第九章　AVR C 语言的应用

★本章程序是用 IAR C正版软件编译通过的,并可产生*.HEX 烧录文件,用其它 C 语言编译是有差异的,不一定能通过,请用户注意这点!本章最后附几种 C 语言的比较,不仿一读。

★ 更详细资料参阅光盘文件<< AVR C 语言的应用>>

9.1　AVR – 支持 C 和高级语言编程的结构

一般高级语言

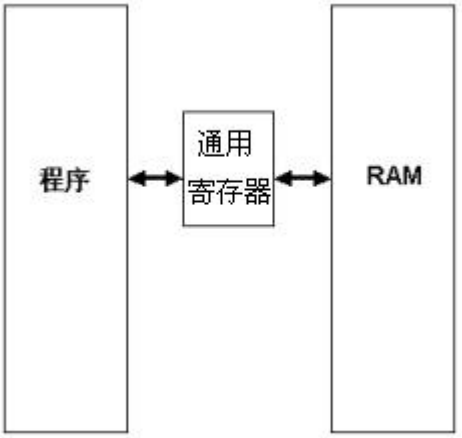
- 提高了 MCU 的重要性–缩短产品上市的时间
  - 简化了维护工作,可读性好
  - 轻便
  - 缩短学习时间
  - 可重复使用,便于移植
  - 方便调用库文件
- 潜在的缺点
  - 代码较大
  - 执行速度慢

为什么 AVR 适宜用高级语言编程?

因为它是为高级语言而设计的!

IAR 对 AVR 结构和指令集的影响

- 在结构/指令集确定之前，编译器的开发就开始了
- 潜在的瓶颈得到确认并消除
- IAR 的反馈在硬件设计上得到了反映
- 几次循环反复
- 修改后的结果从代码当中可看出来



存储器

- 32 通用寄存器
  - 数量多
  - 直接 与 ALU 连接
  - 可保存变量，指针和之间结果
- 线性程序存储空间
  - 1KBytes - 8MBytes
  - 无需页寻址
  - 常数区(SPM 可修改)
- 线性数据存储空间
  - 16 MBytes
  - 无需页寻址

类似于 C 的寻址模式

C 源代码

```
无符号的字符*var1, *var2;
*var1++ = *--var2;
```

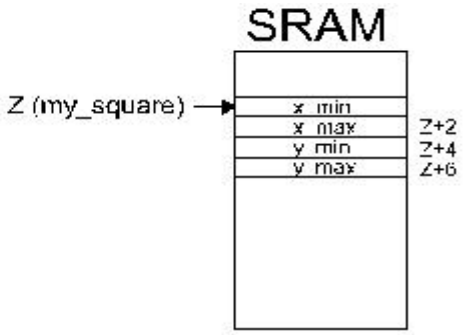
产生的代码

```
LD  R16,-X
ST  Z+,R16
```

带偏移量的间接寻址

- **有效访问数组和结构**
- Auto (local variables)**放置于软件堆栈之中**
  - **为适应重入的要求，高级语言都基于堆栈结构**

```
struct square
{
    int x_min;
    int x_max;
    int y_min;
    int y_max;
} my_square;
```

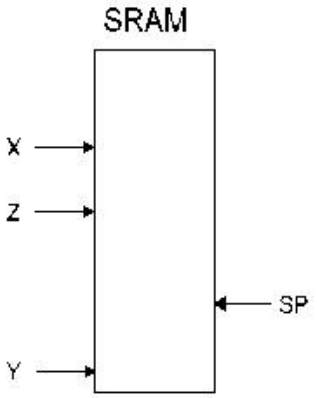


四种指针

16 和 32 位支持

- **加法指令**
  - **加和减**
  - **寄存器之间**
  - **寄存器和立即数之间**
  - Zero **标志的传播**

```
SUB     R16,R24
SUBI    R16,1
SBC     R17,R25      SBCI    R17,0
```

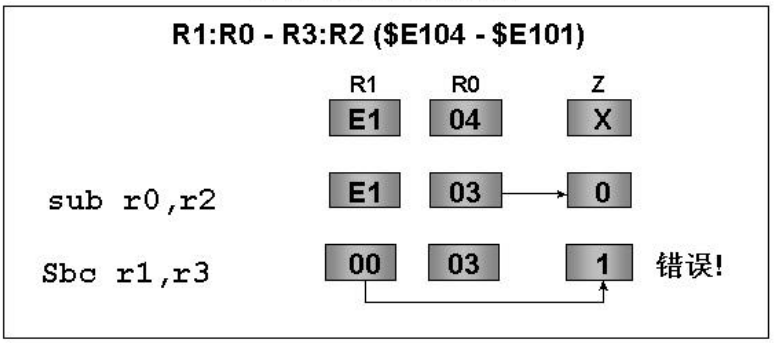


- 从内存拷贝到内存
- 最小化指针的反复加载
- 高功能（HIGH FUNCTIONALITY）

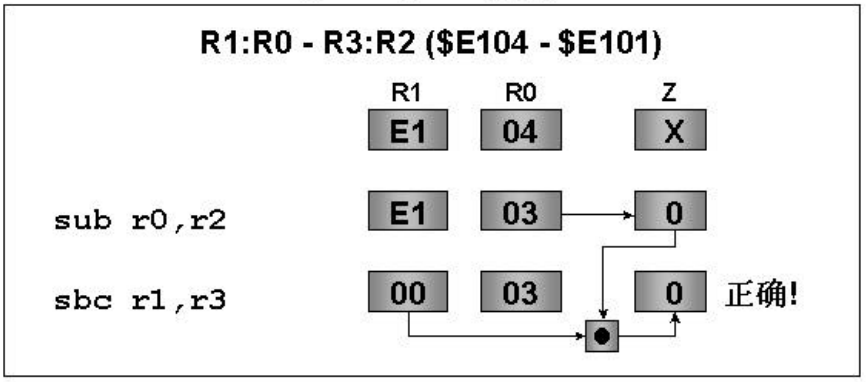
所有的跳转都基于最后结果

两个 16 位数相减

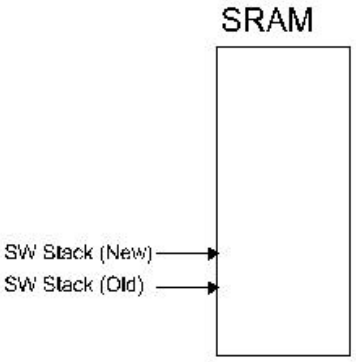
没有Zero标志的传播



有Zero标志的传播



16 位指令



- 小数值的加减
- 适合于索引变量和指针操作
- 软件堆栈

Non-destructive comparison

```
CP      R16,R24
CPC     R17,R25
CPC     R18,R26
CPC     R19,R27
```

- **带进位比较**
- Zero **传播**
- **无需保存结果**
- **可使用所有形式的跳转**

Switch 支持

- Switches **在** CASE **语句中经常遇到**
- Straight forward approach **效率低**
- **间接跳转适合于紧凑的** switch **结构**
- switch **由通用库管理**

摘 要

- AVR **结构从一开始就是针对高级语言设计的**
- Atmel **与** IAR **在结构和指令调整上的合作**

- 从而编译器可以产生高效的代码

AVR–高效的 C 编译器

减少代码的提示和诀窍汇编(Assembly) 与 C 比较

汇编:

- 可以完全控制资源
- 在小应用当中可以产生紧凑的、高速的代码
- 在大的应用当中代码效率低
- 可读性差（Cryptic code）
- 不好维护
- 不易移植（Non-portable）

C 编译器:

对资源的控制有限

- 在小应中产生的代码量大，执行速度慢
- 在大的应用当中代码效率高
- 结构化的代码
- 容易维护
- 容易移植

访问 I/O

- **读 I/O:** temp = PIND;

```
IN      R16,LOW(16)
```

- **写 I/O:** TCCR0 = 0x4F;

```
LDI     R16,79
OUT     LOW(51),R16
```

- **I/O 的位设置与清除**
- **地址小于 0x1F 的 I/O:**

```
PORTB |= (1<<PIND2);
  SBI     LOW(24),LOW(2)
ADCSR &= ~(1<<ADEN);
  CBI     LOW(6),LOW(7)
```

