TICE99 C Compiler

特殊用法

TICE99 C Compiler 注意事項

位元变量（含struct Bit Field ）

一般变量

#program的使用

中断函数的使用

inline asm函数的使用

C和ASM的混合模式

位元变量

有别于ANSI C，TM57的C编译程序支援位变量和Bit Field，但只能宣告在全域变数，不可宣告为本地变量或const变量。建议由编译器安排位址，避免造成位址冲突的情况。

位变量宣告：

位变量宣告的方式，可分为非指定与指定位址，实际情况如下：

- 非指定位址：C编译器会自动把位址安排在可以位操作

例如：

```
bit B1, B2;
main(){….}
```

- 指定位址：须定址在可以位操作区域（**请注意各晶片手册说明**）

例如：

```
bit B1@0x30@0:FPLANE;      // 只可指定在FPLANE位操作的区域
bit B2:bank1;              // 要有bank1的位址才有用
main(){….}
```

位元变量

另外还可以用union和struct的宣告方式，让位变量查看时，更整齐。

- 范例：用union宣告

方式1：

```
union BITVAR{
        unsigned char B0:1;
        unsigned char B1:1;
        unsigned char B2:1;
        unsigned char B3:1;
        unsigned char B4:1;
        unsigned char B5:1;
        unsigned char B6:1;
        unsigned char B7:1;
} BITVAR_FLAG@0x30:FPLANE;
main(){....}
```

方式2：

```
struct BITVAR{
                unsigned char B0:1;
                unsigned char B1:1;
                unsigned char B2:1;
                unsigned char B3:1;
                unsigned char B4:1;
                unsigned char B5:1;
                unsigned char B6:1;
                unsigned char B7:1;
        };
struct BITVAR FLAG_UN0@0x30:FPLANE;
main(){….}
```

一般变量

一般变量的宣告，可以指定所宣告的变量是在FPLANE或是RPLANE，也有分为非指定和指定位址的宣告方式。

- **非指定位址的方式：**

```
unsigned char F_Var;                // 预设在FPLANE的区域
unsigned char R_Var:RPLANE;         // 只可在宣告于全域变量
```

- **指定位址的方式：（只可宣告于全域变量）**

```
unsigned char F_Var@0x30:FPLANE;
unsigned char R_Var@0x40:RPLANE;
```

在有Bank1的晶片下，也可以指定变量是在Bank1区域里，但无法指定位址。

```
unsigned char FB1_Var:bank1;
```

※ RPLANE的位址在有些晶片上只能写不能读，详情请参考各晶片手册

预处理#pragma

#pragma预处理的指令，在應用上有幾種方式，指定函數存在ROM的位址和指定const数组存在ROM的位址。

- 指定函数：

```
#define Max   6
char f1(char, char);
void main()
 {
   int c = 0;
   c = f1(5, 6);
}
int f1(int i, int j)@0x0100
 {   return (i + j);   }
```

- 指定const数组：

```
#pragma tableromaddr (0x200)
const int array[]={1,2,3,4,5,6,7};    /* 数值的部份
                                         会从0x0201安排，
                                         因 为需要多插入一个
                                         addwf 0x02,1 的指令*/

#pragma tableromaddr (off)
main(){....}
```

中断函数的使用

中断函数的使用，目前可经由二种方式加入，一种是在新建专案时，就从Insert Interrupt Function的Detail裡加入，或是在撰写程序时，用手动的方式自己输入。

- **手动输入方式：**

  使用void Interrupt的key word来宣告中断函数的名称和中断入口，

  宣告格式如下：

  void Interrupt <Function_Name>(void) @<Interrupt_Vector_address>

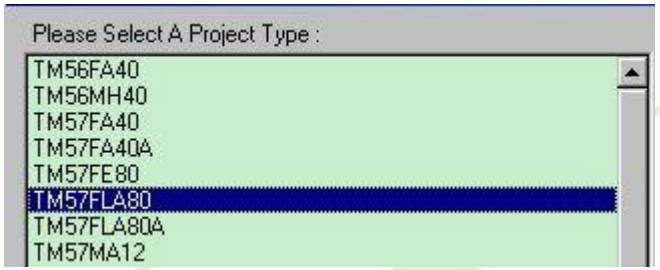
  依TM57FLA80的TM0中断入口为例：

```
void Interrupt TM0(void) @0x01
{……}
main()
{……}
```

