IAR-AVR C 编译器简要

本文所有资料来源于 IAR Embedded Workbench for Atmel AVR V4 Evaluation 里的 EWAVR_CompilerReference.pdf（IAR C/C++编译器参考指南），文中提到“具体参阅 IAR C/++ 编译器参考指南”指的就是该资料。本资料针对于 IAR AVR 4.20A 版本

第一章　数据类型（编译器支持 ISO/ANSI C 基本数据类型和一些附加数据类型。）

1．1．　整型数据

数据类型	大小	范围	对齐
bool	8 bits	0 to 1	1
char	8 bits	0 to 255	1
signed char	8 bits	-128 to 127	1
unsigned char	8 bits	0 to 255	1
signed short	16 bits	-32768 to 32767	1
unsigned short	16 bits	0 to 65535	1
signed int	16 bits	-32768 to 32767	1
unsigned int	16 bits	0 to 65535	1
signed long	32 bits	-2^{31} to $2^{31}-1$	1
unsigned long	32 bits	0 to $2^{32}-1$	1
signed long long	64 bits	-2^{63} to $2^{63}-1$	1
unsigned long long	64 bits	0 to $2^{64}-1$	1

bool 数据类型在C++语言里是默认支持的。如果你在C代码的头文件里包含*stdbool.h*, *bool*数据类型也可以使用在C语言里。也可以使用布尔值 *false*和 *true*。

1．2．浮点数据类型：

数据类型	大小	范围	Exponent	Mantissa
float	32 bits	±1.18E-38 to ±3.39E+38	8 bits	23 bits
double *	32 bits (default)	±1.18E-38 to ±3.39E+38	8 bits	23 bits
double *	64 bits	±2.23E-308 to ±1.79E+308	11 bits	52 bits
long double *	32 bits	±1.18E-38 to ±3.39E+38	8 bits	23 bits
long double *	64 bits	±2.23E-308 to ±1.79E+308	11 bits	52 bits

1．3．指针类型：指针有数据指针和函数指针。

1．3．1 数据指针：

数据指针的大小为8位，16位，24位。定义为：在整型数据类型后加”*”符号。

例：

*char * p;*

整型数据没有24位，具体定义指针见后面扩展关键字章节。

1．3．2 函数指针：函数指针的大小为16位，24位。

指针定义：在函数类型后加”*”符号。具体参阅IAR C/++编译器参考指南。

第二章 扩展关键字

可以用来解决数据，函数的存放等。有了它我们就可以定义变量存放在EEPROM，FLASH空间。定义中断函数，指针等等。IAR关键字很多，这里只列举常用的，其余的参考IAR C/++编译器参考指南

2．1．扩展关键字：用于控制数据和指针。

__eeprom 用于EEPROM 存储空间，控制数据存放，控制指针类型和存放

__tinyflash, __flash, __farflash, __hugeflash 用于flash 存储空间，控制数据存放，控制指针类型和存放：

__ext_io, __io 用于I/O存储空间，控制数据存放，控制指针类型和存放

__regvar 放置一个变量在工作寄存器中

2．2．函数扩展关键字：。

__nearfunc __farfunc 用于控制数据存放,这组关键字必须在函数声明和定义的时候指定：

__interrupt. 关键字控制函数的类型。这组关键字必须在函数声明和定义的时候指定

__root. 关键字仅仅控制有定义的函数：

2．3．其它特别的关键字：

@ 用于变量的绝对地址定位。也可以用*#pragma location* 命令

*#pragma vector*提供中断函数的入口地址。

__root 保证没有使用的函数或者变量也能够包含在目标代码中

__no_init 禁止系统启动的时候初始化变量.

asm, __asm 插入汇编代码

第三章　EEPROM常用类型的具体操作方法

根据第一和第二章节的内容，我们可以对IAR的数据进行具体的定义

3．1．EEPROM 区域数据存储。

用关键字 __*eeprom* 控制来存放， __*eeprom*关键字写在数据类型前后效果一样。

__*eeprom* unsigned char a;//定义一个变量存放在EEPROM空间

unsigned char __*eeprom* a;//效果同上

__*eeprom* unsigned char p[];//定义一个数组存放在EEPROM空间

对于EEPROM空间的变量操作同SRAM数据空间的操作方法一样，编译器会自动调用 __*EEPUT(ADR,VAL)*， __*EEGET(VAR, ADR)*宏函数来对EEPROM变量的操作。

例：

```
#include<iom8.h>
__eeprom unsigned char  p[];
__eeprom unsigned char a;
void main(void)
{p[1]++;// EEPROM变量的操作
a++;// EEPROM数组的操作
}
```

