OpenMV舵机云台使用手册

1.概述

本SDK提供了基于 STM32F103 的舵机云台 控制例程及API，适用于所有总线伺服舵机型号。

API基于 总线伺服舵机通信协议 编写，可点击链接参考。

2.前置参考阅读

- 《总线伺服舵机SDK使用手册（STM32F103）》
- 华馨京官网总线伺服舵机相关文档
- C语言、MicroPython相关书籍
- OpenMV中文入门教程
- STM32F103相关数据手册

建议用户先阅读C语言、MicroPython相关书籍，了解编程的基本语法，以更好理解本手册中提供的SDK；

《总线伺服舵机SDK使用手册（STM32F103）》是对舵机SDK的详细解析，可以帮助用户更好地使用舵机SDK，本舵机云台相关资料中提供的例程也基于该舵机SDK；

华馨京官网总线伺服舵机相关文档内容、OpenMV中文入门教程、STM32F103相关数据手册作为参考资料，在必须时候可以查阅。

1.1.上位机软件

上位机软件可以调试总线伺服舵机，测试总线伺服舵机的功能。

- 上位机软件：FashionStar UART总线伺服舵机上位机软件
- 使用说明：总线伺服舵机上位机软件使用说明

1.2.SDK

舵机控制API下载。

- STM32F103_SDK下载链接：SDK for STM32F103

1.3.开发软件

总线伺服舵机转接板使用的USB转TTL串口芯片是`CH340`，需要在Windows上安装驱动。检查驱动是否安装成功

- keil5：keil5下载链接
- STLink驱动：STLink驱动下载链接
- 串口调试助手：XCOM V2.2下载链接

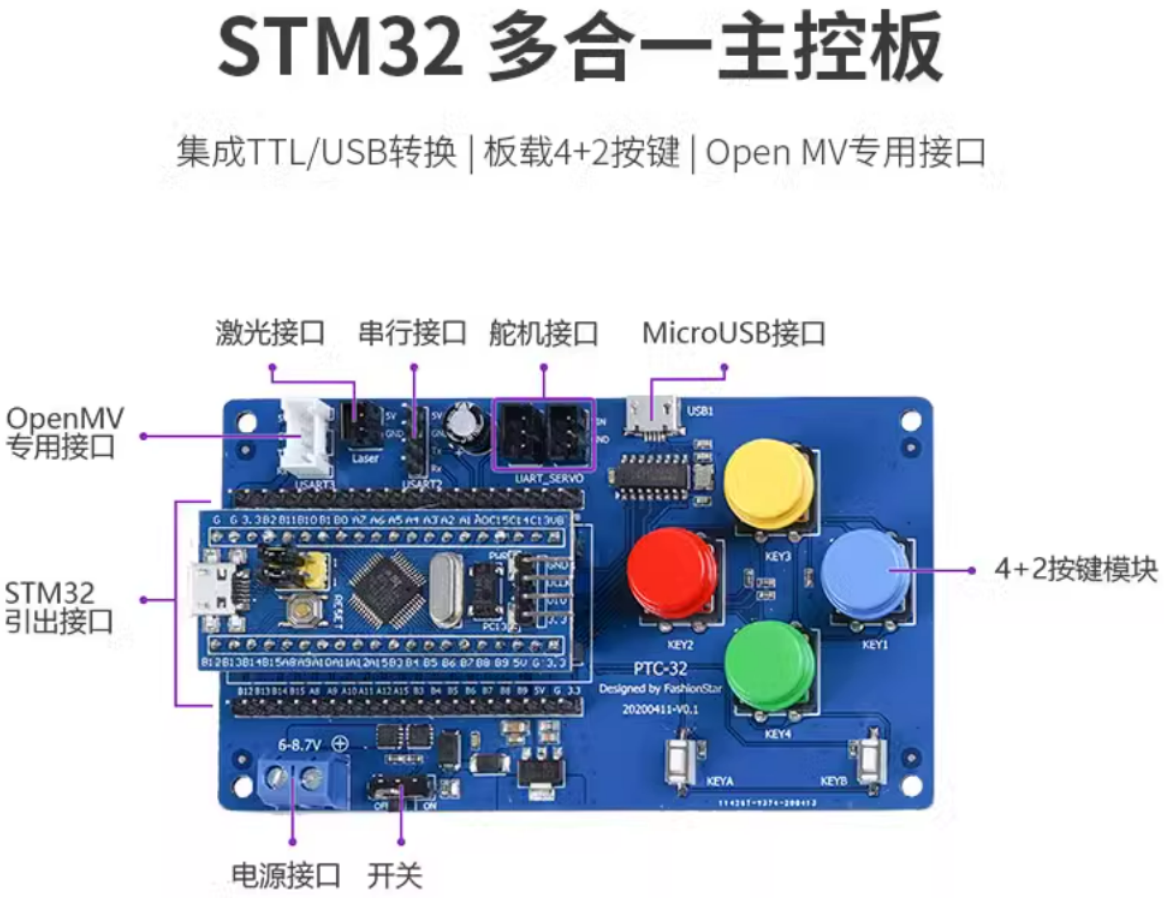
- 串口调试驱动：CH340驱动下载链接

3.STM32F103，OpenMV与舵机云台入门

3.1.STM32 多合一主控板

主控芯片采用STM32F103，集成了TTL/USB转换，4+2按键，OpenMV通信专用接口及舵机接口。

STM32 多合一主控板 可以适配本云台的所有例程，并且具备拓展外设的能力，方便读者发掘云台的更多潜力。

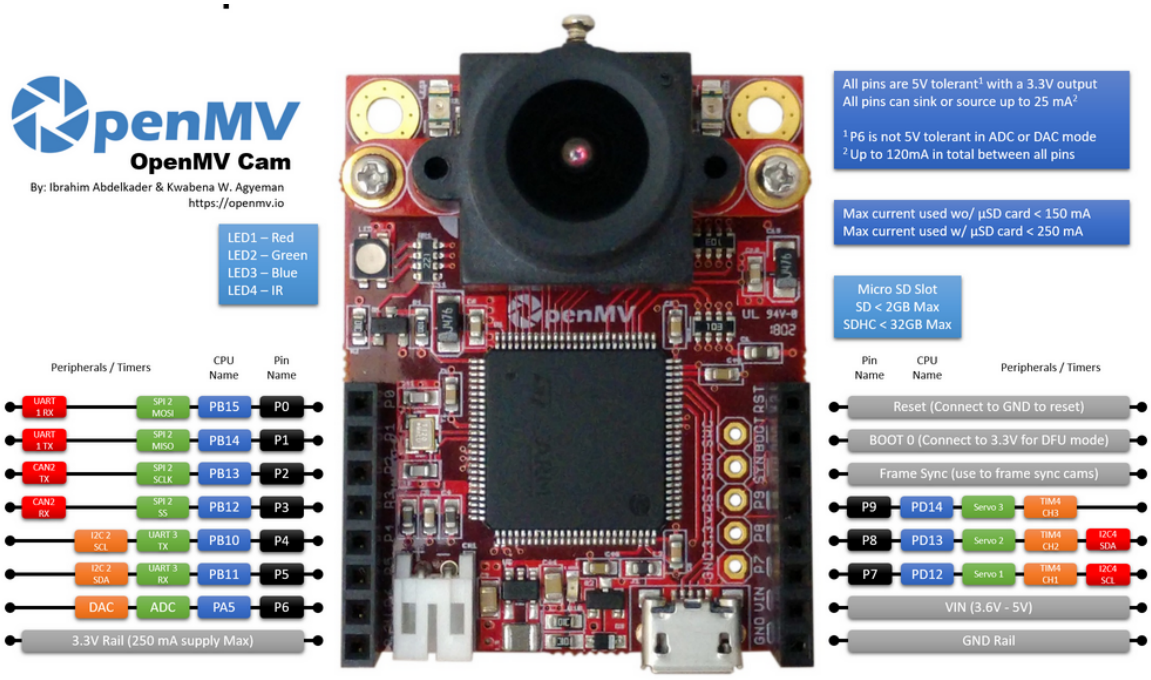


3.2.OpenMV IDE的安装与使用

OpenMV是一款可拓展，支持Python的机器视觉模块，可以通过其内部搭载的MicroPython解释器使用Python编程，实现在舵机云台图像识别并且实时跟随。

OpenMV IDE是集成OpenMV开发的工具，我们需要在PC上下载并安装这个软件，请参考以下视频教程，完成安装。

- OpenMV视频教程-驱动、IDE的安装与使用

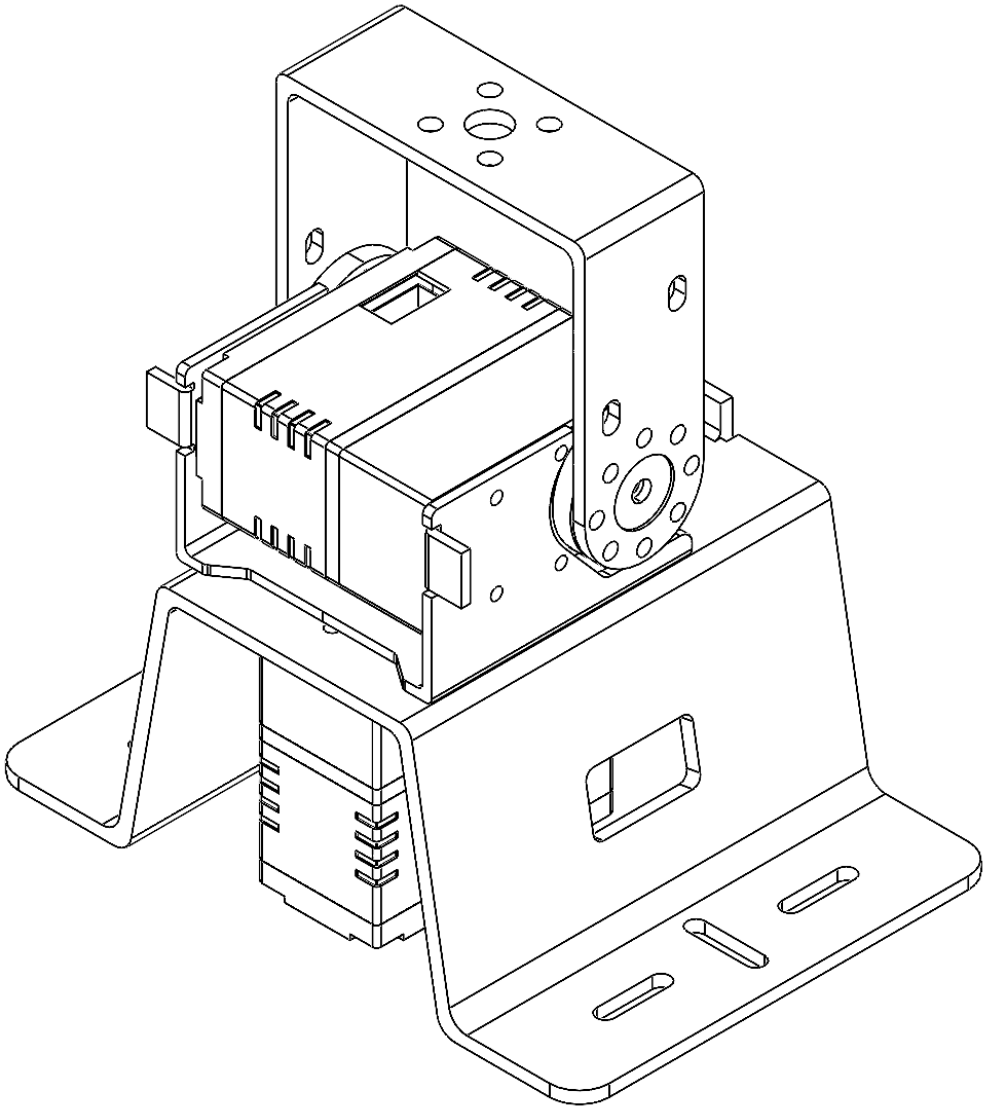


3.3.舵机云台

舵机云台搭载了两颗总线伺服舵机，具备多种功能

- 多种控制模式
- 角度回读
- 零点校准
- 各项异常状态保护
- 舵机状态回读等

并且走线整洁美观，一根3Pin线即可实现主控和总线伺服舵机的供电和通信。



4.舵机云台多模式运动

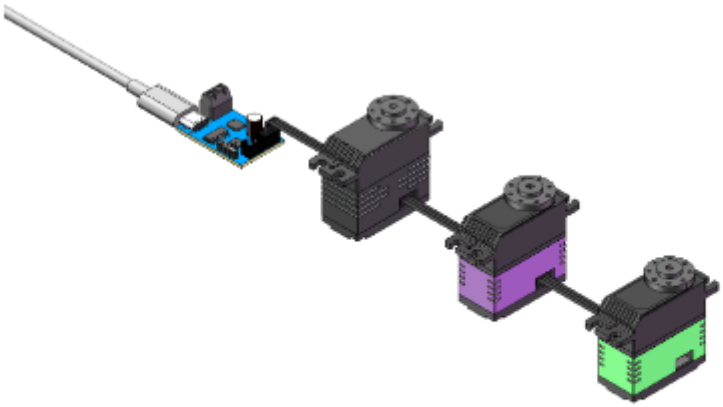
本例程主要演示云台的运动机能，使用 总线伺服舵机SDK使用手册（STM32F103）中的api接口，在没有额外算法辅助的情况下，实现以下功能：

- 以 简易角度控制 的轨迹运动（表现为 启动和停止时没有加减速）
- 以 带加减速的角度控制(指定周期) 的轨迹运动（表现为 启动和停止时有加减速）

4.1.接线说明

可以参考 总线伺服舵机SDK使用手册（STM32F103）以及 连线方式及电源解决方案 文章

- 舵机和UC01的连线（推荐串联，可以保证云台走线的简洁）

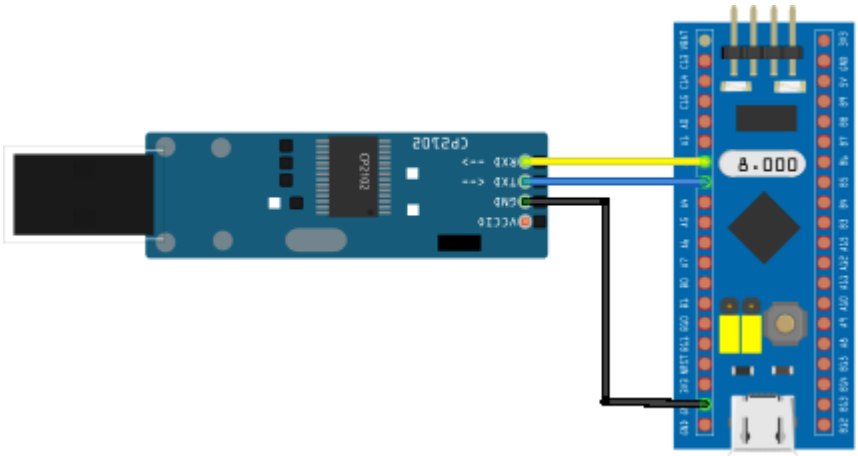


串联

- UC01和STM32F103主控的连线（STM32 多合一主控板有舵机专用接口，无须经由UC01）

STM32F103 GPIO	串口舵机转接板UC01
PA_9 （TX）	RX
PA_10 （RX）	TX
5V（可选）	5V（可选）

- STM32F103主控和USB转TTL模块的连线



USB转TTL模块	STM32F103 GPIO
RX	PA2（TX）
TX	PA3（RX）
5V（可选）	5V（可选）
GND	GND

4.2. STM32主程序源码解析

以下两个宏定义设置了主循环里的运动模式，关于舵机控制API的说明请参考总线伺服舵机SDK（STM32F103）。

```
#define SET_ANGLE_BY_INTERVAL 1 //带加减速的角度控制(指定周期)
#define SET_ANGLE 0 //简易角度控制
```

```
/*****************************************************
 * 舵机云台多模式运动
 *****************************************************/
#include "stm32f10x.h"
#include "math.h"
#include "usart.h"
#include "sys_tick.h"
#include "fashion_star_uart_servo.h"
// #include "gimbal.h"

// 使用串口1作为舵机控制的端口
// <接线说明>
// STM32F103 PA9(Tx)  <----> 串口舵机转接板 Rx
// STM32F103 PA10(Rx) <----> 串口舵机转接板 Tx
// STM32F103 GND      <----> 串口舵机转接板 GND
// STM32F103 V5       <----> 串口舵机转接板 5V
// <注意事项>
// 使用前确保已设置usart.h里面的USART1_ENABLE为1
// 设置完成之后，将下行取消注释
Usart_DataTypeDef *servoUsart = &usart1;

#define YAW_SERVO 0
#define PIT_SERVO 1

// 选择角度控制方式
#define SET_ANGLE_BY_INTERVAL 1 //带加减速的角度控制(指定周期)
#define SET_ANGLE 0 //简易角度控制


int main(void)
{
    // 嘀嗒定时器初始化
    SysTick_Init();
    Usart_Init(); // 串口初始化

    FSUS_SetServoAngleByInterval(servoUsart, YAW_SERVO, -10, 500, 100, 100, 0,
0);
    SysTick_DelayMs(1000);

    FSUS_SetServoAngleByInterval(servoUsart, PIT_SERVO, 0, 500, 100, 100, 0, 0);
    // 等待2s
    SysTick_DelayMs(1000);

    while (1)
```

```
    {
        #if SET_ANGLE_BY_INTERVAL
        //设置舵机的角度(指定周期) 参数:串口结构体、舵机编号、角度、目标时间、加速时间、减速时间、执行功率（默认0）、是否阻塞至执行完成：1
        FSUS_SetServoAngleByInterval(servoUsart, YAW_SERVO, -30, 300, 100, 100, 0, 1);
        FSUS_SetServoAngleByInterval(servoUsart, YAW_SERVO, 30, 600, 300, 300, 0, 1);
        FSUS_SetServoAngleByInterval(servoUsart, YAW_SERVO, 0, 300, 100, 100, 0, 1);

        FSUS_SetServoAngleByInterval(servoUsart, PIT_SERVO, -20, 300, 100, 100, 0, 1);
        FSUS_SetServoAngleByInterval(servoUsart, PIT_SERVO, 20, 600, 300, 300, 0, 1);
        FSUS_SetServoAngleByInterval(servoUsart, PIT_SERVO, 0, 300, 100, 100, 0, 1);


        #elif SET_ANGLE
        //简易角度控制 参数：串口结构体、舵机编号、角度、目标时间、执行功率（默认0）、是否阻塞至执行完成：1
        FSUS_SetServoAngle(servoUsart, YAW_SERVO, -30, 300,  0, 1);
        FSUS_SetServoAngle(servoUsart, YAW_SERVO, 30, 600, 0, 1);
        FSUS_SetServoAngle(servoUsart, YAW_SERVO, 0, 300, 0, 1);

        FSUS_SetServoAngle(servoUsart, PIT_SERVO, -20, 300, 0, 1);
        FSUS_SetServoAngle(servoUsart, PIT_SERVO, 20, 600, 0, 1);
        FSUS_SetServoAngle(servoUsart, PIT_SERVO, 0, 300, 0, 1);

        #endif
    }
}
```

