正点原子

嵌入式单片机 C 代码规范与风格

作者：正点原子团队

声明

本文档为正点原子嵌入式单片机（包括各类 MCU 芯片）C 语言编程风格与规范说明文档，主要是针对正点原子开发团队的嵌入式软件开发，便于规范公司代码以及注释风格，方便阅读与维护，并养成良好的编程习惯，更好的完成团队开发工作。

正点原子全部嵌入式软件开发人员必须遵守本文档的约束，对于 Linux 部门同事，可以针对性的参考《 【正点原子】嵌入式 Linux C 代码规范与风格.pdf 》，本文档与《 【正点原子】嵌入式 Linux C 代码规范与风格.pdf 》高度一致，但是稍有差异，更符合单片机教学使用，同时也便于单片机学习者向 Linux 开发转变，方便后续发展。

对于各位读者，大家可以用根据自己喜欢编写代码，本文档仅提供参考意见，并方便大家阅读正点原子的教程源码。

第一章 规范说明

正点原子之前并没有对代码规范做强制性要求，导致代码规范性较差，随着团队壮大，有必要对正点原子开发团队的嵌入式C语言代码规范和风格做出约束，方便代码维护和团队协助。

对于新开发的各类单片机产品（开发板、模块、产品等）都必须遵循本文档的约束，规范代码编写和风格，对于老产品，也会逐步进行代码更新和重构，最终达到：统一风格、便于阅读、便于使用、便于移植、便于维护的目的。

代码规范整体需要做到以下几点：

1、简单、明了、清晰：

代码写出来重点是给人看的，因此简单、明了、清晰是第一要务！代码的可阅读性要高于代码的性能(除非你的代码以后不需要维护，那你写成啥样都无所谓)。简单、明了、清晰的代码也利于后期维护，尤其是当你写的代码交给他人去维护的时候，请不要祸害别人！

2、精简

代码越长越难看懂，这个大家应该都深有体会，一个 1000 多行的函数和一个最多 100 行的函数哪个好看？所以尽量将把函数写的精简。而且代码越长越容易出错，没有用的代码，变量等一定要及时的清理掉！功能类似或者重复的代码应尽可能提炼成一个函数。

3、保持第三方代码风格

公司内部代码风格必须做到统一，方便维护，如果有第三方代码（比如 HAL 库、FATFS、emWIN、各种 OS、TGFX、Lwip、各种 Lib 等），出现风格冲突，应用程序还是以公司代码风格编写，与第三方代码的接口程序允许两种风格并存，切记不要去修改第三方代码风格。

4、减少封装

我们做嵌入式教学源码的时候，切忌对第三方代码库进行再封装，不要为了让第三方代码和我们的风格统一，而去修改第三方源码风格，或者重新写一套接口函数，以便和我们代码风格统一。

为了统一而再次封装第三方代码会对我们的教学产生不利影响，会给初学者带来困惑，比如 ST 官方的 Cube 库里面就为了兼容自己的代码风格，对 FreeRTOS 的 API 函数做了封装，结果很多客户就问我们为何 ST 官方所调用的任务创建函数和我们的 FreeRTOS 教程不同！他们之间有什么区别？他们之间没有任何区别，只是 ST 对其做了一个简单的封装，结果给学习者带来了困惑！如果不做这个封装的话虽然影响到了代码风格的统一，但是却给学习者减少了困惑，提高了学习效率，而提高客户的学习效率是我们的第一宗旨！

第二章 排版格式和注释

排版是为了在编写代码的时候按照“一定的规矩”来编写，主要目的是为了是代码清晰、易于阅读。注释顾名思义就为注释自己的代码，以方便他人阅读，尤其是尤其维护人员。优美的排版和言简意赅的注释可以提高阅读者的阅读效率，所以在编写代码之前一定要确定好自己打算采用的排版方式和注释方式。

2.1 排版格式

2.1.1 代码缩进

代码缩进要使用制表符，也就是 TAB 键，一般情况下一个 TAB 为 4 个字符，但是也有 8 个甚至更多的情况，为了统一规范，我们统一规定：TAB 键为 4 个字符。

