**软件编码规范**

S/W Coding Rule

|  |  |  |  |
| --- | --- | --- | --- |
| **适用范围/Range** | xxxx（xxxxx）软件组 | | |
| **版次/生效日期 Rev/Date** |  |  |  |
| **制/修订 Modified by** |  |  |  |
| **批准 Approved by** |  |  |  |

**修订历史REVISION HISTORY**：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本VERSION | 修订者AUTHOR | 修订日期DATE | 修订内容  UPDATED CONTENT | 注释  COMMENT |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

总 则

[1. 目的 10](#_Toc330304664)

[1.1 增强可读性。 10](#_Toc330304665)

[1.2 减少潜在性错误。 10](#_Toc330304666)

[1.3 提高可维护性。 10](#_Toc330304667)

[1.4 增加可移植性。 10](#_Toc330304668)

[1.5 增强可配置性。 10](#_Toc330304669)

[1.6 方便调试。 10](#_Toc330304670)

[1.7 为测试工具的验证提供必要条件。 10](#_Toc330304671)

[2. 排版 11](#_Toc330304672)

[2.1 程序块要采用缩进风格编写，缩进量一致并以TAB键为单位，定义TAB为4个空格。 11](#_Toc330304673)

[2.2 相对独立的程序块之间、变量说明之后必须加空行。 11](#_Toc330304674)

[2.3 较长的语句（>80字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。 11](#_Toc330304675)

[2.4 循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，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。 11](#_Toc330304676)

[2.5 若函数或过程中的参数较长，则要进行适当的划分。 11](#_Toc330304677)

[2.6 不允许把多个短语句写在一行中，即一行只写一条语句。 12](#_Toc330304678)

[2.7 if、for、do、while、case、switch、default等语句自占一行，且if、for、do、while等语句的执行语句部分无论多少都要加括号{}。 12](#_Toc330304679)

[2.8 函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case语句下的情况处理语句也要遵从语句缩进要求。 12](#_Toc330304680)

[2.9 定义函数时，若函数名前的修饰符较长时，函数名应另起1行书写 12](#_Toc330304681)

[2.10 程序块的分界符（如C/C++语言的大括号‘{’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语句中的程序都要采用如上的缩进方式。 12](#_Toc330304682)

[2.11 在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者 前后要加空格；对函数定义时，函数名和左括号之间加空格，声明和调用时则不必加；进行非对等操作时，如果是关系密切的立即操作符（如－>），后不应加空格。 13](#_Toc330304683)

[2.12 声明变量时，变量类型、变量名、注释应左对齐。 13](#_Toc330304684)

[2.13 声明并同时初始化一个结构变量或一维数组时，各成员应分行书写，以便对相关成员进行注释;声明并同事初始化一个多维数组时，各维元素集应分行书写。 13](#_Toc330304685)

[2.14 在定义结构体或联合体时，“{”和“}”应独占一行，并左对齐。 14](#_Toc330304686)

[3. 注释 14](#_Toc330304687)

[3.1 针对C语言代码文件中应使用“/\* \*/”进行注释，防止部分编译器不识别“//”注释。 14](#_Toc330304688)

[3.2 一般情况下，源程序有效注释量必须在20％以上。 14](#_Toc330304689)

[3.3 源代码文件（.C, .H 等）头部应进行注释，注释必须列出：项目名称、文件名称、作者、版本号、最后修改者、最后修改日期等，部分项必须符合SVN的关键字替换。 14](#_Toc330304690)

[3.4 函数头部应进行注释，包括函数名称、作者、函数目的，输入参数和返回值等。 14](#_Toc330304691)

[3.5 在文件（.h, .C）的末尾添加文件结束的注释，并在该注释下面添加一行空行。 14](#_Toc330304692)

[3.6 边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性，删除无效的代码注释。 15](#_Toc330304693)

[3.7 注释的内容要清楚、明了，含义准确，防止注释二义性。 15](#_Toc330304694)

[3.8 避免在注释中使用缩写，特别是非常用缩写。 15](#_Toc330304695)

[3.9 注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。 15](#_Toc330304696)

[3.10 对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。 15](#_Toc330304697)

[3.11 数据结构声明(包括数组、结构、类、枚举等)，如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。 15](#_Toc330304698)

[3.12 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。 15](#_Toc330304699)

[3.13 注释与所描述内容进行同样的缩排。 16](#_Toc330304700)

[3.14 将注释与其上面的代码用空行隔开。 16](#_Toc330304701)

[3.15 对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。 16](#_Toc330304702)

[3.16 对于switch语句下的case语句，如果因为特殊情况需要处理完一个case后进入下一个case处理，必须在该case语句处理完、下一个case语句前加上明确的注释。 16](#_Toc330304703)

[3.17 避免在一行代码中间插入注释。 17](#_Toc330304704)

[3.18 在代码的功能、意图层次上进行注释，提供有用、额外的信息。 17](#_Toc330304705)

[3.19 在程序块的结束行右方加注释标记，以表明某程序块的结束。 17](#_Toc330304706)

[3.20 所有的注释语言必须为易懂的英文。 17](#_Toc330304707)

[3.21 修改源程序时，应添加注释，尤其在添加程序块时，应在程序块的头尾分别加上起始和结束注释标识。 17](#_Toc330304708)

[4. 标识符 17](#_Toc330304709)

[4.1 标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。 17](#_Toc330304710)

[4.2 命名中若使用特殊约定或缩写，则要有注释说明。 17](#_Toc330304711)

[4.3 自己特有的命名风格，要自始至终保持一致，不可来回变化。 18](#_Toc330304712)

[4.4 对于变量命名，禁止取单个字符（如i、j、k...），建议除了要有具体含义外，还能表明其变量类型、数据类型等，但i、j、k作局部循环变量是允许的。 18](#_Toc330304713)

[4.5 命名规范必须与所使用的系统风格保持一致，并在同一项目中统一。 18](#_Toc330304714)

[4.6 除非必要，不要用数字或较奇怪的字符来定义标识符。 18](#_Toc330304715)

[4.7 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。 18](#_Toc330304716)

[4.8 除了编译开关/头文件等特殊应用，应避免使用\_EXAMPLE\_TEST\_之类以下划线开始和结尾的定义。 18](#_Toc330304717)

[5. 可读性 18](#_Toc330304718)

[5.1 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。 18](#_Toc330304719)

[5.2 避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。 19](#_Toc330304720)

[5.3 源程序中关系较为紧密的代码应尽可能相邻。 19](#_Toc330304721)

[5.4 不要使用难懂的技巧性很高的语句，除非很有必要时。 19](#_Toc330304722)

[6. 变量、结构 19](#_Toc330304723)

[6.1 去掉没必要的公共变量。 19](#_Toc330304724)

[6.2 仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。 19](#_Toc330304725)

[6.3 明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。 19](#_Toc330304726)

[6.4 防止局部变量与公共变量同名。 20](#_Toc330304727)

[6.5 严禁使用未经初始化的变量作为右值。 20](#_Toc330304728)

[6.6 构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的公共变量，防止多个不同模块或函数都可以修改、创建同一公共变量的现象。 20](#_Toc330304729)

[6.7 使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。 20](#_Toc330304730)

[6.8 结构的功能要单一，是针对一种事务的抽象。 20](#_Toc330304731)

[6.9 不同结构间的关系不要过于复杂。 21](#_Toc330304732)

[6.10 结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构，以减少原结构中元素的个数。 21](#_Toc330304733)

[6.11 仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。 21](#_Toc330304734)

[6.12 在定义结构体时，为了更好地理解各元素占用内存的大小建议最好使用“#program pack(1)”的方式来定义。 22](#_Toc330304735)

[6.13 结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地（如预留一些空间等）。 22](#_Toc330304736)

[6.14 留心具体语言及编译器处理不同数据类型的原则及有关细节。 22](#_Toc330304737)

[6.15 编程时，要注意数据类型的强制转换。 22](#_Toc330304738)

[6.16 对编译系统默认的数据类型转换，也要有充分的认识。除非必要，建议减少不必要的类型转换。（左、右值类型尽可能保持一致） 22](#_Toc330304739)

[6.17 对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其命名方式在同一产品中的统一。 22](#_Toc330304740)

[6.18 当声明用于分布式环境或不同CPU间通信环境的数据结构时，必须考虑机器的字节顺序、使用的位域及字节对齐等问题 。 23](#_Toc330304741)

