DKBA

华为技术有限公司内部技术规范

DKBA 2826-2016.05

C语言编程规范



2016年5月28日发布 2016年5月28日实施

华为技术有限公司

Huawei Technologies Co., Ltd.

版权所有 侵权必究

All rights reserved

修订声明Revision declaration

本规范拟制与解释部门：

本规范的相关系列规范或文件：

相关国际规范或文件一致性：

替代或作废的其它规范或文件：

相关规范或文件的相互关系：

|  |  |  |  |
| --- | --- | --- | --- |
| 规范号 | 主要起草部门专家 | 主要评审部门专家 | 修订情况 |
| DKBA2826-2011.05 | PSST质量部：  郭曙光00121837  网络：张伟00118807  周灿00056781  王晶00041937  陈艺彪00036913  IP开发部：  薛治00038309  核心网：  张小林00058208  王德喜00040674  李明胜00042021  软件公司：  文 滔00119601  无线：  刘爱华00162172  中研：  谭洪00162654 | PSST质量部：  李重霄00117374  郭永生00120218  核心网：  张进柏00120359  中研：  张建保00116237  无线：  苏光牛00118740  郑铭00118617  陶永祥00120482  软件公司：  周代兵00120359  刘心红00118478  朱文琦00172539  网络：  王玎00168059  黄维东49827  IP开发部：  饶远00152313 |  |
| DKBA2826-2013.07 | 研发能力中心  郭曙光00121837 | 网络：  张伟00118807  研发能力中心  王红超00 134169 | 修改部分规则的说明和例子。 |
| DKBA2826-2016.05 | 研发能力中心  郭曙光00340987 | 网络：  张伟00342388  周灿00286455  IP开发部：  陈艺彪00223933  无线：  邱霖00340162  电软：  赵玉锡00232229  研发能力中心：  吴敏00326311  刘进00283514  曹锦业00246228  陈宇00291422  朱喜红00210657 | 新增规则1.9、规则2.8,、规则2.9、建议2.7、建议2.8、建议2.9、规则4.4、建议4.6、建议4.7、建议4.8、规则6.6、规则6.7、规则8.9、建议8.4、建议8.5、建议8.6、规则9.7、建议11.2、建议11.2。  安全章节整体移出到C&C++安全编程规范。  单元测试章节合入可测性章节。 |

目 录Table of Contents

0 规范制订说明 5

0.1 前言 5

0.2 代码总体原则 5

0.3 规范实施、解释 6

0.4 术语定义 6

1 头文件 6

2 函数 12

3 标识符命名与定义 21

3.1 通用命名规则 21

3.2 文件命名规则 23

3.3 变量命名规则 23

3.4 函数命名规则 24

3.5 宏的命名规则 24

4 变量 25

5 宏、常量 28

6 质量保证 32

7 程序效率 36

8 注释 39

9 排版与格式 44

10 表达式 46

11 代码编辑、编译 49

12 可测性 50

13 安全性 51

13.1 字符串操作安全 51

13.2 整数安全 52

13.3 格式化输出安全 56

13.4 文件I/O安全 57

13.5 其它 59

14 单元测试 59

15 可移植性 60

16 业界编程规范 60

C语言编程规范

范 围:

本规范适用于公司内使用C语言编码的所有软件。本规范自发布之日起生效，以后新编写的和修改的代码应遵守本规范。

简 介：

本规范制定了编写C语言程序的基本原则、规则和建议。从代码的清晰、简洁、可测试、安全、程序效率、可移植各个方面对C语言编程作出了具体指导。

# 规范制订说明

## 前言

为提高产品代码质量，指导广大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，编程规范修订工作组分析、总结了我司的各种典型编码问题，并参考了业界编程规范近年来的成果，重新对我司1999年版编程规范进行了梳理、优化、刷新，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。

## 代码总体原则

**1、清晰第一**

**清晰性是易于维护、易于重构的程序必需具备的特征。**代码首先是给人读的，好的代码应当可以像文章一样发声朗诵出来。

目前软件维护期成本占整个生命周期成本的40%~90%。根据业界经验，维护期变更代码的成本，小型系统是开发期的5倍，大型系统（100万行代码以上）可以达到100倍。业界的调查指出，开发组平均大约一半的人力用于弥补过去的错误，而不是添加新的功能来帮助公司提高竞争力。

“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

本规范通过后文中的原则（如优秀的代码可以自我解释，不通过注释即可轻易读懂/头文件中适合放置接口的声明，不适合放置实现/除了常见的通用缩写以外，不使用单词缩写，不得使用汉语拼音）、规则（如防止局部变量与全局变量同名）等说明清晰的重要性。

