μFun C 编程规范

版本: V 0.1　　日期: 2015 年 1 月 10 日

编写: 侯 名　　QQ: 401330597　　Email: omonkerman@qq.com

审校: μFun 项目组　　QQ 群: 4521393　　Github: 待定

版本变更履历

版本	变更内容	作者	时间

目录

第 0 章 背景

μFun 公益开发板项目终于要进入软件开发过程啦. ☺

任何软件项目在随着时间的推移，代码量和功能增加的同时，摆在面前的问题都是如何以系统性、规范化、可定量的过程化方法去开发和维护软件，以及如何把经过时间考验而证明正确的管理技术和当前能够得到的最好的技术方法结合起来. 这是自上世纪 60 年代**”软件危机”**到目前的**”软件工程”**一直面临以及亟待解决的问题. **其实本质就是如何写出正确、可理解、可验证，可维护的软件。**

C 语言是绝大部分嵌入式项目的主要编程语言. 每个 C 程序员都知道用 C 语言编程很自由，但这种自由不可避免的导致它走向复杂，而且不同的程序员水平不一，想法不同，风格迥异，使各种代码参差不齐，难以阅读和维护，容易产生 bug. 而我们制定这些规则在保证代码易于管理，使代码易于管理的方法之一是加强代码一致性. 让任何程序员都可以快速读懂你的代码这点非常重要. 保持统一编程风格并遵守约定意味着可以很容易根据 “模式匹配” 规则来推断各种标识符的含义. 创建通用，必需的习惯用语和模式可以使代码更容易理解. 在一些情况下可能有充分的理由改变某些编程风格，但我们还是应该遵循一致性原则，**尽量不要这么做**. 某些情况下,我们会限制甚至禁止使用某些 C 语言特性. 这么做是为了保持代码清爽，避免这些特性可能导致的各种问题.

手册中列举了这类特性，并解释为什么这些特性被限制使用.

注意: 本手册并不是 C 语言教程，我们假定读者已经对 C 非常熟悉. 文中图片中的代码均采自 Linux Kernel. 不指明具体出处. 由于规则很多，所以可能无法

做到完全遵守，也就没有全部列出.

第 1 章 代码风格

这是一个简短的文档，描述了μFun 软件项目的首选代码风格。代码风格是因人而异的，而且我们并不愿意把我们的观点强加给任何人，不过本文档所讲述的是μFun 项目代码所必须遵守的风格，希望绝大多数其他代码也能遵守这个风格。

所以，请在写代码时尽量遵守本文所述的风格。

1.0 TAB 宽度

TAB 制表符要 8 个空格

理由：缩进的全部意义就在于清楚的定义一个控制块起止于何处，让代码结构层次分明。尤其是当你盯着你的屏幕连续看了几个小时之后，你会发现大一点的缩进会使你更容易分辨缩进。如果你需要 3 级以上的缩进，不管用何种方式你的代码已经有问题了，应该修正你的程序。

另外需要指明的是，**不要**设置编辑器将 TAB 制表符转换为空格.

1.1 行长度

> **Tip:** 每一行代码字符数不超过 80.

我们也认识到这条规则是有争议的，但很多已有代码都已经遵照这一规则，我们感觉一致性更重要.

优点: 提倡该原则的人主张强迫他们调整编辑器窗口大小很野蛮. 很多人同时并排开几个代码窗口,根本没有多余空间拉伸窗口. 大家都把窗口最大尺寸加以限定, 并且 80 列宽是传统标准. 为什么要改变呢?

缺点: 反对该原则的人则认为更宽的代码行更易阅读. 80 列的限制是上个世纪 60 年代的大型机的古板缺陷; 现代设备具有更宽的显示屏, 很轻松的可以显示更多代码.

结论: 80 个字符是最大值.

特例:

- 如果一行注释包含了超过 80 字符的命令或 URL, 出于复制粘贴的方便允许该行超过 80 字符.
- 包含长路径的 #include 语句可以超出 80 列. 但应该尽量避免.
- 头文件保护可以无视该原则.

