第3章　C51语言编程基础与Keil μVision3开发平台

单片机应用系统日趋复杂，对程序的可读性、升级与维护以及模块化要求越来越高，对软件编程要求也越来越高，要求编程人员在短时间内编写出执行效率高、运行可靠的程序代码。同时，也要方便多个编程人员来进行协同开发。

C51语言是近年来在8051单片机开发中，普遍使用的程序设计语言，能直接对8051单片机硬件进行操作，既有高级语言特点，又有汇编语言特点，因此在8051单片机程序设计中，得到广泛使用。

本章介绍8051单片机的C51语言，以及如何使用C51语言集成化开发平台Keil μVision3，进行C51程序设计与开发。

3.1　C51编程语言简介

用于8051单片机编程的C语言，在标准C基础上针对8051硬件特点进行扩展，并向8051上移植，经多年努力，C51已成为公认的高效、简洁的8051单片机的实用高级编程语言。与8051汇编语言相比，C51语言在功能上、结构性、可读性、可维护性上有明显优势，易学易用。

3.1.1　C51语言与8051汇编语言比较

与8051汇编语言相比， C51有如下优点。

（1）可读性好。C51语言程序比汇编语言程序的可读性好，编程效率高，程序便于修改、维护以及升级。

（2）模块化开发与资源共享。用C51开发的程序模块可不经修改，直接被其他工程所用，使得开发者能够很好地利用已有的大量标准C程序资源与丰富的库函数，减少重复劳动，同时也有利于多个工程师进行协同开发。

（3）可移植性好。为某种型号单片机开发的C语言程序，只需把与硬件相关的头文件和编译链接的参数进行适当修改，就可方便地移植到其他型号的单片机上。例如，为8051单片机编写的程序通过改写头文件以及少量的程序行，就可方便地移植到PIC单片机上。

（4）生成的代码效率高。当前较好的C51语言编译系统编译出来的代码效率只比直接使用汇编语言低20%左右，如果使用优化编译选项，最高可达到90%左右。

3.1.2 C51语言与标准C语言的比较

C51语言与标准C语言间有许多相同地方，但也有自身特点。不同的嵌入式C语言编译系统之所以与标准C语言有不同的地方，主要是由于它们所针对的硬件系统不同。对于8051单片机，目前广泛使用的是C51语言。

C51语言基本语法与标准C相同，是在标准C的基础上进行适合8051内核单片机硬件的扩展。深入理解C51语言对标准C语言的扩展部分以及它们的不同之处，是掌握C51语言的关键之一。

C51语言与标准C语言一些差别如下。

（1）库函数不同。标准C中不适合于嵌入式控制器系统的库函数，被排除在C51语言之外，如字符屏幕和图形函数。有些库函数必须针对8051的硬件特点来做出相应的开发。

例如，在标准C中，库函数printf和scanf，常用于屏幕打印和接收字符，而在C51语言中，主要用于串行口数据的收发。

（2）数据类型有一定区别。在C51中增加几种8051单片机的数据类型，在标准C的基础上又扩展了4种类型。例如，8051单片机包含位操作空间和丰富的位操作指令，因此，C51语言与标准C语言相比增加了位类型。

C51语言与标准C语言一些差别如下。

（3）C51语言变量存储模式与标准C语言中变量存储模式数据不一样。标准C最初是为通用计算机设计的，在通用计算机中只有一个程序和数据统一寻址的内存空间，而C51语言中变量的存储模式与8051单片机的各种存储器区紧密相关。

（4）数据存储类型不同。8051存储区可分为内部数据存储区、外部数据存储区以及程序存储区。

内部数据存储区可分为3个不同的C51存储类型：data、idata和bdata。

外部数据存储区分为2个不同的C51存储类型：xdata和pdata。

程序存储区只能读不能写，可能在8051内部或者在外部，C51语言提供的code存储类型用来访问程序存储区。

C51语言与标准C语言一些差别如下。

（5）标准C语言没有处理单片机中断的定义，而C51语言中有专门的中断函数。

（6）头文件不同。C51语言头文件必须把8051单片机内部的外设硬件资源（如定时器、中断、I/O等）相应的特殊功能寄存器写入到头文件内，而标准C不用。

（7）程序结构的差异。由于8051单片机的硬件资源有限，它的编译系统不允许太多的程序嵌套。其次，标准C语言所具备的递归特性不被C51语言支持。

但从数据运算操作、程序控制语句以及函数的使用上来说，C51与标准C几乎没有什么明显差别。

3.2 C51语言程序设计基础

本节在标准C基础上，了解掌握C51的数据类型和存储类型、C51的基本运算与流程控制语句、C51语言构造数据类型、C51函数以及C51程序设计的其他一些问题，为C51的程序开发打下基础。

3.2.1 C51语言中的数据类型与存储类型

1．数据类型

数据是单片机操作的对象，具有一定格式的数字或数值，数据的不同格式就称为**数据类型**。

Keil C51支持的基本数据类型见表3-1。

针对8051的硬件特点，C51在标准C基础上，**扩展了4种数据类型**（见表3-1中最后4行）。

注意，扩展的4种数据类型，不能使用指针来对它们存取。

表 3-1 Keil C51 支持的数据类型

数据类型	位数	字节数	值 域
signed char	8	1	-128～+127，有符号字符变量
unsigned char	8	1	0～255，无符号字符变量
signed int	16	2	-32 768～+32 767，有符号整型数
unsigned int	16	2	0～65 535，无符号整型数
signed long	32	4	-2 147 483 648～+2 147 483 647，有符号长整型数
unsigned long	32	4	0～+4 294 967 295，无符号长整型数
float	32	4	±1.175494E-38~±3.402823E+38
double	32	4	±1.175494E-38~±3.402823E+38
*	8～24	1～3	对象指针
bit	1		0 或 1
sfr	8	1	0～255
sfr16	16	2	0～65 535
sbit	1		可进行位寻址的特殊功能寄存器的某位的绝对地址

2．C51的扩展数据类型

下面对扩展的4种数据类型说明。

（1）**位变量bit的值**可以是1（true），也可是0（false）。

（2）**特殊功能寄存器sfr**。8051单片机的特殊功能寄存器分布在片内数据存储区的地址单元80H～FFH之间，“sfr”数据类型占用一个内存单元。利用它可访问8051单片机内部的所有特殊功能寄存器。

例如：sfr P1=0x90这一语句定义了P1端口在片内的寄存器，在程序后续的语句中可以用“P1=0xff”，使P1的所有引脚输出为高电平的语句来操作特殊功能寄存器。

（3）特殊功能寄存器sfr16

“sfr16”数据类型占用两个内存单元，用于操作占两个字节的特殊功能寄存器。例如：“sfr16 DPTR=0x82”语句定义了片内16位数据指针寄存器DPTR，其低8位字节地址为82H，高8位字节地址为83H。在程序的后续语句中就可对DPTR进行操作。

（4）特殊功能位sbit

sbit是指AT89S51片内特殊功能寄存器的可寻址位。例如：

```
sfr     PSW=0xd0;          //定义PSW寄存器地址为0xd0
sbit    OV=PSW^2;          //定义OV位为PSW.2
```

符号“^”前是特殊功能寄存器名字，“^”后的数字定义特殊功能寄存器可寻址位在寄存器中的位置，取值必须是0～7。

注意，不要把bit与sbit相混淆。bit定义普通的位变量，只能是二进制的0或1。sbit是定义特殊功能寄存器的可寻址位，值是可以进行位寻址的特殊功能寄存器的某位的绝对地址，例如，PSW寄存器OV位的绝对地址0xd2。

上面的例子还涉及到C51注释的写法问题，C51的注释写法有两种：

（1）//……，两个斜杠后面跟着的为注释语句，本写法只能注释一行，当换行时，必须在新行上重新写两个斜杠。

（2）/*……*/，一个斜杠与星号结合使用，本写法可注释任一行，即斜杠星号与星号斜杠之间的所有文字都作为注释，即注释有多行时，只需在注释的开始处，加斜杠星号，在注释的结尾处，加上星号斜杠即可。

加注释的目的是为了便于读懂程序，所有注释都不参与程序编译，编译器在编译过程中会自动删去注释。

3．数据存储类型

在讨论C51数据类型时，须同时提及它的存储类型，以及它与8051单片机存储器结构的关系，因为C51定义的任何数据类型必须以一定的方式，定位在8051单片机的某一存储区中，否则没有任何实际意义。

8051有片内、片外数据存储区，还有程序存储区。

片内的数据存储区是可读写的，8051的衍生系列最多可有256字节的内部数据存储区（例如AT89S52单片机），其中低128字节可直接寻址，高128字节（80H～FFH）只能间接寻址，从地址20H开始的16字节可位寻址。内部数据存储区可分为3个不同的数据存储类型：data、idata和bdata。

访问片外数据存储区比访问片内数据存储区慢，因为访问片外数据存储区要通过对数据指针加载地址来间接寻址访问。

C51提供两种不同的数据存储类型xdata和pdata来访问片外数据存储区。

程序存储区只能读不能写，可能在8051单片机内部或者外部，或外部和内部都有，由8051单片机硬件决定，C51提供了code存储类型来访问程序存储区。

C51存储类型与8051实际的存储空间的对应关系见表3-2。

下面对表3-2各种存储区作以说明。

（1）DATA区。寻址是最快的，应把常使用的变量放在该区，但该区存储空间有限，DATA区除了包含程序变量外，还包含了堆栈和寄存器组。DATA区声明中的存储类型标识符为data，通常指片内RAM128字节的内部数据存储的变量，可直接寻址。

表 3-2　C51 语言存储类型与 8051 存储空间的对应关系

存储区	存储类型	与存储空间的对应关系
DATA	data	片内 RAM 直接寻址区，位于片内 RAM 的低 128 字节
BDATA	bdata	片内 RAM 位寻址区，位于 20H～2FH 空间
IDATA	idata	片内 RAM 的 256 字节，必须间接寻址的存储区
XDATA	xdata	片外 64KB 的 RAM 空间，使用@DPTR 间接寻址
PDATA	pdata	片外 RAM 的 256 字节，使用@Ri 间接寻址
CODE	code	程序存储区，使用 DPTR 寻址

声明举例：

```
unsigned char data system_status=0;
unsigned int data unit_id[8];
char data inp_string[20];
```

标准变量和用户自声明变量都可存储在DATA区中，只要不超过DATA区的范围即可，由于C51用默认的寄存器组来传递参数，这样DATA区至少失去8字节空间。

另外，当内部堆栈溢出的时候，程序会莫名其妙地复位。这是因为8051没有报错机制，堆栈溢出只能以这种方式表示，因此要留有较大的堆栈空间来防止堆栈溢出。

（2）BDATA区。

DATA中的位寻址区，在该区中声明变量就可进行位寻址。BDATA区声明中的存储类型标识符为bdata，指的是片内RAM可位寻址的16字节存储区（字节地址为20H～2FH）中的128个位。下面是在BDATA区中声明的位变量和使用位变量的例子：

```
unsigned char bdata status_byte;
unsigned int bdata status_word;
sbit stat_flag=status_byte^4;
if(status_word^15)
  { …… }
stat_flag=1;
```

