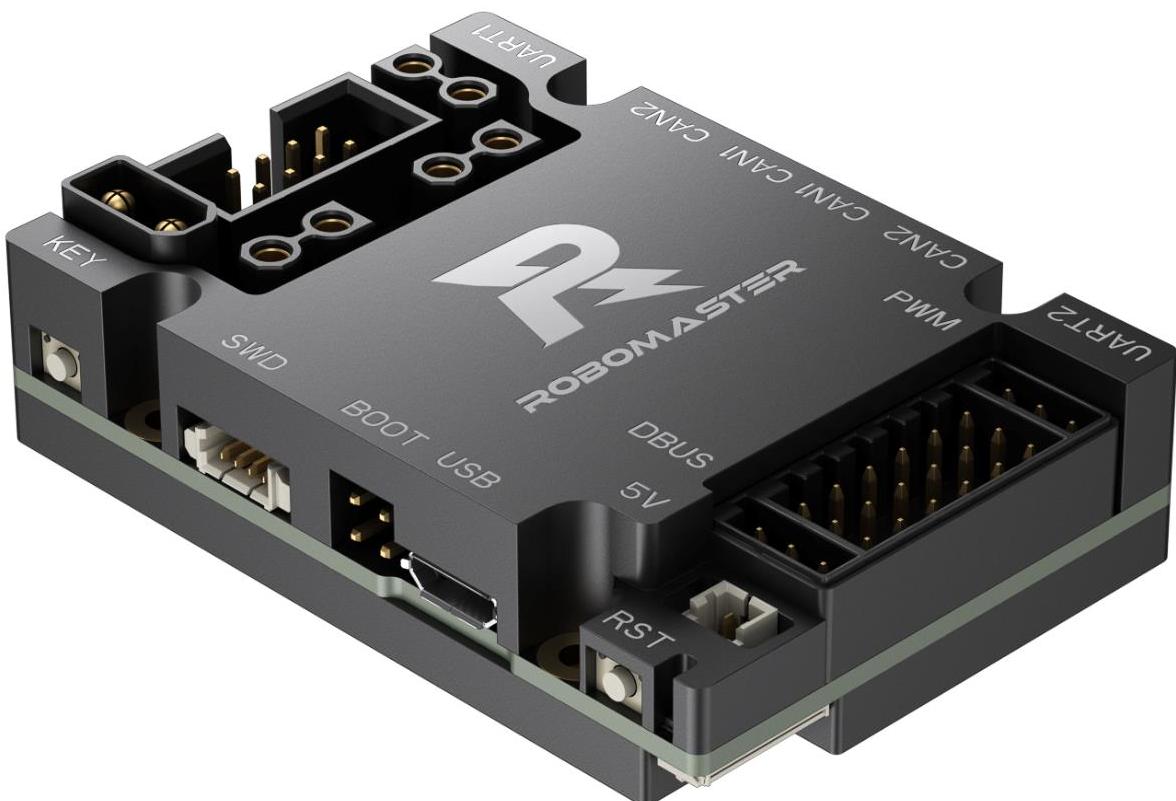


RoboMaster 开发板 C 型嵌入式软件 教程文档

v1.0 2020.01



前置参考阅读

- 1.《RoboMaster 开发板 C 型用户手册》
2. C 语言相关书籍
- 3.《ARM Cortex-M3 与 Cortex-M4 权威指南》
4. STM32F407IG 相关数据手册
5. freeRTOS 官网相关文档

建议用户首先阅读《RoboMaster 开发板 C 型用户手册》，了解 RoboMaster 开发板 C 型（以下简称开发板 C 型）相关功能以及使用方法，正确使用开发板 C 型的相关接口，避免错误的使用方法造成开发板 C 型的损坏；开发板 C 型的例程所使用的编程语言为 C 语言，建议用户学习 C 语言的基本语法，本文档没有针对 C 语言进行系统的讲解；《ARM Cortex-M3 与 Cortex-M4 权威指南》、STM32F407IG 相关数据手册和 freeRTOS 官网相关文档作为参考资料，在必要时候可以进行查阅。

修改日志

日期	版本	修改记录
2020.01.09	V1.0	首次发布。

目录

前置参考阅读.....	2
修改日志.....	2
0. 开发板 C 型, cubeMX 与 keil 入门.....	12
0.1 知识要点.....	12
0.2 课程内容.....	12
0.3 基础学习.....	12
0.4 程序学习.....	14
0.4.1 软件环境.....	14
0.4.2 cubeMX 新建工程.....	15
0.4.3 keil 软件简易入门.....	18
0.4.4 Keil 的调试模式.....	22
0.5 RoboMaster 机器人功能简介.....	24
0.6 课程总结.....	24
1. 点亮 LED.....	25
1.1 知识要点.....	25
1.2 课程内容.....	25
1.3 基础学习.....	25
1.3.1 LED 灯基本知识.....	25
1.4 程序学习.....	26
1.4.1 cubeMX 中配置 GPIO 基本操作.....	26
1.4.2 HAL_GPIO_WritePin 函数讲解.....	27
1.4.3 程序流程.....	28
1.4.4 效果展示.....	29
1.5 进阶学习.....	30
1.5.1 三极管的通断特性.....	30
1.5.2 LED 的下拉电阻.....	30
1.5.3 硬件原理图上的限流电阻.....	32
1.6 课程总结.....	32
2. 闪烁 LED.....	34
2.1 知识要点.....	34
2.2 课程内容.....	34
2.3 基础学习.....	34
2.3.1 GPIO 的翻转速度.....	34

2.4	程序学习	36
2.4.1	计数延时介绍	36
2.4.2	nop 延时介绍	37
2.4.3	滴答计时器介绍以及 HAL_Init 初始化	38
2.4.4	HAL_Delay 介绍	40
2.4.5	HAL_GPIO_TogglePin 介绍	41
2.4.6	程序流程	42
2.4.7	效果展示	43
2.5	课程总结	44
3.	定时器闪烁 LED	45
3.1	知识要点	45
3.2	课程内容	45
3.3	基础学习	45
3.3.1	定时器讲解	45
3.3.2	中断讲解	46
3.4	程序学习	47
3.4.1	定时器在 cubeMX 中配置	47
3.4.2	中断优先级讲解	50
3.4.3	cubeMX 中的中断配置以及中断函数管理	51
3.4.4	定时器回调函数介绍	53
3.4.5	HAL_TIM_Base_Start 函数	54
3.4.6	程序流程	55
3.4.7	效果展示	56
3.5	进阶学习	56
3.5.1	APB 总线计算定时器定时时间	56
3.6	课程总结	59
4.	PWM 控制 LED 的亮度	60
4.1	知识要点	60
4.2	课程内容	60
4.3	基础学习	60
4.3.1	PWM 基本知识	60
4.3.2	aRGB 三原色	61
4.4	程序学习	61
4.4.1	PWM 在 cubeMX 中配置	61

4.4.2	PWM 配置介绍	63
4.4.3	HAL_TIM_PWM_Start 函数介绍	64
4.4.4	程序流程	66
4.4.5	效果展示	67
4.5	课程总结	67
5.	常见的 PWM 设备	68
5.1	知识要点	68
5.2	课程内容	68
5.3	基础学习	68
5.3.1	蜂鸣器	68
5.3.2	舵机的控制	69
5.4	程序学习	71
5.4.1	蜂鸣器的 PWM 在 cubeMX 中配置	71
5.4.2	蜂鸣器的程序流程	73
5.4.3	效果展示	76
5.4.4	舵机的 PWM 在 cubeMX 中配置	76
5.4.5	舵机主程序讲解	79
5.4.6	舵机效果演示	80
5.5	课程总结	81
5.6	上一节作业讲解	82
	程序流程	82
6.	按键的外部中断	85
6.1	知识要点	85
6.2	课程内容	85
6.3	基础学习	85
6.3.1	按键原理图介绍	85
6.3.2	按键软件消抖	86
6.3.3	外部中断	87
6.4	程序学习	87
6.4.1	外部中断在 cubeMX 中的配置	87
6.4.2	HAL_GPIO_ReadPin 函数介绍	89
6.4.3	中断回调函数介绍	89
6.4.4	程序中的前后台	90
6.4.5	程序流程	90

6.4.6	效果展示	91
6.5	课程总结	92
7.	ADC 采样电池电压.....	93
7.1	知识要点	93
7.2	课程内容	93
7.3	基础学习	93
7.3.1	ADC 原理介绍.....	93
7.3.2	电阻分压电路介绍.....	95
7.4	程序学习	96
7.4.1	ADC 在 cubeMX 中的配置.....	96
7.4.2	内部 VREFINT 电压的使用.....	98
7.4.3	ADC 采样相关函数介绍	99
7.4.4	程序流程	101
7.4.5	效果展示	103
7.5	进阶学习	104
7.6	课程总结	106
8.	串口收发.....	107
8.1	知识要点	107
8.2	课程内容	107
8.3	基础学习	107
8.3.1	串口接收中断与空闲中断.....	107
8.4	程序学习	108
8.4.1	串口在 cubeMX 中配置.....	108
8.4.2	串口接收中断与空闲中断.....	110
8.4.3	串口发送函数与中断函数.....	110
8.4.4	程序流程	111
8.4.5	效果展示	112
8.5	进阶学习	114
8.5.1	APB 时钟计算串口波特率.....	114
8.6	课程总结	114
9.	串口打印遥控器数据	116
9.1	知识要点	116
9.2	课程内容	116
9.3	基础学习	116

9.3.1	DMA 功能介绍	116
9.3.2	DBUS 协议介绍	116
9.4	程序学习	117
9.4.1	串口发送的 DMA 配置	117
9.4.2	printf 函数实现过程	120
9.4.3	串口的 DMA 接收与发送配置	120
9.4.4	程序流程	127
9.4.5	效果展示	127
9.5	课程总结	129
10.	Flash 读写	130
10.1	知识要点	130
10.2	课程内容	130
10.3	基础学习	130
10.3.1	stm32 的 flash 介绍	130
10.4	程序学习	131
10.4.1	flash 擦除函数介绍	131
10.4.2	flash 写入函数介绍	131
10.4.3	flash 读取介绍	132
10.4.4	flash 相关操作函数	133
10.4.5	程序流程	134
10.4.6	效果展示	134
10.5	进阶学习	136
10.5.1	flash 页分区	136
10.5.2	boot 作用	137
10.6	课程总结	138
11.	I2C 读取 IST8310	139
11.1	知识要点	139
11.2	课程内容	139
11.3	基础学习	139
11.3.1	I2C 简介	139
11.3.2	磁力计简介	140
11.4	软件学习	141
11.4.1	硬件接线	141
11.4.2	I2C 在 cubeMX 中的配置	141

11.4.3	主要函数介绍	143
11.4.4	程序流程	147
11.4.5	效果展示	148
11.5	进阶学习	148
11.5.1	IST8310 的读写过程	148
11.5.2	IST8310 的寄存器信息	154
11.6	课程总结	159
12.	OLED 显示	160
12.1	知识要点	160
12.2	课程内容	160
12.3	12.3 基础学习	160
12.3.1	OLED 简介	160
12.4	软件学习	161
12.4.1	硬件接线	161
12.4.2	cubeMX 配置过程	162
12.4.3	主要函数介绍	164
12.4.4	程序流程	171
12.4.5	效果展示	172
12.5	进阶学习	173
12.5.1	OLED 通信过程	173
12.5.2	OLED 初始化配置	174
12.6	课程总结	175
13.	BMI088 传感器	176
13.1	知识要点	176
13.2	课程内容	176
13.3	基础学习	176
13.3.1	陀螺仪简介	176
13.3.2	加速度计简介	176
13.3.3	SPI 协议简介	177
13.4	程序学习	179
13.4.1	SPI 在 cubeMX 中的配置	179
13.4.2	BMI088 的寄存器简介	181
13.4.3	BMI088 读取函数介绍	182
13.4.4	程序流程	185

13.4.5	效果演示.....	186
13.5	进阶学习	186
13.6	课程总结	188
14.	CAN 控制 RM 电机	189
14.1	知识要点	189
14.2	课程内容	189
14.3	基础学习	189
14.3.1	CAN 协议简介.....	189
14.3.2	RM 电机使用.....	190
14.4	程序学习	192
14.4.1	CAN 在 cubeMX 中的配置.....	192
14.4.2	CAN 发送函数介绍	195
14.4.3	CAN 接收中断回调介绍	197
14.4.4	程序流程.....	201
14.4.5	效果演示.....	201
14.5	进阶学习	203
14.5.1	CAN 波特率介绍	203
14.5.2	直流电机介绍	204
14.6	课程总结	207
15.	freeRTOS 闪烁 LED	208
15.1	知识要点	208
15.2	课程内容	208
15.3	基础学习	208
15.3.1	操作系统简介	208
15.4	程序学习	209
15.4.1	cubeMX 中 freeRTOS 的配置.....	209
15.4.2	cubeMX 中创建任务	211
15.4.3	程序中创建任务.....	215
15.4.4	程序流程.....	218
15.4.5	效果演示.....	218
15.5	进阶学习	220
15.6	课程总结	223
16.	IMU 温度控制	224
16.1	知识要点	224

16.2	课程内容	224
16.3	基础学习	224
16.3.1	IMU 控制温度的意义.....	224
16.3.2	PID 控制简介	225
16.4	程序学习	228
16.4.1	任务唤醒功能	228
16.4.2	控制链路.....	229
16.4.3	PID 初始化以及计算函数.....	230
16.4.4	程序流程.....	233
16.4.5	效果演示.....	233
16.5	进阶学习	234
16.6	课程总结	240
17.	底盘控制任务	241
17.1	知识要点	241
17.2	课程内容	241
17.3	基础学习	241
17.3.1	麦克纳姆轮的结构.....	241
17.3.2	麦克纳姆轮的安装.....	242
17.3.3	底盘正运动学	243
17.3.4	电机的速度环控制.....	245
17.4	程序学习	245
17.4.1	can 发送电机控制函数回顾.....	245
17.4.2	程序流程讲解	246
17.4.3	关键函数讲解	247
17.4.4	效果展示.....	253
17.5	进阶学习	254
17.5.1	底盘运动学的正运动过程.....	254
17.6	课程总结	257
18.	姿态解算任务	258
18.1	知识要点	258
18.2	课程内容	258
18.3	基础学习	258
18.3.1	姿态角简介	258
18.3.2	四元数与姿态角的转化.....	259

18.4	程序学习	260
18.4.1	cubeMX 中配置 GPIO,SPI 以及 I2C.....	260
18.4.2	mahony 算法移植	262
18.4.3	程序流程.....	267
18.4.4	效果展示.....	271
18.5	进阶学习	271
18.5.1	欧拉角旋转顺序.....	271
18.6	课程总结	273
19.	云台控制任务	274
19.1	知识要点	274
19.2	课程内容	274
19.3	基础学习	274
19.3.1	机器人云台的结构.....	274
19.3.2	串级 PID.....	275
19.4	程序学习	275
19.4.1	can 发送电机控制函数回顾.....	275
19.4.2	程序流程讲解	276
19.4.3	关键函数讲解	277
19.5	效果展示	282
19.6	课程总结	283
20.	机器人功能介绍	284
20.1	知识要点	284
20.2	课程内容	284
20.3	基础学习	284
20.3.1	机器人软件框架.....	284
20.3.2	功能介绍.....	285
20.4	程序学习	286
20.4.1	校准任务以及离线任务介绍	286
20.4.2	OLED 任务框架	290
20.4.3	裁判系统串口协议解包.....	291
20.4.4	外设调用关系	293
20.4.5	效果展示.....	295
20.5	课程总结	295

0. 开发板 C 型, cubeMX 与 keil 入门

0.1 知识要点

- 开发板 C 型出厂程序功能介绍
- cubeMX 从新建 ioc 工程到生成 keil 工程
- keil 工程设置介绍
- keil 软件调试功能介绍
- RoboMaster 机器人总体功能简介

0.2 课程内容

本课程中，首先介绍 RoboMaster 开发板 C 型（以下简称开发板 C 型）出厂程序功能；之后学习如何使用 cubeMX 生成 keil 工程，学习 stm32 的 keil 工程常见的设置，学习 keil 软件如何进入调试模式；最后作为教程的开始，总体了解 RoboMaster 机器人的常见功能以及对应使用到 stm32 的外设功能，指导之后的学习。

0.3 基础学习

RoboMaster 开发板 C 型采用高性能的 stm32 主控芯片，支持宽电压输入，集成专用的扩展接口，通信接口以及高精度 IMU 传感器，可配合 RoboMaster 产品或者其他配件使用。开发板 C 型具有如下外设：用户自定义 LED、5V 接口、BOOT 配置接口、micro USB 接口、SWD 接口、按键、可配置 I/O 接口、UART 接口、CAN 总线接口、PWM 接口、DBUS 接口、数字摄像头 FPC 接口、蜂鸣器、电压检测 ADC、六轴惯性测量单元和磁力计。

开发板 C 型出厂已烧录程序，可通过 micro USB 线连接 PC，使用串口工具对开发板 C 型的常用外设进行操作，操作如下所示：

- 在一级菜单选择界面上，可通过串口工具输入数字 1-9，选择对应的二级显示界面；在二级显示界面通过串口工具输入字母 q 或者 Q 退出二级显示界面。其中一级选择界面如图所示。

```
*****
1. LED TEST
2. BUZZER TEST
3. LASER TEST
4. KEY TEST
5. ADC TEST
6. IMU TEST
7. DBUS TEST
8. PWM TEST
9. CAN TEST
*****
```

- LED 正常状态为三色 LED 依次点亮。在 LED 显示界面显示当前 LED 状态：“LED ON”和“LED OFF”。LED ON 为三色 LED 均点亮，发出白光；LED OFF 指三色 LED 不是全部点亮。可使用串口工具输入 ON 或者 OFF 进行 LED 状态切换，LED 显示界面如下图所示。

```
*****
LED : OFF
*****
LED : ON
*****
```

- 蜂鸣器在开发板 C 型上电时会响起开机音效。在蜂鸣器显示界面显示当前蜂鸣器状态：“BUZZER OFF”和“BUZZER ON”。BUZZER OFF 为蜂鸣器不发出响声；BUZZER ON 为蜂鸣器响起《机甲大师》的主题曲《你》。可使用串口工具输入 ON 或者 OFF 进行蜂鸣器状态切换，蜂鸣器显示界面如下图所示。

```
*****
BUZZER: OFF
*****
BUZZER: ON
*****
```

- 在 5V 接口显示界面显示当前 5V 接口状态：“LASER OFF”和“LASER ON”。LASER OFF 为 5V 接口不输出；LASER ON 为 LASER。可使用串口工具输入 ON 或者 OFF 进行 5V 接口状态切换，5V 接口显示界面如下图所示。

```
*****
LASER: OFF
*****
LASER: ON
*****
```

- 在按键显示界面显示当前按键状态：“KEY OFF”和“KEY ON”。KEY OFF 为按键处在未按下状态；KEY ON 为按键处在已按下状态，按键显示界面如下图所示。

```
*****
KEY : OFF
*****
KEY : ON
*****
```

- 在电源电压显示界面显示当前电源电压：“BATTERY VOLTAGE: 24.000V”。其中

24.000V 为当前电源电压，实际数值以测量为准，LED 显示界面如电源电压显示界面如下图所示。

```
*****
| BATTERY VOLTAGE : 24.383
*****

```

- 在 IMU 显示界面显示陀螺仪、加速度计和磁力计数据，IMU 显示界面如下图所示。

```
*****
| GYRO : 0.366° /s,      0.366° /s,      -0.183° /s
| ACCEL: -0.156m/s²,    0.109m/s,      9.692m/s
| MAG : 4.800uT,        0.000uT,      3872.400uT
| TEMP: 49.500 °C
*****

```

- 在遥控器显示界面显示当前遥控器数据状态，遥控器数据显示界面如下图所示。

```
*****
| RC->CH[0-4]: 0,      24,      57,      5,      94,
| RC->S[0-1] : 3,      3
*****

```

- 在 PWM 显示界面显示当前 PWM 输出高电平时间状态：“PWM: 1000”。其中 1000 代表高电平时间为 1000ms，PWM 显示界面如下图所示。

```
*****
| PWM : 500
*****
| PWM : 2500
*****

```

- 在 CAN 显示界面显示当前 CAN 接收数据包个数情况：“CAN1 RECEIVE NUM: 1000”，“CAN2 RECEIVE NUM: 1000”。其中 1000 代表 1 秒内接收到数据包个数为 1000 个，CAN 显示界面如下图所示。

```
*****
| CAN1 RECEIVE NUM : 502
| CAN2 RECEIVE NUM : 502
*****

```

0.4 程序学习

0.4.1 软件环境

Toolchain/IDE : MDK-ARM V5

STM32F4xx_DFP Packs:2.13.0

STM32CubeMx:5.2.1

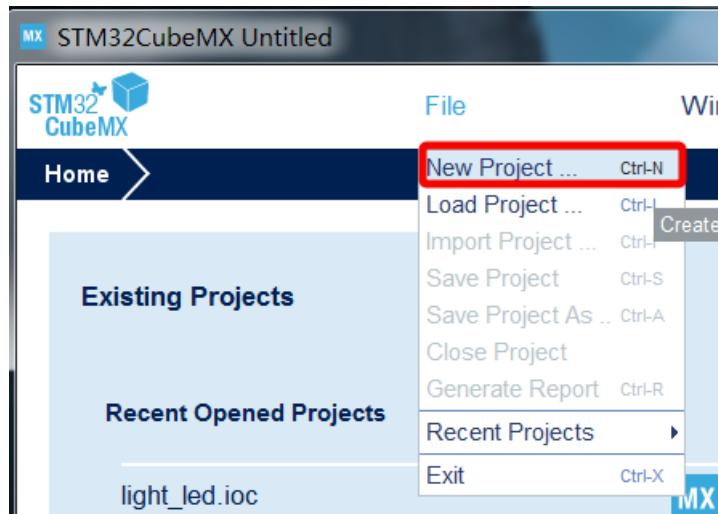
package version: STM32Cube FW_F4 V1.21.1

FreeRTOS version: 10.0.1

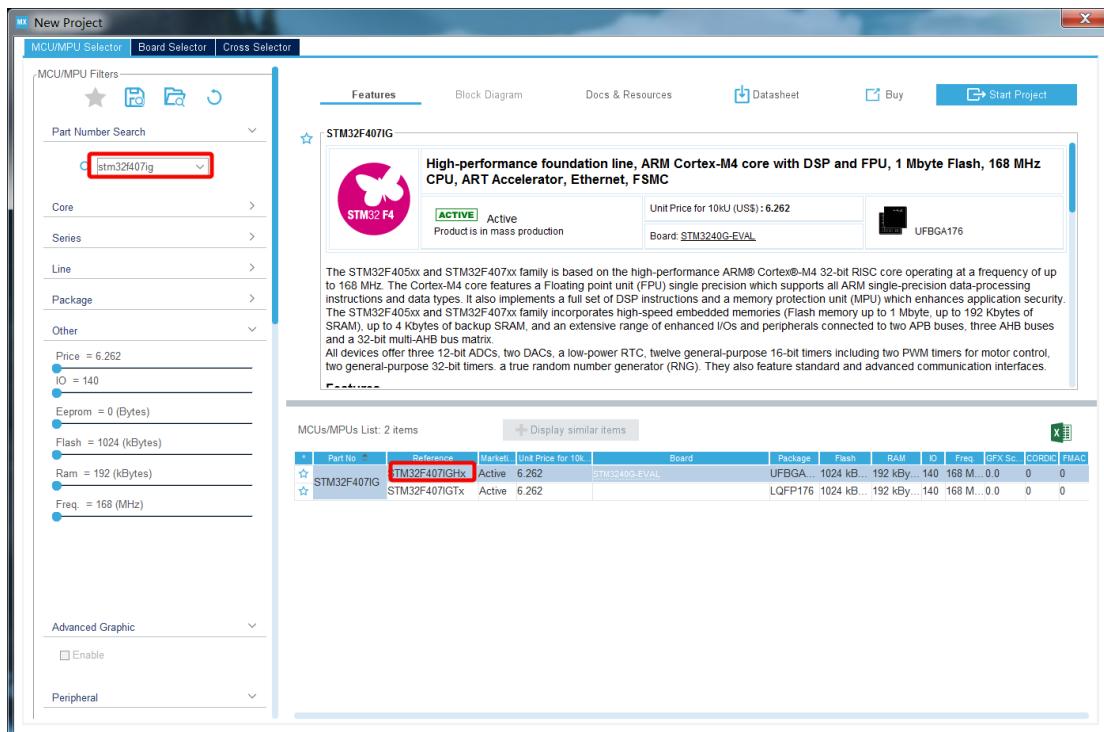
CMSIS-RTOS version: 1.02

0.4.2 cubeMX 新建工程

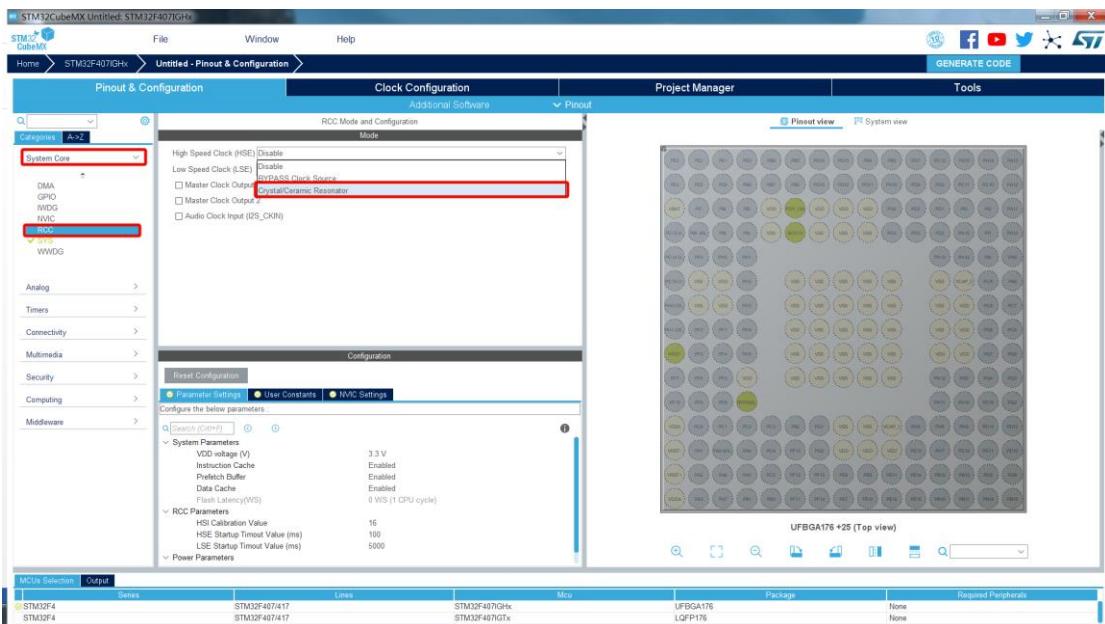
1. 打开 cubeMX 软件，在 file 选项中选择 “New Project”；



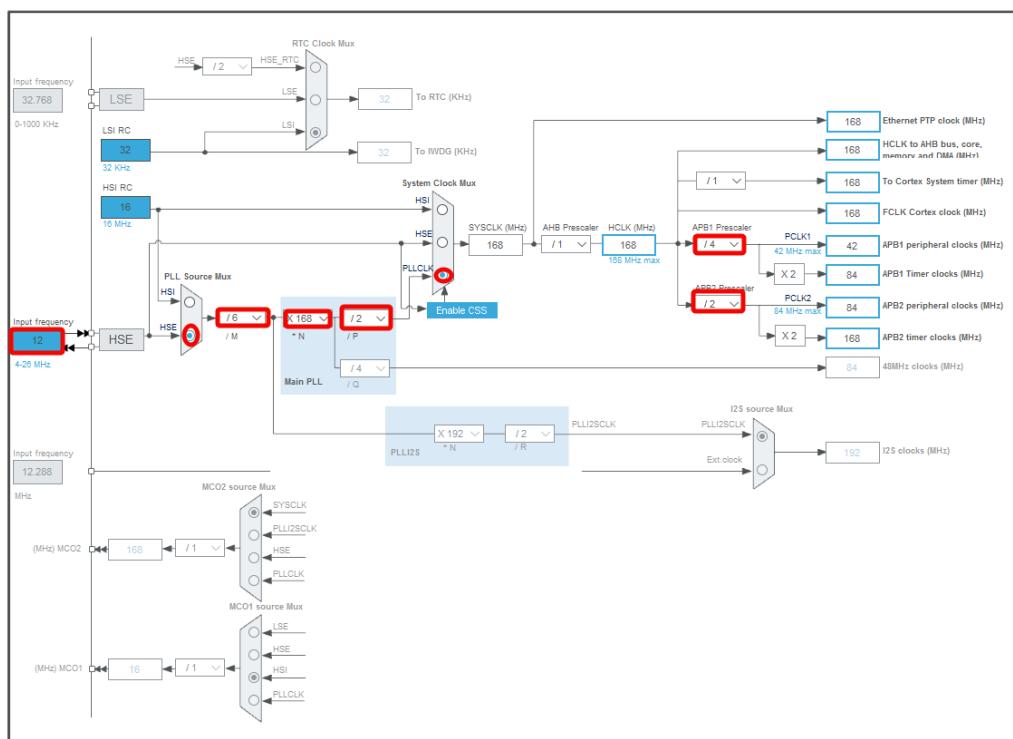
2. 搜索 stm32f407ig，选择 STM32F407IGHx 芯片；



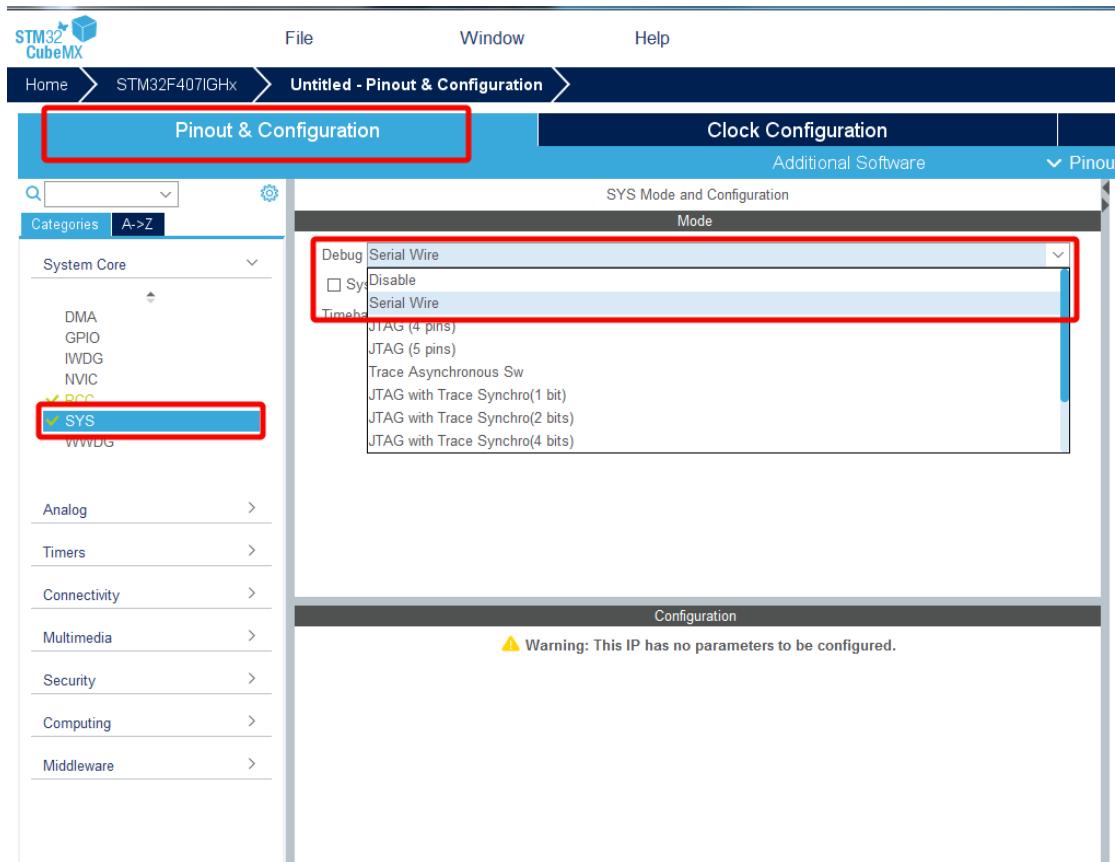
3. 在 System Core 下选择 RCC 选项，在 RCC mode and Configuration 中的 High Speed Clock(HSE)下选择 Crystal/Ceramic Resonator；



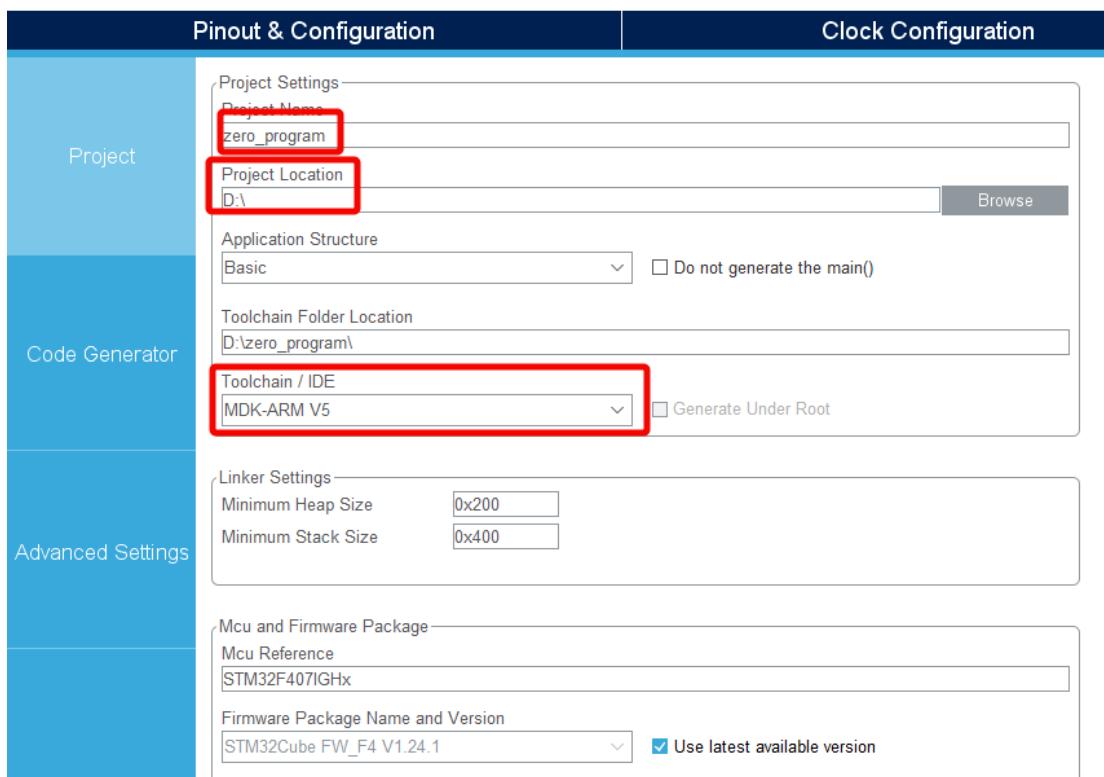
4.点击顶部的 Clock Configuration，进行主频配置；将 Input frequency 设置为 12，点击旁边的 HSE 圆形按钮，配置/M 为/6，配置*N 为 X168，配置/P 为/2，选择 PLLCLK 圆形按钮，配置 APB1 Prescaler 为/4，配置 APB2 Prescaler 为/2；



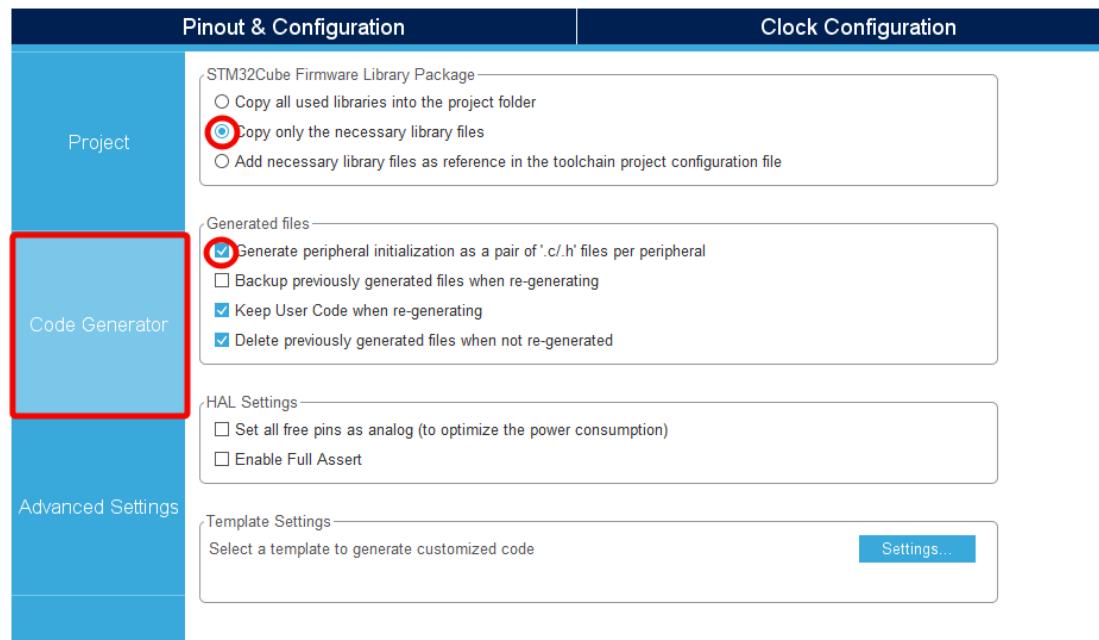
5.点击顶部的 Pinout & Configuration，选择 SYS，在 Debug 下拉框中选择 Serial Wire；



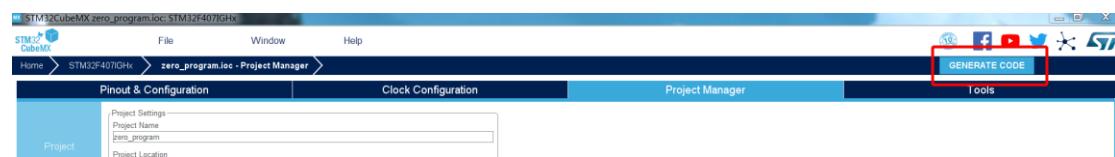
6.点击顶部的 Project Manager, 给工程起名, 选择存放目录, 在 Toolchain/IDE 中选择 MDK-ARM V5;



7.点击旁边的 Code Generator，勾选 Copy only the necessary library files 以及 Generate peripheral initialization as a pair of '.c/.h' files per peripheral;

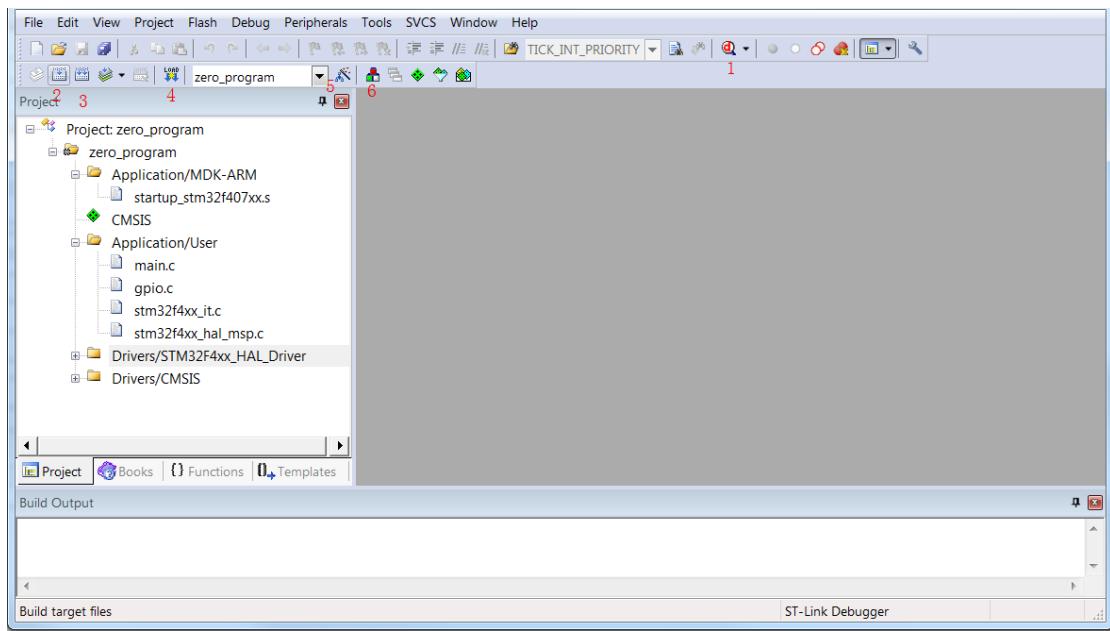


8.点击顶部的 GENERATE CODE，等待代码生成，打开工程。

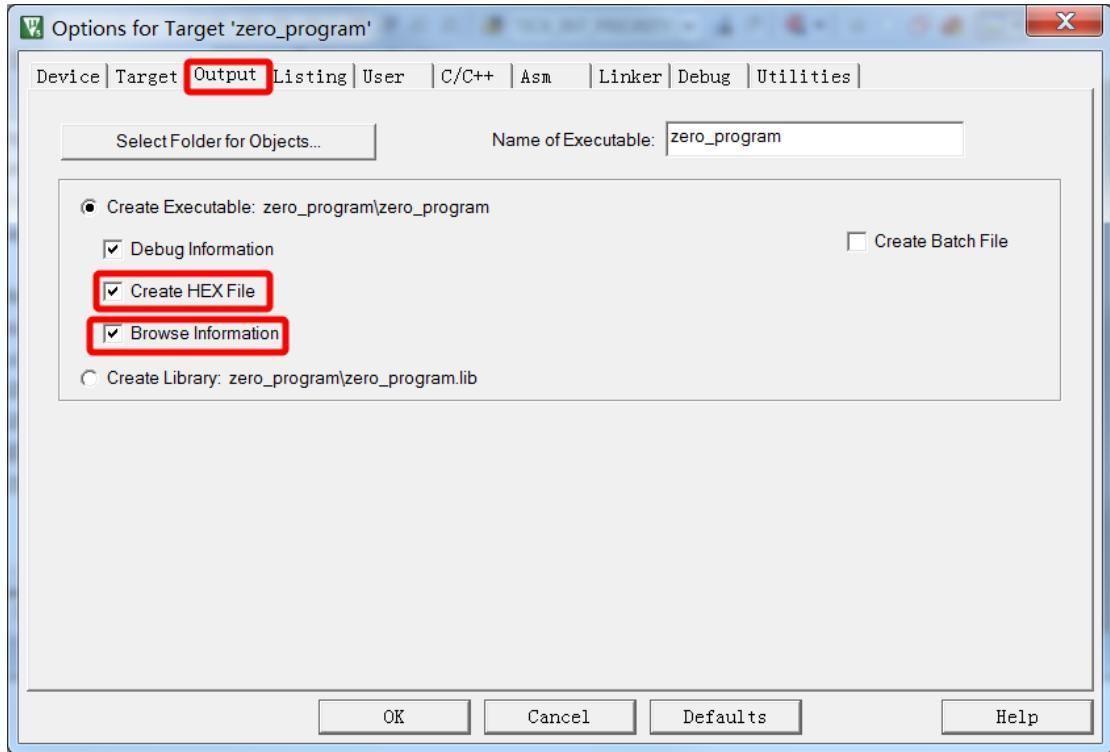


0.4.3 keil 软件简易入门

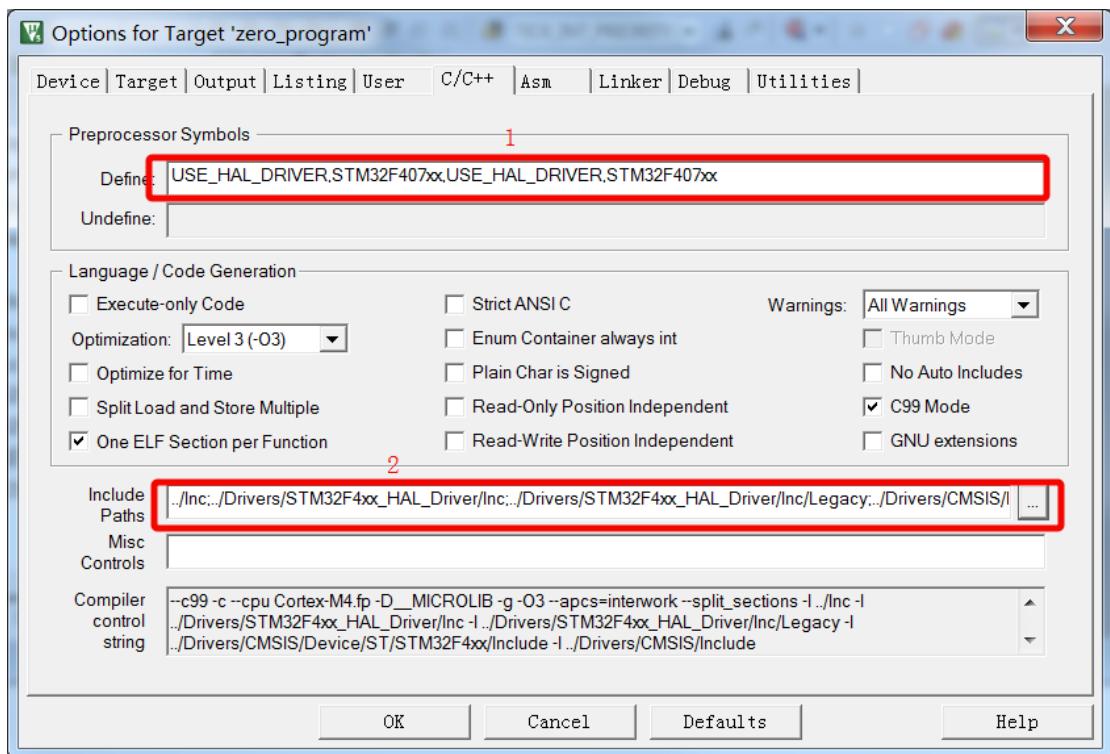
1.打开生成的工程，keil 界面如图所示，其中 1 为调试模式，2 为编译改变的文件，3 为编译全部的文件，4 为下载按键，5 为工程设置选项，6 为工程目录；



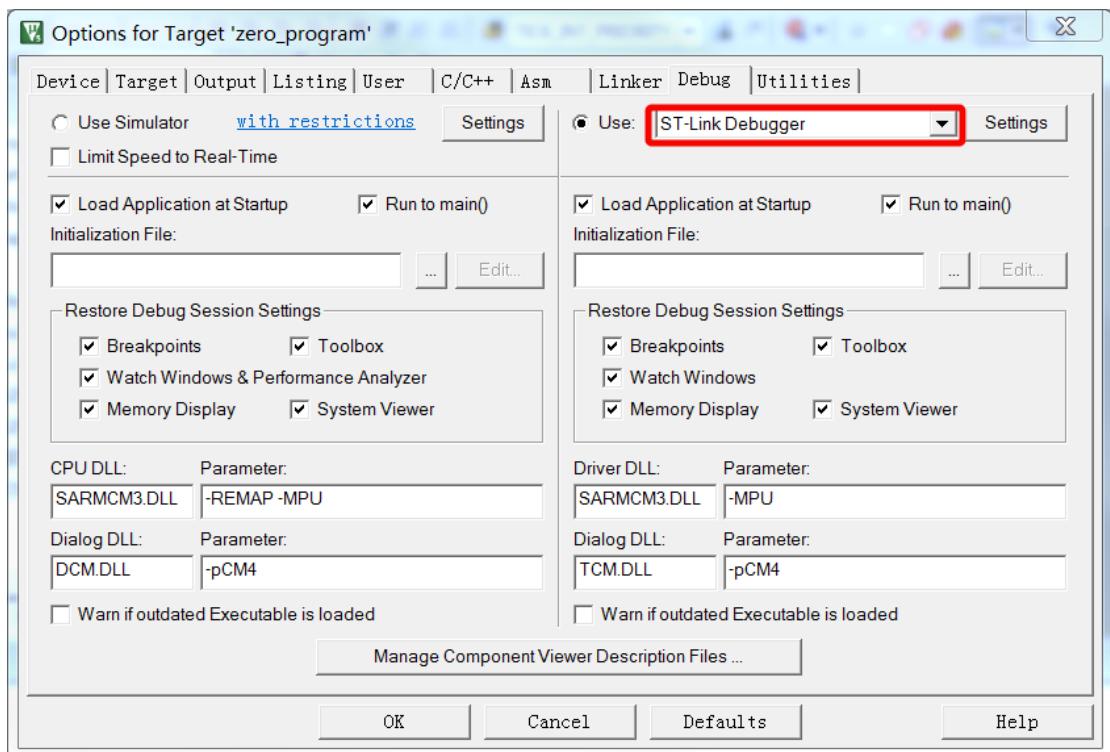
2.点击 5 工程设置，进行工程相关设置，选择 Output，其中 Create HEX file 为是否生成 HEX 文件，Browse Information 为是否增加浏览信息，选择是，可以使用鼠标右键点击函数进行跳转操作，但会增加编译时长；



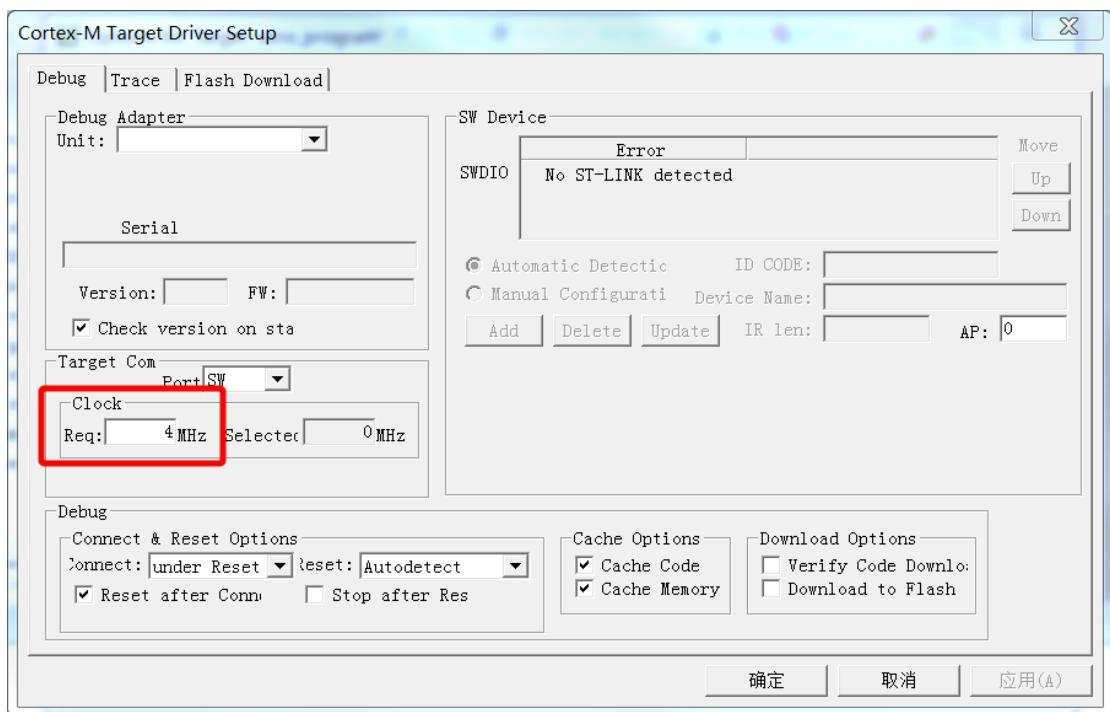
3.点击 C/C++，其中 1 方框为工程宏定义设置，可在此添加宏定义；2 方框为头文件引用目录，对于项目自行建立的 h 文件需要在此处进行目录录入；



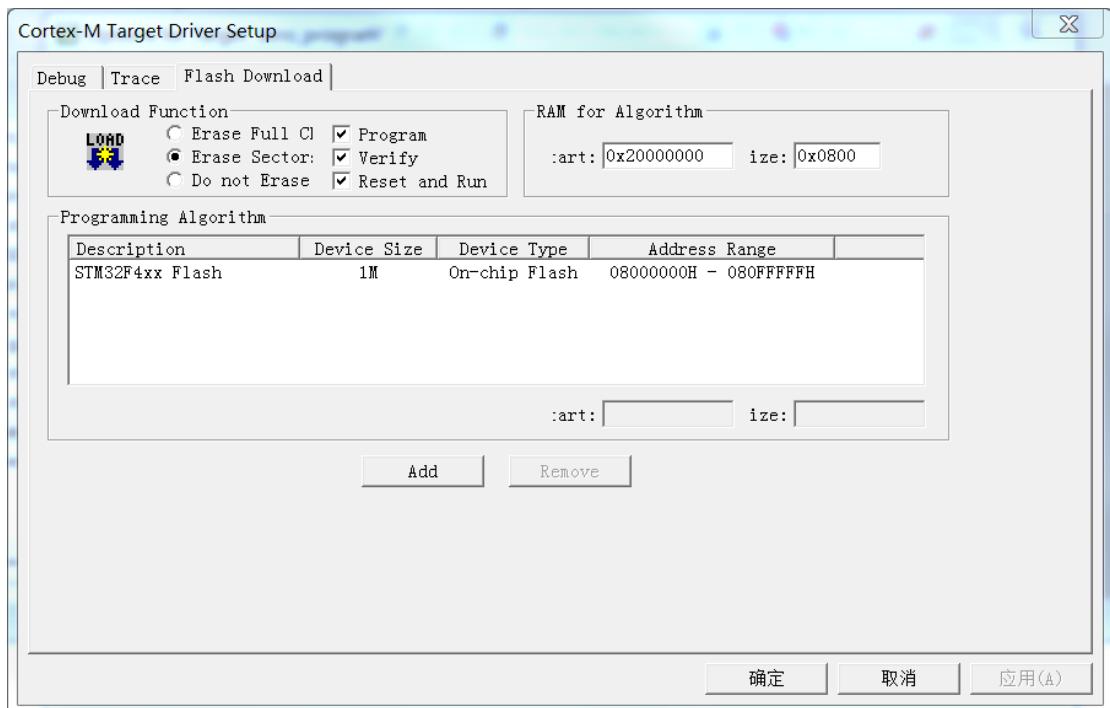
4.点击 Debug 选项，设置好对应的下载器；



5.点击下载器旁边的 Settings 选项，进行下载器相关设置，其中 Clock 中设置下载器的频率，频率越高，下载速度越快，但容易受到干扰；

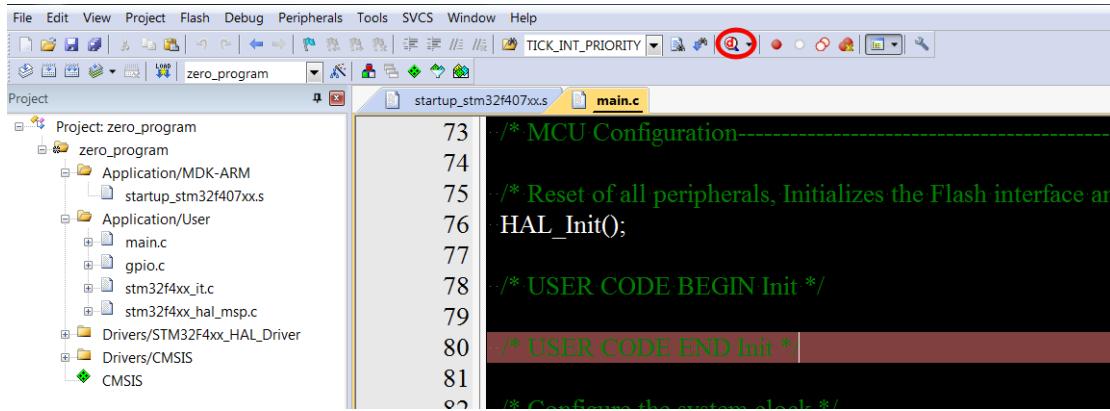


6. 点击 Flash Download, 其中 Erase Full Chip 代表下载时擦除芯片内全部页面的 flash, Erase Sectors 代表下载时擦除部分页面的 flash, Do not Erase 代表下载时不擦除 flash ; 点击 Reset and Run 代表下载完程序后立即运行程序。

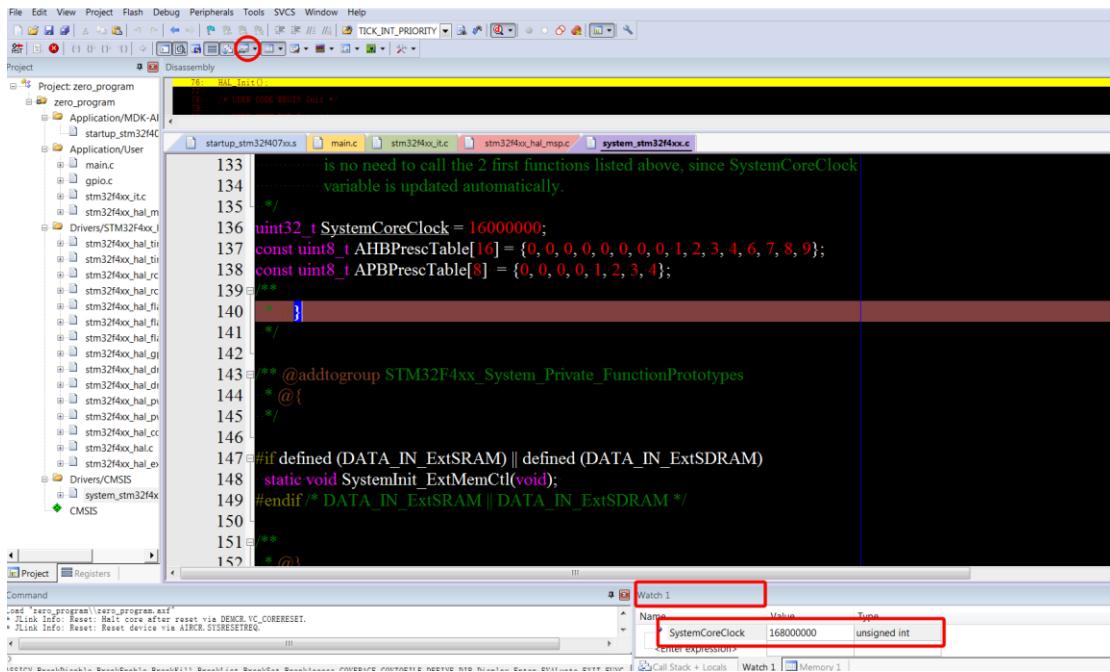


0.4.4 Keil 的调试模式

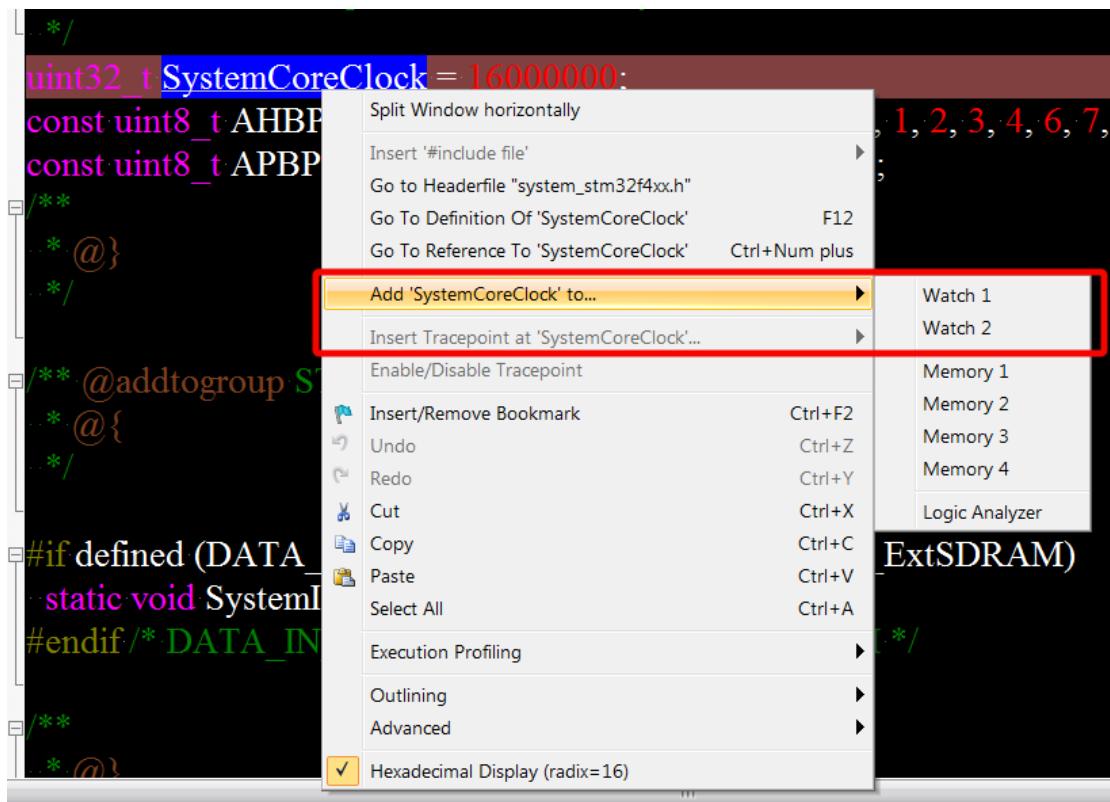
1. 在 keil 中编译之后，点击调试按键进入调试模式；



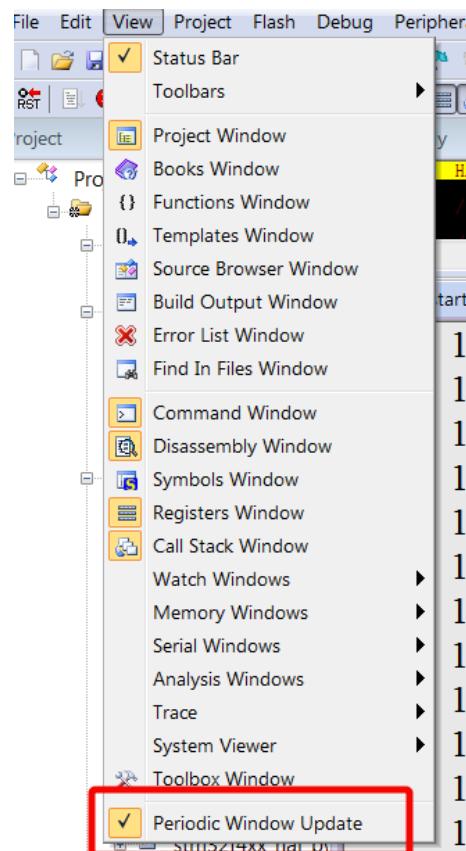
2. 进入调试模式后，可以选择 watch 窗口，watch 窗口可以参看变量的数值大小；



3. 可以在需要参看的变量点击右键，将其加入 watch 窗口；



4.如果数据在程序运行中无变化，除了数据本身没有发生变化的場合，还可能由于未开启PeriodicWindows Update。



0.5 RoboMaster 机器人功能简介

RoboMaster 机器人常见功能如下：

1. 校准功能：提供机器人的校准功能，使用到 **stm32** 的 **flash** 读写功能；
2. 底盘控制功能：完成底盘的麦轮运动控制，使用到 **stm32** 的 **CAN** 总线功能。
3. 离线判断功能：判断设备是否离线，使用到开发板 **C** 型的蜂鸣器；
4. 云台控制功能：完成云台的角度控制，使用到 **stm32** 的 **CAN** 总线功能；
5. 姿态解算功能：完成陀螺仪加速度计的角度融合，解算欧拉角，使用到 **stm32** 的 **SPI** 和 **I2C** 总线读取相关数据，使用到 **stm32** 的外部中断作为数据更新的标志，使用到 **stm32** 的 **PWM** 对加热电阻进行控制；
6. LED 的 RGB 切换：使用三色 **LED** 完成 **RGB** 显示，呼吸灯效，使用到开发板 **C** 型的 **LED** 灯；
7. OLED 显示功能：将信息显示出来，使用到 **stm32** 的 **I2C** 总线；
8. 裁判系统数据解析功能：使用单字节解析裁判系统数据，使用到 **stm32** 的串口以及 **DMA** 功能；
9. 遥控器数据解析功能：解析接收机发送的数据，使用到 **stm32** 的串口以及 **DMA** 功能；
10. 舵机控制功能：通过按键控制舵机，使用 **stm32** 的 **PWM** 功能。
11. 射击控制：控制下供弹装置，完成发射逻辑，使用到 **stm32** 的 **GPIO** 读取以及 **CAN** 总线。
12. 电源采样功能：采样电源电压，并估计当前电池电量，使用到 **stm32** 的 **ADC** 采样功能。

0.6 课程总结

本课程为基础入门课程，学习到如何使用 **cubeMX** 和 **keil** 软件进行工程创建，编译，下载，调试等功能；介绍了开发板 **C** 型出厂自带的程序功能以及 **RoboMaster** 机器人常见功能，作为学习的指导，为之后学习做铺垫。

1. 点亮 LED

1.1 知识要点

- LED 灯基本知识
- 了解三极管的通断特性
- 硬件原理图上的上拉电阻与下拉电阻
- cubeMX 中配置 GPIO 基本操作

1.2 课程内容

本课程中，将学习到如何点亮 LED 灯，了解 LED 基本原理以及简单硬件知识。通过使用 cubeMX 软件完成引脚的配置，再编写程序使得对应引脚的输出一个高电平，通过三极管通断作用，电流将通过 LED，从而发光。

1.3 基础学习

1.3.1 LED 灯基本知识

LED 即发光二极管，当 LED 内有电流通过时会发光，在安全电流范围内，电流越大，亮度越亮。LED 灯实物如图所示：



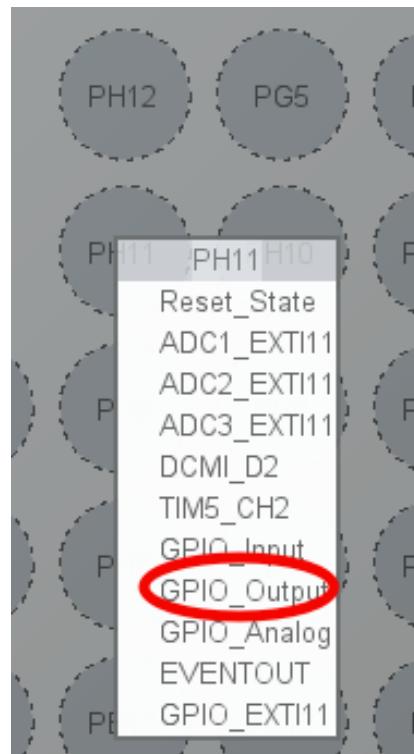
1.4 程序学习

1.4.1 cubeMX 中配置 GPIO 基本操作

- 通过原理图可以看出三个 LED 灯的引脚为 PH10, PH11, PH12, 如图所示:

DCMI_HREF	N12	PH6/I2C2_SMBA/TIM12_
DCMI_D0	M12	PH7/I2C3_SCL/ETH_MII_
LED_B	M13	PH8/I2C3_SDA/DCMI_HS_
LED_G	L13	PH9/I2C3_SMBA/TIM12_
LED_R	L12	PH10/TIM5_CH1/DCMI_L
	K12	PH11/TIM5_CH2/DCMI_L
	E12	PH12/TIM5_CH3/DCMI_L
	E13	PH13/TIM8_CH1N/CAN1
	D13	PH14/TIM8_CH2N/DCMI_L
		PH15/TIM8_CH3N/DCMI_L

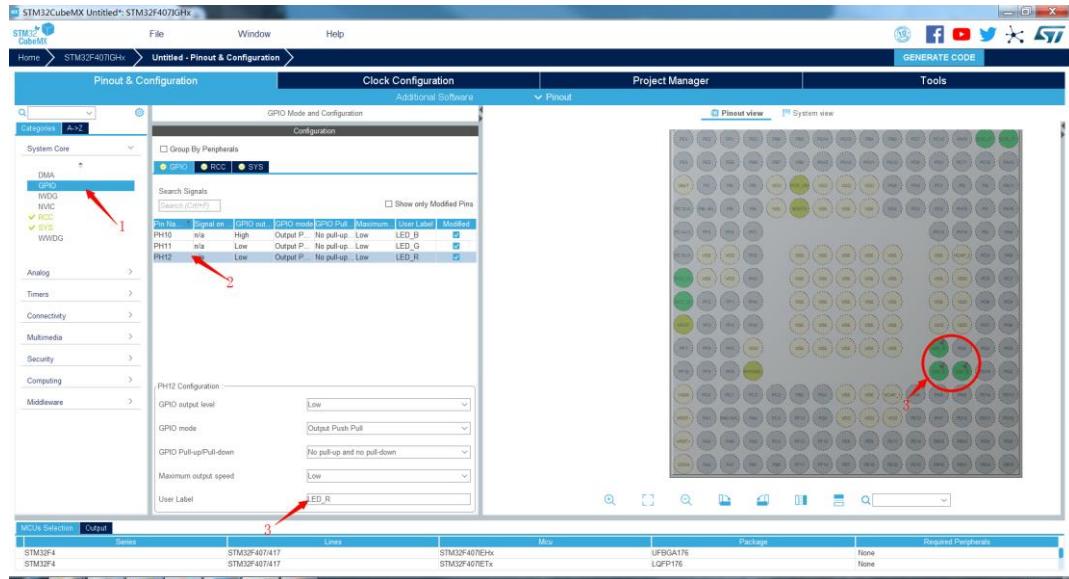
- 在 cubeMX 中配置 GPIO 为输出模式, 在 cubeMX 找到对应引脚, 配置成 GPIO_Output 模式。



- 在 cubeMX 中修改对应引脚的名字。

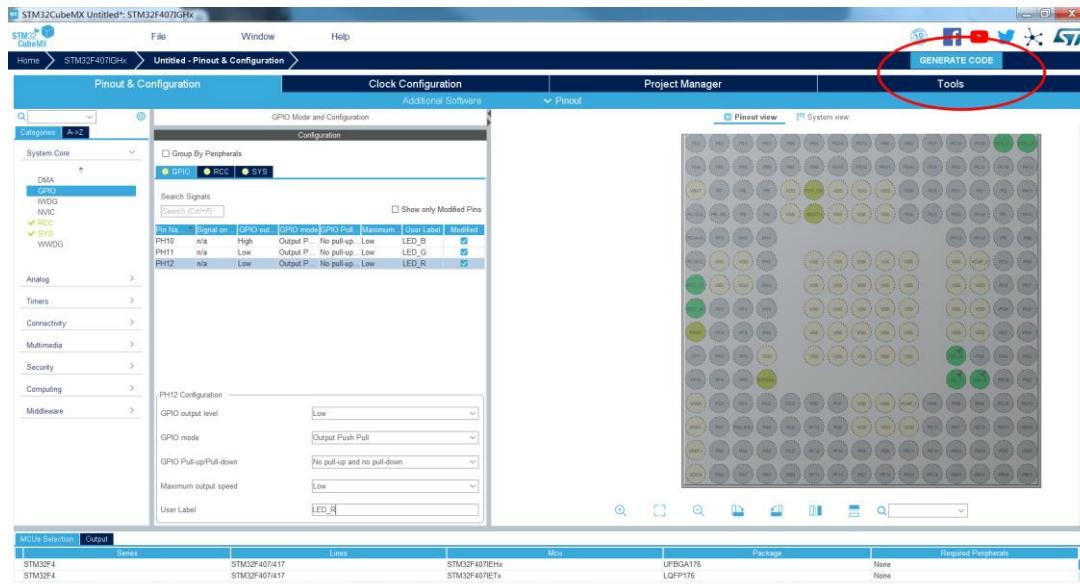
- 1) 在左侧找到 System core->GPIO;

- 2) 找到对应的 GPIO，例如 PH12；
- 3) 在下方的配置单中 user label 填写命名，填好后会在芯片缩略图中更新。



4. 生成代码

点击 GENERATE CODE 按键。



1.4.2 HAL_GPIO_WritePin 函数讲解

HAL 库中提供一个操作 GPIO 电平的函数：HAL_GPIO_WritePin 函数

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)
```

函数名	HAL_GPIO_WritePin
函数作用	使得对应的引脚输出高电平或者低电平
返回值	Void
参数 1: GPIOx	对应 GPIO 总线，其中 x 可以是 A...I。 例如 PH10，则输入 GPIOH
参数 2: GPIO_Pin	对应引脚数。可以是 0-15。 例如 PH10，则输入 GPIO_PIN_10
参数 3: PinState	GPIO_PIN_RESET: 输出低电平 GPIO_PIN_SET: 输出高电平

1.4.3 程序流程

程序经过 HAL_Init 初始化，GPIO 初始化，进入主循环，在主循环中将三个 LED 引脚均输出高电平，从而点亮 LED 灯。主循环代码如下：

```
//set GPIO output high level
//设置 GPIO 输出高电平

HAL_GPIO_WritePin(LED_R_GPIO_Port, LED_R_Pin, GPIO_PIN_SET);

HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_SET);