定义常数在EEPROM空间，只要给变量赋与初值就可以了。由于常数在EEPROM空间的地址是随机分配的，读取变量才可以读取到常数值。

__*eeprom* unsigned char a=9;//定义一个常数存放在EEPROM空间

__*eeprom* unsigned char p[]={1，2，3，4，5，6，7，8};//定义一个组常数存放在EEPROM空间

例：

```
#include<iom8.h>
  __eeprom unsigned char  p[]={1,2,3,4,5,6,7,8};
__eeprom unsigned char a=9;
void main(void)
```

{PORTB=a;//读取EEPROM空间值9

PORTC=p[0]; //读取EEPROM空间值

}

EEPROM空间绝对地址定位：

__*eeprom* unsigned char a @ 0x8;//定义一个变量存放在EEPROM空间0X08单元

__*eeprom* unsigned char p[] @ 0x22//定义一个数组存放在EEPROM空间，开始地址为0X22单元

__*eeprom* unsigned char a @ 0x08=9;//定义一个常数存放在EEPROM空间0X08单元

__*eeprom* unsigned char p[] @0x22={1，2，3，4，5，6，7，8};

//定义一个组常数存放在EEPROM空间开始地址为0X22单元

由于常数在EEPROM空间的地址是已经分配的，读取EEPROM空间值可以用变量和地址。

3．2．与 __*eeprom*有关的指针操作。 __*eeprom*关键字控制指针的存放和属性

3.2.1指向eeprom空间的指针EEPROM指针（控制属性）

unsigned char __*eeprom* * p;//定义一个指向EEPROM空间地址的指针，8位指针本身存放在SARM中。

unsigned int __*eeprom* * p;//定义一个指向EEPROM空间地址的指针，16位指针本身存放在SARM中。

unsigned char __*eeprom* * p;//定义一个指向EEPROM空间地址的指针，指针本身存放在SARM中。P的值代表EEPROM的空间的某一地址。*p表示EEPROM空间某一地址单元存放的内容。例：假定p=10,表示EEPROM空间地址10单元，而EEPROM空间10单元的内容就用*p来读取。

例：

#include<iom8.h>

char __*eeprom* t @ 0x10 ;

char __*eeprom* *p ;

void main(void)

{PORTB=*p; //读取EEPROM空间10单元的值

```
PORTB=*（p+3）; //读取EEPROM空间0X13单元的值
}
```

3.2.2. 存储于eeprom空间的指针数据指针（就象存储与EEPROM空间的数据一样）控制存储

```
__eeprom unsigned char * p; //定义一个指向SARMM空间地址的指针，指针本身存放在
```

EEPROM中。

3．3. 控制数据和指针存放的__eeprom定义必须是全局变量，控制属性（好像只有指针）可以是局部变量。

```
#include<iom8.h>
__eeprom unsigned char p;//控制存放
void main(void)
{unsigned char __eeprom *  t;//控制属性
PORTB=p;
PORTB=*t;
}
```

3．4.　　__root关键字：告诉编译器未使用的代码也要编译。

定义存放在eeprom空间的全局变量在程序编译时会自动生成.eep文件以供烧录。对于程序没有使用也要求编译的数据必须加关键字　__root 限制。

例：

```
#include<iom8.h>
 __root   __eeprom unsigned char p @ 0x10 =0x56;
void main(void)
{}
```

程序没有使用P变量，编译也会生成.eep文件的。

```
:020000020000FC
:0100100056 99
:0400000300000000F9
:00000001FF
```

3．5. EEPROM操作宏取函数：在comp_a90.h　intrinsics.h头文件里有详细说明。

```
__EEPUT(ADR,VAL); //向指定EEPROM空间地址（ADR）中写入数据（VAL）。
__EEGET(VAR,ADR); //从指定EEPROM空间地址（ADR）中读取数据（VAL）。实际上两种
```

函数的原形如下：

```
#define    __EEPUT(ADR,VAL)  {while (EECR & 0x02); \
 EEAR = (ADR); EEDR = (VAL); EECR = 0x04; EECR = 0x02;}
#define    __EEGET(VAR, ADR) {while (EECR & 0x02); \
EEAR = (ADR); EECR = 0x01; (VAR) = EEDR;}
```

对于定义为EEPROM空间的变量操作同SRAM数据空间的操作方法一样，编译器会自动调用 *__EEPUT(ADR,VAL)， __EEGET(VAR, ADR)*宏函数来对EEPROM变量的操作。

例：

```
#include<iom8.h>
__eeprom unsigned char a;
void main(void)
{a++;
// EEPROM变量的操作，自动调用 __EEPUT(ADR,VAL)，__EEGET(VAR, ADR)函数。
}
```

