HOPECHART

嵌入式C编码规范

编号：HQ-C-33-06　　版本：C/0　　发布号：

编制：金丽娇　日期：2019.9.23

审核：沈林强　日期：2019.9.23

批准：徐小清　日期：2019.9.25

保密等级：□绝密 ■机密 □敏感 □公开

受控文件盖章区	作废盖章区

杭州鸿泉物联网技术股份有限公司

基本信息

文档名称	嵌入式 C 编码规范				
文档编号	HQ-C-33-06		当前版本	C/0	页数	共 26 页
起草时间	2015 年 07 月 01 日		定稿时间	2015 年 07 月 22 日	
起草人	姓名	部门	电话	电子邮件	
	刘沾林	研发中心	13989464604	lzl@hopechart.com	

修改记录

序号	修改时间	修改人	主要修改	存档版本
1	2016.08.30	上官丹萍	ISO 9001-2015 转版	B/0
2	2018.01.22	俞月卿	将“杭州鸿泉数字设备有限公司”改为“杭州鸿泉物联网技术股份有限公司”	B/1
3	2018.07.05	沈林强	重新制定 C 代码规范	B/2

4	2019.09.11	沈林强 刘沾林 季华 尤胜坤	新增“类”的定义 新增成员函数的命名描述 新增枚举值的详细说明 新增全局变量的命名说明 新增枚举/结构体重命名的说明 更改公司 Logo 图标 更改页眉和页脚 区分源文件和头文件的注释要求，更改注释规范	B/3
5	2019.09.11	金丽娟	根据质量手册过程识别修改进行梳理修订	C/0
6				

目录

1 目的

为了改善软件的可读性，可以让开发人员尽快而彻底地理解新的代码，确保每个软件开发人员必须一致遵守编码规范，特制定本规范。

2 范围

本规定适用于指导本公司所有产品的 C 编码、评审过程。

3 职责

C 开发工程师：按照本规范执行编码。

评审专家：负责进行代码规范评审。

4 定义

无

5 流程图

无

6 内容

6.1 头文件

6.1.1 头文件名的定义

1. 头文件以.h 结尾。

2. 对于插入的文本，以.inc 结尾，不能以-inc.h 结尾。

6.1.2 防止多次编译的保护

使用条件宏指令保护。

格式：_<PREFIX>_<FILE>_H_

PREFIX：定义前缀，一般用工程名或者库名；全部字母大写

FILE：文件名；全部字母大写

例:

```
#ifndef _FOO_BAZ_H_
#define _FOO_BAZ_H_
...
#endif // _FOO_BAZ_H_
```

6.1.3 内联函数

只有当函数只有 10 行甚至更少时才将其定义为内联函数。

6.1.4 #include 的路径及顺序

使用标准的头文件包含顺序可增强可读性，避免隐藏依赖，顺序如下：

相关头文件；

C 库

其他库的.h

本项目内的.h

说明：

1. 相关头文件的说明：比如有 2 个文件，a.h，a.c，那么 a.c 中第一个引用的头文件应该是 a.h

```
#include "a.h"
```

这个就是相关头文件。

注：

避免使用以"."， ".."开头的相对路径。如：

```
#include "../a.h"
```

6.2 作用域

6.2.1 局部变量

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。

下面的语句

```
int i;
i = f();
```

应该用下面的语句代替：

```
int i = f();
```

6.2.2 静态和全局变量

禁止定义静态储存周期非 POD(Plain Old Data)变量，禁止使用含有副作用的函数初始化 POD 全局变量，因为多编译单元中的静态变量执行时的构造和析构顺序是未明确的，这将导致代码的不可移植。

POD: 原生数据类型，如 **int**, **char**, **float** 等，已经 POD 类型的指针、数组、结构体等。

6.3 函数

6.3.1 参数顺序

函数的参数顺序为：输入参数在前，后跟输出参数。

输入参数通常是值参或者 const 引用。新加的值参也应该加入到输出参数之前，不应该放到输出参数之后。

对于即输入又输出的参数，放输入参数之后,输出参数之前。

6.3.2 编写简短函数名

我们倾向于编写简短、凝练的函数名。

6.4 其他 C 特性

6.4.1 64 位下的可移植性

代码应该对 64 位和 32 位系统友好。处理打印、比较、结构体对齐时应切记。

6.4.2 预处理宏

使用宏时要非常谨慎，尽量以内联函数、枚举和常量代替之。

使用宏应尽可能遵守：

- 尽量不要在 .h 文件中定义宏；
- 在马上要使用时才进行 #define，使用后要立即 #undef；
- 不要只是对已经存在的宏使用#undef，选择一个不会冲突的名称；
- 不建议用 ##处理函数和变量的名字。

