目录

- 使用经边界校验或具备错误反馈的安全函数
- Embedded Shell Scripts Should be Avoided

通用

General

为何要采用编码风格?

Why a coding style?

A consistent coding style makes the code easier to read and maintain.

一致的编码风格使代码更易于阅读和维护。

Some rules have deeper reasons than just the look and feel, and actually reduce the risk for introducing bugs.

有些规则除了外观和感觉之外还有更深层次的原因，实际上还能降低引入错误的风险。

There is no 'one true coding style'. Any coding style is better than no coding style, as long as it is followed by everyone.

没有"唯一正确的编码风格”。任何编码风格都比没有编码风格要好，只要每个人都遵循它。

什么是"编码标准”?

What is a 'coding standard'?

A coding standard goes beyond just the coding style. It includes rules and recommendations for writing the code, not just how to format it.

编码标准不仅仅涉及编码风格。它还包括编写代码的规则和建议，而不仅仅是如何对其进行格式化。

请勿混合编码风格

Do not mix coding styles

Every project has some sources that do not conform to the otherwise accepted coding style.

每个项目都有一些不符合通常所接受的编码风格的代码来源。

However, when in Rome, do as the Romans do. In other words, always try to make the added or changed code 'blend in' with existing code.

然而，应该遵循现有代码的风格。换句话说，始终努力使新增或修改的代码与现有代码"融为一体”。

Similarly, for any 3rd-party code, conform to whatever style is present there (but also consult a senior engineer about the proper procedure for such modifications).

同样，对于任何第三方代码，要遵循其中已有的风格(但也要咨询一位资深工程师，了解此类修改的正确流程)。

不要美化现有的代码

Do not beautify existing code

No drastic changes should be made to code that is already on the master branch. If any fixes have to be merged to or from older branches later on, then such changes can make merging a nightmare.

对于已经在主分支上的代码，不应进行重大修改。如果之后必须将任何修复内容合并到或从较旧的分支合并过来，那么此类更改可能会使合并工作变得极其困难。

If reformatting is approved for a certain module, the cosmetic changes have to be made separately from any functional changes. Never mix cosmetic changes and functional changes in the same commit.

如果某个模块的重新格式化获得批准，那么外观上的改动必须与任何功能上的改动分开进行。切勿在同一个提交中混合外观上的改动和功能上的改动。

缩进与间距

Indentation and Spacing

请勿使用制表符

Do not use tabs

Never use tabs in the code.

代码中切勿使用制表符

While editors can be configured to have tab stops at 4 instead of 8 columns, other tools (like visual diff tools) might not all be configurable.

虽然编辑器可以设置制表符停靠在 4 列而非 8 列，但其他工具(如可视化差异对比工具)可能并非都能进行这样的配置。

Exceptions:

- Makefiles
- Linux kernel code
- any other 3rd-party code that uses hard tabs

When using an editor you are not familiar with, double-check the diff before committing.

使用不熟悉的编辑器时，在提交前请仔细检查差异。

When editing 3rd-party code, double-check the diff before committing.

在编辑第三方代码时，请在提交前仔细检查差异。

缩进大小为 4 个空格

Indent size is 4 spaces

Configure your editor appropriately.

对您的编辑器进行适当的设置。

用"Allman风格"来放置花括号

Use "Allman style" for braces

Allman style is similar to K&R style, except that the opening brace is put on the next line (aligned with the closing brace), and that "else" and "else if" keywords get their own line.

Allman风格与K&R 风格类似，不同之处在于大括号会放在下一行(与闭合大括号对齐)，并且"else"和"else if"关键字会单独占一行。

This rule applies to functions, control statements, structures, and unions.

此规则适用于函数、控制语句、结构体以及联合体。

Exceptions to this rule are listed below, after the examples.

这条规则的例外情况如下所示，列在示例之后。

Example (function):

```
int function()
{
    return 0;
}
```

Example ("if ... else"):

```
if (condition)
{
    do_something();
}
else
{
    do_something_else();
}
```

Example ("if ... else if ... else"):

```
if (condition_1)
{
    do_something();
}
else if (condition_2)
{
    do_something_else();
```

```
}
else
{
    do_last_resort_action();
}
```

Example ("switch"):

```
switch (expression)
{
    case 0: /* falls through */
    case 1:
        ...
        break;
    case 2:
        ...
        break;
    default:
        LOG(...);
        goto error;
}
```

Example ("struct" declaration):

```
struct my_structure
{
    type_1 member_1;
    type_2 member_2;
    ...
};
```

Exceptions:

There is an exception for the "do...while" statements. In this case, both the "do" and the "while" keywords should be on the same line as their respective braces.

对于"do...while"语句有一个例外情况。在这种情况下，"do"和"while"这两个关键字应当与它们各自的花括号置于同一行。

To make it more readable (especially for larger blocks), an empty line can be inserted after the line with the "do" keyword and before the line with the "while" keyword.

为了使其更具可读性(尤其是对于较长的代码段而言)，可以在包含"do"关键字的那行代码之后以及包含"while"关键字的那行代码之前插入一个空行。

Exception ("do ... while"):

```
do {

    something();
    ...

} while (not_done_condition);
```

Another exception is the 'extern "C"' scope block, where the opening brace is also on the same line as the declaration, and the content of the scope is not indented (the same rule applies for the outermost namespace in C++).

另一个例外情况是"extern "C""作用域块，在这种情况下，起始大括号也与声明在同一行，且该作用域内的内容不进行缩进(在 C++ 中，对于最外层的命名空间也适用同样的规则)。

Exception (extern "C"):

```
#ifdef __cplusplus
extern "C" {
#endif

...
...

#ifdef __cplusplus
}
#endif
```

In some cases, exceptions are allowed for control statements. If the code is very simple and repetitive (e.g. merely reporting or logging different things), K&R style may be used.

在某些情况下，控制语句允许存在例外情况。如果代码非常简单且重复性很强(例如仅仅用于报告或记录不同的内容)，则可以采用 K&R 风格。

This should only be used for blocks containing no more than one or two statements.

此功能仅适用于包含不超过一两条语句的代码块。

If K&R style is used, then both the closing brace and the opening brace must be on the same line as the "else" (or "else if") statement.

如果采用 K&R 编码风格，那么闭合大括号和起始大括号都必须与"else"(或"else if")语句处于同一行。

Exception (simple, repetitive code):

```
if (condition) {
    report_something();
} else if (condition_2) {
    report_something_else();
```

```
} else {
    report_default();
}
```

In case of "else" and "else if" statements, all blocks must use the same style, therefore if any of the blocks is larger, then Allman style must be used throughout.

对于"else"和"else if"语句，所有代码块必须采用相同的格式，因此如果其中任何一个代码块的长度较长，那么就必须始终采用Allman格式。

循环和条件语句中应始终包含用花括号括起来的子语句

Loop and conditional statements should always have brace-enclosed sub-statements

The code has a more consistent 'look and feel' and is easier to read if all conditional and loop statements have braces.

该代码的外观和风格更加统一，而且如果所有的条件语句和循环语句都配有花括号，那么代码读起来也会更加容易理解。

This rule also prevents bugs caused by adding a line, but forgetting to add braces.

这条规则还能避免因添加了一行代码但忘记添加大括号而导致的错误。

Furthermore, if a line of code is added later on, the diff will be cleaner and will contain just the relevant functional change.

此外，如果之后再添加一行代码，那么差异报告将会更加清晰，只会包含相关的功能变更内容。

Example ("for" statement):

```
for (i=0; i<table_length; i++)
{
    process(element[i]);
}
```

Exceptions:

For very simple code, loops and conditional statements with a single-statement clause can be written as one-liners.

对于非常简单的代码，带有单句语句的循环和条件语句可以写成一行代码的形式。

For loops, this should only be done if the controlled statement is trivial so that the intention can be described with a single comment before the loop.

对于"for"循环，只有在所控制的语句非常简单的情况下才应采用这种方式，这样在循环之前就可以用一条单独的注释来说明其意图。

Simple conditional statements that test some value and perform a simple action often do not even need a comment.

那些仅用于测试某个值并执行简单操作的简单条件语句通常根本无需添加注释。

Exception (one-liners for trivial constructs):

```
if (elements_ll == NULL) goto error;    /* OK */

// find the last element
ptr = elements_ll;
while (ptr != NULL) ptr = ptr->next;    /* OK */
```

If braces are not used, then the entire "for", "while", "if", or "if...else" statement MUST be on the same line.

如果不使用花括号，那么整个"for"、"while"、"if"或"if...else"语句就必须在同一行书写。

Example (one-liners must be one-liners):

```
if (found == NULL)                      /* WRONG */
    goto error;

if (found == NULL) goto error;          /* Right */

for (i=0; i<QCSAPI_MAX_ETHER; i++)      /* WRONG */
    mac_addr[i] = (uint8_t)tmp[i];

                                        /* either use braces (Right) ... */
for (i=0; i<QCSAPI_MAX_ETHER; i++) {
    mac_addr[i] = (uint8_t)tmp[i];
}

                                        /* ... or a one-liner (Right) */
// populate mac_addr from tmp
for (i=0; i<QCSAPI_MAX_ETHER; i++) mac_addr[i] = (uint8_t)tmp[i];
```

Specifically for the "if...else" statement, if it is written as a one-liner, it is advisable to use braces to make the two clauses stand out.

对于"if...else"语句而言，如果将其写成一行代码的话，建议使用花括号来使两个分支的语句清晰区分。

However, only do this if the intention is easy to understand and there is no need to add a comment for each of the actions. In general code, it is usually better to avoid "if...else" one-liners, but they may be useful in macros.

然而，只有在意图易于理解且无需为每个操作添加注释的情况下才这样做。在一般的代码中，通常最好避免使用"if...else"这种单行代码结构，但在宏中它们可能会有用。

Example ("if ... else" one-liner):

```
for (i=0; i<QCSAPI_MAX_ETHER; i++) {
    if (hide_mac) mac_addr[i] = 0;      /* WRONG */
    else mac_addr[i] = (uint8_t)tmp[i]; /* WRONG */
}

                                        /* either expand it (Right) ... */
for (i=0; i<QCSAPI_MAX_ETHER; i++)
{
    if (hide_mac) {
        mac_addr[i] = 0;
    } else {
        mac_addr[i] = (uint8_t)tmp[i];
    }
}

                                        /* ... or use a one-liner (Right) */
for (i=0; i<QCSAPI_MAX_ETHER; i++) {
    if (hide_mac) { mac_addr[i] = 0; } else { mac_addr[i] = (uint8_t)tmp[i]; }
}
```

One-liners for "if...else" statements are allowed for algorithms which are rarely changed.