对于直接在程序中使用 *__EEPUT(ADR,VAL)， __EEGET(VAR, ADR)*函数必须要包含头文件ina90.h。

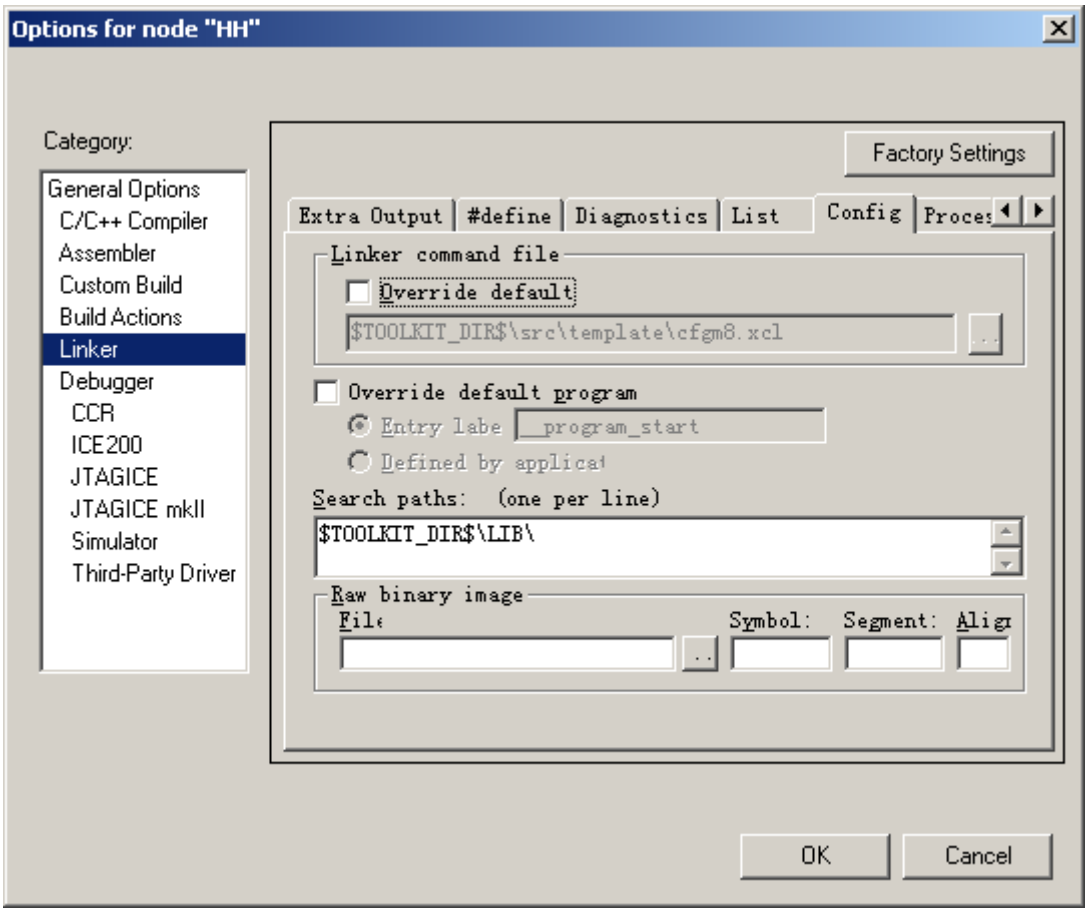
例：

```
#include<iom8.h>
#include<ina90.h>
void main(void)
{char c;
  __EEPUT(0x10,0x23);//向EEPROM空间0X10单元写入值0X23
  __EEGET(c, 0x10); //读取EEPROM空间0X10单元值
PORTB=c;
}
```

3．6．自动生成.eep文件置：

操作步骤：

1、 在Project->Options->linker->config>的linker command line中观察该Project使用了哪个XCL文件 。本文使用M8编译，使用文件是”$TOOLKIT_DIR$\src\template\cfgm8.xcl”



