**C++的编码规范**

# **1 排版**

1-1：程序块要采用缩进风格编写，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。

1-2：相对独立的程序块之间、变量说明之后必须加空行。

示例：

if (!valid\_ni(ni))

{

    ... // program code

}

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

repssn\_ni  = ssn\_data[index].ni;

1-3：较长的语句（>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。

示例：

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

                          + 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));

1-4：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。

示例：

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

    && (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

}

1-5：若函数或过程中的参数较长，则要进行适当的划分。

示例：

n7stat\_str\_compare((BYTE \*) & stat\_object,

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

                   sizeof (\_STAT\_OBJECT));

n7stat\_flash\_act\_duration( stat\_item, frame\_id \*STAT\_TASK\_CHECK\_NUMBER

                                      + index, stat\_object );

1-6：不允许把多个短语句写在一行中，即一行只写一条语句。

示例：

rect.length = 0;

rect.width  = 0;

1-7：if、for、do、while、case、switch、default等语句自占一行，且if、for、do、while等语句的执行语句部分无论多少都要加括号{}。

示例：

if (pUserCR == NULL)

{

    return;

}

1-8：对齐只使用空格键，不使用TAB键。

说明：以免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，不要使用BC作为编辑器合版本，因为BC会自动将8个空格变为一个TAB键，因此使用BC合入的版本大多会将缩进变乱。

1-9：函数或过程的开始、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，case语句下的情况处理语句也要遵从语句缩进要求。

1-10：程序块的分界符（如C/C++语言的大括号‘{’和‘}’）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类的定义、结构的定义、枚举的定义以及if、for、do、while、switch、case语句中的程序都要采用如上的缩进方式。

示例：

for (...)

{

    ... // program code

}

if (...)

{

    ... // program code

}

void example\_fun( void )

{

    ... // program code

}

1-11：在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如－>），后不应加空格。

说明：采用这种松散方式编写代码的目的是使代码更加清晰。

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。

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。给操作符留空格时不要连续留两个以上空格。

**2 注释**

2-1：一般情况下，源程序有效注释量必须在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。

2-2：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有函数功能简要说明。

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

  Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.

  File name:      // 文件名

  Author:       Version:        Date: // 作者、版本及完成日期

  Description:    // 用于详细说明此程序文件完成的主要功能，与其他模块

                  // 或函数的接口，输出值、取值范围、含义及参数间的控

                  // 制、顺序、独立或依赖等关系

  Others:         // 其它内容的说明

  Function List:  // 主要函数列表，每条记录应包括函数名及功能简要说明

    1. ....

  History:        // 修改历史记录列表，每条修改记录应包括修改日期、修改

                  // 者及修改内容简述

    1. Date:

       Author:

       Modification:

    2. ...

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

2-3：源文件头部应进行注释，列出：版权说明、版本号、生成日期、作者、模块目的/功能、主要函数及其功能、修改日志等。

示例：下面这段源文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

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

  Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.

  FileName: test.cpp

  Author:        Version :          Date:

  Description:     // 模块描述

  Version:         // 版本信息

  Function List:   // 主要函数及其功能

    1. -------

  History:         // 历史修改记录

      < author>  <time>   <version >   < desc>

      David    96/10/12     1.0     build this moudle

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

2-4：函数头部应进行注释，列出：函数的目的/功能、输入参数、输出参数、返回值、调用关系（函数、表）等。

示例：下面这段函数的注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

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

  Function:       // 函数名称

  Description:    // 函数功能、性能等的描述

  Calls:          // 被本函数调用的函数清单

  Called By:      // 调用本函数的函数清单

  Table Accessed: // 被访问的表（此项仅对于牵扯到数据库操作的程序）

  Table Updated:  // 被修改的表（此项仅对于牵扯到数据库操作的程序）

  Input:          // 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。

  Output:         // 对输出参数的说明。

  Return:         // 函数返回值的说明

  Others:         // 其它说明

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

2-5：边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。

2-6：注释的内容要清楚、明了，含义准确，防止注释二义性。

说明：错误的注释不但无益反而有害。

## 2-7：避免在注释中使用缩写，特别是非常用缩写。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

2-8：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。

示例：

/\* active statistic task number \*/

#define MAX\_ACT\_TASK\_NUMBER 1000

#define MAX\_ACT\_TASK\_NUMBER 1000 /\* active statistic task number \*/

2-10：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。

示例：可按如下形式说明枚举/数据/联合结构。

/\* sccp interface with sccp user primitive message name \*/

enum  SCCP\_USER\_PRIMITIVE

{

    N\_UNITDATA\_IND, /\* sccp notify sccp user unit data come \*/

    N\_NOTICE\_IND,   /\* sccp notify user the No.7 network can not \*/

                    /\* transmission this message \*/

    N\_UNITDATA\_REQ, /\* sccp user's unit data transmission request\*/

};