C51编译器不允许在BDATA区中声明float和double型变量。

（3）IDATA区。

该区使用寄存器作为指针来对片内RAM进行间接寻址，常用来存放使用比较频繁的变量。与外部存储器寻址相比，它的指令执行周期和代码长度相对较短。

IDATA区声明中的存储类型标识符为idata，指的是片内RAM的256字节的存储区，只能间接寻址，速度比直接寻址慢。

声明举例如下：

```
unsigned char idata system_status=0;
unsigned int idata unit_id[8];
char idata inp_string[16];
float idata out_value;
```


（4）PDATA区和XDATA区

两个区位于片外存储区，PDATA区和XDATA区声明中的存储类型标识符分别为pdata和xdata。

PDATA区只有256字节，仅指定256字节的外部数据存储区。

但XDATA区最多可达64KB，对应的xdata存储类型标识符可指定外部数据区64KB内的任何地址。

对PDATA区的寻址要比对XDATA区寻址快，因为对PDATA区寻址，只需装入8位地址，而对XDATA区寻址要装入16位地址，所以尽量把外部数据存储在PDATA区中。

对PDATA区和XDATA区的声明举例如下：

```
unsigned char xdata system_status=0;
unsigned int pdata unit_id[8];
char xdata inp_string[16];
float pdata out_value;
```

由于外部数据存储器与外部I/O口是统一编址的，外部数据存储器地址段中除了包含数据存储器地址外，还包含外部I/O口的地址。对外部数据存储器及外部I/O口的寻址将在本章的绝对地址寻址中介绍。

（5）程序存储区CODE。

程序存储区CODE声明的标识符为code，储存的数据是不可改变的。在C51编译器中可以用存储区类型标识符code来访问程序存储区。

声明举例如下：

```
unsigned char code a[ ]=
        {0x00,0x01,0x02,0x03,0x04,0x05,0x06, 0x07,0x08};
```

上面介绍了C51的数据存储类型，其大小和值域见表3-3。

单片机访问片内RAM比访问片外RAM相对快一些，所以应尽量把频繁使用的变量置于片内RAM。即采用data、bdata或idata存储类型，而将容量较大或使用不太频繁的那些变量置于片外RAM，即采用pdata或xdata存储类型。常量只能采用code存储类型。

表 3-3　C51 存储类型及其大小和值域

存储类型	长度/bit	长度/byte	值域
data	8	1	0～255
idata	8	1	0～255
bdata	1		0 或 1
pdata	8	1	0～255
xdata	16	2	0～65 535
code	16	2	0～65 535

变量存储类型定义举例：

（1）char data a1;　　/*字符变量a1被定义为data型，分配在片内RAM低128字节中*/

（2）float idata x,y; /*浮点变量x和y被定义为idata型，定位在片内RAM中，只能用间接寻址方式寻址*/

（3）bit bdata p;　　/*位变量p被定义为bdata型，定位在片内RAM中的位寻址区*/

（4）unsigned int pdata var1; /*无符号整型变量var1定义为pdata型，定位在片外RAM中，相当于@Ri间接寻址*/

（5）unsigned char xdata　a[2] [4]; /*无符号字符型二维数组变量a[2][4] 被定义为xdata存储类型，定位在片外RAM中，占据2×4=8字节，相当于@DPTR间接寻址*/

4. 数据存储模式

如果在变量定义时略去存储类型标识符，编译器会自动默认存储类型。进一步由SMALL、COMPACT和LARGE存储模式指令限制。例如，若声明char var1，则在使用SMALL存储模式下，var1被定位在data存储区，在使用COMPACT模式下，var1被定位在idata存储区；在LARGE模式下，var1被定位在xdata存储区中。

下面对存储模式作进一步说明。

（1）SMALL模式。该模式下，所有变量都默认位于8051单片机内部的数据存储器，与使用data指定存储器类型的方式一样。在此模式下，变量访问的效率高，但是所有数据对象和堆栈必须使用内部RAM。

（2）COMPACT模式

本模式下所有变量都默认在外部数据存储器的1页（256字节）内，这与使用pdata指定存储器类型是一样的。该类型适用于变量不超过256字节的情况，此限制是由寻址方式决定的，相当于使用数据指针@Ri寻址。与SMALL模式相比，该存储模式的效率比较低，对变量访问的速度也慢一些，但比LARGE模式快。

（3）LARGE模式

本模式下所有变量都默认位于外部数据存储器，相当于用@DPTR寻址。通过数据指针访问外部数据存储器的效率较低，特别是当变量为2字节或更多字节时，该模式要比SMALL和COMPACT产生更多的代码。

在固定的存储器地址上进行变量传递，是C51的标准特征之一。

3.2.2 C51语言的特殊功能寄存器及位变量定义

介绍C51如何对特殊功能寄存器及位变量进行定义并访问。

1. 特殊功能寄存器的C51定义

C51语言允许通过使用关键字sfr、sbit或直接引用编译器提供的头文件来对特殊功能寄存器（SFR）进行访问。

特殊功能寄存器分布在片内RAM高128字节中，只能采用直接寻址方式。

（1）使用关键字定义sfr。为能直接访问特殊功能寄存器SFR，C51提供了一种定义方法，即引入关键字sfr，语法如下：

sfr 特殊功能寄存器名字=特殊功能寄存器地址；

例如：

```
sfr IE=0xA8;            //中断允许寄存器IE地址A8H
sfr TCON=0x88;          //定时器/计数器控制寄存器地址88H
sfr SCON=0x98;          //串行口控制寄存器地址98H
```

在8051中，要访问16位SFR，要用关键字sfr16。16位SFR的低字节地址须作为“sfr16”的定义地址，例如：

```
sfr16  DPTR=0x82      //DPTR 的低8位地址为82H，高8位地址为83H
```


（2）通过头文件访问SFR。各种衍生型的8051单片机的特殊功能寄存器的数量与类型有时是不相同的，对其访问可通过头文件访问来进行。

为用户处理方便，C51把8051（或8052单片机）常用的特殊功能寄存器和其中的可寻址位进行了定义，放在一个reg51.h（或reg52.h）的头文件中。

当用户要使用时，只需在使用之前用一条预处理命令#include<reg51.h>把这个头文件包含到程序中，就可使用特殊功能寄存器名和其中的可寻址位名称了。用户可对头文件进行增减。

头文件引用举例如下：

```
#include<reg51.h>              //包含8051单片机的头文件
void  main(void)
{
    TL0=0xf0; //给T0低字节TL0设置时间常数，已在reg51.h中定义
    TH0=0x3f; //给T0高字节TH0设置时间常数，已在reg51.h中定义
    TR0=1;    //启动定时器0
    ……
}
```

（3）特殊功能寄存器中的位定义

① sbit 位名=特殊功能寄存器^位置；

例如：

```
sfr   PSW=0xd0;        //定义PSW 寄存器的字节地址0xd0
sbit  CY=PSW^7;        //定义CY位为PSW.7，地址为0xd0
sbit  OV=PSW^2;        //定义OV位为PSW.2，地址为0xd2
```

② sbit 位名=字节地址^位置；

例如：

```
sbit  CY=0xd0^7;              // CY位地址为0xd7
sbit  OV=0xd0^2;              // OV位地址为0xd2
```


③ sbit 位名=位地址；

将位的绝对地址赋给变量，位地址必须在0x80～0xff。

例如：

```
sbit  CY=0xd7;                // CY位地址为0xd7
sbit  OV=0xd2;                // OV位地址为0xd2
```

【例】AT89S51单片机片内P1口的各寻址位的定义如下：

```
sfr   P1=0x90;
sbit  P1_7= P1^7;
sbit  P1_6= P1^6;
sbit  P1_5= P1^5;
sbit  P1_4= P1^4;
sbit  P1_3= P1^3;
sbit  P1_2= P1^2;
sbit  P1_1= P1^1;
sbit  P1_0= P1^0;
```


2. 位变量的C51定义

（1）由于8051可位操作，C51扩展的“bit”数据类型用来定义位变量，这是与标准C的不同之处。

C51采用关键字“bit”来定义位变量，一般格式为： bit bit_name;

例如：

```
bit  ov_flag;              //将ov_flag定义为位变量
bit  lock_pointer;         //将lock_pointer定义为位变量
```

（2）函数可以包含类型为bit的参数，也可将其作为返回值。C51程序函数可以包含类型为“bit”的参数，也可将其作为返回值。例如：

```
bit  func(bit b0, bit b1);    // 位变量b0与b1作为函数func的参数
{
     ......
     return(b1);              // 位变量b1作为return函数的返回值
```


（3）位变量定义的限制。位变量不能用来定义指针和数组。例如：

```
bit  *ptr;              // 错误，不能用位变量来定义指针
bit  array[ ];          // 错误，不能用位变量来定义数组array[ ]
```

定义位变量时，允许定义存储类型，位变量都被放入一个位段，此段总是位于8051的片内RAM中，因此其存储类型限制为DATA或IDATA，如果将位变量定义成其他类型，将会导致编译时出错。

3.2.3 C51语言的绝对地址访问

如何对8051片内RAM、片外RAM及I/O空间进行访问，C51提供两种常用的访问绝对地址的方法。

1. 绝对宏

编译器提供了一组宏定义对code、data、pdata和xdata空间进行绝对寻址。

程序中用“#include<absacc.h>”来对absacc.h中声明的宏来访问绝对地址，包括CBYTE、CWORD、DBYTE、DWORD、XBYTE、XWORD、PBYTE、PWORD，具体使用参见absacc.h头文件。其中：

- CBYTE以字节形式对code区寻址；
- CWORD以字形式对code区寻址；
- DBYTE以字节形式对data区寻址；
- DWORD以字形式对data区寻址；
- XBYTE以字节形式对xdata区寻址；
- XWORD以字形式对xdata区寻址；
- PBYTE以字节形式对pdata区寻址；
- PWORD以字形式对pdata区寻址。

【例】片内RAM、片外RAM及I/O定义的程序如下：

```
#include<absacc.h>
#define PORTA XBYTE[0xFFC0]
                //将PORTA定义为外部I/O口，地址为0xFFC0，长度8位
#define NRAM  DBYTE[0x50]
                //将NRAM定义为片内RAM，地址为0x50，长度8位
main( )
{
    PORTA=0x3d;  //将数据3DH写入地址为0xffc0的外部I/O端口PORTA中
    NRAM=0x01;  //将数据01H写入片内RAM的0x40单元
}
```


2. _at_ 关键字

关键字 _at_ 可对指定的存储器空间的绝对地址访问，格式如下：

[存储器类型] 数据类型说明符 变量名 _at_ 地址常数

其中，存储器类型为C51能识别的数据类型；数据类型为C51支持的数据类型；地址常数用于指定变量的绝对地址，必须位于有效的存储器空间之内；使用 _at_ 定义的变量必须为全局变量。

【例】使用关键字 _at_ 实现绝对地址的访问，程序如下：

```
void  main(void)
{
    data unsigned char y1 _at_ 0x50; //在data区定义字节变量y1，地址为
    50H
   xdata unsigned int y2 _at_ 0x4000;//在xdata区定义字变量y2，地址为
                                     //4000H
    y1=0xff;
    y2=0x1234;
    ……
    while(1);
}
```