5.按键中断控制云台转动

本例程使用按键中断以及总线伺服舵机的单圈角度控制功能，实现用按键控制云台双轴的转动

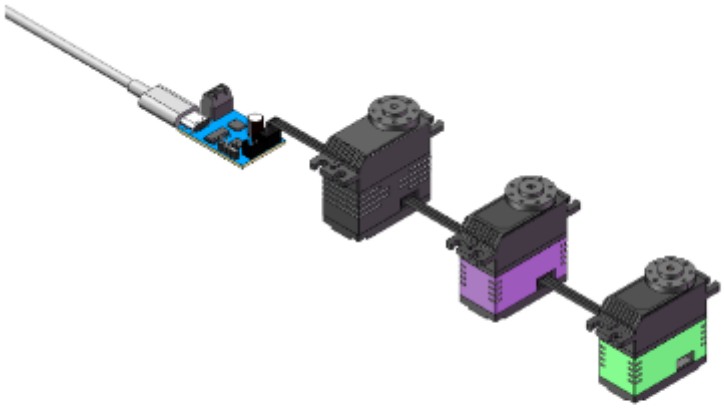
- 按键：是常用的人机交互模块，当按键连接的是高电平时，GPIO扫描到的为高电平，连接低电平同理。
- 外部中断：是STM32F103中断的一种，配置为下降沿触发，当按键按下时，电平由高转低，触发下降沿，立即进入一次外部中断回调函数。因此，结合按键使用，做到实时发送控制命令。
- 串口通信：是STM32F103和总线伺服舵机通信的软件部分，STM32F103串口通信TX RX是独立的，需要借由UC01总线伺服舵机转接板将TX RX分时接入UART总线上，实现与总线上的舵机双向通信。
- 总线伺服舵机：使用UART总线通信，由控制命令控制其工作，控制命令有多种，于云台应用上推荐使用 单圈角度控制 系列API。

有关总线伺服舵机API详细介绍请参考总线伺服舵机SDK使用手册（STM32F103）。

5.1.接线说明

可以参考 总线伺服舵机SDK使用手册（STM32F103）以及 连线方式及电源解决方案 文章

- 舵机和UC01的连线（推荐串联，可以保证云台走线的简洁）

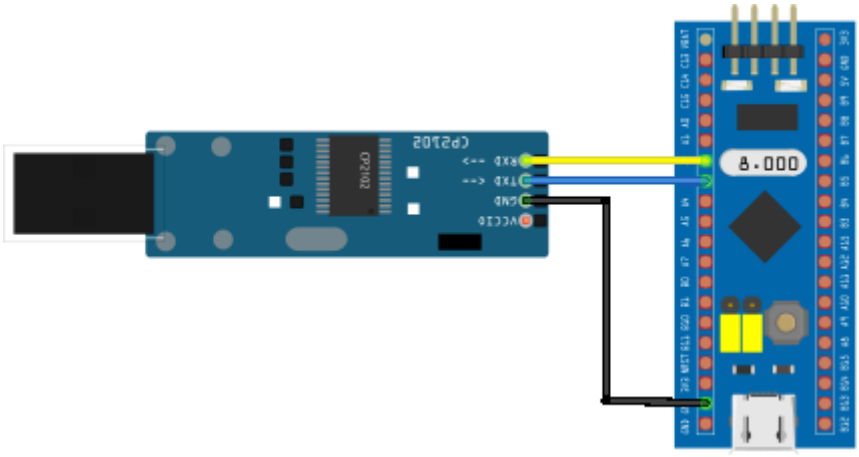


串联

- UC01和STM32F103主控的连线（STM32 多合一主控板有舵机专用接口，无须经由UC01）

STM32F103 GPIO	串口舵机转接板UC01
PA_9 （TX）	RX
PA_10 （RX）	TX
5V（可选）	5V（可选）
GND	GND

- STM32F103主控和USB转TTL模块的连线



USB转TTL模块	STM32F103 GPIO
RX	PA2（TX）

USB转TTL模块	STM32F103 GPIO
TX	PA3（RX）
5V（可选）	5V（可选）
GND	GND

- STM32F103主控和按键的连线

STM32F103	按键	备注
PB_4	KEY1	偏航角增加 + 按键
PB_5	KEY2	偏航角减少 - 按键
PB_6	KEY3	俯仰角增加 + 按键
PB_7	KEY4	俯仰角减少 - 按键

5.2. `STM32主程序源码` 解析

```
button_yaw_Flag、button_pit_Flag
```

是控制程序发送舵机命令的标志，外部中断由于按键单次按下时电平会抖动，因此并非 *单次按下=一次电平跳变*。

在主程序循环中使用Flag标志，可以控制程序发送舵机命令的频率

在 `button.h` 文件中可以设置云台的角度范围和角度步进值

```
// 云台的角度范围
#define YAW_MIN -90
#define YAW_MAX 90
#define PITCH_MIN -90
#define PITCH_MAX 50

// 角度步进值
#define BUTTON_ANGLE_STEP 5
```

```
/**************************************************
 * 按键中断控制云台
 **************************************************/
#include "stm32f10x.h"
#include "usart.h"
#include "sys_tick.h"
#include "fashion_star_uart_servo.h"
```

```c
#include "button.h"

#define YAW_SERVO_ID 0
#define PIT_SERVO_ID 1
#define SERVO_DEAD_BLOCK 2
// 使用串口1作为舵机控制的端口
// <接线说明>
// STM32F103 PA9(Tx)  <----> 串口舵机转接板 Rx
// STM32F103 PA10(Rx) <----> 串口舵机转接板 Tx
// STM32F103 GND      <----> 串口舵机转接板 GND
// STM32F103 V5       <----> 串口舵机转接板 5V
// <注意事项>
// 使用前确保已设置usart.h里面的USART1_ENABLE为1
// 设置完成之后，将下行取消注释
Usart_DataTypeDef *servoUsart = &usart1;
// 使用串口2作为日志输出的端口
// <接线说明>
// STM32F103 PA2(Tx) <----> USB转TTL Rx
// STM32F103 PA3(Rx) <----> USB转TTL Tx
// STM32F103 GND     <----> USB转TTL GND
// STM32F103 V5      <----> USB转TTL 5V (可选)
// <注意事项>
// 使用前确保已设置usart.h里面的USART2_ENABLE为1
Usart_DataTypeDef *loggingUsart = &usart2;

// 重定向c库函数printf到串口，重定向后可使用printf函数
int fputc(int ch, FILE *f)
{
    while ((loggingUsart->pUSARTx->SR & 0X40) == 0)
    {
    }
    /* 发送一个字节数据到串口 */
    USART_SendData(loggingUsart->pUSARTx, (uint8_t)ch);
    /* 等待发送完毕 */
    // while (USART_GetFlagStatus(USART1, USART_FLAG_TC) != SET);
    return (ch);
}

float servoSpeed = 100.0; // 云台旋转速度 (单位: °/s)

int main(void)
{
    SysTick_Init(); // 嘀嗒定时器初始化
    Usart_Init();   // 串口初始化
    // Gimbal_Init(servoUsart);// 云台初始化
    Button_Init();          // 按键初始化
    FSUS_SetServoAngle(servoUsart, YAW_SERVO_ID, yaw_set, 200, 0, 0);
    FSUS_SetServoAngle(servoUsart, PIT_SERVO_ID, pitch_set, 200, 0, 0);
    SysTick_DelayMs(2000); // 等待2s


    while (1)
    {
        if (button_yaw_Flag)
        {
            printf("nextYaw: %.1f  nextPitch: %.1f\r\n", yaw_set, pitch_set);
```

```
            FSUS_SetServoAngle(servoUsart, YAW_SERVO_ID, yaw_set, 200, 0, 0);
            SysTick_DelayMs(100);
            button_yaw_Flag = 0;
        }
        if (button_pit_Flag)
        {
            printf("nextYaw: %.1f  nextPitch: %.1f\r\n", yaw_set, pitch_set);
            FSUS_SetServoAngle(servoUsart, PIT_SERVO_ID, pitch_set, 200, 0, 0);
            SysTick_DelayMs(100);
            button_pit_Flag = 0;

        }

    }
}
```

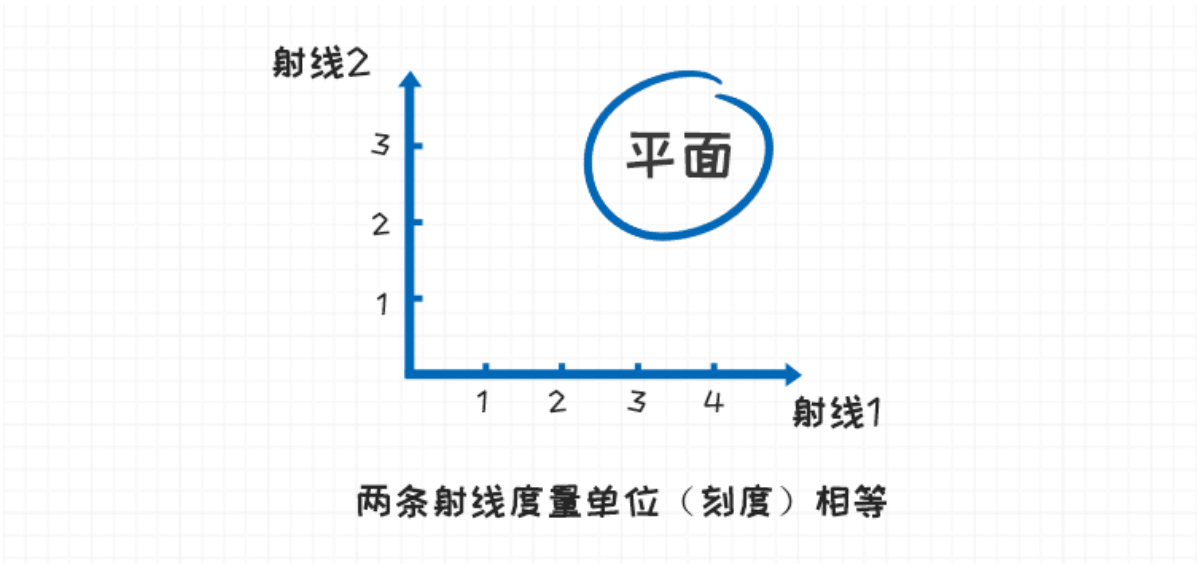
6.机器人的坐标系与位姿

6.1.笛卡尔坐标系

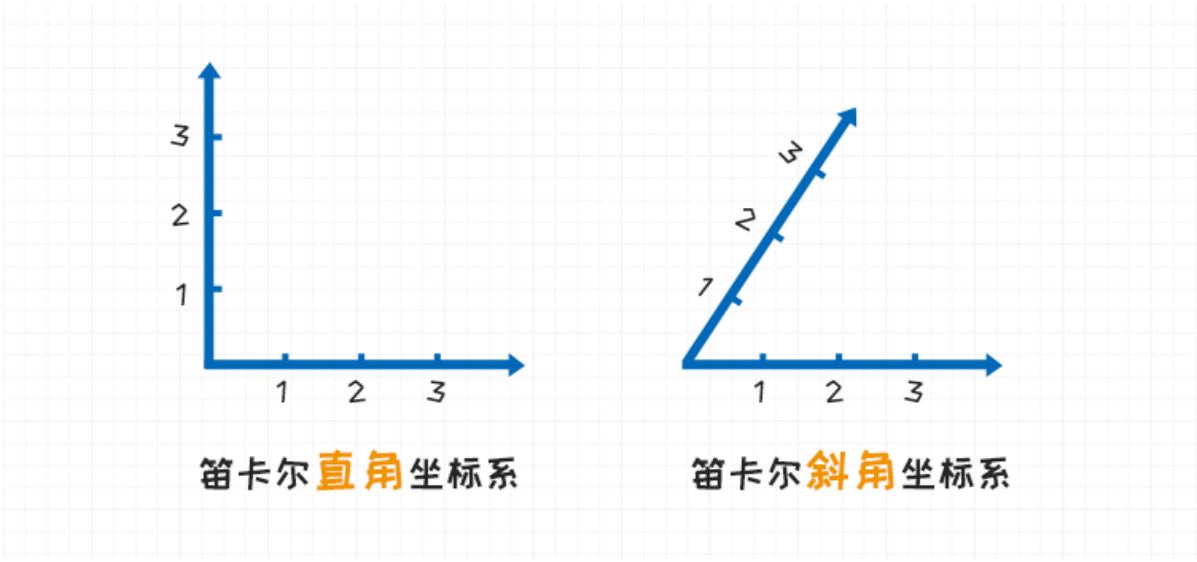
在空间中有两条或三条 **数轴** 相交于原点。

两条数轴构成了一个 **平面** ，三条数轴构成了一个立体的空间。如果数轴上的刻度(度量单位)一致，那么形成的这个坐标系就叫做 **笛卡尔坐标系** 。

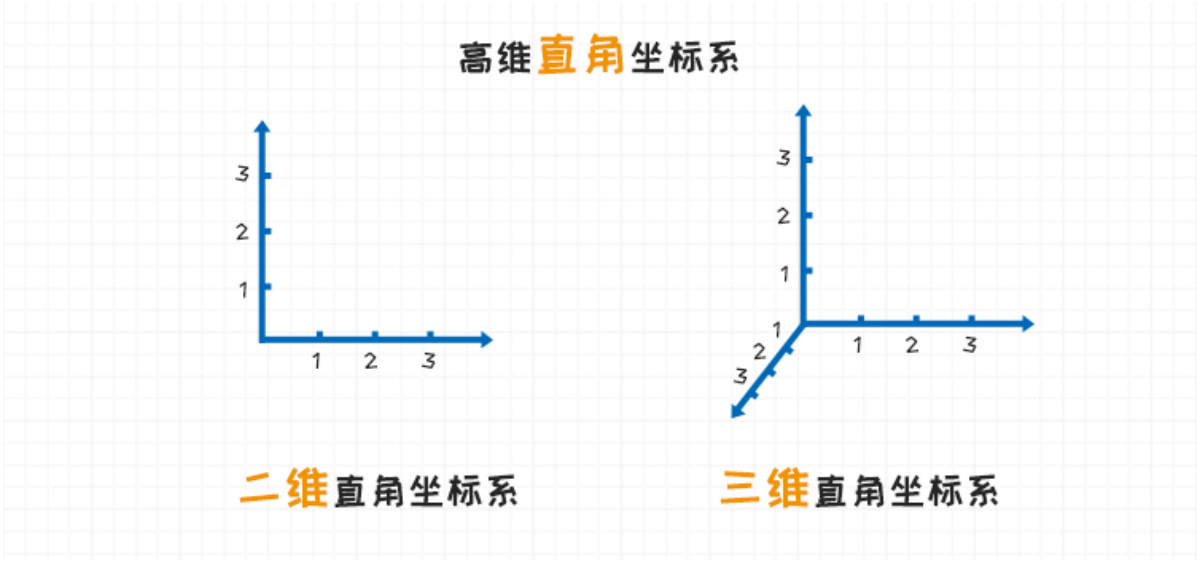
笛卡尔坐标 的英文叫做 **Cartesian Coordinate System**。



根据数轴之间是否垂直，笛卡尔坐标系又可以分为 **笛卡尔直角坐标系** 和 **笛卡尔斜角坐标系** 。其中笛卡尔直角坐标系我们用的比较多。



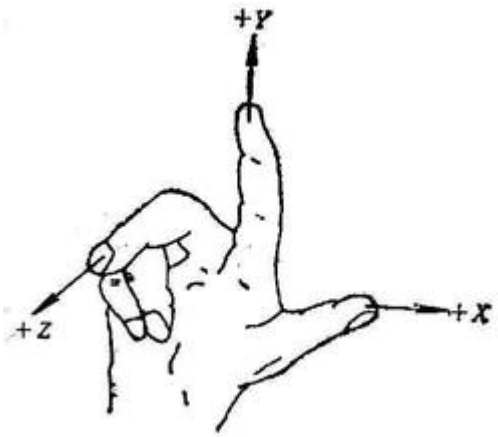
根据维度（数轴个数）的不同，笛卡尔直角坐标系又可以分为 **二维直角坐标系** 和 **三维直角坐标系。**



另外，不管是二维的还是三维的都存在一个问题，如果我修改坐标轴的正方向，那此时点在坐标系中的位置就会是一个相反的值。根据轴与轴之间正方向关系，笛卡尔直角坐标系，又分为 **左手直角坐标系** 和 **右手直角坐标系。**

在空间直角坐标系中，让右手拇指指向x轴的正方向，食指指向y轴的正方向，如果中指能指向z轴的正方向，则称这个坐标系为 **右手直角坐标系** ，反之则是 **左手直角坐标系** 。

下图为右手坐标系的示意图：



在数学中则通常使用右手坐标系，我们研究机器人在空间中的关系时，使用的也是 **右手坐标系。**

6.2.位姿描述

位姿 Pose 是两个属性 **位置 Position** 和 **旋转 Orientation** 的叠加。

说到 **位置 Position** ，大家比较熟悉

- 在二维直角坐标系下，(x, y) 两个值可以确定一个点在平面中的位置。
- 在三维的直角坐标系下，(x, y, z) 三个值可以确定一个点在立体空间中的位置。

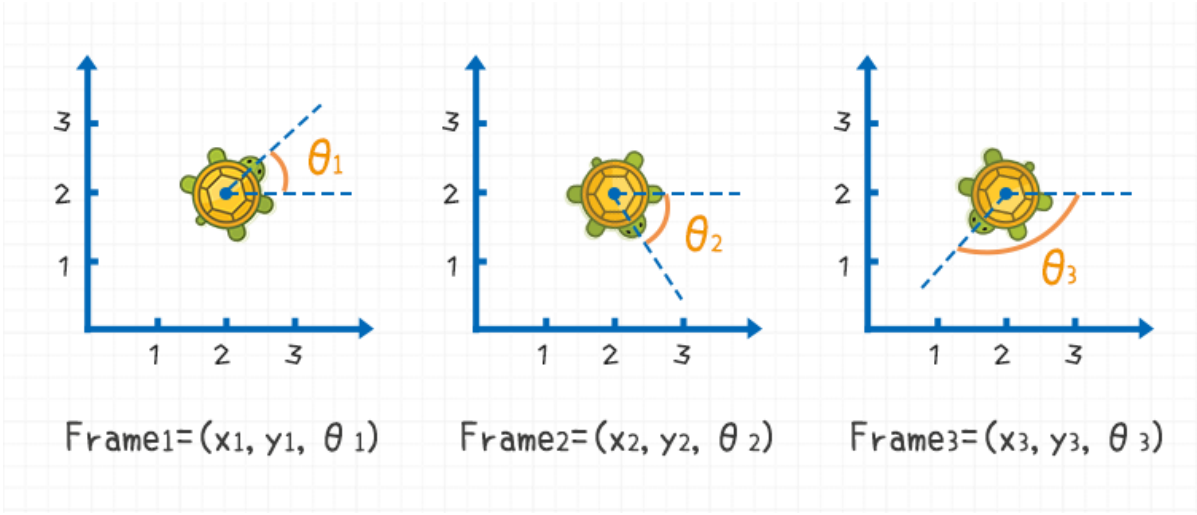
旋转 Orientation 用于刻画机器人在当前坐标系下在各个轴上面的 **倾角** 。

