# 技术路线文档 2

撰写人：陆熠熠

撰写时间：2020.01.20——2020.01.30

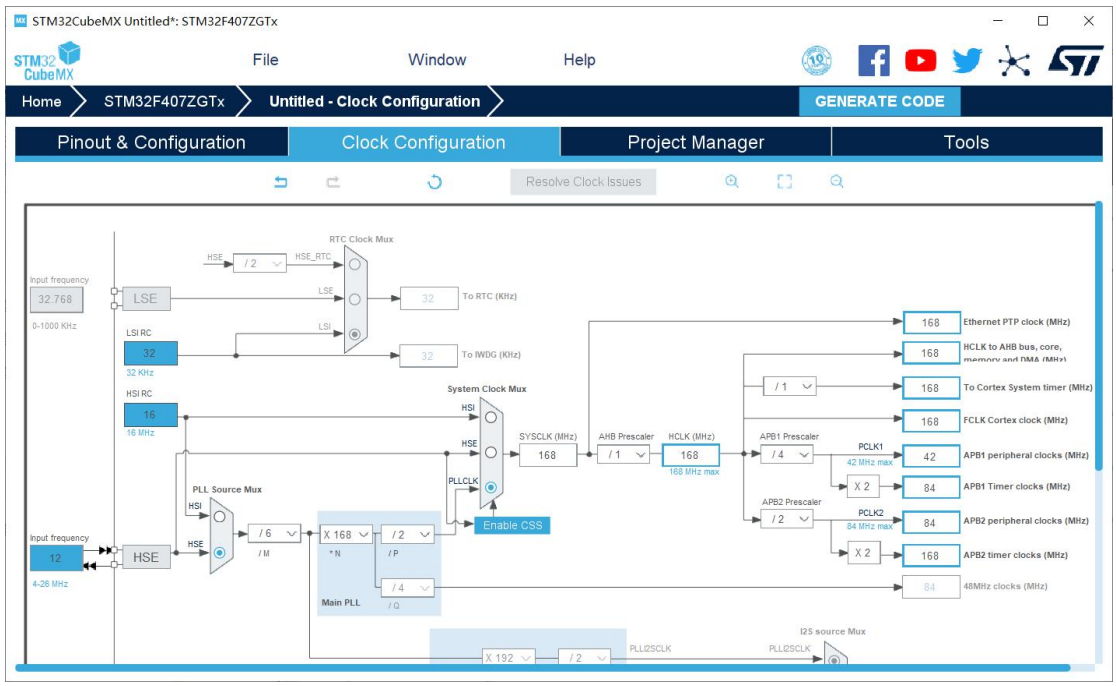
内容：2.0 基于 CubeMX 的环境搭建以及技术路线的继续

# 目录

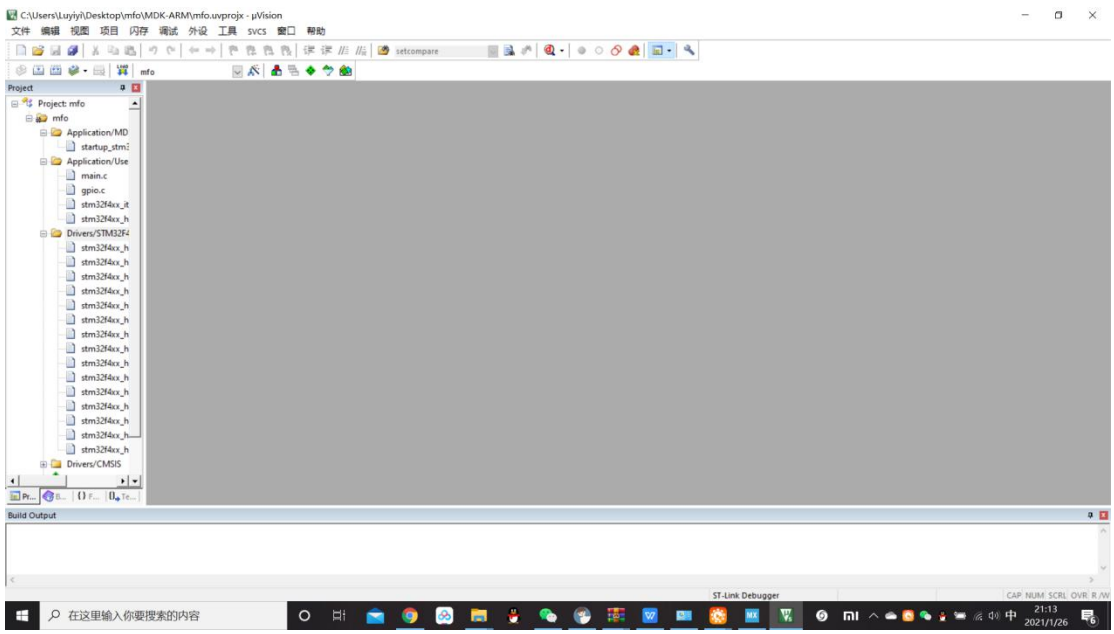
# 一、CubeMX 的环境搭建

## 1.1 过程

新建工程——选择芯片——外部时钟配置（石英/陶瓷 晶振）——主频配置（我还不知道是干嘛的来着）——输出设置——工程管理参数设置——选择存放位置(空格和中文字符)——选择代码调用方式




