#pragma指令用法汇总和解析

1. **一般格式**

其格式一般为: #pragma Para。其中Para为参数，下面来看一些常用的参数

常用参数

1. **message 参数。**

它能够在编译信息输出窗口中输出相应的信息，这对于源代码信息的控制是非常重要的。其使用方法为：

#pragma message(“消息文本”)

当编译器遇到这条指令时就在编译输出窗口中将消息文本打印出来。

当我们在程序中定义了许多宏来控制源代码版本的时候，我们自己有可能都会忘记有没有正确的设置这些宏，此时我们可以用这条

指令在编译的时候就进行检查。假设我们希望判断自己有没有在源代码的什么地方定义了\_X86这个宏可以用下面的方法

#ifdef \_X86

#pragma message(“\_X86 macro activated!”)

#endif

当我们定义了\_X86这个宏以后，应用程序在编译时就会在编译输出窗口里显示“\_X86 macro activated!”。我们就不会因为不记得自己定义的一些特定的宏而抓耳挠腮了

1. **code\_seg**

另一个使用得比较多的#pragma参数是code\_seg。格式如：

#pragma code\_seg( [ [ { push | pop}, ] [ identifier, ] ] [ "segment-name" [, "segment-class" ] )

该指令用来指定函数在.obj文件中存放的节,观察OBJ文件可以使用VC自带的dumpbin命令行程序，函数在.obj文件中默认的存放节为.text节

如果code\_seg没有带参数的话，则函数存放在.text节中

**push (可选参数)**：将一个记录放到内部编译器的堆栈中,可选参数可以为一个标识符或者节名

**pop (可选参数)**：将一个记录从堆栈顶端弹出,该记录可以为一个标识符或者节名

**identifier (可选参数)**：当使用push指令时,为压入堆栈的记录指派的一个标识符,当该标识符被删除的时候和其相关的堆栈中的记录将被弹出堆栈

**"segment-name" (可选参数)**：表示函数存放的节名

例如:

//默认情况下,函数被存放在.text节中

void func1() { // stored in .text

}

//将函数存放在.my\_data1节中

#pragma code\_seg(".my\_data1")

void func2() { // stored in my\_data1

}

//r1为标识，将函数放入.my\_data2节中

#pragma code\_seg(push, r1, ".my\_data2")

void func3() { // stored in my\_data2

}

int main() {

}

1. **#pragma once**

这是一个比较常用的指令,只要在头文件的最开始加入这条指令就能够保证头文件被编译一次

1. **#pragma hdrstop**

表示预编译头文件到此为止，后面的头文件不进行预编译。

BCB可以预编译头文件以加快链接的速度，但如果所有头文件都进行预编译又可能占太多磁盘空间，所以使用这个选项排除一些头文件。

有时单元之间有依赖关系，比如单元A依赖单元B，所以单元B要先于单元A编译。你可以用#pragma startup指定编译优先级，

如果使用了#pragma package(smart\_init)，BCB就会根据优先级的大小先后编译。

1. **#pragma warning**

该指令允许有选择性的修改编译器的警告消息的行为。

指令格式如下:

#pragma warning( warning-specifier : warning-number-list [; warning-specifier : warning-number-list...])

#pragma warning( push[ ,n ] )

#pragma warning( pop )

主要用到的警告表示有如下几个:

**once**:只显示一次(警告/错误等)消息

**default**:重置编译器的警告行为到默认状态

**1,2,3,4**:四个警告级别

**disable**:禁止指定的警告信息

**error:**将指定的警告信息作为错误报告

如果大家对上面的解释不是很理解,可以参考一下下面的例子及说明

#pragma warning( disable : 4507 34; once : 4385; error : 164 )

等价于：

#pragma warning(disable:4507 34) // 不显示4507和34号警告信息

#pragma warning(once:4385) // 4385号警告信息仅报告一次

#pragma warning(error:164) // 把164号警告信息作为一个错误。

同时这个pragma warning 也支持如下格式：

#pragma warning( push [ ,n ] )

#pragma warning( pop )

这里n代表一个警告等级(1---4)。

#pragma warning( push ) 保存所有警告信息的现有的警告状态。

#pragma warning( push, n) 保存所有警告信息的现有的警告状态，

并且把全局警告等级设定为n。

#pragma warning( pop ) 从栈中弹出最后一个警告信息，

在入栈和出栈之间所作的一切改动取消。例如：

#pragma warning( push )

#pragma warning( disable : 4705 )

#pragma warning( disable : 4706 )

#pragma warning( disable : 4707 )

#pragma warning( pop )

在这段代码的最后，重新保存所有的警告信息(包括4705，4706和4707)

在使用标准C++进行编程的时候经常会得到很多的警告信息,而这些警告信息都是不必要的提示,

所以我们可以使用#pragma warning(disable:4786)来禁止该类型的警告

在vc中使用ADO的时候也会得到不必要的警告信息,这个时候我们可以通过

#pragma warning(disable:4146)来消除该类型的警告信息

1. **#pragma resource**

#pragma resource "\*.dfm"表示把\*.dfm文件中的资源加入工程。\*.dfm中包括窗体外观的定义。

1. **pragma comment(...)**

该指令的格式为：

#pragma comment( "comment-type" [, commentstring] )

