目　　录

1 排版

1.1 代码块采用缩进风格编写，缩进使用**4**个空格。**Tab**键设置输入**4**个空格

1.2 独立的程序块之间、变量说明之后必须加空行

1.3 较长的语句（**>80**字符）、循环、判断等语句中较长的表达式要分成多行书写，

长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要

进行适当的缩进，使排版整齐

```
if ((set_logging_filename(filename) != 0)
    || (log_err_msg(startup_msg, LOG_TRACE, 1) != 0)
   || (log_err_msg(run_msg, LOG_TRACE, 1) != 0))
{
    return false;
}
```

1.4 若函数或过程中的参数较长，则要进行适当的划分

```
bool set_up_log_messages(const std::string &filename,
    const std::string &startup_msg,
    const std::string &run_msg)
    {
        do_something();
    }
```

1.5 不允许把多个短语句写在一行中，即一行只写一条语句

1.6 **if**、**for**、**do**、**while**、**case**、**switch**、**default**等语句自占一行，且**if**、**for**、**do**、

while等语句的执行语句部分无论多少都要加括号**{}**

```
if (condition)
{
    do_something();
}
else
{
    while (true)
    {
        do_something_else();
    }
}
```

1.7 对齐只使用空格字符，不使用制表符

1.8 函数的开始、结构的定义及循环、判断等语句中的代码、**case**语句下的情况处理语句都要采用缩进风格

```
switch (season)
{
    case summer:
        surf();
        break;
    case winter:
        ski();
        break;
    default:
        swim();
        break;
}
```

1.9 结构型数组，多维数组等相似数据类型，初始化时，按照矩阵结构分行书写

```
struct point
{
    int x;
    int y;
};
struct point points[2] =
{
    {
        100,
        200,
    },
    {
        300,
        400,
    },
};
```

1.10 程序块的分界符**{}**应各独占一行并且位于同一列，同时与引用它们的语句左对齐，在函数体的开始、类的定义、结构的定义、枚举的定义以及**if**、**for**、**do**、**while**、**switch**、**case**语句中的程序都要采用如上的缩进方式

```
union Location
{
```

```
    struct Point2d
    {
        float x;
        float y;
    };

    struct Point3d
    {
        float x;
        float y;
        float z;
    };
};
```

1.11 双目操作符（**"="**、 **"+="**、**"+"**、**"%"**、**"&&"**、**"&"**按位与、**"<<"**、**"^"**等）

前后要加空格，单目操作符（**"!"**、**"~"**、**"&"**取地址、**"++"**、**"--"**等）前后不加空格，成员选择符（"->"、"."）前后不加空格

```
front = rear = NULL;
square = width * height;
++m_Count;
location->Point2d.x;
```

1.12 逗号、分号只在后面加空格

```
int x, y, z;
for (int i=0; i<100; ++i)
{
}
```

1.13 一行程序以小于**80**字符为宜

2 注释

2.1 源文件头部和说明性文件(如头文件**.h** 文件、**.inc** 文件、**.def** 文件、编译说明文件**.cfg** 等)头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者、内容、功能、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明

```
/*
*************************************************
** Filename:  WifiComm_Task.c
** Abstract:  This file implements WifiComm_Task function.
```

```
** By       :  wangbin  <letlink@126.com>
** Date     :  2016-10-24 17:09:39
** Changelog:1.First Create
*******************************************************
*/
```

2.2 函数头部应进行注释，列出：函数的目的**/**功能、输入参数、输出参数、返回值、调用关系（函数、表）等

```
/*******************************************************
    * Function Name: CreateWifiCommSemaphore
    * Description  : 创建二值信号量，用于实现 wifi 串口的独占
    * Arguments    : NONE
    * Return Value : NONE
*******************************************************/
```

2.3 边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性，不再有用的注释要删除，注释的内容要清楚、明了，含义准确，防止注释二义性

2.4 避免在注释中使用缩写，特别是非常用缩写

2.5 注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开

```
// Suspend tasks
 vTaskSuspend(g_WifiCommTaskHandle);
vTaskSuspend(g_DataManageTaskHandle);
vTaskSuspend(g_SystemErrorTaskHandle);
```

2.6 对于所有有物理含义的变量、常量，数据结构声明**(**包括数组、结构、类、枚举等**)**如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义

2.7 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明

2.8 分支语句（条件分支、循环语句等）必须编写注释

2.9 **switch** 语句下的 **case** 语句，如果因为特殊情况需要处理完一个 **case** 后进入下一个 **case** 处理，必须在该 **case** 语句处理完、下一个 **case** 语句前加上明确的注释

```
switch (cmd)
{
    case cmd_1:
        do_something(); // No break, Jump to cmd_2
    case cmd_2:
        do_cmd();
        break;

    default:
        break;
}
```

