嵌入式系统原理及实验

顾　震

信息科学与工程学院自动化系

华东理工大学

Email: guzhen@ecust.edu.cn

课程大纲

1. 嵌入式系统导论
2. Cortex-M3微处理器
3. STM32最小系统及开发环境
4. 嵌入式C语言
5. 通用输入输出GPIO模块
6. 中断
7. 定时器原理与应用
8. USART通信原理及实现
9. DMA控制器
10. SPI与I2C通信原理及实现
11. 模数转换原理及实现
12. 人工智能辅助的嵌入式项目开发
13. 嵌入式应用前沿

4. 嵌入式C语言

本章知识与能力要求

- 理解和掌握嵌入式C语言的程序结构；
- 掌握嵌入式C语言的数据类型、const、volatile和extern等关键字的使用；
- 掌握嵌入式C语言的条件编译；
- 掌握嵌入式C语言指针的应用

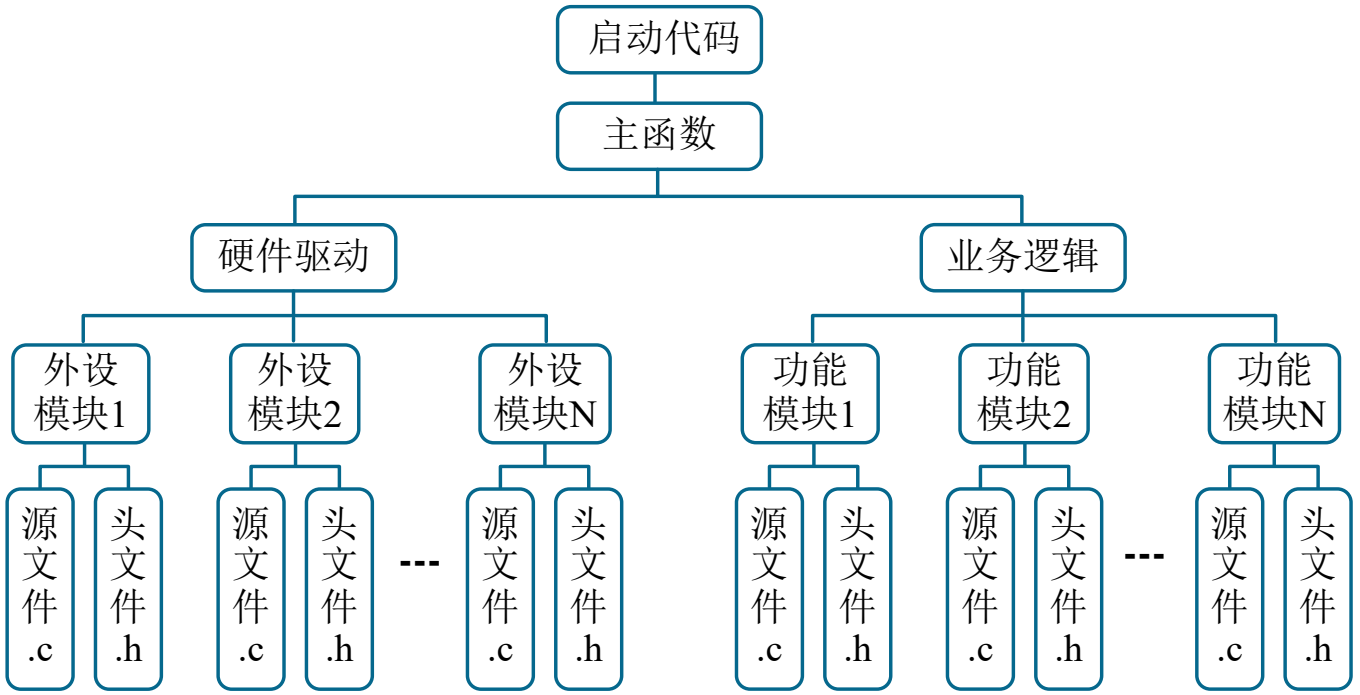
4 嵌入式C语言

- 嵌入式开发中既有**底层硬件**的开发又涉及**上层应用**的开发，即涉及系统的硬件和软件，C语言既具有汇编语言**操作底层**的优势，又具有高级语言**功能性强**的特点。

1. 程序总是从main函数开始执行；
2. 函数是C语言的基本结构；
3. 函数由两部分组成：函数说明部分和函数体。
4. 一个C语言程序包含若干个源程序文件（.c文件）和头文件（.h文件）。
5. 采用**外设功能模块化设计**方法，一个外设功能模块包括一个源文件（.c文件）和一个头文件（.h文件）。

