PIC 单片机的 C 语言编程指南

2005-8

目　录

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 语言写程序和在 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，是无法写出高质量实用的 C 语言程序。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。

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 PIC12F629、PIC12F675、PIC16F84、PIC16F877 和 PIC16F627 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，因此可从 PICC-Lite 入手掌握 PIC 单片机的 C 语言编程。

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。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”的技术以实现对 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。

3．MPLAB-IDE 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。在建立项目时可以选择语言工具为“HI-TECH PICC”，项目建立完成后可以加入 C 或汇编源程序，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 源程序。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 源程序文件实现，如果程序量较大，一般不要把所有的代码写在一个文件

内。

4．C 语言程序基本框架

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。

```
#include <pic.h>
#include <stdio.h>
#include "lcd.h"

/* this is the maximum value of an A2D conversion. */
#define MAXVOLTAGE 5
//定义芯片工作时的配置位
__CONFIG(WDTDIS & XT & UNPROTECT);

void init(void)
{
       lcd_init(FOURBIT_MODE);
       ADON=1;       /* enable A2D converter */
       ADIE=0;       /* not interrupt driven */
       ADCON1=0x0E;
       ADCON0=1;
       TRISB=0x00;   // Set PORTB in output mode
       T1CON=0x31;   // turn on timer 1
       TMR1IE=1;     // timer 1 is interrupt enabled
       PEIE=1;              // enable peripheral interrupts
       GIE=1;        // turn on interrupts
}

void interrupt isr(void)
{
       if(TMR1IE && TMR1IF)
      {

              PORTB++;
              TMR1IF=0;
       }
```

```
}

void main(void)
{
       unsigned char last_value;
       unsigned char volts;
       unsigned char decivolts;
       unsigned char outString[20];

       init();
       lcd_puts("Adust the");
       lcd_home2();  // select line 2
       lcd_puts("potentiometer");

       while(1)
      {
              ADGO=1;
              while(ADGO)continue;
              ADIF=0;
              if(ADRESH!=last_value)
              {
                     volts=0;

                     lcd_clear();
                     sprintf(outString,"A2D = %d.%d volts",volts,decivolts);
                     lcd_puts(outString);
              }
              last_value=ADRESH;
       }
}
```

5．PICC 编译选项设置

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分别是：

－项目维护（Make）：MPLAB 检查项目中的源程序文件，只编译那些在上次编译后又被修改过的源程序，最后进行连接；

－项目重建（Build All）：项目中的所有源程序文件，不管是否有修改，都将被重新编译一次，最后进行连接。可以通过 Project 菜单选择“ Make ”或“ Build All ”实现项目编译。

5．1 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。

5．2 普通编译选项（General）设定

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。

5．3 全局选项设定（PICC Global）

全局选项将影响项目中所有 C 和汇编源程序的编译，其中有：

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。

5．4 编译器选项设定（PICC Compiler）

项目中所有的 C 源程序都将通过 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 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）。而且 PICC 的优化器相当可靠，一般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源程序有漏洞所导致，例如一些变量应该是 volatile 型但编程员没有明确定义，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。

使用优化后可能对源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源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 源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 源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。

5．5 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，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。

5．6 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 编译器在源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源程序文件的扩展名是“ *.as ”，而 MPASM 缺省认定的源程序以“ *.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源程序模块文件。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。

6．PICC 中的变量定义

6．1 基本变量类型

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

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。

6．2 高级变量

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 tm { int tm_sec； int tm_min； int tm_hour； int tm_mday； int tm_mon； int tm_year； int tm_wday； int tm_yday； int tm_isdst； }；
联合	union int_Byte { unsigned char c[2];

	unsigned int i; };

6．3 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 问题交由编程员自己管理，因此在定义用户变量时你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

虽然变量所在的 bank 定位必须由编程员自己决定，但在编写源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。

6．4 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。

6．5 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，以此后推，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

C 语句代码	汇编代码
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。虽然编程时可以不用太关心，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介的 C 语言源程序。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 b6:1;
		unsigned b7:1;
	}oneBit;
	unsigned char allBit;
}MyData;
```

需要存取其中某一位时可以 MyData.oneBit.b3=1; //b3 位置 1
一次性将全部位清零时可以 MyData.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

6．6 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。

6．7 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。编程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。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; //tmpBit0 对应于 tmpData 第 1 位
bit tmpBit2 @ tmpData*8+2; //tmpBit0 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。

6．8 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 test1.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 test2.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对其进行赋值修改的，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，编程员无需太多操心。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，编程员应该自己决定程序中的那些变量是必须声明成“ persisten ”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。

6．9 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

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; //①定义覆盖 bank0/1 的指针

bank2 unsigned char *ptr1; //②定义覆盖 bank2/3 的指针

bank3 unsigned char *ptr2; //③定义覆盖 bank2/3 的指针

上面定义了三个指针变量，其中①指针没有任何 bank 限定，缺省就是指向 bank0 和 bank1；②和③一个指明了 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。

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就必须实现类型匹配，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

- 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。

A. 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，指针变量为于 bank0 中
unsigned char *ptr0;
//定义指向 bank2/3 的指针，指针变量为于 bank0 中
bank2 unsigned char *ptr0;
//定义指向 bank2/3 的指针，指针变量为于 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 修饰词定义了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。

B. 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。

7． 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

7．1 函数的代码长度限制

PICC 决定了 C 源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。

7．2 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。编程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。

7．3 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：

```
#ifndef _LCD_H_
#define _LCD_H_

static bit LCD_RS    @ ((unsigned)&PORTB*8+7);   // Register select
static bit LCD_RW    @ ((unsigned)&PORTB*8+6);   // Register select
static bit LCD_EN    @ ((unsigned)&PORTB*8+5);   // Enable

static bit LCD_D4    @ ((unsigned)&PORTB*8+1);   // Data.4
static bit LCD_D5    @ ((unsigned)&PORTB*8+2);   // Data.5
static bit LCD_D6    @ ((unsigned)&PORTB*8+3);   // Data.6
static bit LCD_D7    @ ((unsigned)&PORTB*8+4);   // Data.7

extern void lcd_write(unsigned char);
extern void lcd_clear(void);
extern void lcd_puts(const char * s);
extern void lcd_goto(unsigned char pos);
extern void lcd_init(void);
extern void lcd_putch(char);

#endif
```

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。

7．4 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源程序的任意位置。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：

```
static void interrupt
isr(void)                   // Here be interrupt function - the name is
                            // unimportant.
{
       if(!T0IF)            // Was this a timer overflow?
              bad_intr = 1; // NO! Shock horror!
       count++;             // Add 1 to count - insert idle comment
       T0IF = 0;            // Clear interrupt flag, ready for next
       PORTA ^= 1;          // toggle bit 0 of Port A, to show we're alive
}
```

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，所以编程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编程员自己决定和实施。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和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。

7．5 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，你就可以通过任何接口输入或输出格式化的数据。

8． PICC 定义特殊区域值

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

8．1 定义工作配置字

在源程序中定义 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 源程序中定义，具体方式如下：

__CONFIG (HS & UNPROTECT & PWRTEN & BORDIS & WDTEN);

上面的关键词“ __CONFIG ”（注意前面有两个下划线符）专门用于是芯片配置字的设定，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
```

8．2 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。

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.

