Broadview

电子工程应用精讲系列

MSP430 单片机C语言应用程序设计实例精讲

秦 龙 编著

实例丰富
即学即用

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview
www.broadview.com.cn
技术凝聚实力 · 专业创新出版

MSP430
单片机C语言应用程序设计
实例精讲

围绕**实践**与**经验**，注重**应用**和**实用**

- 6大篇30个实例，全面覆盖典型应用领域，读者可以直接借鉴使用
- 所有例子全部采用C语言实现，具有非常好的可读性和移植性
- 既介绍设计原理、基本步骤和流程，又穿插技巧与注意事项

包括本书用到的所有程序代码，以及用Protel99软件制成的各章的电路图。读者稍加修改便可应用于自己的工作或完成自己的课题（毕业设计）。

图书分类：计算机>单片机

ISBN 7-121-02371-7

9 787121 023712 >

网上订购：www.ourbook.com.cn
第二书店 · 第一服务

责任编辑：张 毅
责 编
责任美编：张子健

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。
ISBN 7-121-02371-7 定价：42.00元（含光盘1张）

电子工程应用精讲系列

MSP430 单片机C语言应用程序设计实例精讲

秦 龙 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书通过大量的典型实例形式，详细介绍了MSP430单片机各种C语言应用专题程序设计的流程、方法、技巧及设计理念。全书共分31章，其中第1章简单介绍了开发MSP430的C语言的基础知识，第2章至第31章为本书的重点，通过30个典型的C语言专题应用实例，详细介绍了MSP430单片机的各种应用开发和使用技术，实例丰富，代表性强，涉及领域广，每个例子都有具体的硬件电路设计和程序设计，对读者有较高的学习和参考价值。

全书语言简洁，层次清晰，本书的所有程序代码都使用C语言实现，简单易学、易懂。本书比较适合计算机、自动化、电子及硬件等相关专业的院校学生进行学习，同时也可供从事单片机开发的科研设计人员参考使用。

本书配有一张光盘，光盘里包括了书中所有的程序代码，读者可参考借鉴，物超所值。

图书在版编目（CIP）数据

MSP430单片机C语言应用程序设计实例精讲 / 秦龙编著. —北京：电子工业出版社，2006.5
（电子工程应用精讲系列）
ISBN 7-121-02371-7

Ⅰ.M… Ⅱ.秦… Ⅲ.①单片微型计算机，MSP430 ②C语言—程序设计 Ⅳ.①TP368.1 ②TP312

中国版本图书馆CIP数据核字（2006）第018858号

责任编辑：张 毅 葛 娜
印 刷：北京智力达印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编100036
经 销：各地新华书店
开 本：787×1092 1/16 印张：24.75 字数：519千字
印 次：2006年5月第1次印刷
印 数：5000册 定价：42.00元（含光盘1张）

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

丛书说明

工程技术的电子化、集成化和系统化促进了电子工程技术的发展，同时也促进了电子工程技术在社会各行业中的广泛应用，从近年的人才招聘市场来看，电子工程师的人才需求更是一路走高。

电子工程师如此紧俏，除需求不断走高，人才供不应求外，另一重要原因则是电子工程师的门槛相对而言比较高，这个高门槛则来自于工程师的“经验”和“实践”！

因此，为了满足读者学习和工作需要，解决各种工作中的专业问题，我们紧紧围绕“经验”和“实践”，精心策划组织了此套丛书。

1. 丛书范围

现代电子科学技术的一个特点是多学科交叉，因此，工程师应当了解、掌握2门以上的相关学科，知识既精深又广博是优秀的工程师成长为某领域专家的重要标志。本丛书内容涉及软件开发、研发电子以及嵌入式项目开发等，包括单片机、USB 接口、ARM、CPLD/FPGA、DSP、移动通信系统等。

2. 读者对象

本套书面向各领域的初、中级用户。具体为高校计算机、电子信息、通信工程、自动化控制专业在校大学生，以及从事电子开发和应用行业的科研人员。

3. 内容组织形式

本套书紧紧围绕“经验”和“实践”，首先介绍一些相关的基础知识，然后根据不同

的模块或应用领域，分篇安排应用程序实例的精讲。基础知识用来为一些初级读者打下一定的知识功底；基础好一点的读者则可以跳过这一部分，直接进入实例的学习。

4. 实例特色

在应用实例的安排上，着重突出“应用”和“实用”两个基本原则，安排具有代表性、技术领先性，以及应用广泛的典型实例，让读者学习借鉴。这些实例是从作者多年程序开发项目中挑选出的，也是经验的归纳与总结。

在应用实例的讲解上，既介绍了设计原理、基本步骤和流程，也穿插了一些经验、技巧与注意事项。特别在程序设计思路上，在决定项目开发的质量和成功与否的细节上，尽可能地用简洁的语言来清晰阐述大众易于理解的概念和思想；同时，程序代码部分做了很详细的中文注释，有利于读者举一反三，快速应用和提高。

5. 光盘内容

本套书的光盘中包含了丰富的实例原图文件和程序源代码，读者稍加修改便可应用于自己的工作中或者完成自己的课题（毕业设计），物超所值。读者使用之前，最好先将光盘内容全部复制到电脑硬盘中，以便于以后可以直接调用，而不需要反复使用光盘，提高操作速度和学习效率。

6. 学习指南

对于有一定基础的读者，建议直接从实例部分入手，边看边上机练习，这样印象会比较深，效果更好。基础差一点的读者请先详细学习书中基础部分的理论知识，然后再进行应用实例的学习。在学习中，尽量做到反复理解和演练，以达到融会贯通、举一反三的功效；特别希望尽量和自己的工作设计联系起来，以达到“即学即会，学以致用”的最大化境界。

本套书主要偏重于实用性，具有很强的工程实践指导性。期望读者在学习中顺利、如意！

光盘说明

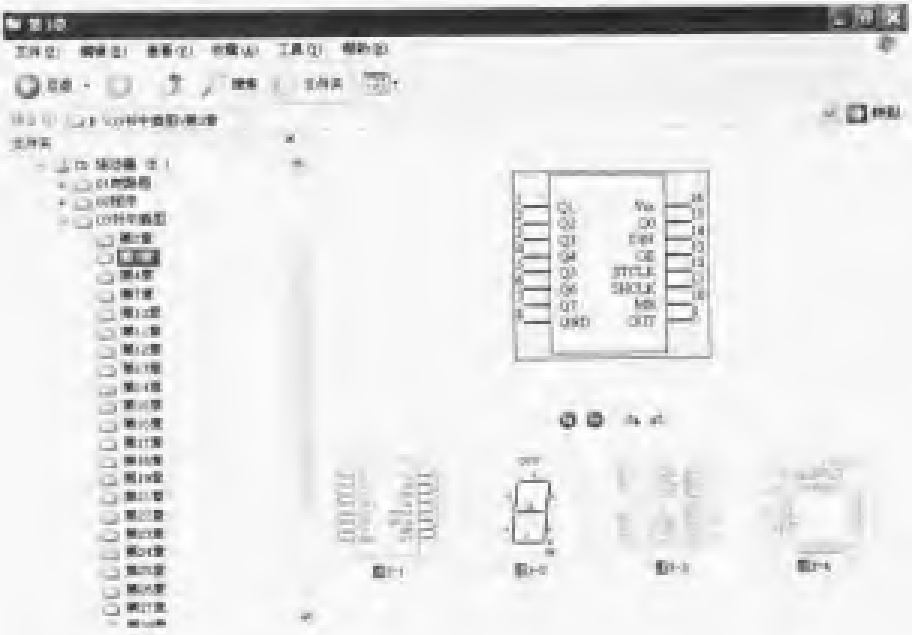
1. 光盘的内容说明

该光盘包括 3 个文件夹：“电路图”、“程序”、“书中插图”。

✓“电路图”文件夹里面的内容为各章的电路图，由 Protel99 软件制成。

✓“程序”文件夹里面的内容为各章的程序代码。

✓“书中插图”文件夹里面的内容为各章的插图，bmp 格式。



2. 光盘的使用说明

光盘里面的程序需要采用 C 语言的编译软件进行打开阅读，也可以使用“UltraEdit”

等软件打开阅读或者编辑。但是这些程序必须使用“Embedded Workbench”集成开发环境进行编译调试。如果需要硬件支持的，则必须有硬件支持，才能进行运行。

光盘里面的电路图是用 Protel99 软件制成的，所以必须使用该软件打开。

3. 系统要求

该光盘需要的硬件系统只需要一般的 PC 机就可以，但考虑到 Word 软件比较耗资源，内存最好在 128MB 以上。操作系统需要是 Windows 98 或者 Windows 98 以上，但不能是 Linux 或者 UNIX 系统。

前　言

MSP430 系列单片机是一种 16 位的单片机。由于它具有集成度高、外围设备丰富、超低功耗等优点，因此在许多领域内除超低功耗外，得到了广泛的应用。特别是它的超低功耗特性是目前其他单片机不可比拟的。另外，由于 MSP430F 系列单片机具有非常强的处理能力，最高可以达到 2MIPS，因此非常适合一些对处理速度要求比较高的嵌入式系统。

MSP430 系列单片机可采用汇编语言和 C 语言进行开发。由于采用 C 语言开发可以大大提高开发效率，缩短开发周期，并且采用 C 语言开发的程序具有非常好的可读性和移植性，因此使用 C 语言开发 MSP430 系列单片机就非常简单；由于适用于 MSP430 系列单片机的 C 语言与标准 C 语言兼容度高，并且 IAR 公司提供的 Embedded Workbench 集成开发环境人机界面友好，能很好地支持 C 语言开发，因此**本书的程序都是采用 C 语言进行开发的**。

本书内容

本书首先介绍了 C 语言的基本知识，使读者能使用 C 语言进行程序设计；然后介绍了 30 个开发例子。全书主要通过实例的形式来介绍 MSP430 单片机在不同领域中的应用，从而使读者掌握 MSP430 单片机开发的流程、方法、技巧及设计思想。第 1 章介绍了 MSP430 开发的 C 语言的基础，对开发 MSP430 单片机有着非常重要的作用。第 2 章至第 31 章重点介绍一些具体的开发实例，包括硬件设计和程序设计。

本书特色

（1）全书实例涉及的应用领域比较广，有一定的代表性。针对每一个例子，都有详细的程序代码，读者可以直接借鉴使用。

（2）本书的例子全部采用C语言实现。对于本书的程序，有的只是实现了一个基本框架或者只介绍了关键代码，读者可以根据自己的情况，举一反三，丰富程序功能，实现自己的更为完整的系统。

（3）本书配套光盘里面包含了本书中用到的所有程序代码，方便读者的学习和使用。通过对本书实例的学习，相信读者能循序渐进地掌握MSP430单片机的开发技术，并且能够做到触类旁通，在自己的开发设计中灵活运用。

本书主要由秦龙编写，另外，参与编写的人还有田莉、钱林杰、张晓平、王渝梅、金成江、李志江、肖毅、刘轶、刘云志、路鸾等，他们在资料收集与整理、硬件设计与程序调试和技术支持方面做了大量的工作，在此一并向他们表示感谢！

由于时间仓促，再加之作者的水平有限，书中难免存在一些不足之处，欢迎广大读者批评和指正。

作　　者

作者简介

本书作者具有多年从事DSP和单片机的开发经验。在DSP方面，先后使用TMS320C54XX系列芯片从事语音信号处理和数字信号处理等相关项目的开发。在单片机方面，先后使用了Cygnal的C8051F0XX系列、Microchip的PIC系列、TI的MSP430系列等多种单片机，在无线通信传输、电子医疗、自动控制等领域进行项目开发，并取得过不错的成绩。另外，本书作者先后在《声学学报》、《计算机工程与设计》、《微计算机信息》、《工业控制计算机》等杂志及“全国数据通信会议”等学术会议上发表过多篇论文。

目　录

第二篇 算法实现

第三篇 存储应用

第四篇 采集与测量

第六篇 控制应用

第 1 章

MSP430 开发的 C 语言基础

MSP430 系列单片机是一种 16 位的单片机。它集成功能丰富，内存也比较大，适合开发比较复杂的系统，C 语言是其开发的首选程序设计语言。采用 C 语言开发主要有以下优点：大大提高软件开发的工作效率；提高程序的可靠性、可读性和可移植性。本章用较短的篇幅来介绍一下 C 语言程序设计的基本概念，同时也介绍一下 MSP430 的 C 语言的扩展特性。

1.1 C 语言基本知识

MSP430 系列支持标准的 C 语言，在标准的 C 语言基础上进行了扩展，因此掌握标准 C 语言对开发 MSP430 系列单片机有着非常重要的作用。下面针对 MSP430 开发介绍一些 C 语言的开发基础。

1.1.1 标志符与关键字

1. 标志符

C 语言中的标志符可以作为变量名、函数名、数组名、类型名及文件名。它可以是一个字符，也可以是多个字符。标志符必须以字母或者下划线开始，后面可以跟字母、数字或者下划线。例如：_Data、nIndex 是正确的形式，而 2Index、n%则是错误的形式。在 C 语言中，标志符要求区分大小写，也就是说大写和小写的标志符被当做不同的标志符。例

如：Index 和 index 是两个不同的标志符，所以在这一点上需要特别引起注意，这就要求在写程序时要有良好的习惯。

2. 关键字

关键字是一种含有特殊意义的标志符。关键字又称保留字。关键字在编译器中已经有了定义，所以不能再进行重新定义，需要加以保留。用户在定义自己的变量或者函数的时候千万不要使用关键字，否则就会出现一些错误。在 C 语言的编译系统中主要有以下几种类型的关键字。

- 数据类型关键字：auto，char，const，double，enum，extern，float，int，long，register，sizeof，short，static，struct，typedef，union，unsigned，void，volitile 等。该类型关键字主要用于定义一些变量或者函数。例如：int nIndex;该语句就是定义一个 int 类型的数据。这里不同的关键字有不同含义，需要在使用的时候加以区分。
- 程序控制关键字：break，case，continue，default，do，else，for，goto，if，return，switch，while 等。该类型的关键字主要用于程序的控制。例如：

```
int n = 9;
int m;
if(n < 10)
{
     m=10;
}
else
{
     m = 11;
}
```

该代码运行后的结果就为：m=10。

- 预处理功能关键字：define，endif，ifdef，ifndef，include，undef 等。该类型的关键字主要用于进行预处理。例如：#include <msp430x14x.h>表示包括 msp430x14x.h 头文件。

1.1.2 数据的基本类型

标准 C 语言中主要有整数型、实型和字符型。下面就这几种类型进行具体的介绍。

1. 整型数据

整型数据里面主要包括 int、short、long 等。不同类型的整数类型的变量具有不同的整

数范围。MSP430 的 C 语言除了支持标准 C 语言的整数类型外，还支持其他的几种整数类型。表 1-1 给出了 MSP430 的 C 语言支持的几种整数类型。

表 1-1　MSP430 的 C 语言支持的整数类型

整数类型	字　节	数值范围	说　明
sfrb	1		定义特殊功能寄存器
sfrw	2		定义特殊功能寄存器
unsigned char	1	0~255	无符号字符
char（默认）	1	0~255	等效于 unsigned char
signed char	1	−128~127	有符号字符
char（-c 选项）	1	−128~127	等效于 signed char
short	2	−32768~32767	短整数
int	2	−32768~32767	整数
unsigned short	2	0~65535	无符号短整数
unsigned int	2	0~65535	无符号整数
long	4	−2147483648~2147483647	长整数
unsigned long	4	0~4294967295	无符号长整数
float	4	±1.18E−38~±3.39E+38	浮点数
double	4	±1.18E−38~+3.39E+38	双精度浮点数
long double	4	±1.18E−38~±3.39E+38	长双精度浮点数
pointer	2		指针
enum	1~4		枚举

整型变量的定义如下面的例子所示。

```
void main()
{
    int sum,n,m;
    n = 12;
    m = 0x15;
    sum = n + m;
}
```

上面的例子给出了整型变量的定义方法。另外，在上面的例子中，整数有常用的两种形式：十进制和十六进制。在数值的前面加上 0x 就表示十六进制数了。

2. 实型数据

实型数据就是浮点数据。它可以含有小数点，但是它表示的数是有精度的。实数有两种具体的表现形式：十进制小数点形式和指数形式。小数点形式为 12.1212。指数形式包括整数部分、尾数部分和指数部分，具体形式为 1.21E+1。

实型变量主要有 float，double 和 long double。实型变量的定义方法很简单。例如：

```
float a;
double b;
```

上面的例子定义了类型为 float 的变量 a 和类型为 double 的变量 b。

3. 字符型数据

字符型数据主要处理字符相关的内容，比如处理英文字母或者汉语句子。一般来说，会将多个字符型变量组成一个字符串来使用。在 C 语言中，字符是按照所对应的 ASCII 码的值来对应的，一个字符占一个字节，例如：英文字母“A”的 ASCII 码值是 65。字符的 ASCII 码值可以在 ASCII 码表里面查找到。在这里需要强调一下整数和字符常量在表现形式上是有区别的，比如‘5’表示的是字符，而 5 表示的是整数。

字符变量主要包括 char。字符变量的定义方法非常简单。例如：

```
char chrTemp;
chrTemp = 'A';
```

上面的例子定义了字符变量 chrTemp，并给它赋值为‘A’。这里需要强调的是：字符是以单引号表示的。

在使用字符变量时，需要了解转义字符。转义字符是一种特殊的字符，通常使用转义符表示 ASCII 码字符中不可打印的控制字符和特殊功能字符。转义字符是使用反斜杠（\）后面跟一个字符来表示。例如：“\n”表示换行，“\r”表示回车，“\‘”表示单引号。

4. 各种数据之间的转换

在某些应用的场合，需要进行数据类型的转换，比如把字符类型的变量转换成整数类型，把 int 类型的数据转换成 long 类型的数据。进行数据类型转换的方法就是采用强制转换类型。下面的例子给出了数据之间的转换。

```
void main()
{
    char chrTemp;
    int n;
    chrTemp = 'A';
    n = (int)(chrTemp);
}
```

上面的例子是从字符类型转换成 int 类型，n 的最终值为 65。需要注意的是：并不是所有类型之间都可以进行类型转换；从占字节多的类型转换成占字节少的类型时，有时候可能会造成数据的丢失。基于以上原因，在使用类型转换的时候需要小心使用。

1.1.3　C 语言的运算符

C 语言的内部运算比较丰富，比如可以进行加、减、乘、除等运算。C 语言的运算主要包括算术运算、关系运算、逻辑运算、赋值运算和位运算。下面就各种具体的运算进行简要的介绍。

1．算术运算

算术运算主要是指加、减、乘、除等运算。表 1-2 给出了算术运算符。

表 1-2　算术运算符

运 算 符	含　义	说　明
++	单目加	只有一个操作数
—	单目减	只有一个操作数
+	加	需要两个操作数
-	减	需要两个操作数
*	乘	需要两个操作数
/	除	当两个操作数是整数的时候，结果为整数
%	模运算（求余）	操作数必须是整数

下面给出算术运算的例子。

```
void main()
{
    int n,m;
    int y;
    n = 5;
    m = 2;
    y = m + n;          //y 的值为 7
    y = m * n;          // y 的值为 10
    y = m / n;          // y 的值为 2
    y = m % n;          // y 的值为 1
    n++;                // 执行这句代码后，n 的值为 6
}
```

上面的例子比较简单，复杂的运算也是同样如此。

2．关系运算

关系运算符主要是对操作数进行某种条件的判断，结果只有 true 和 false 两种结果。表 1-3 给出了关系运算符。

表 1-3 关系运算符

关系运算符	含 义	应用例子（设 a = 3,b = 4）
>	大于	a > b 结果：false
>=	大于等于	a >= b 结果：false
==	等于	a == b 结果：false
<	小于	a < b 结果：true
<=	小于等于	a <= b 结果：true
!=	不等于	a != b 结果：true

由表 1-3 可以看出：关系运算主要就是处理操作数之间的关系。

3. 逻辑运算

逻辑运算和关系运算比较相似，也是处理操作数之间的关系，结果只有 true 和 false 两种结果。表 1-4 给出了逻辑运算符。

表 1-4 逻辑运算符

逻辑运算符	含 义	应用例子（设 a = true,b = false）
&&	与运算	a && b 结果：false
||	或运算	a || b 结果：true
!	非运算	!a 结果：false

表 1-4 给出了三种逻辑运算。针对具体的逻辑运算可以参看逻辑运算的真值表。

4. 赋值运算

通常把“=”称为赋值运算符。该运算符是一个二元运算符，需要两个操作数，左边的操作数是变量或者数组，右边的是表达式。例如：

```
int n,m;
m = 5;              //赋值运算
n = m * m;          //赋值运算
```

另外，“=”还可以和其他的运算符结合起来使用。比如+=、-=、*=、/=、%=等，它们的意义分别是：

x += a; 等价于 x = x + a;

x -= a; 等价于 x = x - a;

x *= a; 等价于 x = x * a;

x /= a; 等价于 x = x / a;

x %= a; 等价于 x = x % a;

当然“=”还可以和“>>”等运算符结合起来使用，使用的含义和上面的含义相同，所以在此不再赘述。

5. 位运算

位运算在单片机的开发中非常重要，比如设置某个管脚的输出电平为高电平的操作就是通过位运算来实现的。位运算主要包括“与”（&）、“或”（|）、“反”（~）、“左移”（<<）和“右移”（>>）等运算。例如：

```
void main()
{
    int m,n,k,result;
    m = 10;
    n = 13;
    k = 0x0a;
    result = m & n;      // result 的值为 8
    result = m | n;      // result 的值为 15
    result = ~(k);       // result 的值为 0xfa
    result = m << 2;     // result 的值为 40
    result = m >> 2;     // result 的值为 2
}
```

通过以上的例子就很容易理解位运算。

6. 运算的优先级

通过前面的介绍，读者应该对 C 语言的几种运算有了基本的了解。在实际的应用中，一个计算可能是上面的几种运算的组合，这样在进行运算的时候，执行的顺序就非常重要，这时就需要了解运算的优先级。表 1-5 给出了运算的优先级。

表 1-5　运算的优先级

优 先 级	符　　号	操作数个数
1	! ~ ++ —等	单操作数
2	* / %	双操作数
3	+ -	双操作数
4	<< >>	双操作数
5	< <= > >= == !=	双操作数
6	&	双操作数
7	|	双操作数
8	&&	双操作数
9	||	双操作数
10	= += -= *= /= 等	双操作数

表 1-5 给出了运算的优先级，如果在有括号运算的时候，应该先运算括号里面的，然后再运算括号外面的表达式，在括号里面的表达式的运算优先级应该按照表 1-5 所列出的优先级来进行运算。

1.1.4 程序设计的基本结构

计算机的程序是由许多条语句按照顺序执行的。计算机的程序有 3 种结构：顺序结构、选择结构和循环结构。下面就每个结构进行详细的介绍。

1. 顺序结构

顺序结构就是程序从前往后依次执行语句的结构。从整体来说，顺序结构是程序的基本结构，只不过在某个地方需要加入选择结构或者循环结构，执行完选择结构或者循环结构又按照顺序执行。下面举一个顺序结构的简单例子。

```
void main()
{
    int n,m;
    n = 5;
    m = 13;
    m += n;
}
```

通过上面的例子可以看出，顺序结构是非常简单的一种结构，只需要简单地按照顺序执行就可以了。

2. 选择结构

选择结构就是程序执行过程中根据一定的条件，程序的任务有多种不同的选择，也就是程序的执行顺序要根据具体的条件来选择。这样选择结构就包括多个分支，根据不同的条件，一般只有一个分支被执行，其他分支则不被执行。

选择结构里面主要用到的是 if 语句和 switch 语句。下面就这两个语句分别进行介绍。

（1）if 语句

if 语句主要有三种基本形式。

```
If（条件表达式）
{
    语句;
}
```

这种形式的 if 语句是在条件表达式的值为 true 的时候执行{}里面的语句，否则不执行

里面的语句。

```
if（条件表达式）
{
    语句 1;
}
else
{
    语句 2;
}
```

这种形式的 if 语句是在条件表达式的值为 true 的时候执行语句 1，否则执行语句 2.

```
If（条件表达式 1）
{
    语句 1;
}
else if（条件表达式 2）
{
    语句 2;
}
else
{
    语句 3;
}
```

这种形式的 if 语句是在条件表达式 1 的值为 true 的时候执行语句 1，否则判断条件表达式 2 的值是否为 true，当为 true 时，执行语句 2；如果条件表达式 1 和条件表达式 2 都不是 true 的话，则执行语句 3。

下面举一个简单的例子。

```
void main()
{
    int n,m,k;
    int sum;
    n = 2;
    m = 3;
    k = 4;
    sum = 0;
    if(m > n) sum += m;     //sum 的值是 3
    if(n > k) sum += n;
    else sum += k;          //sum 的值是 7
    if(m > k) sum += m;
    else if(m > n) sum += m;
    else sum += k;          //sum 的值是 10
}
```

上面的例子给出了 if 语句的使用，在实际程序中，只要适当使用 if 语句就可以设计出满足程序需要的分支结构了。

（2）switch 语句

上面的 if 语句比较适合分支比较少的程序，如果要实现很多个分支里面选一的话，就可以采用 C 语言的另外一种分支语句：switch 语句。switch 语句的语法如下：

```
switch（条件表达式）
{
    case 常量表达式 1：
        语句 1；
        break；
    case 常量表达式 2：
        语句 2；
        break；
        ……
    case 常量表达式 n：
        语句 n；
        break；
    default：语句 n+1
}
```

通过 switch 语句的语法可以看出，当条件满足某个分支的时候，就执行相应的语句；如果都不满足的话，则执行 default 里面的语句。switch 语句执行完某一个分支的时候一定要加 break，否则还会执行下面的分支。下面举例说明 switch 语句的使用。

```
void main()
{
    int n,m,sum;
    n = 10;
    m = 13;
    sum = m - n;
    switch(sum)
    {
        case 1:
            sum += m;
            break;
        case 2:
            sum += n;
            break;
        case 3:
            sum += (m + n);
            break;
        default:break;
    }
}
```

上面的例子的运行结果是：sum = 26。在实际的应用中，对于分支很多的情况下，选用 switch 语句比较合适。

3．循环结构

当在程序某处需要循环执行某部分代码时，就需要用到循环结构。循环结构主要有 for、while 和 do...while 三种语句，另外，还有一些与循环结构相关的语句，比如 continue 语句和 break 语句。下面就这几种语句分别进行介绍。

（1）for 循环

for 循环主要用于循环次数确定的情况下，它的语法格式为：

```
for（表达式 1；表达式 2；表达式 3）
{
    语句 1；
    ……
    语句 n；
}
```

其中表达式 1 是循环开始的初始条件，表达式 2 是判断表达式，表达式 3 是循环计数器的改变表达式。当表达式 2 的值为 true 时，执行 for 循环里面的语句；如果是 false，就不再执行 for 循环里面的语句。下面举一个具体的例子来说明 for 循环的使用。

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

上面的例子就是计算从 0~99 的和。

for 循环的表达式 1 和表达式 3 可以省略。比如上面的程序可以写成下面的形式。

```
void main()
{
    int i,Sum;
    sum = 0;
    i = 0;
    for(; i < 100; )
    {
        sum += i;
        i++;
```

```
    }
}
```

另外，表达式 3 的值可以改变循环的步长。比如下面的程序是计算从 0~99 之间的偶数的和。

```
void main()
{
    int i,sum;
    sum = 0;
    for(i = 0;i < 100;i+=2)
    {
        sum += i;
    }
}
```

（2）while 循环

while 循环主要是执行循环次数不确定的循环。while 循环的语法为：

```
while（条件表达式）
{
    语句 1;
    ……
    语句 n;
}
```

while 循环的执行是先判断条件表达式的值，如果是 true 就执行里面的语句，如果是 false 就不执行里面的语句。循环结束的条件也就是：条件表达式的值是 false。下面给出具体的例子。

```
void main()
{
    int i,sum;
    sum = 0;
    i = 0;
    while(i < 100)
    {
        sum += i;
        i++;
    }
}
```

上面的例子也是计算从 0~99 的和。在执行 while 循环时，里面必须有语句改变条件表达式的值，否则循环就会进入死循环。

（3）do...while 循环

do...while 循环也是用于循环次数不确定的场合。do...while 循环是先执行循环体，再

判断表达式的值是否是 true。表达式的值是 true 就继续执行循环体里面的语句，如果是 false 就跳出循环体。它的语法格式如下：

```
do
{
    语句 1;
    ……
    语句 n;
} while(条件表达式)
```

下面举一个具体的例子来说明 do...while 循环。

```
void main()
{
    int i,sum;
    sum = 0;
    i = 0;
    do
    {
        sum += i;
        i++;
    } while(i < 100)
}
```

上面的例子也是计算从 0~99 的和。在执行 do...while 循环时，里面必须有语句改变条件表达式的值，否则循环就会进入死循环。与 while 循环不同的是，do...while 循环不管条件表达式是否为 true，都会执行一次循环体。

（4）continue 语句

continue 语句是用于结束本次循环执行，开始新一轮的循环执行。在执行到 continue 时，位于 continue 后面的循环体里面的语句不再执行。下面举例说明。

```
void main()
{
    int i,sum;
    int a[10] = {0,1,2,3,4,5,6,7,8,-9};
    for(i = 0;i < 10;i++)
    {
        if(a[i] < 0) continue;
        sum += a[i];
    }
}
```

上面的例子表明，只要 a[i]的值小于 0 就不进行求和运算，并开始下一次的循环，所以 sum 最后的结果为 23。

（5）break 语句

break 语句是跳出循环体，不再继续执行循环体。下面举例说明。

```
void main()
{
    int i,sum;
    sum = 0;
    for(i = 0;i < 100;i++)
    {
        sum += i;
        if(sum > 4000) break;
    }
}
```

在上面的例子里，当 sum 的值大于 4000 时，程序就跳出循环体，不再执行循环体了。

1.1.5 函数

在 C 语言中，函数是程序的基本组成单位。函数不仅可以实现程序的模块化，使程序设计得简洁和直观，提高了程序的易读性和可维护性，而且还可以把程序中经常用到的一些计算或者操作做成通用的函数，以便随时调用。

函数是 C 语言的基本构件，一个 C 语言程序可以由一个主函数和若干个函数组成。由主函数调用其他函数，其他函数间也可以互相调用，同一个函数也可以被多次调用。

1．函数定义

函数定义的语法如下：

函数类型函数名（参数表）

```
{
    语句 1;
    ……
    语句 n;
    返回;
}
```

函数类型确定了函数返回值的类型。函数的类型可以是任何一种有效的类型，比如整数类型，也可以是用户自定义的类型。

函数的参数表确定了函数的输入参数和输出参数，多个参数之间以逗号分开。

函数体主要由一系列的语句组成，语句的组成结构可以是顺序结构，也可以是分支结构或者循环结构。在函数体的最后，需要返回函数的值，如果函数定义成 void，可以不用

返回值，其他类型必须返回函数值，并且返回函数的值的类型必须和函数定义时函数的类型一致。下面举例子来说明函数的定义。

```
void memset(int a[10],n)
{
    int i;
    for(i = 0;i < 10;i++)
    {
        a[i] = n;
    }
    return;
}
```

在上面的例子中，return 可以省略，因为这个函数的返回类型是 void。

```
int sum(int n,int m)
{
    int sum;
    sum = n + m;
    return sum;
}
```

在这个例子里面，return 不能省略，需要将计算的结果返回。

2. 局部变量与全局变量

在引入了函数定义后，需要知道变量的作用域，就是变量的使用范围。根据变量的作用域可以将变量分为全局变量和局部变量。局部变量就是在函数内部定义的变量，局部变量只被函数内部访问。全局变量与局部变量不同，它一般定义在程序的顶端，它能贯穿整个程序，能被任何一个模块使用。在程序的设计中，如果全局变量和某一个局部变量的名字相同，那么在局部变量使用的函数内部，当使用同一名字的这两个变量时，实际使用的是局部变量，这一点需要引起注意。正因为如此，所以在定义变量的时候绝对不能重名。并且在使用变量的时候应该合理使用局部变量和全局变量。

结构化的程序需要程序代码和数据分离，C 语言是通过局部变量和全局变量来实现这一分离的。如果大量使用全局变量就破坏了结构化程序设计的要求。下面举例说明局部变量和全局变量的使用。

```
int a = 3;              //全局变量
int c = 6;              //全局变量
int Mult(int x,y);
void main()
{
    int sum,a;          //局部变量
    sum = Mult(a,c);
```

```
}
int Mult(int x,int y)
{
    int res;
    res = x * y;
    return res;
}
```

上面的例子具体演示了局部变量和全局变量的使用。上面程序的运行结果应该是 48。通过例子也能明确变量的作用域。

3. 形式参数与实际参数

函数定义时的参数成为形式参数，简称形参。它们同函数内部的局部变量作用相同。形参的定义在函数名后的括号内。在进行函数调用时，传入的参数成为实际参数，简称实参。实参和形参的顺序必须一致，否则就会出现错误，这一点上需要引起注意。下面举例子加以说明。

```
int GetMax(int x,int y);
void main()
{
    int m,n,k;
    m = 9;
    n = 10;
    k = GetMax(m,n);  //m,n 为实参
}
int GetMax(int x,int y)   //x,y 为形参
{
    if(x >= y) return x;
    else return y;
}
```

程序中 k 的值为 10。

4. 函数调用方式

在函数体实现完成后，需要具体调用函数才能执行函数，也才能利用函数实现的功能。在 C 语言中，函数有标准的库函数，也有用户自定义的函数。对于库函数而言，需要包括具体的头文件，比如#include<stdio.h>。对于用户定义的函数，如果函数不在调用它的函数的那个 C 文件里面，则需要包括头文件，比如#include "user.h"。这里需要注意的是头文件不能用“<>”包括起来，只能用双引号包括起来。如果函数和调用它的函数在同一个文件里，则只需要在函数前面声明一下就可以了。

关于函数的调用主要有以下形式。

（1）函数作为语句

就是简单地把函数作为一条语句来执行。比如：

```
void memset(int a[10],int n);
int main()
{
    int a[10];
    int n;
    n = 10;
    memset(a,n);   //函数作为执行语句
}
void memset(int a[10],int n)
{
    int i;
    for(i = 0;i < 10;i++)
    {
        a[i] = n;
    }
    return;
}
```

最终结果是 a[0], …, a[9]的值均为 0。

（2）函数作为表达式

这种形式就是将函数作为表达式里面的一部分，下面举例说明。

```
int GetMax(int x,int y);
void main()
{
    int m,n,k;
    m = 9;
    n = 10;
    k = 20;
    k = k - GetMax(m,n);  //函数作为表达式
}
int GetMax(int x,int y)
{
    if(x >= y) return x;
    else return y;
}
```

最终结果是 30。

（3）函数作为参数

这种形式就是将函数作为一个函数的实参进行传递。

```
int GetMax(int x,int y);
void main()
```

```
{
    int m,n,k;
    m = 9;
    n = 10;
    k = 20;
    k = GetMax(m,GetMax(n,k));  //函数作为参数
}
int GetMax(int x,int y)
{
    if(x >= y) return x;
    else return y;
}
```

最终结果是 20。

5. 函数嵌套调用

在 C 语言中，所有的函数都可以互相调用，如果函数调用自己的函数，就是所说的递归。函数也可以调用其他函数，函数之间可以实现多次调用。下面举例说明。

```
int max2(int x,int y);
int max3(int x,int y,int z);
void main()
{
    int n,m,k,res;
    n = 10;
    m = 20
    k = 30;
    res = max3(n,m,k);
}
int max2(int x,int y)
{
    if (x >= y) return x;
    else return y;
}
int max3(int x,int y,int z)
{
    int res;
    res = max2(x,y);
    return = max2(res,z);
}
```

上面的例子给出了函数的嵌套调用。主要强调的是如果实现递归的时候，一定要注意，因为很容易引起死循环。

1.1.6　数组

数组是一个由同种类型变量组成的集合，引用这些变量时可以使用同一名字。数组由连续的存储区域组成，最低地址对应于数组的第一个元素，最高地址对应于最后一个元素。数组可以是一维的，也可以是多维的。

1．一维数组

一维数组的定义为：

数组类型 数组名[数组元素个数]；

例如：

```
int a[10];    //定义一个整数类型的数组 a，数组元素的个数为 10
```

一维数组可以在定义的时候初始化值。例如：

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
```

数组的访问通过下标来实现，数组元素的下标从 0 开始，而不是从 1 开始。比如上面的数组中的第 3 个元素就是 a[2]。在赋值的时候可以全部赋值，也可以部分赋值。如果全部赋值的话，可以写成下面的形式：

```
int a[] = {1,2,3,4,5,6,7,8,9};
```

如果是部分赋值的话，没有被赋值的元素的值默认为 0。例如：

```
int a[10] = {1,2,3,4,5};
```

在上面的数组中，只有前 5 个元素赋了值，而后面的几个元素没有被赋值，则后面的几个元素的值为 0。

数组元素的访问和一般变量的使用差不多。下面举例子说明。

```
void main()
{
    int a[10];
    int n;
    for(n = 0;n < 10;n++)
    {
        a[n] = n;
    }
}
```

上面的例子说明数组的元素 a[n]和一般变量的使用基本相同。

2. 多维数组

C 语言中可以定义多维数组，最简单的多维数组就是二维数组。二维数组的定义方法为：

```
数组类型 数组名[行数][列数];
```

这样二维数组元素的个数为：行数乘以列数。

例如：

```
int a[3][5];   //定义一个 3 行 5 列的数组，数组元素的个数为 15
```

多维数组可以在初始化的时候赋值。例如：

```
int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}}
```

通过上面的例子可以看出：二维数组可以看成多个一维数组的集合。

另外，二维数组也可以部分赋值。例如：

```
int a[3][4] = {{1,2,3},{5,6,7,8},{9,10}}
```

对于二维数组的每个元素而言，也可以看成一般的变量。下面举例子说明。

```
void main()
{
    int a[10][5];
    int n,m;
    for(n = 0;n < 10;n++)
    {
        for(m = 0;m < 5;m++)
        {
            a[n][m] = m - n;
        }
    }
}
```

上面的例子说明数组的元素 a[n][m]和一般变量的使用基本相同。

1.1.7 指针

在 C 语言中，指针是非常重要的概念。运用指针可以增加程序的灵活性，提高程序的运行效率。但是不合理地使用指针，也会降低程序的可读性。

C 语言中对变量存取有如下两种方式。

- 按变量名存取。实际上变量名代表变量存放的首地址，这种按变量地址存取变量值的方式称为“直接访问”。

- 定义另外一种类型的变量专门存放其他变量在内存中所分配存储单元的首地址，称为指针变量。存取变量值时，分两个步骤进行：首先，根据指针在内存中的首地址，读取其中存放的数据，也就是变量所占用内存单元的首地址；然后根据读取的地址存取变量的值。这种存取变量的值的方式称为“间接访问”。

所谓指针就是某个对象所占用存储单元的首地址。专门用来存放某种类型变量的首地址（指针值）的变量称为该类型的指针变量。使用指针访问能使目标程序占用内存少，运行速度快。

1. 指针的定义

指针的定义如下：

```
类型 *指针变量名;
```

其中的“*”表示指针类型的变量，类型表示指针所指向的变量的类型。

例如：

```
int *pint;     //定义一个整数类型的指针变量
```

2. 指针变量的引用

在利用指针变量进行间接访问之前，必须使它指向一个确定的变量。指针变量只能存放地址，不能将一个非地址量赋给指针变量。

C 语言中提供了与指针有关的两个运算符，即指针运算符“*”和取地址运算符“&”。“*”运算符是通过指针变量间接访问它所指向的变量，来存取数据。“&”运算符是取得所占用的存储单元的首地址。下面举例子说明。

```
void main()
{
    int n,m;
    int *p;       //声明一个 int 类型的指针变量
    n = 10;
    p = &n;       //p 指向 n
    m = *p;       //将 n 的值赋给 m，所以 m 的值为 10
}
```

上面的例子演示了指针的两种运算符。

3. 数组的指针

一个数组包含若干个元素，每个数组元素都在内存中占用存储单元，并且它们都有相

应的地址。指针变量可以指向数组和数组中任一个元素。

所谓数组指针就是数组的开始地址，数组元素的指针就是数组元素的地址。引用数组元素可以采用下标来访问，也可以采用指针来访问，即通过数组元素的指针指向所需的元素。一个数组指针的具体定义为：首先定义一个指针，然后用所定义好的指针指向数组的首地址。例如：

```
int a[10];
int *p;
p = a;
p = &a[0];
```

上面的“p = a;”和“p = &a[0];”是等效的，说明数组的指针就是它的首地址。

既然指针可以指向数组，则对数组元素的访问就可以通过指针的方式来访问。例如：在上面的例子中，*(p+1)就是 a[1]。下面举例来说明通过指针来访问数组的元素。

```
void main()
{
    int n;
    int count;
    char chrBuf[10] = {'a','b','c','d','1','2','3','4','q','w'};
    char *pChr;
    pChr = chrBuf;   //指针指向首地址
    count = 0;
    for(n = 0;n <10;n++)
    {
        if(*pChr >= 'a' && *pChr <= 'z')
        {
            count += 1;
        }
        pChr++;  //指针移向下一个单元
    }
}
```

上面的例子演示了指针访问数组的方法。

在实际的应用中还有指针数组，只需要将指针和数组的概念结合起来就能理解，这里就不进行详细的介绍了。

1.1.8 结构

在前面讲的数组虽然可以是多个元素的集合，但是它的元素是同一类型的，因此在使用的时候就缺少一定的灵活性，所以 C 语言里面有结构类型的变量。结构类型可以是多种类型的元素的集合。下面就结构的一些基本概念进行介绍。

1. 结构的定义

结构的定义方法如下：

```
struct 结构名
{
    成员 1;
    ……
    成员 n;
};
```

在结构里面的成员可以是不同类型的变量，下面举例说明结构的定义。

```
struct student
{
    long int id;
    char name[10];
    char sex;
    char address[30];
    int age;
};
```

从上面的例子可以看出，结构里面的成员变量可以是任何类型的变量。另外，结构里面也可以含有结构类型的成员变量，下面举例说明。

```
struct birthday
{
    int year;
    int month;
    int day;
};
struct student
{
    long int id;
    char name[10];
    char sex;
    char address[30];
    struct birthday m_birthday;
};
```

2. 结构类型变量的定义

结构类型变量的定义与其他类型变量的定义是一样的。但是，由于结构类型需要针对具体的问题，事先定义，所以结构类型变量的定义形式就增加了灵活性。它主要有以下 3 种形式。

（1）先定义结构再定义变量

这种形式的具体定义如下：

```
struct student
{
    long int id;
    char name[10];
    char sex;
    char address[30];
    int age;
};
struct student stu;    //定义 stu 为 student 类型的变量
```

（2）定义结构的同时定义变量

这种形式的具体定义如下：

```
struct student
{
    long int id;
    char name[10];
    char sex;
    char address[30];
    int age;
} stu ;
```

（3）直接定义结构型变量

这种形式的具体定义如下：

```
struct
{
    long int id;
    char name[10];
    char sex;
    char address[30];
    int age;
} stu ;
```

这种形式可以没有结构名，但不能在其他地方定义同一结构类型的变量。

3. 结构类型变量的初始化

结构类型变量可以在定义时初始化。下面举例说明。

```
struct student
{
    long int id;
    char name[10];
```

```
    char sex;
    int age;
};
student stu={10001, "Latitude" ,'M',29};     //初始化时赋值
```

4. 结构类型变量的引用

结构类型变量的引用格式为：

结构类型变量名.成员名

例如：

```
stu.sex
```

对结构里面的成员变量的使用和一般变量的使用一样。例如：对上面定义的结构的成员变量赋值。

```
stu.id = 10001;
stu.name = "question";
stu.sex = 'm';
stu.age = 29;
```

1.1.9　预处理功能

预处理功能包括宏定义、文件包含和条件编译 3 个主要部分。这些功能是通过相应的宏定义命令、文件包含命令和条件编译命令来实现的。这些命令不同于 C 语言语句，具有以下特点：

- 多数预处理命令习惯上放在文件开头，但是根据需要也可以放到文件的其他地方。
- 预处理命令在编译前实现，编译是对预处理的结果进行的。
- 预处理命令以“#”开头，后面不加“;”，它与语句不同。
- 预处理命令只是一种简单的替代功能，不进行语法检查。
- 宏定义可以嵌套，即在进行宏定义时，可以应用已定义的命令。

下面就宏定义、文件包含和条件编译 3 个部分分别进行介绍。

1. 宏定义

宏定义有两种形式：简单宏定义和带参数宏定义。

(1) 简单宏定义

简单宏定义的格式如下：

#define 标志符 常量表达式

其中的 define 是关键字，它表示该命令为宏；标志符是宏符号名，它的含义是后面的常量表达式。标志符最好大写，宏定义行不需要加分号。宏定义的具体例子如下：

```
#define PI 3.14
```

上面的就是宏的具体定义。宏的定义可以嵌套，例如：

```
#define SIZE 10
#define MAXSIZE SIZE * 10
```

上面的例子说明了宏的嵌套定义。

宏的定义在整个文件里面都有效，因此不能对同一个宏进行重复定义。如果需要对宏进行重新定义时，需要对以前的宏终止，可以采用#undef 命令来实现。下面举例说明。

```
#define SIZE 10
......
#undef SIZE
#define SIZE 12
```

上面的例子通过#undef 命令来终止宏，并对宏进行重新定义。通过这个例子也可以看出，如果需要对宏重新定义，就必须在重新定义前使用#undef 命令。

（2）带参数宏定义

上面介绍了简单的宏定义，宏在定义的时候也可以带参数，带参数的宏定义如下：

```
#define 宏符号名（参数表）宏体
```

例如：

```
#define PI 3.14
#define S(r) PI * (r) * (r)
```

上面的例子给出了带参数的宏的具体定义。在定义的时候需要将表达式里面的参数用括号括起来，以免在调用的时候出错。下面给出带参数宏的调用的例子。

```
#define PI 3.14
#define S(r) PI * (r) * (r)
void main()
{
    int r;
    int area;
    r = 3;
    area = S(r);     //调用宏
}
```

通过上面的例子可以看出，带参数宏的调用和函数的调用很相似。

2. 文件包含

文件包含是在一个程序文件里面可以包含其他文件的内容，这样就可以访问其他文件里面的函数。在C语言编程中，可能会有很多个文件，文件的一般组织形式是头文件（.h文件）声明函数原形，用函数实现文件（.c文件）来实现具体的函数，这样在需要引用某个文件里面的函数的时候，就需要包括该函数实现的头文件。文件包含的格式如下：

```
#include <标准库头文件>或者
#include "用户定义头文件"
```

下面举例说明头文件的包含

Basic_Op.h 文件

```
void memset(int a[],int len,int value);
```

Basic_Op.c 文件

```
#include "Basic_Op.h"
void memset(int a[],int len,int value)
{
    int n;
    for(n = 0;n < len;n++)
    {
        a[n] = value;
    }
}
```

main.c 文件

```
#include <stdio.h>
#include <stdlib.h>
#include "Basic_Op.h"
void main()
{
    int a[10];
    int n;
    memset(a,10,10);
    for(n = 0;n < 10;n++)
    {
        printf("%d\n",a[n]);
    }
}
```

上面的例子演示了文件的包含。

3. 条件编译

对程序代码的各个部分有选择地进行编译称为条件编译。条件编译可以由常量表达式确定，也可以由标志符确定，下面给出两种具体的定义格式。

```
#ifdef 常量表达式 1
程序段 1
#elif 常量表达式 2
程序段 2
#else
程序段 3
#endif
```

在上面的定义中，如果常量表达式 1 成立，则编译程序段 1；如果常量表达式 1 不成立，而常量表达式 2 成立，则编译程序段 2；如果常量表达式 1 和常量表达式 2 都不成立，则编译程序段 3。

条件编译还有另外一种定义，格式如下：

```
#ifdef 标志符
程序段 1
#else
程序段 2
#endif
```

在上面的定义中，如果标志符被定义，则编译程序段 1；如果标志符没有被定义，则编译程序段 2。

通过以上的介绍，有了 C 语言的基础知识，下面在 C 语言的基础上介绍 MSP430 的 C 语言的扩展特性。

1.2 MSP430 的 C 语言扩展特性

MSP430 系列单片机问世不久，就有很多家公司为它实现了 C 语言程序设计的编译器和调试工具。MSP430 的 C 语言在兼容标准 C 语言的基础上还增加了一些其他特性。与一般 C 语言相比，MSP430 的 C 语言主要表现在反映 MSP430 系列的硬件特性和适应嵌入式系统的软件特点方面。下面就具体的扩展特性进行详细的介绍。

1.2.1 MSP430 的 C 语言的扩展概述

MSP430 的 C 语言在标准 C 语言的基础上主要在关键字、#pragma 编译命令、预定义

符号、本征函数及其他扩展特性等方面进行了扩展。下面就各个部分加以简要说明。

1. 扩展关键字

在默认情况下，MSP430 的 C 语言编译器遵守标准 C 语言规范，MSP430 的 C 语言扩展关键字都不能使用。但是编译器选项“-e”能使扩展关键字可以使用。同时，它们也就不能用作变量名了。扩展关键字有以下几类：

（1）与 I/O 访问有关的关键字：sfrb、sfrw。

（2）非易失 RAM 关键字：no_init。

（3）函数类型关键字：interrupt、monitor。

2. #pragma 编译命令

#pragma 编译命令控制编译器的存储器分配，控制是否允许用扩展关键字，以及是否输出警告消息。它提供符合标准语法的扩展特性。

#pragma 编译命令是否可用与“-e”选项无关。

#pragma 编译命令主要分为以下几类：

（1）位域取向

#pragma bitfields = default

#pragma bitfields = reversed

（2）代码段

#pragma codeseg（段名）

（3）扩展控制

#pragma language = default

#pragma language = extended

（4）函数属性

#pragma function = default

#pragma function = interrupt

#pragma function = monitor

（5）存储器用法

#pragma memory = default

#pragma memory = constseg（段名）

#pragma memory = dataseg（段名）

#pragma memory = no_init

（6）警告消息控制

#pragma warnings = default

#pragma warnings = off

#pragma warnings = on

3. 预定义符号

以下的预定义符号允许检查编译时的环境，注意它们都以双下划线字符开头。

__DATE__	格式为 MM dd yyyy 格式的当前日期
__FILE__	当前源文件名
__IAR_SYSTEMS_ICC	IAR C 编译器的标志符
__LINE__	当前源程序行号
__STDC__	ANSI-C 编译器的标志符
__TID__	目标标志符
__TIME__	格式为 hh:mm:ss 的当前时间
__VER__	返回整型版本号

4. 本征函数

本征函数允许对 MSP430 作低层的控制。为了使它们能在 C 语言程序中使用，程序文件应该包含头文件（In430.h）。经编译，本征函数成为内嵌代码，可能是单个指令，也可能是一段指令序列。

关于本征函数的功能细节，可以参看具体的芯片技术文档。下面给出一些本征函数。

_args$	返回参数数组给函数
_argt$	返回参数类型
_NOP	空操作指令
_EINT	允许中断
_DINT	禁止中断
_BIS_SR	对状态寄存器中某一位置位
_BIC_SR	对状态寄存器中某一位复位
_OPC	插入 DW 常数说明伪指令

5. 其他扩展特性

$字符	为了与 DEC 公司的 VMS-C 兼容，将$加入到有效字符集中

编译时用 SIZEOF　　取消限制 SIZEOF 运算符不能用在#if 和#elif 表达式内的规定

1.2.2　MSP430 的 C 语言的扩展关键字

通过前面的概述部分知道 MSP430 的 C 语言的扩展关键字，下面具体对各个关键字进行介绍。

1．interrupt

该关键字用于中断函数。中断函数的定义如下。

语法：interrupt void 函数名() 或者

　　interrupt[中断向量] void 函数名()

参数：中断函数没有参数。

　　中断函数可能需要指定中断向量。

返回：中断函数一般是 void，没有返回。

说明：interrupt 关键字声明了在处理器发生中断时调用。函数的参数必须为空，如果说明了中断向量，函数地址将插入该向量；如果未说明中断向量，用户必须在向量表中为中断函数提供适当的入口，最好在 cstartup 模块中提供。

下面举例说明：

```
#include <MSP430X14X.h>
//////////////////////////////////////////////
// 初始化定时器模块
void Init_TimerB(void)
{
    TBCTL = TBSSEL0 + TBCLR;          // 选择 ACLK，清除 TAR
    TBCCTL0 = CCIE;                   // TBCCR0 中断允许
    TBCCR0 = 32768;                   // 时间间隔为 1 s

    TBCTL |= MC0;                     // 增计数模式

    //初始化端口
    P1DIR = 0;
    P1SEL = 0;
    P1DIR |= BIT0;
    return;
}
//////////////////////////////////////////////
// 定时器中断
interrupt [TIMERB0_VECTOR] void TimerB_ISR(void)
{
```

```
    int i;
    //翻转 P1.0 管脚
    if(P1OUT & BIT0) P1OUT &= ~(BIT0);
    else P1OUT |= BIT0;
    for(i = 100;i > 0;i--) ;
}
```

上面的例子说明了 interrupt 定义的中断函数的使用。

2. monitor

该关键字是使函数进入原型（atomic）操作状态。

语法：monitor 函数类型 函数名（参数表）

参数：该函数可以有参数，也可以没有参数。

返回：函数可以有返回，也可以没有返回。

说明：monitor 关键字使得在函数执行期间禁止中断，使函数执行不可中断。在其他所有方面，有 monitor 声明的函数与普通函数相同。

下面举例说明 monitor 关键字的用法。下面的例子演示在测试标志时禁止中断。如果标志没有设置，函数将设置，当退出函数时，中断状态恢复到原先的状态。

```
char print_flag;
monitor int GetFlag(char *pFlag)
{
    if(*pFlag == 0)
    {
        *pFlag = 1;
    }
    else
    {
        *pFlag = 0;
    }
    return *pFlag;
}
void Test()
{
    if(GetFlag(&print_flag))
    {
    ......
    }
}
```

上面的例子演示了 monitor 的用法。

3. no_init

该关键字是非易失变量的类型修正符。

语法：no_init 变量声明

说明：在默认情况下，MSP430 的 C 语言编译器将变量存放于主 RAM 中，并在启动时对其进行初始化。no_init 类型修正符使编译器把变量放在非易失 RAM 区中（如 EEPROM、FLASH 等），在启动时也不对它们作初始化。在 no_init 变量的声明中，不能含有初始化。如果用了非易失 RAM，连接时要安排在非易失 RAM 区，地址范围为 0x0000~0xFFFF，而实际可用范围是 0x200~0XFFDF。下面举例说明。

```
no_init int A[20];
no_init n;
```

通过上面的例子知道了 no_init 的使用方法。

4. sfrb

该关键字用于声明单字节 I/O 数据类型对象。

语法：sfrb 标志符=常量表达式

说明：sfrb 表示一个 I/O 寄存器，具有以下特点：

- 它等价于无符号字符。
- 它只能直接寻址。
- 它驻留在地址范围 0x00~0xFF 之内。

下面举例说明：

```
sfrb P1OUT=0x0021;
```

上面的例子定义了 P1 口的输出寄存器。

5. sfrw

该关键字用于声明双字节 I/O 数据类型对象。

语法：sfrw 标志符=常量表达式

说明：sfrw 表示一个 I/O 寄存器，具有以下特点：

- 它等价于无符号字符。
- 它只能直接寻址。
- 它驻留在地址范围 0x100~0x1FF 之内。

下面举例说明：

```
sfrw WDTCTL=0x0021;
```

上面的例子定义了看门狗的寄存器。

1.2.3 MSP430 的#pragma 编译命令

#pragma 编译命令是给 MSP430 的 C 编译器的指令，使 MSP430 的 C 编译器完成特定的编译功能。下面具体介绍各个#pragma 编译命令。

1. bitfields = default

恢复默认的位域存储次序。

语法：#pragma bitfields = default

说明：使编译器按照正常次序分配位域。

2. bitfields = reversed

翻转位域的存储次序。

语法：#pragma bitfields = reversed

说明：使编译器从域的最高有效位开始分配位域，而非从最低有效位开始。

标准 C 语言允许存储顺序与执行相关，用此关键字可避免移植性问题。

例如：下列结构类型在存储器中的默认存储顺序如下。

```
struct
{
    short N0:3;
    short N1:5;
    short N2:4;
    short N3:4;
} bits
```

实际结构如图 1-1 所示。

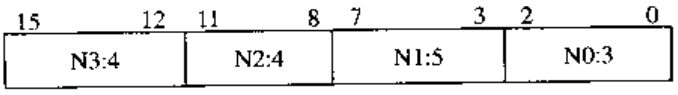


图 1-1 默认存储顺序

下面为翻转位域的存储顺序。

```
#pragma bitfields = reversed
struct
{
    short N0:3;
    short N1:5;
    short N2:4;
```

```
    short N3:4;
} bits
```

如图 1-2 所示为翻转位域的存储顺序。

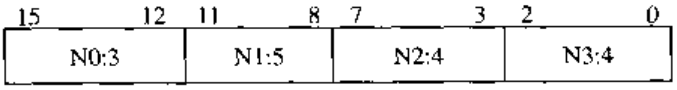


图 1-2　翻转存储顺序

通过图 1-1 和图 1-2 的比较可以理解默认存储顺序和翻转存储顺序。

3. codeseg

设置代码段名.

语法：#pragma codeseg（段名）

其中，段名一定不能与数据段发生冲突。

说明：此编译命令将后续代码放在命名的段内。它等价于使用“-R”选项。此编译命令只能被编译器执行一次。

例如：把代码段定义为 ROM。

```
#pragma codeseg(ROM)
```

4. function = default

将函数定义恢复为默认类型。

语法：#pragma function = default

说明：取消 function = interrupt 和 function = monitor 编译命令。

例如：声明外部函数 fun1 为中断函数，fun2 为普通函数。

```
#pragma function = interrupt
extern void fun1();
#pragma function = default
extern void fun2();
```

5. function = interrupt

将函数定义为 interrupt。

语法：#pragma function = interrupt

说明：此编译命令使后续函数定义为中断类型。这是说明函数属性为中断的另一种形式，但是该编译命令不提供矢量选项。

例如：下面为中断函数，函数的地址必须放入中断向量表中。

```
#pragma function = interrupt
void Timer_ISR()
{
......
}
#pragma function = default
```

上面的例子演示了中断函数的定义。

6. function = monitor

将函数定义为不可中断（atomic）状态。

语法：#pragma function = monitor

说明：使后续函数定义为 monitor 类型，这是说明函数属性为 monitor 的另一种形式。

例如：下面函数执行期间，暂时禁止中断。

```
#pragma function = monitor
void fun()
{
......
}
#pragma function = default
```

上面的例子表明在定义完了 monitor 函数后需要恢复函数为一般属性。

7. language = default

将可用的关键字设置恢复为默认状态。

语法：#pragma language = default

说明：将 C 编译器“-e”选项设置的扩展关键字可用状态恢复到默认状态。

8. language = extended

设置扩展关键字为可用状态。

语法：#pragma language = extended

说明：使扩展关键字可用，与 C 编译器的“-e”选项无关。

例如：将扩展关键字设置为可用，

```
#pragma language = extended
no_init int user_info;
#pragma language = default
int flag;
```

上面例子演示了#pragma language = extended 的作用。

9. memory = constseg

在默认情况下，将常数放入所命名的段中。

语法：#pragma memory = constseg（段名）

说明：在默认情况下，将常数放入所命名的段中。后续声明隐含取得 const 存储类。可用关键字 no_init 和 const 跨越此设置。另外，段名一定不能是编译器保留的段名之一。

例如：将常量数组 const_a 放入 ROM 段的 TABLE 中。

```
#pragma memory = constseg(TABLE)
char const_a[] = {1,2,3,4,5,6,7,8,9};
#pragma memory = default
```

上面的例子演示了#pragma memory = constseg（段名）的使用。

10. memory = dataseg

在默认情况下，将变量放入所命名的段中。

语法：#pragma memory = dataseg（段名）

说明：在默认情况下，将变量放入所命名的段中。可用关键字 no_init 和 const 跨越此设置。如果省略，变量将放入 UDATA0（非初始化变量）或 IDATA0（初始化变量）中。在变量定义中不提供初始化值。在模块中预备有 10 个不同的数据段，用户可以在程序的任何位置切换到任意一个已经定义的数据段。

例如：下面将 5 个变量放入名为 UART 的读/写区域。

```
#pragma memory = dataseg(UART)
char TX_LEN;
char RX_LEN;
char TX_FLAG;
char RX_FLAG;
int Rate;
#pragma memory = default
```

上面的例子演示了#pragma memory = dataseg（段名）的使用。

11. memory = default

将存储器的分配恢复到默认区域。

语法：#pragma memory = default

说明：将对象的存储器分配到默认区域。后续非初始化数据分配到 UDATA0，初始化

数据分配到 IDATA0 中。

12. memory = no_init

在默认情况下，将变量放入 no_init 段。

语法：#pragma memory = no_init

说明：将变量放入 no_init 段，因此将不作初始化，并将驻留在非易失 RAM 中。这是存储器属性 no_init 的另外一种形式。用关键字 const 可以跨越默认情况。no_init 段必须连接到非易失 RAM 的物理地址。

例如：将变量 buf 放到未初始化存储区，变量 n 和 m 放入 DATA 区。

```
#pragma memory = no_init
char buf[100];
#pragma memory = default
int n;
int m;
```

上面的例子演示了#pragma memory = no_init 的使用，

局部变量和参数不能驻留在它们的默认段及堆栈之外的任何其他段中。如果因为存储器编译命令使函数声明中用了非默认存储器段，将产生错误信息。

13. warnings = default

将编译器警告消息输出恢复到默认状态。

语法：#pragma warnings = default

说明：使编译器警告消息的输出恢复到 C 编译器“-w”选项设置的默认状态。

14. warnings = off

关闭编译器警告消息的输出。

语法：#pragma warnings = off

说明：关闭编译器警告消息的输出，与 MSP430 的 C 编译器的“-w”选项状态无关。

15. warnings = on

打开编译器警告消息的输出。

语法：#pragma warnings = on

说明：打开编译器警告消息的输出，与 MSP430 的 C 编译器的“-w”选项状态无关。

1.2.4 MSP430 的预定义符号

在前面给出了 MSP430 的 C 语言的预定义符号，下面就各个符号进行具体的介绍。

1. __DATE__

当前日期。

语法：__DATE__

说明：以 MM dd yyyy 形式返回编译的日期。

2. __FILE__

当前源文件名。

语法：__FILE__

说明：返回当前正在编译的文件名。

3. __IAR_SYSTEMS_ICC

IARC 编译器标志符。

语法：__IAR_SYSTEMS_ICC

说明：返回 1。可用#ifdef … #else … #endif 进行测试，以便检验是否由 IAR 的 C 编译器在进行编译。

4. __LINE__

当前源程序行号。

语法：__LINE__

说明：返回当前在编译的源程序的行号。

5. __STDC__

标准 C 编译器标志符。

语法：__STDC__

说明：返回 1。可用#ifdef … #else … #endif 进行测试，以便检验是否由标准 C 编译器在进行编译。

6. __TID__

目标标志符。

语法：__TID__

说明：目标标志符含有：IAR 的 MSP430 的 C 编译器本征标志、目标标示、“-v”选项值及“-m”选项值。具体的分配如图 1-3 所示。

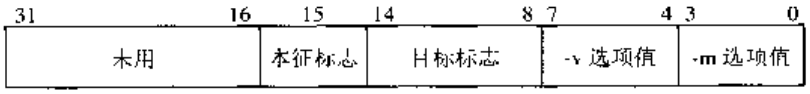


图 1-3　目标标志符

由图 1-3 可以看出标志符的内容。

7. __TIME__

当前时间。

语法：__TIME__

说明：以 hh:mm:ss 形式返回编译时间。

8. __VER__

返回编译器版本号。

语法：__VER__

说明：返回编译器的版本号。版本号为整数。

例如：测试版本号是否是 3.33。

```
#if __VER__ == 333
#message "compiler version is 3.33"
#endif
```

1.2.5　MSP430 的本征函数

前面列出了 MSP430 的一些本征函数，下面就具体的函数进行介绍。

1. _args$

该本征函数返回参数数组。

语法：_args$

说明：_args$是保留字，它返回字符数组。该数组包含当前函数的形式参数的说明列表。具体内容参见表 1-6。

表 1-6　返回字符数组

偏 移 量	内　　容
0	参数 1，类型按照_args$格式
1	参数 1 字节数
2	参数 2，类型按照_args$格式
3	参数 2 字节数
4	参数 3，类型按照_args$格式
5	参数 3 字节数
⋮	⋮
2n−2	参数 n−1，类型按照_args$格式
2n−1	参数 n−1 字节数
2n	\0

通过表 1-6 可以理解_args$返回的数组的内容。如果参数字节大于 127，则最大取 127。_args$只能在定义的函数内使用。如果说明了可变长度参数列表，那么参数表的结束是最后一个显示参数，因此用户无法简单判断可选参数的类型和大小。

2. _argt$

返回参数的类型。

语法：_argt$(v)

说明：该函数可以返回多种类型。具体的类型参看表 1-7。

表 1-7　返回参数类型

返 回 值	类　　型	返 回 值	类　　型
1	unsigned char	8	long
2	char	9	float
3	unsigned short	10	double
4	short	11	long double
5	unsigned int	12	pointer/address
6	int	13	union
7	unsigned long	14	struct

通过表 1-7 可以知道函数可以返回哪些类型的参数。

下面的例子可以来判断返回的类型。

```
switch(_argt$(i))
{
case 1:
    printf("unsigned char \n");
    break;
case 2:
    printf("char \n");
    break;
case 3:
    printf("unsigned short \n");
    break;
case 4:
    printf("short \n");
    break;
case 5:
    printf("unsigned int \n");
    break;
    ……
default:break;
}
```

3. _BIC_SR

状态寄存器屏蔽复位。

语法：unsigned short _BIC_SR(unsigned short mask)

说明：该函数用来屏蔽状态寄存器的某个位。

例如：

```
//关中断
old_SR = _BIC_SR(0x08);
……
//恢复中断
_BIS_SR(old_SR);
```

4. _BIS_SR

状态寄存器屏蔽置位。

语法：unsigned short _BIS_SR(unsigned short mask)

说明：该函数用来对状态寄存器的某个位进行置位。

例如：

```
//进入低功耗 LPM3 模式
_BIS_SR(0xC0);
```

5. _DINT

禁止中断。

语法：_DINT();

说明：该函数禁止中断。

6. _EINT

打开中断。

语法：_EINT();

说明：该函数打开中断。

例如：

```
_DINT();                    //关闭中断
//硬件初始化
……
_EINT();                    //打开中断
```

7. _NOP

执行空操作。

语法：_NOP();

说明：该函数执行空操作。

例如：延迟一点时间。

```
for(int n = 0; n < 10; n++) _NOP();
```

8. _OPC

执行 DW 常数说明伪指令。

语法：_OPC(const unsigned char)

说明：插入 DW 常数说明。

1.2.6　MSP430 的段定义

MSP430 系列芯片的存储空间分为不同的段。MSP430 的 C 语言编译器将程序和数据放到不同的段里。

1. 存储器分布与段定义

MSP430 的 C 语言编译器将代码和数据放入各个命名的段中，并由连接器实现连接。段定义的细节对于汇编语言程序模块的编程和解释编译器的汇编语言输出都是必需的。一般说来，MSP430 系列芯片的存储器的段分配如图 1-4 所示。

图 1-4 只是一个示意图，具体的地址分配需要根据具体的芯片型号来确定。通过图 1-4 可以对存储器的分段有一个宏观的认识，下面就具体的各个段加以说明。

图 1-4 存储器段的分配示意图

2. CCSTR 段

该段作为字符串清单。

类型：只读。

说明：汇编语言可以访问。保存 C 语言程序字符串清单。启动时，段的内容复制到 ECSTR。

3. CDATA0 段

由 CSTARTUP 实现将段中常数对 IDATA0 段中变量作初始化。

类型：只读。

说明：汇编语言可访问。CSTARTUP 将初始化值从该段复制到 IDATA0 段。

4. CODE 段

程序代码。

类型：只读.

说明：汇编语言可访问。保存用户程序代码和各种库子程序。用 C 语言调用汇编语言子程序，必须符合使用中的存储器模块的调用规则。

5. CONST 段

常量段。

类型：只读。

说明：汇编语言可访问。该段用于存放常量。可用于汇编语言程序中声明常量。

6. CSTACK 段

该段为堆栈。

类型：读/写。

说明：汇编语言可访问。该段作为堆栈使用。

7. CSTR 段

字符串清单。

类型：只读。

说明：汇编语言可访问。如果 MSP430 的 C 语言编译器未用“-y”选项时（默认情况），保存 C 程序字符串清单。

8. ECSTR 段

字符串清单的可写复制。

类型：读/写。

说明：汇编语言可访问。保存 C 程序字符串清单。

9. IDATA0 段

变量的初始化静态数据。

类型：读/写。

说明：汇编语言可访问。保存内部数据存储器中的静态变量。

10. INTVEC 段

中断向量段。

类型：只读。

说明：汇编语言可访问。保存用 interrupt 扩展关键字产生的中断向量表，也可以是用户编写的中断向量表的入口。该段的地址空间是固定的，必须是 0xFFE0~0xFFFF。

11. NO_INIT 段

非易失变量。

类型：读/写。

说明：汇编语言可访问。保存存放到非易失存储器中的变量。这些变量可以声明 no_init 类型由编译器分配，也可以用#pragma 编译命令创建 no_init。还可以用汇编语言程序人工创建。

12. UDATA0 段

非初始化静态变量。

类型：读/写。

说明：汇编语言可访问。该段保存存储器变量。该段不作显示初始化，而是由 CSTARTUP 隐式地初始化为 0。

通过这一章对 C 语言的基础知识和 MSP430 的 C 语言扩展知识的介绍，对采用 C 语言开发 MSP430 系列单片机有了基本的认识，也为下面几章介绍具体的开发例子打下坚实的基础。

另外，为了便于计算，MSP430 的 C 语言本身提供了一系列数学运算的函数，也提供一些其他的处理函数（比如字符串处理函数）。为了便于理解和查找，下面列出几个头文件以便参考。

附录 A　相关头文件

