第3章 嵌入式程序设计中的C语言

本章并不详细讲解 C 语言的语法规则，关于 C 语言程序设计有很多优秀的教材和参考资料。这里只针对嵌入式开发中入门者常见的一些问题进行必要的说明和讲解。读者必须有 C 语言程序设计的基础。

3.1 整型


教学视频

3.1.1 整型的位宽

ANSI C 中规定的基本数据类型共有 6 种，分别是 short、int、long、char、float 和 double，其中，float 和 double 为浮点型，short、int、long 为整型，char 为字符型。将一个字符赋值给 char 类型变量时，就是将字符的 ASCII 码，也就是将一个整型数值赋值给变量，所以 char 类型也是一种整型数据类型。

不同数据类型分配的字节数不同，char 为 1 字节，short 为 2 字节，long 为 4 字节。int 类型的长度与机器字长相同，16 位系统中 int 占 2 字节，而 32 位系统中 int 占 4 字节。对于 32 位的单片机来说，int 占 4 字节。如果不确定，则可以用“sizeof(int)”语句测试 int 数据类型的字节数。

整型变量进一步分为无符号整型和有符号整型。定义变量时在数据类型前面添加关键字 unsigned 说明定义无符号变量，signed 关键字说明定义有符号变量。定义 short、int 和 long 类型变量时，在不加说明的情况下，默认定义的是有符号变量。而 char 类型比较复杂，与具体的编译器有关。例如，Windows 环境下的 Keil 5 开发软件中提供了相关的配置项，如图 3.1 所示。若选中该配置项，那么默认情况下所定义的 char 变量为 signed，即有符号整型变量；如果没有选中该配置项，那么在不加说明的情况下所定义的 char 变量为 unsigned 整型变量。

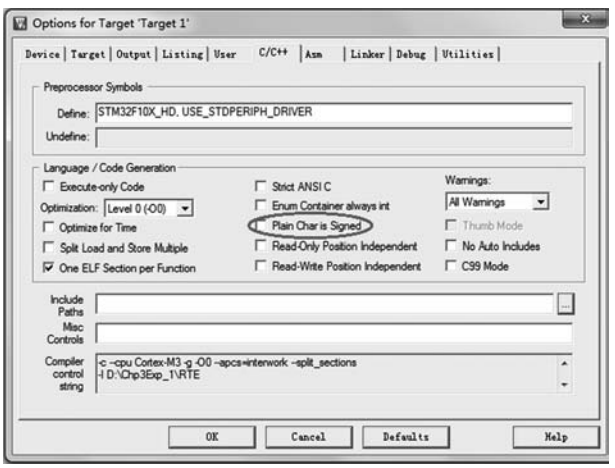

图 3.1 Keil 5 中 char 类型的配置项

定义变量时，数据类型的字节数以及 signed/unsigned 属性一起，决定了变量的数据

范围。例如，unsigned char 为无符号 8 位整型，它的数据范围为 0～255，而 signed char 为有符号 8 位整型，它的数据范围为－128～＋127。对于有符号整型变量来说，变量中存放的是数值的补码。赋值时如果数值超出了变量的数据范围，编译器会给出警告或错误提示信息。

例 3.1：赋值超过变量的数据范围时，编译会产生警告信息。

```
#include  "stm32f10x.h"
signed char  var;
int  main(void)
{  var = 128;
   while(1)
   {  var++;
   };
}
```

上面的代码定义了全局变量 var，赋值 128 时编译时给出了如图 3.2 所示的警告信息。

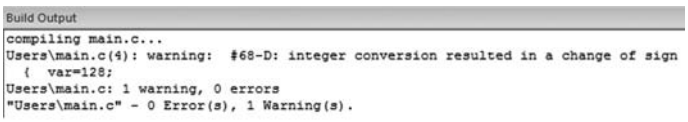

图 3.2　赋值超过数据范围导致的警告信息

数值 128 对应的 8 位二进制数为 10000000，而 var 变量为 signed char 类型，保存的是数值的补码。即最高位为符号位，符号位为 1 表明是负数，为 0 是正数。按照补码规则，二进制编码 10000000 表示－128，而不是＋128，所以编译器发出警告信息，提醒“导致符号改变”。单步调试程序时在 Watch 窗口中可以观察到 var 变量的值，如图 3.3 所示。

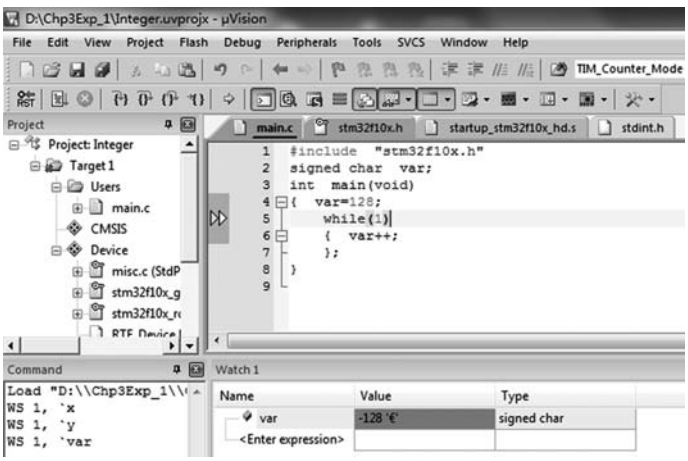

图 3.3　超界时变量中真实的数值

当赋值超过变量存储范围时，编译器会给出警告信息，编程人员应该仔细阅读警告提示信息，这样在软件开发阶段就能及时发现并改正问题。

如果经过算术运算，运算结果超出变量的数据范围，会发生什么情况呢？首先，在编译阶段不会有任何警告或错误提示。其次在程序运行阶段，也不一定会出错。只有当运算结果超出变量的数据范围时才会产生问题，此时变量中保存的结果是错误的，程序继续以错误的数据运行，极有可能导致严重后果，这时才会发现程序有问题。这种只有在运行时才有一定概率发生的错误，危害很大，想要调试定位这样的错误，难度是非常大的。因此定义变量时一定要注意变量的数据范围，根据变量的最大取值范围，定义合适的数据类型。

例 3.2：运算结果超出变量的数据范围，会得到错误的运算结果，如图 3.4 所示。

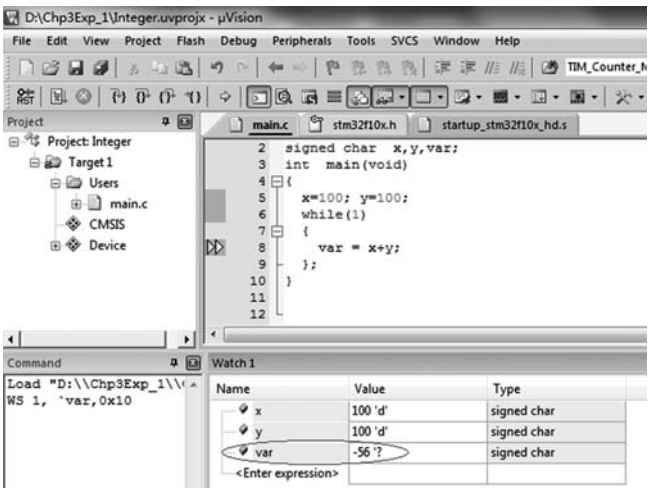

图 3.4　运算结果超出范围

例题中变量 x、y、var 都是 signed char 类型，取值范围为－128～＋127，然而运算结果为 200，超过了 var 变量可以保存的数据范围，因此 var 变量保存的运算结果是错误的。

为什么结果是－56 呢？signed char 类型为 1 字节的有符号整型，x、y 赋值 100，意味着 x、y 变量的 8 位二进制编码为 01100100，两者相加，结果为 11001000，所以 var 变量中保存的二进制编码为 11001000，按照补码规则，对应的数值就是－56。

只要将 var 变量定义为 signed short 类型，就能解决这个问题。

定义变量时一定要明确说明 unsigned 或 signed 属性，不要依赖于默认设置，这样可以增加代码的可移植性和可读性。应注意变量的数据范围，根据可能存储的最大数值来决定变量的数据类型，避免程序执行时发生运算结果超界的情况。

整型数据类型以及对应的数据范围见表 3.1。

表 3.1　数据类型及其数据范围

数据类型	字节数	数据范围	说　明
unsigned char	1	0～255，即 $0\sim2^8-1$	
signed char	1	－128～＋127，即 $-2^7\sim+(2^7-1)$	

续表

数据类型	字节数	数据范围	说　明
unsigned short	2	0～65 535，即 $0\sim2^{16}-1$	
signed short	2	$-32\,768\sim+32\,767$，即 $-2^{15}\sim+(2^{15}-1)$	
unsigned int	4	0～4 294 967 295，即 $0\sim2^{32}-1$	32 位系统
signed int	4	$-2\,147\,483\,648\sim+2\,147\,483\,647$，即 $-2^{31}\sim+(2^{31}-1)$	32 位系统
unsigned long	4	0～4 294 967 295，即 $0\sim2^{32}-1$	
signed long	4	$-2\,147\,483\,648\sim+2\,147\,483\,647$，即 $-2^{31}\sim+(2^{31}-1)$	

