

特别声明：

此教程基于官方C板教程，缺少的相关知识参考：《RoboMaster开发板C型嵌入式软件教程文档.pdf》

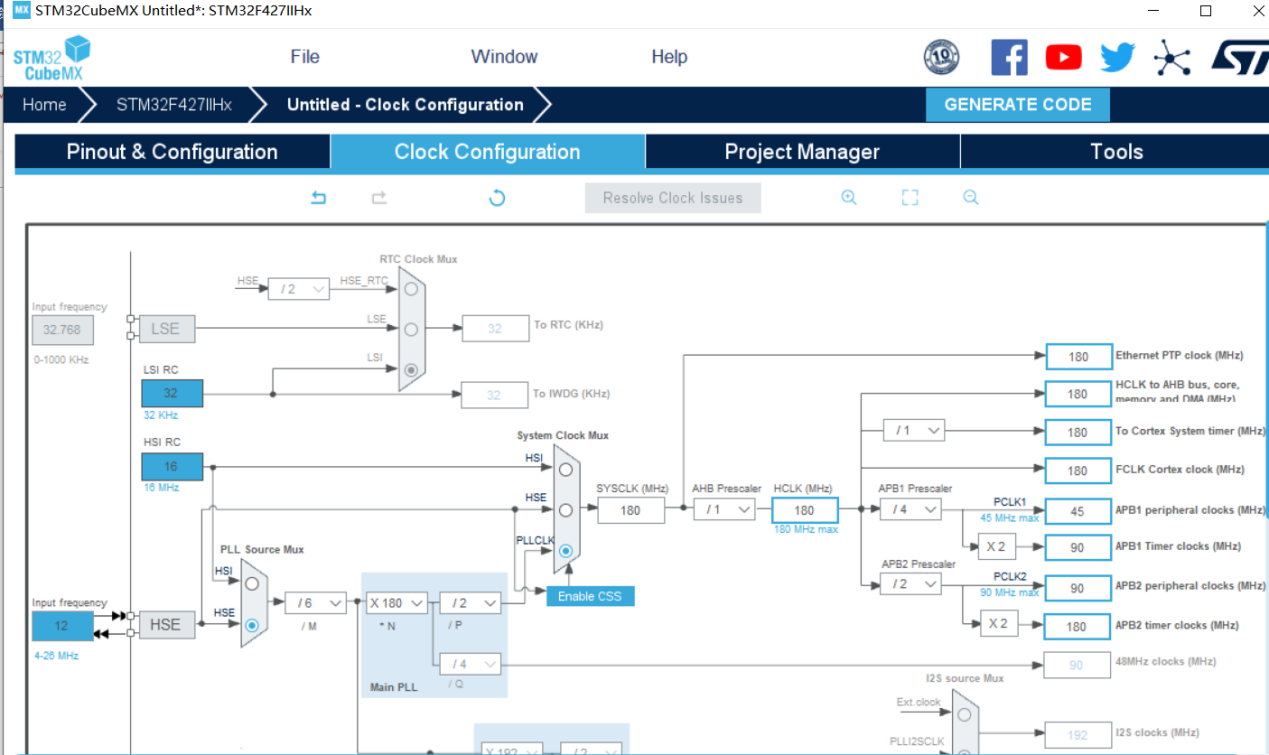
此页放目录

# **cubeMX配置**

1. New Project->F427IIH6;
2. .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；

1. 顶部的 Clock Configuration，进行主频配置；将 Input frequecncy 设置为 12，点击旁边的 HSE 圆形按钮，配置/M 为/6，配置\*N 为 X180，配置/P 为/2 ，选择 PLLCLK 圆形按钮，配置 APB1 Prescaler 为/4，配置 APB2 Prescaler 为/2；



1. 点击顶部的 Pinout & Configuartion，选择 SYS，在 Debug 下拉框中选择 Serial Wire；
2. 点击顶部的 Project Manager，给工程起名，选择存放目录，在 Toolchain/IDE 中选择 MDKARM V5.32；
3. 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；

1. 点击顶部的 GENERATE CODE，等待代码生成，打开工程。

# **点亮LED**

1. 通过原理图可以看出LED\_G、LED\_R为PF14,PE11;



1. 在 cubeMX 中配置 GPIO 为输出模式，在 cubeMX 找到对应引脚，配置GPIO\_Output模式;
2. 在 cubeMX 中修改对应引脚的名字。在左侧找到 System core->GPIO； 找到对应的 GPIO，例如 PF14； 在下方的配置单中 user label 填写命名;
3. 生成代码,点击 GENERATE CODE 按键。
4. **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. 程序流程：

程序开始-》HAL\_Init初始化-》SystemClock\_Config时钟配置-》MX\_GPIO\_Init引脚配置-》while(1)输出高电平

# **闪烁LED**

1. 采用PF14,PE11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；

1. 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。可选的输出速度分为 Low，Medium，High，Very High 四档，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. **HAL\_Delay 函数**

**\_\_weak void HAL\_Delay(uint32\_t Delay)（使用\_weak 修饰符说明该函数是可以用户重定义的）**

**函数名 HAL\_Delay**

**函数作用 使系统延迟对应的毫秒级时间**

**返回值 void**

**参数 Delay，对应的延迟毫秒数，比如延迟 1 秒就为 1000**

1. **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**

1. 计数延时：

**void user\_delay\_us(uint16\_t us)**

**{**

**for(; us > 0; us--)**

**{**

**for(uint8\_t i = 50; i > 0; i--)**

**{**

**;**

**}**

**}**

**}**

void user\_delay\_ms(uint16\_t ms)

{

for(; ms > 0; ms--)

{

user\_delay\_us(1000);

}

}

1. nop 延时：

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();

\_\_nop();

\_\_nop();

\_\_nop();

\_\_nop();

\_\_nop();

\_\_nop();

\_\_nop();

\_\_nop();

}

}

}

void nop\_delay\_ms(uint16\_t ms)

{

for(; ms > 0; ms--)

{

nop\_delay\_us(1000);

}

}

1. HAL\_Delay延时：

HAL\_Delay函数的实现是基于滴答计时器（Systick）。

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。

1. 程序流程：

程序开始-》HAL\_Init初始化-》SystemClock\_Config时钟配置-》MX\_GPIO\_Init引脚配置-》while(1)延时500ms，翻转LED引脚的电平

# **定时器闪烁 LED**

1. 驱动 LED 的 GPIO 配置为LED\_G和LED\_R;
2. 在左侧的标签页中选择 Timer，点击标签页下的 TIM1；
3. 在弹出的 TIM1 Mode and Configuration 中，在 ClockSouce 的右侧下拉菜单中选中

Internal Clock；

1. 接下来需要配置 TIM1 的运转周期。需要打开 Clock Configuration；

通过查阅数据手册资料，可以知道 TIM1 的时钟源来自 APB2 总线；

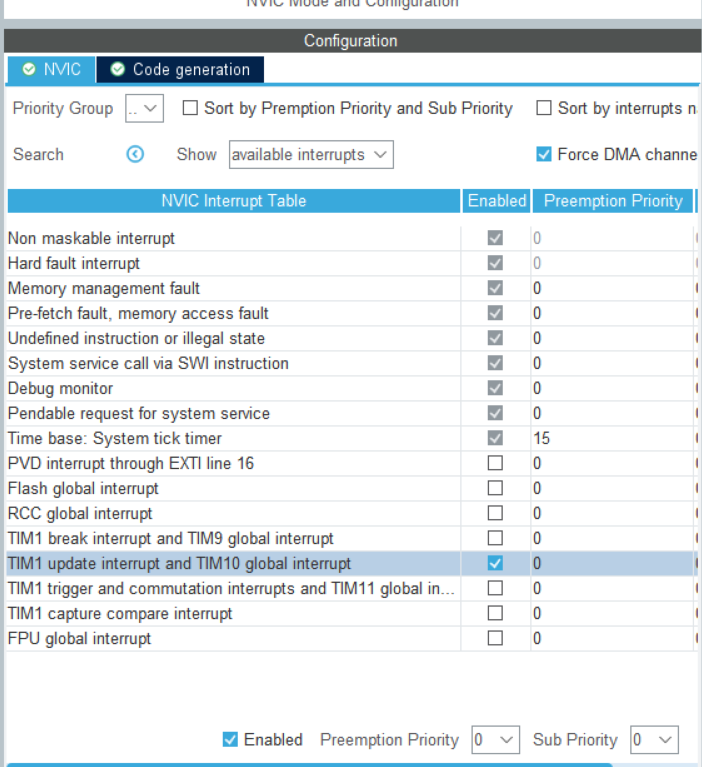
APB2 Timer clocks（MHz）为 180MHz，这意味着提供给TIM1 预分频寄存器的频率就是 180MHz；

500ms 对应的频率为 2Hz，为了得到2Hz 的频率，可以将分频值设为 18000-1，重载值设为 5000-1，则可以计算出定时器触发频率为

1. 在 cubeMX 的 NVIC 标签页下可以看到当前系统中的中断配置

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。

这里为定时器 1 的中断保持默认的 0，0 优先级。



1. 定时器回调函数介绍

**HAL\_TIM\_PeriodElapsedCallback函数**

void HAL\_TIM\_PeriodElapsedCallback(TIM\_HandleTypeDef \*htim)

{

if(htim == &htim1)

{

//500ms trigger

bsp\_led\_toggle();

}

}

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。

bsp\_led\_toggle 函数翻转LED引脚电平

1. **HAL\_TIM\_Base\_Start 函数 仅让定时器以定时功能工作**

**HAL\_StatusTypeDef HAL\_TIM\_Base\_Start(TIM\_HandleTypeDef \*htim)**

**函数名 HAL\_TIM\_Base\_Start**

**函数作用 使对应的定时器开始工作**

**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 \*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**

1. **HAL\_TIM\_Base\_Start\_IT 函数 使用定时中断**

**HAL\_StatusTypeDef HAL\_TIM\_Base\_Start\_IT(TIM\_HandleTypeDef \*htim)**

**函数名 HAL\_TIM\_Base\_Start\_IT**

**函数作用 使对应的定时器开始工作，并使能其定时中断**

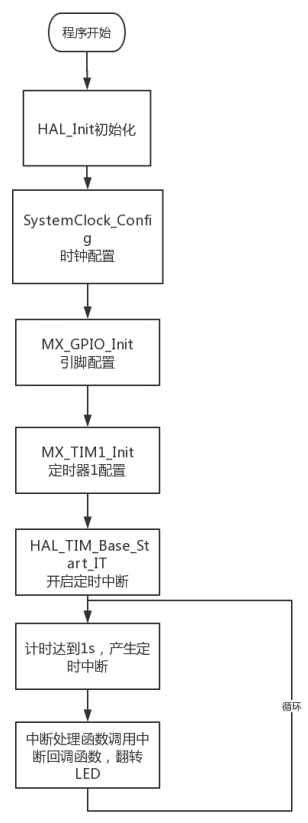
**返回值 HAL\_StatusTypeDef，HAL 库定义的几种状态，**