2、打开该文件，在该XLC文件最后中加入以下两行：

-Ointel-extended,(CODE)=.hex

-Ointel-extended,(XDATA)=.eep

本人用的cfgm8.xcl文件

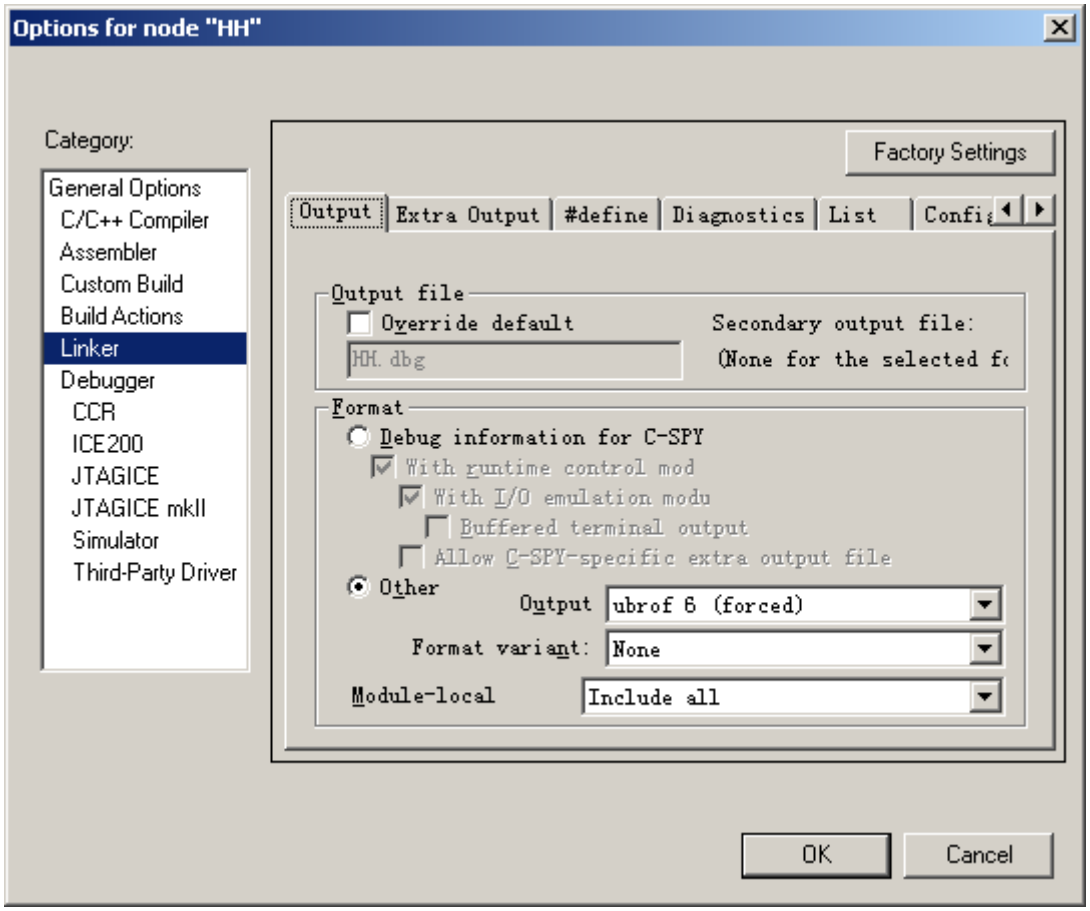
```
////////////////////////////////////////////////////////////////////////////////
/
//
// Contains resource definitions at Atmel ATmega8
//
// File version: $Revision: 1.11 $
//
// The '_..X_' prefix is used by C-SPY as an indication that the label should
```

```
// not be displayed in the dissassembly window.
//

////////////////////////////////////////////////////////////////////////////
/
// Code (flash) segments
-D_..X_INTVEC_SIZE=26     // 2 bytes * 19 vectors
-D_..X_FLASH_NEND=1FFF    // End of near flash memory
-D_..X_FLASH_END=1FFF     // End of flash memory

////////////////////////////////////////////////////////////////////////////
/
// Data (SRAM, external ROM or external NV RAM) memory
-D_..X_SRAM_BASE=60
-D_..X_SRAM_TBASE=_..X_SRAM_BASE // Start of tiny ram memory
-D_..X_SRAM_TSIZE=(100-_..X_SRAM_BASE) //Size of the tiny ram memory
-D_..X_SRAM_END=45F

-D_..X_EXT_SRAM_BASE=_..X_SRAM_END     // External memory not possible
-D_..X_EXT_SRAM_SIZE=0                 // External memory not possible
-D_..X_EXT_ROM_BASE=_..X_SRAM_END      // External memory not possible
-D_..X_EXT_ROM_SIZE=0                  // External memory not possible
-D_..X_EXT_NV_BASE=_..X_SRAM_END       // External memory not possible
-D_..X_EXT_NV_SIZE=0                   // External memory not possible

////////////////////////////////////////////////////////////////////////////
/
// Internal EEPROM
-D_..X_EEPROM_END=1FF     // End of eeprom memory
-D_..X_EEPROM_START=0
```

-Ointel-extended,(CODE)=.hex

-Ointel-extended,(XDATA)=.eep

3、 Project->Options->XLINK->Output->Format->Other中选择UBROF6格式输出。



例：

```
#include<iom8.h>

   __root  __eeprom char a[] @ 5={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

void main(void)

{}
```

经编译生成文件.hex

```
:020000020000FC

:1000000016C0189518951895189518951895189518955F

:1000100018951895189518951895189518951895189578

:10002000189518951895089500008895FECF0FE94A
```