对于那些很少需要修改的算法，允许使用"if...else”语句中的简短语句。

For business logic, place "if" and "else" clauses on their own lines and use braces (K&R style may be used if the code is short and is likely to remain so).

对于业务逻辑部分，将"if”和"else”语句放在单独的行上，并使用花括号(如果代码较短且不太可能变长，可以采用 K&R 格式)。

Exception not allowed for non-trivial code (especially for business logic, which may change):

对于非简单代码不允许有任何例外(尤其是涉及业务逻辑的部分，其内容可能会发生变化)：

```
if (condition) {
    action_1(...);
} else {
    action_2(...);
}
```

每个语句都应该在单独的一行上

Each statement should be on a line of its own

Do not try to make the code compact by putting several statements on the same line.

不要试图将多个语句放在同一行中以保持代码紧凑。

Likewise, do not chain assignment statements.

同样，不要将赋值语句链接起来。

Example (put each statement on a line of its own):

```
a = NULL; b = NULL;      /* BAD CODE */

a = b = NULL;            /* BAD CODE */

a = NULL;                /* Correct */
b = NULL;
```

Also, do not put extra initializations into the 'init expression' of the "for" statement. Only initialize two variables if both are also handled (e.g. incremented or decremented) by the 'loop expression' of the "for" statement.

此外，不要在 for 语句的"初始化表达式"里塞入额外的初始化。只有在两个变量同时也会在"循环表达式"中(如自增或自减)处理时，才允许同时初始化它们。

Example (do not abuse the expressions of the "for" statement):

```
for (found=NULL, i=0; i<table_length; i++) ...   /* BAD CODE */

found = NULL;                                   /* Correct */
for (i=0; i<table_length; i++) ...
```

将复杂"for"语句的每个表达式放在单独的一行中

Put each expression of a complex "for" statement on a line of its own

When a "for" loop is used as an iterator, the expressions can become quite long. Putting them each on a line of its own makes the statement more readable.

将for循环用作迭代器时，表达式可能会变得相当长。将它们放在单独的一行中可以提高语句的可读性。

Example (a complex "for" expression):

```
for ( record = ds_dlist_ifirst(&record_iter, temp_list);
      record != NULL;
      record = ds_dlist_inext(&record_iter))
{
...
```

空块的使用规范

How to use empty blocks

If an empty block is needed for a conditional statement, use "{}" (with no space in-between), rather than a semicolon.

当条件语句需要空块时，应使用 "{}"(中间不留空格)，而非单独一个分号。

Example (empty clause):

```
while (wait_for_something() == RESULT_TIMEOUT);     /* BAD CODE */

while (wait_for_something() == RESULT_TIMEOUT) {}   /* Correct */
```

For dummy functions or empty callbacks, add a "NO-OP" comment.

对于空函数或空回调，应添加 "NO-OP" 注释：

Example (empty void function):

```
void empty_callback(...)
{
    /* NO-OP */
}
```

For an unfinished implementation, add a "TODO" comment, so that it can be found.

对于未完成的实现，添加一个"TODO"注释，以便可以找到它。

Example (unfinished function / placeholder):

```
void report_callback(...)
{
    // TODO
}
```

将复杂函数的每个参数放在单独的一行中

Put each function parameter of a complex function on a line of its own

This prevents the declaration from getting excessively long. It also allows each parameter to have a short comment.

这可以防止声明太长。它还允许每个参数有简短的注释。

Indent parameters by 8 spaces. The first parameter should be already in a separate line, not in the same line as the function name.

将参数缩进8个空格。第一个参数应该已经在单独的一行中，而不是与函数名在同一行。

Put the closing parenthesis on a separate line, followed by the opening brace on a separate line of its own.

将右括号放在单独的一行上，然后将左大括号放在单独的一行上。

Example (parameters of a complex function):

```
return_type complex_function(
        type_1 parameter_1,     /* comment, if needed */
        type_2 parameter_2,     /* comment, if needed */
        type_3 parameter_3,     /* comment, if needed */
        type_4 parameter_4      /* comment, if needed */
)
{
    ...
}
```

For a declaration, also put the closing parenthesis on a new line, followed by the semicolon on the same line.

对于声明，也要在新行上加上右括号，然后在同一行上加上分号。

This style also allows the comments to serve as Doxygen compatible descriptions of parameters.

这种风格还允许注释作为Doxygen兼容的参数描述。

Example (declaration with Doxygen descriptions of parameters):

```
return_type complex_function(
        type_1 parameter_1,     /**< Doxygen description */
        type_2 parameter_2,     /**< Doxygen description */
        type_3 parameter_3,     /**< Doxygen description */
        type_4 parameter_4      /**< Doxygen description */
);
```

When calling functions with many parameters or with complex values (calculated, function calls, etc), use the same indentation style, but keep the closing parenthesis with the last parameter.

当调用带有许多参数的函数或带有复杂值(计算值、函数调用等)的函数时，使用相同的缩进风格，但保留最后一个参数的右括号。

If the name of the function is short (less than 10 characters), keep the first parameter in the same line with the opening parenthesis and align the rest of the parameters with it.

如果函数名比较短(少于10个字符)，则第一个参数与左括号保持同一行，并将其他参数与左括号对齐。

For functions with a format string (e.g. logging or parsing), if more than one line is needed, it is recommended to align the parameters with the format string.

对于带有格式字符串的函数(例如日志记录或解析)，如果需要多行，建议将参数与格式字符串对齐。

Example (calling functions with many/long parameters):

```
ret = some_function(long_parameter_1,    /* WRONG, don't do this */
                    long_parameter_2,
                    long_parameter_3);

ret = some_function(                     /* Indent parameters by 8 spaces */
        long_parameter_1,
        long_parameter_2,
        long_parameter_3);

qsort(long_parameter_1,                  /* For functions with short names, indent
less */
      long_parameter_2,
      long_parameter_3,
      long_parameter_4);

LOG(TRACE, "Got values: %d %d %0.2f",    /* For format strings, align below */
            param_1, param_2, param_3)
```

Exceptions:

Note that this rule is for functions with many parameters, especially functions that are visible outside of a module, in which case it may be useful to use this style for all declarations in the header file (but not necessarily for all definitions in implementation file(s)).

例外情况：请注意，此规则适用于具有许多参数的函数，特别是在模块外部可见的函数，在这种情况下，对头文件中的所有声明使用此样式可能是有用的(但不一定适用于实现文件中的所有定义)。

Local (either static, or visible only within a module) functions can be declared in a single line, provided that the line does not get too long (up to 80-100 characters).

局部函数(静态函数或仅在模块内可见的函数)可以在单行中声明，前提是该行不会太长(不超过80-100个字符)。

Example (declaration of a local function):

```
                                         /* Allowed (if it is not too long) */
static int local_function(type_1 parameter_1, type_2 parameter_2);
```

对于声明指针，星号("*")应与标识符相邻

For declaring pointers, the asterisk ("*") shall be adjacent to the identifier

This rule applies to declaring variables, function parameters, and structure members.

此规则适用于声明变量、函数参数和结构成员。

Example (asterisk placement):

```
struct structure
{
    int counter;
    void *void_ptr;
}

int function(int parameter_1, void *parameter_2, struct structure *parameter_3)
{
    void *tmp_ptr;
    ...
}
```

If aligning the identifiers, then the alignment should include the asterisk ("*"):

如果对齐标识符，则对齐应包含星号("*")：

Example (asterisk alignment):

```
int         len;
char        *pos;
struct_t    *last;
bool        was_found;
```

Exceptions:

An exception to this rule are function return types, where the asterisk ("*") should be adjacent to the type.

该规则的一个例外是函数返回类型，其中星号("*")应该与类型相邻。

The return type should be specified on the same line as the function (helpful when 'grepping').

返回类型应该在函数的同一行上指定(在使用grepping时很有用)。

Exception (asterisk placement for return types):

```
void* function_returning_a_void_pointer(...)
{
    ...
}
```

In header files, groups of simple functions may be written as one-liners.

在头文件中，一组简单的函数可以写成单行代码。

It is not mandatory, but it may be useful to align the function names. If aligning, use standard indents (e.g. column 8, 12, 16...), and do not try to align all functions to the longest return type.

这不是强制的，但对齐函数名可能是有用的。如果对齐，请使用标准缩进(例如列8、12、16...)，不要尝试将所有函数对齐到最长返回类型。

If there are a few functions with a longer return type, let those be exceptions. If it makes sense, group them together, but do not align everything because of those few.

如果有一些函数的返回类型较长，则将其视为异常。如果有意义，将它们放在一起，但不要因为这些少数人而将所有内容对齐。

Example (alignment of prototype one-liners):

```
int     function_returning_an_int(...);
void    function_not_returning_anything(...);
void*   function_returning_a_void_pointer(...);
char*   function_returning_a_char_pointer(...);
info_struct_t*  function_returning_a_pointer_to_something(...);
```

二元算术运算符、位运算符和赋值运算符的两侧必须留空格

Binary arithmetic, bitwise, and assignment operators should be surrounded by spaces

Binary (as opposed to unary) arithmetic, bitwise, and assignment operators should be surrounded by spaces.

二元(相对于一元)运算符、位运算符和赋值运算符的两侧必须留空格。

Example (arithmetic, bitwise, and assignment operators):

```
size = (len + 3) & ~0x0003;     /* Right */
size *= sizeof(int);            /* Right */

size=(len+3)&~0x0003;           /* WRONG */
size*=sizeof(int);              /* WRONG */
```

Exceptions:

Very simple assignments and expressions (using local variables with short names and simple literals) may be written without spaces, especially in "for" statements.

非常简单的赋值和表达式(使用具有短名称和简单字面量的局部变量)可以不使用空格，特别是在"for"语句中。

Exception (omitting spaces for very simple operations):

```
for (i = 0; i < table_length; i++) ...                  /* As per the rules */

for (i=0; i<table_length; i++) ...                      /* Also allowed */

for (idx=0; idx<json_array_size(result); idx++) ...     /* WRONG, too complex */
```

三元条件运算符两侧必须留空格

Ternary conditional operators should be surrounded by spaces

In a ternary conditional operation ("? ... : ..."), both the "?" and the ":" should be surrounded by spaces. Additionally, always enclose the condition in parentheses, so that it resembles the condition of an "if" statement.

在三元条件运算("? ... : ...")中，"?" 和 ":" 两侧都应留空格。此外，始终用括号把条件括起来，使其看起来就像 if 语句中的条件。

Example (ternary operator):

```
retval = (len < 0) ? -1 : len;      /* Right */

retval=(len<0)?-1:len;              /* WRONG */
retval = len < 0 ? -1 : len;        /* WRONG */
```

一元运算符不能和空格一起使用

Unary operators must not be used with spaces

Unary operators must not be used with spaces.

