

科达通信 C++编码规范

（仅供内部使用）

文 件 编 号：	
版 本 号：	V 1.2
实 施 日 期：	2008-11-20
保 密 等 级：	<input checked="" type="checkbox"/> 秘密 <input type="checkbox"/> 机密 <input type="checkbox"/> 绝密
编 制：	和玮萍
审 核：	
会 签：	
批 准：	

修订记录

日期	版本号	描述	作者
2007-03-30	0.1	初稿完成	和玮苹
2007-04-05	0.2	根据评审意见修改，拆分为 C 和 C++两个部分	和玮苹
2007-04-05	0.3	添加示例	和玮苹
2007-04-15	V1.0	最终发布	和玮苹
2008-08-04	V1.1	修改示例	和玮苹
2008-11-04	V1.2	1. 重新整理所有示例，将源码附进来。 2. 转为 PDF 格式。 3. 更新适用范围的描述。	李洪强

目 录

1	目的.....	3
2	适用范围.....	3
3	定义.....	3
4	设计规范.....	3
4.1	函数设计规范.....	3
4.2	结构设计规范.....	6
5	C++语言编码规范.....	6
5.1	排版.....	6
5.2	注释.....	15
5.3	标识符命名.....	25
5.4	可读性.....	30
5.5	变量、结构.....	33
5.6	宏.....	35
5.7	函数.....	37
5.7.1	总则.....	37
5.7.2	接口、参数.....	43
5.7.3	条件、循环、分支语句.....	45
5.7.4	内存、指针.....	49
5.7.5	返回值.....	50
5.8	可维护性.....	52
5.9	可测试性.....	55
5.10	程序效率.....	58
5.11	代码编辑、编译、审查.....	59
6	规范示例代码.....	60

C++编码规范

1 目的

本文档详细描述了软件开发过程中的函数和结构设计规范、C++编码规范，代码编译、编辑、审查规范以及程序设计时需要考虑的可测试性、程序效率等要素，要求开发人员严格遵照该规范实施。

该规范未涉及到的内容，由各项目组自行定义，并上报到流程组，通过公司评审后，由流程组统一纳入到该规范中。

2 适用范围

科达研发中心所有的开发人员在编码时使用本规则。

规范中包括“规则”和“建议”两类。“建议”的强制性略低于“规则”。各规则条款关注的重点不同，在一些具体的实践中可能会出现矛盾的情况。大家在应用时要理解规则制定的初衷和原因，不要断章取义地去卡条条框框。

3 定义

扇入：是指一个函数被直接调用（控制）的上级函数的数目；

扇出：是指一个函数直接调用（控制）其它函数的数目；

断言：断言是在调试版本中使用的一种条件判断，它表示程序执行到某一点时必须满足的条件，若条件不满足则引起程序中断，而在正式版本中断言以空语句代替。

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），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。

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中。

可重入性：是指函数可以被多个任务进程调用。

4 设计规范

4.1 函数设计规范

【规则1】 为简单功能编写函数，一个函数仅完成一件功能，不要设计多用途面面俱到的函数。

说明：虽然为仅用一两行就可完成的功能去编函数好像没有必要，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；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。

【规则2】 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；对于特殊

情况，可以不遵循以上原则，但必须给出注释说明（需要确定这是最好的方式）。

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，则返回为错针。

【示例】：

```
u32 IntegerSum(u32 dwAddNum)
{
    // 注意是 static 类型的，若为 auto 类型，则函数变为可预测的
    static u32 dwSum = 0;
    for(u32 dwIndex = 0; dwIndex < dwAddNum; dwIndex++)
    {
        dwSum += dwIndex;
    }
    return dwSum;
}
```

【建议1】 ▲防止把没有关联的语句放到一个函数中，防止函数内出现随机内聚。

说明：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。

随机内聚给函数的维护、测试及以后的升级等造成了不便，同时也使函数的功能不明确；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。

【不好的示例】：

```
// 矩形的长宽与点的坐标基本上没有任何联系，可以认为是一种随机内聚
void CSample::InitVar(void)
{
    // 初始化矩形的长和宽
    m_rcFrame.dwLength = 0;
    m_rcFrame.dwWidth = 0;

    // 初始化点的坐标
    m_cStartPoint.x = 0;
    m_cStartPoint.y = 0;
}
```

【好的示例】:

```
void CSample::InitRect(void)
{
    // 初始化矩形的长和宽
    m_rcFrame.dwLength = 0;
    m_rcFrame.dwWidth = 0;
}

void CSample::InitPoint(void)
{
    // 初始化点的坐标
    m_cStartPoint.x = 0;
    m_cStartPoint.y = 0;
}
```

【建议2】 ▲▲ 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。

【建议3】 ▲▲ 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：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。

【建议4】 ▲▲ 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。

4.2 结构设计规范

【规则1】 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；不同结构间的关系不要过于复杂。

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；面面俱到、灵活的数据结构反而容易引起误解和操作困难；若两个结构间关系较复杂、密切，那么应合为一个结构。

【规则2】 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。

【规则3】 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全局变量，防止多个不同模块或函数都可以修改、创建同一全局变量的现象。

说明：降低公共变量耦合度。

【建议1】 ▲▲ 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。

5 C++语言编码规范

5.1 排版

【规则1-1】 函数的开始、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，switch-case语句下的 case 处理语句也要遵从语句缩进要求，缩进的空格数为 4 个。

说明：对于由开发工具自动生成的代码可以有不一致。

【不好的示例】：

```
void SampleFor(void)
{
... // 程序代码
for(s32 nIndex = 0; nIndex < nNum; nIndex++)
{
... // 程序代码
if(ptDevInfo->bIsValid)
{
```

```
        ... // 程序代码
    }
    ... // 程序代码
}
return;
}

void SampleCase(void)
{
    ... // 程序代码
    switch(m_dwStatus)
    {
        case MCU_IDLE_STATUS:
        {
            ... // 程序代码
            m_dwStatus = MCU_WORKING_STATUS;
        }
        break;
        case MCU_WORKING_STATUS:
        {
            ... // 程序代码
        }
        break;
        ... // 程序代码
    }
}
```

【好的示例】:

```
void SampleFor(void)
{
    ... // 程序代码
    for (s32 nIndex = 0; nIndex < nNum; nIndex++)
    {
        ... // 程序代码
        if (ptDevInfo->bIsValid)
        {
            ... // 程序代码
        }
        ... // 程序代码
    }
    return;
}
```



```
void SampleCase(void)
{
    ... // 程序代码
    switch (m_dwStatus)
    {
        case MCU_IDLE_STATUS:
        {
            ... // 程序代码
            m_dwStatus = MCU_WORKING_STATUS;
        }
        break;
        case MCU_WORKING_STATUS:
        {
            ... // 程序代码
        }
        break;
        ... // 程序代码
        default:
        {
            ... // 程序代码
        }
    }
}
```

【规则1-2】 对齐只能使用空格键，不使用 TAB 键。

说明：其目的是为了使用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐；编辑时可以把一个 TAB 键设成 4 个空格，然后把 TAB 键用空格键代替；建议采用能自动对齐格式的编辑器（如 Microsoft Visual C++）。

【规则1-3】 在每个类声明之后、每个函数定义结束之后都要加空行；在一个函数体内，逻辑上密切相关的语句之间不加空行，其他地方应加空行。

说明：增加代码的可读性。

【不好的示例】：

```
BOOL32 CVtduAppData::Init()
{
    BOOL32 bRet = FALSE;
    m_bConnected = FALSE;
    m_bRegistered = FALSE;
    //初始化穿 NAT 组件
    if (0 != TarbInit(0, 0))
    {
```

```

        ...    // 代码段 1
    }
    //初始化交换通道
    if (0 != TarbRegisterCallback(TarbCallBack, this))
    {
        ...    // 代码段 2
    }
}

```

【好的示例】:

```

BOOL32 CVtduAppData::Init()
{
    BOOL32 bRet = FALSE;

    m_bConnected = FALSE;
    m_bRegistered = FALSE;

    //初始化穿 NAT 组件
    if (0 != TarbInit(0, 0))
    {
        ...    // 代码段 1
    }

    //初始化交换通道
    if (0 != TarbRegisterCallback(TarbCallBack, this))
    {
        ...    // 代码段 2
    }
}

```

【规则1-4】 一行只能写一条语句。

说明：既增加代码可读性，又便于代码调试（一次调试一个语句）。

【不好的示例】:

```

void Sample(s32 nStartX, s32 nStartY)
{
    s32 nDrawX = nStartX; s32 nDrawY = nStartY;
    ... // 程序代码
}

```

【好的示例】:

```
void Sample(s32 nStartX, s32 nStartY)
{
    s32 nDrawX = nStartX;
    s32 nDrawY = nStartY;
    ... // 程序代码
}
```

【规则1-5】 一程序以小于等于 80 字符为宜，不要写的过长；较长的语句（大于 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。

说明：无。

【不好的示例】：

```
BOOL32 CEqpRPCtrl::ValidPlayGroupId (u32 dwGroupId, u32 *pdwIndex)
{
    ... // 程序代码

    // 下面程序段的注释
    for (u16 i = 0; i < MAX_SYNCMULTI_GROUP; i++)
    {
        // 句子太长没有分行
        if ((m_atPlayGroup[i].m_dwPlayGroupId == dwGroupId) &&
            (m_atPlayGroup[i].m_tRpPlayGroup.m_dwPlayerNum != 0))
        {
            ... //程序段 1
        }
        ... //程序段 2
    }
    ... //程序段 3
}
```

```
BOOL32 CEqpRPCtrl::ValidPlayGroupId (u32 dwGroupId, u32 *pdwIndex)
{
    ... // 程序代码

    // 下面程序段的注释
    for (u16 i = 0; i < MAX_SYNCMULTI_GROUP; i++)
    {
        // 句子有分行，但分行不合理
        if ((m_atPlayGroup[i].m_dwPlayGroupId ==
            dwGroupId) && (m_atPlayGroup[i].m_tRpPlayGroup.m_dwPlayerNum != 0))
        {
```

```
        ...    //程序段 1
    }
    ...    //程序段 2
}
...    //程序段 3
}
```