【例】将片外RAM 2000H开始的连续20字节清0，程序如下：

```
xdata unsigned char buffer[20] _at_ 0x2000;
void main(void)
{
    unsigned char i;
    for(i=0; i<20; i++)
    {
       buffer[i]=0
    }
}
```


【例】如把片内RAM 40H单元开始的8个单元内容清0，程序如下：

```
xdata unsigned char buffer[8] _at_ 0x40;
void  main(void)
{

unsigned char j ;
     for(j=0;j<8;j++)
     {
        buffer[j]=0
      }
}
```


3.2.4 C51的基本运算

与标准C类似，主要包括算术运算、关系运算、逻辑运算、位运算和赋值运算及其表达式等。

1. 算术运算符

算术运算符及说明见表3-4。

表 3-4 算术运算符及其说明

符 号	说 明	举例（设 x=10, y=3）
+	加法运算	z=x+y; //z=13
−	减法运算	z=x-y; //z=7
*	乘法运算	z=x*y; //z=30
/	除法运算	z=x/y; //z=3
%	取余数运算	z=x%y; //z=1
++	自增 1	
−−	自减 1	

C51中表示加1和减1时可以采用自增运算符和自减运算符，自增和自减运算符是使变量自动加1或减1，自增和自减运算符放在变量前和变量之后是不同的，见表3-5。

表 3-5　自增运算符与自减运算符

运算符	说　明	举例（设 x 初值为 4）
x++	先用 x 的值，再让 x 加 1	y=x++;　　// y 为 4，x 为 5
++x	先让 x 加 1，再用 x 的值	y=++x;　　// y 为 5，x 为 5
x--	先用 x 的值，再让 x 减 1	y=x--;　　// y 为 4，x 为 3
--x	先让 x 减 1，再用 x 的值	y=--x;　　// y 为 3，x 为 3

2．逻辑运算符

逻辑运算的结果只有“真”和“假”两种，“1”表示真，“0”表示假。表3-6列出了逻辑运算符及其说明。

表 3-6　逻辑运算符及其说明

运算符	说　明	举例（设 a=2,b=3）
&&	逻辑与	a&&b;　　//返回值为 1
‖	逻辑或	a‖b;　　//返回值为 1
!	逻辑非	!a;　　//返回值为 0

例如条件“10>20”为假，“2<6” 为真，则逻辑与运算为：

(10>20)&&(2<6)=0&&1=0。

3．关系运算符

关系运算符是判断两个数之间的关系。说明如表3-7所示。

表 3-7　关系运算符及其说明

符　号	说　明	举例（设 a=2,b=3）
>	大于	a> b;　//返回值为 0
<	小于	a< b;　//返回值为 1
>=	大于等于	a>=b;　//返回值为 0
<=	小于等于	a<=b;　//返回值为 1
==	等于	a==b;　//返回值为 0
!=	不等于	a!=b;　//返回值为 1

4．位运算

位运算符及其说明见表3-8。

表 3-8　位运算及其说明

符　号	说　明	举例
&	按位逻辑与	0x19&0x4d=0x09
|	按位逻辑或	0x19 | 0x4d =0x5d
^	按位异或	0x19^0x4d =0x54
~	按位取反	x=0x0f，则~x=0xf0
<<	按位左移(高位丢弃，低位补 0)	y=0x3a，若 y<<2，则 y=0xe8
>>	按位右移(高位补 0，低位丢弃)	w=0x0f，若 w>>2，则 w=0x03

在实际应用中，常想改变I/O口中某一位的值，而不影响其他位，如果I/O口可位寻址的，这个问题就很简单。但有时外扩的I/O口只能进行字节操作，要想实现单独位控，就要采用位操作。

【例】 编程将扩展的某I/O 口 PORTA（只能字节操作）的PORTA.5清 0，PORTA.1置1，程序如下：

```
#define <absacc.h>        //定义片外 I/O 口变量PORTA要用该头文件#define
PORTA XBYTE[0xffc0]              //定义一个片外 I/O 口变量PORTA
void main( )
{
    ……
    PORTA=( PORTA&0xdf)|0x02;
    ……
}
```


程序中，第2行定义一个片外 I/O 口变量PORTA，地址为片外数据存储区的0xffc0。在main( )函数中，“PORTA=(PORTA&0xdf)|0x02”的作用是先用运算符“&”将PORTA.5置成0，然后再用“|0x02”运算将PORTA.1置为1。

5. 指针和取地址运算符

指针是C51语言中一个十分重要的概念，指针变量用于存储某个变量的地址，C51用“*”和“&”运算符来提取变量内容和变量地址，见表3-9。

表 3-9 赋值、指针和取值运算及其说明

符 号	说 明
*	提取变量的内容
&	提取变量的地址

提取变量的内容和变量的地址的一般形式分别为：

目标变量=*指针变量 //将指针变量所指的存储单元内容赋值给目标变量

指针变量=&目标变量 //将目标变量的地址赋值给指针变量

例如：

```
a=&b;          //取b变量的地址送至变量a
c=*b;          //把以指针变量b为地址的单元内容送至变量c
```

指针变量中只能存放地址（即指针型数据），不能将非指针类型的数据赋值给指针变量。例如：

```
int i ;                  //定义整型变量i
int *b;                  //定义指向整数的指针变量b
b=& i;                   //将变量i的地址赋给指针变量b
b= i;     //错，指针变量b只能存放变量指针（变量地址），不能存放变量i的值
```


3.2.5 C51的分支与循环程序结构

C51程序按结构可分为3类，即顺序、分支和循环结构。顺序结构是基本结构，程序自上而下，从main( )的函数开始一直到程序结束，只有一条路可走，无其他路径可选，结构较简单和便于理解，这里仅介绍分支结构和循环结构。

1. 分支控制语句

分支控制语句有：if语句和switch语句。

（1）if语句用来判定所给定的条件是否满足，根据判定结果决定执行两种操作之一。

if语句的基本结构如下：

if (表达式) {语句}

括号中的表达式成立时，程序执行大括号内的语句，否则程序跳过大括号中的语句部分，而直接执行下面的其他语句。

程序跳过大括号中的语句部分，而直接执行下面的其他语句。

C51提供3种形式的if语句：

形式1

if (表达式) {语句}

例如：

if (x>y) {max=x; min=y;}

即如果x>y，则x赋给max，y赋给min。如果x>y不成立，则不执行大括号中的赋值运算。

形式2

if (表达式) {语句1;} else {语句2;}

例如：

if (x>y)

{max=x; }

else {max=y;}

本形式相当于双分支选择结构。

形式3

if (表达式1) {语句1;}

else if (表达式2) {语句2;}

else if (表达式3) {语句3;}

……

else {语句n;}

例如：

if (x>100) {y=1;}

else if (x>50) {y=2;}

else if (x>30) {y=3;}

else if (x>20) {y=4;}

else {y=5;}

本形式相当于串行多分支选择结构。

在if语句中又含有一个或多个if语句，这称为if语句的嵌套。应当注意if与else的对应关系，else总是与它前面最近的一个if语句相对应。

（2）switch语句。

if语句只有两个分支可选择，而switch语句是多分支选择语句。

switch语句的一般形式如下：

```
switch (表达式1)
{
        case 常量表达式1:{语句1;}break;
        case 常量表达式2:{语句2;}break;
        ……
        case 常量表达式n:{语句n;}break;
        default:{语句n+1;}
}
```

上述switch语句说明如下。

switch语句

（1）每一case常量表达式须互不相同，否则将混乱。

（2）各个case和default出现次序，不影响程序执行的结果。

（3）switch括号内表达式的值与某case后面的常量表达式的值相同时，就执行它后面的语句，遇到break语句则退出switch语句。若所有的case中的常量表达式的值都没有与switch语句表达式的值相匹配时，就执行default后面的语句。

（4）如果在case语句中遗忘了break语句，则程序执行了本行之后，不会按规定退出switch语句，而是将执行后续的case语句。在执行1个case分支后，使流程跳出switch结构，即中止switch语句的执行，可以用1条break语句完成。switch语句的最后一个分支可以不加break语句，结束后直接退出switch结构。

【例】在单片机程序设计中，常用switch语句作为键盘中按键按下的判别，并根据按下键的键号跳向各自的分支处理程序。

```
nput:  keynum=keyscan( )
witch(keynum)

    case 1:     key1( ); break;//如果按下键为1键，则执行函数key1( )
    case 2:key2( ); break;//如果按下键为2键，则执行函数key2( )
    case 3:key3( ); break;//如果按下键为3键，则执行函数key3( )
    case 4:key4( ); break;//如果按下键为4键，则执行函数key4( )
    ……
    default:goto input
```

例子中的keyscan( )是另行编写的一个键盘扫描函数，如有键按下，该函数就会得到按下键的键值，将键值赋予变量keynum。如果键值为2，则执行键值处理函数key2( )后返回；如果键值为4，则执行key4( )函数后返回。执行完1个键值处理函数后，则跳出switch语句，从而达到按下不同的按键来进行不同的键值处理的目的。

2．循环控制语句

许多实用程序都包含循环结构，熟练掌握和运用循环结构的程序设计是C51语言程序设计的基本要求。

实现循环结构的语句有以下3种：while语句、do-while语句和for语句。

（1）while语句。语法形式为：

```
while(表达式)
    {
        循环体语句;
    }
```

表达式是while循环能否继续的条件，如果表达式为真，就重复执行循环体语句；反之，则终止循环体内的语句。

hile循环结构特点

循环条件测试在循环体开头，要想执行重复操作，首先必须进行循环条件的测试，如条件不成立，则循环体内的重复操作一次也不能执行。

例如：

```
while((P1&0x80)= =0)
{ }
```

while中的条件语句对AT89S8051单片机的P1口的P1.7位进行测试，如果P1.7为低（0），则由于循环体无实际操作语句，故继续测试下去（等待），一旦P1.7的电平变高（1），则循环终止。

（2）do-while语句。

语法形式为：

```
do
  {
    循环体语句;
  }
while(表达式);
```

do-while语句特点是先执行内嵌的循环体语句，再计算表达式，如表达式的值为非0，则继续执行循环体语句，直到表达式的值为0时结束循环。

由do-while构成的循环与while循环的重要区别是：while循环的控制出现在循环体之前，只有当while后面表达式的值非0时，才可能执行循环体；在do-while构成的循环中，总是先执行一次循环体，然后再求表达式的值，因此无论表达式的值是0还是非0，循环体至少要被执行一次。

在do-while循环体中，要有能使while后表达式的值变为0的操作，否则，循环会无限制地进行下去。根据经验，do-while循环用的并不多，大多数的循环用while来实现会直观。

【例】实型数组sample存有10个采样值，编写程序段，要求返回其平均值（平均值滤波）。程序如下：

```
float avg(float *sample)
    {
    float sum=0;
    char n=0;
    do
       {
        sum+=sample[n];
            n++;
       } while(n<10);
       return(sum/10);
    }
```