- **地址高于 0x1F 的 I/O:**TCCR0 &= ~(0x80);

```
IN      R16,LOW(51)
ANDI    R16,LOW(127)
OUT     LOW(51),R16
```

测试 I/O 的单个位

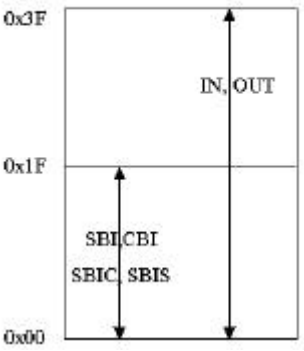
- **等待地址低于 0x1F 的单个位的清除**

```
while(PIND & (1<<PIND6));
  SBIC    LOW(16),LOW(6)
  RJMP    ?0002
```

- **等待地址高于 0x1F 的单个位的设置**

```
while(!(TIFR & (1<<TOV0)));
```

AVR I/O 内存映像



```
IN      R16,LOW(56)
SBRS    R16,LOW(0)
RJMP    ?0004
```

16 位变量• 总是使用最小的数据类型

- **8 位计数器:**

```
char count8 = 5;
do{
}while(--count8);
LDI     R16,5
DEC     R16
BRNE    ?0004
```

- Total 6 bytes

- **16 位计数器:**

```
int count16 = 5;
do{
}while(--count16);
LDI     R24,LOW(5)
LDI     R25,0
SBIW    R24,LWRD(1)
BRNE    ?0004
```

Total 8 Bytes

全局和局部变量

- **全局变量**
  - **在** startup **初始化**
  - **存储于** SRAM
  - **必须加载到寄存器堆中**
- **局部变量**
  - **在函数初期初始化**
  - **存储于寄存器当中直至函数结束**

全局变量和局部变量

- **局部变量**

```
void main(void)
{
   char local;
   local=local - 34;
}
   SUBI    R17,LOW(34)
```

- Total 2 bytes

- **全局变量**

```
char global;
void main(void)
{
   global=global - 45;
```

```
}
LDS    R16,LWRD(global)
SUBI R16,LOW(45)
STS    LWRD(global),R16Total 10 Bytes
```

高效地使用全局变量

- **将全局变量收集到一个结构中:**

```
typedef struct {
   int    t_count;
  char       sec;          // global seconds
  char       min;          // global minutes
}t;
t time;
Void main(void)
{
   t *temp = &time;
   temp->sec++; temp->min++; temp->t_count++;
}
```

带参数的函数调用

- **使用参数将数据传递到函数中去**

```
char add(char number1, char number2)
{
return number1+number2;
}
```

函数间参数的传递通过 R16-R23 **来实现**

循　环

- **死循环** for(; ;)

```
{
}
```

- **循环** char counter = 100;

```
do{
} while(--counter) ;
```

预减变量（Pre-decrement）**代码效率最高**

优化代码的选项

- **代码大小优化编译**
- **使用局部变量• 使用允许的最小数据类型**
- **将全局变量收集到结构中去• 死循环使用** for(;;)
- **使用预减的** do{ } while;

C　AVR 的程序设计

内　容,

- **安装必须的工具**

9.2　C 编译的介绍

- 练习

边学边做

- 用 C 编程
  - 设置编译和链接文件
  - 用 C 访问外围
  - 中断处理
  - 高级调试,使用不同的 AVR 外围
  - 定时器/计数器
  - UART
  - 外部中断 Tool flow
- 器件: AT90S8515
- C Compiler —C 编译器
- AVR Studio —AVR 仿真调试器
- SL-AVR —AVR 下载编程测试器
- 测试程序:死循环
  - 读 Port D 的值 (按键，输入口)
  - 将其值写到 Port B (LED，输出口)

测试程序(设文件名为 920.c)

```
#include <io8515.h>    /*  定义 AT90S8515 */
void main(void)
{
char c;
DDRB = 0xFF;          /* PortB all outputs */
     for(;;)           /* Eternal loop      */
     {
          c = PIND;          /* Read Port D       */
          PORTB = c;         /*  回写到 Port B */
     }
}
```

9.2.1. 安装 C 编译器

根据 IAR Readme 要求,从光盘安装编译器，文件安装好后,可把图标移到桌面成快捷工作图标,如图 921。

正版软件有加密狗,只有把狗插在打印口上，双击快捷图标,才能进入工作窗口。

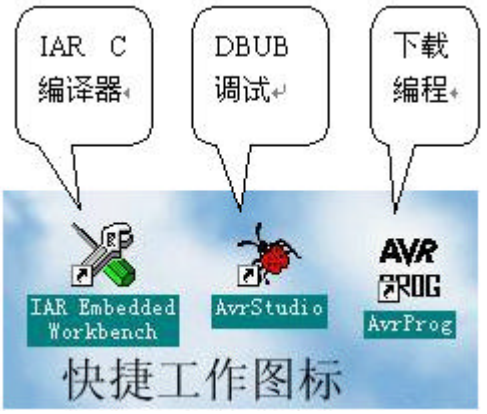


图 921 快捷工作图标

9.2.2 设置 C 编译器

启动 IAR 嵌入式工作台

可双击快捷图标,进入 IAR 编译器窗口。

1. 创建工程文件:按序操作,-File→New→Project 创建新工程文件,如图 922 新建工程文件窗口选择 CPU 为 A90,并输入工程文件名,如:920

