步进电机与网关

参数项	参数值
型号	57SM110 步进电机
电流	2.5A
输出力矩	1.2Nm (牛米)
机身长度	76mm
出轴长度	19mm
出轴轴径	8mm
出轴方式	单出轴
理想转速	1 - 200RPM，最高转速 600RPM (超过 200RPM 扭矩会逐渐下降)
出线方式	二相四根引出线（黑色 A+ 绿色 A- 红色 B+ 蓝色 B-）
编码器输出	红色 5V 电源，黑色 GND，黄色 A 相，绿色 B 相，具体参考电机铭牌
编码器参数	1000 线增量旋转光电编码器，AB 型

通过TB67S109A 芯片细分

TB67S109A功能

1. CLK 功能

CLK 信号的每个上升沿会切换电机的当前步距和电角度。

CLK 信号	功能描述
↑（上升沿）	每一个上升沿切换电角度和当前步距
↓（下降沿）	-（无变化，保持原有状态）

注释：芯片的 CLK 引脚内置 200 ns（±20%）的模拟滤波器。

2. ENABLE 功能

ENABLE 引脚控制步进电机输出的开启与关闭，切换至开启状态时电机启动，切换至关闭状态时电机停止。（关闭状态下，所有输出 MOSFET 关断，进入高阻抗状态（Hi-Z）。）

建议在 VM 上电和掉电序列期间将 ENABLE 引脚设为低电平，以避免 VM 电压低于工作范围时电机发生误动作；因此，推荐在 VM 电压稳定在实际工作电压后，再将 ENABLE 引脚切换为高电平。

ENABLE 引脚状态	功能描述
High（高电平）	电机输出：开启（正常工作）
Low（低电平）	电机输出：关闭（高阻抗状态）

3. CW/CCW 功能 / 输出引脚功能（充电初始阶段输出逻辑）

CW/CCW 引脚控制步进电机的旋转方向。当 CW/CCW 设为高电平时，充电初始阶段 OUT (+) 输出高电平、OUT (-) 输出低电平；当 CW/CCW 设为低电平时，充电初始阶段 OUT (+) 输出低电平、OUT (-) 输出高电平。

CW/CCW 引脚状态	功能描述
High（高电平，CW）	顺时针旋转：A 相（Ach）电流领先 B 相（Bch）电流 90 度相位差
Low（低电平，CCW）	逆时针旋转：B 相（Bch）电流领先 A 相（Ach）电流 90 度相位差

4. DMODE（步距分辨率设置）功能

DMODE 引脚用于设置步进电机运行的步距分辨率。若 3 个引脚（DMODE0、DMODE1、DMODE2）均设为低电平，器件将进入 “待机模式”；待机模式下，部分内部电路完全关断以降低功耗。若 3 个引脚中任意一个设为高电平，TB67S249FTG 将从待机模式重启，但内部电路稳定需要 7.5 μs（典型值），因此需等待该预热时间后再输入启动信号。

DMODE0	DMODE1	DMODE2	功能描述
High	High	High	1/32 步分辨率
High	High	Low	1/16 步分辨率
High	Low	High	1/8 步分辨率
High	Low	Low	1/2（b）步分辨率
Low	High	High	1/4 步分辨率
Low	High	Low	1/2（a）步分辨率
Low	Low	High	1/1 步分辨率（全步）
Low	Low	Low	待机模式（内部振荡器电路（OSCM）和输出 MOSFET 关断）

注释：

1. DMODE 引脚内置 1.25 μs（±20%）的数字滤波器；
2. 待机模式下，DMODE 引脚内置 0.94 μs（±20%）的数字滤波器；
3. 运行期间可切换 DMODE0、DMODE1、DMODE2 引脚状态，切换后下一个步距将采用切换前后最接近的电角度对应的电流。详细功能请参考应用笔记。

转动角度 一个上升沿走一步， 控制上升沿数量 以此控制角度，

控制上升沿产生周期，方波频率，控制电机转速

5. RESET 功能

RESET 引脚用于初始化内部电角度。

RESET 引脚状态	功能描述
High（高电平）	初始化内部电角度
Low（低电平）	正常工作

注释：芯片的 RESET 引脚内置 0.625 μs（±20%）的数字滤波器。

当 RESET 引脚设为高电平时，每个 H 桥（A 相、B 相）的电流设置将如下表所示；此外，当电角度与初始值一致时，MO 引脚输出低电平。

步距分辨率	A 相电流	B 相电流	电角度
1/32 步设置	71%	71%	45°
1/16 步设置	71%	71%	45°
1/8 步设置	71%	71%	45°
1/2（b）步设置	71%	71%	45°
1/4 步设置	71%	71%	45°
1/2（a）步设置	100%	100%	45°
1/1 步设置	100%	100%	45°

