致远电子软件规范-C 代码规范

发布 *latest+2606c771*

广州致远电子有限公司

2017 年 7 月 20 日

目录

箴言

程序设计绝对是一门艺术，而不仅仅是一门技术。首先，程序设计的出发点是给别人看，可读、易理解、好维护。如果你的程序只能自己来维护，到你离开这个程序时，你程序也就与你一起离开了这个世界。为了可读、易理解、好维护，你的程序要有好的设计，而不是一接手就进行东抄抄、西抄抄的写代码工作。写代码是一个工程，程序设计是一种艺术；如果程序员只注重代码，那就像盖房子时的砌砖匠，只知道照图施工而已。世界那么多令人叹为观止的美丽建筑物，那是设计家的艺术杰作，而不是砌砖匠的艺术杰作。代码规范中的很多内容，是前人在很多失败的惨痛教训中，总结出来的宝贵经验，有助于我们提高编程质量，增强程序的可读性以及继承性。

1 变量、结构、类型

1.1 变量初始化

不合理的初始化数据是产生编程错误的常见根源。不恰当的变量初始化所导致的一系列问题都源于变量的默认初始值与预期的不同。70% 软件问题都源于内存错误，指针变量是错误的核心。

要求：

1. 尽量在定义变量的时候初始化。
2. 局部指针变量必须初始化。
3. 在使用全局变量时尽量进行有效范围检查，使用指针型全局变量必须进行有效性检查。
4. 尽量在靠近第一次使用变量的地方声明和定义变量。例如，全局变量可以定义在初始化该变量的文件中。
5. 尽量避免使用全局变量，如果只在一个函数内使用的全局变量，尽量使用 `static` 变量，如 列表 1.1 所示：

列表 1.1: static 变量使用范例

```
int g_state = 0;    /* 状态变量 */

void led_swap (void)
{
    if (g_state == 0) {
        g_state =  1;
        led_on();
    } else {
        g_state =  0;
        led_off();
    }
}

/* 以上程序应该改为如下程序 */
```

```
void led_swap (void)
{
    static int  state = 0;

    if (state == 0) {
        state =  1;
        led_on();
    } else {
        state =  0;
        led_off();
    }
}
```

6. 与第五条类似，如果某全局变量只在一个文件中使用，应定义为 static 变量。

> **注意：** static 变量会导致函数不可重入，同时也限制了模块不能多实例，更推荐的做法是使用 OO 方法来设计模块，即使用结构体来封装变量。

1.2 变量命名

为变量命名时最重要的考虑事项是，该名字要完全、准确地描述出该变量所代表的事物。这个名字应该便于阅读，容易记忆，不能产生歧义。变量名的长度，最佳的变量名长度应该在 8 到 16 个字符之间，当变量名太短时，需要花费大量的时间来判断变量的含义。单字符的变量只能用于循环变量或者数组下标。

要求：

全部使用小写字符，必要时可添加下划线。变量名最多由三部分组成：作用域、类型、描述。

1. 作用域：该变量的作用范围，确定该变量的有效范围是在函数体外还是函数体内。
2. 类型：该变量的类型，使用小写字符。只有指针（包括函数指针）类型变量需要在命名中使用类型。
3. 描述：要完全、准确地描述出该变量所代表的事物，例如：max 、error 、new 等等。

1.2.1 全局变量的定义

全局变量的命名以 g 开头，例如：g_max。如果使用指针类型，则在 g 后面加小写 p ，例如：gp_value 。如果是函数指针类型，则在 g 后面加小写 pfn ，例如：gpfn_io_read 。静态全局变量需要加上 __ 前缀，例如，__g_max 、__gp_value 、__gpfn_io_read 。

变量类型后紧跟变量的描述，变量的描述使用小写加下划线，例如：name、error_number 等等。

要求:

1. 禁止使用拼音作为变量的描述。
2. 变量名全部使用小写加下划线，不允许使用大小写混合方式。
3. 函数指针类型变量统一使用 `pfn` 类型缩写。
4. 为了避免模块与用户程序或其他模块产生命名冲突，模块内部使用的全局变量命名中还应加上特有的标识作为命名空间。

下面给出一些模块内部全局变量的定义范例。例如，我们提供给用户一个直流电机控制软件包，软件包内定义了一些全局变量供我们自己使用，同时为了防止与用户程序或其他模块产生命名冲突，我们在定义模块内全局变量时把 `motor` 作为命名空间，如 列表 1.2 所示。

列表 1.2: 模块内部全局变量定义

```
int                     g_motor_max;                    /* 最大值 */

static unsigned int   __g_motor_error_number;           /* 错误号 */
static signed int     __g_motor_start_value = 0;        /* 起始数值 */

static char           __g_motor_first;                  /* 第一个字符 */
static char           __g_motor_name[] = "zlg mcu";     /* 名字字符串 */
```

其他要求：

1. 在连续定义多个全局变量的时候必须对齐。
2. 右边加入对每一个全局变量作用的行注释。
3. 不允许使用 `TAB` 键 作缩进。（缩进单位为 4 个空格）
4. 在不同类型全局变量或不同含义的全局变量定义之间要加入空行。如下面的程序清单所示，在记录超时时间的三个变量和记录数据报个数的三个变量定义间加一个空行。

列表 1.3: 变量定义中空行的使用范例

```
static unsigned long        __g_ip_default_timeout;  /* IP 数据报超时时间 */
static unsigned long        __g_tcp_default_timeout; /* TCP 数据报超时时间 */
static unsigned long        __g_udp_default_timeout; /* UDP 数据报超时时间 */

static unsigned long        __g_ip_pack_counter;     /* IP 数据报计数器 */
static unsigned long        __g_udp_pack_counter;    /* UDP 数据报计数器 */
static unsigned long        __g_tcp_pack_counter;    /* TCP 数据报计数器 */
```

5. 所有程序中，注释必须使用封闭式注释：`/* ···*/` 。
6. 在变量后空至少 1 个空格开始行注释，行注释起始以相同类型变量中最靠右的为基准对齐，行注释结束不必对齐，但行注释结束处不能超过第 80 列。
7. 行注释的第一个字符与 `/*` 符号间至少保留一个空格。