**如果成功使定时器开始工作，则返回 HAL\_OK**

**参数 \*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**

1. 程序流程：

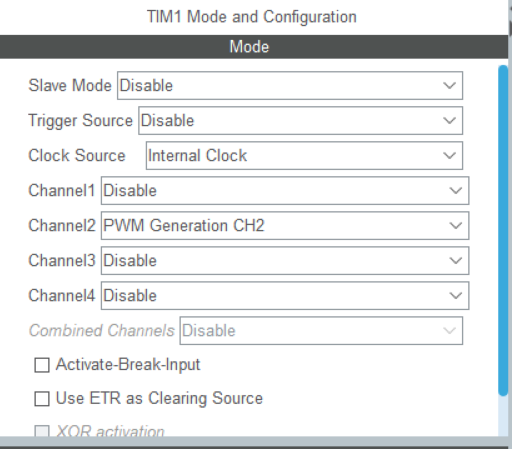


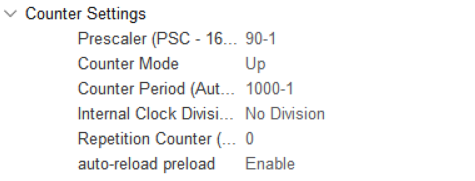
# **PWM控制LED亮度**

1. PWM 在 cubeMX 中配置

在 cubeMX 中设置定时器 1 的通道2为 PWM 输出。可以注意到三个通道对应的引脚正是LED\_R 引脚；

定时器 1 如下配置，设置重载值为 1000-1。



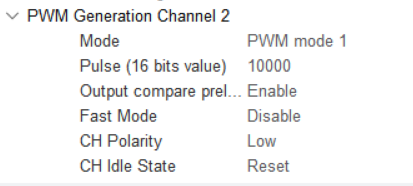


Prescaler (PSC - 16 bits value) 分频系数 90-1

Counter Period (AutoReload Register - 16 bits value ) 重装载值 1000-1

Internal Clock Division No Division 系统不分频

Auto-reload preload Enable 自动重装载 使能



Mode：PWM模式设置，我们选择PWM1模式

Pulse：占空比设置

Output compare preload：通道输出，使能

Fast Mode：快速模式， 不使能

CH Polarity：输出极性 ，低电平有效

1. **HAL\_TIM\_PWM\_Start 函数**

**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**

1. 程序流程：

呼吸灯程序关键：

int i;

HAL\_TIM\_PWM\_Start(&htim3,TIM\_CHANNEL\_2); //打开PWM通道

for(i=0;i<1000;i+=2)

{

TIM3->CCR2 = i; //改变占空比，逐渐增加低电平的时间

HAL\_Delay(1);

}

for(i=1000;i>0;i-=2)

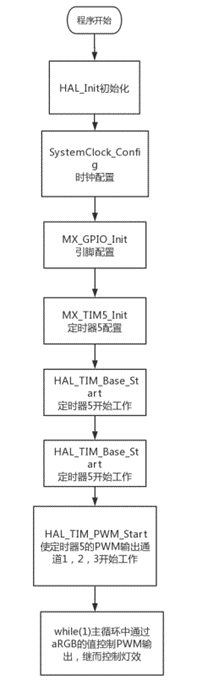
{

TIM3->CCR2 = i; //改变占空比，逐渐降低低电平的时间

HAL\_Delay(1);

}

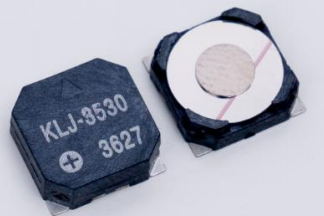
1. 呼吸灯程序流程：



# PWM蜂鸣器

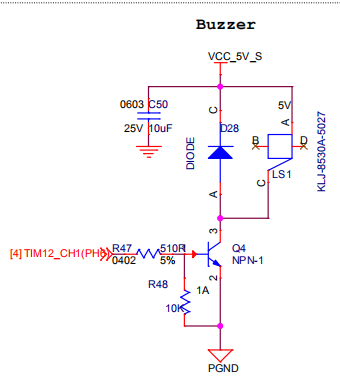
1. 蜂鸣器是一种能够通过电子信号控制的发声器件。





1. 蜂鸣器的 PWM 在 cubeMX 中配置。

蜂鸣器使用的引脚为 PH6，为定时器 12 的通道 1。



打开 cubeMX，使能定时器 12，预分配设置为 0，重载值设置为 20999，设置通道 1 为 PWM输出，其余设置保持默认即可，此时开发板的 PH6 引脚变为绿色。

定时器 12 挂载在 APB1 总线上，对应的总线频率为 90MHz，分频值为 0，重载值为 22500-1，并通过公式计算得到 PWM 波的输出频率为 4000Hz。

1. 蜂鸣器程序说明。

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 12 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。

1. 主函数关键代码：

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);

}

关键函数：

void buzzer\_on(uint16\_t psc, uint16\_t pwm)

{

\_\_HAL\_TIM\_PRESCALER(&htim12, psc);

\_\_HAL\_TIM\_SetCompare(&htim12, TIM\_CHANNEL\_1, pwm);

}

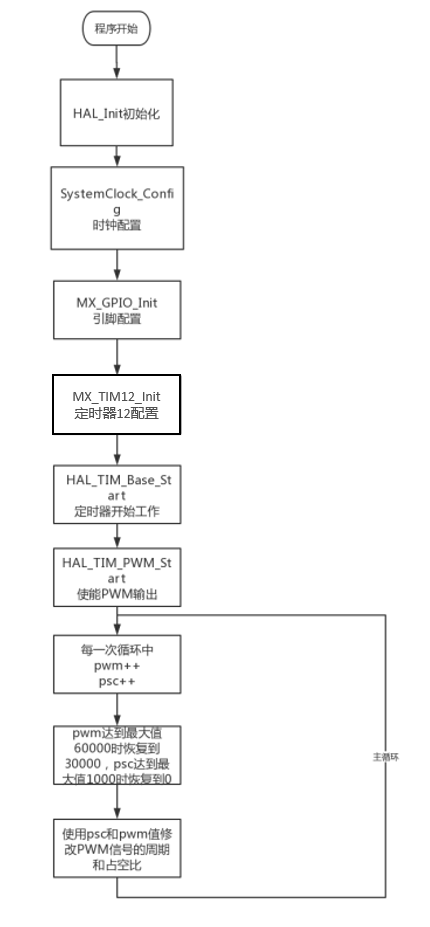
void buzzer\_off(void)

{

\_\_HAL\_TIM\_SetCompare(&htim12, TIM\_CHANNEL\_1, 0);

}

1. 程序流程：



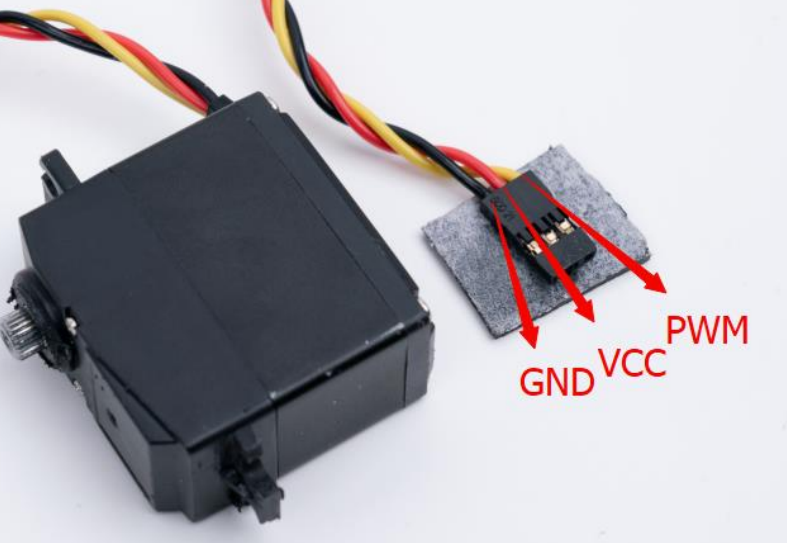
# PWM舵机

1. 舵机介绍。

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。

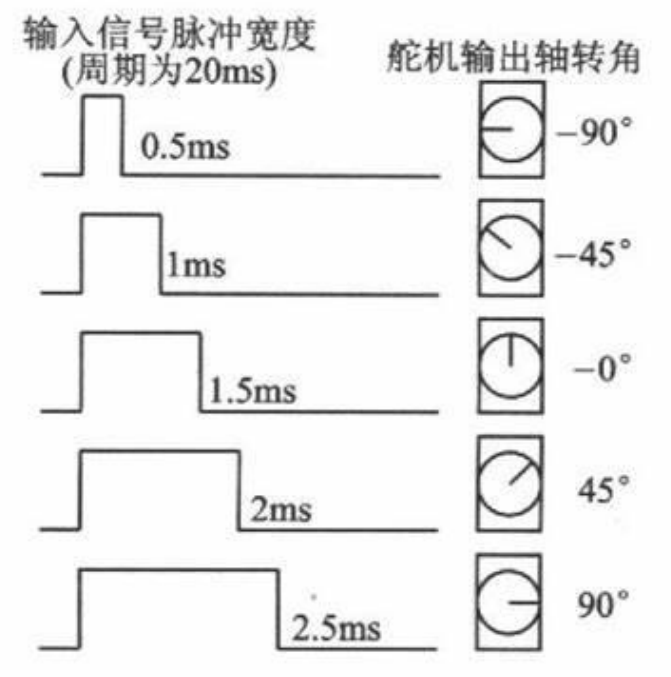


通常舵机的三根线按照颜色分别为：黑色-GND，红色-VCC，黄色-PWM 信号。在使用舵机 时，只需要使用杜邦线或者其他连接线接入对应的 PWM 接口。

1. 舵机控制。

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。



1. 舵机的 PWM 输出接口选择。

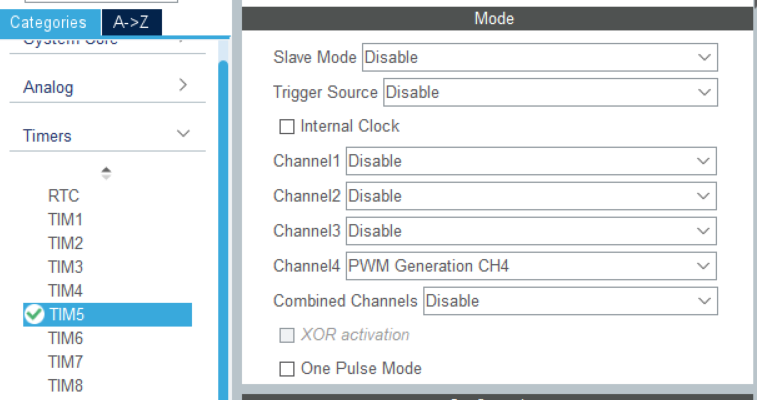
在开发板上具有 16路 PWM 的输出接口，原理图如图所示：

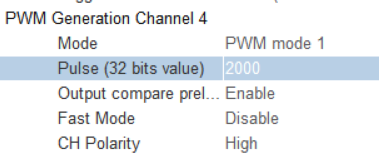
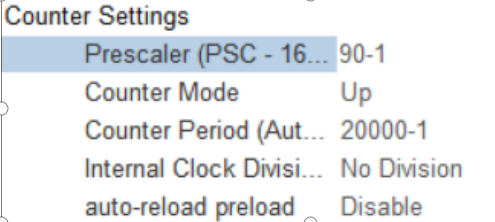


我们选PI0（定时器5通道4）。

1. 舵机的 PWM 在 cubeMX 中配置。

首先开启定时器 5，预分频值设置为 90-1，重载值设置为 20000-1。打开 通道4 的 PWM 输出，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。





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 5 挂载在 APB1 总线上，对 应的总线频率为 90MHz，定时器分频值为 90-1，重载值 20000-1，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。