4 嵌入式C语言

- 嵌入式系统程序的开发多采用**模块化**、**层次化**的设计思想，系统层次架构清晰，便于协同开发。

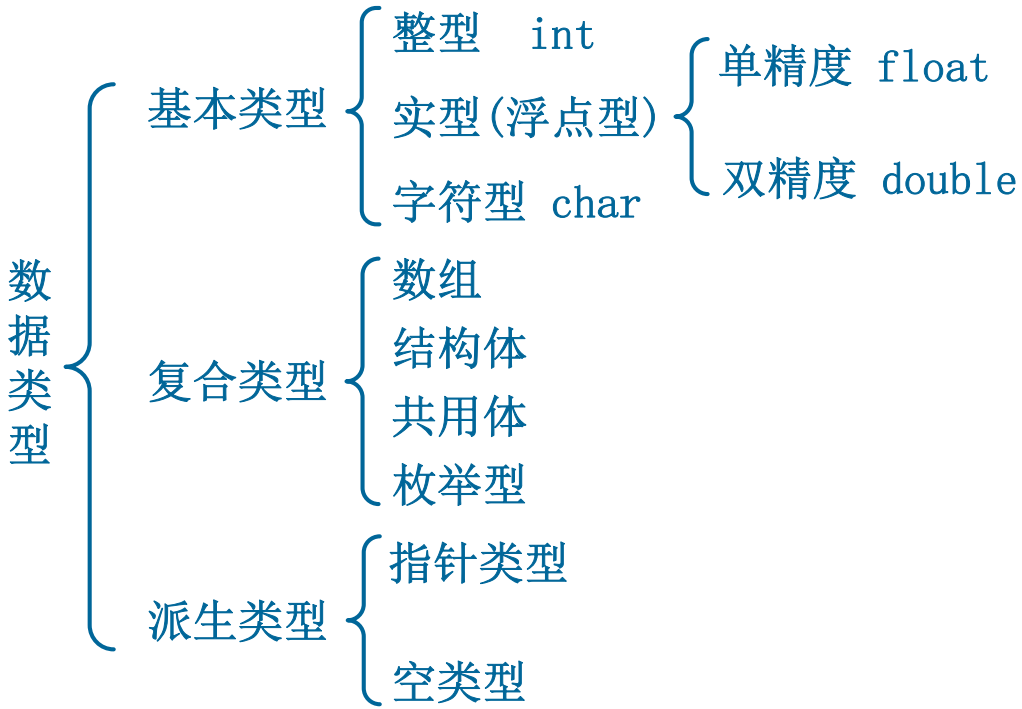


嵌入式系统软件结构框图

4 嵌入式C语言

4.1 STM32的数据类型

- 数据是嵌入式C语言的基本操作对象。
- 数据类型是指数据在计算机内存中的存储方式。



嵌入式C语言的数据类型

4.1 STM32的数据类型

- 不同CPU所定义的数据类型的长度不一致，ST公司为开发人员提供了基于C语言的标准外设库，标准外设库中定义的**数据类型长度**如下表所示。

C语言的数据类型	STM32对应的数据类型	说明
unsigned char	uint8_t	8位无符号数据（0～255）
unsigned short int	uint16_t	16位无符号数据（0～65535）
unsigned int	uint32_t	32位无符号数据（0～2^{32}-1）
unsigned long long	uint64_t	64位无符号数据（0～2^{64}-1）
signed char	int8_t	8位有符号数据（-128～+127）
signed short int	int16_t	16位有符号数据（-32768～+32767）
signed int	int32_t	32位有符号数据（-2^{31}～2^{31}-1）
signed long long	int64_t	64位有符号数据（-2^{63}～2^{63}-1）

HAL库的数据类型基于标准外设库

4.1 STM32的数据类型

- v3.5.0版本标准外设库已不再使用旧的数据类型，为了兼容以前的版本，stm32f10x.h头文件还对标准外设库之前版本所使用的数据类型进行了说明。

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

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

typedef __I int32_t vsc32;  /*!< Read Only */
typedef __I int16_t vsc16;  /*!< Read Only */
typedef __I int8_t vsc8;   /*!< Read Only */
```

STM32开发中同一数据类型有多种表示方式。

例

无符号8位整型数据有unsigned char，uint8_t，u8三种表示方式。

最新的v3.5.0版本采用标准的C99标准，即uint8_t方式。

4.1 STM32的数据类型

__I、__O以及__IO为IO类型限定词，内核头文件core_cm3.h定义了标准外设库所使用的IO类型限定词。

- 数据类型和IO类型限定词结合在一起，在标准外设库中常用来定义寄存器和结构体变量：

IO类型限定词	类型	说明
__I	volatile const	只读操作
__O	volatile	只写操作
__IO	Volatile	读和写操作

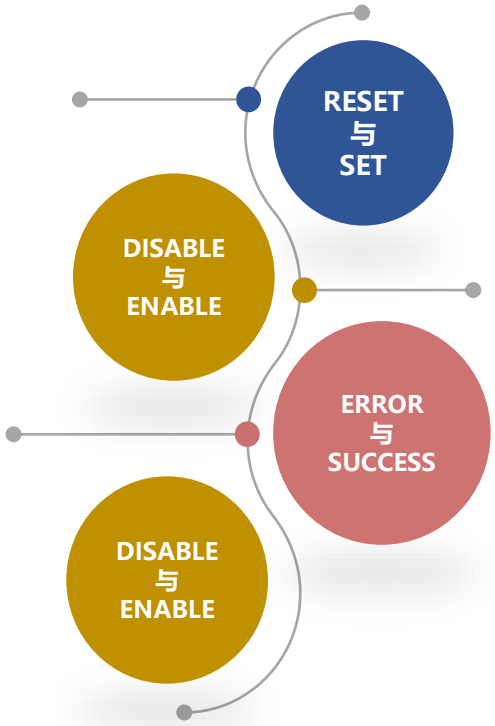
```
1000
1001  typedef struct
1002  {
1003    __IO uint32_t CRL;
1004    __IO uint32_t CRH;
1005    __IO uint32_t IDR;
1006    __IO uint32_t ODR;
1007    __IO uint32_t BSRR;
1008    __IO uint32_t BRR;
1009    __IO uint32_t LCKR;
1010  } GPIO_TypeDef;
```

4.1 STM32的数据类型

stm32f10x.h头文件中常用的布尔类型变量的定义

DISABLE、ENABLE、RESET、SET、SUCCESS、ERROR