:100030000DBF00E00EBFC0E8D0E003D0F4DFF4DF76

:06004000F3CF01E008957A

:0400000300000000F9

:00000001FF

经编译生成文件.eep

:020000020000FC

:0F000500010203040506070809 0A0B0C0D0E0F74

:0400000300000000F9

:00000001FF

第四章　FLASH常用类型的具体操作方法

4．1．FLASH 区域数据存储。

用关键字 __*flash* 控制来存放， __ *flash* 关键字写在数据类型前后效果一样

__*flash* unsigned char a;//定义一个变量存放在*flash*空间

unsigned char __*flash* a;//效果同上

__*flash* unsigned char p[];//定义一个数组存放在*flash*空间

对于*flash*空间的变量的**读**操作同SRAM数据空间的操作方法一样，编译器会自动用 *LPM，ELPM* 指令来操作。

例：

```
#include<iom8.h>
__flash unsigned char  p[];
__flash unsigned char a;
void main(void)
{PORTB=p[1];// 读flash 数组变量的操作
PORTB=a;// 读flash 变量的操作
}
```

由于在正常的程序中，*flash* 空间是只读的，所以没有赋值的变量是没有意义的。定义常数在*flash* 空间，只要给变量赋与初值就可以了。由于常数在*flash*空间的地址是随机分配的，读取变量才可以读取到常数值。

__*flash* unsigned char a=9;//定义一个常数存放在EEPROM空间。

__*flash* unsigned char p[]={1，2，3，4，5，6，7，8};

//定义一个组常数存放在*flash* 空间。

例：

```
#include<iom8.h>
__flash unsigned char  p[]={1,2,3,4,5,6,7,8};
__flash unsigned char a=9;
void main(void)
{PORTB=a;//读取flash 空间值9
PORTC=p[0]; //读取flash 空间值
}
```

4.1.2flash 空间绝对地址定位：

__*flash* unsigned char a @ 0x8;//定义变量存放在*flash* 空间0X08单元__*flash*

unsigned char p[] @ 0x22//定义数组存放在*flash* 空间，开始地址为0X22单元

__*flash* unsigned char a @ 0x08=9;//定义常数存放在*flash* 空间0X08单元

__*flash* unsigned char p[] @ 0x22={1，2，3，4，5，6，7，8};

//定义一个组常数存放在EEPROM空间开始地址为0X22单元

由于常数在*flash* 空间的地址是已经分配的，读取*flash* 空间值可以用变量和地址。

4．2．与 __*flash* 有关的指针操作。 __*flash* 关键字控制指针的存放和类型。

4.2.1指向*flash* 空间的指针*flash* 指针（控制类型属性）

unsigned char __*flash* * *p;*//定义指向*flash* 空间地址的指针，8位。

unsigned int __*flash* * p;//定义个指向*flash* 空间地址的指针，16位。

unsigned int __*farflash* * p;//定义指向*flash* 空间地址的指针，24位。

unsigned int __*hugeflash* * p;//定义指向*flash* 空间地址的指针，24位。

unsigned char __*flash* * p;//定义一个指向*flash* 空间地址的指针，指针本身存放在SARM中。P的值代表*flash* 空间的某一地址。*p表示*flash* 空间该地址单元存放的内容。

例：假定p=10,表示*flash*空间地址10单元，而*flash* M空间10单元的内容就用*p来读取。

例：

```
#include<iom8.h>
```

```
char  __flash t @ 0x10 ;
char  __flash  *p ;
void main(void)
{PORTB=*p;//读取flash 空间10单元的值
PORTB=*（p+3）;//读取flash 空间0x13单元的值
}
```

4.2.2. 存储于*flash* 空间的指针数据指针

就象存储与*flash* 空间的数据一样控制存储属性

__*flash*unsigned char＊ p; //定义指向SARMM空间地址的指针，指针本身存放在*flash*中。

4．3. 控制数据和指针存放的__*flash* 定义必须是全局变量，控制类型属性（好像只有指针）可以是局部变量。

```
#include<iom8.h>
__flash unsigned char p;//控制存放
void main(void)
{unsigned char   __flash* t;//控制属性
PORTB=p;
PORTB=*t;
}
```

4．4. __*root* **关键字**保证没有使用的函数或者变量也能够包含在目标代码中.

定义存放在__*flash* 空间的数据在程序编译时会自动生成代码嵌入到*flash*代码中，对于程序没有使用也要求编译的数据（比如可以在代码中嵌入你的版本号，时间等）必须加关键字__*root* 限制。

例：

```
#include<iom8.h>
 __root  __flash unsigned char p @ 0x10 =0x56;
void main(void)
{}
```

程序没有使用P变量，编译也会生成该代码。

```
:020000020000FC
```

```
:1000000016C0189518951895189518951895189518955F
:1000100056951895189518951895189518953A
:10002000189518951895089500008895FECF0FE94A
:100030000DBF00E00EBFC0E8D0E003D0F4DFF4DF76
:06004000F3CF01E008957A
:0400000300000000F9
:00000001FF
```

4．5．*flash* 操作宏函数：在comp_a90.h intrinsics.h头文件里有详细说明。

flash 空间具正常情况下有只读性能，对于读*flash* 数据编译器会自动编译对应的LPM,ELPM指令，但对于*flash* 空间的自编程写命令SPM就没有对应的C指令了，这里不讲解详细的自编程方法，只是讲解一下对*flash* 的读写函数。

