一、编码与排版

1. 程序块要采用缩进风格编写，缩进的空格数为 4 个。
2. 相对独立的程序块之间、变量之后必须加空行。

```
RtcGet(&rtcObj);

SD[SD_CLOCK_SECOND] = rtcObj.Seconds;
SD[SD_CLOCK_MINUTE] = rtcObj.Minutes;
SD[SD_CLOCK_HOUR] = rtcObj.Hours;
SD[SD_CLOCK_DATE] = rtcObj.Day;
SD[SD_CLOCK_MONTH] = rtcObj.Month;
SD[SD_CLOCK_YEAR] = rtcObj.Year - 2000;
SD[SD_CLOCK_WEEK] = rtcObj.WeekDay;

RtcUpdateFlag = FALSE;
```

3. 较长的语句(>80 字符)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，语句可读。

示例：

```
ActTaskTable[frameId * STAT_TASK_CHECK_NUMBER + index].occupied
                = statPoi[index].occupied;
```

4. 循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，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。

示例：

```
for (i = 0, j = 0;
     (i < firstWordLength) && (j < secondWordLength);
      i++, j++)
{
     ../program code
}
```

5. 若函数或过程中的参数较长，则要进行适当的划分。

```
CmdAdd("uwdg", "list user watchdog information",  \
       CMD_INDENT_USAGE("uwdg\r\n") \
       CMD_INDENT_USAGE("uwdg dump|enable|disable"), UserWDGCommand, NULL);
```

6. 不允许把多个短语句写在一行中，即一行只写一条语句。
7. 对齐只使用空格键，不使用 TAB 键。
8. 函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case 语句下的情况处理语句也要遵从语句缩进要求。
9. 程序块的分界符(如 C/C++语言的大括号”{}”)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 for、do、while、switch、case 语句中的程序都要采用如上的缩进方式。

10. 在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格：进行非对等操作时，如果是关系密切的立即操作符(如->)，后不应加空格。

示例：

(1)逗号、分号只在后面加空格。

```
int a, b, c;
```

(2)比较操作符，赋值操作符"="、"+="，算术操作符"+"、"%"，逻辑操作符"&&"、"&"，位域操作符"<<"、"^"等双目操作符的前后加空格。

```
if (currentTime > MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

(3)"!"、"~"、"++"、"--"、"&"（地址运算符）等单目操作符前后不加空格。

```
*p = 'a';
flag = !isEmpty;
p = &mem;
```

(4)"->"、"."前后不加空格。

```
p->id = pid;
```

(5)if、for、while、switch等与后面的括号间应加空格，使if等关键字更为突出、明显。

```
if (a == b)
```

11. 同一个项目采用统一的编码格式，建议采用 UTF8。

12. 要去除行尾空格，行尾空格可能带来上传到 git 上时，无效的差异

二、变量命名

1. 变量的命名要清晰、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。

```
示例：如下单词的缩写能够被大家基本认可。
temp 可缩写为  tmp  ;
flag 可缩写为  flg  ;
statistic 可缩写为  stat ;
increment 可缩写为  inc  ;
message 可缩写为  msg  ;
```

2. 命名中若使用特殊约定或缩写，则要有注释说明。
3. 对于变量命名，禁止取单个字符(如 i、j、k),建议除了要有具体含义外还能表明其变量类型、数据类型等，但引“i,j,k”等作局部循环变量是允许的。
4. 除非必要，不要用数字或奇怪的字符来定义标识符

```
示例：如下命名，使人产生疑惑。
#define _EXAMPLE_0_TEST_
#define _EXAMPLE_1_TEST_
void SetSls00(BYTE sls );

应改为有意义的单词命名
#define _EXAMPLE_UNIT_TEST_
#define _EXAMPLE_ASSERT_TEST_
void SetUdtMsgSls(BYTE s1s )
```

5. 在同一软件产品内，应规划好接口部分标识符(变量、结构、函数及常量)的命名，防止编译、链接时产生冲突。
6. 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

```
说明：下面是一些在软件中常用的反义词组。
add / remove       begin / end        create / destroy
insert / delete    first / last       get / release
increment / decrement                 put / get
add / delete       lock / unlock      open / close
min / max          old / new          start / stop
next / previous    source / target    show / hide
send / receive     source / destination
cut / paste        up / down
```

7. 除了编译开关/头文件等特殊应用，应避免使用 EXAMPLE TEST 之类以下划线开始和结尾的定义。
8. 没有必要时不要使用公共变量。
9. 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。
10. 当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生。
11. 防止局部变量与公共变量同名。

12. 严禁使用未经初始化的变量作为右值。
13. 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
14. 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。
15. 局部变量使用小驼峰命名法。函数名，全局变量使用大驼峰命名法。

```
// 全局变量
ExceptionCB ApplicationCBFunc = NULL;

void ExceptionGetVersion(CHAR *version, INT32U len){ ... }

#define EXEPTION_FILE_DAYS  7
static void DeleteOldExceptionFiles(CHAR *dir)
{
    CHAR **fileNames;
    time_t tp;
    time(&tp);
    struct tm *pTm = localtime(&tp);
    INT32S thresholdYMD = (1900 + pTm->tm_year) * 10000 + (1 + pTm->tm_mon) * 100 + pTm->tm_mday;
    thresholdYMD -= EXEPTION_FILE_DAYS;      /* 保留指定天数的文件 */
```

16. 结构体，枚举类，共用体命名全部大写，并且两个单词中间以下划线分割。