主频配置（例图）




建立成功的工程模板

## 1.2Yzy 的 csdn 一些知识点

- STM32CubeMX 中外部时钟配置可选类型为 ： Disable 、 BYPASS Clock Source（旁路时钟源） 、Crystal/Ceramic Resonator（石英/陶瓷 晶振） 三种类型。
- 旁路时钟源：指无需使用外部晶体时所需的芯片内部时钟驱动组件，直接从外界导入时钟信号。犹如芯片内部的驱动组件被旁路了。只需要外部提供时钟接入 OSC_IN 引脚，而 OSC_OUT 引脚悬空。
- 外部晶体/陶瓷谐振器(HSE 晶体)模式：该时钟源是由外部无源晶体与 MCU 内部时钟驱动电路共同配合形成，
- 有一定的启动时间，精度较高。OSC_IN 与 OSC_OUT 引脚都要连接。

## 1.3 一些遇到的问题

### 1.3.1 库下载失败

原因：路径问题（管理员模式打开）

### 1.3.2 MDK 代码无法生成

原因：JAVA 没有下载系统推荐的那个

# 二、HAL 库和标准库的区别

1）ST 为开发者提供了非常方便的开发库。到目前为止，有标准外设库(STD 库)、HAL 库、LL 库 三种。其中标准库与 HAL 库最常用，LL 库只是最近新添加的。
2）标准外设库（Standard Peripherals Library）是对 STM32 芯片的一个完整的封装，包括所有标准器件外设的器件驱动器。这应该是目前使用最多的 ST 库，几乎全部使用 C 语言实现。但是，标准外设库也是针对某一系列芯片而言的，没有可移植性。
3）HAL 库与新增的 LL 库，都是 ST 公司提供的新标准库，包含在 ST 为新的标准库注册了一个新商标：STMCube™当中。LL 库和 HAL 库两者相互独立，只不过 LL 库更底层。而且，部分 HAL 库会调用 LL 库（例如：USB 驱动）。同样，LL 库也会调用 HAL 库。
4）HAL 库更高的抽象整合水平，HAL API 集中关注各外设的公共函数功能，这样便于定义一套通用的用户友好的 API 函数接口，从而可以轻松实现从一个 STM32 产品移植到另一个不同的 STM32 系列产品。HAL 库是 ST 未来主推的库，从前年开始 ST 新出的芯片已经没有 STD 库了，比如 F7 系列。现在，ST 主推 HAL 库，目前，HAL 库已经支持 STM32 全线产品。
5）HAL 库与标准库比较，所用的 API 函数不一样，两者也相互独立，优势在于可以用 cubeMX 生成代码。（ST 专门为其开发了配套的桌面软件 STMCubeMX，开发者可以直接使用该软件进行可视化配置，大大节省开发时间。）

## 2.1 有关 HAL 库和标准库的语句差别

STM32 HAL 库与标准库的区别_浅谈句柄、MSP 函数、Callback 函数
https://blog.csdn.net/weixin_43186792/article/details/88759321

### 2.1.1 句柄

1）初始化他们的各个寄存器。在标准库中，这些操作都是利用固件库结构体变量+固件库 Init 函数实现的。需要对六个位置进行赋值，然后引用 Init 函数，并且 USART_InitStructure 并不是一个全局结构体变量，而是只在函数内部的局部变量，初始化完成之后，USART_InitStructure 就失去了作用。

```
    USART_InitTypeDef USART_InitStructure;
    USART_InitStructure.USART_BaudRate = bound;//串口波特率
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;//字长为 8 位数据格式
    USART_InitStructure.USART_StopBits = USART_StopBits_1;//一个停止位
    USART_InitStructure.USART_Parity = USART_Parity_No;//无奇偶校验位
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;//无硬件数
据流控制
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx; //收发模式
    USART_Init(USART3, &USART_InitStructure); //初始化串口 1
```

1）在 HAL 库中，同样是 USART 初始化结构体变量，我们要定义为全局变量。

```
UART_HandleTypeDef UART1_Handler;
```