8. 变量类型必须写在一行的起始位置，除非含有 extern 关键字或者其他宏定义。

模块中指针型全局变量的定义与对齐，如 列表 1.4 所示。

列表 1.4: 模块中指针型全局变量的定义与对齐范例：

```
static int   __g_max;   /* 最大值 */
static int  *__gp_Max;  /* 最大值指针 */
static int **__gpp_max; /* 最大值双指针 */
```

要求：

1. 对齐的基准是英文字符或下划线。
2. 指针变量的类型字符前缀 p 紧跟作用域字符 g 。
3. 指针变量标志 * 必须紧跟变量名。例外情况，如 列表 1.5 所示。

列表 1.5: * 与变量名不贴近的例外情况范例

```
int * const  ptr2;
```

4. 禁止使用三重以上的指针。

要求：

1. 变量有初始化时，赋值符号 = 左右必须使用空格。
2. 左端的空格的个数以上下对齐为准。
3. 在保证右端对齐的情况下，右端的空格数力求最少（不少于一个）。

指针型全局变量的定义，与非指针全局变量的对齐，如 列表 1.6 所示。

列表 1.6: 指针型全局变量的定义与对齐范例

```
int       g_max;    /**< 最大值 */
int      *gp_max;   /**< 最大值指针 */
int     **gpp_max;  /**< 最大值双指针 */
```

含有 extern 关键字或者其他宏定义的变量书写范例，如 列表 1.7 所示。

列表 1.7: 含有 extern 关键字或者其他宏定义的变量书写范例

```
#define EXT       extern

extern int        g_max;    /**< 最大值 */
extern int       *gp_max;   /**< 最大值指针 */
extern int      **gpp_max;  /**< 最大值双指针 */

EXT    int        g_mix;    /**< 最小值 */
EXT    int       *gp_mix;   /**< 最小值指针 */
EXT    int      **gpp_mix;  /**< 最小值双指针 */
```

要求：

1. extern 关键字或者其他宏定义必须写在一行的起始位置。
2. extern 关键字或者其他宏定义后面至少加一个空格。
3. 同类型或含义类似变量类型字段应该对齐。

1.2.2 局部变量的定义

局部变量的定义与全局变量基本相同，但是，局部变量前不加双下划线 __ 和作用域字符。只包含指针类型和变量的描述。下面给出一些全局变量的定义范例。如 列表 1.8 所示。

列表 1.8: 局部变量定义范例

```
int      sheep_number;          /* 绵羊的数量 */
int     *p_car_number = NULL;   /* 指向车辆数目的指针 */
```

要求：

1. 局部变量只包含指针变量类型和变量描述这两部分内容。
2. 定义局部指针型变量时，必须初始化为 NULL 或者一个有意义的地址，以减少发生错误的可能性。不允许出现悬空指针（指向无效地址的指针，俗称：野指针）。
3. 指针变量标志 * 必须紧跟变量名。
4. 变量名对齐时，指针符号应该向左移动，以英文字母或下划线作为起始对齐点。
5. 在不同类型局部变量或不同含义的局部变量定义之间要加入空行。
6. 函数的第一行代码与最后一个局部变量定义之间至少保留一个空行。当变量定义中使用空行时，第一行代码与最后一个局部变量定义之间保留两个空行，其它情况保留一个空行。
7. 静态局部变量命名以 s_ 开头。

指针型变量定义，如 列表 1.9 所示。

列表 1.9: 指针型局部变量定义范例

```
char     *p_name         = NULL;     /* 名字指针 */
int     **pp_car_number  = NULL;     /* 指向汽车数量指针地址的指针 */
```

局部变量中的特例：循环变量与数组下标变量。循环变量与数组下标变量允许超越以上要求，允许使用单字符变量名。如 列表 1.10 所示。

列表 1.10: 局部循环变量与数组下标变量定义

```
int     i;  /* 循环变量 */
short   s;  /* 数组下标 */
```

静态局部变量的定义如 列表 1.11 所示

列表 1.11: 静态局部变量定义

```
int motor_init (void)
{
    static bool  s_inited = FALSE;  /* 模块已初始化标识 */

    int          counter  = 0;      /* 计数器 */


    if (s_inited == TRUE) {
        return -1;
    }
    s_inited = TRUE;
    ...
}
```

1.3 结构变量与类型

1.3.1 模块内结构体命名与定义

为了不污染用户命名空间，非应用程序（模块内部）使用的结构体与类型必须以双下划线开始 __ 。下面是一个非应用程序结构体定义范例，如 列表 1.12 所示。

列表 1.12: 模块内部结构体变量定义范例

```
struct __dlist {
    int                  value;         /* 节点的数值 */

    struct __dlist      *p_next;        /* 双链表前向指针 */
    struct __dlist      *p_prev;        /* 双链表后向指针 */
};
typedef struct __dlist   __dlist_t;     /* 定义类型 */

/* 需要使用 struct __dlist 的结构体 */
struct __my_app_struct {
    struct __dlist   dlist_node;        /* 链表节点 */
    struct __dlist  *p_dlist_head;      /* 链表头指针 */
    void           (*pfn_act)(void);    /* 动作函数指针 */
};
```

如 列表 1.12 所示。结构体的名字应该能够准确描述出该变量所代表的事物，这里将双链表定义为 `__dlist`，清晰易懂。结构体变量名中间允许插入下划线。

组合关键字中间为一个空格，例如：`unsigned char`、`typedef struct` 等等。

要求：

1. `struct` 关键字必须写在一行的起始位置，一般在 `struct` 上部还有对该结构体全局性的描述，采用块注释结构。
2. `struct` 后只能加一个空格，然后键入结构体名。
3. 结构体名称必须为小写，非应用程序（模块内部）使用的结构体名字前，必须有双下划线 `__`。
4. 结构体名字的后面加一个空格，然后键入 `{`。
5. 结构体成员类型写在起始位置后四个空格处。
6. 一个结构体所有成员必须左对齐，以英文字母或下划线作为起始对齐点。
7. 在不同含义或不同类型的成员定义之间要加入空行。
8. 每个成员右边要跟有行注释。
9. 在最后一个成员的下一行起始位置写入：`};`。
10. 不允许在结构体中使用位域。