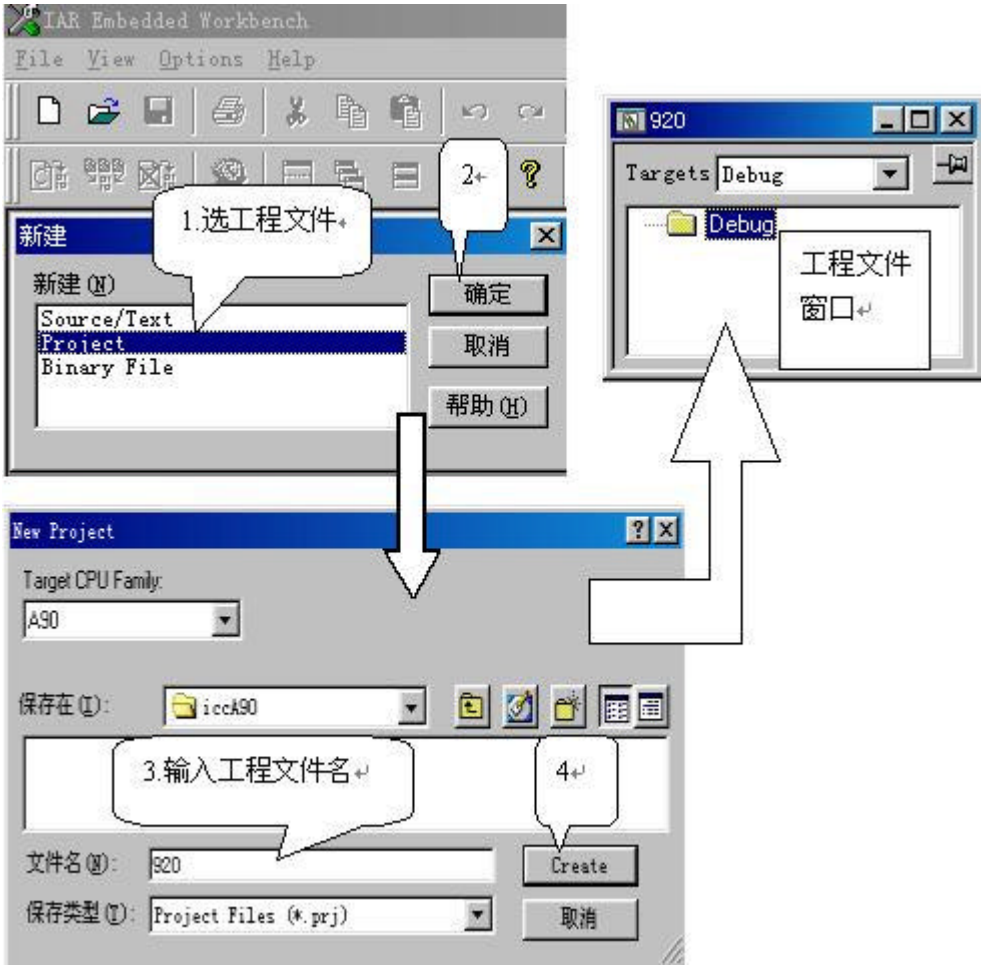


图 922　新建工程文件窗口

2．编辑 C 源程序文件设置，关于 C 语言知识，请参阅有关书籍。

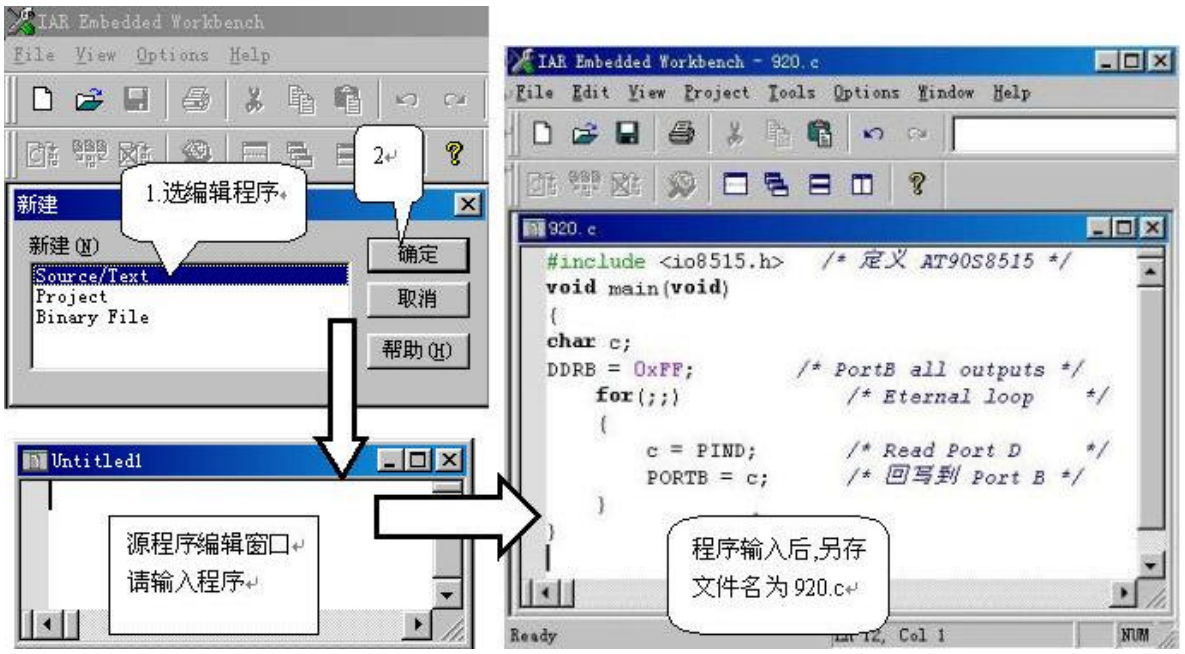


图 923　编辑源程序文件

3． 编辑 C 源程序,设置编译器选项

按序操作: Project -> Options

选择源文件对应器件(本例为 Max64Kbyte data,8Kbyte cod(8515,4414...))及内存模式(Small),见 图 924。

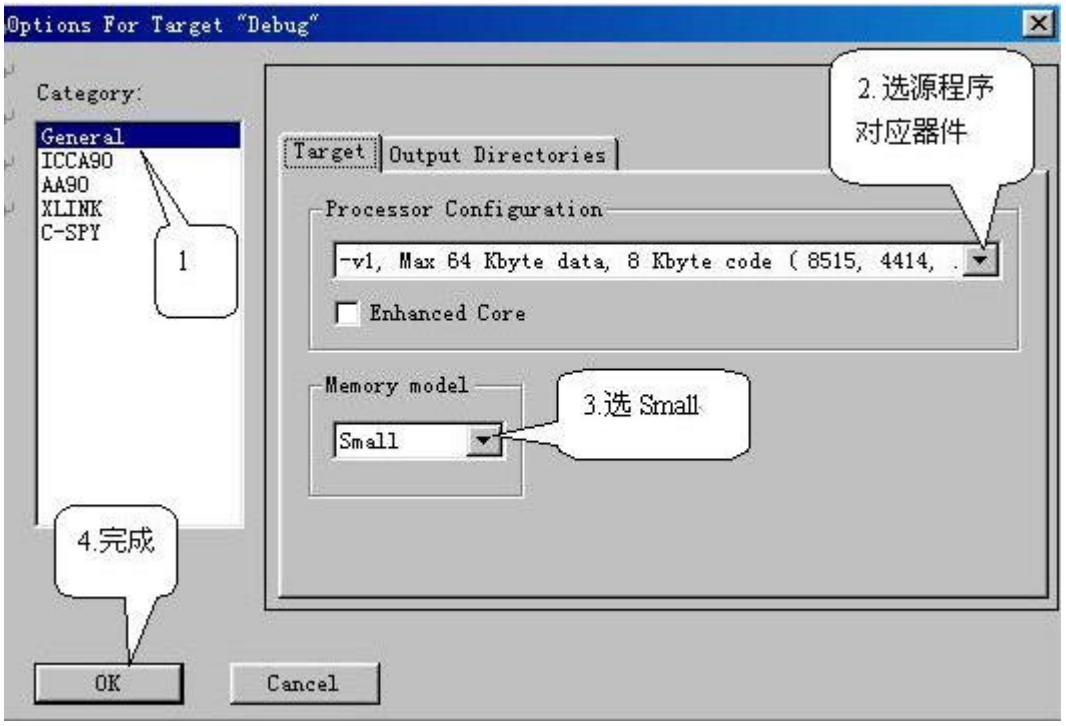


图 924 编辑源程序设置选项

4．设置链接器文件 XLINK 选项

在图 925 中选择 XLINK 选项,为生成下载文件,必须设定为 “intel-extended”输出格式(release only);

