iOS 软件代码规范

目　　录

1. 指导原则

【原则 1-1】首先是为人编写程序，其次才是计算机。

说明：这是软件开发的基本要点，软件的生命周期贯穿产品的开发、测试、生产、用户使用、版本升级和后期维护等长期过程，只有易读、易维护的软件代码才具有生命力。

【原则 1-2】保持代码的简明清晰，避免过分的编程技巧。

说明：简单是最美。保持代码的简单化是软件工程化的基本要求。不要过分追求技巧，否则会降低程序的可读性。

【原则 1-3】编程时首先达到正确性，其次考虑效率。

说明：编程首先考虑的是满足正确性、健壮性、可维护性、可移植性等质量因素，最后才考虑程序的效率和资源占用。

【原则 1-4】函数（方法）是为一特定功能而编写，不是万能工具箱。

说明：方法是一个处理单元，是有特定功能的，所以应该很好地规划方法，不能是所有东西都放在一个方法里实现

【原则 1-5】鼓励多加注释。

说明：注释为了让人更容易理解代码，而不是为了敷衍。例如 sort 函数，注释写的是‘排序’，那么该注释无意义，因为看名字就知道是排序。例如注释为：‘希尔排序实现’，则有意义，因为阅读者能知道名字以外更多的信息。

【原则 1-6】内存空间在哪分配在哪释放。

2. 布局

程序布局的目的是显示出程序良好的逻辑结构，提高程序的准确性、连续性、可读性、可维护性。更重要的是，统一的程序布局和编程风格，有助于提高整个项目的开发质量，提高开发效率，降低开发成本。同时，对于普通程序员来说，养成良好的编程习惯有助于提高自己的编程水平，提高编程效率。因此，统一的、良好的程序布局和编程风格不仅仅是个人主观美学上的或是形式上的问题，而且会涉及到产品质量，涉及到个人编程能力的提高，必须引起大家重视。

2.1. 文件布局

【规则 2-1-1】遵循统一的布局顺序来书写头文件。

说明：以下内容如果某些节不需要，可以忽略。但是其它节要保持该次序。

头文件布局:

文件头（参见“注释”一节）
#import （依次为标准库头文件、非标准库头文件）
全局宏
常量定义
全局数据类型
类定义

【规则 2-1-2】遵循统一的布局顺序来书写实现文件。

说明：以下内容如果某些节不需要，可以忽略。但是其它节要保持该次序。

实现文件布局：

文件头（参见“注释”一节）
#import （依次为标准库头文件、非标准库头文件）
文件内部使用的宏
常量定义
文件内部使用的数据类型
全局变量
本地变量（即静态全局变量）
类的实现

【规则 2-1-3】包含标准库头文件用尖括号 <>，包含非标准库头文件用双引号“”。

正例：

```
#import   <stdio.h>
#import “heads.h”
```

2.2. 基本格式

【规则 2-2-1】if、else、else if、for、while、do 等语句，不论执行语句有多少都要加 { }，而且左大括号要和条件语句在同一行。大括号和语句之间一个空格。

说明：遵循苹果的代码规范，以及很多知名库的规范。

正例：

```
if (varible1 < varible2) {
```

```
        varible1 = varible2;
    }
```

正例：

```
    if (condition1) {
        // do something
    } else if (condition2) {
        // do something
    } else {
        // do something
    }
```

反例：下面的代码执行语句紧跟 if 的条件之后，而且没有加{}，违反规则。

```
    if (varible1 < varible2) varible1 = varible2;
```

【规则 2-2-2】定义指针类型的变量，*应放在变量前。

正例：

```
    float    *pfBuffer;
```

反例：

```
    float*    pfBuffer;
```

〖建议 2-2-3〗源程序中关系较为紧密的代码应尽可能相邻。

说明：这样便于程序阅读和查找。

正例：

```
    iLength      = 10;
    iWidth       = 5;        // 矩形的长与宽关系较密切，放在一起。
    StrCaption   = “Test”;
```