驱动电机

使用定时器产生方波给电机驱动，

OC1M[2:0]位组合	模式名称	功能描述
000	冻结	输出比较寄存器 TIMx_CCR1 与计数器 TIMx_CNT 的比较对 OC1REF 不起作用；OC1REF 保持原有电平
001	匹配时设有效电平	当 TIMx_CNT = TIMx_CCR1 时，强制 OC1REF 为高电平
010	匹配时设无效电平	当 TIMx_CNT = TIMx_CCR1 时，强制 OC1REF 为低电平
011	翻转	当 TIMx_CCR1 = TIMx_CNT 时，翻转 OC1REF 的电平
100	强制为有效电平	强制 OC1REF 为高电平
101	强制为无效电平	强制 OC1REF 为低电平
110	PWM 模式 1	向上计数：TIMx_CNT<TIMx_CCR1 时通道 1 为有效电平，否则为无效电平；向下计数：TIMx_CNT>TIMx_CCR1 时通道 1 为无效电平（OC1REF=0），否则为有效电平（OC1REF=1）
111	PWM 模式 2	向上计数：TIMx_CNT<TIMx_CCR1 时通道 1 为无效电平，否则为有效电平；向下计数：TIMx_CNT>TIMx_CCR1 时通道 1 为有效电平，否则为无效电平

使用定时tim1　011 翻转模式：

在 定时器计数器值(CNT）==比较寄存器(CRR)的值 翻转电平，产生高电平，

一个方波溢出两次

时钟：72mhz　进行72分频　即记一个数 1us

控制电机速度：

控制方波的频率(周期)

1　设置自动重装载寄存器(ARR)的值来控制速度

当定时器溢出翻转产生高电平后，计数达到自动重装载寄存器重装载，再次等在计数溢出，通过控制自动重装载寄存器的值，来控制方波产生的周期

2　中断控制：

当定时器计数器值(CNT) ==比较寄存器(CRR)的值 翻转电平时，会产生中断，在中断中，不断改变CCR的值 ，再次进入时，继续修改 以此控制周期

在一个方波发完 需要产生中断，判断是否发完了方波

步数换算角度

```
if(htim->Instance ==TIM1){
        time_count ++;
        //一个方波溢出两次
        if(time_count==2){
            //两次溢出 走路一步
            step_count++;
              currentAngle = step_to_angle(step_count);
        if(currentAngle>= abs(targetAngle)){
            Inf_Motor_Stop();
            step_count=0;
        }
            time_count=0;
//步数换算角度
// 一步1.8° 8分频  1600步走一圈360°
// 360°/1600 =  当前角度/当前步数
 #define step_to_angle(x) (360 * x / 1600.0)
```

速度与周期的换算

```
//*************************速度与周期的换算********************** */
//速度[步/s]转换周期
//一个脉冲一步  一个脉冲 溢出2次
// n步/s  n个脉冲/s  =2n次溢出/s  =  一次溢出时间周期 = 1/2n s，一次溢出时间周期(重装载值) = 1000000/2n us ，
// 定时器72分频 计一个数1us  计多少数就是多少us
// 一个脉冲需要的计数
#define speed_to_period(x)  (1000000 / (2 * x))
```

const

`const` 是 C 语言的**常量修饰符**，核心作用是：**定义 “只读、不可修改” 的常量**

`const` 修饰的变量一眼就能看出是 “配置项”（比如电机启动相关的固定参数），而非运行时动态变化的变量（如 `current_period`、`step_count`），其他人看代码时能快速区分 “配置” 和 “状态”

`onst` 全局变量默认存储在 **ROM（程序存储区）**，而非 RAM（数据存储区），尤其对于嵌入式系统（如 STM32），RAM 资源有限，用 `const` 可节省宝贵的 RAM 空间（普通全局变量存在 RAM 中）。

步进电机加减速

加速阶段：从低于电机启动极限的基础频率开始，逐步提升脉冲频率。

恒速阶段：保持稳定频率运行；

减速阶段：逐步降低脉冲频率，避免惯性过冲。
加减速算法需匹配电机矩频特性（输出扭矩随频率下降），常见曲线包括直线（梯形）、指数、S型，其中S型曲线通过平滑加速度减少机械冲击，适用于高精度场景。

控制类型	原理	优点	缺点	适用场景
梯形加减速	脉冲频率线性增减，加速度恒定	控制简单，计算量小	高速时易因扭矩不足失步，位置机械冲击明显	轻负载、快速定位（3D 打印机）
S 型曲线加减速	加速度平滑变化，减少启动 / 停止时的加速度突变	机械冲击小，定位精度高，抑制共振	算法复杂，需高算力支持	高精度、重负载（CNC 机床）
闭环控制	通过编码器反馈实时调整脉冲频率，结合 PID 算法补偿误差	抗干扰强，精度高，适应负载变化	成本高，系统复杂度增加	高动态响应场景（自动化产线）

步进电机编码器闭环控制

编码器是**将机械运动（位移、转速、方向）转换为电信号的传感器**，核心是解决 “运动状态可测、可验证

编码器的通用核心作用

作用	具体说明
位置检测	精准测量电机轴的旋转角度 / 圈数，输出 “实际位置” 信号（如转 1 圈输出 1000 个脉冲）；
速度检测	通过单位时间内的脉冲数计算电机实际转速（脉冲频率 = 转速 × 每转脉冲数）；
方向检测	通过两路脉冲（A/B 相）的相位差（超前 / 滞后）判断电机正转 / 反转；
零位校准（原点）	绝对式编码器或增量式的 Z 相脉冲，可标定电机 “机械零点”，避免累计误差；

1000 线增量旋转光电编码器，AB 型　电机转一圈编码器给mcu发1000个脉冲

编码器与步进电机轴刚性连接（同轴旋转），电机转 1 步 / 1 圈，编码器输出对应脉冲：

定时器从模式：sms

SMS[2:0]：从模式选择 (Slave mode selection)

