**嵌入式C语言编程规范（个人规约）**

2015年06月18日 09:51:38 [bahutou](https://me.csdn.net/zhanglianpin) 阅读数：7560

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/zhanglianpin/article/details/46544431

**0规范制定说明**

**0.1箴言**

技术人员设计程序的首要目的是用于技术人员沟通和交流，其次才是用于机器执行。程序的生命力在于用户使用，程序的成长在于后期的维护及根据用户需求更新和升级功能。如果你的程序只能由你来维护，当你离开这个程序时，你的程序也和你一起离开了，这将给公司和后来接手的技术人员带来巨大的痛苦和损失。因此，为了程序可读、易理解、好维护，你的程序需要遵守一定的规范，你的程序需要设计。

**“程序必须为阅读它的人而编写，只是顺便用于机器执行。”**

              ——Harold Abelson 和 Gerald Jay Sussman

**“编写程序应该以人为本，计算机第二。”**

——Steve McConnell

**0.2简介：**

为提高产品代码质量，指导仪表嵌入式软件开发人员编写出简洁、可维护、可靠、可测试、高效、可移植的代码，编写了本规范。

本规范将分为完整版和精简版，完整版将包括更多的样例、规范的解释以及参考材料(what & why)，而精简版将只包含规则部分(what)以便查阅。

在本规范的最后，列出了一些业界比较优秀的编程规范，作为延伸阅读参考材料。

本规范主要包含以下两个方面的内容：

一：为形成统一编程规范，从编码形式角度出发，本规范对标示符命名、格式与排版、注释等方面进行了详细阐述。

二：为编写出高质量嵌入式软件，从嵌入式软件安全及可靠性出发，本规范对由于C语言标准、C语言本身、C编译器及个人理解导致的潜在危险进行说明及规避。

**0.3适用范围：**

本规范适用于济南金钟电子衡器股份有限公司仪表台秤产品部嵌入式软件的开发，也对其他嵌入式软件开发起一定的指导作用。

**0.4术语定义**

**0.4.1 规范术语**

**原则：**编程时必须坚持的指导思想。

**规则：**编程时需要遵循的约定，分为强制和建议（强制是必须遵守的，建议是一般情况下需要遵守，但没有强制性）。

**说明：**对原则/规则进行必要的解释。

**实例：**对此原则/规则从正、反两个方面给出例子。

**材料：**扩展、延伸的阅读材料。

**Unspecified：**未详细说明的行为，这些是必须成功编译的语言结构，但关于结构的行为，编译器的编写者有某些自由。例如C语言中的“运算次序”问题。这样的问题有 22 个。 在某种方式上完全相信编译器的行为是不明智的。编译器的行为甚至不会在所有可能的结构中都是一致的。

**Undefined：**未定义行为，这些是本质的编程错误，但编译器的编写者不一定为此给出错误信息。相应的例子是无效参数传递给函数，或函数的参数与定义时的参数不匹配。从安全性角度这是特别重要的问题，因为它们代表了那些不一定能被编译器捕捉到的错误。

**Implementation-defined：**实现定义的行为，这有些类似于“unspecified ”问题，其主要区别在于编译器要提供一致的行为并记录成文档。换句话说，不同的编译器之间功能可能会有不同，使得代码不具有可移植性，但在任一编译器内，行为应当是良好定义的。比如用在一个正整数和一个负整数上的整除运算“/ ”和求模运算符“% ”。存在76个这样的问题。从安全性角度，假如编译器完全地记录了它的方法并坚持它的实现，那么它可能不是那样至关重要。尽可能的情况下要避免这些问题。

**0.4.2  C语言相关术语**

**声明(declaration)：**指定了一个变量的标识符，用来描述变量的类型，是类型还是对象，

者函数等。声明，用于编译器(compiler)识别变量名所引用的实体。以下这些就是声明：

extern int bar;

extern int g(int,int);

double f(int,double);[对于函数声明，extern关键字是可以省略的。]

**定义(definition)**：是对声明的实现或者实例化。连接器(linker)需要它（定义）来引用内存实体。与上面的声明相应的定义如下：

int bar;

int g(int lhs,int rhs) {returnlhs\*rhs;}

double f(int i,double d) {returni+d;}

**0.5规则的形式**

**规则/原则<序号>(规则类型)：规则内容。**

**[原始参考]**

<序号>：每条规则都有一个序号，序号是按照章节目录-\*\*的形式，从数字1开始。例如，若在此章节有个规则的话，序号为0.5-1。

(规则类型)：或者是‘强制’，或者是‘建议’。

规则内容：此条规则的具体内容。

[原始参考]：指示了产生本条款或本组条款的可应用的主要来源。

**1标示符命名规则**

**1.1标示符命名总则**

**规则1.1-1(强制)：标识符（内部的和外部的）的有效字符不能多于31。**

**[UndefinedImplementation-defined]**

**说明：**ISO 标准要求在内部标识符之间前31 个字符必须是不同的，外部标识符之间前6 个字符必须是不同的（忽略大小写）以保证可移植性。我们这里放宽了此要求，要求内部、外部标示符的有效字符不能多于31即可。这样主要是便于编译器识别，代码清晰易读，并保证可移植性。

**规则1.1-2(强制)：具有内部作用域的标识符不应使用与具有外部作用域的标识符相同的**

**名称，在内部作用域里具有内部标示符会隐藏外部标识符。**

**说明：**外部作用域和内部作用域的定义如下。文件范围内的标识符可以看做是具有最外部

（outermost ）的作用域；块范围内的标识符看做是具有更内部（more inner）的作用域，连续嵌套的块，其作用域更深入。如果内部作用域标示符和外部作用域标示符同名，内部作用域标示符会覆盖外部作用域标示符，导致程序混乱。

**实例：**

INT8U  test;

{

INT8U  test;      /\*定义了两个test  \*/

   test  =  3;      /\*这将产生混淆    \*/

}

**规则1.1-3(建议)：具有静态存储期的对象或函数标识符不能重用。**

**说明：**不管作用域如何，具有静态存储期的标识符都不应在系统内的所有源文件中重用。它包含带有外部链接的对象或函数，及带有静态存储类标识符的任何对象或函数。在一个文件中存在一个具有内部链接的标识符，而在另外一个文件中存在着具有外部链接的相同名字的标识符，或者存在两个标示符相同的外部标示符。对用户来说，这有可能导致混淆。

**实例：**

          test1.c

/\*\*定义了一个静态文件域变量test1\*/

          static INT8U  test1;

          void test\_fun(void)

          {

                   INT8U test1;   /\*定义了一个同名的局部变量test1\*/

}

test2.c

/\*\*在另一个文件又定义了一个具有外部链接的文件域变量test1\*/

INT8U  test1;