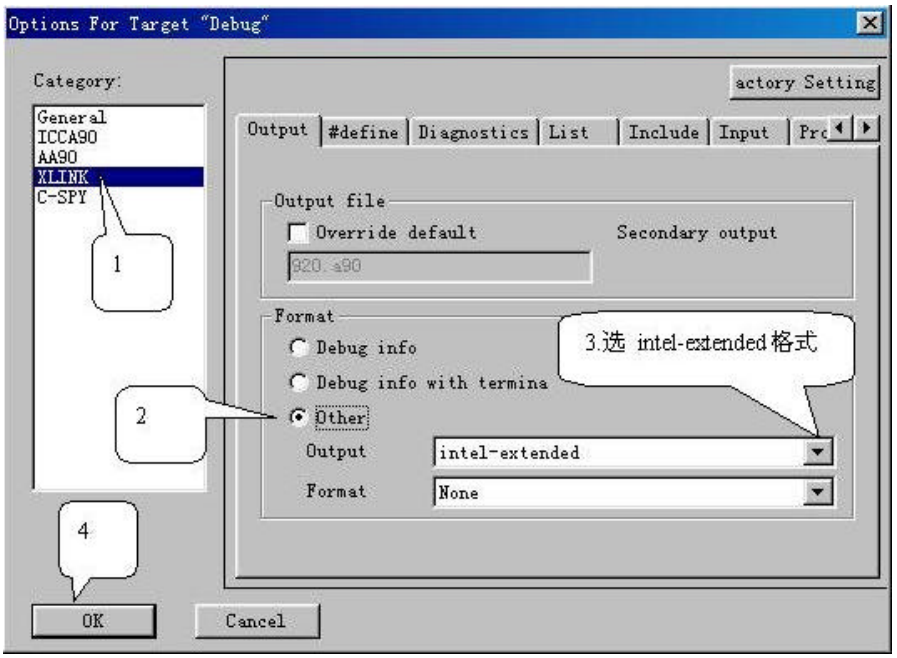


图 925　XLINK 选项,生成下载文件

5.设置链接器文件 XLINK 选项,选择处理器配置,设定内存模式见图 926

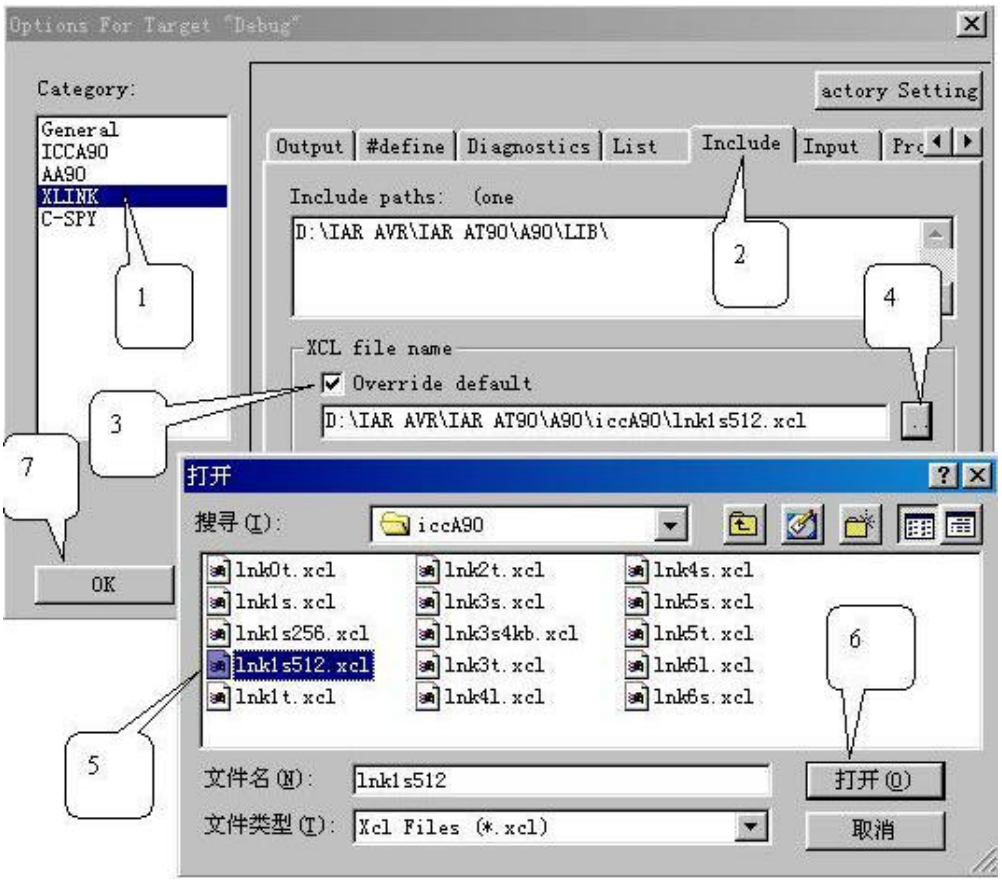


图 926 处理器配置及内存模式选项

6．生成调试文件选项,见图 927

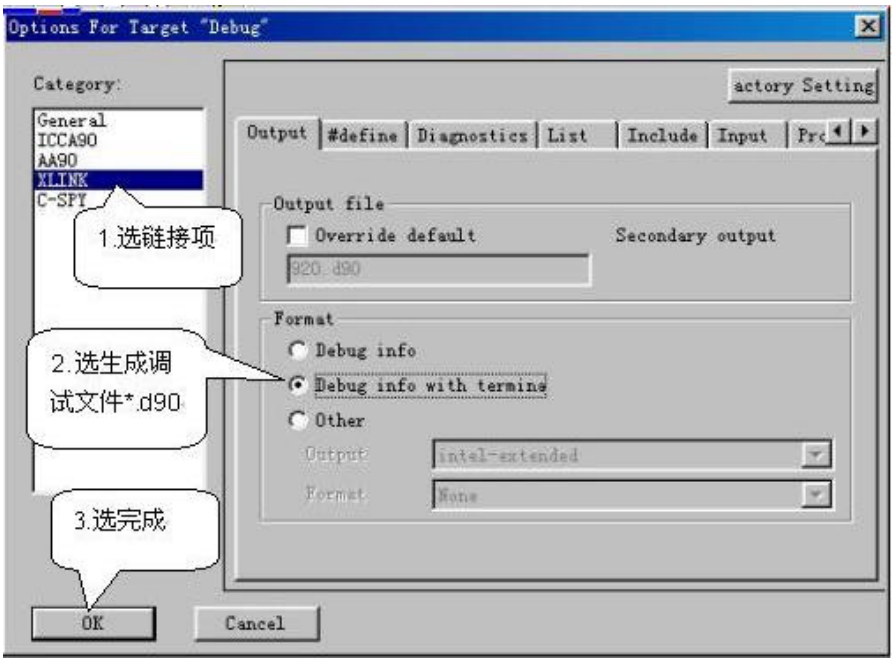


图 927 为生成调试文件选项

以上选项可以统统选完后,一次 OK。

7．将源程序加入到工程文件

按序操作 - Project -> Files ...,见图 928

选择源程序,按 Add 按钮,把上部窗口程序加到下部窗口,如有几个文件需链接,都应加到下部窗口。然后按 OK 按钮,设置编译器选项结束,显示工程文件窗口图 929。

图 928 将源程序加入到工程文件

8.对工程文件进行链接编译

如图 929,双击 *.C 文件,将会弹出源程序窗口。

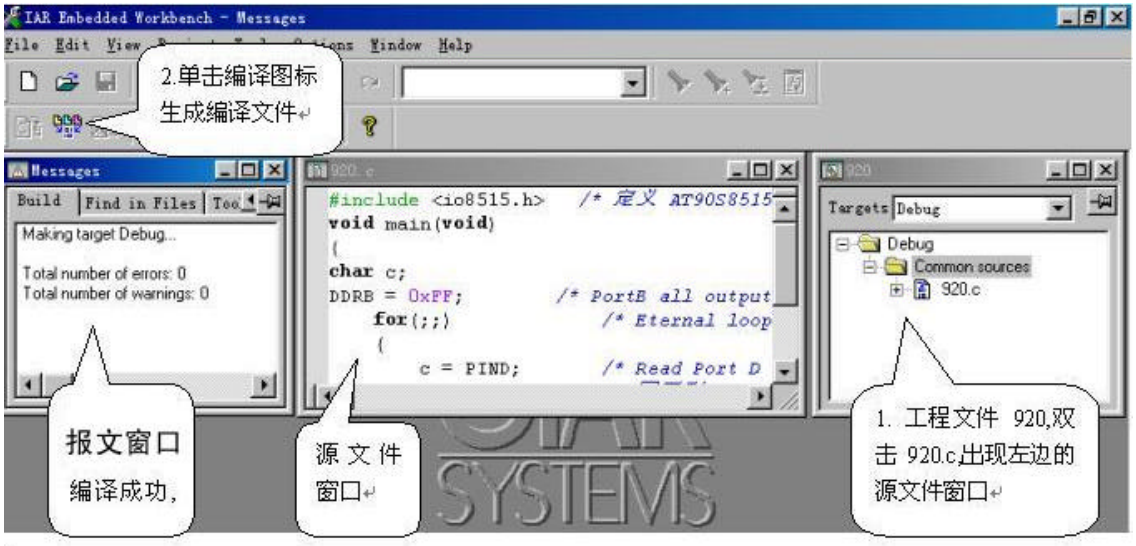


图 929 对工程文件进行链接编译

按序操作,– Project → Make 或键 F9,或快捷工具条图标。如有编译错误提示,请修改程序或检查编译器选项是否有问题,请改正,再编译,直到无错误报告,见图 929 报文窗口。

