



CMS 单片机 C 语言程序设计指南

V1.1

注意

使用手册中所出现的说明在出版当时相信是正确的，然而中微公司对于说明书的使用不负任何责任。文中提到的应用目的仅仅是用来说明，中微公司不保证或表示这些没有进一步修改的应用是适当的，也不推荐它的产品使用在会由于故障或其他原因可能会造成人身危害的应用。中微公司产品不授权使用于救生、维生器件等系统中作为元件。中微公司拥有不事先通知而修改产品的权利，对于最新的资讯，请参考我们的网址 <http://www.mcu.com.cn>



目录

第一章、前言	3
第二章、CMS C程序框架及数据类型.....	4
2.1、CMS 单片机的 C 语言原程序基本框架.....	4
2.2、CMS C 中的变量定义	6
2.2.1、CMS C 中的基本变量类型.....	6
2.2.2、CMS C 中的高级变量.....	6
2.3、CMS C 对数据寄存器 bank 的管理	7
2.4、CMS C 中的局部变量	8
2.5、CMS C 中的位变量	9
2.6、CMS C 中的浮点数	11
2.7、CMS C 中变量的绝对定位	11
2.8、CMS C 的其它变量修饰关键词	12
2.9、CMS C 中的指针	14
第三章、CMS C中的函数	16
3.1、函数概述	16
3.2、函数的代码长度限制	16
3.3、函数调用层次的控制	17
3.4、函数类型声明	18
3.5、中断函数的实现	18
3.6、标准库函数	20
第四章、程序流程控制	21
4.1、if 语句	21
4.2、switch 语句	22
4.3、for 语句.....	23
4.4、while 语句.....	24
4.5、do...while 语句	24
4.6、goto 语句	25
4.7、break 与 continue 语句	25
第五章、快速开始C应用程序的设计.....	27
5.1、定义主函数 main（）	27
5.2、定义全局变量	27
5.3、定义子函数	28
5.4、定义中断服务函数	28
5.5、其它	29



第一章、前言

由于八位微控制器的内存空间，不论是程序内存(program memory space)或是数据存储器 (ram memory space)，皆是有限制的，通常会使用汇编语言开发应用程序。但是越来越多的微控制器支持更多的内存以及更多的功能，使得程序也相对的扩大。如果仍然使用汇编语言开发程序，不但费时费力，未来在维护及扩增功能的工作上也相当困难。因此，使用高阶程序语言，例如C语言，来开发应用程序就是一种可行的趋势。C语言是高阶程序语言中的一种，它具有高度的可读性及可移植性(portability)，除了能够快速地完成应用程序的开发与侦错，也很容易移植到其它的微控制器上。当程序需要缩减或扩充功能时，也很容易的完成，因此很适合于微控制器的程序开发。本书主要是以CMS C语言为主，说明如何使用CMS C语言撰写中微微控制器的应用程序，包括C的程序架构，C语言的一般用法，特殊用法及应用范例。书中将说明在开发微控制器的应用程序时需要注意的地方及如何撰写会比较恰当，并配以实例解释。读者可以参考修改或直接采用到自己的程序中，再用开发工具CMS ICE5验证之。

中微半导体公司的C编译器(CMS C)基本上符合ANSI C的标准，除了一点：它不支持函数的递归调用。其主要原因是由于受限于CMS单片机特殊的堆栈结构，CMS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CMS单片机中实现软件堆栈的效率也不是很高，为此，CMS C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，按实际使用的体会，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CMS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。



第二章、CMS C 程序框架及数据类型

2.1、CMS 单片机的 C 语言原程序基本框架

基于CMS C编译环境编写CMS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提供的“cms.h”文件，实现单片机内特殊寄存器和其它特殊符号的声明；
- 声明本模块内被调用的所有函数的类型，CMS C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 2-1 为一个 C 原程序的范例，供大家参考。

```
#include <cms.h>           //包含单片机内部资源预定义

#include "user.h"           //包含自定义头文件


//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);


//定义变量
```



```
unsigned char second, minute, hour;

bit flag1, flag2;

void main(void)
{
    SetSFR();

    while(1)                //程序工作主循环
    {
        asm("clrwdt");      //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
    }
}
```



2.2、CMS C 中的变量定义

2.2.1、CMS C 中的基本变量类型

CMS C 支持 1,2,4 字节的基本类型，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。表 2-1 列出了所有数据类型及它们所占空间大小。