如果程序中，需要使用类型来访问结构体，那么类型的定义必须紧跟结构体的定义，建议使用结构体名后跟 `_t` 来命名结构体类型，如 `__dlist_t`。

不建议定义结构体类型的指针类型，除非移植等确实需要。

1.3.2 应用程序内结构体命名与定义

在应用程序中定义的结构体前面不需要加入双下划线：`__`，如 列表 1.13 所示。

列表 1.13: 应用程序结构体变量定义范例

```
struct dlist {
    int                  value;      /* 节点的数值 */

    struct dlist        *p_next;     /* 双链表前向指针 */
    struct dlist        *p_prev;     /* 双链表后向指针 */
};
typedef struct dlist     dlist_t;    /* 定义类型 */

/* 需要使用 struct dlist 的结构体 */
struct my_app_struct {
    struct dlist   dlist_node;       /* 链表节点 */
    struct dlist  *p_dlist_head;     /* 链表头指针 */
    void         (*pfn_act)(void);   /* 动作函数指针 */
```

```
};

/*******************************************************************************
  双链表表头指针全局变量定义
*******************************************************************************/
dlist_t     *gp_dlist_header;       /* 表头指针 */
```

要求：

1. 当一个结构在程序中出现次数非常多，应该将此结构体定义为类型。
2. 对于编译器或 CPU 冲突的 C 类型，也可以写成结构，建议使用标准 C 已经定义的类型或结构，例如：`uint8_t` 、`uint32_t` 等（在标准头文件 `stdint.h` 中）。
3. 严禁使用 `#define` 语句将 C 类型定义为宏。如 列表 1.14 所示，就会出现严重错误。

列表 1.14: #define 语句定义的宏类型错误范例

```
#define PINT        int*

PINT    p_a, p_b, p_c;  /* 定义三个指针变量 */
```

以上代码只有 `p_a` 为指针变量，其余都为 `int` 型变量。

4. 如果定义的是结构体类型，类型名尽量与结构体名统一。

下面给出部分类型定义范例，如 列表 1.15 所示。（模块内部类型前面加双下划线 `__` ）

列表 1.15: 类型定义范例

```
typedef struct list           list_t;
typedef struct udp_header     udp_header_t;
typedef void                (*task_t)(void *);
```

数据变量的类型定义要求：

1. `typedef` 关键字必须写在一行的起始处。
2. `typedef` 关键字后加一个空格，然后紧跟原始类型名。
3. 新定义的类型名要求左对齐，以英文字母或下划线作为起始对齐点。
4. 每个类型定义要有相关类型说明的行注释。
5. 当语句长度超过 80 个字符时，可以将行注释独立一行，放在上边。（语句后加上空行）
6. 类型名必须全部小写，且必须是两个字符以上，后跟 `_t` 结束。
7. 与结构体相同，类型名中间允许使用下划线。

函数类型及函数指针类型定义附加要求：

1. 函数或函数指针类型的类型名，与变量类型名要求对齐。以英文字母或下划线作为起始对齐点。
2. 函数的返回类型名要求与变量类型定义的原始类型名对齐。
3. 函数参数括号要紧贴函数类型名。

其他函数定义及声明标准，见后面章节。

2 语句、语句块、赋值与运算符

2.1 语句

语句是组成函数与算法描述的最基本单元。关于花括号的缩进方式，采用 AT&T 实验室 K&R 风格。详细情况请阅读 《**Recommended C style and coding standards**》 。

要求：

1. 语句要采用缩进风格编写，缩进的空格数为 4 个。
2. 关键字（除过 `sizeof` 和函数调用）后凡是带有括号 `()`，在关键字和括号之间要加入一个空格。括号后如有大括号，必须在大括号与 `)` 之间加入一个空格。
3. 较长的语句（>80 个字符）必须分成多行书写，长表达式必须在低优先级操作符处划分新行。划分出的新行要进行适当的缩进，使排版整齐，语句可读。如 列表 2.1 所示。

列表 2.1: 长语句划分范例

```
mem_msg_buffer[i].len = NO7_TO_STAT_PERM_COUNT_LEN
                      + (STAT_SIZE_FILE_LEN * sizeof(char))
                      + (STAT_SIZE_PER_FRAM * sizeof(__MEM_PAGE));
```

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，操作符放在末尾。如 列表 2.2 所示。

列表 2.2: 循环、判断长语句划分范例

```
if (p_dest_ip[0] == p_temp[0] &&
    p_dest_ip[1] == p_temp[1] &&
    p_dest_ip[2] == p_temp[2] &&
    p_dest_ip[3] == p_temp[3]) {     /* 判断 IP 地址 */

    __eth_frame_send(...);           /* 发送数据包 */
}
```

```
for (i = 0;
     (i < 10 && p_temp_name != NULL);
     i += 2) {                          /* for 循环语句 */

    ...
}
```

5. 不允许把多个短语句写在一行中，即一行只写一条语句。
6. if 、for 、do 、while 、case 、switch 、default``等语句与相关条件和大括号自占一行，且``if 、for 、do 、while 等语句的执行语句部分无论多少都要加括号 {} 。
7. 对齐只使用空格键，不使用 TAB 键 。
8. 一行代码必须小于 80 字符。

2.2 语句块、赋值与运算符

语句块是由诸多语句组成，要来完成某一项工作的语句集合。语句块应该清晰、简洁。

要求：

1. 相对独立的语句块之间、变量说明之后必须加空行。如 列表 2.3 所示。

列表 2.3: 空行的插入范例

```
if (!p_name) {
    ...
}
__men_workers[i].p_arg    = 24;
__men_workers[i].p_weight = 60;
```

如果出现连续赋值语句，等号需要对齐，等号左边至少留一个空格，右边力求已最少的空格数保持对齐。

2. 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。如 列表 2.4 所示。

列表 2.4: case 语句缩进范例

```
switch (msg.id) {

case MSG_ID_1:  /* 分支 1 */
    ...
    break;      /* 跳出 */

case MSG_ID_2:  /* 分支 2 */
    ...
    break;      /* 跳出 */
```

```
default:        /* 默认处理 */
    ...
}
```

case 关键字和 switch 关键字对齐。分支语句放在相应 case 关键字的下一行。分支语句相对于 case 关键字的缩进是四个空格。case 后的条件与 case 之间相隔一个空格。: 紧接着条件常量。break 关键字的缩进与分支语句相同。第一个 case 语句与 switch 语句之间空一行。switch 必须要拥有 default 处理分支。

