1.      目的

本规范的目的是使开发人员能以标准的、规范的方式设计和编码，通过建立编码规范，使软件开发人员养成良好的编码风格和习惯，提高程序的可靠性、可读性、可修改性、可维护性和一致性，增进团队间的交流，最终保证软件产品的质量。

2.      范围

本规范适用于C和C++编码的软件开发。

3.      术语定义

规则：编程时强制必须遵守的原则。

建议：编程时必须加以考虑的原则。

说明：对此规则或建议予以必要的解释。

示例：对此规则或建议列举例子。

4.      排版

4.1     **[规则]：采用缩进风格编写，缩进的空格数为4个。**

[说明]：缩进使用空格键，不使用TAB键。避免使用不同编辑器阅读程序时，因TAB键所设置的空格数目不同而造成程序布局不整齐。

在函数或过程的开始、结构的定义及循环、判断等语句中的代码要采用缩进风格。

case语句也要遵从语句缩进要求。

4.2     **[建议]：相对独立的程序块间、变量说明之后必须加空行。**

4.3     **[建议]：较长的语句要分成多行书写。**

[说明]：长语句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]：

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

= stat\_poi[index].occupied;

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

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

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

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

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

{

… // program code

}

for (i = 0, j = 0;

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

i++, j++)

{

… // program code

}

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

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

                 sizeof (\_STAT\_OBJECT));

4.4     **[规则]：不允许把多条短语句写在一行中，即一行只允许写一条语句。**

[示例]：如下例子不符合规范。

rect.length = 0;  rect.width = 0;

应如下书写

rect.length = 0;

rect.width = 0;

4.5     **[规则]：if、for、do、while、case、switch、default等语句独占一行，且if、for、do、while等语句的执行体部分无论多少都要加括号{}。**

[示例]：如下例子不符合规范。

if (pUserCR == NULL) return;

应如下书写：

if (pUserCR == NULL)

{

return;

}

4.6     **[规则]：同一文件中{}的风格应该保持一致。**

4.7     **[建议]：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语句中的代码都要采用如上的缩进方式。**

[示例]：如下例子不符合规范。

for (…) {

… // program code

}

if (…)

{

… // program code

}

void example\_fun( void )

{

… // program code

}

应如下书写。

for (…)

{

… // program code

}

if (…)

{

… // program code

}

void example\_fun( void )

{

… // program code

}

4.8     **[建议]：在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系紧密的立即操作符（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。

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)”!”、”~”、”++”、”�C“、”&”（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++;             // “++”,”�C“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)

4.9     **[规则]：头文件的最后一行为空行，以便消除部分编译器编译时产生的警告。**

5.      注释

5.1     **[建议]：注释的内容要清楚、明了，含义准确，防止注释二义性。**

[说明]：注释的原则是有助于对程序的阅读和理解，注释不宜太多也不能太少，注释内容必须准确、易懂、简洁。

[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，避免不必要的冗余注释。

[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)

5.2     **[规则]：源文件头部应进行注释，列出：版权说明、文件名、修改日志等。**

[示例]：

/\*\*   @file file.h

 \*    @note Hangzhou Hikvision System Technology Co., Ltd. All Right Reserved.

 \*    @brief 这里填写本文件的摘要。

 \*

\*    @author         xxx

 \*    @date            2004/02/14

 \*

\*    @note 下面的note和warning为可选项目

 \*    @note 这里填写本文件的详细功能描述和注解

 \*    @note 历史记录：

 \*    @note V2.0.0  添加了一个导出接口

\*

 \*    @warning 这里填写本文件相关的警告信息

 \*/

5.3     **[规则]：函数头部应进行注释。**

[示例]：

/\* \*  @brief    <这里填写函数的摘要>

 \*    @param c 参数描述.

 \*    @param n 参数描述.