表 2-1 CMS C 支持的基本数据类型

类型	长度(位数)	数字类型	值
bit	1	逻辑类型	0 或 1
char	8	有符号字符型	-128 至+127
unsigned char	8	无符号字符型	0 值 255
short	16	有符号短整型	-32768 至+32767
unsigned short	16	无符号短整型	0 至 65535
int	16	有符号整数型	-32768 至+32767
unsigned int	16	无符号整数型	0 至 65535
long	32	有符号长整型	-2147483648 至+2147483647
unsigned long	32	无符号长整型	0 至 4294967295
float	24	浮点型	
double	24	双精度浮点型	

2.2.2、CMS C 中的高级变量

基于表 2-1 的基本变量，除了 bit 型位变量外，CMS C 完全支持数组、结构体和联合体等复合型高级变量，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];
```

结构体:

```
struct commInData {  
    unsigned char inBuff[8];  
    unsigned char getPtr, putPtr;  
};
```

联合体:

```
union int_Byte {  
    unsigned char c[2];  
    unsigned int i;  
};
```

2.3、CMS C 对数据寄存器 bank 的管理

CMS C 会自动对单片机中数据寄存器的 bank 进行管理。但为了使编译器产生最高效的机器码, 程序员也可在定义用户变量时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, 编译器将自动匹配数据寄存器长度分配空间, 例如下面所定义的这些变量:

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

如果用户想自行决定数据所在 bank, 需要在其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
```

虽然变量所在的 bank 定位可以由程序员自己决定, 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。

2.4、CMS C 中的局部变量

为节约宝贵的存储空间，CMS C 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 CMS 汇编指令实现的效率最高，但这时不能出现函数递归调用。CMS C 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。



2.5、CMS C 中的位变量

bit 型位变量只能是全局的或静态的。CMS C 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 CMS 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。

CMS C 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 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：

$$\text{字节地址} = 0x123 / 8 = 0x24$$

$$\text{位偏移} = 0x123 \% 8 = 3$$

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。

CMS 单片机的位操作指令是非常高效的。因此，CMS C 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	=>	setb 0x20,7
tmp &= 0xf7	=>	clrb 0x20,3
if (tmp&0xfe)	=>	snzb 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 2-5-1 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。



2.6、CMS C 中的浮点数

CMS C 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，CMS C 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。一般控制系统中关心的是单片机的运行效率，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 型浮点数也是 24 位长。

2.7、CMS C 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。

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 CMS C 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 CMS C 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 CMS C 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，CMS C 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 @((unsigned)& tmpData *8)+0;    //tmpBit0 对应于 tmpData 第 0 位  
bit tmpBit1 @((unsigned)& tmpData *8)+1;    //tmpBit0 对应于 tmpData 第 1 位  
bit tmpBit2 @((unsigned)& 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。

2.8、CMS C 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：

```
unsigned char var1, var2;           //定义了两个变量
```

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 unsigned char var1, var2;    //声明外部变量
```

- **volatile** — 易变型变量声明

CMS C 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“`ret`”的指令形式存放在程序空间，但 CMS C 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。CMS C 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，CMS C 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。CMS 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。

2.9、CMS C 中的指针

CMS C 中指针的基本概念和标准 C 语法没有太多的差别。但是在 CMS 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，CMS C 也不例外。为了生成高效的代码，CMS C 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。例如：

```
unsigned char *ptr0;           //定义一个指向无符号字符型 的指针
```

- 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[]="cmsemicon"; //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，CMS C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CMS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：**CMS C**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。



第三章、CMS C 中的函数

3.1、函数概述

函数是叙述的集合。所有可被执行的叙述必须定义于函数中（main() 是主函数）。使用函数前，必须要宣告及定义函数，否则编译器会发出错误讯息。除了内容的叙述，函数最重要的是参数（arguments）及返回值（return values）。它的格式如下：

```
return-type function_name(var-type arg1, var-type arg2, ...)  
{  
    statements;  
}
```

其中，return-type 是函数的返回值的类型，可使用表2-1中的数据类型。function_name 是函数名，可由字符(character)，数字(0~9)，下划线(underscore) 所组成。var_type 是参数的数据类型。arg1, arg2, .. 是参数名称。

3.2、函数的代码长度限制

C系列的CMS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。

CMS C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内。C系列的CMS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。



3.3、函数调用层次的控制

C系列CMS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。

CMS C在最后编译连接成功后可以生成一个连接定位映射文件 (*.lst)，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 Graphs:

```
_main (ROOT)
    _Init_System
        _DelaySomeTime
        _Refurbish_Sfr
        _Set_Disp_Led
            _Led_disp_time
                _Hex_To_Bcd
                _Led_disp_temper
                    _Hex_To_Bcd
            _Disp_Led
            _Set_Time
        _Isr_Timer (ROOT)
```

例3-2-1 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。



3.4、函数类型声明

CMS C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。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；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。

3.5、中断函数的实现

CMS C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，CMS C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。编译器也会自动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 (TOIE && TOIF)             //判TMR0 中断  
    {
```



```
    T0IF = 0;                                //清除TMR0 中断标志

    //在此加入TMR0 中断服务
}

if (TMR1IE && TMR1IF)                        //判TMR1 中断
{
    TMR1IF = 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3-4-1 C语言中断函数举例

CMS C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CMS C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。

C系列CMS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。



3.6、标准库函数

CMS C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。

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CMS C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，CMS C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。



第四章、程序流程控制

4.1、if 语句

if语句用来判断所给定的条件是否满足,根据判定的结果(真或假)决定执行给出的两种操作之一。if语句的通用语法形式如下:

```
if(表达式1)
    语句组1;
[
    else if(表达式2)
        语句组2;
    .
    .
    else if(表达式m)
        语句组m;
    else
        语句组n;
]
```

说明: 这是一个条件判断的控制叙述。在不同的条件下会需要不同的处理时,可以使用这种叙述做不同的流程控制。如果条件“表达式1”的结果为真(不等于零)则执行“语句组1”后退出条件控制;否则,判断条件“表达式2”的结果是否为真,若为真则执行“语句组2”后退出条件控制;以此类推,若前面的表达式结果均为假,则执行else部分的语句组n。其中中括号[]内的部分可有可无,完全视程序有无需要。各个“语句组”可以不只有一个叙述,如果超过一个叙述时,必须要以左右大括号{}包含这些叙述。

范例:

```
if(minutes>59)
```



```
{  
  
    hours++;  
  
    minutes = 0;  
  
}  
  
else  
  
    minutes++;
```

4.2、switch 语句

switch语句是分支选择语句，用来实现多分支的选择结构。if语句只有两个分支可供选择，而实际中常常需要用到多分支的选择，当然这些可以用嵌套的if语句来处理，但如果分支较多，则嵌套的if语句层数多，程序冗长而且可读性降低。在这种情况下使用 switch-case语句会增加程序可读性，更容易了解程序的功能，它的一般语法格式如下：

```
switch(表达式)  
{  
  
    case 常量表达式1: 语句组1;  
  
    case 常量表达式2: 语句组2;  
  
    .  
  
    .  
  
    case 常量表达式n: 语句组n;  
  
    default : 语句组n+1;  
  
}
```

说明：当表达式的值与某一个case后面的常量表达式的值相等时，就执行此case后面的语句，若所有的case中的常量表达式的值都没有与表达式的值匹配的，就执行default后面的语句，另外，default 部分可有可无，switch后面括号内的表达式可以是一个变量也可以是一个能计算出数值的表达式。每个 case 的最后一条语句后应该加上“break；”叙述行，否则一直到下一个 break 叙述之前或 switch 右括号之前的所有叙述皆会被执行，除非程序确



实要如此执行。

范例：

```
switch(keydata)
{
    case 0:value = 0;break;
    case 1:value = 1;break;
    default:value = 2;break;
}
```

4.3、for 语句

for 语句主要是重复执行相同的功能，其一般的语法形式如下：

for(表达式1; 表达式2; 表达式3) 语句

它的执行过程如下：

- 1)、先求解表达式1;
- 2)、求解表达式2，若其值为真（值为非0），则执行for语句中指定的内嵌语句，然后执行下面第3步。若为假（值为0），则结束循环，转到第5步；
- 3)、求解表达式3;
- 4)、转回上面第2步继续执行；
- 5)、循环结束，执行for语句下面的一条语句。

例如：

```
for (int i = 0;i < 100; i++)
{
    sum = sum + i
}
```



4.4、while 语句

while语句主要用来实现“当型”循环结构。其一般的语法形式如下：

while (表达式) 语句

执行流程：当表达式为非0值时，执行while语句中的内嵌语句，直到表达式的值为0时，退出循环，它的主要特点是：先判断表达式，后执行语句。

例如：