3. } 在纵向上，应该与相应的关键字对齐，如 列表 2.5 所示。

列表 2.5: } 对齐与关键字的对齐范例

```
if (1) {
    ...
}
```

4. 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。范例如下：

(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 (tempa == tempb) {
    a  = b + c;
    a += 3;
    b  = c & 0x0000FFFF;
}
```

5. 进行非对等操作时，操作符的前后不应该加空格。范例如下：

(1). ! 、~ 、++ 、-- 、& （地址运算符）等单目操作符前后不加空格:

```
*p_name = &p_buffer[0];
p_arg++;
```

(2). -> 、. 前后不加空格:

```
p_msg->id = 0;
msg.id    = 0;
```

6. 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。如 列表 2.6 所示。

列表 2.6: 使用空格突出关键字的操作范例

```
if (a >= b && c > d) {
    ...
}
while (counter <= 3) {
    ...
    counter++;
}
```

2.2.1 杂项

要求：

1. 非必要不要使用 goto 语句。goto 只能用于从内循环直接跳到循环外部、检测到错误直接跳到错误处理程序。影响可读性时禁止使用 goto 语句。
2. 用 if 语句来强调只执行两组语句中的一组。这种情况尽量不使用 switch case 语句。
3. 用 case 实现多路分支。
4. 不能使用 C 的默认操作优先级，多级操作中，每一级操作一定要使用括号进行优先级的划分。

2.3 结构化

要求：

1. 禁止出现两条等价的支路。
2. 尽量避免从循环引出多个出口。如 列表 2.7 所示。

列表 2.7: 避免从循环引出多个出口范例

```
for (i = 0; ; i++) {
    ...;
    if (i < 100) {
        break;
    }
}

/* 上面的语句段应该应该改为：*/
for (i = 0; i < 100; i++) {
    ...;
}
```

3. 尽量减少函数的出口。
4. 不要轻易用条件分支去替换逻辑表达式。如 列表 2.8 所示。

列表 2.8: 不要轻易用条件分支去替换逻辑表达式

```
if (a == true) {
    b = false;
} else {
    b = true;
}

/* 可以改为： */
b = !a;
```

5. 避免不必要的分支。如 列表 2.9 所示。

列表 2.9: 避免不必要的分支范例

```
if (a == 1) {
    index++;
} else {
    index += a;
}
/* 可以替换为 */
index += a;
```

3 函数、函数名及其参数

3.1 函数

函数是一个命名了的具有独立功能的程序段。所以，当我们要完成一个相对独立的功能，们就需要定义一个函数，但并不是在一个函数中实现所有功能。

3.1.1 函数的基本要求

要求：

1. 函数的规模必须限制在 200 行以内。过于复杂或冗长的函数不利于结构化程序设计，而且不利于调试与错误的排除。
2. 一个函数仅完成一件功能。不要将很多联系较少的功能写在同一个函数中。不要设计多用途面面俱到的函数。
3. 为简单功能编写函数。虽然为仅用一两行就可完成的功能去编函数似乎没有必要，但使用函数可使功能明确化，增加程序可读性，亦可方便维护、测试。
4. 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可以预测的输出。
5. 尽量不要编写依赖于其他函数内部实现的函数。
6. 避免设计多参数函数，不使用的参数从接口中去掉。
7. 非调度函数应减少或防止控制参数，尽量只使用数据参数。本建议目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令，来启动相应的功能实体即函数或过程），而本身并不完成具体功能。控制参数是指改变函数功能行为的参数，即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合，很可能使函数间的耦合度增大，并使函数的功能不唯一。如 列表 3.1 所示。

列表 3.1: 减少控制参数的范例

```
/* 如下函数构造不太合理： */
int int_add_and_sub (int opt, int num1, int num2)
{
    if (opt == INTEGER_ADD) {
        return (num1 + num2);
    } else {
        return (num1 - num2);
    }
}

/* 应该改为： */
int int_add (int num1, int num2)
{
    return (num1 + num2);
}
int int_sub (int num1, int num2)
{
    return (num1 - num2);
}
```

8. 尽量检查函数所有参数输入的有效性。（有条件）

9. 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。

10. 让函数在调用点显得易懂、容易理解。

11. 避免函数中不必要语句，防止程序中的垃圾代码。

12. 函数中不必要的语句可以使用如下两种注释，如 列表 3.2 所示。

列表 3.2: 函数中无效代码的注释范例

```
#if 0
  ...
#endif /* 0 */

/* 调试信息可以使用如下方式： */
#ifdef __DEBUG
  ..
#endif /* __DEBUG */
```

13. 防止把没有关联的语句放到一个函数中。防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易导致代码维护的问题，例如：在一种应用场合需要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。在编程时，经常遇到在不同函数中使用相同的代码，许多开发人员都愿把这些代码提出来，并构成一个新函数。若这些代码关联较大并且是完成一个功能的，那

么这种构造是合理的，否则这种构造将产生随机内聚的函数。如 列表 3.3 所示。

列表 3.3: 随机内聚函数改造范例

```
/* 下面的函数是一种随机内聚：*/
void var_init (void)
{
    g_rect_area.xsize = 100;
    g_rect_area.ysize = 100;

    g_men_child.weight  = 30;
    g_men_child.stature = 140;
}
/* 以上函数中，一个区域的属性与一个人的属性基本没有任何关系，故以上函数是随机内聚。
应如下分为两个函数：*/
void rect_init (void)
{
    g_rect_area.xsize = 100;
    g_rect_area.ysize = 100;
}
void men_init (void)
{
    g_men_child.weight  = 30;
    g_men_child.stature = 140;
}
```

14. 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。若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成一个新的函数。

15. 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。模块中函数划分的过多，一般会使函数间的接口变得复杂。所以过“小”的函数，例如扇入很低的或功能不明确的函数，不值得单独存在。

16. 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；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。函数较合理的扇出（调度函数除外）通常是 3-5。扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

17. 减少函数本身或函数间的递归调用。递归调用特别是函数间的递归调用（如 A->B->C->A），影响程序的可理解性；递归调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便，应减少没必要的递归调用。

18. 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。函数的划分与组织是模块的实现过程中很关键的步骤，如何划分出合理的函数结构，关系到模块的最终效率和可维护性、可测性等。根据模块的功能图或/及数据流图映射出函数结构是常用方法之一。

19. 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。优化函数结构时，要遵守以下原则：

    (a) 不能影响模块功能的实现。

    (b) 仔细考查模块或函数出错处理及模块的性能要求并进行完善。

    (c) 通过分解或合并函数来改进软件结构。

    (d) 考查函数的规模，过大的要进行分解。

    (e) 降低函数间接口的复杂度。

    (f) 不同层次的函数调用要有较合理的扇入、扇出。

    (g) 函数功能应可预测。

    (h) 提高函数内聚。（单一功能的函数内聚最高）

> **注意：** 对初步划分后的函数结构应进行改进、优化，使之更为合理。

20. 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。所有使用外设的代码都可能是不可重入的，需要使用各种方法避免重入。

21. 对于提供了返回值的函数，在引用时最好使用其返回值。如果不使用，尽量在调用时，加入：(void) 语句。如 列表 3.4 所示。

列表 3.4: 不使用函数返回值的范例