```
/*  - in430.h -
   MSP430 的本征函数头文件
*/

#ifndef __IN430_INCLUDED
#define __IN430_INCLUDED
```

```
#if __CTD__ & 0x8000
#pragma function=intrinsic(0)
#endif

unsigned short _BIS_SR(unsigned short);
unsigned short _BIC_SR(unsigned short);
void _DINT(void);
void _EINT(void);
void _NOP(void);
void _OPC(const unsigned short op);

#if __CTD__ & 0x8000
#pragma function=default
#endif

#endif  /* __IN430_INCLUDED */

/*  - MATH.H -
      用于运算的math.h头文件
*/

#ifndef _MATH_INCLUDED
#define _MATH_INCLUDED

#include "sysmac.h"

#ifndef HUGE_VAL
#if __FLOAT_SIZE__ == __DOUBLE_SIZE__
#define HUGE_VAL 3.402823466e+38
#else
#define HUGE_VAL 1.79769313486231580+308
#endif
#endif

#define __EDOM_VALUE    HUGE_VAL

/* PI, PI/2, PI/4, 1/PI, 2/PI */
#define __PI            3.14159265358979323846264
#define __PIO2          1.57079632679489661923
#define __PIO4          0.78539816339744830961
#define __INVPI         0.31830988618379067154
#define __TWOOPI        0.63661977236758134308

/* SQRT(2), SQRT(2) - 1, SQRT(2) - 1,SQRT(2) / 2 */
#define __SQRT2         1.41421356237309504880168872
#define __SQ2P1         2.41421356237309504802
#define __SQ2M1         0.41421356237309504802
#define __SQRTO2        0.70710678118654752440
```

```
/* LN(10), TWO-LOG(e), LN(2) e */
#define __LN10          2.302585092994045684
#define __LOG2E         1.44269504088896340735992477
#define __LOG2          0.69314718055994530941723212
#define __E             2.71828182845904523536028747


#if __IAR_SYSTEMS_ICC__ < 2
#if __TID__ & 0x8000
#pragma function=intrinsic(0)
#endif
#endif

#ifndef MEMORY_ATTRIBUTE
#define MEMORY_ATTRIBUTE
#endif

__INTRINSIC MEMORY_ATTRIBUTE double atan(double);

__INTRINSIC MEMORY_ATTRIBUTE double atan2(double, double);

__INTRINSIC MEMORY_ATTRIBUTE double cos(double);

__INTRINSIC MEMORY_ATTRIBUTE double cosh(double);

__INTRINSIC MEMORY_ATTRIBUTE double fabs(double);

__INTRINSIC MEMORY_ATTRIBUTE double fmod(double, double);

__INTRINSIC MEMORY_ATTRIBUTE double exp(double);

__INTRINSIC MEMORY_ATTRIBUTE double ldexp(double, int);

__INTRINSIC MEMORY_ATTRIBUTE double log(double);

__INTRINSIC MEMORY_ATTRIBUTE double log10(double);

__INTRINSIC MEMORY_ATTRIBUTE double modf(double, double *);

__INTRINSIC MEMORY_ATTRIBUTE double pow(double , double);

__INTRINSIC MEMORY_ATTRIBUTE double sin(double);

__INTRINSIC MEMORY_ATTRIBUTE double sinh(double);

__INTRINSIC MEMORY_ATTRIBUTE double sqrt(double);
```

```
__INTRINSIC MEMORY_ATTRIBUTE double tan(double);

__INTRINSIC MEMORY_ATTRIBUTE double tanh(double);

__INTRINSIC MEMORY_ATTRIBUTE double floor(double);

__INTRINSIC MEMORY_ATTRIBUTE double ceil(double);

__INTRINSIC MEMORY_ATTRIBUTE double frexp(double, int *);

__INTRINSIC MEMORY_ATTRIBUTE double acos(double);

__INTRINSIC MEMORY_ATTRIBUTE double asin(double);

#if __TAR_SYSTEMS_ICC__ < 2
#if __TID__ & 0x8000
#pragma function-default
#endif
#endif

#endif /* _MATH_INCLUDED */

/* - STDLIB.H -
   标准C语言的stdlib.h头文件
*/

#ifndef _STDLIB_INCLUDED
#define _STDLIB_INCLUDED

#include "sysmac.h"

#ifndef NULL
#define NULL    ((void*) 0 )
#endif

typedef struct
{
  int   quot;
  int   rem;
} div_t;

typedef struct
{
  long int      quot;
  long int      rem;
} ldiv_t;

#define RAND_MAX        32767
```

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1

#define MB_CUR_MAX      1

#if __IAR_SYSTEMS_ICC__ < 2
#if __TID__ & 0x8000
#pragma function=intrinsic(0)
#endif
#endif

#ifndef MEMORY_ATTRIBUTE
#define MEMORY_ATTRIBUTE
#endif

#ifndef PTR_ATTRIBUTE
#define PTR_ATTRIBUTE
#endif

__INTRINSIC MEMORY_ATTRIBUTE void *malloc(size_t);

__INTRINSIC MEMORY_ATTRIBUTE void free(void *);

__INTRINSIC MEMORY_ATTRIBUTE void exit(int);

__INTRINSIC MEMORY_ATTRIBUTE void *calloc(unsigned int, size_t);

__INTRINSIC MEMORY_ATTRIBUTE void *realloc(void *, size_t);

__INTRINSIC MEMORY_ATTRIBUTE int  atoi(const char *);

__INTRINSIC MEMORY_ATTRIBUTE long atol(const char *);

__INTRINSIC MEMORY_ATTRIBUTE double atof(const char *);

__INTRINSIC MEMORY_ATTRIBUTE double strtod(const char *, char **);

__INTRINSIC MEMORY_ATTRIBUTE long int strtol(const char *, char **, int);

__INTRINSIC MEMORY_ATTRIBUTE unsigned long int strtoul(const char *, char
**, int);

__INTRINSIC MEMORY_ATTRIBUTE int rand(void);

__INTRINSIC MEMORY_ATTRIBUTE void srand(unsigned int);

__INTRINSIC MEMORY_ATTRIBUTE void abort(void);
```

```
__INTRINSIC MEMORY_ATTRIBUTE int abs(int);

__INTRINSIC MEMORY_ATTRIBUTE div_t div(int, int);

__INTRINSIC MEMORY_ATTRIBUTE long int labs(long int);

__INTRINSIC MEMORY_ATTRIBUTE ldiv_t ldiv(long int, long int);

__INTRINSIC MEMORY_ATTRIBUTE void *bsearch(const void *, const void *,
size_t, size_t, int (*)(const void *,const void *));

__INTRINSIC MEMORY_ATTRIBUTE void qsort(void *, size_t, size_t,
                                        int (*)(const void *, const void *));

#if __IAR_SYSTEMS_ICC__ < 2
#if __TID__ & 0x8000
#pragma function=default
#endif
#endif

#endif /* _STDLIB_INCLUDED */

/*  - STRING.H -
标准C语言的string.h头文件
*/

#ifndef _STRING_INCLUDED
#define _STRING_INCLUDED

#include "sysmac.h"

#ifndef NULL
#define NULL    ((void*)0)    #endif

#if __IAR_SYSTEMS_ICC__ < 2
#if __TID__ & 0x8000
#pragma function=intrinsic(0)
#endif
#endif

#ifndef MEMORY_ATTRIBUTE
#define MEMORY_ATTRIBUTE
#endif

__INTRINSIC MEMORY_ATTRIBUTE void *memcpy(void *, const void *, size_t);

__INTRINSIC MEMORY_ATTRIBUTE void *memmove(void *, const void *, size_t);
```

```
__INTRINSIC MEMORY_ATTRIBUTE void *memchr(const void *, int, size_t);
__INTRINSIC MEMORY_ATTRIBUTE void *memset(void *, int, size_t);
__INTRINSIC MEMORY_ATTRIBUTE int memcmp(const void *, const void *, size_t);
__INTRINSIC MEMORY_ATTRIBUTE char *strchr(const char *, int);
__INTRINSIC MEMORY_ATTRIBUTE int strcmp(const char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE int strncmp(const char *, const char *, size_t);
__INTRINSIC MEMORY_ATTRIBUTE int strcoll(const char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE size_t strlen(const char *);
__INTRINSIC MEMORY_ATTRIBUTE size_t strcspn(const char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE size_t strspn(const char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE char *strpbrk(const char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE char *strrchr(const char *, int);
__INTRINSIC MEMORY_ATTRIBUTE char *strstr(const char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE char *strcat(char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE char *strncat(char *, const char *, size_t);
__INTRINSIC MEMORY_ATTRIBUTE char *strcpy(char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE char *strncpy(char *, const char *, size_t);
__INTRINSIC MEMORY_ATTRIBUTE char *strerror(int);
__INTRINSIC MEMORY_ATTRIBUTE char *strtok(char *, const char *);
__INTRINSIC MEMORY_ATTRIBUTE size_t strxfrm(char *, const char *, size_t);

#if __IAR_SYSTEMS_ICC__ < 2
#if __TID__ & 0x8000
#pragma function=default
#endif
#endif

#endif
```

1.3　MSP430 的开发调试环境

支持 MSP430 的开发调试环境比较多。本章介绍由 IAR 公司提供的开发调试环境：IAR Embedded Workbench 及调试器 C-SPY。

1.3.1　Embedded Workbench 概述

Embedded Workbench 支持多种单片机。它具有以下特性：

- 支持 Windows 98/Windows NT/Windows 2000 操作系统。
- 具有 Windows 风格的可视化开发环境。
- 集成所有的工具（编译、连接等），方便使用。
- 支持直观的拖放功能。
- 具有超文本风格的帮助。
- 可以采用 Make 进行重新编译、连接。

Embedded Workbench 采用创建项目（Project）的方式来进行软件的开发和管理。Embedded Workbench 包含的实用工具有：

- 具有语法突出显示的文本编辑器。
- 编译器。
- 汇编器。
- 连接器。
- 函数管理器。
- Make 工具。
- 调试器 C-SPY。

用户可以使用集成开发环境的文本编辑器编写程序源代码。文本编辑器具有以下特性：

- 根据 C 语言的语法来区别字体的颜色。
- 具有查找和替换功能，能够非常方便地对程序进行编辑。
- 可以从出错的列表中直接跳到文本中相应的出错位置。
- 能够检查括号是否匹配，这在编写程序时非常有用。
- 能够实现程序的缩进，使程序具有良好的代码风格。
- 可以对多个窗口进行编辑。

- 可以对所有文件或某几个文件进行设置单独的选项。

在编辑完程序代码后，用户可以对程序代码进行编译连接；编译连接成功后，可以运行程序，并且可以对程序进行调试。使用 Embedded Workbench 集成的 C-SPY 工具对程序进行运行和调试。

调试器 C-SPY 具有以下特性：

- 具有 Windows 风格的可视化界面。
- 能够对汇编语言或者 C 语言进行调试。
- 能够进行软件仿真（Simulator）。
- 能够进行硬件仿真（Emulator）。
- 能够设置断点，进行单步运行。
- 支持多种单步运行方式（如 Step in，Step over 等）。
- 可以观察寄存器的值。
- 能够查看内存的值。

下面详细介绍 Embedded Workbench 的安装及操作。

1.3.2 Embedded Workbench 的安装

1 首先选中 fet_r300.exe（如图 1-5 所示）并双击该文件，弹出安装对话框，单击“Setup”按钮进行解压（如图 1-6 所示）。

图 1-5 选择安装文件

图 1-6 文件解压

2 解压完成后进入安装界面，单击“Next”按钮，出现“license agreement”对话框。单击“Yes”按钮后显示选择文件夹的对话框，如图 1-7 所示。

图 1-7　选择文件夹

3 如果用户选择默认的文件夹，则单击“Next”按钮；如果用户需要重新指定安装的位置，则单击“Browse”按钮，单击“Browse”按钮后，显示文件夹设置对话框，如图 1-8 所示。

4 在此直接输入文件夹的名字，如果该文件夹不存在，则创建该文件夹。单击“Next”按钮后显示选择安装类型的对话框，如图 1-9 所示。一般选择 Full 安装。

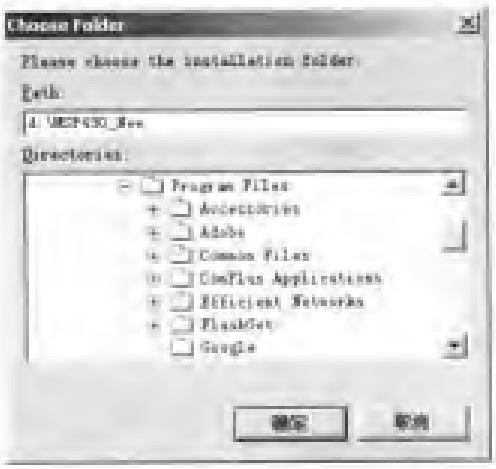


图 1-8　改变文件夹

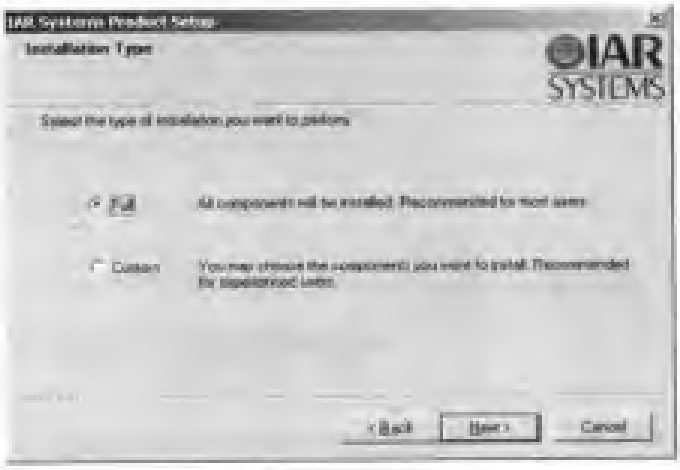


图 1-9　选择安装类型

5 单击“Next”按钮后开始安装程序，最后会显示安装完成的对话框。如图 1-10 所示。

在图 1-10 中，将两个复选框都不选中，然后单击“Finish”按钮，完成 fet_r300.exe 安装。

6 继续安装 fet_r300u.exe。在图 1-5 中，选中 fet_r300u.exe，并双击该文件进行安装，该文件主要用来安装 C-SPY 工具。安装方法和前面相似，在选择安装类型（如图 1-11 所示）时，选择“IAR Full Version”。

图 1-10　安装完成

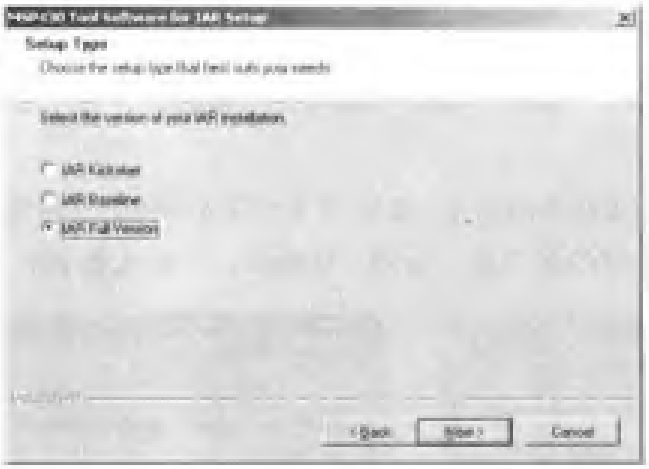


图 1-11　选择安装类型

7 单击“Next”按钮，显示选择文件夹对话框，如图 1-12 所示。

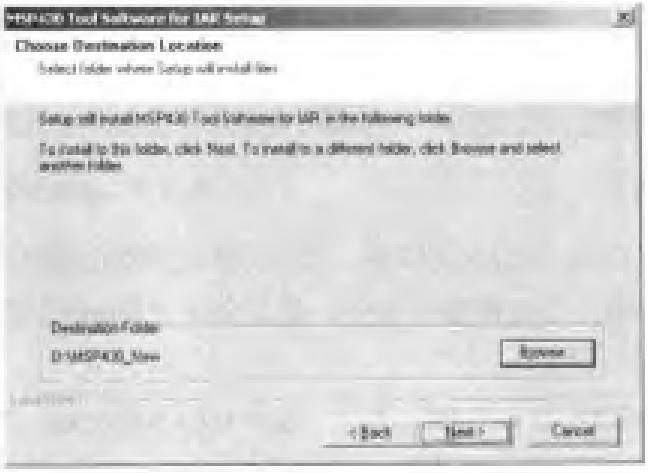


图 1-12　选择文件夹

8 单击“Next”按钮后开始安装程序。安装完成后，在计算机的“开始”|“程序”菜单中就会显示有“IAR Systems”菜单。

1.3.3　Embedded Workbench 的使用

1．开始创建

Embedded Workbench 是采用项目形式组织文件的。因此用户在进行软件开发时，首先需要创建一个新的项目。下面介绍项目的创建过程。

从计算机的开始菜单中启动 Embedded Workbench，如图 1-13 所示。

（1）创建项目

1 单击“File”菜单，在“File”菜单列表中选择“New”，弹出“新建”对话框，如图 1-14 所示。

图 1-13　Embedded Workbench 初始界面

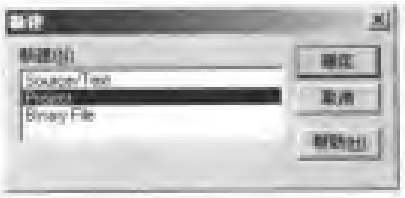


图 1-14　创建项目对话框

2 选择“Project”后单击“确定”按钮，弹出“New Project”（新建项目）对话框。设定项目所在的文件夹，并输入项目的名字。如图 1-15 所示。

3 单击“Create”按钮，这样就完成了项目的创建，如图 1-16 所示。

图 1-15　新建项目对话框

图 1-16　完成项目创建

4 一般创建的项目默认是“Debug”（调试），用户也可以根据需要选择“Targets”

中的“Release”（发行），如图 1-17 所示。

5 在创建完成项目后，用户需要创建程序。单击“File”菜单，在“File”菜单列表中选择“New”，弹出“新建”对话框，如图 1-18 所示。

图 1-17 选择项目的发行模式

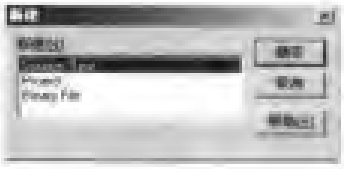


图 1-18 创建文件对话框

6 选择“Source/Text”后单击“确定”按钮，弹出程序编辑窗口，如图 1-19 所示。

图 1-19 程序编辑窗口

7 在该程序编辑窗口中输入下面的代码。

```
#include <stdio.h>

void memset(int *pBuf,int len,int val);
void memAdd(int *in1,int*in2,int *out,int len);
int main(void)
{
    int i;
    int a[10];
    int b[10];

    for(i = 0;i < 10;i++)
    {
       a[i] = i;
    }

    memset(a,10,1);
    memset(b,10,2);

    memAdd(a,b,a,10);
    return 0;
}
void memset(int *pBuf,int len,int val)
{
    int i;
```

```
    for(i = 0;i < len;i++)
    {
        pBuf[i] = val;
    }
    return;
}
void memAdd(int *in1,int*in2,int *out,int len)
{
    int i;
    int temp;
    for(i = 0;i < len;i++)
    {
        temp = in1[i] + in2[i];
        out[i] = temp;
    }
    return;
}
```

8 按“Ctrl+S”快捷键将上面的程序保存。命名为 sample.c，并选择保存的位置，然后单击“保存”按钮。如图 1-20 所示。

虽然创建了项目，也编写了程序文件，但此时项目和文件还没有什么联系，程序文件还不是创建的项目中的程序文件。

9 接下来单击“Project”菜单，在“Project”菜单列表中选择“Files”，弹出一个向项目中添加文件的对话框，如图 1-21 所示。

图 1-20　保存文件

图 1-21　向项目中添加程序文件

10 可以向项目中添加一个文件，也可以添加多个文件，添加完文件后，单击“Done”按钮就实现了文件的添加。如果用户希望从项目中删除某些不需要的文件，可以单击“Remove”和“Remove All”按钮把文件从项目中删除。需要强调的是，文件虽然从项目

中删除，但是该文件还是在硬盘上存在。

通过以上步骤就实现了项目的创建。下面来介绍项目的设置。

（2）项目的设置

在创建完项目后，需要对该项目进行设置，以保证该项目以适当的形式进行编译连接，并形成相应的文件。

1 单击“Project”菜单，在“Project”菜单列表中选择“Options”，弹出项目设置对话框，如图 1-22 所示。

2 选择“Output Directories”，可以设置相应文件的输出位置。对于项目的设置，主要设置“XLINK”和“C-SPY”两个方面的内容。其他方面则根据用户的需要来进行设置，当然使用默认设置也可以。

下面介绍“XLINK”和“C-SPY”的设置。

3 在图 1-22 中的左边选择“XLINK”，则右边出现“XLINK”的设置属性页，选择“Include”属性页，如图 1-23 所示。

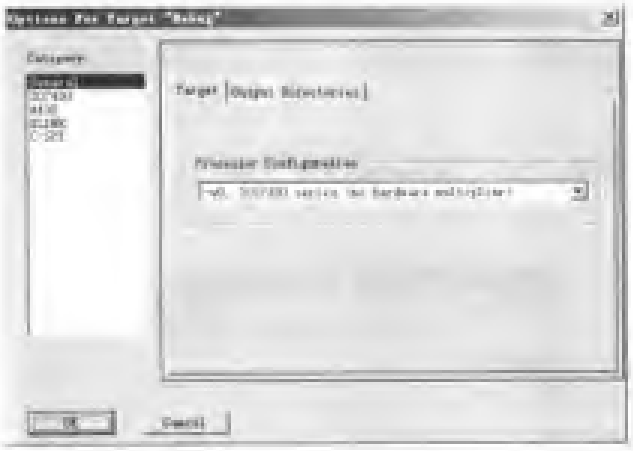


图 1-22 项目设置对话框

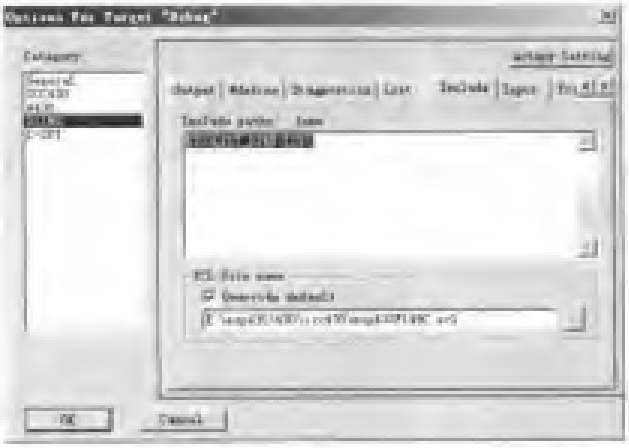


图 1-23 “XLINK”的设置

4 选中“Override default”复选框，然后单击右边的 按钮，则弹出选择文件的“打开“对话框，如图 1-24 所示。

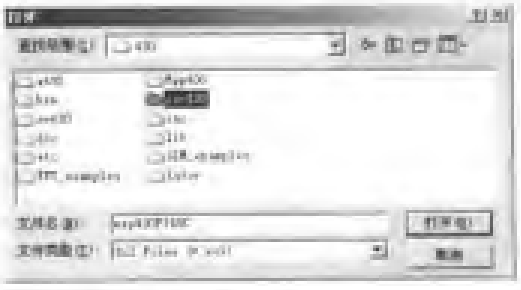


图 1-24 选择 XLINK 文件夹

5 在“查找范围”栏里选择安装 Embedded Workbench 软件的文件夹，并进入到该文件夹下面的“430”文件夹；在“430”文件夹里，双击“icc430”文件夹，弹出“*.xcl”文件的对话框，如图 1-25 所示。

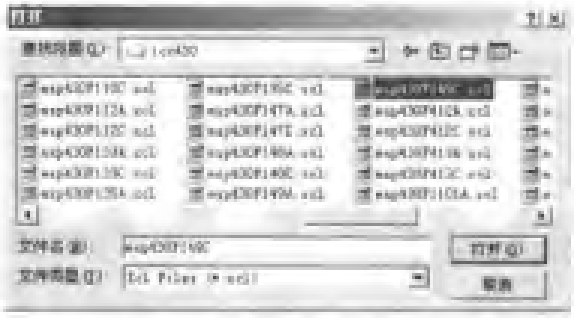


图 1-25　选择“*.xcl”文件

6 选择“msp430F149C.xcl”文件，单击“打开”按钮便完成了“XLINK”的设置。

在选择文件时，用户需要根据自己的芯片来选择相应的文件，比如，使用 MSP430F149 芯片，就选择“msp430F149C.xcl”文件；如果使用其他不同型号的芯片，则需要选择其他相应的文件。此外，同一个型号的芯片对应两个文件，例如“msp430F149C.xcl”和“msp430F149A.xcl”，后缀为 C 的文件主要是针对 C 语言程序，后缀为 A 的文件主要是针对汇编程序。

7 接着在图 1-22 中的左边选择“C-SPY”，则右边出现“C-SPY”的设置属性页，选择“Setup”属性页，如图 1-26 所示。

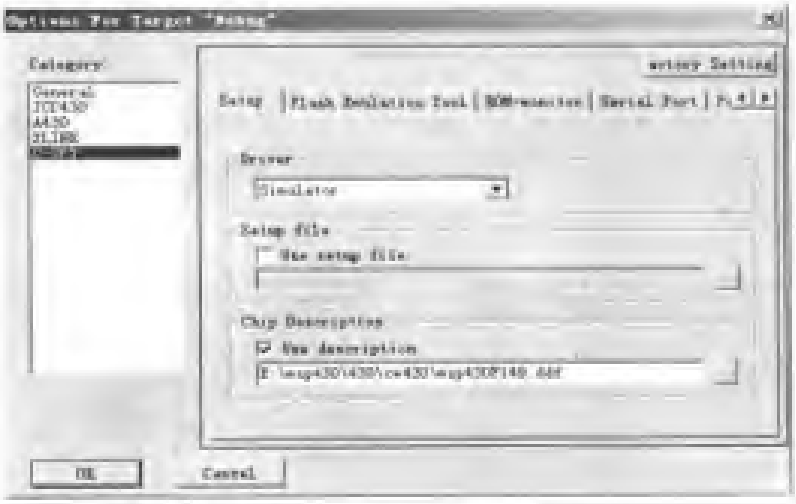


图 1-26　“C-SPY”的设置

8 在“Driver”栏，“C-SPY”的运行模式有 3 种，即“Simulator”、“ROM-Monitor”和“Flash Emulation Tool”，用户可以根据自己的情况选择相应的驱动模式。如果运行程序

时，没有目标板的支持，可以选择“Simulator”模式；如果用户需要将程序下载到目标里去运行，那么可以选择“Flash Emulation Tool”模式。

9 选中“Chip Description”复选框，然后单击右边的 按钮，则弹出选择文件夹的“打开”对话框，如图 1-27 所示。

10 在“查找范围”栏里选择安装 Embedded Workbench 软件的文件夹，并进入到该文件夹下面的“430”文件夹；在“430”文件夹里，双击“cw430”文件夹，弹出选择“*.ddf”文件的对话框，如图 1-28 所示。

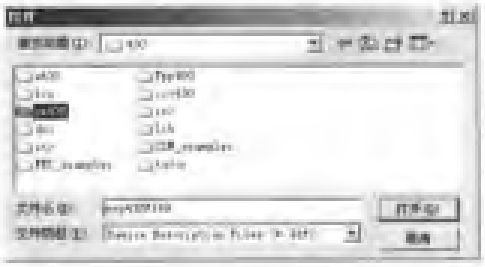


图 1-27 选择 C-SPY 文件夹

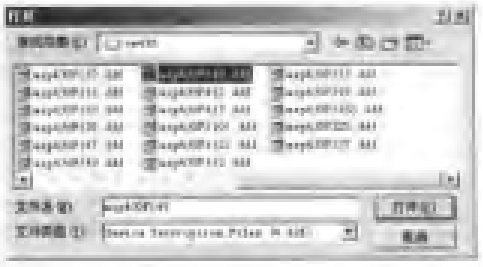


图 1-28 选择“*.ddf”文件

11 选择“msp430F149.ddf”文件，单击“打开”按钮就完成了“C-SPY”的设置。

在选择文件时，用户同样需要根据自己的芯片来选择相应的文件，比如，使用 MSP430F149 芯片，就选择“msp430F149.ddf”文件；如果使用其他不同型号的芯片，则需要选择其他相应的文件。

以上介绍了项目的设置。在项目设置完成后，就可以对程序进行编译、连接了。下面分别介绍程序的编译与连接。

（3）编译

编译是将 C 语言程序或汇编程序生成目标文件。Embedded Workbench 集成了编译器。单击图 1-29 中的编译按钮（第 3 行工具栏的左边第 1 个按钮）后，就会显示编译结果的界面，如图 1-30 所示。

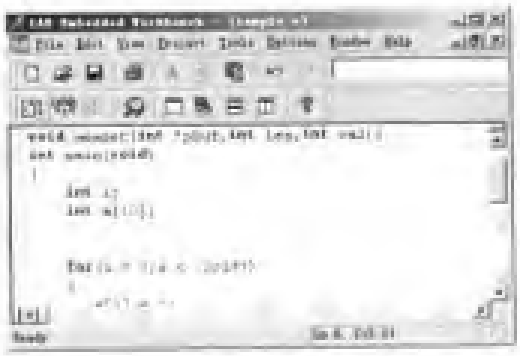


图 1-29 编译程序

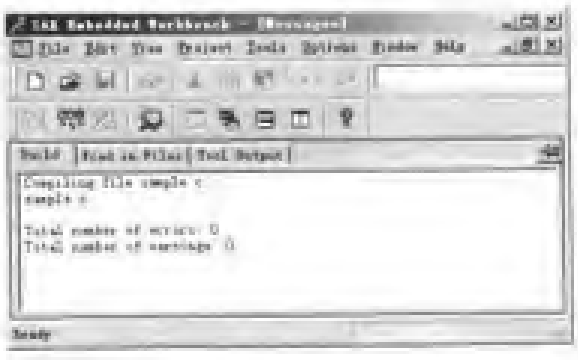


图 1-30 编译结果

从图 1-30 可以看出，若程序编译后没有警告和错误，则编译成功。如果程序有错的话，则编译结果界面就会显示出错误的内容。例如，将前面的程序改成下面的程序：

```
#include <stdio.h>

void memset(int *pBuf,int len,int val);
void memAdd(int *in1,int*in2,int *out,int len);
int main(void)
{
    int i;
    int a[10];
    int b[10];

    for(i = 0;i < 10;i++)
    {
       a[i] = j;
    }

    memset(a,10,1);
    memset(b,10,2);

    memAdd(a,b,a,10);
    return 0;
}
void memset(int *pBuf,int len,int val)
{
    int i;
    for(i = 0;i < len;i+-)
    {
       pBuf[i] = val;
    }
    return;
}
void memAdd(int *in1,int*in2,int *out,int len)
{
    int i;
    int temp;
    for(i = 0;i < len;i++)
    {
       temp = in1[i] + in2[i];
       out[i] = temp;
    }
    return;
}
```

再对上面的程序进行编译，则编译界面就会显示出错误的信息，如图 1-31 所示。

双击某条具体的错误信息，就会跳到具体的程序代码处，并在该行代码处用标记提醒

用户该行代码有错，如图 1-32 所示。

图 1-31　编译结果（有错）

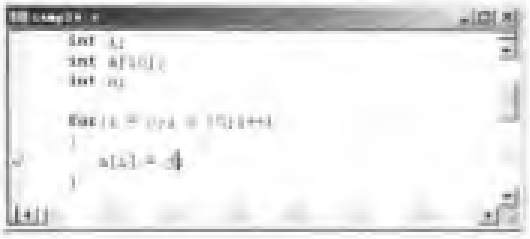


图 1-32　错误指示

通过错误指示，用户很容易找到错误的地方。但需要注意的是：编译器只能检查程序中的语法错误，而不能检查程序中的逻辑错误。

（4）连接

用户通过编译将源程序生成目标文件，通过连接将目标文件生成最终的 MSP430 单片机执行文件。

单击“Project”菜单，在“Project”菜单列表中，单击“Link”菜单后就进行了连接。如果连接成功，则结果界面如图 1-33 所示；如果连接不成功，则结果界面如图 1-34 所示。

图 1-33　连接成功界面

图 1-34　连接不成功界面

连接的错误一般是在编译时有错误引起的，因此需要用户对程序进行检查，保证程序编译成功通过。

（5）编译连接

前面介绍的编译和连接分别是单独进行的。在实际项目开发中，可以一次性完成编译和连接，这主要通过 Embedded Workbench 的 Make 来实现。在 Embedded Workbench 的工

具栏中提供了 Make 按钮（第 3 行工具栏的左边第 2 个按钮），如图 1-35 所示。

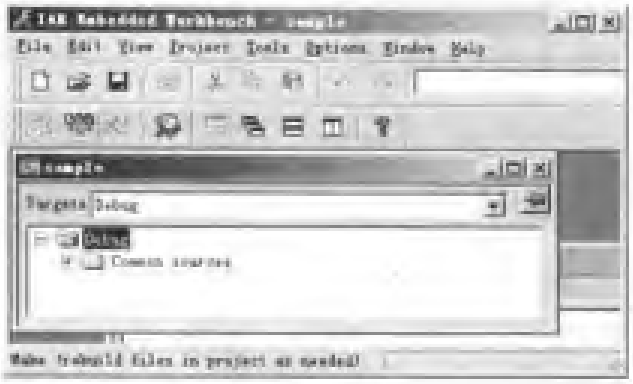


图 1-35　Make 按钮

单击 Make 按钮就可以实现编译和连接。如果成功的话，则结果界面如图 1-36 所示；如果不成功的话，则结果界面如图 1-37 所示。

图 1-36　Make 成功界面

图 1-37　Make 不成功界面

通过 Make 命令，可以一次性实现编译和连接，从而提高开发时的效率。

2. 程序调试

一般说来，调试是软件开发中常用的方法，用户在发现某段程序运行不正确时，可能需要设置断点，进行单步运行。下面进行具体介绍。

（1）启动 C-SPY

用户在编译连接完成后，就可以运行程序了。在 Embedded Workbench 开发环境中，

通过 C-SPY 来运行和调试程序。

单击工具栏中的“Debugger”按钮（第 3 行工具栏的左边第 4 个按钮，如图 1-38 所示）可以启动 C-SPY。启动界面如图 1-39 所示。

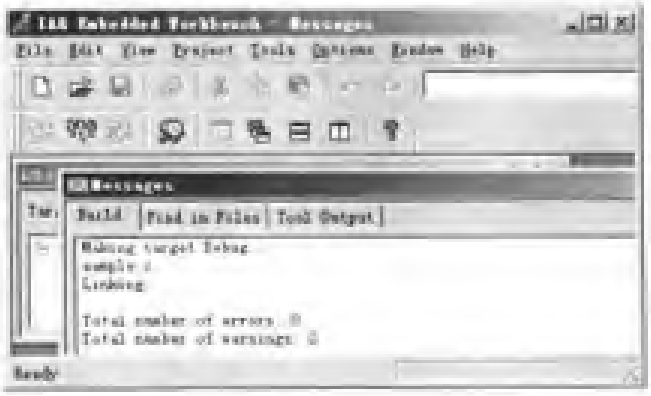


图 1-38 Debugger 按钮

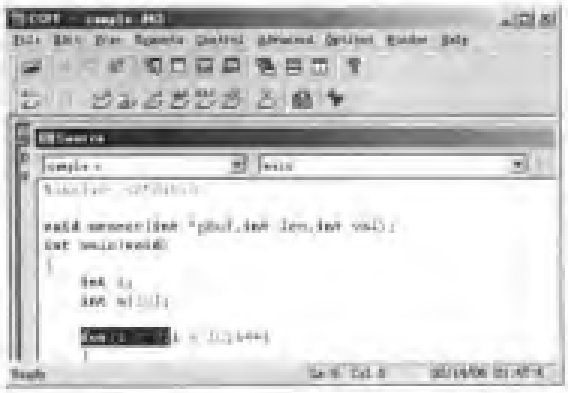


图 1-39 C-SPY 启动界面

（2）单步调试

用户可以选择运行按钮运行程序。如果用户发现某段程序运行的结果不正确，则可以选择单步运行。从图 1-39 中第 3 行工具栏的左边第 3 个按钮起就是单步运行按钮，即“Step”、“Step Into”和“Go Out”。其中“Step”单步运行时，如果单步运行的是函数调用，则直接跳过函数；“Step Into”单步运行时，如果单步运行的是函数调用，则运行到函数里；“Go Out”则从函数里跳出。用户在实际调试程序时，可以根据需要来选择不同的单步运行命令。

（3）查看变量

在调试程序时，用户有时候可能需要监视某个变量的值，这可以通过将该变量加入到一个 Watch 窗口来实现对变量值的观察。

1 单击“Window”菜单，在“Window”菜单列表中，单击“Watch”菜单，则弹出一个 Watch 窗口，如图 1-40 所示。

2 在该 Watch 窗口中单击鼠标右键，在弹出的菜单中选择“Add”，则出现如图 1-41 所示的窗口。

图 1-40 Watch 窗口

图 1-41 Watch 窗口的输入状态

3 在输入框里输入变量，就可以实现对变量的值进行观察，如图 1-42 所示。

在 Watch 窗口可以观察变量的值，也可以观察数组的值。按照上面介绍的方法，在 Watch 窗口输入数组名，就可以对数组的值进行观察，如图 1-43 所示。

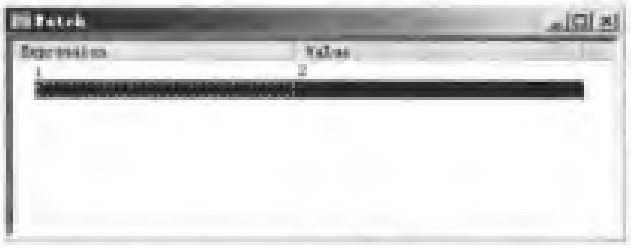


图 1-42　在 Watch 窗口观察变量值

图 1-43　在 Watch 窗口观察数组值

通过 Watch 窗口，可以很方便地查看变量或者数组的值，使用户比较容易地找到程序中逻辑不对的程序代码。

（4）查看内存

虽然前面介绍的 Watch 窗口可以观察变量的值，但是如果需要观察大块数据的值或者某块内存的值时，则需要通过查看内存的方法来查看值。

单击“Window”菜单，在“Window”菜单列表中，单击“Memory”菜单，则弹出一个 Memory 窗口，如图 1-44 所示。

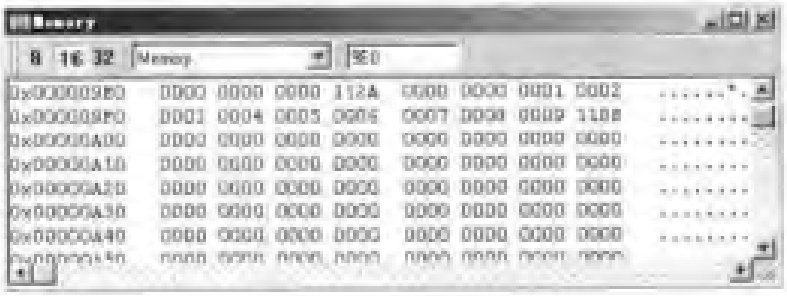


图 1-44　Memory 窗口

在 Memory 窗口的左上角，可以设置数据的显示位数。如果选择“8”，则数据按 8 位显示；如果选择“16”，则数据按 16 位显示；如果选择“32”，则数据按 32 位显示。

（5）查看特殊功能寄存器的值

单击“Window”菜单，在“Window”菜单列表中，单击“SFR”菜单，将弹出一个特殊功能寄存器窗口，如图 1-45 所示。

从特殊功能寄存器窗口左上角的列表框里可以选择观察不同的寄存器，如图 1-46 所示。

图 1-45 特殊功能寄存器窗口

图 1-46 特殊功能寄存器的选择

选择单片机的不同外设，在特殊功能寄存器窗口里就显示相应的寄存器。例如，选择“USARTs”模块，则特殊功能寄存器窗口里就显示与串口相关的寄存器的值。如图 1-47 所示。

图 1-47 查看串口模块寄存器的值

（6）查看处理器的寄存器的值

单击“Window”菜单，在“Window”菜单列表中，单击“Register”菜单，将弹出一个处理器的寄存器窗口，如图 1-48 所示。

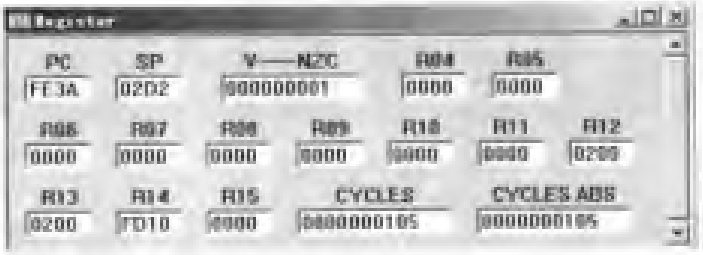


图 1-48 查看处理器的寄存器的值

通过寄存器窗口，可以观察程序运行时，处理器的寄存器的值。

3. 程序例子

经过前面的介绍，相信读者已经掌握了使用 Embedded Workbench 开发程序的步骤。下面给出例子程序，读者可以按照前面介绍的步骤自己进行实验。

```
//func.h 文件
void memset(int *pBuf,int len,int val);
void memcopy(int *in,int *out,int len);
int getMin(int *pBuf,int len);
int getMax(int *pBuf,int len);
void vectorAdd(int *in1,int *in2,int *out,int len);
void vectorSub(int *in1,int *in2,int *out,int len);
void vectorMpy(int *in1,int *in2,int *out,int len);

//func.c 文件
#include "func.h"
void memset(int *pBuf,int len,int val)
{
    int i;
    for(i = 0;i < len;i++)
    {
        pBuf[i] = val;
    }
    return;
}
void memcopy(int *in,int *out,int len)
{
    int i;
    for(i = 0;i < len;i++)
    {
        out[i] = in[i];
    }
    return;
}
int getMin(int *pBuf,int len)
{
    int min = 65555;
    int i;
    int temp;
    for(i = 0;i < len;i++)
    {
        temp = pBuf[i];
        if(temp <= min)
        {
            min = temp;
        }
    }
    return min;
}
int getMax(int *pBuf,int len)
{
    int max = -1;
    int i;
```

```
    int temp;
    for(i = 0;i < len;i++)
    {
        temp = pBuf[i];
        if(temp >= max)
        {
            max = temp;
        }
    }
    return max;
}
void vectorAdd(int *in1,int *in2,int *out,int len)
{
    int i;
    int temp;
    for(i = 0;i < len;i++)
    {
        temp = in1[i] + in2[i];
        out[i] = temp;
    }
}
void vectorSub(int *in1,int *in2,int *out,int len)
{
    int i;
    int temp;
    for(i = 0;i < len;i++)
    {
        temp = in1[i] - in2[i];
        out[i] = temp;
    }
}
void vectorMpy(int *in1,int *in2,int *out,int len)
{
    int i;
    int temp;
    for(i = 0;i < len;i++)
    {
        temp = in1[i] * in2[i];
        out[i] = temp;
    }
}

//main.c 文件
#include <stdio.h>
#include "func.h"

int main(void)
{
```

```
    int a[10];
    int b[10];
    int i;

    int min;
    int max;

    //初始化数组
    memset(a,10,0);
    memset(b,10,0);

    //赋值
    for(i = 0;i < 5;i++)
    {
        a[i] = i;
    }
    for(i = 5;i < 10;i++)
    {
        a[i] = 10 - i;
    }

    //取数组的最大值
    max = getMax(a,10);

    //取数组的最小值
    min = getMin(a,10);

    //赋值
    for(i = 0;i < 10;i++)
    {
        b[i] = min + max;
    }

    //拷贝
    memcopy(b,a,10);

//数组加
    vectorAdd(a,b,a,10);

//数组减
    vectorSub(a,b,a,10);

//数组乘
    vectorMpy(a,b,a,10);
}
```

第一篇

输入显示

第 2 章

4×4 键盘设计

在单片机应用中，有时需要输入配置信息参数。在这些应用中，键盘是一个不可缺少的部分。采用 MSP430 单片机很容易实现矩阵键盘。本章从硬件和软件两个方面介绍 MSP430 单片机的键盘设计。

2.1 键盘电路设计及原理

整个硬件系统主要包括键盘电路、单片机电路和电源电路及复位电路。下面对各个电路进行具体介绍。

2.1.1 键盘电路

矩阵键盘电路主要利用 MSP430 单片机的一般 I/O 口来进行扩展设计。矩阵键盘由行线和列线组成。矩阵键盘通过扫描来实现捕获键盘的输入。所谓扫描就是单片机不断地对行线依次设置低电平，然后检查列线的输入状态，从而确定键盘是否有输入。如图 2-1 所示为键盘的电路设计图。

在图 2-1 中，P1.0、P1.1、P1.2 和 P1.3 分别是键盘的列线。P1.4、P1.5、P1.6 和 P1.7 分别是键盘的行线。列线为输入口，行线为输出口。当往相应的行线上输出低电平，如果键盘中某个键被按下时，则某个列线就为低电平，单片机读取该列线的状态就可以判断哪个键被按下，这就是键盘的扫描原理。

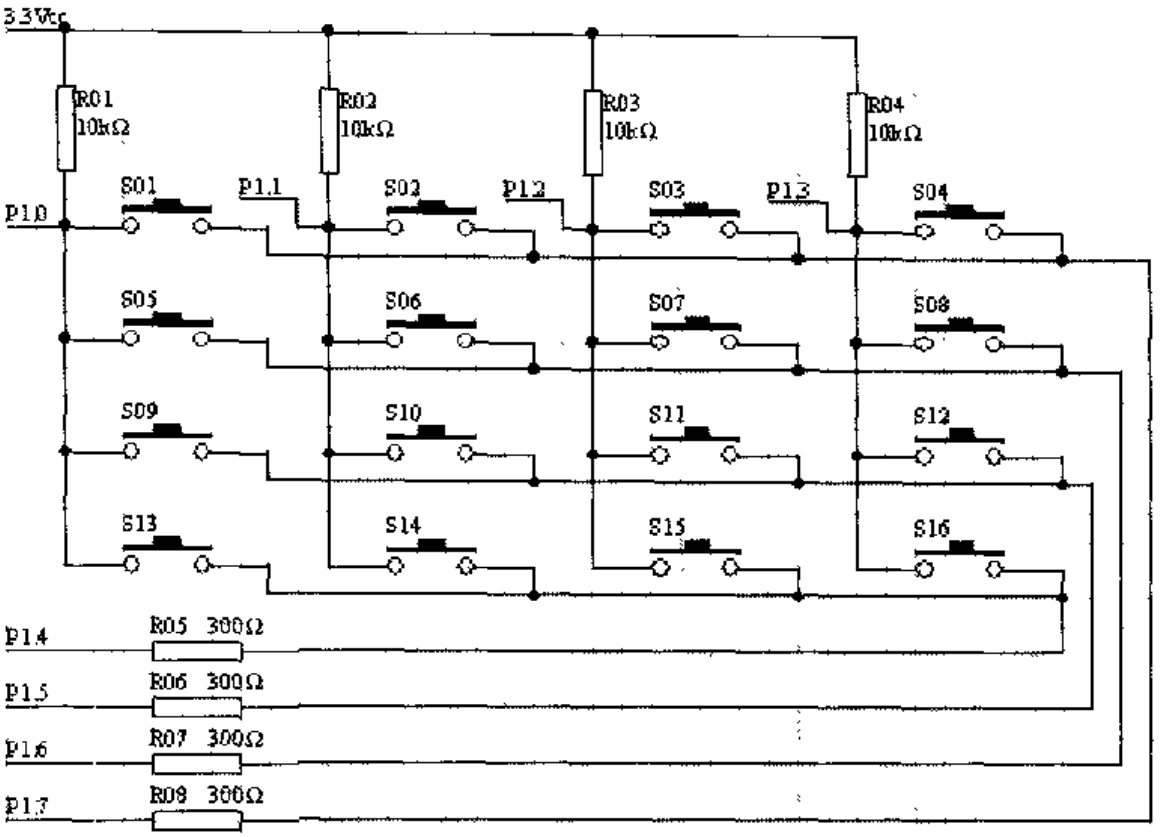


图 2-1　键盘电路

2.1.2　单片机电路

单片机的电路非常简单，单片机的 P1.0、P1.1、P1.2、P1.3、P1.4、P1.5、P1.6 和 P1.7 与键盘电路接口。另外，单片机必须有相应的振荡电路才能使单片机进行工作。如图 2-2 所示为单片机电路图。

现在具体分析键盘的工作原理。由于所有的列线都上拉到 3.3V，所以在没有任何键被按下的时候，所有列线上都是高电平。当在 P1.7 管脚上输出低电平，并且行线的其他管脚上输出高电平时，如果“S01”键被按下，则 P1.0 为低电平；如果“S02”键被按下，则 P1.1 为低电平；如果“S03”键被按下，则 P1.2 为低电平；如果“S04”键被按下，则 P1.3 为低电平。通过设置一条行线的输出就可以获取列线上的相应状态，从而获得键盘输入的值。同理，依次在其他列线上输出低电平，就可以获取其他键的输入值。通过这样的扫描方式，可以实现键盘的输入。由于 MSP430 单片机的 P1 口具有中断功能，因此在软件设计时，可以采用一般 I/O 口来实现键盘输入，也可以利用 P1 口的中断功能来实现键盘输入。

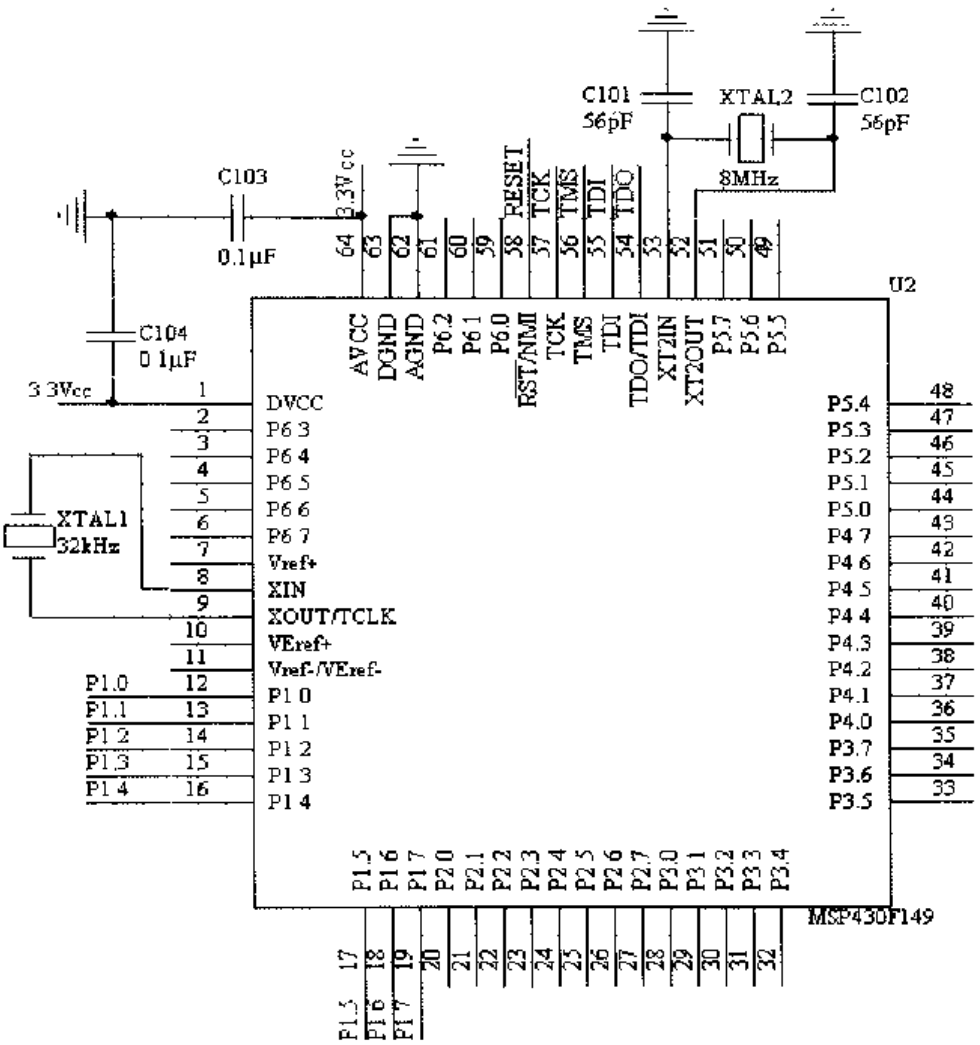


图 2-2　单片机电路

2.1.3　电源电路及复位电路

在单片机应用中必须提供复位信号，以保证单片机能够进行正确的复位，从而进入正确的工作状态。此外，单片机也需要稳定的电压信号，因此必须提供电源电路。如图 2-3 所示为电源电路及复位电路图。

由图 2-3 可以看出，输入的电压经 TPS70633 芯片转换成 3.3V 的电压，以满足单片机的工作电压要求。通过 MAX809STR 产生复位信号送给单片机。为了减小干扰，每个芯片的电源端都加上一个 0.1μF 的电容进行滤波处理。

经过这部分对硬件的介绍，应该对硬件接口电路有了比较清楚的认识，下面介绍整个系统的软件设计。

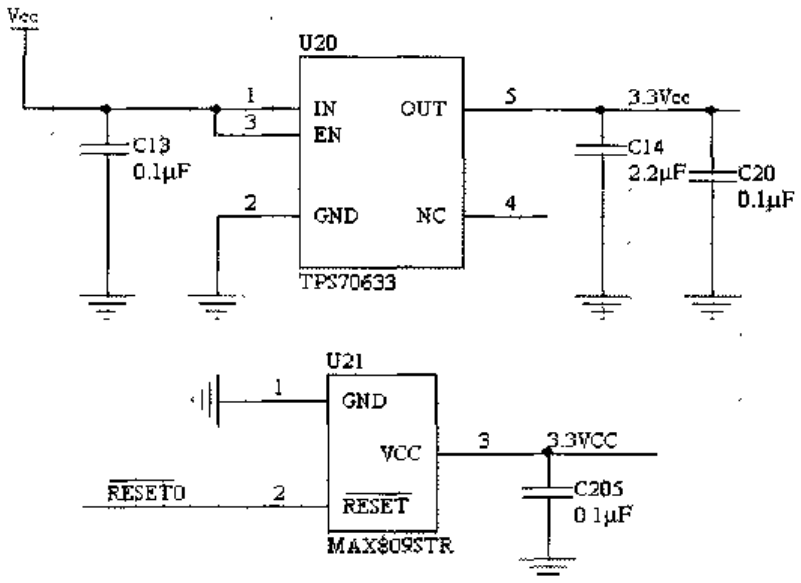


图 2-3　电源电路及复位电路

2.2　一般 I/O 口方式的程序设计

根据上面的原理分析，软件主要是基于扫描实现的。软件通过设置行线上的输出，读取列线上的状态来获取键盘的输入值，整个程序处于键盘扫描状态。在实际的应用中，有时候按键的抖动可能引起误判。比如按了一下键，由于抖动，可能会判断成按了两下键，所以在程序设计的时候必须考虑消除抖动。如图 2-4 所示为程序流程图。

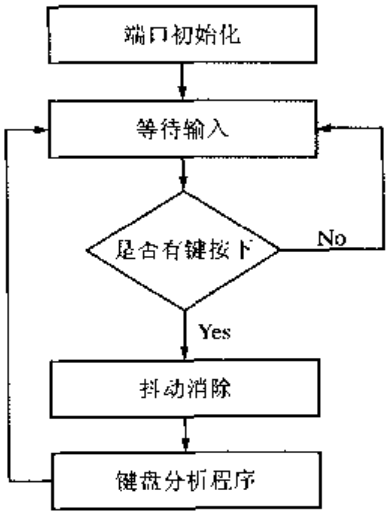


图 2-4　程序流程图

由图 2-4 可以看出，整个程序包括端口初始化、键盘扫描、抖动消除和键盘识别等几

个部分，下面具体分析程序代码。

1. 端口初始化程序

```
void Init_Port(void)
{
    //将 P1 口所有的管脚在初始化的时候设置为输入方式
    P1DIR = 0;
    //将 P1 口所有的管脚设置为一般 I/O 口
    P1SEL = 0;
    // 将 P1.4、P1.5、P1.6、P1.7 设置为输出方向
    P1DIR |= BIT4;
    P1DIR |= BIT5;
    P1DIR |= BIT6;
    P1DIR |= BIT7;
    //先输出低电平
    P1OUT = 0x00;
    return;
}
```

2. 键盘扫描程序

键盘扫描程序主要是等待按键的按下，如果有键按下，则进行按键输入分析。在进行键盘扫描的时候，为了简单起见，在等待键盘输入的时候采用的是死循环等待方式。在实际应用中可以采用其他灵活的方式，具体情况根据系统需求来确定。下面为具体的程序。

```
int KeyScan(void)
{
    int nP10,nP11,nP12,nP13;
    int nRes = 0;
    for(;;)
    {
        //读取各个管脚的状态
        nP10 = P1IN & BIT0;
        nP11 = (P1IN & BIT1) >> 1;
        nP12 = (P1IN & BIT2) >> 2;
        nP13 = (P1IN & BIT3) >> 3;

        //是否有键被按下
        if(nP10 == 0 || nP11 == 0 || nP12 == 0 || nP13 == 0)
        {
            //有键被按下
            break;
        }

    }
```

```
    Delay();  //延迟一点时间，消除抖动
    //读取各个管脚的状态
    nP10 = P1IN & BIT0;
    nP11 = (P1IN & BIT1) >> 1;
    nP12 = (P1IN & BIT2) >> 2;
    nP13 = (P1IN & BIT3) >> 3;

    //是否有键被按下
    if(nP10 == 0 || nP11 == 0 || nP12 == 0 || nP13 == 0)
    {
        //有键被按下，进行键盘输入分析
        nRes = KeyProcess();
    }
    else return -1;  //没有输入，为干扰

    return nRes;
}
```

其中，Delay()为延时程序，用于消除抖动。具体程序如下：

```
void Delay(void)
{
    int i;
    for(i = 100;i--;i > 0) ;  //延迟一点时间
}
```

3. 按键分析程序

在上面的程序中，如果有按键被按下时，则调用按键分析程序进行处理。按键分析程序如下：

```
int KeyProcess(void)
{
    int nRes = 0;
    //P1.4 输出低电平
    P1OUT &= ~(BIT4);
    nP10 = P1IN & BIT0;
    if (nP10 == 0) nRes = 13;
    nP11 = (P1IN & BIT1) >> 1;
    if (nP11 == 0) nRes = 14;
    nP12 = (P1IN & BIT2) >> 2;
    if (nP12 == 0) nRes = 15;
    nP13 = (P1IN & BIT3) >> 3;
    if (nP13 == 0) nRes = 16;

    //P1.5 输出低电平
    P1OUT &= ~(BIT4);
```

```
    nP10 = P1IN & BIT0;
    if (nP10 == 0) nRes = 9;
    nP11 = (P1IN & BIT1) >> 1;
    if (nP11 == 0) nRes = 10;
    nP12 = (P1IN & BIT2) >> 2;
    if (nP12 == 0) nRes = 11;
    nP13 = (P1IN & BIT3) >> 3;
    if (nP13 == 0) nRes = 12;

    //P1.6输出低电平
    P1OUT &= ~(BIT4);
    nP10 = P1IN & BIT0;
    if (nP10 == 0) nRes = 5;
    nP11 = (P1IN & BIT1) >> 1;
    if (nP11 == 0) nRes = 6;
    nP12 = (P1IN & BIT2) >> 2;
    if (nP12 == 0) nRes = 7;
    nP13 = (P1IN & BIT3) >> 3;
    if (nP13 == 0) nRes = 8;
    //P1.7输出低电平
    P1OUT &= ~(BIT4);
    nP10 = P1IN & BIT0;
    if (nP10 == 0) nRes = 1;
    nP11 = (P1IN & BIT1) >> 1;
    if (nP11 == 0) nRes = 2;
    nP12 = (P1IN & BIT2) >> 2;
    if (nP12 == 0) nRes = 3;
    nP13 = (P1IN & BIT3) >> 3;
    if (nP13 == 0) nRes = 4;

    P1OUT = 0x00;  //恢复以前值

    //读取各个管脚的状态
    nP10 = P1IN & BIT0;
    nP11 = (P1IN & BIT1) >> 1;
    nP12 = (P1IN & BIT2) >> 2;
    nP13 = (P1IN & BIT3) >> 3;
    for(;;)
    {
        if(nP10 == 1 && nP11 == 1 && nP12 == 1 && nP13 == 1)
        {
            //等待松开按键
            break;
        }
    }
    return nRes;
}
```

上面的程序通过循环扫描的方式来判断哪个按键被按下，并且等待按键被松开时才返回按键值。在上面的程序中，按键的输入编码只是简单地进行编码处理，在实际中，需要根据具体情况进行编码。

2.3　中断功能方式的程序设计

由于 MSP430 单片机的 P1 口有中断功能，因此可以采用中断的方式进行软件设计。采用中断方式实现的软件不需要扫描处理。另外，在端口的初始化时也有所不同。按键分析程序则与上面的程序完全一致。下面给出端口初始化的具体程序。

```
void Init_Port(void)
{
    //将 P1 口所有的管脚在初始化的时候设置为输入方式
    P1DIR = 0;
    //将 P1 口所有的管脚设置为一般 I/O 口
    P1SEL = 0;
    // 将 P1.4、P1.5、P1.6 、P1.7 设置为输出方向
    P1DIR |= BIT4;
    P1DIR |= BIT5;
    P1DIR |= BIT6;
    P1DIR |= BIT7;
    //先输出低电平
    P1OUT = 0x00;
    // 将中断寄存器清零
    P1IE = 0;
    P1IES = 0;
    P1IFG = 0;
    //打开管脚的中断功能
    //对应的管脚由高到低电平跳变使相应的标志置位
    P1IE |= BIT0;
    P1IES |= BIT0;
    P1IE |= BIT1;
    P1IES |= BIT1;
    P1IE |= BIT2;
    P1IES |= BIT2;
    P1IE |= BIT3;
    P1IES |= BIT3;
    _EINT();  //打开中断
    return;
}
```

上面的程序中增加了中断设置，并设置成低电平触发中断方式。由于使用中断功能，因此必须打开全局中断使能位。上面的“_EINT();”就是打开全局中断功能使能位。

键盘的输入处理可以在P1口的中断服务程序中进行处理。下面为中断服务程序。

```
// 处理来自端口 1 的中断
interrupt [PORT1_VECTOR] void PORT_ISR(void)
{
    Delay();
    KeyProcess();
    if(P1IFG & BIT0)
    {
        P1IFG &= ~(BIT0);  // 清除中断标志位
    }
    if(P1IFG & BIT1)
    {
        P1IFG &= ~(BIT1);  // 清除中断标志位
    }
    if(P1IFG & BIT2)
    {
        P1IFG &= ~(BIT2);  // 清除中断标志位
    }
    if(P1IFG & BIT3)
    {
        P1IFG &= ~(BIT3);  // 清除中断标志位
    }
}
```

在上面的程序中，“Delay();”和“KeyProcess();”与前面的程序是一致的。

2.4 实例总结

本章分析了MSP430单片机的键盘设计。通过本章的介绍可以知道，键盘设计相对比较简单。读者可以在上面的设计上进行调整，从而实现不同按键数的键盘。在软件设计的时候，需要考虑消除抖动，从而避免误判。读者也可以根据自己的要求改进硬件或者软件，设计出满足自己系统需求的键盘。

第 3 章

LED 数码管显示电路的设计

在单片机应用中，有时需要显示系统的工作状态或者运行结果等参数，因此显示是一个不可缺少的部分。本章介绍数码管显示的实现。

3.1 LED 显示电路设计

为了节约管脚，采用移位寄存器芯片来实现 LED 的静态显示。下面首先介绍移位寄存器芯片。

3.1.1 74HC595 芯片

74HC595 是一个 8 位的移位寄存器，可以把串行的输入数据并行输出。该芯片工作电压为 2V~6V，能够和 MSP430 单片机直接进行接口。该芯片有 16 个管脚，如图 3-1 所示为芯片的管脚图。

下面简要介绍 74HC595 芯片的各个管脚。

- Vcc：电源管脚。输入电压为 2V~6V。
- GND：接地管脚。
- DIN：串行数据输入管脚。
- OUT：串行数据输出管脚。
- MR：复位管脚。低电平复位。

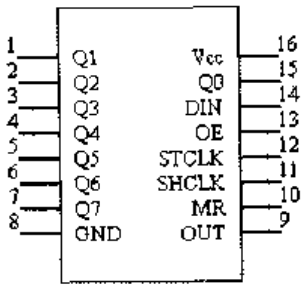


图 3-1　74HC595 管脚图

- OE：输出使能管脚。低电平有效。
- SHCLK：移位时钟信号。
- STCLK：锁存输入信号。
- Q0~Q7：并行输出信号。

3.1.2 LED 数码管

LED 数码管有两种，即共阴数码管和共阳数码管。共阳数码管就是公共管脚接电源，其他管脚与单片机连接，如果单片机在这些管脚上输出低电平，则 LED 就显示相应的值。共阴数码管就是公共管脚接地，其他管脚与单片机连接，如果单片机在这些管脚上输出高电平，则 LED 就显示相应的值。本章是以共阴数码管为基础介绍的。如图 3-2 所示为数码管的管脚示意图。

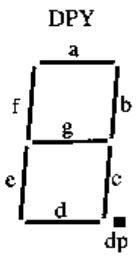


图 3-2　数码管管脚图

由图 3-2 可以看出，数码管主要由不同的段组成，即 a、b、c、d、e、f、g 和 dp 段组成，其中 dp 段用来显示小数点。数码管的每 1 个段都是 1 个发光二极管，对于共阴极二极管来说，a 相当于该段二极管的阳极，而公共管脚为阴极，这样只要在阳极输入高电平，该段就被点亮。只要在不同的管脚输入不同的电平组合，就可以显示相应的数值，比如 a、b、c、d、e、f、g 全部输入高电平，则数码管就显示“8”。

3.1.3 LED 显示电路设计

经过前面对移位寄存器芯片和数码管的介绍，对数码管显示有了基本的概念。下面介绍采用移位寄存器方式设计的数码管显示电路。如图 3-3 所示为数码管显示电路图。

由图 3-3 看出，通过 74HC595 的串行输出管脚实现两个芯片的串行级联，即 U1 的输出作为 U2 的输入，这样可以控制 2 个数码段的显示。U1 作为显示的第一级，它的输入管脚与单片机的 P1.0 进行连接。单片机的 P1.1 和 P1.2 分别产生移位时钟信号和锁存信号，分别与 74HC595 的 SHCLK 和 STCLK 进行连接。74HC595 的并行输出分别与数码管的 a、b、c、d、e、f、g、dp 段连接，实现数据的显示。74HC595 的 OE 接地，使输出使能。基于以上电路设计，就能显示 2 位数字。只要增加 74HC595 就可以增加显示的位数。

基于以上电路设计，数码管显示原理为：单片机通过 P1.0 口串行输出数据到 74HC595，输出完一个字节后，如果给一个锁存信号，则 74HC595 就并行输出，在数码管上显示数

据。单片机可以继续输出显示数据，在输出第二个字节时，第一个 74HC595 将前一个字节的数据通过串行输出管脚输出到第二个 74HC595 的输入管脚，第二个 74HC595 就寄存第一个字节的数据；单片机的第二个数据输出完毕后，两个字节都分别存于两个 74HC595。如果单片机给出锁存信号，则两个 74HC595 通过并行输出将数据分别显示在 2 个二极管上。单片机可以输出一个数据就显示一个数据，也可以输出完要显示的数据后，一次显示所有的数据。

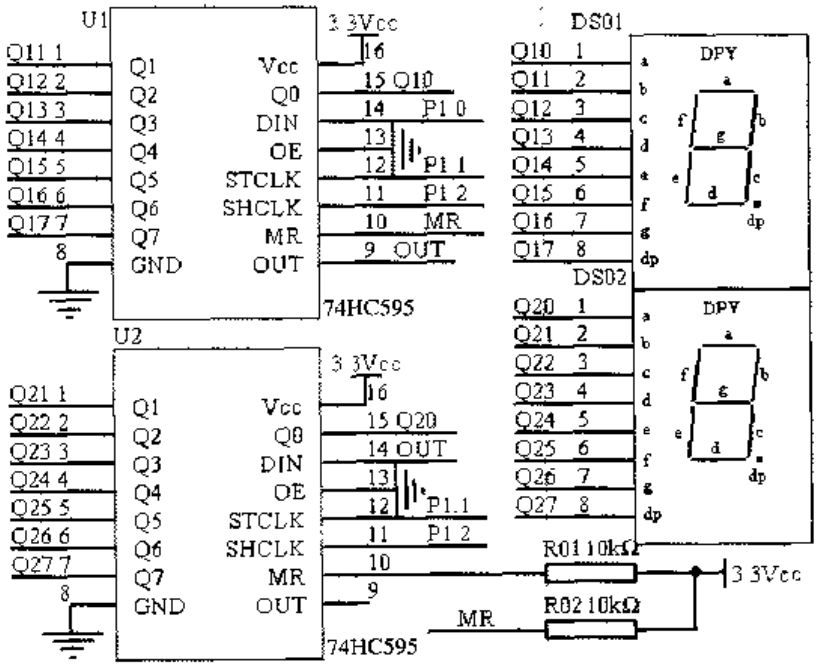


图 3-3　数码管显示接口电路

3.1.4　单片机电路

单片机电路相对比较简单，只需要 3 个一般的 I/O 管脚与 LED 显示电路进行接口就可以了。如图 3-4 所示为单片机电路图。

由图 3-4 可以看出，单片机的电路非常简单，单片机的 P1.0、P1.1 和 P1.2 与显示电路接口。另外，整个电路还需要电源电路和复位电路，具体的电路请参看第 2 章的图 2-3，这里不再介绍。