6.4.3 0,nullptr 和 NULL

整数用 0，实数用 0.0，指针用 nullptr 或 NULL，字符（串）用'\0'。

6.4.4 sizeof

尽可能用 **sizeof**(varname)代替 **sizeof**(type)。

TStruct data, *point_data;

用下面的方法：

```
memset(&data, 0, sizeof(data));

memset(point_data, 0, sizeof(*point_data));
```

不要用下面的方法：

```
memset(&data, 0, sizeof(TStruct));

memset(point_data, 0, sizeof(TStruct));
```

6.5 命名约定

6.5.1 通用命名规则

- 函数命名，变量命名，文件命名要有描述性；少用缩写。

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

上面的比较好理解，下面的就不好理解了。

```
int n;                      // 毫无意义.
int nerr;                   // 含糊不清的缩写.
int n_comp_conns;           // 含糊不清的缩写.
int wgc_connections;        // 只有贵团队知道是什么意思.
int pc_reader;              // "pc"有太多可能的解释了.
int cstmr_id;               // 删减了若干字母.
```

- 名称避免过于相似，不要出现仅靠大小写区分的相似的标识符。

例如:

"i"与"I";

"function"与"Function"。

- 名称避免重名，程序中不要出现名字完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但容易使人误解。

6.5.2 文件命名

文件名要全部小写，可以包含下划线（_），依照项目的约定。

文件命名示例：

my_useful_class.c

useful.c

C 文件要以.c 结尾，头文件以.h 结尾。专门插入文本的文件则以.inc 结尾。

6.5.3 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，但是回调函数变量用大小写方式，中间不带下划线。

普通变量命名：

```
string table_name;  // 好 - 用下划线.

string tablename;   // 不要用

string tableName;   // 不要用
```

结构体变量：

不管是静态的还是非静态的，结构体数据成员都可以和普通变量一样：

```
typedef struct TUrlTablePropertiesTag TUrlTableProperties;

struct TUrlTablePropertiesTag

{
```

```
    char *name;
    int num_entries;
};
```

回调函数变量：

```
typedef struct TDrvCanTag TDrvCan;
typedef void (*TDrvCanOnSent)(TDrvCan *sender);
struct TDrvCanTag
{
PUBLIC
    TDrvCanOnSent OnSent;   // 发送完成事件
    ...
};
```

6.5.4 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，命名时以 “k” 开头，大小写混合。例如:

```
const int kDaysInAWeek = 7;
```

6.5.5 函数命名

常规函数使用大小写混合，如：

MyExcitingFunction(),

MyExcitingMethod(),

对于缩写的字母也是如此，比如：

StartRpc();

而不用

StartRPC();

成员函数名称统一以“T”开头，并进行大小写混合，如:

void TDrvCanSendData(TDrvCan *self, TCanData *can_data);

6.5.6 枚举命名

枚举的类型名称统一以“T”开头，以“Tag”结尾。

枚举类型的值在 c 标准中规定的以“k”开始的基础上，规定后续跟枚举类型的重定义名称（除最前面的“T”外），后续跟值的名称。比如：kDrvUartParityNone。

```
typedef enum TDrvUartParityTag TDrvUartParity;
enum TDrvUartParityTag
{
    kDrvUartParityNone,
    kDrvUartParityEven,
    kDrvUartParityOdd,
```

```
};
```

整个值的名称控制在 24 个字符以内，如果整个值的名称过长，可以缩短枚举类型名称，但前提是尽可能不跟其它枚举类型冲突。如下所示，原 kDrvCanBaudRate250k 缩短为 kCanBaudRate250k：

```
typedef enum TDrvCanBaudRateTag TDrvCanBaudRate;
enum TDrvCanBaudRateTag
{
    kCanBaudRate250k,
    kCanBaudRate500k,
};
```

