CodeVisionAVR C Compiler Reference

CodeVisionAVR C 编译器参考

翻译自 CodeVisionAVR C Compiler Help

目录

1、Comments － 注释

注释由 "/*" 开始，"*/" 结束。

例如：

```
/*   单行注释   */
/* 多行
          注释 */
```

单行注释也可以用 "//"。

例如：

```
//单行注释
```

注意：注释不允许嵌套使用。

2、Reserved Keywords － 保留字

下面是编译器的保留字，他们不能被定义为标识符。

break
bit
case
char
const
continue
default
do
double
eeprom
else
enum
extern
flash
float
for
funcused
goto
if
inline
int
interrupt
long
register
return
short
signed
sizeof
sfrb
sfrw
static

struct
switch
typedef
union
unsigned
void
volatile
while

3、Identifiers － 标识符

标识符是你给变量、函数、标号和其它对象起的名字。标识符可以包含字母（A － Z , a － z），数字（0 － 9）和下划线（_）。标识符以字母或下划线开头，区分大小写，最多不得超过 32 个字符。

4、Data Types － 数据类型

位型（bit）只能定义为全局变量。

数据类型、长度（位）、范围如下表：

Type － 类型	大小（位）－ Size (Bits)	范围 － Range
位型 － bit	1	0，1
字符型 － char	8	－128，127
无符号字符型 － unsigned char	8	0，255
有符号字符型 － signed char	8	－128，127
整型 － int	16	－32768，32767
短整型 － short int	16	－32768，32767
无符号整型 － unsigned int	16	0，65535
有符号整型 － signed int	16	－32768，32767
长整型 － long int	32	-2147483648，2147483647
无符号长整型 － unsigned long int	32	0，4294967295
有符号长整型 － signed long int	32	-2147483648，2147483647
浮点型 － float	32	?.175e-38 ，?.402e38
双精度型 － double	32	?.175e-38 ，?.402e38

如果 Project|Configure|C Compiler|char is unsigned 选项被选中或使用了#pragma uchar+ ，那么 char 的范围为 0 －255 。

5、Constants － 常量

整数常量和长整数常量可以表示为十进制（1243）、二进制（0b 作前缀，0b10010011）、十六进制（0x 作前缀，0x55）、八进制（0 作前缀，077）。

无符号整数常量加后缀 U 表示（10000U）。

长整数常量加后缀 L 表示（99L）。

无符号长整数常量加后缀 UL 表示（99UL）。

浮点数常量加后缀 UL 表示（1.234F）。

字符常量用单引号表示（'a'）。

字符串常量用双引号表示（"Hello world"）。

如果你把一个字符串作为一个函数的参数来引用，这个字符串将自动被当做常量使用，并且会被放在 FLASH 中。

例子：

```
//这是一个显示 RAM 中字符串的函数
void display_ram(char *s) {
/* .......   */
//这是一个显示 FLASH 中字符串的函数
void display_flash(char flash *s) {
/* .......   */
void main(void) {
display_ram("Hello world"); //不工作，因为寻址的字符串在 FLASH 中
display_flash("Hello world");//工作正常
```

常量可以被定义成数组，最多八维。

常量存在 FLASH 中，所以你必须使用关键字 flash 或 const。

常量表达式在编译时自动求解。

例如：

```
flash int    integer_constant=1234+5;
flash char char_constant=d
flash long long_int_constant1=99L;
flash long long_int_constant2=0x10000000;
flash int    integer_array1[]={1,2,3};
flash int    integer_array2[10]={1,2};//前两个为 1，2，其余的为 0
flash int    multidim_array[2,3]={{1,2,3},{4,5,6}};
flash char string_constant1[]=This is a string constant
const char string_constant2[]=This is also a string constant
```

常量可以在函数内部声明。

6、Variables － 变量

变量可声明为全局变量（可被所有函数访问）和局部变量（只能在被声明的函数内部访问）。

如果全局变量没有赋初值，在程序开始时会被自动赋值为 0。局部变量不会自动赋值。

语法如下：

```
[<storage modifier>] <type definition> <identifier>;
[<存储模式>] <类型定义> <标识符>;
```

例子：

```
/*  全局变量  */
char a;
int    b;
/*  赋初值  */
long c=1111111;
void main(void) {
/*  局部变量  */
```

```
char d;
int   e;
/*赋初值*/
long f=22222222;
```

变量也可以组成最多达八维的数组，第一个数组元素编号为 0。
如果全局变量数组没有赋初值，在程序开始时会被自动赋值为 0。
例子：

```
int   global_array1[32];// 所有元素自动赋值为 0
int   global_array2[]={1,2,3};//数组赋初值
int   global_array3[4]={1,2,3,4};
char global_array4[]=This is a string
int   global_array5[32]={1,2,3};//前三个赋初值，其余 29 个自动赋值为 0
int multidim_array[2,3]={{1,2,3},{4,5,6}};//多维数组
void main(void) {
int   local_array1[10];//局部变量数组
int   local_array2[3]={11,22,33};//局部变量数组赋初值
char local_array3[7]="Hello";
```

被不同的函数调用的必须保存其值的局部变量必须被声明为静态变量 － static。
例子：

```
int alfa(void) {
static int n=1;//声明为静态变量
return n++;
void main(void) {
int i;
i=alfa();//返回值为 1
i=alfa();//返回值为 2
```

如果静态变量没有赋初值，在程序开始时会被自动赋值为 0。
如果变量在其它的文件中声明，则必须使用关键字 extern 。
例子：

```
extern int xyz;
#include <file_xyz.h>//包含声明 xyz 的文件
```

为了告诉编译器给某个变量分配寄存器，必须使用 register 修饰符。
例子：

```
register int abc;
```

编译器会给变量分配一个寄存器，即使这个被修饰的变量没有使用。
为了防止把一个变量分配在寄存器，必须使用 volatile 修饰符，并且通知编译器这个变量的赋值受外部变化的支配。
例子：

```
volatile int abc;
```