- 对于一个机器人来讲，它并不能被看成一个点，它有体积，在同一个点上可以有不同的 **朝向** 或者是说 **倾角**

接下来，我们来分析一下在二维和三维下的位姿。

二维直角坐标系

拿小乌龟举例，小乌龟在平面上运动，将小乌龟的头部所指的方向，作为小乌龟的朝向。在地面上建立一个二维直角坐标系。



图中，小乌龟的 **位置Position** 都相等。$x_1 = x_2 = x_3、y_1 = y_2 = y_3$

但是小乌龟的 **旋转** 各不相同，使用θ来表示小乌龟的旋转。

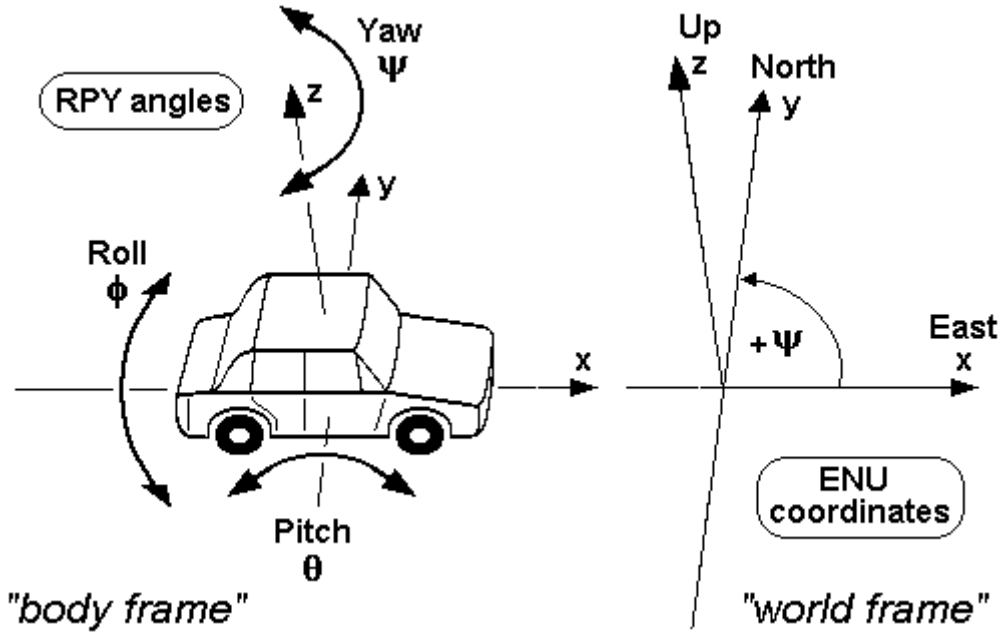
- 将x转向y轴方向作为角度正方向，θ 的角度取值范围为 [-180°, 180°]
- 在工程领域习惯使用 **弧度 Radian** 来表示角度，所以转换为弧度 θ 的取值范围为 $-\pi<= \theta <= \pi$

三维直角坐标系

三维坐标系下，位置描述依然简单，但是描述旋转就有些麻烦了。

我们引入一种方法叫做 **RPY角 (Roll，Pitch，Yaw)** 描述法，也叫 **固定XYZ轴角 (X-Y-Z Fixed Angles)** 描述法

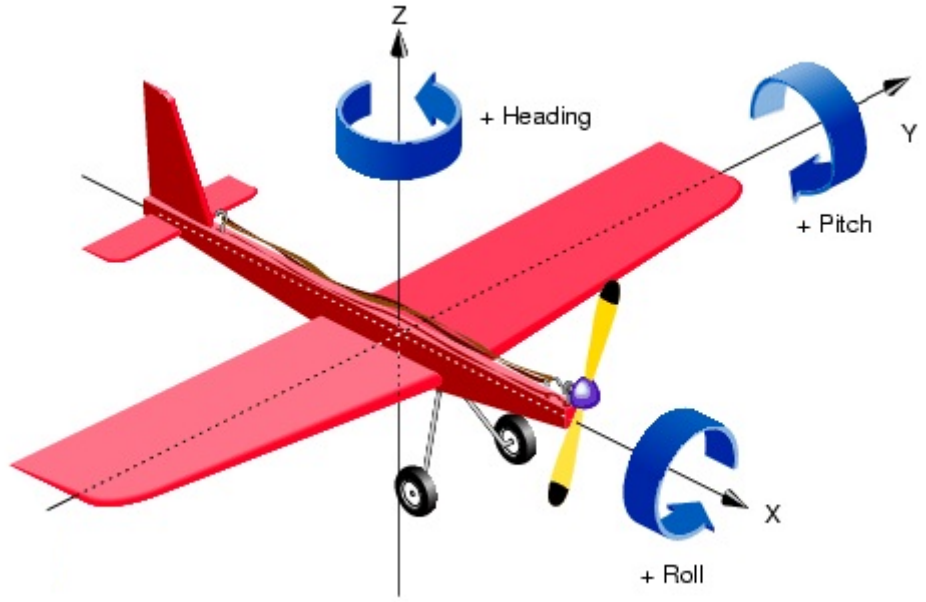
以小车模型为例，小车坐标系定义下图所示：



> 图片来源 :RPY_angles_of_cars

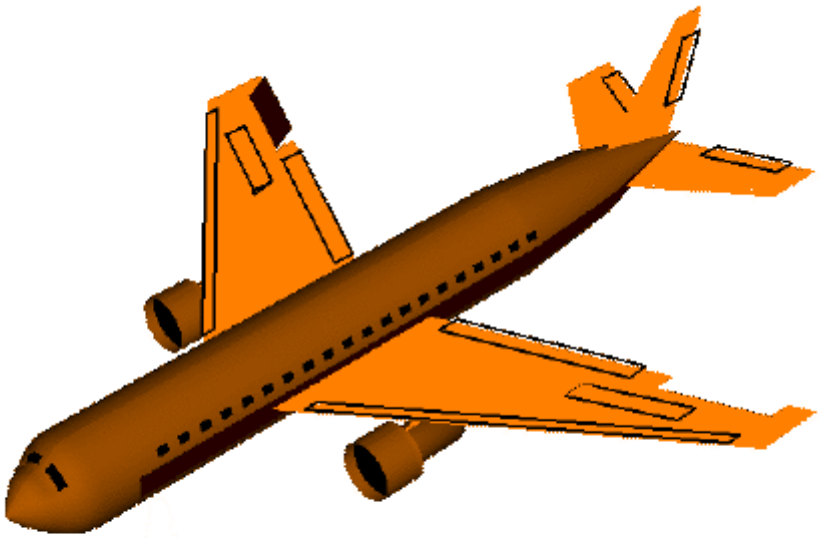
云台的RPY角度

飞机的RPY角如下图所示，

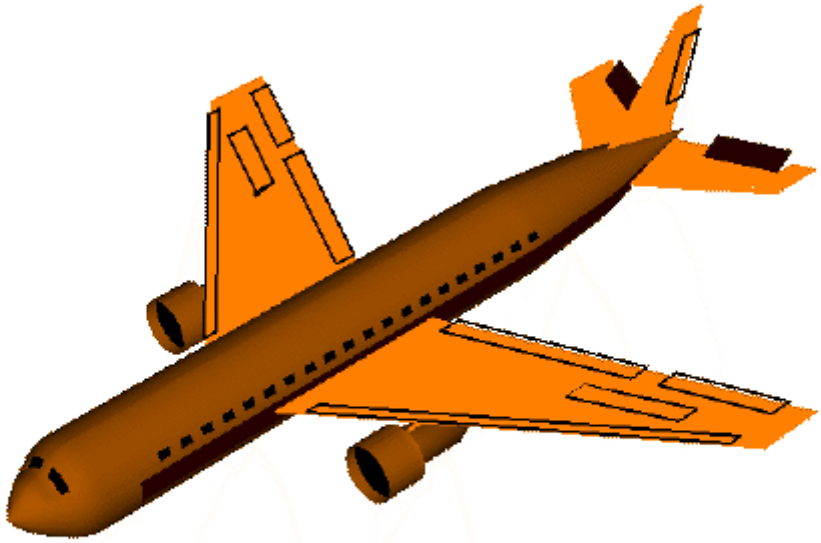


按照先后顺序 **依次** 进行：

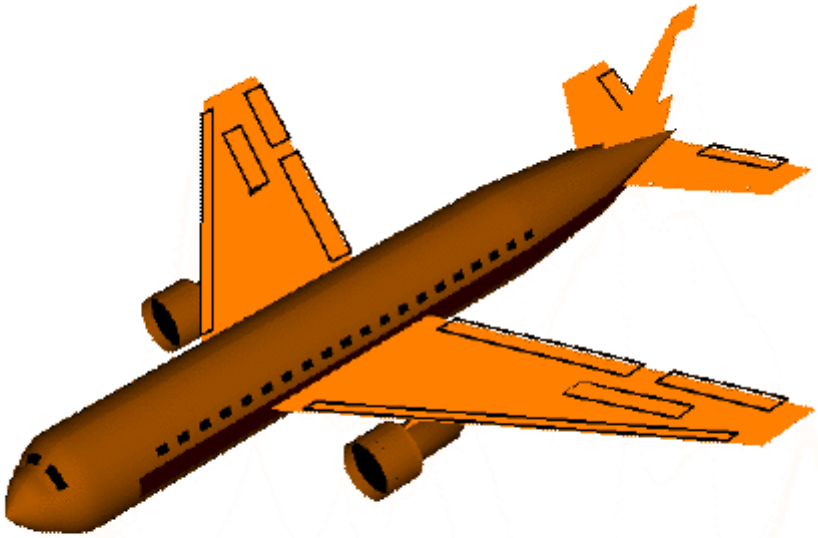
- **步骤 1** 按照参考坐标系的X轴旋转 **Roll** 角度，Roll中文名字叫做 **横滚角** 。



- **步骤2**：飞机按照参考坐标系的Y轴旋转 **Pitch** 角度，Pitch的中文叫做 **俯仰角。**



- **步骤3** 按照参考坐标系的Z轴旋转 **Yaw / Heading** 角度，中文名字叫做 **偏航角/航向角。**



英文	中文	参考轴	别名
Roll	横滚角	X	回转角
Pitch	俯仰角	Y	
Yaw	偏航角	Z	航向角

所以对应在3D空间下的Frame就应该是

$$Frame = (x, y, z, pitch, roll, yaw)$$

7.云台RPY角标定

OpenMV识别时，识别物体的中心坐标并不能直接用于控制云台运动，需要对舵机云台的RPY角和舵机的原始角度进行一次映射。然后使用舵机的单圈角度控制功能控制舵机旋转到对应的角度。

舵机云台的RPY角（Pitch、Yaw）和舵机的原始角度之间存在线性映射关系。

$$yaw = k_{yaw} * angle_0 + b_{yaw}$$
$$pitch = k_{pitch} * angle_1 + b_{pitch}$$

其中

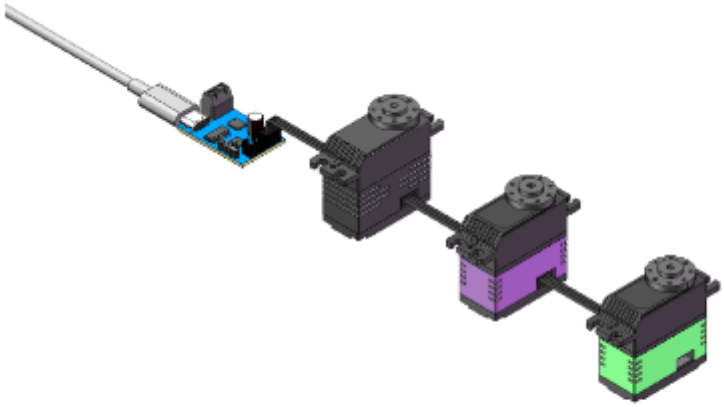
- yaw 、 $pitch$ 是云台的RPY角
- $angle_0$ 、 $angle_1$ 是舵机的原始角度
- $k*{yaw}$ 、 $*b*{yaw}$ 、 $k*{pitch}$ 、 $*b*{pitch}$ 未知。

通过已知云台的RPY角和舵机的原始角度，可以求解 $k*{yaw}$ 、 $*b*{yaw}$ 、 $k*{pitch}$ 、 $*b*{pitch}$。

7.1.接线说明

可以参考 总线伺服舵机SDK使用手册（STM32F103）以及 连线方式及电源解决方案 文章

- 舵机和UC01的连线（推荐串联，可以保证云台走线的简洁）

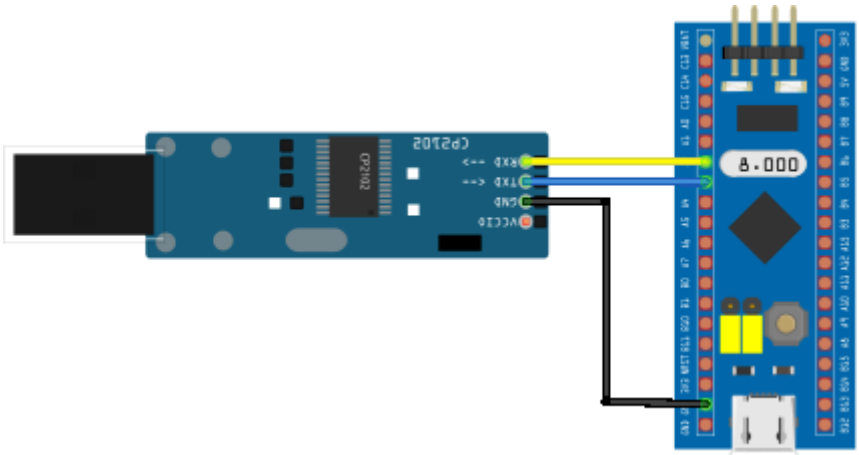


串联

- UC01和STM32F103主控的连线（STM32 多合一主控板有舵机专用接口，无须经由UC01）

STM32F103 GPIO	**串口舵机转接板UC01**
PA_9 （TX）	RX
PA_10 （RX）	TX
5V（可选）	5V（可选）
GND	GND

- STM32F103主控和USB转TTL模块的连线



USB转TTL模块	**STM32F103 GPIO**
RX	PA2（TX）
TX	PA3（RX）
5V（可选）	5V（可选）
GND	GND

USB转TTL模块	STM32F103 GPIO
GND	GND

7.2.采集标定数据

运行 `标定数据采集` 例程，烧录代码。

源代码

```
/**************************************************
 * 标定数据采集
 *   设置舵机为阻尼模式后，用手旋转云台的两个舵机，
 *   串口2每隔一段时间打印一下舵机角度信息
 **************************************************/
#include "stm32f10x.h"
#include "usart.h"
#include "sys_tick.h"
#include "fashion_star_uart_servo.h"

#define SERVO_DOWN 0 // 云台下方的舵机ID
#define SERVO_UP 1 // 云台上方的舵机ID

// 使用串口1作为舵机控制的端口
// <接线说明>
// STM32F103 PA9(Tx)  <----> 串口舵机转接板 Rx
// STM32F103 PA10(Rx) <----> 串口舵机转接板 Tx
// STM32F103 GND      <----> 串口舵机转接板 GND
// STM32F103 V5       <----> 串口舵机转接板 5V
// <注意事项>
// 使用前确保已设置usart.h里面的USART1_ENABLE为1
// 设置完成之后，将下行取消注释
Usart_DataTypeDef* servoUsart = &usart1;
// 使用串口2作为日志输出的端口
// <接线说明>
// STM32F103 PA2(Tx) <----> USB转TTL Rx
// STM32F103 PA3(Rx) <----> USB转TTL Tx
// STM32F103 GND     <----> USB转TTL GND
// STM32F103 V5      <----> USB转TTL 5V (可选)
// <注意事项>
// 使用前确保已设置usart.h里面的USART2_ENABLE为1
Usart_DataTypeDef* loggingUsart = &usart2;

// 重定向c库函数printf到串口，重定向后可使用printf函数
int fputc(int ch, FILE *f)
{
    while((loggingUsart->pUSARTx->SR&0X40)==0){}
    /* 发送一个字节数据到串口 */
    USART_SendData(loggingUsart->pUSARTx, (uint8_t) ch);
    /* 等待发送完毕 */
```

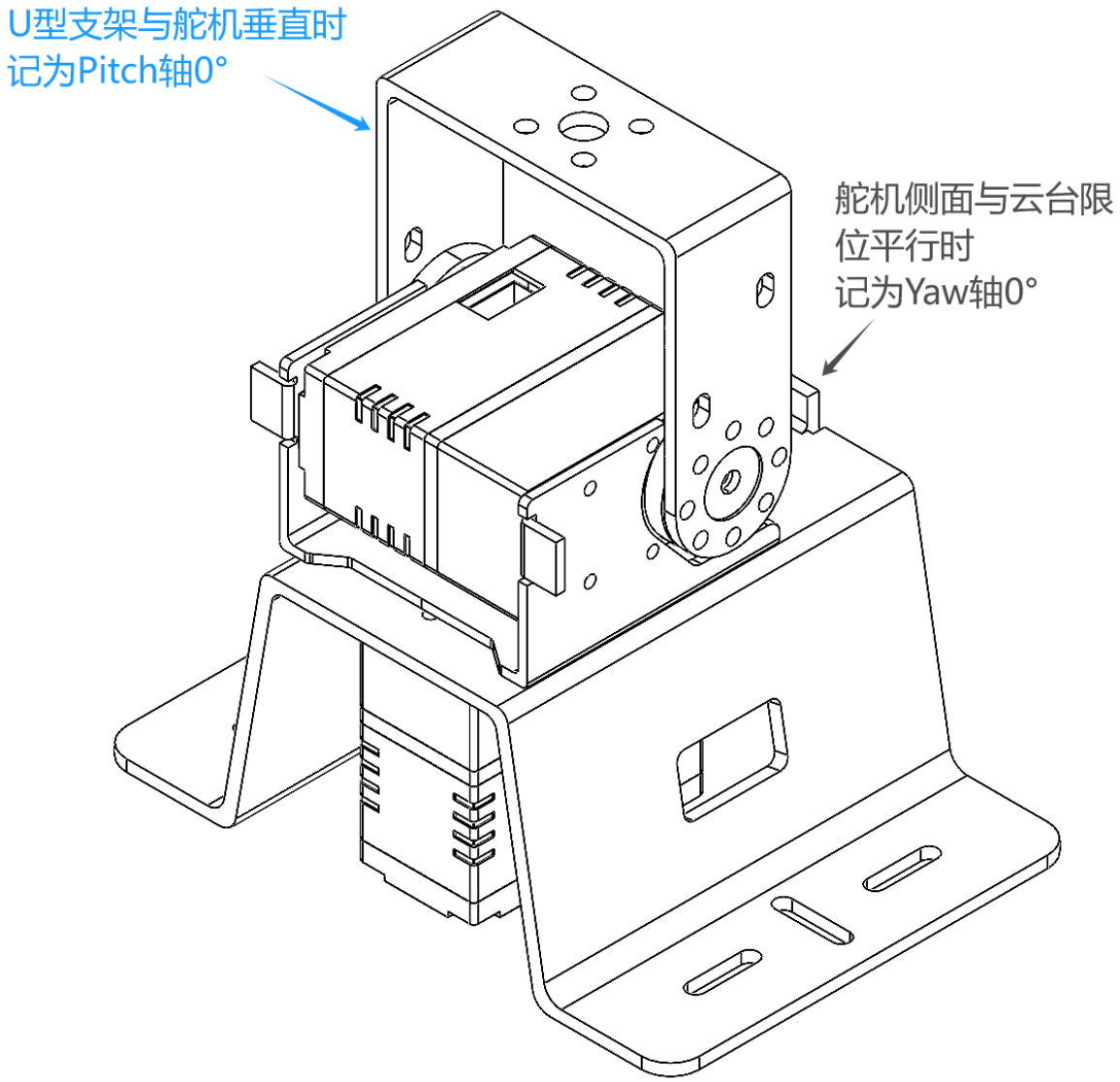
```
    // while (USART_GetFlagStatus(USART1, USART_FLAG_TC) != SET);
    return (ch);
}

FSUS_STATUS statusCode; // 请求包的状态码

float servoDownAngle = 0; // 下方舵机的角度
float servoUpAngle = 0;     // 上方舵机的角度

// 云台初始化-设置为阻尼模式
void InitGimbal(void){
    const uint16_t power = 500; // 阻尼模式下的功率，功率越大阻力越大
    Usart_Init(); // 串口初始化
    FSUS_DampingMode(servoUsart, SERVO_DOWN, power); // 设置舵机0为阻尼模式
    FSUS_DampingMode(servoUsart, SERVO_UP, power); // 设置舵机1为阻尼模式
}

// 更新舵机云台舵机的角度
void UpdateGimbalSrvAngle(void){
    uint8_t code;
    code = FSUS_QueryServoAngle(servoUsart, SERVO_DOWN, &servoDownAngle);
    printf("status code : %d \r\n", code);
    code = FSUS_QueryServoAngle(servoUsart, SERVO_UP, &servoUpAngle);
    printf("status code : %d \r\n", code);
}


int main (void)
{
    // 嘀嗒定时器初始化
    SysTick_Init();
    InitGimbal();

    while (1){
        // 更新云台舵机角度
        UpdateGimbalSrvAngle();
        // 打印一下当前舵机的角度信息
        printf("Servo Down: %.2f;  Servo Up: %.2f \r\n", servoDownAngle,
servoUpAngle);
        // 等待200ms
        SysTick_DelayMs(200);
    }
}
```