[6.19 当一个数组声明为具有外部链接它的大小应该显式声明或者通过初始化进行隐式定义。 23](#_Toc330304742)

[6.20 在枚举列表中“=”不能显式用于除首元素之外的元素上除非所有的元素都是显式初始化的。 23](#_Toc330304743)

[7. 函数、过程 24](#_Toc330304744)

[7.1 对所调用函数的错误返回码要仔细、全面地处理。 24](#_Toc330304745)

[7.2 明确函数功能，精确（而不是近似）地实现函数设计。 24](#_Toc330304746)

[7.3 编写可重入函数时，应注意局部变量的使用（如编写C/C++语言的可重入函数时，应使用auto即缺省态局部变量或寄存器变量）。 24](#_Toc330304747)

[7.4 编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即P、V操作）等手段对其加以保护。 24](#_Toc330304748)

[7.5 在同一项目组应明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责，缺省是由函数调用者负责。最好是接口函数自己负责.这样可以减少参数合法性检查的代码冗余。 24](#_Toc330304749)

[7.6 防止将函数的参数作为工作变量。 25](#_Toc330304750)

[7.7 函数的规模尽量限制在200行以内。 25](#_Toc330304751)

[7.8 一个函数仅完成一件功能。 25](#_Toc330304752)

[7.9 为简单功能编写函数。 25](#_Toc330304753)

[7.10 不要设计多用途面面俱到的函数。 25](#_Toc330304754)

[7.11 函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。 25](#_Toc330304755)

[7.12 尽量不要编写依赖于其他函数内部实现的函数。 26](#_Toc330304756)

[7.13 避免设计多参数函数，不使用的参数从接口中去掉。 26](#_Toc330304757)

[7.14 非调度函数应减少或防止控制参数，尽量只使用数据参数。 26](#_Toc330304758)

[7.15 检查函数所有参数输入的有效性。 27](#_Toc330304759)

[7.16 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。 27](#_Toc330304760)

[7.17 函数名应准确描述函数的功能。 27](#_Toc330304761)

[7.18 使用动宾词组为执行某操作的函数命名。如果是OOP方法，可以只有动词（名词是对象本身）。 27](#_Toc330304762)

[7.19 避免使用无意义或含义不清的动词为函数命名。 27](#_Toc330304763)

[7.20 函数的返回值要清楚、明了，让使用者不容易忽视错误情况。 27](#_Toc330304764)

[7.21 除非必要，最好不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回。 27](#_Toc330304765)

[7.22 在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换。 27](#_Toc330304766)

[7.23 避免函数中不必要语句，防止程序中的垃圾代码。 27](#_Toc330304767)

[7.24 防止把没有关联的语句放到一个函数中。 27](#_Toc330304768)

[7.25 如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。 28](#_Toc330304769)

[7.26 功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在。 28](#_Toc330304770)

[7.27 设计高扇入、合理扇出（小于7）的函数。 28](#_Toc330304771)

[7.28 减少函数本身或函数间的递归调用。 28](#_Toc330304772)

[7.29 改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。优化函数结构时，要遵守以下原则： 28](#_Toc330304773)

[7.30 在多任务操作系统的环境下编程，要注意函数可重入性的构造。 28](#_Toc330304774)

[7.31 对于提供了返回值的函数，在引用时最好使用其返回值。 29](#_Toc330304775)

[7.32 当一个过程（函数）中对较长变量（一般是结构的成员）有较多引用时，可以用一个意义相当的宏代替。 29](#_Toc330304776)

[7.33 浮点表达式不能做相等或不等的检测。 29](#_Toc330304777)

[8. 宏的定义和使用规则 29](#_Toc330304778)

[8.1 所有的头文件（.h）必须有如下宏定义： 29](#_Toc330304779)

[8.2 用宏定义表达式时，要使用完备的括号。 29](#_Toc330304780)

[8.3 将宏所定义的多条表达式放在大括号中。 29](#_Toc330304781)

[8.4 使用宏时，不允许参数发生变化。 30](#_Toc330304782)

[8.5 定义宏时，所有的宏应全部大写。 30](#_Toc330304783)

[8.6 定义多行宏时，应在行末使用斜杠换行，注意最后一行不能用斜杠。 30](#_Toc330304784)

[9. 程序效率 30](#_Toc330304785)

[9.1 编程时要经常注意代码的效率。 30](#_Toc330304786)

[9.2 在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率。 30](#_Toc330304787)

[9.3 局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。 30](#_Toc330304788)

[9.4 通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。 30](#_Toc330304789)

[9.5 循环体内工作量最小化。 31](#_Toc330304790)

[9.6 不应花过多的时间拼命地提高调用不很频繁的函数代码效率。 31](#_Toc330304791)

[9.7 要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数。 31](#_Toc330304792)

[9.8 在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量，来提高空间效率。 31](#_Toc330304793)

[9.9 在多重循环中，应将最忙的循环放在最内层。 31](#_Toc330304794)

[9.10 尽量减少循环嵌套层次。建议最多不超过五层。 32](#_Toc330304795)

[9.11 避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中。 32](#_Toc330304796)

[9.12 尽量用乘法或其它方法代替除法，特别是浮点运算中的除法。 32](#_Toc330304797)

[9.13 不要一味追求紧凑的代码。 32](#_Toc330304798)

[10. 质量保证 33](#_Toc330304799)

[10.1 在软件设计过程中构筑软件质量。(高层次的抽象，对程序员而言较难理解) 33](#_Toc330304800)

[10.2 代码质量保证优先原则。 33](#_Toc330304801)

[10.3 只引用属于自己的存贮空间。 33](#_Toc330304802)

[10.4 防止引用已经释放的内存空间。 33](#_Toc330304803)

[10.5 过程/函数中分配的内存，在过程/函数退出之前要释放。 33](#_Toc330304804)

[10.6 过程/函数中申请的（为打开文件而使用的）文件句柄，在过程/函数退出之前要关闭。 33](#_Toc330304805)

[10.7 防止内存操作越界。 34](#_Toc330304806)

[10.8 认真处理程序所能遇到的各种出错情况。在支持异常处理的语言中要善于使用异常处理。 34](#_Toc330304807)

[10.9 系统运行之初，要初始化有关变量及运行环境，防止未经初始化的变量被引用。 34](#_Toc330304808)

[10.10 系统运行之初，要对加载到系统中的数据进行一致性检查。 34](#_Toc330304809)

[10.11 严禁随意更改其它模块或系统的有关设置和配置。(源码管理中应做到随意不可修改他人代码，只可审阅) 34](#_Toc330304810)

[10.12 不能随意改变与其它模块的接口。模块接口应在设计时就规定好，若发现设计时接口定义得不合理等原因而需修改需征得小组同意。 34](#_Toc330304811)

[10.13 充分了解系统的接口之后，再使用系统提供的功能。 34](#_Toc330304812)

[10.14 编程时，要防止差1错误。 35](#_Toc330304813)

[10.15 要时刻注意易混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。 35](#_Toc330304814)

[10.16 有可能的话，if语句尽量加上else分支，对没有else分支的语句要小心对待；switch语句必须有default分支。 35](#_Toc330304815)

[10.17 不要滥用goto语句。 35](#_Toc330304816)

[10.18 不使用与硬件或操作系统关系很大的语句，而使用建议的标准语句，以提高软件的可移植性和可重用性。 35](#_Toc330304817)

[10.19 除非为了满足特殊需求，避免使用嵌入式汇编。 35](#_Toc330304818)

[10.20 精心地构造、划分子模块，并按“接口”部分及“内核”部分合理地组织子模块，以提高“内核”部分的可移植性和可重用性。 35](#_Toc330304819)

[10.21 时刻注意表达式是否会上溢、下溢。 36](#_Toc330304820)

[10.22 使用变量时要注意其边界值的情况。 36](#_Toc330304821)

[10.23 系统应具有一定的容错能力，对一些错误事件（如用户误操作等）能进行自动补救。 36](#_Toc330304822)

[11. 可测性 36](#_Toc330304823)

[11.1 在同一项目组或产品组内，要有一套统一的为集成测试与系统联调准备的调测开关及相应打印函数，并且要有详细的说明。 36](#_Toc330304824)

[11.2 在同一项目组或产品组内，调测打印出的信息串的格式要有统一的形式。信息串中至少要有所在模块名（或源文件名）及行号。 36](#_Toc330304825)