上面对硬件电路和显示原理进行了介绍，下面分析程序的实现。

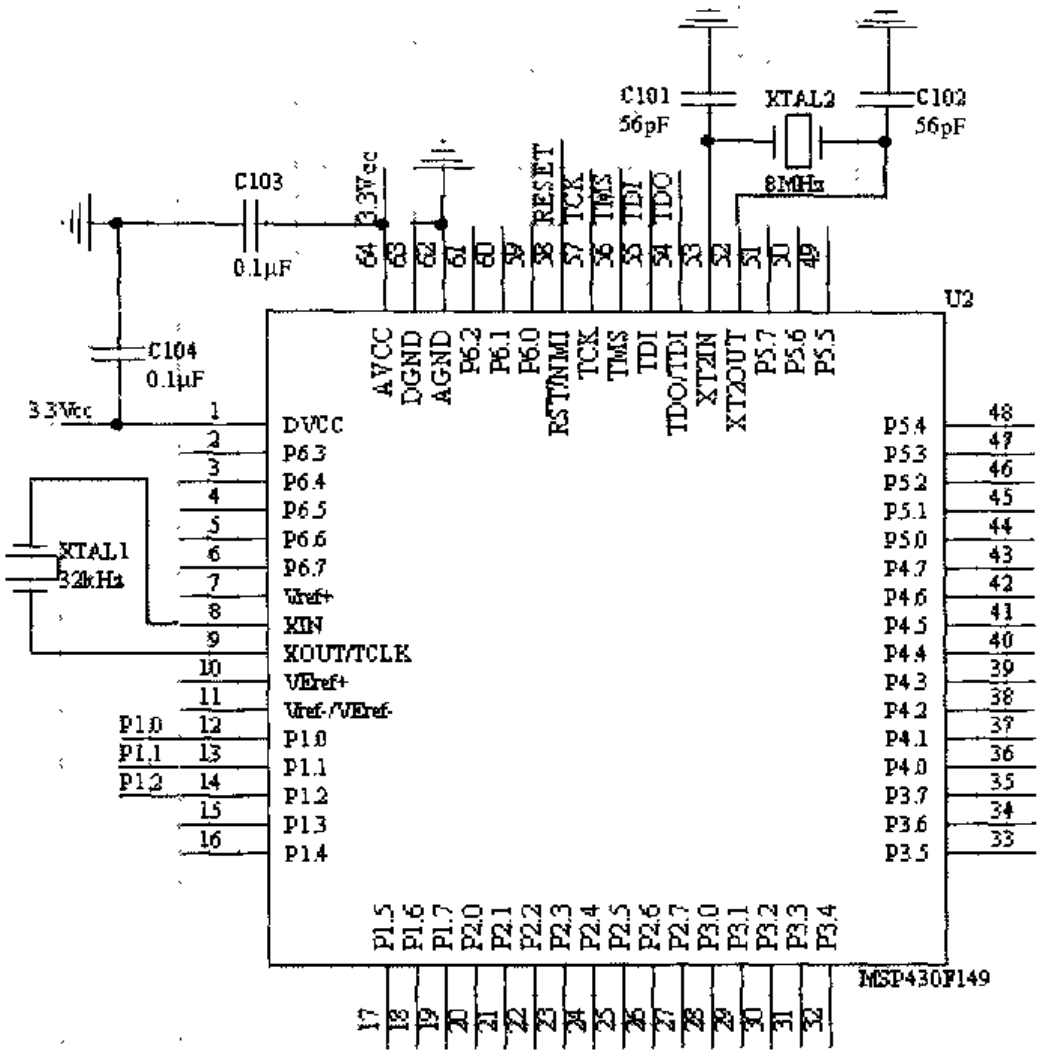


图 3-4　单片机电路

3.2　显示电路的程序设计

显示程序相对简单，就是通过单片机的 P1.2 管脚模拟移位时钟信号。单片机的 P1.0 管脚在输出时钟控制下输出数据，数据是一位一位的输出，单片机的 P1.1 给出锁存信号将数据显示在数码管上。整个显示程序主要由端口初始化、管脚高低电平产生、数据串行输出 3 个部分组成。下面分别对各部分的程序进行分析。

1．端口初始化程序

该部分程序主要是设置管脚的输入输出状态。具体程序如下：

```
void Init_Port(void)
{
    //将 P1 口所有的管脚在初始化的时候设置为输入方式
    P1DIR = 0;

    //将 P1 口所有的管脚设置为一般 I/O 口
    P1SEL = 0;

    // 将 P1.0、P1.1、P1.2 设置为输出方向
    P1DIR |= BIT0;
    P1DIR |= BIT1;
    P1DIR |= BIT2;
    return;
}
```

2. 管脚高低电平产生程序

这部分程序主要模拟时钟信号，具体程序如下：

```
void SHCLK_Hi(void)
{
    //P1.2 管脚输出高电平
    P1OUT |= BIT2;
    return;
}
void SHCLK_Lo(void)
{
    //P1.2 管脚输出低电平
    P1OUT &= ~(BIT2);
    return;
}
void STCLK_Hi(void)
{
    //P1.1 管脚输出高电平
    P1OUT |= BIT1;
    return;
}
void STCLK_Lo(void)
{
    //P1.1 管脚输出低电平
    P1OUT &= ~(BIT1);
    return;
}
```

上面的程序通过在相应的管脚输出低电平或高电平，将高低电平结合使用就产生了相应的时钟信号。

3. 数据显示程序

该程序包括两个部分，即数据的串行输出和数据的显示。下面分别介绍这两个部分的程序。

（1）数据串行输出程序

该程序主要结合 74HC595 的时序串行输出。

```
void DataOut(unsigned char nValue)
{
    int i;
    int j;
    for(i = 0;i < 8;i++)
    {
        if ((nValue & 0x01) == 1)
        {
            P1OUT |= BIT0;          //输出高电平
        }
        else
        {
            P1OUT &= ~(BIT0);       //输出低电平
        }
        SHCLK_Hi();                 //时钟高电平，上升沿有效
        for(j = 10; j > 0; j--) ;   //延迟一点时间
        SHCLK_Lo();                 //时钟低电平
        for(j = 10; j > 0; j--) ;
        nValue >>= 1;
    }
    return;
}
```

这部分程序通过 P1.0 管脚输出数据，在输出数据的时候，P1.2 管脚上需要产生相应的移位时钟信号。

（2）数据显示程序

该部分程序是在前面介绍的函数的基础上实现数据的显示。在前面的 LED 数码管介绍的基础上知道：显示数据就是使相应的段点亮。比如 a、b、c、d、e、f、g 全部输入高电平，则数码管就显示“8”；函数“void DataOut(unsigned char nValue)”的输入参数为“7f”，就在数码管上显示 “8”。这里的“7f”为段码。同理，可以做出其他数字的段码。下面给出段码表。

```
unsigned char seg[]={0x3f,0x06,0x5b,0x4f, /* 0 1 2 3*/
                     0x66,0x6d,0x7d,0x07, /* 4 5 6 7 */
                     0x7f,0x6f,0x77,0x7c, /* 8 9 A B */
```

```
0x39,0x5e,0x79,0x71   /* C D E F */
}
```

根据上面的段码表，如果需要显示某个数值，直接将该值作为下标就可以取出该值的段码，从而实现数据的显示。

下面给出一个显示数据的简单程序。

```
void main(void)
{
    unsigned char nValue;

    //初始化时钟
    Init_CLK();
    //端口初始化
    Init_Port();
    //清除锁存信号
    STCLK_Lo();

    //输出 0
    nValue = 0;
    DataOut(seg[nValue]);
    //输出 2
    nValue = 2;
    DataOut(seg[nValue]);

    //给锁存信号，显示上面的两位数据
    STCLK_Hi();

    return;
}
```

在上面的程序中，“Init_CLK();”为初始化时钟，具体程序如下：

```
void init_CLK(void)
{
    unsigned int i;
    BCSCTL1 = 0X00;     //将寄存器的内容清零
                        //XT2 震荡器开启
                        //LFTX1 工作在低频模式
                        //ACLK 的分频因子为 1

    do
    {
    IFG1 &= ~OFIFG;                          // 清除 OSCFault 标志
    for (i = 0x20; i > 0; i--);
    }
    while ((IFG1 & OFIFG) == OFIFG);     // 如果 OSCFault =1
```

```
    BCSCTL2 = 0X00;              //将寄存器的内容清零
    BCSCTL2 += SELM1;            //MCLK 的时钟源为 TX2CLK，分频因子为 1
    BCSCTL2 += SELS;             //SMCLK 的时钟源为 TX2CLK，分频因子为 1
}
```

3.3 实例总结

本章介绍了数码管的显示。利用移位寄存器方式实现的数码管显示很容易实现硬件和软件的升级。在进行升级时，硬件上不会占用单片机更多的 I/O 管脚，只需要增加 74HC595 和数码管就可以实现更多位的数据显示，并且软件上基本不需要做任何调整。读者可以在本章介绍的基础上适当进行修改，就很容易实现满足自己要求的数码管显示电路和显示程序。

第 4 章

单片机与液晶模块的接口设计与程序

在便携式智能仪器或手持设备等应用中，人机界面是系统中一个非常重要的组成部分。由于液晶显示器（LCD）具有功耗低、体积小、质量轻、超薄等其他显示器无法比拟的优点，因此广泛用于便携式智能仪器或手持设备等产品中。本章介绍 MSP430 单片机与液晶模块的接口设计，并介绍它的程序设计。

4.1 硬件设计

液晶显示器不仅可以显示字符、数字，还可以显示各种图形、曲线及汉字，并且可以实现屏幕上下左右滚动、动画、闪烁、文本特征显示等功能，用途十分广泛。本节介绍 MG-12232 液晶模块与 MSP430 单片机的接口设计，由于液晶模块与单片机的接口是通过液晶驱动芯片与单片机接口实现的。下面首先介绍液晶驱动芯片

4.1.1 驱动芯片

MG-12232 液晶模块是信利公司的产品，MG-12232 模块供电电压的典型值为 3V，工作电流的典型值为 0.3mA，很适合工作电压为 3V 的低功耗环境。该模块可显示范围为 122×32 点阵，即能实现所谓的“双排汉显”。MG-12232 液晶模块采用的驱动芯片是两片 SED1520F0A，每一片 SED1520F0A 控制器可以驱动 16 行×80 列。下面对 SED1520F0A 芯片进行介绍，为了便于硬件电路设计，首先介绍 SED1520F0A 芯片的接口信号。

SED1520F0A 属行列驱动及控制合一的小规模液晶显示驱动芯片，它用来与单片机进行接口。单片机通过该驱动芯片的驱动来实现数据的显示，因此该芯片的管脚信号大致分成两类：单片机接口信号和液晶驱动信号。由于液晶驱动信号与单片机的具体处理无关，用户使用时可以不考虑内部具体是怎样实现的，因此这里只是简单介绍 SED1520F0A 芯片与单片机接口信号。具体信号如下。

- D0~D7：数据总线。
- A0：数据/指令选择信号。当 A0 为 1 时，表示出现在数据总线上的是数据；当 A0 为 0 时，表示出现在数据总线上的是指令或读出的状态。
- RES：接口时序类型选择。当 RES 为 1 时，操作时序为 M6800 时序，其操作信号是 CE 和 R/W；当 RES 为 0 时，操作时序为 Intel8080 时序，操作信号是 RD 和 WR。
- RD（CE）：当操作时序为 Intel8080 时序时，该信号为读，低电平有效；当操作时序为 M6800 时序时，该信号为使能信号。
- WR（R/W）：当操作时序为 Intel8080 时序时，该信号为写信号，低电平有效；当操作时序为 M6800 时序时，该信号为读、写选择信号。当 R/W 为 1 时，为读信号；当 R/W 为 0 时，为写信号。

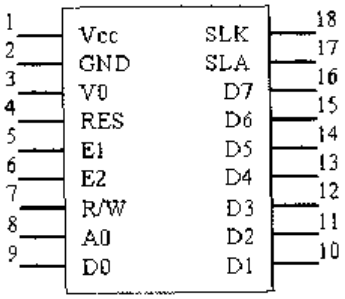


图 4-1　MG-12232 管脚示意图

在了解驱动芯片的接口信号后，下面给出 MG-12232 液晶模块的管脚。如图 4-1 所示。

由图 4-1 可以看出，该模块共有 18 个管脚，下面对具体的管脚进行介绍。

- Vcc：电源管脚。
- GND：接地管脚。
- V0：液晶对比度电压。
- RES：接口时序类型选择管脚。
- E1、E2：使能管脚，分别控制 MG-12232 的两个显示区。
- R/W：读/写控制管脚。
- A0：寄存器选择管脚。
- D0~D7：数据总线。
- SLA、SLK：背光灯管脚。

4.1.2　接口电路设计

由前面的管脚介绍知道，接口电路非常简单。本系统采用 MSP430F149 单片机，单片

机的 P5 口的 P5.0、P5.1、P5.2 和 P5.3 分别与 MG-12232 的 A0、R/W、E1 和 E2 管脚连接。单片机的 P4 口与 MG-12232 的数据总线进行连接。如图 4-2 所示为具体的接口电路图。

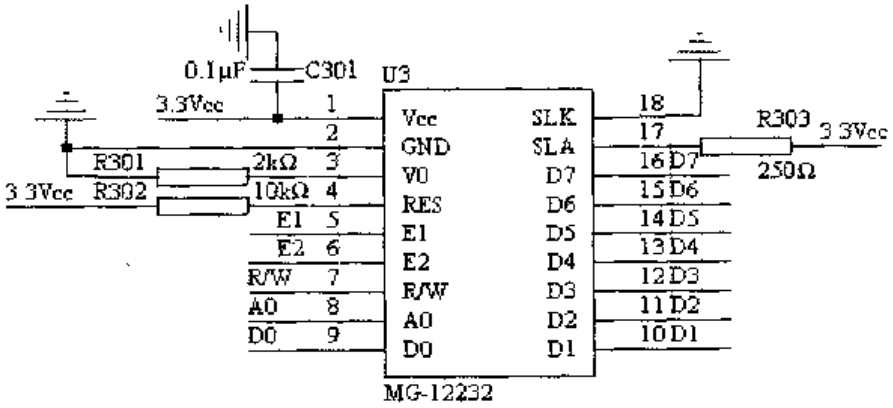


图 4-2　接口电路图

在模块的电源管脚处也放置一个 0.1μF 的滤波电容，以减小干扰。MSP430F149 单片机没有数据总线，在与 MG-12232 模块接口的数据总线接口时，采用软件模拟总线方式实现。由于单片机电路相对比较简单，这里就不再给出具体的电路图了。

4.2　软件设计

软件设计主要是对 MG-12232 模块进行读写操作，从而实现数据的显示。在介绍具体的程序之前，先对 MG-12232 模块的操作进行简单的介绍。

4.2.1　液晶模块操作

MG-12232 模块的显示区域分成 E1 区和 E2 区，每个区包含 4 页，每页有 8 行 62 列。如图 4-3 所示为 MG-12232 模块的显示区域的划分。

E1 区	E2 区
第 2 页	第 2 页
第 3 页	第 3 页
第 0 页	第 0 页
第 1 页	第 1 页

图 4-3　显示区域的划分

单片机是通过操作 MG-12232 模块内部的 RAM 或者寄存器来实现数据的显示，单片机需要向 MG-12232 发送不同的命令来实现具体的操作。表 4-1 中所示为具体的命令。

表 4-1 指令集

指令名称	R/W	A0	D7	D6	D5	D4	D3	D2	D1	D0
显示开/关	0	0	1	0	1	0	1	1	1	1/0
起始行设置	0	0	1	1	0	起始行（0~31）				
设置页地址	0	0	1	0	1	1	0	页地址（0~3）		
设置列地址	0	0	0	列地址（0~79）						
读状态寄存器	1	0	busy	ADC	off/on	reset	0	0	0	0
写数据	0	1	写入的数据							
读数据	1	1	读出的数据							
ADC 选择	0	0	1	0	1	0	0	0	0	0/1
静态驱动开关	0	0	1	0	1	0	0	1	0	0/1
占空比选择	0	0	1	0	1	0	1	0	0	0/1
换行开始	0	0	1	1	1	0	0	0	0	0
换行结束	0	0	1	1	1	0	1	1	1	0
复位	0	0	0	1	0					

通过表 4-1 给出的指令就可以实现对 MG-12232 的相关操作。下面介绍程序的实现。

4.2.2 软件设计

整个程序主要包括单片机初始化、控制管脚电平模拟、液晶模块操作程序等，下面对各个部分进行具体的介绍。

1. 端口初始化程序

端口初始化程序主要初始化 P5 口，下面为具体的初始化程序。

```
void Init_Port(void)
{
    // P5.0、P5.1、P5.2、P5.3 为输出方向
    P5DIR |= BIT0;
    P5DIR |= BIT1;
    P5DIR |= BIT2;
    P5DIR |= BIT3;
    return;
}
```

上面的程序主要是设置 E1、E2、R/W 和 A0 控制管脚的输入输出方向。

2. 控制管脚电平模拟程序

控制管脚电平模拟程序主要是在 E1、E2、R/W 和 A0 控制管脚上产生高电平或者低电平，下面为具体的程序代码。

```
void E1_High(void)
{
    //P5.2 管脚输出为高电平
    P5OUT |= BIT2;
    _NOP();
    _NOP();
    return;
}
void E1_Low(void)
{
    //P5.2 管脚输出为低电平
    P5OUT &= ~(BIT2);
    _NOP();
    _NOP();
    return;
}
void E2_High(void)
{
    //P5.3 管脚输出为高电平
    P5OUT |= BIT3;
    _NOP();
    _NOP();
    return;
}
void E2_Low(void)
{
    //P5.3 管脚输出为低电平
    P5OUT &= ~(BIT3);
    _NOP();
    _NOP();
    return;
}
void A0_High(void)
{
    //P5.0 管脚输出为高电平
    P5OUT |= BIT0;
    _NOP();
    _NOP();
    return;
}
void A0_Low(void)
```

```
{
    //P5.0 管脚输出为低电平
    P5OUT &= ~(BIT0);
    _NOP();
    _NOP();
    return;
}
void R_W_High(void)
{
    //P5.1 管脚输出为高电平
    P5OUT |= BIT1;
    _NOP();
    _NOP();
    return;
}
void R_W_Low(void)
{
    //P5.1 管脚输出为低电平
    P5OUT &= ~(BIT1);
    _NOP();
    _NOP();
    return;
}
```

3. 液晶模块操作程序

液晶模块操作程序主要包括发送命令、显示数据、显示初始化等几个部分。下面给出具体的程序代码。

```
//等待空闲状态
void WaitIdle(int nZone)
{
    char nTemp;
    //设置 P4 口为输入方向
    P4DIR = 0;
    for(;;)
    {
        if(nZone == 0)
        {
            //E1 区
            E1_High();
            E2_Low();
        }
        else
        {
            //E2 区
            E2_High();
```

```
            E1_Low();
        }
        R_W_High();
        A0_Low();
        //取出 D7
        nTemp = (P4IN & BIT7);
        nTemp >>= 7;
        //空闲就跳出循环
        if(nTemp == 0) break;
    }
    return;
}
//发送命令
void SendCommand(int nZone,char nVal)
{
    //等待空闲
    WaitIdle();
    //设置 P4 口为输出方向
    P4DIR = 0xff;
    if(nZone == 0)
    {
        //E1 区
        E1_High();
        E2_Low();
    }
    else
    {
        //E2 区
        E2_High();
        E1_Low();
    }
    R_W_Low();
    A0_Low();
    P4OUT = nVal;
    //写入指令
    if(nZone == 0)
    {
        //E1 区
        E1_Low();
    }
    else
    {
        //E2 区
        E2_Low();
    }
    R_W_High();
    return;
}
```

```
//写显示数据
void WriteData(int nZone,int nVal, char nAddr)
{
    //设置列地址
    SendCommand(nZone,nAddr);
    //等待空闲
    WaitIdle();
    //设置 P4 口为输出方向
    P4DIR = 0xff;
    if(nZone == 0)
    {
        //E1 区
        E1_High();
        E2_Low();
    }
    else
    {
        //E2 区
        E2_High();
        E1_Low();
    }
    R_W_Low();
    A0_High();
    P4OUT = nVal;
    //写入指令
    if(nZone == 0)
    {
        //E1 区
        E1_Low();
    }
    else
    {
        //E2 区
        E2_Low();
    }
    R_W_High();
    return;
}
//打开显示
void DisplayOn(int nZone)
{
    char nTemp;
    for(;;)
    {
        //发送显示开命令
        SendCommand(nZone,0xAF);
        WaitIdle();
        //读状态
```

```
        //设置为输入方向
        P4DIR = 0x00;
        if(nZone == 0)
        {
            //E1区
            E1_High();
            E2_Low();
        }
        else
        {
            //E2区
            E2_High();
            E1_Low();
        }
        R_W_High();
        A0_Low();
        //取出D5
        nTemp = (P4IN & BIT5);
        nTemp >>= 5;
        //如果打开就跳出循环
        if(nTemp == 0) break;
    }
    return;
}
//关闭显示
void DisplayOff(int nZone)
{
    char nTemp;
    for(;;)
    {
        //发送显示开命令
        SendCommand(nZone,0xAE);
        WaitIdle();
        //读状态
        //设置为输入方向
        P4DIR = 0x00;
        if(nZone == 0)
        {
            //E1区
            E1_High();
            E2_Low();
        }
        else
        {
            //E2区
            E2_High();
            E1_Low();
        }
```

```
        R_W_High();
        A0_Low();
        //取出 D5
        nTemp = (P4IN & BIT5);
        nTemp >>= 5;
        //如果打开就跳出循环
        if(nTemp == 1) break;
    }
    return;
}
//清屏操作
void ClearScreen(int nZone)
{
    int i,j;
    char nPageNum;
    //设置页地址代码
    nPageNum = 0xB8;
    for(i = 0;i < 4;i++)
    {
        SendCommand(nCommand);
        for(j = 0;j < 80;j++)
        {
            //等待空闲
            WaitIdle();
            //写入 0x00 以清屏
            WriteData(nZone,0x00,j);
        }
        //页地址增加
        nPageNum += 1;
    }
    return;
}
```

在上面程序的基础上，就基本可以实现数据的显示。在显示数据前，需要对 MG-12232 进行初始化，初始化的顺序为：关显示、正常显示驱动设置、占空比设置、复位、ADC 选择、清屏、开显示。具体程序如下：

```
void LcdDisplayInit(int nZone)
{
    char nCommand;
    //关显示
    DisplayOff(nZone);
    //静态显示驱动
    nCommand = 0xA4;
    SendCommand(nZone,nCommand);
    //占空比设置
    nCommand = 0xA9;
```

```
    SendCommand(nZone,nCommand);
    //复位
    nCommand = 0xE2;
    SendCommand(nZone,nCommand);
    //ADC 选择
    nCommand = 0xA0;
    SendCommand(nZone,nCommand);
    //清屏
    ClearScreen();
    //开显示
    WaitIdle(nZone);
    DisplayOn(nZone);
    return;
}
```

一般说来，汉字的点阵为 16 行×16 列。由于 MG-12232 的每一页只有 8 行，因此显示一个汉字需要在两页上进行显示，下面为具体的程序代码。

```
void LcdDisplayWord(int nZone,char nRow,char nPageNo,char nColAddr,char
nTable[])
{
    int i;
    char nTemp;
    //设置行地址
    SendCommand(nZone,nRow);
    //设置页地址
    SendCommand(nZone,nPageNo);
    //显示前 8 行
    for(i = 0;i < 16;i++)
    {
        nTemp = nTable[i];
        WriteData(nZone,nTemp,i);
    }
    nPageNo += 1;
    //设置页地址
    SendCommand(nZone,nPageNo);
    //显示后 8 行
    for(i = 0;i < 16;i++)
    {
        nTemp = nTable[i + 16];
        WriteData(nZone,nTemp,i);
    }
    return;
}
```

以上给出了液晶模块操作的程序，下面为显示一个“电”字的测试程序，具体程序如下。

```
char WordTable[] = {
    0x00,0xF8,0x48,0x48,0x48,0x48,0xFF,0x48,
    0x48,0x48,0x48,0xFC,0x08,0x00,0x00,0x00,
    0x00,0x07,0x02,0x02,0x02,0x02,0x3F,0x42,
    0x42,0x42,0x42,0x47,0x40,0x70,0x00,0x00
};
void main(void)
{
    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;
    // 关闭中断
    _DINT();

    // 初始化
    Init_CLK();
    Init_Port();

    // 打开中断
    _EINT();
    //初始化MG-12232
    LcdDisplayInit(0);
    LcdDisplayInit(1);

    //显示的起始行为第0行，起始列为第0列
    //起始页为第0页，显示区为两区，内容为“电”
    LcdDisplayWord(0,0xC0,0xB8,0,WordTable);
    return;
}
```

在上面的程序中，直接将“电”字的点阵数据做成一个数组。在实际应用中，可以将点阵数据存放到FLASH里。

4.3 实例总结

本章介绍了MSP430单片机与MG-12232图形点阵液晶模块的接口设计，并详细介绍了程序设计。通过本章介绍的系统，可以很好地实现微功耗的人机界面。虽然本章介绍的系统没有点阵数据模块，但是读者完全可以在本章介绍的硬件基础上增加FLASH模块，用于存放点阵数据，这样就可以实现显示更多汉字或者字符信息的系统，从而满足不同的应用需求。

第二篇

算法实现

第 5 章

MSP430 的 CRC 程序设计实现

在一些数据传输系统中，为了保证传输的可靠性，需要对数据进行差错校验。CRC（Cyclic Redundancy Check，循环冗余校验）是常用的差错校验算法。本章介绍 CRC 的原理，并给出具体的程序设计。

5.1 CRC 的原理与算法

在一些数据传输系统中，由于信道的干扰而经常造成误码的发生。为了保证数据传输的可靠性，需要对数据进行差错校验，而 CRC 是最常用的校验算法。在数据的发送端，计算得到发送时的 CRC 值，并将该值填充到数据帧中的 CRC 字段；在接收端，根据接收到的数据帧计算 CRC 的值，并将计算得到的值与接收到来自发送端的 CRC 值进行比较。如果两个值一致，则数据正确；如果数据不一致，则数据出错。这样，通过采用 CRC 就可以实现差错校验。

5.1.1 CRC 算法的原理

在 CRC 算法中，被处理的数据序列可以看作是二进制多项式。下面的式（5-1）为数据序列的数学表达式。

$$M(X) = a_{n-1}X^{n-1} + a_{n-2}X^{n-2} + a_1X^1 + a_0 \qquad (5\text{-}1)$$

式（5-1）中的 a_{n-1} 为相应阶数的系数。如二进制序列 10101101 就可以表示成下面的

式（5-2）。

$$M(X) = X^7 + X^5 + X^3 + X^2 + 1 \tag{5-2}$$

将欲传输的 k 位数据序列 $M(X)$再增加（$n-k$）位的校验码 $R(X)$，那么在新的数据序列中，新的数据序列多项式就变为 $X^{n-k}M(X)$。假设要传输的数据序列为 $T(X)$，则 $T(X)$为：

$$T(X) = X^{n-k}M(X) + R(X) \tag{5-3}$$

将 $X^{n-k}M(X)$除以预先给定的生成多项式 $G(X)$，得到商 $Q(X)$和余数 $R(X)$，得到式(5-4)。

$$X^{n-k}M(X) / G(X) = Q(X) + R(X)/G(X) \tag{5-4}$$

式（5-4）中的 $R(X)$就是（$n-k$）位校验码。由式（5-3）除以 $G(X)$后，结合式（5-4）得到式（5-5）。

$$T(X)/G(X) = Q(X) + R(X)/G(X) + R(X)/G(X) \tag{5-5}$$

由于在模 2 运算中，$1 + 1 = 0$，所以由式（5-5）可以得到式（5-6）。

$$T(X) / G(X) = Q(X) \tag{5-6}$$

由式（5-6）可以看出：传输的数字序列被生成多项式模 2 相除后应该没有余数，否则表明数据出错。CRC 有不同的生成多项式。表 5-1 列出了几种常用的生成多项式。

表 5-1　CRC 的生成多项式

算法名称	生成多项式
CRC-4	$X^4 + X + 1$
CRC-16	$X^{16} + X^{15} + X^2 + 1$
CRC-CCITT	$X^{16} + X^{15} + X^5 + 1$
CRC-32	$X^{32} + X^{25} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$

由表 5-1 可以看出，CRC 的生成多项式有不同的阶数。CRC 的生成多项式的阶数越高，那么误判的概率就越小。不同的 CRC 生成多项式有不同的应用。CRC-4 主要用于 PCM 的基群设备。CRC-16 主要用在 IBM 的同步数据链路控制规程 SDLC 的帧校验序列 FCS 中。CRC-CCITT 主要用在 CCITT 推荐的高级数据链路控制规程 HDLC 的帧校验序列 FCS 中。CRC-32 主要用于重要数据的传输，在通信、计算机等领域运用十分广泛。以太网卡芯片、MPEG 解码芯片等芯片中，常采用 CRC-32 进行差错控制。

5.1.2　CRC 算法的实现

CRC 算法主要有位运算算法和查表法两种实现方法。下面对这两种方法分别进行介绍。

1．位运算算法

位运算算法就是二进制多项式的除法运算，在做除法运算的时候保留余数，丢弃商。

位运算算法可以采用移位寄存器来实现，如图 5-1 所示为 CRC 位运算算法的移位寄存器实现的示意图。

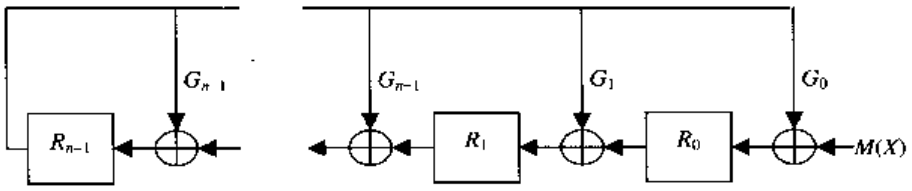


图 5-1　CRC-n 比特运算的移位寄存器实现图

由图 5-1 可以看出，传输的数据序列与生成多项式进行模 2 的除法运算，最后移位寄存器剩下的内容就是 CRC 校验码。根据上面的移位寄存器系统，位运算算法实现的步骤为：

① 初始化 CRC 寄存器。

② 在移入下一个数据位时，将 CRC 的位左移 1 位。

③ 如果被移出的位为 1，则将 CRC 与生成多项式进行异或（XOR）运算。

④ 继续进行第②步，直到数据序列都被移入。

⑤ 将 CRC 里面的内容与最终的异或值进行异或运算。

以上给出了 CRC 算法实现的步骤，但由于每次处理 1 位，所以运算量比较大。下面介绍查表法。

2. 查表法

在某些运算量比较大的情况下，可以事先生成一张表，这样在实际处理的时候不需要进行计算，只需要从表里的相对位置取出数据就可以了，这种方法很适合某些处理能力不强的单片机。另外，查表法主要针对字节或字进行处理。使用查表法的优势是事先将 CRC 的值计算出来做成一张表，当进行实际的 CRC 处理时，只是简单地从表里查出相应的值，这样运算量非常小。该表可以存储在 MSP430 单片机的 FLASH 里。下面给出查表法的实现步骤。

① 初始化 CRC 寄存器。

② 将数据序列与 CRC 寄存器的高字节进行异或运算。

③ 用该字节去索引 CRC 表。

④ 将 CRC 寄存器左移一个字节。

⑤ 将 CRC 寄存器的值与查表得到的值进行异或运算。

⑥ 继续进行第②步，直到数据序列都被移入。

⑦ 将 CRC 里面的内容与最终的异或值进行异或运算。

经过上面对 CRC 的原理介绍和算法实现的分析，对 CRC 有了比较清楚的认识，下面介绍具体的程序实现。

5.2　CRC 的程序实现

上面介绍了 CRC 的原理及算法实现，本节主要介绍 CRC 的程序实现。由前面的介绍知道，CRC 主要由位运算实现和查表实现。下面分别介绍相应的程序实现。

5.2.1　位运算算法的程序实现

位运算算法主要是移位操作和异或操作。按照前面介绍的算法步骤，下面给出具体的程序。

```
unsigned short crc16Bitwise( unsigned short crc,
                             unsigned short poly,
                             unsigned char *pmsg,
                             unsigned int msg_size)
{
    unsigned int i, j, carry;
    unsigned char msg;
    unsigned short temp;

    temp = *pmsg++ << 8;
    temp |= *pmsg++;
    crc ^= temp;

    for(i = 0 ; i < msg_size ; i ++)
    {
        msg = *pmsg++;

        for(j = 0 ; j < 8 ; j++)
        {
            carry = crc & 0x8000;
            crc = (crc << 1) | (msg >> 7);
            if(carry) crc ^= poly;
            msg <<= 1;
        }
    }

    return(crc ^ CRC16_FINAL_XOR);
}
```

上面的程序为 CRC-16 的位运算算法代码。在上面程序的输入参数中，“crc”为初始值，“poly”为生成多项式的值，“*pmsg”为数据序列，“msg_size”为数据序列的长度。

上面的程序也可以演变为 CRC-32 的算法程序，具体程序如下。

```
unsigned long crc32Bitwise( unsigned long crc,
                            unsigned long poly,
                            unsigned char *pmsg,
                            unsigned int msg_size)
{
    unsigned int i, j, carry;
    unsigned char msg;
    unsigned long temp;

    temp = (unsigned long)(*pmsg++) << 24;
    temp |= (unsigned long)(*pmsg++) << 16;
    temp |= (unsigned long)(*pmsg++) << 8;
    temp |= (unsigned long)(*pmsg++);
    crc ^= temp;

    for(i = 0 ; i < msg_size ; i ++)
    {
        msg = *pmsg++;

        for(j = 0 ; j < 8 ; j++)
        {
            carry = crc >> 31;
            crc = (crc << 1)   (msg >> 7);
            if(carry) crc ^= poly;
            msg <<= 1;
        }
    }

    return(crc ^ CRC32_FINAL_XOR);
}
```

上面给出了 CRC-16 和 CRC-32 的算法程序，读者可以根据上面的程序写出 CRC-4 的程序，这里限于篇幅就不再介绍了。

考虑到单片机的处理能力不强，所以需要对算法进行优化。下面给出 CRC-16 算法的优化程序。

```
unsigned short crc16Bitwise2( unsigned short crc,
                              unsigned short poly,
                              unsigned char *pmsg,
                              unsigned int msg_size)
{
    unsigned int i, j;
```

```
    unsigned short msg;

    for(i = 0 ; i < msg_size ; i ++)
    {
        msg = (*pmsg++ << 8);

        for(j = 0 ; j < 8 ; j++)
        {
            if((msg ^ crc) >> 15) crc = (crc << 1) ^ poly;
            else crc <<= 1;

            msg <<= 1;
        }
    }

    return(crc ^ CRC16R_FINAL_XOR);
}
```

由于位运算算法程序是一位一位进行处理的，这样运算量比较大。下面介绍查表法程序。

5.2.2　查表法的程序实现

查表法程序主要包括表的生成程序和查表程序。首先看表的生成程序。

```
void crc16BuildTable(unsigned short *ptable, unsigned short poly)
{
    unsigned int i, j;

    for(i = 0; i <= 255; i++)
    {
        ptable[i] = i << 8;
        for(j = 0; j < 8; j++)
            ptable[i] = (ptable[i] << 1) ^ (ptable[i] & 0x8000 ? poly : 0);
    }
}
```

上面的程序为 CRC-16 的表的生成程序，在上面的程序中需要传入生成多项式的值“poly”。CRC-32 算法也有相应的表生成程序，具体如下。

```
void crc32BuildTable(unsigned long *ptable, unsigned long poly)
{
    unsigned int i, j;

    for(i = 0; i <= 255; i++)
    {
```

```
            ptable[i] = (long)i << 24;
            for(j = 0; j < 8; j++)
                ptable[i] = (ptable[i] << 1) ^ (ptable[i] & 0x80000000 ? poly : 0);
    }
}
```

通过上面的程序就可以生成相应的表。比如 CRC-16 的表如下。

```
unsigned short table16[] = {0x0000, 0x8005, 0x800F, 0x000A,
    0x801B, 0x001E, 0x0014, 0x8011,
    0x8033, 0x0036, 0x003C, 0x8039,
    0x0028, 0x802D, 0x8027, 0x0022,
    0x8063, 0x0066, 0x006C, 0x8069,
    0x0078, 0x807D, 0x8077, 0x0072,
    0x0050, 0x8055, 0x805F, 0x005A,
    0x804B, 0x004E, 0x0044, 0x8041,
    0x80C3, 0x00C6, 0x00CC, 0x80C9,
    0x00D8, 0x80DD, 0x80D7, 0x00D2,
    0x00F0, 0x80F5, 0x80FF, 0x00FA,
    0x80EB, 0x00EE, 0x00E4, 0x80E1,
    0x00A0, 0x80A5, 0x80AF, 0x00AA,
    0x80BB, 0x00BE, 0x00B4, 0x80B1,
    0x8093, 0x0096, 0x009C, 0x8099,
    0x0088, 0x808D, 0x8087, 0x0082,
    0x8183, 0x0186, 0x018C, 0x8189,
    0x0198, 0x819D, 0x8197, 0x0192,
    0x01B0, 0x81B5, 0x81BF, 0x01BA,
    0x81AB, 0x01AE, 0x01A4, 0x81A1,
    0x01E0, 0x81E5, 0x81EF, 0x01EA,
    0x81FB, 0x01FE, 0x01F4, 0x81F1,
    0x81D3, 0x01D6, 0x01DC, 0x81D9,
    0x01C8, 0x81CD, 0x81C7, 0x01C2,
    0x0140, 0x8145, 0x814F, 0x014A,
    0x815B, 0x015E, 0x0154, 0x8151,
    0x8173, 0x0176, 0x017C, 0x8179,
```

```
0x0168, 0x816D, 0x8167, 0x0162,
0x8123, 0x0126, 0x012C, 0x8129,
0x0138, 0x813D, 0x8137, 0x0132,
0x0110, 0x8115, 0x811F, 0x011A,
0x810B, 0x010E, 0x0104, 0x8101,
0x8303, 0x0306, 0x030C, 0x8309,
0x0318, 0x831D, 0x8317, 0x0312,
0x0330, 0x8335, 0x833F, 0x033A,
0x832B, 0x032E, 0x0324, 0x8321,
0x0360, 0x8365, 0x836F, 0x036A,
0x837B, 0x037E, 0x0374, 0x8371,
0x8353, 0x0356, 0x035C, 0x8359,
0x0348, 0x834D, 0x8347, 0x0342,
0x03C0, 0x83C5, 0x83CF, 0x03CA,
0x83DB, 0x03DE, 0x03D4, 0x83D1,
0x83F3, 0x03F6, 0x03FC, 0x83F9,
0x03E8, 0x83ED, 0x83E7, 0x03E2,
0x83A3, 0x03A6, 0x03AC, 0x83A9,
0x03B8, 0x83BD, 0x83B7, 0x03B2,
0x0390, 0x8395, 0x839F, 0x039A,
0x838B, 0x038E, 0x0384, 0x8381,
0x0280, 0x8285, 0x828F, 0x028A,
0x829B, 0x029E, 0x0294, 0x8291,
0x82B3, 0x02B6, 0x02BC, 0x82B9,
0x02A8, 0x82AD, 0x82A7, 0x02A2,
0x82E3, 0x02E6, 0x02EC, 0x82E9,
0x02F8, 0x82FD, 0x82F7, 0x02F2,
0x02D0, 0x82D5, 0x82DF, 0x02DA,
0x82CB, 0x02CE, 0x02C4, 0x82C1,
0x8243, 0x0246, 0x024C, 0x8249,
0x0258, 0x825D, 0x8257, 0x0252,
0x0270, 0x8275, 0x827F, 0x027A,
```

```
    0x826B, 0x026E, 0x0264, 0x8261,
    0x0220, 0x8225, 0x822F, 0x022A,
    0x823B, 0x023E, 0x0234, 0x8231,
    0x8213, 0x0216, 0x021C, 0x8219,
    0x0208, 0x820D, 0x8207, 0x0202};
```

由于事先生成了相应的表，则查表法在处理时不需要计算 CRC，而只是简单地查找表，计算量就小得多。下面给出具体的查表程序。

```
unsigned short crc16TableMethod(unsigned short crc,
                                unsigned short *table,
                                unsigned char *pbuffer,
                                unsigned int length)
{
    while(length--)
        crc = table[((crc >> 8) ^ *pbuffer++)] ^ (crc << 8);

    return(crc ^ CRC16_FINAL_XOR);
}
```

通过上面的程序可以看出，查表法计算量很小。下面给出 CRC-32 的查表程序。

```
unsigned long crc32TableMethod( unsigned long crc,
                                unsigned long *table,
                                unsigned char *pbuffer,
                                unsigned int length)
{
    while(length--)
        crc = table[((crc >> 24) ^ *pbuffer++)] ^ (crc << 8);

    return(crc ^ CRC32_FINAL_XOR);
}
```

以上给出了 CRC 的位运算算法程序和查表法算法程序。读者可以根据系统的需要选择适合自己的算法。

5.3 实例总结

本章介绍了 CRC 算法的原理和算法实现，并给出了相应的程序实现。在本章的基础上，读者可以实现 CRC-4 算法。通过本章的程序介绍，读者也可以将算法应用到自己的数据传输系统中，保证数据传输的可靠性。

第 6 章

基于单片机的中文输入法程序的实现

在某些嵌入式应用系统（如智能终端）中，人机交互界面是不可缺少的组成部分，而某些系统要求能够实现中文输入。本章介绍采用 MSP430 单片机实现的中文输入法程序。

6.1 实现原理

一般说来，在嵌入式应用系统中可以通过键盘输入字符串。如果输入的字符串是合法的拼音串的话，那么可以根据该拼音串获得汉字的内码，由内码可以计算得到汉字的点阵的起始位置，取出点阵数据就可以在 LCD 上显示该汉字。由于用户输入的字符串是任意的，即不一定是合法的拼音，因此需要对用户输入的字符串进行判断。汉语中的拼音串比较多，因此需要考虑数据组织的方法，以便能够快速判别。汉语的拼音由声母和韵母组成，它的组成有一定的规律性，因此可以将所有的拼音串做成一张表。在处理的时候，只需要判断输入的字符串是否在拼音表里面，如果输入的字符串不在拼音表里，那么该字符串不是正确的拼音。

通过查询拼音表得到该拼音所对应的第一个汉字的起始位置 1，然后再查找与该拼音串相邻的下一个拼音串对应的第一个汉字的起始位置 2，则位置 2 与位置 1 之间的汉字就是输入拼音所对应的所有汉字。

为了实现中文输入显示，需要设计 3 张表，即拼音表、内码表和点阵数据表。下面分别介绍各个表的设计。

1. 拼音表

拼音表主要存储拼音数据和该拼音所对应的第一个汉字内码起始位置数据。如图 6-1 所示为拼音表的存储示意图。

拼音串	汉字内码起始位置
拼音数据（6 字节）	相对地址数据（2 字节）

图 6-1　拼音表存储示意图

由图 6-1 可以看出，拼音表由一条一条的拼音记录组成，每条拼音记录由 8 个字节组成。其中拼音串占 6 个字节，如果拼音串没有 6 个字节，则后面填 0 补足。拼音记录里还包含 2 个字节的地址数据，该数据表示该拼音所对应第一个汉字的内码的相对起始位置。例如：拼音“a”在拼音表里的数据为“0x61，0x00，0x00，0x00，0x00，0x00，0x00，0x00”这种形式，同样拼音“ai”在拼音表里的数据为“0x61，0x69，0x00，0x00，0x00，0x00，0x00，0x05”这种形式。拼音表里面的数据是按照拼音排序的。

2. 内码表

内码表主要存储按照拼音排序的汉字的内码。汉字的内码占两个字节，在表中，以汉字内码为单位连续存放。

3. 点阵数据表

点阵数据表存放的是汉字点阵数据。该表存放的是汉字的 16×16 点阵数据，每个汉字的点阵数据占 32 个字节，在该表中，以汉字的点阵数据为单位连续存放。

利用上述 3 张表就很容易实现中文拼音的输入。首先通过键盘输入拼音，然后使用输入的拼音查找拼音表，以得到该拼音所对应的第一个汉字的内码起始位置 1，再从拼音表中取出输入拼音的下一个拼音，并得到它所对应的第一个汉字的内码起始位置 2。则起始位置 2 和起始位置 1 之间的内码就是输入拼音所对应的所有汉字的内码。由汉字的内码就可以计算得到汉字点阵数据的起始位置，根据汉字点阵数据的起始位置取出点阵数据就可以实现显示。根据汉字内码计算汉字点阵的相对起始位置的公式为：相对起始位置=（（内码高字节–0xB0）×94+（内码低字节–0xA1））×32。

6.2　软件设计

该系统的拼音表数据、内码表数据和点阵表数据都是存放在存储器里。整个系统的软

件包括存储器操作、汉字内码获得、点阵数据获取等几部分。由于存储器操作不是本章介绍的重点，因此本章暂时使用 SM 卡作为存储器存放数据表，对存储器的操作本章不做介绍。下面具体介绍汉字内码获得和点阵数据获取两部分。

6.2.1　汉字内码获得

由输入的拼音查询拼音表就可以获得该拼音所对应的第一个汉字的内码的位置。根据输入拼音和它的下一个拼音，可以获得输入拼音所对应的所有汉字的各自内码。下面为具体的程序代码。

```
int GetNeiMaAddr(char *PY,int PY_Len,int *StartNeiMa,int *EndNeiMa)
{
    char buf[6];
    char buf_temp[8];
    int i;
    int j;
    int PYCount;
    int PageCount;
    char buf_sm[528];
    int nStart;
    int nEnd;

    for(i = 0;i < 6;i++)
    {
        buf[i] = '\0';
        buf_temp[i] = '\0';
    }
    //错误拼音输入
    if((PY_Len > 6) && (PY_Len <= 0))
    {
        return -1;
    }
    for(i = 0;i < PY_Len;i++)
    {
        buf[i] = PY[i];
    }
    //错误拼音输入
    if(buf[0] == 'i')
    {
        return -1;
    }
    if(buf[0] == 'u')
    {
        return -1;
```

```
}
if(buf[0] == 'v')
{
    return -i;
}

//搜索表
PYCount = 0;
PageCount = 0;
nStart = 0;
nEnd = 0;
while(PYCount < PY_NUM)
{
    if(PageCount >= 8)
    {
        break;
    }
    //读出一页数据
    PageRead(0,PageCount + PYTABLE,buf_sm);
    for(i = 0;i < 64;i++)
    {
        for(j = 0;j < 8;j++)
        {
            buf_temp[j] = buf_sm[i * 8 + j];
        }
        //比较
        for(j = 0;j < PY_Len;j++)
        {
            if(buf[j] != buf_temp[j])
            {
                break;
            }
        }
        //相等
        if(j == PY_Len - 1)
        {
            //得到开始地址
            nStart = buf_temp[6] * 256 + buf_temp[7];
            //取下一个拼音串
            if(i == 63)
            {
                //该页的末尾，读下一页数据
                PageRead(0,PageCount + 1 + PYTABLE,buf_sm);
                for(j = 0;j < 8;j++)
                {
                    buf_temp[j] = buf_sm[j];
                }
                nEnd = buf_temp[6] * 256 + buf_temp[7];
```

```
                }
                else
                {
                    //不是负的末尾数据
                    for(j = 0; j < 8;j++)
                    {
                        buf_temp[j] = buf_sm[(i + 1) * 8 + j];
                    }
                    nEnd = buf_temp[6] * 256 + buf_temp[7];
                }
            }
            else
            {
                //没找到，继续搜索
                PYCount++;
            }
        }
        PageCount++;
    }
    //没有搜索到数据
    if(PageCount >= 8)
    {
        return -1;
    }
    *StartNeiMa = nStart;
    *EndNeiMa = nEnd;
    return 1;
}
```

通过上面的程序可以得到输入拼音所对应的所有汉字的起始位置和结束位置。通过这两个位置搜索内码表就可以得到输入拼音所对应的所有汉字的内码。内码表搜索程序比较简单，这里就不做介绍了。

6.2.2　点阵数据获取

根据汉字的内码就可以计算得到汉字的点阵数据，下面为具体的程序代码。

```
void GetDianZhen(int NeiMa,char *DianZhen)
{
    int i;
    int nTemp;
    int nTmp;
    long nRow;
    char chrHi;
    char chrLo;
    long offset;
```

```
chrHi = NeiMa / 256 - 0xB0;
chrLo = NeiMa % 256 - 0xA1;
offset = (chrHi * 94 + chrLo) * 32;
offset = offset + DIANZHENTABLE;
nRow = offset / 512;
nTemp = offset % 512;
nTmp = nTemp + 32;
//在 SM 卡的第 1 区
if(nTmp <= 255)
{
    for(i = 0;i < 32;i++)
    {
        DianZhen[i] = ReadByte(0,nTemp + i,nRow);
    }
}
else if((nTmp > 255) && (nTmp < 512))
{
    //在 SM 卡的第 1 区
    if(nTemp < 256)
    {
        for(i = 0;i < 32;i++)
        {
            DianZhen[i] = ReadByte(0,nTemp + i,nRow);
        }
    }
    else
    {
        //在 SM 卡的第 2 区
        for(i = nTemp;i < 256;i++)
        {
            DianZhen[i] = ReadByte(0,i,nRow);
        }
        for(i = 0;i < 32 - (256 - nTemp);i++)
        {
            DianZhen[i + 256 - nTemp] = ReadByte(1,i,nRow);
        }
    }
}
else
{
    //不在同一页
    for(i = nTemp;i < 512;i++)
    {
        DianZhen[i] = ReadByte(1,i,nRow);
    }
    for(i = 0;i < 32 - (512 - nTemp);i++)
    {
```

```
                DianZhen[i + 512 - nTemp] = ReadByte(1,i,nRow + 1);
            }
        }
        return;
    }
```

上面的程序根据汉字的内码获得汉字的点阵数据。由于上面的程序是基于 SM 卡作为存储器，因此读者在实现自己系统时，程序可能不同，具体实现需要由读者设计的具体系统来确定。

6.3　实例总结

本章介绍了采用 MSP430 单片机实现的中文输入法程序。输入法程序首先根据输入的拼音获得汉字的内码，在获得汉字的内码后，由获得的汉字内码获得汉字的点阵数据，最后就可以将汉字点阵数据显示在液晶屏上。本章介绍的方法主要是通过查找相应的表来获得汉字内码和汉字点阵数据，读者也可以选用其他的实现方式来实现。另外，读者可以改进搜索算法，从而提高处理速度。

第 7 章

基于单片机的数据压缩算法的实现

在单片机实现的传输系统中，有时候传输的带宽有限。对于传输数据量比较大的场合，可以对数据进行压缩，压缩后的数据比未压缩的数据在量上小得多，可以实现更加有效的传输。数据压缩也可以用于其他领域，比如数据存储系统。本章介绍采用 MSP430 单片机实现的数据压缩算法。

7.1 压缩算法原理

实现数据压缩的算法有很多，本章介绍无损的数据压缩算法：Huffman 算法。下面对 Huffman 算法进行详细的介绍。

7.1.1 Huffman 算法原理

对于需要处理的信息信源来说，将信源中一定位长的值看作是符号，比如把 8 位长的 256 种值，也就是字节的 256 种值看作是符号。对处理信息中的符号进行统计，每个符号出现的频率不一样，根据这些符号在信源中出现的频率，对这些符号重新编码。在进行编码时，不是采用定长编码，即每个符号不是都用相同长度的位数进行编码，而是采用变长编码。对于出现次数非常多的符号，采用较少的位数来进行编码；对于出现次数非常少的符号，采用较多的位数来进行编码。通过变长编码，信源中的一些部分位数变少了，一些部分位数可能变多了，由于变少的部分比变多的部分多，所以整个信源信息量的大小还是

会减小，因此信源数据得到了压缩。

Huffman 编码方法是：将信源中的符号按照出现的频率从大到小进行排队，将出现频率最小的 2 个作为一组，分别编码为“1”和“0”；再把这个组的频率加起来得到一个新的频率，将这个新的频率与其他未被处理的符号按照频率的大小重新排队，将新生成队列里的频率最小的 2 个作为一组，分别编码为“1”和“0”。按照上面的过程重复下去，直到全部的符号处理完为止。上面的方法可以很容易采用树来实现，即 Huffman 树。下面介绍 Huffman 树。

7.1.2　Huffman 树

要进行 Huffman 编码，首先要把整个信源数据（比如文件）读一遍，在读的过程中，统计每个符号（我们把字节的 256 种值看作是 256 种符号）的出现次数。然后根据符号的出现次数，建立 Huffman 树，通过 Huffman 树得到每个符号的新的编码。对于信源数据中出现次数较多的符号，它的 Huffman 编码的位数比较少；对于信源数据中出现次数较少的符号，它的 Huffman 编码的位数比较多。然后把信源数据中的每个字节替换成它们新的编码。这个过程可以看作是一个树的建立过程。

首先，把所有符号看成是 1 个节点，并且该节点的值为它的出现次数。进一步把这些节点看成是只有 1 个节点的树。每次从所有树中找出值最小的 2 个树，为这 2 个树建立 1 个父节点，然后这 2 个树和它们的父节点组成 1 个新的树，这个新的树的值为它的 2 个子树的值的和。如此往复，直到最后所有的树变成了 1 棵树。我们就得到了 1 棵 Huffman 树。

这棵 Huffman 树，是 1 棵二叉树，它的所有叶子节点就是所有的符号，它的中间节点是在产生 Huffman 树的过程中不断建立的。我们在 Huffman 树的所有父节点到它的左子节点的路径上都标上“0”，右子节点的路径上都标上“1”。现在从根节点开始，到所有叶子节点的路径，就是一个“0”和“1”的序列。我们用根节点到 1 个叶子节点路径上的“0”和“1”的序列，作为这个叶子节点的 Huffman 编码。为了便于理解树的建立过程，下面举一个具体的例子进行说明。

假定信源数据的内容如为“abbbbccccddde”，统计得到各个符号的出现次数分别是：a 为 1 次、b 为 4 次、c 为 4 次、d 为 3 次、e 为 1 次。

首先将所有的符号进行排序，然后开始建立树。建立步骤分别如图 7-1 至图 7-5 所示。

通过图 7-1 至图 7-5 可以理解树的建立过程，通过图 7-5 可以知道各个符号的编码：a 为 110、b 为 00、c 为 01、d 为 10、e 为 111。

图 7-1　树的建立步骤①

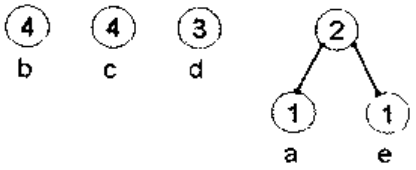


图 7-2　树的建立步骤②

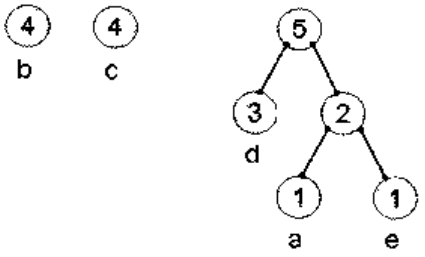


图 7-3　树的建立步骤③

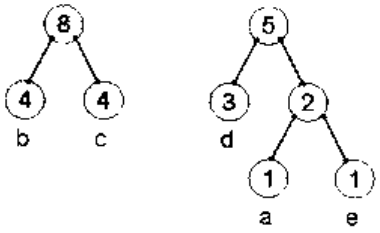


图 7-4　树的建立步骤④

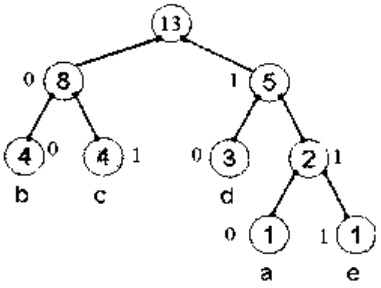


图 7-5　树的建立步骤⑤

7.1.3　使用 Huffman 算法压缩数据

统计信源数据中每个符号的出现次数。根据每个符号的出现次数，建立 Huffman 树，得到每个符号的 Huffman 编码。将每个符号出现次数的信息保存起来（比如保存在压缩文件中），并将信息数据源中的每个符号替换成它的 Huffman 编码并保存。在进行 Huffman 编码时，可以采用静态的编码方法，也可以采用动态的编码方法。静态 Huffman 编码就是使用预先定义好的一套编码进行压缩，解压缩的时候也使用这套编码，这样不需要传递用来生成树的信息。动态 Huffman 编码就是使用统计好的各个符号的出现次数，建立 Huffman 树，产生各个符号的 Huffman 编码，用产生的 Huffman 编码进行压缩，这种方式需要传递生成树的信息。对于静态树来说，不需要传递用来生成树的那部分信息，而动态树需要传递这个信息。当信源数据比较小的时候，传递生成树的信息得不偿失，反而会使压缩后的数据变大。也就是说对于信源数据比较小的时候，就可能会出现使用静态 Huffman 编码比使用动态 Huffman 编码生成的数据块小。

7.2　程序介绍

根据上面的介绍，对 Huffman 算法的实现有了清楚的认识。Huffman 算法主要包括 3

个部分的操作，即队列处理、Huffman 树的生成、Huffman 编码。下面分别对这 3 个部分的程序进行具体的介绍。

7.2.1　队列处理

队列处理是 Huffman 算法的基本组成部分，主要实现对节点进行插入和删除处理，下面为具体的程序代码。

```
void QueueInsert(int nWhich)
{
    int thisnode, previous;
    // 队列是否为空
    if(rootnodes == -1)
    {
        // 空队列
        node[nWhich].nParent = -1;
        rootnodes = nWhich;
    }
    else
    {
        thisnode = rootnodes;
        previous = -1;
        //搜索大的节点
        while((thisnode != -1) &&
            (node[thisnode].nFrequency < node[nWhich].nFrequency))
        {
            previous = thisnode;
            thisnode = node[thisnode].nParent;
        }

        // 连接到第一个大的节点
        node[nWhich].nParent = thisnode;
        if(previous != -1)
        {
            // 拷贝
            node[previous].nParent = nWhich;
        }
        else
        {
            // 插入到开始位置
            rootnodes = nWhich;
        }
    }
}
int QueueDelete()
```

```
{
    int thisnode = rootnodes;
    rootnodes = node[thisnode].nParent;
    return thisnode;
}
```

上面的程序实现了队列的增、删处理，在这部分程序的基础上就可以实现 Huffman 树的生成。

7.2.2 Huffman 树的生成

由前面的介绍知道，在对 Huffman 编码前需要生成 Huffman 树，下面为具体的程序代码。

```
// 形成 Huffman 树
for(nNextNode = nSymbolsNum;
    nNextNode < 2 * nSymbolsNum - 1;
    nNextNode ++)
{
    nLeftNode = QueueDelete();
    nRightNode = QueueDelete();

    // 形成新的树，作为子节点
    node[nLeftNode].nParent = nNextNode;
    node[nRightNode].nParent = nNextNode;
    node[nLeftNode].iSLeft = TRUE;
    node[nRightNode].iSLeft = FALSE;

    // 父节点的频率是两个子节点频率之和
    node[nNextNode].nFrequency =
        node[nLeftNode].nFrequency + node[nRightNode].nFrequency;

    //插入节点
    QueueInsert( nNextNode);
}
```

在上面的程序中，“node”为一个全局的结构变量，该结构变量为 Huffman 树的队列。该结构定义如下。

```
struct NodeType {
    int nFrequency;
    int nParent;
    int iSLeft;
};
```

7.2.3　Huffman 编码

在形成 Huffman 树后就可以进行 Huffman 编码。Huffman 编码是根据 Huffman 树来进行编码的，下面为具体的程序代码。

```
// 根节点
root = QueueDelete();
// 根据树进行编码
for(i = 0; i < nSymbolsNum; i++)
{
    // 搜索初始点
    cd.nStartPos = MAXBITS;

    // 对树进行遍历，对内容进行编码
    thisnode = i;
    while(thisnode != root)
    {
        --cd.nStartPos;
        cd.nBits[cd.nStartPos] = node[thisnode].iSLeft ? 0 : 1;
        thisnode = node[thisnode].nParent;
    }

    // 内容拷贝
    for(nBitCount = cd.nStartPos; nBitCount < MAXBITS; nBitCount++)
    {
        code[i].nBits[nBitCount] = cd.nBits[nBitCount];
    }
    code[i].nStartPos = cd.nStartPos;
}
```

在上面的程序中，首先找到树的根节点，然后对树进行遍历编码，从而实现 Huffman 编码。

在上面的程序中，“code”为一个结构变量，具体定义如下。

```
struct NodeType {
    int nFrequency;
    int nParent;
    int iSLeft;
};
```

将上面的 3 部分程序组合起来就是完整的 Huffman 算法。下面给出一个简单的测试程序。

```
void main()
{
    struct CodeType cd, code[MAXSYMBS];
    int i;
    int nSymbolsNum;
    int nBitCount;
    int nNextNode;
    int nLeftNode, nRightNode;
    int root;
    int thisnode;
    char syrbol, alphabet[MAXSYMBS];

    // 清空字符数组
    for(i = 0; i < MAXSYMBS; i++)
    {
        alphabet[i] = ' ';
    }
    //构造信源数据
    nSymbolsNum = 3;
    alphabet[0] = 'a';
    node[0].nFrequency = 8;
    QueueInsert(0);
    alphabet[1] = 'b';
    node[1].nFrequency = 2;
    QueueInsert(1);
    alphabet[2] = 'c';
    node[2].nFrequency = 5;
    QueueInsert(2);

    // 形成Huffman树
    for(nNextNode = nSymbolsNum;
        nNextNode < 2 * nSymbolsNum - 1;
        nNextNode ++)
    {
        nLeftNode = QueueDelete();
        nRightNode = QueueDelete();
        // 形成新的树，作为子节点
        node[nLeftNode].nParent = nNextNode;
        node[nRightNode].nParent = nNextNode;
        node[nLeftNode].isLeft = TRUE;
        node[nRightNode].isLeft = FALSE;
        // 父节点的频率是两个子节点频率之和
        node[nNextNode].nFrequency =
            node[nLeftNode].nFrequency + node[nRightNode].nFrequency;
        //插入节点
        QueueInsert( nNextNode);
    }
    // 根节点
```

```
    root = QueueDelete();
    // 根据树进行编码
    for(i = 0; i < nSymbolsNum; i++)
    {
        // 搜索初始点
        cd.nStartPos = MAXBITS;
        // 对树进行遍历，对内容进行编码
        thisnode = i;
        while(thisnode != root)
        {
            --cd.nStartPos;
            cd.nBits[cd.nStartPos] = node[thisnode].iSLeft ? 0 : 1;
            thisnode = node[thisnode].nParent;
        }
        // 内容拷贝
        for(nBitCount = cd.nStartPos; nBitCount < MAXBITS; nBitCount++)
        {
            code[i].nBits[nBitCount] = cd.nBits[nBitCount];
        }
        code[i].nStartPos = cd.nStartPos;
    }
}
```

通过上面的程序得到 Huffman 编码：a 为 1、b 为 00、c 为 01。

7.3　实例总结

本章主要介绍了 Huffman 算法的原理及具体的程序实现。通过本章的介绍，读者可以采用单片机实现数据的压缩。通过对数据进行压缩，可以使数据存储系统比未进行压缩时存储更多的数据；通过数据压缩，也可以使单片机实现的数据传输系统在经过数据压缩后进行更加有效的数据传输。由于 MSP430 单片机的处理能力比较强，因此可以有效实现 Huffman 算法。虽然本章介绍的算法处理的数据块不是很大，但是读者可以在本章的程序基础上稍加改动就可以实现更大数据块的压缩处理。

第 8 章

基于 MSP430 实现的 FIR 滤波器

目前，单片机的处理速度越来越高，采用单片机进行简单的数字信号处理也是完全可能的。在数字信号处理中，数字滤波是最基本的处理。本章介绍采用 MSP430 单片机实现 FIR 滤波器。

8.1 FIR 滤波器原理和设计方法

8.1.1 FIR 滤波器的原理

一般而言，一个数字滤波器可以用系统函数来表示，如式（8-1）所示。

$$H(Z)=\frac{\sum_{k=0}^{M}b_kZ^{-k}}{1-\sum_{k=1}^{N}a_kZ^{-k}}=\frac{Y(Z)}{X(Z)} \tag{8-1}$$

由式（8-1）的系统函数可以得到输入输出的差分方程。

$$y(n)=\sum_{k=1}^{N}a_ky(n-k)+\sum_{k=0}^{M}b_kx(n-k) \tag{8-2}$$

如果式（8-2）中的 $a_k(k=0, 1, \ldots, N)$的值为 0，那么得到式（8-3）。

$$y(n)=\sum_{k=0}^{M}b_kx(n-k) \tag{8-3}$$

式（8-3）就是 FIR 滤波器的差分方程，对此式进行 Z 变换就可以得到 FIR 滤波器的系统函数。

$$H(Z)=\sum_{k=0}^{N-1}h(k)Z^{-k} \tag{8-4}$$

由式（8-4）可以看出，FIR 滤波器主要有以下特点。

- 系统的单位冲激响应 $h(n)$在有限个 n 值处不为零。
- 系统函数 $H(Z)$在$|Z|>0$ 处收敛，极点全部在 Z=0 处（因果系统）。
- 结构上主要是非递归结构，没有输出反馈。
- 能够保证精确、严格的线性相位。

8.1.2　FIR 滤波器的设计

FIR 滤波器的设计方法主要有两种，即窗函数法和频率抽样法。窗函数法是从时域角度进行设计，频率抽样法是从频域角度出发进行设计。这里限于篇幅原因只是简单介绍窗函数法。

一般说来，首先给定所希望得到的滤波器的理想响应为 $H_d(e^{jw})$，那么 FIR 滤波器的设计任务就变成了设计出频率响应为 $H(e^{jw})$的系统函数去逼近 $H_d(e^{jw})$。由于设计是在时域进行的，因此先由 $H_d(e^{jw})$的傅里叶反变换导出 $h_d(n)$来。

$$h_d(n)=\frac{1}{2\pi}\int_{-\pi}^{\pi}H_d(e^{jw})e^{jnw}dw \tag{8-5}$$

由于 $H_d(e^{jw})$是矩形频率特性，因此 $h_d(n)$是无限长的序列，并且是非因果的。而设计出的 FIR 滤波器的 $h(n)$是有限长的，所以使用一个有限长度的窗函数 $w(n)$来截取 $h_d(n)$序列，即

$$h(n)=w(n)h_d(n) \tag{8-6}$$

由式（8-6）就可以计算得到需要设计的 FIR 滤波器的 $h(n)$。下面以低通滤波器的设计为例来说明 FIR 滤波器的具体设计。

截止频率为 w_c，群时延为 α 的线性相位的理想低通滤波器的频率响应为

$$H_d(e^{jw})=\begin{cases}e^{-j\alpha w},-w_c\leqslant w\leqslant w_c\\0,-\pi\leqslant w\leqslant -w_c,w_c\leqslant w\leqslant \pi\end{cases} \tag{8-7}$$

通过式（8-5）和式（8-7）可以得到

$$h_d(n)=\frac{w_c}{\pi}\frac{\sin[w_c(n-\alpha)]}{w_c(n-\alpha)} \tag{8-8}$$

由式（8-8）可以知道，$h_d(n)$是以 α 为中心的无限长序列，由于系统为线性相位系统，

因此$\alpha=(N-1)/2$。为了得到有限长序列的 $h(n)$，对 $h_d(n)$序列取矩形窗，并由式（8-6）可以得到式（8-9）。

$$h(n)=\begin{cases}\dfrac{w_c}{\pi}\dfrac{\sin[w_c(n-\frac{N-1}{2})]}{w_c(n-\frac{N-1}{2})}, 0\leqslant n\leqslant N-1\\ 0, \text{其他值}\end{cases} \tag{8-9}$$

通过式（8-9）就可以计算得到 $h(n)$的值，也就完成了滤波器的设计。关于加窗处理对频率的影响，以及窗函数的选择在本章不再进行详细介绍，读者可以查看相关的数字信号处理书籍。

8.2 定点程序实现

MSP430 单片机可以进行浮点运算，但是由于浮点运算相对比较慢，并且 MSP430 单片机本身处理能力不是很强，因此采用定点运算来实现程序。下面介绍定点程序的实现。

8.2.1 运算的定点模拟

所谓定点数就是使用 1 个整数来表示 1 个浮点数。1 个 16 位整数的前面几位来表示整数部分，使用后面的几位来表示小数部分，这样该 16 位整数表示的实际是 1 个浮点数。采用定点数表示浮点数又叫数的定标。定点数的表示有 Q 表示法和 S 表示法，这里介绍 Q 表示法。采用 Q 表示法表示的数据一般记为 Qn（n 为 0～15 之间的值）。对于 Qn 而言，前面的（15–n）位表示整数部分，后面的 n 位表示小数部分，最高位表示符号位。例如：$Q15$ 表示的数的范围为 $-1\leqslant x\leqslant 0.999965$。使用 Q 表示法就可以实现浮点数与定点数之间的转换。由浮点数转换成定点数可以采用下面的式子表示。

$$x_q=(\text{int})(x\times 2^Q) \tag{8-10}$$

在式（8-10）中，x_q为定点数，x 为浮点数。通过式（8-10）可以实现由定点数转换成浮点数。

$$x=(\text{float})(x_q\times 2^{-Q}) \tag{8-11}$$

通过式（8-10）和式（8-11），就很容易实现定点数与浮点数之间的转换。例如：浮点数为 0.25，在 Q=15 时，它的定点数为 8192。同理，定点数为 16384，在 Q=15 时，它的浮点数为 0.5。在了解数的定标后，下面介绍定点数的运算。

