

文件名：C 语言编码风格	文件编号：2006031201
版本号：0.3	密级：机密
产品名称：软件工程规范	部门：软件开发部
创建者：Leo, Aaron, Vik	创建时间：2005-08-08
批准者：Leo, Aaron, Vik, Bob, Lilian	批准时间：2006-03-13

# 软件工程规范

**C**语言编码风格

*October 14, 2006*

***Juphcon***

宁波市菊风系统软件有限公司

版权所有 侵权必究

## 修订历史

版本	主要作者	修订描述	完成时间
0.1	Aaron Zhong Leo Lv Vik Qian	C 语言编码风格(草稿)	2005-08-08
0.2	Lilian Huang	主要修改了一些拼写错误	2005-09-27
0.2	Leo Lv	1. 修改了一些条款的例子 2. 增加和修改了条款说明 3. 删除了不必要的条款	2006-03-12
0.2	Vik Qian	修改了条款说明	2006-03-12
0.2	Lilian Huang	修改了条款说明	2006-03-12
0.3	Leo Lv	1. 修改了一些条款的例子 2. 增加和修改了条款说明 3. 增加了常用文件组织和命名附录 4. 增加了 IEEE Software Quality Glossary 附录	2006-03-13
0.3	Vik Qian	修改了条款说明	2006-03-13
0.3	Lilian Huang	修改了条款说明	2006-03-13

## 摘要

本文是菊风协议软件产品开发的C语言编码风格指导，对C语言中的主要特性进行细致的叙述，同时也对编码过程中应该遵守和避免的风格进行了规定。

## 目录内容

<b>1. 前言</b> .....	<b>9</b>
1.1 读者对象.....	9
1.2 介绍.....	9
<b>2. 命名惯例</b> .....	<b>11</b>
2.1 文件命名.....	11
2.1.1 条款1: 命名字符.....	11
2.1.2 条款2: 命名格式.....	11
2.1.3 条款3: 文件后缀.....	11
2.1.4 条款4: 测试文件命名.....	11
2.1.5 条款5: 样例文件命名.....	12
2.2 变量命名.....	12
2.2.1 条款1: 基本规则.....	12
2.2.2 条款2: 使用前缀命名法.....	12
2.2.3 条款3: 基本类型命名.....	12
2.2.4 条款4: 平台基本类型.....	13
2.2.5 条款5: 结构类型.....	13
2.2.6 条款6: 枚举类型.....	14
2.2.7 条款7: 位类型.....	14
2.2.8 条款8: 变量名字书写.....	14
2.2.9 条款9: 除循环变量外, 不建议使用i, j, k等无意义的单字母作为变量名.....	14
2.2.10 条款10: 在定义多个相关名称的时候, 注意名称意义上的对称.....	14
2.2.11 条款11: 特殊的常量.....	15
2.3 常量命名.....	15
2.3.1 条款1: 常量全用大写的字母, 用下划线分割单词.....	15
2.3.2 条款2: ZOS常量命名约定.....	15
2.4 函数命名.....	16
2.4.1 条款1: 基本规则.....	16
2.4.2 条款2: 函数名与其返回值在语义上不能冲突.....	16
<b>3. 代码注释</b> .....	<b>17</b>
3.1 条款1: 注释风格.....	17
3.2 条款2: 注释语言只用英文.....	17
3.3 条款3: 代码行首注释.....	17
3.4 条款4: 代码行尾注释.....	18

3.5	条款5: 建议在预编译的结束语句右方(一个空格后)加注释标记, 以表明一段程序块的结束.....	19
3.6	条款6: 函数注释.....	19
3.7	条款7: 函数中未实现的代码, 应该添加TODO注释.....	20
3.8	条款8: 用#IF 0 #ENDIF屏蔽代码块.....	20
<b>4.</b>	<b>代码布局.....</b>	<b>21</b>
4.1	空格.....	21
4.1.1	条款1: 基本条款.....	21
4.1.2	条款2: 程序块.....	22
4.1.3	条款3: 缩进.....	22
4.2	代码行.....	23
4.2.1	条款1: 行最大长度为80.....	23
4.2.2	条款2: 语句长度大于80.....	23
4.2.3	条款3: if、for、do、while、case、switch、default等语句独占一行, 建议在if、for、do、while等语句的执行语句部分加上括号{ }.....	24
4.2.4	条款4: 程序块的分界符(‘{’和‘}’)应各独占一行并且位于同一列.....	24
4.2.5	条款5: 相对独立的程序块之间、变量说明之后必须加空行 在函数中的局部变量是在所有变量都说明后加空行。.....	24
4.2.6	条款6: 不建议把多个短语句写在一行中, 即一行只写一条语句.....	25
4.3	头文件.....	25
4.3.1	条款1: 防止头文件重复被引用.....	25
4.3.2	条款2: 让C++编译器使用C的命名方式.....	25
4.3.3	条款3: 文件包含.....	26
4.3.4	条款4: 建议头文件中只存放“声明”而不存放“定义”.....	26
4.3.5	条款5: 头文件布局.....	27
4.3.6	条款6: 头文件行数不要超过1万行.....	27
4.3.7	条款7: 自定义头文件避免使用和系统头文件一样的名字.....	27
4.3.8	条款8: 不建议使用包含绝对路径的用法.....	27
4.3.9	条款9: 不建议使用反斜杠“\”表示路径.....	28
4.4	源文件.....	28
4.4.1	条款1: 源文件布局.....	28
4.4.2	条款2: 源文件行数不要超过3万行.....	28
4.4.3	条款3: 建议不要引用不必要的头文件.....	28
<b>5.</b>	<b>类型, 操作符和表达式.....</b>	<b>30</b>
5.1	数据类型和大小.....	30
5.1.1	条款1: 要始终明确数据类型及其最大范围.....	30
5.2	常量.....	30
5.2.1	条款1: 常量定义.....	30
5.2.2	条款2: 不允许直接出现数字, 必须用有意义的枚举或宏(常量)来代替.....	31
5.2.3	条款3: 常量的位置.....	32
5.3	变量.....	32
5.3.1	条款1: 变量定义.....	32
5.3.2	条款2: 加上const限定词的声明.....	33

5.3.3	条款3: 变量定义顺序.....	33
5.3.4	条款4: 变量定义必须放在函数的入口处, 禁止C++风格的即用即定义(在程序中间进行变量定义).....	33
5.3.5	条款5: 防止局部变量与全局变量同名.....	34
5.4	算术操作符.....	34
5.4.1	条款1: 注意运算符“/”、“%”.....	34
5.5	关系和逻辑操作符.....	34
5.5.1	条款1: 布尔变量与零值比较.....	34
5.5.2	条款2: 浮点变量与零值比较.....	35
5.5.3	条款3: 指针变量与零值比较.....	35
5.5.4	条款4: 注意赋值操作和比较操作的区别.....	35
5.5.5	条款5: 注意数学表达式和逻辑表达式的区别.....	35
5.6	类型转换.....	36
5.6.1	条款1: 注意从char到int的类型转换符号问题.....	36
5.6.2	条款2: 避免有符号和无符号类型之间直接的运算.....	36
5.6.3	条款3: 尽量减少没有必要的数据类型默认转换与强制转换.....	37
5.6.4	条款4: 关注类型转换的代价.....	37
5.7	自增和自减操作符.....	37
5.7.1	条款1: 注意自增和自减操作符的前缀和后缀的区别.....	37
5.8	位操作符.....	37
5.8.1	条款1: 基本条款.....	37
5.8.2	条款2: 关注位移运算符对于有符号整数进行的右移 >> 运算.....	38
5.9	赋值操作符和表达式.....	38
5.9.1	条款1: 推荐使用复合二元操作符的赋值运算, 如 +=.....	38
5.9.2	条款2: 注意赋值操作的返回.....	38
5.10	条件表达式.....	39
5.10.1	条款1: 使用条件表达式简化 if... else.....	39
5.11	优先级和赋值次序.....	39
5.11.1	条款1: 大体上明确操作符的优先级, 或者使用括号 ( ).....	39
5.11.2	条款2: 避免“side effects”语句的不良应用.....	40
<b>6.</b>	<b>控制流.....</b>	<b>41</b>
6.1	IF-ELSE语句.....	41
6.1.1	条款1: 尽量避免在if的判断条件中使用赋值语句.....	41
6.2	SWITCH语句.....	41
6.2.1	条款1: swich语句必须有default语句, 每个case语句建议有break语句.....	41
6.2.2	条款2: 避免相邻的case语句共用语句, 如果一定要共用必须给出明确的注释加以说明 41	
6.2.3	条款3: 在最后一个case中(包括default)加入break, 尽管在逻辑上不需要.....	42
6.3	WHILE和 FOR 循环.....	43
6.3.1	条款1: 建议for语句循环变量的取值采用半开半闭区间.....	43
6.3.2	条款2: 谨慎使用for语句中的逗号.....	43
6.4	GOTO语句和标签.....	43
6.4.1	条款1: 尽量少用和慎用goto语句.....	44

6.4.2	条款2: 如果不得不使用goto, 标签应该单独成行, 并且比下面的代码向左缩进四格	44
<b>7.</b>	<b>函数和程序结构</b>	<b>45</b>
7.1	函数基础	45
7.1.1	条款1: 函数参数	45
7.1.2	条款2: 函数传参时, 禁止直接使用结构进行传参, 而应该使用结构指针	45
7.1.3	条款3: 函数如果没有参数, 建议使用ZVOID填充	45
7.1.4	条款4: 使用ANSI C99定义函数	45
7.1.5	条款5: 尽量少定义具有可变参数的函数	46
7.1.6	条款6: 建议提供函数声明	46
7.1.7	条款7: 函数不能返回指向函数堆栈的指针	46
7.1.8	条款8: 少用ZBOOL类型作为参数和返回值	46
7.1.9	条款9: 对于提供了返回值的函数, 要进行返回值判断, 以保证返回的各种情况都可以得到正确的处理	46
7.1.10	条款10: 如果函数返回值需要详细解释, 在函数说明中予以描述	47
7.1.11	条款11: 函数参数的有效性判断	47
7.1.12	条款12: 不打算被其他文件使用的函数要声明为static	47
7.1.13	条款13: 函数不能返回结构体	47
7.2	全局变量	47
7.2.1	条款1: 尽量减少在模块间直接使用全局变量, 对模块间的数据访问最好提供函数接口	47
7.2.2	条款2: 编写可重入函数时, 若使用全局变量, 可通过互斥量、信号量(即P、V操作)等手段对其加以保护	47
7.3	程序块结构	48
7.3.1	条款1: 程序块内变量的取名避免和程序块外的变量一样	48
7.4	递归	48
7.4.1	条款1: 尽量减少函数本身或函数间的递归调用	48
7.5	预处理器	48
7.5.1	宏	48
7.5.2	条件预编译	51
<b>8.</b>	<b>指针和数组</b>	<b>53</b>
8.1	指针和内存使用	53
8.1.1	条款1: 要进行边界检查, 防止内存操作越界	53
8.1.2	条款2: 严禁使用未经初始化的变量作为右值	53
8.1.3	条款3: 任何一次调用申请内存后要检查是否申请成功	53
8.1.4	条款4: 管理好你的内存	53
8.1.5	条款5: 建议谨慎使用memset、memcpy等大量存在内存拷贝的函数, 如果大量使用会显著降低程序效率	56
8.1.6	条款6: 不推荐过多地使用指针	56
8.1.7	条款7: 不推荐在模块中频繁的使用动态内存分配释放操作	56
8.1.8	条款8: 避免使用已经释放的内存	56
8.2	指针和函数参数	56
8.2.1	条款1: 当函数参数为数组时, 建议写成同类型指针变量的形式	56

8.3	指针和数组.....	57
8.3.1	条款1: 不要试图修改常量字符串.....	57
8.4	地址运算.....	57
8.4.1	条款1: 注意地址运算的有效性.....	57
<b>9.</b>	<b>结构体类型.....</b>	<b>58</b>
9.1	结构体基础.....	58
9.1.1	条款1: 结构体声明.....	58
9.1.2	条款2: 结构体成员要4字节对齐.....	58
9.2	结构体数组.....	59
9.2.1	条款1: 结构体数组的初始化.....	59
9.2.2	条款2: 结构体数组大小的计算.....	59
9.3	TYPEDEF 类型定义.....	59
9.3.1	条款1: 建议使用typedef定义变量类型.....	59
9.4	位域.....	60
9.4.1	条款1: 不要使用位域进行位操作, 建议使用位逻辑运算符.....	60
<b>10.</b>	<b>库函数.....</b>	<b>61</b>
10.1	条款1: 不要在程序中声明系统函数.....	61
10.2	条款2: 不要使用SPRINTF的返回值.....	61
10.3	条款3: GETCHAR返回一个整数, 而不是一个字符.....	61
10.4	条款4: 小心使用PRINTF、SPRINTF、FPRINTF.....	61
<b>11.</b>	<b>编程惯例.....</b>	<b>63</b>
11.1	条款1: 建议函数长度不要超过200行, 否则可能会增加阅读的复杂性, 而且也容易出错	63
11.2	条款2: 建议避免过深(不要超过8重)的嵌套代码.....	63
11.3	条款3: 在定义多个相关名称的时候, 注意名称意义上的对称.....	63
11.4	条款4: 建议要了解CPU类型方面的差异性, 避免不正确的假设.....	64
11.5	条款5: 所有代码要符合ANSI C99标准.....	64
11.6	条款6: 尽量少定义占用太大空间的局部变量, 同时要确认是否会造成堆栈溢出.....	64
11.7	条款7: 小心使用SIZEOF来得到数据类型或空间的大小.....	64
11.8	条款8: 养成良好的字符串使用习惯.....	65
11.9	条款9: 对大块内存的初始化代码, 使用循环展开的方式进行优化.....	65
11.10	条款10: 注意比较同一函数在不同编译环境和系统运行环境的效率差异.....	65
11.11	条款11: 程序性能优化的时机.....	66
11.12	条款12: 用FOR (;)代替WHILE (1).....	66
11.13	条款13: 系统设计应该保证概念完整性.....	66
11.14	条款14: 设计者应该对系统体系结构有比较清晰地感觉, 能清楚的以语言、文字、图形等方式向团队表达, 从中发现抽象模型和通用机制, 目的是使得系统更加简单、轻巧和可靠	66
11.15	条款15: 设计的一个重要原则是简单。简单的系统设计易于理解和维护, 简单设计是建立在艰苦的时间思考和不断的反复修改后达到的, 简单并不意味着草率和粗糙.....	66
11.16	条款16: 设计时应该注意扩展性, 良好的扩展性往往建立在清晰的系统设计和通用的抽象设计上。良好的扩展性能给系统带来更好的适应能力.....	66

11.17	条款17: 理解机制与策略的不同.....	66
11.18	条款18: 使用相同的编辑器, 并使用相同的选项设置.....	66
11.19	条款19: 在软件产品(项目组)中, 要统一编译开关选项.....	67
11.20	条款20: 建议首先在平台和标准库中查找符合所需功能的函数, 避免随意添加冗余功能的函数.....	67
11.21	条款21: 不要引用UNIX和其他存在版权问题的代码, 即使程序本身不作为公司商业软件的一部分.....	67
11.22	条款22: 谨慎使用开放源代码.....	67
11.23	条款23: 建议使用RATIONAL PURIFY工具检查和解决内存越界、泄露等运行问题.....	67
11.24	条款24: 建议使用RATIONAL PURECOVERAGE工具来统计程序代码覆盖率.....	67
11.25	条款25: 建议使用RATIONAL QUANTITY工具来分析程序的执行效率.....	67
11.26	条款26: 编程存在疑问时, 推荐先翻阅C FAQ, 然后再通过其它方式寻求答案.....	67
12.	参考书目.....	68
13.	附录一: 前缀命名法.....	69
14.	附录二: 匈牙利命名法.....	70
15.	附录三: 版权声明模版.....	72
16.	附录四: 源文件说明模版.....	73
17.	附录五: 函数定义说明模版.....	74
18.	附录六: 单词常用缩略语.....	75
19.	附录七: 常用文件组织和命名.....	79
20.	附录八: IEEE SOFTWARE QUALITY GLOSSARY.....	82

## 1. 前言

“Code, the statements or instructions in a computer program.”

“Standard, of recognized excellence or established authority.”

“Style, a particular manner or technique by which something is done, created, or performed.”

Webster

本文是菊风系统软件公司的C语言编程规范，描述了获得菊风软件研发团队一致同意的编码规则。这些规则是公司研发活动中重要的开发指导和编码约束，在未来的C语言编程过程中要严格遵守，而对于已有的程序代码要根据情况发现一个更改一个。

### 1.1 读者对象

本文适合的读者对象是从事公司内部产品和对外合作开发产品的设计、开发和测试人员。同时它也适合文档工程师作为代码规范参考手册。

### 1.2 介绍

“Programmer, a person who writes computer programs.”

