目　录

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：如下例子不符合规范。

```
if (!valid_ni(ni))
{
    ... // program code
}
repssn_ind = ssn_data[index].repssn_index;
repssn_ni  = ssn_data[index].ni;
```

应如下书写

```
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]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]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]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：如下例子不符合规范。

```
rect.length = 0;  rect.width = 0;
```

应如下书写

```
rect.length = 0;
rect.width  = 0;
```

[1]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：如下例子不符合规范。

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

应如下书写：

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

[1]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]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]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：如下例子不符合规范。

```
for (...) {
   ... // program code
}

if (...)
   {
   ... // program code
   }
```

```
void example_fun( void )
   {
   ... // program code
   }
```

应如下书写。

```
for (...)
{
   ... // program code
}

if (...)
{
   ... // program code
}

void example_fun( void )
{
   ... // program code
}
```

[1]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。

示例：

(1) 逗号、分号只在后面加空格。

```
int a, b, c;
```

(2)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)
a = b + c;
a *= 2;
a = b ^ 2;
```

(3)"!"、"~"、"++"、"--"、"&"（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
```

(4)"->"、"."前后不加空格。

```
p->id = pid;     // "->"指针前后不加空格
```

(5) 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。

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

½1-1：一行程序以小于80字符为宜，不要写得过长。

2 注释

[1]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。

[1]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。

示例：下面这段头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.
  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. ...
************************************************/
```

[1]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
************************************************************/
```

说明：Description 一项描述本文件的内容、功能、内部各部分之间的关系及本文件与其它文件关系等。History 是修改历史记录列表，每条修改记录应包括修改日期、修改者及修改内容简述。

[1]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
*************************************************/
```

[1]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。

[1]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。

规则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。

[1]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：如下例子不符合规范。

例 1：

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


repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

例 2：

```
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
/* get replicate sub system index and net indicator */
```

应如下书写

```
/* get replicate sub system index and net indicator */
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

[1]2-9：对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。

示例：

```
/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000
```

```
#define MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */
```

[1]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*/
};
```

[1]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;
```

[1]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
}
```

[1]2-13：将注释与其上面的代码用空行隔开。

示例：如下例子，显得代码过于紧凑。

```
/* code one comments */
program code one
/* code two comments */
program code two
```

应如下书写

```
/* code one comments */
program code one

/* code two comments */
program code two
```

[1]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。

[1]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;
...
```

½2-1：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。

½2-2：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。

½2-3：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：如下注释意义不大。

```
/* 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)
```

½2-4：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：参见如下例子。

```
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
```

½2-5：注释格式尽量统一，建议使用“/* …… */”。

½2-6：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

[1]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。

说明：较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成缩写；一些单词有大家公认的缩写。

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` ;

[1]3-2：命名中若使用特殊约定或缩写，则要有注释说明。

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。

[1]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。

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。(即命名规则中没有规定到的地方才可有个人命名风格)。

[1]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。

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。

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。

¹3-5：命名规范必须与所使用的系统风格保持一致，并在同一项目中统一，比如采用UNIX的全小写加下划线的风格或大小写混排的方式，不要使用大小写与下划线混排的方式，用作特殊标识如标识成员变量或全局变量的m_和g_，其后加上大小写混排的方式是允许的。

示例： Add_User 不允许，add_user、AddUser、m_AddUser 允许。

½3-1：除非必要，不要用数字或较奇怪的字符来定义标识符。

示例：如下命名，使人产生疑惑。

```
#define _EXAMPLE_0_TEST_
#define _EXAMPLE_1_TEST_
void set_sls00( BYTE sls );
```

应改为有意义的单词命名

```
#define _EXAMPLE_UNIT_TEST_
#define _EXAMPLE_ASSERT_TEST_
void set_udt_msg_sls( BYTE sls );
```

½3-2：在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。

说明：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。

½3-3：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。

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：

```
int  min_sum;
```

```
int  max_sum;
int  add_user( BYTE *user_name );
int  delete_user( BYTE *user_name );
```

½3-4：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。

4 可读性

[1]4-1：注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

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。

[1]4-2：避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

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
}
```

½4-1：源程序中关系较为紧密的代码应尽可能相邻。

说明：便于程序阅读和查找。

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;
```

½4-2：不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

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

[1]5-1：去掉没必要的公共变量。

说明：公共变量是增大模块间耦合的原因之一，故应减少没必要的公共变量以降低模块间的耦合度。

[1]5-2：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。

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。

[1]5-3：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。

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
```

注：RELATION 为操作关系；System_Init、Input_Rec、Print_Rec、Stat_Score 为四个不同的函数；Student、Score 为两个全局变量；Create 表示创建，Modify 表示修改，Access 表示访问。

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。

[1]5-4：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。

说明：对公共变量赋值时，若有必要应进行合法性检查，以提高代码的可靠性、稳定性。

[1]5-5：防止局部变量与公共变量同名。

说明：若使用了较好的命名规则，那么此问题可自动消除。

[1]5-6：严禁使用未经初始化的变量作为右值。

说明：特别是在 C/C++中引用未经赋值的指针，经常会引起系统崩溃。

½5-1：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。

说明：降低公共变量耦合度。

½5-2：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。

说明：使用标准的数据类型，有利于程序的移植。

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
}
```

½5-3：结构的功能要单一，是针对一种事务的抽象。

说明：设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

示例：如下结构不太清晰、合理。

```
typedef struct STUDENT_STRU
{
    unsigned char name[8]; /* student's name */
    unsigned char age;    /* student's age */
    unsigned char sex;    /* student's sex, as follows */
                        /* 0 - FEMALE; 1 - MALE */
    unsigned char
          teacher_name[8]; /* the student teacher's name */
    unisgned char
          teacher_sex;    /* his teacher sex */
} STUDENT;
```

若改为如下，可能更合理些。

```
typedef struct TEACHER_STRU
{
   unsigned char name[8]; /* teacher name */
   unisgned char sex;    /* teacher sex, as follows */
                      /* 0 - FEMALE; 1 - MALE */
} TEACHER;

typedef struct STUDENT_STRU
{
   unsigned char name[8];    /* student's name */
   unsigned char age;        /* student's age */
   unsigned char sex;        /* student's sex, as follows */
                          /* 0 - FEMALE; 1 - MALE */
   unsigned int  teacher_ind; /* his teacher index */
} STUDENT;
```

½5-4：不要设计面面俱到、非常灵活的数据结构。

说明：面面俱到、灵活的数据结构反而容易引起误解和操作困难。

½5-5：不同结构间的关系不要过于复杂。

说明：若两个结构间关系较复杂、密切，那么应合为一个结构。

示例：如下两个结构的构造不合理。

```
typedef struct PERSON_ONE_STRU
{
   unsigned char name[8];
   unsigned char addr[40];
   unsigned char sex;
   unsigned char city[15];
} PERSON_ONE;

typedef struct PERSON_TWO_STRU
{
   unsigned char name[8];
   unsigned char age;
   unsigned char tel;
} PERSON_TWO;
```

由于两个结构都是描述同一事物的，那么不如合成一个结构。

```
typedef struct PERSON_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
} PERSON;
```

½5-6：结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构，以减少原结构中元素的个数。

说明：增加结构的可理解性、可操作性和可维护性。

示例：假如认为如上的_PERSON 结构元素过多，那么可如下对之划分。

```
typedef struct PERSON_BASE_INFO_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
} PERSON_BASE_INFO;

typedef struct PERSON_ADDRESS_STRU
{
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
} PERSON_ADDRESS;

typedef struct PERSON_STRU
{
    PERSON_BASE_INFO person_base;
    PERSON_ADDRESS person_addr;
} PERSON;
```

½5-7：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。

说明：合理排列结构中元素顺序，可节省空间并增加可理解性。

示例：如下结构中的位域排列，将占较大空间，可读性也稍差。

```
typedef struct EXAMPLE_STRU
{
    unsigned int valid: 1;
    PERSON person;
    unsigned int set_flg: 1;
} EXAMPLE;
```

若改成如下形式，不仅可节省 1 字节空间，可读性也变好了。

```
typedef struct EXAMPLE_STRU
{
    unsigned int valid: 1;
    unsigned int set_flg: 1;
    PERSON person ;
} EXAMPLE;
```

½5-8：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。

½5-9：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。

½5-10：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。

½5-11：对编译系统默认的数据类型转换，也要有充分的认识。

示例：如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

```
char chr;
unsigned short int exam;

chr = -1;
exam = chr; // 编译器不产生告警，此时 exam 为 0xFFFF。
```

½5-12：尽量减少没有必要的数据类型默认转换与强制转换。

½5-13：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。

½5-14：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。

示例：可参考如下方式声明自定义数据类型。

下面的声明可使数据类型的使用简洁、明了。