反例：

```
    iLength      = 10;
    strCaption   = “Test”; //  导致长和宽分开赋值
    iWidth       = 5;
```

2.3. 对齐

【规则 2-3-1】 禁止使用 TAB 键，必须使用空格进行缩进。缩进为 4 个空格。

说明：消除不同编辑器对 TAB 处理的差异，有的代码编辑器可以设置用空格代替 TAB 键。

【规则 2-3-2】程序的分界符‘{’应和代码在同一行。{ }之内的代码块使用缩进规则对齐。

说明：遵循苹果代码规范。

正例：

```
(void) Function:(int)iVar {          // 独占一行并与引用语句左对齐。
    while (condition) {
        doSomething();     //缩进 4 格
    }
}
```

【规则 2-3-3】结构型的数组、多维的数组如果在定义时初始化，按照数组的矩阵结构分行书写。

正例：

```
int aiNumbers[4][3] = {
        1, 1, 1,
        2, 4, 8,
        3, 9, 27,
        4, 16, 64
}
```

【规则 2-3-4】相关的赋值语句等号对齐。

正例：

```
tPDBRes.wHead      =   0;
tPDBRes.wTail      =   wMaxNumOfPDB - 1;
tPDBRes.wFree      =   wMaxNumOfPDB;
tPDBRes.wAddress   =   wPDBAddr;
tPDBRes.wSize      =   wPDBSize;
```

〖建议 2-3-5〗在 switch 语句中，每一个 case 分支和 default 要用{ }括起来，{ }中的内容需要缩进。

说明：使程序可读性更好。

正例：

```
switch (iCode) {
    case 1: {
        //DoSomething();      // 缩进 4 格
        break;
    }
    case 2: {                              // 每一个 case 分支和 default 要用{}括起来
        //DoOtherThing();
```

```
            break;
        }
        …                           // 其它 case 分支
        default: {
            //DoNothing();
            break;
        }
    }
```

2.4. 空行空格

【规则 2-4-1】函数(方法)块之间使用两个空行分隔。

说明：空行起着分隔程序段落的作用。适当的空行可以使程序的布局更加清晰。

正例：

```
- (void) hey {
    [hey 实现代码]
}
// 空一行
// 空一行
- (void) ack {
    [ack 实现代码]
}
```

反例：

```
-(void) Hey(void) {
    [Hey 实现代码]
}
- (void) Ack(void) {
    [Ack 实现代码]
}
// 两个函数的实现是两个逻辑程序块，应该用空行加以分隔。
```

【规则 2-4-2】一元操作符如“!”、“~”、“++”、“--”、“*”、“&”（地址运算符）等前后不加空格。“[ ]”、“. ”、“->”这类操作符前后不加空格。

正例：

```
!bValue
~iValue
++iCount
*strSource
&fSum
```

```
aiNumber[i] = 5;
tBox.dWidth
tBox->dWidth
```

【规则 2-4-3】多元运算符和它们的操作数之间至少需要一个空格。

正例：

```
fTotal     -=    fValue
iNumber +=    2;
```

【规则 2-4-4】关键字之后要留空格。

说明：if、for、while 等关键字之后应留一个空格再跟左括号‘(’，以突出关键字。

【规则 2-4-5】函数名之后不要留空格。

说明：函数名后紧跟左括号‘(’，以与关键字区别。

【规则 2-4-6】方法名与形参不能留空格，返回类型与方法标识符有一个空格。

说明：方法名后紧跟’:’，然后紧跟形参， 返回类型’(‘与’-‘之间有一个空格。

正例：

```
-␣(BOOL)␣ hasChild:(CCSprite *)leafSprit ␣{
    return [children contains:leafSprite];
}
```

【建议 2-4-7】‘(’向后紧跟，‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格。‘,’之后要留空格。‘;’不是行结束符号时其后要留空格。

正例：

```
例子中的 ␣ 代表空格。
for (int i ␣=␣ 0;␣ i ␣<␣ MAX_BSC_NUM;␣ i++){
    DoSomething(iWidth,␣ iHeight);
}
```