```
typedef enum
{   DISABLE = 0,
    ENABLE = !DISABLE
} FunctionalState;
```




```
typedef enum
{   RESET = 0,
    SET = !RESET
} FlagStatus, ITStatus;
```

```
typedef enum
{  ERROR = 0,
    SUCCESS = !ERROR
} ErrorStatus;
```

```
#define IS_FUNCTIONAL_STATE(STATE)
(((STATE) == DISABLE) || ((STATE) == ENABLE))
```

4.2 const关键字

作用

用于定义**只读**的变量，其值在编译时不能被改变

要求

const关键词修饰的变量在声明时**必须初始化**

目的

为了在编译时**防止变量的值被误修改**，提高程序的安全性和可靠性。

属性

在C标准中，const定义的变量是**全局**的。

格式

const 常量类型 常量名=常量表达式;

例： const uint32_t a=0xffff1111;

4.3 static关键词

作用

修饰变量或函数。修饰后的变量称为**静态变量**。

操作

在全局变量之前加上关键字static，则该全局变量被定义成为一个**静态全局变量**。

目的

作用范围**只在定义该变量的源文件内有效**，其他源文件不能引用该全局变量，避免了在其他源文件中**因引用相同名字的变量而引发错误**，有利于模块化程序设计。

4.3 static关键词

static编程要点1：

模块化的程序设计中，用static声明一个函数，则该函数**只能被该模块内的其它函数调用**。

文件stm32f1xx_hal_dma.c中代码

```
#include "stm32f1xx_hal.h"
static void DMA_SetConfig(DMA_HandleTypeDef *hdma, uint32_t SrcAddress, uint32_t DstAddress, uint32_t DataLength);
… …
HAL_StatusTypeDef HAL_DMA_Start_IT(DMA_HandleTypeDef *hdma, uint32_t SrcAddress, uint32_t DstAddress, uint32_t DataLength)
{
  HAL_StatusTypeDef status = HAL_OK;
… …
  if(HAL_DMA_STATE_READY == hdma->State)
  {
DMA_SetConfig(hdma, SrcAddress, DstAddress, DataLength);
… …
}
… …
}
```

解析： DMA_SetConfig()函数只能被stm32f1xx_hal_dma.c的其它函数调用，不能被其它模块的文件使用。

4.3 static关键词

static编程要点2：

static除了用于静态全局变量，还用于定义静态局部变量，保证静态局部变量在**调用过程中不被重新初始化**。

```
void fun_count( )
{
   static count_num = 0;
        // 声明一个静态局部变量，count_num用作计数器，初值为0。
   count_num ++;
   printf("%d\n", count_num);
}
int main(void)
{
   int i=0;
   for (i = 0;i <= 5;i++)
   {
     fun_count( );
   }
   return 0;
}
```

在main函数中每调用一次fun_count( )函数，则静态局部变量count_num加1，而不是每次都被初始化为初值0。

4.4 volatile关键字

作用

使用volatile就是**不让编译器进行优化**，即**每次读取或者修改值**的时候，都必须重新**从内存中读取或者修改**，而不是使用保存在寄存器里的备份。

使用方式

一个类型修饰符，“易变的”。

volatile char i;

使用volatile关键字定义了一个字符型的变量i，指出i是随时可能发生变化的，每次使用的时候都必须从i的地址中读取。

应用场景

- 中断服务程序中修改的**供其他程序检测的变量**需要使用volatile；
- 多任务环境下**各任务间共享的标志**应添加volatile；
- **存储器映射的硬件寄存器**通常也要加volatile进行说明。

4.5 extern关键词

使用extern是一个声明而不是重新定义

作用

指明此函数或变量的定义在别的文件中，提示编译器遇到此函数或变量时去其他模块中寻找其定义。

```
extern  int  a;
```

声明变量a，而不是在定义变量a，并未为a分配内存空间

```
extern  int  funA( );
```

声明函数funA( )，此函数已在其他文件中定义。

extern "C"进行链接指定，告知编译器这是采用C语言定义的函数

4.5 extern关键词

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

解析：如果定义了__cplusplus（C++编译器中自定义的宏），则执行extern“C”{语句。C++支持函数重载，而C语言不支持函数重载，在C++环境下使用C函数会出现链接时找不到对应函数的情况，这时需要使用extern “C”进行链接指定，告知编译器使用C语言的命名规则来处理函数。

当函数有可能被C语言或C++使用时，将函数声明放在extern “C”中以免出现编译错误。

```
#ifdef __cplusplus
extern "C"{
#endif
//函数声明
#ifdef __cplusplus
}
#endif
```

4.5 extern关键词

为保证全局变量和功能函数的使用，extern一般用在.h头文件中对某个模块提供给其它模块调用的外部函数及变量进行声明，实际编程中只需要将该.h头文件包含进该模块对应的.c文件，即在该模块的.c文件中加入代码#include "xxx.h"。

ADCx.h

```
//声明全局变量
extern  uint16_t  ADC_ConvertedValue; //存放ADC的转换结果
… …
//外部功能函数声明
extern  void  ADC_Init(void);// ADC初始化函数
… …
```

ADCx.c

```
#include  "ADCx.h"
… …
   //定义变量
static uint16_t  pwd=1;  //局部变量，仅作用于本函数
uint16_t  ADC_ConvertedValue; //全局变量
… …
void  ADC_Init(void)
{
  … …
}
… …
```

4.6 struct结构体

- **结构体**一种用户自定义的可用的数据类型，它允许存储不同类型的数据项。

作用

struct用于定义结构体类型，其作用是将不同数据类型的数据组合在一起，构造出一个新的数据类型。

格式

struct一般用法为：