6.5.7 结构体和枚举名称重定义

为了使用方便，可以对结构或者枚举进行重定义，重定义格式：

```
typedef struct TxxxTag Txxx;
typedef enum TxxxTag Txxx;
```

重定义语句必须在实体定义之前，如果没有必要，两者中间不要插入其它内容，如下所示：

```
typedef enum TDrvCanBaudRateTag TDrvCanBaudRate;
enum TDrvCanBaudRateTag
```

```
{
    ...
};
```

两者之间必须插入内容的情况，如下所示：

```
typedef struct TDrvCanTag TDrvCan;
typedef void (*TDrvCanOnSent)(TDrvCan *sender);
struct TDrvCanTag
{
PROTECTED
    TDrvCanOnSent OnSent_;
    ...
};
```

6.5.8 宏命名

宏命名全部大写，使用下划线：MY_MACRO_THAT_SCARES_SMALL_CHILDRE。

```
#define FEATURE_CAN_MAX_MB_NUM (32)
#define MIN(x, y) (x > y ? y : x)
```

6.6 源文件注释

6.6.1 注释风格

单行使用　　// 单行注释

多行

/**

多行注释

*/

源文件中的注释也可参考头文件中的注释。

6.6.2　函数注释

静态函数需要注释，已经在头文件中公开的函数，源文件中函数本身不用注释，但函数实现部分加入必要的注释，静态函数的注释参考头文件的函数注释要求。

6.6.3　变量注释

通常变量名本身足以很好说明变量用途。某些情况下，也需要额外的注释说明。

全局变量：

全局变量定义分 2 种，一种全局变量仅在.c 文件中可以见，另外一种为整个工程中可见。

仅在当前.c 文件中的全局变量以“g_”开头，以“_”结尾，变量类型必须为“static”，如下所示：

```
static int g_count_;
```

整个工程中可见的全局变量以“g_”开头，如：

```
int g_count;
```

需要在头文件中导出 extern int g_count;

和数据成员一样，所有全局变量也要注释说明含义及用途，以及作为全局变量的原因。比如：

```
// The total number of tests cases that we run through in this regression test.

const int kNumTestCases = 6;
```

6.6.4 实现注释

对于代码中巧妙的，晦涩的，有趣的，重要的地方加以注释。

代码前注释：

巧妙或复杂的代码段前要加注释。比如：

```
// Divide result by two, taking into account that x

// contains the carry from the add.

for (int i = 0; i < result->size(); i++) {

    x = (x << 8) + (*result)[i];

    (*result)[i] = x >> 1;

    x &= 1;

}
```

行注释：

比较隐晦的地方要在行尾加入注释。在行尾空两格进行注释。比如：

```
// If we have enough memory, mmap the data portion too.

if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))

    return;  // Error already logged.
```

6.6.5 标点，拼写和语法

注意标点，拼写和语法；写的好的注释比差的要易读的多。

注释的通常写法是包含正确大小写和结尾句号的完整叙述性语句。大多数情况下，完整的句子比句子片段可读性更高。短一点的注释，比如代码行尾注释，可以随意点，但依然要注意风格的一致性。

虽然被别人指出该用分号时却用了逗号多少有些尴尬，但清晰易读的代码还是很重要的。正确的标点，拼写和语法对此会有很大帮助。

6.6.6 TODO 注释

对那些临时的，短期的解决方案，或已经够好但仍不完美的代码使用 TODO 注释。

```
// TODO(kl@gmail.com) : Use a "*" here for concatenation operator.

// TODO(Zeke) : change this to use relations.

// TODO(bug 12345) : remove the "Last visitors" feature
```