HAL_GPIO_WritePin(LED_B_GPIO_Port, LED_B_Pin, GPIO_PIN_SET);
```

程序流程图如图所示：



1.4.4 效果展示

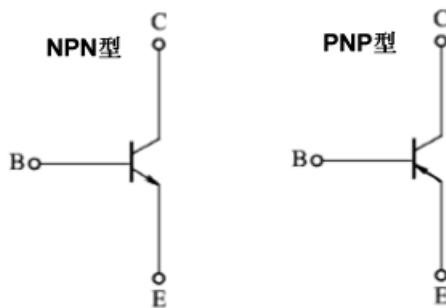
由于三色 LED 均点亮，红，绿，蓝三色为三原色，合成白光，如图所示。



1.5 进阶学习

1.5.1 三极管的通断特性

从 LED 的原理图上，看出 LED 的连线不直接连接到 stm32 引脚上，这是因为对于 stm32 来讲，引脚的输出电流能力有限，需要通过三极管来实现对 LED 灯的点亮与熄灭。常见的三极管分为 NPN 型和 PNP 型，如图所示。

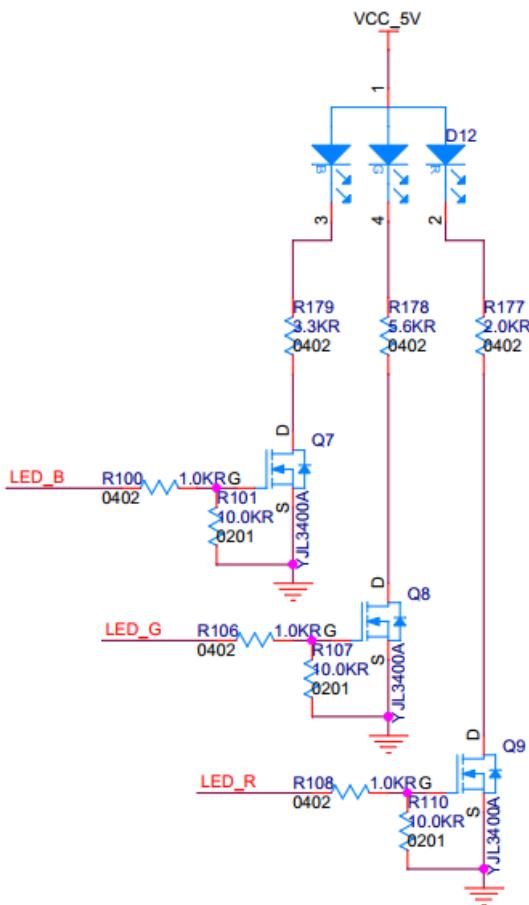


	共同点	不同点
NPN 型	<ul style="list-style-type: none">都具有三个连接点都具有电流放大作用控制方式都是通过 B 点电平控制：<ul style="list-style-type: none">➢ B 点处于高电平：CE 线导通➢ B 点处于低电平：CE 线不导通	当 B 点电压高于 E 点电压，三极管导通，电流方向为 C 点到 E 点
PNP 型		当 B 点电压低于 E 点电压，三极管导通，电流方向为 E 点到 C 点

在本设计中，三极管导通时工作在饱和状态，不导通时工作在截止状态；相当于一个开关，而 B 点相当于按下开关的手，CE 点相当于开关连接的线路。当 B 点处于高电平时，开关闭合，CE 线连通；当 B 点处于低电平的时候，开关断开，CE 线路断开。

1.5.2 LED 的下拉电阻

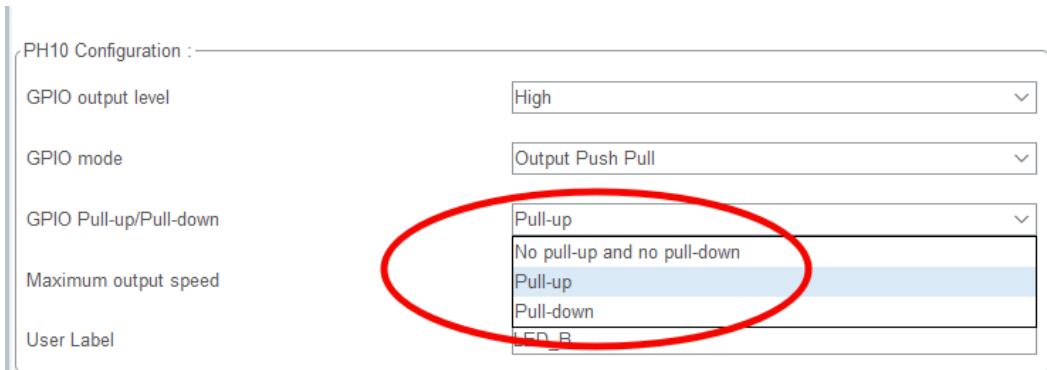
B 点电压对三级管具有控制作用，在程序未发生控制行为的时候，需要将电压控制到低电平，保证器件不被意外触发，对于三极管来说主要为下拉电阻，之外还有上拉电阻，可以从 LED 原理图找到下拉电阻。



LED

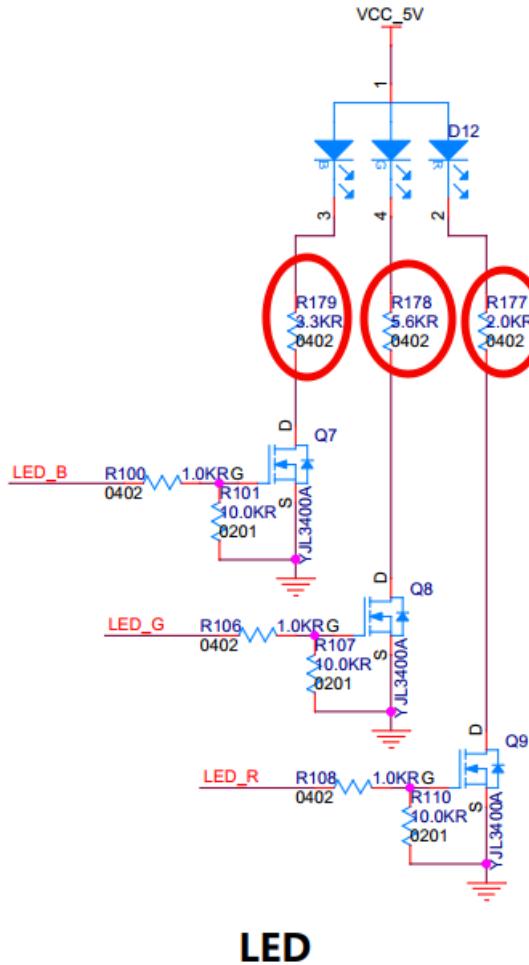
从 LED 灯原理图中，当 LED_B,LED_G 和 LED_R 三个引脚不处于高电平的输出状态，那么三极管的控制端将会被红圈圈中的下拉电阻拉为低电平，当 LED_B,LED_G 和 LED_R 处于高电平的输出状态，通过原理图中 $1\text{k}\Omega$ 与 $10\text{k}\Omega$ 的分压后，三极管的基极控制端将变成高电平，故而三级管的控制端电压将变成高电平。

在 cubeMX 中配置 stm32 的引脚，可以选择配置上拉电阻或者下拉电阻，如图所示。Pull-up 是配置引脚为上拉电阻，Pull-down 是配置引脚为下拉电阻，No pull-up and no pull-down 是配置引脚不上拉也不下拉。



1.5.3 硬件原理图上的限流电阻

LED 灯需要在一个合适的电流范围内工作，所以需要一个限流电阻来限制其电流，如下图的 R177、R178、R179。限流电阻保证电路中通过 LED 的电流不超过其额定电流。此外，还可以用于调节 LED 的亮度。



如上图所示，在 LED 的原理图上，存在三个限流电阻。其中，绿灯的限流电阻较大，蓝灯的限流电阻较小，由于人眼对于不同波长的光敏感程度不同，即人对于相同光强的不同波长的光感受到亮度不同，绿光最为敏感，绿灯需要通过较小的电流，红灯需要通过正常的电流，蓝灯需要通过较大的电流，保证人眼看到三个 LED 灯亮度相同。

1.6 课程总结

GPIO 输出操作是 stm32 中最基础的操作，通过高低电平控制完成一次类似开关的控制，而高电平对应了计算机中的数字 1，而低电平对应了计算机中的数字 0，高低电平的不断变化，对应计算机内部的数字变化，stm32 便是这样输出信号给外部世界。之外本节课还了解 LED

灯，三极管，电阻的使用，这些是机器人中常用的器件，LED 通过灯效形式呈现，给参赛队员提醒。

2. 闪烁 LED

2.1 知识要点

- 三种延时方法
- HAL_Init 作用
- 滴答计时器

2.2 课程内容

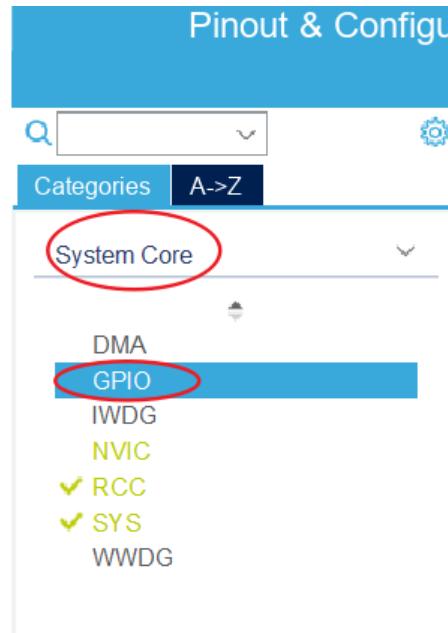
本课程中,将学习如何控制 stm32 的电平变化,学习 stm32 中的三种延时方法,了解 Hal_Init 函数的作用,了解滴答定时器的原理及作用。使用 cubeMX 软件完成引脚的配置,然后编写程序,通过引脚电平反转翻转函数 HAL_GPIO_TogglePin 和延时函数,实现 LED 灯的闪烁效果。

2.3 基础学习

2.3.1 GPIO 的翻转速度

本小节中,将学习如何通过 cubeMX 配置 GPIO 的翻转速度。在不同情况下,需要 GPIO 引脚会输出不同频率的方波。例如在 led 闪烁时,只需要 1Hz 的频率就可以看到明显的闪烁效果,但在软件模拟 I2C 时,需要数百 KHz 频率才能完成正常的通信。在 cubeMX 中可以根据不同的输出频率需求,设置 GPIO 不同的翻转速度。

1. 采用上一讲中介绍的方法开启 PH10,PH11,PH12 引脚的输出功能。
2. 在 cubeMX 的左侧边栏中的 System Core 下有 GPIO 选项,在该选项下可以看到已经开启的引脚的配置信息,如图所示:



3. 在这个标签页下，可以选中需要配置的 GPIO，并查看其详细状态，其中 Maximum output speed 就是可以选择的翻转速度模式，如图所示，对应的翻转模式是 Low，为低速输出模式。

The screenshot shows the 'Configuration' tab selected in the top navigation bar. Under 'Group By Peripherals', 'GPIO' is selected. In the 'Search Signals' section, 'PH10' is entered. The main table lists pins PH10, PH11, and PH12, each with its signal state, GPIO output level, mode, pull-up/pull-down settings, and user label. The 'Modified' column for PH10 is checked. Below the table, a detailed configuration panel for 'PH10 Configuration' is shown, with fields for GPIO output level (High), GPIO mode (Output Push Pull), GPIO Pull-up/Pull-down (Pull-up), and Maximum output speed (Low).

Pin N...	Signal on	GPIO out...	GPIO mode	GPIO Pull...	Maximum	User Label	Modified
PH10	n/a	High	Output Pu...	Pull-up	Low	LED_B	<input checked="" type="checkbox"/>
PH11	n/a	High	Output Pu...	Pull-up	Low	LED_G	<input checked="" type="checkbox"/>
PH12	n/a	High	Output Pu...	Pull-up	Low	LED_R	<input checked="" type="checkbox"/>

PH10 Configuration :

GPIO output level	High
GPIO mode	Output Push Pull
GPIO Pull-up/Pull-down	Pull-up
Maximum output speed	Low
User Label	LED_B

点击右端小箭头，可选的输出速度分为 Low, Medium, High, Very High 四档，如图所示。一般使用 GPIO 输出驱动 LED 等功能时选择 Low 档翻转速度即可，而一般用于通信的 GPIO 需要设置为 High 或者 Very High，具体设置可以根据相关通信协议对 GPIO 的翻转速度的要求进行设置。



2.4 程序学习

2.4.1 计数延时介绍

在 stm32 内执行任何一条指令是需要消耗时间的。实现延时效果可以让 stm32 持续执行一段计数循环，直到其计数到设定的数目后，再让 stm32 退出循环。

在程序中，通过 user_delay_us 函数完成微秒级的延时功能，函数的实现如下所示：

```
void user_delay_us(uint16_t us)
{
    for(; us > 0; us--)
    {
        for(uint8_t i = 50; i > 0; i--)
        {
            ;
        }
    }
}
```

将内层循环变量 i 初始化为 50，外层循环变量的赋值则由函数的入口参数来决定，入口参数对应了外层循环的次数，继而相应的实现了对应时长的延时。

类似于微秒级延迟的实现，再加一层外部循环便可以实现毫秒级的延迟，对应函数为 user_delay_ms，其具体实现如下所示。该函数调用了 user_delay_us 函数，并给其参数赋值 1000，1 毫秒等于 1000 微秒。通过循环调用的方式，将延时时间折算到毫秒级。

```
void user_delay_ms(uint16_t ms)
{
    for(; ms > 0; ms--)
    {
        user_delay_us(1000);
    }
}
```

2.4.2 nop 延时介绍

使用 `nop` 函数是第二种延时方法，其原理与计时延时类似。同样是通过重复执行指令，直到消耗时间，再进行下一步的工作。但与计数延时不同的是，`nop` 延时通过空操作指令`__nop()`函数来实现延时。当 `stm32` 执行到`__nop()`时，可以理解为在当前指令周期中，`stm32` 没有进行任何工作。

同样是通过循环的方式，可以完成微秒级的延迟 `nop_delay_us` 和 `nop_delay_ms`

```
void nop_delay_us(uint16_t us)
{
    for(; us > 0; us--)
    {
        for(uint8_t i = 10; i > 0; i--)
        {
            __nop();
            __nop();
        }
    }
}
```

```
    __nop();
    __nop();
    __nop();
    __nop();
    __nop();
}

}

}

void nop_delay_ms(uint16_t ms)
{
    for(; ms > 0; ms--)
    {
        nop_delay_us(1000);
    }
}
```

2.4.3 滴答计时器介绍以及 HAL_Init 初始化

在介绍第三种延时方法之前，先介绍滴答定时器及其初始化。

滴答定时器也称为 **SysTick**，是 **stm32** 内置的倒计时定时器，每当计数到 0 时，触发一次 **SysTick** 中断，并重载寄存器值。滴答计时器的初始化在 **HAL_Init** 函数中完成，配置成 1ms 的中断。**HAL_Init** 的内容如下，通过注释了解到：在 **HAL_Init** 中，实现了 **SysTick** 以及底层硬件的初始化。

```
HAL_StatusTypeDef HAL_Init(void)

{
    /* Configure Flash prefetch, Instruction cache, Data cache */

#if (INSTRUCTION_CACHE_ENABLE != 0U)
    __HAL_FLASH_INSTRUCTION_CACHE_ENABLE();
#endif /* INSTRUCTION_CACHE_ENABLE */

#if (DATA_CACHE_ENABLE != 0U)
```

```

__HAL_FLASH_DATA_CACHE_ENABLE();

#endif /* DATA_CACHE_ENABLE */

#if(PREFETCH_ENABLE != 0U)

__HAL_FLASH_PREFETCH_BUFFER_ENABLE();

#endif /* PREFETCH_ENABLE */

/* Set Interrupt Group Priority */

HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

/* Use systick as time base source and configure 1ms tick (default clock after Reset is HSI) */

HAL_InitTick(TICK_INT_PRIORITY);

/* Init the low level hardware */

HAL_MspInit();

/* Return function status */

return HAL_OK;
}

```

在 `HAL_Init` 的实现中，关注滴答定时器初始化函数。

```
__weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
```

`HAL_InitTick` 函数将设定 `SysTick` 的定时周期为 1ms，即频率为 1000Hz，并使 `SysTick` 开始工作。

每当滴答计时器递减到 0 时，会触发中断，使程序进入 `SysTick` 中断处理函数

```

void SysTick_Handler(void)
{
    /* USER CODE BEGIN SysTick_IRQn 0 */

    /* USER CODE END SysTick_IRQn 0 */

    HAL_IncTick();

    /* USER CODE BEGIN SysTick_IRQn 1 */
}

```

```
/* USER CODE END SysTick_IRQHandler 1 */  
}
```

在其中会调用 `HAL_IncTick` 函数，该函数实现如下

```
__weak void HAL_IncTick(void)  
{  
    uwTick += uwTickFreq;  
}
```

在 `uwTick` 变量中存储的是从 stm32 的 Systick 初始化以来所经过的时间 (ms)，`uwTick` 的存在相当于给整个程序提供了一个绝对的时间基准，而 `HAL_Delay` 函数延时功能便是通过 `uwTick` 的值完成的。

获取当前的 `uwTick` 值可以使用 HAL 库提供的函数 `HAL_GetTick`

<code>uint32_t HAL_GetTick(void)</code>	
函数名	<code>HAL_GetTick</code>
函数作用	返回当前时刻的 <code>uwTick</code> 值
返回值	当前时刻的 <code>uwTick</code> 值

2.4.4 HAL_Delay 介绍

HAL 库提供了用于毫秒级延迟的函数，`HAL_Delay` 函数（使用 `_weak` 修饰符说明该函数是可以用户重定义的）。

<code>__weak void HAL_Delay(uint32_t Delay)</code>	
函数名	<code>HAL_Delay</code>
函数作用	使系统延迟对应的毫秒级时间
返回值	<code>void</code>
参数	<code>Delay</code> ，对应的延迟毫秒数，比如延迟 1 秒就为 1000

在 **HAL_Delay** 的实现中，调用 **HAL_GetTick** 函数获取基准时间 **uwTick**，说明了 **HAL_Delay** 函数的实现是基于滴答计时器（**Systick**）。

```
__weak void HAL_Delay(uint32_t Delay)

{
    uint32_t tickstart = HAL_GetTick();
    uint32_t wait = Delay;

    /* Add a freq to guarantee minimum wait */

    if (wait < HAL_MAX_DELAY)
    {
        wait += (uint32_t)(uwTickFreq);
    }

    while((HAL_GetTick() - tickstart) < wait)
    {
    }
}
```

以上三种延时方法各自有各自的特点，计数延时与 **nop** 延时都需要用户编写函数来自行实现，比较麻烦，而直接调用 **HAL** 库提供的 **HAL_Delay** 函数会更加方便一些。但 **HAL_Delay** 只能够实现毫秒级的延时，如果需要时间更短的延时函数则必须使用用户编写的延时函数。

2.4.5 HAL_GPIO_TogglePin 介绍

本讲实验所使用的 **cubeMX** 配置与上一讲相同。

在上一讲中，介绍了操作 **GPIO** 电平的函数：**HAL_GPIO_WritePin**，通过这个函数可以设置 **GPIO** 输出高电平或者低电平。为了实现 **LED** 灯的闪烁，只要交替输出高低电平即可，因此可以通过两句 **HAL_GPIO_WritePin**，各自分别将引脚设置为高电平和低电平，就可以实现闪烁功能。

但是除了以上方法之外，还有更简便的函数来实现引脚电平翻转--**HAL** 库提供的 **HAL_GPIO_TogglePin** 函数。

```
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

函数名	HAL_GPIO_TogglePin
函数作用	翻转对应引脚的电平
返回值	Void
参数 1: GPIOx	对应 GPIO 总线，其中 x 可以是 A...I。 例如 PH10，则输入 GPIOH
参数 2: GPIO_Pin	对应引脚数。可以是 0-15。 例如 PH10，则输入 GPIO_PIN_10

2.4.6 程序流程

程序经过 HAL_Init 初始化, SystemClock 初始化, GPIO 初始化后, 进入主循环, 在主循环内分别使用之前介绍过的三种方式进行延时, 然后将 LED 的电平翻转, 从而使 LED 按照 500 毫秒的固定频率进行亮灭, 主循环代码如下。

```

bsp_led_toggle();

//nop delay
nop_delay_ms(500);

bsp_led_toggle();

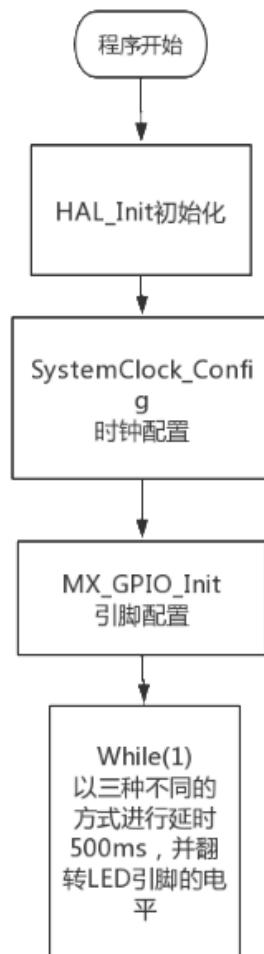
//cycl count delay
user_delay_ms(500);

bsp_led_toggle();

//systick delay
HAL_Delay(500);

```

程序流程图如下：



2.4.7 效果展示

开发板 C 型上的 LED 灯会以约 500ms 跳变一次，呈现闪烁的效果。如图所示，分别为开发板 C 型 LED 点亮和熄灭效果。



LED 点亮图



LED 熄灭图

2.5 课程总结

延时操作也是 `stm32` 中的基本操作，本节课介绍的三种延时方式各有特点，调用 `HAL` 库提供的 `HAL_Delay` 函数最为方便，但是无法完成微秒级的延时，可以通过自己定义的 `nop` 延时或者计数延时的方法来进行实现。

3. 定时器闪烁 LED

3.1 知识要点

- 定时器基本功能
- 定时器的配置
- 定时器中断以及中断优先级讲解
- cubeMX 中的中断管理

3.2 课程内容

本节课将了解定时器的基本功能及其配置方法，还接触 **stm32** 中最重要的概念之一——中断，介绍在 **cubeMX** 中如何对中断进行设置，如何开启中断以及配置中断的优先级等，最后将实现由定时器触发的定时器中断，控制 **LED** 灯的闪烁。

3.3 基础学习

3.3.1 定时器讲解

定时器的基本功能是计时功能，如同闹铃一般，设定好对应的时间后，会在设定的时刻响起铃声。例如上章的滴答定时器，设定为 **1ms** 的定时时间，便每隔 **1ms** 引起中断函数。

在使用定时器时，会涉及到三个非常重要的概念——分频，计数，重载。这三个概念可以结合生活中使用的时钟来理解。

- 分频：时钟上不同的指针需要有不同的速度，也就是不同的频率，从而精确的表示时间，比如秒针，分针，时针，这三者相邻的频率之比都是 **60:1**，即秒针每转过 60 格分针转动 1 格，分针转动 60 格时针转动 1 格，所以分针对于秒针的分频为 60。
- 计数：时钟所对应的值都是与工作时间成正比的，比如秒针转动 10 格，意味着过了 10 秒，同样定时器中的计数也是和计数时间成正比的值，频率越高增长速度越快。
- 重载：时、分、秒的刻度都是有上限的，一个表盘最多记 12 小时，60 分钟，60 秒，如果继续增加的话就会回到 0。同样的在定时器中也需要重载，当定时器中的计数值达到重载值时，计数值就会被清零。

现在介绍与定时器有关的三个重要寄存器。

- 预分频寄存器 **TIMx_PSC**

- 计数器寄存器 **TIMx_CNT**
- 自动重装载寄存器 **TIMx_ARR**

时钟源处的时钟信号经过预分频寄存器，按照预分频寄存器内部的值进行分频。比如时钟源的频率为 **16MHz**，而预分频寄存器中设置的值为 **16: 1**，那么通过预分频后进入定时器的时钟频率就下降到了 **1MHz**。

在已经分频后的定时器时钟驱使下，**TIMx_CNT** 根据该时钟的频率向上计数，直到 **TIMx_CNT** 的值增长到与设定的自动重装载寄存器 **TIMx_ARR** 相等时，**TIMx_CNT** 被清空，并重新从 0 开始向上计数，**TIMx_CNT** 增长到 **TIMx_ARR** 中的值后被清空时产生一个定时中断触发信号。综上定时器触发中断的时间是由设定的 **TIMx_PSC** 中的分频比和 **TIMx_ARR** 中的自动重装载值共同决定的。

定时器是 **stm32** 中非常重要的外设。在大多数应用场景中，部分任务需要周期性的执行，比如上一讲中提到的 **LED** 闪烁，这个功能就可以依靠定时器来实现，此外 **stm32** 的定时器还能够提供 **PWM** 输出，输入捕获，输出比较等多种功能。

3.3.2 中断讲解

RM 赛场上的机器人往往需要处理多种信号的输入，比如遥控器信号，视觉 **PC** 发送的信号，各种传感器信号等等，此外还需要进行多种信号的输出，比如控制电机的 **CAN** 信号，控制舵机的 **PWM** 等等，那么 **STM32** 是如何有序安排这些任务的呢？就是依赖于中断构成的前后台机制。

在 **STM32** 中，对信号的处理可以分为轮询方式和中断方式，轮询方式就是不断去访问一个信号的端口，看看有没有信号进入，有则进行处理，中断方式则是当输入产生的时候，产生一个触发信号告诉 **STM32** 有输入信号进入，需要进行处理。

例如厨房里烧着开水，主人在客厅里看电视。为了防止开水烧干，他有两种方式，第一种是每隔 **10** 分钟就去厨房看一眼，另一种是等水壶烧开了之后开始发出响声再去处理。前者是轮询的方式，后者是中断的方式。

每一种中断都有对应的中断函数，当中断发生时，程序会自动跳转到处理函数处运行，而不需要人为进行调用。

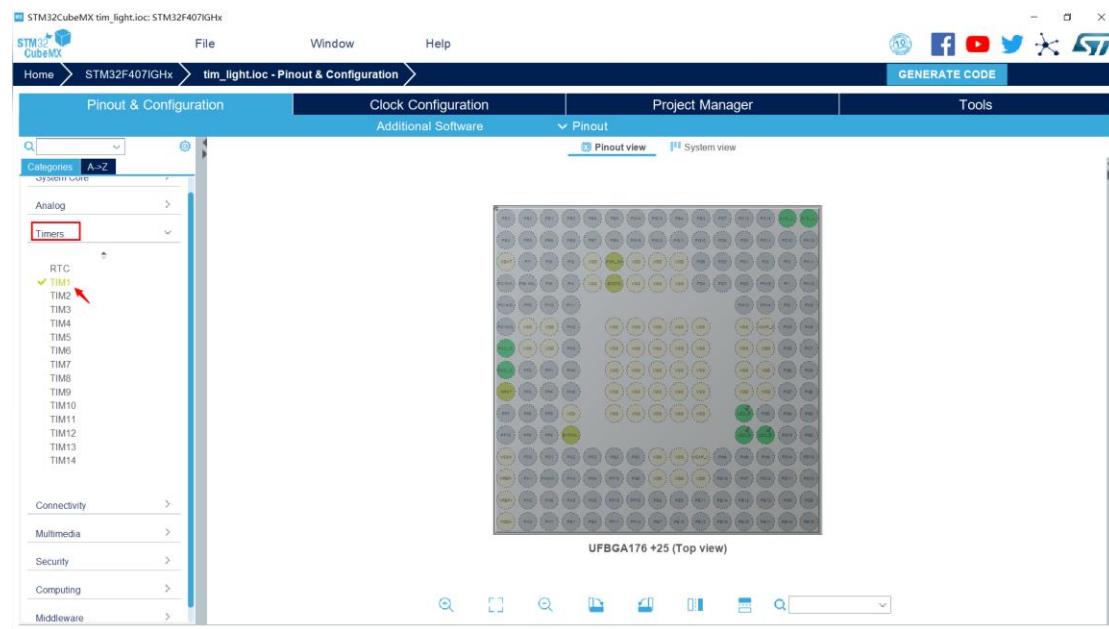
如第一节中，当定时器的计数值增长到重载值时，在清空计数值的同时，会触发一次定时器中断，即定时器更新中断。只要设定好定时器的重载值，就可以保证定时器中断以固定的速度被触发。在 **RM** 比赛中，可以将底盘控制任务，云台控制任务这类需要定时执行的任务放入定时器更新中断中执行。

3.4 程序学习

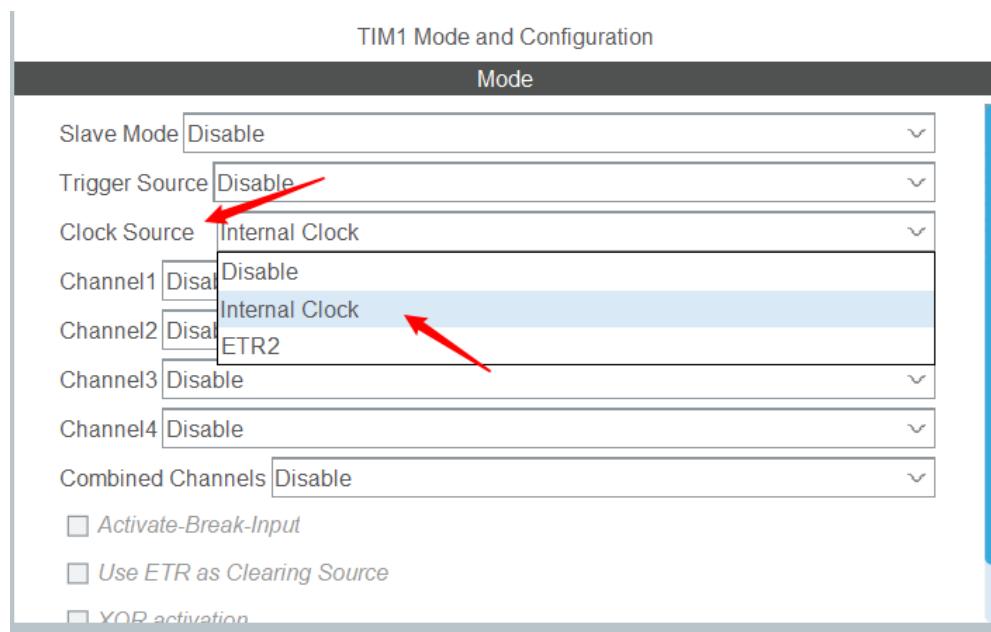
3.4.1 定时器在 cubeMX 中配置

本章驱动 LED 的 GPIO 配置与前两章相同，此外还需要在 cubeMX 中设置定时器。操作如下：

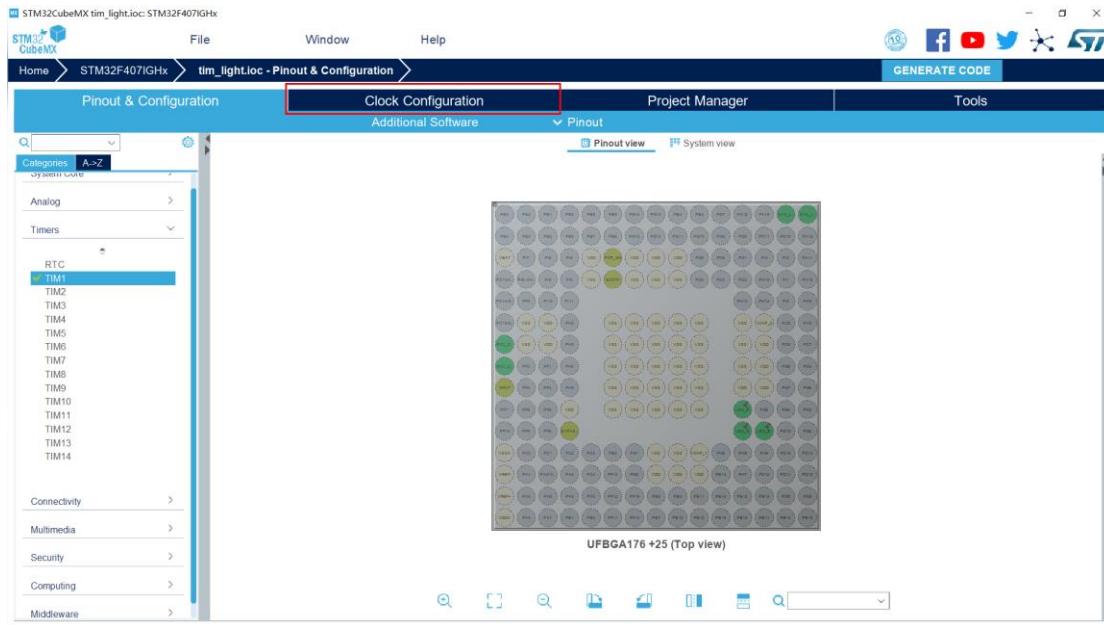
1. 在左侧的标签页中选择 Timer，点击标签页下的 TIM1。



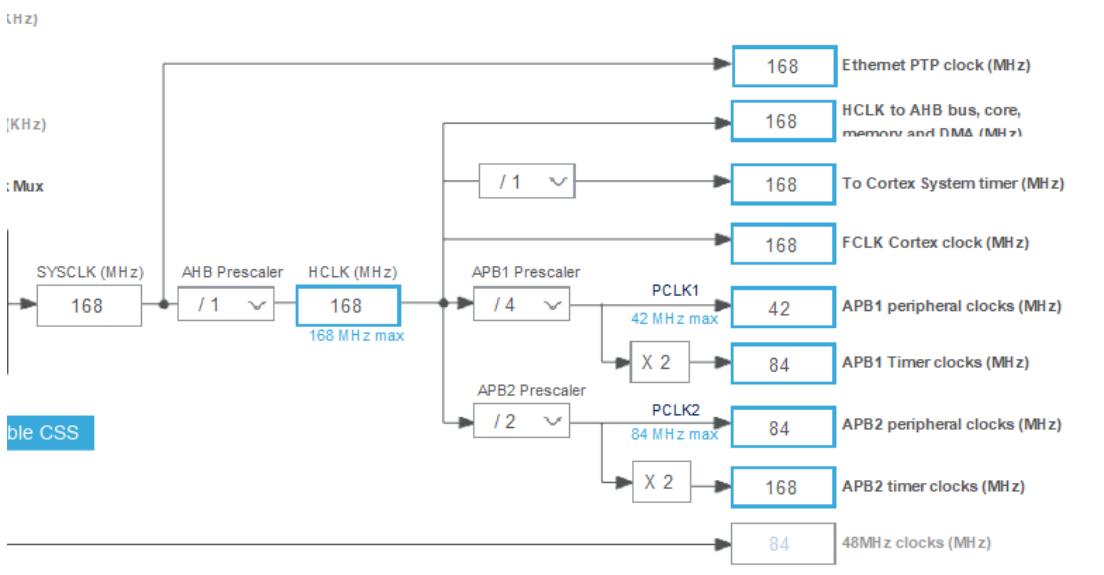
2. 在弹出的 TIM1 Mode and Configuration 中，在 ClockSouce 的右侧下拉菜单中选中 Internal Clock。



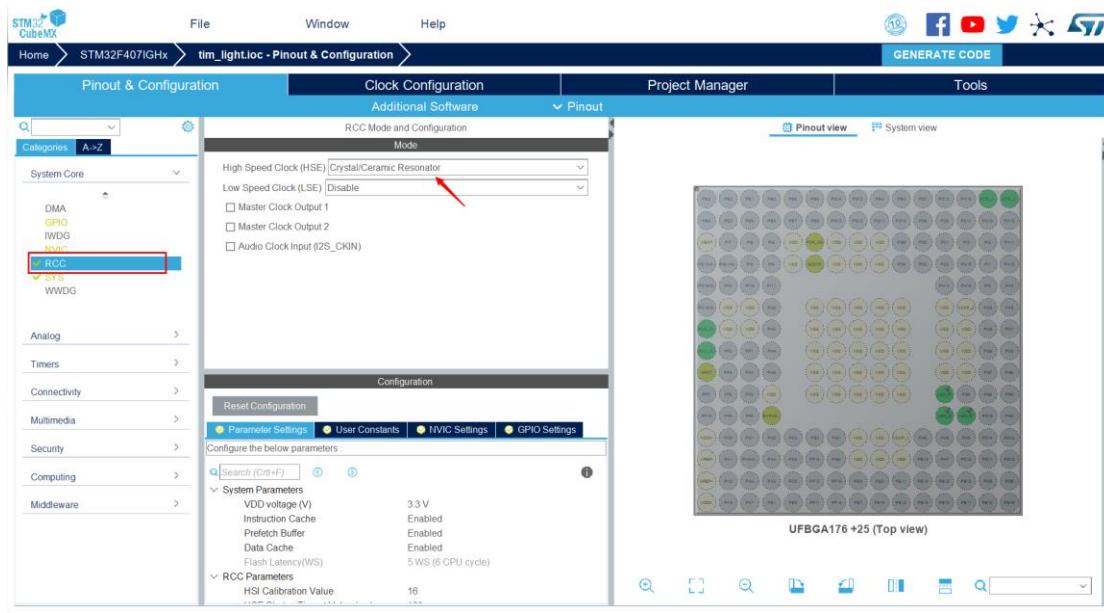
3. 此时 TIM1 得到使能，接下来需要配置 TIM1 的运转周期。需要打开 Clock Configuration 标签页。



下图为 STM32 的时钟树结构，通过配置这个结构中各处的分频/倍频比，可以控制最后输出到各个外设的时钟。



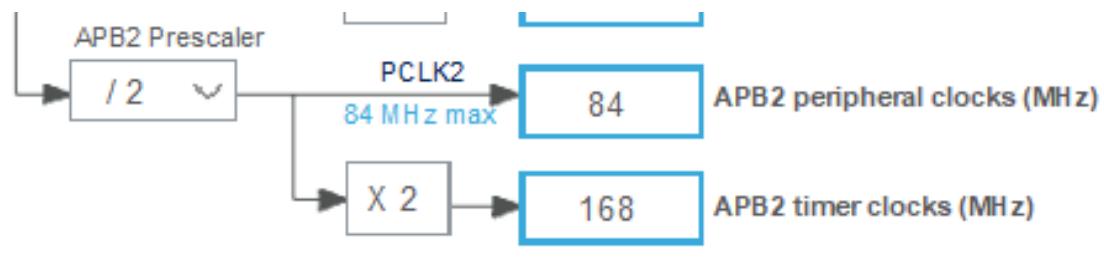
如果 HSE 处为灰色，则先去 Pinout&Configuration 标签页下确定是否使能外部晶振。



通过查阅数据手册资料，可以知道 TIM1 的时钟源来自 APB2 总线，此外也可以在程序中的 `_HAL_RCC_TIM1_CLK_ENABLE` 定义里面找到，可以看到代码中的 APB2，说明 TIM1 挂载在 APB2 上。

```
#define __HAL_RCC_TIM1_CLK_ENABLE()      do { \
                                         \
                                         __IO uint32_t tmpreg = 0x00U; \
                                         \
                                         SET_BIT(RCC->APB2ENR, RCC_APB2ENR_TIM1EN); \
                                         \
                                         /* Delay after an RCC peripheral clock enabling */ \
                                         \
                                         tmpreg = READ_BIT(RCC->APB2ENR, \
                                         \
                                         RCC_APB2ENR_TIM1EN); \
                                         \
                                         UNUSED(tmpreg); \
                                         \
} while(0U)
```

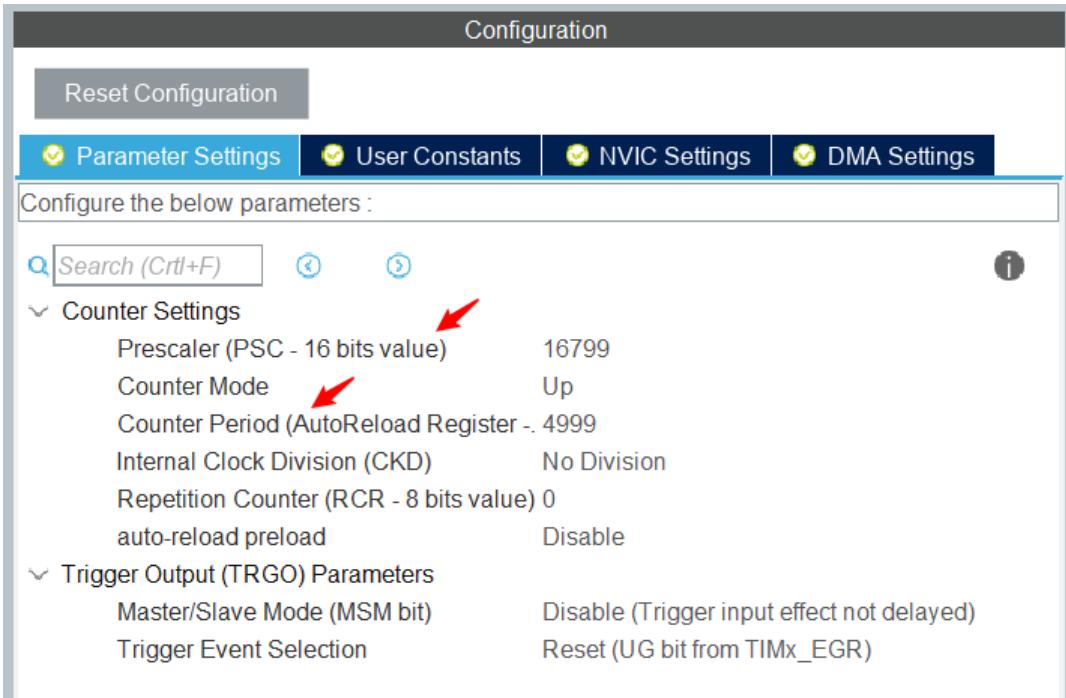
注意到在时钟树配置页面下的 APB2 Timer clocks (MHz) 为 168MHz，这意味着提供给 TIM1 预分频寄存器的频率就是 168MHz。



下面通过设置分频比和重载值来控制定时器的周期，详细的计算步骤可以查看进阶学习的部分。

如果想要得到周期为 500 毫秒的定时器，则可以按照进阶学习介绍的公式来对分频值和重载值进行设定。回到 Pinout&Configuration 标签页下，对应 TIMx_PSC 寄存器的 Prescaler 项和对应 TIMx_ARR 寄存器的 Counter Period 项。500ms 对应的频率为 2Hz，为了得到 2Hz 的频率，可以将分频值设为 16799，重载值设为 4999，则可以计算出定时器触发频率为

$$f = \frac{168000000Hz}{(16799 + 1) * (4999 + 1)} = 2Hz$$



3.4.2 中断优先级讲解

回顾之前所说的中断概念，在 STM32 专门用于处理中断的控制器叫做 NVIC，即嵌套向量中断控制器（Nested Vectored Interrupt Controller）。

NVIC 的功能非常强大，支持中断优先级和中断嵌套的功能，中断优先级即给不同的中断划分不同的响应等级，如果多个中断同时产生，则 STM32 优先处理高优先级的中断。

中断嵌套即允许在处理中断时，如果有更高优先级的中断产生，则挂起当前中断，先去处理产生的高优先级中断，处理完后再恢复到原来的中断继续处理。

这个过程理解起来就像是在上文的情境中，主人听到水烧开了，正打算去厨房时突然听到门口响起了急促的敲门声，那么主人就会先去执行开门的操作，然后再去厨房处理开水。

为了在有限的寄存器位数中实现更加丰富的中断优先级，NVIC 使用了中断分组机制。

STM32 将先将中断进行分组，然后又将优先级划分为抢占优先级（Prem priority）和响应优

先级 (Subpriority)，抢占优先级和响应优先级的数量均可以通过 NVIC 中 AIRCR 寄存器的 PRIGROUP[8:10]位进行配置，从而规定了两种优先级对 NVIC_IPRx[7:4]的划分，根据划分决定两种优先级的数量。总共可以分成下表中的 5 种情况

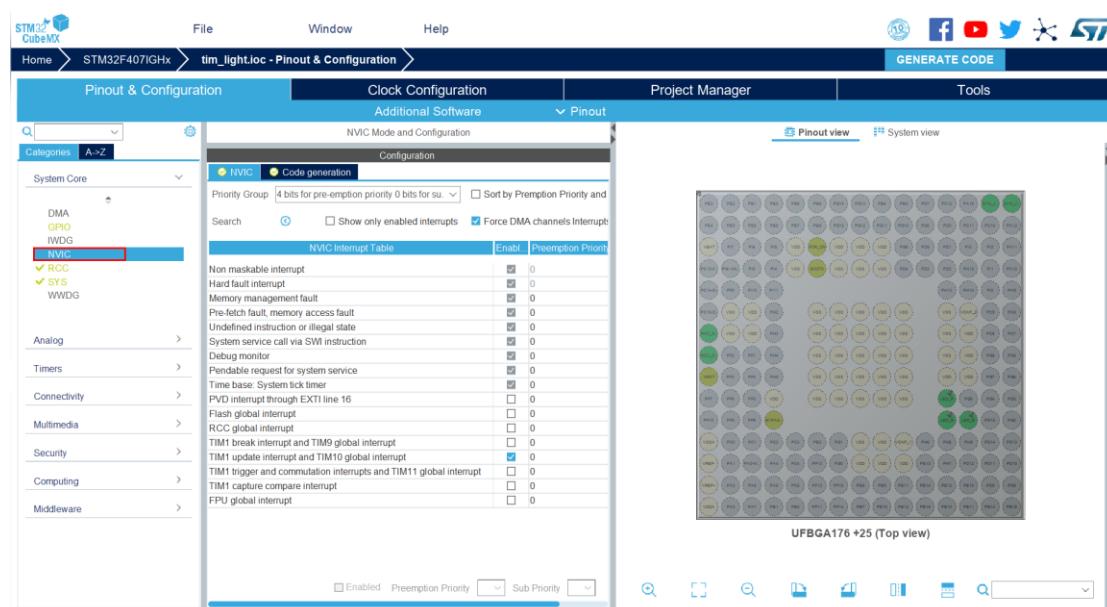
	抢占优先级数	响应优先级数
000	0	0-15
001	0-1	0-7
010	0-3	0-3
011	0-7	0-1
100	0-15	0

拥有相同抢占优先级的中断处于同一个中断分组下。

当多个中断发生时，先根据抢占优先级判断哪个中断分组能够优先响应，再到这个中断分组中根据各个中断的响应优先级判断哪个中断优先响应。

3.4.3 cubeMX 中的中断配置以及中断函数管理

在 cubeMX 的 NVIC 标签页下可以看到当前系统中的中断配置



列表中显示了当前系统中所有中断的使能情况与优先级设置。要使能中断则在 Enable 一栏

打勾，这里选中 TIM1 update interrupt，打勾，开启该中断。此外还可以在该页面下进行抢占优先级和响应优先级的分配和中断的两种优先级的配置。这里为定时器 1 的中断保持默认的 0, 0 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
Pre-fetch fault, memory access fault	<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"/>	0	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
TIM1 break interrupt and TIM9 global interrupt	<input type="checkbox"/>	0	0
TIM1 update interrupt and TIM10 global interrupt	<input checked="" type="checkbox"/>	0	0
TIM1 trigger and commutation interrupts and TIM11 global interrupt	<input type="checkbox"/>	0	0
TIM1 capture compare interrupt	<input type="checkbox"/>	0	0
FPU global interrupt	<input type="checkbox"/>	0	0

点击 Generate code，生成代码。

下面来看一下 HAL 库是如何对中断进行处理的。

在 `stm32f4xx_it.c` 中，找到 cubeMX 自动生成的中断处理函数

```
void TIM1_UP_TIM10_IRQHandler(void)
{
    /* USER CODE BEGIN TIM1_UP_TIM10_IRQHandler 0 */

    /* USER CODE END TIM1_UP_TIM10_IRQHandler 0 */
    HAL_TIM_IRQHandler(&htim1);
    /* USER CODE BEGIN TIM1_UP_TIM10_IRQHandler 1 */

    /* USER CODE END TIM1_UP_TIM10_IRQHandler 1 */
}
```

```
}
```

该函数调用了 HAL 库提供的 HAL_TIM_IRQHandler 这一函数

```
void HAL_TIM_IRQHandler(TIM_HandleTypeDef *htim)
```

函数名	HAL_TIM_IRQHandler
函数作用	HAL 对涉及中断的寄存器进行处理
返回值	void
参数	*htim 定时器的句柄指针, 如定时器 1 就输入&htim1, 定时器 2 就输入&htim2

在 HAL_TIM_IRQHandler 对各个涉及中断的寄存器进行了处理之后, 会自动调用中断回调函数 HAL_TIM_PeriodElapsedCallback, 该函数使用 __weak 修饰符修饰, 即用户可以在别处重新声明该函数, 调用时将优先进入用户声明的函数。

```
__weak void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
```

一般我们需要在中断回调函数中判断中断来源并执行相应的用户操作。

3.4.4 定时器回调函数介绍

如前文所介绍的, HAL 库在完成定时器的中断服务函数后会自动调用定时器回调函数。

通过配置 TIM1 的分频值和重载值, 使得 TIM1 的中断以 500ms 的周期被触发。因此中断回调函数也是以 500ms 为周期被调用。在 main.c 中重新声明定时器回调函数, 并编写内容如下:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim == &htim1)
    {
        //500ms trigger
        bsp_led_toggle();
    }
}
```

可以看到首先在回调函数中进行了中断来源的判断，判断其来源是否是定时器 1。如果有其他的定时器产生中断，同样会调用该定时器回调函数，因此需要进行来源的判断。

在确认了中断源为定时器 1 后，调用 `bsp_led_toggle` 函数，翻转 RGB 三色 LED 引脚的输出电平。

3.4.5 HAL_TIM_Base_Start 函数

如果不开启中断，仅让定时器以定时功能工作，为了使定时器开始工作，需要调用 HAL 库提供的函数。

`HAL_StatusTypeDef HAL_TIM_Base_Start(TIM_HandleTypeDef *htim)`

函数名	<code>HAL_TIM_Base_Start</code>
函数作用	使对应的定时器开始工作
返回值	<code>HAL_StatusTypeDef</code> , HAL 库定义的几种状态，如果成功使定时器开始工作，则返回 <code>HAL_OK</code>
参数	<code>*htim</code> 定时器的句柄指针，如定时器 1 就输入 <code>&htim1</code> , 定时器 2 就输入 <code>&htim2</code>

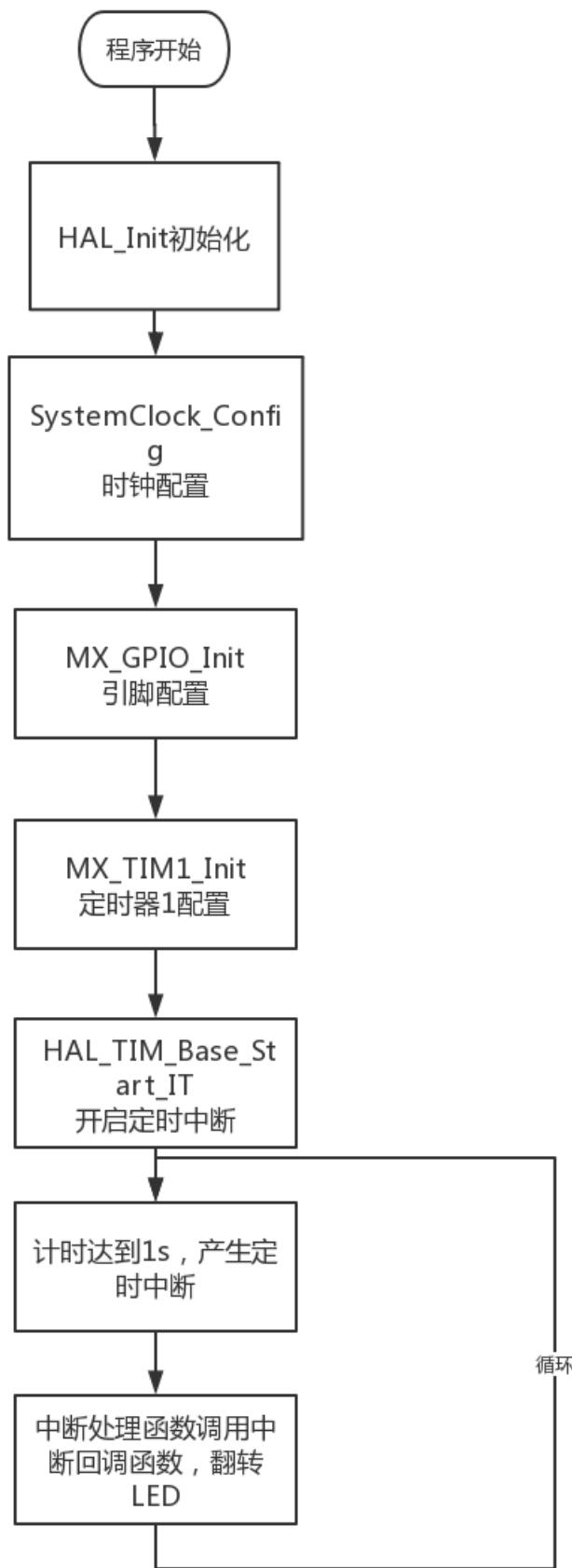
如果需要使用定时中断，则需要调用函数

`HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim)`

函数名	<code>HAL_TIM_Base_Start_IT</code>
函数作用	使对应的定时器开始工作，并使能其定时中断
返回值	<code>HAL_StatusTypeDef</code> , HAL 库定义的几种状态，如果成功使定时器开始工作，则返回 <code>HAL_OK</code>
参数	<code>*htim</code> 定时器的句柄指针，如定时器 1 就输入 <code>&htim1</code> , 定时器 2 就输入 <code>&htim2</code>

以上两个函数如果要使用则都需要在主循环 `while(1)` 之前调用。

3.4.6 程序流程



3.4.7 效果展示

定时器会以 500ms 定时触发中断，使得开发板 C 型上的 LED 灯会以 500ms 跳变一次，呈现闪烁的效果。使用定时器触发中断，时间精度决定于晶振的精度，比软件模拟延时精度大大提升。如图所示，分别为开发板 C 型 LED 点亮和熄灭效果。



LED 点亮图



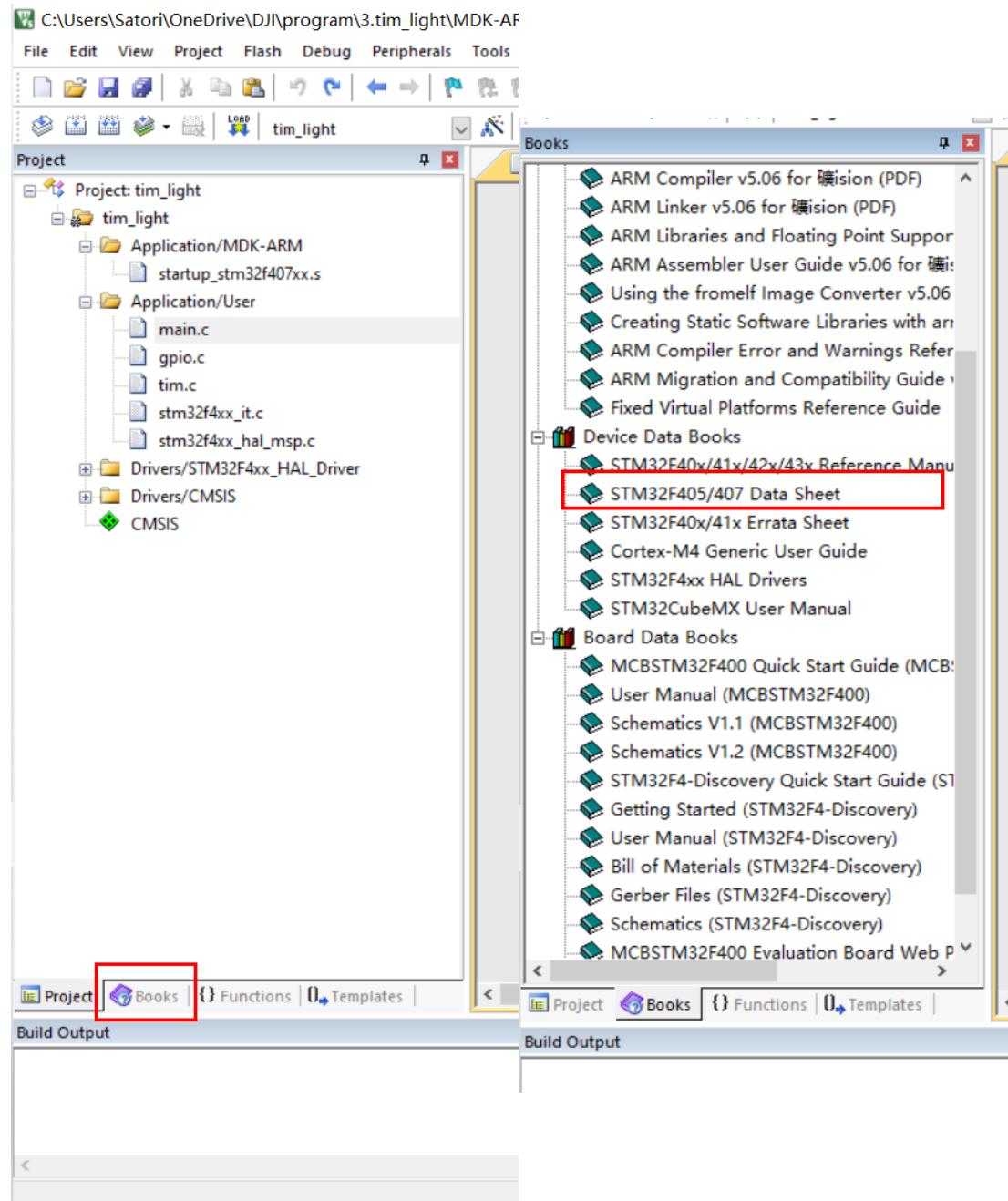
LED 熄灭图

3.5 进阶学习

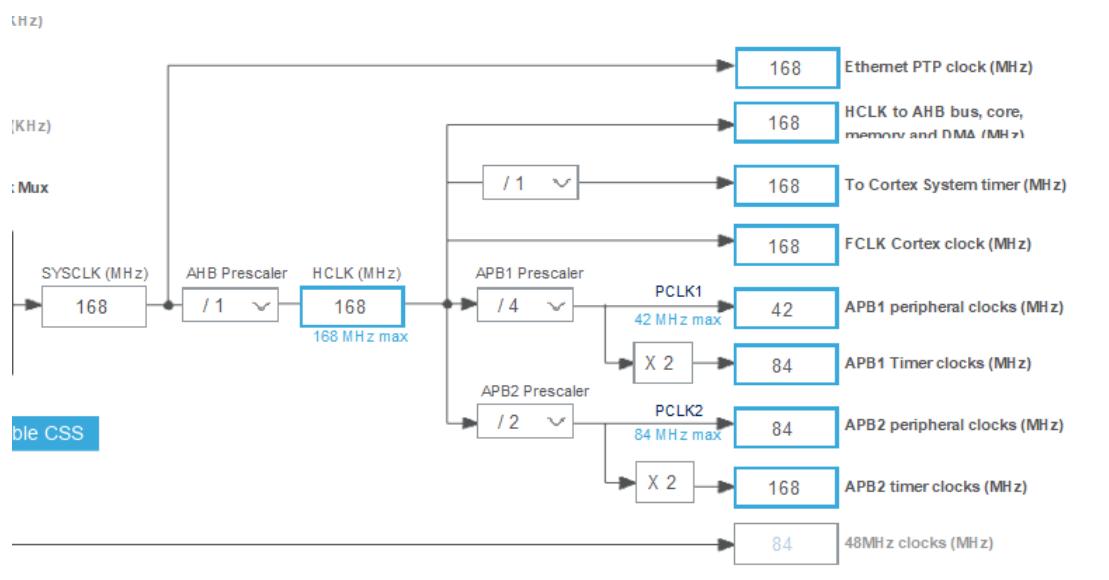
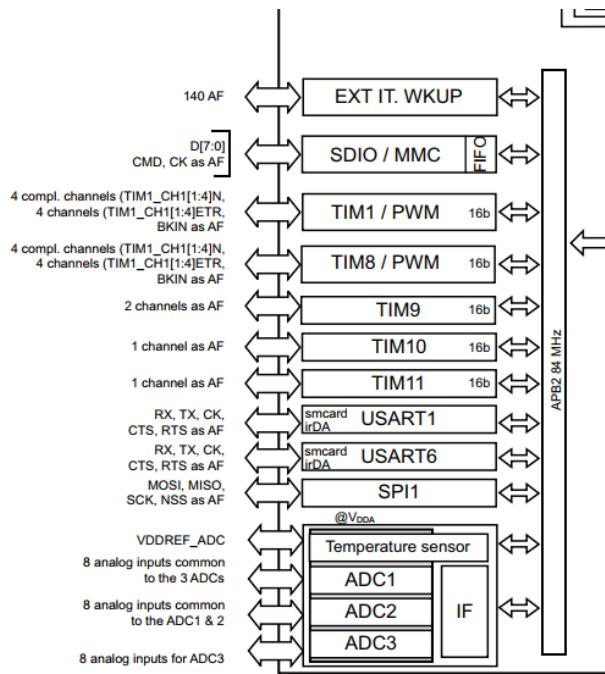
3.5.1 APB 总线计算定时器定时时间

上一小节中提到可以通过设置分频比和自动重装载值来控制定时器的定时周期，这一小节中

结合芯片手册（Datasheet）来详细介绍一下定时周期的计算过程。芯片手册可以在 Keil 的 book 标签页下获取。这里以 STM32F407IGHx 芯片为例，其他芯片同理。



在该文档的第 19 页，也就是 Device overview 章节，可以看到整个 stm32 内部的时钟是如何连接到各个外设上的，在图中可以看到 APB1 和 APB2 总线上挂载了大量的外设。定时器 1, 8, 9, 10, 11 挂载在 APB2 总线上。根据 datasheet 中给出的定时器所挂载的总线就可以确定定时器在分频前的时钟源频率。



在 cubeMX 的 **clock configuration** 标签页下，可以看到时钟树的结构。在整个时钟树的最右端，可以看到 APB1 和 APB2 两个总线的时钟频率设置，其中 **APBx peripheral clocks** 为挂载在总线上的定时器以外的外设提供时钟源，**APBx timer clocks** 为挂载在总线上的定时器提供时钟源。

确定时钟源频率之后，根据数据手册 640 页中的内容如图所示，分频值为 **TIMx_PSC** 中的分频值+1。即 **TIMx_SPC** 为 0 时，分频比刚好为 1:1，如果 **TIMx_SPC** 为 15，则分频比为 16:1，进入的 16MHz 的频率信号会被分频为 1MHz。

The counter clock frequency **CK_CNT** is equal to $f_{CK_PSC} / (PSC[15:0] + 1)$.

分频后的频率就是 **TIMx_CNT** 自增的频率，根据之前介绍的内容，当 **TIMx_CNT** 的值增长到 **TIMx_ARR** 中的值后，就会发生重载，并触发中断信号，相当于使用 **TIMx_ARR** 中的值又进行了一次分频。因此产生这个中断信号的频率应该为（需要加 1 是因为 **CNT** 是从 0 开始计数的）。

$$f = f_{CK_{CNT}} / (ARR[15:0] + 1)$$

结合上面两个式子，可以得到最终的用于计算定时器触发频率的公式

$$f = f_{CK_{PSK}} / ((PSC[15:0] + 1) * (ARR[15:0] + 1))$$

其中 **f_CKPSK** 的值通过之前所说的方法，找到对应的 APB 总线进行确认。

3.6 课程总结

本节课了解了 **stm32** 的定时器外设，学习了如何通过配置几个重要的参数使定时器按照期望的周期运行，利用定时器可以实现嵌入式系统中重要的后台机制，即按照稳定的周期执行定时任务。此外本节课还学习了中断这一重要的概念，通过开启多样的中断，并合理制定其优先级，才能够在机器人上实时地运行丰富的任务。

4. PWM 控制 LED 的亮度

4.1 知识要点

- PWM 基本知识
- aRGB 三原色合成灯效讲解
- cubeMX 中配置 PWM 定时器配置

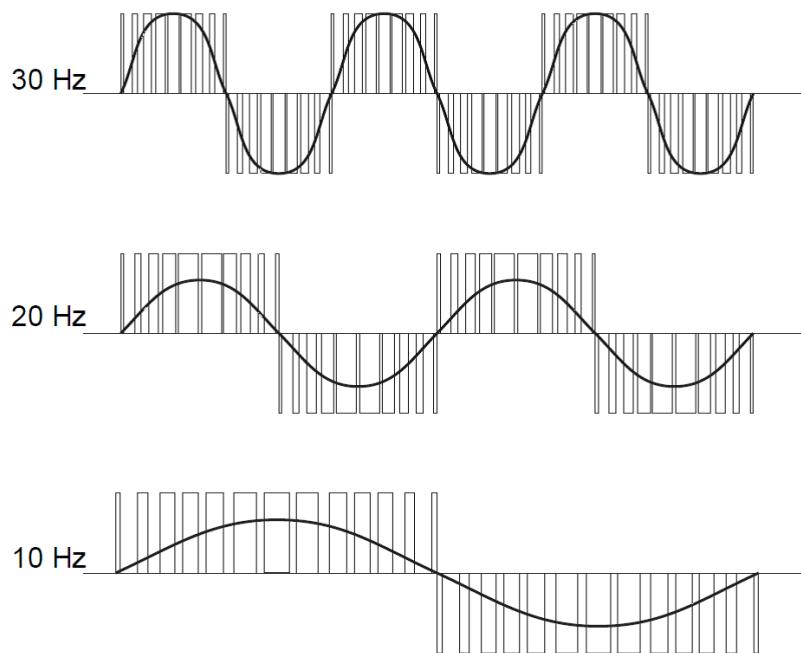
4.2 课程内容

本课程中，将学习到 PWM 的基本原理，如何控制引脚输出 PWM 信号，aRGB 三原色合成灯效的原理。本节课将通过在 cubeMX 中进行引脚配置，建立工程，并使用 PWM 输出实现 aRGB 合成等效。

4.3 基础学习

4.3.1 PWM 基本知识

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



例如上图中，矩形脉冲是 `stm32` 输出的数字信号，当这个信号接到外设上时，效果可以等效为这个正弦波。

一个周期内高电平的持续时间占总周期的比例成为占空比，通过修改占空比，可以改变输出的等效模拟电压。例如输出占空比为 50%，频率为 10Hz 的脉冲，高电平为 3.3V，则其输出的模拟效果相当于输出一个 1.65V 的高电平。此外 PWM 输出的频率也会影响最终的 PWM 输出效果，PWM 输出的频率越高，最终输出的“连续性”越好，越接近模拟信号的效果，频率低则会增强离散性，最终的输出效果会有比较强的“突变”感。

脉冲调制有两个重要的参数，第一个就是输出频率，频率越高，则模拟的效果越好。第二个就是占空比。占空比就是改变输出模拟效果的电压大小。占空比越大则模拟出的电压越大。

4.3.2 aRGB 三原色

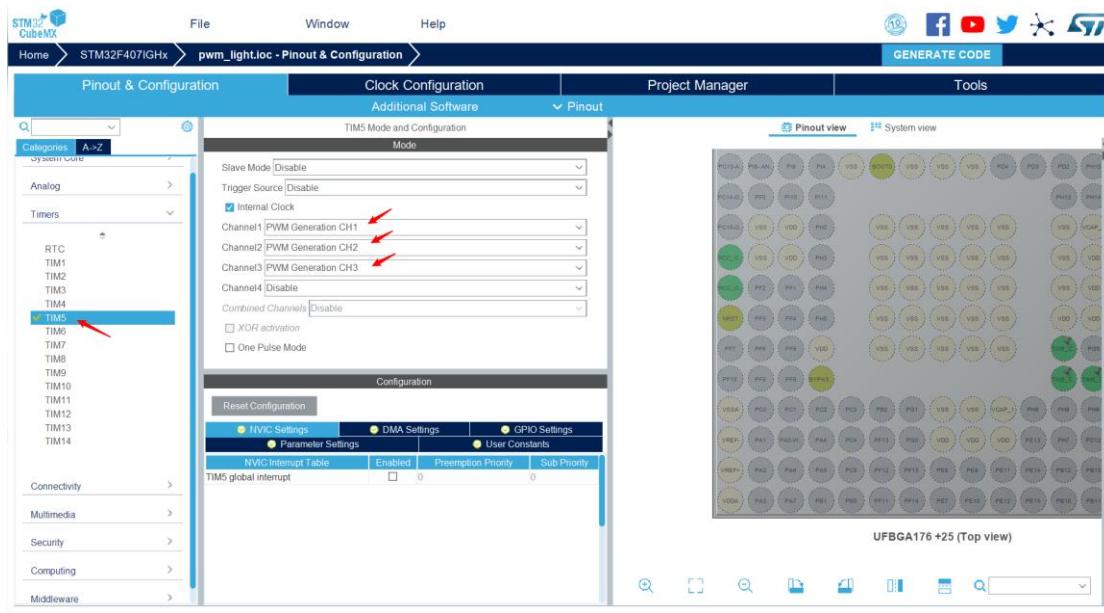
aRGB 为一种色彩模式，aRGB 分别代表了 alpha（透明度）Red（红色）Green（绿色）和 Blue（蓝色）四个要素，一般我们给每个要素设置十进制下 0-255 的取值范围，通过 16 进制表示就是 0x00-0xFF，因此一个 aRGB 值可以通过八位十六进制数来描述，从前到后每两位依次对应 a, R, G, B。

在 aRGB 中，alpha 值越大色彩越不透明，RGB 中哪个值越大，对应的色彩就越强。比如纯红色可以用 8 位 16 进制表示为 0xFFFF0000，纯绿色可以表示为 0xFF00FF00，纯蓝色可以表示为 0xFF0000FF，黄色由蓝色和绿色合成，所以可以表示为 0xFF00FFFF。

4.4 程序学习

4.4.1 PWM 在 cubeMX 中配置

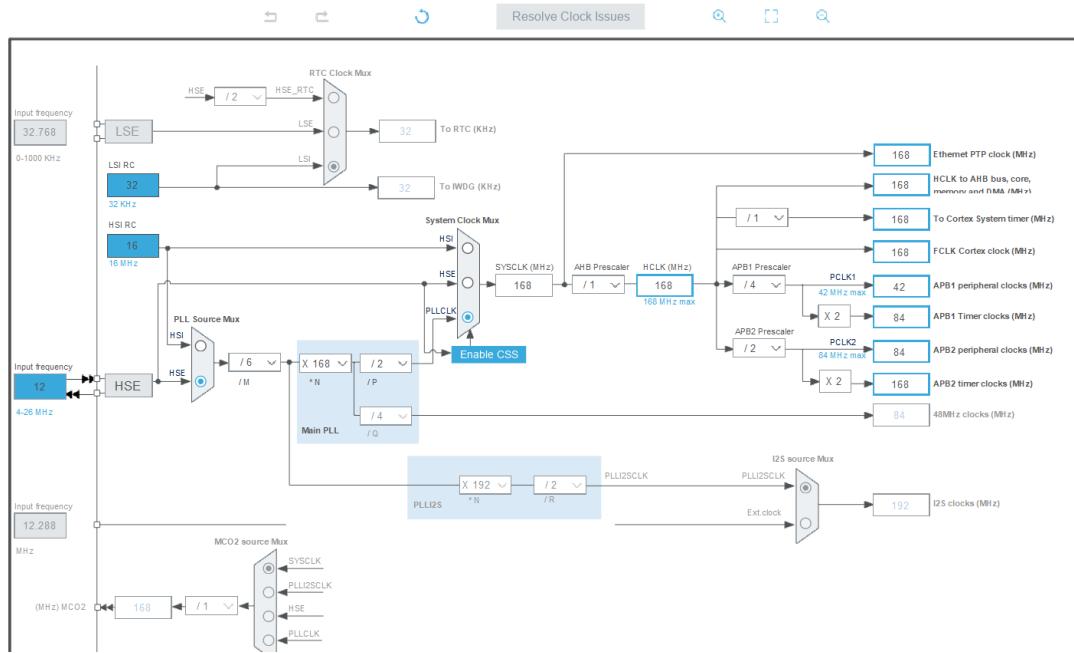
在 cubeMX 中设置定时器 5 的通道 1, 2, 3 为 PWM 输出。可以注意到三个通道对应的引脚正是之前的实验中使用的 LED 引脚。



定时器 5 如下配置，设置重载值为 65535。

Counter Settings	
Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register..)	65535
Internal Clock Division (CKD)	No Division
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)
PWM Generation Channel 1	
Mode	PWM mode 1
Pulse (32 bits value)	10000
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High
PWM Generation Channel 2	
Mode	PWM mode 1
Pulse (32 bits value)	10000
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High
PWM Generation Channel 3	
Mode	PWM mode 1
Pulse (32 bits value)	10000
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High

时钟树配置如下：



点击 **Generate Code**, 生成工程代码。

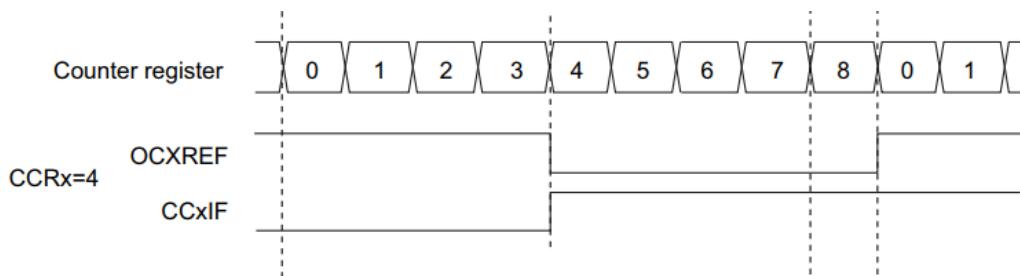
4.4.2 PWM 配置介绍

在上一节课中介绍了 STM32 的定时器，并提到 PWM 输出是 STM32 的定时器的功能之一，为了实现 PWM 功能，需要使用定时器中的比较寄存器（**TIMx_CCRx**）。

当定时器以 PWM 模式工作时，会自动将 **TIMx_CCRx** 的值与 **TIMx_CNT**（计数寄存器）中的值做比较，当 **TIMx_CNT** 中的值小于 **TIMx_CCRx** 的值时，PWM 输出引脚输出高电平，大于时则输出低电平。因此知道了 PWM 信号的周期和占空比可以通过设置比较寄存器 **TIMx_CCRx** 和定时器重载寄存器 **TIMx_ARR** 来控制。PWM 的占空比可以通过下图公式计算：

$$P = \frac{\text{TIM}x_{CCRx} - 1}{\text{TIM}x_{ARR}} * 100\%$$

以下图为例，该定时器的重载值为 8，比较寄存器值为 4，输出信号为 OCXREF，则其占空比为 44.4%。



一个定时器工作在 PWM 输出模式下时，有 4 个通道可以进行 PWM 信号的输出，每一个定时器都有对应标号的比较寄存器，比如 5 号定时器的 1 号通道对应的比较寄存器为 TIM5_CCR1。

修改比较寄存器 TIMx_CCRx 的值来控制 PWM 输出的占空比。在函数 aRGB_led_show 中，首先通过与运算与移位运算提取出对应的 alpha, R, G 和 B 通道值，然后用透明度 alpha 与 R, G, B 三者依次相乘，最后将其赋值通过 __HAL_TIM_SetCompare 函数给对应的比较寄存器 TIM5->CCRx。通过控制不同的 PWM 占空比，控制某个颜色的 LED 的亮度，以这样的方式就可以通过设置 aRGB 的值来控制最后输出的 LED 灯效。

```
void aRGB_led_show(uint32_t aRGB)
{
    static uint8_t alpha;
    static uint16_t red,green,blue;

    alpha = (aRGB & 0xFF000000) >> 24;
    red = ((aRGB & 0x00FF0000) >> 16) * alpha;
    green = ((aRGB & 0x0000FF00) >> 8) * alpha;
    blue = ((aRGB & 0x000000FF) >> 0) * alpha;

    __HAL_TIM_SetCompare(&htim5, TIM_CHANNEL_1, blue);
    __HAL_TIM_SetCompare(&htim5, TIM_CHANNEL_2, green);
    __HAL_TIM_SetCompare(&htim5, TIM_CHANNEL_3, red);
}

#define __HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__) \
(((__CHANNEL__) == TIM_CHANNEL_1) ? ((__HANDLE__)->Instance->CCR1 = (__COMPARE__)) : \
((__CHANNEL__) == TIM_CHANNEL_2) ? ((__HANDLE__)->Instance->CCR2 = (__COMPARE__)) : \
((__CHANNEL__) == TIM_CHANNEL_3) ? ((__HANDLE__)->Instance->CCR3 = (__COMPARE__)) : \
((__HANDLE__)->Instance->CCR4 = (__COMPARE__)))
```

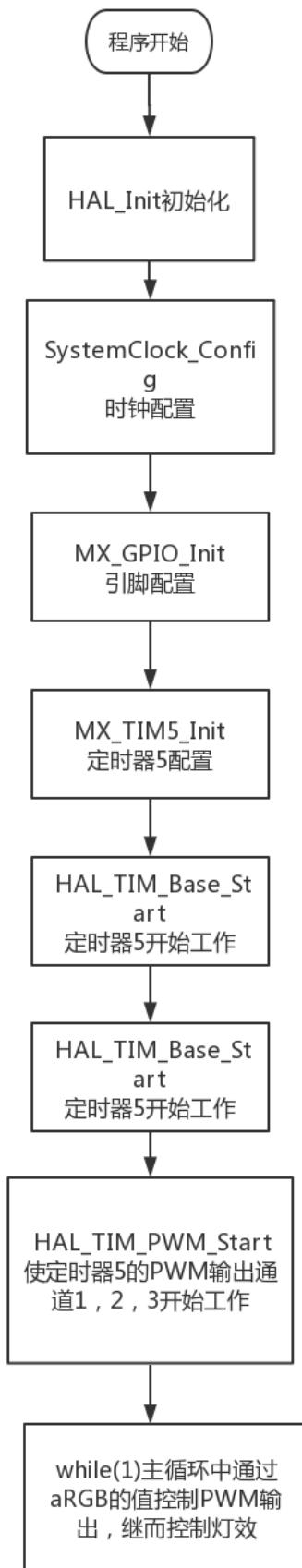
4.4.3 HAL_TIM_PWM_Start 函数介绍

为了使定时器开始 PWM 输出，除了要通过 HAL_TIM_Base_Start 使定时器开始工作，还需要在初始化时调用 HAL 库提供的 PWM 初始化函数 HAL_TIM_PWM_Start。