一般情况下，代码的可阅读性高于性能，只有确定性能是瓶颈时，才应该主动优化。

**2、简洁为美**

**简洁就是易于理解并且易于实现。**代码越长越难以看懂，也就越容易在修改时引入错误。写的代码越多，意味着出错的地方越多，也就意味着代码的可靠性越低。因此，我们提倡大家通过编写简洁明了的代码来提升代码可靠性。

废弃的代码(没有被调用的函数和全局变量)要及时清除，重复代码应该尽可能提炼成函数。

本规范通过后文中的原则（如文件应当职责单一/一个函数仅完成一件功能）、规则（重复代码应该尽可能提炼成函数/避免函数过长，新增函数不超过50行）等说明简洁的重要性。

**3、选择合适的风格，与代码原有风格保持一致**

产品所有人共同分享同一种风格所带来的好处，远远超出为了统一而付出的代价。在公司已有编码规范的指导下，审慎地编排代码以使代码尽可能清晰，是一项非常重要的技能。**如果重构/修改其他风格的代码时，比较明智的做法是根据现有代码的现有风格继续编写代码**，或者使用格式转换工具进行转换成公司内部风格。

## 规范实施、解释

本规范制定了编写C语言程序的基本原则、规则和建议。

本规范适用于公司内使用C语言编码的所有软件。本规范自发布之日起生效，对以后新编写的和修改的代码应遵守本规范。

本规范由系统工程体系发布和维护。实施中遇到问题，可以到论坛<http://hi3ms.huawei.com/group/1735/threads.html>上讨论。

在某些情况下（如BSP软件）需要违反本文档给出的规则时，相关团队必须通过一个正式的流程来评审、决策规则违反的部分，个体程序员不得违反本规范中的相关规则。

## 术语定义

**原则**：编程时必须坚持的指导思想。

**规则**：编程时强制必须遵守的约定。

**建议**：编程时必须加以考虑的约定。

**说明**：对此原则/规则/建议进行必要的解释。

**示例**：对此原则/规则/建议从正、反两个方面给出例子。

**延伸阅读材料**：建议进一步阅读的参考材料。

# 头文件

背景

**对于C语言来说，头文件的设计体现了大部分的系统设计。**不合理的头文件布局是编译时间过长的根因，不合理的头文件实际上反映了不合理的设计。

术语定义：

**依赖**：本章节特指编译依赖。若x.h包含了y.h，则称作x依赖y。依赖关系会进行传导，如x.h包含y.h，而y.h又包含了z.h，则x通过y依赖了z。依赖将导致编译时间的上升。虽然依赖是不可避免的，也是必须的，但是不良的设计会导致整个系统的依赖关系无比复杂，使得任意一个文件的修改都要重新编译整个系统，导致编译时间巨幅上升。

在一个设计良好的系统中，修改一个文件，只需要重新编译数个，甚至是一个文件。

某产品曾经做过一个实验，把所有函数的实现通过工具注释掉，其编译时间只减少了不到10%，究其原因，在于A包含B，B包含C，C包含D，最终几乎每一个源文件都包含了项目组所有的头文件，从而导致绝大部分编译时间都花在解析头文件上。

某产品更有一个“优秀实践”，用于将.c文件通过工具合并成一个比较大的.c文件，从而大幅度提高编译效率。其根本原因还是在于通过合并.c文件减少了头文件解析次数。但是，这样的“优秀实践”是对合理划分.c文件的一种破坏。

大部分产品修改一处代码，都得需要编译整个工程，对于TDD之类的实践，要求对于模块级别的编译时间控制在秒级，即使使用分布式编译也难以实现，最终仍然需要合理的划分头文件、以及头文件之间的包含关系，从根本上降低编译时间。

《google C++ Style Guide》1.2 头文件依赖 章节也给出了类似的阐述：

若包含了头文件aa.h，则就引入了新的依赖：一旦aa.h被修改，任何直接和间接包含aa.h代码都会被重新编译。如果aa.h又包含了其他头文件如bb.h，那么bb.h的任何改变都将导致所有包含了aa.h的代码被重新编译，在敏捷开发方式下，代码会被频繁构建，漫长的编译时间将极大的阻碍频繁构建。因此，我们倾向于减少包含头文件，尤其是在头文件中包含头文件，以控制改动代码后的编译时间。

合理的头文件划分体现了系统设计的思想，但是从编程规范的角度看，仍然有一些通用的方法，用来合理规划头文件。本章节介绍的一些方法，对于合理规划头文件会有一定的帮助。

**原则1.1 头文件中适合放置接口的声明，不适合放置实现。**

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（相当于类的私有方法）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。

延伸阅读材料：《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"）

**原则1.2 头文件应当职责单一。**

说明：头文件过于复杂，依赖过于复杂是导致编译时间过长的主要原因。很多现有代码中头文件过大，职责过多，再加上循环依赖的问题，可能导致为了在.c中使用一个宏，而包含十几个头文件。

