

嵌入式 C 语言开发规范 V1.2.0 版本

一、开发环境

主要为 WIN10 及以上版本，所用到的集成开发环境有：

平台	集成开发环境	调试环境
STM32 HAL 库/ 标准库	MDK V5.35 (AC5 / --gnu / C99mode / USE Micro Lib) STM32CubeMX 6.6.1	调试下载 ST-Linkv2/Jlink-v8 程序下载 STM32CubeProgrammer 内存访问 STM32 ST-LINK Utility 固件烧录 STM32TrustedPackageCreator 变量监测 STM32CubeMonitor
Arduino	Arduino2.0.7	

二、参考规范

进行开发时参考 C99 标准，主要用到的 C99 特性包括：变长数组 VLA、restrict 关键字、inline 关键字、内联函数、_Bool 类型、可移植类型 stdint.h 和 inttypes.h。

三、基本排版格式

- 1. 需要以 4 个空格为单位的缩进;
- 2. 不适用 Tab 键进行缩进;
- 3. UTF-8 编码格式;
- 4. 所有注释，用英文编写，并且格式：/*xxx*/不允许使用//
中文注释在不同版本 IDE 当中会出现编码错误，导致出现乱码

四、文件夹结构

对于 STM32 标准库开发和 Arduino 开发，文件夹结构如下：

母文件夹	子文件夹	作用
Doc	无	用来存放程序说明的文件，一般以 Readme.txt 命名 内容包括项目简介、代码层次、文件夹和文件介绍、 开发环境介绍（IDE 名称及版本、编译器名称及版本、硬件平台信息、 内存空间组织：堆栈尺寸及起始地址、固件库版本、调试设置、公共 宏） 使用的引脚及宏定义、使用的外设作用和参数设置 以英文书写
Lib	无	固件库文件
Listing	无	编译器编译时候产生的 C/汇编/链接文件
Output	无	存放编译产生的调试信息、hex 文件、预览信息、封装库灯
Project	无	用来存放工程
User	Common	通用文件：包括数据类型重定义文件、数据类型转换文件
	Hardware	驱动文件：主要是外设操作函数的封装，驱动模块
	Function	功能文件：主要是对驱动文件的封装，使用模块实现某种功能
	Task	应用文件：主要是调用功能文件中功能函数，实现应用
	Config	设置文件：包括 <div><code>stm32f10x_conf.h</code>: 用来配置库的头文件 <code>stm32f10x_it.h</code> <code>stm32f10x_it.c</code>: 中断相关的函数都在这个文件编写，暂时为空 <code>main.c</code>: main 函数文件</div>

对于 STM32 HAL 库开发，自定义文件夹结构如下：

母文件夹	子文件夹	作用
Doc	无	用来存放程序说明的文件，一般以 Readme.txt 命名 内容包括项目简介、代码层次、文件夹和文件介绍、 开发环境介绍（IDE 名称及版本、编译器名称及版本、硬件平台信息、 内存空间组织：堆栈尺寸及起始地址、固件库版本、调试设置、公共 宏） 使用的引脚及宏定义、使用的外设作用和参数设置 以英文书写
User	Common	通用文件：包括数据类型重定义文件、数据类型转换文件
	Hardware	驱动文件：主要是外设操作函数的封装，驱动模块
	Function	功能文件：主要是对驱动文件的封装，使用模块实现某种功能
	Task	应用文件：主要是调用功能文件中功能函数，实现应用

其余文件夹由 STM32CubeMX 自动生成，无需管理

五、文档与注释

1. 对项目工程要有工程说明，工程说明放在 Doc 文件夹下，工程说明包括如下内容：文件夹文件说明、编程环境说明、修改版本记录；
2. 关于注释长度没有具体限制，只要能提供帮助，就尽可能地注释；
3. 注释应该解释代码为什么要这么做，而不是如何去做(代码本身已经表明了如何去做)；
4. 所有代码，空闲时中文注释与英文注释各一份，忙碌时：

用英文编写，并且格式：/*xxx*/不允许使用//

中文注释在不同版本 IDE 当中会出现编码错误，导致出现乱码

六、文件结构

每个 C++/C 程序通常分为两个文件。一个文件用于保存程序的声明（declaration），称为头文件。另一个文件用于保存程序的实现（implementation），称为定义（definition）文件。C++/C 程序的头文件以“.h”为后缀，C 程序的定义文件以“.c”为后缀。

6.1 文件信息声明

文件信息声明位于头文件和定义文件的开头，主要内容有：

- （1）版权信息；
- （2）文件名称，项目代码，摘要，参考文献；
- （3）当前版本号，作者/修改者，完成日期；
- （4）版本历史信息；
- （5）主要函数描述。