```
HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t Channel)
```

函数名	HAL_TIM_PWM_Start
函数作用	使对应定时器的对应通道开始 PWM 输出
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态, 如果成功使定时器开始工作, 则返回 HAL_OK
参数 1	*htim 定时器的句柄指针, 如定时器 1 就输入 &htim1, 定时器 2 就输入&htim2
参数 2	Channel 定时器 PWM 输出的通道, 比如通道 1 为 TIM_CHANNEL1

4.4.4 程序流程



4.4.5 效果展示

设置 ARGB 值为 0x7F123456，为半亮度偏蓝光灯效，如图所示。



PWM 控制 LED 灯效

4.5 课程总结

aRGB 灯效是一个常见的灯效，可以用于信号指示，状态显示，报错显示等用途。本节课学习的 PWM 输出功能可以用于舵机和电机控制，是非常常用的控制信号，需要熟练掌握其原理。

本节课的课后作业：在开发板上用 PWM 控制 LED 灯颜色循环切换，从红灯变成绿灯，再变成蓝灯，切换过程有呼吸效果。

5. 常见的 PWM 设备

5.1 知识要点

- 蜂鸣器类型知识
- 蜂鸣器的音调
- 舵机控制

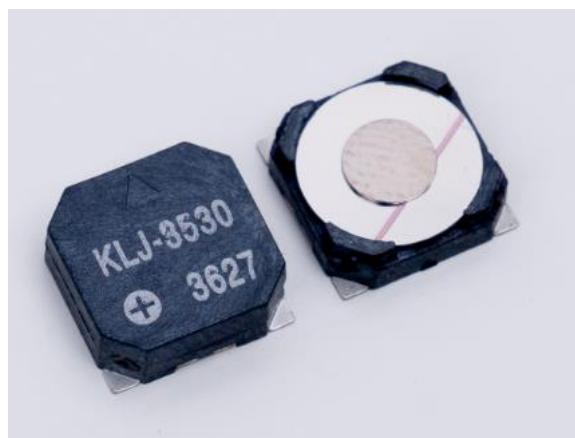
5.2 课程内容

本节课将了解更多使用 PWM 控制的设备-蜂鸣器，舵机和 snail 电机，学习使用 PWM 功能对蜂鸣器和舵机进行控制。同时将学习有源蜂鸣器和无源蜂鸣器的大致结构与原理以及舵机的基本知识。

5.3 基础学习

5.3.1 蜂鸣器

蜂鸣器是一种能够通过电子信号控制的发声器件。在生活中，几乎所有能够发出哔哔响声的电子器件中都装有蜂鸣器。蜂鸣器能够为使用者提供直观的声音信息，是一种常见的人机交互模式。在 RoboMaster 比赛中，常常用于提醒队员完成某项检查或者用于某个模块出问题时的报错。常见的蜂鸣器分为插针型和贴片型，在开发板 C 型使用到蜂鸣器为贴片型，如下图所示。



根据是否内置震荡电路可分为有源蜂鸣器和无源蜂鸣器。有源蜂鸣器只需要提供直流电压就可以通过内部的震荡电路产生震荡电流进而发出声音，而无源蜂鸣器需要输入特定频率的方

波才能发出声音。两者比较起来，有源蜂鸣器的控制更加简单，但是只能发出单一频率的声音，而无源蜂鸣器虽然控制起来比较麻烦，但是可以通过改变输入方波的频率发出不同音调的声音，甚至可以用来演奏乐曲。

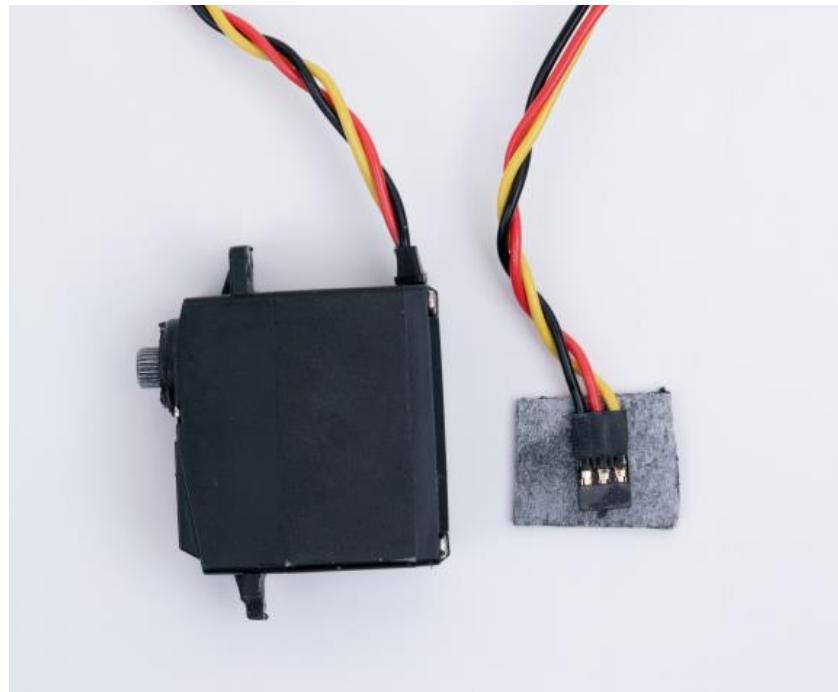
两者的对比可见下表：

	有源蜂鸣器	无源蜂鸣器
内置震荡源	有	无
激励方式	直流电压	特定频率方波
音调	固定	可变

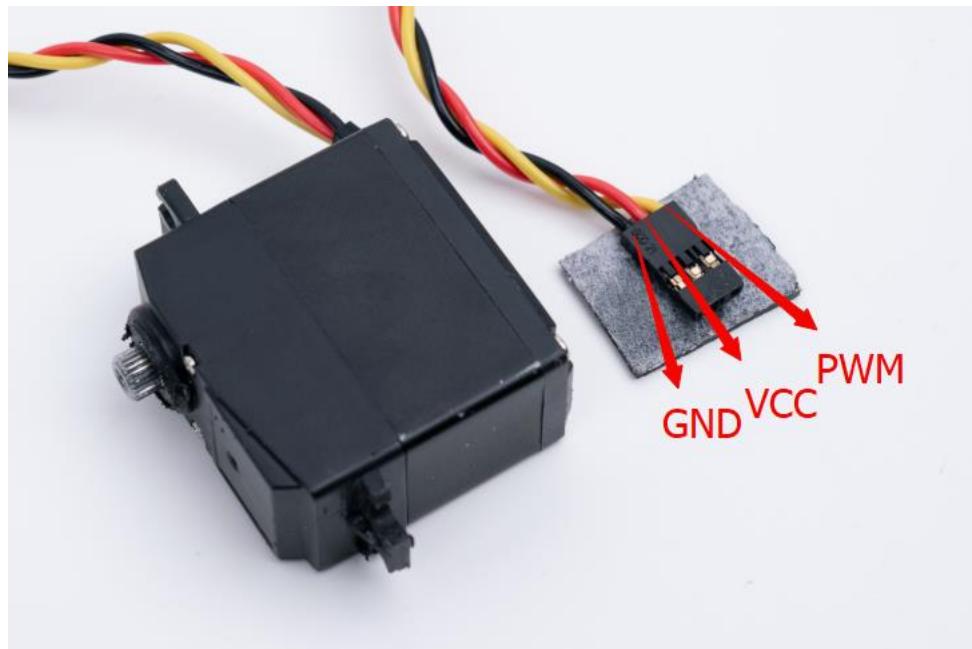
5.3.2 舵机的控制

舵机是机器人中的常见的执行部件，通常使用特定频率的 PWM 进行控制。

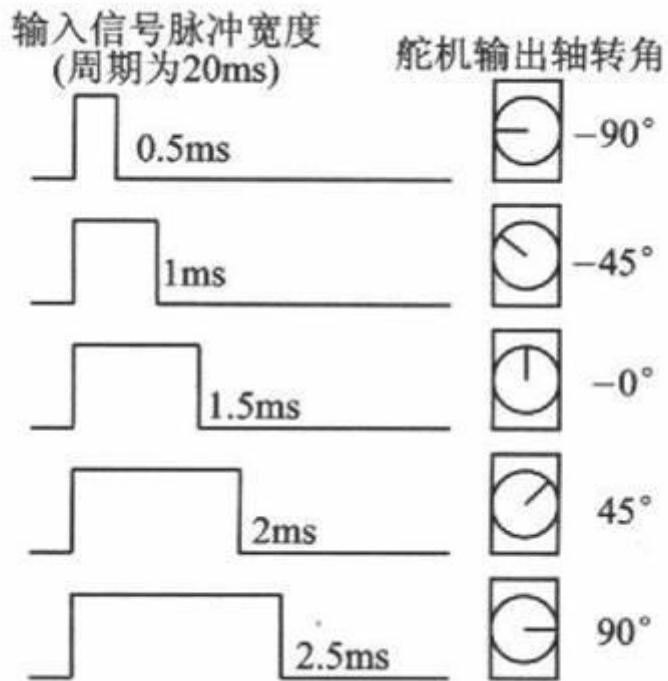
舵机的主要组成部位由一个小型的电机和传动机构（齿轮组）构成，多被用于操控飞行器上的舵面，故而得名舵机。由于控制简单，价格便宜，在 RoboMaster 比赛中，用于简单的动作控制，例如使用舵机控制弹仓盖的开合。下图为市场上常见的舵机。



通常舵机的三根线按照颜色分别为：黑色-GND，红色-VCC，黄色-PWM 信号。在使用舵机时，只需要使用杜邦线或者其他连接线接入开发板 C 型对应的 PWM 接口。



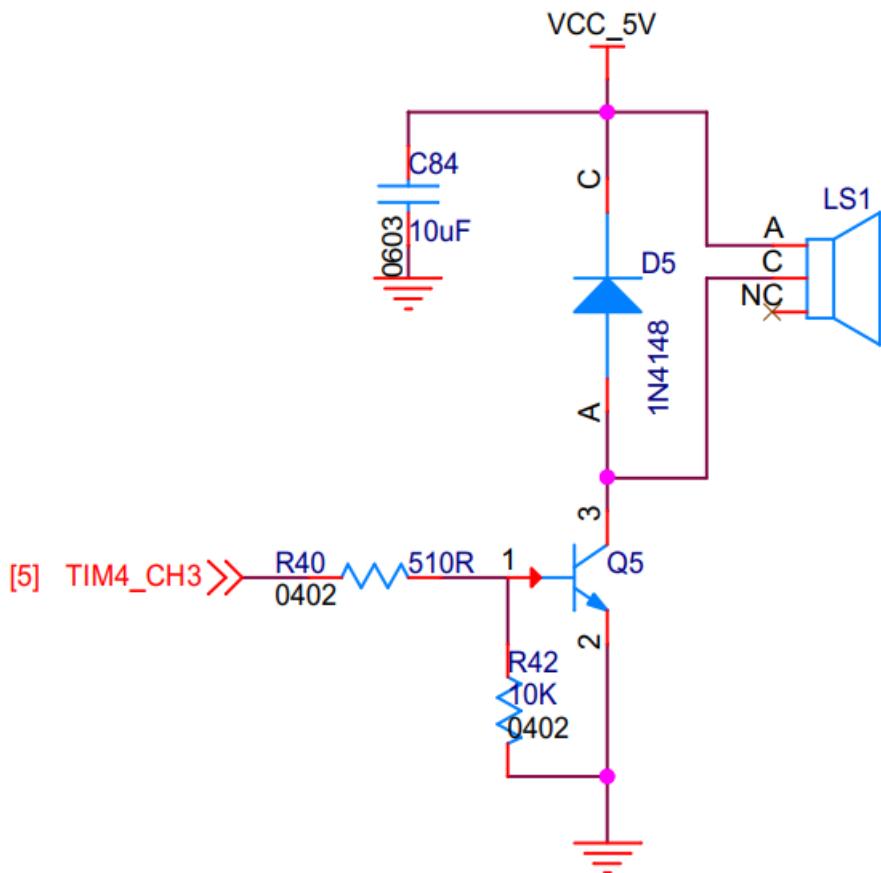
舵机使用的 PWM 信号一般为频率 50Hz，高电平时间 0.5ms-2.5ms 的 PWM 信号，不同占空比的 PWM 信号对应舵机转动的角度，以 180 度舵机为例，对应角度图如下图所示。



5.4 程序学习

5.4.1 蜂鸣器的 PWM 在 cubeMX 中配置

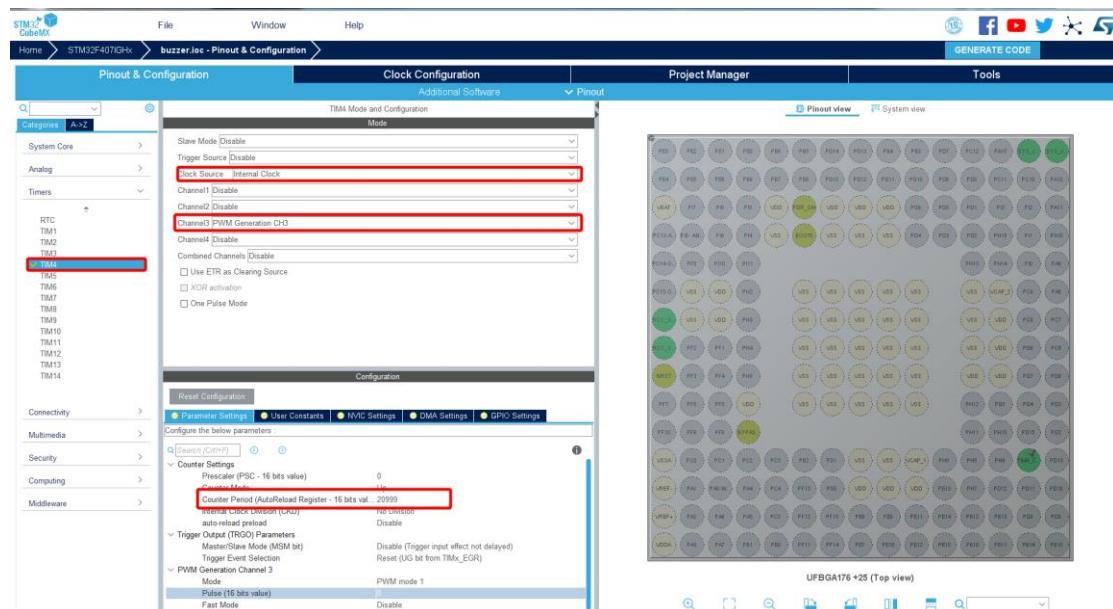
查看开发板的原理图，如图所示。



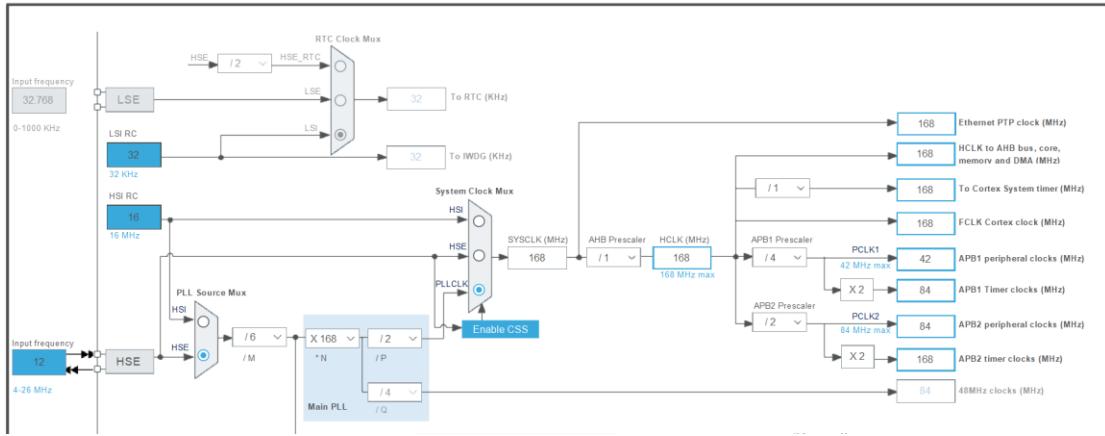
Buzzer

蜂鸣器使用的引脚为 PD14，为定时器 4 的通道 3。

打开 cubeMX，使能定时器 4，预分配设置为 0，重载值设置为 20999，设置通道 3 为 PWM 输出，其余设置保持默认即可，此时开发板的 PD14 引脚变为绿色。



时钟树配置如图。



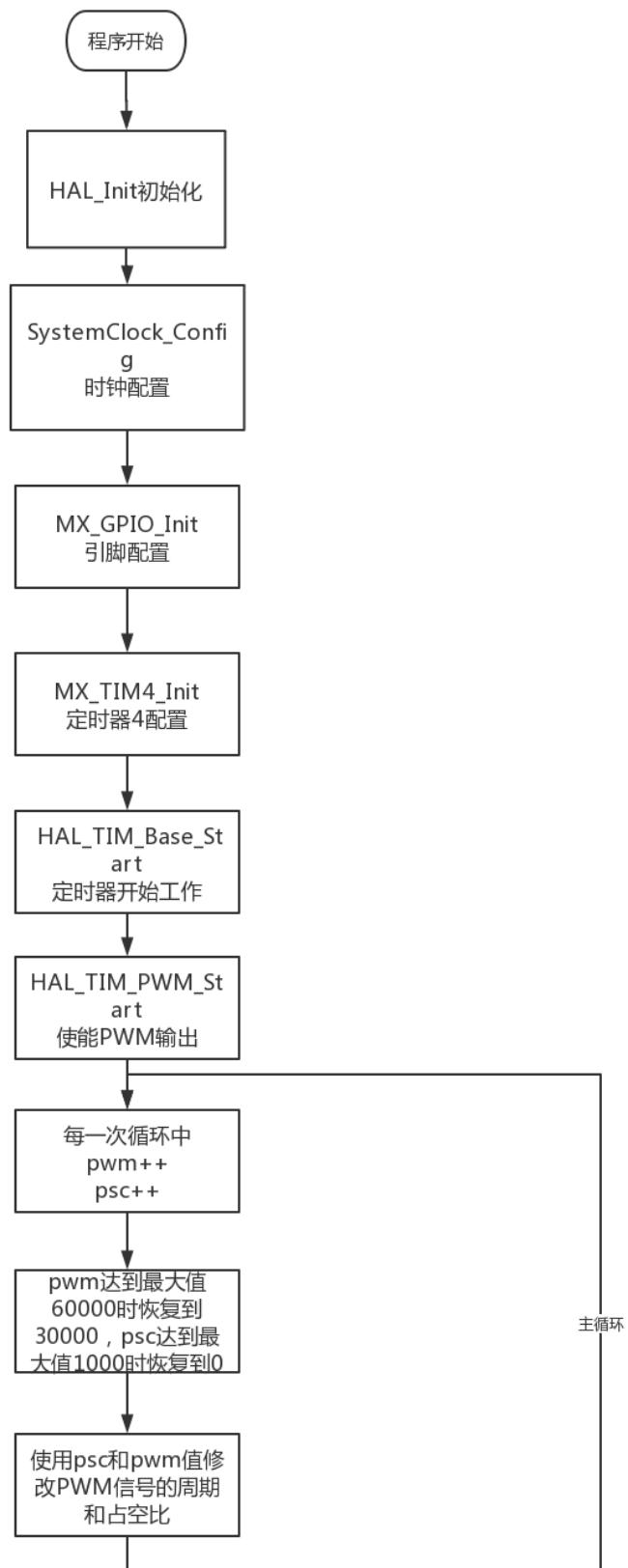
点击 **Generate Code**, 生成工程。

根据在定时器章节的进阶学习部分的知识,可以通过查看源代码或者数据手册的方式知道定时器 4 挂载在 APB1 总线上, 对应的总线频率为 84MHz, 分频值为 0, 重载值为 20999, 并通过公式计算得到 PWM 波的输出频率为 4000Hz。

5.4.2 蜂鸣器的程序流程

在之前的原理介绍中, 改变 PWM 的频率就可以改变无源蜂鸣器的音调。故而改变定时器的分频系数和重载值, 改变 PWM 的频率, 就能够控制无源蜂鸣器发出的响声频率。

在主程序中, 声明了 `psc` 和 `pwm` 两个变量, 分别控制定时器 4 的分频系数和重载值, 每一次循环中这两个变量进行一次自加。通过宏定义的方式, 设置 `pwm` 的值在 `MIN_BUZZER_PWM` (10000) 和 `MAX_BUZZER_PWM` (20000) 之间变动, `psc` 的值在 0 和 `MAX_PSC` (1000) 之间变动, 程序主流程如图所示。



主函数代码如下：

```
while (1)

{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

    pwm++;

    psc++;

    if(pwm > MAX_BUZZER_PWM)
    {
        pwm = MIN_BUZZER_PWM;
    }

    if(psc > MAX_PSC)
    {
        psc = 0;
    }

    buzzer_on(psc, pwm);

    HAL_Delay(1);
}
```

在 `buzzer_on` 函数中，将 `psc` 和 `pwm` 两个值赋值给定时器 4 的预分频计数器和 3 号通道

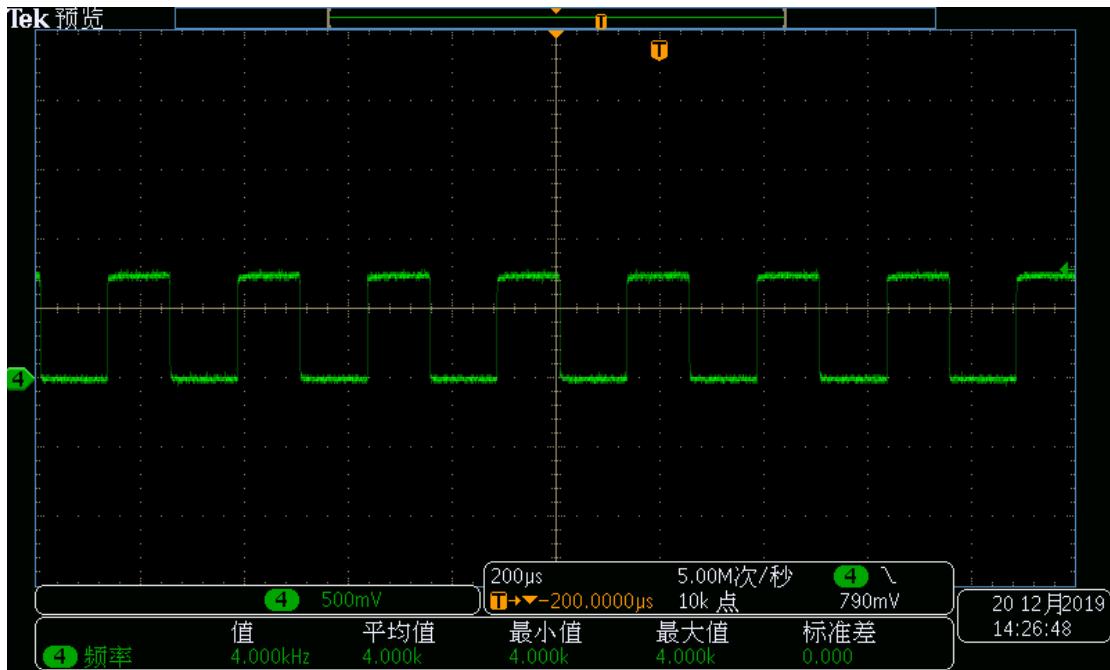
的比较寄存器，从而调整 PWM 信号的周期，进而控制蜂鸣器发出不同声调，不同响度的声音。

```
void buzzer_on(uint16_t psc, uint16_t pwm)
```

函数名	buzzer_on
函数作用	控制蜂鸣器定时器的分频和重载值
返回值	Void
参数 1: psc	设置定时器的分频系数
参数 2: pwm	设置定时器的重载值

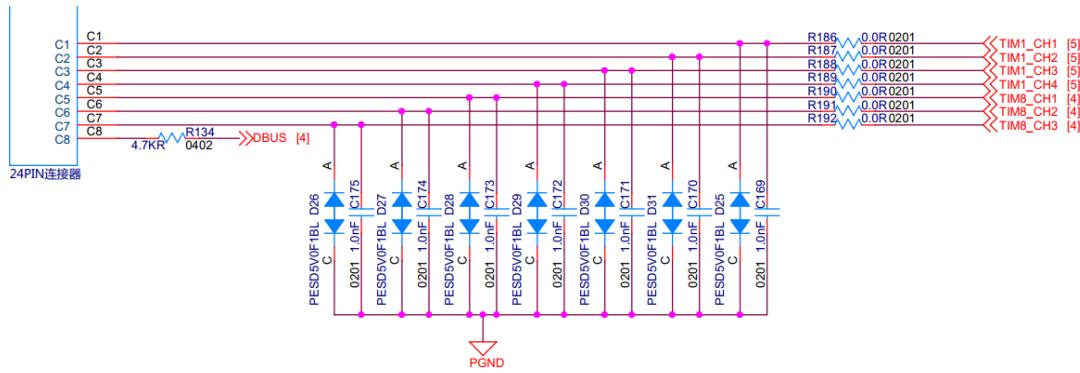
5.4.3 效果展示

使用示波器可以观察 PWM 输出波形，得到 4kHz 的 PWM 波形图，同时蜂鸣器会发生响声。



5.4.4 舵机的 PWM 在 cubeMX 中配置

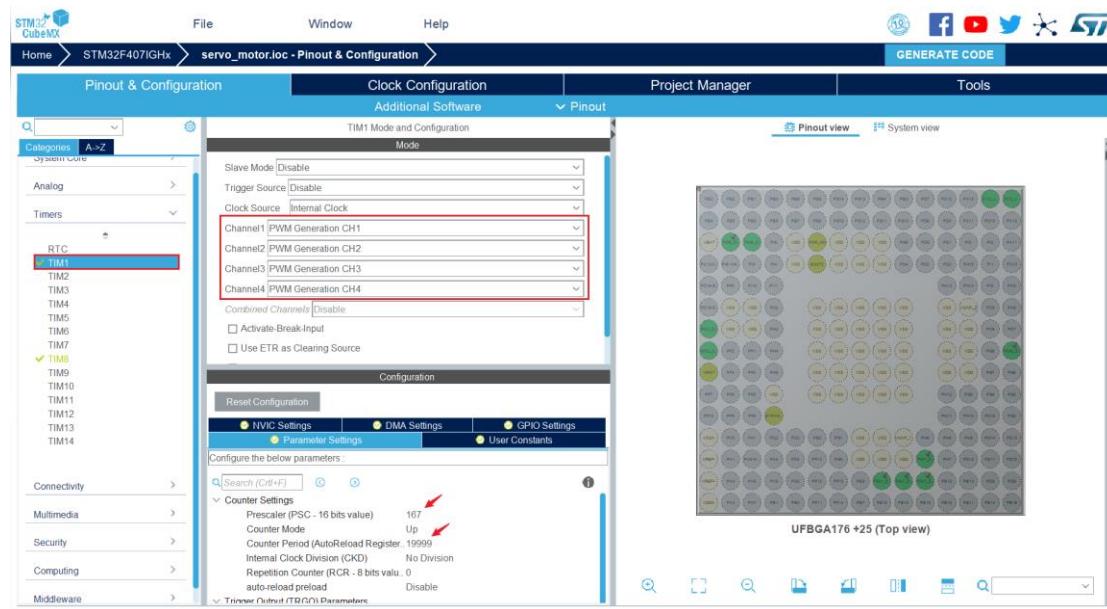
在开发板上具有 7 个 PWM 的输出接口，原理图如图所示，实物图如图所示。



7路PWM输出



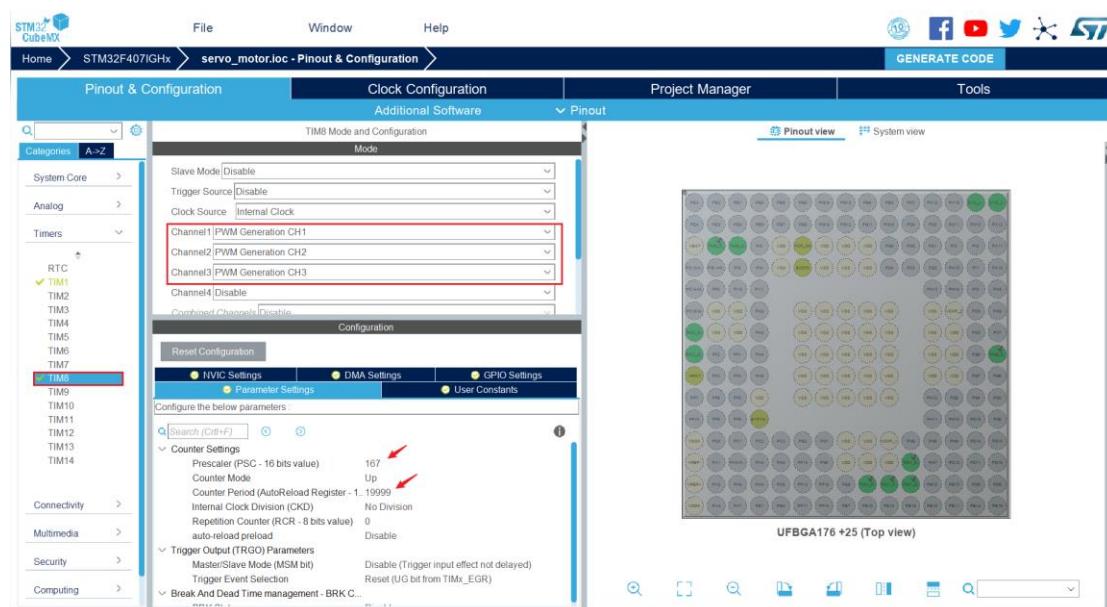
首先开启定时器 1，预分频值设置为 167，重载值设置为 19999。打开 1-4 的 PWM 输出通道，在 PWM 通道的设置中，将 Pulse 值设置为 2000，比较寄存器的初始值就会被设成 2000。



STM32CubeMX Software Screenshot showing PWM Generation Channel settings:

Mode	PWM mode 1
Pulse (16 bits value)	2000
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset
Mode	PWM mode 1
Pulse (16 bits value)	2000
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset
Mode	PWM mode 1
Pulse (16 bits value)	2000
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset
Mode	PWM mode 1
Pulse (16 bits value)	2000
Fast Mode	Disable
CH Polarity	High

接着开启定时器 8，将预分频值设置为 167，重载值设置为 19999。打开 1-3 的 PWM 输出通道，在 PWM 通道的设置中，将 Pulse 值设置为 2000。



✓ PWM Generation Channel 1

Mode	PWM mode 1
Pulse (16 bits value)	2000
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset

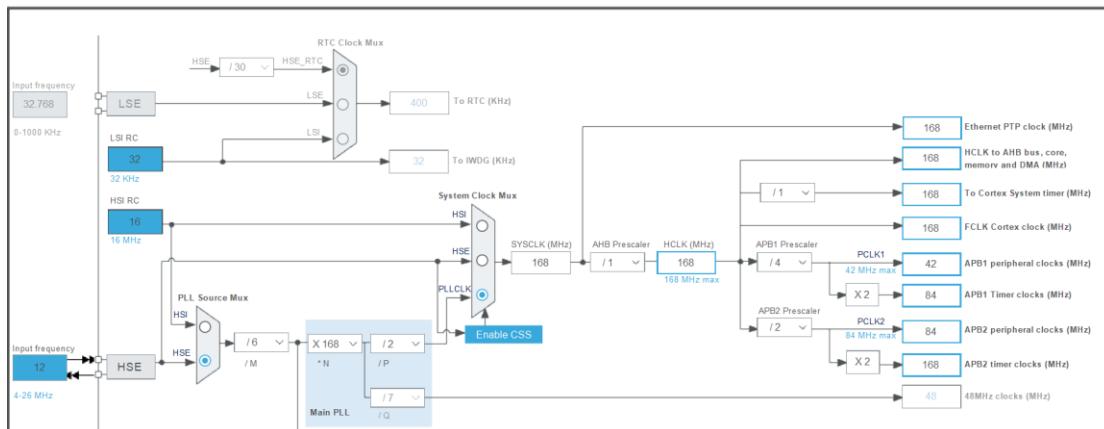
✓ PWM Generation Channel 2

Mode	PWM mode 1
Pulse (16 bits value)	2000
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset

✓ PWM Generation Channel 3

Mode	PWM mode 1
Pulse (16 bits value)	2000
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset

最后将时钟树如下进行配置



可以通过查看源代码或者数据手册的方式我们知道定时器 1 和 8 挂载在 APB2 总线上，对应的总线频率为 168MHz，定时器分频值为 167，重载值 19999，并通过公式计算得到 PWM 波的输出频率为 50Hz，对应的周期为 20ms。

通过 PWM 章节部分学习的知识，计算出 PWM 占空比最小为 500/20000 即 2.5%，对应高电平时间为 20ms 乘以 2.5% 等于 0.5ms，最大为 2000/20000 即 10%，对应高电平时间为 20ms 乘以 10% 等于 2ms。

5.4.5 舵机主程序讲解

在初始化时，通过 `HAL_TIM_Base_Start` 函数启动定时器 1 和定时器 8，再通过

`HAL_TIM_PWM_Start` 函数将定时器 1 的 1,2,3,4 号通道的 PWM 输出和定时器 8 的 1,2,3 通道的 PWM 输出开启。

在主循环中，通过 `_HAL_TIM_SetCompare` 来设置 PWM 的占空比，需要注意的是 `_HAL_TIM_SetCompare` 并非是一个函数，查看定义后会发现这其实是一个宏，最后其功能的依然是依靠将数值赋值给定时器某一通道比较寄存器来实现的。

```
#define __HAL_TIM_SetCompare          _HAL_TIM_SET_COMPARE  
  
#define _HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__) \  
((__CHANNEL__) == TIM_CHANNEL_1) ? ((__HANDLE__)->Instance->CCR1 = (__COMPARE__)) : \  
((__CHANNEL__) == TIM_CHANNEL_2) ? ((__HANDLE__)->Instance->CCR2 = (__COMPARE__)) : \  
((__CHANNEL__) == TIM_CHANNEL_3) ? ((__HANDLE__)->Instance->CCR3 = (__COMPARE__)) : \  
((__HANDLE__)->Instance->CCR4 = (__COMPARE__)))
```

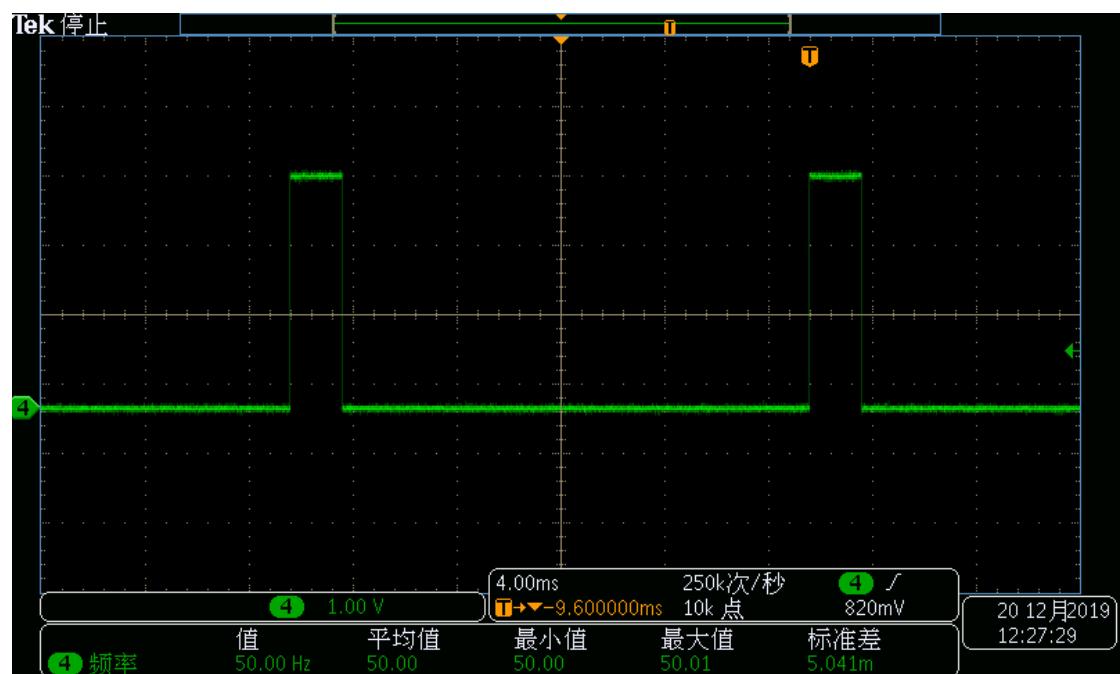
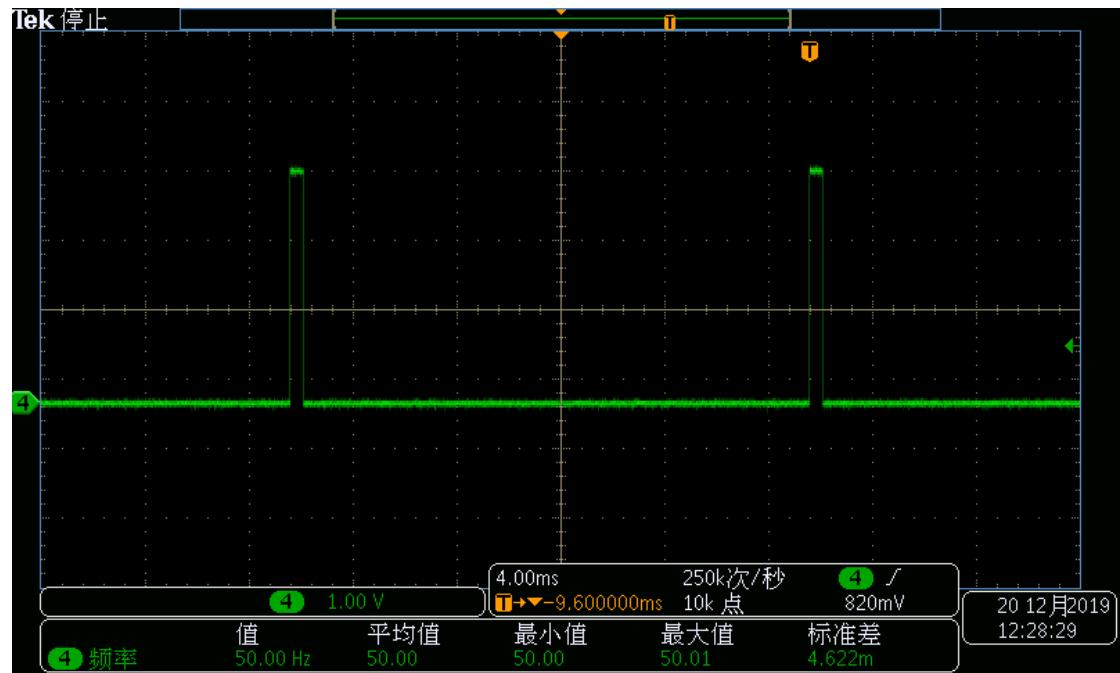
使用 `_HAL_TIM_SetCompare` 进行占空比设置时，依次对应的输入为：

参数 1	*htim 定时器的句柄指针，如定时器 1 就输入 <code>&htim1</code> ，定时器 2 就输入 <code>&htim2</code>
参数 2	Channel 定时器 PWM 输出的通道，比如通道 1 为 <code>TIM_CHANNEL_1</code>
参数 3	需要赋值给比较寄存器的值，PWM 占空比等于比较值除以重载值

随着主循环中各个 PWM 输出的占空比的变化，舵机的转动角度也随之变化。

5.4.6 舵机效果演示

使用示波器观察 PWM 在 0.5ms 高电平和 2.5ms 高电平的输出波形，如下图所示。



5.5 课程总结

蜂鸣器是一种常见的人机交互模式，可以用于提醒队员完成某项检查，或者用于报警，提醒某个模块离线情况。舵机是一种简单易用的动力装置，可以用于机器人弹仓盖的开合，简单的机械爪控制等。

5.6 上一节作业讲解

程序流程

宏定义#define RGB_FLOW_COLOR_CHANGE_TIME 500 规定 RGB 切换颜色的时间为 500ms，接着创建数组 RGB_flow_color 来装载蓝 0xFF0000FF，绿 0xFF00FF00，红 0xFF0000 三种颜色，这里设置数组长度为 4 而不是 3 是为了便于实现后续的循环切换功能。

首先初始化定时器 5 和 1,2,3 通道的 PWM 输出后进入主循环。为了实现呼吸灯灯效，需要以合适的速度一点一点的修改 PWM 的占空比，这样灯的亮灭就会产生流畅的明暗渐变效果，也就是所谓的呼吸灯。

在主循环中，定义了浮点型的每一个周期中各个元素的改变量 delta_alpha, delta_red, delta_green, delta_blue，又定义浮点型的四个元素值 alpha, red, green, blue，最后定义了提供给 PWM 进行输出的 32 位表示的 aRGB 值。

在外层循环中，首先从数组中读取本次循环中灯的颜色，通过位运算将其提取出来，赋值给 alpha, red, green, blue。

```
alpha = (RGB_flow_color[i] & 0xFF000000) >> 24;  
red = ((RGB_flow_color[i] & 0x00FF0000) >> 16);  
green = ((RGB_flow_color[i] & 0x0000FF00) >> 8);  
blue = ((RGB_flow_color[i] & 0x000000FF) >> 0);
```

接着，用下次各个元素值和本次的元素值做差，然后除以切换总时间 (RGB_FLOW_COLOR_CHANGE_TIME) 以求得内层循环中每一次各个元素的改变量 delta。

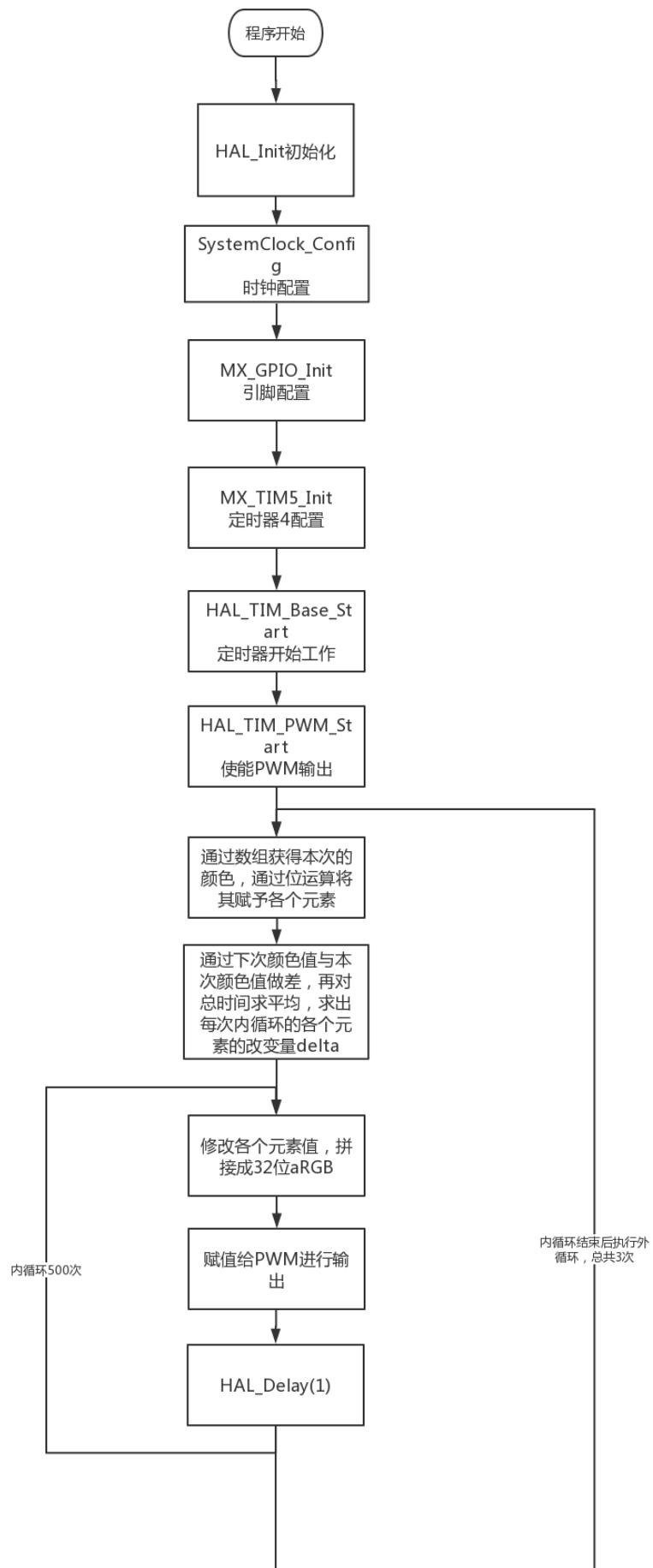
```
delta_alpha = (fp32)((RGB_flow_color[i + 1] & 0xFF000000) >> 24) - (fp32)((RGB_flow_color[i] & 0xFF000000) >> 24);  
delta_red = (fp32)((RGB_flow_color[i + 1] & 0x00FF0000) >> 16) - (fp32)((RGB_flow_color[i] & 0x00FF0000) >> 16);  
delta_green = (fp32)((RGB_flow_color[i + 1] & 0x0000FF00) >> 8) - (fp32)((RGB_flow_color[i] & 0x0000FF00) >> 8);  
delta_blue = (fp32)((RGB_flow_color[i + 1] & 0x000000FF) >> 0) - (fp32)((RGB_flow_color[i] & 0x000000FF) >> 0);  
  
delta_alpha /= RGB_FLOW_COLOR_CHANGE_TIME;  
delta_red /= RGB_FLOW_COLOR_CHANGE_TIME;  
delta_green /= RGB_FLOW_COLOR_CHANGE_TIME;
```

```
delta_blue /= RGB_FLOW_COLOR_CHANGE_TIME;
```

在内层循环中，以 `delta` 值为单位，每一次都对各个元素值进行微小的改变，然后将各个元素值拼接成 32 位 aRGB 表示，用于给 PWM 通道输出，最后进行 1ms 的延时。这样就能够产生灯光明暗渐变地切换颜色的呼吸灯效果。

```
for(j = 0; j < RGB_FLOW_COLOR_CHANGE_TIME; j++)  
{  
    alpha += delta_alpha;  
    red += delta_red;  
    green += delta_green;  
    blue += delta_blue;  
  
    aRGB = ((uint32_t)(alpha) << 24 | ((uint32_t)(red)) << 16 | ((uint32_t)(green)) << 8 | ((uint32_t)(blue)) << 0;  
    aRGB_led_show(aRGB);  
    HAL_Delay(1);  
}
```

程序流程图如下：



6. 按键的外部中断

6.1 知识要点

- 按键的硬件原理知识
- GPIO 的外部中断配置
- 按键软件消抖处理
- 程序中的前后台

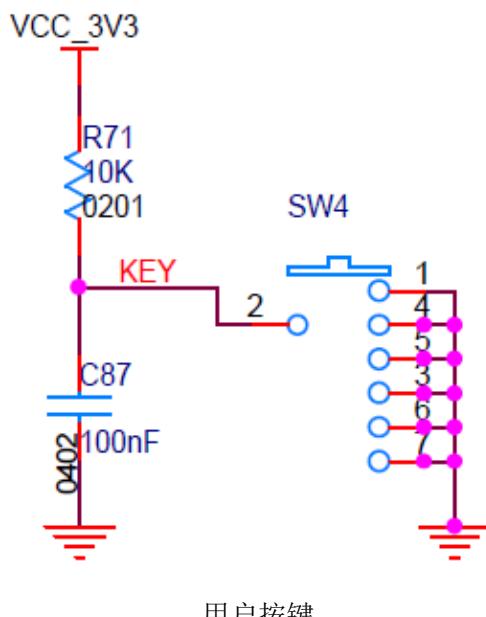
6.2 课程内容

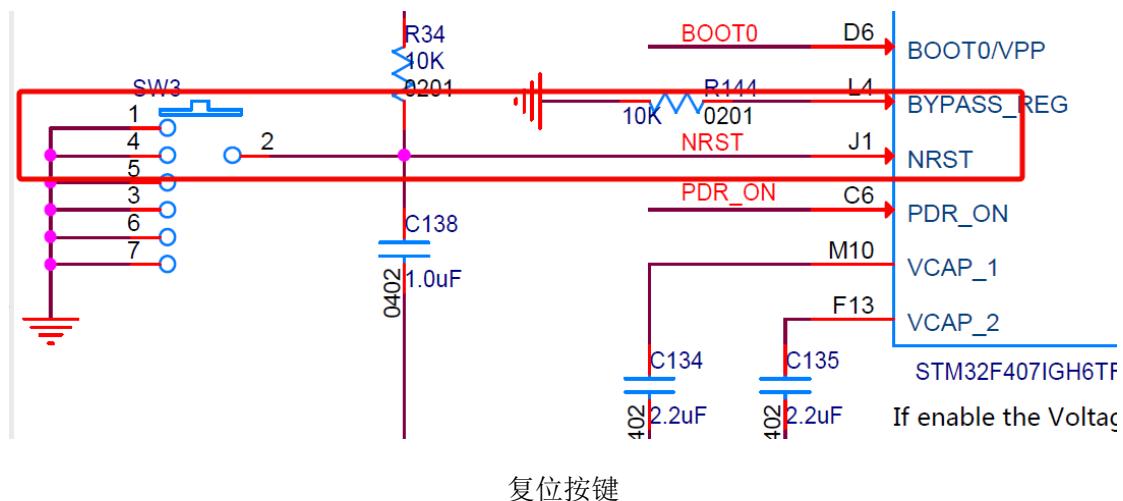
本节课中，将介绍按键的硬件原理，如何使用 stm32 的外部中断功能读取按键输入，如何使用软件消抖来消除按键输入产生的电压的抖动。此外，将再一次借助按键中断的例子帮助大家理解通过中断实现程序前后台的思想。

6.3 基础学习

6.3.1 按键原理图介绍

开发板 C 型有两个按键，其中一个为复位按键，另一个为用户自定义按键，如图所示。当按键未按下时，引脚将被上拉电阻钳位在高电平；当按键按下时引脚将直连 GND，处于低电平。



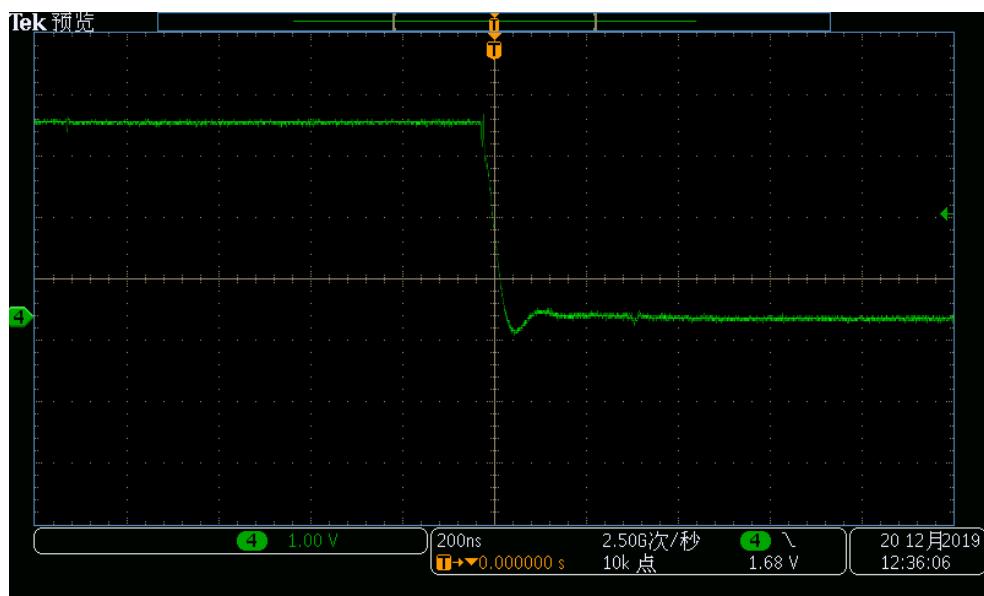


6.3.2 按键软件消抖

首先来介绍一下按键输入的抖动问题，由于按键的机械结构具有弹性，按下时开关不会立刻接通，断开时也不会立刻断开，这就导致按键的输入信号在按下和断开时都会存在抖动，如果不先将抖动问题进行处理，则读取的按键信号可能会出现错误。

为了消除这一问题，可以通过软件消抖和硬件消抖两种方式来实现，本节介绍软件滤波的实现方法。软件滤波的思想非常简单，抖动产生在按键按下和松开的两个边沿时刻，也叫下降沿（电平从高到低）和上升沿（电平从低到高）时刻，所以只需要在边沿时进行延时，等到按键输入已经稳定再进行信号读取即可。

一般采用软件消抖时，会进行 20ms 的延时，示波器采集按键波形如图所示。



按键按下波形图

6.3.3 外部中断

外部中断通常是 GPIO 的电平跳变引起的中断。在 stm32 中，每一个 GPIO 都可以作为外部中断的触发源，外部中断一共有 16 条线，对应着 GPIO 的 0-15 引脚，每一条外部中断都可以与任意一组的对应引脚相连，但不能重复使用。例如外部中断 Line0 可以和 PA0, PB0, PC0 等任意一条 0 号引脚相连，但如果已经和 PA0 相连，就不能同时和 PB0, PC0 其他引脚相连。

外部中断支持 GPIO 的三种电平跳变模式，如下所示：

- 上升沿中断：当 GPIO 的电平从低电平跳变成高电平时，引发外部中断。
- 下降沿中断：当 GPIO 的电平从高电平跳变成低电平时，引发外部中断。
- 上升沿和下降沿中断：当 GPIO 的电平从低电平跳变成高电平和从高电平跳变成低电平时，都能引发外部中断。

6.4 程序学习

6.4.1 外部中断在 cubeMX 中的配置

STM32 的 GPIO 提供外部中断功能，当 GPIO 检测到电压跳变时，就会发出中断触发信号给 STM32，使程序进入外部中断服务函数。

将 PA0 号引脚设置为按键的输入引脚，将其设置为外部中断模式。



接着点开 GPIO 标签页，对引脚进行如下设置，将 GPIO 模式设置为升降沿触发的外部中

断，上下拉电阻设置为上拉电阻，最后设置用户标签为 KEY。

Pin N...	Signal on ...	GPIO out...	GPIO mode	GPIO Pull...	Maximum ...	User Label	Modified
PA0-WK...	n/a	n/a	External I...	Pull-up	n/a	KEY	<input checked="" type="checkbox"/>
PH10	n/a	High	Output Pu...	Pull-up	Low	LED_B	<input checked="" type="checkbox"/>
PH11	n/a	High	Output Pu...	Pull-up	Low	LED_G	<input checked="" type="checkbox"/>
PH12	n/a	High	Output Pu...	Pull-up	Low	LED_R	<input checked="" type="checkbox"/>

PA0-WKUP Configuration :

- GPIO mode: External Interrupt Mode with Rising/Falling edge trigger detection
- GPIO Pull-up/Pull-do...: Pull-up
- User Label: KEY

外部中断一共有三种触发方式：上升沿触发，下降沿触发和上下沿均触发，其异同可见下表：

触发方式	cubeMX 中的名称	触发条件
上升沿触发	External Interrupt Mode with Rising edge trigger detection	外部中断触发引脚上的电平从低电平跳变到高电平
下降沿触发	External Interrupt Mode with Falling edge trigger detection	外部中断触发引脚上的电平从高电平跳变到低电平
上下边沿均触发	External Interrupt Mode with Rising/Falling edge trigger detection	外部中断触发引脚上有电平跳变

在 NVIC 标签页下，可以看到外部中断已经开启。

NVIC Interrupt Table	Enabled	Preemption Prio...	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
Pre-fetch fault, memory access fault	<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"/>	0	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 line0 interrupt	<input checked="" type="checkbox"/>	0	0
FPU global interrupt	<input type="checkbox"/>	0	0

6.4.2 HAL_GPIO_ReadPin 函数介绍

HAL 库提供了读取引脚上的电平的函数 `HAL_GPIO_ReadPin`。该函数说明如下：

GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)	
函数名	<code>HAL_GPIO_ReadPin</code>
函数作用	返回引脚电平
返回值	<code>GPIO_PinState</code> , 如果是高电平则返回 <code>GPIO_PIN_SET</code> (对应为 1) , 如果是低电平则返回 <code>GPIO_PIN_RESET</code> (对应为 0)
参数 1: <code>GPIOx</code>	对应 GPIO 总线, 其中 x 可以是 A...I。 例如 PH10, 则输入 GPIOH
参数 2: <code>GPIO_Pin</code>	对应引脚数。可以是 0-15。 例如 PH10, 则输入 <code>GPIO_PIN_10</code>

6.4.3 中断回调函数介绍

每当产生外部中断时, 程序首先会进入外部中断服务函数。在 `stm32f4xx_it.c` 中, 可以找到函数 `EXTI0_IRQHandler`, 它通过调用函数 `HAL_GPIO_EXTI_IRQHandler` 对中断类型进行

判断，并对涉及中断的寄存器进行处理，在处理完成后，它将调用中断回调函数 `HAL_GPIO_EXTI_Callback`，在中断回调函数中编写在此次中断中需要执行的功能。

6.4.4 程序中的前后台

在本次实验中，发现主循环和中断回调函数中都有代码。这是一个非常典型的以前后台模式组织的工程。什么是前后台模式呢？想象一下一个餐厅的运作模式，餐厅往往分为前台的叫餐员和后台的大厨，前台只有在来了客人，或者后台做好了一道菜时才会工作，而后厨则一直在忙着做菜，只有前台来了新的单子或者已经有菜做好了才会停下一会手中的活。

在单片机中，中断就是前台，而循环就是后台，中断只在中断源产生时才会进行相应的处理，而循环则一直保持工作，只有被中断打断时才会暂停。前后台程序的异同可以参见下表：

	前台程序	后台程序
运行方式	中断	循环
处理的任务类型	突发型任务	重复型任务
任务的特点	任务轻，要求响应及时	任务重，稳定执行

编写前后台程序时，需要注意尽量避免在前台程序中执行过长或者过于耗时的代码，让前台程序能够尽快执行完毕，以保证其能够实时响应突发的事件，比较繁杂和耗时的任务一般放在后台程序中处理。

前后台模式可以帮助我们提高单片机的时间利用率，从而组织起比较复杂的工程。

6.4.5 程序流程

本节课的程序中前后台任务各自承担的任务为：

前台程序	后台程序
记录按键翻转的状态 <code>rising_falling_flag</code>	执行处理工作，根据记录的翻转状态进行按键状态的判断

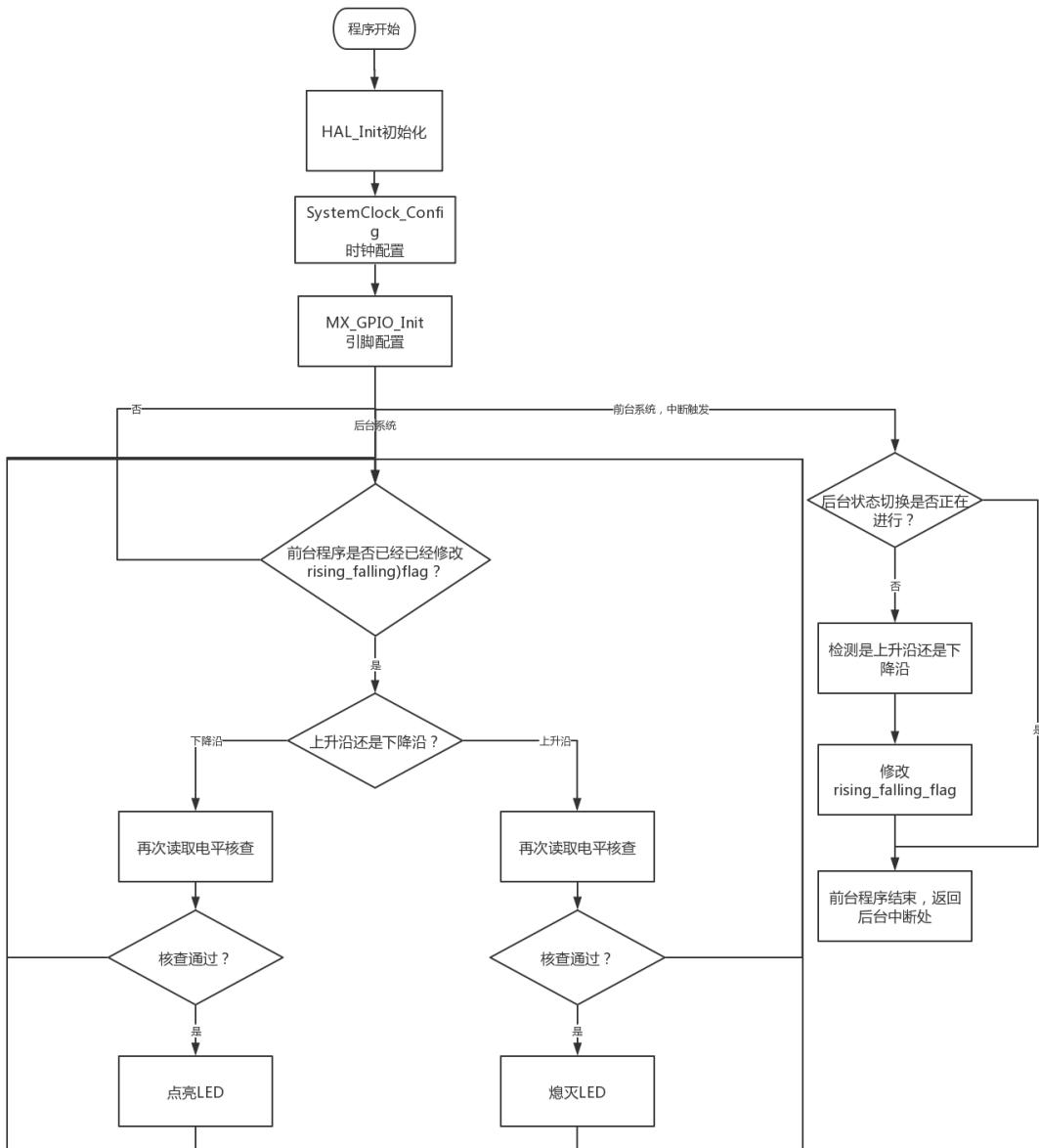
在主循环中，首先通过边沿检测标志 `rising_falling_flag` 来判断按键是处于按下还是松开的边沿，如果是下降的边沿 (`rising_falling_flag == GPIO_PIN_RESET`) 则将 LED 灯点亮，如果是上升的边沿 (`rising_falling_flag == GPIO_PIN_SET`) 则将 LED 灯熄灭。为了防止误触发，通过边沿检测的判断之后，程序还会再对电平进行一次读取，确认下降沿后跟随的是低电平或者上升沿后跟随的是高电平，如果不是则不切换 LED 状态。

在中断回调函数中，利用 `HAL_GPIO_ReadPin` 对 `rising_falling_flag` 进行赋值，从而判断

触发中断的是上升沿还是下降沿。

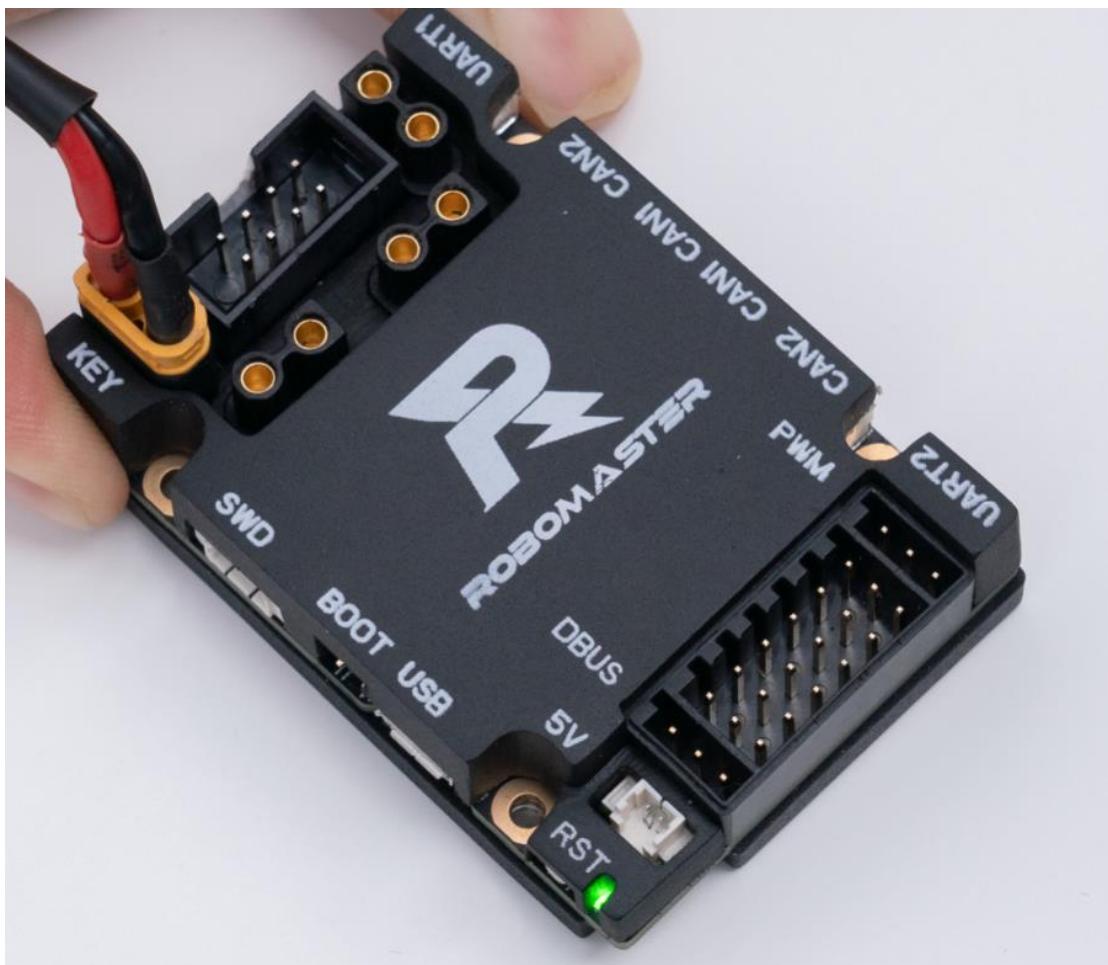
使用 `exit_flag` 来实现主循环和中断回调函数之间的互斥，保证中断处理函数中的功能（判断上升/下降沿）只在主循环完成判断之后进行，或者主循环的判断只在中断处理函数运行（即检测到了一次上升沿或者下降沿）之后再进行。

程序流程图如下：



6.4.6 效果展示

当按键按下时，绿灯亮起；当按键松开时，绿灯熄灭，如图所示。



6.5 课程总结

按键也是一种人机交互的操作，同时也可以设置成 `stm32` 的输入模式，可以用于设置某项功能的开启等作用。例如按下按键则开启 RoboMaster 机器人云台校准操作。外部中断也是 `stm32` 中重要的中断类型，常常用于传感器的数据处理，例如陀螺仪，加速度计，磁力计的数据准备中断，通知 `stm32` 已经有新的传感器数据产生。在哨兵机器人两侧往往安装有红外传感器，当哨兵机器人接近两端立柱时，也会产生外部中断，通知机器人及时掉头。

7. ADC 采样电池电压

7.1 知识要点

- ADC 原理
- 内部 VREFINT 电压介绍
- 电阻分压电路介绍
- ADC 的 cubeMX 配置

7.2 课程内容

本节课将介绍 STM32 的 ADC (Analog to Digital Converter) 模数转换功能。在单片机中传输的信号均为数字信号，通过离散的高低电平表示数字逻辑的 1 和 0，但是在现实的物理世界中只存在模拟信号，即连续变化的信号。将这些连续变化的信号——比如热，光，声音，速度通过各种传感器转化成连续的电信号，再通过 ADC 功能将连续的模拟信号转化成离散的数字信号给单片机进行处理。

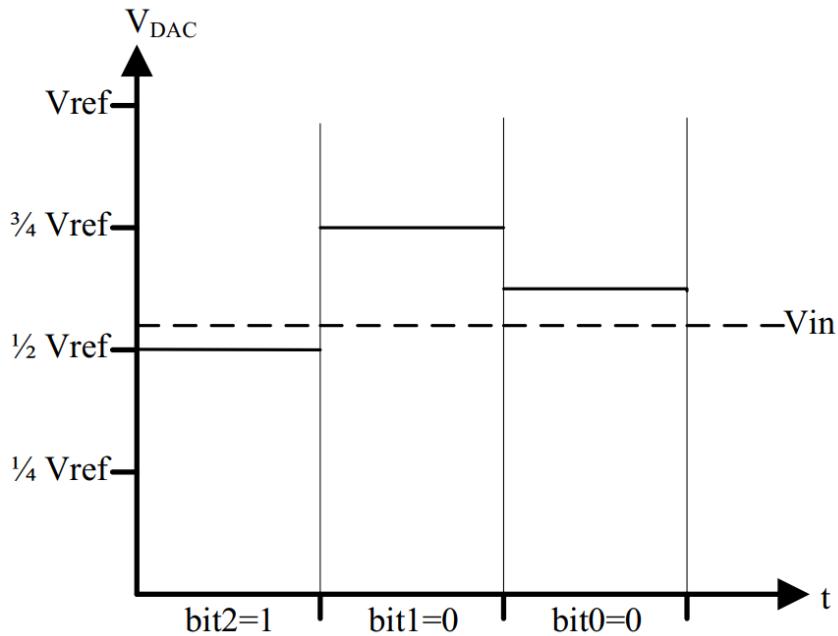
7.3 基础学习

7.3.1 ADC 原理介绍

一般 ADC 的工作流程为采样，比较，转换。

- 采样：是指对某一时刻的模拟电压进行采集，
- 比较：是指将采样的电压在比较电路中进行比较，
- 转换：是指将比较电路中结果转换成数字量。

stm32f4 采用 12 位逐次逼近型 ADC (SAR-ADC)。以下图为例，介绍 3 位 ADC 的比较过程

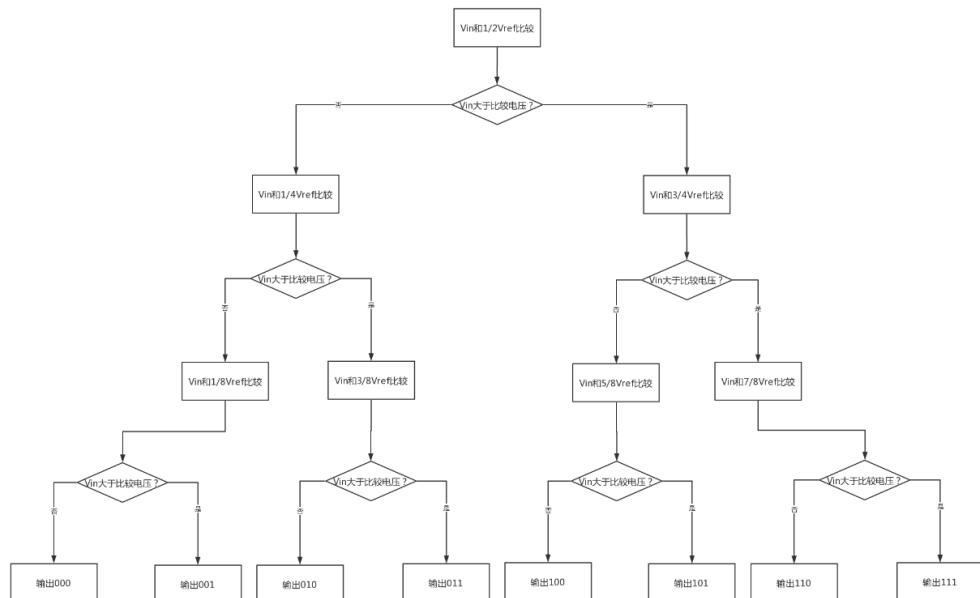


不同的位数分别赋予 $1/2$, $1/4$, $1/8$ 的权值, 模拟信号的采样值为 V_{in} ,

1. 与 $1/2V_{ref}$ 进行比较, V_{in} 大于 $1/2V_{ref}$, 则将第一位标记为 1,
2. 与 $3/4V_{ref}$ 进行比较, V_{in} 小于 $3/4V_{ref}$, 则将第二位标记为 0,
3. 与 $5/8V_{ref}$ 进行比较, V_{in} 小于 $5/8V_{ref}$, 则将第三位标记为 0。

图中的 V_{in} 通过这个三位的 ADC 后输出的结果为 100。转换的结果为 $1/2V_{ref}$, 通过这样逐次比较过程, 将采样取得的模拟电压和内部参考电压 V_{ref} 的加权值进行比较, 不同的位数赋予不同的权值。

如果输入为一个未知的电压, 完整的比较流程图如下:

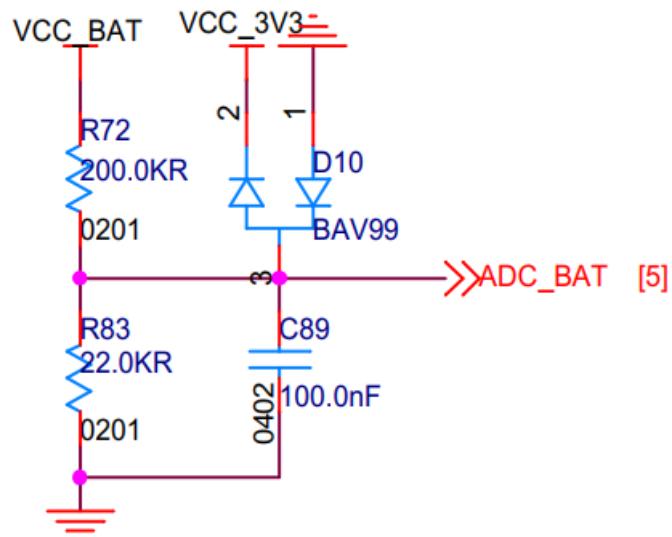


stm32 支持最高 12 位 ADC，一般 ADC 的位数越多则转换精度越高，但与此同时转换的速度也会变慢。

此外，stm32 内部有一个校准电压 VREFINT，电压为 1.2V，当供电电压不为 3.3V，可以使用内部的 vrefint 通道采集 1.2V 电压作为 Vref，以提高精度。

7.3.2 电阻分压电路介绍

下图为一个用于读取电池电压使用的电阻分压电路。由于电池提供的电源是 24V 的高电压，而单片机引脚的耐压只有 0-3.3V，所以需要通过分压电路进行处理，并使用滤波和二极管限幅电路进行保护。



电压检测

这里通过一个 $200\text{K}\Omega$ 和 $22\text{K}\Omega$ 的分压电路将 24V 电压进行分压

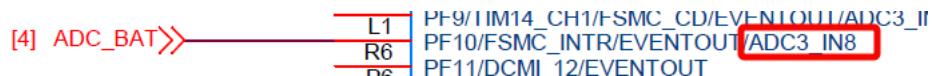
$$V_{out} = 24V * \frac{22\text{K}\Omega}{22\text{K}\Omega + 200\text{K}\Omega} = 2.38V$$

可以得到分压后的电压大约 2.38V，然后将该电压送至次级电路，在次级电路中，首先通过一个 100nF 的电容进行滤波，使输出的电压更加稳定，接着用二极管保护电路将电压限制在 3.3V 和 0V 之间，当电压大于 3.3V 时，二极管正向导通，电压被限制在 3.3V，当产生负压（电压小于 0V）时，二极管反向截止，输出点接地电压被限制在 0V。

7.4 程序学习

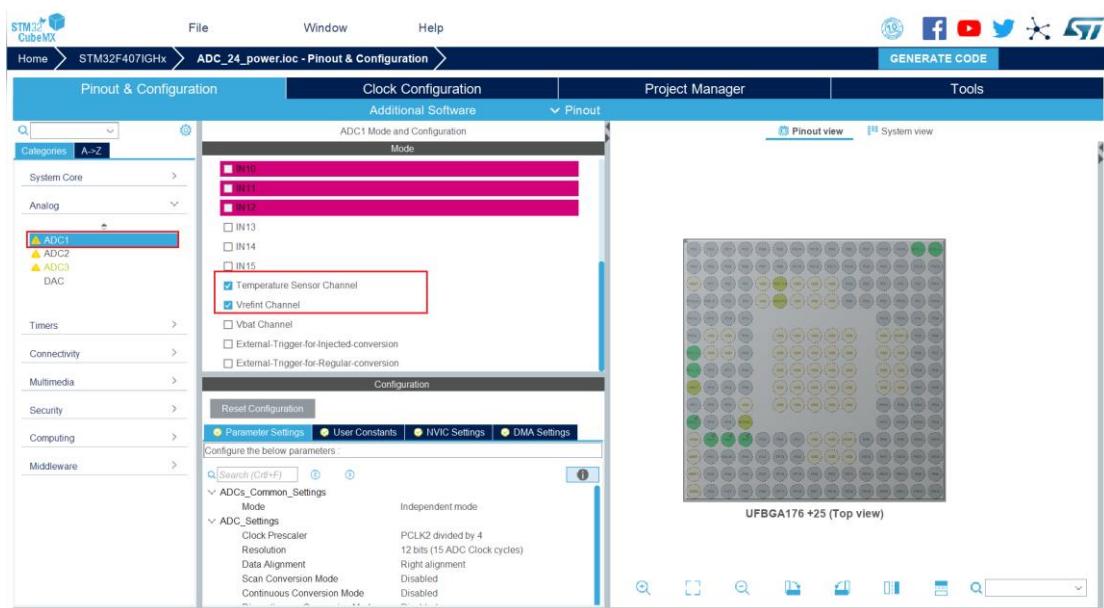
7.4.1 ADC 在 cubeMX 中的配置

其中电源 ADC 引脚为 PF10, 使用 ADC3 的通道 8, 原理图如图所示, 而 stm32 内部的 1.2V 校准电压 Vrefint 在 ADC1 中。



cubeMX 中的配置如下:

1. 开启 ADC1 和 ADC3 分别用于内部 1.2V 的 Vrefint 通道读取和电池电压 ADC3 的通道 8 读取。
2. 在 cubeMX 中开启 ADC1, 在设置中将 Vrefint Channel 勾选, 用于读取内部参考电压。ADC 在 cubeMX 中的设置如图采样频率设置为 PCLK2/4, 采样位数为 12 位, 数据设置为右对齐, 其余均保持默认。其中 Vrefint 在 stm32 内部完成, 没有对应引脚。



最终 ADC 配置如图所示。

<input type="text"/> Search (Ctrl+F)	<input type="button"/>	<input type="button"/>	<input type="button"/>
ADCs_Common_Settings			
Mode	Independent mode		
ADC_Settings			
Clock Prescaler	PCLK2 divided by 4		
Resolution	12 bits (15 ADC Clock cycles)		
Data Alignment	Right alignment		
Scan Conversion Mode	Disabled		
Continuous Conversion Mode	Disabled		
Discontinuous Conversion Mode	Disabled		
DMA Continuous Requests	Disabled		
End Of Conversion Selection	EOC flag at the end of single channel conversio		
ADC-Regular_ConversionMode			
Number Of Conversion	1		
External Trigger Conversion Source	Regular Conversion launched by software		
External Trigger Conversion Edge	None		
Rank	1		
ADC_Injected_ConversionMode			
Number Of Conversions	0		
WatchDog			
Enable Analog WatchDog Mode	<input type="checkbox"/>		

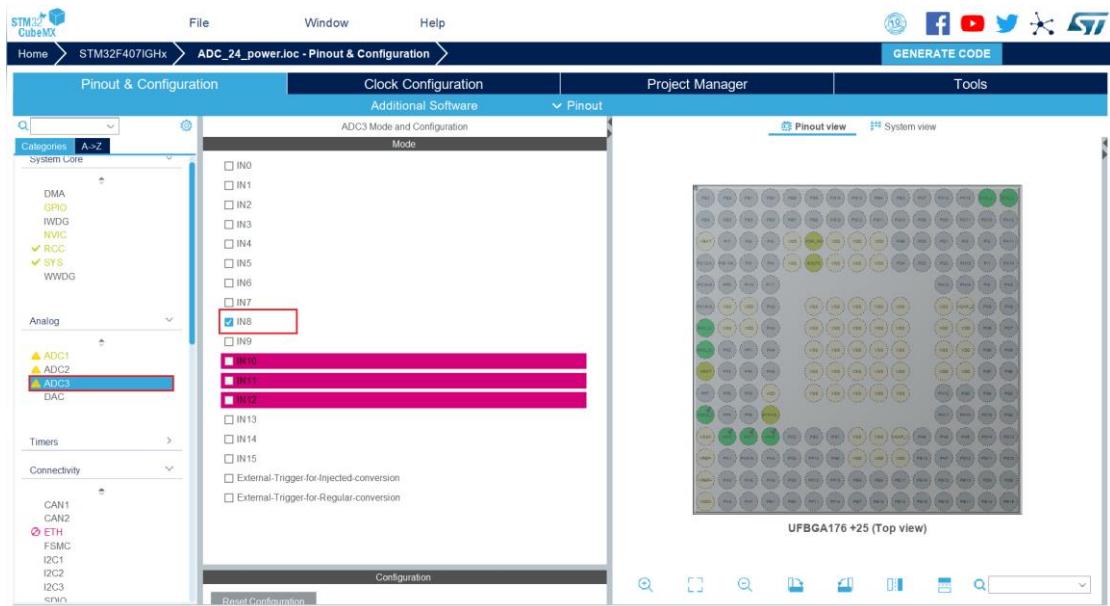
这里用一张表格表示 cubeMX 中 ADC 设置中的功能。

名称	功能
Clock Prescaler	设置采样时钟频率
Resolution	设置采样精度
Data Alignment	设置数据对齐方式
Scan Conversion Mode	扫描转换模式开启/关闭
Continuous Conversion Mode	连续转换模式开启/关闭
Discontinuous Conversion Mode	非连续转换模式开启/关闭
DMA Continuous Requests	DMA 连续启动开启/关闭
End of Conversion Selection	每个通道转换结束后发送 EOC 标志/所有通道转换结束后发送 EOC 标志

3. 在 cubeMX 中开启 ADC3, 并打开其 IN8 用于电池电压的读取, 其设置和 ADC1 一致。

可以看到引脚图像中 ADC3 对应的 PF10 变绿。

<input type="text"/> Search (Ctrl+F)				
ADCs_Common_Settings				
Mode	Independent mode			
ADC_Settings				
Clock Prescaler	PCLK2 divided by 4			
Resolution	12 bits (15 ADC Clock cycles)			
Data Alignment	Right alignment			
Scan Conversion Mode	Disabled			
Continuous Conversion Mode	Disabled			
Discontinuous Conversion Mode	Disabled			
DMA Continuous Requests	Disabled			
End Of Conversion Selection	EOC flag at the end of single channel conversio			
ADC-Regular_ConversionMode				
Number Of Conversion	1			
External Trigger Conversion Source	Regular Conversion launched by software			
External Trigger Conversion Edge	None			
Rank	1			
ADC_Injected_ConversionMode				
Number Of Conversions	0			
WatchDog				
Enable Analog WatchDog Mode	<input type="checkbox"/>			



7.4.2 内部 VREFINT 电压的使用

VREFINT 即 ADC 的内部参照电压 1.2V。通过将采样内部参照电压 1.2V 的 ADC 值和 Vref 的加权值进行比较，进而得到 ADC 的输出值。一般来说 STM32 的 ADC 采用 Vcc 作为 Vref，但为了防止 Vcc 存在波动较大导致 Vref 不稳定，进而导致采样值的比较结果不准确，STM32

可以通过内部已有的参照电压 VREFINT 来进行校准，接着以 VREFINT 为参照来比较 ADC 的采样值，从而获得比较高的精度，VREFINT 的电压为 1.2V。

通过一个函数对 1.2V 的电压进行多次采样，并计算其平均值，接着将其与 ADC 采出的数据值做对比，得到单位数字电压对应的模拟电压值 voltage_vrefint_proportion，其计算公式如下，设采样得到的数字值为 average_adc：

$$\text{average_adc} = \frac{\text{total_adc}}{200}$$

$$\text{voltage_vrefint_proportion} = \frac{1.2\text{v}}{\text{average_adc}} = 200 * 1.2/\text{total_adc}$$