“Engineer, a person who is trained in any of various branches of engineering”

Webster

在日常的软件开发过程中，编程是主要的工作内容。为了写一个程序，我们首先会在脑中形成一个大概的程序结构和过程流，接着编写符合语法规则的程序指令，使它编译通过（0错误0警告），然后修改运行过程的错误并且改进程序设计（比如提高运行效率）。写程序的人也就是程序员，但是，公司需要的开发人员不单是程序员，而是一个合格的软件工程师。软件工程师是指那些不仅能编写正确执行的，而且产出高质量的代码的程序员。高质量的代码具有以下特性：

- 易于阅读，
- 易于理解（可读性的好的并不意味着易于理解，比如冗余或重复的代码）
- 易于维护（易于查找和修改错误，易于添加新的特性）
- 复用率高（数据结构、函数、宏等程序块易于复用）
- 健壮（能够处理异常的输入和条件）
- 高可靠性（程序不可能产生错误的结果，要么是正确的执行，要么是报告错误的情况）

定义编码风格的作用主要是使代码容易阅读，无论是对于实际开发人员，还是程序代码的使用者<sup>1</sup>。良好的风格是编写高质量代码的关键步骤，实际上，书写规范的代码可以保证其中的错误更少，同时，它们比那些马马虎虎堆砌起来的、没有经过仔细推敲的代码更加的短小。草率的代码是很糟糕的代码，它不仅不美观、难以阅读，而且经常崩溃。

编程规范是从实际工作中得到的经验常识，它不是随意的规则或者处方。代码应该是清楚和简单的（具有直截了当的逻辑、自然的表达式、通行的语言使用方式、有意义的名字和有帮助作用的注释等，应该避免耍小聪明的花招，不使用非正规的结构）一致性是非常重要的东

---

<sup>1</sup> 程序的使用者是指那些直接使用或者阅读程序代码的人，比如测试人员，应用开发人员，外部购买者等。

西，如果大家都坚持同样的风格，其他人就会发现你的代码很容易阅读，你也容易读懂其他人的。

好的风格应该成为软件工程师的一种习惯。如果你在开始写代码时就关心风格的问题，如果你花时间去审视和改进它，你将会逐渐养成一种好的编程习惯。一旦这种习惯变成自动的东西，你的潜意识就会帮助你照料许多的细节问题，甚至你在工作压力下写出的代码也会更好。

本文的编程规范将讨论以下问题：

- 具有说明性的命名
- 可读的代码和注释
- 清晰的表达式
- 直截了当的控制流
- 其他有助于提高代码质量的规则和惯例

在本中我们将用一些好和不好的小程序设计例子来说明编码规则，因为对处理同样的事物的两种方式作比较常常很有启发性。这些例子不是人为的臆造的，不好的一个都来自于实际代码，由那些在太多工作负担和太少时间的压力下工作的普通程序员（偶然就是我们自己）写出来的。为了简单，这里对这些代码做了精练，并没有对它们做任何错误的解释。在看到这些代码之后，我们将重写它们，说明如何对它们做些改进。由于这里使用的都是真实代码，所以代码中可能存在多方面问题。要指出代码里的所有缺点，有时可能会使我们远离讨论的主题。因此，在有的好代码例子里也会遗留下一些未加指明的缺陷。

## 2. 命名惯例

“Convention, a general agreement about basic principles or procedures; also, a principle or procedure accepted as true or correct by convention.”

Webster

良好的命名约定能使代码易于理解，以下是一些基本惯例：

- 全局变量应该说明比较详细，局部变量应该简短
- 同一模块应该取相同的命名前缀
- 函数应该使用动词加名词的组合方式
- 坚持准确地文字命名
- 增强命名空间的概念，减少命名冲突

### 2.1 文件命名

#### 2.1.1 条款1：命名字符

- (1) 文件名称由2部分组成：主文件名和文件后缀，格式如：xxx.x
- (2) 文件名称所有字符必须是小写字母（a-z），阿拉伯数字(0-9)和下划线（‘\_’）
- (3) 主文件名称的首字符必须是字母，尾字符不能是下划线
- (4) 主文件名称的最大长度不得超过20个字符串

#### 2.1.2 条款2：命名格式

主文件名称的格式为：模块名（或产品名）+ ‘\_’ + 子模块名 [+ ‘\_’ + 动词]。

**错误示例：**

ZosAbnfDecode.c

abnf\_encode\_zos.c

**正确示例：**

zos\_abnf\_decode.c

zos\_abnf\_encode.c

#### 2.1.3 条款3：文件后缀

头文件的后缀是.h，源代码的后缀是.c

#### 2.1.4 条款4：测试文件命名

用于测试的文件命名只需模块前加 ‘t’ 作为新的模块名即可。

**正确示例：**

tzos\_tmr.c

tlsip\_cfg.c

### 2.1.5 条款5：样例文件命名

用于作为样例的文件命名只需模块前加 ‘e’ 作为新的模块名即可。

**正确示例：**

```
ezos_tmr.c
```

```
esip_decode.c
```

## 2.2 变量命名

### 2.2.1 条款1：基本规则

变量名必须由字母、数字及下划线‘\_’组成；必须由字母开头，数字开头C语言是不允许的，而下划线开头通常用于库文件里面的命名。

一般的变量由各个有意义的简写词组成，除了第一个字母为小写外（前缀命名），每个简写词以大写字母开头，后面的都是小写字母，形式为 **prefix + \* ( Word )**。

对于全局性变量（包括函数名和函数指针名）的命名要求清晰、唯一，不得在整个系统中有二义；而局部变量应该简短，特别是在循环体中的变量更要简短。

**错误示例：**

```
ZINT _myErr;           /* 不应该用 '_' 作前缀，那是系统库使用的命名习惯 */
ZCHAR g_Resource;     /* 全局变量指示不明确 */
```

**正确示例：**

```
ZINT iRet;             /* result */
ZUCHAR g_ucZosDbufInitFlag; /* dbuf initialize flag */
```

### 2.2.2 条款2：使用前缀命名法

在声明变量之时，根据每个标识符所代表的含义给它一个前缀，建议使用前缀命名法，参见附录一：[前缀命名法](#)。

**正确示例：**

```
ZUCHAR aucIp[4];           /* IPV4 address */
ZCHAR acFileName[ZOS_LOG_FILE_NAME_LEN]; /* zos log file name */
FILE *pstFile;            /* log file */
ZCHAR *pcBuf;             /* log buffer */
ST_ZOS_HASH g_stMgcpAllEvtHash; /* all event hash */

ZINT Zos_AbnfAddDbuf(ST_ZOS_ABNF_MSG *pstMsg, ST_ZOS_DBUF *pstBuf);
```

### 2.2.3 条款3：基本类型命名

为提高代码的可移植性，屏蔽不同系统之间的基本数据类型的长度差异，减少代码在不同系统之间的移植工作量，需要对基本数据类型进行类型声明，基本数据类型定义在zos\_type.h，主要有：

**ZDOUBLE** double 类型

**ZFLOAT** float 类型

**ZLONG** long 类型

**ZINT** int 类型

**ZSHORT** short 类型

**ZCHAR** char 类型

**ZULONG** unsigned long类型

**ZUINT** unsigned int类型

**ZSIZE\_T** unsigned int类型

**ZUSHORT** unsigned short类型

**ZUCHAR** unsigned char类型

**ZBOOL** 布尔类型

**ZVOID** void类型

这些基本类型的变量命名要符合[前缀命名法](#)。

#### 2.2.4 条款4：平台基本类型

主要类型有：

**ZMUTEX** 互斥类型

**ZSEM** 信号类型

**ZTIME\_T** 时间类型

**ZHRTIME\_T** 精确时间类型

**ZFUNCPTR** 函数指针类型

**ZVOIDFUNCPTR** 函数指针类型

**ZLOGID** 记录ID类型

**ZMODID** 模块ID类型

**ZINSTID** 实例ID类型

**ZTASKID** 任务ID类型

**ZTIMERID** 计时器ID类型

**ZEVTID** 事件ID类型

**ZPOOLID** 内存池ID类型

#### 2.2.5 条款5：结构类型

结构类型定义分为两个要点：

结构标签(structure tag), **tag + Name**, 如tagZOS\_LIST

数据类型名称, **ST + '\_' + Name**, 如ST\_ZOS\_LIST

**正确示例:**

```
/* IPV4 address */
typedef struct tagADDR_IPV4
{
    ZUCHAR aucIp[4];          /* IPV4 address */
} ST_ADDR_IPV4;
```

## 2.2.6 条款6: 枚举类型

枚举类型定义为,  $\boxed{\text{EN} + \text{'\_'} + \text{Name}}$

当某枚举类型为表示一组类型的时候, 常用的做法是写成  $\boxed{\text{EN} + \text{'\_'} + \text{Name} + \text{'\_'} + \text{TYPE}}$ ,

此时枚举值可以只写成  $\boxed{\text{EN} + \text{'\_'} + \text{Name} + \text{'\_'} + \text{XXX}}$ , XXX表示具体的类型名称, 如下面的例子。

**正确示例:**

```
typedef enum EN_ADDR_IP_TYPE
{
    EN_ADDR_IP_V4,
    EN_ADDR_IP_V6
} EN_ADDR_IP_TYPE;
```

## 2.2.7 条款7: 位类型

位类型定义为,  $\boxed{\text{b} + \langle \text{n} \rangle + \text{Name}}$

## 2.2.8 条款8: 变量名字书写

所有变量的名字采用大小写命名方式, 单词的首字母大写, 其他字母小写, 单词与单词之间不采用下划线连接。

**错误示例:**

```
ST_ZOS_ABNF_MSG stSip_ABNF_msg;_
```

**正确示例:**

```
ST_ZOS_ABNF_MSG stSipAbnfMsg;_
```

## 2.2.9 条款9: 除循环变量外, 不建议使用i, j, k等无意义的单字母作为变量名

当然在循环比较多或者比较深的时候, 不限于使用 l, m, n等其他单字母。

## 2.2.10 条款10: 在定义多个相关名称的时候, 注意名称意义上的对称

如init与destroy, create与delete, malloc与free, get与put配对等等。

**正确示例:**

```

/* zos memory malloc */
ZVOID * Zos_Malloc(ZSIZE_T zSize);

/* zos memory free */
ZVOID Zos_Free(ZVOID *ptr);

```

### 2.2.11 条款11: 特殊的常量

如表示单个字符的 chr, 表示va\_list的 ap 等属于比较特殊的, 因此可不受前缀命名法约束, 当然遵守它也不为过, 如 ucChr, stAp, 最关键的是在一个模块中切不可二种形式混用。

## 2.3 常量命名

### 2.3.1 条款1: 常量全用大写的字母, 用下划线分割单词

**正确示例:**

```

#define LSIP_LOG_MOD_TPT      0      /* log module of transport */
#define LSIP_LOG_MOD_CODE    1      /* log module of decode/encode */
#define LSIP_LOG_MOD_TRANS   2      /* log module of transaction */

#define ZFSM_OK                0     /* fsm run ok */
#define ZFSM_FAIL              -1    /* fsm run fail */
#define ZFSM_ERR_UNKNOWN_STA  -2    /* fsm unknown state error */

```

### 2.3.2 条款2: ZOS常量命名约定

ZOS中的常量命名采用 `'Z'+ Module +'_' + Name` 的形式。有些特殊的常量, 尤其是常用的常量, 为了保持其非常的简短, 而且突出其特殊性, 可以采用 `'Z'+ Name` 的形式, 如无符号整数类型的最大最小值。

**错误示例:**

```

#define ZOS_TASK_PREEMPT      0x00   /* enable task rescheduling */
#define ZOS_TASK_NO_PREEMPT  0x01   /* disable task rescheduling */

#define ZOS_MAXSHORT         0x7FFF  /* maximum signed short value */
#define ZOS_MAXUSHORT        0xFFFF  /* maximum unsigned short value */

```

**正确示例:**

```
#define ZTASK_PREEMPT    0x00    /* enable task rescheduling */
#define ZTASK_NO_PREEMPT 0x01    /* disable task rescheduling */

#define ZMAXSHORT        0x7FFF    /* maximum signed short value */
#define ZMAXUSHORT       0xFFFF    /* maximum unsigned short value */
```

## 2.4 函数命名

### 2.4.1 条款1：基本规则

函数名由各个有意义的简写词组成，第一个简写词是模块名跟上一个下划线作为名称前缀，名称主体是一个动词和一个名词。每个简写词以大写字母开头，后面的都是小写字母，形式为 **`Module-Name + '_' + sub-Module-Name + verb + noun`**。

**正确示例：**

```
/* alloc data memory from dbuf */
ZVOID * Zos_DbufAlloc(ST_ZOS_DBUF *pstBuf, ZULONG dwSize);

/* alloc data memory and clear data to 0 from dbuf */
ZVOID * Zos_DbufAllocClrD(ST_ZOS_DBUF *pstBuf, ZULONG dwSize);

/* only free all data block */
ZVOID Zos_DbufFree(ST_ZOS_DBUF *pstBuf);
```

### 2.4.2 条款2：函数名与其返回值在语义上不能冲突

如系统函数`getchar`，从字面意义上应返回`char`，但是实际上的返回值类型为`int`，所以下面这段代码在编译时会得到类型不匹配的告警。参见“[库函数之条款3：getchar返回一个整数，而不是一个字符](#)”

```
ZCHAR chr;
chr = getchar();
if (chr == EOF)
```

### 3. 代码注释

“Comment, a note explaining, illustrating, or criticizing the meaning of a writing.”

Webster

代码注释是保证代码良好可读性的一种重要手段，能帮助程序阅读者更好的理解代码。注释最好是能简洁而又鲜明的突出程序特征，或者能供简要的相关描述，而不要对代码本身已经能够明确的说明某种事情的代码进行注释，更不要写上莫名其妙、自相矛盾的注释，这反而严重干扰了阅读者的理解进程。

#### 3.1 条款1：注释风格

为了保证在不同编译器之间的移植，所有注释采用ANSI C的/\* \*/注释方法，不允许采用C++的// 注释风格。

**错误示例：**

```
// zos dlist maximum infinite size
```

**正确示例：**

```
/* zos dlist maximum infinite size */
```

#### 3.2 条款2：注释语言只用英文

注意：本文中示例代码中的注释可以使用中文进行说明。

#### 3.3 条款3：代码行首注释

本条款规定在代码行首进行注释时，要从第1列开始写“/\*”，然后是1个空格，接着写注释内容，最后在内容体与“\*/”之间需要有1个空格分隔开。如果本注释行过长，超过80列，则新起一行注释，并且在上一列中“\*”相同位置处为起始列，以下依次类推，以便形成一个注释块。

**错误示例1：**

```
/*zos dlist maximum infinite size*/
```

```
#define ZOS_DLIST_INFINITE_SIZE ZMAXULONG
```

**错误示例2：**

```
/* zos insert one dlist node to the after the
previous node of this dlist
if pstPrevNode is ZNULL, then node will insert
before the head as
the new head */
```

```
ZINT Zos_DlistInsert(ST_ZOS_DLIST *pstDlist, ZVOID *pPrevNode,
ZVOID *pNode);
```

**正确示例1：**

```
/* zos dlist maximum infinite size */
#define ZOS_DLIST_INFINITE_SIZE ZMAXULONG
```

**正确示例2：**

```

/*
 * zos insert one dlist node to the after the previous node of this dlist
 * if pstPrevNode is ZNULL, then node will insert before the head as
 * the new head
 */
ZINT Zos_DlistInsert(ST_ZOS_DLIST *pstDlist, ZVOID *pPrevNode,
                    ZVOID *pNode);

```

或者

```

/* zos insert one dlist node to the after the previous node of this dlist
 if pstPrevNode is ZNULL, then node will insert before the head as
 the new head */
ZINT Zos_DlistInsert(ST_ZOS_DLIST *pstDlist, ZVOID *pPrevNode,
                    ZVOID *pNode);

```

### 3.4 条款4：代码行尾注释

本条款规定在代码行尾进行注释时，要从第38列开始写“/\*”，然后是1个空格，接着写注释内容，最后在内容体与“\*/”之间需要有1个空格分隔开。如果第38列被代码占据，则从代码行尾右退1个空格处开始写注释。如果本注释行过长，超过80列，则另起一行注释，起始列跟上一行的起始列中的“\*”对齐。

**正确示例1：**

```

/* zos bucket configuration info */
typedef struct tagZOS_BKT_INFO
{
    ZULONG dwBktSize;           /* bucket size */
    ZULONG dwMaxCount;         /* bucket maixmum count */
    ZULONG dwIncCount;         /* bucket inc count per time */
} ST_ZOS_BKT_INFO;