[11.3 编程的同时要为单元测试选择恰当的测试点，并仔细构造测试代码、测试用例，同时给出明确的注释说明。测试代码部分应作为（模块中的）一个子模块，以方便测试代码在模块中的安装与拆卸（通过编译开关控制）。 36](#_Toc330304826)

[11.4 在进行集成测试/系统联调之前，要构造好测试环境、测试项目及测试用例，同时仔细分析并优化测试用例，以提高测试效率。 36](#_Toc330304827)

[11.5 用断言来检查程序正常运行时不应发生但在调测时有可能发生的非法情况。 36](#_Toc330304828)

[11.6 用断言确认函数的参数。 36](#_Toc330304829)

[11.7 正式软件产品中应把断言及其它调测代码去掉（即把有关的编译开关关掉）。 37](#_Toc330304830)

[11.8 用调测开关来切换软件的DEBUG版和正式版，而不要同时存在正式版本和DEBUG版本的不同源文件，以减少维护的难度。 37](#_Toc330304831)

[11.9 软件的DEBUG版本和发行版本应该统一维护，不允许分家，并且要时刻注意保证两个版本在实现功能上的一致性。 37](#_Toc330304832)

[12. 代码编译、测试和维护 37](#_Toc330304833)

[12.1 打开编译器的所有告警开关对程序进行编译。 37](#_Toc330304834)

[12.2 在产品软件（项目组）中，要统一编译开关选项。 37](#_Toc330304835)

[12.3 某些语句经编译后产生告警，但如果你认为它是正确的，那么应通过某种手段去掉告警信息。 37](#_Toc330304836)

[12.4 单元测试要求至少达到语句覆盖。 37](#_Toc330304837)

[12.5 正式版本上软件的任何修改都应有详细的文档记录。 37](#_Toc330304838)

# 目的

## 增强可读性。

## 减少潜在性错误。

## 提高可维护性。

## 增加可移植性。

## 增强可配置性。

## 方便调试。

## 为测试工具的验证提供必要条件。

# 排版

## 程序块要采用缩进风格编写，缩进量一致并以TAB键为单位，定义TAB为4个空格。

## 相对独立的程序块之间、变量说明之后必须加空行。

示例：

不符合规范：

if (!valid\_ni(ni))

{

... // program code

}

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

符合规范：

if (!valid\_ni(ni))

{

... // program code

}

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

## 较长的语句（>80字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

示例：

perm\_count\_msg.head.len = NO7\_TO\_STAT\_PERM\_COUNT\_LEN

+ STAT\_SIZE\_PER\_FRAM \* sizeof( \_UL );

act\_task\_table[frame\_id \* STAT\_TASK\_CHECK\_NUMBER + index].occupied

= stat\_poi[index].occupied;

act\_task\_table[taskno].duration\_true\_or\_false

= SYS\_get\_sccp\_statistic\_state( stat\_item );

report\_or\_not\_flag = ((taskno < MAX\_ACT\_TASK\_NUMBER)

&& (n7stat\_stat\_item\_valid (stat\_item))

&& (act\_task\_table[taskno].result\_data != 0));

## 循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，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。

示例：

if ((taskno < max\_act\_task\_number)

&& (n7stat\_stat\_item\_valid (stat\_item)))

{

... // program code

}

for (i = 0, j = 0; (i < BufferKeyword[word\_index].word\_length)

&& (j < NewKeyword.word\_length); i++, j++)

{

... // program code

}

for (i = 0, j = 0;

(i < first\_word\_length) && (j < second\_word\_length);

i++, j++)

{

... // program code

}

## 若函数或过程中的参数较长，则要进行适当的划分。

示例：

n7stat\_str\_compare((BYTE \*) & stat\_object,

(BYTE \*) & (act\_task\_table[taskno].stat\_object),

sizeof (\_STAT\_OBJECT));

n7stat\_flash\_act\_duration( stat\_item, frame\_id \*STAT\_TASK\_CHECK\_NUMBER

+ index, stat\_object );

## 不允许把多个短语句写在一行中，即一行只写一条语句。

示例：

不符合规范：

rect.length = 0; rect.width = 0;

符合规范

rect.length = 0;

rect.width = 0;

## if、for、do、while、case、switch、default等语句自占一行，且if、for、do、while等语句的执行语句部分无论多少都要加括号{}。

示例：

不符合规范：

if (pUserCR == NULL) return;

符合规范：

if (pUserCR == NULL)

{

return;

}

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

## 定义函数时，若函数名前的修饰符较长时，函数名应另起1行书写

示例：

extern “C” (int \*)func(void)

应如下书写:

extern “C” (int \*)

func (void)

{

...//program code

}

## 程序块的分界符（如C/C++语言的大括号‘{’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语句中的程序都要采用如上的缩进方式。

示例：

不符合规范：

for (...) {

... // program code

}

if (...)

{

... // program code

}

void example\_fun( void )

{

... // program code

}

符合规范：

for (...)

{

... // program code

}

if (...)

{

... // program code

}

void example\_fun( void )

{

... // program code

}

## 在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者 前后要加空格；对函数定义时，函数名和左括号之间加空格，声明和调用时则不必加；进行非对等操作时，如果是关系密切的立即操作符（如－>），后不应加空格。

说明：

采用这种松散方式编写代码的目的是使代码更加清晰。

由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在C/C++语言中括号已经是最清晰的标志了。

在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

示例：

(1) 逗号、分号只在后面加空格。

int a, b, c;

(2) 比较操作符, 赋值操作符"="、 "+="，算术操作符"+"、"%"，逻辑操作符"&&"、"&"，位域操作符"<<"、"^"等双目操作符的前后加空格。

if (current\_time >= MAX\_TIME\_VALUE)

a = b + c;

a \*= 2;

a = b ^ 2;

(3)"!"、"~"、"++"、"--"、"&"（地址运算符）等单目操作符前后不加空格。

\*p = 'a'; // 内容操作"\*"与内容之间

flag = !isEmpty; // 非操作"!"与内容之间

p = &mem; // 地址操作"&" 与内容之间

i++; // "++","--"与内容之间

(4)"->"、"."前后不加空格。

p->id = pid; // "->"指针前后不加空格

(5) if、for、while、switch等与后面的括号间应加空格，使if等关键字更为突出、明显。

if (a >= b && c > d)

(6)函数定义时，函数名与左括号间加空格，而调用或声明时则不加空格，以方便搜索。

void func (int i)

{

//program code

}

extern void func(int i);

func(4);

## 声明变量时，变量类型、变量名、注释应左对齐。

示例：

int m\_iNum; //注释1

char \* m\_szName; //注释2

CBufferPool \* m\_pBufferPool; //注释3

## 声明并同时初始化一个结构变量或一维数组时，各成员应分行书写，以便对相关成员进行注释;声明并同事初始化一个多维数组时，各维元素集应分行书写。

示例：

struct PersonInfo employee =

{

“Zhangming”; //name information

MALE; //Sex information

28; //years of age

}

int Triangle[][3] =

{

{ 3, 5 ,6 }, //note.

{ 10, 15, 15 } //note.

}

## 在定义结构体或联合体时，“{”和“}”应独占一行，并左对齐。

示例：

typedef struct \_Point

{

Int8 cx;

Int8 cy;

}Point;

# 注释

## 针对C语言代码文件中应使用“/\* \*/”进行注释，防止部分编译器不识别“//”注释。

## 一般情况下，源程序有效注释量必须在20％以上。

说明：注释的原则是有助于对程序的阅读理解，在该加的地方都加了，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

## 源代码文件（.C, .H 等）头部应进行注释，注释必须列出：项目名称、文件名称、作者、版本号、最后修改者、最后修改日期等，部分项必须符合SVN的关键字替换。

示例：

/\*

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

$Project : 项目名称

$File Name : 文件名称

$Author: 创建者姓名

$Revision: $

$LastChangedBy: $

$LastChangedDate: $

$Description: 文件说明

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

\* COPYRIGHT 2012 DEFA TECHNOLOGY (WUXI) CO.,LTD.

