

## 第十一章 PIC 单片机的 C 语言编程

### 11.1 PIC 单片机 C 语言编程简介

用 C 语言来开发单片机系统软件最大的好处是编写代码效率高、软件调试直观、维护升级方便、代码的重复利用率高、便于跨平台的代码移植等等，因此 C 语言编程在单片机系统设计中已得到越来越广泛的运用。针对 PIC 单片机的软件开发，同样可以用 C 语言实现。

但在单片机上用 C 语言写程序和 PC 机上写程序绝对不能简单等同。现在的 PC 机资源十分丰富，运算能力强大，因此程序员在写 PC 机的应用程序时几乎不用关心编译后的可执行代码在运行过程中需要占用多少系统资源，也基本不用担心运行效率有多高。写单片机的 C 程序最关键的一点是单片机内的资源非常有限，控制的实时性要求又很高，因此，如果没有对单片机体系结构和硬件资源作详尽的了解，以笔者的愚见认为是无法写出高质量实用的 C 语言程序。这就是为什么前面所有章节中的示范代码全部用基础的汇编指令实现的原因，希望籍此能使读者对 PIC 单片机的指令体系和硬件资源有深入了解，在这基础之上再来讨论 C 语言编程，就有水到渠成的感觉。

本书围绕中档系列 PIC 单片机来展开讨论，Microchip 公司自己没有针对中低档系列 PIC 单片机的 C 语言编译器，但很多专业的第三方公司有众多支持 PIC 单片机的 C 语言编译器提供，常见的有 Hitech、CCS、IAR、Bytecraft 等公司。其中笔者最常用的是 Hitech 公司的 PICC 编译器，它稳定可靠，编译生成的代码效率高，在用 PIC 单片机进行系统设计和开发的工程师群体中得到广泛认可。其正式完全版软件需要购置，但在其网站上有限时的试用版供用户评估。另外，Hitech 公司针对广大 PIC 的业余爱好者和初学者还提供了完全免费的学习版 PICC-Lite 编译器套件，它的使用方式和完全版相同，只是支持的 PIC 单片机型号限制在 PIC16F84、PIC16F877 和 PIC16F628 等几款。这几款 Flash 型的单片机因其所具备的丰富的片上资源而最适用于单片机学习入门，因此笔者建议感兴趣的读者可从 PICC-Lite 入手掌握 PIC 单片机的 C 语言编程。

在此列出几个主要的针对 PIC 单片机的 C 编译器相关连接网址，供读者参考：

Hitech-PICC：[www.htsoft.com](http://www.htsoft.com)

IAR：[www.iar.com](http://www.iar.com)

CCS：[www.ccsinfo.com/picc.shtml](http://www.ccsinfo.com/picc.shtml)

ByteCraft：[www.bytecraft.com/mpccaps.html](http://www.bytecraft.com/mpccaps.html)

本章将介绍 Hitech-PICC 编译器的一些基本概念，由于篇幅所限将不涉及 C 语言的标准语法和基础知识介绍，因为在这些方面都有大量的书籍可以参考。重点突出针对 PIC 单片机的特点而所需要特别注意的地方。

### 11.2 Hitech-PICC 编译器

PICC 基本上符合 ANSI 标准，除了一点：它不支持函数的递归调用。其主要原因是因为 PIC 单片机特殊的堆栈结构。在前面介绍 PIC 单片机架构时已经详细说明了 PIC 单片机

中的堆栈是硬件实现的，其深度已随芯片而固定，无法实现需要大量堆栈操作的递归算法；另外在 PIC 单片机中实现软件堆栈的效率也不是很高，为此，PICC 编译器采用一种叫做“静态覆盖”的技术以实现 C 语言函数中的局部变量分配固定的地址空间。经这样处理后产生出的机器代码效率很高，按笔者实际使用的体会，当代码量超过 4K 字后，C 语言编译出的代码长度和全部用汇编代码实现时的差别已经不是很大（<10%），当然前提是在整个 C 代码编写过程中须时时处处注意所编写语句的效率，而如果没有对 PIC 单片机的内核结构、各功能模块及其汇编指令深入了解，要做到这点是很难的。

### 11.3 MPLAB-IDE 内挂接 PICC

PICC 编译器可以直接挂接在 MPLAB-IDE 集成开发平台下，实现一体化的编译连接和原代码调试。使用 MPLAB-IDE 内的调试工具 ICE2000、ICD2 和软件模拟器都可以实现原代码级的程序调试，非常方便。

首先必须要在你的计算机中安装 PICC 编译器，无论是完全版还是学习版都可以和 MPLAB-IDE 挂接。安装成功后可以进入 IDE，选择菜单项 Project → Set Language Tool Locations...，打开语言工具挂接设置对话框，如图 11-1 所示：

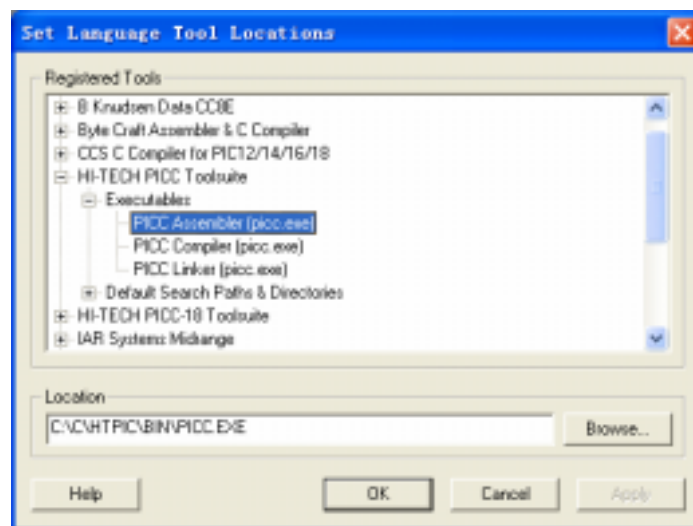


图 11-1 MPLAB-IDE 语言工具设置对话框

在对话框中选择“HI-TECH PICC Toolsuite”栏，展开可执行文件组“Executable”后，列出了将被 MPLAB-IDE 后台调用的编译器所用到的所有可执行文件，其中有汇编编译器“PICC Assembler”、C 原程序编译器“PICC Compiler”和连接定位程序“PICC Linker”。同时在此列表中还显示了对应的可执行程序名，请注意在这里都是“PICC.EXE”。用鼠标分别点击选中这三项可执行文件，观察对话框下面“Location”一栏中显示的文件路径，用“Browse...”按钮，从计算机中已经安装的 PICC 编译器文件夹中选择 PICC.EXE 文件。实际上 PICC.EXE 只是一个调度管理程序，它会按照所输入的文件扩展名自动调用对应的编译器和连接器，用户要注意的是 C 语言原程序扩展名用“.c”，汇编原程序用“.as”即可。

工具挂接完成后，在建立项目时可以选择语言工具为“HI-TECH PICC”，具体步骤可以参阅第三章 3.1.3 节，此处不再重复。项目建立完成后可以加入 C 或汇编原程序，也可以加入已有的库文件或已经编译的目标文件。最常见的是只加入 C 原程序。用 C 语言编程的好

处是可以实现模块化编程。程序编写者应尽量把相互独立的控制任务用多个独立的 C 原程序文件实现，如果程序量较大，一般不要把所有的代码写在一个文件内。图 11-2 列出的是笔者建立的一个项目中所有 C 原程序模块，其中主控、数值计算、I<sup>2</sup>C 总线操作、命令按钮处理和液晶显示驱动等不同的功能分别在不同的独立的原程序模块中实现。