9．1 嵌入行内汇编的方法

在 C 源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。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：

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

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，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“ _ ”,下面的例子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
            movlw 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
```

```
    ;
    }
}
```

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。编程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，

```
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
      movlw 0x10 //设定初值
      movwf _tmpBank1^0x80 //tmpBank1=0x10
      bsf _STATUS,6 //选择 bank2
      bcf _STATUS,5
      movlw 0x20 //设定初值
      movwf _tmpBank1^0x100 //tmpBank2=0x20
      bsf _STATUS,6 //选择 bank3
      bsf _STATUS,5
      movlw 0x30 //设定初值
      movwf _tmpBank1^0x180 //tmpBank1=0x30
    #endasm //结束行内汇编
}
```

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。

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。

在 C 语言程序中用嵌入汇编指令时实现对应变量的存取操作，见下例

```
//C 源程序代码
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
}
```

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
}
```

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。

- 尽量使用嵌入汇编

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）。

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：

```
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);
}
```

- 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。

10．PICC 库函数指南

10.1 ABS 函数

函　数	ABS
提　要	#include <stdlib.h>
描　述	abs( )函数返回变量 j 的绝对值

例　程	#include <stdio.h>
	#include <stdlib.h>
	void main (void)
	{
	int a = -5；
	printf("The absolute value of %d is %d\n"，a，abs(a))；
	}

返回值	j 的绝对值
参　阅	

10.2 ACOS 函数

函　数	ACOS
提　要	#include <math.h> double acos (double f)

描　述	acos( )函数是 cos( )　的反函数。函数参数在[-1，1]区间内，返回值是一个用弧度表示的角度，而且该返回值的余弦值等于函数参数

例　程	#include <math.h>
	#include <stdio.h>
	/*以度为单位，打印[-1，1]区间内的反余弦值*/
	void
	main (void)
	{
	　　float i，a；
	　　for(i = -1.0，i < 1.0；i += 0.1)
	　　{
	　　a = acos(i)*180.0/3.141592；
	　　printf("acos(%f) = %f degrees\n"，i，a)；
	　　}
	}

返回值	返回值是一个用弧度表示的角度，区间是[0，π]。如果函数参数超出区间[-1，1]，则返回值将为 0
参　阅	sin( )，cos( )，tan( )，asin( )，atan( )，atan2( )

10.3　ASCTIME 函数

函　数	ASCTIME
提　要	#include <time.h> char * asctime (struct tm * t)
描　述	asctime( )函数通过指针　t　从上 struct tm 结构体中获得时间，返回描述当前日期和时间的 26 个字符串，其格式如下： Sun Sep 16 01:03:52 1973\n\0 值得注意的是,在字符串的末尾有换行符。字符串中的每个字长是固定的。以下例程得到当前时间，通过 localtime( )函数将其转换成一个 struct　tm 指针，最后转换成 ASCII 码并打印出来。其中，time( )函数需要用户提供（详情请参阅 time( )函数）。

例　程	#include <stdio.h>
	#include <time.h>
	void
	main (void)
	{
	　time_t clock；
	　struct tm * tp；
	　time(&clock)；
	　tp = localtime(&clock)；
	　printf("%s"，asctime(tp))；
	}

返回值	指向字符串的指针。注意：由于编译器不提供 time( )例行程序，故在本例程中它需要由用户提供。详情请参照 time( )函数
参　阅	ctime( )，gmtime( )，localtime( )，time( )

10.4 ASIN 函数

函　数	ASIN
提　要	#include <math.h> double asin (double f)
描　述	asin( )函数是 sin( )的反函数。它的函数参数在[-1，1]区间内，返回一个用弧度表示的角度值，而且这个返回值的正弦等于函数参数。

例　程	#include <math.h>
	#include <stdio.h>
	void
	main (void)
	{
	float i，a；
	for(i = -1.0；i < 1.0 ；i += 0.1)
	{
	a = asin(i)*180.0/3.141592；
	printf("asin(%f) = %f degrees\n"，i，a)；
	}
	}

返回值	本函数返回一个用弧度表示的角度值，其区间为[-π/2，π/2]。如果函数参数的值超出区间[-1，1]，则函数返回值将为 0。
参　阅	Sin( )，cos( )，tan( )，acos( )，atan( )，atan2( )

10.5 ATAN2 函数

函　数	ATAN2
提　要	#include <math.h> double atan2 (double y，double x)
描　述	本函数返回 y/x 的反正切值，并由两个函数参数的符号来决定返回值的象限。

例　程	#include <stdio.h>
	#include <math.h>
	void
	main (void)
	{
	printf("%f\n"，atan2(1.5，1))；
	}

返回值	返回 y/x 的反正切值（用弧度表示），区间为[-π，π]。如果 y 和 x 均为 0，将出现定义域错误，并返回 0。
参　阅	Sin( )，cos( )，tan( )，acos( )，atan( )，atan2( )

10.6 ATAN 函数

函　数	ATAN
提　要	#include <math.h> double atan (double x)
描　述	函数返回参数的反正切值。也就是说，本函数将返回一个在区间[-π/2，π/2]的角度 e，而且有 tan(e)=x（x 为函数参数）。

例　程	#include <stdio.h>
	#include <math.h>
	void
	main (void)
	{
	printf("%f\n"，atan(1.5))；
	}

返回值	返回函数参数的反正切值。
参　阅	sin( )，cos( )，tan( )，acos( )，atan( )，atan2( )

10.7 ATOF 函数

函　数	ATOF
提　要	##include <stdlib.h> double atof (const char * s)
描　述	tof( )函数将扫描由函数参数传递过来的字符串，并跳过字符串开头的空格。然后将一个数的 ASCII 表达式转换成双精度数。这个数可以用十进制数、浮点数或者科学记数法表示。

例　程	#include <stdlib.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buf[80]；
	double i；
	gets(buf)；
	i = atof(buf)；
	printf("Read %s: converted to %f\n"，buf，i)；
	}

返回值	本函数返回一个双精度浮点数。如果字符串中没有发现任何数字，则返回 0.0。
参　阅	atoi( )，atol( )

10.8 ATOI 函数

函　数	ATOI
提　要	#include <stdlib.h> int atoi (const char * s)
描　述	atoi( )函数扫描传递过来的字符串，跳过开头的空格并读取其符号;然后将一个十进制数的 ASCII 表达式转换成整数。

例　程	#include <stdlib.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buf[80]；
	int i；
	gets(buf)；
	i = atoi(buf)；
	printf("Read %s: converted to %d\n"，buf，i)；
	}

返回值	返回一个有符号的整数。如果在字符串中没有发现任何数字，则返回 0。
参　阅	xtoi( )，atof( )，atol( )

10.9 ATOL 函数

函　数	ATOL
提　要	#include <stdlib.h> long atol (const char * s)
描　述	atol( )函数扫描传递过来的字符串，并跳过字符串开头的空格；然后将十进制数的 ASCII 表达式转换成长整型。

例　程	#include <stdlib.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buf[80]；
	long i；
	gets(buf)；
	i = atol(buf)；
	printf("Read %s: converted to %ld\n"，buf，i)；
	}

返回值	返回一个长整型数。如果字符串中没有发现任何数字，返回值为 0。
参　阅	atoi( )，atof( )

10.10 CEIL 函数

函　数	CEIL
提　要	#include <math.h> double ceil (double f)
描　述	本函数对函数参数 f 取整，取整后的返回值为大于或等于 f 的最小整数。

例　程	#include <stdio.h>
	#include <math.h>
	void
	main (void)
	{
	double j；
	scanf("%lf"，&j)；
	printf("The ceiling of %lf is %lf\n"，j，ceil(j))；
	}