【好的示例】:

```
BOOL32 CEqpRPCtrl::ValidPlayGroupId (u32 dwGroupId, u32 *pdwIndex)
{
    ... // 程序代码

    //下面程序段的注释
    for (u16 i = 0; i < MAX_SYNCMULTI_GROUP; i++)
    {
        if ((m_atPlayGroup[i].m_dwPlayGroupID == dwGroupId)
            && (m_atPlayGroup[i].m_tRpPlayGroup.m_dwPlayerNum != 0))
        {
            ...    //程序段 1
        }
        ...    //程序段 2
    }
    ...    //程序段 3
}
```

【规则1-6】 if、else、for、do、while、case、switch、default 等语句自占一行，且 if、else、for、do、while 语句的执行语句部分无论多少都要加括号 {}。

说明：无

【不好的示例】:

```
if ( pNmcMainCtrl == NULL ) return;
if ( pNmcMainCtrl == NULL )
    return;
```

【好的示例】:

```
if( pNmcMainCtrl == NULL )
{
    return;
}
```

【规则1-7】 程序块的分界符（如 C++语言的“{”和“}”）应各独占一行且位于同一列，并与引用他们的语句左对齐；在函数的体开始、类的定义、结构的定义、枚举的定义以及 if、for、do、while、

switch 和 case 语句中的程序都要采用此方式。

说明：无。

『不好的示例』：

```
for(...) {
    ... // 程序代码
}
if(...)
{
    ... // 程序代码
}
void SampleFun(void)
{
    ... // 程序代码
}
```

『好的示例』：

```
for (...)
{
    ... // 程序代码
}
if (...)
{
    ... // 程序代码
}
void SampleFun(void)
{
    ... // 程序代码
}
```

【规则1-8】 逗号、分号只在后面加空格；比较操作符，赋值操作符“=”、“+=”，算数操作符“+”、“%”，逻辑操作符“&&”、“&”，位域操作符“<<”，“^”等双目操作符的前后都应加空格；“!”、“~”、“++”、“--”、“&”（地址运算符）、“*”等单目操作符前后不加空格；“->”、“.”前后都不加空格；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。

说明：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，如最内层的括号内侧（即左括号后面和右括号前面）不要加空格，因为在 C++ 语言中括号已经是最清晰的标志了；另外，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；最后，即使留空格，也不要连续留两个以上空格（为了保证缩进和排比留空除外）。

『不好的示例』：

```
void CMainCtrl::DealAsyncMsg()
{
    BOOL bFind ,bRet;    // 逗号分号只在其后面加空格

    bFind=FALSE;          // 赋值操作符等双目操作符前后都应该加空格
    bRet=pDisp -> DispEvent (cTnmMsg);      // "->"指针前后不加空格

    ...    // 代码段 1

    // 检索消息分发表
    CDispatchMessage * pDisp = NULL;    // 内容操作"*"与内容之间
    ...    // 代码段 2

    // "+"等单目操作符前后不加空格，for、if 等与后面的括号间应加空格
    for( s32 nIndex = 0; nIndex < emHandlerNum; ++ nIndex)
    {
        ...    // 代码段 3
    }

    ...    // 代码段 4
}
```

【好的示例】:

```
void CMainCtrl::DealAsyncMsg()
{
    BOOL bFind, bRet;    // 逗号分号只在其后面加空格

    bFind = FALSE;          // 赋值操作符等双目操作符前后都应该加空格
    bRet = pDisp->DispEvent(cTnmMsg);    // "->"指针前后不加空格

    ...    // 代码段 1

    // 检索消息分发表
    CDispatchMessage *pDisp = NULL;    // 内容操作"*"与内容之间
    ...    // 代码段 2

    // "+"等单目操作符前后不加空格，for、if 等与后面的括号间应加空格
    for ( s32 nIndex = 0; nIndex < emHandlerNum; ++nIndex)
    {
        ...    // 代码段 3
    }

    ...    // 代码段 4
}
```

```
}
```

【规则1-9】 相关的赋值语句的等号对齐；禁止语句连等。

说明：无。

【不好的示例】：

```
BOOL32 CVtduAppData::AddSwitchDst(u16 wDstIndex, const CPduSwitchChannel & cPdu,
u16 wSrcIndex)
{
    assert(0 != wSrcIndex);
    assert(FALSE == m_pcSwitchDst[wDstIndex].m_bValid);

    //保存到表中
    m_pcSwitchDst[wDstIndex].m_bValid = TRUE;
    m_pcSwitchDst[wDstIndex].m_cDstChn = cPdu.m_cDstChn;
    m_pcSwitchDst[wDstIndex].m_cDstGuid = cPdu.m_cDstGuid;
    m_pcSwitchDst[wDstIndex].m_dwDstIP = cPdu.m_dwDstIP;
    m_pcSwitchDst[wDstIndex].m_byDstNetType = cPdu.m_byDstNetType;
    m_pcSwitchDst[wDstIndex].m_wDstPort = cPdu.m_wDstPort;
    m_pcSwitchDst[wDstIndex].m_byMediaType = cPdu.m_byMediaType;
    m_pcSwitchDst[wDstIndex].m_byHasNetBuf = cPdu.m_byDstNetType;
    m_pcSwitchDst[wDstIndex].m_byHasRtcp = cPdu.m_byMediaType;
    m_pcSwitchDst[wDstIndex].m_wSwitchSrc = wSrcIndex;

    ...    //程序段
}
```

【好的示例】：

```
BOOL32 CVtduAppData::AddSwitchDst(u16 wDstIndex, const CPduSwitchChannel &cPdu,
u16 wSrcIndex)
{
    assert(0 != wSrcIndex);
    assert(!m_pcSwitchDst[wDstIndex].m_bValid);

    //保存到表中
    m_pcSwitchDst[wDstIndex].m_bValid    = TRUE;
    m_pcSwitchDst[wDstIndex].m_cDstChn    = cPdu.m_cDstChn;
    m_pcSwitchDst[wDstIndex].m_dwDstIP    = cPdu.m_dwDstIP;
    m_pcSwitchDst[wDstIndex].m_cDstGuid   = cPdu.m_cDstGuid;
    m_pcSwitchDst[wDstIndex].m_wDstPort   = cPdu.m_wDstPort;
    m_pcSwitchDst[wDstIndex].m_byHasRtcp   = cPdu.m_byMediaType;
    m_pcSwitchDst[wDstIndex].m_wSwitchSrc  = wSrcIndex;
```

```
m_pcSwitchDst[wDstIndex].m_byMediaType = cPdu.m_byMediaType;
m_pcSwitchDst[wDstIndex].m_byHasNetBuf = cPdu.m_byDstNetType;
m_pcSwitchDst[wDstIndex].m_byDstNetType = cPdu.m_byDstNetType;

...    //程序段
}
```

【规则1-10】 C++的“.cpp”文件中，函数的组织顺序应按照：先主函数再子函数，先进程主体继之进程分支再各过程，由大到小的顺序排版；类的实现部分应按照先构造/析构函数，再公用函数，最后私有函数的顺序排版。

说明：无。

【规则1-11】 C++的“.h”文件中，接口的定义应按照先函数，后变量；函数要求先构造/析构函数，再公用函数、保护函数、私有函数的顺序；变量要求先公有，再保护，后私有的顺序。

说明：无。

【规则1-12】 头文件的组织顺序应该按照以下顺序：

```
#ifndef    文件名_H

#define    文件名_H

#include    文件引用

宏定义

类型定义

常量定义

类定义

函数原型定义

#endif          /*  <要求必须在此注释出此“#endif”对应于哪个“#ifdef”>  */
```

说明：无。

【规则1-13】 一个头文件内只允许存放一个类相关的部分。

说明：无。

5.2 注释

【规则2-1】 源程序有效注释量必须在 20%以上，注释的内容要清楚、明了，含义准确，防止注释二义性；避免在注释中使用缩写，特别是非常用的缩写。

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，错误的注释不但无益反而有害。

【规则2-2】 每行注释不能超过 80 列；注释要与所描述内容进行同样的缩进。

说明：可使程序排版整齐，并方便注释的阅读与理解。

【不好的示例】：

```
void CVtduData::StopSwitch(u16 wChannelId)
{
    ... // 代码段
// 不处理仅有音频的
    if (AUDIO_ONLY == m_pcSwitchInfo[wChannelId].dwMediatype)
    {
        ... // 代码段
    }

    // 没有建立交换
    if (SWITCH_ON == m_pcSwitchInfo[wChannelId].dwSwitchState)
    {
        ... // 代码段
    }

    ... // 代码段
}
```

【好的示例】：

```
void CVtduData::StopSwitch(u16 wChannelId)
{
    ... // 代码段
// 不处理仅有音频的
    if (AUDIO_ONLY == m_pcSwitchInfo[wChannelId].dwMediatype)
    {
        ... // 代码段
    }

    // 没有建立交换
    if (SWITCH_ON == m_pcSwitchInfo[wChannelId].dwSwitchState)
    {
        ... // 代码段
    }

    ... // 代码段
}
```

【规则2-3】 注释应与其描述的代码相临，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（对代码块的注释）则需与其上面的代码用空行隔开；应该避免在一行代码或表达式的中间插入注释。

说明：不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

【不好的示例】：

```
// 服务器健康状态
const u8 SERVICE_HEALTH_NORMAL      = 0;    // 工作正常
const u8 SERVICE_HEALTH_THREADDEAD  = 1;    // 进程(CMU)中的某一线程已死
                                           // 或线程长时间不响应服务

//主备数据同步的类型
const u8 SYNCDATA_ADD = 1;
//增加数据
const u8 SYNCDATA_MOD = 2;
//修改数据
const u8 SYNCDATA_DEL = 3;
//删除数据
```