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

\*/

## 函数头部应进行注释，包括函数名称、作者、函数目的，输入参数和返回值等。

示例：

/\*========================================================================

Function : 函数名称

Author : 作者

Purpose : 函数说明

Param : 参数说明

Return : 返回值说明

========================================================================\*/

## 在文件（.h, .C）的末尾添加文件结束的注释，并在该注释下面添加一行空行。

说明：有的编译器会检查文件末尾的空行，如果发现没有空行该编译器会报出警告。

示例：

/\* ============================== End Of File ==========================\*/

## 边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性，删除无效的代码注释。

## 注释的内容要清楚、明了，含义准确，防止注释二义性。

说明：错误的注释不但无益反而有害。

## 避免在注释中使用缩写，特别是非常用缩写。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

## 注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

示例：

不符合规范：

例1：

/\* get replicate sub system index and net indicator \*/

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

例2：

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

/\* get replicate sub system index and net indicator \*/

符合规范：

/\* get replicate sub system index and net indicator \*/

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

## 对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。

示例：

/\* active statistic task number \*/

#define MAX\_ACT\_TASK\_NUMBER 1000

或

#define MAX\_ACT\_TASK\_NUMBER 1000 /\* active statistic task number \*/

## 数据结构声明(包括数组、结构、类、枚举等)，如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。

示例：

可按如下形式说明枚举/数据/联合结构：

/\* sccp interface with sccp user primitive message name \*/

enum SCCP\_USER\_PRIMITIVE

{

N\_UNITDATA\_IND, /\* sccp notify sccp user unit data come \*/

N\_NOTICE\_IND, /\* sccp notify user the No.7 network can not \*/

/\* transmission this message \*/

N\_UNITDATA\_REQ, /\* sccp user's unit data transmission request\*/

};

## 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。

示例：

/\* The ErrorCode when SCCP translate \*/

/\* Global Title failure, as follows \*/ // 变量作用、含义

/\* 0 － SUCCESS 1 － GT Table error \*/

/\* 2 － GT error Others － no use \*/ // 变量取值范围

/\* only function SCCPTranslate() in \*/

/\* this modual can modify it, and other \*/

/\* module can visit it through call \*/

/\* the function GetGTTransErrorCode() \*/ // 使用方法

BYTE g\_GTTranErrorCode;

## 注释与所描述内容进行同样的缩排。

说明：可使程序排版整齐，并方便注释的阅读与理解。

示例：

不符合规范：

void example\_fun( void )

{

/\* code one comments \*/

CodeBlock One

/\* code two comments \*/

CodeBlock Two

}

符合规范：

void example\_fun( void )

{

/\* code one comments \*/

CodeBlock One

/\* code two comments \*/

CodeBlock Two

}

## 将注释与其上面的代码用空行隔开。

示例：

不符合规范：

/\* code one comments \*/

program code one

/\* code two comments \*/

program code two

符合规范：

/\* code one comments \*/

program code one

/\* code two comments \*/

program code two

## 对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。

说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

## 对于switch语句下的case语句，如果因为特殊情况需要处理完一个case后进入下一个case处理，必须在该case语句处理完、下一个case语句前加上明确的注释。

说明：这样比较清楚程序编写者的意图，有效防止无故遗漏break语句。

示例（注意斜体红色部分）：

case CMD\_UP:

ProcessUp();

break;

case CMD\_DOWN:

ProcessDown();

break;

case CMD\_FWD:

ProcessFwd();

if (...)

{

...

break;

}

else

{

**ProcessCFW\_B();** // now jump into case CMD\_A

}

case CMD\_A:

ProcessA();

break;

case CMD\_B:

ProcessB();

break;

...

## 避免在一行代码中间插入注释。

## 在代码的功能、意图层次上进行注释，提供有用、额外的信息。

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

示例：

如下注释意义不大：

/\* if receive\_flag is TRUE \*/

if (receive\_flag)

而如下的注释则给出了额外有用的信息。 ：

/\* if mtp receive a message from links \*/

if (receive\_flag)

## 在程序块的结束行右方加注释标记，以表明某程序块的结束。

说明：当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。

示例：

if (...)

{

// program code

while (index < MAX\_INDEX)

{

// program code

} /\* end of while (index < MAX\_INDEX) \*/ // 指明该条while语句结束

} /\* end of if (...)\*/ // 指明是哪条if语句结束

## 所有的注释语言必须为易懂的英文。

## 修改源程序时，应添加注释，尤其在添加程序块时，应在程序块的头尾分别加上起始和结束注释标识。

# 标识符

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

说明：较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成缩写；一些单词有大家公认的缩写。

示例：

如下单词的缩写能够被大家基本认可。

temp 可缩写为 tmp ;

flag 可缩写为 flg ;

statistic 可缩写为 stat ;

increment 可缩写为 inc ;

message 可缩写为 msg ;

## 命名中若使用特殊约定或缩写，则要有注释说明。

说明：应该在源文件的开始之处，对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

## 自己特有的命名风格，要自始至终保持一致，不可来回变化。

说明：个人的命名风格，在符合所在项目组或产品组的命名规则的前提下，才可使用。（即命名规则中没有规定到的地方才可有个人命名风格）。

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

说明：变量，尤其是局部变量，如果用单个字符表示，很容易敲错（如i写成j），而编译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。

## 命名规范必须与所使用的系统风格保持一致，并在同一项目中统一。

## 除非必要，不要用数字或较奇怪的字符来定义标识符。

示例：

如下命名，使人产生疑惑。

#define \_EXAMPLE\_0\_TEST\_

#define \_EXAMPLE\_1\_TEST\_

void set\_sls00( BYTE sls );

应改为有意义的单词命名

#define \_EXAMPLE\_UNIT\_TEST\_

#define \_EXAMPLE\_ASSERT\_TEST\_

void set\_udt\_msg\_sls( BYTE sls );

## 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

说明：

下面是一些在软件中常用的反义词组。

add / remove begin / end create / destroy

insert / delete first / last get / release

increment / decrement put / get

add / delete lock / unlock open / close

min / max old / new start / stop

next / previous source / target show / hide

send / receive source / destination

cut / paste up / down

示例：

int min\_sum;

int max\_sum;

int add\_user( BYTE \*user\_name );

int delete\_user( BYTE \*user\_name );

## 除了编译开关/头文件等特殊应用，应避免使用\_EXAMPLE\_TEST\_之类以下划线开始和结尾的定义。

# 可读性

## 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

示例：

下列语句中的表达式

word = (high << 8) | low (1)

if ((a | b) && (a & c)) (2)

if ((a | b) < (c & d)) (3)

如果书写为

high << 8 | low

a | b && a & c

a | b < c & d

由于

high << 8 | low = ( high << 8) | low,

a | b && a & c = (a | b) && (a & c)，

(1)(2)不会出错，但语句不易理解；

a | b < c & d = a | （b < c） & d，(3)造成了判断条件出错。

## 避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

示例：

如下的程序可读性差。

if (Trunk[index].trunk\_state == 0)

{

Trunk[index].trunk\_state = 1;

... // program code

}

应改为如下形式。

#define TRUNK\_IDLE 0

#define TRUNK\_BUSY 1

if (Trunk[index].trunk\_state == TRUNK\_IDLE)

{

Trunk[index].trunk\_state = TRUNK\_BUSY;

... // program code

}

## 源程序中关系较为紧密的代码应尽可能相邻。

说明：便于程序阅读和查找。

示例：

以下代码布局不太合理。

rect.length = 10;

char\_poi = str;

rect.width = 5;

若按如下形式书写，可能更清晰一些。

rect.length = 10;

rect.width = 5; // 矩形的长与宽关系较密切，放在一起。

char\_poi = str;

## 不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

示例：

如下表达式，考虑不周就可能出问题，也较难理解。

\* stat\_poi ++ += 1;

\* ++ stat\_poi += 1;

应分别改为如下。

\*stat\_poi += 1;

stat\_poi++; // 此二语句功能相当于“ \* stat\_poi ++ += 1; ”

++ stat\_poi;

\*stat\_poi += 1; // 此二语句功能相当于“ \* ++ stat\_poi += 1; ”

# 变量、结构

## 去掉没必要的公共变量。

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

## 仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。

说明：在对变量声明的同时，应对其含义、作用及取值范围进行注释说明，同时若有必要还应说明与其它变量的关系。

## 明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。

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

示例：在源文件中，可按如下注释形式说明。

RELATION System\_Init Input\_Rec Print\_Rec Stat\_Score

Student Create Modify Access Access

Score Create Modify Access Access, Modify

注：RELATION为操作关系；System\_Init、Input\_Rec、Print\_Rec、Stat\_Score为四个不同的函数；Student、Score为两个全局变量；Create表示创建，Modify表示修改，Access表示访问。