当选择了外部信号，触发信号(TRGI)的有效边沿与选中的外部输入极性相关(见输入控制寄存器和控制寄存器的说明)

000：关闭从模式 – 如果CEN=1，则预分频器直接由内部时钟驱动。

001：编码器模式1 – 根据TI1FP1的电平，计数器在TI2FP2的边沿向上/下计数。

010：编码器模式2 – 根据TI2FP2的电平，计数器在TI1FP1的边沿向上/下计数。

011：编码器模式3 – 根据另一个信号的输入电平，计数器在TI1FP1和TI2FP2的边沿向上/下计数。

100：复位模式 – 选中的触发输入(TRGI)的上升沿重新初始化计数器，并且产生一个更新寄存器的信号。

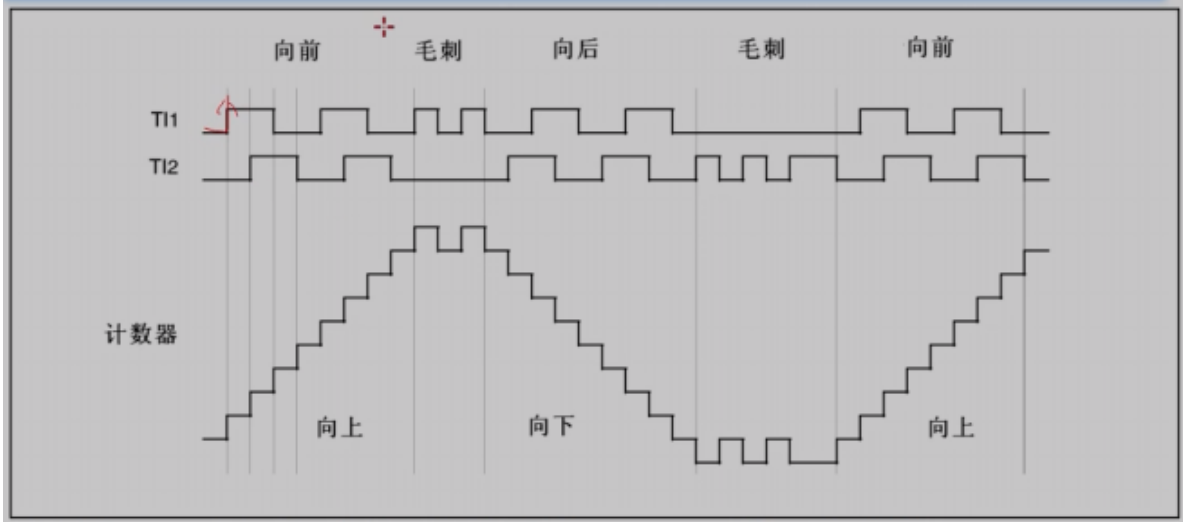
101：门控模式 – 当触发输入(TRGI)为高时，计数器的时钟开启。一旦触发输入变为低，则计数器停止(但不复位)。计数器的启动和停止都是受控的。

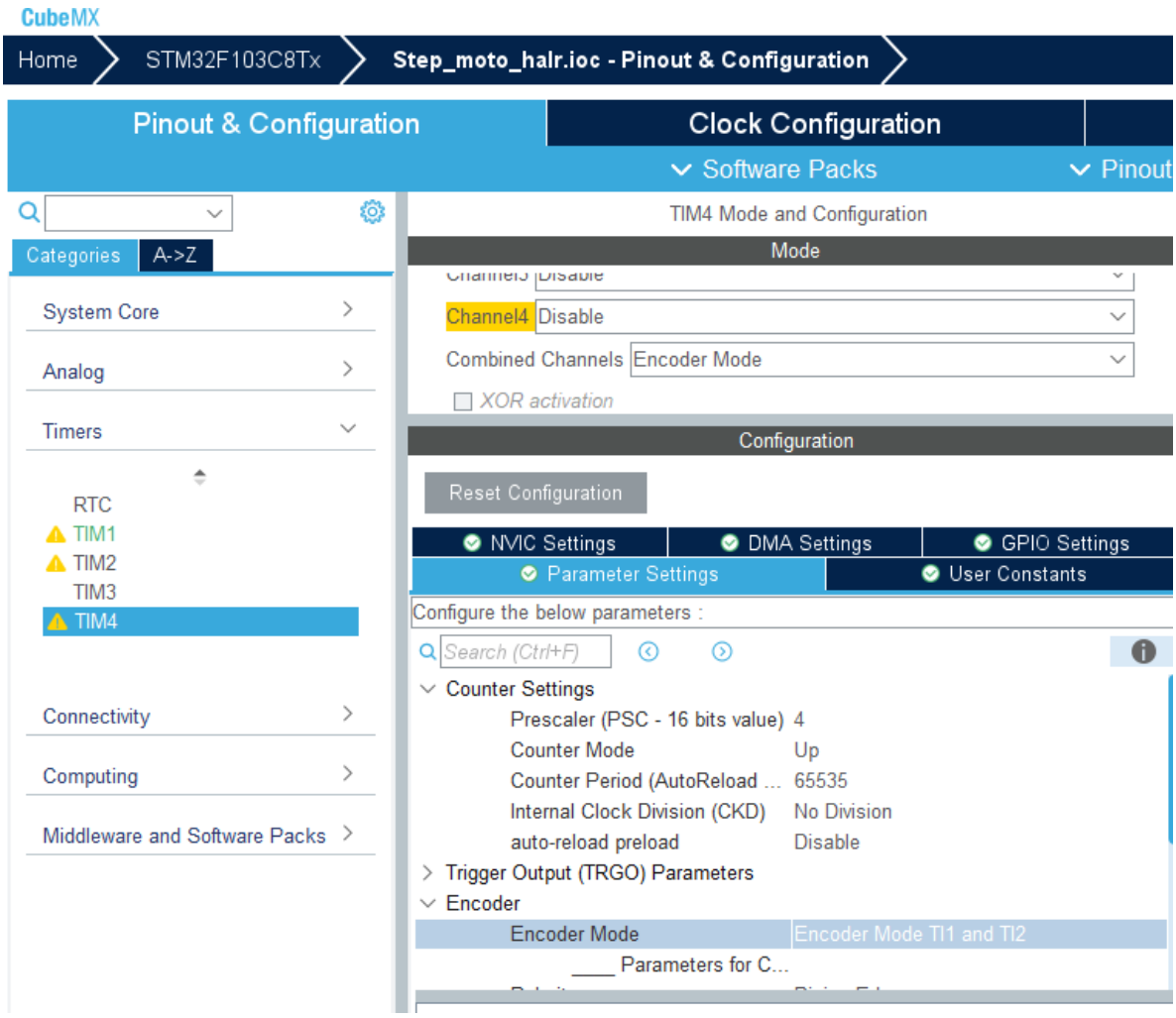
110：触发模式 – 计数器在触发输入TRGI的上升沿启动(但不复位)，只有计数器的启动是受控的。

