四轴tryhard心得

自从我一人成组报名了学校的四轴课程之后，说实话日子不好过了许多，可以说实际项目任务量和难度比应试要高多了。历经一个寒假的tryhard后，项目多少是有了些眉目，所以我先写一篇tryhard心得，主要目的是复习一下学到的各个知识，并仔细复盘，加深印象（毕竟有些代码是抄来的，不复盘等于没学过）

索引：

- 初步知识
- 转接板绘制
- OLED屏幕相关程序
- 蓝牙相关程序
- gy86相关程序
- 遥控器和接收机相关程序
- 电机驱动
- 无人机组装

初步知识

在本文的开头，还是决定写一点初步的知识，毕竟本人四轴的前一个项目就是点灯（莫笑TAT）

GPIO

全称General Purpose Input Output 通用功能输出输出，什么意思呢，个人认为就是说这个GPIO是从芯片内部引出来的一根线，这根线有复用的功能（即根据CPU不同的配置，这根线可以发挥不同的功能）GPIO通过一个电路进行配置，电路图相信大家都看过了，这里就不放上来了。

- 上拉输入模式：连接了上拉输入电阻，I/O端口的电平信号直接进入输入数据寄存器，但在IO端口悬空（无信号输入）时，输入端电平为高电平（也就是输入高电平），IO口输入低电平时，输入端电平就维持为低电平

输入模式下数据从IO口向输入数据寄存器移动，最后被读出（毕竟要输入嘛）

- 下拉输入模式：连接了下拉输入电阻。I/O端口的电平信号直接进入输入数据寄存器，但在I/O端口悬空（在无信号输入）的情况下，输入端的电平保持在低电平；并且在I/O端口输入为高电平的时候，输入端的电平也是高电平。
- 浮空输入模式下，I/O端口的电平信号直接进入输入数据寄存器。但没连上拉或下拉电阻。也就是说，I/O的电平状态完全由外部输入决定；如果在该引脚悬空（在无信号输入）的情况下，读取该端口的电平是不确定的。

  浮空输入模式通常用于IIC、USART。
- 模拟输入模式：模拟输入模式下，I/O端口的模拟信号（电压信号，而非电平信号）直接模拟输入到 片上外设模块（ADC我没用过), 个人感觉应该是特殊性较强的模式?
- 开漏输出模式:输出模式下, 写数据然后输出

  CPU往外写低电平（0）时 此时引脚接VSS(GND)相当于接地
  CPU外外写高电平（1）时 此时引脚的电平状态由上下拉电阻决定
- 推挽输出模式

  CPU往外写高电平（1）时，此时引脚输出一个高电平
  CPU往外写低电平（0）时，此时引脚输出一个低电平
- 开漏和推挽的复用模式: 与原输出模式很是类似。只是输出的高低电平的来源，不是让CPU直接写输出数据寄存器，取而代之利用片上外设模块的复用功能输出来决定的

这段总结参考的是GPIO输入输出模式原理(八种工作方式附电路图详解)_gpio四种输入输出模式-CSDN博客，里面的图很形象，如果想详细分析可以看原blog

UART

通用异步接收器/发送器，通常称为UART，是一种广泛应用于嵌入式领域的串行、异步、全双工通信协议。

UART 通道主要通过交叉连接的 RX 引脚和 TX 引脚传输数据，两端GND相连用于统一地线的电压，这样可以统一两端对高低电平的理解

UART的特点：串行，异步，全双工：

- 串行：串行通信是指利用一根传输线逐位依次传输数据（也可以用两根信号线组成全双工通信）
- 异步通信以一个字符为传输单位。通信中两个字符之间的时间间隔不固定，但同一字符中相邻两位之间的时间间隔固定。所以UART设备间一般没有时钟线，
- 两个设备需要指定相同的传输速率，指定相同的空闲位、起始位、奇偶校验位和结束位。数据传输速率以波特率表示，即每秒传输的位数波特率的匹配是正确传输数据重要因素。
- 数据格式由：起始位，数据位，奇偶校验位组成