另外，我们规定：TAB 键缩进用空格替代。因为不同的编译器，对 TAB 键的长度是不完全一样的，所以如果 TAB 键不用空格替代，在不同编译器/阅读软件打开的时候，会有很大的不同，有可能导致看起来异常难看。最简单的方式你可以用 txt 打开.c/.h 来查看代码，就会发现和编译器有所不同，如果你用空格键替 TAB 缩进字符，则无论用什么软件打开看起来都是差异不大的。

对于 MDK 编译器，我们可以通过在： → Configuration → Edit → C/C++ Files 里面勾选 Insert spaces for tabs 来设置每次按 TAB 都是用空格进行缩进填充。

另外，我们还可以通过在： → Configuration → Edit → General Edit Settings 里面勾选 View White Spaces 来查看所有的空格（用‘.’替代空格），勾选之后如下图所示：

```
/**
 * @brief       初始化LED相关IO口, 并使能时钟
 * @param       无
 * @retval      无
 */
void led_init(void)
{
    LED0_GPIO_CLK_ENABLE();  /* LED0时钟使能 */
    LED1_GPIO_CLK_ENABLE();  /* LED1时钟使能 */
    LED2_GPIO_CLK_ENABLE();  /* LED2时钟使能 */

    sys_gpio_set(LED0_GPIO_PORT, LED0_GPIO_PIN,
                 SYS_GPIO_MODE_OUT, SYS_GPIO_OTYPE_PP, SYS_GPIO_SPEED_MID, SYS_GPIO_PUPD_PU);

    sys_gpio_set(LED1_GPIO_PORT, LED1_GPIO_PIN,
                 SYS_GPIO_MODE_OUT, SYS_GPIO_OTYPE_PP, SYS_GPIO_SPEED_MID, SYS_GPIO_PUPD_PU);

    sys_gpio_set(LED2_GPIO_PORT, LED2_GPIO_PIN,
                 SYS_GPIO_MODE_OUT, SYS_GPIO_OTYPE_PP, SYS_GPIO_SPEED_MID, SYS_GPIO_PUPD_PU);

    LED0(1);    /* 关闭 LED0 */
    LED1(1);    /* 关闭 LED1 */
    LED2(1);    /* 关闭 LED2 */
}
```

图 2.1.1.1 TAB 缩进及空格查看

可以看到，上图中所有的代码都用空格缩进了（‘.’），不过有一处地方，TAB 键还是没用空格替代，就是 LED2(1);这行代码之后的注释缩进，是一个 ⟶ ，表示这个 TAB 键不是用空格替代的，这样在不同的编译器/软件打开这段代码的时候，这个非空格替代 TAB 键会有不同的长度，如果不是 4 个字符，就会和其他位置产生差异，因而无法对齐。

所以，我们规定：无论是代码缩进，还是注释缩进，统一用 TAB 键对齐，且 TAB 键设置为用 4 个空格替代缩进。

2.1.2 代码行相关规范

1、 每一行的代码长度限制在 80 列。如果大于 80 列的话就要分成多行编写，并且在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要适当进行缩进与上一行代码对齐，如下所示：

```
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;

if    ((taskno < max_act_task_number)
       && (n7stat_stat_item_valid (stat_item)))
{
            ... /* program code */
}

report_or_not_flag = ((taskno < MAX_ACT_TASK_NUMBER)
                            && (n7stat_stat_item_valid (stat_item))
                            && (act_task_table[taskno].result_data != 0));

n7stat_str_compare((BYTE *) & stat_object,
                          (BYTE *) & (act_task_table[taskno].stat_object),
                          sizeof (_STAT_OBJECT));
```

2、相对独立的程序块之间、变量说明之后，必须加空行。函数之间，必须加空行。

不规范的写法：

```
void funa(...)
{
      if (!valid_ni(ni))
      {
            ... /* program code */
      }
      repssn_ind = ssn_data[index].repssn_index;
      repssn_ni = ssn_data[index].ni;
      while(x == 0)
      {
            ... /* program code */
      }
}
void funb(...)
{
      ... /* program code */
}
```

应改为：

```
void funa(...)
{
      if (!valid_ni(ni))
      {
            ... /* program code */
      }
```

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

        while(x == 0)
        {
            ... /* program code */
        }
}

void funb(...)
{
        ... /* program code */
}
```

3、不要把多个语句放到一行里面，一行只写一条语句，如下所示：

不规范的写法：

