实验室代码书写规范

作者：郭宗辉

年级：2010

目录

1. 概述

为规范实验室人员的代码编写提供参考依据和统一标准。本文档适合所有人员使用，为个人和实验室发展有深远的意义，请大家遵守。

2. 排版要求

2.1 程序块缩进

缩进的每级之间有四个空格。

2.2 程序块之间空行

相对独立的程序块之间、变量说明之后必须加空行。

示例：如下不符合规范

```
if    (account == true)
{
        … // program code
}
name = wang;
```

应如下书写

```
if    (account == true)
{
        … // program code
}
```

```
name = wang;
```

2.3 长语句和长表达式

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别操作符处划分新行，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：

```
Report_or_not_flag = ((taskno < MAX_ACT_TASK_NUMBER)
                                    && (act_task.result !=0));
```

2.4 循环、判断等长表达式或语句

循环、判断等语句中若有较长的表达式或语句，则要进行适当的划分，长表达式要在低优先级别操作符处划分新行，操作符放在新行之首。

2.5 长参数

若函数或过程中的参数较长，则要进行适当的划分。

2.6 短语句

不允许把多个短语句放在一行中，即一行只写一条语句。

2.7 条件、循环语句

if、for、do、while、case、switch、default 等语句多占一行，而且 if、for、do、while 等语句的执行语句部分无论多少行都要加{}。

2.8 语句对齐

对齐只使用空格键，不使用 TAB 键。

说明：以免使用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐。

2.9 函数、过程和结构等语句块

函数或过程的开始、结构的定义及循环、判断等语句的代码都要采用缩进风格，case 语句下的情况处理语句也要符合语句缩进要求。

2.10 程序块分界符

程序块的分界符（如 C/C++语言的大括号”{”和”}”）应各自独占一行且位于同一列，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、swith、case 语句中的程序都要采用如上的缩进方式。

示例：

如下不符合规范：

```
for （...） {
    ... // program code
}
```

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

应如下书写：

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

2.11 操作符前后空格

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度是相对的，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不需要加空格，多重括号间不必加空格。

在长语句中，如果需要加的空格非常多，那么应保持整体清晰，给操作符留空格时不要连续留两个以上的空格。

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。

（3）"!"、"++"、"--"等单目操作符前后不加空格。

（4）if、for、while、swith 等与后面的括号间应加空格。

3. 注释

3.1 有效注释量

一般情况下，源程序有效注释量必须在 20%以上。

说明：注释的原则是有助于对程序的阅读理解，注释语言必须准确、易懂、简洁。

3.2 说明性文件

说明性文件（如头文件.h 文件、.inc 文件、def 文件等）头部应进行注释，注释必须列出生成日期、作者、内容、功能、与其他文件关系、修改日志，函数功能的简要说明。

3.3 函数头部说明

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、调用关系等。

示例：

```
/*************************************************
Function :        // 函数名称
Description：     // 函数功能、性能等的描述
Calls：           // 被本函数调用的函数清单
Called By：       // 调用本函数的函数清单
Table Accessed：  // 被访问的表
Table Updated：   // 被修改的表
Input：           // 输入参数说明，包括每个参数的作用、取值说
                     明及参数间的关系
Output：          // 对输出参数的说明
Return：          // 函数返回值说明
Others：          // 其他说明
*************************************************/
```

3.4 注释与代码一致

边写代码边写注释，修改代码同时修改相应的注释，以保证注释与代码一致。不再有用的注释要及时删除。

3.5 注释内容

注释的内容要清楚、明了，含义准确。

避免在注释中使用缩写。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

3.6 注释位置

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相邻的位置，不可放在下方，如果放在上方应要与其上面的代码用空行隔开。

3.7 变量、常量注释

对于所有有物理含义的变量、常量，如果其命名不是充分表明其含义的，在声明是就必须加以注释，说明其物理含义。

3.8 数据结构的注释

数据结构声明（包括数组、结构、类、枚举等），如果其命名不能充分表明其含义的，必须加以注释。

3.9 全局变量

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时注意事项等说明。

3.10 注释缩排

注释与所描述内容进行同样的缩排。

示例：如下排版不规范

```
void example_fun （void）
{
/* code one comments */
    Codeblock one
        /*code two comments*/
    Codeblock two
}
```

应该为如下布局：

```
void example_fun （void）
{
    /* code one comments */
    Codeblock one

    /*code two comments*/
    Codeblock two
```