2-11：全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。

示例：

/\* The ErrorCode when SCCP translate \*/

/\* Global Title failure, as follows \*/      // 变量作用、含义

/\* 0 － SUCCESS   1 － GT Table error \*/

/\* 2 － GT error  Others － no use  \*/       // 变量取值范围

/\* only  function  SCCPTranslate() in \*/

/\* this modual can modify it,  and  other \*/

/\* module can visit it through call \*/

/\* the  function GetGTTransErrorCode() \*/    // 使用方法

BYTE g\_GTTranErrorCode;

2-12：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。

示例：如下例子，排版不整齐，阅读稍感不方便。

void example\_fun( void )

{

/\* code one comments \*/

    CodeBlock One

        /\* code two comments \*/

    CodeBlock Two

}

应改为如下布局。

void example\_fun( void )

{

    /\* code one comments \*/

    CodeBlock One

    /\* code two comments \*/

    CodeBlock Two

}

2-13：将注释与其上面的代码用空行隔开。

示例：

/\* code one comments \*/

program code one

/\* code two comments \*/

program code two

2-14：对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。

说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

2-15：对于switch语句下的case语句，如果因为特殊情况需要处理完一个case后进入下一个case处理，必须在该case语句处理完、下一个case语句前加上明确的注释。

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。

示例（注意斜体加粗部分）：

case CMD\_UP:

    ProcessUp();

    break;

case CMD\_DOWN:

    ProcessDown();

    break;

case CMD\_FWD:

    ProcessFwd();

if (...)

{

    ...

    break;

}

else

{

    ProcessCFW\_B();   // now jump into case CMD\_A

}

case CMD\_A:

    ProcessA();

    break;

case CMD\_B:

    ProcessB();

    break;

case CMD\_C:

    ProcessC();

     break;

case CMD\_D:

    ProcessD();

    break;

...

2-16：避免在一行代码或表达式的中间插入注释。

说明：除非必要，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。

2-17：通过对函数或过程、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。

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。

2-18：在代码的功能、意图层次上进行注释，提供有用、额外的信息。

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，防止没必要的重复注释信息。

示例：

/\* if receive\_flag is TRUE \*/

if (receive\_flag)

而如下的注释则给出了额外有用的信息。

/\* if mtp receive a message from links \*/

if (receive\_flag)

2-19：在程序块的结束行右方加注释标记，以表明某程序块的结束。

说明：当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。

示例：

if (...)

{

    // program code

    while (index < MAX\_INDEX)

    {

        // program code

    } /\* end of while (index < MAX\_INDEX) \*/ // 指明该条while语句结束

} /\* end of  if (...)\*/ // 指明是哪条if语句结束

2-20：注释格式尽量统一，建议使用“/\* …… \*/”。

2-21：注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非能用非常流利准确的英文表达。

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，建议使用中文。

# **3 标识符命名**

3-1：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大家基本可以理解的缩写，避免使人产生误解。

示例：如下单词的缩写能够被大家基本认可。

temp 可缩写为  tmp  ;

flag 可缩写为  flg  ;

statistic 可缩写为  stat ;

increment 可缩写为  inc  ;

message 可缩写为  msg  ;

除非必要，不要用数字或较奇怪的字符来定义标识符。

3-2：命名中若使用特殊约定或缩写，则要有注释说明。

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。

3-3：自己特有的命名风格，要自始至终保持一致，不可来回变化。

（1）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。

3-4：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。

示例：下面所示的局部变量名的定义方法可以借鉴。

int liv\_Width

其变量名解释如下：

       l      局部变量（Local）  （其它：g    全局变量（Global）...）

       i      数据类型（Interger）

       v      变量（Variable）   （其它：c    常量（Const）...）

       Width  变量含义

这样可以防止局部变量与全局变量重名。

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。

3-5：命名规范必须与所使用的系统风格保持一致。

示例： Add\_User不允许，add\_user、AddUser、m\_AddUser允许。

# **4 可读性**

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。

示例：下列语句中的表达式

word = (high << 8) | low     (1)

if ((a | b) && (a & c))      (2)

if ((a | b) < (c & d))       (3)

如果书写为

high << 8 | low

a | b && a & c

a | b < c & d

由于

high << 8 | low = ( high << 8) | low,

a | b && a & c = (a | b) && (a & c)，

(1)(2)不会出错，但语句不易理解；

a | b < c & d = a | （b < c） & d，(3)造成了判断条件出错。

4-2源程序中关系较为紧密的代码应尽可能相邻。

4-3：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。

示例：如下的程序可读性差。

if (Trunk[index].trunk\_state == 0)

