C 语言编程规范

USTB Roboteam

Create Date:2007-01
Version: 2.10bate

编码格式

I. 注释

注释是源码程序中非常重要的一部分，一般情况下，源程序有效注释量必须在15％以上。

注释的原则是有助于对程序的阅读理解，所以注释语言必须准确、易懂、简洁。注释不宜太多也不能太少，注释的内容要清楚、明了，含义准确，防止注释二义性，该加的地方一定要加，但不必要的地方一定不要加。

（1）模块（C文件）描述

模块描述中应该包括。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。 例如：

```
/*************************************************************
 Module Name: //模块的名字
 Module Date: //生成日期
 Module Auth: //作者名字
Description:    // 用于详细说明此程序文件完成的主要功能，与其他模块
                // 或函数的接口，输出值、取值范围、含义及参数间的控
                // 制、顺序、独立或依赖等关系
Others:       // 其它内容的说明
    Revision History:
    Date        Rel Ver.    Notes
    月/日/年    x.x      //所进行的修改
*************************************************************/
/*---------------Includes--------------*/
//包含的头文件
/*-----------Local Variables----------- */
//定义一些本地变量
/*------Local Structures and Typedefs ---*/
//要使用的一些数据结构
/*----------Extern Variables ----------*/
```

```
//使用到的一些外部变量

/*------------Definitions--------------*/

//一些#defines及具体的函数实现
```

（2）头文件描述

头文件一般包括了数据结构的定义，函数原形的说明，宏定义等，不许包含函数体和变量实体，文件名使用缺省的后缀 .h ，头文件的注释可如下：

```
#ifndef MODEL_H
#define MODEL_H


/*****************************************************************
 Module Name: model.h
 Module Date: month/day/year
 Module Auth: your name

 Description: a short introduction of this module.

    Revision History:
    Date        Rel Ver.     Notes
    month/day/year      x.x     [e.g.] Module created

*****************************************************************/

/*---------------Includes--------------*/
//the head files that were included
//[e.g.] #include "head.h"

/*--------Structures and Typedefs------*/
/*[e.g.] struct model{
         char MemberOne;
         int  MemberTwo;
         char MemberThree[3];
         struct module *MemberFour;
        }
   enum BOOL {TRUE ,FALSE };
   typedef struct module MODULE;
*/

/*--------------Defines----------------*/
//[e.g.] #define MODLE  2

/*---------extern variables------------*/
//the variables that were defined in other modules
//[e.g.] estern char ExternVariable;
```

```
/*-----External Function Prototypes------*/
/*  External Function Prototypes */
//the functions that were implemented in other modules
//[e.g.] extern unsigned char model(int Input[2]);


#endif
```

（3）函数描述

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（函数、表）等。 例如：

```
/*************************************************************
 Function Name: //函数名
 Function Description: //函数功能、性能等的描述
Inputs:  // 输入参数说明，包括每个参数的作用、取值说明及参数间关系。
 Outputs: / /对输出参数的说明
 Notes: // 本函数调用的函数清单及其他
*************************************************************/
```

（4）其他注释

1. 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。说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

2. 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。

3. 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。变量、常量、宏的注释应放在其上方相邻位置或右方。 例如：

```
/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000

#define MAX_ACT_TASK_NUMBER 1000  / / active statistic task number
```

4. 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。

5. 将注释与其上面的代码用空行隔开。 例如：

```
/* code one comments */
program code one

/* code two comments */
program code two
```

6. 在代码的功能、意图层次上进行注释，提供有用、额外的信息。 说明：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。 例如：如下注释意义不大。

```
/* if receive_flag is TRUE */
if (receive_flag)
```

而如下的注释则给出了额外有用的信息。

```
/* if mtp receive a message from links */
if (receive_flag)
```

II. 排版

（1）程序块要采用缩进风格编写，缩进的空格数为 4 个。把源程序中的Tab字符转换成4个空格；一个缩进等级（Indentation Level）是4个空格；变量定义和可执行语句要缩进一个等级；函数的参数过长时，也要缩进。

（2）程序块的分界符（如C/C++语言的大括号'{'和'}'）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。 例如：

```
for (...)
{
    ... // program code
}
if (...)
{
    ... // program code
}
void example_fun( void )
{
    ... // program code
}
```

（3）相对独立的程序块之间、变量说明之后必须加空行。

```
if (!valid_ni(ni))
{
    ... // program code
}

repssn_ind = ssn_data[index].repssn_index;
repssn_ni  = ssn_data[index].ni;
```