结构体成员

```
    typedef struct
{
    USART_TypeDef                 *Instance;        /*!< UART registers base address        */
    UART_InitTypeDef              Init;             /*!< UART communication parameters      */
    uint8_t                       *pTxBuffPtr;      /*!< Pointer to UART Tx transfer Buffer */
    uint16_t                      TxXferSize;       /*!< UART Tx Transfer size              */
    uint16_t                      TxXferCount;      /*!< UART Tx Transfer Counter           */
    uint8_t                       *pRxBuffPtr;      /*!< Pointer to UART Rx transfer Buffer */
    uint16_t                      RxXferSize;       /*!< UART Rx Transfer size              */
    uint16_t                      RxXferCount;      /*!< UART Rx Transfer Counter           */
    DMA_HandleTypeDef             *hdmatx;          /*!< UART Tx DMA Handle parameters      */
    DMA_HandleTypeDef             *hdmarx;          /*!< UART Rx DMA Handle parameters      */
    HAL_LockTypeDef               Lock;             /*!< Locking object                     */
    __IO HAL_UART_StateTypeDef    State;            /*!< UART communication state           */
    __IO uint32_t                 ErrorCode;        /*!< UART Error code                    */
}UART_HandleTypeDef;
```

2）不仅包含了之前标准库就有的六个成员，还包含过采样、数据缓存、数据指针、串口 DMA 相关的变量、各种标志位等等要在整个项目流程中都要设置的各个成员。

3）该 UART1_Handler 就被称为串口的句柄。

```
HAL_UART_Receive_IT(&UART1_Handler, (u8 *)aRxBuffer, RXBUFFERSIZE);//开启中断
void HAL_UART_MspInit(UART_HandleTypeDef *huart);
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);//MSP 与 Callback 回调函数
```

5）实现函数的时候只需要调用初始化时定义的句柄 UART1_Handler。

### 2.1.2 Callback 函数

1）HAL 的串口中断函数不需要手动设置判断是否接收、读取中断

```
void USART1_IRQHandler(void)
```

```
{
    HAL_UART_IRQHandler(&UART1_Handler);     //调用 HAL 库中断处理公用函数
    /***************省略无关代码****************/
}
```

2）HAL_UART_IRQHandler 这个函数完成了判断是哪个中断（接收？发送？或者其他？），然后读出数据，保存至缓存区，顺便清除中断标志位等等操作

3）提前设置每接收 N 个字节，对 N 字节进行处理。——定义了串口接收缓存区——句柄里设置好了缓存区的地址、大小——接收完五个字节，HAL_UART_IRQHandler 才会执行一次 Callback 函数。

# 三、续 PWM 的研究和应用

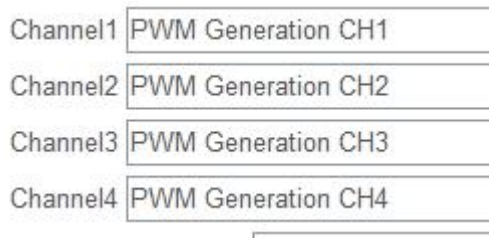
批注：这里的 PWM 的研究和应用由于书本所用的库跟 RM 比赛的库不同，因此理论部分主要来源于书本，而实践部分主要来自于各种搜索引擎、博客和教程。

## 3.1 使用 CubeMX 驱动舵机：

实践来源：Y 神的 csdn 链接。（yzy 简称 Y 神）
https://blog.csdn.net/programmaker3/article/details/112480660

### 3.1.1 前期准备

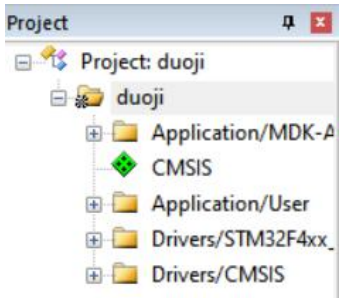
Y 神使用了为复杂功能做准备的 7 路 PWM 波。
开启并配置定时器 1，选择内部时钟，配置 4 个 PWM 通道




设置预分频值，重载值


对 PWM 通道的 Pulse 值进行设置


开启定时器 8，同理设置预分频值，重载值，打开 3 路 PWM 通道（图略）
建立成功的工程模板




### 3.1.2 主程序分析

调用部分库

```
#include "main.h"
#include "tim.h"
#include "gpio.h"
```

使能 HSE，配置 HSE 为 PLL 的时钟源，配置 PLL 的各种分频

```
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 6;
  RCC_OscInitStruct.PLL.PLLN = 168;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 4;
```

如果时钟跟 CubeMX 设置的不通过错误，报错

```
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
```

选择 PLLCLK 作为 SYSCLK，并配置 HCLK, PCLK1 and PCLK2 的时钟分频因子

```
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
```

```
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
```

在 main（）{}中设置的一些参数

```
HAL_TIM_Base_Start(&htimx);
HAL_TIM_PWM_Start(&htimx, TIM_CHANNEL_x);
```

Htimx——设置调用时钟种类 CHANNEL_X——设置调用的 PWM 波通道

```
_HAL_TIM_SetCompare(&htimx, TIM_CHANNEL_x, xxxxx);
```