示例：如下是某平台定义WORD类型的头文件：

#include <VXWORKS.H>

#include <KERNELLIB.H>

#include <SEMLIB.H>

#include <INTLIB.H>

#include <TASKLIB.H>

#include <MSGQLIB.H>

#include <STDARG.H>

#include <FIOLIB.H>

#include <STDIO.H>

#include <STDLIB.H>

#include <CTYPE.H>

#include <STRING.H>

#include <ERRNOLIB.H>

#include <TIMERS.H>

#include <MEMLIB.H>

#include <TIME.H>

#include <WDLIB.H>

#include <SYSLIB.H>

#include <TASKHOOKLIB.H>

#include <REBOOTLIB.H>

…

typedef unsigned short WORD;

…

这个头文件不但定义了基本数据类型WORD，还包含了stdio.h syslib.h等等不常用的头文件。如果工程中有10000个源文件，而其中100个源文件使用了stdio.h的printf，由于上述头文件的职责过于庞大，而WORD又是每一个文件必须包含的，从而导致stdio.h/syslib.h等可能被不必要的展开了9900次，大大增加了工程的编译时间。

**原则1.3 头文件应向稳定的方向包含。**

说明：头文件的包含关系是一种依赖，一般来说，应当让不稳定的模块依赖稳定的模块，从而当不稳定的模块发生变化时，不会影响（编译）稳定的模块。

就我们的产品来说，依赖的方向应该是：**产品依赖于平台，平台依赖于标准库。**某产品线平台的代码中已经包含了产品的头文件，导致平台无法单独编译、发布和测试，是一个非常糟糕的反例。

**除了不稳定的模块依赖于稳定的模块外，更好的方式是两个模块共同依赖于接口**，这样任何一个模块的内部实现更改都不需要重新编译另外一个模块。在这里，我们假设接口本身是最稳定的。

延伸阅读材料：编者推荐开发人员使用“**依赖倒置**”原则，即由使用者制定接口，服务提供者实现接口，更具体的描述可以参见《敏捷软件开发：原则、模式与实践》（Robert C.Martin 著 邓辉 译 清华大学出版社2003年9月） 的第二部分“敏捷设计”章节。

**规则1.1 每一个.c文件应有一个同名.h文件，用于声明需要对外公开的接口。将对内函数的声明放置.c文件的头部，并声明为static，以限制其作用域在该.c 文件定义的编译单元。**

说明：如果一个.c文件不需要对外公布任何接口，则其就不应当存在，除非它是程序的入口，如main函数所在的文件。

现有某些产品中，习惯一个.c文件对应两个头文件，一个用于存放对外公开的接口，一个用于存放内部需要用到的定义、声明等，以控制.c文件的代码行数。编者不提倡这种风格。这种风格的根源在于源文件过大，应首先考虑拆分.c文件，使之不至于太大。另外，一旦把私有定义、声明放到独立的头文件中，就无法从技术上避免别人include之，难以保证这些定义最后真的只是私有的。

本规则反过来并不一定成立。有些特别简单的头文件，如命令ID定义头文件，不需要有对应的.c存在。

示例：对于如下场景，如在一个.c中存在函数调用关系：

void foo()

{

bar();

}

void bar()

{

Do something;

}

必须在foo之前声明bar，否则会导致编译错误。

这一类的函数声明，应当在.c的头部声明，并声明为static的，如下：

static void bar();

void foo()

{

bar();

}

void bar()

{

Do something;

}

**规则1.2 禁止头文件循环依赖。**

说明：头文件循环依赖，指a.h包含b.h，b.h包含c.h，c.h包含a.h之类导致任何一个头文件修改，都导致所有包含了a.h/b.h/c.h的代码全部重新编译一遍。而如果是单向依赖，如a.h包含b.h，b.h包含c.h，而c.h不包含任何头文件，则修改a.h不会导致包含了b.h/c.h的源代码重新编译。

**规则1.3 .c/.h文件禁止包含用不到的头文件。**

说明：很多系统中头文件包含关系复杂，开发人员为了省事起见，可能不会去一一钻研，直接包含一切想到的头文件，甚至有些产品干脆发布了一个god.h，其中包含了所有头文件，然后发布给各个项目组使用，这种只图一时省事的做法，导致整个系统的编译时间进一步恶化，并对后来人的维护造成了巨大的麻烦。

**规则1.4 头文件应当自包含。**

说明：简单的说，自包含就是任意一个头文件均可独立编译。如果一个文件包含某个头文件，还要包含另外一个头文件才能工作的话，就会增加交流障碍，给这个头文件的用户增添不必要的负担。

示例：

如果a.h不是自包含的，需要包含b.h才能编译，会带来的危害：

每个使用a.h头文件的.c文件，为了让引入的a.h的内容编译通过，都要包含额外的头文件b.h。

额外的头文件b.h必须在a.h之前进行包含，这在包含顺序上产生了依赖。

注意：该规则需要与“.c/.h文件禁止包含用不到的头文件”规则一起使用，不能为了让a.h自包含，而在a.h中包含不必要的头文件。a.h要刚刚可以自包含，不能在a.h中多包含任何满足自包含之外的其他头文件。