```
int net_send (...)
{
    return 0;
}
/* 如果调用上面的 net_send 函数，但是不使用其返回值，尽量使用如下方式： */
    (void)net_send(...);
```

22. 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或者使用局部变量。如 列表 3.5 所示。

列表 3.5: 使用宏替换较长的变量范例

```
#define __NETPORT_GET_IP(i)   __g_nd_buffer[i].netport.myip.s_addr

int __ip_recv (int index, uint8_t *p_buffer)
{
  uint32_t myip;
```

```
  ...
  myip = __NETPORT_GET_IP(index);   /* 获得指定网络接口的 IP 地址 */
  ...
}
```

23. 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）。如 列表 3.6 所示。

列表 3.6: 代码中添加测试信息

```
#ifdef  __DEBUG_LOG_EN
#define __DEBUG_LOG(p_str)    uart_send_str(p_str)
#else
#define __DEBUG_LOG(p_str)
#endif

int mem_copy (void *p_dest_addr, void *p_src_addr, int count)
{
    if (!p_dest_addr) {
      __DEBUG_LOG("p_dest_addr is NULL.\n");
      return -1;
    }
    if (!p_src_addr) {
      __DEBUG_LOG("p_src_addr is NULL.\n");
      return -1;
    }
    if (!count) {
      __DEBUG_LOG("count is 0.\n");
      return -1;
    }
    ...
}
```

3.2 函数名及其参数

3.2.1 函数命名规范

函数的命名采用“主 + 动”形式。此命名方式是很多大型软件采取的命名方式，例如：Linux、VxWorks、µC/OS-II 等等。这种命名清晰易懂，产生歧义的可能性小。

最佳的函数名长度应该在 8 到 16 个字符之间。

要求：

1. 函数名以主语开始，这个部分一般说明函数操作的对象，即函数是为处理什么对象而编写的。

例如：task 、semaphore 、motor 等等。

2. 函数名的第二部分是动词，一般描述对象的行为，即函数是为对象做什么服务的。例如：create 、pend 、on 、off 等等。

函数的全名示例如下:

```
task_create、semaphore_pend、motor_on、led_off
```

要求：

1. 函数名的主语部分为小写英文单词或英文单词缩写。
2. 动词部分全部为小写。
3. 主语部分、动词部分及主语动词的链接部分可使用下划线。为了不污染用户命名空间，非应用程序（模块内部）使用的函数必须以双下划线开始 __，例如：__task_init 、__semaphore_read 等等。如果出现名词众多（即对象继承层数过多的情况）可以使用多名词做函数起始主语，所有主语全部小写，主语之间加下划线。例如：EPC 这个模块包含有 WDT 这个子模块，WDT 这个模块的一个 init 操作的函数可以使用如下函数名：epc_wdt_init 。
4. 在 AWorks 平台中要求所有的函数和全局变量命名前加 aw_ 前缀。

3.2.2 函数头缩进格式

当函数的返回值、函数名、参数不超过 80 个字符时写在同一行，如 列表 3.7 所示。

列表 3.7: 80 个字符以内的函数定义范例

```
int motor_run (int speed, int direction)
{
    ...
}
```

要求：

1. 函数返回类型放在一行的起始点。
2. 函数名与函数返回类型同占一行。
3. 函数定义时，参数列表的左“(”与函数名最右边字符之间保留一个空格。
4. 函数参数排列不得超过 80 个字符，超过时使用纵向参数排列。
5. 逗号后面加空格，前面不允许加空格。
6. 参数类型后空两个空格在写入变量名。
7. 函数体的左右大括号放在最左边。
8. 返回类型与函数名之间留 1 到 2 个空格。

当一行 80 个字符写不下一个函数定义时使用纵向排列。如 列表 3.8 所示。

列表 3.8: 80 个字符以上的函数定义范例

```
int  motor_run (int speed,
                int direction,
                int time)
{
    ...
}

/* 当一行中第一个参数也写不下时使用下面的方式： */
static struct motor_control_struct * __motor_create (
    int speed, int direction)
{
    ...
}

/* 当一行中第一个参数也写不下，且第二行也写不下所有参数时，使用纵向排列： */
static struct motor_control_struct * __motor_create (
    int speed,
    int direction,
    int timeout)
{
    ...
}
```

要求：

1. 参数间隔符 ，放在参数右侧。
2. 参数类型与参数名之间空格数不确定。
3. 参数类型纵向对齐第一个参数的类型。
4. 参数名纵向对齐。
5. 参数列表的右括号紧贴最后一个参数。

3.2.3 AWorks 平台函数返回值规范

在 AWorks 平台中，如果函数的返回值表示的是错误类型，则返回值类型必须使用 `aw_err_t`，执行不成功的返回值必须使用 `aw_errno.h` 中定义的错误码的负值（加 - 号），如 `AW_EINVAL`，执行成功返回 `AW_OK`。返回值类型为布尔、指针或数值等的除外，此外若函数要兼容已有的标准的除外。AWorks 平台函数返回值规范示例如 列表 3.9 所示。

列表 3.9: AWorks 平台函数返回值规范示例

```
aw_err_t aw_event_raise (struct event_type *p_event,
                         void              *p_event_data,
                         int                proc_type)
{
    if (NULL == p_event) {
        return -AW_EINVAL; /* 参数无效 */
    }
    ...
    return AW_OK;
}
```

4 可读性结构与标识符

4.1 可读性

要求：

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。
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，必须用有意义的宏来代替。如 列表 4.1 所示。

列表 4.1: 用宏来定义有物理或实际意义的数字范例