```
typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned int   DWORD;
```

下面的声明可使数据类型具有更丰富的含义。

```
typedef float DISTANCE;
typedef float SCORE;
```

½5-15：当声明用于分布式环境或不同CPU间通信环境的数据结构时，必须考虑机器的字节顺序、使用的位域及字节对齐等问题 。

说明：比如 Intel CPU 与 68360 CPU，在处理位域及整数时，其在内存存放的“顺序”正好相反。

示例：假如有如下短整数及结构。

```
unsigned short int exam;
typedef struct EXAM_BIT_STRU
{                    /* Intel 68360 */
   unsigned int A1: 1; /* bit  0     7   */
   unsigned int A2: 1; /* bit  1     6   */
```

```
   unsigned int A3: 1; /* bit  2     5   */
} EXAM_BIT;
```

如下是 Intel CPU 生成短整数及位域的方式。

内存： 0 1 2 ... （从低到高，以字节为单位）

exam exam 低字节 exam 高字节

内存： 0 bit 1 bit 2 bit ... （字节的各“位”）

EXAM_BIT A1 A2 A3

如下是 68360 CPU 生成短整数及位域的方式。

内存： 0 1 2 ... （从低到高，以字节为单位）

exam exam 高字节 exam 低字节

内存： 7 bit 6 bit 5 bit ... （字节的各“位”）

EXAM_BIT A1 A2 A3

说明：在对齐方式下，CPU 的运行效率要快得多。

示例：如下图，当一个 long 型数（如图中 long1）在内存中的位置正好与内存的字边界对齐时，CPU 存取这个数只需访问一次内存，而当一个 long 型数（如图中的 long2）在内存中的位置跨越了字边界时，CPU 存取这个数就需要多次访问内存，如 i960cx 访问这样的数需读内存三次（一个 BYTE、一个 SHORT、一个 BYTE，由 CPU 的微代码执行，对软件透明），所有对齐方式下 CPU 的运行效率明显快多了。

```
1       8       16      24      32
-------- -------- -------- --------
| long1 | long1 | long1 | long1 |
-------- -------- -------- --------
|       |       |       | long2 |
-------- -------- -------- ---------
| long2 | long2 | long2 |       |
-------- -------- -------- ---------
| ....
```

6 函数、过程

[1]6-1：对所调用函数的错误返回码要仔细、全面地处理。

[1]6-2：明确函数功能，精确（而不是近似）地实现函数设计。

[1]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）。

说明：编写 C/C++语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

[1]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。

说明：若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量变为不可知状态。

示例：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
    Exam = para;              // 给 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
                             // 用本信号。
    return temp;
}
```

¹6-5：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。

说明：对于模块间接口函数的参数的合法性检查这一问题，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。

½6-1：防止将函数的参数作为工作变量。

说明：将函数的参数作为工作变量，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;
}
```

½6-2：函数的规模尽量限制在200行以内。

说明：不包括注释和空格行。

½6-3：一个函数仅完成一件功能。

½6-4：为简单功能编写函数。

说明：虽然为仅用一两行就可完成的功能去编函数好象没有必要，但用函数可使功能明确化，增加程序可读性，亦可方便维护、测试。

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))

value = MAX (a, b);
```

½6-5：不要设计多用途面面俱到的函数。

说明：多功能集于一身的函数，很可能使函数的理解、测试、维护等变得困难。

½6-6：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。

说明：带有内部“存储器”的函数的功能可能是不可预测的，因为它的输出可能取决于内部存储器（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。在 C/C++ 语言中，函数的 static 局部变量是函数的内部存储器，有可能使函数的功能不可预测，然而，当某函数的返回值为指针类型时，则必须是 STATIC 的局部变量的地址作为返回值，若为 AUTO 类，则返回为错针。

示例：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;
}
```

½6-7：尽量不要编写依赖于其他函数内部实现的函数。

说明：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
```

½6-8：避免设计多参数函数，不使用的参数从接口中去掉。

说明：目的减少函数间接口的复杂度。

½6-9：非调度函数应减少或防止控制参数，尽量只使用数据参数。

说明：本建议目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令，来启动相应的功能实体（即函数或过程），而本身并不完成具体功能。控制参数是指改变函数功能行为的参数，即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合，很可能使函数间的耦合度增大，并使函数的功能不唯一。

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);
}
```

½6-10：检查函数所有参数输入的有效性。

½6-11：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。

½6-12：函数名应准确描述函数的功能。

½6-13：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 ) ;
```

建议6-14：避免使用无意义或含义不清的动词为函数命名。

说明：避免用含义不清的动词如 process、handle 等为函数命名，因为这些动词并没有说明要具体做什么。

建议6-15：函数的返回值要清楚、明了，让使用者不容易忽视错误情况。

说明：函数的每种出错返回值的意义要清晰、明了、准确，防止使用者误用、理解错误或忽视错误返回码。

½6-16：除非必要，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。

½6-17：让函数在调用点显得易懂、容易理解。

½6-18：在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换。

说明：因为数据类型转换或多或少存在危险。

½6-19：避免函数中不必要语句，防止程序中的垃圾代码。

说明：程序中的垃圾代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。

½6-20：防止把没有关联的语句放到一个函数中。

说明：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 */
```

}

½6-21：如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。

说明：若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成一个新的函数。

½6-22：功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在。

说明：模块中函数划分的过多，一般会使函数间的接口变得复杂。所以过小的函数，特别是扇入很低的或功能不明确的函数，不值得单独存在。

½6-23：设计高扇入、合理扇出（小于7）的函数。

说明：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。

½6-24：减少函数本身或函数间的递归调用。

说明：递归调用特别是函数间的递归调用（如 A->B->C->A），影响程序的可理解性；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。

½6-25：仔细分析模块的功能及性能需求，并进一步细分，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。

说明：函数的划分与组织是模块的实现过程中很关键的步骤，如何划分出合理的函数结构，关系到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。

½6-26：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。

½6-27：在多任务操作系统的环境下编程，要注意函数可重入性的构造。

说明：可重入性是指函数可以被多个任务进程调用。在多任务操作系统中，函数是否具有可重入性是非常重要的，因为这是多个进程可以共用此函数的必要条件。另外，编译器是否提供可重入函数库，与它所服务的操作系统有关，只有操作系统是多任务时，编译器才有可能提供可重入函数库。如 DOS 下 BC 和 MSC 等就不具备可重入函数库，因为 DOS 是单用户单任务操作系统。

½6-28：避免使用BOOL参数。

说明：原因有二，其一是 BOOL 参数值无意义，TURE/FALSE 的含义是非常模糊的，在调用时很难知道该参数到底传达的是什么意思；其二是 BOOL 参数值不利于扩充。还有 NULL 也是一个无意义的单词。

½6-29： 对于提供了返回值的函数，在引用时最好使用其返回值。

½6-30：当一个过程（函数）中对较长变量（一般是结构的成员）有较多引用时，可以用一个意义相当的宏代替。

说明：这样可以增加编程效率和程序的可读性。

示例：在某过程中较多引用 TheReceiveBuffer[FirstSocket].byDataPtr，

则可以通过以下宏定义来代替：

```
# define pSOCKDATA TheReceiveBuffer[FirstScoket].byDataPtr
```

7 可测性

[17-1]：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。

[17-2]：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。

[17-3]：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。

[17-4]：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。

[17-5]：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 */
```

[17-6]：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。

[17-7]：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。

[17-8]：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。

[17-9]：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
}
```

[17-10]：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
}
```

[1]7-11：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 );
```

[1]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。

[1]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。

[1]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。

¹7-15：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。

½7-1：在编写代码之前，应预先设计好程序调试与测试的方法和手段，并设计好各种调测开关及相应测试代码如打印函数等。

说明：程序的调试与测试是软件生存周期中很重要的一个阶段，如何对软件进行较全面、高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。

½7-2：调测开关应分为不同级别和类型。

说明：调测开关的设置及分类应从以下几方面考虑：针对模块或系统某部分代码的调测；针对模块或系统某功能的调测；出于某种其它目的，如对性能、容量等的测试。这样做便于软件功能的调测，并且便于模块的单元测试、系统联调等。

½7-3：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;
    }
}
```

8 程序效率

[1]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。

[1]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。

[1]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。

[1]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;
```

[1]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 */
```

½8-1：仔细分析有关算法，并进行优化。

½8-2：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。

½8-3：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。

½8-4：编程时，要随时留心代码效率；优化代码时，要考虑周全。

½8-5：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。

½8-6：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。

½8-7：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。

½8-8：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];
    }
}
```

½8-9：尽量减少循环嵌套层次。

½8-10：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;
   }
```

```
}
```