【规则 2-4-8】注释符与注释内容之间要用一个空格进行分隔。

正例：

```
/* 注释内容 */
// 注释内容
```

反例：

```
/*注释内容*/
//注释内容
```

2.5. 断行

【规则 2-5-1】长表达式（超过 100 列）要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐。

Google 的 80 字符的标准有点少，这导致过于频繁的换行（Objectve-C 的代码一般都很长）通过“Xcode => Preferences => TextEditing => 勾选 Show Page Guide / 输入 100 => OK”来设置提醒

说明：条件表达式的续行在第一个条件处对齐。
for 循环语句的续行在初始化条件语句处对齐。
函数调用和函数声明的续行在第一个参数处对齐。
赋值语句的续行应在赋值号处对齐。

正例：

```
if ((iFormat == CH_A_Format_M)
    && (iOfficeType == CH_BSC_M)) { // 条件表达式的续行在第一个条件处对齐
     doSomething();
}

for (long_initialization_statement;
      long_condiction_statement;        // for 循环语句续行在初始化条件语句处对齐
      long_update_statement) {
      doSomething();
}

// 函数声明的续行在第一个参数处对齐
BYTE ReportStatusCheckPara(HWND hWnd,
                                    BYTE ucCallNo,
                                    BYTE ucStatusReportNo);

// 赋值语句的续行应在赋值号处对齐
fTotalBill = fTotalBill + faCustomerPurchases[iID]
             + fSalesTax(faCustomerPurchases[iID]);
```

【规则 2-5-2】函数(方法)声明时，类型与名称不允许分行书写。

正例：

```
extern double FAR CalcArea(double dWidth, double dHeight);
```

反例：

```
    extern double FAR
CalcArea(double dWidth, double dHeight);
```

3. 注释

注释有助于理解代码，有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息，而不是代码表面意义的简单重复。

【规则 3-1】C 语言的注释符为“/* … */”。C++语言中，多行注释采用“/* … */”，单行注释采用“// …”。

【规则 3-2】一般情况下，源程序有效注释量大概在 10%。

说明：注释的原则是有助于对程序的阅读理解，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息。

【建议 3-3】注释使用中文。

【规则 3-4】文件头部必须进行注释，包括：.h 文件、.c 文件、.m 文件、.inc 文件、.def 文件、编译说明文件.cfg 等。

说明：注释必须列出：版权信息、内容摘要、作者、完成日期、修改信息等。修改记录部分建议在代码做了大修改之后添加修改记录。备注：文件名称，内容摘要，作者等部分一定要写清楚。

正例：

下面是文件头部的中文注释：

```
/*********************************************************************
* 版权所有 (C)2013 南天电脑有限公司
*
* 文件名称： // 文件名
* 内容摘要： // 简要描述本文件的内容，包括主要模块、函数及其功能的说明
* 其它说明： // 其它内容的说明
* 作　　者： // 输入作者名字及单位
* 完成日期： // 输入完成日期，例：2011 年 11 月 29 日

* 修改记录 1：// 修改历史记录，包括修改日期、修改者及修改内容
*　　修改日期 ：
*　　修 改 人 ：
```

```
*      修改内容：//修改原因以及修改内容说明
* 修改记录 2：…
**********************************************************************/
```

【规则 3-5】方法头部应进行必要的描述，如果函数能望文知义，则可以不加注释。

说明：注释可包含：函数名称、功能描述、输入参数、输出参数、返 回 值、修改信息等。
备注：方法名称、功能描述要正确描述。

正例：

```
/**********************************************************************
* 功能描述：  // 方法功能、性能等的描述
* 输入参数：  // 输入参数说明，包括每个参数的作用、取值说明及参数间关系
* 返 回 值：  // 方法返回值的说明
* 其它说明：  // 其它说明
**********************************************************************/
```

正例：

```
/** 说明：统计指定目录下的文件个数 **/
```

【规则 3-6】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。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

正例：

如下书写比较结构清晰

```
/* 获得子系统索引 */
iSubSysIndex = aData[iIndex].iSysIndex;

/* 代码段 1 注释 */
[ 代码段 1 ]

/* 代码段 2 注释 */
[ 代码段 2 ]
```