```
#define TRUNK_IDLE      0   /**< 空闲 */
#define TRUNK_BUSY      1   /**< 忙 */
```

如 列表 4.1 所示，类似代表状态的数字，必须定义为宏。

3. 源程序中关系较为紧密的代码应尽可能相邻。如 列表 4.2 所示。

列表 4.2: 联系紧密的代码应该相邻范例

```
/* 以下是不合理的： */
int     length = 10;
char    get;
int     weight = 20;

/* 应该改为： */
int     length = 10;
int     weight = 20;

char    get;
```

4. 不要使用难懂的技巧性很高的语句，除非很有必要时。高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。++ ， -- 运算符在一条语句中只允许出现一次。

4.2 标识符

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。

说明：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缩写；一些单词有大家公认的缩写。

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  ;
```

要求：

1. 自己特有的命名风格，要自始至终保持一致，不可来回变化。
2. 除非必要，不要用数字或较奇怪的字符来定义标识符。
3. 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。
4. 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。

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    show / hide         send / receive
cut / paste        up / down           source / destination
```

5 注释、注释对齐

5.1 注释

注释分为行注释和块注释，块注释又分为函数体内块注释与函数体外块注释。行注释主要是说明该行代码执行的功能，函数体内块注释主要数说明下面的语句块执行的功能，函数体外块注释主要说明类型、结构、宏定义以及函数声明、全局变量定义等等。

从注释是否带特殊格式从而可以被文档工具抓取来分注释可分为普通注释和特殊块注释。所有注释要求能使用 Doxygen 抓取，特殊块注释使用 `/**` 开始，代码行后使用 `/**<` 开始。

要求：

1. 一般情况下，源程序有效注释量必须在 15%以上，一般要达到 20% 以上。注释的原则是有助于对程序的阅读理解，在该加的地方都加了，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。
2. 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。
3. 注释的内容要清楚、明了，含义准确，防止注释二义性。错误的注释不但无益反而有害。
4. 避免在注释中使用缩写，特别是非常规使用的缩写。在使用缩写时或之前，应对缩写进行必要的说明。
5. 全局变量要有较详细的注释，包括对其功能、及注意事项等的说明。
6. 对于 `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。这样比较清楚程序编写者的意图，有效防止无故遗漏 `break` 语句。
7. 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。
8. 在语句块的功能、意图层次上进行注释，提供有用、额外的信息。注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信

息。

9. 所有的注释均采用 C89 的 `/* ... */` 封闭式注释。
10. 除固定文件头说明、文件结束注释和函数体内注释外全部使用特殊块注释。
11. 所有特殊注释块建议尽量加 `\brief` 指示。

5.2 注释对齐

5.2.1 单行注释

单行注释（短注释）可单独占用一行，或写在定义或语句后共占一行，结束点不得超过 80 个字符处，若写不下则使用多行注释。

要求：

1. 独立的单行注释的起始点与该处代码的缩进一致。
2. 定义或语句后的单行注释在定义或语句后留至少一个空格后开始行注释。
3. 相似或功能相同的定义或语句的行注释开始处对齐。
4. 行注释的结束点不超过第 80 个字符处。
5. `/*` 和 `*/` 符号在一行以内。如 列表 5.1 所示。

列表 5.1: 单行注释范例

```
uint8_t lsb;            /* 低位字节 */
uint8_t msb;            /* 高位字节 */

lsb = my_data & 0xFFu; /* 获取 my_data 的低 8 位字节 */

/* 获取 my_data 的高 8 位字节 */
msb = (my_data >> 8u) & 0xFFu;
```

6. 注释的内容和 `/*` 及 `*/` 之间留各留一个空格。
7. 单独注释行的上方为空行。
8. 定义或语句后的注释应该只对当前代码行做出注释，否则应使用单独注释行。
9. 如果一行写不下请使用多行注释。

5.2.2 多行注释

要求：

1. 注释的起始点与该处代码的缩进相对应。
2. 注释本身至少占用三行，且注释上方为空行。

   多行注释使用范例，如 列表 5.2 所示。

列表 5.2: 多行注释范例

```
int char_get (void)
{
    /*
     * ... 注释内容 ...
     * ... 注释内容 ...
     */

    /* ... 语句块 ... */
}
```

3. /* 和 */ 各自单独占用一行。
4. 每行注释内容前以 * 开始，且和上一行的 * 对齐。
5. 注释内容与开始的 * 相隔一个空格。

5.3 文件内分割线注释

分割线注释仅用于对函数、定义进行视觉上的分割，便于阅读。

要求：

1. 起始点为第一行，结束点为第 80 行，中间全部用 * 。
2. 不能用于函数体内部。
3. 函数注释被放在头文件的非 static 函数在定义前放置一条分割线。

   分割线注释使用范例如 列表 5.3 所示。

列表 5.3: 分割线注释使用范例

```
/*******************************************************************************/
int foo (int a, int b, int c)
{
    ...
}

/*******************************************************************************/
int bar (int a, int b, int c)
{
    ...
}
```

5.4 文件内代码分组注释

为了阅读代码方便，建议按以下顺序对文件进行简单分组：头文件包含、宏定义、外部变量声明、外部函数声明、本地全局变量声明、本地函数声明、本地全局变量定义、本地函数定义和外部函数定义。

要求：

1. 注释起始点为行起始地点。
2. 注释的第一行没有结束符，除了第一个字符 `/` 其他的字符均为 `*`
3. 第一行最后一个 `*` 出现在第 80 个字符的位置。
4. 第二行空 2 个空格后为简短的分组注释内容。
5. 最后一行的起始点为行起始地点。
6. 最后一行前 79 个字符为 `*` 。
7. 最后一行第 80 个字符为 `/` 。如 列表 5.4 所示。
8. 函数体外的块注释内容从第二行开始，与左边界的距离为两个空格。

列表 5.4: 分组注释

