深圳大铁
SHENZHENDATIE

文件编号：

深圳大铁硬件研发部
部门编码规范

拟制人_____

审核人_____

批准人_____

深圳大铁 SHENZHENDATIE

文档修订记录

版本编号或者更改记录编号	变化状态	简要说明（变更内容和变更范围）	日期	变更人	批准日期	批准人
V0.90	C	初次创建	2015-09-12	张成宇		

*变化状态：C－－创建，A——增加，M——修改，D——删除

1. 引言

1.1 编写目的

本规范说明书是深圳大铁硬件研发部对部门内编码规范的统一。

1.2 背景

软件开发是一个工业化生产过程。在这一过程中，具备严谨性、协作性、以及发展性三个重要特征。严谨性表现为，一个软件通常是复杂逻辑的集合体，需要严格控制其中的逻辑单元才能保证软件产品在各种条件下的功能正常。协作性表现为，一个软件的开发往往是由多人合作完成的、软件的维护也可能由不同的开发人员进行，并且不同项目组可能重用其他项目的工作输出，因而带来项目组间的协作。发展性表现为，一个软件并非在其首个版本编码测试完成就完成了其所有开发工作，在其生存期内，往往需要有其他开发人员介入进行问题修订、功能补充等二次开发和维护工作。

软件的开发过程中，其编码阶段，相对于分析、设计阶段，是由人工主导完成的一个机

械化阶段。那么，由于从业者的技能、素质等因素存在差异，通过自然方式来保障软件开发的严谨性、协作性、和发展性是不现实的。因此，定义一个唯一的工业实施准则，约束编码工作的所有操作过程，按照一个唯一的方式进行，确保消除人为因素带来的影响。

需要明确的是，一个统一的行为准则，其目的并不仅仅保证可靠的代码输出。编码规范的目的在于：

- 尽量避免可能发生的错误；
- 保持输出的一致性，降低协作过程在编码差异上带来的工作；
- 保持输出的可读性，提高产品的可维护性；
- 定义作为内部质量检查的一个可量化标准；

同时，编码规范对编码行为的约束并不是扼杀创造性。就如同工厂作业一样，可以通过创造性优化流程、改良生产设备、改进工艺，但是禁止修改在生产线上操作方法。因此，软件开发人员可以将创造性发挥在改进设计、优化效率、分析处理问题等方面。

本规范试用于本部门采用C类语言以及汇编语言进行软件开发的工作人员。同时，向本部门项目输出代码中提供各种代码单元的非软件开发人员，其提供的代码也应该遵守本规范。

本文中所提级C类语言，包括C、C++、JAVA、C#等语言。然而部门目前的主体工作以C语言和汇编语言为主。所以，本文对于对象编码中，诸如对象定义、包定义等规定暂不做深入定义。待后续将面向对象语言纳为主体时，逐步补充其相关细则。另外，由于硬件平台的差异、汇编器的差异，在条件限制下，汇编语言编码可以适当放宽编码细则的约束。但是，一旦明确硬件平台和汇编器对于本规范的支持，则必须按照本规范进行编码。

本说明书的预期读者包括：

研发部门总负责人、软件部门管理人员、硬件部门固件代码编写人员。

2. 规范细则

2.1 排版规则

2.1.1 缩进

[1] 缩进的 1 个单位宽度需设定为等效的 4 个英文半角字符的宽度。

[2] 对齐只使用空格键，不建议使用 TAB 键进行对齐。实在为了方便使用的话要在当前编

辑器下把 **TAB 设为 4 个宽度，且以空格代替**。因为不同编辑器对 TAB 的宽度设定有差异，会造成布局不整齐。

[3] 方法体、数据类型体、对象体都应该与主体保持 1 个单位的缩进，这些语句包含 if else / switch case / for / do / while / struct / enum / class / function。缩进的示例如下：