1.2 函数定义/声明/调用

> **Tip:** 返回类型和函数名在同一行, 参数也尽量放在同一行.

注意以下几点:

- 返回值总是和函数名在同一行;
- 左圆括号总是和函数名在同一行;
- 函数名和左圆括号间没有空格;
- 圆括号与参数间没有空格;
- 函数声明和实现处的所有形参名称必须保持一致;
- 所有形参应尽可能对齐;

函数调用遵循如下形式:

```
349 asmlinkage int sys_oabi_ipc(uint call, int first, int second, int third,
  1             void __user *ptr, long fifth)
  2 {
  3         switch (call & 0xffff) {
  4         case SEMOP:
  5                 return  sys_oabi_semtimedop(first,
  6                                 (struct oabi_sembuf __user *)ptr,
  7                                 second, NULL);
  8         case SEMTIMEDOP:
  9                 return  sys_oabi_semtimedop(first,
 10                                 (struct oabi_sembuf __user *)ptr,
 11                                 second,
 12                                 (const struct timespec __user *)fifth);
 13         default:
 14                 return sys_ipc(call, first, second, third, ptr, fifth);
 15         }
 16 }
```

```
bool retval = DoSomething(argument1, argument2, argument3);
```

如果同一行放不下，可断为多行，后面每一行都和第一个实参对齐，左圆括号后和右圆括号前不要留空格:

```
bool retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

如果函数参数很多，出于可读性的考虑可以在每行只放一个参数:

```
bool retval = DoSomething(argument1,
                          argument2,
                          argument3,
                          argument4);
```

1.3 条件语句

C 语言风格中另外一个常见问题是大括号的放置。和缩进大小不同，选择或弃用某种放置策略并没有多少技术上的原因，不过首选的方式，就像 Kernighan 和 Ritchie(K&R 风格)展示给我们的，是把起始大括号放在行尾，而把结束大括号放在行首，所以：

```
320
        for (i = 0; i < nsops; i++) {
                __get_user_error(sops[i].sem_num, &tsops->sem_num, err);
                __get_user_error(sops[i].sem_op,  &tsops->sem_op,  err);
                __get_user_error(sops[i].sem_flg, &tsops->sem_flg, err);
                tsops++;
        }

        if (timeout) {
                /* copy this as well before changing domain protection */
                err |= copy_from_user(&local_timeout, timeout, sizeof(*timeout));
                timeout = &local_timeout;
        }

        if (err) {
                err = -EFAULT;
        }
        else {
                mm_segment_t fs = get_fs();
                set_fs(KERNEL_DS);
                err = sys_semtimedop(semid, sops, nsops, timeout);
                set_fs(fs);
        }

```


if/else 最好也遵守同样的规则，这样和 Linux 内核风格是兼容的，不遵守则是与 Linux 风格一致. 各自取舍.

```
    if (user_mode(regs)) {
            current->thread.error_code = err;
            current->thread.trap_no = trap;

            force_sig_info(info->si_signo, info, current);
    } else {
            die(str, regs, err);
    }
```


```
320
        for (i = 0; i < nsops; i++) {
                __get_user_error(sops[i].sem_num, &tsops->sem_num, err);
                __get_user_error(sops[i].sem_op,  &tsops->sem_op,  err);
                __get_user_error(sops[i].sem_flg, &tsops->sem_flg, err);
                tsops++;
        }

        if (timeout) {
                /* copy this as well before changing domain protection */
                err |= copy_from_user(&local_timeout, timeout, sizeof(*timeout));
                timeout = &local_timeout;
        }

        if (err) {
                err = -EFAULT;
        }
        else {
                mm_segment_t fs = get_fs();
                set_fs(KERNEL_DS);
                err = sys_semtimedop(semid, sops, nsops, timeout);
                set_fs(fs);
        }

```


也请注意这种大括号的放置方式也能使空(或者差不多空的)行的数量最小化，

同时不失可读性。因此，由于你的屏幕上的新行是不可再生资源（想想 25 行的终端屏幕），你将会有更多的空行来放置注释。

当只有一个单独的语句的时候，不用加不必要的大括号。但是我们**建议**你在外围加上{}约束，这样是防止后期维护人员维护升级时疏忽而忘记敲入 {}，导致程序 Debug 困难.

如果 if 条件表达式超过了 80 个字符，应该使用以下的方式，逻辑(&&或者||) 运算符总位于行尾:

```
56  if ((this_one_thing > this_other_thing) &&
57          (a_third_thing == a_fourth_thing) &&
58          yet_another && last_one) {
59          // do something
60  }
61
62  if ((this_one_thing > this_other_thing)
63          && (a_third_thing == a_fourth_thing)
64          && yet_another && last_one) {
65          // do something
66  }
67
68
69  if ((this_one_thing > this_other_thing) ||
70          (a_third_thing == a_fourth_thing) ||
71          yet_another || last_one) {
72          // do something
73  }
74
75  if ((this_one_thing > this_other_thing)
76          || (a_third_thing == a_fourth_thing)
77          || yet_another || last_one) {
78          // do something
79  }
80
```

注意，上例代码换行时，所有逻辑运算符均位于行尾/行首，也可以适当插入若干圆括号，合理使用的话对增强代码可读性很有帮助。

1.4 循环和开关选择语句

do...while 循环注意结束大括号独自占据一行，除非它后面跟着同一个语句的剩余部分，也就是 do 语句中的“while”或者 if 语句中的“else”，像这样：

```
568
        do {
                seq_printf(s, "%s", flags[i]);
                if (i > 0) {
                        seq_printf(s, " | ");
                }
        } while (i-- > 0);
}
```

switch 语句中的 case 块可以使用大括号也可以不用，取决于你的个人喜好. 如果用的话，要按照下图所述的样式.如果有不满足 case 条件的枚举值，switch 应该总是包含一个 default 匹配.


```
429
asmlinkage long sys_oabi_socketcall(int call, unsigned long __user *args)
{
        unsigned long r = -EFAULT, a[6];

        switch (call) {
        case SYS_BIND:
                if (copy_from_user(a, args, 3 * sizeof(long)) == 0) {
                        r = sys_oabi_bind(a[0], (struct sockaddr __user *)a[1], a[2]);
                }
                break;
        case SYS_CONNECT:
                if (copy_from_user(a, args, 3 * sizeof(long)) == 0) {
                        r = sys_oabi_connect(a[0], (struct sockaddr __user *)a[1], a[2]);
                }
                break;
        case SYS_SENDTO:
                if (copy_from_user(a, args, 6 * sizeof(long)) == 0) {
                        r = sys_oabi_sendto(a[0], (void __user *)a[1], a[2], a[3],
                                            (struct sockaddr __user *)a[4], a[5]);
                }
                break;
        case SYS_SENDMSG:
                if (copy_from_user(a, args, 3 * sizeof(long)) == 0) {
                        r = sys_oabi_sendmsg(a[0], (struct msghdr __user *)a[1], a[2]);
                }
                break;
        default:
                {
                        r = sys_socketcall(call, args);
                        // Do something
                }
        }

        return r;
}
```

对于空循环体应使用 {} 或 continue，而不是一个简单的分号.

```
while (condition) {
        // Repeat test until it returns false.
}

for (int i = 0; i < kSomeNumber; ++i) {
        // Good - empty body.
}

while (condition) continue;     // Good - continue indicates no logic.

while (condition);              // 不好的样式
```

1.5 空格/空行

μFun 的空格使用方式（主要）取决于它是用于函数还是关键字.（大多数）关键字后要加一个空格。值得注意的例外是 sizeof、typeof、alignof 和__attribute__以及一些编译器自定义扩展的关键字，这些关键字某些程度上看起来更像函数（它们也常常伴随小括号而使用，尽管在 C 语言里这样的小括号不是必需的，就像“struct fileinfo info”声明过后的“sizeof info”）。

所以在这些关键字之后放一个空格：