返回值	
参　阅	

10.11 COSH 函数

函　数	COSH、SINH、TANH
提　要	#include <math.h> double cosh (double f) double sinh (double f) double tanh (double f)
描　述	这些函数都是 cos( )，sin( )和 tan( )的双曲函数

例　程	#include <stdio.h>
	#include <math.h>
	void
	main (void)
	{
	printf("%f\n"，cosh(1.5))；
	printf("%f\n"，sinh(1.5))；
	printf("%f\n"，tanh(1.5))；
	}

返回值	cosh( )函数返回双曲余弦值，sinh( )函数返回双曲正弦值，tanh( )函数返回双曲正切
参　阅	

10.12 COS 函数

函　数	COS
提　要	#include <math.h> double cos (double f)
描　述	本函数将计算函数参数的余弦值。其中，函数参数用弧度表示。余弦值通过多项式级数近似值展开式算得。

例　程	#include <math.h>
	#include <stdio.h>
	#define C 3.141592/180.0
	void
	main (void)
	{
	double i；
	for(i = 0；i <= 180.0；i += 10)
	printf("sin(%3.0f) = %f，cos = %f\n"，i，sin(i*C)，cos(i*C))；
	}

返回值	返回一个双精度数，区间为[-1，1]
参　阅	Sin( )，tan( )，asin( )，acos( )，atan( )，atan2( )

10.13 CTIME 函数

函　数	CTIME
提　要	#include <time.h> char * ctime (time_t * t)
描　述	ctime( )函数将函数参数所指的时间转换成字符串，其结构与 asctime( )函数所描述的相同，并且精确到秒。以下例程将打印出当前的时间和日期

例　程	#include <stdio.h>
	#include <time.h>
	void
	main (void)
	{
	time_t clock；
	time(&clock)；
	printf("%s"，ctime(&clock))；
	}

返回值	本函数返回一个指向该字符串的指针。注意：由于编译器不会提供 time( )程序，故它需要由用户给定。详情请参阅 time( )函数。
参　阅	gmtime( )，localtime( )，asctime( )，time( )

10.14 DIV 函数

函　数	DIV
提　要	#include <stdlib.h> div_t div (int numer，int demon)
描　述	div( )函数实现分子除以分母，得到商和余数

例　程	#include <stdlib.h>
	#include <stdio.h>
	void
	main (void)
	{
	　div_t x；
	　x = div(12345，66)；
	　printf("quotient = %d，remainder = %d\n"，x.quot，x.rem)；
	}

返回值	返回一个包括商和余数的结构体 div_t
参　阅	gmtime( )，localtime( )，asctime( )，time( )

10.15 DI 函数

函　数	DI、EI
提　要	#include <pic.h> void ei(void) void di(void)
描　述	ei( )和 di( )函数分别实现全局中断使能和中断屏蔽，其定义在 pic.h 头文件中。它们将被扩展为一条内嵌的汇编指令，分别对中断使能位进行置位和清零。以下例程将说明 ei( )函数和 di( )函数在访问一个长整型变量时的应用。由于中断服务程序将修改该变量，所以如果访问该变量不按照本例程的结构编程，一旦在访问变量值的连续字期间出现中断，则函数 getticks( ) 将返回错误的值

例　程	#include <pic.h>
	long count；
	void interrupt tick(void)
	{
	　count++；
	}
	
	long getticks(void)
	{
	　long val； /*在访问 count 变量前禁止中断，保证访问的连续性*/
	　di( )；
	　val = count；
	　ei( )；
	　return val；
	}

返回值	

参　阅	

10.16　EEPROM_READ 函数

函　数	EEPROM_READ、EEPROM_WRITE
提　要	#include <pic.h> unsigned char eeprom_read (unsigned char addr)； void eeprom_write (unsigned char addr，unsigned char value)；
描　述	这些函数允许访问片内 EEPROM（如果片内有 EEPROM）。EEPROM 不是可直接寻址的寄存器空间，当需要访问 EEPROM 时，就需要将一些特定的字节序列加载到 EEPROM 控制寄存器中。写 EEPROM 是一个缓慢的过程。故 eeprom_write( ) 函数在写入下一个数据前，会通过查询恰当的寄存器来确保前一个数据已经写入完毕。另外，读 EEPROM 可以在一个指令周期内完成，所以没有必要查询读操作是否完成

例　程	#include <pic.h>
	void
	main (void)
	{
	　unsigned char data；
	　unsigned char address；
	　address = 0x10；
	　data = eeprom_read(address)；
	}

返回值	注意：如果调用 eeprom_write( )函数后需即刻调用 eeprom_read( )函数，则必须查询 EEPROM 寄存器以确保写入完毕。全局中断使能位（GIE）在 eeprom_write( ) 程序中重新恢复（写 EEPROM 时需要关闭总中断）。而且，本函数不会清 EEIF 标志位。
参　阅	

10.17　EVAL_POLY 函数

函　数	EVAL_POLY
提　要	#include <math.h> double eval_poly (double x，const double * d，int n)
描　述	eval_poly( )函数将求解一个多项式的值。这个多项式的系数分别包含在 x 和数组 d 中，例如：y = x*x*d2 + x*d1 + d0 该多项式的阶数由参数 n 传递过来
例　程	#include <stdio.h> #include <math.h> void main (void) {

	double x，y； double d[3] = {1.1，3.5，2.7}； x = 2.2； y = eval_poly(x，d，2)； printf("The polynomial evaluated at %f is %f\n"，x，y)； }
返回值	本函数返回一个双精度数，该数是自变量 x 对应的多项式值
参　阅	

10.18　EXP 函数

函　数	EXP
提　要	include <math.h> double exp (double f)
描　述	exp( )函数返回参数的指数函数值，即 ef（f 为函数参数）

例　程	#include <math.h>
	#include <stdio.h>
	void
	main (void)
	{
	double f；
	for(f = 0.0；f <= 5；f += 1.0)
	printf("e to %1.0f = %f\n"，f，exp(f))；
	}

返回值	本函数返回一个双精度数，该数是自变量 x 对应的多项式值
参　阅	log( )，log10( )，pow( )

10.19　FABS 函数

函　数	FABS
提　要	#include <math.h> double fabs (double f)
描　述	本函数返回双精度函数参数的绝对值

例　程	#include <stdio.h>
	#include <math.h>
	void
	main (void)
	{
	printf("%f %f\n"，fabs(1.5)，fabs(-1.5))；
	}

返回值	
参　阅	abs( )

10.20 FLOOR 函数

函　数	FLOOR
提　要	#Include <math.h> double floor (double f)
描　述	本函数对函数参数取整，取整后的返回值不大于函数参数 f

例　程	#include <stdio.h>
	#include <math.h>
	void
	main (void)
	{
	printf("%f\n"，floor(1.5))；
	printf("%f\n"，floor(-1.5))；
	}

返回值	
参　阅	

10.21 FREXP 函数

函　数	FREXP
提　要	#include <math.h> double frexp (double f，int * p)
描　述	frexp( )函数将一个浮点数分解成规格化小数和 2 的整数次幂两部分，整数幂部分存于指针 p 所指的 int 单元中。本函数的返回值 x 或在区间（0.5，1.0）内，或为 0；而且有 f=x×2p。如果 f 为 0，则分解出来的两部分均为 0

例　程	#include <math.h>
	#include <stdio.h>
	void
	main (void)
	{
	double f；
	int i；
	f = frexp(23456.34，&i)；
	printf("23456.34 = %f * 2^%d\n"，f，i)；
	}

返回值	
参　阅	ldexp( )