推荐的例子

```
if(condition)
{
    //保持 1 个单位缩进开始编码
}
else
{
    //保持 1 个单位缩进开始编码
}
switch(var)
{
case value1:    // case 不需要缩进
    // case 内的语句相对于 case 保持 1 个单位缩进开始编码
    break;    // break  至少要与 case 保持 1 个单位缩进
default:        // default 不需要缩进
    // case 内的语句相对于 case 保持 1 个单位缩进开始编码
    break;
}
for(sentence1; sentence2; sentence3)
{
    //保持 1 个单位缩进开始编码
}
while(condition)
{
    //保持 1 个单位缩进开始编码
```

```
}
do
{
    //保持 1 个单位缩进开始编码
} while(condition) ；
struct  name
{
    //保持 1 个单位缩进开始编码
};
enum
{
    //保持 1 个单位缩进开始编码
};
class name
{
    //保持 1 个单位缩进开始编码
}
int FunctionName(void)
{
    //保持 1 个单位缩进开始编码
}
```

[4] 同一程序中，所有同级程序块的缩进宽度必须保持一致，例子如下：

推荐的例子	不推荐的例子

int FuncName(void)
{
    int iRet;
    iRet = GetSysTickCount();
	int FuncName(void)
	{
	  int iRet;
	      iRet = GetSysTickCount();

return iRet; }	return iRet; }

[5] 语句体的{}不参与缩进，例子如下：

推荐的例子	不推荐的例子

if(condition)
{
Sentence（）；
}
	if(condition)
	{
	Sentence（）；
	}

[6] 在一个函数中，除函数体本身的缩进外，当缩进超过 5 级的时候，应当考虑将函数继续拆分成子函数，以避免缩进过深。

2.1.2 对其和换行

[1] 语句体的的{}必须各占一行，并且成对对齐，例子如下：

推荐的例子	不推荐的例子

if(condition)
{
Sentence（）；
}
	if(condition)
	{
	Sentence（）；
	}
	if(condition) {
	Sentence（）；
	}

[2] 对于一个逻辑程序块的实现注释，需要与该功能块以列对齐书写，例如：

推荐的例子	不推荐的例子

if(condition)
{
//语句 1
Sentence（）；
}
	if(condition)
	{
	//语句 1
	Sentence（）；
	}

[3] 一个代码行一般不超过 80 个字符，当超过 80 个字符的时候，应当进行换行处理。

[4] 代码的换行必须在运算符处断裂，不能在操作数处断裂，例如：

推荐的例子	不推荐的例子
Var = function1() + function2() + function3();	iVar = function1() + function2() + function3();

[5] 长语句换行后必须与同级运算对齐，多次换行应该按运算优先级对齐，对齐同样用 TAB 定位，例如：

推荐的例子	不推荐的例子
Var = (condition1 || condition2 || condition3) ? (value1) : (value2);	Var = (condition1 || condition2 || condition3) ? (value1) : (value2);

[6] 短语句必须独占一行，不能写在同一行内，例如：

推荐的例子	不推荐的例子
Var1 = Var2; Var2 = Var3; Var3 = Var4;	Var1 = Var2; Var2 = Var3; Var3= Var4;

[7] for 语句的换行优先考虑在分号处换行。

[8] 空循环的分号应当另起一行并缩进处理，禁止直接跟在循环语句行尾，例如：

推荐的例子	不推荐的例子

for(sentence1; sentence2; sentence3)
{
;
}
	for(sentence1; sentence2; sentence3) ;

2.1.3 分隔

[1] 数据结构、类、方法、函数、独立程序段、变量声明后，应该加空行进行分隔。

[2] 操作符（运算符）前后应当加空格，使操作符和操作数分离开来，例如：

推荐的例子	不推荐的例子
Var1 = Var2 + Var3 + Var4;	Var1=Var2+Var3+Var4;

[3] 数组下标运算符“[]”、域运算符号“.”和“->”、以及指针运算符号“*”和“&”不加空格分隔。

[4] for 语句中的子语句应该加空格进行分隔，空格加在上一个语句的分号和下一个语句之间，例如：

推荐的例子	不推荐的例子
for(sentence1; sentence2; sentence3)	for(sentence1;sentence2;sentence3)

[5] 单行宏定义中，宏的实现体与宏名称之间用 TAB 进行分隔；一个文件中，尽量做到宏的实现体都从同一列开始对齐。

2.2 命名规则

2.2.1 命名方法

[1] 命名采用英语单词、英语单词词组（单词不超过 4 个）组成。

[2] 长命名使用取元音字符方式缩短命名长度。

[3] 命名中使用单词首字母大写方式区分词组中的不同单词，例如：

推荐的例子	不推荐的例子
int iCaretX, iCaretY;	int icarety, icarety; int i_caret_y, i_caret_y;