**规则1.5 总是编写内部#include保护符（#define 保护）。**

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，命名格式为FILENAME\_H，为了保证唯一性，更好的命名是PROJECTNAME\_PATH\_FILENAME\_H。

注：没有在宏最前面加上单下划线"\_"，是因为一般以单下划线"\_"和双下划线"\_\_"开头的标识符为ANSI C等使用，在有些静态检查工具中，若全局可见的标识符以"\_"开头会给出告警。

定义包含保护符时，应该遵守如下规则：

1）保护符使用唯一名称；

2）不要在受保护部分的前后放置代码或者注释。

示例：假定VOS工程的timer模块的timer.h，其目录为VOS/include/timer/timer.h,应按如下方式保护：

#ifndef VOS\_INCLUDE\_TIMER\_TIMER\_H

#define VOS\_INCLUDE\_TIMER\_TIMER\_H

...

#endif

也可以使用如下简单方式保护:

#ifndef TIMER\_H

#define TIMER\_H

..

#endif

例外情况：头文件的版权声明部分以及头文件的整体注释部分（如阐述此头文件的开发背景、使用注意事项等）可以放在保护符(#ifndef XX\_H)前面。

**规则1.6 禁止在头文件中定义变量。**

说明：在头文件中定义变量，将会由于头文件被其他.c文件包含而导致变量重复定义。

**规则1.7 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extern int foo(int input)；并在a.c中通过#include <b.h>来使用foo。禁止通过在a.c中直接写extern int 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1.8 禁止在extern "C"中包含头文件。**

说明：在extern "C"中包含头文件，会导致extern "C"嵌套，Visual Studio对extern "C"嵌套层次有限制，嵌套层次太多会编译错误。

在extern "C"中包含头文件，可能会导致被包含头文件的原有意图遭到破坏。例如，存在a.h和b.h两个头文件：

|  |  |
| --- | --- |
| #ifndef A\_H\_\_  #define A\_H\_\_  #ifdef \_\_cplusplus  void foo(int);  #define a(value) foo(value)  #else  void a(int)  #endif  #endif /\* A\_H\_\_ \*/ | #ifndef B\_H\_\_  #define B\_H\_\_  #ifdef \_\_cplusplus  extern "C" {  #endif  #include "a.h"  void b();  #ifdef \_\_cplusplus  }  #endif  #endif /\* B\_H\_\_ \*/ |

使用C++预处理器展开b.h，将会得到

extern "C" {

void foo(int);

void b();

}

按照a.h作者的本意，函数foo是一个C++自由函数，其链接规范为"C++"。但在b.h中，由于#include "a.h"被放到了extern "C" { }的内部，函数foo的链接规范被不正确地更改了。

示例：错误的使用方式：

extern “C”

{

#include “xxx.h”

...

}

正确的使用方式：

#include “xxx.h”

extern “C”

{

...

}

**规则1.9 避免文件过长，新增文件不超过2000行。**

说明：本规则仅对新增文件做要求，对已有文件修改时，总行数超过2000行，建议不增加代码行。

过长的文件往往意味着文件（模块）功能不单一，过于复杂。

一个文件的行数（包括空行和注释行）应当小于2000。

**建议1.1 一个模块通常包含多个.c文件，建议放在同一个目录下，目录名即为模块名。为方便外部使用者，建议每一个模块提供一个.h，文件名为目录名。**

说明：需要注意的是，这个.h并不是简单的包含所有内部的.h，它是为了模块使用者的方便，对外整体提供的模块接口。

以Google test（简称GTest）为例，GTest作为一个整体对外提供C++单元测试框架，其1.5版本的gtest工程下有6个源文件和12个头文件。但是它对外只提供一个gtest.h，只要包含gtest.h即可使用GTest提供的所有对外提供的功能，使用者不必关系GTest内部各个文件的关系，即使以后GTest的内部实现改变了，比如把一个源文件c拆成两个源文件，使用者也不必关心，甚至如果对外功能不变，连重新编译都不需要。

对于有些模块，其内部功能相对松散，可能并不一定需要提供这个.h，而是直接提供各个子模块或者.c的头文件。

比如产品普遍使用的VOS，作为一个大模块，其内部有很多子模块，他们之间的关系相对比较松散，就不适合提供一个vos.h。而VOS的子模块，如Memory（仅作举例说明，与实际情况可能有所出入），其内部实现高度内聚，虽然其内部实现可能有多个.c和.h，但是对外只需要提供一个Memory.h声明接口。

**建议1.2 如果一个模块包含多个子模块，则建议每一个子模块提供一个对外的.h，文件名为子模块名。**

说明：降低接口使用者的编写难度。

**建议1.3 头文件不要使用非习惯用法的扩展名，如.inc。**

说明：目前很多产品中使用了.inc作为头文件扩展名，这不符合c语言的习惯用法。在使用.inc作为头文件扩展名的产品，习惯上用于标识此头文件为私有头文件。但是从产品的实际代码来看，这一条并没有被遵守，一个.inc文件被多个.c包含比比皆是。本规范不提倡将私有定义单独放在头文件中，具体见规则1.1。

