内部文件

安徽科大讯飞信息科技有限公司
Anhui USTC iFLYTEK CO., LTD.

平嵌内核研发部
C/C++语言编程规范

版本：1.0

安徽科大讯飞信息科技有限公司
地址：中国安徽省合肥市高新区信息产业基地讯飞语音大厦
电话：86-551-5331800　网址：HTTP://WWW.IFLYTEK.COM
传真：86-551-5331801　邮件：IFLYTEK@IFLYTEK.COM

版本历史

版本	描述	作者
1.0	初始版本，将平台与嵌入式两套标准融合	雷琴辉
1.1	进一步完善版本，增加了枚举类型的作用域	雷琴辉

目　录

1. 概述

1.1 背景

随着信息时代的飞速发展，应用软件的规模越来越大，代码数量越来越多，现在的项目开发基本上都是很多程序员共同完成的。在软件工程领域，源程序的风格统一与否直接影响着软件的可维护性、可读性的好坏，以及日后培训和交流的难易程度，继而对软件开发成本有着直接的关系。

C/C++语言是开发内核引擎的主要工具,然而 C/C++语言由于其灵活性，特别是 VS 开发工具的包容性，从而使得很多不规范代码能够在 Windows 下编译通过,但是在其它相当多的平台上，其编译更为苛刻，更加遵循 C/C++的 ISO 标准；另外，如果不能按照很好的规范来使用，也会带来混乱、低效或维护问题。

1.2 目的

- ✧ 增强代码的可读性、可维护性、可移植性。
- ✧ 使开发人员具备跨平台开发的基本理念。

1.3 原则

- ✧ 在保证代码正确性的基础上，使代码简单、清晰、有效。
- ✧ 保证代码风格在内核产品中的一致性。
- ✧ 在面向嵌入式终端产品中禁用所有 C++代码，编写安全 C 代码，使内核开发少走弯路。
- ✧ 所有引擎开发新增模块，建议一律采用 C 代码实现。
- ✧ 所有引擎代码中，涉及到系统相关的部分，需要集中体现，并考虑不同平台下可移植性。

1.4 工作环境

- ✧ 编程工具：VC6.0 (英文版) + VA6.0 + Kernel.DSM，VS2005(英文版)+ VA-X，项目组可选。
- ✧ 工作平台：Starteam，另外还须将 SCC 接口集成到 VC 中，或者 svn，根据项目配置而定。

1.5 补充说明

- ✧ 本代码规范的适用范围为所有内核产品，使用者为所有内核研发人员。对于产品的配套工具以及针对 PC 上的应用开发，项目组长可以适当放宽此规范的限制或采用其他规范。
- ✧ 特别说明：请使用项目创建时中已定义的修饰符、数据类型、资源地址及尺寸类型等，禁止使用标准 C/C++的通用关键字和编译器内置的数据类型（在面向嵌入式设备的项目中，请使用 ivDefine.h 中定义的数据类型）。

2. 注释规范

2.1 注释量

注释是源码程序中非常重要的一部分，通常源程序有效注释量必须在15%以上，但务必精简，因此注释超过30%，就有画蛇添足之嫌疑。

2.2 注释语言

推荐用英文注释，受英文水平限制实在无法表述之时，可以使用中文。

2.3 注释形式

C 语言支持的注释形式为 /**/，而 C++对此进行了扩展，可以使用//对每行注释。我们规定：在 C 代码中禁止使用//注释，全部采用/* */形式，且被注释内容前后应各留 1 个空格；C++代码中可以使用//注释，但是//之后需要留一个空格；注释内禁止嵌套注释，以防编译出错。示例如下：

```
// 注释          仅 C++代码中允许
/*注释*/             错误
/* / * 注释 * / */ 错误
/* 注释 */       正确
```

2.4 注释位置

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。不应在代码或表达中间插入注释。注释与所描述内容进行同样的缩排，以方便注释的阅读与理解。示例如下：

```
如下例子，注释不整齐，阅读不方便。
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;
}
```