[4] 禁止在全局变量中使用无可读意义的字母和数字进行命名，局部变量推荐用有意义的字母和数字组合命名，允许但是不推荐使用数字后缀命名，例如：

应该避免的命名例子	不推荐的命名例子

int I, j, k;
char c1, c2, c3;
int func1();
int func2();
	int iLoop1, iLoop2;
	char *lpszName1, *lpszName2;

[5] 变量采用匈牙利命名法，需要体现变量的类型和用途信息，类型定义在前，用途定义在后；尤其注意，参与循环决策的变量必须用匈牙利命名正确标注其类型，例如：

推荐的例子	不推荐的例子
int iLoop; char cKey; char *lpszInputString;	int loop; char key; char *lpsz; uint CodeU;

[6] 静态变量和全局变量的命名应当在类型说明前说明其作用域。

[7] 禁止局部变量与全局变量同名，禁止静态变量重名，禁止模块之间的全局标号重名。

[8] 函数的命名采用动宾短语组成，如果函数有明确的模块和对象从属，应该在动词前加模块对象修饰；总体格式为“<模块/对象>[动词][宾语]<副词>”，例如：

推荐的例子
Int SetLightOn(); UINT MmuGetIdlePhyAddress(); UINT MmuReleaseMemoryAbs();

[9] 函数命名中用大小写区分单词，不使用连字符区分单词，例如：

推荐的例子	不推荐的例子
Int SetLightOn();	Int set_light_on();

[10] 非函数重命名方式的宏采用全大写命名；函数重命名用途的、或者封装成函数形式的宏采用与函数一致的命名形式，例如：

推荐的例子
#define MAXUINT ((UINT)-1) #define LightOn() (SetLightOn())

[11] 当需要实现一个过程体的多语句宏时，推荐用 do{}while(0)进行封装；【注意】过程体是指不带返回参数的子程序，类似于 void func()形式的函数，如果宏需要对调用者提供返回值，则不适用此规则，例如：

推荐的例子
#define InitGPIO() \ do \ { \

```
    (*GPIOA_DIR) = 0x55AA;     \
    (*GPIOA_DATA)    = 0x55AA; \
} while(0);
```

[12] 错误编码、状态编码等数值常量采用“<分类名称>_<助记提示>”的方式进行命名，例如：

推荐的例子

#define SYSSTATUS_BUSY (TRUE)
#define SYSSTATUS_IDLE (FALSE)
Enum
{
FILEERROR_BADMEMORY = 0,
FILEERROR_BADNAME,
FILEERROR_BADRIGHT
};

[13] 数据结构类型定义和枚举类型定义应当在类型名前加定义体类型声明，例如：

推荐的例子	不推荐的例子
Typedef struct _tagPoint { …. } CPoint, *PPoint;	Typedef struct point { …. } Point;

[14] 所有的同类被命名对象应该遵守一致的命名规则。

[15] 调试代码的宏开关统一用“__DEBUG__开关名称__”的方式来命名。

2.2.2 常用命名前缀

前缀	常见的对应类型	意义
uc	unsigned char	无符号字节数据
u8	unsigned char	无符号 8 位整形数据
c	char	字符，有符号字节数据
s8	signed char	有符号 8 位整形数据
w	unsigned short	无符号双字节数据，字数据

u16	unsigned short	无符号 16 位整形数据
s16	signed short	有符号 16 位整形数据
dw	unsigned long	无符号双字数据，四字节数据
u32	unsigned long	无符号 32 位整形数据
l	long	有符号长整形，双字数据
s32	signed long	有符号 32 位整形数据
qw	unsigned long long	无符号四字数据
u64	unsigned long long	无符号 64 位整形数据
i	int	有符号整形数据（原子宽度）
u	unsigned int	无符号整形数据（原子宽度）
b	BOOL	布尔类型数据
z		数组数据，但是该前缀前须配一个类型前缀，例如：iz 整形数组、byz 字节数组
sz	char []	字符串
lp 或者 p		指针数据，该数据后一般配一个类型前缀，例如：lpsz 字符串指针，lpi 整形指针
lpfn		函数指针
_tag		类型标号，但非 typedef 的类型名
C 或者 T		数据结构类型声明
P 或者 LP		数据结构指针类型声明
t 或者 v		数据结构的数据
h	HANDLE、HWND	句柄
g		全局数据
s		静态数据
cst		常量
f	float	单精度浮点数据
d	double	双精度浮点数据
fp	FILE *	文件指针
m_		类或者数据结构的成员