```
struct [结构体名]
{
    类型标识符  成员名1;
    类型标识符  成员名2;
        … …
}结构体变量;
```

举例

```
struct person
{   char name[8];
    int age;
    char sex[8];
    char address[20];
}person_liu;
```

结构体中的数据成员可以是基本数据类型（如 int、float、char 等），也可以是其他结构体类型、指针类型等。

4.6 struct结构体

- **结构体变量**（如上例中的person_liu）可以不在定义结构体时定义，后续需要时再进行定义。

定义格式

struct 结构体名 结构体变量;

使用这种定义方式可以很方便地同时定义多个结构体变量。

```
struct person
{
        char name[8];
        int age;
        char sex[8];
        char address[20];
};

struct person person_liu, person_zhang;
```

结构体变量的使用采用如下形式：

结构体变量名.成员名

如：**person_liu.age=35;**

4.6 struct结构体

练习：设计一个用于描述**温度传感器**状态的结构体，其量程为-50~100℃，精度为0.1℃，系统最多连接256个该种传感器

包括信息：

1. 传感器编号
2. 测量温度值(单位℃)
3. 传感器采样周期(1~20s，以1s为间隔)
4. 是否超温报警
5. 超温报警阈值
6. 报警状态
7. 传感器备注(最大20个字符)

```
struct temperatureSensor
{
        uint8_t index;
        int16_t value; //or float value
        uint8_t interval;
        uint8_t isAlarm; // 0 or 1
        int16_t threshold;
        uint8_t alarmState; // 0 or 1
        char comment[20];
};
```

4.7 enum枚举

enum枚举类型的用法：

```
enum 枚举名
{
    枚举成员1,
    枚举成员2,
     ……
}枚举变量;
```

```
enum Weekdays
{
      Monday = 1,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday,
      Sunday
}Mydays，Olddays;
```

- 枚举类型具有自动编号的功能，第一个枚举成员其默认值为整型的0，后续枚举成员的值在前一个成员上加1。
- 枚举成员的值是常量，不是变量，不能被赋值，但可以将枚举值赋给枚举变量。

用来将一个变量或对象的所有可能的值一一列出，**变量取值只限于列举出来的值**。

示例：

```
enum Weekdays classDay;
classDay = Tuesday;
printf("%d", classDay); // 显示2
```

可以自定义枚举成员的值，如果把第一个枚举元素的值定义为1，那么第二枚举成员的值就为2，以此类推，如上述例子中Friday的值为5。

4.8 typedef

作用

- 用来为复杂的声明**定义一个简单的别名**，方便记忆

目的

- 给变量起一个容易记且意义明确的新名字；
- 简化一些比较复杂的类型声明。

不是一个真正意义上的新类型

用法一：typedef的基本应用

格式：

```
typedef 类型名 自定义的别名;
```

为已知的数据类型起一个简单的别名

举例：

```
typedef signed char int8_t;
//给数据类型signed char起个别名int8_t
typedef signed int int32_t;
//给数据类型signed int起个别名int32_t
```

4.8 typedef

用法二：与结构体struct结合使用

STM32标准外设库中stm32f10x_gpio.h头文件中利用结构体别名GPIO_InitTypeDef

```
typedef struct
{
    uint16_t  GPIO_Pin;
    GPIOSpeed_TypeDef  GPIO_Speed;
   GPIOMode_TypeDef  GPIO_Mode;
}GPIO_InitTypeDef;
```

使用typedef为这个新建的结构体起了一个新的名字叫GPIO_InitTypeDef

则可以使用GPIO_InitTypeDef定义一个变量GPIO_InitStructure，从而调用GPIO_Mode。

```
GPIO_InitTypeDef  GPIO_InitStructure;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
```

不用额外再加**struct**关键字

4.8 typedef

用法三：与enum结合使用

stm32f10x_gpio.h头文件中的代码。