9. **查看编译生成的文件**

根据编译器的选项要求,程序编译后生成程序调试文件 *.d90,**编程下载文件** *.a90 **等,见图** 9210**。**

图 9210 **编译生成的文件**

9.2.3 使用 AVR Studio 调试

如程序编译通过,可进入程序调试。 C **语言的调试,见第三章** 3.2 **模拟调试窗口 。**

1. **启动** AVR Studio ,**双击桌面快捷图标**
2. **装入调试文件** (TestProg.d90) - File -> Open **如图** 9211

- **选择** AT90S8515 (**只需在开始时选择一次**)
- **加入视图**（VIEW）
  - I/O (PinB, PortD)
  - Processor
  - Watch

  »c

  »PORTB

  »PIND
- **单步执行**, Toggle PIND bits

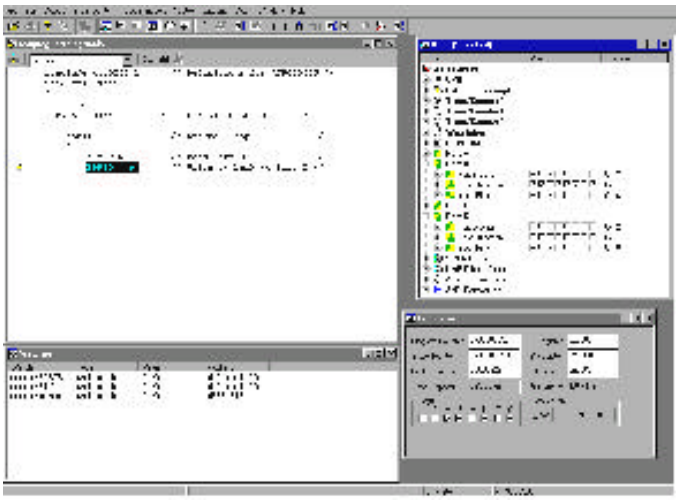


图 9211 “Debug”**调试窗口**

9.2.4 对器件编程

- **选择编程下载窗口** (**见第三章图** 3.30 AVR **下载窗口**)

9.3 测试应用程序

Main 函数

- "main" 是所有 C 程序的入口点
- 不要加入参数，也不要返回值• 语法:

```
void main(void)
{
    /* 代码 */
}
```

访问外围

- 所有 I/O 寄存器在头文件里都被定义为特殊功能寄存器
- 象普通变量一样访问

```
#include <io8515.h>      /* 定义 8515 */
void main(void)
{
   DDRD = 0xFF;          /* Port D 输出 */
}
```

9.3.1 读/写口

```
#include <io8515.h>      /* 定义 AT90S8515 */
void main(void)
{
char c;
DDRB = 0xFF;          /* PortB 输出          */
    for(;;)           /* 死循环              */
    {
        c = PIND;     /* 读 Port D       */
        PORTB = c;    /* 回写到 Port B */
    }
}
```

9.3.2 延时函数

```
#include <io8515.h>      /* 定义 8515 */
void delay(unsigned int delayValue)
{
  unsigned int i;

for(i=0;i<delayValue;i++)      /*  循环  */
        ;                      /* Do nothing  */
}
```

9.3.2A: 延时函数

```
void main(void)
```

```
{
    unsigned char runner = 0x01;

    DDRB = 0xff;                /* Port B 输出 */
  for (;;)                      /*  死循环    */
  {
    if (runner) runner <<= 1;
    else        runner = 0x01;
    PORTB = runner;         /* 设置 LED   */
    delay(100);             /* 调用延时函数 */
   }
}
```

9.3.3　读/写 E2PROM

```
  /* 利用 IAR 标准 I/O 函数来读/写 E2PROM */
  #include <io8515.h>
  #include <ina90.h>
  void main(void)
  {
  char temp = 0;
    _EEPUT(0x10,temp);  /* 写 E2PROM 地址：0x10 */

    _EEGET(temp,0x10);  /* 读 E2PROM 地址：  0x10    */
```

9.3.4　AVR 的 PB 口变速移位

```
/* 文件名:SLAVR934.ASM */
/* 本程序及以下所有程序,编译后在 SL-AVR 编程开发实验器上验证通过 */
/* 位运算符: ~ 按位取反; << 左移; >> 右移; & 按位与; | 按位或; ^ 按位异或;
;i++相当于 i=i+1;  i--相当于 i=i-1 */
#include <io8515.h>       ;       /* 器件配置文件 */
#define BIT(x)(1 << (x));     /* 左移 */
void delay(void)
{
 unsigned char i,j;
  for (i=1;i;i++)
   for(j=1;j;j++);
}
void led_pb(void)
{
 unsigned char i;
   DDRB=0xff;                  /* 设 PB 口输出 */
    for (i=0;i<8;i++)     /* 硬件设定低电平灯亮,LED 的 1 位亮灯从 B 口 PB0→PB7 */
      {
```

```
        PORTB=~BIT(i);     /* LED 亮灯 1 位 */
        delay( );              /* 延时 */
       }
}
void main (void)              /* 主函数 */
{
  while (1)           /* 循环 */
       led_pb( );
}
```

9.3.5　4 个口 LED 亮灯变速移位

```
/* 文件名:SLAVR935.ASM */
/* 请修改程序,改变移位方向,2 位或 3 位或一隔一亮灯移位等 */
#include <io8515.h>              /* 预处理命令,头文件 */
#define BIT(x)  (1 << (x))       ?* 定义位函数,可修改移位方向 */
void delay(unsigned char t) ;    /* 延时函数 */
{
 unsigned char i;
 unsigned char j;
 for (i=0;i<t;i++)
   for(j=1;j;j++);
}
void led_pb(unsigned char t);    /* LED 移位函数 */
{
 unsigned char i;
   DDRB=0xff;            /* 设 PB 口为输出 */
    for (i=0;i<8;i++)    /* 硬件设定低电平灯亮,LED 的 1 位亮灯从 B 口 PB0→PB7 */
      {
       PORTB=~BIT(i);    /* LED 亮灯 1 位 */
       delay(t);     /* 延时 */
      }
   PORTB=0xff;           /* 关 PB 口 */
}
void led_pd(unsigned char t);    /* LED 的 1 位亮灯从 D 口 PD0→PD7 移位函数 */
{
 unsigned char i;
   DDRD=0xff;
    for (i=0;i<8;i++)        /* LED 的 1 位亮灯从 D 口 PD0→PD7 */
      {
       PORTD=~BIT(i);
       delay(t);
      }
```

```
   PORTD=0xff;
}
void led_pc(unsigned char t);    /* C 口 PC0→PC7 移位函数 */
{
 unsigned char i;
   DDRC=0xff;
    for (i=0;i<8;i++)        /* LED 的 1 位亮灯从 C 口 PC0→PDC */
      {
       PORTC=~BIT(i);
       delay(t);
      }
   PORTC=0xff;
}
void led_pa(unsigned char t);    /* A 口 PA7→PA0 移位函数 */
{
 unsigned char i;
   DDRA=0xff;
    for (i=8;i>0;i--)        /* LED 的 1 位亮灯从 A 口 PA7→PA0 */
      {
       PORTA=~BIT(i-1);
       delay(t);
      }
   PORTA=0xff;
}
void main (void)         ;       /* 主函数 */
{
 unsigned char dt;
    while (1)            /* 循环 */
    {
      for (dt=5;dt<200;dt+=25)
        {
         led_pb(dt);     /* LED 发光二极管一亮灯沿四个口移位变速循环 */
         led_pd(dt);
         led_pc(dt);
         led_pa(dt);
        }
    }
}
```