1．定点数的加减法

假设浮点数 z、x 和 y 的 Q 值分别为 Qz、Qx 和 Qy（假定 $Qx>Qy$），则 $z=x+y$ 的运算

可以表示成下面的式子。

$$z_q 2^{-Qz} = x_q 2^{-Qx} + y_q 2^{-Qy} \tag{8-12}$$

由式（8-12）可以得到下式。

$$z_q = [x_q + y_q 2^{(Qx-Qy)}]2^{(Qz-Qx)} \tag{8-13}$$

式（8-13）就为定点加法运算的公式，该式也同样适用于减法运算，在进行减法运算时，只是需要注意符号问题。为了便于理解，举例进行说明。例如：x=0.5，y=3.1，则 z=3.6。进行定点运算时，Qx 的值为 15，Qy 和 Qz 的值都为 13，按照式（8-13）可以计算得到 z_q 的值为 29491，转换成浮点数为 3.599976。

2．定点数的乘除法

假设浮点数 z、x 和 y 的 Q 值分别为 Qz、Qx 和 Qy（假定 $Qx>Qy$），则 $z = x \times y$ 的定点运算可以表示成下面的式子。

$$z_q = x_q y_q 2^{Qz-Qx-Qy} \tag{8-14}$$

为了便于理解，举例进行说明。例如：x=18.4，y=36.8，则 z=677.12。进行定点运算时，Qx 的值为 10，Qy 的值为 9，Qz 的值为 5，按照式（8-14）可以计算得到 z_q 的值为 21666，转换成浮点数为 677.08。同理，也可以得到定点除法运算的式子。

$$z_q = \frac{x_q 2^{Qz-Qx+Qy}}{y_q} \tag{8-15}$$

由式（8-15）可以进行定点数的除法运算。在进行定点数的运算时，一定要根据数的动态范围来进行定标，在定标时需要综合考虑数的动态范围和数的精度要求，根据实际情况进行选择。另外，在进行定点运算时，需要考虑数的动态定标，这是定点运算的难点，这里限于篇幅原因不进行详细的介绍。

8.2.2　定点程序实现

在前面介绍的基础上，就可以实现 FIR 滤波器。由于不同的信号处理所需要的滤波器不同，因此首先是根据系统需要设计出滤波器，设计出滤波器后就可以对输入信号进行滤波处理了。下面给出 1 个低通滤波器的定点程序，该滤波器的截止频率为 800Hz，滤波器的长度为 19 点。具体程序如下。

```
void LowpassFilter(short int nIn[],short int nOut[],int nLen,short int h[])
{
    short int i,j;
    int sum;
    //缓冲区的内容更新
```

```
    for(i = 0;i < FRAME;i++)
    {
        nBuff[i + nLen -1] = nIn[i];
    }
    //FIR滤波处理
    for(i = 0;i < FRAME;i++)
    {
        sum = 0;
        for(j = 0;j < nLen;j++)
        {
            sum += h[j] * nBuff[i - j + nLen -1];
        }
        nOut[i] = sum >> 15;
    }
    //更新缓冲区的内容
    for(i = 0;i < nLen - 1;i++)
    {
        nBuff[nLen - i - 2] = nIn[FRAME - i - 1];
    }
}
```

在上面的程序中，主要是对一帧语音数据进行滤波处理。由于语音数据是一帧一帧进行处理，所以在处理的时候需要更新缓冲区。

上面的程序只是滤波部分，下面给出测试程序，为了简单起见，输入的数据从文件读取，并将滤波后的数据写入文件。

```
#define FRAME       180
short int h[19] = {
    399,-296,-945,-1555,
    -1503,-285,2112,5061,
    7503,8450,7503,5061,
    2112,-285,-1503,-1555,
    -945,-296,399
};
static short int nBuff[FRAME + 20];
void LowpassFilter(short int nIn[],short int nOut[],int nLen,short int h[]);
void main()
{
    FILE *m_pInput;
    FILE *m_pOutput;
    short int input[FRAME];
    short int output[FRAME];
    int count;

    // 打开输入文件
    if((m_pInput = fopen("input.wav", "rb")) == NULL)
    {
```

```
        return;
    }
    //打开输出文件
    if((m_pOutput = fopen("output.wav", "wb")) == NULL)
    {
        return;
    }
    count = 0;
    while( fread(&input[0], sizeof(short int), FRAME, m_pInput) == FRAME)
    {
        printf("Frame =%d\r", count++);
        //滤波处理
        LowpassFilter(input,output,19,h);
        //将滤波后的数据写到文件里
        fwrite(output, sizeof(short int), FRAME, m_pOutput);
    }
    fclose(m_pOutput);
    fclose(m_pInput);
}
```

8.3　实例总结

本章介绍了 FIR 滤波器的原理，并介绍了 FIR 滤波器的设计方法。考虑单片机处理能力有限，采用定点程序实现 FIR 滤波器，为此介绍了定点运算的模拟，并给出了一个具体滤波器的定点程序。通过本章的介绍，读者可以根据自己的要求设计出相应的 FIR 滤波器。在进行滤波处理时，一定要考虑数的动态定标，这是处理的难点。

第 9 章

基于 MSP430 实现的 FFT 算法

目前，单片机的处理速度越来越高，采用单片机进行简单的数字信号处理也是完全可能的。在数字信号处理中，FFT 是一种重要的分析工具。在单片机实现的某些测量系统中，可能会对信号进行谱分析，那么 FFT 基本上就成了必需的工具。虽然 FFT 的运算量可能比较大，但是在不需要进行实时处理的场合下，也可以使用单片机来实现。本章介绍采用 MSP430 单片机实现 FFT 算法。

9.1 算法原理

设 $x(n)$为 N 点有限长序列，则它的 DFT 可以由下式表示。

$$X(k)=\sum_{n=0}^{N}x(n)W_N^{nk},k=0,1,...,N-1 \tag{9-1}$$

在式（9-1）中，W_N^{nk} 为蝶形运算因子，它等于 $e^{-j2kn\pi/N}$。

通过式（9-1）可以看出，计算所有的 $X(k)$大约需要 N^2 次乘加运算。当 N 的值很大时，DFT 的运算量相当大，因此需要对算法进行研究以求能减少运算量。通过蝶形运算因子可以看出：

$$W_N^k=-W_N^{k+N/2} \tag{9-2}$$

$$W_N^k=W_N^{N+k} \tag{9-3}$$

利用式（9-2）和式（9-3）所给出的对称性和周期性就可以减少 DFT 的运算量，实现 DFT 的快速算法，即 FFT 算法。一般而言，FFT 算法可以分为时间抽取 FFT 和频率 FFT，下面主要对时间抽取 FFT 进行介绍。

首先假设序列长度为 $N=2^L$，L 为整数，如果不满足该条件，可以对序列进行补零，使序列满足该条件。将式（9-1）中的序列分成偶数序列和奇数序列可以得到下式。

$$X(k)=\sum_{n=0}^{N/2-1}x(2n)W_N^{2nk}+\sum_{n=0}^{N/2-1}x(2n+1)W_N^{(2n+1)k} \tag{9-4}$$

由于 $W_N^2=W_{N/2}$，所以根据式（9-4）可以得到下式。

$$X(k)=\sum_{n=0}^{N/2-1}x(2n)W_{N/2}^{nk}+W_N^k\sum_{n=0}^{N/2-1}x(2n+1)W_{N/2}^{nk} \tag{9-5}$$

式（9-5）可以进一步表示成下式。

$$X(k)=X_1(k)+X_2(k),k=0,1,...,N/2-1 \tag{9-6}$$

由式（9-6）可以看出，$X_1(k)$和 $X_2(k)$分别是偶数序列和奇数序列的 $N/2$ 点的 FFT。由于式（9-6）计算得到的是前 $N/2$ 点的 FFT 的值，则利用对称性可以得到后 N/2 点的值，如式（9-7）所示。

$$X(k+N/2)=X_1(k+N/2)+W_N^{k+N/2}X_2(k+N/2) \tag{9-7}$$

式（9-7）可以进一步简化为下式。

$$X(k)=X_1(k)-W_N^kX_2(k),k=0,1,...,N/2-1 \tag{9-8}$$

式（9-6）和式（9-8）可以采用如图 9-1 所示的蝶形信号流图符号来表示。

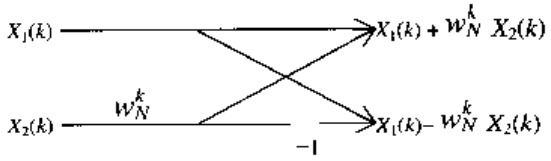


图 9-1　蝶形信号流图符号

对 X_1 和 X_2 可以进一步分解，最后可以将序列长度为 $N=2^L$ 的序列分解 L 级，每级有 $N/2$ 个 2 点的 FFT 蝶形运算，因此 N 点 FFT 总共有$(N/2)*(\log_2 N)$个蝶形运算，运算量得到了很大的简化。如图 9-2 所示为 8 点 FFT 运算的分解示意图。

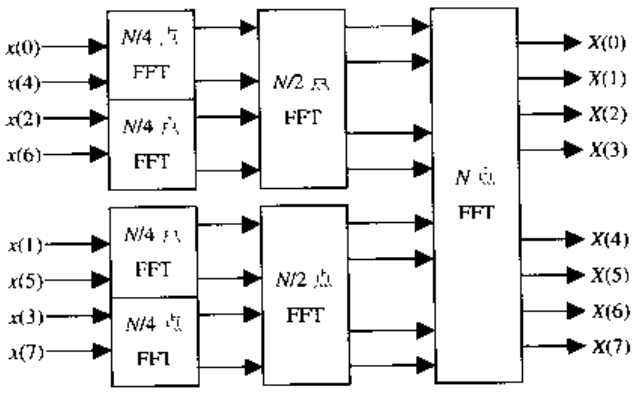


图 9-2　8 点 FFT 分解运算框图

由图 9-2 可以看出，对输入数字序列进行了重新排序，而输出序列是自然顺序。FFT 也可以是输入序列为自然顺序，输出序列则不是自然顺序。上面的重新排序实际就是码位倒序排列。关于 FFT 算法原理限于篇幅就简单介绍到这里，如果读者需要进一步详细了解 FFT 的原理，请查看相关数字信号处理书籍。

9.2 定点程序实现

虽然 MSP430 单片机可以进行浮点运算，但是由于浮点运算相对比较慢，并且 MSP430 单片机本身处理能力不是很强，加上 FFT 的运算量比较大，因此采用定点运算来实现程序。在介绍定点程序实现前，首先介绍定点运算的基本操作。

9.2.1 定点运算的基本操作

关于定点实现在 FIR 滤波器实现时做了介绍，这里就不再介绍。在以下的程序中使用了下面的类型定义。

```
typedef long int Longword;
typedef short int Shortword;
```

下面给出定点运算的加减法运算、乘法运算及移位操作的程序实现。具体的程序代码如下。

```
Shortword add(Shortword var1, Shortword var2)
{
    Longword L_sum;
    Shortword swOut;
    L_sum = (Longword) var1 + var2;
    swOut = saturate(L_sum);
    return (swOut);
}
Shortword sub(Shortword var1, Shortword var2)
{
    Longword L_diff;
    Shortword swOut;
    L_diff = (Longword) var1 - var2;
    swOut = saturate(L_diff);
    return (swOut);
}
```

上面的程序分别为 16 位数的加法程序和减法程序。在上面的程序中，“saturate（）”函数用来进行饱和处理，具体的程序如下。

```
static Shortword saturate(Longword L_var1)
{
    Shortword swOut;
    if (L_var1 > SW_MAX)
    {
        swOut = SW_MAX;
    }
    else if (L_var1 < SW_MIN)
    {
        swOut = SW_MIN;
    }
    else
    {
        swOut = (Shortword) L_var1;
    }
    return (swOut);
}
```

上面的程序主要是判断操作数是否大于最大的正数值或者是否小于最小的负数值，还是属于正常值，并分别对不同情况进行处理。

```
Shortword mult(Shortword var1, Shortword var2)
{
    Longword L_product;
    Shortword swOut;
    L_product = L_mult(var1, var2);
    swOut = extract_h(L_product);
    return (swOut);
}
```

上面的程序为 16 位数的乘法程序。在定点程序中，移位操作是基本的操作，下面给出右移的模拟程序。

```
Shortword shr(Shortword var1, Shortword var2)
{
    Shortword swMask,swOut;
    if (var2 == 0 || var1 == 0)
        swOut = var1;
    else if (var2 < 0)
    {
        if (var2 <= -15)
        {
            swOut = (var1 > 0) ? SW_MAX : SW_MIN;
        }
        else
            swOut = shl(var1, (Shortword)-var2);
    }
    else
```

```
    {
        if (var2 >= 15)
            swOut = (var1 < 0) ? (Shortword) 0xffff : 0x0;
        else
        {
            swMask = 0;
            if (var1 < 0)
            {
                swMask = ~swMask << (16 - var2);
            }
            var1 >>= var2;
            swOut = swMask | var1;
        }
    }
    return (swOut);
}
```

上面的程序主要是针对多种情况，判断比较多。如果知道移位位数的情况下，使用一个语句就可以完成操作，比如“nTemp >>= 1;”表示右移 1 位。

以上的程序都是 16 位数的操作，在运算的时候有可能会涉及到 32 位数的操作，32 位数的操作原理和 16 位数的原理相同，只是在处理是否饱和或者溢出时不同，这里限于篇幅原因就不再详细介绍了，本书的光盘中提供了所有定点模拟操作的程序。需要强调一点的是，定点程序主要是起模拟定点运算的作用，在实际写程序的时候，需要对某些函数直接进行替换，比如将“nTemp = shr(nTemp,1);”替换成“nTemp >>= 1;”。

9.2.2 程序实现

根据前面的介绍，FFT 运算首先需要使用正弦和余弦值，下面给出计算正弦和余弦的函数，具体的程序如下。

```
#define PI_Q13    24576
Shortword sin_fxp(Shortword x)
{
    static Shortword table[129] =
    {
        0,  402,  804, 1206, 1608, 2009, 2411, 2811, 3212,
     3612, 4011, 4410, 4808, 5205, 5602, 5998, 6393, 6787,
     7180, 7571, 7962, 8351, 8740, 9127, 9512, 9896, 10279,
    10660, 11039, 11417, 11793, 12167, 12540, 12910, 13279, 13646,
    14010, 14373, 14733, 15091, 15447, 15800, 16151, 16500, 16846,
    17190, 17531, 17869, 18205, 18538, 18868, 19195, 19520, 19841,
    20160, 20475, 20788, 21097, 21403, 21706, 22006, 22302, 22595,
    22884, 23170, 23453, 23732, 24008, 24279, 24548, 24812, 25073,
    25330, 25583, 25833, 26078, 26320, 26557, 26791, 27020, 27246,
```

```
27467, 27684, 27897, 28106, 28311, 28511, 28707, 28899, 29086,
29269, 29448, 29622, 29792, 29957, 30118, 30274, 30425, 30572,
30715, 30853, 30986, 31114, 31238, 31357, 31471, 31581, 31686,
31786, 31881, 31972, 32058, 32138, 32214, 32286, 32352, 32413,
32470, 32522, 32568, 32610, 32647, 32679, 32706, 32729, 32746,
32758, 32766, 32767
};

Shortword tx, ty;
Shortword sign;
Shortword index1,index2;
Shortword m;
Shortword temp;

sign = 0;
if (x < 0)
{
    tx = -x;
    sign = -1;
}
else
{
    tx = x;
}

if (tx > X05_Q15)
{
    tx = sub(ONE_Q15,tx);
}
//将输入转换成 0~128 范围内
index1 = shr(tx,7);
index2 = add(index1,1);

if (index1 == 128)
{
    if (sign != 0)
        return(-(table[index1]));
    else
        return(table[index1]);
}
m = sub(tx,shl(index1,7));
m = shl(m,8);

temp = sub(table[index2],table[index1]);
temp = mult(m,temp);
ty = add(table[index1],temp);

if (sign != 0)
```

```
        return(-ty);
    else
        return(ty);
}
Shortword cos_fxp(Shortword x)
{
    static Shortword table[129] =
    {
        32767, 32766, 32758, 32746, 32729, 32706, 32679, 32647, 32610,
        32568, 32522, 32470, 32413, 32352, 32286, 32214, 32138, 32058,
        31972, 31881, 31786, 31686, 31581, 31471, 31357, 31238, 31114,
        30986, 30853, 30715, 30572, 30425, 30274, 30118, 29957, 29792,
        29622, 29448, 29269, 29086, 28899, 28707, 28511, 28311, 28106,
        27897, 27684, 27467, 27246, 27020, 26791, 26557, 26320, 26078,
        25833, 25583, 25330, 25073, 24812, 24548, 24279, 24008, 23732,
        23453, 23170, 22884, 22595, 22302, 22006, 21706, 21403, 21097,
        20788, 20475, 20160, 19841, 19520, 19195, 18868, 18538, 18205,
        17869, 17531, 17190, 16846, 16500, 16151, 15800, 15447, 15091,
        14733, 14373, 14010, 13646, 13279, 12910, 12540, 12167, 11793,
        11417, 11039, 10660, 10279,  9896,  9512,  9127,  8740,  8351,
        7962,   7571,  7180,  6787,  6393,  5998,  5602,  5205,  4808,
        4410,   4011,  3612,  3212,  2811,  2411,  2009,  1608,  1206,
        804,     402,     0
    };

    Shortword tx, ty;
    Shortword sign;
    Shortword index1,index2;
    Shortword m;
    Shortword temp;

    sign = 0;

    if (x < 0)
    {
        tx = -x;
    }
    else
    {
        tx = x;
    }

    if (tx > X05_Q15)
    {
        tx = sub(ONE_Q15,tx);
        sign = -1;
    }
    //将输入转换成 0~128 范围内
```

```
    index1 = shr(tx,7);
    index2 = add(index1,1);

    if (index1 == 128)
        return((Shortword)0);
    m = sub(tx,shl(index1,7));
    m = shl(m,8);

    temp = sub(table[index2],table[index1]);
    temp = mult(m,temp);
    ty = add(table[index1],temp);

    if (sign != 0)
        return(-ty);
    else
        return(ty);
}
```

在上面的函数程序基础上，就可以完成 FFT 的初始化，初始化程序主要是计算出实部和虚部的值。具体程序如下：

```
void fs_init()
{
    Shortword i;
    Shortword fft_len2,shift,step,theta;

    fft_len2 = shr(FFTLENGTH,1);
    shift = norm_s(fft_len2);
    step = shl(2,shift);
    theta = 0;

    for (i = 0; i <= fft_len2; i++)
    {
        wr_array[i] = cos_fxp(theta);
        wi_array[i] = sin_fxp(theta);
        if (i >= (fft_len2-1))
            theta = ONE_Q15;
        else
            theta = add(theta,step);
    }
}
```

在完成初始化工作后，获得了正弦和余弦的值，就可以进行 FFT 计算。下面给出 FFT 的具体程序。

```
#define SWAP(a,b) temp1 = (a);(a) = (b); (b) = temp1
void fft(Shortword *datam1,Shortword nn,Shortword isign)
{
```

```
Shortword n,mmax,m,j,istep,i;
Shortword wr,wi,templ;
Longword register L_tempr,L_tempi;
Shortword *data;
Longword L_templ,L_temp2;
Shortword index,index_step;

data = &data[1[-1];

n = shl(nn,1);
j = 1;
for( i = 1; i < n; i+=2 )
{
    if ( j > i)
    {
        SWAP(data[j],data[i]);
        SWAP(data[j+1],data[i+1]);
    }
    m = nn;
    while ( m >= 2 && j > m )
    {
        j = sub(j,m);
        m = shr(m,1);
    }
    j = add(j,m);
}
mmax = 2;
index_step = nn;
while ( n > mmax)
{
    istep = shl(mmax,1);

    index = 0;
    index_step = shr(index_step,1);

    wr = ONE_Q15;
    wi = 0;
    for ( m = 1; m < mmax;m+=2)
    {
        for ( i = m; i <= n; i += istep)
        {
            j = i + mmax;
            //tempr = wr * data[j] - wi * data[j+1]
            L_templ = L_shr(L_mult(wr,data[j]),1);
            L_temp2 = L_shr(L_mult(wi,data[j+1]),1);
            L_tempr = L_sub(L_templ,L_temp2);
            //tempi = wr * data[j+1] + wi * data[j]
            L_templ = L_shr(L_mult(wr,data[j+1]),1);
```

```
                L_temp2 = L_shr(L_mult(wi,data[j]),1);
                L_tempi = L_add(L_temp1,L_temp2);
                //data[j] = data[i] - tempr
                L_temp1 = L_shr(L_deposit_h(data[i]),1);
                data[j] = extract_h(L_sub(L_temp1,L_tempr));
                //data[i] += tempr
                data[i] = extract_h(L_add(L_temp1,L_tempr));
                //data[j+1] = data[i+1] - tempi
                L_temp1 = L_shr(L_deposit_h(data[i+1]),1);
                data[j+1] = extract_h(L_sub(L_temp1,L_tempi));
                //data[i+1] += tempi
                data[i+1] = extract_h(L_add(L_temp1,L_tempi));
            }
            index = add(index,index_step);
            wr = wr_array[index];
            if (isign < 0)
                wi = negate(wi_array[index]);
            else
                wi = wi_array[index];
        }
        mmax = istep;
    }
}
```

以上给出了 FFT 的算法程序，在使用该算法程序的时候，需要对上面的程序进行优化处理。优化处理主要是使用直接的整数加减、移位、乘法操作去替换程序中采用定点模拟实现的加减、移位、乘法等操作。

9.3　实例总结

本章介绍了 FFT 的原理，并给出了具体的实现程序。考虑单片机处理能力有限，采用定点程序实现 FFT 算法，为此给出了定点运算的模拟程序，最后给出了完整的 FFT 程序。通过本章的介绍，读者可以了解 FFT 算法的具体程序实现，也完全可以将本章介绍的程序运用到所需要的系统中去。由于 MSP430 单片机是 16 位单片机，因此比较适合进行 FFT 运算，对于其他具体的某些单片机，可能需要进行相应的优化处理和其他相应的处理。由于 FFT 的运算量比较大，因此对单片机应用来说，只能用于非实时应用场合。如果需要实时应用的话，可能只能考虑采用 DSP 来实现了。

第 10 章

MSP430 串口通信的波特率自动识别

在单片机与 PC 机的串口通信中，通信速率一般是固定的，这样就要求通信双方必须按照规定的速率进行通信。如果通信双方通信速率不一致，则通信将不会成功。本章介绍 MSP430 单片机实现的串口通信的波特率自动识别，并给出实现的具体程序。

10.1 实现原理

在单片机应用中，上位机与下位机的通信一般是采用串口实现的。一般说来，串口通信的速率是固定的，在这种情况下，如果上位机不知道下位机的串口速率，则通信不会成功，因此实现波特率的自动识别是完全有必要的。下面介绍波特率自动识别的实现原理。

10.1.1 系统组成

整个系统主要包括上位机和下位机。上位机就是一般的 PC 或者工控机，下位机主要由 MSP430 组成。上位机与下位机通过串口实现数据通信，由于单片机和 PC 机的串口接口电平不一致，因此需要进行电平转换，具体的实现电路可以参看第 25 章（MSP430 与 PC 机通信的设计与实现）的电路，这里不再进行详细介绍。如图 10-1 所示为系统的组成图。

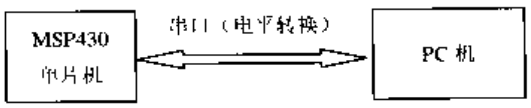


图 10-1 系统组成图

10.1.2　识别原理

对于常用的串口通信速率，一般速率从高到低存在某种分数关系，比如常用的速率 115200 波特/秒、57600 波特/秒、38400 波特/秒、19200 波特/秒、9600 波特/秒、4800 波特/秒、2400 波特/秒，57600 是 115200 的 1/2，38400 是 115200 的 1/3。正因为上述关系的存在，就可以实现速率的自动识别。首先将串口设定为某一个高速率，比如为 115200 波特/秒，这样如果实际速率为 57600 波特/秒的话，那么这个时候位脉冲宽度为 115200 的 2 倍，相当于脉冲的宽度展宽了 2 倍。同理也可以得出 38400 波特/秒以下速率的脉冲宽度。脉冲宽度虽然展宽了，但是在读数时认为周期没有变化，即读数时仍然按照以前的位宽度来读取数据，这个时候读取的数据因为速率的不同而有不同的结果。根据这个原理，在实际处理中，单片机首先设定串口的通信速率为 115200 波特/秒，计算机发送 ASCII 码值为“13”的字符，如果串口实际速率为 115200 波特/秒，那么单片机接收到的就是 ASCII 码值为“13”的字符；如果实际速率不是 115200 波特/秒，那么接收到的数据就不是 ASCII 码值为“13”的字符。如果实际速率为 9600 波特/秒，则接收到的值为“0”，低于 9600 波特/秒以下的速率就不能判断，这时再将串口通信速率设为 9600 波特/秒，继续按照上面的方式进行判断。如图 10-2 所示为速率在 115200 波特/秒与 9600 波特/秒之间的位模式图。

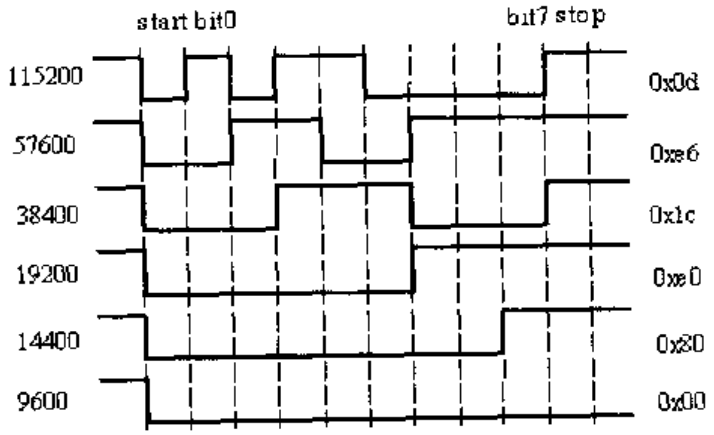


图 10-2　115200 波特/秒与 9600 波特/秒之间的位模式图

由图 10-2 可以看出，当初始速率设置为 115200 波特/秒时，实际速率低于 9600 波特/秒时不能确定，重新将速率设置为 9600 波特/秒，再继续判断。如图 10-3 所示为速率在 9600 波特/秒与 1200 波特/秒之间的位模式图。

通过图 10-2 和图 10-3 基本能理解速率识别的原理，下面在此基础上介绍程序的实现。

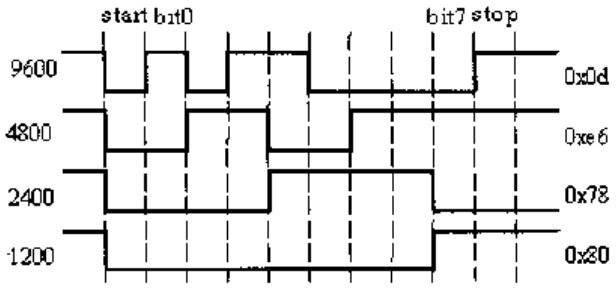


图 10-3　9600 波特/秒与 1200 波特/秒之间的位模式图

10.2　程序实现

整个系统的软件由初始化设置、速率自动识别和串口通信程序 3 个部分组成，下面分别进行详细介绍。

10.2.1　初始化设置

初始化设置主要设置端口、串口和定时器等。设置端口主要是确定管脚的输入输出方向。设置串口主要是设置串口的初始状态，比如传输的位数等。由于需要采用定时器进行延时处理，因此需要初始化定时器。下面为具体的程序代码。

```
void Init_UART1(void)
{
    //将寄存器的内容清零
    U1CTL = 0X00;
    //数据位为 8bit
    U1CTL += CHAR;

    //将寄存器的内容清零
    U1TCTL = 0X00;
    //波特率发生器选择 SMCLK
    U1TCTL += SSEL1;
    //波特率为 115200
    UBR0_1 = 0X45;
    UBR1_1 = 0X00;
    //调整寄存器
    UMCTL_1 = 0X00;

    //使能 UART1 的 TXD 和 RXD
    ME2 |= UTXE1 + URXE1;
    //使能 UART1 的 RX 中断
```

```
    IE2 |= URXIE1;
    //使能 UART1 的 TX 中断
    IE2 |= UTXIE1;

    //设置 P3.6 为 UART1 的 TXD
    P3SEL |= BIT6;
    //设置 P3.7 为 UART1 的 RXD
    P3SEL |= BIT7;
    //P3.6 为输出管脚
    P3DIR |= BIT6;
    return;
}
void Init_Port(void)
{
    //将所有的管脚在初始化的时候设置为输入方式
    P3DIR = 0;
    //将所有的管脚设置为一般 I/O 口
    P3SEL = 0;
    return;
}
// 初始化定时器模块
void Init_TimerA(void)
{
    // 选择 SMCLK，清除 TAR
    TACTL = TASSEL1 + TACLR;
    // 1/8 SMCLK
    TACTL += ID1;
    TACTL += ID0;
    // CCR0 中断允许
    CCTL0 = CCIE;
    // 时间间隔为 33.8ms
    CCR0 = 33800;
    // 增计数模式
    TACTL |= MC0;
}
```

另外，时钟也需要进行初始化，关于时钟的初始化，读者可以参看前面的介绍，这里不再进行介绍。

10.2.2　速率自动识别

根据前面关于速率识别原理的介绍知道，单片机首先设置串口速率为 115200 波特/秒，单片机等待接收 PC 机发送的 ASCII 码值为“13”的字符，单片机根据接收到的值判断速率是否在 115200 波特/秒与 9600 波特/秒之间；如果速率不在 115200 波特/秒与 9600 波特/秒之间，单片机再设置串口速率为 9600 波特/秒，单片机再次等待接收 PC 机发送的 ASCII

码值为“13”的字符，单片机根据接收到的值判断速率是否在 9600 波特/秒与 1200 波特/秒之间。经过上述的两步处理，基本能确定串口通信速率，在确定串口通信速率之后，设置串口的通信速率。这样单片机与 PC 机的速率就匹配好了，可以进行正常的通信了。下面给出实现的具体代码。

```
int AuToBaud(void)
{
    baudrates rate;

    //UART 复位
    U1CTL |= SWRST;
    //接收出错中断允许位
    URCTL1 = URXEIE;
    //UART 模块允许
    U1CTL &= ~SWRST;
    //设置速率为 115200 波特/秒
    SetBaud(BAUD115K);
    //接收字符
    switch (GetChar())
    {
    case 0x0d :
        rate = BAUD115K;
        break;
    case 0xe6 :
        rate = BAUD57600;
        break;
    case 0x1c :
        rate = BAUD38400;
        break;
    case 0xe0 :
        rate = BAUD19200;
        break;
    case 0x80 :
        rate = BAUD14400;
        break;
    case 0x00 :
    {
        // CCR0 中断允许
        CCTL0 = CCIE;
        //延时 33.8ms
        while(true)
        {
            if(nTime_Flag == 1)
            {
                nTime_Flag = 0;
                break;
```

```
            }
        }
        //设置速率为 9600 波特/秒
        SetBaud(BAUD9600);
        //接收字符
        switch (GetChar())
        {
        case 0x0d :
            rate = BAUD9600;
            break;
        case 0xe6 :
            rate = BAUD4800;
            break;
        case 0x78 :
            rate = BAUD2400;
            break;
        case 0x80 :
            rate = BAUD1200;
            break;
        default :
            return -1;
        }
        break;
    }
    default :
        return -1;
    }

    // CCR0 中断允许
    CCTL0 = CCIE;
    //延时 33.8ms
    while(true)
    {
        if(nTime_Flag == 1)
        {
            nTime_Flag = 0;
            break;
        }
    }
    SetBaud(rate);
    return 0;
}
```

在上面的代码中，使用“SetBaud(BAUD9600);”进行串口设置，具体的代码如下。

```
void SetBaud(int baud)
{
    //停止 UART
```

```
    U1CTL |= SWRST;
    //禁止中断
    ME2 &= ~(UTXE0 + URXE0);

    //设置寄存器
    UBR0_1 = baudregs[baud].ubr00;
    UBR1_1 = baudregs[baud].ubr01;
    UMCTL_1 = baudregs[baud].umctl0;
    //使能中断
    ME2 |= UTXE0 + URXE0;
    //UART 模块允许
    U1CTL &= ~SWRST;
}
```

baudregs 定义为串口的基本信息结构，定义如下。

```
const baudreg baudregs[] =
{
    0x45, 0x00, 0x00, "115200",    // BAUD115K
    0x8b, 0x00, 0x00, "57600",     // BAUD57600
    0xd0, 0x00, 0x00, "38400",     // BAUD38400
    0xa0, 0x01, 0x00, "19200",     // BAUD19200
    0x2c, 0x02, 0x00, "14400",     // BAUD14400
    0x41, 0x03, 0x00, "9600",      // BAUD9600
    0x83, 0x06, 0x00, "4800",      // BAUD4800
    0x05, 0x0d, 0x00, "2400",      // BAUD2400
    0x0b, 0x1a, 0x00, "1200"       // BAUD1200
};
```

在上面的速率识别程序中，通过等待“nTime_Flag”的标志为 1 来实现延时，延时程序使用定时器 A 的中断程序来实现。定时器 A 的中断程序很简单，只是在中断产生时设置“nTime_Flag”的值为 1，这里不给出具体的程序代码。

10.2.3 串口通信程序

在前面的串口速率识别后，设置好串口的通信速率，上位机和下位机就能进行正确的通信。单片机的串口通信主要采用中断服务程序处理，下面给出具体的程序代码。

```
interrupt [UART1RX_VECTOR] void UART1_RX_ISR(void)
{
    //接收来自串口的数据
    UART1_RX_BUF[nRX1_Len_temp] = RXBUF1;
    nRX1_Len_temp += 1;
    if(UART1_RX_BUF[nRX1_Len_temp - 1] == 13)
    {
        nRX1_Len = nRX1_Len_temp;
```

```
            nRev_UART1 = 1;
            nRX1_Len_temp = 0;
        }
    }
    interrupt [UART1TX_VECTOR] void UART1_TX_ISR(void)
    {
        if(nTX1_Len != 0)
        {
            // 表示缓冲区里的数据没有发送完
            nTX1_Flag = 0;
            TXBUF1 = UART1_TX_BUF[nSend_TX1];
            nSend_TX1 += 1;
            if(nSend_TX1 >= nTX1_Len)
            {
                nSend_TX1 = 0;
                nTX1_Len = 0;
                nTX1_Flag = 1;
            }
        }
    }
```

上面的程序为串口 1 的收发中断服务程序。收发程序都处于等待状态，一旦外面有数据到来，则触发接收，进入接收中断服务程序，接收中断程序接收数据。中断程序从“RXBUF1”寄存器里读取数据，将得到的数据放到“UART1_RX_BUF[]”全局缓冲区里，在接收到数据后设置一个标志“nRev_UART1”来通知主程序。如果有数据需要发送的时候，主程序设置一个发送标志，并且触发发送中断，进入发送中断服务程序。在发送中断程序里，从“UART1_TX_BUF[]”全局缓冲区里取出数据送给“TXBUF1”寄存器进行发送，发送完数据后，发送中断程序等待下一次中断的到来。

10.3　实例总结

本章介绍了 MSP430 单片机实现串口速率自动识别的原理，并讨论了具体的程序实现。利用本章介绍的方法，可以自动识别串口通信的速率。在使用本方法时需要注意的是，在单片机与 PC 机进行正常的通信前，需要进行串口速率的识别，在识别时需要 PC 机向单片机发送 ASCII 码值为“13”的字符。当速率识别成功后，单片机可以向 PC 机发送识别成功的响应，此后，单片机和 PC 机就可以进行正常的通信了。

第三篇

存储应用

第 11 章

串行存储器 24LC02B 的设计与应用

在单片机应用中，很多时候需要记录一些配置信息或者运行参数，在这种应用下，单片机需要与串行存储器进行接口。单片机系统可以在开机时将初始化信息从串行存储器里读出，在程序运行过程中，也可以记录系统运行的一些参数。本章介绍 MSP430 单片机与串行存储器芯片 24LC02B 的接口设计，并给出相应的程序。

11.1 硬件接口设计

系统的硬件系统相对很简单，主要有电源电路、复位监控电路、串行存储器电路和单片机电路，其中电源电路和复位监控电路参看第 2 章的图 2-3，这里就不再进行介绍了。

11.1.1 24LC02B 芯片

在介绍具体的电路之前，先介绍一下 24LC02B 芯片。很多 IC 厂商都生产 24LC02B 芯片，本章介绍的 24LC02B 芯片是由 Microchip 生产的。该芯片具有以下特性：

- 采用单电源供电。电压范围为 2.5V~5.5V。
- 低功耗。芯片工作时，典型的电流值为 1mA；芯片不工作时，典型的电流值为 1μA。
- 以 256 个字节为单位组成一块。
- I^2C 总线接口。
- 页写缓冲区高达 8 个字节。

- 具有写保护功能，这样可以作为只读 ROM 使用。
- 多达 1 百万次擦写。
- 数据可以长期保存。

为了便于进行硬件电路的设计，下面给出 24LC02B 芯片的管脚图，如图 11-1 所示。

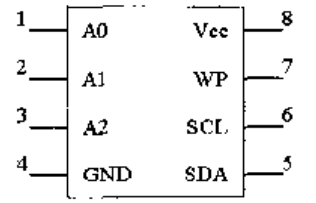


图 11-1　24LC02B 管脚图

由图 11-1 可以看出，该芯片只有 8 个管脚，这样使用起来简单，只需要简单的外围电路即可。下面对具体的管脚进行介绍。

- A0~A2：地址线。用来进行器件寻址。
- GND：电源地。
- SCL：串行时钟输入脚，由于在 SCL 上升/下降沿处理信号，要特别注意 SCL 信号的上升/下降沿升降时间。此管脚通常用一电阻上拉。
- SDA：串行数据输入/输出脚，此管脚通常用一电阻上拉。
- WP：写保护管脚。如果该管脚接高电平，则处于写保护状态；如果该管脚接低电平，则可以进行读写。
- V_{CC}：电源管脚。

经过对串行存储器芯片 24LC02B 的介绍，对串行存储器电路有了基本的认识，下面介绍具体的串行存储器电路。

11.1.2　串行存储器电路

经过前面对 24LC02B 的介绍，该部分电路的设计应该不难。24LC02B 主要是通过 I^2C 实现与单片机的连接，具体的电路如图 11-2 所示。

由图 11-2 可以看出该电路设计简单。24LC02B 的第 7 管脚（写保护管脚）接地，使该芯片始终处于可以进行读写的状态。在实际设计的时候，也可以将 WP 管脚与单片机的一个一般 I/O 口进行连接，通过单片机来控制 24LC02B 的写保护状态，即：单片机在该管脚输出高电平，24LC02B 就处于写保护状态；单片机在该管脚输出低电平，24LC02B 就不处于写保护状态。在本电路中，为了简化设计，直接将 WP 管脚接地，使 24LC02B 不处于写保护状态。24LC02B 的 A0、A1 和 A2 都接地，表示该器件的地址为 000。由于 I^2C 是总线工作方式，该总线上可以挂有很多器件，所以总线上的每个器件都应该有相应的地址，这样才能实现寻址操作。24LC02B 的 SCL 和 SDA 管脚分别与单片机的 P1.2 和 P1.3 进行连接，连接的方式是 I^2C 总线方式。由于 MSP430 系列单片机里有的单片机没有 I^2C 接口，因此本系统在设计时采用 MSP430 单片机的一般 I/O 口 P1.2 和 P1.3 分别作为 I^2C 总

线的 SCL 和 SDA 线，采用软件来模拟 I^2C 总线，从而实现与 24LC02B 进行接口。在设计时，需要将 SCL 和 SDA 分别通过一个 10kΩ的电阻将其拉高，以满足 I^2C 工作的条件。此外，为了减小电源的干扰，还需要在 24LC02B 芯片的电源输入管脚加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

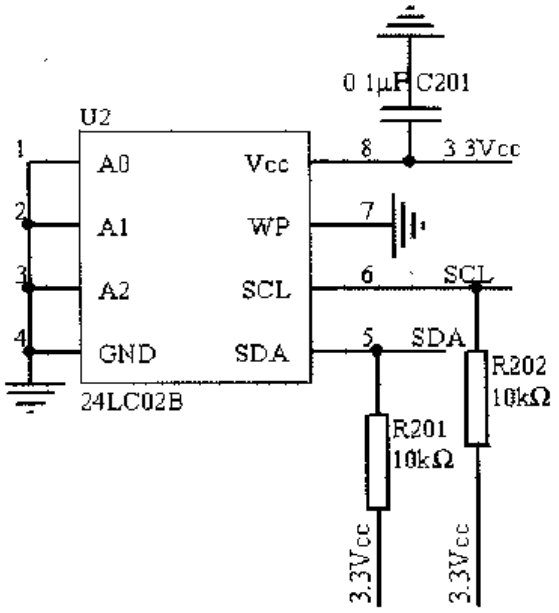


图 11-2　串行存储器电路

单片机电路比较简单，这里就不再进行介绍了。

经过该节的介绍，对系统的硬件设计有了清楚的认识，下一节介绍系统的软件设计。

11.2　软件设计

经过对系统硬件的了解，下面介绍系统的软件设计。系统的软件主要包括 I^2C 协议程序和 24LC02B 读写程序。在具体介绍各个程序之前先介绍一下 I^2C 协议。

11.2.1　I^2C 协议

I^2C 是由 Philips 开发的一种用于内部芯片控制的简单的双向两线串行总线。I^2C 总线支持任何一种芯片制造工艺，并且 Philips 和其他厂商提供了种类非常丰富的 I^2C 兼容芯片。作为一个专利的控制总线，I^2C 已经成为世界性的工业标准。

I^2C 总线支持任何芯片生产过程（如 NMOS、CMOS、双极性）。采用两线，即串行数据线（SDA）和串行时钟线（SCL）在连接到总线的器件间传递信息。每个器件都有一

个惟一的识别地址，而且都可以作为一个发送器或接收器（由器件的功能决定）。除了发送器和接收器外，器件在执行数据传输时也可以被看作是主机或从机。主机是初始化总线的数据传输并产生允许传输的时钟信号的器件，此时任何被寻址的器件都被认为是从机。I^2C 总线是一个多主机的总线，这就是说可以连接多于一个能控制总线的器件到总线。SDA 和 SCL 都是双向线路，都通过一个电流源或上拉电阻连接到正的电源电压，当总线空闲时，这两条线路都是高电平，连接到总线的器件输出级必须是漏极开路或集电极开路才能执行线“与”的功能。下面概述了 I^2C 特征。

- 只需要两线的总线线路：一条串行数据线（SDA）和一条串行时钟线（SCL）。
- 每个连接到总线的器件都可以通过惟一的地址和一直存在的简单的主机/从机关系软件设定地址，主机可以作为主机发送器或主机接收器。
- 它是一个真正的多主机总线，如果两个或更多主机同时初始化数据传输，可以通过冲突检测和仲裁来防止数据被破坏。
- 串行的 8 位双向数据传输位速率在标准模式下可达 100kbit/s，快速模式下可达 400kbit/s，高速模式下可达 3.4Mbit/s。
- 片上的滤波器可以滤去总线数据线上的毛刺波，保证数据完整。
- 连接到相同总线的 IC 数量只受到总线的最大电容 400pF 的限制。

由于连接到 I^2C 总线的器件有不同种类的工艺（如 CMOS、NMOS、双极性），逻辑“0”（低电平）和“1”（高电平）的电平不是固定的，它由 Vcc 的相关电平决定，每传输一个数据位就产生一个时钟脉冲。SDA 线上的数据必须在时钟的高电平周期间保持稳定，数据线的高电平或低电平状态只有在 SCL 线的时钟信号是低电平时才能改变。

I^2C 的时序必须包括起始条件、数据传输、确认和停止条件。下面对这几个部分进行简要介绍。

1．起始条件和停止条件

在 I^2C 总线中惟一出现的是被定义为起始和停止条件。其中一种情况是在 SCL 线是高电平时，SDA 线从高电平向低电平切换，这个情况表示起始条件，所有操作均必须由起始条件开始。当 SCL 是高平时，SDA 线由低电平向高电平切换表示停止条件。在连续读时，如收到一个“停止条件”，则所有读操作将终止，芯片将进入等待模式。起始条件和停止条件一般由主机产生。总线在起始条件后被认为处于忙的状态，在停止条件的某段时间后总线被认为再次处于空闲状态。

2. 数据传输

发送到 SDA 线上的每个字节必须为 8 位。每次传输可以发送的字节数量不受限制，每个字节后必须跟一个响应位。数据传输首先传输的是数据的最高位 MSB。如果从机要在完成一些其他功能后（例如，一个内部中断服务程序）才能接收或发送下一个完整的数据字节，可以使时钟线 SCL 保持低电平迫使主机进入等待状态，当从机准备好接收下一个数据字节并释放时钟线 SCL 后，继续数据传输。

3. 确认

数据传输必须带确认，相关的确认时钟脉冲由主机产生。在确认时钟脉冲期间，发送器释放 SDA 线（高），接收器必须将 SDA 线拉低，使它在这个时钟脉冲的高电平期间保持稳定的低电平，当然必须考虑建立和保持时间。通常被寻址的接收器在接收到的每个字节后必须产生一个确认。当从机不能确认从机地址时（例如，它正在执行一些实时函数而不能接收或发送），从机必须使数据线保持高电平，然后主机产生一个停止条件终止传输或者产生重复起始条件开始新的传输。如果从机接收器确认了从机地址，但是在传输了一段时间后不能接收更多的数据字节，主机必须再一次终止传输。这个情况用从机在第一个字节后没有产生确认来表示，从机使数据线保持高电平，然后主机产生一个停止条件或重复起始条件，如果传输中有主机接收器，它必须确保在从机不产生时钟的最后一个字节不产生一个确认，向从机发送器通知数据结束，从机发送器必须释放数据线，允许主机产生一个停止条件或重复起始条件。

通过对 I^2C 协议的简单介绍，对 I^2C 有了基本的概念，下面给出 I^2C 协议的程序设计。

11.2.2 I^2C 协议的程序实现

I^2C 模块主要包括 SDA 和 SCL 管脚的高低电平的产生、起始条件产生、停止条件产生、确认产生、读取确认、数据发送和数据接收几个部分。下面对各个部分的代码分别进行分析。

1. SDA 高电平的产生

```
void I2C_Set_sda_high( void )
{
    //将 SDA 设置为输出模式
    P1DIR |= SDA;
    //SDA 管脚输出为高电平
```

```
    P1OUT |= SDA;
    _NOP();
    _NOP();
    return;
}
```

其中，_NOP()为标准函数，直接调用就可以了；SDA 为定义的常数。

2. SDA 低电平的产生

```
void I2C_Set_sda_low ( void )
{
    //将 SDA 设置为输出模式
    P1DIR |= SDA;
    //SDA 管脚输出为低电平
    P1OUT &= ~(SDA);

    _NOP();
    _NOP();
    return;
}
```

3. SCL 高电平的产生

```
void I2C_Set_sck_high( void )
{
    //将 SCL 设置为输出模式
    P1DIR |= SCL;
    //SCL 管脚输出为高电平
    P1OUT |= SCL;

    _NOP();
    _NOP();
    return;
}
```

4. SCL 低电平的产生

```
void I2C_Set_sck_low ( void )
{
    //将 SCL 设置为输出模式
    P1DIR |= SCL;
    //SCL 管脚输出为低电平
    P1OUT &= ~(SCL);

    _NOP();
    _NOP();
    return;
```

```
}
```

通过以上的高低电平产生函数，就可以实现 I^2C 的基本操作，比如起始条件产生和停止条件产生，下面介绍具体的实现。

5. 起始条件产生

```
void I2C_START(void)
{
    int i;

    //SDA 管脚输出为高电平
    I2C_Set_sda_high();
    //延迟一点时间
    for(i = 5;i > 0;i--);
    //SCL 管脚输出为高电平
    I2C_Set_sck_high();
    for(i = 5;i > 0;i--);
    //SDA 管脚输出为低电平
    I2C_Set_sda_low();
    for(i = 5;i > 0;i--);
    //SCL 管脚输出为低电平
    I2C_Set_sck_low();
    return;
}
```

结合前面介绍的 I^2C 协议，可以分析上述程序完全按照 I^2C 协议分别在 SDA 和 SCL 管脚上产生相应的高低电平。其中“for(i = 5;i > 0;i--);”主要是为了在高低电平之间插入一定的延迟时间。

6. 停止条件产生

```
void I2C_STOP(void)
{
    int i;

    //SDA 管脚输出为低电平
    I2C_Set_sda_low();
    for(i = 5;i > 0;i--);
    //SCL 管脚输出为低电平
    I2C_Set_sck_low();
    for(i = 5;i > 0;i--);
    //SCL 管脚输出为高电平
    I2C_Set_sck_high();
    for(i = 5;i > 0;i--);
    //SDA 管脚输出为高电平
```

```
    I2C_Set_sda_high();
    for(i = 5;i > 0;i--);
    //SCL 管脚输出为低电平
    I2C_Set_sck_low();
    //延迟一点时间
    Delay_ms(10);
    return;
}
```

在停止函数的结尾，加入一个延时函数，主要是考虑在总线停止后让总线处于一定的空闲状态。上面的延时函数代码如下：

```
void Delay_ms(unsigned long nValue)     //毫秒为单位，8MHz 为主时钟
{
    unsigned long nCount;
    int i;
    unsigned long j;
    nCount = 2667;
    for(i = nValue;i > 0;i--)
    {
        for(j = nCount;j > 0;j--) ;
    }
    return;
}
```

这里的延时函数不是非常精确的，如果需要精确确定的话，可以采用定时器实现，或者采用汇编语言精确知道每条指令的运行周期，这样就可以达到精确目的。在这里由于不需要精确的时间，所以采用的是一个简单的实现。

经过上面的起始条件产生和停止条件产生函数的介绍，可以在该两个函数和基本操作的基础上实现数据的发送和接收，以下分别为数据发送和接收的代码。

7. 数据发送

在下面的程序中，数据发送顺序是从最高位到最低位。

```
void I2C_TxByte(int nValue)
{
    int i;
    int j;

    for(i = 0;i < 8;i++)
    {
        if(nValue & 0x80)
            I2C_Set_sda_high();
        else
            I2C_Set_sda_low();
```

```
        for(j = 30;j > 0;j--);
        I2C_Set_sck_high();
        nValue <<= 1;
        for(j = 30;j > 0;j--);
        I2C_Set_sck_low();
    }

    return;
}
```

8. 数据接收

```
//////////////////////////////////////////
// 接收是从 LSB 到 MSB 的顺序
int  I2C_RxByte(void)
{
    int nTemp = 0;
    int i;
    int j;
    I2C_Set_sda_high();
    P1DIR &= ~(SDA);                    //将 SDA 管脚设置为输入方向
    _NOP();
    _NOP();
    _NOP();
    _NOP();
    for(i = 0;i < 8;i++)
    {
        I2C_Set_sck_high();
        if(P1IN & SDA)
        {
            nTemp |= (0x01 << i);
        }
        for(j = 30;j > 0;j--) ;
        I2C_Set_sck_low();
    }
    return nTemp;
}
```

最后给出确认的函数，具体函数如下。

9. 获得 ACK

```
int  I2C_GetACK(void)
{
    int nTemp = 0;
    int j;

    _NOP();
```

```
    _NOP();
    I2C_Set_sck_low();
    for(j = 30;j > 0;j--);
    P1DIR &= ~(SDA);          //将 SDA 设置为输入方向
    I2C_Set_sck_high();

    for(j = 30;j > 0;j--);
    nTemp = (int)(P1IN & SDA);        //获得数据

    I2C_Set_sck_low();

    return (nTemp & SDA);
}
```

10. ACK 确认

```
void I2C_SetACK(void)
{
    I2C_Set_sck_low();
    I2C_Set_sda_low();

    I2C_Set_sck_high();
    I2C_Set_sck_low();
    return;
}
```

11. NAK 确认

```
void I2C_SetNAk(void)
{
    I2C_Set_sck_low();
    I2C_Set_sda_high();

    I2C_Set_sck_high();
    I2C_Set_sck_low();
    return;
}
```

经过以上函数的介绍，实现了 I^2C 操作模块。

11.2.3　24LC02B 的读写操作

前面介绍的函数实现了 I^2C 协议的基本操作。本节在上面介绍的基础上实现对 24LC02B 的读写操作。

1. 写操作

24LC02B 的写操作有两种形式：单字节写和按页写。下面对单字节写和按页写两种方式分别进行介绍。

（1）单字节写

单字节写就是在指定的地址写入内容。首先单片机发送控制字节，然后发送地址字节，最后输入写的内容。具体程序代码如下：

```
int WriteSingleByte(char nAddr,char nValue)
{
    int nTemp = 0xA0;//写命令
    // 启动数据总线
    I2C_START();
    // 发送控制字节
    I2C_TxByte(nTemp);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 发送地址字节
    I2C_TxByte(nAddr);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 发送数据字节
    I2C_TxByte(nValue);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 停止总线
    I2C_STOP();
    return (nTemp & SDA);
}
```

（2）按页写

按页写是一次写入 8 个字节。按页写操作的第一个字节的操作和单字节写操作是一致的。当写完第一个字节后，单片机继续写下一个内容，在写完最后一个字节后，单片机在总线上产生停止信号。需要注意的是，一次最多只能写入 8 个字节，如果操作多于 8 个字节，则写入的内容会被覆盖。下面为具体的程序代码。

```
int PageWrite(char nAddr,char pBuf[])
{
    int i;
    int nTemp = 0xA0;   //写命令
    // 启动数据总线
    I2C_START();
    // 发送控制字节
    I2C_TxByte(nTemp);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 发送地址字节
    I2C_TxByte(nAddr);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 发送数据字节
    for(i = 0; i < 8;i++)
    {
        I2C_TxByte(pBuf[0]);
        // 等待 ACK
        nTemp = I2C_GetACK();
        if(nTemp & BIT3) return 0;
    }

    // 停止总线
    I2C_STOP();
    return (nTemp & SDA);
}
```

2. 读操作

24LC02B 的读操作有 3 种形式：当前地址读、随机读和顺序地址读。当前地址读需要知道上次读操作后的地址，因此不具有操作独立性，这里不作介绍。下面对随机读和顺序地址读分别作介绍。

（1）随机读

随机读可以读任何地址的数据。首先单片机发送一个控制字节，然后发送地址数据，最后读出该地址的数据。下面给出具体的程序代码。

```
int ReadRandom(char nAddr, char *nValue)
{
    //写命令
    int nTemp = 0xA0;
```

```
    // 启动数据总线
    I2C_START();
    // 发送控制字节
    I2C_TxByte(nTemp);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 发送地址字节
    I2C_TxByte(nAddr);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 启动数据总线
    I2C_START();
    // 发送控制字节
    nTemp = 0xA1;
    I2C_TxByte(nTemp);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    //读取数据
    *nValue = I2C_RxByte();

    // 停止总线
    I2C_STOP();
    //成功返回
    return 1;
}
```

（2）顺序地址读

顺序地址读是连续读出多个字节。它的开始操作和随机读是一样的，在读完一个字节后，对 24LC02B 发送确认（ACK）信号，继续读下一个字节，当读到最后一个字节的时候，单片机需要在总线上产生停止信号。具体程序如下：

```
int ReadSeq(char nAddr, char nValue[], int nLen)
{
    int i;
    //写命令
    int nTemp = 0xA0;
    // 启动数据总线
    I2C_START();
    // 发送控制字节
    I2C_TxByte(nTemp);
    // 等待 ACK
```

```
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 发送地址字节
    I2C_TxByte(nAddr);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    // 启动数据总线
    I2C_START();
    // 发送控制字节
    nTemp = 0xA1;
    I2C_TxByte(nTemp);
    // 等待 ACK
    nTemp = I2C_GetACK();
    if(nTemp & BIT3) return 0;

    //读取数据
    for(i = 0; i < nLen; i++)
    {
        //读一个字节数据
        nValue[i] = I2C_RxByte();
        //发送 ACK
        I2C_SetACK();
    }

    // 停止总线
    I2C_STOP();
    //成功返回
    return 1;
}
```

以上给出了 24LC02B 读写操作的函数。

11.3　实例总结

本章介绍了 24LC02B 与 MSP430 单片机的接口设计。由于采用的是 I^2C 总线接口，接口很简单，很容易实现与不同容量的串行存储器接口，并且不需要改变硬件设计。如果采用更大容量的串行存储器时，可能会需要修改程序，一般只需要修改发送的地址数据就可以了。对于大容量的串行存储器来说，地址字节可能是两个字节，只需要修改该部分代码就可以了。由此可见，本系统在软件和硬件上很容易实现升级。

第 12 章

MSP430 单片机与 NAND FLASH 的接口设计

在单片机应用中，许多领域需要存储大量本地数据。目前，基于 NAND 技术的 FLASH 具有很大的容量。本章介绍 MSP430 单片机与 NAND FLASH（K9F1208U0M）存储器的接口设计，并给出了具体的程序。

12.1 硬件设计

该系统的硬件设计主要是 MSP430 单片机与 K9F1208U0M 芯片的接口设计。由于 MSP430 系列单片机没有数据总线，只能采用一般 I/O 口来模拟数据总线，MSP430 单片机的 I/O 口可以通过端口的方向寄存器来控制输入输出，从而实现数据的读写操作。在介绍具体接口设计之前，先对 K9F1208U0M 芯片进行介绍。

12.1.1 K9F1208U0M 芯片

K9F1208U0M 是采用 NAND 技术实现的 FLASH，它提供按页进行读写等多种数据访问的方法。它只有 8 根数据线，没有专门的地址线，主要通过不同的控制线和发送不同的命令来实现不同的操作。K9F1208U0M 的框图如图 12-1 所示。

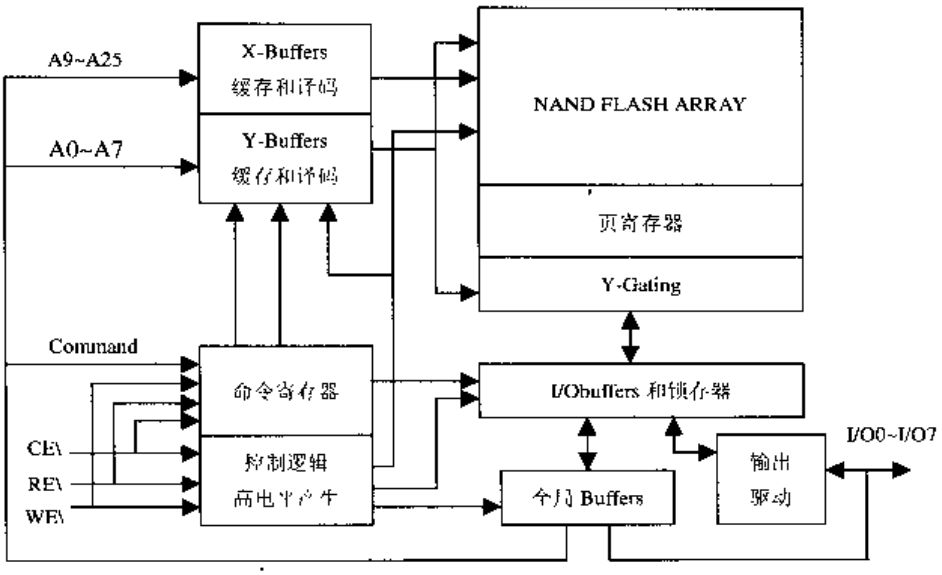


图 12-1　K9F1208U0M 芯片框图

由图 12-1 可以看出，K9F1208U0M 主要有控制逻辑单元、缓存和译码单元、NAND FLASH 存储阵列及输出驱动等几个部分组成。为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 12-2 所示。

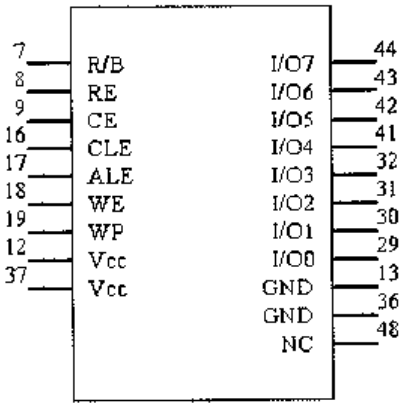


图 12-2　K9F1208U0M 芯片管脚图

由图 12-2 可以看出，该芯片共有 48 个管脚，为了了解各个管脚的功能，下面对具体的管脚进行介绍。

- CLE：命令锁存管脚。该管脚用来表示输入的数据为命令，该管脚高电平有效。当该管脚为高电平的时候，在 WE 信号的上升延时输入的数据为命令数据。
- ALE：地址锁存管脚。该管脚用来表示输入的数据为地址，该管脚高电平有效。当该管脚为高电平的时候，在 WE 信号的上升延时输入的数据为地址数据。
- CE：K9F1208U0M 芯片选择管脚。该管脚低电平有效。当该管脚为低电平的时候，

选通 K9F1208U0M 芯片，否则 K9F1208U0M 芯片不工作。

- RE：K9F1208U0M 芯片读使能管脚。该管脚低电平有效。当该管脚为低电平的时候，对 K9F1208U0M 进行读操作。
- WE：K9F1208U0M 芯片写使能管脚。该管脚低电平有效。当该管脚为低电平的时候，对 K9F1208U0M 芯片进行写操作。
- I/O 口（I/O0~I/O7）：K9F1208U0M 芯片的数据线，用这些数据线来完成地址数据、命令数据和内容数据的输入或者输出。当 K9F1208U0M 芯片的片选信号为高电平的时候，数据线处于高阻状态。
- WP：写保护管脚。该管脚低电平有效。当该管脚为低电平时，写保护起作用。
- R/B：准备好/忙管脚，用来表示 FLASH 芯片是否准备好。
- Vcc：电源管脚。
- GND：接地管脚。

经过对 K9F1208U0M 芯片的介绍，对该芯片有了基本的认识，下面介绍硬件电路的具体设计。

12.1.2 接口电路设计

由于 MSP430 单片机没有数据总线，因此利用 MSP430 单片机的一般 I/O 口来模拟总线。由于 MSP430 单片机能通过端口的方向寄存器来设置端口的输入输出方向，因此能很好地完成总线的数据读写功能。MSP430 单片机的一般 I/O 口与 K9F1208U0M 芯片的相应控制线接口，完成相应的控制功能。如图 12-3 所示给出了 MSP430 单片机与 K9F1208U0M 芯片的接口电路图。

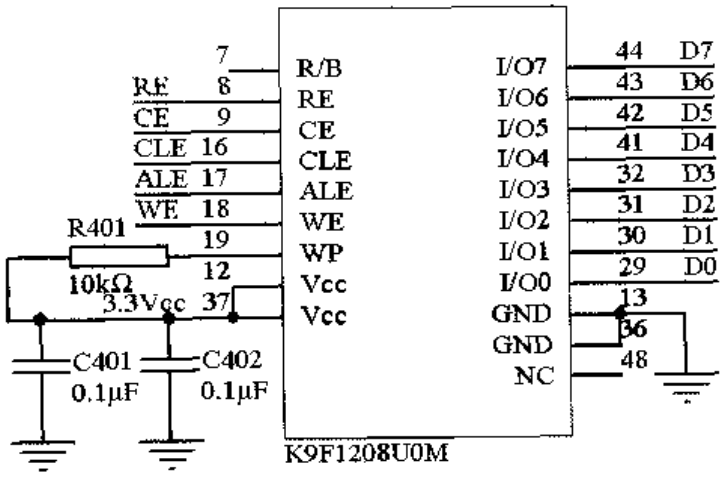


图 12-3 MSP430 单片机与 K9F1208U0M 芯片的接口电路图

从图 12-3 可以看出：MSP430 单片机与 K9F1208U0M 芯片的接口非常简单。将 MSP430 单片机的 P5 口与数据总线相连，P4 口与一些控制线相连。在上面电路设计中，R/B 管脚没有使用，该管脚主要判断数据是否准备好，如果数据准备好，该管脚输出一个低电平，在实际处理中可以将该管脚与 MSP430 的中断 I/O 口连接，采用硬件方式进行判断是否准备好。此外，数据准备好状态的判断也可以通过软件来实现，因此在设计硬件时该管脚可以不使用。由于 WP 管脚是输入管脚，并且低电平有效，因此为了使 FLASH 始终可以进行读写，这样将该管脚通过一个电阻上拉。在实际设计时，也可以与单片机的一般 I/O 口连接，通过单片机来控制是否写保护。

12.1.3　单片机电路

单片机电路非常简单，除了必要的振荡电路外，主要就是与 K9F1208U0M 芯片的接口。单片机的 P5 口与 K9F1208U0M 芯片的数据线进行连接，从而实现数据的读写。单片机的 P4 口的某些管脚与 K9F1208U0M 芯片的控制管脚（如 RE、CLE 等管脚）进行连接，从而实现单片机对 K9F1208U0M 芯片进行不同的操作（如读、写、复位等操作）。如图 12-4 所示为单片机电路图。

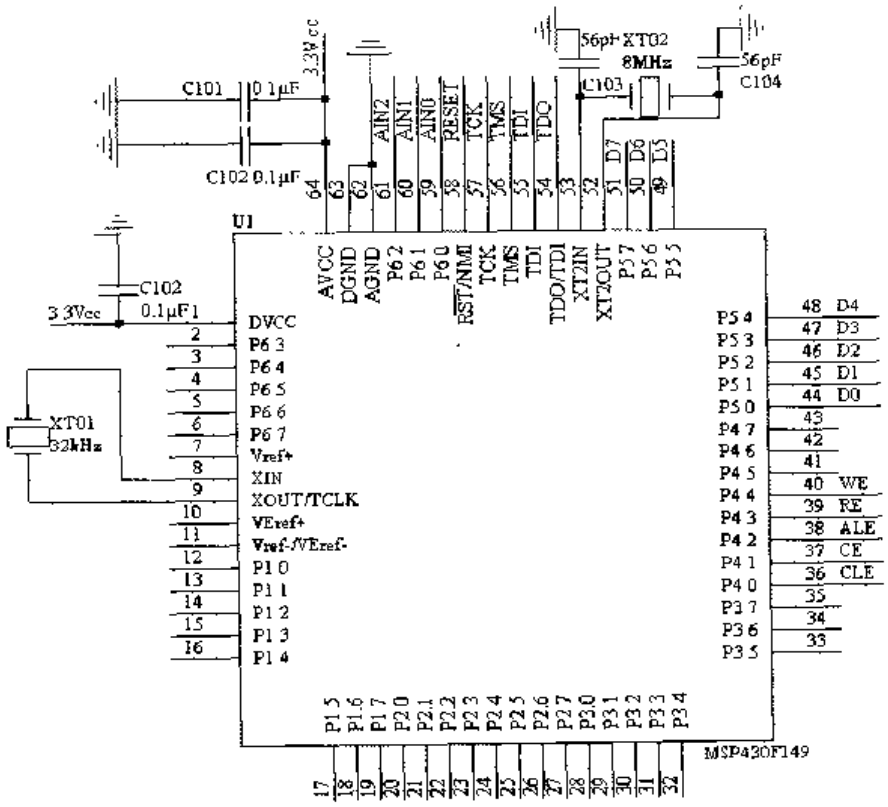


图 12-4　单片机电路

考虑到电源的输入纹波对单片机的影响，在电源的管脚增加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。经过上面的介绍，对系统的硬件接口有了比较清楚的认识，下面结合该硬件接口电路来介绍接口电路的软件设计。

12.2 软件设计

系统的软件设计主要是对 K9F1208U0M 进行读写的操作，在介绍具体的程序之前，先对 K9F1208U0M 芯片的操作进行简单的介绍。

12.2.1 K9F1208U0M 芯片操作

K9F1208U0M 芯片的地址分为行地址和列地址，K9F1208U0M 芯片以字节为单位，这样 K9F1208U0M 芯片的存储阵列可以看成一个三维模型。K9F1208U0M 芯片由很多的页（Page）组成，其中 32 页组成一块（Block），这样整个 K9F1208U0M 芯片可以看成由很多的块组成。K9F1208U0M 芯片的一页由三个区域组成，三个区域分别是第一半区、第二半区和备用区。第一半区和第二半区分别有 256 个字节，用来存放数据，备用区有 16 个字节，用来存放备注信息。K9F1208U0M 芯片通过列地址（A0~A7）来实现对页的某一个地址的寻址，由于 A0~A7 表示数的范围是 0~256，因此必须结合不同的命令才能实现对一页的任意位置进行访问；不同的命令确定了地址位 A8 的值，因此在地址数据中，用户输入的 A8 的值会被忽略。K9F1208U0M 芯片的具体每一页的地址通过行地址（A9~A25）来表示，这样通过利用行地址和列地址并且结合相应的命令就能实现对 K9F1208U0M 芯片任意地址进行访问。

K9F1208U0M 芯片只有 8 根数据总线，然而 K9F1208U0M 芯片却需要完成读、写和擦除等不同的操作，因此 K9F1208U0M 芯片提供了不同的控制线，借助这些不同的控制线可以使 K9F1208U0M 芯片有不同的操作模式。K9F1208U0M 芯片有写模式、读模式、命令模式和地址输入等模式，这些不同的模式通过不同的控制线来完成相应的操作。表 12-1 给出了在 K9F1208U0M 芯片的几种工作模式下控制线的输入情况。

表 12-1 K9F1208U0M 芯片的模式选择

CLE	ALE	CE	WE	RE	WP	模　式
H	L	L		H	X	读模式的命令输入
L	H	L		H	X	读模式的地址输入
H	L	L		H	H	写模式的命令输入

续表

CLE	ALE	CE	WE	RE	WP	模　式
L	H	L	⎍	H	H	写模式的地址输入
L	L	L	⎍	H	H	数据输入
L	L	L	H	⎍	X	序列读和数据输出
X	X	L	X	X	X	读忙
X	X	X	X	X	H	写忙
X	X	X	X	X	H	擦除忙
X	X	X	X	X	L	写保护
X	X	H	X	X	0V/Vcc	停止工作