之所以此处对于位宽长度不同的整形数据有两套前缀，是因为指定位宽的前缀更倾向于硬件相关定义，未指定位宽的前缀更倾向于软件相关定义。这里涉及到纯软件代码的移植扩展性和编译器相关限制。具体的原因见下文描述。

2.2.3 常用命名单词/缩写

单词/缩写	意义	单词/缩写	意义
Cnt	数量	Pos	位置
Ret	返回值	Index	编号、索引
Name	名称	Admin	管理员
Set	设置、集合	Get	获取

Pre	预先的	After	置后的
Tab	表格	List	列表
Array	数组	Map	映射、映射图
Bitmap	位图	Bmp	位图
Dev	设备	Drv	驱动
Item	项	Node	节点
Ctrl	控制	Addr	地址
Value	值	Sum	求和
Total	总计	Per	每一个
Target	目标	Max	最大
Min	最小	Loop	循环
Buf	缓冲区	Ptr	指针
Prj	工程	App	应用
Width	宽度	Height	高度
Ver	版本	Status	状态
Src	源	Dest	目标
Temp	临时的	Len	长度
Obj	对象	Msg	消息
Send	发送	Recv	接收
Read	读	Write	写
Request	请求	Release	释放
Response	应答	Info	信息
Res	资源	Req	请求
Caption	标题	Acquire	获取
Clr	清除	Str	字符串
Ref	引用	Text	文本
Calc	计算	Fill	填充

2.2.4 统一基础数据类型

从移植性角度出发，任何一个软件平台应该具备以下表格所示基础数据类型。无特殊情况下，应用代码禁止自定义基础数据类型。

名称	说明
U8, P_U8	无符号 8 位整形数据和指针
S8, P_S8	有符号 8 位整形数据和指针
U16, P_U16	无符号 16 位整形数据和指针
S16, P_S16	有符号 16 位整形数据和指针

U32, P_U32	无符号 32 位整形数据和指针
S32, P_S32	有符号 32 位整形数据和指针
CHAR	字符<有符号>整形数据
BYTE, PBYTE, LPBYTE	字节整形数据和指针
WORD, PWORD, LPWORD	字整形数据和指针
DWORD, PDWORD, LPDWORD	双字整形数据和指针
INT, PINT, LPINT	有符号整形数据和指针
UINT, PUINT, LPUINT	无符号整形数据和指针
LPSTR	字符串变量指针
LPCSTR	字符串常量指针
VOID, PVOID, LPVOID	无符号数据和指针
LONG, PLONG, LPLONG	有符号长整形数据和指针
ULONG, PULONG, LPULONG	无符号长整形数据和指针
BOOL, PBOOL, LPBOOL	布尔数据和指针
INT8, PINT8, LPINT8	有符号 8 位整形数据和指针
INT16, PINT16, LPINT16	有符号 16 位整形数据和指针
INT32, PINT32, LPINT32	有符号 32 位整形数据和指针
UINT8, PUINT8, LPUINT8	无符号 8 位整形数据和指针
UINT16, PUINT16, LPUINT16	无符号 16 位整形数据和指针
UINT32, PUINT32, LPUINT32	无符号 32 位整形数据和指针

以上对于位宽数据存在两种类型定义，名称中用数字明确指出位宽的数据类型应该应用于硬件相关代码中，而纯软件代码更倾向于使用名称中没有数字标注的数据类型。这出要出于两个方面的考虑：

[1] S8 之类的定义是明确有符号的，正确定义时应该带 signed 前缀，以确保在任何编译配置下的符号位有效。因此 signed char 和 char 对于编译器属于两种类型。此时，如果所有用 char 的数据都用 S8 替换，则执行 str 类函数将会报错。按照最严格编译原则，此错误会无法去除。所以，从硬件相关角度讲，明确位宽和符号的应该使用前一类类型定义，对于纯软件的代码应该避免位宽和符号对软件逻辑的影响，采用后者做定义；