½8-11：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);
```

应如下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;
```

½8-12：不要一味追求紧凑的代码。

说明：因为紧凑的代码并不代表高效的机器码。

9 质量保证

[1]9-1：在软件设计过程中构筑软件质量。

[1]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。

[1]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。

[1]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。

[1]9-5：过程/函数中分配的内存，在过程/函数退出之前要释放。

[1]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
}
```

[1]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;
   }
}
```

[1]9-8：认真处理程序所能遇到的各种出错情况。

[1]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。

[1]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。

[1]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。

[1]9-12：不能随意改变与其它模块的接口。

[1]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。

[1]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。

[1]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 ));
```

[1]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。

9-1：不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。

9-2：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。

9-3：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。

说明：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。

9-4：精心构造算法，并对其性能、效率进行测试。

9-5：对较关键的算法最好使用其它算法来确认。

9-6：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
}
```

9-7：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;
```

½9-8：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。

½9-9：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。

½9-10：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。

½9-11：对一些具有危险性的操作代码（如写硬盘、删数据等）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。

½9-12：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。

说明：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。

½9-13：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）

10 代码编辑、编译、审查

[1]10-1：打开编译器的所有告警开关对程序进行编译。

[1]10-2：在产品软件（项目组）中，要统一编译开关选项。

[1]10-3：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。

[1]10-4：测试部测试产品之前，应对代码进行抽查及评审。

½10-1：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。

½10-2：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。

½10-3：要小心地使用编辑器提供的块拷贝功能编程。

说明：当某段代码与另一段代码的处理功能相似时，许多开发人员都用编辑器提供的块拷贝功能来完成这段代码的编写。由于程序功能相近，故所使用的变量、采用的表达式等在功能及命名上可能都很相近，所以使用块拷贝时要注意，除了修改相应的程序外，一定要把使用的每个变量仔细查看一遍，以改成正确的。不应指望编译器能查出所有这种错误，比如当使用的是全局变量时，就有可能使某种错误隐藏下来。

½10-4：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。

½10-5：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。

½10-6：使用代码检查工具（如C语言用PC-Lint）对源程序检查。

½10-7：使用软件工具（如 LogiSCOPE）进行代码审查。

11 代码测试、维护

[1]11-1：单元测试要求至少达到语句覆盖。

[1]11-2：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。

[1]11-3：清理、整理或优化后的代码要经过审查及测试。

[1]11-4：代码版本升级要经过严格测试。

[1]11-5：使用工具软件对代码版本进行维护。

[1]11-6：正式版本上软件的任何修改都应有详细的文档记录。

½11-1：发现错误立即修改，并且要记录下来。

½11-2：关键的代码在汇编级跟踪。

½11-3：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。

½11-4：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。

½11-5：仔细测试代码处理数据、变量的边界情况。

½11-6：保留测试信息，以便分析、总结经验及进行更充分的测试。

½11-7：不应通过“试”来解决问题，应寻找问题的根本原因。

½11-8：对自动消失的错误进行分析，搞清楚错误是如何消失的。

½11-9：修改错误不仅要治表，更要治本。

½11-10：测试时应设法使很少发生的事件经常发生。

½11-11：明确模块或函数处理哪些事件，并使它们经常发生。

½11-12： 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。

½11-13：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。

12 宏

[1]12-1：用宏定义表达式时，要使用完备的括号。

示例：如下定义的宏都存在一定的风险。

```
#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))
```

[1]12-2:将宏所定义的多条表达式放在大括号中。

示例：下面的语句只有宏的第一条表达式被执行。为了说明问题，for 语句的书写稍不符规范。

```
#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 );
}
```

[1]12-3:使用宏时，不允许参数发生变化。

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。
```

软件需求说明书的编写提示

1 引言

1.1 编写目的

说明编写这份软件需求说明书的目的，指出预期的读者。

1.2 背景

说明：

a. 待开发的软件系统的名称；

b. 本项目的任务提出者、开发者、用户及实现该软件的计算中心或计算机网络；

c. 该软件系统同其他系统或其他机构的基本的相互来往关系。

1.3 定义

列出本文件中用到的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出用得着的参考资料，如：

a. 本项目的经核准的计划任务书或合同、上级机关的批文；

b. 属于本项目的其他已发表的文件；

c. 本文件中各处引用的文件、资料、包括所要用到的软件开发标准。 列出这些文件资料的标题、文件编号、发表日期和出版单位，说明能够得到这些文件资料的来源。

2 任务概述

2.1 目标

叙述该项软件开发的意图、应用目标、作用范围以及其他应向读者说明的有关该软件开

发的背景材料。解释被开发软件与其他有关软件之间的关系。如果本软件产品是一项独立的软件，而且全部内容自含，则说明这一点。如果所定义的产品是一个更大的系统的一个组成部分，则应说明本产品与该系统中其他各组成部分之间的关系，为此可使用一张方框图来说明该系统的组成和本产品同其他各部分的联系和接口。|

2.2 用户的特点

列出本软件的最终用户的特点，充分说明操作人员、维护人员的教育水平和技术专长，以及本软件的预期使甩频度。这些是软件设计工作的重要约束

2.3 假定和约束

列出进行本软件开发工作的假定和约束，例如经费限制、开发期限等。

3 需求规定

3.1 对功能的规定

用列表的方式（例如 IPO 表即输入、处理、输出表的形式），逐项定量和定性地叙述对软件所提出的功能要求，说明输入什么量、经怎样的处理、得到什么输出，说明软件应支持的终端数和应支持的并行操作的用户数。

3.2 对性能的规定

3.2.1 精度

说明对该软件的输入、输出数据精度的要求，可能包括传输过程中的精度。

3.2.2 时间特性要求

说明对于该软件的时间特性要求，如对：

a. 响应时间；

b. 更新处理时间；

c. 数据的转换和传送时间；

d. 解题时间；等的要求。

3.2.3 灵活性

说明对该软件的灵活性的要求，即当需求发生某些变化时，该软件对这些变化的适应能力，如：

a. 操作方式上的变化；

b. 运行环境的变化；

c. 同其他软件的接口的变化；

d. 精度和有效时限的变化；

e. 计划的变化或改进。

对于为了提供这些灵活性而进行的专门设计的部分应该加以标明。

3.3 输人输出要求

解释各输入输出数据类型，并逐项说明其媒体、格式、数值范围、精度等。对软件的数据输出及必须标明的控制输出量进行解释并举例，包括对硬拷贝报告（正常结果输出、状态输出及异常输出）以及图形或显示报告的描述。