**原则1.1-4(强制)：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：**标示符的命名尽量做到见名知意，尽量让别人快速理解你的代码。

**实例：**

好的命名方法：

          INT8U   debug\_message;

          INT16U  err\_num;

      不好的命名方法：

          INT8U   dbmesg;

          INT16U   en;

**原则1.1-5(强制)：常见通用的单词缩写尽量统一，不得使用汉语拼音、英语混用。**

**说明：**简短的单词可以使用略去‘元音’字母形成缩写，较长的单词可以使用音节首字母

者单词前几个字母形成缩写，针对大家公认的单词缩写要统一。对于特定的项目要使

用的专有缩写应该注明或者做统一说明。

**实例：**常见单词缩写表(建议)：

|  |  |  |  |
| --- | --- | --- | --- |
| **单词** | **缩写** | **单词** | **缩写** |
| argument | arg | buffer | buf |
| clock | clk | command | cmd |
| compare | cmp | configuration | cfg |
| device | dev | error | err |
| hexadecimal | hex | increment | inc |
| initialize | init | maximum | max |
| message | msg | minimum | min |
| parameter | param | previous | prev |
| register | reg | semaphore | sem |
| statistic | stat | synchronize | syn |
| temp | tmp |  |  |

**原则1.1-6(建议)：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。**

**实例：**常见反义词表：

|  |  |  |  |
| --- | --- | --- | --- |
| **正义** | **反义** | **正义** | **反义** |
| 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 | copy | pase |
| up | down |  |  |

**原则1.1-7(建议)：标示符尽量避免使用数字编号，除非逻辑上需要。**

**实例：**

#define   DEBUG\_0\_MSG

         #define   DEBUG\_1\_MSG

应改为更有意义的定义：

#define   DEBUG\_WARN\_MSG

         #define   DEBUG\_ERR\_MSG

**材料：**《代码大全第2版》（Steve McConnell 著 金戈/汤凌/陈硕/张菲 译 电子工业出版社

2006年3月）"第11章变量命的力量"。

**1.2文件命名及存储规则**

**规则1.2-1(强制)：文件名使用小写字母。**

**说明：**由于不同系统对文件名大小写处理不同，Windows不区分文件名大小写，而Linux区分。所以文件名命名均采用小写字母，多个单词之间可使用”\_”分隔符。

**实例：**disp.h   os\_sem.c

**规则1.2-2(建议)：工程源码使用GB2312编码方式。**

**说明：**程序里的注释可能会使用中文，GB2312是简体中文编码，大部分的编辑工具和集成IDE环境都支持GB2312编码，为避免中文乱码，建议使用GB2312对源码进行编码。若需要转换成其他编码格式，可使用文本编码转换工具进行转换。

**规则1.2-3(强制)：工程源码使用版本管理工具进行版本管理。**

**说明：**程序一般需要大量更新、修正、维护工作，且有时需要多人合作。使用版本管理工具可以帮助你提高工作效率。建议使用“Git”版本管理工具。

**1.3变量命名规则**

**原则1.3-1(强制)：变量命名应明确所代表的含义或者状态。**

**说明：**变量名称可以使用名词表述清楚的尽量使用名词，使用名词无法描述清楚时，使用形

容词或者描述性的单词+名词的形式。变量一般为实体的属性、状态等信息，使用上

      述方案一般可以解决变量名的命名问题，如果出现命名很困难或者无法给出合理的命

名方式时，问题可能出现在整体设计上，请重新审视设计。

**规则1.3-2(强制)：全局变量添加”*G\_*”前缀，全局静态变量添加” S\_ ”，局部静态变量添加”s\_”前缀。使用大小写混合方式命名，大写字母用于分割不同单词。**

**说明：**添加前缀的原因有两个。首先，使全局变量变得更醒目，提醒技术开发人员使用这些变量时要小心。其次，添加前缀使全局变量和静态变量变得和其他变量不一致，提醒技术开发人员尽量少用全局变量。

**实例：**

/\*\*出错信息       \*/

INT8U     G\_ErrMsg;

/\*\*每秒钟转动圈数 \*/

static INT32U   S\_CirclePerSec;

**规则1.3-3(强制)：局部变量使用小写字母，若标示符比较复杂，使用’\_’分隔符。**

**说明：**局部变量全部使用小写字母，和全局变量有明显区分，使读者看到标示符就知道是何

 种作用域的变量。

**实例：**

      INT32U  download\_program\_address;

**规则1.3-4(强制)：定义指针变量\*紧挨变量名，全局指针变量使用大写P前缀”P\_”，局部指针变量使用小写p前缀”p \_”。**

**实例：**INT8U  \*P\_MsgAddress;          /\*全局变量\*/

INT8U  \*p\_msg;                                  /\*局部变量\*/

**1.4函数命名规则**

**原则1.4-1(强制)：函数命名应该明确针对什么对象做出了什么操作。**

**说明：**函数的功能是获取、修改实体的属性、状态等，采用“动词+名词”的方式可以满足上述需求，若出现使用此方式命名函数很困难或不能命名的情况，问题可能出现在整体设计上，请重新审视设计方案。

**规则1.4-2(强制)：具有外部链接的函数命名使用大小写混合的方式，首字母大写，用于分割不同单词。**

**说明：**函数具有外部链接属性的含义是函数通过头文件对外声明后，对其他文件或模块来

说是可见的。如果一个函数要在其他模块或者文件中使用，需要在头文件中声明该函

数。另外，在头文件声明函数，还可以促使编译器检查函数声明和调用的一致性。

**实例：**

char  \*GetErrMsg(ErrMsg  \*msg);

**规则1.4-3(强制)：具有文件内部链接属性的函数命名使用小写字母，使用’\_’分隔符分割不同单词，且使用static关键字限制函数作用域。**

**说明：**函数具有内部链接属性的含义是函数只能在模块或文件内部调用，对文件或模块外来

说是不可见的。如果一个函数仅在模块内部或者文件内部使用，需要限制函数链接围，

使用static修饰符修饰函数，使其只具有内部链接属性。在源文件中声明一遍具有内

部链接的函数同样具有促使编译器检查函数声明和调用的一致性。

**实例：**

static  char  get\_key(void);

**规则1.4-4(强制)：函数参数使用小写字母，各单词之间使用“\_”分割，尽量保持参数顺序从左到右为：输入、修改、输出。**

**说明：**函数参数顺序为需输入参数值（这个值一般不修改，若不需要修改使用const关键字

修饰），需修改的参数（这个参数输入后用于提供数据，函数内部可以修改此参数），

输出参数（这个参数是函数输出值）。

**1.5常量的命名规则**

**规则1.5-1(强制)：常量（#define定义的常量、枚举、const定义的常量）的定义使用全大写字母，单词之间加 ’\_’分割的命名方式。**

**实例：**

 #define    PI\_ROUNDED   3.14

const double PI\_ROUNDED = 3.14;

enum      weekday{ SUN,MON,TUE,WED,THU,FRI,SAT };

**规则1.5-2(建议)：常数宏定义时，十六进制数的表示方法为0xFF。**

**说明：**前面0x中的x小写，数据中的”A-F”大写。

**1.6新定义的类型命名规范**

**规则1.6-1(强制)：新定义类型名的命名应该明确抽象对象的含义，新类型名使用大写字母，单词之间加’\_’分割，新类型指针在类型名前增加前缀”P\_”。成员变量标示符前加类型名称前缀，首字母大写用于区分各个单词。**

**实例：**typedef struct  \_STUDENT

{

    StudentName;

      StudentAge  ;

    ......

