第 5 章

CHAPTER 5

STM32 单片机编程基础

早期单片机程序多采用汇编语言编写，其程序执行效率高，目标代码精短。由于汇编语言是面向机器的程序设计语言，用助记符代替机器指令的操作码，用地址符号或标号代替指令或操作数的地址，编写的代码非常难懂、不好维护，编程效率非常低。因此，现在汇编语言多被用在底层与硬件操作紧密相关或对性能有特殊要求的场合，而上层应用程序一般用 C 语言进行编写。C 语言是一种接近自然语言的高级语言，其编写的程序可读性高、维护容易、执行效率也比较高。单片机 C 语言是嵌入式 C 语言，遵循标准 C 语言的语法、格式、数据类型等，但也有小部分差异。本章将对 STM32 单片机 C 语言编程中经常使用到的基础 C 语言知识点进行回顾，为后续实验奠定理论基础。

5.1 C 语言关键字

C 语言简洁、紧凑，使用方便、灵活。ANSI C 标准 C 语言共有 32 个关键字、9 种控制语句，程序书写形式自由，且区分大小写。关键字是一类具有固定名称和特定含义的特殊标识符，又称为保留字。在编程时不允许将关键字另作他用，32 个关键字见表 5.1。

表 5.1 C 语言关键字

关键字	含　义	关键字	含　义
auto	声明自动变量	typedef	用以给数据类型取别名
char	声明字符型变量或函数返回值类型	extern	声明变量或函数是在其他文件定义的外部变量
short	声明短整型变量或函数返回值类型	void	声明函数无返回值或无参数，声明无类型指针
int	声明整型变量或函数返回值类型	sizeof	计算数据类型或变量长度(所占字节数)
long	声明长整型变量或函数返回值类型	switch	用于开关语句
float	声明浮点型变量或函数返回值类型	case	开关语句分支
double	声明双精度浮点型变量或函数返回值类型	default	开关语句中的默认分支
signed	声明有符号类型变量或函数	do	循环语句的循环体
unsigned	声明无符号类型变量或函数	while	循环语句的循环条件
const	声明只读变量	for	一种循环语句

续表

关键字	含　义	关键字	含　义
register	声明寄存器变量	continue	结束当前循环，开始下一轮循环
static	声明静态变量	break	跳出当前循环
volatile	声明变量在程序执行中可被隐含地改变	return	子程序返回语句(可以带参数，也可以不带参数)
enum	声明枚举类型	if	条件语句
struct	声明结构体类型	else	条件语句否定分支(与 if 连用)
union	声明共用体类型	goto	无条件跳转语句

1．const

const 是 constant 的缩写，用于声明常量(只读变量)或常数。在整个程序中，常数可以当作一个只读变量，在使用变量的任何地方均可使用，其值只读，但不能修改。在编程中涉及不变的数据时可以用关键字 const 进行修饰，表明该变量是一个常量，关键字 const 用在类型前面或后面是等价的，如下面两条语句是等价的：

```
const uint8_t TMP_VAL = 20;
uint8_t const TMP_VAL  = 20;
```

在声明只读变量时应尽量使用大写字母，以区别于普通变量。关键字 const 也可以修饰函数参数类型，表明该参数值在函数体内不期望被改变，例如：

```
void Function(const uint8_t x);
```

2．register

register 暗示编译器编译程序相应的变量时将被频繁地使用，如果可能的话，应将其保存在 CPU 的寄存器中，以加快其存储速度。register 变量必须是能被 CPU 所接受的类型，这意味着 register 变量必须是单个值，且长度应该小于或等于整型长度。随着编译器编译能力的提高，在决定哪些变量应该被存放到寄存器中时，C 编译环境能比程序员做出更好的决定，所有很少使用关键字 register 进行变量修饰。

3．static

static 在 C 语言中比较常用，主要用于定义局部静态变量、全局静态变量和静态函数。使用恰当能够大大提高程序的模块化特性，有利于程序扩展和维护。

1) 局部静态变量

局部变量是在函数内部定义的变量(其前不加 static 修饰)，存储在进程栈空间，其作用域也仅在此函数内有效，当函数结束退出时局部变量即被销毁释放。编译器一般不对局部变量进行初始化，即它的初始值编译时是不确定的，除非对其显示赋值初始化。