文件信息声明模板如下：

```
/**
*****
* File Name      :
* Description    :
* Version        :
* Author         :
*****
*/
```

6.2 头文件

早期的编程语言如 **Basic**、**Fortran** 没有头文件的概念，**C++/C** 语言的初学者虽然会使用头文件，但常常不明其理。这里对头文件的作用略作解释：

(1) 通过头文件来调用库功能。在很多场合，源代码不便（或不准）向用户公布，只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能，而不必关心接口怎么实现的。编译器会从库中提取相应的代码；

(2) 头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规则能大大减轻程序员调试、改错的负担。

头文件由三部分内容组成：

- (1) 头文件开头处的文件信息声明；
- (2) 预处理块；
- (3) 函数和类结构声明等。

假设头文件名称为 `filesystem.h`，头文件的结构参见如下示例

☆	【规则 1.2-1】	为了防止头文件被重复引用，应当用 <code>ifndef/define/endif</code> 结构产生预处理块；“ <code>ifndef</code> ”或者“ <code>define</code> ”后以 TAB 键代替 SPACE 键做空格；如果头文件名称是由多个单词组成，则各单词间以下划线“ <code>_</code> ”连接，例如有头文件名称为“ <code>filesystem.h</code> ”，则定义如下：“ <code>ifndef FILE_SYSTEM_H</code> ”；
☆	【规则 1.2-2】	用 <code>#include <filename.h></code> 格式来引用标准库的头文件(编译器将从标准库目录开始搜索)；
☆	【规则 1.2-3】	用 <code>#include "filename.h"</code> 格式来引用非标准库的头文件(编译器将从用户的工作目录开始搜索)；
☆	【建议 1.2-1】	头文件中只存放“声明”而不存放“定义”；
☆	【建议 1.2-1】	头文件中应包含所有定义文件所定义的函数声明，如果一个头文件对应多个定义文件，则不同定义文件内实现的函数要分开声明，并作注释以解释所声明的函数从属于那一个定义文件；

头文件模板如下：

```
// 文件信息声明

// 文件名字（大写）_H

#ifndef _File_NAME_H
#define _File_NAME_H

//增加对 C++项目引用的支持

//当在一个 C++项目引用这个头文件时，编译器能知道要按照 C 语言对此文件进行编译
#ifdef __cplusplus
Extern "C" {
#endif

// 引用的标准库头文件

/* Includes -----*/

#include <math.h>

...

// 引用的非标准库头文件

#include "myheader.h"

...

// 公共宏定义

/* Common macro definitions-----*/

// 数据结构定义

/* Data structure declaration-----*/

typedef struct Student

{

...

}Stu;
```

```
// 外部变量声明

/* Extern Variable-----*/

//函数声明

/* Function declaration-----*/

void Function1(...);

...

#ifdef __cplusplus
}
#endif
#endif
```

注意，以上文件//后内容不用写

//文件信息声明 按上文的文件信息声明模板编写

6.3 源文件/定义文件

定义文件有三部分内容：

- (1) 定义文件开头处的文件信息声明；
- (2) 对一些头文件的引用；
- (3) 程序的实现体（包括数据和代码）。

假设定义文件的名称为 `filesystem.c`，定义文件的结构参见示例

注意，以下文件//后内容不用写

//文件信息声明 按上文的文件信息声明模板编写

源文件模板如下：

```
// 文件信息声明

// 引用的头文件

/* Includes -----*/
#include "filesystem.h"
...

// 引用的外部函数声明

/* External function declaration-----*/

// 宏定义

/* Private macro definitions-----*/

// 全局变量定义

/* Global variable-----*/

// 内部函数定义

/* Static function definition-----*/

Static void static_Function1(...)
{
...
}

// 函数定义

/* Function definition-----*/

void Function1(...)
{
...
}
```

3.4 目录结构

如果一个软件的头文件数目比较多（如超过十个），通常应将头文件和定义文件分别保存于不同的目录，以便于维护。

例如可将头文件保存于 `include` 目录，将定义文件保存于 `source` 目录（可以是多级目录）。如果某些头文件是私有的，它不会被用户的程序直接引用，则没有必要公开其“声明”。为了加强信息隐藏，这些私有的头文件可以和定义文件存放于同一个目录。

七、文件命名

以“大文件夹_子文件夹_含义名”进行命名：
例如在 `Hardware` 文件夹下的 `ADC` 文件夹中 `ADC` 操作相关的源文件，命名为 `hardware_adc_operate.c`