}STUDENT , \*P\_ STUDENT;/\* STUDENT 为新类型名称，P\_ STUDENT 为新类型指针名\*/

**2外观布局**

**2.1排版与格式**

**2.1.1 头文件排版**

**规则2.1.1-1(强制)：头文件排版内容依次为包含的头文件、宏定义、类型定义、声明变量、声明函数。且各个种类的内容间空三行。**

**说明：**头文件是模块对外的公用接口。在头文件中定义的宏，可以被其他模块引用。Project

      中不建议使用全部变量，若使用则需在头文件里对外声明。模块对外的函数接口在模

      块头文件里声明。

**2.1.2 源文件排版**

**规则2.1.2-1(强制)：源文件排版内容依次为包含的头文件、宏定义、具有外部链接属性的全局变量定义、模块内部使用的static变量、具有内部链接的函数声明、函数实现代码。且各个种类的内容间空三行。**

**说明：**模块内部定义的宏，只能在该模块内部使用。只在模块内部使用的函数，需在源码文

      件中声明，用于促使编译器检查函数声明和调用的一致性。

**规则2.1.2-2(强制)：程序块采用缩进风格编写，每级缩进4个空格。**

**说明：**当前主流IDE都支持Tab缩进，使用Tab缩进需要打开和设置相关选项。宏定义、编译开关、条件预处理语句可以顶格。

**规则2.1.2-3(强制)：if、for、do、while、case、switch、defaul、typedef等语句独占一行，且这些关键字后需空一格。**

**说明：**执行语句必须用缩进风格写，属于**if**、**for**、**do**、**while**、**case**、**switch**、**default**、**typedef**

等的下一个缩进级别。一般写**if**、**for**、**do**、**while**等语句都会有成对出现的{}‟，**if**、

**for**、**do**、**while**等语句后的执行语句建议增加成对的“{}”； 如果if/else语句块中只

有一条语句，也需增加“{}”。

**实例：**

for  (i  =  0; i  <  max\_num; i++)

{

for  (j  =  0;  j  <  max\_num; j++)

{

        If (name\_found)

                  {

                                                        语句

}

else

{

              语句

}

}

}

**规则2.1.2-4(强制)：进行双目运算、赋值时，操作符之前、之后要加空格；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：**采用这种方式书写代码，主要目的是使代码更清晰，使关键操作符更突出**。**

**实例：**

(1)    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 (a > b)

a += 2;

b = a ^ 3;

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

Search\_dowm = !true;

a++;

(3)    "->"、"."、”[]”前后不加空格。

Weight         =  G\_Car->weight;

eye           =  People.eye;

array[8]        =  8;

**规则2.1.2-5(建议)：一行只定义一个变量，一行只书写一条执行语句，多行同类操作时操作符尽量保持对齐。**

**说明：**一行定义一个变量，一行只书写一条执行语句，方便注释，多行同类操作对齐美观、整洁。

**实例：**

events\_rdy        =  OS\_FALSE;

events\_rdy\_nbr =  0;

events\_stat    =  OS\_STAT\_RDY;

pevents           =  pevents\_pend;

pevent            = \*pevents;

**规则2.1.2-6(建议)：函数内部局部变量定义和函数语句之间应空三行。**

**说明：**局部变量定义和函数语句是相对独立的，而且空三行可以更清晰地表示出这种独立性。

**3注释**

**3.1注释原则**

**原则3.1-1(强制)：注释的内容要清楚、明了，含义准确，在代码的功能、意图层次上进行注释。**

**说明：**注释的目的是让读者快速理解代码的意图。注释不是为了名词解释（what），而是说

明用途（why）。

**实例：**

如下注释纯属多余：

                   ++i;                                     // i增加1

if (data\_ready)                          /\* 如果data\_ready为真 \*/

如下注释无任何参考价值：

// 时间有限，现在是:04，根本来不及想为什么，也没人能帮我说清楚

**原则3.1-2(强制)：注释应分为两个角度进行，首先是应用角度，主要是告诉使用者如何使用接口（即你提供的函数），其次是实现角度，主要是告诉后期升级、维护的技术人员实现的原理和细节。**

**说明：**每一个产品都可以分为三个层次，产品本身是一个层次，这个层次之下的是你使用

的更小的组件，这个层次之上的是你为别人提供的服务。你这个产品的存在的价值

就在于把最底层的小部件的使用细节隐藏，同时给最上层的用户提供方便、简洁的

使用接口，满足用于需求。从这个角度来看软件的注释，你应该时刻想着你写的注释

是给那一层次的人员看的，如果是用户，那么你应该注重描述如何使用，如果是后期

维护者，那么你应该注重原理和实现细节。

**原则3.1-3(强制)：修改代码时，应维护代码周边的注释，使其代码和注释一致，不再使用的注释应删除。**

**说明：**注释的目的在于帮助读者快速理解代码使用方法或者实现细节，若注释和代码不一

      致，会起到相反的作用。建议在修改代码前应该先修改注释。

**规则3.1-4(建议)：代码段不应被“注释掉”（comment out ）。**

**说明：**当源代码段不需要被编译时，应该使用条件编译来完成（如带有注释的#if或#ifdef 结

构）。为这种目的使用注释的开始和结束标记是危险的，因为C 不支持/\*\*/嵌套的注

释，而且已经存在于代码段中的任何注释将影响执行的结果。

**3.2文件注释**

**规则3.2-1(强制)：文件注释需放到文件开头，具体格式见实例。**

**实例：**

stm32f10x\_dac.h

  /\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* @file     stm32f10x\_dac.h

\* @brief    Thisfile contains all the functions prototypes for the DAC firmware

  \*          library.