{

    Trunk[index].trunk\_state = 1;

    ...  // program code

}

应改为如下形式。

#define TRUNK\_IDLE 0

#define TRUNK\_BUSY 1

if (Trunk[index].trunk\_state == TRUNK\_IDLE)

{

    Trunk[index].trunk\_state = TRUNK\_BUSY;

    ...  // program code

}

4-4：不要使用难懂的技巧性很高的语句，除非很有必要时。

示例：如下表达式，考虑不周就可能出问题，也较难理解。

\* stat\_poi ++ += 1;

\* ++ stat\_poi += 1;

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; ”

# **5 变量、结构**

5-1：去掉没必要的公共变量。

5-2：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公共变量，防止多个不同模块或函数都可以修改、创建同一公共变量的现象。

5-3：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。

示例：在源文件中，可按如下注释形式说明。

RELATION    System\_Init    Input\_Rec    Print\_Rec   Stat\_Score

Student     Create         Modify       Access      Access

Score       Create         Modify       Access      Access, Modify

说明：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。

5-4：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。

示例：如下例子（在DOS下BC3.1环境中），在移植时可能产生问题。

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

}

5-5：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。

说明：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。

5-6结构的功能要单一，是针对一种事务的抽象。

5-7：当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生。

5-8：不要设计面面俱到、非常灵活的数据结构。

5-9：防止局部变量与公共变量同名。

5-10：不同结构间的关系不要过于复杂。

5-11：严禁使用未经初始化的变量作为右值。

5-12：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。

5-13：仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。

5-14：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。

5-15：留心具体语言及编译器处理不同数据类型的原则及有关细节。

说明：如在C语言中，static局部变量将在内存“数据区”中生成，而非static局部变量将在“堆栈”中生成。这些细节对程序质量的保证非常重要。

5-16：编程时，要注意数据类型的强制转换。

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。

5-17：对编译系统默认的数据类型转换，也要有充分的认识。

5-18：尽量减少没有必要的数据类型默认转换与强制转换。

5-19：合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。

5-20：对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其命名方式在同一产品中的统一。

说明：使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序清晰、简洁。

# **6 函数、过程**

6-1：对所调用函数的错误返回码要仔细、全面地处理。

6-2：明确函数功能，精确（而不是近似）地实现函数设计。

6-3：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）。

6-4：编写可重入函数时，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。

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;

}

6-5：在同一项目组应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。

6-6：防止将函数的参数作为工作变量。

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;

}

6-7：函数的规模尽量限制在200行以内。

6-8：一个函数仅完成一件功能。

6-9：为简单功能编写函数。

示例：如下语句的功能不很明显。

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))

6-10：不要设计多用途面面俱到的函数。

6-11：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。

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。

                                 // 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;

}

6-12：尽量不要编写依赖于其他函数内部实现的函数。

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

6-13：避免设计多参数函数，不使用的参数从接口中去掉。

6-14：非调度函数应减少或防止控制参数，尽量只使用数据参数。

示例：如下函数构造不太合理。

int add\_sub( int a, int b, unsigned char add\_sub\_flg )

{

    if (add\_sub\_flg == INTEGER\_ADD)

    {

        return (a + b);

    }

    else

    {

        return (a  b);

    }

}

不如分为如下两个函数清晰。

int add( int a, int b )

{

    return (a + b);

}

int sub( int a, int b )

{

    return (a  b);

}

6-15：检查函数所有参数输入的有效性。

6-16：检查函数所有非参数输入的有效性，如数据文件、公共变量等。

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。函数在使用输入之前，应进行必要的检查。

6-17：函数名应准确描述函数的功能。

6-18：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）。

示例：参照如下方式命名函数。

void print\_record( unsigned int rec\_ind ) ;

int  input\_record( void ) ;

unsigned char get\_current\_color( void ) ;

# **7 可测性**

7-1：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。

7-2：在同一项目组或产品组内，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。

7-3：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。

7-4：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。

7-5：使用断言来发现软件问题，提高代码可测性。

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 \*/

7-6：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。

7-7：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。

7-8：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。

7-9：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

}

7-10：用断言保证没有定义的特性或功能不被使用。

7-11：用断言对程序开发环境（OS/Compiler/Hardware）的假设进行检查。

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 );

7-12：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）。

说明：加快软件运行速度。

7-13：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。

7-14：用调测开关来切换软件的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。

7-15：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。

# **8 程序效率**

8-1：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。

8-2：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。

8-3：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。

8-4：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;

