2. C语言类的编程规范

2.1. 文件命名

文件命名的规则按照通俗易懂、简洁明了和统一风格的原则：

2.1.1. 文件名大小写，

文件名统一定为小写方式。

2.1.2. 文件名前缀

文件名按照软件分层架构层次原则进行增加前缀标识，从文件名可以看出本文件处于软件架构中的层次，如APP层定义前缀名为demo\_\*\*\*，如demo\_dc\_drv.c；DEMO层定义前缀名为demo\_\*\*\*，如demo\_gsm\_drv.c，PORT层定义前缀名为port\_\*\*.c，如port\_plat.c；

2.1.3. 文件名含义

从文件名就可以看出本文件实现的内容，如demo\_120r\_send.c；

2.1.4. 文件名对齐

同一类驱动或者同一类功能，命名的关键要排列整齐，如demo\_120r\_com.c、demo\_120r\_send.c、demo\_120r\_recv.c；

2.1.5. 文件名分割

为了比较容易明显看出文件名分割界限，读懂文件名含义，可以在在长文件名中适当加入下划线（“\_”）进行分割，如demo\_jt\_tlink.c；

2.2. 版面排版

2.2.1. 整体排版

整体排版，整体排版的好坏影响到整个代码的可阅读性和易理解性，排版具有一定的顺序和风格，具体排版顺序原则如下：

2.2.1.1. 文件台头注释

文件开头部分为文件的台头注释，台头注释格式见下文详细描述。

2.2.1.2. 头文件包含

文件台头注释后面跟着头文件包含，将本模块要用到的头文件包含进模块，尽量去除那些不需要的头文件包含，如下所示。

#include " includes.h"

#include " dm.h"

#include "tools.h"

2.2.1.3. 宏定义、配置参数

头文件包含后面跟着配置参数、枚举变量等宏定义，如下所示。

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 定义模块配置参数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#define PERIOD\_RECORD SECOND, 1, LOW\_PRECISION\_TIMER

enum {

\_FREE,

\_RECORDING,

\_PLAYING,

\_MAX

};

2.2.1.4. 结构体定义

宏定义配置参数后面跟着模块结构体定义，如下所示：

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 定义模块数据结构

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

typedef struct {

INT8U status;

INT8U folderno\_root; /\* 根目录文件夹 \*/

INT8S folderno\_main; /\* 第一级文件夹编号 \*/

INT8U folderno; /\* 第二级文件夹编号 \*/

void (\*delallinformer)(INT8U result);

} RCBQ\_T;

2.2.1.5. 模块静态变量定义

结构体定义之后跟着模块静态变量定义，如下所示：

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 定义模块变量

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

static RCBQ\_T s\_rcb;

static INT8U s\_scantmr;

2.2.1.6. 函数排列

1) 模块变量之后跟着是模块的局部函数，接着是模块的全局函数，局部函数和全局函数之间用模块初始化函数作为分界线；

2) 模块初始化函数之上紧跟着注册回调处理的函数，注册回调函数之上跟着定时器处理函数；

3) 模块初始化函数之下跟着全局函数；

2.2.2. 缩进对齐

缩进应该是每个程序员都会做的，如果我们部门还有写程序不动缩进的，我们要毫不犹豫的鄙视他，并且要求赔偿读过他程序的人精神损失费。程序块要采用缩进风格编写，使用空格缩进，缩进空格数为4个，禁用TAB键值（除非会设置编辑器）。

针对打下划线的文字说明：对于由开发工具自动生成的代码可以有不一致。如果简单的使用一些开发工具默认的配置来进行自动缩进或者使用纯TAB键来对齐，会导致别人使用其它工具打开时程序版面全部乱了。因此，我们不能使用纯TAB键或自动缩进来进行缩进，一般的编辑器都会提供TAB到空格的替换，如果不能替换，我们就坚决的不能用。

举例：ADS1.2配置、Ultra Edit的配置、SI的配置。

禁止使用BC作为编辑器，因为BC会自动将8个空格识别转换为一个TAB。

在语句上方注释缩进跟代码对齐。

调试宏开关缩进跟代码对齐。

功能宏开关缩进顶格，无需缩进。

2.2.3. 空格

有效的利用空格能给人以赏心悦目的感觉，一堆表达式挤在一起只能让阅读代码的人越来越崩溃。看看下面的代码：

错误示例：

Ha+=(ha\*128+\*key++)%tabPtr->size;

正确示例：

ha += (ha \* 128 + \*key++) % tabPtr->size;

有空格与没空格的区别就很明显了。但是要注意，由于留空格所产生的清晰性是相对的，所以在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在C/C++语言中括号已经是最清晰的标志了。比如：

错误示例：

if ( ( ( i > 10 ) && ( j != 0 ) ) || ( flag == true ) ) {

正确示例：

if ((i > 10 && j != 0) || (flag == true)) {

2.2.3.1. 逗号、分号添加空格

逗号、分号只在后面添加空格，如果分号为该行结束或者跟括号，则无需添加空格，如下所示：

INT8U i, j, k;

for (i = 0; i < 10;) {

i++; // 这么写只是为了表示分号后跟括号无需空格

…

}

2.2.3.2. 操作符添加空格

比较操作符>=、<=、==, 赋值操作符"="、 "+="，算术操作符"+"、"%"，逻辑操作符"&&"、"&"，位域操作符"<<"、"^"等双目操作符的前后加空格。

if (current\_time >= MAX\_TIME\_VALUE)

if (current\_time >= MAX\_TIME\_VALUE && current\_time < 100)

A = b + c;

a \*= 2;

a += 2;

a = b ^ 2;

a = b << 2;

a = b & 2;

2.2.3.3. 操作符不添加空格

"!"、"~"、"++"、"--"、“\*”（指针运算符）、"&"（地址运算符）等单目操作符前后不加空格。

\*p = 'a'; // 内容操作"\*"与内容之间

flag = !isEmpty; // 非操作"!"与内容之间

p = &mem; // 地址操作"&" 与内容之间

i++; // "++","--"与内容之间

2.2.3.4. 访问成员不添加空格

"->"、"."前后不加空格。

p->id = pid; // "->"指针前后不加空格

P.id = pid; // "."成员前后不加空格

2.2.3.5. if、for、while、switch添加空格

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，如下所示：

if (a >= b && c > d) {

;

} else {

;

}

for (i = 0; i < 10; i++) {

;

}

while (1) {

;

}

switch (type)

{

case 0:

break;

case 1:

break;

default:

break;

}

2.2.3.6. 函数名和函数参数不添加空格

函数名和传递参数变量名之间不应使用空格。

result = GetIncResult(&ugroup[i]);

2.2.3.7. 对齐

适当的对齐会增强代码的美观性，对于对齐，我们的基本要求是按照缩进的要求，尽量按4空格对齐，不使用纯TAB键值对齐，对于一些结构体，或者一些有相关性的语句，为了美观，可以进行多空格的对齐。

示例：

student.age = 12;

student.class = 5;

season\_num = SEASON\_SPRING;

season\_temperature = 25;

2.2.4. 换行

2.2.4.1. 条件语句

对于一般的C语言教学书籍中，对于if 、else、for 、do 、while 、case 、switch等语句要求自占一行，无论多少都要加括号{}，且匹配的大括号均需自占一行（BSD风格和GNU风格）。但是对于我们来说，这种写法可能显得过于冗长，使得有效代码长度比例减少。 因此我们会使用K&R风格的括号匹配方式：除switch外，“{”跟在该行语句加一个空格的后面，不必令起一行。对于有else的语句，括号可以在其左右两侧加上。

示例1：错误的写法：

if (uCnt > 0) return 1;

else return 0;

正确的写法：

if (uCnt > 0) {

return 1;

} else {

return 0;

}

示例2：对于switch语句，由于一般的case分支里面可能还有if等其它执行语句，因此我们使用这样的风格来减少缩进的次数：

switch (cmd\_id)

{

case CMD\_A:

break;

case CMD\_B:

if (flag == true) {

return;

}

break;

default:

break;

}

2.2.4.2. 较长语句

较长的语句要分成多行书写，原则上>80 字符或者显示内容接近屏幕宽（普通显示器1024\*768时的宽度，宽屏的1280\*800超过3/4）时，就需要多行书写。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（此时可不按4缩进），使排版整齐，语句可读。

示例：

perm\_count\_msg.head.len = NO7\_TO\_STAT\_PERM\_COUNT\_LEN

+ STAT\_SIZE\_PER\_FRAM \* sizeof( \_UL )

+ STAT\_SIZE\_PER\_FRAM \* sizeof( \_UL );

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

= stat\_poi[index].occupied;

act\_task\_table[taskno].duration\_true\_or\_false

= SYS\_get\_sccp\_statistic\_state( stat\_item );

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

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

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

2.2.4.3. 长表达式

循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，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。一般在屏幕上能显示一行内容可以不用分行。

示例：

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

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

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

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

... // program code

}

for (i = 0, j = 0; (i < BufferKeyword[word\_index].word\_length)

&& (j < NewKeyword.word\_length); i++, j++) {

... // program code

}

for (i = 0, j = 0;

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

i++, j++) {

... // program code

}

2.2.4.4. 多参数

若函数或过程中的参数较长或者参数较多，则要进行适当的划分

示例：

stat\_str\_compare((INT8U \*) & stat\_object,

(INT8U \*) & (act\_task\_table[taskno].stat\_object),

sizeof(\_STAT\_OBJECT));

flash\_act\_duration(stat\_item, frame\_id \*STAT\_TASK\_CHECK\_NUMBER

+ index, stat\_object);

2.2.4.5. 多语句

不允许把多个短语句写在一行中，即一行只写一条语句，变量定义或赋初值也基本按照这个原则，但对于简单的局部变量或者功能类似的变量，可以放在一行书写。

示例1：错误的：

rect.length = 0; rect.width = 0; rect.height = 0;

正确的

rect.length = 0;

rect.width = 0;

rect.height = 0;

