嵌入式C语言风格指南

目录

1.头文件

通常每一个 `.c` 文件都有一个对应的 `.h` 文件. 也有一些常见例外, 如单元测试代码和只包含 `main()` 函数的 `.c` 文件.

正确使用头文件可令代码在可读性、文件大小和性能上大为改观，下面的规则将引导你规避使用头文件时的各种陷阱.

1.1.Self-contained头文件

> **Tip**: 头文件应该能够自给自足（self-contained,也就是可以作为第一个头文件被引入），以 `.h` 结尾。

所有头文件要能够自给自足。换言之，用户和重构工具不需要为特别场合而包含额外的头文件，一个头文件要有 define-guard，统统包含它所需要的其它头文件，也不要求定义任何特别 symbols.

1.2.#define保护

> **Tip**: 所有头文件都应该使用 `#define` 来防止头文件被多重包含, 命名格式应当是: `<PROJECT>_<PATH>_<FILE>_H_` .

为保证唯一性, 头文件的命名应该基于所在项目源代码树的全路径. 例如, 项目 `foo` 中的头文件 `foo/src/bar/baz.h` 可按如下方式保护:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
...
#endif // FOO_BAR_BAZ_H_
```

1.3.#include的路径及顺序

> **Tip**: 使用标准的头文件包含顺序可增强可读性, 避免隐藏依赖: 相关头文件, C 库, 其他库的 `.h`, 本项目内的 `.h`.

项目内头文件应按照项目源代码目录树结构排列, 避免使用特殊的快捷目录 `.` (当前目录) 或 `..` (上级目录). 例如, `src/base/logging.h` 应该按如下方式包含:

```
#include "base/logging.h"
```

又如, `dir/foo.c` 或 `dir/foo_test.c` 的主要作用是实现或测试 `dir2/foo2.h` 的功能, `foo.c` 中包含头文件的次序如下:

1. `dir2/foo2.h` (优先位置, 详情如下)
2. C 系统文件如 `stdint.h,stdio.h,string.h`
3. 其他库的 `.h` 文件
4. 本项目内 `.h` 文件

这种优先的顺序排序保证当 dir2/foo2.h 遗漏某些必要的库时， dir/foo.c 或 dir/foo_test.c 的构建会立刻中止。因此这一条规则保证维护这些文件的人们首先看到构建中止的消息而不是维护其他包的人们。

foo.c 和 foo.h 通常位于同一目录下, 但也可以放在不同目录下,我们可以创建 src目录来存放foo.c文件，创建inc目录来存放foo.h文件

您所依赖的符号 (symbols) 被哪些头文件所定义，您就应该包含（include）哪些头文件，前置声明__ (forward declarations) 情况除外。比如您要用到 bar.h 中的某个符号, 哪怕您所包含的 foo.h 已经包含了 bar.h, 也照样得包含 bar.h, 除非 foo.h 有明确说明它会自动向您提供 bar.h 中的 symbol. 不过，凡是 c 文件所对应的「相关头文件」已经包含的，就不用再重复包含进其 c 文件里面了，就像 foo.c 只包含 foo.h 就够了，不用再管后者所包含的其它内容。

举例来说, foo/internal/fooserver.c 的包含次序如下:

```
#include "foo/public/fooserver.h" // 优先位置

#include <sys/types.h>
#include <unistd.h>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

2.作用域

2.1.静态变量

> **Tip**: 在 `.c` 文件中定义一个不需要被外部引用的变量时，可以将它们声明为 `static` 。但是不要在 `.h` 文件中这么做。

定义:

函数和变量可以经由声明为 `static` 拥有内部链接性，这意味着你在这个文件中声明的这些标识符都不能在另一个文件中被访问。即使两个文件声明了完全一样名字的标识符，它们所指向的实体实际上是完全不同的。

推荐、鼓励在 `.c` 中对于不需要在其他地方引用的标识符使用内部链接性声明，但是不要在 `.h` 中使用。

2.2.局部变量

> **Tip**: 将函数变量尽可能置于最小作用域内, 并在变量声明时进行初始化.

C 允许在函数的任何位置声明变量. 我们提倡在尽可能小的作用域中声明变量, 离第一次使用越近越好. 这使得代码浏览者更容易定位变量声明的位置, 了解变量的类型和初始值. 特别是，应使用初始化的方式替代声明再赋值, 比如:

```
    void demo(){
        /*
        不推荐，初始化和声明分离
        同时如果不及时初始化会对后续引用造成意想不到的后果
        */
        u32 i;
        /*
        To do something
        */
        i = 1000;
    }

    void demo(){
        //推荐，在变量定义时候就需要即使初始化
        u32 i = 0;
        u32 j = 1000;
        /*
        To do something
        */
    }
```

注意

1. 尽量不用全局函数和全局变量, 考虑作用域的限制, 尽量单独形成编译单元，来减少代码的耦合程度，对没有相关性或关联性较低的函数尽可能的拆分;