直接在程序中读取*flash* 空间地址数据：要包含intrinsics.h头文件

*__load_program_memory(const unsigned char __flash *);*//64K空间

//从指定*flash* 空间地址读数据。该函数在intrinsics.h头文件里有详细说明。

在comp_a90.h文件有它的简化书写_LPM(ADDR)。注意汇编指令LPM Rd ,Z中的Z是一个指针。所以用*(const unsigned char __flash *)*来强制转换为指向*flash* 空间地址指针。故该条宏函数的正确写法应该如下：

*__load_program_memory((const unsigned char __flash *) ADDR);*

例：

```
#include<iom8.h>
#include <intrinsics.h>
void main(void)
{PORTB=__load_program_memory((const unsigned char __flash *)0x12);
}
```

该条函数书写不方便，在comp_a90.h文件有简化：

#define _LPM(ADDR) __load_program_memory (ADDR)稍微方便一点。改为

#define _LPM(ADDR) __load_program_memory ((const unsigned char *__flash* *)ADDR)就更方便了，直接使用数据就可以了。

例：

```
#include<iom8.h>
```

```
#include<comp_a90.h>
#include<intrinsics.h>
void main(void)
{PORTB=__LPM(0x12);// 从指定flash 空间地址单元0x12中读数据
}
__extended_load_program_memory(const unsigned char __farflash *);//128K空间
_ELPM(ADDR); //128K空间
```

参照上面的理解修改可以书写更简单。

4.6.自编程函数：

```
_SPM_GET_LOCKBITS(); //读取缩定位
_SPM_GET_FUSEBITS(); //读取熔丝位
_SPM_ERASE(Addr); //16位页擦除
_SPM_FILLTEMP(Addr,Word); //16位页缓冲
_SPM_PAGEWRITE(Addr; )//16位页写入
_SPM_24_ERASE(Addr); //24位页擦除
_SPM_24_FILLTEMP(Addr,Data); //24位页缓冲
_SPM_24_PAGEWRITE(Addr) //24位页写入
```

详细参见IAR C/++编译器参考指南。

第五章 SARM 数据类型的具体操作方法

SARM空间是AVR单片机最重要的部分，所有的操作必须依赖该部分来完成。变量在SARM空间的存储模式有tiny ,small large 三种，也就是对应于__tiny, __near __far三中存储属性。一旦选择为哪种存储模式，对应的数据默认属性也就确定了，但可以采用__tiny, __near __far关键字来更改。

对于程序中的局部变量，编译器会自动处理的，我们也不可能加什么储存属性，但IAR提供了强大的外部变量定义。

5．1．定义变量在工作寄存器

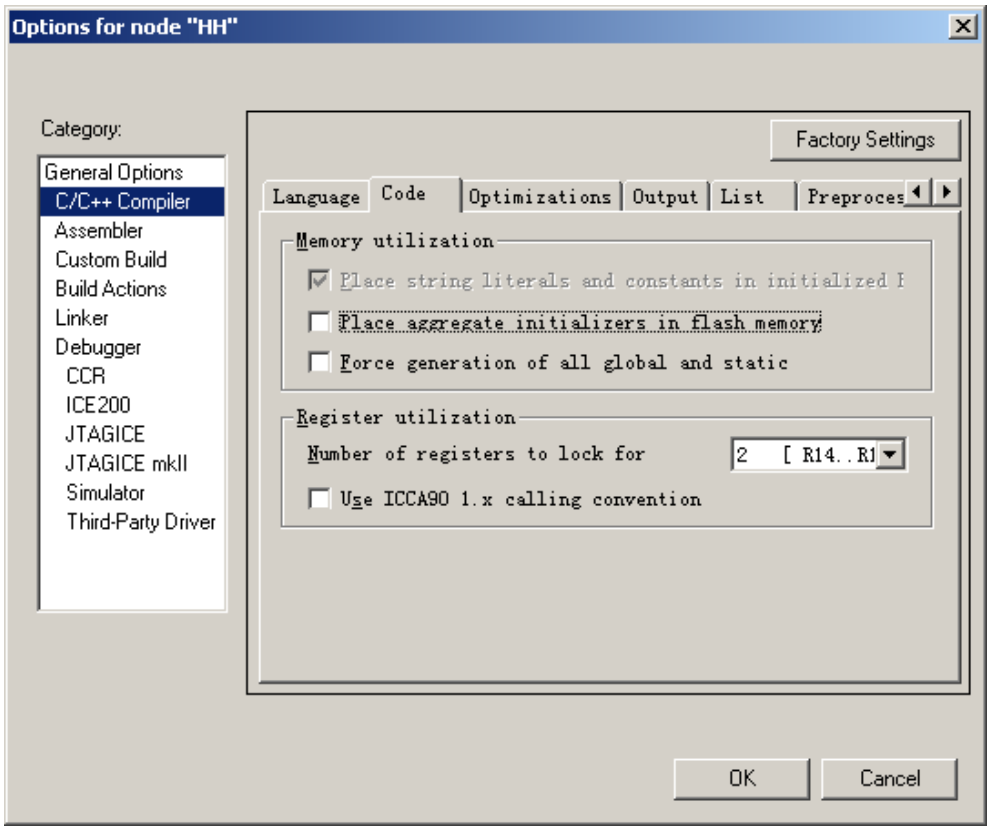
IAR编译器内部使用了部分工作寄存器，留给用户的只有R4-R15供12个寄存器供用户使用，要使用工作寄存器必须在工程选项里打开锁定选项。