示例2：推荐写法：

INT8U i, j, k; // 简单的局部变量

示例3：推荐写法：

static STUD\_STRUCT stud\_1, stud\_2, stud3; // 功能和调用基本一样

2.2.4.6. 变量后空行

相对独立的程序块之间、变量说明之后必须加空行。过多的空行我们是不提倡的，对于函数内容的开头，因紧跟着括号一行下来写。

示例：错误的写法(未空行)：

INT8U repssn\_ind, repssn\_ni;

if (!valid\_ni(ni)) {

... // program code

}

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

推荐写法 ：

INT8U repssn\_ind， repssn\_ni;

if (!valid\_ni(ni)) {

... // program code

}

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni

不提倡的写法（空行过多）：

void DEMO\_150G3\_InitDrv(void)

{

INT8U i;

EXT\_DOWNPARA\_T downpara;

WIRELESS\_DOWN\_T wd\_para;

DEMO\_MEMSET(&s\_tcb, 0, sizeof(s\_tcb));

s\_tcb.curbaud = PE\_BAUD\_9600;

DEMO\_150G3\_InitCom();

DEMO\_150G3\_InitRecv();

DEMO\_150G3\_InitSend();

}

推荐写法：

void DEMO\_150G3\_InitDrv(void)

{

INT8U i;

EXT\_DOWNPARA\_T downpara;

WIRELESS\_DOWN\_T wd\_para;

DEMO\_MEMSET(&s\_tcb, 0, sizeof(s\_tcb));

s\_tcb.curbaud = PE\_BAUD\_9600;

DEMO\_150G3\_InitCom();

DEMO\_150G3\_InitRecv();

DEMO\_150G3\_InitSend();

}

2.2.4.7. 语段间空行

适当的语段间空行会增强代码的美观性，不相关联的语段间适当增加空行分割，增强可阅读性和逻辑性，让人感觉绝赏心悦目。

student.age = 12;

student.class = 5;

season\_num = SEASON\_SPRING;

season\_temperature = 25;

INT8S DEMO\_REC\_GetFileDirByIndex(INT32S index, char \*fullname, INT16U fullnamelen)

{

CELL\_T \*cell;

INT8U i, folderno;

if (!DEMO\_REC\_FileIsIdle()) {

return -1;

}

if (index == -1) {

for (i = 0; i < MAX\_REC\_FOLDER\_NUM; i++) {

if (s\_rcb.audionum[i] < (s\_rcb.maxnum / MAX\_REC\_FOLDER\_NUM)) {

break;

}

}

if (i >= MAX\_REC\_FOLDER\_NUM) {

if (DEMO\_ListItem(&s\_rcb.readylist) > 0) {

cell = (CELL\_T \*)DEMO\_GetListHead(&s\_rcb.readylist);

i = cell->folderno;

} else {

i = 0;

}

}

folderno = i;

} else {

folderno = 0xff;

cell = (CELL\_T \*)DEMO\_GetListHead(&s\_rcb.readylist);

for (;;) {

if (cell == 0) {

break;

}

if (index == cell->index) {

folderno = cell->folderno;

break;

}

cell = (CELL\_T \*)DEMO\_ListNextEle((LISTMEM \*)cell);

}

}

if (folderno == 0xff) {

return -1;

}

OS\_ASSERT((folderno < MAX\_REC\_FOLDER\_NUM), -1);

GetFolderName(folderno, fullname, fullnamelen);

return folderno;

}

2.2.4.8. 文件尾空行

对于一个源文件/头文件的结尾，我们建议保留一个以上的空行，这是为了兼容一些编译器，有些编译器要求源文件必须以空行结尾，否则会引起编译错误。

推荐写法：

void TestFunction(void)

{

…… // program code

}

// 文件末尾空行

2.2.5. 本章小结

排版其实是一个美学工作，通过这个工作，可以改进我们的代码，大家不要忽视。比如：一般来说，一个函数应当控制在一个到两个屏的长度之内，如果你发现经过使用空行排版后，你的函数特别长，那么，不要怀疑规范有误（参照题头），请修改你的函数吧。再比如：如果你发现缩进的层次太多以至于到后面没写几个字就超过屏宽需要换行，那说明你的程序嵌套太多了，那你就该修改你的程序了。

更直接讲，一个版面排版很差的代码，逻辑性就是比较差，代码质量和代码水平也不会高。

2.3. 注释说明

注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码。一般情况下，一个源程序的有效注释量必须在20％以上

说明：注释的原则是有助于对程序的阅读理解，在该加的地方才加，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

总而言之，注释不可少，但也不能滥。清晰明确的函数命名和变量命名，会增加代码的自注释性，提高程序的可读性。

2.3.1. 文件台头注释

2.3.1.1. C/C++语言文件

对于每个C/C++的源程序文件，在每个文件头都应有这样的文件模块注释，文件模块注释主要包括：文件名、版权所有、文件描述、作者、日期时间、修订历史记录等事项。本规范规定文件模块的台头注释格式如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*

\*\* 文件名: demo\_rec\_file.c

\*\* 版权所有: (c) 2007-2008 厦门雅迅网络股份有限公司

\*\* 文件描述: 该模块主要实现录音文件管理

\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 修改历史记录

\*\*========================================================================

\*\*| 日期 | 作者 | 修改记录

\*\*========================================================================

\*\*| 2012/06/21 | 叶德焰 | 创建第一版本

\*\*| 2012/08/21 | 叶德焰 | 修改录音文件扫描排列算法

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

2.3.1.2. 汇编语言文件

对于汇编文件（.src、.a51、.asm、.s等），虽然编译器仅提供单条的注释，但我们也要按照类似C的风格来排版。本规范规定文件模块的台头注释格式如下：

;/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\*\*

;\*\* 文件名: startup.s

;\*\* 版权所有: (c) 2007-2008 厦门雅迅网络股份有限公司

;\*\* 文件描述: 该模块主要实现系统启动引导功能

;\*\*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\*\* 修改历史记录

;\*\*========================================================================

;\*\*| 日期 | 作者 | 修改记录

;\*\*========================================================================

;\*\*| 2012/06/21 | 叶德焰 | 创建第一版本

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

2.3.2. 函数台头注释

对于函数，我们要求每一个函数都需要有台头注释，注释内容主要包括：函数名、函数描述、参数输入、参数输出、返回值等。本规范规定函数台头注释格式如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* 函数名: DEMO\_SM\_ListSend

\*\* 函数描述: 短信发送请求函数，这里传入的data数据指针要求是动态内存申请的，

\*\* 在该层，数据发送成功后会自动释放传过来的动态内存

\*\* 参数: [in] attrib: 发送属性

\*\* [in] tel: 目的手机号码

\*\* [in] tellen: 目的手机号码长度

\*\* [in] data: 待发送的数据内容，传过来的是动态内存

\*\* [in] datalen: 待发送的数据长度

\*\* [in] fp: 发送通知函数

\*\* 返回: 成功返回ture， 传递进来的data动态内存将在本模块自动释放

\*\* 失败返回false，传递进来的data动态内存需要用户自己释放

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOLEAN DEMO\_SM\_ListSend(INT16U attrib, INT8U \*tel, INT8U tellen, INT8U \*data, INT16U datalen, void (\*fp)(INT8U result))

2.3.3. 代码行中注释

2.3.3.1. 注释格式

原则上代码内的注释使用块注释符 /\* ....... \*/进行注释，在注释符两边保留各一个空格，要保持整个文件的注释风格的统一。如下所示：

/\* 注释示例1 \*/

2.3.3.2. 注释语句

注释语句要简洁通俗易懂，具有代表概括性语句，尽量避免那种大篇幅和冗余注释，除非该代码确实难以理解，需要详细描述。

2.3.3.3. 注释有效性

注释的内容要清楚明了，含义准确，防止注释二义性，原则上多用中文注释，除非能流畅的使用英文，不熟悉的英文别用，禁用拼音或错误的英文语法。可以使用缩写，但慎用缩写，尤其是不常见的缩写，TCP/IP这样的就不必用全称或者中文了。

错误示例：

/\* This pointer can’t fu zhi until initialized \*/

// 到底是不能复制还是不能赋值？

/\* RTFSC \*/

// Read The F\*\*king Source Code. 这个还算比较常见的……

2.3.3.4. 注释位置和对齐原则

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，而且与被注释的语句保持同一种缩进风格。放在右方的一些代码注释，应当使用4空格缩进或改进的TAB缩进进行对齐，注释起始列号从80列开始，如果代码的列长度已经超过80列，那么注释紧跟代码后面，或者注释放在代码的上方。

如果是对结构体成员或者变量等进行注释，则注释起始列号从10的整数倍列号开始，但是要保持整齐有序，如下所示：

错误示例1：

void Example(void)

{

/\* get replicate sub system index and net indicator \*/

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

}

错误示例2：

k = 0;

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

/\* get replicate sub system index and net indicator \*/

错误示例3：

typedef struct \_pt\_man\_t\_ {

　　 int numProc; /\* Number of processes \*/

　　int maxProc; /\* Max Number of processes \*/

　　 int numEvnt; /\* Number of events \*/

　　 int maxEvnt; /\* Max Number of events \*/

　　 HANDLE\* pHndEvnt; /\* Array of events \*/

　　 DWORD timeout; /\* Time out interval \*/

　　 HANDLE hPipe; /\* Namedpipe \*/

　　 TCHAR usr[MAXUSR]; /\* User name of the process \*/

　　 Int numMsg; /\* Number of Message \*/

　　 Int Msg[MAXMSG]; /\* Space for process communicate \*/

} PT\_MAN\_T;

推荐写法1：

void Example(void)

{

/\* get replicate sub system index and net indicator \*/

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

}

推荐写法2：

typedef struct \_pt\_man\_t\_ {

　　 int　　 numProc;　　 /\* Number of processes　　　　　　　　 \*/

　　int　　 maxProc;　　 /\* Max Number of processes　　　　　　 \*/

　　 int　　 numEvnt;　　 /\* Number of events　　　　　　　　　　 \*/

　　 int　　 maxEvnt;　　 /\* Max Number of events　　　　　　　　 \*/

　　 HANDLE\* pHndEvnt;　 /\* Array of events　　　　　　　　　　 \*/

　　 DWORD　 timeout;　　 /\* Time out interval　　　　　　　　　 \*/

　　 HANDLE　hPipe;　　　 /\* Namedpipe　　　　　　　　　　　　　 \*/

　　 TCHAR　 usr[MAXUSR];/\* User name of the process　　　　　　\*/

　　 int　　 numMsg;　　 /\* Number of Message　　　　　　　　　 \*/

　　 int　　 Msg[MAXMSG];/\* Space for process communicate \*/

} PT\_MAN\_T;