  \* @author  MCD Application Team

  \* @version  V3.5.0

  \* @date    11-March-2014

\* @par     Modification:添加函数，支持\*\*\*\*\*\*\*\*<br>

\*                     History

\*          Version：V3.0.1 <br>

\*          Author：\*\*\*<br>

\*          Modification:添加函数，支持\*\*\*\*\*\*\*\*<br>

\*          Version：V3.0.0 <br>

\*          Author：\*\*\*<br>

\*          Modification:添加函数，支持\*\*\*\*\*\*\*\*<br>

  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* @attention

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

**说明：**注释格式可被doxygen工具识别，其中@file、@brief、@author等是doxygen工具识别的关键字，注释内容可以为中文。

**3.3函数注释**

**规则3.3-1(强制)：函数注释分为头文件中函数原型声明时的注释和源文件中函数实现时的注释。头文件中的注释注重函数使用方法和注意事项，源文件中的注释注重函数实现原理和方法。具体格式见实例。**

**说明：**函数原型声明的注释按照doxygen工具可以识别的格式进行注释，用于doxygen工具

生成头文件信息以及函数间的调用关系信息。源代码实现主要是注释函数实现原理及

修改记录，不需按照doxygen工具要求的注释格式进行注释。

**实例：**

头文件函数原型声明注释：

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @brief     Configures the discontinuous mode for theselected ADC regular

\*           group channel.

\* @param    ADCx:where x can be 1, 2 or 3 to select the ADC peripheral.

\* @param    Number:specifies the discontinuous mode regular channel

\*           count value. This number must be between 1 and8.

\* @retval    None

\* @par      Usage：

\*           ADC\_DiscModeChannelCountConfig(ADC1,6);<br>

\* @par      Tag:

\*           此函数不能在中断里调用。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void ADC\_DiscModeChannelCountConfig(ADC\_TypeDef\* ADCx, INT8U\_tNumber);

源文件函数实现注释：

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \* @brief    Configures the discontinuousmode for the selected ADC regular

  \*          group channel.

  \* @param   ADCx: where x can be 1, 2 or 3 toselect the ADC peripheral.

  \* @param   Number: specifies the discontinuousmode regular channel

  \*          count value. This number must bebetween 1 and 8.

  \* @retval   None

  \* @par     Modification:修改了\*\*\*\*\*\*\*\*<br>

\*                       History

  \*          Modified by：\*\*\*<br>

\*           Date： 2013-10-10

  \*          Modification:修改了\*\*\*\*\*\*\*\*<br>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

  \*/

void ADC\_DiscModeChannelCountConfig(ADC\_TypeDef\*ADCx, INT8U\_t Number)

{

    赋值语句\*\*\*\*\*\*\*\*\*;       /\*关键语句的注释           \*/

语句\*\*\*\*\*\*\*\*\*\*\*;         /\*关键语句的注释格式       \*/

语句\*\*\*\*\*\*\*;             /\*实现\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*功能\*/

}

**3.4常量及全局变量注释**

**规则3.3-1(强制)：常量、全局变量需要注释，注释格式见实例。**

**实例：**

/\*\* Description of the macro \*/

#define XXXX\_XXX\_XX           0

/\*\*Description of global variable \*/

INT8U  G\_xxx  =  0;

**说明：**若全局变量在.c文件中定义，又在.h文件中声明，则在头文件中使用doxygen

格式注释，在源码文件中使用  /\* Description of the globalvariable \*/的形式。防止doxygen生成两遍注释文档信息。

**3.5局部变量及语句注释**

**规则3.3-1(强制)：局部变量，函数实现关键语句需要注释，注释格式见实例。**

**实例：**

 \*pq->OSQIn++ = pmsg;              /\* Insert message into queue          \*/

       pq->OSQEntries++;                /\* Update the nbr of entries in the queue\*/

      if (pq->OSQIn== pq->OSQEnd)

          {

pq->OSQIn = pq->OSQStart;    /\* Wrap IN ptr if we are at end of queue \*/

      }

**说明：**局部变量，关键语句需要注释，从功能和意图上进行注释，而不是代码的重复。多条注释语句尽量保持对齐，实现美观，整洁。

**材料：**

1.        《代码整洁之道》（RobertC.Martin 著 韩磊 译 人民邮电出版社2010年1月）第四章"注释”。

2.        《Doxygen中文手册》

**4项目版本号命名规范**

项目版本号管理是项目管理的重要方面，我们根据项目不同的开发阶段制定了不同的版本号命名规范。项目开发过程一般分为前期开发测试阶段、发布阶段、维护阶段这三个主要阶段，我们分别制定了命名规范。

**4.1开发、测试阶段版本号命名**

**规则4.1-1(强制)：处于开发、调试阶段的项目，版本号使用“V0.yz”的形式。**

**说明：**处于新开发、调试阶段的项目，版本号使用“V0.yz” 的形式，比如新开发的项目正处在开发、调试阶段，这时可以使用“ V0.10 ”这样的版本号。你认为完成了新的功能模块或整体架构做了很大的修改，可以根据情况增加 Y 或者 Z的值。比如，你开发阶段在“ V0.10 ”基础上新增加了一个功能模块你可以将版本号改为“V0.11”，做了比较大的修改，你可以将版本号定为“V0.20”。

**4.2正式发布阶段版本号命名**

**规则4.2-1(强制)：处于正式发布阶段的项目，版本号使用“Vx.y”的形式。**

**说明：**处于正式发布的项目版本号使用“Vx.y”的形式。比如，你发布了一个正式面向市场的项目，你可以使用“V1.0”作为正式的版本号。在“V1.0”基础上增加功能的正式版本，你可以使用“V1.1”作为下一次正式版本的版本号，在“V1.0”基础上修正了大的BUG或者做了很大的改动，你可以使用“V2.0”作为下一次正式版本号。

**4.3维护阶段版本号命名**

**规则4.3-1(强制)：处于维护阶段的项目，版本号使用“Vx.yz”的形式。**

**说明：**处于维护阶段的项目版本号使用“Vx.yz”的形式。比如在"V1.1"的基础上修改了一个功能实现算法以实现高效率，则可以使用"V1.11" 来表示这是在正式发布版本“V1.1”的基础上进行的一次修正，再次修正可以使用“V1.12”。