__HAL_TIM_SetCompare 进行占空比设置，（比较值除以重载值），占空比的变化象征舵机的变化

### 3.1.3 主程序流程

HAL_Init 初始化——时钟配置——引脚配置——定时器配置——定时器工作——使能 PWM 输出——占空比——延时函数——指令

## 3.2 使用 CubeMX 驱动三种型号的 RM 电机。

### 3.2.1 直流电机调速与调向的原理

1）调向：正传与反转
以 L298N 为例子 IN1 高电平、IN2 低电平（正转），IN1 低电平、IN2 高电平（反转）。

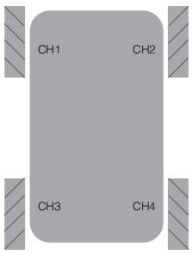
2）调速

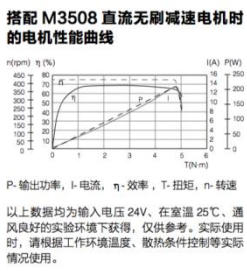
- 直流电机的调速——改变在马达上的供电电压。
- 不是简单的使能，而是通过把 PWM 信号加载使能信号引脚 ENA 上。

PS：不同型号的直流电机 PWM 所用的频率（周期）是不一样的。一般由厂家提供或者实验测得，错误运用可能会堵转或者烧毁电机。

### 3.3.2 底盘电机 M3508

1）使用方法 CH1\2\3\4 标记四个电机、电调位置



2）使用遥感情况下的指令设置（-反推【前右】设置，+正推【后左】）

CH1= Thr+ Rud+ Ail
CH2=- Thr+ Rud+ Ail
CH3= Thr+ Rud- Ail
CH4=- Thr+ Rud- Ail

3）有关程序部分（以大疆给的例程为例，进行剖析）

结构


代码及其批注
该例程用到的库

```
#include "stm32f4xx_hal.h"//调用 HAL 库
#include "cmsis_os.h"//调用系统封装接口层
// 采用这个接口层写程序，基本上可以说不用再去管所用的是什么操作系统。
//相同的代码可以轻而易举的移植到不同的实时系统中。并不是没有改动，但一定是最少的。
#include "can.h"
//上位机软件，需要用到 CAN 通信
```

直接存储器访问。DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据的通路，能使 CPU 的效率大为提高。

```
#include "dma.h"
#include "spi.h"
//SPI 的库的用法初始化，使能 SPI，使能 SPI 中断，SPI 发送接收函数。好像是一个通讯方式
```

```
#include "tim.h"
//定时器
#include "usart.h"
//USART 接收串口助手的发送的数据
#include "gpio.h"
//gpio 输入输出口的配置
```

主函数部分

```
int main(void)
{
HAL_Init();
SystemClock_Config();
MX_GPIO_Init();
MX_DMA_Init();
MX_CANx_Init();
//CAN 配置使用了 1，2
MX_USARTx_UART_Init();
//USART 配置使用了 1，2，3，6
MX_TIMx_Init();
//TIM 配置使用了 2，3，4，5，8，12
```

SPI5 配置使用了 5

HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size, uint32_t Timeout);//发送数据

HAL_StatusTypeDef HAL_SPI_Receive(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size, uint32_t Timeout);//接收数据

```
MX_SPI5_Init();
MX_FREERTOS_Init();
osKernelStart();
```

接下来就是 CubeMX 自带的对于系统时钟的配置。我就不赘述了

主要补充一些零星的知识点

- STM32 可以使用三种不同的时钟源来驱动系统时钟 (SYSCLK)：

HSI 振荡器时钟

HSE 振荡器时钟

主 PLL (PLL) 时钟

一些摘录：

HSE clock：The high speed external clock signal (HSE) can be generated from two possible clock sourses:

– HSE external crystal/ceramic resonator

外部晶体/陶瓷谐振器

–HSE external user clock

震荡器和负载电容器必须尽可能靠近振荡器引脚，以尽量减少输出实真和启动稳定时间。这个负载电容值必须根据所选振荡器进行调整。

HSI clock：The HSI clock signal is generated from an inernal 16MHz RC oscillator(RC 震荡器) and can be used directly as a system clock, or uesd as PLL input.

HSI 时钟信号由内部 16MHz RC 振荡器产生，可以使用直接作为系统时钟，或者做锁相环输入。
HSI RC 振荡器的优点是以低成本提供时钟源，有比 HSE 晶体振荡器更快的启动时间，然而，甚至通过校准，频率不如外部晶体振荡器或陶瓷精确振荡器。
PLL configuration：锁相环，一种振荡器的反馈技术，可以实现倍频
函数初始化
在不使用 FreeRTOS 的时候，SysTick 是默认的 HAL 基础时钟源，但是在 SYS 模块中，也可以选择其他定时器作为基础时钟源，例如可以选择基础定时器 TIM6 作为 HAL 的基础时钟源。