 \*    @return  返回描述

 \*/

5.4     **[建议]：类定义注释。**

[示例]：

/\* \*  @brief 类的摘要。

 \*

 \* 类的细节描述

 \*/

5.5     **[建议]：枚举定义注释。**

[示例]：

/\* \* @brief 枚举说明。

 \*    枚举的细节描述

 \*/

5.6     **[建议]：结构定义注释。**

[示例]：

/\*\*  @brief 结构说明。

 \*

 \* 结构的详细细节

 \*/

5.7     **[建议]：成员注释。注释格式按下列两种示例之一即可。**

[示例一]：

long lValue;     /\*\*< 成员描述，该成员注释对枚举、结构以及类都有效 \*/

[示例二]：

long lValue;     ///< 成员描述，该成员注释对枚举、结构以及类都有效

5.8     **[建议]：宏定义注释。注释格式按下列两种示例之一即可。**

[示例一]：

#define MIN(x,y) ((x)>(y)?(y):(x)) /\*\*< 注释，同成员注释 \*/

[示例二]：

#define MIN(x,y) ((x)>(y)?(y):(x)) ///< 注释，同成员注释

5.9     **[建议]：通过对函数、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释。**

[说明]：函数、变量等命名清晰准确，可增加代码可读性，并减少不必要的注释。

5.10  **[建议]：对变量的定义和分支语句（条件分支、循环语句等）应适当增加注释。**

[说明]：这些语句往往是程序实现某一特定功能的关键，良好的注释有助于维护人员更好地理解程序，有时甚至优于查阅设计文档。

[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，更易读。

[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

5.11  **[建议]：全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数存取该变量以及存取时注意事项等的说明。**

5.12  **[建议]：对数据结构声明（包括数组、结构、类、枚举等）和有物理含义的变量、常量，如果其命名不是充分自注释的，必须加注释。**

5.13  **[建议]：注释应放在被注释代码的上方和右侧，如放于上方的注释则必须与上面的代码用空行隔开。**

5.14  **[规则]：注释与所描述内容的缩进保持一致。**

[说明]：可使程序排版整齐，并方便注释的阅读与理解。

[示例]：如下例子，排版不整齐，阅读稍感不方便。

void example\_fun( void )

{

/\* code one comments \*/

CodeBlock One

        /\* code two comments \*/

CodeBlock Two

}

应改为如下布局。

void example\_fun( void )

{

/\* code one comments \*/

CodeBlock One

/\* code two comments \*/

CodeBlock Two

}

5.15  **[规则]：修改代码同时更新相应的注释，保证注释与代码的一致性。不再有用的注释要删除。**

6.      命名规则

6.1     **[建议]：标识符的命名要清晰、明了，有明确的含义。**

[说明]：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。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;

6.2     **[建议]：命名中若使用特殊约定或缩写，则要有注释说明。**

[说明]：在源文件开头处，应当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。

6.3     **[规则]：变量命名要有具体含义外，还能表明变量类型、数据类型等，禁止取单个字符（如i、j、k…），但允许使用i、j、k等单个字符作为局部循环变量。**

6.4     **[建议]：命名规范必须与所使用的系统风格保持一致，比如采用UNIX的全小写加下划线的风格或大小写混排的方式，不要使用大小写与下划线混排的方式，用作特殊标识如标识成员变量或全局变量的m\_和g\_，其后加上大小写混排的方式是允许的。**

[示例]： Add\_User不允许，add\_user、AddUser、m\_AddUser允许。

6.5     **[建议]：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\_     0

#define \_EXAMPLE\_1\_TEST\_     1

void set\_sls00(BYTE sls);

应改为有意义的单词命名

#define \_EXAMPLE\_UNIT\_TEST\_    0

#define \_EXAMPLE\_ASSERT\_TEST\_  1

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

6.6     **[建议]：在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。**

[说明]：对接口部分的标识符应该有更严格限制，防止冲突。如可规定接口部分的变量与常量之前加上“模块”标识等。

6.7     **[建议]：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 );

7.      表达式和基本语句

7.1     **[规则]：注意运算符的优先级，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。

7.2     **[规则]：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

}

7.3     **[建议]：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;

7.4     **[规则]：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++;     // 这两条语句的功能相当于“ \* stat\_poi ++ += 1; ”

++ stat\_poi;

\*stat\_poi += 1; // 这两条语句的功能相当于“ \* ++ stat\_poi += 1; ”