2.5 命名与声明的注释

对于有物理含义的变量、常量和宏定义，若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。示例如下：

```
#define LTS_COST    255      /* 词典词的词频最低门限 */
ivUInt8 nEtymaLen;           /* 词根部分长度（注意不是原形的!!!） */
```

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，也必须加以注释。

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。

```
/* EBook interface with EBook user primitive message name */
enum EBOOK_USER_PRIMITIVE
{
    N_UNITDATA_IND,     /* EBook notify EBook user unit data come */
    N_NOTICE_IND,       /* EBook notify user the No.7 network can not transmission */
    N_UNITDATA_REQ      /* EBook user's unit data transmission request */
};
```

2.6 文件描述

所有包含代码的文件必须在头部详尽描述本文件相关的信息，如文件名称，所属模块，创建人，创建时间，历史记录等。只要能体现上述信息，具体风格可以有程序员自定。

```
/* ********************************************************************************#
* 文件名     : SplitWord.cpp
* 文件功能   : 分词模块源文件
* 作者       : Linxi
* 创建时间   : 2006年5月15日
* 项目名称   : EnFrontEndFrame
*--------------------------------------------------------------------------------#
* 历史记录：
* 日期        作者        备注
*--------------------------------------------------------------------------------#
* 2008-3-13   jwgao       外部使用音节划分函数
* 2008-4-24   Linxi       解决Mr. Hu's
********************************************************************************* */
```

示例一：符合匈牙利方式的描述

```
/**
* @file     文件名
* @brief    简单说明
*
* detail...    详细说明
*
* @authoryigao
* @version  1.0
* @date     2004年4月22日
*
* @see   参考主题（超链接）
*
* @par      版本记录：
* <table border=1>
*  <tr> <th>版本   <th>日期                <th>作者 <th>备注 </tr>
*  <tr> <td>1.0   <td>2004年4月22日     <td>yigao  <td>创建 </tr>
* </table>
*/
```

示例二：符合 UNIX 的描述方式

2.7 函数描述

每一个不是自动生成的函数，都必须详细说明函数名称，函数功能，作者，创建时间，参数，返回值等。

```
/* ********************************************************************************#
* 函数名称   : PEGenSyllable
* 函数功能   : General Syllable List by phoneme list
* 作者       : Truman
* 创建时间   : 2006年6月8日
* 返回值     : ivUInt32 : Syllable Count
* 参数       :
  @ PSylInfo pOptSyl[out] : To recieve the Syllable result
  @ ivPCUInt8 pPhoneID : Input, ID of the Phoneme List
  @ ivUInt32 nPhone : Phoneme Count
  @ ivCStrA pStressInfo : the Stress info, Only tag at Vowel
******************************************************************************** */
```

2.8 算法描述

典型算法（由项目组长定义）必须加描述性注释。一般选用介于自然语言和程序语言之间的伪语言，清晰、简洁而精确地说明计算过程。举例如下：

```
/* ********************************************************************************#
* 算法名称   : FNevChebP
* 算法功能   : Evaluate a series expansion in Chebyshev polynomials
* 作者       : zhling
* 创建时间   : 2002年7月6日
* 过程描述   :
*   Consider the backward recursion
*     b(i,x) = 2xb(i+1,x) - b(i+2,x) + c(i),
*   with initial conditions b(n,x)=0 and b(n+1,x)=0.  Then dropping the
*   dependence on x,
*     c(i) = b(i) - 2xb(i+1) + b(i+2).
*
*          n-1
*   Y(x) = SUM c(i) T(i)
*          i=0
*
*          n-1
*        = SUM [b(i)-2xb(i+1)+b(i+2)] T(i)
*          i=0
*                                          n-1
*        = b(0)T(0) + b(1)T(1) - 2xb(1)T(0) + SUM b(i) [T(i)-2xT(i-1)+T(i-2)]
*                                          i=2
* The term inside the sum is zero because of the recursive relationship
* satisfied by the Chebyshev polynomials.  Then substituting the values T(0)=1
```

```
* and T(1)=x, Y(x) is expressed in terms of the diff. between b(0) and b(2)
* (errors in b(0) and b(2) tend to cancel),
*   Y(x) = b(0)-xb(1) = [b(0)-b(2)+c(0)] / 2
********************************************************************************* */
```