【好的示例】：

```
//服务器健康状态
const u8 SERVICE_HEALTH_NORMAL      = 0;    // 工作正常
const u8 SERVICE_HEALTH_THREADDEAD  = 1;    // 进程(CMU)中的某一线程已死
                                           // 或线程长时间不响应服务

//主备数据同步的类型
const u8 SYNCDATA_ADD = 1;    // 增加数据
const u8 SYNCDATA_MOD = 2;    // 修改数据
const u8 SYNCDATA_DEL = 3;    // 删除数据
```

【规则2-4】 注释可以使用标准C的“/*……*/”，也可以采用C++的“//”；一般的原则是一行注释用“//”，一段注释用“/*……*/”。

说明：无。

【规则2-5】 注释全部使用为中文描述。

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，全部使用中文。

【规则2-6】 注释与代码必须保持一致，修改代码同时修改相应的注释，不再有用的注释要删除。

说明：边写代码边注释，保持注释与代码的一致性。

【规则2-7】 说明性文件（如头文件“.h”文件、“.inc”文件、“.def”文件、编译说明文件“.cfg”等）头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者、内容、功能、与其它文

件的关系、修改日志等，头文件的注释中还应包含函数功能简要说明。

说明：统一格式如下：

```
/*=====
模块名  :
文件名  :
相关文件:
实现功能:
作者    : (必需用中文名)
版权    : <Copyright(C) 2003-2007 Suzhou Keda Technology Co., Ltd. All rights reserved.>
=====

修改记录:
日期      版本      修改人      走读人      修改记录
1998/09/15  V1.0      某某(中文)  某某(中文)  记录关键内容
=====*/
```

版本控制说明：

- 源代码的版本按文件的粒度进行维护；
- 创建一个新文件时，其初始版本为“1.0”，创建过程中的任何修改都不需要增加修改记录；
- 从软件第一次正式发布开始，对其源文件的每次修改都应该在文件头中加入相应的修改记录，并将文件的子版本加 1；
- 升级软件的主版本时，其源文件的相应主版本号随之增加。与创建新文件时一样，在该主版本第一次发布之前，对文件的任何修改都不需要再增加修改记录。

【不好的示例】：

```
/******
模块名  : PU 主控模块
文件名  : mainctrl.h
作者    : 张三
*****/

#ifndef MAINCTRL_H
#define MAINCTRL_H
```

【好的示例】：

```
/*=====
模块名  : PU 主控模块
文件名  : mainctrl.h
相关文件: magprocess.h, mediactrl.h, kdm2401es.h, kdm2404s.h, kdm2501.h, kdmwin.h
```

recplyctrl.h, pinalarmctrl.h

实现功能：协议栈与主控模块的交互

作者：张三

版权：<Copyright(c) 2003-2007 Suzhou Keda Technology Co.,Ltd. All right reserved.>

修改记录：

日期	版本	修改人	走读人	修改记录
2008/8/14	V1.0	李 四	王 五	记录修改的关键内容
2008/9/20	V1.1	李 四	王 五	增加了 XX 接口

=====*/

#ifndef MAINCTRL_H

#define MAINCTRL_H

【规则2-8】 源文件头部应进行注释，列出：模块名、文件名、相关文件、实现功能、作者、版权修改记录等。

说明：统一格式如下：

```
/*=====
模块名：
文件名：
相关文件：
实现功能：
作者：（必需用中文名）
版权：<Copyright(c) 2003-2007 Suzhou Keda Technology Co.,Ltd. All right reserved.>
=====
日期 版本 修改人 走读人 修改记录
1998/09/15 V1.0 某某（中文） 某某（中文） 记录关键内容
=====*/
```

示例参见规则 2-7：

【规则2-9】 函数头部应进行注释，列出：函数的名称、目的/功能、输入参数、输出参数、返回值、算法实现等。

说明：统一格式如下：

```
/*=====
函数名：
功能：
算法实现：<可选项>
参数说明：[I/O/IO]参数：含义（说明：按照参数的顺序排序）
返回值说明：
=====
```

修改记录：

日期	版本	修改人	走读人	修改记录
1998/09/15	V1.0	某某（中文）	某某（中文）	记录关键内容

=====*/

对于实现比较复杂的函数或主要功能函数，程序员应该加上“算法实现”部分；建议每个函数都在此处详细注释说明其输入输出参数的含义和有效范围。

【不好的示例】:

```

/*=====
函数名      : SlowGroupPlay
功能        : 组同步慢放
=====*/
u32 CEqpRPCtrl::SlowGroupPlay( u32 dwGroupId, u8 *pbyPreStatus )
{
    ...    //程序
}
  
```

【好的示例】:

```

/*=====
函数名      : SlowGroupPlay
功能        : 组同步慢放
算法实现    :
参数说明    : u32 dwGroupId          [in]准备慢放放像的组编号
               u8 *pbyPreStatus      [out]返回前一个播放状态
返回值说明: RET_SUCESS              成功
               RET_FAIL              失败
=====
  
```

修改记录：

日期	版本	修改人	走读人	修改记录
2008/05/28	V1.0	张三（中文）	李四（中文）	记录关键内容

=====*/

```

u32 CEqpRPCtrl::SlowGroupPlay( u32 dwGroupId, u8 *pbyPreStatus )
{
    ...    //程序
}
  
```

【规则2-10】与头文件一样，每个类应该有一个注释头用来说明该类，列出：类的目的/功能、主要接口等信息。

说明：统一格式如下：

```

/*=====
  
```

类名 :
功能 : <简要说明该类所完成的功能>
主要接口: <简要说明该类的主要接口>
备注 : <使用该类时需要注意的问题（如果有的话）>

修改记录:
日 期 版本 修改人 走读人 修改记录
1998/09/15 V1.0 某某（中文） 某某（中文） 记录关键内容
=====*/

【不好的示例】:

```
/*=====
类名      : CCmuSsnIns
功能      : 继承 OSP 模板类 CInstance 的类，用于维护和异域 CMUI 的连接
=====*/

class CCmuSsnIns : public CInstance
{
public:
    CCmuSsnIns();
    virtual ~CCmuSsnIns();
private:
    enum{IDLE, WAITING_CON, WAITING_REG, SERVICE};

    LPCSTR GetStrState();
    void InstanceEntry(CMessage *const pcMsg);

    ...    //其它接口的声明
private:
    ...    //成员变量的定义
};
```

【好的示例】:

```
/*=====
类名      : CCmuSsnIns
功能      : 继承 OSP 模板类 CInstance 的类，用于维护和异域 CMUI 的连接
主要接口: void InstanceEntry: 实例消息处理入口函数，必须 override，参数为传入的消息，无返回值
...      //其它更多类似的接口说明，在本例子中由于篇幅限制，省略
备注      :
```

修改记录:
日 期 版本 修改人 走读人 修改记录

2008/05/28 V1.0 张三（中文） 李四（中文） 记录关键内容

```
=====*/
class CCmuSsnIns : public CInstance
{
public:
    CCmuSsnIns();
    virtual ~CCmuSsnIns();
private:
    enum{IDLE, WAITING_CON, WAITING_REG, SERVICE};

    LPCSTR GetStrState();
    void InstanceEntry(CMessage *const pcMsg);

    ...    //其它接口的声明
private:
    ...    //成员变量的定义
};
```

【规则2-11】 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，使代码成为自注释的；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；建议对函数、变量、结构的命名全部写成自注释的。

说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

『不好的示例』：

```
void CCmuCcSsnIns::Broadcast (u16 wEvent, u8 * const pbyMsg, u16 wLen)
{
    ...    //程序段 1

    //非自注释的，且没有注释说明
    u32 k = g_cCmuCcData.GetOnlineCmuNum();
    ...    // 程序段 2
    u32 i = 0;
    for(i = 0; i < k; i++)
    {
        ...    // 程序段 3
    }
}
```

『好的示例』：

```
void CCmuCcSsnIns::BroadcastToAllCmu (u16 wEvent, u8 * const pbyMsg, u16 wLen)
{
    ...    //程序段 1
```

```

u32 dwCmuNum = g_cCmuCcData.GetOnlineCmuNum();
...    //程序段 2
u32 dwIndex = 0;
for (dwIndex = 0; dwIndex < dwCmuNum; dwIndex++)
{
    ...    //程序段 3
}
}

```

【规则2-12】 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；建议数据结构声明的命名全部写成自注释的。

说明：无。

【不好的示例】：

```

enum EM_CU_ERROR_TYPE
{
    ERROR_CMU_UNREACHABLE,
    ERROR_USER_NOT_EXIT,
    ERROR_PWD_FAIL,
    ERROR_USER_HAS_NO_RIGHT,
    ...
};

```

【好的示例】：

```

enum EM_CU_ERROR_TYPE
{
    ERROR_CMU_UNREACHABLE    = 1,    // CMU 连接不上
    ERROR_USER_NOT_EXIT,        // 登录用户不存在
    ERROR_PWD_FAIL,            // 登录密码不正确
    ERROR_USER_HAS_NO_RIGHT,    // 登录用户无权操作
    ...
};

```

【规则2-13】 全局变量要有较详细的注释，包括对其功能、取值范围、使用注意事项等的说明。

说明：无

【不好的示例】：

```

BOOL g_bWaitForPuAck = FALSE;

```


【好的示例】:

```
// g_bWaitForPuAck 用于标识 PU 是否在在等待前端的确认消息。
// 取值: TRUE: 当前正在待回应 FALSE: 前端没有等待 PU 的确认消息
// 注意事项: g_bWaitForPuAck 在每次发消息后被置为 TRUE, 收到回应后
//           被置回来。同时为防止意外修改, 只允许通过 SetWaitStatus()
//           修改其他状态, 不允许直接修改。
BOOL g_bWaitForPuAck = FALSE;
```