```

**正确示例2：**

```

/* sip rpi-priv-element */
typedef struct tagSIP_RPI_PRIV_ELEM
{
    ZUCHAR ucPres;             /* present flag */
    ZUCHAR ucNetPres;         /* network present flag */
    ZUCHAR ucElemType;        /* rpi-priv-element type
                               EN_SIP_RPI_PRIV_ELEM */
    ZUCHAR ucPolicyType;      /* network type
                               EN_SIP_RPI_PRIV_ELEM_POLICY */
    ST_ZOS_SSTR stOtherElem;   /* other element token */
    ST_ZOS_SSTR stOtherPolicy; /* other policy token */
} ST_SIP_RPI_PRIV_ELEM;

```

### 3.5 条款5: 建议在预编译的结束语句右方（一个空格后）加注释标记，以表明一段程序块的结束

*正确示例:*

```
/* zos abnf library headers */
#ifdef ZOS_SUPPORT_LIB_ABNF
#include "zos_abnf_type.h"          /* zos abnf lib */
#include "zos_abnf_util.h"         /* zos abnf utility */
#include "zos_abnf_err.h"          /* zos abnf error */
#include "zos_abnf_chrset.h"       /* zos abnf character set */
#include "zos_abnf_tkn.h"          /* zos abnf token management */
#include "zos_abnf_decode.h"       /* zos abnf decode */
#include "zos_abnf_encode.h"       /* zos abnf encode */
#endif /* support abnf lib */
```

### 3.6 条款6: 函数注释

每个函数必须注释，在函数声明前面进行简单的注释，而在函数定义前面需要进行格式化的注释，参见附录四：[函数定义说明模板](#)。

*正确示例1:*

```
/* alloc data memory from dbuf */
ZVOID * Zos_DbufAlloc(ST_ZOS_DBUF *pstBuf, ZULONG dwSize);

/* alloc data memory and clear data to 0 from dbuf */
ZVOID * Zos_DbufAllocClrd(ST_ZOS_DBUF *pstBuf, ZULONG dwSize);
```

*正确示例2:*

```

/*****
Function Name: Zos_DlistCreate
Description:
    ZOS creates a dlist.

Input Parameters:
    ST_ZOS_DLIST *pstDlist    A pointer to ZOS dlist structure.
    ZULONG dwMaxNum          The maximum number of nodes in the dlist.

Output Parameters:
    None.

Return Values:
    Returns ZOK on success, or ZFAILED on failure.

*****/
ZINT Zos_DlistCreate(ST_ZOS_DLIST *pstDlist, ZULONG dwMaxNum)
{
    ...
    return ZOK;
}

```

### 3.7 条款7：函数中未实现的代码，应该添加todo注释

注释格式是 `/* todo by xxx */`，其中xxx是添加注释的程序员名字。当然也可以在注释中添加其他说明性的文字，如todo的原因和内容等。记录todo的好处是方便查找未实现功能的代码块。

#### 正确示例：

```

ZINT Zos_DumpInit()
{
    /* todo by leo */
    return ZOK;
}

```

### 3.8 条款8：用#if 0 #endif屏蔽代码块

对于需要屏蔽的大块代码，可以使用`#if 0 ... #endif`，因为C语言中的`/* ... */`注释块是不支持嵌套的。要注意`#if 0 ... #endif`是用来屏蔽暂时不使用的代码块的，而不是用来注释的，不要滥用`#if 0 ... #endif`。

另外，不要使用类似`#if NO_DEFINED_VAL_XXX ... #endif`来屏蔽代码块，因为虽然可以碰运气的确定`NO_DEFINED_VAL_XXX`是没有定义过的，但是这种做法即不规范，风险也很大。

## 4. 代码布局

“Layout, the plan or design or arrangement of something laid out.”

Webster

代码的布局是非常重要的，在一个系统中，它的代码布局应该是一致，清晰的。

一个代码文件由具有一定次序关系的上下结构单位组成，而无论是行单位或功能单位，都由一定的元素组成，如最简单的行单位就是一个“\n”。因此，布局风格就涉及到相邻单位、相邻元素之间的编码风格，其风格的一致性，也将影响软件整体的概念完整性。

### 4.1 空格

#### 4.1.1 条款1：基本条款

- (1) if、for、while等关键字之后应留一个空格再跟左括号‘(’，以突出关键字。
- (2) 函数名之后不要留空格，紧跟左括号‘(’，以与关键字区别。
- (3) ‘(’ 向后紧跟，‘)’、‘,’、‘;’ 向前紧跟，紧跟处不留空格。
- (4) ‘,’ 之后要留空格，如Function(x, y, z)。如果‘;’不是一行的结束符号，其后要留空格，如for (initialization; condition; update)。
- (5) 赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“=”、“+=”、“>=”、“<=”、“+”、“\*”、“%”、“&&”、“||”、“<<”，“^”等二元操作符的前后应当加空格。
- (6) 一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。
- (7) 象“[]”、“.”、“->”这类操作符前后不加空格。

错误示例：

```
void Func1 (int x,int y,int z);
if(dwYear>=2000)
if(a>=b&& c<=d)
for(i=0;i<10;i++)
for (i = 0; I < 10; i ++)
x=a<b?a:b;
int * piData1 = & iData2;
acStr [ 5 ] = 0;
stClass . stMember;
pstClass -> stMember;
```

正确示例：

```
ZVOID Funcl(ZINT x, ZINT y, ZINT z);
if (dwYear >= 2000)
if ((a >= b) && (c <= d))
for (i = 0; i < 10; i++)
x = a < b ? a : b;
ZINT *piData1 = &iData2;
acStr[5]= '\0';
stClass.stMember;
pstClass->stMember;
```

### 4.1.2 条款2: 程序块

程序块或复合语句 (Block or Compound-Statement) 是一个执行单元, 是组合的逻辑行, 语法上跟一个语句等价。语句 (Statement) 是以分号结尾的一个程序逻辑单元。一对大括号括在若干语句之外, 便形成了一个程序块。程序块在函数定义、if、else、while、for或do之后应用, 并且在这些位置中, 只允许一个程序块存在。注意程序块后面是不跟分号的, 如果跟了分号, 编译器便会认为是另外一个语句。为阅读清晰, 程序块的大括号各占一行。

### 4.1.3 条款3: 缩进

对于需要缩进的程序块, 以4个空格为一个单位缩进, 禁止使用tab来缩进, 当然在某些编辑环境中, 可以设置tab等价于4个空格, 但最终要在源文件中形成4个space, 而不是tab。

错误示例:

程序块的大括号没有单独一行。

程序块后面不允许加上';', 如果在 if 后面的程序块后加上';', 编译就通不过了。

因为下面的缩进用了tab, 所以看上去有点对不齐。

```
/* insert before the head node */
if (pstPrevNode == ZNULL) {
    pstNode->pstNext = pstDlist->pstHead;
    pstNode->pstPrev = ZNULL;
    pstDlist->pstHead = pstNode;
}
else /* insert after previous node in the list */
{
    pstNode->pstNext = pstPrevNode->pstNext;
    pstNode->pstPrev = pstPrevNode;
    pstPrevNode->pstNext = pstNode;    }
```

正确示例:

```

/* insert before the head node */
if (pstPrevNode == ZNULL)
{
    pstNode->pstNext = pstDlist->pstHead;
    pstNode->pstPrev = ZNULL;
    pstDlist->pstHead = pstNode;
}
else /* insert after previous node in the list */
{
    pstNode->pstNext = pstPrevNode->pstNext;
    pstNode->pstPrev = pstPrevNode;
    pstPrevNode->pstNext = pstNode;
}

```

## 4.2 代码行

### 4.2.1 条款1：行最大长度为80

代码行最大长度控制在80个字符以内，这是因为早期的显示器一行只能显示80个字符。

长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

正确示例：

```

ZINT Zos_SocketSelect (ZINT iMaxFd, ZFD_SET *pzReadFds,
                      ZFD_SET *pzWriteFds, ZINT *piMsecTimeout, ZINT *piNumFds);

```

### 4.2.2 条款2：语句长度大于80

较长的语句（>80字符）要分成多行书写，分行点最好选择操作符处。长表达式要在低优先级操作符处划分新行，操作符放在新行之首。如果是关系操作符，操作符两侧的语句左对齐。否则划分出的新行要进行适当的缩进，使排版整齐，语句可读。

正确示例1：

```

if (pstPool == ZNULL
    || pstPool->pstBktGrp == ZNULL
    || pstPool->ucBktGrpSize == 0)
{
    ZOS_LOG_ERROR((ZOS_LOGID, "BktGrpDelete invalid parameter(s)."));
    Zos_ErrnoSet(ZERR_POOL_INV);
    return;
}

```

说明：分行是从操作符||处开始，并且与上一行中的'(‘之后下一个字符列对齐。

正确示例2：

```
/* start to run timer */
if (Zos_QTimerStart(pstQTimer, zTimerId, dwTimerType, dwTimeLen, dwParm,
                   pfnActive) != ZOK)
{
    ZOS_LOG_ERROR((ZOS_LOGID, "TimerStart start queue node.));
    Zos_ErrnoSet(ZERR_TIMER_START);
    return ZFAILED;
}
```

说明：分行是从函数参数处开始，并且与上一行中函数名字后的‘(’之后的下一个字符列对齐  
正确示例3：

```
ZINT Zos_SocketSelect (ZINT iMaxFd, ZFD_SET *pzReadFds,
                      ZFD_SET *pzWriteFds, ZINT *piMsecTimeOut, ZINT *piNumFds);
```

说明：分行是从函数参数声明处开始，并且是从4次缩进（16个空格）后的第17列开始

### 4.2.3 条款3：if、for、do、while、case、switch、default等语句独占一行，建议在if、for、do、while等语句的执行语句部分加上括号{}

在程序块中只有一条语句时，如果其后有空行来分隔不同意义的程序块，可以不用括号把程序块括起来。

正确示例3：

```
if (m_dwZosDbufInitFlag)
    return ZOK
```

....

4.2.4 条款4：程序块的分界符（‘{’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语句中的代码都要采用如上的缩进方式。

4.2.5 条款5：相对独立的程序块之间、变量说明之后必须加空行在函数中的局部变量是在所有变量都说明后加空行。

正确示例：

```
ZVOID * Zos_DbufAlloc(ST_ZOS_DBUF *pstBuf, ZULONG dwSize)
{
    ST_ZOS_DBUF_DATA *pstData;
    ZCHAR *pcMemAddr = ZNULL;
    ZULONG dwFreeSize, dwAllocSize;

    ...
}
```

#### 4.2.6 条款6: 不建议把多个短语写在一行中, 即一行只写一条语句

有时候为了保证某种简短的关联语句组合, 可以尝试使用。

正确示例:

```
#define ZOS_POOL_MUTEX_LOCK(_pool) \
    if ((_pool)->ucIsNeedMutex) Zos_MutexLock(&((_pool)->zMutex))
```

### 4.3 头文件

早期的编程语言如Basic、Fortran没有头文件的概念, 但为什么C语言需要头文件呢? 下面是解释:

(1) 通过头文件来调用库功能。在很多场合, 源代码不便(或不准)向用户公布, 只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能, 而不必关心接口怎么实现的。编译器会从库中提取相应的代码。

(2) 头文件能加强类型安全检查。如果某个接口被实现或被使用时, 其方式与头文件中的声明不一致, 编译器就会指出错误, 这一简单的规则能大大减轻程序员调试、改错的负担。

#### 4.3.1 条款1: 防止头文件重复被引用

为了防止头文件被重复引用, 应当用ifndef/define/endif结构产生预处理块。预处理块中的字符串的命名参照文件名的每个单词, 展开成形式 `*('_' + word) + '_'`。

正确示例:

```
#ifndef _SIP_ENCODE_H__
#define _SIP_ENCODE_H__
```

#### 4.3.2 条款2: 让C++编译器使用C的命名方式

编译后生成的目标代码中的符号命名和函数中参数的入栈方式, C++编译器一般与C编译器不同, 因此在C语言的头文件中需要使用extern "C"{}来包含所有代码, 来告诉C++编译器对于本文件中的符号命名和函数参数入栈使用C的方式。

正确示例:

```
#ifdef __cplusplus
extern "C" {
#endif
...
#ifdef __cplusplus
}
#endif
```

### 4.3.3 条款3：文件包含

用 `#include <filename.h>` 格式来引用标准库的头文件（编译器将从标准库目录开始搜索）。用 `#include "filename.h"` 格式来引用非标准库的头文件（编译器将从用户的工作目录开始搜索）。

除了以此来确定头文件的查找路径，通过规范命名使标准库与非标准库有所差别，不建议把标准库中的头文件以“xxx”的方式来引用，尽管在编译时也能正确地找到。

### 4.3.4 条款4：建议头文件中只存放“声明”而不存放“定义”

声明和定义包括函数、结构、枚举、联合和变量等各类C语言的identifiers。为什么建议在头文件中存放“声明”，想一想头文件被include以后，预处理器在源文件中展开的情形吧。头文件被包含之后，对于用于声明的代码，编译器（包括连接器）的工作就是建立符号和关联，而对于用于实现的代码，编译器便会生成实际代码，如果实现代码被多次包含，则编译出来的程序可能会变得十分庞大。

当然，为了封装算法，而在头文件中定义宏的情况是例外的；而实际上在C++标准中封装算法的template函数和为了避免函数调用开销的inline函数也是在头文件中定义的。

### 4.3.5 条款5：头文件布局

segments	content
part 1	Copyright-Declaration
part 2	File-Description
part 3	<pre>#ifndef _FILENAME_H__ #define _FILENAME_H__  #ifdef __cplusplus extern "C" { #endif</pre>
part 4	Head-File-inclusions
part 5	Typedefs Enums Constant-Macro-Defines Parameterized-Macro-Defines <b>Shared-Declarations</b>
part 6	External-Data-Declarations
part 7	External-Function-Declarations
part 8	<pre>#ifdef __cplusplus } #endif  #endif /* _FILENAME_H__ */</pre>

### 4.3.6 条款6：头文件行数不要超过1万行

如果超出1万行，分割成多个头文件，文件命名采用序号的方式，如xxxx1.h、xxxx2.h等。

### 4.3.7 条款7：自定义头文件避免使用和系统头文件一样的名字

如果自定义头文件的名字和系统头文件一样，当在用户的工作目录找不到该头文件时，编译器会在标准库目录中找同名的头文件，这样就有可能发生包含了不想包含的文件的情况。因此规范命名显得尤为重要。

错误示例：

```
#include "math.h"
```

正确示例：

```
#include <math.h>
#include "mymath.h"
```

### 4.3.8 条款8：不建议使用包含绝对路径的用法

当编译环境发生变化时，使用绝对路径有可能导致编译失败。建议使用相对路径，通过在编译时指定目录的方式来增加头文件的搜索路径。

错误示例：

```
#include "d:/zocol/src/protocol/sdp/sdp.h"
```

正确示例:

```
#include "sdp/sdp.h"
```

### 4.3.9 条款9: 不建议使用反斜杠“\”表示路径

使用反斜杠会降低可移植性, 建议用斜杠“/”代替, 即使在某些使用反斜杠作的操作系统, 它们的系统库函数可以实现正反斜杠的互相转换。

错误示例:

```
#include "sdp\sdp.h"
```

正确示例:

```
#include "sdp/sdp.h"
```

## 4.4 源文件

### 4.4.1 条款1: 源文件布局

segments	content
part 1	Copyright-Declaration
part 2	File-Description
part 3	Head-File-inclusions
part 4	Typedefs Enums Local-Declarations Constant-Macro-Defines Parameterized-Macro-Defines
part 5	Global-And-Local-Static-Data-Definitions
part 6	Local-functions-Declaration
part 7	Shared-And-Local-Functions-Definition

### 4.4.2 条款2: 源文件行数不要超过3万行

如果超出3万行, 分割成多个源文件。有些编译器对过大的源文件会出现编译问题, 应尽量避免此种情况发生。文件命名采用序号的方式, 如xxxx1.c、xxxx2.c等。

### 4.4.3 条款3: 建议不要引用不必要的头文件

- 头文件中引用不必要的头文件

```
File a.h      #include "b.h"
```

```
      c.h      #include "a.h"
```

```
          #include "b.h" /* 不必要的头文件 */
```

- 源文件中引用不必要的头文件

```
File a.h      #include "b.h"
```

```
a.c      #include "a.h"  
        #include "b.h" /* 不必要的头文件 */
```

## 5. 类型，操作符和表达式

“Type, a particular kind, class, or group.”

“Operator, something and especially a symbol that denotes or performs a mathematical or logical operation.”

“Expression, a mathematical or logical symbol or a meaningful combination of symbols.”