反例 1：

如下例子注释与描述的代码相隔太远。

```
/* 获得子系统索引 */

iSubSysIndex = aData[iIndex].iSysIndex;
```

反例 2：

如下例子注释不应放在所描述的代码下面。

```
iSubSysIndex = aData[iIndex].iSysIndex;
/* 获得子系统索引 */
```

反例 3：

如下例子，显得代码与注释过于紧凑。

```
/* 代码段 1 注释 */
[ 代码段 1 ]
/* 代码段 2 注释 */
[ 代码段 2 ]
```

【规则 3-7】全局变量要有详细的注释，包括对其功能、取值范围、访问信息及访问时注意事项等的说明。

正例：

```
/*
* 变量作用说明
* 变量值说明
*/
BYTE g_ucTranErrorCode;
```

【规则 3-8】注释与所描述内容进行同样的缩排。

说明：可使程序排版整齐，并方便注释的阅读与理解。

正例：

如下注释结构比较清晰

```
- (int) doSomething {
    /* 代码段 1 注释 */
    [ 代码段 1 ]

    /* 代码段 2 注释 */
    [ 代码段 2 ]
}
```

反例：

如下例子，排版不整齐，阅读不方便；

```
- (int) DoSomething {{
/* 代码段 1 注释 */
    [ 代码段 1 ]

/* 代码段 2 注释 */
    [ 代码段 2 ]
```

}

〖建议 3-9〗尽量避免在注释中使用缩写，特别是不常用缩写。

说明：在使用缩写时，应对缩写进行必要的说明。

4. 命名规则

4.1. 基本规则

好的命名规则能极大地增加可读性和可维护性。同时，对于一个有上百个人共同完成的大项目来说，统一命名约定也是一项必不可少的内容。本章对程序中的所有标识符（包括变量名、常量名、函数名、类名、结构名、宏定义等）的命名做出约定。

【规则 4-1】标识符要采用英文单词或其组合，便于记忆和阅读，切忌使用汉语拼音来命名。

说明：标识符应当直观且可以拼读，可望文知义，避免使人产生误解。程序中的英文单词一般不要太复杂，用词应当准确。

【规则 4-2】严格禁止在名字中间使用下划线，下划线只能在标识符头。(预编译开关除外)。

说明：这样做的目的是为了使程序易读。

【规则 4-3】程序中不要出现仅靠大小写区分的相似的标识符。

【规则 4-4】类名。

类名（及其 category name 和 protocal name）的首字母大写的形式分割单词。

【规则 4-5】方法名用小写字母开头,其余单词大写的单词组合而成。

说明：方法名力求清晰、明了，通过方法名就能够判断方法的主要功能。方法名中不同意义字段之间不要用下划线连接，而要把每个字段的首字母大写以示区分。方法命名采用大小写字母结合的形式，但专有名词不受限制。

【规则 4-6】类中的成员变量以下划线‘_’开头。属性命名，与去掉“_”前缀的成员变量相同，使用@synthesize 将二者联系起来。

正例：

```
@interface CMyTestClass {
    BOOL _isRunning;
    NString *_destFileName;
}
@property (copy, nonatomic) NSString *destFileName;
```