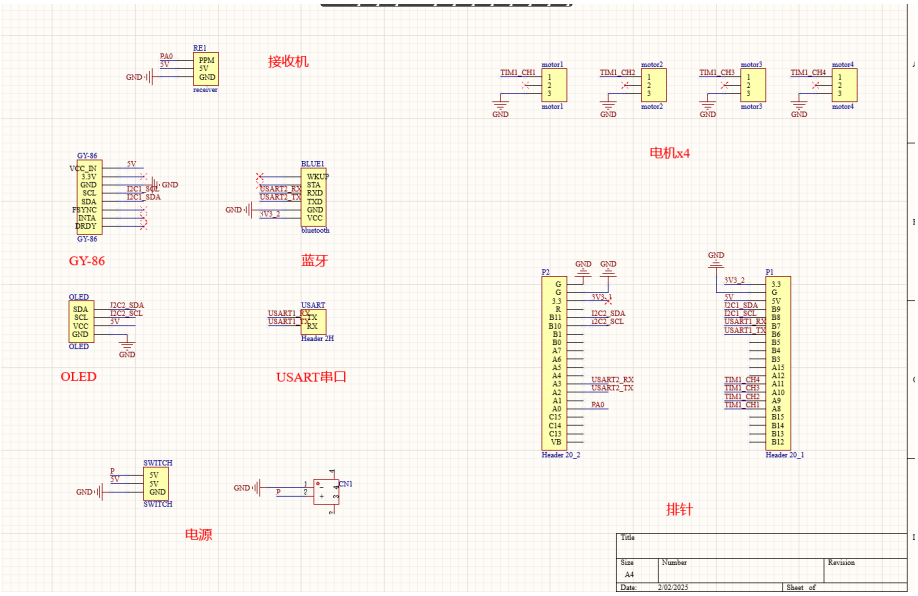
还有其他各方面的知识，时间关系不一一细说，之后用到了再介绍。

转接板绘制

转接板是让芯片中的响应引脚统合到相邻位置的板子，这个板子上没有任何元件，只是电路的排布。转接板是板子中最好画的一类（虽然我也花了老半天才搞懂😇）

转接板绘制的第一步是搞清楚自己要整合芯片的哪些引脚，也就是要罗列出本次项目要用到的所有元件，找到对应的引脚，进行标记。本次项目我使用了OLED、蓝牙、GY-86、接收机。我使用CUBEMX为其分配对应的引脚，不得不说，还是很好用的。

接下来是转接板绘制的第二步，我们需要在AD上先画出原理图，因为我们画的是转接板，所以几乎所有“元件”在图中都是以排针的形式出现的，基本不用去网站上下载太多素材。我们只需要为每个元件分配对应的排针，然后利用AD中的功能把该整合在一起的引脚整合起来就行，最后就是这种效果：



再下来一步就是导入PCB图，排列位置（最好参考一下各个元件的大小，不然可能要画很多次，我居然画了3次，花了好多钱🤡）

不得不说如果在嘉立创打板子的话用嘉立创EDA应该要好一点，AD在嘉立创打板还是有很多局限的。

OLED屏幕

接下来就是对各个程序的分析了，OLED自认为比其他的元件简单，出错可能性比较低，原因之一就是有很多已经写好了的代码可以直接用；同时四轴项目中用到OLED的地方不多，所以功能代码本来就少。

OLED屏幕利用I2C传输数据，控制屏幕某处的像素亮灭，最终形成图像。这里主要注意不同型号的OLED屏幕启动程序不一样，还有就是画图的时候搞清楚比例就行了，(毕竟大部分代码都是抄的)难度不高

蓝牙

蓝牙组件可以向手机/电脑传输数据，会使未来调试十分方便，由于现在项目中的蓝牙功能只需要发送调试数据即可，所以功能较为简单，我就放在OLED后面说了。

说是蓝牙，其实只是蓝牙透传模块，这里我们只是初步的使用了蓝牙的功能，最终只达成单向的传输数据，以起到帮助调试的作用。

蓝牙透传相当于一个不连线的串口，这里我们有DMA, IT，串口等多种形式可以选择，关于如何选择，主要看怎样才适合程序，由于GY86没到，所以先完成了遥控器接收机PPM信号的接收并使用蓝牙传递，这点我们在接收机模块详细叙述吧。

蓝牙透传比较需要注意的只有参数的配置和接线了，注意一下波特率的对应，同时接线时TX和RX交叉接线（虽然这很基础，但是有一次接线的时候我居然忘了，想了半天出错在哪）