八、变量命名标准

采用匈牙利命名法，具体规则如下：

【规则 2.2-1】	变量的命名规则要求采用“匈牙利法则”，即开头字母用变量的类型，其余部分用变量的英文意思或其英文意思的缩写，尽量避免采用中文拼音，要求单词的第一个字母大写； 即：变量名=变量类型+变量英文意思(或缩写) 变量类型请参见附表 1—变量类型表；
【规则 2.2-2】	类名和函数名用大写字母开头的单词组合而成；对 <code>struct</code> 、 <code>union</code> 、 <code>class</code> 变量的命名要求定义的类型用大写，结构采用 <code>S</code> 开头，联合体采用 <code>U</code> 开头，类采用 <code>C</code> 开头； 例如： <pre>struct SPoint { int m_nX; int m_nY; }; union URecordLen { BYTE m_byRecordNum; BYTE m_byRecordLen; } class CNode { //类成员变量或成员函数 };</pre>

【规则 2.2-3】	<p>指针变量命名的基本原则为：</p> <p>一重指针变量的基本原则为：</p> <p> 变量名= “p” + 变量类型前缀 + 命名</p> <p>对多重指针变量的基本原则为：</p> <p>二重指针：</p> <p> 变量名= “pp” + 变量类型前缀 + 命名</p> <p>三重指针：</p> <p> 变量名= “ppp” + 变量类型前缀 + 命名</p> <p>.....</p> <p>例如一个 short*型的变量应该表示为 pnStart;</p>
【规则 2.2-4】	全局变量用 g_ 开头；例如一个全局的长型变量定义为 g_lFileNum，
	即：变量名=g_+变量类型+变量的英文意思(或缩写)；
【规则 2.2-5】	静态变量采用 s_ 开头；例如一个静态的指针变量定义为 s_plPrevInst，
	即：变量名=s_+变量类型+变量的英文意思(或缩写)；
【规则 2.2-6】	类成员变量采用 m_ 开头；例如一个长型成员变量定义为 m_lCount，
	即：变量名=m_+变量类型+变量的英文意思(或缩写)；
【规则 2.2-7】	<p>对 const 的变量要求在变量的命名规则前加入 c_ (若作为函数的输入参数，可以不加)，</p> <p>即：变量名=c_+变量命名规则，例如：</p> <p>const char* c_szFileName;</p>
【规则 2.2-8】	<p>对枚举类型(enum)中的变量，要求用枚举变量或其缩写做前缀，且用下划线隔离变量名，所有枚举类型都要用大写，例如：</p> <pre>enum EMDAYS { EMDAYS_MONDAY; EMDAYS_TUESDAY; </pre>
☆ 【规则 2.2-9】	<p>对常量(包括错误的编码)命名，要求常量名用大写，常量名用英文意思表示其意思，用下划线分割单词，例如：</p> <pre>#define CM_7816_OK 0x9000;</pre>
☆ 【规则 2.2-10】	<p>为了防止某一软件库中的一些标识符和其它软件库中的冲突，可以为各种标识符加上能反映软件性质的前缀。例如三维图形标准 OpenGL 的所有库函数均以 gl 开头，所有常量（或宏定义）均以 GL 开头。</p>

九、类型前缀缩写

a 数组 (Array)
b 布尔值 (Boolean)
by 字节 (Byte)
c 有符号字符 (Char)
w 字 (Word)
dw 双字 (Double Word)
h Handle (句柄)
i 整形 (Int)
f 浮点数 (float)
p 指针 (Pointer)
s 字符串 (String)

十、基本数据类型

为保证程序在不同系统间的可移植性，引入 C99 标准库中可移植类型 `stdint.h` 和 `inttypes.h`。同时根据单片机特点（绝大部分单片机字长为 32 位）和实际需要，弃用字长为 64 位的数据类型。同时为了兼容 ST 单片机的旧类型，重新定义数据类型如下，并将重新定义的数据类型重新写入自定义的 `numtype.h` 文件夹：

```
/* ===== Basic data types===== */

/* Signed */
typedef signed char      int8_t;
typedef short int        int16_t;
typedef int              int32_t;
/* Unsigned */
typedef unsigned char     uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int      uint32_t;

/* Float */
typedef float             float32_t;
typedef double            float64_t;
```

```

/* Signed pointer */
typedef signed char*      p_int8_t;
typedef short int*       p_int16_t;
typedef int*             p_int32_t;

/* Unsigned pointer */
typedef unsigned char*    p_uint8_t;
typedef unsigned short int* p_uint16_t;
typedef unsigned int*     p_uint32_t;

/* Void pointer */
typedef void*             p_void;

/* Float pointer */
typedef float*            p_float32_t;
typedef double*           p_float64_t;

/* ===== Compound data type ===== */

/* Function return type */
typedef enum
{
    Operation_Success = 1,
    Operation_Fail    = 0,
    Operation_Wait     = 2
} t_FuncRet;

/* Boolean type definition */
typedef enum
{
    FALSE = 0,
    TRUE  = 1
} bool;