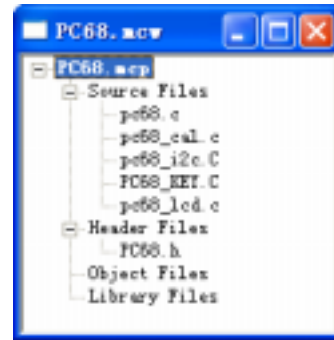


图 11-2 C 语言多模块编程

#### 11.4 PIC 单片机的 C 语言原程序基本框架

基于 PICC 编译环境编写 PIC 单片机程序的基本方式和标准 C 程序类似，程序一般由以下几个主要部分组成：

- 在程序的最前面用#include 预处理指令引用包含头文件，其中必须包含一个编译器提供的“pic.h”文件，实现单片机内特殊寄存器和其它特殊符号的声明；
- 用“\_\_CONFIG”预处理指令定义芯片的配置位；
- 声明本模块内被调用的所有函数的类型，PICC 将对所调用的函数进行严格的类型匹配检查；
- 定义全局变量或符号替换；
- 实现函数（子程序），特别注意 main 函数必须是一个没有返回的死循环。

下面的例 11-1 为一个 C 原程序的范例，供大家参考。

```
#include <pic.h>           //包含单片机内部资源预定义
#include "pc68.h"           //包含自定义头文件

//定义芯片工作时的配置位
__CONFIG (HS & PROTECT & PWRTEN & BOREN & WDTDIS);

//声明本模块中所调用的函数类型
void SetSFR(void);
void Clock(void);
void KeyScan(void);
void Measure(void);
void LCD_Test(void);
void LCD_Displ(unsigned char);

//定义变量
unsigned char second, minute, hour;
bit flag1, flag2;

//函数和子程序
```

```
void main(void)
{
    SetSFR();
    PORTC = 0x00;
    TMR1H += TMR1H_CONST;
    LED1 = LED_OFF;

    LCD_Test();

    //程序工作主循环
    while(1) {
        asm("cli rwdt");           //清看门狗
        Clock();                   //更新时钟
        KeyScan();                 //扫描键盘
        Measure();                 //数据测量
        SetSFR();                 //刷新特殊功能寄存器
    }
}
```

例 11-1 C 语言原程序框架举例

## 11.5 PICC 中的变量定义

### 11.5.1 PICC 中的基本变量类型

PICC 支持的基本变量类型见表 11-1：

类型	长度 (位数)	数学表达
bit	1	布尔型位变量，0 或 1 两种取值
char	8	有符号或无符号字符变量，PICC 缺省认定 char 型变量为无符号数，但可以通过编译选项改为有符号字节变量
unsigned char	8	无符号字符变量
short	16	有符号整型数
unsigned short	16	无符号整型数
int	16	有符号整型数
unsigned int	16	无符号整型数
long	32	有符号长整型数
unsigned long	32	无符号长整型数
float	24	浮点数
double	24 或 32	浮点数，PICC 缺省认定 double 型变量为 24 位长，但可以改变编译选项改成 32 位

表 11-1 PICC 的基本变量类型

PICC 遵循 Little-endian 标准，多字节变量的低字节放在存储空间的低地址，高字节放在高地址。

### 11.5.2 PICC 中的高级变量

基于表 11-1 的基本变量，除了 bit 型位变量外，PICC 完全支持数组、结构和联合等复合型高级变量，这和标准的 C 语言所支持的高级变量类型没有什么区别。例如：

```
数组: unsigned int data[10];  
结构: struct commInData {  
    unsigned char inBuff[8];  
    unsigned char getPtr, putPtr;  
};  
联合: union int_Byte {  
    unsigned char c[2];  
    unsigned int i;  
};
```

例 11-2 C 语言高级变量举例

### 11.5.3 PICC 对数据寄存器 bank 的管理

为了使编译器产生最高效的机器码，PICC 把单片机中数据寄存器的 bank 问题交由程序员自己管理，因此在定义用户变量时必须自己决定这些变量具体放在哪一个 bank 中。如果没有特别指明，所定义的变量将被定位在 bank0，例如下面所定义的这些变量：

```
unsigned char buffer[32];  
bit flag1, flag2;  
float val[8];
```

除了 bank0 内的变量声明时不需特殊处理外，定义在其它 bank 内的变量前面必须加上相应的 bank 序号，例如：

```
bank1 unsigned char buffer[32];    //变量定位在 bank1 中  
bank2 bit flag1, flag2;            //变量定位在 bank2 中  
bank3 float val[8];                //变量定位在 bank3 中
```

中档系列 PIC 单片机数据寄存器的一个 bank 大小为 128 字节，刨去前面若干字节的特殊功能寄存器区域，在 C 语言中某一 bank 内定义的变量字节总数不能超过可用 RAM 字节数。如果超过 bank 容量，在最后连接时会报错，大致信息如下：

```
Error[000] : Can't find 0x12C words for psect rbss_1 in segment BANK1
```

连接器告诉你总共有 0x12C (300) 个字节准备放到 bank1 中但 bank1 容量不够。显然，只有把一部分原本定位在 bank1 中的变量改放到其它 bank 中才能解决此问题。

虽然变量所在的 bank 定位必须由程序员自己决定，但在编写原程序时进行变量存取操作前无需再特意编写设定 bank 的指令。C 编译器会根据所操作的对象自动生成对应 bank 设定的汇编指令。为避免频繁的 bank 切换以提高代码效率，尽量把实现同一任务的变量定位在同一个 bank 内；对不同 bank 内的变量进行读写操作时也尽量把位于相同 bank 内的变量归并在一起进行连续操作。



#### 11.5.4 PICC 中的局部变量

PICC 把所有函数内部定义的 auto 型局部变量放在 bank0。为节约宝贵的存储空间，它采用了一种被叫做“静态覆盖”的技术来实现局部变量的地址分配。其大致的原理是在编译器编译原代码时扫描整个程序中函数调用的嵌套关系和层次，算出每个函数中的局部变量字节数，然后为每个局部变量分配一个固定的地址，且按调用嵌套的层次关系各变量的地址可以相互重叠。利用这一技术后所有的动态局部变量都可以按已知的固定地址地进行直接寻址，用 PIC 汇编指令实现的效率最高，但这时不能出现函数递归调用。PICC 在编译时会严格检查递归调用的问题并认为这是一个严重错误而立即终止编译过程。

既然所有的局部变量将占用 bank0 的存储空间，因此用户自己定位在 bank0 内的变量字节数将受到一定的限制，在实际使用时需注意。

#### 11.5.5 PICC 中的位变量

bit 型位变量只能是全局的或静态的。PICC 将把定位在同一 bank 内的 8 个位变量合并成一个字节存放于一个固定地址。因此所有针对位变量的操作将直接使用 PIC 单片机的位操作汇编指令高效实现。基于此，位变量不能是局部自动型变量，也无法将其组合成复合型高级变量。

PICC 对整个数据存储空间实行位编址，0x000 单元的第 0 位是位地址 0x0000，以后类推，每个字节有 8 个位地址。编制位地址的意义纯粹是为了编译器最后产生汇编级位操作指令而用，对编程人员来说基本可以不管。但若能了解位变量的位地址编址方式就可以在最后程序调试时方便地查找自己所定义的位变量，如果一个位变量 flag1 被编址为 0x123，那么实际的存储空间位于：

字节地址 =  $0x123/8 = 0x24$

位偏移 =  $0x123\%8 = 3$

即 flag1 位变量位于地址为 0x24 字节的第 3 位。在程序调试时如果要观察 flag1 的变化，必须观察地址为 0x24 的字节而不是 0x123。

PIC 单片机的位操作指令是非常高效的。因此，PICC 在编译原代码时只要有可能，对普通变量的操作也将以最简单的位操作指令来实现。假设一个字节变量 tmp 最后被定位在地址 0x20，那么

```
tmp |= 0x80      =>  bsf      0x20, 7
tmp &= 0xf7      =>  bcf      0x20, 3
if (tmp&0xfe)    =>  btfsc    0x20, 0
```

即所有只对变量中某一位操作的 C 语句代码将被直接编译成汇编的位操作指令。虽然编程时不用太关心，但如果能了解编译器是如何工作的，那将有助于引导我们写出高效简洁的 C 语言原程序。