Webster

类型决定了一个对象的值集合和对象可以操作的方式。操作符指明了要对变量和常量等对象做什么操作。表达式就是组合各种变量和常量从而产生新的值。

ANSI C具有有一些操作特性，比如在编译时可以将多个相邻的字符常量连接在一起。

对象可以声明为const类型，表明它可以初始对象，但随后就不能被修改。const的目的是说明对象处于只读的内存空间，这样或许能提高编译优化的机会。

对象也可以声明为volatile类型，以通知编译器在优化的时候，它要保留特有的特性。因为编译器在优化的时候，为了加快速度，就可能直接读取寄存器备份的值，但变量的值可能随时会变化，所以要用volatile限定词加以强调。所以说，volatile的目的是抑制编译优化。

### 5.1 数据类型和大小

#### 5.1.1 条款1：要始终明确数据类型及其最大范围

小心使用ZINT类型（int类型），C标准没有定义int类型是16bit的还是32bit，而是由编译器来决定。如果要明确使用32bit的整型，可以使用long，因为C语言标准规定long类型至少为32bit，而short至少为16bit。

同样，不要没有检查便假设数据类型的大小和精度，C语言标准没有规定int的最大范围，没有规定float、double最大范围和精度，这个依赖于编译器的实现。所以有必要时，我们要检查这些范围，我们可以通过访问标准头文件<limits.h>和<float.h>，来得到某数据类型的大小范围。

错误示例：

```
ZINT iLogLevel = 0xFFFF0000; /* 不应假设ZINT具有32bit */
```

正确示例：

```
ZULONG dwLogLevel = 0xFFFF0000;
```

### 5.2 常量

#### 5.2.1 条款1：常量定义

在C语言中，我们把常量分为宏定义常量、字符串常量、枚举常量。其中，宏定义常量是在编译前处理，字符串常量、枚举常量在编译时处理。这些变量在定义时，建议在定义关键字后一个空格处书写名称。

**正确示例：**

```
#define ZOS_LOGID g_zZosLogId

ZCHAR m_acZosVersion[] = "1.0_Beta";
ZCHAR m_acZosVersion[] = "1.0" "_Beta";

/* zos ascii character table */
typedef enum EN_ZOS_CHR
{
    EN_CHR_EOS = 0x00,          /* end '\0' */

    EN_CHR_HT = 0x09,          /* horizontal tab */
    EN_CHR_LF = 0x0A,          /* \n */
    EN_CHR_VT = 0x0B,          /* vertical tab */
    EN_CHR_CR = 0x0D,          /* \r */
    EN_CHR_WSP = 0x20,          /* white space */
    EN_CHR_DQUOTE = 0x22,      /* " */
    ...
    EN_CHR_GENRALSTR = 0x100    /* general token */
} EN_ZOS_CHR;
```

### 5.2.2 条款2: 不允许直接出现数字, 必须用有意义的枚举或宏(常量)来代替

```
if (pstTrans->ucType == EN_LSIP_TRANS_ICT
    && pstTrans->iState == EN_LSIP_ICT_STATE_TERMINATED)
    return ZTRUE;
```

应改为如下形式。

```

/* sip lite transaction type */
typedef enum EN_LSIP_TRANS_TYPE
{
    EN_LSIP_TRANS_ICT,          /* invite client transaction */
    ...
} EN_LSIP_TRANS_TYPE;

/* sip invite client transaction state */
typedef enum EN_LSIP_ICT_STATE_TYPE
{
    ...
    EN_LSIP_ICT_STATE_TERMINATED = 5,
    ...
} EN_LSIP_ICT_STATE_TYPE;

if (pstTrans->ucType == EN_LSIP_TRANS_ICT
    && pstTrans->iState == EN_LSIP_ICT_STATE_TERMINATED)
    return ZTRUE;

```

### 5.2.3 条款3: 常量的位置

需要对外公开的常量放在头文件中, 不需要对外公开的常量放在定义文件的头部。为便于管理, 可以把不同模块的常量集中存放在一个公共的头文件中。

如果某一常量与其它常量密切相关, 应在定义中包含这种关系, 而不应给出一些孤立的值。

#### 正确示例:

```

/* zos log id */
extern ZLOGID g_zZosLogId;
#define ZOS_LOGID g_zZosLogId

```

## 5.3 变量

### 5.3.1 条款1: 变量定义

应该保证每一行定义一个变量。因为这样便于对每个变量作出注释并初始化。

注意, 对于全局变量和静态变量, 编译系统会自动地把变量初始化为 0, 但对于一般的变量, 系统初始值是未定义的, 所以在变量定义的同时应该初始化。但为了保证可读性, 建议在初始化赋予初值, 尽管其值为0。

#### 错误示例:

```

ZINT iLower, iUpper, iStep;
ZCHAR chr, acLineStr[1000];

```

#### 正确示例:

```
ZINT iLower = 20;          /* XXXXXXXX */
ZINT iUpper = 50;         /* XXXXXXXX */
ZINT iStep = 10;          /* XXXXXXXX */
ZCHAR chr = 'a';
ZCHAR acLineStr[1000];
```

### 5.3.2 条款2: 加上const限定词的声明

在变量前面加上限定词“const”，便意味着此变量是不允许被改变的，编译器可以根据这个const指示，检查此变量是否被改变。

如果希望在一个参数传入函数以后而不被修改，也要使用const限定词进行函数的参数声明，这样用户就能够明确哪些参数是只读的，参见函数命名之条款3: [函数参数](#)。

### 5.3.3 条款3: 变量定义顺序

推荐按数据类型的长度排序本地变量。如果第一个变量对齐了，其它变量就会连续的存放，而且不用填充字节自然就会对齐。有些编译器在分配变量时不会自动改变变量顺序，有些编译器不能产生4字节对齐的栈，所以可能4字节不对齐。

### 5.3.4 条款4: 变量定义必须放在函数的入口处，禁止C++风格的即用即定义（在程序中间进行变量定义）

但有时在由括号或大括号括起来的语句块中，变量定义是可以的，但要注意其作用域，尤其是在函数中定义了多个相同名称的变量。

#### 正确示例:

```
ZINT Zos_Printf(const ZCHAR *pcFormat ,...)
{
...
    /* is it print runtime */
    if (g_dwZosPrintFlag)
    {
        ZCHAR acTempBuf[2048];

        iRet = Zos_VSNPrintf(acTempBuf, 2047, pcFormat, ap);
        if (iRet >= 0)
        {
            acTempBuf[iRet] = '\\0';
            ...
        }
    }
...
}
```

如果说acTempBuf在函数中被不同的语句块使用时，建议放置在函数的开始处定义，扩大其作用域。

### 5.3.5 条款5：防止局部变量与全局变量同名

局部变量与全局变量同名会导致名字空间污染，并有可能影响程序的正确性，降低程序的可读性。

## 5.4 算术操作符

### 5.4.1 条款1：注意运算符“/”、“%”

(1) C标准规定，对于 float 和 double 类型，运算符“%”是不可运算的；而有些编译器会自动地把算术表达式中的浮点类型转换为整型，此时要注意小数点以后的精度问题。

(2) 对于负数的运算，“/”运算符截取整型的方向（向上或者向下）和“%”运算符的运算的正负结果是未定义的，因为运算结果是依赖于机器的。所以，我们不能预期确定在这种情况下运算的结果。

**错误示例：**

```
ZLONG dwData;
/* these are wrong expects! */
dwData = 5 % -3;      /* expect data is 2 at every machine */
dwData = -5 % -3;    /* expect data is -2 at every machine */
dwData = -5 % 3;     /* expect data is -2 at every machine */

dwData = 5 / -3;     /* expect data is -1 at every machine */
dwData = -5 / -3;   /* expect data is 1 at every machine */
dwData = -5 / 3;    /* expect data is -1 at every machine */
```

todo: the value of dwData needs to be tested at other machines.

## 5.5 关系和逻辑操作符

### 5.5.1 条款1：布尔变量与零值比较

不可将布尔变量直接与TRUE、FALSE，1、0或其他任何自定义布尔值进行比较。

根据布尔类型的语义，零值为“假”（记为FALSE），任何非零值都是“真”（记为TRUE）。TRUE的值究竟是什么并没有统一的标准。例如Visual C++ 将TRUE定义为1，而Visual Basic则将TRUE定义为-1。使用ZOS平台的代码，布尔类型的值一律用ZTRUE与ZFALSE来进行比较。

**错误示例：**

```
if (bFlag == TRUE)
if (bFlag == 1 )
if (bFlag == FALSE)
if (bFlag == 0)
```

**正确示例：**

```
if (bFlag)      /* indicate the bFlag is true */
if (!bFlag)     /* indicate the bFlag is false*/
if (bFlag == ZTRUE) /* indicate the bFlag is using ZOS platform */
if (bFlag!= ZTRUE) /* indicate the bFlag is using ZOS platform */
if (bFlag == ZFALSE) /* indicate the bFlag is using ZOS platform */
```

### 5.5.2 条款2: 浮点变量与零值比较

不可将浮点变量用“==”或“!=”与任何数字比较。

千万要留意，无论是float还是double类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“!=”与数字比较，应该设法转化成“>=”或“<=”形式。

假设浮点变量的名字为x，应当将

```
if (fData == 0.0)      /* wrong compare */
```

转化为

```
if ((fData >= -EPSINON) && (fData <= EPSINON))
```

其中EPSINON是允许的误差（即精度）。

### 5.5.3 条款3: 指针变量与零值比较

应当将指针变量用“==”或“!=”与ZNULL或其他自定义ZNULL值比较。

指针变量的零值是“空”（记为ZNULL）。尽管ZNULL的值与0相同，但是两者意义不同。假设指针变量的名字为p，它与零值比较的if语句如下：

错误示例：

```
if (p == 0)
if (p != 0)
if (p == NULL)
if (p != NULL)
```

正确示例：

```
if (p == ZNULL)
if (p != ZNULL)
```

### 5.5.4 条款4: 注意赋值操作和比较操作的区别

有时候我们可能会看到 `if (ZNULL == p)` 这样的格式，那是程序员为了防止将 `if (p == ZNULL)` 误写成 `if (p = ZNULL)`，而有意把p和ZNULL颠倒。编译器认为 `if (p = ZNULL)` 是合法的，但是会指出 `if (ZNULL = p)`是错误的，因为ZNULL不能被赋值。

但在ZOS平台中，为了保证代码统一，不建议使用此种风格。一方面此种问题(`if p = ZNULL`)可能会在编译时被警告，另外代码在测试和发布前需要通过pclint检查，很方便的定位。重要的是养成良好的习惯，并经常检视代码。

### 5.5.5 条款5: 注意数学表达式和逻辑表达式的区别

不要把程序中的复合表达式与“真正的数学表达式”混淆。

例如：

```
if (a < b < c)           /* a < b < c is mathematics expression! */
```

并不表示

```
if ((a < b) && (b < c))
```

而是成了令人费解的

```
if ((a < b) < c)
```

虽然编译器认为这是合法的，但是事实上上式的含义是，让变量 `c` 跟一个逻辑运算的表达式 `(a < b)` 去比较大小了，而逻辑表达式 `(a < b)` 的值永远为 0 或 1。

## 5.6 类型转换

### 5.6.1 条款1：注意从char到int的类型转换符号问题

虽然，实际上可以认为char就是一个8bit的整型，但C语言没有定义char类型最高位是否是用来表示正负符号的，所以当把一个任意的char类型的变量转换为int时，得到的数据的正负值是依赖于机器的。也就是说有的机器把char转换int的时候保留第一位，并把它前移到int的第一位，那么得到的可能就是负的；而有的机器直接在char前面加了多个0，那么得到的永远是正的。

对于这个问题，如果char类型的变量中存放着并非表示字符的整型数据，建议明确使用unsigned char或signed char来保证可移植性。

### 5.6.2 条款2：避免有符号和无符号类型之间直接的运算

以下是没有符号的情况下的C语言默认转换规则：

- (1) 如果操作变量中有一个long double，则把其他的变量都转换为long double；
- (2) 否则，如果操作变量中有一个double，则把其他的变量都转换为double；
- (3) 否则，如果操作变量中有一个float，则把其他变量都转换为float；
- (4) 否则，把char和short类型转换为int类型。
- (5) 然后，如果操作变量中有long类型的，把其他的变量都转换为long类型。

然而，加上了正负符号之后的类型转换规则就复杂得多了，所以，要避免有符号和无符号类型之间的直接的运算，因为在不同类型比较过程当中，C编译器会进行隐式的类型转换，而那种转换可能会是令人迷惑的复杂转换。比如，假设系统中int类型的长度是16bit的；那么  $-1L < 1U$ ，因为系统把1U转换成了long类型了；而  $-1L > 1UL$ ，因为系统把-1L转换为unsigned long，而变成一个很大的数值了。

在这种有符号和无符号的类型的运算的情况下，用户要进行显式的进行类型转换。事实上，有些编译器也会提醒这一点的。

错误示例：

```
ZULONG dwValueX = 1;
ZLONG dwValueY = -1;
ZBOOL bRet;
bRet= (dwValueX > dwValueY);    /* expect bRet will be true, but it does
                                not work */
```

正确示例:

```
ZULONG dwValueX = 1;
ZLONG dwValueY = -1;
ZBOOL bRet;
bRet= ((ZLONG)dwValueX > dwValueY);    /* expect bRet will be true */
```

### 5.6.3 条款3: 尽量减少没有必要的数据类型默认转换与强制转换

只要你注意, 代码当中很多地方存在类型转换, 包括赋值运算, 数学表达式、类型比较、传入参数等, 一般转换的规则是小的往大的转, 低精度的往高精度的转, 这不但使代码变得冗长, 而且每次类型转换, 编译器都必须分配一个空间来存放新的类型, 降低了效率, 所以, 要减少在没有必要的情况下的数据类型转换。

### 5.6.4 条款4: 关注类型转换的代价

在各种情况下的类型转换, 是要付出代价的, 除了每次类型转换, 编译器都必须分配一个空间来存放新的类型以外, 类型转换的精度也可能丢失。

## 5.7 自增和自减操作符

### 5.7.1 条款1: 注意自增和自减操作符的前缀和后缀的区别

自增和自减操作符作用在变量的前面和后面之后, 如 `iValue++` (后缀)、`++iValue` (前缀), 变量都会增 1 或者减 1。其区别是, 如果用于前缀则先再加 1 或减 1 再返回变量值, 而用于后缀的话则先返回变量值再加 1 或减 1。例如:

```
ZINT i = 0;
ZINT iVal = 1;
iVal = i++;    /* iVal will be 0 and i will be 1 */
iVal = ++i;    /* iVal will be 2 */
```

## 5.8 位操作符

### 5.8.1 条款1: 基本条款

在C语言中为了提高运算效率, 在不影响理解的情况下, 尽量使用位操作运算, 比如 `>> 2` 相当于除以4, 但是速度上会有很大差异。C语言的位运算符有:

&	bitwise AND
	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement (unary)

注意位运算符 | 和 & 跟逻辑运算符 || 和 && 的区别，它们是没有任何联系的。

举例：

```
/* GetBits: get n bit from position iPos */
ZUINT GetBits(ZUINT iVal, ZINT iPos, ZINT n)
{
    return (iVal >> (iPos + 1 - n) & ~(~0 << n))
}
```

其中，~0表示所有的位为 1 。

### 5.8.2 条款2：关注位移运算符对于有符号整数进行的右移 >> 运算

关于位移运算符对于符号的处理有下列规则：

(1) 位移运算符 >> 和 <<, 对于 unsigned 整型类型进行移动，直接移开的几位，后面填 0，这个没有任何问题。

(2) 而左移位运算符 << 对于 signed 整型类型进行移动时，也是直接向左移开的几位，后面填 0，结果就是把符号位移出去，这个称为逻辑位移。

(3) 而右移位运算符 >> 对于有符号整型进行操作之时，如果被操作变量是负数的，那么位移以后的正负是依赖于机器的，有的机器保留原先的正负标志（称为算数位移），而有的机器则直接填零（称为逻辑位移）。

## 5.9 赋值操作符和表达式

### 5.9.1 条款1：推荐使用复合二元操作符的赋值运算，如 +=

可以复合赋值的二元操作包括：

+ - \* / % << >> & ^ |

复合操作符的赋值运算的好处是，一方面可读性较强，另一方面提高编译器翻译成汇编语言的效率。对于 `i += 10;` 和 `i = i + 10;` 运行结果是一样的，但前者不需要另外分配空间，直接把10加给了i 。

注意，复合操作符的赋值运算的优先级是最低的，`x *= y + 1` 等价于 `x = x * (y + 1)`，而不是 `x = x * y + 1` 。

### 5.9.2 条款2：注意赋值操作的返回

赋值操作也可以出现在表达式中。可以把赋值操作看作一个函数，它返回的类型就是赋值操作符 “=” 左边的变量类型，它的返回值是赋值操作完成以后的值。

例如：

```
while ((chr = getchar()) != EOF )
{
    ...
}
```

## 5.10 条件表达式

### 5.10.1 条款1：使用条件表达式简化 if ... else

使用表达式 `expr1 ? expr2 : expr3` 代替简单的 `if ... else` 可以使程序更加简洁。

比如，可以将下列程序

```
if (condition)
{
    return statement1;
}
else
{
    return statement2;
}
```

改写为更加简洁的形式

```
return (condition ? statement1 : statement2);
```

## 5.11 优先级和赋值次序

### 5.11.1 条款1：大体上明确操作符的优先级，或者使用括号 ()

要大体上明确操作符的优先级，至少明确算数操作符要高于逻辑操作符，逻辑操作符高于赋值操作符就可以了，下表列出了C语言的操作符的优先等级。当你忘记某些优先级的细节之时，可以使用括号 ()，因为括号的优先级是最高的。

优先级	运算符	结合律
from	() [] -> .	left to right
	! ~ ++ -- + - * & (type) sizeof	right to left
	* / %	left to right
	+ -	left to right
higher	<< >>	left to right
	< <= > >=	left to right
to	== !=	left to right
	&	left to right
lower	^	left to right
		left to right
	&&	left to right
		right to left
	?:	right to left
	= += -= *= /= %= &= ^=  = <<= >>=	left to right

	,	left to right
--	---	---------------

### 5.11.2 条款2: 避免“side effects”语句的不良应用

所谓side effects语句就是在计算表达式的时候，同时进行了表达式中的变量的变化。有一种情况就会让人感觉很不舒服，比如：

```
a[i] = i++;
```

到底是要给当前的 a[i] 赋值呢，还是个给 i++以后的 a[i] 赋值？这个问题是依赖于编译器的。那么为什么C标准没有做出这个规定，而把这个判断权力交给编译器？那是因为由编译器决定操作的顺序是可以最大程度的发挥不同机器的效率。

总体上来说，这种依赖于计算顺序的编程风格是应该避免的。

## 6. 控制流

“The control-flow of a language specifies the order in which computations are performed.”