3.4 数据管理能力要求

说明需要管理的文卷和记录的个数、表和文卷的大小规模，要按可预见的增长对数据及其分量的存储要求作出估算。

3.5 故障处理要求

列出可能的软件、硬件故障以及对各项性能而言所产生的后果和对故障处理的要求。

3.6 其他专门要求

如用户单位对安全保密的要求，对使用方便的要求，对可维护性、可补充性、易读性、可靠性、运行环境可转换性的特殊要求等。

4 运行环境规定

4.1 设备

列出运行该软件所需要的硬设备。说明其中的新型设备及其专门功能，包括：

a. 处理器型号及内存容量；

b. 外存容量、联机或脱机、媒体及其存储格式，设备的型号及数量；

c. 输入及输出设备的型号和数量，联机或脱机；

d. 数据通信设备的型号和数量；

e. 功能键及其他专用硬件

4.2 支持软件

列出支持软件,包括要用到的操作系统、编译（或汇编）程序、测试支持软件等。

4.3 接口

说明该软件同其他软件之间的接口、数据通信协议等。

4.4 控制

说明控制该软件的运行的方法和控制信号，并说明这些控制信号的来源。

GB8567——88

可行性研究报告

1 引言

1.1 编写目的

说明编写本可行性研究报告的目的，指出预期的读者。

1.2 背景

说明：

A. 所建议开发的软件系统的名称；

B. 本项目的任务提出者、开发者、用户及实现该软件的计算中心或计算机网络；

C. 该软件系统同其他系统或其他机构的基本的相互来往关系。

1.3 定义

列出本文件中用到的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出用得着的参考资料，如：

1. 本项目的经核准的计划任务书或合同、上级机关的批文；

2. 属于本项目的其他已发表的文件；

3. 本文件中各处引用的文件、资料，包括所需用到的软件开发标准。

列出这些文件资料的标题、文件编号、发表日期和出版单位，说明能够得到这些文件资料的来源。

2 可行性研究的前提

说明对所建议的开发项目进行可行性研究的前提，如要求、目标、假定、限制等。

2.1 要求

说明对所建议开发的软件的基本要求，如：

A. 功能；

B. 性能；

C. 输出如报告、文件或数据，对每项输出要说明其特征，如用途、产生频度、接口以及分发对象；

D. 输入说明系统的输入，包括数据的来源、类型、数量、数据的组织以及提供的频度；

E. 处理流程和数据流程用图表的方式表示出最基本的数据流程和处理流程，并辅之以叙述；

F. 在安全与保密方面的要求；

G. 同本系统相连接的其他系统；

H. 完成期限。

2.2 目标

说明所建议系统的主要开发目标，如：

A. 人力与设备费用的减少；

B. 处理速度的提高；

C. 控制精度或生产能力的提高；

D. 管理信息服务的改进；

E. 自动决策系统的改进；

F. 人员利用率的改进。

2.3 条件、假定和限制

说明对这项开发中给出的条件、假定和所受到的限制，如：

a．所建议系统的运行寿命的最小值；

b．进行系统方案选择比较的时间；

c．经费、投资方面的来源和限制；

d．法律和政策方面的限制；

e．硬件、软件、运行环境和开发环境方面的条件和限制；

f．可利用的信息和资源；

g．系统投入使用的最晚时间。

2.4 进行可行性研究的方法

说明这项可行性研究将是如何进行的，所建议的系统将是如何评价的。摘要说明所使用的基本方法 和策略，如调查、加权、确定模型、建立基准点或仿真等。

2.5 评价尺度

说明对系统进行评价时所使用的主要尺度，如费用的多少、各项功能的优先次序、开发时间的长短 及使用中的难易程度。

3 对现有系统的分析

这里的现有系统是指当前实际使用的系统，这个系统可能是计算机系统，也可能是一个机械系统甚 至是一个人工系统。

分析现有系统的目的是为了进一步阐明建议中的开发新系统或修改现有系统的必要性。

3.1 处理流程和数据流程

说明现有系统的基本的处理流程和数据流程。此流程可用图表即流程图的形式表示，并加以叙述。

3.2 工作负荷

列出现有系统所承担的工作及工作量。

3.3 费用开支

列出由于运行现有系统所引起的费用开支，如人力、设备、空间、支持性服务、材料等项开支以及开 支总额。

3.4 人员

列出为了现有系统的运行和维护所需要的人员的专业技术类别和数量。

3.5 设备

列出现有系统所使用的各种设备。

3.6 局限性

列出本系统的主要的局限性，例如处理时间赶不上需要，响应不及时，数据存储能力不足，处理功能 不够等。并且要说明，为什么对现有系统的改进性维护已经不能解决问题。

4 所建议的系统

本章将用来说明所建议系统的目标和要求将如何被满足。

4.1 对所建议系统的说明

概括地说明所建议系统，并说明在第 2 章中列出的那些要求将如何得到满足，说明所使用的基本方法及理论根据。

4.2 处理流程和数据流程

给出所建议系统的处理流程和数据流程。

4.3 改进之处

按 2.2 条中列出的目标，逐项说明所建议系统相对于现存系统具有的改进。

4.4 影响

说明在建立所建议系统时，预期将带来的影响，包括：

4.4.1 对设备的影响

说明新提出的设备要求及对现存系统中尚可使用的设备须作出的修改。

4.4.2 对软件的影响

说明为了使现存的应用软件和支持软件能够同所建议系统相适应。而需要对这些软件所进行的修改和补充。

4.4.3 对用户单位机构的影响

说明为了建立和运行所建议系统，对用户单位机构、人员的数量和技术水平等方面的全部要求。

4.4.4 对系统运行过程的影响

说明所建议系统对运行过程的影响，如：

a．用户的操作规程；

b．运行中心的操作规程；

c．运行中心与用户之间的关系；

d．源数据的处理；

e．数据进入系统的过程；

f．对数据保存的要求，对数据存储、恢复的处理；

g．输出报告的处理过程、存储媒体和调度方法；

h．系统失效的后果及恢复的处理办法。

4.4.5 对开发的影响

说明对开发的影响，如：

a. 为了支持所建议系统的开发，用户需进行的工作；

b. 为了建立一个数据库所要求的数据资源；

c. 为了开发和测验所建议系统而需要的计算机资源；

d. 所涉及的保密与安全问题。

4.4.6 对地点和设施的影响

说明对建筑物改造的要求及对环境设施的要求。

4.4.7 对经费开支的影响

扼要说明为了所建议系统的开发，设计和维持运行而需要的各项经费开支。

4.5 局限性

说明所建议系统尚存在的局限性以及这些问题未能消除的原因。

4.6 技术条件方面的可行性

本节应说明技术条件方面的可行性，如：

a．在当前的限制条件下，该系统的功能目标能否达到；

b．利用现有的技术，该系统的功能能否实现；

c．对开发人员的数量和质量的要求并说明这些要求能否满足；

d．在规定的期限内，本系统的开发能否完成。

5 可选择的其他系统方案

扼要说明曾考虑过的每一种可选择的系统方案，包括需开发的和可从国内国外直接购买的，如果没有供选择的系统方案可考虑，则说明这一点。

5.1 可选择的系统方案 1

参照第 4 章的提纲，说明可选择的系统方案 1，并说明它未被选中的理由。

5.2 可选择的系统方案 2

按类似 5.1 条的方式说明第 2 个乃至第 n 个可选择的系统方案。

.....

6 投资及效益分析

6.1 支出

对于所选择的方案，说明所需的费用。如果已有一个现存系统，则包括该系统继续运行期间所需的费用。

6.1.1 基本建设投资

包括采购、开发和安装下列各项所需的费用，如：

a. 房屋和设施；

b. ADP 设备；

c. 数据通讯设备；

d. 环境保护设备；

e. 安全与保密设备；

f. ADP 操作系统的和应用的软件；

g. 数据库管理软件。

6.1.2 其他一次性支出

包括下列各项所需的费用，如：

a．研究（需求的研究和设计的研究）；

b．开发计划与测量基准的研究；

c．数据库的建立；

d．ADP 软件的转换；

e．检查费用和技术管理性费用；

f. 培训费、旅差费以及开发安装人员所需要的一次性支出；

g．人员的退休及调动费用等。

6.1.3 非一次性支出

列出在该系统生命期内按月或按季或按年支出的用于运行和维护的费用，包括：

a．设备的租金和维护费用；

b．软件的租金和维护费用；

c．数据通讯方面的租金和维护费用；

d．人员的工资、奖金；

e．房屋、空间的使用开支；

f. 公用设施方面的开支；

g. 保密安全方面的开支；

h. 其他经常性的支出等。

6.2 收益

对于所选择的方案，说明能够带来的收益，这里所说的收益，表现为开支费用的减少或避免、差错的减少、灵活性的增加、动作速度的提高和管理计划方面的改进等，包括；

6.2.1 一次性收益

说明能够用人民币数目表示的一次性收益，可按数据处理、用户、管理和支持等项分类叙述，如：

a. 开支的缩减包括改进了的系统的运行所引起的开支缩减，如资源要求的减少，运行效率的改进，数据进入、存贮和恢复技术的改进，系统性能的可监控，软件的转换和优化，数据压缩技术的采用，处理的集中化／分布化等；

b. 价值的增升包括由于一个应用系统的使用价值的增升所引起的收益，如资源利用的改进，管理和运行效率的改进以及出错率的减少等；

c. 其他如从多余设备出售回收的收入等。

6.2.2 非一次性收益

说明在整个系统生命期内由于运行所建议系统而导致的按月的、按年的能用人民币数目表示的收益，包括开支的减少和避免。

6.2.3 不可定量的收益

逐项列出无法直接用人民币表示的收益，如服务的改进，由操作失误引起的风险的减少，信息掌握情况的改进，组织机构给外界形象的改善等。有些不可捉摸的收益只能大概估计或进行极值估计（按最好和最差情况估计）。

6.3 收益／投资比

求出整个系统生命期的收益／投资比值。

6.4 投资回收周期

求出收益的累计数开始超过支出的累计数的时间。

6.5 敏感性分析

所谓敏感性分析是指一些关键性因素如系统生命期长度、系统的工作负荷量、工作负荷的类型与这些不同类型之间的合理搭配、处理速度要求、设备和软件的配置等变化时，对开支和收益的影响最灵敏的范围的估计。在敏感性分析的基础上做出的选择当然会比单一选择的结果要好一些。