9.3.6　音符声程序

```
/* 源程序 SLAVR936.ASM*/
/* 可改变 t 函数,改变发音快慢 */
#include <io8515.h>              /* 预处理命令 */
#define uchar unsigned char
#define uint  unsigned int
void delay(uchar t)
{
 uchar i,j;
 for (i=0;i<t;i++)
   for(j=1;j<150;j++);
}
void sound_pc0(uchar t)
{
 uint i;
   DDRC=0xff;
   PORTC=0xff;
    for (i=0;i<350-t*t;i++) /* 改变发音快慢 ,另见 SLAVR936B.ASM 程序*/
      {
       PORTC^=(1<<0);
       delay(t);
      }
}
void main (void)     /* 主函数 */
{
 uchar dt;
  for( ; ; )
    for(dt=1;dt<14;dt++)  /* 改变发音数量 */
     sound_pc0(dt);
}
```

9.3.7　8 字循环移位显示程序

```
/* 源程序 SLAVR937.ASM*/
/*在 SL-AVR 开发实验器 LED 数码管上,8 字符循环移位显示程序*/
#include <io8515.h>              /*器件配置文件*/
#define uchar unsigned char    /*定义缩写*/
#define uint  unsigned int
void delay(uint t)
{
 uint i;
 for (i=0;i<t;i++);
}
void init_disp(void)  /*B 口,D 口初始化*/
{
```

```
  DDRB=0xff;
  DDRD=0xff;
  PORTB=0x7f;    /*B 口送 8 字符,字形可修改*/
}
void scan(void) /*位选扫描*/
 {
  uchar i,j;
  for (i=0;i<6;i++)    /*i++可修改为一位隔一位或隔 2 位显示或改变移位方向*/
    {
       j=150;      /*可改变移位速度*/
     do
       {
        PORTD=~(0x01<<i);
        delay(150);    /*可改变 LED 显示亮度*/
        PORTD=0xff;
        delay(2100);  /*可改变 LED 显示亮度*/
        }
      while(--j);
     }
 }
void main(void)    /*主程序*/
 {
   init_disp();    /*初始化*/
   for( ; ; )
     scan( );    /*位选扫描*/
 }
```

9.3.8　按键加 1 计数显示程序

```
/*在 SL-AVR 开发实验器上,用 SHIFT 键,按 1 次键加 1 计数显示程序*/
#include <io8515.h> /*头文件*/
#define uchar unsigned char /*缩写定义*/
#define uint  unsigned int
flash uchar DATA_7SEG[ ]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,
                          0x7f,0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71};/*LED 字形表*/
uchar led[6]; /*显示缓冲*/
uint count; /*延时子程序*/
void delay(uint t)
{
 uint i;
 for (i=0;i<t;i++);
}
void init_disp(void)  /*初始化 B 口,D 口*/
```

```
{
  DDRB=0xff;
  DDRD=0x7f;
  PORTD|=0x80;
}
void disp(void)   /*键盘显示*/
 {
  uchar i;
  for (i=0;i<6;i++)
     {
       PORTD=~(0x01<<i);
       PORTB=DATA_7SEG[led[i]];
       delay(1000);
     }
  PORTB=0x00;
  PORTD=0xff;
 }
void be_pc0(void) /*发出一声响子程序*/
{
 uint i;
   DDRC|=0x01;
    for (i=0;i<350;i++)
      {
       PORTC^=0x01;
       delay(350);
      }
}
void conv(void) /*计数值转换成十进制数*/
 {
  led[5]=0;
  led[4]=count/10000;
  led[3]=count/1000%10;
  led[2]=count/100%10;
  led[1]=count/10%10;
  led[0]=count%10;
 }
void main(void) /*主程序*/
 {
  init_disp( ); /*初始化 B 口,D 口*/
  count=0;      /*开始计数值是零*/
  conv();   /*转换*/
  for( ; ; )
    {
      while((PIND&0x80)==0x80)  /*没有键按下等待*/
```

```
        disp();  /*显示*/
      be_pc0();   /*发出一声响*/
      count++;    /*计数器加 1*/
      conv( );    /*转换成十进制数*/
      while((PIND&0x80)==0) /*有键按下*/
          disp(); /*显示*/
    }
}
```

几种 C 语言的比较测试报告

詹卫前

自 ATMEL 的 AT90 系列单片机诞生以来，有很多第三方厂商为 AT90 系列开发了用于程序开发的 C 语言工具。本报告测试了以下四家厂商的 C 语言工具：IAR 的 ICC90、ImageCraft 的 ICCAVR、CodeVision AVR 和 SPJ 的 AVRC，其中 IAR 的 ICC90 是与 ATMEL 的 AT90 系列单片机同步开发的，是一个老牌的 C 语言工具，而其余三家是后来独立开发的。

在这四种 C 语言工具中，以 SPJ 的 AVRC 最不理想，其 IDE 工作环境不可与前三种相提并论，而且它的编译器工作方式与 CodeVisionAVR 相类似。 经初步测试其生成的代码，也不很理想，其版本更新的速度也较慢，所以没作进一步详细的测试。下面的比较只是对前三种 C 语言工具的比较。

一、IDE 工作环境的比较

IAR 的 ICC90 由于诞生的比较早，再加上其 IDE 为了和 IAR 其它系列单片机的开发环境相兼容，应该说其 IDE 环境不如 ICCAVR 和 CodeVisionAVR，在使用上也没有其余两个方便。但它也有自己的特点，即 IAR 有自己的源程序调试工具软件 C-SPY，而其余两家均只能通过生成 COFF 格式文件，在 ATMEL 的 AVR Studio 环境中进行源程序调试，而 IAR 在两个调试环境中均可以正常工作。

在 IDE 工作环境方面的差异主要有以下几个方面：

应用程序向导
串行通信调试终端
工具配置菜单
工程属性窗口

（一）、应用程序向导

IAR 没有应用程序向导，而 ICCAVR 与 CodeVision AVR 都具有应用程序向导，它们的共同点有：

可以根据选择的器件来产生 I/O 端口、定时器、中断系统、UART、SPI、模拟量比较器、片外 SRAM 配置的初始化代码。
都可以根据选定的晶振频率和设定的波特率，来计算波特率发生器 UBRR 的常数。
都可以自动生成相应的 C 语言文件
它们的区别是：

ICCAVR 除自动计算波特率外，还可以根据定时器的工作方式自动计算有关寄存器

的定时常数。而 CodeVisionAVR 则需要用户手工计算后，再输入相应的文本框中。CodeVisionAVR 除了可以产生 MCU 本身所固有的硬件的初始化代码外，还可以产生一些常用的外部硬件设备的初始化代码，如 I^2C 总线接口、Dallas 的单总线接口、字符型 LCD 接口，实时时钟 DS1302 的接口等等。

（二）、串行通信调试终端

ICCAVR 和 CodeVisionAVR 都有一个终端调试程序，用户可以根据需要自由地设置波特率、数据位和停止位、奇偶校验等参数，然后用于通信程序的调试。

在终端的功能方面 CodeVisionAVR 要强一些，其既可以十六进制数的形式进行发送、接受和显示数据，又可以文本的形式来发送、接受和显示数据。而 ICCVAR 只可以文本的形式来发送、接受和显示数据。

IAR 没有终端调试窗口。

（三）、工具配置菜单

在工具配置菜单方面 CodeVisionAVR 和 ICCAVR 比 IAR 出色，IAR 在菜单中只增加了一个配置菜单命令，用户可以将一些工具软件的启动命令加入其中。ICCAVR 在 IAR 的基础上增加了一些项目，如 AVR 资源计算器、支持 STK200/300 接口的在线编程（ISP）和基于串口通信的 ISP 编程。

CodeVisionAVR 除了具有用户可自己配置工具的特点外，增加了调试菜单命令和工具栏图标，但其只可以使用 ATMEL 的 AVR Studio 调试器。CodeVisionAVR 支持的在线编程器种类较多，其支持 STK200/300/500、DT006、VTEC-ISP 和 ATCPU/Mega2000 六种编程器。