**5嵌入式软件安全性相关规范**

**5.1头文件**

**原则5.1-1(强制)：头文件用于声明模块对外接口，包括具有外部链接的函数原型声明、全局变量声明、定义的类型声明等。**

**说明：**头文件是模块（Module）或单元（Unit）的对外接口。头文件中应放置对外部的声

明，如对外提供的函数声明、宏定义、类型定义等。内部使用的函数声明不应放在

头文件中。 内部使用的宏、枚举、结构定义不应放入头文件中。变量定义不应放

在头文件中，应放在.c文件中。 变量的声明尽量不要放在头文件中，亦即尽量不要

使用全局变量作为接口。变量是模块或单元的内部实现细节，不应通过在头文件中

声明的方式直接暴露给外部，应通过函数接口的方式进行对外暴露。即使必须使用

全局变量，也只应当在.c中定义全局变量，在.h中仅声明变量为全局的。

**材料：**《C语言接口与实现》（David R. Hanson著 傅蓉 周鹏 张昆琪权威 译 机械工业出

版社 2004年1月）（英文版： "C Interfaces and Implementations"）

**规则5.1-2(强制)：只能通过包含头文件的方式使用其他.c提供的接口，禁止在.c中通过extern的方式使用外部函数接口、变量。**

**说明：**若a.c使用了b.c定义的foo()函数 ，则应当在b.h中声明externint foo(int input);并

在a.c中通过#include <b.h>来使用foo。禁止通过在a.c中写externint foo(int input);

来使用foo，后面这种写法容易在foo改变时可能导致声明和定义不一致。

**规则5.1-3(强制)：使用#define定义保护符，防止头文件重复包含。**

说明：多次包含一个头文件可以通过认真的设计来避免。如果不能做到这一点，就需要采取

阻止头文件内容被包含多于一次的机制。通常的手段是为每个文件配置一个宏，当头

文件第一次被包含时就定义这个宏，并在头文件被再次包含时使用它以排除文件内

容。所有头文件都应当使用#define 防止头文件被多重包含，命名格式FILENAME\_H\_，

其中FILENAME 为头文件的名称。

**实例：**

若文件名为：stm32f10x\_adc.h。

#ifndef          STM32F10x\_DAC\_H\_

#define             STM32F10x\_DAC\_H\_

…………

受保护的代码

#endif

**5.2预处理命令**

**规则5.2-1(强制)：C的宏只能扩展为用大括号括起来的初始化、常量、小括号括起来的**

**表达式、类型限定符、存储类标识符或do-while-zero 结构。**

**说明：**这些是宏当中所有可允许使用的形式。存储类标识符和类型限定符包括诸如extern 、

static和const这样的关键字。使用任何其他形式的#define 都可能导致非预期的行为，

或者是非常难懂的代码。特别的，宏不能用于定义语句或部分语句，除了do-while 结

构。宏也不能重定义语言的语法。宏的替换列表中的所有括号，不管哪种形式的 ()、

{} 、[]  都应该成对出现。 do-while-zero 结构（见下面实例）是在宏语句体中唯一可接受的具有完整语句的形式。do-while-zero 结构用于封装语句序列并确保其是正确的。

注意：在宏语句体的末尾必须省略分号。

**实例**：

                   以下是合理的宏定义：

       #define   PI 3.14159F                  /\*Constant \*/

        #define  XSTAL 10000000                /\*Constant \*/

        #define  CLOCK (XSTAL / 16)             /\*Constant expression \*/

        #define   PLUS2(X)  ( (X) + 2 )            /\* Macro expanding to expression \*/

        #define  STOR extern                  /\*storage class specifier \*/

        #define   INIT(value)  { (value), 0, 0 }       /\*braced initialiser \*/

        #define   READ\_TIME\_32() \

        do  { \

                     DISABLE\_INTERRUPTS(); \

                    time\_now = (INT32U) TIMER\_HI << 16; \

                    time\_now = time\_now | (INT32U) TIMER\_LO; \

                     ENABLE\_INTERRUPTS(); \

           } while(0)                           /\* example of do-while-zero \*/

             以下是不合理的宏定义：

                   #define unsigned int  long     /\* use typedef instead \*/

        #defineSTARTIF  if(           /\* unbalanced () and languageredefinition \*/

**规则5.2-2(强制)：在定义函数宏时，每个参数实例都应该以小括号括起来。**

**实例：**

一个abs 函数可以定义成：

     #define  abs (x) ( ( (x) >= 0 ) ? (x) : -(x) )

不能定义成：

    #define   abs(x)  ( ( (x) >= 0 ) ? x : -x )

如果不坚持本规则，那么当预处理器替代宏进入代码时，操作符优先顺序将不会给出要

求的结果。

考虑前面第二个不正确的定义被替代时会发生什么：

       z = abs ( a – b );

将给出如下结果：

       z = ( ( a – b >= 0 ) ? a – b : -a – b );

子表达式  – a - b  相当于 (-a)-b ，而不是希望的 –(a-b) 。把所有参数都括进小括号中就可以避免这样的问题。

**规则5.2-3(建议)：使用宏时，不允许参数数值发生变化。**

**实例：**

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。

正确的用法是：

b = SQUARE(a);

a++;                                      /\*结果：a = 6，即只执行了一次增\*/

同样建议在调用函数时，参数也不要变化，如果某次软件升级将其中一个接口由函数实现转换成宏，那参数数值发生变化的调用将产生非预期效果。

**规则5.2-4(建议)：除非必要，应尽可能使用函数代替宏。**

**说明：**宏能提供比函数优越的速度，但是没有参数检查机制，不当的使用可能产生非预期后果。

**5.3类型及类型转换**

**规则5.3-1(强制)：应该使用标明了大小和符号的typedef代替基本数据类型。不应使用基本数值类型char、int、short、long、float和double，而应使用typedef进行类型的定义。**

**说明：**为了程序的跨平台移植性，我们使用typedef定义指明了大小和符号的数据类型。

**实例：**

此实例是根据keil for ARM的数据类型大小进行的定义。

|  |  |  |
| --- | --- | --- |
| No. | 基本数据类型 | Typedef定义 |
| 1 | typedef unsigned char | BOOLEAN |
| 2 | typedef unsigned char | INT8U |
| 3 | typedef signed   char | INT8S |
| 4 | typedef unsigned short | INT16U |
| 5 | typedef signed   short | INT16S |
| 6 | typedef unsigned int | INT32U |
| 7 | typedef signed   int | INT32S |
| 8 | typedef float | FP32 |
| 9 | typedef double | FP64 |