（3）基于for语句的循环。

3种循环常用的是for循环。不仅可用于循环次数已知的情况，也可用于循环次数不确定而只给出循环条件情况，完全可替代while语句。

for循环的一般格式为：

```
for(表达式1;表达式2;表达式3)
    {
     循环体语句;
    }
```

for是关键字，括号中常含有3个表达式，各表达式间用“；”隔开。这3个表达式可以是任意形式的表达式，通常主要用于for循环控制。紧跟在for()之后的循环体，在语法上要求是 1 条语句；若在循环体内需要多条语句，应用大括号括起来组成复合语句。

for执行过程如下：

① 计算“表达式1”，表达式1通常称为“初值设定表达式”。

② 计算“表达式2”，表达式2通常称为“终值条件表达式”，若满足条件，转下一步，若不满足条件，则转步骤⑤。

③ 执行1次for循环体。

④ 计算“表达式3”，“表达式3”通常称为“更新表达式”转向步骤②。

⑤ 结束循环，执行for循环之后的语句。

下面对for语句的几个特例进行说明。

① for语句中的小括号内的3个表达式全部为空。

例如：

```
for( ; ; )
{
    循环体语句;
}
```

在小括号内只有两分号，无表达式，这意味着没有设初值，无判断条件，循环变量为增值，它的作用相当于while(1)，这将导致一个无限循环。一般在编程时，需要无限循环时，可采用这种形式的for循环语句。

② for语句的3个表达式中，表达式1缺省。

例如：

```
for(;i<=100;i++)sum=sum+i;
```

即不对i设初值。

③ for语句的3个表达式中，表达式2缺省。

例如：

```
for(i=1;;i++)sum=sum+i;
```

即不判断循环条件，认为表达式始终为真，循环将无休止地进行下去。

④ for语句的3个表达式中，表达式1、表达式3省略。

例如：

```
for(;i<=100;)
{
    sum=sum+i;
    i++;
}
```

⑤ 没有循环体的for语句。

例如：

```
int a=1000;
for(t=0;t<a;t++)
{;}
```

本例典型应用就是软件延时。可用循环结构来实现，即循环执行指令，消磨一段已知的时间。指令的执行时间是靠一定数量的时钟周期来计时的，如果使用12MHz晶振，则12个时钟周期花费的时间为1μs。

【例】编写一个延时1ms程序。

```
void delayms( unsigned char int j)
{
    unsigned char i;
    while(j- -)
    {
       for(i=0;i<125;i++)
       {;}
    }
}
```

如把上述程序段编译成汇编代码分析，用for的内部循环大约延时8μs，但不是特别精确。不同编译器会产生不同延时，因此i的上限值125应根据实际情况进行补偿调整。

【例】求1+2+3…+100的累加和。

用for语句编写的程序如下：

```
#include <reg51.h>
#include <stdio.h>
main( )
{
    int  nvar1, nsum;
  for(nvar1=0,nsum=1;nsum<=100;nsum++)
  nVar1+=ncount;                                    //累加求和
  while(1);
}
```

编写无限循环程序段，可用以下3种结构。

【例】无限循环的结构实现。

① 使用while(1)的结构：

```
while(1)
{    代码段;
}
```

② 使用for（；；）的结构：

```
for(;;)
{    代码段;
}
```

③ 使用do-while(1)的结构：

```
do
{    代码段;
} while(1);
```

3. break语句、continue语句和goto语句

在循环体执行中，如满足循环判定条件的情况下跳出代码段，可使用 break语句或continue语句；如要从任意地方跳转到代码某地方，可使用goto语句。

（1）break语句

循环结构中，可使用break语句跳出本层循环体，马上结束本层循环。

【例】执行如下程序段。

```
void  main(void )
{    int i,sum;
     sum=0;
     for(i=1;i<=10;i++)
     {  sum=sum+i;
        if(sum>5) break;
 print(“sum=%d\n”, sum); /*通过串口向计算机屏幕输出显示sum值*/
     }
}
```

本例如没有break语句，程序将进行10次循环；当i=3 时，sum的值为6，此时，if语句的表达式“sum>5”的值为1，于是执行break语句，跳出for循环，从而提前终止循环。

因此在一个循环程序中，既可通过循环语句中的表达式来控制循环是否结束，还可直接通过break语句强行退出循环结构。

（2）continue语句

作用及用法与break语句类似，区别：当前循环遇到break，是直接结束循环，若遇上continue，则是停止当前这一层循环，然后直接尝试下一层循环。可见，continue并不结束整个循环，而仅仅是中断这一层循环，然后跳到循环条件处，继续下一层的循环。当然，如果跳到循环条件处，发现条件已不成立，那么循环也会结束。

【例】输出整数1～100的累加值，但要求跳过所有个位为3的数。

为完成题目要求，在循环中加一个判断，如果该数各位是3，就跳过该数不加。如何来判断1～100的数中那些数个位是3呢？用求余数的运算符“%”，将一个两位以内的正整数，除以10后，余数是3，就说明这个数的个位为3。例如对于数73，除以10后，余数是3。

根据以上分析，参考程序如下：

```
void  main(void )
{       int i, sum=0;
        sum=0;
        for(i=1;i<=100;i++)
        {       if(i%10==3)                continue;
                sum=sum+i;
        print(“sum=%d\n”, sum); /*在计算机屏幕显示sum值,
}
```

（3）goto语句

无条件转移语句，当执行goto语句时，将程序指针跳转到goto给出的下一条代码。基本格式如下：

goto　标号

【例】计算整数1～100的累加值，存放到sum中。

```
void  main(void )
{   unsigned char i
int  sum;
    sumadd:
sum=sum+i;
i++;
if(i<101)
    { goto sumadd;
    }
}
```

goto语句在C51中经常用于无条件跳转某条必须执行的语句以及在死循环程序中退出循环。为方便阅读，也为了避免跳转时引发错误，在程序设计中要慎重使用goto语句。

3.2.6 C51的数组

在C51程序设计中，数组使用的较为广泛。

1. 数组简介

数组是同类数据的一个有序结合，用数组名来标识。整型变量的有序结合称为整型数组，字符型变量的有序结合称为字符型数组。数组中的数据，称为数组元素。

数组中各元素的顺序用下标表示，下标为n的元素可以表示为数组名[n]。改变 [ ]中的下标就可以访问数组中的所有的元素。

数组有一维、二维、三维和多维数组之分。C51语言中常用的一维、二维数组和字符数组。

（1）一维数组

具有一个下标的数组元素组成的数组成为一维数组，一维数组形式如下：

类型说明符　数组名[元素个数]；

其中，数组名是一个标识符，元素个数是一个常量表达式，不能是含有变量的表达式：

例如：

```
int array1[8]
```

定义名为array1的数组 ，包含8个整型元素，在定义数组时，可对数组进行整体初始化，若定义后对数组赋值，则只能对每个元素分别赋值。例如：

```
int a[3]={2,4,6};    /*给全部元素赋值，a[0]=2，a[1]=4，a[2]=6 * /
int b[4]={5,4,3,2};
```

/*给全部元素赋值，b[0]=5，b[1]=4，b[2]=3，b[3]=2* /

（2）二维数组或多维数组

具有两个或两个以上下标的数组，称为二维数组或多维数组。定义二维数组的一般形式如下：

类型说明符 数组名[行数] [列数]；

其中，数组名是一个标识符，行数和列数都是常量表达式。例如：

```
float  array2 [4][3] /* array2数组，4行3列共12个浮点型元素* /
```

二维数组可以在定义时进行整体初始化，也可在定义后单个地进行赋值。

例如：

```
int a[3][4]={1,2,3,4},{5,6,7,8},{9,10,11,12}; /*a数组全部初始化*/
int b[3][4]={1,3,5,7},{2,4,6,8},{ };  /* b数组部分初始化，未初始化的元素为0* /
```

（3）字符数组

若一个数组的元素是字符型的，则该数组就是一个字符数组。例如：

```
char a[10]= { 'B', 'E', 'I', ' ', 'J', 'I', 'N', 'G',
  '\0' };                              /*字符串数组* /
```

定义了一个字符型数组a[ ]，有10个数组元素，并且将9个字符（其中包括一个字符串结束标志 '\0' ）分别赋给了a[0]～a[8]，剩余的a[9]被系统自动赋予空格字符。

C51还允许用字符串直接给字符数组置初值，例如：

```
char a[10]= {"BEI JING"};
```

用双引号括起来的一串字符，成为字符串常量，C51编译器会自动地在字符串末尾加上结束符 '\0' 。

用单引号括起来的字符为字符的ASCII码值，而不是字符串。例如 'a' 表示a的ASCII码值61H，而 "a" 表示一个字符串，由两个字符a和\0组成。

一个字符串可以用一维数组来装入，但数组的元素数目一定要比字符多一个，以便C51编译器自动在其后面加入结束符 '\0' 。

2．数组的应用

在C51的编程中，数组一个非常有用的功能是查表。例如数学运算，编程者更愿意采用查表计算而不是公式计算。例如，对于传感器的非线性转换需要进行补偿，使用查表法就要有效的多。再如，LED显示程序中根据要显示的数值，找到对应的显示段码送到LED显示器显示。表可以事先计算好后装入程序存储器中。

【例】使用查表法，计算数0～9的平方。

```
#define uchar unsigned char
uchar code square[0,1,4,9,16,25,36,49,64,81] ; /*0～9的平方表*/
uchar fuction(uchar number)
{    return  square[number]};      /*返回要求得其平方的数 */
main( )
{    result= fuction(7); /*函数fuction( )的返回值为7，其平方49存入result
     单元 */
}
```

程序开始，"uchar code square[0,1,4,9,16,25,36,49,64,81] ; "定义了一个无符号字符型的数组square[ ]，并对其进行了初始化，将数0～9的平方值赋予了数组square[ ]，数据类型代码code指定编译器将平方表定位在程序存储器中。

主函数调用函数fuction( )，假设得到返回值number=7；从square数组中查表获得相应的求得其平方的数为49。执行result= fuction(7)后，result的结果为相应的平方数49

3．数组与存储空间

当程序中设定了一个数组时，C51编译器就会在系统的存储空间中开辟一个区域，用于存放数组的内容。数组就包含在这个由连续存储单元组成的模块的存储体内。对字符数组而言，占据了内存中一连串的字节位置。对整型（int）数组而言，将在存储区中占据一连串连续的字节对的位置。对长整型（long）数组或浮点型（float）数组，一个数组成员将占有4字节的存储空间。

当一维数组被创建时，C51编译器就会根据数组的类型在内存中开辟一块大小等于数组长度乘以数据类型长度（即类型占有的字节数）的区域。

对于二维数组a[m][n]而言，其存储顺序是按行存储，先存第0行元素的第0列、第1列、第2列，直至第n-1列，然后返回到存第1行元素的第0列、第1列、第2列，直至第n-1列，…… ，如此顺序存储，直到第m-1行的第n-1列。

当数组特别是多维数组中大多数元素没有被有效利用地利用时，就会浪费大量的存储空间。对于51单片机，不拥有大量的存储区，其存储资源极为有限，因此在进行C51语言编程开发时，要仔细地根据需要来选择数组的大小。

