IFR 嵌入式编程规范补充附录

编程规范不是按照我的风格来写，也不是自己怎么舒服怎么写。而是以兼容、通用、易于 DEBUG 为目标去写。但是我毕竟已经踩了三年的坑，除非你已经明确知道这段代码为什么这么写并觉得有更好方案或觉得对自己的功能不合适。否则建议想一下为什么我这么写。

1、代码工程各个文件命名、文件夹内容分工明确：(以下顺序不分前后)【注：下面说的只是工程内文件夹规范，不是实际文件储存规范，比如 FWLIB 实际储存有很多子文件夹 inc src core 等，但是工程内只有一个　就是 FWLIB】

USER:存放 main.c 或者 MyDefine.c,这个一般不会错

APP:凡是涉及上层应用、逻辑性、功能性文件，都放在这里，不要和底层数据解析/协议解析 analysis 弄混，该层只关心数据怎样利用、或处理，不关心怎样发送、怎样编排、怎样接受。

BSP:存放底层初始化文件，命名格式：外设名(含编号)_功能名.c　如：usart1_remote.c gpio_led.c pwm_buzzer.c 部分多功能外设可以不遵循此规范 如 can.c（如有更好方案欢迎提出）

ALGO:存放通用算法文件，如 pid.c CRC_check.c

ANALYSIS:存放底层数据解码/协议解码文件。如遥控器数据经过该层文件编程可直接读取的 RC_Ctl 。原始 WiFiDebug 数据经过该层通过我的协议解析变成可直接发送的/读取的数据，然后 APP 应用层的 wifidebug 只需要操作需要发什么数据而不需要关心数据协议需要怎么样编排，做到模块分离。底盘、云台、取单等等逻辑的分析不要放在此层！放在 APP 层

FWLIB:存放所有 stm 库文件，因为平时不会去看它，所以用一个文件放进所有需要的就行，我个人觉得正点原子的 sys.c 属于 APP 层，不属于 FWLIB 层。

2、函数形参、调用、注释规范

假想你现在正在写两辆底盘结构一样但是电机不一样的车，或者在写两辆视觉反馈不同电机也不同但是都要实现自瞄的云台。请问你现在的代码能实现 80%以上复用吗？我指的复用是函数不需要改什么就直接能在两辆车通用。以及如果另一个人来帮你调试代码能不能很快理解你写的是什么--怎样实现代码的复用性

功能分离思想

一个函数只做一种功能，不要图一时快捷把多个功能写在一个函数，就好比云台不要写底盘里，底盘不要写云台里。在一些小函数、小功能上道理一样。这样不管是新建还是 DEBUG，只需要一个一个小功能的测试、调试。人的大脑同一时间面对的事物是有限的，你如果把很多小功能写在一个函数里，那就是在自找自闭。不要钻牛角尖。

代码复用思想

凡是相同类的计算、逻辑，并使用多次的或者未来可能使用多次的，封装为一个函数在其他地方调用，提炼出每个计算所需变量写成结构体，若干个相同类的写成结构体数组去调

用。这样：1、代码清晰，代码量少。2、后期优化修改方便，只需要改该功能函数，所有调用功能函数的地方就都改了。

举例：遥控器接收数据解析电脑键盘，识别长按、短按等，因为若干个按键需要解析的逻辑、计算都是一样的，所以对按键识别解析封装为函数

*void ButtonStatu_Verdict(KeyBoardTypeDef * Key)*

建立结构体

```
typedef struct
{
    u16 count;
    u8 value;
    u8 last;
    u8 statu;
}KeyBoardTypeDef;
```

储存对于单个按键所需的所有变量

建立结构体数组，数组每个元素代表一个按键，为了更方便明了的调用结构体数组，使用枚举来定义每一位代表什么意思

```
enum KEYBOARDID      //键位标识
{
    KEY_W,
    KEY_S,
    KEY_A,
    KEY_D,
    KEY_SHIFT,
    KEY_CTRL,
    KEY_Q,
    KEY_E,\
    \
    KEY_R,
    KEY_F,
    KEY_G,
    KEY_Z,
    KEY_X,
    KEY_C,
    KEY_V,
    KEY_B,
    KEY_NUMS,
};
```