```
a = x+y; b = x-y;
```

应改为：

```
a=x+y;
b=x-y;
```

4、 不要在一行里面放置多个赋值语句。

不规范的写法：

```
a = b = 0;
```

应改为：

```
a = 0;
b = 0;
```

5、if、for、do、while、case、swich、default 等语句单独占用一行。且 if、for、do、while 等语句的执行语句部分无论多少都要加括号{}，当且仅当 while 后为空，可以不加{}。

不规范的写法：

```
if (p_gpiox->IDR & pinx)    return 1;    /* pinx 的状态为 1 */
else return 0;                           /* pinx 的状态为 0 */
```

应改为：

```
if (p_gpiox->IDR & pinx)
{
        return 1;     /* pinx 的状态为 1 */
}
else
{
        return 0;     /* pinx 的状态为 0 */
}
```

6、对齐全部用空格，或者说用空格填充的 TAB 键。

用空格填充 TAB 是为了避免垮编译器/不同软件打开时的差异导致代码排版不一的情况，详情见：2.1.1 节说明。

2.1.3 括号与空格

1、括号

代码中用到大括号“{”和“}”的地方，对于**单片机开发**来说，左括号“{”一律新起一行，且位于程序块开始的同一列。对于 Linux 开发，则允许“{”放在行位，参见：《 【正点原子】嵌入式 Linux C 代码规范与风格.pdf 》，以单片机开发为例：

不规范的写法：

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

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

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

应改为：

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

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

void example_fun( void )
{
```

```
    ... /* program code */
}
```

当 while 语句没有代码的时候，可以不用加“{}”，但是，只要 while 有哪怕 1 条语句，就必须加“{}”如下所示：

不规范的写法：

```
while (((RCC->CR & (1 << 17)) == 0) && (retry < 0X7FFF)){ retry++; }
```

应改为：

```
while (((RCC->CR & (1 << 17)) == 0) && (retry < 0X7FFF))
{
    retry++;
}
```

while 后面没有代码的时候，可以不要“{}”

```
while ((QUADSPI->SR & (1 << 1)) == 0);   /* 等待指令发送完成 */
```

2、空格

(1)、在一些关键字后面要添加空格，如：

```
if、swich、case、for、do、while
```

但是不要在 sizeof、typedof、alignof 或者__attribute__这些关键字后面添加空格，因为这些大多数后面都会跟着小括号，因此看起来像个函数，如：

```
s = sizeof(struct file);
```

(2)、如果要定义指针类型，或者函数返回指针类型时，“*”应该放到靠近变量名或者函数名的一侧，而不是类型名，如：

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

(3)、二元或者三元操作符两侧都要加一个空格，例如下面所示操作符：

```
=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

(4)、一元操作符后不要加空格，如：

```
&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

(5)、后缀自加或者自减的一元操作符前后都不加空格，如：

```
++  --
```

(6)、“.”和“->”这两个结构体成员操作符前后不加空格。

(7)、逗号、分号只在后面添加空格，如：

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

(8)、注释符“/*”和“*/”与注释内容之间要添加一个空格。

以上 8 点举例如下：

不规范的写法：

```
void test_error(datax* p,int num,char baseval)
{
    int x1 ,x2;
    int t=0;
    x1=32;
```

```
        x2=23;
        for(t=0;t<=num;t ++ )    /*循环赋值*/
        {
            datax -> buf[ t ]=x1*(x2+t)+baseval;
        }
}
```

应改为：

```
void test_error(datax *p, int num, char baseval)
{
        int x1, x2;
        int t = 0;
        x1 = 32;
        x2 = 23;
        for (t = 0; t <= num; t++)       /*  循环赋值  */
        {
            datax->buf[t] = x1 * (x2 + t) + baseval;
        }
}
```

2.2 注释

2.2.1 注释风格

注释可以让别人一看你的代码就明白其中的含义和用法，但是不要过度注释，不要在注释里解释代码是任何运行的，一般你的注释应该告诉别人代码做了什么，而不是怎么做的，即结果，而非过程！

正点原子代码注释统一采用 doxgen 风格，并统一使用如下注释风格：

具体代码的注释：

```
/*   ……   */
```

函数/文件说明注释格式：

```
/**
*    ……
*    ……
*    ……
*/
```

放弃使用：

```
//    ……
///   ……
```

多行注释，在每一行的开始处都应该放置符号“*”，并且所有的行的“*”要对齐在一列上。

注意，当且仅当屏蔽掉部分功能代码（以便后续调试/修改使用）时，可以使用 // 注释。

如：

代码段 1：

```
if (timeout == 0)
{
    //printf("r fifo time out\r\n");
    SDMMC1->ICR = 0X1FE00FFF;       /* 清除所有标记 */
    sys_intx_enable();              /* 开启总中断 */
    return SD_DATA_TIMEOUT;
}
```