一元运算符不能和空格一起使用

Example (unary operators):

```
a = -a;                 /* Right */
i++;                    /* Right */
if (!is_local) ...      /* Right */

a = - a;                /* WRONG */
i ++;                   /* WRONG */
if (! is_local) ...     /* WRONG */
```

逗号或分号前不要使用空格

Do not use spaces before a comma or a semicolon

Do not place spaces before a comma or a semicolon. This is also true for the 'comma' operator (but note that it should be used sparingly).

不要在逗号或分号前加空格。对于逗号操作符也是如此(但请注意，应该谨慎使用)。

Example (no spaces before commas or semicolons):

```
for (i=0; i<table_length; i++)      /* Right */
{
    process(element[i], true);      /* Right */
}

for (i=0 ; i<table_length ; i++)    /* WRONG */
{
    process(element[i] , true);     /* WRONG */
}
```

在控制语句及其括号之间使用空格

Use spaces between control statements and their parentheses

Place spaces between control statements and their parentheses.

在控制语句和它们的括号之间放置空格。

Example (spaces after control statements):

```
if (condition) ...          /* Right */

if(condition) ...           /* WRONG */

for (i=0; i<len; i++) ...   /* Right */

for(i=0; i<len; i++) ...    /* WRONG */
```

函数参数的括号中不能有空格

Parentheses of function parameters should be written without any spaces

Do not place spaces between a function and its parentheses, or between a parenthesis and its content.

不要在函数和圆括号之间，或者圆括号和内容之间放置空格。

Example (no spaces around function's parentheses):

```
function_name(parameter_1, parameter_2);    /* Right */

function_name (parameter_1, parameter_2);   /* WRONG */

function_name( parameter_1, parameter_2 );  /* WRONG */
```

简单的return语句不应该使用括号

Simple return statements should not use parentheses

Return statements should not use parentheses for simple return values (literals, variables, simple function calls).

Return语句不应该对简单的返回值(字面量、变量、简单的函数调用)使用括号。

If a return expression uses operators, it may be enclosed in parentheses, and there should be a space between the "return" keyword and the parenthesis.

如果返回表达式使用了操作符，可以用括号括起来，而且"return"关键字和括号之间应该有一个空格。

Example ("return" statements):

```
return 1;                           /* Right */

return retval;                      /* Right */

return helper_function(...);        /* Right */

return (length * sizeof(long));     /* Right */

return(length * sizeof(long));      /* WRONG */

return (0);                         /* WRONG */

return (retval);                    /* WRONG */
```

初始化方法的大括号内部应该有额外的空格

Curly braces of an initializer should have an extra space on the inner side

When curly braces are used with their content on the same line (e.g. for a simple initializer), use a space on the inner side.

当花括号和它们的内容在同一行时(例如，对于简单初始化程序)，在内部使用空格。

Example (initializers):

```
struct some_state my_state = { true, false };       /* Recommended */

int array[MAX_ELEMENTS] = { 0 };                    /* Recommended */

int array[MAX_ELEMENTS] = {0};                      /* Compact, allowed */

int array[MAX_ELEMENTS] = { };                      /* WRONG, may produce warnings
*/

struct some_struct array_s[ARRAY_SIZE] = { 0 };     /* WRONG, may produce warnings
*/
```

```
struct some_struct array_s[ARRAY_SIZE] = { { 0 } }; /* Correct for an array of
structs */
```

对于复杂的表达式，请使用括号和额外的空格

Use parentheses and extra spacing for complex expressions

Parentheses should be used generously. Relying on operator precedence often leads to mistakes, and such code is also harder to read when reviewing.

括号应该大方使用。依赖运算符优先级通常会导致错误，并且在审阅时，这样的代码也很难阅读。

When two parentheses are adjacent to each other, adding an extra space usually makes the expression easier to read, especially if the operands are long (e.g. function calls).

当两个括号相邻时，添加额外的空格通常会使表达式更容易阅读，特别是当操作数很长时(例如函数调用)。

Example (parentheses for complex expressions):

```
result = (a * b) + (c * d);     /* Right */

result = a * b + c * d;         /* WRONG */

if ( (a * 4) <= (b + 1) ) ...   /* Right */

if (a * 4 <= b + 1) ...         /* WRONG */

if ((a * 4) <= (b + 1)) ...     /* Allowed, but less readable */
```

在多行中编写复杂的表达式

Writing complex expressions over several lines

Arithmetic expressions can (and should) be simplified using intermediate results, either by accumulating the result into the final variable, or by using additional temporary variables.

算术表达式可以(也应该)使用中间结果进行简化，可以将结果累加到final变量中，也可以使用额外的临时变量。

For boolean expressions, the usual approach is to break them into multiple lines. Use line breaks at a consistent nesting level, and put the operator at the beginning of the next line.

对于布尔表达式，通常的做法是将它们分成多行。在一致的嵌套级别上使用换行符，并将该操作符放在下一行的开头。

Example (multi-line boolean expression, compact):

```
if (long_condition_1
    || long_condition_2
    || (condition_3 && condition_4))
```

```
{
...
```

The above compact example can be made more readable by aligning the individual expressions and by moving the last parenthesis to the next line, aligned with its opening parenthesis.

通过对齐各个表达式，并将最后一个括号移动到下一行，使上述紧凑的示例更具可读性。

Example (multi-line boolean expression, aligned):

```
if (   long_condition_1
    || long_condition_2
    || (condition_3 && condition_4)
)
{
...
```

每行不应超过120个字符

Lines should not exceed 120 characters

Older coding styles usually only allow lines of up to 78 or 80 characters.

旧的编码风格通常只允许每行最多78或80个字符。

Nowadays such a limitation can be counterproductive. But bear in mind that although editors and terminal output can handle long lines fairly well, various diff tools and especially web tools (e.g. GitHub) may not be so flexible, so try to keep the amount of long lines low.

如今，这样的限制可能会适得其反。但是请记住，尽管编辑器和终端输出可以很好地处理长行，但各种diff工具，特别是web工具(例如GitHub)可能不是那么灵活，所以尽量保持长行数量少。

Make sure to follow recommendations elsewhere in this coding style so that indentation levels do not grow unnecessarily:

请务必遵循此编码风格中其他地方的建议，以免不必要地增加缩进级别：

- Write parameters of complex functions each in its own line 将复杂函数的参数写在单独的一行中
- Use "goto" statements to simplify error handling, cleanup of resources, etc. 使用"goto”语句来简化错误处理、资源清理等。
- Avoid redundant "else" and "else if" statements 避免多余的else和else if语句
- Test for failure rather than success to avoid unnecessary nesting 测试失败而不是成功，以避免不必要的嵌套

If the code starts to stray too much to the right, it is probably time to refactor the code and introduce some helper functions.

如果代码开始向右偏离太多，那么可能是时候重构代码并引入一些辅助函数了。

Note however that function header comments and other comments spanning multiple lines should be limited to 78 characters (see also the "Comments" section).

不过要注意，函数头注释和其他跨行注释应该限制在78个字符以内(参见"注释"一节)。

文件

Files

将文件名区分大小写

Treat filenames as case sensitive

This rule seems redundant for developers using Linux for their development environment, but it is a reminder in case any developer uses Windows or OS X.

对于使用Linux作为开发环境的开发人员来说，这条规则似乎是多余的，但对于使用Windows或OS X的开发人员来说，这是一个提醒。

C源文件的扩展名必须是".c"

C source files must have extension ".c"

Example: `util.c`

C头文件的扩展名必须是".h"

C header files must have extension ".h"

Example: `util.h`

在'内联定义文件'中定义内联函数(扩展名".c.h")

Define inline functions in 'inline definition files' (extension ".c.h")

Define inline functions in 'inline definition files' (extension ".c.h") and declare them in header files

在内联定义文件(扩展名".c.h")中定义内联函数，并在头文件中声明它们

Example: `ds_tree.c.h`, `ds_tree.h`

The "inline" keyword must be used in both places.

"inline"关键字必须在这两个地方都使用。

A separate inline file keeps header files clean and small. It also allows the project to be adjusted to not use inlining, either for performance reasons or for troubleshooting purposes.

一个单独的内联文件可以保持头文件的整洁和小巧。它还允许调整项目以不使用内联，无论是出于性能原因还是故障排除目的。

头文件必须包含'包含保护'

Header files must have '#include guards'

Include guards are used to prevent the problem of 'double inclusion'.

使用包含保护来防止"重复包含"问题。

Include guards should be used consistently and preemptively, not only when double inclusion arises.

包含保护应始终如一地、预先地使用，而不仅仅是在出现双重包含时。

Example (#include guard):

```
#ifndef FILE_H_INCLUDED
#define FILE_H_INCLUDED
...
...
...
#endif /* FILE_H_INCLUDED */
```

"#include guard" 的预处理器符号名称应组成为：<filename_without_extension>_H_INCLUDED

The name of the preprocessor symbol for the '#include guard' shall be composed as: <filename_without_extension>_H_INCLUDED

Example: `STRUTIL_H_INCLUDED`

Note:

Another common naming scheme uses just `<filename_without_extension>_H` (for example STRUTIL_H). This naming scheme was specifically NOT chosen and should not be used.

另一种常见的命名方案是仅使用 `<filename_without_extension>_H`(例如 STRUTIL_H)。此命名方案并非特意选择，不应使用。

The naming scheme that uses `__<filename_without_extension>_H__` (for example **STRUTIL_H**) is used by compilers and standard C libraries and MUST NOT be used (symbols that start with two underscores are reserved for the compiler).

编译器和标准 C 库使用 `__<filename_without_extension>_H__`(例如 **STRUTIL_H**)的命名方案，因此不得使用(以两个下划线开头的符号为编译器保留)。

头文件应自包含

Interface header files should be self-contained

Each interface header file should be self-contained, meaning that when it is included, there are no dependencies that would need to be included before it.

每个头文件都应自包含，这意味着当它被包含时，无需在其之前添加任何依赖项。

An interface header file should declare only the exposed interface and any types needed for it. It should not declare anything that is only needed for the module's implementation.

头文件应仅声明公开的接口及其所需的任何类型。它不应声明任何仅用于模块实现所需的内容。

The implementation file (or files, if there are more than one) must always include the interface header file.

实现文件(或多个文件，如果有多个)必须始终包含头文件。

By convention, system headers are included first, followed by 3rd-party headers (if any). Project headers are included last.

按照惯例，首先包含系统头文件，然后包含第三方头文件(如果有)。最后包含项目头文件。

Example (foobar.c):

```
#include <limits.h>
#include <errno.h>

#include "util.h"
#include "os_util.h"

#include "foobar.h" /* interface */
...
```

To test whether the interface header is really self-contained, one can temporarily include the interface header at the top of the implementation file (but make sure it is removed before committing).