在有些应用中需要将一组位变量放在同一个字节中以便需要时一次性地进行读写，这一功能可以通过定义一个位域结构和一个字节变量的联合来实现，例如：

```
union {  
    struct {  
        unsigned b0: 1;  
        unsigned b1: 1;  
        unsigned b2: 1;  
        unsigned b3: 1;  
        unsigned b4: 1;  
        unsigned b5: 1;  
        unsigned : 2;    //最高两位保留  
    } oneBit;  
    unsigned char allBits;  
} myFlag;
```

例 11-3 定义位变量于同一字节

需要存取其中某一位时可以

```
myFlag.oneBit.b3=1;    //b3 位置 1
```

一次性将全部位清零时可以

```
myFlag.allBits=0;    //全部位变量清 0
```

当程序中把非位变量进行强制类型转换成位变量时,要注意编译器只对普通变量的最低位做判别:如果最低位是 0,则转换成位变量 0;如果最低位是 1,则转换成位变量 1。而标准的 ANSI-C 做法是判整个变量值是否为 0。另外,函数可以返回一个位变量,实际上此返回的位变量将存放于单片机的进位位中带出返回。

#### 11.5.6 PICC 中的浮点数

PICC 中描述浮点数是以 IEEE-754 标准格式实现的。此标准下定义的浮点数为 32 位长,在单片机中要用 4 个字节存储。为了节约单片机的数据空间和程序空间,PICC 专门提供了一种长度为 24 位的截短型浮点数,它损失了浮点数的一点精度,但浮点运算的效率得以提高。在程序中定义的 float 型标准浮点数的长度固定为 24 位,双精度 double 型浮点数一般也是 24 位长,但可以在程序编译选项中选择 double 型浮点数为 32 位,以提高计算的精度。

一般控制系统中关心的是单片机的运行效率,因此在精度能够满足的前提下尽量选择 24 位的浮点数运算。

#### 11.5.7 PICC 中变量的绝对定位

首先必须强调,在用 C 语言写程序时变量一般由编译器和连接器最后定位,在写程序之时无需知道所定义的变量具体被放在哪个地址(除了 bank 必须声明)。

真正需要绝对定位的只是单片机中的那些特殊功能寄存器,而这些寄存器的地址定位在 PICC 编译环境所提供的头文件中已经实现,无需用户操心。程序员所要了解的也就是 PICC 是如何定义这些特殊功能寄存器和其中的相关控制位的名称。好在 PICC 的定义标准基本上按照芯片的数据手册中的名称描述进行,这样就秉承了变量命名的一贯性。一个变量绝对定位的例子如下:

```
unsigned char tmpData @ 0x20;    //tmpData 定位在地址 0x20
```

千万注意，PICC 对绝对定位的变量不保留地址空间。换句话说，上面变量 tmpData 的地址是 0x20，但最后 0x20 处完全有可能又被分配给了其它变量使用，这样就发生了地址冲突。因此针对变量的绝对定位要特别小心。从笔者的应用经验看，在一般的程序设计中用户自定义的变量实在是没有绝对定位的必要。

如果需要，位变量也可以绝对定位。但必须遵循上面介绍的位变量编址的方式。如果一个普通变量已经被绝对定位，那么此变量中的每个数据位就可以用下面的计算方式实现位变量指派：

```
unsigned char tmpData @ 0x20;    //tmpData 定位在地址 0x20  
bit tmpBit0 @ tmpData*8+0;      //tmpBit0 对应于 tmpData 第 0 位  
bit tmpBit1 @ tmpData*8+1;      //tmpBit1 对应于 tmpData 第 1 位  
bit tmpBit2 @ tmpData*8+2;      //tmpBit2 对应于 tmpData 第 2 位
```

如果 tmpData 事先没有被绝对定位，那就不能用上面的位变量定位方式。

#### 11.5.8 PICC 的其它变量修饰关键词

- extern — 外部变量声明

如果在一个 C 程序文件中要使用一些变量但其原型定义写在另外的文件中，那么在本文件中必须将这些变量声明成“extern”外部类型。例如程序文件 code1.c 中有如下定义：

```
bank1 unsigned char var1, var2;    //定义了 bank1 中的两个变量
```

在另外一个程序文件 code2.c 中要对上面定义的变量进行操作，则必须在程序的开头定义：

```
extern bank1 unsigned char var1, var2;    //声明位于 bank1 的外部变量
```

- volatile — 易变型变量声明

PICC 中还有一个变量修饰词在普通的 C 语言介绍中一般是看不到的，这就是关键词“volatile”。顾名思义，它说明了一个变量的值是会随机变化的，即使程序没有刻意对它进行任何赋值操作。在单片机中，作为输入的 IO 端口其内容将是随意变化的；在中断内被修改的变量相对主程序流程来讲也是随意变化的，很多特殊功能寄存器的值也将随着指令的运行而动态改变。所有这种类型的变量必须将它们明确定义成“volatile”类型，例如：

```
volatile unsigned char STATUS @ 0x03;  
volatile bit commFlag;
```

“volatile”类型定义在单片机的 C 语言编程中是如此的重要，是因为它可以告诉编译器的优化处理器这些变量是实实在在存在的，在优化过程中不能无故消除。假定你的程序定义了一个变量并对其作了一次赋值，但随后就再也没有对其进行任何读写操作，如果是非 volatile 型变量，优化后的结果是这个变量将有可能被彻底删除以节约存储空间。另外一种情形是在使用某一个变量进行连续的运算操作时，这个变量的值将在第一次操作时被复制到中间临时变量中，如果它是非 volatile 型变量，则紧接其后的其它操作将有可能直接从临时变量中取数以提高运行效率，显然这样做后对于那些随机变化的参数就会出问题。只要将其



定义成 volatile 类型后，编译后的代码就可以保证每次操作时直接从变量地址处取数。

- const — 常数型变量声明

如果变量定义前冠以“const”类型修饰，那么所有这些变量就成为常数，程序运行过程中不能对其修改。除了位变量，其它所有基本类型的变量或高级组合变量都将被存放在程序空间（ROM 区）以节约数据存储空间。显然，被定义在 ROM 区的变量是不能再在程序中进行赋值修改的，这也是“const”的本来意义。实际上这些数据最终都将以“retlw”的指令形式存放在程序空间，但 PICC 会自动编译生成相关的附加代码从程序空间读取这些常数，程序员无需太多操心。例如：

```
const unsigned char name[]="This is a demo"; //定义一个常量字符串
```

如果定义了“const”类型的位变量，那么这些位变量还是被放置在 RAM 中，但程序不能对其赋值修改。本来，不能修改的位变量没有什么太多的实际意义，相信大家在实际编程时不会大量用到。

- persistent — 非初始化变量声明

按照标准 C 语言的做法，程序在开始运行前首先要把所有定义的但没有预置初值的变量全部清零。PICC 会在最后生成的机器码中加入一小段初始化代码来实现这一变量清零操作，且这一操作将在 main 函数被调用之前执行。问题是作为一个单片机的控制系统有很多变量是不允许在程序复位后被清零的。为了达到这一目的，PICC 提供了“persistent”修饰词以声明此类变量无需在复位时自动清零，程序员应该自己决定程序中的那些变量是必须声明成“persistent”类型，而且须自己判断什么时候需要对其进行初始化赋值。例如：

```
persistent unsigned char hour, minute, second; //定义时分秒变量
```

经常用到的是如果程序经上电复位后开始运行，那么需要将 persistent 型的变量初始化，如果是其它形式的复位，例如看门狗引发的复位，则无需对 persistent 型变量作任何修改。PIC 单片机内提供了各种复位的判别标志，用户程序可依具体设计灵活处理不同的复位情形。