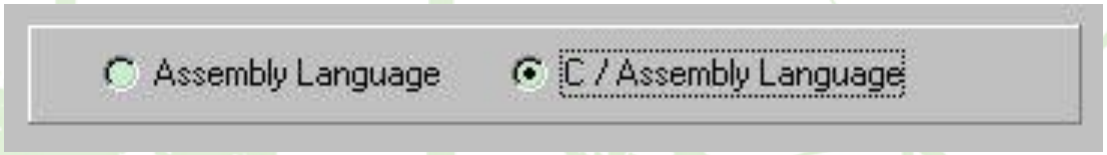
中断函数的使用

- 新增专案时加入中断函数：

1. 先选择使用那颗晶片，此处以TM57FLA80为例：



2. 在晶片列表下方，把专案的类型选择为C / Assembly Language：



中断函数的使用

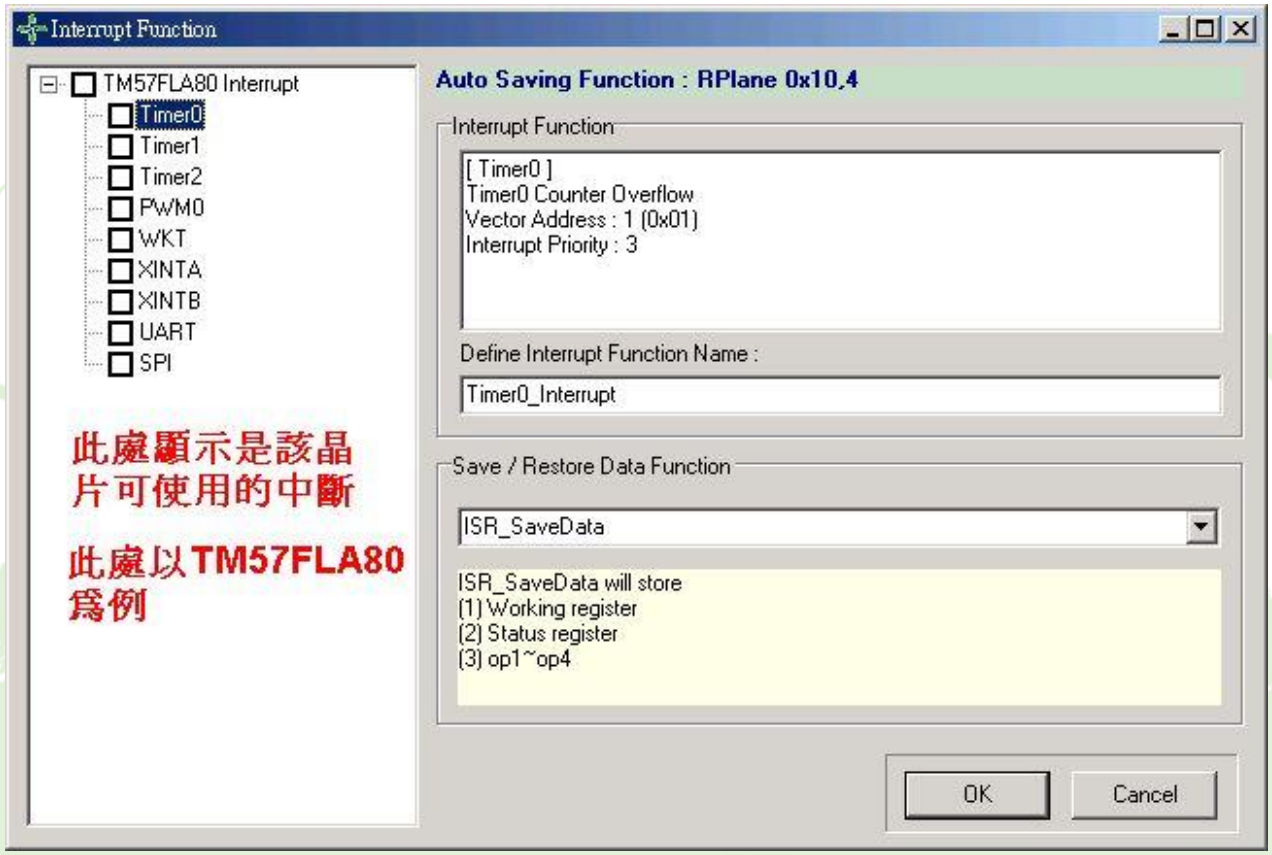
- 新增专案时加入中断函数：

3. 然后把最下面的Insert Interrupt Function的选项：

中断函数的使用

- 新增专案时加入中断函数：

4. 产生的视窗中，可看出TM57FLA80可使用的中断方式，使用者可自行选择自己需要的：

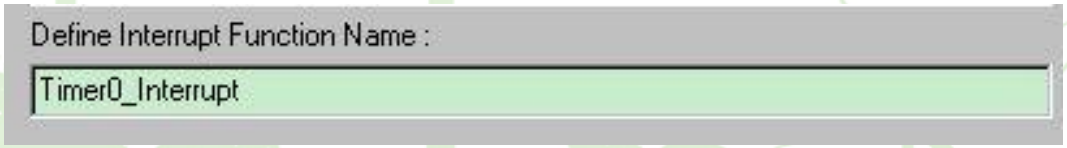


中断函数的使用

- 新增专案时加入中断函数：

5. 右侧最上面所显示的是中断产生的要件、向量入口和优先性：

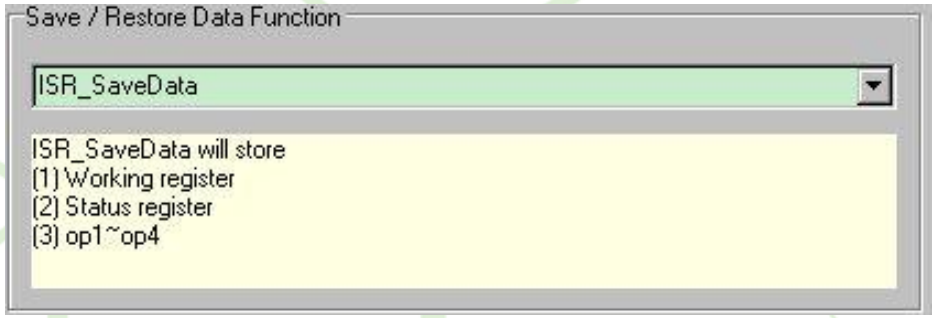
6. 中间部份显示的是预设的中断函数的名称，可自行修改名称：

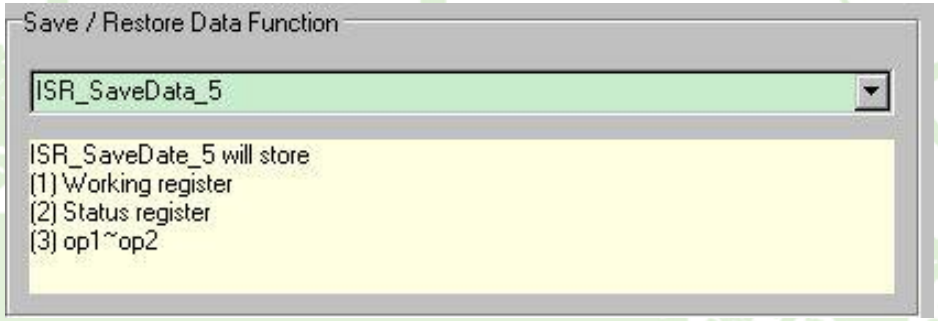


中断函数的使用

新增专案时加入中断函数：

7. 右侧最上面所显示的是中断右侧下方是选择在进入中断函数时，避免一些运算暂存位址的值被修改，所提供的中断保存方式：(注意：每颗晶片不同，保存中所需要用到的暂存位址有所差异)





中断函数的使用

新增专案时加入中断函数：

8. 按下OK键后，就会回到新增专案的视窗，输入专案名称并且选择要储存的目录后，按下OK键，就会把中断函数写入main.c的程序档里。

*main.c

```
1  #include "tenx_TM57FLA80.h"
2
3
4  // Function prototype of asm codes
5  void ISR_SaveData(void);
6  void ISR_RestoreData(void);
7  //---------------------------------------
8  // Timer0 Counter Overflow //
9  void interrupt Timer0_Interrupt(void)@0x01
10 {
11          ISR_SaveData();         // Save operation data
12          // [TODO] write interrupt function here
13
14          ISR_RestoreData();      // restore operation data
15 }
16
17 //---------------------------------------
18 //main function
19 void main(void)
20 {
21 }
22 //---------------------------------------
```

※ 注意：中断保存函数若在只有单一中断向量入口的晶片上，会采用MACRO的方式插入以减少堆栈的使用。

inline asm函数

搭配asm函数的使用，就可以在C文件中直接撰写纯汇编的程序，定义的方式，如下：

asm(<string literal>[,optional parameters]);

