## 1 初级要求

1. 不用u8 u32 u16 ，unsigned char,unsigned short,这些类型， 使用uint8\_t ，uint16\_t ，uint32\_t 。
2. 非外部使用的接口函数，都使用static声明和定义。
3. 全局变量都定义成static，限制作用域，禁止从外部直接访问。
4. 模块化编程
   1. 每个模块的文件命名都加上模块所属层级的前缀。
   2. 模块内的给外部调用的接口函数都加上模块层级的前缀。
5. 变量和函数都使用下划线命名法。
6. 只有枚举值和宏定义使用全大写。
7. 枚举类型和结构体类型，函数类型 都是用小写加\_t后缀。
8. 模块的接口函数声明放在.h文件中，非接口的内部函数声明放在.c文件中。
9. =，==，>,<, &,|| &&,| 等运算符号的左右各加一个空格。
10. 变量命名不要使用 a,b,c,d这种没有实际意义的名称。
11. 注释要求：
    1. 注释风格要符合doxygen风格，具体参照<https://www.jianshu.com/p/9464eca6aefe>。
    2. 每个函数都要有注释。函数被调用次数在2次以上的要写函数描述，输入参数描述，输出参数描述。
    3. 每个全局变量都要有注释。
    4. 每个结构体，枚举类型，宏定义都要有注释。
    5. 关键语句增加注释。
12. 判断条件中不要出现数字，全部使用宏定义和枚举值。
13. 函数定义和调用时，逗号后＋空格。
14. 左右大括号“{”,”}”,都另起一行写。左大括号“{”后，下一行代码前+4个空格（一个制表符）和左大括号“{”对齐。（typedef结构体时的“}”例外）。
15. 头文件比文件文件最开始引用，而不可以在代码中间引用。头文件中引用另一个头文件，应当写在“本头文件防止重复引用的宏定义”的后面。
16. 对所有两次及以上的运算语句或比较语句，要对每一个子运算添加括号，来人为指明先算哪个后算哪个。

## 2 高级要求

模块化分层，一般建议自底向上分为板级驱动层，外设驱动层，基础服务层（外设驱动的封装），应用功能模块层，应用层，系统层

* 板级驱动层：芯片库文件
* 外设驱动层：按键，灯，加速度计，外部flash，字库，摄像头，灯
* 基础服务层：对外设驱动的封装，上层只掉寄出服务层对驱动的封装，这样切换驱动时只需要添加外设驱动模块和修改相应的基础服务模块即可。
* 应用功能模块层：对整个产品的需求进行细分，每一个需求都是一个功能模块。
* 应用层：应用层应该按照产品使用场景和方式定义状态机，不同情况启动和关闭相应的功能。
* 系统层：一般包含，系统调度模块，系统定时器模块，事件队列模块。
* 全局层：定义一些通用的算法，函数，配置，比如CRC校验，内部flash存储配置信息

板级驱动层

外设驱动层

基础服务驱动层

功能模块层

应用层

系

统

层

全

局

层

1. 一般一个模块一个文件，相对复杂的模块可以分多个文件，外部接口都放在一个文件中。
2. 只可以上层调用下层的接口，下层不能直接调用上层接口
3. 下层向上层反馈只可以采用回调函数的方式或者使用系统事件队列。
4. 尽量不要跨层调用接口

## 3 SPOS系统

**Spos\_schedue**:调度器,从中断中发数据给主循环请使用spos\_post\_task（）；

**Spos\_event\_queue**：事件队列，下层传递信息给上层的一种方式，特点：事件可以同时触发多个事件，同一个事件可以被多个接收处理函数处理，事件队列是同步的，随分发随处理，说是队列，实际只是一个事件分发控制器。

**Spos\_msg\_queue**:消息队列，下层传递消息给上层的另一种方式：特点：消息是点对点的，特定的消息只能被固定的处理函数接收处理。消息队列是同步执行的，也是随抛出随处理。说是队列，实际上只是一个消息分发器。

## 4 GIT 开发同步要求

1. Git commit 要求
   1. 起标题，以版本号命名
   2. 标题后空一行，写具体的提交内容
   3. 每次commit，需要详细比较每个更改的内容，确认是否预期的改动，防止误操作导致的改动。
2. 多人协同开发时，要求每个人从主分支新建一个分支，实现每个新功能时，或者修复BUG时，按照以下步骤完成协同开发：
   1. 如果个人没有分支，首先从最新的主分支创建一个分支；
   2. 从主分支拉取最新的代码到当前分支；
   3. 在这个分支上完成开发，验证；
   4. 验证通过后，检查主分支有没有新提交
   5. 如果有新提交，从主分支拉取最新代码MERGE到当前分支，再次验证；
   6. 重复d,e直到主分支没有新提交，然后PUSH当前分支到GITLAB
3. 版本迭代规则（主要针对定位器程序，部分内容其他工程也可通用）
   1. 版本号：a.b.c.d;
   2. a:设备版本，b:主板版本，c:软件版本，d:软件子版本；
   3. 开发人员每次要开发新功能或者修改新BUG时都要从主分支拉取最新“软件版本”，并在这个软件版本上进行开发；
   4. 开发人员的分支每次提交都是且只能是“d”的增加；
   5. 开发且验证完毕，且主分支没有新的commit是，可以同步到主分支，此时可以另“c”+1，同时“d”清0；
   6. 因此主分支的版永远是“a.b.c.0”；
   7. 发布给生产的程序也只能是“a.b.c.0”
   8. 当主板有所改动，程序和当前主板不能兼容时，“b”需要+1；
   9. a,b,c,d的取值范围均为 0~255；
   10. 当c为255，仍需增加时，可以进位，b+1，c清0；

## 5 Doxygen注释举例

函数注释:注意 @brief ,@detail,@param[in],@param[out],@return,@ref,@retval都在合时使用。

|  |
| --- |
| 1. /\*\*@brief 函数简要说明 2. \* 3. \* @detail 函数详细说明 4. \* 5. \* @param[in]   in\_arg  输入参数解释 6. \* @param[out]  out\_arg 输出参数解释 7. \* 8. \* @return 返回值解释, @ref ret\_code\_t 9. \* @retval SUCCESS  返回值SUCCCESS代表什么意思的解释 10. \* @retval FALSE    返回值FALSE代表什么意思的解释 11. \*/ 12. ret\_code\_t example\_func(**int** in\_arg, **int** \*out\_arg) 13. { 14. \*out\_arg = in\_arg; 15. **if**(in\_arg) 16. **return** FALSE; 17. **else** 18. **return** SUCCESS; 19. } |

下一行注释

|  |
| --- |
| 1. //下一行注释 2. example\_func(in\_arg, &out\_arg); 4. /\* 下一行注释 \*/ 5. example\_func(in\_arg, &out\_arg); |

同行注释

|  |
| --- |
| 1. example\_func(in\_arg, &out\_arg); /\*\*< 同行注释 \*/ 2. example\_func(in\_arg, &out\_arg); //!< 同行注释 3. example\_func(in\_arg, &out\_arg); ///< 同行注释 |

大范围宏判断

|  |
| --- |
| 1. #ifdef ENABLE\_FUNCA 3. #ifdef ENABLE\_FUNCB 4. #define ENABLE\_FUNCE 5. #endif //ENABLE\_FUNCB 7. #ifdef ENABLE\_FUNCC 8. #define ENABLE\_FUNCD 9. #endif //ENABLE\_FUNCC 11. #endif //ENABLE\_FUNCA |