2.10 注释与所描述内容进行同样的缩排

2.11 通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的以减少不必要的注释

2.12 注释语言尽量使用中文，除非能用准确的英文表达

3 标识符命名

3.1 标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解，较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成

temp 缩写成 **tmp**

message 缩写成 **msg**

常用缩写：**Argument/Arg, Maximum/Max, Buffer/Buf, Message/Msg, Clea/Clr, Minimum/Min, Clock/Clk, Multiplex/Mux, Compare/Cmp, Operating System/OS, Configuration/Cfg, Overflow/Ovf, Context/Ctx, Parameter/Param, Delay/Dly, Pointer/Ptr, Device/Dev, Previous/Prev, Disable/Dis, Priority/Prio, Display/Disp, Read/Rd, Enable/En, Ready/Rdy, Error/Err, Register/Reg, Function/Fnct, Schedule/Sched, Hexadecimal/Hex,**

Semaphore/Sem, High Priority Task/HPT, Stack/Stk, I/O System/IOS, Synchronize/Sync, Initialize/Init, Timer/Tmr, Mailbox/Mbox, Trigger/Trig, Manager/Mgr, Write/Wr.

3.2 命名中若使用特殊约定或缩写，则要有注释说明

3.3 变量名要表明其作用范围，全局变量用 **g**_前缀，模块内全局变量用 **s**_前缀，局部变量则不需要前缀，**C++**类成员加 **m**_前缀，枚举成员加 **N**_前缀

```
TaskHandle_t g_IoManageTaskHandle;
static TS_IO_CTRL s_IoCtrl;
typedef enum {
    N_BATTERY_NORMAL = 0,
    N_BATTERY_CHARGEING,
    N_BATTERY_LOW_VOLTAGE,
    N_BATTERY_IDLE,
} TE_BATTERY_STATE;
```

3.4 命名规范与所使用的操作系统风格保持一致，如采用 **Linux** 的全小写加下划线的风格，**FreeRTOS** 的大驼峰命名风格

3.5 结构名、联合名、枚举名由前缀“**T**_”开头，**TS** 表示结构体,**TU** 表示联合体,**TE** 表示枚举

3.6 鉴于 **IDE** 的提示功能已经很强大，不建议使用匈牙利命名法

3.7 接口部分的函数名、变量与常量之前加上“模块”标识

void adc_set_ample_rate(int sample_rate);

void WIFI_SendMsg(TS_WIFI_MSG msg);

4 可读性

4.1 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级

```
if ((a | b) && (a & c))
```

4.2 避免直接使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替

```
if (battery_state == 1)
```

改为

```
typedef enum {
    N_BATTERY_NORMAL = 0,
    N_BATTERY_CHARGEING,
    N_BATTERY_LOW_VOLTAGE,
    N_BATTERY_IDLE,
} TE_BATTERY_STATE;
if (battery_state == N_BATTERY_CHARGEING)
```

4.3 源程序中关系较为紧密的代码应尽可能相邻

较差	较好
`rect.length = 10;`	`rect.length = 10;`
`char_poi = str;`	`rect.width = 5;`
`rect.width = 5;`	`char_poi = str;`

4.4 不要使用难懂的技巧性很高的语句

```
*stat_poi ++ += 1;
```

5 变量、结构

5.1 尽量减少没必要的公共变量以降低模块间的耦合度

5.2 仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系，在对变量声明的同时，应对其含义、作用及取值范围进行注释说明

5.3 当向公共变量赋值时，要十分小心，防止赋与不合理的值或越界等现象发生

5.4 防止局部变量与公共变量同名

5.5 严禁使用未经初始化的变量作为右值，特别是在 **C/C++**中引用未经赋值的指针，经常会引起系统崩溃

5.6 结构的功能要单一，是针对一种事务的抽象，结构中的各元素应代表同一事务的不

同侧面

```
typedef struct
{
    unsigned char name[8]; /* student's name */
    unsigned char age; /* student's age */
    unsigned char sex; /* student's sex, as follows */
    /* 0 - FEMALE; 1 - MALE */

    unsigned char teacher_name[8]; /* the student teacher's name */
    unisgned char teacher_sex; /* his teacher sex */
} TS_STUDENT;
```

应修改为

```
typedef struct
{
    unsigned char name[8]; /* teacher name */
    unisgned char sex; /* teacher sex, as follows */
    /* 0 - FEMALE; 1 - MALE */
} TS_TEACHER;
typedef struct
{
    unsigned char name[8]; /* student's name */
    unsigned char age; /* student's age */
    unsigned char sex; /* student's sex, as follows */
    /* 0 - FEMALE; 1 - MALE */

    unsigned int teacher_ind; /* his teacher index */
} TS_STUDENT;
```