7.5     **[建议]：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。

7.6     **[规则]：switch语句必须有default分支。**

7.7     **[建议]：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 \*/

7.8     **[建议]：在多重循环中，应将执行最频繁的循环放在最内层。**

[说明]：减少CPU切入循环层的次数，某些场合下（如下面的示例）可提高cache命中率。

[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];

}

}

7.9     **[建议]：尽量减少循环嵌套层次。**

7.10  **[建议]：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;

}

}

7.11  **[建议]：尽量用乘法或其他方法代替除法，特别是在浮点运算中。**

[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 PI 3.1416

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

应如下把浮点除法改为浮点乘法。

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

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

7.12  **[建议]：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。写完代码后，应对这些操作符进行彻底检查。

7.13  **[建议]：编写代码时，要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。修改如下。

int size; // 从unsigned char改为int，若改为char，在ARM平台上可能仍有问题。

while (size-- >= 0)

{

… // program code

}

8.      变量、结构

8.1     **[建议]：尽量避免使用全局变量。**

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

8.2     **[建议]：仔细定义并明确全局变量的含义、作用、取值范围及全局变量间的关系。**

[说明]：在声明变量的同时，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他变量的关系。

8.3     **[建议]：向全局变量传递数据时，要十分小心，防止赋给不合理的值或越界等现象发生。**

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

8.4     **[规则]：严禁局部变量与全局变量同名。**

[说明]：使用合理的命名规则，可最大限度地避免该问题。

8.5     **[规则]：严禁使用未经初始化的变量作为右值。**

[说明]：特别是在C/C++中引用未经赋值的指针，经常会引起程序崩溃。

8.6     **[建议]：使用严格定义的、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。

8.7     **[建议]：结构的功能要单一，是针对一种事务的抽象。不要设计面面俱到、非常灵活的数据结构。**

[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 �C FEMALE; 1 �C MALE \*/

      unsigned char  teacher\_name[8]; /\* teacher’s name \*/

      unsigned char  teacher\_sex;     /\* teacher’s sex \*/

} STUDENT;

若改为如下，可能更合理些。

typedef struct TEACHER\_STRU

{

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

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

                         /\* 0 �C FEMALE; 1 �C 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 �C FEMALE; 1 �C MALE \*/

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

} STUDENT;

8.8     **[建议]：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;

8.9     **[建议]：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;

8.10  **[建议]：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，可节省空间并易于理解。

[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;

8.11  **[建议]：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，是软件产品成功与否的重要标志之一。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。

8.12  **[建议]：编程时，要注意数据类型的强制转换。**

[说明]：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。

8.13  **[建议]：对编译系统默认的数据类型转换，也要有充分的认识。**

[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

8.14  **[建议]：尽量减少没有必要的数据类型默认转换与强制转换，同时合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。**

8.15  **[建议]：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;

8.16  **[规则]：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序、使用的位域及字节对齐等问题 。**

[说明]：在声明结构体时，必须使用#pragma pack(n)/#pragma pack()指定对齐方式。

[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 |     |

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

8.17  **[建议]：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;

8.18  **[建议]：留心程序机器码大小（如指令空间大小、数据空间大小、栈空间大小等）是否超出系统有关限制。**

8.19  **[建议]：修饰符 \* 和 ＆ 应当紧邻类型。**

[示例]

char\* name;

const std::string& str;

9.      函数、过程

9.1     **[建议]：函数的规模尽量限制在100行以内。**

[说明]：不包括注释和空白行。

9.2     **[建议]：函数名应准确描述函数的功能。**

[说明]：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）。避免使用无意义或含义不清的动词为函数命名。

9.3     **[建议]：一个函数仅完成一个功能，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。

9.4     **[建议]：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专门编写函数似乎没有必要，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);

9.5     **[建议]：编写可重入函数时，应注意局部变量的使用。**

[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。

9.6     **[规则]：编写可重入函数时，若使用全局变量，应对其加以保护。**

[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。

9.7     **[建议]：避免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]：下面函数的实现不太好。

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;

}

9.8     **[建议]：检查函数输入的有效性。**

[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。

9.9     **[规则]：函数参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字。**

[示例]：

void SetValue(int width, int height);      // 良好的风格

void SetValue(int, int);                         // 不良的风格

9.10  **[建议]：如果参数是指针，且仅作输入用，则应在类型前加const，以防止该指针在函数体内被意外修改。**

[示例]：

void StringCopy(char \*strDestination， char \* const strSource);