为了测试接口头是否真正自包含，可以暂时将接口头包含在实现文件的顶部(但确保在提交之前将其删除)。

Exceptions:

A header file may be defined that is included before any other header files:

可以定义一个头文件，使其在任何其他头文件之前被包含：

- Such a header file can be used to resolve issues that have different solutions on different platforms, or address differences between compilers and compiler versions. 此类头文件可用于解决在不同平台上有不同解决方案的问题，或解决不同编译器和编译器版本之间的差异。
- If such a header file is used, source files and other header files may rely on this header file to include certain prerequisite header files. 如果使用此类头文件，源文件和其他头文件可能依赖此头文件来包含某些先决条件头文件。

Another example of a header file that is not an interface header file is an 'internal to implementation' header file:

另一个非头文件的示例是"实现内部”头文件：

- Such an 'implementation' header file is useful when a module is implemented in several source files. 当一个模块在多个源文件中实现时，这种"实现"头文件非常有用。
- An 'implementation' header file can be used to share types and declarations that are needed by several source files, but should not be exposed in the interface header file. "实现"头文件可用于共享多个源文件所需的类型和声明，但不应在头文件中公开。
- The name of an 'implementation' header file should have an "_impl" suffix (e.g. foobar_impl.h) "实现"头文件的名称应带有"_impl"后缀(例如 foobar_impl.h)。

In some cases including 3rd-party headers may be tricky because of their prerequisites, or if they cause name clashes:

在某些情况下，包含第三方头文件可能会比较棘手，因为它们有先决条件，或者会导致名称冲突：

- Whenever possible, 3rd-party headers should only be used in implementation files and the presence of the 3rd-party types hidden from the caller. 尽可能仅在实现文件中使用第三方头文件，并且第三方类型的存在对调用者隐藏。
- A decision can be made to make an exception for certain 3rd-party headers, which are then included according to specific rules. 可以决定对某些第三方头文件进行例外处理，然后根据特定规则将其包含在内。

使用括号 (<>) 包含系统头文件，使用引号 ("") 包含项目和第三方头文件

Use brackets (<>) to include system header files and quotes ("") to include project and 3rd-party header files

Using brackets versus quotes in include directives affects the search order.

在 include 指令中使用括号还是引号会影响搜索顺序。

Not using them properly affects compilation performance, or may even fail to compile on some compilers (or in some circumstances).

不正确使用括号会影响编译性能，甚至可能在某些编译器上(或在某些情况下)编译失败。

将"#include"指令放在文件顶部

Put the "#include" directives at the top of a file

All #include directives should be grouped at the top of the file.

所有 #include 指令都应放在文件顶部。

Using an #include directive at the point where it is needed is only used in specific cases and should generally be avoided.

仅在特定情况下才在需要的地方使用 #include 指令，通常应避免使用。

注释

Comments

所有注释必须使用英文编写

Comments must be written exclusively in English

Never use any language other than English in source code.

源代码中切勿使用英语以外的任何语言。

注释必须严格遵循专业规范

Comments must be strictly professional

Use comments to describe the code. Any other use of comments is unprofessional.

使用注释来描述代码。任何其他形式的注释都是不专业的。

充分利用垂直空间

Use vertical space generously

Just like good comments, empty lines make the code more readable.

就像好的注释一样，空行也能提高代码的可读性。

Use an empty line after the declarations, before the first executable statement.

在声明之后、第一个可执行语句之前使用一个空行。

Always add an empty line before a comment.

注释前务必添加一个空行。

Separate logical groups of code with empty lines whether they are preceded by a comment or not.

无论代码逻辑组前面是否有注释，都应使用空行分隔它们。

代码注释最好使用"C++ 风格"("//")注释

'C++ style' ("//") comments are preferred for commenting the code

'C++ style' ("//") comments are easier to format and maintain, and are therefore preferred for general comments in the code.

"C++ 风格"("//")注释更易于格式化和维护，因此更适合用于代码中的常规注释。

The more prominent multi-line comments using "/.../" style are used for function headers, and occasionally for comments indicating workarounds or some other important information that should stand out.

使用"/.../"风格的多行注释更为醒目，通常用于函数头，有时也用于指示解决方法或其他一些需要突出显示的重要信息的注释。

Example (comment styles):

```
/**
 * Standard format for functions headers
 * (starts with two asterisks if it is intended to be recognised by Doxygen)
 */

// The '//' style comments are easier to format
// and are therefore preferred for general comments in the code
```

较大的注释应位于它们所描述的行(或块)上方，缩进相同

Larger comments should be above the line (or block) they describe, indented identically

A comment before a control statement should describe either just the condition (or scope in case of a "for" statement) of the action, or the intent of the action (including the controlled statements).

控制语句前的注释应该仅描述操作的条件(或"for”语句的范围)，或者操作的意图(包括受控语句)。

Example (a comment describing the whole loop):

```
// verify all elements and log failures
for (...)
{
    ...
}
```

A comment at the top of a controlled statement (within an "if", "else", "for", ...) should describe the particular action taken. It can also clarify the reason why that code is reached.

受控语句顶部的注释(在"if”、"else”、"for”等语句内)应该描述所采取的具体操作。它还可以阐明到达该代码的原因。

Example (comments for individual controlled blocks):

```
// check configuration options and thresholds
if (... complex_condition ...) {
    // log and abort
    ...
} else {
    // conditions not met, emit a warning before proceeding
    ...
}
```

"行尾”注释可用于对操作进行简要描述

'End-of-line' comments can be used for brief descriptions of actions

'End-of-line' comments should only be used when the comment is brief enough to fit comfortably in a single line.

仅当注释足够简短以适合一行时才应使用"行尾"注释。

Example ('end-of-line' comments):

```
for (...)
{
    ...
    if (condition1) break;   // Right: brief comment
    ...
    ...
    if (condition2) break;  // WRONG: don't use end-of-line style
                            // for long comments that
                            // don't fit in a single line
    ...
    ...
    if (condition3)
    {
        // Right: if a long explanation is needed,
        // use a normal comment rather
        // than an 'end-of-line' comment
        break;
    }
    ...
}
```

"行尾"注释应与代码之间用两个空格分隔

'End-of-line' comments should be separated from the code by two (2) spaces

Use two spaces rather than one before an 'end-of-line' comment. This separates it from the code and makes both the code and the comment more readable.

"行尾"注释前应使用两个空格而不是一个空格。这样可以将注释与代码分隔开来，使代码和注释都更具可读性。

Example (use two spaces before an 'end-of-line' comment):

```
                                                /* Right */
do_something(...);  // will return immediately
```

```
                                                /* WRONG */
do_something(...); // a single space makes it less readable
```

For multiple, related actions, the 'end-of-line' comments may be aligned (provided that they don't stray too far to the right). The minimum allowed distance of two spaces from the code still applies.

对于多个相关操作，"行尾"注释可以对齐(前提是它们不会向右偏离太远)。与代码之间允许的最小距离为两个空格仍然适用。

Example (aligning 'end-of-line' comments):

```
if (condition_1) {
    do_something();             // log if configured
} else if (condition_2) {
    do_something_else();        // not fatal, but logged unconditionally
} else {
    unhandled_state_action();   // will log and abort (exit)
}
```

Exceptions:

A single space is recommended for 'end-of-block comments' (described next).

建议在"块尾注释"(下文介绍)中使用单个空格。

对大型控制语句块使用"块尾"注释

Use an 'end-of-block' comment for large control statement blocks

If a control statement block is large (e.g. spans more than one screen), or includes several levels of nested control statements, then it is hard to quickly deduct which closing brace belongs to which block.

如果控制语句块很大(例如，跨越多个屏幕)，或者包含多层嵌套的控制语句，则很难快速推断出哪个右括号属于哪个块。

Generally, in such situations one should consider refactoring the code (e.g. by introducing helper functions) to keep functions – and especially control statement blocks – down to a reasonable size.

通常，在这种情况下，应该考虑重构代码(例如，通过引入辅助函数)以将函数(尤其是控制语句块)的大小控制在合理的范围内。

However, it is not always possible (or worthwhile) to restructure the code, and in such cases it is advisable to add a short comment (an 'end-of-block' comment) that clarifies what the closing brace belongs to.

但是，重构代码并不总是可行(或值得)，在这种情况下，建议添加一个简短的注释("块尾"注释)，以阐明右括号的归属。

To make it different from typical 'end-of-line' comments, use the "/* ... */" comment style and a single space between the closing brace and the comment. This way the entire construct functions as a whole, and even somewhat resembles the "end if" and "end loop" statements of Ada.

为了使其与典型的"行尾"注释区分开来，请使用"/* ... */"注释样式，并在右括号和注释之间添加一个空格。这样，整个结构就可以作为一个整体发挥作用，甚至有点类似于 Ada 的"end if"和"end loop"语句。

Example ('end-of-block' comments for control statements):

```
        ...
    } /* for */
} /* while */
```

对大型预处理块使用"块结束”注释

Use an 'end-of-block' comment for large preprocessor blocks

Just like with large control statement blocks, there is a readability problem with large preprocessor blocks (#if, #ifdef, #ifndef).

与大型控制语句块一样，大型预处理块(#if、#ifdef、#ifndef)也存在可读性问题。

So, likewise, use 'end-of-block' comments for large preprocessor blocks (especially those that do not fit on one screen), and put the preprocessor symbol into the comment.

因此，同样，对大型预处理块(尤其是那些无法在一个屏幕上显示的块)使用"块结束”注释，并将预处理符号放入注释中。

Again, use the "/* ... */" comment style and a single space between the closing brace and the comment.

同样，请使用"/* ... */”注释样式，并在右括号和注释之间添加一个空格。

Example ('end-of-block' comment for a preprocessor block):

```
#ifdef TARGET_NATIVE
    ...
    ...
    ...
#endif /* TARGET_NATIVE */
```

For complex conditions, do not repeat the entire condition, just use a suitable distinguishing symbol from the condition (so one can search for it) and add an ellipsis ("...") after it.

对于复杂的条件，不要重复整个条件，只需使用适合该条件的区分符号(以便可以搜索它)并在其后添加省略号("...”)。

Example ('end-of-block' comment for a complex condition):

```
#if (__GNUC__ == 4 && (__GNUC_MINOR__ > 3))
    ...
    ...
    ...
#endif /* __GNUC__ ... */
```

For preprocessor conditional blocks with an "#else" directive, it is useful to add "(else)" in the end-of-block comments of the "#else" and "#endif" directives.

