STM32　C 语言开发规范 V1.0.0 版本

开发环境

主要为 WIN10 及以上版本，所用到的集成开发环境有：

MDK V5.35 — 标准库和寄存器开发

CubeMX、Cube IDE — HAL 库开发

参考规范

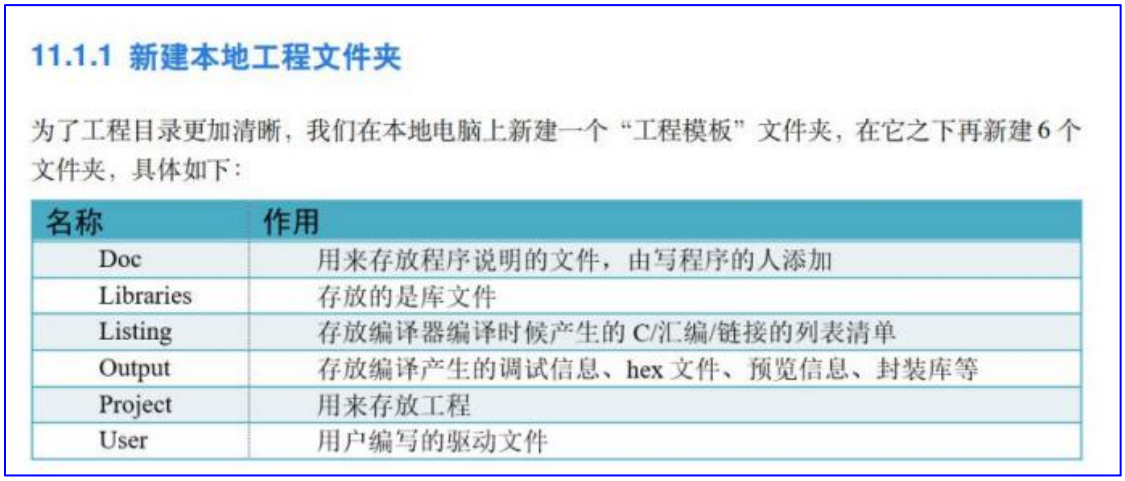
进行开发时参考 C99 标准，主要用到的 C99 特性包括：变长数组 VLA、restrict 关键字、inline 关键字、内联函数、_Bool 类型、可移植类型 stdint.h 和 inttypes.h。

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；

文件夹结构

对于标准库开发，文件夹结构如下：



11.1.1 新建本地工程文件夹

为了工程目录更加清晰，我们在本地电脑上新建一个“工程模板”文件夹，在它之下再新建 6 个文件夹，具体如下：

名称	作用
Doc	用来存放程序说明的文件，由写程序的人添加
Libraries	存放的是库文件
Listing	存放编译器编译时候产生的 C/汇编/链接的列表清单
Output	存放编译产生的调试信息、hex 文件、预览信息、封装库等
Project	用来存放工程
User	用户编写的驱动文件

在本地新建好文件夹后，把准备好的库文件添加到相应的文件夹下：

名称	作用
Doc	工程说明.txt
Libraries	CMSIS：里面放着跟 CM3 内核有关的库文件
	STM32F10x_StdPeriph_Driver：STM32 外设库文件
Listing	暂时为空
Output	暂时为空
Project	暂时为空
User	stm32f10x_conf.h：用来配置库的头文件
	stm32f10x_it.h stm32f10x_it.c：中断相关的函数都在这个文件编写，暂时为空
	main.c：main 函数文件

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，中文注释与英文注释各一份；

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。

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：

```
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// Copyright (c) 2004, Department of Mathematics, Zhejiang University
// All rights reserved.
//
// Filename : filename.h
// Project Code : The project code about this file
// Abstract : Describe the content of this file summarily
// Reference : ......
//
// Version : 1.1
// Author : the name of author(mender)// Accomplished date : September 2, 2004
//
// Replaced version : 1.0
// Original Author : the name of original author(mender)
// Accomplished date : September 10, 2003
//
// Main functions :
// Function 1 Return code
Function name(Parameter Explain)
// Function 2 Return code
Function name(Parameter Explain)
//
...
// Function n Return code
Function name(Parameter Explain)
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
```

☆	【规则 1.1-1】	文件信息声明以两行斜杠开始，以两行斜杠结束，每一行都以两个斜杠开始；
☆	【规则 1.1-2】	文件信息声明包含五个部分，各部分之间以一空行间隔；
☆	【规则 1.1-3】	在主要函数部分描述了文件所包含的主要函数的声明信息，如果是头文件，这一部分是可以省略的。

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用使用头文件，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 ifndef/define/endif 结构产生预处理块；“#ifndef”或者“#define”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“_”连接，例如有头文件名称为“filesystem.h”，则定义如下：“#ifndef _FILE_SYSTEM_H_”；
☆	【规则 1.2-2】	用 #include <filename.h> 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 #include “filename.h” 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

#ifndef _FILE_SYSTEM_H_

#define _FILE_SYSTEM_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

#include <math.h>

…

//   引用的非标准库头文件

#include “myheader.h”

…

typedef struct Student

{

…

}Stu;

//函数声明

void Function1(…);

…

#ifdef __cplusplus
}
#endif

#endif
```

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

源文件模板如下：

```
//   文件信息声明
//   引用的头文件
#include “filesystem.h”
…
//   引用的外部函数声明
//   宏定义
//   静态全局变量
//   非静态全局变量
//   内部函数定义
Static   void   static_Function1(…)
{
…
}

//   函数定义
void Function1(…)
{
…
}
```

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。

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

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 struct、union、class 变量的命名要求定义的类型用大写，结构采用 S 开头，联合体采用 U 开头，类采用 C 开头；

例如：

```
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
};
```

【规则 2.2-3】 指针变量命名的基本原则为：

一重指针变量的基本原则为：

变量名＝ “p”＋变量类型前缀＋命名

对多重指针变量的基本原则为：

二重指针：

变量名＝“pp”＋变量类型前缀＋命名

三重指针：

变量名＝“ppp”＋变量类型前缀＋命名

……

例如一个 short*型的变量应该表示为 pnStart；

【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】 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)，

即：变量名＝c_＋变量命名规则，例如：

```
const char* c_szFileName;
```

【规则 2.2-8】 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：

```
enum      EMDAYS
{
    EMDAYS_MONDAY;
    EMDAYS_TUESDAY;
    ……
};
```

☆ 【规则 2.2-9】 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：

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

☆ 【规则 2.2-10】 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。

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）

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：

```
/* 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;
```

```
/* 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;
```

以上数据变量命名方式遵循类型前缀缩写规定。

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;
```

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）。

函数注释

模板如下：

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

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 chat *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。

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；
3.

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。

分层设计

为了便于系统移植，整体程序应采用分层设计：