7 社会因素方面的可行性

本章用来说明对社会因素方面的可行性分析的结果，包括：

7.1 法律方面的可行性

法律方面的可行性问题很多，如合同责任、侵犯专利权、侵犯版权等方面的陷井，软件人员通常是不熟悉的，有可能陷入，务必要注意研究。

7.2 使用方面的可行性

例如从用户单位的行政管理、工作制度等方面来看，是否能够使用该软件系统;从用户单位的工作人员的素质来看,是否能满足使用该软件系统的要求等等，都是要考虑的。

8 结论

在进行可行性研究报告的编制时，必须有一个研究的结论。结论可以是：

a．可以立即开始进行；

b．需要推迟到某些条件（例如资金、人力、设备等）落实之后才能开始进行；

c．需要对开发目标进行某些修改之后才能开始进行；

d．不能进行或不必进行（例如因技术不成熟、经济上不合算等）。

概要设计说明书

1 引言

1.1 编写目的

说明编写这份概要设计说明书的目的，指出预期的读者。

1.2 背景

说明：

a. 待开发软件系统的名称；
b. 列出此项目的任务提出者、开发者、用户以及将运行该软件的计算站（中心）。

1.3 定义

列出本文件中用到的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出有关的参考文件，如：

a. 本项目的经核准的计划任务书或合同，上级机关的批文；
b. 属于本项目的其他已发表文件；
c. 本文件中各处引用的文件、资料，包括所要用到的软件开发标准。列出这些文件的标题、文件编号、发表日期和出版单位，说明能够得到这些文件资料的来源。

2 总体设计

2.1 需求规定

说明对本系统的主要的输入输出项目、处理的功能性能要求，详细的说明可参见附录 C。

2.2 运行环境

简要地说明对本系统的运行环境（包括硬件环境和支持环境）的规定，详细说明参见附录 C。

2.3 基本设计概念和处理流程

说明本系统的基本设计概念和处理流程，尽量使用图表的形式。

2.4 结构

用一览表及框图的形式说明本系统的系统元素（各层模块、子程序、公用程序等）的划分，扼要说明每个系统元素的标识符和功能，分层次地给出各元素之间的控制与被控制关系.

2.5 功能器求与程序的关系

本条用一张如下的矩阵图说明各项功能需求的实现同各块程序的分配关系：

	程序 1	程序 2	……	程序 n
功能需求 1	√			
功能需求 2		√		
……				
功能需求 n		√		√

2.6 人工处理过程

说明在本软件系统的工作过程中不得不包含的人工处理过程（如果有的话）。

2.7 尚未问决的问题

说明在概要设计过程中尚未解决而设计者认为在系统完成之前必须解决的各个问题。

3 接口设计

3.1 用户接口

说明将向用户提供的命令和它们的语法结构，以及软件的回答信息。

3.2 外部接口

说明本系统同外界的所有接口的安排包括软件与硬件之间的接口、本系统与各支持软件之间的接口关系。

3.3 内部接口

说明本系统之内的各个系统元素之间的接口的安排。

4 运行设计

4.1 运行模块组合

说明对系统施加不同的外界运行控制时所引起的各种不同的运行模块组合，说明每种运行所历经的内部模块和支持软件。

4.2 运行控制

说明每一种外界的运行控制的方式方法和操作步骤。

4.3 运行时间

说明每种运行模块组合将占用各种资源的时间。

5 系统数据结构设计

5.1 逻辑结构设计要点

给出本系统内所使用的每个数据结构的名称、标识符以及它们之中每个数据项、记录、文卷和系的标识、定义、长度及它们之间的层次的或表格的相互关系。

5.2 物理结构设计要点

给出本系统内所使用的每个数据结构中的每个数据项的存储要求，访问方法、存取单位、存取的物理关系（索引、设备、存储区域）、设计考虑和保密条件。

5.3 数据结构与程序的关系

说明各个数据结构与访问这些数据结构的形式:

6 系统出错处理设计

6.1 出错信息

用一览表的方式说朗每种可能的出错或故障情况出现时，系统输出信息的形式、含意及处理方法。

6.2 补救措施

说明故障出现后可能采取的变通措施，包括：

a. 后备技术说明准备采用的后备技术，当原始系统数据万一丢失时启用的副本的建立和启动的技术，例如周期性地把磁盘信息记录到磁带上去就是对于磁盘媒体的一种后备技术；
b. 降效技术说明准备采用的后备技术，使用另一个效率稍低的系统或方法来求得所需结果的某些部分，例如一个自动系统的降效技术可以是手工操作和数据的人工记录；
c. 恢复及再启动技术说明将使用的恢复再启动技术，使软件从故障点恢复执行或使软件从头开始重新运行的方法。

6.3 系统维护设计

说明为了系统维护的方便而在程序内部设计中作出的安排，包括在程序中专门安排用于系统的检查与维护的检测点和专用模块。 各个程序之间的对应关系，可采用如下的矩阵图的形式；

详细设计说明书

1 引言

1.1 编写目的

说明编写这份详细设计说明书的目的，指出预期的读者。

1.2 背景

说明：

a. 待开发软件系统的名称；

b. 本项目的任务提出者、开发者、用户和运行该程序系统的计算中心。

1.3 定义

列出本文件中用到专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出有关的参考资料，如：

a. 本项目的经核准的计划任务书或合同、上级机关的批文；

b. 属于本项目的其他已发表的文件；

c. 本文件中各处引用到的文件资料，包括所要用到的软件开发标准。列出这些文件的标题、文件编号、发表日期和出版单位，说明能够取得这些文件的来源。

2 程序系统的结构

用一系列图表列出本程序系统内的每个程序（包括每个模块和子程序）的名称、标识符和它们之间 的层次结构关系。

3 程序 1（标识符）设计说明

从本章开始，逐个地给出各个层次中的每个程序的设计考虑。以下给出的提纲是针对一般情况的。对于一个具体的模块，尤其是层次比较低的模块或子程序，其很多条目的内容往往与它所隶属的上一层 模块的对应条目的内容相同，在这种情况下，只要简单地说明这一

点即可。

3.1 程序描述

给出对该程序的简要描述，主要说明安排设计本程序的目的意义，并且，还要说明本程序的特点（如 是常驻内存还是非常驻？是否子程序？是可重人的还是不可重人的？有无覆盖要求？是顺序处理还是并发处理等）。

3.2 功能

说明该程序应具有的功能，可采用 IPO 图（即输入一处理一输出图）的形式。

3.3 性能

说明对该程序的全部性能要求，包括对精度、灵活性和时间特性的要求。

3.4 输人项

给出对每一个输入项的特性，包括名称、标识、数据的类型和格式、数据值的有效范围、输入的方式。数量和频度、输入媒体、输入数据的来源和安全保密条件等等。

3.5 输出项

给出对每一个输出项的特性，包括名称、标识、数据的类型和格式，数据值的有效范围，输出的形式、数量和频度，输出媒体、对输出图形及符号的说明、安全保密条件等等。

3.6 算法

详细说明本程序所选用的算法，具体的计算公式和计算步骤。

3.7 流程逻辑

用图表（例如流程图、判定表等）辅以必要的说明来表示本程序的逻辑流程。

3.8 接口

用图的形式说明本程序所隶属的上一层模块及隶属于本程序的下一层模块、子程序，说明参数赋值和调用方式，说明与本程序相直接关联的数据结构（数据库、数据文卷）。

3.9 存储分配

根据需要，说明本程序的存储分配。

3.10 注释设计

说明准备在本程序中安排的注释，如：

a. 加在模块首部的注释；

b. 加在各分枝点处的注释；

c. 对各变量的功能、范围、缺省条件等所加的注释；

d. 对使用的逻辑所加的注释等等。

3.11 限制条件

说明本程序运行中所受到的限制条件。

3.12 测试计划

说明对本程序进行单体测试的计划，包括对测试的技术要求、输入数据、预期结果、进度安排、人员职责、设备条件驱动程序及桩模块等的规定。

3.13 尚未解决的问题

说明在本程序的设计中尚未解决而设计者认为在软件完成之前应解决的问题。

4 程序 2（标识符）设计说明

用类似 F．3 的方式，说明第 2 个程序乃至第 N 个程序的设计考虑。

.....