（4）不允许把多个短语句写在一行中，即一行只写一条语句。

```
rect.length = 0;
rect.width  = 0;
```

（5）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 (pUserCR == NULL)
{
    return;
}
```

（6）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。

（7）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。 说明：采用这种松散方式编写代码的目的是使代码更加清晰。 由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在C语言中括号已经是最清晰的标志了。 在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

- 逗号、分号只在后面加空格。

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

- 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 (current_time >= MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

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

```
*p = 'a';        // 内容操作"*"与内容之间
flag = !isEmpty; // 非操作"!"与内容之间
p = &mem;        // 地址操作"&" 与内容之间
i++;             // "++","--"与内容之间
```

- "->"、"."前后不加空格。

```
p->id = pid;     // "->"指针前后不加空格
```

- 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 && c > d)
```

可读性

- I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。

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。
示例：下列语句中的表达式

```
word = (high << 8) | low     (1)
if ((a | b) && (a & c))      (2)
if ((a | b) < (c & d))       (3)
```

如果书写为

```
high << 8 | low
a | b && a & c
a | b < c & d
```

由于

```
high << 8 | low = ( high << 8) | low,
a | b && a & c = (a | b) && (a & c),
```

(1)(2)不会出错，但语句不易理解；
a | b < c & d = a | （b < c） & d，(3)造成了判断条件出错。

- 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。

示例：如下的程序可读性差。

```
if (Trunk[index].trunk_state == 0)
{
    Trunk[index].trunk_state = 1;
    ...  // program code
}
```

应改为如下形式。

```
#define TRUNK_IDLE 0
#define TRUNK_BUSY 1

if (Trunk[index].trunk_state == TRUNK_IDLE)
{
    Trunk[index].trunk_state = TRUNK_BUSY;
    ...  // program code
}
```

- III 源程序中关系较为紧密的代码应尽可能相邻。

说明：便于程序阅读和查找。 示例：以下代码布局不太合理。

```
rect.length = 10;
char_poi = 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; // 矩形的长与宽关系较密切，放在一起。
char_poi = str;
```

- IV 不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：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。
示例：如下表达式，考虑不周就可能出问题，也较难理解。

```
* stat_poi ++ += 1;

* ++ stat_poi += 1;
```

应分别改为如下。

```
*stat_poi += 1;
stat_poi++;     // 此二语句功能相当于“ * stat_poi ++ += 1; ”

++ stat_poi;
*stat_poi += 1; // 此二语句功能相当于“ * ++ stat_poi += 1; ”
```

命名规则

I. 函数

- 函数的规模尽量限制在200行以内

- 函数名应该能体现该函数完成的功能，建议采用动词+名次的形式。关键部分应该采用完整的单词，辅助部分若太常可采用缩写，缩写应符合英文的规范。每个单词的第一个字母大写。

```
ShowPoints CtrlDestBoard SendResetMsg
```

- （3）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。

```
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  ;
```

I. 变量

- 在命名常量时，用大写：MAX_VALUE

- 结构：结构的定义有两个名称，一个是该结构的类型名，一个是变量名。按照C语言的语法，这两个名称都是可选的，但二者必有其一。我们要求写类型名，类型名以tag作前缀。

```
Struct tagModel {
  ……//结构成员
}Model;
tagModel  Direction;
```

对于程序中的常用的结构，希望能使用typedef定义，

```
tTypedef Struct tagModel {
  ……//结构成员
}Model, pModel;
```

- 联合:(同上)

- 变量的命名建2议采用名词和匈牙利命名规则，变量的第一个字母小写，表示其数据类型，

```
int iIndex,iCount;
char cIndex,cSerialOut;
```

指针类型 pSever, pMsg;

- 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i、j、k作局部循环变量是允许的。

  说明：变量，尤其是局部变量，如果用单个字符表示，很容易敲错（如i写成j），而编译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。
  示例：下面所示的局部变量名的定义方法可以借鉴。

  ```
  int liv_Width
  其变量名解释如下：
    l      局部变量（Local）    （其它：g    全局变量（Global）...）
    i      数据类型（Interger）
    v      变量（Variable）     （其它：c    常量（Const）...）
    Width  变量含义
  这样可以防止局部变量与全局变量重名。
  ```

- 采用UNIX的大小写混排的方式，不要使用大小写与下划线混排的方式，用作特殊标识如标识成员变量或全局变量的m_和g_，其后加上大小写混排的方式是允许的。

```
AddUser  m_AddUser
```

- 除非必要，不要用数字或较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 set_sls00( BYTE sls );
```

  应改为有意义的单词命名