```

以上数据变量命名方式遵循类型前缀缩写规定。

十一、宏定义

常量宏采用全部大写以及用 `_` 分隔符，尽量少用或者不用函数宏。

示例：

```
#define CM_7816_OK 0x9000;
```

十二、函数

函数接口的两个要素是参数和返回值。C 语言中，函数的参数和返回值的传递方式有两种：值传递（pass by value）和指针传递（pass by pointer）。

12.1 函数注释

模板如下：

```
/**
 * @description:
 * @param {数据类型}
 * @return {数据类型}
 * @author: leeqingshui
 */
```

12.2 参数规则

1. 参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字，如果函数没有参数，则用 void 填充；例如：

`void SetValue(int nWidth, int nHeight);` // 良好的风格

`void SetValue(int, int);` // 不良的风格

`float GetValue(void);` // 良好的风格

`float GetValue();` // 不良的风格

2. 参数命名要恰当，顺序要合理。

例如，编写字符串拷贝函数 `StringCopy`，它有两个参数，如果把参数名字起为 `str1` 和 `str2`，例如：`void StringCopy(char *str1, char *str2);` 那么我们很难搞清楚究竟是把 `str1` 拷贝到 `str2` 中，还是刚好倒过来， 可以把参数名字起得更有意义，如叫 `strSource` 和 `strDestination`。这样从名字上就可以看出应该把 `strSource` 拷贝到 `strDestination`。还有一个问题，这两个参数那一个该在前那一个该在后？参数的顺序要遵循程序员的习惯。

一般地，应将目的参数放在前面，源参数放在后面。

如果将函数声明为：

```
void StringCopy(char *strSource, char *strDestination);
```

别人在使用时可能会不假思索地写成如下形式：

```
char str[20];
```

```
StringCopy(str, "Hello World"); // 参数顺序颠倒
```

3. 如果参数是指针，且仅作输入用，则应在类型前加 `const`，以防止该指针在函数体内被意外修改。例如：

```
void StringCopy(char *strDestination, const char *strSource);
```

4. 避免函数有太多的参数，参数个数尽量控制在 5 个以内。如果参数太多，在使用时容易将参数类型或顺序搞错；

5. 尽量不要使用类型和数目不确定的参数；C 标准库函数 `printf` 是采用不确定参数的典型代表，其原型为：`int printf(const char *format[, argument]...)`；这种风格的函数在编译时丧失了严格的类型安全检查。

12.3 返回值规则

1. 不要省略返回值的类型；
2. 不可返回局部指针；

12.4 函数实现

1. 模块化设计：函数的功能要单一，不要设计多用途的函数；函数体的规模要小，尽量控制在 150 行代码之内。

2. 尽量避免函数带有“记忆”功能（全局变量、静态局部变量）。相同的输入应当产生相同的输出带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易理解又不利于测试和维护。在 C 语言中，函数的 `static` 局部变量是函数的“记忆”存储器。建议尽量少用 `static` 局部变量，除非必需。