对于带有"#else"指令的预处理器条件块，在"#else"和"#endif"指令的块结束注释中添加"(else)"很有用。

Example ('end-of-block' comments with "#else" directives):

```
#if defined(SO_NOSIGPIPE)
    ...
    ...
#else /* SO_NOSIGPIPE (else) */
    ...
    ...
#endif /* SO_NOSIGPIPE (else) */
```

Do not overuse 'end-of-block' comments. Only use them for larger blocks, never for blocks containing a single line or just a couple of lines. Rather, always make the blocks stand out by using empty lines before and after the conditional directives.

不要过度使用"块结尾"注释。它们只用于较大的块，切勿用于包含一行或几行的块。相反，应始终在条件指令前后使用空行来突出块。

Example (no 'end-of-block' comments for short blocks):

```
#ifndef ARRAY_SIZE
#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))
#endif /* ARRAY_SIZE */                                  /* WRONG */

#ifndef ARRAY_SIZE
#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))
#endif                                                   /* Right */
```

使用空格将注释内容与"//"、"/"、""或"*/"分隔开

Separate the content of the comment from any of "//", "/", "", or "*/" with a space

Example (use spaces around the content of a comment):

```
/*
 * a longer comment,
 * spanning multiple lines
 */                                                     /* Right */

// a longer comment
// using the '//' style                                 /* Right, recommended */

/* an in-line comment */                                /* Right */
```

```
// a short comment                                          /* Right */

/*
 *a longer comment,
 *spanning multiple lines
 */                                                         /* WRONG */

//a longer comment
//using the '//' style                                     /* WRONG */

/*an in-line comment*/                                     /* WRONG */

//a short comment                                          /* WRONG */
```

Exceptions:

An exception is made when commenting out lines of code, described in the next rule.

注释掉代码行时有一个例外，详见下一条规则。

使用"#if 0”而不是"/* ... */”来禁用代码块

Use "#if 0" instead of "/* ... */" to disable blocks of code

If a block of code needs to be disabled temporarily, it is better to use "#if 0" than to comment it out using "/*" *and* "*/".

如果需要暂时禁用某个代码块，最好使用"#if 0”，而不是使用"/*”*和*"*/”将其注释掉。

This works without problems even if the code that needs to be disabled contains "/* ... */" style comments.

即使需要禁用的代码包含"/* ... */”类型的注释，这种方法也能正常工作。

Additionally, by using an "#else" directive, it is possible to easily switch between two implementations (change "#if 0" to "#if 1").

此外，通过使用"#else”指令，可以轻松地在两种实现之间切换(将"#if 0”更改为"#if 1”)。

Always add a "/* 0 */" comment after the "#endif".

务必在"#endif”后添加"/* 0 */”注释。

Example (disabling code with "#if 0"):

```
#if 0
    ...
    ...
    /* no problems with existing
    multi-line comments */
    ...
    ...
#endif /* 0 */
```

Exceptions:

To disable a single line of code, just use "//".

要禁用单行代码，只需使用"//"。

To make it different from regular comments, do not insert a space after the "//" (this also makes it possible to find such code to clean-up the sources).

为了与常规注释区分开来，请勿在"//"后插入空格(这样也更容易找到此类代码来清理源代码)。

Example (disabling a single line with "//"):

```
...
//if (!connected) goto retry_connect;
...
```

废弃的代码应该直接删除

Remove Obsolete Code Immediately

废弃的代码应该直接删除，除非有充分的理由(比如需要等待server-side API就绪)才用#if 0注释。

Obsolete code shall be deleted outright unless there is a compelling justification (such as waiting for a server-side API to become ready) in which case it may be temporarily disabled with #if 0.

Example (use #if 0 to save code for future use):

```
#if 0  /* TODO: activate after server-side API merge */
    ...
    new_foo_api_call();
    ...
#endif
```

每个函数都应该有一个描述其用途的注释

Every function should have a comment that describes its purpose

Ideally, every function should have a Doxygen compatible comment.

理想情况下，每个函数都应该有一个与 Doxygen 兼容的注释。

Functions that are not exposed from modules may be described just with short descriptions if the purpose of the parameters is obvious enough. For related functions, instead of repeating the purpose of the parameters, highlight the differences and/or relationships between the functions.

如果参数的用途足够明显，未从模块公开的函数可以仅使用简短描述。对于相关函数，无需重复参数的用途，而是突出显示函数之间的差异和/或关系。

Note: even a short description is better than no description.

注意：即使是简短的描述也比没有描述要好。

仅在适用的情况下使用 Doxygen 风格的注释

Use Doxygen style comments only where applicable

Doxygen recognizes and processes comments written as "/** ... */" or "/// ...".

Doxygen 可以识别并处理以"/** ... */"或"/// ..."形式编写的注释。

Do not use this style for comments within the body of a function.

请勿将此风格用于函数主体内的注释。

名称

Names

所有名称均应使用英语

Always use English for all names

Never use any language other than English for any names, even if they are not visible to the user.

切勿使用英语以外的任何语言命名任何名称，即使用户看不到它们。

This applies not only to names in the code, but also to filenames, database entities and so on.

这不仅适用于代码中的名称，也适用于文件名、数据库实体等。

使用描述性名称

Use descriptive names

Names should indicate the content or the intent.

名称应表明内容或意图。

Do not use cryptic names or names based on internal jokes.

不要使用神秘的名字或基于内部笑话的名字。

Look at the surrounding code and try to use the same approach.

查看周围的代码并尝试使用相同的方法。

Although most symbol names are not visible in the compiled code, try not to misspell them.

尽管大多数符号名称在编译后的代码中是不可见的，但尽量不要拼错。

Misspelled identifiers can be distracting and can cause confusion when searching for something.

拼写错误的标识符可能会分散注意力，并在搜索时造成混淆。

具有较大作用域的变量必须具有较长的名称，而具有较小作用域的变量可以具有较短的名称

Variables with a large scope must have long names, variables with a small scope may have short names

Variables with a large scope should have long, descriptive names.

作用域大的变量应该有很长的描述性名称。

The same is also true for a variable with a smaller scope, if it is used in only a few places.

对于作用域较小的变量，如果只在少数地方使用，也同样适用。

Local variables can (and should) have shorter names.

局部变量可以(也应该)使用更短的名称。

Variables for loops and iteration should typically have quite short names, but should follow some conventions (e.g. i, j for integers, ptr or pos for general pointers, item or iter for pointers to structures or other objects).

用于循环和迭代的变量通常应该有相当短的名称，但应该遵循一些约定(例如，i、j表示整数，ptr或pos表示一般指针，item或iter表示指向结构体或其他对象的指针)。

Temporary variables should also have short, conventional names (e.g. result, lresult, bresult, ret, tmp, tmp2, ch).

临时变量也应该有简短的、常规的名称(例如result、lresult、bresult、ret、tmp、tmp2、ch)。

A variable that stores the intended return value should always be named "retval".

存储预期返回值的变量应该命名为"retval”。

对不同模块中声明的标识符使用前缀

Use prefixes for identifiers declared in different modules

Both libraries and other modules evolve over time. Prefixing externally visible identifiers with the name of the module avoids name clashes.

库和其他模块都会随着时间的推移而演变。为外部可见的标识符添加模块名前缀，可以避免名称冲突。

Often the same name that is used for prefixing is also the directory name of the module.

通常用作前缀的名称也是模块的目录名。

Typically the module name is a three-letter abbreviation or acronym. For libraries it may be longer, like 5 to 7 letters (e.g. "libos").

通常，模块名是三个字母的缩写或首字母缩写。对于库来说，它可能更长，比如5到7个字母(例如："libos”)。

命名约定

Naming conventions

Entity	Examples
Global variables shall be prefixed with "g_"	g_foo_bar
Local variables shall be lower case	foo, bar, foobar, foo_bar
Parameter variables shall be lower case and words shall be separated by underscores ("_")	foo_bar
Functions shall be lower case and words shall be separated by underscores ("_")	foo_bar()
Types shall be lower case and words shall be separated by underscores ("_")	foo_bar
Typedefs shall be suffixed with "_t"	foo_bar_t
Enumerators shall be upper case and words shall be separated by underscores ("_")	FOO_BAR
Macros shall be upper case and words shall be separated by underscores ("_")	FOO_BAR

实体	示例
全局变量前缀为"g_"	g_foo_bar
局部变量应为小写	foo, bar, foobar, foo_bar
参数变量小写，单词之间用下划线分隔("_")	foo_bar
函数小写，单词以下划线("")分隔("")	foo_bar()
类型应为小写，单词之间以下划线("")分隔("")	foo_bar
类型定义(Typedefs)应以"_t"为后缀	foo_bar_t
枚举数应为大写，单词应以下划线("_")分隔	FOO_BAR
宏必须大写，单词之间用下划线("")分隔("")	FOO_BAR

声明

Declarations

在单独的声明中声明每个变量

Declare each variable in a separate declaration

Declaring each variable in a separate declaration has several advantages:

分别声明两个变量有以下优点。

- There can be no confusion with declarations like "int *p, i;", where one might mistakenly assume that "i" is a pointer. 不能与"int *p, i；"这样的声明混淆，人们可能会错误地认为"i"是一个指针。

- If some declarations declare just a single variable and some declare multiple variables, it is easy to overlook the additional variables at a glance. 如果有些声明只声明一个变量，有些声明多个变量，那么很容易一眼就忽略额外的变量。
- If each variable is declared separately, then 'end-of-line' comments to describe the purpose of each variable can be added easily and consistently. 如果每个变量都是单独声明的，那么可以轻松且一致地添加`end- line`注释来描述每个变量的用途。

在函数声明中提供参数名

Provide names of parameters in function declarations

Although compilers allow the function declarations (in header files) to list just the parameter types without the names of parameters, it is a good practice to also provide parameter names in declarations.

虽然编译器允许函数声明(在头文件中)只列出参数类型而不列出参数名，但最好在声明中也提供参数名。

Often parameter names themselves provide enough information so that one can understand them without additional documentation. Modern IDEs also use information from function declarations for features like 'autocomplete' and 'peek declaration'.

通常，参数名本身就提供了足够的信息，因此无需额外的文档就可以理解它们。现代ide也使用来自函数声明的信息来实现像 '自动补全(autocomplete)' 和 '速览声明(peek declaration)' 这样的功能。

绝对不能使用隐式的"int”返回类型

Implicit "int" return type must never be used

C89 standard still allowed functions to be defined without the return type, in which case "int" was assumed, but it was always considered a bad practice to omit the return type for functions returning an "int".

C89标准仍然允许定义没有返回类型的函数，在这种情况下假定为"int”，但对于返回"int”的函数省略返回类型始终被认为是一个不好的实践。

C99 standard no longer allows this archaic feature.

C99标准不再允许这种过时的功能。

使用"(void)”声明没有参数的函数，而不是使用空的参数列表

Declare functions with no parameters using "(void)" instead of empty parameter list

An obsolescent feature of C allows functions declared with no parameters (empty parameter list) to be called with parameters.