这样在 chassis 调用键位只需要 *KeyBoardData[KEY_W].value* 就可以了

枚举最后面的 *KEY_NUMS* 意义在于记录共有几个枚举，这样初始化结构体时直接

KeyBoardTypeDef KeyBoardData[KEY_NUMS]={0};

就可以了，而无需思考需要建立多大的数组，而且就算之后新增了一个键位或者删除了一个键位，该值大小也会自动调整，一劳永逸。

拿一个实际的例子来说一下：（注意，橙黄色为反例）

```
void StoreDisData(void)                                    //在 bsp_timer.c 里面发送
{
    static u16 avert_front_up_left      = 0;
    static u16 avert_front_up_right     = 0;
    static u16 avert_front_down_left    = 0;
    static u16 avert_front_down_right = 0;
    static u16 avert_behind_left         = 0;
    static u16 avert_behind_right        = 0;
    static u16 avert_check                 = 0;
    ////////////////////////////////////////////////////////////////////////////
    u8 ele_front_up_left =   GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_6);
    u8 ele_front_up_right = GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_7);
    u8 ele_front_down_left = GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_4);
    u8 ele_front_down_right = GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_5);
    u8 ele_behind_left   = GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_8);
    u8 ele_behind_right = GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_9);
    u8 ele_check_yes_no = GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_1);
    ////////////////////////////////////////////////////////////////////////////
    text_frnt = ele_front_up_left;
    if (ele_front_up_left == 0)
    {
        if (avert_front_up_left < 5)
        {
            avert_front_up_left++;
        }
        else
        {
            if (ele_front_up_left == 0)
            {
                DisStore.LIGHTELE_FRONT_UP_LEFT = ele_front_up_left;
//车前方左边上岛消抖
                avert_front_up_left = 0;
            }
        }
    }
    else
    {
        if (avert_front_up_left < 5)
        {
            avert_front_up_left++;
        }
        else
        {
            if (ele_front_up_left == 1)
```

```
                {
                    DisStore.LIGHTELE_FRONT_UP_LEFT = ele_front_up_left;
//车前方左边上岛消抖
                    avert_front_up_left = 0;
                }
            }
        }
        ////////////////////////////////////////////////////////////////////////
        if (ele_front_up_right == 0)
        {
            if (avert_front_up_right < 5)
            {
                avert_front_up_right++;
            }
            else
            {
                if (ele_front_up_right == 0)
                {
                    DisStore.LIGHTELE_FRONT_UP_RIGHT = ele_front_up_right;
//车前方右边上岛消抖
                    avert_front_up_right = 0;
                }
            }
        }
        else
        {
            if (avert_front_up_right < 5)
            {
                avert_front_up_right++;
            }
            else
            {
                if (ele_front_up_right == 1)
                {
                    DisStore.LIGHTELE_FRONT_UP_RIGHT = ele_front_up_right;
//车前方左边上岛消抖
                    avert_front_up_right = 0;
                }
            }
        }
////////////////////////////////////////////////////////////////////////
```

……部分代码面积过大未贴出，总之还有数十行形如上面的 if else

修改意见：

1、变量命名：一定要突出这个变量的本质特征再加上他的逻辑特征

建议：直接读取 GPIO 的变量末尾加_value 或者_rawdata

即 ele_front_up_left 改为 *xxxxxx_value* 或 *xxxxxxx_rawdata*

建议：计数型变量末尾加_count

即：avert_front_up_left 改为 *xxxxxx_count*

2、函数封装：将重复性的逻辑、计算封装为函数

建议：将形如：

```
static u16 avert_front_up_left       = 0;
```

```
u8 ele_front_up_left =    GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_6);
if (ele_front_up_left == 0)
      {
            if (avert_front_up_left < 5)
            {
                  avert_front_up_left++;
            }
            else
            {
                  if (ele_front_up_left == 0)
                  {
                        DisStore.LIGHTELE_FRONT_UP_LEFT = ele_front_up_left;
//车前方左边上岛消抖
                        avert_front_up_left = 0;
                  }
            }
      }
      else
      {
            if (avert_front_up_left < 5)
            {
                  avert_front_up_left++;
            }
            else
            {
                  if (ele_front_up_left == 1)
                  {
                        DisStore.LIGHTELE_FRONT_UP_LEFT = ele_front_up_left;
//车前方左边上岛消抖
                        avert_front_up_left = 0;
                  }
            }
      }
```