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{


  if (htim->Instance == TIM6) {
    HAL_IncTick();
  }
```

在使用 TIM6 作为 HAL 的基础时钟时，TIM6 完全替代了 SysTick 的功能
*HAL_Init()
此函数用来初始化 HAL 库，应当作为主函数中第一个被调用的函数。其主要功能:
1.配置 FLASH 预取，指令以及 Data 缓存
2.配置好 SysTick，使其每 1ms 产生一次中断
3.设置 NVIC 优先级分组为第四组(主优先级[0~15],子优先级[0])
4.调用回调函数 HAL_MspInit() 来进行全局低级硬件初始化(MSP:MCU Support Package).

4）部分解释以及原理理解

3508 用 can 只能是控制电流，电流对应电机的转矩。
具体的方向还是要看正转和反转，通过给定电流的正负来控制小车的方向。

5）问题

1）FreeRTOS 的概念还没有解决，有关其的库都已经搜出来了。

2）3508 在空载情况下，哪怕给一个较小的电流，电机也会一直转到一个很大的转速（只是力矩很小）。
解决：通过反馈的转速或者转角，外加一个 PID，来实现对电机的转速或转角的控制。

### 3.3.3 云台电机 GM6020

未完待续.....

### 3.3.4 摩擦轮电机 M2006

未完待续.....

# 四、CAN 通信的中断接收——电机底盘

## 4.1 有关 CAN 的概念

CAN：Controller Area Network，控制器局域网络。
1）CAN 总线上所有连接节点并没有地址的概念，通过发送数据中的标识符 ID（仲裁场）进行区分，多个单元同时发送消息时也通过标识符的优先级来仲裁优先发送方。
2）CAN 总线的信号是通过两根信号线之间的电压差值来确定信号类型，显性电平对应逻辑 0，隐形电平对应逻辑 1。（这部分都是由硬件电路实现的）
3）CAN 协议包括数据帧、遥控帧、错误帧、过载帧、间隔帧。重点关注数据帧，包括：帧起始、仲裁段、控制段、数据段、CRC 段、ACK 段、帧结束。

| 总线空闲 | 帧起始 | 仲裁场 | 控制场 | 数据场 | CRC 场 | 应答场 | 帧结尾 | 帧间隔 |
|---|---|---|---|---|---|---|---|---|

4）CAN 通信是一种半双工、异步通信。
5）CAN 总线由 CAN_H 和 CAN_L 两个线构成，各个设备一起挂在总线上。

## 4.2 有关数据帧 ID 的概念

ID 存储在数据帧的仲裁场，CAN 的 ID 分为标准 ID 和拓展 ID，标准 ID 长度位 11 位，拓展 ID 长度为 29 位。

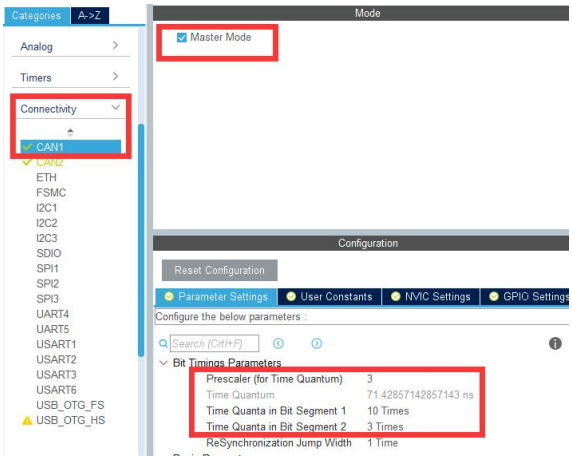
| 帧起始 | 仲裁场 | | | | | | | | | | | 控制场 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 标识符（ID）11 位 | | | | | | | | | | RTR | |
| | Bit 10 | Bit 9 | Bit 8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | | |

## 4.3 RM 比赛中的电调接受格式

将发送的 CAN 数据帧的 ID 设置为 0x20x（电调 ID），数据域中的 8Byte 数据按照电调 1 到 4 的搞八位和低八位的顺序装填，帧格式改为 DATA，DLC 是八字节，最后进行数据的发送。最后接受的数据和做出的指令如下。

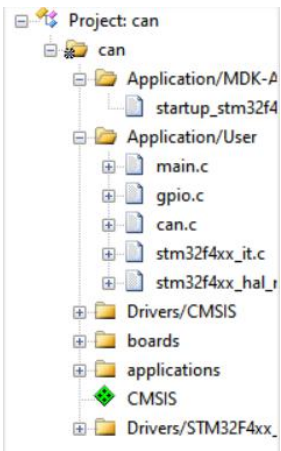
| 数据域 | 内容 |
| --- | --- |
| DATA[0] | 转子机械角度高 8 位 |
| DATA[1] | 转子机械角度低 8 位 |
| DATA[2] | 转子转速高 8 位 |
| DATA[3] | 转子转速低 8 位 |
| DATA[4] | 实际转矩电流高 8 位 |
| DATA[5] | 实际转矩电流低 8 位 |
| DATA[6] | 电机温度 |
| DATA[7] | Null |

## 4.4 CubeMX 的相关配置




选择芯片型号及封装类型——配置 CAN1（在 Connectivity 中）——Mode 中选择 Master Mode ——配置图示波特率——CAN2 同理

搭建好的模板如图所示：




### 4.4.1 需要修改和理解的 HAL 库函数

1）CAN_cmd_chassis 函数【CAN 库】