```
unsigned char value[20];  
unsigned char times = 0;  
while(times < 20)  
{  
    value[times] = times;  
    times++;  
}
```

4.5、do...while 语句

do...while语句与while语句类似，不同点在于，do...while语句先执行一次循环体语句，然后在判断表达式。其一般语法形式如下：

```
do  
    循环体语句  
while (表达式)
```

例如：

```
unsigned char buffer[20];  
unsigned char times = 0;  
do  
{
```




```
        buffer[times] = 0;

        times++;

    }while(times < 20)
```

4.6、goto 语句

goto语句主要用于无条件跳转，它的一般形式为：

```
goto label;
```

说明：语句标号label 必须和 goto必须在同一个函数内，也就是不允许跳到其它的函数里。

一般来说，它主要有两种用途：

- 1)、与if语句一起构成循环结构；
- 2)、从循环体中跳转到循环体外；

goto语句的使用会大大降低程序的可读性，不符合结构化的程序设计原则，建议不宜随便使用，只有在不得已时（例如可以大大提高程序效率）才使用。

例如：参加4.7节中的例子

4.7、break 与 continue 语句

语法格式：

```
break;

continue;
```

说明：break 是从循环或是 switch case 跳出，一次只能跳出一层；continue 是跳过 continue 语句之后的其它语句，再从下一循环继续执行。

例如：

```
for( int i =0 ; i < 20 ; i++ )
{
    if(i != 10 ) continue ;
}
```



```
        goto label_1 ;  
    }  
label_1:  
    ...
```



第五章、快速开始 c 应用程序的设计

本章介绍如何快速编写微控制器的C语言应用程序。已熟悉 ANSI C 标准语言的用法或有撰写的经验者，在阅读此章后即可开始设计编写微控制器的C应用程序，以下各节是基本的C 程序成员，某些是必须要有的，如5.1，其它的则视微控制器的功能及应用来决定是否需要。

5.1、定义主函数 main ()

```
#include <cms.h>           //包含单片机相关资源定义的头文件

void main (void)
{
    TRISA = 0;              //一些初始化操作
    TRISB = 0xFF;
    PORTA = 0;
    while(1)                //主循环
    {
        ...                 //程序体
    }
}
```

5.2、定义全局变量

程序在运行中会需要一些变量做为数据存放的地方，最好将经常需使用的变量定义为全局型的变量，在编译程序的大小与执行上皆较佳。具体的定义方式参见第二章。



5.3、定义子函数

视程序的大小及功能决定是否需要定义子函数。基本上，主函数应将应用程序的架构做成模块化，不需要将所有的程序皆放在主函数中。为了能很快的完成及了解应用程序，主函数中只需要包含(调用) 定义各功能的子函数即可，无论在设计或维护程序时皆能很快的进入与完成。例如，关于 LED 的开启，显示及关闭等功能就可分别定义为单独的子函数，如下例。任何其它的函数或其它的应用项目都可去调用这些子函数。若设计成通用型的，也方便建立程序档案库，以供其它应用项目使用。

```
void TurnOn_LED (void)
{
}

void Display_LED (unsigned char *data)
{
}

void TurnOff_LED (void)
{
}
```

子函数的参数和返回值可以是基本类型: int, float, char, void; 也可以是构造类型: struct, union, enum, 数组, 指针类型。

5.4、定义中断服务函数

若微控制器的周边装置具有中断功能，程序也需要此中断机能以完成工作时，则必须定义此周边装置的中断服务函数 (Interrupt Service Routine, ISR)

```
void interrupt ISR(void)
{
}
```



中断服务函数返回的数据类型必须是 `void`，并且不能有参数（必须为 `void`）。详见3.5节中断函数的实现。

5.5、其它

上述的主函数，子函数及中断服务函数不需要定义在同一个原始程序内。为符合结构化程序设计理念，最好是将有相似功能的函数放于同一文件中(如1628、1637等显示芯片驱动类程序)，且每个原文件相应定义一个头文件，声明原文件中定义的所有函数，并使用有意义的函数名，方便日后找寻所要的函数，都文件的定义建议用于条件编译，防止多次包含，例如：在delay.h文件中声明所有延时相关的函数，可以如下写法：

```
#ifndef _DELAY_H_

#define _DELAY_H_


void delayms(unsigned int ms);

void delayus(unsigned char us);


#endif
```