[2] 对于部分纯软件的代码，像一些返回错误代码、指定长度等数据，通常是对类型敏感，但对宽度不敏感，如果强制使用带宽度限制的类型，则会造成在不同位宽 CPU 间移植时，纯软件代码对于 CP U 环境的性能扩展性受到损失。所以，对于纯软件代码建议使用位宽不敏感的类型定义。

这样的情况例如：一个函数需要输入一个 Len 长度参数，但是对齐位宽的要求并不敏感；如果在 16 位环境中用 S16 定义其类型，则移植到 32 位环境中会损失其高位，如果在 32 位环境中用 S32 定义其类型，则移植到 16 位环境中会增加调用消耗；所以，可以直接用 INT 定义该参数的类型，因为 INT 在大多数环境中都是与调用消耗的基础单位一致的宽度。

2.3 注释规则

2.3.1 文档注释

[1] 文件起始第 1 行应该定义文件的文档注释，其注释格式如下：

```
/*****************************************************************
* Module: <模块名>
* Function: <功能描述>
* Description: <使用注意事项等描述>
* Log: <编辑人> <编辑日期><编辑内容>
*                 ……
*****************************************************************/
```

以上注释格式中，在采用版本管理的代码里，Log 行可以忽略。

[2] 如果一个C文件唯一配套一个H文件，则两个文件的文档注释可以统一写在C文件中；

[3] 每个函数的实现体前，必须编写函数的文档注释，注释的格式如下：

```
/*****************************************************************
* Procedure: <函数名称>
* Function: <函数功能>
* Parameter: <参数描述>
* Result: <返回值描述>
* Description: <函数注意事项描述>
*****************************************************************
* LOG    DATE          AUTHOR                    ACTION
*****************************************************************
* <更新日期>          <更改人员>                <更新原因动作>
*****************************************************************/
```

以上注释格式中，在采用版本管理的代码里，Log 行可以忽略。

[4] 对于在一个系统设计中约定有统一函数原型和用途的函数，可以将其文档注释简化为

“函数功能描述”一项内容。

2.3.2 实现注释

[1] 文档注释统一使用/**/的块注释，实现注释统一使用//的行注释方式；

[2] 数据结构的定义前应该加注释，说明数据结构的中文名称和用途；

[3] 数据结构或者对象的每一个成员均需要加注释，注释可以加在每一个成员的声明上一行或者加在同一行的右侧，但是同一份代码必须采用唯一的风格，并且禁止加在声明的下一行；例如：

推荐的例子	不推荐的例子

struct _tagDataType
{
// 长度
int m_iLen;
// 宽度
int m_iWidth;
};
 	
struct _tagDataType
{
int m_iLen; // 长度
int m_iWidth; // 宽度
};
	struct _tagDataType
	{
	// 长度
	int m_iLen;
	int m_iWidth; // 宽度
	};
	
	struct _tagDataType
	{
	int m_iLen;
	// 长度
	int m_iWidth;
	// 宽度
	};

[4] 当一个代码段超过 40 行的时候，必须为该段代码编写描述其逻辑的实现注释；

[5] 当代码段中出现 3 层以上（含 3 层）的分支嵌套语句的时候，必须为每个分支编写实现注释；

[6] 分支体的注释加在分支体开始的第一行，并且与分支体语句对齐；对于仅有一个分支的分支语句，可以将注释加在分支语句的上一行；例如：

推荐的例子	不推荐的例子

if(condition)
{
// 条件成立的注释
	if(condition)
	{// 条件成立的注释
	……

……
}
else
{
// 条件不成立的注释
……
}
 	
// 当条件成立时的注释
if(condition)
{
……
}
 	
switch(var)
{
case value1:
// 条件 1 的注释
break;
 	
default:
// 条件 2 的注释
break;
}
	}
	else
	{// 条件不成立的注释
	……
	}
	
	// 条件的注释
	if(condition)
	{
	……
	}
	else
	{
	……
	}
	switch(var)
	{
	case value1:
	// 条件 1 的注释
	break;
	