C语言的一个过时的特性是允许使用参数调用声明没有参数的函数(空参数列表)。

Functions which accept no parameters should be declared using "(void)". This way the compiler will issue an error if the function is mistakenly called with an argument.

不接受参数的函数应该使用"(void)”声明。这样，如果错误地使用实参调用函数，编译器就会报错。

Example (function with no parameters):

```
int function_requiring_no_parameters();          /* WRONG */

int function_requiring_no_parameters(void);      /* Right */
```

对于指向函数的指针，使用typedefs

Use typedefs for pointers to functions

Pointers to functions is a feature that is not used daily, and the syntax is far from 'easy on the eyes'. Also, declaring this type in several places can quickly lead to inconsistencies.

函数指针不是一个日常使用的功能，它的语法也远非"简单易懂”。此外，在多个地方声明这种类型很快就会导致不一致。

It is a common practice to use a typedef to define a type for the pointer to the function, which makes subsequent use of this type much easier.

通常使用typedef为指向函数的指针定义类型，这使得后续使用该类型容易得多。

Example (typedef for a pointer to function):

```
// pointer to callback type
typedef void (*some_pfn_t)(void *ctx, int status);

// callback implementation declaration
void dummy_callback(void *ctx, int status);

static some_pfn_t g_some_cb;

void set_some_cb(some_pfn_t p_callback)
{
    g_some_cb = p_callback;
}

void dummy_callback(void *ctx, int status)
{
    /* NO-OP */
}

set_some_cb(dummy_callback);
```

Another method is slightly different in that it defines the type for the function signature, and then declares variables or parameters as pointers to that type.

另一个方法稍有不同，它定义了函数签名的类型，然后将变量或形参声明为指向该类型的指针。

Example (typedef for the function signature):

```
// callback function type
typedef void some_fn_t(void *ctx, int status);

// callback implementation declaration
some_fn_t dummy_callback;

static some_fn_t *g_some_cb;

void set_some_cb(some_fn_t *p_callback)
{
    g_some_cb = p_callback;
}

void dummy_callback(void *ctx, int status)
{
    /* NO-OP */
}
set_some_cb(dummy_callback);
```

The second approach has two benefits:

第二种方法有两个好处：

- Variables (like "g_some_cb" in the example) and parameters (like "p_callback" in the example) are declared with an asterisk, which makes it more clear that they are pointers. 变量(如示例中的"g_some_cb")和参数(如示例中的"p_callback")使用星号声明，这使它们是指针更加清晰。
- Function implementations (like "dummy_callback" in the example) are declared with the typedefed type, therefore the compiler actually verifies that the types match, even for implementations of functions that are not used within the module (e.g. are provided for other modules). 函数的实现(例如例子中的"dummy_callback")是用typedefed类型声明的，因此编译器实际上会验证类型是否匹配，即使是对于模块中没有使用的函数的实现(例如，为其他模块提供的函数)。

As seen in the example, the function type should have a "_fn_t" suffix, while function implementations, variables, and parameters usually have a suffix that indicates what they are used for (e.g. "_cb" for callbacks, "_cmp" for comparison functions, etc).

如上例所示，函数类型应该有一个"_fn_t"后缀，而函数实现、变量和形参通常都有一个后缀，表示它们的用途(例如："_cb"表示回调函数，"_cmp"表示比较函数，等等)。

Note:

Functions are similar to arrays in that a function name without the parentheses is interpreted to mean the address of the function. The ampersand operator therefore does not need to be used (and makes no difference if it is). The following two lines are equivalent:

函数和数组类似，没有括号的函数名会被解释为函数的地址。因此不需要使用&操作符(即使使用也没有区别)。下面两行代码是等价的：

Example (ampersand operator is not needed for obtaining pointers to functions):

```
set_some_cb(dummy_callback);

set_some_cb(&dummy_callback); /* means the same, the ampersand operator is not
needed */
```

不使用固定长度数组的typedef

Do not use a typedef of a fixed-length array

If a fixed-length array is typedefed, it still behaves like an array. This means that when it is passed to a function, it is passed by reference, not by value.

如果一个定长数组是有类型定义的，它的行为仍然像一个数组。这意味着当它被传递给函数时，它是按引用传递的，而不是按值。

A workaround is to enclose it in a struct:

一种解决方法是将其封装在结构体中：

Example (typedefing fixed-length arrays):

```
typedef uint8_t os_macaddr_t[6];                    /* would be passed by
reference */
typedef struct { uint8_t addr[6]; } os_macaddr_t;   /* will be passed by value */
```

避免不必要的typedefs

Avoid unnecessary typedefs

Use of typedefs is encouraged for:

以下情况建议使用typedefs：

- Structures that are internal to implementation (meaning that the members should not be accessed by the caller). 实现内部的结构(意味着调用者不能访问其成员)。
- Complex structures that include unions. 包含联合体的复杂结构。

Use of typedefs is discouraged for:

不建议使用"typedef"关键字的情况包括：

- Structures with members that the caller is expected to inspect or modify. 包含供调用者进行检查或修改的成员的结构。
- Structures that are used only within a module and are not exposed in the interface. 仅在模块内部使用、且未在接口中公开的结构。 枚举类型的别名通常不被提倡使用。对"枚举"关键字的语法着色是一种提示，表明该类型并非通过别名定义的。

Typedefs of enums are generally discouraged. Syntax coloring of the "enum" keyword is a helpful hint that a typedefed type lacks.

枚举类型的别名通常不被提倡使用。对"枚举"关键字的语法着色是一种提示，表明该类型并非通过别名定义的。

Example ("enum" vs. typedefed "enum"):

```
int some_function(enum some_action action);

typedef enum some_action some_action_e;
int some_function(some_action_e action);
```

Use built-in fixed-width integer types (e.g. "uint32_t" or "int64_t") where needed.

在需要的地方使用内置的固定宽度整数类型(例如"uint32_t"或"int64_t")。

Use plain integer types ("int" or "long", as appropriate) where fixed width is not needed.

在不需要固定宽度的情况下，应使用普通的整数类型("int"或"long"，视情况而定)。

要谨慎选择整数类型，以避免类型转换

Choose integer types carefully to avoid the need for typecasts

Typecasting hinders type checking. As much as possible, typecasting should be limited to cases where it is really unavoidable.

类型转换会妨碍类型检查。在尽可能的情况下，类型转换应仅限于确实无法避免的那些情况。

谨慎使用全局变量和静态变量

Use global and static variables carefully

Depending on the architecture of a project, global and static variables may be either perfectly acceptable, or can be a nuisance (causing hard-to-find bugs).

根据项目的架构，全局变量和静态变量可能是完全可以接受的，也可能是一种麻烦(导致难以发现的错误)。

If unsure whether, where, and how global or static variables can be used in a particular module or area, consult a senior engineer.

如果不确定在特定模块或区域中，全局变量或静态变量是否可以使用，以及如何使用，应咨询资深工程师。

声明变量立即显式初始化

Declare and explicitly initialize variables immediately

Explicit initialization is a cornerstone of defensive programming in C; it eliminates crashes, logic errors, and security flaws caused by uninitialized variables:

显式初始化是 C 语言中防御式编程的核心原则之一，能根本性避免因未初始化变量导致的崩溃、逻辑错误和安全漏洞：

- Prevents undefined behavior—local variables without an initializer have indeterminate values (C17 §6.7.9/10). 避免未定义行为, 未初始化的局部变量值是未定义的（C17 §6.7.9/10）
- Removes the risk of irreproducible bugs; an uninitialized pointer, for example, may reference an arbitrary address. 依赖未初始化变量的默认值会导致难以复现的 bug，比如指针未初始化，可能指向任意地址
- Signals design intent and improves readability by making a variable's initial state explicit. 显式初始化直接表明变量的设计意图，增加可读性

Variables should be declared together at the top of the function for better readability.

变量声明应当在函数顶部集中声明, 便于阅读。

好的与不好的做法

Good and Bad Practices

使用"goto”语句来简化错误处理、资源清理以及重试操作

Use "goto" statements to simplify error handling, cleanup of resources, and retries

Goto statements have a bad reputation, but are a very efficient tool for certain purposes. Proper use of goto statements makes the code cleaner, easier to read, and less error-prone.

"goto”语句曾饱受诟病，但在某些情况下却是非常高效的工具。正确使用"goto”语句能使代码更简洁、更易于阅读且更不易出错。

When a function has to free a resource (e.g. allocated memory, or a handle), it is best to write this code once, at the end of the function, and jump to it from wherever it is intended to return from the function.

当一个函数需要释放某种资源(例如已分配的内存或一个句柄)时，最好将这部分代码一次性编写在函数的末尾，并从该代码所在的位置跳转回函数的入口处。

An additional variable might need to be introduced for the return value (typically named "retval"), but such a variable is often useful even if the code uses just control statements and no "goto's".

可能还需要引入一个额外的变量来存储返回值(通常将其命名为"retval”)，但即便代码中仅使用控制语句而没有"goto”语句，这样的变量也往往很有用。

Several labels may be used if different behaviors are required (e.g. just clean-up, vs. log an error and then clean-up).

如果需要不同的处理方式(例如，仅进行清理操作，还是先记录错误信息然后再进行清理操作)，则可以使用多个标签。

An added benefit of such code is that there is a single exit point from the function, which can be useful for debugging, or if additional logging is needed.

这种代码的一个额外优点是，函数只有一个出口点，这在调试时非常有用，或者在需要进行额外日志记录时也同样适用。

Example (cleanup using "goto's"):

```
int function(...)
{
    resource_t resource = NULL;
    int retval = -1;

    ...
    if (condition) goto cleanup;
    ...
    if (condition) goto error;
    ...

    retval = 0;  /* SUCCESS */
cleanup:
    if (resource != NULL) free_resource(resource);
    return retval;
error:
    LOG(...);
    retval = -2;
    goto cleanup;
}
```

Another case where "goto" statements are useful is handling of retries. In this case the "goto" statement jumps 'backward', but this is again nothing to be afraid of.

"goto"语句在处理重试情况时也非常有用。在这种情况下，"goto"语句会"向后跳转"，但这同样无需担心。

In simple cases retries can be achieved with a "while" loop (using "break" when retries are no longer necessary).

在简单的情况下，可以通过使用"while"循环来实现重试(在不再需要重试时使用"break"语句)。

But in some cases the "break" statement cannot be used as it is nested within another loop and would require a less-feared-of 'forward goto' to achieve a 'multi-level break', or an additional variable (e.g. "bool keep_retrying" or something similar), which is an even worse solution.