【规则2-14】 在 switch-case 语句中, case 语句最后必须用 break 显式地标明结束, 若要继续执行下一条语句, 一定要在该 case 语句处理完、下一个 case 语句前注释说明不用 break 结束的理由。

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。

【不好的示例】:

```
switch (pcMsg->event)
{
case CMU_DEV_FORCE_DISCONNECT:    ////设备强制退网
{
    ...    // 代码段 1
}
break;
case CMU_CUI_REGISTER_ACK:        //Cmu 注册请求 Ack
{
    ...    // 代码段 2
}
case CMU_CUI_REGISTER_NACK:       //Cmu 注册请求 Nack
{
    ...    // 代码段 3
}
break;

...    // 其它 case 语句
}
```

【好的示例】:

```
switch (pcMsg->event)
{
case CMU_DEV_FORCE_DISCONNECT:    // 设备强制退网
{
    ...    // 代码段 1
}
break;
```

```
case CMU_CUI_REGISTER_ACK:           // Cmu 注册请求 Ack
{
    ...    // 代码段 2
}
// 在此不使用 break, 原因是...
case CMU_CUI_REGISTER_NACK:         // Cmu 注册请求 Nack
{
    ...    // 代码段 3
}
break;

...    // 其它 case 语句
default:
    break;
}
```

5.3 标识符命名

【规则3-1】 变量、函数、宏命名只能由 26 个字母、10 个数字、下划线组成，不能使用“\$”等特殊符号。

说明：无。

【规则3-2】 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相应的缩写，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，命名中若使用特殊约定或缩写，则要有注释说明。

【不好的示例】：

```
s32 m_nMCN;           // Maximum Channel Number 的缩写
BOOL m_bDefNSet;      // Use Default NAT Setting 的缩写
s32 m_nCFPLen;        // Configure File Path Length 的缩写
```

【好的示例】：

```
s32 m_nMaxChanNum;    // Maximum Channel Number 的缩写
BOOL m_bUseDefNATSet; // Use Default NAT Setting 的缩写
s32 m_nCfgFilePathLen; // Configure File Path Length 的缩写
```

【规则3-3】 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；一个符号名中间不应该出现连续两个‘_’，因为两个‘_’与一个‘_’之间的区别不明显。

说明：无。

【规则3-4】 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。

说明：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，其后加上大小写混排的方式是允许的。

【规则3-5】 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：无。

【规则3-6】 文件后缀的命名规则：对头文件采用 “.h” 作为后缀，C++的源文件以 “.cpp” 作为后缀。

说明：无。

【规则3-7】 函数的命名规则如下：

- (1) 函数名一般采用大小写字母结合的形式；
- (2) 函数名的首字母一般情况下建议大写；
- (3) 函数名中不同意义字段之间不要用下划线链接，而要把每个字段的首字母大写以示区分；
- (4) 函数名的命名采用动宾或者宾动格式，但在一个模块内部只使用一种格式；
- (5) 回调函数以 “CB” 开头；
- (6) 专有名词和缩写按照习惯的大小写形式嵌入函数命中，不受第一条限制；
- (7) 函数名的长度必须控制在 30 字符内，建议控制在 20 个字符之内；
- (8) 对提供给大部分模块用的一些公用函数，如操作系统所提供的函数等，不在以上规定中；
- (9) 具有互斥意义的变量或相反动作的函数，建议用反义词组命名。

说明：无。

【不好的示例】：

```
void Send_msg_allSubCmu (u16 wEvent, u8 * const pbyMsg = NULL, u16 wLen = 0 );
void send_msg__allCmuExceptDescendant (u16 wEvent, u8 * const pbyMsg = NULL, u16
wLen = 0 );
BOOL32 transfer_msg_otherCmu (const CPduDevNo & tCmuNo, u16 wEvent, u8 * const
pbyMsg = NULL );
void CompressFrameInfo(u8 *pbyFrameBuf, void *pbyContext);
```

【好的示例】：

```
void SendMsgToAllSubCmu (u16 wEvent, u8 * const pbyMsg = NULL, u16 wLen = 0 );
void SendMsgToAllCmuExceptDescendant (u16 wEvent, u8 * const pbyMsg = NULL , u16
```

```
wLen = 0 );
BOOL32 TransferMsgToOtherCmu (const CPduDevNo & tCmuNo, u16 wEvent, u8 * const
pbyMsg = NULL );
void CB_CompressFrameInfo(u8 *pbyFrameBuf, void *pbyContext);
```

【规则3-8】 类型的命名规则如下：

- (1) 结构、联合、枚举类型定义统一使用 typedef struct tagStructTypeName {...} TStructTypeName 的形式，即结构名采用大小写结合的方法，在开头加上 T，表明是类型定义；如果该类型为模块间通信使用，则在开头加上 I，表明为通信接口；

【不好的示例】

```
typedef struct tag_PuDevCom_status
{
protected:
    u8  m_byDevType;           // 设备类型
    u8  m_byVidEncChanNum;     // 有效视频编码通道数量
    u8  m_byVidDecChanNum;     // 有效视频解码通道数量
    ...    //其它成员变量的声明

}PuDevCom_status;
```

【好的示例】

```
typedef struct tagPuDevComStatus
{
protected:
    u8  m_byDevType;           // 设备类型
    u8  m_byVidEncChanNum;     // 有效视频编码通道数量
    u8  m_byVidDecChanNum;     // 有效视频解码通道数量
    ...    //其它成员函数的声明

}TPuDevComStatus, *PTPuDevComStatus;
```

- (2) 定义结构、联合、枚举的指针类型应在类型名前再加上 P，即采用 PTAgentErrorMsg, PITAgentErrorMS 的形式。

说明：无。

【规则3-9】 类的命名规则如下：

- (1) 类名采用大小写结合的方法，不同意义字段之间不要用下划线链接，而要把每个字段的首字母大写以示区分；

- (2) 在非接口类开头加上 C，表明是类（class）的定义；
- (3) 类的成员变量统一在前面加 m_前缀，其中 m 为 Member 的缩写；
- (4) 成员函数的定义参照前面有关函数定义的规则；
- (5) 如果该类型为模块间通信使用，在开头加上 I。

说明：无。

【不好的示例】：

```
class Pds_Ssn_Ins
{
public:
    CPdsSsnIns();
    virtual ~CPdsSsnIns();
private:
    u32 DstIIId;           // 目的端实例的 ID
    u32 NodeId;           // 对端 OSP 结点号

private:
    ... // 成员函数的定义

};
```

【好的示例】：

```
class CPdsSsnIns
{
public:
    CPdsSsnIns();
    virtual ~CPdsSsnIns();
private:
    u32 m_dwDstIIId;       // 目的端实例的 ID
    u32 m_dwPeerNodeId;    // 对端 OSP 结点号

private:
    ... // 成员函数的定义

};
```

【规则3-10】 宏定义的命名规则如下：

- (1) 统一使用大写字母和数字，中间可以加下划线；
- (2) 不允许以下划线开头；

- (3) 常量的定义加上强制类型说明；
- (4) 如果宏中含有表达式、结构等，则必须使用括号将其括起来；
- (5) 接口事件的定义在前面加上 EV；

说明：单下划线或双划线在系统定义中使用的比较多，程序中如果大量使用会造成不必要的混淆。

【不好的示例】：

```
#define __MAX_SLOT_NUM      16           // 最大的 slot 数
#define _DEC_IMG_START_TAG  0xFCFE      // 解图像的起始标志
#define calc_area(x, y)     (x * y)      // 计算面积
```

【好的示例】：

```
#define MAX_SLOT_NUM        (u8)16       // 最大的 slot 数
#define DEC_IMG_START_TAG   (u16)0xFCFE  // 解图像的起始标志
#define CALC_AREA(x, y)     ((x) * (y))  // 计算面积
```

【规则3-11】 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。

【不好的示例】：

```
s32 Area(s32 a, b)
{
    s32 c = a * b;
    return c;
}
```

【好的示例】：

```
s32 CalcArea(s32 nWidth, s32 nHeight)
{
    s32 nArea = nWidth * nHeight;
    return nArea;
}
```

【规则3-12】 变量的命名规则如下：

- (1) 对于全局变量，要求在变量前加 g_前缀，g 是 global 的缩写；
- (2) 所有静态变量，包括全局的、局部的或类的静态变量都使用前缀 s_；
- (3) 变量名采用匈牙利命名法（[作用范围前缀（g_/m_）]+[公共前缀]+基本类型前缀+变量
仅供内部使用

名)，变量名最前有类型前缀（公共前缀和基本类型前缀），类型前缀后面是变量的含义，变量含义采用大小写结合的办法；；

- (4) 类的变量名应体现其功能含义和类型，采用类型加英文含义的命名规则，其前缀表示类的缩写（缩写原则上采用去元音方式），随后的每一单词(或单词缩写) 的首字母为大写，如 cstrName, cattrCellId。

说明：基本类型前缀定义如下：

前缀	类型	示例
b	Boolean 布尔	BOOL bIsValidUser;
by	Byte 字节	u8 byChannelType;
ch	Char 字符	s8 chStartCode;
w	不带符号的 Word 字	u16 wMsgLen;
dw	不带符号的 Double word 字符	u32 dwTickCount;
sw	带符号的 16 位整数	s16 swUserNum;
n	带符号的 32 位整数	s32 nCountNum
f	单精度浮点数	float fBoxLen;
sz	以 NULL 终结的字符	s8 szUserName;
t	自定义类型	TPuFullInfo tPuFullInfo;
c	class 类	CCmuCtrlApp cCtrlApp;