例 3.3：signed 类型变量保存的是数值的补码，如图 3.5 所示。

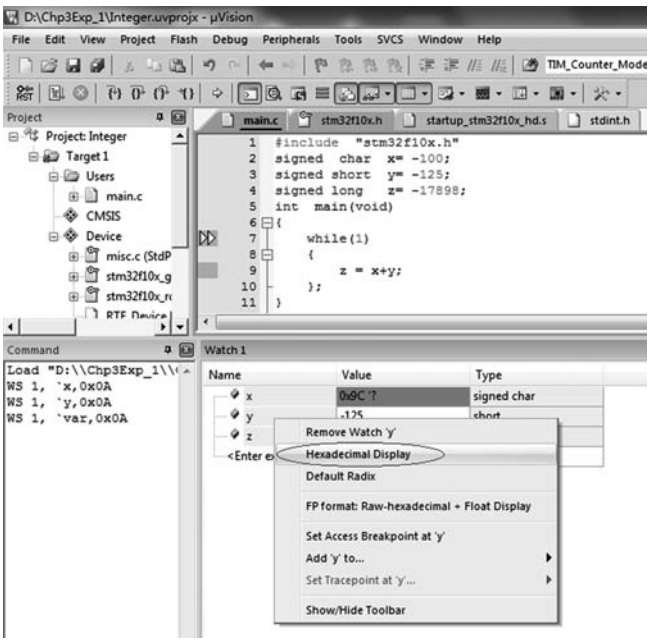

图 3.5　例 3.3 程序截图

单步调试程序时，在 Watch 窗口中可以修改变量的进制，右击，在弹出的快捷菜单中选择 Hexadecimal Display 命令，即“十六进制显示”，就能看到变量实际存储的十六进制数值。取消这个设置，看到的就是对应的十进制数值。

C 语言用 0x 前缀说明十六进制。每 4 位二进制对应转换 1 位十六进制，二进制与十六进制之间的换算非常方便，而二进制数据位数过多，查看和录入时容易出错，所以在调试程序时通常使用十六进制，而不直接用二进制。

例题中，x 变量赋值为－100，而在 Watch 窗口中观察到，所存储的十六进制数值为 0x9C，即二进制编码为 10011100，这就是－100 的补码。

例 3.4：运算结果超出变量的数据范围时，进位丢失了，如图 3.6 所示。

程序执行后，在 Watch 窗口中观察到变量 x 的数值为 44。

x 是无符号 8 位整型，加法运算后，运算结果大于 255，运算向前产生了进位，但变量

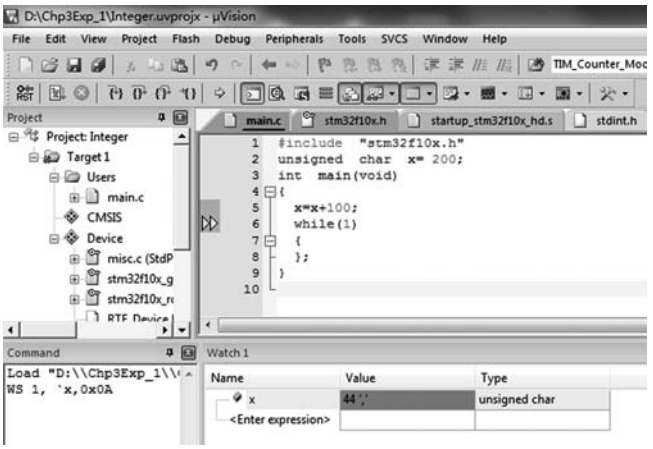

图 3.6　例 3.4 程序截图

x 是 8 位的整型变量，只能保存 8 位的运算结果，因此加法指令执行后，x 变量保存的值为 44。

例 3.5：两个正数相加，运算结果却变成负数，如图 3.7 所示。

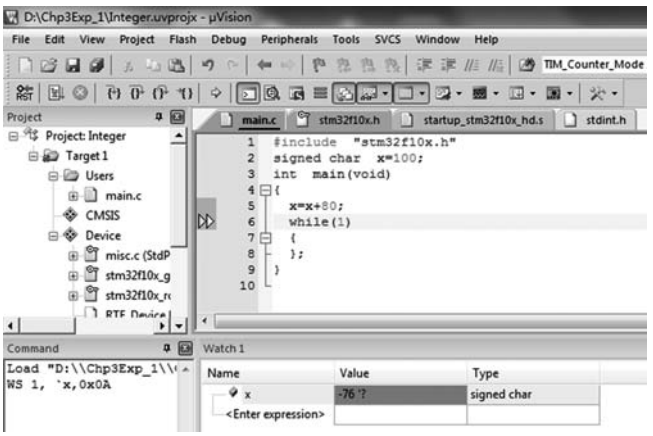

图 3.7　例 3.5 程序截图

signed char 类型的 x 变量初始化为数值 100，即变量 x 的二进制数值为 01100100。加 80，数值 80 对应的 8 位二进制数值为 01010000。两个 8 位二进制数相加，结果存回变量 x。相加后，变量 x 的二进制数值变为 10110100。作为有符号数变量，它的最高位为符号位，符号位为 1，说明是负数，按照补码的运算法则，可以得到变量 x 的十进制数值为－76。


教学视频

3.1.2　访问硬件模块的寄存器

单片机芯片中集成有多个硬件模块，每个片上硬件模块都提供有多个寄存器，通过对这些硬件模块内的寄存器进行读写操作，就可以设定硬件模块的工作方式，控制它们

实现指定的功能，因此在嵌入式系统开发中，必须要访问片上硬件内部的寄存器。

嵌入式系统开发中会定义整型变量来访问硬件模块内部的寄存器，整型变量必须声明为 unsigned，并且变量的位宽必须与寄存器的位宽一致。例如，STM32F103 单片机中 GPIO 模块的配置寄存器 CRL，这个寄存器是 32 位的，所以必须定义 32 位的无符号整型变量，才能正确地访问这个寄存器。

由于定义整型变量访问寄存器时整型变量的位宽必须与寄存器的位宽一致，为此，在 Keil 5 软件提供的系统头文件 stdint.h 中，用 typedef 关键字重新命名了整型数据类型，如图 3.8 所示。重新命名后的数据类型可以直接看出数据类型的位宽，以及是否为有符号数。

```
   /* exact-width signed integer types */
typedef   signed          char int8_t;
typedef   signed short     int int16_t;
typedef   signed           int int32_t;
typedef   signed       __INT64 int64_t;

   /* exact-width unsigned integer types */
typedef unsigned          char uint8_t;
typedef unsigned short     int uint16_t;
typedef unsigned           int uint32_t;
typedef unsigned       __INT64 uint64_t;
```

图 3.8　重命名的整型数据类型

库函数中定义整型变量时不再直接使用 ANSI C 中的关键字来定义整型，而是用重命名后的 uint8_t、uint16_t、uint32_t 和 uint64_t 来定义不同位宽的无符号整型变量，用 int8_t、int16_t、int32_t 和 int64_t 定义有符号整型变量。

作为嵌入式系统的开发人员，定义整型变量时也应该采用重新命名后的数据类型，与库函数保持一致，又能直接看出整型的位宽。由于必须在当前 C 语言源文件中包含 typedef 相关语句后，才能使用重命名后的数据类型，因此 C 语言源程序中必须先用 #include 语句包含 stdint.h 头文件后，才能使用重命名后的类型来定义整型变量。

例 3.6：定义 int8_t 类型的变量，如图 3.9 所示。

```
1  int8_t  x=100;
2  int  main(void)
3  {
4     x=x+80;
5     while(1)
6     {
7     };
8  }
9
```

```
Build Output
Build target 'Target 1'
compiling main.c...
Users\main.c(1): error:  #20: identifier "int8_t" is undefined
  int8_t  x=100;
Users\main.c: 0 warnings, 1 error
".\Objects\Integer.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:00
```

图 3.9　例 3.6 截图

由于没有包含 stdint.h 头文件，编译器给出了错误信息，提示“int8_t 标识符未定义”。必须先用 typedef 语句完成对整型数据类型的重命名后，才能使用重命名后的数据类型。上述程序只需要在开始处添加“#include "stdint.h""语句即可。

3.2　volatile 关键字

3.2.1　C 语言编译器的优化功能

C 语言的编译器会对代码进行优化，删除冗余代码。如果编译器认为对某些变量的


教学视频

读写操作是无用的，那么编译时就会删除相关的读写语句。如果编译器认为某些变量在程序中没有使用，那么编译时甚至可能不定义这些变量。

例 3.7：编译器对代码的优化作用，如图 3.10 所示。

```
1  #include  "stdint.h"
2  int  main( void )
3  {   uint8_t  x, y=35, z=10;
4  x=y;
5  x=z;
6  x=z+20;
7  while(1);
8  }
9
10
Build Output
Build target 'Target 1'
compiling main.c...
Users\main.c(3): warning:  #550-D: variable "x" was set but never used
  {   uint8_t  x, y=35, z=10;
Users\main.c: 1 warning, 0 errors
linking...
Program Size: Code=652 RO-data=320 RW-data=0 ZI-data=1632
".\Objects\Integer.axf" - 0 Error(s), 1 Warning(s).
Build Time Elapsed:  00:00:00
```

图 3.10　例 3.7 截图

编译 main.c 源文件时，编译器给出了警告信息“变量 x 设置了值，但从未使用过”。虽然有警告信息，但程序编译连接成功，可以执行。但是单步调试程序时发现在 Watch 窗口中无法观察到变量 x、y、z 的值，如图 3.11 所示。

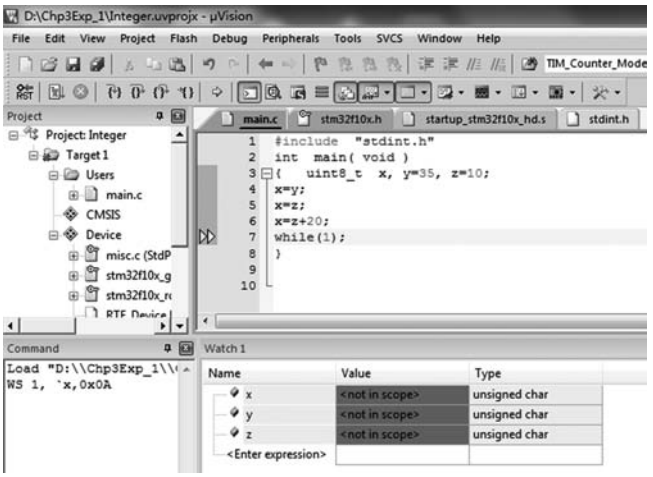

图 3.11　编译器优化的结果

这个现象就是编译器对程序进行优化而导致的。由于变量 x 赋值后从未使用，而变量 y 和 z 只用于给变量 x 赋值，所以编译器优化后，x、y 和 z 变量都不存在了，而读写这些变量的代码，作为冗余代码，也被编译器删除了。虽然看上去执行了 x＝y 算赋值指令，但是打开 Disassembly 窗口，观察实际执行的机器码指令，就会发现定义变量，以及对 x 变量的 3 条赋值语句，实际上都是 NOP，也就是空操作指令，如图 3.12 所示。