10.22 GET_CAL_DATA 函数

函　数	GET_CAL_DATA
提　要	#include <pic.h> double get_cal_data (const unsigned char * code_ptr)
描　述	本函数从 PIC 14000 标定空间返回一个 32 位的浮点标定数据。只有利用这个函数才能访问 KREF、KBG、BHTHERM 和 KTC 单元（32 位浮点参数）。由于 FOSC 和 TWDT 均是一个字节长度，故可以直接访问它们。

例　程	#include <pic.h>
	void
	main (void)
	{
	double x；
	unsigned char y；
	x = get_cal_data(KREF)；/*获得参考斜率（slope reference ratio）*/
	y =TWDT；/*获得 WDT 溢出时间*/
	}

返回值	返回定标参数值。注意：本函数仅用于 PIC 14000
参　阅	

10.23 GMTIME 函数

函　数	GMTIME
提　要	#include <time.h> struct tm * gmtime (time_t * t)
描　述	本函数把指针 t 所指的时间分解,并且存于结构体中，精确度为秒。其中，t 所指的时间必须自1970年1月1日0时0分0秒起。本函数所用的结构体被定义在time.h文件中，可参照本节“数据类型”部分

例　程	#include <stdio.h>
	#include <time.h>
	void
	main (void)
	{
	time_t clock；
	struct tm * tp；
	time(&clock)；
	tp = gmtime(&clock)；
	printf("It’s %d in London\n"，tp->tm_year+1900)；
	}

返回值	返回 tm 类型的结构体。注意：由于编译器不会提供 time( )程序，故它需要由用户给定。详情请参阅 time( )函数。
参　阅	ctime( )，asctime( )，time( )，localtime( )

10.24 ISALNUM 函数

函　数	ISALNUM，ISALPHA，ISDIGIT，ISLOWER
提　要	#include <ctype.h> int isalnum (char c) int isalpha (char c) int isascii (char c) int iscntrl (char c) int isdigit (char c) int islower (char c) int isprint (char c) int isgraph (char c) int ispunct (char c) int isspace (char c) int isupper (char c) int isxdigit(char c)
描　述	以上函数都被定义在 ctype.h 文件中。它们将测试给定的字符，看该字符是否为已知的几组字符中的成员。 isalnum (c)　　c 在 0～9、a～z 或者 A～Z 范围内； isalpha (c)　　c 在 A～Z 或 a～z 范围内； isascii (c)　　c 为 7 位 ASCII 字符； iscntrl (c)　　c 为控制字符； isdigit (c)　　c 为十进制阿拉伯数字； islower (c)　　c 在 a～z 范围内； isprint (c)　　c 为打印字符； isgraph (c)　　c 为非空格可打印字符； ispunct (c)　　c 不是字母数字混合的； isspace (c)　　c 是空格键、TAB 键或换行符； isupper (c)　　c 在 A～Z 范围内； isxdigit (c)　　c 在 0～9、a～f 或 A～F 范围内。

例　程	#include <ctype.h>
	#include <stdio.h>
	void
	main (void)
	{
	　char buf[80]；
	　int i；
	　gets(buf)；
	　i = 0；
	　while(isalnum(buf[i]))
	　i++；
	　buf[i] = 0；
	　printf(" ’ %s ’　is the word\n"，buf)；
	}

返回值	
参　阅	toupper( )，tolower( )，toascii( )

10.25 KBHIT 函数

函　数	KBHIT
提　要	#include <conio.h> bit kbhit (void)
描　述	如果键盘上的字符被按下，函数返回 1；否则返回 0。通常，该字符可通过 getch( ) 函数读取。

例　程	#include <conio.h>
	void
	main (void)
	{
	int i；
	while(!kbhit())
	{
	cputs("I＇m waiting..")；
	for(i = 0 ； i != 1000 ； i++)
	continue；
	}
	}

返回值	如果有键被按下，函数将返回 1；否则返回 0。此外，返回值为 1 位。注意：程序的主体需由用户实现，其主要框架可以从 sources 目录下直接获得。
参　阅	etch( )，getche( )

10.26 LDEXP 函数

函　数	LDEXP
提　要	#include <math.h> double ldexp (double f，int i)
描　述	ldexp( )函数是 frexp( )的反函数。它先进行浮点数 f 的指数部分与整数 i 的求和运算，然后返回合成结果

例　程	#include <math.h>
	#include <stdio.h>
	void
	main (void)
	{
	double f；
	f = ldexp(1.0，10)；
	printf("1.0 * 2^10 = %f\n"，f)；
	}

返回值	本函数返回浮点数 f 指数部分加上整数 i 后得到的新浮点数

参　阅	frexp()

10.27　LDIV 函数

函　数	LDIV
提　要	#include <stdlib.h> ldiv_t ldiv (long number，long denom)
描　述	ldiv( )函数实现分子除以分母，得到商和余数。商的符号与精确商的符号一致，绝对值是一个小于精确商绝对值的最大整数。Ldiv( )函数与 div( )函数类似；不同点在于，前者的函数参数和返回值（结构体 ldiv_t）的成员都是长整型数据。

例　程	#include <stdlib.h>
	#include <stdio.h>
	void
	main (void)
	{
	　ldiv_t lt；
	　lt = ldiv(1234567，12345)；
	　printf("Quotient = %ld，remainder = %ld\n"，lt.quot，lt.rem)；
	}

返回值	返回值是结构体 ldiv_t
参　阅	div( )

10.28　LOCALTIME 函数

函　数	LOCALTIME
提　要	#include <time.h> struct tm * localtime (time_t * t)
描　述	本函数把指针 t 所指的时间分解并且存于结构体中，精确度为秒。其中，t 所指的时间必须自 1970 年 1 月 1 日 0 时 0 分 0 秒起，所用的结构体被定义在 time.h 文件中。localtime( )函数需要考虑全局整型变量 time_zone 中的内容，因为它包含有本地时区位于格林威治以西的时区数值。由于在 MS-DOS 环境下无法预先确定这个值，所以，在缺省的条件下，localtime( )函数的返回值将与 gmtime( )的相同

例　程	#include <stdio.h>
	#include <time.h>
	char * wday[] = {
	　"Sunday"，"Monday"，"Tuesday"，"Wednesday"，
	　"Thursday"，"Friday"，"Saturday"
	}；
	void
	main (void)
	{
	　time_t clock；

	struct tm * tp； time(&clock)； tp = localtime(&clock)； printf("Today is %s\n"，wday[tp->tm_wday])； }
返回值	本函数返回 tm 结构体型数据。注意：由于编译器不会提供 time( )程序，故它需要由用户给定。详情请参阅 time( )函数。
参　阅	ctime( )，asctime( )，time( )

10.29　LOG 函数

函　数	LOG、LOG10
提　要	#include <math.h> double log (double f) double log10 (double f)
描　述	log( )函数返回 f 的自然对数值。log10( )函数返回 f 以 10 为底的对数值

例　程	#include <math.h>
	#include <stdio.h>
	void
	main (void)
	{
	double f；
	for(f = 1.0；f <= 10.0；f += 1.0)
	printf("log(%1.0f) = %f\n"，f，log(f))；
	}

返回值	如果函数参数为负，返回值为 0
参　阅	exp( )，pow( )