111：外部时钟模式1 – 选中的触发输入(TRGI)的上升沿驱动计数器。

注：如果TI1F_EN被选为触发输入(TS=100)时，不要使用门控模式。这是因为，TI1F_ED在每次TI1F变化时输出一个脉冲，然而门控模式是要检查触发输入的电平。

有效边沿	相对信号的电平 (TI1FP1对应TI2, TI2FP2对应TI1)	TI1FP1信号		TI2FP2信号	
		上升	下降	上升	下降
仅在TI1计数	高	向下计数	向上计数	不计数	不计数
	低	向上计数	向下计数	不计数	不计数
仅在TI2计数	高	不计数	不计数	向上计数	向下计数
	低	不计数	不计数	向下计数	向上计数
在TI1和TI2上计数	高	向下计数	向上计数	向上计数	向下计数
	低	向上计数	向下计数	向下计数	向上计数





定时器在初始化时会产生一次中断 更新事件 中断标记未清除 、

我们的电机编码器是1000线,代表电机转一圈会发1000个脉冲

- 我们使用的事编码器模式3计数,一个脉冲会计数4次,意味着电机转一圈会发1000个脉冲会计数4000次

PID

```
// 积分项：累加误差×时间间隔（Δt=0.01s为例）

integral += error * 0.01;

 // 微分项：(当前误差-上一次误差)÷时间间隔

derivative = (error - last_error) / 0.01;
```

系统类型	Kp（比例系数）	Ki（积分系数）	Kd（微分系数）
轻负载（<0.5N · m）	0.5~2.0	0.01~0.1	0.1~1.0
中负载（0.5~2N · m）	2.0~5.0	0.1~0.5	1.0~5.0
重负载（>2N · m）	5.0~10.0	0.5~2.0	5.0~10.0