所有没被分配到寄存器的全局变量存放在 SRAM 的全局变量区。

所有没被分配到寄存器的局部变量动态地存放在 SRAM 的数据堆栈区。

7、Specifying the SRAM Storage Address for Global Variables － 指定全局变量在 SRAM 的地址

在设计时可以使用@操作符指定全局变量在 SRAM 的地址。

例子：

```
int a @0x80;// 整型变量 a 存在 SRAM 地址 80h
struct x {
        int a;
        char c;
        } alfa @0x90;// 结构 alfa 存在 SRAM 地址 90h
```

8、Bit Variables － 位变量

位变量是存储在寄存器 R2 到 R15 的特殊全局变量，用关键字 bit 声明。

语法：

```
bit <标识符>;
```

例子：

```
/* 声明和赋初值 */
bit alfa=1; /* bit0 of R2 */
bit beta; /* bit1 of R2 */
void main(void)
if (alfa) beta=!beta;
/* ..... */
```

根据声明的顺序，位变量的分配从 R2 的 bit 0 开始，然后 bit 1，等等，按升序排列。

最多可声明 112 个位变量。

给位变量分配的空间可以在 Project|Configure|C Compiler|Compilation|Bit Variables Size 指定。为了给其它的全局变量分配寄存器，给位变量分配的空间要尽可能的小。

如果位变量没有赋初值，在程序开始时会自动赋值为 0。

在表达式赋值中位变量自动转变为无符号字符型。

9、Allocation of Variables to Registers － 给变量分配寄存器

为了充分利用 AVR 的构架和指令，编译器把一些变量分配在寄存器中。寄存器 R2 到 R15 可以分配给位变量。

给位变量分配的空间可以在 Project|Configure|C Compiler|Compilation|Bit Variables Size 指定。为了给其它的全局变量分配寄存器，给位变量分配的空间要尽可能的小。

如果选择了 Project|Configure|C Compiler|Compilation|Automatic Register Allocation 选项或使用了 #pragma regalloc+ 编译指示，在 R2 到 R15 中没被位变量使用的剩下的寄存器将分配给字符型和整形全局变量，直到 R15 也分配完。

如果自动寄存器分配被禁止，你可以用关键字 register 指定那个全局变量分配在寄存器。

例子：

```
#pragma regalloc-//禁止自动寄存器分配
```

register int alfa;//分配变量 alfa 到一个寄存器
register int beta @14;// 分配变量 beta 到寄存器对 R14，R15

字符型和整型局部变量根据声明的先后分配在 R16－R21。

10、Structures － 结构体

结构体是用户定义的已命名的成员的集合。
结构体的成员可以是任何类型的数据、数组、指针。
结构用关键字 struct 定义。
语法：

```
[<存储修饰符>] struct [<结构体名称>] {
        [<类型> <成员名[, 成员名, ...]>];
        [<类型> <成员名[, 成员名, ...]>];
              .....
                    } [<结构体变量表列>];
```

例子：

```
/* SRAM 中的全局结构体变量 */
struct ram_structure {
                    char a,b;
                    int   c;
                    char d[30],e[10];
                    char *pp;
                    } sr;
/* FLASH 中的全局结构体常量 */
flash struct flash_structure {
                    int   a;
                    char b[30], c[10];
                    } sf;
/* EEPROM 中的全局变量结构 */
eeprom struct eeprom_structure {
                    char a;
                    int   b;
                    char c[15];
                    } se;
void main(void) {
/* 局部结构体变量 */
struct local_structure {
                    char a;
                    int   b;
                    long c;
                    } sl;
/* ..... */
}
```

结构体在存储器中占用的空间等于它的成员占用的空间的总和。

不过存储在 FLASH 和 EEPROM 中的结构体有一些限制。

由于指针只能在 SRAM 中，所以指针不能用 FLASH 和 EEPROM 中的结构体。

在 FLASH 中由于 AVRASM32 在汇编时对于用 .DB 定义的单个字节的数据要占用两个字节，所以 CodeVisionAVR C 编译器会将结构中的字符型成员用整型代替。同样，这会将结构体中的字符型数组成员的空间大小扩为偶数个字节。

结构体不能组成多维。

下面的例子给出了怎样初始化和访问在 EEPROM 中的全局结构体。

```
/* 定义在 EEPROM 中的全局结构体 */
eeprom struct eeprom_structure {
                char a;
                int  b;
                char c[15];
                } se[2]={{'a',25,"Hello"},{'b',50,"world"}};
void main(void) {
char k1,k2,k3,k4;
int i1, i2;
/* 定义一个指向结构体的指针 */
struct eeprom_structure eeprom *ep;
/* 直接读取结构体成员 */
k1=se[0].a;
i1=se[0].b;
k2=se[0].c[2];
k3=se[1].a;
i2=se[1].b;
k4=se[1].c[2];
/* 用指针作同样的读取 */
ep=&se; /* 初始化指针地址 */
k1=ep->a;
i1=ep->b;
k2=ep->c[2];
++ep;   /* 指针加一 */
k3=ep->a;
i2=ep->b;
k4=ep->c[2];
}
```

由于一些 AVR 芯片的 SRAM 较小，为了保持数据堆栈的空间较小，结构体不能直接作为函数的参数。结构体只能使用指针传给函数或用函数返回。

例子：

```
struct alpha {
                int a,b, c;
                } s={2,3};
/* 定义一个函数 */
struct alpha *sum_struct(struct alpha *sp) {
/* 成员 c=成员 a + 成员 b */
```

```
sp->c=sp->a + sp->b;
/* 返回一个指向结构体的指针 */
return sp;
}
void main(void) {
int i;
/* s->c=s->a + s->b */
/* i=s->c */
i=sum_struct(&s)->c;
}
```

结构体中不能用位类型的成员，所以位的存储只能用位变量。

11、Unions － 联合（也称为共用体）

联合是指几个不同的变量共占用同一存储空间的结构。

联合的成员可以是任何类型的数据、数组、指针。

联合用关键字 unions 定义。

语法：

```
[<存储修饰符>] unions [<联合名称>] {
        [<类型> <成员名[, 成员名, ...]>];
        [<类型> <成员名[, 成员名, ...]>];
      ......
        } [<联合变量表列>];
```

