger Event Selection	Reset (UG bit from TIMx_EGR)

### 预分频器（Prescaler, PSC - 16 bits value）

预分频器用于分频，它可以将定时器的输入时钟频率除以 1 至 65536 之间的任意因子。这是通过一个 16 位的计数器实现的，控制通过一个 16 位的寄存器（TIMx\_PSC 寄存器）进行。预分频器的值可以动态更改，新的预分频比例将在下一个更新事件时生效。

### 计数器模式（Counter Mode）：

定时器可以在不同的计数模式下工作，例如向上计数或向下计数。在“向上”模式中，每个时钟周期计数器的值增加 1。

### 计数周期（Counter Period）：

自动重装载寄存器（ARR）定义了计数器的最大值。当计数器达到这个值时，它将重新开始从 0 计数，并可能触发中断或其他事件。

### 内部时钟分频（Internal Clock Division, CKD）：

这个设置用于进一步分频定时器的时钟源，但通常不是主要关注的焦点。

### 重复计数器（Repetition Counter, RCR - 8 bits value）：

重复计数器用于确定在触发更新事件之前定时器需要完成多少计数周期。

### 自动重装载预加载（auto-reload preload）：

自动重装载预加载功能允许预先加载 ARR 寄存器的值。

## 中断列表

Parameter Settings	User Constants	NVIC Settings	DMA Settings
NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM1 break interrupt	<input type="checkbox"/>	0	0
TIM1 update interrupt	<input checked="" type="checkbox"/>	0	0
TIM1 trigger and commutation interrupts	<input type="checkbox"/>	0	0
TIM1 capture compare interrupt	<input type="checkbox"/>	0	0

### TIM1 Break Interrupt (TIM1 断开中断)

这种中断通常用于 PWM 通道。在出现故障或“trip”情况时，可以通过 break 输入电路禁用 PWM 通道。

### TIM1 Update Interrupt (TIM1 更新中断)

Update 中断通常用于处理计数器溢出/下溢或计数器初始化（通过软件或内部/外部触发）的情况。

### TIM1 Trigger and Commutation Interrupts (TIM1 触发和换相中断)

触发中断与计数器的启动、停止、初始化或通过内部/外部触发进行计数有关。这些中断通常用于更复杂的定时器操作，其中定时器的行为依赖于外部或内部事件的发生。

### TIM1 Capture Compare Interrupt (TIM1 捕获比较中断)

捕获比较中断用于处理输入捕获和输出比较事件。在输入捕获模式下，当检测到外部信号的变化时（如边缘触发），定时器的当前计数值会被捕获。在输出比较模式下，定时器值与预设的比较值相匹配时，可以执行特定的动作，如改变输出引脚的状态。

## 1.13.3 时间基准模式相关 API

### 1. HAL\_TIM\_Base\_Init

```
/**  
 * 函数功能：根据 TIM_HandleTypeDef 中指定的参数初始化 TIM Time base 单元，并创建关联的句柄。  
 *  
 * HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *     htim - 指向 TIM_HandleTypeDef 结构的指针，包含 TIM 模块的配置信息。  
 * 返回值：HAL 状态。  
 */
```

### 2. HAL\_TIM\_Base\_DeInit

```
/**  
 * 函数功能：反初始化 TIM Base 外设。  
 *  
 * HAL_StatusTypeDef HAL_TIM_Base_DeInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *     htim - TIM Base 句柄。  
 * 返回值：HAL 状态。  
 */
```

```
 */
```

### 3. HAL\_TIM\_Base\_MspInit

```
/**  
 * 函数功能: 初始化 TIM Base MSP。  
 *  
 * void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数:  
 *     htim - TIM Base 句柄。  
 * 返回值: 无。  
 */
```

### 4. HAL\_TIM\_Base\_MspDeInit

```
/**  
 * 函数功能: 反初始化 TIM Base MSP。  
 *  
 * void HAL_TIM_Base_MspDeInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数:  
 *     htim - TIM Base 句柄。  
 * 返回值: 无。  
 */
```

### 5. HAL\_TIM\_Base\_Start

```
/**  
 * 函数功能: 启动 TIM Base。  
 *  
 * HAL_StatusTypeDef HAL_TIM_Base_Start(TIM_HandleTypeDef *htim)  
 *  
 * 参数:  
 *     htim - TIM Base 句柄。  
 * 返回值: HAL 状态。  
 */
```

### 6. HAL\_TIM\_Base\_Stop

```
/**  
 * 函数功能: 停止 TIM Base。  
 *  
 * HAL_StatusTypeDef HAL_TIM_Base_Stop(TIM_HandleTypeDef *htim)  
 *  
 * 参数:  
 *     htim - TIM Base 句柄。  
 * 返回值: HAL 状态。
```

```
 */
```

## 7. HAL\_TIM\_Base\_Start\_IT

```
/**  
 * 函数功能：以中断模式启动 TIM Base。  
 *  
 * HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *     htim - TIM Base 句柄。  
 * 返回值：HAL 状态。  
 */
```

## 8. HAL\_TIM\_Base\_Stop\_IT

```
/**  
 * 函数功能：以中断模式停止 TIM Base。  
 *  
 * HAL_StatusTypeDef HAL_TIM_Base_Stop_IT(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *     htim - TIM Base 句柄。  
 * 返回值：HAL 状态。  
 */
```

## 9. HAL\_TIM\_Base\_Start\_DMA

```
/**  
 * 函数功能：以 DMA 模式启动 TIM Base。  
 *  
 * HAL_StatusTypeDef HAL_TIM_Base_Start_DMA(TIM_HandleTypeDef *htim,  
 uint32_t *pData, uint16_t Length)  
 *  
 * 参数：  
 *     htim - TIM Base 句柄。  
 *     pData - 源缓冲区地址。  
 *     Length - 从内存到外设传输的数据长度。  
 * 返回值：HAL 状态。  
 */
```

## 10. HAL\_TIM\_Base\_Stop\_DMA

```
/**  
 * 函数功能：以 DMA 模式停止 TIM Base。  
 *  
 * HAL_StatusTypeDef HAL_TIM_Base_Stop_DMA(TIM_HandleTypeDef *htim)  
 */
```

```
* 参数:  
*     htim - TIM Base 句柄。  
* 返回值: HAL 状态。  
*/
```

## 11. \_\_HAL\_TIM\_GetCounter

```
/**  
 * 宏功能: 获取 TIM 计数器的当前值。  
 *  
 * #define __HAL_TIM_GetCounter(__HANDLE__) ((__HANDLE__)->Instance->CNT)  
 *  
 * 参数:  
 *     __HANDLE__ - 指向 TIM 句柄的指针。  
 * 返回值: 当前计数值。  
 */
```

## 12. \_\_HAL\_TIM\_SetCounter

```
/**  
 * 宏功能: 设置 TIM 计数器的计数值。  
 *  
 * #define __HAL_TIM_SetCounter(__HANDLE__, __COUNTER__) ((__HANDLE__)-  
>Instance->CNT = (__COUNTER__))  
 *  
 * 参数:  
 *     __HANDLE__ - 指向 TIM 句柄的指针。  
 *     __COUNTER__ - 要设置的计数值。  
 * 返回值: 无。  
 */
```

### 1.13.4 hal 工程函数详解

1.CubeMX 配置，生成基本代码

基本参数配置。预分频 7199，计数周期 9999，定时 1s 钟

## 中断选择

<input checked="" type="checkbox"/> Parameter Settings	<input checked="" type="checkbox"/> User Constants	<input checked="" type="checkbox"/> NVIC Settings	<input checked="" type="checkbox"/> DMA Settings
NVIC Interrupt Table			
TIM1 break interrupt	<input type="checkbox"/>	0	0
TIM1 update interrupt	<input checked="" type="checkbox"/>	0	0
TIM1 trigger and commutation interrupts	<input type="checkbox"/>	0	0
TIM1 capture compare interrupt	<input type="checkbox"/>	0	0

## 同时初始化 PE5



## 2.CubeMX 配置，生成基本代码

在 tim.c 中的定时器配置函数

```
void MX_TIM1_Init(void)
{
    /* USER CODE BEGIN TIM1_Init_0 */
    // 用户代码区域，用于初始化之前的自定义代码。
    /* USER CODE END TIM1_Init_0 */

    // 定义时钟源配置结构体，并初始化为零。
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    // 定义主控制器配置结构体，并初始化为零。
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    /* USER CODE BEGIN TIM1_Init_1 */
    // 用户代码区域，用于初始化之前的自定义代码。
    /* USER CODE END TIM1_Init_1 */
}
```

```
// 设置 TIM1 为定时器实例。
htim1.Instance = TIM1;
// 设置预分频器值为 7199。
htim1.Init.Prescaler = 7199;
// 设置计数模式为向上计数。
htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
// 设置定时器周期为 9999。
htim1.Init.Period = 9999;
// 设置时钟分频因子为无分频。
htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
// 设置重复计数器值为 0。
htim1.Init.RepetitionCounter = 0;
// 设置自动重载预装载为禁用。
htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
// 初始化 TIM Base 并检查返回状态。
if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
{
    // 错误处理程序。
    Error_Handler();
}
// 设置时钟源为内部时钟。
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
// 配置时钟源并检查返回状态。
if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
{
    // 错误处理程序。
    Error_Handler();
}
// 设置主输出触发为重置。
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
// 设置主从模式为禁用。
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
// 配置主控制器同步并检查返回状态。
if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
{
    // 错误处理程序。
    Error_Handler();
}
/* USER CODE BEGIN TIM1_Init_2 */
// 用户代码区域，用于初始化之后的自定义代码。
/* USER CODE END TIM1_Init_2 */
}
```

初始化 TIM1 定时器的底层硬件特性（如时钟和中断）函数。该函数在 HAL\_TIM\_Base\_Init() 函数中被调用。

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* tim_baseHandle)
{
    // 检查传入的定时器句柄是否是 TIM1 的实例。
    if(tim_baseHandle->Instance==TIM1)
    {
        /* USER CODE BEGIN TIM1_MspInit_0 */
        // 用户自定义代码区域，初始化 TIM1 MSP 前。
        /* USER CODE END TIM1_MspInit_0 */

        // 启用 TIM1 时钟。
        __HAL_RCC_TIM1_CLK_ENABLE();

        // 初始化 TIM1 中断。
        // 设置 TIM1 更新 (UP) 中断的优先级。
        HAL_NVIC_SetPriority(TIM1_UP_IRQn, 0, 0);
        // 启用 TIM1 更新 (UP) 中断。
        HAL_NVIC_EnableIRQ(TIM1_UP_IRQn);

        /* USER CODE BEGIN TIM1_MspInit_1 */
        // 用户自定义代码区域，初始化 TIM1 MSP 后。
        /* USER CODE END TIM1_MspInit_1 */
    }
}
```

在 stm32103f1xx\_it.c 文件中

TIM1 的更新中断服务函数

```
void TIM1_UP_IRQHandler(void)
{
    /* USER CODE BEGIN TIM1_UP_IRQn_0 */

    /* USER CODE END TIM1_UP_IRQn_0 */
    HAL_TIM_IRQHandler(&htim1);
    /* USER CODE BEGIN TIM1_UP_IRQn_1 */

    /* USER CODE END TIM1_UP_IRQn_1 */
}
```

HAL\_TIM\_IRQHandler() 是一个中断处理函数，用于处理与 TIM 相关的所有中断事件。

该函数首先检查哪种类型的中断事件发生（如捕获比较、更新、断开、触发等）。

之后根据中断类型，该函数清除中断标志，并根据事件类型调用相应的回调函数，以执行用户定义的操作。在 HAL\_TIM\_IRQHandler() 中相关代码片段如下：

```

/* TIM Update event */
// 处理 TIM 更新事件。
if (__HAL_TIM_GET_FLAG(htim, TIM_FLAG_UPDATE) != RESET)
{
    if (__HAL_TIM_GET_IT_SOURCE(htim, TIM_IT_UPDATE) != RESET)
    {
        // 清除中断标志。
        __HAL_TIM_CLEAR_IT(htim, TIM_IT_UPDATE);
        // 调用周期结束的回调函数。
#if (USE_HAL_TIM_REGISTER_CALLBACKS == 1)
        htim->PeriodElapsedCallback(htim);
#else
        HAL_TIM_PeriodElapsedCallback(htim);
#endif /* USE_HAL_TIM_REGISTER_CALLBACKS */
    }
}

```

与 `HAL_GPIO_EXTI_Callback` 函数类似，`HAL_TIM_PeriodElapsedCallback()` 也是一个弱回调函数，用于处理 TIM 的周期结束事件。默认情况下，它不执行任何操作，但可以被用户在自己的文件中重写以执行特定操作。这种设计允许用户根据需要定制化处理 TIM 事件，而不需要修改 HAL 库本身的代码。

```

/**
 * 函数功能：TIM 周期结束回调。
 *
 * __weak void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
 *
 * 参数：
 *     htim - 指向 TIM_HandleTypeDef 结构的指针，包含 TIM 模块的配置信息。
 *
 * 返回值：无。
 *
 * 注意：此函数在默认状态下不应被修改。当需要使用回调时，
 *       用户应在自己的文件中实现 HAL_TIM_PeriodElapsedCallback 函数。
 */
__weak void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    /* 防止未使用参数引起的编译警告 */
    UNUSED(htim);
    /* 注意：当需要回调时，应在用户文件中实现此函数 */
}

```

### 1.13.5 定时器使灯光闪烁

1. 在 main.c 中启动定时器

```
/* USER CODE BEGIN 2 */  
HAL_TIM_Base_Start_IT(&htim1);  
/* USER CODE END 2 */
```

2.在 main.c 实现 HAL\_TIM\_PeriodElapsedCallback()

```
/* USER CODE BEGIN 4 */  
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)  
{  
    // 确保此回调是为了正确的 TIM 实例  
    if (htim->Instance == TIM1) //  
    {  
        // 切换 E5 引脚的状态  
        HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5); // 假设 E5 引脚已经配置为输出模  
式  
    }  
}  
  
/* USER CODE END 4 */
```

3.实验效果，LED 灯每秒闪烁 1 次

## 1.14 定时器输出比较模式（Output Compare）

### 1.14.1 输出比较模式概述

在微控制器编程领域，定时器的输出比较模主要功能是当定时器计数达到由自动重载寄存器（ARR）设定的最大值时，触发一系列精确控制动作。包括对应输出引脚的可编程值设定，中断状态寄存器中标志的设置，以及在适当的中断屏蔽启用的情况下产生中断信号。此外，该模式还能在对应的使能位被激活时发出直接存储器访问（DMA）请求。该模式在脉宽调制（PWM）输出的生成上起重要作用。

PWM 是通过定时器生成的，用于控制开关时间与关断时间之间的比率，通常称为占空比。占空比可以在 0 到 1 之间变化，通常以百分比表示。PWM 的脉宽在高频（kHz）下变化，可以被视为连续的，用于控制电机的转速、调光 LED、驱动编码器、电源转换等。

### 1.14.2 输出比较模式模式相关参数

#### ▼ PWM Generation Channel 2

Mode	PWM mode 1
Pulse (16 bits value)	0
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High

**PWM Mode (PWM 模式):**

STM32 提供了两种 PWM 模式，即 PWM 模式 1 和模式 2。

PWM 模式 1: 当 TIMx\_CNT (定时器的当前计数值) 小于 TIMx\_CCR1 (捕获/比较寄存器的值) 时，通道 1 处于激活状态，否则处于非激活状态。

PWM 模式 2: 当 TIMx\_CNT 小于 TIMx\_CCR1 时，通道 1 处于非激活状态，否则激活。

这两种模式的差别主要在于输出的高低电平状态。例如，在 PWM 模式 1 中，如果 TIM 计数器的值小于 CCR 值，则输出为高电平（假设 CH Polarity 设置为高电平有效）；而在 PWM 模式 2 中，如果 TIM 计数器的值小于 CCR 值，则输出为低电平（同样假设 CH Polarity 设置为高电平有效）。通过配置 TIMx\_CCMR1.OC1M 位域来选择这两种模式中的任一种

### **脉冲 (Pulse, 16 位值):**

在 PWM 模式中，捕获比较寄存器 (CCR) 中设置的值决定了脉冲的宽度。定时器的计数和比较值之间的比较结果影响 PWM 波形的输出。例如，在 CCR1 寄存器中设置的值会影响通道 1 的输出。

### **输出比较预加载 (Output Compare Preload):**

启用此位时，写入比较寄存器（如 CCR1）的值不会立即设置，而是保留在一个中间的‘预加载’寄存器中，并在更新事件（如计数器 (CNT) 溢出时）时复制到‘活动比较寄存器’中。这可以防止在计数过程中更改 CCR 时漏掉脉冲（在 PWM 模式下，比较输出只在匹配时刻改变）。

### **快速模式 (Fast Mode):**

在单脉冲模式中，TIx 输入的边缘检测设置了 CEN 位，从而启用计数器。然后计数器和比较值之间的比较使输出切换。但是这些操作需要几个时钟周期，限制了我们可以获得的最小延迟。为了以最小延迟输出波形，用户可以在 TIMx\_CCMRx 寄存器中设置 OCxFE 位。然后 OCxRef (和 OCx) 将响应刺激而被强制输出，而不考虑比较。其新水平与比较匹配发生时的水平相同。OCxFE 仅在通道配置为 PWM1 或 PWM2 模式时起作用。

### **CH Polarity (通道极性):**

在 STM32 的 PWM 模式中，可以通过设置 TIMx\_CCER.CCxP 位来切换 PWM 的极性。这意味着 PWM 信号可以是高电平有效或低电平有效，根据具体的应用需求进行调整。例如，在某些应用中，当 TIM 计数器的值低于比较值 (CCR) 时，输出可能需要保持在高电平，而在其他应用中可能需要保持在低电平。通过改变 CH Polarity 的设置，可以实现这种输出行为的切换。

## **1.14.3 输出比较模式模式相关 API**

### **1. HAL\_TIM\_PWM\_Init**

```
/**  
 * 函数功能：根据 TIM_HandleTypeDef 中指定的参数初始化 TIM PWM。  
 *  
 * HAL_StatusTypeDef HAL_TIM_PWM_Init(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *   htim - 指向 TIM_HandleTypeDef 结构的指针，包含 TIM PWM 模块的配置信息。  
 * 返回值：HAL 状态。  
 */
```

## 2. HAL\_TIM\_PWM\_DeInit

```
/**  
 * 函数功能：反初始化 TIM PWM 外设。  
 *  
 * HAL_StatusTypeDef HAL_TIM_PWM_DeInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *   htim - 指向 TIM_HandleTypeDef 结构的指针，包含 TIM PWM 模块的配置信息。  
 * 返回值：HAL 状态。  
 */
```

## 3. HAL\_TIM\_PWM\_MspInit

```
/**  
 * 函数功能：初始化 TIM PWM MSP。  
 *  
 * void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *   htim - 指向 TIM_HandleTypeDef 结构的指针。  
 * 返回值：无。  
 */
```

## 4. HAL\_TIM\_PWM\_MspDeInit

```
/**  
 * 函数功能：反初始化 TIM PWM MSP。  
 *  
 * void HAL_TIM_PWM_MspDeInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *   htim - 指向 TIM_HandleTypeDef 结构的指针。  
 * 返回值：无。  
 */
```

## 5. HAL\_TIM\_PWM\_Start

```
/**  
 * 函数功能：启动 PWM 信号生成。  
 *  
 * HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t  
 Channel)  
 *  
 * 参数：  
 *   htim - 指向 TIM_HandleTypeDef 结构的指针。  
 *   Channel - 要启用的 TIM 通道。  
 * 返回值：HAL 状态。  
 */
```

## 6. HAL\_TIM\_PWM\_Stop

```
/**  
 * 函数功能：停止 PWM 信号生成。  
 *  
 * HAL_StatusTypeDef HAL_TIM_PWM_Stop(TIM_HandleTypeDef *htim, uint32_t  
 Channel)  
 *  
 * 参数：  
 *   htim - 指向 TIM_HandleTypeDef 结构的指针。  
 *   Channel - 要停用的 TIM 通道。  
 * 返回值：HAL 状态。  
 */
```

## 7. HAL\_TIM\_PWM\_Start\_IT

```
/**  
 * 函数功能：以中断模式启动 PWM 信号生成。  
 *  
 * HAL_StatusTypeDef HAL_TIM_PWM_Start_IT(TIM_HandleTypeDef *htim,  
 uint32_t Channel)  
 *  
 * 参数：  
 *   htim - 指向 TIM_HandleTypeDef 结构的指针。  
 *   Channel - 要启用的 TIM 通道。  
 * 返回值：HAL 状态。  
 */
```

## 8. HAL\_TIM\_PWM\_Stop\_IT

```
/**  
 * 函数功能：以中断模式停止 PWM 信号生成。  
 */
```

```
* HAL_StatusTypeDef HAL_TIM_PWM_Stop_IT(TIM_HandleTypeDef *htim, uint32_t
Channel)
*
* 参数:
*   htim - 指向 TIM_HandleTypeDef 结构的指针。
*   Channel - 要停用的 TIM 通道。
* 返回值: HAL 状态。
*/

```

## 9. HAL\_TIM\_PWM\_Start\_DMA

```
/**
* 函数功能: 以 DMA 模式启动 TIM PWM 信号生成。
*
* HAL_StatusTypeDef HAL_TIM_PWM_Start_DMA(TIM_HandleTypeDef *htim,
uint32_t Channel, const uint32_t *pData, uint16_t Length)
*
* 参数:
*   htim - 指向 TIM_HandleTypeDef 结构的指针。
*   Channel - 要启用的 TIM 通道。
*   pData - 数据源缓冲区地址。
*   Length - 要从内存传输到 TIM 外设的数据长度。
* 返回值: HAL 状态。
*/

```

## 10. HAL\_TIM\_PWM\_Stop\_DMA

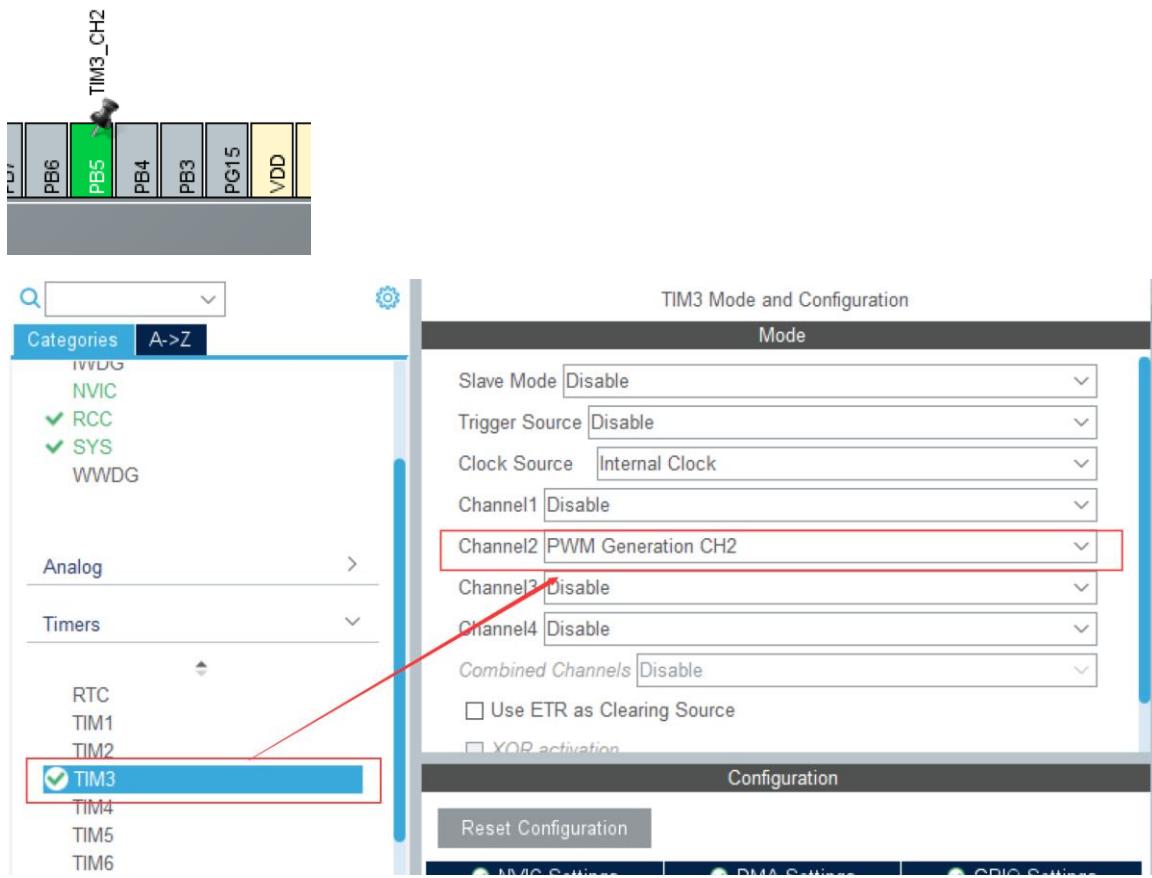
```
/**
* 函数功能: 以 DMA 模式停止 TIM PWM 信号生成。
*
* HAL_StatusTypeDef HAL_TIM_PWM_Stop_DMA(TIM_HandleTypeDef *htim,
uint32_t Channel)
*
* 参数:
*   htim - 指向 TIM_HandleTypeDef 结构的指针。
*   Channel - 要停用的 TIM 通道。
* 返回值: HAL 状态。
*/

```

### 1.14.4 PWM 实现呼吸灯

1.根据电路图，打开 TIM3 的通道 2，对应引脚为 GPIOB\_5。

注意，需要先在此处配置，之后再打开 TIM3



2.配置参数如下图，其中参数可在工程程序中重新调整

### 3.生成工程函数解析

```

void MX_TIM3_Init(void)
{
    /* USER CODE BEGIN TIM3_Init_0 */
    // 用户自定义代码区域，通常用于变量定义或者预处理。

    /* USER CODE END TIM3_Init_0 */

    // 定义时钟配置结构体，并初始化为 0。
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};

    // 定义主配置结构体，并初始化为 0。
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    // 定义输出比较配置结构体，并初始化为 0。
    TIM_OC_InitTypeDef sConfigOC = {0};

    /* USER CODE BEGIN TIM3_Init_1 */
    // 用户自定义代码区域，通常用于初始化前的准备工作。

    /* USER CODE END TIM3_Init_1 */

    // 设置 TIM3 为该函数处理的定时器实例。
    htim3.Instance = TIM3;
}

```

```
// 配置 TIM3 的预分频器，计数模式，周期，时钟分频和自动重载预装载设置。  
htim3.Init.Prescaler = 71;  
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;  
htim3.Init.Period = 5000;  
htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;  
htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;  
  
// 初始化 TIM3 基本计时部分。  
if (HAL_TIM_Base_Init(&htim3) != HAL_OK)  
{  
    Error_Handler(); // 错误处理函数  
}  
  
// 配置 TIM3 的时钟源为内部时钟。  
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;  
if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)  
{  
    Error_Handler(); // 错误处理函数  
}  
  
// 初始化 TIM3 的 PWM 功能。  
if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)  
{  
    Error_Handler(); // 错误处理函数  
}  
  
// 配置 TIM3 的主输出触发和主从模式。  
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;  
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;  
if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) !=  
HAL_OK)  
{  
    Error_Handler(); // 错误处理函数  
}  
  
// 配置 TIM3 的 PWM 通道 2。  
sConfigOC.OCMode = TIM_OCMODE_PWM1;  
sConfigOC.Pulse = 0;  
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;  
sConfigOC.OCFastMode = TIM_OCFAST_ENABLE;  
if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_2) !=  
HAL_OK)  
{  
    Error_Handler(); // 错误处理函数
```

```
}

/* USER CODE BEGIN TIM3_Init_2 */
// 用户自定义代码区域，通常用于初始化后的额外配置或处理。

/* USER CODE END TIM3_Init_2 */

// 调用后处理函数，进一步初始化。
HAL_TIM_MspPostInit(&htim3);
}
```

#### 4.修改 main.c

启动 PWM

```
/* USER CODE BEGIN 2 */

HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
/* USER CODE END 2 */
```

在 whlie 循环中，修改 CRR 寄存器的值，实现不同亮度变化。

```
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    // 呼吸灯效果：逐渐增加亮度
    for (uint16_t i = 0; i < 5000; i++)
    {
        htim3.Instance->CCR2 = i; // 直接设置 TIM3 通道 2 的 CCR 寄存器
        HAL_Delay(1); // 延时以调整变化速度
    }

    // 呼吸灯效果：逐渐减少亮度
    for (uint16_t i = 5000; i > 0; i--)
    {
        htim3.Instance->CCR2 = i; // 直接设置 TIM3 通道 2 的 CCR 寄存器
        HAL_Delay(1); // 延时以调整变化速度
    }
}
/* USER CODE END 3 */
```

此外，也可以通过宏来实现

```
while (1)
{
```

```

/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
    // 呼吸灯效果
    for (uint16_t i = 0; i < 5000; i++)
    {
        __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, i); // 设置 TIM3 通
道 2 的 CCR 值
        HAL_Delay(1); // 延时以调整变化速度
    }

    for (uint16_t i = 5000; i > 0; i--)
    {
        __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, i); // 设置 TIM3 通
道 2 的 CCR 值
        HAL_Delay(1); // 延时以调整变化速度
    }
}

/* USER CODE END 3 */

```

### 1.14.5 SG90 伺服电机基本原理

SG90 是一种常见的微型伺服电机，广泛用于远程控制汽车、船只、直升机以及机器人等领域。图示如下：



sg90 舵机是分为两种的，一种是转角范围 180° 的舵机，另外一种是 360° 转角的舵机。

180° 舵机是给一个 PWM 信号就转动到一定的角度，然后保持在这个转动之后的位置，直到有下一个不同的 PWM 信号，才会转到其他的角度。

360° 舵机是给一个 PWM 信号，就会按照一定的速度转动。

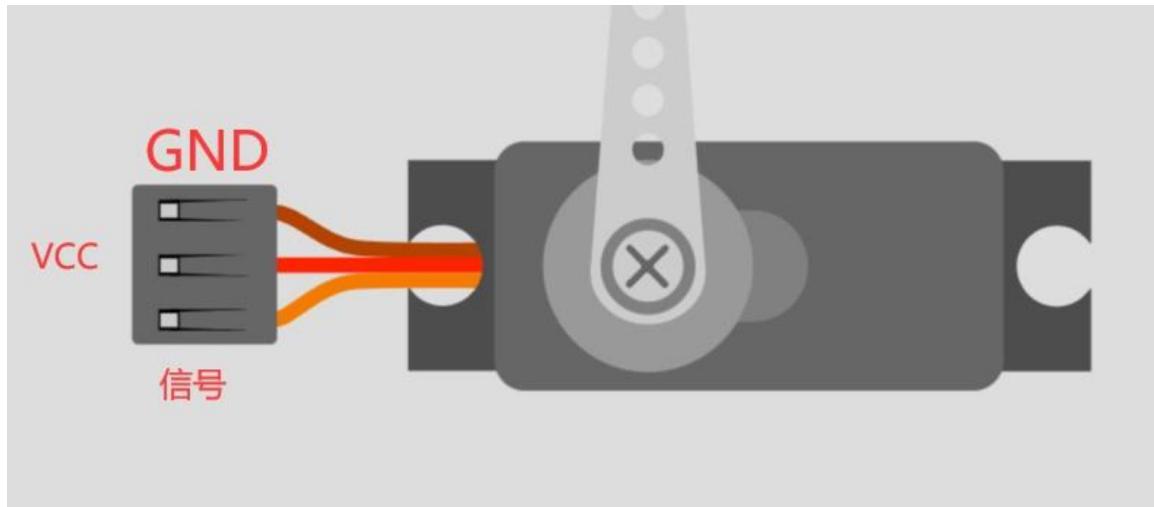
我们使用的是 360° 舵机，电平持续时间与转速关系如下：

电平时间	转速
0.5ms	正转最大速度
1.5ms	转速为 0
2.5ms	反转最大速度

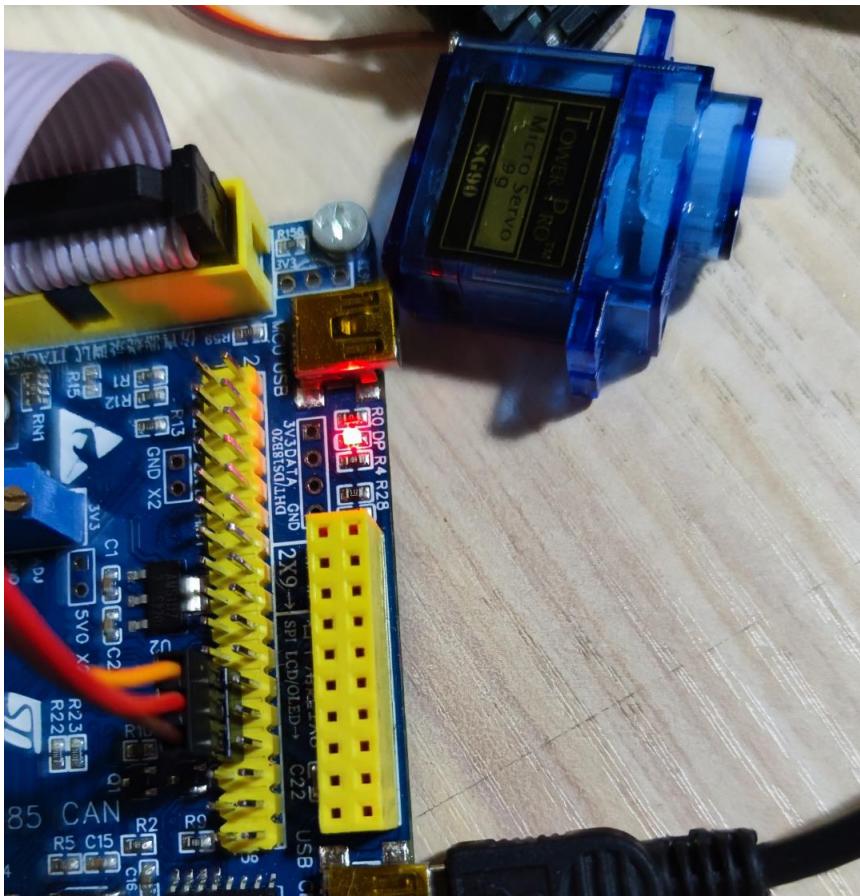
即周期 20ms 时，占空比在 2.5%-7.5% 时为正转，2.5% 时为正转最大速度，占空比在 7.5%-12.5% 时为反转，12.5% 时为反转最大速度。

#### 1.14.6 PWM 驱动舵机

1.接线方式如下：



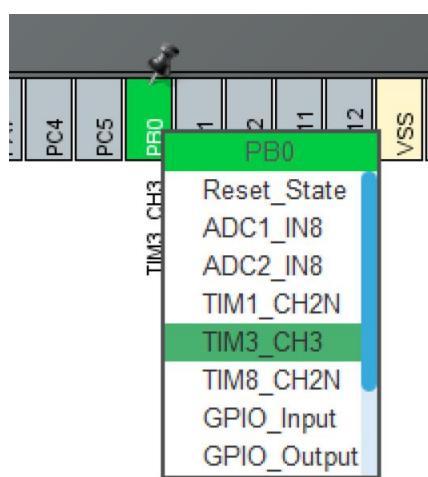
实物接线图如下：信号线使用 GPIOB\_0 引脚。

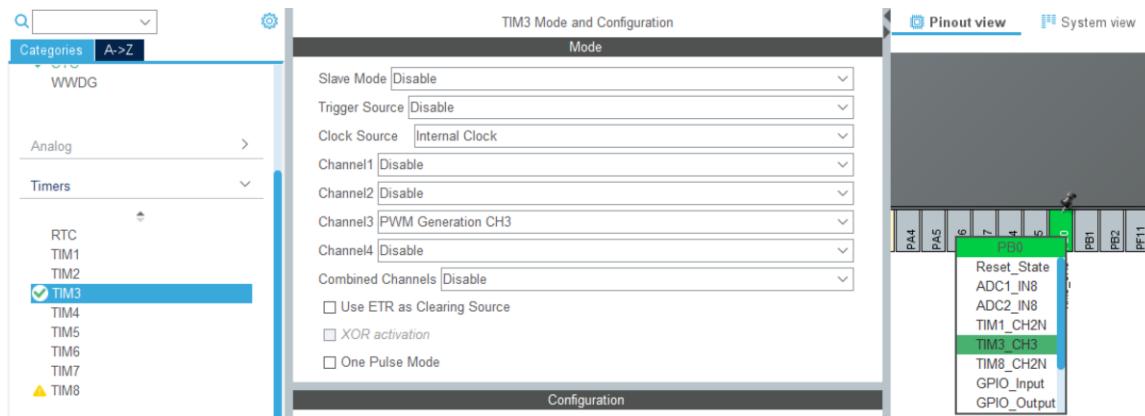


## 2.CubeMX 配置

根据 SG90 舵机驱动原理，我们首先配置 PWM 周期为 20ms（对应 50Hz 频率）

配置 CubeMX 中 PB0 引脚为 TIM3 CH3，即定时器 3 的通道 3。





TIM3 时钟为 72MHz, 我们可以如下计算预分频器和周期值:

PWM 频率 = 50Hz (20ms 周期)

预分频器 = (TIM 时钟频率 / PWM 频率) / 定时器计数器范围 - 1

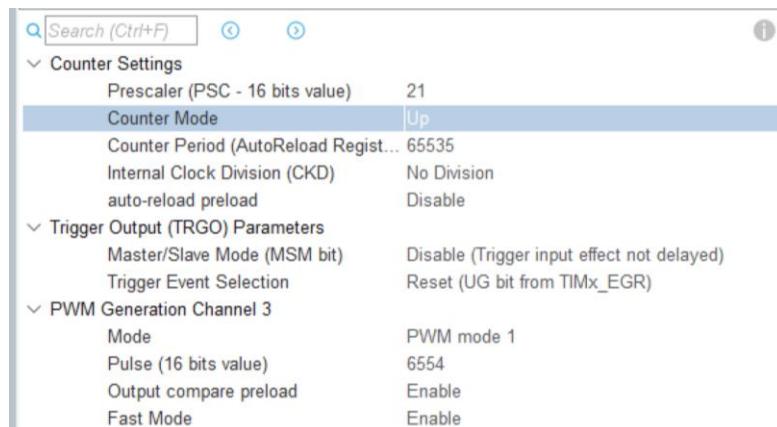
例如, 如果定时器计数器范围是 65536 (16 位定时器), 则预分频器 =  $(72000000 / 50) / 65536 - 1 \approx 21$

接下来, 配置 CCR 寄存器以生成 10% 的占空比:

CCR 值 = (占空比 \* 定时器计数器范围)

例如, CCR 值 =  $(10\% * 65536) \approx 6554$

具体配置如下



### 3.工程修改

生成工程后, 在 main.c 添加 PWM 启动函数