推荐写法3：

typedef struct {

INT32U status;

INT8U step;

INT8U ct\_reset;

INT8U ct\_reg; /\* 对网络状态进行过滤 \*/

INT8U ct\_gprs; /\* 对GPRS状态进行过滤 \*/

INT32U sim\_status; /\* sim卡状态,见SIM\_STATE\_E \*/

INT32U gsm\_status; /\* gsm网络状态，见NETWORK\_STATE\_E \*/

INT32U gprs\_status; /\* gprs网络状态，见NETWORK\_STATE\_E \*/

INT8U rssi;

INT8U biterror; /\* 位错误率 \*/

INT8U signallevel; /\* 信号强度等级 \*/

INT32U signalvalue; /\* 信号强度值 \*/

void (\*handler[MAX\_HANDLER])(INT8U isreg);

} GCB\_T;

2.3.3.5. 习惯原则

平时注意养成边写代码边注释的习惯，在代码作出修改时确保注释进行相应的修改，以保证其一致性。不再有用的注释要删除。

2.3.3.6. 注释节奏和频度

注释时，一般对函数的关键位置进行注释，如条件的跳转、算法、连续一段时序操作等部位进行注释，对于那些很容易理解的代码，无需注释，尽量避免冗余注释，以确保代码可阅读性。如下所示：

示例1，switch语句条件：

char \*DEMO\_GPS\_GetVersion(void)

{

INT8U gpstype;

gpstype = DEMO\_GPS\_GetType();

switch (gpstype)

{

case GPS\_TYPE\_L10: /\* L10模块 \*/

DEMO\_MEMCPY(s\_ccb.version, sizeof(s\_ccb.version), "L10", sizeof(s\_ccb.version));

break;

case GPS\_TYPE\_GA: /\* GA302/GA802 \*/

DEMO\_MEMCPY(s\_ccb.version,sizeof(s\_ccb.version), "GA302/GA802", sizeof(s\_ccb.version));

break;

case GPS\_TYPE\_TD: /\* 泰斗 \*/

break;

case GPS\_TYPE\_ZKW: /\* 中科微 \*/

break;

default:

DEMO\_MEMCPY(s\_ccb.version, sizeof(s\_ccb.version), "unknow", sizeof(s\_ccb.version));

break;

}

return (char \*)s\_ccb.version;

}

示例2，if语句条件：

void DEMO\_InformGprsRegistStatus(INT32U status)

{

if ((s\_gcb.status & \_SIM\_DETECT) == 0) { /\* 未检测到sim卡 \*/

return;

}

if (status == NETWORK\_STATE\_REGISTERED) { /\* 搜索到gprs网络 \*/

s\_gcb.ct\_gprs = 0;

if ((s\_gcb.status & \_GPRS\_REG) == 0) {

#if DEBUG\_NETWORK > 0

debug\_printf("<搜索到GPRS网络(%d)>\r\n", status);

#endif

s\_gcb.status |= \_GPRS\_REG;

}

} else { /\* 未搜索到gprs网络 \*/

if ((s\_gcb.status & \_GPRS\_REG) != 0) {

if (++s\_gcb.ct\_gprs >= 3) { /\* 连续3次检测无GPRS网络才认为无GPRS信号 \*/

#if DEBUG\_NETWORK > 0

debug\_printf("<未搜索到GPRS网络(%d)>\r\n", status);

#endif

s\_gcb.ct\_gprs = 0;

s\_gcb.status &= (~\_GPRS\_REG);

}

}

}

}

2.3.3.7. 变量、常量、宏、结构体注释

对于所有有物理含义的变量、常量、宏、结构体，如果其命名不能充分体现其物理含义，在声明时都需要添加注释，以说明其物理含义。注释应放在其相邻上方或者右方，注意按4空格对齐。

示例1：

/\* 空闲任务列表中的最大任务数 \*/

#define MAX\_TASK\_NUMBER 64

示例2

#define MAX\_TASK\_NUMBER 64 /\* 空闲任务列表中的最大任务数 \*/

示例3

/\* GPRS网络状态枚举 \*/

enum GPRS\_NET\_STATE {

G\_NET\_IDLE, /\* 网络空闲 \*/

G\_NET\_CONNECTING, /\* 网络正在连接 \*/

G\_NET\_CONNECTED, /\* 网络已连接成功 \*/

};

2.3.3.8. 无效注释

禁止使用无效的注释。另外，如果变量名、函数名、结构名已足够清晰，满足自注释的要求，那么就无需再增加注释显得重复。

错误的示例：

memset(&procInfo, 0, sizeof(procInfo)); // rocky add 070520

//rocky 是谁？干嘛滴？这样的注释在临时使用还行，归档正式发布时一定要去除

/\* 如果cflag为真 \*/ // 这种属于无效注释、重复注释。下面的注释就有效

if (cflag == true) { /\*如果链路层收到信息条件为真，则执行下面的代码 \*/

2.3.3.9. 结尾注释

对于程序结束、条件宏的结束以及函数的结束，可以增加注释来表示一个结束。在这样的注释不算无效，特别在条件宏较多、逻辑嵌套较多的环境下使用是很有用的。

示例：

if (aaa) {

#if USE\_PPP > 0

aaa = 0;

#if LOOP\_TEST > 0

while (a < 10) {

……

}

#endif /\* end of #if LOOP\_TEST > 0 \*/

#endif /\* end of #if USE\_PPP > 0 \*/

} else if (bbb) {

……

}

2.3.3.10. 插入注释

除非必要，不应在一行代码或者表达式中间插入注释或者屏蔽什么，否则会降低代码的可理解性。

2.3.3.11. 无用代码

有不用的代码，在归档或者正式发布时最好去掉。临时的屏蔽不应使用块注释符进行屏蔽，否则如果该代码在内部有用到块注释符做注释时会出现匹配问题。建议使用#if 0这样的宏进行屏蔽。

2.4. 标识符命名

本来打算先暂不制定命名规则，先搜集大家的命名使用习惯，然后整理出一套命名规则，但是想想既然我们这是讨论版，就不先搜集了，先定下一个基准，由大家来提意见建议。

据考察，没有一种命名规则可以让所有的程序员赞同，程序设计教科书一般都不指定命名规则。命名规则对软件产品而言并不是成败攸关的事，我们不要化太多精力试图发明世界上最好的命名规则，而应当制定一种令大多数项目成员满意的命名规则，并在项目中贯彻实施。这里主要讲的是适用于C语言的命名规则，汇编可以参考命名。

C语言是一种简洁的语言，标识符命名不应当过长，并且我们为了确保程序的跨平台的可移植性，使用的是自定义的数据类型，因此微软的那套繁琐的匈牙利命名规则我们是绝不采用的。我们不要求一套死死的命名规则，只求能提高代码可读性。以下列出几点命名的大致规则，用来指导标识符的命名。

C语言是类英语的编程语言，标识符的命名应当采用直观的英文单词或者可拼读的单词组合，组合一般以“名词”和“名词＋形容词”的方式进行组合，切忌使用汉语拼音命名。英语用词要准确，不建议用复杂生僻的单词。

示例：

表示当前值可用CurrentValue，NowValue或者DQValue则错误。

window下的程序的标识符一般以“大小写混排”的方式，如“MaxValue”，而Unix下的程序的标识符一般以“纯小写加下划线”的方式，如“max\_value”，我们目前结合一下彼此的长处，给出如下定义供大家参考：

如果不好实施，可以尽量与开发的平台使用的软件风格保持一致，如在window下可以使用大小写混排的风格，在Unix类下可以使用小写加下划线的风格，但一定要统一。

标识符命名不宜过长（早期的ANSI C还要求名称不得超过6个字符，现在没这个限制了）。过长的名称显得繁琐而且使得可读性变差了。使用完整的英文单词或大家可以基本理解的单词缩写进行命名。适当的缩写可以减小程序容量，也提高了可读性，使得程序短小而迷人。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；一些公认的谐音的替换写法可以允许使用

示例：如下单词的缩写能够被大家基本认可。

statistic 可缩写为 stat;

increment 可缩写为 inc;

message 可缩写为 msg;

Operating System可缩写为OS;

Minium Value可缩写为MinVal;

HexToBin可以缩写为Hex2Bin

允许有个人特点的命名风格，但风格要统一，不要来回变化。

说明：个人的命名风格，在符合所在项目组或产品组的命名规则的前提下，才可使用，最好在源文件开头增加注释进行说明。（即命名规则中没有规定到的地方才可有个人命名风格）。

命名中如果有特殊的约定或者缩写，需要有注释说明。说明：应该在源文件的开始之处，对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

对于全局变量命名，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，使用时须谨慎。

如非必要，尽量不要使用单个数字或者非标准ASCII字符作为标识符；不要使用名称一样，但大小写不同的标识符，或者仅靠大小写来区分的标识符，有的编译器是不会区分大小写的，也很容易造成歧义；有些字符尽量不要作为一个短语使用在命名规则中，如“1”和“l”，“0”和“o”。

以下的命名容易让人迷惑：

void Func\_1(void);

void func\_1(void);

void Func\_l(void); // 这三者容易混淆

INT8U x, X; // 这样的定义容易混淆

void MyFunc56(void);// 函数名没有意义

INT8U value\_『6』; // 使用非标准的字符

/\* 作者：︻┳═一 \*/ //\*&^%$#……这样的同志太有才了，放在雅迅委屈了

在同一软件产品内，当多人协作按模块划分时，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。

说明：对接口部分的标识符应该有更严格限制，防止冲突。如可规定接口部分的变量与常量之前加上“模块”标识等。如LCD的可以增加lcd\_前缀。这个取决于项目组内部协商。