CPU 执行的是机器码指令，通过反汇编窗口可以查看 CPU 真正执行的指令。所谓的“反汇编”，就是将机器码指令“反过来”翻译成汇编指令。

图 3.12 中箭头指向的是待执行的机器码指令，该机器码指令是 C 语言指令“x＝y；”的编译结果。0x080003B0 是机器码指令存放的地址，BF00 是十六进制的机器码，该条机器码指令占 2 字节，而 NOP 则是机器码指令反汇编后的汇编指令。

```
Disassembly
0x080003AA E000      DCW      0xE000
     3: {   uint8_t  x, y=35, z=10;
0x080003AC BF00      NOP
0x080003AE BF00      NOP
     4: x=y;
0x080003B0 BF00      NOP
     5: x=z;
0x080003B2 BF00      NOP
     6: x=z+20;
0x080003B4 BF00      NOP
     7: while(1);
0x080003B6 BF00      NOP
0x080003B8 E7FE      B        0x080003B8
```

图 3.12　例 3.7 程序的机器码截图

从图 3.12 显示的反汇编代码可以看出，无论是定义变量，还是赋值语句，对应的汇编指令都是 NOP 空指令，这就是编译器对代码进行优化的结果，编译器认为对变量的读写操作是无用的，将冗余代码全部都删除了。只有最后一条“while(1);”指令是有效的指令，编译成汇编指令 B 0x080003B8。这是一条跳转指令，跳转到地址 0x080003B8，即跳转到自身，实现了“死循环”。

3.2.2　用 volatile 关键字避免优化

定义变量访问硬件模块内的寄存器时，需要严格执行每次对变量的读操作或写操作，而编译器的优化功能可能导致对寄存器的读写操作失败。如果对变量 dr 的赋值操作意味着通过串口发送相应的数据，那么对变量 dr 的每一次赋值，都意味着一次数据传输，编译器不应该对变量 dr 的赋值语句进行优化。

定义变量时添加 volatile 关键字，说明该变量的值会被某些编译器未知的因素改变，如操作系统、硬件模块等，编译器对访问该变量的代码就不再进行优化了。

例 3.8：volatile 关键字的作用。

如图 3.13 所示，例 3.7 中的程序只需要在定义变量时添加 volatile 关键字加以声明，编译时就不再有警告信息，而且反汇编程序也不再是 NOP，变量定义以及变量的赋值都真实执行了。

```
1  #include  "stdint.h"
2  int  main( void )
3  {
4    volatile uint8_t  x, y=35, z=10;
5    x=y;
6    x=z;
7    x=z+20;
8    while(1);
9  }
```

(a) C语言源程序

```
     3: {
0x080003AC B50E      PUSH     {r1-r3,lr}
     4:         volatile uint8_t  x, y=35, z=10;
0x080003AE 2023      MOVS     r0,#0x23
0x080003B0 9001      STR      r0,[sp,#0x04]
0x080003B2 200A      MOVS     r0,#0x0A
0x080003B4 9000      STR      r0,[sp,#0x00]
     5:         x=y;
0x080003B6 F89D0004  LDRB     r0,[sp,#0x04]
0x080003BA 9002      STR      r0,[sp,#0x08]
     6:         x=z;
0x080003BC F89D0000  LDRB     r0,[sp,#0x00]
0x080003C0 9002      STR      r0,[sp,#0x08]
     7:         x=z+20;
0x080003C2 F89D0000  LDRB     r0,[sp,#0x00]
0x080003C6 3014      ADDS     r0,r0,#0x14
0x080003C8 B2C0      UXTB     r0,r0
0x080003CA 9002      STR      r0,[sp,#0x08]
     8:         while(1);
0x080003CC BF00      NOP
0x080003CE E7FE      B        0x080003CE
```

(b) 反汇编程序

图 3.13　例 3.8 截图

用 volatile 关键字定义的变量，编译时不再对该变量的访问进行优化，会严格按照指令执行每一次对该变量的读写操作。因此定义变量访问硬件模块的寄存器时，一定要用

volatile 关键字加以修饰。Keil 5 软件在 core_cm3.h 头文件中定义了相关的宏，定义变量访问片上硬件模块的寄存器时都使用了 volatile 关键字，如图 3.14 所示。

```
#ifdef __cplusplus
  #define   __I     volatile
#else
  #define   __I     volatile const
#endif
#define     __O     volatile
#define     __IO    volatile
```

(a) core_cm3.h中volatile相关宏定义

```
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;
```

(b) stm32f10x.h中GPIO模块相关定义

图 3.14　Keil 5 中访问片上模块寄存器的相关定义

CMSIS 固件库头文件 stm32f10x.h 中为所有片上硬件模块定义了结构体数据类型，每一个结构体成员对应硬件模块中的一个寄存器。例如，GPIO 模块内有 CRL、CRH 等多个 32 位的寄存器，为访问这些寄存器，在 GPIO_TypeDef 结构体中为每个寄存器定义了一个 uint32_t 类型的变量，从图 3.14(a)可知，__IO 实质就是 volatile 关键字。从图 3.14 可以看出，系统定义变量访问片上硬件模块寄存器时全部添加了 volatile 关键字。

除了定义变量访问片上硬件模块寄存器时需要用 volatile 关键字以外，有时定义普通变量时也会添加 volatile 关键字，以避免编译器进行优化。例如，软件延时函数通过一个循环程序消耗 CPU 的执行时间，达到延时的目的。此时就希望严格按照指令顺序，执行一个没有实际作用的程序段。在软件延时函数中定义的变量，也会用 volatile 关键字加以说明，以达到软件延时的目的。软件延时程序如图 3.15 所示。

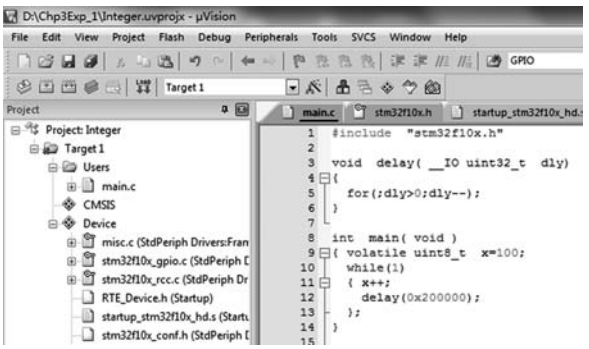

图 3.15　软件延时程序

图 3.15 中的 delay()函数中只有一条 for 循环语句，这个函数不实现任何具体的功能，仅仅是通过执行循环语句，消耗 CPU 的执行时间，达到延时的目的。

3.3　结构体数据类型

3.3.1　struct 关键字

通过关键字 struct，可以将多个不同类型的变量组合起来，作为一个整体，这就是结

构体数据类型。例如“学生”这个对象至少应该有“学号”“姓名”“性别”等属性，其中“学号”可以是32位整型，“姓名”应该是字符型数组，而“性别”可以用单个字符说明，定义为字符型即可。

例3.9：定义结构体数据类型Stu。

程序见图3.16。代码首先包含了两个头文件stdint.h和string.h，其中string.h是C语言标准库提供的头文件，其中包含常用的字符串处理函数的函数声明，代码中调用的字符串复制函数strcpy()就是其中之一。

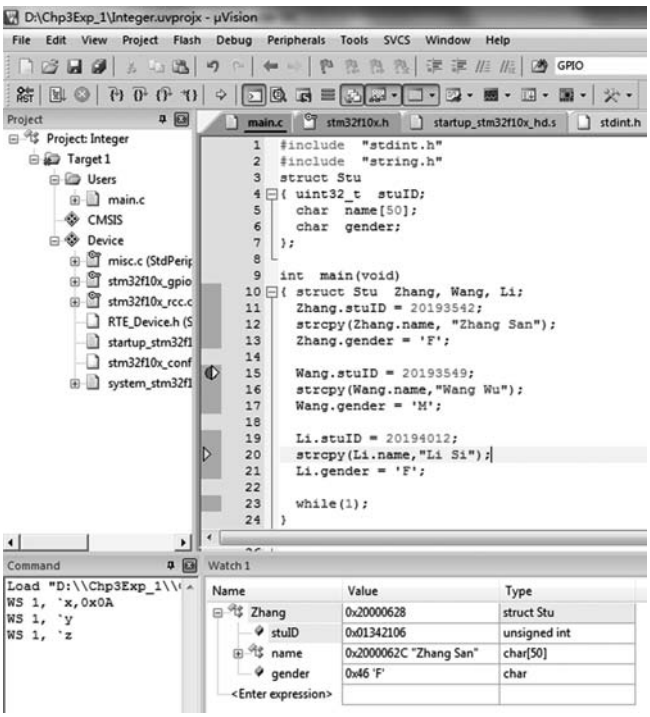

图3.16　例3.9截图

Keil 5支持一些C语言标准库函数，在Keil 5安装文件夹的“..\ARM\ARMCC\include”中，可以看到Keil 5提供的标准库函数的头文件，包括stdint.h、string.h、stdio.h等。

main()函数中定义了3个Stu结构体类型的变量Zhang、Wang和Li，然后对变量的数据成员进行了赋值，name数据成员是char类型的数组，调用strcpy()函数，完成字符串复制。