1. 舵机主程序讲解。

初始化：HAL\_TIM\_Base\_Start 函数启动定时器 5

HAL\_TIM\_PWM\_Start 函数将定时器 5 的 4 号通道的 PWM 输出开启。

主循环：通过\_\_HAL\_TIM\_SetCompare 来设置 PWM 的占空比

int cnt=500;

for(int i=1;i<=5;i++)

{

\_\_HAL\_TIM\_SetCompare(&htim5,TIM\_CHANNEL\_4,cnt\*i);

HAL\_Delay(2000);

}

由于实物舵机由90度变到-90度消耗时间较长故延时2秒。

#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\_\_)))

**\_\_HAL\_TIM\_SET\_COMPARE 宏**

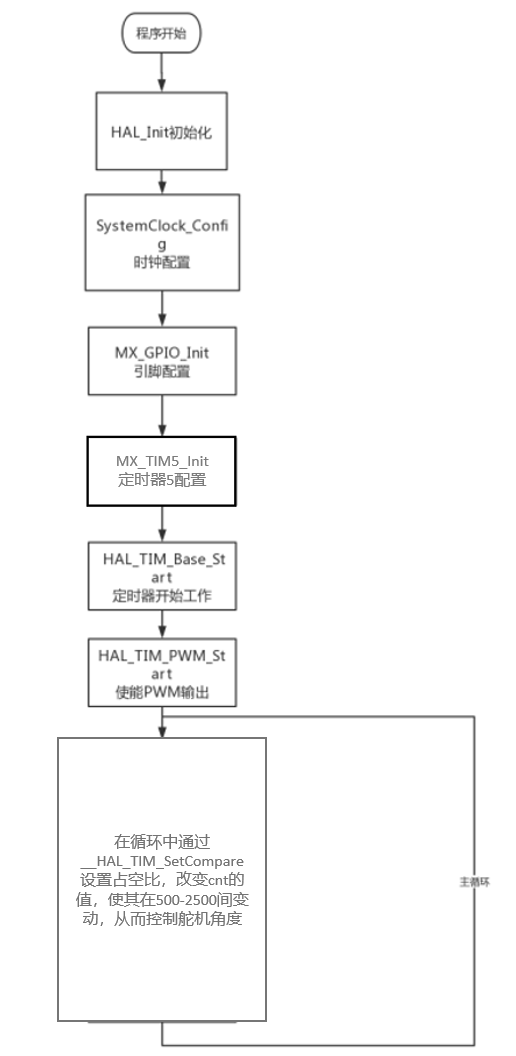
**\_\_HAL\_TIM\_SET\_COMPARE(\_\_HANDLE\_\_, \_\_CHANNEL\_\_, \_\_COMPARE\_\_)**

**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\_CHANNEL\_1**

**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**

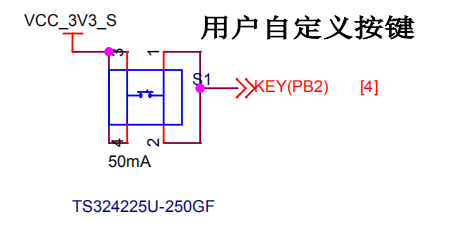
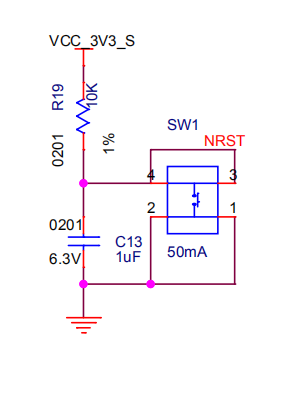
1. 程序流程：



# 按键的外部中断

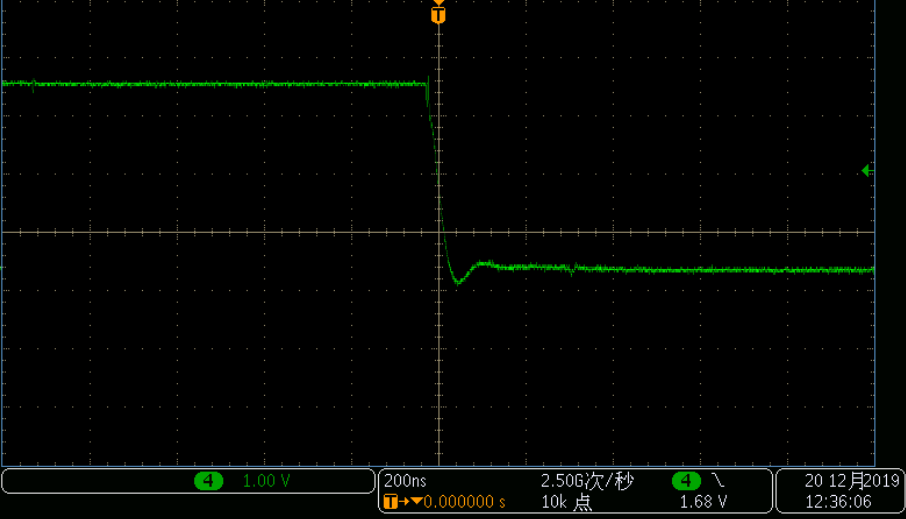
1. 按键原理图介绍

开发板 A 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(PB2)，如图所示。



1. 按键软件消抖

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。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。



1. 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

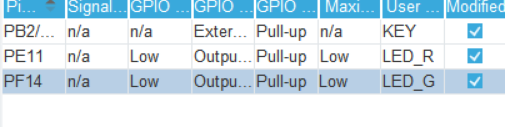
平时，都能引发外部中断。

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

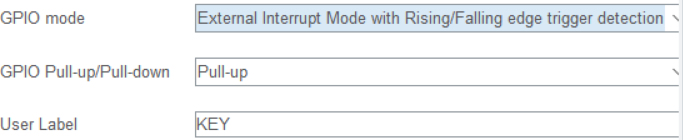
给 STM32，使程序进入外部中断服务函数。

将 PB2 号引脚设置为按键的输入引脚，将其设置为外部中断模式。

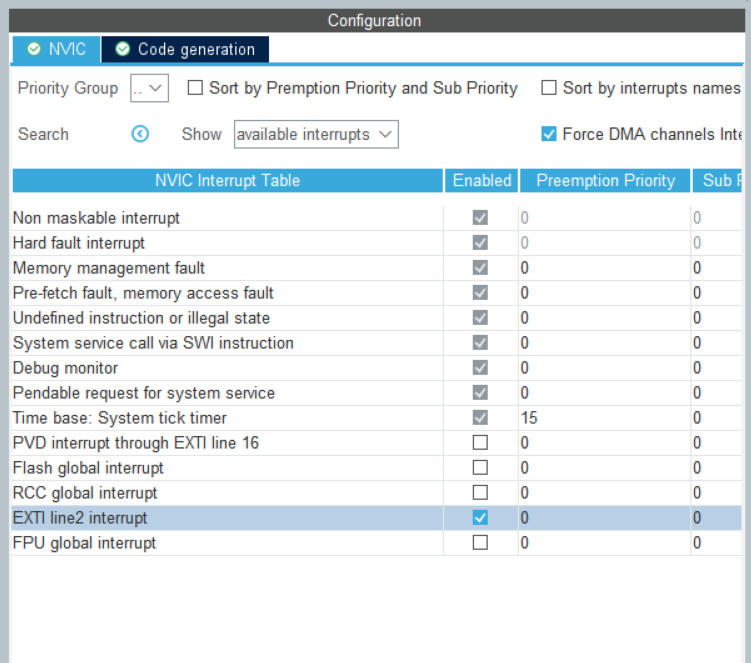


接着点开 GPIO 标签页，对引脚进行如下设置，将 GPIO 模式设置为升降沿触发的外部中

断，上下拉电阻设置为上拉电阻，最后设置用户标签为 KEY。



在 NVIC 标签页下，将外部中断开启。



1. **HAL\_GPIO\_ReadPin 函数介绍**

**GPIO\_PinState HAL\_GPIO\_ReadPin(GPIO\_TypeDef\* GPIOx, uint16\_t GPIO\_Pin)**

**函数名 HAL\_GPIO\_ReadPin**

**函数作用 返回引脚电平**

**返回值 GPIO\_PinState，如果是高电平则返回 GPIO\_PIN\_SET（对应**

**为 1），如果是低电平则返回 GPIO\_PIN\_RESET（对应为 0）**

**参数 1：GPIOx 对应 GPIO 总线，其中 x 可以是 A…I。**

**例如 PH10，则输入 GPIOH**

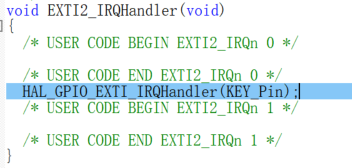
**参数 2：GPIO\_Pin 对应引脚数。可以是 0-15。**

**例如 PH10，则输入 GPIO\_PIN\_10**

1. 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 EXTI2\_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。



1. 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。

1. 主要程序：

前台程序：记录按键翻转的状态 rising\_falling\_flag

后台程序：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

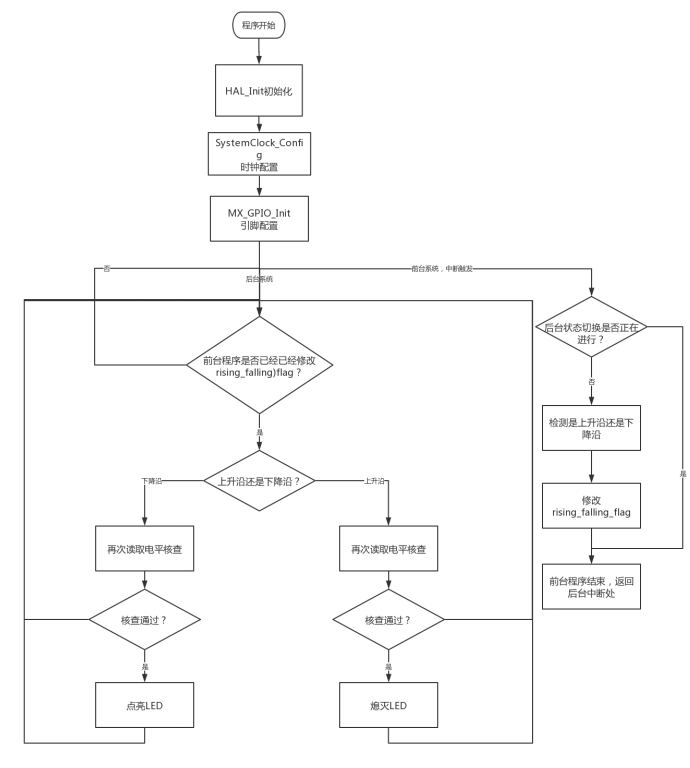
触发中断的是上升沿还是下降沿。

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。

1. 程序流程：



# 串口收发

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

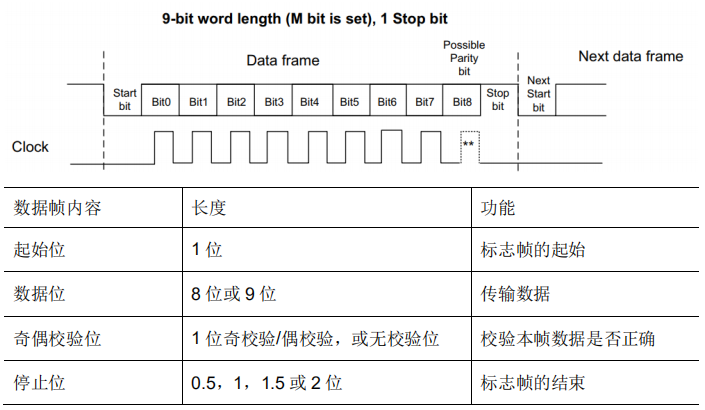
的起始，接着跟随 8 位或者 9 位数据位，之后为校验位，分为奇校验，偶校验和无校验，最后以一个先高后低的脉冲表示结束位，长度可以设置为 0.5，1，1.5 或 2 位长度。