9.11  **[建议]：如果输入参数以值传递的方式传递对象，C++则宜改用“const &”方式来传递，这样可以省去临时对象的构造和析构过程，从而提高效率。**

9.12  **[建议]：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。

9.13  **[建议]：仔细、全面地处理被调用函数的返回值。**

9.14  **[建议]：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局部变量的地址作为返回值。

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，变量sum是static的。

for (index = 1; index <= base; index++)

{

        sum += index;

}

return sum;

}

9.15  **[建议]：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，往往还会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。

9.16  **[建议]：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。

9.17  **[建议]：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。**

[说明]：优化函数结构时，要遵循以下原则：

（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）通过拆分或合并函数来改进软件结构。

（4）考查函数的规模，过大的要进行拆分。

（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。）

9.18  **[规则]：文件句柄的打开与关闭必须配对。**

10. 内存管理

10.1  **[规则]：用malloc或new申请内存之后，应该立即检查指针值是否为NULL。防止引用值为NULL的指针变量。**

[说明]：在C++标准中，new申请内存失败时默认抛出std::bad\_alloc，如果要返回NULL值，可使用std::nothrow进行显示声明。

[说明]：1993年前，C++一直要求在内存分配失败时operator new要返回0，现在则是要求operator new抛出std::bad\_alloc异常。很多C++程序是在编译器开始支持新规范前写的。C++标准委员会不想放弃那些已有的遵循返回0规范的代码，所以他们提供了另外形式的operator new以继续提供返回0功能。这些形式被称为“无抛出”，因为他们没用过一个throw，而是在使用new的入口点采用了nothrow对象：

class widget { … };

widget \*pw1 = new widget;// 分配失败抛出std::bad\_alloc

if (pw1 == 0) …    // 这个检查一定失败

widget \*pw2 = new (nothrow) widget;       // 若分配失败返回0

if (pw2 == 0) …    // 这个检查可能会成功

不管是用“正规”(即抛出异常)形式的new还是“无抛出”形式的new，重要的是你必须为内存分配失败做好准备。

10.2  **[建议]：避免数组或指针的下标越界，特别要当心发生“多1”或者“少1”操作。**

10.3  **[规则]：动态内存的申请与释放必须配对，防止内存泄漏。**

10.4  **[规则]：用free或delete释放了内存之后，立即将指针设置为NULL，防止产生“野指针”。**

11. 宏

11.1  **[规则]：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))

11.2  **[规则]：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，即只执行了一次增1。

12. STL：容器

12.1  **[建议]：用vector和string代替数组。**

[说明]：当今软件中缓冲区溢出和安全缺陷是罪魁祸首，造成这一现象的主要原因就是固定长度的数组，因此尽量不要使用C语言风格的数组、指针运算和内存管理原语操作实现数组抽象，使用vector或者string不仅更轻松，而且还有助于编写更安全、伸缩性更好的软件。