如果需要在代码中的某个位置设置标签,可以用 DONE

```
// DONE(fox) : xxx
```

6.7 头文件注释

6.7.1 注释风格

单行注释：

/// 前置注释

///< 后置注释

说明：这里的’/’和’<’后面需要空一格空格，前置注释为注释后置项，后置注释为注释前置项，具体可以参考枚举和结构体的注释例子。

多行注释：

```
    /**
     多行注释
     */
```

6.7.2 “类”注释

```
/**
 * 这里说明的类的功能和用途以及类使用过程中的注意事项
 * @par 示例
 * @code
   TStringTest dox_test;

   TStringTestCreate(&dox_test, "Alise", strlen("Alise"));

   if (TStringTestNameLen(&dox_test) > 0)
   {
        ...
   }

   DestroyObject((TObject *)&dox_test);
 * @endcode
*/
typedef struct TStringTestTag TStringTest;
struct TStringTestTag
{
```

```
    TObject parent_;    ///< 基类


PRIVATE
    char *name_;        ///< 名字
    int name_len;        ///< 名字长度
    int seq_id_;         ///< 流水号
    char flag_;          ///< 标志

    /// 测试值
    char *test_value_;
};
```

说明：类中的注释中需要说明类的功能，示例代码等。

6.7.3 结构体注释

```
/**
 * XXXXXX...(结构体名注释)
 */
typedef struct TStringStructTag TStringStruct;
struct TStringStructTag
{
    char *value1;       ///< (后置注释)XXXXXX...
    int value2;         ///< (后置注释)XXXXXX...
    int value3;         ///< (后置注释)XXXXXX...
    char value4;        ///< (后置注释)XXXXXX...
```

```
    /// (前置备注结构体成员变量,)XXXXXX...
    char *test_value;
};
```

说明：

结构体名的注释采用多行注释,/**开头 */结尾，成员变量的注释用前置注释或后置注释的方式，“()”中的内容只是在文档中说明意义，如果在变量的末尾加注释统称后置注释，在变量的前面加注释统称前置注释。

6.7.4 枚举注释

```
/**
 * XXXXXX...(枚举名注释)
 */
typedef enum TStringEnumTag TStringEnum;
enum TStringEnumTag
{
    kStringEnumTest0    = 0,     ///< (后置备注)XXXXXX...
    kStringEnumTest1    = 1,     ///< (后置备注)XXXXXX...

    /// (前置备注)XXXXXX...
    kStringEnumTest2    = 2,
};
```

说明：

枚举名的注释采用多行注释,/**开头 */结尾，枚举项的注释用前置注释或后置注释的方式，“()”中的内容只是在文档中说明意义，如果在枚举项的末尾加注释统称后置注释，

在枚举项的前面加注释统称前置注释。

6.7.5 变量注释

```
/// (前置注释)XXXXX...
extern unsigned int g_string_test_value1;
extern unsigned int g_string_test_value2; ///< (后置注释)XXXXXX...
```

6.7.6 函数注释

函数声明处的注释描述函数功能；定义处的注释描述函数实现，需要注明多线程是否安全，是否可重入。

函数声明：

函数声明处注释的内容：

函数的输入输出。

函数是否分配了必须由调用者释放的空间。

参数是否可以为空指针。

是否存在函数使用上的性能隐患。

如果函数是可重入的，其同步前提是什么？

是否支持多线程等

示例：

```
/**
 * 简要说明
 *
 * 详细说明（非必选项，上面的空行只在有需要详细说明的情况下才需要）
```

```
 * @param param1: 参数 1
 * @param param2: 参数 2
 * @todo XXXXXX...(非必选项，待办事项)
 * @note XXXXXX...(非必选项，注意事项)
 * @bug XXXXXX...(非必选项，Bug 事项)
 * @since XXXXXX...(非必选项，自从)
 * @pre XXXXXX...(非必选项，前置条件)
 * @post XXXXXX...(非必选项，后置条件)
 * @deprecated XXXXXX...(非必选项，弃用)
 * @par 代码示例 (非必选项,结合 code 与 endcode)
 * @code
     XXXXXX...
 * @endcode
 * @return
 * @see (非必选项，参见)
 */
int Function2(int param1, int param2);
```

说明：以上的文件注释过程中“()”中的是文档中的说明参数的含义。

6.7.7 回调函数注释

```
/**
 * 数据收发完成事件的函数类型
 * @param sender: 事件的产生者
 * @param server_req: 服务器是否有数据请求发送
 * @return
 */
typedef void (*TDrvXcuSpiDoComplete)(void *sender, Bool server_req);
```