数据库设计说明书（GB8567——88）

1 引言

1.1 编写目的

说明编写这份数据库设计说明书的目的，指出预期的读者。

1.2 背景

说明：

a. 说明待开发的数据库的名称和使用此数据库的软件系统的名称；

b. 列出该软件系统开发项目的任务提出者、用户以及将安装该软件和这个数据库的计算站（中心）。

1.3 定义

列出本文件中用到的专门术语的定义、外文首字母组词的原词组。

1.4 参考资料

列出有关的参考资料：

a. 本项目的经核准的计划任务书或合同、上级机关批文；

b. 属于本项目的其他已发表的文件；

c. 本文件中各处引用到的文件资料，包括所要用到的软件开发标准。

列出这些文件的标题、文件编号、发表日期和出版单位，说明能够取得这些文件的来源。

2 外部设计

2.1 标识符和状态

联系用途，详细说明用于唯一地标识该数据库的代码、名称或标识符，附加的描述性信息亦要给出。如果该数据库属于尚在实验中、尚在测试中或是暂时使用的，则要说明这一特点及其有效时间范围。

2.2 使用它的程序

列出将要使用或访问此数据库的所有应用程序，对于这些应用程序的每一个，给出它的名称和版本号。

2.3 约定

陈述一个程序员或一个系统分析员为了能使用此数据库而需要了解的建立标号、标识的约定，例如用于标识数据库的不同版本的约定和用于标识库内各个文卷、、记录、数据项的命名约定等。

2.4 专门指导

向准备从事此数据库的生成、从事此数据库的测试、维护人员提供专门的指导，例如将被送入数据库的数据的格式和标准、送入数据库的操作规程和步骤，用于产生、修改、更新或使用这些数据文卷的操作指导。如果这些指导的内容篇幅很长，列出可参阅的文件资料的名称和章条。

2.5 支持软件

简单介绍同此数据库直接有关的支持软件，如数据库管理系统、存储定位程序和用于装入、生成、修 改、更新数据库的程序等。说明这些软件的名称、版本号和主要功能特性，如所用数据模型的类型、允许 的数据容量等。列出这些支持软件的技术文件的标题、编号及来源。

3 结构设计

3.1 概念结构设计

说明本数据库将反映的现实世界中的实体、属性和它们之间的关系等的原始数据形式，包括各数据项、记录、系、文卷的标识符、定义、类型、度量单位和值域，建立本数据库的每一幅用户视图。

3.2 逻辑结构设计

说明把上述原始数据进行分解、合并后重新组织起来的数据库全局逻辑结构，包括所确定的关键字和属性、重新确定的记录结构和文卷结构、所建立的各个文卷之间的相互关系，形成本数据库的数据库管理员视图。

3.3 物理结构设计

建立系统程序员视图，包括：

a. 数据在内存中的安排，包括对索引区、缓冲区的设计；

b. 所使用的外存设备及外存空间的组织，包括索引区、数据块的组织与划分；

c. 访问数据的方式方法。

4 运用设计

4.1 数据字典设计

对数据库设计中涉及到的各种项目，如数据项、记录、系、文卷、模式、子模式等一般要建立起数据字典，以说明它们的标识符、同义名及有关信息。在本节中要说明对此数据字典设计的基本考虑。

4.2 安全保密设计

说明在数据库的设计中，将如何通过区分不同的访问者、不同的访问类型和不同的数据对象，进行分别对待而获得的数据库安全保密的设计考虑。

数据要求说明书

1 引言

1.1 编写目的

说明编写这份数据要求说明书的目的，指出预期的读者。

1.2 背景

说明：

a. 待开发软件系统的名称；
b. b.列出本项目的任务提出者、开发者、用户以及将运行该项软件的计算站（中心）或计算机网络系统。

1.3 定义

列出本文件中用到的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出有关的参考资料，如：

a. 本项目的经核准的计划任务书或合同，上级机关的批文；
b. 属于本项目的其他已发表文件；
c. 本文件中各处引用的文件、资料，包括所要用到的软件开发标准。列出这些文件的标题、文件编号、发表日期和出版单位。说明能够得到这些文件资料的来源。

2 数据的逻辑描述

对数据进行逻辑描述时可把数据分为动态数据和静态数据。所谓静态数据，指在运行过程中主要作为参考的数据，它们在很长的一段时间内不会变化，一般不随运行而改变。所谓动态数据，包括所有在运行中要发生变化的数据以及在运行中要输入、输出的数据。进行描述时应把各数据元素逻辑地分成若干组，列如函数、源数据或对于其应用更为恰当的逻辑分组。给出每一数据元的名称（包括缩写和代码）、定义（或物理意义）度量单位、值域、格式和类型等有关信息。

2.1 静态数据

列出所有作为控制或参考用的静态数据元素。

2.2 动态输人数据

列出动态输入数据元素（包括在常规运行中或联机操作中要改变的数据）。

2.3 动态输出数据

列出动态输出数据元素（包括在常规运行中或联机操作中要改变的数据）。

2.4 内部生成数据

列出向用户或开发单位中的维护调试人员提供的内部生成数据。

2.5 数据约定

说明对数据要求的制约。逐条列出对进一步扩充或使用方面的考虑而提出的对数据要求的限制（容 量、文卷、记录和数据元的个数的最大值）。对于在设计和开发中确定是临界性的限制更要明确指出。

3 数据的采集

3.1 要求和范围

按数据元的逻辑分组来说明数据采集的要求和范围，指明数据的采集方法，说明数据采集工作的承担者是用户还是开发者。具体的内容包括：

a. 输入数据的来源，例如是单个操作员、数据输入站，专业的数据输入公司或它们的一个分组；
b. 数据输入（指把数据输入处理系统内部）所用的媒体和硬设备。如果只有指定的输入点的输入才是合法的，则必须对此加以说明；
c. 接受者说明输出数据的接受者；
d. 输出数据的形式和设备列出输出数据的形式和硬设备。无论接受者将接收到的数据是打印输出，还是 CRT 上的一组字符、一帧图形，或一声警铃，或向开关线圈提供的一个电脉冲，或常用介质如磁盘、磁带、穿孔卡片等，均应具体说明；
e. 数据值的范围给出每一个数据元的合法值的范围；
f. 量纲给出数字的度量单位、增量的步长、零点的定标等。在数据是非数字量的情况下，

要给出每一种合法值的形式和含意；

g. 更新和处理的频度给出预定的对输入数据的更新和处理的频度。如果数据的输入是随机的，应给出更新处理的频度的平均值，或变化情况的某种其他度量。

3.2 输人的承担者

说明预定的对数据输入工作的承担者。如果输入数据同某一接口软件有关，还应说明该接口软件的来源。

3.3 预处理

对数据的采集和预处理过程提出专门的规定，包括适合应用的数据格式、预定的数据通信媒体和对输入的时间要求等。对于需经模拟转换或数字转换处理的数据量，要给出转换方法和转换因子等有关信息，以便软件系统使用这些数据。

3.4 影响

说明这些数据要求对于设备、软件、用户、开发单位所可能产生的影响，例如要求用户单位增设某个机构等。

项目开发计划（GB856T——88）

1 引言

1.1 编写目的

说明：编写这份软件项目开发计划的目的，并指出预期的读者。

1.2 背景

说明：

a. 待开发的软件系统的名称；
b. 本项目的任务提出者、开发者、用户及实现该软件的计算中心或计算机网络；
c. 该软件系统同其他系统或其他机构的基本的相互来往关系。

1.3 定义

列出本文件中用到的专门术语的定义和外文的首字母组词的原词组。

1.4 参考资料

列出用得着的参考资料，如：

a. 本项目的经核准的计划任务书和合同、上级机关的批文；
b. 属于本项目的其他已发表的文件；
c. 本文件中各处引用的文件、资料，包括所要用到的软件开发标准。列出这些文件资料的标题、文件编号、发表日期和出版单位，说明能够得到这些文件资料的来源。

2 项目概述

2.1 工作内容

简要地说明在本项目的开发中须进行的各项主要工作。

2.2 主要参加人员

扼要说明参加本项目开发的主要人员的情况，包括他们的技术水平。

2.3 产品

2.3.1 程序

列出须移交给用户的程序的名称、所用地编程语言及存储程序的媒体形式，并通过引用相关文件，逐项说明其功能和能力。

2.3.2 文件

列出须移交用户的每种文件的名称及内容要点。

2.3.3 服务

列出需向用户提供的各项服务，如培训安装、维护和运行支持等，应逐项规定开始日期、所提供支持的级别和服务的期限。

2.3.4 非移交的产品

说明开发集体应向本单位交出但不必向用户移交的产品（文件甚至某些程序）。

2.4 验收标准

对于上述这些应交出的产品和服务，逐项说明或引用资料说明验收标准。

2.5 完成项目的最迟期限

2.6 本计划的批准者和批准日期

3 实施计划

3.1 工作任务的分解与人员分工

对于项目开发中需要完成的各项工作，从需求分析、设计、实现、测试直到维护，包括文件的编制、审批、打印、分发工作，用户培训工作，软件安装工作等，按层次进行分解，指明每项任务的负责人和参加人员。

3.2 接口人员

说明负责接口工作的人员及他们的职责，包括：

a. 负责本项目同用户的接口人员；
b. 负责本项目同本单位各管理机构，如合同计划管理部门、财务部门、质量管理部门等的接口人员；
c. 负责本项目同个份合同负责单位的接口人员等。

3.3 进度

对于需求分析、设计、编码实现、测试、移交、培训和安装等工作，给出每项工作任务的预定开始日期、完成日期及所需资源，规定各项工作任务完成的先后顺序以及表征每项工作任务完成的标志性事件（即所谓“里程碑）。