代码段 2：

```
while (res)     /* 读出错 */
{
    if (res != 2)sd_init();     /* 重新初始化 SD 卡 */

    res = sd_read_disk(buff, sector, count);
    //printf("sd rd error:%d\r\n", res);
}
```

以上代码使用 // 屏蔽 printf， 因为这是一行有效代码，可以辅助分析 SD 卡问题，只是这里屏蔽掉，不使用而已。当 SD 卡通信出现问题的时候，我们可以通过取消这个屏蔽，从而通过 printf 输出辅助信息，方便查找问题。

2.2.2 文件信息注释

在文件开始的地方应该对本文件做一个总体的、概括性的注释，比如：文件名（@file）、作者（@author）、当前版本（@version）、修改日期（data）、简要说明（@brief）、版权声明（@copyright）和注意事项（@attention）等，注意事项包括：平台、网站、版本修改等内容。具体风格如下：

```
/**
****************************************************************************
 * @file        delay.c
 * @author      正点原子团队(ALIENTEK)
 * @version     V1.0
 * @date        2020-03-12
 * @brief       串口初始化代码(一般是串口 1)
 * @copyright   Copyright (c) 2020-2032, 广州市星翼电子科技有限公司
****************************************************************************
 * @attention
 *
 * 实验平台:正点原子 STM32H750 开发板
 * 在线视频:www.yuanzige.com
 * 技术论坛:www.openedv.com
 * 公司网址:www.alientek.com
 * 购买地址:openedv.taobao.com
 *
```

```
 * 修改说明
 * V1.0 20200312
 * 第一次发布
 *
********************************************************************************
 */

#include "delay.h"
```

注意，以上注释由于 word 宽度的问题，*数量并不完整，这里规定文件信息注释的间隔*数量个数为 100 个。且文件间隔信息和第一行代码之间必须空一行。

2.2.3 函数的注释

函数注释包括：参数简要说明（@brief）、参数说明（@param）、参量列表说明（@arg）注释说明（@note）和返回值说明（@retval）等。函数注释放在函数实体前面，紧挨函数放置，具体风格如下：

```
/**
 * @brief       GPIO 通用设置
 * @param       p_gpiox: GPIOA~GPIOK, GPIO 指针
 * @param       pinx: 0X0000~0XFFFF, 引脚位置, 每个位代表一个 IO,
 *              第 0 位代表 Px0, 第 1 位代表 Px1, 依次类推.
 *              比如 0X0101, 代表同时设置 Px0 和 Px8.
 *    @arg      SYS_GPIO_PIN0~SYS_GPIO_PIN15, 1<<0 ~ 1<<15
 *
 * @param       mode: 0~3; 模式选择, 设置如下:
 *    @arg      SYS_GPIO_MODE_IN,   0, 输入模式(系统复位默认状态)
 *    @arg      SYS_GPIO_MODE_OUT, 1, 输出模式
 *    @arg      SYS_GPIO_MODE_AF,   2, 复用功能模式
 *    @arg      SYS_GPIO_MODE_AIN, 3, 模拟输入模式
 *
 * @param       otype:0~3; 输出类型选择, 设置如下:
 *    @arg      SYS_GPIO_MODE_IN,   0, 输入模式(系统复位默认状态)
 *    @arg      SYS_GPIO_MODE_OUT, 1, 输出模式
 *    @arg      SYS_GPIO_MODE_AF,   2, 复用功能模式
 *    @arg      SYS_GPIO_MODE_AIN, 3, 模拟输入模式
 *
 * @param       ospeed:0~3; 输出速度, 设置如下:
 *    @arg      SYS_GPIO_SPEED_LOW,   0, 低速
 *    @arg      SYS_GPIO_SPEED_MID,   1, 中速
 *    @arg      SYS_GPIO_SPEED_FAST, 2, 快速
 *    @arg      SYS_GPIO_SPEED_HIGH, 3, 高速
 *
 * @param       pupd:0~3: 上下拉设置, 设置如下:
 *    @arg      SYS_GPIO_PUPD_NONE, 0, 不带上下拉
```

```
 *     @arg        SYS_GPIO_PUPD_PU,    1, 上拉
 *     @arg        SYS_GPIO_PUPD_PD,    2, 下拉
 *     @arg        SYS_GPIO_PUPD_RES,   3, 保留
 *
 * @note:        注意: 在输入模式(普通输入/模拟输入)下, OTYPE 和 OSPEED 参数无效!!
 * @retval       无
 */
void sys_gpio_set(GPIO_TypeDef *p_gpiox, uint16_t pinx, uint32_t mode, uint32_t otype,
                  uint32_t ospeed, uint32_t pupd)
{
    uint32_t pinpos = 0, pos = 0, curpin = 0;
    ……   /* 省略代码 */
}
```