```
/* USER CODE BEGIN 2 */
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);
/* USER CODE END 2 */
```

## 1.15 定时器输入捕获模式 (Input Capture)

### 1.15.1 输入捕获模式概述

STM32 的定时器输入捕获模式允许微控制器捕获外部信号的时间信息，从而能够测量频率、周期和脉冲宽度等参数。这个模式对于那些需要精确测量外部事件时间的应用尤为重要。

### 1.15.2 输入捕获模式相关配置参数

Input Capture Channel 2	
Polarity Selection	Rising Edge
IC Selection	Direct
Prescaler Division Ratio	No division
Input Filter (4 bits value)	0

**极性选择 (Polarity Selection) :**

这个参数设置的是 TIMx\_CCER->CCxP，用于配置捕获触发的信号极性，即决定是在信号的上升沿还是下降沿进行捕获。

**IC 选择 (IC Selection) :**

该设置涉及 TIMx\_CCMR->CCxS，用于配置通道的方向（输入或输出）以及输入脚的选择。例如，Direct 表示 CCx 通道配置为输入，ICx 映射到 TI1 上。

**预分频比 (Prescaler Division Ratio) :**

此参数设置的是 TIMx\_CCMR->ICxPSC，用于配置输入捕获的预分频系数。预分频系数决定了捕获操作的频率，从而可以影响捕获精度和反应时间。

**输入过滤器 (Input Filter) :**

该设置涉及 TIMx\_CCMR->ICxF，用于配置输入捕获的滤波器。输入过滤器的作用是对输入信号进行去噪或防抖处理，特别有用于处理机械开关或类似的高噪声信号源。

### 1.15.3 输入捕获模式相关 API

#### 1. HAL\_TIM\_IC\_Init

```
/**  
 * 函数功能：根据 TIM_HandleTypeDef 中指定的参数初始化 TIM 输入捕获。  
 *  
 * HAL_StatusTypeDef HAL_TIM_IC_Init(TIM_HandleTypeDef *htim)  
 *  
 * 参数：  
 *     htim - 指向 TIM_HandleTypeDef 结构的指针，包含 TIM 输入捕获模块的配置信息。  
 */
```

```
* 返回值: HAL 状态。  
*/
```

## 2. HAL\_TIM\_IC\_DeInit

```
/**  
 * 函数功能: 反初始化 TIM 外设。  
 *  
 * HAL_StatusTypeDef HAL_TIM_IC_DeInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数:  
 *     htim - 指向 TIM_HandleTypeDef 结构的指针。  
 * 返回值: HAL 状态。  
 */
```

## 3. HAL\_TIM\_IC\_MspInit

```
/**  
 * 函数功能: 初始化 TIM 输入捕获 MSP。  
 *  
 * void HAL_TIM_IC_MspInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数:  
 *     htim - 指向 TIM_HandleTypeDef 结构的指针。  
 * 返回值: 无。  
 */
```

## 4. HAL\_TIM\_IC\_MspDeInit

```
/**  
 * 函数功能: 反初始化 TIM 输入捕获 MSP。  
 *  
 * void HAL_TIM_IC_MspDeInit(TIM_HandleTypeDef *htim)  
 *  
 * 参数:  
 *     htim - 指向 TIM_HandleTypeDef 结构的指针。  
 * 返回值: 无。  
 */
```

## 5. HAL\_TIM\_IC\_Start

```
/**  
 * 函数功能: 启动 TIM 输入捕获测量。  
 *  
 * HAL_StatusTypeDef HAL_TIM_IC_Start(TIM_HandleTypeDef *htim, uint32_t  
 Channel)  
 *  
 * 参数:
```

```
*      htim - 指向 TIM_HandleTypeDef 结构的指针。
*      Channel - 要启用的 TIM 通道。
* 返回值: HAL 状态。
*/
```

## 6. HAL\_TIM\_IC\_Stop

```
/**
 * 函数功能: 停止 TIM 输入捕获测量。
*
* HAL_StatusTypeDef HAL_TIM_IC_Stop(TIM_HandleTypeDef *htim, uint32_t
Channel)
*
* 参数:
*      htim - 指向 TIM_HandleTypeDef 结构的指针。
*      Channel - 要禁用的 TIM 通道。
* 返回值: HAL 状态。
*/
```

## 7. HAL\_TIM\_IC\_Start\_IT

```
/**
 * 函数功能: 以中断模式启动 TIM 输入捕获测量。
*
* HAL_StatusTypeDef HAL_TIM_IC_Start_IT(TIM_HandleTypeDef *htim, uint32_t
Channel)
*
* 参数:
*      htim - 指向 TIM_HandleTypeDef 结构的指针。
*      Channel - 要启用的 TIM 通道。
* 返回值: HAL 状态。
*/
```

## 8. HAL\_TIM\_IC\_Stop\_IT

```
/**
 * 函数功能: 以中断模式停止 TIM 输入捕获测量。
*
* HAL_StatusTypeDef HAL_TIM_IC_Stop_IT(TIM_HandleTypeDef *htim, uint32_t
Channel)
*
* 参数:
*      htim - 指向 TIM_HandleTypeDef 结构的指针。
*      Channel - 要禁用的 TIM 通道。
* 返回值: HAL 状态。
*/
```

## 9. HAL\_TIM\_IC\_Start\_DMA

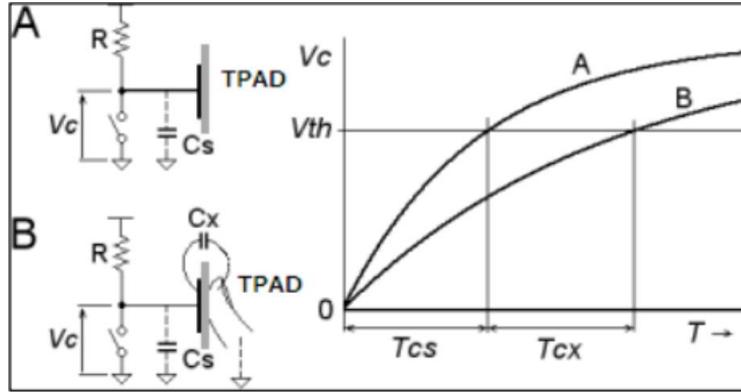
```
/**  
 * 函数功能：以 DMA 模式启动 TIM 输入捕获测量。  
 *  
 * HAL_StatusTypeDef HAL_TIM_IC_Start_DMA(TIM_HandleTypeDef *htim,  
 uint32_t Channel, uint32_t *pData, uint16_t Length)  
 *  
 * 参数：  
 *      htim - 指向 TIM_HandleTypeDef 结构的指针。  
 *      Channel - 要启用的 TIM 通道。  
 *      pData - 目标缓冲区地址。  
 *      Length - 从 TIM 外设到内存的数据传输长度。  
 * 返回值：HAL 状态。  
 */
```

## 10. HAL\_TIM\_IC\_Stop\_DMA

```
/**  
 * 函数功能：以 DMA 模式停止 TIM 输入捕获测量。  
 *  
 * HAL_StatusTypeDef HAL_TIM_IC_Stop_DMA(TIM_HandleTypeDef *htim, uint32_t  
 Channel)  
 *  
 * 参数：  
 *      htim - 指向 TIM_HandleTypeDef 结构的指针。  
 *      Channel - 要禁用的 TIM 通道。  
 * 返回值：HAL 状态。  
 */
```

### 1.15.4 电容触摸按键原理

我们使用的是检测电容充放电时间的方法来判断是否有触摸，图中的 A、B 分别表示有人体按下时电容的充放电曲线。其中 R 是外接的电容充电电阻，Cs 是没有触摸按下时 TPAD 与 PCB 之间的杂散电容。而 Cx 则是有手指按下的时候，手指与 TPAD 之间形成的电容。图中的开关是电容放电开关（实际使用时，由 STM32F1 的 IO 代替）。



先用开关将  $C_s$  (或  $C_s+C_x$ ) 上的电放尽，然后断开开关，让  $R$  给  $C_s$  (或  $C_s+C_x$ ) 充电，当没有手指触摸的时候， $C_s$  的充电曲线如图中的 A 曲线。而当有手指触摸的时候，手指和 TPAD 之间引入了新的电容  $C_x$ ，此时  $C_s+C_x$  的充电曲线如图中的 B 曲线。从上图可以看出，A、B 两种情况下， $V_c$  达到  $V_{th}$  的时间分别为  $T_{cs}$  和  $T_{cs}+T_{cx}$ 。其中，除了  $C_s$  和  $C_x$  我们需要计算，其他都是已知的，根据电容充放电公式：

$$V_c = V_0 \times (1 - e^{(-\frac{t}{RC})})$$

我们使用 PA1 来检测 TPAD 是否有触摸，在每次检测之前，我们先配置 PA1 为推挽输出，将电容  $C_s$  (或  $C_s+C_x$ ) 放电，然后配置 PA1 为浮空输入，利用外部上拉电阻给电容  $C_s(C_s+C_x)$  充电，同时开启 TIM5\_CH2 的输入捕获，检测上升沿，当检测到上升沿的时候，就认为电容充电完成了，完成一次捕获检测。

每次复位重启的时候，我们执行一次捕获检测（可以认为没触摸），记录此时的值，记为 `tpad_default_val`，作为判断的依据。在后续的捕获检测，我们就通过与 `tpad_default_val` 的对比，来判断是不是有触摸发生。

### 1.15.5 电容按键控制 LED 亮灭

1. 使用跳线帽，将 TPAD 键和 PA1 引脚连接



2. CubeMX 配置, 配置 PB5 (LED) , RCC (时钟)等参数后, 打开 TIM2 的通道 2.

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM2 global interrupt	<input checked="" type="checkbox"/>	0	0

TIM2 具体配置参数如下

▼ Counter Settings

Prescaler (PSC - 16 bits value)	71
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value)	65535
Internal Clock Division (CKD)	No Division
auto-reload preload	Enable

➤ Trigger Output (TRGO) Parameters ← 不配置

▼ Input Capture Channel 2

Polarity Selection	Rising Edge
IC Selection	Direct
Prescaler Division Ratio	No division
Input Filter (4 bits value)	0

在 NVIC 选项中打开中断

NVIC Settings	DMA Settings	GPIO Settings	
Parameter Settings	User Constants		
NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM2 global interrupt	<input checked="" type="checkbox"/>	0	0

生成 TAPD 响应工程

### 3.工程修改

在工程目录下 Core 文件夹中的 Src 与 Inc 文件夹分别新建 tpad.c 和 tpad.h

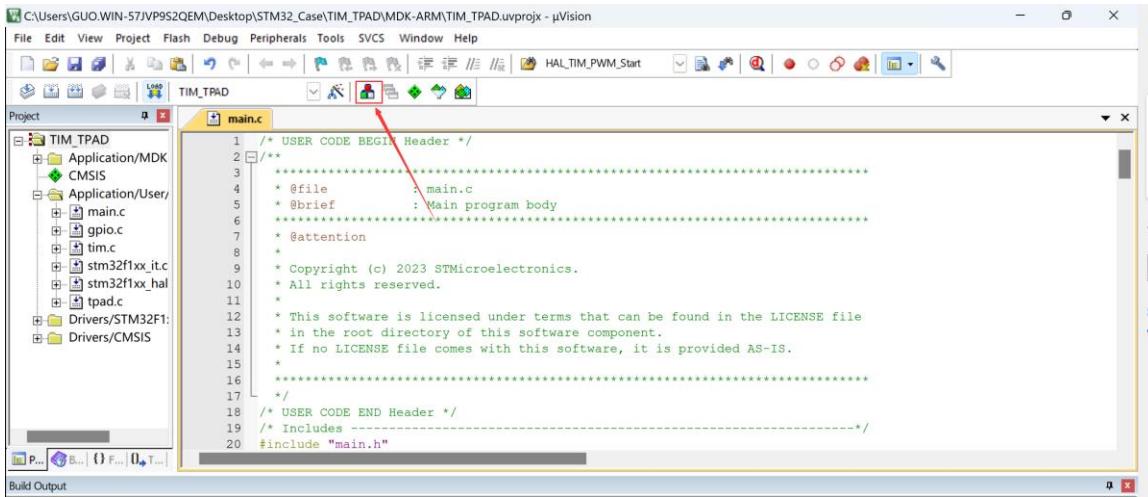
STM32\_Case > TIM\_TPAD > Core > Src

名称	修改日期	类型	大小
gpio.c	2023/11/29 14:28	C 源文件	
main.c	2023/11/29 14:49	C 源文件	
stm32f1xx_hal_msp.c	2023/11/29 14:28	C 源文件	
stm32f1xx_it.c	2023/11/29 14:37	C 源文件	
system_stm32f1xx.c	2023/7/4 0:07	C 源文件	1
tim.c	2023/11/29 14:37	C 源文件	
tpad.c	2023/11/29 14:32	C 源文件	

STM32\_Case > TIM\_TPAD > Core > Inc

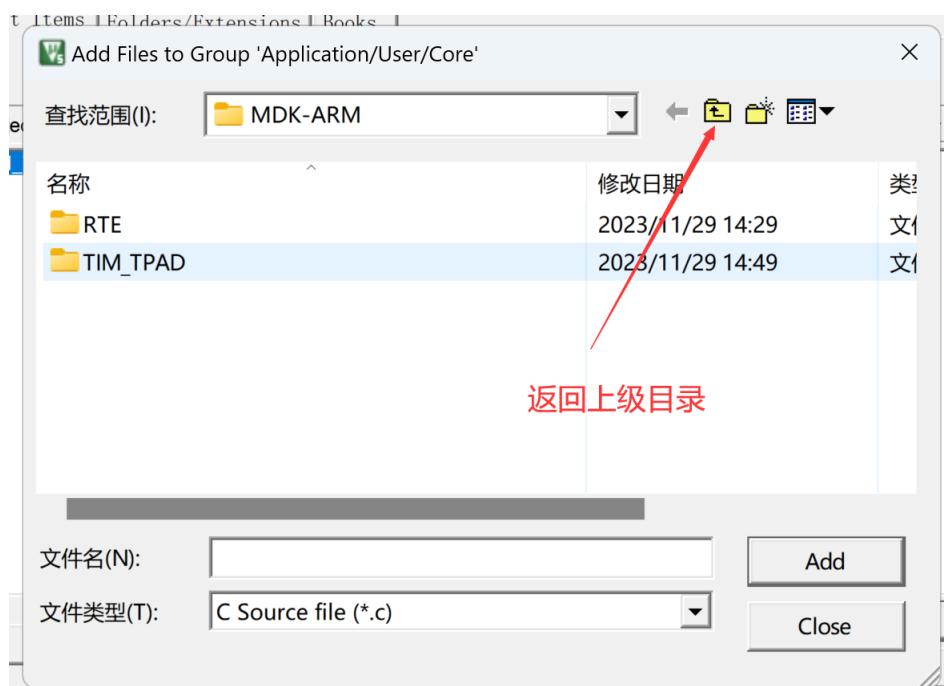
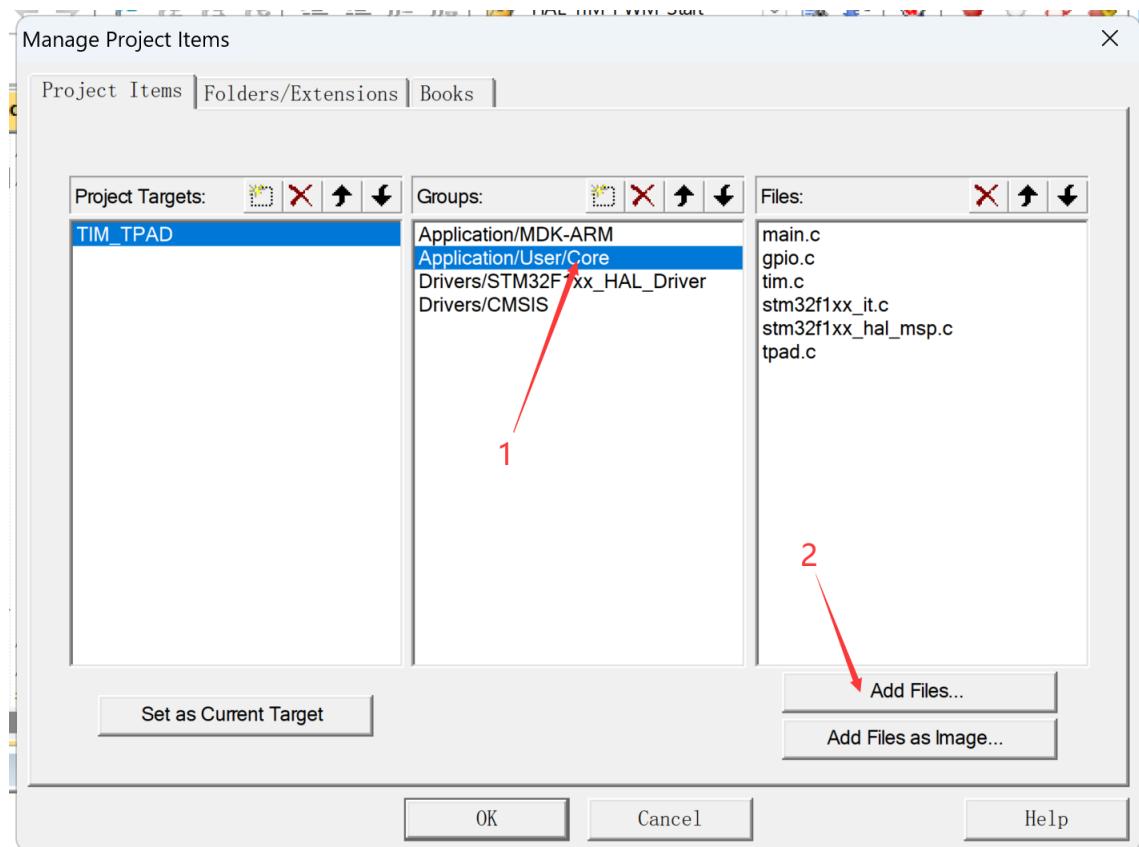
名称	修改日期	类型	大小
gpio.h	2023/11/29 14:28	C Header 源文件	
main.h	2023/11/29 14:28	C Header 源文件	
stm32f1xx_hal_conf.h	2023/11/29 14:28	C Header 源文件	
stm32f1xx_it.h	2023/11/29 14:37	C Header 源文件	
tim.h	2023/11/29 14:28	C Header 源文件	
tpad.h	2023/11/29 14:31	C Header 源文件	

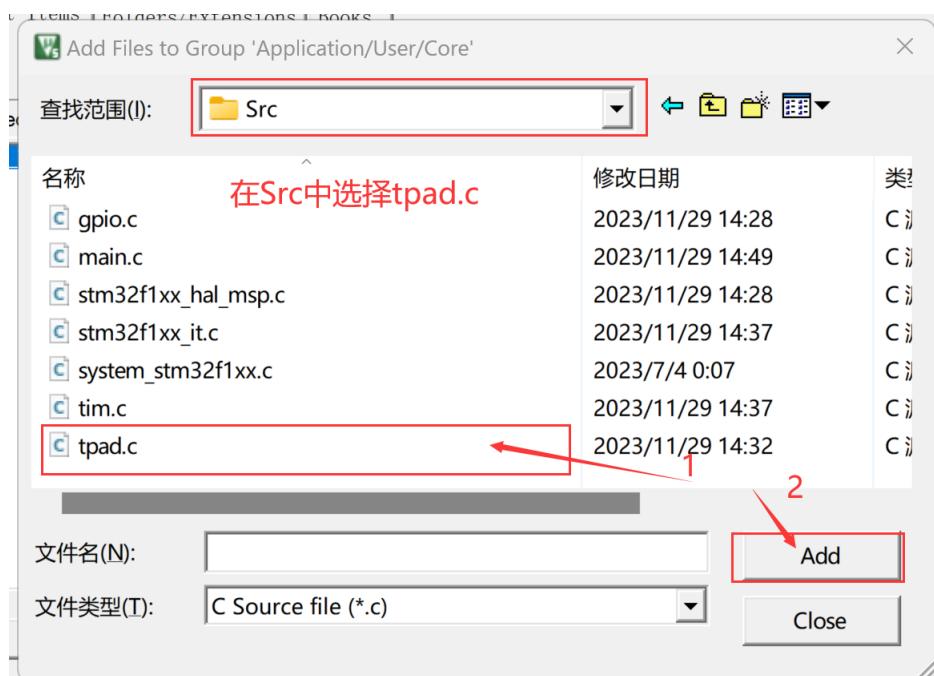
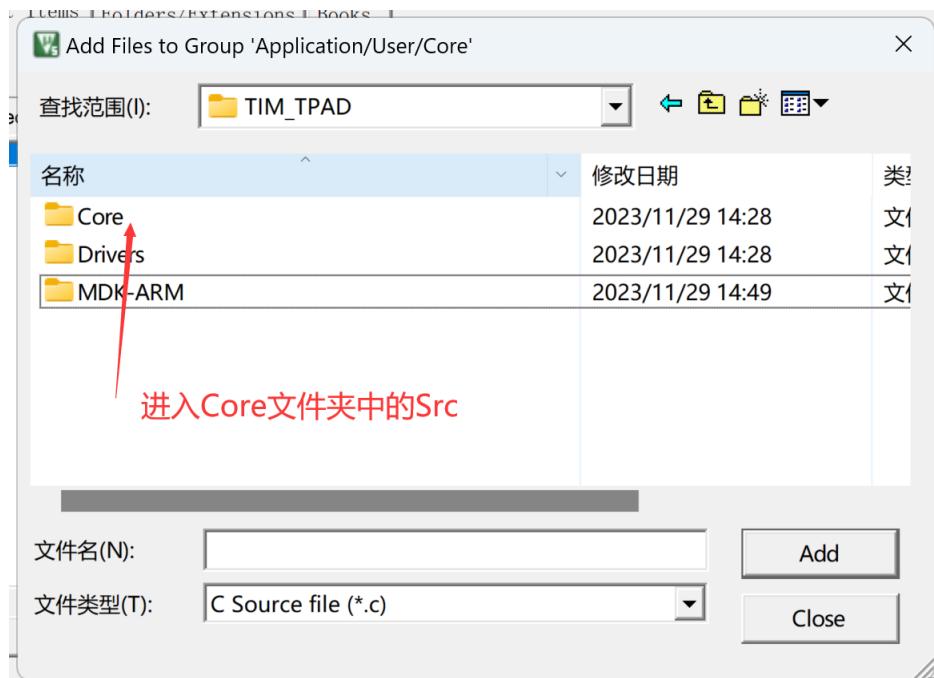
## Keil 软件中



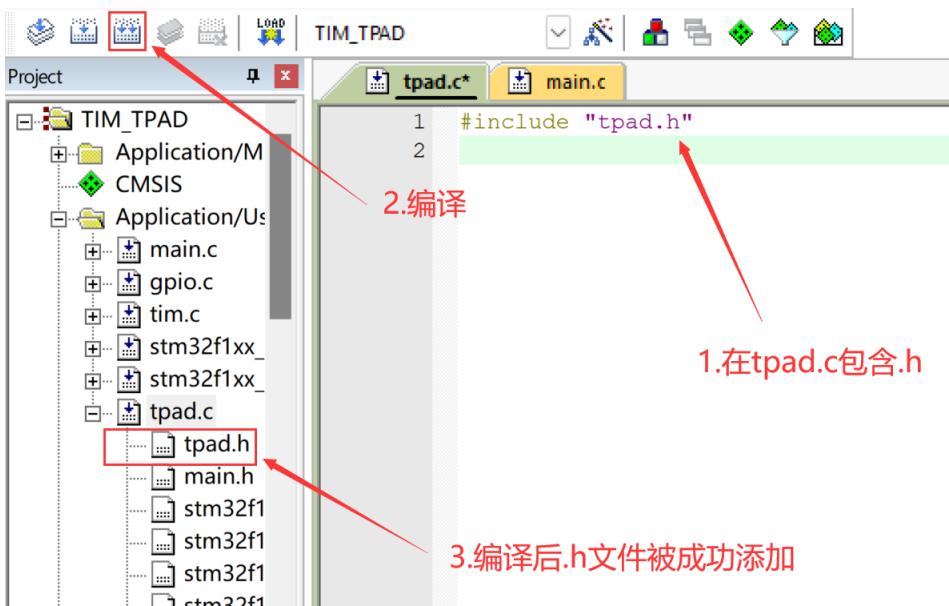
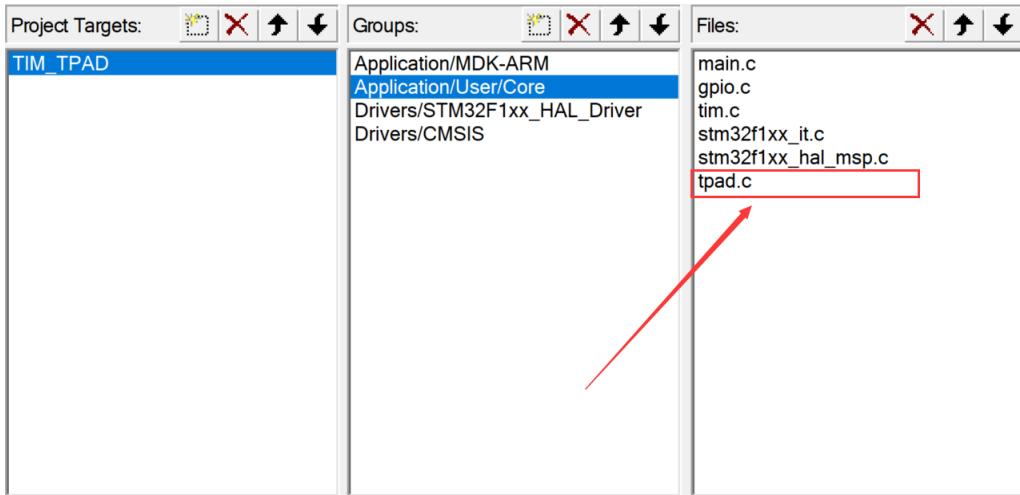
The screenshot shows the Keil uVision IDE interface. The project tree on the left shows files like Application/MDK, CMSIS, and various source files (main.c, gpio.c, tim.c, etc.). The main editor window displays the content of main.c, which includes a header comment and an #include directive for "main.h".