2.9 类注释

类注释通常放在其对应的文件中，位于类声明之前。示例如下：

```
/**
* @class 类名称
*
* @brief
*
* detail...
*
* @authoryigao
* @date  2004年4月22日
*
* @see
*
* @par 备注:
*
*/
```

2.10 类成员注释

① 数据成员和成员函数，数据成员在声明的头文件中进行文档注释，成员函数在其实现体前添加文档注释，大多位于.cpp文件。示例如下：

```
/**
* @brief    简单说明
*
* detail...    详细说明
*
* @authoryigao
* @date     2004年4月16日
* @returnvoid
* @param
* @see
* @exception
*/
```

② 对于复杂的数据成员，需要详细说明的采用如下的形式注释。示例如下：

```
  /**
```

```
 * @brief  简单说明
 *
 * detail...  详细说明
 */
 m_Data;
```

③ 对于简单的数据成员可采用如下的单行注释。示例如下：

```
/// 说明
m_Data;
```

2.11 其它注释

包括全局变量/常量，宏定义，全局函数等，遵循类成员注释规范，在其声明之前添加文档注释，同样要求将全局函数的文档注释添加在其实现体之前。

```
/**
* @brief 简单说明
*
* detail...详细说明
*/
#define MAX(a,b) (((a)>(b))?(a):(b))

/// 注释说明
static const int MAX_SIZE = (1024);
```

3. 排版规范

3.1 程序块排版

程序块要采用缩进风格编写。把源程序中的**Tab字符转换成4个空格**（在VC设置中可选择用空格填充Tab字符），一个缩进等级（Indentation Level）是4个空格；变量定义和可执行语句要缩进一个等级；函数的参数过长时，也要缩进。

同时，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语句中的程序也都要采用如上的缩进方式。

另外，程序块的分界符（如大括号‘{’和‘}’）应各独占一行并且位于同一列，而且要与引用它们的语句左对齐。示例如下：

```
for (...)
{
    if (...)
    {
        Program Code
    }
}
```

3.2 留空和换行

① 二元的数学和逻辑运算符号（双目运算符）两边都加上一个空格，如：a += (- b) * c;
② 逗号、分号只在后面加空格。
③ 单目操作前后不加空格，"->"、"."前后不加空格。
④ if、while、for、switch、分号、逗号与其后的代码之间加一个空格，如：

```
for ( i = 0; i <= 10; i++ )
while ( i > pProject->Find( chFind, 0 ) )
if ( dwPos && iIndex )
```

⑤ 一行代码的字符个数应在 80 之内，以便于程序的阅读和输出。
⑥ 不允许把多个短语句写在一行中，即一行只写一条语句，如：

```
pL3Pos[1] = 0; pL3Pos[2] = 0;    /* 错误 */

pL3Pos[1] = 0;                   /* 正确 */
pL3Pos[2] = 0;
```

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{}，如：

```
if( nSylLeft <= 16 )
{
    return nSylLeft;
}
```

⑧ 文件之中不得存在无规则的空行，比如说连续十个空行，一般来讲函数与函数之间的空行为 2 行。但在源代码文件末尾必须留一个空行，相对独立的程序块之间、变量说明之后也必须加空行。

4. 命名规范

由于历史原因内核研发部存在匈牙利命名规范和 UNIX 命名规范。在评测组内部，两种规范都符合标准，但是最终倾向与统一到 UNIX 命名规范。在新开工项目中，如果新代码没有超过整个项目的 70%，代码风格务必继承原有代码风格，如果超过 70%可以综合考虑是否统一为 UNIX 风格。