10.30　MEMCHR 函数

函　数	MEMCHR
提　要	#include <string.h> /* 初级和中级系列单片机 */ const void * memchr (const void * block，int val，size_t length) /* 高级系列单片机*/ void * memchr (const void * block，int val，size_t length)
描　述	memchr( )函数与 strchr( )函数在功能上类似;但前者没有在字符串中寻找 null（空）中止字符的功能。memchr( )函数实现在一段规定了长度的内存区域中寻找特定的字节。它的函数参数包括指向被寻内存区域的指针、被寻字节的值和被寻内存区域的长度。函数将返回一个指针，该指针指向被寻内存区域中被寻字节首次出现的单元。

例　程	#include <string.h>
	#include <stdio.h>
	unsigned int ary[ ] = {1，5，0x6789，0x23}；
	void
	main (void)
	{
	　char * cp；
	　cp = memchr(ary，0x89，sizeof ary)；
	　if(!cp)
	　　printf("not found\n")；
	　else
	　　printf("Found at offset %u\n"，cp - (char *)ary)；
	}

返回值	函数返回指针。该指针指向被寻内存区域中被寻字节首次
参　阅	strchr( )

10.31　MEMCMP 函数

函　数	MEMCMP
提　要	#include <string.h> int memcmp (const void * s1, const void * s2, size_t n)
描　述	memcmp()函数的功能是比较两块长度为 n 的内存中变量的大小，类似 strncmp()函数返回一个有符号数。与 strncmp()函数不同的是，memcmp()函数没有空格结束符。ASCII 码字符顺序被用来比较；但如果内存块中包含非 ASCII 码字符，则返回值不确定。测试是否相等总是可靠的。

例　程	#include <stdio.h>
	#include <string.h>
	void
	main (void)
	{
	　int buf[10], cow[10], i;
	　buf[0] = 1;
	　buf[2] = 4;
	　cow[0] = 1;
	　cow[2] = 5;
	　buf[1] = 3;
	　cow[1] = 3;
	　i = memcmp(buf, cow, 3*sizeof(int));
	　if(i < 0)
	　　printf("less than\n");
	　else if(i > 0)
	　　printf("Greater than\n");
	　else
	　　printf("Equal\n");
	}

返回值	当内存块变量 s1 分别小于、等于或大于内存块 s2 变量时，函数返回值分别为-1，0 或 1。

参　阅	strncpy()， strncmp()， strchr()， memset()， memchr()

10.32　MEMCPY 函数

函　数	MEMCPY
提　要	#include <string.h> /* 低级或中级系列单片机 */ void * memcpy (void * d, const void * s, size_t n) /* 高级系列单片机 */ far void * memcpy (far void * d, const void * s, size_t n)
描　述	memcpy()函数的功能是，将指针 s 指向的、内存开始的 n 个字节复制到指针 d 指向的、内存开始的单元。复制重叠区的结果不确定。与 strcpy()函数不同的是，memcpy()复制的是一定数量的字节，而不是复制所有结束符前的数据。

例　程	#include <string.h>
	#include <stdio.h>
	void
	main (void)
	{
	　char buf[80];
	　memset(buf, 0, sizeof buf);
	　memcpy(buf, "a partial string", 10);
	　printf("buf = ’%s’\n", buf);
	}

返回值	memcpy()函数返回值为函数的第一个参数
参　阅	strncpy()， strncmp()， strchr()， memset()

10.32　MEMMOVE 函数

函　数	MEMMOVE
提　要	#include <string.h> /* 低级或中级系列单片机 */ void * memmove (void * s1, const void * s2, size_t n) /* 高级系列单片机 */ far void * memmove (far void * s1, const void * s2, size_t n)
描　述	memmove()函数与 memcpy()函数相似，但 memmove()函数能对重叠区进行准确的复制。也就是说，它可以适当向前或向后，正确地从一个块复制到另一个块，并将它覆盖。
例　程	
返回值	memmove()函数同样返回它的第一个参数
参　阅	strncpy(), strncmp(), strchr(), memcpy()

10.33 MEMSET 函数

函　数	MEMSET
提　要	#include <string.h> /* 低级和中级系列的单片机 */ void * memset (void * s, int c, size_t n) /* 高级系列单片机 */ far void * memset (far void * s, int c, size_t n)
描　述	memset()函数将字节 c 存储到指针 s 指向的，内存开始的 n 个内存字节。

例　程	#include <string.h>
	#include <stdio.h>
	void
	main (void)
	{
	　char abuf[20];
	　strcpy(abuf, "This is a string");
	　memset(abuf, 'x', 5);
	　printf("buf = '%s'\n", abuf);
	}

返回值	
参　阅	strncpy(), strncmp(), strchr(), memcpy(), memchr()

10.34 MODF 函数

函　数	MODF
提　要	#include <math.h> double modf (double value, double * iptr)
描　述	modf()函数将参数 value 分为整数和小数 2 部分，每部分都和 value 的符号相同。例如，-3.17 将被分为整数部分（-3）和小数部分（-0.17）。其中整数部分以双精度数据类型存储在指针 iptr 指向的单元中。

例　程	#include <math.h>
	#include <stdio.h>
	void
	main (void)
	{
	　double i_val, f_val;
	　f_val = modf(-3.17, &i_val);
	}

返回值	函数返回值为 value 的带符号小数部分
参　阅	

10.35 PERSIST_CHECK 函数

函　数	PERSIST_CHECK, PERSIST_VALIDATE
提　要	#include <sys.h> int persist_check (int flag) void persist_validate (void)
描　述	persist_check()函数要用到非可变（non-volatile）的 RAM 变量，这些变量在定义时被加上限定词 persistent。在测试 NVRAM（非可变 RAM）区域时，先调用 persist_validate()函数，并用到一个存储在隐藏变量中的虚拟数据，且由 persist_validate()函数计算得到一个测试结果。如果虚拟数据和测试结果都正确，则返回值为真（非零）。如果都不正确，则返回零。在这种情况下，函数返回零并且重新检测 NVRAM 区域（通过调用 persist_validate()函数）。函数被执行的条件是标志变量变量不为 0。persist_validate()函数应在每次转换为永久变量之后调用。它将重新建立虚拟数据和计算测试结果。

例　程	#include <sys.h>
	#include <stdio.h>
	persistent long reset_count;
	void
	main (void)
	{
	if(!persist_check(1))
	printf("Reset count invalid - zeroed\n");
	else
	printf("Reset number %ld\n", reset_count);
	reset_count++; /* update count */
	persist_validate(); /* and checksum */
	for(;;)
	continue; /* sleep until next reset */
	}

返回值	如果 NVRAM 区域无效，则返回值为假（零）；如果 NVRAM 区域有效，则返回值为真（非零）。
参　阅	

10.36 POW 函数

函　数	POW
提　要	#include <math.h> double pow (double f, double p)
描　述	pow()函数表示第一个参数 f 的 p 次幂
例　程	#include <math.h> #include <stdio.h> void main (void) {

	double f;
	for(f = 1.0 ; f <= 10.0 ; f += 1.0)
	printf("pow(2, %1.0f) = %f\n", f, pow(2, f));
	}

返回值	返回值为 f 的 p 次幂
参 阅	log(), log10(), exp()