表 12-1 中的 H 表示逻辑高电平，L 表示逻辑低电平，X 表示输入任意。通过表 12-1 可以看出，只要适当设置控制线的输入状态就能完成相应的操作。

对 K9F1208U0M 芯片的操作具体有写操作、读操作和擦除等操作，其中写操作可以是单字节写、多字节写和页写等操作，读操作也可以是单字节读、多字节读和页读等操作，对于擦除操作只能是块擦除操作。具体对 K9F1208U0M 芯片的操作是通过向 K9F1208U0M 芯片发送不同的命令来实现不同的操作的。表 12-2 给出了 K9F1208U0M 芯片操作的全部命令集，除了表中列出来的命令外，其余所有的命令都是非法的，K9F1208U0M 芯片不能识别那些表中以外的命令。

表 12-2　K9F1208U0M 芯片的操作命令

功　能	第一周期	第二周期
读命令 1	00/01	
读命令 2	0x50	
读 ID 命令	0x90	
复位命令	0xff	
页写命令	0x80	0x10
多块编程命令	0x80	0x15
块擦除命令	0x60	0xd0
多块擦除命令	0x60...0x60	0xd0
读状态命令	0x70	
读多块状态命令	0x71	

表 12-2 中的读命令 1 的 00 表示操作页的第一半区，01 表示操作页的第二半区。读命令 2 表示操作页的备用区。通过将表中列出来的这些命令和前面介绍的模式选择结合起来就可以完成相应的操作。

通过以上的介绍，对 K9F1208U0M 芯片的操作有了基本的认识，下面介绍具体的程

序实现。

12.2.2 控制线模拟程序

该部分软件主要完成端口的初始化功能、控制线的高电平产生和控制线的低电平产生。分别根据不同的操作来产生相应控制线的高电平或者低电平，这样就能产生相应的控制信号和读写信号。下面就各个部分进行具体介绍。

1. 端口初始化

端口初始化的主要作用是为控制信号线设置正确的输入输出方向。另外，MSP430 单片机的一般 I/O 口也可以作为第二功能脚使用，因此在设置的时候需要将端口的管脚设置为一般 I/O 口。具体的程序如下：

```
void SM_Port_Init(void)
{
    P4DIR = 0;
    P4DIR |= BIT0;          //设置 CLE 为输出管脚
    P4DIR |= BIT1;          //设置 CE~为输出管脚
    P4DIR |= BIT2;          //设置 ALE 为输出管脚
    P4DIR |= BIT3;          //设置 RE~为输出管脚
    P4DIR |= BIT4;          //设置 WE~为输出管脚
    P1DIR &= ~(BIT1);       //设置 R/B 为输入管脚
    //将 P4、P5 口的管脚设置为一般 I/O 口
    P4SEL = 0;
    P5SEL = 0;
    return;
}
```

2. 控制线的模拟

控制线模拟程序主要在控制管脚（如 RE、WE 等管脚）产生高电平或者低电平，并且需要设置正确的输入输出方向。下面以 CLE 管脚为例给出具体的程序。

```
void CLE_Enable(void)
{
    P4OUT |= BIT0; // 产生高电平

    return;
}
void CLE_Disable(void)
{
    P4OUT &= ~(BIT0); // 产生低电平
    return;
}
```

对于其他控制管脚的模拟程序与上面的程序相似，这里限于篇幅不再介绍。

12.2.3　数据读操作程序

对 K9F1208U0M 的读操作必须严格按照该芯片的读时序进行操作，具体的读时序这里不给出，读者可以查看该芯片的数据手册，下面给出读操作的具体程序。

```
void PageRead(int nCol,unsigned long nRow,char *pBuf)
{
    int i;
    int j;
    unsigned char nADD1;
    unsigned char nADD2;
    unsigned char nADD3;

    //处理最高地址的时候必须注意的是其余没有用的位必须是 0
    nADD1 = (unsigned char)((nRow & 0x000000ff) >> 0);
    nADD2 = (unsigned char)((nRow & 0x0000ff00) >> 8);
    nADD3 = (unsigned char)((nRow & 0x00010000) >> 16);

    CE_Enable();

    P5DIR = 0xff; // 设置 P5 口为输出方向

    CLE_Enable();
    WE_Enable();
    P5OUT = 0x00; // 输出读命令代码 0x00
    WE_Disable();
    CLE_Disable();
    // 发送列起始地址
    ALE_Enable();
    WE_Enable();
    P5OUT = (unsigned char)(nCol);
    WE_Disable();
    // 发送行地址第一字节
    WE_Enable();
    P5OUT = (unsigned char)(nADD1);
    WE_Disable();
    // 发送行地址第二字节
    WE_Enable();
    P5OUT = (unsigned char)(nADD2);
    WE_Disable();
    // 发送行地址第三字节
    WE_Enable();
    P5OUT = (unsigned char)(nADD3);
    WE_Disable();
```

```
    ALE_Disable();

    //延迟一点时间，等待 R/B 低电平
    for(i = 100;i > 0;i--) ;

    P5DIR = 0; // 设置 P5 口为输入方向
    for(j = 0;j < 528;j++)
    {
        RE_Enable();
        pBuf[j] = P5IN;
        RE_Disable();
    }

    CE_Disable();
    return;
}
```

上面给出的是按页读的程序代码。为了增加处理的灵活性，读操作也可以按字节进行读，与按页读不同的是，按字节读必须分别传送不同的读命令（0x00，0x01，x50）来指定操作的是第一半区还是第二半区，或者是备用区。由于按字节操作和按页操作有很大的相似性，在这里就不进行详细叙述了。

12.2.4 数据写操作程序

对 K9F1208U0M 的写操作必须严格按照该芯片的写时序进行操作，具体的写时序这里不给出，读者可以查看该芯片的数据手册，下面给出写操作的具体程序。

```
int PageWrite(int nCol,unsigned long nRow,char *pBuf)
{
    int nTemp = 0;
    int i;
    int j;
    unsigned nADD1;
    unsigned nADD2;
    unsigned nADD3;
    //处理最高地址的时候必须注意的是其余没有用的位必须是 0
    nADD1 = (unsigned char)((nRow & 0x000000ff) >> 0);
    nADD2 = (unsigned char)((nRow & 0x0000ff00) >> 8);
    nADD3 = (unsigned char)((nRow & 0x00010000) >> 16);

    CE_Enable();

    P5DIR = 0xff; //设置 P5 口为输出方向

    CLE_Enable();
```

```
WE_Enable();
P5OUT = 0x80; //页写命令
WE_Disable();
CLE_Disable();

ALE_Enable();
WE_Enable();
P5OUT = (unsigned char)(nCol); // 行的起始地址
WE_Disable();
// 发送行地址第一字节
WE_Enable();
P5OUT = nADD1;
WE_Disable();
// 发送行地址第二字节
WE_Enable();
P5OUT = nADD2;
WE_Disable();
// 发送行地址第三字节
WE_Enable();
P5OUT = nADD3;
WE_Disable();
ALE_Disable();
//写入数据
for(j = 0;j < 528;j++)
{
     WE_Enable();
     P5OUT = pBuf[j];
     WE_Disable();
}

CLE_Enable();
WE_Enable();
P5OUT = 0x10; // 写操作确认命令
WE_Disable();
CLE_Disable();

//延迟一点时间，等待 R/B 低电平
for(i = 100;i > 0;i--) ;

CLE_Enable();
WE_Enable();
P5OUT = 0x70;
WE_Disable();
CLE_Disable();

P5DIR = 0x00; //设置 P5 口为输入方向
// 读状态寄存器
for(i = 1000;i > 0;i--)
```

```
    {
        RE_Enable();
        nTemp = P5IN;
        RE_Disable();
        if(nTemp == 0xc0) break;
    }

    if(nTemp == 0xc0) return 1;
    else return 0;
}
```

上面给出的是按页写的程序代码。为了增加处理的灵活性，写操作也可以按字节进行写，与按页写不同的是，按字节写必须分别设置指针（0x00，0x01，x50）来指定操作的是第一半区还是第二半区，或者是备注区。由于按字节操作和按页操作有很大的相似性，在这里就不进行详细叙述了。

12.2.5 擦除程序

为了能够对 K9F1208U0M 芯片进行正确写数据，必须满足的条件是即将写入的单元内容必须是 0xFF。K9F1208U0M 芯片提供了擦除操作，通过擦除操作能将已有内容的单元清除，并使里面的内容为 0xFF，从而确保正确的写入数据。与读操作和写操作不同的是：擦除必须是以块（Block）为单位。下面给出擦除操作的具体程序。

```
int BlockErase(unsigned long nAddr)
{
    int nTemp = 0;
    int i;
    unsigned char nADD1;
    unsigned char nADD2;
    unsigned char nADD3;
    //处理最高地址的时候必须注意的是其余没有用的位必须是 0
    nADD1 = (unsigned char)((nAddr & 0x000000ff) >> 0);
    nADD2   (unsigned char)((nAddr & 0x0000ff00) >> 8);
    nADD3 = (unsigned char)((nAddr & 0x00010000) >> 16);

    CE_Enable();

    P5DIR = 0xff; //设置 P5 口为输出方向

    CLE_Enable();
    WE_Enable();
    P5OUT = 0x60; //输出块擦除命令
    WP_Disable();
    CLE_Disable();
```

```
    // 发送行地址第一字节
    ALE_Enable();
    WE_Enable();
    P5OUT = (unsigned char)(nADD1);
    WE_Disable();
    // 发送行地址第二字节
    WE_Enable();
    P5OUT = (unsigned char)(nADD2);
    WE_Disable();
    // 发送行地址第三字节
    WE_Enable();
    P5OUT = (unsigned char)(nADD3);
    WE_Disable();
    ALE_Disable();
    // 发送擦除确认命令
    CLE_Enable();
    WE_Enable();
    P5OUT = 0xd0;
    WE_Disable();
    CLE_Disable();

    //延迟一点时间，等待 R/B 低电平
    for(i = 200;i > 0;i --);
    // 发送读状态寄存器命令
    CLE_Enable();
    WE_Enable();
    P5OUT = 0x70;
    WE_Disable();
    CLE_Disable();

    P5DIR = 0;  //设置 P5 口为输入方向
    // 读状态寄存器的内容
    RE_Enable();
    nTemp = P5IN;
    RE_Disable();

    CE_Disable();

    if(nTemp & 0x01) return 0;
    else return 1;
}
```

12.2.6　测试程序

经过上面的介绍，基本可以实现 K9F1208U0M 的操作。下面给出一个简单的测试程序，具体代码如下：

```
int main(void)
{
    unsigned long nAddr;
    int j;
    int n;
    int nTemp;
    char pBuf0[528];
    char pBuf1[528];
    int nCount_Err;
    int nLen;
    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;
    // 关闭中断
    _DINT();

    // 初始化
    Init_CLK();
    SM_Port_Init();
    // 打开中断
    _EINT();

    // 擦除块地址为 1024 的块
    nAddr = 1024;
    BlockErase(nAddr);
    // 延迟一点时间
    for(j = 1000;j > 0;j--) ;
    // 判断擦除的块是否为 0xFF，起始页地址为 1024
    for(j = 0;j < 528;j++) pBuf0[j] = j;
    // 按页写入数据，地址为 1024
    PageWrite(0,1024,pBuf0);
    // 延迟一点时间
    for(j = 1000;j > 0;j--) ;
    //按页读操作
    PageRead(0,1024,pBuf1);
    //比较读出数据和写入数据是否相等
    nCount_Err = 0;
    for(j = 0;j < 528;j++)
    {
        if(pBuf0[j] != pBuf1[j])
        {
            nCount_Err += 1;
            break;
        }
    }

    return 0;
}
```

12.3　实例总结

本章主要详细介绍了 MSP430 单片机与 NAND FLASH 芯片 K9F1208U0M 的硬件设计和软件设计。由于该系统具有存储量大、功耗低等特点，因此该系统能够在很多数据存储领域得到大量的应用。通过本章的介绍，使读者能够掌握整个系统的软件和硬件设计，也使读者可以按照自己的要求适当修改硬件和软件就能实现自己的存储系统。

第四篇

采集与测量

第 13 章

A/D 转换器 TLV2541 的设计与应用

在单片机应用中，很多领域都涉及到模拟数据的采集。在 MSP430 系列单片机中，有的单片机自身不带 A/D 转换通道。对于自身不带 A/D 转换通道的单片机，就需要外接 A/D 芯片来实现数据采集。本章介绍 MSP430 单片机与 A/D 转换芯片 TLV2541 的接口设计，并给出相应的程序。

13.1 硬件接口电路设计

硬件接口电路相对比较简单，主要就是单片机与 TLV2541 芯片的连接。为了便于理解接口，在介绍接口之前，先简要介绍一下 TLV2541 芯片。

13.1.1 TLV2541 芯片

TLV2541 是一款高性能、低功耗的 12 位精度的 A/D 转换芯片，它具有很小的封装，只有 8 个管脚。该芯片具有以下特性：

- 最大转换速率达到 200KSPS。内部有转换时钟。
- 采用 SPI 串口与其他芯片接口，并能与 DSP 进行接口。
- 单电源供电，并且有较宽的供电范围，供电电压范围为 2.7V~5.5V。
- 低功耗，并且能自动降低功耗。在 2.7V 工作电压时，电流只有 1mA。处于低功耗状态情况下，在 2.7V 工作电压时，电流只有 2.2μA。

为了增加对该芯片的认识，下面给出该芯片的框图，如图 13-1 所示。

由图 13-1 可以看出，该芯片内部结构简单。内部提供振荡器和转换时钟信号。该芯片通过 SPI 串口与单片机进行连接。为了便于进行硬件电路设计，下面给出该芯片的管脚图，如图 13-2 所示。

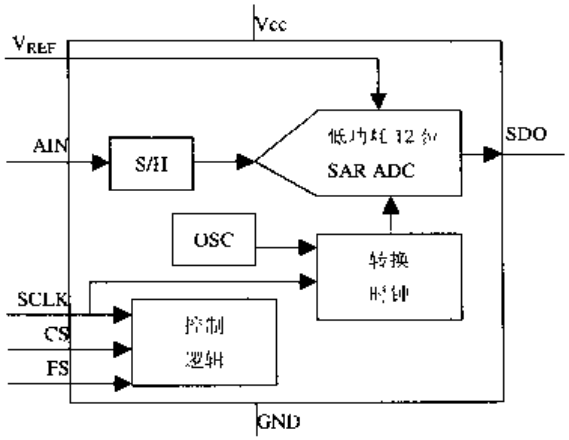


图 13-1　TLV2541 芯片框图

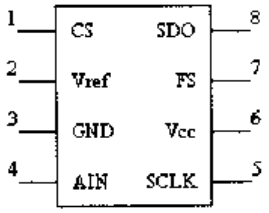


图 13-2　TLV2541 管脚图

由图 13-2 可以看出，该芯片只有 8 个管脚，只需要简单的外围电路。下面对具体的管脚进行介绍。

- CS：片选信号。低电平有效。
- Vref：外部参考管脚。
- GND：接地管脚。
- AIN：模拟输入管脚。
- SCLK：SPI 串口的串行时钟输入管脚。
- Vcc：电源管脚。
- FS：DSP 帧同步输入管脚。指示串行数据帧的开始，如果该管脚不使用，则应该将该管脚通过 10kΩ的电阻拉高。
- SDO：SPI 串口的数据输出管脚。

经过对 TLV2541 芯片的介绍，对该款 A/D 转换芯片有了基本的认识，下面介绍硬件电路的具体设计。

13.1.2　接口电路设计

硬件的接口电路主要就是 SPI 接口的设计。在 MSP430 系列单片机中，很多系列单片机都有串口模块，并且串口模块都可以工作在 UART 方式和 SPI 方式，因此 MSP430 单片机很容易通过片内的串口实现与 TLV2541 芯片进行接口。如图 13-3 所示为接口电路图。

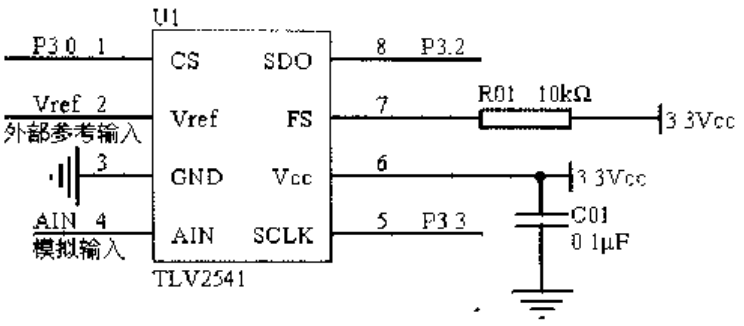


图 13-3　MSP430 单片机与 TLV2541 的接口电路

由图 13-3 可以看出，整个接口电路很简单。单片机的 P3.0 管脚与 TLV2541 的 CS 管脚连接，实现片选控制。单片机的 P3.2 管脚和 P3.3 管脚分别与 TLV2541 的 SDO 管脚和 SCLK 管脚进行连接，实现 SPI 口的数据通信功能。TLV2541 的 Vref 管脚外接参考源，TLV2541 的 AIN 管脚接模拟输入信号。由于该应用中 TLV2541 的 FS 管脚不使用，因此需要通过一个 10kΩ的电阻将该管脚拉高。另外，为了减小干扰，需要在电源管脚处外加一个 0.1μF 的滤波电容用来进行滤波处理。

13.1.3　单片机电路

单片机电路非常简单，单片机的 P3.0、P3.2 和 P3.3 与 TLV2541 进行接口，另外，单片机必须有相应的振荡电路以使单片机能够进行工作。如图 13-4 所示为单片机电路图。

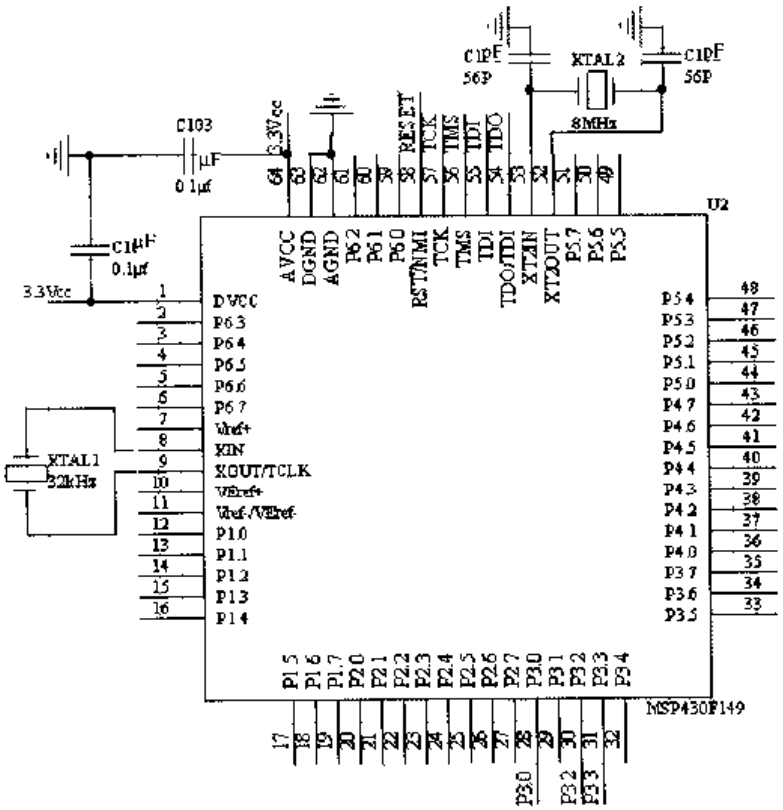


图 13-4　单片机电路

经过上面的介绍，对系统的硬件接口有了比较清楚的认识，下面结合该硬件接口电路来介绍接口电路的软件设计。

13.2　软件设计

整个软件设计相对很简单，主要包括时钟初始化、端口初始化、SPI 串口初始化、A/D 转换等模块，下面就各个部分的程序分别进行详细介绍。

1. 时钟初始化程序

这部分程序主要是选择单片机的 SMCLK 和 MCLK 的信号源，从而提供单片机的某些外设工作的时钟。下面为时钟的初始化程序。

```
void Init_CLK(void)
{
    unsigned int i;
    BCSCTL1 = 0X00; //将寄存器的内容清零
                    //XT2 震荡器开启
                    //LFXT1 工作在低频模式
                    //ACLK 的分频因子为 1

    do
    {
        // 清除 OSCFault 标志
        IFG1 &= ~OFIFG;
        for (i = 0x20; i > 0; i--);
    }
    while ((IFG1 & OFIFG) == OFIFG); // 如果 OSCFault =1

    BCSCTL2 = 0X00;          //将寄存器的内容清零
    BCSCTL2 += SELM1;        //MCLK 的时钟源为 TX2CLK，分频因子为 1
    BCSCTL2 += SELS;         //SMCLK 的时钟源为 TX2CLK，分频因子为 1
}
```

2. 端口初始化程序

该部分程序主要设置 P3.0、P3.1 和 P3.3 管脚的属性。具体的程序如下：

```
void Init_Port(void)
{
    //将 P3 口所有的管脚在初始化的时候设置为输入方式
    P3DIR = 0;

    //将 P3 口所有的管脚设置为一般 I/O 口
    P3SEL = 0;
```

```
    //P3.1、P3.2、P3.3 被分配为 SPI 口
    P3SEL = BIT3 + BIT2 + BIT1;
    //P3.0 作为输出管脚
    P3DIR |= BIT0;
    //P3.3 作为输出管脚
    P3DIR |= BIT3;
    //P3.0 输出高电平，使 TLV2541 不被选通
    P3OUT |= BIT0;
    return;
}
```

在上面的程序中，需要将 P3.1、P3.2 和 P3.3 三个管脚设置成 SPI 口的管脚，并且将 P3.0 和 P3.3 设置成输出管脚。

3. SPI 串口初始化程序

该部分程序设置 SPI 串口的相应参数，如传输速率、传输格式等。具体程序如下：

```
void Init_SPI (void)
{
    //SPI0 模块允许
    ME1 |= USPIE0;
    //将寄存器的内容清零
    U0CTL = 0X00;
    //数据为 8 比特，选择 SPI 模式，单片机为主机模式
    U0CTL |= CHAR + SYNC + MM;

    //将寄存器的内容清零
    U0TCTL = 0X00;
    // 时钟源为 SMCLK，选择 3 线模式
    U0TCTL = CKPH + SSEL1 + SSEL0 + STC;

    //传输时钟为 SMCLK / 4
    UBR0_0 = 0X02;
    UBR1_0 = 0X00;
    //调整寄存器，没有调整
    UMCTL_0 = 0X00;
}
```

在上面的程序中，首先通过“ME1 |= USPIE0;”来使能 SPI 模块，然后分别设置串口通信的帧格式和通信速率。

4. 转换程序

该部分程序主要根据 TLV2541 的工作时序来完成转换。MSP430 的转换程序主要包括下面 3 个部分。第一，MSP430 单片机首先在 P3.0 管脚上送出低电平，选通 TLV2541。第

二，MSP430 单片机需要给 TLV2541 提供转换的时钟信号。第三，MSP430 单片机通过串口读取数据，在读取数据完毕后，在单片机的 P3.0 管脚输出高电平，使 TLV2541 处于非选通状态。根据上面描述的过程，下面给出具体的实现代码。

```
int convert (void)
{

    unsigned int hi_byte;
    unsigned int lo_byte;
    unsigned int temp;
    unsigned int trash;

    //选通 TLV2541
    CS_Enable();
    // 发送数据，以便产生时钟信号送给 TLV2541
    U0TXBUF = 0x00;
    //等待传输完成
    complete();
    //读取转换的高字节
    hi_byte = U0RXBUF;
    //将高字节左移 8 位
    hi_byte = hi_byte << 8;
    // 发送数据，以便产生时钟信号送给 TLV2541
    U0TXBUF = 0x00;
    //等待传输完成
    complete();
    //读取转换的低字节
    lo_byte = U0RXBUF;

    //不选通 TLV2541
    CS_Disable();

    //当 TLV2541 处于不选通状态时，必须保持 1 个时钟的下降沿
    // 发送数据，以便产生时钟信号送给 TLV2541
    U0TXBUF = 0x00;
    //等待传输完成
    complete();
    //无用数据
    trash = U0RXBUF;

    //将高字节和低字节组合起来形成一个字
    temp = hi_byte + lo_byte;
    //由于 TLV2541 为 12 位采集，因此右移 4 位
    temp = temp >> 4;

    //延迟一点时间
    Delay();
```

```
    //返回采集得到的结果
    return temp;
}
```

上面的程序代码使用查询中断标志的方式来判断传输数据是否完成。在上面的处理程序里，由于 TLV2541 在片选信号输入为高电平时，必须保持一个时钟的下降沿，所以在读取完采集的数据后，仍然发送数据来产生时钟，读取一个无用的字节。

在上面的程序中用到了一些函数，用到的函数代码如下。

```
void CS_Enable(void)
{
    //P3.0 输出低电平
    P3OUT &=~(BIT0);
    return ;
}
void CS_Disable(void)
{
    //P3.0 输出高电平
    P3OUT|=BIT0;
    return ;
}
void Delay(void)
{
    int i;
    for(i = 1000; i > 0; i--) ;
    return ;
}
void complete(void)
{
    //等待传输完成的中断标志
    do
    {
        IFG1 &=~ URX1IFG0;
    }
    while (URX1IFG0 & IFG1);
}
```

下面给出一个简单的测试程序，代码如下。

```
void main()
{
    int nRes[512];
    int i;

    //初始化时钟
    Init_CLK();
```

```
    //端口初始化
    Init_Port();
    //SPI 初始化
    Init_SPI();

    for(i = 0; i < 512; i++)
    {
        nRes[i] = convert();
    }

    return;
}
```

上面的程序只是简单地给出数据采集的例子，读者可以根据自己的需要，按照自己系统的处理流程来实现代码。

13.3　实例总结

本章介绍了 MSP430 单片机与 TLV2541 的接口设计，并给出了相应的程序。TLV2541 通过 SPI 口与单片机进行连接，使系统的复杂度非常小。在程序设计时，需要注意的是必须满足 TLV2541 的工作时序。通过本章的介绍，使读者能够进行一个简单的模拟数据采集系统的设计，读者也可以扩展硬件和软件功能，设计出满足自己系统需要的模拟数据采集系统。

第 14 章

D/A 转换器 DAC8830 接口设计与应用

在单片机应用中，某些便携式设备或者自动测量设备中需要输出波形显示，即需要单片机输出模拟信号。MSP430 系列单片机中，有的单片机自身不带 D/A 转换通道。对于自身不带 D/A 转换通道的单片机，如果需要模拟输出，就需要外接 D/A 芯片来实现模拟输出。本章介绍 MSP430 单片机与 D/A 转换芯片 DAC8830 的接口设计，并给出相应的程序。

14.1 硬件接口电路设计

硬件接口电路相对比较简单，主要就是单片机与 DAC8330 芯片的连接。为了便于理解接口电路，在介绍接口之前，先简要介绍一下 DAC8830 芯片。

14.1.1 DAC8830 芯片

DAC8830 是一款高性能、低功耗的 16 位精度的 D/A 转换芯片，该芯片为电压输出。该芯片具有以下特性：

- 16 位 D/A 转换，具有非常高的精度。输入有直接的光耦接口。
- 采用 SPI 串口与其他芯片接口，SPI 的 SCLK 最高可以达到 50MHz。
- 单电源供电，并且有较宽的供电范围，供电电压范围为 2.7V~5.5V。

- 低功耗。在 3V 电压工作时，功率只有 15μW。

为了增加对该芯片的认识，下面给出该芯片的框图，如图 14-1 所示。

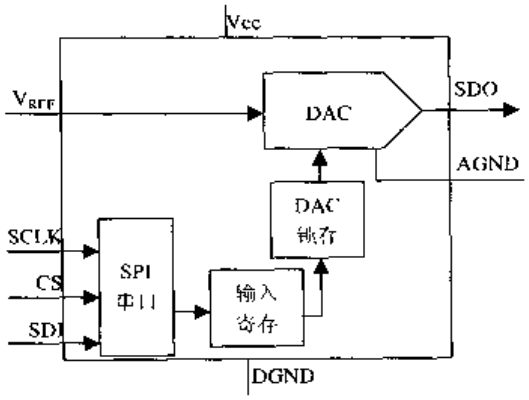


图 14-1　DAC8830 芯片框图

由图 14-1 可以看出，该芯片内部结构简单。该芯片通过 SPI 串口与单片机进行连接。为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 14-2 所示。

由图 14-2 可以看出，该芯片只有 8 个管脚，这样使用起来简单，只需要简单的外围电路。下面对具体的管脚进行介绍。

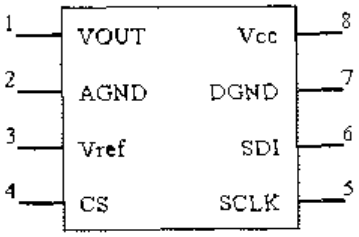


图 14-2　DAC8830 管脚图

- CS：片选信号。低电平有效。
- Vref：外部参考管脚。
- DGND：数字接地管脚。
- AGND：模拟接地管脚。
- SCLK：SPI 串口的串行时钟输入管脚。
- Vcc：电源管脚。
- VOUT：模拟输出。输出为 D/A 转换的结果。
- SDI：SPI 串口的数据输入管脚。

经过对 DAC8830 芯片的介绍，对该款 D/A 转换芯片有了基本的认识，下面介绍硬件电路的具体设计。

14.1.2　接口电路设计

硬件的接口电路相对简单，主要就是 SPI 接口的设计。在 MSP430 系列单片机中，MSP430F12X、MSP430F13X、MSP430F14X、MSP430F44X 等系列单片机都有串口模块，并且串口模块都可以工作在 UART 方式和 SPI 方式，因此 MSP430 单片机很容易通过片内的串口实现与 DAC8830 芯片进行接口。如图 14-3 所示为接口电路图。

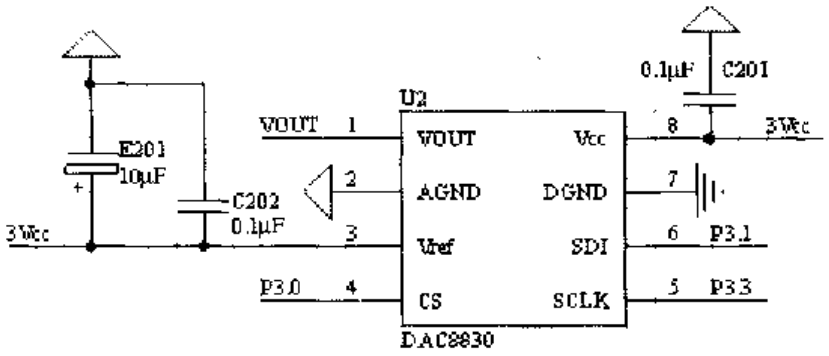


图 14-3　DAC8830 接口电路

由图 14-3 可以看出，整个接口电路很简单。单片机的 P3.0 管脚与 DAC8830 芯片的 CS 管脚连接，实现片选控制。单片机的 P3.1 管脚和 P3.3 管脚分别与 DAC8830 芯片的 SDI 管脚和 SCLK 管脚进行连接，实现 SPI 口的数据通信功能。DAC8830 的 Vref 管脚外接参考源，在外接参考电源的时候，为了减小干扰，在电压参考管脚需要进行滤波处理，图中的 E201 和 C202 就是起滤波作用。DAC8830 的 VOUT 管脚输出信号转换后的模拟信号。由于 DAC8830 芯片有单独的模拟地和数字地，因此在设计的时候需要将模拟地和数字地分开。另外，为了减小干扰，需要在电源管脚处外加一个 0.1μF 的滤波电容用来进行滤波处理。

14.1.3　电源电路

由于本系统使用的是 3V 电压工作，因此与第 2 章的电源电路不一致。这里采用 TPS76030 芯片来实现，具体电路如图 14-4 所示。

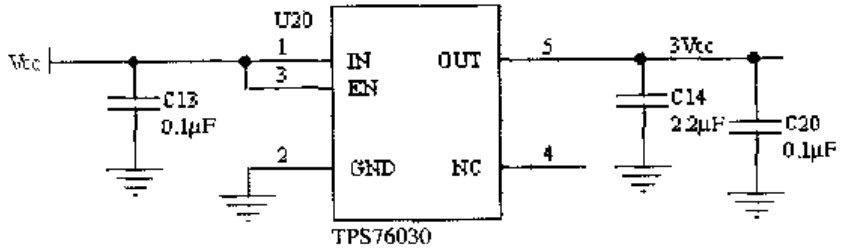


图 14-4　电源电路

在图 14-4 中，为了减少干扰，在各个电源管脚处需要外加滤波电容进行滤波处理。

经过上面的介绍，对系统的硬件接口有了比较清楚的认识，下面结合该硬件接口电路来介绍接口电路的软件设计。

14.2　软件设计

整个软件设计相对很简单，主要包括时钟初始化、端口初始化、SPI 串口初始化、D/A 转换等模块，下面就各个部分的程序分别进行详细介绍。

1．时钟初始化程序

这部分程序主要是选择单片机的 SMCLK 和 MCLK 的信号源，从而提供单片机的某些外设工作的时钟。下面为时钟的初始化程序。

```
void Init_CLK(void)
{
    unsigned int i;
    BCSCTL1 = 0X00; //将寄存器的内容清零
                    //XT2 震荡器开启
                    //LFTX1 工作在低频模式
                    //ACLK 的分频因子为 1

    do
    {
    // 清除 OSCFault 标志
    IFG1 &= ~OFIFG;
    for (i = 0x20; i > 0; i--);
    }
    while ((IFG1 & OFIFG) == OFIFG); // 如果 OSCFault =1

    BCSCTL2 = 0X00;          //将寄存器的内容清零
    BCSCTL2 += SELM1;        //MCLK 的时钟源为 TX2CLK，分频因子为 1
    BCSCTL2 += SELS;         //SMCLK 的时钟源为 TX2CLK，分频因子为 1
}
```

2．端口初始化程序

该部分程序主要设置 P3.0、P3.1 和 P3.3 管脚的属性。具体的程序如下：

```
void Init_Port(void)
{
    //将 P3 口所有的管脚在初始化的时候设置为输入方式
    P3DIR = 0;

    //将 P3 口所有的管脚设置为一般 I/O 口
    P3SEL = 0;

    //P3.1、P3.2、P3.3 被分配为 SPI 口
```

```
    P3SEL = BIT3 + BIT2 + BIT1;
    //P3.0 作为输出管脚
    P3DIR |= BIT0;
    //P3.3 作为输出管脚
    P3DIR |= BIT3;
    //P3.1 作为输出管脚
    P3DIR |= BIT1;
    //P3.0 输出高电平，使 DAC8830 不被选通
    P3OUT |= BIT0;
    return;
}
```

在上面的程序中，需要将 P3.1、P3.2 和 P3.3 三个管脚设置成 SPI 口的管脚，并且将 P3.0、P3.1 和 P3.3 设置成输出管脚。

3. SPI 串口初始化程序

```
void Init_SPI (void)
{
    //SPI0 模块允许
    ME1 |= USPIE0;
    //将寄存器的内容清零
    U0CTL = 0X00;
    //数据为 8 比特，选择 SPI 模式，单片机为主机模式
    U0CTL |= CHAR + SYNC + MM;

    //将寄存器的内容清零
    U0TCTL = 0X00;
    // 时钟源为 SMCLK，选择 3 线模式
    U0TCTL = CKPH + SSEL1 + SSEL0 + STC;

    //传输时钟为 SMCLK / 4
    UBR0_0 = 0X02;
    UBR1_0 = 0X00;
    //调整寄存器，没有调整
    UMCTL_0 = 0X00;

    //发送中断允许
    IE1 |= UTXIE0;
}
```

在上面的程序中，首先通过“ME1 |= USPIE0;”来使能 SPI 模块，然后分别设置串口通信的帧格式和通信速率。最后，使用语句“IE1 |= UTXIE0;”打开发送中断。

4. 转换程序

该部分程序主要根据 DAC8830 的工作时序来完成转换。MSP430 的转换程序主要包

括下面两个部分。第一，MSP430 单片机首先在 P3.0 管脚上送出低电平，选通 DAC8830。第二，MSP430 单片机通过串口输出数据，在数据输出完毕后，在单片机的 P3.0 管脚输出高电平，使 DAC8830 处于非选通状态。根据上面描述的过程，下面给出具体的实现代码。

```
void convert (int nValue)
{

    unsigned char hi_byte;
    unsigned char lo_byte;

    //取出高字节
    hi_byte = (char)(nValue >> 8);
    //取出低字节
    lo_byte = (char)(nValue & 0x0ff);

    //选通 DAC8830
    CS_Enable();
    //等待传输完成
    while ((IFG1 & UTXIFG0) == 0) ;
    //发送数据
    U0TXBUF = lo_byte;

    //等待传输完成
    while ((IFG1 & UTXIFG0) == 0) ;
    //发送数据
    U0TXBUF = hi_byte;

    //不选通 DAC8830
    CS_Disable();

    return ;
}
```

上面的程序代码使用查询中断标志的方式来处理传输数据完成。在上面的程序中用到了一些函数，用到的函数代码如下。

```
void CS_Enable(void)
{
    //P3.0 输出低电平
    P3OUT &=~(BIT0);
    return ;
}
void CS_Disable(void)
{
    //P3.0 输出高电平
    P3OUT|=BIT0;
    return ;
```

```
}
void complete(void)
{
    //等待传输完成的中断标志
    do
    {
        IFG1 &=~ UTXIFG0;
    }
    while (UTXIFG0 & IFG1);
}
```

通过上面的转换函数可以看出，实际上就是单片机通过 SPI 串口向 DAC8830 发送数据。数据发送很容易通过串口中断来实现，下面给出串口发送的中断程序。

```
interrupt [UART0TX_VECTOR] void UART0_TX_ISR(void)
{
    if(nTX0_Len != 0)
    {
        // 表示缓冲区里的数据没有发送完
        nTX0_Flag = 0;

        TXBUF0 = UART0_TX_BUF[nSend_TX0];
        nSend_TX0 += 1;

        if(nSend_TX0 >= nTX0_Len)
        {
            nSend_TX0 = 0;
            nTX0_Len = 0;
            nTX0_Flag = 1;
        }
    }
}
```

在上面的中断处理函数中，主要通过“nTX0_Len”、“nSend_TX0”、“nTX0_Flag”和“UART0_TX_BUF[]”全局变量实现与其他函数进行数据交互。

根据上面的查询方式和中断方式，下面给出简单的测试代码。

（1）查询方式的测试代码

```
void main_normal()
{
    int i;

    //初始化时钟
    Init_CLK();
    //端口初始化
    Init_Port();
    //SPI 初始化
```

```
    Init_SPI();

    //初始化表
    InitTables(4);

    for(i = 0; i < 256; i++)
    {
        convert(sinetable[i]);
    }

    return;
}
```

（2）中断方式的测试代码

```
void main_int()
{
    int i;
    //初始化时钟
    Init_CLK();
    //端口初始化
    Init_Port();
    //SPI 初始化
    Init_SPI();

    //初始化表
    InitTables(4);

    //打开中断
    _EINT();

    nTX0_Len = 0;
    nTX0_Flag = 0;
    nSend_TX0 = 0;

    //选通 DAC8830
    CS_Enable();

    for(i = 0;i < 6;i++)
    {
        UART0_TX_BUF[2 * i] = (char)(sinetable[i] & 0x00ff);
        UART0_TX_BUF[2 * i + 1] = (char)((sinetable[i] >> 8) & 0x00ff);
    }
    nTX0_Len = 12;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;
    for(;;)
    {
```

```
            if(nTX0_Flag == 1) break;
        }

        //不选通 DAC8830
        CS_Disable();
        return;
    }
```

在上面的程序中，使用“_EINT();”打开全局中断。如果需要发送数据，则将数据放入到“UART0_TX_BUF[]”缓冲区里，通过“IFG1 |= UTXIFG0;”进入中断，通过“nTX0_Len = 12;”设置需要发送数据的字节数，数据发送完毕后，中断程序设置“nTX0_Flag”的值为“1”。主程序通过这个标志来判断数据是否发送完毕。

14.3 实例总结

本章介绍了 MSP430 单片机与 DAC8830 的接口设计，并给出了相应的程序。DAC8830 通过 SPI 口与单片机进行连接，使系统的复杂度非常小。在程序设计时，需要注意的是必须满足 DAC8830 的工作时序。通过本章的介绍，读者可以进一步扩展硬件和软件功能，设计出满足自己系统需要的 D/A 转换系统。

第 15 章

ADS1241 的接口设计与实现

在高精度称重、血液分析等应用领域里，对数据采集要求有非常高的精度。虽然在 MSP430 系列单片机中，有的单片机本身集成了 12 位或者 14 位的 A/D 转换通道，但是该 A/D 转换通道的采集精度远达不到高精度采集的要求。本章介绍 MSP430 单片机与高精度的 A/D 转换芯片 ADS1241 的接口设计，并给出相应的程序。

15.1 硬件接口电路设计

本系统的硬件接口电路相对比较简单，主要就是 MSP430 单片机与 A/D 转换芯片 ADS1241 的连接。为了便于理解接口，在介绍接口之前，先简要介绍一下 A/D 转换芯片 ADS1241。

15.1.1 ADS1241 芯片

ADS1241 是一款高性能、宽动态范围、高精度的 24 位 A/D 转换芯片。该芯片具有以下特性：

- 24 位的 A/D 转换。
- 可以通过程序来设置增益和输出速率。
- 外部提供差分方式的参考电压源，外部参考电压为 0.1V~5V。
- 具有片内自校正功能。

- 采用 SPI 串口与其他芯片接口。
- 低功耗。低功耗只有 600μW。
- 具有较宽的工作电压，电压范围为 2.7V~5.5V。
- 最多可以实现 8 路模拟采集。

为了增加对 ADS1241 芯片的认识，下面给出该芯片的框图，如图 15-1 所示。

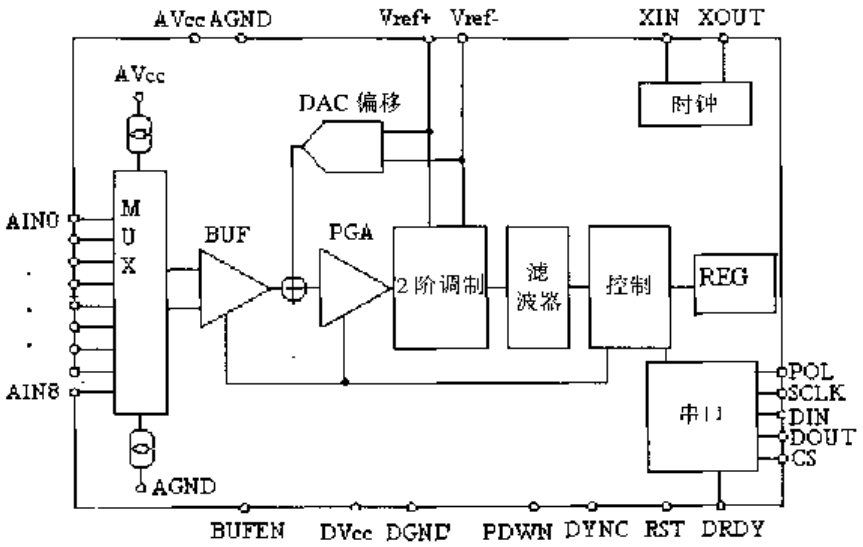


图 15-1 ADS1241 芯片框图

由图 15-1 可以看出，ADS1241 芯片内部提供复用器和缓冲区来实现多路采集。该芯片通过 SPI 串口与单片机进行连接。此外，芯片内部还进行滤波处理。为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 15-2 所示。

由图 15-2 可以看出，该芯片共有 28 个管脚，为了了解各个管脚的功能，下面对具体的管脚进行介绍。

- DVcc：数字电源管脚。
- DGND：数字接地管脚。
- AVcc：模拟电源管脚。
- AGND：模拟接地管脚。
- XIN：时钟输入管脚。
- XOUT：时钟输出管脚。XOUT 和 XIN 可以外接晶体，形成振荡电路。
- DYNC：同步控制管脚，低电平有效。
- PWND：低功耗控制管脚。低电平有效。

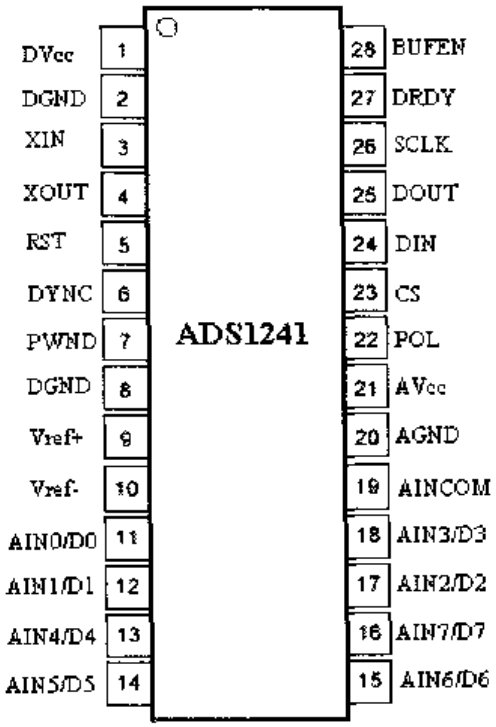


图 15-2　ADS1241 管脚图

- POL：串行时钟极性控制管脚。
- DRDY：数据准备好管脚。
- BUFEN：缓冲区使能管脚。
- Vref+、Vref-：外部差分参考的正、负输入管脚。
- CS：片选信号。低电平有效。
- RST：复位管脚。低电平复位方式。
- AINn/Dn：模拟输入管脚或者数字 I/O 管脚。
- AINCOM：模拟输入公共管脚。如果不使用，连接到 AGND 管脚。
- SCLK：SPI 串口的串行时钟输入管脚。
- DOUT：SPI 串口的数据输出管脚。
- DIN：SPI 串口的数据输入管脚。

通过对 ADS1241 芯片的介绍，对该款 A/D 转换芯片有了基本的认识，下面介绍硬件电路的具体设计。

15.1.2　接口设计

ADS1241 需要外部时钟才能工作，因此需要在 XIN 和 XOUT 管脚外接晶体，提供芯

片工作时所需要的时钟。本系统采用的是频率为 2.4576MHz 的晶体，电容为 20pF，电容的选择是和晶体的频率有关系的，具体可以查看 ADS1241 的数据手册。ADS1241 最多可以实现 8 通道采集，本系统只是简单地实现单通道采集，在实际的应用中，A/D 采集前端需要外加系统需要的一些滤波处理。ADS1241 通过 SPI 串口与单片机进行连接，这里使用的是 4 线方式，即 SCLK、DIN、DOUT 和 CS 管脚与单片机进行连接。另外，ADS1241 的 DRDY 管脚与单片机的一般 I/O 管脚进行连接，这样可以通过该管脚来判断是否准备好，由于该管脚输出低电平有效，因此需要将该管脚拉高。如图 15-3 所示为接口电路图。

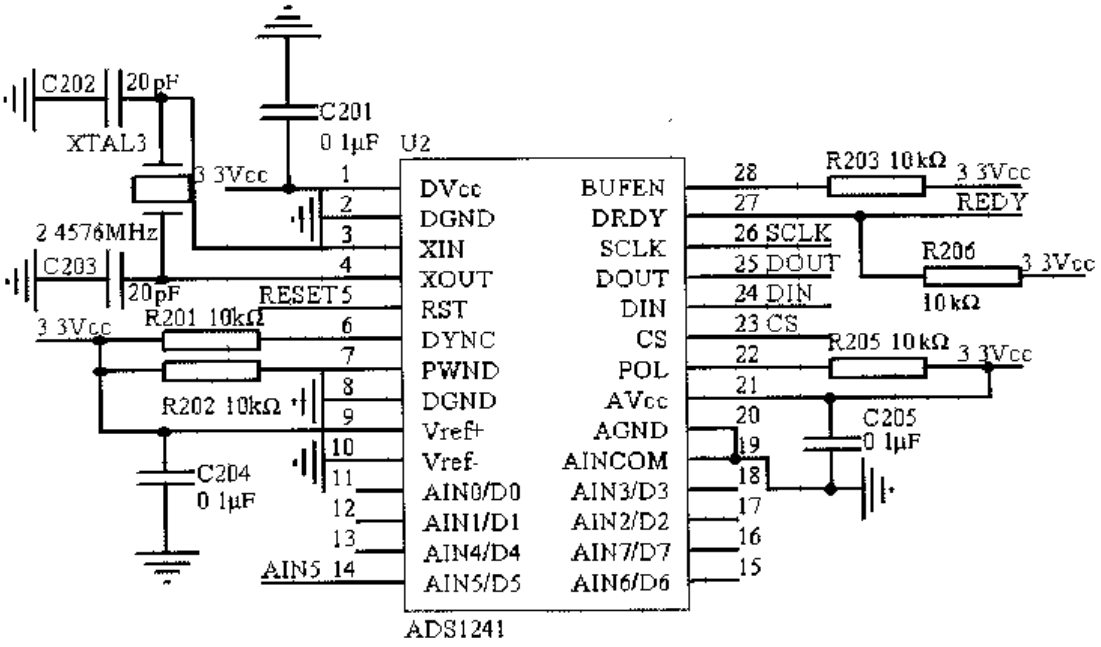


图 15-3　接口电路

在图 15-3 中，数字电源和模拟电源都采用 3.3V 电压供电，为了减小电源处的干扰，因此需要加 0.1μF 的电容进行滤波处理。本系统中也将数字地和模拟地接在一起，但在某些具体的应用中可能需要将数字地和模拟地分开。ADS1241 的外部参考源可以是差分方式，也可以是非差分方式，本系统中采用非差分方式，因此只需要将 Vref+管脚接外部参考电源（本系统接 3.3V 的参考源），Vref-管脚接地就可以了。关于外部参考电压的详细选择，可以参考 ADS1241 的数据手册。在本系统中，将 PWND 管脚接高电平，使该芯片一直处于工作状态；在某些低功耗场合下，可以将该管脚与单片机的一般 I/O 口进行连接，通过单片机来控制 ADS1241 的低功耗状态。图 15-3 只是一个实验电路，在实际应用中，需要根据系统的需要增加相应的处理，比如 A/D 通道的前置滤波处理等。

15.1.3　单片机电路

单片机电路非常简单，除了必要的振荡电路外，主要就是 SPI 串口电路。单片机可以采用外设中的串口与 ADS1241 的串口进行连接，也可以采用一般 I/O 管脚与 ADS1241 的串口进行连接。此外，单片机的一般 I/O 口的管脚与 ADS1241 的 DRDY 管脚进行连接，以便单片机对 ADS1241 数据的准备状态进行判断。如图 15-4 所示为单片机电路图。

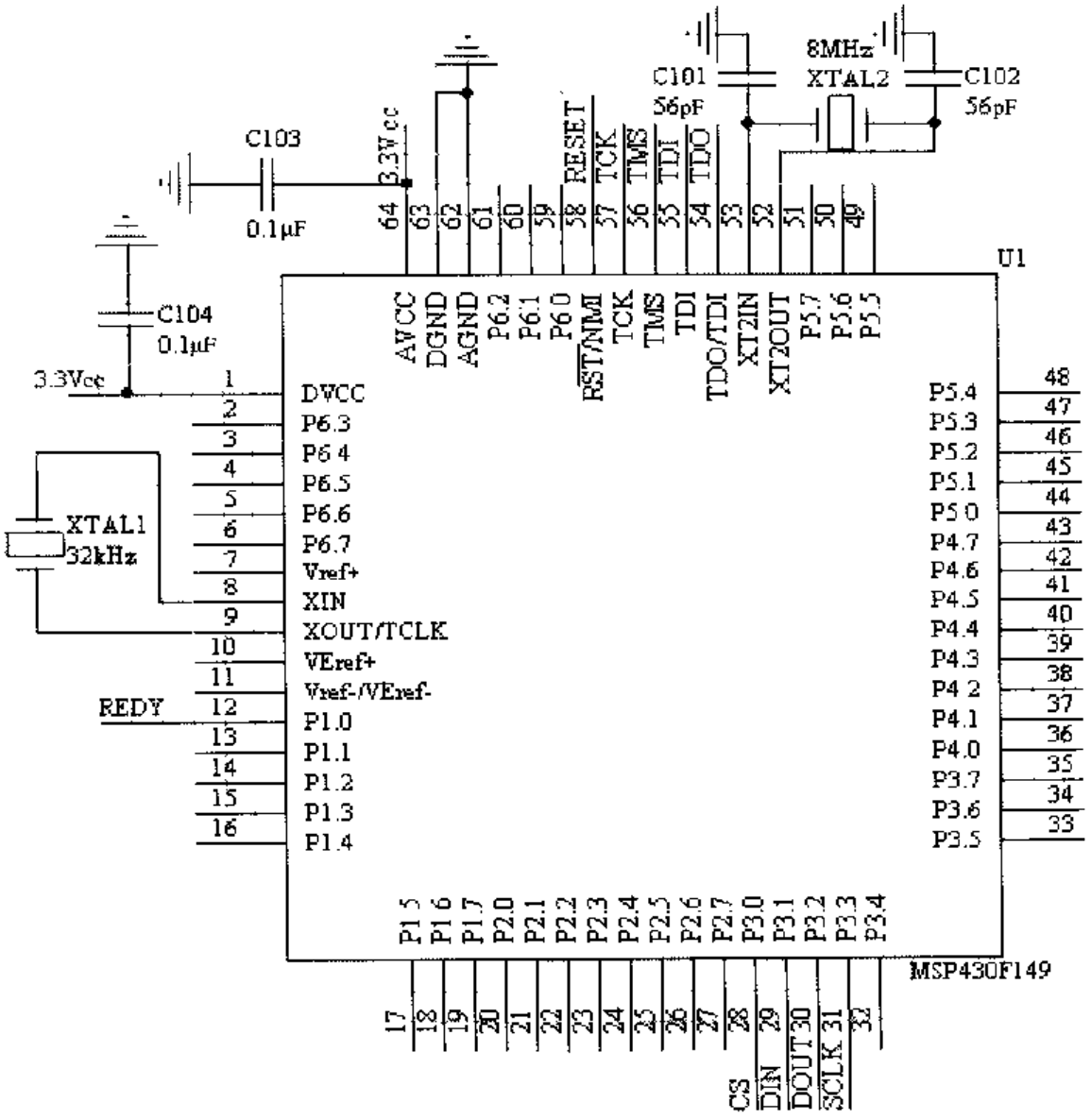


图 15-4　单片机电路

15.2 软件设计

本系统的软件设计主要是单片机从 ADS1241 读取数据的操作。ADS1241 的操作主要是通过不同的命令操作不同的寄存器，从而实现不同的操作。下面首先介绍 ADS1241 的寄存器和控制命令。

15.2.1 寄存器及控制命令

ADS1241 的所有操作都是通过设置相应的寄存器来实现的。这些寄存器包含了所有需要设置的信息，比如数据格式、复用设置、数据速率等信息。表 15-1 中所示为 ADS1241 的 16 个寄存器。

表 15-1 ADS1241 的寄存器

地 址	寄 存 器	功 能
0x00	SETUP	主要设置增益
0x01	MUX	复用控制寄存器
0x02	ACR	模拟控制寄存器
0x03	ODAC	偏移量寄存器
0x04	DIO	数字 I/O 寄存器
0x05	DIR	数字 I/O 的方向寄存器
0x06	IOCON	I/O 设置寄存器
0x07	OCR0	偏移校正系数寄存器 0
0x08	OCR1	偏移校正系数寄存器 1
0x09	OCR2	偏移校正系数寄存器 2
0x0A	FSR0	全尺度寄存器 0
0x0B	FSR1	全尺度寄存器 1
0x0C	FSR2	全尺度寄存器 2
0x0D	DOR2	数据输出寄存器 2
0x0E	DOR1	数据输出寄存器 1
0x0F	DOR0	数据输出寄存器 0

单片机实现对 ADS1241 的不同操作是通过向 ADS1241 发送不同命令来实现的。表 15-2 中所示为 ADS1241 的操作命令。

表 15-2　ADS1241 的操作命令

命　令	操 作 码	第 2 个命令字节	描　述
RDATA	00000001（0x01）		读数
RDATAC	00000011（0x03）		连续读数
STOPC	00001111（0xF）		停止连续读数
RREG	0001rrrr（0x1x）	xxxx_nnnn	读寄存器
WREG	0101rrrr（0x5x）	xxxx_nnnn	写寄存器
SELFCAL	11110000（0xF0）		偏移与增益自校正
SELFOCAL	11110001（0xF1）		偏移自校正
SELFGCAL	11110010（0xF2）		增益自校正
SYSOCAL	11110011（0xF3）		系统偏移校正
SYSGCAL	11110100（0xF4）		系统增益校正
WAKEUP	11111011（0xFB）		从休眠中唤醒
DSYNC	11111100（0xFC）		同步
SLEEP	11111101（0xFD）		休眠
RESET	11111110（0xFE）		复位

在表 15-2 的操作码中，r 代表寄存器的地址，第 2 个命令字节中的 n 代表操作寄存器的个数，这里的个数为实际操作寄存器的个数减 1。比如，从 0x01 开始读取 2 个寄存器的值，它的命令就应该是：00010001 00000001。

通过对寄存器和操作命令的介绍，对 ADS1241 的操作应该有了基本的认识，下面介绍具体操作的实现。

15.2.2　ADS1241 的操作实现

由表 15-2 可以知道，对 ADS1241 的所有操作都是通过发送命令来实现的，因此 ADS1241 的操作主要包括两个部分的内容，即 SPI 口的实现和命令处理。下面对 SPI 口的实现和命令处理两个部分分别进行介绍。

1．SPI 口实现

MSP430 系列单片机里很多型号的单片机都带有硬件的 SPI 串口，采用自带的 SPI 串口，程序设计非常简单。但 MSP430 系列单片机里有的型号单片机不带 SPI 串口，基于这方面的考虑，本章采用一般 I/O 口模拟来实现 SPI 口的收发程序。该部分程序主要包括初始化、数据发送和数据接收，下面为具体的程序代码。

```
int ADS1241Init(void)
{
```

```
    // 定义初始状态
    P3OUT &= ~(ADS1241_SCLK  | ADS1241_DIN);
    P3OUT |=  (ADS1241_CS);
    // 方向
    P3DIR = (ADS1241_SCLK | ADS1241_CS | ADS1241_DOUT);
    //设置 DRDY 管脚
    P1SEL &= ~(ADS1241_DRDY);
    P1DIR &= ~(ADS1241_DRDY);

    return ADS1241_NO_ERROR;
}
```

上面的程序主要是设置 SPI 口管脚的输入输出方向。

```
void ADS1241SendByte(int Byte)
{
    int i,j;
    for (i=0; i<8; i++)
    {
        // 输出数据
        if (Byte & 0x80)
            P3OUT |=  ADS1241_DIN;
        else
            P3OUT &= ~ADS1241_DIN;

        // 时钟管脚输出高电平
        P3OUT |=  ADS1241_SCLK;
        //延迟一点时间
        for(j = 20;j > 0;j--) ;
        // 时钟管脚输出低电平
        P3OUT &= ~ADS1241_SCLK;
        //延迟一点时间
        for(j = 20;j > 0;j--) ;
        Byte <<= 1;
    }
}
```

以上程序是按照 ADS1241 发送数据的时序来进行模拟的，具体的可以参照 ADS1241 的数据手册。

```
unsigned char ADS1241ReceiveByte(void)
{
    unsigned char Result = 0;
    int i,j;
    for (i=0; i<8; i++)
    {
        Result <<= 1;
        // 时钟管脚输出高电平
```

```
            P3OUT |=  ADS1241_SCLK;
            //延迟一点时间
            for(j = 20;j > 0;j--) ;

            //读数据
            if (P3IN & ADS1241_DOUT) Result |=  1;
            // 时钟管脚输出低电平
            P3OUT &= ~ADS1241_SCLK;
            //延迟一点时间
            for(j = 20;j > 0;j--) ;
        }
        return Result;
}
```

以上程序是按照 ADS1241 接收数据的时序来进行模拟的，具体的可以参照 ADS1241 的数据手册。

2. 命令处理

命令处理主要处理数据的读取、寄存器的读写等操作，下面进行具体的程序介绍。

（1）读寄存器操作

```
int ADS1241ReadRegister(int StartAddress, int NumRegs, unsigned * pData)
{
    int i;

    // 选通 ADS1241
    ADS1241AssertCS(1);

    // 发送命令
    ADS1241SendByte(ADS1241_CMD_RREG | (StartAddress & 0x0f));

    // 发送参数
    ADS1241SendByte(NumRegs-1);

    //延迟一点时间
    for(i = 50;i > 0;i--);

    // 读取数据
    for (i=0; i< NumRegs; i++)
    {
        *pData++ = ADS1241ReceiveByte();
    }

    // 不选通 ADS1241
    ADS1241AssertCS(0);
    return ADS1241_NO_ERROR;
```

```
}
```

读寄存器操作的命令通过表 15-2 可以知道，需要发送 2 个命令字节。

在上面的程序代码中使用到“ADS1241AssertCS(0);”，其代码如下。

```
void ADS1241AssertCS( int fAssert)
{
    //分别设置高电平或者低电平
    if (fAssert)
        P3OUT &= ~ADS1241_CS;
    else
        P3OUT |=  ADS1241_CS;
}
```

（2）写寄存器操作

```
int ADS1241WriteRegister(int StartAddress, int NumRegs, unsigned * pData)
{
    int i;

    // 选通 ADS1241
    ADS1241AssertCS(1);

    // 发送命令
    ADS1241SendByte(ADS1241_CMD_WREG | (StartAddress & 0x0f));

    // 发送参数
    ADS1241SendByte(NumRegs-1);

    //延迟一点时间
    for(i = 50;i > 0;i--) ;

    // 发送数据
    for (i=0; i< NumRegs; i++)
    {
        ADS1241SendByte(*pData++);
    }

    // 不选通 ADS1241
    ADS1241AssertCS(0);
    return ADS1241_NO_ERROR;
}
```

写寄存器操作的命令通过表 15-2 可以知道，需要发送 2 个命令字节。

（3）读取数据操作

```
long ADS1241ReadData(int fWaitForDataReady)
{
```

```
    int j;
    long Data;
    // 如果需要，同步到/DRDY 管脚
    if (fWaitForDataReady)
        ADS1241WaitForDataReady(0);

    // 选通 ADS1241
    ADS1241AssertCS(1);

    // 发送命令字节
    ADS1241SendByte(ADS1241_CMD_RDATA);

    //延迟一点时间
    for(j = 50;j > 0;j--) ;

    // 得到转换结果
    Data = ADS1241ReceiveByte();
    Data = (Data << 8) | ADS1241ReceiveByte();
    Data = (Data << 8) | ADS1241ReceiveByte();

    // 符号扩展
    if (Data & 0x800000)
        Data |= 0xff000000;

    // 不选通 ADS1241
    ADS1241AssertCS(0);
    return Data;
}
```

在读取数据操作中需要使用 DRDY 管脚来进行同步。使用到的同步函数的代码如下。

```
int ADS1241WaitForDataReady(int Timeout)
{
    if (Timeout > 0)
    {
        // 判断 ADS1241 的管脚输出是否是高电平
        while (!(P1IN & ADS1241_DRDY) && (Timeout-- >= 0)) ;
        // 判断 ADS1241 的管脚输出是否是低电平
        while ( (P1IN & ADS1241_DRDY) && (Timeout-- >= 0)) ;
        if (Timeout < 0)
            return ADS1241_TIMEOUT_WARNING;
    }
    else
    {
        // 判断 ADS1241 的管脚输出是否是高电平
        while (!(P1IN & ADS1241_DRDY)) ;
        // 判断 ADS1241 的管脚输出是否是低电平
        while ( (P1IN & ADS1241_DRDY)) ;
```

```
    }
    return ADS1241_NO_ERROR;
}
```

在以上介绍的函数基础上，基本可以实现对 ADS1241 的操作。

15.2.3 测试程序

下面给出一个简单的测试程序，以便了解 ADS1241 工作的顺序。

```
void main(void)
{
    char ACRVal;
    long nRes;

    // 初始化时钟
    Init_CLK();
    // ADS1241 的初始化
    ADS1241Init();
    // 复位 ADS1241
    ADS1241SendResetCommand();

    // 设置增益和复用方式
    ADS1241SetGain(ADS1241_GAIN_1);
    ADS1241SetChannel(0x05 | ADS1241_MUXN_AINCOM);

    // 速率 = 15Hz (2.4576MHz, SPEED = 0)
    ACRVal = 0;
    ADS1241WriteRegister(ADS1241_ACR_REGISTER, 1, &ACRVal);

    // 内部自校准
    ADS1241AssertCS(1);
    ADS1241SendByte(ADS1241_CMD_SELFCAL);
    ADS1241AssertCS(0);

    for (i=0; i<4; i++)
        ADS1241WaitForDataReady(0);

    // 读取数据
    nRes = ADS1241ReadData(1);

    return ;
}
```

上面的程序只是简单地给出利用 ADS1241 进行数据采集的例子，读者可以根据自己的需要，按照自己系统的处理流程来实现代码。

15.3　实例总结

本章介绍了 MSP430 单片机与 ADS1241 的接口设计，并给出了相应的程序。ADS1241 通过 SPI 口与单片机进行连接，使系统的复杂度非常小。通过本章的介绍，使读者能够了解高精度模拟数据采集系统的实现，读者也可以扩展硬件和软件功能，设计出满足自己系统需要的高精度模拟数据采集系统。

第 16 章

基于 MSP430 实现的数字温度测量系统

温度采集广泛应用于人们的生产和生活中。本章介绍一种采用数字温度传感器 TMP100 实现的温度测量系统，该系统可以方便地实现温度实时监控。

16.1 硬件设计

本系统的硬件接口电路相对比较简单，主要就是 MSP430 单片机与数字温度传感器 TMP100 的连接。在介绍具体的电路之前，先简要介绍一下数字温度传感器 TMP100。

16.1.1 TMP100 芯片

TMP100 是 TI 公司的一款采用 I^2C 接口的数字温度传感器芯片，它具有很小的封装，只有 6 个管脚。该芯片具有以下特性：

- 采用 I^2C 接口方式。
- 用户可以设置分辨率为 9 位到 12 位。
- 功耗很低。工作时电流为 45μA，处于低功耗时电流仅为 0.1μA。
- 单电源供电，并且有较宽的供电范围，供电电压范围为 2.7V~5.5V。
- 小封装。只有 6 个管脚，采用 SOT-6 封装。

为了增加对 TMP100 芯片的认识，下面给出该芯片的框图，如图 16-1 所示。

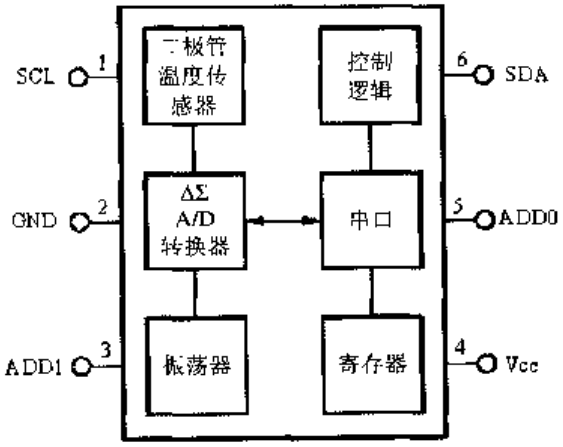


图 16-1　TMP100 芯片框图

图 16-1 中也给出了 TMP100 的管脚图。下面对具体的管脚进行介绍。

- Vcc：电源管脚。
- GND：接地管脚。
- ADD1：地址线 1。用来进行器件寻址。
- ADD0：地址线 0。用来进行器件寻址。
- SDA：I^2C 的数据线。
- SCL：I^2C 的时钟线。

经过对 TMP100 芯片的介绍，对该款数字温度传感器芯片有了基本的认识，下面介绍硬件电路的具体设计。

16.1.2　接口电路设计

硬件的接口电路相对简单，主要就是 I^2C 接口的设计。在 MSP430 系列单片机中，MSP430F12X、MSP430F13X 和 MSP430F14X 等系列单片机都没有 I^2C 模块，考虑这种情况，将 TMP100 的 I^2C 口与 MSP430 单片机的一般 I/O 口连接就行了。如图 16-2 所示为接口电路图。

在图 16-2 中，将 ADD1 和 ADD0 管脚接地，这意味该器件的地址为“00”。将 TMP100 的 I^2C 总线的两个管脚分别与 MSP430 单片机的两个一般 I/O 管脚连接。TMP100 的 SCL 管脚与单片机的 P1.2 管脚连接，TMP100 的 SDA 管脚与单片机的 P1.3 管脚连接，但需要注意的是 I^2C 两个管脚都需要拉高。由于单片机电路相对比较简单，这里就不再给出具体的电路图了。

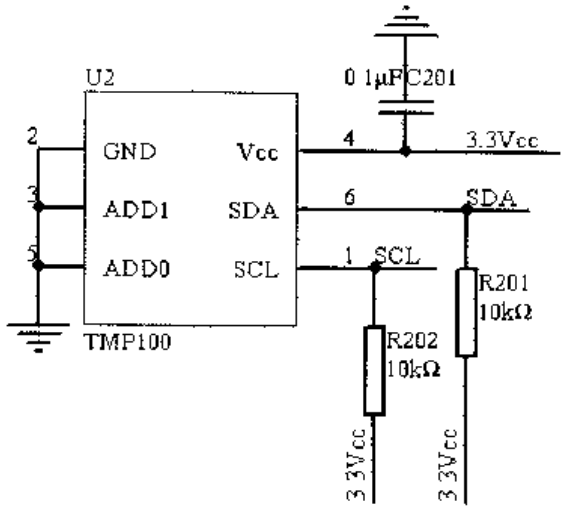


图 16-2　TMP100 的接口电路设计

16.2　软件设计

本系统的软件设计主要包括 I^2C 总线的实现和 TMP100 操作的实现。由于在第 11 章里已经介绍了 I^2C 总线的实现，这里就不再进行介绍。下面主要介绍 TMP100 操作的实现。