4.1 匈牙利命名规范

4.1.1 变量命名

① 变量名必须简练，长度不要超过 20 个字符。
② 变量名必须容易记忆，尽量使用名词为主的完整单词（或单词组合），也可以使用常用缩写，但是工程中相同缩写的意义必须相同。
③ 禁止取单个字符（如 i、j、k...）为变量命名，但 i、j、k 作局部循环变量是允许的。
④ 常数名(#define 定义)使用大写字母和下划线的组合，例如：

```
#define MAX_USER_LEMMA_SIZE  32       /* 用户词在资源中占的最大 ivUInt16 数 */
```

⑤ 采用匈牙利命名法和大小写混排方式，变量的取名式为：<scope>_<prefix>< qualifier>。

⑥ 作用域<scope>可以取下表中的值：

作用域	描述	示例
m_	成员变量	m_pDoc, m_nCustomers
l_	局部变量（可以不加）	l_iIndex
g_	全局常量	g_psChsSM_zcs

⑦ 前缀<prefix>可以取下表中的值：

前缀	类型	描述	示例
ch	ivChar	8位字符类型	chGrade
by	ivByte	8位内存单元	byAccent
b	ivBool	布尔型	bEnabled
n	ivInt8/ivUInt8/ivInt16…	各种整型变量	nLength
p	ivPointer	任意类型的指针	pDoc
sz	ivStr	字符串类型	szFileName

⑧ 结构和联合类型名以 tag 作前缀，对于程序中的常用的结构，应当使用 typedef 定义，举例如下：

```
typedef struct tagAccentMode {
  ivCStrA szModeName;          /* 轻重读模式名称 */
  ivCharA chAccentPos;             /* 重读音节 */
  ivCharA chAccentPoS2;            /* 重读音节 */
  ivCharA chSecondaryAccent;       /* 次重读音节 -1表示没有 */
}TAccentMode, *PAccentMode;
```

4.1.2 函数命名

① 函数名必须简练，长度不要超过 20 个字符。

② 函数名应该能体现该函数完成的功能，一般采用动词+名词的形式，每个单词的第一个字母大写。

③ 关键部分应该使用完整单词，辅助部分若太长可采用缩写，缩写应符合英文的规范。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，如 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 等。

4.1.3 文件/文件夹命名规范

文件/文件命名方法与变量命名相同，注意在文件引用的时候，一定要区分文件名大小写。比如文件名为“EOTSearhc.h”，如果在某一个文件中引用该头文件，务必大小写完全一致。

4.2 UNIX 命名规范

4.2.1 文件与文件夹

所有字母小写，各单词之间用下划线分隔，如“score_mapping.h”。

4.2.2 类与结构体

① 类名：所有字母小写，各单词之间用下划线分隔；如“class score_mapping”；实现类需要在最后加上“_impl”。
② 成员函数：所有字母小写，各单词之间用下划线分隔；如“void score_mapping()”。
③ 成员变量：所有字母小写，各单词之间用下划线分隔，最后一个单词后加下划线；如“ivFloat32 cur_score_”。

4.2.3 枚举

① 枚举名：所有字母小写，各单词之间用下划线分隔；如“enum police_weekday”。
② 枚举变量：所有字母小写，各单词之间用下划线分隔；如“sun”。
③ 枚举变量作用域：枚举变量访问无需指定枚举名称，因此使用枚举变量时谨防符号重定义；比如如下代码是错误的。

```
int sun = 1;                        /* 与枚举变量sun冲突 */
#define   mon     (1)               /* 与枚举变量mon冲突 */
enum police_weekday {sun, mon, tue, wed, thu, fri};
```

4.2.4 宏与常量

① 所有字母大写，各单词之间用下划线分隔。
② 宏所展开后的表达式务必用小括号括起来。
③ 如“#define SE_MAX(a,b)   ((a)>(b)?(a),(b))”

4.2.5 函数

① 接口函数：沿用现有引擎接口风格，基本上采用类似匈牙利方法，比如 SE/AiET 引擎分辨采用“iSEInitialize( )”,“ivAiET_Create( )”两种固定风格。
② 其余函数：所有字母小写，各单词之间用下划杠分开，比如“void score_mapping()”。