```
struct USRINT_MSG{
    enum USRINT_TYPE Type;
    USRINT_CALLBACK CallBack;
    void *Data;
};

enum USRINT_STAT{
    USRINT_STAT_NON,
    USRINT_STAT_INIT,
    USRINT_STAT_STOP,
    USRINT_STAT_RUN,
    MAX_USRINT_STAT,
};
```

三、函数与过程

1．对所调用函数的错误返回码要仔细、全面地处理。
2．明确函数功能，精确（而不是近似）地实现函数设计。
3．编写可重入函数时，应注意局部变量的使用
4．编写可重入函数时，若使用全局变量，则应通过关中断、信号量(即 P、V 操作)等手段对其加以保护。
5．在同一项目组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。
6．防止将函数的参数作为工作变量。

示例：下函数的实现不太好。

```
void SumData(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。*/
    }
}
```

若改为如下，则更好些。

```
void SumData(unsigned int num, int *data, int *sum)
{
    unsigned int count;
    int sumTemp;
    sumTemp = 0;
    for (count 0; count < num; count++)
    {
        sumTemp += data[count];
    }
    *sum = sumTemp;
}
```

7．函数的规模尽量限制在 200 行以内。
8．一个函数仅完成一件功能。
9．为简单功能编写函数。

示例：如下语句的功能不明显。

```
value = ( a > b ) ? a : b;
```

改为如下就很清晰了。

```
int Max(int a, int b)
{
    return ((a > b) ? a : b);
}
value = Max(a, b);
```

10．不要设计多用途面面俱到的函数。

11．函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。

示例：如下函数，其返回值（即功能）是不可预测的。

```
unsigned int Integer_Sum(unsigned int base)
{
    unsigned int index;
    static unsigned int sum = 0;        /*注意，是static类型的。*/
                                        /*若改为auto类型，则函数即变为可预测。*/
    for (index = 1; index <= base; index++)
    {
        sum += index;
    }
    return sum;
}
```

12．避免设计多参数函数，不使用的参数从接口中去掉。

13．非调度函数应减少或防止控制参数，尽量只使用数据参数。

示例：如下函数构造不太合理。

```
int AddSub(int a, int b, unsigned char addSubFlg)
{
    if (addSubFlg == INTEGER_ADD)
    {
        return (a + b);
    }
    else
    {
        return (a - b);
    }
```

不如分为如下两个函数清晰。

```
int Add(int a, int b)
    return (a + b);
}

int Sub(int a, int b)
    return (a + b);
}
```

14. 检查函数所有参数输入的有效性
15. 检查函数所有非参数输入的有效性，如数据文件、公共变量等。
16. 函数名应准确描述函数的功能。
17. 使用动宾词组为执行某操作的函数命名。如果是 OOP 方法，可以只有动词（名词是对像本身）。

示例：参照如下方式命名函数。

```
void PrintRecord(unsigned int recInd);
int InputRecord(void);
unsigned char GetCurrentColor(void);
```

18. 避免使用无意义或含义不清的动词为函数命名。
19. 函数的返回值要清楚、明了，让使用者不容易忽视错误情况。
20. 除非必要，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。
21. 让函数在调用点显得易懂、容易理解。
22. 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
23. 避免函数中不必要语句，防止程序中的垃圾代码。
24. 防止把没有关联的语句放到一个函数中。

示例：如下函数就是一种随机内聚。

```
void InitVar(void){
    Rect.length = 0;
    Rect.width = 0;      /*初始化矩形的长与宽*/

    Point.x = 10;
    Point.y = 10;        /*初始化“点”的坐标*/
}
```

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。
应如下分为两个函数：

```
void InitRect(void)
{
    Rect.length = 0;
    Rect,width = 0;      /*初始化矩形的长与宽*/
}

void Init Point(void)
{
    Point.x = 10;
    Point.y = 10;        /*初始化“点”的坐标*/
}
```

25. 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。
26. 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。
27. 设计高扇入、合理扇出(小于 7)的函数。
28. 减少函数本身或函数间的递归调用。
29. 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。
30. 改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。
31. 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。
32. 避免使用 BOOL 参数。
33. 对于提供了返回值的函数，在引用时最好使用其返回值。
34. 当一个过程（函数）中对较长变量（一般是结构的成员）有较多引用时，可以用一个意义相当的宏代替。

四、宏定义

1. 用宏定义表达式时，要使用完备的括号。

```
#define BLOG_USER_DATA_NUM          (BLOG_USER_MEMORY_8M / (BLOG_USER_DATA_SIZE + 24))
```

2. 将宏定义的多条表达式放在大括号中。