```
void CAN_cmd_chassis_reset_ID(void)
{
    uint32_t send_mail_box;
    chassis_tx_message.StdId = CAN_CHASSIS_ALL_ID;
    chassis_tx_message.IDE = CAN_ID_STD;
    chassis_tx_message.RTR = CAN_RTR_DATA;
    chassis_tx_message.DLC = 0x08;
    chassis_can_send_data[0] = motor1 >> 8;
    chassis_can_send_data[1] = motor1;
    chassis_can_send_data[2] = motor2>>8;
    chassis_can_send_data[3] = motor2;
    chassis_can_send_data[4] = motor3>>8;
    chassis_can_send_data[5] = motor3;
    chassis_can_send_data[6] = motor4>>8;
    chassis_can_send_data[7] = motor4;
    HAL_CAN_AddTxMessage(&CHASSIS_CAN, &chassis_tx_message, chassis_can_send_data, &send_mail
_box);
}
```

### 2）实现 CAN 发送的函数 HAL_CAN_AddTXMessage【CAN 库】

功能：将一段数据通过 CAN 总线发送

```
HAL_StatusTypeDef HAL_CAN_AddTxMessage(CAN_HandleTypeDef *hcan, CAN_TxHeaderTypeDef *pHeader,
 uint8_t aData[], uint32_t *pTxMailbox);
```

| | |
|---|---|
| CAN_HandleTypeDef *hcan | can 的指针，如果 canx 就输入&hcanx |
| CAN_TxHeaderTypeDef *pHeader | 包含 CAN 的 ID 格式等信息，CAN 数据帧信息的结构体 |
| uint8_t aData[] | 待发送的数据组名称 |
| uint32_t *pTxMailbox | 存储 CAN 发送所使用的邮箱号 |

### 3）向云台点击和发射机构点击发射控制信号的 CAN_cmd_gimbal

| | |
|---|---|
| Yaw | 轴电机 |
| Pitch | 轴电机 |
| Shoot | Rev |

```
void CAN_cmd_gimbal(int16_t yaw, int16_t pitch, int16_t shoot, int16_t rev)
{
    uint32_t send_mail_box;
    gimbal_tx_message.StdId = CAN_GIMBAL_ALL_ID;
    gimbal_tx_message.IDE = CAN_ID_STD;
    gimbal_tx_message.RTR = CAN_RTR_DATA;
    gimbal_tx_message.DLC = 0x08;
    gimbal_can_send_data[0] = (yaw >> 8);
    gimbal_can_send_data[1] = yaw;
    gimbal_can_send_data[2] = (pitch >> 8);
```

```
    gimbal_can_send_data[3] = pitch;
    gimbal_can_send_data[4] = (shoot >> 8);
    gimbal_can_send_data[5] = shoot;
    gimbal_can_send_data[6] = (rev >> 8);
    gimbal_can_send_data[7] = rev;
    HAL_CAN_AddTxMessage(&GIMBAL_CAN, &gimbal_tx_message, gimbal_can_send_data, &send_mail_bo
x);
}
```

### 4.4.2 CAN 终端回调函数

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    CAN_RxHeaderTypeDef rx_header;
    uint8_t rx_data[8];

    HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &rx_header, rx_data);
    switch (rx_header.StdId)
    {
        case CAN_3508_M1_ID:
        case CAN_3508_M2_ID:
        case CAN_3508_M3_ID:
        case CAN_3508_M4_ID:
        case CAN_YAW_MOTOR_ID:
        case CAN_PIT_MOTOR_ID:
        case CAN_TRIGGER_MOTOR_ID:
        {
            static uint8_t i = 0;
            //get motor id
            i = rx_header.StdId - CAN_3508_M1_ID;
            get_motor_measure(&motor_chassis[i], rx_data);
            break;
        }
        default:
        {
            break;
        }
    }
}
```

CAN 完成数据帧接受——调用中断处理函数——寄存器处理——调用 CAN 中断回调函数处理过程：判断 ID 是否正确——解码数据——装入点击数组 motor_chassis。

1）HAL_CAN_GetRxMessage，接受的时候嗲用了 HAL 库的接收函数，用来接收 CAN 总线上发送来的数据。

```
HAL_StatusTypeDef HAL_CAN_GetRxMessage(CAN_HandleTypeDef *hcan, uint32_t RxFifo, CAN_RxHeader
TypeDef *pHeader, uint8_t aData[]);
```

| | |
|---|---|
| HAL_StatusTypeDef | CAN 接收成功返回值 HAL_Ok |
| CAN_HandleTypeDef*hcan | 同上例 |
| Unint32_t RxFfifo | 接受 FIFO 号，CAN_RX_FIFO |
| CAN_RxHeaderTypeDef *pHeader | 同上例 |
| Unint8_t aData[] | 存储接收到的数据的数组名称 |