联合通常存储在 SRAM 中。

联合在存储器中占用的空间等于占用空间最大的成员所占用的空间。

联合成员的访问与结构成员的访问的方式相同。

例子：

```
/* 声明一个联合 */
union alpha {
        unsigned char lsb;
        unsigned int  word;
        } data;
void main(void) {
unsigned char k;
/* 定义一个指向联合的指针 */
union alpha *dp;
/* 直接读取联合的成员 */
data.word=0x1234;
k=data.lsb; /* 得到 LSB 为 0x1234 */
/* 通过指针作同样的读取 */
dp=&data;  /* 初始化指针地址 */
dp->word=0x1234;
k=dp->lsb; /* 得到 LSB 为 0x1234 */
}
```

由于一些 AVR 芯片的 SRAM 较小，为了保持数据堆栈的空间较小，联合不能直接作为函数的参数。

联合只能使用指针传给函数或用函数返回。

例子：

```
#include <stdio.h> /* 有 printf 函数 */
union alpha {
                unsigned char lsb;
                unsigned int    word;
                } data;
/* 定义一个函数 */
unsigned char low(union alpha *up) {
/* 返回字的低字节 */
return up->lsb;
}
void main(void) {
data.word=0x1234;
printf("the LSB of %x is %2x",data.word,low(&data));
}
```

12、Enumerations － 枚举

枚举是将变量的值一一列出，变量的值只限于列出的值的范围内。标识符是为了便于记忆，可以自己定义。

使用关键字 enum 声明。

语法：

```
[<存储修饰符>] enum [<枚举名称>] {
        [<元素名称[[=初始值], 元素名称, ...]>]}
        [<枚举变量表列>];
```

例子：

```
/* 枚举类型元素这样定义：编译器会按定义的顺序使它们的值为 0，1，2.....
sunday=0 , monday=1 , tuesday=2 ,..., saturday=6 */
enum days {
        sunday, monday, tuesday, wednesday,
          thursday, friday, saturday} days_of_week;
/*   枚举类型元素这样定义：编译器会按定义的顺序使它们的值顺序加一
january=1 , february=2 , march=3 ,..., december=12 */
 enum months {
  january=1, february, march, april, may, june,
  july, august, september, october, november, december}
  months_of_year;
 void main {
 /* 变量 days_of_week 初始化为 6 */
 days_of_week=saturday;
 }
```

枚举类型可以存储在 SRAM 或 EEPROM。

为了指定其存储在 EEPROM，必须使用 eeprom 关键字。

例子：

```
eeprom enum days {
                    sunday, monday, tuesday, wednesday,
                    thursday, friday, saturday} days_of_week;
```

13、Global Variables Memory Map File － 全局变量存储器分配映象文件

在编译过程中，编译器会产生全局变量存储器分配映象文件。文件中包含了程序中详细的 SRAM 地址分配，寄存器分配和全局变量所占存存储器大小的信息。

文件使用.map 后缀，可以用菜单中的 File|Open 命令或按工具条的 Open 按钮打开观看。

结构体和联合的成员与它们相应的地址和大小分别列出。

这个文件在用 AVR Studio debugger 进行程序调试时非常有用。

14、Type Conversions － 类型转换

在一个表达式中如果有两个操作数的类型不同，编译器会将其转换为同一类型的数据。

转换遵循以下原则：

如果有两个数有一个是浮点型，另一个操作数要转换成浮点型。

如果有两个数有一个是长整型或无符号长整型，另一个操作数要转换成相同的类型。

如果有两个数有一个是整型或无符号整型，另一个操作数要转换成相同的类型。

字符型和无符号字符型优先级最低。

可以使用强制类型转换运算符转换类型。

例子：

```
void main(void) {
int   a, c;
long b;
/* 长整型变量 b 被视为整型 */
c=a+(int) b;
}
```

注意：如果没有选择 Project|Configure|C Compiler|Promote char to int 选项或没有使用 #pragma promotechar+ ，字符型（无符号字符型）不会在对 16 位或 32 位 CPU 编译时自动转变为整型（无符号整型）。这使象 AVR 这种 8 位 CPU 可以写出更快更小的代码。

为了防止在 8 位加法或乘法时产生溢出，有必要进行强制类型转换。这时编译器会给出 warning。

例子：

```
void main(void) {
unsigned char a=30;
unsigned char b=128;
unsigned int c;
/* 这样会产生不正确的结果，由于乘法运算是 8 位乘 8 位的，它的结果也是 8 位，这就产生溢出。
只有在乘法后把 8 位的结果转位无符号整型。 */
c=a*b;
/* 这里使用强制类型转换，把乘法变为 16 位，其结果也是 16 位，这样就不会溢出。*/
c=(unsigned int) a*b;
}
```

对于下面这些运算符，编译器的处理会有所不同：

+=

-=
*=
/=
%=
&=
|=
^=
<<=
>>=

由于这些运算符运算后的结果要写回到左边的操作数（必须是变量）。所以编译器总是把右边的操作数转换为左边的类型。

15、Operators － 运算符

编译器支持以下运算符：

+	-
*	/
%	++
--	=
==	~
!	!=
<	>
<=	>=
&	&&
|	||
^	?
<<	>>
-=	+=
/=	%=
&=	*=
^=	|=
>>=	<<=

16、Functions － 函数

你可以使用函数的本身声明一个函数。声明包含了函数参数的信息。

例子：

```
int alfa(char par1, int par2, long par3);
```

实际的函数定义可以这样写：

```
int alfa(char par1, int par2, long par3) {
/* 这里可以写一些说明*/
}
```

老式的 Kernighan & Ritchie 形式的函数定义不再支持。

函数参数通过数据堆栈传递。

函数的返回值放在寄存器 R30、R31、R22 和 R23（从 LSB 到 MSB）。

17、Pointers － 指针

基于 AVR 单片机的 Harvard 结构，数据（SRAM）、程序（FLASH）和 EEPROM 的地址是分开的，编译器有三种类型的指针。

变量存放在 SRAM，使用通常的指针。

常量存放在 FLASH，要使用 flash 关键字。

存放在 EEPROM 的变量，要使用 eeprom 关键字。

虽然指针指向不同的存储区域，但它们都放在 SRAM 中。

例子：

```
/* 指向 SRAM 中的字符串的指针  */
char *ptr_to_ram="This string is placed in SRAM";
/*  指向 FLASH 中的字符串的指针  */
char flash *ptr_to_flash="This string is placed in FLASH";
/*  指向 EEPROM 中的字符串的指针  */
char eeprom *ptr_to_eeprom="This string is placed in EEPROM";
```