10.38 PRINTF 函数

函 数	PRINTF
提 要	#include <stdio.h> unsigned char printf (const char * fmt, ...)
描 述	printf()函数是一个格式输出子程序，其运行的基础是标准输出（staout）。它有对应的程序形成字符缓冲区（sprintf()函数）。printf()函数以格式字符串、一系列 0 及其它作为参数。格式字符串都转换为一定的格式，每一规格化都用来输出变量表。 转换格式的形式为%m.nc。其中%表示格式，m 表示选择的字符宽度，n 表示选择的精度，c 为一个字母表示规格类型。字符宽度和精度只适于中级和高级系列单片机，并且精度只对格式%s 有效。 如果指针变量为十进制常数，例如格式为%*d 时，则一个整型数将从表中被取出，提供给指针变量。对低级系列单片机而言，有下列转换格式： o x X u d 即分别为整型格式——八进制、十六进制、十六进制、十进制和十进制。其中 d 为有符号十进制数，其它为无符号。其精度值为被输出数的总的位数，也可以强制在前面加 0。例如%8.4x 将产生一 8 位十六进制数，其中前 4 位为 0，后为十六进制数。X 输出的十六进制数中，字母为 A～F，x 输出的十六进制中字母为 a～f。当格式发生变化时，八进制格式前要加 0，十六进制格式的前面要加 0x 或 0X。 S 打印一个字符串——函数参数值被认为是字符型指针。最多从字符串中取 n 个字符打印，字符宽度为 m。 C 函数参数被认为一个单字节字符并可自由打印。 任何其它有格式规定的字符将被输出。那么%%将产生一个百分号。 对中级和高级系列单片机而言，转换格式在低级系列单片机的基础上再加上： l 长整型格式——在整型格式前加上关键字母 l 即表示长整型变量。 f 浮点格式——总的宽度为 m，小数点后的位数为 n。如果 n 没有写出，则默认为 6。如果精度为 0，则小数点被省略，除非精度已预先定义
例 程	printf("Total = %4d%%", 23) 输出为 ’Total = 23% ’ printf("Size is %lx" , size) 这里 size 为长整型十六进制变量。注意当使用%s 时，精度只对中级和高级系列单片机有效。 printf("Name = %.8s", "a1234567890") 输出为 ’Name = a1234567 ’ 字符变量宽度只对中级和高级系列单片机有效。 printf("xx%*d", 3, 4)

	输出为 ’xx 4’
	/* vprintf 例程 */
	#include <stdio.h>
	int
	error (char * s, ...)
	{
	va_list ap;
	va_start(ap, s);
	printf("Error: ");
	vprintf(s, ap);
	putchar(’\n’);
	va_end(ap);
	}
	void
	main (void)
	{
	int i;
	i = 3;
	error("testing 1 2 %d", i);
	}

返回值	printf()将返回的字符值写到标准输出口。注意返回值为字符型，而不是整形。注意：printf 函数的部分特征只对中级和高级系列单片机有效。详见描述部分。输出浮点数要求浮点数不大于最大长整型变量。为了使用长整型变量或浮点数格式必须将适当的函数库包含进来。参见有关 PICC －L 的描述以及有关 HPDPIC 长整型格式在 printf 的菜单选项。
参 阅	

10.39 RAND 函数

函 数	RAND
提 要	#include <stdlib.h> int rand (void)
描 述	rand()函数用来产生一个随机数数据。它返回一个 0～32767 的整数，并且这个整数在每次被调用后，以随机数据形式出现。这一运算规则将产生一个从同一起点开始的确定顺序。起点通过调用 srand()函数获得。下面的例程说明了每次通过调用 time()函数获得不同的起点。

例 程	#include <stdlib.h>
	#include <stdio.h>
	#include <time.h>
	void
	main (void)
	{
	time_t toc;
	int i;
	time(&toc);
	srand((int)toc);
	for(i = 0 ; i != 10 ; i++)

	printf("%d\t", rand());
	putchar('\n');
	}

返回值	注意：例程中需要用户自己提供 time()函数，因为它不能由汇编器产生。更详细的情况参见 time()函数。
参　阅	参见 srand()函数

10.40　SIN 函数

函　数	SIN
提　要	#include <math.h> double sin (double f)
描　述	这个函数返回参数的正弦值

例　程	#include <math.h>
	#include <stdio.h>
	#define C 3.141592/180.0
	void
	main (void)
	{
	double i;
	for(i = 0 ; i <= 180.0 ; i += 10)
	printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i*C), cos(i*C));
	}

返回值	返回值为参数 f 的正弦值
参　阅	cos(), tan(), asin(), acos(), atan(), atan2()

10.41　SPRINTF 函数

函　数	SPRINTF
提　要	#include <stdio.h> /* 中级和低级系列单片机 */ unsigned char sprintf (char *buf, const char * fmt, ...) /* 高级系列单片机 */ unsigned char sprintf (far char *buf, const char * fmt, ...)
描　述	sprintf()函数和 printf()函数操作基本相同; 只是输出在不同的输出终端，所有的字符被放到 buf 缓冲器。字符串带有空格结束符，buf 缓冲器中的数据被返回。
例　程	参见 printf()函数。
返回值	sprintf()函数的返回值为被放入缓冲器中的数据。注意，返回值为字符型而非整型。注意：对高级单片机而言，缓冲器是通过长指针访问的
参　阅	

10.42 SQRT 函数

函　数	SQRT
提　要	#include <math.h> double sqrt (double f)
描　述	sqrt()函数利用牛顿法得到参数的近似平方根

例　程	#include <math.h>
	#include <stdio.h>
	void
	main (void)
	{
	　double i;
	　for(i = 0 ; i <= 20.0 ; i += 1.0)
	　　printf("square root of %.1f = %f\n", i, sqrt(i));
	}

返回值	返回值为参数的平方根。注意：如果参数为负则出现错误
参　阅	参见 exp()函数。

10.43 SRAND 函数

函　数	SRAND
提　要	#include <stdlib.h> void srand (unsigned int seed)
描　述	srand()函数是在调用 rand()函数时被用来初始化随机数据发生器的。它为 rand()函数产生不同起点虚拟数据顺序提供一个机制。在 z80 上，随机数据最好从新的寄存器获得。否则，控制台的响应时间或系统时间将充当这一数据。

例　程	#include <stdlib.h>
	#include <stdio.h>
	#include <time.h>
	void
	main (void)
	{
	　time_t toc;
	　int i;
	　time(&toc);
	　srand((int)toc);
	　for(i = 0 ; i != 10 ; i++)
	　printf("%d\t", rand());
	　putchar('\n');
	}

返回值	
参　阅	参见 rand()函数。

10.44 STRCAT 函数

函　数	STRCAT
提　要	#include <string.h> /* 中级和低级系列单片机 */ char * strcat (char * s1, const char * s2) /* 高级系列单片机 */ far char * strcat (far char * s1, const char * s2)
描　述	这个函数将字符串 s2 连接到字符串 s1 的后面。新的字符串以空格作为结束符。指针型参数 s1 指向的字符数组必须保证大于结果字符串。

例　程	#include <string.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buffer[256];
	char * s1, * s2;
	strcpy(buffer, "Start of line");
	s1 = buffer;
	s2 = " ... end of line";
	strcat(s1, s2);
	printf("Length = %d\n", strlen(buffer));
	printf("string = \"%s\"\n", buffer);
	}

返回值	即为字符串 s1
参　阅	strcpy(), strcmp(), strncat(), strlen()

10.45 STRCHR 函数

函　数	STRCHR, STRICH
提　要	#include <string.h> /* 中级和低级系列单片机 */ const char * strchr (const char * s, int c) const char * strichr (const char * s, int c) /* 高级系列单片机 */ char * strchr (const char * s, int c) char * strichr (const char * s, int c)
描　述	strchr()函数查找字符串 s 中是否出现字符变量 c。如果找到了，则字符指针被返回;否则返回 0。Strichr()函数与 strchr()函数作用相同
例　程	#include <strings.h> #include <stdio.h> void main (void) {

	static char temp[] = "Here it is...";
	char c = 's';
	if(strchr(temp, c))
	printf("Character %c was found in string\n", c);
	else
	printf("No character was found in string");
	}