在局部变量前添加 static 关键字修饰时，即成为局部静态变量，此时该变量在进程的全局数据区分配内存空间，并始终驻留在全局数据区，直到程序运行结束。局部静态变量的作用域仍然是局部作用域，即在定义的函数内部有效，当退出函数时其作用域也随之结束，但其存储空间不释放。函数退出返回时它的值将保持，待下次再次进入此函数时，其值不被初始化显示赋值，仍然为前次函数运行退出前的值。换句话说，局部静态变量成为其作用域的全局变量，只有在整个程序退出时才会释放，其初值仅在第一次进入时被初始化，再次进入

将忽略赋值初始化语句。局部静态变量若定义时未显示赋值初始化，在编译时会自动初始化为0。

2）全局静态变量

全局变量定义在函数体外部，在整个工程中都可以被访问，且在全局数据区分配存储空间。定义时若未显示赋值初始化，在编译时会自动初始化为0。全局变量在一个文件中定义，其他文件可以使用extern外部声明后就可以直接使用。即在其他文件中不能再定义一个与其名字相同的变量，否则编译器会认为它们是同一个变量而报错。

在全局变量前添加static关键字修饰时，即成为全局静态变量，其作用域被限制在定义的文件内可以被访问，即使在其他文件使用extern外部声明也不能访问。在其他文件中可以定义与其同名的变量，两者互不影响。因此，此处static的作用就成了限定作用域，在定义不需要与其他文件共享的全局变量时，加上static关键字能够有效地降低程序模块之间的耦合，避免不同文件同名变量的冲突，且不会误使用。全局静态变量也在全局数据区分配存储空间，定义时若未显示赋值初始化，在编译时会自动初始化为0。

3）静态函数

函数的使用方式与全局变量类似，可以在另一个文件中直接引用，甚至不必使用extern进行声明。在函数的返回类型前加上static就是静态函数，其只能在声明它的文件中可见，其他文件不能引用该函数。不同文件可以定义相同名字的静态函数，互不影响。另外，在文件作用域中声明的inline函数默认为static类型。

4. volatile

volatile在嵌入式应用中常用于描述一个内存映射的I/O端口或硬件寄存器。凡是有关键字volatile修饰的变量，在用到这个变量时必须重新读取这个变量的值（从内存存储器单元重新读取值），即每次读/写都必须访问实际地址存储器的内容，而不是使用保存在寄存器中的备份。一般来说，volatile用在以下几个地方：

（1）中断服务程序中修改其值，并供其他程序检测的变量需要加volatile关键字进行修饰；

（2）存储器映射的硬件寄存器通常也要加volatile关键字进行修饰，用于强制每次访问时都要重新读写端口，因为每次对它的读写都可能有不同的值；

（3）在多任务环境下各任务间共享的标志应该加volatile关键字。

5.2 支持数据类型

C语言支持short、int、long、float、double、char六种基本数据类型，结合关键字signed和unsigned可以构成有符号和无符号数据类型。在基本数据类型的基础上，可以定义数组、结构体struct、共用体union、枚举类型enum、指针类型和空类型void，使得数据类型多且复杂。相同数据类型在不同机器内存中占据的字节长度是不一样的。在32位STM32单片机上，六种基本数据类型在内存中占据的字节数如下：

char——占据的内存大小是1B；

short——占据的内存大小是2B；

int——占据的内存大小是4B；

long——占据的内存大小是 4B；

float——占据的内存大小是 4B；

double——占据的内存大小是 8B。

为进一步明确各数据类型在机器中占据内存的字节宽度，在 ISO C99 中对基本数据类型进行了扩展，并在 stdint.h 头文件中进行了声明。例如：

```
/* exact-width signed integer types */
typedef signed char       int8_t;          //有符号 8 位整型
typedef signed short int  int16_t;         //有符号 16 位整型
typedef signed int        int32_t;         //有符号 32 位整型
typedef signed __INT64    int64_t;         //有符号 64 位整型

/* exact-width unsigned integer types */
typedef unsigned char     uint8_t;         //无符号 8 位整型
typedef unsigned short int uint16_t;       //无符号 16 位整型
typedef unsigned int      uint32_t;        //无符号 32 位整型
typedef unsigned __INT64  uint64_t;        //无符号 64 位整型
```