该指令将一个注释记录放入一个对象文件或可执行文件中,

comment-type(注释类型):可以指定为五种预定义的标识符的其中一种

五种预定义的标识符为:

**compiler**: 将编译器的版本号和名称放入目标文件中，本条注释记录将被编译器忽略。如果你为该记录类型提供了commentstring参数,编译器将会产生一个警告。

例如:#pragma comment( compiler )

exestr: 将commentstring参数放入目标文件中，在链接的时候这个字符串将被放入到可执行文件中。当操作系统加载可执行文件的时候，该参数字符串不会被加载到内存中。但是，该字符串可以被dumpbin之类的程序查找出并打印出来，你可以用这个标识符将版本号码之类的信息嵌入到可执行文件中。

**lib**: 这是一个非常常用的关键字，用来将一个库文件链接到目标文件中。

常用的lib关键字，可以帮我们连入一个库文件。

例如:

#pragma comment(lib, "user32.lib")

该指令用来将user32.lib库文件加入到本工程中

**linke**r: 将一个链接选项放入目标文件中，你可以使用这个指令来代替由命令行传入的或者在开发环境中设置的链接选项，你可以指定/include选项来强制包含某个对象。

例如：#pragma comment(linker, "/include:\_\_mySymbol")

你可以在程序中设置下列链接选项

/DEFAULTLIB

/EXPORT

/INCLUDE

/MERGE

/SECTION

这些选项在这里就不一一说明了,详细信息请看msdn!

**user**: 将一般的注释信息放入目标文件中commentstring参数包含注释的文本信息，这个注释记录将被链接器忽略。

例如:

#pragma comment( user, "Compiled on " \_\_DATE\_\_ " at " \_\_TIME\_\_ )

1. **#pragma disable**

在函数前声明，只对一个函数有效。该函数调用过程中将不可被中断。一般在C51中使用较多。

1. **#pragma data\_seg**

**应用一：单应用程序。**

有的时候我们可能想让一个应用程序只启动一次，就像单例模式(singleton)一样，实现的方法可能有多种，这里说说用#pragma data\_seg来实现的方法，很是简洁便利。

应用程序的入口文件前面加上

#pragma data\_seg("flag\_data")

int app\_count = 0;

#pragma data\_seg()

#pragma comment(linker,"/SECTION:flag\_data,RWS")

然后程序启动的地方加上

if(app\_count>0) // 如果计数大于0，则退出应用程序。

{

//MessageBox(NULL, "已经启动一个应用程序", "Warning", MB\_OK);

//printf("no%d application", app\_count);

return FALSE;

}

app\_count++;

1. **#pragma region**

#pragma region是Visual C++中特有的预处理指令。它可以让你折叠特定的代码块，从而使界面更加清洁，便于编辑其他代码。折叠后的代码块不会影响编译。你也可以随时展开代码块以进行编辑等操作。

格式：#pragma region name#pragma endregion comment

使用示例如下：

#pragma region Variables

HWND hWnd;

const size\_t Max\_Length = 20;

//other variables

#pragma endregion This region contains global variables.

如上边所示，需要折叠的代码必须包含在#pragma region和#pragma endregion之间。#pragma region和#pragma endregion之后可以添加一些用来注释的文字。当你折叠代码块后，这些文字会显示在折叠的位置。

折叠代码块的方法：如同Visual C++中折叠函数、类、命名空间，当代码被包含在如上所述的指令之间后，#pragma region这一行的左边会出现一个“-”号，单击以折叠内容，同时“-”号会变成“+”号，再次单击可以展开代码块。

此预编译指令在Visual Studio 2005及以上版本可以使用。但是在Visual Studio 2005中，当#pragma region之后包含类似“1st”这类的文字，会导致“error C2059: syntax error : 'bad suffix on number'”的编译错误。避免使用数字或者将数字与字母分离可以解决这个问题，

1. **#pragma pack(n)**

**应用二：通过#pragma pack(n)改变C编译器的字节对齐方式**

在C语言中，结构是一种复合数据类型，其构成元素既可以是基本数据类型（如int、long、float等）的变量，也可以是一些复合数据类型（如数组、结构、联合等）的数据单元。在结构中，编译器为结构的每个成员按其自然对界（alignment）条件分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同。

例如，下面的结构各成员空间分配情况：

struct test

{

char x1; // 偏移地址为0

short x2;// 偏移地址为[2，3]

float x3;// 偏移地址为[4，7]

char x4; // 偏移地址为8

};

结构的第一个成员x1，其偏移地址为0，占据了第1个字节。第二个成员x2为short类型，其起始地址必须2字节对界，因此，编译器在x2和x1之间填充了一个空字节。结构的第三个成员x3和第四个成员x4恰好落在其自然对界地址上，在它们前面不需要额外的填充字节。在test结构中，成员x3要求4字节对界，是该结构所有成员中要求的最大对界单元，因而test结构的自然对界条件为4字节，编译器在成员x4后面填充了3个空字节。整个结构所占据空间为12字节。更改C编译器的