返回值	如果找到，则返回第一个字符的指针；否则返回 0。注意：函数对字符使用整型参数，只有低 8 位有效。
参　阅	strrchr(), strlen(), strcmp()

10.46　STRCMP 函数

函　数	STRCMP, STRICMP
提　要	#include <string.h> int strcmp (const char * s1, const char * s2) int stricmp (const char * s1, const char * s2)
描　述	strcmp()函数用来比较 2 个字符串的大小。字符串带有空格结束符，根据字符串 s1 是否小于、等于或大于字符串 s2，返回一个有符号整数。比较是根据 ASCII 字母的顺序表进行的。Stricmp()函数功能和 strcmp()函数完全一样。

例　程	include <string.h>
	#include <stdio.h>
	void
	main (void)
	{
	int i;
	if((i = strcmp("ABC", "ABc")) < 0)
	printf("ABC is less than ABc\n");
	else if(i > 0)
	printf("ABC is greater than ABc\n");
	else
	printf("ABC is equal to ABc\n");
	}

返回值	返回一个有符号整数.注意：其它的 C 应用程序也可以采用不同的字母顺序表。返回值为正、零或负，也就是说不一定是-1 或 1。
参　阅	strlen(), strncmp(), strcpy(), strcat()

10.47　STRCPY 函数

函　数	STRCPY
提　要	#include <string.h> /* 低级和中级系列单片机 */ char * strcpy (char * s1, const char * s2)

	/* 高级系列单片机 */ far char * strcpy (far char * s1, const char * s2)
描　述	这个函数将以空格键结束的字符串 s2 拷贝到 s1 指向的字符数组。目的数组必须足够大，以容纳包括空格在内的字符串 s2。

例　程	#include <string.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buffer[256];
	char * s1, * s2;
	strcpy(buffer, "Start of line");
	s1 = buffer;
	s2 = " ... end of line";
	strcat(s1, s2);
	printf("Length = %d\n", strlen(buffer));
	printf("string = \"%s\"\n", buffer);
	}

返回值	目的数组被返回
参　阅	strncpy(), strlen(), strcat(), strlen()

10.48　STRCSPN 函数

函　数	STRCSPN
提　要	#include <string.h> size_t strcspn (const char * s1, const char * s2)
描　述	strcspn()函数用于取得在字符串常数 s1 中有而字符串常数 s2 中没有的字符的长度。

例　程	#include <stdio.h>
	#include <string.h>
	void
	main (void)
	{
	static char set[] = "xyz";
	printf("%d\n", strcspn("abcdevwxyz", set));
	printf("%d\n", strcspn("xxxbcadefs", set));
	printf("%d\n", strcspn("1234567890", set));
	}

返回值	为剩余部分的长度
参　阅	参见 strspn()函数

10.49　STRNCAT 函数

函　数	STRNCAT
提　要	#include <string.h> /* 低级和中级系列单片机 */ char * strncat (char * s1, const char * s2, size_t n) /* 高级系列单片机 */ far char * strncat (far char * s1, const char * s2, size_t n)
描　述	函数将字符串 s2 连接到字符串 s1 的尾端。最多只有 n 个字符被拷贝，结果包含空格结束符。指针 s1 指向的字符数组应足够大，以容纳结果字符串。

例　程	#include <string.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buffer[256];
	char * s1, * s2;
	strcpy(buffer, "Start of line");
	s1 = buffer;
	s2 = " ... end of line";
	strncat(s1, s2, 5);
	printf("Length = %d\n", strlen(buffer));
	printf("string = \"%s\"\n", buffer);
	}

返回值	即为字符串 s1
参　阅	strcpy(), strcmp(), strcat(), strlen()

10.50　STRNCMP 函数

函　数	STRNCMP, STRNICMP
提　要	#include <string.h> int strncmp (const char * s1, const char * s2, size_t n) int strnicmp (const char * s1, const char * s2, size_t n)
描　述	strncmp()函数用来比较两个带有空格结束符的字符串的大小，最多比较 n 个字符。根据字符串 s1 是否小于、等于或大于字符串 s2，返回一个有符号数。比较是根据 ASCII 字母顺序进行的。Strnicmp()函数与之相同。

例　程	#include <stdio.h>
	#include <string.h>
	void
	main (void)
	{
	int i;
	i = strcmp("abcxyz", "abcxyz");
	if(i == 0)
	printf("Both strings are equal\n");
	else if(i > 0)
	printf("String 2 less than string 1\n");
	else
	printf("String 2 is greater than string 1\n");

	}
返回值	有符号整数。注意：其它 C 应用函数可以采用不同的字母顺序。返回值为负、零或正，并不一定是-1 或 1
参　阅	strlen(), strcmp(), strcpy(), strcat()

10.51　STRNCPY 函数

函　数	STRNCPY
提　要	#include <string.h> /* 低级和中级系列单片机 */ char * strncpy (char * s1, const char * s2, size_t n) /* 高级系列单片机 */ far char * strncpy (far char * s1, const char * s2, size_t n)
描　述	这个函数将带结束符的字符串 s2 拷贝到字符指针 s1 指向的字符数组。最多有 n 个字符被拷贝。如果 s2 的长度大于 n，则结果中不包含结束符。目的数组必须足够大，以容纳包括结束符在内的新字符串。

例　程	#include <string.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buffer[256];
	char * s1, * s2;
	strncpy(buffer, "Start of line", 6);
	s1 = buffer;
	s2 = " ... end of line";
	strcat(s1, s2);
	printf("Length = %d\n", strlen(buffer));
	printf("string = \"%s\"\n", buffer);
	}

返回值	指针 s1 指向的目的缓冲区。
参　阅	strcpy(), strcat(), strlen(), strcmp()

10.52　STRPBRK 函数

函　数	STRPBRK
提　要	#include <string.h> /* 低级和中级系列单片机 */ const char * strpbrk (const char * s1, const char * s2) /* 高级系列单片机 */ char * strpbrk (const char * s1, const char * s2)
描　述	strpbrk()函数查找字符串 s1 是否包含字符串 s2 的字符。如果包含，则返回被找到

	的第一个字符的指针；否则不返回任何值

例　程	#include <stdio.h>
	#include <string.h>
	void
	main (void)
	{
	　char * str = "This is a string.";
	　while(str != NULL)
	　{
	　　printf("%s\n", str);
	　　str = strpbrk(str+1, "aeiou");
	　}
	}

返回值	第一个匹配的字符，否则返回值为空
参　阅	

10.53 STRRCHR 函数

函　数	STRRCHR, STRRICHR
提　要	#include <string.h> /* 中级和低级系列单片机 */ const char * strrchr (char * s, int c) const char * strrichr (char * s, int c) /* 高级系列单片机 */ char * strrchr (char * s, int c) char * strrichr (char * s, int c)
描　述	strrchr()函数和 strchr()函数相似；但它从字符串的尾端开始查找，也就是说，其返回值为字符 c 最后一次在字符串中出现时的指针。如果没出现，则返回值为空。strrichr()函数和 strrchr()函数完全一样。

例　程	#include <stdio.h>
	#include <string.h>
	void
	main (void)
	{
	　char * str = "This is a string.";
	　while(str != NULL)
	　{
	　　printf("%s\n", str);
	　　str = strrchr(str+1, 's');
	　}
	}