6.7.8 文件注释

```
/**
 * @file XXXXXX.h
 * @brief XXXXXX...
 * @note XXXXXX...
 * @author XXXXXX...
 * @date YYYY-MM-DD
 * @version X.X
 *
 * @par 修改日志
 * <table>
 * <tr><th>Date        <th>Version          <th>Author          <th>Description
 * <tr><td>YYYY-MM-DD  <td>X.X              <td>XXXXXX...       <td>XXXXXX...
 * <tr><td>YYYY-MM-DD  <td>X.X              <td>XXXXXX...
   <td> 1.XXXXXX...  \n
        2.XXXXXX...  \n
        3.XXXXXX...
 * @copyright 杭州鸿泉物联网技术股份有限公司
 */
```

说明：

1.修改日志可分单行和多行编写，若是“类”，则需要写明注意事项，例如多线程是否安全，是否可重入。

2.file 字段必须与文件同名(不含路径，必需项,例如 rfc_string.h)

3.brief 为文件说明

4.note 为注意事项

5. author 为作者

6.8 格式

6.8.1 行长度

每一行代码字符数不超过 120 列。

包含长路径的**#include** 语句可以超出 120 列。

函数尽量控制在 60 行以内。

头文件保护可以无视该原则。

6.8.2 非 ASCII 字符

尽量不使用非 ASCII 字符，使用时尽量使用 UTF-8 编码。

6.8.3 空格还是制表位

只使用空格，每次缩进 4 个空格。

6.8.4 常规符合左右是否有空格的约定

除特殊说明外，一般的约定如下：

1. {: 前面留一个空格，后面不需要。
2. (,), [,], }: 左右不留空格。
3. =, ==, +, -, *, /, %, &, |, ^, &&, ||, >, <, >=, <=, !=: 左右各留一个空格。
4. 运算符和=连用是，如+=: 左右各留一个空格。
5. !, ~: 后面不跟空格。

6.8.5 作为块分隔符的{}放置位置

建议尽量用方案二

- 方案一

{放置在结构、枚举、函数、语句等名称所在行的后面，如下所示：

```
typedef struct TStructTag TStruct;
struct TStructTag {  // {放在这一行
    ...
};                  // }单独一行，位置与前面定义开始的地方对齐。
```

- 方案二

{放置在结构、枚举、函数、语句等名称所在行的下一行，如下所示：

```
typedef struct TStructTag TStruct;
struct TStructTag
{  // {单独一行
    ...
};                  // }单独一行，位置与前面定义开始的地方对齐。

int count(){return count_;}  // 仅允许这样的函数把}放置在后面。
Struct a = {1, 2};           // 或者这样的语句可以把}放置在后面。
```

6.8.6 函数声明与定义

返回类型和函数名在同一行，参数也尽量放在同一行，如果放不下就对形参分行，分行方式与函数调用一致。

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
```

```
    Type par_name2, Type par_name3) {
    DoSomething();
    ...
}
```

或者

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
Type par_name2, Type par_name3)
{   // 单独一行
    DoSomething();
    ...
}
```

6.8.7 函数调用

要么一行写完函数调用，要么在圆括号里对参数分行，要么参数另起一行且缩进四格。如果没有其它顾虑的话，尽可能精简行数，比如把多个参数适当地放在同一行里。

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

6.8.8 列表初始化格式

您平时怎么格式化函数调用，就怎么格式化列表初始化。

```
// 一行列表初始化示范.
return {foo, bar};
```

```
functioncall({foo, bar});

// 当不得不断行时.

SomeType variable{

    "This is too long to fit all in one line"};  // 字符串过长，因此无法放在同一行.

MyType m = {  // 注意了，您可以在 { 前断行.

    superlongvariablename1,

    superlongvariablename2,

    {short, interior, list},

    {interiorwrappinglist,

    interiorwrappinglist2}

};
```

6.8.9 条件语句

倾向于不在圆括号内使用空格。关键字 if 和 else 另起一行，建议尽量用方案二。

方案一：

```
if (condition) {   // if 后面空一格，圆括号里没有空格.

    ...            // 4 空格缩进.

} else if (...) {  // }后空一格，else 与 if 的右括号同一行.

    ...

} else {
```

```
    ...
}
```

或者

方案二：

```
if (condition) // if 后面空一格，圆括号里没有空格.
{
    ...             // 4 空格缩进.
}
else if (...)
{
    ...
}
else
{
    ...
}
```