16.2.1　TMP100 操作

对 TMP100 的操作是通过操作内部的寄存器来实现的。如图 16-3 所示为 TMP100 的片内寄存器结构图。

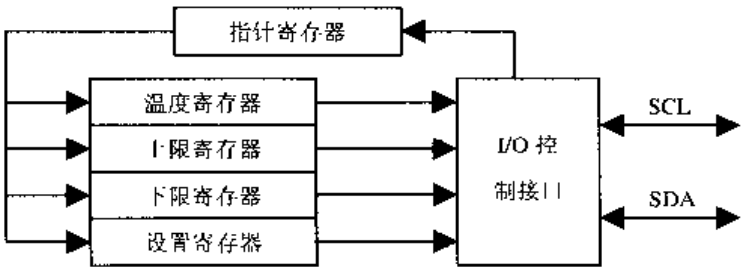


图 16-3　TMP100 片内寄存器结构

由图 16-3 可以看出，单片机要访问 TMP100 片内的某个寄存器，需要通过指针寄存器来实现。指针寄存器的最低两位（P1、P0）用来寻址片内其他寄存器。表 16-1 中所示为 P1、P0 的寻址编码。

表 16-1　P1、P0 的寻址编码

P1	P0	寄 存 器
0	0	温度寄存器（只读）
0	1	设置寄存器（读/写）
1	0	下限寄存器（读/写）
1	1	上限寄存器（读/写）

由表 16-1 可以看出，单片机要访问 TMP100 片内的某个寄存器，首先要设置指针寄存器的最低两位的值。为了进一步了解 TMP100 的操作，下面对寄存器进行简单的介绍。

温度寄存器是用来存放温度数据的，占 2 个字节，但是有效的数据只有 14 位，即低字节的 8 位和高字节的低 4 位。上、下限寄存器主要用于存放温度的上限和下限，当温度高于上限寄存器所设置的温度时，则会产生报警状态位；下限寄存器的作用类似。设置寄存器主要用来设置或者读取 TMP100 的工作状态，该寄存器有 8 位，如图 16-4 所示为设置寄存器的示意图。

OS/ALARM	R1	R0	F1	F0	POL	TM	SD

图 16-4　设置寄存器示意图

下面对设置寄存器的各个位进行简单介绍。

- SD：低功耗模式设置位。如果设置该位为 1，则 TMP100 进入低功耗模式。
- TM：比较模式还是中断模式设置位。如果该位为 1，则为中断模式；如果该位为 0，则为比较模式。
- POL：极性位。与 OS/ALARM 结合使用。
- F1、F0：错误队列位。在温度超出了上、下限寄存器所设置的上限和下限时，该两位用来确定连续产生多少个错误后就设置报警信息。
- R1、R0：分辨率位。该两位用来设置采集的分辨率。该两位的值为 00 时，分辨率为 9 位；该两位的值为 01 时，分辨率为 10 位；该两位的值为 10 时，分辨率为 11 位；该两位的值为 11 时，分辨率为 12 位。
- OS/ALARM：当处于低功耗状态时，设置该位用来执行单次转换，转换完后继续进入低功耗状态。当 TMP100 不处于低功耗状态时，该位表示报警状态信息。

以上对 TMP100 的寄存器进行了简单介绍，读者可以参看 TMP100 数据手册了解更加详细的信息。下面介绍 TMP100 操作程序的实现。

16.2.2 TMP100 操作的实现

TMP100 的操作主要包括读寄存器操作和写寄存器操作。TMP100 的读寄存器操作和写寄存器操作函数都是建立在 I^2C 程序基础上的，I^2C 程序在第 11 章做了详细的介绍，本章不做重点介绍。考虑到 TMP100 收发数据都是从最高位开始的，和第 11 章的有一点区别，本章给出收发函数。下面给出 I^2C 的数据收发、读操作和写操作 3 个部分的具体程序代码。

1. I^2C 的数据收发函数

TMP100 的数据收发都是从最高位开始，因此 I^2C 的数据收发也必须是从最高位开始。下面为 I^2C 的收发函数的代码。

```
void I2C_TxByte(int nValue)
{
    int i;
    int j;

    for(i = 0;i < 8;i++)
    {
        if(nValue & 0x80)
            I2C_Set_sda_high();
        else
            I2C_Set_sda_low();
        for(j = 30;j > 0;j--);
        I2C_Set_sck_high();
        nValue <<= 1;
        for(j = 30;j > 0;j--);
        I2C_Set_sck_low();
    }

    return;
}
int  I2C_RxByteHiToLow(void)
{
    int nTemp = 0;
    int i;
    int j;

    I2C_Set_sda_high();

    P1DIR &= ~(SDA);                    //将 SDA 管脚设置为输入方向
    _NOP();
    _NOP();
```

```
    _NOP();
    _NOP();
    for(i = 0;i < 8;i++)
    {
        I2C_Set_sck_high();

        if(P1IN & SDA)
        {
        nTemp |= (0x01 << (8-i));
        }
        for(j = 30;j > 0;j--);
        I2C_Set_sck_low();
    }

    return nTemp;
}
```

以上给出了 I²C 的收发函数，在此基础上，就可以实现 TMP100 的读写操作。

2. 读操作

TMP100 的读操作主要是从不同的寄存器里读取数据，下面为具体的程序。

```
int TMP_ReadData(char nPointer,char pBuf[],int nLen)
{
    int nTemp = 0;
    int i,j;

    // 启动数据总线
    I2C_START();
    // 发送写命令
    I2C_TxByte(0x90);
    // 等待 ACK
    nTemp = I2C_GetACK();

    //发送指针寄存器
    I2C_TxByte(nPointer);
    // 等待 ACK
    nTemp = I2C_GetACK();

    // 发送写命令
    I2C_TxByte(0x91);
    // 等待 ACK
    nTemp = I2C_GetACK();

    //读数据
    for(i = 0;i < nLen - 1;i++)
    {
```

```
        //读数据
        pBuf[i] = I2C_RxByteHiToLow();
        //设置 ACK
        I2C_SetACK();
        //延迟一点时间
        for(j = 10;j > 0;j--) ;
    }

    // 停止总线
    I2C_STOP();
    return nTemp;
}
```

上面的函数是严格按照 TMP100 的读操作时序实现的。在上面的函数入口参数中，“nPointer”为指针寄存器的值，“pBuf[]”表示从 TMP100 里读出返回给用户的数据，“nLen”表示读出数据的个数。

3. 写操作

TMP100 的写操作主要是往不同的寄存器里写入数据，下面为具体的程序。

```
int TMP_WriteData(char nPointer,char pBuf[],int nLen)
{
    int nTemp = 0;
    int i,j;

    // 启动数据总线
    I2C_START();
    // 发送写命令
    I2C_TxByte(0x90);
    // 等待 ACK
    nTemp = I2C_GetACK();

    //发送指针寄存器
    I2C_TxByte(nPointer);
    // 等待 ACK
    nTemp = I2C_GetACK();

    //写数据
    for(i = 0;i < nLen - 1;i++)
    {
        I2C_TxByte(pBuf[i]);
        //延迟一点时间
        for(j = 10;j > 0;j--) ;
    }

    // 停止总线
```

```
    I2C_STOP();
    return nTemp;
}
```

上面的函数是严格按照 TMP100 的写操作时序实现的。在上面的函数入口参数中，“nPointer”为指针寄存器的值，“pBuf[]”表示写入到 TMP100 的数据，“nLen”表示要写入数据的个数。

通过上面的函数就可以实现 TMP100 的相关操作，下面为一个简单的测试程序。

```
void main(void)
{
    char pBuf[10];
    int i;
    int nRes;

    Init_CLK();

    //设置 TMP100 12 位精度
    pBuf[0] = 0xFF;
    TMP_WriteData(0x01,pBuf,1);

    for(i = 1000;i > 0;i--) ;
    //读温度
    TMP_ReadData(0x00,pBuf,2);

    return;
}
```

以上代码只是简单地设置 TMP100，并从 TMP100 里读取温度数据。

16.3　实例总结

本章介绍了采用 TMP100 实现简单的温度采集系统，首先介绍电路的接口设计，然后介绍了相应的程序。TMP100 通过 I^2C 总线与单片机进行连接，使系统的复杂度非常小。在程序设计时，需要注意的是必须满足 TMP100 的工作时序。通过本章的介绍，使读者能够了解 TMP100 的具体操作。读者也可以扩展硬件和软件功能，设计出满足自己系统需要的温度采集系统。

第 17 章

基于 MSP430 定时器实现的 DAC

在单片机应用中，有时需要将数字信号转换成模拟信号。这种转换一般是采用 DAC 转换芯片来实现，或者利用单片机自带的 DAC 外设来实现。但是在某些成本敏感的应用领域或者单片机不带 DAC 外设时，可以利用单片机的定时器来实现 DAC。本章介绍采用 MSP430 单片机的定时器 A 来实现 DAC。

17.1 硬件设计

MSP430 单片机的定时器 A 有比较/捕获功能，利用这种功能可以输出 PWM（脉冲宽度调制）信号，利用 PWM 信号就可以实现 DAC。下面从实现原理、滤波器设计和电路设计 3 个方面来介绍基于 MSP430 单片机实现 DAC。

17.1.1 实现原理

定时器的每个比较/捕获模块都有 1 个输出单元，用于产生输出信号。每个输出单元有 8 种工作模式，选择相应的输出模式就可以得到相应的信号。本章介绍采用输出模式中的“复位/置位”模式来产生 PWM 信号。对于“复位/置位”模式，当 TAR 的值小于 CCR1 时，输出高电平；当 TAR 的值大于 CCR1 时，输出低电平。高低电平的时间由 CCR1 和 CCR0 寄存器的内容确定。如图 17-1 所示为“复位/置位”模式下的输出示意图。

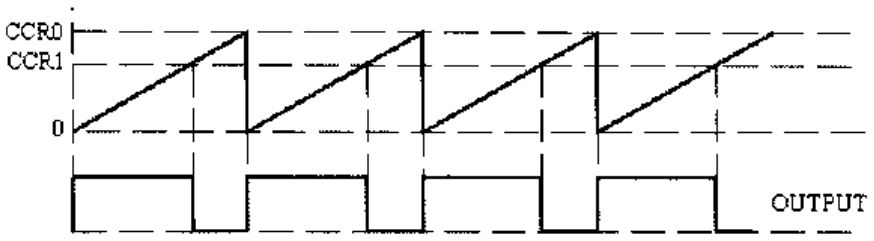


图 17-1　输出示意图

由图 17-1 可以看出，只要改变 CCR1 和 CCR0 的值就可以改变输出波形的脉冲宽度。输出的信号就是 PWM 信号。PWM 信号是一种周期（T）固定、占空比（τ）变化的数字信号，如图 17-2 所示。

如果 PWM 信号的占空比随时间变化，那么通过滤波之后的输出信号将是幅度变化的模拟信号。因此通过控制 PWM 信号的占空比，就可以产生不同的模拟信号。在本章介绍的例子中，采用 MSP430 单片机的定时器 A 的 CCR0 来控制周期，采用定时器 A 的 CCR1 来控制占空比，从而产生需要的 PWM 信号。

由于 PWM 是数字信号，但是由于它的占空比是变化的，根据信号分析的原理知道该信号经过滤波后就可以变成模拟信号。如图 17-3 所示为 MSP430 单片机实现 DAC 的原理框图。

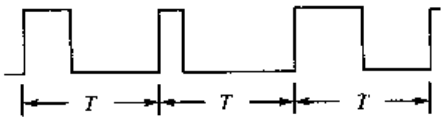


图 17-2　PWM 信号波形示意图

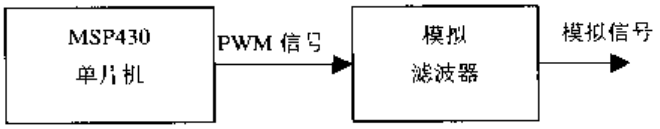


图 17-3　MSP430 单片机实现 DAC 的原理框图

由图 17-3 知道，PWM 信号只有经过模拟滤波处理后才能得到所需要的模拟信号，下面简单介绍滤波器的设计。

17.1.2　滤波器设计

一般说来，最基本的滤波器是 RC 滤波器。本章采用简单的 RC 滤波器来实现滤波处理，之所以采用这种结构，一是因为 RC 滤波器结构简单，二是采用 RC 滤波器可以实现低功耗应用。如果需要更加复杂的滤波处理，需要使用滤波器组来进行滤波处理。本章介绍一个简单的双极点

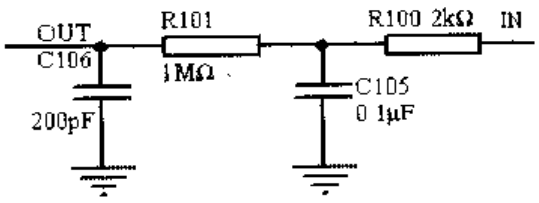


图 17-4　滤波器的示意图

级联 RC 滤波器，如图 17-4 所示为该滤波器的示意图。

由图 17-4 可以计算出该滤波器的截止频率。截止频率可以通过下式计算得到。

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi R_{101}C_{106}} = \frac{1}{2\pi R_{100}C_{105}} \tag{17-1}$$

通过式（17-1）知道，如果需要改变滤波器的截止频率，那么只需要改变 R_{101}、C_{106}、R_{100}、C_{105} 的值就可以了。在滤波器设计时，当 R_{101} 远大于 R_{100} 时，滤波器的响应较好。由于截止频率很接近信号带宽边沿时，将会导致非常大的衰减。因此为了减小滤波器的衰减，截止频率应该大于信号带宽边沿，但是要远小于 PWM 信号的频率。

17.1.3 电路设计

硬件电路的设计非常简单，采用定时器 A 输出 PWM 信号，PWM 信号通过滤波器进行滤波处理就能得到所需要的模拟信号。如图 17-5 所示为系统的电路图。

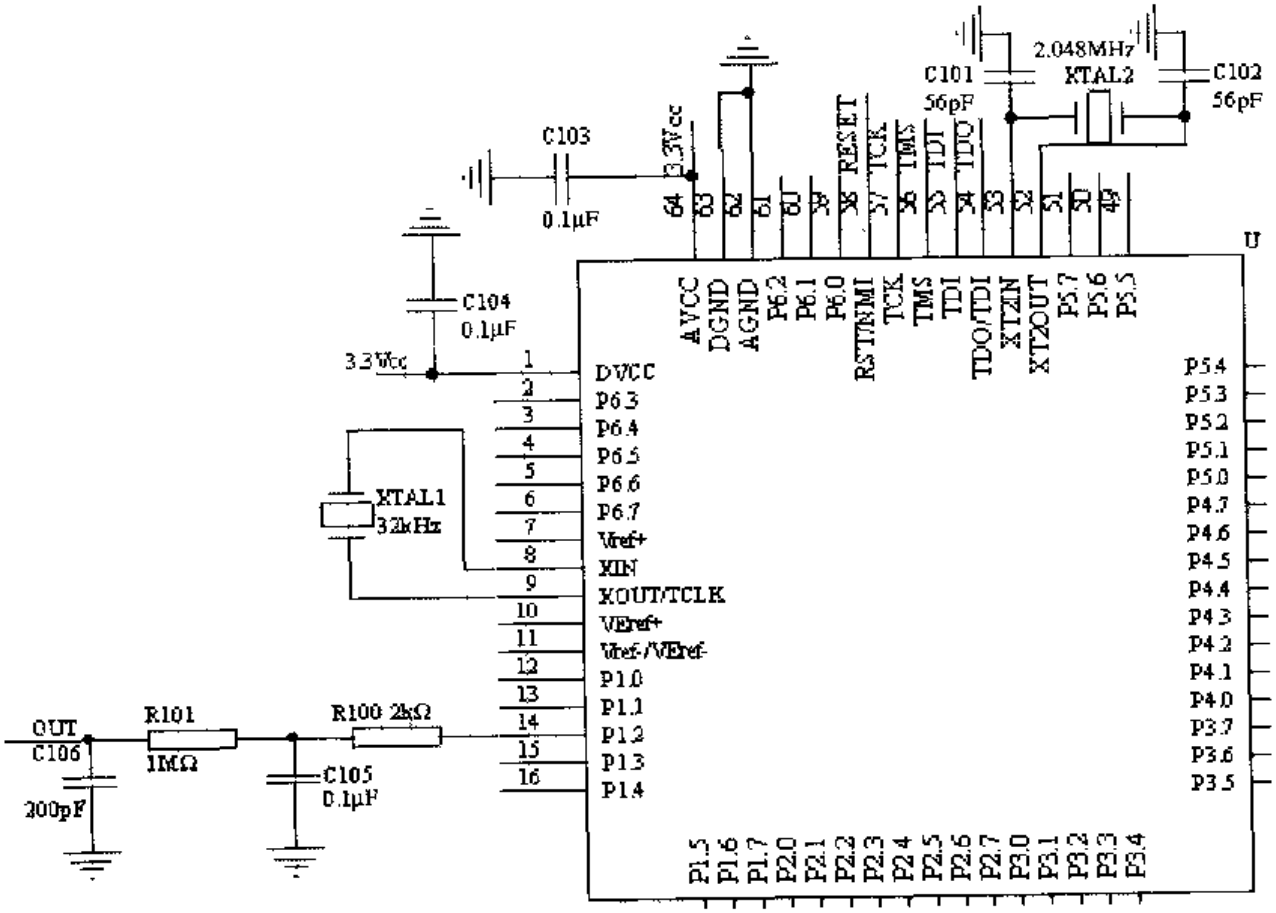


图 17-5　系统的电路图

通过图 17-5 可以看出，单片机的定时器 A 的 TA1（P1.2 管脚）输出 PWM 信号，PWM 信号经过滤波器处理后得到模拟信号。至于具体的 PWM 信号的产生则通过程序来完成，下面介绍具体的程序设计。

17.2　软件设计

本系统的软件设计主要就是利用定时器 A 实现 PWM 信号输出，从而实现 DAC。在介绍具体程序之前，先介绍一下 DAC 的分辨率和信号的频率这两个方面的概念。

17.2.1　DAC 分辨率

基于 MSP430 单片机的定时器 A，PWM 的 DAC 分辨率就等于计数器的长度，通常是 CCR0 寄存器的值。PWM 的 DAC 的最低有效位是 1 个计数值，分辨率就是总的计数值。分辨率可以用下式来表示。

$$R_{\text{counts}} = L_{\text{counts}} \tag{17-2}$$

其中，R_{counts} 是以计数值为单位的分辨率，L_{counts} 是计数器的总计数值。例如：8 位 DAC 的计数器的长度为 8 位，或者 256 个总计数值，那么分辨率也就是 8 位，或者 256。在一般情况下，基于定时器和滤波器的 PWM 实现的 DAC，它的分辨率等于产生模拟信号的 PWM 信号的分辨率。PWM 信号的分辨率由计数器的长度和 PWM 计数器能够实现的最小占空比来决定，具体关系可以用下面的数学表达式来表示。

$$R_{\text{counts}} = \frac{L}{C} \tag{17-3}$$

其中，L 是计数器的总计数值，C 是占空比的最小值。根据式（17-3）可以得到位分辨率的计算式如下。

$$R_{\text{BIT}} = \log_2(R_{\text{counts}}) \tag{17-4}$$

通过式（17-3）和式（17-4）就可以计算得到分辨率和位分辨率。例如：PWM 计数器的长度为 512 个计数值，最小的占空比为 2 个计数值，那么 PWM 的 DAC 的分辨率就为 256，位分辨率就为 8。

17.2.2　信号的频率

PWM 信号所需要的输出频率等于 DAC 的更新频率，因为 PWM 信号占空比的每 1 次变化都等效于 1 次 DAC 采样。PWM 定时器所需的频率取决于 PWM 信号频率和所需的分

辨率。具体关系可以用下面的表达式来表示。

$$F_{clock} = F_{PWM} \times 2^n \tag{17-5}$$

在式（17-5）中，F_{clock} 是 PWM 定时器频率，F_{PWM} 是 PWM 信号的频率，也就是 DAC 的更新频率，n 是位分辨率。本章介绍的是采用 8 位 PWM 的 DAC 来产生一个 250Hz 的正弦波。由 Nyquist 抽样定理可得，最低的采样频率应该为 500Hz。但是通常情况下，PWM 信号的频率要远高于 Nyquist 采样频率。这是因为 PWM 信号的频率越高，对滤波器的阶数要求就越低，滤波器就越容易实现。通常采样频率取 Nyquist 频率的 16 倍或者 32 倍。由上面的介绍知道，本系统的 PWM 信号的频率为 8kHz，PWM 定时器的频率为 2.048MHz。

17.2.3 程序设计

经过前面的介绍，对分辨率、PWM 信号的频率和 PWM 定时器频率等有了基本的了解，下面介绍具体的程序设计。本章的程序主要包括初始化程序、定时器 A 的中断程序和主程序 3 个部分，下面分别对这 3 个部分进行介绍。

1．初始化程序

初始化程序主要初始化时钟和定时器 A，下面为具体的初始化程序。

```
void Init_CLK(void)
{
    unsigned int i;
    //将寄存器的内容清零
    //XT2 振荡器开启
    //LFTX1 工作在低频模式
    //ACLK 的分频因子为 1
    BCSCTL1 = 0X00;

    do
    {
        // 清除 OSCFault 标志
        IFG1 &= ~OFIFG;
        for (i = 0x20; i > 0; i--);
    }
    while ((IFG1 & OFIFG) == OFIFG);

    BCSCTL2 = 0X00;
    //MCLK 的时钟源为 TX2CLK:2.048MHz，分频因子为 0
    BCSCTL2 += SELM1 + DIVM_0;
    //SMCLK 的时钟源为 TX2CLK:2.048MHz，分频因子为 1
    BCSCTL2 += SELS + DIVS_0;
    return;
```

```
}
```

上面程序主要是设置系统的工作时钟。

```
void Init_TimerA(void)
{
    nCount = 0;
    // P1.2 为输出管脚
    P1DIR |= BIT2;
    // 选择 P1.2 为 TA1 管脚
    P1SEL |= BIT2;
    // 选择 SMCLK，清除 TAR
    TACTL = TASSEL1 + TACLR;
    // CCR0 中断允许
    CCTL0 = CCIE;
    // PWM 周期为 256
    CCR0 = 256 - 1;
    // CCR1 输出模式为“复位/置位”模式
    CCTL1 = OUTMOD_7;
    CCR1 = nSinTable[nCount];
    // 增计数模式
    TACTL |= MC1;
    return;
}
```

上面的程序主要设置定时器 A 的工作方式，设置 PWM 信号的参数，并设置输出单元的输出模式。

2. 定时器 A 的中断程序

定时器 A 的中断程序主要是负责修改 CCR1 的值，通过修改 CCR1 的值就可以改变 PWM 信号的占空比，从而实现所需要的 PWM 信号的输出。具体程序如下：

```
interrupt [TIMERA0_VECTOR] void Timer_A0_ISR(void)
{
    nCount += 1;
    if(nCount >= 32)
    {
        nCount = 0;
    }
    //将新的抽样值装入 CCR1
    CCR1 = nSinTable[nCount];
}
```

上面的程序很简单，从“nSinTable[]”里面取出值赋给 CCR1。

3. 主程序

主程序非常简单，只是在简单的初始化后进入处理循环，下面为主程序的代码。

```
//定义正弦表，并用 32 个抽样值初始化正弦表，不要用“0”抽样
int nSinTable[] = {
255,254,246,234,
219,199,177,153,
128,103,79,57,
37,22,10,2,
1,2,10,22,
37,57,79,103,
128,153,177,199,
219,234,246,255
};
int nCount;
int main(void)
{
    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;

    // 关闭中断
    _DINT();

    // 初始化
    Init_CLK();
    Init_TimerA();
    // 打开中断
    _EINT();
    for(;;)
    {
        //CUP 进入低功耗模式
        _BIS_SR(LPM0_bits);
        _NOP();
    }
}
```

在上面程序中，“nCount”和“nSinTable[]”为全局变量。使用这两个全局变量可以实现上程序与中断程序之间进行数据交互。

17.3 实例总结

本章介绍了采用 MSP430 单片机的定时器 A 来实现 DAC，具体介绍了硬件电路的设计和相应的程序设计。本章系统的硬件和软件都很简单，关键是理解滤波器的设计、分辨

率、频率等概念。通过本章的介绍，使读者能够了解使用 MSP430 单片机的定时器来实现 DAC，也可以使读者设计出适合自己系统的滤波器，修改相应的频率参数，设计出满足自己系统需要的 DAC。虽然本章的系统是使用定时器 A 来实现，但也可以采用 MSP430 单片机的定时器 B（如果该款 MSP430 单片机有定时器 B 的话）来实现。

第 18 章

数据采集系统的设计与实现

在单片机应用中，数据采集系统的应用越来越多。本章介绍采用 MSP430 单片机实现的数据采集系统。该系统利用 MSP430 单片机片内的 A/D 转换通道来实现，下面具体介绍该系统的实现。

18.1 硬件电路设计

本系统的硬件接口电路相对比较简单，主要是单片机利用 A/D 通道采集数据，并将采集得到的数据传往 PC 机。单片机将数据传往 PC 机主要通过串口来实现（参看第 25 章的详细介绍）。下面对模拟接口电路和单片机电路分别进行详细介绍。

18.1.1 接口电路设计

在该系统中主要考虑模拟前端为传感器，从传感器送来的是标准信号，即 4mA~20mA，这样设计具有一定的通用性，只要前端接不同的传感器就可以采集不同的信号源。由于 A/D 转换基准为电压，也就是参考源为电压，所以 A/D 转换的是电压，这样需要将电流信号转换成电压信号。如图 18-1 所示为具体的模拟量采集电路图。

由图 18-1 可以看出，采集电路通过一个电阻将电流信号转换成电压信号，为了提高采集的进度，需要采用高精度的电阻，这里采用的是精度为 1%的电阻。电路中采用二极管作为 ESD 保护电路，考虑到干扰问题，采用电容进行滤波处理，以增加采集电路的抗干扰性。

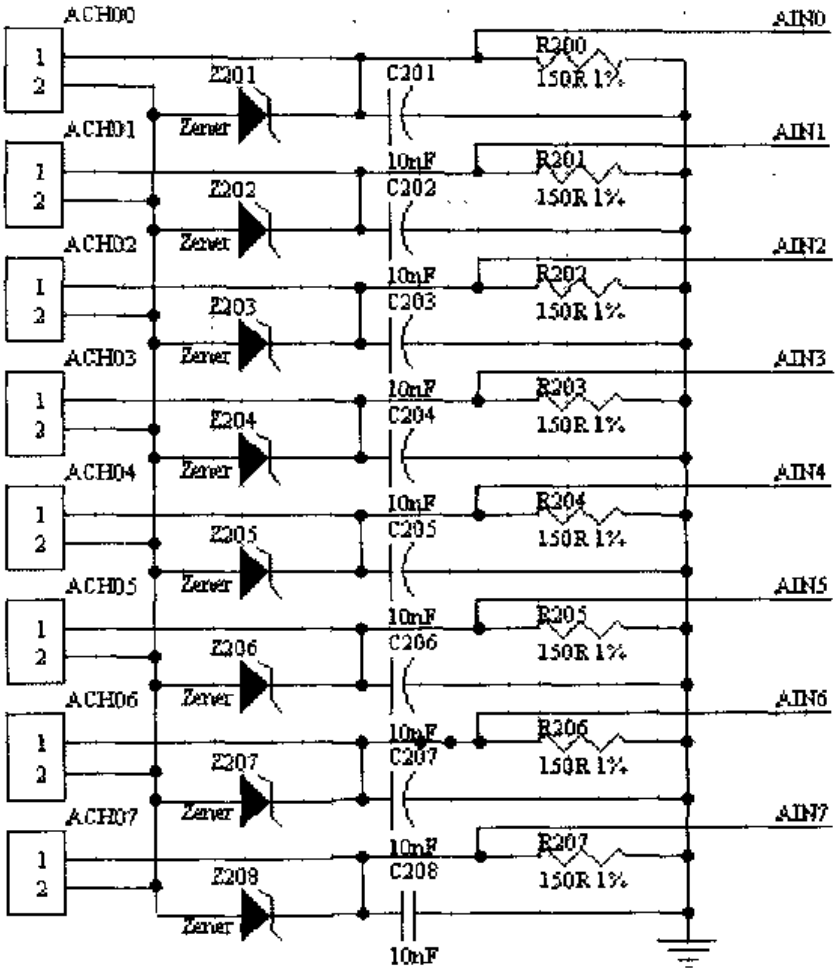


图 18-1　模拟量采集电路

18.1.2　单片机电路

单片机电路作为整个系统的核心控制部分，主要是完成与其他电路的接口。在该系统中，单片机主要负责对模拟量进行采集，并将采集得到的数据通过串口传给上位机，如图 18-2 所示为单片机电路图。

通过图 18-2 可以看出，单片机的接口电路非常简单，通过片内的 A/D 通道实现模拟量采集。采用片内的 A/D 转换部分不仅可以降低系统设计的复杂性，而且可以提高系统的可靠性，避免接口的复杂性，同时还可以减小 PCB 板的面积。模拟量采集的参考电压采用的是片内提供的参考电压。考虑到电源的输入纹波对单片机的影响，在电源的管脚增加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。另外，单片机还要模拟电源的输入端，因此在这里需要考虑干扰问题。在该系统中的干扰比较小，因此模拟地和数字地

共地，模拟电源输入端增加一个滤波电容以减小干扰。

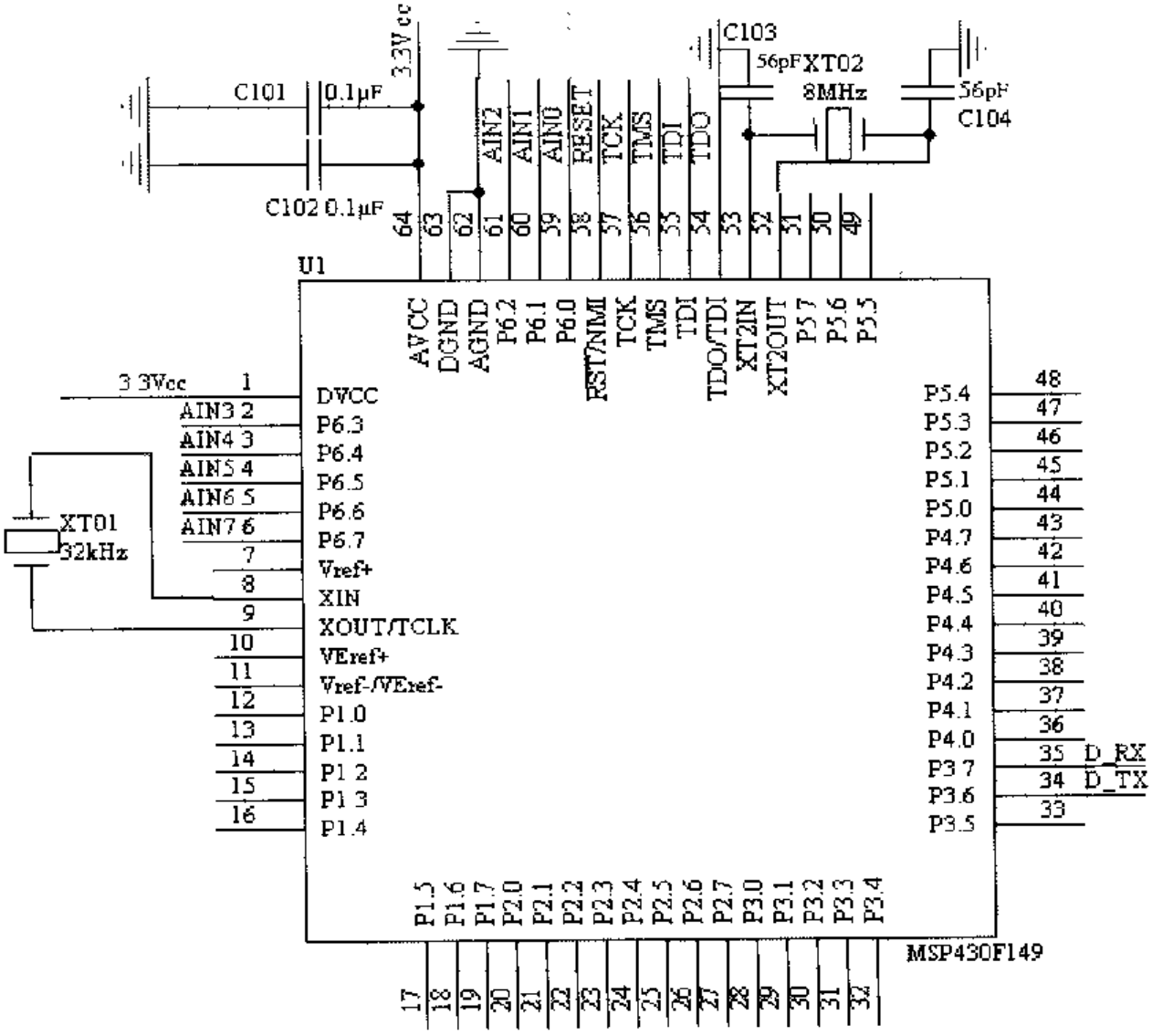


图 18-2　单片机电路

通过该节的介绍，对系统的硬件系统有了清楚的认识，下一节具体介绍系统的软件设计。

18.2　软件设计

整个软件设计包括初始化设置、中断服务程序和主处理程序，下面对各个部分程序分别进行详细介绍。

18.2.1　初始化设置

初始化程序主要包括时钟初始化、A/D 初始化、串口初始化和定时器初始化。其中时钟初始化和串口初始化参看第 25 章的详细介绍。下面为 A/D 初始化和定时器初始化的程序代码。

```
void Init_ADC(void)
{
    //设置 P6.0 为模拟输入通道
    P6SEL = 0X07;
    //设置 ENC 为 0，从而修改 ADC12 寄存器的值
    ADC12CTL0 &= ~(ENC);
    //转换的起始地址为：ADCMEM0
    ADC12CTL1 |= CSTARTADD_0;
    //设置参考电压分别为 AVSS 和 AVCC，输入通道为 A0
    ADC12MCTL0 = INCH_0;
    //设置参考电压分别为 AVSS 和 AVCC，输入通道为 A1
    ADC12MCTL1 = INCH_1;
    //设置参考电压分别为 AVSS 和 AVCC，输入通道为 A2
    ADC12MCTL2 = INCH_2;
    //设置参考电压分别为 AVSS 和 AVCC，输入通道为 A3
    ADC12MCTL3 = INCH_3;
    //设置参考电压分别为 AVSS 和 AVCC，输入通道为 A4
    ADC12MCTL4 = INCH_4;
    //设置参考电压分别为 AVSS 和 AVCC，输入通道为 A5
    ADC12MCTL5 = INCH_5;
    //设置参考电压分别为 AVSS 和 AVCC，输入通道为 A6
    ADC12MCTL6 = INCH_6
    //设置参考电压分别为 AVSS 和 AVCC，输入通道为 A7
    ADC12MCTL7 = INCH_7 + EOS;

    ADC12CTL0 |= ADC12ON;
    ADC12CTL0 |= MSC;

    //转换模式为：多通道、单次转换
    ADC12CTL1 |= CONSEQ_1;

    //SMCLK，时钟分频为 1，采样脉冲由采用的定时器产生
    ADC12CTL1 |= ADC12SSEL_1;
    ADC12CTL1 |= ADC12DIV_0;
    ADC12CTL1 |= (SHP);
    //使能 ADC 转换
    ADC12CTL0 |= ENC;
    return;
}
```

```
void Init_TimerA(void)
{
    // 选择 SMCLK，清除 TAR
    TACTL = TASSEL1 + TACLR;
    // 1/8 SMCLK
    TACTL += ID1;
    TACTL += ID0;
    // CCR0 中断允许
    CCTL0 = CCIE;
    // 频率为 250Hz
    CCR0 = 4000;
    // 增计数模式
    TACTL |= MC0;

    return;
}
```

18.2.2 中断服务程序

该部分主要完成 8 通道模拟数据的采集，并且通过定时器 A 来控制采集的频率。另外，也设置一个标志来通知主程序已经获得新的数据，通过全局变量来实现与主处理程序进行数据交互。这部分采用中断服务程序实现，在定时器 A 里先停止 A/D 转换，读取数据后启动 A/D 转换，然后再等待下一次中断的到来。下面为定时器 A 处理和 A/D 转换部分的程序代码。

```
interrupt [TIMERA0_VECTOR] void TimerA_ISR(void)
{
    int results[8];
    int i;
    // 关闭转换
    ADC12CTL0 &= ~ENC;
    // 读出转换结果
    ADC_BUF0[nADC_Count] = ADC12MEM0;
    // 读出转换结果
    ADC_BUF1[nADC_Count] = ADC12MEM1;
    // 读出转换结果
    ADC_BUF2[nADC_Count] = ADC12MEM2;
    // 读出转换结果
    ADC_BUF3[nADC_Count] = ADC12MEM3;
    // 读出转换结果
    ADC_BUF4[nADC_Count] = ADC12MEM4;
    // 读出转换结果
    ADC_BUF5[nADC_Count] = ADC12MEM5;
    // 读出转换结果
    ADC_BUF6[nADC_Count] = ADC12MEM6;
```

```
    // 读出转换结果
    ADC_BUF7[nADC_Count] = ADC12MEM7;

    nADC_Count += 1;
    if(nADC_Count == 10)
    {
        //设置标志
        nADC_Flag = 1;
        nADC_Count = 0;
        // 将数据倒向数据缓冲区
        for(i = 0;i < 10;i++) ADC_BUF_Temp0[i] = ADC_BUF0[i];
        for(i = 0;i < 10;i++) ADC_BUF_Temp1[i] = ADC_BUF1[i];
        for(i = 0;i < 10;i++) ADC_BUF_Temp2[i] = ADC_BUF2[i];
        for(i = 0;i < 10;i++) ADC_BUF_Temp3[i] = ADC_BUF3[i];
        for(i = 0;i < 10;i++) ADC_BUF_Temp4[i] = ADC_BUF4[i];
        for(i = 0;i < 10;i++) ADC_BUF_Temp5[i] = ADC_BUF5[i];
        for(i = 0;i < 10;i++) ADC_BUF_Temp6[i] = ADC_BUF6[i];
        for(i = 0;i < 10;i++) ADC_BUF_Temp7[i] = ADC_BUF7[i];
    }
    // 开启转换
    ADC12CTL0 |= ENC + ADC12SC;
}
```

以上程序使用了全局变量“nADC_Flag”通知主程序有新的采集数据获得；全局变量“nADC_Count”用来计数处理；“ADC_BUF0[]”等全局变量用来临时存放数据；“ADC_BUF_Temp0[]”等全局变量用来作为与主程序交换数据的缓冲区。

18.2.3　主处理程序

主处理程序主要是负责与各个中断服务程序进行数据交互。主程序与 A/D 采集中断服务程序进行数据交互，将得到的数据封装后，与串口中断服务程序进行数据交互，由串口中断程序将数据发送到 PC 机。下面为具体的程序代码。

```
void main(void)
{
    int i;
    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;
    // 关闭中断
    _DINT();
    // 初始化
    Init_CLK();
    Init_ADC();
    Init_TimerA();
    Init_Port();
```

```
Init_TimerB();
//初始化变量
nADC_Flag = 0;
nADC_Count = 0;
nSend_TX1 = 0;
nTX1_Len = 0;
nTX1_Flag = 0;
nRX1_Len_temp = 0;
nRev_UART1 = 0;
nRX1_Len = 0;
// 打开中断
_EINT();
// 开始循环
for(;;)
{
// 处理模拟量采集并发送
if(nADC_Flag == 1)
{
    nADC_Flag = 0;
    // 等待缓冲区里的数据发送完毕
    while(1)
    {
        if(nTX1_Flag == 1) break;
    }
    // 将数据由字转换成字节
    for(i = 0;i < 10;i++)
    {
        UART1_TX_BUF[2 * i] = (char)(ADC_BUF_Temp0[i] & 0x00ff);
        UART1_TX_BUF[2*i+1]=(ADC_BUF_Temp0[i]>>8)&0x00ff;
    }
    // 设置帧结束标志
    UART1_TX_BUF[20] = 0xaa;
    UART1_TX_BUF[21] = 0xaa;
    //发送数据的长度
    nTX1_Len = 22;
    // 设置中断标志
    IFG2 |= UTXIFG1;
    // 等待缓冲区里的数据发送完毕
    while(1)
    {
        if(nTX1_Flag == 1) break;
    }
    // 将数据由字转换成字节
    for(i = 0;i < 10;i++)
    {
        UART1_TX_BUF[2 * i] = (char)(ADC_BUF_Temp1[i] & 0x00ff);
        UART1_TX_BUF[2*i+1]=(ADC_BUF_Temp1[i]>>8)&0x00ff;
    }
```

```
// 设置帧结束标志
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
//发送数据的长度
nTX1_Len = 22;
// 设置中断标志
IFG2 |= UTXIFG1;
// 等待缓冲区里的数据发送完毕
while(1)
{
    if(nTX1_Flag == 1) break;
}
// 将数据由字转换成字节
for(i = 0;i < 10;i++)
{
    UART1_TX_BUF[2 * i] = (char)(ADC_BUF_Temp2[i] & 0x00ff);
    UART1_TX_BUF[2*i+1]=(ADC_BUF_Temp2[i]>>8)&0x00ff;
}
// 设置帧结束标志
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
//发送数据的长度
nTX1_Len = 22;
// 设置中断标志
IFG2 |= UTXIFG1;
// 等待缓冲区里的数据发送完毕
while(1)
{
    if(nTX1_Flag == 1) break;
}
// 将数据由字转换成字节
for(i = 0;i < 10;i++)
{
    UART1_TX_BUF[2 * i] = (char)(ADC_BUF_Temp3[i] & 0x00ff);
    UART1_TX_BUF[2*i+1]=(ADC_BUF_Temp3[i]>>8)&0x00ff;
}
// 设置帧结束标志
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
//发送数据的长度
nTX1_Len = 22;
// 设置中断标志
IFG2 |= UTXIFG1;
// 等待缓冲区里的数据发送完毕
while(1)
{
    if(nTX1_Flag == 1) break;
}
```

```
// 将数据由字转换成字节
for(i = 0;i < 10;i++)
{
    UART1_TX_BUF[2 * i] = (char)(ADC_BUF_Temp4[i] & 0x00ff);
    UART1_TX_BUF[2*i+1]=(ADC_BUF_Temp4[i]>>8)&0x00ff;
}
// 设置帧结束标志
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
//发送数据的长度
nTX1_Len = 22;
// 设置中断标志
IFG2 |= UTXIFG1;
// 等待缓冲区里的数据发送完毕
while(1)
{
    if(nTX1_Flag == 1) break;
}
// 将数据由字转换成字节
for(i = 0;i < 10;i++)
{
    UART1_TX_BUF[2 * i] = (char)(ADC_BUF_Temp5[i] & 0x00ff);
    UART1_TX_BUF[2*i+1]=(ADC_BUF_Temp5[i]>>8)&0x00ff;
}
// 设置帧结束标志
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
//发送数据的长度
nTX1_Len = 22;
// 设置中断标志
IFG2 |= UTXIFG1;
// 等待缓冲区里的数据发送完毕
while(1)
{
    if(nTX1_Flag == 1) break;
}
// 将数据由字转换成字节
for(i = 0;i < 10;i++)
{
    UART1_TX_BUF[2 * i] = (char)(ADC_BUF_Temp6[i] & 0x00ff);
    UART1_TX_BUF[2*i+1]=(ADC_BUF_Temp6[i]>>8)&0x00ff;
}
// 设置帧结束标志
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
//发送数据的长度
nTX1_Len = 22;
// 设置中断标志
```

```
            IFG2 |= UTXIFG1;
            // 等待缓冲区里的数据发送完毕
            while(1)
            {
                if(nTX1_Flag == 1) break;
            }
            // 将数据由字转换成字节
            for(i = 0;i < 10;i++)
            {
                UART1_TX_BUF[2 * i] = (ADC_BUF_Temp7[i] & 0x00ff);
                UART1_TX_BUF[2*i+1]=(ADC_BUF_Temp7[i]>>8)&0x00ff;
            }
            // 设置帧结束标志
            UART1_TX_BUF[20] = 0xaa;
            UART1_TX_BUF[21] = 0xaa;
            //发送数据的长度
            nTX1_Len = 22;
            // 设置中断标志
            IFG2 |  UTXIFG1;
        }

        // 处理接收数据
        if(nRev_UART1 == 1)
        {
            nRev_UART1 = 0;
            for(i = 0;i < nRX1_Len;i++)
                UART1_RX_TEMP[i] = UART1_RX_BUF[i];
            }

        }
    }
```

通过以上程序可以看出，上程序主要处理来自中断的数据，主程序通过查询标志来判断是否有新的采集数据到来或者上位机数据的到来。当主程序检测到“nADC_Flag”标志变量为“1”时，表示采集得到新的模拟量数据，从“ADC_BUF_Temp0[]”等全局缓冲区里取得采集得到的模拟量数据。主程序检测到“nRev_UART1”标志变量的值为 1 时，表示有来自上位机的数据，从“UART1_RX_BUF[]”全局缓冲区里取得来自上位机的数据。上程序也通过设置标志来通知串口发送中断程序向上位机发送数据，当主程序需要向上位机发送数据时，先将需要发送的数据装入到“UART1_TX_BUF[]”全局缓冲区里；再设置“nTX1_Len”变量，表示需要发送数据的长度；最后设置“IFG2 |= UTXIFG1”，进入串口中断发送程序发送数据。

18.3 实例总结

本章介绍了 MSP430 单片机实现的数据采集系统，先后介绍了电路的接口设计和相应的程序。在程序设计时采用中断服务程序的结构，这样便于进行多任务处理，充分利用单片机的硬件资源。在本章的设计基础上，也可以外加 A/D 芯片来增加采集的通道数，以满足用户系统的要求。

第 19 章

基于 MSP430 单片机实现的交流电压测量

在单片机的一些测量应用中，有时需要直接测量交流信号。本章介绍采用 MSP430 单片机实现的交流电压测量。首先介绍交流电压测量的电路设计，然后介绍交流电压测量的具体程序实现。

19.1 电路设计

为了保证硬件电路设计的通用性，采用单极性电压测量的方法，将输入的双极性电压转换成单极性电压进行测量。整个电路主要包括极性转换电路和输入处理电路。其中，极性转换电路主要由放大电路实现，本章采用的放大芯片为 MCP601。在介绍具体电路之前，先介绍一下 MCP601 芯片。

19.1.1 MCP601 芯片

MCP601 是 Microchip 公司的一款高性能的放大芯片。为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 19-1 所示。

由图 19-1 可以看出，该芯片共有 8 个管脚，为了了解各个管脚的功能，下面对具体的管脚进行介绍。

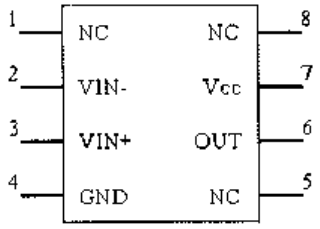


图 19-1　MCP601 芯片管脚图

- Vcc 管脚：电源管脚。
- GND 管脚：接地管脚。
- VIN-管脚：负输入管脚。
- VIN+管脚：正输入管脚。
- OUT 管脚：输出管脚。

经过对 MCP601 芯片管脚的介绍，对该款放大芯片的管脚有了基本的认识，下面介绍利用该款芯片设计的电压极性转换电路。

19.1.2　极性转换电路设计

在进行 A/D 转换时，一般采用芯片的工作电压作为 A/D 转换的参考电源。由于芯片的工作电压一般为正电压，因此需要对输入的交流信号进行极性转换，将双极性变成单极性。具体的电路如图 19-2 所示。

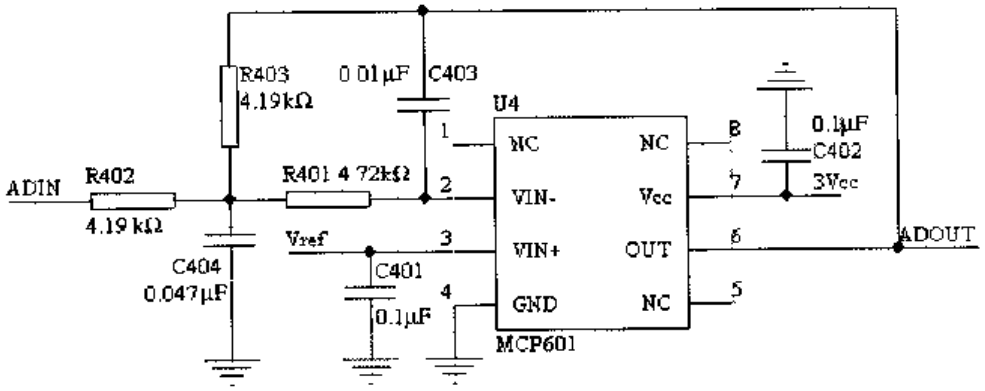


图 19-2　极性转换电路

在极性转换电路中，ADOUT 为输出信号。输出信号是在输入信号 ADIN 的基础上叠加了一个直流分量，调节上面的 Vref 的值就可以改变直流分量的值。如果调节 Vref 使直流分量的值为 1.5V，并且此时输入信号是幅值为 1.5V 的交流正弦信号，那么输出信号就为最大值为 3V，最小值为 0V 的单极性正弦信号。在极性转换电路基础上，很容易设计出输入电路，下面介绍输入处理电路的设计。

19.1.3　输入处理电路设计

在极性转换电路基础上，输入处理电路需要将 220V 的交流电压信号变为幅值为 1.5V

左右的交流信号，另外，还需要为 MCP601 提供适当的参考电压信号。具体的电路如图 19-3 所示。

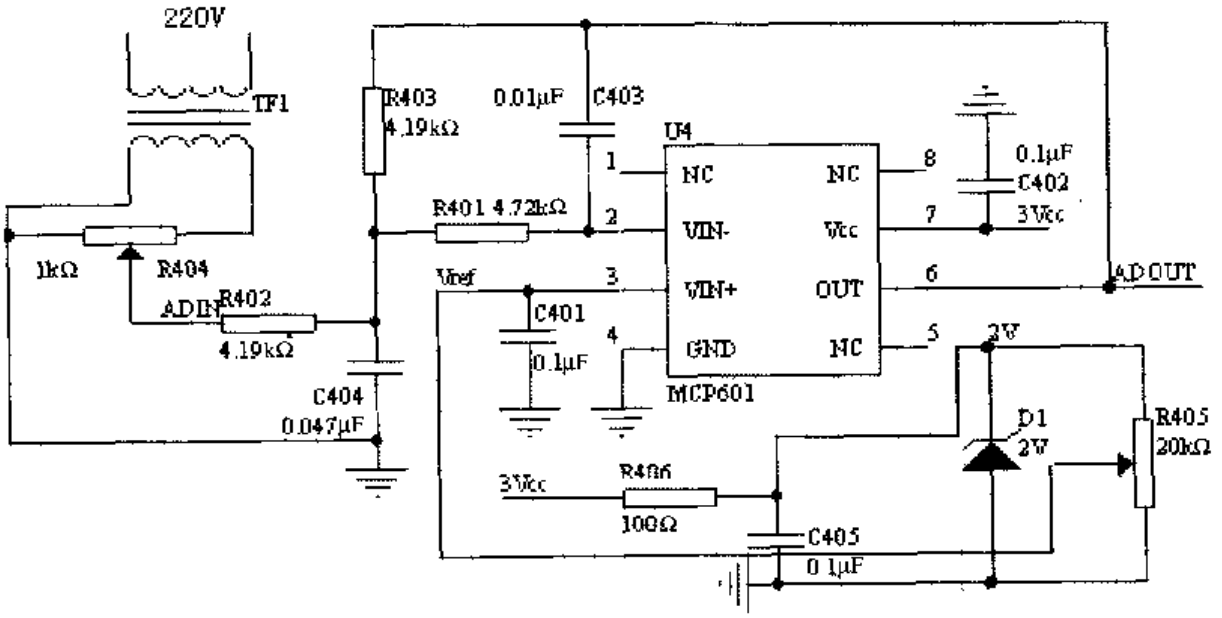


图 19-3　输入处理电路

在上面的电路中，首先通过变压器将 220V 的交流电压降压成 8V 的交流电压，再经过极性转换电路将双极性的交流电压转换为单极性的交流电压。电路中的 R405 电位器主要用于调节参考电压，R404 电位器用于调节交流输入电压的幅度。经过上面电路的处理，可以将输入的交流电压转换成 0~3V 的单极性交流电压，这样很容易使用 MSP430 单片机自带的 A/D 转换通道进行模拟量采集，从而实现交流电压的测量。

在上面的电路中，电压采用 3V 供电，电源芯片采用 TPS76030 芯片实现，具体的电路如图 19-4 所示。

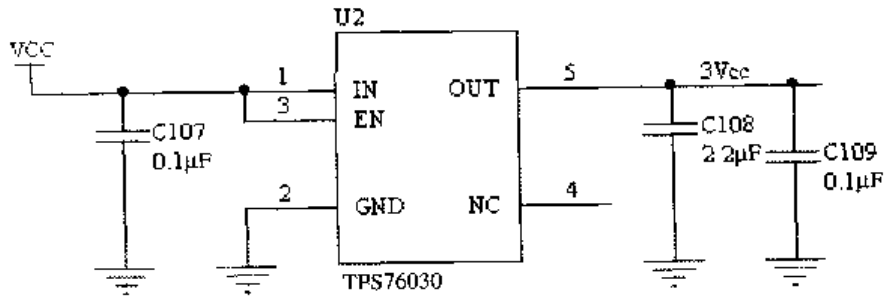


图 19-4　电源电路

以上介绍了具体的电路设计，下面介绍交流电压测量的程序设计。

19.2 程序设计

对于交流采集，需要在1个工频周期内采集40个点，即时间间隔为500μs，时间间隔采用定时器实现。整个程序主要包括初始化程序和采集程序，下面分别对各个部分的程序进行具体的介绍。

19.2.1 初始化程序

初始化程序主要是设置A/D采集通道和定时器A，下面为具体的程序代码。

```
void Init_ADC(void)
{
    //设置P6.0为模拟输入通道
    P6SEL = 0X01;
    //设置ENC为0，从而修改ADC12寄存器的值
    ADC12CTL0 &= ~(ENC);
    //设置参考电压分别为AVSS和AVCC，输入通道为A0
    ADC12MCTL0 = INCH_0 + EOS;
    //转换的起始地址为：ADCMEM0
    ADC12CTL1 = 0X00;
    ADC12CTL1 += CSTARTADD_0;
    //采样脉冲由采用的定时器产生
    ADC12CTL1 += SHP;
    //转换模式为：多通道、多次转换
    ADC12CTL1 += CONSEQ_1;
    //内部时钟源
    ADC12CTL1 += ADC12SSEL_0;
    //时钟分频为1
    ADC12CTL1 += ADC12DIV_0;

    ADC12CTL0 += 8 * 0x100;
    ADC12CTL0 += MSC;
    ADC12CTL0 += ADC12ON;

    ADC12IE = 0;
    //关闭各个通道的转换中断
    ADC12IE |= 0X00;
    //使能ADC转换
    ADC12CTL0 |= ENC;
    return;
}
```

上面的代码主要设置 A/D 采集的时钟源选择、通道选择及参考电压选择等。

```
void Init_TimerA(void)
{
    // 选择 SMCLK，清除 TAR
    TACTL = TASSEL1 + TACLR;
    // 1/8 SMCLK
    TACTL += ID1;
    TACTL += ID0;
    // CCR0 中断允许
    CCTL0 = CCIE;

    // 时间间隔为 500μs
    CCR0 = 500;
    // 增计数模式
    TACTL |= MC0;
    return;
}
```

上面的代码主要设置定时器的工作方式及时间间隔。

19.2.2　采集程序

采集程序主要是通过定时器来实现每间隔 500μs 采集 1 次。采集程序使用定时器中断服务程序实现。在定时器中断里读出数据，当采集完 40 个点的数据后，设置一个标志通知主程序已经采集完 40 个点的数据，主程序通过全局的数据缓冲区与定时器中断服务程序实现数据的交互。下面为具体的程序代码。

```
interrupt [TIMERA0_VECTOR] void TimerA_ISR(void)
{
    int results;
    // 关闭转换
    ADC12CTL0 &= ~ENC;
    // 读出转换结果
    results = ADC12MEM0;
    ADC_BUF[nADC_Count] = results;
    // 计数器加 1
    nADC_Count += 1;
    // 采集完 40 个点
    if(nADC_Count == 40)
    {
        // 设置标志
        nADC_Flag = 1;
        // 计数器清零
        nADC_Count = 0;
```

```
    }
    // 开启转换
    ADC12CTL0 |= ENC + ADC12SC;
}
```

以上代码主要实现数据的采集。在上面代码中，如果数据采集完 40 个点后，设置“nADC_Flag”为“1”；上程序如果检测到该标志，则从全局缓冲区“ADC_BUF[]”里读出数据。下面为测试用的上程序代码。

```
void main(void)
{
    int ADC_BUF_Temp[40];
    int i;
    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;
    // 关闭中断
    _DINT();

    // 初始化
    Init_CLK();
    Init_ADC();
    Init_TimerA();

    // 打开中断
    _EINT();
    // 循环处理
    for(;;)
    {
        if(nADC_Flag == 1)
        {
            nADC_Flag = 0;
            for(i = 0;i < 40;i++)
            {
                ADC_BUF_Temp[i] = ADC_BUF[i];
            }
        }
    }
}
```

19.3 实例总结

本章介绍了采用 MSP430 单片机实现的交流电压的采集测量。首先介绍了交流电压的极性转换电路，通过电压的极性转换电路，使双极性信号转换成单极性信号，便于单片机的测量。然后介绍了输入处理电路，在测量的输入处理电路中，需要对输入的交流电压进

行降压处理，以满足单片机的测试范围。在进行程序设计时，主要采用定时器中断服务程序进行交流电压数据的采集。虽然本章介绍的主处理程序相对比较简单，但是读者可以在此基础上增加数据分析的处理（比如计算有效值、进行谐波分析等），从而实现更为完整的测量系统。

第 20 章

基于 MSP430 单片机实现的车速里程表

随着汽车的发展，电子技术在汽车领域的应用越来越多。本章介绍采用 MSP430 单片机实现的车速里程表。首先介绍车速里程表的硬件设计，然后介绍它的软件设计。

20.1 硬件设计

一般说来，传统的汽车车速里程表主要使用机械式仪器仪表的指针来指示汽车行驶的瞬时车速，另外，通过机械计数器来记录汽车行驶的累计里程。但由于传统车速里程表是使用软轴驱动的，软轴在高速旋转时，由于受钢丝交变应力极限的限制而容易断裂，同时，软轴布置过长会出现形变过大或运动迟滞等现象，所以传统的车速里程表完全有可能被电子式的车速里程表所代替。电子式的车速里程表采用非接触的转速传感器来实现，能很好地克服传统车速里程表的不足。

电子式的车速里程表可以采用霍尔型非接触式转速传感器来进行速度的测量和里程的测量。车速里程表转轴每转 1 圈，霍尔传感器就会感应发出 8 个脉冲，这样就可以根据脉冲数来确定行驶的距离，通过行使的距离除以所花的时间就可以获得速度参数。一般说来，汽车的转轴转 n（比如 624）圈后就表示行使了 1 公里；汽车行使 1 公里，霍尔传感器就会发出 $8n$ 个脉冲；单片机就可以记录霍尔传感器发出的脉冲数，当记录到 $8n$ 个脉冲

后，就表示汽车行使了 1 公里，这样就实现了里程的测量。只要记录下时间，就可以计算得到速度的参数。系统将测量得到的参数显示出来。另外，考虑需要记录累计的里程数，因此需要使用存储器记录累计的里程数。如图 20-1 所示为系统实现的框图。

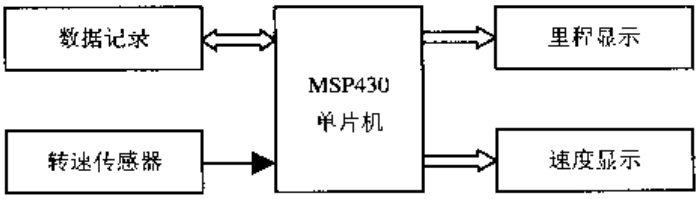


图 20-1　系统实现的框图

由图 20-1 可以看出，单片机采集来自转速传感器的脉冲得到里程参数，经过计算可以得到速度参数。系统对速度和里程参数采用数码管分别进行显示，对于里程参数使用 6 个数码管显示，对于速度只需要 3 个数码管显示就够了。另外，采用串行的 EEPROM 来记录累计的里程数，虽然 MSP430 单片机本身有 FLASH，但考虑数据的安全性，还是采用串行的 EEPROM 来记录数据。下面介绍具体的硬件电路。

20.1.1　显示电路

系统的显示电路有两个部分，一部分显示里程数据，一部分显示速度数据。显示电路采用数码管实现，在具体实现时，使用移位寄存器来进行串行输出显示，这种方式比直接连接数码管的方式可以减少管脚使用数。由于里程参数采用 6 位显示，因此需要 6 个数码管，而速度参数只需要 3 个数码管。单片机使用 3 个管脚控制一组的显示，这样单片机需要 6 个管脚实现两组数据的显示。具体的显示电路可以参看第 3 章，这里不再进行具体介绍。

20.1.2　存储器电路

存储器电路主要是串行 EEPROM 与 MSP430 单片机的连接。存储芯片选用 24LC02B，24LC02B 主要是通过 I^2C 实现与 MSP430 单片机的连接。在设计时，直接将 24LC02B 芯片的 WP 管脚接地，使 24LC02B 不处于写保护状态。由于 I^2C 是总线工作方式，该总线上可以挂有很多器件，所以总线上的每个器件都应该有相应的地址，这样才能实现寻址操作。24LC02B 的 A0、A1 和 A2 都接地，表示该器件的地址为 000。24LC02B 的 SCL 和 SDA 管脚分别与单片机的一般 I/O 口进行连接，连接方式是 I^2C 总线方式。在设计时，需要将 SCL 和 SDA 分别通过一个 10kΩ的电阻将其拉高，以满足 I^2C 工作的条件。具体的存储器电路可以参考第 11 章，这里不再进行具体介绍。

20.1.3 单片机电路

单片机电路作为系统处理的核心，主要完成与显示电路和存储器电路的接口，另外，还需要实现对霍尔型非接触式转速传感器的脉冲进行采集。对于 MSP430 单片机的某些型号（例如 MSP430F149）来说，P1 口和 P2 口具有中断功能，因此在采集霍尔型非接触式转速传感器发出的脉冲时，可以使用管脚的中断功能，这样也非常便于软件设计。本系统使用 P2.0 管脚来采集脉冲信号，采用上升沿触发方式。单片机的 P1.0、P1.1 和 P1.2 分别连接里程显示电路，实现里程数据的显示。单片机的 P1.3、P1.4 和 P1.5 分别连接速度显示电路，实现速度数据的显示。由于大部分 MSP430 单片机没有 I^2C 接口，因此使用一般 I/O 管脚来模拟 I^2C 接口。本系统使用单片机的 P1.6 和 P1.7 管脚与 24LC02B 芯片进行连接。如图 20-2 所示为单片机电路的示意图。

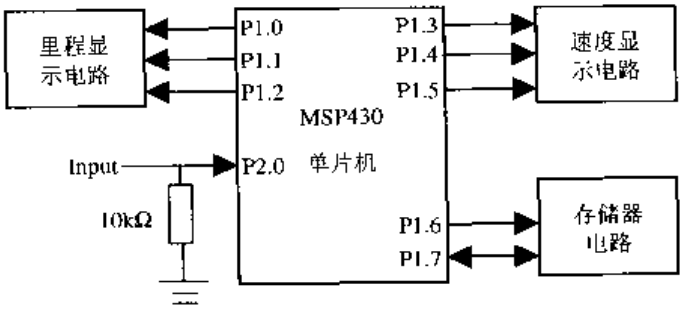


图 20-2 单片机电路示意图

以上介绍了系统的硬件设计，下面介绍系统的软件设计。

20.2 软件设计

整个软件包括初始化、显示、存储器操作和中断处理等部分。其中显示部分和存储器操作部分的具体实现在第 3 章和第 11 章做了介绍，这里就不再进行具体介绍了。下面对初始化、中断处理和主处理 3 个部分进行简单介绍。

20.2.1 初始化

初始化部分主要是设置端口管脚和定时器 B。下面为具体的程序代码。

```
void Init_Port(void)
{
```

```
    //将 P1 口所有的管脚在初始化的时候设置为输入方式
    P1DIR = 0;
    //将 P1 口所有的管脚设置为一般 I/O 口
    P1SEL = 0;

    // 将 P1.0、P1.1、P1.2 设置为输出方向
    P1DIR |= BIT0;
    P1DIR |= BIT1;
    P1DIR |= BIT2;
    // 将 P1.3、P1.4、P1.5 设置为输出方向
    P1DIR |= BIT3;
    P1DIR |= BIT4;
    P1DIR |= BIT5;

    //将 P2 口所有的管脚在初始化的时候设置为输入方式
    P2DIR = 0;
    //将 P2 口所有的管脚设置为一般 I/O 口
    P2SEL = 0;
    //将中断寄存器清零
    P2IE = 0;
    P2IES = 0;
    P2IFG = 0;
    //管脚 P2.0 使能中断
    P2IE |= BIT0;
    //对应的管脚由低到高电平跳变使相应的标志置位
    P2IES &= ~(BIT0);
    return;
}
```

在上面的程序中，设置 P1 口的输入输出方向，设置 P2.0 为中断方式，并且设置为低电平到高电平的触发方式，这样就可以利用中断功能来处理脉冲的采集计数了。

```
void Init_TimerB(void)
{
    // 选择 ACLK，清除 TAR
    TBCTL = TBSSEL0 + TBCLR;
    // TBCCR0 中断允许
    TBCCTL0 = CCIE;
    // 时间间隔为 1s
    TBCCR0 = 32768;
    // 增计数模式
    TBCTL |= MC0;
    return;
}
```

上面的程序设置了定时器 B 的工作方式，并设置了定时器 B 的时间间隔为 1s。

20.2.2 中断处理

中断处理包括 P2.0 管脚中断处理和定时器 B 中断处理。P2.0 管脚中断处理用于进行脉冲计数。下面为具体的程序代码。

```
interrupt [PORT1_VECTOR] void INPUT_ISR(void)
{
    if(P2IFG & BIT0)
    {
        //增加脉冲计数器
        nInputCount += 1;
        if(nInputCount >= 8 * N)
        {
            //行使到 1 公里
            nInputCount = 0;
            //设置标志
            nIn_Flag = 1;
        }
        // 清除中断标志位
        P2IFG &= ~(BIT0);
    }
}
```

在上面的中断处理程序中，每次进入中断都对脉冲数加 1，当脉冲数达到某一个值（1 公里的脉冲数）时，就设置“nIn_Flag”为 1，以通知其他程序进行处理。

定时器 B 中断处理程序主要是记录时间，下面为具体的程序代码。

```
interrupt [TIMERB0_VECTOR] void TimerB_ISR(void)
{
    //设置写标志
    nWrite_Flag = 1;
    //秒计数器加 1
    nTimeCount += 1;
    //解除写标志
    nWrite_Flag = 0;
}
```

在上面的程序中，每进入 1 次中断，就对秒计数器“nTimeCount”加 1，考虑其他模块也可能修改该变量，因此使用一个标志变量“nWrite_Flag”来进行数据保护。

20.2.3 主处理

主处理程序主要是检测是否行驶到 1 公里，当行驶到 1 公里时，就显示里程数据，并

往 EEPROM 里写入新的里程数据。主处理程序还需要从定时器 B 里获取秒计数器的值，从而获得速度信息，并对速度参数进行显示。下面为具体的程序代码。

```
#include "I2C.h"
#include "Led.h"

#define N    624
int nInputCount;
int nIn_Flag;
int nTimeCount;
int nWrite_Flag;

unsigned char seg[]={0x3f,0x06,0x5b,0x4f, /* 0 1 2 3*/
 0x66,0x6d,0x7d,0x07, /* 4 5 6 7 */
 0x7f,0x6f,0x77,0x7c, /* 8 9 A B */
 0x39,0x5e,0x79,0x71  /* C D E F */
};

void Init_TimerB(void);

void main(void)
{
    unsigned char nValue;
    int nTemp0;
    int nTemp1;
    int nTemp2;
    int nTemp3;
    int nTemp4;
    int nTemp5;
    int nSpeed;
    int nSpeed1;
    int nSpeed2;
    int nSpeed3;

    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;
    //关闭中断
    _DINT();

    //初始化时钟
    Init_CLK();
    //端口初始化
    Init_Port();
    I2C_Initial();

    nInputCount = 0;
    nIn_Flag = 0;
```

```
nTimeCount = 0;
nWrite_Flag = 0;
nTemp0 = 0;
nTemp1 = 0;
nTemp2 = 0;
nTemp3 = 0;
nTemp4 = 0;
nTemp5 = 0;

//打开中断
_EINT();

//读出初始的里程数
ReadRandom(0,&nTemp0);
if(nTemp0 == 0xff) nTemp0 = 0;
ReadRandom(1,&nTemp1);
if(nTemp1 == 0xff) nTemp1 = 0;
ReadRandom(2,&nTemp2);
if(nTemp2 == 0xff) nTemp2 = 0;
ReadRandom(3,&nTemp3);
if(nTemp3 == 0xff) nTemp3 = 0;
ReadRandom(4,&nTemp4);
if(nTemp4 == 0xff) nTemp4 = 0;
ReadRandom(5,&nTemp5);
if(nTemp5 == 0xff) nTemp5 = 0;
//显示初始里程数据
//清除锁存信号
STCLK_Lo();
DataOut(seg[nTemp5]);
DataOut(seg[nTemp4]);
DataOut(seg[nTemp3]);
DataOut(seg[nTemp2]);
DataOut(seg[nTemp1]);
DataOut(seg[nTemp0]);
//给锁存信号，显示上面的 6 位数据
STCLK_Hi();
for(;;)
{
    if(nIn_Flag == 1)
    {
        nIn_Flag = 0;
        //等待时间计数器设置完
        while(1)
        {
            if(nWrite_Flag == 0)
            {
                break;
            }
```

```
}

//计算速度
nSpeed = (int)(1 * 3600 / nTimeCount);
nTimeCount = 0;
nSpeed3 = (int)(nSpeed / 100);
nSpeed2 = (int)((nSpeed - 100 * nSpeed3) / 10);
nSpeed1 = nSpeed - 100 * nSpeed3 - 10 * nSpeed2;
//显示速度
//清除锁存信号
STCLK_Lo1();
DataOut1(seg[nSpeed3]);
DataOut1(seg[nSpeed2]);
DataOut1(seg[nSpeed1]);
//给锁存信号，显示上面的 3 位数据
STCLK_Hi1();

//处理里程参数
nTemp0 += 1;
if(nTemp0 >= 10)
{
    nTemp0 = 0;
    nTemp1 += 1;
    if(nTemp1 >= 10)
    {
        nTemp1 = 0;
        nTemp2 += 1;
        if(nTemp2 >= 10)
        {
            nTemp2 = 0;
            nTemp3 += 1;
            if(nTemp3 >= 10)
            {
                nTemp3 = 0;
                nTemp4 += 1;
                if(nTemp4 >= 10)
                {
                    nTemp4 = 0;
                    nTemp5 += 1;
                    if(nTemp5 >= 10)
                    {
                        nTemp0 = 0;
                        nTemp1 = 0;
                        nTemp2 = 0;
                        nTemp3 = 0;
                        nTemp4 = 0;
                        nTemp5 = 0;
                    }//if(nTemp5 >= 10)
```

```
                                }//if(nTemp4 >= 10)
                            }//if(nTemp3 >= 10)
                        }//if(nTemp2 >= 10)
                    }//if(nTemp1 >= 10)
                }//if(nTemp0 >= 10)
                //显示里程数据
                //清除锁存信号
                STCLK_Lo();
                DataOut(seg[nTemp5]);
                DataOut(seg[nTemp4]);
                DataOut(seg[nTemp3]);
                DataOut(seg[nTemp2]);
                DataOut(seg[nTemp1]);
                DataOut(seg[nTemp0]);
                //给锁存信号，显示上面的 6 位数据
                STCLK_Hi();
                //将里程数据写入 EEPROM
                WriteSingleByte(0,nTemp0);
                WriteSingleByte(1,nTemp1);
                WriteSingleByte(2,nTemp2);
                WriteSingleByte(3,nTemp3);
                WriteSingleByte(4,nTemp4);
                WriteSingleByte(5,nTemp5);
            }
        }

        return;
    }
```