缺省字节对齐方式在缺省情况下，C编译器为每一个变量或是数据单元按其自然对界条件分配

空间。一般地，可以通过下面的方法来改变缺省的对界条件：

· 使用伪指令#pragma pack (n)，C编译器将按照n个字节对齐。

· 使用伪指令#pragma pack ()，取消自定义字节对齐方式。

另外，还有如下的一种方式：

· \_\_attribute((aligned (n)))，让所作用的结构成员对齐在n字节自然边界上。如果结构中有成员的长度大于n，则按照最大成员的长度来对齐。

· \_\_attribute\_\_ ((packed))，取消结构在编译过程中的优化对齐，按照实际占用字节数进行对齐。

以上的n = 1, 2, 4, 8, 16... 第一种方式较为常见。

**应用三：定义协议结构：**

在网络协议编程中，经常会处理不同协议的数据报文。一种方法是通过指针偏移的方法来得到各种信息，但这样做不仅编程复杂，而且一旦协议有变化，程序修改起来也比较麻烦。在了解了编译器对结构空间的分配原则之后，我们完全可以利用这一特性定义自己的协议结构，通过访问结构的成员来获取各种信息。这样做，不仅简化了编程，而且即使协议发生变化，我们也只需修改协议结构的定义即可，其它程序无需修改，省时省力。下面以TCP协议首部为例，说明如何定义协议结构。

其协议结构定义如下：

#pragma pack(1) // 按照1字节方式进行对齐

struct TCPHEADER

{

short SrcPort; // 16位源端口号

short DstPort; // 16位目的端口号

int SerialNo; // 32位序列号

int AckNo; // 32位确认号

unsigned char HaderLen : 4; // 4位首部长度

unsigned char Reserved1 : 4; // 保留6位中的4位

unsigned char Reserved2 : 2; // 保留6位中的2位

unsigned char URG : 1;

unsigned char ACK : 1;

unsigned char PSH : 1;

unsigned char RST : 1;

unsigned char SYN : 1;

unsigned char FIN : 1;

short WindowSize; // 16位窗口大小

short TcpChkSum; // 16位TCP检验和

short UrgentPointer; // 16位紧急指针

};

#pragma pack() // 取消1字节对齐方式

#pragma pack规定的对齐长度，实际使用的规则是：

结构，联合，或者类的数据成员，第一个放在偏移为0的地方，以后每个数据成员的对齐，按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行。

也就是说，当#pragma pack的值等于或超过所有数据成员长度的时候，这个值的大小将不产生任何效果。

而结构整体的对齐，则按照结构体中最大的数据成员 和 #pragma pack指定值 之间，较小的那个进行。

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。

**1 .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

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)

**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。

**3.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 \*);

**4.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

**5.半字节交换指令**

\_\_swap\_nibbles(unsigned char);

**6.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()；

注意：在实现过程中可能涉及到.XCL连接文件的更改，请保存好原来的.XCL文件！

1.打开相应的\*c.xcl文件,用"-Z(CONST)段名=程序定位的目标段-FFDF"定义段的起始地址.

2.在自己的C程序中用#pragma constseg(段名)定位自己的程序

3.结束后恢复编译器的默认定位#pragma default

IAR 1.26b环境下：

1、将常量数组放在FLASH段自定议的MYSEG段中

原来的MSP430F149 XCL文件如下：

// Constant data

-Z(CONST)DATA16\_C,DATA16\_ID,DIFUNCT,CHECKSUM=1100-FFDF

如果想从中分出一部分做数据存储区，做如下修改：

-Z(CONST)DATA16\_C,DATA16\_ID,DIFUNCT,CHECKSUM=1500-FFDF //将1100-14FF从ROM中分出存储arry数

组

-Z(CONST)MYSEG=1100-14FF区间大小可自行决定

在程序中描写如下即可：

#pragma memory = constseg(MYSEG) //在.XCL文件中修改

char arry[]={1,2,3,4,5,6,7};

#pragma memory = default

2、将变量放入所命名的段

在XCL文件中开辟一段MYSEG段，如上所述

#pragma memory = dataseg(MYSEG)

char i;

char j;

int k;

#pragma memory = default

IAR3.10A环境下

xcL文件的更改方法如上

数据定位方法如下三种

1、\_\_no\_init char alpha @ 0x0200;

2、#pragma location=0x0202

const int beta;

3、const int gamma @ 0x0204 = 3;

或；

1、\_\_no\_init int alpha @ "MYSEGMENT"; //MYSEGMENT段可在XCL中开辟

2、#pragma location="MYSEGMENT"

const int beta;

3、const int gamma @ "MYSEGMENT" = 3;

函数定位如下面两种写法

1、

void g(void) @ "MYSEGMENT" //MYSEGMENT段可在XCL中开辟

{

}

2、

#pragma location="MYSEGMENT"

void h(void)

{

}