K&R C

### 6.1 If-Else语句

If-Else的语法是:

```
if (expression)
    statement1
else
    statement2
```

注意: *statement<sub>1</sub>*和*statement<sub>2</sub>*是语句或者复合语句, 因此如果*statement<sub>1</sub>*被写成{};的形式相当于一个不带else子句的if语句加一个空语句, else被认为是一个独立的语句, 编译器在遇到这种情况时发生错误。相同的情况发生在*statement<sub>2</sub>*时没有问题, 因为此时相当于一个 If-Else语句加一个空语句。

#### 6.1.1 条款1: 尽量避免在if的判断条件中使用赋值语句

错误示例:

```
if ((iRet = Zos_AbnfGetIPv4(pstAbnfMsg, &pstAddr->u.stIPv4.u.dwIp)) == ZOK)
{
    pstAddr->ucType = EN_SDP_ADDR_IPV4;
    goto suss;
}
```

正确示例:

```
iRet = Zos_AbnfGetIPv4(pstAbnfMsg, &pstAddr->u.stIPv4.u.dwIp);
if (iRet == ZOK)
{
    pstAddr->ucType = EN_SDP_ADDR_IPV4;
    goto suss;
}
```

### 6.2 Switch语句

#### 6.2.1 条款1: switch语句必须有default语句, 每个case语句建议有break语句

在那些case中执行的语句是相同时, 可以在多个case后使用一个break语句。

#### 6.2.2 条款2: 避免相邻的case语句共用语句, 如果一定要共用必须给出明确的注释加以说明

相邻的case共用语句会使代码变得不够健壮, 当程序被修改时容易变得支离破碎, 如果一定要这样做, 必须小心谨慎, 并且给出明确的注释加以说明。

错误示例:

```
switch ((re_opcode_t)*p1++)
{
case no_op:
case begline:
case endline:
case begbuf:
case endbuf:
case wordbeg:
case wordend:
case wordbound:
case notwordbound:
    break;
...
}
```

**正确示例:**

```
switch ((re_opcode_t) *p1++)
{
case no_op:          /* some comment on why fall through
                      the next case*/
case begline:
case endline:
case begbuf:
case endbuf:
case wordbeg:
case wordend:
case wordbound:
case notwordbound:
    break;
...
}
```

**6.2.3 条款3: 在最后一个case中（包括default）加入break，尽管在逻辑上不需要**

default和其他case一样，不会自动break，因此如果有一天有人在default后新加了一个case，而default后又没有加break，那么当运行时程序一旦执行了default后的语句，还会继续执行新加的case里的语句。

**正确示例:**

```
switch (pstTask->ucState)
{
case EN_TASK_READY:
    pcState = "ready";
    break;
case EN_TASK_RUNNING:
    pcState = "runing";
    break;
default:
    break;
}
```

## 6.3 While和 For 循环

### 6.3.1 条款1: 建议for语句循环变量的取值采用半开半闭区间

使用半开半闭区间的目的是为了增加程序的可读性。

**错误示例:**

```
for (j = 1; j <= pstMgr->dwTableSize; j++)
{
    ...
}
```

**正确示例:**

```
for (j = 0; j < pstMgr->dwTableSize; j++)
{
    ...
}
```

### 6.3.2 条款2: 谨慎使用for语句中的逗号

‘,’是C语言中的操作符,经常用在for语句中,一对用逗号分隔的表达式是从左向右计算值的,最终的计算结果是操作符右边的操作数的计算结果,如:

```
ZINT i = 1;
ZINT j;
j = (i - 1, i + 1);
```

j的值为2。for中的逗号应该被小心使用。

注意: 函数参数中和变量声明中的逗号不是操作符。

## 6.4 Goto语句和标签

goto语句在C语言中用于实现无条件跳转,从实用角度讲, goto语句并不是实现程序结构所必需的,它可以完全被其他代码所代替。

### 6.4.1 条款1: 尽量少用和慎用goto语句

goto语句应该被小心对待, 一般来说, goto语句主要用于从多层的switch、for或while嵌套中跳转出来。具有goto的代码比没有goto的代码要更难于理解和维护, 在我们的程序中应该尽量少用goto语句。

### 6.4.2 条款2: 如果不得不使用goto, 标签应该单独成行, 并且比下面的代码向左缩进四格

标签的命名方式是全部小写, 尽量控制一个单词内, 并且要能正确反应跳转的本意, 像g1, g2这些毫无意义的标签就是拙劣的命名方式。

正确示例:

```
/* set the ipv6 type */
pstAddr->ucType = EN_SDP_ADDR_IPV6;
suss:
/* set addr present to true */
pstAddr->ucPres = ZTRUE;
```

## 7. 函数和程序结构

“Function, any of a group of related actions contributing to a larger action

Structure, the aggregate of elements of an entity in their relationships to each other.”

Webster

函数提供了一种方便的方式来封装一些计算,使得用户只管使用函数而不需要关心函数的实现。正确的设计函数接口,应该是只需要知道它会做什么,而不用关心它是如何实现的。

ANSI C支持inline函数, inline可以出现多次,但效果等同于一次。inline函数并不提供外部的定义,如果需要外部定义,必须要用extern限定词。

### 7.1 函数基础

#### 7.1.1 条款1: 函数参数

(1) 不推荐函数参数过多,一般超过5个就过长了,建议以某种结构代替某些参数。

(2) 如果参数是指针,且仅作输入用,则建议在类型前加const,以防止该指针在函数体内被意外修改。

**正确示例:**

```
ZINT Zos_StrLen(const ZCHAR *pcStr);
```

**7.1.2 条款2: 函数传参时,禁止直接使用结构进行传参,而应该使用结构指针**  
结构传参会增加函数堆栈的大小,而且也会降低函数执行效率。

#### 7.1.3 条款3: 函数如果没有参数,建议使用ZVOID填充

此条款不强制使用,二种声明方式都是正确的。

**正确示例:**

```
ZINT Zos_TimerInit();
```

**正确示例:**

```
ZINT Zos_TimerInit(ZVOID);
```

#### 7.1.4 条款4: 使用ANSI C99定义函数

使用ANSI C定义函数的目的是为了增加代码的可移植性。

**错误示例:**

```
ZINT Zos_SocketOpen(ucType, ucProtocol, pzSock)
{
    ZUCHAR ucType;
    ZUCHAR ucProtocol;
    ZSOCKET *pzSock;
    ...
}
```

**正确示例:**

```
ZINT Zos_SocketOpen(ZUCHAR ucType, ZUCHAR ucProtocol, ZSOCKET *pzSock)
{
    ...
}
```

### 7.1.5 条款5: 尽量少定义具有可变参数的函数

具有可变参数的函数不能被很好的移植,同时不能进行严格的参数检查。如果一定要定义可变参数,使用系统宏来定义它们。

### 7.1.6 条款6: 建议提供函数声明

提供函数声明的目的是为了进行严格的参数和返回值检查,如果没有函数声明,编译器会在第一次碰到函数的地方隐含声明该函数,这个地方可能是第一次调用函数的地方,也可能是定义函数的地方。

### 7.1.7 条款7: 函数不能返回指向函数堆栈的指针

在函数内部定义的变量都从函数堆栈分配空间,当从该函数退出时,指向这部分空间的指针变得没有意义。

**错误示例:**

```
ZCHAR * foo(ZINT n)
{
    ZCHAR acBuf[BUF_SIZE];
    ...
    return acBuf;
}
```

### 7.1.8 条款8: 少用ZBOOL类型作为参数和返回值

原因有二,其一是ZBOOL值含义不明确,在调用时很难知道该参数到底传达的是什么意思;其二是ZBOOL参数值不利于扩充,而其所占的空间确跟整型一样。

### 7.1.9 条款9: 对于提供了返回值的函数,要进行返回值判断,以保证返回的各种情况都可以得到正确的处理

同样,也要规划函数的返回值,一个函数的返回值应当是合理清晰的,并且整个软件系统中所有的函数的返回值也要保证相对的一致性。另外,也要避免返回没有意义的返回值,使得上层调用对于返回值的处理逻辑清晰有效。

### 7.1.10 条款10: 如果函数返回值需要详细解释, 在函数说明中予以描述

正确示例:

```
Returns ZOK on success, or ZFAILED on failure.
```

### 7.1.11 条款11: 函数参数的有效性判断

模块对外提供的接口函数在函数的入口处必须进行参数有效性判断, 内部函数如果可以保证在函数调用处的参数判断, 可以在函数入口处进行参数有效性判断。

**正确示例:**

```
ZVOID Zos_LogStr(ST_ZOS_LOG_MGR *pstLogMgr, ZCHAR *pcDesc)
{
    ZULONG dwLen;

    if (pstLogMgr == ZNULL || pcDesc == ZNULL)
        return;
    ...
}
```

### 7.1.12 条款12: 不打算被其他文件使用的函数要声明为static

声明为static后, 此函数的作用域便缩小为本文件可见, 即刻看作私有函数, 在其他文件中即使定义了同名的函数也不会引起链接时的错误。

### 7.1.13 条款13: 函数不能返回结构体

不同的编译器对函数返回结构体的处理不同, 因此在链接由不同编译器编译的目标文件时会出现问题。用指向结构体的指针代替结构体。

## 7.2 全局变量

全局变量指定义于任何函数之外的变量, 它们的空间从任务的静态变量区分配。缺省情况下, 全局变量可以被任何函数引用 (与全局变量不在同一个源文件中的函数通过extern声明来引用), 在使用了static限定词后, 全局变量只能被本源文件中定义的函数所引用。

### 7.2.1 条款1: 尽量减少在模块间直接使用全局变量, 对模块间的数据访问最好提供函数接口

如果必须使用全局变量, 一种建议的方法是, 通过static修饰符把全局变量封装为私有的, 然后通过一对存取函数对此操作。这种方式带来的灵活性有, 如果这个全局资源需要被多个任务访问, 则可以把存取函数改为可重入函数; 另外, 还可以在函数中加上对资源的存取的校验工作, 以保证其安全性。如果需要频繁的存取, 为了减小函数调用开销, 保证性能, 则可通过宏来封装对全局资源的存取。

### 7.2.2 条款2: 编写可重入函数时, 若使用全局变量, 可通过互斥量、信号量 (即P、V操作) 等手段对其加以保护

当我们的软件存在对全局变量的操作时, 我们固然需要如此保护; 但同时我们也应看到, 通过互斥量、信号量的一些不可避免副作用, 即性能问题, 处理逻辑的复杂化等等。因此, 我们应当减少不是必需的全局变量。

## 7.3 程序块结构

C语言允许在以左括号开始的程序块中定义变量，以这种方式定义的变量可以与程序块外的变量具有相同的名字。这些变量在程序执行到定义它们的程序块以外时便不复存在。

### 7.3.1 条款1：程序块内变量的取名避免和程序块外的变量一样

虽然在名字一样的情况下，程序块内的代码真正引用的是在程序块内定义的变量，但是同名会造成阅读上的困惑，应该尽量避免出现这种情况。

## 7.4 递归

### 7.4.1 条款1：尽量减少函数本身或函数间的递归调用

尽管递归在描述数据结构等问题是非常方便的，代码实现也非常简洁，但有时候递归调用特别是函数间的递归调用（如A->B->C->A），也会影响程序的可理解性；同时，递归调用一般都占用较多的系统资源（如栈空间），过深的递归将导致栈溢出。若要编写商业软件，除了下面两种情况，应减少递归调用：

- （1）某些必须用递归实现的算法（如汉诺塔问题的处理）。
- （2）递归确实能够简化问题，同时递归的深度是可控制的。

在其它场合中，一旦规划出递归算法之后，可以使用堆栈来保存递归调用的临时信息，模拟实现递归调用，而不是通过函数直接递归调用。

## 7.5 预处理器

“Preprocess, to do preliminary processing of (as data).”

Webster

### 7.5.1 宏

#### 7.5.1.1 条款1：所有宏定义采用大写字母，单词与单词之间以下划线相连

正确示例：

```
/* zos alloc memory of specific type */
#define ZOS_ABNF_ALLOC_ELEM(_membuf, _type) \
    (_type *)ZOS_ABNF_ALLOC(_membuf, sizeof(_type))
```

#### 7.5.1.2 条款2：宏中的操作符必须以下划线\_开始，然后全部字母小写

正确示例：

```
/* zos alloc memory of specific type */
#define ZOS_ABNF_ALLOC_ELEM(_membuf, _type) \
    (_type *)ZOS_ABNF_ALLOC(_membuf, sizeof(_type))
```

#### 7.5.1.3 条款3：存在操作符的宏定义必须加括号()

错误示例：

```
#define ZOS_SLIST_HEAD_NODE(_slist) _slist->pstHead
#define ZOS_SLIST_TAIL_NODE(_slist) _slist->pstTail
```

**正确示例:**

```
#define ZOS_SLIST_HEAD_NODE(_slist) (_slist)->pstHead
#define ZOS_SLIST_TAIL_NODE(_slist) (_slist)->pstTail
```

#### 7.5.1.4 条款4: 使用操作符的宏定义, 宏名称与括号之间不能有空格

**错误示例:**

```
#define ZOS_SLIST_HEAD_NODE (_slist) _slist->pstHead
```

此时ZOS\_SLIST\_HEAD\_NODE (&m\_stZosLogMgrLst) 会被预处理成:

```
(_slist) _slist->pstHead (&m_stZosLogMgrLst)
```

上列代码编译时便出错。

**正确示例:**

```
#define ZOS_SLIST_HEAD_NODE(_slist) (_slist)->pstHead
```

此时ZOS\_SLIST\_HEAD\_NODE (&m\_stZosLogMgrLst) 会被预处理成:

```
(&m_stZosLogMgrLst)->pstHead
```

#### 7.5.1.5 条款5: 宏所定义的多条表达式应放在do {} while (0) 或大括号中

当宏中有多条表达式时, 一般用大括号对扩成复合语句, 如下:

```
/* zos alloc memory of one abnf list node */
#define ZOS_ABNF_ALLOC_LIST_DATA(_membuf, _type, _data) { \
    _data = (_type *)ZOS_ABNF_ALLOC(_membuf, sizeof(_type)\
        + sizeof(ST_ZOS_SLIST_NODE)); \
    if (_data) \
    { \
        ((ST_ZOS_SLIST_NODE *)_data)->pstNext = ZNULL; \
        ((ST_ZOS_SLIST_NODE *)_data)->pData = (ZCHAR *)_data \
            + sizeof(ST_ZOS_SLIST_NODE); \
        _data = (_type *)((ZCHAR *)_data + sizeof(ST_ZOS_SLIST_NODE)); \
    } \
}
```

这种写法在遇到if...else时会遇到问题, 如:

```
if (...)
    ZOS_ABNF_ALLOC_LIST_DATA(pstAbnfMsg->pstMemBuf, ST_SDP_ANCMT, pstAncmt);
else
    ...;
```