4.2.6 变量

① 所有字母小写，单词之间用下划线分隔。
② 全局变量以“g_”开头。
③ 需要注意局部变量与函数参数的冲突。
④ 比如“ivFloat32 g_cur_score”。

4.2.7 命名空间

① 所有字母小写，词间加下划线，如“namespace scr_map”;
② 域名空间需要在相应的“readme.txt”中进行说明。

4.2.8 其它注意事项

① 保持统一的命名风格。
② 除局部循环变量，其余禁止使用 i，j，k 等单个字符命名。
③ 除非必要，否则避免使用数字命名。
④ 采用准确的单词和众所周知的缩写。

5. 编程规范

5.1 变量

① 在使用变量之前一定要将其初始化。
② 避免使用全局变量（包括 global 和 static 的变量），**在面向嵌入式设备的引擎开发中禁止使用全局变量（包括 global 和 static 的变量）**。
③ 避免局部变量中定义大的结构体、联合体变量和数组，以防止栈溢出，**在面向嵌入式设备的引擎开发中禁止在局部变量中定义大的结构体、联合体变量和数组，以防止栈溢出**。
④ 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。
⑤ 当向公共变量传递数据时，要十分小心，若有必要应进行合法性检查，提高代码的可靠性、稳定性，防止赋予不合理的值或越界等现象发生。
⑥ 使用变量时要注意其取值范围，防止越界。

5.2 常量与预编译宏

① 代码中一般不要直接使用常量数字或常量字符串，而应该用 const/ivConst 或者宏定义，统一管理。
② 尽量避免字符串常量中出现非 ASCII 字符，**在面向嵌入式设备的引擎开发中禁止在字符串常量中出现汉字等非 ASCII 字符**。
③ 在定义宏时，如果包含参数或运算，要使用完备的括号，示例如下：

```
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))
```

④ 将宏所定义的多条表达式放在大括号中，示例如下：

```
#define INIT_RECT_VALUE( a, b )\
{\
    a = 0;\
    b = 0;\
}
```

⑤ 使用宏时，不允许参数发生变化，示例如下：

```
如下用法可能导致错误：
#define SQUARE( a ) ((a) * (a))

int a = 5;
int 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 */
```

⑥ 为了避免重复包含，头文件需要使用预编译宏。示例如下

```
#ifndef _SESSION_MANAGER_H_
#define _SESSION_MANAGER_H_

/* 这里声明你的全局函数、类等 */

#endif   /* _SESSION_MANAGER_H_ */
```

5.3 结构体

① 结构体的功能要单一，是针对一种事务的抽象。结构体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体中。示例如下：

```
如下结构体不太清晰、合理：
typedef struct STUDENT_STRU
{
    unsigned char szName[8]; /* student's name */
    unsigned char byAge;     /* student's age */
    unsigned char bSex;      /* student's sex, as follows */
                             /* 0 - FEMALE; 1 - MALE */
    unsigned char szTeacherName[8];  /* the student teacher's name */
    unisgned char bTeacherSex;       /* his teacher sex */
} TSTUDENT;

若改为如下，更合理些。
typedef struct TEACHER_STRU
{
    unsigned char szName[8]; /* teacher name */
    unisgned char bSex;      /* teacher sex, as follows */
                             /* 0 - FEMALE; 1 - MALE */
} TTEACHER;

typedef struct STUDENT_STRU
{
    unsigned char szName[8];     /* student's name */
    unsigned char byAge;         /* student's age */
    unsigned char bSex;          /* student's sex, as follows */
                                 /* 0 - FEMALE; 1 - MALE */
    unsigned int  nTeacherIndex;  /* his teacher index */
} TSTUDENT;
```

② 不同结构体间的关系不要过于复杂。若两个结构体间关系较复杂、密切，那么应合为一个结构体。

③ 结构体中元素的个数应适中。若结构体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体，以减少原结构体中元素的个数。这样能增加结构体的可理解性、可操作性和可维护性。