例：

定义两个变量使用工作寄存器R14，R15。

```
#include<iom8.h>

__regvar __no_init char g @ 15;

__regvar __no_init char P @ 14;

void main(void)

{   g++;

    P++;

 }
```

在工程选项里c/c++ complier>code里打开要使用的寄存器R14-R15。



编译结果就如下，看看是不是直接使用了寄存器做为数据应用

```
//    4 void main(void)

main:

        CFI Block cfiBlock0 Using cfiCommon0

        CFI Function main

//    5 {   g++;
```

```
        REQUIRE ?Register_R14_is_global_regvar
        REQUIRE ?Register_R15_is_global_regvar
        INC     R15
//    6     P++; }
        INC     R14
        RET
```

注意：定义在寄存器里变量不能带有初始值。最好不要使用超过 9 个寄存器变量，不然可能引起潜在的危险，因为建立库的时候没有锁定任何寄存器。

5．2．定义变量的绝对地址.没有特性的变量是随机分配的，要给变量分配地址必须加以特性修饰注意在定义地址的时候千万不要和片内寄存器地址重合了。

5.2.1 定义没有存储特性的绝对地址变量必须加__no_init 或者 const 对象特性

```
__no_init char t @ 0x65;//定义在 I/O 地址以外
const char t @ 0x65;//定义只读变量的地址
```

例：

```
#include<iom8.h>
 __no_init char u @ 0x65 ;
void main(void)
{u++;}
```

对应汇编：

```
void main(void)
   \                    main:
              {u++;}
   \   00000000   E6E5             LDI     R30, 101
   \   00000002   E0F0             LDI     R31, 0
   \   00000004   8100             LD      R16, Z
   \   00000006   9503             INC     R16
   \   00000008   8300             ST      Z, R16
   \   0000000A   9508             RET
```

5．2．2 带存储特性的关键字定义变量的绝对地址__io,__ext_io 定义变量在 i/o 空间

```
    #include<iom8.h>
```

```
 __io char u @ 0x65 ;
void main(void)
{u++;}
```

对应汇编：

```
void main(void)
   \                     main:
           {u++;}
 \   00000000   91000065          LDS     R16, 101
   \   00000004   9503              INC     R16
   \   00000006   93000065          STS     101, R16
   \   0000000A   9508              RET
```

从 5.2.1 和 5.2.2 对比，发现用 5.2.2 方法定义代码小多了。

5.3.关键字 volatile 保证从最原始的位置读取变量。在 IAR 编译器里,除了__no_init 和__root 定义的变量外，其他的类型的变量都包含有 volatile 和__no_init 特性。

5．4．强大为位操作：

5．4．1． 在 c 语言里对位的操作如一般如下：

```
PORTB|=（1<<2）;//置 PORTB 的第 2 位=1
PORTB&=~（1<<2）;//置 PORTB 的第 2 位=0
PORTB^|=（1<<2）;//取反 PORTB 的第 2 位
While(PORTB&(1<<2));//判断 1
While(!(PORTB&(1<<2)));//判断为 0
```

5．4．2． IAR 编译器对位的支持更强大，除了上面的方法外还有以下更简单的操作方法：

```
PORTB_ Bit2=1; //置 PORTB 的第 2 位=1
PORTB_ Bit2=0; //置 PORTB 的第 2 位=0
PORTB_ Bit2=~ PORTB_ Bit2;//取反 PORTB 的第 2 位
While(PORTB_ Bit2);或者 while(PORTB_Bit2==1);//判断 1
while(PORTB_ Bit2==0);//判断 0
PORTC_Bit4=PORTB_Bit2;//把 PORTB 的第 2 位传送到 PORTC 的第 4 位
```

5．4．3．位变量定义：

由于 iar 使用了扩展语言，它对位域的支持变为最小为 char 类型，我们可以很方便地用

来定义位变量。

采用结构体来定义位变量：

```
struct
{
unsigned char bit0:1;
unsigned char bit1:1;
unsigned char bit2:1;
unsigned char bit3:1;
unsigned char bit4:1;
unsigned char bit5:1;
unsigned char bit6:1;
unsigned char bit7:1;
}t;
```