为了提高代码效率，有两种内存编译模式可以选择。

TINY 模式，SRAM 中的变量使用 8 位的存储指针。这种模式下，你只能访问 SRAM 的前 256 字节。

SMALL 模式，SRAM 中的变量使用 16 位的存储指针。这种模式下，你可以访问 SRAM 的 65536 字节。

为提高运行速度、减小代码体积，要尽量使用 TINY 模式。

FLASH 和 EEPROM 的存储指针为 16 位。

由于 FLASH 的存储指针为 16 位，所以 ATmega103 的二进制代码不能超过 64K。

指针可以组成最多到八维的数组。

例子：

```
/* 声明和初始化在 SRAM 中的字符串的全局指针数组 */
char *strings[3]={"One","Two","Three"};
/* 声明和初始化在 FLASH 中的字符串的全局指针数组。注意：字符串存在 FLASH，但指针数组本身存在 SRAM 中 */
char flash *messages[3]={"Message 1","Message 2","Message 3"};
/* 声明 EEPROM 中的字符串 */
eeprom char m1[]="aaaa";
eeprom char m2[]="bbbb";
void main(void) {
/* 声明和初始化在 EEPROM 中的字符串的局部指针数组。注意：字符串存在 FLASH，但指针数组本身存在 SRAM 中 */
char eeprom *pp[2];
/* 初始化数组元素 */
pp[0]=m1;
pp[1]=m2;
}
```

指向函数的指针也是 16 位，因为是指向 FLASH 区。但是并不需要 flash 关键字。

Example:

```
/* 声明一个函数 */
```

```
int sum(int a, int b) {
return a+b;
}
/* 声明并初始化一个指向函数 sum 的全局指针 */
int (*sum_ptr) (int a, int b)=sum;
void main(void) {
int i;
/* 使用指针调用函数 sum */
i=(*sum_ptr) (1,2);
}
```

18、Accessing the I/O Registers － 访问 I/O 寄存器

编译器使用 sfrb 和 sfrw 关键字访问 AVR 单片机的 I/O 寄存器。它使用 IN 和 OUT 汇编指令。

例子：

```
/* 定义 SFRs */
sfrb PINA=0x19;   /*   SFR 的 8 位访问 */
sfrw TCNT1=0x2c; /* SFR 的 16 位访问 */
void main(void) {
unsigned char a;
a=PINA;           /* 读 A 口的引脚值 */
TCNT1=0x1111; /* 写寄存器 TCNT1L & TCNT1H */
}
```

I/O 寄存器的地址已经定义在..\INC 目录下的下列的头文件里：

```
tiny22.h
90s2313.h
90s2323.h
90s2333.h
90s2343.h
90s4414.h
90s4433.h
90s4434.h
90s8515.h
90s8534.h
90s8535.h
mega603.h
mega103.h
mega161.h
mega163.h
mega32.h
94k.h
```

在你的程序的开始，可以 #include 与你使用的单片机所对应的文件。

19、Bit level access to the I/O Registers － I/O 寄存器的位访问

I/O 寄存器的位访问使用 I/O 寄存器名称后的位。

由于 I/O 寄存器的位访问使用 CBI, SBI, SBIC 和 SBIS 指令，所以 sfrb 寄存器的地址要在 0 到 1Fh 之间，sfrw 寄存器的地址要在 0 到 1Eh 之间。

例子：

```
sfrb PORTA=0x1b;
sfrb DDRA=0x18;
sfrb PINA=0x19;
void main(void) {
/* 置 A 口的 0 位为输出 */
DDRA.0=1;
/* 置 A 口的 1 位为输入 */
DDRA.1=0;
/* 在 A 口的 0 为输出 1 */
PORTA.0=1;
/* 检测 A 口的 1 位的输入 */
if (PINA.1) { /* 放一些测试代码 */ };
/* ..... */
}
```

你可以使用 #define 给 I/O 寄存器定义标号：

```
sfrb PINA=0x19;
#define alarm_input PINA.2
void main(void)
{
/* 检测 A 口的 2 位的输入 */
if (alarm_input) { /* 放一些测试代码 */ };
/* ..... */
}
```

20、Accessing the EEPROM － 访问 EEPROM

使用前面加 eeprom 关键字全局变量就能完成对 AVR 内部的 EEPROM 的访问。

例子：

```
/* 对芯片编程时 1 写入 EEPROM */
eeprom int alfa=1;
eeprom char beta;
eeprom long array1[5];
/* 对芯片编程时字符串写入 EEPROM */
eeprom char string[]="Hello";
void main(void) {
int i;
/* 指向 EEPROM 的指针 */
int eeprom *ptr_to_eeprom;
/* 把 0x55 直接写入 EEPROM */
alfa=0x55;
/* 或使用指针间接写入 */
ptr_to_eeprom=&alfa;
```

```
*ptr_to_eeprom=0x55;
/* 直接从 EEPROM 中读出  */
i=alfa;
/* 或使用指针间接读出 */
i=*ptr_to_eeprom;
}
```

指向 EEPROM 的指针为 16 位。

21、Using Interrupts － 使用中断

访问 AVR 的中断系统只需使用 interrupt 关键字。

例子：

```
/*  AT90S8515 的中断向量号 */
/* 有外部中断时自动调用 */
interrupt [2] void external_int0(void) {
/* 这里放你的代码 */
}
/*  TIMER0 溢出时自动调用 */
interrupt [8] void timer0_overflow(void) {
/* 这里放你的代码  */
}
```