示例如下：

```
typedef struct PERSON_BASE_INFO_STRU
{
    unsigned char szName[8];
    unsigned char byAge;
    unsigned char bSex;
} PERSON_BASE_INFO;

typedef struct PERSON_ADDRESS_STRU
{
    unsigned char szAddr[40];
    unsigned char szCity[15];
    unsigned char byTel;
} PERSON_ADDRESS;

typedef struct PERSON_STRU
{
    PERSON_BASE_INFO tPersonBase;
    PERSON_ADDRESS tPersonAddr;
} PERSON;
```

④ 仔细设计结构体中元素的布局与排列顺序，使结构体容易理解、节省占用空间，并减少引起误用现象。示例如下：

```
如下结构体中的位域排列，将占 6 字节空间（设最小内存单元为 2 字节），可读性也稍差。
typedef struct EXAMPLE_STRU
{
    unsigned char byValid;   1;
    PERSON person;           2;
    unsigned char bySetFlg;  1;
} EXAMPLE;

若改成如下形式，不仅可节省 2 字节空间，可读性也变好了。
typedef struct EXAMPLE_STRU
{
    unsigned char byValid;   1;
    unsigned char bySetFlg;  1;
    PERSON person;           2;
} EXAMPLE;
```

⑤ 设计对外接口中的结构体时，必须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）。设计内部使用的结构体时遵循效率优先原则，但也要尽量考虑周全。

5.4 枚举

① 枚举类型和枚举变量要求意义明确，不容易与其它符号产生混淆。

② 枚举变量作用域：枚举变量访问无需指定枚举名称，因此使用枚举变量时谨防符号重定义；比如如下代码是错误的。

```
int sun = 1;                    /* 与枚举变量sun冲突 */
#define   mon     (1)           /* 与枚举变量mon冲突 */
enum police_weekday {sun, mon, tue, wed, thu, fri};
```

③ 枚举类型的使用：在 C 语言当中，在使用枚举类型定义枚举对象时，必须显示申明枚举类型。

```
int main()
{
  enum police_weekday {sun, mon, tue, wed, thu, fri};  /* 定义枚举类型 */
  police_weekday my_day = sun; /* gcc 编译不通过 */
  enum police_weekday his_day = mon;

  printf("size police_weekday = %d\n",sizeof(police_weekday)); /* gcc 编译不通过 */
  printf("size enum police_weekday = %d\n",sizeof(enum police_weekday));
  printf("my_day = %d\n", my_day); /* gcc 编译不通过 */
  printf("his_day = %d\n", his_day);
  return 0;
}
```

5.5 函数

① 函数具有单一而又完整的功能，长度尽量控制在 200 行左右。

② 局部函数一律要在代码开头处（宏定义以下、实现函数以上）统一声明，一行一个。

③ 明确函数参数中指针参数的[in]、[out]或[in/out]属性，并且在注释中写明。

④ 函数参数、返回值都不得直接使用大的结构（可以使用结构的指针），并且要对输入参数做合法性检查，对非参数输入的有效性也要做严格检查，如数据文件格式、公共变量等。但需要说明的是：合法性检查容易产生冗余降低效率，添加检查代码时需要权衡函数在项目中调用的频次。

⑤ 如果函数的输入参数在函数内部不会修改，声明为 ivConst，防止该参数被错误更改。而对于函数的输出参数，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。示例如下：

下列函数将参数作为工作变量，严重不推荐：

```
void SumData( unsigned int num, int *pData, int *sum )
{
    unsigned int nCount;

    *sum = 0;
    for (nCount = 0; nCount < num; nCount ++)
    {
        *sum  += pData[nCount];      /* sum 成了工作变量，不好 */
    }
}
```

若改为如下，则更好些。

```
void SumData( unsigned int num, int * pData, int *sum )
{
    unsigned int nCount;
    int sum_temp;

    sum_temp = 0;
    for (nCount = 0; nCount < num; nCount ++)
    {
        sum_temp  += pData[nCount];
    }

    *sum = sum_temp;
}
```

⑥ 减少函数本身或函数间的递归调用。递归调用特别是函数间的递归调用（如 A->B->C->B->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。