在Debug调试环境下可以看到，Zhang的类型是struct Stu，作为一个复合型的变量，Value栏目下显示的十六进制数值0x20000628实际上是结构体变量Zhang的地址。单击复合型变量左边的“+”号，可以观察其中的数据成员。

name的十六进制数值0x2000062C是数组首地址，“Zhang San”是具体的字符串，单击左边的“+”号，打开数组，可以看到每个数组成员的具体情况。


教学视频

3.3.2 访问单片机片上外设寄存器

Keil 5 为所支持的单片机提供了相关的头文件，在头文件中为访问单片机片上外设的寄存器定义了相关的结构体数据类型。例如，为 STM32F10X 系列单片机提供了头文件 stm32f10x.h，该头文件中为每一种片上外设定义了一个结构体数据类型，每个数据成员对应片上外设中的一个寄存器。

单片机的参考手册中详细说明了片上外设的工作情况，从中截取了 GPIO 模块寄存器的缩略说明，如图 3.17 所示。

偏移	寄存器	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
000h	GPIOx_CRL	CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]		CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
	复位值	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
004h	GPIOx_CRH	CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]		CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
	复位值	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
008h	GPIOx_IDR	保留																IDR[15:0]															
	复位值																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00Ch	GPIOx_ODR	保留																ODR[15:0]															
	复位值																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
010h	GPIOx_BSRR	BR[15:0]																BSR[15:0]															
	复位值	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
014h	GPIOx_BRR	保留																BR[15:0]															
	复位值																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
018h	GPIOx_LCKR	保留															LCK	LCK[15:0]															
	复位值																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 3.17 GPIO 模块寄存器列表

图 3.17 显示了 GPIO 模块寄存器的相关信息，“偏移”指寄存器相对于模块首地址的偏移地址，汇编语言中用字符 h 说明为十六进制。后面给出了每个寄存器 d0～d31 位的说明。从图 3.17 中可以看出，IDR 寄存器只有低 16 位有效，而高 16 位都是保留位，目前没有任何作用。

stm32f10x.h 头文件中为访问 GPIO 寄存器定义了结构体数据类型 GPIO_TypeDef，如图 3.18 所示。

```
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;
```

图 3.18 GPIO 结构体定义

定义 CRL 变量用于访问 CRL 寄存器，以此类推。结构体中所定义的每一个数据成员对应访问片上外设的一个寄存器，寄存器的位宽就决定了结构体成员的数据类型。

ANSI C 按变量定义的先后顺序为变量分配存储空间，这意味着结构体中定义数据成员的先后顺序就决定了数据成员之间的相对偏移地址，所以为访问片上外设寄存器而定义的结构体，其数据成员的类型以及定义的先后顺序都必须与参考手册中的规定一致。

单片机芯片中集成有多个 GPIO 模块，每个模块的起始地址不同。STM32F103 单片机中最多集成 7 个 GPIO 模块，起始地址见表 3.2。

表 3.2 片上 GPIO 模块地址表

片上 GPIO	起 始 地 址
GPIO 端口 A	0x4001 0800
GPIO 端口 B	0x4001 0C00
GPIO 端口 C	0x4001 1000
GPIO 端口 D	0x4001 1400
GPIO 端口 E	0x4001 1800
GPIO 端口 F	0x4001 2000
GPIO 端口 G	0x4001 2400

stm32f10x.h 中定义了 GPIO_Typedef 结构体数据类型后，用宏定义将指定地址强制转换为 GPIO_Typedef 结构体指针，如图 3.19 所示，通过结构体指针就能够访问片上 GPIO 的寄存器了。

```
#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)
```

图 3.19 片上 GPIO 模块的结构体指针定义

stm32f10x.h 头文件为所有的片上外设都定义了结构体数据类型，声明了基地址，并通过宏定义，将片上外设基地址强制转换为相应的结构体指针，所以 C 语言源程序中只需要包含 stm32f10x.h 头文件，就能直接通过结构体指针读写片上外设的寄存器了（见图 3.20）。Keil 5 开发环境中会自动提示结构体的数据成员，大大方便了开发人员的工作。

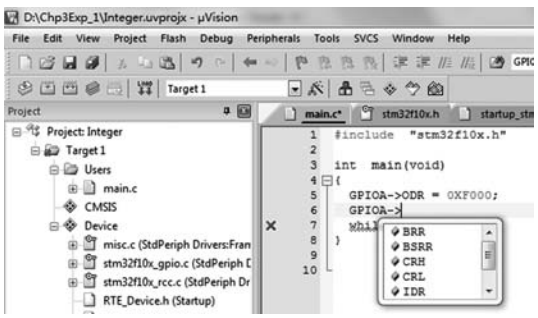

图 3.20 读写片上外设寄存器

3.4 枚举数据类型

生活中常常会遇到只有有限个选项的情况，例如，一天是星期几的问题，只可能是星期一到星期日中的一个，而性别只有两个选项：男或女。

当定义变量时，若希望限定对变量的有效赋值只能是指定选项中的一个，此时就需要定义枚举数据类型。

对于C语言来说，枚举数据类型与结构体数据类型一样，都是“构造类型”。枚举类型将变量的取值用一组常数逐一列出来。枚举类型中的每一个取值只能是整数，默认情况下取值从0开始。

例3.9中定义的结构体Stu中，结构体成员gender说明学生性别，定义为char类型。性别只有男或女，其有效值只有'M'(男)或'F'(女)两种取值。但是作为char类型，其取值范围就远远不止这两种了。

如果将gender改为枚举数据类型，并指定其取值为'M'或'F'，那么代码中对gender变量进行赋值时，就需要用枚举数据类型中定义的选项，否则编译时会产生警告信息。

例3.10：性别只能是男或女。

如图3.21所示，代码中定义了枚举数据类型Gender，并且将结构体Stu中的数据成员gender定义为枚举类型。对gender进行赋值时，就只有Male或Female两种情况。若使用其他赋值，则编译器会给出警告信息。

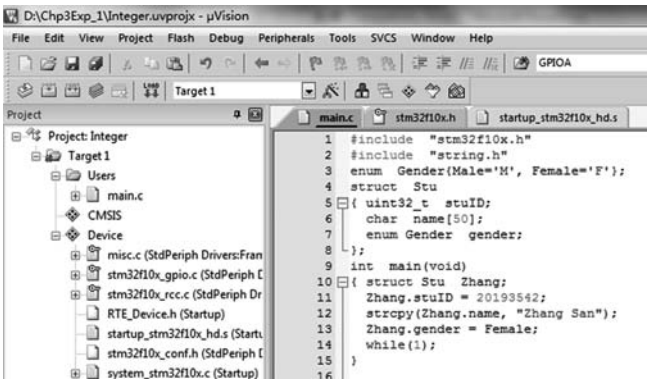

图3.21　例3.10截图

CMSIS固件库提供的头文件中定义了很多枚举数据类型，例如，标志位只有置位或复位两种状态，而片上外设可以使能或禁止，为此stm32f10x.h头文件中分别定义了FlagStatus和FunctionalState枚举类型，CMSIS库函数中使用这些枚举类型作为参数，而不是直接用整型，相关代码片段如图3.22所示。

stm32f10x.h中的枚举数据类型定义

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

GPIO模块的库函数

```
void GPIO_PinRemapConfig(uint32_t GPIO_Remap, FunctionalState NewState);
```

图3.22　stm32f10x.h头文件中定义的枚举数据类型

调用库函数时需要注意参数的类型，枚举类型的参数应该用规定的选项作为实参，而不应该直接用整数数值，以避免编译时产生大量的警告信息。此外，由于枚举类型定

义中选项的名称非常直观，作为参数传递时，可以增加程序的可读性。


教学视频

3.5 static 关键字

static 语义为“静态”。定义变量时添加 static 关键字，就定义了“静态”变量；定义函数时添加它，就定义了“静态”函数。static 关键字的作用各有不同，是非常有用的一个关键字。

3.5.1 静态全局变量

1. 全局变量

全局变量的作用域是整个项目。在一个 C 语言源文件中定义了全局变量，定义之后就可以访问这个变量了。其他 C 语言源文件只需要用 extern 关键字声明一下，就可以访问这个全局变量了。

全局变量为静态分配，编译时就为全局变量分配了存储空间，全局变量的生命周期是整个程序运行期间。只要程序还在运行，全局变量就一直占据着存储空间，直到程序结束运行，才释放存储空间。

由于全局变量的作用域为整个项目，因此不能在多个 C 语言源文件中定义同名的全局变量，否则编译时会提示“重复定义”的错误。

2. 静态全局变量

定义全局变量时前面添加 static 关键字，就定义了静态全局变量。这里 static 关键字限制了全局变量的作用域，将其作用范围局限在当前的 C 语言源文件中。也就是说，静态全局变量只能在定义它的 C 语言源文件中访问，其他 C 语言源文件不能访问它。静态全局变量的生命周期与全局变量一样，都是在整个程序运行期间有效。

由于静态全局变量作用域局限在定义它的文件中，所以其他 C 语言源文件中可以定义同名的静态全局变量或全局变量，但是这是不同的变量。

虽然语法上允许在不同 C 语言源文件中定义同名的静态全局变量，但是这很容易引起逻辑错误，编程过程中很容易弄不清楚当前访问的是哪个静态全局变量。编程中应该避免定义同名的变量，不管是局部变量、全局变量，还是静态全局变量。

例 3.11：全局变量的定义与访问。

如图 3.23 所示，例题项目中有两个源文件：main.c 和 myMath.c。

全局变量 gfPI，在 myMath.c 中定义，在 main.c 中作了 extern 声明，并且在 main() 函数中又定义了同名的局部变量。那么 main()函数的 while(1)循环中计算面积 area 时访问的 gfPI 变量是全局变量，还是局部变量呢？