公共前缀定义如下：

前缀	类型	示例
a	数组	s8 achBookName[20]
p	指针	u8 *pbyBuf;
t	自定义类型变量(例如：结构)	TPuInfo tPuInfo;
em	枚举类型	EM_PU_STATUS emCurStatus;
aa	多维数组	int aaScoreInfo[100][3];
ap	存放指针的数组	s8 *apUserAddr[256];
at	存放自定义类型变量的数组	TPuInfo atPuInfo[MAX_NUM]
pa	指向数组的指针	s8 *paPuName;
pp	多维指针	s8 **ppbyAddrInfo;
pt	指向自定义类型的指针变量	TPuInfo *ptCurOperPuInfo;
pf	函数指针	int (*pfGetUnitSize)(int);
注：如果使用多维数组和指针时需要报项目经理审批		

5.4 可读性

【规则4-1】 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。

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。

【不好的示例】:

```
wOutValu = wHasHighByte << 8 | wHasLowByte;
if (bIsAdmin | bIsUser && byCameCtrlStatus & byCanCtrlMask)
if (nPreVal | nCurVal < nTestVal & dwValidBitMask))
```

【好的示例】:

```
wOutValu = (wHasHighByte << 8) | wHasLowByte;
if ((bIsAdmin | bIsUser) && (byCameCtrlStatus & byCanCtrlMask))
if ((nPreVal | nCurVal) < (nTestVal & dwValidBitMask))
```

【规则4-2】 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。

说明：无。

【不好的示例】:

```
if ( 0 == m_byCurStatus )
{
    m_byCurStatus = 1;
    ... // 代码段
}
```

【好的示例】:

```
#define REPLAY_STATUS_STOP          (u8)0          // 停止状态
#define REPLAY_STATUS_PLAY          (u8)1          // 正常播放状态
#define REPLAY_STATUS_PAUSE         (u8)2          // 正常播放状态

if ( REPLAY_STATUS_PLAY == m_byCurStatus )
{
    m_byCurStatus = REPLAY_STATUS_PAUSE;
    ... // 代码段
}
```

【规则4-3】 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。

【不好的示例】:

```
rcMainFrame.nLength = 200;
```



```
strFrameTitle = "";  
... // 代码段  
rcMainFrame.nWidth = 100;
```

【好的示例】:

```
rcMainFrame.nLength = 200;  
rcMainFrame.nWidth = 100; // 矩形的长宽关系较紧密，放在一起。  
  
strFrameTitle = "";
```

【规则4-4】 禁止使用难懂的技巧性很高的语句。

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，必要时需要给项目经理审批。

【不好的示例】:

```
*pValue++ += 1;  
*++pValue += 1;
```

【好的示例】:

```
*pValue += 1;  
pValue++; // 此两句相当于*pValue++ += 1;  
  
++pValue;  
*pValue += 1; // 此两句相当于*++pValue += 1;
```

【建议4-1】 ▲ 当一个函数中对较长变量（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（或者变量），或者取指针来代替。

说明：这样可以增加编程效率和程序的可读性。

【示例】:

```
m_patPuVidEncChanCfg[byNum * PUMAX_ENCCHAN_NUM + nIndex].ptImageEncParam->byFrameRate = PU_DEFCFG_VIDFRMRATE;  
m_patPuVidEncChanCfg[byNum * PUMAX_ENCCHAN_NUM + nIndex].ptImageEncParam->byImageQuality = PU_VIDEO_SPEED_PRI;  
m_patPuVidEncChanCfg[byNum * PUMAX_ENCCHAN_NUM + nIndex].ptImageEncParam->byCbrVbr = PU_BITRATE_VARIABLE;  
...
```

可以改为:

```
#define PENCPARAM (m_patPuVidEncChanCfg[byNum * PUMAX_ENCCHAN_NUM + nIndex].ptImageEncParam)  
PENCPARAM->byFrameRate = PU_DEFCFG_VIDFRMRATE;  
PENCPARAM->byImageQuality = PU_VIDEO_SPEED_PRI;  
PENCPARAM->byCbrVbr = PU_BITRATE_VARIABLE;  
...
```

或者：

```
TImageEncParam *ptEncParam = m_patPuVidEncChanCfg[byNum * PUMAX_ENCCHAN_NUM + nIndex].ptImageEncParam;
ptEncParam->byFrameRate = PU_DEFCFG_VIDFRMRATE;
ptEncParam->byImageQuality = PU_VIDEO_SPEED_PRI;
ptEncParam->byCbrVbr = PU_BITRATE_VARIABLE;
...
```

5.5 变量、结构

【规则5-1】 局部变量必须赋初值，声明过的变量必须被使用；严禁使用未经初始化的变量作为右值。

说明：没有被使用的变量应从程序中取消，以提高代码的可读性和可靠性；在 C/C++中引用未经赋值的指针，经常会引起系统崩溃。

【规则5-2】 严禁局部变量与全局变量同名，禁止同一函数中两个局部变量同名。

说明：若使用了较好的命名规则，那么此问题可自动消除。

【规则5-3】 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，例如：尽量少用 int 等与具体机器相关的类型。

说明：使用标准的数据类型，有利于程序的移植；常用数据类型及其含义如下：

类型	类型描述	示例
u8	无符号的 8bit 字节	unsigned char byLetter -> u8 byLetter
u16	无符号的 16bit 整数	unsigned short wValue -> u16 wValue
u32	无符号的 32bit 整数	unsigned int dwValue -> u32 dwValue
u64	无符号的 64bit 整数	unsigned long long ullValue -> u64 u64Value
s8	有符号的 8bit 字节	signed char byLetter -> s8 byLetter
s16	有符号的 16bit 整数	signed short wValue -> s16 wValue
s32	有符号的 32bit 整数	signed int nValue -> s32 nvalue
s64	有符号的 64bit 整数	signed long long sllValue -> s64 s64Value
BOOL	有符号的 32bit 整数	bool bIsValid -> BOOL bIsValid
LPCSTR	const 字符串指针	const char *pchName -> LPCSTR lpstrName
LPSTR	字符串指针	char *pchName -> LPSTR lpstrName
注：如果使用多维数组和指针时需要报项目经理审批		

【规则5-4】 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】：

```
typedef struct tagPuInfo
{
```

```
public:
    inline u32 GetIpValue() const;        // 取得 32 位的值, 返回值为本机序
    inline void SetIpValue(u32 dwIp);     // 设置 32 位的值, 传入参数本机序
    inline u16 GetPortValue() const;      // 取得 32 位的值, 返回值为本机序
    inline void SetPort(u16 wPort);       // 设置 32 位的值, 传入参数本机序
private:
    u32 dwPuIp;
    u16 wPuPort;
}TPuInfo;

// 取得 32 位的值, 返回值为本机序
inline u32 tagPuInfo::GetIpValue() const
{
    return ntohl(dwPuIp);
}

// 设置 32 位的值, 传入参数本机序
inline void tagPuInfo::SetIpValue(u32 dwIp)
{
    dwPuIp = htonl(dwIp);
}

// 取得 32 位的值, 返回值为本机序
inline u16 tagPuInfo::GetPortValue() const
{
    return ntohs(wPuPort);
}

// 设置 32 位的值, 传入参数本机序
inline void tagPuInfo::SetPort(u16 wPort)
{
    wPuPort = htons(wPort);
}
```

【建议5-1】 ▲防止使用既作为输入，又作为输出的变量。

说明：既作输入又做输出的变量会复用变量的含义，使用调用的流程发生跳转，不易阅读和维护。

【不好的示例】：

```
void GetProcess(u8 *pbyFlag)
{
    u8 byCondition = *pbyFlag;
    ... // 代码段
```

```
    if (...)
    {
        *pbyFlag = byRetValueOk;
    }
    else
    {
        *pbyFlag = byFailReason;
    }
    ... // 代码段
}
```

【好的示例】:

```
void GetProcess(const u8 *pbyFlag, *pbyResult)
{
    u8 byCondition = *pbyFlag;
    ... // 代码段
    if (...)
    {
        *pbyResult = byRetValueOk;
    }
    else
    {
        *pbyResult = byFailReason;
    }
    ... // 代码段
}
```

5.6 宏

【规则6-1】 用宏定义表达式时，要使用完备的括号。

说明：无。

示例参见规则 3-10。

【规则6-2】 使用宏时，不允许参数发生变化。

说明：宏的参数如果是一个表达式或一个会发生变化的值，则展开后实际的逻辑会变得比较复杂，不易控制，易出错。

【不好的示例】:

```
#define SQUARE(x) ((x) * (x))
...// 其他代码
```

```
// nArea = ((++nValue) * (++nValue)); nValue 的值累加了两次
// 与宏定义的字面表述不一致，容易引起混淆。
nArea = SQUARE(++nValue);
```

【好的示例】:

```
#define SQUARE(x) ((x) * (x))
...// 其他代码
nArea = SQUARE(nValue);
++nValue;
```

【规则6-3】 对于所有的常数，应使用 const 常量，enum 或宏定义。

说明：常数一般会在多处使用，定义为 const 常量，enum 枚举或宏可以减少后续修改维护的工作量和出错的可能。在 C++中建议更多地使用 const 常量或枚举。使用宏定义写起来简单，但要小心规范使用。

【规则6-4】 应使用 TRUE 和 FALSE 作为布尔表达式结果的标识符，禁用 0、1 等。

说明：不同的系统对真假的定义不一定相同，有些真为 1，有些为 0。不同的程序员对 0、1 的使用习惯也不同。把真假的定义统一为 TRUE 和 FALSE，可以增加代码的可移植性和可读性。

【不好的示例】:

```
bool bFound = 0;
bool bFlag = 1;
```

【好的示例】:

```
BOOL bFound = FALSE;
BOOL bTestFalg = TRUE;
```

【建议6-1】 ▲▲ 宏定义不能隐藏重要的细节，尽量避免有 return，break 等导致程序流程转向的语句。

说明：无。

【示例】:

```
#define FOR_ALL for (nIndex = 0; nIndex < MAX_FILE; nIndex++)
...// 其他代码
FOR_ALL
{
    adwFileStatus[nIndex] = INIT_FILE_STATUS;
}

#define CLOSE_FILE
```

```

    {
        fclose(fpLocalRec); \
        fclose(fpTempRec); \
        return; \
    }
```