12.2  **[规则]：使用vector和string::c\_str与非C++ API交换数据。**

[说明]：vector的存储区总是连续的，因此与C语言API使用字符数组指针转换数据时，不用担心访问区间被截断问题。

12.3  **[建议]：在容器中只存储值和智能指针。**

12.4  **[建议]：用push\_back代替其他扩展序列的方式。**

[说明]：如果不需要操心插入位置，就应该使用push\_back在序列中添加元素，其他方法可能极慢而且不简明。

12.5  **[建议]：多用范围操作，少用单元素操作。**

[说明]：调用范围操作通常更容易编写，也更易于阅读，而且比显式循环的效率更高。

13. STL：算法

13.1  **[建议]：使用带检查的STL实现。**

[说明]：与指针错误一样，迭代器也很容易出现错误。需加强对下列错误的检查：

使用已失效的或未初始化的迭代器。

传递越界索引：例如，一个容器只有100个元素，却要访问它的第113个元素。

使用并非真是“范围”的迭代器范围：传递的两个迭代器中，第一个并不在第二个以前，或者两个所指向的并非同一个容器。

传递无效的迭代器位置：调用一个以迭代器位置（比如传给insert的位置）为参数的容器成员函数时，传递的迭代器却指向了另一个容器。

使用无效顺序：为关联容器排序，或作为排序算法的比较依据，所提供的是无效排序规则。

13.2  **[建议]：用算法调用代替手工编写的循环。**

[说明]：编写算法调用代替手工编写的循环，可以使表达力更强、维护性更好、更不易出错，而且同样高效。调用算法时，应该考虑编写自定义的函数对象以封装所需的逻辑。

13.3  **[建议]：使谓词成为纯函数。**

[说明]：谓词就是返回是或否（返回值通常为bool类型）的函数对象。

算法会以不可知的顺序、不可知的次数生成不可知数量的谓词副本，然后将这些 四处传递，而且会很漫不经心地认为这些副本都是等价的，因此我们需要负责确保谓词副本确实等价，也就是说它们必须是纯函数。

13.4  **[建议]：算法和比较器的参数应多用函数对象少用函数。**

[说明]：关联容器的比较器必须是函数对象，函数对象的适配性好，而且产生的代码一般比函数快。

14. 类的设计

14.1  **[规则]：如果类A和类B毫不相关，不可以为了使B的功能更多些而让B继承A的功能和属性。**

14.2  **[规则]：若在逻辑上B是A的“一种”，并且A的所有功能和属性对B而言都有意义，则允许B继承A的功能和属性。**

[示例]：从生物学角度讲，鸵鸟（Ostrich）是鸟（Bird）的一种，按理说类Ostrich应该可以从类Bird派生。但是鸵鸟不能飞，那么Ostrich::Fly是无意义的，所以类Ostrich不应该从类Bird派生。

class Bird

{

public:

       virtual void Fly(void);

…

};

class Ostrich : public Bird

{

…

};

14.3  **[规则]：若在逻辑上A是B的“一部分”（a part of），则不允许B从A派生，而是要用A和其他类组合出B。**

[示例]：眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类Head应该由类Eye、Nose、Mouth、Ear组合而成，不是派生而成。

14.4  **[建议]：用小类代替巨类**

[说明]：小类更易于编写，更易于保证正确、测试和使用。小类更有可能适用于各种不同情况。应该用这种小类体现简单概念，不要用大杂烩式的类，它们要实现的概念既多又复杂。

14.5  **[建议]：构造函数的初始化**

[说明]：如果类存在继承关系，派生类必须在其初始化表里调用基类的构造函数。类的const常量只能在初始化表里被初始化，因为它不能在函数体内用赋值的方式来初始化。类的数据成员的初始化可以采用初始化表或函数体内赋值两种方式，非内部数据类型的成员对象应当采用第一种方式初始化，以获取更高的效率。

14.6  **[规则]：以同样的顺序定义和初始化成员变量**

[说明]：成员变量初始化的顺序要与类定义中的声明的顺序始终保持一致；不用考虑构造函数初始化列表中编写的顺序。要确保构造函数代码不会导致混淆地指定不同的顺序。

[示例]：

class Employee

{

public:

       Employee(const char\* firstName, const char\* lastName)

       : firstName\_(firstName), lastName\_(lastName), email\_(firstName\_ + “.” + lastName\_ + “@acme.com”) {}

private:

string email\_;

string firstName\_;

string lastName\_;

};

上面这段代码隐藏着一个错误，危害很大，而且很难发现。因为类定义中email\_是在firstName\_和lastName\_之前被声明的，它将首先被初始化，然后试图使用其他未初始化的字段。

14.7  **[建议]：派生类中实现类的虚函数**

[说明]：派生类的构造函数应在其初始化表里调用基类的构造函数。基类与派生类的析构函数应该为虚，即加virtual关键字，在编写派生类的赋值函数时，注意不要忘记对基类的数据成员重新赋值。