7.3.记录标定数据

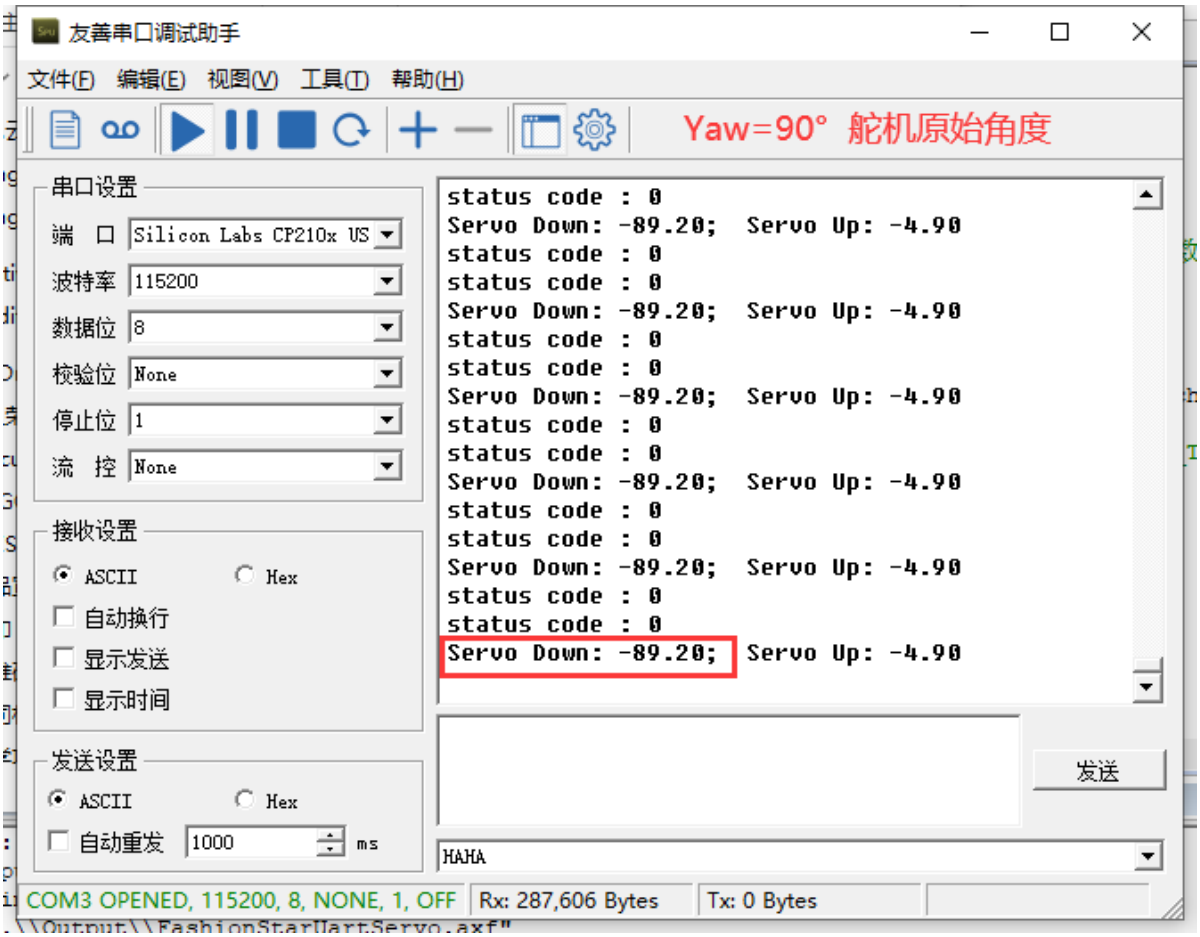
烧录代码后，手动调整舵机，

Yaw轴角度以逆时针方向增加，当舵机侧面与云台限位平行时，记为Yaw轴0°。

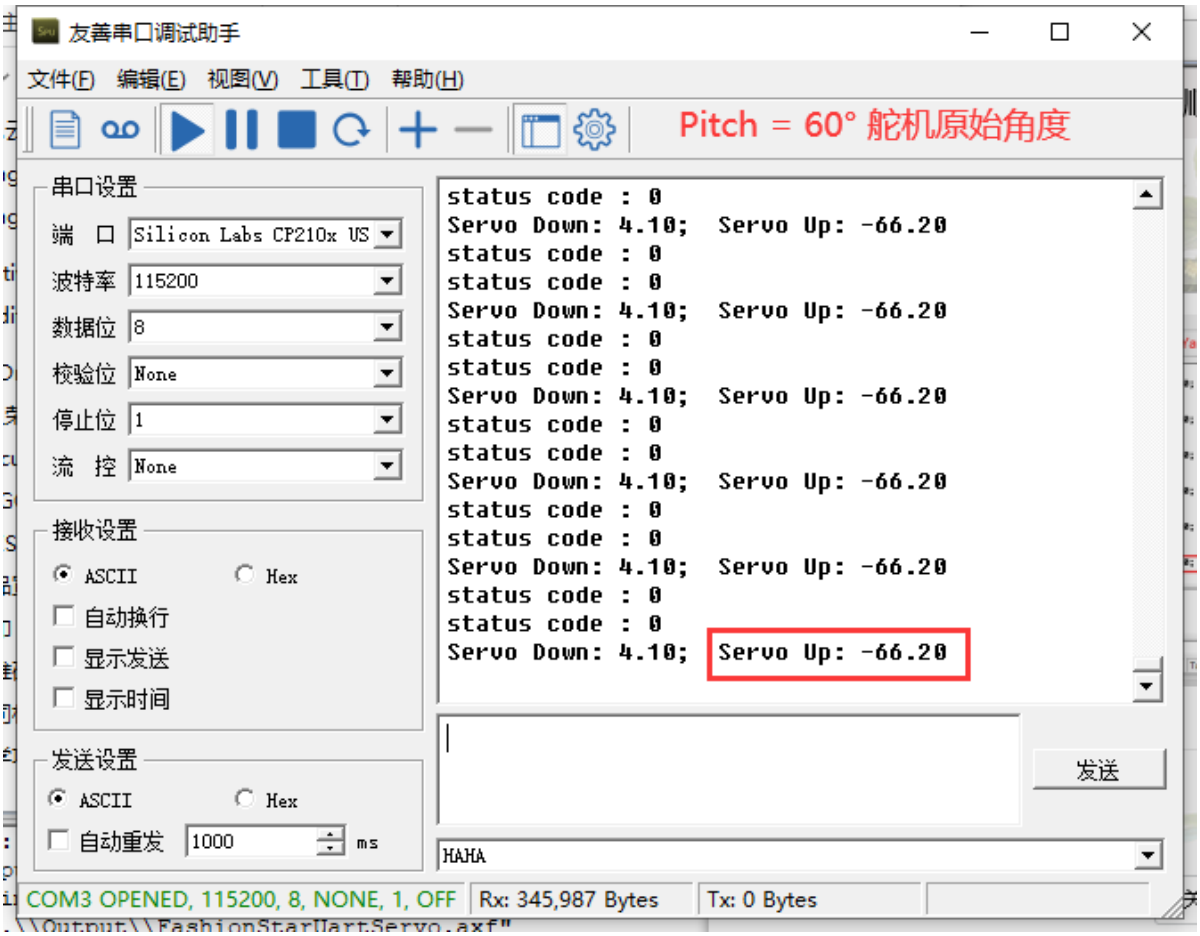
Pitch轴角度从舵机侧面看支架与侧面平行时，记为90°，U型支架与舵机垂直时，记为0°，即如图所示情况。



1.手持舵机云台，转动云台，调整舵机云台Yaw轴为90°，通过串口助手查看并且记录原始舵机角度，然后调整舵机云台Yaw轴为90°，记录舵机原始角度。



2.重复上述步骤，记录云台Pitch轴为-90°和60°时舵机的原始角度。



7.4.写入标定数据

> **注意事项**:
>
> 更换例程时需要重新写入标定数据到对应的 gimbal.h 文件。

打开欲使用工程的 FashionStarUartServo/User/gimbal/ 目录下的.h文件 `gimbal.h` 。

将在上一步中采集得到的标定数据写入到 `gimbal.h` 里面的宏定义中。

例如:

采集到的数据

偏航角Yaw	宏定义	舵机ID	原始舵机角度
90.0	YAW1_SERVO_ANGLE	0	-89.2
-90.0	YAW2_SERVO_ANGLE	0	94

俯仰角 Pitch	宏定义	舵机ID	原始舵机角度
60.0	PITCH1_SERVO_ANGLE	1	-66.2
-90	PITCH2_SERVO_ANGLE	1	86.8

写入 `gimbal.h`

```
// 云台舵机的标定数据
#define YAW1 90.0
#define YAW1_SERVO_ANGLE -89.2
#define YAW2 -90.0
#define YAW2_SERVO_ANGLE 94

#define PITCH1 60.0
#define PITCH1_SERVO_ANGLE -66.2
#define PITCH2 -90.0
#define PITCH2_SERVO_ANGLE 86.8
```

7.5.`STM32主程序源码` 解析

将标定数据写入本例程，对标定结果进行测试。

本例程会控制云台舵机在两个不同的位姿之间循环转动，请在确认云台在转动不受限的情况下使用。

修改 `while(1)` 循环中的角度，检验标定结果

```c
/*****************************************************
 * 标定结果测试
 * 主循环里设置云台控制目标，在两个姿态之间重复旋转。
 *****************************************************/
#include "stm32f10x.h"
#include "usart.h"
#include "sys_tick.h"
#include "fashion_star_uart_servo.h"
#include "gimbal.h"


// 使用串口1作为舵机控制的端口
// <接线说明>
// STM32F103 PA9(Tx)  <----> 串口舵机转接板 Rx
// STM32F103 PA10(Rx) <----> 串口舵机转接板 Tx
// STM32F103 GND      <----> 串口舵机转接板 GND
// STM32F103 V5       <----> 串口舵机转接板 5V
// <注意事项>
// 使用前确保已设置usart.h里面的USART1_ENABLE为1
// 设置完成之后，将下行取消注释
Usart_DataTypeDef* servoUsart = &usart1;

float servoSpeed = 200.0; // 云台旋转速度 (单位: °/s)
int main (void)
{
    // 嘀嗒定时器初始化
    SysTick_Init();
    Usart_Init(); // 串口初始化
    // 云台初始化
    Gimbal_Init(servoUsart);
    // 等待2s
    SysTick_DelayMs(2000);

    while (1){
        // 设置云台目标位姿
        Gimbal_SetYaw(servoUsart, 60, servoSpeed);
        Gimbal_SetPitch(servoUsart, 45, servoSpeed);
        // 等待云台旋转到目标位置
        Gimbal_Wait(servoUsart);

        // 延时1s
        SysTick_DelayMs(1000);

        // 设置云台目标位姿
        Gimbal_SetYaw(servoUsart, -60, servoSpeed);
        Gimbal_SetPitch(servoUsart, -45, servoSpeed);
        // 等待云台旋转到目标位置
        Gimbal_Wait(servoUsart);

        // 延时1s
        SysTick_DelayMs(1000);
    }
}
```

8.OpenMV色块识别

8.1.find_blobs函数

参考文档：寻找色块

find_blobs可以找到色块，它可以接受的参数很多

```
image.find_blobs(thresholds, roi=Auto, x_stride=2, y_stride=1, invert=False,
area_threshold=10, pixels_threshold=10, merge=False, margin=0, threshold_cb=None,
merge_cb=None)
```

- thresholds是颜色的阈值，注意：这个参数是一个列表，可以包含多个颜色。如果你只需要一个颜色，那么在这个列表中只需要有一个颜色值，如果你想要多个颜色阈值，那这个列表就需要多个颜色阈值。注意：在返回的色块对象blob可以调用code方法，来判断是什么颜色的色块。

  ```
  # 例如
  red = (xxx,xxx,xxx,xxx,xxx,xxx)
  blue = (xxx,xxx,xxx,xxx,xxx,xxx)
  yellow = (xxx,xxx,xxx,xxx,xxx,xxx)

  img=sensor.snapshot()
  red_blobs = img.find_blobs([red])

  color_blobs = img.find_blobs([red,blue, yellow])
  ```

- roi是"感兴趣区"，find_blobs将在这个划定区域内寻找色块

  ```
  left_roi = [0,0,160,240]
  blobs = img.find_blobs([red],roi=left_roi)
  ```

- x_stride 就是查找的色块的x方向上最小宽度的像素，默认为2
- y_stride 就是查找的色块的y方向上最小宽度的像素，默认为1
- invert 反转阈值，把阈值以外的颜色作为阈值进行查找
- area_threshold 面积阈值，如果色块被框起来的面积小于这个值，会被过滤掉
- pixels_threshold 像素个数阈值，如果色块像素数量小于这个值，会被过滤掉
- merge 合并重叠的blob，如果设置为True，那么合并所有重叠的blob为一个。
- margin 边界，如果设置为1，那么两个blobs如果间距1一个像素点，也会被合并。

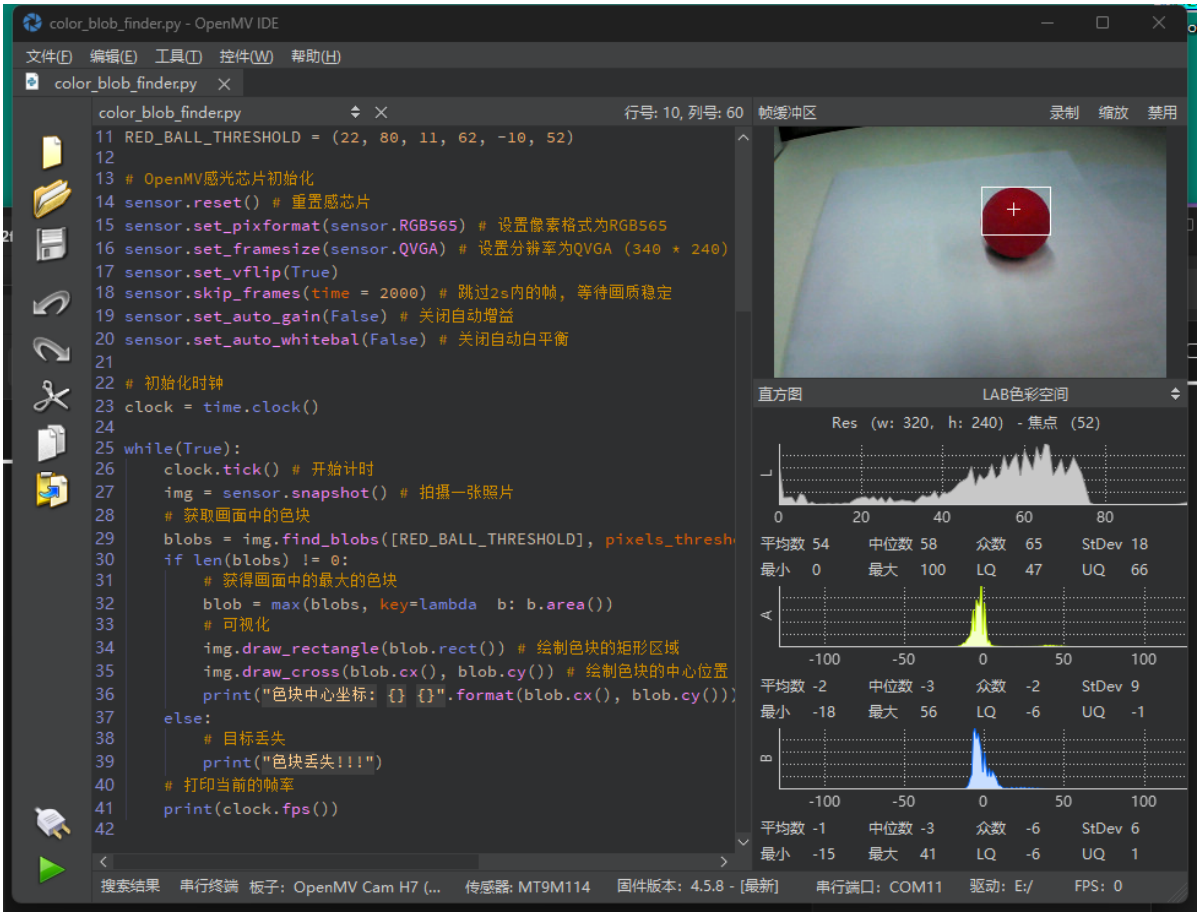
8.2.划定阈值

色块跟踪，首先要让OpenMV知道自己需要跟踪哪个色块。

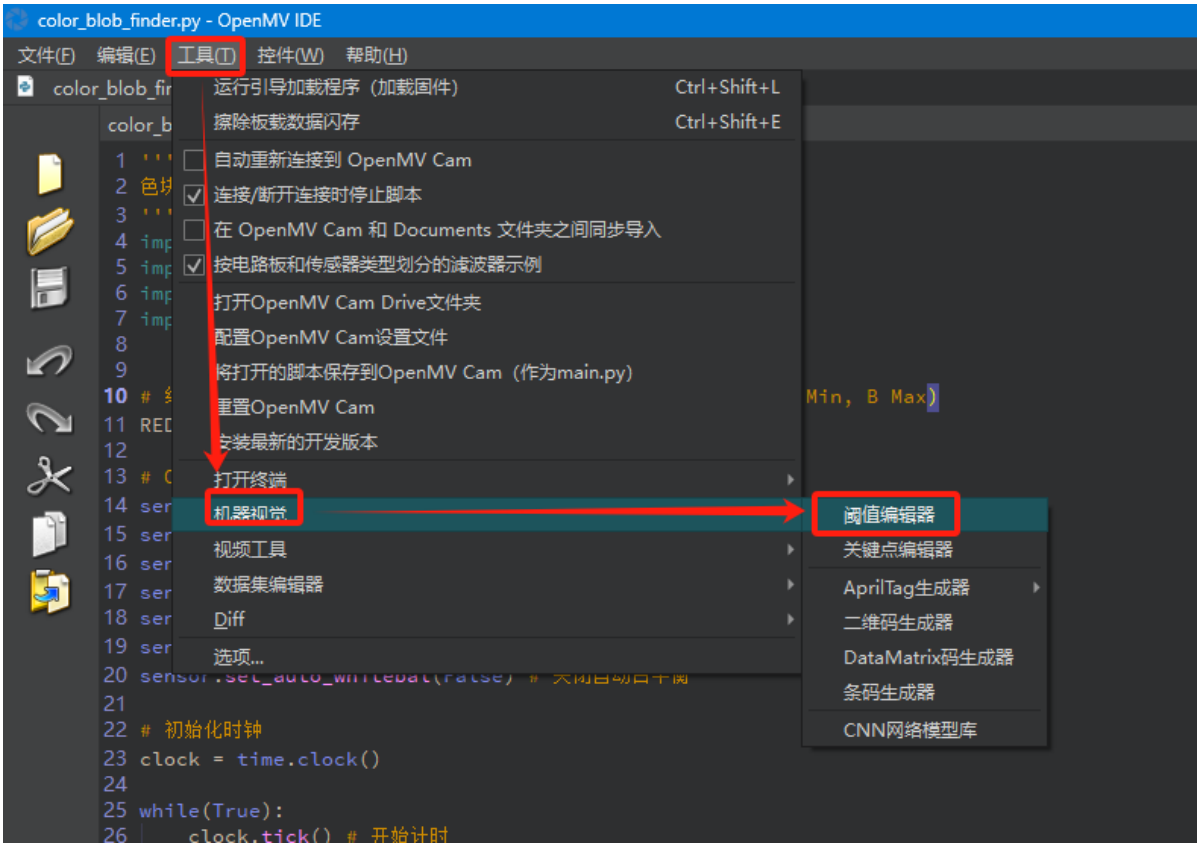
thresholds参数可以设置颜色的阈值，但获取这个阈值还需要一个小工具来协助。