3.2.7　C51的指针

C51支持基于存储器的指针和一般指针两种指针类型。当定义一个指针变量时，若未给出它所指向的对象的存储类型，则被认为是一般指针，反之若给出了它所指向对象的存储类型，则被认为是基于存储器的指针。

基于存储器的指针类型由C51语言源代码中存储类型决定，用这种指针可以高效访问对象，且只需1~2字节。

一般指针占用3字节：1个字节为存储器类型，2个字节为偏移量。存储器类型决定了对象所用的8051的存储空间，偏移量指向实际地址。一个一般指针可以访问任何变量而不管它在8051存储器的位置。

1．通用指针

C51提供一个3字节的通用指针，通用指针声明和使用与标准C语言完全一样。通用指针的形式如下：

数据类型 *指针变量；

例如：

```
uchar *pz
```

例中pz就是通用指针，用3字节来存储指针，第一字节表示存储器类型，第二、三字节分别是指针所指向数据地址的高字节和低字节，这种定义很方便但速度较慢，在所指向的目标存储器空间不明确时普遍使用。

2．基于存储器的指针

存储器指针在定义时指明了存储器类型，并且指针总是指向特定的存储器空间（片内数据RAM、片外数据RAM或程序ROM）。例如：

```
char xdata *str;          // str指向xdata区中的char型数据
int xdata *pd;            // pd指向外部RAM区中的int型整数
```

由于定义中已经指明了存储器类型，因此，相对于通用指针而言，指针第一个字节省略，对于data、bdata、idata与pdata存储器类型，指针仅需要1B，因为它们的寻址空间都在256B以内，而code和xdata存储器类型则需要2B指针，因为它们的寻址空间最大为64KB。

使用存储器指针好处是节省了存储空间，编译器不用为存储器选择和决定正确的存储器操作指令来产生代码，使代码更加简短，但必须保证指针不指向所声明的存储区以外的地方，否则会产生错误。通用指针产生的代码执行速度比指定存储区的指针要慢，因为存储区在运行前是未知的，编译器不能优化存储区访问，必须产生可以访问任何存储区的通用代码。

由上所述，使用存储器指针比使用通用指针效率高，存储器指针所占空间小，速度更快，在存储器空间明确时，建议使用存储器指针，如果存储器空间不明确则使用通用指针。

3.3　C51语言的函数

函数是一个完成一定相关功能的执行代码段。在高级语言中，函数与另外两个名词“子程序”和“过程”用来描述同样的事情。在C51语言中使用的是函数这个术语。

C51语言中函数的数目是不限制的，但是一个C51程序必须至少有一个函数，以main为名，称为主函数，主函数是唯一的，整个程序从这个主函数开始执行。

C51语言还可建立和使用库函数，可由用户根据需求调用。

3.3.1　函数的分类

从结构上分，C51语言函数可分为主函数main( )和普通函数两种。而普通函数又划分为两种：标准库函数和用户自定义函数。

1．标准库函数

标准库函数是由C51编译器提供的。编程者在进行程序设计时，应该善于充分利用这些功能强大、资源丰富的标准库函数资源，以提高编程效率。用户可直接调用C51库函数而不需为这个函数写任何代码，只需要包含具有该函数说明的头文件即可。例如调用输出函数printf时，要求程序在调用输出库函数前包含以下的include 命令：

```
#include <stdio.h>
```

2. 用户自定义函数

用户自定义函数是用户根据需要所编写的函数。从函数定义的形式分为：无参函数、有参函数和空函数。

（1）无参函数

此种函数在被调用时，既无参数输入，也不返回结果给调用函数，只是为完成某种操作而编写的函数。

无参函数的定义形式为：

返回值类型标识符　函数名（）

```
{
    函数体;
}
```

无参函数一般不带返回值，因此函数的返回值类型的标识符可省略。

例如函数：main（），为无参函数，返回值类型的标识符可省略，默认值是int类型。

（2）有参函数

调用此种函数时，必须提供实际的输入函数。有参函数的定义形式为：

返回值类型标识符　函数名（形式参数列表）

形式参数说明

```
{
    函数体;
}
```

【例】定义一个函数max( )，用于求两个数中的大数。

```
int a,b
int max(a, b)
{
        if (a>b) return (a) ;
        else  return (b) ;
}
```

程序段中，a、b为形式参数。return（ ）为返回语句。

（3）空函数

此种函数体内是空白的。调用空函数时，什么工作也不做，不起任何作用。定义空函数的目的，并不是为了执行某种操作，而是为了以后程序功能的扩充。先将一些基本模块的功能函数定义成空函数，占好位置，并写好注释，以后再用一个编好的函数代替它。这样整个程序的结构清晰，可读性好，以后扩充新功能方便。

空函数的定义形式为：

返回值类型标识符 函数名（）

{ }

例如：

```
float min( )
```

3.3.2　函数的参数与返回值

1. 函数的参数

C语言采用函数之间的参数传递方式，使一个函数能对不同的变量进行功能相同的处理，从而大大提高了函数的通用性与灵活性。

函数之间的参数传递，由主函数调用时主调函数的实际参数与被调函数的形式参数之间进行数据传递来实现。

被调用函数的最后结果由被调用函数的return语句返回给调用函数。

函数的参数包括形式参数和实际参数。

（1）形式参数：函数的函数名后面括号中的变量名称为形式参数，简称形参。

（2）实际参数：在函数调用时，主调函数名后面括号中的表达式称实际参数，简称实参。

在C语言的函数调用中，实际参数与形式参数之间的数据传递是单向进行的，只能由实际参数传递给形式参数，而不能由形式参数传递给实际参数。

实际参数与形式参数的类型必须一致，否则会发生类型不匹配的错误。被调用函数的形式参数在函数未调用之前，并不占用实际内存单元。只有当函数调用发生时，被调用函数的形式参数才分配给内存单元，此时内存中调用函数的实际参数和被调用函数的形式参数位于不同的单元。

在调用结束后，形式参数所占有的内存被系统释放，而实际参数所占有的内存单元仍保留并维持原值。

2. 函数的返回值

函数返回值是通过return语句获得的。一个函数可有一个以上的return语句，但是多于一个的return语句必须在选择结构（if或do/case）中使用（例如前面求两个数中的大数函数max( )的例子），因为被调用函数一定只能返回一个变量。

函数返回值的类型由返回值的标识符来指定。例如在函数名之前的int指定函数的返回值的类型为整型数（int）。若没有指定函数的返回值类型，默认返回值为整型类型。

当函数没有返回值时，则使用标识符void进行说明。

3.3.3 函数的调用

在一个函数中需要用到某个函数的功能时，就调用该函数。调用者称为主调函数，被调用者称为被调函数。

1. 函数调用的一般形式

函数调用的一般形式：

函数名（实际参数列表）;

若被调函数是有参函数，则主调函数必须把被调函数所需的参数传递给被调函数。传递给被调函数的数据称为实际参数（简称实参），必须与形参的数据在数量、类型和顺序上都一致。实参可以是常量、变量和表达式。实参对形参的数据是单向的，即只能将实参传递给形参.

2. 函数调用的方式

主调用函数对被调用函数的调用有以下3种方式。

（1）函数调用语句

函数调用语句把被调用函数的函数名作为主调函数的一个语句。例如：

```
print_message( );
```

此时，并不要求函数返回结果数值，只要求函数完成某种操作。

（2）函数结果作为表达式的一个运算对象

函数结果作为表达式的一个运算对象，例如：

```
result=2*gcd(a,b);
```

被调用函数以一个运算对象出现在表达式中。这要求被调用函数带有return语句，以便返回一个明确的数值参加表达式的运算。被调用函数gcd为表达式的一部分，它的返回值乘2再赋给变量result。

（3）函数参数

函数参数即被调用函数作为另一个函数的实际参数。例如：

```
m=max(a,gcd(u,v));
```

其中，gcd(u,v)是一次函数调用，它的值作为另一个函数的max( )的实际参数之一。

3. 对调用函数的说明

在一个函数调另一个函数调用另一个函数时，须具备以下条件：

（1）被调用函数必须是已经存在的函数（库函数或用户自定义的函数）。

（2）如果程序中使用了库函数，或使用了不在同一文件中的另外自定义函数，则应该在程序的开头处使用#include包含语句，将所有的函数信息包含到程序中来。 在程序编译时，系统会自动将函数库中的有关函数调入到程序中去，编译出完整的程序代码。

例如，#include<stdio.h>，将标准的输入、输出头文件stdio.h（在函数库中）包含到程序中来。

（3）如果程序中使用了自定义函数，且该函数与调用它的函数同在一个文件中，则应根据主调用函数与被调用函数在文件中的位置，决定是否对被调用函数作出说明。

a. 如果被调用函数在主调用函数之后，一般应在主调用函数中，在被调用函数调用之前，对被调用函数的返回值类型作出说明。

b. 如果被调用函数出现在主调用函数之前，不用对被调用函数进行说明。

c. 如果在所有函数定义之前，在文件的开头处，在函数的外部已经说明了函数的类型，则在主调用函数中不必对所调用的函数再做返回值类型说明。

3.3.4 中断服务函数

由于标准C没有处理单片机中断的定义，为能进行8051的中断处理，C51编译器对函数定义进行了扩展，增加了一个扩展关键字**interrupt**。

使用interrupt可将一个函数定义成**中断服务函数**。由于C51编译器在编译时对声明为中断服务程序的函数自动添加了相应的现场保护、阻断其他中断、返回时自动恢复现场等处理的程序段，因而在编写中断服务函数时可不必考虑这些问题，减小了用户编写中断服务程序的繁琐程度。

中断服务函数的一般形式为：

函数类型 函数名（形式参数表） interrupt n using n

关键字**interrupt**是中断号，对于51单片机，n取值为0~4。

关键字**using**后的 n是所选择的寄存器组，using是一个选项，可省略。

如果不用关键字using指明寄存器组，中断函数中的所有工作寄存器的内容将被保存到堆栈中。

有关中断服务函数的具体使用注意事项，将在中断系统一章中详细介绍。

3.3.5 变量及存储方式

1. 变量

（1）局部变量

是某一个函数中存在的变量，它只在该函数内部有效。

（2） 全局变量

在**整个源文件中都存在的变量**。有效区间是从**定义点开始到源文件结束**，其中的所有函数都可直接访问该变量。如果定义前的函数需要访问该变量，则需要使用extern关键词对该变量进行说明，如果全局变量声明文件之外的源文件需要访问该变量，也需要使用extern关键词进行说明。

由于全局变量一直存在，占用了大量的内存单元，且加大了程序的耦合性，不利于程序的移植或复用。

全局变量可以使用static关键词进行定义，该变量只能在变量定义的源文件内使用，不能被其他源文件引用，这种全局变量称为静态全局变量。如果一个其他文件的非静态全局变量需要被某文件引用，则需要在该文件调用前使用extern关键词对该变量声明。

2. 变量的存储方式

单片机的存储区间，可以分为程序存储区、静态存储区和动态存储区3个部分。数据存放在静态存储区或动态存储区。其中全局变量存放在静态存储区，在程序开始运行时，给全局变量分配存储空间；局部变量存放在动态存储区，在进入拥有该变量的函数时，给这些变量分配存储间。