临界比例度法（Ziegler-Nichols 法，适合有经验者）

核心逻辑：**先找到系统 “临界震荡状态”（持续等幅震荡），再根据震荡周期计算参数**，适合对响应速度有要求的场景。

对物体进行位置控制时，目标值=目标位置，反馈值=当前位置，输出值=施加的驱动力大小，PID就能实时计算出驱动力使物体到达目标位置

比例消除偏差，积分消除稳态误差，微分超前改善动态性能，需谨慎调整以避免误动作。 感知也最明显

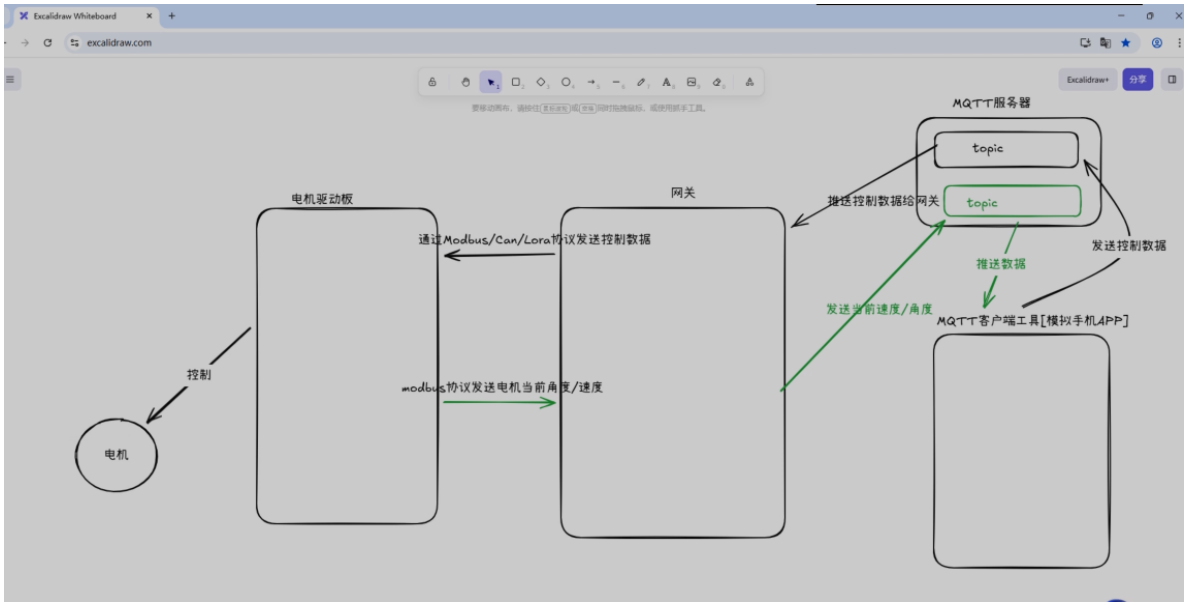
容易引起系统不稳定，比例控制容易不稳定。 调节速度快，控制及时，但有差调节，过大易不稳定。

积分的作用是减小或消除偏差，消除稳态误差

微分产生超前作用，改善动态性能，但抗干扰能力差(容易也对干扰信号也求导) 过大加剧震荡 过小抑制不明显

极性一般相同

网关



下层逻辑(Inf)调用 上层(App) 使用 回调函数

核心是通过 **“上层注册回调→底层保存回调→底层触发回调”** 的流程，实现分层解耦（inf 层只负责网络收发，不关心数据如何处理；app 层负责数据处理，不依赖 inf 层的具体实现）

mqtt powershell mosquitto -v -c 'D:\asoftware\mosquitto\mosquitto.conf'

{

"humidifier":"close"

}

json 格式

```
{
  "id": 5,
  "connectType":"modbus",
  "is_start": 1,
   "targetAngle": 3600,
   "targetSpeed": 1600
}
```

Modbus

核心架构：主从式通信

Modbus 采用**严格的主从 (Master-Slave) 架构**，通信过程由主设备 (如工控机、PLC) 发起，从设备 (如传感器、执行器) 被动响应：

1 主从架构

Modbus 只有 “主站” 和 “从站” 两种角色，绝对不会混乱：

- 主站：主动发指令的设备（比如你的网关 MCU）；
- 从站：被动响应的设备（比如电机驱动板、传感器）；
- 规则：只有主站能发指令，从站只能响应（且只响应给自己的指令），避免设备抢着说话导致通信混乱。

四种数据类型

2 两种主流通信模式（适配不同硬件）

模式	传输载体	特点	你的场景适配
Modbus RTU	串口（RS485/RS232）	二进制编码，数据紧凑、速度快、抗干扰（工业首选）	网关←→电机驱动板

模式	传输载体	特点	你的场景适配
RTU	（RS485/RS232）	快、抗干扰（工业首选）	
Modbus TCP	以太网 / 网络	基于 TCP/IP，把 RTU 数据包封装成 TCP 包，适配网络设备	若驱动板带网口，可直接通过 W5500 通信

1）Modbus RTU（Remote Terminal Unit）

传输方式：二进制编码，通过串行通信（RS-485/RS-232）传输。

帧格式：

设备地址（1字节） + 功能码（1字节） + 数据（N字节） + CRC校验（2字节）。

特点：高效紧凑，适合工业现场环境，是应用最广泛的版本。

2）Modbus ASCII

传输方式：ASCII 字符编码，每个字节以十六进制字符表示。

帧格式：

起始符 : + 设备地址（2字符） + 功能码（2字符） + 数据 + LRC校验（2字符） + 结束符 \r\n。

特点：可读性强，但效率低于 RTU，适用于调试场景。

3）Modbus TCP

传输方式：基于 TCP/IP 协议，通过以太网传输。

帧格式：

MBAP头（7字节，含事务标识、协议标识、长度、单元标识） + 功能码（1字节） + 数据。

特点：支持高速网络通信，常用于现代工业以太网系统，默认端口 502。

在十六进制中，每一位（0-F）可以用 4 个二进制位（bit）来表示。 • 0x00 有两位十六进制数字 (0 和 0)，所以它代表 2 * 4 = 8 个比特 (bit)。 • 8 个比特正好等于 1 个字节 (Byte)。

3 极简的核心数据模型（只定义 4 类数据）

Modbus 不关心设备内部具体逻辑，只定义 “能读写的 4 类数据”

数据类型	功能描述	主机读写属性	数据格式	你的电机场景举例
线圈（Coil）	表示开关类布尔状态	可读写	uint8_t 数组（元素为 0/1）	控制电机 “启停”（1 = 启动，0 = 停止）
离散输入	表示外部输入的布尔状态	只读	uint8_t 数组（元素为 0/1）	读取电机 “限位开关状态”（1 = 触发，0 = 未触发）
保持寄存器	存储设备参数 / 运行状态	可读写	uint16_t 数组	写入电机 “目标角度”（如 100°）、读取 “实际转速”
输入寄存器	存储传感器采集的数值	只读	uint16_t 数组	读取电机 “工作电流 / 绕组温度”（驱动板采集后存储）