5.7 函数

5.7.1 总则

【规则7-1.1】 函数的规模尽量限制在 100 行以内（不包括注释和空格行），避免函数中不必要语句，防止程序中的垃圾代码。

说明：如果超过 100 行，就应该考虑将此函数分为几个；太长的函数理解不容易，而且一般来说功能也比较复杂，不容易维护；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。

【规则7-1.2】 函数中申请的局部资源（如：文件句柄、socket 等），在同一函数中退出之前必须释放；如果不释放，须在函数说明中说明“由谁在什么时间释放”。

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。

【不好的示例】:

```
s32 ReceiveData(s32 nDataLen, s8 *pbyBuf)
{
    ... // 代码段 1
    s8 *pbyRcvBuf = malloc(nDataLen);
    ... // 代码段 2
    if (...)
    {
        // 忘记释放空间
        return PROCESS_ERROR_INVALID_HEAD;
    }
    ... // 代码段 3
    free(pbyRcvBuf);
    return PROCESS_OK;
}
```

【好的示例】:

```
s32 ReceiveData(s32 nDataLen, s8 *pbyBuf)
{
    ... // 代码段 1
```

```
s8 *pbyRcvBuf = (s8*)malloc(nDataLen);
... // 代码段 2
if (...)
{
    free(pbyRcvBuf);
    return PROCESS_ERROR_INVALID_HEAD;
}
... // 代码段 3
free(pbyRcvBuf);
return PROCESS_OK;
}
```

【规则7-1.3】 禁止直接调用非线程安全的接口，必须使用经过 OSP 封装过的对应接口。例如直接使用 inet_ntoa, localtime, asctime, ctime, gmtime 等。

说明：非线程安全的接口在内部实现时往往采用静态分配的存储空间。多次调用时，最后的调者返回结果是正确的，但以前的调用者返回的结果就被修改了。虽然系统一般会提供功能类似的线程安全的接口，但考虑到平台差异，减少出错的可能，我们要求统一使用 OSP 封装的接口，由 OSP 来保证接口的线程安全性。

【规则7-1.4】 编程时，要注意数据类型的强制转换，应尽量减少指针的强制类型转换。

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。

【规则7-1.5】 尽量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，一般都占用较多的系统资源（如栈空间），对程序的测试有一定影响；因此除非为了某些算法或功能的实现方便，应减少不必要的递归调用。

【规则7-1.6】 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。

【不好的示例】：

```
s32 GetSpace()
{
    ... // 代码段 1
    if (NULL == g_pbyFrameBuf)
    {
        g_pbyFrameBuf = (s8*)malloc(MAX_FRAME_SIZE);
    }
    ... // 代码段 2
}
```

```

        if (...)
        {
            free(g_pbyFrameBuf);
            // 释放了空间但没有置空指针
            return GET_FRAME_INFO_FAIL;
        }
        ... // 代码段 3
        return GET_FRAME_OK;
    }
void DecodeFrame()
{
    if (NULL == g_pbyFrameBuf)
    {
        return;
    }
    // 指针非空，但如果 GetSpace 中出错，g_pbyFrameBuf 指向的空间已释放，
    // 再调用就会导致当机
    DecOneFrame(g_pbyFrameBuf);
    ... // 代码段
}

```

【不好的示例】:

```

// 释放空间的同时将指针置空
#define FREE_PTR(p)      {free(p); p = NULL;}
s32 GetSpace()
{
    ... // 代码段 1
    if (NULL == g_pbyFrameBuf)
    {
        g_pbyFrameBuf = (s8*)malloc(MAX_FRAME_SIZE);
    }
    ... // 代码段 2
    if (...)
    {
        // 使用宏来防止人为的疏漏
        FREE_PTR(g_pbyFrameBuf);
        return GET_FRAME_INFO_FAIL;
    }
    ... // 代码段 3
    return GET_FRAME_OK;
}

void DecodeFrame()

```



```
{
    if (NULL == g_pbyFrameBuf)
    {
        return;
    }
    // 同样的调用，此处就没有问题了。
    DecOneFrame(g_pbyFrameBuf);
    ... // 代码段
}
```

【规则7-1.7】 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。

【不好的示例】：

```
#define PU_USER_NAME_LEN    (u8)16

s8 g_achUserName[PU_USER_NAME_LEN];

// 当 pchInputName 长度大于 16 时内存就会越界。
strcpy(g_achUserName, pchInputName);

// 看似没有越界，比较安全，实际上 16 字节的空间全被用完了。
// g_achUserName 后面内存时的内容不一定为空，如果第 17 个字节非零时，
// 后面的内容就会和前面的内存连在一起，形成一个大于 16 字节的字串。拿这个字串
// 再去进行其他操作就很危险了。
strncpy(g_achUserName, pchInputName, PU_USER_NAME_LEN);
```

【好的示例】：

```
s32 nLen = strlen(pchInputName);
if( nLen < PU_USER_NAME_LEN)
{
    strcpy(g_achUserName, pchInputName);
}
else
{
    // 输入字串过长，在调试窗口中报一下错。
    ::OspPrintf(TRUE, FALSE, "[xx 模块]输入名称过长,超过%d 个字符，内容被截掉!\n",
        PU_USER_NAME_LEN - 1);

    // 牺牲一个字节的空間，保证字串以'\0' 结尾。
    strncpy(g_achUserName, pchInputName, PU_USER_NAME_LEN);
}
```

```
g_achUserName[PU_USER_NAME_LEN - 1] = '\0';
}
```

【规则7-1.8】 编程时，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。

【规则7-1.9】 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。

【规则7-1.10】 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）；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：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；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】：

A 线程（一般是 API）先 SyncWaitInit，然后进行发送消息（或类似处理），然后 SyncWait。
B 线程（一般是回调通知）收到对应的应答消息（或类似处理）后，SyncInform 通知 A 线程。
如果 WaitId 不匹配，则丢弃。WaitId 一般使用用于标识该请求和应答关系的 id。
注意：A 必须先 SyncWaitInit 再发消息。如果颠倒，可能由于 B 收到应答特别快而导致先处理了 SyncInform。
该例子只考虑了单个 A 线程。多 A 的时候，相应的 TSynWait 需要多个，TSynWait 需要和消息相捆绑。如果不考虑多个线程进行消息收发，可对消息发送使用消息队列进行串行化。
该例子同时使用了线程间访问保护。

```
typedef struct tagSynWait
{
    SEMHANDLE m_hSem;           // 用于保护 TSynWait 结构
    SEMHANDLE m_hSemSyn;        // 用于通知
    u32 m_dwWaitId;             // 用于记录等待的 ID，防止两次等待出现串扰
} TSynWait;

//Global
s32 SynWaitCreate(TSynWait* ptSynWait)
{
    OspSemBCreate(&ptSynWait->m_hSem);
    OspSemBCreate(&ptSynWait->m_hSemSyn);
}
```

```
        ptSynWait->m_dwWaitId = 0;

        return 0;
    }

// A 线程
s32 SyncWaitInit(TSynWait* ptSynWait, u32 dwWaitId)
{
    OspSemTake(ptSynWait->m_hSem);
    ptSynWait->m_dwWaitId = dwWaitId;
    OspSemTakeByTime(ptSynWait->m_hSemSyn, 0);    // clear hSemSyn
    OspSemGive(ptSynWait->m_hSem);

    return 0;
}

s32 SyncWait(TSynWait* ptSynWait, u32 dwTimeOut)
{
    BOOL32 bGet;

    bGet = OspSemTakeByTime(ptSynWait->m_hSemSyn, dwTimeOut);
    if (!bGet)
    {
        return -1;    // wait timeout
    }

    return 0;
}

// B 线程
s32 SyncInform(TSynWait* ptSynWait, u32 dwWaitId)
{
    OspSemTake(ptSynWait->m_hSem);
    if (ptSynWait->m_dwWaitId == dwWaitId)
    {
        OspSemGive(ptSynWait->m_hSemSyn);
    }
    OspSemGive(ptSynWait->m_hSem);

    return 0;
}
```

5.7.2 接口、参数

【规则7-2.1】 明确规定对接口函数参数的合法性、有效性检查应由函数的提供者负责。

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；若函数的传入参数处理不当，会产生较严重的后果，函数在使用参数输入之前，应进行必要的合法性、有效性检查；在提供接口时，必须对参数进行详细的说明（参数的类型、输入参数/输出参数、参数的有效值等），并负责合法性、有效性检查。

总的原则是各模块保证自己的健壮性，不管外部传入的参数是否合法有效，尽最大可能使本模块的正常功能。如本模块无法判断入参数的有效性，则需与相关调用者明确责任，由其模块确保入参的正确性。

【规则7-2.2】 函数中参数的顺序，应按照使用频率和重要性从左至右排列。

说明：无。

【规则7-2.3】 避免设计多参数函数，一般不能多于 6 个，不使用的参数从接口中去掉；函数的参数表中所有参数的总长度不能大于 40 个字节，一些自定义的结构参量推荐采用传递指针/引用的方式进行参数传递，这样可以减少函数调用对系统堆栈的需求，同时防止因堆栈溢出引起的软件故障。

说明：目的减少函数间接口的复杂度，如果实在比较多，可以考虑通过传入结构指针/引用的方法解决。如果接口参数定义为类对象，则压栈时还会调用拷贝构造函数，负作用更大。

【不好的示例】：

```
#define MAX_PU_MSG_LEN      (u32) (16 * 1024)    // 最大长度 16k
typedef struct tagPuMsg
{
    u8  byMsgType;           // 消息类型
    u8  abyMsgBody[MAX_PU_MSG_LEN]; // 消息体
    ...                     // 其它成员声明

} TPuMsg, *PTPuMsg;

// 压栈空间过大，有多余的拷贝开销
void OnReceiveMsg(TPuMsg tMsg)
{
    ... // 代码段
}
```