### 11.5.9 PICC 中的指针

PICC 中指针的基本概念和标准 C 语法没有太多的差别。但是在 PIC 单片机这一特定的架构上，指针的定义方式还是有几点需要特别注意。

- 指向 RAM 的指针

如果是汇编语言编程，实现指针寻址的方法肯定就是用 FSR 寄存器，PICC 也不例外。为了生成高效的代码，PICC 在编译 C 原程序时将指向 RAM 的指针操作最终用 FSR 来实现间接寻址。这样就势必产生一个问题：FSR 能够直接连续寻址的范围是 256 字节（bank0/1 或 bank2/3），要覆盖最大 512 字节的内部数据存储空间，又该如何让定义指针？PICC 还是将这一问题留给程序员自己解决：在定义指针时必须明确指定该指针所适用的寻址区域，例如：

```
unsigned char *ptr0;           // 定义覆盖 bank0/1 的指针
bank2 unsigned char *ptr1;    // 定义覆盖 bank2/3 的指针
bank3 unsigned char *ptr2;    // 定义覆盖 bank2/3 的指针
```

上面定义了三个指针变量，其中 指针没有任何 bank 限定，缺省就是指向 bank0 和 bank1；  
和 一个指明了 bank2，另一个指明了 bank3，但实际上两者是一样的，因为一个指针可以同时覆盖两个 bank 的存储区域。另外，上面三个指针变量自身都存放在 bank0 中。我们将在稍后介绍如何在其它 bank 中存放指针变量。

既然定义的指针有明确的 bank 适用区域，在对指针变量赋值时必须实现类型匹配，下面的指针赋值将产生一个致命错误：

```
unsigned char *ptr0;           //定义指向 bank0/1 的指针
bank2 unsigned char buff[8];   //定义 bank2 中的一个缓冲区
程序语句：
ptr0 = buff; //错误！试图将 bank2 内的变量地址赋给指向 bank0/1 的指针
```

若出现此类错误的指针操作，PICC 在最后连接时会告知类似于下面的信息：

```
Fixup overflow in expression (...)
```

同样的道理，若函数调用时用了指针作为传递参数，也必须注意 bank 作用域的匹配，而这点往往容易被忽视。假定有下面的函数实现发送一个字符串的功能：

```
void SendMessage(unsigned char *);
```

那么被发送的字符串必须位于 bank0 或 bank1 中。如果你还要发送位于 bank2 或 bank3 内的字符串，必须再另外单独写一个函数：

```
void SendMessage_2(bank2 unsigned char *);
```

这两个函数从内部代码的实现来看可以一模一样，但传递的参数类型不同。

按笔者的应用经验体会，如果你看到了“Fixup overflow”的错误指示，几乎可以肯定是指针类型不匹配的赋值所致。请重点检查程序中有关指针的操作。

### ● 指向 ROM 常数的指针

如果一组变量是已经被定义在 ROM 区的常数，那么指向它的指针可以这样定义：

```
const unsigned char company[]="Microchip";           //定义 ROM 中的常数
const unsigned char *romPtr;                          //定义指向 ROM 的指针
```

程序中可以对上面的指针变量赋值和实现取数操作：

```
romPtr = company; //指针赋初值
data = *romPtr++; //取指针指向的一个数，然后指针加 1
```

反过来，下面的操作将是一个错误，因为该指针指向的是常数型变量，不能赋值。

```
*romPtr = data;    //往指针指向的地址写一个数
```

### ● 指向函数的指针

单片机编程时函数指针的应用相对较少，但作为标准 C 语法的一部分，PICC 同样支持函数指针调用。如果你对编译原理有一定的了解，就应该明白在 PIC 单片机这一特定的架构上实现函数指针调用的效率是不高的：PICC 将在 RAM 中建立一个调用返回表，真正的

调用和返回过程是靠直接修改 PC 指针来实现的。因此，除非特殊算法的需要，建议大家尽量不要使用函数指针。

- 指针的类型修饰

前面介绍的指针定义都是最基本的形式。和普通变量一样，指针定义也可以在前面加上特殊类型的修饰关键词，例如“persistent”、“volatile”等。考虑指针本身还要限定其作用域，因此 PICC 中的指针定义初看起来显得有点复杂，但只要了解各部分的具体含义，理解一个指针的实际用图就变得很直接。

(一) bank 修饰词的位置含义

前面介绍的一些指针有的作用于 bank0/1，有的作用于 bank2/3，但它们本身的存放位置全部在 bank0。显然，在一个程序设计中指针变量将有可能被定位在任何可用的地址空间，这时，bank 修饰词出现的位置就是一个关键，看下面的例子：

```
//定义指向 bank0/1 的指针，指针变量为 bank0 中
unsigned char *ptr0;

//定义指向 bank2/3 的指针，指针变量为 bank0 中
bank2 unsigned char *ptr0;

//定义指向 bank2/3 的指针，指针变量为 bank1 中
bank2 unsigned char * bank1 ptr0;
```

从中可以看出规律：前面的 bank 修饰词指明了此指针的作用域；后面的 bank 修饰词定义此指针变量自身的存放位置。只要掌握了这一法则，你就可以定义任何作用域的指针且可以将指针变量放于任何 bank 中。

(二) volatile、persistent 和 const 修饰词的位置含义

如果能理解上面介绍的 bank 修饰词的位置含义，实际上 volatile、persistent 和 const 这些关键词出现在前后不同位置上的含义规律是和 bank 一词相一致的。例如：

```
//定义指向 bank0/1 易变型字符变量的指针，指针变量位于 bank0 中且自身为非易变型
volatile unsigned char *ptr0;

//定义指向 bank2/3 非易变型字符变量的指针，指针变量位于 bank1 中且自身为易变型
bank2 unsigned char * volatile bank1 ptr0;

//定义指向 ROM 区的指针，指针变量本身也是存放于 ROM 区的常数
const unsigned char * const ptr0;
```

亦即出现在前面的修饰词其作用对象是指针所指处的变量；出现在后面的修饰词其作用对象就是指针变量自己。

## 11.6 PICC 中的子程序和函数

中档系列的 PIC 单片机程序空间有分页的概念，但用 C 语言编程时基本不用太多关心代码的分页问题。因为所有函数或子程序调用时的页面设定（如果代码超过一个页面）都由编译器自动生成的指令实现。

### 11.6.1 函数的代码长度限制

PICC 决定了 C 原程序中的一个函数经编译后生成的机器码一定会放在同一个程序页面内。中档系列的 PIC 单片机其一个程序页面的长度是 2K 字,换句话说,用 C 语言编写的任何一个函数最后生成的代码不能超过 2K 字。一个良好的程序设计应该有一个清晰的组织结构,把不同的功能用不同的函数实现是最好的方法,因此一个函数 2K 字长的限制一般不会对程序代码的编写产生太多影响。如果为实现特定的功能确实要连续编写很长的程序,这时就必须把这些连续的代码拆分成若干函数,以保证每个函数最后编译出的代码不超过一个页面空间。

### 11.6.2 调用层次的控制

中档系列 PIC 单片机的硬件堆栈深度为 8 级,考虑中断响应需占用一级堆栈,所有函数调用嵌套的最大深度不要超过 7 级。程序员必须自己控制子程序调用时的嵌套深度以符合这一限制要求。

PICC 在最后编译连接成功后可以生成一个连接定位映射文件 (\*.map),在此文件中有详细的函数调用嵌套指示图“call graph”,建议大家要留意一下。其信息大致如下(取自于一示范程序的编译结果):

```
Call graph:
*_main size 0,0 offset 0
  _RightShift_C
*  _Task size 0,1 offset 0
  lwtoft
  ftmul size 0,0 offset 0
    ftunpack1
    ftunpack2
  ftadd size 0,0 offset 0
    ftunpack1
    ftunpack2
  ftdenorm
```

例 11-4 C 函数调用层次图

上面所举的信息表明整个程序在正常调用子程序时嵌套最多为两级(没有考虑中断)。因为 main 函数不可能返回,故其不用计算在嵌套级数中。其中有些函数调用是编译代码时自动加入的库函数,这些函数调用从 C 原程序中无法直接看出,但在此嵌套指示图上则一目了然。

