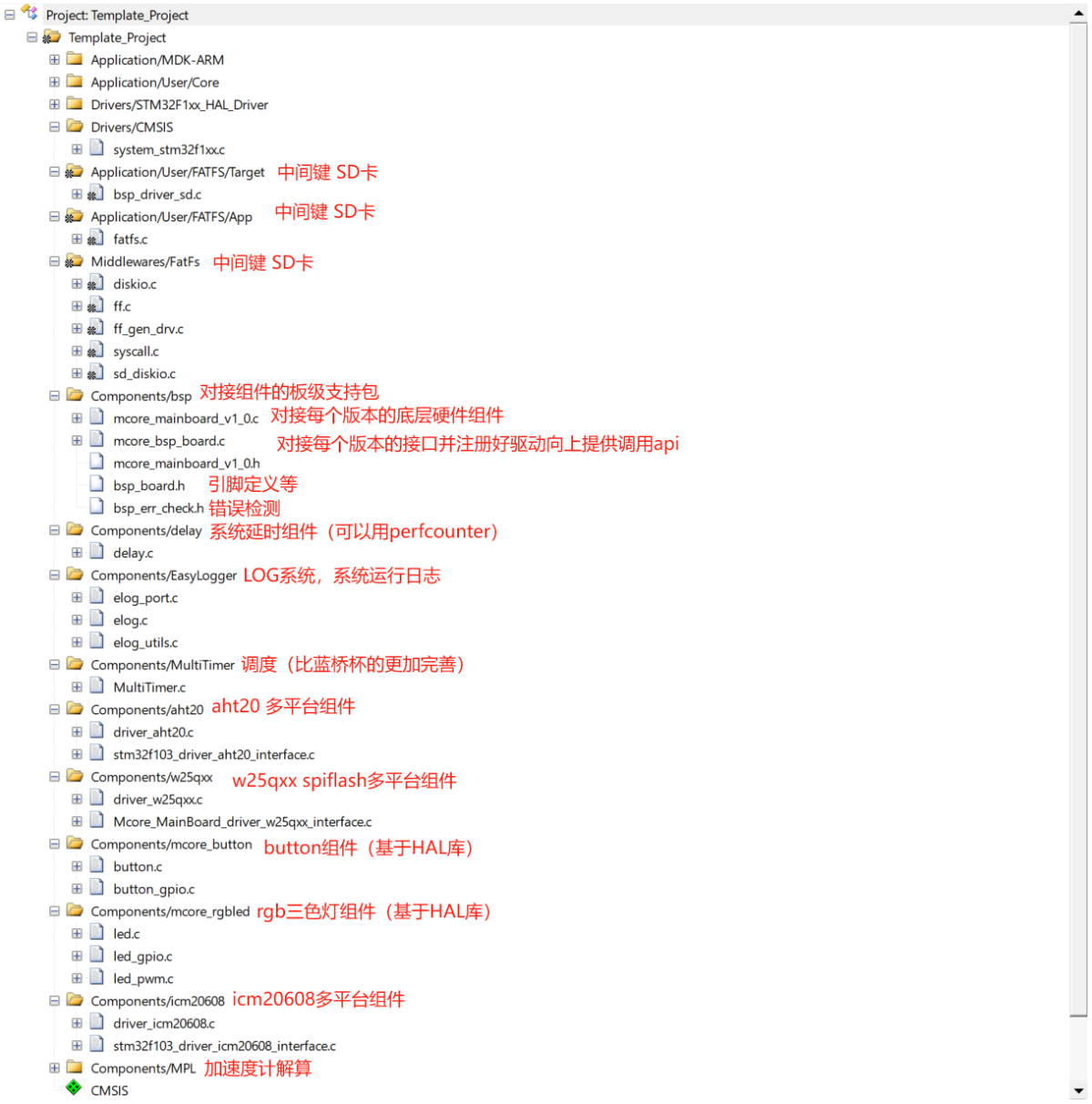
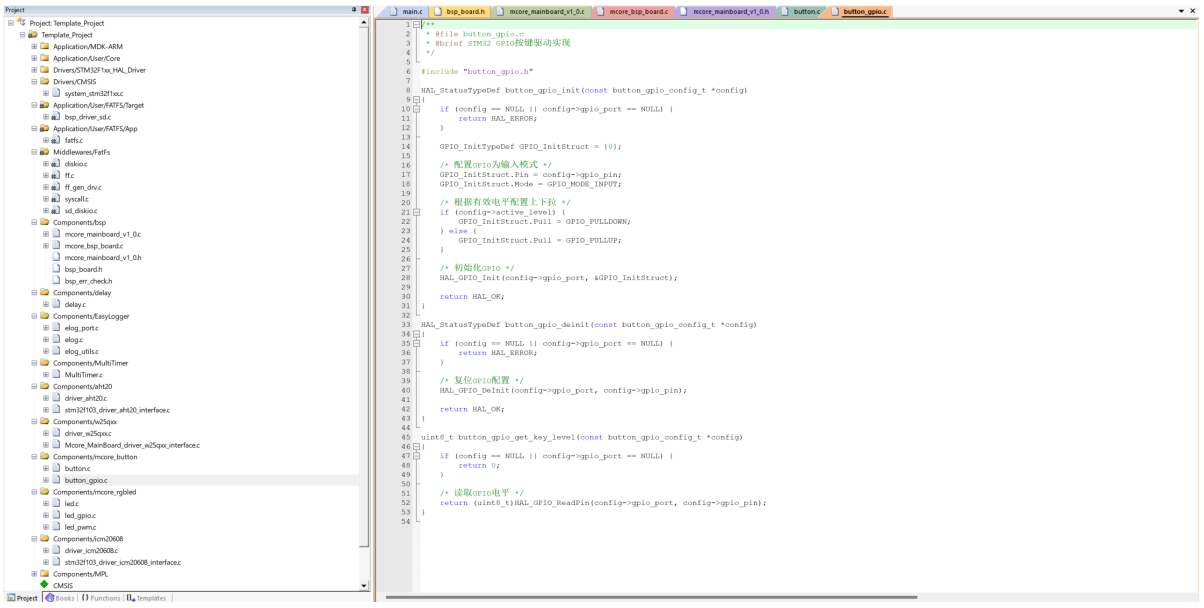
一、目录结构