20.3 实例总结

本章介绍了使用 MSP430 单片机实现的电子车速里程表，讨论了它的硬件设计，并给出了它的软件设计。随着电子技术在汽车行业应用的增加，电子式的车速里程表的应用会越来越多。本章只是给出一个最简单的设计，读者可以在此基础上设计更加完备的车速里程表，丰富它的功能，以使它能够得到大量的应用。

第21章

MSP430单片机与DS1820的接口设计与程序

美国Dallas半导体公司推出的数字化温度传感器DS1820采用单总线协议，无须任何外部元件，直接将温度转化成数字信号。本章介绍MSP430单片机与DS1820的接口设计，并给出具体的程序设计。

21.1 硬件设计

本系统的硬件接口电路相对比较简单，主要就是MSP430单片机与数字温度传感器DS1820的连接。在介绍具体的电路之前，先简要介绍一下数字温度传感器DS1820。

21.1.1 DS1820芯片

DS1820是美国Dallas半导体公司生产的数字式温度传感器，该芯片采用单总线协议，只需要单片机的一个一般I/O口就可以实现连接。该芯片封装很小，只有3个管脚。DS1820具有以下特性：

- 独特的单线接口方式，DS1820在与微处理器连接时仅需要一条总线即可实现微处理器与DS1820的双向通信。
- DS1820支持多点组网功能，多个DS1820可以并联在惟一的总线上，实现多点测温。

- DS1820 在使用中不需要任何外围元件。
- 可以由数据线供电。
- 测温范围–55℃～+125℃，固有测温分辨率 0.5℃。
- 测量结果以 9 位数字量方式串行传送。
- 用户可以设置报警温度。

为了增加对该芯片的认识，下面给出该芯片的方框图，如图 21-1 所示。

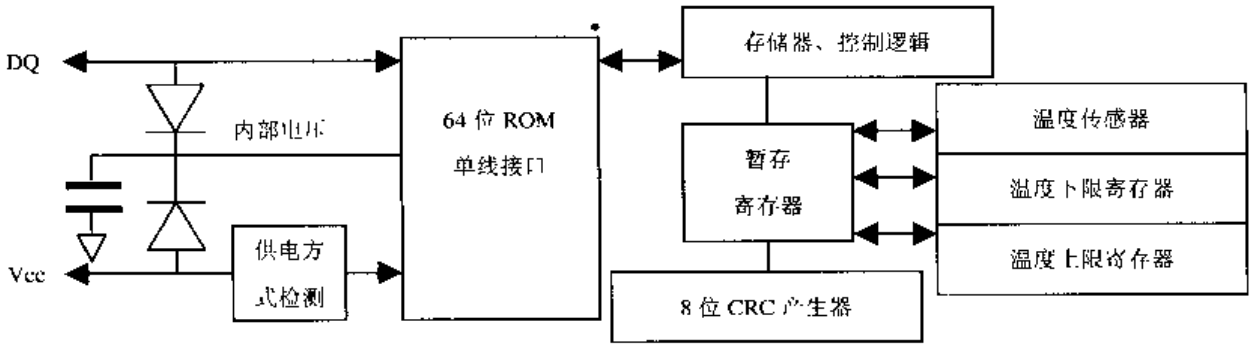


图 21-1　DS1820 芯片方框图

由图 21-1 可以看出，DS1820 主要有存储器、寄存器、控制逻辑、单线接口、温度传感器和温度上、下限寄存器等几个部分组成。为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 21-2 所示。

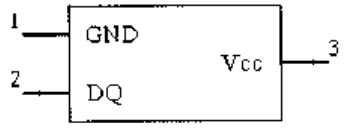


图 21-2　DS1820 芯片管脚图

由图 21-2 可以看出，该芯片共有 3 个管脚，为了了解各个管脚的功能，下面对具体的管脚进行介绍。

- Vcc：电源管脚。供电电压范围为 2.8~5.5V。
- GND：接地管脚。
- DQ：单线接口管脚，与单片机接口实现数据的传输。

经过上面对 DS1820 芯片的介绍，对该芯片有了基本的认识，下面介绍该芯片硬件电路的具体设计。

21.1.2　接口电路设计

由于 DS1820 芯片采用单线协议，即只需要一根线与 MSP430 单片机进行接口，因此接口电路非常简单。MSP430 单片机能通过端口的方向寄存器来设置端口的输入输出方向，因此能很好地实现单总线的数据读写功能。使用 MSP430 单片机的一般 I/O 口 P1.0 与 DS1820 芯片进行接口。如图 21-3 所示为具体的接口电路图。

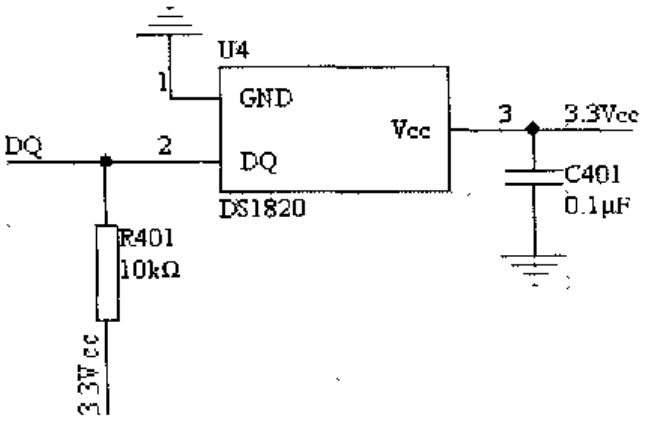


图 21-3　接口电路图

从图 21-3 可以看出：MSP430 单片机与 DS1820 芯片的接口非常简单。将 MSP430 单片机的 P1.0 管脚与 DS1820 芯片连接，实现数据的传输。在上面电路设计中，DQ 线通过 10kΩ的电阻拉高，保证总线在没有数据传输时始终为高电平。为了减少干扰，在电源的管脚增加一个 0.1μF 的电容来进行滤波处理。

21.2　软件设计

软件设计主要是对 DS1820 芯片进行读取温度的操作，在介绍具体的程序之前，先对单总线协议进行简单的介绍。

21.2.1　单总线协议

单总线协议能够实现数据的双向传输，这样就必须包括数据的读写，另外该总线还必须具有复位功能。下面就对各个方面进行具体的介绍。

1．总线复位

总线复位主要是起初始化总线的作用，在数据读写之前，都必须对总线进行复位。总线复位就是由单片机向总线发送低电平脉冲，然后等待 DS1820 的响应，DS1820 在接收到总线复位信号后给出相应的电平。如图 21-4 所示为总线复位的时序图。

由图 21-4 可以看出，低电平和高电平保持的时间是有限制的，这就要求在软件实现时特别注意高低电平的保持时间。

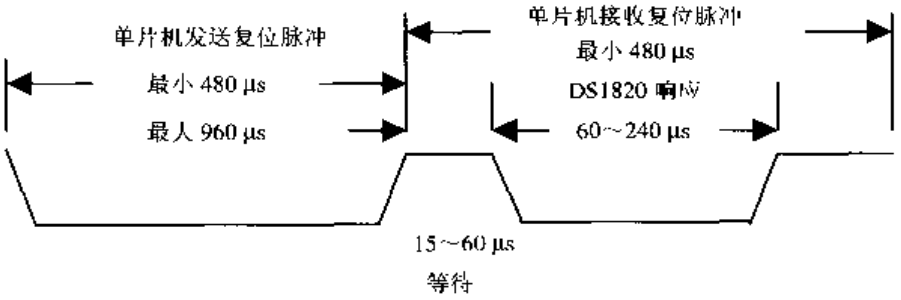


图 21-4　总线复位时序图

2. 总线写

总线写是单片机向 DS1820 写数据，由于总线数据传输是以位为单位的，总线写包括写“1”和写“0”。如图 21-5 所示为写“1”和写“0”的时序图。

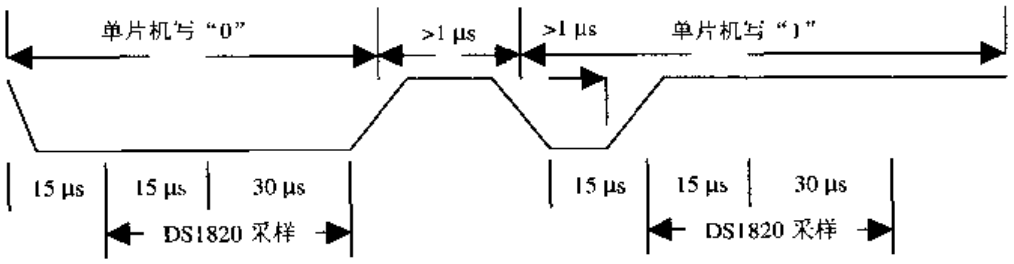


图 21-5　总线写时序图

由图 21-5 可以看出，单片机在写“1”或者写“0”的时候，首先需要将总线拉低，然后根据是“1”还是“0”来确定之后是保持高电平还是低电平，DS1820 根据总线的电平来进行采样，获得总线上写入的数据。低电平和高电平保持的时间是有限制的，这就要求在软件实现时特别注意高低电平的保持时间。

3. 总线读

总线读是单片机从 DS1820 读取数据，由于总线数据传输是以位为单位的，总线读包括读“1”和读“0”。如图 21-6 所示为读“1”和读“0”的时序图。

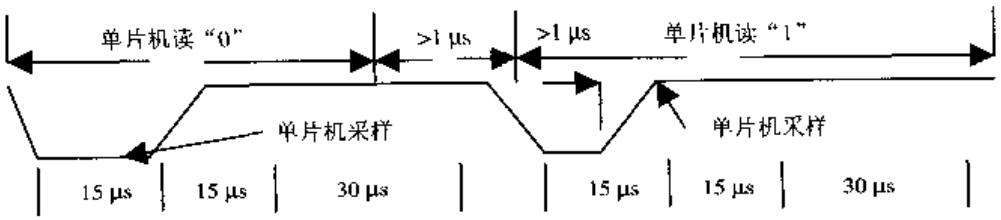


图 21-6　总线读时序图

由图 21-6 可以看出，单片机在读“1”或者读“0”的时候，首先需要将总线拉低，然后将总线拉高，再根据总线是高电平还是低电平来判断是“1”还是“0”，从而使单片机获取总线上的数据。

21.2.2　DS1820 操作

对 DS1820 的操作是通过操作内部的存储器来实现的。DS1820 的操作指令分为 ROM 操作命令和存储器操作命令。下面对操作命令分别进行简单介绍。

1．ROM 操作命令

- Read ROM（0x33）：如果只有一片 DS1820，可用此命令读出其序列号；若在线 DS1820 多于一个，将发生冲突。
- Match ROM（0x55）：多个 DS1820 在线时，可用此命令匹配一个给定序列号的 DS1820，此后的命令就针对该 DS1820。
- Skip ROM（0xCC）：此命令执行后的存储器操作将针对在线的所有 DS1820。
- Search ROM（0xF0）：用以读出在线的 DS1820 的序列号。
- Alarm Search（0xEC）：当温度值高于 TH 或低于 TL 中的数值时，此命令可以读出报警的 DS1820。

2．存储器操作命令

- Write Scratchpad（0x4E）：写两个字节的数据到温度寄存器。
- Read Scratchpad（0xBE）：读取温度寄存器的温度值。
- Copy Scratchpad（0x48）：将温度寄存器的数值拷贝到 EERAM 中，保证温度值不丢失。
- Convert T（0x44）：启动在线 DS1280 进行温度 A/D 转换。
- Recall E2（0xB8）：将 EERAM 中的数值拷贝到温度寄存器中。
- Read Power Supply（0xB4）：在本命令送到 DS1280 之后的每一个读数据间隙，指出电源模式：“0”为寄生电源；“1”为外部电源。

根据上面的操作命令，下面给出温度测量的步骤。

① Read ROM（33 h），每次对 DS1820 进行操作之前都要对它进行初始化，主要目的在于确定传感器已经连接到单总线上。

② Search ROM（F0h），这条指令使处理器用排除的方法去辨别总线上的 DS1820。

③ Match ROM（55h），只有准确地符合 64 位 ROM 序列的 DS1820 才能响应其后的指令，当然，单点测温时可以使用 Skip ROM（CCh）指令来跳过这一步。

④ Convert T（44h），发完指令后应查询总线上的电平，当电平位高时温度转换完成。

⑤ Read Scratchpad（BEh），将读指令发出后，就可从总线上获得表示温度的 2 字节二进制数。

21.2.3 DS1820 操作的程序实现

DS1820 的操作主要是按照上面的介绍操作相应的 ROM 或者 RAM 存储器，从而获得温度数据。DS1820 的读写存储器操作都是建立在单总线协议程序基础上的。下面首先介绍单总线协议的实现，然后介绍 DS1820 的温度数据读取程序。

1. 单总线协议的实现

单总线协议主要包括总线复位、读写字节操作，下面给出具体的程序代码。

```
char DS1820_Reset(void)
{
    char presence;
    // 设定管脚为输出方向
    P1DIR |= DQ;

    // 将 DQ 管脚拉低
    P1OUT &= ~(DQ);
    // 延时 480μs
    delay(480);
    // 将 DQ 管脚拉高
    P1OUT |= DQ;
    // 延时 60μs
    delay(60);

    // 设定管脚为输入方向
    P1DIR &= ~(DQ);
    // 读取数据
    presence = (char)(P1IN & DQ);
    // 延时 60μs
    delay(25);
    return(presence);
}
char DS1820_ReadByte(void)
{
    char i;
    char value = 0;
```

```
    char presence;

    for (i = 8;i > 0;i--)
    {
        value >>= 1;
        // 设定管脚为输出方向
        P1DIR |= DQ;
        // 将 DQ 管脚拉低
        P1OUT &= ~(DQ);
        // 将 DQ 管脚拉高
        P1OUT |= DQ;
        // 延时 1μs
        delay(1);
        // 设定管脚为输入方向
        P1DIR &= ~(DQ);
        // 读取数据
        presence = (char)(P1IN & DQ);
        if(presence) value |= 0x80;
        // 延时 60μs
        delay(60);
    }
    return value;
}
void DS1820_WriteByte(char val)
{
    char i;
    char nBit;
    for (i=8; i>0; i  )
    {
        // 设定管脚为输出方向
        P1DIR |= DQ;
        // 将 DQ 管脚拉低
        P1OUT &= ~(DQ);

        // 输出数据
        nBit = val & 0x01;
        if (nBit)
        {
            P1OUT |= DQ;
        }
        else
        {
            P1OUT &= ~(DQ);
        }
        // 延时 50μs
        delay(50);
        // 将 DQ 管脚拉高
        P1OUT |= DQ;
```

```
            val >>= 1;
        }
        // 延时5μs
        delay(5);
    }
```

上面的程序需要在总线的读或者写时切换管脚的输入输出方向，另外，需要注意延时的时间，这主要根据系统的时钟进行相应的调整。

2. DS1820温度数据的读取

由前面的操作介绍知道，操作DS1820的某些存储器就可以获得温度数据，下面给出具体的程序。

```
char Read_Temperature(void)
{
    union
    {
        byte c[2];
        int x;
    } temp;

    // 复位
    DS1820_Reset();

    // Skip ROM
    DS1820_WriteByte(0xCC);

    // 开始转换
    DS1820_WriteByte(0x44);
    // Read Scratch
    DS1820_WriteByte(0xBE);

    //读取温度数据
    temp.c[1]=DS1820_ReadByte();
    temp.c[0]=DS1820_ReadByte();

    // 返回数据
    return temp.x;
}
```

上面的程序只是进行简单的处理，在实践中，可能需要增加ROM命令的处理。总线上也可能有多个DS1820，那时需要根据具体的应用进行相应的修正。

21.3　实例总结

本章介绍了采用 MSP430 单片机与 DS1820 的接口设计，首先介绍电路的接口设计，然后介绍了相应的程序。DS1820 通过单总线与单片机进行连接，使系统的硬件复杂度非常小。在进行程序设计时，需要注意的是必须满足单总线协议工作的时序。通过本章的介绍，使读者能够了解 DS1820 的具体操作，读者也可以扩展硬件和软件功能，设计出满足自己系统需要的温度采集系统。

第 22 章

实时时钟芯片 DS1302 的设计与应用

在一些采集系统或者数据记录系统中需要记录事件发生的时间，在这些应用中，实现一个实时时钟是非常有必要的，这就要求单片机系统能够提供实时时钟功能。本章介绍 MSP430 单片机与实时时钟芯片 DS1302 的接口设计，并给出相应的程序设计。

22.1 硬件设计

本系统的硬件接口电路相对比较简单，主要就是 MSP430 单片机与时钟芯片 DS1302 的连接。为了便于理解接口，在介绍接口之前，先简要介绍一下时钟芯片 DS1302。

22.1.1 DS1302 芯片

DS1302 是美国 Dallas 公司推出的一款高性能、低功耗、带 RAM 的实时时钟芯片，它具有以下特点。

- 可以对年、月、日、周日、时、分、秒进行计时，且具有闰年补偿功能。
- 有 31 个字节的 RAM 可以存放数据。
- 采用串行接口，减少了管脚数。
- 工作电压为 2.0~5.5V。
- 功耗低，在 2V 电压工作时，电流只有 300nA。
- 可以实现单字节读写，也可以实现多字节读写。
- 具有很小的封装，只有 8 个管脚。

为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 22-1 所示。

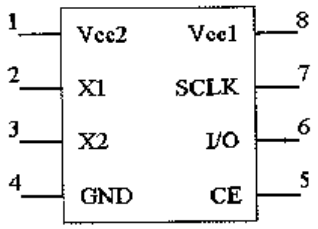


图 22-1　DS1302 芯片管脚图

由图 22-1 可以看出，该芯片只有 8 个管脚，这样使用起来简单，只需要简单的外围电路。下面对具体的管脚进行介绍。

- Vcc1、Vcc2：电源管脚。其中 Vcc1 为备用电源管脚。
- X1、X2：外接 32kHz 晶体的管脚。
- GND：接地管脚。
- CE：片选管脚。
- I/O：数据输入输出管脚。
- SCLK：串行时钟输入管脚。

经过对 DS1302 芯片的简单介绍，对该款时钟芯片有了基本的认识，下面介绍硬件电路的具体设计。

22.1.2　接口电路设计

硬件的接口电路主要就是 MSP430 单片机与时钟芯片 DS1302 的接口设计，如图 22-2 所示为接口电路图。

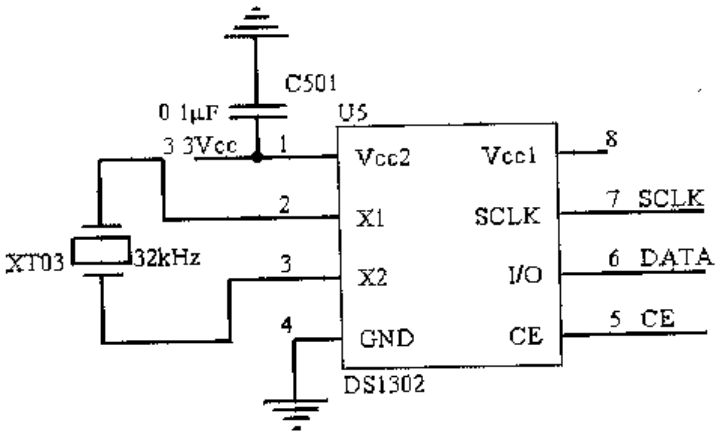


图 22-2　接口电路图

由图 22-2 可以看出，DS1302 的外围电路非常简单。X1 和 X2 管脚外接 32kHz 的晶体，提供振荡源。Vcc2 管脚为主要的供电管脚。Vcc1 管脚为备用电源输入，可以不使用，如果需要使用时，可以外接电池或者充电电容。DS1302 工作的时候，在 Vcc2 和 Vcc1 两个管脚中，选择电压高的那个管脚的电源作为工作的电源。DS1302 的 CE、I/O 和 SCLK 管脚主要与单片机的一般 I/O 口进行连接，实现数据的传输，从而使单片机能够对 DS1302 进行读写操作。

22.2 软件设计

整个软件设计相对比较简单，主要完成对时钟芯片 DS1302 的读写，从而获得日历数据。在介绍具体程序实现前，先对 DS1302 的操作进行简单的介绍。

22.2.1 DS1302 的操作

对 DS1302 的操作主要是通过对它的寄存器操作来实现，其中 DS1302 的控制寄存器用来选择操作哪个寄存器，并且控制数据输入和输出的方向，即读或者写。如图 22-3 所示为控制寄存器的各个位。

1	RAM/CK	A4	A3	A2	A1	A0	RD/WR

图 22-3 控制寄存器的各个位

由图 22-3 可以看出，控制字节的最高有效位（第 7 位）必须是逻辑 1，如果它为 0，则不能把数据写入到 DS1302 中。第 6 位表示为日历数据还是 RAM 数据，如果该位为 0，则表示存取日历时钟数据；如果该位为 1，则表示存取 RAM 数据。第 5 位至第 1 位用来指示操作单元的地址。最低有效位（第 0 位）表示读写控制位，如果该位为 0，则表示要进行写操作；如果该位为 1，则表示进行读操作。控制字节总是从最低位开始输出。

针对单片机适当设置 DS1302 的控制寄存器，就能操作 DS1302 内部具体的某个寄存器或某个地址的 RAM。DS1302 内部有“年、月、日、时、分、秒”等寄存器，也有 RAM，具体的寄存器这里不再详细介绍，读者可以参看 DS1302 的数据手册。由于 DS1302 的“年、月、日、时、分、秒”寄存器里面的内容是按 BCD 编码进行存储的，所以在写程序时需要注意。

一般情况下，对 DS1302 的操作可以进行单字节读写，在进行单字节读写时，需要发送一个命令，再进行一个字节读写。为了提高处理的效率，对 DS1302 的操作也可以实现

连续读写，比如连续读出“年、月、日、时、分、秒”内容来。单字节操作和多字节操作区别在于命令不同，并且一次操作的数据个数不同。经过对 DS1302 操作的简单介绍，对 DS1302 的数据读写有了一个基本的概念，下面介绍具体的程序实现。

22.2.2　程序设计

整个程序设计主要包括端口初始化及端口电平模拟、读写操作、时间处理等。下面对各个部分程序分别进行详细介绍。

1. 端口初始化及端口电平模拟

端口初始化的主要作用是为控制信号线设置正确的输入输出方向。端口电平模拟主要作用是在 CE 管脚和 SCLK 管脚上产生高电平或者低电平。具体的程序如下：

```
void Port_Init(void)
{
    P1DIR = 0;
    //设置 CE 为输出管脚
    P1DIR |= CE;
    //设置 SCLK 为输出管脚
    P1DIR |= SCLK;
    return;
}
void CE_Enable(void)
{
    P1OUT |= CE;
    return;
}
void CE_Disable(void)
{
    P1OUT &= ~(CE);
    return;
}
void SCLK_HI(void)
{
    P1OUT |= SCLK;
    return;
}
void SCLK_LO(void)
{
    P1OUT &= ~(SCLK);
    return;
}
```

在上面的程序中，“CE”、“SCLK”分别为定义的常量，对应相应的管脚。

2. 读写操作

读写操作主要处理数据的发送与接收，并且能对相应的地址进行读写操作以获得相关的数据。具体程序如下：

```
void WriteByte(char nVal)
{
    char i,j;
    char nTemp = nVal;
    char nSend;
    //设置 DATA 为输出管脚
    P1DIR |= DATA;
    _NOP();
    _NOP();
    _NOP();
    _NOP();
    for(i = 0; i < 8; i++)
    {
        nSend = (nTemp & 0x01);
        if(nSend == 1)
        {
            P1OUT |= DATA;
        }
        else
        {
            P1OUT &= ~(DATA);
        }
        SCLK_HI();
        for(j = 10;j > 0;j--) ;
        SCLK_LO();
        for(j = 10;j > 0;j--) ;
        nTemp >>= 1;
    }
}
char ReadByte(void)
{
    char nTemp = 0;
    int i;
    int j;
    //设置 DATA 为输入管脚
    P1DIR |= DATA;
    _NOP();
    _NOP();
    _NOP();
    _NOP();
```

```
    for(i = 0; i < 8; i++)
    {
        SCLK_HI();
        if(P1IN & SDA)
        {
            nTemp |= (0x01 << i);
        }
        for(j = 10;j > 0;j--);
        SCLK_LO();
    }
    return nTemp;
}
```

上面的程序主要处理数据的发送与接收。在上面的程序基础上，可以实现对 DS1302 按照地址读写的程序。具体程序如下：

```
void WriteTo1302(char nAddr, char nVal)
{
    CE_Disable();
    SCLK_LO();
    CE_Enable();
    //地址，命令
    WriteByte(nAddr);
    //写 1Byte 数据
    WriteByte(nVal);
    SCLK_HI();
    CE_Disable();
    return;
}
char ReadFrom1302(char nAddr)
{
    char nData;
    CE_Disable();
    SCLK_LO();
    CE_Enable();
    //地址，命令
    WriteByte(nAddr);
    //读 1Byte 数据
    nData = ReadByte();
    SCLK_HI();
    CE_Disable();
    return(nData);
}
```

上面的程序可以实现往指定的地址写入数据，也可以从指定的地址读出数据。

3. 时间处理

在上面的程序基础上，就可以实现时间的设置、时间的读取、寄存器的读写等操作，在操作的时候需要向 DS1302 发送不同的命令以区别进行不同的操作。另外，在操作的时候可以按照单个字节进行处理，也可以一次处理多个字节。具体程序如下：

```
void BurstWriteTime(char *pClock)
{
    char i;
    //控制命令，WP=0，写操作
    WriteTo1302(0x8e,0x00);
    CE_Disable();
    SCLK_LO();
    CE_Enable();
    //0xbe:时钟多字节写命令
    WriteByte(0xbe);
    //8Byte = 7Byte 时钟数据 + 1Byte 控制
    for (i = 8; i > 0; i--)
    {
        //写 1Byte 数据
        WriteByte(*pClock);
        pClock++;
    }
    SCLK_HI();
    CE_Disable();
    return;
}
```

上面的代码主要采用“Burst”的方式设置时间信息。

```
void BurstReadTime(char *pClock)
{
    char i;
    CE_Disable();
    SCLK_LO();
    CE_Enable();
    //0xbf:时钟多字节读命令
    WriteByte(0xbf);
    for (i = 8; i > 0; i--)
    {
        //读 1Byte 数据
        *pClock = ReadByte();
        pClock++;
    }
    SCLK_HI();
    CE_Disable();
```

```
    return ;
}
```

上面的代码主要采用“Burst”的方式读取时间信息。

```
void BurstWriteRam(char *pReg)
{
    char i;
    //控制命令，WP=0，写操作
    WriteTo1302(0x8e,0x00);
    CE_Disable();
    SCLK_LO();
    CE_Enable();
    //0xfe:时钟多字节写命令
    WriteByte(0xfe);

    //31Byte 寄存器数据
    for (i = 31; i > 0; i--)
    {
        //写 1Byte 数据
        WriteByte(*pReg);
        pReg++;
    }
    SCLK_HI();
    CE_Disable();
    return;
}
```

上面的代码主要采用“Burst”的方式设置寄存器。

```
void BurstReadRam(char *pReg)
{
    char i;
    CE_Disable();
    SCLK_LO();
    CE_Enable();
    //0xff:时钟多字节读命令
    WriteByte(0xff);
    //31Byte 寄存器数据
    for (i = 31; i > 0; i--)
    {
        //读 1Byte 数据
        *pReg = ReadByte();
        pReg++;
    }
    SCLK_HI();
    CE_Disable();
    return;
}
```

上面的代码主要采用“Burst”的方式读取寄存器内容。

```
void SetTime(char *pClock)
{
    char i;
    char nAddr = 0x80;
    //控制命令，WP=0，写操作
    WriteTo1302(0x8e,0x00);
    for(i = 7; i > 0; i--)
    {
        //秒 分 时 日 月 星期 年
        WriteTo1302(nAddr,*pClock);
        pClock++;
        nAddr += 2;
    }
    //控制命令，WP=1，写保护
    WriteTo1302(0x8e,0x80);
    return;
}
```

上面的代码主要采用单个字节的方式设置时间信息。

```
void GetTime(char pTime[])
{
    char i;
    char nAddr = 0x81;
    for (i = 0; i < 7; i++)
    {
        //格式为：秒 分 时 日 月 星期 年
        pTime[i] = ReadFrom1302(nAddr);
        nAddr += 2;
    }
}
```

上面的代码主要采用单个字节的方式读取时间信息。由于 DS1302 的时间寄存器都是按照 BCD 码进行存放数据的，因此上面的时间信息是按照 BCD 进行处理的；如果入口参数是十进制或十六进制数据的话，则需要进行相应的处理。

22.3 实例总结

DS1302 只有 8 个管脚，使系统的硬件设计非常简单。在软件设计时，使用 MSP430 单片机的一般 I/O 口按照 DS1302 的时序来实现数据读写，这样可以使程序应用于任何一款 MSP430 单片机。本章介绍的系统虽然简单，但用户完全可以在该基础上进行扩展升级，实现更为完善的时钟系统，以满足用户的要求。

第 23 章

基于 BQ26500 实现的电源监测系统

在 PDA、手机等手持设备中，经常需要对使用的电池进行检测，判断电量还有多少，或者在充电的时候判断是否充满。本章介绍利用 BQ26500 实现的一个电源监测系统，首先介绍它的硬件设计，然后给出它的软件设计。

23.1 硬件设计

本系统的硬件接口电路相对比较简单，主要就是 MSP430 单片机与 BQ26500 芯片的连接。在介绍具体的电路之前，首先简要介绍一下 BQ26500 芯片。

23.1.1 BQ26500 芯片

BQ26500 是 TI 公司的一款电池测量芯片。该芯片具有以下特性：

- 能够报告电池（如锂电池）的充电状态。
- 能够报告电池的温度和电压。
- 不需要偏移校正。
- I/O 端口可以通过程序来设置。
- 不需要外接时钟源。
- 有 4 种低功耗模式。

为了便于进行硬件电路的设计，下面给出该芯片的管脚图，如图 23-1 所示。

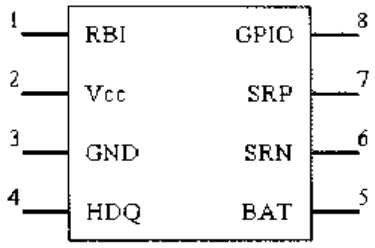


图 23-1　芯片管脚图

由图 11-1 可以看出，该芯片只有 8 个管脚，这样使用起来简单，只需要简单的外围电路，下面对具体的管脚进行介绍。

- RBI：寄存器备份输入。
- Vcc：电源管脚。
- GND：接地管脚。
- HDQ：HDQ 管脚，用来与单片机进行数据传输。
- BAT：电池电压传感输入管脚。
- SRN：电流传感负输入管脚。
- SRP：电流传感正输入管脚。
- GPIO：一般 I/O 口，可以设置为输入管脚，也可以设置为输出管脚。

经过对 BQ26500 芯片的管脚介绍，对各个管脚有了基本的认识，下面介绍具体的接口电路。

23.1.2　接口电路设计

由前面介绍的管脚知道，BQ26500 芯片主要通过 HDQ 管脚与单片机进行接口。此外，BQ26500 有电压传感输入和电流传感输入，因此可以实现电压的测量和充电放电的测量。下面给出该芯片的简单接口电路，如图 23-2 所示。

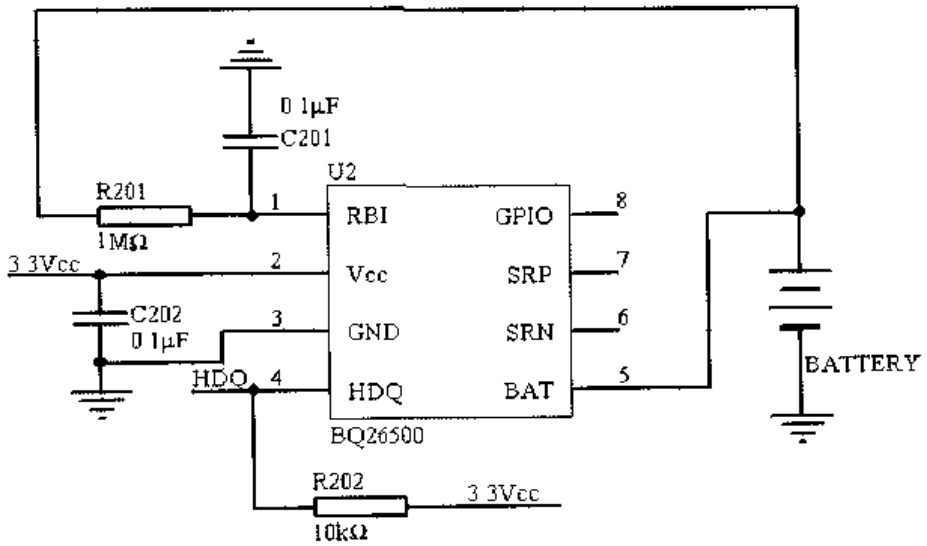


图 23-2　接口电路

在图 23-2 中，BQ26500 的 HDQ 管脚与单片机的 P1.1 管脚进行连接，实现数据传输

的功能；在利用 P1.1 管脚时，使用的是定时器 A 的功能，而不是一般 I/O 口。将 BQ26500 芯片的 BAT 管脚与电池连接，实现对电池电压的测量。将 RBI 管脚通过 1MΩ的电阻与电池连接，并且该管脚连接一个电容，这样当电池的电压高于门限值时就充电；当电压低于门限值时，就可以继续工作一段时间，使该芯片记录当前的状态信息。GPIO、SRP 和 SRN 管脚暂时没有使用，在实际使用中可以充分利用 SRP 和 SRN 管脚实现充电和放电的测量，也可以利用 GPIO 外接发光二极管来实现一些状态信息的指示。本章主要讨论 BQ26500 与单片机的数据通信，因此对接口电路做了一定的简化，如果需要设计更加完善的接口电路，读者可以查看 BQ26500 的数据手册。由于单片机电路相对比较简单，这里就不再给出具体的电路图了。

23.2　软件设计

本系统的软件设计主要就是单片机对 BQ26500 进行读写操作，从而获取电池的容量等状态信息。由于单片机和 BQ26500 只通过 HDQ 管脚进行连接，所以数据的传输是通过一根线来完成的。在使用一线通信时，必须满足一定的规范，即满足 HDQ 协议。下面首先对 HDQ 总线做一个简单介绍。

23.2.1　HDQ 总线

HDQ 总线是一线传输总线，它是主从方式的总线，可以实现数据的双向传输。HDQ 总线一般需要使用外部电阻进行上拉。数据传输采用异步方式，数据传输的速率大约为 5kbps。HDQ 总线在传输数据时，以 8 位为一组，一位一位进行传输，并且是从最低位开始传输的。每一位的传输都是以高电平转换为低电平开始，“0”和“1”是通过低电平保持的时间不同来区分的。如图 23-3 所示为 HDQ 总线的“1”和“0”的表示。

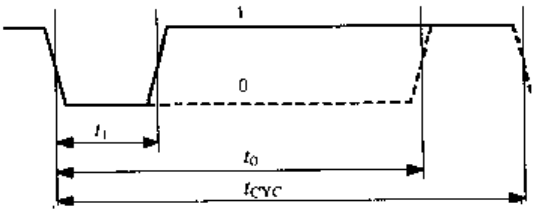


图 23-3　HDQ 总线的“1”和“0”的表示

由图 23-3 可以看出，“1”和“0”都是以高电平到低电平转换的时候开始为低电平，如果在经过时间 t_1 后，电平变为高电平，则代表“1”；如果经过时间 t_0 后才变为高电平，

则代表“0”。t_{CYC} 为位的周期时间。需要注意的是主机发送和 BQ26500 发送数据的时间有所不同，具体的时间参考值查看具体的数据手册。

HDQ 总线一般包括两个部分：Break 及 Break 恢复和数据传输。其中在数据传输的时候，根据传输的方向，位的时序在时间和周期上不同，即单片机发送的“1”和“0”的时序与 BQ26500 发送的“1”和“0”的时序不一致，具体的时序参看 BQ26500 的数据手册。在传输数据前，都需要 Break 总线，这和 I^2C 总线的起始条件类似。如图 23-4 所示为 HDQ 总线的 Break 及 Break 恢复的时序图。

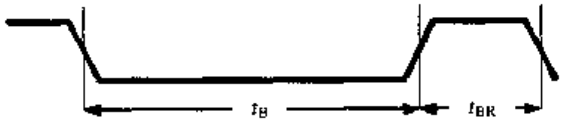


图 23-4　Break 及 Break 恢复时序

由图 23-4 可以看出，当电平由高电平变为低电平，并且低电平保持 t_B 时间时，则为 Break；然后由低电平变为高电平，并且高电平持续时间为 t_{BR} 时，则为 Break 恢复，在这以后 HDQ 总线就可以开始传输数据。传输数据一般传输两个字节，第一个字节由 7 个地址位和 1 个读写位组成，读写位主要用来指示是读出数据还是写入数据；第二个字节为数据字节。如图 23-5 所示为 HDQ 总线数据传输的时序图。

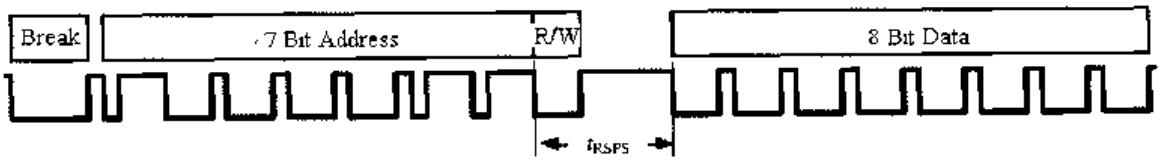


图 23-5　HDQ 总线数据传输时序图

在图 23-5 中，t_{RSPS} 表示 BQ26500 响应单片机的时间。以上介绍了 HDQ 总线，下面介绍 HDQ 协议的实现。

23.2.2　HDQ 协议的实现

HDQ 协议的实现主要包括初始化的设置、Break 的实现、数据读写操作等。下面对各个部分进行具体的介绍。

1．初始化设置

初始化设置这部分代码主要是完成时钟的初始化和端口管脚的初始化，下面为具体的程序代码。

```
void Init_CLK(void)
{
    unsigned int i;
    //将寄存器的内容清零
    //XT2 震荡器开启
    //LFTX1 工作在低频模式
    //ACLK 的分频因子为 1
    BCSCTL1 = 0X00;

    do
    {
        // 清除 OSCFault 标志
        IFG1 &= ~OFIFG;
        for (i = 0x20; i > 0; i--);
    }
    while ((IFG1 & OFIFG) == OFIFG);

    BCSCTL2 = 0X00;
    //MCLK 的时钟源为 TX2CLK，分频因子为 4
    BCSCTL2 += SELM1 + DIVM_2;
    //SMCLK 的时钟源为 TX2CLK，分频因子为 4
    BCSCTL2 += SELS + DIVS_2;
}
void HDQSetup(void)
{
    // P1.1 为输出
    P1DIR |= BIT1;
    // 选择 TA0 功能
    P1SEL |= 0x02;
}
```

在上面的程序中，将 SMCLK 的频率设置为 2MHz，将 P1.1 管脚设置为 TA0 功能。

2. Break 的实现

Break 的实现主要按照图 23-4 所给的时序来实现。具体代码如下：

```
static void HDQBreak(void)
{
    // 设置 Break 的时间
    TACCR0 = TAR + tBreak * 2;
    // 复位 OUT0，使能中断
    TACCTL0 = OUTMOD_0 + CCIE;
    // 设置中断服务为延时模式
    TSRMode = imDelay;
    // 进入低功耗模式
    _BIS_SR(LPM0_bits);
    // 设置 Break 的恢复时间
```

```
    TACCR0 += tBR;
    // 设置 OUT0，使能中断
    TACCTL0 = OUTMOD_0 + OUT + CCIE;
    // 进入低功耗模式
    _BIS_SR(LPM0_bits);
}
```

在上面的程序中，设置好 Break 的时间后就进入中断服务程序，进入低功耗状态；在将低电平延时所给的时间后，设置 Break 的恢复时间，进入低功耗状态，等待中断处理的完成。以上代码是在设置好响应的时间后就交给中断服务程序处理，具体的中断服务程序在后面进行解释。

3. 数据的读写

HDQ 数据的读写也是按照总线的读写时序来进行处理的。下面为具体的程序代码。

```
static void HDQBasicWrite(unsigned char Data)
{
    Xfer = Data;
    // 复位 OUT0
    TACCTL0 = OUTMOD_0;

    TACCR0 = TAR;
    // 位开始的时间戳
    Ticks = TACCR0;

    // 设置低电平的时间
    if (Xfer & 0x01)
        // "1"
        TACCR0 += tHW1;
    else
        // "0"
        TACCR0 += tHW0;

    // Toggle OUT0，中断使能
    TACCTL0 = OUTMOD_4 + CCIE;
    // 设置传输的位数为 8
    BitCnt = 8;
    // 中断服务为写模式
    ISRMode = imWrite;
    // 进入低功耗模式
    _BIS_SR(LPM0_bits);
}
```

上述代码主要完成数据按一位一位进行发送的处理，具体的处理也是由中断服务程序来完成。具体写操作的代码这里不单独介绍，下面结合 BQ26500 具体的操作进行介绍。

23.2.3　BQ26500 操作的实现

BQ26500 的操作主要是按照如图 23-5 所示的时序来进行数据的读写。下面给出具体的程序。

```
void HDQWrite(unsigned char Addr, unsigned char Data)
{
    // 选择 SMCLK 为时钟源，连续模式
    TACTL = TASSEL_2 + MC_2;
    // P1.1 为输出
    P1DIR |= 0x02;
    // 总线 Break
    HDQBreak();
    // 发送地址，确保 R/W 比特为 0
    HDQBasicWrite(Addr | 0x80);

    // 延时
    // 设置时间
    TACCR0 += tCYCH;
    // 设置 OUT0，中断使能
    TACCTL0 = OUTMOD_0 + OUT + CCIE;
    // 设置中断服务为延时模式
    ISRMode = imDelay;
    // 进入低功耗
    _BIS_SR(LPM0_bits);

    // 写入数据
    HDQBasicWrite(Data);
    // P1.1 为输入
    P1DIR &= ~0x02;
    // 停止 Timer_A
    TACTL = 0;
}
// HDQ 的读操作
unsigned int HDQRead(unsigned char Addr)
{
    // 选择 SMCLK 为时钟源，连续模式
    TACTL = TASSEL_2 + MC_2;
    // P1.1 为输出
    P1DIR |= 0x02;
    // 总线 Break
    HDQBreak();
    // 发送地址
    HDQBasicWrite(Addr);
    // P1.1 为输入
```

```
    P1DIR &= ~0x02;

    // 设置数据为8位
    BitCnt = 8;
    // 设置中断服务为读模式
    ISRMode = imRead;

    // 在P1.1管脚捕获下降沿
    TACCTL0 = CM_2 + CCIS_0 + SCCI + CAP + CCIE;
    // 设置超时时间
    TACCR1 = TAR + tTO;
    // 使能中断
    TACCTL1 = CCIE;

    // 进入低功耗
    _BIS_SR(LPM0_bits);
    // 停止Timer_A
    TACTL = 0;

    // 判断数据是否有效
    if (BitCnt)
        return 0xffff;
    else
        return Xfer;
}
```

在上面的程序中，都是设置相应的时间，然后具体的处理都由中断服务程序来完成。中断服务程序的代码如下：

```
interrupt [TIMERA0_VECTOR] void Timer_A0_ISR(void)
{

    switch (ISRMode)
    {
    case imWrite :
        if (--BitCnt)
        {
            // 设置时间
            TACCR0 = Ticks + tCYCH;
            ISRMode = imWriteE;
        }
        else
        {
            // 设置OUT0，禁止中断
            TACCTL0 = OUTMOD_0 + OUT;
            // 返回激活状态
            _BIC_SR(LPM0_bits);
        }
```

```
        break;
    case imWriteE :
        // 位开始时间戳
        Ticks = TACCR0;

        if ((Xfer >>= 1) & 0x01)
            // "1"
            TACCR0 += tHW1;
        else
            // "0"
            TACCR0 += tHW0;

        ISRMode = imWrite;
        break;
    case imRead :
        // 停止超时
        TACCTL1 = 0;
        // 在位的中央采样
        TACCR0 += (tDW0 + tDW1) / 2;
        // 捕获模式
        TACCTL0 &= ~CAP;
        ISRMode = imReadE;
        break;
    case imReadE :
        Xfer >>= 1;
        // 检查 Timer_A 的锁存
        if (TACCTL0 & SCCI)
            Xfer |= 0x80;
        if (--BitCnt)
        {
            // 捕获模式
            TACCTL0 |= CAP;
            // 设置超时
            TACCR1 = TAR + tTO;
            // 使能中断
            TACCTL1 = CCIE;
            // 读模式
            ISRMode = imRead;
        }
        else
        {
            // 设置 OUT0，禁止中断
            TACCTL0 = OUTMOD_0 + OUT;
            // 返回激活状态
            _BIC_SR(LPM0_bits);
        }
        break;
    case imDelay :
```

```
            // 禁止中断
            TACCTL0 &= ~CCIFG;
            // 返回激活状态
            _BIC_SR(LPM0_bits);
            break;
        }
    }
```

上面的中断服务程序根据不同的状态进行相应的处理。此外，由于设置了超时，具体的超时也是由中断服务程序来进行处理。具体程序如下：

```
interrupt [TIMERA1_VECTOR] void Timer_A1_ISR(void)
{
    if (TAIV == 0x02)
    {
        TACCTL0 = 0;
        TACCTL1 = 0;
        // 返回激活状态
        _BIC_SR(LPM0_bits);
    }
}
```

以上介绍了 BQ26500 操作的实现，用户如果需要读取数据（比如电压、温度），只需要调用“HDQ Read(unsigned char Addr)”函数就可以了，在调用的时候，需要传入相应寄存器的地址，寄存器的地址查看 BQ26500 数据手册就可以知道。

23.3 实例总结

本章介绍了以 BQ26500 芯片为测量芯片的电池检测系统。在实现该系统的时候，主要需要实现 HDQ 协议，实现 HDQ 协议的时候必须满足相应的时间参数。虽然本章介绍的系统相对比较简单，但是读者可以在此基础上适当进行修改，就很容易实现满足自己要求的电池检测系统。

第五篇

通信应用

第 24 章

基于 MSP430 实现的红外传输系统

随着移动计算设备和移动通信设备的日益普及，红外数据通信应用越来越多。红外通信技术由于成本低廉和广泛的兼容性等优点，红外通信将在近距离的无线数据传输领域扮演重要角色。本章介绍一种基于 MSP430 单片机实现的红外传输系统，首先介绍它的硬件设计，然后介绍它的软件设计。

24.1 硬件设计

红外数据通信指的是两台设备之间通过红外线进行无线数据传输的一种数据传输方式，一般采用红外波段内的近红外线，波长在 0.75μm～25μm之间。红外数据协会成立后，为了保证不同厂商的红外产品能够获得最佳的通信效果，将红外数据通信所采用的光波波长的范围限定在 850nm～900nm 之间。红外通信的最大特点在于它替代了设备与设备之间传统的线缆连接，进而摆脱了不同平台设备连接时对于特制接口的要求，使得跨平台设备间的数据交换简单到只需彼此相对。红外通信主要有以下特点：

- 它是目前在世界范围内被广泛使用的一种无线连接技术，被众多的硬件和软件平台所支持。
- 小角度（30° 以内）、短距离、点对点直线数据传输。
- 通信过程中不能移动，遇到障碍物就会使通信中断。
- 传输速率较高，SIR 标准下的最高速率可以达到 152kbps，FIR 标准下的最高速率为 4Mbps。

本章介绍的红外通信系统主要由编解码器、收发器及单片机组成。收发器主要实现以红外线方式进行数据的收发。编解码器主要是完成对发送的数据进行编码和对接收到的数据进行解码。本系统的编解码器为 HDSL-7001 芯片，收发器为 HDSL-3201 芯片，在介绍具体的接口电路前，先对这两款芯片进行介绍。

24.1.1　HDSL-7001 芯片

在红外传输中，串行红外传输采用特定的脉冲编码标准，这种标准与单片机的 RS232 串行传输标准不同。采用单片机实现的红外传输系统，就需要进行 RS232 编码和红外传输编码之间的转换，本系统使用的 HDSL-7001 芯片就是实现它们之间的编码转换。HDSL-7001 芯片主要有以下特点：

- 满足 IrDA1.0 物理层规范。
- 接口与 SIR 收发器相兼容。
- 可与标准的串口芯片 16550 连接使用，也可以与单片机连接使用。
- 可发送/接收 1.63μs 或 3/16 脉冲形式。
- 内部或外部时钟模式。
- 波特率可编程。
- 工作电压范围为 2.7V～5.5V。
- 采用 16 脚 SOIC 封装。

为了便于进行硬件电路的设计，下面给出 HDSL-7001 芯片的管脚图，如图 24-1 所示。

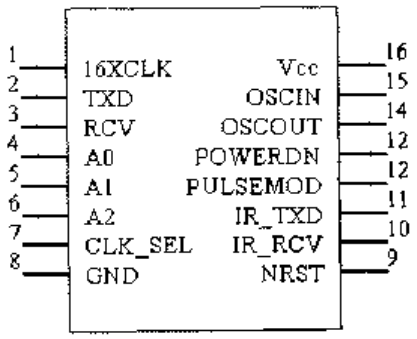


图 24-1　HDSL-7001 芯片管脚图

由图 24-1 可以看出，该芯片有 16 个管脚。下面对具体的管脚进行介绍。

- 16XCLK：外部时钟输入管脚。输入的时钟是数据传输波特率的 16 倍。
- TXD：串口输入管脚。接收来自串口的数据，将数据调制成红外发送数据。
- RCV：串口输出管脚。将接收到的红外数据解调后输出给串口。

- A0、A1、A2：时钟的除法系数选择管脚。
- CLK_SEL：时钟选择管脚。该管脚输入为高电平，选择外部时钟，即时钟为 16XCLK 管脚输入的时钟。
- GND：接地管脚。
- Vcc：电源管脚。
- NRST：复位管脚。
- IR_RCV：红外数据接收管脚。
- IR_TXD：红外数据发送管脚。
- PULSEMOD：脉冲模式选择管脚。
- POWERDN：低功耗选择管脚。如果该管脚输入高电平，芯片进入低功耗模式。
- OSCOUT、OSCIN：晶体振荡电路的输入输出管脚。

以上只是简单介绍了 HDSL-7001 芯片的各个管脚，如果需要详细了解的话，可以参看 HDSL-7001 的数据手册。

24.1.2 HDSL-3201 芯片

HDSL-3201 芯片是一种廉价的红外收发器模块，工作电压为 2.7V～3.6V。由于发光二极管的驱动电流是内部供给的恒流 32mA，因此确保了连接距离符合 IrDA1.2（低功耗）物理层规范。HDSL-3201 芯片主要有以下特点：

- 超小型表面封装。
- 最小高度为 2.5mm。
- 发光二极管电压范围为 2.7V～6.0V。
- 温度范围：–25℃～85℃。
- 发光二极管驱动电流为 32mA。
- 边缘检测输入：避免发光二极管的开启时间过长。

为了便于进行硬件电路的设计，下面给出 HDSL-3201 芯片的管脚图，如图 24-2 所示。由图 24-2 可以看出，该芯片只有 8 个管脚。下面对具体的管脚进行介绍。

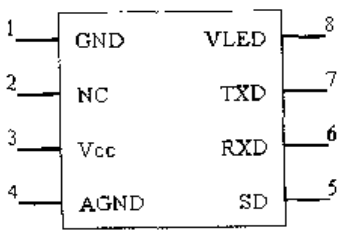


图 24-2 HDSL-3201 芯片管脚图

- Vcc：电源管脚。
- GND：接地管脚。
- AGND：模拟接地管脚。
- SD：低功耗管脚。如果该管脚输入高电平，则芯片进入低功耗状态。
- TXD：传输数据输入管脚。

- RXD：接收数据输出管脚。
- VLED：LED 的电源管脚。

经过对 HDSL-7001 芯片和 HDSL-3201 芯片的介绍，对该红外数据传输有了基本的认识，下面介绍硬件电路的具体设计。

24.1.3　接口电路设计

本系统主要是将单片机串口发送的数据由 HDSL-7001 芯片按照红外传输的格式进行编码，将编码后的数据由 HDSL-3201 芯片进行发送。HDSL-3201 芯片接收另一个红外设备发送的数据，将接收到的红外数据交给 HDSL-7001 芯片进行解码处理，HDSL-7001 芯片将解码后的数据传给单片机。

由上面的分析可以知道，整个硬件电路包括单片机电路、红外编解码电路、红外收发电路。单片机电路通过串口与 HDSL-7001 芯片进行连接，从而实现串口数据与红外编解码电路进行数据传输。红外编解码器主要由 HDSL-7001 芯片实现，如图 24-3 所示为红外编解码电路图。

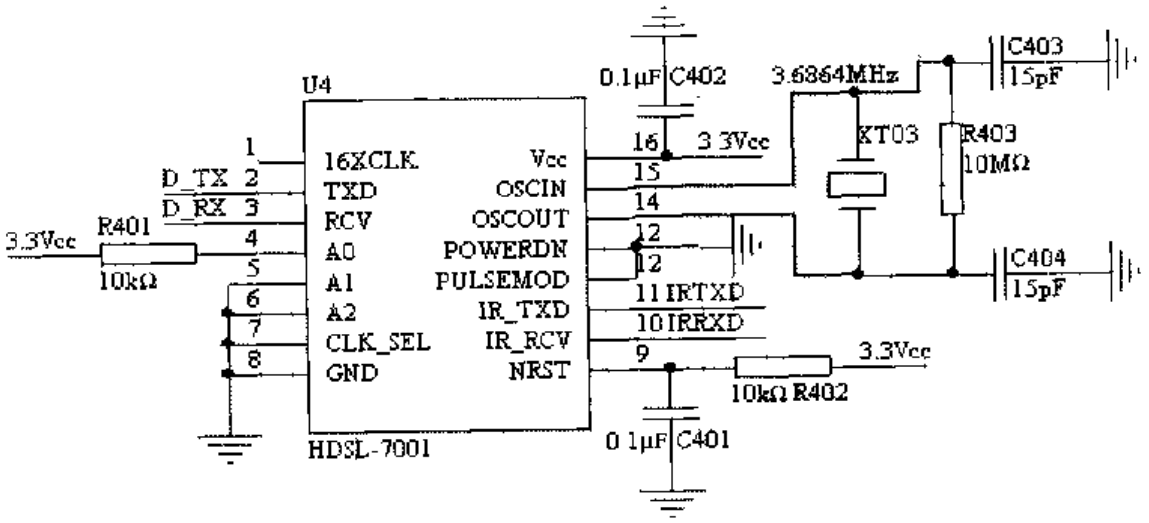


图 24-3　红外编解码电路

上面的电路主要完成数据按照红外传输的格式进行编码或者解码。编码后的数据需要采用红外线发送，这样就需要发送和接收电路，如图 24-4 所示为红外收发电路图。

由图 24-4 可以看出，红外收发电路主要通过 HDSL-3201 的 TXD 和 RXD 管脚与红外编解码电路进行数据传输。

单片机电路主要是利用串口 1（P3.6 和 P3.7 管脚）与 HDSL-7001 芯片接口，由于单片机电路相对比较简单，这里就不再给出具体的电路图了。

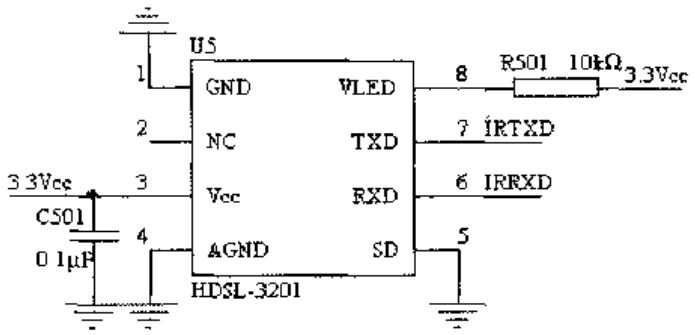


图 24-4　红外收发电路

24.2　软件设计

由前面的硬件设计可以知道，串口发送的数据经过红外编码器和红外发送器就可以实现红外数据的发送，同理也可以实现红外数据的接收，因此，整个软件设计包括初始化设置、串口中断服务程序和主处理程序。下面对各个部分程序分别进行详细介绍。

24.2.1　初始化设置

初始化程序主要包括时钟初始化、端口初始化和串口初始化。由于时钟初始化和第 3 章（LED 数码管显示电路的设计）的一致，这里不再进行介绍，读者可以参看前面的时钟初始化程序。下面为端口初始化和串口初始化的程序代码。

```
void Init_Port(void)
{
    //将所有的管脚在初始化的时候设置为输入方式
    P3DIR = 0;

    //将所有的管脚设置为一般 I/O 口
    P3SEL = 0;
    return;
}
```

上面的程序主要是初始化端口，由于串口 1 的管脚为 P3.6 和 P3.7，因此上面的程序初始化的是 P3 口。下面为串口 1 的初始化设置。

```
void Init_UART1(void)
{
    //将寄存器的内容清零
    U1CTL = 0X00;
    //数据位为 8bit
```

```
    U1CTL += CHAR;

    //将寄存器的内容清零
    U1TCTL = 0X00;
    //波特率发生器选择 SMCLK
    U1TCTL += SSEL1;

    //波特率为 57600 波特/秒
    UBR0_1 = 0X8B;
    UBR1_1 = 0X00;
    //调整寄存器
    UMCTL_1 = 0X00;

    //使能 UART1 的 TXD 和 RXD
    ME2 |= UTXE1 + URXE1;
    //使能 UART1 的 RX 中断
    IE2 |= URXIE1;
    //使能 UART1 的 TX 中断
    IE2 |= UTXIE1;

    //设置 P3.6 为 UART1 的 TXD
    P3SEL |= BIT6;
    //设置 P3.7 为 UART1 的 RXD
    P3SEL |= BIT7;

    //P3.6 为输出管脚
    P3DIR |= BIT6;
    return;
}
```

上面的程序主要设置串口 1 的参数，比如速率的设置等。另外上面的程序还将管脚 P3.6 和 P3.7 设置为串口 1 的管脚，而不是一般 I/O 管脚。

24.2.2　中断服务程序

串口通信采用中断机制，发送数据和接收数据都采用中断方式。当接收到数据时，设置一个标志来通知主程序有数据到来，当主程序有数据要发送的时候，设置一个中断标志进入中断发送数据。下面具体分析程序的代码。

```
interrupt [UART1RX_VECTOR] void UART1_RX_ISR(void)
{
    //接收来自串口的数据
    UART1_RX_BUF[nRX1_Len_temp] = RXBUF1;

    nRX1_Len_temp += 1;
```

```
        if(UART1_RX_BUF[nRX1_Len_temp - 1] == 13)
        {
            nRX1_Len = nRX1_Len_temp;
            nRev_UART1 = 1;
            nRX1_Len_temp = 0;
        }
    }
    interrupt [UART1TX_VECTOR] void UART1_TX_ISR(void)
    {
        if(nTX1_Len != 0)
        {
            // 表示缓冲区里的数据没有发送完
            nTX1_Flag = 0;

            TXBUF1 = UART1_TX_BUF[nSend_TX1];
            nSend_TX1 += 1;

            if(nSend_TX1 >= nTX1_Len)
            {
                nSend_TX1 = 0;
                nTX1_Len = 0;
                nTX1_Flag = 1;
            }
        }
    }
```

上面的程序为串口 1 的收发中断服务程序。收发程序都处于等待状态，一旦外面有数据到来，则触发接收，进入接收中断服务程序，接收中断程序接收数据。中断程序从“RXBUF1”寄存器里读取数据，将得到的数据放到“UART1_RX_BUF[]”全局缓冲区里，在接收到数据后设置一个标志“nRev_UART1”来通知主程序。如果有数据需要发送的时候，主程序设置一个发送标志，并且触发发送中断，进入发送中断服务程序。在发送中断程序里，从“UART1_TX_BUF[]”全局缓冲区里取出数据送给“TXBUF1”寄存器进行发送，发送完数据后，发送中断程序等待下一次中断的到来。

通过以上代码可以发现，采用中断服务机制有比较好的结构，只需要在中断服务程序里处理接收和发送数据，然后与主程序进行数据交互，这样比较容易实现多任务操作，很好地利用了单片机的资源。

24.2.3 主处理程序

主处理程序主要是根据业务处理的不同而不同，下面只是介绍一个简单的主处理程序。

```
void main(void)
{
    int nRes_UART1;
    int nRes = 0;
    char UART1_RX_Temp[60];

    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;

    // 关闭中断
    _DINT();

    // 初始化时钟
    Init_CLK();
    // 初始化端口
    Init_Port();
    // 初始化串口 1
    Init_UART1();

    // 打开中断
    _EINT();

    // 先发送数据
    UART1_TX_BUF[0] = 'O';
    UART1_TX_BUF[1] = 'K';
    UART1_TX_BUF[2] = 13;
    nTX1_Len = 3;
    // 设置中断标志，进入发送中断程序
    IFG2 |= UTXIFG1;

    // 进入处理循环，接收数据
    for(;;)
    {
        if(nRev_UART1 == 1)
        {
            nRev_UART1 = 0;
            for(i = 0;i < nRX1_Len;i++)
                UART1_RX_Temp[i] = UART1_RX_BUF[i];

            break;
        }
    }
}
```

24.3 实例总结

本章介绍了 MSP430 单片机实现的红外数据传输系统。在介绍红外编解码芯片和红外收发芯片的基础上，介绍了硬件设计，然后介绍了相应的程序设计。在进行程序设计时，由于红外的编解码处理和红外收发处理都是由硬件处理完成，因此软件主要就是串口通信的设计。串口通信采用中断服务程序的结构，这样便于进行多任务处理，充分利用了单片机的硬件资源。虽然本章介绍的主处理程序相对比较简单，但是读者可以在此基础上扩展命令处理函数，从而增强用户处理的功能。

第 25 章

MSP430 与 PC 机通信的设计与实现

在很多单片机应用领域里，下位机都需要与上位机进行通信。本章介绍 MSP430 单片机与 PC 机的通信设计，首先介绍接口设计，然后再给出具体的程序实现。

25.1 硬件设计

本系统的硬件接口电路相对比较简单，主要就是单片机与 SP3220 芯片的连接。为了便于理解接口，在介绍接口之前，先简要介绍一下 SP3220 芯片。

25.1.1 SP3220 芯片

SP3220 是一款低功耗的 RS232 驱动芯片，该芯片具有以下特性。

- 宽电压供电。供电电压为 3.3V～5.0V。
- 上传速率可以达到 235kbps。
- 低功耗的电流为 1μA。
- 符合增强性 ESD 规范。
- 处于低功耗状态下，仍然可以接收数据。

为了便于进行硬件电路的设计，下面给出 SP3220 芯片的管脚图，如图 25-1 所示。

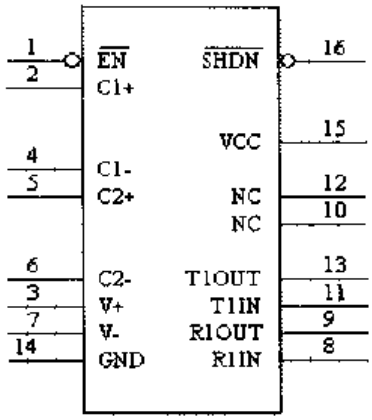


图 25-1　SP3220 芯片的管脚图

由图 25-1 可以看出，该芯片有 16 个管脚。下面对具体的管脚进行介绍。

- $\overline{EN}$：接收使能管脚。
- $\overline{SHDN}$：低功耗控制管脚。
- C1+、C1−：电压增倍的充电电容的正极和负极。
- C2+、C2−：倒置充电电容的正极和负极。
- V+、V−：由充电电容产生的 5.5V 的正极和负极。
- Vcc：电源管脚。
- GND：接地管脚。
- T1OUT：RS232 驱动的输出。
- T1IN：TTL/CMOS 的输入。
- R1OUT：TTL/CMOS 的输出。
- R1IN：RS232 的输入。

经过对 SP3220 芯片的介绍，对 RS232 有了基本的认识，下面介绍硬件电路的具体设计。

25.1.2　接口设计

硬件的接口电路相对比较简单，主要就是串口的设计。在 MSP430 系列单片机中，MSP430F12X、MSP430F13X、MSP430F14X、MSP430F44X 等系列单片机都有串口模块，因此 MSP430 单片机很容易通过片内的串口实现与 SP3220 芯片进行接口。如图 25-2 所示为接口电路图。

由图 25-2 可以看出，通过一个上拉电阻将 $\overline{SHDN}$ 管脚拉高，使该芯片一直处于工作状态。如果系统需要处于低功耗状态，也可以通过单片机来控制该管脚，工作的时候将该

管脚设置为低电平，需要处于低功耗的时候将该管脚设置为高电平，这样很容易实现控制。在管脚 C1+、C1-、C2+、C2−、V+和 V−处分别放置 0.1 μF 的电容实现充电作用，以满足相应的充电泵的要求。管脚 T1OUT、T1IN、R1OUT 和 R1IN 分别是 RS232 转换的输入输出脚，实现单片机的 TTL 电平与上位机的接口电平的转换。为了减小输入端受到的干扰，还需要在芯片的电源输入管脚处加一个 0.1μF 的电容来实现滤波。

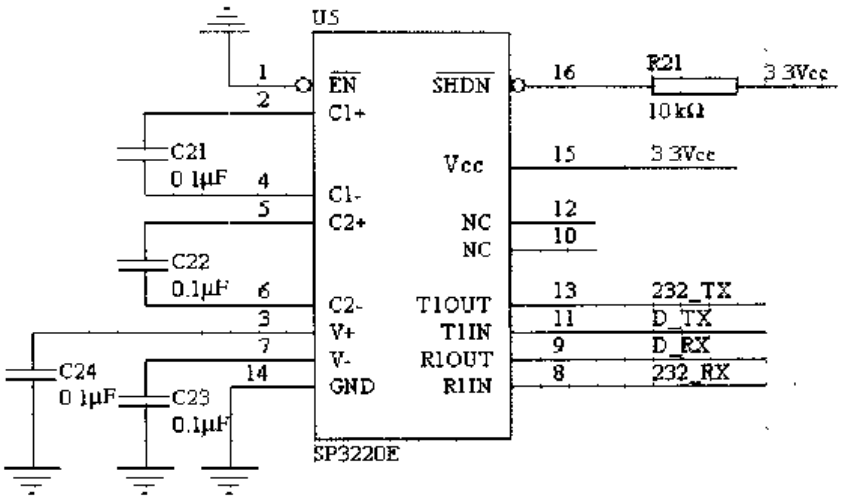


图 25-2　接口电路设计

单片机电路主要是利用串口 1（P3.6 和 P3.7 管脚）与 SP3220 芯片接口，由于单片机电路相对比较简单，这里就不再给出具体的电路图了。

25.2　软件设计

整个软件设计包括初始化设置、串口中断服务程序和主处理程序，下面对各个部分程序分别进行详细介绍。

25.2.1　初始化设置

初始化程序主要包括时钟初始化、端口初始化和串口初始化。关于时钟初始化请参看第 3 章，这里就不再进行介绍。下面为端口初始化和串口初始化的程序代码。

```
void Init_Port(void)
{
    //将所有的管脚在初始化的时候设置为输入方式
    P3DIR = 0;
```

```
    //将所有的管脚设置为一般 I/O 口
    P3SEL = 0;
    return;
}
```