在 STM32 单片机的寄存器定义及存储区映射头文件 stm32f10x.h 中，保留了 STM32F10x 标准外设库(STM32F10x Standard Peripheral Library)中使用的旧数据类型声明。例如：

```
/*!< STM32F10x Standard Peripheral Library old types */
typedef int32_t           s32;
typedef int16_t           s16;
typedef int8_t            s8;

typedef const int32_t     sc32;            //定义"只读"权限
typedef const int16_t     sc16;            //定义"只读"权限
typedef const int8_t      sc8;             //定义"只读"权限

typedef __IO int32_t      vs32;
typedef __IO int16_t      vs16;
typedef __IO int8_t       vs8;

typedef __I int32_t       vsc32;           //定义"只读"权限
typedef __I int16_t       vsc16;           //定义"只读"权限
typedef __I int8_t        vsc8;            //定义"只读"权限

typedef uint32_t          u32;
typedef uint16_t          u16;
typedef uint8_t           u8;

typedef const uint32_t    uc32;            //定义"只读"权限
typedef const uint16_t    uc16;            //定义"只读"权限
typedef const uint8_t     uc8;             //定义"只读"权限

typedef __IO uint32_t     vu32;
typedef __IO uint16_t     vu16;
```

```
typedef __IO uint8_t      vu8;

typedef __I uint32_t      vuc32;          //定义"只读"权限
typedef __I uint16_t      vuc16;          //定义"只读"权限
typedef __I uint8_t       vuc8;           //定义"只读"权限
```

在 Cortex-M3 内核声明头文件 core_cm3.h 中，对__I 和__O 做了如下宏定义：

```
#ifdef __cplusplus
  #define    __I      volatile          //定义"只读"权限
#else
  #define    __I      volatile const    //定义"只读"权限
#endif
#define    __O        volatile          //定义"只写"权限
#define    __IO       volatile          //定义"读写"权限
```

在进行 STM32 单片机程序编写时，应尽量使用 ISO C99 扩展数据类型进行变量、常量的声明，以明确数据在内存中占据的字节数。

5.3 常用布尔型变量

在程序开发过程中，适当应用数据类型定义变量将会使程序撰写事半功倍。程序开发者可以根据自己的需求定义布尔型变量，使程序开发中的条件判断、赋值更加简单、含义明确。在 STM32 单片机的 stm32f10x.h 文件中定义了常用布尔型变量及枚举类型：

```
//用 typedef 关键字将枚举类型定义成别名 FlagStatus 和 ITStatus
//枚举变量 SET 为 1,RESET 为 0
typedef enum {RESET = 0, SET = !RESET} FlagStatus, ITStatus;
//用 typedef 关键字将枚举类型定义成别名 FunctionalState
//枚举变量 ENABLE 为 1,DISABLE 为 0
typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;
//用 typedef 关键字将枚举类型定义成别名 ErrorStatus
//枚举变量 SUCCESS 为 1,ERROR 为 0
typedef enum {ERROR = 0, SUCCESS = !ERROR} ErrorStatus;
```

上述三行代码定义了 FlagStatus、ITStatus、FunctionalState、ErrorStatus 四种枚举类型。在 STM32 单片机程序开发过程中可以用于定义标志状态、中断状态、函数功能状态、错误状态类型变量。同时还定义了 SET、RESET、ENABLE、DISABLE、SUCCESS、ERROR 六个具有一定含义的布尔变量，在 STM32 程序开发中可以应用这六个布尔变量进行赋值与判断，从而增强程序的可读性。

5.4 C 语言编程基础

C 语言是 STM32 单片机程序编写的基础，拥有扎实的 C 语言编程基础将能够快速、高效地编写出 STM32 单片机程序，并使程序代码精简，提高运行效率。

5.4.1 位运算

位运算是程序设计中对位模式按位或二进制数的一元和二元进行操作，可以对基本类型变量在位级别进行操作运算。位运算允许对一个字节或更大的数据单位中独立的位做处理，可以清除、设定、倒置任何位或多个位，也可以将一个整数的位向右或向左移动。C 语言支持六种位运算，如表 5.2 所示。在 STM32 单片机程序设计中，经常会使用位运算进行位设置、位清除、位取反、位移位等操作。

表 5.2 位运算符

运 算 符	含 义	运 算 符	含 义
&	按位与	~	取反
|	按位或	<<	左移
^	按位异或	>>	右移

1. 不改变其他位值的状况下，对某几个位进行设值

在 STM32 编程中经常需要对某个寄存器中的某位进行清零、置位操作，其方法是首先对需要设置的位用“&”运算符进行清零操作，然后用“|”运算符进行置位操作。例如，要改变 GPIOB 的输出数据寄存器 ODR 中某些位的状态，可以首先对寄存器的值进行“&”运算符清零操作，然后与需要设置的值用“|”运算符进行位设置：

```
GPIOB->ODR& = 0xFFFFFFF0;                    //对第 0～3 位清零
GPIOB->ODR| = 0x0000000A;                    //设置相应位的值，不改变其他位的值
```