颜色阈值调整

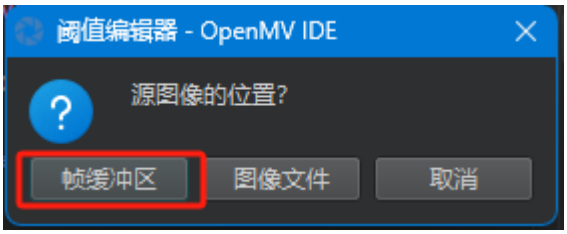
1.用数据线连接OpenMV板，使用OpenMV IDE打开 `OpenMV色块识别例程` ，点击运行，可以看到摄像头的识别效果。



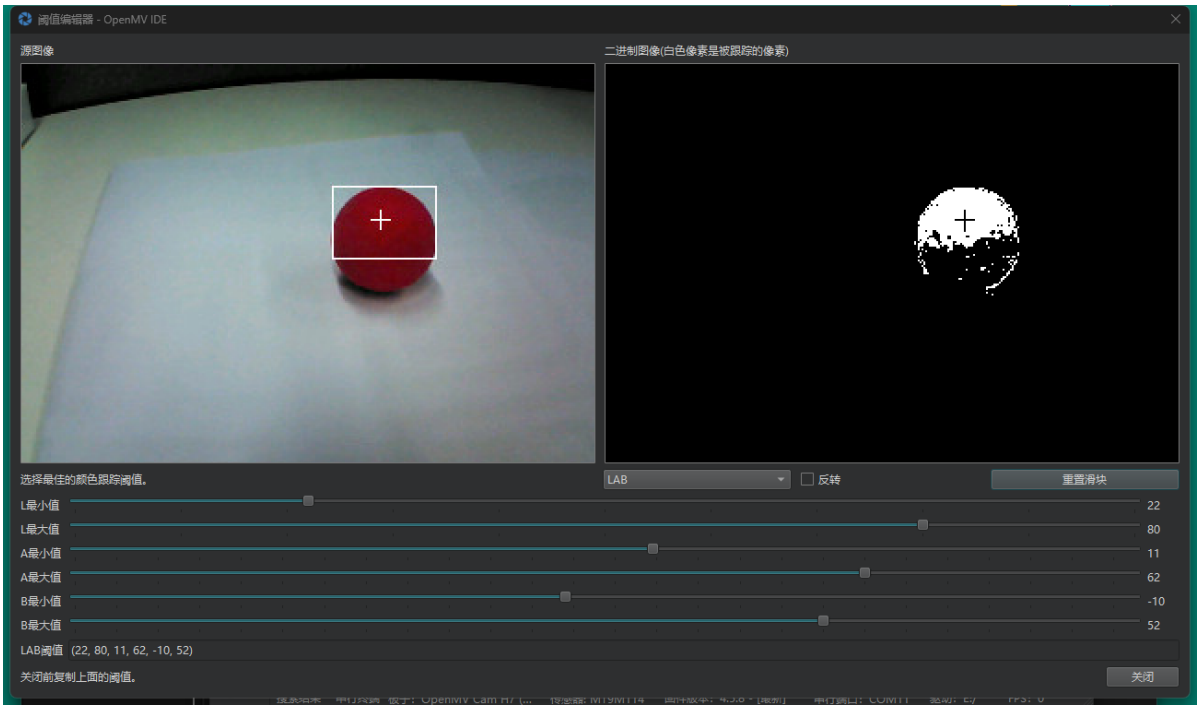
2.接下来打开阈值编辑器



3.选择帧缓冲区



4.右边图像中的白色部分是正在跟踪的像素，黑色是不在跟踪的像素。（此处白色方框和十字准星可以忽略，是程序在画面中绘制了识别像素区域）。

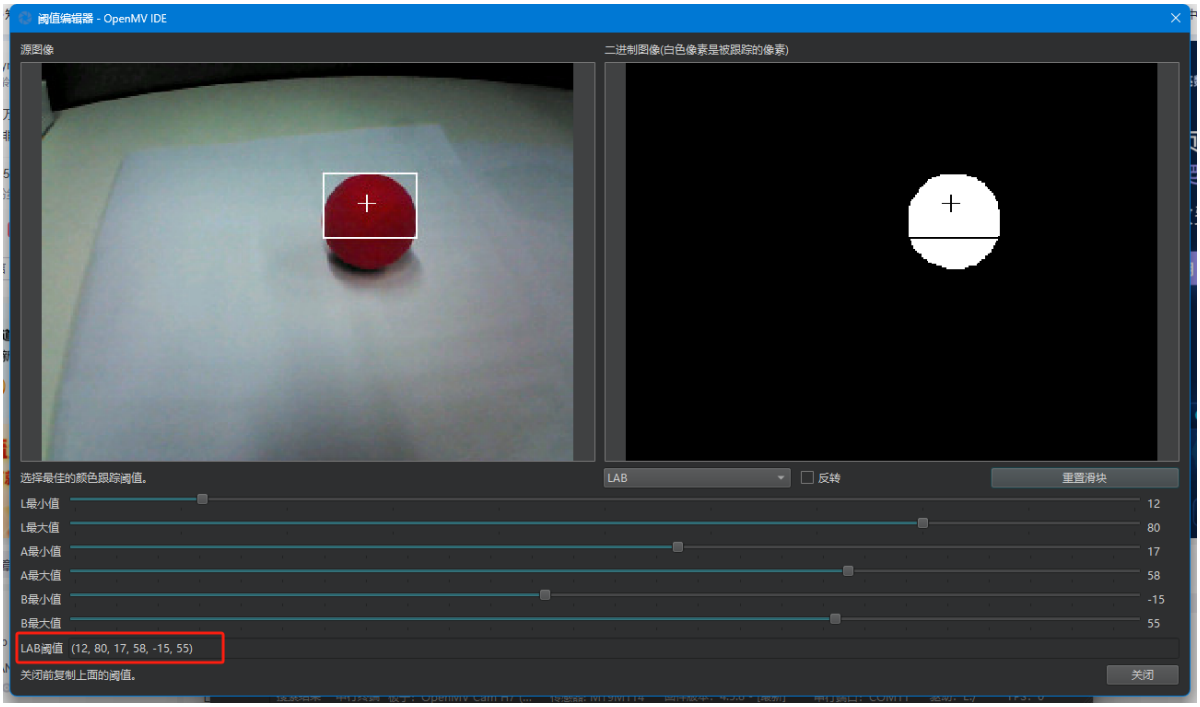


5.拖动6个条，调整LAB阈值，使得识别的小球尽可能完整，也即白色像素尽可能多且正确。

- L：亮度
- A：从绿色到红色的分量
- B：从蓝色到黄色的分量

6.现在识别阈值已经很好了，复制LAB阈值，关闭窗口，在代码对应的地方填入阈值

```
# 红色小球的LAB色彩空间阈值 (L Min, L Max, A Min, A Max, B Min, B Max)
RED_BALL_THRESHOLD = (12,80,17,58,-15,55)
```



7.保存.py文件，重新运行代码，现在已经能很好识别整个小球了。

8.3. OpenMV主程序源码 解析

导入模块

```
import sensor
import image
import time
import ustruct as struct
from pyb import LED
```

识别阈值，find_blobs的参数之一

```
# 红色小球的LAB色彩空间阈值 (L Min, L Max, A Min, A Max, B Min, B Max)
RED_BALL_THRESHOLD = (12,80,17,58,-15,55)
```

定义LED灯，方便看出识别状态，因为本文识别是红色，所以我们led也用红色灯光。

```
def led_control(x):
    if  x == 1:
        red_led.on()
        green_led.off()
        blue_led.off()
    elif  x == 2:
        red_led.off()
        green_led.on()
        blue_led.off()
    elif  x == 3:
        red_led.off()
        green_led.off()
        blue_led.on()
    else :
        red_led.off()
        green_led.off()
        blue_led.off()
```

OpenMV感光芯片初始化

```
# OpenMV感光芯片初始化
sensor.reset() # 重置感芯片
sensor.set_pixformat(sensor.RGB565) # 设置像素格式为RGB565
sensor.set_framesize(sensor.QVGA) # 设置分辨率为QVGA (340 * 240)
sensor.set_vflip(True) # 画面反转
sensor.skip_frames(time = 2000) # 跳过2s内的帧，等待画质稳定
sensor.set_auto_gain(False) # 关闭自动增益
sensor.set_auto_whitebal(False) # 关闭自动白平衡
```

初始化时钟作为帧率。

主循环里拍摄一次照片，然后对图像使用find_blobs方法，获取其中最大的色块，用白色方框和十字准星圈出。如果识别成功，会有红色LED灯亮起，否则没有。

```
# 初始化时钟
clock = time.clock()

while(True):
    clock.tick() # 开始计时
    img = sensor.snapshot() # 拍摄一张照片
    # 获取画面中的色块(选定阈值，像素数量大于100，色块面积大于100，合并重叠的blob)
    blobs = img.find_blobs([RED_BALL_THRESHOLD], pixels_threshold=100,
area_threshold=100, merge=True)
    if len(blobs) != 0:
        # 获得画面中的最大的色块
        blob = max(blobs, key=lambda  b: b.area())
        # 可视化
        led_control(1)
        img.draw_rectangle(blob.rect()) # 绘制色块的矩形区域
        img.draw_cross(blob.cx(), blob.cy()) # 绘制色块的中心位置
        print("色块中心坐标: {} {}".format(blob.cx(), blob.cy()))
    else:
        # 目标丢失
        led_control(0)
```

```
        print("色块丢失!!!")
    # 打印当前的帧率
    print(clock.fps())
```

9.STM32与OpenMV串口通信

参考视频教程：OpenMV视频教程-串口发送数据 ， OpenMV视频教程-串口接收数据

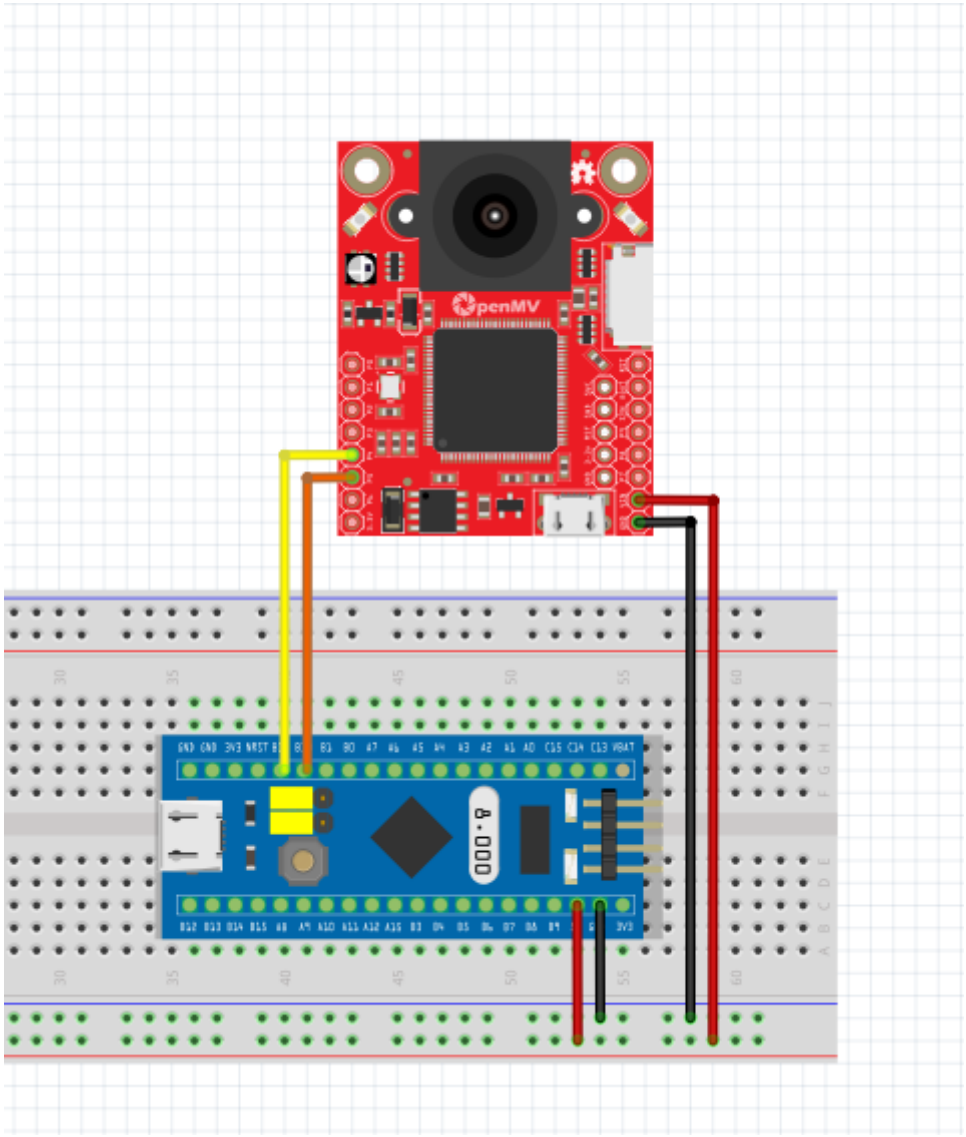
现在OpenMV的颜色识别已经完成了，但要实现跟踪效果，还需要有舵机支持，我们使用STM32F103控制舵机，其他主控也可以控制。

OpenMV需要将识别到的色块中心坐标发送给STM32主控，后者会使用pid算法实现跟随效果。

9.1.接线说明

（STM32 多合一主控板有OpenMV专用接口，用线接入即可）

STM32F103	**OpenMV3**
PB_11 (UART3 Rx)	P4 (PB10, UART3 Tx)
PB_10 (UART3 Tx)	P5 (PB11, UART3 Rx)
GND	GND
5V	VIN

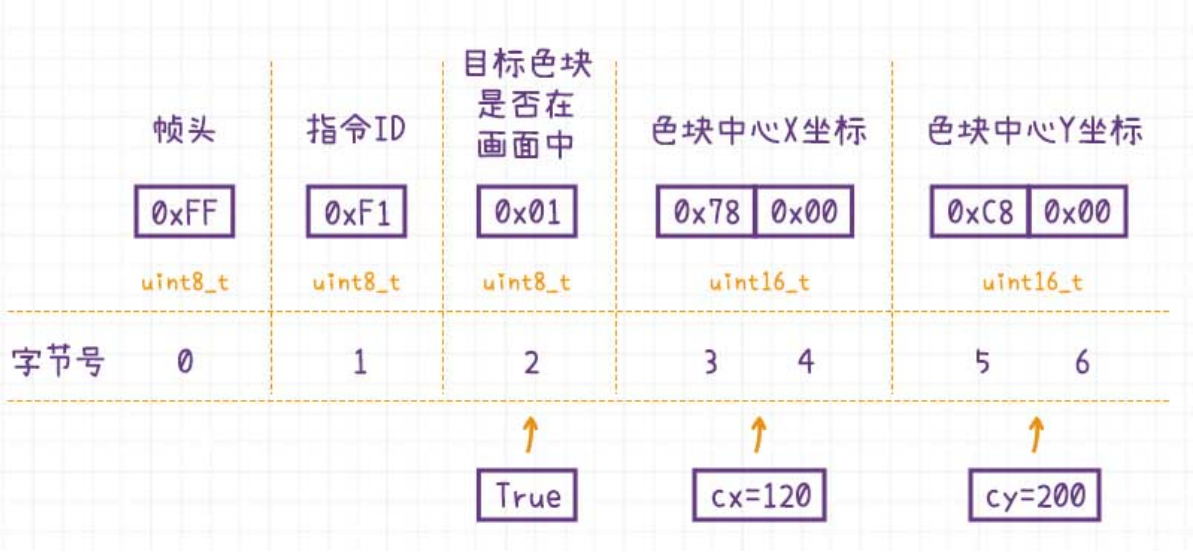


9.2.自定通信协议

接线完成后，我们还需要定义一套通信协议，让OpenMV和STM32遵循这个通信协议，以此交换信息。

字节序号	数据类型	字段名称	字段功能描述	字节长度
0	uint8_t	HEAD	帧头	1
1	uint8_t	CMD_ID	指令ID：F1	1
2	uint8_t	HAS_BLOB	画面中是否存在色块 1：存在 0：不存在	1
3-4	uint16_t	BLOB_CX	色块中心的x坐标	2
5-6	uint16_t	BLOB_CY	色块中心的y坐标	2

示例



9.3.OpenMV字节数据发送

OpenMV发送字节流，需要通过`ustruct`模块进行打包。代码比较简单，一行代码就可以搞定。

```
# 通过串口发送数据(二进制  低字节序)
uart.write(struct.pack('<BBBHH', 0xFF, 0xF1, True, blob.cx(), blob.cy()))
```

9.4.STM32字节数据解包

核心代码`updateBlobInfo`函数

```
// 更新色块的信息
void updateBlobInfo(Usart_DataTypeDef* blobUsart){
    uint8_t tempByte;
    while(RingBuffer_GetByteUsed(blobUsart->recvBuf)){
        // 弹出队首元素
        tempByte = RingBuffer_Pop(blobUsart->recvBuf);

        if (blobPkgIdx == 0 && tempByte != BLOB_PKG_HEADER){
            // 帧头还未接收且帧头不匹配
            continue;
        }else if(blobPkgIdx == 1 && tempByte != BLOB_PKG_CMD_ID){
            // 数据指令不匹配
            blobPkgIdx = 0;
            continue;
        }

        // 缓冲区内追加数据
        blobPkgBuf[blobPkgIdx] = tempByte;
        blobPkgIdx += 1;

        if (blobPkgIdx >= BLOB_PKG_LEN){
            // 数据接收完成，解析更新数据
            hasBlob = blobPkgBuf[2];
            blobCx =(uint16_t)(blobPkgBuf[3] | blobPkgBuf[4] << 8);
            blobCy = (uint16_t)(blobPkgBuf[5] | blobPkgBuf[6] << 8);

            blobPkgIdx = 0; // 游标清零
        }
```

```
    }
}
```