3.3.6 宏定义与文件包含

在C51程序设计中要经常用到宏定义、文件包含与条件编译。

1. 宏定义

宏定义语句属于C51语言的预处理指令，使用宏可以使变量书写简化，增加程序的可读性、可维护性和可移植性。宏定义分为简单的宏定义和带参数的宏定义。

（1）简单的宏定义

格式如下：

#define 宏替换名 宏替换体

#define是宏定义指令的关键词，宏替换名一般用大写字母来表示，而宏替换体可以是数值常数、算术表达式、字符和字符串等。宏定义可以出现在程序的任何地方，例如宏定义：

```
#define uchar unsigned char
```

在编译时可由C51编译器把“unsigned char”用“uchar”来替代

例如，在某程序的开头处，进行了3个宏定义：

```
#define uchar unsigned char    /*宏定义无符号字符型变量方便书写*/
```



```
#define uint unsigned int /*宏定义无符号整型变量方便书写*/
#define gain 4    /*宏定义增益*/
```

……

由上见，宏定义不仅可以方便无符号字符型和无符号整型变量的书写，而且当增益需要变化时，只需要修改增益gain的宏替换体4即可，而不必在程序的每处修改，大大增加了程序的可读性和可维护性

（2）带参数的宏定义

格式如下：

#define 宏替换名（形参） 带形参宏替换体

#define是宏定义指令的关键词，宏替换名一般用大写字母来表示，而宏替换体可以是数值常数、算术表达式、字符和字符串等。带参数的宏定义可以出现在程序的任何地方，在编译时可由编译器替换为定义的宏替换体，其中的形参用实际参数代替，由于可以带参数，这就增强了带参数宏定义的应用。

2. 文件包含

是指一个程序文件将另一个指定的文件的内容包含进去。文件包含的一般格式为：

#include <文件名> 或 #include “文件名”

上述两种格式的差别：采用<文件名>格式时，在头文件目录中查找指定文件。采用“文件名”格式时，应在当前的目录中查找指定文件。例如：

```
#include<reg51.h>    /*将特殊功能寄存器包含文件包含到程序中来*/
#include<stdio.h>        /*将标准的输入、输出头文件stdio.h（在函数库中）包含到程序中来*/
#include<stdio.h>/*将函数库中专用数学库的函数包含到程序中来*/
```

当程序中需调用编译器提供的各种库函数时，须在文件的开头使用#include命令将相应函数的说明文件包含进来

3.3.7 库函数

C51语言的强大功能及其高效率在于提供了丰富的可直接调用的库函数。库函数可以使程序代码简单、结构清晰、易于调试和维护。

下面介绍几类重要的库函数。

(1) 特殊功能寄存器包含文件reg51.h或reg52.h。reg51.h中包含所有的8051的sfr及其位定义。reg52.h中包含所有8052的sfr及其位定义，一般系统都包含reg51.h或reg52.h。

(2) 绝对地址包含文件absacc.h：该文件定义了几个宏，以确定各类存储空间的绝对地址。

(3) 输入/输出流函数位于stadio.h文件中。流函数默认8051的串口来作为数据的输入/输出。如果要修改为用户定义的I/O口读写数据，例如，改为LCD显示，可以修改lib目录中的getkey.c及putchar.c源文件，然后在库中替换它们既可。

(4) 动态内存分配函数，位于stdlib.h中。

(5) 能够对方便地对缓冲区进行处理的缓冲区处理函数位于string.h中。其中包括复制、移动、比较等函数。

3.4 Keil μVision3环境下的C51程序开发

Keil C51语言（简称C51语言）是德国Keil software公司开发的用于8051单片机的C51语言开发软件。目前，Keil C51已被完全集成到一个功能强大的全新集成开发环境IDE（Intergrated Development Eviroment）Keil μVision3中。

Keil μVision3 是一款用于8051单片机的集成开发环境，为软件开发提供了全新的C51语言开发环境。它支持众多的8051架构的芯片，同时集编辑、编译、仿真等功能于一体，具有强大的软件调试功能。Keil μVision3增加了很多与8051单片机硬件相关的编译特性，使得应用程序的开发更为方便和快捷，生成的程序代码运行速度快，所需要的存储器空间小，完全可以和汇编语言相媲美，是目前单片机应用开发软件中的最优秀软件开发工具之一。该开发环境下集成了文件编辑处理、编译链接、工程（Project）管理、窗口、工具引用和仿真软件模拟器以及Monitor51硬件目标调试器等多种功能，所有这些功能均可在Keil μVision3的开发环境中极为简便地进行操作。

3.4.1 Keil μVision3的基本操作

1. 软件安装与启动

Keil μVision3集成开发环境的安装，同大多数软件安装一样，根据提示进行。

Keil μVision3安装完毕后，可在桌面上看到Keil μVision3软件的快捷图标。单击桌该快捷图标，即可启动该软件，几秒种后，就会出现如图3-1所示的Keil μVision3界面，图中标出了Keil μVision3界面各窗口的名称。

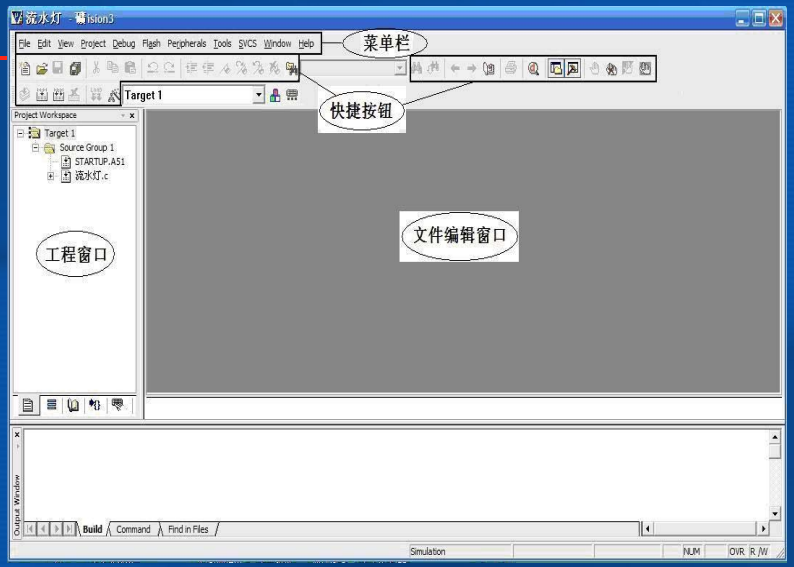


图3-1 Keil软件开发环境界面

2. 创建工程

编写一个新的应用程序前，首先要建立工程（Project）。Keil μVision3把用户的每一个应用程序设计都当作一个工程，用工程管理的方法把一个程序设计中所需要用到的、互相关联的程序链接在同一工程中。这样，打开一个工程时，所需要的关联程序也都跟着进入了调试窗口，方便用户对工程中各个程序的编写、调试和存储。用户也可能开发了多个工程，每个工程用到了相同或不同的程序文件和库文件，采用工程管理，很容易区分不同工程中所用到的程序文件和库文件，非常容易管理。因此，在使用μVision3对程序进行编辑、调试与编译之前，需要首先创建一个新的工程。

在编辑界面下，首先单击“Project”菜单，选择下拉菜单中的“New Project”，弹出文件对话窗口，选择要保存的路径，在“文件名”中输入一个工程的名称，保存后的文件扩展名为“.uv2”，这是Keil μVision3工程文件的扩展名，以后可直接单击此文件就可打开先前建立的工程。

（1）在图3-1所示窗口,单击菜单栏中的【Project】（工程），再点击下拉菜单选项“New Project…”，见图3-2。

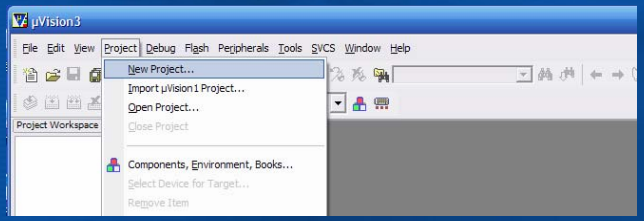


图3-2 新建工程菜单

（2）单击“New Project…”选项后，如图3-3所示，就会弹出“Create New Project”窗口。

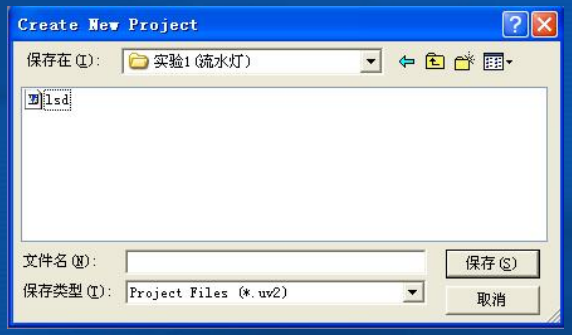


图3-3 “Create New Project”窗口

在该窗口中，需在“文件名（N）”窗口中输入新建工程的名字，并且在“保存在（I）”下拉框中选择工程的保存目录，为工程输入文件名后，单击“保存（S）”即可。

（3）单片机选择，单击“保存（S）”后，会弹出如图3-4所示“Select Device for Target”（选择MCU）窗口，按照界面的提示选择相应的MCU。选择“Atmel”目录下的“AT89C51”(对于AT89S51，也是选择AT89C51)。

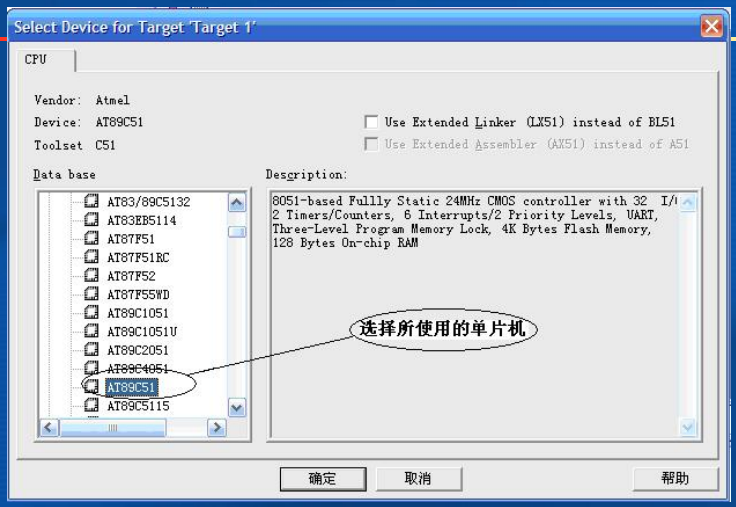


图3-4 “Select Device for Target”窗口

（4）点击“确定”按钮后，会出现图3-5所示的对话框。如果需要复制启动代码到新建的工程，单击“是”，不需要就单击“否”。单击“是”后会出现图3-6的窗口，这时新的工程已经建立完毕。