⑦ 如果函数有返回值，调用者必须根据返回值做仔细、全面地处理，而不能对返回值不闻不问。

⑧ **在面向嵌入式设备的引擎开发中禁止使用任何标准库或平台库函数**，大部分函数我们都已实现在 Common 库中，可以安全使用。

⑨ 特殊情况时，在项目负责人的许可下，可以使用平台库函数，针对特定平台进一步提升产品效率。

5.6 模板类

① 在使用模板类申明一个对象时，必须明确写出模板参数。

② 在使用模板类对象时，访问父类成员时，必须使用“this”指针明确指出（对于非模板类，也建议如此使用）。

```
template <class T>
class tpl_test
{
    private:
        T       item;
    public:
        void        set_elem(T x);
};

template <class T>
void tpl_test<T>::set_elem(T x)
{
    this->item = x;
    return;
}
```

```
template <class Tp>
class tpl_derive : public tpl_test<Tp>
{
    public:
        void tpl_set(Tp x);
};

template <class Tp>
void tpl_derive<Tp>::tpl_set(Tp x)
{
    std::cout<<"tpl_set"<<std::endl;
    this->set_elem(x);  /* 必须添加 this->，否则 g++编译报错 */
    return;
}
```

5.7 流程控制语句

① 如果是一个常量和一个变量进行比较的条件判断，常量在前，变量在后。示例如下：

```
if (NULL == pThis) …
```
反过来变量在前常量在后，上式如果漏写一个“=”就变成了赋值语句 pThis = NULL，很危险。

② if 尽量加上 else 分支，对没有 else 分支的语句要小心对待。当出现多个条件分支，如：if(A)，else if(B)，else if(C)，else(D)，必须计算 A、B、C、D 的出现概率，按概率从大到小排序，减少平均判断次数。

③ 一般情况下要尽量避免循环体内含判断语句，与循环变量无关的判断语句可以移到循环体外，这样能有效减少判断次数。示例如下：

如下代码效率较低：

```
for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
{
    if ( RECT_AREA == data_type )
    {
        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 ( RECT_AREA == data_type )
{
    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;
    }
}
```

④ 在多重循环中，应将最忙的循环放在最内层，提高 CPU Cache 的命中率。示例如下：

```
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];
    }
}
```

⑤ 自加自减运算

⑥ 尽量不要使用 goto 语句。

⑦ 不论是否执行到，switch 语句必须包含 default 分支。

5.8 标准库函数使用

① 面向嵌入式终端的引擎，禁止使用任何库函数。

② 在允许使用库函数的引擎中，库函数使用必须遵循 ISO C++规范，必须确保在 VC 及 gcc、g++下所表述意义完全相同。

③ 关于 strcmp 的应用举例

```
int nResult
nResult = strcmp(m_ptHMMSet->ppHMMIndex[m]->szHMMName,szName);
// strcmp返回值为int，若szHMMName大于szName返回值大于0，相同返回为0，小于则返回值小于0
// 以下代码虽然在VC下执行正确，但在gcc下执行不正确，且微软提供的msdn也并非如此描述
switch(nResult)
{
case 0:
    bFind = true;                    //Modified by Gump Duo
    break;
case 1:
    j = m - 1;
    break;
case -1:
    i = m + 1;
    break;
default:
    return NULL;
}
// 以下代码正确
if (nResult > 0) {
    j = m - 1;
}
else if(nResult < 0) {
    i = m + 1;
}
else {
    bFind = true;
}
```

5.9 程序效率

程序效率分为全局效率、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。

编程时，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，必须要时时刻刻注意代码的效率。规范如下：

① 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。
② 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。示例如下：