除此之外，使用.inc还导致source insight、Visual stduio等IDE工具无法识别其为头文件，导致很多功能不可用，如“跳转到变量定义处”。虽然可以通过配置，强迫IDE识别.inc为头文件，但是有些软件无法配置，如Visual Assist只能识别.h而无法通过配置识别.inc。

**建议1.4 同一产品统一包含头文件排列方式。**

说明：常见的包含头文件排列方式：功能块排序、文件名升序、稳定度排序。

示例1：

以升序方式排列头文件可以避免头文件被重复包含，如：

#include <a.h>

#include <b.h>

#include <c/d.h>

#include <c/e.h>

#include <f.h>

示例2：

以稳定度排序，建议将不稳定的头文件放在前面，如把产品的头文件放在平台的头文件前面，如下：

#include <product.h>

#include <platform.h>

相对来说，product.h修改的较为频繁，如果有错误，不必编译platform.h就可以发现product.h的错误，可以部分减少编译时间。

# 函数

背景

**函数设计的精髓：编写整洁函数，同时把代码有效组织起来。**

整洁函数要求：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。

**原则2.1 一个函数仅完成一件功能。**

说明：一个函数实现多个功能给开发、使用、维护都带来很大的困难。

将没有关联或者关联很弱的语句放到同一函数中，会导致函数职责不明确，难以理解，难以测试和改动。

案例：realloc。在标准C语言中，realloc是一个典型的不良设计。这个函数基本功能是重新分配内存，但它承担了太多的其他任务：如果传入的指针参数为NULL就分配内存，如果传入的大小参数为0就释放内存，如果可行则就地重新分配，如果不行则移到其他地方分配。如果没有足够可用的内存用来完成重新分配（扩大原来的内存块或者分配新的内存块），则返回NULL，而原来的内存块保持不变。这个函数不易扩展，容易导致问题。例如下面代码容易导致内存泄漏：

char \*buffer = (char \*)malloc(XXX\_SIZE);

.....

buffer = (char \*)realloc(buffer, NEW\_SIZE);

如果没有足够可用的内存用来完成重新分配，函数返回为NULL，导致buffer原来指向的内存被丢失。

延伸阅读材料：《敏捷软件开发：原则、模式与实践》 第八章，单一职责原则(SRP)

**原则2.2 重复代码应该尽可能提炼成函数。**

说明：重复代码提炼成函数可以带来维护成本的降低。

重复代码是我司不良代码最典型的特征之一。在“代码能用就不改”的指导原则之下，大量的烟囱式设计及其实现充斥着各产品代码之中。新需求增加带来的代码拷贝和修改，随着时间的迁移，产品中堆砌着许多类似或者重复的代码。

项目组应当使用代码重复度检查工具，在持续集成环境中持续检查代码重复度指标变化趋势，并对新增重复代码及时重构。当一段代码重复两次时，即应考虑消除重复，当代码重复超过三次时，应当立刻着手消除重复。

一般情况下，可以通过提炼函数的形式消除重复代码。

示例：

UC ccb\_aoc\_process( )

{

... ...

struct AOC\_E1\_E7 aoc\_e1\_e7;

aoc\_e1\_e7.aoc = 0;

aoc\_e1\_e7.e[0] = 0;

... ... //aoc\_e1\_e7.e[i]从到赋值，下同

aoc\_e1\_e7.e[6] = 0;

aoc\_e1\_e7.tariff\_rate = 0;

... ...

**if (xxx)**

**{**

**if (xxx)**

**{**

**aoc\_e1\_e7.e[0] = 0;**

**... ...**

**aoc\_e1\_e7.e[6] = 0;**

**aoc\_e1\_e7.tariff\_rate = 0;**

**}**

**... ...**

**}**

**else if (xxx)**

**{**

**if (xxx)**

**{**

**aoc\_e1\_e7.e[0] = 0;**

**... ...**

**aoc\_e1\_e7.e[6] = 0;**

**aoc\_e1\_e7.tariff\_rate = 0;**

**}**

ccb\_caller\_e1 = aoc\_e1\_e7.e[0];

... ...

ccb\_caller\_e7 = aoc\_e1\_e7.e[6];

ccb\_caller\_tariff\_rate = aoc\_e1\_e7.tariff\_rate;

... ...

**}**

... ...