当同名的局部变量和全局变量都有效时，使用局部变量。所以 while(1)循环中计算

(a) Project窗口　　(b) main.c源文件　　(c) myMath. c源文件

图 3.23　例 3.11 程序截图

area 时访问的是局部变量 gfPI，这导致计算结果不够精确，然而无论怎么修改全局变量 gfPI 都无法解决问题。

在 main. c 和 myMath. c 中都定义了静态全局变量 gcVal，初始化数值分别为 20 和 100。那么 main()函数的 while(1)循环中调用 Inc_gcVal()函数时访问的是哪个 gcVal 变量呢？

由于 Inc_gcVal()函数是在 myMath. c 中定义的，当调用该函数时，程序会跳转到 myMath. c 中的函数体执行，因此这里访问的是在 myMath. c 中定义的静态局部变量 gcVal，而不是在 main. c 中定义的 gcVal。

虽然由于作用域不同，使得语法上允许定义同名的静态全局变量，或与全局变量同名的局部变量，但是这只会造成编程人员的混乱，程序中如果隐藏着这样的问题(bug)，是非常难以调试和定位的，因此不要在程序中定义同名的静态全局变量。

3.5.2　静态局部变量

在函数内部或程序块内定义的变量为局部变量，其作用域局限在函数或程序块内。在程序运行期间，调用函数或执行到程序块时才会为局部变量分配存储空间，创建变量，而函数结束或离开程序块时就会释放存储空间，销毁局部变量。局部变量的生命周期以及作用域都是有限的。

定义局部变量时添加 static 关键字，就定义了静态局部变量。这里 static 关键字改变了局部变量的生命周期，静态局部变量创建后，不再被销毁，直到整个程序结束运行，退出时才会释放存储空间。

静态局部变量的作用域依然有限，但是它的生命周期延长了——从创建开始，持续到程序运行结束。

例 3.12：记录函数被调用的次数。

修改例 3.11 的 Inc_gcVal()函数，定义静态局部变量记录下函数被调用的次数。程序如图 3.24 所示。

只在第一次调用函数时创建并初始化静态局部变量，此后静态局部变量就一直有效，因此 main()函数每调用一次 Inc_gcVal()函数，u16Times 变量的数值就加 1，从而记

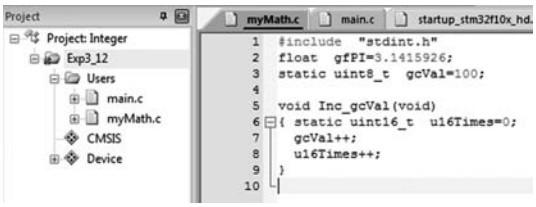

图 3.24　例 3.12 程序截图

录了函数被调用的次数。

3.5.3　静态函数

函数的作用域为整个项目，在一个 C 语言源文件中定义了函数，其他 C 语言源文件中只需要完成函数声明，就可以调用这个函数了。因此，整个项目中不允许定义同名的函数，否则连接时就会产生“重复定义”的错误。

定义函数时添加 static 关键字，就定义了静态函数，此时函数的作用域局限在当前文件中，其他 C 语言源文件不能调用。

由于静态函数的作用域局限在当前文件中，因此其他 C 语言源文件中可以定义同名的函数，但是通常在程序设计中应该尽量避免定义同名的函数，一不小心就容易在程序中埋下隐患。

当程序员定义了一个静态函数时，其实就等同于在宣布“该函数仅供本文件中其他函数调用，其他文件别调用它！”。这样可以为代码提供一定的“保护”，若其他文件试图调用它，编译时就会报错。还增强了代码的可读性，后续其他人员接手这个程序，继续开发时，看到了静态函数，就明白不应该在其他文件中调用它。

STM32F10x 系列单片机的库函数文件 system_stm32f10x.c 中定义了静态函数 SetSysClock()，该函数完成了系统总线时钟初始化，只在 SystemInit()函数中调用它。其他 C 源文件应该调用 SystemInit()函数，而不是 SetSysClock()，因此程序中将 SetSysClock()函数定义为静态函数。

总的来说，只有当确定该函数只在当前文件中被调用，不期望其他源文件调用它时，才将函数定义为静态函数。

3.6　宏定义

#define 宏定义指令实质上是为常数或表达式取了一个别名。宏定义没有数据类型，也不会分配存储单元，编译时直接将代码中的宏用宏定义中的常数或表达式进行替换。

例 3.13：计算圆面积的宏。

如图 3.25 所示，main.c 源文件中首先定义了两个宏，圆周率 PI 为一个常数，而计算

圆面积的带参宏 area(x)是一个表达式。在 main()函数中,两次调用了 area 宏。调试程序时,在 Watch 窗口中观察变量 x 和 y 的数值,可以发现第二次调用时,也就是 area(4+3)处,结果出错。

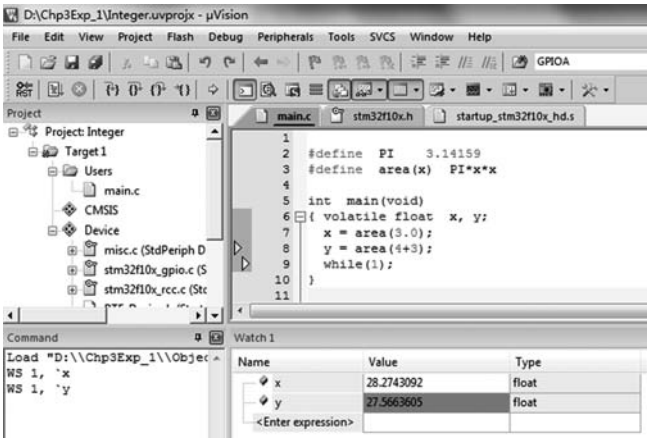

图 3.25　例 3.13 截图

main()函数中调用宏 area(x),编译时直接进行替换。程序中的 area(4+3),4+3 作为 x 进行替换,替换后的表达式是 3.14159×4+3×4+3。这是一个常数运算式,编译时会直接计算出结果,也就是 27.56636,所以编译后,这条赋值语句实际上就是“y=27.56636;”。

由于宏定义的这种特性,定义宏时一定要注意加上括号,设定表达式中各个运算的优先级。上面的代码只需要在定义 area(x)宏时注意加上括号,就能正常计算圆面积了。

```
#define  area(x)  (PI*(x)*(x))
```

如果留意一下 CMSIS 固件库提供的头文件中的宏定义,就会发现头文件中定义宏时都添加了括号,以避免调用宏时产生错误。

结构体指针 GPIOA 宏定义的相关代码如下。

```
#define PERIPH_BASE          ((uint32_t)0x40000000)
#define APB2PERIPH_BASE      (PERIPH_BASE + 0x10000)
#define GPIOA_BASE           (APB2PERIPH_BASE + 0x0800)
#define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
```

片上外设 GPIOA 的起始地址为 0x40010800,GPIOA 外设在 APB2 总线上,而 APB2 总线上有多个片上外设。在 stm32f10x.h 头文件中,首先从外设基地址 PERIPH_BASE 开始定义宏,每个宏定义中都添加了括号,确保多个宏互相调用时不会由于运算优先级而导致出错。

用#define 定义的宏,用#undef 可以撤销,撤销后,就不能再调用这个宏了。

宏定义只在当前的 C 语言源文件中有效,一个项目中若包含多个 C 语言源文件,那

么每个C源文件中都需要用#define语句实现宏定义。但是这样编写代码，一旦宏定义的表达式写错了，解决问题时会非常麻烦，需要逐一修改所有C源文件中相关的宏定义。因此，在项目开发中，通常在头文件中实现宏定义，C源文件中只需要包含头文件，就能够调用头文件中定义的宏了。

3.7 条件编译与头文件

3.7.1 条件编译指令

Keil 5提供了很多头文件，对于开发人员来说，只需要包含头文件，就能调用系统提供的各种接口函数，非常方便。

随意打开几个系统提供的头文件，对比一下就会发现，前几行代码很相似，都是下面这样的格式，其中“XXX”直接用文件名来替换即可。

```
#ifndef  __XXX_H
#define  __XXX_H
…
#endif
```

代码中的#ifndef和#endif就是条件编译指令。条件编译指令属于预处理指令，这些指令不可执行，没有对应的机器码，而是在编译时起作用。编译器根据条件满足与否，决定是否编译相关的代码段。ANSI C中的条件编译指令见表3.3。

表3.3 条件编译指令

指　令	功　能
#if 表达式 #else #endif	#if后面有一个常量表达式。编译时若表达式的值为true，则编译#if分支的代码，跳过#else分支的代码。如果表达式的值为false，则编译时跳过#if分支的代码，改为编译#else与#endif之间的代码 #if命令的功能有些类似于C语言中的if-else-end if指令，可以根据实际情况决定，是否要包含#else分支
#if 表达式1 #elif 表达式2 #else #endif	编译时若表达式1的值为true，则编译#if分支的代码，否则需要判断表达式2，若表达式2的值为true，则编译#elif分支的代码；若表达式1和表达式2的值都为false，则编译#else分支的代码 与if指令相似，#elif可以多重嵌套，形成多种不同的编译情况
#ifdef 符号 #endif	编译时，如果定义了符号，则编译#ifdef与#endif之间的代码，否则就跳过这段代码，不编译
#ifndef 符号 #endif	如果没有定义符号，才编译#ifndef与#endif之间的代码；如果符号已经定义了，则跳过这段代码，不编译

stm32f10x.h头文件中的条件编译指令如下。

```
#ifndef __STM32F10x_H
#define __STM32F10x_H
…
#endif
```

代码首先判断是否定义了符号“__STM32F10X_H”，如果没有定义才编译后面的代码段。如果已经定义了这个符号，编译时就跳过后续的代码段，也就是整个 stm32f10x.h 头文件的所有代码。