	// 条件 2 的注释
	default:
	break;
	}

[7] 每个循环体的上一行需要加注释，说明其循环的逻辑和功能；

[8] 宏的注释编写在宏定义的上一行，不要写在宏定义行的右侧和下一行,例如；

推荐的例子	不推荐的例子
// 布尔真值	#define TRUE (1 == 1) // 布尔真值

#define TRUE　　(1 == 1)	

[9] 为编码注意事项定义一致的开头，常用的注意事项注释开头如下：

注释的开头	意义
TODO:	暂时未完成功能的代码，提示应当补充什么功能
TIP:	变动的提示
NOTE:	值得注意的地方
DEBUG:	调试说明

2.4 子程序规则

[1] 函数应该实现一个独立的功能，避免多个功能由同一个子程序完成；

[2] 函数应该正好完成设计的独立功能，不要过多或缺少设计的功能要求；

[3] 会被函数实现所在文件以外代码调用的函数，必须做输入参数检查；

[4] 有可能执行失败的函数，必须有返回值，明确标注函数的执行情况；

[5] 函数应该明确标注其可见域范围，对于明确不会在实现文件外被调用的函数，应该使用 static 之类的关键字限制其可见性；

[6] 函数如果允许对外被调用，则必须在对应的头文件中进行导出声明；

[7] 调用其他文件的函数时，必须通过包含其文件实现函数定义的导入，不能自行通过 extern 等方式来引入函数定义；

[8] 对于支持多种调用约定的平台，函数实现部分应当明确定义其遵从的调用约定；

[9] 函数的递归实现必须能够明确计算其递归深度，禁止可能出现无限递归的条件；

[10] 函数的形参必须有类型有标识符，不能只有类型；

[11] 函数的的形式定义采用“类型　函数名（形参类型 1 形参 1，…　，形参类型 n 形参 n）”的形式；不采用“类型　函数名（形参表）　形参类型声明”的形式，例如：

推荐的例子	不推荐的例子

int CalcSum(int　iSrc1, int iSrc2, int iSrc3)
{
　　……
}
 	
	int CalcSum(iSrc1, iSrc2, iSrc3)
	int iSrc1, int iSrc2, int iSrc3
	{
	　　……
	}

[12] 函数的有效长度应当控制在 200 行左右；

[13] 控制函数的参数数量，推荐参数数量不超过 4 个，可以接受的最多参数数量为 8 个，更多的参数必须考虑精简；

[14] 推荐函数的扇出数（调用其他函数的次数）在 7 个以下，提高函数的扇入数（被调用数）；

[15] 一个函数应该能够设计独立的与其他未测试函数无关的测试代码，并在输入一定的情况下得到一致结果（随机数发生器除外）；

[16] 函数如果没有参数，那么其原型中应该用 void 定义其参数列表；例如：

推荐的例子	不推荐的例子

int CalcSum(void)
{
……
}
	int CalcSum()
	{
	……
	}

2.5 避免错误的规则

[1] 局部变量不能与全局变量或者其上一可见级别的变量同名；
[2] 无特殊情况时，避免使用函数的形式参数作为函数的工作变量；
[3] 避免使用 goto 语句，尽量用结构化语句替代所有的 goto 逻辑；
[4] 避免使用 abort 和 exit 函数来终止流程；
[5] 函数应该只能有唯一的一个出口，具体即一个函数尽量做到只有一个 return 语句，函数的最后一条语句之外不要出现 return 语句；
[6] “if else”“for”“do”“while”语句的执行体一律使用{}进行封闭；
[7] switch 中必须配置 default 语句；
[8] 无特殊情况时，switch 的内部 case 和 default 分支都需要用 break 做终止；
[9] 除乘除法于加减法的优先级外，其他运算的优先级均使用()来强制定义，即以书写形式保证运算与设计的一致性，不依靠人的记忆来确保运算的优先级符合设计目的；
[10] 尽量在函数开始的语句处定义整个函数中用到的局部变量，避免在函数内部某个分支的{}开始处声明局部变量，以避免出现重复定义的不同作用域的变量，造成原算逻辑错误；
[11] 函数中不要编写用不可能成立的条件执行体；
[12] 在多任务平台中编写函数，或者编写可同时被中断和非中断调用的函数时，应该注意函数的重入性，并且在函数文档注释中说明函数的可重入性；
[13] 可重入函数应当避免使用全局变量或者静态变量；
[14] 避免在代码中直接使用数字常量，使用数字常量的地方一律使用#def ine 定义的替代宏或者对应的枚举值来实现；
[15] 不在模块间直接公用全局变量的标号，即使要公用全局变量，也应该使用函数对变量的访问进行封装；如果目标环境有代码空间限制，也应该用函数形式的宏对变量的访问进行封装；
[16] 全局变量和静态变量必须在模块的初始化函数中进行显示的初始化赋值处理；
[17] 避免将不符合函数原型类型定义的指针数据，做为参数传递给函数；对于可接受任意类型指针的函数参数，应该声明为 void *类型指针；
[18] 对于同函数的一个参数既能接受变量指针又能接受常量指针的情况，此参数必须用 const 进行修饰；
[19] 尽量避免强制类型转换；
[20] 重复的功能应该设计成函数/子程序/宏，避免重复功能代码出现的 BUG 造成的多处修改的问题；
[21] 不能用断言类作为正式输出的异常检查语句；
[22] 必须用调试开关宏来封闭所有需要保留的调测试代码；
[23] 避免使用浮点运算，以回避精确性问题对运算逻辑的干扰；