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。

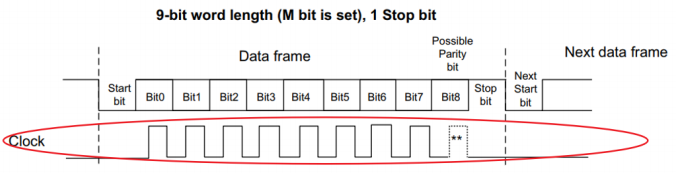
下图是一个 8 位数据位，1 位奇偶校验位，1 位结束位的串口数据帧。



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

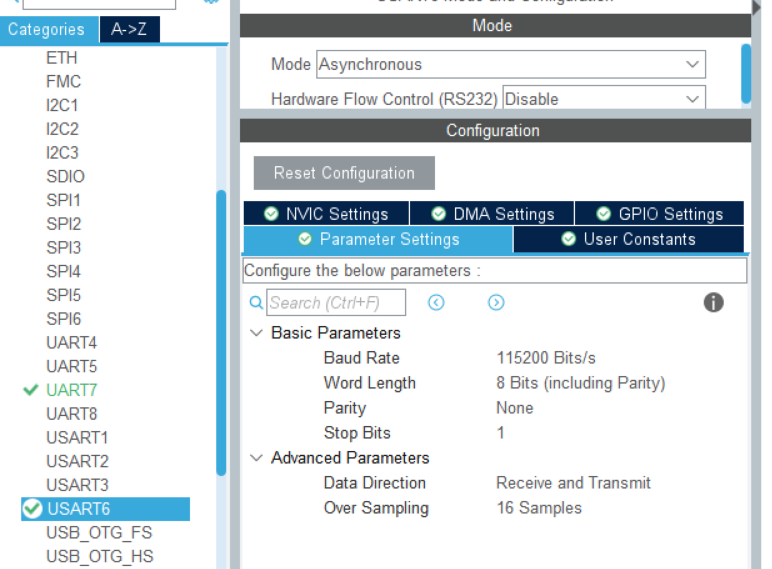
发送数据的速率,单位为波特每秒，一般串口常用的波特率有 115200，38400，9600 等。串口的波特率和总线时钟周期（clock）成倒数关系，即总线时钟周期越短，单位时间内发送的码元数量越多，串口波特率就越高。



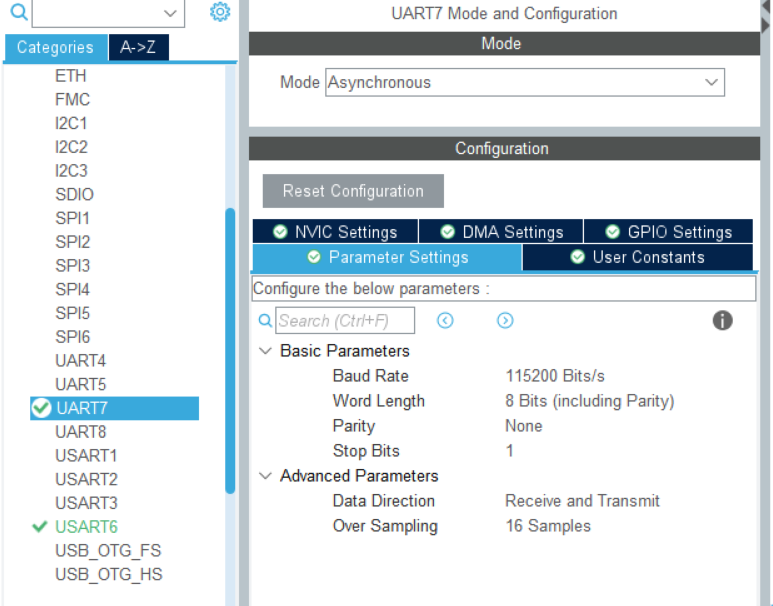
1. 串口在 cubeMX 中配置

1). 首先在 Connectivity 标签页下将 USART 6 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。

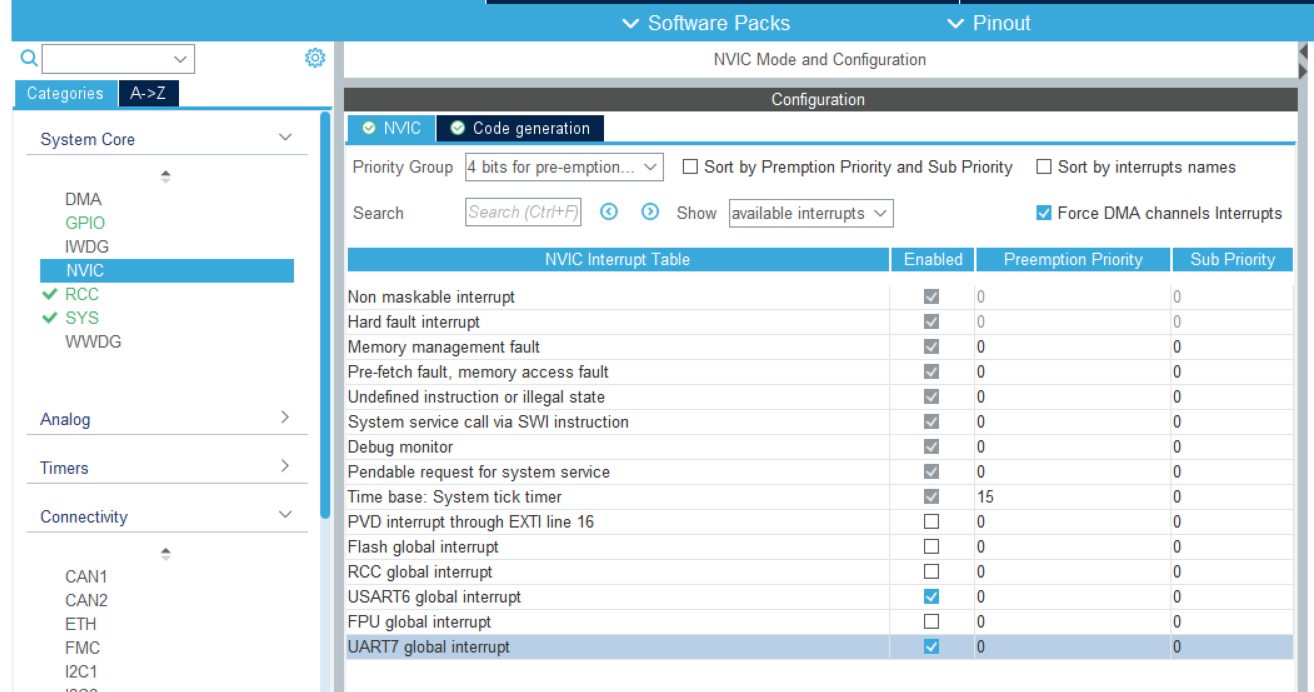
2). 接着将其波特率设置为 115200，数据帧设置为 8 位数据位，无校验位，1 位停止位。



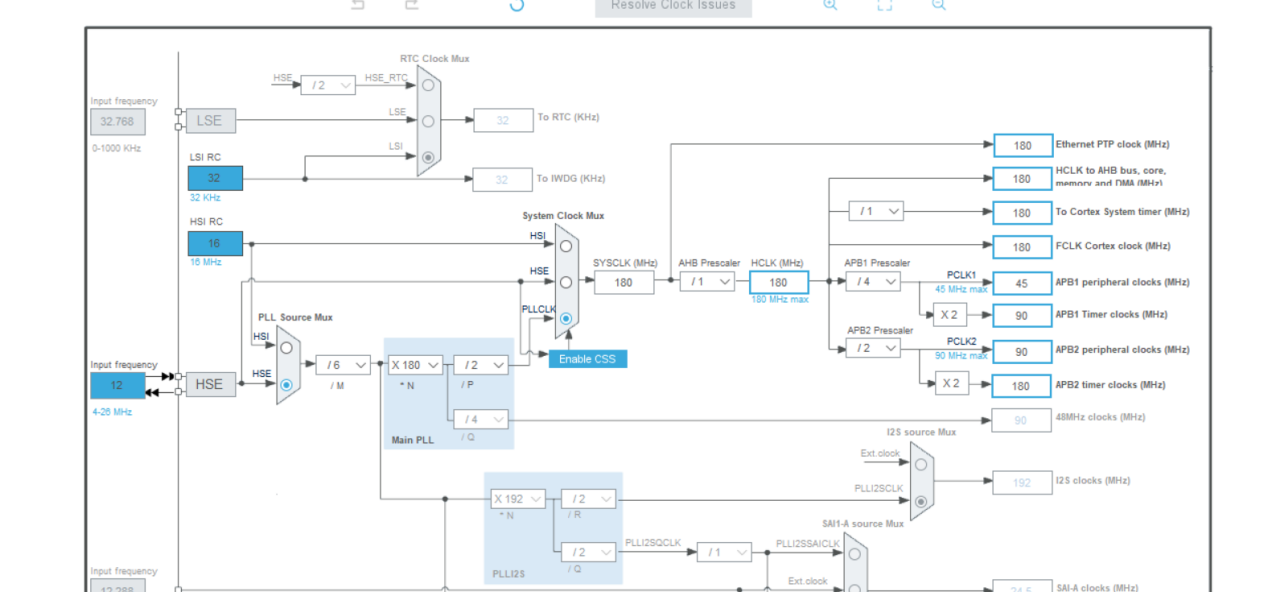
3).同样地，打开 UART7，将其以和 USART6 同样的方式进行设置。



4).接着前往 NVIC 标签页下，开启 USART6 和 UART7 的中断。



工程时钟配置如图：



1. 串口接收中断与空闲中断

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。

1. 串口发送函数与中断函数

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[]=”Helloword”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。

1. 主要程序

在中断处理函数USARTx\_IRQHandler编写代码。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 7 和串口 6 的接收中断和空闲中断，在主循环中通过

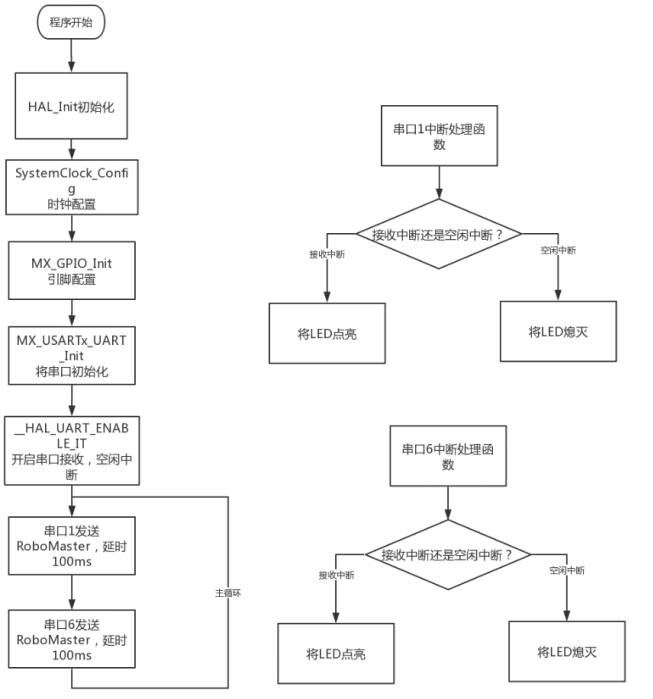
HAL\_UART\_Transmit函数完成串口 7 和串口的 6 的数据发送，发送内容为“RoboMaster\r\n”这一字符串。