会被预处理成:

```

if (...)
{
    _data = (_type *)ZOS_ABNF_ALLOC(_membuf, sizeof(_type)
        + sizeof(ST_ZOS_SLIST_NODE));

    if (_data)
    {
        ((ST_ZOS_SLIST_NODE *)_data)->pstNext = ZNULL;
        ((ST_ZOS_SLIST_NODE *)_data)->pData = (ZCHAR *)_data
            + sizeof(ST_ZOS_SLIST_NODE);
        _data = (_type *)((ZCHAR *)_data + sizeof(ST_ZOS_SLIST_NODE));
    }
};
else
    ...;

```

此时发生编译错误，参考[If-Else语法](#)。在大括号外加上do...while(0)的技巧可以解决这个问题。另外通过规范if...else、do...while、for语句的写法可以避免出现这种问题，参考[4.2.3 条款3](#)。

#### 正确示例:

```

/* zos alloc memory of one abnf list node */
#define ZOS_ABNF_ALLOC_LIST_DATA(_membuf, _type, _data) do { \
    _data = (_type *)ZOS_ABNF_ALLOC(_membuf, sizeof(_type)\
        + sizeof(ST_ZOS_SLIST_NODE)); \

    if (_data) \
    { \
        ((ST_ZOS_SLIST_NODE *)_data)->pstNext = ZNULL; \
        ((ST_ZOS_SLIST_NODE *)_data)->pData = (ZCHAR *)_data \
            + sizeof(ST_ZOS_SLIST_NODE); \
        _data = (_type *)((ZCHAR *)_data + sizeof(ST_ZOS_SLIST_NODE)); \
    } \
} while (0)

```

### 7.5.1.6 条款6: 使用宏时，不允许参数发生变化

#### 错误示例:

```

#define SQUARE(_a) ((_a) * (_a))

ZINT a = 5;
ZINT b;
b = SQUARE(a++);

```

#### 正确示例:

```
b = SQUARE(a);  
a++;
```

**7.5.1.7 条款7：**每一个#*endif*都要有注释，除非#*if*和#*endif*之间的代码只有数行

**7.5.1.8 条款8：**在宏中应该避免使用全局变量

当宏在函数中被替换时，有可能发生全局变量和函数内的局部变量同名的情况，这种情况可以通过规范变量命名来避免。

**7.5.1.9 条款9：**避免宏本身或宏之间的递归调用

虽然大多数编译器支持对于递归宏的预处理，但是宏本身或者宏之间的递归调用大大增加了预处理逻辑的复杂度，编译器进行宏代码替换的结果很有可能不是作者预期的效果。事实上，绝大多数的宏递归是可以通过定义第三个宏等方式消除的。

## 7.5.2 条件预编译

条件预编译类似if...else。但是if...else在运行时处理而条件预编译在编译前处理。条件预处理一般应用于如下场合。

目标代码需要向多个平台移植。

目标代码需要产生debug和release版本。

头文件嵌套包含处理。

通过#*if* 0...#*endif*来进行代码屏蔽。

条件预处理的关键标识为5个：

```
#ifdef / #ifndef
```

```
#if
```

```
#defined
```

```
#else
```

```
#elif
```

以#*if*，#*ifdef*或#*ifndef*作为开头，以#*endif*结束。

**7.5.2.1 条款1：**尽量使用#*if*来代替#*ifdef*和#*ifndef*

#*if*除了可以判断宏是否被定义，还可以支持宏表达式，因此建议使用#*if*来代替#*ifdef*和#*ifndef*，事实上，#*ifdef*等价于#*if* defined()，#*ifndef*等价于#*if* !defined()

**7.5.2.2 条款2：**使用#*error*在某种条件下中止编译

条件预编译和#*error*的结合使用可以使得在满足某种条件时中止编译。

正确示例：

```
#if ZOS_PLATFORM == ZOS_PLATFORM_WIN32
#define ZTASK_PRIORITY_MAX      (2)
#define ZTASK_PRIORITY_NORMAL   (0)
#define ZTASK_PRIORITY_MIN      (-2)
#define ZTASK_PRIORITY_INCREMENT (+1)
#else
#error "Unsupported os"
#endif
```

### 7.5.2.3 条款3：使用#warning在某种条件下给出编译告警

#warning用法与#error相似，与#error不同的是，当满足条件时，#warning只给出告警，不中止编译。一般在已经废弃的（但还需要保留的）头文件中使用#warning，告诉用户应该使用正确的头文件版本。

### 7.5.2.4 条款4：不建议使用#pragma

#pragma与条件预编译的结合用于实现在特定编译器下对编程语言进行扩充，#pragma的实现依赖于特定的编译器。因此不建议使用#pragma。

## 8. 指针和数组

“Pointer, is a variable that contains the address of a variable.”

K&R C

“Array, to set or place in order.”

Webster

### 8.1 指针和内存使用

#### 8.1.1 条款1：要进行边界检查，防止内存操作越界

内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界经常发生在使用数组时的下标“多1”或者“少1”操作。在for循环语句中，循环次数很容易搞错，也会导致数组操作越界。

内存操作越界是软件系统主要错误之一，后果往往非常严重，所以进行此类操作时一定要仔仔细心。

#### 8.1.2 条款2：严禁使用未经初始化的变量作为右值

把未经初始化的变量作为右值是非常危险的，特别是指针变量。如果是指针变量做右值，则有可能错误的修改程序空间中任意位置的数据或导致程序崩溃。

错误示例：

```
ZCHAR *p;
ZCHAR *pcStr = p;    /* now the p and pcStr point to unexpected address */
...
*pcStr = 'a';
```

#### 8.1.3 条款3：任何一次调用申请内存后要检查是否申请成功

正确示例：

```
pstNode = Zos_HeapAlloc(ZOS_BKT_NODE_SIZE(iSize));
if (pstNode == ZNULL)
{
    ZOS_LOG_ERROR(LOG_MGR_ZOS, "bkt heap alloc fail.");
    return ZNULL;
}
```

#### 8.1.4 条款4：管理好你的内存

我们可以看到许多编程的建议，比如，“任何一次调用申请或释放(malloc或free)前都要检查内存指针”，“释放内存后要将指向该内存的指针置为空”，“指针变量在定义的同时要正确初始化，要么让指针指向一个有意义的对象，要么把指针赋值为NULL”。这些都是有益的忠告，但是如果把这些放到我们的编码风格中去却不太合适。因为这些建议的最终目的是要程序员管理好内存，避免内存泄露，避免错误的释放内存，这些是解决问题的方法，而不是必须遵守的规则，也许还存在更好的管理内存的方法呢。所以我们只能说，要管理好你的内存。我们有下列建议：

- (1) 统一规划内存的分配和释放，对于内存的管理，要一切在控制中。
- (2) 指针变量在定义的同时要正确初始化，要么让指针指向一个有意义的对象，要么把指针赋值为NULL。
- (3) 除了内存管理模块，建议不要把本模块分配的内存交由其他模块来释放。

第(2)和第(3)点容易理解，这里解释一下第(1)点。

统一规划内存的分配和释放，就是对于内存的操作是有一套固定的规则或者处理方法的。比如：

#### 方案一、利用NULL进行控制

方法就是，在指针变量free之前检查其值是否为NULL，在指针变量free之后，把指针赋值为NULL。在简单的情况下，这能够在一定程度上保证内存释放不出问题。但是如果两个指针同时指向一个地址的话，这种做法就会出问题了。所以这个方案只能应付比较简单的应用情况了。

错误示例：

```
ZCHAR *p1 = malloc(100);
ZCHAR *p2 = p1;
if (ZNULL != p2)
{
    free(p2);
    p2 = ZNULL;
}
if (ZNULL != p1)
{
    free(p1);
    p1 = ZNULL;
}
```

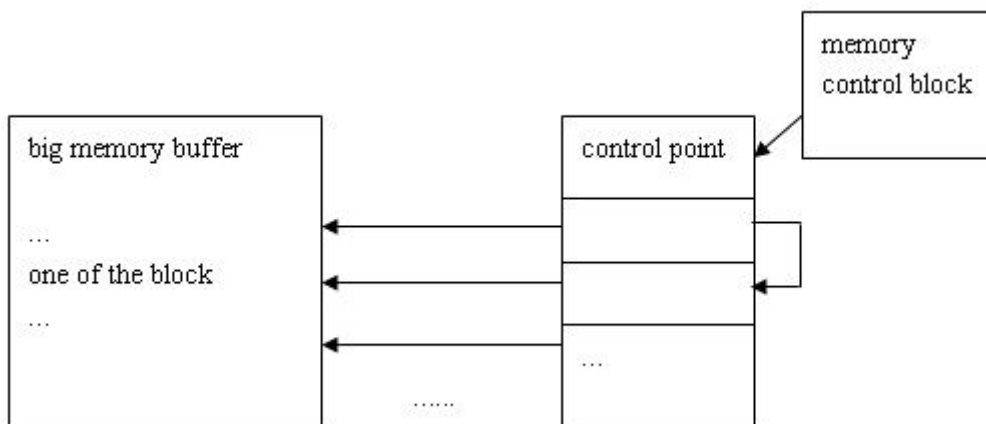
#### 方案二、利用智能指针

在C++世界中，有各种各样的智能指针(包括应用的不怎么好的标准库智能指针std::auto\_ptr，也有比较优秀的Boost的智能指针和更优秀的Loki智能指针，甚至还有应用于COM的智能指针)，以提供避免内存泄露的方案，同时提高内存存取效能，有的智能指针还支持在多任务环境下工作。这些智能指针的实现都使用了C++的模板技术，超出了本文的讨论范围。我们不建议使用c语言去构建智能指针，因为困难重重。但如果你对智能指针非常了解，当然可以尝试利用智能指针的原理在C语言的世界里规划一番。

#### 方案三、利用内存控制块

对于需要全局存储的内存，可利用内存控制块来进行统一管理。使用内存控制块作为规划内存管理的方案的好处是，安全性高，申请和释放的效率高。最简单的例子就是固定数量和大小的控制块，实现方法是，事先定义一个静态的数组，或者从堆当中申请一块大的内存，通

过规划好的内存控制点进行控制，如下图所示。对于特定场合的内存的应用管理是可以这样处理的。



关于内存控制块的结构定义大概可以如此：

```
typedef struct tagMEM_CTRL_POINT
{
    struct tagMEM_CTRL_POINT *pstNext; /* next memory control block */
    ZULONG *pBlockAddr; /* point to the address in the memory block's */
} ST_MEM_CTRL_POINT;

typedef struct tagMEM_CTRL_BLOCK
{
    ZMUTEX zMutex; /* mutex */
    ZUCHAR ucCount; /* control points count */
    ZUCHAR aucSpare[3]; /* for 32 bit alignment */
    struct tagMEM_CTRL_POINT *pstCurFreePoint; /* current free memory block */
    ZINT iMemBlockSize; /* one of the memory block size */
} ST_MEM_CTRL_BLOCK;
```

初始化时，

首先，分配好big memory buffer，大小为 ucCount \* iMemBlockSize。

其次，分配好个数为ucCount个内存控制点（tagMEM\_CONTROL\_POINT），为了加快访问内存的速度，直接为每个内存控制点的地址pBlockAddr赋值，并且将free的内存控制点串起来（一开始都是free的）。

最后，将内存控制块的iCurFreePoint赋值为第一个free的内存控制点的指针。

申请内存时，将pstCurFreePoint指向的内存地址给用户，而下一个free的内存控制点作为当前的free内存控制点。

释放内存时，将已经释放的内存控制点的地址赋值给pstCurFreePoint，而其下一个free的内存控制点重新设置为原来的pstCurFreePoint。

这些操作，都需要互斥操作来保证同步，所以要加一个zMutex。

注意，为了保证问题的简洁性，这个例子没有对边界保护等健壮性需求进行考虑。

如果需要更加灵活的应用，可以设计更加复杂的内存管理模块，比如可以实现动态扩充、分配不同大小等功能。当然这就需要稍微精巧一些的设计了。

而对于局部的存储空间，如果需要的空间不大的话，可使用数组，利用函数的栈来自动分配及释放；如果需要空间较大的话可以在堆中分配，并记住一定在函数出口之前释放内存。

### 8.1.5 条款5：建议谨慎使用memset、memcpy等大量存在内存拷贝的函数，如果大量使用会显著降低程序效率

在写程序的时候，经常会对未初始化的内存调用memset来初始化，但需要注意的是，并不是所有未初始化的内存空间都需要清0，如通过空间长度来控制实际使用的空间等措施就能避免内存拷贝（清0实际上也是一种拷贝）。谨慎使用此类函数也能使程序员更加关注内存空间边界，提高代码的安全性。

### 8.1.6 条款6：不推荐过多地使用指针

(1) 在分配内存空间时，能够用数组的地方尽量使用数组，这有助于提高系统安全性和稳定性。因为在函数体内部定义的数组空间位于函数堆栈中，在函数退出时随着堆栈释放而释放，基本上不存在内存泄漏问题。而在函数体外部定义的数组，即全局数组，其空间位于系统的静态数据区，静态数据区的大小固定，在任务退出时空间被释放，因此使用数组是安全的。

(2) 在传递地址时，可以按需要使用指针，不必刻意避免。

### 8.1.7 条款7：不推荐在模块中频繁的使用动态内存分配释放操作

频繁的进行malloc和free操作将会使程序运行效率低下并且不易于管理，如果能够在本模块做到单独的内存管理，将使程序更易管理并有助于减少系统内存碎片的产生。

### 8.1.8 条款8：避免使用已经释放的内存

使用已经释放的内存一般发生在以下三种情况：

(1) 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存。此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

(2) 函数返回了指向堆栈的指针。

(3) 使用free或删除释放了内存后，没有将指针设置为NULL，导致下次使用时判断失误。

## 8.2 指针和函数参数

### 8.2.1 条款1：当函数参数为数组时，建议写成同类型指针变量的形式

在C语言中，当传递一个数组参数给函数时，实际上是把指向数组第一个元素的指针赋给了函数内的一个局部指针变量，因此在声明函数时，参数写成数组和写成同类型指针是完全等价的，但是写成指针的形式因为符合实际情况而更直观，更不容易出错。

**错误示例：**

```
ZVOID Foo(ZCHAR acStr[100])
{
    ...
}
```

**正确示例:**

```
ZVOID Foo(ZCHAR *pcStr)
{
    ...
}
```

## 8.3 指针和数组

在C语言里，指针和数组有着非常紧密的联系，当我们这样定义了一个数组和一个指针：

```
ZCHAR acStr[10]
ZHCAR *pcStr = acStr;
```

我们可以像使用数组名一样使用指针：`pcStr [2]`，也可以像使用指针一样使用数组名：`*(acStr + 2)`，但是在指针和数组名之间存在一个差别，指针是一个变量而数组名不是，因此`acStr = 'c'`和`acStr++`的操作都是不合法的。

### 8.3.1 条款1：不要试图修改常量字符串

看如下示例：

```
ZCHAR *pcMsg = "now is the time";
```

`pcMsg`指针变量被初始化成指向一个字符串常量，该字符串常量存储于只读变量区中，当试图通过`pcMsg [2] = 'x'`修改字符串时，结果是未定义的。

## 8.4 地址运算

### 8.4.1 条款1：注意地址运算的有效性

指针变量的运算只有在属于下面的几种情况时才是有效的：

同一种指针之间的赋值操作；

指针变量和整形之间的加减运算；

指向同一个数组成员的指针变量之间的比较运算或减运算；

用零值进行赋值或与零值比较；

其他所有的指针变量运算都是非法的，包括两个指针之间的加乘除运算、与浮点数的运算等。

`void`指针是个例外，`void`指针和其他类型指针之间的互相赋值是合法的。

## 9. 结构体类型

“Structure, is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.”