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

## 防止局部变量与公共变量同名。

说明：若使用了较好的命名规则，那么此问题可自动消除。

## 严禁使用未经初始化的变量作为右值。

说明：特别是引用未经赋值的指针，经常会引起系统崩溃。

## 构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的公共变量，防止多个不同模块或函数都可以修改、创建同一公共变量的现象。

说明：降低公共变量耦合度。

## 使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。

说明：使用标准的数据类型，有利于程序的移植。

示例：

如下例子（在DOS下BC3.1环境中），在移植时可能产生问题。

void main()

{

register int index; // 寄存器变量

\_AX = 0x4000; // \_AX是BC3.1提供的寄存器“伪变量”

... // program code

}

## 结构的功能要单一，是针对一种事务的抽象。

说明：设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

示例：如下结构不太清晰、合理。

typedef struct STUDENT\_STRU

{

unsigned char name[8]; /\* student's name \*/

unsigned char age; /\* student's age \*/

unsigned char sex; /\* student's sex, as follows \*/

/\* 0 - FEMALE; 1 - MALE \*/

unsigned char

teacher\_name[8]; /\* the student teacher's name \*/

unisgned char

teacher\_sex; /\* his teacher sex \*/

} STUDENT;

若改为如下，可能更合理些。

typedef struct TEACHER\_STRU

{

unsigned char name[8]; /\* teacher name \*/

unisgned char sex; /\* teacher sex, as follows \*/

/\* 0 - FEMALE; 1 - MALE \*/

} TEACHER;

typedef struct STUDENT\_STRU

{

unsigned char name[8]; /\* student's name \*/

unsigned char age; /\* student's age \*/

unsigned char sex; /\* student's sex, as follows \*/

/\* 0 - FEMALE; 1 - MALE \*/

unsigned int teacher\_ind; /\* his teacher index \*/

} STUDENT;

## 不同结构间的关系不要过于复杂。

说明：若两个结构间关系较复杂、密切，那么应合为一个结构。

示例：

如下两个结构的构造不合理。

typedef struct PERSON\_ONE\_STRU

{

unsigned char name[8];

unsigned char addr[40];

unsigned char sex;

unsigned char city[15];

} PERSON\_ONE;

typedef struct PERSON\_TWO\_STRU

{

unsigned char name[8];

unsigned char age;

unsigned char tel;

} PERSON\_TWO;

由于两个结构都是描述同一事物的，那么不如合成一个结构。

typedef struct PERSON\_STRU

{

unsigned char name[8];

unsigned char age;

unsigned char sex;

unsigned char addr[40];

unsigned char city[15];

unsigned char tel;

} PERSON;

## 结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构，以减少原结构中元素的个数。

说明：增加结构的可理解性、可操作性和可维护性。

示例：

假如认为如上的\_PERSON结构元素过多，那么可如下对之划分。

typedef struct PERSON\_BASE\_INFO\_STRU

{

unsigned char name[8];

unsigned char age;

unsigned char sex;

} PERSON\_BASE\_INFO;

typedef struct PERSON\_ADDRESS\_STRU

{

unsigned char addr[40];

unsigned char city[15];

unsigned char tel;

} PERSON\_ADDRESS;

typedef struct PERSON\_STRU

{

PERSON\_BASE\_INFO person\_base;

PERSON\_ADDRESS person\_addr;

} PERSON;

## 仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。

说明：合理排列结构中元素顺序，可节省空间并增加可理解性。

示例：

如下结构中的位域排列，将占较大空间，可读性也稍差。

typedef struct EXAMPLE\_STRU

{

unsigned int valid: 1;

PERSON person;

unsigned int set\_flg: 1;

} EXAMPLE;

若改成如下形式，不仅可节省1字节空间，可读性也变好了。

typedef struct EXAMPLE\_STRU

{

unsigned int valid: 1;

unsigned int set\_flg: 1;

PERSON person ;

} EXAMPLE;

## 在定义结构体时，为了更好地理解各元素占用内存的大小建议最好使用“#program pack(1)”的方式来定义。

示例：

#program pack(push)

#program pack(1)

typedef struct \_XX {

…

}XX, \*LPXX;

#program pack(pop)

## 结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地（如预留一些空间等）。

说明：软件向前兼容的特性，是软件产品是否成功的重要标志之一。如果要想使产品具有较好的前向兼容，那么在产品设计之初就应为以后版本升级保留一定余地，并且在产品升级时必须考虑前一版本的各种特性。

## 留心具体语言及编译器处理不同数据类型的原则及有关细节。

说明：如在C语言中，static局部变量将在内存“数据区”中生成，而非static局部变量将在“堆栈”中生成。这些细节对程序质量的保证非常重要。

## 编程时，要注意数据类型的强制转换。

说明：当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而这些细节若考虑不周，就很有可能留下隐患。

## 对编译系统默认的数据类型转换，也要有充分的认识。除非必要，建议减少不必要的类型转换。（左、右值类型尽可能保持一致）

示例：

如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

char chr;

unsigned short int exam;

chr = -1;

exam = chr; // 编译器不产生告警，此时exam为0xFFFF。

## 对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其命名方式在同一产品中的统一。

说明：使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序清晰、简洁。

示例：

可参考如下方式声明自定义数据类型。

下面的声明可使数据类型的使用简洁、明了。

typedef unsigned char BYTE;

typedef unsigned short WORD;

typedef unsigned int DWORD;

下面的声明可使数据类型具有更丰富的含义。

typedef float DISTANCE;

typedef float SCORE;

## 当声明用于分布式环境或不同CPU间通信环境的数据结构时，必须考虑机器的字节顺序、使用的位域及字节对齐等问题 。

说明：

比如Intel CPU与68360 CPU，在处理位域及整数时，其在内存存放的“顺序”正好相反。

示例：

假如有如下短整数及结构。

unsigned short int exam;

typedef struct EXAM\_BIT\_STRU

{ /\* Intel 68360 \*/

unsigned int A1: 1; /\* bit 0 7 \*/

unsigned int A2: 1; /\* bit 1 6 \*/

unsigned int A3: 1; /\* bit 2 5 \*/

} EXAM\_BIT;

如下是Intel CPU生成短整数及位域的方式。

内存： 0 1 2 ... （从低到高，以字节为单位）

exam exam低字节 exam高字节

内存： 0 bit 1 bit 2 bit ... （字节的各“位”）

EXAM\_BIT A1 A2 A3

如下是68360 CPU生成短整数及位域的方式。

内存： 0 1 2 ... （从低到高，以字节为单位）

exam exam高字节 exam低字节

内存： 7 bit 6 bit 5 bit ... （字节的各“位”）

EXAM\_BIT A1 A2 A3

说明：在对齐方式下，CPU的运行效率要快得多。

示例：如下图，当一个long型数（如图中long1）在内存中的位置正好与内存的字边界对齐时，CPU存取这个数只需访问一次内存，而当一个long型数（如图中的long2）在内存中的位置跨越了字边界时，CPU存取这个数就需要多次访问内存，如i960cx访问这样的数需读内存三次（一个BYTE、一个SHORT、一个BYTE，由CPU的微代码执行，对软件透明），所有对齐方式下CPU的运行效率明显快多了。

1 8 16 24 32

------- ------- ------- -------

| long1 | long1 | long1 | long1 |

------- ------- ------- -------

| | | | long2 |

------- ------- ------- --------

| long2 | long2 | long2 | |

------- ------- ------- --------

| ....