用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。注意不要与系统的关键字冲突。

说明：下面是一些在软件中常用的反义词组。

add / remove begin / end create / destroy

insert / delete first / last get / release

increment / decrement put / get add / delete

lock / unlock open / close min / max

old / new start / stop next / previous

source / target show / hide send / receive

source / destination cut / paste up / down

示例：

INT8U min\_sum;

INT8U max\_sum;

INT8U add\_user(INT8U \*user\_name);

INT8U delete\_user(INT8U \*user\_name);

除了编译开关/头文件、内嵌汇编等特殊应用，应避免使用\_EXAMPLE\_TEST\_ 之类以下划线开始和结尾的定义，这样很容易与编译器编译生成的标识符名称相同，造成不必要的错误。

本章小结：说了那么多，无非就两点：整个程序的标识符命名风格统一，标识符表达含义清晰明确，能够有自注释性。能够真正满足这两点，就可以有足够理由自恋了。

2.5. 可读性

能够满足上面四点的要求，代码的可读性已经有很大增强了，但是还是有些地方需要我们去加强，去完善的。

2.5.1. 运算符增加括号

注意运算符的优先级，适当的增加括号明确表达式的操作顺序，避免由于使用默认优先级导致程序运行时产生逻辑上的错误。

示例：如下三个表达式

word = (high << 8) | low (1)

if ((a | b) && (a & c)) (2)

if ((a | b) < (c & d)) (3)

如果书写为：

Word = high << 8 | low

if (a | b && a & c)

if (a | b < c & d)

1和2不会出错，但可读性变得很差了；3则变成了if (a |(b < c) & d)了，判断逻辑上就犯错误了。

2.5.2. 宏定义替换具体数值

在常量上，尽量使用有意义的宏来进行替换，以增加程序的可读性。涉及到物理状态或者有物理意义的常量，不能使用数字，而应使用宏或者枚举来进行替换。我们尽量不要使用const定义常量，而是尽量用宏去替代。

举例：

if (g\_gprs\_state == 0) {

g\_gprs\_state = 1;

}

这样的语句没有错，但是毫无可读性而言。如果改成如下形式：

#define GPRS\_STATE\_IDLE 0

#define GPRS\_STATE\_CONNECTING 1

if (g\_gprs\_state == GPRS\_STATE\_IDLE) {

g\_gprs\_state = GPRS\_STATE\_CONNECTING;

}

或者使用枚举：

/\* GPRS状态迁移枚举说明 \*/

enum GPRS\_STATE {

GPRS\_STATE\_IDLE, /\* GPRS空闲 \*/

GPRS\_STATE\_CONNECTING, /\* GPRS连接中 \*/

GPRS\_STATE\_CONNECTED, /\* GPRS已连接 \*/

GPRS\_STATE\_DISCONNECTING, /\* GPRS断链中 \*/

GPRS\_STATE\_MAX

};

2.5.3. 寄存器访问

对于和硬件相关的寄存器等硬件资源，不允许在程序中直接访问，这样会导致程序的混乱。

2.5.4. 连续关联性语句

源程序中关系紧密、相关性较强的代码尽可能的相邻放置，以便于程序阅读和查找。

示例：以下代码布局不太合理。

rect.length = 10;

char\_poi = str;

rect.width = 5;

若按如下形式书写，可能更清晰一些。

rect.length = 10;

rect.width = 5; // 矩形的长与宽关系较密切，放在一起。

char\_poi = str;

2.5.5. 排除复杂难懂语句

如非必要，不要使用复杂难懂的语句，不定义复杂的数据类型。忌用技巧性较高的语句。高技巧的语句在嵌入式环境编译后往往等于低执行效率，一般场合下我们在保证基本效率的同时，用朴素的编程语言实现，对今后的维护工作帮助很大。

示例：以下的算法和变量定义不太合理

MY\_STRUCT test\_val[10][20][30][40]; // 数组定义最好不要超过二维

\*stat\_poi++ += 1;

\*++stat\_poi += 1; // 指针的表达过于复杂，看完要晕

应分别改为如下:

\*stat\_poi += 1;

stat\_poi++; // 此二语句功能相当于“ \* stat\_poi++ += 1; ”

++stat\_poi;

\*stat\_poi += 1; // 此二语句功能相当于“ \* ++stat\_poi += 1; ”

2.5.6. 模块内杂项静态变量统一管理

一般模块内都会定义一些杂项的状态管理等静态变量，比较零散，最好用结构体方式管理这些零散静态变量，同时便于对齐注释，在模块在进行统一初始化，如果后面维护时需要新增静态变量则无需关心是否已初始化。

错误示例：

INT8U s\_status, ct\_query, ct\_baud, verlen, ver[MAX\_VER];

正确示例：

typedef struct {

INT8U status; /\* 150G3状态 \*/

INT8U ct\_query; /\* 链路维护计数器 \*/

INT8U ct\_baud; /\* 波特率改变计数器 \*/

INT8U verlen; /\* 版本长度 \*/

INT8U ver[MAX\_VER]; /\* 版本 \*/

} TCB\_T;

static TCB\_T s\_tcb;

DEMO\_MEMSET(&s\_tcb, 0, sizeof(s\_tcb));

2.6. 变量和结构

2.6.1. 变量命名规则

2.6.1.1. 变量命名大小写

变量命名的规则按照通俗易懂、简洁明了和统一风格的原则，变量命名基本原则为统一小写组合方式。除了一些如.def注册表中自动生成的特殊变量名等除外。

示例：

static ECB\_T s\_lcb;

static INT8U s\_scantmr;

2.6.1.2. 变量名前缀

变量类型划分为：局部变量、静态变量、全局变量、参数变量，按照变量类型的不同，变量定义的前缀名也不同。

局部变量，函数内的定义的变量，无需前缀名，如下所示：

static void DiagnoseProc(void)

{

INT32U count;

count = DEMO\_ListItem(&s\_lcb.usedlist) + DEMO\_ListItem(&s\_lcb.freelist);

if (DEMO\_ListItem(&s\_lcb.usedlist) > 0) {

OS\_ASSERT(DEMO\_TmrIsRun(s\_scantmr), RETURN\_VOID);

}

}

静态变量，模块中的静态定义变量，前缀名为"s\_"，如下所示：

static ECB\_T s\_lcb;

static INT8U s\_scantmr;

全局变量，模块之间可以共用的变量，前缀名为“g\_”，如下所示：

#ifdef GLOBAL\_DEMO\_FS\_CFG

#define DEMO\_FS\_CFG\_EXT

INT32U g\_folder\_para\_memory = MAX\_PARA\_DIR\_MEMORY;

INT32U g\_folder\_wd\_memory = MAX\_WD\_DIR\_MEMORY;

#else

#define DEMO\_FS\_CFG\_EXT extern

extern INT32U g\_folder\_para\_memory;

extern INT32U g\_folder\_wd\_memory;

#endif

参数变量，函数中的的传递变量，无需前缀名，传递的变量名称要具有代表意义，如下所示：

void OS\_PostMsg(INT8U tskid, INT32U msgid, INT32U lpara, INT32U hpara);

2.6.1.3. 变量名含义

从变量名就可以看出本变量含义和作用，如：

typedef struct {

INT8U tskid;

INT32U msgid;

INT32U lpara;

INT32U hpara;

} OS\_MSG\_T;。

2.6.1.4. 变量名对齐

同一类的变量名，命名的关键要排列整齐，如下所示；

extern const char s\_record\_maindir[];

extern const char s\_record\_seconddir[][2];

extern const char s\_dc\_extname[5];

extern const char s\_dc\_firstdir[];

extern const char s\_dc\_seconddir[][3];

2.6.1.5. 变量名分割

为了比较容易明显看出变量名分割界限，读懂变量名含义，可以在在长变量名中适当加入下划线（“\_”）进行分割，如：

extern const char s\_record\_maindir[];

extern const char s\_record\_seconddir[][2];

extern const char s\_dc\_extname[5];

extern const char s\_dc\_firstdir[];

extern const char s\_dc\_seconddir[][3];

2.6.2. 少用全局变量

尽量少用公共变量，去掉没必要的公共变量。禁止有不用的全局变量。由于公共变量是增大模块间耦合度的原因之一，因此减少公共变量的使用能够降低模块间的耦合度。

2.6.3. 变量定义

仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。在对变量声明的同时，应对其含义、作用及取值范围进行注释说明，同时若有必要还应说明与其它变量的关系。

2.6.4. 变量与函数关系（不要求）

明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等，明确过程操作变量的关系后，将有利于程序的进一步优化、单元测试、系统联调以及代码维护等。这种关系的说明可在注释或文档中描述。

示例：在源文件中，可按如下注释形式说明。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 变量名 System\_Init Input\_Rec Print\_Rec Stat\_Score

\* Student 初值/创建 写 读 读

\* Score 初值 写 读 读, 写

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

注：Student、Score为两个全局变量，System\_Init、Input\_Rec、Print\_Rec、Stat\_Score为四个不同的函数。其中，函数Input\_Rec、Stat\_Score都可修改变量Score，故此变量将引起函数间较大的耦合，并可能增加代码测试、维护的难度。

2.6.5. 变量越界

向变量传递数据时，要十分小心，最好不要使用强制类型转换的方式来进行赋值，防止出现赋值不合理或者越界的现象。

在一般的PC机编程，如果有必要，需要对公共变量的赋值进行合法性检查。我们可能会在测试时添加这样的测试性代码来进行检查，平时的编程工作中只要求大家小心使用，毕竟前期小心注意好过后期补救。

示例:下面由于len的类型与调用函数的maxlen类型不一样，导致传递参数时，2字节数据被截成1字节数据，被自动强制转换出现错误。

INT8U DEMO\_GetDriverLicenceCodeAndLen(INT8U \*ptr, INT8U maxlen)

{

;

}

INT8U Test(void)

{

INT16U len;

INT8U tempbuf[20];

len = 2000;

DEMO\_GetDriverLicenceCodeAndLen(tempbuf, len);