然后就可以用以下位变量了。

```
t.bit0=1;
t.bit0=~t.bit0;
```

但是采用以上结构体做出来的位变量只可以访问 t 的位，不能够直接访问变量 t，和标准的 IAR 位操作也不一样。采用联合体来定义效果更佳。

```
#include<iom8.h>
union
{
unsigned char t;
struct
{unsigned char t_bit0:1,
              t_bit1:1,
              t_bit2:1,
              t_bit3:1,
              t_bit4:1,
              t_bit5:1,
              t_bit6:1,
```

```
                t_bit7:1;
};
};

void main(void)
{t_bit0=1;//访问变量 t 的位
t_bit0=~t_bit0;
PORTB=t;//直接访问变量 t
}
```

位变量也可以直接定义在工作寄存器里。

5.4.4 *bool* 数据类型在C++语言里是默认支持的。如果你在C代码的头文件里包含 *stdbool.h*, *bool* 数据类型也可以使用在C语言里。也可以使用布尔值 *false* 和 *true*。不过是占用8位1个字节。

```
#include<iom8.h>
#include<stdbool.h>
bool y=0;//定义位变量
void main(void)
{y=!y;//取反位变量
PORTB_Bit3=y;//传递位变量
}
```

第六章　　函数

6．1．中断函数：

在IAR编译器里用关键字来__interrupt来定义一个中断函数。用#pragma vector来提供中断函数的入口地址

```
#pragma vector=0x12//定时器0溢出中断入口地址
 __interrupt void time0(void)
{
。。。。。。。。。。。
}
```

上面的入口地址写成#pragma vector=TIMER0_OVF_vect更直观，每种中断的入口地址在头文件里有描述。函数名称time0可以为任意名称。中断函数会自动保护局部变量，但不会保护全局变量。

6.2.内在函数也可以称为本征函数

编译器自己编写的能够直接访问处理器底层特征的函数。在 intrinsics.h 中有描述完整类型在 comp_a90.h 里有进一步的简化书写方式

6．2．1延时函数，以周期为标准

__delay_cycles(unsigned long);

如果处理器频率为1M，延时100us,如下：

__delay_cycles(100);

当然你也可以对该函数进行修改：

#define CPU_F 1000000

*#define delay_us (unsigned long) __delay_cycles((unsigned long)*CPU_F)*

*#define delay_ms (unsigned long) __delay_cycles((unsigned long)*CPU_F/1000)*

6．2．2中断指令

__disable_interrupt( );//插入CLI指令，也可以用_CLI()；也可以SREG_Bit7=0;

__enable_interrupt( );// 插入SEI指令，也可以用_SEI()；也可以SREG_Bit7=1;

其实对于状态字的置位和清零只有BSET S 和BCLR S两条指令。像SEI不过是BSET 7；的另一个名字而已。AVR指令中还有很多类似的现象，如：ORI 和 SBR 指令完全一样，号称130多条指令的AVR其实没有那么多指令的。

6．2．2从FLASH空间指定地址读取数据

__extended_load_program_memory(unsigned char __farflash *);

__load_program_memory(unsigned char __flash *);

该条指令以及正确的使用方法在4．5．*flash* 操作宏函数里详细讲解，这里不再重复

6．2．2乘法函数

__fracdtional_multiply_signed(signed char, signed char);

__fractional_multiply_signed_with_unsigned(signed char, unsigned char);

__fractional_multiply_unsigned(unsigned char, unsigned char);

//以上为定点小数乘法

__multiply_signed(signed char, signed char);//有符号数乘法

__multiply_signed_with_unsigned(signed char, unsigned char);

//有符号数和无符号数乘法

__multiply_unsigned(unsigned char, unsigned char);//无符号数乘法

6．2．4 半字节交换指令

__swap_nibbles(unsigned char);

6．2．3 MCU 控制指令

__no_operation();//空操作指令

_NOP();

__sleep(); //休眠指令

_SLEEP();

__watchdog_reset();//看门狗清零

_WDR();

第七章　头文件

avr_macros.h里面包含了读写16位寄存器的简化书写，和几个位操作函数

comp_a90.h对大量的内在函数做了简要书写

ina90.h包含"inavr.h" "comp_A90.h"文件

intrinsics.h内在函数提供最简单的操作处理器底层特征。休眠，看门狗，FLASH函数。

iomacro.H I/O寄存器定义文件样本。

iom8.h　包含I/O等寄存器定义

第八章　嵌入汇编语言

1. 在线汇编：使用asm或者__asm,推荐使用__asm。

```
#include<iom8.h>
void main()
{
asm(
    "NOP \n"
    "CLH \n"
    "OR R16,R17 \n"
     );
```

}

不过IAR提供了完全可以访问底层的函数，建议不要频繁使用汇编。C语言和汇编接口参阅C/C++编译器参考指南

chenjianlin

2006 年 10 月 17 日