Doxgen 的常用注释命令如下：

@exception <exception-object> {exception description} 对一个异常对象进行注释。

@warning {warning message } 一些需要注意的事情

@todo { things to be done } 对将要做的事情进行注释，链接到所有 TODO 汇总的 TODO 列表

@bug 缺陷，链接到所有缺陷汇总的缺陷列表

@see {comment with reference to other items } 一段包含其他部分引用的注释，中间包含对其他代码项的名称，自动产生对其的引用链接。

@relates <name> 通常用做把非成员函数的注释文档包含在类的说明文档中。

@since {text} 通常用来说明从什么版本、时间写此部分代码。

@deprecated

@pre { description of the precondition } 用来说明代码项的前提条件。

@post { description of the postcondition } 用来说明代码项之后的使用条件。

@code 在注释中开始说明一段代码，直到@endcode 命令。

@endcode 注释中代码段的结束。

@code .. @endcode 包含一段代码

@addtogroup 添加到一个组。

@brief 概要信息

@deprecated 已废弃函数

@details 详细描述

@note 开始一个段落，用来描述一些注意事项

@par 开始一个段落，段落名称描述由你自己指定

@param 标记一个参数的意义

@fn 函数说明

@ingroup 加入到一个组

@return 描述返回意义

@retval 描述返回值意义

@include 包含文件

@var、@enum、@struct、@class 对变量、枚举、结构体、类等进行标注

@arg 对函数参数进行列举说明

@attention 注意事项

2.2.4 代码注释

对于单行代码、变量、枚举、宏定义、函数块等，在有必要的情况下，都可以添加注释说明，注释采用：/* …… */（/*后和*/前有空格！）风格，当注释比较长时，放到被注释的代码前面，当比较短时，可以放到代码后面，以减少文件长度。

另外，特别注意的是代码注释要尽可能对齐，这样看起来更简洁。实例如下：

```
void sys_stm32_clock_init(uint32_t plln, uint32_t pllm, uint32_t pllp, uint32_t pllq)
{
    RCC->CR = 0x00000001;               /* 设置 HISON, 开启 RC 振荡，其他位全清零 */
    RCC->CFGR = 0x00000000;             /* CFGR 清零 */
    RCC->D1CFGR = 0x00000000;           /* D1CFGR 清零 */
    RCC->D2CFGR = 0x00000000;           /* D2CFGR 清零 */
    RCC->D3CFGR = 0x00000000;           /* D3CFGR 清零 */
    RCC->PLLCKSELR = 0x00000000;        /* PLLCKSELR 清零 */
    RCC->PLLCFGR = 0x00000000;          /* PLLCFGR 清零 */
    RCC->CIER = 0x00000000;             /* CIER 清零, 禁止所有 RCC 相关中断 */

    /* AXI_TARG7_FN_MOD 寄存器, 在 stm32h750xx.h 里无定义, 所以, 只能用直接 */
    /* 操作地址的方式, 来修改, 在 <<参考手册>>第 115 页, AXI_TARGx_FN_MOD */
    *(( volatile uint32_t *)0x51008108) = 0x00000001; /* 设 AXI SRAM 矩阵读能力为 1 */

    sys_clock_set(plln, pllm, pllp, pllq);      /* 设置时钟 */
    sys_qspi_enable_memmapmode();               /* 使能 QSPI 内存映射模式 */
    sys_cache_enable();                         /* 使能 L1 Cache */

    /* 配置中断向量偏移 */
#ifdef  VECT_TAB_RAM
    sys_nvic_set_vector_table(D1_AXISRAM_BASE, 0x0);
#else
```

```
        sys_nvic_set_vector_table(FLASH_BANK1_BASE, 0x0);
    #endif
}
```