1 输入寄存器： “从机向主机上报自身采集的被动数据”。

- 从机（比如你的设备）主动采集自身的状态 / 数据（比如电流、温度、传感器值），并把这些数据存在`REG_INPUT_BUF`（输入寄存器缓冲区）中；
- 主机（比如网关）只能通过**功能码 04（读输入寄存器）**读取这些数据，**不能写入**—— 因为这些数据是从机 “被动采集的、主机无需干预的状态”

1. **保持寄存器（Holding Register）**：
    - 协议属性：主机可读写、从机可读写；
    - 用途：存储主机下发的控制参数（比如电机目标角度、PID 参数）；
    - 对应从机回调：`eMBRegHoldingCB`（同时支持 “读保持寄存器（功能码 03）” 和 “写保持寄存器（功能码 06/16）”）。
2. **线圈（Coil）**：
    - 协议属性：主机可读写、从机可读写；
    - 用途：控制从机的开关状态（比如电机启停、继电器通断）；
    - 对应从机回调：`eMBRegCoilsCB`

Modbus 在场景里的典型数据流转（结合 MQTT）

云端（MQTT客户端）→ MQTT服务器 → W5500（网关）→ 网关MCU（Modbus主站）→ 电机驱动板（Modbus从站）→ 电机

1. 云端通过 MQTT 给网关发 “电机转到 100°” 的指令；
2. 网关的 W5500 收到 MQTT 数据，传给 MCU；
3. MCU（Modbus 主站）把 “100°” 封装成 Modbus RTU 指令，通过串口发给电机驱动板（从站）；
4. 驱动板（从站）解析指令，把 100° 写入 “目标角度保持寄存器”，驱动电机转动；
5. 驱动板把电机实际角度存在 “保持寄存器”，MCU 主动读取该寄存器，再通过 W5500/MQTT 上报给云端。

Modbus 时序

modbus 底层依赖串口通信

1. Modbus RTU 数据帧格式（发送方）

数据发送方发送的 Modbus 数据帧格式为：

设备 ID [1 字节] + 功能码 [1 字节] + 数据 [N 字节] + CRC 校验码 [2 字节]