注意：condition 应使用 bool 型，如果是整型的话可以使用 param != 0 得到 bool 型；指针可以用 p != NULL 得到 bool 型。

6.8.10 循环和开关选择语句

switch 语句可以使用大括号分段， 以表明 cases 之间不是连在一起的。在单语句循环里，括号可用可不用。空循环体应使用{}或 continue。

```
switch (var) {
case 0: {     // 0 空格缩进
    ...       // 4 空格缩进
              // 如果没有变量定义，可以去掉这对{}


    }
    break;
    // 此处空一行
default: {
    // 如果 default 应该永远执行不到，这里为空，并且可以去掉这对{}
    }
    break;
}
```

或者

```
switch (var)
{
case 0:
    {
```

```
        ...         // 4 空格缩进

                    // 如果没有变量定义，可以去掉这对{}


        }

        break;

        // 此处空一行

    default:

        {

        // 如果 default 应该永远执行不到，这里为空，并且可以去掉这对{}

        }

        break;

}
```

6.8.11 指针和取地址表达式

句点或箭头前后不要有空格。指针/地址操作符（*，&）之后不能有空格。

```
x = *p;

p = &x;

x = r.y;

x = r->y;
```

在声明指针变量或参数时，星号与变量名紧挨：

```
char *c;
```

6.8.12 布尔表达式

如果一个布尔表达式超过标准行宽，断行方式要统一一下。

```
if (this_one_thing > this_other_thing
    && a_third_thing == a_fourth_thing
    && yet_another && last_one) {
    ...
}
```

6.8.13 函数返回值

不要在 return 表达式里加上非必须的圆括号。

```
return result;  // 返回值很简单，没有圆括号.
// 可以用圆括号把复杂表达式圈起来，改善可读性.
return (some_long_condition
    && another_condition);
```

6.8.14 变量及数组初始化

用=初始化。

```
int x = 3;
int x(3); 不用这种初始化
int x{3}; 不用这种初始化
```

6.8.15 预处理指令

预处理指令不要缩进，从行首开始。

```
// 指令从行首开始
    if (lopsided_score) {
#if DISASTER_PENDING      // 正确 - 从行首开始
        DropEverything();
# if NOTIFY               // 不要在#后跟空格
        NotifyClient();
# endif
#endif
        BackToNormal();
    }
```

6.8.16 水平留白

水平留白的使用根据在代码中的位置决定。永远不要在行尾添加没意义的留白。

6.8.17 空行

以功能子块为单位，进行空行，不要有超过 2 行的空行。

6.9 “类”定义

类的定义实际上是结构的定义，但作为类，又进行了进一步的限制。

6.9.1 “类”定义的固定结构：

所有类的祖先类必须是 TObject。除 TObject 外，其它类必须具有如下格式：

```
struct TxxxTag
```

```
{
PRIVATE
    Txxxx parent_;
    TVirtualMethod Destroy_;
    ...
PUBLIC
    ...
};
```

6.9.2 “类”的事件函数定义：

事件函数类型名称统一以“T”开头，与函数类型的定义相同，函数的第一个参数必须是 sender，类型为该类的类型，如:

```
void (*TDrvCanOnSent)(TDrvCan *sender, ...);
```

6.9.3 带有事件函数的“类”的定义：

1. 在类的 PUBLIC 中必须加入一个 TObject *类型的 receiver，该参数存放事件函数的对象。

2. 事件函数变量的命名方式同函数名的命名方式，即驼峰式。

如：

```
struct TDrvCanTag
{
    ...
```

```
PUBLIC
    TObject *receiver;
    TDrvCanOnSent OnSent;
    ...
};
```

6.9.4 “类”的函数定义：

类的函数名以类名开头（不含末尾的 Tag），之后为函数名。函数的第一个参数为该类类型的 self。如果类函数仅在.c 文件中使用，函数前面加上“static”。如：

```
void TDrvCanCreate(TDrvCan *self);
static void TDrvCanDonePort(TDrvCan *self)
{
    ...
}
```

7 相关文件

无

8 相关记录

无

9 附录

9.1 头文件编写规范请参考 rfc_string_test.h