2. 作用域的使用, 除了考虑名称污染, 可读性之外, 主要是为降低耦合, 提高编译/执行效率.
3. 局部变量在声明的同时进行显式值初始化，比起隐式初始化再赋值的两步过程要高效，同时也贯彻了计算机体系结构重要的概念「局部性」。

3.类型约定

3.1.常规类型

在C语言环境下，一般的，默认使用如下方式来表示一个变量类型

```
char
unsigned char
short int
unsigned short int
int
...
```

或者

```
typedef  signed char         int8_t
typedef  unsigned char       uint8_t
typedef  signed short int    int16_t
typedef  unsigned short int  uint16_t
typedef  signed int          int32_t
typedef  unsigned int        uint32_t
```

3.2.约定类型

在这里我们约定使用的方式，使用简写尽可能的减少字符输入次数.

```
typedef  unsigned long       u32;
typedef  unsigned int        u16;
typedef  unsigned char       u8;

typedef  signed long         s32;
typedef  signed int          s16;
typedef  signed char         s8;
```

4.函数

4.1.参数顺序

C语言 中的函数参数不论是函数的输入还是或者是函数的输出, 或兼而有之. 输入参数通常是值参或 `const` 引用, 输出参数或输入/输出参数则一般为非 `const` 指针. 在排列参数顺序时, 将所有的输入参数置于输出参数之前. 特别要注意, 在加入新参数时不要因为它们是新参数就置于参数列表最后, 而是仍然要按照前述的规则, 即将新的输入参数也置于输出参数之前.

这并非一个硬性规定. 输入/输出参数 (通常是类或结构体) 让这个问题变得复杂. 并且, 有时候为了其他函数保持一致, 还需要有所变通.

```
    void Foo(u8 in1,u8 in2,u8 in3,u16 *out1,u16 *out2) {
        //To do something
    }

    //参数不定长或者过多，通过这种方式进行传递
    void uart_printf(char *fmt, ...) {

    }
```

4.2.编写简短函数

我们倾向于编写简短, 凝练的函数，长函数有时是合理的, 因此并不硬性限制函数的长度. 如果函数超过 40 行, 可以思索一下能不能在不影响程序结构的前提下对其进行分割.

即使一个长函数现在工作的非常好, 一旦有人对其修改, 有可能出现新的问题, 甚至导致难以发现的 bug. 使函数尽量简短, 以便于他人阅读和修改代码.

在处理代码时, 有可能会发现复杂的长函数. 如果证实这些代码使用 / 调试起来很困难, 或者你只需要使用其中的一小段代码, 考虑将其分割为更加简短并易于管理的若干函数.

4.3.引用参数

在程序中, 传递参数鼓励使用指针的方式，而不是通过赋值的方式对函数进行传参

```
    //推荐 - 参数较多的情况下使用，但是由于是不可重入函数，多线程时候慎用
    void Foo(u8 *in,u16 *out) {
        //To do something
    }

    //推荐
    u16 Foo(u8 in) {
        u16 out++;
        //To do something
        return out;
    }
```

```
//不推荐 - 在函数内部对全局变量进行修改，增加耦合性
u16 out = 0;
void Foo(u8 in) {
    out++;
    //To do something
    return out;
}
```

5.命名约定

最重要的一致性规则是命名管理. 命名的风格能让我们在不需要去查找类型声明的条件下快速地了解某个名字代表的含义: 类型, 变量, 函数, 常量, 宏, 等等, 甚至. 我们大脑中的模式匹配引擎非常依赖这些命名规则.

5.1.通用命名规则

尽可能使用描述性的命名, 不要心疼空间,函数命名, 变量命名, 文件命名要有描述性; 少用缩写。毕竟相比之下让代码易于新读者理解更重要.

```
u32 price_count_reader;    // 无缩写
u32 num_errors;            // "num" 是一个常见的写法
u32 num_dns_connections;   // 人人都知道 "DNS" 是什么
```

不好的示例

```
u32 n;                     // 毫无意义.
u32 nerr;                  // 含糊不清的缩写.
u32 n_comp_conns;          // 含糊不清的缩写.
u32 wgc_connections;       // 只有写程序的热呢才知道变量是什么意思.
u32 pc_reader;             // "pc" 有太多可能的解释了.
u32 cstmr_id;              // 删减了若干字母.
```

注意, 一些特定的广为人知的缩写是允许的, 例如用 `i,j,k,index` 表示迭代变量,`tmp, var`代表临时变量.

5.2.文件命名

文件名全部小写, 可以包含下划线 (_),可接受的文件命名示例:

- `my_useful.c`
- `my-useful.c`
- `myuseful.c`
- `myuseful_test.c`

通常应尽量让文件名更加明确，定义类时文件名一般成对出现, 如 `foo_bar.h` 和 `foo_bar.c`.

5.3.变量命名

变量 (包括函数参数) 和数据成员名一律小写, 单词之间用下划线连接. 类的成员变量以下划线结尾, 但结构体的就不用, 如: `a_local_variable`, `a_struct_data_member`.