[24] 条件判断语句中，目标值写在==运算左边，被判断的变量写在==运算的右边；

[25] 所有布尔值均必须用布尔运算来定义，禁止使用 0/1 等数值定义布尔值；例如：

推荐的例子	不推荐的例子
#define TRUE (1 == 1) #define FALSE (1 == 0) BOOL bRet = TRUE;	#define TRUE (1) #define FALSE (0) BOOL bRet = 1;

[26] 取值类的宏其实现体必须用用()封闭，例如：

推荐的例子	不推荐的例子
#define SUM(a,b) ((a) + (b))	#define SUM(a,b) (a) + (b)

[27] 宏的每一个参数在实现体中都应该用()封闭，例如：

推荐的例子	不推荐的例子
#define SUM(a,b) ((a) + (b))	#define SUM(a, b) (a + b)

[28] 避免宏的实现体修改宏的参数值，例如：

应该避免的例子
#define SUM(a, b) ((a ++) + (b ++))

[29] 避免有符号数与无符号数进行比较作为循环条件；

[30] 函数必须对输入参数做合法性检查，不做检查的函数只能是用 static 修饰的，只在一个文件内使用的函数；

[31] 定义变量时注意区分常量数据指针和数据指针常量，例如：const char *与 char * const 的定义效果存在差别；

[32] 头文件需要采用 include guards（即#ifdef #define #endif）进行保护，避免重复定义重复引用等问题；

[33] 枚举的起始值强制用手工定义，不使用编译器的默认起始值；

推荐的例子	不推荐的例子

Enum
{
E_OK = 0,
E_ERROR
};
Enum
{
	Enum
	{
	E_OK,
	E_ERROR
	};

METHOD_READ = 0x02, METHOD_WRITE };	

[34] 不要将 enum 当 define 用，即一般 enum 只用于声明连续整形常数；

应该避免的例子

Enum
{
STATUS_IDLE = 0x01,
STATUS_HOLD = 0x02,
STATUS_KILL = 0x04,
STATUS_SLEEP = 0x08
};

[35] 所有通过动态形式获取到的指针，在其使用前，均需要进行有效性判断；

[36] 对于不定长常量字符串组成的数组，不推荐使用二维数组形式进行定义，例如：

推荐的例子	不推荐的例子

const char * const szzCaption[] =
{
“Hello”,
“Good Morning”,
“Bye”
};
	const char szzCaption[][20] =
	{
	“Hello”,
	“Good Morning”,
	“Bye”
	};

[37] 一般情况下，避免函数返回值的与其原型定义不符；

[38] 避免函数参数列表中出现没有函数功能所不使用的参数；

[39] 避免定义不使用的局部变量，禁止定义不使用的全局变量；

2.6 工程管理规则

[1] 主模块与子模块的目录呈层次关系嵌套在一起；

- 一个模块的目录中用一个 Header 子目录存放当前级别的头文件，但不存放该目录下级目录子模块的头文件；
- 子模块目录的内部结构也与主模块的目录结构一致，包含“Header 目录”“子模块目录”“当前模块代码文件”三个可选组成部分；