```
void init_vrefint_reciprocal(void)
{
    uint8_t i = 0;
    uint32_t total_adc = 0;
    for(i = 0; i < 200; i++)
    {
        total_adc += adcx_get_chx_value(&hadc1, ADC_CHANNEL_VREFINT);
    }

    voltage_vrefint_proportion = 200 * 1.2f / total_adc;
}
```

7.4.3 ADC 采样相关函数介绍

HAL 库提供了以下与 ADC 采样有关的函数：

HAL_StatusTypeDef HAL_ADC_ConfigChannel(ADC_HandleTypeDef* hadc, ADC_ChannelConfTypeDef* sConfig)	
函数名	HAL_ADC_ConfigChannel
函数作用	设置 ADC 通道的各个属性值，包括转换通道，序列排序，采样时间等
返回值	HAL_StatusTypeDef，HAL 库定义的几种状态，如果成功使 ADC 开始工作，则返回 HAL_OK
参数 1	TIM_HandleTypeDef * hadc 即 ADC 的句柄指针，如果是 adc1 就输入

	&hadc1, adc2 就输入&adc2
参数 2	ADC_ChannelConfTypeDef* sConfig 即指向 ADC 设置的结构体指针。 我们先对 sConfig 结构体进行赋值，然后再将其指针作为参数输入函数

HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef* hadc)

函数名	HAL_ADC_Start
函数作用	开启 ADC 的采样
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态，如果成功使 ADC 开始工作，则返回 HAL_OK
参数	TIM_HandleTypeDef * hadc 即 ADC 的句柄指针，如果是 adc1 就输入&hadc1, adc2 就输入&adc2

HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc, uint32_t Timeout)

函数名	HAL_ADC_PollForConversion()
函数作用	等待 ADC 转换结束
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态，如果成功使 ADC 开始工作，则返回 HAL_OK
参数 1	TIM_HandleTypeDef * hadc 即 ADC 的句柄指针，如果是 adc1 就输入&hadc1, adc2 就输入&adc2
参数 2	uint32_t Timeout 等待的最大时间

uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc)

函数名	HAL_ADC_GetValue
函数作用	获取 ADC 值
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态，如果成功使 ADC 开始工作，则返回 HAL_OK

参数	TIM_HandleTypeDef * hadc 即 ADC 的句柄指针, 如果是 adc1 就输入&hadc1, adc2 就输入&adc2
----	--

7.4.4 程序流程

本程序中，首先对内部参考电压电压 VREFINT 进行 adc 采样将其作为校准值，在 init_vrefint_reciprocal 中，对 VREFINT 电压进行 200 次的采样，接着求其均值后使用 VREFINT 的电压值 1.2V 去除以该 ADC 采样得到的均值，算出 voltage_vrefint_proportion，后续 ADC 中采样到的电压值与 voltage_vrefint_proportion 相乘就可以计算出以内部参考电压做过校准的 ADC 值了。

```
void init_vrefint_reciprocal(void)
{
    uint8_t i = 0;
    uint32_t total_adc = 0;
    for(i = 0; i < 200; i++)
    {
        total_adc += adcx_get_chx_value(&hadc1, ADC_CHANNEL_VREFINT);
    }

    voltage_vrefint_proportion = 200 * 1.2f / total_adc;
}
```

接着通过 ADC 对经过了分压电路的电池电压值进行采样，将该采样结果与 voltage_vrefint_proportion 相乘，就得到了取值范围在 0-3.3V 之间的 ADC 采样值，由于这个采样值是分压后的结果，需要反向计算出电压的值。分压的电阻值为 $200\text{K}\Omega$ 和 $22\text{K}\Omega$ ，由于 $(22\text{K}\Omega + 200\text{K}\Omega) / 22\text{K}\Omega = 10.09$ ，乘以这个值之后就可以得到电池的电压值。

```
fp32 get_battery_voltage(void)
{
    fp32 voltage;
    uint16_t adcx = 0;

    adcx = adcx_get_chx_value(&hadc3, ADC_CHANNEL_8);
```

还可以通过 ADC 获得板载的温度传感器的温度值，同样是先经过 ADC 值进行采样，采样结束后，将 ADC 采样结果 `adc` 带入公式 $\text{temperate} = (\text{adc} - 0.76f) * 400.0f + 25.0f$ ，从而计算出温度值。

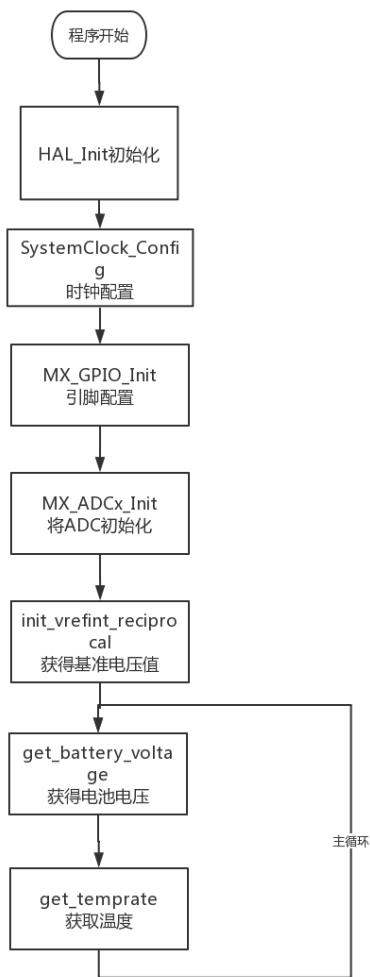
```
fp32 get_temprate(void)
{
    uint16_t adcx = 0;
    fp32 temperate;

    adcx = adcx_get_chx_value(&hadc1, ADC_CHANNEL_TEMPSENSOR);

    temperate = (fp32)adcx * voltage_vrefint_proportion;
    temperate = (temperate - 0.76f) * 400.0f + 25.0f;

    return temperate;
}
```

程序流程如下：



7.4.5 效果展示

使用电压表测量电源电压与程序测量的电压进行对比，如图所示。



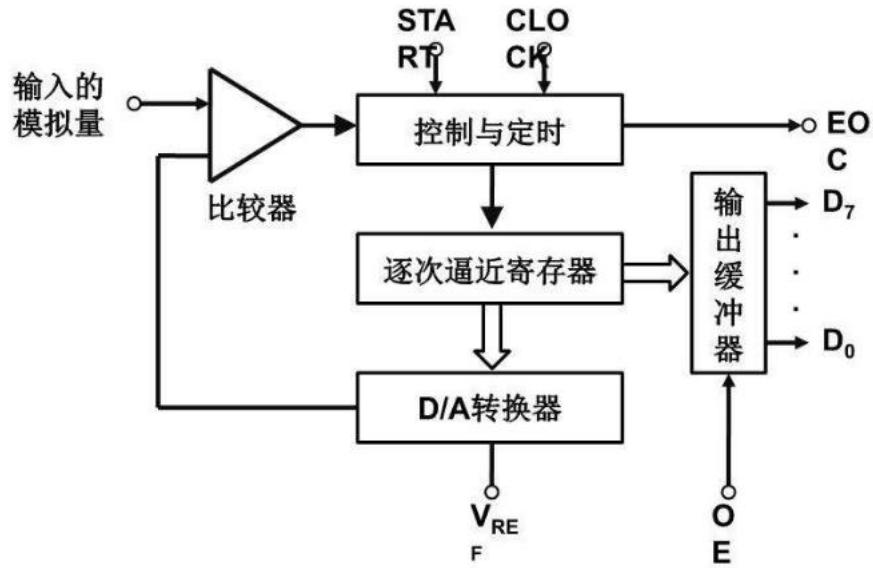
电压表测量图

Watch 1		
Name	Value	Type
voltage	25.5901451	float
temperature	27.6057968	float
hardware_version	2	unsigned char
<Enter expression>		

程序测量图

7.5 进阶学习

逐次型 ADC 是通过一位位进行比较得出转换值，其原理图如下所示。



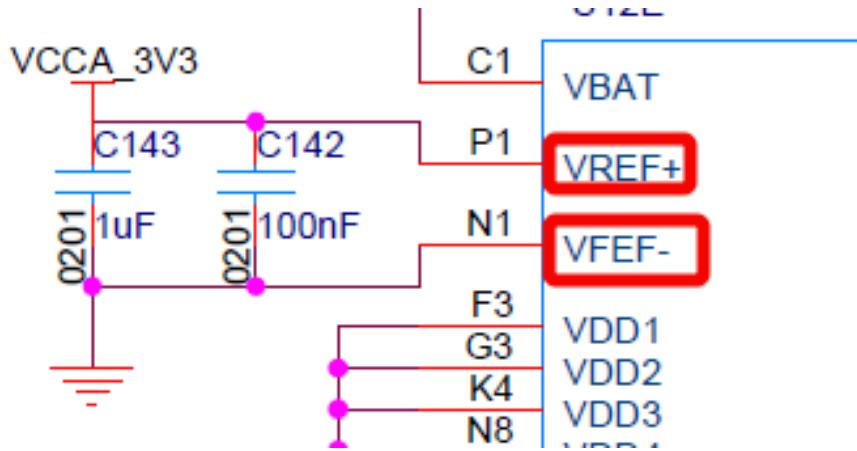
如上图所示，整个电路由比较器、D/A 转换器、缓冲寄存器和若干控制逻辑电路构成。作用如下所示：

- 比较器：用于输入电压值 D/A 转换器输出电压进行比较，当输入电压大于 D/A 转换器电压时，输出为 1，反之输出为 0；
- D/A 转换器：ADC 的逆向过程，将缓冲寄存器的记录数字量转换成模拟量。
- 缓冲寄存器：记录当前转换的数字量。

整个过程如下：

1. 将缓冲寄存器清零；
2. 将逐次逼近寄存器最高位置 1；
3. 把数字值送入 D/A 转换器，经 D/A 转换后的模拟量送入比较器，称为 V_o ；
4. V_o 与比较器的待转换的模拟量 V_i 比较，若 $V_o < V_i$ ，该位被保留，否则被清 0。
5. 再置寄存器次高位为 1，将寄存器中新的数字量送 D/A 转换器，
6. 输出的 V_o 再与 V_i 比较，若 $V_o < V_i$ ，该位被保留，否则被清 0。
7. 循环此过程，直到寄存器最低位，得到数字量的输出。

在这个过程中，D/A 转换器使用的电压为 stm32 的电源电压 V_{ref+} 和 V_{ref-} ，如图所示：



stm32 的标准电压为 3.3V。例如第一次转换时，输出为 $1/2V_{ref}$ 电压，即 1.65V。但由于外部供电电压不一定为 3.3V。故而为了在这种情况下提高 ADC 精度，stm32 内部有 $V_{REFINT1.2V}$ 稳定电压，可以使用 ADC 采样该电压来提高 ADC 的精度。

7.6 课程总结

ADC 是模拟量变成数字量的过程，单片机获取电压，电流传感器等模拟量的方法。通过 ADC 功能，我们能够获取各种传感器的模拟值，并将其转化为单片机可以处理的数字量，也可以获取电池的电压值，并将其通过电量形式显示出来。

8. 串口收发

8.1 知识要点

- 串口简介
- 串口在 cubeMX 的配置
- 串口接收中断与空闲中断
- 串口发送函数与中断函数
- APB 时钟计算串口波特率

8.2 课程内容

本节课将介绍在单片机中常用的串口功能。串口是一种在单片机，传感器，执行模块等诸多设备上常用的通讯接口，在比赛中，可以通过串口读取遥控器发送来的数据，也可以通过串口读取超声波等传感器的数据，也可以使用串口在单片机和运行计算机视觉的电脑之间进行通讯。

通过本节课的学习，将掌握如何通过 APB 时钟计算串口的波特率，串口在 cubeMX 中的配置方法，串口的接收中断与空闲中断功能，串口的发送函数与发送中断。

8.3 基础学习

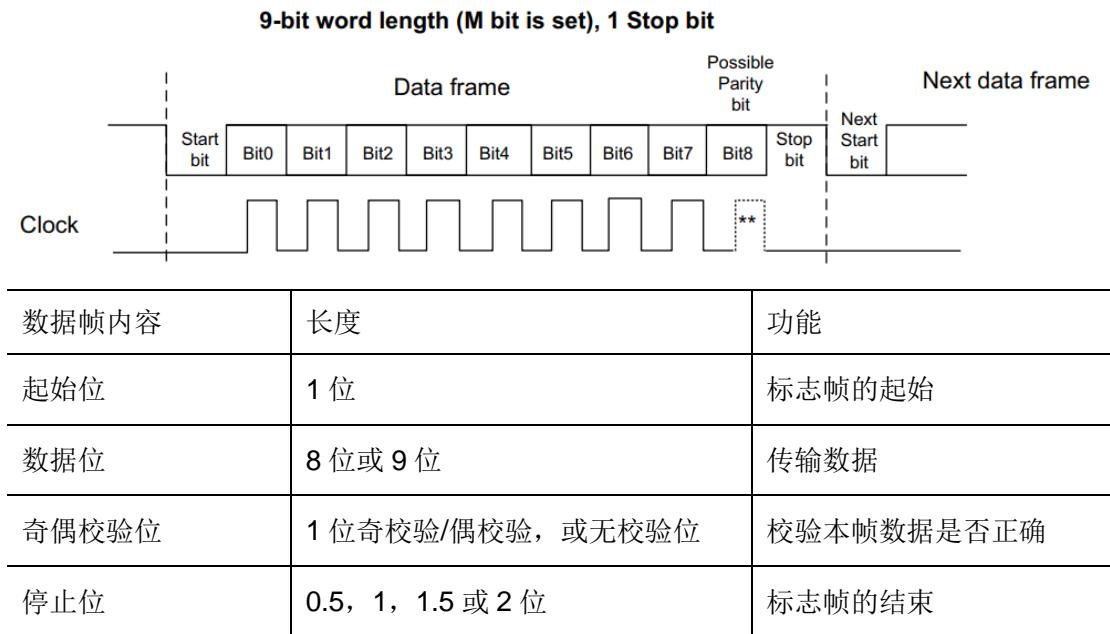
8.3.1 串口接收中断与空闲中断

串口全称为通用串行通信接口，是一种非常常用的通信接口。串行即以高低电平表示 1 和 0，将数据一位一位顺序发送给接收方。通用串行通信接口有着协议简单，易于控制的优点。

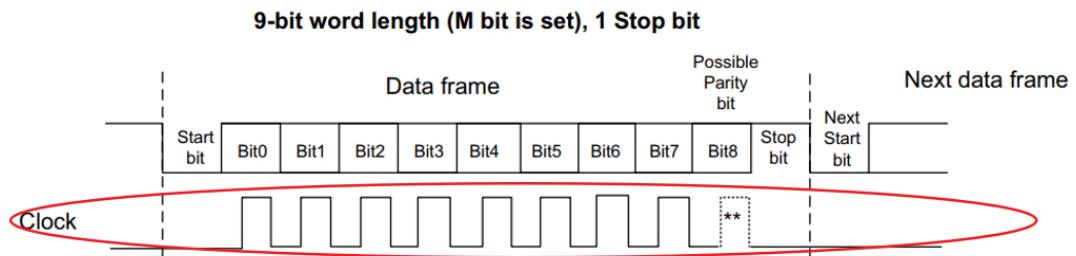
串口的通讯协议由开始位，数据位，校验位，结束位构成。一般以一个低电平作为一帧数据的起始，接着跟随 8 位或者 9 位数据位，之后为校验位，分为奇校验，偶校验和无校验，最后以一个先高后低的脉冲表示结束位，长度可以设置为 0.5, 1, 1.5 或 2 位长度。

奇偶校验位的原理是统计发送数据中高电平即'1'的奇偶，将结果记录在奇偶校验位中发送给接收方，接收方收到奇偶校验位后和自己收到的数据进行对比，如果奇偶性一致就接受这帧数据，否则认为这帧数据出错。

下图是一个 8 位数据位，1 位奇偶校验位，1 位结束位的串口数据帧。



一般进行串口通讯时，收发双方要保证遵守同样的协议才能正确的完成收发，除了协议要一致之外，还有一个非常重要的要素要保持一致，那就是通讯的速率，即波特率。波特率是指发送数据的速率，单位为波特每秒，一般串口常用的波特率有 115200, 38400, 9600 等。串口的波特率和总线时钟周期（clock）成倒数关系，即总线时钟周期越短，单位时间内发送的码元数量越多，串口波特率就越高。

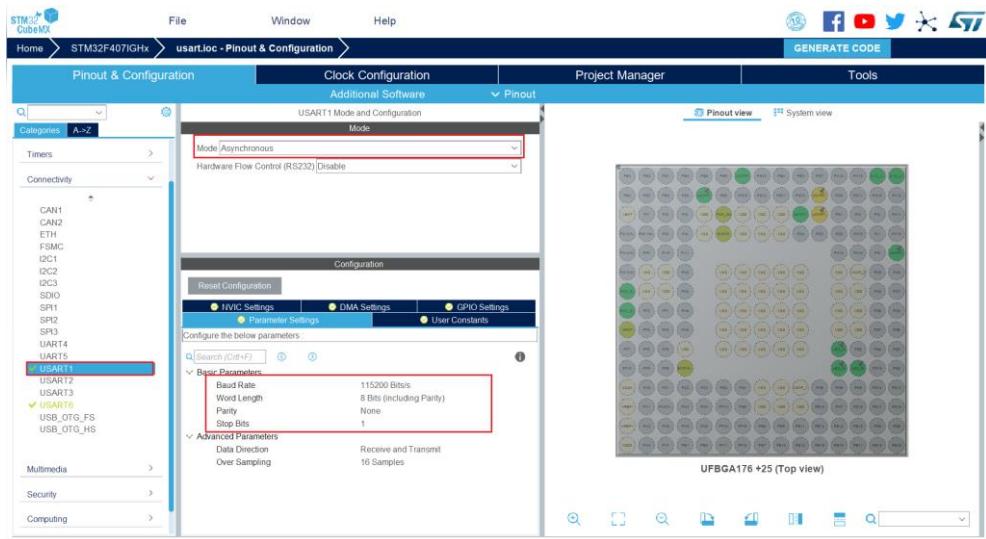


8.4 程序学习

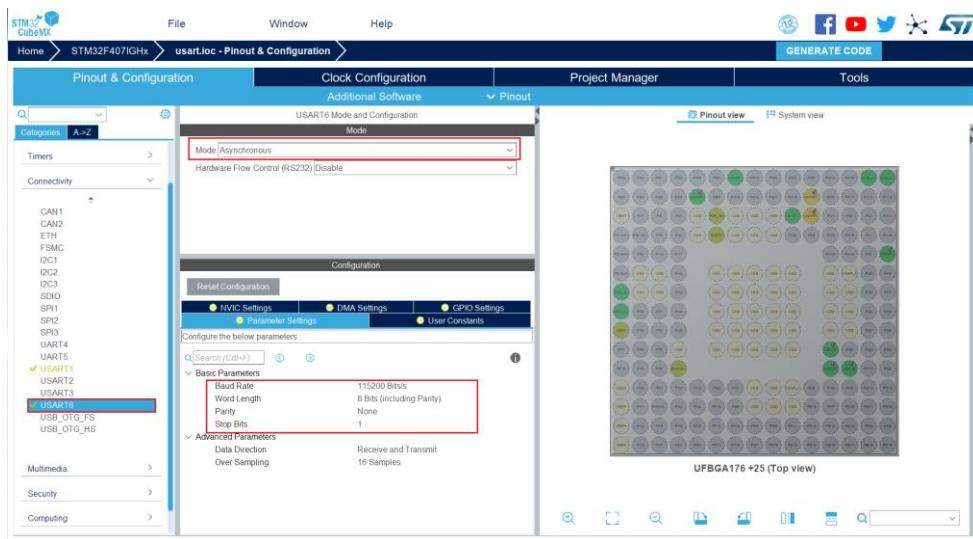
8.4.1 串口在 cubeMX 中配置

串口在 cubeMX 中的配置过程如下：

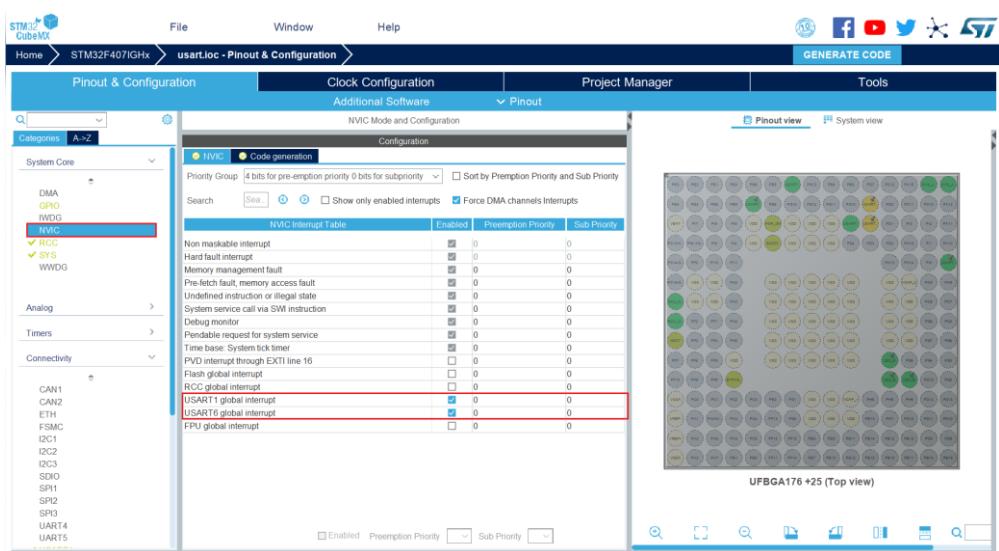
1. 首先在 Connectivity 标签页下将 USART1 打开，将其 Mode 设置为 Asynchronous 异步通讯方式。异步通讯即发送方和接收方间不依靠同步时钟信号的通讯方式。
2. 接着将其波特率设置为 115200，数据帧设置为 8 位数据位，无校验位，1 位停止位。



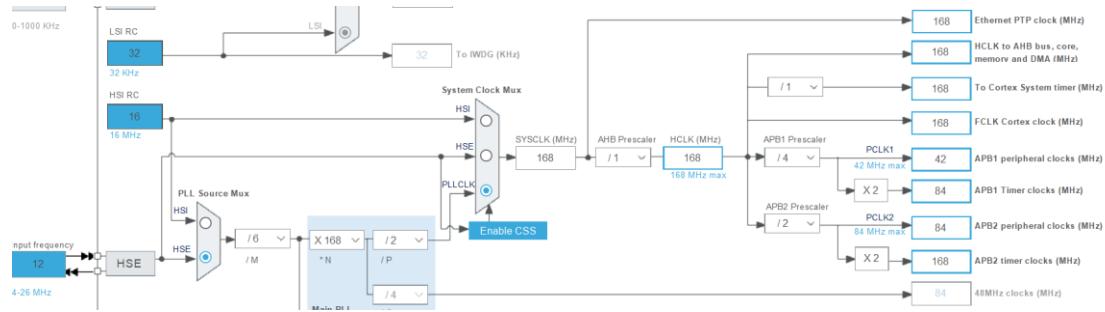
3. 同样地，打开 USART6，将其以和 USART1同样的方式进行设置。



4. 接着前往 NVIC 标签页下，开启 USART1 和 USART6 的中断。



工程时钟配置如图：



5. 点击 Generate Code 生成工程。

8.4.2 串口接收中断与空闲中断

本小节介绍串口的接收中断与空闲中断，这两种中断都是在串口进行接收时可能会发生的中断。

	接收中断	空闲中断
处理函数	USARTx_IRQHandler	USARTx_IRQHandler
回调函数	HAL_UART_RxCpltCallback	HAL 库没有提供
USART 状态寄存器中的位	UART_FLAG_RXNE	UART_FLAG_IDLE
触发条件	完成一帧数据的接收之后触发一次中断	串口接收完一帧数据后又过了一个字节的时间没有接收到任何数据

串口接收中断即每当串口完成一次接收之后触发一次中断。在 STM32 中相应的中断处理函数为 `USARTx_IRQHandler`，中断回调函数为 `HAL_UART_RxCpltCallback`。可以通过 USART 状态寄存器中的 `UART_FLAG_RXNE` 位判断 USART 是否发生了接收中断。

串口空闲中断即每当串口接收完一帧数据后又过了一个字节的时间没有接收到任何数据则触发一次中断，中断处理函数同样为 `USARTx_IRQHandler`，可以通过 USART 状态寄存器中的 `UART_FLAG_IDLE` 判断是否发生了空闲中断。

8.4.3 串口发送函数与中断函数

本小节将介绍串口的发送函数和中断函数。

HAL 库提供了串口发送函数 `HAL_UART_Transmit`，通过这个函数可以从指定的串口发送

数据。

HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout)	
函数名	HAL_UART_Transmit
函数作用	从指定的串口发送一段数据
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态, 如果成功发送本次数据, 则返回 HAL_OK
参数 1	UART_HandleTypeDef *huart 要进行发送的串口的句柄指针, 如串口 1 就输入&huart1, 串口 2 就输入&huart2
参数 2	uint8_t *pData 要发送的数据的首地址, 比如要发送 buf[]="Helloworld" 则输入 buf, 也可以直接输入要发送的字符串
参数 3	uint16_t Size 要发送的数据的大小, 即输入的字符串的长度, 也可以通过 sizeof 关键字获取数据大小
参数 4	uint32_t Timeout 发送超时时间, 如果发送时间超出该时间则取消本次发送

使能发送完成中断后, 每当完成一次串口发送, 串口会触发一次发送完成中断, 对应的中断回调函数为 HAL_UART_TxCpltCallback。

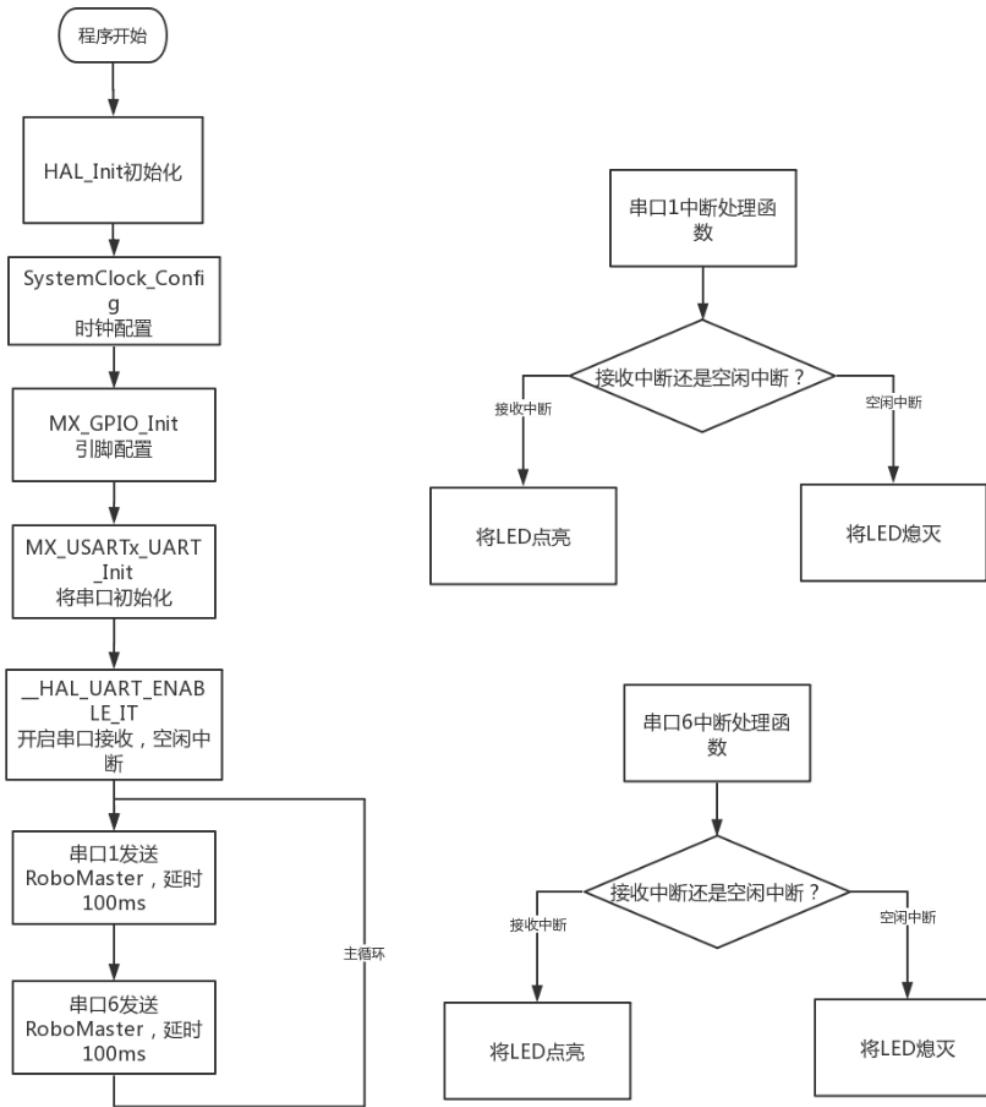
8.4.4 程序流程

本次示例代码中, 没有在中断回调函数中编写代码, 而是直接修改了中断处理函数 USARTx_IRQHandler。因此发生中断后会直接进入中断处理函数执行用户代码。

首先在初始化时初始化串口 1 和串口 6 的接收中断和空闲中断, 在主循环中通过 HAL_UART_Transmit 函数完成串口 1 和串口的 6 的数据发送, 发送内容为“RoboMaster\r\n”这一字符串。

当串口发生接收中断或者空闲中断时, 会进入 USARTx_IRQHandler 中断处理函数。在中断处理函数中通过串口的状态寄存器来判断产生中断的是接收中断还是空闲中断, 如果是接收中断则翻转红色 LED 的电平将其点亮, 如果是空闲中断则将其熄灭。

程序流程图如下:



8.4.5 效果展示

当串口 1 接收到数据的接收中断，LED 红灯会点亮，当串口 1 进入空闲中断，会将 LED 红灯熄灭；同样的当串口 6 接收到数据的接收中断，LED 绿灯会点亮，当串口 1 进入空闲中断，会将 LED 绿灯熄灭，接收效果如图所示。



绿灯亮起



红灯亮起

8.5 进阶学习

8.5.1 APB 时钟计算串口波特率

本小节将学习如何通过 APB 时钟计算串口的波特率。计算串口的波特率时首先要通过在定时器章节介绍的方法，通过 datasheet 查找到使用的串口对应的总线是 APB1 还是 APB2，然后再在 cube 中找到对应总线的速率。

在获取总线速率后，可以通过 USART_BRR 寄存器中的 USARTDIV 值来计算串口的通讯速率。在该寄存器中，前 12 位用来表示 USARTDIV 的整数部分，后 4 位用来表示小数部分，小数部分乘以 16 后取整，然后存储在后 4 位中。

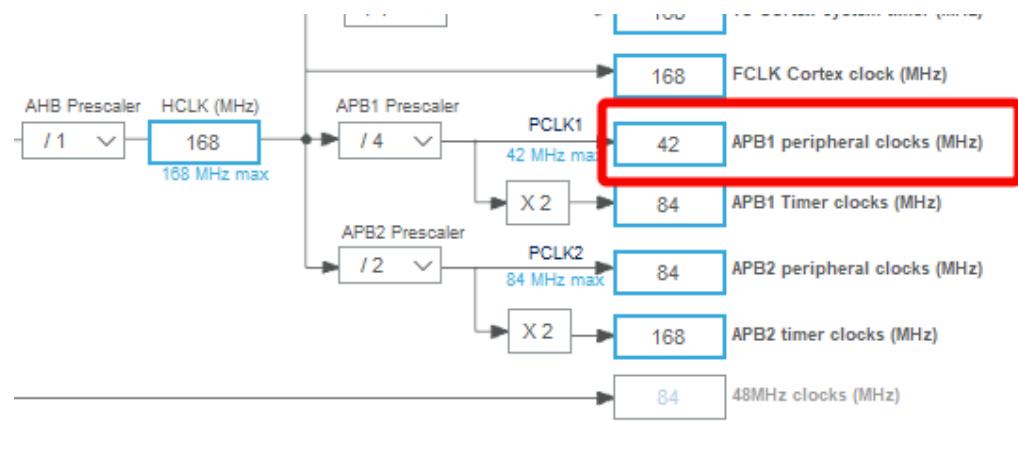
使用 USARTDIV 计算波特率的公式为：

$$\text{Tx / Rx 波特率} = f_{\text{PCLKx}} / (16 * \text{USARTDIV})$$

在数据手册中查出 USART1 连接在 APB1 总线上，APB1 速率 PCLK1 为 42MHz。而期望的 USART1 的串口通讯速率为 115200。

1. 通过上述的波特率计算公式，计算出 USARTDIV 值为 $42\text{MHz}/115200/16=22.786$ ；
2. 将整数 22 转化为 16 进制 $0x16$ ，小数 $0.786*16=12.576$ （取整 12）用 16 进制表示为 $0x0C$ ，所以存储在 USART_BRR 中的值最终为 $0x16C$ 。

由于这里采取了一些取整和舍弃尾数的近似手段，所以通过寄存器产生的波特率和最终期望的波特率之间是存在误差的，一般较小的误差不会影响最终的串口通讯，但是如果通讯时出现问题的话，要记得检查是否与通讯速率的实际值与理想值相差较大有关。



8.6 课程总结

本节课学习了 STM32 的串口功能，串口是一种在诸多设备上使用的通讯接口，通过串口通

讯，我们才能够在这些设备中传递信息。在比赛中，可以通过串口读取遥控器发送来的数据，也可以通过串口读取超声波等传感器的数据，也可以使用串口在单片机和运算自瞄程序的电脑之间进行通讯。

9. 串口打印遥控器数据

9.1 知识要点

- DMA 功能介绍
- 串口 DMA 接收与发送配置
- DBUS 协议介绍
- printf 函数实现
- 串口的 DMA 接收与发送

9.2 课程内容

本节课程中将介绍 STM32 串口的 DMA 功能, DMA 是在使用串口进行通讯时常用的一个功能, 使用该功能能够完成串口和内存之间直接的数据传送, 而不需要 CPU 进行控制, 从而节约 CPU 的处理时间。

通过实验的方式学习如何通过 DMA 功能读取遥控器的数据, 接着将学习如何在 STM32 上实现用 DMA 进行串口输出的 printf 函数, 并使用其将遥控器的数据传输到串口工具。

9.3 基础学习

9.3.1 DMA 功能介绍

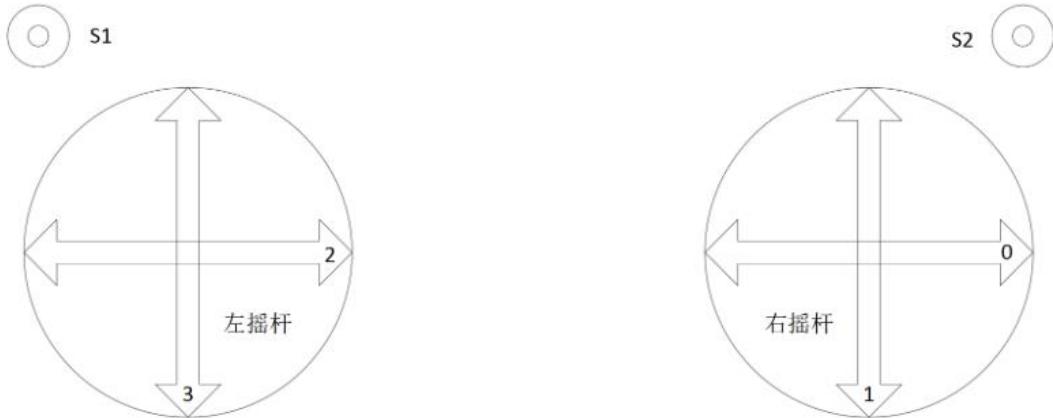
DMA 全称为 Direct Memory Access (直接存储器访问), 当需要将外部设备发来的数据存储在存储器中时, 如果不使用 DMA 方式则首先需要将外部设备数据先读入 CPU 中, 再由 CPU 将数据存储到存储器中, 如果数据量很大的话, 那么将会占用大量的 CPU 时间, 而通过使用 DMA 控制器直接将外部设备数据送入存储器, 不需要占用 CPU。

STM32 中的许多通讯如 USART, SPI, IIC 都支持 DMA 方式进行数据的收发。

9.3.2 DBUS 协议介绍

遥控器和 stm32 之间采用 DBUS 协议进行通讯。DBUS 通讯协议和串口类似, DBUS 的传输速率为 100k bit/s, 数据长度为 8 位, 奇偶校验位为偶校验, 结束位 1 位。需要注意的是 DBUS 使用的电平标准和串口是相反的, 在 DBUS 协议中高电平表示 0, 低电平表示 1, 如果使用串口进行接收需要在接收电路上添加一个反相器。

使用 DBUS 接收遥控器的数据，一帧数据的长度为 18 字节，一共 144 位，根据遥控器的说明书可以查出各段数据的含义，从而进行数据拼接，完成遥控器的解码，如图所示。



域	通道 0	通道 1	通道 2	通道 3	S1	S2
偏移	0	11	22	33	44	46
长度(bit)	11	11	11	11	2	2
符号位	无	无	无	无	无	无
范围	最大值 1684 中间值 1024 最小值 364	最大值 3 中间值 1 最小值 1	最大值 3 中间值 1 最小值 1			
功能	无符号类型 遥控器通道 0	无符号类型 遥控器通道 1	无符号类型 遥控器通道 2	无符号类型 遥控器通道 3	遥控器发射机 S1 开关位	遥控器发射机 S2 开关位置

域	鼠标 X 轴	鼠标 Y 轴	鼠标 Z 轴	鼠标左键	鼠标右键	按键
偏移	48	64	80	96	104	112
长度	16	16	16	8	8	16
符号位	有	有	有	无	无	无

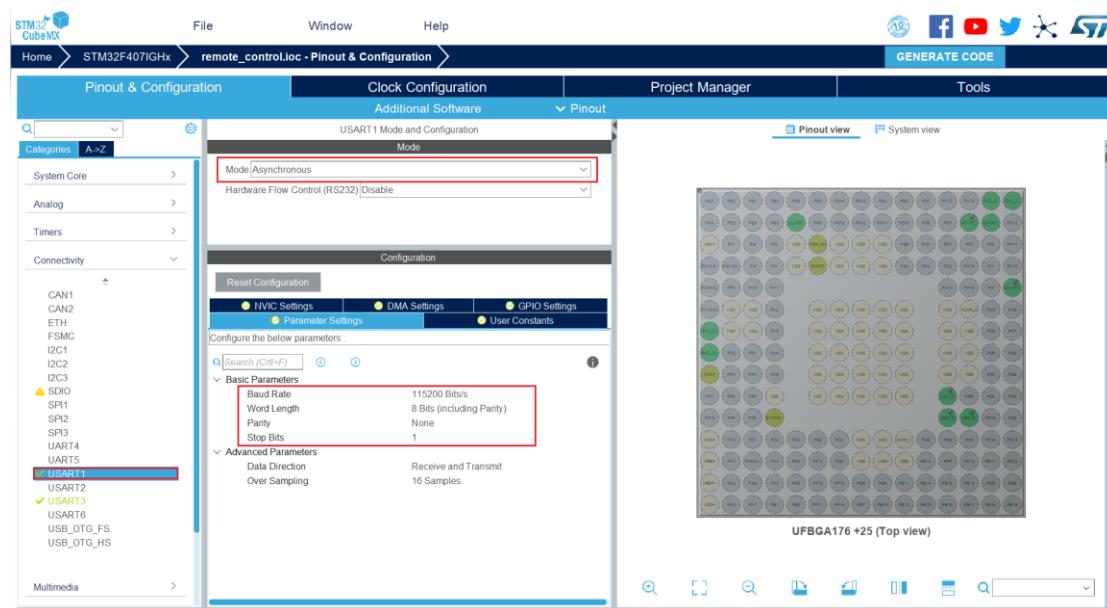
9.4 程序学习

9.4.1 串口发送的 DMA 配置

首先开启 USART1 和 USART3 并进行配置，其中 USART1 开启串口的 DMA 发送，用于数据发送 PC 的串口工具，USART3 开启串口的 DMA 接收，用于遥控器数据的接收；配置如下：

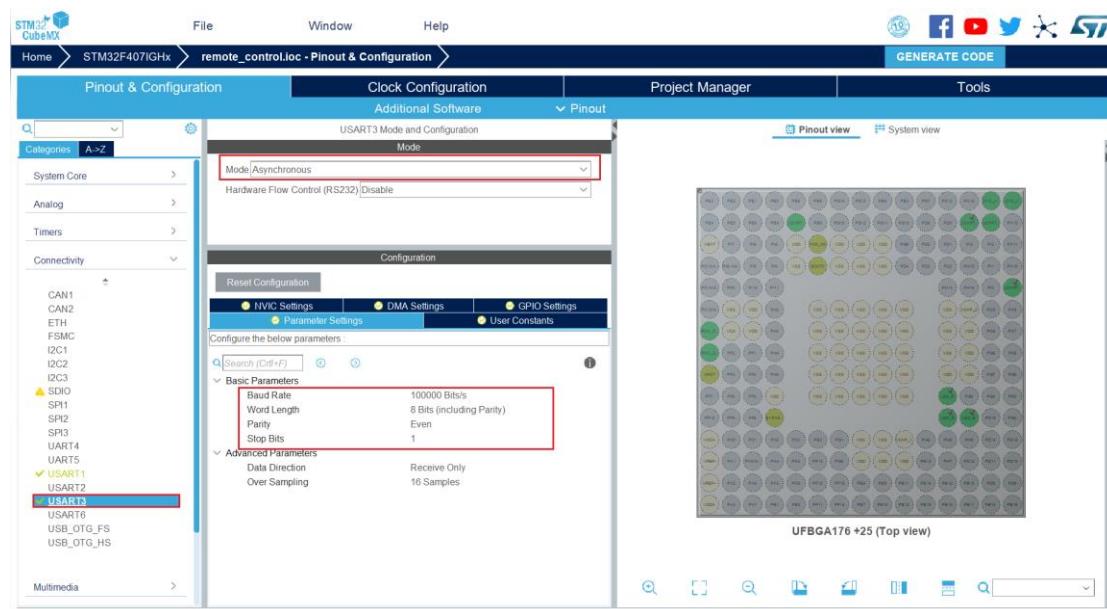
- 在 Connectivity 标签页下将 USART1 打开，将其 Mode 设置为 Asynchronous 异步通讯方式。异步通讯即发送方和接收方间不依靠同步时钟信号的通讯方式。

2. 将其波特率设置为 115200，数据帧设置为 8 位数据位，无校验位，1 位停止位。

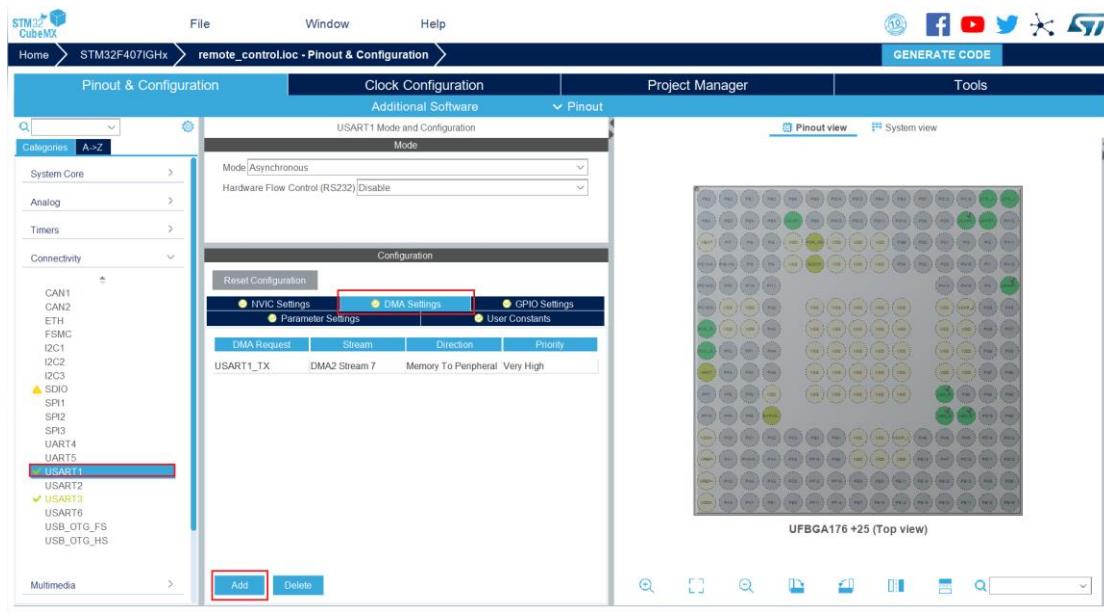


3. 在 Connectivity 标签页下将 USART3 打开,将其 Mode 设置为 Asynchronous 异步通讯方式。

4. 将其波特率设置为 100000，数据帧设置为 8 位数据位，无校验位，1 位停止位。



5. 接着分别开启 USART1 和 USART3 的 DMA 功能。点开 USART1 的设置页面，打开 DMA Settings 的标签页，点击 Add。



6. 在弹出的新条目中，将 DMA Request 选为 USART1_TX，数据从存储器流向外设，Priority 选为 Very High。

DMA Request	Stream	Direction	Priority
USART1_TX	DMA2 Stream 7	Memory To Peripheral	Very High

Add Delete

DMA Request Settings

Mode: Normal	Increment Address: <input type="checkbox"/>	Peripheral	Memory: <input checked="" type="checkbox"/>
Use Fifo: <input type="checkbox"/>	Threshold:	Data Width: Byte	Byte
		Burst Size:	

7. 同样，在 USART3 下找到 DMA Settings 标签呀，在 USART3 中将 DMA Request 选为 USART3_RX，数据从外设流向存储器，Priority 选为 Very High。

DMA Request	Stream	Direction	Priority
USART3_RX	DMA1 Stream 1	Peripheral To Memory	Very High

Add Delete

DMA Request Settings

Mode: Circular	Increment Address: <input type="checkbox"/>	Peripheral	Memory: <input checked="" type="checkbox"/>
Use Fifo: <input type="checkbox"/>	Threshold:	Data Width: Byte	Byte
		Burst Size:	

通过以上的设置，完成了 cubeMX 中对两个串口的 DMA 的设置。

9.4.2 printf 函数实现过程

利用 stdarg.h 下的 va_start 函数和 vsprintf 函数再配合串口的 DMA 发送功能来实现 C 语言中的 printf。通过以上函数的操作，将要发送的数据内容存储在 tx_buf 中，将要发送的数据长度存储在 len 变量中，接着将 tx_buf 的首地址和数据长度 len 传递给 DMA 发送函数，完成本次的 DMA 数据发送。

```
void usart_printf(const char *fmt,...)
{
    static uint8_t tx_buf[256] = {0};

    static va_list ap;

    static uint16_t len;

    va_start(ap, fmt);

    //return length of string
    //返回字符串长度

    len = vsprintf((char *)tx_buf, fmt, ap);

    va_end(ap);

    usart1_tx_dma_enable(tx_buf, len);
}
```

9.4.3 串口的 DMA 接收与发送配置

在本次实验中，使用 USART3 的 DMA 接收功能来接收遥控器数据。

通过函数 remote_control_init 进行 USART3 的 DMA 接收的初始化。在初始化时，使能 DMA 串口接收和空闲中断，配置当外设数据到达之后的存储的缓冲区，在这里开启了双缓冲区功能，每一帧 sbus 数据为 18 字节，而开启的双缓冲区总大小为 36 字节，这样可以避免 DMA 传输越界。

```

void RC_init(uint8_t *rx1_buf, uint8_t *rx2_buf, uint16_t dma_buf_num)

{
    //enable the DMA transfer for the receiver request
    //使能 DMA 串口接收
    SET_BIT(huart3.Instance->CR3, USART_CR3_DMAR);

    //enable idle interrupt
    //使能空闲中断
    __HAL_UART_ENABLE_IT(&huart3, UART_IT_IDLE);

    //disable DMA
    //失效 DMA
    __HAL_DMA_DISABLE(&hdma_usart3_rx);
    while(hdma_usart3_rx.Instance->CR & DMA_SxCR_EN)
    {
        __HAL_DMA_DISABLE(&hdma_usart3_rx);
    }

    hdma_usart3_rx.Instance->PAR = (uint32_t) & (USART3->DR);
    //memory buffer 1
    //内存缓冲区 1
    hdma_usart3_rx.Instance->M0AR = (uint32_t)(rx1_buf);
    //memory buffer 2
    //内存缓冲区 2
    hdma_usart3_rx.Instance->M1AR = (uint32_t)(rx2_buf);
    //data length
    //数据长度
    hdma_usart3_rx.Instance->NDTR = dma_buf_num;
    //enable double memory buffer
    //使能双缓冲区
}

```

```

SET_BIT(hdma_usart3_rx.Instance->CR, DMA_SxCR_DBM);

//enable DMA
//使能 DMA
__HAL_DMA_ENABLE(&hdma_usart3_rx);

}

```

在完成初始化之后，每当 USART3 产生空闲中断时就会进入 USART3_IRQHandler 进行处理，在 USART3_IRQHandler 中，进行寄存器中断标志位的处理，然后判断进行接收的缓冲区是 1 号缓冲区还是 2 号缓冲区，使用设定长度减去剩余长度，获取本次 DMA 得到的数据的长度，判断是否与一帧数据（18 字节）长度相等，如果相等则调用函数 sbus_to_rc 进行遥控器数据的解码。

```

void USART3_IRQHandler(void)
{
    if(huart3.Instance->SR & UART_FLAG_RXNE)//接收到数据
    {
        __HAL_UART_CLEAR_PEFLAG(&huart3);
    }
    else if(USART3->SR & UART_FLAG_IDLE)
    {
        static uint16_t this_time_rx_len = 0;

        __HAL_UART_CLEAR_PEFLAG(&huart3);

        if ((hdma_usart3_rx.Instance->CR & DMA_SxCR_CT) == RESET)
        {
            /* Current memory buffer used is Memory 0 */

            //disable DMA
            //失效 DMA
            __HAL_DMA_DISABLE(&hdma_usart3_rx);
        }
    }
}

```

```
//get receive data length, length = set_data_length - remain_length  
//获取接收数据长度,长度 = 设定长度 - 剩余长度  
this_time_rx_len = SBUS_RX_BUF_NUM - hdma_usart3_rx.Instance->NDTR;  
  
//reset set_data_lenght  
//重新设定数据长度  
hdma_usart3_rx.Instance->NDTR = SBUS_RX_BUF_NUM;  
  
//set memory buffer 1  
//设定缓冲区 1  
hdma_usart3_rx.Instance->CR |= DMA_SxCR_CT;  
  
//enable DMA  
//使能 DMA  
__HAL_DMA_ENABLE(&hdma_usart3_rx);  
  
if(this_time_rx_len == RC_FRAME_LENGTH)  
{  
    sbus_to_rc(sbus_rx_buf[0], &rc_ctrl);  
}  
}  
else  
{  
    /* Current memory buffer used is Memory 1 */  
    //disable DMA  
    //失效 DMA  
    __HAL_DMA_DISABLE(&hdma_usart3_rx);  
  
    //get receive data length, length = set_data_length - remain_length
```

```
//获取接收数据长度,长度 = 设定长度 - 剩余长度

this_time_rx_len = SBUS_RX_BUF_NUM - hdma_usart3_rx.Instance->NDTR;

//reset set_data_lenght

//重新设定数据长度

hdma_usart3_rx.Instance->NDTR = SBUS_RX_BUF_NUM;

//set memory buffer 0

//设定缓冲区 0

DMA1_Stream1->CR &= ~(DMA_SxCR_CT);

//enable DMA

//使能 DMA

__HAL_DMA_ENABLE(&hdma_usart3_rx);

if(this_time_rx_len == RC_FRAME_LENGTH)

{

    //处理遥控器数据

    sbus_to_rc(sbus_rx_buf[1], &rc_ctrl);

}

}

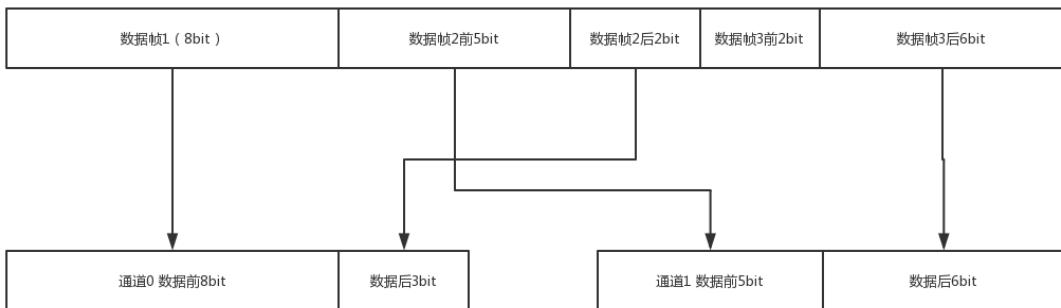
}

}
```

遥控器数据处理函数 `sbus_to_rc` 的功能是将通过 `DMA` 获取到的原始数据，按照遥控器的数据协议拼接成完整的遥控器数据，以通道 0 的数据为例，从遥控器的用户手册中查到通道 0 的长度为 11bit，偏移为 0。

域	通道 0
偏移	0
長度(bit)	11
符号位	无
范围	最大值 1684 中间值 1024 最小值 364
功能	无符号类型 遥控器通道 0

这说明如果想要获取通道 0 的数据就需要将第一帧的 8bit 数据和第二帧数据的后三 bit 数据拼接，如果想要获取通道 1 的数据就将第二帧数据的前 5bit 和第三帧数据的后 6bit 数据进行拼接，不断通过拼接就可以获得所有数据帧，拼接过程的示意图如下：



解码函数 `sbus_to_rc` 通过位运算的方式完成上述的数据拼接工作，十六进制数 `0x07ff` 的二进制是 `0b0000 0111 1111 1111`，也就是 11 位的 1，和 `0x07ff` 进行与运算相当于截取出 11 位的数据。

通道 0 的数据获取：首先将数据帧 1 和左移 8 位的数据帧 2 进行或运算，拼接出 16 位的数据，前 8 位为数据帧 2，后 8 位为数据帧 1，再将其和 `0x07ff` 相与，截取 11 位，就获得了由数据帧 2 后 3 位和数据帧 1 拼接成的通道 0 数据。其过程示意图如下：



通过上述方式就可以获取遥控器各个通道和开关，以及键鼠的数据值。

```

static void sbus_to_rc(volatile const uint8_t *sbus_buf, RC_ctrl_t *rc_ctrl)
{
    if (sbus_buf == NULL || rc_ctrl == NULL)
    {
        return;
    }

    rc_ctrl->rc.ch[0] = (sbus_buf[0] | (sbus_buf[1] << 8)) & 0x07ff;           //!< Channel 0
    rc_ctrl->rc.ch[1] = ((sbus_buf[1] >> 3) | (sbus_buf[2] << 5)) & 0x07ff;       //!< Channel 1
    rc_ctrl->rc.ch[2] = ((sbus_buf[2] >> 6) | (sbus_buf[3] << 2) |
                           (sbus_buf[4] << 10)) &0x07ff;                                //!< Channel 2
    rc_ctrl->rc.ch[3] = ((sbus_buf[4] >> 1) | (sbus_buf[5] << 7)) & 0x07ff;       //!< Channel 3
    rc_ctrl->rc.s[0] = ((sbus_buf[5] >> 4) & 0x0003);                         //!< Switch left
    rc_ctrl->rc.s[1] = ((sbus_buf[5] >> 4) & 0x000C) >> 2;                   //!< Switch right
    rc_ctrl->mouse.x = sbus_buf[6] | (sbus_buf[7] << 8);                      //!< Mouse X axis
    rc_ctrl->mouse.y = sbus_buf[8] | (sbus_buf[9] << 8);                      //!< Mouse Y axis
    rc_ctrl->mouse.z = sbus_buf[10] | (sbus_buf[11] << 8);                     //!< Mouse Z axis
    rc_ctrl->mouse.press_l = sbus_buf[12];                                     //!< Mouse Left Is Press ?
    rc_ctrl->mouse.press_r = sbus_buf[13];                                     //!< Mouse Right Is Press ?
    rc_ctrl->key.v = sbus_buf[14] | (sbus_buf[15] << 8);                     //!< KeyBoard value
    rc_ctrl->rc.ch[4] = sbus_buf[16] | (sbus_buf[17] << 8);                   //NULL

    rc_ctrl->rc.ch[0] -= RC_CH_VALUE_OFFSET;
    rc_ctrl->rc.ch[1] -= RC_CH_VALUE_OFFSET;
    rc_ctrl->rc.ch[2] -= RC_CH_VALUE_OFFSET;
    rc_ctrl->rc.ch[3] -= RC_CH_VALUE_OFFSET;
    rc_ctrl->rc.ch[4] -= RC_CH_VALUE_OFFSET;
}

```

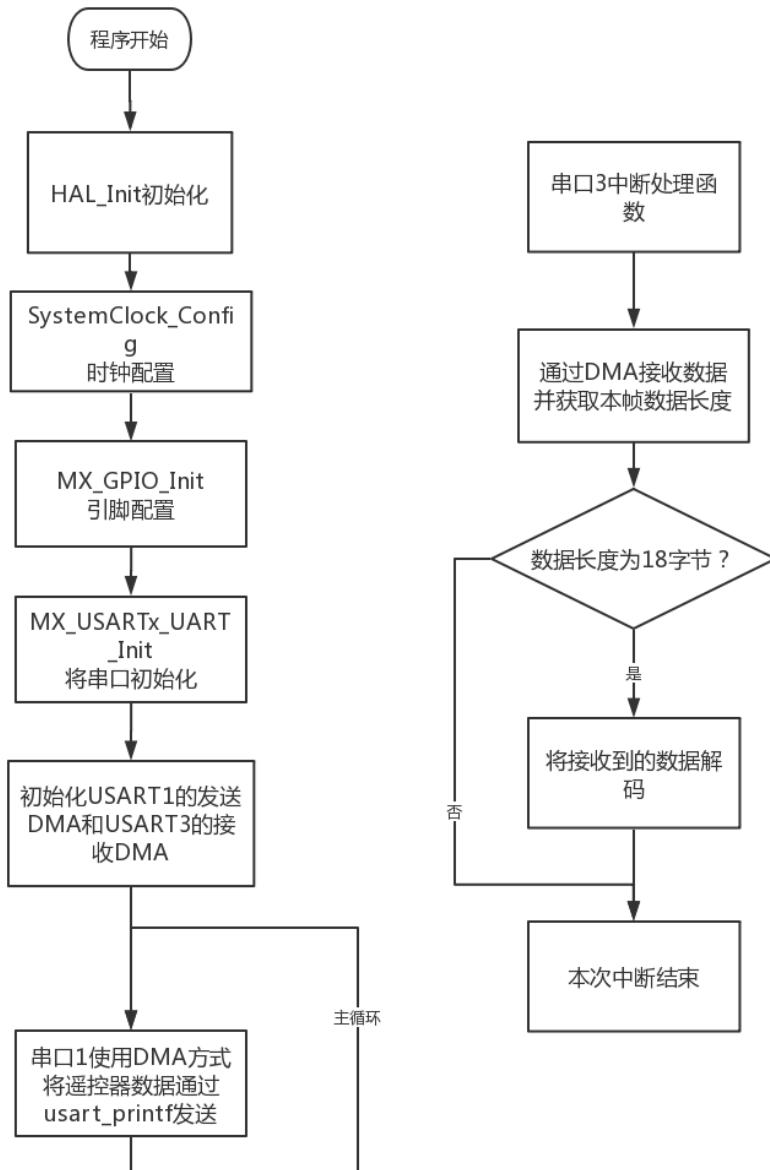
接着使用 **USART1** 用 DMA 方式进行发送，将接收到的遥控器数据发送出来。首先通过 **uart1_tx_dma_init** 函数进行 dma 发送的初始化，在主循环中，调用 **uart_print** 函数，将解码完成的遥控器数据从 **USART1** 使用 DMA 方式发送出来。

9.4.4 程序流程

本程序的流程为在初始化时进行 USART1 的 DMA 发送初始化和 USART3 的 DMA 接收初始化，接着在 USART3 的串口接收中断中使用 DMA 接收遥控器的数据，并使用解码函数将数据进行解码。

接着在主循环中调用串口实现的 `uart_printf` 函数，将解码完成的遥控器函数通过 USART1 的 DMA 发送功能发送出来。

程序流程图如下：



9.4.5 效果展示

遥控器接收机连接到开发板 C 型上的 DBUS 口，在 debug 窗口和串口工具上进行查看遥控

器通道值如图所示。



遥控器接线图

Name	Value	Type
rc_ctrl	0x20000260 &rc_ctrl	struct <untagged>
rc	0x20000260 &rc_ctrl	struct <untagged>
ch	0x20000260 &rc_ctrl	short[5]
[0]	-306	short
[1]	126	short
[2]	307	short
[3]	381	short
[4]	0	short
s	0x2000026A "□□"	char[2]
[0]	3	char
[1]	3	char
mouse	0x2000026C	struct <untagged>
x	0	short
y	0	short
z	0	short
press_l	0	unsigned char
press_r	0	unsigned char
key	0x20000274	struct <untagged>
v	0	unsigned short

程序调试遥控器数据图

```
*****  
ch0:-623  
ch1:-376  
ch2:312  
ch3:249  
ch4:0  
s1:3  
s2:3  
mouse_x:0  
mouse_y:0  
press_l:0  
press_r:0  
key:0  
*****
```

串口工具接收遥控器数据图

9.5 课程总结

本节课学习了串口的 DMA 发送接收功能。DMA 发送接收功能使得 CPU 能够高效的完成数据接收发送，此外本节课还学习了遥控器的接收和解码，以及使用串口实现 `printf` 功能。遥控器是 RM 机器人最重要的人机交互装置，用于机器人的控制输入。`Printf` 函数是标准化输出的函数，主要用于调试的功能。

10. Flash 读写

10.1 知识要点

- stm32 的 flash 介绍
- stm32 的 boot 功能
- flash 擦除介绍
- flash 写入与读取

10.2 课程内容

本节课程将学习 stm32 的 flash (闪存) 外设。flash 用于存放数据，在 RM 比赛中常常将机器人特征数据例(如云台电机中值)保存进 flash。在本课程内，学习 stm32 对 flash 进行数据的读写，将 RoboMaster 字符写入 flash 中，并且从 flash 中读取写入的字符串数据。

10.3 基础学习

10.3.1 stm32 的 flash 介绍

个人计算机都均有内存和硬盘(外存),开发板芯片 stm32 同样具有内存 192Kbytes 的 SRAM 和 1Mbytes 的外存 flash。Flash 和 SRAM 有如下特点。

FLASH	SRAM
容量较大	容量较小
读取和写入速度较慢	读取和写入速度较快
掉电不会丢失数据	掉电丢失数据
分扇区	不分扇区
写入需要先擦除	写入不需擦除
存储程序，长时间保存的数据等	变量等

对于 flash 的使用，需要了解以下两点：

1. 为了保护数据的安全性，flash 有专门的锁寄存器，每次要对 flash 页面进行修改时首先要通过锁寄存器对页面进行解锁，修改完成后要进行加锁。

- flash 是不支持在保存原有数据的情况下进行修改的，因此要改变 flash 页面数据时，需要对这个页面进行擦除，擦除之后再写入新的数据。

10.4 程序学习

10.4.1 flash 擦除函数介绍

本小节将介绍 flash 的擦除函数。HAL 库提供了 flash 的擦除函数 HAL_FLASHEx_Erase。

HAL_StatusTypeDef HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit, uint32_t *SectorError)	
函数名	HAL_FLASHEx_Erase
函数作用	擦除指定的 flash 页面
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态, 如果成功完成 flash 页面擦除, 则返回 HAL_OK, 失败会返回 HAL_ERROR, 超时会返回 HAL_TIMEOUT
参数 1	FLASH_EraseInitTypeDef *pEraseInit 擦除 flash 时使用的结构体指针, 需要创建一个 FLASH_EraseInitTypeDef 类型的结构体 flash_erase, 需要赋予这个结构体以下参数: Sector(要擦除的页面的首地址), TypeErase(擦除方式), VoltageRange(电压范围), NbSectors(待擦除页面数), 最后我们将&flash_erase 作为参数输入函数
参数 2	uint32_t *SectorError 如果本次 flash 擦除产生了错误, 则发生擦除错误的页面号存储在 SectorError 中。创建一个 uint32_t error, 将&error 输入即可, 如果擦除出错, 可以去读取 error 中的值确认出错的页面号。

10.4.2 flash 写入函数介绍

本小节将介绍 flash 的写入函数, HAL 库提供向 flash 中写入数据的函数 HAL_FLASH_Program

HAL_StatusTypeDef HAL_FLASH_Program(uint32_t TypeProgram, uint32_t Address, uint64_t Data)	
函数名	HAL_FLASH_Program

函数作用	以指定的方式，向 flash 中的一页写入数据
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态，如果成功完成向 flash 的数据写入，则返回 HAL_OK，失败会返回 HAL_ERROR，超时会返回 HAL_TIMEOUT
参数 1	uint32_t TypeProgram 选择写入的数据格式，可以选择 8 位字节 FLASH_TYPEPROGRAM_BYTE , 16 位半字 FLASH_TYPEPROGRAM_HALFWORD , 32 位字 FLASH_TYPEPROGRAM_WORD , 或者 64 位双字 FLASH_TYPEPROGRAM_DOUBLEWORD
参数 2	uint32_t Address 需要写入数据的地址
参数 3	uint64_t Data 需要写入的数据

10.4.3 flash 读取介绍

flash 的读取相对简单，可以通过 memcpy 函数进行读取

```
void flash_read(uint32_t address, uint32_t *buf, uint32_t len)
{
    memcpy(buf, (void*)address, len *4);
}
```

函数名	flash_read
函数作用	从 flash 读取数据
返回值	None
参数 1	Flash 地址
参数 2	读取后的存储变量地址

参数 3	字节长度
------	------

10.4.4 flash 相关操作函数

本小节将继续介绍 flash 的相关操作函数，主要是 flash 的解锁和加锁功能。

在原理部分介绍过，写入或擦除数据时需要为 flash 进行解锁和加锁。首先介绍 flash 的解锁函数 HAL_FLASH_Unlock。这个函数的功能其实非常简单，只是为锁寄存器写入特定的 KEY 值以完成解锁。

HAL_StatusTypeDef HAL_FLASH_Unlock(void)

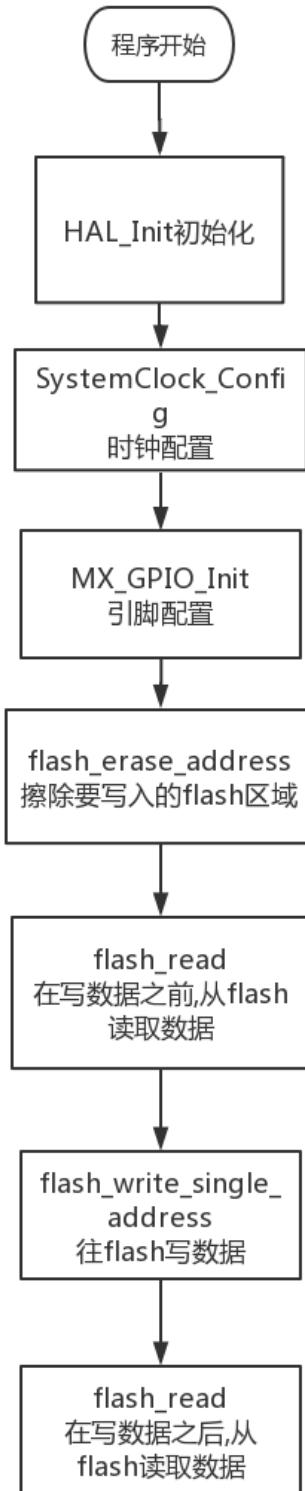
函数名	HAL_FLASH_Unlock
函数作用	为 flash 解锁，使用户可以修改 flash 内容
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态，如果成功完成 flash 的解锁，则返回 HAL_OK，失败会返回 HAL_ERROR

接着介绍 flash 加锁操作函数 HAL_FLASH_Lock，该函数的功能同样简单，即清楚已解锁的锁寄存器的 KEY 值，便完成了加锁功能。

HAL_StatusTypeDef HAL_FLASH_Lock(void)

函数名	HAL_FLASH_Lock
函数作用	为 flash 加锁，后续操作时只有先解锁才能对 flash 内容进行修改
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态，如果成功完成 flash 的加锁，则返回 HAL_OK，失败会返回 HAL_ERROR

10.4.5 程序流程



10.4.6 效果展示

先擦除 0x80E0000 开始的一页 flash，如图所示：

之后写入 RoboMaster 字符，如图所示：

从 flash 读取数据到变量中，如图所示：

Watch 1

Name	Value	Type
after_erase_data	0x2000002C after...	unsigned char[12]
[0]	0xFF ' '	unsigned char
[1]	0xFF ' '	unsigned char
[2]	0xFF ' '	unsigned char
[3]	0xFF ' '	unsigned char
[4]	0xFF ' '	unsigned char
[5]	0xFF ' '	unsigned char
[6]	0xFF ' '	unsigned char
[7]	0xFF ' '	unsigned char
[8]	0xFF ' '	unsigned char
[9]	0xFF ' '	unsigned char
[10]	0xFF ' '	unsigned char
[11]	0xFF ' '	unsigned char
write_data	0x20000000 write...	unsigned char[12]
after_write_data	0x20000038 after...	unsigned char[12]
[0]	0x52 'R'	unsigned char
[1]	0x6F 'o'	unsigned char
[2]	0x62 'b'	unsigned char
[3]	0x6F 'o'	unsigned char
[4]	0x4D 'M'	unsigned char
[5]	0x61 'a'	unsigned char
[6]	0x73 's'	unsigned char
[7]	0x74 't'	unsigned char
[8]	0x65 'e'	unsigned char
[9]	0x72 'r'	unsigned char
[10]	0x0D	unsigned char
[11]	0x0A	unsigned char
<Enter expression>		

10.5 进阶学习

10.5.1 flash 页分区

stm32f407IG 共有 12 个 flash 页分区，如下表所示。

	名称	地址	大小
主存储器	扇区 0	0x08000000 - 0x08003FFF	16 Kbytes
	扇区 1	0x08004000 - 0x08007FFF	16 Kbytes
	扇区 2	0x08008000 - 0x0800BFFF	16 Kbytes
	扇区 3	0x0800C000 - 0x0800FFFF	16 Kbytes
	扇区 4	0x08010000 - 0x0801FFFF	64 Kbytes
	扇区 5	0x08020000 - 0x0803FFFF	128 Kbytes
	扇区 6	0x08040000 - 0x0805FFFF	128 Kbytes
	扇区 7	0x08060000 - 0x0807FFFF	128 Kbytes
	扇区 8	0x08080000 - 0x0809FFFF	128 Kbytes
	扇区 9	0x080A0000 - 0x080BFFFF	128 Kbytes
	扇区 10	0x080C0000 - 0x080DFFFF	128 Kbytes
	扇区 11	0x080E0000 - 0x080FFFFFF	128 Kbytes

其中 **stm32** 上电时，会从 **0x08000000** 读取程序开始运行，故而在扇区 0 开始往往回存储程序，需要使用程序之后的扇区作为存储扇区。

10.5.2 boot 作用

boot 用于控制芯片在上电之后如何引导程序加载到芯片中开始执行。

STM32 的启动方式分为三种方式，采用哪一种方式启动可以通过芯片的 **boot0** 和 **boot1** 引脚进行设置。

第一种方式是最常用的方式，即从 **flash** 中读取烧录的程序，将其引导到 **STM32** 中执行。

第二种方式，芯片从一块特殊的系统存储区启动，在系统存储区中存有芯片生产厂商已经写好的 **bootloader** 程序，这个程序的功能是通过串口将程序读取到 **flash** 中去，再将 **flash** 中的程序引导到单片机中执行。有一些特殊的下载软件可以让我们通过串口进行 **STM32** 程序的下载，其原理就是使用了这种 **boot** 方式。使用这种 **boot** 的优点是不需要额外的下载器，缺点是下载速度比较慢。

第三种方式是从 **STM32** 内嵌的 **SRAM** 中启动，可以将一小段程序写入 **SRAM** 中用于调试，但是因为 **SRAM** 有着掉电丢失的特性，所以一般不会采用这种方式进行 **boot**，否则每次重

新给芯片上电就必须重新给 SRAM 写入一次程序。

三种方式的对比可见下表：

	方式 1	方式 2	方式 3
BOOT 引脚状态	BOOT0:GND BOOT1:GND or VCC	BOOT0:VCC BOOT1:GND	BOOT0:VCC BOOT1:VCC
启动区域	Flash 的用户程序存储区	Bootloader 程序存储区	内嵌 SRAM
功能	将用户通过下载器烧录的程序引导到单片机执行，是最常用的方式	通过串口将程序读取到 flash 中去，再将 flash 中程序引导到单片机中执行。	将 SRAM 中的程序引导到单片机执行，一般用于调试

10.6 课程总结

本节课介绍了 STM32 的 flash 原理以及其读写功能，flash 是一种常用的非易失存储区，既可以存储待运行的程序，也可以存储用户数据。Flash 可用于参数保存，读取不同机器人的配置参数，例如保存云台电机中值信息。

11. I2C 读取 IST8310

11.1 知识要点

- I2C 协议简介
- 磁力计简介
- IST8310 的寄存器配置
- cubeMX 中 I2C 配置以及 hal 库函数

11.2 课程内容

本课程中，将学习一种常用的串行同步通讯总线协议--I2C，并学习使用 I2C，配置和读取 IST8310 磁力计的设置和数据。I2C 不仅仅用于磁力计的读取，而且可以读取其他众多的传感器，例如温度传感器，气压传感器，多路 ADC 模块等。磁力计是一种测量地球磁场强度的传感器，又名电子罗盘，可用于计算机器人的朝向。

11.3 基础学习

11.3.1 I2C 简介

I2C 是 PHILIPS 公司开发的一种半双工、双向二线制同步串行总线。两线制代表 I2C 只需两根信号线，一根数据线 SDA，另一根是时钟线 SCL。

I2C 总线允许挂载多个主设备，但总线时钟同一时刻只能由一个主设备产生，并且要求每个连接到总线上的器件都有唯一的 I2C 地址，从设备可以被主设备寻址。

I2C 通信具有几类信号：

- 开始信号 S: 当 SCL 处于高电平时，SDA 从高电平拉低至低电平，代表数据传输的开始。
- 结束信号 P: 当 SCL 处于高电平时，SDA 从低电平拉高至高电平，代表数据传输结束。
- 数据信号：数据信号每次都传输 8 位数据，每一位数据都在一个时钟周期内传递，当 SCL 处于高电平时，SDA 数据线上的电平需要稳定，当 SCL 处于低电平的时候，SDA 数据线上的电平才允许改变。
- 应答信号 ACK/NACK: 应答信号是主机发送 8bit 数据，从机对主机发送低电平，表示已经接受数据。

常见用于读取传感器数据的 I2C 传输过程如下表所示:

S	从设备地址 7bit	R/W 读 写 位	ACK	寄存器地址 +ACK	N 字节数据 +ACK	ACK/NACK	P 停 止 信 号
---	---------------	-----------------	-----	---------------	----------------	----------	-----------------

整个 I2C 通信过程理解成收发快递的过程，设备 I2C 地址理解成学校快递柜的地址，读写位代表寄出和签收快递，寄存器地址则是快递柜上的箱号，而数据便是需要寄出或者签收的快递。整个过程便是如同到学校的快递柜（从机 I2C 地址），对第几号柜箱（寄存器地址），进行寄出或者签收快递（数据）的过程。详细的过程可以参考进阶知识中的 IST8310 的读写过程。

11.3.2 磁力计简介

IST8310 是一款由 ISentek 公司推出的 3 轴磁场传感器，尺寸为 3.0*3.0*1.0mm，支持快速 I2C 通信，可达 400kHz，14 位磁场数据，测量范围可达 1600uT(x,y-axis) 和 2500uT(z-axis)，最高 200Hz 输出频率。

磁场传感器常常用于电子罗盘，计算地磁场角度，而地磁场是源自于地球内部，并延伸到太空的磁场。磁场在地表上的强度在 25—65 微特斯拉之间。同时地磁场与地球自转轴并不重合，存在 11° 的夹角，故而在地球表面存在一定磁偏角，并且随着纬度升高而变化越大。在中国大陆的大部分地区，磁偏角大概在 -10° ~+2° 之间。使用 IST8310 磁力计可以检测地磁场强度，用于计算磁场角度。

IST8310 的 GPIO 管脚各个功能如下表所示：

管脚	功能
SCL	I2C 的时钟线
SDA	I2C 的数据线
RSTN	IST8310 的 RESET，低电平重启 IST8310
DRDY	IST8310 的数据准备 (data ready)

11.4 软件学习

11.4.1 硬件接线

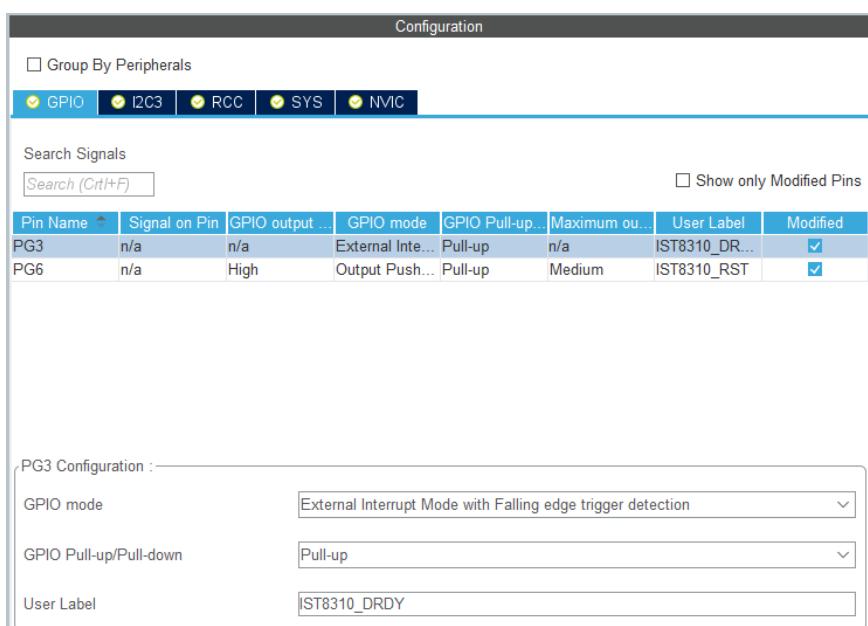
由于 IST8310 磁力计在开发板上集成，PCB 上已经接线完毕，故而不需要外部接线。对应管脚如下表所示。

IST8310 管脚	MCU 管脚
SCL	PA8
SDA	PC9
RSTN	PG6
DRDY	PG3

11.4.2 I2C 在 cubeMX 中的配置

本例程，我们新建了 `ist8310driver.c`, `ist8310driver.h`, `ist8310driver_middle.c` 以及 `ist8310driver_middle.h` 四个文件，其中 `ist8310driver.h`, `ist8310driver_middle.h` 用于申明驱动函数，`ist8310driver.c` 实现驱动函数，`ist8310driver_middle.c` 作为中间层实现 I2C 通信封装以及延迟函数，方便移植。

首先先看 cubeMX 的配置，PG3 配置外部中断，下降沿触发，外部中断的配置过程可以参考第 6 章节按键的外部中断配置，PG6 配置成 GPIO 的输出模式，上拉模式，如图所示。



The screenshot shows the Configuration software interface. At the top, there are tabs for Group By Peripherals (GPIO, I2C3, RCC, SYS, NVIC) and a search bar. Below that is a table of pins:

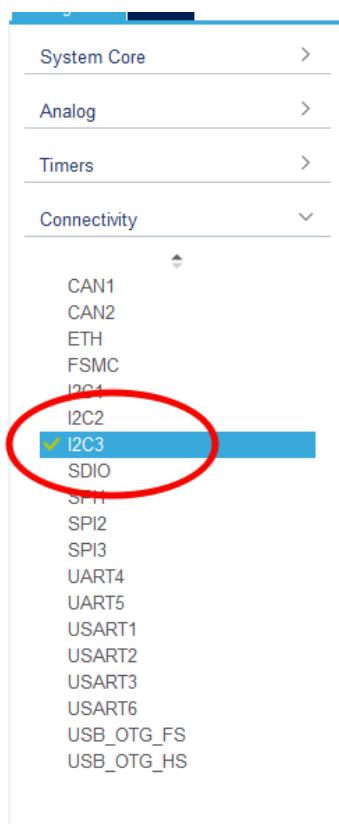
Pin Name	Signal on Pin	GPIO output...	GPIO mode	GPIO Pull-up...	Maximum ou...	User Label	Modified
PG3	n/a	n/a	External Inte...	Pull-up	n/a	IST8310_DR...	<input checked="" type="checkbox"/>
PG6	n/a	High	Output Push ...	Pull-up	Medium	IST8310_RST	<input checked="" type="checkbox"/>

Below the table, a section titled "PG6 Configuration:" shows the following settings:

- GPIO output level: High
- GPIO mode: Output Push Pull
- GPIO Pull-up/Pull-down: Pull-up
- Maximum output speed: Medium
- User Label: IST8310_RST

I2C3 的配置如下，最终配置成 I2C3 成快速模式，通信频率设置为 400k，I2C 地址配置成 7 位等等，管脚配置为 PA8，PC9，如图所示：

1. 在 connectivity 下找到 I2C3；
2. 配置成 I2C；
3. 配置成 Fast Mode；
4. 其他保持默认。



I2C3 Mode and Configuration

Mode

I2C

- Disable
- I2C**
- SMBus-Alert-mode
- SMBus-two-wire-Interface

Configuration

Reset Configuration

Parameter Settings User Constants NVIC Settings DMA Settings GPIO Settings

Configure the below parameters :

Search (Ctrl+F) ⌂ ⌂ ⓘ

Master Features

I2C Speed Mode	Fast Mode
I2C Clock Speed (Hz)	Standard Mode
Fast Mode Duty Cycle	Fast Mode

Slave Features

Clock No Stretch Mode	Disabled
Primary Address Length selection	7-bit

Configuration

Reset Configuration

Parameter Settings User Constants NVIC Settings DMA Settings GPIO Settings

Configure the below parameters :

Search (Ctrl+F) ⌂ ⌂ ⓘ

Master Features

I2C Speed Mode	Fast Mode
I2C Clock Speed (Hz)	400000
Fast Mode Duty Cycle	Duty cycle Tlow/Thigh = 2

Slave Features

Clock No Stretch Mode	Disabled
Primary Address Length selection	7-bit
Dual Address Acknowledged	Disabled
Primary slave address	0
General Call address detection	Disabled

11.4.3 主要函数介绍

首先介绍一下 I2C 的通信函数, `ist8310_IIC_read_single_reg`, `ist8310_IIC_write_single_reg`, `ist8310_IIC_read_multi_reg`, `ist8310_IIC_write_multi_reg` 四个函数, 它们都是对 hal 库的 I2C 函数的封装, 因为不只可以通过 stm32 上的硬件 I2C 进行通信, 也可以通过软件模拟

I2C 进行通信，为了移植的方便性，重新对 I2C 函数进行封装。

库函数 HAL_I2C_Mem_Read 和 HAL_I2C_Mem_Write 是 HAL 库里面自带 I2C 读取和写入函数，它实现 I2C 简介中的对 IST8310 的寄存器读取和写入过程。

11.4.3.1 HAL_I2C_Mem_Read 函数

HAL_StatusTypeDef HAL_I2C_Mem_Read(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData, uint16_t Size, uint32_t Timeout)	
函数名	HAL_I2C_Mem_Read
函数作用	从 I2C 设备的寄存器读取数据
函数返回	HAL 状态， HAL_OK 代表读取成功
参数 1: hi2c	I2C 句柄
参数 2: DevAddress	I2C 从机地址
参数 3: MemAddress	寄存器地址
参数 4: MemAddSize	寄存器地址增加大小 I2C_MEMADD_SIZE_8BIT: 增加八位 I2C_MEMADD_SIZE_16BIT: 增加十六位
参数 5: pData	数据指针
参数 6: Size	数据长度
参数 7: Timeout	超时时间

11.4.3.2 HAL_I2C_Mem_Write 函数

HAL_StatusTypeDef HAL_I2C_Mem_Write(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData, uint16_t Size, uint32_t Timeout)	
函数名	HAL_I2C_Mem_Write

函数作用	往 I2C 设备的寄存器写入数据
函数返回	HAL 状态, HAL_OK 代表读取成功
参数 1: hi2c	I2C 句柄
参数 2: DevAddress	I2C 从机地址
参数 3: MemAddress	寄存器地址
参数 4: MemAddSize	寄存器地址增加大小 I2C_MEMADD_SIZE_8BIT: 增加八位 I2C_MEMADD_SIZE_16BIT: 增加十六位
参数 5: pData	数据指针
参数 6: Size	数据长度
参数 7: Timeout	超时时间

接着介绍一下 IST8310 的初始化过程。

步骤	语句	功能	备注
1. 初始化 GPIO 和通信	<code>ist8310_GPIO_init();</code> <code>ist8310_com_init();</code>	初始化管脚和 I2C 通信接口，保证正常通信。	Hal 库初始化已经配置，如果使用模拟 I2C 可以在这里进行初始化。
2. 重启设备	<code>ist8310_RST_L();</code> <code>ist8310_delay_ms(sleepTime);</code> <code>ist8310_RST_H();</code> <code>ist8310_delay_ms(sleepTime);</code>	通过 IST8310 重启管脚进行重启。	

3. 验证设备 ID	通过读取 WHO_AM_I 寄存器判断	判断 IST8310 通信是否正常	详细寄存器介绍 参考进阶学习中的 IST8310 寄存器介绍。
4. 配置 IST8310	配置 IST8310 四个寄存器 0x0B: 中断寄存器, 配置成开启中断, 中断时为低电平; 0x41: 采样次数寄存器, 配置成 x,y,z 均是 2 次采样 0x42: 需要配置成 0xC0; 0x0A: 配置成 200Hz 输出频率		

其次, 再介绍读取处理函数 `ist8310_read_over`, 该函数用于已经读取从 STAT1 寄存器到 DATAZH 寄存器共 7 个数据, 将其处理成单位是 uT 的磁场强度数据。函数原型如图所示,主要是通过判断 `stat1` 寄存器值判断有没有新的数据产生, 将两个八位数据合并成一个 16 位整型数据, 再乘以 0.3 灵敏度变成单位是 ut 的磁场强度数据。