【好的示例】：

```
#define MAX_PU_MSG_LEN      (u32) (16 * 1024)    // 最大长度 16k
typedef struct tagPuMsg
```

```
{
    u8  byMsgType;           // 消息类型
    u8  abyMsgBody[MAX_PU_MSG_LEN]; // 消息体
    ...                      // 其它成员声明

}TPuMsg, *PTPuMsg;

// 传引用
void OnReceiveMsg(const TPuMsg &tMsg)
{
    ... // 代码段
}

// 传指针
void OnReceiveMsg(const TPuMsg *ptMsg)
{
    ... // 代码段
}
```

【规则7-2.4】 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，如果函数没有参数，则用 void 填充。

说明：写上参数名字可以让读者更清楚地知道每个参数的含义。

【不好的示例】：

```
s32 GetValue();
void SetValue(s32, s32);
```

【好的示例】：

```
s32 GetValue(void);
void SetValue(s32 nWidth, s32 nHeight);
```

【规则7-2.5】 禁止使用函数作为函数的参数。

说明：将函数作为函数的参数看上去代码比较简洁，但函数返回值的类型不明显，且直接作为入参压入栈中，风险较高（见规则 7-2.3）。另外多个复杂函数放在行中，断点只能打一个，从调试来讲也不方便。

【不好的示例】：

```
GetProcess(GetTestCondition(), GetPuInfo());
```

【好的示例】：

```
BOOL bCanBeProcess = GetTestCondition();
TPuInfo tPuInfo;
GetPuInfo(&tPuInfo);
GetProcess(bCanBeProcess, &tPuInfo);
```

5.7.3 条件、循环、分支语句

【规则7-3.1】 在代码中，先按照正常情况的路径往下写，然后再写异常情况；应该把正常情况放在 if 后面而不是 else 后面。

说明：如果先写异常情况再写正常情况有利于代码简化和提高，可以不采用这条原则。

【不好的示例】：

```
if ( TRUE != bDateExpire)
{
    // 用户没有过期, 可登录
    bUserCanLogin = TRUE;
}
else
{
    // 用户过期, 不可登录
    bUserCanLogin = FALSE;
}
```

【好的示例】：

```
if ( TRUE == bDateExpire)
{
    // 用户过期, 不可登录
    bUserCanLogin = FALSE;
}
else
{
    // 用户没有过期, 可登录
    bUserCanLogin = TRUE;
}
```

【规则7-3.2】 if、while 等条件判断语句中必须是布尔表达式。

说明：无。

【不好的示例】：

```
s32 Value;
if (Value) // 会让人误解 Value 是布尔型变量
```

```
{
    ... 代码段
}

if (!Value) // 会让人误解 Value 是布尔型变量
{
    ... 代码段
}

// 或类似以下的语句
if (nValue = nInput)
{
    ... 代码段
}

if (GetCondition())
{
    ... 代码段
}
```

【好的示例】:

```
s32 nValue;
if (INVALID_VALUE == nValue)
{
    ... 代码段
}

if (INVALID_VALUE != nValue)
{
    ... 代码段
}

BOOL bCondOk = GetCondition();

if (bCondOk)
{
    ... 代码段
}
```

【规则7-3.3】 if 语句尽量加上 else 分支；switch-case 语句中，default 不能忽略，其后要加入异常处理；switch 语句的每个分支里的语句数一般应少于 20 行，以提高程序的可读性，否则考虑采用函数调用。

说明：无。

参见 1-1 排版示例

【规则7-3.4】 for 循环语句内的计数器不能在循环体内被修改，且必须是局部变量；建议不使用 i、j、k 作为循环的控制变量，应采用更有意义的变量。

说明：这样将减少出错的危险，并提高程序的可读性。

【不好的示例】：

```
for (s32 i = 0; i < 100; i++)
{
    if (abCanSkip[i])
    {
        i += nStep; // i 改变后循环次变得不确定，易出错
    }
    ... 代码段
}
```

【好的示例】：

```
for (s32 nIndex = 0; nIndex < sizeof(abCanSkip) / sizeof(abCanSkip[0]); nIndex++)
{
    if (abCanSkip[nIndex])
    {
        continue; // 效率可能会低一些，但逻辑简单，不易出错
    }
    ... 代码段
}
```

【规则7-3.5】 对于复杂的逻辑判断应建立中间的逻辑变量以简化代码和增加可读性；在单个 if、while 等判断语句中，尽量少于四个条件测试部分，且各测试部分必须使用括号括起来。

说明：如果条件较多，建议使用中间变量，将相关部分的测试结果赋给变量。

【规则7-3.6】 尽量减少循环嵌套层次，函数体内不要使用过多的嵌套，嵌套层次不能超过 3 层，否则应考虑重新设计程序。

说明：无。

【规则7-3.7】 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。

【不好的示例】：


```
for (s32 nIndex = 0; nIndex < MAX_RECT_NUM; nIndex++)
{
    if (SUM_AREA == byActType)
    {
        g_nTotalArea += atRect[nIndex].nArea;
    }
    else
    {
        g_nTotalWidth += atRect[nIndex].nWidth;
        g_nTotalHeight += atRect[nIndex].nHeight;
    }
}
```

【好的示例】:

```
if (SUM_AREA == byActType)
{
    for (s32 nIndex = 0; nIndex < MAX_RECT_NUM; nIndex++)
    {
        g_nTotalArea += atRect[nIndex].nArea;
    }
}
else
{
    for (s32 nIndex = 0; nIndex < MAX_RECT_NUM; nIndex++)
    {
        g_nTotalWidth += atRect[nIndex].nWidth;
        g_nTotalHeight += atRect[nIndex].nHeight;
    }
}
```

【规则7-3.8】 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。

【不好的示例】:

```
for (s32 nIndex = 0; nIndex < MAX_RECT_NUM; nIndex++)
{
    g_nTotalArea += atRect[nIndex].nArea * nFixRatio * GetZoomRatio();
}
```

【好的示例】:

```
// 当 GetZoomRatio() 返回值为确定值时可以从循环中提出来
s32 nRatio = nFixRatio * nZoomRatio;
for (s32 nIndex = 0; nIndex < MAX_RECT_NUM; nIndex++)
{
    g_nTotalArea += atRect[nIndex].nArea * nRatio;
}
```

5.7.4 内存、指针

【规则7-4.1】 尽量避免在函数中动态申请内存返回指针、在函数外释放的情况；如果必须使用仅限于哪些用于初始化不会被反复调用的函数。

说明：如果需要返回数组，建议采用以下函数接口形式：

```
void GetArray (LPSTR lpszBuf[], u16 wBufSize);
```

【规则7-4.2】 如果指针本身或指向内容明确不会改变时，应该制为 `const` 类型的指针，以加强编译器的检查。

说明：无。

【规则7-4.3】 指针释放时，应将指针置为 `NULL`，且尽量用宏释放。

说明：各产品组都已定义有相应用宏，请大家尽可能多地使用。

参见 7-1.6 示例

【规则7-4.4】 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。

说明：无。

【规则7-4.5】 动态内存的申请与释放必须配对，防止内存泄漏。其他的系统资源申请释放时也要注意释配对使用。

说明：无。

『示例』：

```
// 例 1
u8 *pbyRecvBuf = new u8[nLen];
...// 代码段
delete []pbyRecvBuf;
pbyRecvBuf = NULL;

// 例 2
CString strShowText = "Hello World!";
```

```
LPSTR *lpstrText = strShowText.GetBuffer();
...// 代码段
strShowText.ReleaseBuffer();

// 例 3
HDC hDC = ::GetDC(m_hSnapWnd);
::DrawText(hDC, "Hello world", nXPos, &rcShowText, DT_CENTER);
...// 代码段
::ReleaseDC(m_hSnapWnd, hDC);

// 例 4
::SetTimer(m_hMainWnd, nReDrawTimer, nTimerLap, NULL);
...// 代码段
::KillTimer(m_hMainWnd, nReDrawTimer);
```

【规则7-4.6】 定义指针类型的数据，应将“*”放在变量前。

说明：无。

【不好的示例】：

```
u32* pStatus, Score;    // 此处 Score 会被误认为是指针
```

【好的示例】：

```
u32 *pdwStatus = NULL;
u32 dwScore = 0;
```

5.7.5 返回值

【规则7-5.1】 函数定义应明确注明返回值，如果函数无返回值应明确表明 void，调用具有返回值的函数时，必须判断返回的结果，并根据返回值作相应处理。

说明：无。

【不好的示例】：

```
SetPuInfo(const TPuInfo *ptPuInfo)
{
    ...// 代码段
}
```

【好的示例】：

```
void SetPuInfo(const TPuInfo *ptPuInfo)
{
```

```
...// 代码段
}
```

【规则7-5.2】 不要把与函数返回值类型不同的变量，以编译系统默认转换方式或强制转换方式作为返回值返回。

说明：无。

【不好的示例】：

```
u32 GetAverageScore()
{
    u16 wScore = CalcAllItem();
    ...// 代码段
    return wScore;
}
```

【好的示例】：

```
u32 GetAverageScore()
{
    // 与函数返回值类型相同
    u32 dwScore = CalcAllItem();
    ...// 代码段
    return dwScore;
}
```

【规则7-5.3】 对于返回值不是基本类型的则采用指针/引用返回。

说明：应当确保指针或引用在函数返回后仍然有效才能按指针或引用返回，否则应当按值返回。特别不能返回临时变量。

【不好的示例】：

```
TPuInfo& GetPuInfo(void)
{
    TPuInfo tCurPu;

    GetCurInfoInfo(&tCurPu);

    ...// 代码段

    return tCurPu; // 返回的是临时变量
}
```

【好的示例】：

```
TPuInfo & TPuInfo::operatte = (const TPuInfo &tOtherPu)
{
    if (this == &tOtherPu)
    {
        return *this;
    }

    ...// 代码段
    return *this;    // 返回的是*this 的引用，无需拷贝过程
}
```