K&R C

ANSI标准对C语言所作的主要修改之一是定义了结构体的赋值，结构体能够被拷贝、赋值、作为参数传递给函数和作为函数的返回值。事实上这些特性很早就被大多数编译器所支持，但是ANSI标准对它们做了精确定义。

ANSI C 还规定了自动的结构体和数组可以被初始化。

### 9.1 结构体基础

#### 9.1.1 条款1：结构体声明

对于结构体声明，每个成员变量必须单独成行，并且其后跟一个注释来描述它。推荐使用typedef的方式来声明结构体。

正确示例：

```
typedef struct tagSDP_TKN_MGR
{
    ZULONG dwTknType;           /* token type */
    ST_SDP_TKN_TABLE *pstTable; /* token table */
    ZULONG dwTableSize;        /* token table size */
} ST_SDP_TKN_MGR;
```

#### 9.1.2 条款2：结构体成员要4字节对齐

数据对齐的问题由来已久，在一些机器上，一个整形变量可以从任何地址开始，而在另一些机器上，只能从偶数地址开始，甚至只能从4的倍数的地址开始。因此在一个结构体中，相同的成员变量在不同机器上的地址偏移很有可能是不一样的。不同类型的指针是不能自由互换的，通过一个指向奇地址的指针存储一个整形数值，有时候能成功，有时候会造成core dump，有时候是没有任何提示的操作失败，这些都是跟机器相关的。

通过在声明结构体时人为的进行成员变量对齐和规范结构体成员变量的引用方式，可以很大程度上减少不同机器上的数据对齐方式对编程的困扰。

尽管有些编译器具有自动结构体成员4字节对齐的功能选项，但是为了统一代码风格，不希望在大部分代码已经人工字节对齐的情况下，却存在个别现象，因此本条款为强制执行。

需要说明的是，为了说明本文中的某些规则，一些结构体没有采用字节对齐的方式定义，只是为了简化描述而已。

**正确示例：**

```
typedef struct tagSIP_PARM
{
    ZUCHAR ucPres;           /* present flag */
    ZUCHAR ucValPres;       /* pvalue present flag */
    ZUCHAR aucSpare[2];     /* for 32 bit alignment */
    ST_ZOS_SSTR stPname;    /* pname */
    ST_ZOS_SSTR stPval;     /* pvalue */
} ST_SIP_PARM;
```

## 9.2 结构体数组

### 9.2.1 条款1: 结构体数组的初始化

用初始化子 (initializer) 对结构体数组进行初始化时应使用嵌套括号的形式, 属于同一个数组成员的初始化子被括在同一个大括号中:

错误示例:

```
static ST_SIP_TKN_TABLE m_astSipTknReqUriType[] =
{
    EN_SIP_REQ_URI_SIP, "sip", 3,
    EN_SIP_REQ_URI_SIPS, "sips", 4,
    EN_SIP_REQ_URI_IM, "im", 2,
    EN_SIP_REQ_URI_TEL, "tel", 3
};
```

正确示例:

```
static ST_SIP_TKN_TABLE m_astSipTknReqUriType[] =
{
    {EN_SIP_REQ_URI_SIP, "sip", 3},
    {EN_SIP_REQ_URI_SIPS, "sips", 4},
    {EN_SIP_REQ_URI_IM, "im", 2},
    {EN_SIP_REQ_URI_TEL, "tel", 3}
};
```

### 9.2.2 条款2: 结构体数组大小的计算

建议使用如下宏计算结构体数组大小, 相比分母使用sizeof(结构体类型)的写法, 这种方式的好处是结构体数组大小的计算可以独立于结构体类型。

正确示例:

```
#define ZOS_GET_TABLE_SIZE(_table) (sizeof(_table) / sizeof(_table[0]))
```

## 9.3 Typedef 类型定义

### 9.3.1 条款1: 建议使用typedef定义变量类型

使用宏来定义变量类型和使用typedef来定义变量类型看起来是等价的, 但是typedef具有更好的灵活性和可移植性, 看下面用宏定义类型的例子:

```
#define T1 struct foo *  
T1 a, b;
```

看起来好像定义了两个指向struct foo的指针，但是事实上只有a是指针，因为T1 a, b在经过宏替换后变成了struct foo \* a, b，用typedef定义类型不会有这个问题。

示例：

```
typedef struct foo *T1;
```

## 9.4 位域

位域被C语言提供用来进行比特位的直接存取操作（“直接”相对于位逻辑运算符而言），一个位域是一组彼此邻接的，在同一个存储单元（位域字）之内的比特位的集合，位域字的大小依赖于具体实现。

对位域的最大争议是位域的实现依赖性，这些依赖于具体实现的特性包括一个位域是否能跨越位域字的边界、位域的分配是从左到右还是从右到左、以及上面提到的位域字的大小。

### 9.4.1 条款1：不要使用位域进行位操作，建议使用位逻辑运算符

位域的实现依赖性降低了代码的可移植性，建议用位逻辑运算符进行位操作。

示例：

```
/* 在flags中设置EXTERNAL和STATIC位 */  
flags |= EXTERNAL | STATIC;  
/* 在flags中清除EXTERNAL和STATIC位 */  
flags &= ~(EXTERNAL | STATIC);  
/* flags中是否同时设置了EXTERNAL和STATIC位 */  
if ((flags & (EXTERNAL | STATIC)) == 0)
```

## 10. 库函数

使用库函数之前最好先搞清楚库函数的实现原理，从而比较深刻的理解库函数接口的含义。C语言的库函数其实是比较短小精悍的，不象C++那么庞大复杂，但也有一些细节需要我们注意。

### 10.1 条款1：不要在程序中声明系统函数

不要试图自己声明系统函数，要通过include系统头文件来使用系统函数。要调用系统库函数，虽然说自己声明其系统函数往往是可以工作的，但是实在没有什么理由不使用系统提供的头文件而要自己声明一套的，为了减少失误，少浪费时间，建议使用系统提供的头文件。

### 10.2 条款2：不要使用sprintf的返回值

在大部分的Unix操作系统中，如BSD，`sprintf(string, fmt, ...)`返回的是指向string的指针（`char*`），而ANSI C要求`sprintf`返回一个表示成功写入string的字符数的整形，Linux下的`sprintf`即遵循此标准。因此出于移植性的考虑，不要在程序中使用`sprintf`的返回值。

### 10.3 条款3：getchar返回一个整数，而不是一个字符

我们来看下面的程序：

```
#include <stdio.h>

ZVOID main()
{
    ZCHAR c;
    while ( (c = getchar()) != EOF )
    {
        putchar( c );
    }
}
```

看起来这段程序应该把它的标准输入拷贝到输出，实际运行情况并非如此。原因在于 `c` 声明为 `char` 型而非 `int` 型，这就意味着 `c` 并不能捕捉到每一次EOF。所以，有两种可能。或者 `c` 可能在正常输入情况下捕捉到EOF，或者 `c` 根本不可能捕捉到EOF。前一种情况，程序将中途停止拷贝。后一种情况，程序将进入死循环。实际上，如果编译器不是太严格的遵守C99的话，程序将随机运行。

### 10.4 条款4：小心使用printf、sprintf、fprintf

在C语言中，`printf`（包括`sprintf`、`fprintf`）库函数是一个使用率非常高的函数。`printf`比较充分的考虑灵活性，却不是十分安全，下面是几个要注意的地方。

(1)注意`printf(aucString)`跟`printf(“%s”, aucString)`的区别(其中`aucString`是一个字符串变量)。

考虑一下如果`aucString`有`'%'`的情况吧，因为`printf`的第一个参数是“`const char *format`”，从语义上来说就是字符串格式，如果这个`format`参数中含有`'%'`，`printf`函数就会在后续的参数中寻找匹配的参数。如果不能确定`aucString`的具体内容，使用`printf(aucString)`的结果也是未

定义的。所以，要输出一个字符串，我们不能将“字符串格式”等价于字符串本身。正确的用法是printf(“%s”, aucString)。

(2) printf在多任务下是缺乏控制的。

printf默认的输出资源就是屏幕控制台，如果多个任务同时调用printf，打印到屏幕的结果便会交错的出现各种字符。为了达到有序输出的目的，一种做法就是每个任务都输出到统一管理的缓冲区，然后请printf自己从缓冲区取数据打印。

## 11. 编程惯例

软件工程师的编程中会考虑诸多原则，如可读性、效率等，这些原则有时候会互相矛盾的，这时就需要给出各个原则的的优先级别了，就像没有绝对的真理一样，没有绝对正确的优先级，我们推荐如下优先原则，以作参考：

- 正确性，指程序要实现设计要求的功能
- 规范化，指程序书写风格、命名规则等要符合规范
- 可读性，程序代码必须具有良好的可读性
- 稳定性，指程序能稳定和长时间的运行
- 安全性，指程序能可靠和安全的运行
- 可移植性，程序能在在多种系统环境良好和快速的被移植
- 可维护性，用户能方便查错和解决问题
- 可测试性，指程序要具有良好的可测试性，如丰富的测试用例，自动测试等
- 全局效率，整体上，程序具有比较高的时间和空间效率
- 局部效率，局部范围内，程序具有比较高的时间和空间效率
- 复用性，程序要具有良好的复用设计

在IEEE的 Software Quality 定义了相关术语，参见附录八 [IEEE Software Quality Glossary](#)。

Webster

### 11.1 条款1：建议函数长度不要超过200行，否则可能会增加阅读的复杂性，而且也容易出错

尽量写一些简短的函数，这也常常使得代码易于阅读和理解，而且也提高了代码的重用性和可维护性。

### 11.2 条款2：建议避免过深(不要超过8重)的嵌套代码

如#ifdef、#else、#endif嵌套过深

错误示例：

```
#ifdef M1
#else
#ifdef M2
#else
...
#else
#ifdef M12
#else
#endif
```

### 11.3 条款3：在定义多个相关名称的时候，注意名称意义上的对称

如create的反义是destroy， malloc的反义是free等等

正确示例：

```
/* zos memory malloc */
ZVOID * Zos_Malloc(ZSIZE_T zSize);

/* zos memory free */
ZVOID Zos_Free(ZVOID *ptr);
```

#### 11.4 条款4：建议要了解CPU类型方面的差异性，避免不正确的假设

如字节序、字节对齐、整型范围、浮点型精度、类型转换方式等。

#### 11.5 条款5：所有代码要符合ANSI C99标准

ANSI C99是一个相对成熟的C语言标准版本，大多数主流的编译器是对它支持的。作为一个C语言特性基准，我们必须明确出来，并坚持之。

#### 11.6 条款6：尽量少定义占用太大空间的局部变量，同时要确认是否会造成堆栈溢出

#### 11.7 条款7：小心使用sizeof来得到数据类型或空间的大小

sizeof是C语言的一个关键字，如果不小心使用，很容易出问题。下面就是关于sizeof使用规则介绍：

- (1) sizeof作用于某数据类型或数据类型的实例时，将返回其字节数。
- (2) sizeof作用于任意类型的指针变量，则返回指针本身的大小，在32位系统中返回4个字节。
- (3) sizeof作用于某数据类型的数组之时，将返回数组所占的字节数，注意不是返回数组内元素的个数。
- (4) sizeof作用于函数数组参数时，则等价于作用于指针，在32位系统中返回4个字节。相关介绍见“[指针和函数参数之条款1](#)”。
- (5) sizeof作用于字符串常量之时，注意跟strlen区分，前者返回的是字符串缓冲区的整个长度，后者返回'\0'之前的字符个数，前者应该比后者大1。

错误示例：

```
ST_ZOS_ABNF_MSG *pstAbnfMsg;

sizeof(pstAbnfMsg) /* 获取ST_ZOS_ABNF_MSG的大小 */

ST_TKN_HDR_TABLE m_astSipTknHdrTable[] =
{
    ...
};

sizeof(*m_astSipTknHdrTable) /* 获取m_astSipTknHdrTable表的大小 */
sizeof(m_astSipTknHdrTable) /* 获取ST_TKN_HDR_TABLE的大小 */
```

正确示例：

```

ST_ZOS_ABNF_MSG *pstAbnfMsg;

sizeof(*pstAbnfMsg) /* 获取ST_ZOS_ABNF_MSG的大小 */

ST_TKN_HDR_TABLE m_astSipTknHdrTable[] =
{ ...
};

sizeof(m_astSipTknHdrTable) /* 获取m_astSipTknHdrTable表的大小 */
sizeof(m_astSipTknHdrTable[0]) /* 获取ST_TKN_HDR_TABLE的大小 */

```

## 11.8 条款8：养成良好的字符串使用习惯

一般来说，字符串应该是以'\0'作为结束，因为很多字符处理函数(如strcpy, strcmp等)常常需要用'\0'判断字符串是否终结，良好的字符串使用习惯，能够避免无谓的memset等函数操作，而且能够提高程序的安全和稳定性

## 11.9 条款9：对大块内存的初始化代码，使用循环展开的方式进行优化

假设a是条件判断指令的时钟周期（CMP），b是内存赋值指令的时钟周期（MOV），c是条件变量累加指令的时钟周期（DEC），N是循环次数，p是展开重数，那么经过优化后减少的执行时间是  $(a+b+c)*N - ((a+c)*N/p + b*N) = (1-1/p)*(a+c)*N$ ，优化带来的性能提升还是相当可观的。

### 优化前：

```

for (i=0; i<N; i++)
{
    aucBuf[i] = 'a';
}

```

### 优化后：展开重数为8

```

for (i=0; i<N; i += 8)
{
    aucBuf[i] = 'a';
    aucBuf[i+1] = 'a';
    aucBuf[i+2] = 'a';
    aucBuf[i+3] = 'a';
    aucBuf[i+4] = 'a';
    aucBuf[i+5] = 'a';
    aucBuf[i+6] = 'a';
    aucBuf[i+7] = 'a';
}

```

## 11.10 条款10：注意比较同一函数在不同编译环境和系统运行环境的效率差异

如strcpy、memcpy等函数在不同的操作系统中具有不同的效率。比较这些很关键，尤其用户自定义了库类似函数，如Zos\_MemCpy，其作用等价于memcpy，但性能是存在差异的，收集此类数据对于提高效率很有必要，尤其此类函数被频繁调用的时候。

### 11.11 条款11：程序性能优化的时机

程序的性能优化只有在存在系统执行瓶颈时，而且不会带来太多的开发开销，综合考虑其他因素，才去做代码优化。做优化前应该对代码执行性做出分析，根据80-20原则，找出最重要的问题代码

### 11.12 条款12：用for (;;)代替while (1)

对于无限循环，推荐使用for (;);，相对于while (1)， for (;;)的指令少，不占用寄存器而且没有判断跳转。

### 11.13 条款13：系统设计应该保证概念完整性

11.14 条款14：设计者应该对系统体系结构有比较清晰地感觉，能清楚的以语言、文字、图形等方式向团队表达，从中发现抽象模型和通用机制，目的是使得系统更加简单、轻巧和可靠

11.15 条款15：设计的一个重要原则是简单。简单的系统设计易于理解和维护，简单设计是建立在艰苦的时间思考和不断的反复修改后达到的，简单并不意味着草率和粗糙

11.16 条款16：设计时应该注意扩展性，良好的扩展性往往建立在清晰的系统设计和通用的抽象设计上。良好的扩展性能给系统带来更好的适应能力

### 11.17 条款17：理解机制与策略的不同

“Mechanism, a process, technique, or system for achieving a result.”

“Strategy, a careful plan or method.”

Webster

机制是指做一件事的目标、方向、过程、技术或系统，策略是指做这件事的具体方法或计划。在很多系统设计，机制是相同的，策略却根据应用不同差别很大。比如，ZOS提供了对数据缓冲的区管理的机制，但如何基于缓冲区管理机制，使上层应用得以实现（如协议解码，协议事务处理等）却是可以有多种策略的。因此，程序设计的时候，在某种机制下如何设计或选择一个理想的策略是非常重要的。

### 11.18 条款18：使用相同的编辑器，并使用相同的选项设置

同一项目组最好采用相同的IDE编辑器，如Souce Insight、Visual Studio、UltraEdit、Notepad等，并设计、使用一套缩进宏及注释宏等，将缩进等问题交由编辑器处理。

**11.19 条款19:** 在软件产品（项目组）中，要统一编译开关选项

**11.20 条款20:** 建议首先在平台和标准库中查找符合所需功能的函数，避免随意添加冗余功能的函数

**11.21 条款21:** 不要引用Unix和其他存在版权问题的代码，即使程序本身不作为公司商业软件的一部分

**11.22 条款22:** 谨慎使用开放源代码

建议尽量避免使用开放源代码中的程序片断，如果处于必要性(如时间紧迫性、实现难度等)，要对代码来源进行说明，并按照代码所有者的要求保留其版权声明。

**11.23 条款23:** 建议使用Rational Purify工具检查和解决内存越界、泄露等运行问题

**11.24 条款24:** 建议使用Rational PureCoverage工具来统计程序代码覆盖率

**11.25 条款25:** 建议使用Rational Quantity工具来分析程序的执行效率

**11.26 条款26:** 编程存在疑问时，推荐先翻阅C Faq，然后再通过其它方式寻求答案

<http://www.faqs.org/faqs/C-faq/> 或 <http://www.eskimo.com/~scs/C-faq/top.html>

## 12. 参考书目

- [1] 《The C Programming Language》, Brian W. Kernighan and Dennis M. Ritchie.
- [2] 《The Practice of Programming》, Brian W. Kernighan, Rob Pike, Addison-Wesley 1999
- [3] 《Coding Standards And Guidelines For Good Software Engineer Practice In C++》, Kosmas Karadimitriou, 2001
- [4] 《IEEE Std 610.12-1990 Standard Glossary Of Software Engineering Terminology》
- [5] 《IEEE-ISO 9126-2001 Software Quality》
- [6] 《C++ Coding Guideline》, 林锐
- [7] 《Recommended C Style and Coding Standards》, L.W. Cannon, .etc
- [8] 《C Traps and Pitfalls》, Andrew Koenig

### 13. 附录一：前缀命名法

前缀	类型	描述
b	Bool	布尔整数
uc	Unsigned Char	无符号字符
c	Signed Char	有符号字符
w	Unsigned Short	无符号短整数
w	Signed Short	有符号短整数
i	Unsigned Integer	无符号整数
i	Signed Integer	有符号整数
dw	Unsigned Long	无符号长整数
dw	Signed Long	有符号长整数
p	Pointer	指针
p	Void Pointer	Void 指针
pdw	Long pointer	长指针
pc/puc	String pointer	字符串指针
a	Array	数组
ac/auc	String	字符串
ap	Array Pointer	数组指针
fn	Function	函数
pfn	Void Function Pointer	Void 函数指针
st	Structure	结构
en	Enum	枚举
b<n>	Bit Field	位域
g_	Global	全局的
m_	Module (Local File)	模块的
z	ZOS type	ZOS 平台相关的类型

注：前缀命名法类似于匈牙利命名法，但有所不同。了解更多的匈牙利命名法可阅读相关资料。

## 14. 附录二：匈牙利命名法

Charles Simonyi is credited with first discussing Hungarian Notation. It is a variable naming convention that includes C++ information about the variable in its name (such as data type, whether it is a reference variable or a constant variable, etc). Every company and programmer seems to have their own flavor of Hungarian Notation. The following is just what we thought seemed easy for beginning students to understand.