普通变量命名举例:

```
u8 table_name;  // 推荐 - 用下划线.
u8 tableName;   // 推荐 - 驼峰命令，混合大小写

u8 tablename;   // 不推荐 - 全小写，不容易分清楚单词间隔.
```

对于具有同类意义的变量，尽可能的使用结构体来进行归类.

```
//推荐 - 对同类型的变量进行归类
typedef struct {
    u32 ts_motor_start;
    u32 ts_motor_reminder;
} MOTO_STATE_S;

//不推荐 - 同类型变量分散开
u32 ts_motor_start;
u32 ts_motor_reminder;
```

5.4.常量命名

声明为 `const` 的变量, 或在程序运行期间其值始终保持不变的, 大小写混合以及小写加下划线均可，例如:

```
const s32 DaysInAWeek = 7;
const char string[] = "hello word";
```

5.5.函数命名

常规函数使用大小写混合(相同的函数使用场景尽量使用同样的命名规则), 取值和设值函数则要求与变量名匹配:

```
MyExcitingFunction(),              //初始化函数使用的方式
MyExcitingMethod(),
my_exciting_member_variable(),     //普通函数的命名方式
set_my_exciting_member_variable(),
```

一般来说, 函数名的每个单词通过下划线来进行划分

```
add_table_entry();
delete_url();
open_file_ordie();
```

对于同类型的函数尽量使用一致的开头

```
```

```
task_add();
task_delect();
task_subtract()
task_multiply();
task_led_power_on();
task_led_power_off();
```

取值和设值函数的命名与变量一致. 一般来说它们的名称与实际的成员变量对应, 但并不强制要求. 例如 `int count()` 与 `void set_count(int count)`.

5.6.结构体命名

创建新的结构体时候，参照5.3变量命名 方式，采用小写加下划线的方式定义结构体成员。

```
struct MOTO_MODE_PARA_S {
    u16 motor_freq;
    u16 motor_duty;
};
```

定义新的结构体类型，结构体类型命令采用大写以及_S后缀的方式

```
typedef struct
{
    u16 motor_freq;
    u16 motor_duty;
    u16 motor_time;
    u16 circ_time;
    u16 reminder;
} MOTO_MODE_PARA_S;
```

5.7.枚举命名

枚举的命名应当和 5.3.变量命名 或 5.8.宏命名 一致: `ENUM_NAME`，并且通过_E后缀类明显的加以区别。

单独的枚举值应该优先采用 5.3.变量命名 的命名方式. 但 5.8.宏命名 方式的命名也可以接受. 枚举名 `UrlTableErrors` (以及 `AlternateUrlTableErrors`) 是类型, 所以要用大小写混合的方式.

```
//推荐 - 有统一的标识符 SCENE_ENV
typedef enum {
    SCENE_ENV_NONE,
    SCENE_ENV_HAND,
    SCENE_ENV_BREAK,
} SCENE_ENV_E;

typedef enum {
```

```
        //推荐 - 有统一的标识符 SCENE_TYPE
        SCENE_TYPE_1  = 0,
        SCENE_TYPE_2  = 1,
        SCENE_TYPE_3  = 2,
        SCENE_TYPE_4  = 3,
        SCENE_TYPE_5  = 4,
        SCENE_TYPE_6  = 5,
        SCENE_TYPE_7  = 6,
        SCENE_TYPE_8  = 7,
        SCENE_TYPE_MAX
    } SCENE_TYPE_E;

    enum AlternateUrlTableErrors {
        //不推荐 - 没有统一的标识符，不便于代码阅读
        OK = 0,
        OUT_OF_MEMORY = 1,
        MALFORMED_INPUT = 2,
    };
```

建议采用 5.8.宏命名 的方式命名枚举值. 由于枚举值和宏之间的命名冲突, 直接导致了很多问题. 由此, 这里改为优先选择常量风格的命名方式. 新代码应该尽可能优先使用常量风格. 但是老代码没必要切换到常量风格, 除非宏风格确实会产生编译期问题.

5.8.宏命名

对于函数中超过一处引用了同一个变量值，约定使用宏来代替该变量，如果该变量有变，即可通过宏定义来快速修改。与此同时，读者也可以快速通过宏定义来获取变量的含义，而不会造成奇怪的魔法数字。

宏定义使用大写加下划线的方式命令。

```
    #define ROUND(x) ...
    #define PI_ROUNDED 3.0

    #define BUTTON_SHORT_PRESS_THRESHOLD    100     //ms
    #define BUTTON_LONG_PRESS_THRESHOLD     1000    //ms

    #define MOTOR_STANDARD_MODE_PERIOD      ((unsigned int)3620)
    #define MOTOR_STANDARD_MODE_DEADZONE    ((unsigned int)340)
    #define MOTOR_STANDARD_MODE_DUTY        ((unsigned int)1810 -
MOTOR_STANDARD_MODE_DEADZONE/2)
```