如果 C 语言源程序中用 #include 指令包含了两次 stm32f10x.h 头文件，那么编译时，第一次包含头文件时，#ifndef 指令条件满足，此时符号“__STM32F10X_H”未定义，编译后续的代码，立刻就用 #define 语句定义了这个符号；而第二次包含头文件时，#ifndef 指令的条件不满足，后面的代码不会再次被编译。

头文件一定会用条件编译语句将文件中真正的内容包括起来，只有这样才能避免重复包含头文件时产生重复定义或重复声明错误。

程序设计中，在头文件中包含头文件是非常常见的现象，这就可能导致多次包含某个头文件的情况。例如，调用片上外设库函数完成项目开发时，需要用到 RCC 模块和 GPIO 模块，而对应库函数的头文件 stm32f10x_rcc.h 和 stm32f10x_gpio.h 中都包含了 stm32f10x.h 头文件，而 main.c 源文件中需要包含 RCC 模块和 GPIO 模块的头文件，导致 stm32f10x.h 头文件被包含了多次。

用 #include 指令包含头文件，实质就是将头文件的所有内容直接插入 #include 指令所在位置。多次包含意味着某个头文件的内容会被插入多次，这就是编写头文件时，必须用条件编译指令将头文件真正的内容囊括在内的原因。

#ifndef 和 #endif 是一对条件编译指令，#ifndef __xxx_H 判断是否定义了符号 __xxx_H，如果没有定义，那么会编译其中的内容；如果已经定义了这个符号，那么编译器会忽略 #ifndef 和 #endif 之间的代码。


教学视频

3.7.2 头文件

对于在 xxx.c 源文件中编写的函数来说，其他源文件要调用这个函数，就必须先完成函数声明。为了方便其他源文件调用，通常会为 xxx.c 源文件编写一个对应的头文件 xxx.h，将函数声明、全局变量的 extern 声明、宏定义、结构体定义和枚举类型定义等都放在头文件中。这样其他源文件只需要包含这个头文件，就可以调用其中声明的函数、宏等。通常习惯上会将 C 源文件与对应头文件命名为相同的文件名，当然这不是硬性规定。

一般情况下，头文件中只会做函数声明，而不会定义函数，这是因为项目中可能会有多个源文件包含这个头文件，这就意味着在多个源文件中定义了函数，编译单个源文件时不会有问题，但是编译连接整个项目时就会出现函数重复定义的错误。

由于函数和全局变量的作用范围是全局的，在整个项目中都有效，因此通常情况下，头文件中不会有函数定义和全局变量定义。

如果要在头文件中定义函数或全局变量，那么只能在一个源文件中包含这个头文件，或者定义函数或全局变量时用 static 关键字，将函数和全局变量的作用范围限制在当前文件中，但这时在多个源文件中包含这个头文件，就会多次定义这些函数和全局变量，每个源文件中访问的是在自己文件中定义的全局变量。项目开发中后者容易导致错误，并且很难调试定位这样的错误。

例 3.14：系统头文件 core_cm3.h 中的函数定义。

在 CMSIS 固件库中，绝大多数头文件都只进行了函数声明，但是内核相关头文件 core_cm3.h 以及 core_cmFunc.h 有所不同，定义了操作单片机内核的接口函数。函数定义的前面全部添加了“__STATIC_INLINE”，这是在 core_cm3.h 中定义的宏，相关宏定义见图 3.26。

core_cmFunc.h | stm32f10x_gpio.h | main.c | stm32f10x.h | startup_stm32f10x_hd.s | core_cm3.h

```
78
79  #if   defined ( __CC_ARM )
80    #define __ASM            __asm                      /*!< asm keyword for ARM Compiler          */
81    #define __INLINE         __inline                   /*!< inline keyword for ARM Compiler       */
82    #define __STATIC_INLINE  static __inline
83
84  #elif defined ( __GNUC__ )
85    #define __ASM            __asm                      /*!< asm keyword for GNU Compiler          */
86    #define __INLINE         inline                     /*!< inline keyword for GNU Compiler       */
87    #define __STATIC_INLINE  static inline
88
89  #elif defined ( __ICCARM__ )
90    #define __ASM            __asm                      /*!< asm keyword for IAR Compiler          */
91    #define __INLINE         inline                     /*!< inline keyword for IAR Compiler. Only available in High optimization mode! */
92    #define __STATIC_INLINE  static inline
93
94  #elif defined ( __TMS470__ )
95    #define __ASM            __asm                      /*!< asm keyword for TI CCS Compiler       */
96    #define __STATIC_INLINE  static inline
97
98  #elif defined ( __TASKING__ )
99    #define __ASM            __asm                      /*!< asm keyword for TASKING Compiler      */
100   #define __INLINE         inline                     /*!< inline keyword for TASKING Compiler   */
101   #define __STATIC_INLINE  static inline
102
103 #elif defined ( __CSMC__ )
104   #define __packed
105   #define __ASM            _asm                      /*!< asm keyword for COSMIC Compiler      */
106   #define __INLINE         inline                    /*use -pc99 on compile line !< inline keyword for COSMIC Compiler   */
107   #define __STATIC_INLINE  static inline
108
109 #endif
```

图 3.26　core_cm3.h 中的宏定义

core_cm3.h 中通过条件编译指令，针对 Keil 5 所用的 C 编译器，完成了相应的宏定义。默认情况下，Keil 5 使用 ARM 的 C 编译器，也就是图 3.26 中矩形框部分。

core_cm3.h 和 core_cmFunc.h 中都定义了接口函数，所有函数定义都添加了“__STATIC_INLINE”，也就是说，在头文件中定义的所有函数都是静态函数，如图 3.27 所示。

编写头文件时，一定要用条件编译指令，以避免多次包含头文件导致重复定义错误。

总的来说，如果没有特殊要求，不要在头文件中定义函数或全局变量。

通常头文件中只包含有 typedef 定义、宏定义等这些作用范围只局限在当前文件的定义，以及函数声明和全局变量的 extern 说明。

core_cmFunc.h | stm32f10x_gpio.h | main.c | stm32f10x.h | startup_stm32f10x_hd.s | core_cm3.h

```
1299 /** \brief  Set Priority Grouping
1300
1301   The function sets the priority grouping field using the required unlock sequence.
1302   The parameter PriorityGroup is assigned to the field SCB->AIRCR [10:8] PRIGROUP field.
1303   Only values from 0..7 are used.
1304   In case of a conflict between priority grouping and available
1305   priority bits (__NVIC_PRIO_BITS), the smallest possible priority group is set.
1306
1307     \param [in]      PriorityGroup  Priority grouping field.
1308  */
1309 __STATIC_INLINE void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)
1310 {
1311   uint32_t reg_value;
1312   uint32_t PriorityGroupTmp = (PriorityGroup & (uint32_t)0x07);     /* only values 0..7 are used          */
1313
1314   reg_value  =  SCB->AIRCR;                                        /* read old register configuration    */
1315   reg_value &= ~(SCB_AIRCR_VECTKEY_Msk | SCB_AIRCR_PRIGROUP_Msk); /* clear bits to change               */
1316   reg_value  =  (reg_value                                   |
1317                 ((uint32_t)0x5FA << SCB_AIRCR_VECTKEY_Pos) |
1318                 (PriorityGroupTmp << 8));                          /* Insert write key and priorty group */
1319   SCB->AIRCR =  reg_value;
1320 }
1321
```

图 3.27　core_cm3.h 中的函数定义

教学视频

3.8　变量在哪里

3.8.1　堆、栈和静态区

C 语言中将内存分为 3 部分：堆(heap)、栈(stack)和静态区(static area)，栈也常被称为“堆栈”。

堆栈区为按照“先进后出”原则进行操作的连续存储空间。将数据存入堆栈称为“压栈”操作，而将数据弹出堆栈称为“弹栈”操作。对于堆栈的操作，始终都在堆栈栈顶进行，堆栈栈顶位置随着压弹栈操作而上下浮动。可以将堆栈区想象为一个弹夹，从一个口将子弹压进或弹出，所以最后压入弹夹的子弹，最先被弹出。

堆栈区用于保存局部变量，进入函数时在堆栈区为函数内的局部变量分配存储空间，函数执行结束时弹栈恢复堆栈空间。堆栈的执行效率高，但是堆栈区的大小是有限的。

堆区用于动态分配内存，即调用 malloc()或 new()函数分配的内存。对于动态分配的内存，必须调用 free()函数或 delete()函数来释放，在主动释放之前内存始终被占用。动态分配内存比较灵活，但如果忘记释放内存，则会造成内存“泄漏”。

静态区用于保存全局变量和静态变量，包括静态局部变量和静态全局变量。在静态区分配存储空间的变量，其生命周期为整个程序运行期间。

静态区的存储空间分配由编译器在编译时分配，而堆区和栈区的存储空间分配是在程序运行过程中进行的。栈区空间是有限的，如果分配的存储空间超过栈区剩余空间大小，则会导致堆栈“溢出”，程序崩溃。

3.8.2　单片机中变量的存储空间分配

单片机芯片内集成有 Flash 存储器和 SRAM 存储器，两者的容量都有限。STM32F10x

系列单片机采用哈佛架构，存储器以及片上外设都在一个 4GB 的地址空间中。其中 Flash 地址从 0x0800000 开始，SRAM 地址从 0x20000000 开始，片上外设地址从 0x40000000 开始。

堆区、栈区以及静态区都在 SRAM 存储器中。在系统提供的启动文件中可以设定堆区、栈区的大小。

例 3.15：大容量设备启动文件 startup_stm32f10x_hd.s 中定义堆区和栈区。

启动文件为汇编语言源程序。根据单片机芯片中集成的存储器容量大小，芯片分为小容量、中容量和大容量，启动文件也有所不同。stm32f103vet6 芯片内部集成有 512KB 的 Flash 和 64KB 的 SRAM，属于大容量芯片，启动文件为 startup_stm32f10x_hd.s。

