正点原子

正点原子 Linux 团队编著

嵌入式 Linux C 代码规范和风格

作者：左忠凯

声明

本文档为作者在嵌入式和嵌入式 linux C 语言的学习和工作中所总结的代码规范，是作者从 STM32 单片机开发向 Linux C 开发的时候为了摆脱遗留的编码规范陋习而编写的。因此，本文档主要面向 Linux C，会根据实际情况兼容单片机的开发。文档主要以 Linux 源码下的 CodingStyle 文档为蓝本而编写，本文档主要是为了解决作者的实际需求，并不能照顾到所有的开发人员，因此编码规范可能不适合某些程序员朋友，因此，不喜勿喷!!右上角点“X”关闭文档即可。本文档仅作为参考资料，不应该作为强制性要求！文档参考资料如下：

1、Linux 源码下的《CodingStyle》文档。

2、《代码整洁之道》。

3、《GNU 编码规范》。

4、《华为技术有限公司 c 语言编程规范》。

套用 CodingStyle 下的一句话：

“代码风格是因人而异的，而且我不愿意把自己的观点强加给任何人，但这就像我去做任何事情都必须遵循的原则那样，我也希望在绝大多数事上保持这种的态度。”

第一章 规范说明

之所以会写这份文档是要下定决心修改自己那写的跟一坨屎一样的垃圾代码规范！相信正在阅读本文档的读者出发点也是如此（可能你的代码规范还没有像一坨屎那么严重）。因为作者工作内容的原因(做单片机开发板的)，此前没有代码规范化的思维，变量，函数的命名随心所欲，大小写混用；代码注释“//”和“/* */”混用等等很多陋习，这样的陋习写出的例程供阅读者学习也会带坏人家哒，所以痛定思痛，一定要改掉这些陋习。在这个看脸的时代，优美的代码风格让人看着都赏心悦目。

当然，代码规范和风格不是用来赏心悦目的，要不然就是花瓶了，代码规范和风格的目的是要编写出简洁明了、可维护、可测试、可靠、可移植、高效的代码。

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、尽量与原有代码风格保持一致

一个公司内部的代码风格一般都是统一的，但是如果你跳槽到其他公司去，或者有时候因为业务原因需要维护别的公司的代码，此时代码风格出现冲突的话尽可能使用现在维护的代码风格。

4、减少封装

此规则仅适用于作者所在公司，作者公司是做开发板的，所写的所有例程代码都需要开源给客户阅读学习。在编写的过程中会遇到使用很多第三方库，比如 ST 的 HAL 库、NXP 的 FSL 库，LWIP 协议栈、UCOS 操作系统、FreeRTOS 操作系统等等，这些第三方库的编码规范和风格各不相同。有人为了统一自己的编码风格会对这些第三方库的 API 函数做封装，如果是做产品的话这样做无可厚非，毕竟为了代码风格的统一，但是作为做开发板的，尽量不要对第三方库做封装！因为每一次封装都会将原有 API 函数的本意遮蔽，读者第一眼看懂这个 API 函数的具体函数，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公司的第一宗旨！

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，不要使用空格键缩进！一般情况下一个 TAB 为 4 个字符，但是 Linux 建议 TAB 键为 8 个字符，因为 8 个字符缩进比较多，因为有利于长时间阅读代码，可以很清晰的分辨出多级嵌套，但是 8 个字符太多，代码向右移动的太远了，这样的话如果屏幕横向分辨率小的话每行编写的代码就会少，尤其是当一个代码块有多级缩进的时候，Linux 建议你应该修改你的代码。这里我设置的 TAB 键为 4 个字符，因为在我阅读 Linux 内核源码的时候发现大部分都是 4 个字符。