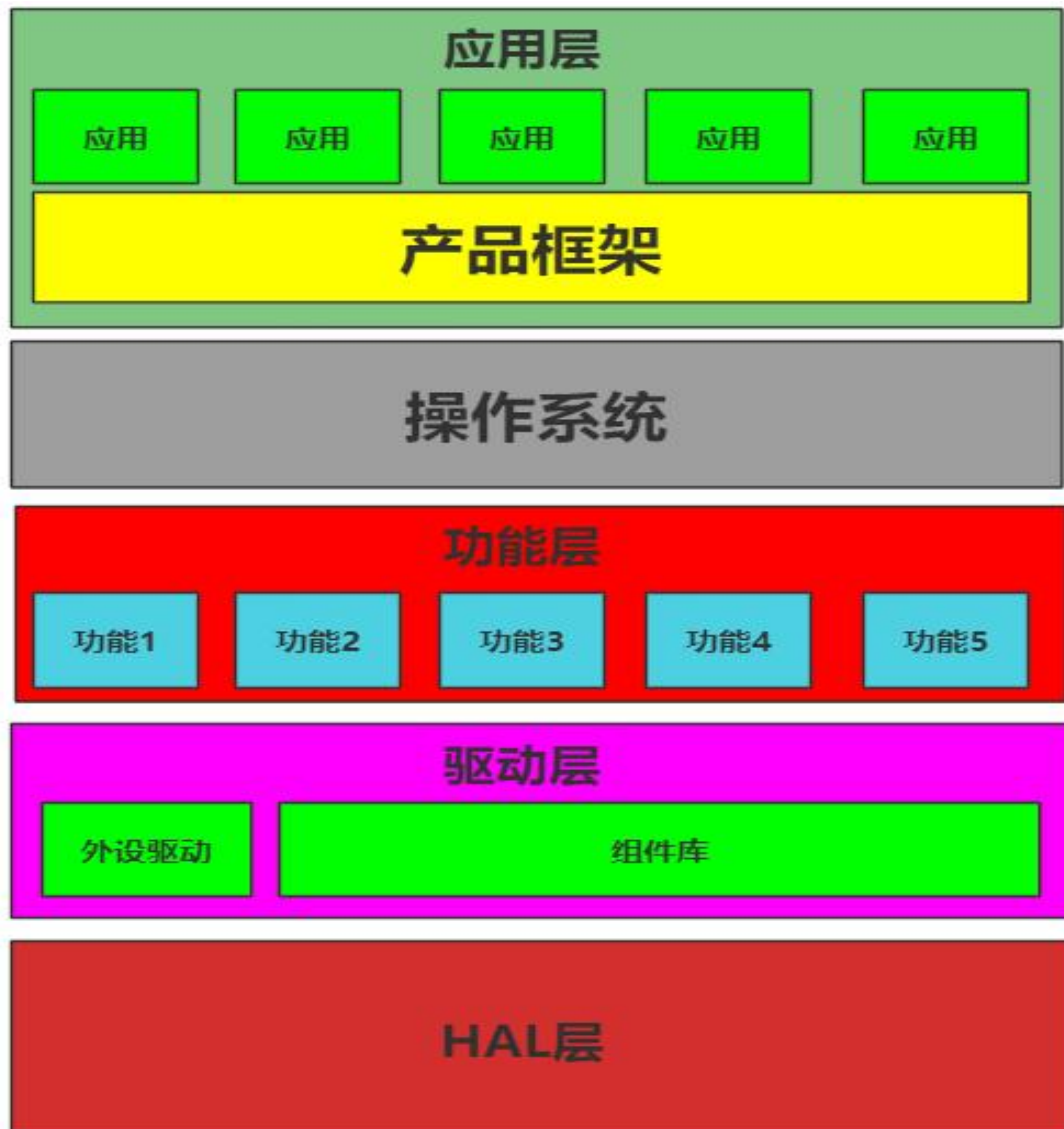
3. 在函数体的“入口处”，对参数的有效性进行检查；很多程序错误是由非法参数引起的，我们应该充分理解并正确使用“断言”（`assert`）来防止此类错误。同时，不仅要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

4. 用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。
5. 避免函数嵌套过深，新增函数代码嵌套不超过 4 层。
6. 设计高扇入，合理扇出（小于 7）的函数：

扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。扇出过大，表明函数过分复杂，控制和协调下级函数过多；扇出过小，表明函数的调用层次过多，不利于程序阅读和函数结构分析，并且运行时会对系统资源（如堆栈空间）造成压力。通常函数比较合理的扇出（调度函数除外）通常是 3~5。

十三、分层设计

为了便于系统移植，整体程序应采用分层设计：



十四、STM32Readme 文件模板

/*=====Project description=====*/

Function brief :

/*=====code layers=====*/

1. Folder organization :

2. code layer:

/*=====Development Environment Description=====*/

1. Development Board:

2. CPU internal resources :

- (1) CPU model:
- (2) Main frequency:
- (3) Internal Flash capacity:
- (4) Internal SRAM capacity:

3. Memory space allocation:

IROM1 : START - SIZE -
IRAM1 : START - SIZE -
STACK SIZE :
HEAP SIZE :

4. Software Development Related:

- (1) IDE :
- (2) Firmware Package Name and Version :
- (3) Compiler :
- (4) Common macro definition :

5. Debug Settings : ST-LINK/V2

- (1) SWD Req: Selected:
- (2) Connect & Reset Options :
- (3) Cache Options :
- (4) Trace Options :
- (5) Flash Download :

/*=====GPIO pin use with peripherals=====*/

/*===== peripherals Parameter =====*/

/*=====Key points=====*/