```
1 /* USER CODE BEGIN Header */
2 /**
3  * @file    : main.c
4  * @brief   : Main program body
5  *
6  * @attention
7  *
8  * Copyright (c) 2023 STMicroelectronics.
9  * All rights reserved.
10 *
11 *
12 * This software is licensed under terms that can be found in the LICENSE file
13 * in the root directory of this software component.
14 * If no LICENSE file comes with this software, it is provided AS-IS.
15 *
16 */
17 */
18 /* USER CODE END Header */
19 /* Includes */
20 #include "main.h"
```





只单击一次，添加成功系统无提示，如下图显示，成功添加 tpad.c。



在 tpad.c 中添加

```
#include "tpad.h"
#include "tim.h"          // 需要使用到定时器的捕获功能

/*
@brief      复位 TPAD
#node       将 TPAD 按键看作是一个电容，手指按下和不按下电容值有变化
           先将 GPIO 设置为推挽输出，输出 0，进行放电，在设置为 GOIP 为
           浮空输入，等待电容充电，并且捕获上拉
*/

static uint8_t flag = 0;          // 检测是否执行了输入捕获回调
函数
uint16_t temp = 0;              // 存取捕获的值
```

```
uint16_t tpad_default_val; // 手指不按下时，电容充电到高  
电平的时间  
  
static void tpad_reset(void)  
{  
    GPIO_InitTypeDef GPIO_InitStruct = {0};  
  
    GPIO_InitStruct.Pin = GPIO_PIN_1; // 将 PA1 设置为推挽输出  
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;  
    GPIO_InitStruct.Pull = GPIO_PULLUP; // 上拉电阻  
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;  
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);  
  
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_RESET); // 将 PA1 置 0，  
对电容进行放电  
  
    htim2.Instance->SR = 0; // 清除标记  
    htim2.Instance->CNT = 0; // 归零  
  
    GPIO_InitStruct.Pin = GPIO_PIN_1; // 设置为输入模  
式，进行输入捕获  
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;  
    GPIO_InitStruct.Pull = GPIO_NOPULL;  
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);  
  
    // 设置为上升沿捕获  
    __HAL_TIM_SET_CAPTUREPOLARITY(&htim2, TIM_CHANNEL_2,  
TIM_INPUTCHANNELPOLARITY_RISING);  
  
    HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_2); // 开启定时器捕获  
}  
  
static void tpad_get_val(void)  
{  
    tpad_reset(); // 先进行复位(放电，置位，  
开启定时器捕获)  
    while(flag == 0) // 判断捕获中断回调函数  
是否捕获完成  
    {  
        HAL_Delay(1);  
    }  
    flag = 0; // 捕获完成后再次将 flag  
置 0  
}
```

```

// 捕获中断回调函数
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    HAL_TIM_IC_Stop_IT(&htim2, TIM_CHANNEL_2);           // 先关闭定时器捕获
    if (TIM2 == htim->Instance){
        temp = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_2);   // 获取
当前捕获值
        flag = 1;                                              // 将 flag 置 1, 说明捕获
完成
    }
}

/**
* @brief 读取 n 次, 取最大值
* @param n : 连续获取的次数
* @retval n 次读数里面读到的最大读数值
*/
static uint16_t tapt_get_maxval(uint8_t n)
{
    uint16_t maxval = 0;
    while (n--) {
        tpad_get_val();          // 进行数据捕获, 捕获成功后 temp 的值更新为最新的捕
获值
        if (temp > maxval) {
            maxval = temp;    // 取捕获的最大值
        }
    }
    return maxval;             // 将数值返回
}

// 初始化触摸按键
void tpad_init(void)
{
    uint16_t buf[10];
    uint16_t m;
    uint8_t i, j;

    for (i = 0; i < 10; i++) { // 获取 10 个数据
        tpad_get_val();
        buf[i] = temp;
    }

    for (i = 0; i < 9; i++) { // 进行排序
        for (j = i+1; j < 10; j++) {

```

```

        if (buf[i] < buf[j]){
            m = buf[i];
            buf[i] = buf[j];
            buf[j] = m;
        }
    }
}

m = 0;
for (i = 2; i < 8; i++) {           // 取中间的 6 个数据
    m += buf[i];
}

tpad_default_val = m / 6;          // 求平均值作为没有触摸时的值
}

/***
 * @brief 扫描触摸按键
 * @param mode : 扫描模式
 * @arg 0, 不支持连续触发(按下一次必须松开才能按下次);
 * @arg 1, 支持连续触发(可以一直按下)
 * @retval 0, 没有按下; 1, 有按下;
 */
uint8_t tapd_scan(uint8_t mode)      // 扫秒模式
{
    static uint8_t keyen = 0;         /* 0, 可以开始检测; >0, 还不能开始检测;
*/
    uint8_t res = 0;
    uint8_t sample = 3;              /* 默认采样次数为 3 次 */
    uint16_t rval = 0;

    if (mode) {                     // mode = 1 为扫描模式,
        sample = 6;                 /* 支持连接的时候, 设置采样次数为 6 次 */
        keyen = 0;                  /* 支持连接, 每次调用该函数都可以检测 */
    }
    rval = tapt_get_maxval(sample); // 获取读取的值
    if (rval > (tpad_default_val + 15)) {
        if (keyen == 0) {
            res = 1;
        }
        keyen = 3;
    }
    if (keyen) keyen--;             // 单次触发松手三次后将 keyen 置 0, 然后才
能将 res 赋值为 1
    return res;
}

```

```
}
```

在 tpad.h 中添加

```
#ifndef __TPAD_H__
#define __TPAD_H__

#include "main.h"

void tpad_init(void);           // 初始化 tpad
uint8_t tpad_scan(uint8_t mode); // 扫描函数 tpad_scan

#endif
```

在 main.c 中修改

首先包含 tpad.h

```
/* USER CODE BEGIN Includes */
#include "tpad.h"
/* USER CODE END Includes */
```

在 main 函数中初始化 tpad

```
/* USER CODE BEGIN 2 */
tpad_init();
/* USER CODE END 2 */
```

在主循环中循环检测

```
while (1)
{
    if (tpad_scan(0)) {          // 非连续触发模式，返回 1 按下，返回 0 没有按下
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_5);
    }
    HAL_Delay(100);
/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
}
```

问题：

```
//设置极性
//设置TIM2的通道2为上升沿捕获
HAL_TIM_SetCapturePolarity(&htim2, TIM_CHANNEL_2, TIM_INPUTCHANNELPOLARITY_RISING);
error: expected expression
```

```

1731 .ERVALUE(__HANDLE__, __CHANNEL__)
1732 IANNEL_1) ? ((__HANDLE__)->Instance->CCMR1 &= ~TIM_CCMR1_IC1PSC) :\ 
1733 IANNEL_2) ? ((__HANDLE__)->Instance->CCMR1 &= ~TIM_CCMR1_IC2PSC) :\ 
1734 IANNEL_3) ? ((__HANDLE__)->Instance->CCMR2 &= TIM_CCMR2_IC3PSC) :\ 
1735 ->CCMR2 &= ~TIM_CCMR2_IC4PSC)
1736 
1737 
1738 IARITY(__HANDLE__, __CHANNEL__, __POLARITY__)
1739 IANNEL_1) ? ((__HANDLE__)->Instance->CCER |= (__POLARITY__)) :\ 
1740 IANNEL_2) ? ((__HANDLE__)->Instance->CCER |= ((__POLARITY__) << 4U)) :\ 
1741 IANNEL_3) ? ((__HANDLE__)->Instance->CCER |= ((__POLARITY__) << 8U)) :\ 
1742 ->CCER |= (((__POLARITY__) << 12U)))
1743 
1744 IOLARITY(__HANDLE__, __CHANNEL__)
1745 IANNEL_1) ? ((__HANDLE__)->Instance->CCER &= ~(TIM_CCER_CC1P | TIM_CCER_CC1NP)) :\ 
1746 IANNEL_2) ? ((__HANDLE__)->Instance->CCER &= ~(TIM_CCER_CC2P | TIM_CCER_CC2NP)) :\ 
1747 IANNEL_3) ? ((__HANDLE__)->Instance->CCER &= ~(TIM_CCER_CC3P)) :\ 
1748 ->CCER &= (TIM_CCER_CC4P))
1749 
1750 
1751 
1752 -----*/
1753 
1754 
1755 /* module */
1756 
1757

```

实验效果：按下触摸键，则改变 LED 灯状态。

## 1.15.6 作业

电容按键控制舵机启停，key1/0 控制舵机转速

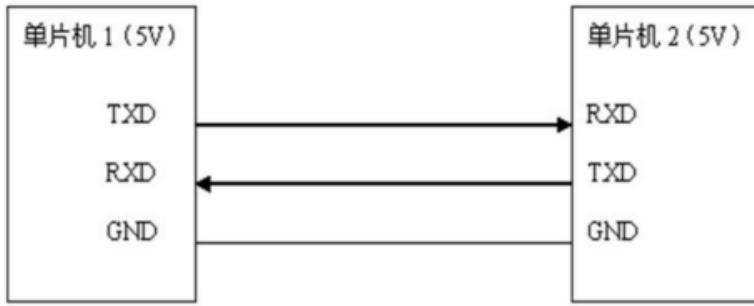
## 1.16 串口通信

### 1.16.1 串口通信概述

通用同步异步收发器(Universal Synchronous Asynchronous Receiver and Transmitter)是一个串行通信设备，可以灵活地与外部设备进行全双工数据交换。有别于 USART 还有一个 UART(Universal Asynchronous Receiver and Transmitter)，它是在 USART 基础上裁剪掉了同步通信功能，只有异步通信。简单区分同步和异步就是看通信时需不需要对外提供时钟输出，我们平时用的串口通信基本都是 UART。

串行通信一般是以帧格式传输数据，即是一帧一帧的传输，每帧包含有起始信号、数据信息、停止信息，可能还有校验信息。USART 就是对这些传输参数有具体规定，当然也不是只有唯一一个参数值，很多参数值都可以自定义设置，只是增强它的兼容性。USART 满足外部设备对工业标准 NRZ 异步串行数据格式的要求，并且使用了小数波特率发生器，可以提供多种波特率，使得它的应用更加广泛。

接口通过三个引脚与其他设备连接在一起，任何 USART 双向通信至少需要两个脚：接收数据输入(RX)和发送数据输出(TX)。RX：接收数据串行输入。通过过采样技术来区别数据和噪音，从而恢复数据。TX：发送数据输出。当发送器被禁止时，输出引脚恢复到它的 I/O 端口配置。当发送器被激活，并且不发送数据时，TX 引脚处于高电平。



### 电平标准:

1. TTL 电平: +3.3V 或 +5V 表示 1, 0V 表示 0
2. RS232 电平: -3~-15V 表示 1, +3~+15V 表示 0
3. RS485 电平: 两线压差 +2~+6V 表示 1, -2~-6V 表示 0 (差分信号)

### 带流控的串口数据传输

此种方式使用 4 根线传输数据, 即, TXD、RXD、CTS、RTS。和基本数据传输方式相比增加了 CTS 和 RTS 两个管脚, CTS 和 RTS 也是交叉相接。

**RTS:** Require To Send 缩写, 请求发送, 此管脚为输出管脚, 用于指示自己可以接收数据, 输出低电平表示可以接收数据, 输出高电平指示不能接收数据。

**CTS:** Clear To Send 缩写, 允许发送, 此管脚为输入管脚, 用于判断发送方是否能够接收数据, 读取到低电平表示对方能够接收数据, 读取到高电平表示对方不能接收数据。

这种传输方式能够保证数据传输不会丢数据, 保证数据的完整性。例如, MCU1 向 MCU2 发送数据, 此时 MCU2 正在忙于其他任务, 无暇顾及串口接收, MCU1 发送的数据把 MCU2 的 FIFO 填满后, MCU2 会把自己的 RTS 拉高, 指示自己不能接收数据, 此时 MCU1 发现自己的 CTS 变高了, MCU1 的数据发送会进入等待状态, 直到 MCU2 把自己的串口 FIFO 中的数据读取出来后, MCU2 的 RTS 会自动变低, 这时候 MCU1 可以继续发送数据。这种带硬件流控的传输方式, 保证了数据的完整性。

### 1.16.2 串口通信时序

一个数据帧包括一个起始位、数据位、校验位和停止位。起始位用于表示数据帧的开始, 停止位用于表示数据帧的结束。校验位用于检测传输错误。在数据帧发送之前, 发送方会设置校验位, 使得数据帧具有偶数或奇数的奇偶校验。接收方和发送方会提前协商使用哪种类型的奇偶校验。如果接收到的数据帧校验错误, 接收方会知道传输过程中发生了错误, 并可以要求发送方重新发送数据帧。



起始位：占用 1Bit，低电平有效

数据位：可以是 5bit、6Bit、7Bit、8Bit，其中最常用是 8Bit

校验位：奇校验、偶校验、无校验，占用 1bit，无校验位时不占用。

偶校验（even parity）：校验原则是，数据位和校验位中 1 的个数为偶数

奇校验（odd parity）：校验原则是，数据位和校验位中 1 的个数为奇数

无校验：即时序图中没有校验位

停止位：占用 1Bit、1.5Bit、2Bit，高电平有效

波特率

波特率是衡量在通信通道中传输信息速率的指标，常用于串行通信。例如，

“9600 波特”意味着串口每秒最多能传输 9600 比特。波特率代表每秒传输的信号变化或符号数量，以波特（Bd）为单位。较高的波特率意味着通信设备之间信息的传输和接收更快。

常用的串口通讯速率：2400bps、4800bps、9600bps、19200bps、38400bps、115200bps。现在最常用的应该是 115200bps 的速率。

### 1.16.3 串口通信结构体详解

```
/**
 * @brief  UART handle Structure definition
 */
typedef struct __UART_HandleTypeDef
{
    USART_TypeDef                *Instance;          /*!< USART 寄存器基地址 */
    UART_InitTypeDef             Init;              /*!< USART 通信参数 */
    const uint8_t                 *pTxBuffPtr;        /*!< 指向 USART Tx 传输缓冲
区的指针 */
    uint16_t                      TxXferSize;        /*!< USART Tx 传输大小 */
}
```

```

__IO uint16_t          TxXferCount;      /*!< UART Tx 传输计数器 */

uint8_t                *pRxBuffPtr;        /*!< 指向 UART Rx 传输缓冲
区的指针 */

uint16_t               RxXferSize;        /*!< UART Rx 传输大小 */

__IO uint16_t          RxXferCount;        /*!< UART Rx 传输计数器 */

__IO HAL_UART_RxTypeTypeDef ReceptionType; /*!< 正在进行的接收类型 */

__IO HAL_UART_RxEventTypeTypeDef RxEventType; /*!< Rx 事件类型 */

DMA_HandleTypeDef      *hdmatx;           /*!< UART Tx DMA 处理参数
*/
DMA_HandleTypeDef      *hdmarx;           /*!< UART Rx DMA 处理参数
*/
HAL_LockTypeDef         Lock;              /*!< 锁定对象 */

__IO HAL_UART_StateTypeDef gState;         /*!< 与全局句柄管理和 Tx 操
作相关的 UART 状态信息
此参数可以是@ref
HAL_UART_StateTypeDef 的一个值 */

__IO HAL_UART_StateTypeDef RxState;        /*!< 与 Rx 操作相关的 UART
状态信息
此参数可以是@ref
HAL_UART_StateTypeDef 的一个值 */

__IO uint32_t            ErrorCode;         /*!< UART 错误代码 */

#if (USE_HAL_UART_REGISTER_CALLBACKS == 1)
  void (* TxHalfCpltCallback)(struct __UART_HandleTypeDef
*huart);      /*!< UART Tx 半完成回调 */
  void (* TxCpltCallback)(struct __UART_HandleTypeDef
*huart);      /*!< UART Tx 完成回调 */
  void (* RxHalfCpltCallback)(struct __UART_HandleTypeDef
*huart);      /*!< UART Rx 半完成回调 */
  void (* RxCpltCallback)(struct __UART_HandleTypeDef
*huart);      /*!< UART Rx 完成回调 */
  void (* ErrorCallback)(struct __UART_HandleTypeDef
*huart);      /*!< UART 错误回调 */

```

```

    void (* AbortCpltCallback)(struct __UART_HandleTypeDef
*huart); /*!< UART 中止完成回调 */
    void (* AbortTransmitCpltCallback)(struct __UART_HandleTypeDef *huart);
/*!< UART 中止传输完成回调 */
    void (* AbortReceiveCpltCallback)(struct __UART_HandleTypeDef
*huart); /*!< UART 中止接收完成回调 */
    void (* WakeupCallback)(struct __UART_HandleTypeDef
*huart); /*!< UART 唤醒回调 */
    void (* RxEventCallback)(struct __UART_HandleTypeDef *huart, uint16_t
Pos); /*!< UART 接收事件回调 */

    void (* MspInitCallback)(struct __UART_HandleTypeDef
*huart); /*!< UART Msp 初始化回调 */
    void (* MspDeInitCallback)(struct __UART_HandleTypeDef
*huart); /*!< UART Msp 去初始化回调 */
#endif /* USE_HAL_UART_REGISTER_CALLBACKS */

} UART_HandleTypeDef;

```

其中相关寄存器相关结构体如下

```

/**
 * @brief 通用同步异步接收器发射器 (USART)
 */
typedef struct
{
    __IO uint32_t SR;           /*!< USART 状态寄存器, 地址偏移: 0x00 */
    __IO uint32_t DR;           /*!< USART 数据寄存器, 地址偏移: 0x04 */
    __IO uint32_t BRR;          /*!< USART 波特率寄存器, 地址偏移: 0x08 */
    __IO uint32_t CR1;          /*!< USART 控制寄存器 1, 地址偏移: 0x0C */
    __IO uint32_t CR2;          /*!< USART 控制寄存器 2, 地址偏移: 0x10 */
    __IO uint32_t CR3;          /*!< USART 控制寄存器 3, 地址偏移: 0x14 */
    __IO uint32_t GTPR;         /*!< USART 保护时间和预分频器寄存器, 地址偏移:
0x18 */
} USART_TypeDef;

```

其中 UART 通信参数结构体如下

```

/**
 * @brief USART 初始化结构体定义
 */
typedef struct
{
    uint32_t BaudRate;          /*!< 此成员配置 USART 通信的波特率。
    uint32_t WordLength;        /*!< 指定在一帧中传输或接收的数据位数。

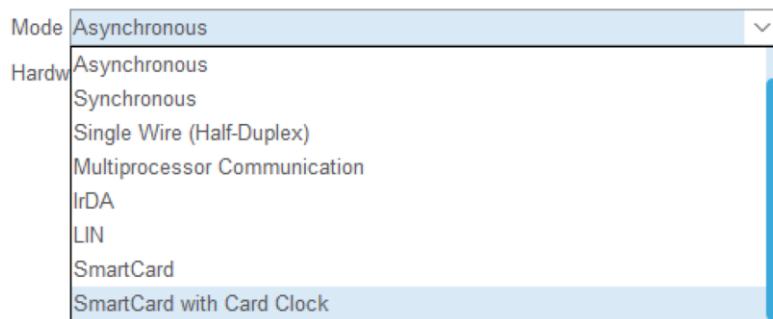
```

```

    uint32_t StopBits;           /*!< 指定传输的停止位数。
    uint32_t Parity;            /*!< 指定奇偶校验模式。
    uint32_t Mode;              /*!< 指定接收或发送模式是否启用或禁用。
    uint32_t HwFlowCtl;         /*!< 指定硬件流控制模式是否启用或禁用。
    uint32_t OverSampling;      /*!< 指定是否启用过采样 8 以实现更高速度
} UART_InitTypeDef;

```

#### 1.16.4 串口通信基础参数



以下是对 STM32 的 CubeMX 串口配置中提及的参数的解释：

##### **异步/同步模式 (Asynchronous /Synchronous):**

这个设置决定了串口是以异步模式还是同步模式工作。在异步模式下，数据传输不依赖于一个共享时钟信号；而在同步模式下，发送方和接收方都依赖于一个共同的时钟信号。

##### **单线（半双工）模式 (Single Wire Half-Duplex):**

单线模式允许数据通过单个线路进行传输，实现半双工通信，即数据在同一时间内只能单向传输，要么发送要么接收。

##### **多处理器通信 (Multiprocessor Communication):**

这个设置用于优化多处理器系统中的串口通信。它允许多个处理器通过同一串口连接进行有效的数据交换。

##### **IrDA 模式:**

IrDA 模式使串口能够与遵循红外数据协会标准的设备进行通信。这通常用于短距离、低功耗的无线通信。

##### **LIN 模式:**

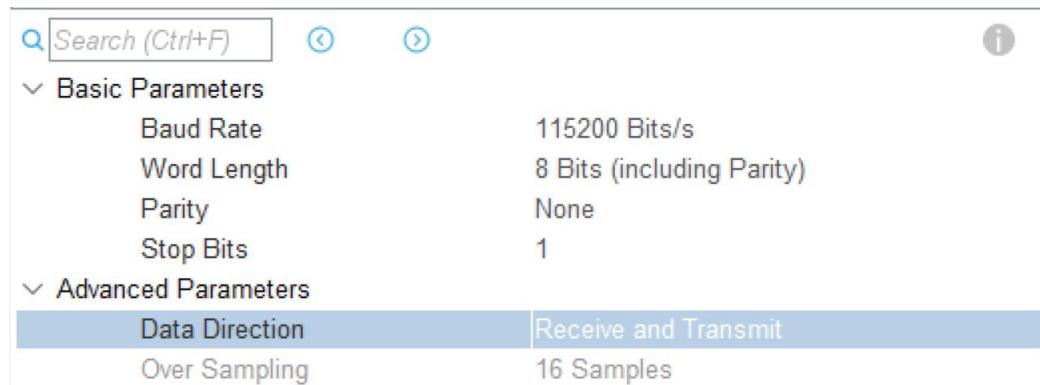
LIN 是一种用于汽车和工业应用的串行网络协议。此设置使串口能够支持 LIN 网络通信，适用于简单的、成本效益高的网络应用。

### 智能卡模式 (SmartCard):

智能卡模式用于与智能卡进行通信。这种模式支持特定的通信协议和电压等级，适用于安全应用，如身份验证和支付系统。

### 带卡时钟的智能卡模式 (SmartCard with Card Clock):

这个设置是智能卡模式的扩展，它提供了一个额外的时钟信号，用于与需要外部时钟源的智能卡进行通信。这有助于同步数据传输和处理。



### 波特率 (Baud Rate):

波特率定义了串口通信的速率，即每秒传输的符号数量。在 UART 通信中，这个参数决定了数据传输的速度。一个较高的波特率意味着更快的数据传输，但也可能增加错误的风险。

### 数据位 (Word Length):

字长决定了串口通信中每个数据包的位数。这可以是 8 位、9 位等，常用的是 8 位。字长的选择取决于通信协议和所需的数据精度。

### 奇偶校验 (Parity):

奇偶校验是一种错误检测机制。它可以是偶校验、奇校验或无校验。在奇校验中，数据字加上校验位后，1 的总数为奇数；在偶校验中，1 的总数为偶数；无校验则不进行这样的检查。

### 停止位 (Stop Bits):

停止位用于标识每个数据包的结束。它可以是 1 位、1.5 位或 2 位。停止位的选择取决于通信协议和硬件要求。

### 数据方向 (Data Direction):

数据方向参数决定了串口通信的流向。这可以是发送（Tx）、接收（Rx）或双向（Tx/Rx）。选择正确的数据方向对于确保通信有效性至关重要。

### 过采样 (Over Sampling):

过采样是一种技术，用于提高数据传输的准确性。它通过在一个波特周期内采集多个样本（例如 8 个或 16 个样本）来实现。这有助于更精确地确定信号的起始位和停止位，从而提高通信的可靠性。

## 1.16.5 串口 IO 相关 API

### 1. HAL\_UART\_Transmit

```
/**  
 * 函数功能：以阻塞模式发送数据。  
 *  
 * HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, const  
 * uint8_t *pData, uint16_t Size, uint32_t Timeout)  
 *  
 * 参数：  
 *   huart - 指向 UART_HandleTypeDef 结构的指针，包含指定 UART 模块的配置信息。  
 *   pData - 指向数据缓冲区的指针（u8 或 u16 数据元素）。  
 *   Size - 要发送的数据元素（u8 或 u16）的数量。  
 *   Timeout - 超时持续时间。  
 * 返回值：HAL 状态。  
 */
```

### 2. HAL\_UART\_Receive

```
/**  
 * 函数功能：以阻塞模式接收数据。  
 *  
 * HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t  
 * *pData, uint16_t Size, uint32_t Timeout)  
 *  
 * 参数：  
 *   huart - 指向 UART_HandleTypeDef 结构的指针，包含指定 UART 模块的配置信息。  
 *   pData - 指向数据缓冲区的指针（u8 或 u16 数据元素）。  
 *   Size - 要接收的数据元素（u8 或 u16）的数量。  
 *   Timeout - 超时持续时间。  
 * 返回值：HAL 状态。  
 */
```

### 3. HAL\_UART\_Transmit\_IT

```
/**  
 * 函数功能：以非阻塞模式发送数据。  
 */
```

```
* HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart, const
uint8_t *pData, uint16_t Size)
*
* 参数:
*   huart - 指向 UART_HandleTypeDef 结构的指针, 包含指定 UART 模块的配置信息。
*   pData - 指向数据缓冲区的指针 (u8 或 u16 数据元素)。
*   Size - 要发送的数据元素 (u8 或 u16) 的数量。
* 返回值: HAL 状态。
*/
```

#### 4. HAL\_UART\_Receive\_IT

```
/***
* 函数功能: 以非阻塞模式接收数据。
*
* HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart,
uint8_t *pData, uint16_t Size)
*
* 参数:
*   huart - 指向 UART_HandleTypeDef 结构的指针, 包含指定 UART 模块的配置信息。
*   pData - 指向数据缓冲区的指针 (u8 或 u16 数据元素)。
*   Size - 要接收的数据元素 (u8 或 u16) 的数量。
* 返回值: HAL 状态。
*/

```

#### 5. HAL\_UART\_Transmit\_DMA

```
/***
* 函数功能: 以 DMA 模式发送数据。
*
* HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart,
const uint8_t *pData, uint16_t Size)
*
* 参数:
*   huart - 指向 UART_HandleTypeDef 结构的指针, 包含指定 UART 模块的配置信息。
*   pData - 指向数据缓冲区的指针 (u8 或 u16 数据元素)。
*   Size - 要发送的数据元素 (u8 或 u16) 的数量。
* 返回值: HAL 状态。
*/

```

#### 6. HAL\_UART\_Receive\_DMA

```
/***
* 函数功能: 以 DMA 模式接收数据。
*
* HAL_StatusTypeDef HAL_UART_Receive_DMA(UART_HandleTypeDef *huart,
uint8_t *pData, uint16_t Size)
```

```
/*
 * 参数:
 *     huart - 指向 UART_HandleTypeDef 结构的指针, 包含指定 UART 模块的配置信息。
 *     pData - 指向数据缓冲区的指针 (u8 或 u16 数据元素)。
 *     Size - 要接收的数据元素 (u8 或 u16) 的数量。
 * 返回值: HAL 状态。
 */
```

## 7. HAL\_UART\_DMAPause

```
/**
 * 函数功能: 暂停 DMA 传输。
 *
 * HAL_StatusTypeDef HAL_UART_DMAPause(UART_HandleTypeDef *huart)
 *
 * 参数:
 *     huart - 指向 UART_HandleTypeDef 结构的指针, 包含指定 UART 模块的配置信息。
 * 返回值: HAL 状态。
 */
```

## 8. HAL\_UART\_DMAResume

```
/**
 * 函数功能: 恢复 DMA 传输。
 *
 * HAL_StatusTypeDef HAL_UART_DMAResume(UART_HandleTypeDef *huart)
 *
 * 参数:
 *     huart - 指向 UART_HandleTypeDef 结构的指针, 包含指定 UART 模块的配置信息。
 * 返回值: HAL 状态。
 */
```

## 9. HAL\_UART\_DMAStop

```
/**
 * 函数功能: 停止 DMA 传输。
 *
 * HAL_StatusTypeDef HAL_UART_DMAStop(UART_HandleTypeDef *huart)
 *
 * 参数:
 *     huart - 指向 UART_HandleTypeDef 结构的指针, 包含指定 UART 模块的配置信息。
 * 返回值: HAL 状态。
 */
```

## 10. HAL\_UARTEx\_ReceiveToIdle

```
/**
 * 函数功能: 以阻塞模式接收数据, 直到接收到指定数量的数据或发生 IDLE 事件。
```

```
/*
 * HAL_StatusTypeDef HAL_UARTEx_ReceiveToIdle(UART_HandleTypeDef *huart,
 * uint8_t *pData, uint16_t Size, uint16_t *RxLen, uint32_t Timeout)
 *
 * 参数:
 *   huart - UART 句柄。
 *   pData - 指向数据缓冲区的指针（uint8_t 或 uint16_t 数据元素）。
 *   Size - 要接收的数据元素（uint8_t 或 uint16_t）的数量。
 *   RxLen - 最终接收的数据元素数量。
 *   Timeout - 超时持续时间（以 ms 为单位）。
 * 返回值: HAL 状态。
 */
```

## 11. HAL\_UARTEx\_ReceiveToIdle\_IT

```
/**
 * 函数功能: 以中断模式接收数据, 直到接收到指定数量的数据或发生 IDLE 事件。
 *
 * HAL_StatusTypeDef HAL_UARTEx_ReceiveToIdle_IT(UART_HandleTypeDef *huart,
 * uint8_t *pData, uint16_t Size)
 *
 * 参数:
 *   huart - UART 句柄。
 *   pData - 指向数据缓冲区的指针（uint8_t 或 uint16_t 数据元素）。
 *   Size - 要接收的数据元素（uint8_t 或 uint16_t）的数量。
 * 返回值: HAL 状态。
 */
```

## 12. HAL\_UARTEx\_ReceiveToIdle\_DMA

```
/**
 * 函数功能: 以 DMA 模式接收数据, 直到接收到指定数量的数据或发生 IDLE 事件。
 *
 * HAL_StatusTypeDef HAL_UARTEx_ReceiveToIdle_DMA(UART_HandleTypeDef
 * *huart, uint8_t *pData, uint16_t Size)
 *
 * 参数:
 *   huart - UART 句柄。
 *   pData - 指向数据缓冲区的指针（uint8_t 或 uint16_t 数据元素）。
 *   Size - 要接收的数据元素（uint8_t 或 uint16_t）的数量。
 * 返回值: HAL 状态。
 */
```

## 13. HAL\_UARTEx\_GetRxEventType

```
/**
 * 函数功能: 获取导致 RxEvent 回调执行的 Rx 事件类型。
```

```
*  
* HAL_UART_RxEventTypeTypeDef  
HAL_UARTEEx_GetRxEventType(UART_HandleTypeDef *huart)  
*  
* 参数:  
*     huart - UART 句柄。  
* 返回值: Rx 事件类型 (将返回@ref UART_RxEvent_Type_Values 中的一个值)。  
*/
```

## 14. HAL\_UART\_Abort

```
/**  
* 函数功能: 中断模式下中止正在进行的传输。  
*  
* HAL_StatusTypeDef HAL_UART_Abort(UART_HandleTypeDef *huart)  
*  
* 参数:  
*     huart - UART 句柄。  
* 返回值: HAL 状态。  
*/
```

## 15. HAL\_UART\_AbortTransmit

```
/**  
* 函数功能: 中断模式下中止正在进行的发送传输。  
*  
* HAL_StatusTypeDef HAL_UART_AbortTransmit(UART_HandleTypeDef *huart)  
*  
* 参数:  
*     huart - UART 句柄。  
* 返回值: HAL 状态。  
*/
```

## 16. HAL\_UART\_AbortReceive

```
/**  
* 函数功能: 中断模式下中止正在进行的接收传输。  
*  
* HAL_StatusTypeDef HAL_UART_AbortReceive(UART_HandleTypeDef *huart)  
*  
* 参数:  
*     huart - UART 句柄。  
* 返回值: HAL 状态。  
*/
```

## 17. HAL\_UART\_Abort\_IT

```
/**  
 * 函数功能：中断模式下中止正在进行的传输。  
 *  
 * HAL_StatusTypeDef HAL_UART_Abort_IT(UART_HandleTypeDef *huart)  
 *  
 * 参数：  
 *   huart - UART 句柄。  
 * 返回值：HAL 状态。  
 */
```

## 18. HAL\_UART\_AbortTransmit\_IT

```
/**  
 * 函数功能：中断模式下中止正在进行的发送传输。  
 *  
 * HAL_StatusTypeDef HAL_UART_AbortTransmit_IT(UART_HandleTypeDef *huart)  
 *  
 * 参数：  
 *   huart - UART 句柄。  
 * 返回值：HAL 状态。  
 */
```

## 19. HAL\_UART\_AbortReceive\_IT

```
/**  
 * 函数功能：中断模式下中止正在进行的接收传输。  
 *  
 * HAL_StatusTypeDef HAL_UART_AbortReceive_IT(UART_HandleTypeDef *huart)  
 *  
 * 参数：  
 *   huart - UART 句柄。  
 * 说明：  
 *   此过程可用于中止任何在中断或 DMA 模式下启动的正在进行的 Rx 传输。  
 *   此过程执行以下操作：  
 *     - 禁用 UART 中断（Rx）。  
 *     - 如果启用，禁用外设寄存器中的 DMA 传输。  
 *     - 如果在 DMA 模式下进行传输，通过调用 HAL_DMA_Abort_IT 中止 DMA 传输。  
 *     - 将句柄状态设置为 READY。  
 *     - 在中止完成时，调用用户中止完成回调。  
 *   此过程在中断模式下执行，这意味着只有当用户中止完成回调执行时，中止过程才被视为完成（不是在退出函数时）。  
 *   返回值：HAL 状态。  
 */
```

## 20. HAL\_UART\_IRQHandler

```
/**
```

```
/* 函数功能：处理 UART 中断请求。
*
* void HAL_UART_IRQHandler(UART_HandleTypeDef *huart)
*
* 参数：
*     huart - 指向包含指定 UART 模块配置信息的 UART_HandleTypeDef 结构的指针。
* 返回值：无。
*/

```

## 21. HAL\_UART\_TxCpltCallback

```
/** 
* 函数功能：Tx 传输完成回调。
*
* void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
*
* 参数：
*     huart - 指向包含指定 UART 模块配置信息的 UART_HandleTypeDef 结构的指针。
* 返回值：无。
*/

```

## 22. HAL\_UART\_TxHalfCpltCallback

```
/** 
* 函数功能：Tx 半传输完成回调。
*
* void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart)
*
* 参数：
*     huart - 指向包含指定 UART 模块配置信息的 UART_HandleTypeDef 结构的指针。
* 返回值：无。
*/

```

## 23. HAL\_UART\_RxCpltCallback

```
/** 
* 函数功能：Rx 传输完成回调。
*
* void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
*
* 参数：
*     huart - 指向包含指定 UART 模块配置信息的 UART_HandleTypeDef 结构的指针。
* 返回值：无。
*/

```

## 24. HAL\_UART\_RxHalfCpltCallback

```
/**
```

```
/* 函数功能: Rx 半传输完成回调。
*
* void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart)
*
* 参数:
*   huart - 指向包含指定 UART 模块配置信息的 UART_HandleTypeDef 结构的指针。
* 返回值: 无。
*/

```

## 25. HAL\_UART\_ErrorCallback

```
/** 
* 函数功能: UART 错误回调。
*
* void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart)
*
* 参数:
*   huart - 指向包含指定 UART 模块配置信息的 UART_HandleTypeDef 结构的指针。
* 返回值: 无。
*/

```

## 26. HAL\_UART\_AbortCpltCallback

```
/** 
* 函数功能: UART 中止完成回调。
*
* void HAL_UART_AbortCpltCallback(UART_HandleTypeDef *huart)
*
* 参数:
*   huart - UART 句柄。
* 返回值: 无。
*/

```

## 27. HAL\_UART\_AbortTransmitCpltCallback

```
/** 
* 函数功能: UART 中止发送完成回调。
*
* void HAL_UART_AbortTransmitCpltCallback(UART_HandleTypeDef *huart)
*
* 参数:
*   huart - UART 句柄。
* 返回值: 无。
*/

```

## 28. HAL\_UART\_AbortReceiveCpltCallback

```
/**
```

```
* 函数功能: UART 中止接收完成回调。  
*  
* void HAL_UART_AbortReceiveCpltCallback(UART_HandleTypeDef *huart)  
*  
* 参数:  
*     huart - UART 句柄。  
* 返回值: 无。  
*/
```

## 29. HAL\_UARTEEx\_RxEventCallback

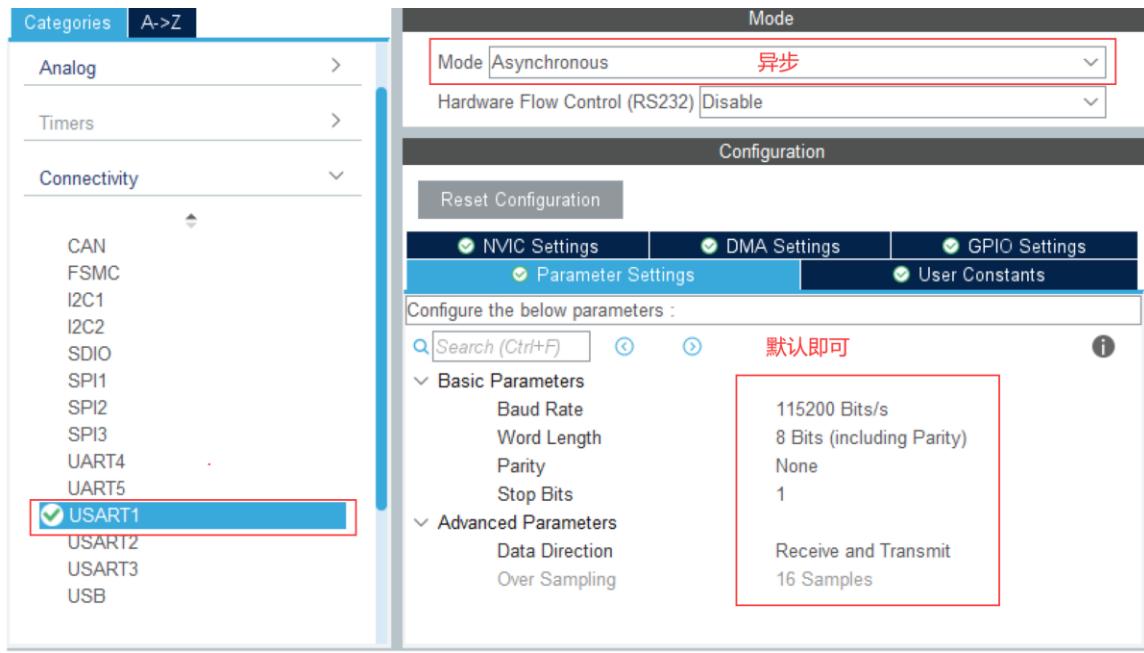
```
/**  
* 函数功能: 接收事件回调 (在使用高级接收服务后调用的 Rx 事件通知)。  
*  
* void HAL_UARTEEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t  
Size)  
*  
* 参数:  
*     huart - UART 句柄。  
*     Size - 应用接收缓冲区中可用的数据数量 (指示接收缓冲区中可用数据的位置)。  
* 返回值: 无。  
*/
```

### 1.16.6 串口轮询模式实现回显

串口轮询模式，亦称为阻塞模式，是一种在数据传输和接收过程中应用程序需等待的简单设备间通信方法。在此模式中，UART（通用异步接收/发送器）通信通过逐比特发送数据来实现，即每个数据字节会被分解成独立的比特。当 UART 发射器接收到一个数据字节时，它会将该字节转换成一系列比特并进行发送。发送完一个字节后，UART 发射器便进入空闲状态，此时可以继续发送下一个字节。

UART 接收器在轮询模式下的工作原理与发射器类似。当接收器接收到一个完整的数据字节并检测到停止位时，它会将这个数据字节存储在 UART 数据寄存器中，并生成一个接收标志。这个接收标志可以被不断轮询以检索接收到的字节。

串口 1 CubeMX 配置



## 生成工程后串口配置函数分析