2）motor_chassis 为 motor_measure_t 类型的数组
包含：电机转子交角度，电机转子转速，控制电流，温度。

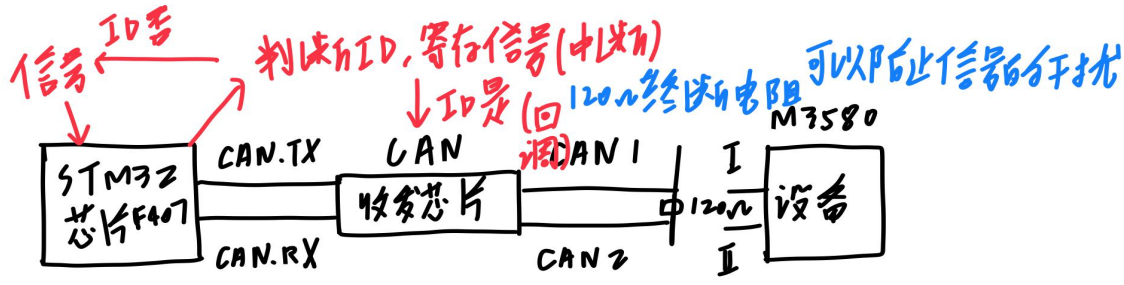
### 4.4.3 需要自己写的函数部分

解码：将接收到的数据按照高八位和低八位的方式进行拼接，从而得到电机的各个参数。

```
#define get_motor_measure(ptr, data)                                    \
    {                                                                   \
        (ptr)->last_ecd = (ptr)->ecd;                                   \
        (ptr)->ecd = (uint16_t)((data)[0] << 8 | (data)[1]);            \
        (ptr)->speed_rpm = (uint16_t)((data)[2] << 8 | (data)[3]);      \
        (ptr)->given_current = (uint16_t)((data)[4] << 8 | (data)[5]);  \
        (ptr)->temperate = (data)[6];                                   \
    }
```

### 4.4.5 完整流程和程序解释

硬件层




这里是 CAN 中断接收函数，接收电机数据,CAN 发送函数发送电机电流控制电机.

```
#ifndef CAN_RECEIVE_H
#define CAN_RECEIVE_H
```

```c
#include "struct_typedef.h"
#define CHASSIS_CAN hcan1
#define GIMBAL_CAN hcan2
typedef enum
{
    CAN_CHASSIS_ALL_ID = 0x200,
    CAN_3508_M1_ID = 0x201,
    CAN_3508_M2_ID = 0x202,
    CAN_3508_M3_ID = 0x203,
    CAN_3508_M4_ID = 0x204,
    CAN_YAW_MOTOR_ID = 0x205,
    CAN_PIT_MOTOR_ID = 0x206,
    CAN_TRIGGER_MOTOR_ID = 0x207,
    CAN_GIMBAL_ALL_ID = 0x1FF,
} can_msg_id_e;
//rm motor data
typedef struct
{
    uint16_t ecd;
    int16_t speed_rpm;
    int16_t given_current;
    uint8_t temperate;
    int16_t last_ecd;
} motor_measure_t;
/**
  * @brief          发送电机控制电流(0x205,0x206,0x207,0x208)
  * @param[in]      yaw: (0x205) 6020 电机控制电流, 范围 [-30000,30000]
  * @param[in]      pitch: (0x206) 6020 电机控制电流, 范围 [-30000,30000]
  * @param[in]      shoot: (0x207) 2006 电机控制电流, 范围 [-10000,10000]
  * @param[in]      rev: (0x208) 保留, 电机控制电流
  * @retval         none
  */
extern void CAN_cmd_gimbal(int16_t yaw, int16_t pitch, int16_t shoot, int16_t rev);
/**
  * @brief          发送 ID 为 0x700 的 CAN 包,它会设置 3508 电机进入快速设置 ID
  * @param[in]      none
  * @retval         none
  */
extern void CAN_cmd_chassis_reset_ID(void);
/**
  * @brief          发送电机控制电流(0x201,0x202,0x203,0x204)
  * @param[in]      motor1: (0x201) 3508 电机控制电流, 范围 [-16384,16384]
  * @param[in]      motor2: (0x202) 3508 电机控制电流, 范围 [-16384,16384]
  * @param[in]      motor3: (0x203) 3508 电机控制电流, 范围 [-16384,16384]
```