5.7 不同结构间的关系不要过于复杂，如果两个结构间关系较复杂、密切，那么应合为一个结构

```
typedef struct
{
    unsigned char name[8];
    unsigned char addr[40];
    unsigned char sex;
    unsigned char city[15];
} TS_PERSON_ONE;
typedef struct
{
    unsigned char name[8];
    unsigned char age;
```

```
    unsigned char tel;
} TS_PERSON_TWO;
```

应合并为

```
typedef struct
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
} TS_PERSON;
```

5.8 结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构，以减少原结构中元素的个数

5.9 仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间

5.10 结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用预留一些空间等

5.11 要注意数据类型的强制转换，对编译系统默认的数据类型转换，也要有充分的认识，尽量减少没有必要的数据类型默认转换与强制转换

```
char chr;
unsigned short int exam;
chr = -1;
exam = chr; // 编译器不产生告警，此时 exam 为 0xFFFF
```

5.12 对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性

```
typedef float DISTANCE;
typedef float SCORE;
```

5.13 当声明用于分布式环境或不同 **CPU** 间通信环境的数据结构时，必须考虑机器的字节顺序、使用的位域及字节对齐等问题

6 函数

6.1 对所调用函数的错误返回码要仔细、全面地处理

6.2 明确函数功能，精确地实现函数设计

6.3 编写可重入函数时，应注意局部变量的使用，小心使用 **static** 局部变量，若使用全局变量，则应通过关中断、信号量等手段对其加以保护

6.4 应明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责 ,缺省是由函数调用者负责 ,调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程 , 造成问题隐患 , 调用者和被调用者均对参数进行合法性检查 , 这种情况虽不会造成问题 , 但产生了冗余代码 , 降低了效率

6.5 防止将函数的参数作为工作变量 , 将函数的参数作为工作变量 , 有可能错误地改变参数内容 , 所以很危险 , 对必须改变的参数 , 最好先用局部变量代之 , 最后再将该局部变量的内容赋给该参数

```
void sum_data( unsigned int num, int *data, int *sum )
{
    unsigned int count;
    *sum = 0;
    for (count = 0; count < num; count++)
    {
        *sum += data[count]; // sum 成了工作变量，不太好。
    }
}
```

6.6 函数的规模尽量限制在 **200** 行以内（不含空行和注释）

6.7 一个函数仅完成一件功能，不要设计多用途面面俱到的函数

6.8 避免设计多参数函数 , 不使用的参数从接口中去掉以减少函数间接口的复杂度 , 如果参数个数过多，可把参数组合成结构，再把结构的指针当成参数输入

```
void Function(int para_a, float para_b, char para_c);
```

改为

```
typedef struct
{
```

```
    int a;
    float b;
    char c;
} TS_FUNC_PARA;
Void Function(TS_FUNC_PARA *p_func_para);
```

6.9 检查函数所有参数输入，所有非参数输入（数据文件、公共变量等）的有效性

6.10 函数名应准确描述函数的功能，使用动宾词组为执行某操作的函数命名，避免使用无意义或含义不清的动词，如果是接口函数则加上模块名前缀

```
void print_msg(char *msg);
void adc_set_sample_rate(int sample_rate);
```

6.11函数的返回值要清楚、明了，让使用者不容易忽视错误情况，最好不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回

6.12在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换

6.13如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题，可考虑把此段代码构造成一个新的函数

6.14功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在

6.15 设计高扇入、合理扇出（小于 **7**）的函数，扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它；扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数，一般是由于缺乏中间层次，可适当增加中间层次的函数；而扇出过小，如总是 **1**，表明函数的调用层次可能过多，这样不利程序阅读和函数结构的分析，并且程序运行时会对系统资源如堆栈空间等造成压力，可把下级函数进一步分解多个函数，或合并到上级函数中

6.16 尽量避免函数本身或函数间的递归调用，递归调用影响程序的可理解性，一般都

占用较多的系统资源

6.17 仔细分析模块的功能及性能需求，并进一步细分，同时若有必要画出有关数据流图，据此来进行模块的函数划分与组织

6.18 在多任务操作系统的环境下编程，要注意函数可重入性的构造

7 可测性

7.1 有统一的为集成测试与系统联调准备的调测开关及相应打印函数，并且要有详细的说明，调测打印出的信息串的格式要有统一的形式，信息串中至少要有所在模块名（或源文件名）及行号

```
#define DBG(fmt, …) do \
{ \
    printf(“%s %d ”fmt ,__FILE__, ___LINE__, ##__VA_ARGS__); \
} while(0)
```

7.2 软件的 **Debug** 版本和 **Release** 版本应该统一维护，不允许分家，并且要时刻注意保证两个版本在实现功能上的一致性

8 程序效率

8.1 在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率

8.2 局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响

8.3 通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率

8.4 循环体内工作量最小化，避免循环体内含判断语句，在多重循环中，应将最忙的循环放在最内层，尽量减少循环嵌套层次

```
for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
{
    if (data_type == RECT_AREA)
    {
        area_sum += rect_area[ind];
```

```
        }
        else
        {
            rect_length_sum += rect[ind].length;
            rect_width_sum += rect[ind].width;
        }
    }
```