.m 中

```
@synthesize destFileName = _ destFileName;
```

【规则 4-7】宏、常量名都要使用大写字母, 用下划线 ‘_’ 分割单词。预编译开关的定义使用下划线 ‘_’ 开始。

正例：如 DISP_BUF_SIZE、MIN_VALUE、MAX_VALUE 等等。

【规则 4-8】程序中局部变量不要与全局变量重名。

说明：尽管局部变量和全局变量的作用域不同而不会发生语法错误，但容易使人误解。

【规则 4-9】使用一致的前缀来区分变量的作用域。

说明：变量活动范围前缀规范如下（少用全局变量）：

g_	：	全局变量
s_	：	模块内静态变量
空	：	局部变量不加范围前缀

〖建议 4-10〗尽量避免名字中出现数字编号，如 Value1、Value2 等，除非逻辑上的确需要编号。

〖建议 4-11〗标识符前最好不加项目、产品、部门的标识。

说明：这样做的目的是为了代码的可重用性。

5. 变量，常量，宏与类型

变量、常量和数据类型是程序编写的基础，它们的正确使用直接关系到程序设计的成败，变量包括全局变量、局部变量和静态变量，常量包括数据常量和指针常量，类型包括系统的数据类型和自定义数据类型。本章主要说明变量、常量与类型使用时必须遵循的规则和一些需注意的建议，关于它们的命名，参见命名规则。

5.1. 变量、常量以及宏

【规则 5-1-1】一个变量有且只有一个功能，尽量不要把一个变量用作多种用途。

说明：一个变量只用来表示一个特定功能，不能把一个变量作多种用途，即同一变量取值不同时，其代表的意义也不同。

【规则 5-1-2】循环语句与判断语句中，不允许对其它变量进行计算与赋值。

说明：循环语句只完成循环控制功能，if 语句只完成逻辑判断功能，不能完成计算赋值功能。

正例：

```
do {
    [处理语句]
    cInput = GetChar();
} while (cInput == 0);
```

反例：

```
do {
    [处理语句]
} while (cInput = GetChar());
```

【规则 5-1-3】宏定义中如果包含表达式或变量，表达式和变量必须用小括号括起来。

说明：在宏定义中，对表达式和变量使用括号，可以避免可能发生的计算错误。

正例：

```
#define    HANDLE(A, B)     (( A ) / ( B ))
```

反例：

```
#define    HANDLE(A, B)     (A / B)
```

【规则 5-1-4】宏名大写字母。

正例：

```
#define    BUTTON_WIDTH       (int)320
```

反例：

```
#define    kButtonWidth       (int)320
```

【规则 5-1-5】宏常量要指定类型。

说明：不同的编译器，默认类型不一样。

正例：

```
#define    BUTTON_WIDTH       (int)320
```

反例：

```
#define    BUTTON_WIDTH       320
```

〖建议 5-1-6〗对于全局变量通过统一的函数访问。

说明：可以避免访问全局变量时引起的错误。

〖建议 5-1-7〗最好不要在语句块内声明局部变量。

正例：

```
 int tmpRows;
for (int i = 0; i < maxFiles; i++) {
      tempRows = [self getFileCount:fileNames[i]];
      ……
}
```

反例：

```
for (int i = 0; i < maxFiles; i++) {
      int tempRows = [self getFileCount:fileNames[i]];
      ……
}
```

5.2. 类型

【规则 5-2-1】Protocol

▪类型标示符、代理名称、尖括号间不留空格。

▪该规则同样适用于：类声明、实例变量和方法声明。

▪如果类声明中包含多个 Protocol ，每个 Protocol 占用一行，缩进 2 个字符。

正例：

```
@interface MyProtocoledClass : NSObject<NSWindowDelegate> {
    @private
        id<MyFancyDelegate> _delegate;
}

- (void) setDelegate:(id<MyFancyDelegate>)aDelegate;

@end
```

正例：

```
@interface CustomViewController : ViewController<
    AbcDelegate,
    DefDelegate
> {
    ...
}
```

〖建议 5-2-2〗结构是针对一种事务的抽象，功能要单一，不要设计面面俱到的数据结构。

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。

正例：

```
typedef struct TeacherStruct {
    BYTE    aucName[8];
    BYTE    ucSex;
}Teacher;

typedef struct StudentStruct {
    BYTE    ucName[8];
    BYTE    ucAge;
    BYTE    ucSex;
    WORD    wTeacherInd;
}Student;
```

反例：

如下结构不太清晰、合理。

```
typedef struct PeopleStruct {
        BYTE     aucName[8];
        BYTE     ucAge;
        BYTE     ucSex;
        WORD    wTeacherInd;
 }People;
```