```

void MX_USART1_UART_Init(void)
{
    /* USER CODE BEGIN USART1_Init_0 */
    // 这里可以放置用户自定义的初始代码

    /* USER CODE END USART1_Init_0 */

    /* USER CODE BEGIN USART1_Init_1 */
    // 这里也可以放置用户自定义的初始代码

    /* USER CODE END USART1_Init_1 */

    // 设置 USART1 为该函数的实例
    huart1.Instance = USART1;

    // 设置波特率为 115200
    huart1.Init.BaudRate = 115200;

    // 设置字长为 8 位
    huart1.Init.WordLength = UART_WORDLENGTH_8B;

    // 设置停止位为 1 个停止位
    huart1.Init.StopBits = UART_STOPBITS_1;

    // 设置无奇偶校验
    huart1.Init.Parity = UART_PARITY_NONE;
}

```

```

// 设置模式为发送和接收
huart1.Init.Mode = UART_MODE_TX_RX;

// 设置无硬件流控制
huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;

// 设置过采样为 16
huart1.Init.OverSampling = UART_OVERSAMPLING_16;

// 初始化配置的 UART，并在初始化失败时调用错误处理函数
if (HAL_UART_Init(&huart1) != HAL_OK)
{
    Error_Handler();
}

/* USER CODE BEGIN USART1_Init_2 */
// 这里可以放置用户自定义的额外初始化代码

/* USER CODE END USART1_Init_2 */
}

```

对于阻塞模式，只需在主循环中阻塞接收即可

定义变量存储接收到的 UART 数据。

```

/* USER CODE BEGIN PV */
// 定义并初始化一个 uint8_t 类型的变量 data，初始值为 0。这个变量将用于存储接收到的 UART 数据。
uint8_t data=0;
/* USER CODE END PV */

```

在主循环阻塞接收

```

/* USER CODE BEGIN WHILE */
while (1)
{
    // 尝试从 USART1 接收一个字节的数据，将其存储在 data 中。如果接收成功（即
    HAL_UART_Receive 返回 HAL_OK），则执行 if 语句块。
    if(HAL_UART_Receive(&huart1,&data,1,0)==HAL_OK){
        // 将接收的字符发送至串口 1
        HAL_UART_Transmit(&huart1,&data,1,0);
    }
    /* USER CODE BEGIN 3 */

    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */

```

实验效果：

## 使用串口线连接单片机



打开串口调试助手，配置波特率等参数，发送任意字符



### 1.16.7 串口中断模式控制 LED

在嵌入式系统中，串口中断模式与阻塞模式各有其优势和劣势。中断模式的主要优势在于其高效的资源利用和更好的实时响应能力。在中断模式下，CPU 可以在没有数据传输时执行其他任务，只有在必要的数据到来时才会中断当前任务，处理串口事件，这种方式显著提高了系统的整体效率和响应速度。

#### CubeMX 配置

在上述阻塞模式的基础上，打开串口中断及 PE5（LED 灯）

NVIC Settings	DMA Settings	GPIO Settings	
Parameter Settings		User Constants	
NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
USART1 global interrupt	<input checked="" type="checkbox"/>	0	0

工程生成后，在 HAL\_UART\_MspInit 函数中已经开启了 UART1 中断

```
/* USART1 interrupt Init */
HAL_NVIC_SetPriority(USART1_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(USART1_IRQn);
/* USER CODE BEGIN USART1_MspInit 1 */
```

在 main.c 中，需要重写接收回调函数，实现控制 LED

定义宏

```
/* USER CODE BEGIN PTD */
#define USART_REC_LEN 256 // 定义接收数据的最大长度
/* USER CODE END PTD */
```

定义变量

```
/* USER CODE BEGIN PV */
uint16_t USART_RX_STA = 0; // 定义 USART 接收状态标志
char USART_RX_BUF[USART_REC_LEN]; // 定义 USART 接收缓冲
uint8_t rxData; // 用于单字节接收
/* USER CODE END PV */
```

重写接收回调函数

```
/* USER CODE BEGIN 4 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance == USART1)
    {
        if((USART_RX_STA & 0x8000) == 0) // 未完成接收
        {
            if(USART_RX_STA & 0x4000) // 已经接收到 0xd
            {
                if(rxData != 0xa)
                {
                    USART_RX_STA = 0; // 接收错误，重新开始
                }
                else
                {
                    USART_RX_STA |= 0x8000; // 标记接收完成
                }
            }
            else
            {
                if(rxData == 0xd)
                {
                    USART_RX_STA |= 0x4000; // 接收到 0xd，等待 0xa
                }
            }
        }
    }
}
```

```

        else
        {
            USART_RX_BUF[USART_RX_STA & 0X3FFF] = rxData;
            USART_RX_STA++;
            if(USART_RX_STA > (USART_REC_LEN - 1)) USART_RX_STA = 0; // 如果接收数据超出缓冲长度, 重新开始
        }
    }

    // 如果接收数据完成
    if (USART_RX_STA & 0x8000)
    {
        // LED 控制逻辑
        if (strncmp((char *)USART_RX_BUF, "ON", 2) == 0)
        {
            HAL_GPIO_WritePin(GPIOE, GPIO_PIN_5, GPIO_PIN_RESET); // 低电平亮
        }
        else if (strncmp((char *)USART_RX_BUF, "OFF", 3) == 0)
        {
            HAL_GPIO_WritePin(GPIOE, GPIO_PIN_5, GPIO_PIN_SET); // 高电平熄灭
        }

        HAL_UART_Transmit(&huart1, (uint8_t*)USART_RX_BUF,
USART_RX_STA & 0X3FFF, 1000); // 1000ms
        USART_RX_STA = 0; // 重置状态, 准备下次接收
    }

    // 重新启动异步接收, 接收一个字节
    HAL_UART_Receive_IT(huart, &rxData, 1);
}
/* USER CODE END 4 */

```

实验效果如下：

当输入 ON\r\n 时，LED 亮，同时回显。

当输入 OFF\r\n 时，LED 灭，同时回显。

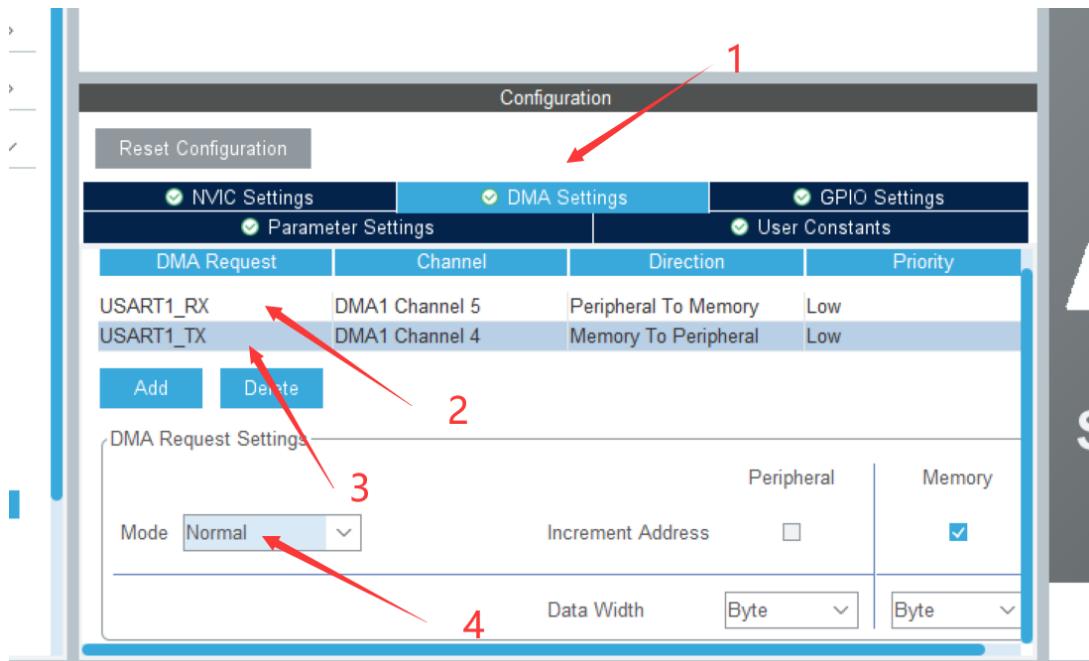


### 1.16.8 串口 DMA 模式控制舵机

在嵌入式系统中，使用 DMA（Direct Memory Access）模式进行串口通信，尤其是在控制诸如舵机这类对实时性要求较高的应用时，与阻塞模式和中断模式相比具有其独特的优势和劣势。DMA 模式的最大优势在于其高效的数据传输能力和较低的 CPU 占用率。在 DMA 模式下，数据直接在外设和内存之间传输，无需 CPU 干预，这使得 CPU 能够同时处理其他任务，极大提高了系统的并行处理能力。这对于舵机控制这样需要快速且连续数据流的应用来说，能够提供更流畅和稳定的控制。

#### CubeMX 配置

在上述基础上，需要打开 TIM3\_CH3，配置具体见 1.14。同时串口配置 DMA 相关。



生成工程后，在 main.c 中

定义宏

```
/* USER CODE BEGIN PTD */
#define uart_rx_buffer_size 256 //uart1 接收缓冲区长度
/* USER CODE END PTD */
```

定义变量

```
/* USER CODE BEGIN PV */
uint8_t uart_rx_buffer[uart_rx_buffer_size];// uart1 接收缓冲区
/* USER CODE END PV */
```

开启 DMA 模式接收数据，直到接收到指定数量的数据或发生 IDLE 事件(空闲)。  
当 USART 接口在数据帧之间检测到空闲线（即在一段时间内没有接收到新的数据）时，就会生成空闲中断。USART 接口能识别出来空闲线，它会设置 USART\_SR 寄存器的 IDLE 位并产生一个中断。

```
/* USER CODE BEGIN 2 */
HAL_UARTEx_ReceiveToIdle_DMA(&huart1, uart_rx_buffer,
uart_rx_buffer_size);
/* USER CODE END 2 */
```

重写中断回调函数

```
/* USER CODE BEGIN 4 */
void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size)
{
```

```

if (huart->Instance == USART1)
{
    // 执行回显操作
    HAL_UART_Transmit(&huart1, uart_rx_buffer, Size, 1000);

    // 确保接收的数据以空字符结尾
    uart_rx_buffer[Size] = '\0';

    // 检查接收到的命令
    if (strcmp((char *)uart_rx_buffer, "ON") == 0)
    {
        // 打开舵机
        HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3); // 启动 PWM;
    }
    else if (strcmp((char *)uart_rx_buffer, "OFF") == 0)
    {
        // 关闭舵机
        HAL_TIM_PWM_Stop(&htim3, TIM_CHANNEL_3);
    }

    // 重新启动 DMA 接收
    HAL_UARTEx_ReceiveToIdle_DMA(&huart1, uart_rx_buffer,
        uart_rx_buffer_size);
}
/* USER CODE END 4 */

```

实验效果：

当输入 ON 时，舵机启动，同时回显。

当输入 OFF 时，舵机关闭，同时回显。

### 1.16.9 作业

串口输入"speed:-100~100"时，控制舵机转速，同时回显，当输入"switch:ON"时，舵机启动，同时回显。当输入" switch:OFF"时，舵机关闭，同时回显。

## 1.17.1 单总线通信

### 1.17.1 单总线通信概述

单总线通信与 SPI、I2C 等串行数据通信方式不同，因为它仅使用单根信号线进行数据传输。这种通信方式的数据传输是双向的，既能传输时钟信号，也能传输数据。单总线通信因其简单的结构、低成本以及易于维护等优点而被广泛应用于各种系统中，特别适用于单个主机系统，在使用 DS18B20 这种单总线温度传感器时，整个

通信过程包括初始化（复位/应答）、写字节、读字节、计算温度等步骤。在初始化阶段，必须确保主机在单总线上允许强上拉，并且在此期间不能进行其他数据传输。写入和读取数据时也有严格的时隙要求，例如写 0 时需要保持总线低电平在 60-120 微秒之间，而写 1 时需要在拉低总线 15 微秒之内将总线拉高，并保持 60 微秒以上。

### 1.17.2 DHT11 概述

DHT11 数字温湿度传感器是一款含有已校准数字信号输出的温湿度复合传感器，它应用专用的数字模块采集技术和温湿度传感技术，确保产品具有极高的可靠性和卓越的长期稳定性。传感器包括一个电阻式感湿元件和一个 NTC 测温元件，并与一个高性能 8 位单片机相连接。因此该产品具有品质卓越、超快响应、抗干扰能力强、性价比极高等优点。每个 DHT11 传感器都在极为精确的湿度校验室中进行校准。校准系数以程序的形式存在 OTP 内存中，传感器内部在检测信号的处理过程中要调用这些校准系数。单线制串行接口，使系统集成变得简易快捷。超小的体积、极低的功耗，使其成为该类应用中的最佳选择。

### 1.17.3 DHT11 时序

采用单总线双向串行通信协议，每次采集都要由单片机发起开始信号，然后 DHT11 会向单片机发送响应并开始传输 40 位数据帧，高位在前。数据格式为：8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit 温度小数数据+8bit 校验位，温湿度小数部分默认为 0，即单片机采集的数据都是整数，校验位为 4 个字节的数据相加取结果的低 8 位数据作为校验和：

示例一：接收到的 40 位数据为：

<u>0011 0101</u>	<u>0000 0000</u>	<u>0001 1000</u>	<u>0000 0000</u>	<u>0100 1101</u>
湿度高 8 位	湿度低 8 位	温度高 8 位	温度低 8 位	校验位

计算：

$0011\ 0101 + 0000\ 0000 + 0001\ 1000 + 0000\ 0000 = 0100\ 1101$

接收数据正确：

湿度：0011 0101=35H=53%RH

温度：0001 1000=18H=24°C

示例二：接收到的 40 位数据为：

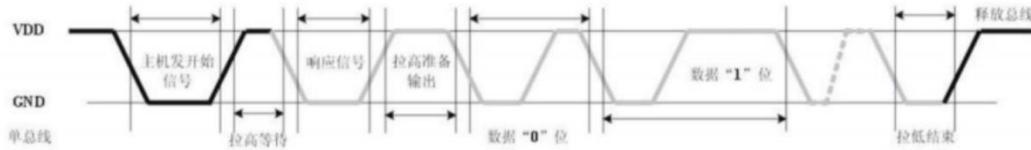
<u>0011 0101</u>	<u>0000 0000</u>	<u>0001 1000</u>	<u>0000 0000</u>	<u>0100 1001</u>
湿度高 8 位	湿度低 8 位	温度高 8 位	温度低 8 位	校验位

计算：

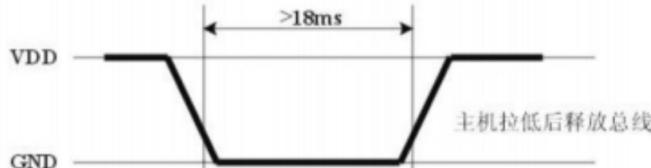
$0011\ 0101 + 0000\ 0000 + 0001\ 1000 + 0000\ 0000 = 0100\ 1101$

01001101 不等于 0100 1001

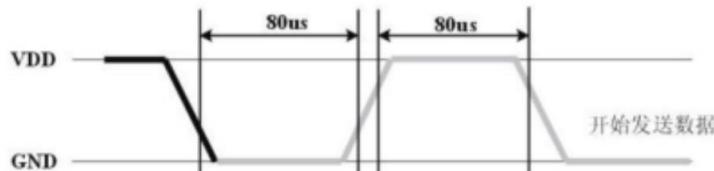
具体时序图如下：



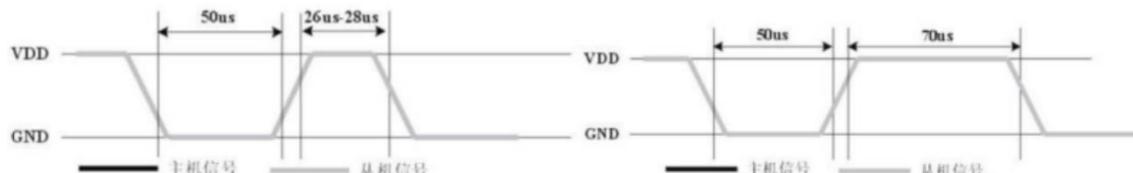
主机或者单片机需要发送一个开始信号给 DHT11 传感器：主机先将 IO 设置为输出，然后主机拉低总线（时间需要大于 18ms）后设置为输入并释放数据总线，等待从机（DHT11）响应，主机开始的信号时序为：



如果传感器正常且存在，则会在收到主机的开始信号后拉低总线并持续 80us 来通知主机此时传感器正常，然后拉高总线 80us，通知主机准备接收，响应的时序为：



接着传感器开始按照高位在前的顺序将数据按照如下的格式，一位一位的输出给主机：



程序要区分数据 0 和数据 1 的格式：先判断此时引脚的电平状态，如果是低电平就一直循环等待，直到高电平出现，高电平出现后延时 40us，并读取延时后的电平状态，如果此时是高电平，则数据为 1，否则为 0。

传输完 40 位数据后，传感器再次输出一个 50us 的低电平后，将数据总线释放，采集过程结束。

#### 1.17.4 读取 DHT11 数据并串口显示

1.Cubemx 确定数据引脚，配置 UART1，TIM1。

2.添加并编写 DHT11 驱动文件

DHT11.c

```
#include "dht11.h"
```

```
uint8_t Data[5]={0x00,0x00,0x00,0x00,0x00};

//将 GPIO 调节为输入模式
void SET_PIN_INPUT(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    GPIO_InitStruct.Pin = GPIO_PIN_5;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

//将 GPIO 调节为输出模式
void SET_PIN_OUTPUT(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    GPIO_InitStruct.Pin = GPIO_PIN_5;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

//微秒级延时函数(向下)
void Delay_us(uint16_t us)
{
    uint16_t dif=10000;
    uint16_t us_con=10000-us;
    __HAL_TIM_SET_COUNTER(&htim1,10000); //设置计数值为 10000
    HAL_TIM_Base_Start(&htim1); //启动定时器
    while(dif>us_con)
    {
        dif=__HAL_TIM_GET_COUNTER(&htim1); //获取定时器的计数值
    }
    HAL_TIM_Base_Stop(&htim1); //停止定时器
}

//电平读取函数
uint8_t DHT_read_byte(void)
{
    uint8_t read_byte=0;//存放 8bit 的高低电平数据
    uint8_t temp;//存放读取到的高低电平
```

```

    uint8_t res=0;
    uint8_t i;
    for(i=0;i<8;i++)
    {
        //等待高电平到来
        while(HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_5)==0&&res<100)
        {
            Delay_us(1); //1US 读取一次
            res++; //防止卡死
        }
        res=0;
        Delay_us(40);
        if(HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_5)==1)
        {
            temp=1;
        }
        else temp=0;
        //等待低电平到来，开启下一次读取或读取结束
        while(HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_5)==1&&res<100)
        {
            Delay_us(1); //1US 读取一次
            res++; //防止卡死
        }
        res=0;
        read_byte<<=1;//左移一位，空出末尾
        read_byte |= temp;
    }
    return read_byte;
}

//DHT11 读取温湿度程序
uint8_t DHT_reade(void)
{
    uint8_t res=0;
    uint8_t i;
    //首先发送 18ms 低电平
    SET_PIN_OUTPUT();
    HAL_GPIO_WritePin(GPIOA,GPIO_PIN_5,GPIO_PIN_RESET);
    HAL_Delay(18);
    //拉高 20us 高电平等待
    HAL_GPIO_WritePin(GPIOA,GPIO_PIN_5,GPIO_PIN_SET);
    Delay_us(20);
    //32 向 DHT11 发送请求完成，之后等待接收消息

    //IO 变为输入模式
}

```

```

SET_PIN_INPUT();
Delay_us(20);
//开始检测
//如果检测到低电平，说明 DHT11 有响应
if(HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_5)==0)
{
    while(HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_5)==0&&res<100)
    {
        Delay_us(1); //1US 读取一次
        res++; //防止卡死
    }
    res=0;
    while(HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_5)==1&&res<100)
    {
        Delay_us(1); //1US 读取一次
        res++; //防止卡死
    }
    res=0;
    //反转电平过后，DHT11 开始传输数据
    for(i=0;i<5;i++)
    {
        Data[i]=DHT_read_byte();
    }
    Delay_us(50);
}
//校验
uint32_t sum=Data[0]+Data[1]+Data[2]+Data[3];
if((sum)==Data[4])
{
    return 1;
}
else return 0;
}

```

## DHT11.h

```

#ifndef __DHT11_H
#define __DHT11_H
#include "stm32f1xx.h"//包含 hal 库
#include "tim.h"//包含定时器

void SET_PIN_INPUT(void);//将 GPIO 调节为输入模式
void SET_PIN_OUTPUT(void) ;//将 GPIO 调节为输出模式
void Delay_us(uint16_t us);//微秒级延时
uint8_t DHT_read_byte(void);//电平读取函数

```

```
uint8_t DHT_reade(void);

#endif
```

在 main.c 中编写主循环

```
while (1)
{
    if(DHT_reade())
    {
        //组包，并准备发送
        len=sprintf((char
*)message,sizeof(message),"Humi:%d%%,Temp:%d\r\n",Data[0],Data[2]);
        //通过串口发送
        HAL_UART_Transmit(&huart1,message,len,1000);
    }
    else
    {
        const char*error_msg="DHT11 fail\r\n";
        HAL_UART_Transmit(&huart1,(uint8_t
*)error_msg,strlen(error_msg),1000);
    }
    HAL_Delay(2000);
/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
}
```

3.实验现象：

```
[2023-12-05 10:35:43.715]# RECV ASCII>
Humi:48%, Temp:23

[2023-12-05 10:35:45.737]# RECV ASCII>
Humi:48%, Temp:23

[2023-12-05 10:35:47.757]# RECV ASCII>
Humi:48%, Temp:23
```

## 1.18 I2C 通信

### 1.18.1 I2C 通信概述

I2C（Inter-Integrated Circuit）通信是一种由 Philips 公司开发的串行通信协议，主要用于近距离、低速的芯片间通信。它只需要两条双向的信号线，一条数据线

SDA（用于收发数据）和一条时钟线 SCL（用于通信双方时钟的同步）。I2C 通信是半双工通信，意味着数据在一条数据线上传输，且同一时刻只能单向传输，要么从主机到从机，要么从从机到主机。每个数据包的大小为 8 位，采用高位在前、低位在后的传输顺序。

I2C 通信中，主机通过时钟线 SCL 发送时钟信号，并通过数据线 SDA 发送数据（如从机地址、指令、数据包等）。在发送完一帧数据后，主机需等待从机的响应才能继续发送下一帧数据。I2C 总线的操作时序包括写单个存储字节、写多个存储字节、读单个存储字节和读多个存储字节等不同模式。

I2C 总线支持多主机模式，即可以有多个主机设备在总线上发起通信。在多主机系统中，仲裁过程是必要的，以避免冲突。当两个主机同时发送数据时，谁先发送高电平的将会输掉仲裁，即仲裁失败的主机将不再产生时钟脉冲，并需等待总线空闲时才能重新传输。

I2C 通信中，每个从机都有一个唯一的 7 位或 10 位地址，用于区分不同的从机设备。I2C 的标准传输速率有多种模式，包括标准模式（100 kbit/s）、快速模式（400 kbit/s）、快速+模式（1 Mbit/s）等，适用于不同的应用场景。

### 1.18.2 ssd1306 屏幕概述

SSD1306 是一种广泛用于 OLED（Organic Light-Emitting Diode）屏幕的控制器。它支持多种通信接口，包括 I2C、SPI 和并行接口，但在许多应用中，特别是小型设备和微控制器项目中，常通过 I2C 接口与 SSD1306 进行通信。这种屏幕控制器特别适合于需要小型、低功耗显示屏的应用，如可穿戴设备、便携式仪器等。

SSD1306 控制器能够驱动多达 128x64 像素的显示屏，提供了丰富的图形处理功能，包括点、线、矩形以及字符的绘制。通过编程，可以在 OLED 屏幕上显示文本、图形和动画。SSD1306 的低功耗特性和灵活的通信接口使其成为小型显示项目的热门选择。

### 1.18.3 ssd1306 I2C 通信分析

ssd1306 的 I2C 通信接口由从机地址位 SA0、I2C 总线数据信号、SDA（SDAOUT/D2 用于输出，SDAIN/D1 用于输入）和 I2C 总线时钟信号 SCL (D0) 组成。数据和时钟信号都必须连接到上拉电阻。

**从设备地址：** SSD1306 在通过 I2C 总线传输或接收信息前必须识别从设备地址。设备响应从设地址，后跟从设地址位(SA0)和读/写选择位(R/W#)。

b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>
0	1	1	1	1	0	SA0	R/W#

SA0 位是从设地址的扩展位。可以选择“0111100”或“0111101”作为 SSD1306 的从设地址。D/C#引脚充当 SA0 进行从设地址选择。

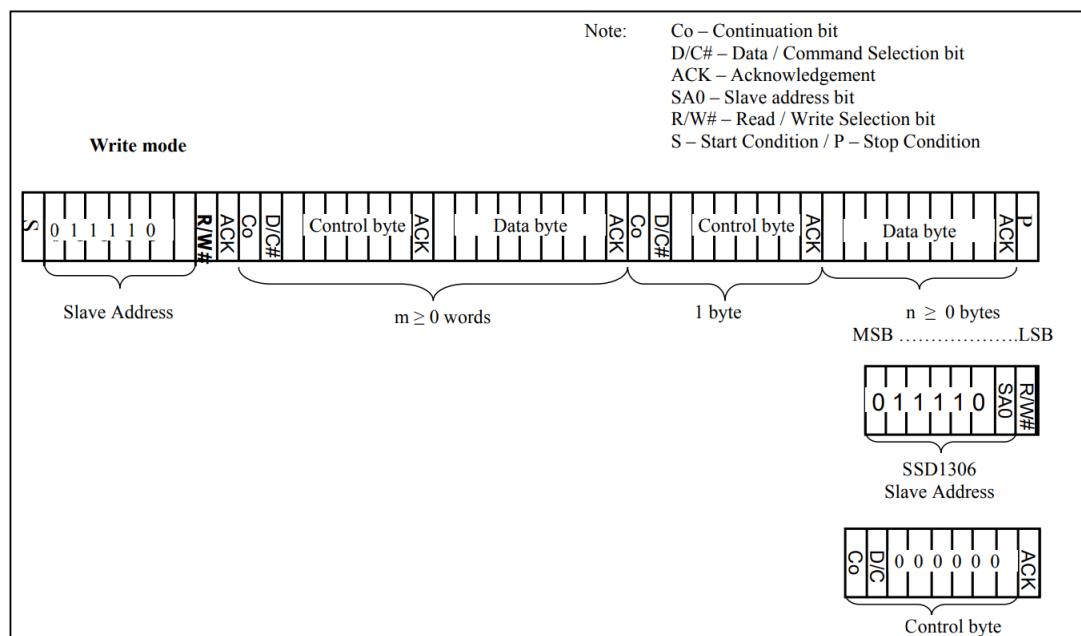
R/W#位用于确定 I2C 总线接口的操作模式，R/W#=1 时为读模式，R/W#=0 时为写模式。

如 SA0 位为 0，R/W#=0，此时 I2C 通信设备通信地址为 01111000，即 0x78。

**I2C 数据信号(SDA):** SDA 用作发送者和接收者之间的通信通道。数据和确认信号通过 SDA 发送。SDA 用作通信通道。需要注意的是，ITO 轨道电阻和 SDA 引脚上的上拉电阻会形成电压势分器，这可能导致无法在 SDA 上获得有效的逻辑 0 级别的确认信号。SDAIN 和 SDAOUT 引脚被绑定在一起，作为 SDA 使用。SDAIN 引脚必须连接以充当 SDA，SDAOUT 引脚可以不连接。当 SDAOUT 引脚断开时，I2C 总线中的确认信号将被忽略。

**I2C 时钟信号(SCL):** 在 I2C 总线中，信息的传输遵循时钟信号 SCL。每个数据位的

I2C 总线接口允许将数据和命令写入设备。如下图所示为：I2C 总线的写入模式按时间顺序排列。



#### 1.18.4 ssd1306 I2C 写入程序的实现

ssd1306 I2C 写入需注意：

**启动条件：** 主设备通过启动条件开始数据通信。启动条件是在时钟线 SCL 保持高电平的情况下，数据线 SDA 从高电平拉低到低电平。

**从设备地址：**启动条件后，会发送从设备地址以供识别。对于 SSD1306，从设备地址可以是“b0111100”或“b0111101”，这取决于 SA0 (D/C 引脚作为 SA0) 是低电平还是高电平。

**写模式：**通过将 R/W#位设置为逻辑“0”来建立写模式。

**确认信号：**接收一个数据字节（包括从设备地址和 R/W#位）后，会产生一个确认信号。确认位是在相关时钟脉冲的高电平期间，SDA 线被拉低定义的。

**控制字节或数据字节的传输：**传输从设备地址后，可以传送控制字节或数据字节。

如果 Co 位设置为逻辑“0”，随后的信息传输将只包含数据字节。

D/C#位决定下一个数据字节是作为命令还是数据。如果 D/C#位设置为逻辑“0”，则定义接下来的数据字节为命令。如果 D/C#位设置为逻辑“1”，则定义接下来的数据字节为数据，这些数据将存储在 GDDRAM 中。每次数据写入后，GDDRAM 列地址指针自动增加。

**每个控制字节或数据字节后的确认位：**接收每个控制字节或数据字节后，都会生成确认位。

**写模式的结束：**当应用停止条件时，写模式结束。停止条件是在时钟线 SCL 保持高电平的情况下，将“SDA in”从低电平拉高到高电平来建立。

具体时序图如下：

Figure 8-8 : Definition of the Start and Stop Condition

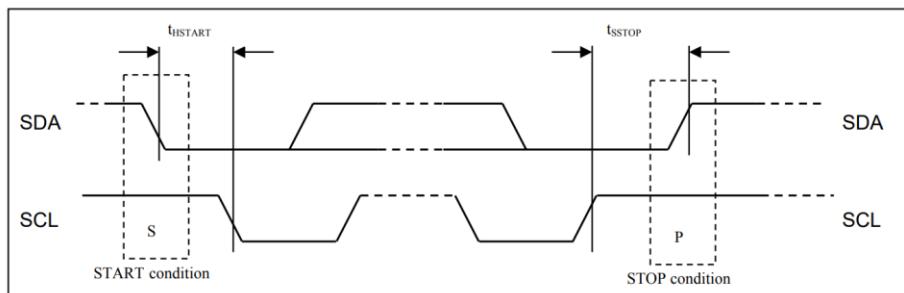
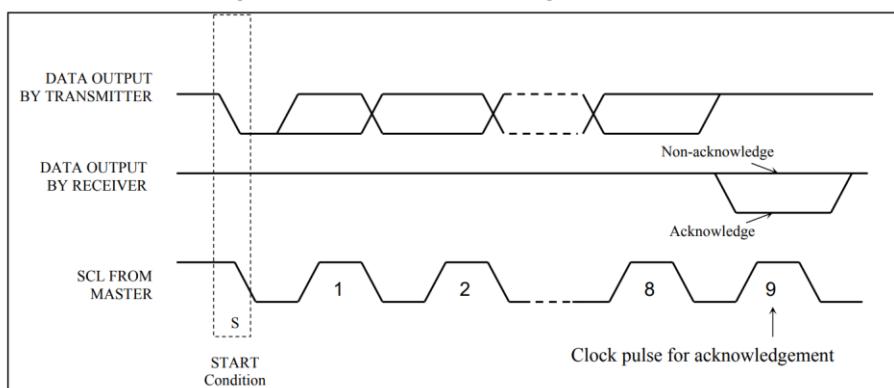


Figure 8-9 : Definition of the acknowledgement condition



通过上述分析，首先通过函数实现 I2C 写入 ssd1306  
HAL 库相关 API

### HAL\_I2C\_Master\_Transmit

```
/**  
 * 函数功能：在主模式下，以阻塞模式传输一定数量的数据。  
 *  
 * HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c,  
 * uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout)  
 *  
 * 参数：  
 *   hi2c - 指向 I2C_HandleTypeDef 结构的指针，包含指定 I2C 模块的配置信息。  
 *   DevAddress - 目标设备地址：数据表中的设备 7 位地址值在调用接口前必须左移。  
 *   pData - 指向数据缓冲区的指针。  
 *   Size - 要发送的数据量。  
 *   Timeout - 超时持续时间。  
 * 返回值：HAL 状态。  
 */
```

### HAL\_I2C\_Master\_Transmit

`HAL_I2C_Master_Transmit` 函数通过 I2C 总线向特定设备发送单个字节数据的函数。它接受两个参数：一个是目标设备的寄存器地址（addr），另一个是要写入的数据值（data）。在函数内部，首先创建了一个包含这两个参数的数组 TxData，数组的第一个元素是寄存器地址，第二个元素是要发送的数据。接着，函数调用 `HAL_I2C_Master_Transmit` 函数。在这次调用中，传递了 I2C 句柄（&IICx）、目标设备的固定 I2C 地址（0X78）、指向数据数组的指针、要发送的数据量（2 个字节），以及操作的超时时间（10 毫秒）。通过这个过程，函数能够将一个字节的数据安全地发送到通过 I2C 连接的外部设备的指定寄存器地址。

简单来说，该函数通过调用 `HAL_I2C_Master_Transmit` 函数，实现向 0x78 地址设备（SSD1306 屏幕）的 `addr`（指定寄存器地址）,写入 `data`（数据）。数据地址一般为 0x40;指令地址为 0x00

```
void HAL_I2C_WriteByte(uint8_t addr,uint8_t data)  
{  
    uint8_t TxData[2] = {addr,data};  
    HAL_I2C_Master_Transmit(&IICx,0X78,(uint8_t*)TxData,2,10);  
}
```

通过上述函数实现 I2C 写入命令与数据功能

```
void WriteCmd(uint8_t IIC_Command)
{
    HAL_I2C_WriteByte(0x00, IIC_Command);
}

void WriteDat(uint8_t IIC_Data)
{
    HAL_I2C_WriteByte(0x40, IIC_Data);
}
```

### 1.18.5 ssd1306 I2C 初始化程序

根据手册，对屏幕进行相关初始化配置。

格式为先发设置的寄存器地址，后发向寄存器地址写入的值

例如：