（四）、工程属性窗口

IAR 的工程属性窗口可设置的项目较多，但对初学者使用反而不如 CodeVisionAVR 和 ICCAVR 方便，主要有以下几点原因：

1、IAR 的属性窗口不可以设置到具体的器件型号和准确地配置片外 SRAM，而 CodeVisionAVR 和 ICCAVR 可以设置到具体的器件型号，并且可以对片外 SRAM 进行较准确的配置。这样在使用时有些区别，如我们使用 AT90S8515 器件并且不使用片外 SRAM，在 IAR 的初始化程序中一定要加一行“MCUCR=0x00；”，否则在程序运行时 8515 的 PORTA 和 PORTC 两个端口会输出总线信号。而 CodeVisionAVR 和 ICCAVR 只需在工程属性窗口中设置即可，其余的工作由编译器自动完成。

2、如果用户需要修改 C 编译器的堆栈空间大小，IAR 的属性窗口对此无能为力，它需要修改相应的 XCL 文件才能达到目的；而 CodeVisionAVR 在工程属性窗口中可以直接修改软件堆栈的空间大小；ICCAVR 在工程属性窗口中可以直接修改硬件返回堆栈的空间大小，而 ICCAVR 的 RAM 除了用作硬件返回堆栈、全局变量和堆外，剩余的内存均是软件堆栈。

3、在一些应用中用户可能需要使用自己的启动文件，IAR 同样需要修改相应的 XCL 文件才能达到目的，而 CodeVisionAVR 和 ICCAVR 在工程属性窗口中可以直接指定使用外部启动文件。

4、当用户使用自己的库文件时，ICCAVR 可以直接指定相应的库文件；IAR 需要修改相应的 XCL 文件才能使用相应的库文件；而 CodeVisionAVR 必须在头文件或 C 语言文件中使用预处理命令#pragma library，才可以使用相应的库文件。

5、在 IAR 和 ICCAVR 中还有一项功能，即空余程序存贮空间的填充功能。用户使用这个功能，可以在空余的程序存贮器中填入特定的数据字节，如设置软件陷井等，而 CodeVisionAVR 没有这个功能。但 CodeVisionAVR 有另外一个特点，它自动将所有没有使用的中断向量均指向了复位向量入口，这也是一种抗干扰措施。

6、IAR 中有一个函数__low_level_init(void)，当程序在某些时候不需要初始化全部内存或需要初始化指定的端口时，可在 int__low_level_init(void)中加入自己的代码，让函数返回一个非 0 数值。这是另外两个软件所不具有的，也弥补了其不能方便地指定自定义启动文件的缺点。

二、C 语法扩充

由于 PC 机为冯-依曼结构，而 MCS51 和 AVR 均为哈佛结构，另外单片机的程序存贮器都是存放在 ROM 中的。因此几种 C 语言都进行了不同的语法扩充， 以适应结构的变化。

1、IAR 和 CodeVisionAVR 都定义了新的数据类型 sfrb 和 sfrw，使 C 语言可以直接访问 MCU 的有关寄存器，如 sfrb DDRD=0x11。

而 ICCAVR 没有定义 sfrb 和 sfrw 数据类型，而是采用强制类型换和指针的概念来实现访问 MCU 的寄存器，如

```
#define DDRD (*(volatile unsigned char *)0x31)
```

前者 sfrb 定义中的 0x11 为 DDRD 寄存器的 IO 地址，而后者定义中的 0x31 为 DDRD 寄存器在数据内存中的映射地址。

2、由于 AVR 单片机内部有三种类型的存贮器 RAM、EEPROM 和 FLASH 存贮器，为了能有效地访问这些存贮器三种 C 语言分别进行了不同的语法扩充。

IAR 中只扩充了一个关键词 flash。由于 AVR 的内部 RAM 数量有限，使用 flash 关键词可以将使用 const 类型定义的常量分配进 FLASH 存贮器，以节省 RAM 的使用。在 IAR 中对片内 EEPROM 的访问，只能通过函数_EEPUT 和_EEGET 进行访问。

在 ICCAVR 中，对 const 类型进行了扩充，编译器自动将 const 类型数据分配进 FLASH 存贮器中。对片内 EEPROM 存贮器，C 语言可以通过头文件 eeprom.h 中的函数对 EEPROM 中某一个具体地址进行访问；ICCAVR 同时也扩充了一个新的 eeprom 存贮区域，可以在 eeprom 区域中定义变量，然后再通过“&”运算符获取变量的地址对其进行访问。

在 CodeVisionAVR 中，扩充了 flash 和 eeprom 两个关键词，flash 的用法同 IAR。而由 eeprom 关键词限定的变量被分配进片内 EPROM 中，在 C 语言中访问 EEPROM 中变量的方法使用形式上和访问 RAM 中的变量完全相同（包括指针形式的访问）。

这三种 C 语言工具对 FLASH 中的代码和常数均可以生成 ROM 文件或 INTEL HEX 格式文件，而 ICCAVR 和 CodeVisionAVR 还可以对 EEPROM 的初始化数据生成 INTEL HEX 格式的.EEP 文件，IAR 没有这个功能。

3、由于在 C 程序中需要对 MCU 的中断进行处理，所以它们分别进行了语法扩充。IAR 和 CodeVisionAVR 都扩充了 interrupt 关键词，由该关键词限定的函数为中断处理函数。在 interrupt 关键词后面方括号中的内容为中断向量号，只不过 IAR 和 CodeVisionAVR 在有关头文件中用不同的符号对同一个中断号进行了宏定义。如：

IAR 中：interrupt [TIMER1_OVF1_vect] void timer1_overflow(void)

CodeVisionAVR 中：interrupt [TIM1_OVF] void timer1_overflow(void)

实际上它们是对应于同一个中断向量的。

ICCAVR 使用预处理命令#pragma interrupt_handler 来说明一个函数为中断处理函数。ICCAVR 采用这种方法的一个优点是可以将若干个中断向量指向同一个中断处理函数。如：

```
#pragma interrupt_handler timer:4 timer:5
```

中断向量 4 和 5 都指向中断处理函数 timer( )。

4、位操作

C 语言本身有较强的位处理功能，但在控制领域有时经常需要控制某一个二进制位，为此在 MCS-51 的 C 语言中（如 KEIL51）扩充了两个数据类型 bit 和 sbit，前者可以在 MCS-51 的位寻址区进行分配，而后者只能定义为可位寻址的特殊功能寄存器（SFR）中的某一位。这两个扩充为 MCS-51 应用 C 语言编程带来很大的方便。

而在针对 AVR 的三种 C 语言中，除 CodeVisionAVR 定义了 bit 数据类型外，其余两种语言都没有类似的定义；而 sbit 类型三种 C 语言都没有定义。

经过比较，在 AVR 中进行位操作运算 CodeVisionAVR 的功能最强。它一方面有 bit 类型的数据可用于位运算，另外在访问 IO 寄存器时可以直接访问 IO 寄存器的某一位。如访问 DDRB 的 D3 位，可以这样访问：

DDRB.3=1 或 DDRB.3=0

而在 IAR 和 ICCAVR 中没有 bit 类型的运算，当它们需要访问 IO 寄存器的某一位只能使用 ANSI C 语言的位运算功能。如访问 DDRB 的 D3 位，可以这样来访问（CodeVisionAVR 也可这样访问）：

DDRB|=(1<<3)　或　DDRB&=~(1<<3)

5、在线汇编

IAR 不支持在线汇编，而 ICCAVR 和 CodeVisionAVR 均支持在线汇编，即可在 C 语言高级语言程序中直接嵌入汇编语言程序，ICCAVR 甚至可以将汇编语言放在所有的 C 函数体之外。

在 ICCAVR 中，在线汇编使用虚假的 asm(“string”)函数，如访问 DDRB 的 D3 位，也可以这样访问：

asm(“sbi　0x17, 3”)或 asm(“cbi 0x17 , 3”)