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。

1. 程序流程：



# 串口打印遥控器数据

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。

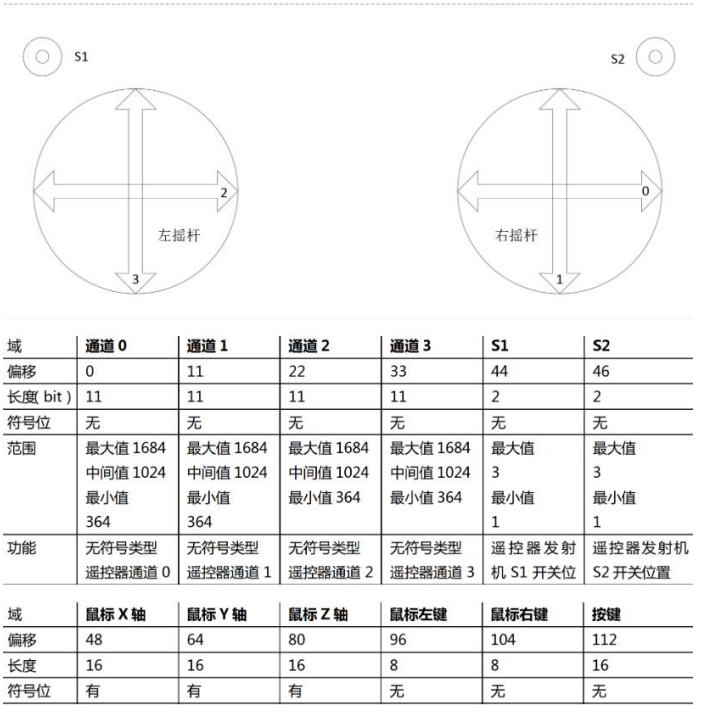
STM32 中的许多通讯如 USART，SPI，IIC 都支持 DMA 方式进行数据的收发。

1. 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。



1. 串口发送的 DMA 配置

首先开启 USART1 进行配置，其中 USART1 开启串口的 DMA 接收，用于遥控器数据的接收；配置如下：

1)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 100000，数据帧设置为 9 位数据位，无校验位，1 位停止位。(8位可能会出错）

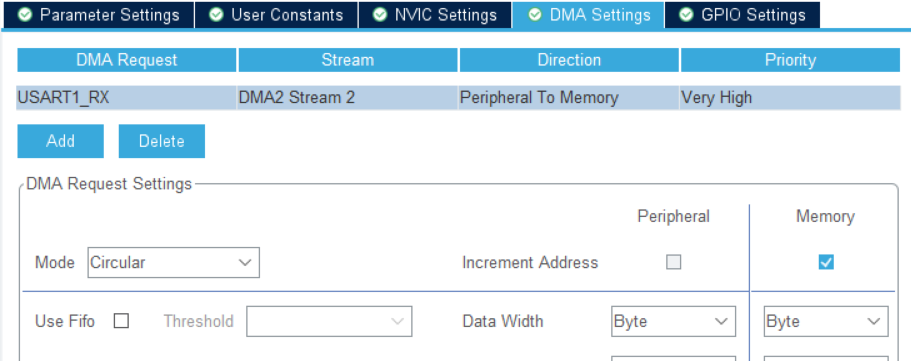


3)接着分别开启 USART1 的 DMA 功能。点开 USART1 的设置页面，打开

DMA Settings 的标签页，点击 Add。

4)在弹出的新条目中，将 DMA Request 选为 USART1\_RX，数据从存储器流向外设，

Priority 选为 Very High。



如果Mode是Circular，则在回调函数（和串口中断的回调函数是同一个）中不需要再次开启HAL\_UART\_Receive\_DMA(&huart1,sbus\_rx\_buffer,18);

1. 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);

}

1. 串口的 DMA 接收与发送配置

使用 USART1 的 DMA 接收功能来接收遥控器数据。

DMA的默认中断使能已经开启，实测可以不用再次开启串口中断，如果不用DMA，使用串口接受中断，那么需要在NVIC（中断向量控制器中开启对应的中断）。

**HAL\_UART\_Receive\_DMA(&huart1,sbus\_rx\_buffer,18)**;在初始化的时候使用串口DMA接受中断函数带入缓存的地址和接受数据的长度。

有当串口接收到了18个字符，才会进入这个中断函数

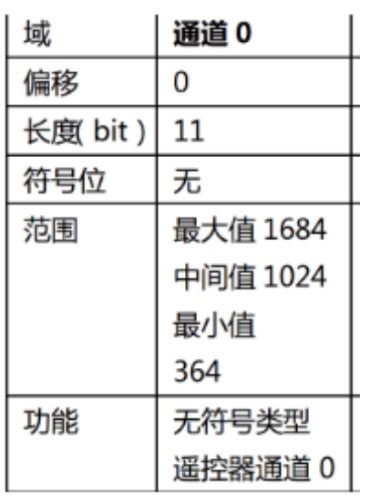
**void HAL\_UART\_RxCpltCallback(UART\_HandleTypeDef \*UartHandle)**

**（数据接收错误时，假如只收到17个字符，那下一段18个字符的第1个字符将接到17个字符后面，从而导致后续数据接收出错）。**

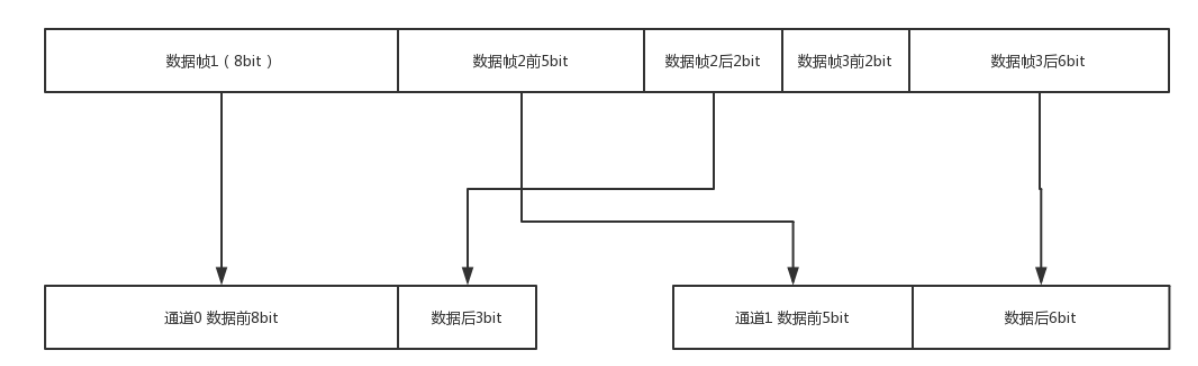
1. DBUS数据解析

遥控器数据处理函数DBUS数据解析在中断函数中进行。

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。



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。

void HAL\_UART\_RxCpltCallback(UART\_HandleTypeDef \*UartHandle)

{

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;

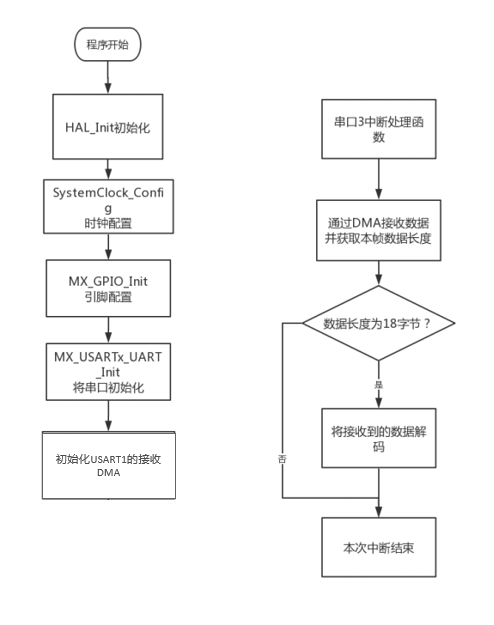
rc\_ctrl->rc.ch[2] -= RC\_CH\_VALUE\_OFFSET;

rc\_ctrl->rc.ch[3] -= RC\_CH\_VALUE\_OFFSET;

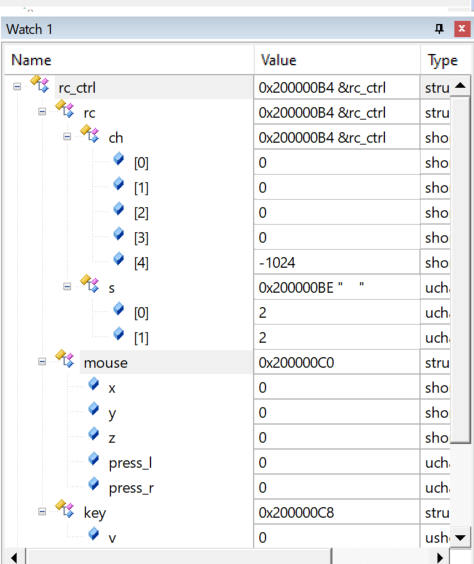
rc\_ctrl->rc.ch[4] -= RC\_CH\_VALUE\_OFFSET;

}

1. 程序流程：



开启DBUG查看数据接收情况：



# I2C和SPI读取传感器数据

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 处于高电平时候，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：



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。

1. 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

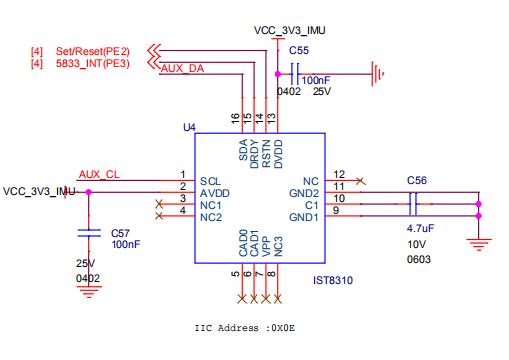
地磁场强度，用于计算磁场角度。

**IST8310 的 GPIO 管教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）** |

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

管脚如下图所示。



|  |  |
| --- | --- |
| IST8310 管脚 MCU 管脚 | 管脚 MCU 管脚 |
| SCL | AUX\_CL(MPU6600的引脚） |
| SDA | AUX\_DA(MPU6600的引脚） |
| RSTN | PE2 |
| DRDY | PE3 |