2. Modbus RTU 帧结束的判断方式

Modbus RTU 底层通过串口传输数据，接收方判断 “一帧数据发送完成” 的规则：

通过字节之间的时间间隔确定 —— 若两个字节之间的时间间隔 > 3.5 个字符时间，则认为一帧数据发送完成。

3. 串口通信的单个字节组成

串口通信中，一个字节的传输结构为：

起始位 [1bit] + 数据位 [8bit] + 校验位 [1bit] + 停止位 [1bit] = 共 11bit

4. 波特率的定义

串口通信需约定波特率，以保证双方数据正常交互：

波特率的含义是 “1 秒内发送的 bit 数量”。

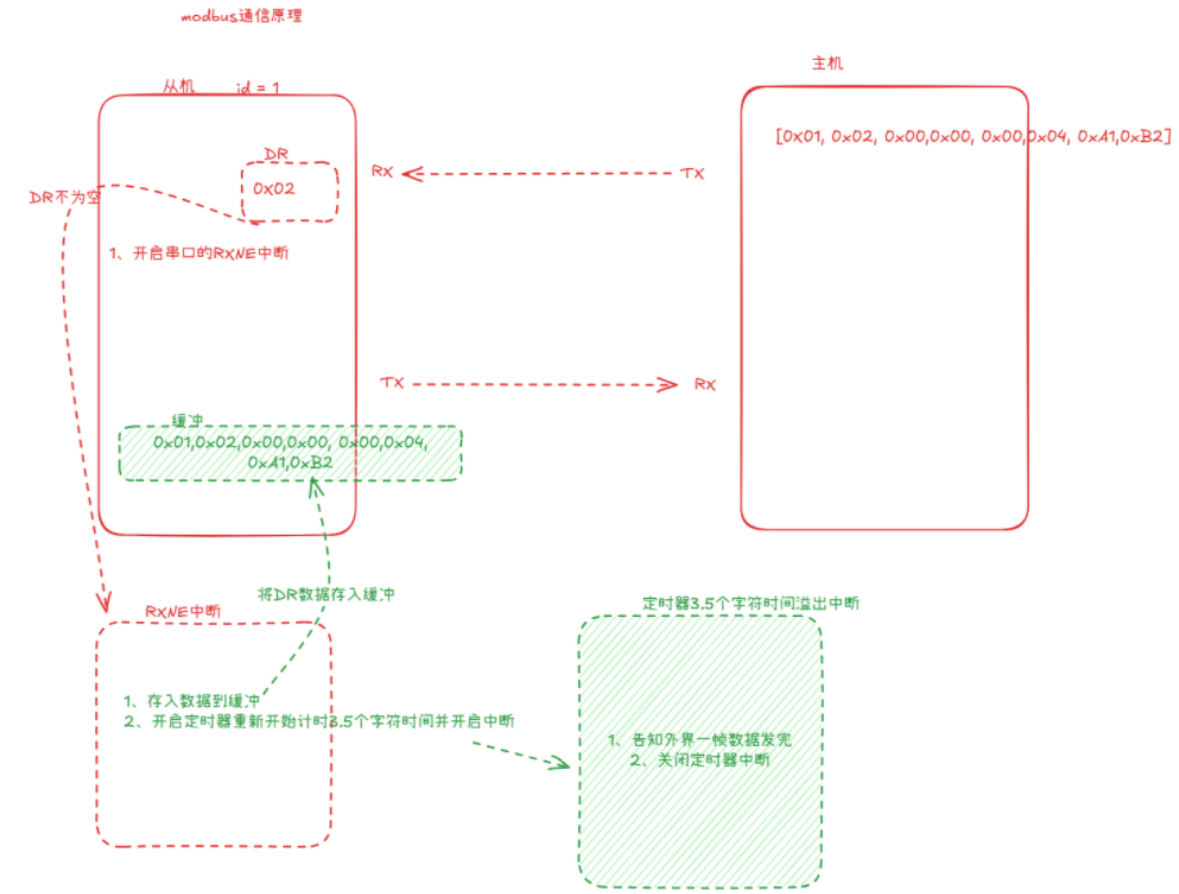
5. 波特率与传输时间计算（以 9600 为例）

假设波特率 = 9600：

- 1bit 的传输时间 = 1/9600 s
- 1 个字节的传输时间 = 11 × 1/9600 s
- 3.5 个字符时间 = 3.5 × 11 × 1/9600 s

6. 不同波特率下 “3.5 个字符时间” 的计算规则

- 低速波特率（≤19200）： 3.5 个字符时间严格按照上述公式计算；
- 高速波特率（>19200）： 3.5 个字符时间 = 35 个 50μs。



---

modbus 配入电机

modbus 移植 从机

CubeMax 开启串口 自己写中断函数

portserial.c 开启串口接收发送中断 串口初始化 (main函数中完成) 放入接收发送字节

串口中断函数 调用接收发送函数

porttimer.c 启动定时器 定时器溢出中断函数

```
# FreeModbus 移植流程

## 1. 准备阶段
├── 下载源码
|   ├── GitHub：https://github.com/cwalter-at/freemodbus
|   └── 或使用资料包
└── 搭建目录
    ├── 新建 Freemodbus/
    ├── 复制 modbus/ 源码
    └── 复制 demo/BARE/port/

## 2. 硬件接口移植
├── 串口移植（USART）
|   ├── 配置波特率与中断
|   ├── 修改 portserial.c
|   └── 实现中断回调
└── 定时器移植（TIM）
    ├── 配置为 50us 定时
    ├── 用于 3.5 字符时间检测
    └── 修改 porttimer.c

## 3. 协议栈回调实现
├── 定义数据缓冲区（port.h）
|   ├── 输入/保持寄存器
|   ├── 线圈/离散输入
|   └── 大小宏定义
└── 实现回调函数（port.c）
    ├── eMBRegInputCB（04）
    ├── eMBRegHoldingCB（03/06/16）
    ├── eMBRegCoilsCB（01/05/15）
    └── eMBRegDiscreteCB（02）

## 4. 中断关联
├── 串口接收 → pxMBFrameCBByteReceived
├── 串口发送 → pxMBFrameCBTransmitterEmpty
└── 定时器溢出 → pxMBPortCBTimerExpired

## 5. 编译与调试
├── 添加 -DNDEBUG 去除断言
├── 调整缓冲区与寄存器数量
└── 使用 Modbus 工具测试
```

大小端系统	多字节数据的 “存储规则”：规定「高 8 位 / 低 8 位」放在内存的 “低地址” 还是 “高地址”，是 CPU / 系统的硬件特性

1. 小端序（Little-Endian，嵌入式主流：STM32/PC/51/ARM 绝大多数 MCU）

核心规则：低字节放内存低地址，高字节放内存高地址（“低位低址”）。

2. 大端序（Big-Endian，少见：部分串口设备 / 网络协议）

核心规则：高字节放内存低地址，低字节放内存高地址（“高位低址”），和人类读写数字的习惯一致。

联合体union

union

联合体是 C 语言中**所有成员共享同一块内存空间**的自定义数据类型，核心特性是「内存复用」—— 总大小等于最大成员的字节数（含内存对齐），同一时间只有一个成员的值有效（修改一个成员会覆盖其他成员）。对 Modbus 嵌入式开发而言，union 是拆分 / 合并 16 位数据（如寄存器地址、CRC 值、写入数据）的“神器”，

特性	联合体（union）	结构体
内存分配	所有成员共享内存，总大小 = 最大成员大小（对齐）	成员独立分配，总大小 = 各成员之和（对齐）
成员有效性	仅当前“激活”的成员有效（修改一个覆盖全部）	时访问所有成员（值互不影响）
核心用途	数据类型转换、16/32 位数据拆分为 8 位字节、节省内存	封装不同类型的独立数据（如 Modbus 帧结构体）

联合体就是「一块内存，多种用法」—— 你可以把它当成 16 位的整体用，也可以拆成两个 8 位的部分用，同一时间只能用一种用法，改了其中一种

```
typedef union
{
        float value;
        uint16_t arr[2];
}float_2_uint16;
```