## 当一个数组声明为具有外部链接它的大小应该显式声明或者通过初始化进行隐式定义。

说明：

尽管可以在数组声明不完善时访问其元素然而仍然是在数组的大小可以显式确定的情况下这样做才会更为安全。

示例：

extern int array2[] ; /\* Not compliant \*/

int array1[10] ; /\* Compliant \*/

int array2[] = { 0, 10, 15 }; /\* Compliant \*/

## 在枚举列表中“=”不能显式用于除首元素之外的元素上除非所有的元素都是显式初始化的。

说明：

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

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

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

示例：

enum colour { red = 3, blue, green, yellow = 5 } ; /\* not compliant \*/

/\* green and yellow represent the same value – this is duplication \*/

enum colour { red = 3, blue = 4, green = 5, yellow = 5 }; /\* compliant \*/

/\* green and yellow represent the same value – this is duplication \*/

# 函数、过程

## 对所调用函数的错误返回码要仔细、全面地处理。

## 明确函数功能，精确（而不是近似）地实现函数设计。

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

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

## 编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即P、V操作）等手段对其加以保护。

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

示例：假设Exam是int型全局变量，函数Squre\_Exam返回Exam平方值。那么如下函数不具有可重入性。

unsigned int example( int para )

{

unsigned int temp;

Exam = para; // （\*\*）

temp = Square\_Exam( );

return temp;

}

此函数若被多个进程调用的话，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进程可能正好被激活，那么当新激活的进程执行到此函数时，将使Exam赋与另一个不同的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。

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

Exam = para; // 给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。若申请到信号，则可继续执行，但其

// 它进程必须等待本进程释放信号量后，才能再使

// 用本信号。

return temp;

}

## 在同一项目组应明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责，缺省是由函数调用者负责。最好是接口函数自己负责.这样可以减少参数合法性检查的代码冗余。

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。

## 防止将函数的参数作为工作变量。

说明：将函数的参数作为工作变量，有可能错误地改变参数内容，所以很危险。对必须改变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

示例：

下函数的实现不太好。

void sum\_data( unsigned int num, int \*data, int \*sum )

{

unsigned int count;

\*sum = 0;

for (count = 0; count < num; count++)

{

\*sum += data[count]; // sum成了工作变量，不太好。

}

}

若改为如下，则更好些。

void sum\_data( unsigned int num, int \*data, int \*sum )

{

unsigned int count ;

int sum\_temp;

sum\_temp = 0;

for (count = 0; count < num; count ++)

{

sum\_temp += data[count];

}

\*sum = sum\_temp;

}

## 函数的规模尽量限制在200行以内。

说明：不包括注释和空格行。

## 一个函数仅完成一件功能。

## 为简单功能编写函数。

说明：虽然为仅用一两行就可完成的功能去编函数好象没有必要，但用函数可使功能明确化，增加程序可读性，亦可方便维护、测试。

示例：

如下语句的功能不很明显。

value = ( a > b ) ? a : b ;

改为如下就很清晰了。

int max (int a, int b)

{

return ((a > b) ? a : b);

}

value = max (a, b);

或改为如下。

#define MAX (a, b) (((a) > (b)) ? (a) : (b))

value = MAX (a, b);

## 不要设计多用途面面俱到的函数。

说明：多功能集于一身的函数，很可能使函数的理解、测试、维护等变得困难。

## 函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。

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/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，然而，当某函数的返回值为指针类型时，则必须是STATIC的局部变量的地址作为返回值，若为AUTO类，则返回为指针。

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;

}

## 尽量不要编写依赖于其他函数内部实现的函数。

说明：此条为函数独立性的基本要求。由于目前大部分高级语言都是结构化的，所以通过具体语言的语法要求与编译器功能，基本就可以防止这种情况发生。但在汇编语言中，由于其灵活性，很可能使函数出现这种情况。

示例：如下是在DOS下TASM的汇编程序例子。过程Print\_Msg的实现依赖于Input\_Msg的具体实现，这种程序是非结构化的，难以维护、修改。

... // 程序代码

proc Print\_Msg // 过程（函数）Print\_Msg

... // 程序代码

jmp LABEL

... // 程序代码

endp

proc Input\_Msg // 过程（函数）Input\_Msg

... // 程序代码

LABEL:

... // 程序代码

endp

## 避免设计多参数函数，不使用的参数从接口中去掉。

说明：目的减少函数间接口的复杂度。

## 非调度函数应减少或防止控制参数，尽量只使用数据参数。

说明：本建议目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令，来启动相应的功能实体（即函数或过程），而本身并不完成具体功能。控制参数是指改变函数功能行为的参数，即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合，很可能使函数间的耦合度增大，并使函数的功能不唯一。

示例：

如下函数构造不太合理。

int add\_sub( int a, int b, unsigned char add\_sub\_flg )

{

if (add\_sub\_flg == INTEGER\_ADD)

{

return (a + b);

}

else

{

return (a ˜ b);

}

}

不如分为如下两个函数清晰。

int add( int a, int b )

{

return (a + b);

}

int sub( int a, int b )

{

return (a ˜ b);

}

## 检查函数所有参数输入的有效性。

## 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。函数在使用输入之前，应进行必要的检查。

## 函数名应准确描述函数的功能。

## 使用动宾词组为执行某操作的函数命名。如果是OOP方法，可以只有动词（名词是对象本身）。

示例：

参照如下方式命名函数。

void print\_record( unsigned int rec\_ind ) ;

int input\_record( void ) ;

unsigned char get\_current\_color( void ) ;

## 避免使用无意义或含义不清的动词为函数命名。

说明：避免用含义不清的动词如process、handle等为函数命名，因为这些动词并没有说明要具体做什么。

## 函数的返回值要清楚、明了，让使用者不容易忽视错误情况。

说明：函数的每种出错返回值的意义要清晰、明了、准确，防止使用者误用、理解错误或忽视错误返回码。

## 除非必要，最好不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回。

## 在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换。

说明：因为数据类型转换或多或少存在危险。

## 避免函数中不必要语句，防止程序中的垃圾代码。

说明：程序中的垃圾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。

## 防止把没有关联的语句放到一个函数中。

说明：防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。

在编程时，经常遇到在不同函数中使用相同的代码，许多开发人员都愿把这些代码提出来，并构成一个新函数。若这些代码关联较大并且是完成一个功能的，那么这种构造是合理的，否则这种构造将产生随机内聚的函数。

示例：

如下函数就是一种随机内聚。

void Init\_Var( void )

{

Rect.length = 0;

Rect.width = 0; /\* 初始化矩形的长与宽 \*/

Point.x = 10;

Point.y = 10; /\* 初始化“点”的坐标 \*/

}

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。

应如下分为两个函数：

void Init\_Rect( void )

{

Rect.length = 0;

Rect.width = 0; /\* 初始化矩形的长与宽 \*/

}

void Init\_Point( void )

{

Point.x = 10;

Point.y = 10; /\* 初始化“点”的坐标 \*/

}

## 如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。

说明：若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成一个新的函数。

## 功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在。

说明：模块中函数划分的过多，一般会使函数间的接口变得复杂。所以过小的函数，特别是扇入很低的或功能不明确的函数，不值得单独存在。

## 设计高扇入、合理扇出（小于7）的函数。

说明：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，如总是1，表明函数的调用层次可能过多，这样不利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。函数较合理的扇出（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。

## 减少函数本身或函数间的递归调用。

说明：递归调用特别是函数间的递归调用（如A->B->C->A），影响程序的可理解性；递归调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便，应减少没必要的递归调用。

## 改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。优化函数结构时，要遵守以下原则：

（1）不能影响模块功能的实现。

（2）仔细考查模块或函数出错处理及模块的性能要求并进行完善。

（3）通过分解或合并函数来改进软件结构。

（4）考查函数的规模，过大的要进行分解。

（5）降低函数间接口的复杂度。

（6）不同层次的函数调用要有较合理的扇入、扇出。

（7）函数功能应可预测。

（8）提高函数内聚。（单一功能的函数内聚最高）

说明：对初步划分后的函数结构应进行改进、优化，使之更为合理。

## 在多任务操作系统的环境下编程，要注意函数可重入性的构造。

说明：可重入性是指函数可以被多个任务进程调用。在多任务操作系统中，函数是否具有可重入性是非常重要的，因为这是多个进程可以共用此函数的必要条件。另外，编译器是否提供可重入函数库，与它所服务的操作系统有关，只有操作系统是多任务时，编译器才有可能提供可重入函数库。如DOS下BC和MSC等就不具备可重入函数库，因为DOS是单用户单任务操作系统。

## 对于提供了返回值的函数，在引用时最好使用其返回值。

## 当一个过程（函数）中对较长变量（一般是结构的成员）有较多引用时，可以用一个意义相当的宏代替。

说明：这样可以增加编程效率和程序的可读性。

示例：

在某过程中较多引用TheReceiveBuffer[FirstSocket].byDataPtr，

则可以通过以下宏定义来代替：

# define pSOCKDATA TheReceiveBuffer[FirstScoket].byDataPtr

## 浮点表达式不能做相等或不等的检测。

说明：这是浮点类型的固有特性等值比较通常不会计算为true即使期望如此而且这种比较行为不能在执行前做出预测它会随着实现的改变而改变。

示例：

下面代码中的测试结果就是不可预期的

float32\_t x, y;

/\* some calculations in here \*/

if ( x == y) /\* not compliant \*/

{ /\* … \*/ }

if (x == 0.0f) /\* not compliant \*/

间接的检测同样是有问题的在本规则内也是禁止的例如

if ( ( x <= y ) && ( x >= y ) )