6. 表达式与语句

表达式是语句的一部分，它们是不可分割的。表达式和语句虽然看起来比较简单，但使用时隐患比较多。本章归纳了正确使用表达式和 if、for、while、goto、switch 等基本语句的一些规则与建议。

【规则 6-1】一条语句只完成一个功能。

说明：复杂的语句阅读起来，难于理解，并容易隐含错误。变量定义时，一行只定义一个变量。

正例：

```
int    iHelp;
int    iBase;
int    iResult;

iHelp     =    iBase;
iResult    =    iHelp + GetValue(&iBase);
```

反例：

```
int iBase, iResult;                          // 一行定义多个变量

iResult = iBase + GetValue(&iBase); // 一条语句实现多个功能，iBase 有两种用途。
```

【规则 6-2】在表达式中使用括号，使表达式的运算顺序更清晰。

说明：由于将运算符的优先级与结合律熟记是比较困难的，为了防止产生歧义并提高可读性，即使不加括号时运算顺序不会改变，也应当用括号确定表达式的操作顺序。

正例：

```
if    (((iYear % 4 == 0) && (iYear % 100 != 0)) || (iYear % 400 == 0))
```

反例：

```
if (iYear % 4 == 0 && iYear % 100 != 0 || iYear % 400 == 0)
```

【规则 6-3】避免表达式中的附加功能，不要编写太复杂的复合表达式。

说明：带附加功能的表达式难于阅读和维护，它们常常导致错误。对于一个好的编译器，下面两种情况效果是一样的。

正例 ：

```
aiVar[1] = aiVar[2] + aiVar[3];
aiVar[4]++;
iResult = aiVar[1] + aiVar[4];
aiVar[3]++;
```

反例：

```
iResult = (aiVar[1] = aiVar[2] + aiVar[3]++) + ++aiVar[4] ;
```

【规则 6-4】不可将布尔变量和逻辑表达式直接与 YES、NO 或者 1、0 进行比较。

说明：TURE 和 FALSE 的定义值是和语言环境相关的，且可能会被重定义的。

正例：

```
设 bFlag  是布尔类型的变量

if (bFlag)          // 表示 flag 为真
if (!bFlag)         // 表示 flag 为假
```

反例：

```
设 bFlag  是布尔类型的变量

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

【规则 6-5】在条件判断语句中，当整型变量与 0 比较时，不可模仿布尔变量的风格，应当将整型变量用“==”或“!=”直接与 0 比较。

正例：

```
if (iValue == 0)

if (iValue != 0)
```

反例：

```
if (iValue)           // 会让人误解 iValue 是布尔变量

if (!iValue)
```

【规则 6-6】不可将浮点变量用“==”或“!=”与任何数字比较。

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

正例：

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

反例：

```
if (fResult == 0.0)      // 隐含错误的比较
```

其中 EPSINON 是允许的误差（即精度）。

【规则 6-7】应当将指针变量用“==”或“!=”与 nil 比较。

说明：指针变量的零值是“空”（记为 NULL），即使 NULL 的值与 0 相同，但是两者意义不同。

正例：

```
if (pHead == nil)    // pHead 与 NULL 显式比较，强调 pHead 是指针变量
if (pHead != nil)
```

反例：

```
if (pHead == 0)            // 容易让人误解 pHead 是整型变量
if (pHead != 0)
```

或者

```
if (pHead)                 // 容易让人误解 pHead 是布尔变量
if (!pHead)
```

【规则 6-8】在 switch 语句中，每一个 case 分支必须使用 break 结尾，最后一个分支必须是 default 分支。

说明：避免漏掉 break 语句造成程序错误。同时保持程序简洁。

对于多个分支相同处理的情况可以共用一个 break，但是要用注释加以说明。

正例：

```
switch (iMessage) {
    case SPAN_ON: {
            [处理语句]
            break;
    }
    case SPAN_OFF: {
            [处理语句]
            break;
    }
    default: {
             [处理语句]
             break;
```