**if (xxx)**

**{**

**if (xxx)**

**{**

**if (xxx)**

**{**

**aoc\_e1\_e7.e[0] = 0;**

**... ...**

**aoc\_e1\_e7.e[6] = 0;**

**aoc\_e1\_e7.tariff\_rate = 0;**

**}**

**... ...**

**}**

**else if (xxx)**

**{**

**if (xxx)**

**{**

**aoc\_e1\_e7.e[0] = 0;**

**... ...**

**aoc\_e1\_e7.e[6] = 0;**

**aoc\_e1\_e7.tariff\_rate = 0;**

**}**

ccb\_caller\_e1 = aoc\_e1\_e7.e[0];

... ...

ccb\_caller\_e7 = aoc\_e1\_e7.e[6];

ccb\_caller\_tariff\_rate = aoc\_e1\_e7.tariff\_rate;

... ...

**}**

return 1;

}

else

{

return 0;

}

}

刺鼻的代码坏味充斥着这个函数。红色字体的部分是简单的代码重复，粗体字部分是代码结构的重复，将重复部分提炼成一个函数即可消除重复。

**规则2.1 避免函数过长，新增函数不超过50行（非空非注释行）。**

说明：本规则仅对新增函数做要求，对已有函数修改时，建议不增加代码行。

过长的函数往往意味着函数功能不单一，过于复杂（参见原则2.1：一个函数只完成一个功能）。

函数的有效代码行数，即NBNC（非空非注释行）应当在[1，50]区间。

例外：某些实现算法的函数，由于算法的聚合性与功能的全面性，可能会超过50行。

延伸阅读材料：业界普遍认为一个函数的代码行不要超过一个屏幕，避免来回翻页影响阅读；一般的代码度量工具建议都对此进行检查，例如Logiscope的函数度量："Number of Statement" （函数中的可执行语句数）建议不超过20行，QA C建议一个函数中的所有行数（包括注释和空白行）不超过50行。

**规则2.2 避免函数的代码块嵌套过深，新增函数的代码块嵌套不超过4层。**

说明：本规则仅对新增函数做要求，对已有的代码建议不增加嵌套层次。

函数的代码块嵌套深度指的是函数中的代码控制块（例如：if、for、while、switch等）之间互相包含的深度。每级嵌套都会增加阅读代码时的脑力消耗，因为需要在脑子里维护一个“栈”（比如，进入条件语句、进入循环……）。应该做进一步的功能分解，从而避免使代码的阅读者一次记住太多的上下文。优秀代码参考值：[1, 4]。

示例：如下代码嵌套深度为5。

void serial (void)

{

if (!Received)

{

TmoCount = 0;

switch (Buff)

{

case AISGFLG:

if ((TiBuff.Count > 3)

&& ((TiBuff.Buff[0] == 0xff) || (TiBuf.Buff[0] == CurPa.ADDR)))

{

Flg7E = false;

Received = true;

}

else

{

TiBuff.Count = 0;

Flg7D = false;

Flg7E = true;

}

break;

default:

break;

}

}

}

**规则2.3 可重入函数应避免使用共享变量；若需要使用，则应通过互斥手段（关中断、信号量）对其加以保护。**

说明：可重入函数是指可能被多个任务并发调用的函数。在多任务操作系统中，函数具有可重入性是多个任务可以共用此函数的必要条件。共享变量指的全局变量和static变量。

编写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。

示例：函数square\_exam返回g\_exam平方值。那么如下函数不具有可重入性。

int g\_exam;

unsigned int example( int para )

{

unsigned int temp;

g\_exam = para; // （\*\*）

temp = square\_exam ( );

return temp;

}

此函数若被多个线程调用的话，其结果可能是未知的，因为当（\*\*）语句刚执行完后，另外一个使用本函数的线程可能正好被激活，那么当新激活的线程执行到此函数时，将使g\_exam赋于另一个不同的para值，所以当控制重新回到“temp =square\_exam ( )”后，计算出的temp很可能不是预想中的结果。此函数应如下改进。

int g\_exam;

unsigned int example( int para )

{

unsigned int temp;

[申请信号量操作] // 若申请不到“信号量”，说明另外的进程正处于

g\_exam = para; //给g\_exam赋值并计算其平方过程中（即正在使用此

temp = square\_exam( ); // 信号），本进程必须等待其释放信号后，才可继

[释放信号量操作] // 续执行。其它线程必须等待本线程释放信号量后

// 才能再使用本信号。

return temp;

}

**规则2.4** **对参数的合法性检查，由调用者负责还是由接口函数负责，应在项目组/模块内应统一规定。缺省由调用者负责。**

说明：对于模块间接口函数的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

示例：下面红色部分的代码在每一个函数中都写了一次，导致代码有较多的冗余。如果函数的参数比较多，而且判断的条件比较复杂（比如：一个整形数字需要判断范围等），那么冗余的代码会大面积充斥着业务代码。

void PidMsgProc(MsgBlock \*Msg)

{

MsgProcItem \*func = NULL;

**if (Msg == NULL)**

**{**

**return;**

**}**

... ...

GetMsgProcFun(Msg, &func);

func(Msg);

return;

}

int GetMsgProcFun(MsgBlock \*Msg, MsgProcItem \*\*func)

{

**if (Msg == NULL)**

**{**

**return 1;**

**}**

... ...