```
void ist8310_read_over(uint8_t *status_buf, ist8310_real_data_t *ist8310_real_data)
{
    if (status_buf[0] & 0x01)
    {
        int16_t temp_ist8310_data = 0;
        ist8310_real_data->status |= 1 << IST8310_DATA_READY_BIT;

        temp_ist8310_data = (int16_t)((status_buf[2] << 8) | status_buf[1]);
        ist8310_real_data->mag[0] = MAG_SEN * temp_ist8310_data;
        temp_ist8310_data = (int16_t)((status_buf[4] << 8) | status_buf[3]);
        ist8310_real_data->mag[1] = MAG_SEN * temp_ist8310_data;
        temp_ist8310_data = (int16_t)((status_buf[6] << 8) | status_buf[5]);
        ist8310_real_data->mag[2] = MAG_SEN * temp_ist8310_data;
    }
    else
    {
        ist8310_real_data->status &= ~(1 << IST8310_DATA_READY_BIT);
    }
}
```

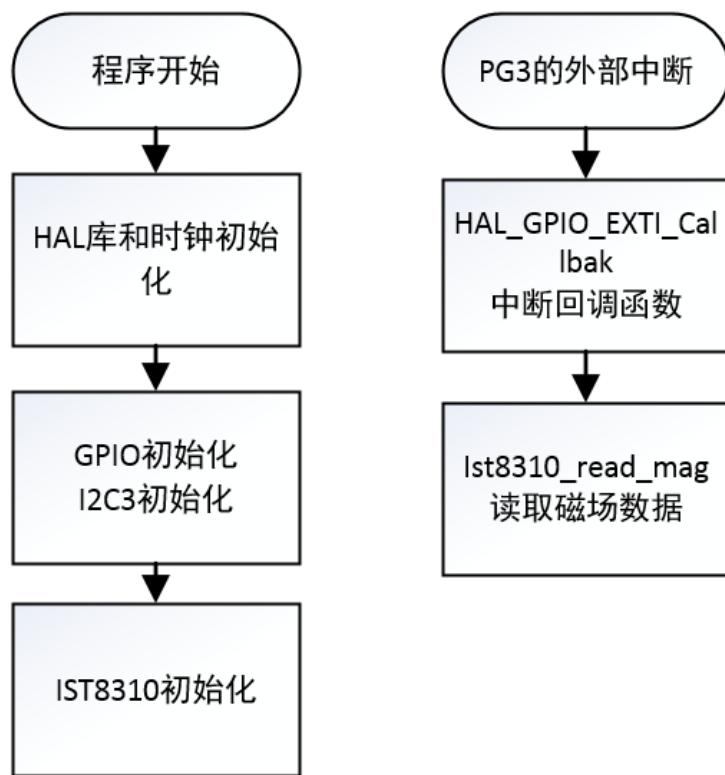
最后介绍一个通用的读取磁场数据函数 `ist8310_read_mag`, 函数原型如图所示, 主要通过 I2C 读取多个字节函数, 读取 DATAXL 寄存器, 并将两个八位数据合并成一个 16 位整型数据, 再乘以 0.3 灵敏度变成单位是 ut 的磁场强度数据, 输出到传入的数组中。

```
void ist8310_read_mag(fp32 mag[3])
{
    uint8_t buf[6];
    int16_t temp_ist8310_data = 0;
    //read the "DATAXL" register (0x03)
    ist8310_IIC_read_muli_reg(0x03, buf, 6);

    temp_ist8310_data = (int16_t)((buf[1] << 8) | buf[0]);
    mag[0] = MAG_SEN * temp_ist8310_data;
    temp_ist8310_data = (int16_t)((buf[3] << 8) | buf[2]);
    mag[1] = MAG_SEN * temp_ist8310_data;
    temp_ist8310_data = (int16_t)((buf[5] << 8) | buf[4]);
    mag[2] = MAG_SEN * temp_ist8310_data;
}
```

11.4.4 程序流程

程序开始先进行 HAL 库自带的初始化，包括时钟，GPIO，I2C3 的初始化；之后完成配置 IST8310，IST8310 的 DRDY 引脚会产生 200Hz 的周期信号；当 DRDY 下降沿，会引起单片机的下降沿外部中断；在外部中断回调函数中，调用 ist8310 的读取函数，便可以读取磁场数据。



11.4.5 效果展示

读取数据后，进入 Debug 模式，从 watch 窗口中观察数据大小。

Name	Value	Type
mag	0x20000010 mag	float[3]
[0]	17.1000004	float
[1]	-16.5	float
[2]	24.3000011	float
<Enter expression>		

11.5 进阶学习

11.5.1 IST8310 的读写过程

11.5.1.1 IST8310 读取单字节过程

IST8310 单字节读取过程如所示。

1. 发送一个起始信号；
2. 发送 IST8310 的 I²C 地址和读写位；其中读写位为写（0）；
3. 等待 IST8310 从机的 ACK 位；
4. 发送 IST8310 需要读取的寄存器地址；
5. 等待 IST8310 从机的 ACK 位；
6. 又一次产生一个起始信号；
7. 在发送 IST8310 的 I²C 地址和读写位，读写位为读（1）；
8. 在等待 IST8310 从机的 ACK 位后；
9. IST8310 从机会发送对应寄存器的数据；
10. 由于主机只接受一个字节数据，故而主机不发送 ACK 位；
11. 主机在发送停止信号后，停止这次通信。

Single Byte Read:										
SA	Slave Address+ RW	ACK	Reg Address	ACK	SP	Slave Address+RW	ACK	DATA	NA	ST

Figure 4. I²C Single Byte Read Operation

对于开发板 C 型的 IST8310，I²C 的地址是 0x0E，读取 0x00 寄存器的值，整个发送过程如下表所示。

序号	1	2	3	4
信号	起始信号	I ² C 地址(写操作)	ACK	寄存器地址
发送值	SCL 高 SDA 从高电平拉低	0x0E<<1	SDA 从高电平拉低	0x00
发送者	主机 MCU	主机 MCU	从机 IST8310	主机 MCU
序号	5	6	7	8
信号	ACK	起始信号	I ² C 地址(读操作)	ACK

发送值	SDA 从高电平拉低	SCL 高 SDA 从高电平拉低	0x0E<<1 0x01	SDA 从高电平拉低
发送者	从机 IST8310	主机 MCU	主机 MCU	从机 IST8310
序号	9	10	11	
信号	IST8310 的 0x00 寄存器值	NACK	停止信号	
发送值	0x10	SDA 保持高电平	SCL 高 SDA 从低电平拉高	
发送者	从机 IST8310	主机 MCU	主机 MCU	

11.5.1.2 IST8310 读取多字节过程

IST8310 多字节读取过程如图所示。与单字节读取过程不同在于第 10 步中。

主机接收到一个字节后，主机发送 ACK 信号，则从机 IST8310 会接着再发送下一个寄存器的值，直到主机发送一个 NACK 信号，从机便停止发送数据。

Multiple Byte Read:

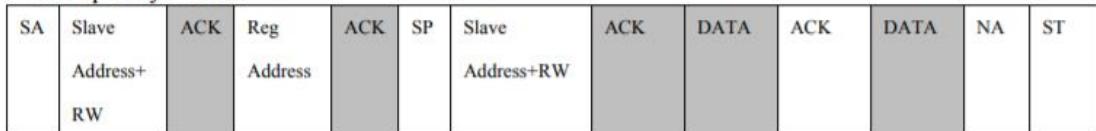


Figure 5. I²C Multiple Byte Read Operation

ACK: Acknowledge, NA: Not Acknowledge, SA: START Condition, SP: Repeat Start Condition, ST: STOP Condition

■: Slave to Master □: Master to Slave

对 IST8310 的寄存器 0x00 连续读取数据，整个发送过程如下表所示。

序号	1	2	3	4
信号	起始信号	I2C 地址(写操作)	ACK	寄存器地址

发送值	SCL 高 SDA 从高电平拉低	0x0E<<1	SDA 从高电平拉低	0x00
发送者	主机 MCU	主机 MCU	从机 IST8310	主机 MCU
序号	5	6	7	8
信号	ACK	起始信号	I2C 地址(读操作)	ACK
发送值	SDA 从高电平拉低	SCL 高 SDA 从高电平拉低	0x0E<<1 0x01	SDA 从高电平拉低
发送者	从机 IST8310	主机 MCU	主机 MCU	从机 IST8310
序号	9	10	11	12
信号	IST8310 的 0x00 寄存器值	ACK	IST8310 的 0x01 寄存器值	ACK
发送值	0x10	SCL 高 SDA 从高电平拉低	0xNN	SCL 高 SDA 从高电平拉低
发送者	从机 IST8310	主机 MCU	从机 IST8310	主机 MCU
序号	13N 个数据后	15	16
信号	IST8310 的 0x02 寄存器值	NACK	停止信号
发送值	X-axis 的低八位数据		SDA 保持高电平	SCL 高 SDA 从低电平拉高

发送者	从机 IST8310		主机 MCU	主机 MCU
-----	---------------	--	-----------	-----------

11.5.1.3 IST8310 写入单字节过程

IST8310 的写入数据过程比读取过程简单，少了中间重新产生起始信号以及第二次发送 I²C 地址的过程，IST8310 单字节写入过程如图 3.12.5 所示。

1. 首先发送一个起始信号；
2. 发送 IST8310 的 I²C 地址和写操作(1)；
3. 等待 IST8310 从机的 ACK 位；
4. 发送需要写入的寄存器地址值；
5. 等待 IST8310 的 ACK 位；
6. 主机发送需要写入的值；
7. 等待 IST8310 的 ACK 位后；
8. 主机产生一个停止信号，完成本次通信。

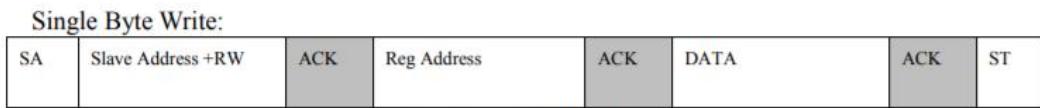


Figure 6. I²C Single Byte Write Operation

ACK: Acknowledge, NA: Not Acknowledge, SA: START Condition, SP: Repeat Start Condition, ST: STOP Condition
■: Slave to Master □: Master to Slave

假设在向 IST8310 的寄存器 0x0A 地址写入 0x0B 值，整个过程如下表所示。

序号	1	2	3	4
信号	起始信号	I ² C 地址(写操作)	ACK	寄存器地址
发送值	SCL 高 SDA 从高电平拉低	0x0E<<1	SDA 从高电平拉低	0x0A
发送者	主机 MCU	主机 MCU	从机 IST8310	主机 MCU

序号	5	6	7	8
信号	ACK	写入值	ACK	停止信号
发送值	SDA 从高电平拉低	0x0B	SDA 从高电平拉低	SCL 高 SDA 从低电平拉高
发送者	从机 IST8310	主机 MCU	从机 IST8310	主机 MCU

11.5.1.4 IST8310 写入多字节过程

IST8310 的写入多字节数据过程，与写入一个字节数据的过程相似，不同在于在写入一个字节的数据后不发送停止信号，而是接着发送数据，如图所示。

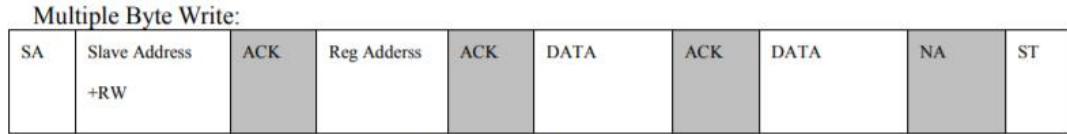


Figure 7. I²C Multiple Byte Write Operation

ACK: Acknowledge, NA: Not Acknowledge, SA: START Condition, SP: Repeat Start Condition, ST: STOP Condition
■: Slave to Master □: Master to Slave

假设在向 IST8310 的寄存器 0x0A 地址写入 0x0B 值，往 0x0B 写入 0x08，整个过程如下表所示。

序号	1	2	3	4
信号	起始信号	I ² C 地址(写操作)	ACK	寄存器地址
发送值	SCL 高 SDA 从高电平拉低	0x0E<<1	SDA 从高电平拉低	0x0A
发送者	主机 MCU	主机 MCU	从机 IST8310	主机 MCU
序号	5	6	7	8

信号	ACK	0x0A 写入值	ACK	0x0B 写入值
发送值	SDA 从高电平拉低	0x0B	SDA 从高电平拉低	0x08
发送者	从机 IST8310	主机 MCU	从机 IST8310	主机 MCU
序号	9	10		
信号	ACK	停止信号		
发送值	SDA 从高电平拉低	SCL 高 SDA 从低电平拉高		
发送者	从机 IST8310	主机 MCU		

11.5.2 IST8310 的寄存器信息

IST8310 一共 13 个寄存器，如下所示：

1. Who Am I 寄存器 0x00

该寄存器中保存设备 ID，对于 IST8310 芯片，该寄存器值固定为 0x10，只读。

Who Am I (0x00)			
位	说明	读写	默认值
7:0	设备 ID	只读	0x10

2. 状态 1 寄存器 0x02

该寄存器提供 IST8310 的状态信息。

STAT1(0x02)			
位	说明	读写	默认值

7:2	保留		
1	数据溢出，当该位置 1 代表有旧的数据未被读取，而新数据已经产生。	只读	0
0	数据准备状态：当该位置 1，代表已经有新的测量数据产生。	只读	0

3. 输出数据寄存器 0x03-0x08

DATAXL(0x02)			
位	说明	读写	默认值
7:0	X 轴磁场数据低八位	只读	0
DATAXH(0x04)			
位	说明	读写	默认值
7:0	X 轴磁场数据高八位	只读	0
DATAYL(0x05)			
位	说明	读写	默认值
7:0	Y 轴磁场数据低八位	只读	0
DATAYH(0x06)			
位	说明	读写	默认值
7:0	Y 轴磁场数据高八位	只读	0
DATAZL(0x07)			
位	说明	读写	默认值

7:0	Z 轴磁场数据低八位	只读	0
DATAZH(0x08)			
位	说明	读写	默认值
7:0	Z 轴磁场数据高八位	只读	0

4. 状态 2 寄存器 0x09

该寄存器提供 IST8310 的状态信息。

STAT2(0x09)			
位	说明	读写	默认值
7:4	保留		
3	中断标识，当中断事件发生了，该位将置 1	只读	0
2:0	保留		

5. 控制设置 1 寄存器 0x0A

该寄存器设置 IST8310 的测量模式。

CNTL1(0x0A)			
位	说明	读写	默认值
7:4	保留		
3:0	测量模式： 0: 休眠模式 1: 单次测量模式 11: 连续测量模式输出频率 200Hz	可读/可写	0

6. 控制设置 2 寄存器

该寄存器主要设置中断 IO。

CNTL2(0x0B)			
位	说明	读写	默认值
7:4	保留		
3	中断功能使能: 0: 不启用 1: 开启中断	可读/可写	1
2	DRDY 管脚中断时电平 0: 低电平 1: 高电平	可读/可写	1
1	保留		
0	软件重启 0: 不重启 1: 重启	可读/可写	0

7. 自检寄存器 0x0C

CNTL2(0x0C)			
位	说明	读写	默认值
7	保留		
6	自检: 当该位置 1, 芯片进入自检模式	可读/可写	0
5:0	保留		

8. 温度寄存器

TEMP(L)(0x1C)			
位	说明	读写	默认值
7:0	温度数据低八位	只读	0
TEMP(H)(0x1D)			
位	说明	读写	默认值
7:0	温度数据高八位	只读	0

9. 采样平均寄存器

该寄存器设置一次循环中的采样次数来降低数据噪声，高采样次数可以降低数据噪声。

AVGCNTL(0x41)			
位	说明	读写	默认值
7:6	保留		0
5:3	Y 轴的采样次数 3b00: 一次采样 3b001: 两次采样平均 3b010: 四次采样平均 (推荐) 3b011: 八次采样平均 3b100: 十六次采样平均 其他: 一次采样	可读/可写	0
2:0	X 轴和 Z 轴的采样次数 3b00: 一次采样 3b001: 两次采样平均	可读/可写	0

	3b010:四次采样平均（推荐） 3b011:八次采样平均 3b100:十六次采样平均 其他:一次采样		
--	--	--	--

11.6 课程总结

I2C 通信是一种常见的通信方式，在许多传感器都能发现这种通信方式，例如温度传感器等，而传感器是机器人的感知系统，如同机器人的眼睛，没有感知外界环境的变化，机器人便如同在黑夜里走路，无法在正确的轨道上运行。磁力计是感知地球磁场强度的传感器，鸽子便是通过地球磁场感应方向，寻找北归的路线，机器人也能通过磁力计算出朝向，可以在后面的姿态解算章节阐述。

12. OLED 显示

12.1 知识要点

- OLED 简介
- OLED 通信协议
- OLED 配置命令介绍
- 简单画图函数介绍

12.2 课程内容

在前面一章中，学习了 I2C 通信方式，使用 I2C 获取 IST8310 的磁场数据，本课程中接着使用 I2C 通信方式点亮 OLED 模块，并显示 Robomaster LOGO。由于 OLED 模块支持多种通信方式，OLED 模块的 I2C 通信过程与 IST8310 存在不同之处，通过在数据层进行二次打包，以达到分类数据包的目的，以便适配 OLED 的多种通信方式。

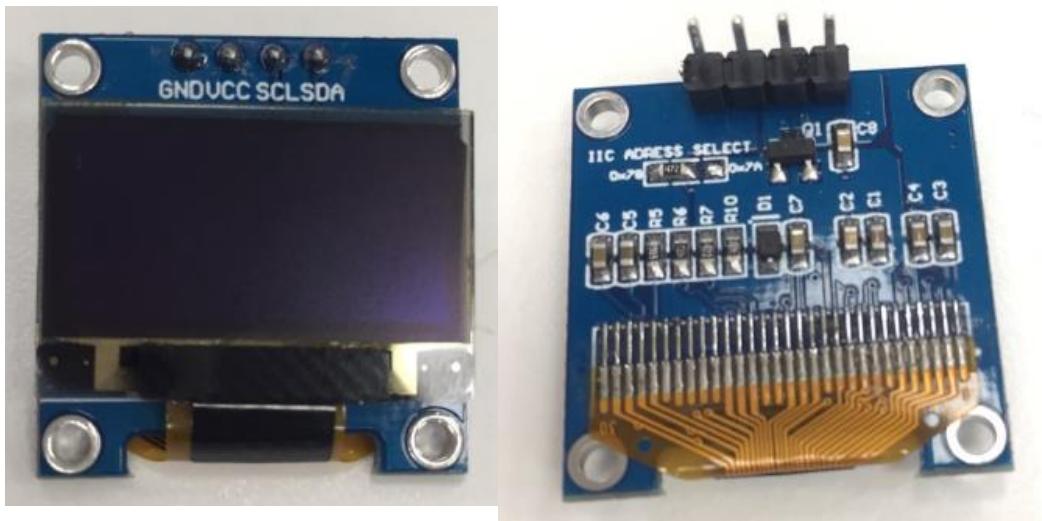
12.3 12.3 基础学习

12.3.1 OLED 简介

OLED (OrganicLight-Emitting Diode)，又称为有机电激光显示、有机发光半导体(Organic Electroluminescence Display, OLED)。OLED 属于一种电流型的有机发光器件，发光原理是通过载流子的注入和复合而致发光，发光强度与注入的电流成正比。OLED 在电场的作用下，阳极产生的空穴和阴极产生的电子就会发生移动，分别向空穴传输层和电子传输层注入，迁移到发光层。当二者在发光层相遇时，产生能量激子，从而激发发光分子最终产生可见光。[百度百科]

OLED 屏幕模块在 RM 比赛中，具有显示数据，指引操作等作用，例如显示传感器数据，提醒队员检查机器人。OLED 模块价格便宜，很多店家都有出售，常见屏幕尺寸为 0.96 寸，分辨率为 128*64，常见的通信方式支持 6800,8080 两种并口接口，或者 SPI 接口和 I2C 接口。以下以一款引出 I2C 接口的 OLED 模块为例。

该模块的外观如图所示：



OLED 模块外观图

该 OLED 模块使用 I2C 通信，I2C 设备的默认地址为 0x78，可以通过手动修改左上方的电阻来配置 I2C 地址为 0x7A。

由于 OLED 模块本身不会像 IST8310 有读取数据过程，整个过程均为主机不断向从机 OLED 模块发送数据，发送的数据中 I2C 地址后的第一个数据表示的是控制指令（0x00）还是数据指令（0x40）。它们的异同点如下表所示：

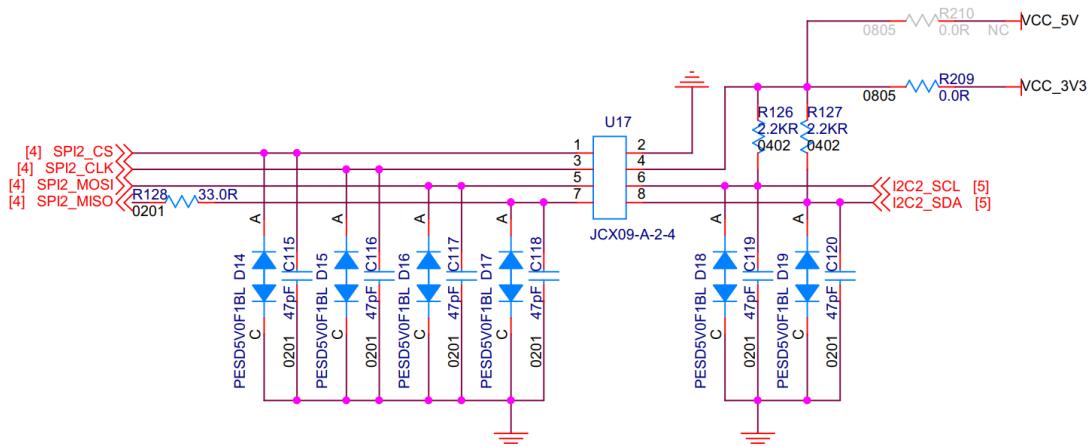
	IST8310	OLED
不同点	有读操作和写操作	只有写操作
	发送 I2C 地址后，发送寄存器地址信息	发送 I2C 地址后，发送数据类型再发送数据，其中数据类型为 0x00 表示控制指令，0x40 表示数据指令。
相同点	<ul style="list-style-type: none"> ● 都有固定的 I2C 地址， ● 都支持最高 400K 的 I2C 频率 ● 都传输 8 字节数据 	

可以看出 OLED 通信过程与 IST8310 通信过程最大不同在于传输 I2C 地址后的数据上，OLED 通过一个字节的数据，来区分之后的数据类型。

12.4 软件学习

12.4.1 硬件接线

本例程使用 C 型开发板上的用户 I2C 接口，原理图如图所示：



用户 I2C 原理图

使用的 I2C 为 I2C2，开发板 C 型上的 I2C 线序与 OLED 模块上的线序保持一致，故而只需要正常连线即可。开发板 IO 口对于关系如下：

- PF0 对应 I2C2_SDA，与 OLED 的 SDA 相连；
- PF1 对应 I2C2_SCL，与 OLED 的 SCL 相连。

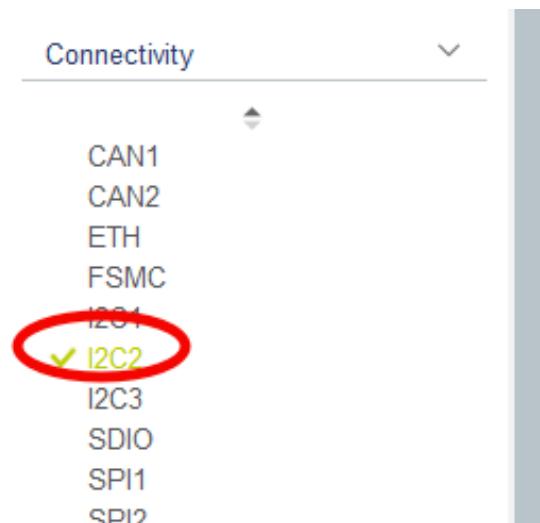
注意使用杜邦线连接 OLED 模块，注意杜邦线长度不能过长，否则可能导致 I2C 通信受到干扰而通信失败。

12.4.2 cubeMX 配置过程

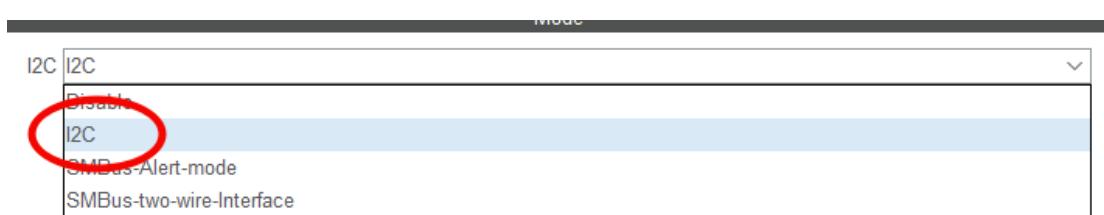
本例程，新建了 OLED.c，OLED.h 以及 OLEDfont.h 三个文件，其中 OLED.h 用于申明驱动函数，OLED.c 实现驱动函数，OLEDfont.h 存放了 ASCII 字符编码以及 Robomaster LOGO 编码。

cubeMX 中配置 I2C 如下：通信频率设置为 400k，管脚配置为 PF0，PF1，I2C 地址配置成 7 位等等，如图所示：

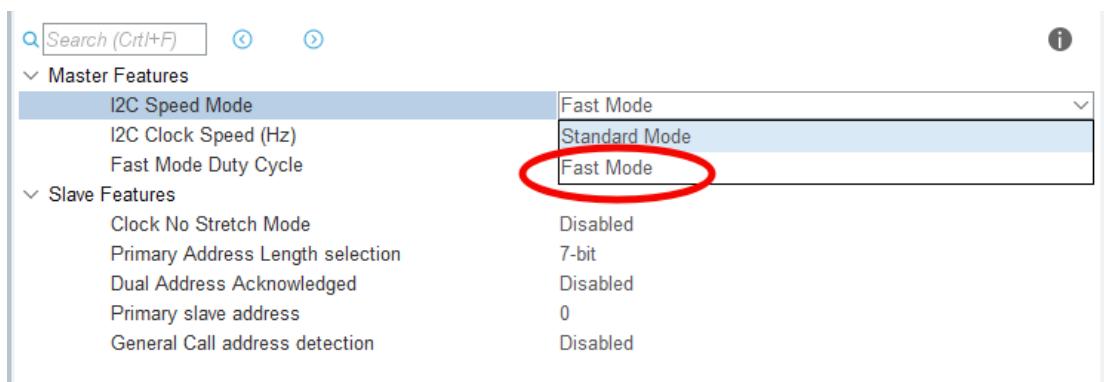
1. 在 connectivity 下找到 I2C2；
2. 配置成 I2C；
3. 配置成 Fast Mode；
4. 其他保持默认。



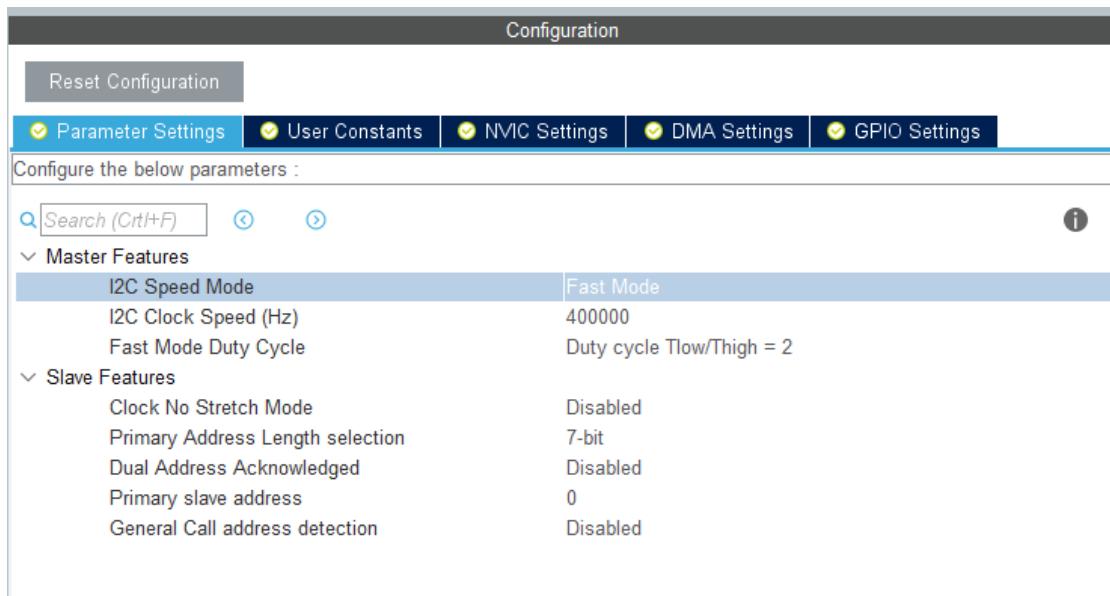
Connectivity 下的 I2C2



配置 I2C



配置 Fast Mode



I2C 配置最终图

12.4.3 主要函数介绍

1. HAL_I2C_Master_Transmit 函数

HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout)	
函数名	HAL_I2C_Master_Transmit
函数作用	向某个 I2C 设备传输数据
函数返回	HAL 状态, HAL_OK 代表读取成功
参数 1: hi2c	I2C 句柄
参数 2: DevAddress	I2C 从机地址
参数 3: pData	数据指针
参数 4: Size	数据长度
参数 5: Timeout	超时时间

2. oled_write_byte 函数

根据 OLED 的通信方式，需要在第一个字节中指明之后的数据类型，如果是控制指令则需要发送 0x00，如果是数据指令则需要发送 0x40。函数原型如下：

```
/**  
 * @brief      write data/command to OLED, if you use spi, please rewrite the function  
 * @param[in]   dat: the data ready to write  
 * @param[in]   cmd: OLED_CMD means command; OLED_DATA means data  
 * @retval     none  
 */  
  
/**  
 * @brief      写数据或者指令到 OLED， 如果使用的是 SPI， 请重写这个函数  
 * @param[in]   dat: 要写入的字节  
 * @param[in]   cmd: OLED_CMD 代表写入的字节是指令; OLED_DATA 代表写入的字节是数据  
 * @retval     none  
 */  
  
void oled_write_byte(uint8_t dat, uint8_t cmd)  
{  
    static uint8_t cmd_data[2];  
  
    if(cmd == OLED_CMD)  
    {  
        cmd_data[0] = 0x00;  
    }  
    else  
    {  
        cmd_data[0] = 0x40;  
    }  
}
```

```
cmd_data[1] = dat;  
  
HAL_I2C_Master_Transmit(&hi2c2, OLED_I2C_ADDRESS, cmd_data, 2, 10);  
}
```

函数名	Oled_write_byte
函数作用	向 OLED 发送一个指令
函数返回	None
参数 1: data	指令数据
参数 2: cmd	控制指令或者数据指令 OLED_CMD 为控制指令 OLED_DATA 为数据指令

3. oled_init 函数

介绍 OLED_init 函数，该函数主要配置 OLED 参数，通过调用 oled_write_byte 传入 OLED_CMD，传输控制指令完成配置，如图所示。详细功能参考进阶学习。

```

void OLED_init(void)
{
    oled_write_byte(0xAE, OLED_CMD);           //display off
    oled_write_byte(0x20, OLED_CMD);           //Set Memory Addressing Mode
    oled_write_byte(0x10, OLED_CMD);           //00,Horizontal Addressing Mode;01,Vertical Addressing Mode;10,Pa
    oled_write_byte(0xb0, OLED_CMD);           //Set Page Start Address for Page Addressing Mode,0-7
    oled_write_byte(0xc8, OLED_CMD);           //Set COM Output Scan Direction
    oled_write_byte(0x00, OLED_CMD);           //---set low column address
    oled_write_byte(0x10, OLED_CMD);           //---set high column address
    oled_write_byte(0x40, OLED_CMD);           //---set start line address
    oled_write_byte(0x81, OLED_CMD);           //---set contrast control register
    oled_write_byte(0xff, OLED_CMD);           //brightness 0x00~0xff
    oled_write_byte(0xa1, OLED_CMD);           //---set segment re-map 0 to 127
    oled_write_byte(0xa6, OLED_CMD);           //---set normal display
    oled_write_byte(0xa8, OLED_CMD);           //---set multiplex ratio(1 to 64)
    oled_write_byte(0x3f, OLED_CMD);           //
    oled_write_byte(0xa4, OLED_CMD);           //0xa4,Output follows RAM content;0xa5,Output ignores RAM content
    oled_write_byte(0xd3, OLED_CMD);           //---set display offset
    oled_write_byte(0x00, OLED_CMD);           //---not offset
    oled_write_byte(0xd5, OLED_CMD);           //---set display clock divide ratio/oscillator frequency
    oled_write_byte(0xf0, OLED_CMD);           //---set divide ratio
    oled_write_byte(0xd9, OLED_CMD);           //---set pre-charge period
    oled_write_byte(0x22, OLED_CMD);           //
    oled_write_byte(0xda, OLED_CMD);           //---set com pins hardware configuration
    oled_write_byte(0x12, OLED_CMD);           //
    oled_write_byte(0xdb, OLED_CMD);           //---set vcomh
    oled_write_byte(0x20, OLED_CMD);           //0x20,0.77xVcc
    oled_write_byte(0x8d, OLED_CMD);           //---set DC-DC enable
    oled_write_byte(0x14, OLED_CMD);           //
    oled_write_byte(0xaf, OLED_CMD);           //---turn on oled panel
}

```

OLED_init 函数图

4. OLED_display_on 函数与 OLED_display_off 函数

OLED_display_on 和 OLED_display_off 函数分别是用来关闭 OLED 显示和开启 OLED 显示。

函数	功能
OLED_display_on	打开 OLED 显示
OLED_display_off	关闭 OLED 显示

5. OLED_operate_gram 函数

OLED_operate_gram 函数是操作 OLED_GRAM[128][8]数组的，我们对整个数组进行操作，操作完成后再通过 OLED_refresh_gram 函数整体刷新 OLED 内部的 GRAM。

OLED_operate_gram 如图所示：

函数名	OLED_operate_gram
功能	打开 OLED 显示
函数返回	None

参数 1: pen	<p>操作类型:</p> <p>WRITE: 代表将所有像素点亮;</p> <p>CLEAR: 代表将所有像素熄灭;</p> <p>INVERSION: 代表将所有像素状态反转。</p>
-----------	--

```

/**
 * @brief  operate the graphic ram(size: 128*8 char)
 * @param  pen: the type of operate.
 *         PEN_CLEAR: set ram to 0x00
 *         PEN_WRITE: set ram to 0xff
 *         PEN_INVERSION: bit inversion
 * @retval none
 */
void OLED_operate_gram(pen_typedef pen)
{
    uint8_t i, n;

    for (i = 0; i < 8; i++)
    {
        for (n = 0; n < 128; n++)
        {
            if (pen == PEN_WRITE)
            {
                OLED_GRAM[n][i] = 0xff;
            }
            else if (pen == PEN_CLEAR)
            {
                OLED_GRAM[n][i] = 0x00;
            }
            else
            {
                OLED_GRAM[n][i] = 0xff - OLED_GRAM[n][i];
            }
        }
    }
}

```

OLED_operate_gram 提供三种操作方式:

- 第一种: **PEN_WRITE** 是将数组都设置为 **0xff**, 对于 OLED 屏幕即为全亮,
- 第二种: **PEN_CLEAR** 是将数组都设置为 **0x00**,对于 OLED 屏幕即为全灭,
- 第三种: **PEN_INVERSION** 是将数组的值全部反转, 通过与 **0xff** 相减来实现。

6. OLED_refresh_gram 函数

OLED_refresh_gram 函数功能是将内部的 **GRAM[8][128]** 传输到 OLED 模块的 **GRAM**, 这样 OLED 就会显示图像。实现过程是先设置显示起始页地址, 然后连续刷新这一页

的 128 行数据，调用 `oled_write_byte` 函数，传入 `OLED_DATA` 参数，表示传输的数据类型为“数据指令”。函数实现如图所示：

```
/**  
 * @brief    send the data of gram to oled screen  
 * @param    none  
 * @retval   none  
 */  
void OLED_refresh_gram(void)  
{  
    uint8_t i, n;  
  
    for (i = 0; i < 8; i++)  
    {  
        OLED_set_pos(0, i);  
        for (n = 0; n < 128; n++)  
        {  
            oled_write_byte(OLED_GRAM[n][i], OLED_DATA);  
        }  
    }  
}
```

7. OLED_set_pos 函数

`OLED_set_pos` 函数是设置 OLED 光标位置，之后如果传输显示数据，会在对应的位置进行显示。函数的实现原理可以参考 `SSD1306` 数据手册 34-35 页中 10.1.3 章和 10.1.13 章讲解。

```
/**  
 * @brief    cursor set to (x,y) point  
 * @param    x:X-axis, from 0 to 127  
 * @param    y:Y-axis, from 0 to 7  
 * @retval   none  
 */  
void OLED_set_pos(uint8_t x, uint8_t y)  
{  
    oled_write_byte((0xb0 + y), OLED_CMD);           //set page address y  
    oled_write_byte(((x&0xf0)>>4)|0x10, OLED_CMD); //set column high address  
    oled_write_byte((x&0x0f), OLED_CMD);             //set column low address  
}
```

OLED_set_pos 函数图

还有一些画图函数，如下表所示。

画图函数功能表

函数	功能
OLED_draw_point	对(x,y)坐标的一个像素点进行操作
OLED_draw_line	从(x1,y1)到(x2,y2)的直线经过的像素点进行操作
OLED_show_char	显示一个字符
OLED_show_string	显示一个字符串
OLED_printf	Printf 函数功能
OLED_LOGO	显示 Robomaster LOGO

oled.c 文件中的函数介绍到此，oled.h 内容主要为一些定义，oledfrot.h 文件主要定义了字符的编码和 Robomaster LOGO 编码。

显示过程如下：

1. OLED 模块的初始化，调用 OLED_init
2. 通过画图函数，对 stm32 内的 GRAM 数组进行操作
3. 调用 OLED_refresh_gram 函数将 GRAM 数据传输到 OLED 模块的 GRAM 进行显示。
4. 其中 OLED_LOGO 函数内集成将 GRAM 刷新成 LOGO 的数据，以及最后调用了 OLED_refresh_gram 函数。

主函数 main 如下所示。

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

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */
    OLED_LOGO();
    HAL_Delay(1000);
    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
}
```

main 函数图

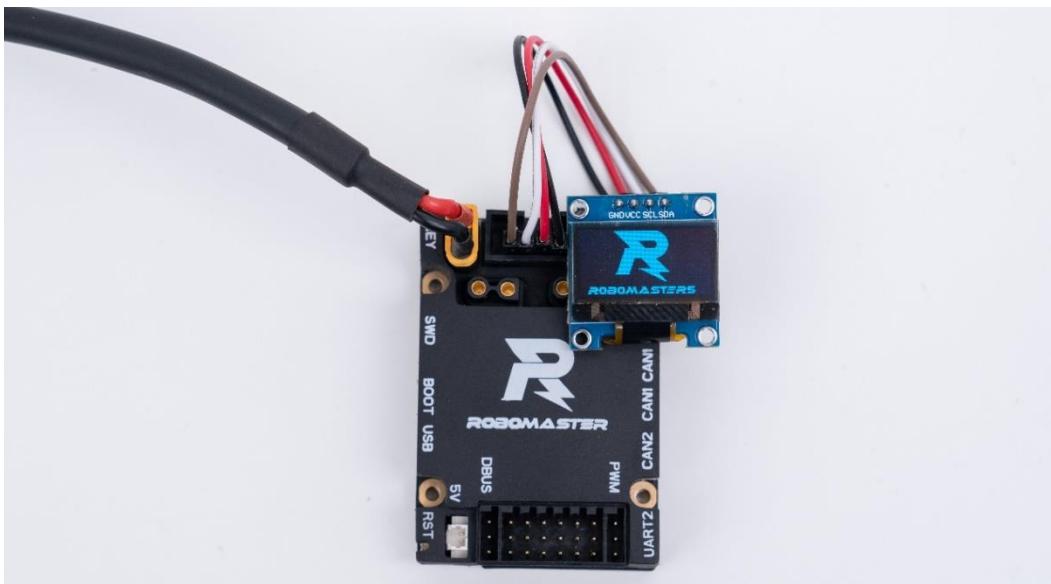
12.4.4 程序流程

HAL 库自带的初始化过后，进行 OLED 的初始化，进入主循环，每一秒刷新一次 OLED 屏幕，显示 Robomaster LOGO.



12.4.5 效果展示

外接 OLED 模块后，可以直接在 OLED 屏幕上观察 RoboMaster LOGO。



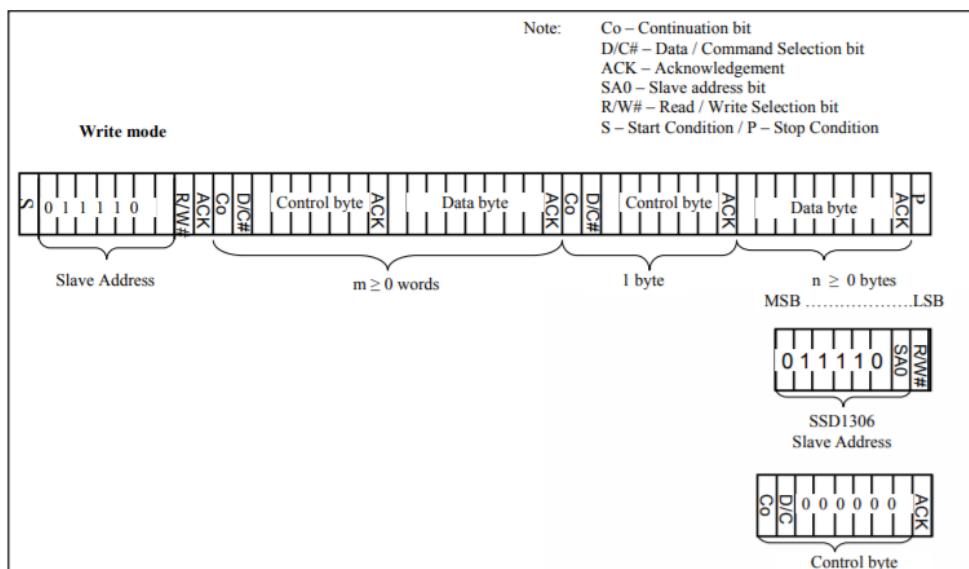
LOGO 显示效果图

12.5 进阶学习

12.5.1 OLED 通信过程

OLED 模块使用的控制器是 SSD1306, 具体可以参考 SSD1306 的数据手册, 了解 SSD1306 如何控制 OLED 屏幕显示字符, SSD1306 控制器支持 6800,8080 两种并行接口通信方式, 以及 4 线 SPI 通信和 I²C 通信方式, 我们以下介绍 I²C 通信方式。从 SSD1306 的数据手册第 20 页, 我们可以看到如下通信时序图, 如图所示

Figure 8-7 : I²C-bus data format



I²C 通信时序图

整个 I2C 通信过程如下所示：

1. 产生 I2C 的起始位；
2. 发送 OLED 的 7 位 I2C 地址，对于 SSD1306 的 I2C 地址，根据 SA0 的电平高低，即电阻的焊接方式的不同，分为“b0111100”(SA0 低电平)和 “b0111101”(SA0 高电平)；
3. 发送读写位，对于 SSD1306，此项为低电平 0，在需要发送的 I2C 地址加上写读位 0，为 0x78("b0111100 0")或者为 0x7A("b0111101 0");
4. 等待从机 OLED 的 ACK 回应；
5. 传输一个控制字节，该字节将决定之后传输的数据是一个控制指令还是一个数据指令。这个控制字节包括一个 Co 位以及 D/C 位，6 个 0 位；
 - 1) Co 位设置成 0，那么之后传输的数据只包含数据字节；
 - 2) D/C 位决定接下来的传输数据是控制指令还是数据，D/C 位设置为 0，则是控制指令，D/C 位设置为 1 则为数据；
6. 等待从机 OLED 的 ACK 回应；
7. 发送若干个数据信息，并等待从机 OLED 的 ACK 回应
8. 当主机发送一个停止位之后，整个传输过程结束。停止位是当 SCL 管脚处于高电平的时候，SDA 管脚从低电平拉高至高电平。

从上述过程中，看出如果需要对 OLED 进行发送控制指令，那么需要在发送完 I2C 地址后发送 0x00 数据；如果需要对 OLED 进行发送数据指令，那么需要在发送完 I2C 地址后发送 0x40 数据。

12.5.2 OLED 初始化配置

初始化过程中，配置的参数可以参考下表：

OLED_init 配置表

指令	功能	参考
0xAE	关闭 OLED 显示	SSD1306 数据手册 37 页中 10.1.12 章讲解
0x20 0x10	设置 OLED 的内存地址模式为页地址模式。	SSD1306 数据手册 34-35 页中 10.1.3 章讲解
0xb0	设置起点页地址为第一页地址	SSD1306 数据手册 37 页中 10.1.13 章讲解
0xC8	设置 SSD1306 的 COM 扫描方式	SSD1306 数据手册 37 页中 10.1.14 章讲解

指令	功能	参考
0x00 0x10	设置页地址的行起始地址 其中 0x00 设置行起始地址的低 4 位, 0x10 设置行起始地址的高 4 位。	SSD1306 数据手册 34 页中 10.1.1 章和 10.1.2 讲解
0x40	设置起始线地址	SSD1306 数据手册 36 页中 10.1.6 讲解
0x81 0xFF	设置 OLED 亮度	SSD1306 数据手册 36 页中 10.1.7 讲解
0xA1	设置 Segment 重定向	SSD1306 数据手册 36 页中 10.1.8 讲解
0xA6	设置正常显示, 不反转显示	SSD1306 数据手册 37 页中 10.1.10 讲解
0xA8 0x3F	设置多态比 设置 63 (0x3F)	SSD1306 数据手册 37 页中 10.1.11 讲解
0xA4	设置输出根据 GDDRAM 内容	SSD1306 数据手册 37 页中 10.1.9 讲解
0xD3	设置显示偏移	SSD1306 数据手册 37 页中 10.1.15 讲解
0xAF	开启 OLED 显示	SSD1306 数据手册 37 页中 10.1.12 章讲解

12.6 课程总结

OLED 是 RoboMaster 比赛中常见的显示设备, 通过 OLED 显示可以查看机器人的状态和各种信息, 方便对机器人进行检查。本章学习到 OLED 在通信过程中, 使用数据类型字节作为区分控制指令和数据指令。这种在数据层上重新定义协议的方式, 也常在其他协议中体现, 例如 RoboMaster 裁判系统串口协议中, 便定义帧头, CMD_ID, 数据长度等, 其中 CMD_ID 类似 OLED 的这个字节, 指明数据的类型。

13. BMI088 传感器

13.1 知识要点

- 陀螺仪简介
- 加速度计简介
- SPI 协议简介
- BMI088 寄存器介绍
- BMI088 读取函数介绍

13.2 课程内容

本节课将介绍 BMI088 传感器的相关知识，BMI088 传感器是一个六轴惯性测量单元 (IMU)，能够应用于机器人上的姿态解算，将在本章节学习六轴惯性测量单元的内部结构——三轴陀螺仪和三轴加速度计，惯性测量单元的功能及其基本原理，同时我们将介绍一个重要的通信协议——SPI 协议，以及 BMI088 传感器的使用方法，并编写工程进行 BMI088 传感器的驱动。

13.3 基础学习

13.3.1 陀螺仪简介

Bosch 公司生产的 BMI088 高性能惯性测量单元 (IMU) 专门设计用于无人机和机器人应用。该款 6 轴传感器在 $3 \times 4.5 \times 0.95\text{mm}^3$ 小尺寸 LGA 封装中集成了 16 位 ADC 精度的三轴陀螺仪和三轴加速度计。

陀螺仪是测量角速度的传感器，是 IMU 的重要组成部分。陀螺仪能测量在三个正交方向上旋转的角速度，也可以用于估算在三个方向上的旋转角度。

陀螺仪有许多种类，不同的陀螺仪一般基于不同的工作原理，能够达到的精度也不一样，市场上最常用的微机电 (MEMS) 陀螺仪的基本原理是利用旋转时产生的科里奥利力引发电容的变化，从而将旋转的角速度转化为电信号。

13.3.2 加速度计简介

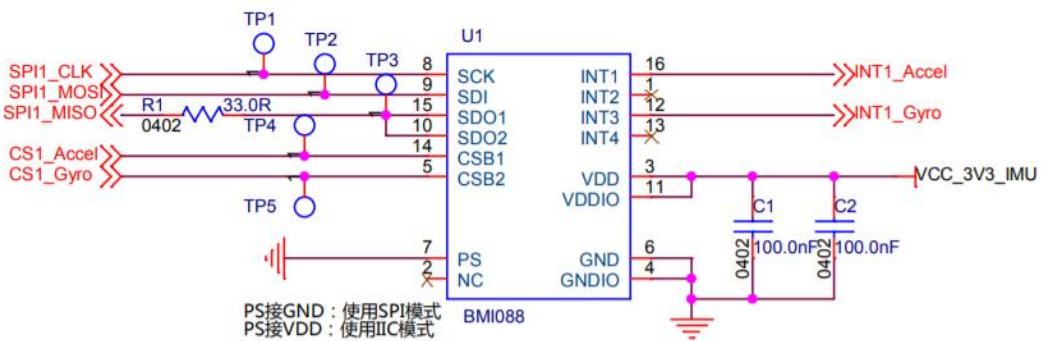
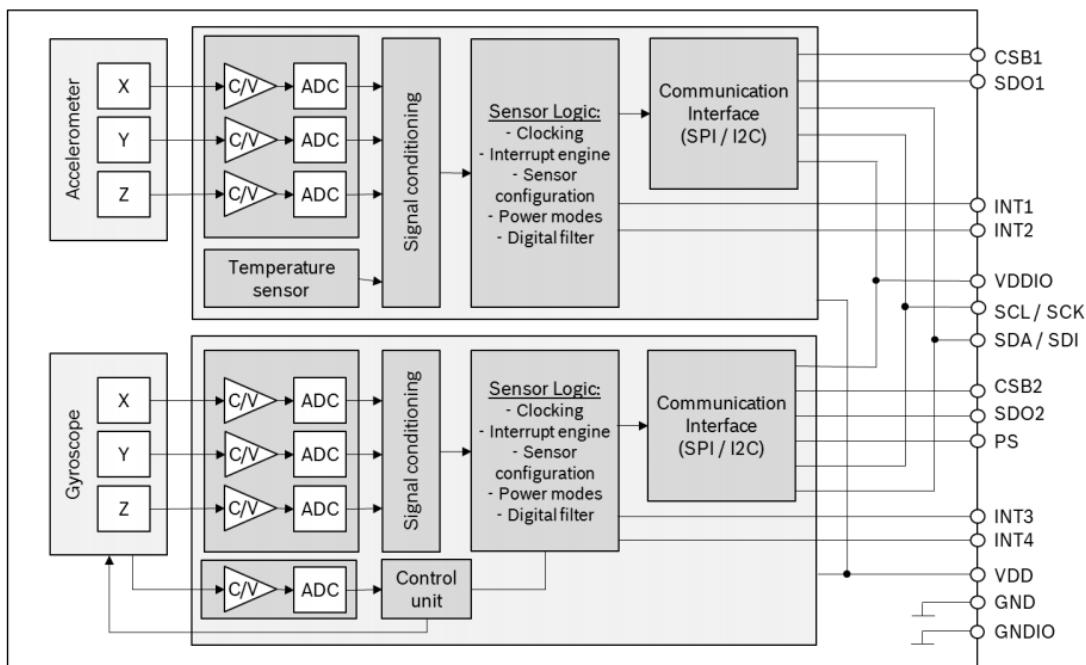
六轴 IMU 中的另一重要组成部分是加速度计；顾名思义，加速度计能够测量三个正交方向

上的加速度。MEMS 加速度计原理是利用加速度变化使内部质量块产生的力发生变化，从而改变电容大小，转化为电信号。

当物体静止时，加速度计测量重力加速度在三个正交方向上的分量，配合陀螺仪的角速度信息可以解算出物体的空间位姿，详细的解算过程可以参考第 18 章姿态解算任务章节。

13.3.3 SPI 协议简介

BMI088 支持 SPI 协议和 I2C 协议，通过 PS 引脚的电平状态决定是 SPI 协议或者 I2C 协议，BMI088 芯片内部框架图如下：



当 BMI088 使用 SPI 协议时，各个管脚的功能可见下表：

管脚	功能
CSB1, CSB2	连接 SPI 片选信号线, 低电平有效; CSB1 用于选中加速度计, CSB2 用于选中陀螺仪。
PS	模式选择引脚, 连接低电平时 BMI088 工作于 SPI 模式
SCK	连接 SPI 时钟线
SDI	数据输入 BMI088
SDO1	BMI088 输出加速度数据
SDO2	BMI088 输出角速度数据
INT1, INT2	发送加速度数据时产生中断信号
INT3, INT4	发送角速度数据时产生中断信号

SPI 协议是摩托罗拉公司开发的一种高速的, 全双工, 同步的通信总线, 使用四根线进行通信, 具有简单易用, 通讯速度高的特点。SPI 总线上可以挂载多个设备, 这些设备被区分为主设备 (**Master**) 和从设备 (**Slave**), 主设备通过时钟线和片选线对从设备进行控制。

SPI 协议所使用到的引脚及其功能见下表

名称	功能
SCK (Serial Clock)	SPI 是一种同步通信总线协议, 主设备通过 SCK 向各个从设备提供时钟信号
SDI (Serial Data Input) /MISO (Master In Slave Out)	SPI 的数据线之一, 传输方向为从设备发出数据, 主设备接收
SDO (Serial Data Output) /MOSI (Master Out Slave In)	SPI 的数据线之一, 传输方向为主设备发出数据, 从设备接收
SS (Slave Select) /CS (Chip Select)	SPI 的片选线, 主设备通过片选线控制从设备的工作状态, 选中需要通信的目标

SPI 是一种全双工的通信协议, 主设备和从设备通信时, 两端的收发是同步进行的, 即主设备和从设备在向对方发送数据的同时, 也在接收对方发来的数据。

SPI 的通信过程如下:

1. 主设备将要进行通讯的从设备的 SS/CS 片选拉低,
2. 主设备通过 SCK 向从设备提供同步通讯所需要的时钟信号,

3. 主设备通过 MOSI 向从设备发送 8 位数据，同时通过 MISO 接收从设备发来的 8 位数据。
4. 通信结束，主设备拉高 SS/CS 片选。

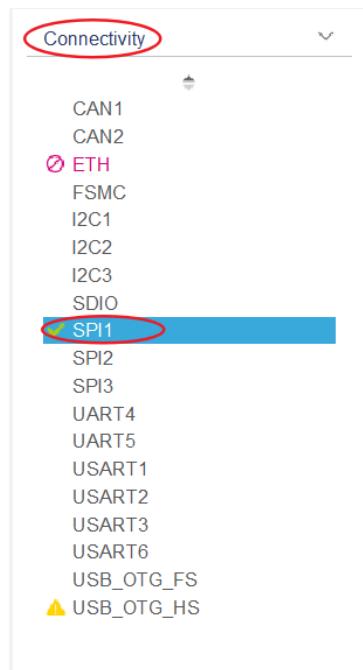
详细的 SPI 总线的连接方法和通信过程介绍可以见进阶学习部分。

13.4 程序学习

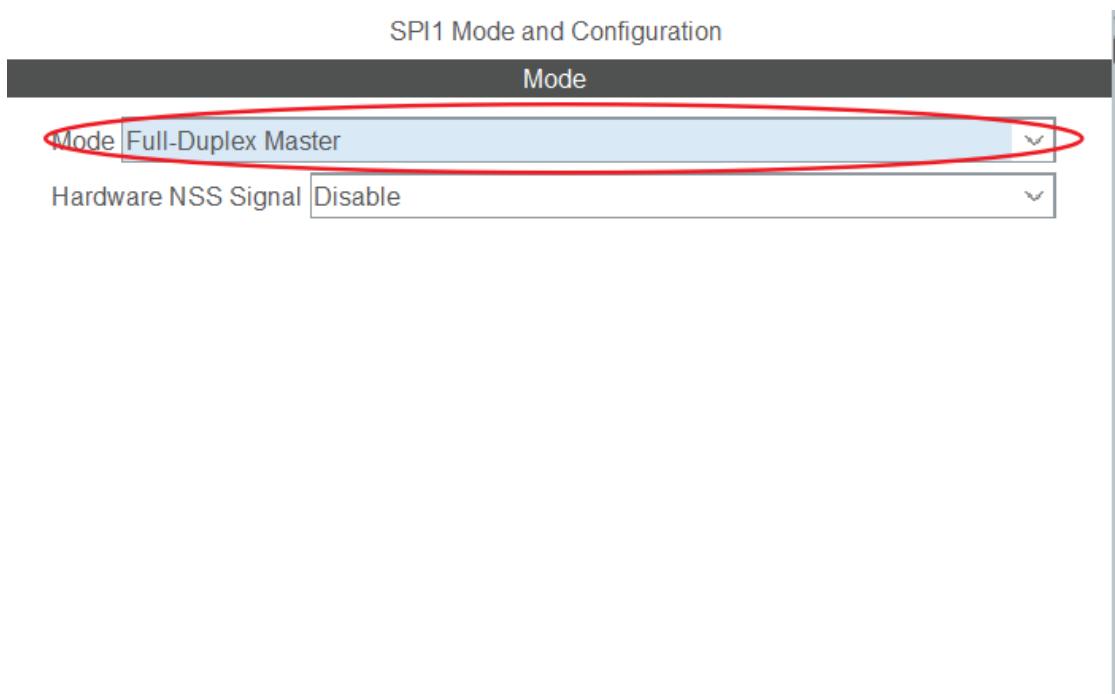
13.4.1 SPI 在 cubeMX 中的配置

本小节将学习 SPI 在 cubeMX 中的配置。

1. 首先在 Connectivity 标签页下选中 SPI1，进入 SPI1 的配置页面；



2. 在 Mode 页面中，将模式选择为 Full-Duplex Master，即让 stm32 工作在全双工 SPI 下，作为主机使用，将硬件片选信号 Hardware NSS Signal 设为 Disable；



3. 在 configuration 页面中，进行如下设置：

Configuration

Reset Configuration

NVIC Settings DMA Settings GPIO Settings

Parameter Settings User Constants

Configure the below parameters :

Search (Ctrl+F)

Basic Parameters

Frame Format	Motorola
Data Size	8 Bits
First Bit	MSB First

Clock Parameters

Prescaler (for Baud Rate)	256
Baud Rate	328.125 KBits/s
Clock Polarity (CPOL)	High
Clock Phase (CPHA)	2 Edge

Advanced Parameters

CRC Calculation	Disabled
NSS Signal Type	Software

配置页面中重要参数的含义可见下表：

参数	功能
Frame Format	设置 SPI 帧格式，可选 Motorola（摩托罗拉）格式和 TI（德州仪器）格式
Data Size	一帧中的数据长度，一般选为 8bit
First Bit	发送时先发最高位 MSB 还是最低位 LSB
Prescaler	总线分频值，设置 SPI 的通讯时钟频率
Clock Polarity & Clock Phase	用于设置 SPI 的时序功能
CRC Calculation	CRC 校验计算功能
NSS Signal Type	片选信号类型

13.4.2 BMI088 的寄存器简介

本小节将介绍 BMI088 中的重要寄存器。BMI088 中包含了用于控制陀螺仪，加速度计以及通讯接口的控制寄存器，存储传感器数据的数据寄存器以及存储传感器 ID 的寄存器。当要进行通讯时，首先读取 ID 寄存器，确认 ID 正确之后，先往控制寄存器中写入数据，设置传感器的工作状态，然后再从数据寄存器中进行数据的读取。

加速度计可以使用软件重置寄存器对所有寄存器的值进行恢复，对地址为 0x7E 的 ACC_SOFTRESET 寄存器写入 0xB6，就会将加速度计内所有寄存器恢复为默认值（一般为 0）。在初始化时，会通过软件重置寄存器进行重置。

0x7E	ACC_SOFTRESET	0x00	softreset_cmd (0xb6)
------	---------------	------	----------------------

同样的，陀螺仪初始化时，向地址为 0x14 的软件重置寄存器 GYRO_SOFTRESET 写入 0xB6 进行重置。

0x14	GYRO_SOFTRESET	N/A	softreset
------	----------------	-----	-----------

加速度计三轴的加速度值（3 个数据，每个数据长度为 16 位）分成高八位和第八位分别存储在地址从 0x12 到 0x17 的 6 个八位寄存器 ACC_Z_MSB 到 ACC_X_LSB 中。

0x17	ACC_Z_MSB	0x00	acc_z[15:8]
0x16	ACC_Z_LSB	0x00	acc_z[7:0]
0x15	ACC_Y_MSB	0x00	acc_y[15:8]
0x14	ACC_Y_LSB	0x00	acc_y[7:0]
0x13	ACC_X_MSB	0x00	acc_x[15:8]
0x12	ACC_X_LSB	0x00	acc_x[7:0]

陀螺仪三轴的速度值（3个数据，每个数据长度为16位）同样分成高八位和第八位分别存储在地址从0x02到0x07的6格八位寄存器中，为 RATE_Z_MSB 到 RATE_X_LSB。

0x07	RATE_Z_MSB	N/A	rate_z[15:8]
0x06	RATE_Z_LSB	N/A	rate_z[7:0]
0x05	RATE_Y_MSB	N/A	rate_y[15:8]
0x04	RATE_Y_LSB	N/A	rate_y[7:0]
0x03	RATE_X_MSB	N/A	rate_x[15:8]
0x02	RATE_X_LSB	N/A	rate_x[7:0]

此外加速度计内部还有存储温度值的数据寄存器，温度数据总长度为11位，前3位存储在八位的 TEMP_LSB 寄存器中的第六位到第八位，后8位存储在八位寄存器 TEMP_MSB 寄存器中。

0x23	TEMP_LSB	0x00	temperature[2:0]	-
0x22	TEMP_MSB	0x00	temperature[10:3]	

为了验证加速度计和陀螺仪工作正常，需要读取它们各自的 ID 寄存器，ID 寄存器内存有特定的 ID 值，用户可以将读取到的寄存器值和标准 ID 值进行对比。

加速度计的 ID 寄存器地址为 0x00，标准 ID 值为 0x1E，陀螺仪的 ID 寄存器地址为 0x00，标准 ID 值为 0x0F

0x00	ACC_CHIP_ID	0x1E	acc_chip_id
0x00	GYRO_CHIP_ID	0xF	gyro_chip_id

13.4.3 BMI088 读取函数介绍

本次的程序通过 BMI088_read 函数进行角速度和加速度数据的读取，该函数的功能包括了寄存器读取和数据拼接两部分。

读取加速度数据时，首先通过 BMI088_accel_read_multi_reg 函数，片选信号选中加速度计，

然后用 SPI 将加速度计数据寄存器中的数据读入 buf 中，按照高八位和第八位完成数据的拼接。

读取角速度数据时，首先通过 BMI088_gyro_read_muli_reg 函数，片选信号选中陀螺仪，然后用 SPI 将角速度计的 ID 和数据寄存器中的数据读入 buf。检测 ID 正确后，将数据按照高八位和低八位进行拼接。

读取温度数据时，读取加速度计中的温度寄存器的数据，并进行数据拼接。

```
void BMI088_read(fp32 gyro[3], fp32 accel[3], fp32 *temperate)
{
    uint8_t buf[8] = {0, 0, 0, 0, 0, 0, 0, 0};

    int16_t bmi088_raw_temp;

    BMI088_accel_read_muli_reg(BMI088_ACCEL_XOUT_L, buf, 6);

    bmi088_raw_temp = (int16_t)((buf[1]) << 8) | buf[0];
    accel[0] = bmi088_raw_temp * BMI088_ACCEL_SEN;
    bmi088_raw_temp = (int16_t)((buf[3]) << 8) | buf[2];
    accel[1] = bmi088_raw_temp * BMI088_ACCEL_SEN;
    bmi088_raw_temp = (int16_t)((buf[5]) << 8) | buf[4];
    accel[2] = bmi088_raw_temp * BMI088_ACCEL_SEN;

    BMI088_gyro_read_muli_reg(BMI088_GYRO_CHIP_ID, buf, 8);

    if(buf[0] == BMI088_GYRO_CHIP_ID_VALUE)
    {
        bmi088_raw_temp = (int16_t)((buf[3]) << 8) | buf[2];
        gyro[0] = bmi088_raw_temp * BMI088_GYRO_SEN;
        bmi088_raw_temp = (int16_t)((buf[5]) << 8) | buf[4];
        gyro[1] = bmi088_raw_temp * BMI088_GYRO_SEN;
        bmi088_raw_temp = (int16_t)((buf[7]) << 8) | buf[6];
        gyro[2] = bmi088_raw_temp * BMI088_GYRO_SEN;
    }
}
```

```

BMI088_accel_read_muli_reg(BMI088_TEMP_M, buf, 2);

bmi088_raw_temp = (int16_t)((buf[0] << 3) | (buf[1] >> 5));

if (bmi088_raw_temp > 1023)
{
    bmi088_raw_temp -= 2048;
}

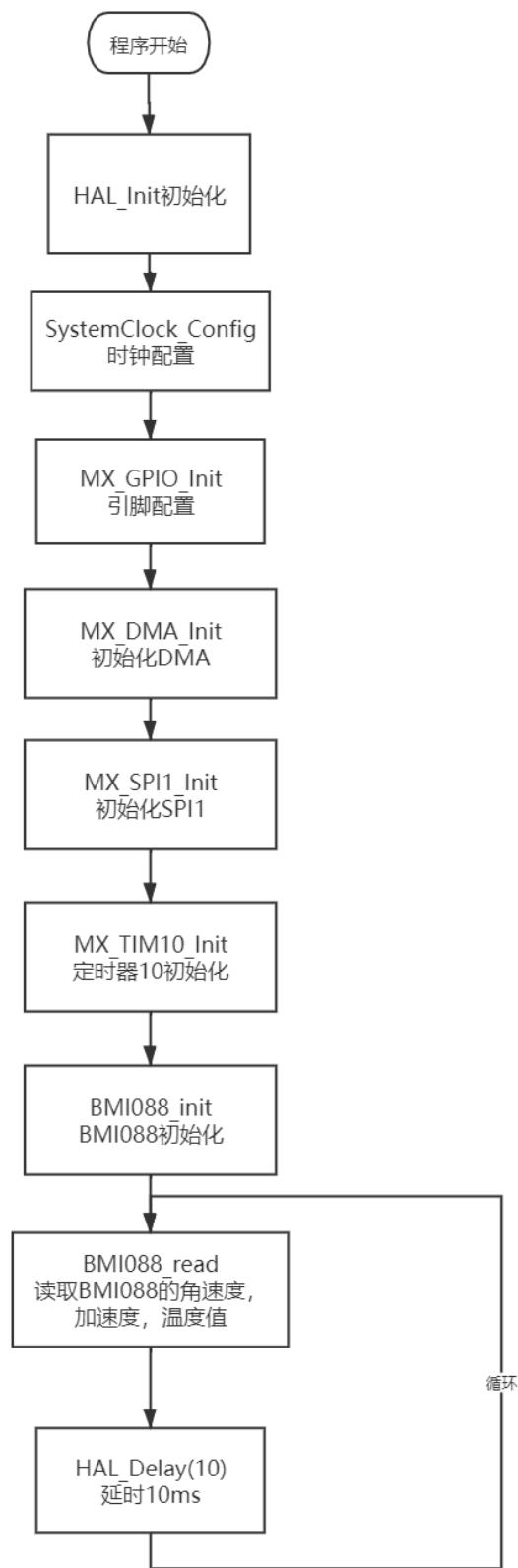
*temperate = bmi088_raw_temp * BMI088_TEMP_FACTOR + BMI088_TEMP_OFFSET;
}