Prefix	Type	Example
b	boolean	bool bStillGoing;
c	character	char cLetterGrade;
str	C++ String	string strFirstName;
si	short integer	short siChairs;
i	integer	int iCars;
li	long integer	long liStars;
f	floating point	float fPercent;
d	double-precision floating point	double dMiles;
ld	long double-precision floating point	long double ldLightYears;
sz	Old-Style Null Terminated String	char szName[NAME_LEN];
if	Input File Stream	ifstream ifNameFile;
is	Input Stream	void fct(istream &isIn);
of	Output File Stream	ofstream ofNameFile;
os	Output Stream	void fct(ostream &rosIn);
S	declaring a struct	struct SPoint {
C	declaring a class	class CPerson {
struct name or abbrev	declaring an instance of a struct	SPoint pointLeft; SPoint ptLeft; // or abbrev. (be consistent)
class name or abbrev	declaring an instance of a class	CPerson personFound; CPerson perFound; // or abbrev. (be consistent)

The following table contains letters that go before the above prefixes.

Pre-prefix	Type	Example
u	unsigned	unsigned short usiStudents;

k	constant formal parameter	void fct(const long kliGalaxies)
r	reference formal parameter	void fct(long &liGalaxies)
s	static	static char scChoice;
rg	array (stands for range)	float rgfTemp[MAX_TEMP];
m_	member variable of a struct or class	char m_cLetterGrade;
p	pointer to a single thing	char *pcGrade;
prg	dynamically allocated array	char *prgcGrades;

### 15. 附录三：版权声明模版

/\*\*\*\*\*\*

(c) COPYRIGHT 2005-2010 by Juphoon System, Inc.

All rights reserved.

This software is confidential and proprietary to Juphoon System , Inc. No part of this software may be reproduced, stored, transmitted, disclosed or used in any form or by any means other than as expressly provided by the written license agreement between Juphoon and its licensee.

Juphoon warrants that for a period, as provided by the written license agreement between Juphoon and its licensee, this software will perform substantially to Juphoon specifications as published at the time of shipment and the media used for delivery of this software will be free from defects in materials and workmanship.

JUPHOON MAKES NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE WITH REGARD TO THIS SOFTWARE OR ANY RELATED MATERIALS.

IN NO EVENT SHALL JUPHOON BE LIABLE FOR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE USE OF, OR INABILITY TO USE, THIS SOFTWARE, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY, OR OTHERWISE, AND WHETHER OR NOT IT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Restricted Rights Legend

Juphoon Systems Software, Inc.

Email: support@juphoon.com

Web: http://www.juphoon.com

\*\*\*\*\*/

## 16. 附录四：源文件说明模版

```
/*  
File Name: XXX.X  
Module: XXXX  
Author: XXXXXX  
Version: 0.1  
Created on : 2003-11-XX  
Description:  
XXXXXXXXXXXXXXXXXXXXX  
  
Modify History:  
1. Date: Author: Modification:  
*/
```

## 17. 附录五：函数定义说明模版

```
/*  
Function name: xxxx  
Description:  
  XXXXXXXXXXXXXXXXXXXX  
  
Input parameter:  
  P1 XXXX  
  P2 XXXX  
  
Output parameter:  
  P3 XXXX  
  
Return value:  
  Returns ZOK on success, or ZFAILED on failure.  
*/
```

## 18. 附录六：单词常用缩略语

单词全称	缩略语	中文词义
Absolute	Abso	相对
Accept	Acpt	接受
Acknowledge	Ack	认可
Adapter	Adp	适配
Address	Addr	地址
Adjust	Adj	调整
Algorithm	Algo	算法
Alignment	Algn	对齐
Allocate	Alloc	分配
Anonymity	Anony	匿名
Application	App	应用
Argument	Arg	自变量
Attribute	Attr	属性
Authenticate	Authen	鉴别
Authorization	Author	授权
Bucket	Bkt	桶
Buffer	Buf	缓冲区
Character	Chr	字符
Check	Chk	检查
Clear	Clr	清除
Clock	Clk	时钟
Conference	Conf	会议
Compare	Cmp	比较
Config	Cfg	配置
Confirm	Cnf	确认
Connection	Conn	连接
Context	Ctx	上下文
Control	Ctrl	控制
Copy	Cpy	拷贝
Database	Db	数据库
Debug	Dbg	调试
Decrement	Dec	递减
Description	Desc	描述
Default	Dft	默认
Delete	Del	删除

单词全称	缩略语	中文词义
Deinitialize	Deinit	反初始化
Dependence	Dep	依赖
Disable	Dis	禁止
Display	Disp	显示
Disconnection	Disc	拆除连接
Element	Elem	元素
Enable	En	使能
Endpoint	Endp	端点
Environment	Env	环境
Error	Err	错误
Extend	Ext	扩展
Extension	Extn	扩展
Event	Evt	事件
Format	Fmt	格式
Function	Func	函数
Generic	Gen	通用, 一般
Group	Grp	组
Handling	Hdl	处理
Header	Hdr	头
Identifier	Id	标识
Increment	Inc	递增
Independence	Indep	独立
Indication	Ind	指示
Information	Info	信息
Initialize	Init	初始化
Instance	Inst	实例
Kernel	Ker	内核
Language	Lang	语言
Length	Len	长度
List	Lst	队列
Manager	Mgr	管理器
Manual	Man	手册
Maximum	Max	最大值
Memory	Mem	内存
Message	Msg	消息
Minimum	Min	最小值
Module	Mod	模块

单词全称	缩略语	中文词义
Multiplex	Multi	多重的
Network	Net	网络
Notify	Ntfy	通知
Number	Num	数量（号码时用全写）
Option	Opt	选项
Organization	Org	组织
Package	Pkg	包
Parameter	Parm	参数
Pointer	Ptr	指针
Present	Pres	存在
Previous	Prev	前
Primitive	Prm	原语
Priority	Prio	优先级
Private	Prv	私有
Process	Proc	处理
Property	Prop	属性
Purpose	Purps	目的
Read	Rd	读
Ready	Rdy	准备好的
Record	Rec	记录
Receive	Recv	接收
Register	Reg	注册
Reject	Rej	拒绝
Release	Rel	释放
Remove	Rem	删除
Request	Req	请求
Resource	Res	资源
Response	Rsp	响应
Return	Ret	返回
Schedule	Sched	调度
Script	Scrp	脚本
Seperator	Sepa	分隔符
Semaphore	Sem	信号
Sequence	Seq	序列
Session	Sess	会话
Socket	Sock	套接口
Space	Sp	空格

单词全称	缩略语	中文词义
Specification	Spec	规格
Statistics	Sts	统计
Status	Sta	状态
String	Str	字符串
System	Sys	系统
Subscription	Subs	订购
Success	Suss	成功
Summation	Sum	总合
Support	Supt	支持
Synchronize	Sync	同步
Timer	Tmr	定时器
Transaction	Trans	事务
Transport	Tpt	传输
Trigger	Trig	触发
Telephone	Tele	电话
Token	Tkn	标志
Utility	Util	工具
Value	Val	价值、值
Version	Ver	版本
Whitespace	Ws	空白符
Write	Wr	写

以上缩略语只是在实际编程中的参考写法，但具体情况具体分析，灵活处理，最主要的原则是以简洁和一致的命名方式表达正确而又清晰的语意，切不可在同一环境（如一个模块、同一产品）采用纷乱的命名形式，不仅影响了程序的美观度，而且干扰了读者的正常理解过程。

## 19. 附录七：常用文件组织和命名

在做系统设计和实现的时候，很多模块之间实际上存在很多的相似性，例如各种ABNF协议编解码本质上都差不多。然后，在我们把系统进行模块分解后，很多独立的功能单位会组织成不同的文件形式，而有些文件要实现多个功能之间的公共功能。因此，从文件组织上我们就能看出系统的主要功能和模块组织。

在平常的系统设计中，如概要设计，详细设计的时候，往往过度注重于结构定义，算法和过程描述，而忽略了作为主要产品形式的文件的组织和命名。从快速编程的角度来说，定义组织良好的、文字清晰、含义统一的文件将会提高开发速度，而且还能改善系统设计。有些人可能会不解，文件组织难道这么重要吗？

从设计上来说，定义文件组织，实际上没有想象中那么简单。它需要设计人员不断在脑中分析系统，进行模块分解，分析需求，提炼细节，而且文件组织不是说一次写好就会不变的，它就像代码一样也是需要不断重构的。每次更深层次的思考，都会使得文件组织更加切合目标产品的组织。在开发的早期就思考文件组织，它就像做菜的调料一样，能够使我们的系统设计的早期阶段就能考虑的更加实际。所以，它是有助于改善系统设计的。

从开发过程上来说，一旦我们文件组织定下来后，我们可以毫不犹豫的把文件雏形建立起来，把概要设计、详细设计的数据结构，算法等代码写进文件中。这样就可以在早期就把编译环境建立起来。随着文件内容不断的增多，伴随着每天的成功编译，将有助于我们提高开发进度和增强团队的自信心。

此外，从概念完整性角度来说。文件组织的含义包括**文件的组成**和**文件的命名**两方面。文件命名至关重要，而且，文件命名也是没有想象中那么简单而随意。比如，Juphoon的文件常用的模块入口文件为x\_task.c，如果有2个大的产品模块，它们都有一个文件专门负责任务管理的，它们分别起名为 a\_start.c. 和 b\_task.c. 从文件的名称上，我们很难一眼看出他们的共同点，如果这时候有个员工来维护这2个模块，由于他熟悉原先其它产品的文件组织，他可能一眼就看出b模块的任务入口点在哪里，然后就可以通过入口点调试跟踪，然后逐渐深入，最后理解了b模块的设计与实现。接着，当他去看a模块的代码时，就很难一眼看出模块的入口点在哪里，增加了阅读难度，而且也影响了维护进度和质量。

因此，在多个模块之间，尤其是具有相似的功能集合中，推广统一的文件命名就具有积极意义。下面就是描述了在Juphoon公司的项目开发中常用的文件组织。

下面将以 xxx 作为模块名称进行举例。

- ABNF 编解码文件组织

文件名称	文件含义
xxx_abnf.c	Xxx abnf 模块初始化实现
xxx_abnf.h	Xxx abnf 模块初始化声明
xxx_abnf_cfg.c	Xxx abnf 模块配置实现
xxx_abnf_cfg.h	Xxx abnf 模块配置声明

文件名称	文件含义
xxx_abnf_chrset.c	Xxx abnf 模块字符集管理实现
xxx_abnf_chrset.h	Xxx abnf 模块字符集管理声明
xxx_abnf_decode.c	Xxx abnf 模块解码实现
xxx_abnf_decode.h	Xxx abnf 模块解码声明
xxx_abnf_encode.c	Xxx abnf 模块编码实现
xxx_abnf_encode.h	Xxx abnf 模块编码声明
xxx_abnf_tkn.c	Xxx abnf 模块 token 管理实现
xxx_abnf_tkn.h	Xxx abnf 模块 token 管理声明
xxx_abnf_type.h	Xxx abnf 模块基本数据结构声明
xxx_abnf_util.c	Xxx abnf 模块工具实现
xxx_abnf_util.h	Xxx abnf 模块工具声明
xxx_abnf_version.c	Xxx abnf 模块版本实现
xxx_abnf_version.h	Xxx abnf 模块版本声明

- 基本模块文件组织

文件名称	文件含义
xxx.h	Xxx 模块公开的头文件声明
xxx_prv.h	Xxx 模块私有的头文件声明
xxx_codec.c	Xxx 模块编解码实现
xxx_codec.h	Xxx 模块编解码声明
xxx_cfg.c	Xxx 模块配置实现
xxx_cfg.h	Xxx 模块配置公开声明
xxx_cfg_prv.h	Xxx 模块配置私有声明
xxx_dbg.c	Xxx 模块调试实现
xxx_dbg.h	Xxx 模块调试声明
xxx_sres.c	Xxx 模块服务资源（Service Resource）实现
xxx_sres.h	Xxx 模块服务资源（Service Resource）声明
xxx_task.c	Xxx 模块任务实现
xxx_task.h	Xxx 模块任务公开声明
xxx_task_prv.h	Xxx 模块任务私有声明
xxx_tmr.c	Xxx 模块计时器实现
xxx_tmr.h	Xxx 模块计时器声明
xxx_tpt.c	Xxx 模块传输实现
xxx_tpt.h	Xxx 模块传输声明
xxx_trans.c	Xxx 模块事务处理实现
xxx_trans.h	Xxx 模块事务处理声明

文件名称	文件含义
xxx_type.h	Xxx 模块基本数据接口公开声明
xxx_type_prv.h	Xxx 模块基本数据接口私有声明
xxx_li.c	Xxx 模块下层接口实现
xxx_li.h	Xxx 模块下层接口声明
xxx_li_ds.c	Xxx 模块下层接口下行实现
xxx_li_us.c	Xxx 模块下层接口上行实现
xxx_mi.c	Xxx 模块层管理接口实现
xxx_mi.h	Xxx 模块层管理接口声明
xxx_mi_ds.c	Xxx 模块层管理接口下行实现
xxx_mi_us.c	Xxx 模块层管理接口上行实现
xxx_ui.c	Xxx 模块上层接口实现
xxx_ui.h	Xxx 模块上层接口声明
xxx_ui_ds.c	Xxx 模块上层接口下行实现
xxx_ui_us.c	Xxx 模块上层接口上行实现
xxx_util.c	Xxx 模块工具实现
xxx_util.h	Xxx 模块工具声明

## 20. 附录八：IEEE Software Quality Glossary

术语	含义	注释
Functionality	The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions	
Suitability	The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.	Suitability also affects operability
Accuracy	The capability of the software product to provide the right or agreed results or effects with the needed degree of precision.	
Interoperability	The capability of the software product to interact with one or more specified systems.	
Security	The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them.	
Reliability	The capability of the software product to maintain a specified level of performance when used under specified conditions.	
Maturity	The capability of the software product to avoid failure as a result of faults in the software.	1. Availability is the capability of the software product to be in a state to perform a required function at a given point in time, under stated conditions of use 2. Availability is a combination of maturity
Fault tolerance	The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.	
Recoverability	The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure.	Following a failure, a software product will sometimes be down for a certain period of time, the length of which is assessed by its recoverability.
Usability	The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.	1. Some aspects of functionality, reliability and efficiency will also affect usability 2. Users may include operators,

术语	含义	注释
		end users and indirect users who are under the influence of or dependent on the use of the software. Usability should address all of the different user environments that the software may affect, which may include preparation for usage and evaluation of results.
Understandability	The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.	
Learnability	The capability of the software product to enable the user to learn its application.	
Operability	The capability of the software product to enable the user to operate and control it.	1. Aspects of suitability, changeability, adaptability and installability may affect operability. 2. Operability corresponds to controllability, error tolerance and conformity with user expectations
Attractiveness	The capability of the software product to be attractive to the user.	
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.	1. Time behaviour 2. Resource utilisation
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.	
Analysability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.	
Changeability	The capability of the software product to enable a specified modification to be implemented.	Implementation includes coding, designing and documenting changes.
Stability	The capability of the software product to avoid unexpected effects from modifications of the software.	

术语	含义	注释
Testability	The capability of the software product to enable modified software to be validated.	
Portability	The capability of the software product to be transferred from one environment to another.	The environment may include organisational, hardware or software environment.
Adaptability	The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.	Adaptability includes the scalability of internal capacity (e.g. screen fields, tables, transaction volumes, report formats, etc.)
Installability	The capability of the software product to be installed in a specified environment.	
Co-existence	The capability of the software product to co-exist with other independent software in a common environment sharing common resources.	
Replaceability	The capability of the software product to be used in place of another specified software product for the same purpose in the same environment.	Replaceability may include attributes of both installability and adaptability