```
        }
    }
```

【规则 6-9】不可在 for 循环体内修改循环变量，防止 for 循环失去控制。

〖建议 6-10〗循环嵌套层数不大于 3 次。

〖建议 6-11〗do while 语句和 while 语句仅使用一个条件。

说明：保持程序简洁。如果需要判断的条件较多，建议用临时布尔变量先计算是否满足条件。

正例：

```
BOOL bCondition;

do {
    ……..
    bCondition = ((tAp[iPortNo].bStateAcpActivity != PASSIVE)
                || (tAp[iPortNo].bStateLacpActivity != PASSIVE))
                && (abLacpEnabled[iPortNo])
                && (abPortEenabled[iPortNo])

} while (bCondition);
```

7. 函数、方法、接口

〖规则 7-1〗在组件接口中应该尽量少使用外部定义的类型（重用，减少耦合）。

〖建议 7-2〗避免函数有太多的参数，参数个数尽量控制在 5 个以内。

说明：如果参数太多，在使用时容易将参数类型或顺序搞错，而且调用的时候也不方便。如果参数的确比较多，而且输入的参数相互之间的关系比较紧密，不妨把这些参数定义成一个结构，然后把结构的指针当成参数输入。

〖建议 7-3〗函数(方法)体的规模不能太大，尽量控制在 200 行代码之内。

说明：冗长的函数不利于调试，可读性差。

8. 头文件

【规则 8-1】如果不是确实需要，应该尽量避免头文件包含其它的头文件。

说明：头文件中应避免包含其它不相关的头文件，一次头文件包含就相当于一次代码拷贝。

【规则 8-2】申明成员类，应该引用该类申明，而不是包含该类的头文件。

说明：正例：

```
@class SubClassName;

@interface ClassName : NSObject
{
    SubClassName *_pSubClassName;
}
```

反例：

```
#import “SubClassName.h”;
@interface ClassName : NSObject
{
    SubClassName *_pSubClassName;
}
```

9. 可靠性

为保证代码的可靠性，编程时请遵循如下基本原则，优先级递减：

- 正确性，指程序要实现设计要求的功能。
- 稳定性、安全性，指程序稳定、可靠、安全。
- 可测试性，指程序要方便测试。
- 规范/可读性，指程序书写风格、命名规则等要符合规范。
- 全局效率，指软件系统的整体效率。
- 局部效率，指某个模块/子模块/函数的本身效率。
- 个人表达方式/个人方便性，指个人编程习惯。

9.1. 类

【规则 9-1-1】在编写派生类的赋值时，注意不要忘记对基类的成员变量重新赋值。

说明：除非在派生类中调用基类的赋值函数，否则基类变量不会自动被赋值。

正例：

```
- (void) viewDidLoad {
    [super viewDidLoad];
}
```

【规则 9-1-2】私有方法应该在实现文件中申明。

正例：

```
@interface ClassName(Private)

- (void) test;

@end

- (void) test{
}
```

10.其它补充

【规则 10-1】避免过多直接使用立即数。

正例：

```
ViewBounds.size.height = VIEW_BOUNDS_HEIGHT;
```

反例：

```
ViewBounds.size.height = 150;
Height = 150;
```

【规则 10-2】枚举第一个成员要赋初始值。

正例：

```
typedef enum
{
    WIN_SIZE_NORMAL = 0,
    WIN_SIZE_SMALL
}WinSize;
```

反例：

```
typedef enum
{
    WIN_SIZE_NORMAL,
    WIN_SIZE_SMALL
```

```
}WinSize;
```

【规则 10-3】addObject 之前要非空判断。

【规则 10-4】release 版本代码去掉 NSLog 打印，除了保留异常分支的 NSLog。

【规则 10-5】禁止在代码中直接写死字符串资源，必须要用字符串 ID 替代。

说明：应该要考虑多语言国际化，尽量使用 `NSLocalizedStringFromTable` 实现对字符串 `ID` 的引用

【规则 10-6】对于框架设计，逻辑层应尽量与 UI 层分离，降低耦合度。

11.参考文档

《基于 BREW 平台的软件编程规范 20040518》
《P650A1 的编码规范补充》
《C_Cpp 编程规范》
《网络相关编码规范资料》