```

其中 `BMI088_accel_read_muli_reg` 和 `BMI088_gyro_read_muli_reg` 调用了 HAL 库的 SPI 通信函数 `HAL_SPI_TransmitReceive`

HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t *pTxData, uint8_t *pRxData, uint16_t Size, uint32_t Timeout)	
函数名	<code>HAL_SPI_TransmitReceive</code>
函数作用	通过 SPI 进行主机和从机的通信
返回值	<code>HAL_StatusTypeDef</code> , HAL 库定义的几种状态, 如果本次 SPI 通信成功, 则返回 <code>HAL_OK</code>
参数 1	<code>SPI_HandleTypeDef *hspi</code> 即 SPI 的句柄指针, 如果是 SPI1 就输入 <code>&hspi1</code> , SPI2 就输入 <code>&hspi2</code>
参数 2	<code>uint8_t *pTxData</code> 待发送数据的首地址指针
参数 3	<code>uint8_t *pRxData</code> 接收数据的区域的首地址
参数 4	<code>uint16_t Size</code> 待发送的数据长度
参数 5	<code>uint32_t Timeout</code> 最大发送时长

13.4.4 程序流程



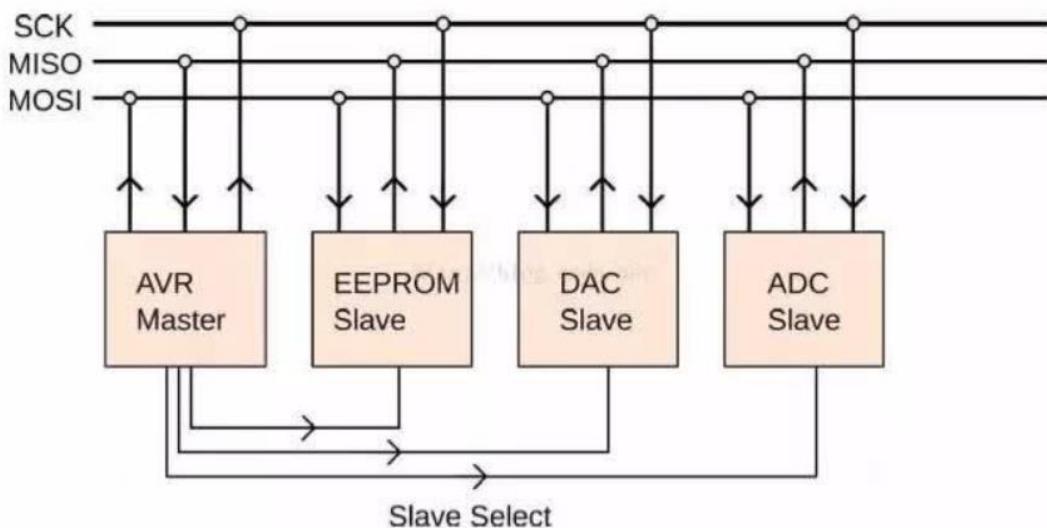
13.4.5 效果演示

下载程序进入 C 型开发板，进入 Debug 模式，在 watch 窗口查看角速度，加速度和温度数据，如图所示。

Name	Value	Type
gyro	0x20000040 gyro	float[3]
[0]	0.00639158674	float
[1]	0.00213052891	float
[2]	0.00319579337	float
accel	0x2000004C accel	float[3]
[0]	0.0888461545	float
[1]	-0.439743578	float
[2]	9.74525642	float
temp	28.875	float
<Enter expression>		

13.5 进阶学习

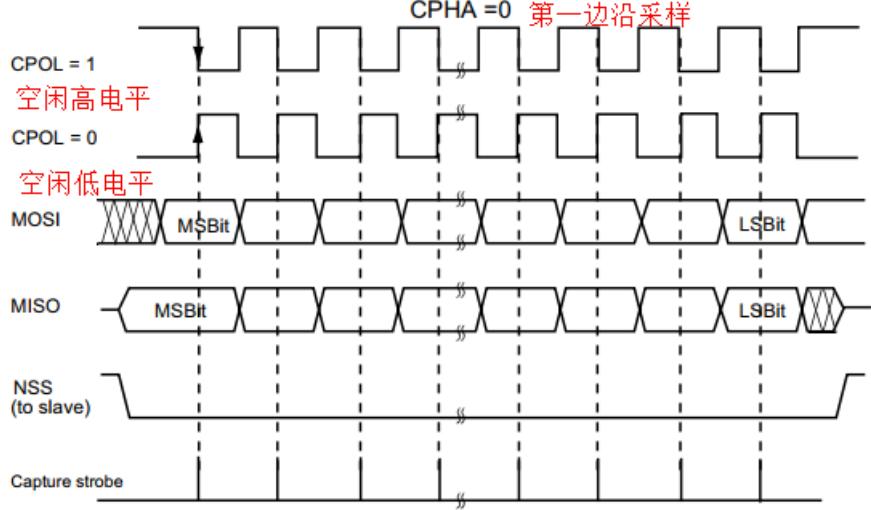
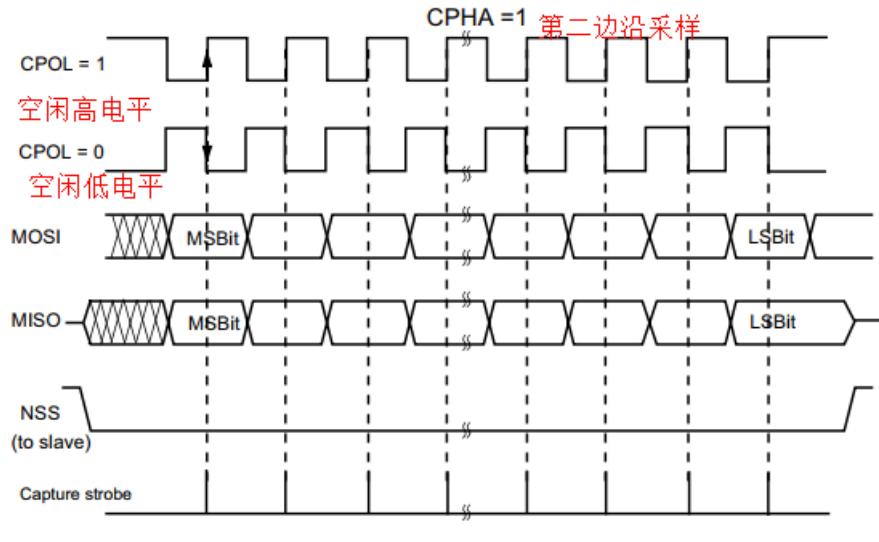
使用 SPI 进行通信时，各个设备的时钟引脚一起挂载在 SCK 线上，主设备的输入和各个从设备的输出连接 MISO/SDI，主设备的输出和各个从设备的输入连接 MOSI/SDO，同时主设备的 GPIO 口连接各个从设备的 CS。



SPI 的通讯过程开始时，如果一个主设备和一个从设备要进行通信，则主设备会通过 CS 信号线选中该从设备，其他未被选中的从设备不参与此次通信。

SPI 的时序规定了不同的工作模式，如下表所示：

工作模式	配置值	模式
CPOL 空闲时钟电平	0	时钟空闲时为低电平
	1	时钟空闲时为高电平
CPHA 边沿采样位置	0	第一个边沿采样数据
	1	第二个边沿采样数据



当主设备和从设备通信时，双方内部同时使用一个移位寄存器进行数据的存储和发送，每当主设备（从设备）完成一位发送的同时，移位寄存器进行一位的移位，并将接收到的数据存储入移位产生的新空间内，这样双方就可以同步同时的完成数据收发。

这里以主机要将数据 10101010 发送给从机，从机要将数据 01010101 发送给主机为例，展示整个数据发送的过程，假设主机和从机移位寄存器移位方向向左。

次序	主机移位寄存器	从机移位寄存器
0	10101010	01010101
1	01010100	10101011
2	10101001	01010110
3	01010010	10101101
4	10100101	01011010
5	01001010	10110101
6	10010101	01101010
7	00101010	11010101
8	01010101	10101010

可见到第 8 步时，主机数据要发送的数据全部进入从机的移位寄存器中，从机要发送的数据全部进入主机寄存器中，双方完成了此次的数据交换，此后主机取消对该从机的片选，继续和下一对象进行通信。

13.6 课程总结

陀螺仪模块以及加速度计模块是常用传感器，用于姿态解算以及云台速度环控制。本节课介绍了 BMI088 六轴 IMU 模块，并学习了如何通过 SPI 协议对其进行驱动并读取数据。

14. CAN 控制 RM 电机

14.1 知识要点

- CAN 协议简介
- RM 电机使用说明
- cubeMX 中的 CAN 波特率计算
- CAN 发送和接收中断介绍

14.2 课程内容

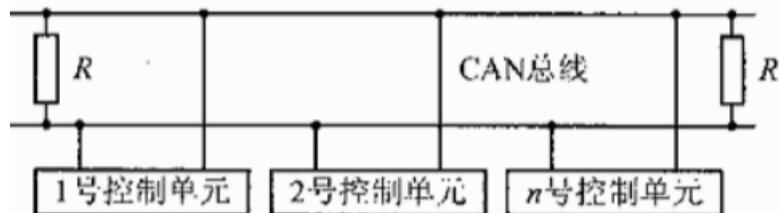
本节课将学习如何通过 CAN 通讯来控制 RM 电机，CAN 通讯是一种常见的现场总线通信方式，RoboMaster 大部分电机均是使用 CAN 通信进行控制的。

14.3 基础学习

14.3.1 CAN 协议简介

CAN 是控制器域网 (Controller Area Network, CAN) 的简称，是由研发和生产汽车电子产品著称的德国 BOACH 公司开发，并最终成为国际标准 (ISO11898)，CAN 是国际上应用最广泛的现场总线之一。在北美和西欧，CAN 总线协议已经成为汽车计算机控制系统和嵌入式工业控制局域网的标准总线，并且拥有以 CAN 为底层协议专为大型货车和重工机械车辆设计的 J1939 协议。

CAN 总线由 CAN_H 和 CAN_L 两根线构成，各个设备一起挂载在总线上。



RoboMaster 系列电机也采用 CAN 协议进行通信，CAN 协议比较复杂，一个完整的数据帧由下图中的各个部分组成：

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
------	-----	-----	-----	-----	-------	-----	-----	-----

这里重点介绍 CAN 的仲裁场和数据场的内容。和 I2C 总线一样，每一个挂载在 CAN 总线上的 CAN 都有一个自己独属的 ID，每当一个设备发送一帧数据时，总线其他设备会检查这个 ID 是否是自己需要接收数据的对象，如果是则接收本帧数据，如果不是则忽略。

ID 存储在数据帧最前头的仲裁场内，CAN 的 ID 分为标准 ID 和拓展 ID 两类，标准 ID 长度为 11 位。如果设备过多，标准 ID 不够用的情况下，可以使用拓展 ID，拓展 ID 的长度有 29 位。

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

在通过 ID 判断本帧数据可以接收后，控制场中的 DLC 规定了本帧数据的长度，而数据场内的数据的大小为 8 Byte，即 8 个 8 位数据。CAN 总线的一个数据帧中所需要传输的有效数据实际上就是这 8Byte。

控制场				数据场				CRC 场
其他	DLC (长度)			数据				
	Bit 3	Bit 2	Bit 1	Bit 0	Byte 7	...	Byte 1	Byte 0

14.3.2 RM 电机使用

使用 RM 系列电机时，在 RM 官网上下载电机和电调的数据手册，并在数据手册中查找和 CAN 通讯有关的内容，这里以 RM3508 电机为例，在 RM3508 的配套电调 C620 的数据手册中，可以找到如下内容：

标识符: 0x200 帧格式: DATA
帧类型: 标准帧 DLC: 8 字节

数据域	内容	电调 ID
DATA[0]	控制电流值高 8 位	1
DATA[1]	控制电流值低 8 位	
DATA[2]	控制电流值高 8 位	2
DATA[3]	控制电流值低 8 位	
DATA[4]	控制电流值高 8 位	3
DATA[5]	控制电流值低 8 位	
DATA[6]	控制电流值高 8 位	4
DATA[7]	控制电流值低 8 位	

这是电调接收报文格式，即如果要发送数据给 1 号到 4 号电调，控制电机的输出电流，从而控制电机转速时，需要按照表中的内容，将发送的 CAN 数据帧的 ID 设置为 0x200，数据域中的 8Byte 数据按照电调 1 到 4 的高八位和第八位的顺序装填，帧格式和 DLC 也按照表中内容进行设置，最后进行数据的发送。

而当要接收电调发送来的数据时，则按照下表进行：

标识符: 0x200 + 电调 ID

(如: ID 为 1, 该标识符为 0x201)

帧类型: 标准帧

帧格式: DATA

DLC: 8 字节

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

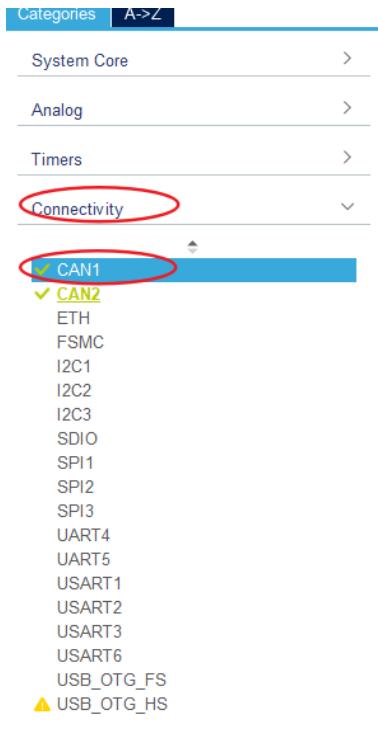
首先根据接收到的 ID 判断究竟接收到的是哪个电调发送来的数据, 手册中规定 1 号电调 ID 为 0x201, 2 号为 0x202, 3 号为 0x203, 4 号为 0x204。判断完数据来源之后, 就可以按照手册中的数据格式进行解码, 通过高八位和第八位拼接的方式, 得到电机的转子机械角度, 转子转速, 转矩电流, 电机温度等数据。

14.4 程序学习

14.4.1 CAN 在 cubeMX 中的配置

本小节将介绍 CAN 在 cubeMX 中的波特率的计算方法。

1. 首先在 cubeMX 中将 CAN1 开启, 打开 Connectivity 下的 CAN1, 进行 CAN1 的配置。

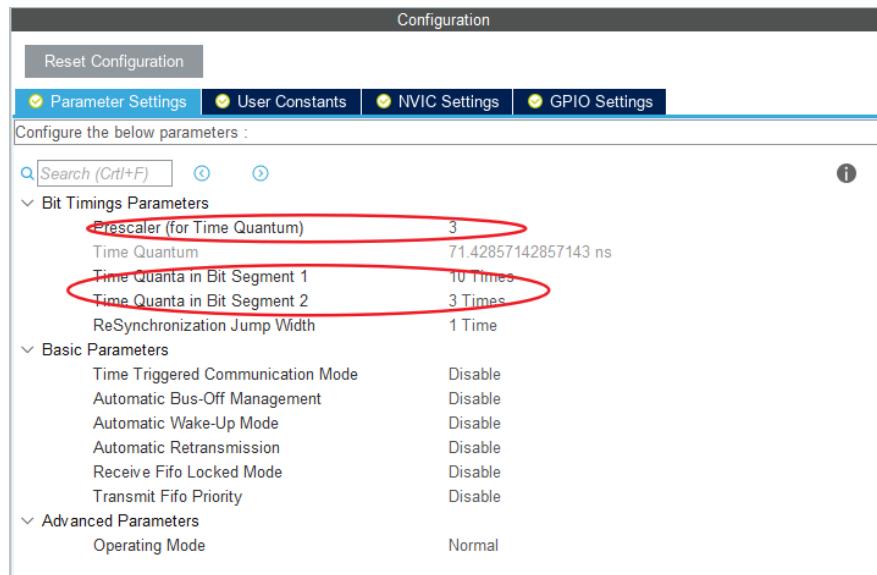


2. 在 Mode 中，将 Master Mode 选中打勾。



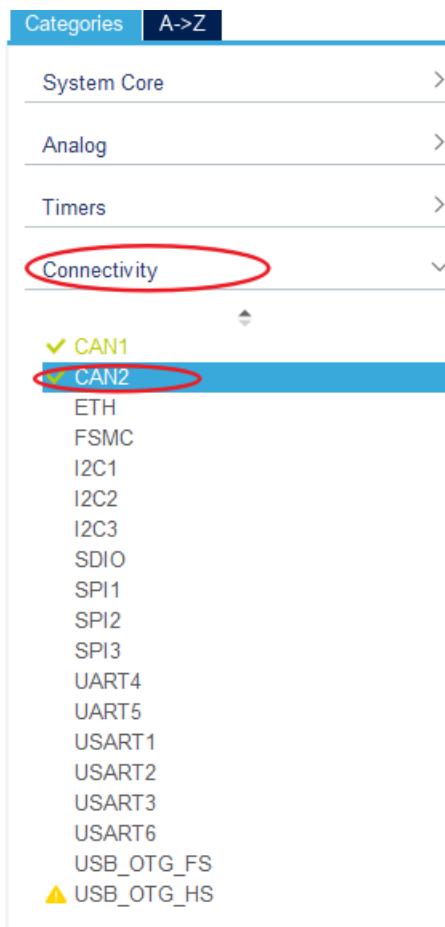
在 Configuration 界面中，需要进行 CAN 的波特率的配置，设置完分频系数（Prescaler）后，cubeMX 会自动完成 Time Quantum（简写为 tq）的计算，将 tq 乘以 tBS1 (Time Quanta in Bit Segment 1)，tBS2 (Time Quanta in Bit Segment 1)，RJW (ReSynchronization Jump Width) 之和刚好为 1 微秒，对应波特率为 1M，这是 CAN 总线支持的最高通讯速率。

$$71.42857142857143ns * (10 + 3 + 1) = 1000ns = 1\mu s$$



详细的 CAN 波特率计算原理见进阶学习部分。

1. CAN2 的配置和计算和 CAN1 类似。首先在 cubeMX 中将 CAN2 开启，打开 Connectivity 下的 CAN2，进行 CAN2 的配置。



2. 在 Mode 中，将 Slave Mode 选中打勾。



14.4.2 CAN 发送函数介绍

本小节将介绍 CAN 的发送函数。本程序中提供了 `CAN_cmd_chassis` 函数和 `CAN_cmd_gimbal` 函数，用于向底盘电机和云台电机发送 CAN 信号，控制电机运动。

`CAN_cmd_chassis` 函数的输入为电机 1 到电机 4 的驱动电流期望值 `motor1` 到 `motor4`，函数会将期望值拆分成高八位和第八位，放入 8Byte 的 CAN 的数据域中，然后添加 ID (`CAN_CHASSIS_ALL_ID 0x200`)，帧格式，数据长度等信息，形成一个完整的 CAN 数据帧，发送给各个电调。

```
void CAN_cmd_chassis(int16_t motor1, int16_t motor2, int16_t motor3, int16_t motor4)
{
    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);
}

```

HAL 库提供了实现 CAN 发送的函数 HAL_CAN_AddTXMessage

函数名	HAL_CAN_AddTXMessage
函数功能	将一段数据通过 CAN 总线发送
返回值	HAL_StatusTypeDef, HAL 库定义的几种状态, 如果本次 CAN 发送成功, 则返回 HAL_OK
参数 1	CAN_HandleTypeDef *hcan, 即 can 的句柄指针, 如果是 can1 就输入 &hcan1, can2 就输入 &hcan2
参数 2	CAN_TxHeaderTypeDef *pHeader, 待发送的 CAN 数据帧信息的结构体指针, 包含了 CAN 的 ID, 格式等重要信息
参数 3	uint8_t aData[], 装载了待发送的数据的数据名称
参数 4	uint32_t *pTxMailbox, 用于存储 CAN 发送所使用的邮箱号

`CAN_cmd_gimbal` 函数的功能为向云台电机和发射机构电机发送控制信号，输入参数为 yaw 轴电机，pitch 轴电机，发射机构电机的驱动电流期望值 yaw, pitch, shoot (rev 为保留值)，函数会将期望值拆分成高八位和第八位，放入 8Byte 的 CAN 的数据域中，然后添加 ID (CAN_GIMBAL_ALL_ID 0x1FF)，帧格式，数据长度等信息，形成一个完整的 CAN 数据帧，发送给各个电调。

```
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_box);
}
```

14.4.3 CAN 接收中断回调介绍

本小节将介绍 CAN 的接收中断回调，HAL 库提供了 CAN 的接收中断回调函数 `HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)`，每当 CAN 完成一帧数据的接收时，就会触发一次 CAN 接收中断处理函数，接收中断函数完成一些寄存器的处理之后会调用 CAN 接收中断回调函数。

在本次的程序中，在中断回调函数中首先判断接收对象的 ID，是否是需要的接收的电调发来的数据。完成判断之后，进行解码，将对应的电机的数据装入电机信息数组 `motor_chassis`

各个对应的位中。

```
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;
            }
    }
}
```

接收时调用了 HAL 库提供的接收函数 `HAL_CAN_GetRxMessage`

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

函数名	<code>HAL_CAN_GetRxMessage</code>
函数功能	接收 CAN 总线上发送来的数据
返回值	<code>HAL_StatusTypeDef</code> , HAL 库定义的几种状态, 如果本次 CAN 接收成功, 则返回 <code>HAL_OK</code>
参数 1	<code>CAN_HandleTypeDef *hcan</code> , 即 can 的句柄指针, 如果是 can1 就输入 <code>&hcan1</code> , can2 就输入 <code>&hcan2</code>
参数 2	<code>uint32_t RxFifo</code> , 接收时使用的 CAN 接收 FIFO 号, 一般为 <code>CAN_RX_FIFO0</code>
参数 3	<code>CAN_RxHeaderTypeDef *pHeader</code> , 存储接收到的 CAN 数据帧信息的结构体指针, 包含了 CAN 的 ID, 格式等重要信息
参数 4	<code>uint8_t aData[]</code> , 存储接收到的数据的数据名称

`motor_chassis` 为 `motor_measure_t` 类型的数组, 其中装有电机转子角度, 电机转子转速, 控制电流, 温度等信息。

```
typedef struct  
{  
    uint16_t ecd;  
    int16_t speed_rpm;  
    int16_t given_current;  
    uint8_t temperate;  
    int16_t last_ecd;  
} motor_measure_t;
```

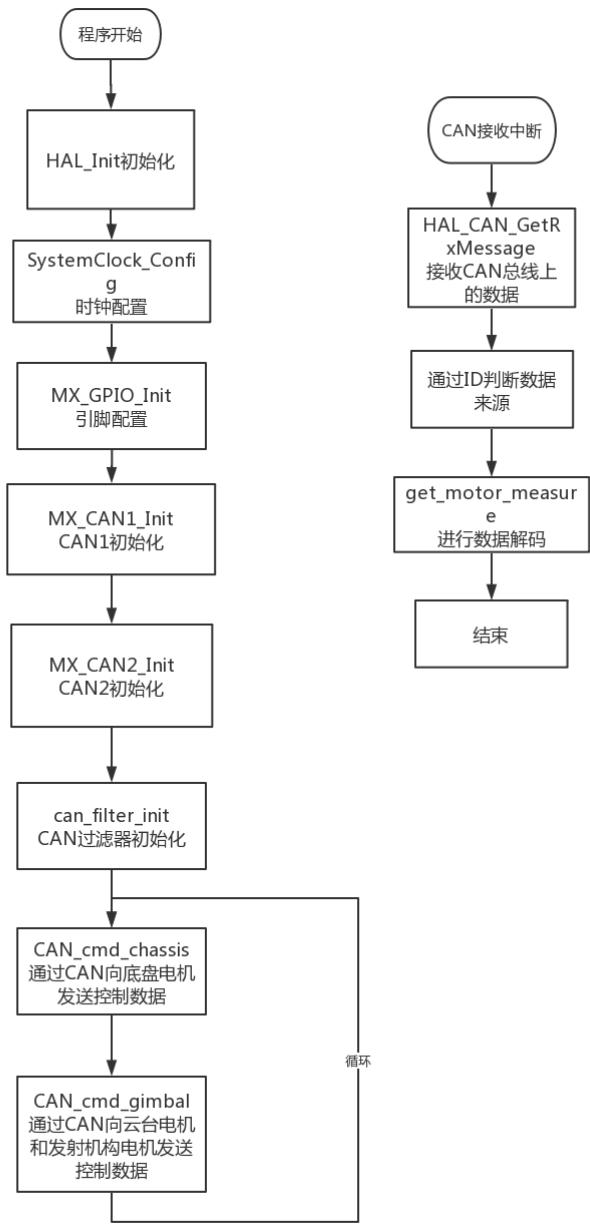
解码功能实际上完成的工作是将接收到的数据按照高八位和第八位的方式进行拼接, 从而得到电机的各个参数。

```
#define get_motor_measure(ptr, data)
```

\

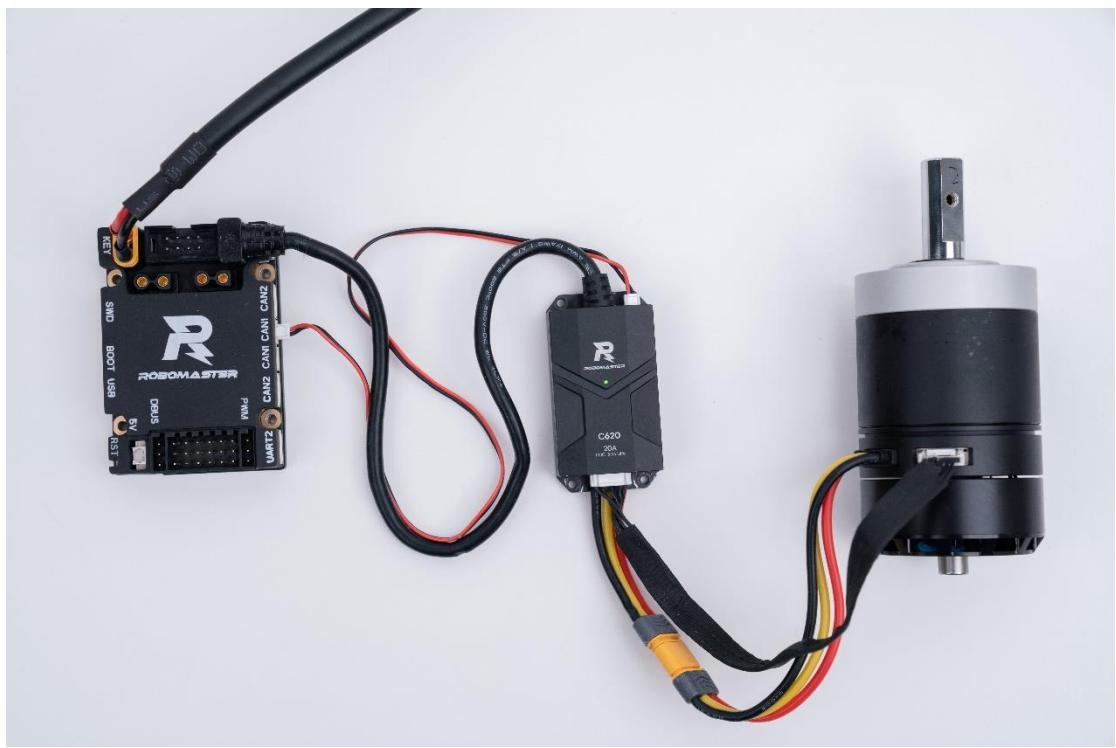
```
{\n    (ptr)->last_ecd = (ptr)->ecd;\n    (ptr)->ecd = (uint16_t)((data)[0] << 8 | (data)[1]);\n    (ptr)->speed_rpm = (uint16_t)((data)[2] << 8 | (data)[3]);\n    (ptr)->given_current = (uint16_t)((data)[4] << 8 | (data)[5]);\n    (ptr)->temperate = (data)[6];\n}\n
```

14.4.4 程序流程

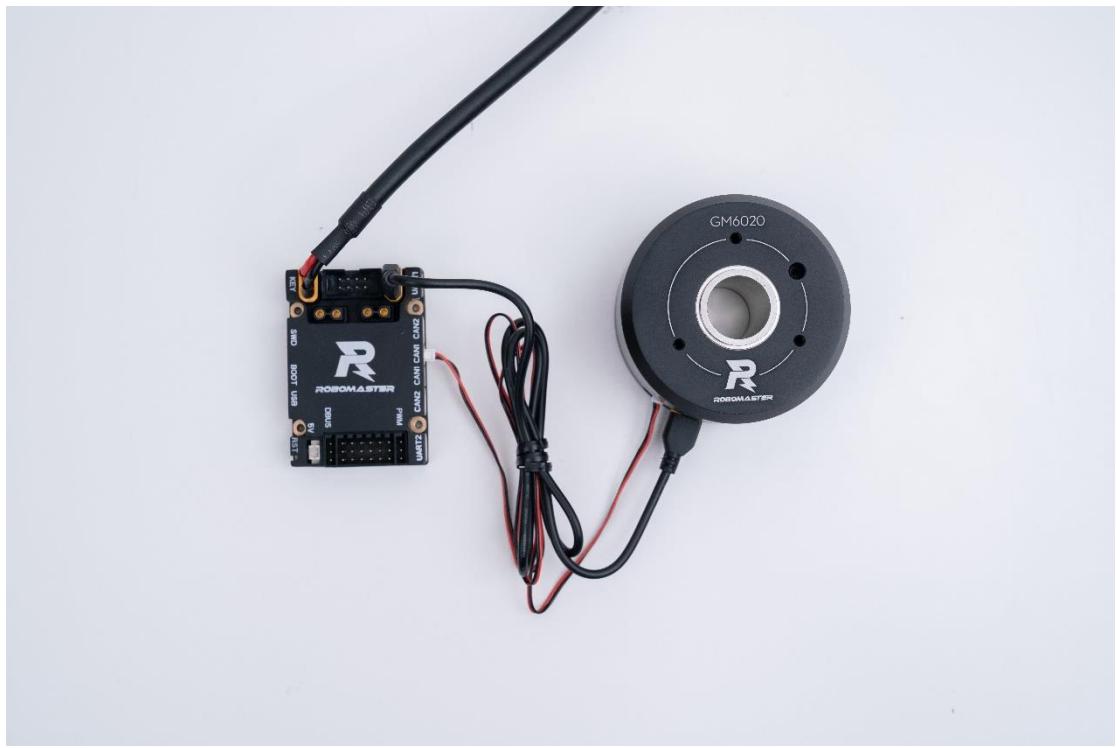


14.4.5 效果演示

可以使用遥控器对电机进行控制，整体接线如图所示。



开发板 C 型连接 M3508 电机

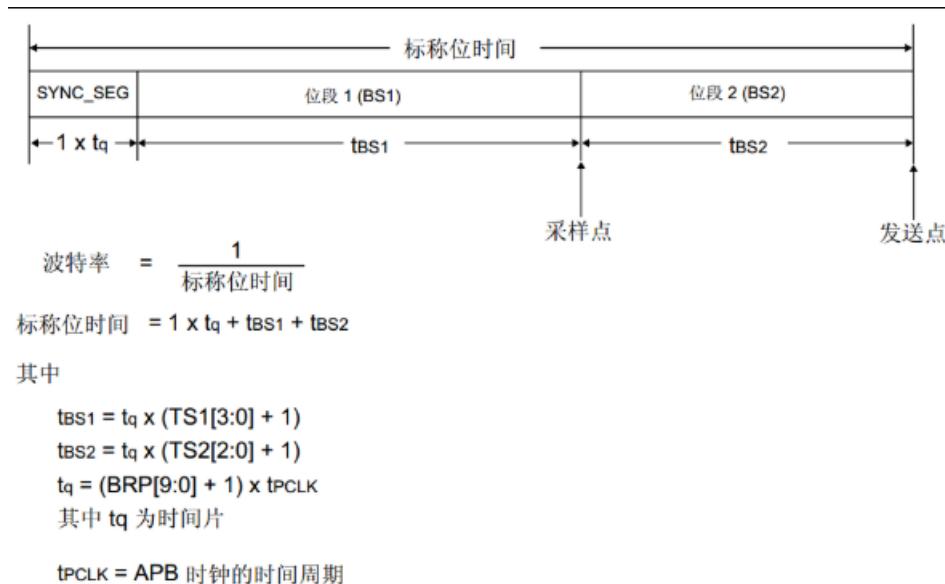


开发板 C 型连接 GM6020 电机

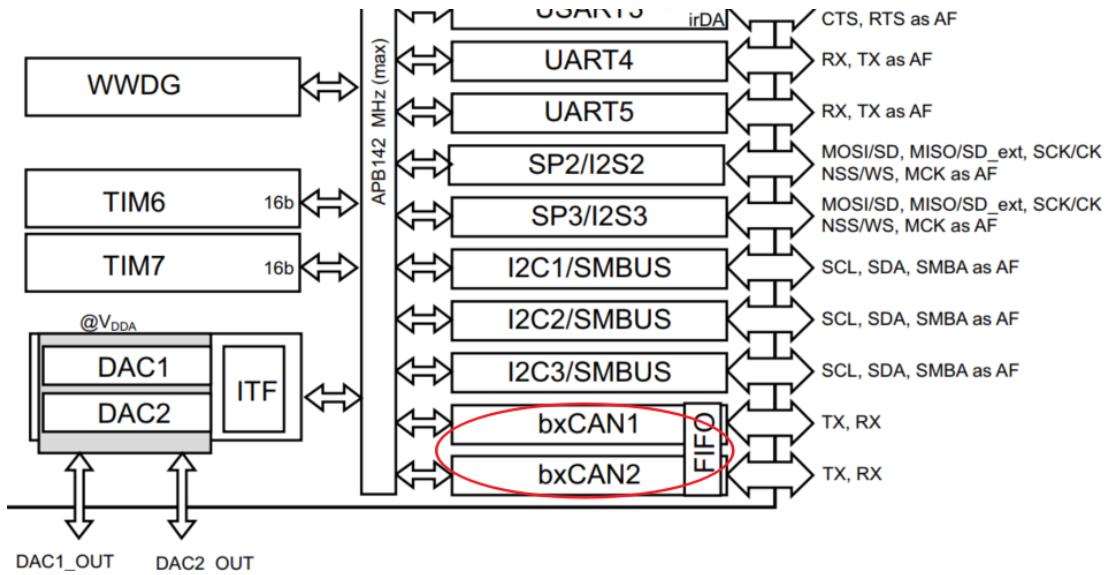
14.5 进阶学习

14.5.1 CAN 波特率介绍

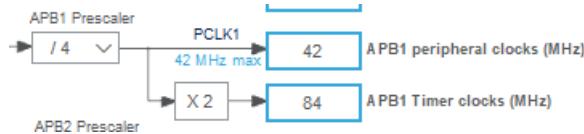
本节将学习 CAN 总线波特率的具体计算方法。CAN 总线波特率计算的原理可见下图：



根据上图，CAN 总线波特率和 tq (Time Quantum)， $tBS1$ (Time Quanta in Bit Segment 1) 和 $tBS2$ (Time Quanta in Bit Segment 1) 的值直接相关， tq 通过总线分频后直接得到， $tBS1$ 和 $tBS2$ 则通过 $TS1$ 和 $TS2$ 放大为 tq 的若干倍，需要注意的是下图中 $tBS1$ 和 $tBS2$ 的计算是通过 $tq \times (TS1+1)$ 和 $tq \times (TS2+1)$ 得到的，而在 cubeMX 中，我们配置的 Time Quanta in Bit Segment 的值对应的就是 $(TS1+1)$ 和 $(TS2+1)$ 。本小节以 CAN1 为例，演示 CAN 的波特率计算过程。首先通过数据手册可以知道 CAN1 和 CAN2 都挂载在 APB1 总线上。



在 cubeMX 的 Clock Configuration 中，设置 APB1 总线的外设频率为 42MHz



如下图所示，在 Configuration 中，将分频设置为 3，所以首先可以计算出 t_q 的值为：

$$t_q = \frac{1}{42MHz} = 71.42857142857143ns$$

将 TS1+1 设置为 10，TS2+1 设置为 3，通过 t_q ，TS1 和 TS2 计算出 tBS1 和 tBS2 了

$$t_{BS1} = t_q * (TS1 + 1) = 71.42857142857143 * 10 = 714.2857142857143ns$$

$$t_{BS2} = t_q * (TS2 + 1) = 71.42857142857143 * 3 = 214.28571428571428ns$$

可以计算出完整的标称位时间和波特率为

$$t = t_q + t_{BS1} + t_{BS2} = 1000ns$$

$$R_b = \frac{1}{t} = \frac{1}{1000ns} = 1Mbps$$

通过以上配置就可以配置出和电机通讯所需要的波特率 1Mbps。

14.5.2 直流电机介绍

直流电机能将直流电转换成机械能，根据有无电刷机构分为有刷直流电机和无刷直流电机。

直流电机的基本公式如下：

$$U = E_a + IR \quad (14-1)$$

$$E_a = C_e \phi n \quad (14-2)$$

$$T = C_t \phi I \quad (14-3)$$

$$C_t = 9.55 C_e \quad (14-4)$$

其中：

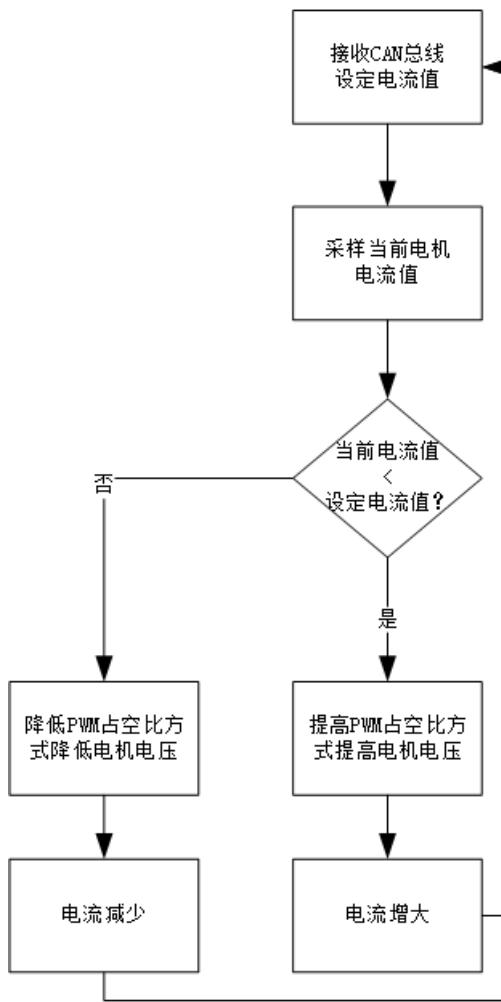
- U 为电机电枢两端电压，
- E_a 为转子中的感应电压，
- I 为电机电枢电流，
- R 为电机电阻，
- C_e 为电机的电势常数，
- Φ 为电机的磁通，
- n 为电机的转速，单位 RPM，
- T 为电机的扭矩，
- C_t 为电机的转矩常数。

整理可得：

$$n = \frac{U}{C_e \phi} - \frac{R}{C_e C_t \phi^2} T \quad (14-5)$$

可从 14-3 中得出电机扭矩与电机电流成正比，可从 14-5 中得出当负载扭矩一定时，电机转速与电机电压成正比。

在 RoboMaster 系列电机中，GM6020 电机是通过 CAN 总线控制电机电压，M3508 电机是通过 CAN 总线控制电机电流。故而当 M3508 电机接收到 CAN 总线发送的电流设定值后，会通过一套控制算法来保证电机电流恒定，整个流程如下图所示。



而 GM6020 电机接收到 CAN 总线设置的电压值后，会根据设定电压值调节电机电压。



M3508 电机比 GM020 电机相比，多了电流比较部分，通过一套控制算法调节电机电压，保证电机电流恒定，由于电机扭矩与电机电流成正比，进而控制电机扭矩恒定。M3508 在空载启动时，电机负载扭矩较小，即使设定较小的电流控制值，电机也会很快加速，直至电机

电压到达最大电压，之后电流逐渐减少，直至最大转速。

14.6 课程总结

CAN 通信是常见的现场总线通信方式，RM 电机均是使用 CAN 通信进行控制的。本节课我们学习了 CAN 协议的数据帧格式，波特率的计算和配置方法，并学习了如何使用 HAL 库的函数进行 CAN 的发送和接收，并与电调进行通信，从而控制 RM 电机的转动，并接收其反馈的数据。

15. freeRTOS 闪烁 LED

15.1 知识要点

- 操作系统简介
- cubeMX 中 freeRTOS 的配置
- 任务的创建方式
- 任务切换过程

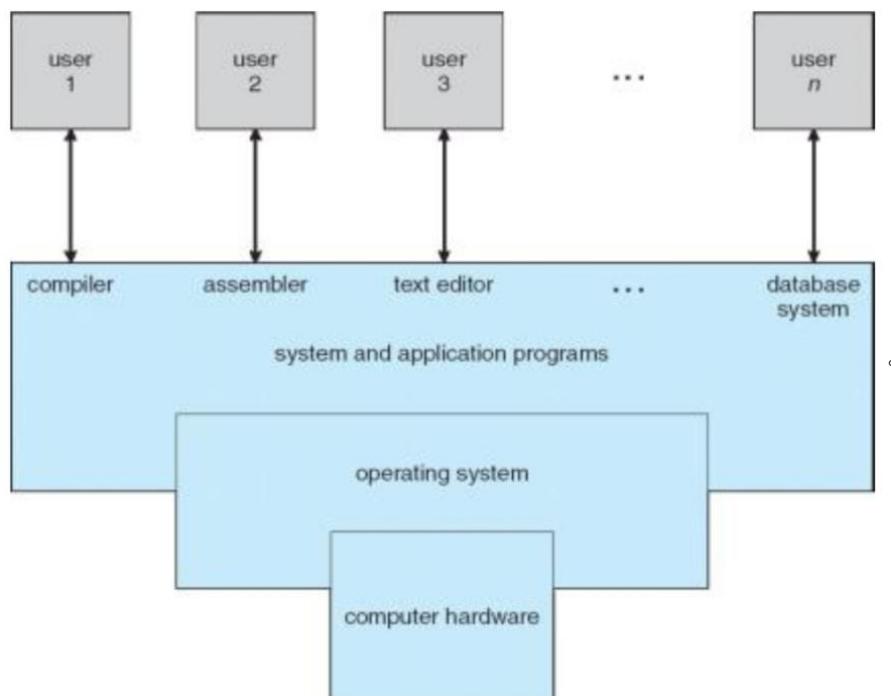
15.2 课程内容

本节课将嵌入式操作系统的基本概念，并介绍一个普及的嵌入式操作系统 freeRTOS，还将学习如何通过 cubeMX 进行 freeRTOS 的配置，并自己创建任务，实现 LED 闪烁效果。

15.3 基础学习

15.3.1 操作系统简介

操作系统 (Operating System) 的本质是一个帮助用户进行功能管理的软件。操作系统运行在硬件之上，为其他工作的软件执行资源分配等管理工作。



一般称呼不使用操作系统的单片机开发方式为“裸机开发”，当进行裸机开发时，需要自己设计循环，中断，定时等功能来控制各个任务的执行顺序。

而使用操作系统进行开发时，只需要创建任务，操作系统会自动按照一些特定的机制自动进行任务的运行和切换。

除了任务管理之外，操作系统还可以提供许多功能，比如各个任务之间的通信，同步，任务的堆栈管理，控制任务对重要资源的互斥访问等。

由于单片机的资源比较少，显然无法运行如 Windows, MacOS 等计算机操作系统，一般在单片机上运行的是经过专门设计的嵌入式实时操作系统 (RTOS)。

本节课要介绍的 freeRTOS 就是其中一种，其他比较常见的嵌入式实时操作系统还有 uCOSII, RTThread 等。freeRTOS 操作系统是完全免费的操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种单片机上运行，并且有着庞大的社区和生态。

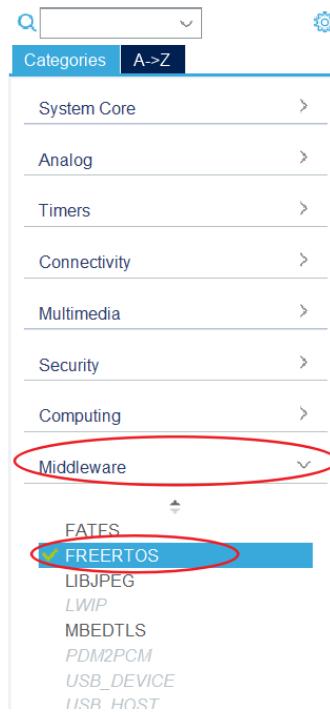
freeRTOS 对各个任务的切换调度需要遵循一定的规则，在进阶学习部分中有详细的介绍。

15.4 程序学习

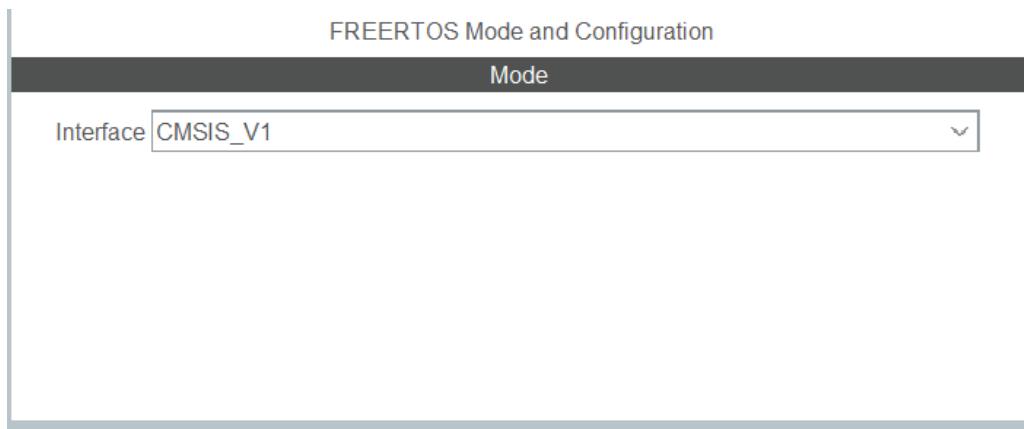
15.4.1 cubeMX 中 freeRTOS 的配置

本小节将学习如何通过 cubeMX 进行 freeRTOS 的配置。

首先打开 **Middleware** 标签，选中其中的 **FREERTOS** 选项，进入 FREERTOS 的配置页面。



在 Mode 页面下，选择 Interface 的版本，这里选择 CMSIS_V1。CMSIS 是由 Keil 提供的一套特殊的函数接口，他对 freeRTOS 的功能函数进行了封装，使其变得更加易用，在使用 freeRTOS 时，不需要再去直接调用 freeRTOS 的函数，只需要调用 CMSIS 为我们提供的函数即可，目前 CMSIS 有 V1 和 V2 两个版本。

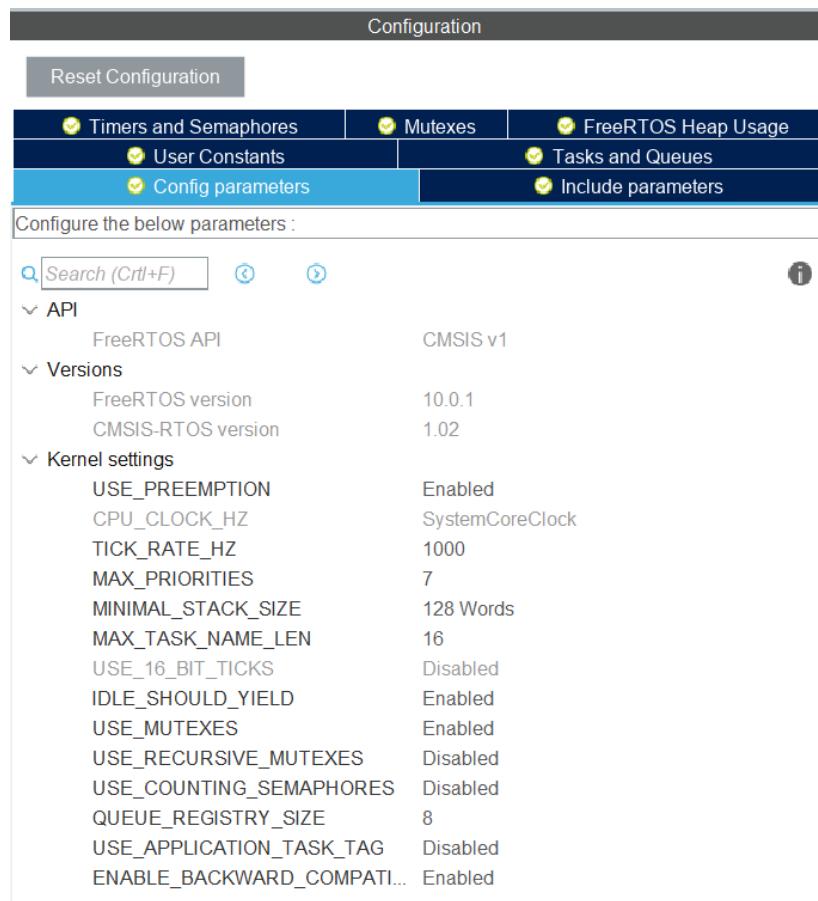


通过以上方式就可以完成 freeRTOS 的开启，在生成代码之后，freeRTOS 会自动被移植到工程中。接着在 Configuration 页面下进行 freeRTOS 的配置

在页面中，可以配置 freeRTOS 的一些重要的属性，包括是否支持抢占机制，freeRTOS 的系统时钟速率，最大优先级数量，最小任务栈尺寸，最大任务名称长度等。针对一些比较重要的配置列表如下：

名称	功能
USE_PREEMPTION	是否支持抢占机制，支持则设为 Enabled

TICK_RATE_HZ	系统时钟速率，时钟按照该速率为该 freeRTOS 中各个任务执行计时，设置为 1000Hz，则每个任务的最小调度时间为 1ms
MAX_PRIORITIES	最大优先级数量，默认为 7
MINIMAL_STACK_SIZE	最小任务栈大小，每当创建一个任务时，都需要为该任务分配一定大小的栈空间，任务需要使用的变量等都存储在该栈空间中。默认的最小值为 128 个字。
MAX_TASK_NAME_LEN	最大任务名称长度，在创建任务时，需要给每个任务起名作为标识，这个名称可以用一个字符串表示，本参数规定了字符串长度的上限值，默认为 16



15.4.2 cubeMX 中创建任务

本小节将学习如何在 cubeMX 中创建一个任务，首先在 freeRTOS 的配置页面中的 Configuration 下，选中 Tasks and Queues 标签页，存在一个已经创建的默认任务为“defaultTask”，点击进入配置选项修改为如下图所示。

Configuration

Reset Configuration

Tasks and Queues Timers and Semaphores Mutexes FreeRTOS Heap Usage
 Config parameters Include parameters User Constants

Tasks

Task Name	Priority	Stack Size...	Entry Func...	Code Gene...	Parameter	Allocation	Buffer Name	Control Blo...
defaultTask	osPriorityN...	128	StartDefaul...	Default	NULL	Dynamic	NULL	NULL

Add Delete

Queues

Queue Name	Queue Size	Item Size	Allocation	Buffer Name	Control Block Name
------------	------------	-----------	------------	-------------	--------------------

Edit Task

Task Name	LED_RED
Priority	osPriorityNormal
Stack Size (Words)	128
Entry Function	red_led_task
Code Generation Option	As weak
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

OK Cancel

如果需要增加一个新任务，点击 Add，就可以创建一个新的任务。

Configuration

Reset Configuration

Timers and Semaphores Mutexes FreeRTOS Heap Usage
 User Constants Tasks and Queues Include parameters
 Config parameters

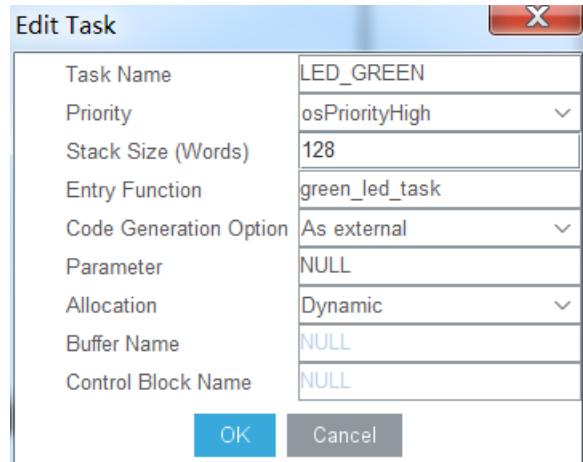
Tasks

Task Na...	Priority	Stack Si...	Entry Fu...	Code G...	Parameter	Allocation	Buffer N...	Control ...
defaultT...	osPriorit...	128	StartDef...	Default	NULL	Dynamic	NULL	NULL

Add Delete

在弹出的页面中，可以配置任务名称，优先级，栈大小，入口函数等。各个参数的具体功能见下表：

名称	功能
Task Name	任务名称
Priority	任务创建时的优先级
Stack Size (Words)	任务栈的大小，默认单位为字
Entry Function	任务函数的入口
Code Generation Option	<p>任务函数代码生成方式，设置为 Default 则会产生一个普通的任务函数，</p> <p>As weak: 产生一个用 <code>_weak</code> 修饰符修饰的任务函数；</p> <p>As external: 产生一个外部引用的任务函数，用户需要自己实现该函数；</p> <p>Default: 产生一个默认格式的任务函数，用户需要在该函数实现功能。</p>



设置完毕之后点击 **OK**，就可以看到列表中多出了自己创建的任务。

Task Name	Priority	Stack Size ...	Entry Function	Code Generat...	Parameter	Allocation	Buffer Name	Control Block...
LED_RED	osPriorityNor...	128	red_led_task	As weak	NULL	Dynamic	NULL	NULL
LED_GREEN	osPriorityHigh	128	green_led_task	As external	NULL	Dynamic	NULL	NULL

在 `freertos.c` 中也可以找到修改的默认任务函数。

```
__weak void red_led_task(void const * argument)
{
    /* USER CODE BEGIN red_led_task */
    /* Infinite loop */
    for(;;)
    {
        osDelay(1);
    }
    /* USER CODE END red_led_task */
}
```

由于选择了通过`__weak`修饰符创建一个弱函数，可以在别处实现该任务函数。程序执行时会自动寻找到这个另外实现的任务函数。

```
void red_led_task(void const * argument)
{
    while(1)
    {
        HAL_GPIO_WritePin(LED_R_GPIO_Port, LED_R_Pin, GPIO_PIN_SET);
        osDelay(500);
        HAL_GPIO_WritePin(LED_R_GPIO_Port, LED_R_Pin, GPIO_PIN_RESET);
        osDelay(500);
        HAL_GPIO_WritePin(LED_R_GPIO_Port, LED_R_Pin, GPIO_PIN_RESET);
        osDelay(500);

    }
}
```

由于新建了任务“LED_GREEN”，并且设置为 As external，故而需要再在别处实现任务函数。程序执行时会自动寻找到实现的任务函数。

```
void green_led_task(void const * argument)
```

```
{  
    while(1)  
    {  
        HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_RESET);  
        osDelay(500);  
        HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_SET);  
        osDelay(500);  
        HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_RESET);  
        osDelay(500);  
    }  
}
```

15.4.3 程序中创建任务

本小节将学习如何通过程序来创建任务。

打开 `freertos.c` 文件，在 `MX_FREERTOS_Init` 中，找到创建两个进程的代码，分别是 `LED_RED` 和 `LED_GREEN`。可以看到要创建任务时，只需要调用 `osThreadDef` 和 `osThreadCreate` 即可，创建“`LED_BLUE`”任务。

```
/* Create the thread(s) */  
  
/* definition and creation of LED_RED */  
osThreadDef(LED_RED, red_led_task, osPriorityNormal, 0, 128);  
LED_REDHandle = osThreadCreate(osThread(LED_RED), NULL);  
  
/* definition and creation of LED_GREEN */  
osThreadDef(LED_GREEN, green_led_task, osPriorityHigh, 0, 128);  
LED_GREENHandle = osThreadCreate(osThread(LED_GREEN), NULL);  
  
/* USER CODE BEGIN RTOS_THREADS */  
/* add threads, ... */  
osThreadDef(LED_BLUE, blue_led_task, osPriorityHigh, 0, 128);  
led_blue_handle = osThreadCreate(osThread(LED_BLUE), NULL);
```

```
/* USER CODE END RTOS_THREADS */
```

首先介绍 `osThreadDef`, 实际上这不是一个函数, 而是一个由 CMSIS 提供的宏定义, 用于对要创建的任务进行设置

```
#define osThreadDef(name, thread, priority, instances, stacksz) \
extern const osThreadDef_t os_thread_def_##name
```

名称	<code>osThreadDef</code>
功能	对要创建的任务进行设置
参数 1	<code>name</code> , 要创建的任务的名称
参数 2	<code>thread</code> , 要创建的任务代码的入口名称
参数 3	<code>priority</code> , 要创建的任务的优先级
参数 4	<code>instances</code> , 任务下可以创建的线程的数量
参数 5	<code>stacksz</code> , 任务栈大小

接着通过 CMSIS 提供的 `osThreadCreate` 函数来创建任务

```
osThreadId osThreadCreate (const osThreadDef_t *thread_def, void *argument);
```

函数名称	<code>osThreadCreate</code>
函数功能	创建一个任务
返回值	<code>osThreadId</code> , 任务 ID, ID 是一个任务的重要标识, 当在创建完任务后需要执行修改这个任务的优先级, 或者销毁该任务时, 就需要调用任务 ID, 需要提前声明一个类型为 <code>osThreadId</code> 的变量在此处存储返回值。
参数 1	<code>const osThreadDef_t *thread_def</code> , 我们通过 <code>osThreadDef</code> 所设置的任务参数, 采用强制转换+任务名的方式进行输入, 比如在 <code>osThreadDef</code> 中设置任务名为 <code>LED_RED</code> , 则在此处输入 <code>osThread(LED_RED)</code>
参数 2	<code>void *argument</code> , 任务需要的初始化参数, 一般填为 <code>NULL</code>

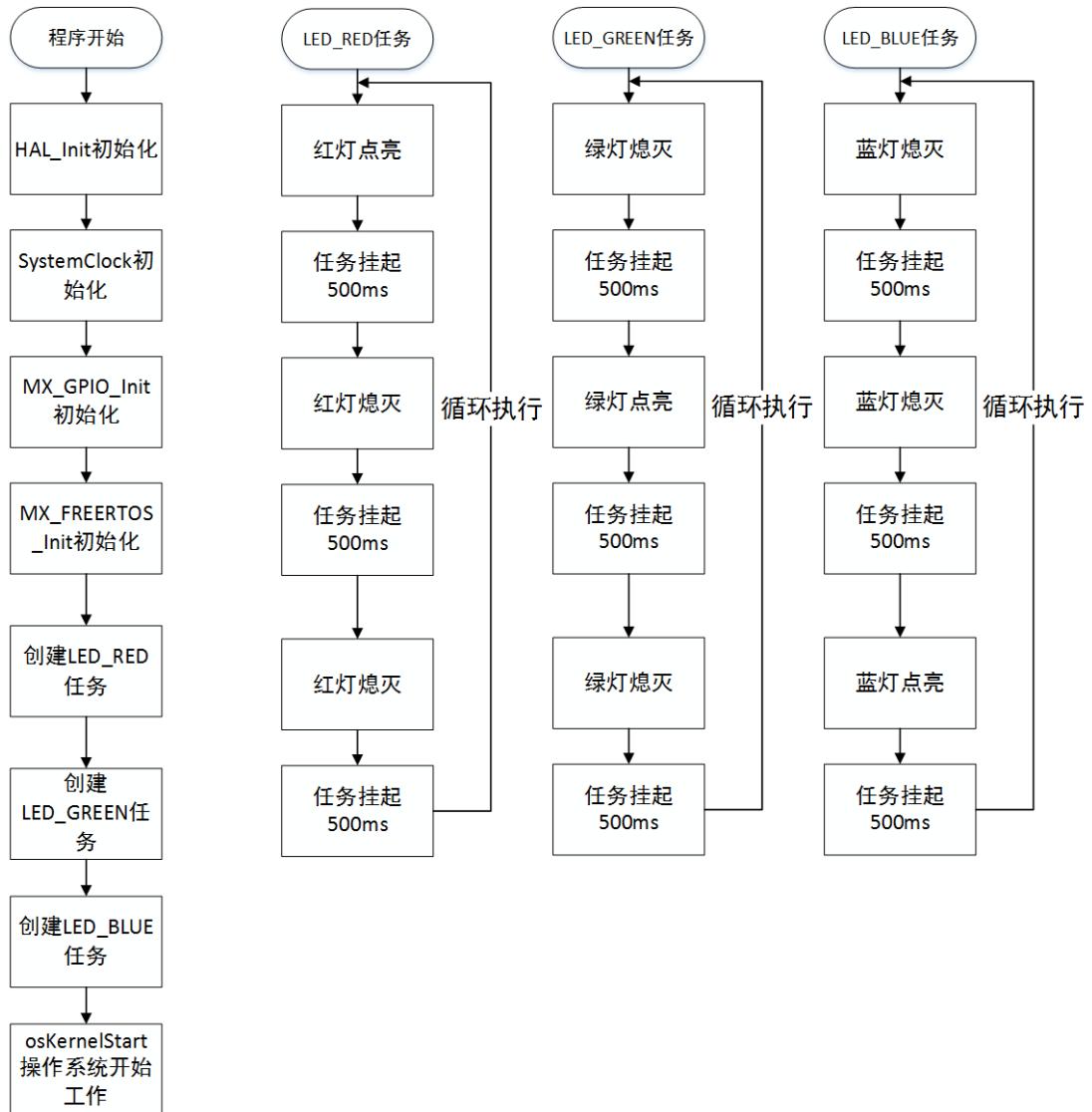
通过以上两步，一个任务就成功创建了，创建一个名称和 osThreadDef 中的 `thread` 参数一致的函数，操作系统会自动找到该函数并将其作为一个进程来执行。比如声明 `thread` 为 `blue_led_task`，则还需要执行函数 `void blue_led_task(void const * argument)`，`while(1)` 循环中的内容为用户自己的代码，这里是控制蓝色 led 灯闪烁。

```
void blue_led_task(void const * argument)
{
    while(1)
    {
        HAL_GPIO_WritePin(LED_B_GPIO_Port, LED_B_Pin, GPIO_PIN_RESET);
        osDelay(500);

        HAL_GPIO_WritePin(LED_B_GPIO_Port, LED_B_Pin, GPIO_PIN_RESET);
        osDelay(500);

        HAL_GPIO_WritePin(LED_B_GPIO_Port, LED_B_Pin, GPIO_PIN_SET);
        osDelay(500);
    }
}
```

15.4.4 程序流程



15.4.5 效果演示

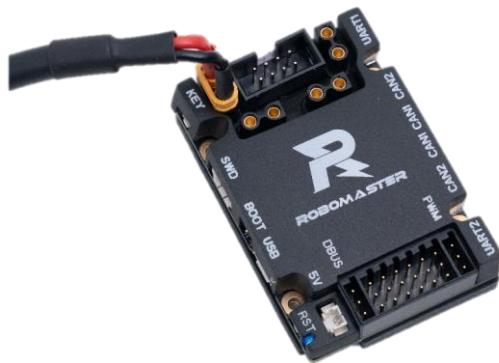
创建了三个任务，分别让三个 LED 依次闪烁，效果如图所示：



红灯点亮



绿灯点亮



蓝灯点亮

15.5 进阶学习

进阶学习部分将介绍操作系统中的任务切换机制。

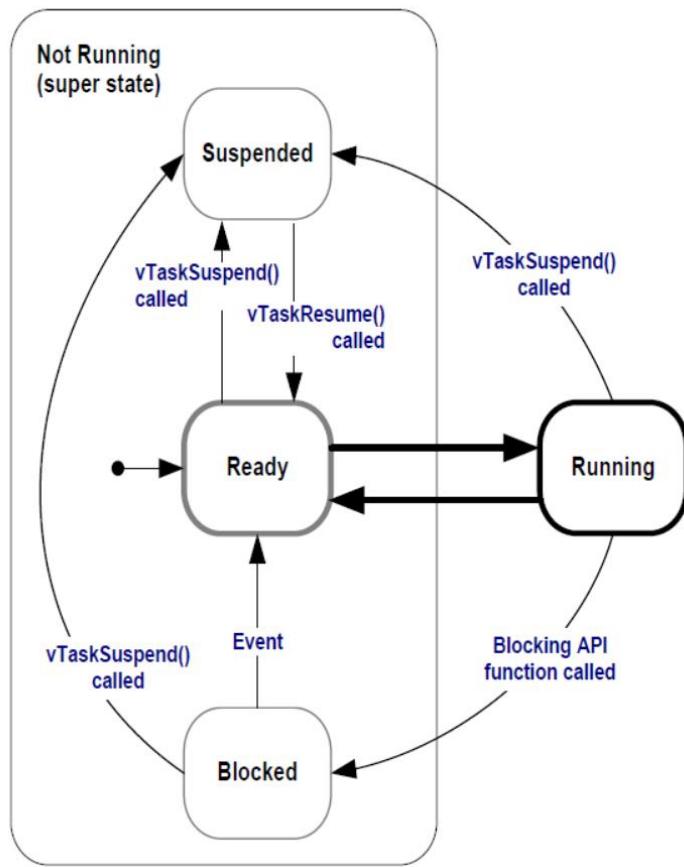
在操作系统，每一个要执行的任务，也就是一段程序的运行过程被称为一个进程。进程包含着动态的概念，它是一个程序的运行过程，而不是一个静态的程序。进程体现在程序中形式实际上就是一段循环执行的代码，以下图为例，`green_led_task` 是一个让绿色 led 灯闪烁的进程，使用操作系统的任务创建函数创建了这个进程之后，操作系统就自动找到这段代码并执行。

```
void green_led_task(void const * argument)
{
    while(1)
    {
        HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_RESET);
        osDelay(500);

        HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_SET);
        osDelay(500);

        HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_RESET);
        osDelay(500);
    }
}
```

一段程序执行时，一般划分成三个阶段，开始执行--->执行中--->执行完成。这也恰好对应了进程的工作状态：就绪态（Ready）--->运行态（Running）--->终止态（Blocked）。如果在一个进程执行的过程中，调用了将进程挂起的功能函数，或者是进程执行时有更高优先级的任务就绪了，则进程会进入挂起状态（Suspended）。



操作系统的一个重要工作就是执行各个进程的状态切换，因为实际上单片机每次只能运行一个进程，而操作系统通过适当的管理，让每一个进程都可以得到及时的响应，让多个进程呈现出一种同时运行的“并发”感。

操作系统执行任务切换时必须得要遵循一定的调度算法，操作系统会根据能否将正在运行的进程打断分为抢占式操作系统和合作式操作系统。

合作式操作系统不能够打断正在运行的进程，当多个进程就绪时，必须等待目前正在执行的进程结束，实现起来更加简单，但是降低了进程执行的实时性。抢占式操作系统可以将正在运行的进程打断，因此有着更加复杂的调度机制，但是也有更好的实时性。

本小节将操作系统的三种重要的调度算法，分别为：

- 先来先服务（FCFS）调度算法
- 优先级调度算法
- 时间片轮转调度算法。

首先介绍 FCFS 调度算法，该算法的原理非常简单，当存在多个任务时，先就绪的任务占有 CPU，直到任务执行结束，进入阻塞态后释放 CPU，此时下一个任务才可以占有 CPU。比

如下图中 3 个任务的就绪顺序为 P1, P2, P3:



而下图的任务就绪顺序为 P2, P3, P1:



接着介绍优先级调度算法，我们可以提前给各个任务赋予不同的优先级，当一个进程占有了 CPU 时，假如有一个更高优先级的任务就绪，则正在运行的任务会被挂起，转而执行高优先级的任务，知道高优先级任务执行完毕之后，再恢复原任务继续执行。

在 cubeMX 生成的 freeRTOS 中，每个任务默认有 7 个可选优先级，取值为 -3 到 3，值越大则优先级越高。

```
typedef enum {
    osPriorityIdle          = -3,           ///
```

最后介绍时间片轮转算法，为了避免某个任务执行时间过长，导致其他任务一直等待的情况出现，给每个任务划定一段特定长度的运行时间，称为时间片。如果任务执行时间超出了时间片，则该任务将被挂起，转而执行其他的任务。

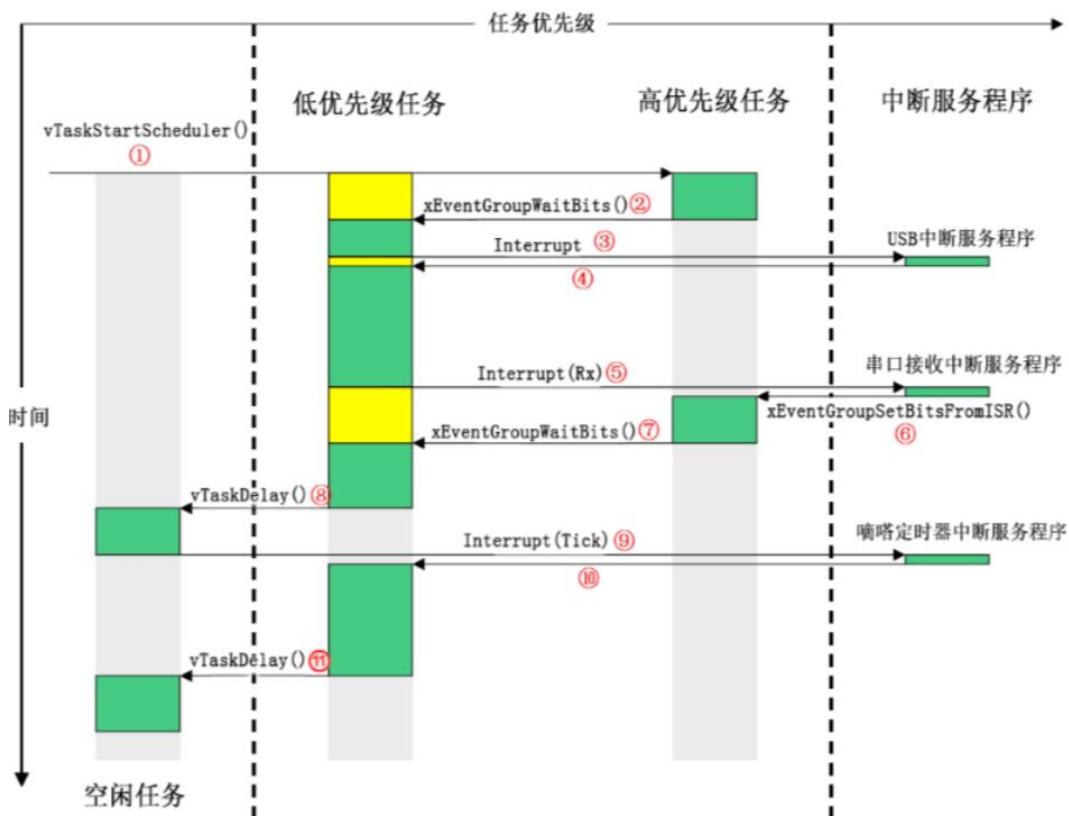
以下图为例，四个任务所需要执行的时长分别为 P1(53), P2(17), P3(68), P4(24) 时间片为 20。

任务	P1	P2	P3	P4	P1	P3	P4	P1	P3	P4
----	----	----	----	----	----	----	----	----	----	----

累 计 时 间 片	20	37	57	77	97	117	121	134	154	174
	未 完 成 P1	完 成 P2	未 完 成 P3	未 完 成 P4	未 完 成 P1	未 完 成 P3	完 成 P4	完 成 P1	未 完 成 P3	完 成 P3

当 freeRTOS 开始运行后，将同时遵从以上三个算法的原则对各个任务进行调度。对于不同优先级的任务采用优先级调度算法，对于同优先级的任务采用 FCFS 算法和时间片轮转算法。

需要注意的是 freeRTOS 中的任务，不论优先级，都是可以被中断给打断的。另外当需要在 freeRTOS 中进行延时操作时，可以通过将任务挂起一段时间的方式来实现，在该任务挂起时，其他任务可以占有 CPU，等到延时结束后，原先被挂起的任务又恢复执行。下图演示了 freeRTOS 中挂起，低优先级任务，高优先级任务和中断之间的切换过程。



15.6 课程总结

freeRTOS 是一款开源的嵌入式操作系统，使用可以很方便管理任务，进行多任务运行。本节课学习了嵌入式操作系统的基本原理以及任务管理机制，并学习和如何通过 cubeMX 创建一个包含 freeRTOS 的工程，以及通过 cubeMX 和程序编写两种方式来创建任务。

16. IMU 温度控制

16.1 知识要点

- IMU 控制温度的意义
- PID 控制简介
- 特殊的任务切换-唤醒任务

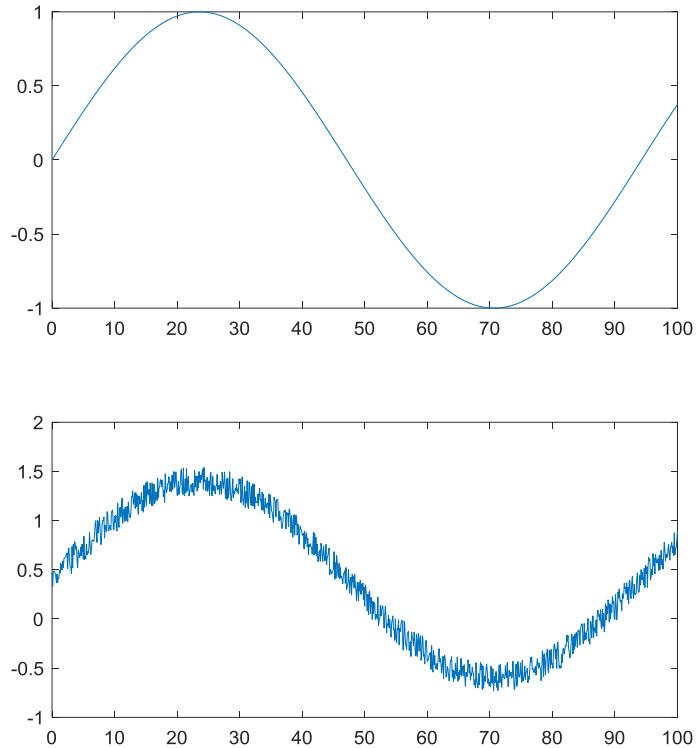
16.2 课程内容

本节课将学习使用 PID 控制算法对 IMU 进行温度控制，对 IMU 进行温度控制可以有效降低温度零漂。其中在文档控制中使用 PID 控制算法，而 PID 控制算法则是一种常见的控制算法，也将在这章进行简单的介绍。

16.3 基础学习

16.3.1 IMU 控制温度的意义

在第 13 章 BMI088 章节介绍过了 BMI088 由陀螺仪和加速度计两个传感器组合而成。传感器的基本功能便是把外界不同的物理量，化学量和生物量变换容易处理的电信号或者数字信号并输出。而由于实际上传感器中在放大，滤波，采样等过程不可避免地引入噪声，其中一个重要的噪声便是热噪声。热噪声是指由于电子热运动产生的随机电信号，如图所示，上图是一个理想的正弦信号，在实际传播过程中正弦信号叠加热噪声后会变成下图中带有杂波且 y 轴方向具有偏移的信号。



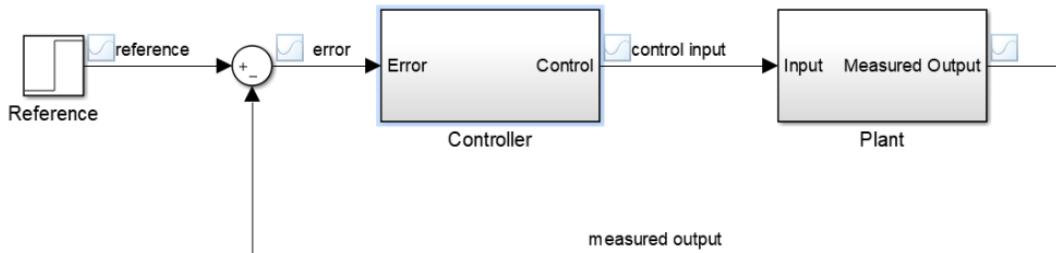
热噪声导致 IMU 数据偏移是 IMU 会产生零漂现象的重要原因之一。

零漂现象是指当物理量输入为零，传感器测量的输出量不为零的现象。即 IMU 没有任何运动，陀螺仪和加速度计也会读取到一定大小的数据，并将其当作是由 IMU 运动产生的。因此需要在 IMU 上电时首先测量出零漂的大小，将 IMU 测量到的值与零漂值相减，从而减少零漂的影响。这种方法的前提是 IMU 的零漂值保持固定，否则依然会引入误差。

热噪声的幅度和温度成正相关，所以零漂值受到温度的影响，因此需要控制 IMU 的温度保持一个恒定的范围内，从而减少零漂带来的影响。

16.3.2 PID 控制简介

PID 是一种常用的控制算法，其基本思想是利用期望值和实际值的误差作为控制量决定最终的输出。



如上图所示，一个控制过程分别为：

- 控制目标输入值 (reference)
- 作差得出误差值 (error)
- 控制器 (controller)
- 系统对象 (plant)
- 反馈值 (measured output)

而 PID 控制属于控制器的一种，由 P (比例) , I (积分) , D (微分) 三项构成。

PID 输出值 $U(t)$ 的表达式如下，其中 $err(t)$ 即误差值， K_p , K_i , K_d 分别为比例，积分，微分三项的系数。

$$U(t) = K_p * err(t) + K_i * \int err(t) dt + K_d * \frac{derr(t)}{dt}$$

比例项 K_p : 控制器比例项输出值和误差值保持线性关系，误差值放大一倍则输出值也同样放大一倍，误差值缩小一倍则输出值也同样缩小一倍。只依靠比例项进行控制的方法称为比例控制，比例控制可以很简单的实现控制器的基本功能，但往往存在静差以及过大引起系统振荡的问题。

