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。