}

2.6.6. 变量重名

禁止全局变量与静态变量或者局部变量或者函数的参数同名。

2.6.7. 变量未初始化

严禁使用未经初始化的变量作为右值。右值就是逻辑操作符、赋值操作符、算术操作符、比较操作符等右边的变量。特别是指针，如果未经初始化就使用，会使得系统崩溃。

2.6.8. 变量修改权限

对于公共变量，最好是仅仅一个模块或者函数能够对其进行修改和创建，其余的模块或函数只能读取它。防止多个模块都可以修改、创建同一个公共变量。

2.6.9. 变量与硬件密切相关

尽量不要使用与硬件相关或编译器环境关系密切的变量，如果有这样的变量，尽量归在一个模块中操作，以便于移植。多使用自定义的数据类型，以提高程序可移植性。

示例：下段代码不应放在main中，最好放在专门的硬件操作模块中供调用。

void main()

{

register int index; // 寄存器变量

\_AX = 0x4000; // \_AX是BC3.1提供的寄存器“伪变量”

... // program code

}

2.6.10. 数据类型

使用可移植的数据类型。由于我们部门的硬件平台跨度从8位到32位，如果使用标准的C语言的数据类型，在移植时就会出现较大的问题（int型在32位和8位的长度是不同的）。因此我们会在每个平台的config文件下对这些数据类型进行重新定义，这样就确保只修改这个配置文件，而源文件都无需做调整。

借鉴了UCOS-II的自定义数据类型，一般使用到的数据类型有：

char

INT8U 无符号8位整型

INT16U 无符号16位整型

INT32U 无符号32位整型

INT8S 有符号8位整型

INT16S 有符号16位整型

INT32S 有符号32位整型

BOOLEAN 布尔型

FP32 单精度浮点型

FP64 双精度浮点型 // 尽量少用浮点型，因为一般的MCU均不能支持浮点型，都是编译器用整型模拟的，这样效率很低

我们尽量使用无符号整型数。因为负数在不同的平台下，表达方式不一样，有的是拿最高位作为负号，有的是专门的寄存器表示。

数据类型定义采用全大写的方式，我们定义方式如下：

typedef unsigned char BOOLEAN;

typedef unsigned char INT8U;

typedef signed char INT8S;

typedef unsigned short INT16U;

typedef signed short INT16S;

typedef unsigned int INT32U;

typedef unsigned long long INT64U;

typedef signed int INT32S;

typedef float FP32;

typedef double FP64;

typedef unsigned long ip\_addr;

2.6.11. 变量大小端

另外，嵌入式的系统经常有小端和大端的区别，因此我们推荐使用自定义的数据类型。此外，我们对于需要进行网络字节操作的超过8位的数据，以及需要进行8位到16/32位操作的时候，尽量使用自定义的数据类型来规避大小端对程序移植的影响。

示例1：

网络传输这样的8个字节数据：四个字节的姓名（ASCII码）和4个字节的学号（bin），如果我们使用默认的数据类型或者用INT32U类型来处理，在大端和小端时就会有问题。

static INT32U name = 0;

static INT32U number = 0;

void HdlReceive(INT8U \*ptr, INT8U len)

{

INT8U i;

for (i = 0; i < 4; i++) {

name |= (\*ptr)<<(8\*i);

\*ptr++;

}

for (i = 0; i < 4; i++) {

number |= (\*ptr)<<(8\*i);

\*ptr++;

}

}

这样的代码，在小端和大端的系统下处理起来刚好相反。因此我们建议用联合的方式来定义一种类型进行这类操作的处理

在小端系统下，可以定义：

typedef union {

INT32U longword;

struct {

INT8U low\_1;

INT8U low\_2;

INT8U high\_1;

INT8U high\_2;

} bytes;

} LONG\_UNION;

在大端系统下，可以定义：

typedef union {

INT32U longword;

struct {

INT8U high\_2;

INT8U high\_1;

INT8U low\_2;

INT8U low\_1;

} bytes;

} LONG\_UNION;

这样的话，这么使用，就有移植性了，虽然朴素，但是效率反而高过循环体操作：

static LONG\_UNION name;

static LONG\_UNION number;

void HdlReceive(INT8U \*ptr, INT8U len)

{

name.bytes.high\_2 = \*ptr++;

name.bytes.high\_1 = \*ptr++;

name.bytes.low\_2 = \*ptr++;

name.bytes.low\_1 = \*ptr++;

number.bytes.high\_2 = \*ptr++;

number.bytes.high\_1 = \*ptr++;

number.bytes.low\_2 = \*ptr++;

number.bytes.low\_1 = \*ptr++;

}

2.6.12. 结构体定义

结构在设计上需要考虑充分，结构的元素不宜过多，如果结构需要表示的元素确实很多，可以在结构里面定义子结构，把相似类别的元素分别归放到子结构中。结构中元素的设计很重要，适当的元素顺序可以提高效率，节约占用空间。

结构体可分为变量结构体和枚举结构体，不同的同构提类型定义的名称也不一样，如下所示：

变量结构体，以"\_T"为后缀，并且全大小方式。

typedef struct {

INT8U tskid;

INT32U msgid;

INT32U lpara;

INT32U hpara;

} OS\_MSG\_T;

枚举结构体，以"\_E"为后缀，并且全大小方式。

typedef enum {

RECORD\_CH\_NULL = 0,

RECORD\_CH\_1,

RECORD\_CH\_MAX

} RECORD\_CH\_E;

2.6.13. 强制类型转换

注意强制类型转换的使用。尽量减少没有必要的强制类型转换以及默认数据类型转换。因为在进行转换后，数据的含义、值都有可能受到影响。

char chr;

unsigned short int exam;

chr = -1;

exam = chr; // 编译器不产生报警或者出错，此时exam为0xFFFF。

2.7. 宏

宏是个好东西，在节约空间、提高代码可读性、提高编译灵活性方面给我们带来很多便利，但是我们在使用中同样要注意，小心对宏的使用。我们要少用宏来定义表达式，可以用内联函数来替代（虽然内联函数移植性差点），如果要用宏表达式，要注意以下几点

宏作为条件编译开关时，一定要保持风格的统一。

比如某人对宏编译开关使用的是：定义为0关闭，非零打开，会这么写

#define EN\_DEBUG 0

#if EN\_DEBUG > 0

#endif

如果另外一个人对宏编译开关使用的是：未定义为关闭，定义为打开，会这么写

#define EN\_DEBUG

#ifdef EN\_DEBUG

#endif

如果两个风格混搭在一起，比如这么写：

#define EN\_DEBUG 0

#ifdef EN\_DEBUG

#endif

就起不到控制的作用了。

2.7.1. 宏宽度

使用宏来表达一个值的时候，请注意宏的数据宽度不要超过使用宏的变量的宽度。因为不能确保每个编译器都能够对此进行警告提示。

2.7.2. 宏括号使用

7-3 使用宏表达式时，一定要使用完备的括号，不要怕麻烦。

示例：

#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))

如果宏只是替换某一个数据或者某些参数、而不参加运算时，尽量维持它替换时的原样，不应加括号。

如 #define PERIOD\_TIMEOUT (SECOND, 2)

StartTmr(testtmr, PERIOD\_TIMEOUT); // 在编译时就出错了

7-4 如果要用宏来定义多条表达式，一定要将所有的表达式放在大括号中

示例：下面的表达式中，只有a = 0被执行

#define INTI\_RECT\_VALUE(a, b)\

a = 0;\

b = 0;

for (index = 0; index < RECT\_TOTAL\_NUM; index++)

INTI\_RECT\_VALUE(rect.a, rect.b);

正确的用法应为：

#define INTI\_RECT\_VALUE(a, b)\

{\

a = 0;\

b = 0;\

}

for (index = 0; index < RECT\_TOTAL\_NUM; index++) {

INTI\_RECT\_VALUE(rect[index].a, rect[index].b);

}

7-5 使用宏时，不允许在调用宏时参数发生变化

示例：如下用法可能导致错误。

#define SQUARE(a) ((a) \* (a))

int a = 5;

int b;

b = SQUARE(a++); // 结果：a = 7，即执行了两次增1。

正确的用法是：

b = SQUARE(a);

a++; // 结果：a = 6，即只执行了一次增1。

2.8. 函数和过程

2.8.1. 函数命名规则

2.8.1.1. 函数命名大小写

文件命名的规则按照通俗易懂、简洁明了和统一风格的原则，函数命名基本原则为大小写组合方式，各单词开头以大写字母开头。除了一些特殊函数、系统函数等除外。

有前缀名：void DEMO\_150G3\_InitDrv(void)

无前缀名：static void ScanTmrProc(void \*pdata)

有关键字：static void HdlMsg\_UP\_120R\_CMD\_LINK\_REQ(INT8U cmd, INT8U \*data, INT16U datalen)

系统函数：void main(void)

printf()

2.8.1.2. 函数名前缀

文件名按照软件分层架构层次原则进行增加前缀标识，从文件名可以看出本文件处于软件架构中的层次，如APP层定义前缀名为demo\_\*\*\*，如demo\_dc\_drv.c；DEMO层定义前缀名为demo\_\*\*\*，如demo\_gsm\_drv.c，PORT层定义前缀名为port\_\*\*.c，如port\_plat.c；

则一般函数对外的全局公共函数在文件名有前缀的情况下，则需要加上大写前缀名，如void DEMO\_150G3\_InitDrv(void)。

2.8.1.3. 函数名含义

从函数名就可以看出本函数实现的主要内容，如：

DEMO\_120R\_InitCom();

DEMO\_120R\_InitRecv();

DEMO\_120R\_InitSend();

DEMO\_120R\_InitCommon();

DEMO\_120R\_InitInfo();

DEMO\_120R\_InitPhone();

DEMO\_120R\_InitCan();

可以看出以上是初始化函数模块。

2.8.1.4. 函数名对齐

同一个模块或者同一类模块的函数名，命名的关键要排列整齐，；

DEMO\_120R\_InitCom();

DEMO\_120R\_InitRecv();

DEMO\_120R\_InitSend();

DEMO\_120R\_InitCommon();

DEMO\_120R\_InitInfo();