而在字符串中可以包含下表列的格式符号。

格式符号	说明
%b	8位数字值
%w	16位数字值
%l	32位数字值
%v	(全域)变量或函数的汇编名称
%o	(区域)变量堆栈偏移量
%%	%符号

inline asm函数

可以搭配c语言的#define的预处理来使用，例如：

```
#define   OFFS   0x23
main()
{
   unsigned char ch;
   asm("MOVLW %b", OFFS);
   asm("MOVWF %o", ch);          // 因为ch是区域变量，所以用%o的格式符号
}
```

在此有一点需要注意的地方，因为大部份的MCU最大的位址只有0x7F所以数字值如果超出0x7F的话，需要改用『 %w』的格式符号，否则Compiler会有错误。

```
#define   OFFS    0x89
main()
{
   unsigned char ch;
   asm("MOVLW %w", OFFS);    //因为OFFS所定义的值超过0x7F，所以需要使用%w
   asm("MOVWF %o",  ch);
}
```

C和ASM的混用

撰写单晶片程序时，有时候为了提高程序执行效率，会使用C文件和汇编的混用方式，下列将进行一些使用的方式。

- **C文件调用无需传入参数和无回传值的汇编函数：**
  - C文件：

```
1 void add();
2 main()
3 {
4     add();
5 }
```

  - 汇编文件：

```
1  .AUTOIMPORT on
2  .DEBUGINFO  on
3  .EXPORT     _add
4  .DECLFUNC   sum(0,0)
5
6  Var  .equ    30H
7
8  .CODE
9  .PROC _add
10
11   MOVLW 05H
12   MOVWF Var
13   MOVLW 0Ah
14   ADDWF Var,1
15 .ENDPROC
16
17
```

※注意：混用的情况只能在C专案的前提下使用，并且需要把汇编文件加入专案列表中。

C和ASM的混用

- C文件调用需传入参数和有回传值的汇编函数：(汇编函数的局部变量位址设定顺序是”由下至上”，而传入汇编函数中的参数变量的位址顺序是”由右至左”，add_PARAM和add_LOCAL的位址，编译器会自行安排)
  - C文件：

```
1  #define uchar unsigned char
2  uchar i, j, k;
3
4  uchar add(uchar, uchar);
5
6  main()
7  {
8      i = 10;
9      j = 20;
10     k = add(i,j);
11 }
```

  - 汇编文件：

```
1  .AUTOIMPORT on
2  .DEBUGINFO  on
3  .EXPORT     _add
4  .DECLFUNC   add(0,2)
5
6
7  .CODE
8  .PROC _add
9     MOVFW add_PARAM+0
10    ADDWF add_PARAM+1,0
11    MOVWF op2
12 .ENDPROC
13
14
```

※注意：在有回传值的情况下，回传值必须固定放在op2的位址，op2的位址编译器会自行安排。

※有关位址安排详细介绍可查看UM-TM57XX_C_Compiler.pdf手册中第75页介绍。

C和ASM的混用

在此说明关于.DECLFUNC的中，所代表的数值关系。在下图中所看到的.DECLFUNC add(0,2)的部份，就是告知编译器，在这个函数中，局部的变量占0个Byte，带入的参数占了2个Bytes。

```
1 .AUTOIMPORT on
2 .DEBUGINFO  on
3 .EXPORT     _add
4 .DECLFUNC   add(0,2)
```

※ 此部份务必要把位址的数量填写清楚，否则会导致编译器位址安排错误，造成不可预期的情况。

C和ASM的混用

在此，我们用下面的范例来做位址数量填写的说明。

- C文件：

```c
#define uchar unsigned char
uchar i, j, k;

uchar add(uchar a, uchar b);

main()
{
    i = 10;
    j = 20;
    k = add(i,j);
}
```

- 汇编文件：

```
.AUTOIMPORT on
.DEBUGINFO  on
.EXPORT     _add
.DECLFUNC   add(2,2)


.CODE
.PROC _add
   MOVFW add_PARAM+0
   MOVWF add_LOCAL+0
   MOVFW add_PARAM+1
   MOVWF add_LOCAL+1

   MOVFW add_LOCAL+0
   ADDWF add_LOCAL+1,0
   MOVWF op2
.ENDPROC
```

在C程序中的带入参数分别是uchar i和uchar j，共传入2个Bytes，则在.DECLFUNC的括号中的第二个数值就要填入2；而在汇编程序中用到了2个局部变量(add_LOCAL+0，add_LOCAL+1)，则第一个数值也要填入2。

※ 建议尽量做用inline_asm的方式来撰写，以免位址计算错误，造成不可预期的错误情况。