二、分层的必要性

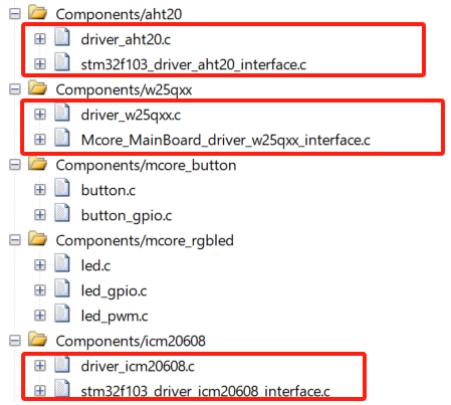
我们平时写的代码很多都有共同性，我们可以把一部分可以复用到其他平台的代码给他独立封装出来，比如说这里的`button`，不同平台其实只有读取`gpio`的操作不一样，其他的按键逻辑都是差不多的。这里我们按键相比蓝桥杯多加入了按键注册的功能，通过回调来具体到每个按键每个事件执行什么操作，就不需要都写到一个`handle`里面，利于代码阅读跟维护。

此外按钮的注册使用到了内存分配，这里我们可以使用静态分配的方式去注册，但是这里由于时间关系才临时写成了动态，在实际开发中我们需要保证产品的稳定运行，防止内存溢出，所以在一些简单的产品时间开发中，程序运行中我们的内存是一定要确定的。



三、组件

组件顾名思义就是组成我们功能的一个一个部分，我们在不同产品不同项目开发中，有很多组件是可以复用的，但是可能因为成本的问题，我们会选型不同的mcu平台，所以我们需要把底层接口给他抽象出来，这里我们使用到的一些组件库是来自libdriver的开源组件库，他家写的外设组件库是比较全的。