应改为

```
    if (data_type == RECT_AREA)
    {
        for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
        {
            area_sum += rect_area[ind];
        }
    }
    else
    {
        for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
        {
            rect_length_sum += rect[ind].length;
            rect_width_sum += rect[ind].width;
        }
}
```

8.5 对模块中函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高程序效率

8.6 仔细分析有关算法，并进行优化

8.7 不应花过多的时间拼命地提高调用不很频繁的函数代码效率

8.8 尽量用乘法或其它方法代替除法，特别是浮点运算中的除法

```
#define PAI 3.1416
radius = circle_length / (2 * PAI);
```

应如下把浮点除法改为浮点乘法。

```
#define PAI_RECIPROCAL (1 / 3.1416 ) // 编译器将生成具体浮点数
radius = circle_length * PAI_RECIPROCAL / 2;
```

8.9 模 **2** 的乘除法可以用移位代替

`X / 8` 可以用 `x >> 3` 代替

X * 8 可以用 x << 3 代替

9 质量保证

9.1 防止引用已经释放的内存空间

9.2 函数中分配的内存，函数退出之前要释放

9.3 防止内存操作越界

```
#define MAX_USR_NUM 10
unsigned char usr_login_flg[MAX_USR_NUM]= "";
```

当下标用 10 时就越界了

9.4 认真处理程序所能遇到的各种出错情况

9.5 系统运行之初，要初始化有关变量及运行环境，防止未经初始化的变量被引用

9.6 系统运行之初，要对加载到系统中的数据进行一致性检查

9.7 严禁随意更改其它模块或系统的有关设置和配置如常量、数组的大小等

9.8 不能随意改变与其它模块的接口

9.9 编程时，要防止差 **1** 错误，如数组边界访问错误

9.10 尽量使用库函数

9.11 注意易混淆的操作符，如”**=**”与“**==**，“**|**”与“**||**”、“**&**”与“**&&**”等

```
ret_flg = (pmsg->ret_flg & RETURN_MASK);
被写为：
ret_flg = (pmsg->ret_flg && RETURN_MASK);
```

9.12 **if** 语句尽量加上 **else** 分支，**switch** 语句必须有 **default** 分支

9.13 不要滥用 **goto** 语句

9.14 除非为了满足特殊需求，避免使用嵌入式汇编

9.15 精心地构造、划分子模块，并按“接口”部分及“内核”部分合理地组织子模块，以提高“内核”部分的可移植性和可重用性

9.16 对较关键的算法最好使用其它算法来确认

9.17 注意表达式是否会上溢、下溢

```
unsigned char size ;

while (size-- >= 0) // 将出现下溢

{
    ... // program code
}
```

当 size 等于 0 时，再减 1 不会小于 0，而是 0xFF，是死循环

应改为

```
signed char size; // 从 unsigned char 改为 signed char

while (size-- >= 0)
{
    ... // program code
}
```

9.18 使用变量时要注意其边界值的情况

如 C 语言中字符型变量，有效值范围为-128 到 127。故以下表达式的计算存在一定风险。

```
char chr = 127;
int sum = 200;

chr += 1; // 127 为 chr 的边界值，再加 1 将使 chr 上溢到-128，而不是 128。

sum += chr; // 故 sum 的结果不是 328，而是 72
```

9.19 使用第三方提供的软件开发工具包或控件时，要注意充分了解应用接口、使用环境及使用时注意事项；不能过分相信其正确性；除非必要，不要使用不熟悉的第三方工具包与控件

10 代码编辑、编译

10.1 打开编译器的所有告警开关对程序进行编译

10.2 在项目中，要统一编译开关选项

10.3 最好使用相同的编辑器，并使用相同的设置选项

10.4 合理地设计软件系统目录，方便开发人员使用

10.5 使用静态代码检查工具（如 ***PC-Lint***）对源程序检查

11 宏

11.1 用宏定义表达式时，要使用完备的括号

```
#define RECTANGLE_AREA( a, b ) ((a) * (b))
```

11.2 将宏所定义的多条表达式放在 **do {} while (0)**中

```
#define XXX(x) do { \
    Do_something(); \
    Do_somethingElse(); \
} while (0)
```

11.3 使用宏时不允许参数发生变化

11.4 宏命名用全大写加下划线分隔的风格，配置类宏命名用下划线前缀

```
#define _CONFIG_USING_QUEUE      (1)
```

11.5 对于宏定义的常数，必须指出其类型

```
#define MAX_COUNT     ((int) 100)
```