```
typedef  enum
{
        GPIO_Speed_10MHz = 1,
        GPIO_Speed_2MHz,
        GPIO_Speed_50MHz
}GPIOSpeed_TypeDef;
```

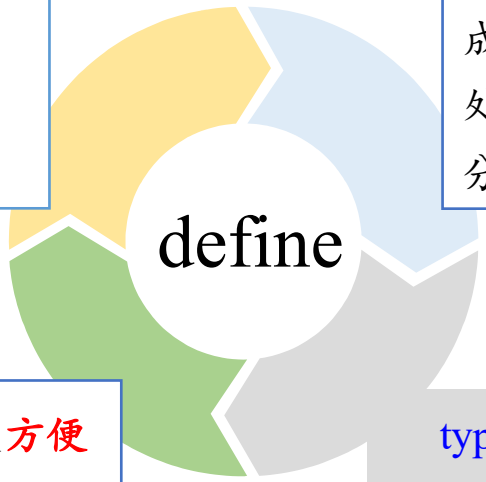
解析1：利用typedef关键字将此枚举类型定义一个别名GPIOSpeed_TypeDef，这里省略了枚举类型的枚举名，只用typedef起了个别名。

解析2：enum枚举类型共三个成员，并将第一个枚举成员GPIO_Speed_10MHz赋值为1，enum枚举类型会将枚举成员的赋值在第一个枚举成员赋值的基础上加1，因此，GPIO_Speed_2MHz默认值为2。

4.9 #define

#define是C语言中的预处理命令，它用于**宏定义**，用来**将一个标识符定义为一个字符串**，该标识符称为宏名，被定义的字符串称为**替换文本**。

所谓预处理是指在**编译之前**所做的工作，由预处理程序负责完成，编译时，系统将自动引用预处理程序对源程序中的预处理部分进行处理。



采用宏定义的目的主要是**方便程序编写**，一般放在源文件的前面，称为预处理部分。

typedef与#define的区别：

- typedef是在编译阶段处理的；
- #define是在预处理阶段处理的。

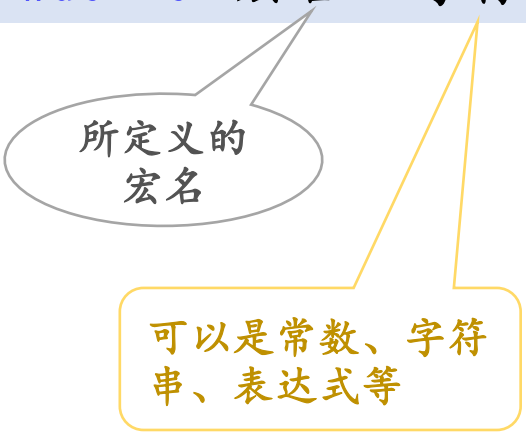
计算机科学里的宏是一种抽象的，根据一系列预定义的规则进行文本替换。

4.9 #define

用法一：无参数宏定义

定义格式：

#define <宏名><字符串>



例1：#define UINT8_MAX 255

解析：定义宏名UINT8_MAX，代表255。

例2：#define RCC_AHBPeriph_DMA1 ((uint32_t)0x00000001)

解析：定义RCC_AHBPeriph_DMA1宏名，代表32位的无符号数据0x00000001。

4.9 #define

用法一：无参数宏定义

标准外设库v3.5.0的stm32f10x_rcc.h文件中APB2_peripheral外设基地址的定义

```
]/** @defgroup APB2_peripheral
  * @{
  */

#define RCC_APB2Periph_AFIO              ((uint32_t)0x00000001)
#define RCC_APB2Periph_GPIOA             ((uint32_t)0x00000004)
#define RCC_APB2Periph_GPIOB             ((uint32_t)0x00000008)
#define RCC_APB2Periph_GPIOC             ((uint32_t)0x00000010)
#define RCC_APB2Periph_GPIOD             ((uint32_t)0x00000020)
#define RCC_APB2Periph_GPIOE             ((uint32_t)0x00000040)
#define RCC_APB2Periph_GPIOF             ((uint32_t)0x00000080)
#define RCC_APB2Periph_GPIOG             ((uint32_t)0x00000100)
#define RCC_APB2Periph_ADC1              ((uint32_t)0x00000200)
#define RCC_APB2Periph_ADC2              ((uint32_t)0x00000400)
#define RCC_APB2Periph_TIM1              ((uint32_t)0x00000800)
#define RCC_APB2Periph_SPI1              ((uint32_t)0x00001000)
#define RCC_APB2Periph_TIM8              ((uint32_t)0x00002000)
#define RCC_APB2Periph_USART1            ((uint32_t)0x00004000)
#define RCC_APB2Periph_ADC3              ((uint32_t)0x00008000)
#define RCC_APB2Periph_TIM15             ((uint32_t)0x00010000)
#define RCC_APB2Periph_TIM16             ((uint32_t)0x00020000)
#define RCC_APB2Periph_TIM17             ((uint32_t)0x00040000)
#define RCC_APB2Periph_TIM9              ((uint32_t)0x00080000)
#define RCC_APB2Periph_TIM10             ((uint32_t)0x00100000)
#define RCC_APB2Periph_TIM11             ((uint32_t)0x00200000)
```

APB2_peripheral各外设基地址的定义

4.9 #define

用法二：带参数的宏定义

定义格式：

#define<宏名>（参数1，参数2，...参数n）<替换列表>

例4：#define SUM(x,y) (x+y)

……

a = SUM(2,2);

解析：将SUM(x,y)定义为x+y，预编译时会将SUM(x,y)替换为x+y，a的结果是4

例5：#define IS_GPIO_SPEED(SPEED) (((SPEED) == GPIO_Speed_10MHz) || ((SPEED)
== GPIO_Speed_2MHz) || ((SPEED) == GPIO_Speed_50MHz))

解析：使用宏定义#define将IS_GPIO_SPEED(SPEED)替换为GPIO_Speed_10MHz或者GPIO_Speed_2MHz或者GPIO_Speed_50MHz。

4.9 #define

用法二：带参数的宏定义

STM32标准外设库
stm32f10x.h头文件中的代码：

```
#define
IS_FUNCTIONAL_STATE(STATE)
(((STATE) == DISABLE) ||
((STATE) == ENABLE))
```

解析：该函数为外设时钟使能函数，第一个参数为要使能的外设，第二个参数为是否使能。

在stm32f10x_rcc.c源文件中函数
RCC_APB2PeriphClockCmd()使用了
IS_FUNCTIONAL_STATE(STATE)这个宏
RCC_APB2PeriphClockCm()源码如下：