应根据硬件平台和编译器的信息对基本类型进行定义。

**规则5.3-2(建议)：浮点应用应该适应于已定义的浮点标准。**

**说明：**浮点运算会带来许多问题，一些问题（而不是全部）可以通过适应已定义的标准来克服。其中一个合适的标准是  ANSI/IEEE Std 754 [1] 。

**5.3.1 显式数据类型转换**

C 语言给程序员提供了相当大的自由度并允许不同数值类型可以自动转换。由于某些功能

性的原因可以引入显式的强制转换，例如：

1.        用以改变类型使得后续的数值操作可以进行

2.        用以截取数值

3.        出于清晰的角度，用以执行显式的类型转换

为了代码清晰的目的而插入的强制转换通常是有用的，但如果过多使用就会导致程序的

可读性下降。正如下面所描述的，一些隐式转换是可以安全地忽略的，而另一些则不能。

**规则5.3.1-1(强制)：强制转换只能向表示范围更窄的方向转换，且与被转换对象的类**

**型具有相同的符号。浮点类型值只能强制转换到更窄的浮点类型。**

**说明：**这条规则主要是要求需要强制转换时，须明确被转换对象的表示范围及转换后的表示

范围。转换时尽量保持符号一致，不同符号对象之间不应出现强制转换。向更宽数据

范围转换并不能提高数据精确度，并没有实际意义。在程序中尽量规划好变量范围，

尽量少使用强制转换。

**规则5.3.1-2(强制)：如果位运算符 ~  和 <<  应用在基本类型为unsigned char或unsigned**

**short 的操作数，结果应该立即强制转换为操作数的基本类型。**

**说明：**当这些操作符（~ 和<<）用在 small integer 类型（unsigned char 或unsigned short ）时，运算之前要先进行整数提升，结果可能包含并非预期的高端数据位。

**例如：**

INT8U port= 0x5aU;

        INT8U result\_8;

        INT16U result\_16;

        INT16U mode;

       result\_8 = (~port) >> 4;    /\* 不合规范 \*/

~port的值在16位机器上是 0xffa5 ，而在 32 位机器上是 0xffffffa5 。在每种情况下，result的值是0xfa ，然而期望值可能是0x0a 。这样的危险可以通过如下所示的强制转换来避免：

result\_8 = ( (INT8U) (~port  )) >> 4;            /\* 符合规范 \*/

result\_16 = ( (INT16U ) (~(INT16U) port ) ) >> 4 ;   /\*符合规范  \*/

当<<操作符用在 smallinteger 类型时会遇到类似的问题，高端数据位被保留下来。

例如：

result\_16 = ( ( port << 4 ) & mode ) >> 6 ;         /\*不符合规范 \*/

result\_16 的值将依赖于 int 实现的大小。附加的强制转换可以避免任何模糊性。

result\_16 = ( ( INT16U) ( ( INT16U ) port << 4 ) & mode )>> 6 ;  /\*  符合规范 \*/

**5.3.2隐式类型转换**

**规则5.3.2-1(强制)：以下类型之间不应该存在隐式类型转换。**

**1)        有符号和无符号之间没有隐式转换**

**2)        整型和浮点类型之间没有隐式转换**

**3)        没有从宽类型向窄类型的隐式转换**

**4)        函数参数没有隐式转换**

**5)        函数的返回表达式没有隐式转换**

**6)        复杂表达式没有隐式转换**

**5.3.3 整数后缀**

**规则5.3.3-1(强制)：后缀“U”应该用在所有unsigned 类型的常量上。**

整型常量的类型是混淆的潜在来源，因为它依赖于许多因素的复杂组合，包括：

1)        常数的量级

2)        整数类型实现的大小

3)        任何后缀的存在

4)        数值表达的进制（即十进制、八进制或十六进制）

例如，整型常量“40000”在32位环境中是 int 类型，而在 16位环境中则是long 类型。

值0x8000 在16位环境中是 unsigned int 类型，而在 32 位环境中则是（signed ）int 类型。

注意：

1)        任何带有“U”后缀的值是unsigned 类型

2)        一个不带后缀的小于231的十进制值是signed 类型

但是：

1)        不带后缀的大于或等于215的十六进制数可能是 signed 或unsigned 类型

2)        不带后缀的大于或等于231的十进制数可能是 signed 或unsigned 类型

常量的符号应该明确。符号的一致性是构建良好形式的表达式的重要原则。如果一个常

数是unsigned 类型，为其加上“U”后缀将有助于避免混淆。当用在较大数值上时，后缀也许是多余的（在某种意义上它不会影响常量的类型）；然而后缀的存在对代码的清晰性是种有价值的帮助。

**5.3.4 指针类型转换**

指针类型可以归为如下几类：

1)        对象指针

2)        函数指针

3)        void 指针

4)        空（null ）指针常量（即由数值 0 强制转换为 void\*类型）

涉及指针类型的转换需要明确的强制，除非在以下时刻：

1)        转换发生在对象指针和void 指针之间，而且目标类型承载了源类型的所有类型标识符。

2)        当空指针常量（void\*）被赋值给任何类型的指针或与其做等值比较时，空指针常量被自动转化为特定的指针类型。

C 当中只定义了一些特定的指针类型转换，而一些转换的行为是实现定义的。

**规则5.3.9-1(强制)：转换不能发生在函数指针和其他除了整型之外的任何类型指针之间。**

**[Undefined]**

**说明：**

函数指针到不同类型指针的转换会导致未定义的行为。这意味着一个函数指

针不能转换成指向不同类型函数的指针。

**规则5.3.9-2(强制)：对象指针和其他除整型之外的任何类型指针之间、对象指针和其他类**

**型对象的指针之间、对象指针和void指针之间不能进行转换。**

**[Undefined]**

**规则5.3.9-3(强制)：不应在某类型对象指针和其他不同类型对象指针之间进行强制转换。**

**说明：**如果新的指针类型需要更严格的分配时这样的转换可能是无效的。

**实例：**

INT8U   \*p1;

                 INT32U  \*p2;

                p2= (INT32U \*) p1;  /\*不符规范\*/

**5.4初始化、声明与定义**

**规则5.4-1(强制)：所有自动变量在使用前都应被赋值。**

**[Undefined]**

**说明：**注意，根据ISO C[2] 标准，具有静态存储期的变量缺省地被自动赋予零值，除非经过了显式的初始化。实际中，一些嵌入式环境没有实现这样的缺省行为。静态存储期是所有以static存储类形式声明的变量或具有外部链接的变量的共同属性，自动存储期变量通常不是自动初始化的。

**规则5.4-2(强制)：应该使用大括号以指示和匹配数组和结构的非零初始化构造。**

**[Undefined]**

**说明：**

              ISO C[2]要求数组、结构和联合的初始化列表要以一对大括号括起来（尽管不这样做的行为是未定义的）。本规则更进一步地要求，使用附加的大括号来指示嵌套的结构。它迫使程序员显式地考虑和描述复杂数据类型元素（比如，多维数组）的初始化次序。