从图 3.28 可以看出，启动文件中规定了栈区大小为 0x00000400，而堆区大小为 0x00000200，两者都在 SRAM 中。根据项目需求，可以自行调整栈区大小。

startup_stm32f10x_hd.s

```
34
35  Stack_Size      EQU     0x00000400
36
37                  AREA    STACK, NOINIT, READWRITE, ALIGN=3
38  Stack_Mem       SPACE   Stack_Size
39  __initial_sp
40
41  ; <h> Heap Configuration
42  ;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
43  ; </h>
44
45  Heap_Size       EQU     0x00000200
46
47                  AREA    HEAP, NOINIT, READWRITE, ALIGN=3
48  __heap_base
49  Heap_Mem        SPACE   Heap_Size
50  __heap_limit
51
```

图 3.28　例 3.15 截图

例 3.16：调试环境下确定栈区在 SRAM 存储器中的位置。

STM32F10x 系列单片机 SRAM 存储区地址从 0x20000000 开始，根据芯片具体型号可以确定片上集成的 SRAM 存储器大小。

启动文件中定义了栈区和堆区的大小，若不修改启动文件，则默认情况下栈区大小为 0x00000400，而堆区大小为 0x00000200。启动后，CPU 内的 SP 寄存器指向堆栈栈顶，压栈时 SP 指针减小，而弹栈时 SP 指针增大，随着压栈弹栈操作，SP 始终指向堆栈栈顶。只要观察 SP 寄存器，就能了解当前堆栈的使用情况，如图 3.29 所示。

主程序中没有定义任何全局变量或局部变量，单步调试程序，进入 main()函数时，观察 CPU 内部寄存器的情况，可以看到 R13(SP)寄存器的值为 0x20000660，目前堆栈区中没有压入任何数据，SP 指向堆栈区的尾部，而堆栈区大小为 0x00000400，也就是说，地址 0x20000260～0x2000065F 这 0x00000400 个字节单元为堆栈区。

当修改启动文件，将堆栈区大小改为 0x00001000 时，编译连接后，再次单步调试，可以观察到 SP 寄存器的情况如图 3.30 所示。

将堆栈区大小改为 0x00001000 后，从图 3.30 可以看出，堆栈尾部地址变为 0x20001260，此时堆栈区地址为 0x20000260～0x200125F。

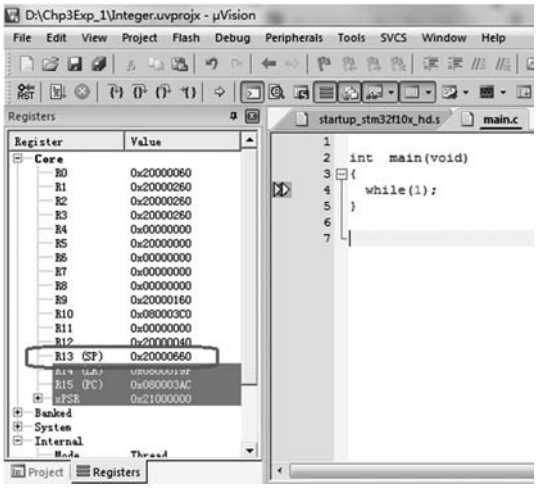

图 3.29　堆栈区示例 1

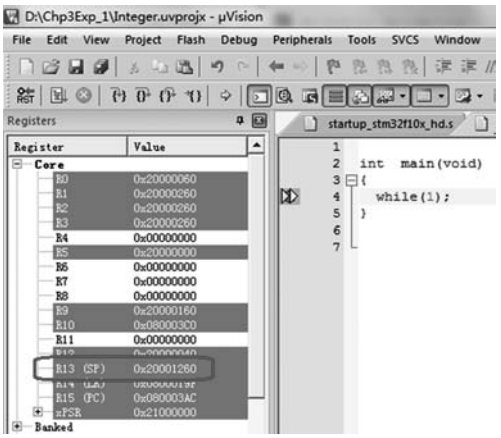

图 3.30　堆栈区示例 2

(1) 局部变量的存储空间分配。

函数内部定义的局部变量,其生存周期和作用域都局限在函数内。调用函数时,在堆栈中为函数内的局部变量分配存储空间。当函数执行完毕时才会弹栈释放堆栈空间。

函数中可以调用其他函数或者调用自身。函数调用自身,也就是所谓的“递归”函数。理论上函数调用嵌套的层数是没有限制的,但实际上每一次调用函数,都会消耗堆栈空间。然而堆栈空间是有限的,如果堆栈溢出,那么整个程序就崩溃了。这是非常严重的错误,所以规划堆栈区大小时一定要考虑函数调用嵌套的情况,而设计递归函数时更要注意最大的嵌套次数,避免发生堆栈溢出的错误。

(2) 静态局部变量的存储空间分配。

静态局部变量指在函数内部定义局部变量时用 static 关键字加以说明的变量。静态局部变量的作用域依然局限在函数体内,但是其生命周期会发生改变,函数结束时不会销毁静态局部变量,它在整个程序运行期间都有效。对于单片机程序来说,只要上电程序始终处于运行状态,就不会退出运行。

静态局部变量的存储空间是在编译时就在 SRAM 的空闲区域为其分配了存储空间,不会释放。但是作为局部变量,对它的访问依然局限在函数内部。

(3) 全局变量的存储空间分配。

全局变量作用域是整个项目,生命周期则是整个程序运行期间。静态全局变量仅仅是改变了作用域,其生命周期是不变的。

整个程序运行期间都有效的变量,都是在编译时分配存储空间,并且在程序运行期间存储空间都不释放,不能另做它用。

上述类型的变量都是在 SRAM 存储器中分配,而不同型号单片机内集成的 SRAM 存储容量差异极大,例如,STM32F103T4 内部只有 6KB 的 SRAM,而 STM32F103VE 内部有 64KB 的 SRAM。所以在嵌入式软件开发中,需要关注程序对存储空间的使用。

(4) 只读变量的存储空间分配。

定义变量时可以添加 const 关键字加以修饰,将变量定义为只读变量。只读变量与宏定义不同,只读变量有数据类型,并且会分配存储空间。

定义为只读变量,只是限定了对变量的写操作,只能在定义时初始化变量,在代码中只能读变量,而不能改写变量的数值。const 关键字仅仅限定了对变量的写操作,并没有改变变量的生命周期和作用域。

定义局部变量时,若添加了 const 关键字,则该只读的局部变量依然在栈区分配存储空间。但是当定义全局变量时,若添加了 const 关键字,则由于全局变量的存储空间为静态分配,又限定为只读,从其特性来看,完全可以在只读的 Flash 存储器中为其分配存储空间。

单片机芯片中集成的 SRAM 存储容量较小,而 Flash 存储容量较大,所以编译时会在 Flash 存储器中为只读的全局变量分配存储空间。

STM32F10x 系列单片机芯片片上集成的 Flash 存储器地址从 0x08000000 开始,而 SRAM 存储器地址从 0x20000000 开始。除了 const 全局变量在 Flash 存储器中分配存储空间以外,堆区、栈区以及静态分配的变量都在 SRAM 中分配存储空间。

例 3.17: 调试程序,观察变量的地址。

(1) 堆栈区在哪里?

如图 3.31 所示,编译连接成功后,Keil 5 会给出提示,其中 RW-data 指在 SRAM 中静态分配并且初始化了的变量字节数,ZI-data 指 SRAM 中分配的,没有指定初始化数值,初始化为零的字节数,这里包括堆区和栈区。RW-data 字节数加上 ZI-data 字节数,就是 SRAM 存储区中已经使用了的字节数。

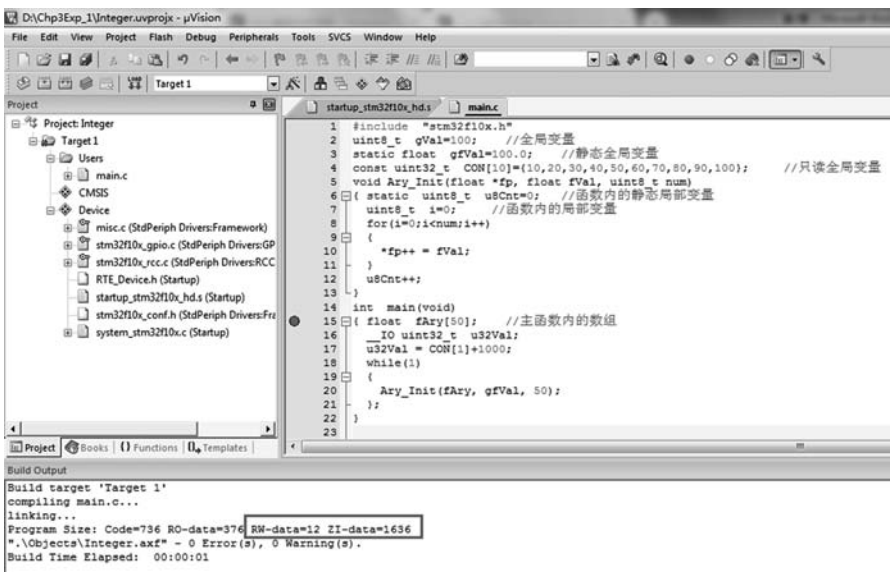

图 3.31 例 3.17 程序截图

栈区位于 SRAM 已使用区域的末端，所以当堆栈没有压入任何数据时，SP 指针的值为堆栈区最后一个字节地址＋1，即 SRAM 首地址 0x20000000＋RW-data＋ZI-data。调试环境下可以看到，进入 main()函数时 SP 寄存器的值为 0x20000670。