2. 移位运算可提高代码可读性

移位操作在 STM32 单片机开发中非常重要，可以快速实现数据变换和乘除运算。移位运算符将左操作数的位模式移动数个位置，至于移动几个位置由右操作数指定。移位运算符的操作数必须是整数，右操作数不可以为负值，并且必须少于左边操作数的位长。如果不符合这些条件，程序运行结果将无法确定。移位运算结果的类型等于左操作数的类型：

```
GPIOB->BSRR = (((uint32_t)0x01) << pinpos);   // 将 BSRR 寄存器的第 pinpos 位设置为 1
GPIOB->ODR| = 1 << 5;                         // GPIOB.5 输出高，不改变其他位
```

3. 取反运算

STM32 单片机寄存器的每一位都代表一个状态，某个时刻希望设置某一位的值为 0，同时其他位都保留为 1，简单的做法是直接给寄存器设置一个值：

```
GPIOB->BSRR = 0xFFF7;                         //设置第 3 位为 0
```

直接给寄存器设置一个值可以实现所需功能，但是这样设置可读性很差。正常做法是首先通过宏定义定义一个值，以表达一定的含义；然后用取反运算符“～”实现值设置。例如：

```
#define LED1 ((uint16_t)0x0001)
#define LED2 ((uint16_t)0x0002)

GPIOB->BSRR = (uint16_t)~LED1;                //将第 0 位设置为 0
```

5.4.2　逻辑运算

C语言中提供了“&&”(与运算)、“‖”(或运算)、“!”(非运算)三种逻辑运算符,用于逻辑运算,详细说明如表5.3所示。

表5.3　逻辑运算符

运　算　符	说　　明	结　合　性	举　　例
&&	与运算,双目,对应数学中的“且”	左结合	1&&0、(9>3)&&(b>a)
‖	或运算,双目,对应数学中的“或”	左结合	1‖0、(9>3)‖(b>a)
!	非运算,单目,对应数学中的“非”	右结合	!a、!(2<5)

一般将零值称为“假”,将非零值称为“真”,逻辑运算的结果也只有“真”和“假”。在STM32单片机编程中“真”对应的值为1,“假”对应的值为0。灵活使用逻辑运算符构成逻辑表达式,用于各种条件控制的条件语句。

5.4.3　宏定义

宏定义是C语言中的预处理命令,其关键字为define,使用宏定义可以提高源代码的可读性,为编程提供方便。其格式如下:

```
#define     宏名      字符串
#:表示这是一条预处理命令,所有的预处理命令都以 # 开头。
宏名:是标识符的一种,命名规则和变量相同。
字符串:可以是常数数字、格式串、表达式、if 语句、函数等。
#define SYSCLK_72MHz 72000000                    //定义标识符 SYSCLK_72MHz 的值为 72000000
```

宏名可以带参数,此时就是带参数的宏定义,宏名中不能有空格,宏名与形参表之间也不能有空格,而形参表中形参之间可以有空格。例如:

```
#define LED_ONOFF(GPIOX,Pin,n)  GPIO_WriteBit(GPIOX,Pin,(BitAction)n);
```

宏定义也可以定义表达式或多个语句,例如:

```
#define AB(a,b) a = i + 5;b = j + 3;              //定义多个语句
```

5.4.4　条件编译

STM32单片机程序开发过程中,经常会遇到当某条件满足时对一组语句进行编译,而当条件不满足时则编译另一组语句。条件编译命令最常见的形式如下:

```
#ifdef 标识符
  程序段 1
#else
  程序段 2
#endif
```

它的作用是:当标识符已经被定义过(一般是用#define命令定义),则对程序段1进行编译;否则,编译程序段2。其中#else部分也可以没有:

```
#ifdef 标识符
  程序段 1
#endif
```

在程序实现代码对应的头文件中，必须包含这样的条件编译，例如：

```
#ifndef __LED_H
#define __LED_H

#include "stm32f10x.h"

//参数 GPIOX:LED 连接的 GPIO 端口
// Pin : LED 连接的具体引脚 GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5
// n : n = 1 LED 灭 ;n = 0 LED 亮
#define LED_ONOFF(GPIOX,Pin,n) GPIO_WriteBit(GPIOX,Pin,(BitAction)n);

void LED_Init(void);
void LED_Ctr_M1(void);
void LED_Ctr_M2(void);

#endif
```