{ /\* … \*/ }

# 宏的定义和使用规则

## 所有的头文件（.h）必须有如下宏定义：

（1）防止重复包含宏定义（必须为大写）。

（2）C++可引用宏定义。

示例：

#ifndef \_\_项目名简称\_文件名简称\_H\_\_

#define \_\_项目名简称\_文件名简称\_H\_\_

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*\*\* BODY \*\*\*/

#ifdef \_\_cplusplus

}/\*extern C\*/

#endif

#endif /\* \_\_项目名简称\_文件名简称\_H\_\_ \*/

## 用宏定义表达式时，要使用完备的括号。

示例：

如下定义的宏都存在一定的风险。

#define RECTANGLE\_AREA( a, b ) a \* b

#define RECTANGLE\_AREA( a, b ) (a \* b)

#define RECTANGLE\_AREA( a, b ) (a) \* (b)

正确的定义应为：

#define RECTANGLE\_AREA( a, b ) ((a) \* (b))

## 将宏所定义的多条表达式放在大括号中。

示例：

下面的语句只有宏的第一条表达式被执行。为了说明问题，for语句的书写稍不符规范。

#define INTI\_RECT\_VALUE( a, b )\

a = 0;\

b = 0;

for (index = 0; index < RECT\_TOTAL\_NUM; index++)

INTI\_RECT\_VALUE( rect.a, rect.b );

正确的用法应为：

#define INTI\_RECT\_VALUE( a, b )\

{\

a = 0;\

b = 0;\

}

for (index = 0; index < RECT\_TOTAL\_NUM; index++)

{

INTI\_RECT\_VALUE( rect[index].a, rect[index].b );

}

## 使用宏时，不允许参数发生变化。

示例：

如下用法可能导致错误。

#define SQUARE( a ) ((a) \* (a))

int a = 5;

int b;

b = SQUARE( a++ ); // 结果：a = 7，即执行了两次增1。

正确的用法是：

b = SQUARE( a );

a++; // 结果：a = 6，即只执行了一次

## 定义宏时，所有的宏应全部大写。

## 定义多行宏时，应在行末使用斜杠换行，注意最后一行不能用斜杠。

示例：

#define LB\_ASSERT(condition, ...) { \

if (!(condition)) { \

printf("LB\_ASSERT: [%s:%d] ",\_\_FUNCTION\_\_,\_\_LINE\_\_); \

printf(\_\_VA\_ARGS\_\_); \

while (1); \

}\

} /\* 此处不能有斜杠。 \*/

# 程序效率

## 编程时要经常注意代码的效率。

说明：代码效率分为全局效率、局部效率、时间效率及空间效率。全局效率是站在整个系统的角度上的系统效率；局部效率是站在模块或函数角度上的效率；时间效率是程序处理输入任务所需的时间长短；空间效率是程序所需内存空间，如机器代码空间大小、数据空间大小、栈空间大小等。

## 在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率。

说明：不能一味地追求代码效率，而对软件的正确性、稳定性、可读性及可测性造成影响。

## 局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

## 通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。

说明：这种方式是解决软件空间效率的根本办法。

示例：

如下记录学生学习成绩的结构不合理。

typedef unsigned char BYTE;

typedef unsigned short WORD;

typedef struct STUDENT\_SCORE\_STRUT

{

BYTE name[8];

BYTE age;

BYTE sex;

BYTE class;

BYTE subject;

float score;

} STUDENT\_SCORE;

因为每位学生都有多科学习成绩，故如上结构将占用较大空间。应如下改进（分为两个结构），总的存贮空间将变小，操作也变得更方便。

typedef struct STUDENT\_STRU

{

BYTE name[8];

BYTE age;

BYTE sex;

BYTE class;

} STUDENT;

typedef struct STUDENT\_SCORE\_STRU

{

WORD student\_index;

BYTE subject;

float score;

} STUDENT\_SCORE;

## 循环体内工作量最小化。

说明：应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。

示例：

如下代码效率不高。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++)

{

sum += ind;

back\_sum = sum; /\* backup sum \*/

}

语句“back\_sum = sum;”完全可以放在for语句之后，如下。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++)

{

sum += ind;

}

back\_sum = sum; /\* backup sum \*/

## 不应花过多的时间拼命地提高调用不很频繁的函数代码效率。

说明：对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

## 要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数。

说明：只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方式。嵌入汇编可提高时间及空间效率，但也存在一定风险。

## 在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量，来提高空间效率。

说明：这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。

## 在多重循环中，应将最忙的循环放在最内层。

说明：减少CPU切入循环层的次数。

示例：

如下代码效率不高。

for (row = 0; row < 100; row++)

{

for (col = 0; col < 5; col++)

{

sum += a[row][col];

}

}

可以改为如下方式，以提高效率。

for (col = 0; col < 5; col++)

{

for (row = 0; row < 100; row++)

{

sum += a[row][col];

}

}

## 尽量减少循环嵌套层次。建议最多不超过五层。

## 避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中。

说明：目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不可以。

示例：

如下代码效率稍低。

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

if (data\_type == RECT\_AREA)

{

area\_sum += rect\_area[ind];

}

else

{

rect\_length\_sum += rect[ind].length;

rect\_width\_sum += rect[ind].width;

}

}

因为判断语句与循环变量无关，故可如下改进，以减少判断次数。

if (data\_type == RECT\_AREA)

{

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

area\_sum += rect\_area[ind];

}

}

else

{

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

rect\_length\_sum += rect[ind].length;

rect\_width\_sum += rect[ind].width;

}

}

## 尽量用乘法或其它方法代替除法，特别是浮点运算中的除法。

说明：浮点运算除法要占用较多CPU资源。

示例：

如下表达式运算可能要占较多CPU资源。

#define PAI 3.1416

radius = circle\_length / (2 \* PAI);

应如下把浮点除法改为浮点乘法。

#define PAI\_RECIPROCAL (1 / 3.1416 ) // 编译器编译时，将生成具体浮点数

radius = circle\_length \* PAI\_RECIPROCAL / 2;

## 不要一味追求紧凑的代码。

说明：因为紧凑的代码并不代表高效的机器码。

# 质量保证

## 在软件设计过程中构筑软件质量。(高层次的抽象，对程序员而言较难理解)

## 代码质量保证优先原则。

（1）正确性，指程序要实现设计要求的功能。

（2）稳定性、安全性，指程序稳定、可靠、安全。

（3）可测试性，指程序要具有良好的可测试性。

（4）规范/可读性，指程序书写风格、命名规则等要符合规范。

（5）全局效率，指软件系统的整体效率。

（6）局部效率，指某个模块/子模块/函数的本身效率。

（7）个人表达方式/个人方便性，指个人编程习惯。

## 只引用属于自己的存贮空间。

说明：若模块封装的较好，那么一般不会发生非法引用他人的空间。

## 防止引用已经释放的内存空间。

说明：在实际编程过程中，稍不留心就会出现在一个模块中释放了某个内存块（如C语言指针），而另一模块在随后的某个时刻又使用了它。要防止这种情况发生。

## 过程/函数中分配的内存，在过程/函数退出之前要释放。

## 过程/函数中申请的（为打开文件而使用的）文件句柄，在过程/函数退出之前要关闭。

说明：分配的内存不释放以及文件句柄不关闭，是较常见的错误，而且稍不注意就有可能发生。这类错误往往会引起很严重后果，且难以定位。

示例：

以下函数在退出之前，没有把分配的内存释放。

typedef unsigned char BYTE;

int example\_fun( BYTE gt\_len, BYTE \*gt\_code )

{

BYTE \*gt\_buf;

gt\_buf = (BYTE \*) malloc (MAX\_GT\_LENGTH);

... //program code, include check gt\_buf if or not NULL.

/\* global title length error \*/

if (gt\_len > MAX\_GT\_LENGTH)

{

return GT\_LENGTH\_ERROR; // 忘了释放gt\_buf

}

... // other program code

}

应改为如下。

int example\_fun( BYTE gt\_len, BYTE \*gt\_code )

{

BYTE \*gt\_buf;

gt\_buf = (BYTE \* ) malloc ( MAX\_GT\_LENGTH );

... // program code, include check gt\_buf if or not NULL.