【规则7-5.4】 函数不要使用带有二义性的返回值。

说明：无。

【不好的示例】：

```
// 若成功返回文件句柄；若失败，返回错误码
FHANDLE OpenFile(const s8 *lpstrFilePath, u32 dwOpenType)
{
    ...// 代码段
}
```

【好的示例】：

```
// pdwResult 为操作返回结果，如果成功则*pdwResult 为 SUCCESS,
// 函数返回值为文件句柄；如果失败则*pdwResult 为错误码，返回值为无效值
FHANDLE OpenFile(const s8 *lpstrFilePath, u32 dwOpenType, u32 *pdwResult)
{
    ...// 代码段
}
```

【建议7-5.1】 ▲▲ 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。

5.8 可维护性

【规则8-1】 预编译条件不应分离一条完整的语句。

说明：无。

【不好的示例】：

```
if ((TRUE == bGateIsOpen)
```

```
#ifdef DEBUG
    || TRUE == bWndIsOpen)
#endif
)
...// 代码段
```

【好的示例】:

```
#ifdef DEBUG
if ((TRUE == bGateIsOpen) || (TRUE == bWndIsOpen))
#else
if (TRUE == bGateIsOpen)
#endif
```

【规则8-2】 在宏定义中，建议合并预编译条件（通过预编译将宏分开，代码保持一份）。

说明：

【不好的示例】:

```
#ifdef USE_IN_LARGE_SCALE
for (s32 nIndex = 0; nIndex < 1000000; nIndex++)
#else
for (s32 nIndex = 0; nIndex < 2000; nIndex++)
#endif
```

【好的示例】:

```
#ifdef USE_IN_LARGE_SCALE
#define MAX_PROCESS_NUM    (s32)1000000
#else
#define MAX_PROCESS_NUM    (s32)2000
#endif // USE_IN_LARGE_SCALE

for (s32 nIndex = 0; nIndex < MAX_PROCESS_NUM; nIndex++)
```

【规则8-3】 结构中元素布局合理，一行只定义一个元素。

说明：无。

【不好的示例】:

```
typedef struct tagAgentErrorMsg
{
    u8 byErrorNo, u8 abyErrorMsg[MAX_ERROR_MSG_LEN];
}TAgentErrorMsg;
```

【好的示例】:

```
typedef struct tagAgentErrorMsg
{
    u8 byErrorNo;
    u8 abyErrorMsg[MAX_ERROR_MSG_LEN];
}TAgentErrorMsg;
```

【规则8-4】 枚举值按照从小到大顺序定义。

说明:

【不好的示例】:

```
enum WEEKDAY
{
    SUNDAY      = 7,
    MONDAY      = 1,
    TUESDAY     = 2,
    WEDNESDAY   = 3,
    TUESDAY     = 4,
    FRIDAY      = 5,
    SATURDAY    = 6
};
```

【好的示例】:

```
enum WEEKDAY
{
    MONDAY      = 1,
    TUESDAY     = 2,
    WEDNESDAY   = 3,
    TUESDAY     = 4,
    FRIDAY      = 5,
    SATURDAY    = 6,
    SUNDAY      = 7
};
```

【规则8-5】 禁止使用复杂的操作符组合。

说明: 不要把“++”、“--”操作符与其他如“+=”、“-=”等组合在一起形成复杂奇怪的表达式。

示例参见 4-3

【规则8-6】 用 sizeof 来确定结构、联合或变量占用的空间。

说明：这样可提高程序的可读性、可维护性，同时也增加了程序的可移植性。

【示例】：

```
TScore atAllScore[MAX_STUDENT_NUM];
memset(&atAllScore, 0, sizeof(atAllScore));
for(s32 nIndex = 0; nIndex < sizeof(atAllScore) / sizeof(atAllScore[0]); nIndex++)
{
    ...// 代码段
}

TPuInfo *patPuInfo = (TPuInfo*)malloc(sizeof(TPuInfo) * nPuNum);
```

【规则8-7】 避免相同的代码段在多个地方出现。

说明：当某段代码需在不同的地方重复使用时，应根据代码段的规模大小使用函数调用或宏调用的方式代替。这样，对该代码段的修改就可在一处完成，增强代码的可维护性。

【规则8-8】 不要滥用 goto 语句，但如果是用 goto 语句可以使处理变得更加简单时，可以考虑，但必须注明使用的原因，并向直接上级申请。

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。

5.9 可测试性

【规则9-1】 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存周期中很重要的一个阶段，如何对软件进行较全面、高效率的测试并尽可能地找出软件中的错误就成为很关键的问题。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，为单元测试、集成测试及系统联调提供方便。

【规则9-2】 模块要有一套统一的为集成测试和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。

【规则9-3】 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（通过调测开关）。

说明：为单元测试而准备。

【规则9-4】 禁止使用程序自带的断言；应广泛采用自定义的断言，以增强程序的调试功能；对较复杂的断言应该加上明确的注释；正式软件产品中应把断言及其它调测代码去掉（即把有关的调测开关

关掉)，加快软件运行速度。

说明：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，不能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；断言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，用其它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。

在程序设计中，对异常情况的处理方法有两种：若该异常情况会在正式版本中出现，则应保留对该情况的处理分支；若该异常情况只会在调试阶段出现，则应对该异常做断言检查。

指针断言：提倡对指针的合法性设置断言。

前束断言：在函数的人口处应设置前束断言，对本函数输入参数在正常运行时不可能出现的情况做断言检查；该断言主要用于在调试版本中尽早暴露函数调用者的错误。

后束断言：在函数的出口处设置后束断言，对本函数返回值在正常运行时不可能出现的情况做断言检查；该断言主要用于在调试版本中尽早暴露函数处理过程中的错误。

循环不变式：在各种可变条件循环中，对在循环过程中应满足的条件作断言检查；该断言主要用于在调试版本中检查循环的合法性和数据结构的完整性。

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）。

【规则9-5】 在软件系统中设置与取消有关测试手段时，不能对软件实现的功能等产生影响。

说明：即有测试代码的软件和关掉测试代码的软件，在功能行为上应一致。

【规则9-6】 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；出于某种其它目的，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。

【规则9-7】 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。


说明：无。

【规则9-8】 要求用编译开关来切换软件的 DEBUG 版和正式版，但不要同时存在正式版本和 DEBUG 版本的源文件，以减少维护的难度。

说明：无。

【规则9-9】 对 Release 版本，所有的调试开关缺省状态下应全部关闭（调试命令不能有任何输出，必要的时候再通过手动的方式打开）。

说明：无。

【建议9-1】  编写防错程序，然后在处理错误之后可用断言宣布发生错误。

说明：无。

【示例】：

```


#ifdef _EXAM_ASSERT_TEST // 使用断言测试
void AssertReport(const char * lpFileName, u32 dwLineNo)
{
    printf("\n[EXAM]Error Report:%s,line %d\n", lpFileName, dwLineNo);
}

#define ASSERT_REPORT(cond) \
    if (cond) \
    { \
        NULL; \
    } \
    else \
    { \
        AssertReport(__FILE__, __LINE__); \
    }
#else
#define ASSERT_REPORT(cond) NULL
#endif // _EXAM_ASSERT_TEST


void RecordInput(LPSTR lpstrName)
{
    ASSERT((NULL != lpstrName) && (strlen(lpstrName) > 0));

    if ((NULL == lpstrName) || 0 == strlen(lpstrName))
    {
        return;
    }
    ... // 代码段
}

```

【建议9-2】  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。

【建议9-3】  模块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。

5.10 程序效率

【建议10-1】▲▲ 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。

【建议10-2】▲▲ 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。

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。

【建议10-3】▲▲ 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。

【建议10-4】▲▲ 在优化程序的效率时，应当先找出限制效率的“瓶颈”，不要在无关紧要之处优化。

说明：无。

【建议10-5】▲▲ 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。

【不好的示例】：

```
// 如下表达式运算可能会占用较多的 CPU 资源
#define PAI 3.1416

fRadius = fCircleLen / (2 * PAI);
```

【好的示例】：

```
#define PAI_RECIPROCAL (float)(1 / 3.1416 * 2) // 编译时将生成具体的浮点数

fRadius = fCircleLen * PAI_RECIPROCAL;
```

【建议10-6】▲▲ 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。

5.11 代码编辑、编译、审查

【规则11-1】 在产品软件（项目组）中，要统一编译开关选项，并要求打开编译器的所有告警开关对程序进行编译。

说明：无。

【规则11-2】 同产品软件（项目组）内，要求使用相同的编译器，并使用相同的设置选项。

说明：无。

【规则11-3】 编译时，缺省级别的 warning 要尽可能的少。

说明：无。

【规则11-4】 测试部测试产品之前，应对代码进行抽查及评审。

说明：无。

【建议11-1】 ▲▲ 通过代码走读及审查方式对代码进行检查。

说明：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。

【建议11-1】 ▲▲ 编写代码时要注意随时保存，并定期提交，防止由于断电、硬盘损坏等原因造成代码丢失。

说明：无。

【建议11-2】 ▲▲ 合理地设计软件系统目录，方便开发人员使用。

说明：方便、合理的软件系统目录，可提高工作效率。目录构造的原则是方便有关源程序的存储、查询、编译、链接等工作，同时目录中还应具有工作目录----所有的编译、链接等工作应在此目录中进行，工具目录----有关文件编辑器、文件查找等工具可存放在此目录中。

【建议11-3】 ▲▲ 要小心地使用编辑器提供的块拷贝功能编程。

说明：当某段代码与另一段代码的处理功能相似时，许多开发人员都用编辑器提供的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错误，比如当使用的是全局变量时，就有可能使某种错误隐藏下来。

类似的代码应提炼出来成为共用的接口，而不是简单的 copy&paste。

6 规范示例代码

为方便大家查阅，把示例源代码附在这里。



sample.cpp