```
WriteCmd(0x81); //参数为寄存器地址 0x81，调整显示屏的对比度
WriteCmd(0xff); //显示屏的对比度的值
```

0	81	1	0	0	0	0	0	0	1	Set Contrast Control	Double byte command to select 1 out of 256 contrast steps. Contrast increases as the value increases. (RESET = 7Fh )
0	A[7:0]	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>		

通过查阅寄存器表，0x81 寄存器用于调整显示屏的对比度。取值范围为 0-255，以便调整屏幕的显示效果。

其他寄存器相关配置见手册，此处不再赘述，具体初始化如下。

```
void OLED_Init(void)

{
    HAL_Delay(500); //HAL 延时函数

    WriteCmd(0xAE); //关闭显示
    WriteCmd(0x20); //设置内存寻址模式

    WriteCmd(0x10); //00, 水平寻址模式;01, 垂直寻址模式;10, 页面寻址模式(重置);11, 无效
    WriteCmd(0xb0); //为页面寻址模式设置页面开始地址, 0-7
    WriteCmd(0x00); //---设置低列地址
    WriteCmd(0x10); //---设置高列地址

    WriteCmd(0xc8); //设置 COM 输出扫描方向
    WriteCmd(0x40); //---设置起始行地址
```

```

        WriteCmd(0x81); //--set contrast control register
        WriteCmd(0xff); //对比度调节 0x00~0xff
        WriteCmd(0xa1); //--设置段重新映射 0 到 127
        WriteCmd(0xa6); //--设置正常显示
        WriteCmd(0xa8); //--设置复用比(1 ~ 64)
        WriteCmd(0x3F); //
        WriteCmd(0xa4); //0xa4,输出遵循 RAM 内容;0xa5,Output 忽略 RAM 内容
        WriteCmd(0xd3); //设置显示抵消
        WriteCmd(0x00); //not offset
        WriteCmd(0xd5); //--设置显示时钟分频/振荡器频率
        WriteCmd(0xf0); //--设置分率
        WriteCmd(0xd9); //--设置 pre-charge 时期
        WriteCmd(0x22); //
        WriteCmd(0xda); //--设置 com 大头针硬件配置
        WriteCmd(0x12);
        WriteCmd(0xdb); //--设置 vcomh
        WriteCmd(0x20); //0x20,0.77xVcc
        WriteCmd(0x8d); //--设置 DC-DC
        WriteCmd(0x14); //
        WriteCmd(0xaf); //--打开 oled 面板

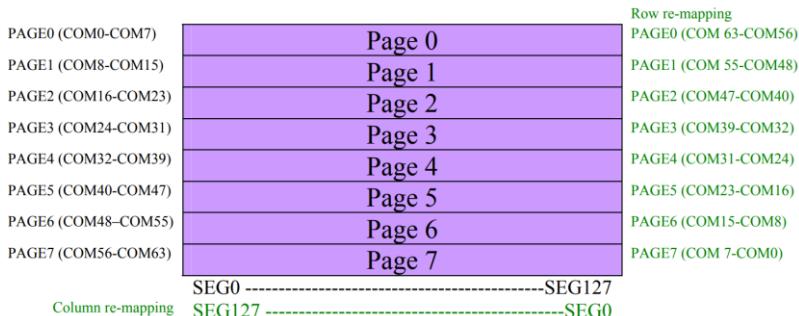
        OLED_FullyClear(); //清屏
    }
}

```

### 1.18.6 ssd1306 内存寻址模式

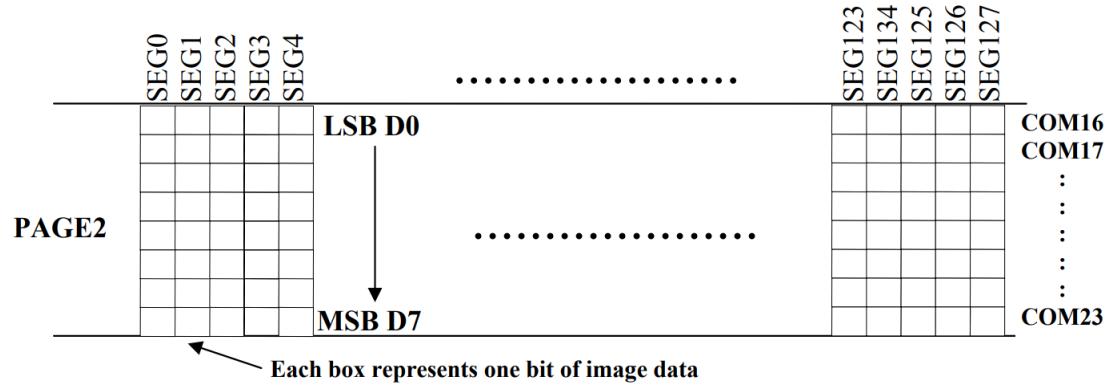
OLED 的分辨率是  $128 \times 64$ ，那么用显示器的术语来说就是水平方向上有 128 段（segments），竖直方向上有 64 个公共引脚（commons）。这与电脑的分辨率类似，比如电脑的分辨率是  $1920 \times 1080$ ，那么水平方向上（宽度）就分布着 1920 个像素点，竖直方向上（高度）就分布着 1080 个像素点。

在 OLED 竖直方向上将 64 个点阵分为 8 组 page，从 page0 到 page7，每页由 8 位像素点组成。这就像我们控制单片机开发板上面的 LED 需要配置寄存器一样，控制 OLED 上像素的亮灭也需要配置 PAGE。但是要注意的是 page 是个常用的用于描述显示器分区的术语，并不是特殊的寄存器。



GDDRAM (Graphic Display Data RAM, 图形数据显示内存) 是一个位映射静态内存，用与保存要显示的位模式，内存的大小就是  $128 \times 64$  bit，同时内存被细分成如上所述的 8 个 page。

如下是 GDDRAM 的局部放大图片，以 PAGE2 为例：



当一个字节数据被写入进 GDDRAM 时，当前列中所有处于同一 page 的行图像数据都会被填充。OLED 的每一页可以管理 8 行，我们以 PAGE2 为例，如图，在水平方向上就有 128 列，每列都有 8 个像素点，最上方的像素点是 bit0，最下方的像素点是 bit7。假设我们写入的数据是 0x08，那么其二进制就是 0000 1000，对应的就会点亮位于 SEG0 且处于 PAGE2 上 bit4 这一个像素点，同时熄灭其他像素点。

不论 OLED 上显示的是什么，我们首要的工作是做到如何点亮 OLED 上的一个点，步骤是先选一个 page，然后向其发送一个字节数据，点亮 page 的对应位置。而在在此之前，我们需要先了解地址指针在内存 GDDRAM 中的移动方式。OLED 对 page 有三种寻址办法，如下：

#### 页面寻址模式 (Page Addressing Mode, A[1:0]=10xb):

图中的箭头是页寻址模式下地址指针的移动方向，以在 PAGE0 上写入数据 0x80 为例，如果写入 1 次，那么仅能点亮 COL0 上处于 PAGE0 的 bit4 这一个像素；在这种寻址模式下，GDDRAM 的列地址指针在每次数据写入后会自动加 1，如果写入 128 次，那么就能显示水平的一条线，并且 PAGE0 中的 COL127 写完后又会回到 PAGE0 的 COL0。

在这种模式下，读/写显示 RAM 后，列地址指针自动增加 1。如果列地址指针达到列终止地址，它会重置到列起始地址，但页面地址指针不变。用户需设置新的页面和列地址来访问下一页 RAM 内容。

设置起始 RAM 访问指针位置需执行以下步骤：

通过 B0h 到 B7h 命令设置目标显示位置的页面起始地址。

通过 00h~0Fh 命令设置指针的下列起始地址。

通过 10h~1Fh 命令设置指针的上列起始地址。

	COL0	COL 1	.....	COL 126	COL 127
PAGE0					→
PAGE1					→
:	:	:	:	:	:
PAGE6					→
PAGE7					→

### 水平寻址模式 (Horizontal Addressing Mode, A[1:0]=00b):

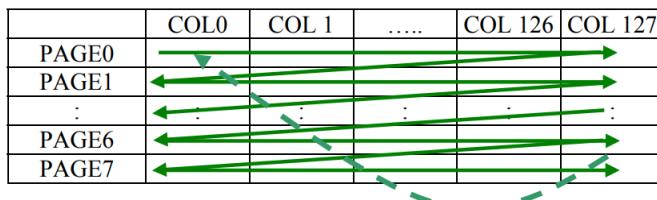
图中的箭头是水平寻址模式下地址指针的移动方向，在这种寻址模式下，GDDRAM 的列地址指针在每次数据写入后会自动加 1，但是在当前 PAGE 的最后一列写完之后，也就是 COL127 写完之后，地址指针会移动到下一页的首列，比如 PAGE0 的 COL127 写完之后，地址指针会移动到 PAGE1 的 COL0

在此模式下，读/写显示 RAM 后，列地址指针自动增加 1。如果列地址指针到达列终止地址，它会重置到列起始地址，页面地址指针增加 1。当列和页面地址指针都到达终止地址时，指针重置到列起始地址和页面起始地址。

设置 RAM 访问指针位置需执行以下步骤：

通过 21h 命令设置目标显示位置的列起始和终止地址。

通过 22h 命令设置目标显示位置的页面起始和终止地址。



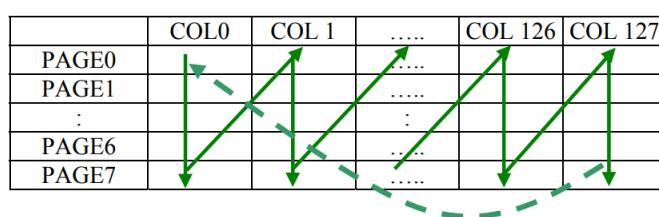
### 垂直寻址模式 (Vertical Addressing Mode, A[1:0]=01b):

图中的箭头是垂直寻址模式下地址指针的移动方向。

在此模式下，读/写显示 RAM 后，页面地址指针自动增加 1。如果页面地址指针到达页面终止地址，它会重置到页面起始地址，列地址指针增加 1。

当列和页面地址指针都到达终止地址时，指针重置到列起始地址和页面起始地址。

设置 RAM 访问指针位置需执行与水平寻址模式相同的步骤。



目前我们设置的为页面寻址模式

### 1.18.7 ssd1306 显示程序分析

显示字符串程序

```
void OLED_ShowStr1(unsigned char x, unsigned char y, int *ch, unsigned
char s, unsigned char TextSize)
{
    unsigned char c = 0, i = 0, j = 0;
    int ascii_offset = 32; // 字库中字符与 ASCII 码的偏移量

    switch (TextSize)
    {
        case 1: // 6x8 字符大小
        {
            while (s--) // 遍历每个字符
            {
                c = ch[j] - ascii_offset; // 计算字符在字库中的位置
                if (x > 126) // 检查 x 坐标是否超出屏幕限制
                {
                    x = 0; // 重置 x 坐标
                    y++; // 移动到下一行
                }
                OLED_SetPos(x, y); // 设置字符显示位置
                for (i = 0; i < 6; i++) // 遍历每个字符的 6 个字节数据
                    WriteDat(F6x8[c][i]); // 将数据写入 OLED
                x += 6; // 移动 x 坐标以显示下一个字符
                j++; // 移到字符串的下一个字符
            }
        } break;

        case 2: // 8x16 字符大小
        {
            while (s--) // 遍历每个字符
            {
                c = ch[j] - ascii_offset; // 计算字符在字库中的位置
                if (x > 120) // 检查 x 坐标是否超出屏幕限制
                {
                    x = 0; // 重置 x 坐标
                    y++; // 移动到下一行
                }
                OLED_SetPos(x, y); // 设置字符显示位置
                for (i = 0; i < 8; i++) // 遍历字符的前 8 个字节数据
                    WriteDat(F8X16[c * 16 + i]);
                OLED_SetPos(x, y + 1); // 移动到下一行，显示字符的下半部分
            }
        }
    }
}
```

```

        for (i = 0; i < 8; i++) // 遍历字符的后 8 个字节数据
            WriteDat(F8X16[c * 16 + i + 8]);
        x += 8; // 移动 x 坐标以显示下一个字符
        j++; // 移到字符串的下一个字符
    }
} break;
}
}

```

显示汉字程序

```

void OLED_ShowCN(unsigned char x, unsigned char y, unsigned char N)
{
    unsigned char wm = 0; // 循环计数变量
    unsigned int adder = 32 * N; // 计算字符在字库中的起始位置

    // 首先设置显示位置
    OLED_SetPos(x, y);

    // 显示字符的上半部分（前 16 字节）
    for(wm = 0; wm < 16; wm++)
    {
        WriteDat(F16x16[adder]); // 将字符的一个字节写入 OLED
        adder += 1; // 移动到下一个字节
    }

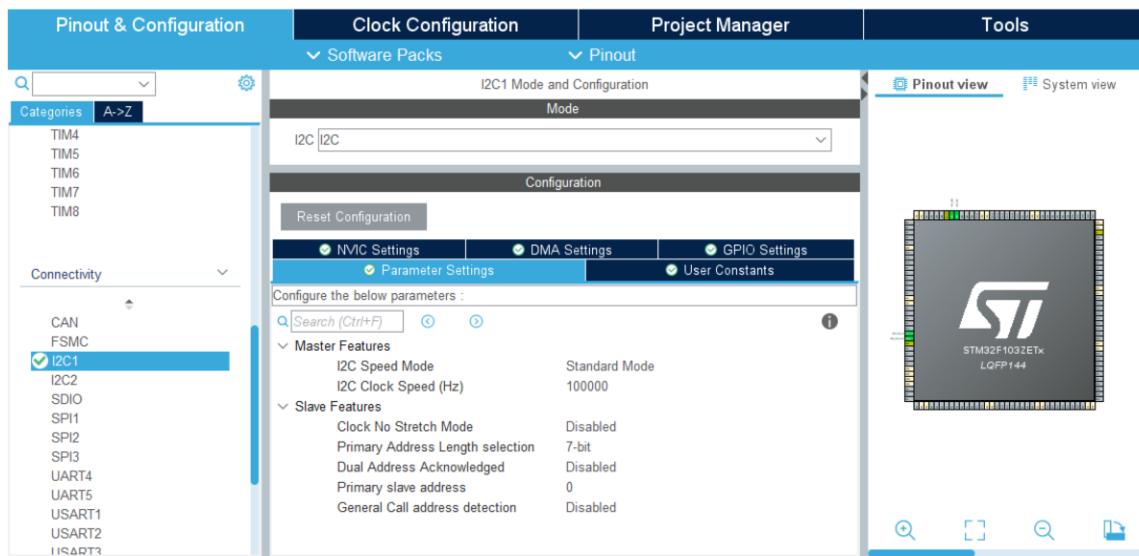
    // 设置显示下半部分的位置
    OLED_SetPos(x, y + 1);

    // 显示字符的下半部分（后 16 字节）
    for(wm = 0; wm < 16; wm++)
    {
        WriteDat(F16x16[adder]); // 将字符的一个字节写入 OLED
        adder += 1; // 移动到下一个字节
    }
}

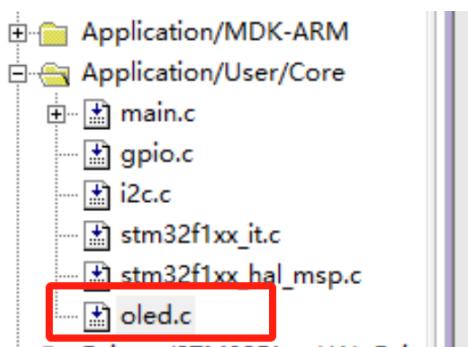
```

### 1.18.8 显示“千锋欢迎你”

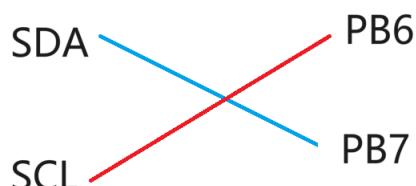
新建 CubeMX 工程，配置 RCC 等信息，同时配置 I2C 如下



生成工程后，添加 oled.c



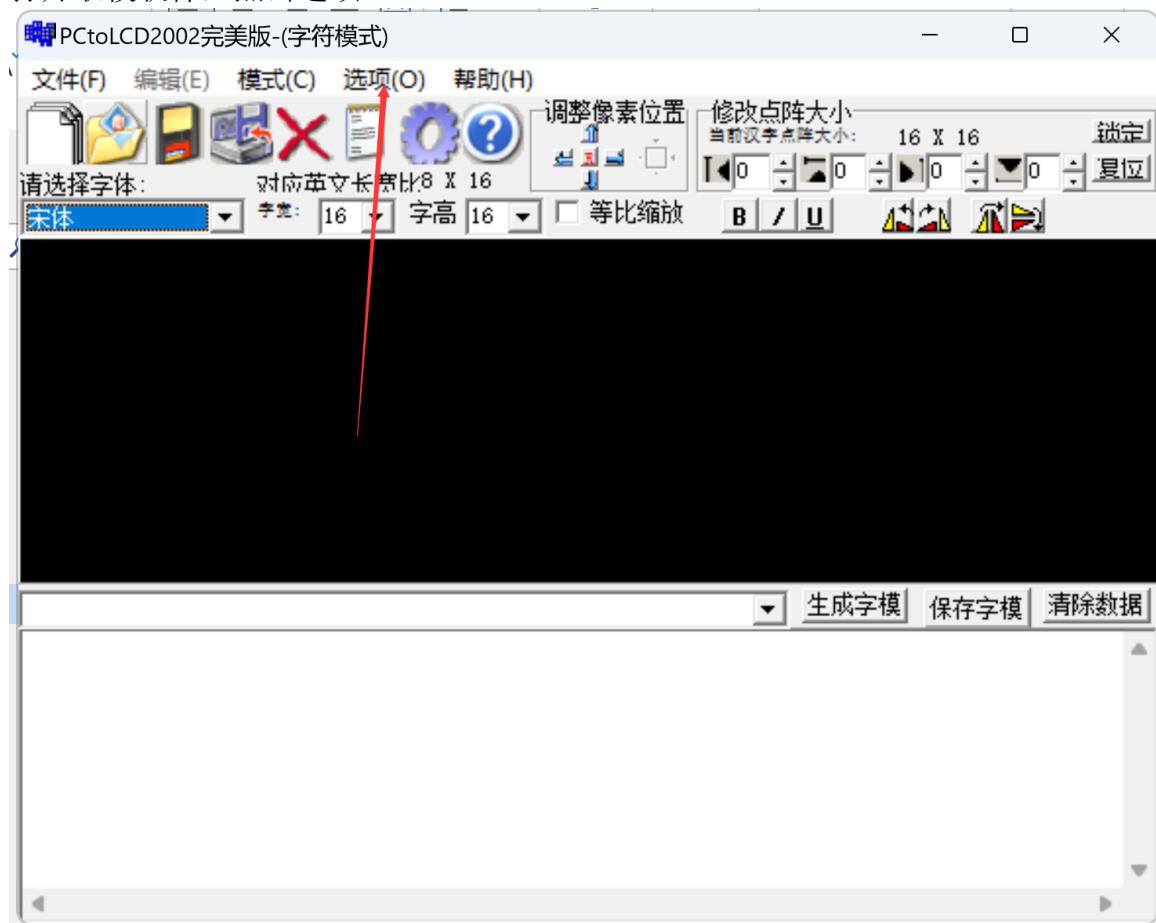
接线方式



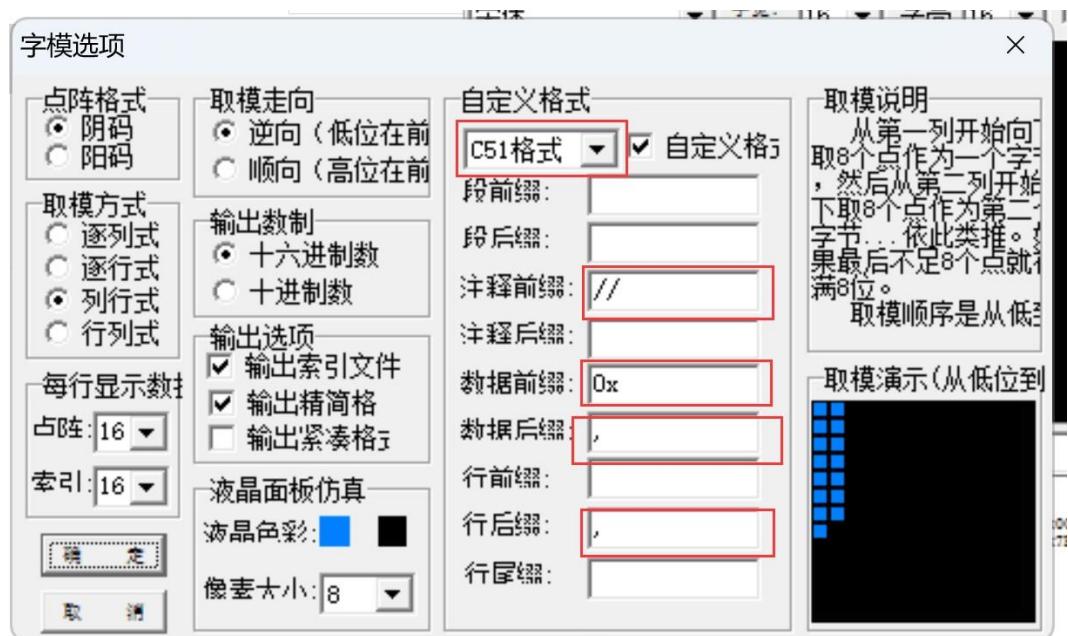
GND ————— GND

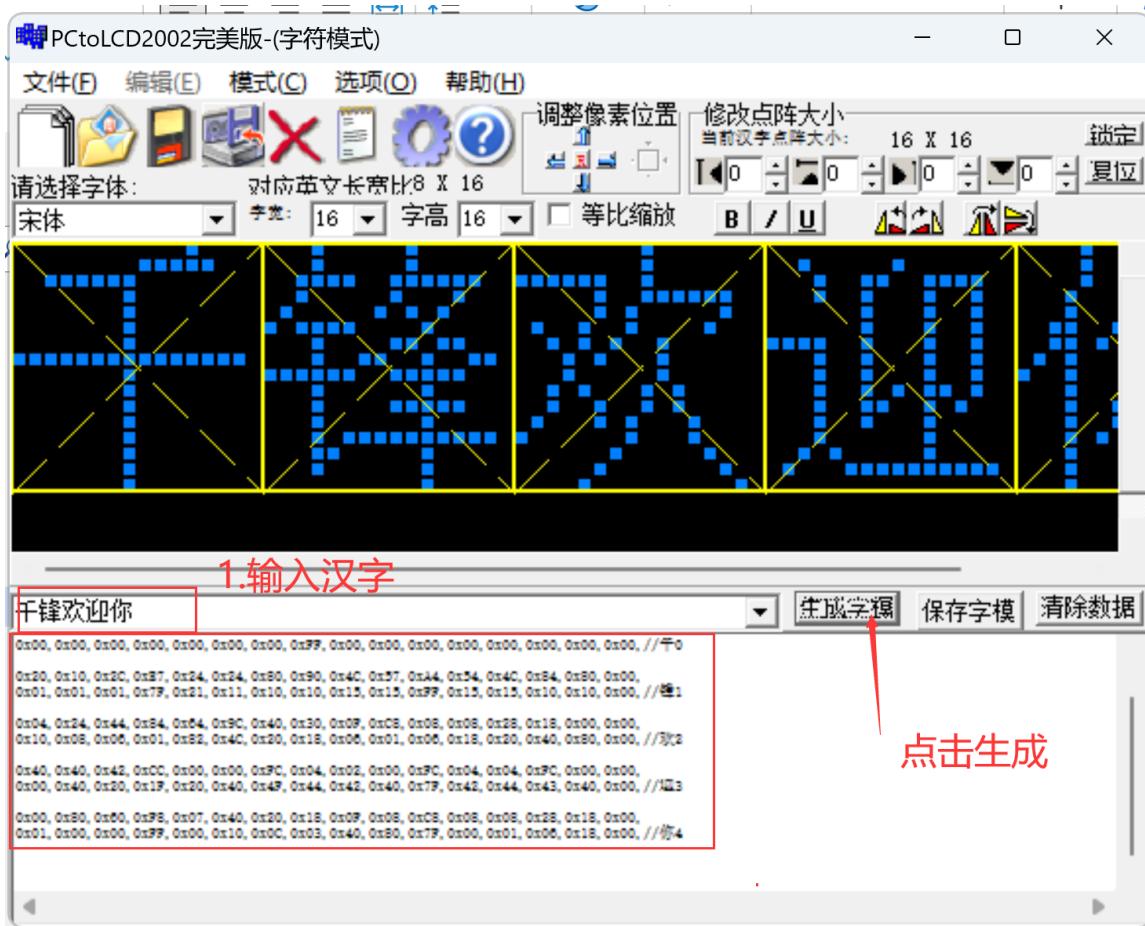
VCC ————— 5V

打开取模软件，点击选项



按如下方式配置





将上述字模复制到 codetap.h

```

3 const unsigned char F16x16[] =
4 {
5
6     0x80, 0x80, 0x84, 0x84, 0x84, 0x84, 0x84, 0xFC, 0x82, 0x82, 0x82, 0x83, 0x82, 0x82, 0x80, 0x80, 0x00,
7     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // 千0
8
9     0x20, 0x10, 0x2C, 0xE7, 0x24, 0x24, 0x80, 0x90, 0x4C, 0x57, 0xA4, 0x54, 0x4C, 0x84, 0x80, 0x00,
0.0 0x01, 0x01, 0x01, 0x7F, 0x21, 0x11, 0x10, 0x10, 0x15, 0x15, 0xFF, 0x15, 0x15, 0x10, 0x10, 0x00, // 锋1
.1
.2     0x04, 0x24, 0x44, 0x84, 0x64, 0x9C, 0x40, 0x30, 0x0F, 0xC8, 0x08, 0x08, 0x28, 0x18, 0x00, 0x00,
.3 0x10, 0x08, 0x06, 0x01, 0x82, 0x4C, 0x20, 0x18, 0x06, 0x01, 0x06, 0x18, 0x20, 0x40, 0x80, 0x00, // 欢2
.4
.5     0x40, 0x40, 0x42, 0xCC, 0x00, 0x00, 0xFC, 0x04, 0x02, 0x00, 0x00, 0x04, 0x04, 0x04, 0x00, 0x00,
.6 0x00, 0x40, 0x20, 0x1F, 0x20, 0x40, 0x4F, 0x44, 0x42, 0x40, 0x7F, 0x42, 0x44, 0x43, 0x40, 0x00, // 迎3
.7
.8     0x00, 0x80, 0x60, 0xF8, 0x07, 0x40, 0x20, 0x18, 0x0F, 0x08, 0x08, 0x28, 0x18, 0x00,
.9 0x01 0x00 // 你4

```

主函数初始化并显示

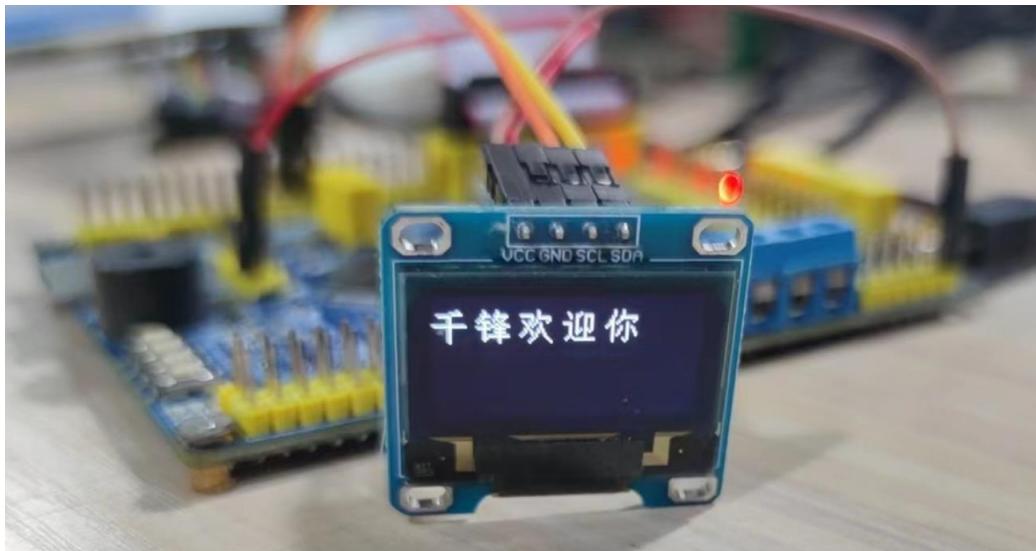
```

/* USER CODE BEGIN 2 */
OLED_Init();
OLED_ShowCN(0, 1, 0);
OLED_ShowCN(20, 1, 1);
OLED_ShowCN(40, 1, 2);

```