积分项 K_i : 控制器积分项输出值与误差值的积分值成线性关系，即误差值的累计值乘以一个常数。积分项科研加速系统趋近设定值的过程，但积分增益过大容易引起积分超调的现象。

微分项 K_d : 微分项的大小和输出值的变化量成正相关，微分项计算误差的一阶导数，并和一个常数相乘，得到微分项的输出值。微分项可以对系统的改变做出反应，对系统的短期改变很有帮助。

下表总结了 PID 控制器中各项的特点。

	作用	缺点
比例项 P	对误差信号进行放大或衰减，比例系数大小决定了控制作用的强弱。	增益过大可能会引起系统振荡，使稳定性变差，降低了系统的相对稳定性，增益过小控制效果不明显，反应迟钝，无法修正干扰的影响；并且比例项不能完全消除系统的稳态误差。
积分项 I	通过对误差累积的作用影响控制器的输出，并通过系统的负反馈作用减小偏差，只要有足够的时间，积分控制将能够消除稳态误差。	增益过大可能出现系统积分超调，导致调整时间过长，增益过小可能出现收敛过长，不能及时地克服干扰的影响。
微分项 D	能够反应出反应误差信号变化的速度，在误差刚刚出现时产生很大的控制作用，具有超前控制的作用，有助于减小调整时间，改善系统的动态品质；	增益过大容易引起系统振荡，导致稳定性下降；增益过小可能改善效果不明显；同时微分容易引入高频噪声，导致系统不稳定。

由于数字系统是离散的，在单片机中实现 PID 控制算法时，需要将 PID 控制器的输出表达式改写成离散形式，其具体的做法就是将输出 $u(t)$ 和误差 $e(t)$ 由函数改成数组 $u(k)$ 和 $e(k)$ ，积分换成求和，微分换成差分。离散化后的 PID 表达式如下：

$$u(k) = K_p * e(k) + K_i * \sum_{i=0}^k e(i) + K_d * [e(k) - e(k-1)]$$

PID 控制器可以分为增量式 PID 控制器和位置式 PID 控制器，上文介绍的都是位置式 PID 控制器，即误差值直接决定最后的输出，而增量式 PID 控制器则用误差值来控制每次输出的改变量 Δu

$$\Delta u(k) = u(k) - u(k-1)$$

其表达式为：

$$\Delta u(k) = K_p * [e(k) - e(k-1)] + K_i * e(k) + K_d * [e(k) - 2 * e(k-1) + e(k-2)]$$

相比位置式 PID，增量式 PID 控制有以下优点：

- 不需要累加计算累加，输出增量只和前两次误差采样值有关，参数更容易调节
- 每次只输出控制增量，故发生故障时产生的影响较小

使用增量式 PID 时需要记忆上一次的输出值，将上一次的输出值和增量相加才能得到本次

输出值。

16.4 程序学习

16.4.1 任务唤醒功能

在某些情况下，某些任务需要通过某些特定条件（比如中断或者是另一个任务执行完成）来触发执行。通常使用任务通知功能来完成，主任务在没有接到任务通知时会保持休眠，当满足触发条件时发出任务通知，等待触发的主任务接到任务通知后就会进入就绪状态等待一次执行，在执行完毕之后又进入休眠状态，直到下一次任务通知到来。

在实例程序中使用外部中断作为触发条件，在外部中断处理函数中发出任务通知，IMU 的温度控制任务 `imu_temp_control_task` 等待任务通知的唤醒。

freeRTOS 提供了从中断发出任务通知的函数 `vTaskNotifyGiveFromISR`

<code>void vTaskNotifyGiveFromISR(TaskHandle_t xTaskToNotify, BaseType_t *pxHigherPriorityTaskWoken)</code>	
函数名	<code>vTaskNotifyGiveFromISR</code>
函数功能	从中断处理函数中发出一个任务通知，唤醒等待中的任务
返回值	<code>void</code>
参数 1	<code>TaskHandle_t xTaskToNotify</code> , 被通知的任务的任务句柄
参数 2	<code>BaseType_t *pxHigherPriorityTaskWoken</code> , 用于保存是否有高优先级任务准备就绪。如果函数执行完毕后，此参数的数值是 <code>pdTRUE</code> ，说明有高优先级任务要执行，否则没有

可以通过 freeRTOS 提供的 `xTaskGetHandle` 函数来为等待唤醒的任务获取任务句柄，这个任务句柄需要和发送任务通知中的参数一致。

<code>TaskHandle_t xTaskGetHandle(const char *pcNameToQuery)</code>	
函数名	<code>xTaskGetHandle</code>
函数功能	获取任务句柄

返回值	TaskHandle_t, 任务句柄
参数 1	const char *pcNameToQuery, 需要获取句柄的任务的名称

这里获取任务名称的功能通过 freeRTOS 提供的函数 pcTaskGetName 实现

```
char *pcTaskGetName( TaskHandle_t xTaskToQuery )
```

函数名	pcTaskGetName
函数功能	获取任务名称
返回值	char *, 任务名称
参数 1	TaskHandle_t xTaskToQuery, 需要获取名称的任务的句柄, 如果为 NULL 则获取当前运行任务的名称

通过 freeRTOS 提供的 ulTaskNotifyTake 函数实现接收任务通知, 唤醒休眠任务的机制。

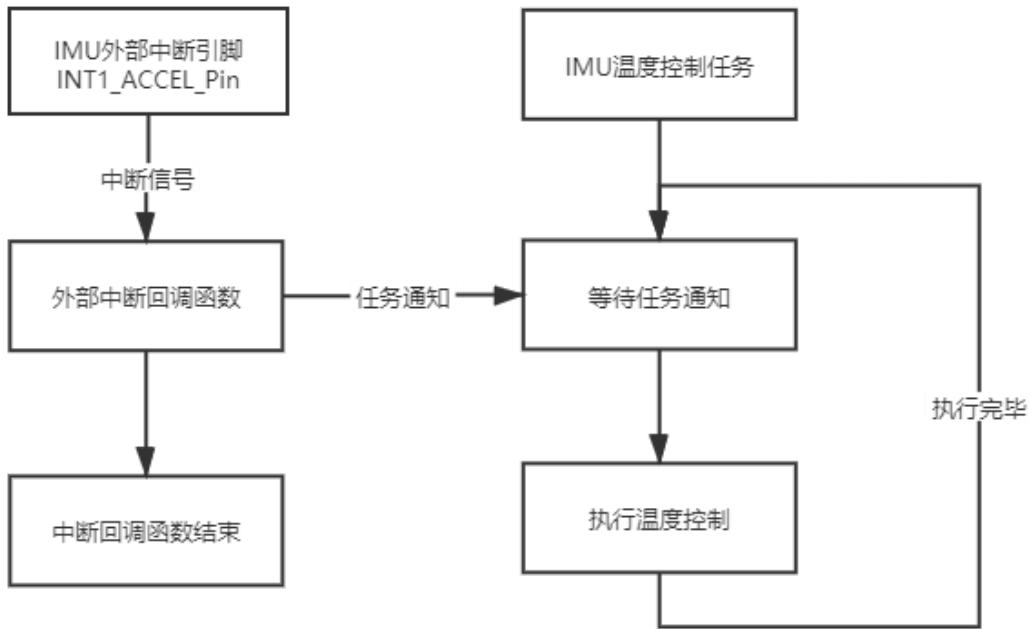
```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t xTicksToWait )
```

函数名	ulTaskNotifyTake
函数功能	接收任务通知, 唤醒休眠任务
返回值	任务通知值
参数 1	BaseType_t xClearCountOnExit , 选择是否清零用于任务通知的 ulNotifiedValue, 配置为 pdFALSE 表示函数返回前用于任务信号量的内部 变量 ulNotifiedValue 数值减一, 这种方式用于任务计数信号量。 参数配置 为 pdTRUE 表示函数返回前用于任务信号量的内部变量
参数 2	TickType_t xTicksToWait, 等待信号量可用的最大等待时间

16.4.2 控制链路

整个运行任务的控制链路为: IMU 的加速度计外部中断引脚 INT1_ACCEL_Pin 触发外部中断, 外部中断回调函数 HAL_GPIO_EXTI_Callback 中调用 vTaskNotifyGiveFromISR 函数发送任务通知, IMU 温度控制任务 imu_temp_control_task 接收到任务通知之后开始执行,

执行完毕之后进入休眠，等待下一次任务通知，流程图如下：



16.4.3 PID 初始化以及计算函数

本小节将介绍 PID 初始化及其计算函数。示例程序中通过 `PID_Init` 函数完成 PID 的初始化，`PID_Init` 函数的功能为设置 PID 控制器的模式，各项系数，最大输出值和积分上限，最后将 PID 控制器的输入和输出均初始化为 0。

```
void PID_init(pid_type_def *pid, uint8_t mode, const fp32 PID[3], fp32 max_out, fp32 max_iout)
{
    if (pid == NULL || PID == NULL)
    {
        return;
    }

    pid->mode = mode;
    pid->Kp = PID[0];
    pid->Ki = PID[1];
    pid->Kd = PID[2];
    pid->max_out = max_out;
    pid->max_iout = max_iout;
```

```

pid->Dbuf[0] = pid->Dbuf[1] = pid->Dbuf[2] = 0.0f;

pid->error[0] = pid->error[1] = pid->error[2] = pid->Pout = pid->Iout = pid->Dout =
pid->out = 0.0f;
}

```

而计算 PID 使用的 `PID_calc` 函数则首先区分 PID 的模式是位置式还是增量式，位置式则使用公式

$$u(k) = K_p * e(k) + K_i * \sum_{i=0}^k e(i) + K_d * [e(k) - e(k-1)]$$

增量式则使用公式：

$$\Delta u(k) = K_p * [e(k) - e(k-1)] + K_i * e(k) + K_d * [e(k) - 2 * e(k-1) + e(k-2)]$$

当前误差值 `e(k)` 使用期望值 `set` 与通过 IMU 内部的温度寄存器读取到的反馈值 `ref` 相减得到 `pid->error[0]` 来表示。程序中将 PID 控制器的输出 `pid->output` 作为施加给 IMU 的加热电阻的 PWM 值，输出值越大，加热电阻的温度越高。

```

fp32 PID_calc(pid_type_def *pid, fp32 ref, fp32 set)

{
    if (pid == NULL)
    {
        return 0.0f;
    }

    pid->error[2] = pid->error[1];
    pid->error[1] = pid->error[0];
    pid->set = set;
    pid->fdb = ref;
    pid->error[0] = set - ref;
    if (pid->mode == PID_POSITION)
    {
        pid->Pout = pid->Kp * pid->error[0];
        pid->Iout += pid->Ki * pid->error[0];
        pid->Dbuf[2] = pid->Dbuf[1];
    }
}

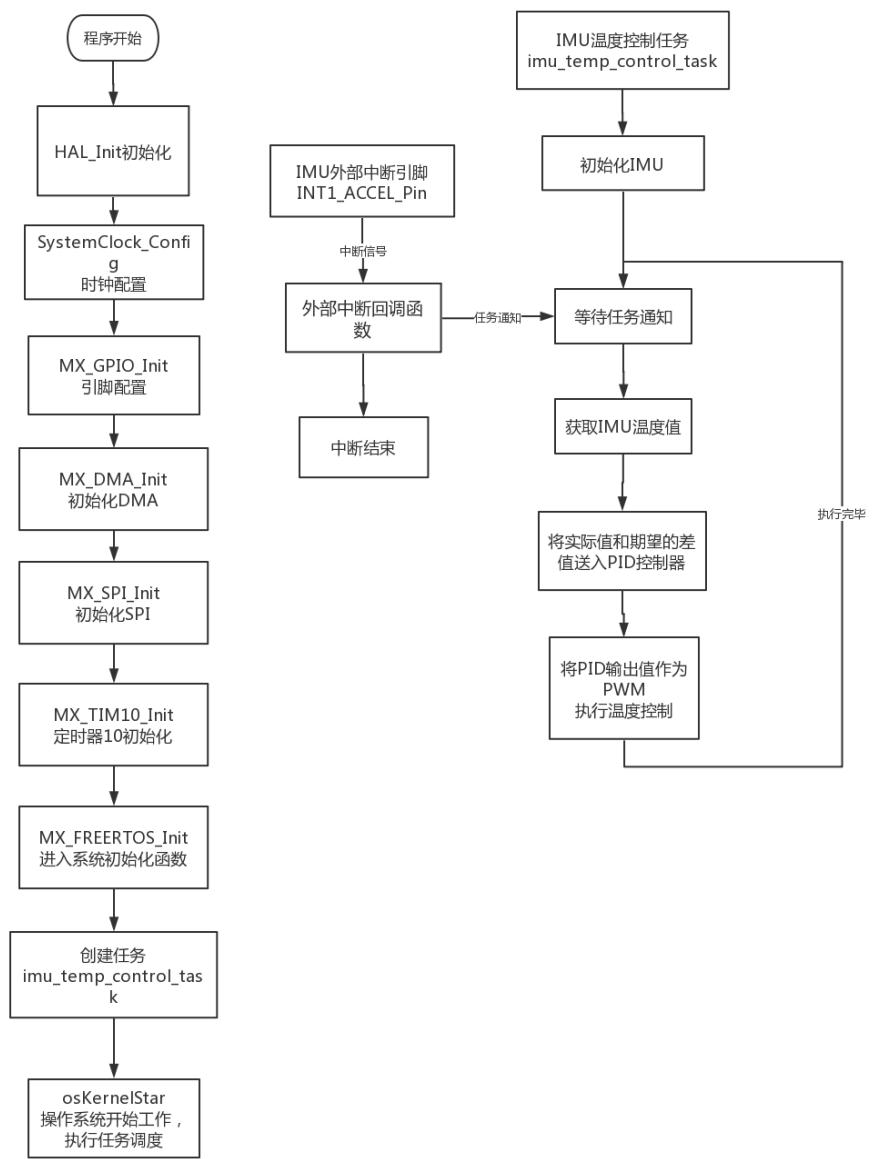
```

```
pid->Dbuf[1] = pid->Dbuf[0];
pid->Dbuf[0] = (pid->error[0] - pid->error[1]);
pid->Dout = pid->Kd * pid->Dbuf[0];
LimitMax(pid->Iout, pid->max_iout);
pid->out = pid->Pout + pid->Iout + pid->Dout;
LimitMax(pid->out, pid->max_out);
}

else if (pid->mode == PID_DELTA)
{
    pid->Pout = pid->Kp * (pid->error[0] - pid->error[1]);
    pid->Iout = pid->Ki * pid->error[0];
    pid->Dbuf[2] = pid->Dbuf[1];
    pid->Dbuf[1] = pid->Dbuf[0];
    pid->Dbuf[0] = (pid->error[0] - 2.0f * pid->error[1] + pid->error[2]);
    pid->Dout = pid->Kd * pid->Dbuf[0];
    pid->out += pid->Pout + pid->Iout + pid->Dout;
    LimitMax(pid->out, pid->max_out);
}

return pid->out;
}
```

16.4.4 程序流程



16.4.5 效果演示

使用热成像仪可以观察 IMU 的温度控制效果，如图所示：



16.5 进阶学习

在各种电子器件的实际使用中经常会碰到器件的实际输出与输入的期望值不一致，甚至相差很远的情况。以电机为例，假设我们输入电机的转速期望为 100rpm，当电机为空载时它的输出转速也恰好是 100rpm，但是当我们给电机加上轮子或者其他负载之后，它的转速就无法达到 100rpm 的期望值了，而是降低到 80rpm。假如电机转动了一段时间后电池电压开始下降了，电机的转速又会进一步的降低。

例子中的电机就是一个典型的无控制系统。在无控制系统中，输入期望和实际输出之间没有关联，两者之间可能会存在巨大的误差，从而导致系统可靠性很差。而控制系统则会根据实际的输出对输入进行调整，从小缩小误差。

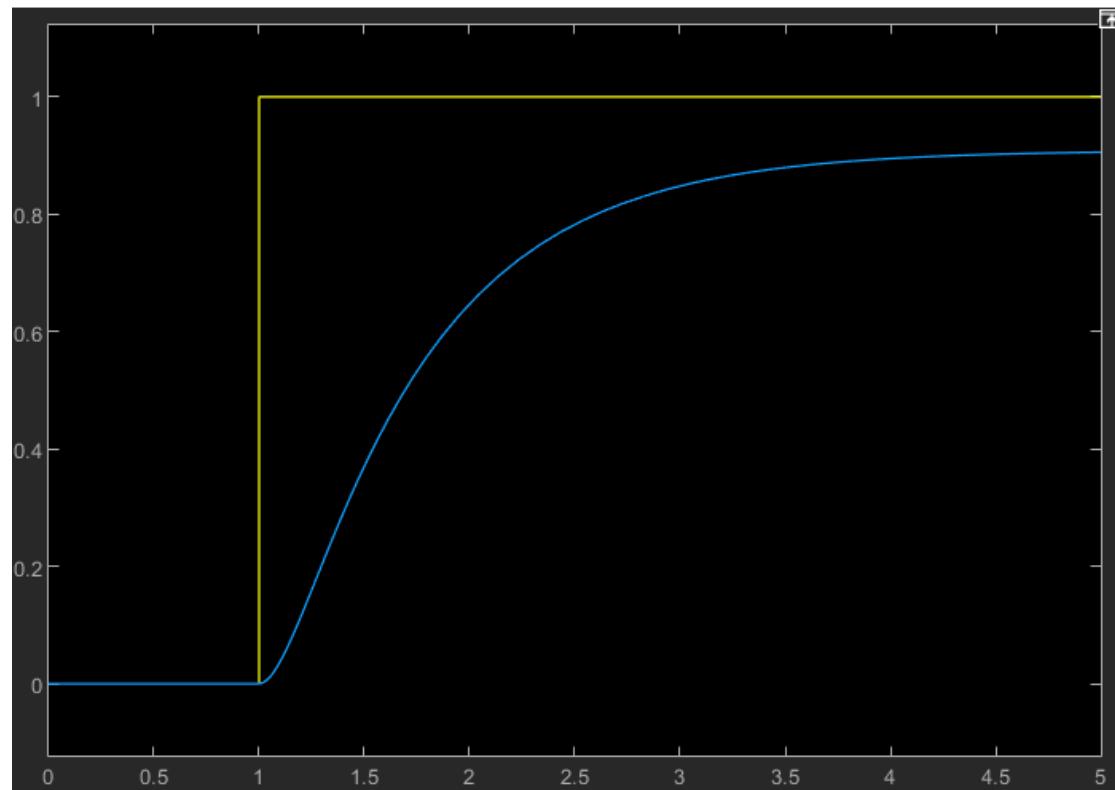
实现控制系统的一个常用方法是在无控制系统中加入 PID 控制器，PID 控制器能够根据实际输出值和期望值之间的误差调整输入值，使最终的输出值靠近期望值。

依然以电机为例，一个电机在挂了 10 斤负载的情况下，如果给电机的输入电压刚好是 24V，则电机的实际输出只有 80rpm，输入和实际输出的误差为 20rpm，将这个差值 20rpm 输入 PID 控制器，PID 控制器就会将输入量提高到 26V，这时电机的实际输出就能够达到 100rpm 了。如果差值变大到 40rpm，PID 控制器就将给电机的电压输入量继续提高。总之 PID 控制器会根据误差的大小自动完成对电机电压输入值的调节，从而保持输出值和期望值一致。

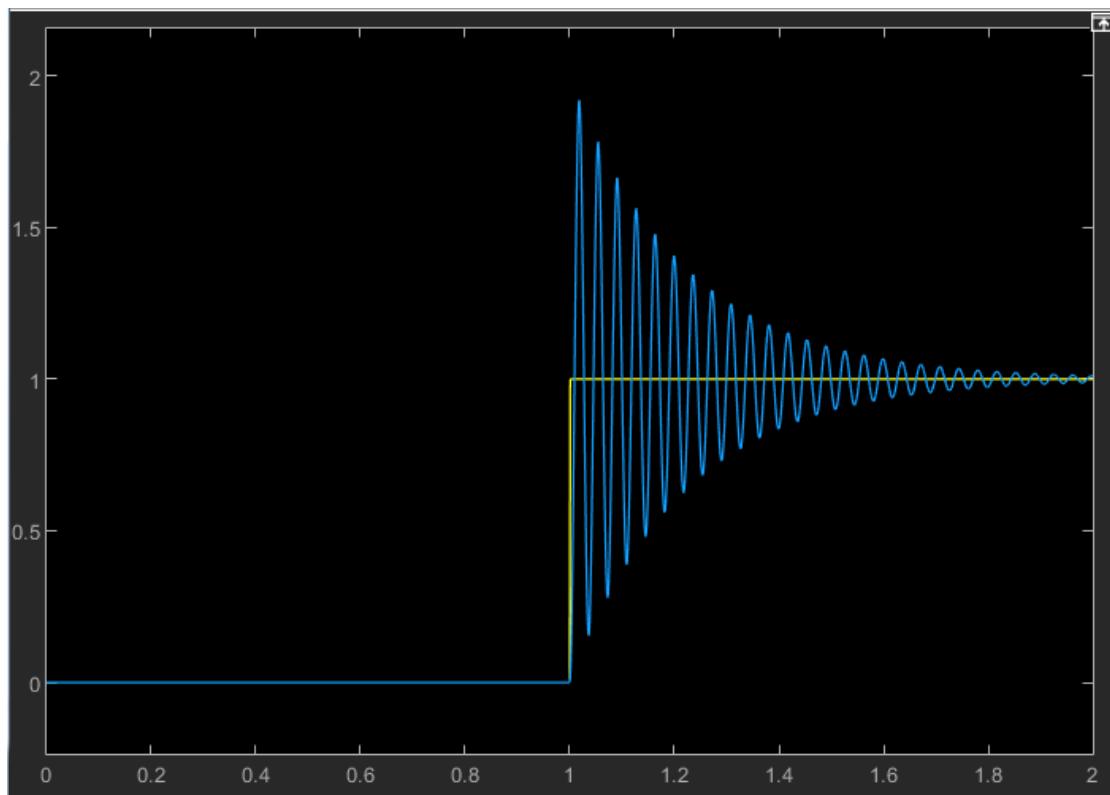
下面将对基础学习中的 PID 控制缺点，进行效果图呈现。

比例项 K_p 过小时, PID 控制器的反应速度较慢且存在静差。静差是指控制器的最终输出保持为一个和期望值存在一定误差的值, 引发静差的原因时由于比例控制的输出和误差成线性关系, 如果当误差值减小时, 比例控制器的输出值同样会减少, 导致比例控制器不可能达到和期望值完全相同, 即误差值为零的情况, 因为如果误差值为零则比例控制器的输出也会为零。

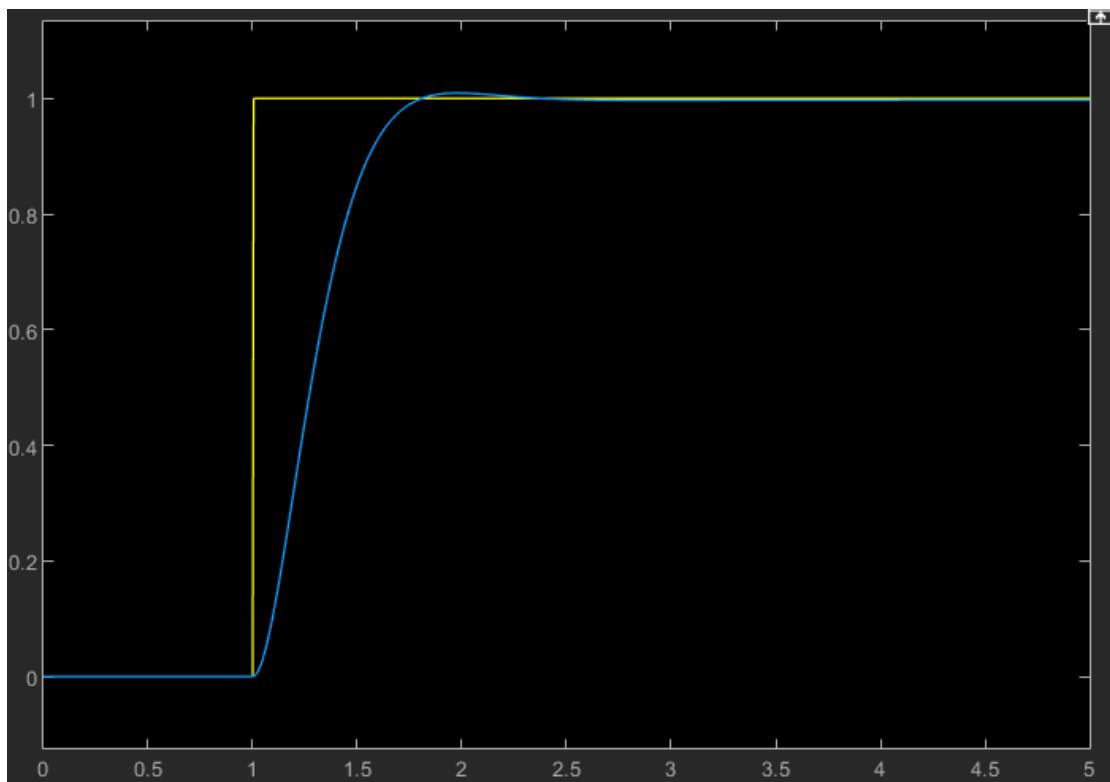
下图为 PID 控制器比例项过小的仿真, 蓝色曲线为系统的输出, 黄色曲线为设定的输入, 可以看到最终输出值和期望值之间保持一个恒定的误差值, 这个误差值就是静差。



通过增大比例系数 K_p 可以减小静差, 提高反应速度。但如果 K_p 过大则会导致输出严重震荡, 如下图所示。震荡会使系统的稳定性大大下降。



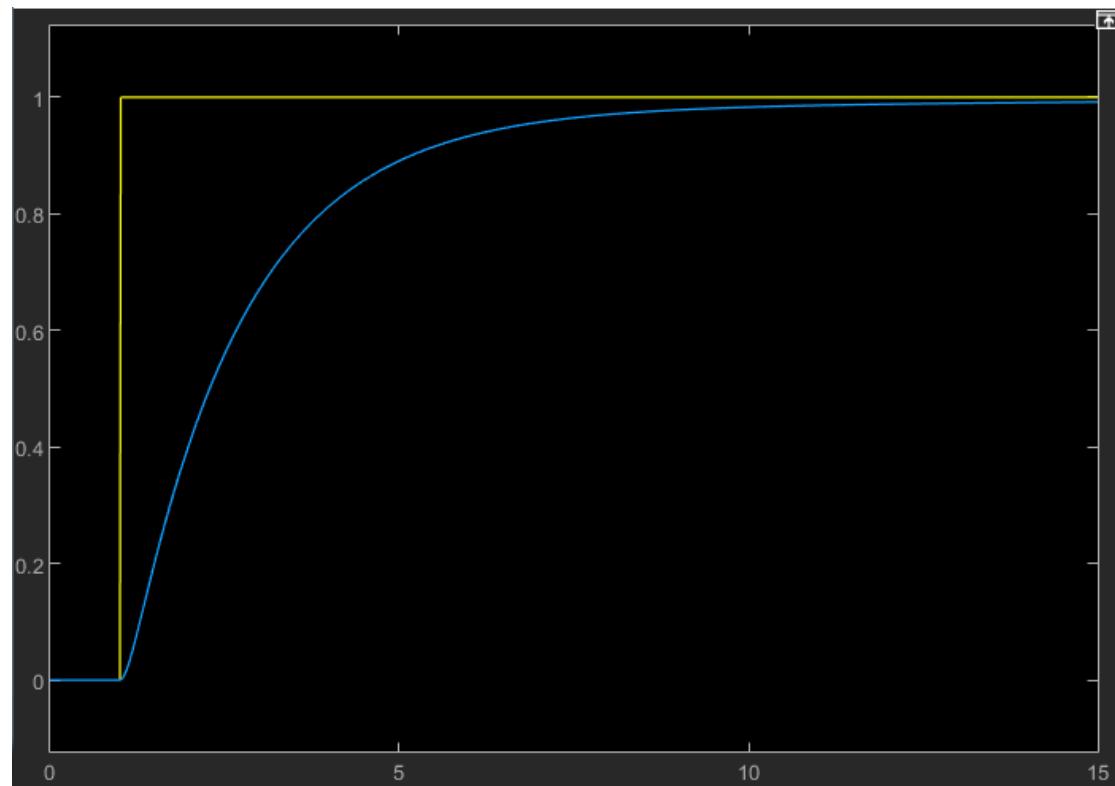
为了克服静差，可以引入积分项，由于积分的累积性，积分项的输出值由之前所有时刻的误差值决定。因此即使在误差值为零的情况下，积分项也会保持作用，使控制器的输出量不为零，引入积分项之后控制器的输出值才可以真正等于期望值。



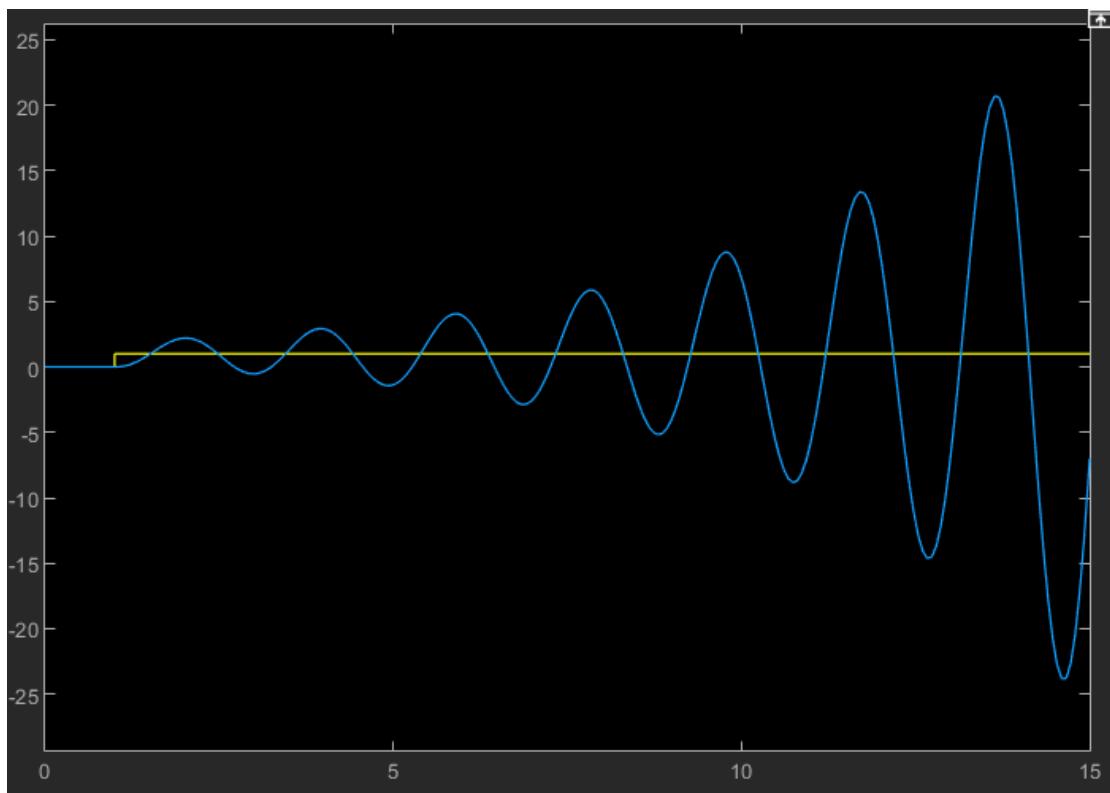
但是引入积分控制器后会延长系统的调节时间，比例-积分控制器的输出值需要相较比例控

制器更长的时间趋于稳定，导致系统的响应速度变慢。

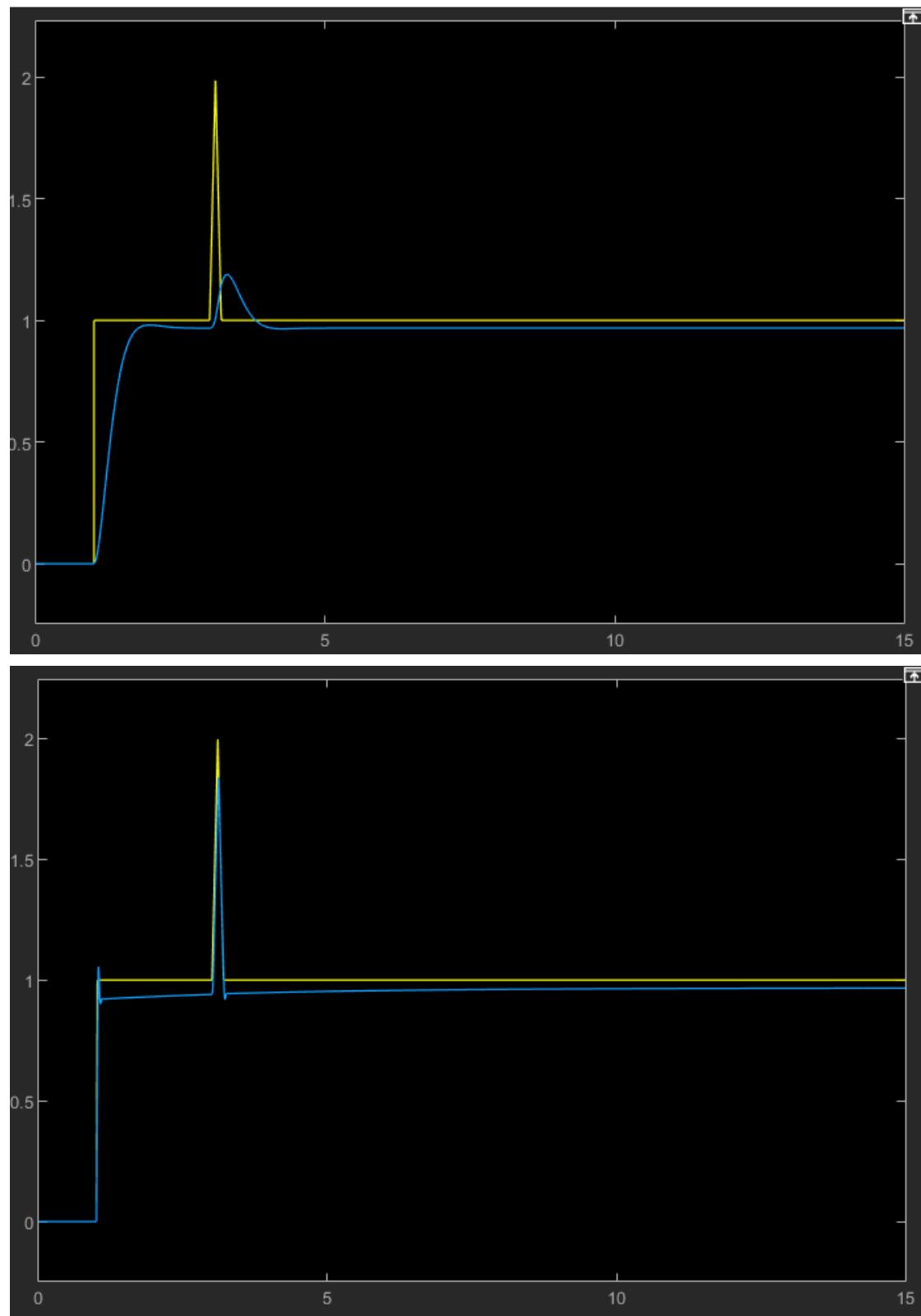
如果积分项过小，或者积分上限幅过小，则输出收敛的时间会非常长，这就导致系统的响应速度非常慢，如下图所示。



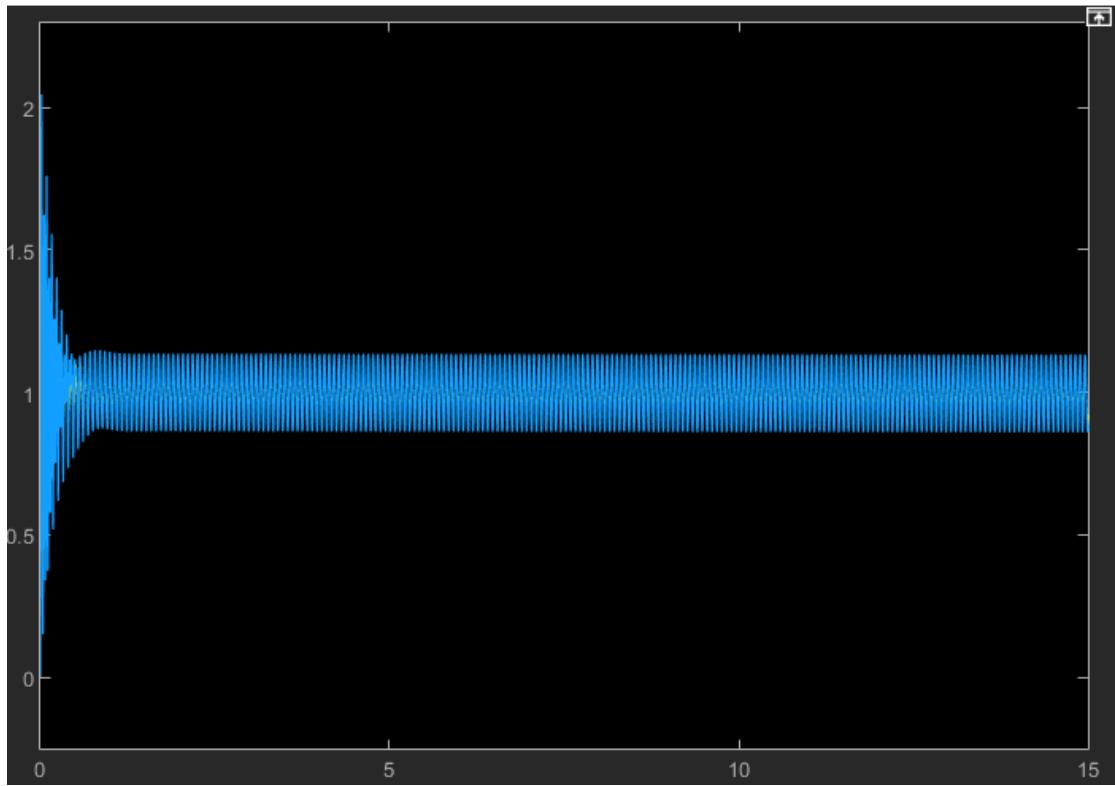
而如果积分项过大，或者积分上限过大，则会带来严重的超调。超调即系统输出的峰值远远高于期望值，超调会导致系统需要很长时间才能够趋于稳定，甚至可能无法趋于稳定，会严重影响系统的稳定性，降低系统响应速度，如下图所示。



为了提高系统的响应速度和减少超调量，我们可以引入微分项，微分项的大小和输出值的变化量成正相关，因此每当输入产生急剧变化时，就会产生一个很大的微分项来迅速跟上这一变化，这就可以使得超调量减小。另外由于微分项抵消变化的这一性质，输出波动就可以更快的衰减，减少系统输出稳定需要的时间，从而加快响应速度。



但是如果 D 项过大则输出的一点小变化都会产生很大的输出变化，导致系统发生震荡，如下图所示：



一般我们设计 PID 控制器时，需要根据 PID 的实际输出情况对 PID 三项的系数的进行调节，从而让系统的输出快速且稳定。

下表总结了 PID 控制器中各项的功能：

	功能
比例项 P	根据误差值线性调控输出，存在静差和超调
积分项 I	能够消除静差，但会导致系统响应速度变慢
微分项 D	减小超调，加快系统响应

16.6 课程总结

PID 控制是常见的控制方法，温度控制有利于抑制 IMU 的零漂的影响。本节课我们学习了 PID 控制的基本原理，并编写程序，通过 PID 控制的方法来保持 IMU 的工作温度恒定，从而减少 IMU 的零漂的影响。

17. 底盘控制任务

17.1 知识要点

- 麦克纳姆轮的结构
- 电机速度环控制
- 底盘运动学的正运动过程

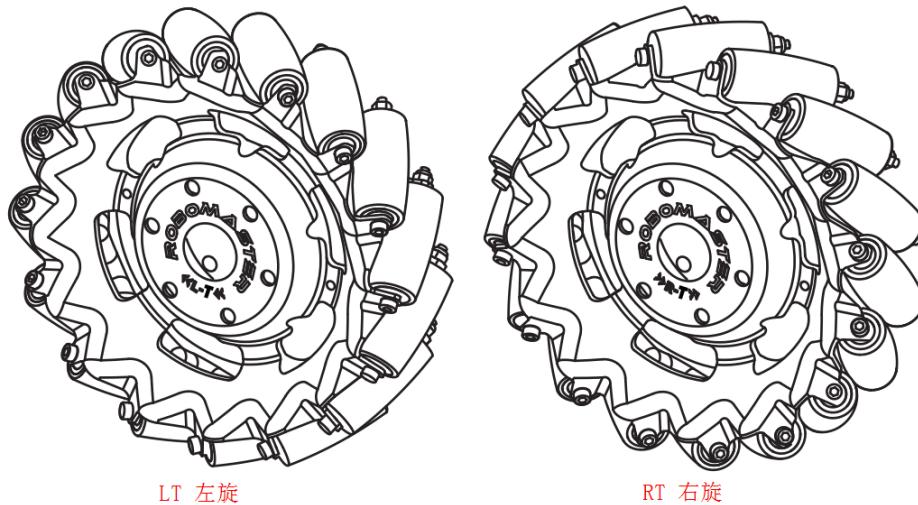
17.2 课程内容

在本课程中，将学习 RoboMaster 机器人的底盘控制。底盘是承载 RoboMaster 机器人运动的机械结构，底盘一般由轮组，悬挂，车架等组成。其中 RoboMaster 比赛底盘轮组使用的是麦克纳姆轮，每个麦克纳姆轮都由单独的电机驱动，以完成全向移动的功能。RoboMaster 机器人底盘往往需要配合云台进行配合控制，例如 RoboMaster 机器人底盘角度控制就需要跟随云台角度进行控制，同时底盘在运动过程也需要云台角度进行旋转操作以到达平稳运动的效果。

17.3 基础学习

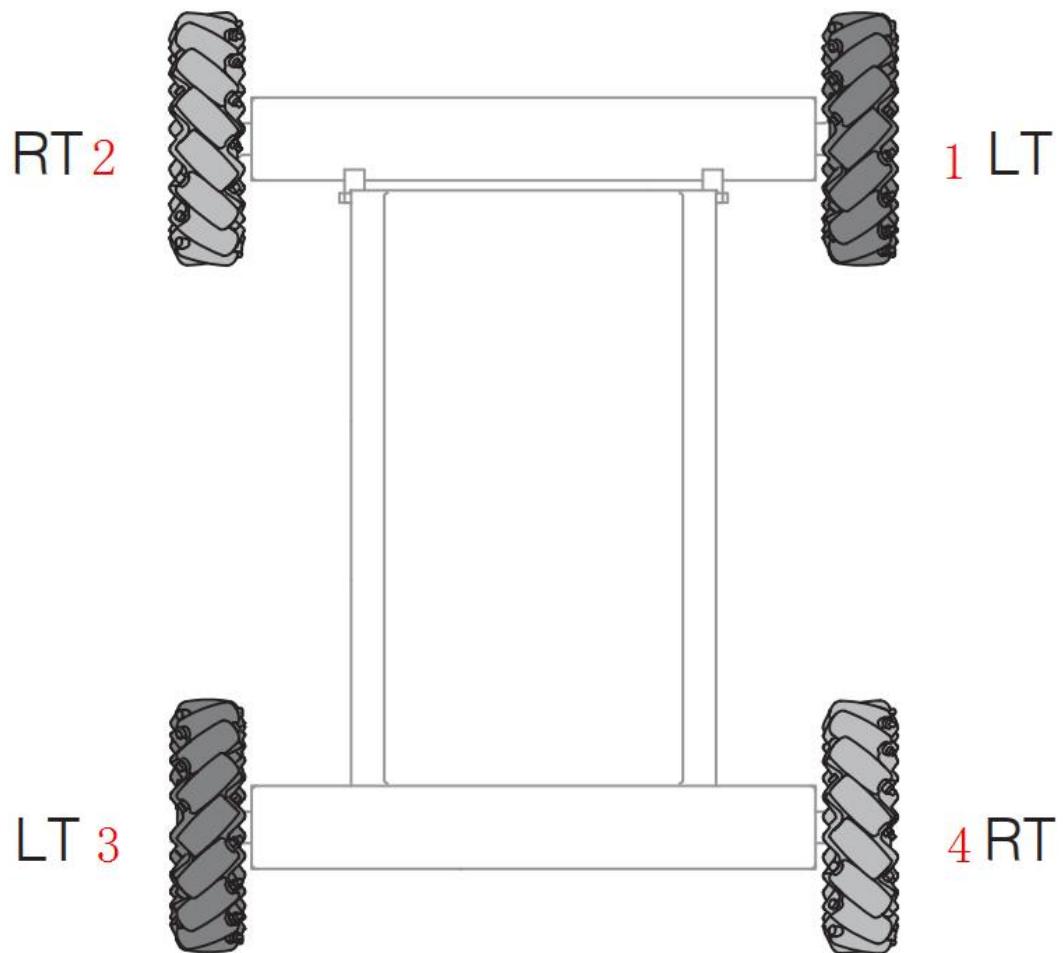
17.3.1 麦克纳姆轮的结构

RoboMaster 麦克纳姆轮是一款直径为 152.5mm 的 45° 全向轮，在轮毂周围分布着 16 个橡胶小滚子，与车轮的轴线成 45° 角，机械结构如图所示。麦克纳姆轮根据小胶轮的角度分为左旋麦克纳姆轮和右旋麦克纳姆轮，它们是手性对称，必须成套使用。



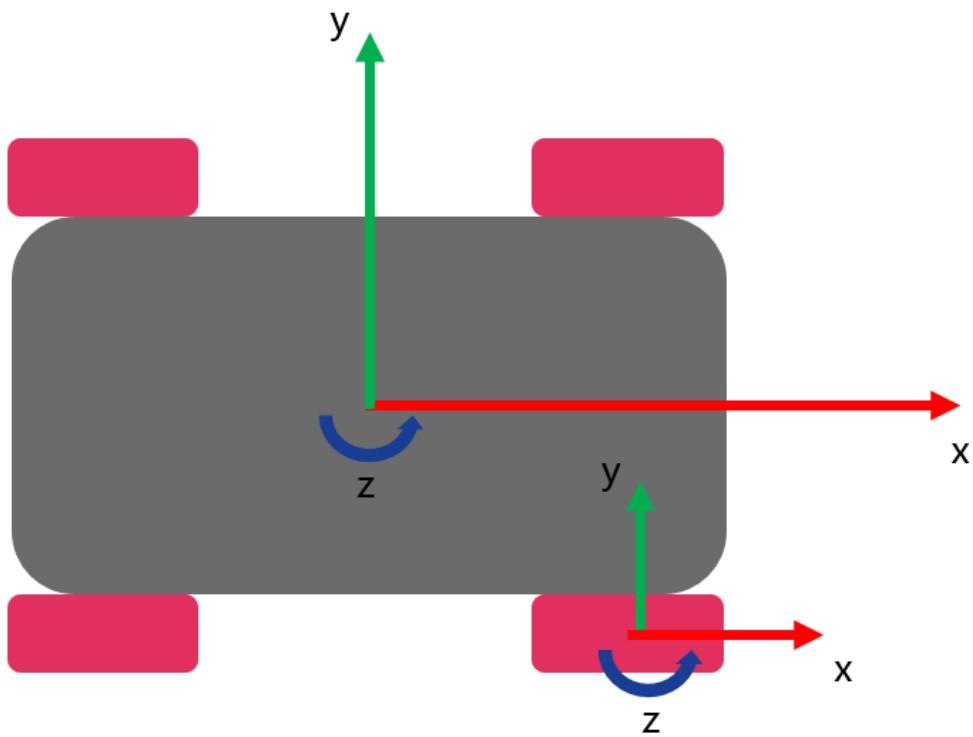
17.3.2 麦克纳姆轮的安装

底盘麦克纳姆轮安装俯视图如图所示，可以看出 1、2、3 和 4 号轮子在不同的方位，1 和 3 为左旋麦克纳姆轮，2 和 4 为右旋麦克纳姆轮。



17.3.3 底盘正运动学

底盘运动方向，需要建立一个坐标系进行讲解，如图所示。底盘前进方向为x轴，左移方向为y轴，z轴竖直向上，底盘逆时针旋转为旋转正方向。



一个刚体在平面内具有三个自由度运动，沿 x 轴方向的前后运动 V_x ，沿 y 轴方向的左右运动 V_y ，绕 z 轴的旋转运动 W_z 。根据底盘的三个自由度运动计算四个麦克纳姆轮的运动的过程，称为底盘的正运动过程。根据底盘速度计算麦克纳姆轮的速度如下表所示。

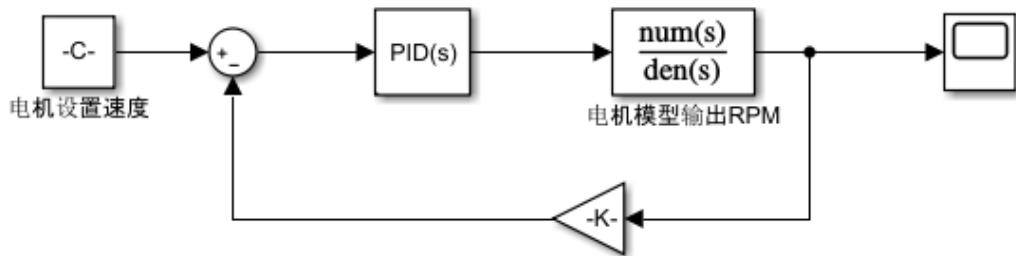
底盘速度	麦克纳姆轮速度
前进速度 V_x	$V1 = V_x$ $V2 = V_x$ $V3 = V_x$ $V4 = V_x$
左右速度 V_y	$V1 = V_y$ $V2 = -V_y$ $V3 = V_y$ $V4 = -V_y$
旋转速度 W_z	$V1 = W_z$ $V2 = -W_z$ $V3 = -W_z$ $V4 = W_z$

17.3.4 电机的速度环控制

从上节课的学习中，了解 PID 控制方法。本课程中，将应用 PID 控制方法，进行 3508 电机的速度控制。

底盘装配四个电机，分别控制 4 个麦克纳姆轮。底盘完成全向移动，需要对 4 个车轮进行精准的速度控制，才能保证底盘移动的方向不会因为车轮转速不同，而导致方向出现偏差。

控制环路如图所示：



17.4 程序学习

17.4.1 can 发送电机控制函数回顾

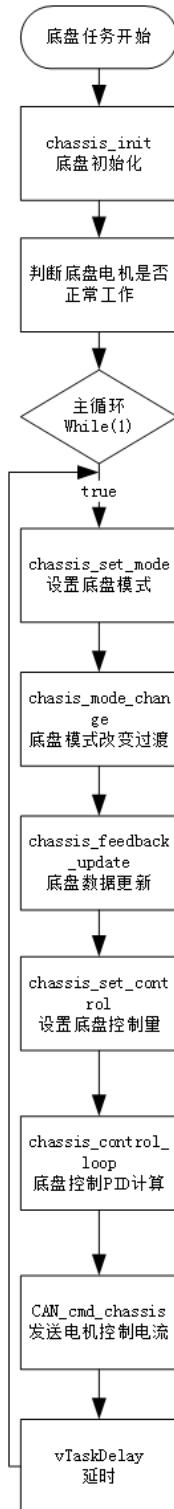
在 CAN 通信章节中，学习到如何使用 CAN 通信进行电机控制。底盘电机分配的 ID 分别是 0x201, 0x202, 0x203 和 0x204，它们都是由 ID 为 0x200 的 CAN 包进行控制，而在 CAN_receive.c 中封装底盘控制函数 CAN_cmd_chassis。

函数名	CAN_cmd_chassis
函数功能	通过 can 总线对电机电流进行控制
函数返回	None
参数 1: motor1	ID 为 0x201 的电机电流控制值，范围[-16384,16384]
参数 2: motor2	ID 为 0x202 的电机电流控制值，范围[-16384,16384]
参数 3: motor3	ID 为 0x203 的电机电流控制值，范围[-16384,16384]

参数 4: motor4	ID 为 0x204 的电机电流控制值, 范围[-16384,16384]
--------------	---------------------------------------

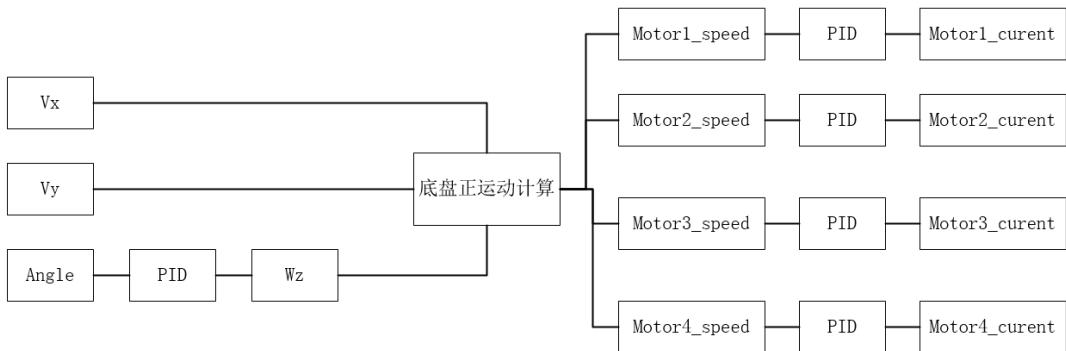
17.4.2 程序流程讲解

使用 freeRTOS 创建底盘任务，底盘任务主循环流程如下。



流程	功能
chassis_init	底盘初始化，主要初始化电机速度 PID，遥控器指针和姿态角度指针。
chassis_set_mode	根据遥控器的开关值，以及云台状态决定底盘控制行为，根据底盘控制行为决定底盘控制模式。
chassis_mode_change_control_transit	底盘控制模式切换过程中保存数据，以保证控制切换的控制目标平稳过渡。
chassis_feedback_update	底盘电机反馈速度，姿态角度，并更新底盘运动速度。
chassis_set_control	根据遥控器和键盘按键的输入，计算不同电机控制速度
chassis_control_loop	底盘根据不同控制模式，计算不同 PID。
CAN_cmd_chassis	发送电机电流控制值

RoboMaster 机器人常见的控制路线如图所示，由遥控器以及键盘的通道值计算出移动速度 V_x, V_y ，由云台角度 PID 计算旋转速度 W_z ，经过底盘正运动学计算出四个电机控制速度，通过各自的速度 PID 计算控制电流值。



17.4.3 关键函数讲解

以下讲解几个主要函数：

1. chassis_set_mode 函数和 chassis_behaviour_mode_set 函数

chassis_set_mode 函数调用 chassis_behaviour_mode_set 函数完成设置控制模式。而在 chassis_behaviour_mode_set 函数通过遥控器的开关决定底盘的行为，然后通过行

为决定底盘控制方式，行为模式和控制模式可以参考下表。

```
static void chassis_set_mode(chassis_move_t *chassis_move_mode)

{

    //in file "chassis_behaviour.c"

    chassis_behaviour_mode_set(chassis_move_mode);

}

void chassis_behaviour_mode_set(chassis_move_t *chassis_move_mode)

{

    //remote control  set chassis behaviour mode

    //遥控器设置模式

    if (switch_is_mid(chassis_move_mode->chassis_RC->rc.s[CHASSIS_MODE_CHANNEL]))


    {

        chassis_behaviour_mode = CHASSIS_NO_FOLLOW_YAW;

    }

    else if (switch_is_down(chassis_move_mode->chassis_RC->rc.s[CHASSIS_MODE_CHANNEL]))


    {

        chassis_behaviour_mode = CHASSIS_NO_MOVE;

    }

    else if (switch_is_up(chassis_move_mode->chassis_RC->rc.s[CHASSIS_MODE_CHANNEL]))


    {

        chassis_behaviour_mode = CHASSIS_INFANTRY_FOLLOW_GIMBAL_YAW;

    }

    //accord to behaviour mode, choose chassis control mode

    //根据行为模式选择一个底盘控制模式
```

```

if (chassis_behaviour_mode == CHASSIS_ZERO_FORCE)

{
    chassis_move_mode->chassis_mode = CHASSIS_VECTOR_RAW;

}

.....



else if (chassis_behaviour_mode == CHASSIS_OPEN)

{
    chassis_move_mode->chassis_mode = CHASSIS_VECTOR_RAW;

}
}

```

行为模式	底盘表现以及应用
CHASSIS_ZERO_FORCE	底盘电机电流控制值为 0，底盘表现为无力状态，应用于遥控器掉线或者需要底盘上电时方便推动的场合
CHASSIS_NO_MOVE	底盘电机速度控制值为 0，由于有电机的速度环控制，底盘表现为不移动，但推动底盘存在抵抗力，应用于遥控器开关处下位，需要底盘停止运动的场合。
CHASSIS_INFANTRY_FOLLOW_GIMBAL_YAW	底盘移动速度由遥控器和键盘按键一起决定，同时会控制底盘跟随云台，从而计算旋转速度，底盘存在电机速度环和云台角度环控制，应用于遥控器开关处于上位，正常跟随云台角度的场合，是常见的 RoboMaster 机器人控制逻辑。
CHASSIS_ENGINEER_FOLLOW_CHASSIS_YAW	底盘移动速度由遥控器和键盘按键一起决定，同时会控制底盘跟随底盘角度，从而计算旋转速度，底盘存在电机速度环和底盘角度环控制，常见于工程机器人控制场合。
CHASSIS_NO_FOLLOW_YAW	底盘移动速度和旋转速度均由遥控器决定，不存在角度闭环控制，故而底盘只有速度环控制，应用于只需要底盘控制的场合
CHASSIS_OPEN	底盘开环控制，不存在程序的闭环控制，遥控器的通道值直接转化成电机电流值发送到 CAN 总线上。

底盘控制模式分为四种，如下表所示。

控制模式	功能以及应用
CHASSIS_VECTOR_FOLLOW_GIMBAL_YAW	底盘移动速度由遥控器和键盘决定，旋转速度由云台角度差计算得出，是 CHASSIS_INFANTRY_FOLLOW_GIMBAL_YAW 选择的控制模式
CHASSIS_VECTOR_FOLLOW_CHASSIS_YAW	底盘移动速度由遥控器和键盘决定，旋转速度由底盘角度差计算得出，是 CHASSIS_ENGINEER_FOLLOW_CHASSIS_YAW 选择的控制模式
CHASSIS_VECTOR_NO_FOLLOW_YAW	底盘移动速度和旋转速度由遥控器决定，无角度环控制，是 CHASSIS_NO_FOLLOW_YAW 和 CHASSIS_NO_MOVE 选择的控制模式
CHASSIS_VECTOR_RA_W	底盘电机电流控制值是直接由遥控器通道值计算出来的，将直接发送到 CAN 总线上，是 CHASSIS_OPEN 和 CHASSIS_ZERO_FORCE 选择的控制模式。

2. chassis_set_control 函数和 chassis_behaviour_conotrol_set 函数

chassis_set_control 函数调用 chassis_behaviour_conotrol_set 函数得到底盘三个自由度运动的控制值，当底盘控制模式为 CHASSIS_VECTOR_FOLLOW_GIMBAL_YAW，经过一次旋转操作，将底盘控制速度旋转成云台角度方向，保证底盘运动的平稳，同时这种旋转控制也可以用于小陀螺旋转中的底盘运动控制。

```
static void chassis_set_control(chassis_move_t *chassis_move_control)
{
    fp32 vx_set = 0.0f, vy_set = 0.0f, angle_set = 0.0f;

    chassis_behaviour_control_set(&vx_set, &vy_set, &angle_set, chassis_move_control);

    if (chassis_move_control->chassis_mode == CHASSIS_VECTOR_FOLLOW_GIMBAL_YAW)
    {
        fp32 sin_yaw = 0.0f, cos_yaw = 0.0f;
```

```

//rotate chassis direction, make sure vertical direction follow gimbal

//旋转控制底盘速度方向，保证前进方向是云台方向，有利于运动平稳

sin_yaw = arm_sin_f32(-chassis_move_control->chassis_yaw_motor->relative_angle);

cos_yaw = arm_cos_f32(-chassis_move_control->chassis_yaw_motor->relative_angle);

chassis_move_control->vx_set = cos_yaw * vx_set + sin_yaw * vy_set;

chassis_move_control->vy_set = -sin_yaw * vx_set + cos_yaw * vy_set;

//set control relative angle set-point

//设置控制相对云台角度

chassis_move_control->chassis_relative_angle_set = rad_format(angle_set);

//calculate rotation speed

//计算旋转 PID 角速度

chassis_move_control->wz_set = -PID_calc(&chassis_move_control->chassis_angle_pid, chassis_move_control->chassis_yaw_motor->relative_angle,
chassis_move_control->chassis_relative_angle_set);

//speed limit

//速度限幅

chassis_move_control->vx_set = fp32_constrain(chassis_move_control->vx_set, chassis_move_control->vx_min_speed, chassis_move_control->vx_max_speed);

chassis_move_control->vy_set = fp32_constrain(chassis_move_control->vy_set, chassis_move_control->vy_min_speed, chassis_move_control->vy_max_speed);

}

.....
}

```

3. chassis_vector_to_mecanum_wheel_speed 函数

chassis_vector_to_mecanum_wheel_speed 函数功能是将底盘三自由度运动速度转化成麦轮速度，根据基础学习中，可以得出以下的公式

$$v_1 = v_x + v_y + w_y$$

$$v_2 = v_x - v_y - w_y$$

$$v_3 = v_x + v_y - w_y$$

$$v_4 = v_x - v_y + w_y$$

而由于当电机 1 和电机 4 处于前进速度的时候，电机 1 和电机 4 是顺时针旋转为负值，电机 2 和电机 3 是逆时针旋转为正值，故而修正后的公式为：

$$v_1 = -v_x - v_y - w_y$$

$$v_2 = v_x - v_y - w_y$$

$$v_3 = v_x + v_y - w_y$$

$$v_4 = -v_x + v_y - w_y$$

由于云台安装位置不在底盘中心，往往是在底盘前部，故而在旋转时，电机 1 和电机 2 需要较慢的速度，电机 3 和电机 4 需要较快的速度，需要一个修正因子 CHASSIS_WZ_SET_SCALE(小于 1)，假定为 a 参数，故而修改后的公式为：

$$v_1 = -v_x - v_y + (a - 1) * w_y$$

$$v_2 = v_x - v_y + (a - 1) * w_y$$

$$v_3 = v_x + v_y + (-a - 1) * w_y$$

$$v_4 = -v_x + v_y + (-a - 1) * w_y$$

故而函数实现如下：

```
static void chassis_vector_to_mecanum_wheel_speed(const fp32 vx_set, const fp32 vy_set, const fp32 wz_set, fp32 wheel_speed[4])

{
    //because the gimbal is in front of chassis, when chassis rotates, wheel 0 and wheel 1 should be slower and wheel 2 and wheel 3 should be faster

    //旋转的时候， 由于云台靠前， 所以是前面两轮 0 , 1 旋转的速度变慢， 后面两轮 2,3 旋转的速度变快

    wheel_speed[0] = -vx_set - vy_set + (CHASSIS_WZ_SET_SCALE - 1.0f) * MOTOR_DISTANCE_TO_CENTER * wz_set;

    wheel_speed[1] = vx_set - vy_set + (CHASSIS_WZ_SET_SCALE - 1.0f) * MOTOR_DISTANCE_TO_CENTER * wz_set;

    wheel_speed[2] = vx_set + vy_set + (-CHASSIS_WZ_SET_SCALE - 1.0f) * MOTOR_DISTANCE_TO_CENTER * wz_set;

    wheel_speed[3] = -vx_set + vy_set + (-CHASSIS_WZ_SET_SCALE - 1.0f) * MOTOR_DISTANCE_TO_CENTER * wz_set;
}
```

4. chassis_control_loop 函数

chassis_control_loop 函数先进入底盘正运动计算出麦轮速度，之后判断如果是 CHASSIS_VECTOR_RAW，就直接赋值完成计算，其他情况下判断轮子最大速度，进行比例缩小，之后进行速度环闭环控制，其中速度环控制的代码如下：

```

//calculate pid

//计算 pid

for (i = 0; i < 4; i++)

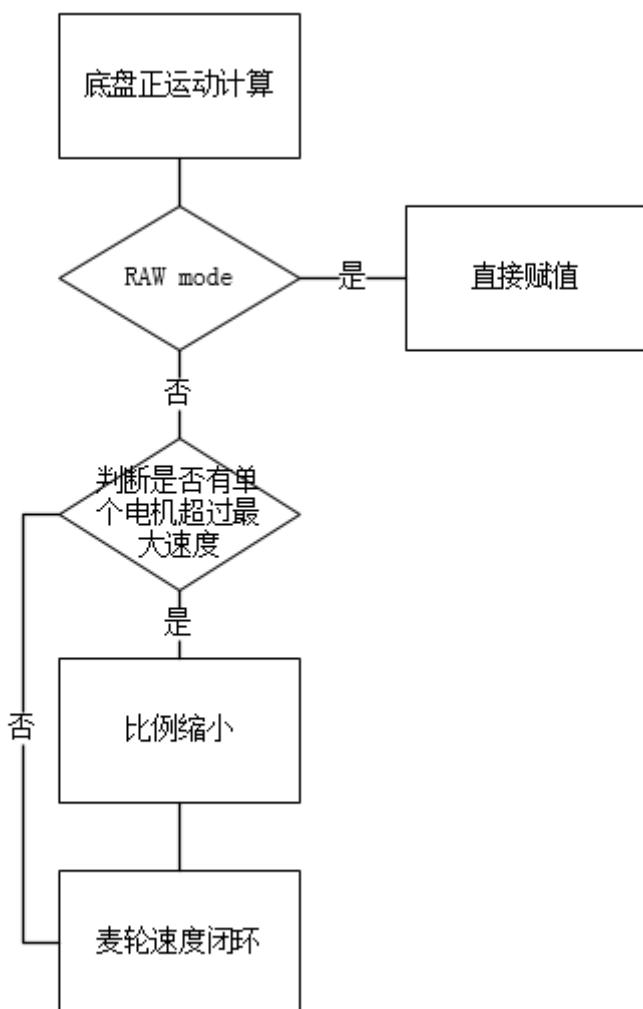
{

    PID_calc(&chassis_move_control_loop->motor_speed_pid[i],  

chassis_move_control_loop->motor_chassis[i].speed, chassis_move_control_loop->motor_chassis[i].speed_set);

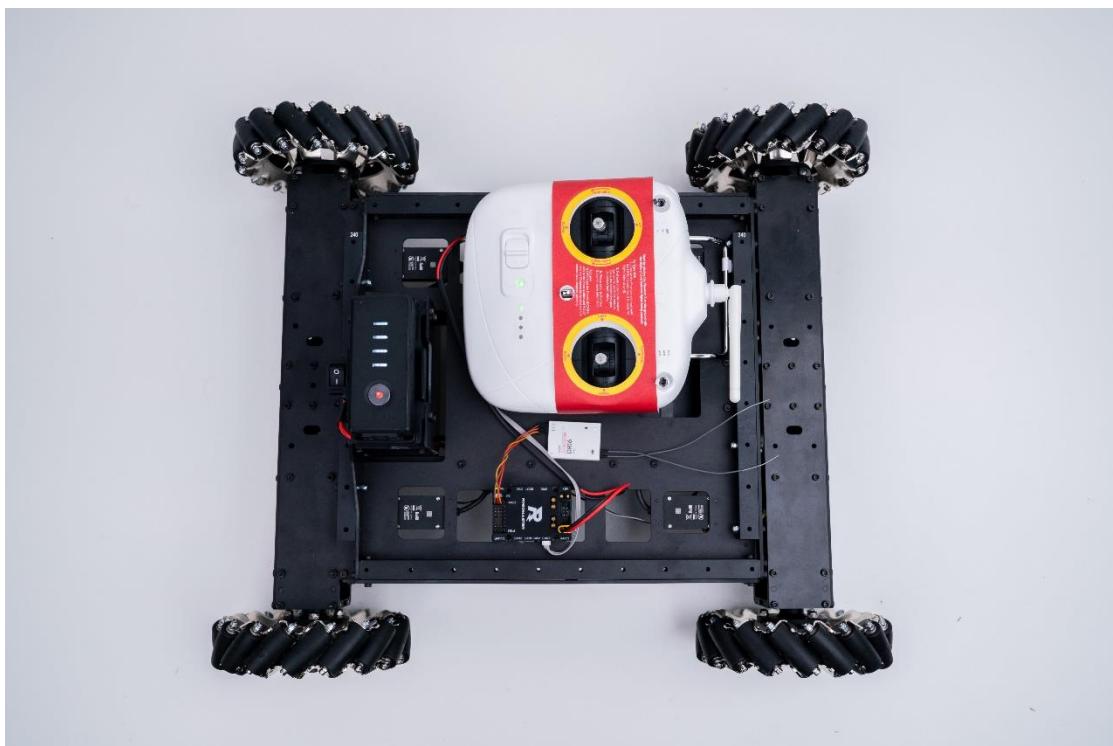
}

```



17.4.4 效果展示

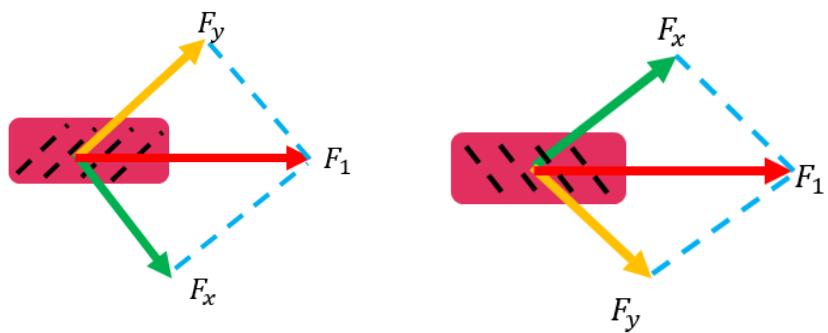
使用遥控器可以进行底盘运动控制，底盘如图所示：



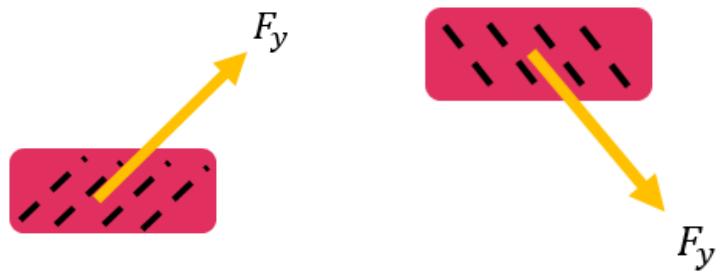
17.5 进阶学习

17.5.1 底盘运动学的正运动过程

麦克纳姆轮与普通橡胶轮的不同在于外圈的小胶轮。小胶轮在接触地面时，由于与轴线成 45° 角，会将地面的摩擦力分解成沿小胶轮轴线方向的 F_y 和垂直于轴线方向的 F_x ，其中垂直于轴线的分力 F_x 会使得小胶轮旋转，而使得其分力对整体运动不造成影响， F_y 会对底盘整体运动造成影响。受力分析如图：



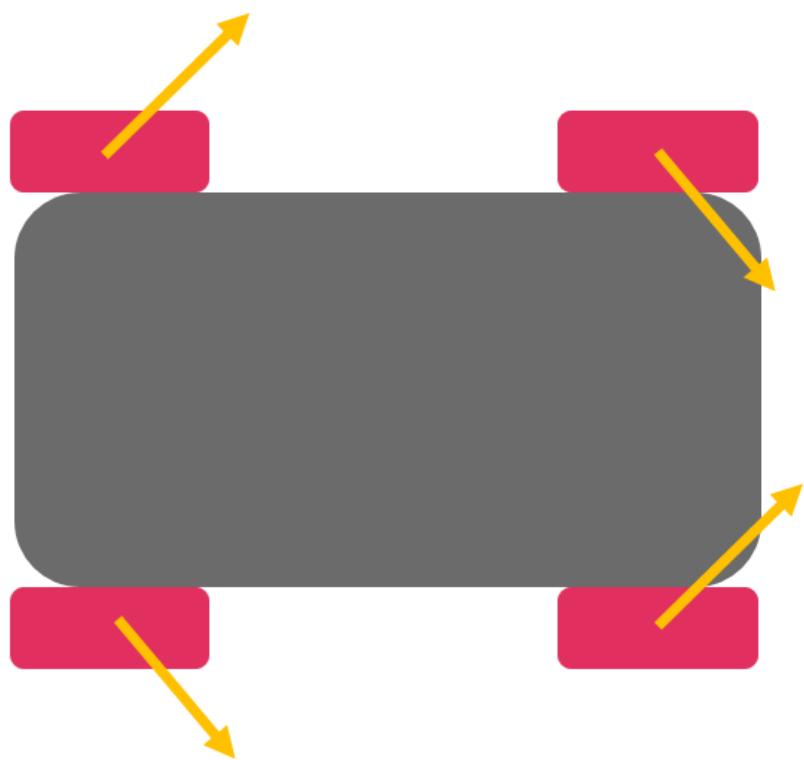
麦克纳姆轮受力分析



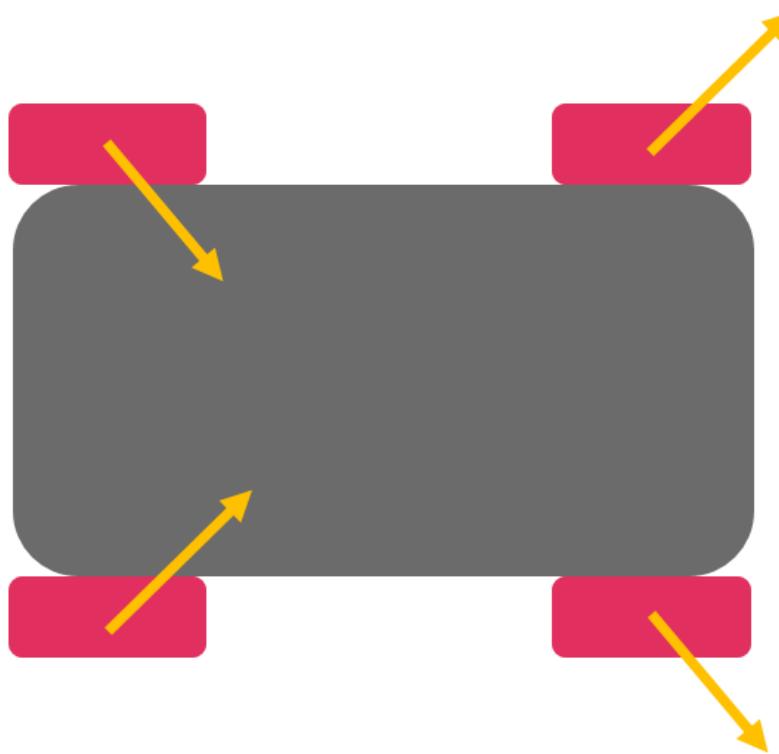
麦克纳姆轮 F_y 分力

由于分力方向不同, F_y 对底盘运动造成的影响不同, 通过控制四个电机的旋转进而控制底盘运动方向。

底盘上安装 4 个麦克纳姆轮, 需要两套左右旋的麦克纳姆轮, 完成底盘前进和平移方向具有两种安装方式, 分别为 O 型和 X 型。



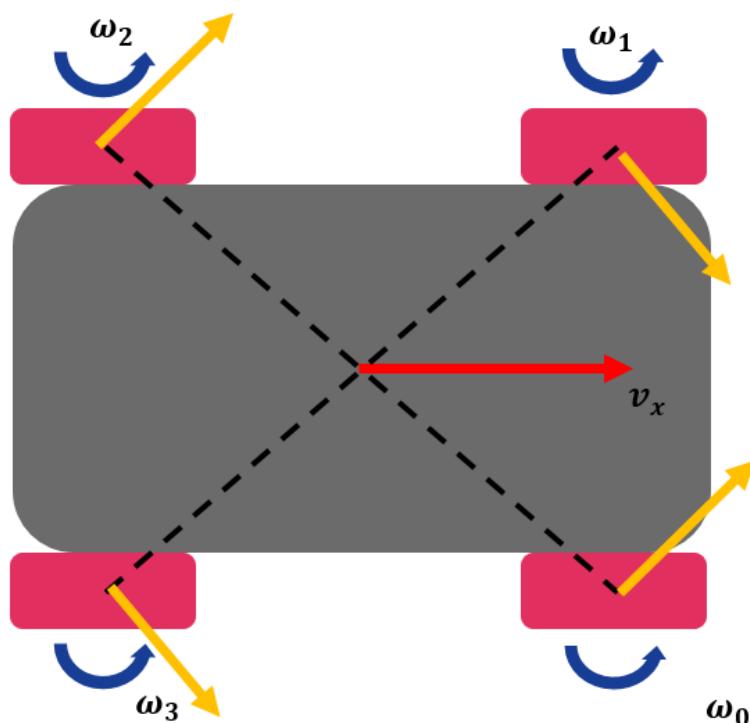
O型



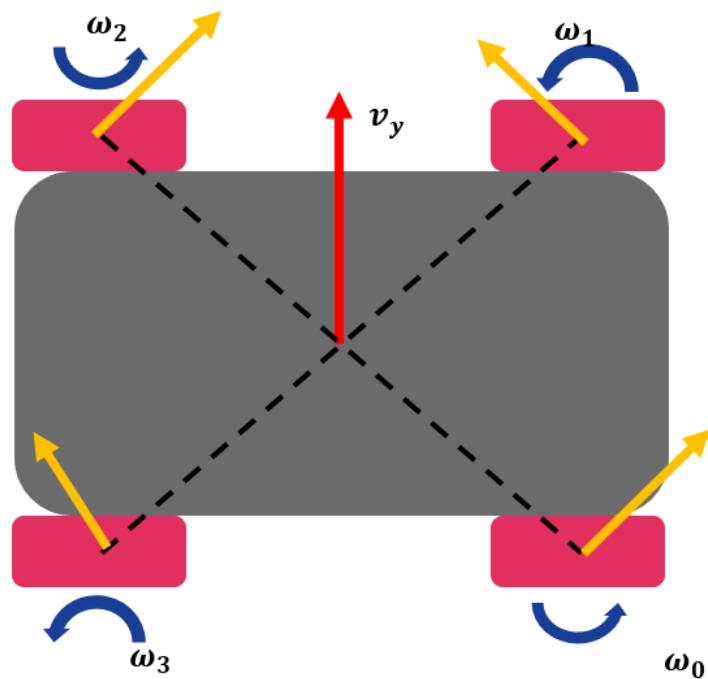
X型

其中X型旋转困难，所以底盘的麦克纳姆轮安装形式为O型。

底盘前进速度为 $x \text{ m/s}$ 时，四个底盘电机速度均为 $x \text{ m/s}$ 。如图所示，前两轮左右方向的受力相互抵消，只留下前后方向的受力；后两轮同样左右方向的受力相互抵消，只留下前后方向的受力。