中断向量号从 1 开始。

编译器在中断调用时会自动保存所有使用的寄存器，并在中断返回时恢复它们的值。

中断结束的地方是一条 RETI 指令。

中断没有返回值，也不能带参数。

你必须设置相关的控制寄存器中的位以允许中断。

你可以使用 #pragma savereg 指示符打开或关闭中断对 R0, R1, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31 和 SREG 的自动保存和恢复。

例子：

```
/* 关闭寄存器自动保存和恢复 */
#pragma savereg-
/* 中断句柄 */
interrupt [1] void my_irq(void) {
/* 只保存使用的寄存器 R30, R31 和 SREG */
#asm
    push r30
    push r31
    in   r30,SREG
    push r30
#endasm
/* .... */
/* 恢复 SREG, R31 和 R30 */
#asm
    pop r30
```

```
    out SREG,r30
    pop r31
    pop r30
#endasm
}
/* 打开寄存器自动保存和恢复 */
#pragma savereg+
```

缺省设置是打开寄存器自动保存和恢复。

22、The Preprocessor － 预编译

宏可以有参数。预编译会按照宏的展开替换宏，并用真正的实参代替宏的形参。

例子：

```
#define SUM(a,b) a+b
/* 下面的代码会被 int i=2+3;代替 */
int i=SUM(2,3);
```

当定义一个宏时，你可以使用操作符“#”把宏的参数转变为字符串。

例子：

```
#define PRINT_MESSAGE(t)    printf(#t)
/* ..... */
/* 下面的代码会被 printf("Hello");代替 */
PRINT_MESSAGE(Hello);
```

可以用“##”连接两个参数。

例子：

```
#define ALFA(a,b) a ## b
/* 下面的代码会被 char xy=1;代替 */
char ALFA(x,y)=1;
```

定义宏时可以用“\”换行

例子：

```
#define MESSAGE "This is a very \
long text..."
```

一个宏可以用 #undef 终止。

例子：

```
#undef ALFA
```

#ifdef , #ifndef , #else 和 #endif 指示符用于条件编译。

语法：

```
#ifdef macro_name
[set of statements 1]
#else
```

[set of statements 2]
#endif

如果'alfa'是一个已定义的宏的名字，那么若#ifdef 的值为真，那么就编译第一部分。否则编译第二部分。根据具体情况 #else 和第二部分也可以不要。

如果'alfa'没有定义，则 #ifndef 的值为真，情况与 #ifdef 相同。

#if , #else 和 #endif 也可以用于条件编译。

语法：

#if expression1
[set of statements 1]
#else
[set of statements 2]
#endif

如果表达式 1 的值为真，就编译第一部分。否则编译第二部分。

#elif 对条件编译也很有用：

#if expression1
[set of statements 1]
#elif expression1
[set of statements 2]
#else
[set of statements 3]
#endif

如果表达式 1 的值为真，就编译第一部分。如果表达式 2 的值为真，就编译第二部分。否则编译第三部分。根据具体情况 #else 和第三部分也可以不要。

下面是系统已经预定义的宏：

__LINE__ 编译文件的当前行号
__FILE__ 当前编译的文件
__TIME__ 当前时间，hh:mm:ss 格式
__DATE__ 当前日期， mmm dd yyyy 格式
_MODEL_TINY_ 是否在编译时使用了 TINY 模式
_MODEL_SMALL_ 是否在编译时使用了 SMALL 模式
_OPTIMIZE_SIZE_ 是否在编译时对程序空间大小优化
_OPTIMIZE_SPEED_是否 在编译时对程序运行速度优化
_UNSIGNED_CHAR_ 是否在编译时将 char 当做无符号字符型编译
_GLOBAL_DEFINES_ 是否使宏在所有的目标文件中可见

使用 #line 指示符可以修改__LINE__ 和 __FILE__ 的预定义值。

语法：

#line integer_constant ["file_name"]

例子：

```
/* 设 __LINE__ 为 50 ， __FILE__ 为 "file2.c" */
#line 50 "file2.c"
/* 设 __LINE__ 为 100 */
#line 100
```

使用 #error 指示符可以停止编译并显示一条错误信息。
语法：

```
#error error_message
```

例子：

```
#error This is an error!
```

使用#pragma 指示符可以改变编译选择编译选项。
你可以使用 #pragma warn 指示符打开或关闭编译警告。
例子：

```
/* 关闭警告 */
#pragma warn-
/* 可以在这里写你的代码 */
/* 打开警告 */
#pragma warn+
```

使用 #pragma opt 指示符可以打开或关闭编译代码优化。这个指示符只能放在源文件的开始处。缺省状态是打开代码优化。
例子：

```
/* 关闭代码优化，测试时使用 */
#pragma opt-
/* 打开代码优化 */
#pragma opt+
```

如果打开了代码优化，你可以使用 #pragma optsize 指示符对程序的一部分或全部进行代码大小优化或代码执行速度优化。缺省状态由 Project|Configure|C Compiler|Compilation|Optimization for 来设置。
例子：

```
/* 对代码大小优化 */
#pragma optsize+
/* 你的代码 */
/* 对代码执行速度优化 */
#pragma optsize-
/* 你的代码 */
```

使用 #pragma savereg 指示符可以打开或关闭中断时对 R0, R1, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31 和 SREG 的自动保存与恢复。
例子：

```
/* 关闭自动保存 */
#pragma savereg-
/* 中断句柄 */
```

```
interrupt [1] void my_irq(void) {
/* 只保存你使用到的寄存器 R30, R31 和 SREG */
#asm
    push r30
    push r31
    in   r30,SREG
    push r30
#endasm
/* 你的 C 代码 */
/* .... */
/* 恢复 SREG, R31 和 R30 */
#asm
    pop r30
    out SREG,r30
    pop r31
    pop r30
#endasm
}
/* 打开自动保存 */
#pragma savereg+
```

缺省状态是打开自动保存。
使用 #pragma regalloc 指示符可以打开或关闭自动分配寄存器给全局变量这一选项。
缺省状态由 Project|Configure|C Compiler|Compilation|Automatic Register Allocation 决定。
例子：

```
/* 下面的全局变量将自动分配在寄存器中 */
#pragma regalloc+
unsigned char alfa;
/* 下面的全局变量将不会自动分配在寄存器中而是放在 SRAM */
#pragma regalloc-
unsigned char beta;
```