```
/*******************************************************************************
  includes
*******************************************************************************/
```

5.5 全局常量宏和全局变量注释

全局常量宏和全局变量应该全部要进行注释，尤其全局变量。

要求：

1. 使用 `/**< \brief` 开始注释，其中 `\brief` 可选，但建议使用。
2. 在宏或变量定义的后面编写注释。
3. 相似或功能相同的定义后的注释开始处对齐。
4. 若一行写不下，则在定义的上面一行使用单独行注释。
5. 单独行注释使用 `/** \brief` 开始。

常量宏和全局变量定义注释范例如 列表 5.5 所示。

列表 5.5: 常量宏和全局变量定义注释范例

```
#define AW_I2C_M_7BIT           0x0000u     /**< \brief 7 位设备地址（默认）*/
#define AW_I2C_M_10BIT          0x0001u     /**< \brief 10 位设备地址 */

/** \brief 忽略所有无应答错误（包括从机地址无应答和数据无应答）*/
#define AW_I2C_M_IGNORE_NAK     0x0020u

int  g_i2c_ctlr_num;    /**< \brief I2C 控制器个数 */
```

5.6 枚举和结构的注释

枚举和结构的注释相同，现以结构体的定义为例说明。

要求：

1. 结构体定义必须有结构体本身的注释和结构体成员的注释。如 列表 5.6 所示为结构体定义范例。

列表 5.6: 结构体注释范例

```
1  /**
2   * \brief 环形缓冲区管理结构
3   *
4   * 这是一个免锁的环形缓冲区，不需要记录缓冲中数据的个数。
5   * 这个结构可以静态或动态创建，如果使用动态创建，则调用者必须负责内存管理。
6   *
7   * \note 不要直接操作本结构的成员
8   */
9  struct aw_ring_buffer {
10     int      in;     /**< \brief 缓冲区数据写入位置 */
11     int      out;    /**< \brief 缓冲区数据读出位置 */
12     int      size;   /**< \brief 缓冲区大小 */
13
14     /**
15      * \brief 缓冲区指针
16      *
17      * 这个指针指向的内存空间可以静态或动态创建，如果使用动态创建，
18      * 则调用者必须负责内存管理。
19      */
20     char    *p_buf;
21 };
```

2. 结构体的注释使用特殊注释块 `/**` 开始，至少使用一个 `\brief` 进行简短描述，其它的注释可视结构体复杂程序酌情注释。

3. 结构体成员后的注释（10~12 行）使用特殊注释块 `/**<``开始，``\brief` 是可选的，但建议使用，这样生成的文档阅读更方便。

4. 若结构体成员后的空间不够书写（如 14~20 行），可写在上方。注意此时使用注释块 `/**` ，而不是 `/**<` 。

5.7 全局带参数宏和函数体的注释

全局带参数宏和每个函数的开头都必须有注释，它们使用相同的注释规则，以函数注释为例，如 列表 5.7 所示。优先使用中文。

列表 5.7: 函数头的注释

```
/**
 * \brief 简短的作用描述
 *
 * \details 详细描述（可选），如算法的介绍等
 *
 * \param[in]       输入参数  参数描述
 * \param[out]      输出参数  参数描述
 * \param[in,out]   双向参数  参数描述
 *
 * \retval 返回值 1  返回值 1 描述
 * \retval 返回值 2  返回值 2 描述
 *
 * \note 特殊说明，如注意事项，函数使用条件等
 */
```

其中 `\brief` 简短描述是强制的，`\details` 是可选的，建议如果函数比较复杂，或使用了某种算法，建议在 `\details` 中进行说明。其余的若有则写，没有则不写。

`\param` 后面的方向指示如 `[in]` 等在不会造成误会的情况下可以省略。`\retval` 可根据实际情况使用 return 替代（如返回的是不确定的数据个数而不是错误码时）。

函数开头注释示例如 列表 5.8 所示。示例函数的参数 `p_msgs` 用 `[in,out]` 表示双向参数，返回值是消息个数，无法罗列，使用 `\return` 进行说明，函数内部不对参数有效性进行检查，故使用 `\note` 特意说明。

列表 5.8: 函数头注释示例

```
/**
 * \brief  开始一次 I2C 传输
 *
 * \param[in]       p_i2c   I2C 控制块指针
 * \param[in,out]   p_msgs  I2C 传输消息数组
 * \param[in]       num     I2C 传输消息个数
 *
 * \return I2C 传输完成的消息个数
 *
 * \note p_i2c 指向的控制块和 p_msgs 指向的消息数组必须在调用本函数前初始化
 */
```

```
int i2c_transfer (struct i2c_type *p_i2c, struct i2c_msg p_msgs[], size_t num);
```

如 列表 5.9 所示为另一个注释示例，函数的参数的输入输出特性十分明显，故省略 [in] 、[out] 等，且返回值只有 3 个，使用 \retval 进行罗列。

列表 5.9: 函数开头注释简化版示例

```
/**
 * \brief 读 I2C 从机寄存器
 *
 * \param  p_dev I2C 从机设备描述符
 * \param  reg   寄存器起始地址，从此地址开始读取数据，
 * \param  p_buf 数据缓冲区，读取的数据存放于此
 * \param  len   要读取的数据个数
 *
 * \retval AW_OK      读取成功
 * \retval -AW_EINVAL 参数错误，设备描述错误或 buf 为空指针时返回这个错误
 * \retval -AW_EIO    物理错误，物理设备不存在，或设备损坏不应答等
 */
aw_err_t aw_i2c_read (aw_i2c_device_t  *p_dev,
                      uint32_t          reg,
                      uint8_t          *p_buf,
                      size_t            len);
```

6 宏定义

6.1 编译预处理

6.1.1 普通宏定义

要求：

1. 宏定义的符号所有字符必须使用大写字符。
2. 内部使用的宏定义符号前必须加双下划线 `__` 。如 列表 6.1 所示。

列表 6.1: 宏定义范例

```
#define __INTERNAL_MACRO        xxxx    /**< \brief 内部使用宏定义 */
#define EXTERNEL_MACRO          xxxx    /**< \brief 外部使用宏定义 */
```

3. `#define` 、`#if` 等等编译预处理命令必须写在一行的起始位置。
4. 每一个宏定义必须加入相关的注释，可以是行注释，也可以是函数体外的块注释。
5. `#define` 和被定义的符号之间留一个字符的宽度
6. `#define` 定义的宏名和宏体之间保留的空格数，以左对齐为标准。
7. AWorks 平台还要求宏定义的符号必须使用 `AW_` 开头。

6.1.2 带参数的宏定义

要求：

1. 简单的操作，尽量使用宏来完成。
2. 调用带参数宏定义时，不可使用单目运算符，例如：`++` 、`--` 等等。

带参数的宏定义范例，如 列表 6.2 所示。

列表 6.2: 带参数的宏定义范例