14.8  **[规则]：避免在构造函数和析构函数中调用虚拟函数**

[说明]：从构造函数或析构函数直接或间接调用未实现的纯虚拟函数，会导致未定义的行为。

14.9  **[建议]：显式地禁止复制和赋值**

[说明]：在下述三种复制行为之间谨慎选择――使用编译器生成的复制构造函数和赋值操作符；编写自己的版本；如果不应允许复制，显式地禁用前两者。

[示例]：显示地禁止复制和赋值

class T

{

private:                                       // 使T不能复制

       T(const T&);                       // 未实现

       T& operator=(const T&);     // 未实现

}

或从下列类继承

class noncopyable

{

protected:

       noncopyable() {}

       ~noncopyable() {}

private:  // the following members are private

       noncopyable(const noncopyable&);

       const noncopyable& operator=(const noncopyable&);

};

14.10   **[建议]：避免提供隐式转换**

[说明]：隐式转换所带来的影响经常是弊大于利。在为自定义类型提供隐式转换之前，请三思而行，应该依赖的是显示转换（explicit构造函数和命名转换函数）。

[示例]：默认时，为单参数构造函数加上explicit

class Widget

{

**explicit** Widget(unsigned int widgetizationFactor);

**explicit** Widget(const char\* name, const Widget\* other = 0);

}

14.11   **[建议]：将数据成员设置为私有的，无行为的聚集（C语言形式的struct）除外**

[说明]：简单的C语言形式的struct类型只是将一组值聚集在一起，并不封装或者提供行为，只有在这种struct类型中才可以将所有数据成员都设成公用的。要避免将公用数据和非公用数据混合在一起，因为这几乎总是设计混乱的标志。

15. 名字空间与模块

15.1  **[建议]：将类型及其非成员函数接口置于同一名字空间中。**

[说明]：公有成员函数和非成员函数都是类的公有接口的组成部分。对于一个类X而言，所有在同一个名字空间中“提及”X和“随”X“一起提供的”函数（包括非成员函数）逻辑上都是X的一部分，因为它们形成了X接口的组成部分。

15.2  **[建议]：应该将类型和函数分别置于不同的名字空间中，除非有意想让它们一起工作。**

[说明]：通过将类型以及与其直接相关的非成员函数置于自己单独的名字空间中，可以使类型与无意的ADL（参数依赖查找）隔离开来，促进有意的ADL，不仅可以避免追踪代码中难以辨识的错误，而且还可以避免了解那些难以理解的语言细节。

15.3  **[规则]：不要在头文件中的#include之前编写名字空间using。**

15.4  **[建议]：要避免在不同的模块中分配和释放内存。**

[说明]：在一个模块中分配内存，而在另一个模块中释放它，会在这两个模块之间产生微妙的远距离依赖，使程序变得脆弱，因此要尽最大可能在同一模块或者子系统中分配和释放内存。

15.5  **[建议]：不允许异常跨越模块边界传播。**

[说明]：C++异常处理没有普遍通用的二进制标准，因此不要在两段代码之间传播异常，因为模块可能无法支持可兼容地实现异常传播。

15.6  **[规则]：不要特化函数模板。**

[说明]：在扩展其他人的函数模板时，要避免常识编写特化代码；相反，要编写函数模板的重载，将其放在重载所用的类型的名字空间中。编写自己的函数模版时，要避免鼓励他人直接特化函数模板本身。

16. 可测性

16.1  **[建议]：在同一项目组或产品组内，要针对集成测试与系统联调准备一套统一的调测开关及相应打印函数，并且提供详细说明。**

[说明]：本建议是针对项目组或产品组的。

16.2  **[建议]：在同一项目组或产品组内，调测打印出的信息串要有统一的格式。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。

16.3  **[建议]：在编写代码之前，应预先设计好程序调试与测试的方法和手段，并设计好各种调测开关及相应测试代码如打印函数等。**

[说明]：程序的调试与测试是软件生命周期中非常重要的一个阶段，如何对软件进行较全面、高率的测试并尽可能地找出软件中的错误就成为关键的问题。因此在编写源代码之前，除了要有一套比较完善的测试计划外，还应设计出一系列代码测试手段，便于单元测试、集成测试及系统联调。

16.4  **[建议]：调测开关应分为不同级别和类型。**

说明：调测开关的设置及分类应从以下几方面考虑：针对模块或系统某部分代码的调测；针对模块或系统某功能的调测；出于其他某种目的，如对性能、容量等的测试。这样做便于软件功能的调测，并且便于模块的单元测试、系统联调等。