例如，下面的例子是二维数组初始化的有效（在ISO C [2]中）形式，但第一个与本规

则相违背：

在结构中以及在结构、数组和其他类型的嵌套组合中，规则类似。

还要注意的是，数组或结构的元素可以通过只初始化其首元素的方式初始化（为 0 或

NULL）。如果选择了这样的初始化方法，那么首元素应该被初始化为0（或NULL），

此时不需要使用嵌套的大括号。

**实例：**

INT16U  test[3][2] = { 1, 2, 3, 4, 5, 6 };          /\* 不符合此规则    \*/

INT16U  test[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };   /\* 符合此规则       \*/

**规则5.4-3(强制)：在枚举列表中，“= ”不能显式用于除首元素之外的元素上，除非所有**

**的元素都是显式初始化的。**

**说明：**

如果枚举列表的成员没有显式地初始化，那么C 将为其分配一个从0 开始的整数序列，首元素为0 ，后续元素依次加 1 。

如上规则允许的，首元素的显式初始化迫使整数的分配从这个给定的值开始。当采用这种方法时，重要的是确保所用初始化值一定要足够小，这样列表中的后续值就不会超出该枚举常量所用的int 存储量。

列表中所有项目的显式初始化也是允许的，它防止了易产生错误的自动与手动分配的混合。然而，程序员就该担负职责以保证所有值都处在要求的范围内以及值不是被无意复制的。

**实例：**

enum  colour { red = 3, blue, green, yellow = 5 };  /\* 不符合此规则 \*/

enum  colour { red = 3, blue = 4, green = 5, yellow= 5 };  /\* 符合此规则 \*/

虽然green和yellow的值都是5，但这符合规则。

enum  colour { red = 1, blue, green, yellow };  /\* 符合此规则 \*/

**规则5.4-4(强制)：函数应当具有原型声明，且原型在函数的定义和调用范围内都是可见**

**的。**

**[Undefined]**

**说明：**原型的使用使得编译器能够检查函数定义和调用的完整性。如果没有原型，就不会迫使编译器检查出函数调用当中的一定错误（比如，函数体具有不同的参数数目，调用和定义之间参数类型的不匹配）。事实证明，函数接口是相当多问题的肇因，因此本规则是相当重要的。对外部函数来说，我们建议采用如下方法，在头文件中声明函数（亦即给出其原型），并在所有需要该函数原型的代码文件中包含这个头文件，在实现函数功能的.c文件中也包含具有原型声明的头文件。 为具有内部链接的函数给出其原型也是良好的编程实践。

**规则5.4-5(强制)：定义或声明对象、函数时都应该显示指明其类型。**

**规则5.4-6(强制)：函数的每个参数类型在声明和定义中必须是等同的，函数的返回类型**

**也该是等同的。**

**[Undefined]**

**规则5.4-6(强制)：函数应该声明为具有文件作用域。**

**[Undefined]**

**说明：**在块作用域中声明函数会引起混淆并可能导致未定义的行为。

**规则5.4-7(强制)：在文件范围内声明和定义的所有对象或函数应该具有内部链接，除非**

**是在需要外部链接的情况下，具有内部链接属性的对象或函数应该使用static关键字修饰。**

**说明：**如果一个变量只是被同一文件中的函数所使用，那么就用static。类似地，如果一个函数只是在同一文件中的其他地方调用，那么就用 static。使用 static存储类标识符将确保标识符只是在声明它的文件中是可见的，并且避免了和其他文件或库中的相同标识符发生混淆的可能性。具有外部链接属性的对象或函数在相应模块的头文件中声明，在需要使用这些接口的模块中包含此头文件。

**规则5.4-8(强制)：当一个数组声明为具有外部链接，它的大小应该显式声明或者通过初**

**始化进行隐式定义。**

**[Undefined]**

**实例：**

INT8U array[10] ;               /\* 符合规范  \*/

extern  INT8U  array[] ;         /\* 不符合规范\*/

INT8U  array[] = { 0, 10, 15};       /\* 符合规范   \*/

尽管可以在数组声明不完善时访问其元素，然而仍然是在数组的大小可以显式确定的情

况下，这样做才会更为安全。

**5.5控制语句和表达式**

**规则5.5-1(建议)：不要过分依赖C 表达式中的运算符优先规则。**

**说明：**括号的使用除了可以覆盖缺省的运算符优先级以外，还可以用来强调所使用的运算符。使用相当复杂的C 运算符优先级规则很容易引起错误，那么这种方法就可以帮助避免这样的错误，并且可以使得代码更为清晰可读。然而，过多的括号会分散代码使其降低了可读性**。**因此，请合理使用括号来提高程序清晰度和可读性。

**规则5.5-1(强制)：不能在具有副作用的表达式中使用sizeof 运算符。**

**说明：**当一个表达式使用了sizeof运算符，并期望计算表达式的值时，表达式是不会被计算的。sizeof只对表达式的类型有用。

实例：

INT32S  i;

             INT32S  j;

           j = sizeof (i = 1234); /\* j的值是i类型的大小，但i的值并没有赋值成1234 \*/

**规则5.5-2(强制)：逻辑运算符 && 或 || 的右手操作数不能包含副作用。**

**说明：**C语言中存在表达式的某些部分不会被计算到，这取决于表达式中的其他部分。逻辑操作符&&或||在进行逻辑判断时，若仅判别左操作数就能确定true or false的情况下，逻辑操作符的右操数将被忽略。

**实例：**

if ( high && ( x == i++ ) )    /\* 不符合规则 \*/

           若high为false，则整个表达式的布尔值也即为false，不用再去执行和判断右操作数。

**规则5.5-3(建议)：逻辑运算符（&&、| |  和 ! ）的操作数应该是有效的布尔数。有效布尔**

**类型的表达式不能用做非逻辑运算符（&&、| |  和 ! ）的操作数。**

**说明：**有效布尔类型是表示真、假的一种数据类型，产生布尔类型的可以是比较，逻辑运算，但布尔类型数据只能进行逻辑运算。

**规则5.5-4(强制)：位运算符不能用于基本类型（underlying type ）是有符号的操作数上。**

**[Implementation-defined]**

**说明：**位运算（~ 、<<、>>、&、^  和 |  ）对有符号整数通常是无意义的。比如，如果右移运算把符号位移动到数据位上或者左移运算把数据位移动到符号位上，就会产生问题。

**规则5.5-6(建议)：在一个表达式中，自增（++）和自减（- - ）运算符不应同其他运算符**

**混合在一起。**

**说明：**不建议使用同其他算术运算符混合在一起的自增和自减运算符是因为

1）它显著削弱了代码的可读性;

2）在不同的变异环境下，会执行不同的运算次序，产生不同结果。

**实例：**

u8a = ++u8b +u8c--;  /\* 不符合规范 \*/

下面的序列更为清晰和安全：

++u8b;

                u8a = u8b + u8c;

                 u8c--;