使用 #pragma promotechar 指示符可以打开或关闭字符型转换为整型。
例子：

```
/* 打开转换 */
#pragma promotechar+
/* 关闭转换 */
#pragma promotechar-
```

这个选项也可以用 Project|Configure|C Compiler|Promote char to int 设定。

Treating char by default as an unsigned 8 bit can be turned on or off using the
使用 #pragma uchar 指示符可以打开和关闭将字符型看做 8 位无符号数选项。
例子：

```
/* char 为无符号数 */
#pragma uchar+
/* char 为有符号数 */
#pragma uchar-
```

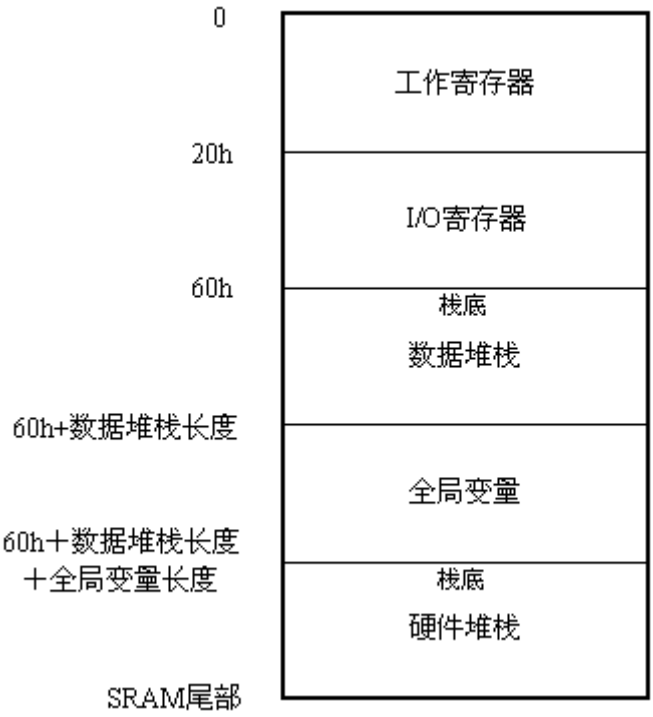
这个选项也可以用 Project|Configure|C Compiler|char is unsigned 设定

使用 #pragma library 指示符可以编译或连接指定的库文件。
例子：

```
#pragma library mylib.lib
```

23、SRAM Memory Organization － SRAM 结构

下面是编译完成后的 SRAM 结构图：



工作寄存器区包括 32 个 8 位寄存器。编译器使用了其中的 R0, R1, R22, R23, R24, R25, R26, R27, R28, R29, R30 和 R31。 R2 到 R15 中的一些寄存器可能会用来存放全局位变量。剩下的寄存器会分配给全局字符型变量和全局整型变量。 R16 到 R21 分配给局部字符型变量和局部整型变量。

I/O 寄存器区包括 64 个 CPU 外围功能的地址，有端口控制寄存器、定时器和其它 I/O 功能。你可以在程序中自由地使用它们。

数据堆栈区用于动态存储局部变量，传递函数参数和中断期间保存 R0, R1, R22, R23, R24, R25, R26, R27, R30, R31 和 SREG。

数据堆栈区指针使用 Y 寄存器。

数据堆栈指针的处始值为 5Fh＋数据堆栈长度。

在数据堆栈保存数据时，数据堆栈指针是递减的。数据取出时，数据堆栈指针是递增的。

在配置编译选项时，你必须在 Project|Configure|C Compiler 中为数据堆栈分配足够的空间，以免导致程序运行时数据堆栈于 I/O 寄存器区重叠。

全局变量区在程序运行时存放全局变量。这部分的空间大小可以由所有声明的全局变量的空间和算

出。

硬件堆栈区存放函数的返回地址。

SP 寄存器是堆栈指针，处始值为 SRAM 的尾部。

程序运行时硬件堆栈向全局变量区增长。

配置编译器时你可以选择在数据堆栈区（硬件堆栈区）的尾部放置字符串 DSTACKEND（ HSTACKEND）。当你使用 AVR Studio 调试程序时，如果看到这些字符串被覆盖，那么你就要改变你的 Project|Configure|C Compiler 中的配置。当程序调试好，你可以禁止放置这些字符串以减小代码长度。

24、Including Assembly Language in Your Program － 在程序中嵌入汇编语言

你可以在程序的任何地方使用 #asm 和 #endasm 指示符嵌入汇编语言。

例子：

```
void delay(unsigned char i) {
while (i--) {
        /* 汇编语言代码 */
        #asm
              nop
              nop
        #endasm
        };
}
```

行汇编的例子：

```
#asm("sei") /* 打开中断 */
```

在一行中使用多条汇编指令可以用“\”分开。

例子：

```
#asm("nop\nop\nop")
```

寄存器 R0, R1, R22, R23, R24, R25, R26, R27, R30 和 R31 可以在汇编子程序中自由使用。如果在中断过程使用这些寄存器，就必须在进入时保存，返回时恢复。

25、Calling Assembly Functions from C － 在 C 中调用汇编子程序

下面的例子给出了如何在 C 程序中调用汇编子程序：

```
// 这个函数用汇编写成，返回值是 a+b+c
#pragma warn- // 禁止 warning
int sum_abc(int a, int b, unsigned char c) {
#asm
      ldd     r30,y+3 ;R30=LSB a
      ldd     r31,y+4 ;R31=MSB a
      ldd     r26,y+1 ;R26=LSB b
      ldd     r27,y+2 ;R27=MSB b
      add     r30,r26 ;(R31,R30)=a+b
      adc     r31,r27
```

```
        ld      r26,y     ;R26=c
        clr     r27       ;转换无符号字符型变量 c 为整型
        add     r30,r26 ;(R31,R30)=(R31,R30)+c
        adc     r31,r27
    #endasm
    }
    #pragma warn+ // 允许 warning
    void main(void) {
    int r;
    // 调用函数 sum_abc，并把结构存在 r
    r=sum_abc(2,4,6);
    }
```