```
void RCC_APB2PeriphClockCmd(uint32_t
RCC_APB2Periph, FunctionalState NewState)
{
        /* Check the parameters */

assert_param(IS_RCC_APB2_PERIPH(RCC_APB2Periph));
 assert_param(IS_FUNCTIONAL_STATE(NewState));
 if (NewState != DISABLE)
   {
    RCC->APB2ENR |= RCC_APB2Periph;
   }
 else
   {
    RCC->APB2ENR &= ~RCC_APB2Periph;
   }
}
```

4.10 条件编译

只有**满足一定条件**才进行编译，一般用在**头文件**或**文件开头**部分

嵌入式C语言常使用条件编译，通过条件判断来确定是否对某段源程序进行编译。

条件编译的用途：可以用**源程序产生不同版本**。

嵌入式C语言常用的条件编译命令

条件编译	说明
#define	宏定义
#undef	撤销已定义的宏名
#if	条件编译命令，如果# if后面的表达式为true，则执行语句
#ifdef	判断某个宏是否被定义，若被定义，则执行语句
#ifndef	判断某个宏是否未被定义，若未被定义，则执行语句，与#ifdef相反
#elif	#else指令用于#if指令之后，当#if指令的条件不为真时，就编译#else后面的代码，elif相当于else if
#endif	条件编译的结束命令，用在#if、#ifdef、#ifndef之后

4.10 条件编译

形式一：

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

功能：当指定的标识符已被#define定义过，则只编译程序段1，否则编译程序段2。

```
#ifdef IN_XXX
  #define XXX_EXT
#else
  #define XXX_EXT extern
#endif
…….
XXX_EXT volatile u16 Name;
```

如果定义了IN_XXX，则定义XXX_EXT，否则定义XXX_EXT为extern。

4.10 条件编译

形式二：

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

功能：当指定的标识符没有被#define定义过，则编译程序段1，否则编译程序段2。

标准外设库v3.5.0版本中的stm32f10x_rcc.h头文件中的源码

```
#ifndef STM32F10X_CL
  #define RCC_USBCLKSource_PLLCLK_1Div5  ((uint8_t)0x00)
  #define RCC_USBCLKSource_PLLCLK_Div1  ((uint8_t)0x01)
  #define IS_RCC_USBCLK_SOURCE(SOURCE) (((SOURCE)
  == RCC_USBCLKSource_PLLCLK_1Div5) ||\
  ((SOURCE) == RCC_USBCLKSource_PLLCLK_Div1))
#else
  #define RCC_OTGFSCLKSource_PLLVCO_Div3
((uint8_t)0x00)
  #define RCC_OTGFSCLKSource_PLLVCO_Div2
((uint8_t)0x01)
  #define IS_RCC_OTGFSCLK_SOURCE(SOURCE)
(((SOURCE) == RCC_OTGFSCLKSource_PLLVCO_Div3)||\
  ((SOURCE) == RCC_OTGFSCLKSource_PLLVCO_Div2))
#endif
```

4.10 条件编译

形式三：

```
#ifdef 标识符1（或表达式1）
    程序段1
#elif 标识符2（或表达式2）
    程序段2
#endif
```

功能：当定义了标识符1或表达式1，则编译程序段1；否则，如果定义了标识符2或表达式2，则编译程序段2。

标准外设库v3.5.0版本中的stm32f10x_rcc.h头文件中的源码。

```
#ifdef STM32F10X_CL
  /* PREDIV1 clock source (for STM32 connectivity line devices) */
  #define  RCC_PREDIV1_Source_HSE   ((uint32_t)0x00000000)
  #define  RCC_PREDIV1_Source_PLL2  ((uint32_t)0x00010000)
  #define IS_RCC_PREDIV1_SOURCE(SOURCE)
  (((SOURCE)==RCC_PREDIV1_Source_HSE) ||\
       ((SOURCE) == RCC_PREDIV1_Source_PLL2))
#elif defined (STM32F10X_LD_VL) || defined (STM32F10X_MD_VL)
||\
  defined (STM32F10X_HD_VL)
  #define  RCC_PREDIV1_Source_HSE   ((uint32_t)0x00000000)
  #define IS_RCC_PREDIV1_SOURCE(SOURCE) (((SOURCE) ==
  RCC_PREDIV1_Source_HSE))
#endif
```

4.10 条件编译

练习：假设有一个加法函数dataplus，有V1和V2两个版本，在V1版本中其所有加法函数均对8位数据进行加和，并返回16位结果，V2版本中则改为对16位数据进行加和，并返回32位结果。给出加法函数的实现代码，并利用条件编译来方便版本的切换。

```
#define V1 // or V2

#ifdef V1
int16_t dataplus (int8_t d1, int8_t d2)
{
    return d1+d2;
}
#elif defined V2
int32_t dataplus (int16_t d1, int16_t d2)
{
    return d1+d2;
}
#endif
```

```
#define V1 // or V2

#ifdef V1
#define plusdef  int16_t dataplus (int8_t d1, int8_t d2)
#elif defined V2
#define plusdef  int32_t dataplus (int16_t d1, int16_t d2)
#endif

plusdef { return d1+d2;}
```

4.11 union共用体

- **共用体**是一种特殊的数据类型，允许在相同的内存位置存储不同的数据类型。

定义格式：

```
union [共用体名]
{
    类型标识符  成员名1;
    类型标识符  成员名2;
        …… …
} 共用体变量;
```

1. 共用体占用的内存应足够存储共用体中**最大**的成员；
2. 访问共用体的成员，使用**成员访问运算符**“.”。

```
union Data
{
  int i;
  float f;
  char  str[20];
};
```

```
int main( )
{
  union Data data;
  data.i = 10;
  data.f = 220.5;
  strcpy( data.str, "C Programming");
  printf( "data.i : %d\n", data.i);
  printf( "data.f : %f\n", data.f);
  printf( "data.str : %s\n", data.str);
  return 0;
}
```

运行结果