第三章 标识符命名

3.1 命名规则

C 语言中的命名有多种风格，有 unix 风格的、有 windows 风格的、还有匈牙利命名法的等等。我们使用大部分软件工程师常用的命名方式（unix 风格）：单词用小写，然后每个单词用下划线“_”连接在一起，比如：read_adc1_value()，因此在函数和变量的命名上就要使用此种方法，这也是 Linux 内核里面所使用的命名方法。

注意事项：

1、命名一定要清晰！清晰是首位，要使用完整的单词或者大家都知道的缩写，让别人一读就懂，避免不必要的误会，比如：

```
int book_number;
int number_of_beautiful_gril;
```

2、除了常用的缩写以外，不要使用单词缩写，更不要用汉语拼音!!!

3、具有互斥意义的变量或者动作相反的函数应该是用互斥词组命名，如：

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	copy/paste	up/down

4、如果是移植的其它的代码，比如驱动，命名风格应该和原风格一致。

5、不要使用单字节命名变量，但是允许使用 i，j，k 这样的作为局部循环变量。

3.2 文件命名

文件统一采用小写命名。

3.3 变量命名

变量名一定要有意义，并且意义准确，单词都采用小写，用下划线“_”连接。比如表示图书的数量的变量，就可以使用如下命名：

```
int number_of_book;
```

不要采用匈牙利命名法，尽量避免使用全局变量。

另外，变量命名可以使用 g_、p_开头，来表示该变量是一个：全局变量、指针等，可以提高代码的可读性。

我们规定数据类型简写：u8、u16、u32 等不再使用，统一改成更为规范的简写：

```
int8_t      /* 8    位有符号 char 型 */
int16_t     /* 16   位有符号 short 型 */
int32_t     /* 32   位有符号 int 型 */
int64_t     /* 64   位有符号 long long 型（对 stm32 来说） */
uint8_t     /* 8    位无符号 unsigned char 型 */
uint16_t    /* 16   位无符号 unsigned short 型 */
uint32_t    /* 32   位无符号 unsigned int 型 */
uint64_t    /* 64   位无符号 unsigned long long 型（对 stm32 来说） */
```

3.4 函数命名

和变量命名一样。

3.5 宏命名

对于数值等常量宏定义的命名，如非特殊情况，一般使用大写，单词之间使用下划线“_”连接在一起，比如：

```
#define PI_ROUNDED          3.14
```

第四章 函数

函数要简短而且漂亮、并且只能完成一件事，函数的本地变量数量最好不超过 5-10 个，否则函数就有问题，需要重新构思函数，将其分为更小的函数，函数要注意的事项如下：

1、一个函数只能完成一个功能

如果一个函数实现多个功能的话将会给开发、使用和维护带来很大的困难。因此，在跟函数无关或者关联很弱的代码不要放到函数里面，否则的话会导致函数职责不明确，难以理解和修改。

2、重复代码提炼成函数

重复的代码给人的直观感受就是啰嗦，明明说一遍别人就能记住的事情，非要说好几遍！因此一定要消除重复代码，将其提炼成函数。

3、不同函数用空行隔开

不同的函数使用空行隔开，如果函数需要导出的话，它的 EXPORT*宏应该紧贴在他的结束大括号下，比如：

```
int system_is_up(void)
{
    return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

4、函数集中退出方法

我们在学习 C 语言的时候都会听到或者看到这种说法：少用、最好不要用 goto 语句，但是 linux 源码中确大量的使用到了 goto 语句，linux 源码使用 goto 语句来实现函数退出。当一个函数从多个位置推出，并且需要做一些清理的常见操作的时候，goto 语句就很方便，如果不需要清理操作的话就直接使用 return 即可，如下所示：