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_STRU
{
    BYTE szName[8];
    BYTE byAge;
    BYTE bySex;
    BYTE byClass;
    BYTE bySubject;
    float fScore;
} 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 szName[8];
    BYTE byAge;
    BYTE bySex;
    BYTE byClass;
} STUDENT;

typedef struct STUDENT_SCORE_STRU
{
    WORD wStudentIndex;
    BYTE bySubject;
    float fScore;
} STUDENT_SCORE;
```

③ 优化代码时，必须要考虑周全，且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。

④ 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。

⑤ 尽量用乘法或其它方法代替除法，使用位操作代替特定的一些整型乘法、取模。示例如下：

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

fRadius = fCircleLength / (2 * PAI);              /* 采用了浮点除法，效率较低 */
fRadius = fCircleLength * PAI_RECIPROCAL * 0.5;   /* 浮点乘法比浮点除法快很多 */
```

“n%2”相当于“n&1”；
“n%8”相当于“n&7”；
“n*2”相当于“n<<1”；
“n/2”相当于“n>>1”；
“n/8”，相当于“n>>3”；
……

⑥ 软件系统的效率与算法、处理任务方式、系统功能及函数结构有很大关系，仅在代码上下功夫一般不能解决根本问题。一方面，必须仔细分析有关算法，并进行优化。另一方面要仔细考查、分析系统及模块处理输入（如事务、消息等）的方式，并加以改进。
⑦ 减少类之间的编译依赖。
⑧ 最后，不要一味追求紧凑的代码，紧凑的代码并不一定代表高效的机器码。

5.10 质量保证

① 精心构造算法，并对其性能、效率进行测试，对较关键的算法最好使用其它算法来对比确认。
② 只引用属于自己的存贮空间。模块封装的好，一般不会发生非法引用他人的空间。
③ 防止引用已经释放的内存空间。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（如指针），而另一模块在随后的某个时刻又使用了它，要严格防止这种情况发生。
④ 过程/函数中分配的内存，在过程/函数退出之前必须要释放。过程/函数中申请的（为打开文件而使用的）文件句柄，在过程/函数退出之前必须要关闭。否则很可能会引起很严重后果，且问题难以定位。
⑤ 防止内存操作（主要是指对数组、指针、内存地址等的操作）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，当我们进行这些操作时一定要仔细小心。
⑥ 严禁随意更改模块或系统的有关设置和配置。即编程时，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，也不能随意改变自己模块与其它模块的接口。
⑦ 编程时，要防止差 1 错误。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。当编完一段程序后，应对这些操作符进行彻底检查。
⑧ 要时刻注意易混淆的操作符。当编完一段程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。如 “=”与“==”、“|”与“||”、“&”与“&&”等，若拼写错了，编译器不一定能够检查出来。
⑨ 时刻注意表达式是否会上溢、下溢。示例如下：

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 */
}
```

5.11 Starteam 及 SVN

① 编写代码要在 Starteam/SVN 下进行，注意随时保存，防止由于断电、硬盘损坏等原因造成代码丢失，代码完成编写或修改后要及时上传。

② 修改代码之前，先锁住代码，如代码已被他人锁住，需等他人释放锁以后再修改。
③ 提交代码时，需注明本次修改的范围。

5.12 代码编辑、编译与审查

① 软件系统目录由质管部设计，项目负责人可根据具体项目需要进行合理定制，方便开发人员使用。
② 要小心地使用编辑器提供的块拷贝功能编程。当某段代码与另一段代码的处理功能相似时，许多开发人员都用块拷贝功能来完成这段代码的编写。由于程序功能相近，故所使用的变量、采用的表达式等在功能及命名上可能都很相近，所以使用块拷贝时要注意，除了修改相应的程序外，一定要把使用的每个变量仔细查看一遍，以改成正确的。不应指望编译器能查出所有这种隐藏很深的错误。
③ 具体项目组中，要统一编译选项。所有项目组成员对程序进行编译时，必须打开编译器的所有告警开关。
④ 项目组定期通过代码走读及审查方式对代码进行检查。代码走读主要是对程序的编程风格如注释、命名等以及编程时易出错的内容进行检查，可由开发人员自己或开发人员交叉的方式进行；代码审查主要是对程序实现的功能及程序的稳定性、安全性、可靠性等进行检查及评审，可通过自审、交叉审核或指定部门抽查等方式进行。
⑤ 测试部测试产品之前，应对代码进行抽查及评审。