在上述头文件编译时，编译器首先判断标识符__LED_H 是否已经被定义过，然后据此决定是否对下面的代码进行预编译。如果已经定义过__LED_H，则该头文件不会被编译。如果__LED_H 未被定义过，则该头文件内的代码会被编译。

条件编译在 STM32 单片机的寄存器定义及存储器映射头文件 stm32f10x.h 中经常会看到，如下面的语句用于定义中等容量芯片的片上资源。

```
#ifdef STM32F10X_MD      // STM32F10X_MD 是通过#define 定义的预处理符号中等容量芯片需要
                         // 的一些变量定义
#endif
```

5.4.5 结构体

在 C 语言中可以使用结构体(struct)来存放一组不同类型的数据。结构体的定义形式如下：

```
struct 结构体名{
  结构体所包含的变量或数组;
}变量名列表;
```

结构体是一种集合，它里面包含了多个变量或数组，它们的类型可以相同，也可以不同，每个这样的变量或数组都称为结构体的成员。在定义结构体的同时可以定义结构体变量，且将变量名放在结构体定义的最后。例如：

```
struct stu{
  char *name;                          //姓名
  int num;                             //学号
  char group;                          //所在学习小组
  float score[2];                      //成绩
} stu1, *stu2;
```

stu为结构体名,它包含了4个成员,分别是name、num、group、score。结构体成员的定义方式与变量和数组的定义方式相同,只是不能初始化。结构体也是一种数据类型,定义结构体后,可用它直接定义结构体变量、结构体指针等,如结构体变量stu1、结构体指针*stu2。定义结构体类型时变量名列表可以省略,在定义具体变量时再给出。例如:

```
struct stu{
  char   * name;                                    //姓名
  int  num;                                         //学号
  char  group;                                      //所在学习小组
  float score[2];                                   //成绩
};
```

上述代码定义了结构体类型,若要定义结构体变量stu1和结构体变量指针stu2,可以随后这样定义(注意关键字struct不能少):

```
struct stu stu1, * stu2;
```

对于结构体变量成员需要使用“.”运算符进行引用,结构体指针变量成员需要使用“箭头”运算符进行引用。例如:

```
stu1.num = 12;
stu2 -> num = 12;
```

在STM32单片机程序开发过程中,经常会对片上外设进行初始化,而一个外设的初始化经常由几个属性来决定。如串口初始化涉及串口号、波特率、奇偶校验、工作模式等,对于这种情况,不使用结构体变量来保存参数对其进行初始化,一般方法是:

```
void USART_Init(uint16_t usartx, uint32_t BaudRate, uint8_t parity, uint8_t mode);
```

这种方式是有效的,同时在一定场合是可取的。但如果希望往这个函数里面再传入一个参数,那么必须修改这个函数的定义,重新加入新的入口参数。如加入传送的字符长度,于是修改函数如下:

```
void USART_Init(uint16_t usartx, uint32_t BaudRate, uint8_t parity, uint8_t mode, uint8_t
wordlength);
```

但是,如果这个函数的入口参数是随着开发不断地增多,那么是否就要不断地修改函数的定义呢?这是否给开发带来很多的麻烦?怎样解决这种情况呢?

如果使用结构体就能解决这个问题,可以在不改变入口参数的情况下,只需要改变结构体的成员变量,就可以达到上面改变入口参数的目的。结构体就是将多个变量组合为一个有机的整体。参数BaudRate、parity、mode、wordlength对于串口而言是一个有机整体,都是来设置串口的属性参数的,所以可以通过定义一个结构体将它们组合在一起。在MDK的串口初始化函数中是这样定义的:

```
typedef struct
{
  uint32_t USART_BaudRate;
  uint16_t USART_WordLength;
  uint16_t USART_StopBits;
```

```
  uint16_t USART_Parity;
  uint16_t USART_Mode;
  uint16_t USART_HardwareFlowControl;
} USART_InitTypeDef;
```

于是在初始化串口时，入口参数就可以用 USART_InitTypeDef 类型的变量或者指针变量了。MDK 中是这样做的：

```
void USART_Init(USART_TypeDef * USARTx, USART_InitTypeDef * USART_InitStruct);
```