[2] 实现同一功能模块的单元代码应当存放在同一目录下，禁止交叉目录存放模块模块文件。

[3] C 文件中的导出函数声明，应当定义在与其同名的头文件中；尽量避免多个 C 文件共用一个头文件；禁止将 C 文件的导出或其他声明定义在其他 C 文件对应的头文件中。

[4] 一个文件可以引用它的同级模块的头文件，可以引用它的下级子模块的头文件，但是禁止引用不直接属于它的下级模块的头文件，例如：

一个模块的目录结构如下：

说明
├──Module1 │　　│　file1.c │　　│ │　　├──Header │　　│　　file1.h │　　│ │　　└──SubMod1 │　　　　├──Header │　　　　│　　SubMod1.h │　　　　│ │　　　　└──SubMod2 │　　　　　　└──Header │　　　　　　　　SubMod2.h │ └──Module2 　　│　file2.c 　　│ 　　├──Header 　　│　　file2.h 　　│ 　　└──SubMod3 　　　　└──Header 　　　　　　SubMod3.h 正确的引用关系有： file1.c 中引用 file1.h、SubMod1.h、file2.h 不正确的引用关系有： file1.c 中引用 SubMod2.h、SubMod3.h file2.c 中引用 SubMod1.h、SubMod2.h

[5] 父模块的导出头文件，应当包含他的子模块的导出头文件，即实现其下级模块可以向其父模块的同级模块发布导出功能。

[6] 当一个头文件 a 已经通过 include 方法被引用到一个上一级的头文件 b 中时，其上级代码文件对于 a 内容的引用，应当优先采用引用 b 的方式来进行，避免直接引用 a 文件。

[7] 一份代码文件中，有效的代码行数应当控制在 2000 行以内。

[8] 在一个不支持运行时二进制复用的工程内，只能采用唯一一种高级语言配合汇编语言来实现；禁止采用超过一种的高级语言实现不支持运行时二进制复用的同一工程。

[9] 整个工程中，禁止出现同名文件。

[10] 文件命名应给用可阅读的单词、缩写、短语组成，避免使用数字索引命名。

[11] 文件扩展名的规范定义如下，禁止自定义扩展名。

扩展名	文件内容
c	c 语言源代码文件
h	c/c++语言头文件（存放导出的数据结构定义、常数定义、API 形式声明等信息）

s 或 asm	汇编语言源代码文件
inc	汇编语言头文件
def	配置定义文件
cfg	配置文件
cpp	c++语言源代码文件
cs	c#语言源代码文件
java	java 语言源代码文件
txt	文本文件（不参与编译）
o 或 obj	编译输出的中间文件（编译生成，参与链接）
a 或 lib	静态库文件（不参与编译，参与链接）

[12] 工程配置文件需要进行版本管理。

[13] 使用第三方库或者是开源代码，必须能够明确其官方来源以及功能描述。

[14] 移植开源必须建立在阅读和理解代码基础上，并在移值时为代码补充文档注释。

[15] 引用到工程中的开源代码必须是有正规组织维护的，应该避免不确定来源或者个人发布的代码使用。

2.7 编译控制规则

[1] 编译中应当开启所有检查参数，以最完整的编译检查辅助控制代码的编码质量。

[2] 编译过程应当视所有警告为错误，即最终输出的软件应该为无警告无错误编译通过的。

[3] 整个工程中使用一致的编译参数，避免为特殊的几个文件单独进行编译参数的配置。如果因为特殊原因，需要为单独的文件设置不同于全局的编译参数，则应该在文档注释的“Description”项后，增加编译配置的说明。增加编译配置说明后，文档注释格式如下：

说明
/*** * Module:<模块名> * Function:<功能描述> * Description:<使用注意事项等描述> * Compile:<编译配置说明> * Log:<编辑人> <编辑日期><编辑内容> * …… ***/

[4] 整个工程必须用最高优化编译发行其黑盒测试版本以及最终的发布版本（如果编译器的优化器有 BUG 可以适当放低标准）。

3. 参考文档

[1]《编码规范细则》 福建星网锐捷升腾资讯有限公司

[2]《华为软件编码规范》

[3]《程序的书写规则》

[4]《C++编码规范与指导》 版本：1.33 作者：白杨

[5]《百度 C++编程规范》