查询得知，A板IST8310的RSTN和DRDY与MPU6600传感器外部引脚相连，共同组成9轴传感器。

1. SPI 协议简介

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，这些设备被区分成

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。
5. 陀螺仪简介

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。

1. 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。

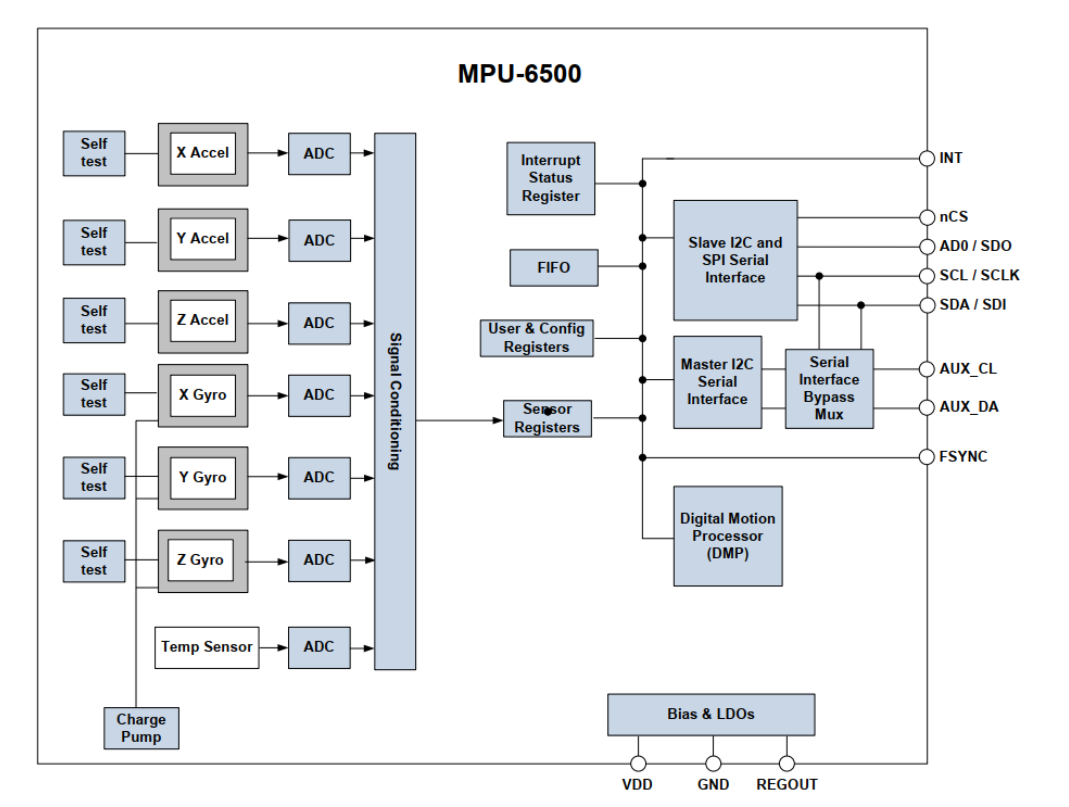
当物体静止时，加速度计测量重力加速度在三个正交方向上的分量，配合陀螺仪的角速度信

息可以解算出物体的空间位姿。

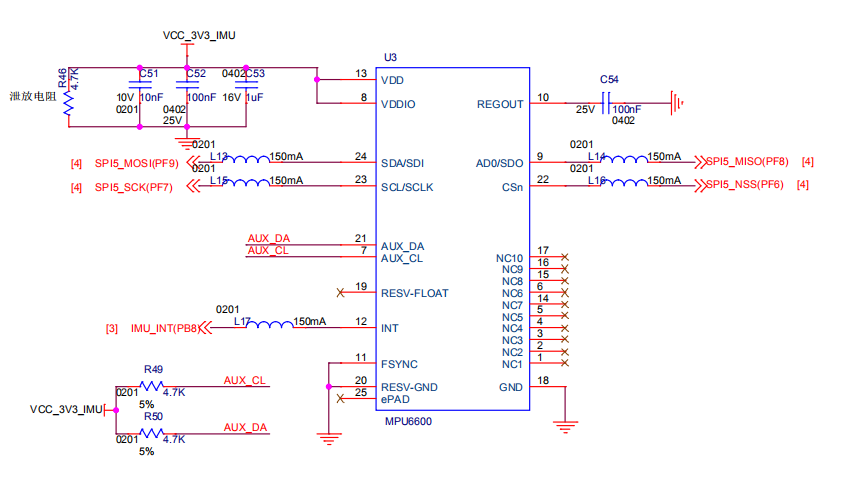
1. MPU6600

六轴加速度陀螺仪MPU6600在开发板上集成，PCB 上已经接线完毕，故而不需要外部接线。

内部框图如下。



对应管脚如下图所示。

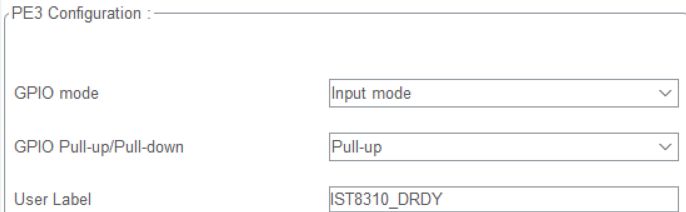
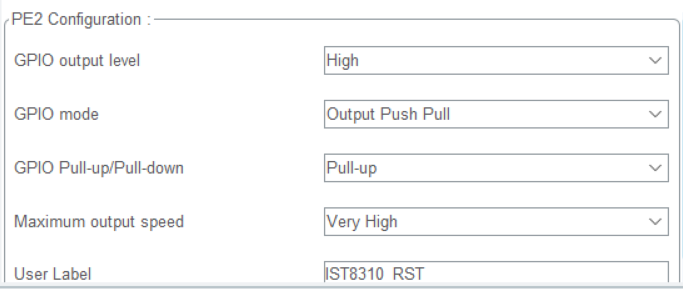


|  |  |
| --- | --- |
| **管脚** | **功能** |
| **SCL/SCLK** | **I2C 的串行时钟/SPI串行时钟** |
| **SDA/SDI** | **I2C 的串行数据/SPI串行数据输入** |
| **AD0/SD0** | **I2C从地址LSB/SPI串行数据输出** |
| **DRDY** | **IST8310 的数据准备（data ready）** |
| **CSn** | **SPI片选线** |
| **AUX\_DA** | **I2C主串行数据，外接磁力计AUX\_DA** |
| **AUX\_CL** | **I2C主串行时钟，外接磁力计AUX\_CL** |
| **INT** | **中断数据输出（推挽或开漏）** |

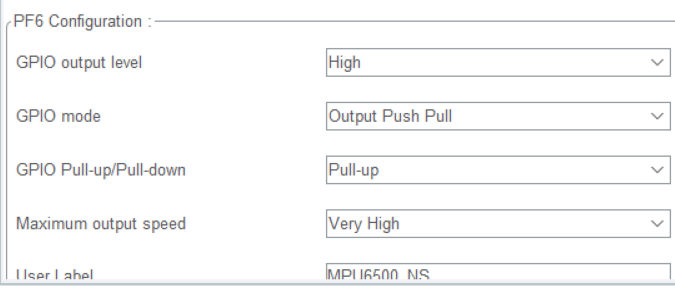
1. cubeMX 中的配置

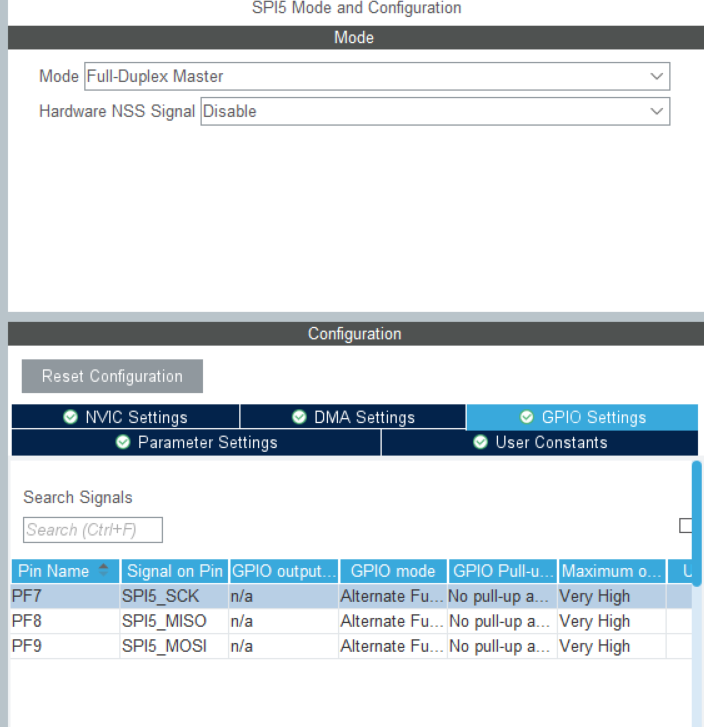
ist8310driver.c ， ist8310driver.h, ist8310driver\_middle.c 以 及ist8310driver\_middle.h 四个文件，其中 ist8310driver.h，ist8310driver\_middle.h 用于申明驱动函数，ist8310driver.c 实现驱动函数，ist8310driver\_middle.c 作为中间层实现 I2C 通 信封装以及延迟函数，方便移植。

配置IST8310的RSTN、DRDY，



配置MPU6600





1. Mpu6500初始化函数

uint8\_t mpu6500\_init(void)

{

uint8\_t res = 0;

uint8\_t wait\_time = 150;

uint8\_t sleepTime = 50;

uint8\_t writeNum = 0;

//mpu6500 gpio init

mpu6500\_GPIO\_init();//初始化

// mpu6500 com init

mpu6500\_com\_init();//初始化

//check commiunication is normal检查通讯情况

mpu6500\_read\_single\_reg(MPU\_WHO\_AM\_I);

mpu6500\_delay\_us(wait\_time);

mpu6500\_read\_single\_reg(MPU\_WHO\_AM\_I);

mpu6500\_delay\_us(wait\_time);

mpu6500\_write\_single\_reg(MPU\_PWR\_MGMT\_1, MPU\_DEVICE\_RESET);

mpu6500\_delay\_ms(sleepTime);

//check commiunication is normal after reset

mpu6500\_read\_single\_reg(MPU\_WHO\_AM\_I);

mpu6500\_delay\_us(wait\_time);

mpu6500\_read\_single\_reg(MPU\_WHO\_AM\_I);

mpu6500\_delay\_us(wait\_time);

//read the register "WHO AM I"

res = mpu6500\_read\_single\_reg(MPU\_WHO\_AM\_I);

mpu6500\_delay\_us(wait\_time);

if (res != DEVICE\_ID)

{

return NO\_Sensor;

}

//set mpu6500 sonsor config and check

for (writeNum = 0; writeNum < MPU6500\_Write\_Reg\_Num; writeNum++)

{

mpu6500\_write\_single\_reg(write\_mpu6500\_reg\_data\_error[writeNum][0], write\_mpu6500\_reg\_data\_error[writeNum][1]);

mpu6500\_delay\_us(wait\_time);

res = mpu6500\_read\_single\_reg(write\_mpu6500\_reg\_data\_error[writeNum][0]);

mpu6500\_delay\_us(wait\_time);

if (res != write\_mpu6500\_reg\_data\_error[writeNum][1])

{

return write\_mpu6500\_reg\_data\_error[writeNum][2];

}

}

// NO error

return MPU6500\_NO\_ERROR;

}

1. ist8310初始化函数，由于A板ist8310接在mpu6600上，其初始化应后于mpu6600。

uint8\_t ist8310\_init(void)

{

static const uint8\_t wait\_time = 150;

static const uint8\_t sleepTime = 50;

uint8\_t res = 0;

uint8\_t writeNum = 0;

ist8310\_GPIO\_init();//初始化

ist8310\_com\_init();

ist8310\_RST\_L();

ist8310\_delay\_ms(sleepTime);

ist8310\_delay\_ms(sleepTime);

ist8310\_RST\_H();

res = ist8310\_IIC\_read\_single\_reg(IST8310\_WHO\_AM\_I);//检查通讯情况

if (res != IST8310\_WHO\_AM\_I\_VALUE)

{

return IST8310\_NO\_SENSOR;

}

//set mpu6500 sonsor config and check

for (writeNum = 0; writeNum < IST8310\_WRITE\_REG\_NUM; writeNum++)