返回值	字符指针，或者返回值为空。
参　阅	strchr(), strlen(), strcmp(), strcpy(), strcat()

10.54 STRSPN 函数

函　数	STRSPN
提　要	#include <string.h> size_t strspn (const char * s1, const char * s2)
描　述	strspn()函数返回字符串 s1 中包含的、完全由字符串 s2 组成的字符的长度。

例　程	#include <stdio.h>
	#include <string.h>
	void
	main (void)
	{
	　printf("%d\n", strspn("This is a string", "This"));
	　printf("%d\n", strspn("This is a string", "this"));
	}

返回值	部分长度
参　阅	参见 strcspn()函数

10.55 STRSTR 函数

函　数	STRSTR, STRISTR
提　要	#include <string.h> /*中级和低级系列单片机 */ const char * strstr (const char * s1, const char * s2) const char * stristr (const char * s1, const char * s2) /* 高级系列单片机 */ char * strstr (const char * s1, const char * s2) char * stristr (const char * s1, const char * s2)
描　述	strstr()函数返回字符数组 s1 中第一次出现字符数组 s2 的指针位置。stristr()与之一样。

例　程	#include <stdio.h>
	#include <string.h>
	void
	main (void)
	{
	　printf("%d\n", strstr("This is a string", "str"));
	}

返回值	字符指针。如果每有字符串被找到，则返回为空
参　阅	

10.55 STRTOK 函数

函　数	STRTOK
提　要	#include <string.h> /*中级和低级系列单片机 */ char * strtok (char * s1, const char * s2) /*高级系列单片机 */ far char * strtok (far char * s1, const char * s2)
描　述	多次调用 strtok()函数可以将字符串 s1 分为几个独立的部分。s1 中包含 0 或者其它一些包含在字符串 s2 中的字符。这个调用返回一个指向分隔符的第一个字符的指针。如果不存在分隔符则返回为空。分隔符将被空格所覆盖，从而使目前的分隔标记不再起作用。调用 strtok()函数之后，应使指针 s1 为空。这样，将从后向前查找，又返回分隔符中第一个字符的指针。如果没找到，则返回为空

例　程	#include <stdio.h>
	#include <string.h>
	void
	main (void)
	{
	char * ptr;
	char * buf = "This is a string of words.";
	char * sep_tok = ".,?! ";
	ptr = strtok(buf, sep_tok);
	while(ptr != NULL)
	{
	printf("%s\n", ptr);
	ptr = strtok(NULL, sep_tok);
	}
	}

返回值	分隔符中的第一个字符的指针，或者返回为空。注意：每次调用函数时,分隔字符串 s2 可以不一样。
参　阅	

10.56 TAN 函数

函　数	TAN
提　要	#include <math.h> double tan (double f)
描　述	tan()函数用来计算参数 f 的正切值
例　程	#include <math.h> #include <stdio.h> #define C 3.141592/180.0 void main (void) { double i;

	for(i = 0 ; i <= 180.0 ; i += 10)
	printf("tan(%3.0f) = %f\n", i, tan(i*C));
	}

返回值	f 的正切值
参　阅	sin(), cos(), asin(), acos(), atan(), atan2()

10.57　TIME 函数

函　数	TIME
提　要	#include <time.h> time_t time (time_t * t)
描　述	数需要目标系统提供当前时间，函数没有给出。这个函数需由用户实现。在运行时，函数以秒为单位返回当前时间。当前时间从 1970 年 1 月 1 日 0 点 0 分 0 秒开始有效。如果参数 t 不为空，那么这个值同样被保存到 t 所指的内存单元。

例　程	#include <stdio.h>
	#include <time.h>
	void
	main (void)
	{
	time_t clock;
	time(&clock);
	printf("%s", ctime(&clock));
	}

返回值	被执行的函数将返回从 1970 年 1 月 1 日 0 点 0 分 0 秒开始的精确到秒的当前时间。注意：time()函数没有被提供，用户必须采用前面提到的规范来执行这一程序
参　阅	ctime(), gmtime(), localtime(), asctime()

10.58　TOLOWER 函数

函　数	TOLOWER, TOUPPER, TOASCII
提　要	#include <ctype.h> char toupper (int c) char tolower (int c) char toascii (int c)
描　述	toupper()函数将小写字母转换为大写字母；而 tolower()则与之相反；toascii()用来保证得到一个 0～0177 之间的结果。如果参数不为字母表中的字母，则 toupper()函数和 tolower()函数都返回它们原来的参数值
例　程	#include <stdio.h> #include <ctype.h> #include <string.h> void main (void) { char * array1 = "aBcDE";

	int i;
	for(i=0;i < strlen(array1); ++i)
	{
	printf("%c", tolower(array1[i]));
	}
	printf("\n");
	}

返回值	islower(), isupper(), isascii()等
参　阅	

10.59　STRLEN 函数

函　数	STRLEN
提　要	#include <string.h> size_t strlen (const char * s)
描　述	strlen()用来测量字符串 s1 的长度，不包括空格结束符。

例　程	#include <string.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buffer[256];
	char * s1, * s2;
	strcpy(buffer, "Start of line");
	s1 = buffer;
	s2 = " ... end of line";
	strcat(s1, s2);
	printf("Length = %d\n", strlen(buffer));
	printf("string = \"%s\"\n", buffer);
	}

返回值	即为不包括结束符在内的字符长度
参　阅	

10.60　VA_START 函数

函　数	VA_START,VA_ARG,VA_END
提　要	#include <stdarg.h> void va_start (va_list ap, parmN) type va_arg (ap, type) void va_end (va_list ap)
描　述	这些函数的宏提供一个方便的路径来传递函数参数。函数定义时，函数参数用省略号替代。这里函数参数的个数及类型在汇编时并不知道。 函数最右端的参数（用 parmN 表示），在宏汇编中起着非常重要的作用，因为它是得到更多参数的开始。函数可以取不同数量的参数，不同类型的 va_list（变量表）应事先定义，然后激活带有名为 parmN 的一系列参数的宏 va_start()。将变

	量初始化，从而允许调用宏 va_arg()，得到其它的参数。 每次调用 va_arg()需有两个参数；一个在前面已定义，另一个参数也需要说明其类型。注意所有的参数都将被自动加宽为整型、无符号整型和双精度型。例如，如果一个参数为字符型，则字符型自动转换为整型，函数调用的形式相当于 va_arg(ap,int)。 下面用例子说明带有一个整型变量和一些其它变量作为参数的函数。在这个例子中，函数将得到一个字符型指针；但要注意编译器并不知道，程序员对参数正确性负责

例　程	#include <stdio.h>
	#include <stdarg.h>
	void
	pf (int a, ...)
	{
	va_list ap;
	va_start(ap, a);
	while(a--)
	puts(va_arg(ap, char *));
	va_end(ap);
	}
	void
	main (void)
	{
	pf(3, "Line 1", "line 2", "line 3");
	}

返回值	
参　阅	

10.61　XTOI 函数

函　数	XTOI
提　要	#include <stdlib.h> unsigned xtoi (const char * s)
描　述	xtoi()函数扫描参数中的字符串。它跳过前面的空格，读到符号后，将用 ASCII 码表示的十六进制数转换为整型。

例　程	#include <stdlib.h>
	#include <stdio.h>
	void
	main (void)
	{
	char buf[80];
	int i;
	gets(buf);
	i = xtoi(buf);
	printf("Read %s: converted to %x\n", buf, i);
	}

返回值	有符号整数。如果字符串中不包含数，则返回零
参　阅	参见 atoi()函数