这样，任何时候只需要修改结构体成员变量，往结构体中加入新的成员变量，而不需要修改函数定义就可以达到修改入口参数的目的。这样的好处是不用修改任何函数定义就可以达到增加变量的目的。使用结构体组合参数可以提高代码的可读性，不会让人觉得变量定义混乱。

5.4.6 类型定义

C 语言允许用户使用 typedef 关键字来定义自己习惯的数据类型名称，以替代系统默认的基本数据类型名称、数组类型名称、指针类型名称与用户自定义的结构体类型名称、共用型名称、枚举型名称等。一旦用户在程序中定义了自己的数据类型名称，就可以在程序中用自己的数据类型名称来定义变量的类型、数组的类型、指针变量的类型与函数的类型等。代码如下：

```
#define TRUE 1
#define FALSE 0
typedef uint16_t BOOL;                          //自定义一个布尔数据类型 BOOL
BOOL bflag = TRUE;
```

typedef 用于为现有类型创建一个新的名字，用来简化变量的定义。typedef 在 MDK 用得最多的就是定义结构体的类型别名和枚举类型。

若定义一个结构体：

```
struct _GPIOInit
{
    uint16_t GPIO_Pin;
    GPIOSpeed_TypeDef GPIO_Speed;
    GPIOMode_TypeDef GPIO_Mode;
};
```

用该结构体定义变量的方式：

```
struct _GPIOInit GPIOInitStruct;                //定义结构体变量 GPIOInitStruct
```

上述方式定义变量很烦琐，MDK 中有很多这样的结构体变量需要定义，故可以为结构体定义一个别名 GPIO_InitTypeDef，这样就可以在其他地方通过别名 GPIO_InitTypeDef 来定义结构体变量。具体方法如下：

```
typedef struct
{
```

```
    uint16_t GPIO_Pin;
    GPIOSpeed_TypeDef GPIO_Speed;
    GPIOMode_TypeDef GPIO_Mode;
}GPIO_InitTypeDef;
```

随后可以使用结构体类型名 GPIO_InitTypeDef 进行变量定义：

```
GPIO_InitTypeDef GPIOInitStruct;                    //定义结构体变量 GPIOInitStruct
```

5.4.7 外部变量声明

C 语言中 extern 可以置于变量或函数前，以表示变量或函数在别的文件中进行的定义，提示编译器遇到此变量和函数时需要在其他文件模块中寻找其定义。需要注意，用 extern 声明的变量可以多次进行声明，但其定义只能有一次。在代码中会看到这样的语句：

```
extern uint16_t USART_RXTX_FLAG;
```

这个语句是声明 USART_RXTX_FLAG 变量是在其他文件中已经定义，在这里要使用到。所以，肯定可以在某个文件的某个位置找到 USART_RXTX_FLAG 变量定义语句：

```
uint16_t USART_RXTX_FLAG;
```

下面通过一个例子说明其使用方法。在 main.c 文件中定义全局变量 RUN_Flag，并在 main.c 里面进行 RUN_Flag 的初始化。例如：

```
main.c 文件
uint8_t RUN_Flag;                                   //定义只允许一次
main()
{
  RUN_Flag = 1;
  if(RUN_Flag)
    printf("System Run Flag :d %",RUN_Flag);
  else
    printf("System Run Flag :d %",RUN_Flag);
}
```

另外，假设在 Led.c 的 Run_LedFlash(void)函数中要使用变量 RUN_Flag，这个时候就需要在 Led.c 文件开头处声明变量 RUN_Flag 是外部定义的变量。因为如果不声明为外部变量，变量 RUN_Flag 的作用域就到不了 Led.c 文件中。下面是 Led.c 中的代码：

```
extern uint8_t RUN_Flag;                            //声明变量 RUN_Flag 是在其他文件定义的变量
void Run_LedFlash (void)
{
RUN_Flag = 0;
}
```

在 Led.c 中声明变量 RUN_Flag 在外部定义，然后在 Led.c 中就可以使用变量 RUN_Flag。

另外，extern 还可以声明函数在外部文件中定义的应用，这里不再赘述。

5.5 本章小结

本章简要介绍了 STM32 单片机 C 程序设计中使用的编程基础知识，对 C 语言关键字、C 语言六种基本数据类型、STM32 单片机固件库函数中使用的扩展数据类型等进行了简单介绍与说明，并对 C 语言编程中使用的位运算、逻辑运算、宏定义、条件编译、类型定义、结构体、外部变量声明等进行了回顾，为后续实验实战奠定了 C 语言基础知识。