```
OLED_ShowCN(60, 1,3);  
OLED_ShowCN(80, 1,4);  
/* USER CODE END 2 */
```

显示效果



### 1.18.9 作业

制作简易屏幕显示界面，显示当前温湿度信息。

## 1.19 SPI 通信

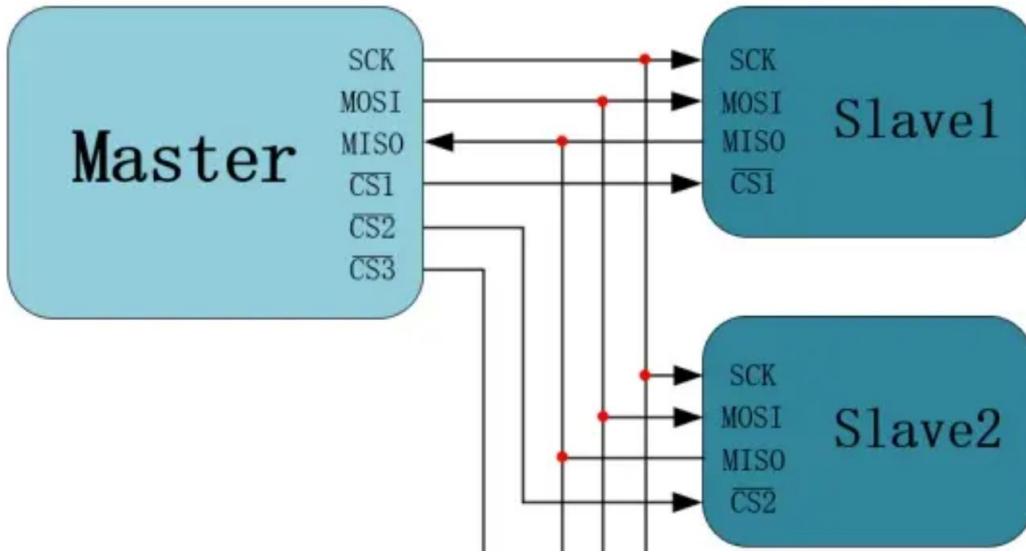
### 1.19.1 SPI 概述

SPI ( Serial Peripheral Interface, 串行外围设备接口) 通讯协议，是 Motorola 公司提出的一种同步串行接口技术，是一种高速、全双工、同步通信总线，在芯片中只占用四根管脚用来控制及数据传输。

应用:EEPROM、Flash、RTC、ADC、DSP 等。

优缺点：全双工通信，通讯方式较为简单，相对数据传输速率较快;没有应答机制确认数据是否接收，在数据可靠性上有一定缺陷（与 I2C 相比）。

SPI 通讯设备的通讯模式是主从通讯模式，通讯双方有主从之分，根据从机设备的个数，SPI 通讯设备之间的连接方式可分为一主一从和一主多从。



SPI 通讯协议包含 1 条时钟信号线、2 条数据总线和 1 条片选信号线，时钟信号线为 SCK，2 条数据总线分别为 MOSI(主输出从输入)、MISO(主输入从输出)，片选信号线为，它们的作用介绍如下：

**SCK (Serial Clock):** 时钟信号线，用于同步通讯数据。由通讯主机产生，决定了通讯的速率，不同的设备支持的最高时钟频率不同，两个设备之间通讯时，通讯速率受限于低速设备。

**MOSI (Master Output, Slave Input):** 主设备输出/从设备输入引脚。主机的数据从这条信号线输出，从机由这条信号线读入主机发送的数据，数据方向由主机到从机。

**MISO (Master Input, Slave Output):** 主设备输入/从设备输出引脚。主机从这条信号线读入数据，从机的数据由这条信号线输出到主机，数据方向由从机到主机。

**(Chip Select):** 片选信号线，也称为 CS\_N，以下用 CS\_N 表示。当有多个 SPI 从设备与 SPI 主机相连时，设备的其它信号线 SCK、MOSI 及 MISO 同时并联到相同的 SPI 总线上，即无论有多少个从设备，都共同使用这 3 条总线；而每个从设备都有独立的这一条 CS\_N 信号线，本信号线独占主机的一个引脚，即有多少个从设备，就有多少条片选信号线。

I2C 协议中通过设备地址来寻址、选中总线上的某个设备并与之进行通讯；而 SPI 协议中没有设备地址，它使用 CS\_N 信号线来寻址，当主机要选择从设备时，把该从设备的 CS\_N 信号线设置为低电平，该从设备即被选中，即片选有效，接着主机开始与被选中的从设备进行 SPI 通讯。所以 SPI 通讯以 CS\_N 线置低电平为开始信号，以 CS\_N 线被拉高作为结束信号。

SPI 通讯协议的 4 种模式如下。

模式 0: CPOL= 0, CPHA=0。空闲状态时 SCK 串行时钟为低电平；数据采样在 SCK 时钟的奇数边沿，本模式中，奇数边沿为上升沿；数据更新在 SCK 时钟的偶数边沿，本模式中，偶数边沿为下降沿。

模式 1: CPOL= 0, CPHA=1。空闲状态时 SCK 串行时钟为低电平；数据采样在 SCK 时钟的偶数边沿，本模式中，偶数边沿为下降沿；数据更新在 SCK 时钟的奇数边沿，本模式中，偶数边沿为上升沿。

模式 2: CPOL= 1, CPHA=0。空闲状态时 SCK 串行时钟为高电平；数据采样在 SCK 时钟的奇数边沿，本模式中，奇数边沿为下降沿；数据更新在 SCK 时钟的偶数边沿，本模式中，偶数边沿为上升沿。

模式 3: CPOL= 1, CPHA=1。空闲状态时 SCK 串行时钟为高电平；数据采样在 SCK 时钟的偶数边沿，本模式中，偶数边沿为上升沿；数据更新在 SCK 时钟的奇数边沿，本模式中，偶数边沿为下降沿。

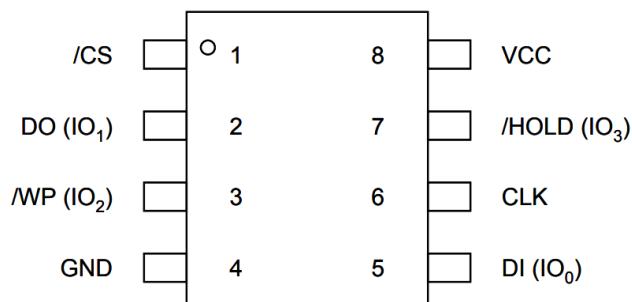
### 1.19.2 W25Q16 存储芯片概述

FLASH 存储器又称闪存，是一种长寿命的非易失性（在断电情况下仍能保持所有存储的数据信息）存储器，一般用来保存一些重要的设置信息或者程序等等。通常采用 SPI 协议跟 FLASH 芯片进行数据交互。W25Q16BV 是串行闪存为空间，引脚和功耗有限的系统提供存储解决方案。25Q 系列提供的灵活性和性能远远超过普通串行闪存设备。

W25Q16 存储芯片的特点是

- 1、低功耗
- 2、具有 4KB 扇区的灵活架构
- 3、高效的连续读取模式
- 4、最高性能串行闪存
- 5、16Mbit(2MB)

引脚说明如下：



PIN NO.	PIN NAME	I/O	FUNCTION
1	/CS	I	Chip Select Input
2	DO (IO1)	I/O	Data Output (Data Input Output 1)* <sup>1</sup>
3	/WP (IO2)	I/O	Write Protect Input ( Data Input Output 2)* <sup>2</sup>
4	GND		Ground
5	DI (IO0)	I/O	Data Input (Data Input Output 0)* <sup>1</sup>
6	CLK	I	Serial Clock Input
7	/HOLD (IO3)	I/O	Hold Input (Data Input Output 3)* <sup>2</sup>
8	VCC		Power Supply

1 号引脚：这是我们的片选信号，并且是低电平有效，一般引脚名前加/表示低电平有效。

2、5 号引脚：FLASH 芯片的数据输入输出引脚( (Standard and Dual SPI 模式下)

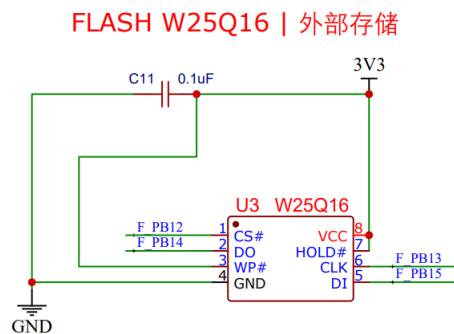
3 号引脚：写保护引脚 (Quad SPI 模式下被用作数据输出引脚，低电平有效，用来防止状态寄存器被写入。

4、8 号引脚：GND 和电源

6 号引脚：FLASH 的时钟驱动时钟

7 号引脚：HOLD 引脚，也是低电平有效，在 Standard and Dual SPI 模式下相当于 FLASH 暂停信号，当 HOLD 引脚和 CS 引脚同时拉低时，虽然此时 FLASH 芯片是被选中的，但是 DO 引脚会处于高阻抗状态，DI 和 CLK 这两个引脚会忽略输入的数据和时钟，相当于 DI 和 CLK 处于无效状态。(在多从机的情况下有用)

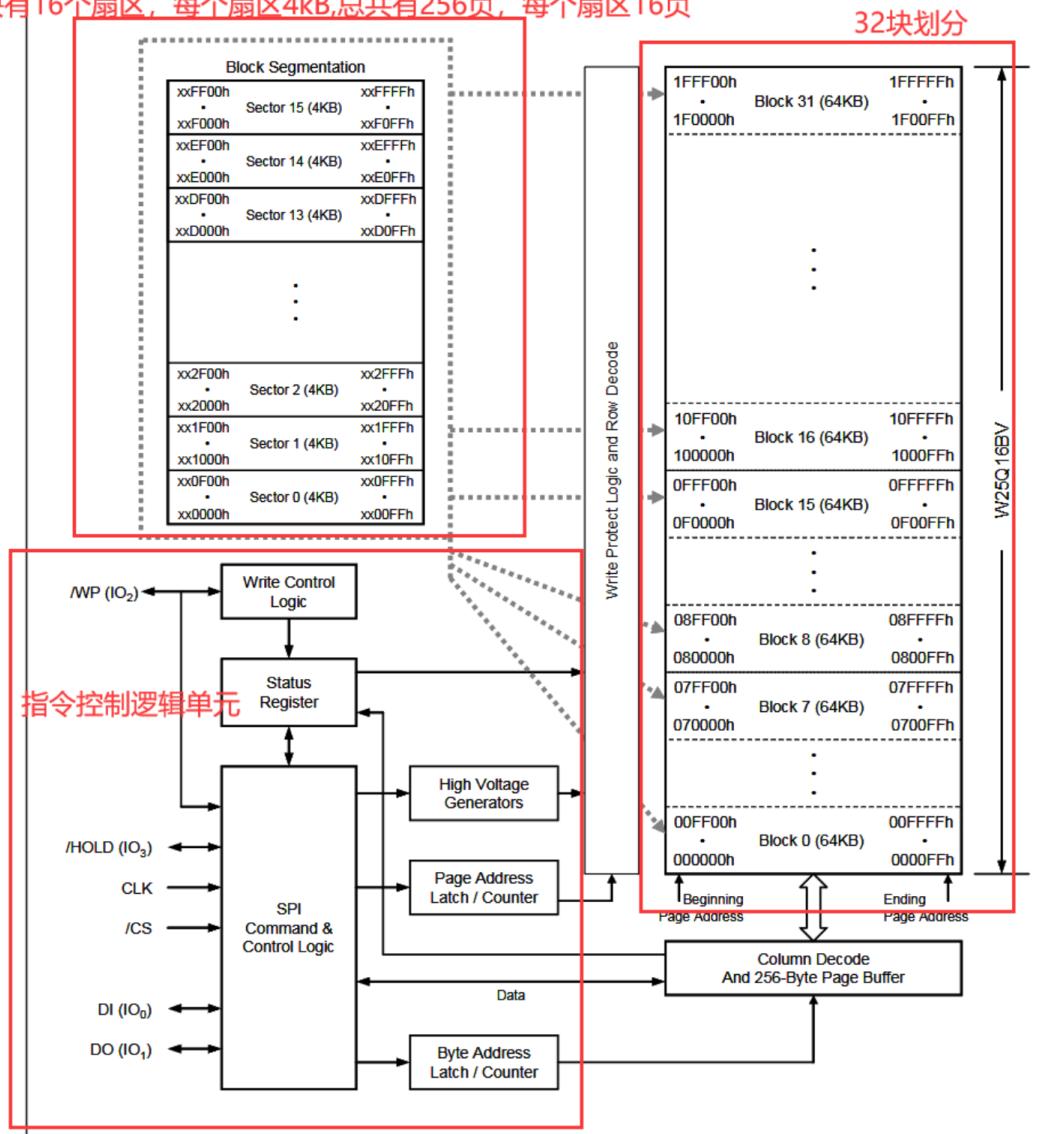
本开发板的原理图如下：



查阅手册，内部结构示意图如下：

每块有16个扇区，每个扇区4kB,总共有256页，每个扇区16页

32块划分



查阅操作手册可知，操作指令如下：

INSTRUCTION NAME	BYTE 1 (CODE)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Write Enable	06h					
Write Disable	04h					
Read Status Register-1	05h	(S7-S0) <sup>(2)</sup>				
Read Status Register-2	35h	(S15-S8) <sup>(2)</sup>				
Write Status Register	01h	(S7-S0)	(S15-S8)			
Page Program	02h	A23-A16	A15-A8	A7-A0	(D7-D0)	
Quad Page Program	32h	A23-A16	A15-A8	A7-A0	(D7-D0, ...) <sup>(3)</sup>	
Sector Erase (4KB)	20h	A23-A16	A15-A8	A7-A0		
Block Erase (32KB)	52h	A23-A16	A15-A8	A7-A0		
Block Erase (64KB)	D8h	A23-A16	A15-A8	A7-A0		
Chip Erase	C7h/60h					
Erase Suspend	75h					
Erase Resume	7Ah					
Power-down	B9h					
Continuous Read Mode Reset <sup>(4)</sup>	FFh	FFh				

INSTRUCTION NAME	BYTE 1 (CODE)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Read Data	03h	A23-A16	A15-A8	A7-A0	(D7-D0)	
Fast Read	0Bh	A23-A16	A15-A8	A7-A0	dummy	(D7-D0)
Fast Read Dual Output	3Bh	A23-A16	A15-A8	A7-A0	dummy	(D7-D0, ...) <sup>(1)</sup>
Fast Read Dual I/O	BBh	A23-A8 <sup>(2)</sup>	A7-A0, M7-M0 <sup>(2)</sup>	(D7-D0, ...) <sup>(1)</sup>		
Fast Read Quad Output	6Bh	A23-A16	A15-A8	A7-A0	dummy	(D7-D0, ...) <sup>(3)</sup>
Fast Read Quad I/O	EBh	A23-A0, M7-M0 <sup>(4)</sup>	(x,x,x,x, D7-D0, ...) <sup>(5)</sup>	(D7-D0, ...) <sup>(3)</sup>		
Word Read Quad I/O <sup>(7)</sup>	E7h	A23-A0, M7-M0 <sup>(4)</sup>	(x,x, D7-D0, ...) <sup>(6)</sup>	(D7-D0, ...) <sup>(3)</sup>		
Octal Word Read Quad I/O <sup>(8)</sup>	E3h	A23-A0, M7-M0 <sup>(4)</sup>	(D7-D0, ...) <sup>(3)</sup>			

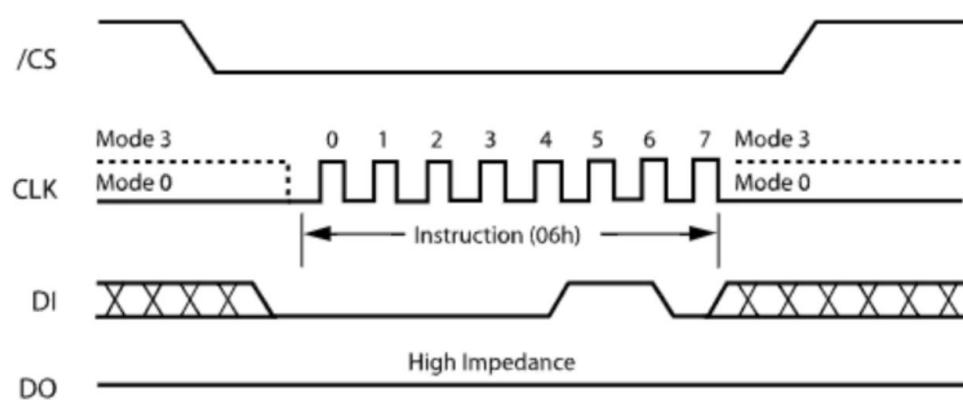
INSTRUCTION NAME	BYTE 1 (CODE)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Release Power down / Device ID	ABh	dummy	dummy	dummy	(ID7-ID0) <sup>(1)</sup>	
Manufacturer/ Device ID <sup>(2)</sup>	90h	dummy	dummy	00h	(MF7-MF0)	(ID7-ID0)
Manufacturer/Device ID by Dual I/O	92h	A23-A8	A7-A0, M[7:0]	(MF[7:0], ID[7:0])		
Manufacture/Device ID by Quad I/O	94h	A23-A0, M[7:0]	xxxx, (MF[7:0], ID[7:0])	(MF[7:0], ID[7:0], ...)		
JEDEC ID	9Fh	(MF7-MF0) Manufacturer	(ID15-ID8) Memory Type	(ID7-ID0) Capacity		
Read Unique ID	4Bh	dummy	dummy	dummy	dummy	(ID63-ID0)

### 1.19.3 操作示例时序图

操作示例时序图如下：

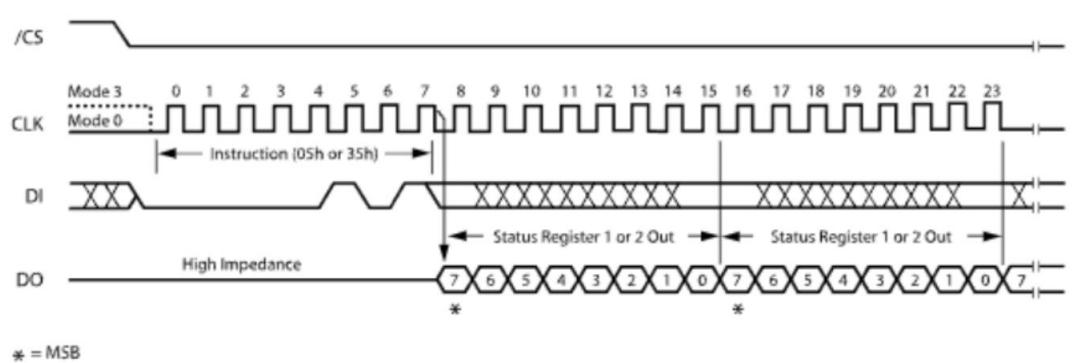
#### 1.写入启用指令

Write Enable (06h)：使能指令，Write Enable 指令将状态寄存器中的 Write Enable Latch (WEL)位设置为高电平，在执行页编辑、扇区擦除、块擦除、芯片擦除和写状态寄存器指令前必须先执行 Write Enable 指令。



#### 2.读取状态寄存器指令：Read Status Register-1 (05h)

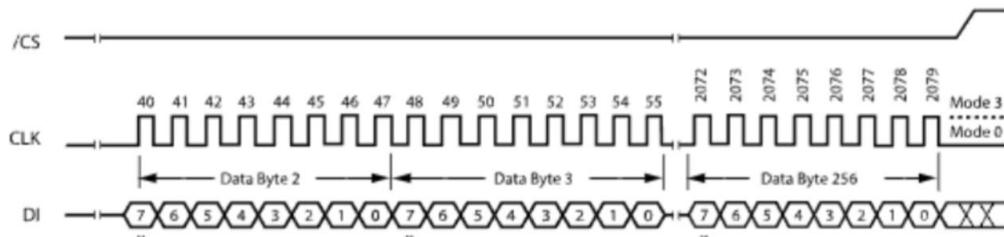
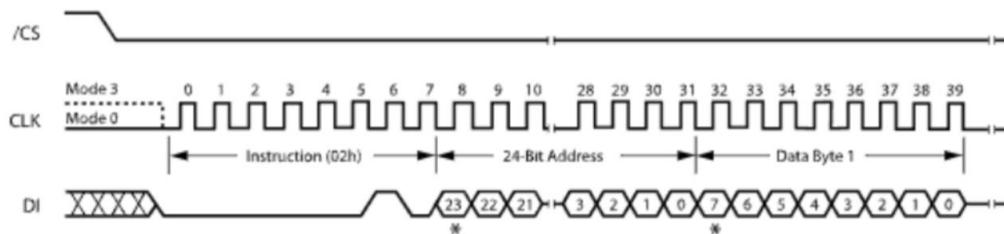
Read Status Register-1 (05h)：读取状态寄存器 1 指令，这条指令的作用就是读取状态寄存器 1 的值。在 W25Q16BV 中存在两个状态寄存器可以用来指示 FLASH 芯片的可用性状态或者配置 SPI 的相关设置。后面我们主要是读取状态寄存器 1 的值，并且检测它的第零位也就是 BUSY 位是零还是一。当 FLASH 处于擦除或者写入数据时，状态寄存器 1 的 BUSY 位会拉高，当 BUSY 位重新恢复成低电平时代表 FLASH 擦除或者写入数据完成。



#### 3.写数据指令：Page Program (02h)

页编辑指令，可以理解成写数据指令，在上文中我们已经介绍过 W25Q16BV 的

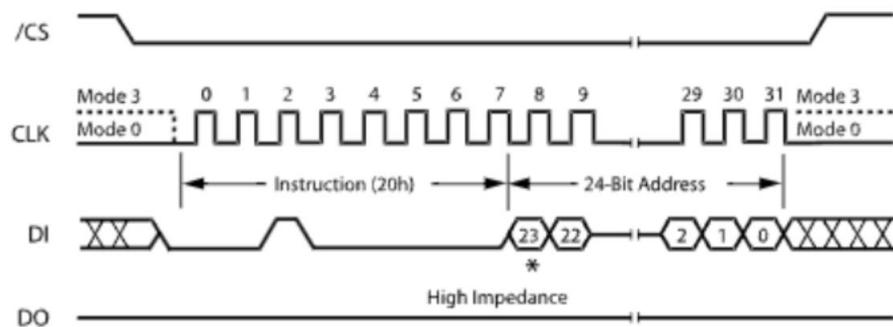
存储空间是分为扇区和页的，每一页又有 256 个字节的存储空间。当执行页编辑指令时就可以往对应的扇区对应的页中写入数据，一次性最多写入 256 个字节数据。这里尤其要注意一点，当整页写数据时你可以不必写满 256 个字节数据，小于 256 个字节也是可以正常写入的，但是千万不能超过 256 个字节数据，因为一旦超过 256 个字节，多余的数据就会返回这一页的开头重新写入，这样就会覆盖之前已经写入的数据。举个例子我们一次性写入 260 个数据，这样就多了 4 个数据出来，那么这 4 个数据就会回到这一页的开头把第 0、1、2 和 3 这四个数据覆盖掉。



\* = MSB

#### 4. 区块擦除指令：Sector Erase-4KB (20h)

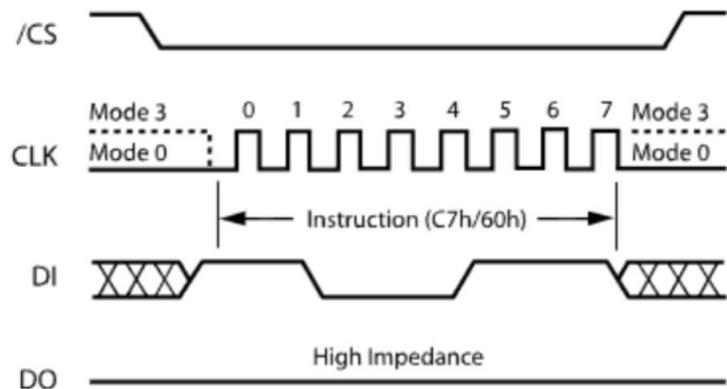
在上文中我们已经提到过芯片内部的存储空间是被划分成一个一个小块的，我们可以直接对这些小块执行擦除指令。Sector Erase 指令就是对 4KB 的小块执行擦除操作。除此之外还有 Block Erase -32KB (52h) 和 Block Erase -64KB (D8h) 指令，用于擦除更大的存储块。



\* = MSB

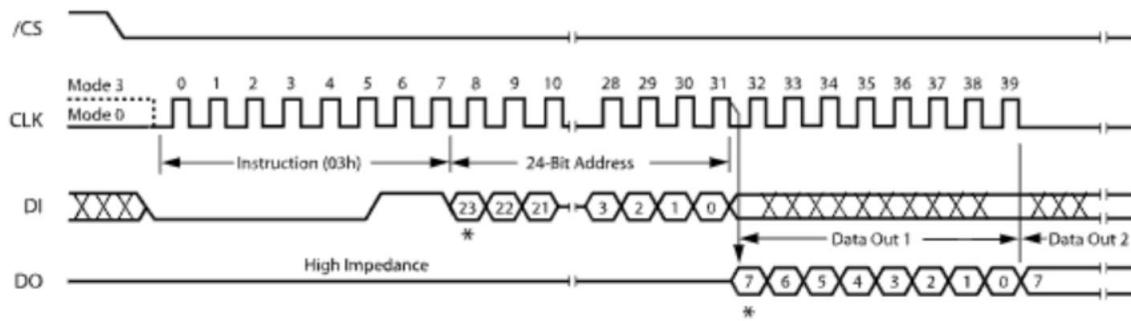
## 5. 全擦除指令: Chip Erase (c7h)

全擦除指令会擦除整个 FLASH 芯片的所有存储数据。需要注意的是全擦除指令执行的比较慢，通常需要几秒钟才能完成全擦除。在芯片处于全擦除期间我们只能对它执行访问状态寄存器指令操作，不能执行其他例如读写等操作。在擦除期间，状态寄存器 1 的最低位（BUSY 位）始终处于高电平（不仅仅是全擦除指令，区块擦除指令、写数据指令执行时 BUSY 位也会拉高），当完成全擦除后 BUSY 位拉低，此时可以执行其他指令。



## 6. 读取数据指令: Read Data (03h)

读取数据指令，当 FLASH 中被写入数据后我们可以使用 Read Data 指令将数据读取出来。



\* = MSB

上述延迟时间可以查阅手册末尾的时间表

DESCRIPTION	SYMBOL	ALT	SPEC			UNIT
			MIN	TYP	MAX	
Clock frequency for all instructions, except Read Data (03h) & Octal Word Read (E3h) 2.7V-3.6V VCC & Industrial Temperature	$F_R$	$f_C$	D.C.		80	MHz
Clock frequency for all instructions, except Read Data (03h) & Octal Word Read (E3h) 3.0V-3.6V VCC & Commercial Temperature	$F_R$	$f_C$	D.C.		104	MHz
Clock frequency for Octal Word Read (E3h) 3.0V-3.6V VCC & Industrial Temperature	$F_R$	$f_C$	D.C.		50	MHz
Clock freq. Read Data instruction (03h)	$f_R$		D.C.		50	MHz
Clock High, Low Time except Read Data (03h)	$t_{CLH},$ $t_{CLL}^{(1)}$		4.5			ns

#### 1.19.4 W25Q16 读写程序分析

本次采取软件模拟 SPI

Spi.c

```
#include "spi.h"

/*spi 延时函数，微秒*/
static void spi_delay(uint16_t time)
{
    uint16_t i=0;
    while(time--)
    {
        i=10;
        while(i--);
    }
}

//CPOL: 规定了 SCK 时钟信号空闲状态的电平(0-低电平,1-高电平)
//CPHA: 规定了数据是在 SCK 时钟的上升沿还是下降沿被采样(0-第一个时钟边沿开始采样,1-第二个时钟边沿开始采样)

//模式 0: CPOL=0, CPHA =0 SCK 空闲为低电平, 数据在 SCK 的上升沿被采样(提取数据)
//模式 1: CPOL=0, CPHA =1 SCK 空闲为低电平, 数据在 SCK 的下降沿被采样(提取数据)
//模式 2: CPOL=1, CPHA =0 SCK 空闲为高电平, 数据在 SCK 的下降沿被采样(提取数据)
//模式 3: CPOL=1, CPHA =1 SCK 空闲为高电平, 数据在 SCK 的上升沿被采样(提取数据)

/* CPOL = 0, CPHA = 0 */
uint8_t SOFT_SPI_RW_MODE0(uint8_t write_dat)
{
    uint8_t i,read_dat = 0;
    SCK_L;
    for(i=0;i<8;i++)
    {
        if((write_dat & 0x80) == 0)
            SCK_H;
        else
            SCK_L;
        write_dat = write_dat << 1;
        delay();
    }
    read_dat = SDO;
}
```

```

{
    if(write_dat&0x80)
        MOSI_H;
    else
        MOSI_L;
    write_dat <= 1;
    spi_delay(1);
    SCK_H;
    read_dat <= 1;

    if(MISO)
        read_dat++;
    spi_delay(1);
    SCK_L;
    __nop();
}
return read_dat;
}

/* CPOL=0, CPHA=1 */
uint8_t SOFT_SPI_RW_MODE1(uint8_t write_dat)
{
    uint8_t i,read_dat = 0;
    SCK_L;
    for(i=0;i<8;i++)
    {
        SCK_H;
        if(write_dat&0x80)
            MOSI_H;
        else
            MOSI_L;
        write_dat <= 1;
        spi_delay(1);
        SCK_L;
        read_dat <= 1;

        if(MISO)
            read_dat++;
        spi_delay(1);
    }
    return read_dat;
}

/* CPOL=1, CPHA=0 */
uint8_t SOFT_SPI_RW_MODE2(uint8_t write_dat)

```

```

{
    uint8_t i,read_dat = 0;
    SCK_H;
    for(i=0;i<8;i++)
    {
        if(write_dat&0x80)
            MOSI_H;
        else
            MOSI_L;
        write_dat <<= 1;
        spi_delay(1);
        SCK_L;
        read_dat <<= 1;

        if(MISO)
            read_dat++;
        spi_delay(1);
        SCK_H;
    }
    return read_dat;
}

/* CPOL = 1, CPHA = 1 */
uint8_t SOFT_SPI_RW_MODE3(uint8_t write_dat)
{
    uint8_t i,read_dat = 0;
    SCK_H;
    for(i=0;i<8;i++)
    {
        SCK_L;
        if(write_dat&0x80)
            MOSI_H;
        else
            MOSI_L;
        write_dat <<= 1;
        spi_delay(1);
        SCK_H;
        read_dat <<= 1;

        if(MISO)
            read_dat++;
        spi_delay(1);
        __nop();
    }
    return read_dat;
}

```

```

}

//SPI2 读写一个字节
//TxData:要写入的字节
//返回值:读取到的字节
uint8_t SPI2_ReadWriteByte(uint8_t TxData)
{
    uint8_t Rxdata;
    Rxdata = SOFT_SPI_RW_MODE3(TxData); //使用模式 3
    return Rxdata;
}

```

## Spi.h

```

#ifndef __SPI_H
#define __SPI_H

#include "main.h"

#define MOSI_H HAL_GPIO_WritePin(MOSI_GPIO_Port, MOSI_Pin,
GPIO_PIN_SET)
#define MOSI_L HAL_GPIO_WritePin(MOSI_GPIO_Port, MOSI_Pin,
GPIO_PIN_RESET)
#define SCK_H HAL_GPIO_WritePin(SCLK_GPIO_Port, SCLK_Pin,
GPIO_PIN_SET)
#define SCK_L HAL_GPIO_WritePin(SCLK_GPIO_Port, SCLK_Pin,
GPIO_PIN_RESET)
#define MISO HAL_GPIO_ReadPin(MISO_GPIO_Port, MISO_Pin)
#define F_CS_H HAL_GPIO_WritePin(F_CS_GPIO_Port, F_CS_Pin,
GPIO_PIN_SET)
#define F_CS_L HAL_GPIO_WritePin(F_CS_GPIO_Port, F_CS_Pin,
GPIO_PIN_RESET)

uint8_t SOFT_SPI_RW_MODE0(uint8_t write_dat);
uint8_t SOFT_SPI_RW_MODE1(uint8_t write_dat);
uint8_t SOFT_SPI_RW_MODE2(uint8_t write_dat);
uint8_t SOFT_SPI_RW_MODE3(uint8_t write_dat);

uint8_t SPI2_ReadWriteByte(uint8_t TxData);

#endif

```

## w25q16.c

```

#include "w25q16.h"
#include "spi.h"

```

```

#include "tim.h"

//微秒级延时函数(向下)
void Delay_Us(uint16_t us)
{
    uint16_t dif=10000;
    uint16_t us_con=10000-us;
    __HAL_TIM_SET_COUNTER(&htim1,10000); //设置计数值为 10000
    HAL_TIM_Base_Start(&htim1); //启动定时器
    while(dif>us_con)
    {
        dif=__HAL_TIM_GET_COUNTER(&htim1); //获取定时器的计数值
    }
    HAL_TIM_Base_Stop(&htim1); //停止定时器
}

//容量为 16M bit,2M byte,共有 32 个块,512 个扇区
//4Kbytes 为一个扇区,16 个扇区为 1 个
块

//读取 SPI_FLASH 的状态寄存器
//BIT7 6 5 4 3 2 1 0
//SPR RV TB BP2 BP1 BP0 WEL BUSY
//SPR:默认 0,状态寄存器保护位,配合 WP 使用
//TB,BP2,BP1,BP0:FLASH 区域写保护设置
//WEL:写使能锁定
//BUSY:忙标记位(1,忙;0,空闲)
//默认:0x00
uint8_t W25QXX_ReadSR(void)
{
    uint8_t byte=0;
    F_CS_L; //使能器件
    SPI2_ReadWriteByte(W25X_ReadStatusReg); //发送读取状态寄存器命令
    byte=SPI2_ReadWriteByte(0Xff); //读取一个字节
    F_CS_H; //取消片选
    return byte;
}

//写 SPI_FLASH 状态寄存器
//只有 SPR,TB,BP2,BP1,BP0(bit 7,5,4,3,2)可以写!!!
void W25QXX_Write_SR(uint8_t sr)
{
    F_CS_L; //使能器件
    SPI2_ReadWriteByte(W25X_WriteStatusReg); //发送写取状态寄存器命令
}

```

```
SPI2_ReadWriteByte(sr); //写入一个字节
F_CS_H;//取消片选
}

//等待空闲
void W25QXX_Wait_Busy(void)
{
    while((W25QXX_ReadSR()&0x01)==0x01); //等待 BUSY 位清空
}

//SPI_FLASH 写使能
//将 WEL 置位
void W25QXX_Write_Enable(void)
{
    F_CS_L;//使能器件
    SPI2_ReadWriteByte(W25X_WriteEnable); //发送写使能
    F_CS_H;//取消片选
}

//SPI_FLASH 写禁止
//将 WEL 清零
void W25QXX_Write_Disable(void)
{
    F_CS_L;//使能器件
    SPI2_ReadWriteByte(W25X_WriteDisable); //发送写禁止指令
    F_CS_H;//取消片选
}

//读取芯片 ID W25X16 的 ID:0xEF14
uint16_t W25QXX_ReadID(void)
{
    uint16_t Temp = 0;
    F_CS_L; //使能器件
    SPI2_ReadWriteByte(0x90); //发送读取 ID 命令
    SPI2_ReadWriteByte(0x00);
    SPI2_ReadWriteByte(0x00);
    SPI2_ReadWriteByte(0x00);
    Temp|=SPI2_ReadWriteByte(0xFF)<<8;
    Temp|=SPI2_ReadWriteByte(0xFF);
    F_CS_H;//取消片选
    return Temp;
}

//读取 SPI FLASH
//在指定地址开始读取指定长度的数据
```

```

// pBuffer:数据存储区
//ReadAddr:开始读取的地址(24bit)
//NumByteToRead:要读取的字节数(最大 65535)
void W25QXX_Read(uint8_t* pBuffer,uint32_t ReadAddr,uint16_t
NumByteToRead)
{
    uint16_t i;
    F_CS_L;//使能器件
    SPI2_ReadWriteByte(W25X_ReadData);//发送读取命令
    SPI2_ReadWriteByte((uint8_t)((ReadAddr)>>16)); //发送 24bit 地址
    SPI2_ReadWriteByte((uint8_t)((ReadAddr)>>8));
    SPI2_ReadWriteByte((uint8_t)ReadAddr);
    for(i=0;i<NumByteToRead;i++)
    {
        pBuffer[i]=SPI2_ReadWriteByte(0xFF); //循环读数
    }
    F_CS_H;//取消片选
}

//SPI 在一页(0~65535)内写入少于 256 个字节的数据
//在指定地址开始写入最大 256 字节的数据
//pBuffer:数据存储区
//WriteAddr:开始写入的地址(24bit)
//NumByteToWrite:要写入的字节数(最大 256),该数不应该超过该页的剩余字节数!!!
void W25QXX_Write_Page(uint8_t* pBuffer,uint32_t WriteAddr,uint16_t
NumByteToWrite)
{
    uint16_t i;
    W25QXX_Write_Enable(); //SET WEL
    F_CS_L; //使能器件
    SPI2_ReadWriteByte(W25X_PageProgram); //发送写页命令
    SPI2_ReadWriteByte((uint8_t)((WriteAddr)>>16)); //发送 24bit 地址
    SPI2_ReadWriteByte((uint8_t)((WriteAddr)>>8));
    SPI2_ReadWriteByte((uint8_t)WriteAddr);
    for(i=0;i<NumByteToWrite;i++) SPI2_ReadWriteByte(pBuffer[i]); //循环写数
    F_CS_H;//取消片选
    W25QXX_Wait_Busy(); //等待写入结束
}

//无检验写 SPI FLASH
//必须确保所写的地址范围内的数据全部为 0xFF,否则在非 0xFF 处写入的数据将失败!
//具有自动换页功能
//在指定地址开始写入指定长度的数据,但是要确保地址不越界!
//pBuffer:数据存储区
//WriteAddr:开始写入的地址(24bit)

```

```

//NumByteToWrite:要写入的字节数(最大 65535)
//CHECK OK
void W25QXX_Write_NoCheck(uint8_t* pBuffer,uint32_t WriteAddr,uint16_t NumByteToWrite)
{
    uint16_t pageremain;
    pageremain=256-WriteAddr%256; //单页剩余的字节数
    if(NumByteToWrite<=pageremain)pageremain=NumByteToWrite;//不大于 256 个字节
    while(1)
    {
        W25QXX_Write_Page(pBuffer,WriteAddr,pageremain);
        if(NumByteToWrite==pageremain)break;//写入结束了
        else //NumByteToWrite>pageremain
        {
            pBuffer+=pageremain;
            WriteAddr+=pageremain;

            NumByteToWrite-=pageremain;           //减去已经写入了的字节数
            if(NumByteToWrite>256)pageremain=256; //一次可以写入 256 个字节
            else pageremain=NumByteToWrite;      //不够 256 个字节了
        }
    };
}

//写 SPI FLASH
//在指定地址开始写入指定长度的数据
//该函数带擦除操作!
//pBuffer:数据存储区
//WriteAddr:开始写入的地址(24bit)
//NumByteToWrite:要写入的字节数(最大 65535)
uint8_t W25QXX_BUFFER[4096];
void W25QXX_Write(uint8_t* pBuffer,uint32_t WriteAddr,uint16_t NumByteToWrite)
{
    uint32_t secpos;
    uint16_t secoff;
    uint16_t secremain;
    uint16_t i;

    secpos=WriteAddr/4096;//扇区地址 0~511 for w25x16
    secoff=WriteAddr%4096;//在扇区内的偏移
    secremain=4096-secoff;//扇区剩余空间大小
}

```

```

    if(NumByteToWrite<=secremain)secremain=NumByteToWrite;//不大于 4096 个字
    节
    while(1)
    {
        W25QXX_Read(W25QXX_BUFFER,secpos*4096,4096);//读出整个扇区的内容
        for(i=0;i<secremain;i++)//校验数据
        {
            if(W25QXX_BUFFER[secoff+i]!=0xFF)break;//需要擦除
        }
        if(i<secremain)//需要擦除
        {
            W25QXX_Erase_Sector(secpos);//擦除这个扇区
            for(i=0;i<secremain;i++)//复制
            {
                W25QXX_BUFFER[i+secoff]=pBuffer[i];
            }
            W25QXX_Write_NoCheck(W25QXX_BUFFER,secpos*4096,4096);//写入整个
        扇区
        }
        else W25QXX_Write_NoCheck(pBuffer,WriteAddr,secremain);//写已经擦除了的,
        直接写入扇区剩余区间。
        if(NumByteToWrite==secremain)break;//写入结束了
        else//写入未结束
        {
            secpos++;//扇区地址增 1
            secoff=0;//偏移位置为 0

            pBuffer+=secremain; //指针偏移
            WriteAddr+=secremain;//写地址偏移
            NumByteToWrite-=secremain;//字节数递减
            if(NumByteToWrite>4096)secremain=4096;//下一个扇区还是写不完
            else secremain=NumByteToWrite;//下一个扇区可以写完了
        }
    };
}