### 11.6.3 函数类型声明

PICC 在编译时将严格进行函数调用时的类型检查。一个良好的习惯是在编写程序代码前先声明所有用到的函数类型。例如:

```
void Task(void);
unsigned char Temperature(void);
void BIN2BCD(unsigned char);
void TimeDisplay(unsigned char, unsigned char);
```

这些类型声明确定了函数的入口参数和返回值类型,这样编译器在编译代码时就能保证生成

正确的机器码。笔者在实际工作中有时碰到一些用户声称发现 C 编译器生成了错误的代码，最后究其原因就是因为没有事先声明函数类型所致。

建议大家在编写一个函数的原代码时，立即将此函数的类型声明复制到原文件的起始处，见例 11-1；或是复制到专门的包含头文件中，再在每个原程序模块中引用。

#### 11.6.4 中断函数的实现

PICC 可以实现 C 语言的中断服务程序。中断服务程序有一个特殊的定义方法：

```
void interrupt ISR(void);
```

其中的函数名“ISR”可以改成任意合法的字母或数字组合，但其入口参数和返回参数类型必须是“void”型，亦即没有入口参数和返回参数，且中间必须有一个关键词“interrupt”。

中断函数可以被放置在原程序的任意位置。因为已有关键词“interrupt”声明，PICC 在最后进行代码连接时会自动将其定位到 0x0004 中断入口处，实现中断服务响应。编译器也会实现中断函数的返回指令“retfie”。一个简单的中断服务示范函数如下：

```
void interrupt ISR(void) //中断服务程序
{
    if (TO1E && TO1F) //判 TMRO 中断
    {
        TO1F = 0; //清除 TMRO 中断标志
        //在此加入 TMRO 中断服务
    }
    if (TMR11E && TMR11F) //判 TMR1 中断
    {
        TMR11F = 0; //清除 TMR1 中断标志
        //在此加入 TMR1 中断服务
    }
} //中断结束并返回
```

例 11-5 C 语言中断函数举例

PICC 会自动加入代码实现中断现场的保护，并在中断结束时自动恢复现场，所以程序员无需象编写汇编程序那样加入中断现场保护和恢复的额外指令语句。但如果在中断服务程序中需要修改某些全局变量时，是否需要保护这些变量的初值将由程序员自己决定和实施。

用 C 语言编写中断服务程序必须遵循高效的原则：

- 代码尽量简短，中断服务强调的是一个“快”字。
- 避免在中断内使用函数调用。虽然 PICC 允许在中断里调用其它函数，但为了解决递归调用的问题，此函数必须为中断服务独家专用。既如此，不妨把原本要写在其它函数内的代码直接写在中断服务程序中。
- 避免在中断内进行数学运算。数学运算将很有可能用到库函数和许多中间变量，就算不出现递归调用的问题，光在中断入口和出口处为了保护 and 恢复这些中间临时变量就需要大量的开销，严重影响中断服务的效率。



中档系列 PIC 单片机的中断入口只有一个，因此整个程序中只能有一个中断服务函数。

#### 11.6.5 标准库函数

PICC 提供了较完整的 C 标准库函数支持，其中包括数学运算函数和字符串操作函数。在程序中使用这些现成的库函数时需要注意的是入口参数必须在 bank0 中。

如果需要用到数学函数，则应在程序前 “#include <math.h>” 包含头文件；如果要使用字符串操作函数，就需要包含 “#include <string.h>” 头文件。在这些头文件中提供了函数类型的声明。通过直接查看这些头文件就可以知道 PICC 提供了哪些标准库函数。

C 语言中常用的格式化打印函数 “printf/sprintf” 用在单片机的程序中时要特别谨慎。printf/sprintf 是一个非常大的函数，一旦使用，你的程序代码长度就会增加很多。除非是在编写试验性质的代码，可以考虑使用格式化打印函数以简化测试程序；一般的最终产品设计都是自己编写最精简的代码实现特定格式的数据显示和输出。本来，在单片机应用中输出的数据格式都相对简单而且固定，实现起来应该很容易。

对于标准 C 语言的控制台输入 (scanf) / 输出 (printf) 函数，PICC 需要用户自己编写其底层函数 getch() 和 putch()。在单片机系统中实现 scanf/printf 本来就没什么太多意义，如果一定要实现，只要编写好特定的 getch() 和 putch() 函数，你可以通过任何接口输入或输出格式化的数据。

### 11.7 PICC 定义特殊区域值

PICC 提供了相关的预处理指令以实现在原程序中定义单片机的配置字和标记单元。

#### 11.7.1 定义工作配置字

在原程序中定义 PIC 单片机工作配置字的重要性在前面章节中已经阐述。在用 PICC 写程序时同样可以在 C 原程序中定义，具体方式如下：

```
__CONFIG (HS & UNPROTECT & PWRTE & BODIS & WDTEN);
```

上面的关键词 “\_\_CONFIG” (注意前面有两个下划线符) 专门用于芯片配置字的设定，后面括号中的各项配置位符号在特定型号单片机的头文件中已经定义 (注意不是 pic.h 头文件)，相互之间用逻辑 “与” 操作符组合在一起。这样定义的配置字信息最后将和程序代码一起放入同一个 HEX 文件。

在这里列出了适用于 16F7x 系列单片机配置位符号预定义，其它型号或系列的单片机配置字定义方式类似，使用前查阅一下对应的头文件即可。

```
/*振荡器配置*/  
#define RC      0x3FFF    // RC 振荡  
#define HS      0x3FFE    // HS 模式
```

```
#define XT          0x3FFD    // XT 模式
#define LP          0x3FFC    // LP 模式

/*看门狗配置*/
#define WDTEN       0x3FFF    // 看门狗打开
#define WDTDIS      0x3FFB    // 看门狗关闭

/*上电延时定时器配置*/
#define PWRTEN       0x3FF7    // 上电延时定时器打开
#define PWRTDIS     0x3FFF    // 上电延时定时器关闭

/*低电压复位配置*/
#define BOREN        0x3FFF    // 低电压复位允许
#define BORDIS      0x3FBF    // 低电压复位禁止

/*代码保护配置*/
#define UNPROTECT    0x3FFF    // 没有代码保护
#define PROTECT      0x3FEF    // 程序代码保护
```

例 11-6 头文件预定义的配置信息符号



## 11.7.2 定义芯片标记单元



PIC 单片机中的标记单元定义可以用下面的\_\_IDLOC ( 注意前面有两个下划线符 ) 预处理指令实现，方法如下：

```
__IDLOC (1234);
```

其特殊之处是括号内的值全部为 16 进制数，不需要用“0x”引导。这样上面的定义就设定了标记单元内容为 01020304。

## 11.8 MPLAB-IDE 中实现 PICC 的编译选项设置

在 11.3 节中已经介绍了如何实现 PICC 和 MPLAB-IDE 开发平台的挂接。一旦项目建立成功、程序编写完成后即可以通过 MPLAB 环境下的项目管理工具实现程序的编译、连接和调试。对应于整个项目编译最常用的 MPLAB 快捷图标为“”和“”。它们的含义分别是：

-  - 项目维护 ( Make )：MPLAB 检查项目中的原程序文件，只编译那些在上次编译后又被修改过的原程序，最后进行连接；
-  - 项目重建 ( Build All )：项目中的所有原程序文件，不管是否有修改，都将被重新编译一次，最后进行连接。

也可以通过 Project 菜单选择“Make”或“Build All”实现项目编译。不管采用何种方式，在启动编译过程前一般都要设定一些编译选项。

### 11.8.1 选择单片机型号

在选择 PICC 作为语言工具并建立了项目后，同样通过菜单项 Configure→Select Device 在 MPLAB 环境中选择具体单片机型号。请回顾一下例 11-1 的代码，我们在原程序一开始使用了“#include <pic.h>”实现了相关单片机的一些预定义符号的直接引用，但没有具体指明是哪一个型号。实际上，“pic.h”头文件只是一个简单的管理工具（条件判别），它会按照 MPLAB 所选择的特定型号的单片机，把真正对应的头文件包含进来。有兴趣者可以直接用文本编辑工具打开 pic.h 文件查看其是如何根据不同的单片机型号包含对应的头文件。