蓝牙透传的程序相对简单，我们就以一个测试程序作为例子吧：

```
  char defaultData[] = "Bluetooth";
  uint16_t len_default = sizeof(defaultData) - 1;  // 去掉
末尾的\0
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      if(!uart_tx_busy){
              HAL_UART_Transmit(&huart2,
(uint8_t*)defaultData, len_default, 500);
          }
  }
```

可以成功收到蓝牙信息，当时忘记截图了，之后补上

GY-86相关程序

gy86应该是较为复杂的一部分了，它是一个使用IIC通信的十轴姿态传感器，主要包含MPU6050、HMC5883L、MS5611三个模块

MPU6050

MPU6050是一个六轴传感器，不仅可以测量x，y，z三轴的加速度，还可以测量三轴的角速度

```
#define MPU6050_ADDRESS 0xD0
#define MPU6050_PWR_MGMT_1 0x6B //电源管理寄存器1
#define MPU6050_PWR_MGMT_2 0x6C //电源管理寄存器2
#define MPU6050_SMPRT_DIV 0x19 //采样频率分频器
#define MPU6050_CONFIG 0x1A //配置寄存器
#define MPU6050_GYRO_CONFIG 0x1B //陀螺仪配置寄存器
#define MPU6050_ACCEL_CONFIG 0x1C //加速度传感器配置寄存器
#define MPU6050_USER_CTRL 0x6A //用户控制寄存器
#define MPU6050_INT_PIN_CFG 0x37 //旁路and引脚启用配置
```

```c
#define MPU6050_ACCEL_XOUT_H 0x3B
#define MPU6050_ACCEL_XOUT_L 0x3C
#define MPU6050_ACCEL_YOUT_H 0x3D
#define MPU6050_ACCEL_YOUT_L 0x3E
#define MPU6050_ACCEL_ZOUT_H 0x3F
#define MPU6050_ACCEL_ZOUT_L 0x40
#define MPU6050_GYRO_XOUT_H 0x43
#define MPU6050_GYRO_XOUT_L 0x44
#define MPU6050_GYRO_YOUT_H 0x45
#define MPU6050_GYRO_YOUT_L 0x46
#define MPU6050_GYRO_ZOUT_H 0x47
#define MPU6050_GYRO_ZOUT_L 0x48
//各个寄存器的位置和配置，当然是查表得

void MPU6050_Init(){
    HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDRESS,
MPU6050_PWR_MGMT_1, I2C_MEMADD_SIZE_8BIT, 0x01, 1,
HAL_MAX_DELAY); //解除睡眠，选择陀螺仪时钟
    HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDRESS,
MPU6050_PWR_MGMT_2, I2C_MEMADD_SIZE_8BIT, 0x00, 1,
HAL_MAX_DELAY); //六个轴都不待机
    HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDRESS,
MPU6050_SMPRT_DIV, I2C_MEMADD_SIZE_8BIT, 0x09, 1,
HAL_MAX_DELAY); //10分频
    HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDRESS,
MPU6050_CONFIG, I2C_MEMADD_SIZE_8BIT, 0x06, 1,
HAL_MAX_DELAY); //滤波参数最大
    HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDRESS,
MPU6050_GYRO_CONFIG, I2C_MEMADD_SIZE_8BIT, 0x18, 1,
HAL_MAX_DELAY); //陀螺仪最大量程
    HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDRESS,
MPU6050_ACCEL_CONFIG, I2C_MEMADD_SIZE_8BIT, 0x18, 1,
HAL_MAX_DELAY); //加速度计最大量程
    HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDRESS,
MPU6050_USER_CTRL, I2C_MEMADD_SIZE_8BIT, 0x80, 1,
HAL_MAX_DELAY); //控制I2C主模式使能?
    HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDRESS,
MPU6050_INT_PIN_CFG, I2C_MEMADD_SIZE_8BIT, 0x02, 1,
HAL_MAX_DELAY); //旁路模式?

}

void MPU6050_GetData(int16_t* AccX, int16_t* AccY,
int16_t* AccZ, int16_t* GyroX, int16_t* GyroY, int16_t*
GyroZ) {
    uint8_t buffer[14]; // 加速度计+温度+陀螺仪共14字节

    // 一次性读取所有传感器数据（0x3B开始，连续14字节）
    HAL_I2C_Mem_Read(&hi2c1, MPU6050_ADDRESS,
MPU6050_ACCEL_XOUT_H, I2C_MEMADD_SIZE_8BIT, buffer, 14,
HAL_MAX_DELAY);
```

```
    // 解析加速度计数据（X/Y/Z各占2字节）
    *AccX  = (buffer[0]  << 8) | buffer[1];
    *AccY  = (buffer[2]  << 8) | buffer[3];
    *AccZ  = (buffer[4]  << 8) | buffer[5];
//左移8位后|下一个字节，这个操作相当于把buffer[0]和buffer[1]拼接成了一个16位数据:buffer[0]-buffer[1],其他同理
    // 跳过温度数据（buffer[6]和buffer[7]）

    // 解析陀螺仪数据（X/Y/Z各占2字节）
    *GyroX = (buffer[8]  << 8) | buffer[9];
    *GyroY = (buffer[10] << 8) | buffer[11];
    *GyroZ = (buffer[12] << 8) | buffer[13];
}
```