8-5：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 \*/

# **9 质量保证**

9-1：在软件设计过程中构筑软件质量。

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：只引用属于自己的存贮空间。

说明：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。

9-4：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。

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。

示例：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。

说明：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。

说明：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：不能随意改变与其它模块的接口。

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。以下示例引自B型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。

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。

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。

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。

示例：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 ));

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。

9-17：Unix下，多线程的中的子线程退出必需采用主动退出方式，即子线程应return出口。

9-18：不要滥用goto语句。

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。

示例代码：

Main.c

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

  文件名: main.c

 作者:薛涵             日期:2019/6/18

学院：信息科学与技术学院 班级：信1604-1

 版本:1.0         // 版本信息

历史更改：无

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

# include "delay.h"

# include "sys.h"

# include "led.h"

# include "chaochuan.h"

# include "usart.h"

# include "wifi.h"

#include "stdio.h"

#include "yuyinkou.h"

#include "dht11.h"

#include "fengshan.h"

#include "hongwai.h"

#include "moto.h"

int main(void)

{

int col;

int shi;

LED\_Init();

HONGWAI\_Init();

FENGSHAN\_Init();

DHT11\_Start(); //调用DHT11的预备函数

delay\_init();

Moto\_Init();

US100\_Init(115200); //串口2初始化

usart3\_Init(9600); //串口3初始化

uart\_init(9600); //串口1初始化

LED0=1;

LED1=1;

connect\_wifi() ; //WiFi连接函数

while(1)

{

hongwai();

col=get\_col(); //得到衣物信号

shi=get\_shi(); //得到湿度信号

if(col!=100)

{

LED0=!LED0;

ZZ(col); //向电机传递衣物信号

}

if(shi>=60) FORWARD2(); //两个风扇开启

if(shi>=30&&shi<60) FORWARD1(); //一个风扇开启

if(shi<30) STOP2(); //全都停止

delay\_ms(200);

}

}

dht11.c

#include "dht11.h" //导入头文件dht11.h

#include "stdio.h"

#include "stdlib.h"

# include "chaochuan.h"

unsigned char strgai[30];

//strgai[0]='$';

unsigned char strHumi[30];

//strHumi[0]='$'

static void DHT11\_Input(void); //声明static函数DHT11\_Input,用于配置DHT11的引脚为输入模式

static void DHT11\_Output(void); //声明static函数DHT11\_Output,用于配置DHT11的引脚为输出模式

int m;

int nn=0;

/\*定义全局变量,分别用于统计湿度高位;湿度低位;温度高位;温度低位\*/

u8 DHT11\_hem\_high,DHT11\_hem\_low,DHT11\_temp\_high,DHT11\_temp\_low;

static void DHT11\_Input(void) //设置DHT11数据引脚的输入模式的配置

{

GPIO\_InitTypeDef GPIO\_InitStructure;

RCC\_APB2PeriphClockCmd(DHT11\_RCC,ENABLE); //开启GPIOD的时钟

GPIO\_InitStructure.GPIO\_Pin = DHT11\_PIN; //设置引脚为PD13

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz; //速率50MHz

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_IPU; //配置输入模式为上拉输入模式

GPIO\_Init(DHT11\_GPIO, &GPIO\_InitStructure);//把上面的配置初始化

}

static void DHT11\_Output(void)//设置DHT11数据引脚的输出模式的配置

{

GPIO\_InitTypeDef GPIO\_InitStructure;

RCC\_APB2PeriphClockCmd(DHT11\_RCC,ENABLE);//开启GPIOD时钟

GPIO\_InitStructure.GPIO\_Pin = DHT11\_PIN;//设置引脚为PD13

GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;

GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_Out\_PP;//配置为推挽输出

GPIO\_Init(DHT11\_GPIO, &GPIO\_InitStructure);//初始化

}

/\*从这里开始就根据DHT11使用说明的流程进行,确保万无一失\*/

void DHT11\_Start(void)//DHT11的开始程序

{

delay\_init();//调用Systick,开启

DHT11\_Output();//设置为输出模式

DHT11\_hem\_high = 0;

DHT11\_hem\_low = 0;

DHT11\_temp\_high = 0;

DHT11\_temp\_low = 0;