```
int fun(int a)
{
    int result = 0;
    char *buffer;

    buffer = kmalloc(SIZE, GFP_KERNEL);
    if (!buffer)
    {
        return -ENOMEM;
    }

    if (condition1)
    {
        while (loop1)
        {
            ...
```

```
            }
            result = 1;
            goto out_buffer;
        }
        ...
out_buffer:
        kfree(buffer);
        return result;
}
```

5、函数嵌套不能过深，新增函数最好不超过 4 层

函数嵌套深度指的是函数中的代码控制块(例如：if、for、while、switch 等)之间互相包含的深度，嵌套会增加阅读代码时候的脑力，嵌套太深非常不利于阅读！

6、对函数的参数做合法性检查

函数要对其参数做合法性的检查，比如参数如果有指针类型数据的话如果不做检查，当传入野指针的话就会出错。比如参数的范围不做检查的话可能会传递进来一个超出范围的参数值，导致函数计算溢出等。因此函数必须对参数做合法性检查，比如 STM32 的官方库函数就会对函数的参数做合法性的检查。

7、对函数的错误返回要做全面的处理

一个函数一般都会提供一些指示错误发生的方法，一般都用返回值来表示，因此我们必须对这些错误标识做处理。

8、源文件范围内定义和声明的函数，除非外部可见，否则都应该用 static 函数

如果一个函数只在同一个文件的其它地方调用，那么就应该用 static，static 确保这个函数只在声明它的文件是可见的，这样可以避免和其它库中相同标识符的函数或变量发生混淆。

第五章 变量

1、一个变量只能有一个功能，不能把一个变量当作多用途

一个变量只能有一个特定功能，不能把一个变量作为多用途使用，即一个变量取值不同的时候其代表的含义也不同，如：

```
int time;
time = 200;          /* 表示时间 */
time = getvalue();   /* 用作返回值 */
```

上述代码中变量 time 用作时间，但是也用作了函数 getvalue 的返回值。应该改为如下：

```
int time,ret;
time = 200;
ret = getvalue();
```

上述代码中变量 time 用作时间，但是也用作了函数 getvalue 的返回值。应该改为如下：

2、不用或者少用全局变量

单个文件内可以使用 static 修饰的全局变量，这可以为文件的私有变量，全局变量应该是模块的私有数据，不能作用对外的接口，使用 static 类型的定义，可以防止外部文件对本文件的全局变量的非正常访问。直接访问其它模块的私有数据，会增强模块之间的耦合关系。

3、防止局部变量和全局变量重名

局部变量和全局变量重名会容易使人误解！

4、严禁使用未经初始化的变量作为右值

如果使用变量作为右值的话，在使用前一定要初始化变量，禁止使用未经初始化的变量作为右值，而且在首次使用前初始化变量的地方离使用的地方越近越好！未初始化变量是 C 和 C++最常见的错误源，因此在变量首次使用前一定要确保正确初始化，如：

```
/*  不可取的初始化：无意义 */
int num = 2;
if(a)
{
    num = 3;
}
else
{
    num=4
}

/* 不可取的初始化：初始化和声明分离 */
int num;
if(a)
{
    num = 3;
}
else
```

```
{
    num=4
}

/* 较好的初始化：使用默认有意义的初始化 */
int num = 3;
if(a)
{
    num = 4;
}

/* 较好的初始化：?:减少数据流和控制流的混合 */
int num=a?4:3;
```

5、明确全局变量的初始化顺序

系统启动阶段，使用全局变量前，要考虑到全局变量该在什么地方初始化！使用全局变量和初始化全局变量之间的时序关系一定要分析清楚！

6、尽量减少不必要的数据类型转换

进行数据类型转换的时候，其数据的意义、转换后的取值等都有可能发生变化，因此尽量减少不必要的数据类型转换。

第六章 宏和常量

1、宏命名

常量宏和枚举标签，一般采用大写定义，特殊情况下可以用小写，如：

```
#define PI                                  3.1415962

#define USART_UX                            USART
#define USART_UX_IRQn                       USART1_IRQn
#define USART_UX_IRQHandler                 USART1_IRQHandler

typedef enum
{
  HAL_CRC_STATE_RESET      = 0x00U,    /*!< CRC not yet initialized or disabled */
  HAL_CRC_STATE_READY      = 0x01U,    /*!< CRC initialized and ready for use */
  HAL_CRC_STATE_BUSY       = 0x02U,    /*!< CRC internal process is ongoing */
  HAL_CRC_STATE_TIMEOUT    = 0x03U,    /*!< CRC timeout state */
  HAL_CRC_STATE_ERROR      = 0x04U     /*!<CRC error state */
} HAL_CRC_StateTypeDef;
```

在定义几个变量的常量时，最好使用枚举。

2、函数宏的命名

宏的名字一般用大写，但是形如函数的宏，其名字可以用大写 / 小写，可以根据实际需要选择使用。如果能写成内联函数的就不要写成像函数的宏。

另外，函数宏定义，不管有几个参数，一定要使用 do{ }while(0)的形式，避免出错/报错（具体原因参见：https://blog.csdn.net/xiaoyilong2007101095/article/details/77067686 ），事实上，对于大于等于 2 条语句的宏定义，也都建议用 do{ }while(0)的方式，如下所示：