```
/** \brief 计算两个整形数中的最大值 */
#define __MAX(num1, num2)       (((num1) > (num2)) ? (num1) : (num2))
```

其他方面缩进与对其与语句块基本相同。

头文件（.h）中，必须使用防止嵌套引用的语句，该语句的定义如 列表 6.3 所示。

列表 6.3: 防止嵌套引用的编译预处理范例

```
#ifndef __XXX_H
#define __XXX_H

    ... /*  头文件内容  */

#endif /* __XXX_H */
```

要求：

1. 以上的范例中，XXX 为该头文件的文件名。
2. `#ifndef` 和 `#define` 之间不允许加入空行。
3. `#define __XXX_H` 后面加一行空行。
4. `#endif` 上面留一行空行。

如果需要声明 C++ 兼容的函数，如 列表 6.4 所示。

列表 6.4: 声明 C++ 兼容函数范例

```
#ifdef __cplusplus
extern "C" {
#endif

#ifdef __cplusplus
}
#endif
```

7 文件相关

7.1 文件说明

文件说明是指对 C 语言源代码文件的说明。它主要用于说明源代码文件的主要内容、主要作用、创建信息及更新信息。格式如 列表 7.1 所示。

列表 7.1: 源代码文件说明格式

```
/*******************************************************************************
*                                 ZY Modbus
*                            ----------------------
*                            The embed Modbus stack
*
* Copyright (c) 2001-present Guangzhou ZHIYUAN Electronics Co., Ltd..
* All rights reserved.
*
* Contact information:
* web site:    http://www.zlg.cn
* e-mail:      support@zlg.cn
*******************************************************************************/

/**
 * \file
 * \brief Data type defines for ZY-Modbus
 *
 * detailed descriptions for this file (optional).
 *
 * \internal
 * \par Modification history
 * - 1.01 12-08-28  yangtao, fix xxxx() function
 * - 1.00 12-06-20  liuweiyun, first implementation
 * \endinternal
 */
```

如 列表 7.1 所示的文件说明各项内容说明如下。1~12 行为软件模块的固定信息，包含如下内容：

1. 软件模块信息（2~4 行），软件模块名以及软件模块的简单描述等。
2. 版权声明（6、7 行），填写公司名称等。
3. 技术支持（9~11 行），公司部门网址，技术支持邮箱等。

14~25 行为文件相关信息：

1. 特殊注释块（14 行），使用 /** 作为注释的开始，方便生成 Doxygen 文档。
2. 文件名（15 行），填写 \file 即可，由 Doxygen 自动引用。
3. 文件简短描述（16 行），在 \brief 后空一格填写本文件的简单描述。
4. 文件详细描述（18 行），在简短描述后空一行填写详细的描述（可选）。
5. 修改记录（20~24）。

文件修改记录格式如下：

1. 标记为内部文档（20 行），使用 \internal 开始。
2. 修改记录开始（21 行），填写 \par Modification history 。
3. 修改记录（22~23 行），每条记录的格式为：“- 修订版本修改日期修改人，改动说明”。每条记录以 - 开始，使生成的修改记录文档具有列表样式。修改说明应尽可能短，但不能遗漏，一行写不下可以在下一行对齐修改人处继续写。新的修改记录放在旧的记录上面。
4. 修改记录结束（24 行），填写 \endinternal 。

具体实例，见代码规范的示例程序附件。

文件的尾部格式，如 列表 7.2 所示。

列表 7.2: 文件的尾部格式

```
/* end of file */
```

7.2 模块文件的组织

我们编写一个模块时，通常需要若干个文件（.c 和 .h 等等）分别用于模块的实现，模块的说明，模块的配置，模块的测试和示例等等。

模块目录的组织要求：

1. 至少具备存放头文件的 include 和存放源码文件的 source 目录。
2. 根据模块实际情况可选的目录有：config 存放配置文件，docs 存放文档，example 存放示例代码，test 存放测试用例。

模块文件要求：

1. 文件命名全部小写，分隔符使用下划线 _ 。
2. 必须存在 xxx.h 文件，将模块内的类型与接口函数引出。
3. 引出部分不能包含模块内使用的变量、函数、类型、控制参数等等。
4. 如果模块需要配置时，必须存在至少一个专用头文件用来对模块的功能进行配置。
5. 配置头文件内部的配置宏，最好使用保护机制。如 列表 7.3 所示。

列表 7.3: 配置宏的保护机制范例

```
/*******************************************************************************
  必须需要定义的宏
*******************************************************************************/
#ifndef TASK_NUMBER
#error  “TASK_NUMBER undeclared”
#endif

/*******************************************************************************
  可以有默认选项的宏
*******************************************************************************/
#ifndef TASK_NUMBER
#define TASK_NUMBER             10
#endif
```

6. AWorks 平台要求文件名必须以 aw_ 开头，以减少对文件命名空间的污染。

7.3 其他

在嵌入式系统中，对硬件某个寄存器进行操作，可能需要对寄存器的某位置位, 要用移位的方法，这样一目了然，维护也容易维护。如 列表 7.4 所示。

列表 7.4: 用移位方法表示对寄存器某个位进行置位

```
IOOSET |= (1u << 7);
IOOSET |= (1u << 7) + (1u << 3);

#define BIT_CLR(reg, bit)  (reg &= ~(1u << bit))
#define BIT_SET(reg, bit)  (reg |=  (1u << bit))
```

8 更改记录

未发布

- [支持] #5058 : 更新文件头的版权信息

`1.5.0-a1 <2017-05-27>`

- [功能] #4970 : 添加“更改记录”章节
- [支持] #4971 : 由 Word 转换为 reStructuredText 格式

`1.4.0 <2014-10-23>`

•函数指针变量命名前缀由 `pfunc` 改为 `pfn`

•增加源码目录组织结构规范

•增加 Apollo 平台相关的代码规范，全局宏定义 `AW_` 前缀，文件名和全局函数名添加 `aw_` 前缀

•增加 Apollo 平台函数返回值规范说明