建议：将形如这样的几十行代码封装为一个函数

*SingleSwitch_Data_Filter(SingleSwitch_Data * pdata)*

将下面三个相关变量封装为结构体：

```
static u16 avert_front_up_left       = 0;
u8 ele_front_up_left =    GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_6);
DisStore.LIGHTELE_FRONT_UP_LEFT = ele_front_up_left;
```

即：

```
Typedef struct
{
      U8 raw_value;
      Ele_count;
      U8 deal_value;
      ……
} SingleSwitchDataTypeDef;
```

然后其他 8 个传感器

建立枚举 ID 表

```
Typedef enum
```

```
{
    SWITCH_FRONT_UP_LEFT,\
    SWITCH_FRONT_UP_RIGHT,\
    SWITCH_FRONT_DOWN_LEFT,\
    ..........
    SWITCH_NUMS,
}
```

初始化 SingleSwitch_Data 时直接

```
SingleSwitchDataTypeDef SingleSwitch_Data[SWITCH_NUMS];
```

处理时直接

```
For(int i=0;i< SWITCH_NUMS;i++)
{
    SingleSwitch_Data_Filter(&SingleSwitch_Data[i]);
}
```

代码移植思想：

函数传参规范化，移植性：

思想：函数传入参数应该仅仅是这个函数关心的内容，多余的信息尽量不要传入。举个例子，你要写个滤波函数，滤云台电机速度反馈的。在你设计函数时。你可以直接把整个云台反馈结构体指针作为一个参数传递给这个函数，没错，这样没问题，但是一旦你换了个机器，或者换了下反馈，甚至只是想重构代码，换了下结构体内部变量的名字。你这个函数就得改，因为你的函数内部是直接 *结构体.变量* 外面的结构体任何构造信息一改，你的函数就也得改。麻烦么？这个也别钻牛角尖。而且如果你想和其他人交流代码，别人根本没法用你的，因为两辆车可能底盘、云台不一样……但是换种规范，如果你直接把这个函数所关心的结构体中的那个变量作为参数传递进函数，那即使外面结构体怎么改，你只需要改调用就行，这个函数根本就不用改，代码交流时也能直接 copy 不需要改什么。

反例：

```
float Yaw_Error_Angle(VISION_DATA *pData)
{
    float YawAngle_error = 0;
    float pixel_error_x = 0;

    pixel_error_x = pData->tar_x - cs_vision_x;
    //YawAngle_error = atan(pixel_error_x/VISION_PARAMETER_VALUE_X)*57.3f;
    YawAngle_error = atan(pixel_error_x/VISION_BUFF_VALUE_X)*57.3f;
    return YawAngle_error;
}
```

问题：上图函数传入参数为 VISION_DATA *pData，相当于把整个结构体内容都传进来了但是函数里只用了一个结构体变量，而且是不需要修改的变量（就是只取值，不修改值），如果只取值就没必要传指针，一方面没必要，另一方面也更安全，防止自己对不该修改的变量进行了修改。 如图

```
float Yaw_Error_Angle(VISION_DATA *pData)
{
    float YawAngle_error = 0;
    float pixel_error_x = 0;

    pixel_error_x = pData->tar_x - cs_vision_x;
    //YawAngle_error = atan(pixel_error_x/VISION_PARAMETER_VALUE_X)*57.3f;
    YawAngle_error = atan(pixel_error_x/VISION_BUFF_VALUE_X)*57.3f;
    return YawAngle_error;
}
```

建议：函数形参改为 float Yaw_Error_Angle(u16 tar)

因为不需要修改变量值，所以不用指针。因为只用到这一个变量，所以参数就是一个变量。这样该函数就和 VISION_DATA 结构体信息无关了。便于交流、修改。