\*func = VOS\_NULL\_PTR;

for (Index = 0; Index < NELEM(g\_MsgProcTable); Index++)

{

if ((g\_MsgProcTable[Index].FlowType == Msg->FlowType)

&& (g\_MsgProcTable[Index].Status == Msg->Status)

&& (g\_MsgProcTable[Index].MsgType == Msg->MsgType))

{

\*func = &(g\_MsgProcTable[Index]);

return 0;

}

}

return 1;

}

int ServiceProcess(int CbNo, MsgBlock \*Msg)

{

**if ( Msg == NULL)**

**{**

**return 1;**

**}**

... ...

// 业务处理代码

... ...

return 0;

}

**规则2.5 对函数的错误返回码要全面处理。**

说明：一个函数（标准库中的函数/第三方库函数/用户定义的函数）能够提供一些指示错误发生的方法。这可以通过使用错误标记、特殊的返回数据或者其他手段，不管什么时候函数提供了这样的机制，调用程序应该在函数返回时立刻检查错误指示。

示例：下面的代码导致宕机

FILE \*fp = fopen( "./writeAlarmLastTime.log","r");

if(fp == NULL)

{

return;

}

char buff[128] = "";

fscanf(fp,“%s”, buff); /\* 读取最新的告警时间；由于文件writeAlarmLastTime.log为空，导致buff内容为空 \*/

fclose(fp);

long fileTime = getAlarmTime(buff); /\* 解析获取最新的告警时间；getAlarmTime函数未检查buff内容，导致宕机 \*/

正确写法：

FILE \*fp = fopen( "./writeAlarmLastTime.log","r");

if(fp == NULL)

{

return;

}

char buff[128] = "";

if (fscanf(fp,“%s”,buff) == EOF) //检查函数fscanf的返回值，确保读到数据

{

fclose(fp);

return;

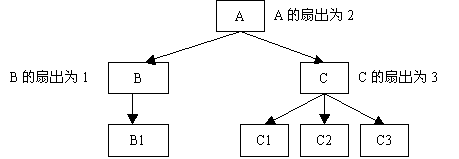
}

fclose(fp);

long fileTime = getAlarmTime(buff); //解析获取最新的告警时间；

**规则2.6 设计高扇入，合理扇出（小于7）的函数和模块。**

说明：对于函数而言，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，例如：总是1，表明函数的调用层次可能过多，这样不利于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。通常函数比较合理的扇出（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。

对于模块（文件）而言，一条#include语句数量就是需要调用一个其他模块（文件）。

延伸阅读材料：扇入（Fan-in）和扇出（Fan-out）是Henry和Kafura在1981年引入，用来说明模块间的耦合（coupling），后面人们扩展到函数/方法、模块/类、包等。

The Fan-in (Informational fan-in) metric measures the fan-in of a module. The fan-in of a module A is the number of modules that pass control into module A.

The Fan-out metric measures the number of the number of modules that are called by a given module.

**规则2.7 废弃代码（没有被调用的函数和变量)要及时清除。**

说明：程序中的废弃代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。**规则2.8 使用强类型参数，避免使用void\***

说明：尽管不同的语言对待强类型和弱类型有自己的观点，但是一般认为c/c++是强类型语言，既然我们使用的语言是强类型的，就应该保持这样的风格。

使用强类型便于编译器帮我们发现错误，如下代码使用了void \*传递参数：

void foo( void \*pThis)

{

LOCAL\_DATA\_S \*pstLocalData =( LOCAL\_DATA\_S \*) pThis;

SERVICE\_NODE\_S \*pstNode = (SERVICE\_NODE\_S \*)MALLOC(sizeof(SERVICE\_NODE\_S));

pstNode->stLink.next = pstLocalData->stServiceTable.next;

pstNode->stLink.prev = &(pstLocalData->stServiceTable);

pstLocalData->stServiceTable.next = &(pstNode->stLink);

pstNode->stLink.next->prev = &(pstNode->stLink);

ServiceTable\_AddNode(&(pstLocalData->stServiceTable), pstNode); //正确的

ServiceTable\_AddNode(&(pstLocalData->stAnotherServiceTable), pstNode); //误写成stAnotherServiceTable

}

函数ServiceTable\_AddNode的入参是void\*,问题直到网上才发现：

void ServiceTable\_AddNode(void \*pstServiceTable, SERVICE\_NODE\_S \*pstNewNode)

{

VOS\_ListAdd(&(pstNewNode->stLink), &(pstServiceTable->stHead));

}

如果明确ServiceTable\_AddNode的入参是SERVICE\_TABLE\_S\*，在编译阶段就能发现上述错误。

void ServiceTable\_AddNode(SERVICE\_TABLE\_S \*pstServiceTable, SERVICE\_NODE\_S \*pstNewNode)