在 switch 语句中，“swich”和“case”标签应该对齐处于同一列，不需要缩进 case 标签，如下所示：

```
switch (suffix) {
case 'G':
case 'g':
        mem <<= 30;
        break;
case 'M':
case 'm':
        mem <<= 20;
        break;
case 'K':
case 'k':
        mem <<= 10;
        /* fall through */
default:
        break;
}
```

2.1.3 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(其实当前高分辨屏幕已经很常见了，基本都是 1080P 起，甚至 4K，所以可以设置更大值)，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，一般进行一级缩进即可，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
}
```

2、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;
```

应该为：

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

3、 不要在一行里面防止多个赋值语句。

4、if、for、do、while、case、swich、default 等语句单独占用一行。

2.1.2 括号与空格

1、括号

代码中用到大括号“{”和“}”的地方，应该把起始大括号“{”放到行尾，把结束大括号“}”放到行首，如下所示：

```
if (x is true) {
    we do y
}
```

上面大括号“{”和“}”的用法适用于所有的非函数程序块，如 if、switch、for、do、while 等，比如：

```
switch (action) {
case KOBJ_ADD:
    return "add";
case KOBJ_REMOVE:
    return "remove";
case KOBJ_CHANGE:
    return "change";
default:
    return NULL;
}
```

这里要注意!!! 函数的起始大括号要放置到下一行的开头!! 如下所示：

```
int function(int x)
{
    body of function
}
```

结束的大括号“}”要独自占用一行，除非后面跟着同一个语句的其它剩余部分，比如 do 语句中的“while”或者 if 语句中的“else”，如下代码所示：

```
do {
    body of do-loop
} while (condition);
```

和

```
if (x == y) {
        ..
} else if (x > y) {
        ...
} else {
        ....
}
```

当只有一个单独的语句的时候就可以不必要加大括号了，比如：

```
if (condition)
        action();
```

和

```
if (condition)
        do_this();
else
        do_that();
```

在上面的 if else 条件语句中，所有的分支都只有一行语句，所以可以都不加大括号。但是如果在条件语句中任何一个分支有大于一行语句的时候都必须加上大括号，如下所示代码：

```
if (condition) {
        do_this();
        do_that();
} else {
        otherwise();
}
```

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。

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！注释要放到函数的头部，尽量不要在函数体里面放置注释，注释的风格应该选择：

```
/*  ……  */
```

而非：

```
//……
```

对于多行注释，应该选择如下风格：

```
/*
* This is the preferred style for multi-line
* comments in the Linux kernel source code.
* Please use it consistently.
*
* Description:  A column of asterisks on the left side,
* with beginning and ending almost-blank lines.
*/
```

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。

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：版权声明、版本号、作者、文件简介、修改日志等，如下所示的注释：

```
/************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2018. All rights reserved.
File name:      // 文件名
Author:         //作者
Version:        //版本号
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
Log:            // 修改日志，包括修改内容，日期，修改人等
************************************************/
```

2.2.3 函数的注释

函数需要注释其作用，参数的含义以及返回值的含义，注释可以放在函数声明的地方，也可以放在函数头，如下：

```
/*
*@ Description:  函数描述，描述本函数的基本功能
```

```
 * @param 1 - 参数 1.
 * @param 2 - 参数 2
 * @return - 返回值
 */
```

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。因为我们是编写 Linux 代码的，所以要使用 unix 风格，而 Linux 属于类 unix 系统。unix 命名风格是单词用小写，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。

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，建议使用大写，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();   //ret 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();
```

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

用于定义常量的宏的名字及枚举里的标签需要大写，如：

```
#define CONSTANT 0x12345
```

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，其名字可以用小写，如果能写成内联函数的就不要写成像函数的宏。如下所示：

```
#define macrofun(a, b, c)       \
        do {                            \
            if (a == 5)             \
                do_this(b, c);          \
        } 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、将宏所定义的多条表达式放在大括号中。

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) {                        \
        printf("arg is %s\n", x);       \
        do_something_useful(x);         \
}
```

5、常量建议使用 const 定义来代替宏