/\* global title length error \*/

if (gt\_len > MAX\_GT\_LENGTH)

{

free( gt\_buf ); // 退出之前释放gt\_buf

return GT\_LENGTH\_ERROR;

}

... // other program code

}

## 防止内存操作越界。

说明：内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当我们进行这些操作时一定要仔细小心。

示例：

假设某软件系统最多可由10个用户同时使用，用户号为1-10，那么如下程序存在问题。

#define MAX\_USR\_NUM 10

unsigned char usr\_login\_flg[MAX\_USR\_NUM]= "";

void set\_usr\_login\_flg( unsigned char usr\_no )

{

if (!usr\_login\_flg[usr\_no])

{

usr\_login\_flg[usr\_no]= TRUE;

}

}

当usr\_no为10时，将使用usr\_login\_flg越界。可采用如下方式解决。

void set\_usr\_login\_flg( unsigned char usr\_no )

{

if (!usr\_login\_flg[usr\_no - 1])

{

usr\_login\_flg[usr\_no - 1]= TRUE;

}

}

## 认真处理程序所能遇到的各种出错情况。在支持异常处理的语言中要善于使用异常处理。

## 系统运行之初，要初始化有关变量及运行环境，防止未经初始化的变量被引用。

## 系统运行之初，要对加载到系统中的数据进行一致性检查。

说明：使用不一致的数据，容易使系统进入混乱状态和不可知状态。

## 严禁随意更改其它模块或系统的有关设置和配置。(源码管理中应做到随意不可修改他人代码，只可审阅)

说明：编程时，不能随心所欲地更改不属于自己模块的有关设置如常量、数组的大小等。

## 不能随意改变与其它模块的接口。模块接口应在设计时就规定好，若发现设计时接口定义得不合理等原因而需修改需征得小组同意。

## 充分了解系统的接口之后，再使用系统提供的功能。

示例：

在B型机的各模块与操作系统的接口函数中，有一个要由各模块负责编写的初始化过程，此过程在软件系统加载完成后，由操作系统发送的初始化消息来调度。因此就涉及到初始化消息的类型与消息发送的顺序问题，特别是消息顺序，若没搞清楚就开始编程，很容易引起严重后果。以下示例引自B型曾出现过的实际代码，其中使用了FID\_FETCH\_DATA与FID\_INITIAL初始化消息类型，注意B型机的系统是在FID\_FETCH\_DATA之前发送FID\_INITIAL的。

MID alarm\_module\_list[MAX\_ALARM\_MID];

int FAR SYS\_ALARM\_proc( FID function\_id, int handle )

{

\_UI i, j;

switch ( function\_id )

{

... // program code

case FID\_INITAIL:

for (i = 0; i < MAX\_ALARM\_MID; i++)

{

if (alarm\_module\_list[i]== BAM\_MODULE // \*\*）

|| (alarm\_module\_list[i]== LOCAL\_MODULE)

{

for (j = 0; j < ALARM\_CLASS\_SUM; j++)

{

FAR\_MALLOC( ... );

}

}

}

... // program code

break;

case FID\_FETCH\_DATA:

... // program code

Get\_Alarm\_Module( ); // 初始化alarm\_module\_list

break;

... // program code

}

}

由于FID\_INITIAL是在FID\_FETCH\_DATA之前执行的，而初始化alarm\_module\_list是在FID\_FETCH\_DATA中进行的，故在FID\_INITIAL中（\*\*）处引用alarm\_module\_list变量时，它还没有被初始化。这是个严重错误。

应如下改正：要么把Get\_Alarm\_Module函数放在FID\_INITIAL中（\*\*）之前；要么就必须考虑（\*\*）处的判断语句是否可以用（不使用alarm\_module\_list变量的）其它方式替代，或者是否可以取消此判断语句。

## 编程时，要防止差1错误。

说明：此类错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，由此引起的后果，很多情况下是很严重的，所以编程时，一定要在这些地方小心。当编完程序后，应对这些操作符进行彻底检查。

## 要时刻注意易混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。

说明：形式相近的操作符最容易引起误用，如C/C++中的“=”与“==”、“|”与“||”、“&”与“&&”等，若拼写错了，编译器不一定能够检查出来。

示例：

如把“&”写成“&&”，或反之。

ret\_flg = (pmsg->ret\_flg & RETURN\_MASK);

被写为：

ret\_flg = (pmsg->ret\_flg && RETURN\_MASK);

rpt\_flg = (VALID\_TASK\_NO( taskno ) && DATA\_NOT\_ZERO( stat\_data ));

被写为：

rpt\_flg = (VALID\_TASK\_NO( taskno ) & DATA\_NOT\_ZERO( stat\_data ));

## 有可能的话，if语句尽量加上else分支，对没有else分支的语句要小心对待；switch语句必须有default分支。

## 不要滥用goto语句。

说明：goto语句会破坏程序的结构性，所以除非确实需要，最好不使用goto语句。

## 不使用与硬件或操作系统关系很大的语句，而使用建议的标准语句，以提高软件的可移植性和可重用性。

## 除非为了满足特殊需求，避免使用嵌入式汇编。

说明：程序中嵌入式汇编，一般都对可移植性有较大的影响。

## 精心地构造、划分子模块，并按“接口”部分及“内核”部分合理地组织子模块，以提高“内核”部分的可移植性和可重用性。

说明：对不同产品中的某个功能相同的模块，若能做到其内核部分完全或基本一致，那么无论对产品的测试、维护，还是对以后产品的升级都会有很大帮助。

## 时刻注意表达式是否会上溢、下溢。

示例：

如下程序将造成变量下溢。

unsigned char size ;

while (size-- >= 0) // 将出现下溢

{

... // program code

}

当size等于0时，再减1不会小于0，而是0xFF，故程序是一个死循环。应如下修改。

char size; // 从unsigned char 改为char

while (size-- >= 0)

{

... // program code

}

## 使用变量时要注意其边界值的情况。

示例：

如C语言中字符型变量，有效值范围为-128到127。故以下表达式的计算存在一定风险。

char chr = 127;

int sum = 200;

chr += 1; // 127为chr的边界值，再加1将使chr上溢到-128，而不是128。

sum += chr; // 故sum的结果不是328，而是72。

若chr与sum为同一种类型，或表达式按如下方式书写，可能会好些。

sum = sum + chr + 1;

## 系统应具有一定的容错能力，对一些错误事件（如用户误操作等）能进行自动补救。

# 可测性

## 在同一项目组或产品组内，要有一套统一的为集成测试与系统联调准备的调测开关及相应打印函数，并且要有详细的说明。

说明：本规则是针对项目组或产品组的。

## 在同一项目组或产品组内，调测打印出的信息串的格式要有统一的形式。信息串中至少要有所在模块名（或源文件名）及行号。

说明：统一的调测信息格式便于集成测试。

## 编程的同时要为单元测试选择恰当的测试点，并仔细构造测试代码、测试用例，同时给出明确的注释说明。测试代码部分应作为（模块中的）一个子模块，以方便测试代码在模块中的安装与拆卸（通过编译开关控制）。

说明：为单元测试而准备。

## 在进行集成测试/系统联调之前，要构造好测试环境、测试项目及测试用例，同时仔细分析并优化测试用例，以提高测试效率。

说明：好的测试用例应尽可能模拟出程序所遇到的边界值、各种复杂环境及一些极端情况等。

## 用断言来检查程序正常运行时不应发生但在调测时有可能发生的非法情况。

## 用断言确认函数的参数。

示例：

假设某函数参数中有一个指针，那么使用指针前可对它检查，如下。

int exam\_fun( unsigned char \*str )

{

EXAM\_ASSERT( str != NULL ); // 用断言检查“假设指针不为空”这个条件

... //other program code

}

## 正式软件产品中应把断言及其它调测代码去掉（即把有关的编译开关关掉）。

说明：加快软件运行速度。

## 用调测开关来切换软件的DEBUG版和正式版，而不要同时存在正式版本和DEBUG版本的不同源文件，以减少维护的难度。

## 软件的DEBUG版本和发行版本应该统一维护，不允许分家，并且要时刻注意保证两个版本在实现功能上的一致性。

# 代码编译、测试和维护

## 打开编译器的所有告警开关对程序进行编译。

## 在产品软件（项目组）中，要统一编译开关选项。

## 某些语句经编译后产生告警，但如果你认为它是正确的，那么应通过某种手段去掉告警信息。

## 单元测试要求至少达到语句覆盖。

## 正式版本上软件的任何修改都应有详细的文档记录。