编译器使用 Data Stack 传递参数。

首先把整型参数 a 压栈，然后是 b，最后把无符号字符型参数 c 压栈。

每一次压栈，Y 寄存器对会根据参数的长度递减（长整型为 4，整型为 2，字符型为 1）。多字节时，MSB 先入栈。

这样看来数据堆栈是向下增长的。

所有参数压入数据堆栈后，Y 寄存器指向最后一个参数 c ，所以可以用“ld r26,y”把它的值读进 R26。

参数 b 比 c 先入栈，所以它在数据堆栈的地址比 c 高。

用“ldd r27,y+2”（MSB）和“ldd r26,y+1”（LSB）读出 b 。

由于 MSB 先入栈，所以它的高地址。

参数 a 比 b 先入栈，所以它在数据堆栈的地址更高。

用“ldd r31,y+4”（MSB）和“ldd r30,y+3”（LSB）读出 c 。

函数使用寄存器返回它的值（从 LSB 到 MSB ）：

R30 － 字符型和无符号字符型

R30 ，R31 － 整型和无符号整型

R30 ，R31 ，R22 ，R23 － 长整型和无符号长整型

所以这个函数用 R30 ，R31 返回结果。

函数返回后编译器会自动产生代码释放函数的参数占用的数据堆栈空间。

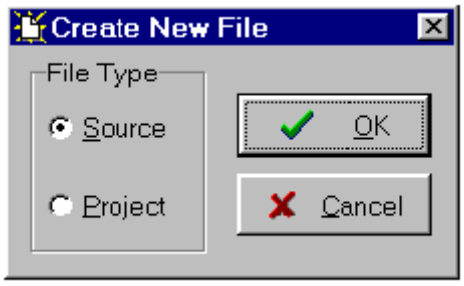
使用 #pragma warn- 编译指示符可以禁止编译器产生的函数没有返回值的警告。使用它是因为编译器不知道汇编的部分作了什么。

26、Creating Libraries － 建立自己的库

为了建立自己的库，你需要以下步骤：

1)以库里的函数的原型建立头文件。

选择 File|New 菜单命令或工具条的 New 按钮，会打开一个对话框：

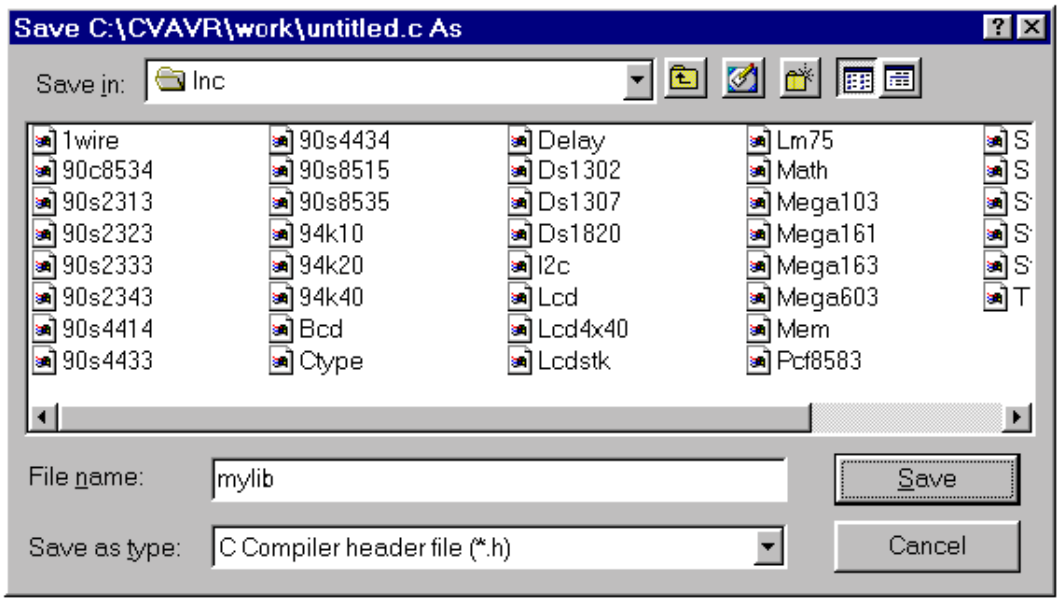


选择 Source 然后点 OK ，会打开一个新的源文件编辑窗口，文件名是 untitled.c 。这时你可以键入你的函数的原型了。

例子：

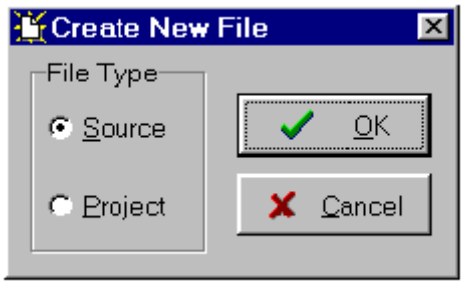
```
// 关闭编译时的 warning
#pragma used+
// 库函数原型
int sum(int a, int b);
int mul(int a, int b);
#pragma used-
// #pragma 指示符告诉编译器从 mylib.lib 库中编译或连接函数
#pragma library mylib.lib
```