```
data.i : 1917853763
data.f : 412236058032779486045275994368.000000
data.str : C Programming
```

4.11 union共用体

练习：利用共用体的方式，实现uint16变量与2个uint8进行转换，其中uint8分别对应uint16的高8位和低8位。

方法1

```
union data16to8
{
   uint16_t d16;
   uint8_t d8[2];
};
```

方法2

```
union d16
{
   uint8_t u8;
   uint8_t d8;
};

union data16to8
{
   uint16_t data;
   union d16 d;
};
```

4.12 指针

- **指针**
  用于存放地址的变量

- **两个要素：**
  - 值：指的是某个对象的位置（即内存地址）；
  - 类型：是指对象所在位置上所存储数据的类型。

指针理解1

◆ 有确定的数据类型

数据类型 *变量名;

如 int *p ;

特殊的：void * 类型代表通用指针

指针理解2

◆取地址运算符&指向一个变量的存储地址；

```
int i;
int *p;
p = &i;
//将int类型的指针p指向变量i的地址
```

4.12 指针

指针理解3

◆取值运算符*访问指针变量所指向地址单元的内容；

◆& 运算符访问变量的地址

```
int i = 0;
int *p;
p = &i;
printf("%d \n", *p);
  //通过*p获取i的值用于输出
*p = 200;  //通过*p修改变量i的值
printf("%d \n", i);
  //输入i 的值，i=200
```

指针理解4

◆指针和数组

在数组中使用指针，可以通过移动指针，对数组中的元素进行搜索。

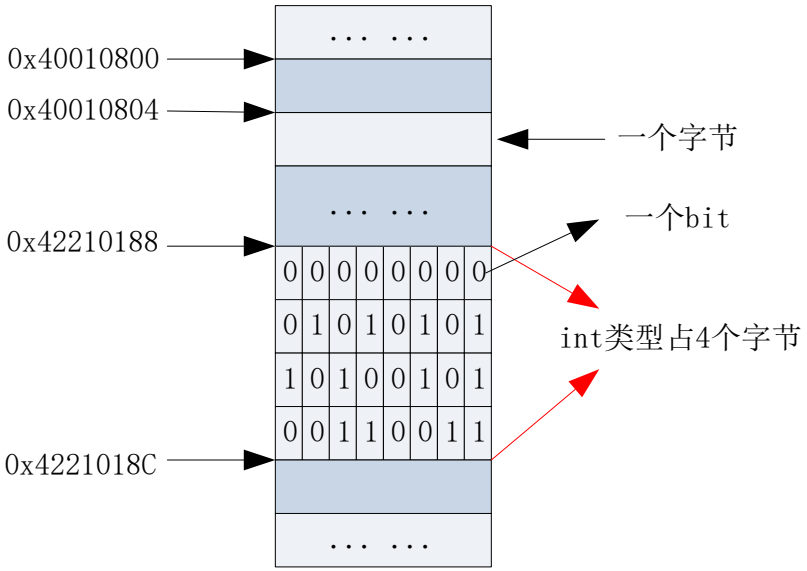
```
int Data_Array[] = {1,2,3,4,5,6};
    //定义一个数组
int *pr; //定义一个指针pr
pr = & Data_Array[0];
      //将指针指向数组的首地址
```

则，可以利用指针对数组的元素进行赋值

```
*pr=8; //用指针给数组的第一个元素赋值
*(pr + 1)=9; //用指针给数组的第二个元素赋值
*(pr + 2)=7; //用指针给数组的第三个元素赋值
```

4.12 指针

32位微处理器中int类型的变量占4个字节的空间。



指针理解5

◆ 将指针从一种类型强制转换成另外一种类型。

pr=（int *）0x42210188;

解析：0x42210188是一个32位的数据，表示的是存储空间中的一个地址，但是在程序中如果写成

pr=0x42210188；//无法确认数据类型

只是将0x42210188这个32位的数据赋给pr，而使用强制类型转换**（int *）**，则是告知编译器这是一个**整型数据所占内存空间的首地址**，如果没有这个类型转换，则编译器会报错。*pr表示取这个指针所指向内存空间中存储的数据，如图，*pr=0x0055。

4.12 指针

函数指针的定义方式为：

函数返回值类型 (*指针变量名) (函数参数列表);

函数地址

编译器会为函数分配一段连续的存储空间，其**首地址**就是这个函数名所定义的变量的地址

函数指针

定义一个指针变量来存放函数地址，这个指针变量就叫作**函数指针变量**，简称函数指针。

用途

在程序中可以通过这个函数指针变量**调用**这个函数。

示例

例如：

```
int f（int x, int *p）;//函数
……
int (* fp) (int, int *); //声明指针fp
fp = f;  //将函数f赋给该指针
……
int y = 1;
int result = fp(3, &y);
    //用指针来调用f函数
```

4.12 指针

STM32标准外设库中，指针的类型为32位整型数据。

STM32中片上外设都是挂接在不同的总线上的，stm32f10x.h文件中利用C语言的宏定义和指针来一步步地封装总线和外设的基地址。

STM32标准外设库中的指针应用

通过宏定义为外设基地址0x40000000取一个宏名PERIPH_BASE，方便使用，stm32f10x.h中FLASH、SRAM和PERIPH（外设）的基地址定义如下：

```
#define FLASH_BASE  ((uint32_t)0x08000000)
#define SRAM_BASE  ((uint32_t)0x20000000)
#define PERIPH_BASE  ((uint32_t)0x40000000)
```

STM32的总线有APB1、APB2和AHB总线组成，各自的总线基地址定义如下：