```
if, switch, case, for, do, while
```

但是不要在 sizeof、typeof、alignof 或者__attribute__这些关键字之后放空格。例如，

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

不要在小括号里的表达式两侧加空格。这是一个反例：

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

当声明指针类型或者返回指针类型的函数时，“*”的首选使用方式是使之靠近变量名或者函数名，而不是靠近类型名。例子：

```
3
1 char *linux_banner;
2 unsigned long long memparse(char *ptr, char **retptr);
3 char *match_strdup(substring_t *s);
4
~
```

在大多数二元和三元操作符两侧使用一个空格，例如下面所有这些操作符：

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

但是一元操作符后不要加空格：

& * + - ~ ! sizeof typeof alignof __attribute__ defined

后缀自加和自减一元操作符前不加空格：

++ --

前缀自加和自减一元操作符后不加空格：

++ --

“.”和“->”结构体成员操作符前后不加空格。不要在行尾留空白。有些可以自动缩进的编辑器会在新行的行首加入适量的空白，然后你就可以直接在那一行输入代码。不过假如你最后没有在那一行输入代码，有些编辑器就不会移除已经加入的空白，就像你故意留下一个只有空白的行。包含行尾空白的行就这样产生了。

另外就是如何合理的使用空行了. 很简单, 就是将逻辑功能相关的语句放在一起, 与不相关的语句用空行隔开.这样非常有助于阅读理解代码意图.

```
251  static int __die(const char *str, int err, struct thread_info *thread, struct pt_regs *regs)
252  {
253          struct task_struct *tsk = thread->task;
254          static int die_counter;
255          int ret;
256
257          printk(KERN_EMERG "Internal error: %s: %x [#%d]" S_PREEMPT S_SMP
258                            S_ISA "\n", str, err, ++die_counter);
259
260          /* trap and error numbers are mostly meaningless on ARM */
261          ret = notify_die(DIE_OOPS, str, regs, err, tsk->thread.trap_no, SIGSEGV);
262          if (ret == NOTIFY_STOP)
263                  return ret;
264
265          print_modules();
266          __show_regs(regs);
267          printk(KERN_EMERG "Process %.*s (pid: %d, stack limit = 0x%p)\n",
268                            TASK_COMM_LEN, tsk->comm, task_pid_nr(tsk), thread + 1);
269
270          if (!user_mode(regs) || in_interrupt()) {
271                  dump_mem(KERN_EMERG, "Stack: ", regs->ARM_sp,
272                                   THREAD_SIZE + (unsigned long)task_stack_page(tsk));
273                  dump_backtrace(regs, tsk);
274                  dump_instr(KERN_EMERG, regs);
275          }
276
277          return ret;
278  }
```

1.6 预处理左对齐

即使预处理指令位于缩进代码块中，指令也应从行首开始.