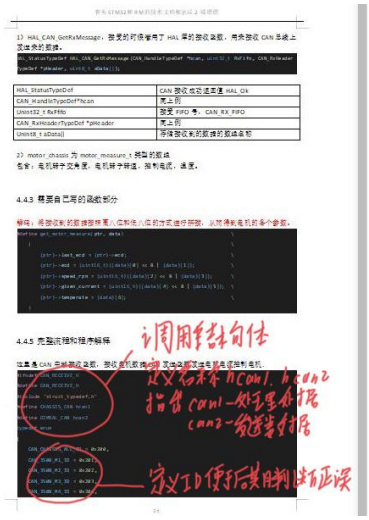
```
  * @param[in]      motor4: (0x204) 3508 电机控制电流, 范围 [-16384,16384]
  * @retval         none
  */
extern void CAN_cmd_chassis(int16_t motor1, int16_t motor2, int16_t motor3, int16_t motor4);
/**
  * @brief          返回 yaw 6020 电机数据指针
  * @param[in]      none
  * @retval         电机数据指针
  */
extern const motor_measure_t *get_yaw_gimbal_motor_measure_point(void);
/**
  * @brief          返回 pitch 6020 电机数据指针
  * @param[in]      none
  * @retval         电机数据指针
  */
extern const motor_measure_t *get_pitch_gimbal_motor_measure_point(void);
/**
  * @brief          返回拨弹电机 2006 电机数据指针
  * @param[in]      none
  * @retval         电机数据指针
  */
extern const motor_measure_t *get_trigger_motor_measure_point(void);
/**
  * @brief          返回底盘电机 3508 电机数据指针
  * @param[in]      i: 电机编号,范围[0,3]
  * @retval         电机数据指针
  */
extern const motor_measure_t *get_chassis_motor_measure_point(uint8_t i);
#endif
```

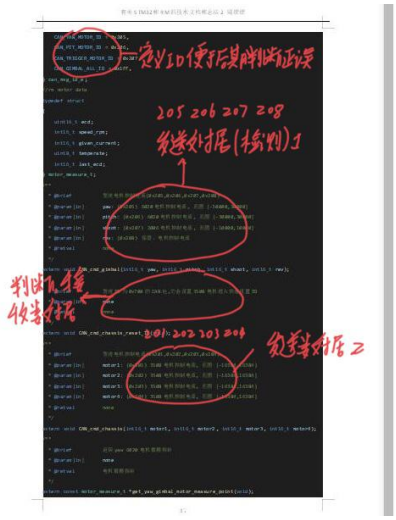
给电机发送的数据（我只是猜是这个其实我也不是很肯定）

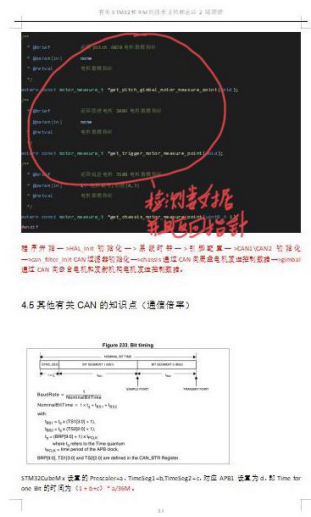
```
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
        CAN_cmd_chassis(4000, 4000, 4000, 4000);
        HAL_Delay(2);
        CAN_cmd_gimbal(10000, 10000, 10000, 10000);
        HAL_Delay(2);
  }
```

基本结构



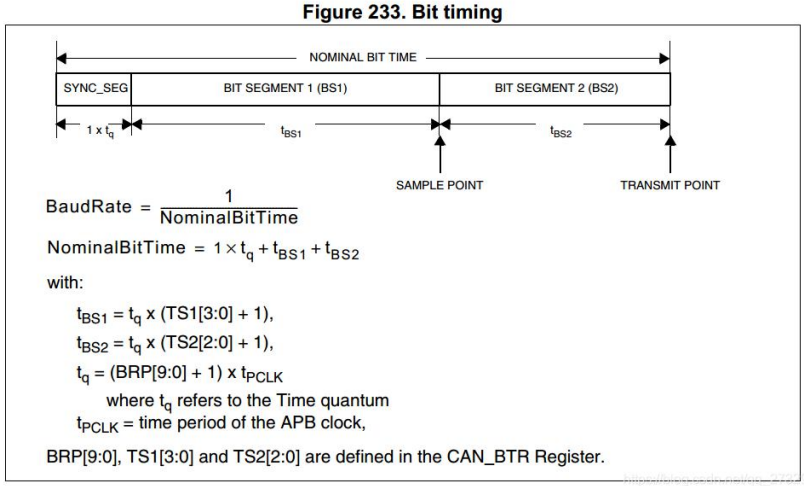




程序开始—>HAL_Init 初始化—>系统时钟—>引脚配置—>CAN1\CAN2 初始化—>can_filter_init CAN 过滤器初始化—>chassis 通过 CAN 向底盘电机发送控制数据—>gimbal 通过 CAN 向云台电机和发射机构电机发送控制数据。

## 4.5 其他有关 CAN 的知识点（通信倍率）

**Figure 233. Bit timing**




STM32CubeMx 设置的 Prescaler=a，TimeSeg1=b,TimeSeg2=c，对应 APB1 设置为 d，即 Time for one Bit 的时间为（1 + b+c）* a/36M。