定义的这个 `float_2_uint16` 联合体**完全可以实现“浮点值 ↔ 两个 16 位整型”的互相拆分 / 合并**—— 核心是联合体共享 4 字节内存（`float` 占 4 字节，`uint16_t arr[2]` 是 2 个 16 位整数，也占 4 字节），本质是“同一坨二进制数据，既可以解读为 float，也可以解读为两个 uint16_t”

Modbus 协议本身不支持直接传输 float，但可以把 float 拆成两个 16 位整数，通过「10H 写多个保持寄存器」传输，接收端再拼回 float

联合体的核心：给同一块二进制内存“多套解读规则”

关键：没有“转换过程”，只有“解读方式”

你不用手动写代码把 `float` 转二进制，也不用把二进制转 `uint16_t` —— 编译器会帮你：

- 给 `data.value = 25.5` 时，编译器自动把 25.5 转成二进制 `0x41C80000` 存进内存；
- 读data.arr[1]时，编译器直接从这 4 字节里，按uint16_t的规则取高 2 字节，解读成 0x41C8；全程没有额外的“转换函数 / 计算”，只是“直接读内存，按不同规则翻译”。

✅ 联合体的本质：**数据本身以二进制存在于内存，联合体允许你用不同的类型（float/uint16_t 等）去解读同一块二进制内存，拿的时候直接按想要的类型读，无需手动转二进制 / 转类型。**

对你的 Modbus 开发来说，这个特性的价值就是：

不用手动把 float 转成二进制再拆分，也不用手动把两个 uint16_t 的二进制拼起来转 float—— 联合体帮你 “一键切换解读方式”，代码简洁还不容易错。

modbus 功能码

功能码最高位为1 异常数据

写单个线圈

Modbus 0x05 写单线圈的 RTU 帧共 8 字节，拆分如下：

字节位置	内容	是否参与 CRC 计算
0	从站地址	✅ 是（第 1 个参与字节）
1	功能码 0x05	✅ 是
2	线圈地址高字节	✅ 是
3	线圈地址低字节	✅ 是
4	输出值高字节	✅ 是
5	输出值低字节	✅ 是（第 6 个参与字节）
6	CRC 低字节	❌ 否（计算结果，不参与）
7	CRC 高字节	❌ 否（计算结果，不参与

具体可参照手册

Modbus RTU通讯协议详解与实例演示 - 成都亿佰特 - 博客园.html

```
/**
 * @brief 写单个线圈
 *
 * @param id 从设备id
 * @param index 线圈数组的角标 0x0000~0xffff
 * @param data  待写入的数据 只有0和1  0x0000 0xff00
 */
void Inf_Modbus_WriteColil(uint8_t id,uint16_t index,uint8_t data){
    uint8_t cmd[8]={0};
    cmd[0]=id;    //设备id
    cmd[1]=0x05;  // 功能码
    cmd[2]=index>>8;  //地址角标  高字节在前
    cmd[3]=index;    //低8位
    // 线圈写1 0xFF00 写0  0x0000  仅这两个值有效,依旧高位在前
    if(data==1){
        cmd[4]=0xFF;
    }else if (data==0)
    {
        cmd[4]=0x00;
```

```
    }
    cmd[5]=0x00;
    //计算校验码  低位在前 前6个字节参与校验码计算
    USHORT code = usMBCRC16(cmd,6);
    cmd[6] = code;
    cmd[7] = code >> 8;

    Inf_Modbus_Send(cmd,8);
}
```

modbus数据响应

Modbus 是「主从架构」，从站的响应数据**不会主动发送**，而是**被动触发自动返回**—— 只有当从站收到「合法且匹配的主站请求」后，才会由协议层（硬件 / 软件库）自动组装响应帧并回传给主站，无需人工干预（嵌入式开发中体现为 “配置好从站后，无需手动调用发送函数，协议栈自动处理”）。

```
主站（如PLC/上位机）                                    从站（STM32+Modbus协议栈）
  1. 发送请求帧（如02h读离散输入） → 总线 →  2. 硬件（UART/RS485）自动接收
                                                3. 协议栈自动校验：
                                                   - 从站地址是否匹配（自己的地址）？
                                                   - 功能码是否支持（如02h是否启用）？
                                                   - 数据格式/CRC是否正确？
                                                4. 校验通过 → 自动解析请求（地址+数量）
                                                5. 自动读取对应寄存器/IO状态（如离散输
入）
                                                6. 自动组装响应帧（数据+CRC）
  7. 接收响应帧 → 处理数据 ← 总线 ←  8. 自动发送响应帧（协议栈触发，无需手动调用
HAL_UART_Transmit）
```

“自动响应” 的 3 个前提条件

1. 从站地址匹配（最基础）
2. 请求帧合法（协议层面无错误）
3. 从站正常工作（硬件 + 软件就绪）

、Modbus 的优势（为什么不用自定义协议

1. **跨厂商兼容**：几乎所有工业设备（电机驱动、传感器、PLC）都支持 Modbus，不用和厂商对接自定义协议；
2. **极简易实现**：嵌入式 MCU（比如 STM32）只需几十行代码就能实现 Modbus RTU，不用复杂的解析逻辑；
3. **抗干扰强**：RTU 模式基于 RS485，能在工业现场长距离传输（几百米），抗电磁干扰；
4. **轻量**：没有复杂的包头 / 校验，适配 MCU、驱动板这类算力 / 内存有限的设备。