这样对程序员而言，程序中只需加上一句“#include <pic.h>”即可。

### 11.8.2 PICC 普通编译选项（General）设定

参考第三章 3.2.7 节的内容和图 3-20 的指示说明，启动编译选项设定对话框。在使用 PICC 语言工具时对话框的内容和用 MPASM 汇编工具相比完全不同。图 11-3 为 PICC 编译环境下普通选项设定的界面。

在此界面中用户唯一能改变的是编译器查找头文件时的指定路径（Include Path），实际上如果编译器安装没有问题，在此界面中这些普通选项的设定无需任何改动，编译器会自动到缺省认定的路径中（编译器安装后的相关路径）查找编译所需的各类文件。



图 11-3 PICC 普通选项设定



图 11-4 PICC 全局选项设定

### 11.8.3 PICC 全局选项设定（PICC Global）

全局选项将影响项目中所有 C 和汇编原程序的编译，详细的设定内容见图 11-4。其中必须关注的有：

- Compile for MPLAB ICD：如果你准备用 ICD 调试 C 语言编译后的代码，那么此项就必须打钩选中。这样编译后的结果就能保证 ICD 本身使用的芯片资源（一小部分的程序和数据空间）不被应用程序所占用。
- Treat ‘char’ as signed：为了提高编译后的代码效率，PICC 缺省认定‘char’型变量也

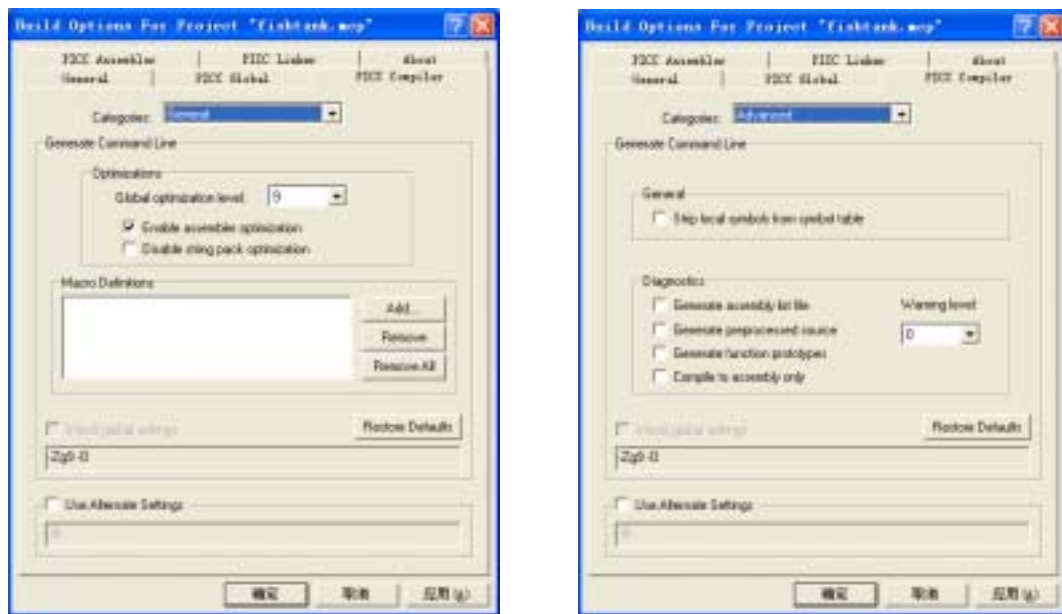
是无符号数。如果在设计中需要使用带符号的‘char’型变量，此项就应该被选中。

- Floating point ‘double’ width：同样为了提高编译后的代码效率，PICC 缺省认定‘double’型的双精度浮点数变量的实现长度为 24 位（等同于普通 float 型浮点数）。在这里可以选择使其长度达 32 位。这样数值计算的精度将得到提高，但代码长度将增加，计算速度也会降低，所以请在权衡利弊后作出你自己的决定。

#### 11.8.4 C 编译器选项设定 (PICC Compiler)

项目中所有的 C 原程序都将通过 C 编译器编译成机器码，这些选项决定了 C 编译器是如何工作的。所有选项又分为两组：普通选项 (General) 和高级选项 (Advanced)，分别见图 11-5A 和 11-5B。

C 编译器的普通选项最重要的就是针对代码优化的设定。如果没有特殊原因，应该设定全局优化级别为 9 级（最高级别优化），同时使用汇编级优化，这样最终得到的代码效率最高（长度和执行速度两方面）。按笔者的使用经验，仅从代码长度去比较，使用最高级别优化后代码长度至少可以减少 20%（2K 字以上的程序）。而且 PICC 的优化器相当可靠，一般



(A) 常用选项

(B) 高级选项

图 11-5 C 编译器选项设定

不会因为使用优化从而使生成的程序出现错误。碰到的一些问题也基本都是用户编写的原程序有漏洞所导致，例如一些变量应该是 volatile 型但程序员没有明确定义，在优化前程序可以正常运行，一旦使用优化，程序运行就出现异常。显然，把出现的这些问题归罪到编译器是毫无道理的。

使用优化后可能对原程序级的调试带来一些不便之处。因 PICC 可能会重组编译后的代码，例如多处重复的代码可能会改成同一个子程序调用以节约程序空间，这样在调试过程中跟踪原程序时可能会出现程序乱跳的现象，这基本是正常的。若为了强调更直观的代码调试过程，你可以将优化级别降低甚至关闭所有优化功能，这样调试时程序的运行就可以按部就班了。



C 编译器的高级选项设定基本都是针对诊断信息输出的，和生成的代码无关。用得相对较多的选项有：

- Generate assembly list file：编译器生成 C 原程序的汇编列表文件（\*.lst）。在此文件中列出了每一行 C 原代码对应的汇编指令，但这些都是优化前的代码。简单的一条 C 语句被翻译成汇编指令后可能有好几条。有时汇编列表文件可以作为解决问题的辅助手段。如果你怀疑编译器生成的代码有错误，不妨先产生对应的汇编列表文件，看看在优化前一条 C 语句被编译后的汇编码到底是什么。
- Compile to assembly only：这一选项的作用是把 C 原程序编译成汇编指令文件（\*.as），此时将不生成目标文件，也不进行最后的连接定位。这一选项在 C 和汇编混合编程时特别有用。通过解读 C 程序对应的汇编指令，可以掌握 C 程序中存取变量的具体方法，然后用在自己编写的汇编指令中。我们将在稍后专门做介绍。

### 11.8.5 连接器选项设定（PICC Linker）

连接器 PICC Linker 的选项基本不用作太多的改变，在图 11-6 的对话框中显示了可设定的各类项目。其中有两项有用的信息输出可以考虑加以利用：

- Generate map file：生成连接定位映射文件。在此映射文件中详细列出了所有程序用到的变量的具体物理地址；所有函数的入口地址；函数相互之间调用的层次关系和深度等。这些信息对于程序的调试将非常有用。此文件将以扩展名“\*.map”的形式存放在同一个项目路径下，需要时可以用任何文本编辑器打开观察。
- Display memory-segment usage：显示详细的内存分配和使用情况报告。用户可以了解到程序空间和数据存储器空间资源分配的细节。下面列举了在一个项目编译后实际的内存使用信息，为方便理解笔者用“//”添加了一些注释：