3.4 预算

逐项列出本开发项目所需要的劳务（包括人员的数量和时间）以及经费的预算（包括办公费、差旅费、机时费、资料费、通讯设备和专用设备的租金等）和来源。

3.5 关键问题

逐项列出能够影响整个项目成败的关键问题、技术难点和风险，指出这些问题对项目的影响。

4 支持条件

说明为支持本项目的开发所需要的各种条件和设施。

4.1 计算机系统支持

逐项列出开发中和运行时所需的计算机系统支持，包括计算机、外围设备、通讯设备、模拟器、编译（或汇编）程序、操作系统、数据管理程序包、数据存储能力和测试支持能力等，逐项给出有关到货日期、使用时间的要求。

4.2 需由用户承担的工作

逐项列出需要用户承担的工作和完成期限。包括需由用户提供的条件及提供时间。

4.3 由外单位提供的条件

逐项列出需要外单位分合同承包者承担的工作和完成的时间，包括需要由外单位提供的条件和提供的时间。

5 专题计划要点

说明本项目开发中需制定的各个专题计划（如分合同计划、开发人员培训计划、测试计划、安全保密计划、质量保证计划、配置管理计划、用户培训计划、系统安装计划等）的要点。

开发进度月报（GB8567——88）

l标题

开发中的软件系统的名称和标识符
分项目名称和标识符
分项目负责人签名
本期月报编写人签名
本期月报的编号及所报告的年月

2 工程进度与状态

2.1 进度

列出本月内进行的各项主要活动，并且说明本月内遇到的重要事件，这里所说的重要事件是指一个开发阶段（即软件生存周期内各个阶段中的某一个，例如需求分析阶段）的开始或结束，要说明阶段名称及开始（或结束）的日期。

2.2 状态

说明本月的实际工作进度与计划相比，是提前了、按期完成了、或是推迟了？如果与计划不一致，说 明原因及准备采取的措施。

3 资额耗用与状态

3.1 资额耗用

主要说明本月份内耗用的工时与机时。

3.1.1 工时

分为三类：

a. 管理用工时包括在项目管理（制订计划、布置工作、收集数据、检查汇报工作等）方面耗用的工时；
b. 服务工时包括为支持项目开发所必须的服务工作及非直接的开发工作所耗用的工时；

c. 开发用工时要分各个开发阶段填写。

3.1.2 机时

说明本月内耗用的机时，以小时为单位，说明计算机系统的型号。

3.2 状态

说明本月内实际耗用的资源与计划相比，是超出了、相一致、还是不到计划数？如果与计划不一致，说明原因及准备采取的措施。

4 经费支出与状态

4.1 经费支出

4.1.1 支持性费用

列出本月内支出的支持性费用，一般可按如下七类列出，并给出本月支持费用的总和：

a. 房租或房屋折旧费；
b. 社工资、奖金、补贴；
c. 培训费包括给教师的酬金及教室租金；
d. 资料费包括复印及购买参考资料的费用；
e. 会议费召集有关业务会议的费用；
f. 旅差费；
g. 其他费用。

4.1.2 设备购置费

列出本月内支出的设备购置费，一般可分如下三类：

a. 购买软件的名称与金额；
b. 购买硬设备的名称、型号、数量及金额；
c. 已有硬设备的折旧费。

4.2 状态

说明本月内实际支出的经费与计划相比较，是超过了。相符合、还是不到计划数？如果与计划不一致，说明原因及准备采取的措施。

5 下个月的工作计划

6 建议

本月遇到的重要问题和应引起重视的问题以及因此产生的建议。

模块开发卷宗（GB8567——88）

1 标题

软件系统名称和标识符

模块名称和标识符（如果本卷宗包含多于一个的模块，则用这组模块的功能标识代替模块名）

程序编制员签名

卷宗的修改文本序号

修改完成日期

卷宗序号（说明本卷宗在整个卷宗中的序号）

编排日期（说明整个卷宗最近的一次编排日期）

2 模块开发情况表

3 功能说明

扼要说明本模块（或本组模块）的功能，主要是输入、要求的处理、输出。可以从系统设计说明书中摘录。同时列出在软件需求说明书中对这些功能的说明的章、条、款。

4 设计说明

说明本模块（或本组模块）的设计考虑，包括：

a. 在系统设计说明书中有关对本模块（或本组模块）设计考虑的叙述，包括本模块在软件系统中所处的层次，它同其他模块的接口；
b. 在程序设计说明书中有关对本模块（或本组模块）的设计考虑，包括本模块的算法、处理流程、牵涉到的数据文卷设计限制、驱动方式和出错信息等；
c. 在编制目前已通过全部测试的源代码时实际使用的设计考虑。

5 原代码清单

要给出所产生的本模块（或本组模块）的第一份无语法错的源代码清单以及已通过全部测试的当前有效的源代码清单。

6 测试说明

说明直接要经过本模块（或本组模块）的每一项测试，包括这些测试各自的标识符和编号、进行这些测试的目的、所用的配置和输入、预期的输出及实际的输出。

7 复审的结论

把实际测试的结果，同软件需求说明书、系统设计说明书、程序设计说明书中规定的要求进行比较和给出结论。

项目开发总结报告（GB8567——88）

1 引言

1.1 编写目的

说明编写这份项目开发总结报告的目的，指出预期的阅读范围。

1.2 背景

说明：

a. 本项目的名称和所开发出来的软件系统的名称；

b. 此软件的任务提出者、开发者、用户及安装此软件的计算中心。

1.3 定义

列出本文件中用到的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出要用到的参考资料，如：

a. 本项目的已核准的计划任务书或合同、上级机关的批文；

b. 属于本项目的其他已发表的文件；

c. 本文件中各处所引用的文件、资料，包括所要用到的软件开发标准。列出这些文件的标题、文件编号、发表日期和出版单位，说明能够得到这些文件资料的来源。

2 实际开发结果

2.1 产品

说明最终制成的产品，包括：

a. 程序系统中各个程序的名字，它们之间的层次关系，以千字节为单位的各个程序的程序量、存储媒体的形式和数量；

b. 程序系统共有哪几个版本，各自的版本号及它们之间的区别；

c. 每个文件的名称；

d. 所建立的每个数据库。 如果开发中制订过配置管理计划，要同这个计划相比较。

2.2 主要功能和性能

逐项列出本软件产品所实际具有的主要功能和性能，对照可行性研究报告、项目开发计划、功能需求说明书的有关内容，说明原定的开发目标是达到了、未完全达到、或超过了。

2.3 基本流程

用图给出本程序系统的实际的基本的处理流程。

2.4 进度

列出原定计划进度与实际进度的对比，明确说明，实际进度是提前了、还是延迟了，分析主要原因。

2.5 费用

列出原定计划费用与实际支出费用的对比，包括：

a. 工时，以人月为单位，并按不同级别统计；

b. 计算机的使用时间，区别 CPU 时间及其他设备时间；

c. 物料消耗、出差费等其他支出。

明确说明，经费是超出了、还是节余了，分析其主要原因。

3 开发工作评价

3.1 对生产效率的评价

给出实际生产效率，包括：

a. 程序的平均生产效率，即每人月生产的行数；

b. 文件的平均生产效率，即每人月生产的千字数；

并列出原订计划数作为对比。

3.2 对产品质量的评价

说明在测试中检查出来的程序编制中的错误发生率，即每千条指令（或语句）中的错误指令数（或语句数）。如果开发中制订过质量保证计划或配置管理计划，要同这些计划相比较。

3.3 对技术方法的评价

给出对在开发中所使用的技术、方法、工具、手段的评价。

3.4 出错原因的分析

给出对于开发中出现的错误的原因分析。

4 经验与教训

列出从这项开发工作中所得到的最主要的经验与教训及对今后的项目开发工作的建议。

操作手册（GB8567——88）

1 引言

1.1 编写目的

说明编写这份操作手册的目的，指出预期的读者。

1.2 前景

说明：

a. 这份操作手册所描述的软件系统的名称；
b. 该软件项目的任务提出者、开发者、用户（或首批用户）及安装该软件的计算中心。

1.3 定义

列出本文件中用到的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出有用的参考资料，如：

a. 本项目的经核准的计划任务书或合同、上级机关的批文；
b. 属于本项目的其他已发表的文件；
c. 本文件中各处引用的文件、资料，包括所列出的这些文件资料的标题、文件编号、发表日期和出版单位，说明能够得到这些文件资料的来源。