//擦除整个芯片
//整片擦除时间:
//W25X16:25s
//W25X32:40s
//W25X64:40s
//等待时间超长...
void W25QXX_Erase_Chip(void)
{

```

```

W25QXX_Write_Enable(); //SET WEL
W25QXX_Wait_Busy();
F_CS_L;//使能器件
SPI2_ReadWriteByte(W25X_ChipErase); //发送片擦除命令
F_CS_H;//取消片选
W25QXX_Wait_Busy();//等待芯片擦除结束
}

//擦除一个扇区
//Dst_Addr:扇区地址 0~511 for w25x16
//擦除一个扇区的最少时间:150ms
void W25QXX_Erase_Sector(uint32_t Dst_Addr)
{
    Dst_Addr*=4096;
    W25QXX_Write_Enable(); //SET WEL
    W25QXX_Wait_Busy();
    F_CS_L;//使能器件
    SPI2_ReadWriteByte(W25X_SectorErase); //发送扇区擦除指令
    SPI2_ReadWriteByte((uint8_t)((Dst_Addr)>>16)); //发送 24bit 地址
    SPI2_ReadWriteByte((uint8_t)((Dst_Addr)>>8));
    SPI2_ReadWriteByte((uint8_t)Dst_Addr);
    F_CS_H;//取消片选
    W25QXX_Wait_Busy();//等待擦除完成
}

//进入掉电模式
void W25QXX_PowerDown(void)
{
    F_CS_L;//使能器件
    SPI2_ReadWriteByte(W25X_PowerDown); //发送掉电命令
    F_CS_H;//取消片选
    Delay_Us(3);//等待 TPD
}

//唤醒
void W25QXX_WAKEUP(void)
{
    F_CS_L;//使能器件
    SPI2_ReadWriteByte(W25X_ReleasePowerDown); //send W25X_PowerDown
    command 0xAB
    F_CS_H;//取消片选
    Delay_Us(3); //等待 TPD
}

```

w25q16.h

```
#ifndef __W25Q16_H
#define __W25Q16_H

#include "main.h"

extern uint8_t W25QXX_BUFFER[4096];

//W25X16 读写指令表
#define W25X_WriteEnable      0x06
#define W25X_WriteDisable     0x04
#define W25X_ReadStatusReg    0x05
#define W25X_WriteStatusReg   0x01
#define W25X_ReadData         0x03
#define W25X_FastReadData    0x0B
#define W25X_FastReadDual    0x3B
#define W25X_PageProgram     0x02
#define W25X_BlockErase       0xD8
#define W25X_SectorErase      0x20
#define W25X_ChipErase        0xC7
#define W25X_PowerDown        0xB9
#define W25X_ReleasePowerDown 0xAB
#define W25X_DeviceID         0xAB
#define W25X_ManufactDeviceID 0x90
#define W25X_JedecDeviceID    0x9F

uint16_t W25QXX_ReadID(void); //读取 FLASH ID

void W25QXX_Read(uint8_t* pBuffer, uint32_t ReadAddr, uint16_t NumByteToRead); //读取 flash
void W25QXX_Write(uint8_t* pBuffer, uint32_t WriteAddr, uint16_t NumByteToWrite); //写入 flash

void W25QXX_Erase_Chip(void); //整片擦除
void W25QXX_Erase_Sector(uint32_t Dst_Addr); //扇区擦除

void W25QXX_PowerDown(void); //进入掉电模式
void W25QXX_WAKEUP(void); //唤醒

#endif
```

### 1.19.5 循环读写实验

GPIO 配置，注意别名

Pin Name	Signal on Pin	GPIO output	GPIO mode	GPIO Pull-up	Maximum ...	User Label	Modified
PB12	n/a	Low	Output Pu...	No pull-up ...	High	F_CS	✓
PB13	n/a	Low	Output Pu...	No pull-up ...	High	SCLK	✓
PB14	n/a	n/a	Input mode	No pull-up ...	n/a	MISO	✓
PB15	n/a	Low	Output Pu...	No pull-up ...	High	MOSI	✓

配置定时器 1

Configure the below parameters :

Search (Ctrl+F) ⌂ ⓘ

Counter Settings

- Prescaler (PSC - 16 bits value) **71**
- Counter Mode Down
- Counter Period (AutoReload Register - 16 bits) 65535
- Internal Clock Division (CKD) No Division
- Repetition Counter (RCR - 8 bits value) 0
- auto-reload preload Disable

打开串口 1

修改 main.c 中主循环

```
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    sprintf(buffer, "\r\nStart Write W25Q16....\r\n");
    HAL_UART_Transmit(&huart1, (uint8_t*)buffer, strlen(buffer), 1000);
    W25QXX_Write((uint8_t*)TEXT_Buffer, FLASH_SIZE-100, SIZE); //从倒数第 100
个地址处开始,写入 SIZE 长度的数据
    sprintf(buffer, "W25Q16 Write Finished!\r\n");
    HAL_UART_Transmit(&huart1, (uint8_t*)buffer, strlen(buffer), 1000);
    HAL_Delay(1000);
    W25QXX_Read(datatemp, FLASH_SIZE-100, SIZE); //从倒数第 100 个地址处开始,
    读出 SIZE 个字节
    sprintf(buffer, "The Data Readed Is: ");
    HAL_UART_Transmit(&huart1, (uint8_t*)buffer, strlen(buffer), 1000);
    HAL_Delay(1000);
    sprintf(buffer, "%s\r\n", datatemp);
    HAL_UART_Transmit(&huart1, (uint8_t*)buffer, strlen(buffer), 1000);
    HAL_Delay(1000);
}
```

实验结果：

```
[2023-12-19 15:35:59.399]# RECV ASCII>  
Start Write W25Q16....  
  
[2023-12-19 15:35:59.713]# RECV ASCII>  
W25Q16 Write Finished!  
  
[2023-12-19 15:36:00.715]# RECV ASCII>  
The Data Readed Is:  
  
[2023-12-19 15:36:01.724]# RECV ASCII>  
W25Q16 TEST
```

## 1.20 看门狗

### 1.20.1 看门狗概述

MCU 微控制器构成的微型计算机系统中，由于微控制器的工作常常会受到来自外界电磁场的干扰，造成各种寄存器和内存的数据混乱，从而导致程序指针错误、不在程序区、取出错误的程序指令等，都有可能会导致程序执行陷入死循环，程序的正常运行被打断，由微控制器控制的系统无法继续正常工作，导致整个系统的陷入停滞状态，发生不可预料的后果。

为了解决以上问题，在微控制器内部集成了一个定时器复位电路，即看门狗电路。

在 stm32 微控制器中集成了两个看门狗外设，分别是独立看门狗和窗口看门狗，提供了更高的安全性，时间的精确性和使用灵活性，两个看门狗设备（独立看门狗、窗口看门狗）可以用来监测和解决由软件错误引起的故障，当计算器达到给定的超时值时，产生系统复位或者触发一个中断（仅适用于窗口看门狗）

### 1.20.2 独立看门狗

由专用的低速时钟（LSI）驱动，即使主时钟发生故障，任能够继续有效。独立看门狗适用于需要看门狗作为一个在主程序之外能够完全独立工作，并且对时间精度要求低的场合。

1、IWDG 的主要特性

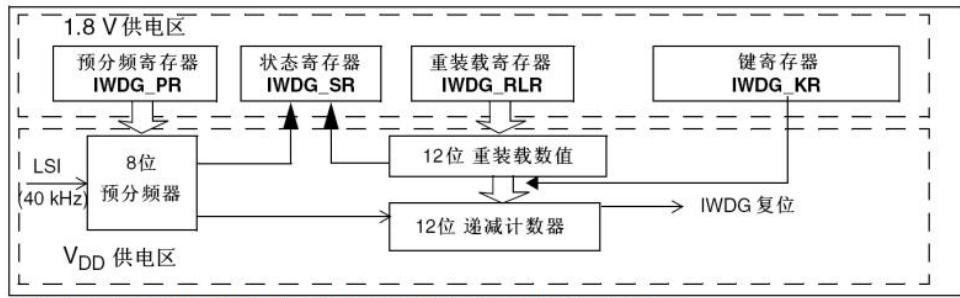
自由运行的递减计数器

时钟由独立的 RC 振荡器提供(可在停止和待机模式下工作)

看门狗被激活后，则在计数器计数至 0x000 时产生复位

2、看门狗框架

图157 独立看门狗框图



注：看门狗功能处于VDD供电区，即在停机和待机模式时仍能正常工作。

表83 看门狗超时时间(40kHz的输入时钟(LSI))<sup>(1)</sup>

预分频系数	PR[2:0]位	最短时间(ms) RL[11:0] = 0x000	最长时间(ms) RL[11:0] = 0xFFFF
/4	0	0.1	409.6
/8	1	0.2	819.2
/16	2	0.4	1638.4
/32	3	0.8	3276.8
/64	4	1.6	6553.6
/128	5	3.2	13107.2
/256	(6或7)	6.4	26214.4

注：这些时间是按照40kHz时钟给出。实际上，MCU内部的RC频率会在30kHz到60kHz之间变化。此外，即使RC振荡器的频率是精确的，确切的时序仍然依赖于APB接口时钟与RC振荡器时钟之间的相位差，因此总会有一个完整的RC周期是不确定的。

通过对LSI进行校准可获得相对精确的看门狗超时时间。有关LSI校准的问题，详见6.2.5节。

### 3、时钟

由LSI提供时钟，时钟频率40KHz，经过预分频器分频后的时钟，提供给12bit递减计数器，作为向下计数的频率。

### 4、预分频寄存器(IWDG\_PR)

预分频器的分频系数由IWDG\_PR预分频寄存器设置：地址偏移：0x04 复位值：0x0000 0000

地址偏移: 0x04

复位值: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16											
保留																										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
保留										PR[2:0]																
<small>rW      rW      rW</small>																										
<table border="1"><tr><td>位31:3</td><td>保留, 始终读为0。</td></tr><tr><td>位2:0</td><td><b>PR[2:0]: 预分频因子 (Prescaler divider)</b> 这些位具有写保护设置, 参见17.3.2节。通过设置这些位来选择计数器时钟的预分频因子。要改变预分频因子, IWDG_SR寄存器的PVU位必须为0。 <table><tr><td>000: 预分频因子=4</td><td>100: 预分频因子=64</td></tr><tr><td>001: 预分频因子=8</td><td>101: 预分频因子=128</td></tr><tr><td>010: 预分频因子=16</td><td>110: 预分频因子=256</td></tr><tr><td>011: 预分频因子=32</td><td>111: 预分频因子=256</td></tr></table><p>注意: 对此寄存器进行读操作, 将从VDD电压域返回预分频值。如果写操作正在进行, 则读回的值可能是无效的。因此, 只有当IWDG_SR寄存器的PVU位为0时, 读出的值才有效。</p></td></tr></table>															位31:3	保留, 始终读为0。	位2:0	<b>PR[2:0]: 预分频因子 (Prescaler divider)</b> 这些位具有写保护设置, 参见17.3.2节。通过设置这些位来选择计数器时钟的预分频因子。要改变预分频因子, IWDG_SR寄存器的PVU位必须为0。 <table><tr><td>000: 预分频因子=4</td><td>100: 预分频因子=64</td></tr><tr><td>001: 预分频因子=8</td><td>101: 预分频因子=128</td></tr><tr><td>010: 预分频因子=16</td><td>110: 预分频因子=256</td></tr><tr><td>011: 预分频因子=32</td><td>111: 预分频因子=256</td></tr></table> <p>注意: 对此寄存器进行读操作, 将从VDD电压域返回预分频值。如果写操作正在进行, 则读回的值可能是无效的。因此, 只有当IWDG_SR寄存器的PVU位为0时, 读出的值才有效。</p>	000: 预分频因子=4	100: 预分频因子=64	001: 预分频因子=8	101: 预分频因子=128	010: 预分频因子=16	110: 预分频因子=256	011: 预分频因子=32	111: 预分频因子=256
位31:3	保留, 始终读为0。																									
位2:0	<b>PR[2:0]: 预分频因子 (Prescaler divider)</b> 这些位具有写保护设置, 参见17.3.2节。通过设置这些位来选择计数器时钟的预分频因子。要改变预分频因子, IWDG_SR寄存器的PVU位必须为0。 <table><tr><td>000: 预分频因子=4</td><td>100: 预分频因子=64</td></tr><tr><td>001: 预分频因子=8</td><td>101: 预分频因子=128</td></tr><tr><td>010: 预分频因子=16</td><td>110: 预分频因子=256</td></tr><tr><td>011: 预分频因子=32</td><td>111: 预分频因子=256</td></tr></table> <p>注意: 对此寄存器进行读操作, 将从VDD电压域返回预分频值。如果写操作正在进行, 则读回的值可能是无效的。因此, 只有当IWDG_SR寄存器的PVU位为0时, 读出的值才有效。</p>	000: 预分频因子=4	100: 预分频因子=64	001: 预分频因子=8	101: 预分频因子=128	010: 预分频因子=16	110: 预分频因子=256	011: 预分频因子=32	111: 预分频因子=256																	
000: 预分频因子=4	100: 预分频因子=64																									
001: 预分频因子=8	101: 预分频因子=128																									
010: 预分频因子=16	110: 预分频因子=256																									
011: 预分频因子=32	111: 预分频因子=256																									

## 5、键寄存器

地址偏移: 0x00

复位值: 0x0000 0000 (在待机模式复位)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16				
保留																			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
KEY[15:0]																			
<small>w      w      w      w      w      w      w      w      w      w      w      w      w      w      w      w</small>																			
<table border="1"><tr><td>位31:16</td><td>保留, 始终读为0。</td></tr><tr><td>位15:0</td><td><b>KEY[15:0]: 键值(只写寄存器, 读出值为0x0000) (Key value)</b> 软件必须以一定的间隔写入0xAAAA, 否则, 当计数器为0时, 看门狗会产生复位。 写入0x5555表示允许访问IWDG_PR和IWDG_RLR寄存器。(见17.3.2节) 写入0xFFFF, 启动看门狗工作(若选择了硬件看门狗则不受此命令字限制)。</td></tr></table>																位31:16	保留, 始终读为0。	位15:0	<b>KEY[15:0]: 键值(只写寄存器, 读出值为0x0000) (Key value)</b> 软件必须以一定的间隔写入0xAAAA, 否则, 当计数器为0时, 看门狗会产生复位。 写入0x5555表示允许访问IWDG_PR和IWDG_RLR寄存器。(见17.3.2节) 写入0xFFFF, 启动看门狗工作(若选择了硬件看门狗则不受此命令字限制)。
位31:16	保留, 始终读为0。																		
位15:0	<b>KEY[15:0]: 键值(只写寄存器, 读出值为0x0000) (Key value)</b> 软件必须以一定的间隔写入0xAAAA, 否则, 当计数器为0时, 看门狗会产生复位。 写入0x5555表示允许访问IWDG_PR和IWDG_RLR寄存器。(见17.3.2节) 写入0xFFFF, 启动看门狗工作(若选择了硬件看门狗则不受此命令字限制)。																		

IWDG\_PR 和 IWDG\_RLR 寄存器具有写保护功能。要修改这两个寄存器的值, 必须先向 IWDG\_KR 寄存器中写入 0x5555。重装载操作(即写入 0xFFFF)也会启动写保护功能。

## 6、重装载寄存器

地址偏移: 0x08

复位值: 0x0000 0FFF(待机模式时复位)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留		RL[11:0]													
		RW													
位31:12		保留, 始终读为0。													
位11:0		<b>RL[11:0]: 看门狗计数器重装载值 (Watchdog counter reload value)</b> 这些位具有写保护功能, 参看17.3.2节。用于定义看门狗计数器的重装载值, 每当向IWDG_KR寄存器写入0xAAAA时, 重装载值会被传送到计数器中。随后计数器从这个值开始递减计数。 看门狗超时周期可通过此重装载值和时钟预分频值来计算, 参照表83。 只有当IWDG_SR寄存器中的RVU位为0时, 才能对此寄存器进行修改。 注: 对此寄存器进行读操作, 将从VDD电压域返回预分频值。如果写操作正在进行, 则读回的值可能是无效的。因此, 只有当IWDG_SR寄存器的RVU位为0时, 读出的值才有效。													

## 7、功能总结

在键寄存器(IWDG\_KR)中写入 0xCCCC, 开始启用独立看门狗; 此时计数器开始从其复位值 0xFFFF 递减计数。当计数器计数到末尾 0x000 时, 会产生一个复位信号 (IWDG\_RESET)。无论何时, 只要在键寄存器 IWDG\_KR 中写入 0xAAAA, IWDG\_RLR 中的值就会被重新加载到计数器, 从而避免产生看门狗复位。

## 8、功能总结

### HAL\_IWDG\_Init

```
/**  
 * 函数功能: 初始化和启动独立看门狗 (IWDG)。  
 *  
 * HAL_StatusTypeDef HAL_IWDG_Init(IWDG_HandleTypeDef *hiwdg)  
 *  
 * 参数:  
 *     hiwdg - 指向 IWDG_HandleTypeDef 结构的指针, 该结构包含指定 IWDG 模块的配置  
 *             信息。  
 *     返回值: HAL 状态。  
 */
```

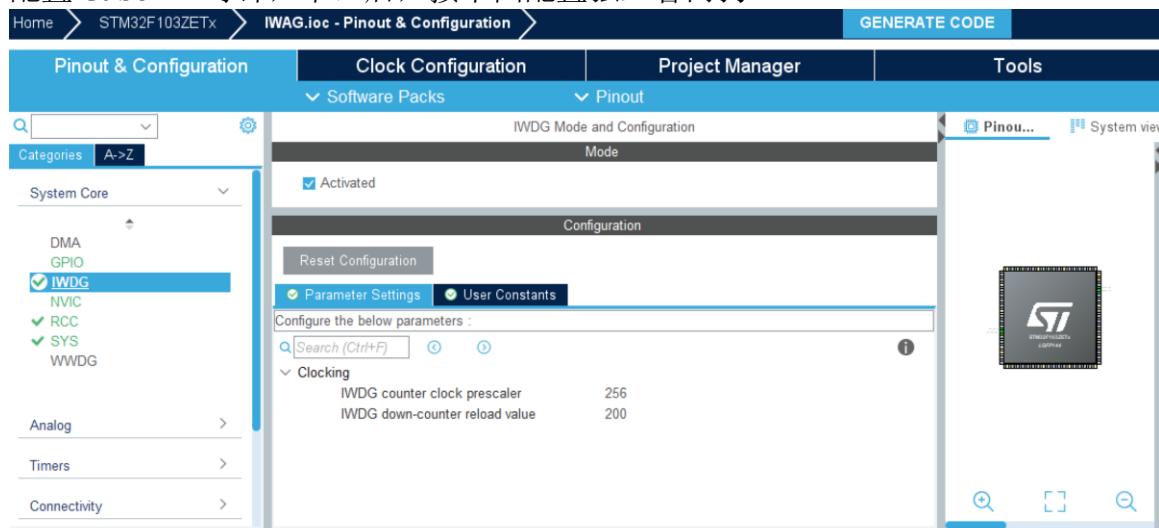
### HAL\_IWDG\_Refresh

```
/**  
 * 函数功能: 刷新独立看门狗 (IWDG)。  
 *  
 * HAL_StatusTypeDef HAL_IWDG_Refresh(IWDG_HandleTypeDef *hiwdg)  
 *  
 * 参数:  
 *     hiwdg - 指向 IWDG_HandleTypeDef 结构的指针, 该结构包含指定 IWDG 模块的配置  
 *             信息。
```

```
* 返回值: HAL 状态。  
*/
```

### 1.20.3 独立看门狗案例

配置 CubeMX 时钟，串口后，按下图配置独立看门狗



配置后生成工程，相关函数解析如下

```
void MX_IWDG_Init(void)  
{  
    /* 用户代码区域 IWDG_Init_0 开始 */  
  
    /* 用户代码区域 IWDG_Init_0 结束 */  
  
    /* 用户代码区域 IWDG_Init_1 开始 */  
  
    /* 用户代码区域 IWDG_Init_1 结束 */  
  
    // 设置 IWDG 实例  
    hiwdg.Instance = IWDG;  
  
    // 初始化 IWDG 的配置参数  
    hiwdg.Init.Prescaler = IWDG_PRESCALER_256; // 设置预分频器为 256  
    hiwdg.Init.Reload = 200; // 设置重装载值为 200  
  
    // 调用 HAL_IWDG_Init 函数初始化 IWDG  
    if (HAL_IWDG_Init(&hiwdg) != HAL_OK)  
    {  
        Error_Handler(); // 如果初始化失败，则调用错误处理函数  
    }  
}
```

```

/* 用户代码区域 IWDG_Init_2 开始 */

/* 用户代码区域 IWDG_Init_2 结束 */
}

```

根据相关公式：超时时间 = (预分频器 \* 重装载值) / LSI 频率

预分频器（Prescaler）设置为 256

重装载值（Reload）设置为 200

低速内部时钟（LSI）频率为 40 kHz

计算得到的超时时间为 1.28 秒。

在 main.c 中

```

/* USER CODE BEGIN 2 */
HAL_UART_Transmit(&huart1, "IOT\r\n", strlen("IOT\r\n"), 1000);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */
    HAL_UART_Transmit(&huart1, "HELLO\r\n", strlen("HELLO\r\n"), 1000);
    HAL_Delay(2000);
    HAL_IWDG_Refresh(&hiwdg);
    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */

```

实验结果：

此时 2s 刷新一次看门狗，大于超时时间，系统复位，再次打印 IOT。



修改主循环中延时为 1000

```
while (1)
{
    /* USER CODE END WHILE */
    HAL_UART_Transmit(&huart1, "HELLO\r\n", strlen("HELLO\r\n"), 1000);
    HAL_Delay(1000);
    HAL_IWDG_Refresh(&hiwdg);
    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

此时 1s 刷新一次看门狗，大于超时时间，系统不复位，只打印一次 IOT。



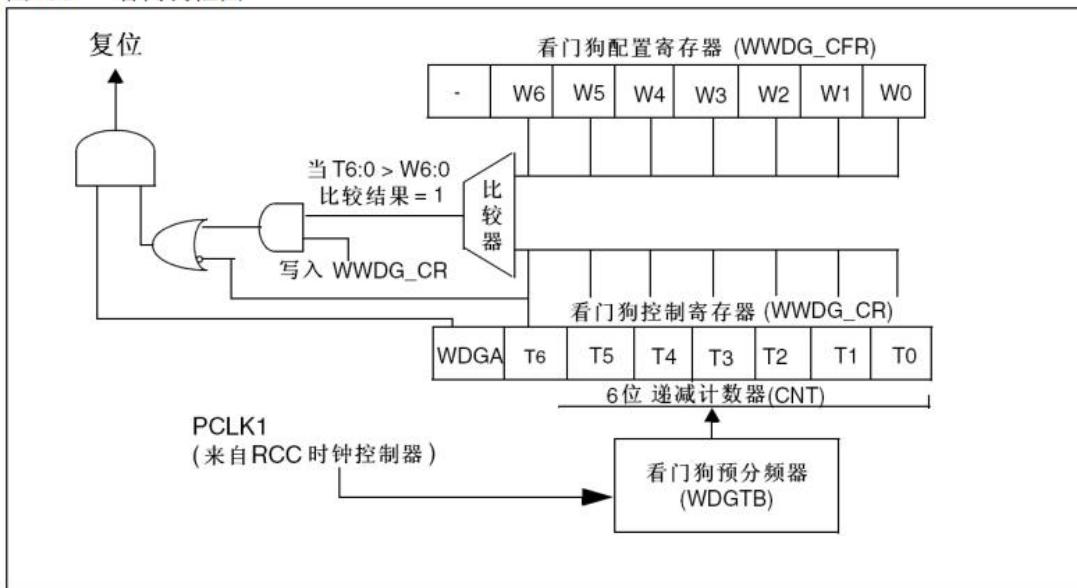
#### 1.20.4 窗口看门狗

通常被用来监测，由外部干扰或不可预见的逻辑条件造成应用程序背离正常的运行序列而产生的软件故障。

由从 APB1 时钟分频后得到的时钟驱动，通过可配置的时间窗口来监测应用程序非正常的过迟或过早操作。窗口看门狗最合适那些要求看门狗在精确计时窗口起作用的程序。

#### 1、WWDG 框架图

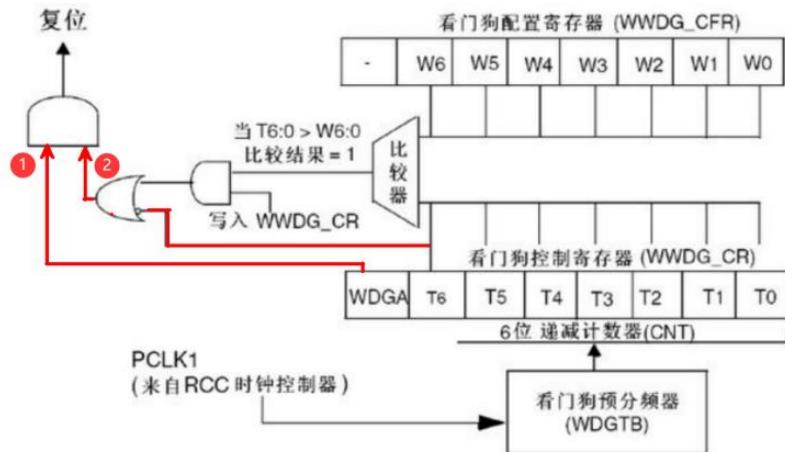
图158 看门狗框图



320/754

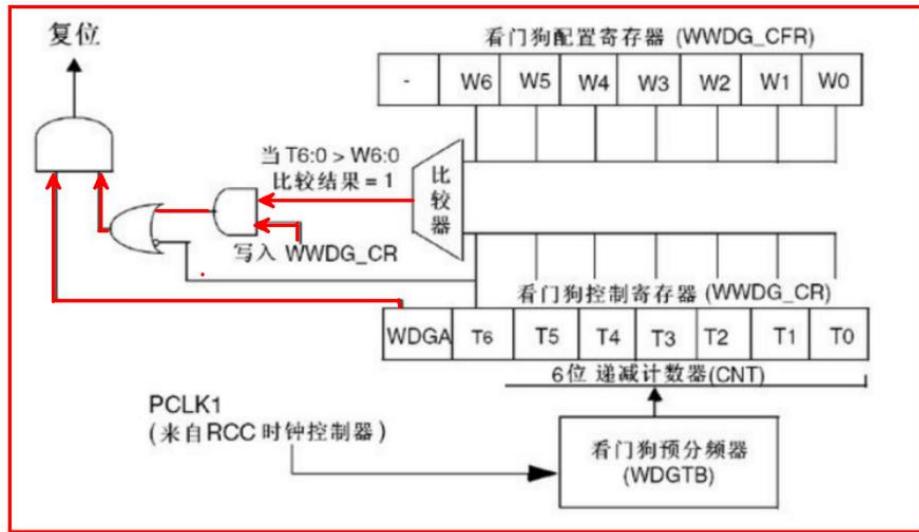
产生复位的两种情况：

第一种：



1 和 2 经过与门后，产生复位。即：WDGA 位为 1，T6 为 0（取反后为 1），经过或门电路后路径 2 为 1，也就是 WWDG\_CR 寄存器递减到 0x40 后，再减 1，编程 0x3F 的时候，T6 位，由 1 变为 0。

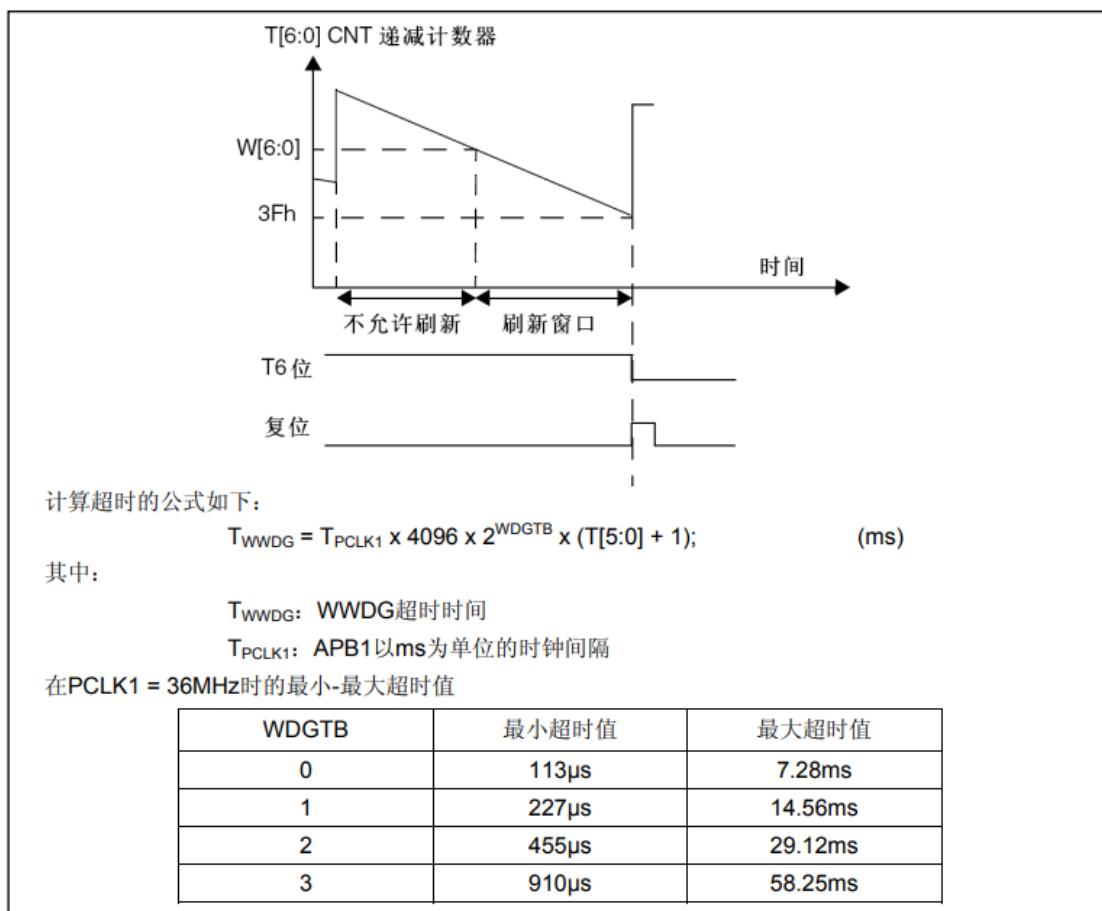
第二种：



WDGA 位为 1 时，当  $T6:0 > W6:0$  且写入 WWDG\_CR (即刷新计数值) 产生复位中断。

WWDG 时序：

图159 窗口看门狗时序图



配置寄存器(WWDG\_CFR)中包含窗口的上限值：要避免产生复位，递减计数器必须在其值小于窗口寄存器的数值并且大于 0x3F 时被重新装载。

## 2、寄存器

#### 配置寄存器(WWDG\_CFR)

WWDG 时钟来自于 PCLK1 (36MHz)，由窗口看门狗 WDGTR 预分频器分频后，提供给 6bit 递减计数器作为向下计数得频率。

地址偏移量: 0x04

复位值: 0x7F

## 控制寄存器(WWDG CR)

地址偏移量: 0x00

复位值: 0x7F

### 3、相关 API

HAL\_WWDG\_Init

```
/**  
 * 函数功能：初始化窗口看门狗（WWDG）。  
 *  
 * HAL_StatusTypeDef HAL_WWDG_Init(WWDG_HandleTypeDef *hwdg)  
 *  
 * 参数：  
 *     hwdg - 指向 WWDG_HandleTypeDef 结构的指针，该结构包含指定 WWDG 模块的配置  
 * 信息。  
 *     返回值：HAL 状态。  
 */
```

HAL\_WWDG\_MspInit

```
/**  
 * 函数功能：初始化窗口看门狗（WWDG）的 MSP。  
 *  
 * void HAL_WWDG_MspInit(WWDG_HandleTypeDef *hwdg)  
 *  
 * 参数：  
 *     hwdg - 指向 WWDG_HandleTypeDef 结构的指针，该结构包含指定 WWDG 模块的配置  
 * 信息。  
 *     返回值：无。  
 */
```

HAL\_WWDG\_RegisterCallback

```
/**  
 * 函数功能：注册用户窗口看门狗（WWDG）回调函数。  
 *  
 * HAL_StatusTypeDef HAL_WWDG_RegisterCallback(WWDG_HandleTypeDef *hwdg,  
 *                                              HAL_WWDG_CallbackIDTypeDef CallbackID,  
 *                                              pWWDG_CallbackTypeDef  
 * pCallback)  
 *  
 * 参数：  
 *     hwdg - WWDG 句柄。  
 *     CallbackID - 要注册的回调函数的 ID。  
 *     pCallback - 指向回调函数的指针。  
 *     返回值：状态。  
 */
```

HAL\_WWDG\_UnRegisterCallback

```
/**  
 * 函数功能：取消注册窗口看门狗（WWDG）回调函数。  
 *  
 * HAL_StatusTypeDef HAL_WWDG_UnRegisterCallback(WWDG_HandleTypeDef *hwdg,  
 HAL_WWDG_CallbackIDTypeDef CallbackID)  
 *  
 * 参数：  
 *      hwdg - WWDG 句柄。  
 *      CallbackID - 要取消注册的回调函数的 ID。  
 * 返回值：状态。  
 */
```

## HAL\_WWDG\_Refresh

```
/**  
 * 函数功能：刷新窗口看门狗（WWDG）。  
 *  
 * HAL_StatusTypeDef HAL_WWDG_Refresh(WWDG_HandleTypeDef *hwdg)  
 *  
 * 参数：  
 *      hwdg - 指向 WWDG_HandleTypeDef 结构的指针，该结构包含指定 WWDG 模块的配置  
信息。  
 *      返回值：HAL 状态。  
 */
```

## HAL\_WWDG\_IRQHandler

```
/**  
 * 函数功能：处理窗口看门狗（WWDG）中断请求。  
 *  
 * void HAL_WWDG_IRQHandler(WWDG_HandleTypeDef *hwdg)  
 *  
 * 参数：  
 *      hwdg - 指向 WWDG_HandleTypeDef 结构的指针，该结构包含指定 WWDG 模块的配置  
信息。  
 *      返回值：无。  
 */
```