```
#define USART_UX_CLK_ENABLE()   do{ RCC->APB2ENR |= 1 << 4; } while(0)

#define macrofun(a, b, c)           \
    do {                            \
            if (a == 5)             \
            {                       \
                do_this(b, c);      \
            }                       \
    } while (0)

#define DHT11_IO_IN()       \
    do{                     \
        GPIOC->MODER &= ~(3 << (13 * 2));       \
        GPIOC->MODER |= 0 << 13 * 2;            \
    } while(0)
```

3、使用宏的注意事项

1）避免影响控制流程的宏，如下：

```
#define FOO(x)                          \
        do {                            \
                if (blah(x) < 0)        \
                        return -EBUGGERED;  \
        } while (0)
```

上述代码中就很不好，它看起来像个函数，但是却能导致“调用”它的函数退出。

2）作为左值的带参数的宏：FOO(x) = y，如果有人把 FOO 变成一个内联函数的话，这种用法就会出错了。

3）忘记优先级：使用表达式定义常量的宏必须将表达式置于一对小括号之内，如：

```
#define CONSTANT    0x4000
#define CONSTEXP    (CONSTANT | 3)
```

4）使用宏时，不允许参数发生变化，如：

```
#define SQUARE( a )   do{ ((a) * (a)) } while(0)

int a = 5;
int b;
b = SQUARE( a++ );    /* 结果：a = 7，即执行了两次增 1 */
```

应改为：

```
b = SQUARE( a );
a++;                  /* 结果：a = 6，即只执行了一次增 1 */
```

4、将宏所定义的多条表达式放在 do{ }while(0)中。

如果有多条语句的话，最好的写法就是写成 do while(0)的方式，如下所示示例：

```
#define FOO(x)              \
        printf("arg is %d\n", x);  \
        do_something_useful(x);
```

为了说明这个问题，下面以 for 语句为例：

```
for (blah = 1; blah < 10; blah++)
        FOO(blah);
```

为了修改上面的 for 循环的错误，可以通过大括号来解决，如下：

```
#define FOO(x)
do{  \
        printf("arg is %s\n", x);       \
        do_something_useful(x);     \
}while(0)
```

5、常量建议使用 const 定义来代替宏

宏定义常量是没有数据类型的，而 const 定义是有数据类型的。

6、宏定义函数/超过 1 行代码的宏定义，一般全部用 do{…}while(0)结构，避免使用出错

对于宏定义函数 或者 超过 1 行代码的宏定义，建议全部采用：do{…}while(0)结构，因为这样的结构，可以保证不管如何使用宏定义，都会以你期望的方式运行，否则可能会运行出错。

举个简单的例子：

```
#define foo(x)    fun1(x);    fun2(x);
```

当进行如下调用时：

```
if(!x)
    foo(x);
```

展开后，变成：

```
if(!x)
    fun1 (x);
fun2(x);
```

这明显出错了，fun2 不管 x 的值是不是 0，都会执行。

那么宏定义加括号会不会运行正确呢？

```
#define foo(x)    {fun1(x);    fun2(x);}
```

这个时候，上面的 if 判断就是正确的了，但是使用如下调用方式的时候又有问题：

```
if(!x)
    foo(x) ;
else
    bin(x);
```

展开后，变成：

```
if(!x)
{
    fun1(x) ;
    fun2(x) ;
};
else
    bin(x);
```

标红部分会导致编译报错。

最后，当我们使用 do{…}while(0)结构来定义宏的时候，如下：

```
#define foo(x)    do{ fun1(x);    fun2(x); }while(0)
```

代入以上 2 个出错情况，都不会有问题了，执行正确，且不会报错。

因此，我们建议对于宏定义函数/超过 1 行代码的宏定义，全部使用 do{…}while(0)结构。