```
80  void dump_backtrace_entry(unsigned long where, unsigned long from, unsigned long frame)
81  {
82  #ifdef CONFIG_KALLSYMS
83          printk("[<%08lx>] (%pS) from [<%08lx>] (%pS)\n", where, (void *)where, from, (void *)from);
84  #else
85          printk("Function entered at [<%08lx>] from [<%08lx>]\n", where, from);
86  #endif
87
88          if (in_exception_text(where)) {
89                  dump_mem("", "Exception stack", frame + 4, frame + 4 + sizeof(struct pt_regs));
90          }
91  }
92
```


1.4 typedef 别名

不要使用类似“vps_t”之类的东西。

对结构体和指针使用 typedef 是一个错误。当你在代码里看到：

vps_t a;

这代表什么意思呢？相反，如果是这样

struct virtual_container *a;

你就知道“a”是什么了。很多人认为 typedef“能提高可读性”。实际不是这样的。它们只在下列情况下有用：

a) 完全不透明的对象（这种情况下要主动使用 typedef 来隐藏这个对象实际上是什么）. 例如：“pte_t”等不透明对象，你只能用合适的访问函数来访问它们。注意！不透明性和“访问函数”本身是不好的。我们使用 pte_t 等类型的原因在于真的是完全没有任何共用的可访问信息。

b) 清楚的整数类型，如此，这层抽象就可以帮助消除到底是“int”还是“long”的混淆。 u8/u16/u32 是完全没有问题的 typedef，不过它们更符合类别(d)而不是这里。再次注意！要这样做，必须事出有因。如果某个变量是

“unsigned long “，那么没有必要 typedef unsigned long myflags_t; 不过如果有一个明确的原因，比如它在某种情况下可能会是一个“unsigned int”而在其他情况下可能为“unsigned long”，那么就不要犹豫，请务必使用 typedef。

c) 和标准 C99 类型相同的类型，在某些例外的情况下，虽然让眼睛和脑筋来适应新的标准类型比如“uint32_t”不需要花很多时间，可是有些人仍然拒绝使用它们。因此，Linux 特有的等同于标准类型的“u8/u16/u32/u64”类型和它们的有符号类型是被允许的——尽管在你自己的新代码中，它们不是强制要求要使用的。当编辑已经使用了某个类型集的已有代码时，你应该遵循那些代码中已经做出的选择。

d) 可以在用户空间安全使用的类型。 在某些用户空间可见的结构体里，我们不能要求 C99 类型而且不能用上面提到的“u32”类型。因此，我们在与用户空间共享的所有结构体中使用__u32 和类似的类型。

可能还有其他的情况，不过基本的规则是永远不要使用 typedef，除非你可以明确的应用上述某个规则中的一个。总的来说，如果一个指针或者一个结构体里的元素可以合理的被直接访问到，那么它们就不应该是一个 typedef。

第 2 章　注释规则

参考 Linux，加入版权说明

2.0 文件注释

2.1 函数注释

2.3 变量注释

2.4 实现注释

2.5 TODO 注释

2.6 块注释

If/else/for/while/dowhile/switch

2.7 弃用说明注释

第 3 章　变量规则

第 4 章　函数规则

函数应该简短而漂亮，并且只完成一件事情。函数应该可以一屏或者两屏显示完，只做一件事情，而且把它做好，当然，对于一些底层代码可以根据实际情况作出调整。

遵循的原则就是: 高扇入，合理扇出. 即尽量多的让自己被别的模块调用而少调用别的模块(< 7)，扇出过高，说明要考虑的问题就越多，考虑的东西越多，人脑犯错的可能就越大.

一个函数的最大长度是和该函数的复杂度和缩进级数成反比的。所以，如果你有

一个理论上很简单的只有一个很长（但是简单）的 case 语句的函数，而且你需要在每个 case 里做很多很小的事情，这样的函数尽管很长，但也是可以的。

不过，如果你有一个复杂的函数，而且你怀疑一个天分不是很高的高中一年级学生可能甚至搞不清楚这个函数的目的，你应该严格的遵守前面提到的长度限制。使用辅助函数，并为之取个具描述性的名字（如果你觉得它们的性能很重要的话可以让编译器内联它们，这样的效果往往会比你写一个复杂函数的效果要好。）函数的另外一个衡量标准是本地变量的数量。此数量不应超过 5 - 10 个，否则你的函数就有问题了。重新考虑一下你的函数，把它分拆成更小的函数。人的大脑一般可以轻松的同时跟踪 7 个不同的事物，如果再增多的话，就会糊涂了。即便你聪颖过人，你也可能会记不清你 2 个星期前做过的事情。在源文件里，使用空行隔开不同的函数。

第 5 章　Git Commit 规则

第 6 章　文档规则

参考资料:

1) Linux 内核代码风格(中文版)

2) Linux kernel coding style(English)

3) Google C++ 风格指南 - 中文版

4) Google C++ Style Guide(English)

5) MISRA-C-2004_工业标准的 C 编程规范_中文版

6) MISRA-2004 IAR WorkBench 版本(英文)

推荐书目:

TODO...