DEMO\_120R\_InitPhone();

DEMO\_120R\_InitCan();

2.8.1.5. 函数名分割

为了比较容易明显看出函数名分割界限，读懂函数含义，可以在在长函数名中适当加入下划线（“\_”）进行分割，如：

DEMO\_120R\_InitCom();

DEMO\_120R\_InitRecv();

DEMO\_120R\_InitSend();

DEMO\_120R\_InitCommon();

DEMO\_120R\_InitInfo();

DEMO\_120R\_InitPhone();

DEMO\_120R\_InitCan();

HdlMsg\_UP\_120R\_CMD\_LINK\_REQ

HdlMsg\_UP\_120R\_CMD\_BEAT\_REQ

HdlMsg\_UP\_120R\_ACK\_POWER\_CONTROL\_REQ

HdlMsg\_UP\_120R\_ACK\_RELINK\_REQ

2.8.2. 函数名应准确描述函数的功能

使用动宾词组为执行某操作的函数命名。如果是OOP 方法，可以只有动词（名词是对象本身）。避免使用无意义或者含义不清的动词如handle、process、function单独修饰，除非函数本身在设计时就已经被明确解释了。

明确函数功能，精确（而不是近似）地实现函数设计

示例：推荐的命名方式。

void InputKey(void);

int GetStrLen(char\* sptr);

void PrintFromUART(char ch);

static void ResetTmrProc(void \*pdata)

static void LinkTmrProc(void \*pdata)

BOOLEAN DEMO\_120R\_DirSend(INT16U cmd, INT8U \*ptr, INT16U len)

void \*DEMO\_MemMalloc(INT32U datalen);

void DEMO\_MemFree(void \*sptr);

2.8.3. 可重入函数

编写可重入函数时，应注意局部变量的使用，不应使用静态局部/全局变量或者全局。如果确实要用，则须进行一些特殊手段的处理才行，如关中断、申请信号量的方式。

示例：假设Exam是int型全局变量，函数Squre\_Exam返回Exam平方值。那么如下函数不具有可重入性。

unsigned int example( int para )

{

unsigned int temp;

Exam = para; // （\*\*）

temp = Square\_Exam( );

return temp;

}

此函数若被多个进程调用的话，其结果可能是未知的，因为当（\*\*）语句刚执行完后，另外一个使用本函数的进程可能正好被激活，那么当新激活的进程执行到此函数时，将使Exam赋与另一个不同的para值，所以当控制重新回到“temp = Square\_Exam()”后，计算出的temp很可能不是预想中的结果。此函数应如下改进。

unsigned int example( int para )

{

unsigned int temp;

[申请信号量操作] // 若申请不到“信号量”，说明另外的进程正处于给

Exam = para; // Exam赋值并计算其平方过程中（即正在使用此

temp = Square\_Exam(); // 信号），本进程必须等待其释放信号后，才可继

[释放信号量操作] // 续执行。若申请到信号，则可继续执行，但其它

//进程必须等待本进程释放信号量后，才能再使用

return temp; //本信号。

}

2.8.4. C51局部变量

C51平台下面的可重入函数的编写就要特别注意了，因为51平台是没有堆栈的，因此如果函数内哪怕只有有局部变量，或者有参数传递时，在链接时均是放置在内存中的，这样就无法做到可重入，必须增加上面7-2的特殊操作才行。如果使用的是Keil C51来作为软件开发平台，它提供了一个关键字reentrant来实现可重入，它会占用一块内存虚拟出栈的操作，以此来实现可重入，这块ram的区域和大小在startup.a51中进行配置。不过这么操作程序的执行效率可想而知。另外，它也能虚拟堆的操作，不过需要指定一块内存（最好是绝对定位），然后在malloc之前调用init\_mempool函数进行初始化。

2.8.5. 函数功能简化

一个函数最好仅实现一个功能。万能的函数好是好，不过过于复杂不容易被理解，效率也受影响，可裁减性也不好，也不利于测试和维护。

例如：在C51下，我们只需要计算累加和和累加和的异或，有一个万能的校验和计算工具，可以计算数十种校验和，对于我们来说，就有点杀鸡用牛刀了。另外，这样可能会造成写程序的人一种盲目，认为写出这样的函数才是牛，其实不是。

另外，我们也不提倡在这样的场合使用系统的标准库函数（如printf、memcpy），因为系统函数对于我们来说是个黑盒，而且复杂性很高，效率也得不到保障。

2.8.6. 函数分解

为了使程序结构和过程层次清晰、目的明确、可读性良好，要有目的的多编写函数。

一个2000行的代码，就一个main函数全部实现，谁受得了？因此多编写函数，按照功能、按照过程进行编写函数，可以使得程序结构清晰。

为简单功能编写函数。虽然有时对于简单的一两行代码编写一个函数有些浪费，但是对于代码的可读性来讲有所提高。特别是对于一些实现简单功能的、多次被调用的代码，最好是将它编写成函数或者用宏来表示。

示例：以下的表达方式不是很清晰

value = ( a > b ) ? a : b ;

改为如下就很清晰了。

int max (int a, int b)

{

return ((a > b) ? a : b);

}

value = max (a, b);

或改为如下。

#define MAX (a, b) (((a) > (b)) ? (a) : (b))

value = MAX (a, b);

但是不能盲目的多造函数，函数划分太多，接口就显得很多很乱，不利于理解和测试。对于功能不明确的、较小的、仅有一个扇入的函数，应该合并到上一级的函数去。比如上一点的示例，如果仅有一个上级函数调用它，就不必造成一个函数了，加行注释就好了。

2.8.7. 函数功能可预测性

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 。

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，当函数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，则返回的指针为错误的。

示例：如下函数，其返回值（即功能）是不可预测的。

unsigned int Integer\_Sum( unsigned int base )

{

unsigned int index;

static unsigned int sum = 0; // 注意，是static类型的，因为如果第二次 //调用了函数，那么sum的值就无法确定了。

// 若改为auto类型，则函数即变为可预测。

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

sum += index;

}

return sum;

}

2.8.8. 函数依赖性

尽量不要编写依赖于其他函数内部实现的函数。

此条为函数独立性的基本要求。由于目前大部分高级语言都是结构化的，所以通过具体语言的语法要求与编译器功能，基本就可以防止这种情况发生。但在汇编语言中，由于其灵活性，很可能使函数出现这种情况。

示例：如下是在DOS下TASM的汇编程序例子。过程Print\_Msg的实现依赖于Input\_Msg的具体实现，这种程序是非结构化的，难以维护、修改。

... // 程序代码

proc Print\_Msg // 过程（函数）Print\_Msg

... // 程序代码

jmp LABEL

... // 程序代码

endp

proc Input\_Msg // 过程（函数）Input\_Msg

... // 程序代码

LABEL:

... // 程序代码

Endp

2.8.9. 函数内聚

防止函数或过程内出现随机内聚。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或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。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。

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。

示例：如下函数就是一种随机内聚。

void Init\_Var( void )

{

Rect.length = 0;

Rect.width = 0; /\* 初始化矩形的长与宽 \*/

Point.x = 10;

Point.y = 10; /\* 初始化“点”的坐标 \*/

}

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。

应如下分为两个函数：

void Init\_Rect( void )

{

Rect.length = 0;

Rect.width = 0; /\* 初始化矩形的长与宽 \*/

}

void Init\_Point( void )

{

Point.x = 10;

Point.y = 10; /\* 初始化“点”的坐标 \*/

}

2.8.10. 函数扇入和扇出(扇入其实就是函数被调用的次数，扇出就是函数调用别的函数的数目)

设计高扇入、合理扇出（小于7 ）的函数，扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，如总是1，表明函数的调用层次可能过多，这样不利程序阅读和函数结构的分析，并且程序运行时会对系统资源如堆栈空间等造成压力。函数较合理的扇出（调度函数除外）通常是3-5。扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

2.8.11. 函数递归调用

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->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。

2.8.12. 函数输入输出参数

2.8.12.1. 函数参数合法性检查

在同一项目组中，应当明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责。缺省是由函数调用者负责。这部分处理不好，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

函数中要有对所输入的参数进行有必要有效性检查。函数的输入有两种：一种是参数输入；一种是非参数输入，如全局变量、数据文件的输入。在使用输入之前，应做必要的检查。

示例：下面的两个函数很容易引起系统崩溃，未检查ptr指针，未对除数是否为0进行检查。

int func1（char \*ptr）

{

int len;

len = strlen(ptr);

return MAX\_LEN /len;

}

Char \*GetUserName(struct user \*pUser)

{

　　return pUser->name;

}

2.8.12.2. 函数传递参数过多

避免设计出多参数函数，不使用的参数要从接口中去除。参数较多不仅影响执行效率，也增加了复杂性，降低了可测试性。当需要传递很多参数时，最好使用结构体指针传递方式。

示例：

/\* 串口配置参数 \*/

typedef struct {

INT32U port; /\* 串口编号,见SHELL\_UART\_PORT\_E \*/

INT32U baud; /\* 波特率, 1200~115200 \*/

INT32U parity; /\* 奇偶校验位, 见PE\_CHKBIT\_E \*/

INT32U databit; /\* 数据位, 见PE\_DATABIT\_E \*/

INT32U stopbit; /\* 停止位，见PE\_STOPBIT\_E \*/

INT32U rx\_len; /\* 配置接收缓存长度 \*/

INT32U tx\_len; /\* 配置发送缓存长度 \*/

INT8U \*rx\_ptr; /\* 配置接收缓存指针 \*/

INT8U \*tx\_ptr; /\* 配置发送缓存指针 \*/

} UART\_CFG\_T;

BOOLEAN PORT\_InitUart(UART\_CFG\_T \*cfg);

2.8.12.3. 传递参数作为工作变量

防止将函数的参数作为工作变量。这样有可能错误地改变参数内容，所以很危险。对必须改变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

示例：下函数的实现不太好。

void sum\_data(unsigned int num, int \*data, int \*sum)

{

unsigned int count;

\*sum = 0;

for (count = 0; count < num; count++) {

\*sum += data[count]; // sum成了工作变量，不太好。

}

}

若改为如下，则更好些。

void sum\_data( unsigned int num, int \*data, int \*sum )