16.5  **[建议]：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（通过调测开关）。**

[说明]：为单元测试而准备。

16.6  **[建议]：调试时使用断言来发现软件问题，提高代码可测性。**

[说明]：断言是对某种假设条件进行检查（可理解为若条件成立则无动作，否则应报告），可以快速发现并定位软件问题，同时对系统错误进行自动报警。断言可以对在系统中隐藏很深，用其他手段极难发现的问题进行定位，从而缩短软件问题定位时间，提高系统的可测性。实际应用时，可根据具体情况灵活地设计断言。

示例：下面是C语言中的一个断言，用宏来设计的。（其中NULL为0L）

#ifdef \_EXAM\_ASSERT\_TEST\_  // 若使用断言测试

void exam\_assert( char \* file\_name, unsigned int line\_no )

{

printf( “\n[EXAM]Assert failed: %s, line %u\n”, file\_name, line\_no );

abort( );

}

#define  EXAM\_ASSERT( condition )

if (condition) // 若条件成立，则无动作

        NULL;

else  // 否则报告

        exam\_assert( \_\_FILE\_\_, \_\_LINE\_\_ )

#else  // 若不使用断言测试

#define EXAM\_ASSERT(condition)  NULL

#endif  /\* end of ASSERT \*/

16.7  **[建议]：正式软件产品中应把断言及其他调测代码去掉（即把有关的调测开关关掉）。**

[说明]：加快软件运行速度。

16.8  **[规则]：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。**

16.9  **[规则]：不要用断言来检查最终产品肯定会出现且必须处理的错误情况。**

[说明]：断言是用来处理不应该发生的错误情况的，对于可能会发生的且必须处理的情况要写防错程序，而不是断言。如某模块收到其他模块或链路上的消息后，要对消息的合理性进行检查，此过程为正常的错误检查，不能用断言来实现。

16.10   **[建议]：编写防错程序，然后在处理错误之后可用断言报告错误。**

[示例]：假如某模块收到通信链路上的消息，则应对消息的合法性进行检查，若消息类别不是通信协议中规定的，则应进行出错处理，之后可用断言报告，如下例。

#ifdef \_EXAM\_ASSERT\_TEST\_ // 若使用断言测试

/\* Notice: this function does not call ‘abort’ to exit program \*/

void assert\_report( char \* file\_name, unsigned int line\_no )

{

printf( “\n[EXAM]Error Report: %s, line %u\n”, file\_name, line\_no );

}

#define  ASSERT\_REPORT( condition )

if ( condition ) // 若条件成立，则无动作

        NULL;

else // 否则报告

        assert\_report ( \_\_FILE\_\_, \_\_LINE\_\_ )

#else // 若不使用断言测试

#define ASSERT\_REPORT( condition )  NULL

#endif /\* end of ASSERT \*/

int msg\_handle( unsigned char msg\_name, unsigned char \* msg )

{

switch( msg\_name )

{

        case MSG\_ONE:

            … // 消息MSG\_ONE处理

            return MSG\_HANDLE\_SUCCESS;

            … // 其他合法消息处理

        default:

            … // 消息出错处理

            ASSERT\_REPORT( FALSE );  // “合法”消息不成立，报告

            return MSG\_HANDLE\_ERROR;

}

}

17. 补充

17.1  **[建议]：全局变量使用static定义，访问全部变量使用get()、set()，尽量避免使用extern。**

17.2  **[建议]：在C++中推荐使用const定义常量。**

17.3  **[规则]：指针判断是否为空，建议直接与“NULL”比较，而不是使用“！”。**

[示例]：

if (pUserCR == NULL)

{

return;

}

而不是：

if (!pUserCR)

{

return;

}

17.4  **[规则]：路径使用斜线“/”，而不是反斜线“\”,“/”能被不同的平台识别**

[示例]：

使用

#include “./NetPreview/PreviewEngine.h”

而不是

#include “.\NetPreview\PreviewEngine.h”

17.5  **[建议]：不允许包含无用的头文件，不要重复包含头文件。**

17.6  **[建议]：仅在一个C/CPP文件中使用的结构可以不放在头文件中，直接放在C/CPP文件中。**