```
#define APB1PERIPH_BASE  PERIPH_BASE
#define APB2PERIPH_BASE  (PERIPH_BASE + 0x10000)
#define AHBPERIPH_BASE  (PERIPH_BASE + 0x20000)
```

4.12 指针

stm32f10x.h文件中定义了这7组GPIO端口的基地址的宏。

```
#define GPIOA_BASE   (APB2PERIPH_BASE + 0x0800)
#define GPIOB_BASE   (APB2PERIPH_BASE + 0x0C00)
#define GPIOC_BASE   (APB2PERIPH_BASE + 0x1000)
#define GPIOD_BASE   (APB2PERIPH_BASE + 0x1400)
#define GPIOE_BASE  (APB2PERIPH_BASE + 0x1800)
#define GPIOF_BASE   (APB2PERIPH_BASE + 0x1C00)
#define GPIOG_BASE   (APB2PERIPH_BASE + 0x2000)
```

ST公司的工程师采用了C语言结构体的形式封装了这些寄存器组：

```
#define GPIOA  ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB  ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC  ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD  ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE  ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF  ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG  ((GPIO_TypeDef *) GPIOG_BASE)
```

结构体类型后面加上“*”号，表示是**结构体类型的指针**，(GPIO_TypeDef *)则是把GPIOA_BASE等的地址强制转换为GPIO_TypeDef结构体类型的指针，实际编程时可以直接用该宏名访问寄存器。

4.12 指针

GPIO_TypeDef的相关定义：

```
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

声明了名为GPIO_TypeDef的结构体类型，C语言中struct结构体中成员在内存中是按顺序存储的。

有了寄存器的地址，就可以使用指针进行读写操作了。

```
GPIOA->BSRR | = (1 << 10) ;
    //等同于 *((int *) (0x 0x4001 0810) ) | = (1 <<10);
GPIOA->ODR =0xFF;
    //等同于 *((int *) (0x 0x4001 080C) )  = 0xFF;
```

ST工程师把对底层寄存器的操作进行了封装，以API的形式供开发人员调用，如STM32提供的库函数GPIO_SetBits()，用于设置某位引脚为高电平：

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    GPIOx->BSRR = GPIO_Pin;
}
```

库函数GPIO_SetBits()实质上还是对寄存器BSRR进行读写操作，只是进行了封装，开发人员只需调用此函数就可以实现对BSRR寄存器的操作。

思考：为什么不直接把寄存器地址定义为变量？

4.12 指针

练习：在已设计温度传感器结构体的基础上，进一步设计一个温度传感器初始化函数。程序中采用一个全局的温度传感器数组，数组中有128个温度传感器，在主程序中通过for循环调用温度传感器初始化函数，为数组中每个温度传感器进行初始化，即为每个温度传感器按照顺序进行编号，测量间隔时间为1s，并分别配置为打开报警状态，报警阈值为50℃。

```
typedef struct
{
    uint8_t index;
    int16_t value; //当前温度，单位：0.1℃
    uint8_t interval;//测量间隔，单位：1s
    uint8_t isAlarm; //是否打开报警
    int16_t threshold;//报警阈值
    uint8_t alarmState; //报警是否激活
    char comment[20];//备注
} TemperatureSensor_InitTypeDef;
```

4.13 回调函数

- 通过**函数指针调用的函数**。
- 操作系统中的某些函数常需要**调用用户定义的函数**来实现其功能，由于与**常用的用户程序调用系统函数的调用方向相反**，因此将这种调用称为“**回调**”（Callback），而被系统函数调用的函数称为“回调函数”。
- 回调函数是指通过调用其他函数反过来调用某个函数。

STM32的HAL库stm32f1xx_hal_gpio.c源码

GPIO中断处理函数

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
    if (__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)
    {
        __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
        HAL_GPIO_EXTI_Callback(GPIO_Pin);
    }
}

__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    UNUSED(GPIO_Pin);
}
```

调用回调函数

STM32的HAL库中GPIO引脚触发中断后的回调函数

STM32的HAL库的回调函数的函数体需要用户自己编写，其实质是通过中断处理函数调用回调函数来实现中断服务功能

4.14 位运算符

- 位运算符作用于位，并逐位执行操作

“与”运算： &

“或”运算： |

“异或”运算： ^

“取反”运算： ~

真值表

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

```
A       =    0011 1100
B       =    0000 1101
-----------------------------
A&B     =    0000 1100
A|B     =    0011 1101
A^B     =    0011 0001
~A      =    1100 0011
```

- 移位操作运算

左移：<<　（左边的二进制位丢弃，右边补0）

右移：>>　（正数左补 0，负数左补 1，右边丢弃）

A << n

将A按照二进制左移n位，n为非负整数

4.14 位运算符

- 示例：将16位整型uint16_t，转换为2个8位整型uint8_t

```
uint8_t a1, a2; // a1为高8位，a2为低8位
uint16_t b = 0xff1c;

a1 = (b >> 8) & 0xff;
a2 = b & 0xff;
```

- 将2个8位整型uint8_t，组合成1个16位整型uint16_t

```
uint8_t a1 = 0xff;
uint8_t a2 = 0x1c; // a1为高8位，a2为低8位
uint16_t b;

b = (a1 << 8) + a2;  //必须加括号
b = (a1 << 8) | a2;
```

本章小结

4.1 STM32的数据类型

4.2 const 关键字

4.3 static 关键字

4.4 volatile 关键字

4.5 extern 关键字

4.6 struct结构体

4.7 enum

4.8 typedef

4.9 #define

4.10 #ifdef、#if条件编译

4.11 union共用体

4.12 指针

4.13 回调函数

4.14 位运算符