{

ist8310\_IIC\_write\_single\_reg(ist8310\_write\_reg\_data\_error[writeNum][0], ist8310\_write\_reg\_data\_error[writeNum][1]);

ist8310\_delay\_us(wait\_time);

res = ist8310\_IIC\_read\_single\_reg(ist8310\_write\_reg\_data\_error[writeNum][0]);

ist8310\_delay\_us(wait\_time);

if (res != ist8310\_write\_reg\_data\_error[writeNum][1])

{

return ist8310\_write\_reg\_data\_error[writeNum][2];

}

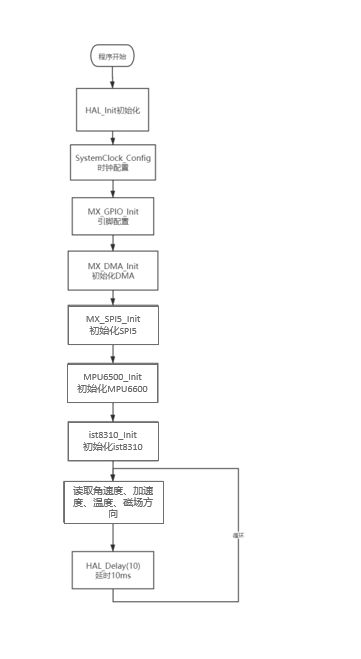
}

ist8310\_auto\_com\_by\_mpu6500();

return IST8310\_NO\_ERROR;

}

1. 数据读取函数（交给有缘人补充）
2. 程序流程



IMU温度控制(PID)

1. IMU温控意义

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

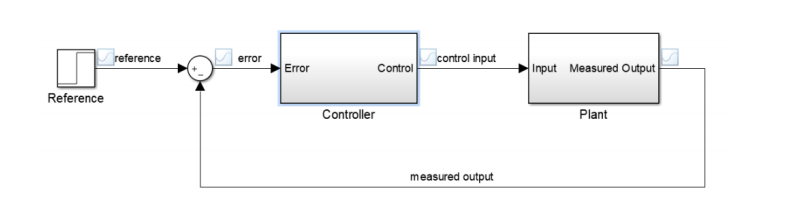
此需要在 IMU 上电时首先测量出零漂的大小，将 IMU 测量到的值与零漂值相减，从而减少零漂的影响。这种方法的前提是 IMU 的零漂值保持固定，否则依然会引入误差。

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。

1. 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控制器中各项特点：

|  |  |  |
| --- | --- | --- |
|  | **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。** |

PID 初始化函数

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;

}

PID计算函数

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;

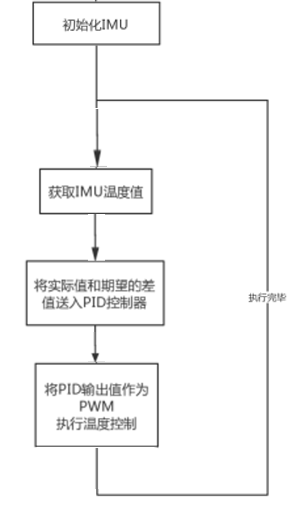
LimitMax(pid->out, pid->max\_out);

}

return pid->out;

}

1. 程序流程：



# IMU姿态解算

1. mahony 算法

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/>

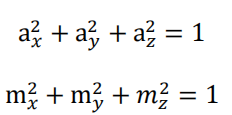
|  |  |  |
| --- | --- | --- |
| 参数 | 意义 | 设置值 |
| sampleFreq | 采样频率 | 1000.0f （原始值 512.0f） |
| twoKpDef | 加速度计以及磁力计融合权重 Kp，值越大，收敛速度越快。 | 2.0f\*0.5f |
| twoKpDef | 加速度计以及磁力计融合权重 Ki，值越大，收敛速度越快。 | 2.0f\*0.0f |
| twoKp | twoKpDef 宏赋值变量 | twoKpDef |
| TwoKi | twoKiDef 宏赋值变量 | twoKiDef |
| q0,q1,q2,q3 | 四元数 | [1.0,0,0,0] |
| integralFBx  integralFBy  integralFBz | 误差积分值 | 0.0f |

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** | **None** |
| **参数 1：q[4]** | **四元数数组** |
| **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** |
| **参数 8,9,10：mx,my,mz** | **磁力计磁场数据** |

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即：



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更新四元数

1. 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** |

1. invSqrt 函数

**float invSqrt(float x)**

|  |  |
| --- | --- |
| **函数名** | **invSqrt** |
| **功能** | **计算平方根的倒数，即求** |
| **函数返回** | **平方根的倒数** |
| **参数 1:x** | **待计算的浮点数** |

1. AHRS\_init 函数

**void AHRS\_init(fp32 quat[4], fp32 accel[3], fp32 mag[3])**

|  |  |
| --- | --- |
| **函数名 AHRS\_init** | **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。

1. 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。

1. 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]));**

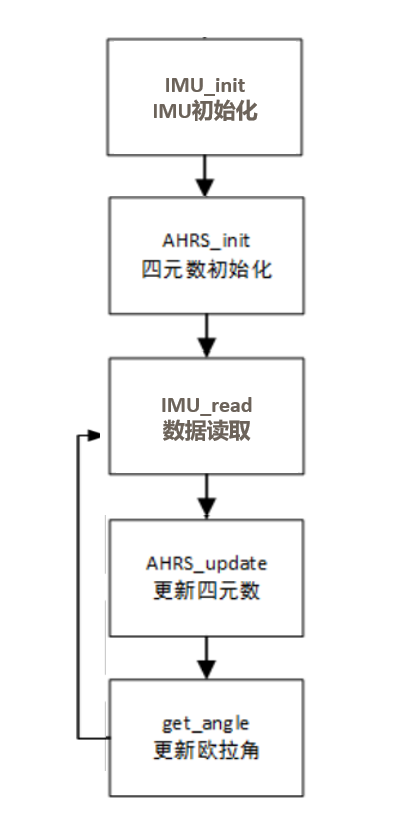
**\*roll = atan2f(2.0f\*(q[0]\*q[1]+q[2]\*q[3]),2.0f\*(q[0]\*q[0]+q[3]\*q[3])-1.0f);**

**}**

|  |  |
| --- | --- |
| **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 3：roll** | **Roll 角的指针** |



1. 程序流程：



# CAN控制电机

1. CAN 协议

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

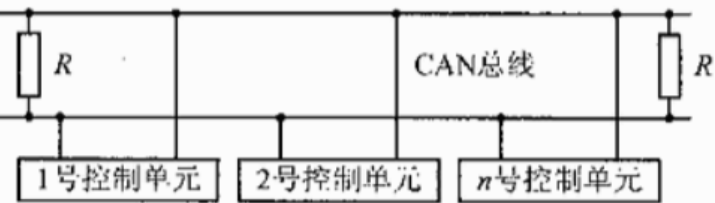
品著称的德国 BOACH 公司开发，并最终成为国际标准（ISO11898），CAN 是国际上应用

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

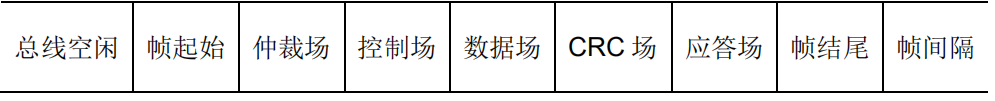
入式工业控制局域网的标准总线，并且拥有以 CAN 为底层协议专为大型货车和重工机械车

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。



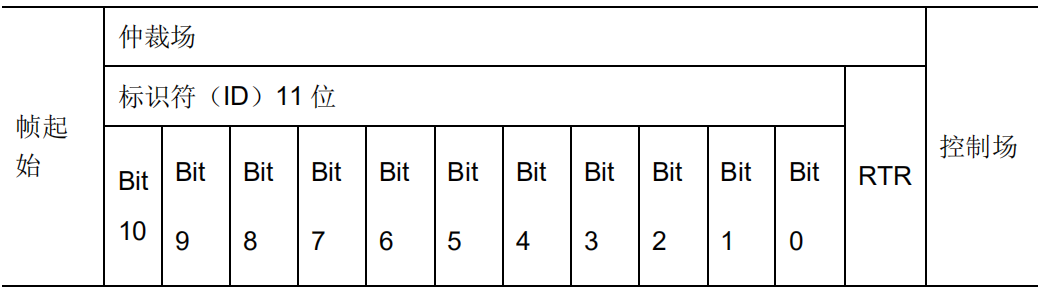
RoboMaster 系列电机也采用 CAN 协议进行通信，CAN 协议比较复杂，一个完整的数据帧 由下图中的各个部分组成：



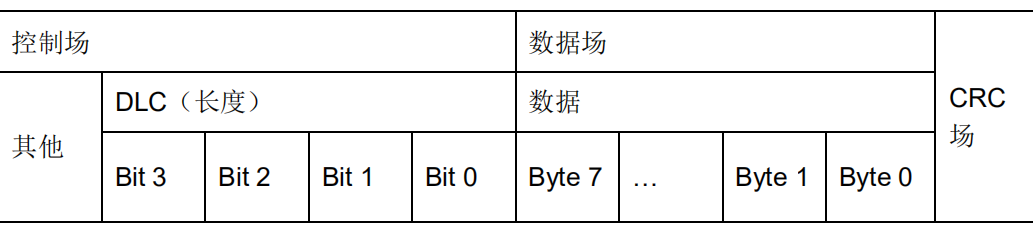
重点介绍 CAN 的仲裁场和数据场的内容。和 I2C 总线一样，每一个挂载在 CAN 总线

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。



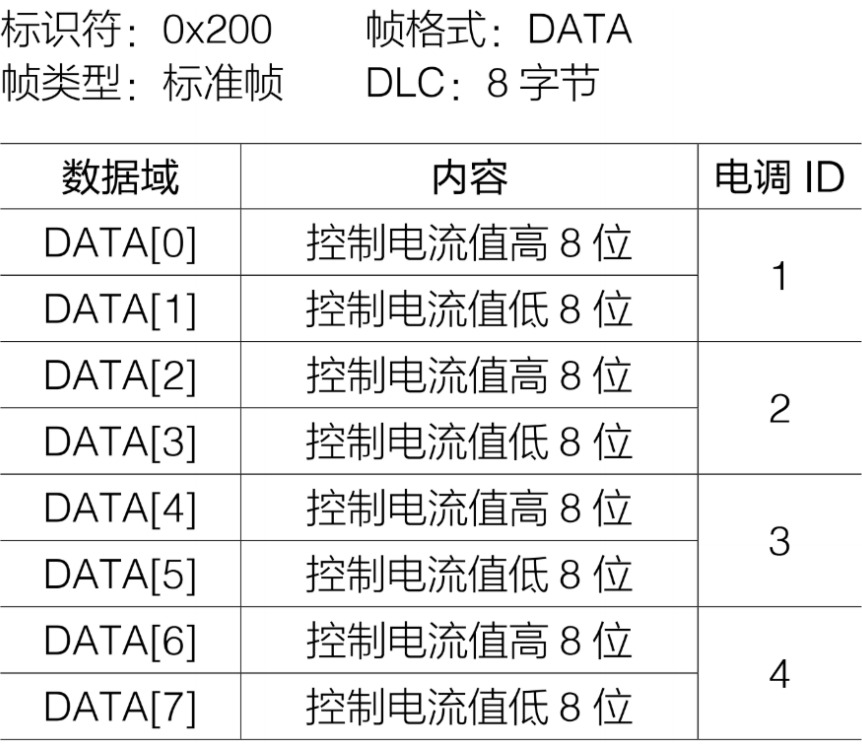
在通过 ID 判断本帧数据可以接收后，控制场中的 DLC 规定了本帧数据的长度，而数据场内的数据的大小为 8 Byte，即 8 个 8 位数据。CAN 总线的一个数据帧中所需要传输的有效数据实际上就是这 8 Byte。