例程中没有修改启动文件，栈区大小为 0x00000400，因此堆栈区域地址为 0x20000270～0x2000066F。

(2) main()函数中定义的局部变量在哪里？

局部变量无论是否初始化，都是在程序执行过程中调用函数时从栈区分配存储空间。进入函数时，在栈区中为函数内的局部变量分配存储空间，离开函数时释放堆栈空间。

在 main()函数开始位置设置断点调试程序。进入 main()函数后，打开 Register 和 Watch 窗口，观察 CPU 寄存器和变量的值。在 Watch 窗口中输入变量名时前方添加了 C 语言中的取地址符号 &，Value 栏显示的是变量的地址，如图 3.32 所示。

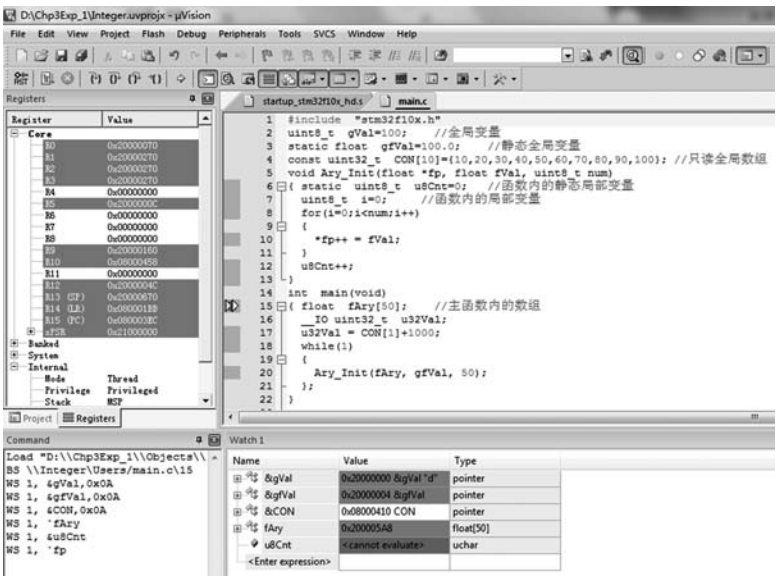

图 3.32　例 3.17 程序调试截图 1

此时，程序执行暂停在 float fAry[50]这行代码上。main()函数中定义了 fAry 数组和 u32Val 变量，两者一共占 204 字节。先从栈区中为 fAry 分配存储空间，即压栈 fAry[50]数组，执行后 SP 指针数值减 200，变为 0x200005A8。然后压栈 u32Val 变量，SP 指针数值减 4，变为 0x200005A4。所以两条定义变量的指令执行后，SP 指针数值为 0x200005A4，栈顶元素即为变量 u32Val，数组 fAry 的首地址为 0x200005A8，如图 3.33 所示。

(3) Ary_Init()函数中定义的静态局部变量在哪里？

main()函数中在 while(1)循环体中调用了函数 Ary_Init()，而在 Ary_Init()函数中定义了静态局部变量 u8Cnt，在图 3.33 的 Watch 窗口里显示为“< cannot evaluate >”，这是因为当前程序还在 main()函数中执行，没有调用过 Ary_Init()函数，还无法访问 u8Cnt 变量。当程序执行到 Ary_Init()函数时，情况如图 3.34 所示。

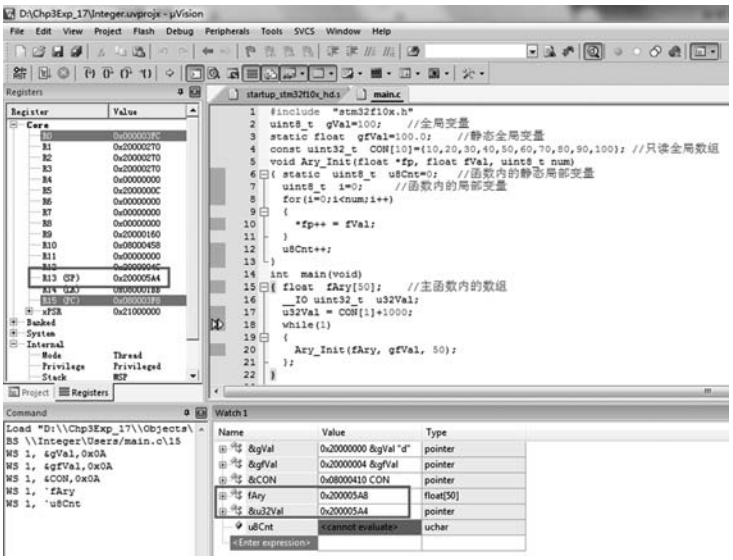

图 3.33　例 3.17 程序调试截图 2

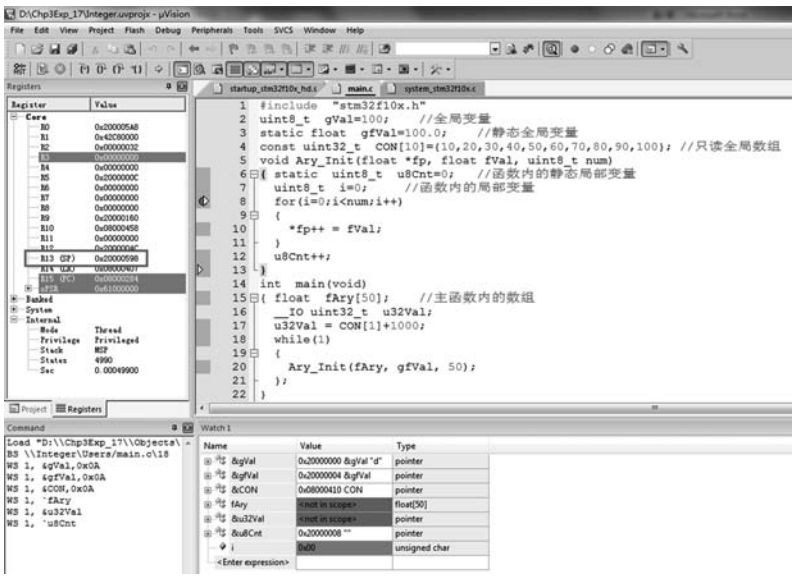

图 3.34　例 3.17 程序调试截图 3

u8Cnt 变量的地址为 0x20000008，从地址可以看出，该变量不在堆区或栈区范围，而是在静态分配的区域。

当程序执行离开 Ary_Init()函数时，u8Cnt 变量依然存在，也就是说，分配给它的存储空间没有释放。在整个程序运行期间，存储空间都不会释放。

(4) 函数嵌套调用对栈区的消耗。

由于需要在栈区压栈函数调用的返回地址、压栈局部变量等，所以进入 Ary_Init()

函数后，堆栈指针 SP 寄存器的值变为 0x20000598。

函数的每一次嵌套调用都需要消耗堆栈空间，一定要在启动文件中定义足够大的堆栈区！

递归函数会嵌套调用自身，根据函数调用时传递的参数数值决定嵌套调用的层数。嵌入式系统开发中需要小心谨慎地使用递归函数，调试递归函数程序时，应注意观察 SP 寄存器的数值，注意堆栈“溢出”错误。

(5) 全局变量在哪里？

调试程序进入 main()函数时，在 Watch 窗口注意观察全局变量的情况，如图 3.35 所示。

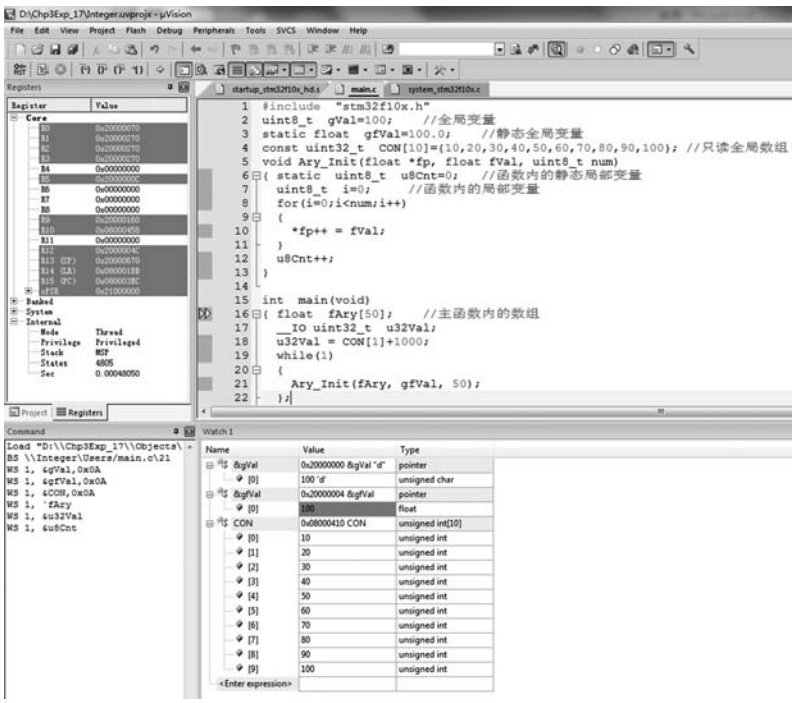

图 3.35　例 3.17 程序调试截图 4

全局变量是静态分配的，也就是说，编译时已经为变量分配了存储空间。只要程序加载，开始执行，任何时候都能访问全局变量。

程序中定义了全局变量 gVal、静态全局变量 gfVal 和 const 全局变量 CON，从图 3.35 可以看到，gVal 的地址为 0x20000000，gfVal 地址为 0x20000004，也就是说，编译器首先在 SRAM 中完成静态分配，然后才是堆区，最后是栈区。

片上 Flash 存储器地址从 0x08000000 开始，主要用于存放程序代码，const 类型的全局变量也在 Flash 存储器中分配存储空间。const 全局数组 CON 首地址为 0x08000410，从地址可以看出 CON 数组是在片上 Flash 中分配的存储空间。

在嵌入式系统开发中，由于片上集成的存储器大小有限，所以开发人员需要注意对存储空间的使用，根据项目实际情况定义栈区大小，注意避免在函数内定义较大的数组，以免造成堆栈“溢出”。