但在某些情况下，"break"语句无法直接使用，因为它嵌套在另一个循环中，需要使用一种相对不那么令人畏惧的"向前跳转"来实现"多级跳出"，或者需要添加一个额外的变量(例如"bool keep_retrying"或类似的变量)，而这种解决方案则更加糟糕。

In such cases it is better to use a 'backward goto' statement instead of the "while" loop, which also gets rid of a level of indentation.

在这种情况下，最好使用"后退跳转"语句来代替"while"循环，这样还能省去一层缩进。

在以"break"、"goto"或"return"结束的代码块之后，不要使用"else"语句

Do not use an "else" after a block that ends with a "break", "goto" or "return"

If the last statement of a controlled block is an explicit or implicit unconditional jump (like a "break", a "goto" or a "return"), then an "else" statement is redundant.

如果一个受控块中的最后一条语句是显式或隐式的无条件跳转(例如"break"、"goto"或"return"语句)，那么"else"语句就是多余的。

Example (redundant "else"):

```
if (error_condition)
{
    ...
    goto error;
}
else                                    /* BAD CODE */
{
    // no error
    ...
}
```

By removing the redundant "else", the code is easier to follow and does not stray to the right due to unnecessary indentation.

通过删除多余的"else"，代码更容易理解，并且不会因为不必要的缩进而偏离右侧。

Example (redundant "else" removed):

```
if (error_condition)
{
    ...
    goto error;
}

                                        /* Correct */
// no error
...
```

进行失败情况的测试，而非成功情况的测试，以避免不必要的重复层级设置

Test for failure rather than success to avoid unnecessary nesting

If the last statement of an "else" block is an unconditional jump (either a "return" or a "goto"), then the code can be simplified, and an extra indent level can be avoided.

如果"else"块中的最后一行语句是无条件跳转(即"return"语句或"goto"语句)，那么代码就可以简化，并且可以省去额外的缩进层级。

Typically this happens if error checking tests for success and handles failures in an "else" block.

通常情况下，这种情况会出现在这样的场景中：错误检查会针对成功情况进行测试，并在"else"代码块中处理失败情况。

Example (code that tests for success is hard to read):

```
do_the_first_thing(...);
if (success)                            /* BAD CODE */
{
    do_the_second_thing(...);
    if (success)                        /* BAD CODE */
    {
        do_the_third_thing(...);
        ...
    }
    else                                /* BAD CODE */
    {
        goto error;
    }
}
else                                    /* BAD CODE */
{
    goto error;
}
```

It is more efficient to test for failure, handle the failure, and bail out. The normal flow (without failures) then continues at the same indent level and is easier to follow.

进行故障检测、处理故障并退出程序这种方式更为高效。在没有故障的情况下，正常的流程会以相同的缩进级别继续进行，并且更易于理解。

Also, any code that handles or logs failures remains close to where the failure occurred (and was tested for), again making the code easier to read.

此外，任何处理或记录故障的代码都应紧邻故障发生的位置(并且经过了相应的测试)，这样能进一步简化代码的可读性。

Example (code that tests for failure is simpler and more straightforward):

```
do_the_first_thing(...);
if (!success) goto error;               /* Correct */

do_the_second_thing(...);
if (!success) goto error;               /* Correct */

do_the_third_thing(...);
...
```

在条件语句中切勿依赖隐式转换为"bool"类型

Do not rely on implicit conversion to "bool" in conditions

Although a logical expression evaluates to 'true' for a non-zero integer, or a pointer with a value other than NULL, it is better and safer to explicitly write the intended comparison.

尽管对于非零整数或值非 NULL 的指针，逻辑表达式的结果会为"真"，但最好还是明确地写出预期的比较语句，这样既更清晰也更安全。

Example (avoid implicit conversion to bool):

```
if (result) ...                 /* WRONG */

if (result != 0) ...            /* Right */

if (ptr) ...                    /* WRONG */

if (ptr != NULL) ...            /* Right */
```

不要在应填写"NULL"(空值)的地方使用"0"(零)

Do not use "0" (zero) where "NULL" is intended

Always use "NULL" when assigning to or comparing with a pointer.

在给指针赋值或进行指针比较时，务必使用"NULL"值。

The main reason is readability.

主要原因在于可读性。

不要将布尔值与"true"或"false"进行比较

Do not test bool values against "true" or "false"

Before the C99 standard, comparing an 'assumed boolean' value (which was actually an "int") to "TRUE" could result in obscure and unpredictable bugs.

在 C99 标准之前，将"假定布尔值"(实际上是"int")与"TRUE"进行比较可能会导致难以捉摸且难以预测的错误。

Although this danger is no longer present with the C99 "bool" type, comparing bool values against "true" or "false" is too verbose, especially when variables or functions have intuitive names.

虽然 C99 的"bool"类型不再存在这种风险，但将布尔值与"true"或"false"进行比较过于繁琐，尤其是在变量或函数具有直观名称的情况下。

Example (use bool values directly):

```
if (state->is_clean == true) ...        /* WRONG */
if (state->is_clean) ...                /* Right */
```

不要使用"Yoda 条件”，例如"if (NULL == result)”

Do not use 'Yoda conditions', for example "if (NULL == result)"

Yoda conditions is a programming style which reverts the variable and the constant in the logical expression of a conditional statement.

Yoda 条件是一种编程风格，它会将条件语句中的逻辑表达式中的变量和常量进行互换。

This style reduces the risk for bugs caused by mistakenly using a single "=" (assignment) instead of the intended "==" (comparison).

这种写法降低了因错误地使用单一的"=”(赋值)符号而引发错误的可能性，避免了使用不当导致的"==”(比较)符号的使用情况。

However such bugs are relatively rare, especially since modern compilers and static analyzers produce warnings for such code.

然而，这类错误其实相对少见，尤其是因为现代编译器和静态分析工具会对这类代码发出警告。

It is believed that the consequences of poor readability of such conditions outweigh the benefits, therefore this style is deprecated.

人们认为，此类内容阅读难度大所带来的负面影响超过了其带来的好处，因此这种写作风格已被摒弃。

Example (conventional vs. Yoda conditions):

```
if (result == NULL) ...         /* Right - conventional style, intuitive */

if (NULL == result) ...         /* WRONG - 'Yoda style' (reverse grammar) */
```

Some argue that with the constant at the beginning of the expression it is clearer what is tested for, especially if the other operand is a long function call. But note that in this case there is no danger of having an accidental assignment, and also that while the constant now stands out, the action performed by the function call does not. For testing outcomes of function calls it is much better to use a temporary variable, as described next.

有人认为，如果表达式开头带有常量，那么就能更清晰地明确所要测试的内容，尤其是当另一个操作数是一个较长的函数调用时更是如此。但请注意，在这种情况下，不存在意外赋值的风险，而且尽管这个常量现在更加显眼了，但函数调用所执行的操作却并未受到影响。

使用临时变量来测试函数调用的返回值

Consider using a temporary variable for testing the return value of a function call

Putting a large function call (a long function name, many parameters) into a conditional statement is not very readable, regardless of the situation.

将一个大型函数调用(一个很长的函数名以及众多参数)放入一个条件语句中，无论在何种情况下都很难读取清楚。

For testing outcomes of function calls, it is much better to use a temporary variable (which can also be used to log the value, if needed).

对于函数调用的测试结果，使用一个临时变量会更好(如果需要的话，这个临时变量还可以用于记录值)。

Additionally, if the function call is within the conditional expression, the line becomes so long that even a simple action must be put into a block.

此外，如果函数调用位于条件表达式内部，那么这一行代码就会变得非常冗长，以至于即使是简单的操作也必须放入一个代码块中。

By separating the function call from the conditional statement one also has the option to write the parameters each in its own line, which would become quite unreadable if used within the conditional statement.

将函数调用与条件语句分开后，还可以选择将参数分别写在各自的行上。但如果将这些参数放在条件语句中，那么其可读性将会大打折扣。

Example (testing the return value of a function call):

```
                                                        /* BAD CODE */
if (function_call(parameter_1, parameter_2, parameter_3, ...) != SUCCESS_VALUE) {
    goto error;
}

                                                        /* Correct */
result = function_call(parameter_1, parameter_2, parameter_3, ...);
if (result != SUCCESS_VALUE) goto error;
```

如何避免在条件表达式中赋值操作时出现警告信息

How to avoid warnings for assignments in conditional expressions

If putting an assignment into the conditional expression is actually intended, the gcc warning can be avoided by enclosing the assignment in additional parentheses.

如果确实是要将赋值语句放入条件表达式中，那么可以通过在赋值语句外再添加一对括号来避免 gcc 提出的警告。

However, since relying on implicit conversion to bool is not allowed, the value should be tested explicitly anyway, which also avoids the warning.

然而，由于不允许使用隐式转换来转换为布尔类型，所以无论如何都应该对这个值进行显式测试，这样也能避免出现警告。

Example (suppressing warnings for assignments in conditional expressions):

```
...
} while (element = element->next);                  /* warning */

...
} while ((element = element->next));                /* no warning, but still
relying
                                                       on implicit conversion */

...
} while ( (element = element->next) != NULL );      /* Correct (and no warning) */
```

Note that the above example would have to separately test for an empty list. The loop can be written differently, and then the assignment in the conditional statement is completely avoided.

请注意，上述示例必须单独对空列表进行测试。该循环可以以不同的方式编写，这样就可以完全避免在条件语句中进行赋值操作。

Example (avoiding assignments in conditional expressions):

```
while (element != NULL)
{
    ...
    element = element->next;
}
```

合理使用宏程序

Use macros sensibly

There are many uses for macros, but one should be aware of the caveats (type safety issues, multiple evaluation, and more – see Macro Pitfalls).

宏有很多用途，但使用者应当了解其中的注意事项(类型安全问题、多重评估等等——详情请参阅《宏的陷阱》)。

In some cases macros are the only solution and actually the right tool for the job. But they should not be considered to be some sort of light-weight functions. If the macro looks like a helper function, then it should probably be replaced by an inline function.

在某些情况下，宏是唯一的解决方案，而且实际上也是完成这项任务的理想工具。但不应将其视为某种轻量级的函数。如果宏看起来像一个辅助函数，那么或许就应该将其替换为内联函数。

在代码中切勿使用"魔法数字"

Never use 'magic numbers' in the code

There should be no so-called 'magic numbers' in the code.

代码中不应存在所谓的"神奇数字"。

Some literal values are acceptable, like 0 and 1, sometimes 2 or 8 (but consider using sizeof() where the size of an integer is needed), 1024 (if used as a multiplier, no need to define it as KILOBYTE or something like that), or small values used to shift bits.