HMC5883L

HMC5883L是基于霍尔效应和磁场测量的三轴磁力传感器, 借由HMC5883L, 我们可以得到x, y, z轴表示磁场信息的数字信号.

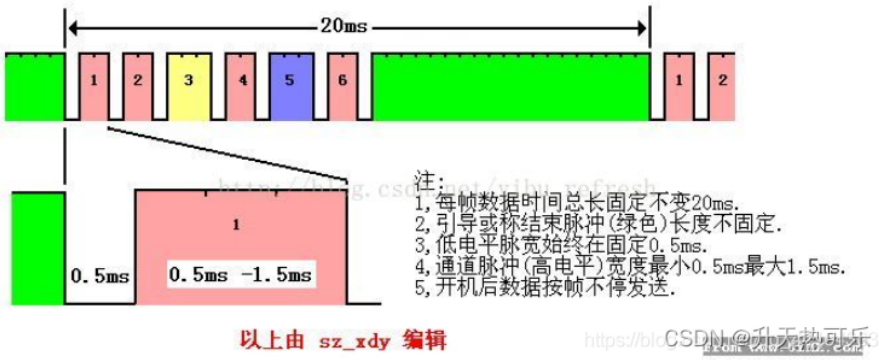
```
// HMC5883L的配置地址
#define HMC5883L_ADDRESS 0x1E // HMC5883L的I2C地址（7位地址）
#define HMC5883L_REG_CONFA 0x00 // 配置寄存器A
#define HMC5883L_REG_CONFB 0x01 // 配置寄存器B
#define HMC5883L_REG_MODE 0x02 // 模式寄存器
// HMC5883L的配置指令
#define HMC5883L_REG_X_OUT_L 0x03 // X轴输出低字节
#define HMC5883L_MODE_CONTINUOUS 0x00 // 连续模式
#define HMC5883L_AVERAGING_8 0x40 // 8次平均值
#define HMC5883L_RATE_15 0x03 // 15Hz的数据输出率

void HMC5883L_Init(){
    HAL_I2C_Mem_Write(&hi2c1, HMC5883L_ADDRESS, HMC5883L_REG_CONFA, I2C_MEMADD_SIZE_8BIT, HMC5883L_AVERAGING_8 | HMC5883L_RATE_15, 1, HAL_MAX_DELAY);
    //MC5883L_AVERAGING_8 表示每次测量时对 8 个样本进行平均，以提高测量的精度；
    //HMC5883L_RATE_15 表示数据输出速率为 15Hz。
    HAL_I2C_Mem_Write(&hi2c1, HMC5883L_ADDRESS, HMC5883L_REG_CONFB, I2C_MEMADD_SIZE_8BIT, 0x00, 1, HAL_MAX_DELAY);
    //配置寄存器 B 写入值 0x00表示使用默认的增益设置。
    HAL_I2C_Mem_Write(&hi2c1, HMC5883L_ADDRESS, HMC5883L_REG_MODE, I2C_MEMADD_SIZE_8BIT, HMC5883L_MODE_CONTINUOUS, 1, HAL_MAX_DELAY);
    //表示将传感器设置为连续测量模式。在连续测量模式下，传感器会不断地进行磁场测量，并更新测量数据。
}
void HMC5883L_ReadData(int16_t *x, int16_t *y, int16_t *z)
```

```
{
    uint8_t data[6];
    // 读取X、Y、Z轴的数据
    HAL_I2C_Mem_Read(&hi2c1, HMC5883L_ADDRESS,
HMC5883L_REG_X_OUT_L, I2C_MEMADD_SIZE_8BIT, data, 6,
HAL_MAX_DELAY);
    // 将读取到的高低字节组合成16位整数
    *x = (data[1] << 8) | data[0];
    *y = (data[3] << 8) | data[2];
    *z = (data[5] << 8) | data[4];
    // 应用校准和滤波处理（省略具体实现）
}
void getXYZ(int16_t *x, int16_t *y, int16_t *z){
    uint16_t sum_array[3] = {0, 0, 0};
        int16_t a, b, c;
        for (int i = 0; i < 3; i++) {
            HMC5883L_ReadData(&a, &b, &c);
            sum_array[0] += a;
            sum_array[1] += b;
            sum_array[2] += c;
        }
        *x = sum_array[0] / 3;
        *y = sum_array[1] / 3;
        *z = sum_array[2] / 3;
}
```