底盘左移速度为 y m/s 时，四个底盘电机速度分别为 y m/s, $-y$ m/s, y m/s, $-y$ m/s。如图所示，前两轮前后方向的受力相互抵消，只留下前左右方向的受力；后两轮同样前后方向的受力相互抵消，只留下左右方向的受力。



17.6 课程总结

底盘是机器人运动的载体，其性能决定机器人的运动性能。使用麦克纳姆轮能够完成底盘全向移动，原因在于麦克纳姆轮的特殊结构和各自独立的电机速度环控制。通过底盘正运动学解算，可以将底盘三自由度的运动向量解算成四个底盘电机的速度，再通过电机反馈的速度进行速度环控制。

18. 姿态解算任务

18.1 知识要点

- 姿态角简介
- 四元数与姿态角的转化
- mahony 算法移植
- SPI 的 DMA 传输

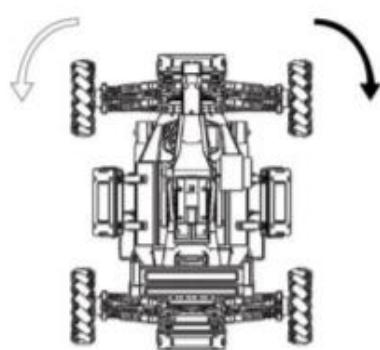
18.2 课程内容

本节课程我们将学习到姿态解算任务，姿态解算是将陀螺仪的角速度数据，加速度计的加速度计数据，磁力计的磁场数据进行融合，以解算出当前载体的姿态角。姿态解算算法的好坏将影响到姿态角度的精度。我们以 mahony 算法为例，移植相关算法，创建姿态解算任务，同时也学习如何使用 SPI 的 DMA 方式，节约 CPU 处理时间。

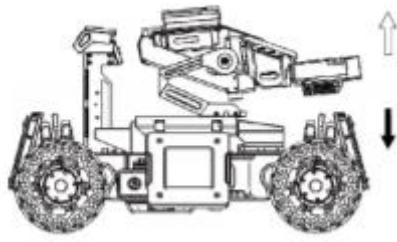
18.3 基础学习

18.3.1 姿态角简介

姿态是反应物体相对于参考坐标系的指向。常使用欧拉角代表物体的姿态，描述物体姿态需要三个角度，分别是 yaw(偏航角), pitch(俯仰角), roll(横滚角)。Yaw(偏航角)是绕 z 轴的角度，pitch 是绕 y 轴的角度，roll 是绕 x 轴的角度。RoboMaster 机器人通常具有两轴云台，分别是 yaw 轴和 pitch 轴，如图所示：



RoboMaster 机器人 Yaw 轴



RoboMaster 机器人 Pitch 轴

18.3.2 四元数与姿态角的转化

除了使用欧拉角代表姿态，还可以使用单位四元数代表姿态，四元数使用 4 个参数代表。如下式所示：

$$Q = q_0 + q_1 i + q_2 j + q_3 k \quad (18-1)$$

由于是单位四元数，需要满足平方和等于 1，如下式所示：

$$q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1 \quad (18-2)$$

在欧拉角旋转顺序为 yaw-pitch-roll，四元数转化成欧拉角的公式为：

$$\text{yaw} = \arctan\left(\frac{q_0 * q_3 + q_1 * q_2}{q_0 * q_0 + q_1 * q_1 - 0.5}\right) \quad (18-3)$$

$$\text{pitch} = \arcsin(2 * q_0 * q_2 - 2 * q_1 * q_3) \quad (18-4)$$

$$\text{roll} = \arctan\left(\frac{q_0 * q_1 + q_2 * q_3}{q_0 * q_0 + q_3 * q_3 - 0.5}\right) \quad (18-5)$$

四元数使用陀螺仪的数据进行积分，积分迭代公式为：

$$q_0^k = q_0^{k-1} - (q_1^{k-1} * g_x - q_2^{k-1} * g_y - q_3^{k-1} * g_z) * 0.5 * t \quad (18-6)$$

$$q_1^k = q_1^{k-1} + (q_0^{k-1} * g_x + q_2^{k-1} * g_z - q_3^{k-1} * g_y) * 0.5 * t \quad (18-7)$$

$$q_2^k = q_2^{k-1} + (q_0^{k-1} * g_y - q_1^{k-1} * g_z + q_3^{k-1} * g_x) * 0.5 * t \quad (18-8)$$

$$q_3^k = q_3^{k-1} + (q_0^{k-1} * g_z + q_1^{k-1} * g_y - q_2^{k-1} * g_x) * 0.5 * t \quad (18-9)$$

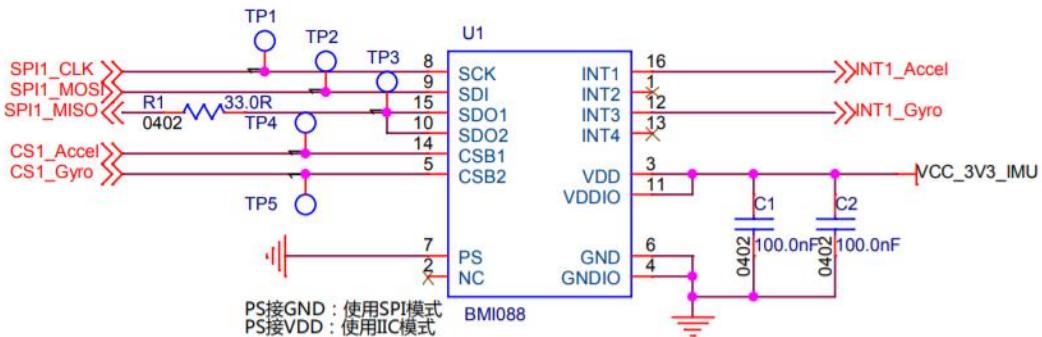
其中：

- g_x, g_y, g_z 为陀螺仪的 x, y, z 轴数据，
- t 为定时时间，程序设定更新频率为 1000Hz，故而 $t=0.001$ 。

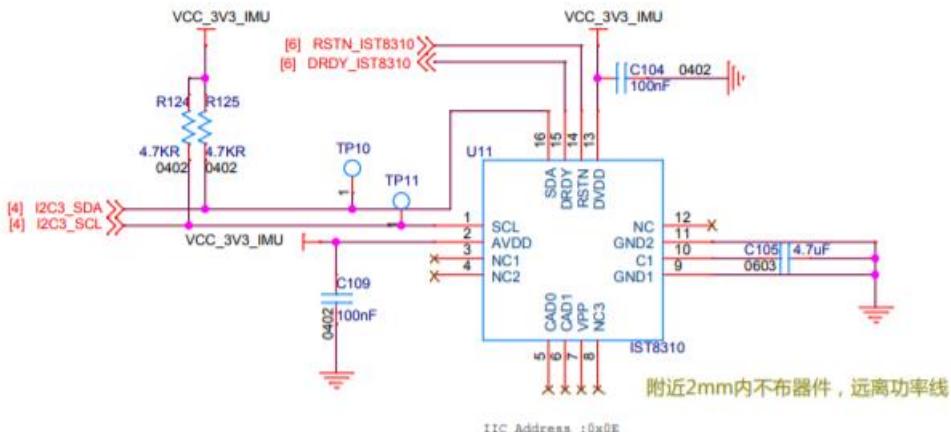
18.4 程序学习

18.4.1 cubeMX 中配置 GPIO,SPI 以及 I2C

在第 11 章 I2C 课程和第 13 章 SPI 课程中，学习了如何配置 I2C 和 SPI。对于本课程，我们需要融合磁力计数据，陀螺仪数据，加速度计数据，故而需要配置陀螺仪额外配置陀螺仪和加速度计的 DRDY 引脚，并配置成外部中断的模式，下图为陀螺仪相关的硬件原理图。



BMI088 硬件原理图



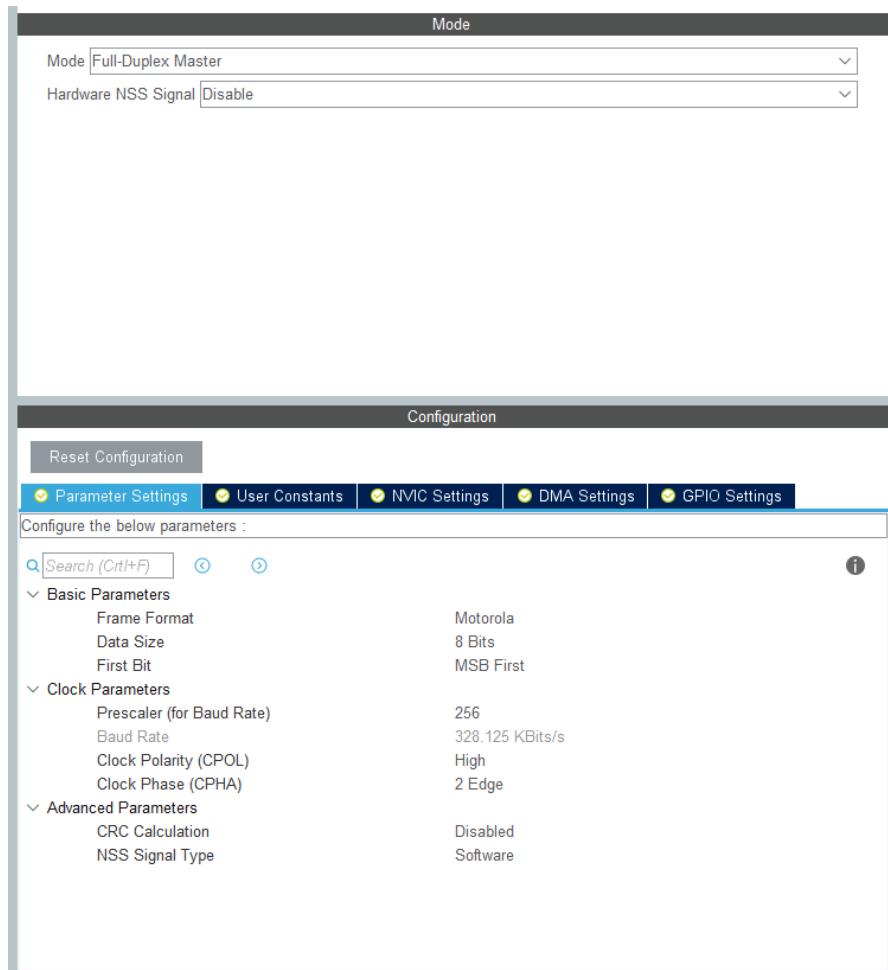
IST8310 硬件原理图

MCU 硬件引脚对应表

MCU 引脚	硬件原理图名称	功能
PA7	SPI1_MOSI	SPI1 的 MOSI
PB3	SPI1_CLK	SPI1 的 CLK

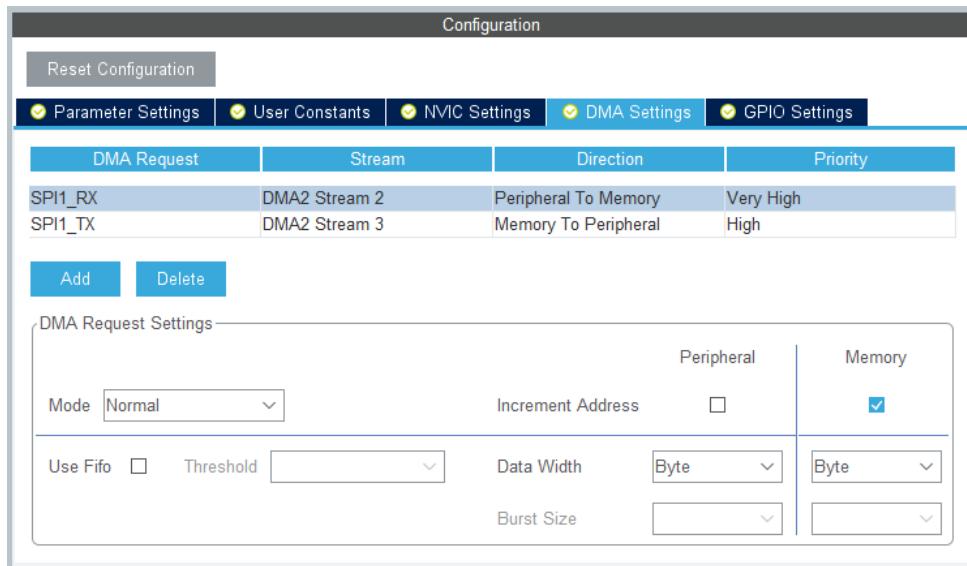
MCU 引脚	硬件原理图名称	功能
PB4	SPI_MISO	SPI1 的 MISO
PA4	CS1_Accel	加速度计的片选，低电平有效
PB0	CS1_Gyro	陀螺仪的片选，低电平有效
PC4	INT1_Accel	加速度计的 DRDY，外部下降沿中断
PC5	INT1_Gyro	陀螺仪的 DRDY，外部下降沿中断
PA8	I2C3_SDA	I2C 的 SCL
PC9	I2C3_SCL	I2C 的 SDA
PG3	DRDY_IST8310	磁力计的 DRDY，外部下降沿中断
PG6	RSTN_IST8310	磁力计的 RSTN，低电平有效

在 cubeMX 中，开启 SPI1，并且配置 SPI，如下图所示：



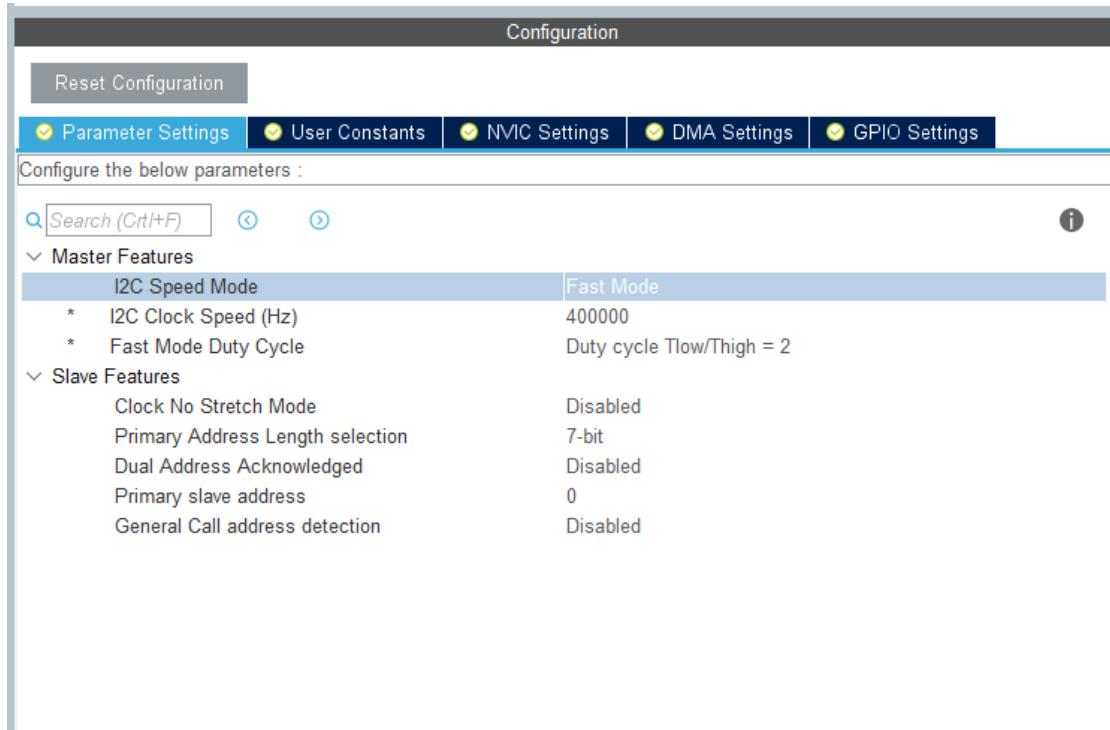
cubeMX 中 SPI 配置图

在上图 SPI 的配置中，点击 DMA Setting 选项，添加 SPI 的 DMA，并配置 DMA 如图所示。



cubeMX 中 SPI DMA 配置图

在 cubeMX 中，开启 I2C3，并配置 I2C 如下图所示：



cubeMX 中 I2C 配置图

18.4.2 mahony 算法移植

mahony 算法是常见的姿态融合算法，将加速度计，磁力计，陀螺仪共九轴数据，融合解算

出载体四元数，mahony 算法可以到如下网站中下载源码。

<https://x-io.co.uk/open-source-imu-and-ahrs-algorithms/>

下载压缩包 `madgwick_algorithm_c.zip`,解压后移植 MahonyAHRS 文件下的文件到程序工程内。原始文件内有以下参数。

参数	意义	设置值
<code>sampleFreq</code>	采样频率	<code>1000.0f</code> (原始值 <code>512.0f</code>)
<code>twoKpDef</code>	加速度计以及磁力计融合权重 <code>Kp</code> , 值越大, 收敛速度越快。	<code>2.0f*0.5f</code>
<code>twoKiDef</code>	加速度计以及磁力计融合权重 <code>Ki</code> , 值越大, 收敛速度越快。	<code>2.0f*0.0f</code>
<code>twoKp</code>	<code>twoKpDef</code> 宏赋值变量	<code>twoKpDef</code>
<code>TwoKi</code>	<code>twoKiDef</code> 宏赋值变量	<code>twoKiDef</code>
<code>q0,q1,q2,q3</code>	四元数	<code>[1.0,0,0,0]</code>
<code>integralFBx</code>		
<code>integralFBy</code>	误差积分值	<code>0.0f</code>
<code>integralFBz</code>		

为了方便使用，对原先函数进行参数传入上的修改，引入四元数数组变量。

1. MahonyAHRSupdate 函数

```
void MahonyAHRSupdate(float q[4], float gx, float gy, float gz, float ax, float ay, float az, float mx, float my, float mz)
```

函数名	MahonyAHRSupdate
功能	融合陀螺仪，加速度，磁力计数据，更新四元数
函数返回	<code>None</code>
参数 1: <code>q[4]</code>	四元数数组
参数 2,3,4: <code>gx,gy,gz</code>	陀螺仪角速度数据
参数 5,6,7: <code>ax,ay,az</code>	加速度计加速度数据
参数 8,9,10: <code>mx,my,mz</code>	磁力计磁场数据

简单介绍函数实现过程：

1. 对加速度计数据以及磁力计数据进行归一化处理，使得加速度和磁场数据平方和等于 1. 即：

$$a_x^2 + a_y^2 + a_z^2 = 1$$

$$m_x^2 + m_y^2 + m_z^2 = 1$$

2. 旋转磁场数据，计算当前四元数姿态方向。
3. 当前四元数姿态方向与加速度测量方向和磁力计测量方向做差
4. 进行 PI 补偿陀螺仪数据。
5. 根据公式 18-6,18-7,18-8,18-9，使用陀螺仪数据更新四元数数据，具体代码如下：

```
gx *= (0.5f * (1.0f / sampleFreq)); // pre-multiply common factors  
  
gy *= (0.5f * (1.0f / sampleFreq));  
  
gz *= (0.5f * (1.0f / sampleFreq));  
  
qa = q[0];  
  
qb = q[1];  
  
qc = q[2];  
  
q[0] += (-qb * gx - qc * gy - q[3] * gz);  
  
q[1] += (qa * gx + qc * gz - q[3] * gy);  
  
q[2] += (qa * gy - qb * gz + q[3] * gx);  
  
q[3] += (qa * gz + qb * gy - qc * gx);
```

这段程序过程如下：

1. 对陀螺仪的数据乘以 $0.5 * 1 / \text{sampleFreq}$, 其中 `sampleFreq` 为采样频率, $1 / \text{sampleFreq}$ 为更新间隔时间。
2. 将 `q0,q1,q2` 赋值给 `qa,qb,qc`,
3. 根据四元数更新公式 18-6,18-7,18-8,18-9，更新四元数。

对于融合算法的具体过程以及实现原理可以参考网站上的文档《madgwick_internal_report.pdf》。

2. MahonyAHRSupdateIMU 函数

```
void MahonyAHRSupdateIMU(float q[4], float gx, float gy, float gz, float ax, float ay, float az)
```

函数名	MahonyAHRSupdateIMU
功能	融合陀螺仪，加速度，更新四元数
函数返回	None
参数 1: q[4]	四元数组
参数 2,3,4: gx,gy,gz	陀螺仪角速度数据
参数 5,6,7: ax,ay,az	加速度计加速度数据

3. invSqrt 函数

```
float invSqrt(float x)
```

函数名	invSqrt
功能	计算平方根的倒数，即求 $\frac{1}{\sqrt{x}}$
函数返回	平方根的倒数
参数 1:x	待计算的浮点数

4. AHRS_init 函数

```
void AHRS_init(fp32 quat[4], fp32 accel[3], fp32 mag[3])
```

函数名	AHRS_init
功能	根据加速度计，磁力计数据初始化四元数
函数返回	None
参数 1:quat[4]	四元数组
参数 2:accel[3]	加速度计

参数 3:mag[3]	磁场强度数据
-------------	--------

该函数将四元数赋值成[1.0, 0.0f, 0.0f, 0.0f]，为四元数的初始值。

5. AHRS_update 函数

```
void AHRS_update(fp32 quat[4], fp32 time, fp32 gyro[3], fp32 accel[3], fp32 mag[3])
```

函数名	AHRS_update
功能	根据陀螺仪角速度数据，加速度计加速度数据，磁力计磁场数据进行四元数迭代计算
函数返回	None
参数 1: quat[4]	待更新的四元数
参数 2: time	迭代时间，单位 s，由于姿态解算任务为 1ms，故而输入 0.001f
参数 3: gyro[3]	陀螺仪的角速度数据
参数 4: accel[3]	加速度计的加速度数据
参数 5: mag[3]	磁力计的磁场强度数据

该函数调用 MahonyAHRSupdate 函数进行迭代计算。

6. get_angle 函数

```
void get_angle(fp32 q[4], fp32 *yaw, fp32 *pitch, fp32 *roll)
{
    *yaw = atan2f(2.0f*(q[0]*q[3]+q[1]*q[2]), 2.0f*(q[0]*q[0]+q[1]*q[1])-1.0f);
    *pitch = asinf(-2.0f*(q[1]*q[3]-q[0]*q[2]));
    *roll = atan2f(2.0f*(q[0]*q[1]+q[2]*q[3]), 2.0f*(q[0]*q[0]+q[3]*q[3])-1.0f);
}
```

该程序计算公式参考基础学习中的四元数转化成欧拉角的公式 18-3,18-4 和 18-5。

函数名	get_angle
函数功能	根据四元数获取欧拉角
函数返回	None
参数 1: q[4]	四元数数组
参数 2: yaw	Yaw 角的指针
参数 3: pitch	Pitch 角的指针
参数 4: roll	Roll 角的指针

18.4.3 程序流程

介绍完了融合算法以及四元数的相关函数，接下来介绍 SPI 的 DMA 传输过程，使用 SPI 的 DMA 传输，可以节约 CPU 处理时间。由于 SPI 是 MISO，MOSI 同时进行传输数据，故而需要同时开启 SPI 的 RX 和 TX。

1. 开启 SPI 的 DMA 传输函数

```
void SPI1_DMA_enable(uint32_t tx_buf, uint32_t rx_buf, uint16_t ndtr)
```

函数名	SPI1_DMA_enable
函数功能	开启 SPI1 的 DMA 传输
函数返回	None
参数 1: tx_buf	发送数据的地址
参数 2: rx_buf	接收数据的地址
参数 3: ndtr	数据长度

2. SPI 的 DMA 调度函数

使用 SPI 通信时，需要保证同一时刻只有一个设备在通信，即同时只能获取陀螺仪的数

据或者加速度计的数据，故而需要对 SPI 的通信进行调度。置位 gyro_update_flag, accel_update_flag 和 accel_temp_update_flag 三个变量的标志位，代表陀螺仪，加速度计和温度三个传输数据的不同阶段。

变量	阶段
gyro_update_flag	BIT [0]: 进入陀螺仪的 data ready 的下降沿外部中断后置 1; BIT [1]: 成功开启陀螺仪的 SPI 的 DMA 传输; BIT [2]: 已完成陀螺仪的 SPI 的 DMA 传输; BIT [3:7]: 保留。
accel_update_flag	BIT [0]: 进入加速度计的 data ready 的下降沿外部中断后置 1; BIT [1]: 成功开启加速度数据的 SPI 的 DMA 传输; BIT [2]: 已完成加速度数据的 SPI 的 DMA 传输; BIT [3:7]: 保留。
accel_temp_update_flag	BIT [0]: 进入加速度计的 data ready 的下降沿外部中断后置 1， (由于温度寄存器在加速度计内，故而是进入同一中断)； BIT [1]: 成功开启温度数据的 SPI 的 DMA 传输; BIT [2]: 已完成温度数据的 SPI 的 DMA 传输; BIT [3-7]: 保留。

```
static void imu_cmd_spi_dma(void)
```

函数名	imu_cmd_spi_dma
函数功能	根据 xxx_update_flag 的第 0 位决定开启对应的片选和 SPI 的 DMA
函数返回	None
参数	None

3. 主任务唤醒功能

同时为了开启唤醒主任务，使能了外部中断 Line_0，并在中断函数加入唤醒功能。

```

else if(GPIO_Pin == GPIO_PIN_0)

{

    //wake up the task

    //唤醒任务

    if(xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED)

    {

        static BaseType_t xHigherPriorityTaskWoken;

        vTaskNotifyGiveFromISR(INS_task_local_handler, &xHigherPriorityTaskWoken);

        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

    }

}

```

在陀螺仪数据的 SPI 的 DMA 传输完成后，软件开启外部中断 Line_0。

```

if(gyro_update_flag & (1 << IMU_UPDATE_SHFITS))

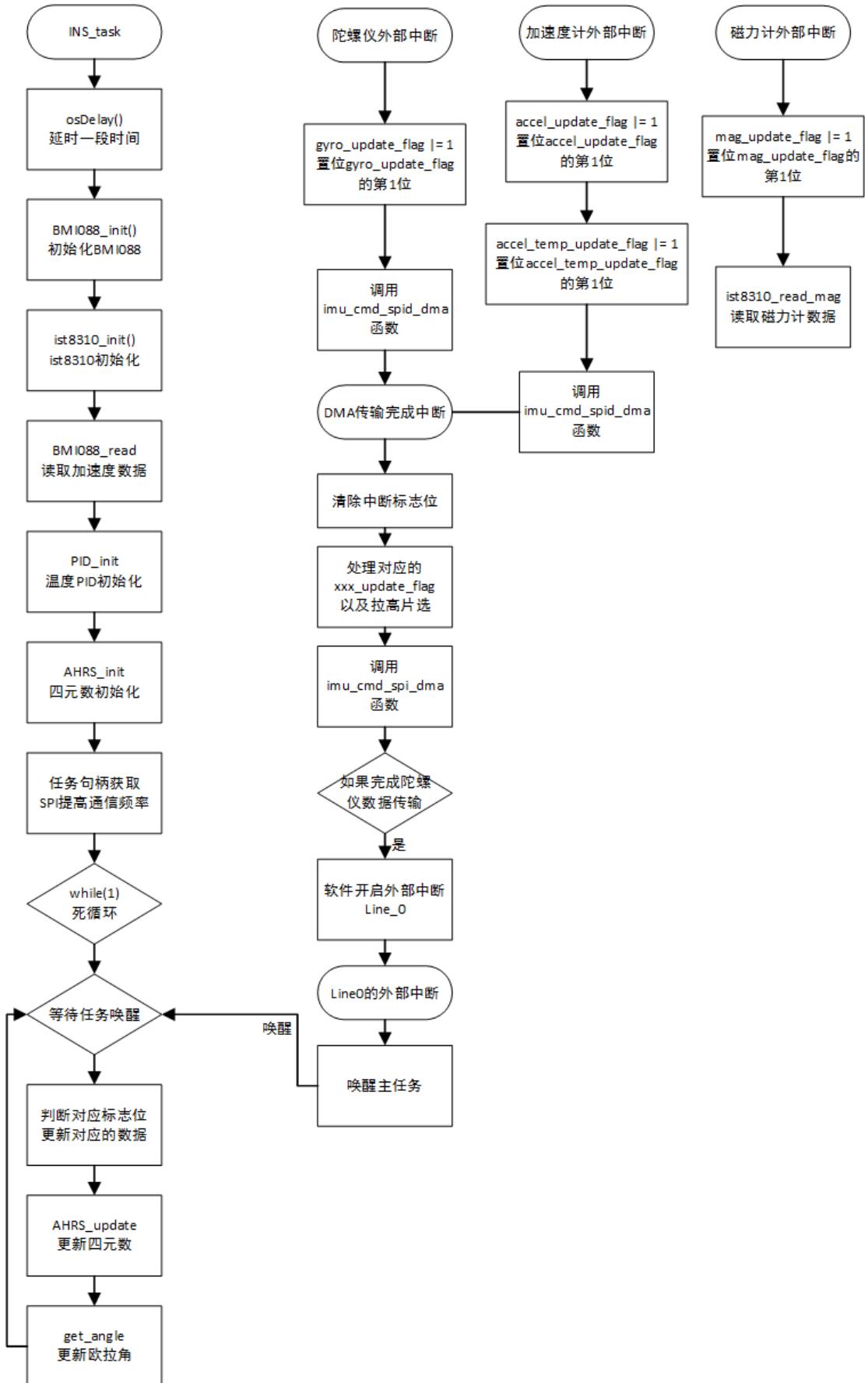
{

    __HAL_GPIO_EXTI_GENERATE_SWIT(GPIO_PIN_0);

}

```

整体过程便如下，主任务在初始化陀螺仪，加速度计和磁力计后，进入等待唤醒状态。程序等待陀螺仪和加速度计的外部中断，在外部中断中使能对应的 SPI 的 DMA 传输。在 SPI 的 DMA 传输完成中断后，引起 DMA 完成中断，其中在陀螺仪数据完成传输后，开启外部中断 Line_0 来唤醒主任务。主任务在唤醒后，处理陀螺仪，加速度和温度数据后，调用 mahony 算法融合九轴数据更新四元数，计算欧拉角，再次进入等待唤醒。



程序流程图

18.4.4 效果展示

进入 Debug 模式, 可以参考 INS_quat, INS_angle, ist8310_real_data, bmi088_real_data 下的数据, 注意 INS_angle 的单位为 rad。

Name	Value	Type
INS_quat	0x20006C9C INS_...	float[4]
[0]	0.975852072	float
[1]	0.0300978851	float
[2]	0.0253171232	float
[3]	-0.206903562	float
INS_angle	0x20006CAC INS_...	float[3]
[0]	-0.417758465	float
[1]	0.0618976578	float
[2]	0.0486940593	float
ist8310_real_data	0x20006C44 &ist...	struct ist8310_real...
status	0x00	unsigned char
mag	0x20006C48	float[3]
[0]	20.1000004	float
[1]	34.2000008	float
[2]	106.200005	float
bmi088_real_data	0x20006C20 &bmi...	struct BMI088_RE...
status	0x00	unsigned char
accel	0x20006C24	float[3]
[0]	0.000897435879	float
[1]	-0.741282046	float
[2]	9.70128155	float
temp	47.25	float
gyro	0x20006C34	float[3]
[0]	0.00319579337	float
[1]	-0.00319579337	float
[2]	-0.00213052891	float
time	3750	float

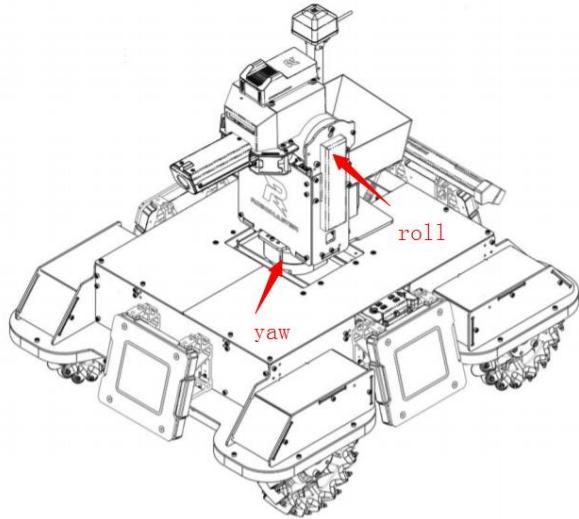
程序效果图

18.5 进阶学习

18.5.1 欧拉角旋转顺序

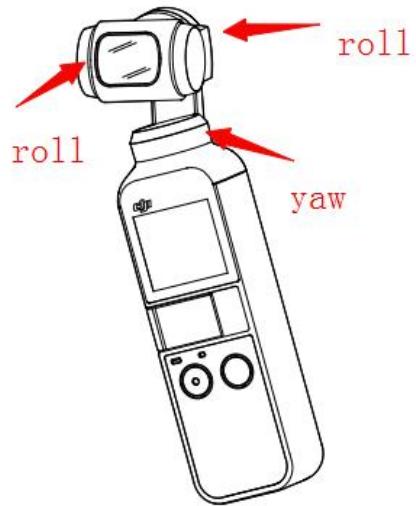
欧拉角旋转顺序分为两种: yaw-pitch-roll 和 yaw-roll-pitch, RoboMaster 机器人底盘到云台先后经过 yaw 轴电机, 再经过 pitch 轴电机; 故而 RoboMaster 机器人欧拉角旋转顺序为

yaw-pitch-roll。



RoboMaster 机器人旋转顺序

而以口袋灵眸为例，从口袋灵眸手柄到摄像头，经过 yaw 轴电机，再经过 roll 轴电机，最后经过 pitch 轴电机，故而口袋灵眸欧拉角旋转顺序为 yaw-roll-pitch。



口袋灵眸旋转顺序

在欧拉角旋转顺序为 yaw-roll -pitch，四元数转化成欧拉角的公式为：

$$\text{yaw} = \arctan\left(\frac{q_0 * q_3 - q_1 * q_2}{q_0 * q_0 + q_2 * q_2 - 0.5}\right)$$
$$\text{pitch} = \arctan\left(\frac{q_0 * q_2 - q_1 * q_3}{q_0 * q_0 + q_3 * q_3 - 0.5}\right)$$
$$\text{roll} = \arcsin(2 * q_0 * q_1 + 2 * q_2 * q_3)$$

而程序中采用哪套公式，需要和云台结构相关，如果是和 RoboMaster 机器人一样，是 yaw-pitch-roll 旋转顺序，则采用第一套公式，如果和口袋灵眸一样，是 yaw-roll-pitch，则采用第二套公式。

18.6 课程总结

通过本课程的学习，掌握欧拉角与四元数的转化公式，以及四元数更新公式，学习 mahony 算法，此外还学习 SPI 的 DMA 传输。姿态角是机器人重要的反馈数据，在 RoboMaster 机器人的云台控制，自动瞄准等领域中有着重要的应用。

19. 云台控制任务

19.1 知识要点

- 机器人云台的结构
- 串级 PID 控制
- 机器人云台角度环和速度环串级 PID 控制

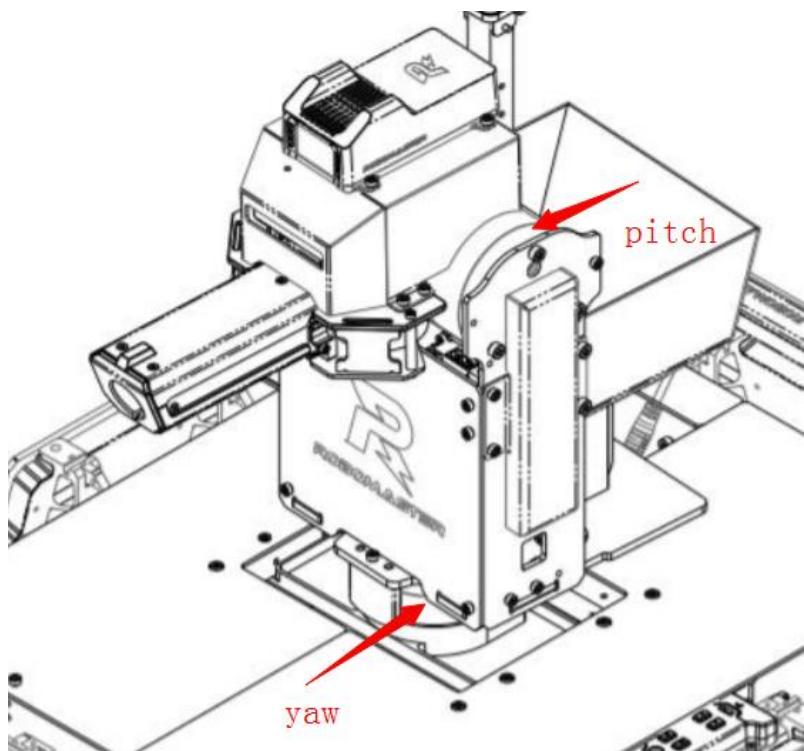
19.2 课程内容

云台是机器人上的增稳机构，用于减少机器人顶部结构的振动，增稳图像。在 RoboMaster 比赛中，机器人都配有一个云台结构，用于搭载图传，发射激光等装置。在本节课程中，学习云台相关的结构与控制，学习云台角度环和角速度环双环串级 PID 控制。

19.3 基础学习

19.3.1 机器人云台的结构

RoboMaster 机器人云台是一种两轴云台结构，可以控制 yaw 轴，pitch 轴的旋转。外观如图所示：

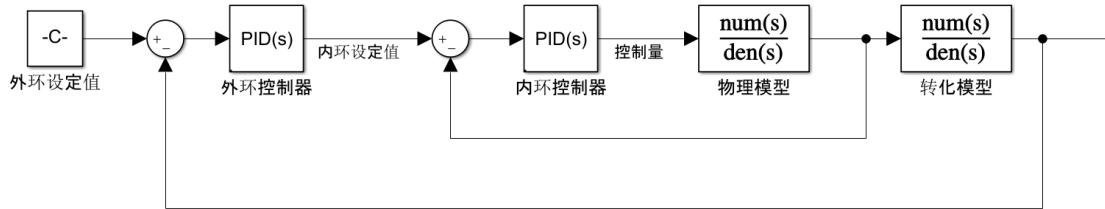


云台结构图

19.3.2 串级 PID

串级 PID 是通过两个控制器串级控制，串级控制往往是对同一自由度的不同物理量进行控制，例如机器人的云台控制是使用角速度环和角度环串级控制，角速度和角度均是属于旋转自由度，角速度是角度的微分。两个控制环节分为外环控制环节和内环控制，一般角度属于外环控制环节，角速度属于内环控制环节。

串联是将外环的输出与内环的输入相连，外环角度环的输入是控制目标，外环角度环的输出是设定的角速度，为内环角速度环的输入。



19.4 程序学习

19.4.1 can 发送电机控制函数回顾

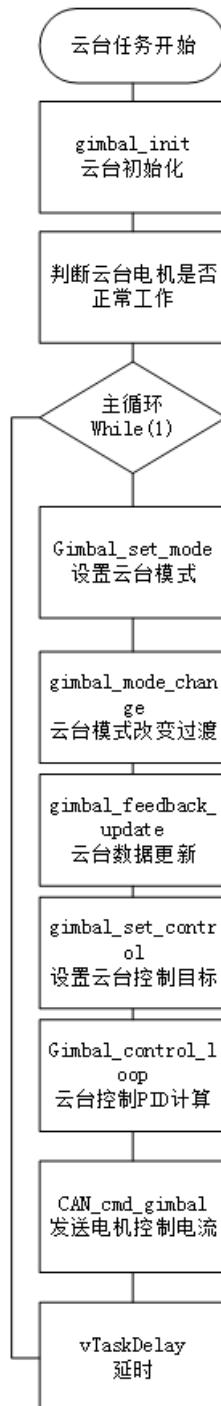
在 CAN 通信章节中，学习到如何使用 CAN 通信进行控制电机。云台电机分配的 ID 分别是 0x205, 0x206, 0x207 和 0x208，它们都是由 ID 为 0x1FF 的 CAN 包进行控制，其中默认 0x205 为 yaw 轴电机，0x206 为 pitch 轴电机，0x207 为拨弹电机，而在 CAN_receive.c 中封装云台控制函数 CAN_cmd_gimbal。

函数名	CAN_cmd_gimbal
函数功能	发送 ID 为 0x205-0x208 的电机电流控制值
函数返回	None
参数 1: yaw	ID 为 0x205 的电机电流控制值，范围[-30000,30000]
参数 2: pitch	ID 为 0x206 的电机电流控制值，范围[-30000,30000]

参数 3: shoot	ID 为 0x207 的电机电流控制值, 范围[-16384,16384]
参数 4: res	ID 为 0x208 的电机电流控制值, 为保留字节

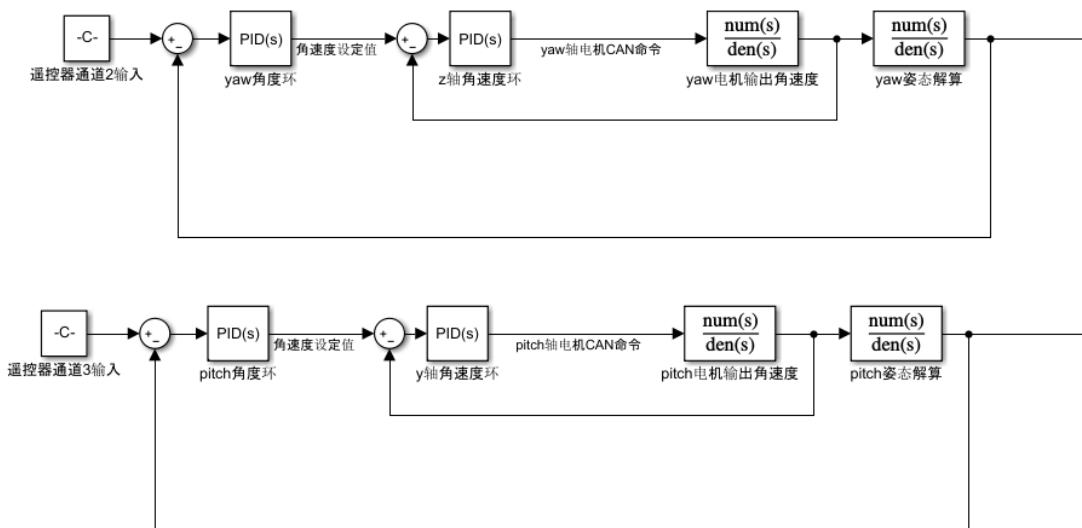
19.4.2 程序流程讲解

使用 freeRTOS 创建云台任务，云台任务主循环流程如下：



流程	功能
gimbal_init	云台初始化，主要初始化电机角度 PID，角速度 PID，初始化遥控器指针，初始化姿态角度指针，初始化电机数据指针。
gimbal_set_mode	根据遥控器的开关值决定云台控制行为，根据云台控制行为决定云台控制模式。
gimbal_mode_change_control_transit	云台控制模式切换过程中数据保存，以保证控制切换的控制目标平稳过渡。
gimbal_feedback_update	云台电机反馈速度，姿态角度，角速度更新。
gimbal_set_control	根据遥控器和鼠标的输入，计算角度的控制目标。
gimbal_control_loop	云台根据不同控制模式，计算不同 PID。
CAN_cmd_gimbal	发送电机电流控制值

机器人云台常见的控制路线如图所示，由遥控器以及鼠标的通道值计算出角度 yaw, pitch 的控制目标值，由云台角度 PID 计算对应轴的角速度设定值，由角速度 PID 计算出电机的 CAN 命令控制量，再通过 CAN 总线发送到电机。



19.4.3 关键函数讲解

以下讲解几个主要函数：

1. `gimbal_set_mode` 函数和 `gimbal_behaviour_mode_set` 函数

`gimbal_set_mode` 函数调用 `gimbal_behaviour_mode_set` 函数完成设置控制模式。而在 `gimbal_behaviour_mode_set` 函数通过遥控器的开关决定云台的不同行为，然后通过不同行为决定不同云台控制方式，行为模式和控制模式可以参考下表。

```
static void gimbal_set_mode(gimbal_control_t *set_mode)
{
    gimbal_behaviour_mode_set(set_mode);
}

static void gimbal_behaviour_set(gimbal_control_t *gimbal_mode_set)
{
    ...
    //开关控制 云台状态

    if (switch_is_down(gimbal_mode_set->gimbal_rc_ctrl->rc.s[GIMBAL_MODE_CHANNEL]))
    {
        gimbal_behaviour = GIMBAL_ZERO_FORCE;
    }

    else if (switch_is_mid(gimbal_mode_set->gimbal_rc_ctrl->rc.s[GIMBAL_MODE_CHANNEL]))
    {
        gimbal_behaviour = GIMBAL_RELATIVE_ANGLE;
    }

    else if (switch_is_up(gimbal_mode_set->gimbal_rc_ctrl->rc.s[GIMBAL_MODE_CHANNEL]))
    {
        gimbal_behaviour = GIMBAL_ABSOLUTE_ANGLE;
    }

    ...
}
```

行为模式	云台表现以及应用
GIMBAL_ZERO_FORCE	云台电机 CAN 发送的控制值为 0，云台表现为无力状态，应用于遥控器开关处于下位，需要云台停止运动的场合。
GIMBAL_INIT	云台初始化模式，云台先缓慢抬起 pitch 轴，之后旋转 yaw 轴至云台中点，应用于云台从停止运动到开启运动控制的过程阶段，防止云台在开电时过度运动导致机械机构损坏。
GIMBAL_CALC	云台校准模式，用于云台计算云台中值的场合，云台先放下 pitch 轴，再抬起 pitch 轴，再逆时针旋转 yaw 轴，最后顺时针旋转 yaw 轴。在这个过程中，采集电机的反馈角度，用于计算云台中值。
GIMBAL_ABSOLUTE_ANGLE	云台使用姿态解算出的姿态角进行角度控制，由于姿态角是相对地面坐标系，不随底盘的姿态角度而不变化，适用于机器人正常运动控制。
GIMBAL_RELATIVE_ANGLE	云台使用电机反馈的角度进行角度控制，由于电机角度是相对底盘坐标系，跟随底盘的姿态角度，适用于机器人在特殊场合下的控制运动控制。
GIMBAL_MOTIONLESS	云台静止不动，保证原先的电机角度控制，适用于机器人在长时间静止不动的控制运动控制，减少陀螺仪漂移造成的影响。

云台电机控制模式分为三种，如下表所示。

控制模式	功能以及应用
GIMBAL_MOTOR_RAW	云台电机控制值直接发送 CAN 包，是 GIMBAL_CALI 和 GIMBAL_ZERO_FORCE 选择的电机控制模式。
GIMBAL_MOTOR_GYRO	云台电机的控制目标是陀螺仪解算的角度，是 GIMBAL_ABSOLUTE_ANGLE 选择的电机控制模式。

控制模式	功能以及应用
GIMBAL_MOTOR_ENCONDE	云台电机的控制目标是电机反馈的角度，是 GIMBAL_RELATIVE_ANGLE，GIMBAL_MOTIONLESS, GIMBAL_INIT 选择的电机控制模式。

2. gimbal_set_control 函数和 gimbal_behaviour_conotrol_set 函数

gimbal_set_control 函数调用 gimbal_behaviour_conotrol_set 函数得到云台两个自由度运动的控制目标值。gimbal_behaviour_control_set 函数根据不同的云台行为模式调用不同函数，在对应的函数设置不同控制量。

```
static void gimbal_set_control(gimbal_control_t *set_control)
{
    fp32 add_yaw_angle = 0.0f;
    fp32 add_pitch_angle = 0.0f;

    gimbal_behaviour_control_set(&add_yaw_angle, &add_pitch_angle, set_control);

    //yaw 电机模式控制
    if (set_control->gimbal_yaw_motor.gimbal_motor_mode == GIMBAL_MOTOR_RAW)
    {
        //raw 模式下，直接发送控制值
        set_control->gimbal_yaw_motor.raw_cmd_current = add_yaw_angle;
    }

    else if (set_control->gimbal_yaw_motor.gimbal_motor_mode == GIMBAL_MOTOR_GYRO)
    {
        //gyro 模式下，陀螺仪角度控制
        gimbal_absolute_angle_limit(&set_control->gimbal_yaw_motor, add_yaw_angle);
    }

    else if (set_control->gimbal_yaw_motor.gimbal_motor_mode == GIMBAL_MOTOR_ENCONDE)
```

```

{
    //enconde 模式下，电机编码角度控制

    gimbal_relative_angle_limit(&set_control->gimbal_yaw_motor, add_yaw_angle);

}
...
}

```

3. gimbal_absolute_angle_limit 函数

`gimbal_absolute_angle_limit` 函数功能是云台在陀螺仪角度控制模式下，限制最大角度以防止云台旋转超过最大角度，损坏限位机构。

- 1) 计算当前设定目标角度与当前角度的误差角度；
- 2) 判断电机反馈的角度加上误差角度和增加角度是否会超过最大相对角度；
- 3) 如果超过最大角度，修改增加角度；
- 4) 将增加角度加到设定目标角度上。

```

static void gimbal_absolute_angle_limit(gimbal_motor_t *gimbal_motor, fp32 add)

{
    static fp32 bias_angle;

    static fp32 angle_set;

    if (gimbal_motor == NULL)

    {
        return;
    }

    //now angle error

    //当前控制误差角度

    bias_angle = rad_format(gimbal_motor->absolute_angle_set - gimbal_motor->absolute_angle);

    //relative angle + angle error + add_angle > max_relative_angle

    //云台相对角度+ 误差角度 + 新增角度 如果大于 最大机械角度

```

```
if (gimbal_motor->relative_angle + bias_angle + add > gimbal_motor->max_relative_angle)

{

    //如果是往最大机械角度控制方向

    if (add > 0.0f)

    {

        //calculate max add_angle

        //计算出一个最大的添加角度，

        add = gimbal_motor->max_relative_angle - gimbal_motor->relative_angle - bias_angle;

    }

}

else if (gimbal_motor->relative_angle + bias_angle + add < gimbal_motor->min_relative_angle)

{

    if (add < 0.0f)

    {

        add = gimbal_motor->min_relative_angle - gimbal_motor->relative_angle - bias_angle;

    }

}

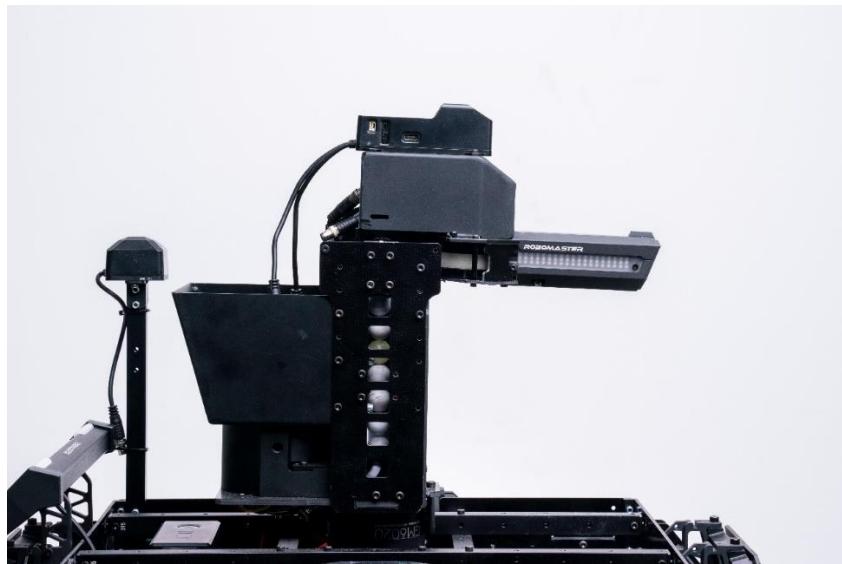
angle_set = gimbal_motor->absolute_angle_set;

gimbal_motor->absolute_angle_set = rad_format(angle_set + add);

}
```

19.5 效果展示

使用遥控器可以进行云台控制，云台如图所示保持水平状态。



19.6 课程总结

云台是 RoboMaster 机器人的关键组件，云台结构一般为 yaw-pitch 两轴云台结构，可以进行 yaw 和 pitch 轴控制，用于增稳图像和承载发射机构等。云台性能将直接影响到操作体验和自动瞄准，为了提高响应，提高云台控制性能，往往采用串级 PID 控制。

20. 机器人功能介绍

20.1 知识要点

- 机器人总体功能介绍
- 机器人其他任务介绍

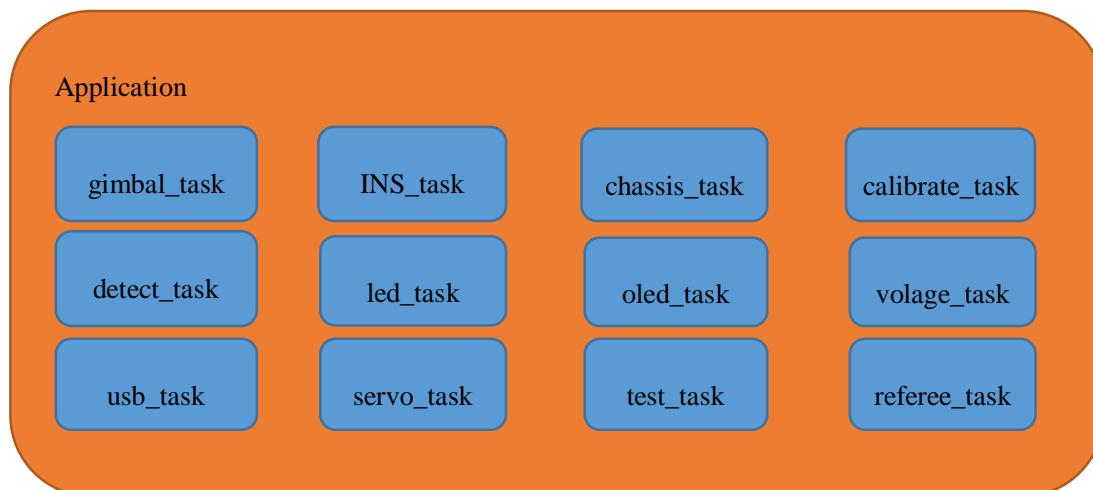
20.2 课程内容

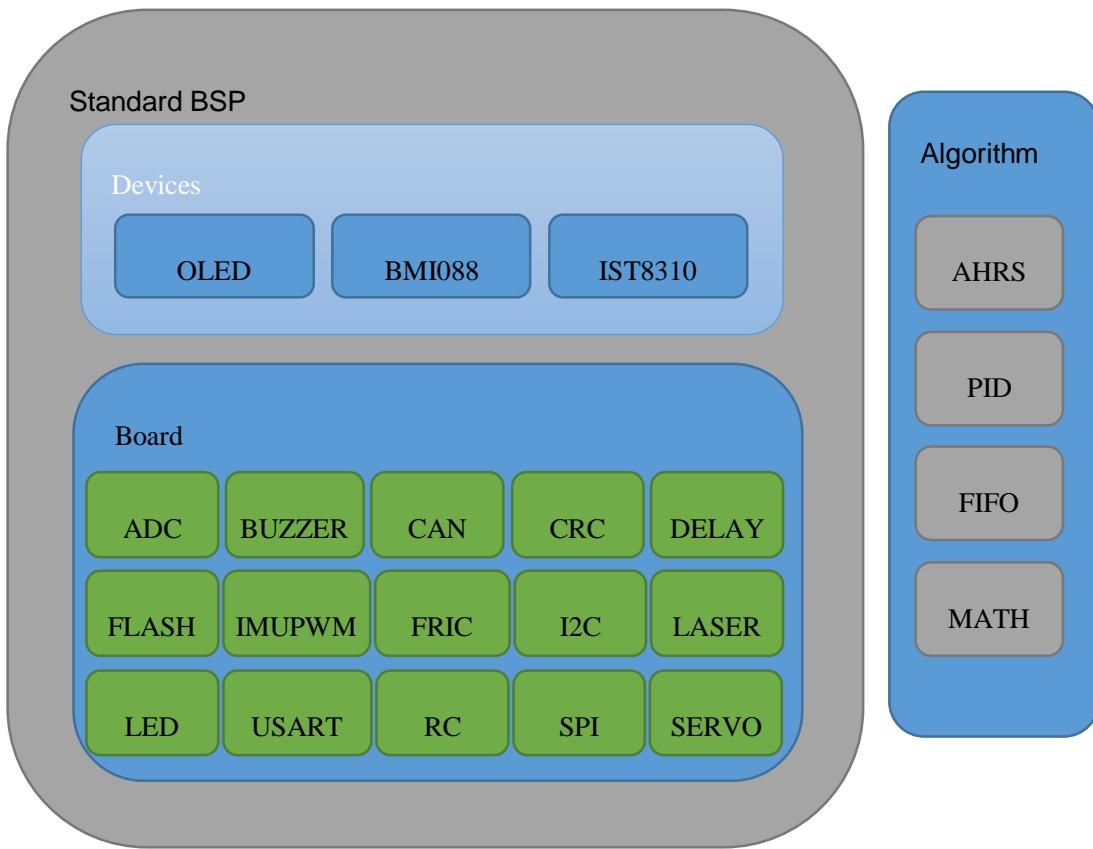
RoboMaster 机器人除了基本的底盘运动控制，云台运动控制等功能，还需要其他辅助功能来更好在比赛中发挥作用，例如离线检查，校准保存数据，裁判系统串口解析等功能。本章内容将介绍机器人的总体功能以及部分任务函数讲解。

20.3 基础学习

20.3.1 机器人软件框架

机器人的软件框架如下所示，主要包括 **application** 层以及 **BSP** 层。**BSP** 层提供对底层硬件功能的封装以及 bmi088, ist8310 和 OLED 的驱动实现，算法层实现姿态解算算法，PID 控制算法，FIFO 数据结构以及常见的数学处理函数。**Application** 层主要为任务函数实现，包括云台任务，姿态解算任务，底盘任务，校准任务，离线检测任务等等。





20.3.2 功能介绍

以下针对 application 层进行简单的介绍，外设调用关系可以参考最后的调用关系小节。

1. 校准功能 (`calibrate_task`): 提供云台校准，陀螺仪零漂校准，底盘重设 ID 的功能
2. 底盘控制功能 (`chassis_task`): 完成底盘的麦轮运动控制，底盘功率控制，提供 4 种控制模式：跟随云台角度闭环控制，跟随底盘角度闭环控制，底盘旋转无角度闭环控制，原生 CAN 控制。
3. 离线判断功能 (`detect_task`): 根据数据反馈的时间戳来判断设备是否离线。
4. 云台控制功能 (`gimbal_task`): 完成云台的角度控制。提供 3 种控制模式，陀螺仪角度控制，电机码盘角度控制，原生 CAN 控制。
5. 姿态解算功能 (`ins_task`): 完成陀螺仪加速度计的角度融合，解算欧拉角。
6. LED 的 RGB 切换 (`led_trigger_task`): 使用三色 LED 完成 RGB 显示，呼吸灯效。用于显示程序是否死机。
7. OLED 显示功能 (`oled_task`): 将电池电量，设备错误信息显示出来，方便使用者定位问题。
8. 裁判系统数据解析 (`referee_usart_task`): 使用单字节解析裁判系统数据，适用于 2019

年裁判系统，裁判系统需要升级总决赛版本。

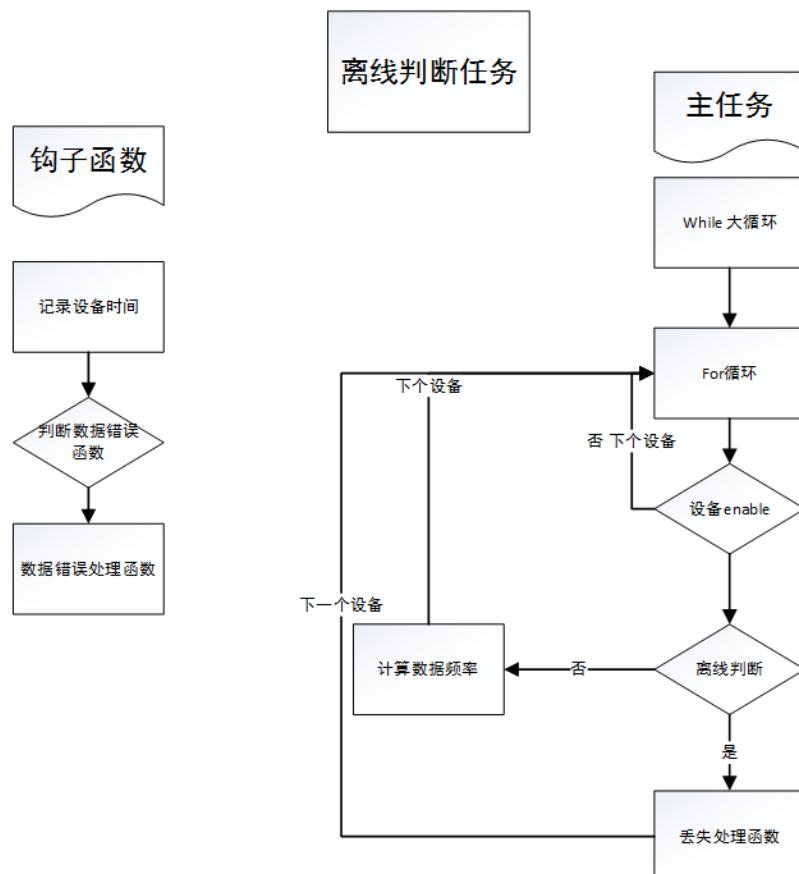
9. 遥控器数据解析 (`remote_control`): 使用串口空闲中断函数，解析接收机发送的数据。
10. 舵机控制 (`servo_task`): 将 4 个空闲的 PWM 输出舵机信号，通过按键进行控制，方便之后添加弹仓控制或者简易的机械装置。
11. 射击控制 (`shoot`): 控制下供弹装置，完成发射逻辑。
12. 电源采样 (`voltage_task`): 采样电源电压，并估计当前电池电量，作为简单电量判断，用于电池在机器人内部，不方便观测电量的场合。

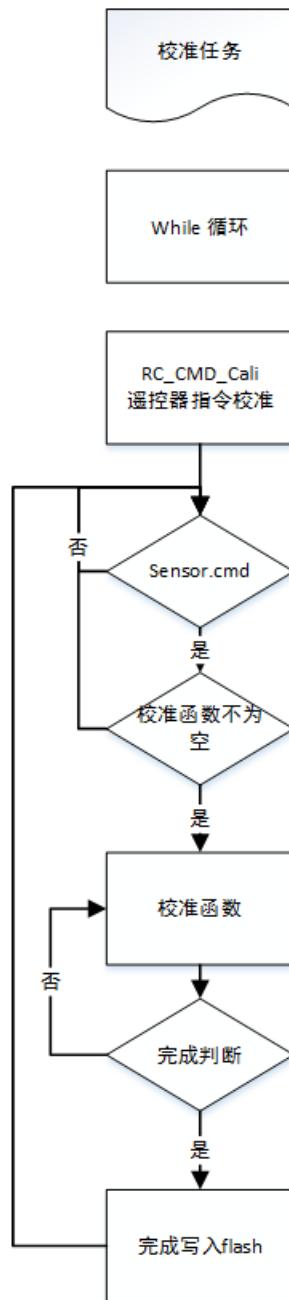
20.4 程序学习

我们在之前的章节中介绍了云台控制任务，姿态解算任务，底盘控制任务。以下仅介绍离线检测任务，校准任务和裁判系统解包任务，以及 OLED 任务。

20.4.1 校准任务以及离线任务介绍

校准任务主要完成陀螺仪零漂校准，云台中值校准，底盘进入快速设置 ID 模式。模块离线判断任务主要通过判断模块的数据发送时间，与当前系统时间的差值来判断是否掉线。这两个任务都是主要通过指针函数来完成。流程图如下所示。





其中离线检测任务使用到数据结构体如下所示：

```

typedef __packed struct
{
    uint32_t new_time;
    uint32_t last_time;
    uint32_t lost_time;
    uint32_t work_time;
}

```

```

    uint16_t set_offline_time : 12;

    uint16_t set_online_time : 12;

    uint8_t enable : 1;

    uint8_t priority : 4;

    uint8_t error_exist : 1;

    uint8_t is_lost : 1;

    uint8_t data_is_error : 1;

    fp32 frequency;

    bool_t (*data_is_error_fun)(void);

    void (*solve_lost_fun)(void);

    void (*solve_data_error_fun)(void);

} error_t;

```

变量	功能
New_time	设备数据更新时间
Last_time	设备数据上次更新时间
Lost_time	设备离线丢失时间
Work_time	设备上线时间
Set_offline_time	判断设备离线的数据中断时间，用于设备数据更新时间超过该设定值后判断为设备离线
Set_online_time	设备上线后稳定时间，用于设备上线后稳定该设定值后判断为正常
enable	设备使能
Priority	设备优先级

变量	功能
Error_exist	设备是否出错，包括离线和数据异常
Is_lost	设备是否离线
Data_is_error	设备数据是否错误
frequency	设备更新频率
Data_is_error_fun	设备数据判断错误的函数指针
Solve_lost_fun	设备离线修复的函数指针
Solve_data_error_fun	设备数据错误修复的函数指针

校准任务中使用到的数据结构体如下所示。

```
typedef __packed struct
{
    uint8_t name[3];                                //device name

    uint8_t cali_done;                             //0x55 means has
been calibrated

    uint8_t flash_len : 7;                          //buf lenght

    uint8_t cali_cmd : 1;                           //1 means to run cali
hook function,

    uint32_t *flash_buf;                           //link to device
calibration data

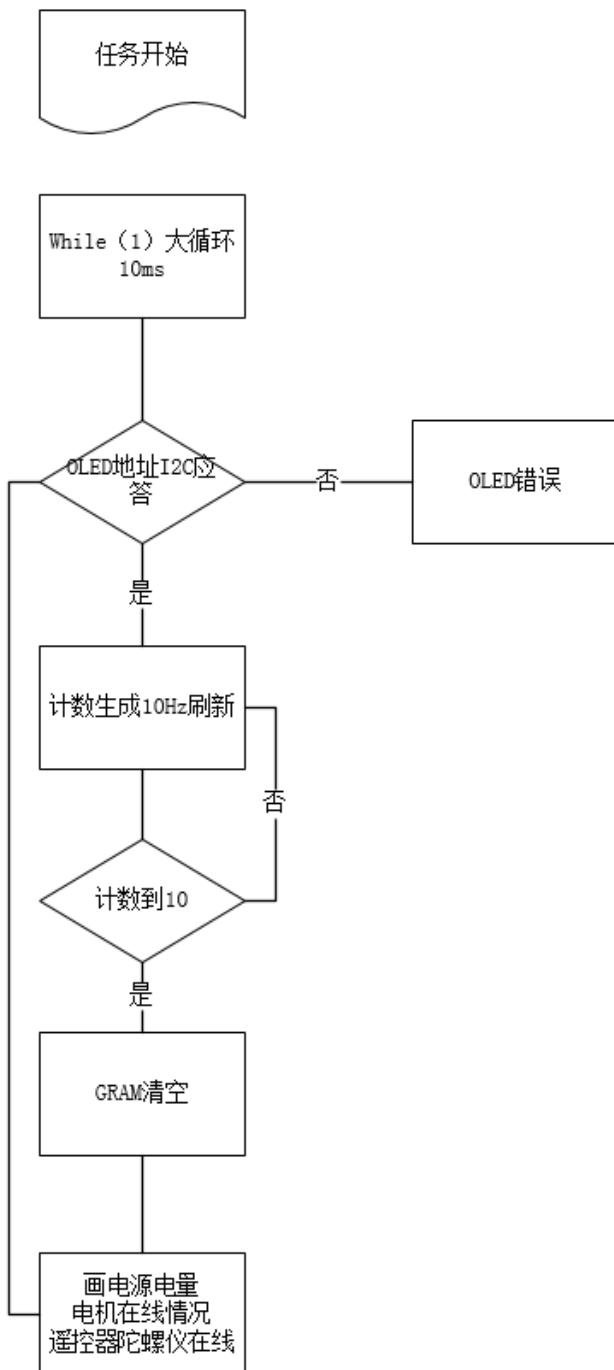
    bool_t (*cali_hook)(uint32_t *point, bool_t cmd); //cali function
} cali_sensor_t;
```

变量	功能
Name	设备名
Cali_done	设备校准标志位，为 0x55 代表已校准
Flash_len	校准数据长度
Cali_cmd	校准使能
Flash_buf	校准数据的数据指针
Cali_hook	校准函数的函数指针

20.4.2 OLED 任务框架

OLED 任务通过 100Hz 查询 oled 的 I2C 地址来确认 OLED 连接情况，并以 10Hz 刷新 OLED 屏幕。OLED 支持 SSD1306 驱动的单色和双色模块，对于单色和双色模块，请在 OLED.C 文件中修改对应的宏定义即可，如下所示：

```
#define OLED_ONE_COLOR  
  
//#define OLED_TWO_COLOR
```



20.4.3 裁判系统串口协议解包

裁判系统通过电源模块用户串口发送数据，包括机器人 ID，底盘功率，枪口热量等信息。数据在传输过程中，由于线缆长度过程，外部电压不稳定等因素导致数据错误。为了能判断出错误信息，需要经过 CRC 校验算法来判断数据正确性。裁判系统串口协议如下：

frame_header (5-byte)	cmd_id(2-byte)	data(n-byte)	frame_tail (2-byte, CRC16, whole package check)
-----------------------	----------------	--------------	---

分为帧头， cmd_id， 数据段， 帧尾四部分， 详细介绍参考《裁判系统串口协议》。

CRC 是校验的一种算法， 采用二进制除法相除所得到的余数。而关于 CRC 校验提供以下函数。

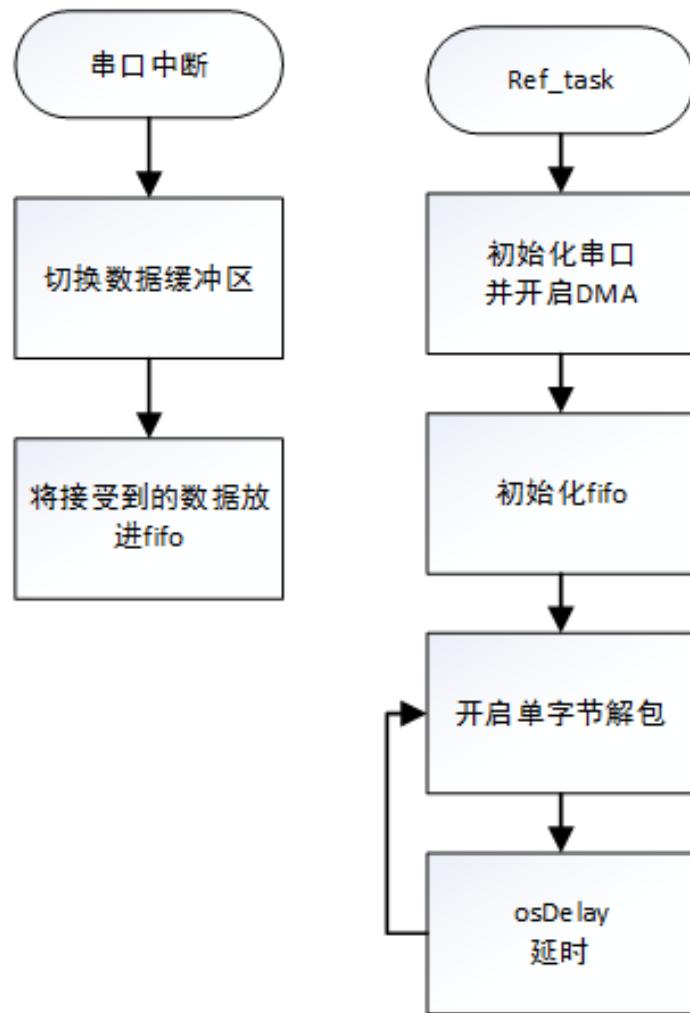
```
uint32_t verify_CRC8_check_sum(unsigned char *pch_message, unsigned int dw_length);
uint32_t verify_CRC16_check_sum(uint8_t *pchMessage, uint32_t dwLength);
```

函数名	Verify_CRC8_check_sum Verify_CRC16_check_sum
函数作用	校验一段数据是否通过 CRC 校验
返回值	通过返回 true 不通过返回 false
参数 1: pch_message	数据指针
参数 2: dw_length	数据长度

```
void append_CRC8_check_sum(unsigned char *pch_message, unsigned int dw_length)
void append_CRC16_check_sum(uint8_t * pchMessage, uint32_t dwLength)
```

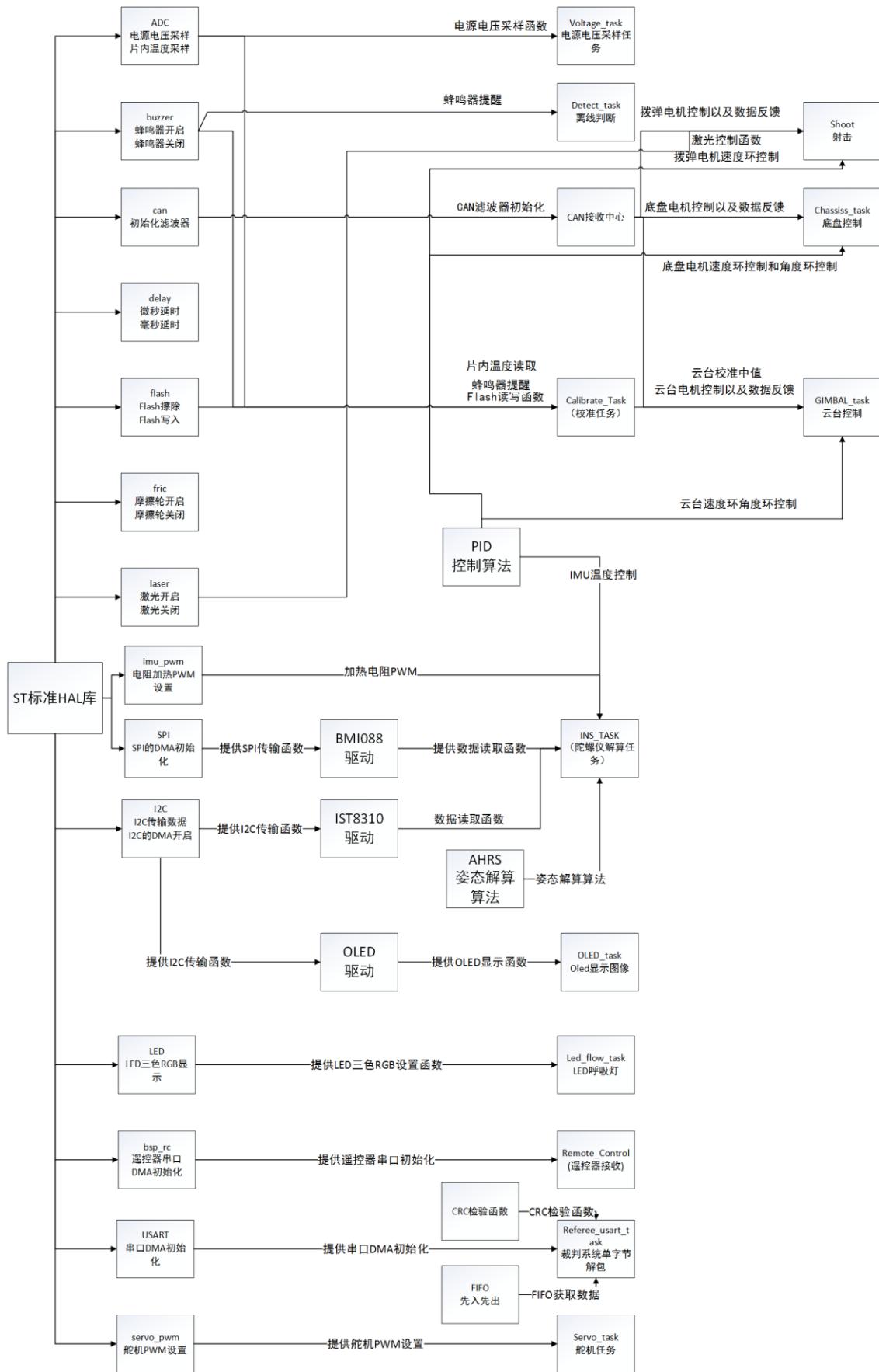
函数名	append_CRC8_check_sum append_CRC16_check_sum
函数作用	添加 CRC 校验值在一段数据的结尾
返回值	None
参数 1: pch_message	数据指针
参数 2: dw_length	数据长度

裁判系统系统解包过程采用单字节解包过程，解包过程如下。



20.4.4 外设调用关系

在所有任务使用外设的调用关系，如图所示：



20.4.5 效果展示

使用遥控器可以进行底盘和云台的运动控制，机器人如图所示：



20.5 课程总结

本章介绍了机器人整体架构，功能特点以及机器人的辅助功能任务，例如离线检测任务，校准任务，裁判系统解包任务等。这些辅助功能能够使得机器人在比赛中提高稳定性，方便操作。



邮箱: robomaster@dji.com

论坛: <http://bbs.robomaster.com>

官网: <http://www.robomaster.com>

电话: 0755-36383255 (周一至周五10:30-19:30)

地址: 广东省深圳市南山区西丽镇茶光路1089号集成电路设计应用产业园2楼202