图 11-6 PICC 连接器选项设定

Psect Usage Map: //程序段定位表

Psect	Contents	Memory Range
powerup	Power on reset code	\$0000 - \$0003
intentry	Interrupt service routine	\$0004 - \$000C
intcode	Interrupt service routine	\$000D - \$002C
intret	Interrupt service routine	\$002D - \$0035
init	Initialization code	\$0036 - \$003D
end_init	Initialization code	\$003E - \$0040



clrtext	Memory clearing code	\$0041 - \$0047
const3	Strings and constant data	\$0048 - \$0060
const	Strings and constant data	\$0061 - \$0071
const2	Strings and constant data	\$0072 - \$0076
text	Program and library code	\$0576 - \$0582
text	Program and library code	\$0583 - \$07C7
float_te	Arithmetic routine code	\$07C8 - \$07FF
rbss_0	Bank 0 RAM variables	\$0021 - \$0042
temp	Temporary RAM data	\$0043 - \$0047
nvrn	Persistent RAM data	\$0048 - \$004A
intsav	Registers saved on interrupt	\$004B - \$004D
intsav	Registers saved on interrupt	\$007F - \$007F
intsav_1	Saved copy of W in bank 1	\$00FF - \$00FF
rbit_0	Bank 0 bit variables	\$0100 - \$0104
config	User-programmed CONFIG bits	\$2007 - \$2007

```

Memory Usage Map:           //存储空间使用情况报告
//程序空间代码定位地址分布
Program ROM   $0000 - $0076 $0077 ( 119) words
Program ROM   $0576 - $07FF $028A ( 650) words
                $0301 ( 769) words total Program ROM

//bank0 数据空间变量地址分布
Bank 0 RAM    $0021 - $004D $002D ( 45) bytes
Bank 0 RAM    $007F - $007F $0001 ( 1) bytes
                $002E ( 46) bytes total Bank 0 RAM

//bank1 数据空间变量地址分布
Bank 1 RAM    $00FF - $00FF $0001 ( 1) bytes total Bank 1 RAM
//bank0 数据空间位变量地址分布
Bank 0 Bits   $0100 - $0104 $0005 ( 5) bits total Bank 0 Bits
//配置字地址
Config Data   $2007 - $2007 $0001 ( 1) words total Config Data

Program statistics:         //程序总体资源消耗统计

Total ROM used   769 words (18.8%)      //生成代码字总数和程序空间使用率
Total RAM used   48 bytes (25.0%)       //使用数据字节数和数据空间使用率

```

例 11-7 编译后程序使用的内存信息

### 11.8.6 汇编器选项设定 (PICC Assembler)

PICC 环境提供了自己的汇编编译器，它和 Microchip 公司提供的 MPASM 编译器在原程序的语法表达方面要求稍有不同。另外，PICC 的汇编编译器要求输入原程序文件的扩展名是 “\*.as”，而 MPASM 缺省认定的原程序以 “\*.asm” 为扩展名。

在基于 PICC 编译环境下开发 PIC 单片机的 C 语言应用程序时基本无需关心其汇编编译器，除非是在混合语言编程时用汇编语言编写完整的汇编原程序模块文件。其编译选项设定的对话框见图 11-7，最重要的是优化使能控制项“Enable optimization”，一般情况下应该使用汇编器的优化以节约程序空间。



图 11-7 PICC 汇编器选项设定

## 11.9 C 和汇编混合编程

有两个原因决定了用 C 语言进行单片机应用程序开发时使用汇编语句的必要性：单片机的一些特殊指令操作在标准的 C 语言语法中没有直接对应的描述，例如 PIC 单片机的清看门狗指令“clrwdt”和休眠指令“sleep”；单片机系统强调的是控制的实时性，为了实现这一要求，有时必须用汇编指令实现部分代码以提高程序运行的效率。这样，一个项目中就会出现 C 和汇编混合编程的情形，我们在此讨论一些混合编程的基本方法和技巧。

### 11.9.1 嵌入行内汇编的方法

在 C 原程序中直接嵌入汇编指令是最直接最容易的方法。如果只需要嵌入少量几条的汇编指令，PICC 提供了一个类似于函数的语句：

```
asm("clrwdt");
```

双引号中可以编写任何一条 PIC 的标准汇编指令。例如：

```
for (;;) {  
    asm("clrwdt");    //清看门狗  
    Task();  
    ClockRun();  
    asm("sleep");     //休眠  
    asm("nop");       //空操作延时  
}
```

例 11-8 逐行嵌入汇编的方式

如果需要编写一段连续的汇编指令，PICC 支持另外一种语法描述：用“#asm”开始汇编指令段，用“#endasm”结束。例如下面的一段嵌入汇编指令实现了将 0x20~0x7F 间的 RAM 全部清零：

```
#asm
    movl w    0x20
    movwf    _FSR
    clrf     _INDF
    incf     _FSR, f
    btfss    _FSR, 7
    goto     $-3
#endasm
```

例 11-9 整段嵌入汇编的方式

### 11.9.2 汇编指令寻址 C 语言定义的全局变量

C 语言中定义的全局或静态变量寻址是最容易的，因为这些变量的地址已知且固定。按 C 语言的语法标准，所有 C 中定义的符号在编译后将自动在前面添加一下划线符“\_”，因此，若要在汇编指令中寻址 C 语言定义的各类变量，一定要在变量前加上一“\_”符号，我们在上面例 11-9 中已经体现了这一变量引用的法则，因为 FSR 和 INDF 等特殊寄存器是以 C 语言语法定义的，因此汇编中需要对其寻址时前面必须添加下划线。

对于 C 语言中用户自定义的全局变量，用行内汇编指令寻址时也同样必须加上“\_”，下面的例 11-10 说明了具体的引用方法：

```
volatile unsigned char tmp;           //定义位于 bank0 的字符型全局变量

void Test(void)                       //测试程序
{
    #asm                              //开始行内汇编
    clrf    _STATUS                   //选择 bank0
    movl w   0x10                      //设定初值
    movwf    _tmp                     //tmp=0x10
    #endasm                            //结束行内汇编
    if (tmp==0x10) {                  //开始 C 语言程序
        ;
    }
}
```

例 11-10 行内汇编寻址 C 全局变量（位于 bank0）

上面的例子说明了汇编指令中寻址 C 语言所定义变量的基本方法。PICC 在编译处理嵌入的行内汇编指令时将会原封不动地把这些指令复制成最后的机器码。所有对 C 编译器所作的优化设定对这些行内汇编指令而言将不起任何作用。程序员必须自己负责编写最高效的汇编代码，同时处理变量所在的 bank 设定。对于定义在其它 bank 中的变量，还必须在汇编指令中加以明确指示，见例 11-11 的代码范例。

```
volatile bank1 unsigned char tmpBank1;    //定义位于 bank1 的字符型全局变量
volatile bank2 unsigned char tmpBank2;    //定义位于 bank2 的字符型全局变量
volatile bank3 unsigned char tmpBank3;    //定义位于 bank3 的字符型全局变量

void Test(void)                            //测试程序
{
    #asm                                    //开始行内汇编
    bcf    _STATUS, 6                      //选择 bank1
    bsf    _STATUS, 5
    movl w  0x10                            //设定初值
    movwf  _tmpBank1^0x80                  //tmpBank1=0x10

    bsf    _STATUS, 6                      //选择 bank2
    bcf    _STATUS, 5
    movl w  0x20                            //设定初值
    movwf  _tmpBank1^0x100                 //tmpBank2=0x20

    bsf    _STATUS, 6                      //选择 bank3
    bsf    _STATUS, 5
    movl w  0x30                            //设定初值
    movwf  _tmpBank1^0x180                 //tmpBank1=0x30
    #endasm                                //结束行内汇编
}
```

例 11-11 行内汇编寻址 C 全局变量（非 bank0 变量）

通过上面的代码实例，我们可以掌握这样一个规律：在行内汇编指令中寻址 C 语言定义的全局变量时，除了在寻址前设定正确的 bank 外，在指令描述时还必须在变量上异或其在所在 bank 的起始地址，实际上位于 bank0 的变量在汇编指令中寻址时也可以这样理解，只是异或的是 0x00，可以省略。如果你了解 PIC 单片机的汇编指令编码格式，上面异或的 bank 起始地址是无法在真正的汇编指令中体现的，其目的纯粹是为了告诉 PICC 连接器变量所在的 bank，以便连接器进行 bank 类别检查。