```
#define _EXAMPLE_UNIT_TEST_
#define _EXAMPLE_ASSERT_TEST_
void set_udt_msg_sls( BYTE sls );
```

预编译宏

- **用宏定义表达式时，要使用完备的括号。**

示例：如下定义的宏都存在一定的风险。

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

正确的定义应为：

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

- **将宏所定义的多条表达式放在大括号中。**

示例：下面的语句只有宏的第一条表达式被执行。为了说明问题，for语句的书写稍不符规范。

```
#define INTI_RECT_VALUE( a, b )\
    a = 0;\
    b = 0;

for (index = 0; index < RECT_TOTAL_NUM; index++)
    INTI_RECT_VALUE( rect.a, rect.b );
```

正确的用法应为：

```
#define INTI_RECT_VALUE( a, b )\
{\
    a = 0;\
    b = 0;\
}

for (index = 0; index < RECT_TOTAL_NUM; index++)
{
   INTI_RECT_VALUE( rect[index].a, rect[index].b );
}
```

- **使用宏时，不允许参数发生变化。**

示例：如下用法可能导致错误。

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

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

正确的用法是：

```
b = SQUARE( a );
a++; // 结果：a = 6，即只执行了一次增1。
```

程序效率

- 编程时要经常注意代码的效率。

  说明：代码效率分为全局效率、局部效率、时间效率及空间效率。全局效率是站在整个系统的角度上的系统效率；局部效率是站在模块或函数角度上的效率；时间效率是程序处理输入任务所需的时间长短；空间效率是程序所需内存空间，如机器代码空间大小、数据空间大小、栈空间大小等。

- 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。

  说明：不能一味地追求代码效率，而对软件的正确性、稳定性、可读性及可测性造成影响。

- III 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。

- IV 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。

  说明：这种方式是解决软件空间效率的根本办法。
  示例：如下记录学生学习成绩的结构不合理。

```
typedef unsigned char  BYTE;
typedef unsigned short WORD;

typedef struct STUDENT_SCORE_STRU
{
    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
    BYTE subject;
    float score;
} STUDENT_SCORE;
```

因为每位学生都有多科学习成绩，故如上结构将占用较大空间。应如下改进（分为两个结构），总的存贮空间将变小，操作也变得更方便。

```
typedef struct STUDENT_STRU
{
    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
} STUDENT;

typedef struct STUDENT_SCORE_STRU
{
    WORD student_index;
```

```
    BYTE subject;
    float score;
} STUDENT_SCORE;
```

- 循环体内工作量最小化。

说明：应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。
示例：如下代码效率不高。

```
for (ind = 0; ind < MAX_ADD_NUMBER; ind++)
{
    sum += ind;
    back_sum = sum; /* backup sum */
}
```

语句“back_sum = sum;”完全可以放在for语句之后，如下。

```
for (ind = 0; ind < MAX_ADD_NUMBER; ind++)
{
    sum += ind;
}
back_sum  = sum; /* backup sum */
```

- 仔细分析有关算法，并进行优化。

- 仔细考查、分析系统及模块处理输入（如事务、消息等）的方式，并加以改进。

- 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。

说明：软件系统的效率主要与算法、处理任务方式、系统功能及函数结构有很大关系，仅在代码上下功夫一般不能解决根本问题。

- 编程时，要随时留心代码效率；优化代码时，要考虑周全。

- 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。

说明：对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

- XI 要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数。

说明：只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方式。嵌入汇编可提高时间及空间效率，但也存在一定风险。

- XII在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量，来提高空间效率。

说明：这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。

- 在多重循环中，应将最忙的循环放在最内层。

说明：减少CPU切入循环层的次数。
示例：如下代码效率不高。

```
for (row = 0; row < 100; row++)
{
    for (col = 0; col < 5; col++)
    {
        sum += a[row][col];
    }
}
```

可以改为如下方式，以提高效率。

```
for (col = 0; col < 5; col++)
{
    for (row = 0; row < 100; row++)
    {
        sum += a[row][col];
    }
}
```

- 尽量减少循环嵌套层次。

- 避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中。

说明：目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不可以。
示例：如下代码效率稍低。

```
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;
    }
}
```

因为判断语句与循环变量无关，故可如下改进，以减少判断次数。

```
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;
    }
}
```

- 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。

说明：浮点运算除法要占用较多CPU资源。
示例：如下表达式运算可能要占较多CPU资源。

```
#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 ) // 编译器编译时，将生成具体浮点数
radius = circle_length * PAI_RECIPROCAL / 2;
```

- 不要一味追求紧凑的代码。