}

u16 DHT11\_ReadByte(void)//读取DHT11每个字节的函数

{

u8 temp=0,i,cout;//三个变量的含义(1)temp用于储存数据(2)i表示字节的位数(3)cout用于保证执行条件或函数

DHT11\_Input();

for(i=0;i<8;i++)//这个for循环的意义就是依次读8位字节,所以只执行8次,

{

cout=1;//确保执行while

while(!DHT11\_ReadBit() && cout++);//当 [读取到PE13的电平不为1 且 cout++都为真] 的条件下执行循环,否则跳出循环

delay\_us(30);//延时30us,再读取PE13的状态

temp = temp << 1;//表示把上次的temp的值左移一位,因为DHT11读取的数据是先出高值再出低值,

if(DHT11\_ReadBit() == Bit\_SET)//如果 [ 读取到PE13的电平等于Bit\_SET的值 为真 ] ,则执行表达式

{

temp |=1; //temp=temp|1,进行或运算,有1得1

}

//等待输出的电平为低电平,进入下一位数据接收

cout=1;

while(DHT11\_ReadBit() && cout++);

}

return temp;//返回接收到的temp

}

void DHT11\_ReadData(void)//读取DHT11的数据,把读取到的自己整合

{

u16 cout = 1;//用于保证语句执行的变量

u16 temp\_high,temp\_low,hem\_high,hem\_low,Check;//用于整合字节的变量,分别是温度高位,温度低位,湿度高位,湿度低位,校验和

DHT11\_Output();//把引脚切换为输出模式

DHT11\_ResetBit();//拉低引脚电平,就是输出高电平

delay\_ms(20);//持续20ms,数据手册上说要至少拉低18ms,保证DHT11能够检测到起始信号

DHT11\_SetBit();//抬高引脚的电平;也就是输出低电平

delay\_us(30);//持续30us,数据手册上说的区间是20-40us

DHT11\_Input();//设置PE13为输入模式

/\* 如果MCU采集到PE13的数据输入是低电平的话则执行表达式 \*/

if(DHT11\_ReadBit() == Bit\_RESET)

{

cout = 1;//低电平响应信号,等待DHT11响应完毕

while(!DHT11\_ReadBit() && cout++);

cout = 1;//高电平传输数据,等待DHT11传输完毕,持续26us - 28us表示的是0;持续70us表示的是1

while(DHT11\_ReadBit() && cout++);

/\*开始按顺序接收DHT11的回传数据\*/

/\*根据DHT11的使用说明一共会接收5个字节的数据\*/

hem\_high = DHT11\_ReadByte();//第一个接收到的是湿度的高八位,温度整数数据

hem\_low = DHT11\_ReadByte();//第二个接收到的是湿度的低八位,温度小数数据

temp\_high = DHT11\_ReadByte();//第三个接受到的是温度的高八位,湿度整数数据

temp\_low = DHT11\_ReadByte();//第四个接收到的是温度的低八位,湿度小数数据

Check = DHT11\_ReadByte();//最后接收到的是前四位的和,用于校验,确保精度

if(Check == ( temp\_high + temp\_low + hem\_high + hem\_low ))//校验和.如果前4个字节加起来的和等于Check

{

DHT11\_hem\_high = hem\_high;//写入湿度整数变量

DHT11\_hem\_low = hem\_low;//写入湿度小数变量

DHT11\_temp\_high = temp\_high;//写入温度整数变量

DHT11\_temp\_low = temp\_low;//写入温度小数变量

}

}

}

u16 DHT11\_GetTemp(void) //获取温度值

{

/\* 这句话的意思是 \*/

/\* [把DHT11\_temp\_high左移8位] 再与[DHT11\_temp\_low]进行 或运算,有1得1 \*/

/\* 因为DHT11只能采集整数数据,所以做了或运算后还是原先采集到的整数数据 \*/

return(DHT11\_temp\_high <<8 | DHT11\_temp\_low);

}

u16 DHT11\_GetHem(void)//获取湿度值

{

/\* [把DHT11\_hem\_high左移8位] 再与[DHT11\_hem\_low]进行 或运算,有1得1 \*/

return(DHT11\_hem\_high <<8 | DHT11\_hem\_low);

}

int get\_shi(void)

{

//uint8\_t sendText\_shi[]={"AT+CIPSEND\r\n#"};

// int m;

// int nn;

DHT11\_ReadData();

sprintf(strHumi,"%d",DHT11\_GetHem()/256); //调用Sprintf函数把DHT11的湿度数据格式化到字符串数组变量strHumi中

m=atoi(strHumi); //调用

strgai[0]='$';

strgai[1]=strHumi[0];

strgai[2]=strHumi[1];

// strgai[3]=strHumi[2];

// strgai[4]=strHumi[3];

// strgai[5]=strHumi[4];

// strgai[6]=strHumi[5];

// strgai[7]=strHumi[6];

// strgai[8]=strHumi[7];

if(m!=nn)

{

USART2\_SendDataString(strgai);

}

nn=m;

return m;

}