1. 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：



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。

当要接收电调发送来的数据时，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。



1. CAN 在 cubeMX 中的配置

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

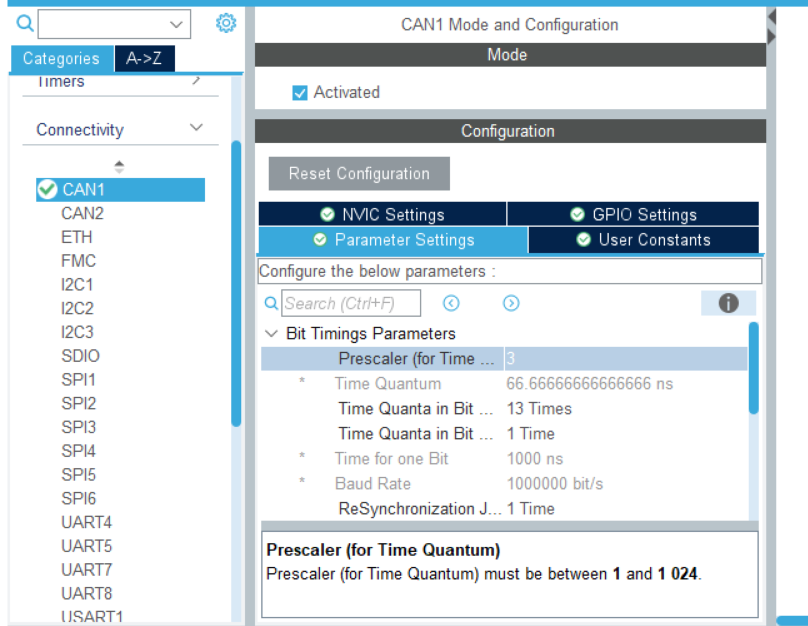
2在 Mode 中，将 Activated 选中打勾。

3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。





1. CAN 发送函数介绍

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

**HAL\_StatusTypeDef HAL\_CAN\_AddTxMessage(CAN\_HandleTypeDef \*hcan,**

**CAN\_TxHeaderTypeDef \*pHeader, uint8\_t aData[], uint32\_t \*pTxMailbox)**

|  |  |
| --- | --- |
| **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组名称** |
| **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);**

**}**

1. CAN 接收中断回调介绍

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** | **HAL\_CAN\_GetRxMessage** |
| **函数功能** | **接收 CAN 总线上发送来的数据** |
| **返回值** | **HAL\_StatusTypeDef，HAL 库定义的几种状态，如果本次 CAN 接收成功，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** | **uint32\_t RxFifo,接收时使用的 CAN 接收 FIFO 号，一般为 CAN\_RX\_FIFO0** |
| **参数 3** | **CAN\_RxHeaderTypeDef \*pHeader，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** | **uint8\_t aData[]，存储接收到的数据的数组名称** |

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) \{ \**

**(ptr)->last\_ecd = (ptr)->ecd; \**

**(ptr)->ecd = (uint16\_t)((data)[0] << 8 | (data)[1]); \**

**(ptr)->speed\_rpm = (uint16\_t)((data)[2] << 8 | (data)[3]); \**

**(ptr)->given\_current = (uint16\_t)((data)[4] << 8 | (data)[5]); \**

**(ptr)->temperate = (data)[6]; \**

**}**

1. 程序流程

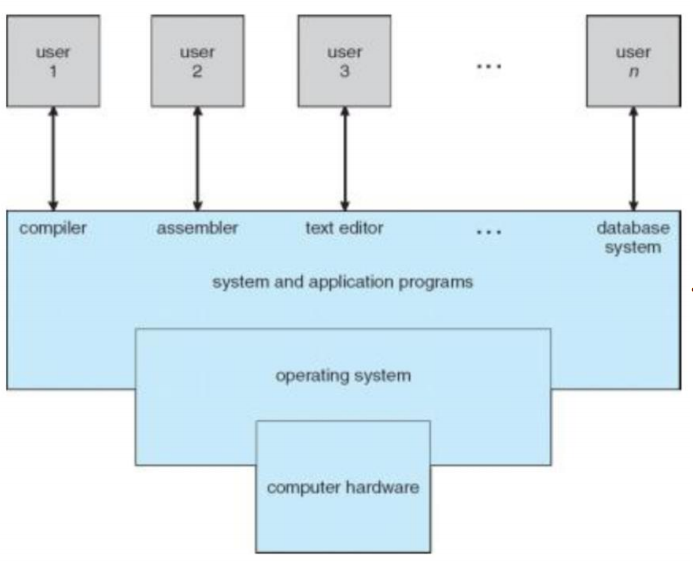


# FreeRTOS

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)。

1. cubeMX 中 freeRTOS 的配置

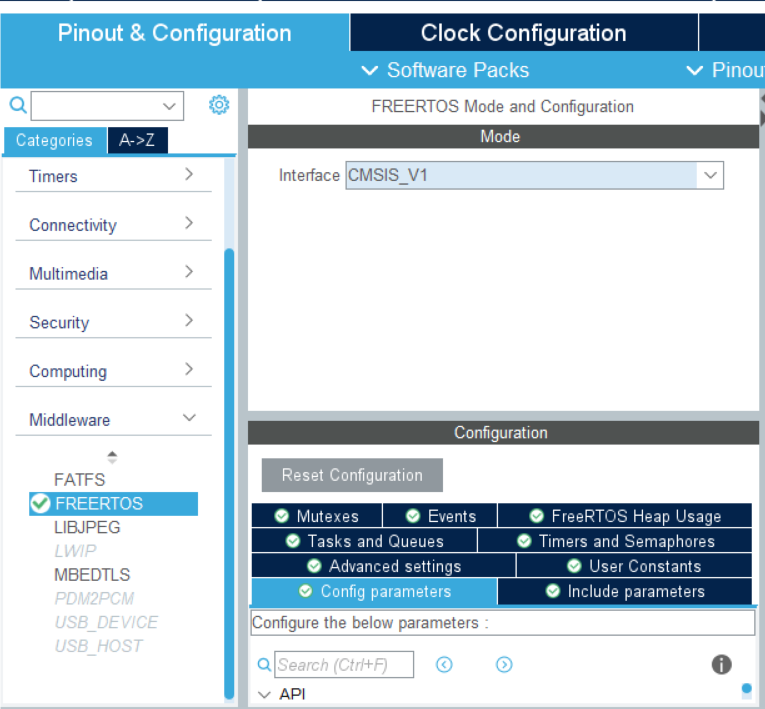
打开 Middleware 标签，选中其中的 FREERTOS 选项，进入 FREERTOS 的配置页面。

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。



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

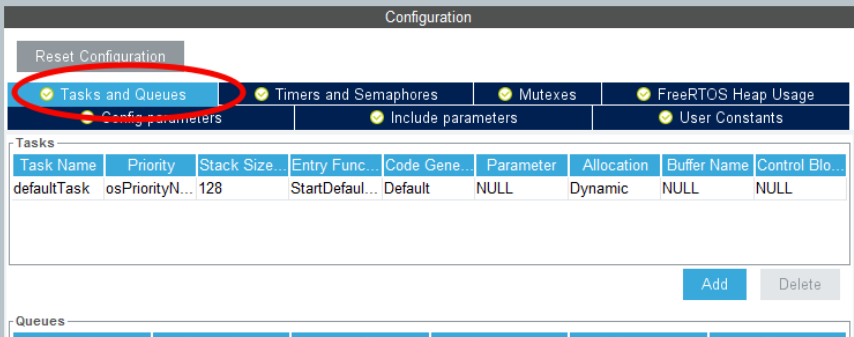
系统时钟速率，最大优先级数量，最小任务栈尺寸，最大任务名称长度等。针对一些比较重

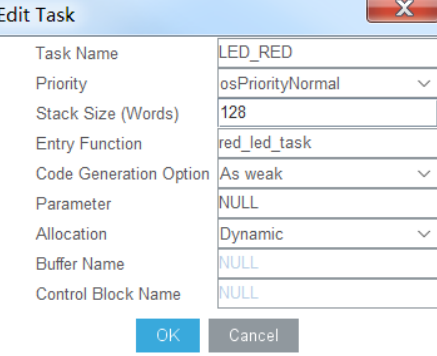
要的配置列表如下：

|  |  |
| --- | --- |
| **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** | **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** |

1. cubeMX 中创建任务

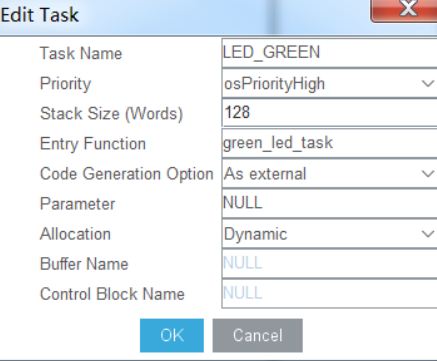
在 freeRTOS 的配置页面中的Configuration 下，选中 Tasks and Queues 标签页，存在一个已经创建的默认任务为 “defaultTask”，点击进入配置选项修改为如下图所示。



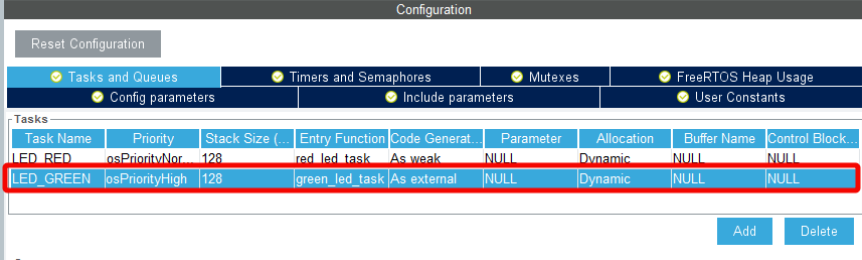


需要增加一个新任务，点击 Add，就可以创建一个新的任务。在弹出的页面中，可以配置任务名称，优先级，栈大小，入口函数等。各个参数的具体功能见下表：

|  |  |
| --- | --- |
| **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** | **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，**  **As weak：产生一个用\_\_weak 修饰符修饰的任务函数；**  **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；**  **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。** |



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。



在 freertos.c 中也可以找到修改的默认任务函数。

选择了通过\_\_weak 修饰符创建一个弱函数，可以再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。

**\_\_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 \*/**

**}**

**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);**

**}**

**}**

1. 程序中创建任务

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”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, ... \*/**

**/\* USER CODE END RTOS\_THREADS \*/**

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** | **osThreadDef** |
| **功能** | **对要创建的任务进行设置** |
| **参数 1** | **name，要创建的任务的名称** |
| **参数 2** | **thread，要创建的任务代码的入口名称** |
| **参数 3** | **priority，要创建的任务的优先级** |
| **参数 4** | **instances，任务下可以创建的线程的数量** |
| **参数 5** | **stacksz，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 | osThreadCreate |
| 函数功能 | 创建一个任务 |
| 返回值 | osThreadId，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 osThreadId 的变量在此处存储返回值。 |
| 参数 1 | const osThreadDef\_t \*thread\_def，我们通过 osThreadDef 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 osThreadDef 中设置任务名为 LED\_RED，则在此处输入 osThread(LED\_RED) |
| 参数 2 | void \*argument，任务需要的初始化参数，一般填为 NULL |

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。

1. 程序流程：