}

3.11 注释与代码之间空行

将注释与其上面的代码用空行隔开。

示例：示例：如下排版不规范

/* code one comments */

Codeblock one

/*code two comments*/

Codeblock two

应该为如下布局：

/* code one comments */

Codeblock one

/*code two comments*/

Codeblock two

3.12 变量定义、分支语句

对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。

3.13 其他

1、避免在一行代码或表达式的中间插入注释（除非必要）。

2 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，是代码成为自注释的。

3、在代码功能、意图层次上进行注释，提供有用、额外的信息。

4、在程序块的结束行右方加注释标记，以表明某程序块的结束。

5、注释格式尽量统一，建议使用”/* •••• */”。

6、注释应考虑程序的易读以及外观排版的因素，使用的语言若是中、英兼有，建议多使用中文，除非能用非常流利准确的英文表达。

4. 标识符命名

4.1 命名清晰

标识符的命名要清晰、明了、有明确的含义，同时使用完整单词或大家基本可以理解的缩写，避免使人产生误解。

说明：较短的单词可以通过去掉“元音”形成缩写；较长的单词可取单词的前几个字母。

4.2 变量命名

对于变量命名，禁止取单个字符，建议除了要具有具体含义外，还能表明其变量类型、数据类型等，但是 i、j、k 等作局部循环变量是允许的。

4.3 互斥意义的变量

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	open / close	start / stop	show / hide
add / delete	min / max	next / previous	send / receive
cut / paste	lock / unlock	old / new	source / target
source / destination		up / down	

示例：int min_sum;

int max_sum;

int add_user(BYTE *user_name);

int delete_user(BYTE *user_name);

4.4 其它

1、命名规范要与相关语言标准和所用系统的风格保持一致。

2、除非必要，不要用数字或较奇怪的字符来定义标识符。

3、除了编辑开关/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定义。

5. 可读性

5.1 运算符优先级

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。

5.2 避免直接使用数字作为标识符

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
}
```

5.3 其它

1、源程序中关系较为紧密的代码应尽可能相邻。

2、不要使用难懂得技巧性很高的语句，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;
```

6. 变量、结构

6.1 公共变量

去掉没有必要的公共变量。

说明：公共变量是增大模块间耦合的原因之一，估应减少没必要的公共变量以降低模块间的耦合度。

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

6.2 公共变量访问说明

明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。

说明：明确过程操作变量的关系后，将有利于程序进一步优化、单元测试、系统联调以及代码维护等，这种关系的说明可在注释或文档中描述。

示例：在源文件中，可按如下注释形式说明：

RELATION	System_Int	Input_Rec	Print_Rec	Stat_Source
Student	Create	Modify	Access	Access
Score	Create	Modify	Access	Access, Modify

注：RELATION 为操作关系；System_Init、Input_Rec、Print_Rec、Stat_Score 为四个不 同的函数；Student、Score 为两个全局变量；Create 表示创建，Modify 表示修改，Access 表 示访问。

其中，函数 Input_Rec、Stat_Score 都可修改变量 Score，故此变量将引起函数间较大的耦合， 并可能增加代码测试、维护的难度。

6.3 公共变量赋值

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。 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。

说明：对公共变量赋值时，若有必要应进行合法性检查，以提高代码的可靠性、稳定性。

6.4 防止局部变量与公共变量同名

若使用了较好的命名规则，那么此问题可自动消除。

6.5 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。

6.6 其它

1、防止多个模块公用公共变量。

说明：构造仅有一个函数或模块可以修改、创建，而其余有关函数或模块只访问的公共变量，防止多个不同的函数或模块都可以修改、创建同一公共变量的现象。

2、结构功能要单一，是针对一种事物的抽象。

3、不同结构间的关系不要过于复杂。

说明：若两个结构间的关系较为复杂、密切，那么应合为一个结构。

4、结构中的元素的个数要适中。

5、编程时要注意数据类型的强制转换。

6、合理地设计数据并使用自定义数据类型，避免数据间进行不必要的数据类型转换。

7. 函数、过程

7.1 对所调用函数的错误返回码要仔细、全面地处理

7.2 明确函数的功能，精确（而不是近似）地实现函数设计

7.3 局部变量

编写可重入函数时，应注意局部变量的使用（如编写 C/C++语言的可重入函数时，应使用 auto 即缺省态局部变量或寄存器变量）。

说明：编写 C/C++语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

7.4 全局变量

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。

说明：若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量变为不可知状态。

7.5 其它

1、防止函数的参数作为工作变量。

2、函数的规模尽量限制在 200 行内（不包括注释和空格行）

3、一个函数仅完成一件功能。

4、为简单功能编写函数，可增加程序的可读性，便于维护、测试。

5、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相同的输出。

6、尽量保持函数的对立性，减少对其他函数的依赖性。

7、避免设计多参数函数，不使用的参数在接口中去掉。

8、检查函数输入、输出的有效性。

9、函数名应准确描述函数的功能。

10、防止把没有关联的语句放在函数中。

11、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。

12、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。

8. 可测性

代码的可测试性也是代码的非常重要部分。主要是断言的使用。

9. 程序效率

9.1 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。

9.2 提高代码效率

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。

9.3 其它

1、全局效率应高于局部效率。

2、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代码的空间效率。

3、循环体内工作量的最小化。

4、在多重循环中应将最忙的循环放在最内层。

5、对模块中的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。

6、尽量减少循环嵌套层次。

7、尽量避免循环体内包含判断语句。

8、尽量有乘法或其它方法代替除法，特别是浮点运算中的除法。

9、不要一味追求紧凑的代码，因为紧凑的代码不代表高效率。

10. 总结

本文档是参照华为的代码规范撰写的，我现在所用到的编程语言有限，经验更是有限，所以文档内容不全面，希望大家将本规范实际

运用到编写代码中，并提出宝贵意见。