比如说这里的aht20 w25qxx都是基于他开源出来，我们移植他组件的时候，我们只需要关注引脚 跟协议的初始化，并向他的组件提供这些api即可完成移植。但是跟hal库生成的却偶尔会有些小冲突，所以我们也可以写一个接口转换的.c.h也可以直接注释掉Cubemx生成的，我们现在要转换一个思想，就是hal库生成出来的一些东西只是为了方便我们开发，实际开发中，有一些逻辑只要不符合我们业务的需要，我们是可以大胆去改的，无论是HAL库还是标准库，工具如何组合，都取决于我们如何使用，HAL库只是因为ST做了一个生成的工具，但是我们不能仅局限于那种框架。我们可以学习他的组件是如何能够在不同平台F1 F4 G4 H7等等平台中兼容，这些是值得我们去学习的。

```
 * @note   none
 */
uint8_t aht20_interface_iic_init(void)
{
    return iic_init();
}

/**
 * @brief  interface iic bus deinit
 * @return status code
 *         - 0 success
 *         - 1 iic deinit failed
 * @note   none
 */
uint8_t aht20_interface_iic_deinit(void)
{
    return iic_deinit();
}

/**
 * @brief      interface iic bus read
 * @param[in]  addr is the iic device write address
 * @param[out] *buf points to a data buffer
 * @param[in]  len is the length of the data buffer
 * @return     status code
 *             - 0 success
 *             - 1 read failed
 * @note       none
 */
uint8_t aht20_interface_iic_read_cmd(uint8_t addr, uint8_t *buf, uint16_t len)
{
    return iic_read_cmd(addr, buf, len);
}

/**
 * @brief      interface iic bus write
 * @param[in] addr is the iic device write address
 * @param[in] *buf points to a data buffer
 * @param[in] len is the length of the data buffer
 * @return     status code
 *             - 0 success
 *             - 1 write failed
 * @note       none
 */
uint8_t aht20_interface_iic_write_cmd(uint8_t addr, uint8_t *buf, uint16_t len)
{
    return iic_write_cmd(addr, buf, len);
}
```

好比说这里的iic，在libdriver里面提供了这些interface接口出来需要我们去注册，我们就直接往里面填充就好了。没有这种接口我们就去造。

```
/**
 * @brief      iic bus write with 16 bits register address
 * @param[in] addr is the iic device write address
 * @param[in] reg is the iic register address
 * @param[in] *buf points to a data buffer
 * @param[in] len is the length of the data buffer
 * @return     status code
 *             - 0 success
 *             - 1 write failed
 * @note       addr = device_address_7bits << 1
 */
uint8_t iic_write_address16(uint8_t addr, uint16_t reg, uint8_t *buf, uint16_t len)
{
    if (HAL_I2C_Mem_Write(&hi2c1, addr << 1, reg, I2C_MEMADD_SIZE_16BIT, buf, len, 1000) != HAL_OK)
    {
        return 1;
    }

    return 0;
}

/**
 * @brief       iic bus read command
 * @param[in]  addr is the iic device write address
 * @param[out] *buf points to a data buffer
 * @param[in]  len is the length of the data buffer
 * @return     status code
 *             - 0 success
 *             - 1 read failed
 * @note       addr = device_address_7bits << 1
 */
uint8_t iic_read_cmd(uint8_t addr, uint8_t *buf, uint16_t len)
{
    if (HAL_I2C_Master_Receive(&hi2c1, addr, buf, len, 1000) != HAL_OK)
    {
        return 1;
    }

    return 0;
}
```

上面的driver_xxx不同组件我们在不同平台移植，我们只需要更改这个interface即可。

这里的mcore_button是基于之前的版本，按键的业务裸机混乱而改出来，我们单击，双击，释放等事件都可以很轻易的注册到回调里面去书写业务代码，在实际开发我们也不会经常使用到多按键触发，如果有，我还有一个按键框架，但是对于现在来说，这个按键框架已经非常实用。

四、log系统

我们不能确保我们的系统实时运行在理想状态中，我们写的代码也会出错，我们需要日志的引入来帮助我们来排除，一个项目若没有一个好的调试手段，那么在后面的功能拓展中，我们继续添加功能会显得非常吃力，我们并不能知道哪些地方没有出错。所以我们需要加入一个log系统来帮助我们来排除，比如说有一些log是错误日志，他只是偶尔触发，我们却不能一直盯着日志来排除，我们可以引入flash或者eeprom，sd卡这种方式把日志存储在外部，以便我们读取。

五、MultiTimer

这个组件是蓝桥杯调度器相似的比较满血的版本，这种方式相比传统我们用数组的方式来注册更加灵活，我们可以自定义他的启动时间点，你把启动在初始化的时候运行一次，然后在他对应的回调里面继续放一个start就可以实现轮询的效果，此外我们可以引入其他的设计模式来更加灵活的满足我们的业务代码，比如说rtos中的各种事件，我们是否可以书写一个简单的裸机版本来帮我们实现这个东西。相比之前的调度器优势，比如说有一个东西我需要在他触发多少秒后，我才执行某个功能，我可以使用这个东西，设置一个超时时间之后，直接把要做的东西丢在那个回调里面即可实现。