如需要嵌入多行汇编指令，可以使用 “\n”分隔，如：

asm(“nop\n nop\n nop”)。

在 CodeVisionAVR 中在线汇编有两种格式，一种是使用#asm 和#endasm 预处理命令来说明它们之间的代码为汇编语言程序。如访问 DDRB 的 D3 位，可以这样访问：

```
#asm
  sbi  0x17, 3”
  nop
  cbi 0x17 , 3
#endasm
```

另外一种方式和 ICCAVR 有点类似，使用#asm (“string”)的形式。如上述程序我们改写一下，#asm(“sbi 0x17,3\n　nop\n　cbi 0x17,3”)，同样符号“\n”表示汇编指令换行。

6、内存模式

为了提高代码效率，C 语言一般都设置了一些内存模式，它们决定了编译时所使用指针的长度，下面依次介绍。

在 CodeVisionAVR 中有两种内存模式 Tiny 和 small，在 Tiny 模式访问 RAM 中变量使用的指针是 8 位的，此时如果使用指针访问 SRAM 只能访问 SRAM 的最低的 256 字节，而且此时不可以使用外部 SRAM。在 small 模式，使用 16 位的指针访问 SRAM，可以访问多达 64K 的 SRAM，此时可以使用外部 SRAM。对访问 FLASH 和 EEPROM 的指针，程序使用 16 位的指针，因此最多可以访问 64K 的空间。注意由于访问 FLASH 的程序指针为 16 位，因此对 ATmega103 编程时编译生成的二进制代码不能超越 64K。另外，使用 Tiny 模式可以获得较快的执行速度和较短的代码长度。

在 IAR 中由于工程属性配置不能具体到某一个特定器件，所以其目标处理器的配

置有两个项目。一个是处理器配置，其有从 v0 到 v6 七种配置，另外一个是内存模式，下面是七种配置的内存使用情况：

1)、v0 数据 SRAM 最大 256 字节、代码最大 8 K 字节

编译时只可以使用 Tiny 模式，IAR 中扩充的 near、far 和 huge 关键词不可使用。

2)、v1 数据 SRAM 最大 64K 字节、代码最大 8 K 字节

编译时只可以使用 Tiny 和 small 模式，Tiny 模式默认使用 256 字节数据 SRAM，small 模式默认使用最多 64K 字节 SRAM。系统默认使用 Tiny 内存模式，可以使用 tiny 和 near 关键词。

3)、v2 数据 SRAM 最大 256 字节、代码最大 128 K 字节

编译时只可以使用 Tiny 模式。

4)、v3 数据 SRAM 最大 64K 字节、代码最大 128 K 字节

编译时只可以使用 Tiny 和 small 模式。

5)、v4 数据 SRAM 最大 16M 字节、代码最大 128 K 字节

编译时只可以使用 small 和 Large 模式。

6)、v5 数据 SRAM 最大 64K 字节、代码最大 8M 字节

编译时只可以使用 Tiny 和 small 模式。

7)、v6 数据 SRAM 最大 16M 字节、代码最大 8M 字节

编译时只可以使用 small 和 Large 模式。

在 ICCAVR 中，没有专门设置编译内存模式，在编译时根据用户在工程属性中对 SRAM 的设置，自动决定使用哪一种指针。ICCAVR 中对 printf( )的版本是分等级的，使用等级越高的 printf( )，其功能越强，但要求代码空间也越大。

7、库文件

在 C 语言中一般都有很多库文件，IAR 只有一些常用的库，只可以实现对 IO 寄存器的访问；ICCAVR 有一些改进，在库文件封装了一些常用的低层操作，如：访问 EEPROM、UART、SPI 等等。CodeVisionAVR 在这方面做得较为出色，其不仅有与 IAR 和 ICCAVR 相同的库，而且增加了一些常用的硬件接口访问，并且也以库的形式封装起来。CodeVisionAVR 有一些比较有特点的库：

1)、访问 I^2C 接口的库

2)、访问 Dallas 的单总线协议接口的库

3)、访问 2 总线协议接口的库

3)、常用延时函数 delay_us( )和 delay_ms( )

4)、访问常用的字符 LCD 的库

5)、对一些常用的实时时钟或温度传感芯片，如 DS1302、DS1307、DS1621、DS1820/22、LM75、PCF8563、PCF8535 等，也提供了库文件支持。

8、对器件的支持

ICCAVR 和 CodeVisionAVR 均已支持到 AT94K，IAR 未见有说明。但对不含片内 SRAM 的 AVR 族系芯片如 AT90S1200、Tiny 系列（不含 Tiny22），均不支持。ImageCraft 另有一专门用于不含片内 SRAM 的 AVR 族系芯片的 ICCTiny C 语言工具。

三、代码的效率和速度

在代码效率方面，IAR 和 CodeVisionAVR 均有 speed 和 size 两种方式优化，其中 IAR 的优化等级又分为 0 到 9 级。而 ICCAVR 出于商业因素，其将代码优化和压缩功能放在了专业版中，如果在标准版中使用代码压缩功能，程序代码可以压缩，但有

部分程序可能不能正常工作。
对下面的程序我们进行代码效率分析：

```
#include <io8515.h>
void Delay(void)
    {
    unsigned char a, b;
    for (a = 1; a; a++)
        for (b = 1; b; b++)
            ;
    }

void LED_On(int i)
    {
    PORTB=~(1<<i);
    Delay();
    }

void main(void)
    {
    int i;
    MCUCR=0x00;
    DDRB = 0xFF;
    PORTB = 0xFF;
    while (1)
        {
        for (i = 0; i < 8; i++)
            LED_On(i);
            for (i = 8; i > 0; i--)
            LED_On(i);
        for (i = 0; i < 8; i += 2)
            LED_On(i);
        for (i = 7; i > 0; i -= 2)
            LED_On(i);
        }
    }
```

编译后生成的程序代码：

编译器	程序代码字节数
IAR	413
ICCAVR	311
CodeVisionAVR	327
KEIL51	136（LED 变化的速度明显慢得多）

注：对于 KEIL PORTB 换成 P1。
又比如对 ATMEL 文档中的例子程序：

```
int max(int *array)
   {
    char a;
    int maximum = -32768;
    for (a=0;a<16;a++)
    if (array[a]>maximum) maximum = array[a];
    return (maximum);
    }
```

编译后生成的代码和在 8MHZ 晶振下运行所需时间对比如下：

编译器名称	代码字节数	执行时间（8MHZ）	效率
IAR	58	47.63us	23.58
ICCAVR	62	50.75us	22.14
CodeVisionAVR	60	179.38us	6.26
KEIL51	57	1.1235ms	1

最后我们再看一个浮点运算程序：

```
#include <math.h>
 void main(void)
       {
         float x,y,z;
          x=1.0;
          y=2.0;
          z=sin(x+y);
       }
```

编译后生成的代码和在 8MHZ 晶振下运行所需时间对比如下：

编译器名称	代码字节数	执行时间（8MHZ）	效率
IAR	1237	747.5us	7.09
ICCAVR	1991	950.75us	5.58
CodeVisionAVR	1267	521us	10.17
KEIL51	1403	5.301ms	1

通过以上几张对比表格可以看出：

1、C 语言密度并不单纯地决定于 AVR 的结构，和编译器有很大关系。

2、在应用于 AVR 的三种 C 语言中：

1)、从代码效率和速度的平衡来看，应该是 IAR 最好。而 ICCAVR 和 CodeVisionAVR 各有千秋，ICCAVR 在由指针参与的数组运算中速度较快，而 CodevisionAVR 的浮点库函数运算较快。

2)、从 IDE 界面和库函数功能来看，应该是 CodeVisionAVR 最好，其次为 ICCAVR 和 IAR。

3)、对 64K 以上的代码和数据空间，IAR 的支持较好。

AVR 和 MCS-51 相比最突出的是 AVR 结构上的先进性以及其高速运算能力。