9.5.OpenMV主程序源码解析

程序在 OpenMV色块识别例程 上增加了串口发送代码，下面讲解串口发送代码，其他部分请参考 OpenMV色块识别例程 解析。

导入模块

```
import ustruct as struct
from pyb import UART
```

按照通信协议发送数据

```
# 通过串口发送数据(二进制  低字节序) 识别成功
uart.write(struct.pack('<BBBHH', 0xFF, 0xF1, True, blob.cx(), blob.cy()))
```

```
# 通过串口发送数据(二进制  低字节序) 没有识别到
uart.write(struct.pack('<BBBHH', 0xFF, 0xF1, False, 0, 0))
```

```
'''
串口通信
1. 识别画面中的红色小球
2. 将识别的到的色块位置,通过串口发送
'''
import sensor
import image
import time
import ustruct as struct
from pyb import UART

# 红色小球的LAB色彩空间阈值 (L Min, L Max, A Min, A Max, B Min, B Max)
RED_BALL_THRESHOLD = (57, 74, 38, 85, -21, 62)

# 串口初始化
uart = UART(3, 115200)

# OpenMV感光芯片初始化
sensor.reset() # 重置感芯片
sensor.set_pixformat(sensor.RGB565) # 设置像素格式为RGB565
sensor.set_framesize(sensor.QVGA) # 设置分辨率为QVGA (340 * 240)
sensor.set_vflip(True)
sensor.skip_frames(time = 2000) # 跳过2s内的帧, 等待画质稳定
sensor.set_auto_gain(False) # 关闭自动增益
sensor.set_auto_whitebal(False) # 关闭自动白平衡

# 初始化时钟
```

```
clock = time.clock()

while(True):
    clock.tick() # 开始计时
    img = sensor.snapshot() # 拍摄一张照片
    # 获取画面中的色块
    blobs = img.find_blobs([RED_BALL_THRESHOLD], pixels_threshold=100,
area_threshold=100, merge=True)
    if len(blobs) != 0:
        # 获得画面中的最大的色块
        blob = max(blobs, key=lambda  b: b.area())
        # 可视化
        img.draw_rectangle(blob.rect()) # 绘制色块的矩形区域
        img.draw_cross(blob.cx(), blob.cy()) # 绘制色块的中心位置
        print("色块中心坐标: {} {}".format(blob.cx(), blob.cy()))
        # 通过串口发送数据(二进制  低字节序)
        uart.write(struct.pack('<BBBHH', 0xFF, 0xF1, True, blob.cx(), blob.cy()))
    else:
        # 目标丢失
        print("色块丢失!!!")
        # 通过串口发送数据(二进制  低字节序)
        uart.write(struct.pack('<BBBHH', 0xFF, 0xF1, False, 0, 0))
    # 打印当前的帧率
    print(clock.fps())
```

9.6. STM32主程序源码解析

STM32f103程序使用串口3来接收来自OpenMV的通信数据，接收方式为串口中断，进入中断后会存储到一个循环数组队列里，并且在主循环读取该数组，对其进行解码，最后打印出来。

```
/*****************************************************
 * STM32与OpenMV串口通信
 *   OpenMV识别到色块之后，通过串口通信将色块的中心
 *   坐标发送给STM32，STM32解析串口的字节流数据，
 *   并将解析得到的色块坐标通过USB转TTL输出到串口调试助手上.
 *****************************************************/
#include "stm32f10x.h"
#include "usart.h"
#include "sys_tick.h"
#include "fashion_star_uart_servo.h"
#include "gimbal.h"


#define TRUE 1
#define FALSE 0

#define IMG_WIDTH 320 // blob画面分辨率 宽度
#define IMG_HEIGHT 240 // blob画面分辨率 高度
#define BLOB_PKG_LEN 7 // blob数据包的长度
#define BLOB_PKG_HEADER 0xFF // 帧头
#define BLOB_PKG_CMD_ID 0xF1 // 指令ID
```

```
// 使用串口1作为舵机控制的端口
// <接线说明>
// STM32F103 PA9(Tx)  <----> 串口舵机转接板 Rx
// STM32F103 PA10(Rx) <----> 串口舵机转接板 Tx
// STM32F103 GND      <----> 串口舵机转接板 GND
// STM32F103 V5       <----> 串口舵机转接板 5V
// <注意事项>
// 使用前确保已设置usart.h里面的USART1_ENABLE为1
// 设置完成之后，将下行取消注释
Usart_DataTypeDef* servoUsart = &usart1;
// 使用串口2作为日志输出的端口
// <接线说明>
// STM32F103 PA2(Tx) <----> USB转TTL Rx
// STM32F103 PA3(Rx) <----> USB转TTL Tx
// STM32F103 GND     <----> USB转TTL GND
// STM32F103 V5      <----> USB转TTL 5V (可选)
// <注意事项>
// 使用前确保已设置usart.h里面的USART2_ENABLE为1
Usart_DataTypeDef* loggingUsart = &usart2;
// 使用串口3接收来自blob的消息
// <接线说明>
// STM32F103 PB10(Tx) <----> blob P5 (UART3 Rx)
// STM32F103 PB11(Rx) <----> blob P4 (UART3 Tx)
// STM32F103 GND      <----> blob GND
// STM32F103 V5       <----> blob Vin (5v)
// <注意事项>
// 使用前确保已设置usart.h里面的USART3_ENABLE为1
Usart_DataTypeDef* blobUsart = &usart3;


uint8_t blobPkgBuf[BLOB_PKG_LEN]; // blob数据帧缓冲区
uint8_t blobPkgIdx = 0;

// 重定向c库函数printf到串口，重定向后可使用printf函数
int fputc(int ch, FILE *f)
{
    while((loggingUsart->pUSARTx->SR&0X40)==0){}
    /* 发送一个字节数据到串口 */
    USART_SendData(loggingUsart->pUSARTx, (uint8_t) ch);
    /* 等待发送完毕 */
    // while (USART_GetFlagStatus(USART1, USART_FLAG_TC) != SET);
    return (ch);
}

float servoSpeed = 100.0; // 云台旋转速度 (单位: °/s)

uint8_t hasBlob = FALSE; // 画面中有无色块
uint16_t blobCx = 0; // 色块中心的x坐标
uint16_t blobCy = 0; // 色块中心的y坐标

// 更新色块的信息
void updateBlobInfo(Usart_DataTypeDef* blobUsart){
    uint8_t tempByte;
    while(RingBuffer_GetByteUsed(blobUsart->recvBuf)){
        // 弹出队首元素
```

```
        tempByte = RingBuffer_Pop(blobUsart->recvBuf);

        if (blobPkgIdx == 0 && tempByte != BLOB_PKG_HEADER){
            // 帧头还未接收且帧头不匹配
            continue;
        }else if(blobPkgIdx == 1 && tempByte != BLOB_PKG_CMD_ID){
            // 数据指令不匹配
            blobPkgIdx = 0;
            continue;
        }

        // 缓冲区内追加数据
        blobPkgBuf[blobPkgIdx] = tempByte;
        blobPkgIdx += 1;

        if (blobPkgIdx >= BLOB_PKG_LEN){
            // 数据接收完成，解析更新数据
            hasBlob = blobPkgBuf[2];
            blobCx =(uint16_t)(blobPkgBuf[3] | blobPkgBuf[4] << 8);
            blobCy = (uint16_t)(blobPkgBuf[5] | blobPkgBuf[6] << 8);

            blobPkgIdx = 0; // 游标清零
        }
    }
}

int main (void)
{
    SysTick_Init();          // 嘀嗒定时器初始化
    Usart_Init();            // 串口初始化
    Gimbal_Init(servoUsart);// 云台初始化

    while(1){
        // 更新Buffer
        if (RingBuffer_GetByteUsed(blobUsart->recvBuf) >= BLOB_PKG_LEN) {
            updateBlobInfo(blobUsart);
        }
        // 打印日志
        printf("has blob: %d blob_cx: %d blob_cy: %d\r\n", hasBlob, blobCx,
blobCy);;
        // 延迟100ms
        SysTick_DelayMs(100);
    }
}
```

10.云台色块追踪

云台色块追踪项目是将 `OpenMV色块识别` ， `串口通信` 两个例程综合起来实践的项目。分为两个部分：

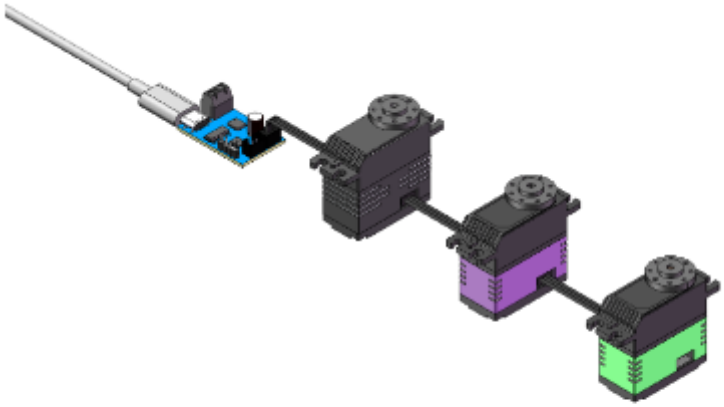
1. **视觉部分**：OpenMV识别到目标色块，并将物块的坐标通过串口通信协议发送给STM32。
2. **控制部分**：STM32负责控制舵机云台，更新舵机云台当前的角度。根据物块的坐标信息控制云台，控制物块保持在OpenMV画面的中心位置。