图3-5 是否复制启动代码到工程对话框

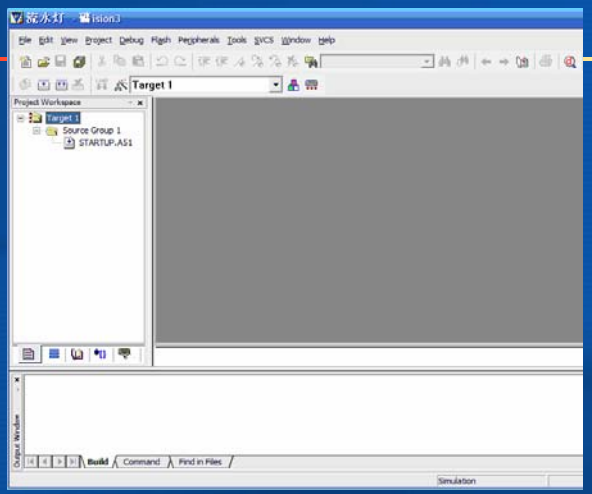


图3-6 完成工程的创建

3.4.2 添加用户源程序文件

在一个新的工程创建完成后，就需要将自己编写的用户源程序代码添加到这个工程中，添加用户程序文件通常有两种方式：一种是新建文件，另一种是添加已创建的文件。

1. 新建文件

（1）单击图3-1中快捷按钮（或单击菜单栏【File】→“New”选项），这时会出现如图3-7所示窗口。在这个窗口会出现一个空白的文件编辑画面，用户可在这里输入编写的程序源代码。

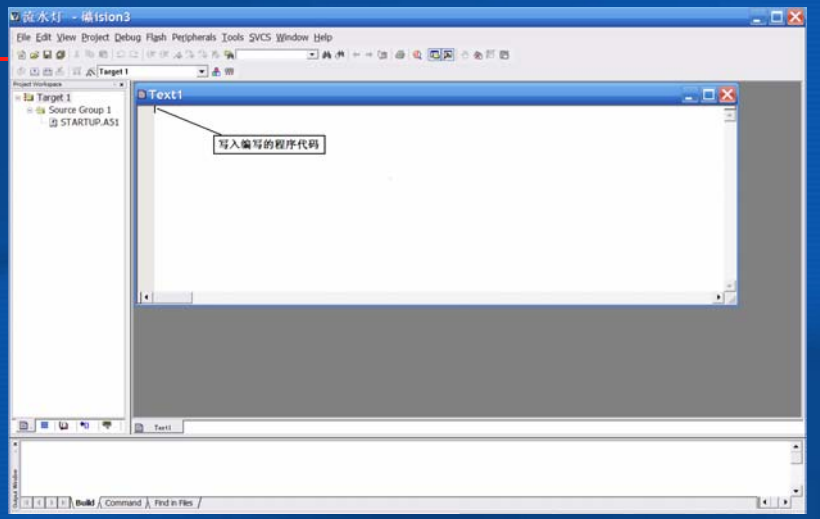


图3-7 建立新文件

（2）单击图3-1中快捷按钮（或单击【File】→“Save”选项），保存文件，这时会弹出如图3-8所示窗口。

（3）在图3-8“Save As”对话框中，在“保存（S）”下拉框中选择新文件的保存目录，这样就将这个新文件与刚才建立的工程保存在同一个文件夹下，然后在“文件名（N）”窗口中输入新建文件的名字，由于使用C51语言编程，则文件名的扩展名应为“.c”，这里我们新建的文件名为“流水灯.c”。如果用汇编语言编程，那么文件名的扩展名应为“.asm”。完成上述步骤后单击“保存”，即可，这时新文件已经创建完成。

如果将这个新文件添加到刚才创建的工程中，操作步骤与下面的“添加已创建文件”步骤相同。

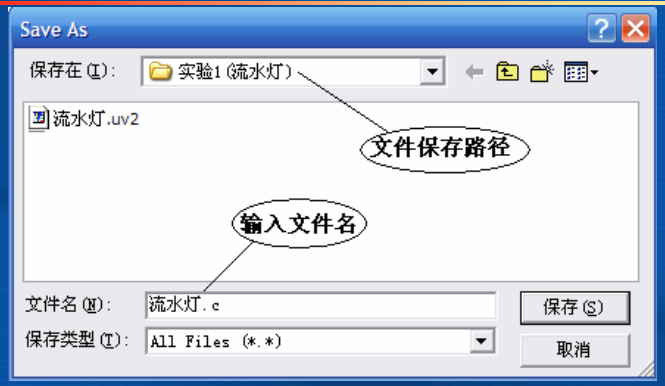


图3-8 “Save As”对话框

2. 添加已创建文件

（1）在工程窗口（图3-1）中，右键单击“Source Group1”，选择“Add File to‘Source Group1’” 选项，见图3-9。

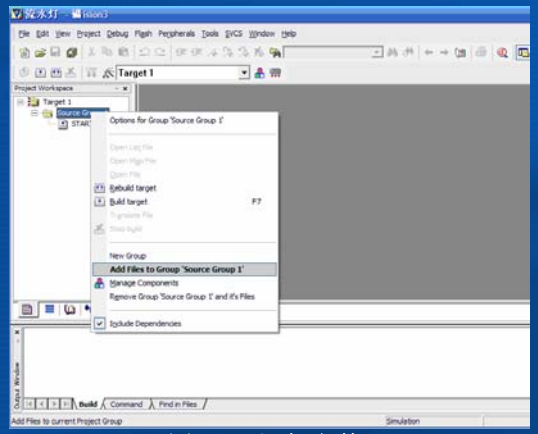


图3-9 添加文件

（2）完成上述操作后会出现如图3-10“Add File to‘Source Group1’”所示的对话框。在该窗口中选择要添加的文件，这里只有刚刚建立的文件“流水灯.c”，点击这个文件后，单击“Add”按钮，再单击“Close”按钮，文件添加已经完成了，这时的工程窗口如图3-11所示，流水灯.c文件已经出现在“Source Group1”目录下了。

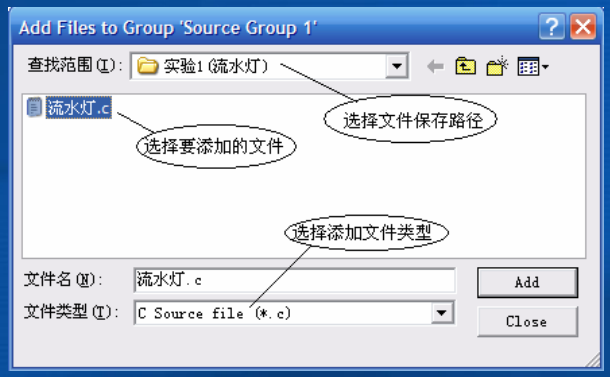


图3-10 “Add File to ‘Source Group1’ ”对话框

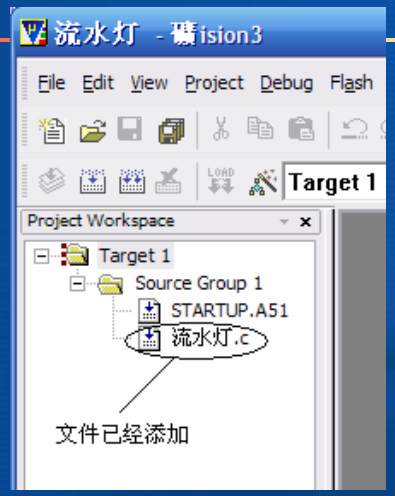


图3-11 文件已添加到工程中

3.4.3 程序的编译与调试

在文件编辑窗口建立了文件“流水灯.c”，并且将文件添加到工程中，然后需将文件编译和调试，最终生成能够执行的.hex文件，步骤如下。

1. 程序编译

单击快捷按钮中的，对当前文件进行编译，在图3-12中的输出窗口会出现提示信息。

从输出窗口中的提示信息可以看到，程序中有2个错误，认真检查程序找到错误并改正，改正后再次单击进行编译，直至提示信息显示没有错误为止，如图3-13所示。

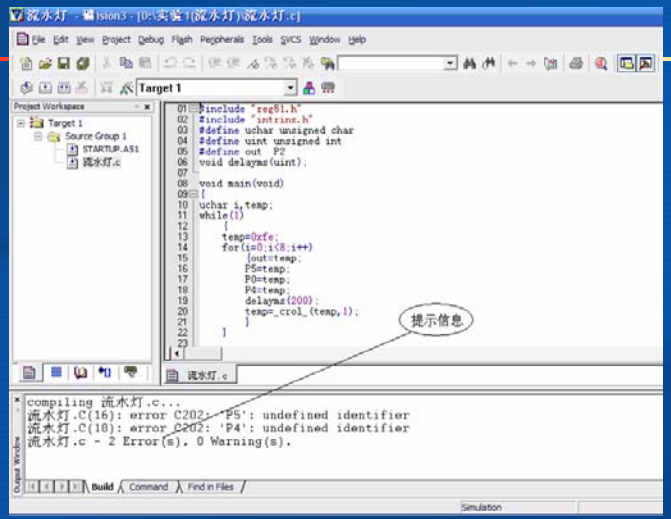


图3-12 文件编译信息

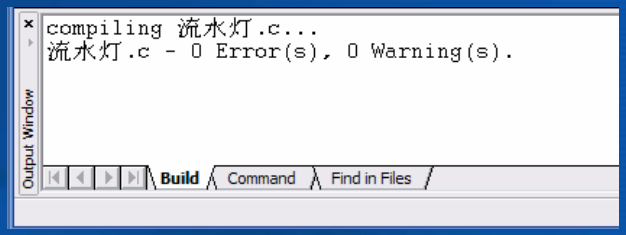


图3-13 提示信息显示没有错误

2. 程序调试

程序编译没有错误后，就可以进行调试与仿真。单击开始/停止调试的快捷按钮（或在主界面点击【Debug】菜单中的“Start/Stop Debug Session”选项），进入程序调试状态，如图3-14所示。

图3-14左面的工程窗口给出了常用的寄存器R0~R7以及A、B、SP、DPTR、PC、PSW等特殊功能寄存器的值，这些值会随着程序的执行发生相应的变化。

在图3-14存储器窗口的地址栏处输入0000H后回车，则可查看单片机片内程序存储器的内容，单元地址前有“C：”，表示程序存储器。如要查看单片机片内数据存储器的内容，在存储器窗口的地址栏处输入D：00H后回车，则可以看到数据存储器的内容。单元地址前有“D：”，表示数据存储器。