## HAL\_WWDG\_EarlyWakeupCallback

```
/**  
 * 函数功能：窗口看门狗（WWDG）提前唤醒回调函数。  
 */
```

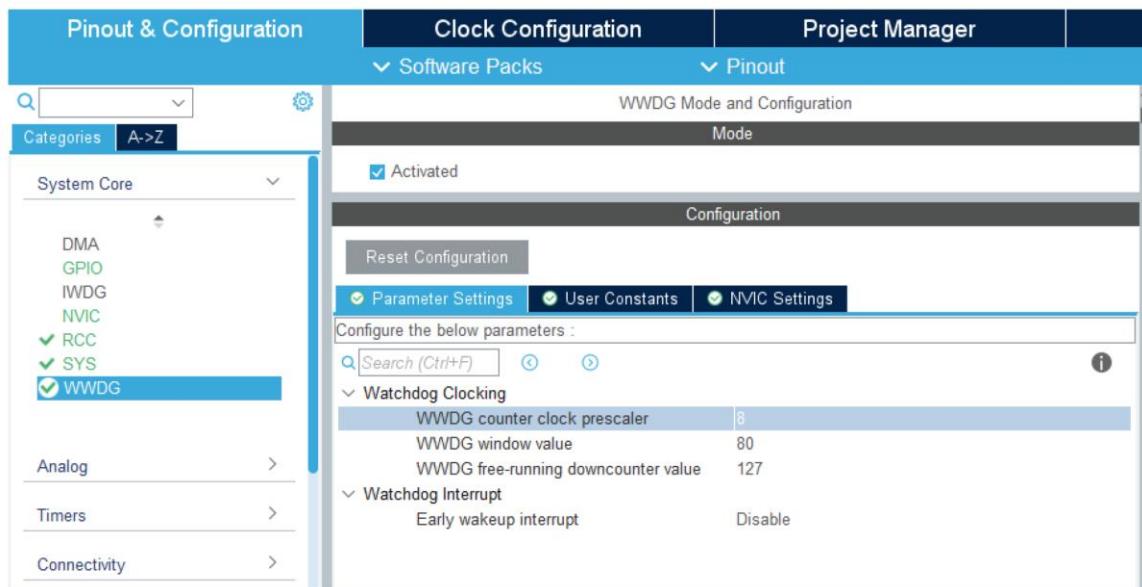
```

* void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef *hwdg)
*
* 参数:
*   hwdg - 指向 WWDG_HandleTypeDef 结构的指针, 该结构包含指定 WWDG 模块的配置
信息。
*   返回值: 无。
*/

```

### 1.20.5 窗口看门狗案例

配置 CubeMX 时钟，串口后，按下图配置窗口看门狗



生成工程后，相关函数解析如下：

```

/**
 * 函数功能：初始化窗口看门狗（WWDG）。
 */
void MX_WWDG_Init(void)
{
    /* 用户代码区域 WWDG_Init_0 开始 */

    /* 用户代码区域 WWDG_Init_0 结束 */

    /* 用户代码区域 WWDG_Init_1 开始 */

    /* 用户代码区域 WWDG_Init_1 结束 */

    // 设置 WWDG 实例
}

```

```

hwwdg.Instance = WWDG;

// 初始化 WWDG 的配置参数
hwwdg.Init.Prescaler = WWDG_PRESCALER_8; // 设置预分频器为 8
hwwdg.Init.Window = 80; // 设置窗口值为 80
hwwdg.Init.Counter = 127; // 设置计数器的值为 127
hwwdg.Init.EWIMode = WWDG_EWI_DISABLE; // 设置提前唤醒中断（EWI）为禁用

// 调用 HAL_WWDG_Init 函数初始化 WWDG
if (HAL_WWDG_Init(&hwwdg) != HAL_OK)
{
    Error_Handler(); // 如果初始化失败，则调用错误处理函数
}

/* 用户代码区域 WWDG_Init_2 开始 */

/* 用户代码区域 WWDG_Init_2 结束 */
}

/**
 * 函数功能：初始化窗口看门狗（WWDG）的 MSP。
 */
void HAL_WWDG_MspInit(WWDG_HandleTypeDef* wwdgHandle)
{
    if(wwdgHandle->Instance == WWDG)
    {
        /* 用户代码区域 WWDG_MspInit_0 开始 */

        /* 用户代码区域 WWDG_MspInit_0 结束 */

        // 使能 WWDG 时钟
        __HAL_RCC_WWDG_CLK_ENABLE();

        /* 用户代码区域 WWDG_MspInit_1 开始 */

        /* 用户代码区域 WWDG_MspInit_1 结束 */
    }
}

```

根据上述参数计算窗口看门狗(WWDG)的超时时间

prescaler = 8 # 预分频器设置为 8

window = 80 # 窗口值设置为 80

counter = 127 # 计数器设置为 127

```

pclk1_frequency = 36000000 # APB1 时钟频率, 为 36MHz

# WWDG 的时钟频率计算公式

# WWDG clock (Hz) = PCLK1 / (4096 * Prescaler)

wwdg_clock = pclk1_frequency / (4096 * prescaler)

# WWDG 的最大时间计算公式

# max time (ms) = 1000 * (Counter - 0x40) / WWDG clock

max_timeout = 1000 * (counter - 0x40) / wwdg_clock

max_timeout 为 57.344 毫秒

# WWDG 的最小时间计算公式

# min time (ms) = 1000 * (Counter - Window) / WWDG clock

min_timeout = 1000 * (counter - window) / wwdg_clock

min_timeout 为 42.78 毫秒

```

在 main.c 中

```

/* USER CODE BEGIN 2 */
HAL_UART_Transmit(&huart1, "IOT\r\n", strlen("IOT\r\n"), 1000);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    HAL_UART_Transmit(&huart1, "HELLO\r\n", strlen("HELLO\r\n"), 1000);
    HAL_Delay(38);
    HAL_WWDG_Refresh(&hwdg);
/* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */

```

此时 38 毫秒刷新一次看门狗，小于最小时间，系统复位,再次打印 IOT。

```

[2023-12-08 10:37:24.728]# RECV ASCII>
?IOT
HELLO
?IOT
HELLO
?IOT
HELLO

```

修改主循环中延时为 50

```
while (1)
{
    HAL_UART_Transmit(&huart1, "HELLO\r\n", strlen("HELLO\r\n"), 1000);
    HAL_Delay(50);
    HAL_WWDG_Refresh(&hwdg);
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}
```

此时 50 毫秒刷新一次看门狗，大于最长时间，小于最大时间，系统不会复位,不会再次打印 IOT。

```
[2023-12-08 10:41:45.196]# RECV ASCII>
IOT
HELLO
```

```
[2023-12-08 10:41:45.273]# RECV ASCII>
HELLO
HELLO
```

```
[2023-12-08 10:41:45.321]# RECV ASCII>
HELLO
```

修改主循环中延时为 65

```
while (1)
{
    HAL_UART_Transmit(&huart1, "HELLO\r\n", strlen("HELLO\r\n"), 1000);
    HAL_Delay(65);
    HAL_WWDG_Refresh(&hwdg);
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}
```

此时 65 毫秒刷新一次看门狗，大于最大时间，系统复位,再次打印 IOT。

```
IOT  
HELLO  
  
[2023-12-08 10:42:48.643]# RECV ASCII>  
IOT  
HELLO  
  
[2023-12-08 10:42:48.705]# RECV ASCII>  
IOT  
HELLO
```

## 1.21 模数转换 ADC

### 1.21.1 ADC 概述

ADC(Analog-to-Digital Converter) 指模数转换器。是指将连续变化的模拟信号转换为离散的数字信号的器件。ADC 相关参数说明：

分辨率： 分辨率以二进制（或十进制）数的位数来表示，一般有 8 位、10 位、12 位、16 位等，它说明模数转换器对输入信号的分辨能力，位数越多，表示分辨率越高，恢复模拟信号时会更精确。精度： 精度表示 ADC 器件在所有的数值点上对应的模拟值和真实值之间的最大误差值，也就是输出数值偏离线性最大的距离。

转换速率： 转换速率是指 A/D 转换器完成一次从模拟到数字的 AD 转换所需时间的倒数。例如，某 A/D 转换器的转换速率为 1MHz，则表示完成一次 AD 转换时间为 1 微秒。

### 1.21.2 ADC 结构体详解

```
/**  
 * @brief ADC 句柄结构定义  
 */  
typedef struct __ADC_HandleTypeDef  
{  
    ADC_TypeDef                *Instance;           /*!< 寄存器地址  
 */  
    ADC_InitTypeDef            Init;                 /*!< ADC 所需参数 */  
    DMA_HandleTypeDef          *DMA_Handle;         /*!< DMA 处理器的指  
针 */  
    HAL_LockTypeDef            Lock;                /*!< ADC 锁定对象 */
```

```

__IO uint32_t State; /*!< ADC 通信状态
(ADC 状态的位图) */

__IO uint32_t ErrorCode; /*!< ADC 错误代码 */

#if (USE_HAL_ADC_REGISTER_CALLBACKS == 1)
  void (*ConvCpltCallback)(struct __ADC_HandleTypeDefDef
*hadc); /*!< ADC 转换完成回调函数 */
  void (*ConvHalfCpltCallback)(struct __ADC_HandleTypeDefDef
*hadc); /*!< ADC 转换 DMA 半传输完成回调函数 */
  void (*LevelOutOfWindowCallback)(struct __ADC_HandleTypeDefDef
*hadc); /*!< ADC 模拟看门狗 1 回调函数 */
  void (*ErrorCallback)(struct __ADC_HandleTypeDefDef
*hadc); /*!< ADC 错误回调函数 */
  void (*InjectedConvCpltCallback)(struct __ADC_HandleTypeDefDef
*hadc); /*!< ADC 注入组转换完成回调函数 */
  void (*MspInitCallback)(struct __ADC_HandleTypeDefDef
*hadc); /*!< ADC MSP 初始化回调函数 */
  void (*MspDeInitCallback)(struct __ADC_HandleTypeDefDef
*hadc); /*!< ADC MSP 去初始化回调函数 */
#endif /* USE_HAL_ADC_REGISTER_CALLBACKS */
}ADC_HandleTypeDef;

```

其中 ADC\_InitTypeDef 参数详细解析如下：

#### **uint32\_t DataAlign:**

指定 ADC 数据对齐方式。右对齐（默认设置，最高有效位在寄存器第 11 位，最低有效位在寄存器第 0 位）或左对齐（如果是常规组：最高有效位在寄存器第 15 位，最低有效位在寄存器第 4 位；如果是注入组（最高有效位保持为有符号值，由于应用偏移后可能为负值）：最高有效位在寄存器第 14 位，最低有效位在寄存器第 3 位）。此参数可以是 @ref ADC\_Data\_align 的值。

#### **uint32\_t ScanConvMode:**

配置常规和注入组的顺序器。此参数可以与 'DiscontinuousConvMode' 参数结合使用，以将主序列细分为连续部分。如果禁用：在单模式下进行转换（转换一个通道，即在排名 1 中定义的那个）。如果启用：在序列模式下进行转换（由 'NbrOfConversion'/'InjectedNbrOfConversion' 定义的多个排名，以及每个通道排名）。此参数可以是 @ref ADC\_Scan\_mode 的值。

#### **FunctionalState ContinuousConvMode:**

指定常规组的转换是在单模式下（一次转换）还是在连续模式下进行，在选定的触发器发生后（软件启动或外部触发器）。此参数可以设置为 ENABLE 或 DISABLE。

**uint32\_t NbrOfConversion:** 指定将在常规组顺序器中转换的排名数量。要使用常规组顺序器并转换多个排名，必须启用参数 'ScanConvMode'。此参数必须是介于 Min\_Data = 1 和 Max\_Data = 16 之间的数字。

#### **FunctionalState DiscontinuousConvMode:**

指定常规组的转换序列是以完整序列/非连续序列的方式进行（主序列细分为连续部分）。只有在启用顺序器（参数 'ScanConvMode'）时才使用非连续模式。如果顺序器被禁用，此参数将被忽略。只有在连续模式被禁用时，才能启用非连续模式。如果连续模式被启用，此参数设置将被忽略。此参数可以设置为 ENABLE 或 DISABLE。

#### **uint32\_t NbrOfDiscConversion:**

指定常规组主序列（参数 NbrOfConversion）将细分为多少个非连续转换。如果参数 'DiscontinuousConvMode' 被禁用，此参数将被忽略。此参数必须是介于 Min\_Data = 1 和 Max\_Data = 8 之间的数字。

#### **uint32\_t ExternalTrigConv:**

选择用于触发常规组转换开始的外部事件。如果设置为 ADC\_SOFTWARE\_START，则禁用外部触发器。如果设置为外部触发源，则触发在事件上升沿。此参数可以是 @ref ADC\_External\_trigger\_source-Regular 的值。

### 1.21.3 ADC 相关 API

#### HAL\_ADC\_Init

```
/**  
 * 函数功能：初始化 ADC 外设和常规组。  
 *  
 * HAL_StatusTypeDef HAL_ADC_Init(ADC_HandleTypeDef* hadc)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针，该结构包含指定 ADC 模块的配置信息。  
 *      返回值：HAL 状态。  
 */
```

#### HAL\_ADC\_DeInit

```
/**  
 * 函数功能：将 ADC 外设寄存器重置为默认值，并进行去初始化。  
 *  
 * HAL_StatusTypeDef HAL_ADC_DeInit(ADC_HandleTypeDef* hadc)  
 */
```

```
* 参数:  
*      hadc - 指向 ADC_HandleTypeDef 结构的指针, 该结构包含指定 ADC 模块的配置信息。  
* 返回值: HAL 状态。  
*/
```

## HAL\_ADC\_MspInit

```
/**  
 * 函数功能: 初始化 ADC MSP。  
 *  
 * __weak void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc)  
 *  
 * 参数:  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针, 该结构包含指定 ADC 模块的配置信息。  
 * 返回值: 无。  
 */
```

## HAL\_ADC\_MspDeInit

```
/**  
 * 函数功能: 去初始化 ADC MSP。  
 *  
 * __weak void HAL_ADC_MspDeInit(ADC_HandleTypeDef* hadc)  
 *  
 * 参数:  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针, 该结构包含指定 ADC 模块的配置信息。  
 * 返回值: 无。  
 */
```

## HAL\_ADC\_UnRegisterCallback

```
/**  
 * 函数功能: 注销一个 ADC 回调函数。  
 *  
 * HAL_StatusTypeDef HAL_ADC_UnRegisterCallback(ADC_HandleTypeDef *hadc,  
 * HAL_ADC_CallbackIDTypeDef CallbackID)  
 *  
 * 参数:  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 *      CallbackID - 要注销的回调函数的 ID。  
 * 返回值: HAL 状态。  
 */
```

## HAL\_ADC\_Start

```
/**  
 * 函数功能：启用 ADC，开始常规组转换。此函数中未启用中断。  
 *  
 * HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef* hadc)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 * 返回值：HAL 状态。  
 */
```

### HAL\_ADC\_Stop

```
/**  
 * 函数功能：停止常规组转换，禁用 ADC 外设。  
 *  
 * HAL_StatusTypeDef HAL_ADC_Stop(ADC_HandleTypeDef* hadc)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 * 返回值：HAL 状态。  
 */
```

### HAL\_ADC\_PollForConversion

```
/**  
 * 函数功能：等待常规组转换完成。  
 *  
 * HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc,  
 uint32_t Timeout)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 *      Timeout - 超时值，单位为毫秒。  
 * 返回值：HAL 状态。  
 */
```

### HAL\_ADC\_PollForEvent

```
/**  
 * 函数功能：轮询转换事件。  
 *  
 * HAL_StatusTypeDef HAL_ADC_PollForEvent(ADC_HandleTypeDef* hadc,  
 uint32_t EventType, uint32_t Timeout)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 *      EventType - ADC 事件类型。
```

```
*      Timeout - 超时值，单位为毫秒。  
* 返回值: HAL 状态。  
*/
```

### HAL\_ADC\_Start\_IT

```
/**  
 * 函数功能: 启用 ADC, 开始常规组转换并启用中断。  
 *  
 * HAL_StatusTypeDef HAL_ADC_Start_IT(ADC_HandleTypeDef* hadc)  
 *  
 * 参数:  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 * 返回值: HAL 状态。  
 */
```

### HAL\_ADC\_Stop\_IT

```
/**  
 * 函数功能: 停止常规组转换并禁用中断, 禁用 ADC 外设。  
 *  
 * HAL_StatusTypeDef HAL_ADC_Stop_IT(ADC_HandleTypeDef* hadc)  
 *  
 * 参数:  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 * 返回值: HAL 状态。  
 */
```

### HAL\_ADC\_Start\_DMA

```
/**  
 * 函数功能: 启用 ADC, 开始常规组转换并通过 DMA 传输结果。  
 *  
 * HAL_StatusTypeDef HAL_ADC_Start_DMA(ADC_HandleTypeDef* hadc, uint32_t*  
 pData, uint32_t Length)  
 *  
 * 参数:  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 *      pData - 目标缓冲区地址。  
 *      Length - 从 ADC 外设到内存的数据传输长度。  
 * 返回值: HAL 状态。  
 */
```

### HAL\_ADC\_Stop\_DMA

```
/**  
 * 函数功能: 停止常规组转换, 禁用 ADC DMA 传输, 禁用 ADC 外设。  
 */
```

```
* HAL_StatusTypeDef HAL_ADC_Stop_DMA(ADC_HandleTypeDef* hadc)
*
* 参数:
*   hadc - 指向 ADC_HandleTypeDef 结构的指针。
* 返回值: HAL 状态。
*/
```

#### HAL\_ADC\_GetValue

```
/***
* 函数功能: 获取常规组转换结果。
*
* uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc)
*
* 参数:
*   hadc - 指向 ADC_HandleTypeDef 结构的指针。
* 返回值: ADC 常规组转换数据。
*/
```

#### HAL\_ADC\_ConvCpltCallback

```
/***
* 函数功能: 非阻塞模式下的转换完成回调函数。
*
* __weak void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
*
* 参数:
*   hadc - 指向 ADC_HandleTypeDef 结构的指针。
* 返回值: 无。
*/
```

#### HAL\_ADC\_ConvHalfCpltCallback

```
/***
* 函数功能: 非阻塞模式下的 DMA 半传输完成回调函数。
*
* __weak void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc)
*
* 参数:
*   hadc - 指向 ADC_HandleTypeDef 结构的指针。
* 返回值: 无。
*/
```

#### HAL\_ADC\_LevelOutOfWindowCallback

```
/***
* 函数功能: 非阻塞模式下的模拟看门狗回调函数。
* 
```

```
* __weak void HAL_ADC_LevelOutOfWindowCallback(ADC_HandleTypeDef* hadc)
*
* 参数:
*   hadc - 指向 ADC_HandleTypeDef 结构的指针。
* 返回值: 无。
*/
```

### HAL\_ADC\_ErrorCallback

```
/**
* 函数功能: 非阻塞模式下的 ADC 错误回调函数。
*
* __weak void HAL_ADC_ErrorCallback(ADC_HandleTypeDef *hadc)
*
* 参数:
*   hadc - 指向 ADC_HandleTypeDef 结构的指针。
* 返回值: 无。
*/
```

### HAL\_ADC\_ConfigChannel

```
/**
* 函数功能: 配置选定的通道与常规组的连接。
*
* HAL_StatusTypeDef HAL_ADC_ConfigChannel(ADC_HandleTypeDef* hadc,
* ADC_ChannelConfTypeDef* sConfig)
*
* 参数:
*   hadc - 指向 ADC_HandleTypeDef 结构的指针。
*   sConfig - 指向 ADC_ChannelConfTypeDef 结构的指针。
* 返回值: HAL 状态。
*/
```

### HAL\_ADC\_AnalogWDGConfig

```
/**
* 函数功能: 配置模拟看门狗。
*
* HAL_StatusTypeDef HAL_ADC_AnalogWDGConfig(ADC_HandleTypeDef* hadc,
* ADC_AnalogWDGConfTypeDef* AnalogWDGConfig)
*
* 参数:
*   hadc - 指向 ADC_HandleTypeDef 结构的指针。
*   AnalogWDGConfig - 指向 ADC_AnalogWDGConfTypeDef 结构的指针。
* 返回值: HAL 状态。
*/
```

### HAL\_ADC\_GetState

```
/**  
 * 函数功能：返回 ADC 状态。  
 *  
 * uint32_t HAL_ADC_GetState(ADC_HandleTypeDef* hadc)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 * 返回值：HAL 状态。  
 */
```

### HAL\_ADC\_GetError

```
/**  
 * 函数功能：返回 ADC 错误代码。  
 *  
 * uint32_t HAL_ADC_GetError(ADC_HandleTypeDef *hadc)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 * 返回值：ADC 错误代码。  
 */
```

### ADC\_Enable

```
/**  
 * 函数功能：启用所选的 ADC。  
 *  
 * HAL_StatusTypeDef ADC_Enable(ADC_HandleTypeDef* hadc)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 * 返回值：HAL 状态。  
 */
```

### ADC\_ConversionStop\_Disable

```
/**  
 * 函数功能：停止 ADC 转换并禁用所选的 ADC。  
 *  
 * HAL_StatusTypeDef ADC_ConversionStop_Disable(ADC_HandleTypeDef* hadc)  
 *  
 * 参数：  
 *      hadc - 指向 ADC_HandleTypeDef 结构的指针。  
 * 返回值：HAL 状态。  
 */
```

### ADC\_DMAConvCplt

```

/**
 * 函数功能: DMA 传输完成回调。
 *
 * void ADC_DMAConvCplt(DMA_HandleTypeDef *hdma)
 *
 * 参数:
 *   hdma - 指向 DMA_HandleTypeDef 结构的指针。
 * 返回值: 无。
 */

```

### ADC\_DMAHalfConvCplt

```

/**
 * 函数功能: DMA 半传输完成回调。
 *
 * void ADC_DMAHalfConvCplt(DMA_HandleTypeDef *hdma)
 *
 * 参数:
 *   hdma - 指向 DMA_HandleTypeDef 结构的指针。
 * 返回值: 无。
 */

```

### ADC\_DMAError

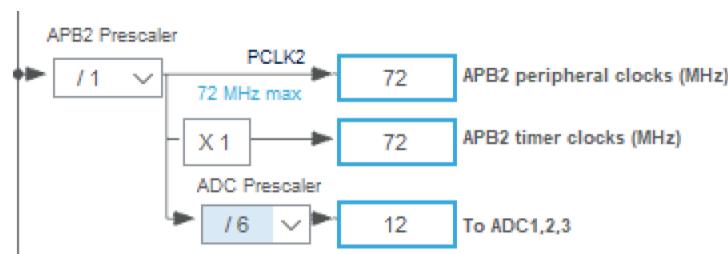
```

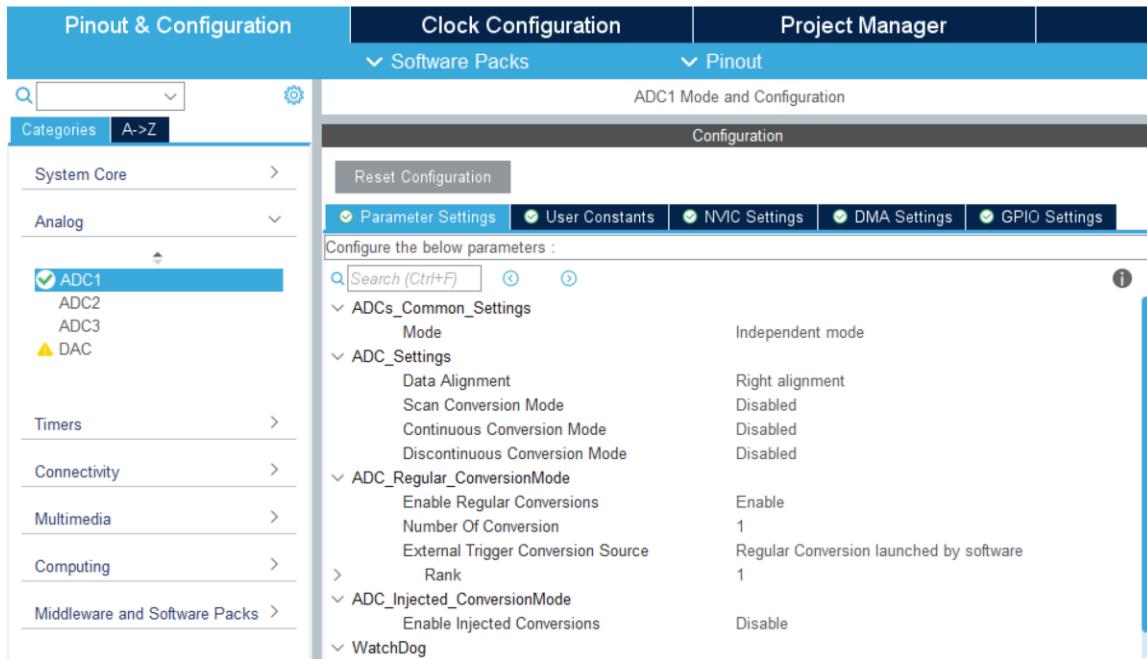
/**
 * 函数功能: DMA 错误回调。
 *
 * void ADC_DMAError(DMA_HandleTypeDef *hdma)
 *
 * 参数:
 *   hdma - 指向 DMA_HandleTypeDef 结构的指针。
 * 返回值: 无。
 */

```

## 1.21.4 ADC 测量电压

在 CubeMX 中配置串口 1, **时钟**等参数后, 选择 ADC1,通道 5 (任意通道) 按照如下配置 (保持默认)。





生成工程后，相关函数解析如下：

```

/* ADC1 初始化函数 */
void MX_ADC1_Init(void)
{
    /* 用户代码段 ADC1_Init_0 开始 */

    /* 用户代码段 ADC1_Init_0 结束 */

    ADC_ChannelConfTypeDef sConfig = {0}; // ADC 通道配置结构体初始化

    /* 用户代码段 ADC1_Init_1 开始 */

    /* 用户代码段 ADC1_Init_1 结束 */

    /** 通用配置 */
    hadc1.Instance = ADC1; // 设置 ADC 实例为 ADC1
    hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE; // 禁用扫描模式
    hadc1.Init.ContinuousConvMode = DISABLE; // 禁用连续转换模式
    hadc1.Init.DiscontinuousConvMode = DISABLE; // 禁用非连续转换模式
    hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START; // 设置外部触发转换为软件开始
    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT; // 设置数据右对齐
    hadc1.Init.NbrOfConversion = 1; // 设置转换数量为 1
    if (HAL_ADC_Init(&hadc1) != HAL_OK) // 初始化 ADC
    {

```

```
        Error_Handler(); // 如果初始化不成功，调用错误处理函数
    }

    /** 配置常规通道 */
    sConfig.Channel = ADC_CHANNEL_5; // 设置通道为 ADC_CHANNEL_5
    sConfig.Rank = ADC_REGULAR_RANK_1; // 设置通道排名为 1
    sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5; // 设置采样时间
    if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) // 配置 ADC 通道
    {
        Error_Handler(); // 如果配置不成功，调用错误处理函数
    }

    /* 用户代码段 ADC1_Init_2 开始 */

    /* 用户代码段 ADC1_Init_2 结束 */
}

/* ADC MSP 初始化 */
void HAL_ADC_MspInit(ADC_HandleTypeDef* adcHandle)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0}; // GPIO 初始化结构体
    if(adcHandle->Instance==ADC1) // 如果是 ADC1 实例
    {
        /* 用户代码段 ADC1_MspInit_0 开始 */

        /* 用户代码段 ADC1_MspInit_0 结束 */

        /* 启用 ADC1 时钟 */
        __HAL_RCC_ADC1_CLK_ENABLE();

        __HAL_RCC_GPIOA_CLK_ENABLE(); // 启用 GPIOA 时钟
        /**ADC1 GPIO 配置
        PA5 -----> ADC1_IN5
        */
        GPIO_InitStruct.Pin = GPIO_PIN_5; // 设置 GPIO 引脚为 PA5
        GPIO_InitStruct.Mode = GPIO_MODE_ANALOG; // 设置 GPIO 模式为模拟
        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct); // 初始化 GPIO

        /* 用户代码段 ADC1_MspInit_1 开始 */

        /* 用户代码段 ADC1_MspInit_1 结束 */
    }
}

/* ADC MSP 去初始化 */
```

```

void HAL_ADC_MspDeInit(ADC_HandleTypeDef* adcHandle)
{
    if(adcHandle->Instance==ADC1) // 如果是 ADC1 实例
    {
        /* 用户代码段 ADC1_MspDeInit 0 开始 */

        /* 用户代码段 ADC1_MspDeInit 0 结束 */

        /* 禁用 ADC1 时钟 */
        __HAL_RCC_ADC1_CLK_DISABLE();

        /**ADC1 GPIO 配置
        PA5 -----> ADC1_IN5
        */
        HAL_GPIO_DeInit(GPIOA, GPIO_PIN_5); // 去初始化 GPIO

        /* 用户代码段 ADC1_MspDeInit 1 开始 */

        /* 用户代码段 ADC1_MspDeInit 1 结束 */
    }
}

```

在 main.c 中

```

while (1)
{
    /* 开始 ADC 转换 */
    HAL_ADC_Start(&hadc1);

    /* 轮询 ADC 转换结果，等待最多 100ms */
    if (HAL_ADC_PollForConversion(&hadc1, 100) == HAL_OK)
    {
        /* 如果转换成功，读取 ADC 值 */
        ADC_data = HAL_ADC_GetValue(&hadc1);
    }

    /* 停止 ADC 转换 */
    HAL_ADC_Stop(&hadc1);

    /* 计算转换后的电压值，假设参考电压为 3.3V，12 位 ADC */
    val = (float)ADC_data * 3.3 / 4096;

    /* 将电压值格式化为字符串 */
    sprintf(msg, "val = %f V", val);
}

```

```

/* 通过 UART 发送格式化后的电压值字符串 */
HAL_UART_Transmit(&huart1, (uint8_t *)msg, strlen(msg), 1000);

/* 延时 1 秒 */
HAL_Delay(1000);

/* 用户代码段结束 */

/* 用户代码段开始 */
}

```

实验结果如下：

```

[2023-12-08 11:25:01.334]# RECV ASCII>
val = 0.000000 V

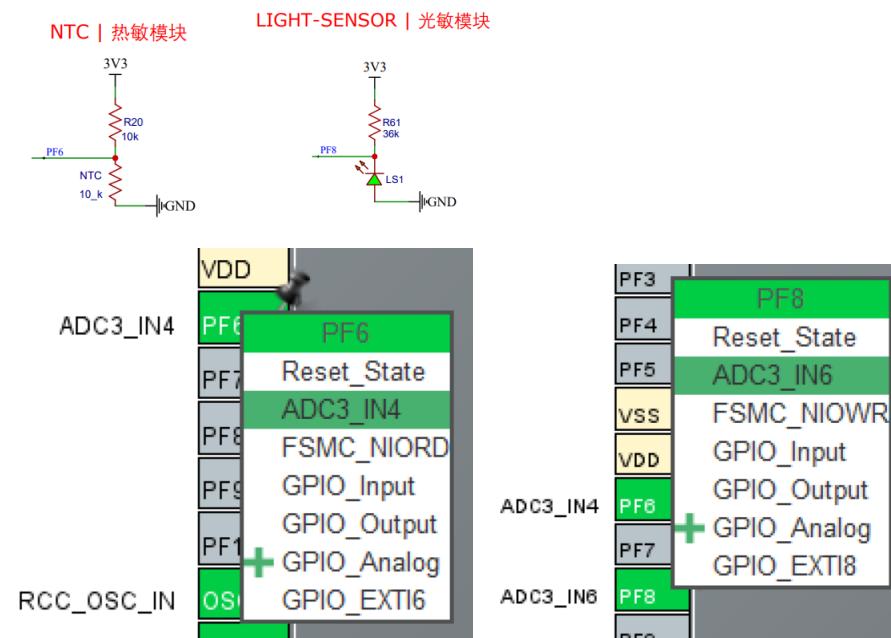
[2023-12-08 11:25:02.333]# RECV ASCII>
val = 2.749732 V

[2023-12-08 11:25:03.338]# RECV ASCII>
val = 2.839966 V

```

### 1.21.5 ADC 使用光敏/热敏测量光照/温度

查阅原理图，找到光敏和热敏对应引脚。



分别对应的 ADC 的通道 3 和通道 4。配置时钟，串口后生成工程目录

修改主函数

```

/* USER CODE BEGIN PV */
char msg[100]; // 用于存储格式化的字符串

```

```
ADC_ChannelConfTypeDef sConfig = {0}; // 定义 sConfig
/* USER CODE END PV */
```

在主循环中

```
while (1)
{
/* 测量光敏电阻 */
sConfig.Channel = ADC_CHANNEL_6; // 设置为通道 6
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5; // 设置适合该通道的采样时间
HAL_ADC_ConfigChannel(&hadc3, &sConfig); // 重新配置 ADC

// 开始 ADC 转换
HAL_ADC_Start(&hadc3);

// 等待转换完成
if (HAL_ADC_PollForConversion(&hadc3, 100) == HAL_OK)
{
    // 读取 ADC 值
    uint32_t light_sensor_data = HAL_ADC_GetValue(&hadc3);
    // 将 ADC 值转换为 0-100 范围
    int light_sensor_percentage = (int)((light_sensor_data * 100) / 4095);

    // 格式化并发送数据
    sprintf(msg, "Light Sensor = %d\r\n", light_sensor_percentage);
    HAL_UART_Transmit(&huart1, (uint8_t *)msg, strlen(msg), 1000);
}

// 停止 ADC 转换
HAL_ADC_Stop(&hadc3);

HAL_Delay(200); // 稍微延迟, 以便区分两个测量

/* 测量热敏电阻 */
sConfig.Channel = ADC_CHANNEL_4; // 设置为通道 4
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5; // 可以根据需要调整
HAL_ADC_ConfigChannel(&hadc3, &sConfig); // 重新配置 ADC

// 开始 ADC 转换
HAL_ADC_Start(&hadc3);

// 等待转换完成
if (HAL_ADC_PollForConversion(&hadc3, 100) == HAL_OK)
{
```

```

// 读取 ADC 值
uint32_t thermistor_data = HAL_ADC_GetValue(&hadc3);
// 将 ADC 值转换为 0-100 范围
int thermistor_percentage = (int)((thermistor_data * 100) / 4095);

// 格式化并发送数据
sprintf(msg, "Thermistor = %d\r\n", thermistor_percentage);
HAL_UART_Transmit(&huart1, (uint8_t *)msg, strlen(msg), 1000);
}

// 停止 ADC 转换
HAL_ADC_Stop(&hadc3);

// 延迟 1 秒
HAL_Delay(2000);
}

/* USER CODE END 3 */

```

实验结果：

[2023-12-08 11:46:15.514]# RECV ASCII>  
Thermistor = 47

[2023-12-08 11:46:17.518]# RECV ASCII>  
Light Sensor = 70

[2023-12-08 11:46:17.716]# RECV ASCII>  
Thermistor = 46

[2023-12-08 11:46:19.723]# RECV ASCII>  
Light Sensor = 65

[2023-12-08 11:46:19.924]# RECV ASCII>  
Thermistor = 47

## 1.21.6 STM32 大作业

使用 STM32 实现如下功能：

1. 单总线采集 DHT11 温湿度。
2. ADC 采集环境光照。
3. 将各环境信息显示在屏幕上，并实时以“light\_intensity:xxx”格式发送至串口 2。

- 4.当串口 2 收到" control: servo\_motor:on"时控制舵机， " control: led:on"控制灯。
- 5.当温度超过阈值，蜂鸣器报警，舵机转动，报警灯闪烁。