### 11.9.3 汇编指令寻址 C 函数的局部变量

前面已经提到，PICC 对自动型局部变量（包括函数调用时的入口参数）采用一种“静态覆盖”技术对每一个变量确定一个固定地址（位于 bank0），因此嵌入的汇编指令对其寻址时只需采用数据寄存器的直接寻址方式即可，唯一要考虑的是如何才能在编写程序时知道这些局部变量的寻址符号（具体地址在最后连接后才能决定，编程时无需关心）。一个最实用也是最可靠的方法是先编写一小段 C 代码，其中有最简单的局部变量操作指令，然后参考图 11-5(B)对话框选择“Compile to assembly only”，把此 C 原代码编译成对应的 PICC 汇编指令；查看 C 编译器生成的汇编指令是如何寻址这些局部变量的，你自己编写的行内汇编指令就采用同样的寻址方式。例如，例 11-12 的一小段 C 原代码编译出的汇编指令

```
//C 原程序代码
void Test(unsigned char inVar1, inVar2)
```

```
{
    unsigned char tmp1, tmp2;
    inVar1++;
    inVar2--;
    tmp1 = 1;
    tmp2 = 2;
}

//编译器生成的汇编指令
_Test
;    _tmp1 assigned to ?a_Test+0      //tmp1的寻址符为 ?a_Test+0
_Test$tmp1    set  ?a_Test
;    _tmp2 assigned to ?a_Test+1      //tmp2的寻址符为 ?a_Test+1
_Test$tmp2    set  ?a_Test+1
;    _inVar1 assigned to ?a_Test+2    //inVar1的寻址符为 ?a_Test+2
_Test$inVar1  set  ?a_Test+2
    line 44
;_inVar1 stored from w                //第一个字符型行参由W寄存器传递
    bcf 3,5
    bcf 3,6
    movwf    ?a_Test+2
;ht16.c: 43: unsigned char tmp1, tmp2;
    incf ?a_Test+2
    line 45
;ht16.c: 45: inVar2--;
    decf ?_Test                      //行参inVar2的寻址符为 ?_Test
    line 46
;ht16.c: 46: tmp1 = 1;
    clrf ?a_Test
    incf ?a_Test
    line 47
;ht16.c: 47: tmp2 = 2;
    movlw    2
    movwf    ?a_Test+1
    line 48
;ht16.c: 48: }
    return
```

例 11-12 PICC 实现局部变量操作的寻址方式

基于上面得到的 PICC 编译后局部变量的寻址方式,我们在 C 语言程序中用嵌入汇编指令时必须采用同样的寻址符以实现对应变量的存取操作,见下面的例 11-13。

```
//C 原程序代码
void Test(unsigned char inVar1, inVar2)
{
```



```
unsigned char tmp1, tmp2;

#asm                                //开始嵌入汇编
incf    ?a_Test+0, f                //tmp1++;
decf     ?a_Test+1, f                //tmp2--;

movlw    0x10
addwf    ?a_Test+2, f                //inVar1 += 0x10;
rrf      ?_Test, w                    //inVar2 循环右移一位
rrf      ?_Test, f

#endasm                              //结束嵌入汇编
}
```

例 11-13 嵌入汇编指令实现局部变量寻址操作

如果局部变量为多字节形式组成，例如整型数、长整型等，必须按照 PICC 约定的存储格式进行存取。前面已经说明了 PICC 采用“Little endian”格式，低字节放在低地址，高字节放在高地址。下面的例 11-14 实现了一个整型数的循环移位，在 C 语言中没有直接针对循环移位的语法操作，用标准 C 指令实现的效率较低。

```
//16 位整型数循环右移若干位
unsigned int RR_Shift16(unsigned int var, unsigned char count)
{
    while(count--)                    //移位次数控制
    {
        #asm                          //开始嵌入汇编
        rrf ?_RR_Shift16+0, w          //最低位送入 C
        rrf ?_RR_Shift16+1, f          //var 高字节右移 1 位，C 移入最高位
        rrf ?_RR_Shift16+0, f          //var 低字节右移 1 位
        #endasm                        //结束嵌入汇编
    }
    return(var);                      //返回结果
}
```

例 11-14 嵌入汇编指令对多字节变量的操作

#### 11.9.4 混合编程的一些经验

C 和汇编语言混合编程可以使单片机应用程序的开发效率和程序本身的运行效率达到最佳的配合。笔者从实际应用中得到一些经验供读者一起分享。

##### (一) 慎用汇编指令

相比于汇编语言，用 C 语言编程的优势是毋庸置疑的：开发效率大大提高、人性化的语句指令加上模块化的程序易于日常管理和维护、程序在不同平台间的移植方便。所以既然用了 C 语言编程，就尽量避免使用嵌入汇编指令或整个地编写汇编指令模块文件。PICC 已具备高效的优化功能，如果在写 C 原程序时就十分注意程序的编译和运行效率问题，加上 PICC 的后道编译优化，最后得到的代码效率不会比全部用汇编编写的代码差多少，尤其是程序量较大时。另外，PICC 对数据存储空间的利用率肯定比用户人工定位变量时的利用率

要高，同时还提供完整的库函数支持。C 语言的语法功能强大，能够高效率地实现绝大部分控制和运算功能。因此，除了一些十分强调单片机运行时间的代码或 C 语言没有直接对应的操作可以考虑用汇编指令实现外，其它部分都应该用 C 语言编写。

以上的例 11-14 进一步说明，变量的循环右移操作用 C 语言实现非常不方便，PIC 单片机已有对应的移位操作汇编指令，因此用嵌入汇编的形式实现效率最高。同时对移位次数的控制，本质上说变量 count 的递减判零也可以直接用汇编指令实现，但这样做节约不了多少代码，用标准的 C 语言描述更直观，更易于维护。

一句话：用了 C 语言后，就不要再老想着用汇编。

## (二) 尽量使用嵌入汇编

这和上面的慎用汇编指令的说法并不矛盾。如果确实需要用汇编指令实现部分代码以提高运行效率，应尽量使用行内汇编，避免编写纯汇编文件 (\*.as 文件)。

虽然 PICC 支持 C 和汇编原程序模块存在于同一个项目中，但要编写纯汇编文件必须首先了解 PICC 特有的汇编语法结构。Hitech 公司提供了完整的文档介绍其汇编器的使用方法，有兴趣者可以从其网站上下载 PICC 的用户使用手册查看。

笔者认为，类似于纯汇编文件的代码也可以在 C 语言框架下实现，方法是基于 C 标准语法定义所有的变量和函数名，包括需要传递的形式参数、返回参数和局部变量，但函数内部的指令基本用嵌入汇编指令编写，只有最后的返回参数用 C 语句实现。这样做后函数的运行效率和纯汇编编写时几乎一模一样，但各参数的传递统一用 C 标准实现，这样管理和维护就比较方便。例如下面的例 11-15 实现一个字节变量的偶校验位计算。

```
bit EvenParity(unsigned char data)
{
    #asm
    swapf    ?a_EvenParity+0, w           //入口参数 data 的寻址符为 ?a_EvenParity+0
    xorwf    ?a_EvenParity+0, f
    rrf      ?a_EvenParity+0, w
    xorwf    ?a_EvenParity+0, f
    btfsc    ?a_EvenParity+0, 2
    incf     ?a_EvenParity+0, f
    #endasm
    //至此，data 的最低位即为偶校验位
    if (data&0x01) return(1);
    else return(0);
}
```

例 11-15 C 函数框架中使用嵌入汇编指令

## (三) 尽量使用全局变量进行参数传递

使用全局变量最大的好处是寻址直观，只需在 C 语言定义的变量名前增加一个下划线

符即可在汇编语句中寻址；使用全局变量进行参数传递的效率也比形参高。编写单片机的 C 程序时不能死硬强求教科书上的模块化编程而大量采用行参和局部变量的做法，在开发编程时应视实际情况灵活变通，一切以最高的代码效率为目标。