为了方便理解，在实现的时候先只考虑云台的其中一个轴，假设舵机云台下方的舵机是可以旋转的，而上方的舵机保持静止。

10.1.接线说明

可以参考 总线伺服舵机SDK使用手册（STM32F103）以及 连线方式及电源解决方案 文章

- 舵机和UC01的连线（推荐串联，可以保证云台走线的简洁）

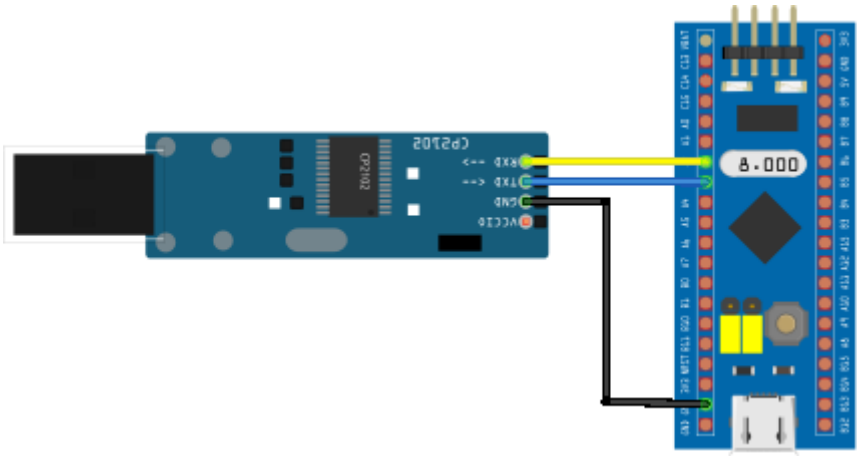


串联

- UC01和STM32F103主控的连线（STM32 多合一主控板有舵机专用接口，无须经由UC01）

STM32F103 GPIO	**串口舵机转接板UC01**
PA_9 （TX）	RX
PA_10 （RX）	TX
5V（可选）	5V（可选）
GND	GND

- STM32F103主控和USB转TTL模块的连线

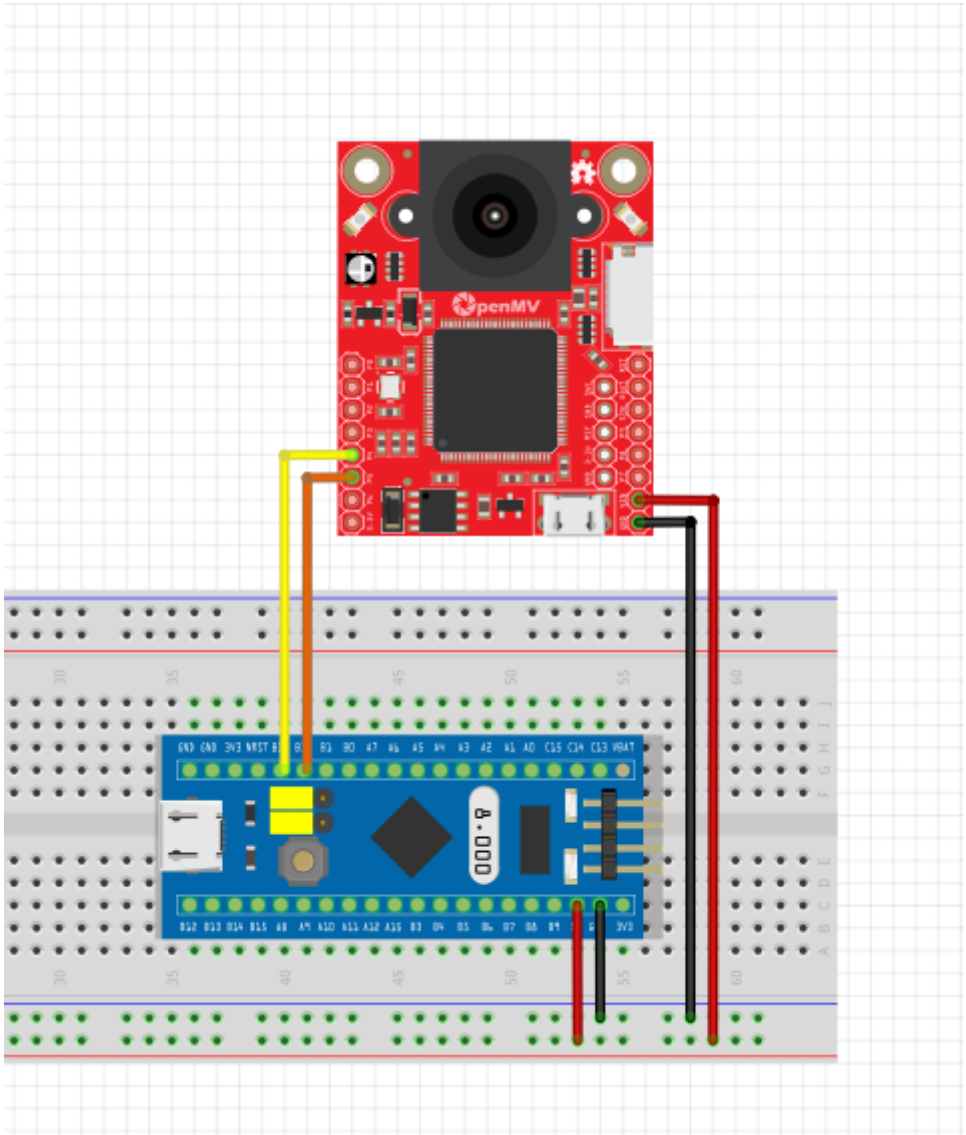


USB转TTL模块	**STM32F103 GPIO**
RX	PA2（TX）
TX	PA3（RX）
5V（可选）	5V（可选）
GND	GND

USB转TTL模块	STM32F103 GPIO
GND	GND

（STM32 多合一主控板有OpenMV专用接口，用线接入即可）

STM32F103	OpenMV3
PB_11 (UART3 Rx)	P4 (PB10, UART3 Tx)
PB_10 (UART3 Tx)	P5 (PB11, UART3 Rx)
GND	GND
5V	VIN



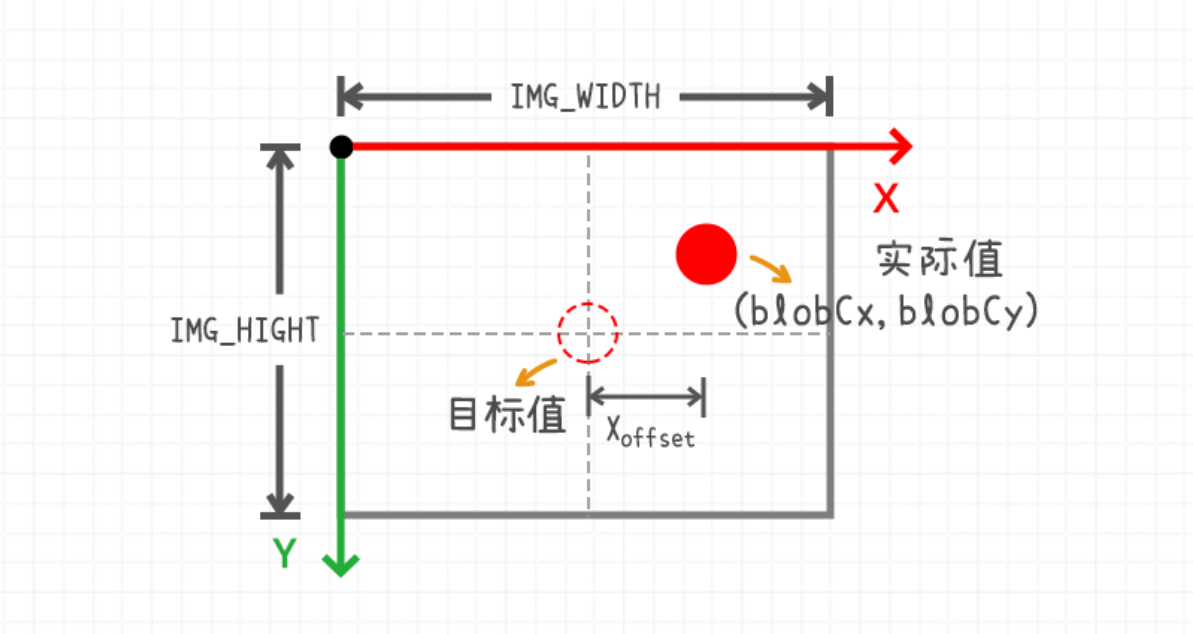
10.2.PID控制原理

PID控制全称为比例-积分-微分控制器，是自控原理里面最经典、使用最广泛的算法。

在云台色块追踪项目里，它承担了将识别到的偏移量转换为云台转动角度目标的功能。

缩写	名称	英文
P	比例	Proportion
I	积分	Integral
D	微分	Derivative

OpenMV图像坐标



OpenMV画面的宽度为 `IMG_WIDTH` ，高度为 `IMG_HEIGHT` ，单位为像素。

假设此时小球中心的坐标为 `(blobCx, blobCy)` ，中心坐标为 `（IMG_WIDTH,IMG_HIGHT）` 。

目标值（target）

是小球目标的位置，小球x坐标的目标值是图像中心处。

实际值（real）

是小球当前在画面中的位置，有时也被称为 测量值（measurement）。

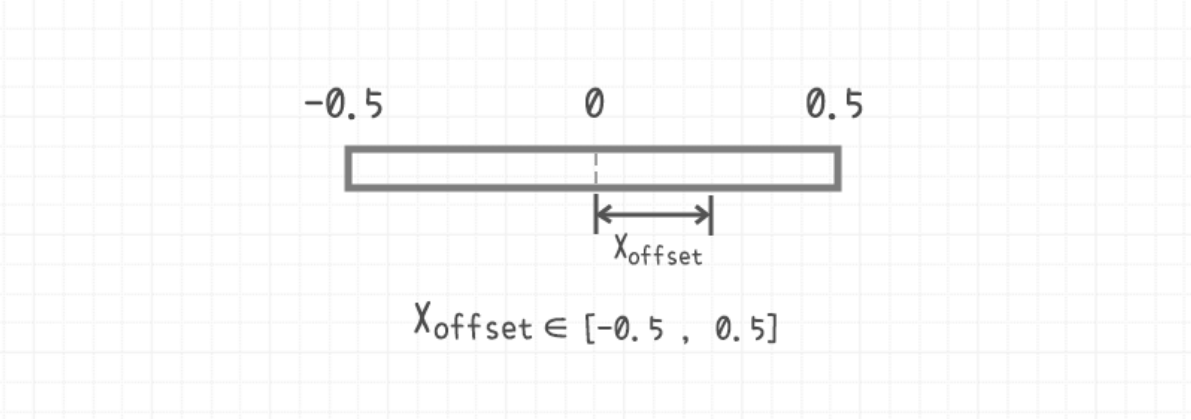
偏移量（offset）

是实际值与目标值之间的误差，偏移量有时候也被称为 误差（error），记做$x_{offset} $。

为了方便后续PID的计算不受图像分辨率的影响，一般需要对测量值/偏移量进行归一化操作。

归一化之后，偏移量的取值范围为

$$x_{offset} \in [-0.5, 0.5]$$

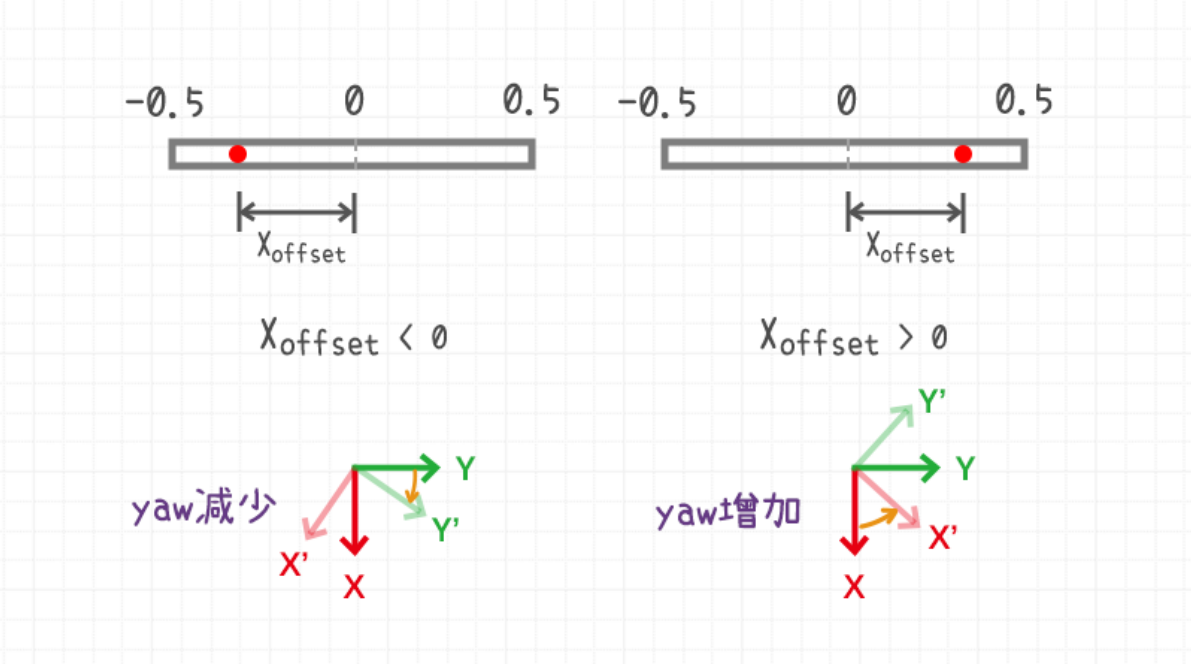



```
// 计算当前的偏移量
curXOffset =((blobCx / IMG_WIDTH) - 0.5);
```

首先，我们需要根据偏移量x_{offset}的正负判断舵机云台该向那个方向移动。

$x_{offset} > 0$ 时，舵机应该向云台偏航角增加的方向旋转。

$x_{offset} < 0$ 时，舵机应该向云台偏航角减少的方向旋转。



比例系数K_p

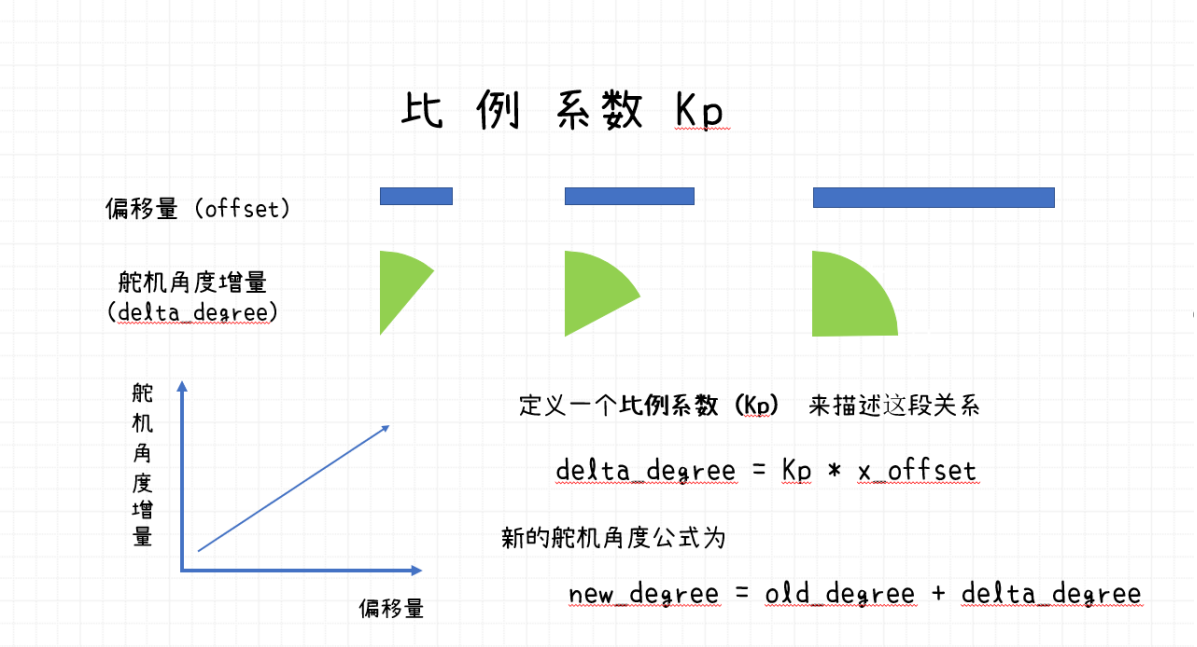
在获得舵机下一个控制周期的旋转方向之后，还需要知道旋转的**幅度**。

偏移量x_{offset} 绝对值越大，舵机旋转的幅度也就应该越大，线性关系是描述这种关系的其中一种表示方法。

这个线性关系的斜率即为PID控制器里面的比例系数K_p 。

$$dyaw = K_p * x_{offset}$$

$dyaw$ 是偏航角的增量。



当K_p 过小时，舵机会出现动作迟缓，容易丢失目标的问题。

当Kp 过大时，舵机就会出现**震荡**的问题，舵机频繁的来回摆动。

微分系数K_d

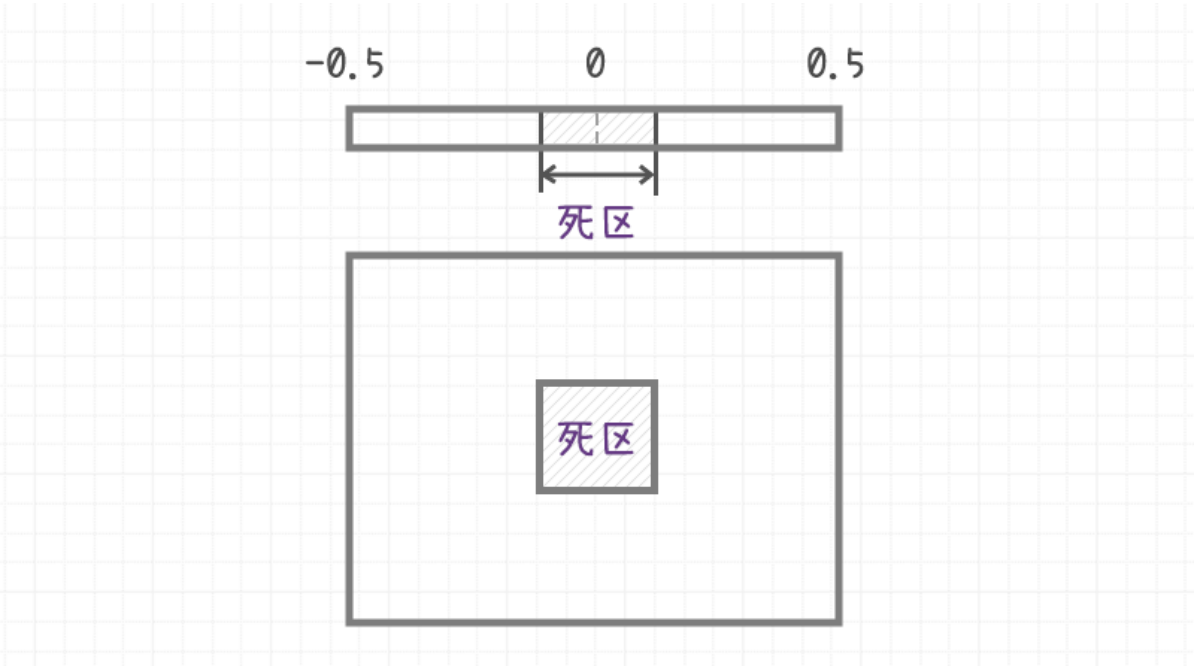
所谓微分指的就是$x{*offset*}$ *的变化率，旧的*$x{offset}$ 减去新的x_{offset} 。

$$dx_{offset} = old\_x_{offset} - x_{offset}$$

$$dyaw = K_p * x_{offset} + K_d * dx_{offset}$$

微分系数也叫阻尼系数，Kd 的作用是给舵机角度变换添加一定的阻力，抑制舵机角度震荡。

死区 Deadblock



为了防止舵机在目标值附近的震荡，会设置死区。

```
#define DEAD_BLOCK 0.05
```

当偏移量x_{offset} 的绝对值小于死区时，将偏移量置为0。

```
// 判断是否在死区内
if (__fabs(curXOffset) <= DEAD_BLOCK){
    curXOffset = 0;
}
```

10.3.云台PID参数调节、速度设置

```
// 云台偏航角PID控制
#define DEAD_BLOCK 0.05
#define GIMBAL_YAW_KP 30.0
#define GIMBAL_YAW_KD 1.0
```

以上3个宏定义，对应死区，Kp和Kd

1. 初始调节时，将`GIMBAL_YAW_KD`设置为0
2. 将逐渐增加比例系数`GIMBAL_YAW_KP`，直到舵机云台发生明显的抖动
3. 此时调节`GIMBAL_YAW_KD`（阻尼力）让舵机云台变稳定。

Pitch稍小的PID控制也是同理，但是一般来讲，设置Kp和Kd会比Yaw轴稍小。

这两轴的PID调节，只要能满足需求即可，不必追求极限参数。

速度设置在`GimbalPitchPIDCtl`和`GimbalYawPIDCtl()`里`servoSpeed`对应速度。

```
// 云台偏航角控制
    Gimbal_SetYaw(servoUsart, curYaw + dYaw, servoSpeed);
```

```
// 云台偏航角 PID控制
void GimbalYawPIDCtl(){
    float dYaw; // 偏航角的增量

    // 更新上一次的偏移量
    lastXOffset = curXOffset;
    // 计算当前的偏移量
    curXOffset =((blobCx / IMG_WIDTH) - 0.5);
    // 判断是否在死区内
    if (__fabs(curXOffset) <= DEAD_BLOCK){
        curXOffset = 0;
    }

    // 计算得到偏航角的增量
    dYaw = GIMBAL_YAW_KP * curXOffset + GIMBAL_YAW_KD * (lastXOffset -
curXOffset);
```

```
    // 云台偏航角控制
    Gimbal_SetYaw(servoUsart, curYaw + dYaw, servoSpeed);

    printf("YAW PID: dYaw = %.1f next yaw = %.1f\r\n", dYaw, curYaw + dYaw);
}
```

舵机PID控制的核心算法C语言实现

```
// 云台偏航角PID控制
void GimbalYawPIDCtl(){
    float dYaw; // 偏航角的增量

    // 更新上一次的偏移量
    lastXOffset = curXOffset;
    // 计算当前的偏移量
    curXOffset =((blobCx / IMG_WIDTH) - 0.5);
    // 判断是否在死区内
    if (__fabs(curXOffset) <= DEAD_BLOCK){
        curXOffset = 0;
    }

    // 计算得到偏航角的增量
    dYaw = GIMBAL_YAW_KP * curXOffset + GIMBAL_YAW_KD * (lastXOffset -
curXOffset);
    // 云台偏航角控制
    Gimbal_SetYaw(servoUsart, curYaw + dYaw, servoSpeed);

    printf("YAW PID: dYaw = %.1f next yaw = %.1f\r\n", dYaw, curYaw + dYaw);
}
```

10.4. OpenMV主程序源码 解析

```
'''
色块识别
1. 识别画面中的红色小球
2. 将识别的到的色块位置,通过串口发送
'''
import sensor
import image
import time
import ustruct as struct
from pyb import UART

# 红色小球的LAB色彩空间阈值 (L Min, L Max, A Min, A Max, B Min, B Max)
RED_BALL_THRESHOLD = (57, 74, 38, 85, -21, 62)

# 串口初始化
uart = UART(3, 115200)

# OpenMV感光芯片初始化
```

```
sensor.reset() # 重置感芯片
sensor.set_pixformat(sensor.RGB565) # 设置像素格式为RGB565
sensor.set_framesize(sensor.QVGA) # 设置分辨率为QVGA (340 * 240)
sensor.set_vflip(True)
sensor.skip_frames(time = 2000) # 跳过2s内的帧, 等待画质稳定
sensor.set_auto_gain(False) # 关闭自动增益
sensor.set_auto_whitebal(False) # 关闭自动白平衡

# 初始化时钟
clock = time.clock()

while(True):
    clock.tick() # 开始计时
    img = sensor.snapshot() # 拍摄一张照片
    # 获取画面中的色块
    blobs = img.find_blobs([RED_BALL_THRESHOLD], pixels_threshold=100,
area_threshold=100, merge=True)
    if len(blobs) != 0:
        # 获得画面中的最大的色块
        blob = max(blobs, key=lambda  b: b.area())
        # 可视化
        img.draw_rectangle(blob.rect()) # 绘制色块的矩形区域
        img.draw_cross(blob.cx(), blob.cy()) # 绘制色块的中心位置
        print("色块中心坐标: {} {}".format(blob.cx(), blob.cy()))
        # 通过串口发送数据(二进制  低字节序)
        uart.write(struct.pack('<BBBHH', 0xFF, 0xF1, True, blob.cx(), blob.cy()))
    else:
        # 目标丢失
        print("色块丢失!!!")
        # 通过串口发送数据(二进制  低字节序)
        uart.write(struct.pack('<BBBHH', 0xFF, 0xF1, False, 0, 0))
    # 打印当前的帧率
    print(clock.fps())
```

10.5. STM32主程序源码 解析

画面分辨率需要通过宏定义设置与OpenMV一致

```
#define IMG_WIDTH 320.0 // blob画面分辨率 宽度
#define IMG_HEIGHT 240.0 // blob画面分辨率 高度
```

主循环里使用PID控制云台运动。

```
/**************************************************
 * 云台色块追踪-双轴单级PID控制
 **************************************************/
#include "stm32f10x.h"
```

```c
#include "usart.h"
#include "sys_tick.h"
#include "fashion_star_uart_servo.h"
#include "gimbal.h"

#define TRUE 1
#define FALSE 0

#define IMG_WIDTH 320.0 // blob画面分辨率 宽度
#define IMG_HEIGHT 240.0 // blob画面分辨率 高度
#define BLOB_PKG_LEN 7 // blob数据包的长度
#define BLOB_PKG_HEADER 0xFF // 帧头
#define BLOB_PKG_CMD_ID 0xF1 // 指令ID

// 云台偏航角PID控制
#define DEAD_BLOCK 0.05
#define GIMBAL_YAW_KP 50.0
#define GIMBAL_YAW_KD 2.0
// 云台俯仰角PID控制
#define GIMBAL_PITCH_KP 40.0
#define GIMBAL_PITCH_KD 3.0

// 使用串口1作为舵机控制的端口
// <接线说明>
// STM32F103 PA9(Tx)  <----> 串口舵机转接板 Rx
// STM32F103 PA10(Rx) <----> 串口舵机转接板 Tx
// STM32F103 GND      <----> 串口舵机转接板 GND
// STM32F103 V5       <----> 串口舵机转接板 5V
// <注意事项>
// 使用前确保已设置usart.h里面的USART1_ENABLE为1
// 设置完成之后，将下行取消注释
Usart_DataTypeDef* servoUsart = &usart1;
// 使用串口2作为日志输出的端口
// <接线说明>
// STM32F103 PA2(Tx) <----> USB转TTL Rx
// STM32F103 PA3(Rx) <----> USB转TTL Tx
// STM32F103 GND     <----> USB转TTL GND
// STM32F103 V5      <----> USB转TTL 5V (可选)
// <注意事项>
// 使用前确保已设置usart.h里面的USART2_ENABLE为1
Usart_DataTypeDef* loggingUsart = &usart2;
// 重定向c库函数printf到串口，重定向后可使用printf函数
int fputc(int ch, FILE *f)
{
    while((loggingUsart->pUSARTx->SR&0X40)==0){}
    /* 发送一个字节数据到串口 */
    USART_SendData(loggingUsart->pUSARTx, (uint8_t) ch);
    /* 等待发送完毕 */
    // while (USART_GetFlagStatus(USART1, USART_FLAG_TC) != SET);
    return (ch);
}


// 使用串口3接收来自blob的消息
// <接线说明>
// STM32F103 PB10(Tx) <----> blob P5 (UART3 Rx)
```

```
// STM32F103 PB11(Rx) <----> blob P4 (UART3 Tx)
// STM32F103 GND      <----> blob GND
// STM32F103 V5       <----> blob Vin (5v)
// <注意事项>
// 使用前确保已设置usart.h里面的USART3_ENABLE为1
Usart_DataTypeDef* blobUsart = &usart3;


uint8_t blobPkgBuf[BLOB_PKG_LEN]; // blob数据帧缓冲区
uint8_t blobPkgIdx = 0;

float servoSpeed = 400.0; // 云台旋转速度 (单位: °/s)
uint8_t hasBlob = FALSE; // 画面中有无色块
uint16_t blobCx = 0; // 色块中心的x坐标
uint16_t blobCy = 0; // 色块中心的y坐标

float curXOffset = 0; // 当前x轴方向上的偏移量
float lastXOffset = 0; // 上一次x轴方向上的偏移量

float curYOffset = 0; // 当前y轴方向上的偏移量
float lastYOffset = 0; // 上一次y轴方向上的偏移量

// 更新色块的信息
void updateBlobInfo(Usart_DataTypeDef* blobUsart){
    uint8_t tempByte;
    while(RingBuffer_GetByteUsed(blobUsart->recvBuf)){
        // 弹出队首元素
        tempByte = RingBuffer_Pop(blobUsart->recvBuf);

        if (blobPkgIdx == 0 && tempByte != BLOB_PKG_HEADER){
            // 帧头还未接收且帧头不匹配
            continue;
        }else if(blobPkgIdx == 1 && tempByte != BLOB_PKG_CMD_ID){
            // 数据指令不匹配
            blobPkgIdx = 0;
            continue;
        }

        // 缓冲区内追加数据
        blobPkgBuf[blobPkgIdx] = tempByte;
        blobPkgIdx += 1;

        if (blobPkgIdx >= BLOB_PKG_LEN){
            // 数据接收完成, 解析更新数据
            hasBlob = blobPkgBuf[2];
            // 高通滤波, 抵抗图像处理噪声
            blobCx = 0.3*blobCx + 0.7*(uint16_t)(blobPkgBuf[3] | blobPkgBuf[4] <<
8);
            blobCy = 0.3*blobCy + 0.7*(uint16_t)(blobPkgBuf[5] | blobPkgBuf[6] <<
8);

            blobPkgIdx = 0; // 游标清零
        }
    }
}
```

```
// 云台偏航角 PID控制
void GimbalYawPIDCtl(){
    float dYaw; // 偏航角的增量

    // 更新上一次的偏移量
    lastXOffset = curXOffset;
    // 计算当前的偏移量
    curXOffset =((blobCx / IMG_WIDTH) - 0.5);
    // 判断是否在死区内
    if (__fabs(curXOffset) <= DEAD_BLOCK){
        curXOffset = 0;
    }

    // 计算得到偏航角的增量
    dYaw = GIMBAL_YAW_KP * curXOffset + GIMBAL_YAW_KD * (lastXOffset -
curXOffset);
    // 云台偏航角控制
    Gimbal_SetYaw(servoUsart, curYaw + dYaw, servoSpeed);

    printf("YAW PID: dYaw = %.1f next yaw = %.1f\r\n", dYaw, curYaw + dYaw);
}


// 云台俯仰角 PID控制
void GimbalPitchPIDCtl(){
    float dPitch; // 俯仰角的增量

    // 更新上一次的偏移量
    lastYOffset = curYOffset;
    // 计算当前的偏移量
    curYOffset =((blobCy / IMG_HEIGHT) - 0.5);
    // 判断是否在死区内
    if (__fabs(curYOffset) <= DEAD_BLOCK){
        curYOffset = 0;
    }

    // 计算得到偏航角的增量
    dPitch = GIMBAL_PITCH_KP * curYOffset + GIMBAL_PITCH_KD * (lastYOffset -
curYOffset);
    // 云台偏航角控制
    Gimbal_SetPitch(servoUsart, curPitch + dPitch, 500);

    printf("YAW PID: dPitch = %.1f next pitch = %.1f\r\n", dPitch, curPitch +
dPitch);
}



int main (void)
{
    SysTick_Init();         // 嘀嗒定时器初始化
    Usart_Init();           // 串口初始化
    Gimbal_Init(servoUsart);// 云台初始化

    while(1){
        // 更新色块的位置信息
```