有些具体的数值是可以接受的，比如 0 和 1，有时也可以是 2 或 8(但在需要确定整数大小时，建议使用 sizeof() 函数)，1024(如果用作乘数，无需将其定义为 KILOBYTE 或类似名称)，或者用于位移操作的小数值。

Any number that may potentially be changed in future should be defined in a single place.

任何可能在未来发生变化的数字都应在一个单一的地点进行定义。

Function return values should also avoid magic numbers. Even if the only possible returns are -1 and 0, give them named constants via macros or an enum so the code is self-explaining.

函数返回值也应该避免使用魔法数字，即使返回值只有 -1 和 0，也要用宏/枚举起名让代码便于阅读。

使用 strscpy() 或 STRSCPY() 函数可防止缓冲区溢出

Use strscpy() or STRSCPY() to prevent buffer overruns

Never use strncpy() to prevent buffer overruns because it does not nul-terminate the result if truncation occurs!

切勿使用 strncpy() 函数来防止缓冲区溢出，因为如果出现截断情况，该函数不会在结果末尾添加空字符！

Using snprintf() is better, and is the right choice if formatting (or even just simple concatenation) is needed.

使用 snprintf() 函数会更好，而且如果需要进行格式化处理(甚至只是简单的字符串拼接)的话，这也是正确的选择。

If a string needs to be merely copied, use the dedicated strscpy() or STRSCPY() function/macro.

如果只需要对字符串进行复制操作，那么请使用专门的 strscpy() 函数或 STRSCPY() 宏。

Example (preventing buffer overruns):

```
strscpy(buf, name, len);                                /* Correct (e.g. when len
is
                                                           supplied as a
parameter) */

strscpy(cfg->name, name, sizeof(cfg->name));            /* STRSCPY() does that
                                                           and is easier to read
*/

STRSCPY(cfg->name, name);                               /* Correct */

snprintf(buf, sizeof(buf), "%s%s", part1, part2);       /* Correct */
```

The above examples all silently truncate the input string. This is acceptable in cases where it is believed that truncation will not occur in normal operation, or that if truncation does occur it only causes an acceptable loss of non-essential functionality.

上述示例均会默默地截断输入字符串。在认为在正常运行中不会出现截断情况，或者即便出现截断也只是会导致可接受的非关键功能损失的情况下，这种做法是可以接受的。

For critical operations the code should check the return value, and if truncation occurred, the operation should be aborted.

对于关键操作，代码应当检查返回值，如果出现截断情况，则应终止该操作。

Example (checking for truncation with strscpy or STRSCPY):

```
if (strscpy(buf, name, len) < 0)            /* strscpy (or STRSCPY) returns -E2BIG
*/
{
    LOGE(...);
    goto error;
}
```

When snprintf is used, make use of its return value (even when the output is truncated, it returns the number of characters, excluding the terminating nul-byte, that would have been written).

在使用 snprintf 函数时，请利用其返回值(即便输出被截断，它也会返回实际写入的字符数，不包括终止的空字符)。

Example (checking for truncation with snprintf):

```
rv = snprintf(buf, sizeof(buf), "%s%s", part1, part2);
if (rv >= (int)sizeof(buf))
{
    LOGE(...);
    goto error;
}
```

提炼公共函数以避免跨文件重复实现

Extract Common Functions to Eliminate Cross-File Duplication

Duplicated code bloats binaries, complicates testing, and spreads bugs. Consolidate identical or highly similar logic into a shared library to guarantee uniform behavior (fix once, benefit everywhere) and drastically cut maintenance overhead.

重复代码导致体积膨胀、测试复杂、Bug 蔓延。将相同或高度相似逻辑提取到公共库，可统一行为、一处修复、全局受益，显著降低维护成本。

新增的功能，增加新的功能型宏包裹

Wrap functional code with feature-based macros

Wrap functional code with feature-based macros, never with device-specific macros, to avoid hard-coding model names.

功能性代码的包裹逻辑不要用模型宏，替换为功能型宏，避免硬编码型号。

Example:

```
#ifdef CONFIG_OSP_UNIT_CBE1V1K          /* BAD CODE */
...
do something;
...
#endif

#ifdef CONFIG_MANAGER_LTEM              /* Correct */
...
do something;
...
#endif
```

New feature toggle macros and their configuration options must be introduced through Kconfig and default to disabled, so normal code paths remain unaffected.

新功能的开关宏和配置文件应该通过Kconfig设置，并且默认值设置为disable，避免对正常的功能代码造成影响。

Example:

```
menuconfig MANAGER_LTEM2
    bool "LTE Manager (LTEM)"
    default n                          # disable by default
    help
        Enable LTE Manager (LTEM)

    config TARGET_LTE_NAME
       depends on MANAGER_LTEM2
       string "LTE name"
       default "rmnet_data0"
       help
            LTE name that will be used
```

缓冲区操作接口应该显式传递「地址」与「长度」

Buffer-manipulating interfaces shall explicitly pass both address and length

Any interface that reads from or writes to a buffer shall accept two independent parameters: void *buf and size_t len.

任何对缓冲区进行读写的接口，应该同时接收 void *buf 与 size_t len 两个独立形参。

It is the caller's responsibility to ensure these two values match. The callee must not deduce the buffer size from global variables, the length of a statically allocated array, strlen, or any other implicit means, thereby eliminating the risk of out-of-bounds access.

调用方有责任保证二者匹配，避免在函数内部通过全局变量、静态数组长度、strlen 或其他隐含手段推导缓冲区大小，从根本上杜绝越界访问风险。

```
                                                        /* Bad code */
uint32_t copy_buf(const void *src, char *dst);

                                                        /* Correct */
uint32_t copy_buf(const void *src, size_t src_len,
                  char       *dst, size_t dst_max);
```

函数应该对输入参数进行合法性校验

Functions shall validate all input parameters

Check for invalid inputs—such as NULL pointers or out-of-range values—to prevent subsequent dereferences of NULL pointers, buffer overruns, and other undefined behaviors. This directly reduces the risk of crashes, data corruption, and exploitable security vulnerabilities.

在函数中检查空指针、超范围值等非法输入，避免后续解引用空指针、越界读写等未定义行为，直接降低崩溃、数据破坏、被利用的安全漏洞概率。

Every invalid input shall be explicitly rejected and logged, ensuring predictable function behavior and eliminating elusive bugs such as "occasional crashes" or "random results."

所有非法输入都被明确拒绝并添加log记录，函数行为变得确定，避免"偶发死机""随机结果"这类难以复现的Bug。

Input parameter validation must be grouped at the top of the function.

入参错误检查须集中放在函数的顶部。

不得忽略任何编译器告警

Never ignore any compiler warning

A compiler warning is a latent defect. Maintaining a "zero-warning" build forces the code to meet the highest baseline for syntax, portability, and latent errors.

编译器告警即潜在缺陷，保持"零告警"可迫使代码在语法、可移植性与隐含错误等方面达到最高质量基线。

Do not cast pointers to void * merely to silence compiler warnings; such casts bypass type checking and can hide alignment or size mismatches. Keep the original pointer type and use explicit casts only when absolutely necessary, accompanied by a clarifying comment.

不要为了消除编译器警告就随意将指针类型转换为 void*；这种做法会绕过类型检查，并可能掩盖对齐或大小不匹配的问题。请尽量保持原始指针类型，只有在绝对必要的情况下才进行显式类型转换，并添加相应的注释说明原因。

文件权限须精确授予

File permissions must be granted precisely; over-permission is strictly prohibited.

Configuration files, source code, templates, JSON/YAML, and other pure data files must not have any executable (x) bit set, ensuring they cannot be executed even if invoked by mistake.

配置文件、代码文件、模板、JSON/YAML 等纯数据文件不要设置任何 x（executable）位，确保即使被误调用也无法直接执行。

Note: When modifying a Linux repository over Windows network shares (SMB), Samba's create mask may forcibly set files to 755.

Note: 通过 Windows 网络共享（SMB）修改 Linux 仓库时，Samba 的 create mask 可能将文件强制设为 755。

保持空行与行尾零空格

Keep empty lines and line endings free of trailing whitespace

Spaces at the end of lines or on otherwise-empty lines create "noise diffs" that clutter comparisons.

空行或行尾的空格会造成"无意义差异"，在 diff 时产生大量干扰。

使用字符串时须保证以 '\0' 结尾

When dealing with strings, ensure they are always '\0'-terminated

Passing a string that is not '\0'-terminated to strlen() or similar functions results in undefined behavior, which can produce an incorrect length, out-of-bounds access, or even a segmentation fault.

把不以 '\0' 结尾的字符串传入strlen()等函数将导致未定义行为，可能返回错误长度、越界访问甚至段错误。

使用经边界校验或具备错误反馈的安全函数

Use functions that perform boundary checks or provide explicit error feedback

Avoid functions known to cause buffer overflows or lack error detection (e.g., gets, sprintf, atoi).

不要使用已知易导致缓冲区溢出或缺乏错误检测的函数（如 gets、sprintf、atoi 等）。

Prefer length-limited, verifiable-return variants (e.g., fgets, snprintf, strtol) and always inspect their return values.

使用带长度限制、返回值可校验的函数（如 fgets、snprintf、strtol），并在使用时对返回值进行检查。

The unsafe functions are listed below but not limited to. 以下列出的函数是不安全的，但并非仅限于这些函数。

Function(s) to Avoid	Safer Alternative(s)

Function(s) to Avoid	Safer Alternative(s)
gets()	fgets()
strcpy()	strncpy(), snprintf()
strcat()	strncat()
sprintf()	snprintf()
vsprintf()	vsnprintf()
atoi(), atol(), atof()	strtol(), strtod(), strtoll()
strtok()	strtok_r() (POSIX) or strtok_s() (C11 Annex K)
mktemp()	mkstemp()
stelen()	stenlen()

Embedded Shell Scripts Should be Avoided

应避免使用嵌入式Shell脚本。

Sometimes we can't avoid using Linux commands to get output, but we should still strive to use C code to find the data from the output. We should avoid using "awk", "grep", "sed", "echo", the pipe "|" operator, and "cat" (when possible) in C files (there's probably others). Using Shell within C code uses more resources than just parsing the string within C (it opens up a Shell session). The first "popen()" call causes a new thread to be opened, and every pipe "|" causes another thread to be opened, all using resources.

有时我们可能无法避免使用Linux命令来获取输出结果，但我们仍然应该尽量使用C语言代码来解析这些输出数据。在C代码中，我们应该尽量避免使用“awk”、“grep”、“sed”、“echo”、“|”（管道符号）和“cat”（可能还有其他命令）。在C代码中使用Shell命令比直接在C语言中解析字符串消耗更多资源（因为这会启动一个Shell进程）。第一次调用“popen()”函数会创建一个新的线程，而每次使用管道符号“|”都会创建另一个线程，这些都会占用系统资源。