使用 File|Save As 菜单命令在 ..\INC 目录保存新文件，例如：mylib.h 。如下图：



2) 建立库文件。

选择 File|New 菜单命令或工具条的 New 按钮，会打开一个对话框：



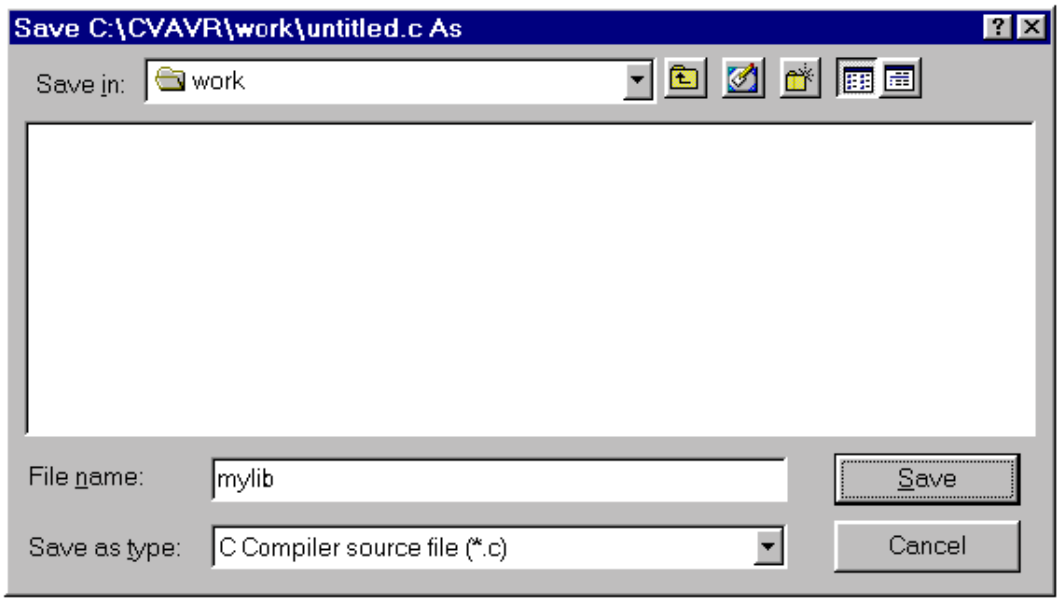
选择 Source 然后点 OK ，会打开一个新的源文件编辑窗口，文件名是 untitled.c 。这时你可以键入你的函数的原型了。

例子：

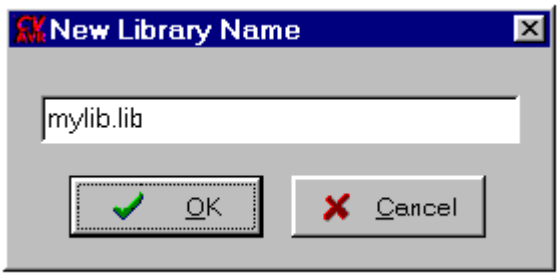
```
#if funcused sum
int sum(int a, int b) {
return a+b;
}
#endif
#if funcused mul
int mul(int a, int b) {
return a*b;
}
#endif
```

#if funcused 和 #endif 预编译指令使函数只在其有效时才被编译。

使用 File|Save As 菜单命令保存新文件，例如：mylib.c 。如下图：



最后使用 File|Convert to Library 菜单命令把目前打开的文件用文件名 mylib.lib 保存到 ..\LIB 目录：



为了使用新建的库 mylib.lib ，要在你的程序的开始用 #include 包含 mylib.h 头文件。

例子：

#include <mylib.h>

所有的库文件必须放在 ..\LIB 目录。

27、Using the AVR Studio Debugger － 使用 AVR Studio Debugger 调试程序

CodeVisionAVR 可以应用于 Atmel AVR Studio debugger 3.22 及以上版本。

为了能在 AVR Studio 中使用 C 源代码级调试在 Project|Configure|C Compiler|Compilation 选择输出文件格式为 COFF 。

你可以使用 Tools|Debugger 菜单命令调用 AVR Studio Debugger 或使用 Debugger 工具条上的按钮。 AVR Studio 调出后，你需要使用 File|Open 调试用的 COFF 文件。

程序加载后就可以用 Debug|Go 菜单命令运行，也可以按 F5 键或工具条上的 Run 按钮。

程序运行过程中可以用 Debug|Break 菜单命令停止，也可以按 Ctrl＋F5 键或工具条上的 Break 按钮。

单步运行可以用 Debug|Trace Into （F11 键），Debug|Step （F10 键），Debug|Step Out 菜单命令或工具条上对应的按钮。

用 Breakpoints|Toggle Program Breakpoint 菜单命令，F9 键或工具条上的 Toggle Breakpoint 按钮可以设置断点使程序停在指定的地方。

为了观察程序的变量，你可以用 Watch|Add Watch 菜单命令或工具条上的 Add Watch 按钮，并在 Watch 区内指定变量的名称。

可以用 View|Registers 菜单命令或按 Alt+0 键观察 AVR 的寄存器。

可以用 View|Processor 菜单命令或按 Alt+3 键观察 AVR 的 PC, SP, X, Y, Z 寄存器和系统标志位。

可以用 View|New Memory View 菜单命令或按 Alt+4 键观察 FLASH，SRAM 和 EEPROM 的内容。

可以用 View|New IO View 菜单命令或按 Alt+5 键观察 AVR 的 I/O 寄存器。

你可以使用 View|Terminal I/O 菜单命令调用终端通讯功能，它可以与 AVR 的模拟串行口通讯。但你在 Project|Configure|C Compiler 选择文件输出格式为 COFF，并选上 Use the Terminal I/O in AVR Studio 选项。

要获得更多关于 AVR Studio 使用方面的信息，请参考其帮助。

28、Hints － 提示

为了减小代码体积和加快程序运行速度，你要使用以下规则：

- 尽可能使用无符号变量；
- 使用最小的数据类型，例如位型和无符号字符型；
- 通过 Project|Configure|C Compiler|Compilation|Bit Variables Size 分配的位变量的空间要尽可能的小，以便空出寄存器可以分配给其它全局变量；

- 尽可能使用 TINY 模式；
- 使用 flash 关键字把常量放在 FLASH 中；
- 程序调试结束后，要关闭 Stack End Markers 选项把程序再编译一次；
- 与时间有关的部分用汇编来写。

29、Limitations － 限制

这个版本的 CodeVisionAVR C 编译器有如下限制：

- 不能用指向指针的指针
- 结构体或联合只能是一维的；
- 结构体或联合不能作为函数的参数，你只能使用指针来完成这个功能；
- 结构体中的成员不能使用位型。位的存储要使用位变量；
- 评估版有代码大小限制；
- 评估版没有以下功能：Philips PCF8563，Philips PCF8583，Dallas 温度计 DS1302，DS1307，单线通讯协议和 DS1820/DS1822 温度传感器，4×40 字符 LCD。