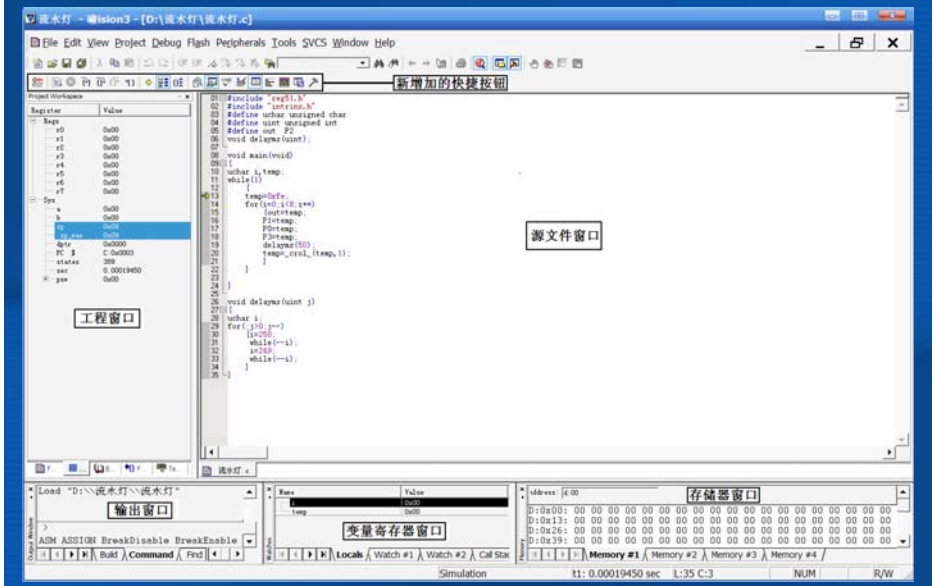


图3-14 程序调试界面

在图3-14中出现了一行新增加的用于调试的快捷命令图标，见图3-15。还有几个原来就有的用于调试的快捷图标，见图3-16。

图3-15 调试状态下的新增加的快捷命令按钮图标

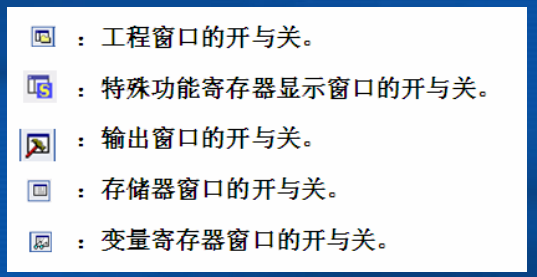
图3-16 用于调试的其他几个快捷命令按钮图标

在程序调试状态下，可运用快捷按钮进行单步、跟踪、断点、全速运行等方式进行调试，也可观察单片机资源的状态，例如程序存储器、数据存储器、特殊功能寄存器、变量寄存器及I/O端口的状态。这些图标大多数是与菜单栏命令【Debug】下拉菜单中的各项子命令是一一对应的，只是快捷按钮图标要比下拉菜单使用起来更加方便快捷。

图3-15与图3-16中常用的快捷按钮图标的功能介绍如下。

（1）各调试窗口显示的开关按钮

下面的图标控制图3-14中各个窗口的开与关。



：调试状态的进入/退出。

：复位 CPU。在程序不改变的情况下，若想使程序重新开始运行，单击本图标命令即可。执行此命令后程序指针返回到 0000H 地址单元。另外，一些内部特殊功能寄存器在复位期间也将重新赋值。例如，A 将变为 00H，SP 变为 07H，DPTR 变为 0000H，P3~P0 口变为 FFH。

：全速运行。单击本图标命令，即可实现全速运行程序。当然若程序中已经设置断点，程序将执行到断点处，并等待调试指令。在全速运行期间，不允许对任何资源进行查看，也不接受其他命令。

：单步跟踪。可以单步跟踪程序。每执行一次此命令，程序将运行一条指令。当前的指令用黄色箭头标出，每执行一步箭头都会移动，已执行过的语句呈绿色。

：单步运行。本命令实现单步运行程序，此时单步运行命令将把函数和函数调用当作一个实体来看待，因此单步运行是以语句(该语句不管是单一命令行还是函数调用)为基本执行单元。

：运行到光标行。

：停止程序运行。

程序调试中，上述的几种运行方式都要用到，灵活地运用这些手段，可大大提高差找差错的效率。

（3）断点操作的快捷按钮

在程序调试中常常要设置断点，一旦执行到该程序行即停止，可在断点处观察有关变量值，以确定问题所在。图 4-17 中有关断点操作的命令快捷按钮的功能如下：

：插入/清除断点。

：清除所有的断点设置。

：使能/禁止断点，是开启或暂停光标所在行的断点功能。

：禁止所有断点，是暂停所有断点。

此外，插入或清除断点最简单的方法，即将鼠标移至需要插入或清除断点的行首双击鼠标即可。

上述的 4 个快捷图标命令，也可从菜单命令【Debug】的下拉子菜单找到。

3.4.4 工程的设置

工程创建后，还需对工程进一步设置。右键单击工程窗口的 “Target 1”，选择“Options for Target‘Target1”，见图3-17，即出现工程设置对话框，见图3-18。该对话框下有多个页面，通常需要设置的有两个，一个是Target页面，另一个是Output页面，其余设置取默认值就可。

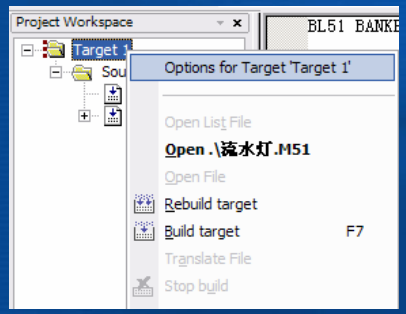


图3-17 工程调试的选择

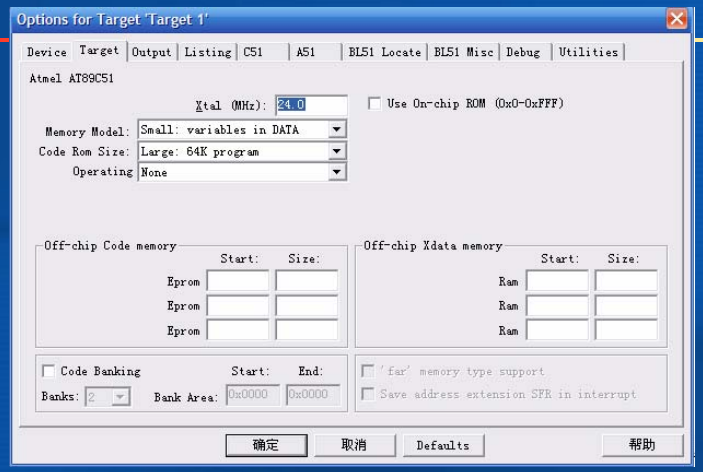


图3-18 “Options for Target‘Target1’”窗口

1.Target页面

（1）Xtal(MHz)—设置晶振频率值，默认值是所选目标CPU的最高可用频率值，可根据需要重新设置。该设置与最终产生的目标代码无关，仅用于软件模拟调试时显示程序执行时间。正确设置该数值可使显示时间与实际所用时间一致，一般将其设置成与硬件目标样机所用的频率相同，如果没必要了解程序执行的时间，也可以不设置。

（2）Memory Model—设置RAM的存储器模式，有3个选项。

① Small—所有变量都在单片机的内部RAM中。

② Compact—可以使用1页外部RAM。

③ Large—可以使用全部外部的扩展RAM。

（3）Code Rom Size—设置ROM空间的使用，即程序的代码存储器模式，有3个选项。

① Small—只使用低于2K的程序空间。

② Compact—单个函数的代码量不超过2K，整个程序可以使用64K程序空间。

③ Large—可以使用全部64K程序空间。

（4）Use on-chip ROM—是否仅使用片内ROM选项。注意，选中该项并不会影响最终生成的目标代码量。

（5）Operation—操作系统选项。Keil提供了两种操作系统：Rtx tiny和Rtx full。通常不选操作系统，所以选用默认项None。

（6）off-chip Cod Memory—用以确定系统扩展的程序存储器的地址范围。

（7）off-chip Xdata Memory—用以确定系统扩展的数据存储器的地址范围。

2. Output页面

点击“Options for Target‘Target1’”窗口的“Output”选项，会出现Output页面，如图3-19所示。

（1）Create HEX File—生成可执行文件代码文件。选择此项后即可生成单片机可以运行的二进制文件（.hex格式文件），文件的扩展名为.hex。

（2）Select Folder for objects—选择最终的目标文件所在的文件夹，默认与工程文件在同一文件夹中，通常选默认。

（3）Name of Executable—用于指定最终生成的目标文件的名字，默认与工程文件相同，通常选默认。

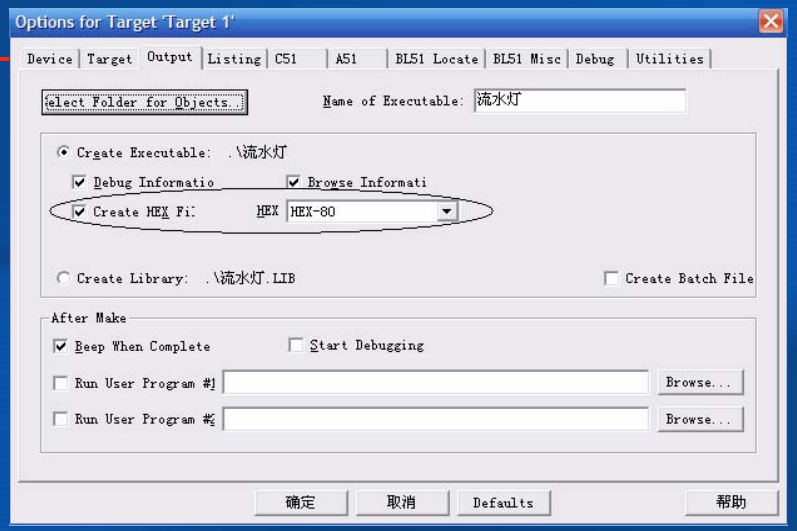


图3-19 Output页面

（4）Debug information—将会产生调试信息，这些信息用于调试，如果需要对程序进行调试，应选中该项。其他选项选默认即可。

完成设置后，就可在程序编译时，单击快捷按钮，此时会产生如图3-20的提示信息。该信息中说明程序占用片内RAM共11字节，片外RAM共0字节，

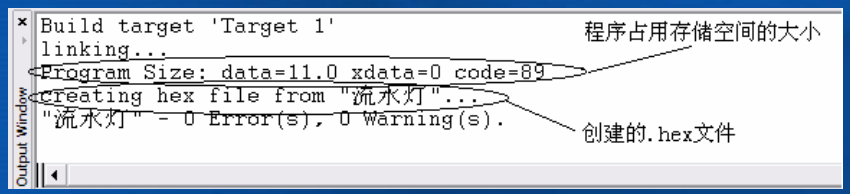


图3-20 hex文件生成的提示信息

占用程序存储器共89字节。最后生成的.hex文件名为“流水灯.hex”，至此，整个程序编译过程就结束了，生成的.hex文件就可在后面介绍的Proteus环境下进行虚拟仿真时，装入单片机运行。

下面对用于编译、连接时的快捷按钮与作简要说明：

（1） “Build target”按钮，即建立工程按钮，用来编译、连接当前工程，产生相应目标文件，如.hex文件。

（2） “Rebuild all target files”按钮，全部重建工程按钮，用于在工程文件有改动时，来全部重建整个工程，并产生相应的目标文件，如.hex文件。

用C51编写的源代码程序不能直接使用，需对该源代码程序编译，生成可执行的目标代码.hex文件，并加载到Proteus环境下的虚拟单片机中，才能进行虚拟仿真。