2 软件征述

2.1 软件的结构

结合软件系统所具有的功能包括输入、处理和输出提供该软件的总体结构图表。

2.2 程序表

列出本系统内每个程序的标识符、编号和助记名。

2.3 文卷表

列出将由本系统引用、建立或更新的每个永久性文卷，说明它们各自的标识符、编号、助记名、存储媒体和存储要求。

3 安装与初始化

一步一步地说明为使用本软件而需要进行的安装与初始化过程，包括程序的存载形式，安装与初始化过程中的全部操作命令，系统对这些命令的反应与答复，表征安装工作完成的测试实例等。如果有的话，还应说明安装过程中所需用到的专用软件。

4 运行说明

所谓一个运行是指提供一个启动控制信息后，直到计算机系统等待另一个启动控制信息时为止的计算机系统执行的全部过程。

4.1 运行表

列出每种可能的运行，摘要说明每个运行的目的，指出每个运行各自所执行的程序。

4.2 运行步骤

说明从一个运行转向另一个运行以完成整个系统运行的步骤。

4.3 运行 1（标识符）说明

把运行 1 的有关信息，以对操作人员为最方便最有用的形式加以说明。

4.3.1 运行控制

列出为本运行所需要”的运行流向控制的说明。

4.3.2 操作信息

给出为操作中心的操作人员和管理人员所需要的信息，如：

a. 运行目的；
b. 操作要求；

c. 启动方法 如应请启动（由所遇到的请求信息启动）、预定时间启动、…，•• 等；
d. 预计的运行时间和解题时间；
e. 操作命令；
f. 与运行有联系的其他事项。

4.3.3 输入—输出文卷

提供被本运行建立、更新或访问的数据文卷的有关信息，如：
a. 文卷的标识符或标号；
b. 记录媒体；
c. 存留的目录表；
d. 文卷的支配如确定保留或废弃的准则、是否要分配给其他接受者、占用硬设备的优先级以及保密控制等有关规定。

4.3.4 输出文段

提供本软件输出的每一一个用于提示、说明、或应答的文段（包括“菜单”）的有关信息，如：
a. 文段的标识符；
b. 输出媒体（屏幕显示、打印、……）；
c. 文字容量；
d. 分发对象；
e. 保密要求。

4.3.5 输出文段的复制

对由计算机产生，而后需用其他方法复制的那些文段提供有关信息，如：
a. 文段的标识符；
b. 复制的技术手段；
c. 纸张或其他媒体的规格；
d. 装订要求；
e. 分发对象；
f. 复制份数。

4.3.6 恢复过程

说明本运行故障后的恢复过程。

4.4 运行 2（标识符）说明

用与本手册 4.3 条相类似的方式介绍另一个运行的有关信息。

5 非常规过程

提供有关应急操作或非常规操作的必要信息，如出错处理操作、向后备系统的切换操作以及其他必须向程序维护人员交待的事项和步骤。

6 远程操作

如果本软件能够通过远程终端控制运行，则在本章说明通过远程终端运行本软件的操作过程。

用户手册（GB8567——88）

1 引言

1.1 编写目的

说明编写这份用户手册的目的，指出预期的读者。

1.2 背景

说明：

a. 这份用户手册所描述的软件系统的名称；
b. 该软件项目的任务提出者、开发者、用户（或首批用户）及安装此软件的计算中心。

1.3 定义

列出本文件中用到的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出有用的参考资料，如：

a. 项目的经核准的计划任务书或合同、上级机关的批文；
b. 属于本项目的其他已发表文件；
c. 本文件中各处引用的文件、资料，包括所要用到的软件开发标准。列出这些文件资料的标题、文件编号、发表日期和出版单位，说明能够取得这些文件资料的来源。

2 用途

2.1 功能

结合本软件的开发目的逐项地说明本软件所具有各项功能以及它们的极限范围。

2.2 性能

2.2.1 精度

逐项说明对各项输入数据的精度要求和本软件输出数据达到的精度，包括传输中的精度要求。

2.2.2 时间特性

定量地说明本软件的时间特性，如响应时间，更新处理时间，数据传输、转换时间，计算时间等。

2.2.3 灵活性

说明本软件所具有的灵活性，即当用户需求（如对操作方式、运行环境、结果精度、时间特性等的要求）有某些变化时，本软件的适应能力。

2.3 安全保密

说明本软件在安全、保密方面的设计考虑和实际达到的能力。

3 运行环境

3.1 硬设备

列出为运行本软件所要求的硬设备的最小配置，如：

a. 处理机的型号、内存容量；
b. 所要求的外存储器、媒体、记录格式、设备的型号和台数、联机／脱机；
c. I／O 设备（联机／脱机？）；
d. 数据传输设备和转换设备的型号、台数。

3.2 支持软件

说明为运行本软件所需要的支持软件，如：

a. 操作系统的名称、版本号；
b. 程序语言的编译／汇编系统的名称和版本号；
c. 数据库管理系统的名称和版本号；
d. 其他支持软件。

3.3 数据结构

列出为支持本软件的运行所需要的数据库或数据文卷。

4 使用过程

在本章，首先用图表的形式说明软件的功能同系统的输入源机构、输出接收机构之间的关系。

4.1 安装与初始化

一步一步地说明为使用本软件而需进行的安装与初始化过程，包括程序的存储形式、安装与初始化过程中的全部操作命令、系统对这些命令的反应与答复。表征安装工作完成的测试实例等。如果有的话，还应说明安装过程中所需用到的专用软件。

4.2 输入

规定输入数据和参量的准备要求。

4.2.1 输入数据的现实背景

说明输入数据的现实背景，主要是

a. 情况——例如人员变动、库存缺货；
b. 情况出现的频度——例如是周期性的、随机的、一项操作状态的函数；
c. 情况来源—一例如人事部门、仓库管理部门；
d. 输入媒体———例如键盘、穿孔卡片、磁带；
e. 限制——出于安全、保密考虑而对访问这些输入数据所加的限制；
f. 质量管理——例如对输入数据合理性的检验以及当输入数据有错误时应采取的措施，如建立出错情况的记录等；
g. 支配——例如如何确定输入数据是保留还是废弃，是否要分配给其他的接受者等。

4.2.2 输入格式

说明对初始输入数据和参量的格式要求，包括语法规则和有关约定，如：

a. 长度—一例如字符数／行，字符数／项；
b. 格式基准——例如以左面的边沿为基准；
c. 标号——例如标记或标识符；
d. 顺序——例如各个数据项的次序及位置；
e. 标点——例如用来表示行、数据组等的开始或结束而使用的空格、斜线、星号、字

符组等。

f. 词汇表——给出允许使用的字符组合的列表，禁止使用＊的字符组合的列表等；
g. 省略和重复——给出用来表示输人元素可省略或重复的表示方式；
h. 控制——给出用来表示输入开始或结束的控制信息。

4.2.3 输入举例

为每个完整的输入形式提供样本，包括：

a. 控制或首部——例如用来表示输入的种类和类型的信息，标识符输入日期，正文起点和对所用编码的规定；
b. 主体——输入数据的主体，包括数据文卷的输入表述部分；
c. 尾部——用来表示输入结束的控制信息，累计字符总数等；
d. 省略——指出哪些输入数据是可省略的；
e. 重复——指出哪些输入数据是重复的。

4.3 输出对每项输出作出说明

4.3.1 输出数据的现实背景

说明输出数据的现实背景，主要是：

a. 使用——这些输出数据是给谁的，用来干什么；
b. 使用频度——例如每周的、定期的或备查阅的；
c. 媒体——打印、CRI 显示、磁带、卡片、磁盘，
d. 质量管理—一例如关于合理性检验、出错纠正的规定；
e. 支配——例如如何确定输出数据是保留还是废弃，是否要分配给其他接受者等。

4.3.2 输出格式

给出对每一类输出信息的解释，主要是：

a. 首部——如输出数据的标识符，输出日期和输出编号；
b. 主体——输出信息的主体，包括分栏标题；
c. 尾部——包括累计总数，结束标记。

4.3.3 输出举例

为每种输出类型提供例子。对例子中的每一项，说明：

a. 定义——每项输出信息的意义和用途；
b. 来源——是从特定的输入中抽出、从数据库文卷中取出、或从软件的计算过程中得到；
c. 特性——输出的值域、计量单位、在什么情况下可缺省等。

4.4 文卷查询

这一条的编写针对具有查询能力的软件，内容包括：同数据库查询有关的初始化、准备、及处理所需 要的详细规定，说明查询的能力、方式，所使用的命令和所要求的控制规定。

4.5 出错处理和恢复

列出由软件产生的出错编码或条件以及应由用户承担的修改纠正工作。指出为了确保再启动和恢复的能力，用户必须遵循的处理过程。

4.6 终端操作

当软件是在多终端系统上工作时，应编写本条，以说明终端的配置安排、连接步释、数据和参数输入步骤以及控制规定. 说明通过终端操作进行查询、检索、修改数据文卷的能力、语言、过程以及辅助性程序等。