**规则5.5-7(强制)：浮点表达式不能做像‘>’ ‘<’ ‘==’ ‘!=’等 关系运算。**

**说明：**float、double类型的数据都有一定的精确度限制，使用不同浮点数表示规范或者不同硬件平台可能导致关系运算的结果不一致。

**规则5.5-8(强制)：for语句的三个表达式应该只关注循环控制，for循环中用于计数的变量不应在循环体中修改。**

**说明：**for 语句的三个表达式都给出时它们应该只用于如下目的：

第一个表达式初始化循环计数器；

第二个表达式包含对循环计数器和其他可选的循环控制变量的测试；

第三个表达式循环计数器的递增或递减。

**规则5.5-9(强制)：组成switch、while、do...while 或for 结构体的语句应该是复合语句。即使该复合语句只包含一条语句也要扩在{}里。**

**实例：**

for ( i = 0 ; i< N\_ELEMENTS ; ++i )

    {

         buffer[i] = 0;    /\* 仅有一条语句也需使用{} \*/

    }

**规则5.5-10(强制)：if /else应该成对出现。所有的if ... else if 结构应该由else 子句结束。**

**规则5.5-11(强制)：switch 语句中如果case 分支的内容不为空，那么必须以break 作为结束，最后分支应该是default分支。**

**5.6函数**

**原则5.6-1(强制)：编写整洁函数，同时把代码有效组织起来。**

**说明：**代码简单直接、不隐藏设计者的意图、用干净利落的抽象和直截了当的控制语句将函数有机组织起来。代码的有效组织包括：逻辑层组织和物理层组织两个方面。逻辑层，主要是把不同功能的函数通过某种联系组织起来，主要关注模块间的接口，也就是模块的架构。物理层，无论使用什么样的目录或者名字空间等，需要把函数用一种标准的方法组织起来。例如：设计良好的目录结构、函数名字、文件组织等，这样可以方便查找。

**规则5.6-2(强制)：一定要显示声明函数的返回值类型，及所带的参数。如果没有要声明为void。**

**说明：**C语言中不加类型说明的函数，一律自动按整型处理。

**规则5.6-3(建议)：不建议使用递归函数调用。**

**说明：**有些算法使用分而治之的递归思想，但在嵌入式中栈空间有限，递归本身承载着可

用堆栈空间过度的危险，这能导致严重的错误。除非递归经过了非常严格的控制，

否则不可能在执行之前确定什么是最坏情况（worst-case）的堆栈使用。

**5.7指针与数组**

**规则5.7-1(强制)：除了指向同一数组的指针外，不能用指针进行数学运算，不能进行关系运算。**

**说明：**这样做的目的一是使代码清晰易读，另外避免访问无效的内存地址。

**规则5.7-2(强制)：指针在使用前一定要赋值，避免产生野指针。**

**规则5.7-3(强制)：不要返回局部变量的地址。**

**说明：**局部变量是在栈中分配的，函数返回后占用的内存会释放，继续使用这样的内存是

危险的。因此，应该避免出现这样的危险。

**实例：**

           INT8U \*foobar (void)

          {

                  INT8U local\_auto;

                 return(&local\_auto);     /\* 不符合规范 \*/

      }

**5.8结构与联合**

**原则5.8-1(强制)：结构功能单一，不要设计面面俱到的数据结构。**

**说明：**相关的一组信息才是构成一个结构体的基础，结构的定义应该可以明确的描述一个对象，而不是一组相关性不强的数据的集合。设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

**5.9标准库**

**规则5.9-1(强制)：标准库中保留的标识符、宏和函数不能被定义、重定义或取消定义。**

**[Undefined]**

**说明：**通常 #undef  一个定义在标准库中的宏是件坏事。同样不好的是，#define 一个宏名字，而该名字是C 的保留标识符或者标准库中做为宏、对象或函数名字的C 关键字。例如，存在一些特殊的保留字和函数名字，它们的作用为人所熟知，如果对它们重新定义或取消定义就会产生一些未定义的行为。这些名字包括defined、\_\_LINE\_\_、\_\_FILE\_\_、\_\_DATE\_\_ 、\_\_TIME\_\_、\_\_STDC\_\_、errno和assert。

**规则5.9-2(强制)：传递给库函数的值必须检查其有效性。**

**说明：**C 标准库中的许多函数根据ISO [2] 标准 并不需要检查传递给它们的参数的有效性。即使标准要求这样，或者编译器的编写者声明要这么做，也不能保证会做出充分的检查。因此，程序员应该为所有带有严格输入域的库函数（标准库、第三方库及自己定义的库）提供适当的输入值检查机制。

具有严格输入域并需要检查的函数例子为：

math.h 中的许多数学函数，比如：

负数不能传递给sqrt 或log函数；

fmod 函数的第二个参数不能为零

  toupper 和tolower：当传递给toupper函数的参数不是小写字符时，某些实现能产生并非预期的结果（tolower 函数情况类似）

     如果为ctype.h 中的字符测试函数传递无效的值时会给出未定义的行为

     应用于大多数负整数的abs 函数给出未定义的行为 在math.h 中，尽管大多数数学库函数定义了它们允许的输入域，但在域发生错误时它们的返回值仍可能随编译器的不同而不同。因此，对这些函数来说，预先检查其输入值的有效性就变得至关重要。

程序员在使用函数时，应该识别应用于这些函数之上的任何的域限制（这些限制可能

会也可能不会在文档中说明），并且要提供适当的检查以确认这些输入值位于各自域

中。当然，在需要时，这些值还可以更进一步加以限制。

有许多方法可以满足本规则的要求，包括：

1.  调用函数前检查输入值

2.  设计深入函数内部的检查手段。这种方法尤其适应于实验室内开发的库，纵然它也可以用于买进的第三方库（如果第三方库的供应商声明他们已内置了检查的话）。

3.  产生函数的“封装”（wrapped）版本，在该版本中首先检查输入，然后调用原始的函数。

4.  静态地声明输入参数永远不会采取无效的值。

注意，在检查函数的浮点参数时（浮点参数在零点上为奇点），适当的做法是执行其是否为零的检查。然而如果当参数趋近于零时，函数值的量级趋近无穷的话，仍然有必要检查其在零点（或其他任何奇点）上的容限，这样可以避免溢出的发生。

**6参考文献**

[1]  ANSI/IEEE Std 754, IEEE Standard for Binary Floating-Point Arithmetic,1985

[2]   ISO/IEC 9899:1990. Programming languages - C.International Organization for

Standardization. 1990

[3]  GB/T 15272-94 程序设计语言 C

[4]  MISRA-C-:2004 Guidelines for the use of the C language in critical systems