gy86还没到，配好了再说吧。

遥控器和接收机

此次项目遥控器和接收机采用了PPM方式进行传递，什么是PPM呢，相信大家都知道PWM，PPM个人理解就是把需要多个通道实现的多个PWM信号整合到了一个时间段内：



可以看到，一个数据帧固定有20ms，每个高电平之间固定间隔0.5ms低电平，用于隔开不同的数据。每个高电平的长度在0.5ms到1.5ms，不同的高电平长度表述不同的数据。接收完数据后，便会如上图绿色区域一样进入持续高电平，一直直到此次接收时间超过20ms时，就进入下一帧了。

可以看到，若油门处于最低时，接收的时间应该是1ms（0.5 +0.5）处于最大时，为2ms（0.5 + 1.5）若接收时间大于2ms，则进入下一轮解析。

我们买的接收机是只有PPM，GND，5V三根引脚的简化版，我们只能通过PPM的信号进行遥控器指令的解析，首先，我们给与PPM连接的引脚开启中断（由上图我们可以看到，是下降沿触发中断）

然后，由于PPM是以高电平持续时间判断传输数据的，我们还需要开启一个计时器，用于记录每个高电平的持续时间。详细的配置和解析实验的思路我主要参考了一篇blog，如果想深入实践，可以去看那篇blog的内容，放在了这一段结尾。

```
#include "ppm.h"
#include "main.h"

uint16_t PPM_Sample_Cnt = 0;//通道
uint16_t PPM_Chn_Max = 8;//最大通道数
uint32_t PPM_Time = 0;//获取通道时间
uint16_t PPM_OK = 0;//下一次解析状态,初始为0

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_0)
    {
        // 移除所有Delay函数
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);

        // 进入临界区（可选）
        __disable_irq();
        //每次遇到下降沿,就读取一次计时器,并将计时器清零,这样可以保证每一次读计时器时,都可以读到两次下降沿之间的时间间隔
        PPM_Time = TIM2->CNT;
        TIM2->CNT = 0;
        //这个地方看着有些怪
        if(PPM_OK){
            //记录时间并存入数组
            if(PPM_Sample_Cnt < PPM_Chn_Max){
                PPM_Databuf[PPM_Sample_Cnt] = PPM_Time;
                PPM_Sample_Cnt++;
            }
            if(PPM_Sample_Cnt >= PPM_Chn_Max){
                //如果超过了最大通道数,就不读了,并将ppm_data_ready置1,表示可以通过蓝牙传递ppm数据了
                PPM_OK = 0;
                ppm_data_ready = 1;
            }
        }

//如果读到的PPM_Time大于等于2050,也就是进入了上图绿色的区域,我们就认为这一帧内的数据发送完毕,所以PPM_OK=1,我们可以进入解析下一帧数据的环节(也就是上面那个if)
       if(PPM_Time >= 2050){
            PPM_OK = 1;
```

```
                PPM_Sample_Cnt = 0;
            }
            __enable_irq(); // 退出临界区
        }
    }
```