```
#define BLOG_STRING_T(format, ...) \
        do { \
            snprintf(paramData, PARAM_MAX_LEN - 1, (CHAR *)"[(%d) %s,%s,%d]" format, BLogGetThreadId(), __FILE__, __func__, __LINE__, ##__VA_ARGS__); \
            len = strlen(paramData); \
        }while(0)
```

3. 使用宏时，不允许参数发生变化。

```
#define  SQUARE(a) ((a) * (a))

int a = 5;
int b;
b = SQUARE(a++);    /*结果：a = 7，执行了两次增1*/

正确用法是：
b = SQUARE(a);
a++;              /*结果：a = 6，只执行了一次增1*/
```

五、注释

1. 头部注释推荐使用 doxygen 风格进行注释编写，这样方便文档生成。

```
/**
 * @file testdoxygen.c
 * @author your name (you@domain.com)
 * @brief
 * @version 0.1
 * @date 2019-01-23
 *
 * @copyright Copyright (c) 2019
 *
 */
```

2. 函数的注释方法，根据参数和返回值来编写，参数要注明输入还是输出，返回值如果有不同的含义需要注明一下。

```
/**
 * @brief          初始化
 * @details
 *
 * @param[in]      memSize 内存池的总空间(单位：字节)，xlib中使用的堆空间都是从这个内存池中分配。
 *
 * @return         BOOL    初始化结果
 * @retval         TRUE    成功
 * @retval         FALSE   失败
 *
 * @note
 * @attention
 */
XLIB_API BOOL XLibInit(XLIB_INIT_PARAM initParam);
```

3. 代码之间的注释分割线使用这种多行注释的风格。

```
/*****************************************************************************
 *      软元件编码
 *****************************************************************************/
```

4. 关键代码的解释使用多行注释风格。

```
/* 检查是否有冒号，表示该数据需要重复多次 */
pSemicolon = strstr(tmp, ":");

INT32U val;
if (pSemicolon)
    *pSemicolon = 0;

/* 配置数据可能是十进制，也可能是十六进制 */
if (strstr(tmp, "0x") || strstr(tmp, "0X"))
```

5. 结构体，枚举类，共用体也是使用 doxygen 风格，结构内部的变量使用“///<”在同行进行注释。

```
typedef enum{
    BLOG_LEVEL_DEBUG = 1,
    BLOG_LEVEL_INFO  = 2,
    BLOG_LEVEL_WARN  = 3,
    BLOG_LEVEL_ERROR = 4
}BLOG_LEVEL;

/**
* @brief            日志元数据，24字节
* @details
* @note
* @attention
*/
typedef struct
{
    INT16U  mainIndex;                  ///< 模块主索引
    INT16U  subIndex;                   ///< 模块次索引
    INT16U  eventId;                    ///< 模块事件Id
    INT16U  level;                      ///< 级别
    INT32U  YMD;                        ///< 年月日
    INT32U  HMS;                        ///< 时分秒
    INT32U  us;                         ///< 微秒
    INT32U  dataSize;                   ///< 用户数据长度
    CHAR    data[];                     ///< 用户数据
}BLOG_ITEMDATA;
```

6. 变量注释应使用多行注释格式并与对应代码在同一行。

```
INT32U pos = 0;                 /* 开始索引 */
memset(SSFD + pos, -1, sizeof(INT16U) * 80);                    /* m210 configure */
```

六、可读性

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。

```
logData->YMD = (((tmLocal.tm_year - 100) & 0xFF) << 16) | (((tmLocal.tm_mon + 1) & 0xFF) << 8) | (tmLocal.tm_mday & 0xFF);
logData->HMS = ((tmLocal.tm_hour & 0xFF) << 16) | ((tmLocal.tm_min & 0xFF) << 8) | (tmLocal.tm_sec & 0xFF);
```

2. 避免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 (Trunk[index].trunkState = 0)
{
    Trunk[index].trunkState = 1;
    ../program code
}
```

应改为如下形式。

```
#define TRUNK_IDLE 0
#define TRUNK_BUSY 1
if (Trunk[index].trunkState = TRUNK_IDLE)
{
    Trunk[index].trunkState = TRUNK_BUSY;
    ../program code
}
```

3. 源程序中关系较为紧密的代码应尽可能相邻。

示例：以下代码布局不太合理。

```
RECT.length = 10;
charPoi = str;
RECT.width = 5;
```

若按如下形式书写，可能更清晰一些。

```
RECT.length = 10;
RECT.width = 5;                    /*柜形的长与宽关系较密切，放在一起*/
charPoi = str;
```

4. 不要使用难懂的技巧性很高的语句，除非很有必要时。高技巧不代表高效率。

七、可测性

1. 在同一项目组或产品组内，要有一套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。

2. 在同一项目组或产品组内，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。

3. 编程的同时要为单元测试选择恰当的测试点，并仔细构造测试代码、测试用例，同时给

出明确的注释说明。测试代码部分应作为（模块中的）一个子模块，以方便测试代码在模块中的安装与拆卸（通过调测开关）

4. 在进行集成测试/系统联调之前，要构造好测试环境、测试项目及测试用例，同时仔细分析并优化测试用例，以提高测试效率。

5. 使用断言来发现软件问题，提高代码可测性。