{

unsigned int count ;

int sum\_temp;

sum\_temp = 0;

for (count = 0; count < num; count ++) {

sum\_temp += data[count];

}

\*sum = sum\_temp;

}

2.8.12.4. 函数的返回值

函数的返回值要清晰明了，符合调用者的需要，在设计时充分考虑返回值与出错返回值的区别，防止调用者被误用。

示例：

int func（int \*sptr, int len）

{

int cnt = 0;

while (len--) {

cnt += \*sptr++;

}

if (cnt == 0) {

return 0xffff;

} else {

return cnt; // 这里就没有考虑到cnt有可能本身就是0xffff

}

}

如非必要，最好不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回。

示例：

int func（int\* sptr, int len）

{

unsigned char cnt = 0;

while (len--) {

cnt += \*sptr++;

}

return (int)cnt;

}

2.8.13. 本章小结

我们在设计模块的时候，应当仔细分析模块的功能及性能需求，并进行初步划分，同时若有必要画出有关数据流图，据此来进行模块的函数划分与组织。这样能够划分出合理的函数结构，提高模块的最终效率和可维护性、可测性等。根据模块的功能图或/及数据流图映射出函数结构是常用方法之一。

模块设计完毕时可以倒过来重新检查模块，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。

2.9. 9．质量保证（以下几章都是照搬华为编程规范的）

9-1 在软件设计过程中构筑软件质量保证。

9-2 代码质量保证优先原则：

（1）正确性，指程序要实现设计要求的功能。

（2）稳定性、安全性，指程序稳定、可靠、安全。

（3）可测试性，指程序要具有良好的可测试性。

（4）规范/可读性，指程序书写风格、命名规则等要符合规范。

（5）全局效率，指软件系统的整体效率。

（6）局部效率，指某个模块/子模块/函数的本身效率。

（7）个人表达方式/个人方便性，指个人编程习惯。

9-3 尽量仅只引用属于自己的存贮空间

若模块封装的较好，那么一般不会发生非法引用他人的空间。不在特别情况下，不使用他人的公共变量。

9-4 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。

9-5 过程/ 函数中分配的内存，在过程/ 函数退出之前要释放

9-6 过程/ 函数中申请的（为打开文件而使用的）文件句柄，在过程/ 函数退出之前要关闭

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。

示例：下函数在退出之前，没有把分配的内存释放。

typedef unsigned char BYTE;

int example\_fun(BYTE gt\_len, BYTE \*gt\_code)

{

BYTE \*gt\_buf;

gt\_buf = (BYTE \*)malloc(MAX\_GT\_LENGTH);

... //program code, include check gt\_buf if or not NULL.

/\* global title length error \*/

if (gt\_len > MAX\_GT\_LENGTH) {

return GT\_LENGTH\_ERROR; // 忘了释放gt\_buf

}

... // other program code

}

应改为如下：

int example\_fun(BYTE gt\_len, BYTE \*gt\_code)

{

BYTE \*gt\_buf;

gt\_buf = (BYTE \*)malloc(MAX\_GT\_LENGTH);

... // program code, include check gt\_buf if or not NULL.

/\* global title length error \*/

if (gt\_len > MAX\_GT\_LENGTH) {

free(gt\_buf ); // 退出之前释放gt\_buf

return GT\_LENGTH\_ERROR;

}

... // other program code

}

9-7 防止内存操作越界。

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。

示例：数组的下标越界是比较常见的。假设某软件系统最多可由10个用户同时使用，用户号为1-10，那么如下程序存在问题。

#define MAX\_USR\_NUM 10

unsigned char usr\_login\_flg[MAX\_USR\_NUM]= "";

void set\_usr\_login\_flg(unsigned char usr\_no) {

if (!usr\_login\_flg[usr\_no]) {

usr\_login\_flg[usr\_no] = TRUE;

}

}

当usr\_no为10时，将使用usr\_login\_flg越界。可采用如下方式解决。

void set\_usr\_login\_flg(unsigned char usr\_no) {

if (!usr\_login\_flg[usr\_no - 1]) {

usr\_login\_flg[usr\_no - 1] = TRUE;

}

}

9-8 认真处理程序所能遇到的各种出错情况。

9-9 系统运行之初，要初始化有关变量及运行环境，防止未经初始化的变量被引用。

9-10 系统运行之初，要对加载到系统中的数据进行一致性检查。使用不一致的数据，容易使系统进入混乱状态和不可知状态。

9-11 严禁随意更改其它模块或系统的有关设置和配置。

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。

9-12 一个项目组中，不能随意改变与其它模块的接口。必须与其它人协商好了之后再做调整。

9-13 充分了解系统的接口之后，再使用系统提供的功能。

示例：在B型机的各模块与操作系统的接口函数中，有一个要由各模块负责编写的初始化过程，此过程在软件系统加载完成后，由操作系统发送的初始化消息来调度。因此就涉及到初始化消息的类型与消息发送的顺序问题，特别是消息顺序，若没搞清楚就开始编程，很容易引起严重后果。以下引自B型机曾出现过的实际代码，其中使用了FID\_FETCH\_DATA与FID\_INITIAL初始化消息类型，注意B型机的系统是在FID\_FETCH\_DATA之前发送FID\_INITIAL的。

MID alarm\_module\_list[MAX\_ALARM\_MID];

int FAR SYS\_ALARM\_proc(FID function\_id, int handle)

{

\_UI i, j;

switch (function\_id)

{

... // program code

case FID\_INITAIL:

for (i = 0; i < MAX\_ALARM\_MID; i++) {

if (alarm\_module\_list[i]== BAM\_MODULE // \*\*）

|| (alarm\_module\_list[i]== LOCAL\_MODULE) {

for (j = 0; j < ALARM\_CLASS\_SUM; j++) {

FAR\_MALLOC(...);

}

}

}

... // program code

break;

case FID\_FETCH\_DATA:

... // program code

Get\_Alarm\_Module(); // 初始化alarm\_module\_list

break;

... // program code

}

}

由于FID\_INITIAL是在FID\_FETCH\_DATA之前执行的，而初始化alarm\_module\_list是在FID\_FETCH\_DATA中进行的，故在FID\_INITIAL中（\*\*）处引用alarm\_module\_list变量时，它还没有被初始化。这是个严重错误。

应如下改正：要么把Get\_Alarm\_Module函数放在FID\_INITIAL中（\*\*）之前；要么就必须考虑（\*\*）处的判断语句是否可以用（不使用alarm\_module\_list变量的）其它方式替代，或者是否可以取消此判断语句。

9-14 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。当编完程序后，应对这些操作符进行彻底检查。

9-15 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。

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，如把“&”写成“&&”，或反之。

ret\_flg = (pmsg->ret\_flg & RETURN\_MASK);

被写为：

ret\_flg = (pmsg->ret\_flg && RETURN\_MASK);

rpt\_flg = (VALID\_TASK\_NO(taskno) && DATA\_NOT\_ZERO(stat\_data));

被写为：

rpt\_flg = (VALID\_TASK\_NO(taskno) & DATA\_NOT\_ZERO(stat\_data));

if (flag == true) 被写为 if (flag = true)

9-16 有可能的话，if 语句尽量加上else 分支，对没有else 分支的语句要小心对待；switch 语句必须有default 分支。对于switch语句，每个case都必须以break结束。如果在特别情况需要不用break，直接跳到下一个case执行，必须以明确的注释说明。

9-17 不要滥用goto 语句。

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。这对于写惯了汇编的人会特别不习惯。我们仅允许一种场合使用goto，那就是从循环体内向外且向下跳出的情况下使用goto，严禁跳入循环体以及向上跳转。

9-18 不在非硬件模块中使用与硬件或操作系统关系很大的语句，而使用建议的标准语句，以提高软件的可移植性和可重用性。

9-19 除非为了满足特殊需求，避免使用嵌入式汇编。

说明：程序中嵌入式汇编，一般都对可移植性有较大的影响。

9-20 精心地构造、划分子模块，并按“接口”部分及“内核”部分合理地组织子模块，以提高“内核”部分的可移植性和可重用性。

对不同产品中的某个功能相同的模块，若能做到其内核部分完全或基本一致，那么无论对产品的测试、维护，还是对以后产品的升级都会有很大帮助。

9-21 精心构造算法，并对其性能、效率进行测试。

9-22 对较关键的算法最好使用其它算法来确认。

9-23 时刻注意表达式是否会上溢、下溢。

示例：如下程序将造成变量下溢。

unsigned char size;

while (size-- >= 0) {// 将出现下溢

... // program code

}

当size等于0时，再减1不会小于0，而是0xFF，故程序是一个死循环。应如下修改。

char size; // 从unsigned char 改为char

while (size-- >= 0) {

... // program code

}

9-24 使用变量时要注意其边界值的情况

示例：如C语言中字符型变量，有效值范围为-128到127。故以下表达式的计算存在一定风险。

char chr = 127;

int sum = 200;

chr += 1; // 127为chr的边界值，再加1将使chr上溢到-128，而不是128。

sum += chr; // 故sum的结果不是328，而是72。

若chr与sum为同一种类型，或表达式按如下方式书写，可能会好些。

sum = sum + chr + 1;

9-25 留心程序机器码大小（如指令空间大小、数据空间大小、堆栈空间大小等）是否超出系统有关限制。

9-26 为用户提供良好的接口界面，使用户能较充分地了解系统内部运行状态及有关系统出错情况。

9-27 系统应具有一定的容错能力，对一些错误事件（如用户误操作等）能进行自动补救。

9-28 对一些具有危险性的操作代码（如写flash、断油断电等）要仔细考虑，防止对数据、硬件等的安全构成危害，以提高系统的安全性。

9-29 使用第三方提供的软件开发工具包或控件时，要注意以下几点：

（1）充分了解应用接口、使用环境及使用时注意事项。

（2）不能过分相信其正确性。

（3）除非必要，不要使用不熟悉的第三方工具包与控件。

使用工具包与控件，可加快程序开发速度，节省时间，但使用之前一定对它有较充分的了解，同时第三方工具包与控件也有可能存在问题。

9-30 资源文件（多语言版本支持），如果资源是对语言敏感的，应让该资源与源代码文件脱离，具体方法有下面几种：使用单独的资源文件、DLL 文件或其它单独的描述文件（如数据库格式）。