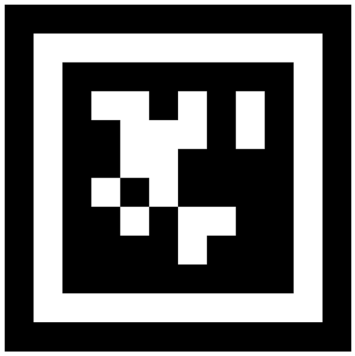
```
        if (RingBuffer_GetByteUsed(blobUsart->recvBuf) >= BLOB_PKG_LEN) {
            updateBlobInfo(blobUsart);
        }
        // 更云台位姿
        Gimbal_Update(servoUsart);

        if (hasBlob){
            GimbalYawPIDCtl();
            GimbalPitchPIDCtl();
        }else{
            // TODO 拓展部分-添加色块丢失的处理策略
        }
        // 延时10ms
        // SysTick_DelayMs(10);

    }
}
```

11.OpenMV AprilTag标记

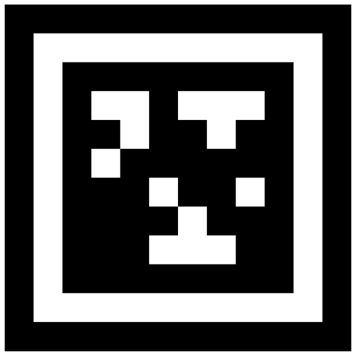
参考文档：AprilTag原理简介及原代码，OpenMV AprilTag标记跟踪

AprilTag 是一个视觉基准库，在AR，机器人等领域广泛使用，通过类似二维码的标记，可以快速地检测标志并且计算出其位姿。

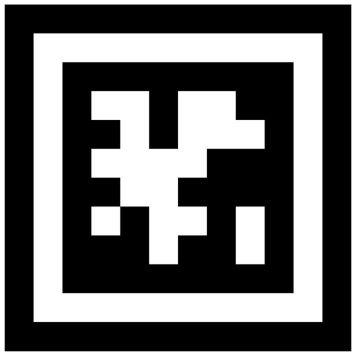
我们主要使用TAG36H11这一系列标记，它的校验信息很多，因此识别出错率很低。



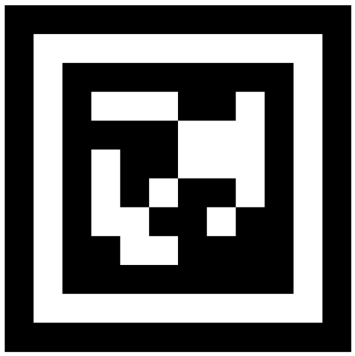
TAG36H11 - 0



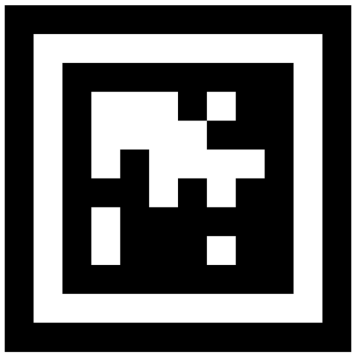
TAG36H11 - 2



TAG36H11 - 1



TAG36H11 - 3

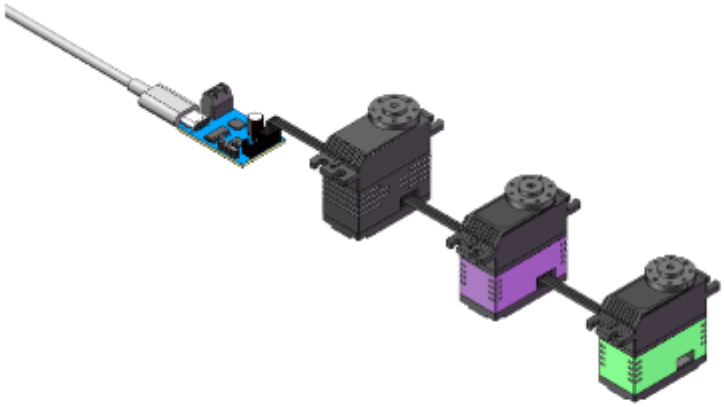


TAG36H11 - 4

11.1.接线说明

可以参考 总线伺服舵机SDK使用手册（STM32F103）以及 连线方式及电源解决方案 文章

- 舵机和UC01的连线（推荐串联，可以保证云台走线的简洁）

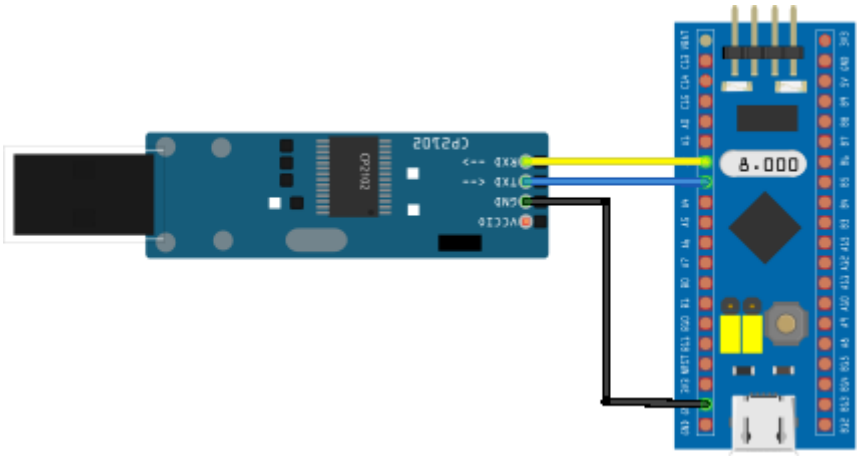


串联

- UC01和STM32F103主控的连线（STM32 多合一主控板有舵机专用接口，无须经由UC01）

STM32F103 GPIO	**串口舵机转接板UC01**
PA_9 （TX）	RX
PA_10 （RX）	TX
5V（可选）	5V（可选）
GND	GND

- STM32F103主控和USB转TTL模块的连线

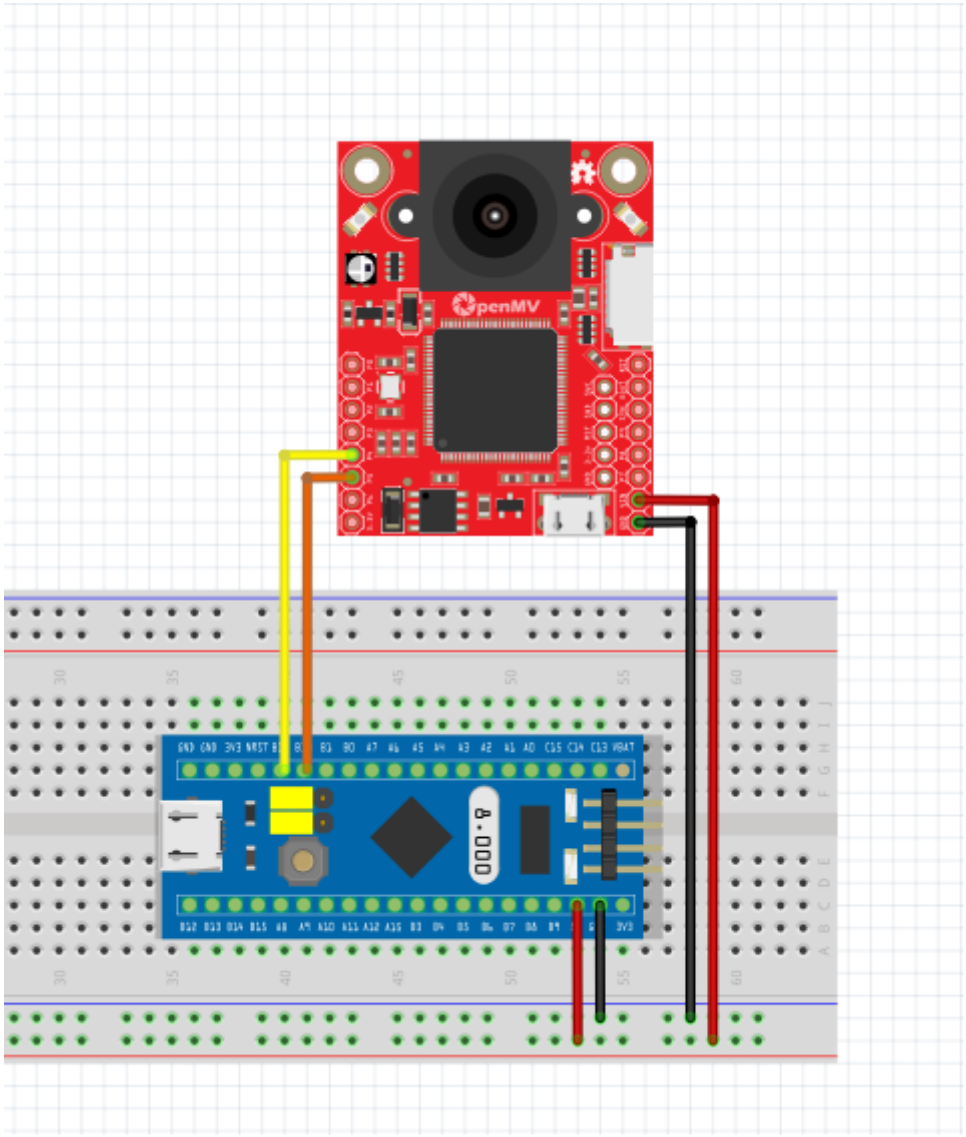


USB转TTL模块	**STM32F103 GPIO**
RX	PA2（TX）
TX	PA3（RX）
5V（可选）	5V（可选）

USB转TTL模块	STM32F103 GPIO
GND	GND

（STM32 多合一主控板有OpenMV专用接口，用线接入即可）

STM32F103	OpenMV3
PB_11 (UART3 Rx)	P4 (PB10, UART3 Tx)
PB_10 (UART3 Tx)	P5 (PB11, UART3 Rx)
GND	GND
5V	VIN



11.2.AprilTag 步骤

AprilTag 主要包含以下3个步骤

1. 获取一张图片，根据梯度检测出图像边缘。
2. 在图像边缘中使用图像处理筛选出类似四边形的图案。
3. 对筛选出的四边形图案识别，解码，根据库内的编码进行匹配，得到校验信息通过的图案。

对于 OpenMV ，由于其内置了AprilTag库，我们可以不用自己动手处理图像，仅了解原理，会如何使用库即可。

11.3.AprilTag标记应用

AprilTag标记由于拥有的信息量比色块更多，因此他能够给我们带来的应用也更加丰富，下面我们编写3个小功能。

我们使用OpenMV识别 TAG36H11 系列标记0~2号，识别成功后，OpenMV将获取的信息发送给STM32，接着控制舵机云台实现不同的功能。

- 识别到0号标记时，云台Pitch轴和Yaw轴都转动回到0点。
- 识别到1号标记时，云台进入巡视模式。
- 识别到2号标记时，云台跟随标记，进入标记跟随模式。

11.4.自定通信协议

我们在原有协议上，在指令ID处进行升级。因为需要使用到AprilTag标记的ID对应不同的命令，我们使用A0,A1,A2来记录识别的标记。

字节序号	数据类型	字段名称	字段功能描述	字节长度
0	uint8_t	HEAD	帧头	1
1	uint8_t	CMD_ID	指令ID A0：识别到0号标记 A1：识别到1号标记 A2：识别到2号标记 F1：没识别到标记	1
2	uint8_t	HAS_BLOB	画面中是否存在标记 1：存在 0：不存在	1
3-4	uint16_t	BLOB_CX	标记中心的x坐标	2
5-6	uint16_t	BLOB_CY	标记中心的y坐标	2

11.5.LED提示

当OpenMV脱离PC独自运行的时候，我们无法直观地观察其运行状态。因此引入LED灯作为提示

- 当识别到0号标记时，LED切换成红色
- 当识别到1号标记时，LED切换成绿色
- 当识别到2号标记时，LED切换成蓝色

LED对应的代码也很简单

1. 导入LED模块

```
from pyb import LED
```

2. 初始化led对象

```
# 设置led
red_led  = LED(1)
green_led = LED(2)
blue_led  = LED(3)
```

3. 定义控制函数

```
# 定义led控制，0:红灯，1：绿灯，2：蓝灯
def led_control(x):
    if  x == 0:
        red_led.on()
        green_led.off()
        blue_led.off()
    elif  x == 1:
        red_led.off()
        green_led.on()
        blue_led.off()
    elif  x == 2:
        red_led.off()
        green_led.off()
        blue_led.on()
    else :
        red_led.off()
        green_led.off()
        blue_led.off()
```

4. 点亮led灯

```
led_control(0)
```

11.6.OpenMV主程序源码解析

```
'''
AprilTag识别
'''
```

```
import sensor, image, time, math
import ustruct as struct
from pyb import UART
from pyb import LED

# 串口3 速率115200
uart = UART(3, 115200)
# 设置led
red_led   = LED(1)
green_led = LED(2)
blue_led  = LED(3)
# 定义led控制，0:红灯，1：绿灯，2：蓝灯
def led_control(x):
    if  x == 0:
        red_led.on()
        green_led.off()
        blue_led.off()
    elif  x == 1:
        red_led.off()
        green_led.on()
        blue_led.off()
    elif  x == 2:
        red_led.off()
        green_led.off()
        blue_led.on()
    else :
        red_led.off()
        green_led.off()
        blue_led.off()

# OpenMV感光芯片初始化
sensor.reset()
sensor.set_pixformat(sensor.RGB565)# 设置像素格式为RGB565
sensor.set_framesize(sensor.QQVGA) # 设置分辨率为QQVGA (160 * 120)
sensor.skip_frames(30) # 跳过30ms内的帧，等待画质稳定
sensor.set_auto_gain(False)   # 关闭自动增益
sensor.set_auto_whitebal(False)  # 关闭自动白平衡
sensor.set_vflip(True) # 纵向翻转

# 初始化时钟
clock = time.clock()


def degrees(radians):
    return (180 * radians) / math.pi

while(True):
    clock.tick()# 开始计时
    img = sensor.snapshot()# 拍摄一张照片
    # 获取画面中的标记()
    tags = img.find_apriltags()

    if tags:
        # 找到至少一个 AprilTag
        for tag in tags:
            # 可视化
```

```
            img.draw_rectangle(tag.rect(), color = (255, 0, 0))
            img.draw_cross(tag.cx(), tag.cy(), color = (0, 255, 0))
            # 找到TAG36H11标签
            if tag.family() == image.TAG36H11:
                # 识别到id为0
                # 红灯亮
                # 通过串口发送数据(二进制  低字节序)
                if tag.id() == 0:
                    led_control(0)
                    uart.write(struct.pack('<BBBHH', 0xFF,0xA0 , True, tag.cx(),
tag.cy()))
                # 识别到id为1
                # 绿灯亮
                # 通过串口发送数据(二进制  低字节序)
                elif tag.id() == 1:
                    led_control(1)
                    uart.write(struct.pack('<BBBHH', 0xFF,0xA1 , True, tag.cx(),
tag.cy()))
                # 识别到id为2
                # 蓝灯亮
                # 通过串口发送数据(二进制  低字节序)
                elif tag.id() == 2:
                    led_control(2)
                    uart.write(struct.pack('<BBBHH', 0xFF,0xA2 , True, tag.cx(),
tag.cy()))
    else:
        # 没有找到任何 AprilTag
        # 熄灯
        # 通过串口发送数据(二进制  低字节序)
        led_control(3)
        uart.write(struct.pack('<BBBHH', 0xFF, 0xF1, False, 0, 0))
#    print(clock.fps())
```

11.7. STM32主程序解析

云台控制部分依旧和前面例程相同，使用了PID控制。

```
// 云台俯仰角 PID控制
void GimbalPitchPIDCtl()
{
    float dPitch; // 俯仰角的增量

    // 更新上一次的偏移量
    lastYOffset = curYOffset;
    // 计算当前的偏移量
    curYOffset = ((blobCy / IMG_HEIGHT) - 0.5);
    // 判断是否在死区内
    if (__fabs(curYOffset) <= DEAD_BLOCK)
    {
        curYOffset = 0;
    }

    // 计算得到偏航角的增量
    dPitch = GIMBAL_PITCH_KP * curYOffset + GIMBAL_PITCH_KD * (lastYOffset -
curYOffset);
```

```
    // 云台偏航角控制
    Gimbal_SetPitch(servoUsart, curPitch + dPitch, 500);

    printf("YAW PID: dPitch = %.1f next pitch = %.1f\r\n", dPitch, curPitch +
dPitch);
}
```

主循环中对curCmdId进行判断，如果是识别到标记，就用curMode记录ID。

```
while(1)
{
...
    if (curCmdId != BLOB_PKG_CMD_ID)
    {
        printf("enter %d\r\n",curCmdId);
        curMode = curCmdId;
        if(curMode != BLOB_PKG_CMD_ID_TAG1)
            curState = 0;
    }
...
}
```

判断curMode，进入不同的工作模式。

```
while(1)
{
...
        switch (curMode)
        {
        case BLOB_PKG_CMD_ID_TAG0:
            if (hasBlob)
                Gimbal_Reset(servoUsart);
            break;
        case BLOB_PKG_CMD_ID_TAG1:
        {
            PatrolMode();
        }
        break;
        case BLOB_PKG_CMD_ID_TAG2:
            if (hasBlob)
            {
                GimbalYawPIDCtl();
                GimbalPitchPIDCtl();
            }
            break;
        default:
            break;
        }
...
}
```

进入不同的模式前，需要用自定通信协议中的hasBlob来判断标记是否存在。

唯一例外的是`标记1`，识别到一次`标记1`后即进入巡逻模式，巡逻模式是自动循环运行。

```
if (hasBlob)
```