{

VOS\_ListAdd(&(pstNewNode->stLink), &(pstServiceTable->stHead));

}

除非是通用库，不得不传入各种类型指针的，否则不要用void\*作为入参，让编译器在编译阶段就检查出对象类型不匹配的问题

**规则2.9 使用模块内的函数，确保参数类型一致，避免类型转换；使用其他模块的接口函数，不一致的参数类型，建议统一在接口处进行类型转换。**

说明：使用模块内的函数，如果参数类型不一致，往往是使用方式不对或者设计不好。函数的参数类型转换，存在潜在的危险，可能是不安全的，例如：

* 数值的丢失：转化后的类型其数值量级不能被体现
* 符号的丢失：从有符号类型转换为无符号类型会导致符号的丢失
* 精度的丢失：从浮点类型转换为整型会导致精度的丢失

**建议2.1** **函数不变参数使用const。**

说明：不变的值更易于理解/跟踪和分析，把const作为默认选项，在编译时会对其进行检查，使代码更牢固/更安全。

示例：C99标准 7.21.4.4 中strncmp 的例子，不变参数声明为const。

int strncmp(const char \*s1, const char \*s2, register size\_t n)

{

register unsigned char u1, u2;

while (n-- > 0)

{

u1 = (unsigned char) \*s1++;

u2 = (unsigned char) \*s2++;

if (u1 != u2)

{

return u1 - u2;

}

if (u1 == '\0')

{

return 0;

}

}

return 0;

}

延伸阅读：pc-lint 8.0的帮助材料（pc-lint.pdf）11.4 const Checking

**建议2.2 函数应避免使用全局变量、静态局部变量和I/O操作，不可避免的地方应集中使用。**

说明：带有内部“存储器”的函数的功能可能是不可预测的，因为它的输出可能取决于内部存储器（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。在C语言中，函数的static局部变量是函数的内部存储器，有可能使函数的功能不可预测。

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;

}

延伸阅读材料：erlang语言中关于dirty的概念，函数式语言的优势

**建议2.3 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。**

说明：函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入参数之前，应进行有效性检查。

示例：下面的代码导致宕机

hr = root\_node->get\_first\_child(&log\_item); // list.xml 为空，导致读出log\_item为空

…..

hr = log\_item->get\_next\_sibling(&media\_next\_node); // log\_item为空，导致宕机

正确写法：确保读出的内容非空。

hr = root\_node->get\_first\_child(&log\_item);

…..

if (log\_item == NULL) //确保读出的内容非空

{

return retValue;

}

hr = log\_item->get\_next\_sibling(&media\_next\_node);

**建议2.4 函数的参数个数不超过5个。**

说明：函数的参数过多，会使得该函数易于受外部（其他部分的代码）变化的影响，从而影响维护工作。函数的参数过多同时也会增大测试的工作量。

函数的参数个数不要超过5个，如果超过了建议拆分为不同函数。

**建议2.5 除打印类函数外，不要使用可变长参函数。**

说明：可变长参函数的处理过程比较复杂容易引入错误，而且性能也比较低，使用过多的可变长参函数将导致函数的维护难度大大增加。

**建议2.6 在源文件范围内声明和定义的所有函数，除非外部可见，否则应该增加static关键字。**

说明：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声明。使用static确保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。

建议定义一个STATIC宏，在调试阶段，将STATIC定义为static，版本发布时，改为空，以便于后续的打热补丁等操作。

#ifdef \_DEBUG

#define STATIC static

#else

#define STATIC

#endif

**建议2.7 定义函数时，参数顺序为：输入参数在前，输出参数在后。**

说明：将参数排序，并首先将输入参数分组，再将输出参数放置最后。在参数组内，按照能够帮助程序员输入正确值的原则来将参数排序。比如，如果一个函数带有2个参数， “left” 和 “right” ，将 “left” 置于 “right” 之前，则它们的放置顺序符合其参数名。当设计一系列具有相同参数的函数时，在各函数内使用一致的顺序。

**建议2.8 函数使用前必须显式声明。**

说明：即使函数只在一个源文件里面使用，也必须在源文件开始处显式声明。

**建议2.9 不能有不可到达的代码。**

说明：针对那些在任何环境中都不能到达的代码，这些代码在编译时就能被标识出不可到达。建议排除了可以到达，但永远不会执行的代码，如保护性编程代码。

如果从相关的入口到某部分代码之间不存在控制流路径，那么这部分代码就是不可到达的。例如，在无条件控制转移代码后的未标记代码，就是不可到达的。

示例：不好的例子

switch (cmd\_flag)

{

case case CMD\_A:

ProcessCMD();

break;

do\_more(); /\* 不可达代码 \*/

/\* … \*/

default:

/\* … \*/

break;

}

**建议2.10 避免函数提前返回。**

说明： 理想状态下，所有函数都应该只在函数底部有一个返回点，所有的执行路径都会通过该点返回。

例外：函数开头的参数验证，可以提前返回。