2.10. 10．可测性

10-1 在同一项目组或产品组内，要有一套统一的为集成测试与系统联调准备的调试开关及相应打印函数，并且要有详细的说明。本规则是针对项目组或产品组的。

10-2 在同一项目组或产品组内，调试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）及位置。

说明：统一的调试信息格式便于集成测试。

10-3 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（通过调试开关）。

说明：为单元测试而准备。

10-4 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。

10-5 使用断言来发现软件问题，提高代码可测性。

说明：断言是对某种假设条件进行检查（可理解为若条件成立则无动作，否则应报告），它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。实际应用时，可根据具体情况灵活地设计断言。

示例：下面是C语言中的一个断言，用宏来设计的。（其中NULL为0L）

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

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

{

printf("\n[EXAM]Assert failed: %s, line %u\n",

file\_name, line\_no);

abort( );

}

#define EXAM\_ASSERT( condition )

if (condition) // 若条件成立，则无动作

NULL;

else // 否则报告

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

#else // 若不使用断言测试

#define EXAM\_ASSERT(condition) NULL

#endif /\* end of ASSERT \*/

10-6 用断言来检查程序正常运行时不应发生但在调试时有可能发生的非法情况。

10-7 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。

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

10-8 对较复杂的断言加上明确的注释。

说明：为复杂的断言加注释，可澄清断言含义并减少不必要的误用。

10-9 用断言确认函数的参数。

示例：假设某函数参数中有一个指针，那么使用指针前可对它检查，如下

int exam\_fun(unsigned char \*str)

{

EXAM\_ASSERT(str != NULL); // 用断言检查“假设指针不为空”这个条件

... //other program code

}

10-10 用断言保证没有定义的特性或功能不被使用

示例：假设某通信模块在设计时，准备提供“无连接”和“连接” 这两种业务。但当前的版本中仅实现了“无连接”业务，且在此版本的正式发行版中，用户（上层模块）不应产生“连接”业务的请求，那么在测试时可用断言检查用户是否使用“连接”业务。如下。

#define EXAM\_CONNECTIONLESS 0 // 无连接业务

#define EXAM\_CONNECTION 1 // 连接业务

int msg\_process(EXAM\_MESSAGE \*msg)

{

unsigned char service; /\* message service class \*/

EXAM\_ASSERT(msg != NULL);

service = get\_msg\_service\_class(msg);

EXAM\_ASSERT(service != EXAM\_CONNECTION); // 假设不使用连接业务

... //other program code

}

10-11 用断言对程序开发环境（OS/Compiler/Hardware）的假设进行检查

说明：程序运行时所需的软硬件环境及配置要求，不能用断言来检查，而必须由一段专门代码处理。用断言仅可对程序开发环境中的假设及所配置的某版本软硬件是否具有某种功能的假设进行检查。如某网卡是否在系统运行环境中配置了，应由程序中正式代码来检查；而此网卡是否具有某设想的功能，则可由断言来检查。

对编译器提供的功能及特性假设可用断言检查，原因是软件最终产品（即运行代码或机器码）与编译器已没有任何直接关系，即软件运行过程中（注意不是编译过程中）不会也不应该对编译器的功能提出任何需求。

示例：用断言检查编译器的int型数据占用的内存空间是否为2，如下。

EXAM\_ASSERT(sizeof( int ) == 2);

10-12 正式软件产品中应把断言及其它调试代码去掉（即把有关的调试开关关掉）

说明：加快软件运行速度。

10-13 在软件系统中设置与取消有关测试手段，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。

10-14 用调试开关来切换软件的DEBUG 版和正式版，而不要同时存在正式版本和DEBUG 版本的不同源文件，以减少维护的难度。

10-15 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。

10-16 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，并设计好各种调试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。

10-17 调试开关应分为不同级别和类型

说明：调试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试；针对模块或系统某功能的调试；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试，并且便于模块的单元测试、系统联调等。

10-18 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

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

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

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

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

{

printf("\n[EXAM]Error Report: %s, line %u\n",

file\_name, line\_no);

}

#define ASSERT\_REPORT(condition)

if (condition) // 若条件成立，则无动作

NULL;

else // 否则报告

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

#else // 若不使用断言测试

#define ASSERT\_REPORT( condition ) NULL

#endif /\* end of ASSERT \*/

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

{

switch(msg\_name)

{

case MSG\_ONE:

... // 消息MSG\_ONE处理

return MSG\_HANDLE\_SUCCESS;

... // 其它合法消息处理

default:

... // 消息出错处理

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

return MSG\_HANDLE\_ERROR;

}

}

2.11. 11．程序效率

11-1 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。

11-2 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。

11-3 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。

11-4 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。

示例：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 name[8];

BYTE age;

BYTE sex;

BYTE class;

BYTE subject;

float score;

} 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 name[8];

BYTE age;

BYTE sex;

BYTE class;

} STUDENT;

typedef struct STUDENT\_SCORE\_STRU

{

WORD student\_index;

BYTE subject;

float score;

} STUDENT\_SCORE;

11-5 循环体内工作量最小化。

说明：应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。

示例：如下代码效率不高。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++) {

sum += ind;

back\_sum = sum; /\* backup sum \*/

}

语句“back\_sum = sum;”完全可以放在for语句之后，如下。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++) {

sum += ind;

}

back\_sum = sum; /\* backup sum \*/

11-6 仔细分析有关算法，并进行优化。

11-7 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。

11-8 对模块中函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高程序效率。

说明：软件系统的效率主要与算法、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。

11-9 编程时，要随时留心代码效率；优化代码时，要考虑周全。

11-10 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。

说明：对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

11-11 要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数。

说明：只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方式。嵌入汇编可提高时间及空间效率，但也存在一定风险。

11-12 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。

说明：这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。

11-13 在多重循环中，应将最忙的循环放在最内层。

说明：减少CPU切入循环层的次数。

示例：如下代码效率不高。

for (row = 0; row < 100; row++) {

for (col = 0; col < 5; col++) {

sum += a[row][col];

}

}

可以改为如下方式，以提高效率。

for (col = 0; col < 5; col++) {

for (row = 0; row < 100; row++) {

sum += a[row][col];

}

}

11-14 尽量减少循环嵌套层次。

11-15 避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中。

说明：目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不可以。

示例：如下代码效率稍低。

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++) {

if (data\_type == RECT\_AREA) {

area\_sum += rect\_area[ind];

} else {

rect\_length\_sum += rect[ind].length;

rect\_width\_sum += rect[ind].width;

}

}

因为判断语句与循环变量无关，故可如下改进，以减少判断次数。

if (data\_type == RECT\_AREA) {

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++) {

area\_sum += rect\_area[ind];

}

} else {

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++) {

rect\_length\_sum += rect[ind].length;

rect\_width\_sum += rect[ind].width;

}

}

11-16 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。

示例：如下表达式运算可能要占较多CPU资源。

#define PAI 3.1416

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

应把浮点除法改为浮点乘法。

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

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

11-17 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。

2.12. 12．代码编辑、编译、审查

12-1 打开编译器的所有告警开关对程序进行编译。

12-2 在产品软件（项目组）中，要统一编译开关选项。

12-3 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。

12-4 测试部测试产品之前，应对代码进行抽查及评审。

12-5 编写代码时要注意随时保存，并定期备份，防止由于断电、硬盘损坏等原因造成代码丢失。

12-6 同产品软件（项目组）内，最好使用相同的编辑器，并使用相同的设置选项。

说明：同一项目组最好采用相同的智能语言编辑器，如Muiti Editor，Visual Editor等，并设计、使用一套缩进宏及注释宏等，将缩进等问题交由编辑器处理。

12-7 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。

12-8 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。

12-9 某些语句经编译后产生告警，但如果你认为它是正确的，那么应通过某种手段去掉告警信息。

说明：在Borland C/C++中，可用“#pragma warn”来关掉或打开某些告警。

示例：

#pragma warn -rvl // 关闭告警

int examples\_fun( void )

{

// 程序，但无return语句。

}

#pragma warn +rvl // 打开告警

编译函数examples\_fun时本应产生“函数应有返回值”告警，但由于关掉了此告警信息显示，所以编译时将不会产生此告警提示。

12-10 使用代码检查工具（如C 语言用PC-Lint ）对源程序检查。

12-11 使用软件工具（如 LogiSCOPE ）进行代码审查。

2.13. 13．代码测试、维护

13-1 单元测试要求至少达到语句覆盖。

13-2 单元测试开始要跟踪每一条语句，并观察数据流及变量的变化。

13-3 清理、整理或优化后的代码要经过审查及测试。

13-4 代码版本升级要经过严格测试。

13-5 使用工具软件对代码版本进行维护。

13-6 正式版本上软件的任何修改都应有详细的文档记录。

13-7 发现错误立即修改，并且要记录下来。

13-8 关键的代码在汇编级跟踪。

13-9 仔细设计并分析测试用例，使测试用例覆盖尽可能多的情况，以提高测试用例的效率。

13-10 尽可能模拟出程序的各种出错情况，对出错处理代码进行充分的测试。

13-11 仔细测试代码处理数据、变量的边界情况。

13-12 保留测试信息，以便分析、总结经验及进行更充分的测试。

13-13 不应通过“试”来解决问题，应寻找问题的根本原因。

13-14 对自动消失的错误进行分析，搞清楚错误是如何消失的。

13-15 修改错误不仅要治标，更要治本。

13-16 测试时应设法使很少发生的事件经常发生。

13-17 明确模块或函数处理哪些事件，并使它们经常发生。

13-18 坚持在编码阶段就对代码进行彻底的单元测试，不要等以后的测试工作来发现问题。

13-19 去除代码运行的随机性（如去掉无用的数据、代码及尽可能防止并注意函数中的“内部寄存器”等），让函数运行的结果可预测，并使出现的错误可再现。