上面的程序主要是初始化端口，由于串口 1 的管脚为 P3.6 和 P3.7，因此上面的程序初始化的是 P3 口。下面为串口 1 的初始化设置。

```
void Init_UART1(void)
{
    //将寄存器的内容清零
    U1CTL = 0X00;
    //数据位为 8bit
    U1CTL += CHAR;

    //将寄存器的内容清零
    U1TCTL = 0X00;
    //波特率发生器选择 SMCLK
    U1TCTL += SSEL1;

    //波特率为 57600 波特/秒
    UBR0_1 = 0X8B;
    UBR1_1 = 0X00;
    //调整寄存器
    UMCTL_1 = 0X00;

    //使能 UART1 的 TXD 和 RXD
    ME2 |= UTXE1 + URXE1;
    //使能 UART1 的 RX 中断
    IE2 |= URXIE1;
    //使能 UART1 的 TX 中断
    IE2 |= UTXIE1;

    //设置 P3.6 为 UART1 的 TXD
    P3SEL |= BIT6;
    //设置 P3.7 为 UART1 的 RXD
    P3SEL |= BIT7;

    //P3.6 为输出管脚
    P3DIR |= BIT6;
    return;
}
```

上面的程序主要设置串口 1 的参数，比如速率的设置等。另外上面的程序还将管脚 P3.6 和 P3.7 设置为串口 1 的管脚，而不是一般 I/O 管脚。

25.2.2　串口中断服务程序

串口通信采用中断机制，发送数据和接收数据都采用中断方式。当接收到数据时，设置一个标志来通知主程序有数据到来，当主程序有数据要发送的时候，设置一个中断标志进入中断发送数据。串口通信模块的程序流程图如图 25-3 所示。

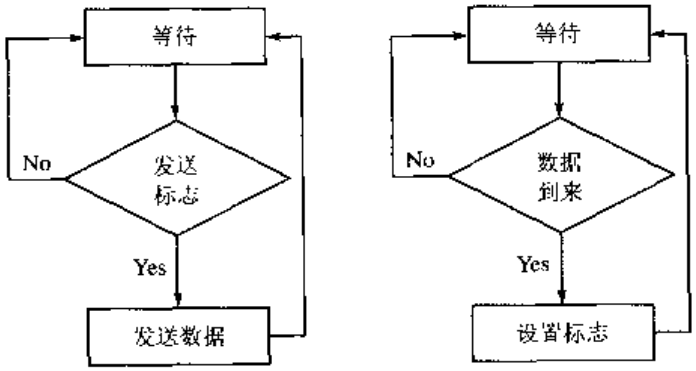


图 25-3　串口通信模块的程序流程图

对于发送中断，程序处于等待状态，如果检测到有发送的标志，则从缓冲区里取出数据发送；对于接收中断，等待数据的到来，如果有数据到来则设置标志通知主程序。下面具体分析程序的代码。

```
interrupt [UART1RX_VECTOR] void UART1_RX_ISR(void)
{
    //接收来自串口的数据
    UART1_RX_BUF[nRX1_Len_temp] = RXBUF1;

    nRX1_Len_temp += 1;

    if(UART1_RX_BUF[nRX1_Len_temp - 1] == 13)
    {
        nRX1_Len = nRX1_Len_temp;
        nRev_UART1 = 1;
        nRX1_Len_temp = 0;
    }
}
interrupt [UART1TX_VECTOR] void UART1_TX_ISR(void)
{
    if(nTX1_Len != 0)
    {
        // 表示缓冲区里的数据没有发送完
        nTX1_Flag = 0;

        TXBUF1 = UART1_TX_BUF[nSend_TX1];
```

```
            nSend_TX1 += 1;

            if(nSend_TX1 >= nTX1_Len)
            {
                nSend_TX1 = 0;
                nTX1_Len = 0;
                nTX1_Flag = 1;
            }
        }
    }
```

上面的程序为串口 1 的收发中断服务程序。收发程序都处于等待状态，一旦外面有数据到来，则触发接收，进入接收中断服务程序，接收中断程序接收数据。中断程序从“RXBUF1”寄存器里读取数据，将得到的数据放到“UART1_RX_BUF[]”全局缓冲区里，在接收到数据后设置一个标志“nRev_UART1”来通知主程序。如果有数据需要发送的时候，主程序设置一个发送标志，并且触发发送中断，进入发送中断服务程序。在发送中断程序里，从“UART1_TX_BUF[]”全局缓冲区里取出数据送给“TXBUF1”寄存器进行发送，发送完数据后，发送中断程序等待下一次中断的到来。

通过以上代码可以发现，采用中断服务机制有比较好的结构，只需要在中断服务程序里处理接收和发送数据，然后与主程序进行数据交互，这样比较容易实现多任务操作，很好地利用了单片机的资源。

25.2.3 主处理程序

主处理程序主要是处理接收到的数据和封装需要发送的数据。下面给出一个简单的主处理程序。

```
void main(void)
{
    int nRes_UART1;
    int nRes = 0;
    char UART1_RX_Temp[60];

    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;

    // 关闭中断
     _DINT();

    // 初始化时钟
    Init_CLK();
    // 初始化端口
    Init_Port();
```

```
// 初始化串口 1
Init_UART1();

// 打开中断
_EINT();

// 进入处理循环
for(;;)
{
      if(nRev_UART1 == 1)
      {
           nRev_UART1 = 0;
           // 将接收到的数据拷贝到临时缓冲区
           for(i = 0;i < nRX1_Len;i++)
                 UART1_RX_Temp[i] = UART1_RX_BUF[i];
           nRes = ProcessCMD(UART1_RX_Temp,nRX1_Len);
           switch(nRes)
           {
           case 1:
                 UART1_TX_BUF[0] = 'O';
                 UART1_TX_BUF[1] = 'K';
                 UART1_TX_BUF[2] = 13;
                 nTX1_Len = 3;
                 // 设置中断标志，进入发送中断程序
                 IFG2 |= UTXIFG1;
                 nRX1_Len = 0;
                 break;
           case 2:
                 for(n = 0;n < nRX1_Len;n++)
                       UART1_TX_BUF[n] = UART1_RX_Temp[n];
                 UART1_TX_BUF[nRX1_Len] = 'O';
                 UART1_TX_BUF[nRX1_Len+1] = 'K';
                 UART1_TX_BUF[nRX1_Len+2] = 13;
                 nTX1_Len = nRX1_Len + 3;
                 // 设置中断标志，进入发送中断程序
                 IFG2 |= UTXIFG1;
                 nRX1_Len = 0;
                 break;
           case -1:
                 UART1_TX_BUF[0] = 'E';
                 UART1_TX_BUF[1] = 'R';
                 UART1_TX_BUF[2] = 'R';
                 UART1_TX_BUF[3] = 'O';
                 UART1_TX_BUF[4] = 'R';
                 UART1_TX_BUF[5] = 13;
                 nTX1_Len = 6;
                 // 设置中断标志，进入发送中断程序
                 IFG2 |= UTXIFG1;
                 nRX1_Len = 0;
```

```
                    break;
                }
            }
        }
    }
```

在上面的程序中，主要根据“ProcessCMD(UART1_RX_Temp,nRX1_Len)”对得到的结果进行相应的处理，向 PC 机发送响应数据，如果接收到的数据有错，则向 PC 机返回“ERROR”。数据封装完后，设置“nTX1_Len”的长度，并通过“IFG2 |= UTXIFG1;”触发发送中断，从而使中断处理程序进行数据发送。

在上面的程序中，“ProcessCMD(UART1_RX_Temp,nRX1_Len)”主要处理接收到的数据，根据接收到的数据返回相应的代码，以便主程序进行处理。该函数的具体代码如下：

```
int ProcessCMD(char pBuf[],int nLen)
{
    int nTemp = -1;
    int i;

    if(nLen <= 2) return -1;

    if (nLen == 5)
    {
        if((pBuf[0] == 'A') && (pBuf[1] == 'T')
            && (pBuf[2] == 'E') && (pBuf[3] == '0'))
            nTemp = 1;
        if((pBuf[0] == 'A') && (pBuf[1] == 'T')
            && (pBuf[2] == 'E') && (pBuf[3] == '1'))
            nTemp = 2;
    }

    return nTemp;
}
```

上面的程序只是简单地处理了两个命令。在实际应用中，可以进一步扩展上面的函数，处理更多的命令，这主要需要结合具体的应用系统来确定。

25.3 实例总结

本章介绍了 MSP430 单片机与 PC 机的通信设计。首先介绍了接口电路的设计，然后介绍了相应的程序。在进行程序设计的时候，采用中断服务程序的结构，这样便于进行多任务处理，充分利用了单片机的硬件资源。虽然本章介绍的主处理程序相对比较简单，但是读者可以在此基础上扩展命令处理函数，从而增强用户处理的功能。

第 26 章

基于 MSP430 单片机实现的无线 MODEM

目前，无线通信广泛应用于电台系统、导航系统、数据采集与工业控制和家庭网络系统等。在无线通信系统中，调制解调是一个非常重要的组成部分。本章介绍采用 MSP430 单片机作为 MCU 的无线 MODEM，首先介绍系统的硬件实现，然后介绍系统的软件实现。

26.1 硬件设计

系统硬件主要是 MSP430 单片机的串口设计，以及与无线 MODEM 芯片的接口。系统通过串口与数据源（如语音、数据等）进行通信，将数据调制后进行发送。同样，系统接收到无线数据后，经过解调后通过串口将解调的数据送给数据源。如图 26-1 所示为系统的构成。

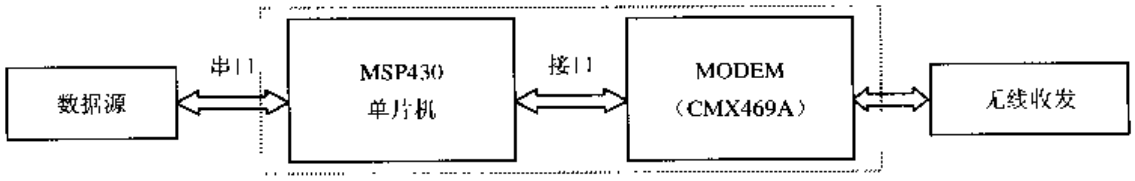


图 26-1 系统的构成

在图 26-1 中，数据源的处理是信源的处理，本身涉及的处理可能比较复杂，比如压缩处理，本章对数据源部分不进行介绍，只是介绍通过串口进行数据通信。同样，无线收发

在本章中也不进行介绍。在介绍具体的硬件电路之前，先介绍一下无线 MODEM 芯片，本章介绍的无线 MODEM 芯片为 CMX469A 芯片。

26.1.1 CMX469A 芯片

CMX469A 芯片是英国 CML 公司推出的全双工无线 MODEM 芯片，它采用 FFSK/MSK 调制方式。CMX469A 芯片主要有以下特点：

- 有很宽的工作电压，电压范围为 2.7V～5.5V。
- 具有单独的发送和接收使能控制。
- 通过管脚可以选择 1200bps、2400bps 和 4800bps 三种速率。
- 通过管脚可以选择时钟频率为 1.008MHz 或者 4.032MHz。
- 具有载波检测功能。
- 具有接收时钟故障恢复功能。
- 低电压、低功耗（电源为 3V 时，典型工作电流为 2mA）。
- 抗干扰性能优良，在信号条件比较差的情况下具有优良的灵敏度，同时可通过外部电容设置载波检测周期，以使器件在高噪声环境下具有更完善的性能。

为了便于后面的硬件设计，下面给出 CMX469A 芯片的管脚图。如图 26-2 所示。

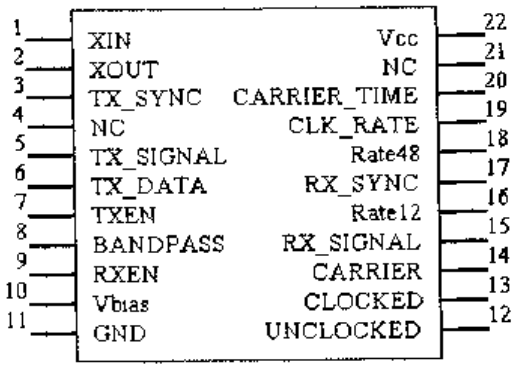


图 26-2 CMX469A 芯片的管脚图

由图 26-2 可以看出，CMX469A 芯片共有 22 个管脚，下面对具体的管脚进行介绍。

- XIN、XOUT：晶体管脚。
- TX_SYNC：发送同步时钟输出。
- TX_SIGNAL：FFSK/MSK 信号输出。
- TX_DATA：发送数据串行输入端。
- TXEN：发送使能。

- BANDPASS：RX 带通滤波器输出。
- RXEN：接收使能。
- Vbias：偏置电压输出。
- GND：接地管脚。
- UNCLOCKED：接收异步数据输出。
- CLOCKED：接收同步数据输出。
- CARRIER：载波检测输出，当接收到 FFSK/MSK 信号时，该管脚输出“1”。
- RX_SIGNAL：FFSK/MSK 信号输入。
- Rate12：1200bps/2400bps 速率选择。
- RX_SYNC：接收同步时钟输出。
- Rate48：4800bps 速率选择。
- CARRIER_TIME：载波检测的时间设置管脚。
- Vcc：电源管脚。

经过对 CMX469A 芯片的简单介绍，对该款芯片有了基本的认识，下面介绍硬件电路的具体设计。

26.1.2　CMX469A 芯片接口设计

CMX469A 芯片需要与单片机进行连接以实现数据通信。单片机与 CMX469A 芯片之间的通信采用同步方式，发送和接收都有各自的时钟线和数据线。此外，单片机还需要分别控制 CMX469A 芯片的发送使能管脚和接收使能管脚。由于 CMX469A 芯片有载波检测管脚，因此可以将该管脚与单片机的一般 I/O 口进行连接，这样可以判断是否有数据到来。由于 MSP430 单片机的 P1 口有中断功能，因此可以使 CMX469A 芯片的接收时钟管脚和发送时钟管脚与 MSP430 单片机的中断管脚进行连接，以便后面的程序设计。此外，通过单片机的 P2.0、P2.1 和 P2.2 管脚分别和 CMX469A 芯片的时钟速率选择管脚和时钟选择管脚进行连接，以便可以采用软件进行速率选择。如图 26-3 所示为具体的接口电路图。

由图 26-3 可以看出，该接口电路比较简单，只需要很少的外围器件。

26.1.3　串口设计

由于本系统涉及到与数据源之间进行通信，因此需要考虑通信接口设计。本系统采用的通信方式为串口通信方式。由于 MSP430 单片机一般都有硬件串口，因此实现起来非常容易。如图 26-4 所示为具体的串口电路图。

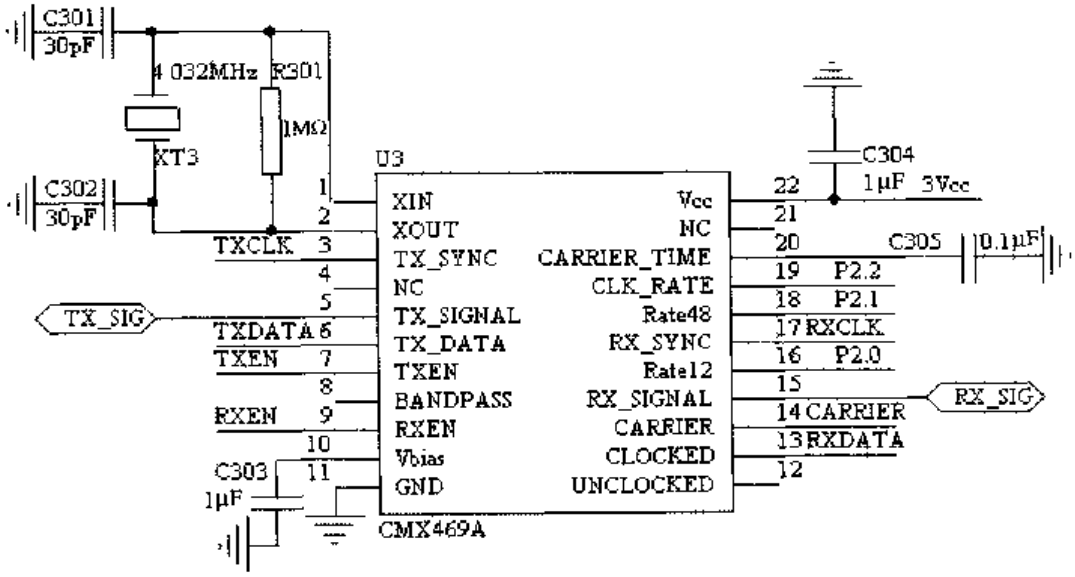


图 26-3　CMX469A 芯片接口电路

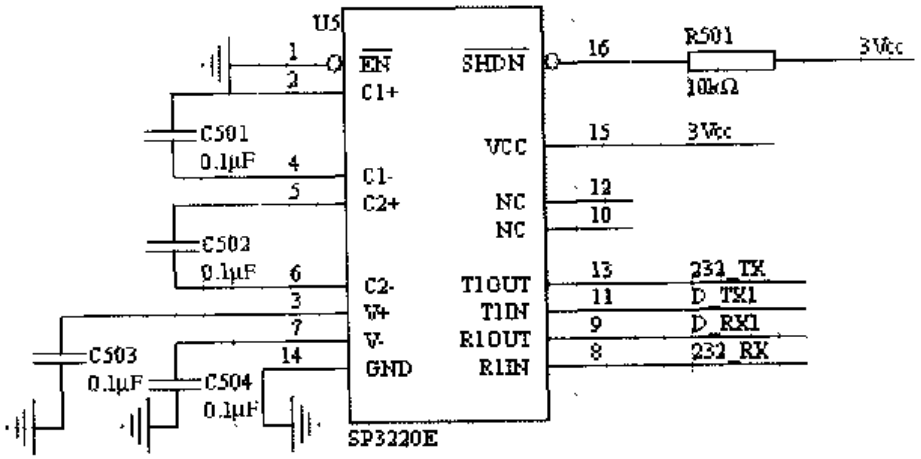


图 26-4　串口电路

由于串口电路比较简单，并且读者对串口电路也比较熟悉，在此不进行详细介绍。经过上面对硬件的介绍，对系统的工作有了一定的了解，下面介绍系统的软件设计。

26.2　软件设计

整个软件系统主要包括 3 个部分，即初始化及管脚模拟部分、CMX469A 操作部分和 UART 串口通信部分。下面对这 3 个部分分别进行介绍。

26.2.1　初始化及管脚模拟

该部分程序主要是对管脚进行初始化，并初始化串口。此外，对某些管脚产生相应的高电平或者低电平。下面为具体的程序代码。

```
void Port_Init(void)
{
      P1DIR = 0;
      //设置 P1.1 为输出管脚
      P1DIR |= BIT1;
      //设置 P1.2 为输出管脚
      P1DIR |= BIT2;
      //设置 P1.5 为输出管脚
      P1DIR |= BIT5;
      //设置 P2.0 为输出管脚
      P2DIR |= BIT0;
      //设置 P2.1 为输出管脚
      P2DIR |= BIT1;
      //设置 P2.2 为输出管脚
      P2DIR |= BIT2;
      //将中断寄存器清零
      P1IE = 0;
      P1IES = 0;
      P1IFG = 0;
      //管脚 P1.0 使能中断
      P1IE |= BIT0;
      //对应的管脚由高到低电平跳变使相应的标志置位
      P1IES |= BIT0;
      //管脚 P1.3 使能中断
      P1IE |= BIT3;
      //对应的管脚由高到低电平跳变使相应的标志置位
      P1IES |= BIT3;

      //设置速率为 2400bps
      P2OUT |= CLKRATE;
      P2OUT &= ~(RATE48);
      P2OUT &= ~(RATE12);

      return;
}

void Init_UART1(void)
{
      //将寄存器的内容清零
      U1CTL = 0X00;
```

```
    //数据位为 8bit
    U1CTL += CHAR;
    //将寄存器的内容清零
    U1TCTL = 0X00;
    //波特率发生器选择 SMCLK
    U1TCTL += SSEL1;
    //波特率为 57600 波特/秒
    UBR0_1 = 0X8B;
    UBR1_1 = 0X00;
    //调整寄存器
    UMCTL_1 = 0X00;
    //使能 UART1 的 TXD 和 RXD
    ME2 |= UTXE1 + URXE1;
    //使能 UART1 的 RX 中断
    IE2 |= URXIE1;
    //使能 UART1 的 TX 中断
    IE2 |= UTXIE1;
    //设置 P3.6 为 UART1 的 TXD
    P3SEL |= BIT6;
    //设置 P3.7 为 UART1 的 RXD
    P3SEL |= BIT7;
    //P3.6 为输出管脚
    P3DIR |= BIT6;
    return;
}
void TXDATA_HI(void)
{
    P1OUT |= BIT1;
    return;
}
void TXDATA_LO(void)
{
    P1OUT &= ~(BIT1);
    return;
}
void TXEN_HI(void)
{
    P1OUT |= BIT2;
    return;
}
void TXEN_LO(void)
{
    P1OUT &= ~(BIT2);
    return;
}
void RXEN_HI(void)
{
    P1OUT |= BIT5;
```

```
    return;
}
void RXEN_LO(void)
{
    P1OUT &= ~(BIT5);
    return;
}
```

26.2.2　CMX469A 操作

CMX469A 操作程序主要是实现数据的发送与接收。由于数据的收发是采用同步方式进行的，即发送和接收分别由发送时钟和接收时钟控制，因此采用 P1 口的中断功能来实现数据的收发。下面为具体的程序代码。

```
interrupt [PORT1_VECTOR] void MODEM_TXRX_ISR(void)
{
    int i;
    char chrTemp;
    char chrTmp;
    char chrRXTmp;
    //处理发送
    if(P1IFG & BIT0)
    {
        //清除中断标志位
        P1IFG &= ~(BIT0);
        for(i = 3;i > 0;i--) ;
        //是否有数据
        if(nCM_TX_Len != 0)
        {
            //表示缓冲区里的数据没有发送完
            nCM_Flag = 0;
            chrTemp = CM_TX_BUF[nCM_Send];
            chrTmp = (chrTemp >> (7 - nCM_Count)) & 0x01;
            if(chrTmp == 1)
            {
                //发送“1”
                TXDATA_HI();
            }
            else
            {
                //发送“0”
                TXDATA_LO();
            }
            //位计数器
            nCM_Count += 1;
            //是否发送完一个字节
```

```
            if(nCM_Count >= 8)
            {
                nCM_Count = 0;
                nCM_Send += 1;
            }
        //是否发送完数据
        if(nCM_Send >= nCM_TX_Len)
        {
                //清除标志、长度等
                nCM_Send = 0;
                nCM_TX_Len = 0;
                nCM_Flag = 1;
        }
      }
    }
    //处理接收
    if(P1IFG & BIT3)
    {
        //清除中断标志位
        P1IFG &= ~(BIT3);
        for(i = 3;i > 0;i--) ;
        //接收来自串口的数据
        nCM_RXCount += 1;
        //设置为输入方向
        P1DIR &= ~(BIT4);
        chrRXTmp = (char)((P1IN & BIT4) >> 4);
        if(chrRXTmp == 1)
        {
            CM_REV_CHAR  = (char)(1 << (7 - nCM_RXCount));
        }
        if(nCM_RXCount >= 8)
        {
            CM_RX_BUF[nCM_RX_LenTemp] = CM_REV_CHAR;
            CM_REV_CHAR = 0;
            nCM_RX_LenTemp += 1;
            nCM_RX_Len = nCM_RX_LenTemp;
            nCM_Rev = 1;
            nCM_RXCount = 0;
        }
    }
}
```

在上面的程序中，分别根据不同的中断标志位对数据的发送和数据的接收进行处理。对于发送而言，当将发送缓冲区里所有的数据发送完毕后，设置全局的标志“nCM_Flag”为“1”来通知其他程序模块数据发送完毕。对于接收而言，只要接收到一个字节数据后，就设置“nCM_Rev”为“1”来通知其他程序模块已经接收到数据。数据的发送和接收都

是按位进行的。此外，上面的程序需要和其他模块结合使用才能实现数据的完整收发功能。在上面的程序中，主要通过全局的缓冲区和全局的标志变量与其他模块的程序进行数据交互。

26.2.3　UART 串口通信

串口通信采用中断机制，发送数据和接收数据都采用中断方式。当接收到数据时，设置一个标志来通知主程序有数据到来；当主程序有数据要发送的时候，设置一个中断标志进入中断发送数据。下面为具体的程序代码。

```
interrupt [UART1RX_VECTOR] void UART1_RX_ISR(void)
{
    //接收来自串口的数据
    UART1_RX_BUF[nRX1_Len_temp] = RXBUF1;
    nRX1_Len_temp += 1;

    if(UART1_RX_BUF[nRX1_Len_temp - 1] == 13)
    {
        nRX1_Len = nRX1_Len_temp;
        nRev_UART1 = 1;
        nRX1_Len_temp = 0;
    }
}
interrupt [UART1TX_VECTOR] void UART1_TX_ISR(void)
{
    if(nTX1_Len != 0)
    {
        //表示缓冲区里的数据没有发送完
        nTX1_Flag = 0;
        TXBUF1 = UART1_TX_BUF[nSend_TX1];
        nSend_TX1 += 1;

        if(nSend_TX1 >= nTX1_Len)
        {
            nSend_TX1 = 0;
            nTX1_Len = 0;
            nTX1_Flag = 1;
        }
    }
}
```

上面的程序为串口 1 的收发中断服务程序。收发程序都处于等待状态，一旦外面有数据到来，则触发接收，进入接收中断服务程序，接收中断程序接收数据。中断程序从

“RXBUF1”寄存器里读取数据，将得到的数据放到“UART1_RX_BUF[]”全局缓冲区里，在接收到数据后设置一个标志“nRev_UART1”来通知主程序。如果有数据需要发送的时候，主程序设置一个发送标志，并且触发发送中断，进入发送中断服务程序。在发送中断程序里，从“UART1_TX_BUF[]”全局缓冲区里取出数据送给“TXBUF1”寄存器进行发送，发送完数据后，发送中断程序等待下一次中断的到来。

以上给出了软件的各个部分的具体程序代码，下面给出一个简单的主处理程序代码以供参考。

```
void main(void)
{
    char n;
    int i;
    char chrTemp;
    char chrRXLen;
    char chrRX_BUF_M[100];
    char chrRX_BUF[100];

    WDTCTL = WDTPW + WDTHOLD;    //关闭看门狗
    _DINT();                     //关闭中断

    Init_CLK();
    Init_Port();
    Init_UART1();

    _EINT();                     //打开中断

    nCM_Count = 0;
    nCM_RXCount = 0;
    CM_RFV_CHAR = 0;
    nCM_Send = 0;
    nCM_TX_Len = 0;
    nCM_Rev = 0;
    nCM_RX_Len = 0;
    nCM_RX_LenTemp = 0;
    nCM_Flag = 0;
    chrTemp = 0;
    chrRXLen = 0;
    for(i = 0;i < 120;i++)
    {
        CM_TX_BUF[i] = 0;
    }
    for(i = 0;i < 100;i++)
    {
        CM_RX_BUF[i] = 0;
    }
```

```
for(;;)
{
    //处理来自 MODEM 解调的数据
    if(nCM_Rev == 1)
    {
        nCM_Rev = 0;
        if(nCM_RX_LenTemp == 0)
        {
            //第一个数据
            //检查载波
            chrTemp = (char)((P1IN & BIT6) >> 6);
            if(charTemp != 1)
            {
                //载波无效
                nCM_RX_LenTemp = 0;
                nCM_RX_Len = 0;
            }
        }
        else if(nCM_RX_LenTemp == 2)
        {
            if((CM_RX_BUF[0] != 0xaa) || (CM_RX_BUF[1] != 0xaa))
            {
                //同步错误
                nCM_RX_LenTemp = 0;
                nCM_RX_Len = 0;
            }
        }
        else if(nCM_RX_LenTemp >= 3)
        {
            //取出数据包长度字节
            chrRXLen = CM_RX_BUF[2];
            if(chrRXLen == nCM_RX_Len)
            {
                for(i = 0;i < nCM_RX_Len;i++)
                {
                    chrRX_BUF_M[i] = CM_RX_BUF[i];
                }

                //接收完一帧数据
                //将数据发送给串口
                UART1_TX_BUF[0] = 0xaa;
                UART1_TX_BUF[1] = 0xaa;
                //表示数据
                UART1_TX_BUF[2] = 0x01;
                for(i = 2;i < nCM_RX_Len;i++)
                {
                    UART1_TX_BUF[i + 1] = chrRX_BUF_M[i];
                }
```

```
                UART1_TX_BUF[nCM_RX_Len] = 13;
                nTX1_Len = nCM_RX_Len + 2;
                //设置中断标志，进入中断发送程序
                IFG2 |= UTXIFG1;
                chrRXLen = 0;
                nCM_RX_Len = 0;
                nCM_RX_LenTemp = 0;
            }
        }
    }//处理来自 MODEM 解调的数据

    //处理来自串口的数据
    if(nRev_UART1 == 1)
    {
        nRev_UART1 = 0;
        CM_TX_BUF[0] = 0xaa;
        CM_TX_BUF[1] = 0xaa;
        for(i = 0;i < nRX1_Len - 1;i++)
        {
            CM_TX_BUF[i + 2] = UART1_RX_BUF[i];
        }
        nCM_TX_Len = nRX1_Len + 1;
        nRX1_Len = 0;
        while(1)
        {
            //等待发送完毕
            if(nCM_Flag == 1)
            {
                break;
            }
            //发送使能
            TXEN_H1();
        }
        while(1)
        {
            //等待发送完毕
            if(nCM_Flag == 1)
            {
                break;
            }
            //发送不使能
            TXEN_L0();
        }
        //往串口发送响应数据
        UART1_TX_BUF[0] = 0xaa;
        UART1_TX_BUF[1] = 0xaa;
        //表示 MODEM 数据发送完
        UART1_TX_BUF[2] = 0x02;
```

```
                UART1_TX_BUF[3] = 13;
                nTX1_Len = 4;
                //设置中断标志，进入中断发送程序
                IFG2 |= UTXIFG1;
            }//处理来自串口的数据
        }
    }
```

上面的程序只是简单地演示了从串口接收数据，并将数据发送给 MODEM；从 MODEM 接收数据，并将数据发送给串口。由于主处理程序根据系统功能不同而不同，所以需要读者根据自己的要求来进行重写。

26.3　实例总结

本章简单介绍了无线 MODEM 芯片 CMX469A 的硬件设计和软件设计。由于本系统的速率为 1200bps/2400bps/4800bps，因此在短波电台或者超短波电台等领域中有很大的应用前景。虽然本系统介绍的例子比较简单，但是读者完全可以在此基础上进行改进，以便满足自己系统的要求。在实际应用中，还需要考虑具体的通信协议，这样才能保证通信的有效性。此外，由于无线通信相对干扰比较大，所以在实际应用中还需要考虑检错、纠错功能。

第 27 章

基于 MSP430 实现的楼宇对讲系统

随着智能小区的建设，楼宇对讲系统成为其不可缺少的组成部分。本章介绍采用 MSP430 单片机实现的楼宇对讲系统，首先介绍它的硬件设计，然后介绍它的软件设计。

27.1 硬件设计

整个系统由门口主机、楼层译码器和室内分机组成。其中门口主机接收用户输入数据、进行呼叫处理及控制电控锁。系统由统一的电源供电。整个系统采用总线方式布线，门口主机到楼层译码器采用 3 总线方式，楼层译码器到室内分机采用 2 总线方式，并且该 2 总线方式为无极性的 2 总线方式，这样便于接线，即使用户线短路也不会影响整个系统。如图 27-1 所示为系统构成图。

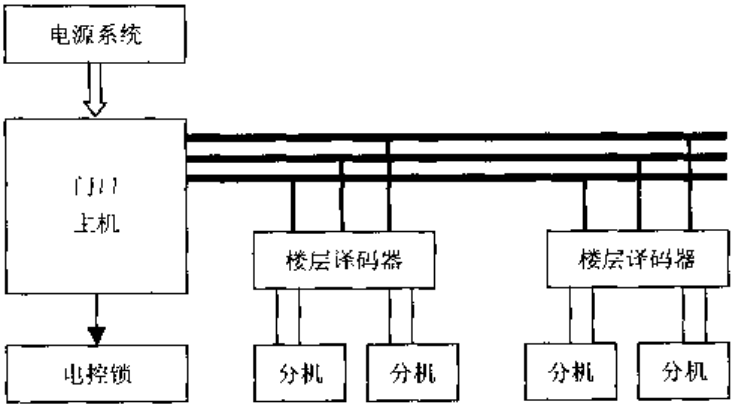


图 27-1 系统构成图

由图 27-1 可以看出，主机发出编码信息（比如房间号码），楼层译码器根据接收到的数据进行译码，如果是本层的某个房间，则将分机的线路接通，主机检测到线路状态后就可以发送振铃信号，分机应答后可以进行通话，并给出开锁信号，这样主机在收到分机给出的开锁信号后打开电控锁。整个硬件系统包括 3 个部分，即主机、楼层译码器和分机。由于分机的设计相对简单，这里就不进行介绍了。下面对主机和楼层译码器进行简单介绍。

27.1.1　主机设计

主机主要包括键盘输入电路、存储器电路、逻辑控制电路、比较器电路、驱动电路和音频处理电路。键盘输入电路主要用于接收用户输入的数据，比如密码设置、振铃时间设置、房间号码设置及管理员设置等。键盘采用一般的矩阵键盘。存储器电路主要用于存储系统的信息，比如房间模式、密码等信息。逻辑控制电路主要是产生某些逻辑控制信号，比如开锁信号、对话信号、振铃信号等。由于单片机产生的控制信号一般是 3V，不能满足控制器件的电压要求，因此控制信号需要通过驱动芯片来进行处理。比较器电路主要用于判断分机的状态，比如摘机、挂机等。音频电路主要实现对讲功能。如图 27-2 所示为主机的组成框图。

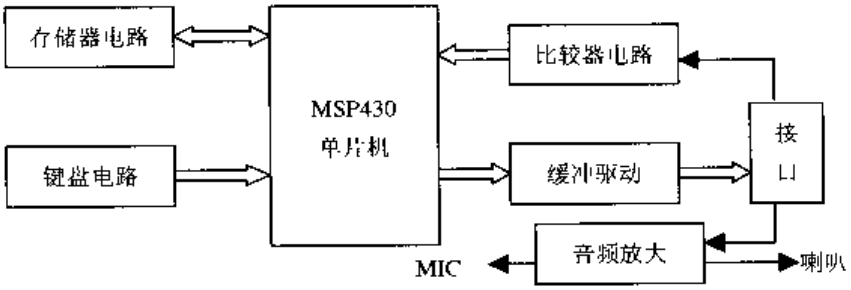


图 27-2　主机的组成框图

27.1.2　楼层译码器设计

楼层译码器用来接收主机发出的编码数据，并对接收到的编码数据进行译码处理，如果是本楼层的分机，则接通被呼叫的分机。楼层译码器选用 HT12D 芯片来实现。如图 27-3 所示为译码器的电路图。

由图 27-3 可以看出，主机发出的编码数据经过 HT12D 芯片译码，如果编码数据中的地址部分正好和该译码器的地址（通过 JP101 跳线来设置）相同，那么 HT12D 芯片就在 D8、D9、D10 或者 D11 中的某一个管脚输出高电平，可以通过这个高电平来接通分机（可

以使用三极管来实现）。由于 HT12D 芯片有锁存功能，因此它只有在下一个编码数据到来后才会改变输出管脚的状态。

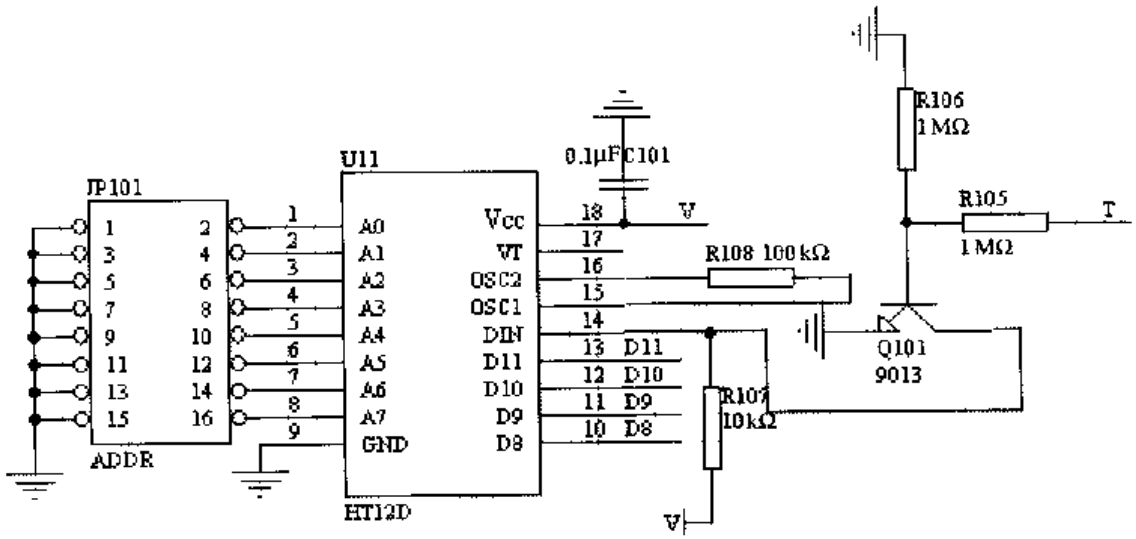


图 27-3　译码器电路

27.2　软件设计

整个系统软件比较复杂，主要包括显示模块、存储器操作模块（I^2C 模块）、键盘输入模块、拨号处理模块、主处理模块等。如图 27-4 所示为整个软件系统的模块构成图。

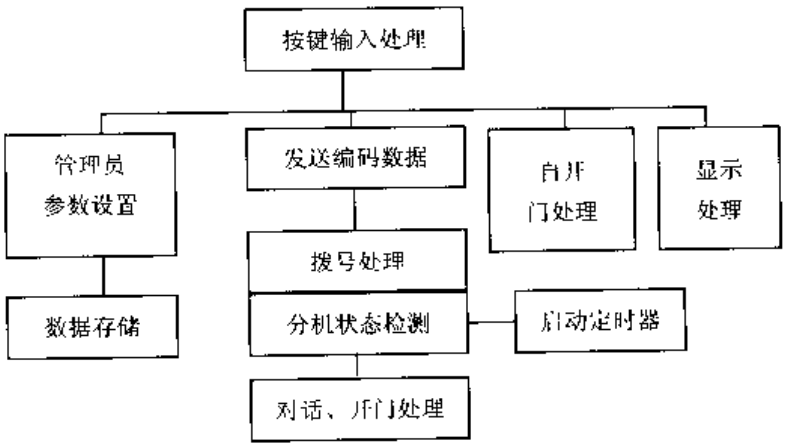


图 27-4　系统软件构成图

由于显示模块、存储器操作模块（I^2C 模块）、键盘输入模块分别在第 3、11、2 章中作了介绍，本章就不作介绍了。管理员参数和自开门处理实现起来比较容易，这里限于篇幅也不作介绍了。下面主要对发送编码数据处理和拨号处理进行介绍。

27.2.1　发送编码数据处理

发送的编码数据总共有 12 位，其中前面的 8 位为地址数据，后面的 4 位为数据信息。数据在发送时是一位一位发送的。数据发送时，首先需要发送起始信息，然后再将编码数据一位一位发送。由于起始信息、“1”和“0”的高低电平持续时间的不同，因此需要分别进行处理，下面为具体的程序。

```
void Send_Bit1()
{
    char i;
    P4OUT &= ~(BIT4);
    for(i = 55;i > 0;i- ) ;
    for(i - 55;i > 0;i--) ;
    P4OUT |= BIT4;
    for(i = 55;i > 0;i--) ;
}
void Send_Bit0()
{
    char i;
    P4OUT &= ~(BIT4);
    for(i - 55;i > 0;i--) ;
    P4OUT |= BIT4;
    for(i = 55;i > 0;i--) ;
    for(i - 55;i > 0;i--) ;
}
void Send_Pre()
{
    char i,j;

    P4OUT &= ~(BIT4);
    for(j = 0;j < 36;j++)
        for(i = 0;i < 55;i++)
            ;
    P4OUT |= BIT4;
    for(i - 55; i > 0;i--) ;
}
```

上面的程序给出了起始信息位、电平“1”和电平“0”的发送程序，在上面的程序中，必须严格按照 HT12D 芯片的时序要求来确定高低电平的时间。

在上面的程序基础上，下面给出发送编码数据的程序。

```
void Send_data(unsigned int numCode)  //要发送的数据是 numCode
{
    int CodeBit;
```

```
    int SendBit;
    int nHigh;
    int nLow;
    int n;
    int i;

    nLow = numCode & 0x0f;
    nHigh = numCode >> 4;

    Send_Pre();
    for(i = 0;i < 8;i++)
    {
        SendBit = nHigh & 0x01;
        if(SendBit == 1)
            Send_Bit1();
        else
            Send_Bit0();
        nHigh >>= 1;
    }
    for(i = 0;i < 4;i++)
    {
        SendBit = nLow & 0x01;
        if(SendBit == 1)
            Send_Bit1();
        else
            Send_Bit0();
        nLow >>= 1;
    }
}
```

在上面的程序中，首先发送 8 位的地址数据，然后再发送 4 位的某楼层房间号数据。由于后面的 4 位数据是表示某一个分机，因此需要进行编码处理，即后面的 4 位数据中只能有 1 个“1”，比如“0100”表示房间号为“3”（类似于 113 表示第 11 层的 3 号房间）。

27.2.2 拨号处理

在主机发送完编码数据后就进行拨号处理。拨号处理主要根据线路的状态来判断分机的状态，并根据分机的不同状态进行相应处理。线路状态的检测是通过比较器输入模块进行处理的。下面为拨号处理的具体程序。

```
void DialAndProcess(void)
{
    int i;
    int nBit5;
    int nBit6;
```

```
//判断选通房间分机是否存在
Delay();
Delay();
Delay();
nBit5 = (P1IN & BIT5) >> 5;
nBit6 = (P1IN & BIT6) >> 6;
if((nBit6 == 1) && (nBit5 == 0))
{
    //分机存在
    //清除锁存信号
    STCLK_Lo();
    //输出 1
    DataOut(nLed[1]);
    DataOut(nLed[nValue]);
    //给锁存信号，显示数据
    STCLK_Hi();

    //判断分机是否短路
    nBit5 = (P1IN & BIT5) >> 5;
    nBit6 = (P1IN & BIT6) >> 6;
    if((nBit6 == 1) && (nBit5 == 1))
     {
         //分机短路
         //提示音三次
         Ring();
         P4OUT &= ~(BIT2);
         Delay_1s();
         Ring();
         P4OUT &= ~(BIT2);
         Delay_1s();
         Ring();
         Delay_1s();
         P4OUT &= ~(BIT2);
     }//分机短路
     else
     {
         //分机不短路
         //发出振铃信号
         //初始化摘机标志，为 0 代表没有摘机
         ZhaijiFlag = 0;
         //定时器中断允许
         TBCTL |= TBCLR;
         TBCCTL0 |= CCIE;
         //在振铃时间 30s 内
         while(RingFlag)
         {
             //发出振铃信号
             Ring();
```

```
        Delay_1s();
        P4OUT &= ~(BIT2);
        //判断分机是否摘机
        nBit5 = (P1IN & BIT5) >> 5;
        nBit6 = (P1IN & BIT6) >> 6;
        if((nBit6 == 0) && (nBit5 == 0))
        {
            //分机摘机
            //摘机标志为 1，代表摘机
            ZhaijiFlag = 1;
        }//分机摘机
        if(ZhaijiFlag == 1) break;
    }//振铃时间内

    //定时器中断禁止
    TBCCTL0 &= ~(CCIE);

    //假如分机摘机
    if(ZhaijiFlag == 1)
    {
        //分机摘机
        //对话
        //定时器中断允许
        TBCTL |= TBCLR;
        TBCCTL0 |= CCIE;
        P4OUT &= ~(BIT2);
        DialogFlag = 1;
        //在对话时间 60s 内
        while( DialogFlag)
        {
            //打开对讲
            P4OUT |= BIT0;
            P4OUT |= BIT1;
            P4OUT &= ~(BIT2);
            P4OUT &= ~(BIT4);

            //判断分机是否挂机
            nBit5 = (P1IN & BIT5) >> 5;
            nBit6 = (P1IN & BIT6) >> 6;
            if((nBit6 == 0) && (nBit5 == 0))
            {
                //分机挂机
                //关闭对讲
                P4OUT &= ~(BIT0);
                P4OUT &= ~(BIT3);
            }//分机挂机，返回
            else
            {
```

```
                        //分机没有挂机
                        //判断是否开锁
                        nBit5 = (P1IN & BIT5) >> 5;
                        nBit6 = (P1IN & BIT6) >> 6;
                        if((nBit6 == 1) && (nBit5 == 1))
                        {
                            //分机开锁
                            for(i = 0;i < 10;i++)
                            {
                                //开电锁命令
                                P4OUT &= ~(BIT1);
                            }
                            break;
                        }//分机开锁
                        else//不开锁
                        {
                            ;
                        }//不开锁
                    }//分机没有挂机
                }//对话时间内

                //定时器中断禁止
                TBCCTL0 &= ~(CC1E);
            }//分机摘机
        }//分机不短路
    }//分机存在
    else //分机不存在
    {
        //清除锁存信号
        STCLK_Lo();
        //输出 0
        DataOut(nLed[0]);
        DataOut(nLed[nValue]);
        //给锁存信号，显示数据
        STCLK_Hi();

        //提示音两次
        Ring();
        Delay_1s();
        Ring();
    }

    nidle = 1;    //重新开始扫描键盘
}
```

在上面的拨号处理中，根据比较器处理模块的输出来确定线路的状态，再根据线路状态分别判断分机是否存在、分机是否摘机。当分机存在时，发出振铃信号；当分机摘机后，

则进行对话处理，并且判断对话过程中，分机是否发出开锁信号，如果主机检测到有开锁信号则进行开锁处理。在上面的程序中，可能有线路状态相同的情况，但是由于这些情况处于不同的阶段，因此能够正确判断分机的状态。

27.3 实例总结

本章介绍了采用 MSP430 单片机实现的楼宇对讲系统。首先介绍了系统的构成图，然后简要介绍了硬件的设计，最后介绍了软件的构成图，并给出了发送编码数据处理和拨号处理的具体程序。由于 MSP430 单片机具有丰富的串口资源，因此读者可以利用串口实现 485 总线，从而使系统具有联网功能。

第 28 章

MSP430 单片机与 DSP 的 HPI 接口的设计与实现

在一些高速处理系统中，经常使用 DSP 进行高速运算处理。由于 DSP 芯片的控制功能不强，因此需要将单片机和 DSP 结合使用。单片机完成系统的控制功能，DSP 完成系统的高速运算处理功能。本章介绍 MSP430 单片机与 DSP 连接的 HPI 接口设计，并给出相应的程序代码。

28.1 硬件设计

本系统的硬件设计主要是 MSP430 单片机和 DSP 芯片的 HPI 接口设计。为了便于理解 HPI 接口设计，首先简要介绍一下 HPI 口。

28.1.1 HPI 口

TI 公司的 TMS320VC54XX 系列 DSP 芯片都有主机接口（Host Port Interface，即 HPI 口）。HPI 口具有以下特性：

- 能够按序列传输数据，也能随机传输数据。DSP 和单片机可以互相中断。
- 有丰富的控制管脚，接口非常灵活，能在停止硬件评估时继续传输数据。
- 能够通过 DMA 访问片内的 RAM，能按字的方式传输地址和数据。

TMS320VC54XX 系列 DSP 的 HPI 口有 8 位模式，也有 16 位模式，本章介绍的是 8 位模式，并且以 TMS320VC5409 芯片为基础。为了增加对该 HPI 口的认识，下面给出 HPI 口的框图，如图 28-1 所示。

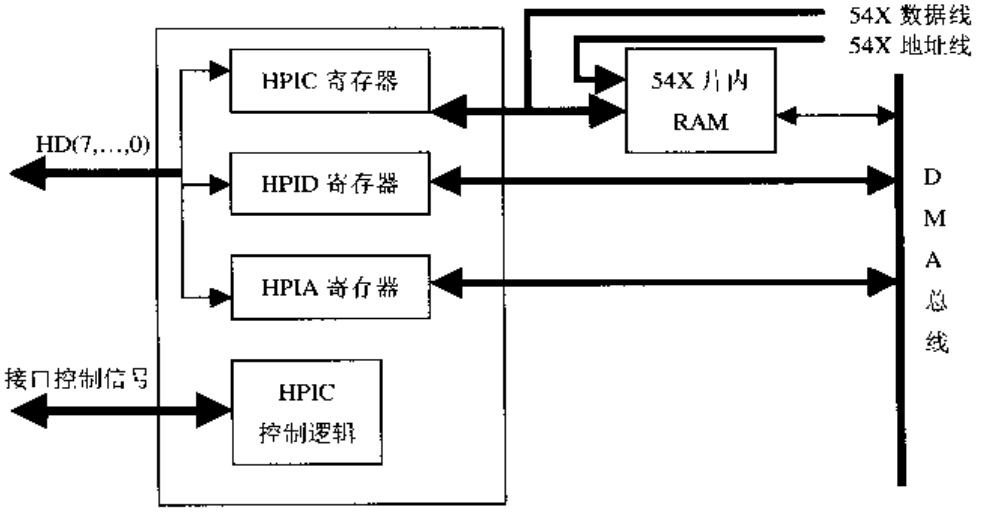


图 28-1　HPI 口框图

由图 28-1 可以看出，单片机与 DSP 主要通过数据线和控制线进行接口。为了便于理解硬件的接口设计，下面介绍 HPI 口的管脚。

- HD7~HD0：双向的数据总线。当 HPI16 为 1 时或者 HPI8 不使能时，这些管脚可以作为一般 I/O 管脚使用。在没有数据输出的情况下，这些管脚处于高阻状态。
- HCNTL1、HCNTL0：控制管脚。使单片机用来选择访问 HPI 口的哪一个寄存器。当 HCNTL1、HCNTL0 的值为 00 时，访问 HPIC 寄存器；当值为 01 时，访问 HPID 寄存器，并且此时 HPIA 的值自动增加；当值为 10 时，访问 HPIA 寄存器；当值为 11 时，访问 HPID 寄存器，此时 HPIA 的值不发生变化。内部接有上拉电阻，只在 HPIEA 为 0 时有效。
- HBIL：字节识别管脚。用来表示传输的是第一个字节还是第二个字节。内部接有上拉电阻，只在 HPIEA 为 0 时有效。
- HCS：片选信号，低电平有效。内部接有上拉电阻，只在 HPIEA 为 0 时有效。
- HDS1、HDS2：数据选通信号。用来控制传输，被主机的读写选通来驱动。内部接有上拉电阻，只在 HPIEA 为 0 时有效。
- HAS：地址选通。内部接有上拉电阻，只在 HPIEA 为 0 时有效。
- HR/W：读写控制管脚。用来控制 HPI 的数据传输方向。内部接有上拉电阻，只在

HPIEA 为 0 时有效。

- HRDY：准备就绪信号。DSP 给单片机的确认信号，表示可以传输下一个数据。
- HINT：中断管脚。DSP 给单片机的中断信号。
- HPIEA：HPI 模块选择管脚。内部接有下拉电阻。
- HPI16：16 位模式选择管脚。如果该管脚输入为高电平，则选择 16 位模式；如果该管脚输入为低电平，则选择 8 位模式。

经过对 HPI 口的介绍，对 HPI 口有了基本的认识，下面介绍硬件电路的具体设计。

28.1.2　DSP 的 HPI 接口设计

在本章的具体应用中，主要介绍 DSP 的 HPI 接口设计，DSP 的其他相关电路这里不作介绍。如图 28-2 所示为 DSP 的 HPI 接口示意图。

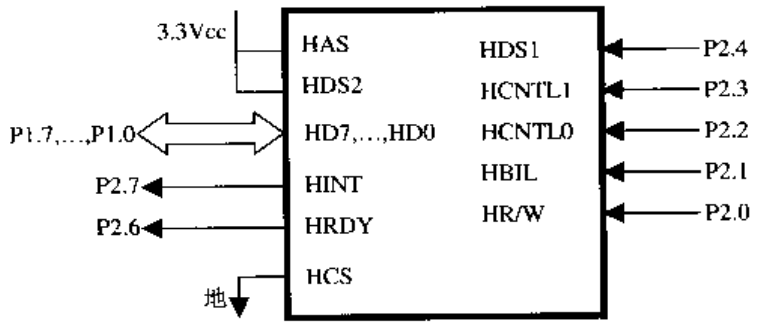


图 28-2　DSP 的 HPI 接口示意图

由图 28-2 可以看出，单片机的 P1 口与 DSP 的 HPI 口的数据线进行连接；单片机的 P2 口与 DSP 的 HPI 口的控制线进行连接；DSP 的 HCS 管脚接地，使 DSP 芯片处于选通状态；DSP 的 HAS 和 HDS2 两个管脚不使用，需要将这两个管脚拉高，一般是通过一个 10kΩ的电阻拉高。

28.1.3　单片机电路

单片机电路非常简单，单片机的 P1 口与 DSP 的 HPI 口的数据线进行连接，单片机的 P2 口与 DSP 的 HPI 口的控制线进行连接。如图 28-3 所示为单片机电路图。

以上对硬件设计作了介绍，对硬件的接口有了清楚的认识，下面在此基础上来介绍系统的软件设计。

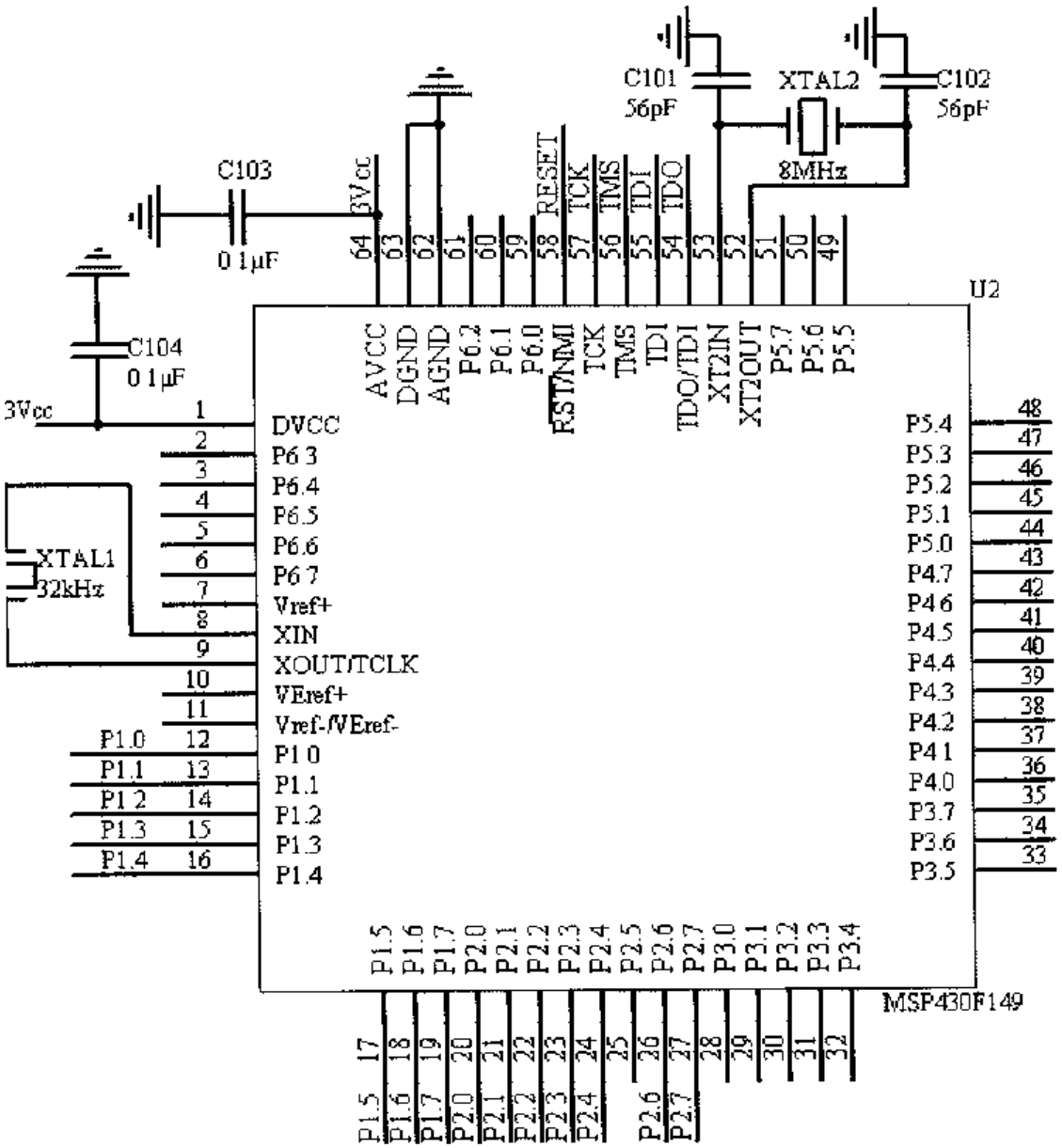


图 28-3　单片机电路

28.2　软件设计

本系统的软件主要包括单片机端的程序和 DSP 端的程序。在介绍具体的程序之前，先介绍一下 DSP 的 HPI 口的寄存器。

28.2.1　HPI 口的寄存器

单片机作为主机访问 DSP 的内存主要是通过 3 个寄存器来实现的。这 3 个寄存器分别是地址寄存器（HPIA）、数据寄存器（HPID）和控制寄存器（HPIC）。单片机能访问各个寄存器，但是 DSP 只能访问 HPIC 寄存器。由于 HPIC 寄存器控制 HPI 口的工作状态，所以下面对该寄存器进行详细介绍。

HPIC 寄存器的各个位字段用来控制、监视 HPI 口的操作。这些位字段为 BOB、DSPINT、HINT、XHPIA 和 HPIENA。这些位字段的作用分别是：

- BOB：表示字节号。在进行双字节传输时，该位用来指示字节的位置。BOB 为 1 时，传输的第一个字节为字的低字节；BOB 为 0 时，传输的第一个字节为字的高字节。该位只能由单片机进行读写。该位应该在传输开始前初始化。
- DSPINT：单片机到 DSP 的中断。当单片机设置该位为 1 时，DSP 就产生中断。该位只能由单片机进行设置。如果单片机或者 DSP 读该位时，该位的值始终为 0。
- HINT：DSP 到单片机的中断。该位确定 DSP 的中断输出状态，用来中断单片机。当该位设置为 1 时，HINT 管脚输出低电平；当该位设置为 0 时，HINT 管脚输出高电平。该位只能由 DSP 进行设置，并且该位只能由单片机清除，单片机向该字段写入 1 就清除该字段的值，即该字段的值就变成“0”。
- XHPIA：扩展地址使能。
- HPIENA：HPI 使能状态位。

在单片机端，HPIC 是由同样内容的高低字节组成的一个 16 位寄存器。在 DSP 端，高字节没有被使用。当单片机往 HPIC 寄存器写入内容时，第一个字节和第二个字节的内容必须完全一致。经过对 HPIC 的介绍，对 HPI 口的操作有了基本的认识。下面结合上面的介绍来分析具体的程序实现。

28.2.2　单片机程序

单片机程序主要包括初始化程序和 HPI 口程序，下面对程序进行具体分析。

1．初始化程序

初始化程序用来初始化端口和一些全局变量。下面为初始化程序代码。

```
void Init_Port(void)
{
```

```
    //将 P1、P2 口所有的管脚在初始化的时候设置为输入方式
    P1DIR = 0;
    P2DIR = 0;

    //将 P1、P2 口所有的管脚设置为一般 I/O 口
    P1SEL = 0;
    P2SEL = 0;

    //P2.0 作为输出管脚
    P2DIR |= BIT0;
    //P2.1 作为输出管脚
    P2DIR |= BIT1;
    //P2.2 作为输出管脚
    P2DIR |= BIT2;
    //P2.3 作为输出管脚
    P2DIR |= BIT3;
    //P2.4 作为输出管脚
    P2DIR |= BIT4;

    //HDS1 输出高电平
    P2OUT = 0x10;

    // 管脚 P2.7 使能中断
    P2IE |= BIT7;
    // 对应的管脚由高到低电平跳变使相应的标志置位
    P2IES |= BIT7;

    //打开中断
    _EINT();

    nLen = 0;
    nFlag = 0;
    return;
}
```

另外，初始化程序还包括时钟的初始化，时钟的初始化程序和第 3 章介绍的一致，请具体参看第 3 章的程序代码。

2. HPI 口程序

HPI 口程序主要提供 HPI 口的读写操作。下面给出 HPI 口的读写代码。

```
void HPI_Write(int nValue,char nRegCode)
{
    char hi;
    char lo;
    char temp;
```

```
    hi = (char)((nValue >> 8) & 0x0ff);
    lo = (char)(nValue & 0x0ff);
    //P1 口为输出方向
    P1DIR = 0xFF;

    //传输第 1 个字节
    P1OUT = lo;
    //设置 HDS1 为高
    temp = (char)(nRegCode | 0x10);
    //往 P2 口输出控制信号
    P2OUT = temp;
    //设置 HDS1 为低，开始 HPI
    P2OUT &= ~(BIT4);
    //设置 HDS1 为高，结束 HPI
    P2OUT |= BIT4;

    //传输第 2 个字节
    P1OUT = hi;
    //设置 HDS1 为高、HBIL 为高
    temp = (char)(nRegCode | 0x12);
    //往 P2 口输出控制信号
    P2OUT = temp;
    //设置 HDS1 为低，开始 HPI
    P2OUT &= ~(BIT4);
    //设置 HDS1 为高，结束 HPI
    P2OUT |= BIT4;

    return;
}
int HPI_Read(char nRegCode)
{
    char hi;
    char lo;
    char temp;
    int res;

    //P1 口为输入方向
    P1DIR = 0x00;

    //设置 HDS1 为高，HR/W 为高
    temp = (char)(nRegCode | 0x11);
    //往 P2 口输出控制信号
    P2OUT = temp;
    //设置 HDS1 为低，开始 HPI
    P2OUT &= ~(BIT4);
    lo = P1IN;
    //设置 HDS1 为高，结束 HPI
```

```
    P2OUT |= BIT4;

    //读第2个字节
    //设置HDS1为高、HBIL为高
    temp = (char)(nRegCode | 0x13);
    //往P2口输出控制信号
    P2OUT = temp;
    //设置HDS1为低，开始HPI
    P2OUT &= ~(BIT4);
    hi = P1IN;
    //设置HDS1为高，结束HPI
    P2OUT |= BIT4;

    res = (int)(hi);
    res <<= 8;
    res = res + lo;
    return res;
}
```

通过上面的基本操作就可以实现单片机与DSP之间的数据交互。

下面给出一个简单的测试程序。在测试程序里，使用了全局变量和常量。具体代码如下：

```
#define HPI_CONTROL_REG              0x00
#define HPI_DATA_WITHADDR_INC        0x04
#define HPI_ADDR_REG                 0x08
#define HPI_DATA_REG                 0x0C

#define SET_BOB                      0x101
#define CLR_HINT                     0x909
#define INT_DSP                      0x505

int pBuf[128];
char nFlag;
void main(void)
{
    // 关闭看门狗
    WDTCTL = WDTPW + WDTHOLD;
    // 关闭中断
    _DINT();

    Init_CLK();
    Init_Port();
    //初始化HPI
    HPI_Write(SET_BOB,HPI_CONTROL_REG);
    //初始地址
    HPI_Write(0x60,HPI_ADDR_REG);
```

```
    //往 DSP 写入一个数据
    HPI_Write(12,HPI_DATA_REG);
    //往 DSP 写入一个数据，地址增加
    HPI_Write(12,HPI_DATA_WITHADDR_INC);

    //中断 DSP
    HPI_Write(INT_DSP,HPI_CONTROL_REG);
    for(;;)
    {
        //等待 DSP 返回
        if(nFlag == 1) break;
    }
}
```

上面的程序将数据发送给 DSP，并等待 DSP 的返回。由前面的接口电路设计知道，当 DSP 有返回时，会在 P2.7 管脚产生中断，下面为具体的中断程序。

```
interrupt [PORT2_VECTOR] void R_B_ISR(void)
{
    if(P2IFG & BIT7)
    {
        // 清除中断标志位
        P2IFG &= ~(BIT7);
        //清除 DSP 中断
        HPI_Write(CLR_HINT,HPI_CONTROL_REG);
        //从地址为 62 处开始读数
        //先确定起始地址
        HPI_Write(0x62,HPI_ADDR_REG);
        //读数
        pBuf[0] = HPI_Read(HPI_DATA_REG);
        //设置标志
        nFlag = 1;
    }
}
```

在上面的中断程序里，单片机需要清除管脚的中断标志。另外，也需要清除 DSP 中断。通过“pBuf[0]”和“nFlag”实现中断程序与主程序之间的数据交互。

28.2.3　DSP 程序

DSP 程序主要包括主程序、初始化 DSP 程序和中断服务程序。

1. 主程序

```
extern init_54();
int flag=0;
```

```
int main()
{
    asm(" ssbx frct ");
    asm(" nop ");
    asm(" ssbx sxm ");
    asm(" nop ");
    asm(" ssbx ovm ");
    asm(" nop ");

    init 54();

    asm("   rsbx   intm ");
    asm("   nop     ");
    asm("   nop     ");
    asm("   nop     ");

    for (;;)
    {
        if (flag==1)
        {
            flag=0;
        }
    }
}
```

上面的上程序首先设置 DSP 状态寄存器的几个位，然后调用 DSP 的初始化程序。

2. 初始化程序

初始化程序由 DSP 汇编实现（在 DSP 开发中，C 语言程序和汇编程序中的函数和变量可以互相调用）。初始化程序如下：

```
_init_54:
          STM     #K_PMST,PMST
          STM     #K_SWWSR,SWWSR
          STM     #K_BSCR,BSCR
          NOP
          NOP
          NOP
          STM     #0,CLKMD
          NOP
          NOP
          STM     #K_CLKMD,CLKMD
          STM     #0, IMR
          NOP
          NOP
          NOP
```

```
NOP
STM     #0FFFFh, IFR ;
STM     #0200h, IMR
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
RET
```

上面的程序主要是设置几个寄存器，并且设置 HPI 的中断。在上面的程序中，需要注意的是：汇编指令不能和标号在同一列，即标号“_init_54”必须位于最左边，而指令必须从左边开始向右边空几列。

3. 中断服务程序

DSP 处理基本是基于中断服务结构的，即硬件中断都在中断程序里进行处理。下面为中断处理程序。

```
_HPI_ISR:
        PSHM        AL
        PSHM        AH
        PSHM        AG
        PSHM        BL
        PSHM        BH
        PSHM        BG
        PSHM        AR1
        PSHM        AR2
        PSHM        AR3
        PSHM        T
        PSHM        BRC
        PSHM        RSA
        PSHM        REA

        STM #60, AR3
        STM #61, AR4
        STM #62, AR5
        ADD *AR3, *AR4, A
        STH A, *AR5
        STM #08h, HPIC ; INTERRUPT
        STM #0FFFFh, IFR

        ST  #1,*(_Flag)
```

```
        POPM        REA
        POPM        RSA
        POPM        BRC
        POPM        T
        POPM        AR3
        POPM        AR2
        POPM        AR1
        POPM        BG
        POPM        BH
        POPM        BL
        POPM        AG
        POPM        AH
        POPM        AL
        POPM        ST1
        POPM        ST0
        RETE
```

上面的程序主要是从 60、61 两个地址里取出数据进行加，然后将结果放到地址为 62 的存储单元，并且中断单片机；同时设置“Flag”为 1，通知主程序接收到来自单片机的数据。DSP 的中断是由它的中断向量表确定入口的。不同的 DSP 有不同的中断向量表，下面给出 TMS320VC5409 的中断向量表。

```
        .sect "vectors"

start:  B    _c_int00
        NOP
        NOP
nmi:    B       start
        NOP
        NOP

; software interrupts
sint17 .space 4*16
sint18 .space 4*16
sint19 .space 4*16
sint20 .space 4*16
sint21 .space 4*16
sint22 .space 4*16
sint23 .space 4*16
sint24 .space 4*16
sint25 .space 4*16
sint26 .space 4*16
sint27 .space 4*16
sint28 .space 4*16
sint29 .space 4*16
sint30 .space 4*16
int0:  B        start
```

```
    NOP
    NOP
int1: B         start
    NOP
    NOP
int2: B         start
    NOP
    NOP
tint0:B         start
    NOP
    NOP
brint0:B        start
    NOP
    NOP
bxint0:B        start
    NOP
    NOP
    NOP
DMAC0:B         start
    NOP
    NOP
TINT1: B        start
    NOP
    NOP
INT3:B          start
    NOP
    NOP
hpint:  _HPI_ISR
    NOP
    NOP
    NOP
brint1:B        start
    NOP
    NOP
    NOP
bxint1:B        start
    NOP
    NOP
DMAC4:B         start
   NOP
    NOP
DMAC5:B     start
    NOP
    NOP
```

在上面的代码中，“.sect "vectors"”是必需的，用来指明所在的段。

关于 DSP 的程序设计这里不再进行详细介绍。

28.3 实例总结

本章介绍了 MSP430 单片机与 DSP 的 HPI 口的接口设计，并给出了相应的程序。通过 HPI 口，单片机可以与 DSP 进行数据交互，这样可以使单片机和 DSP 结合起来使用，使 DSP 处理运算量比较大的任务，而单片机则完成复杂的控制功能。使两者实现互补，实现更为完备的功能。

第 29 章

基于 MSP430 单片机实现的无线传输模块

在一些工业控制、消费电子等领域，无线应用越来越多，一些无线鼠标、无线键盘、无线抄表系统应运而生。本章介绍采用 nRF2401 芯片实现的无线传输模块，首先介绍它的硬件设计，然后介绍它的软件设计。

29.1 硬件设计

系统硬件主要是 MSP430 单片机与 nRF2401 芯片的接口。在介绍具体的硬件电路之前，首先介绍一下 nRF2401 芯片。

29.1.1 nRF2401 芯片

nRF2401 是单片射频收发芯片，工作于 2.4GHz～2.5GHz ISM 频段。nRF2401 芯片具有以下特点：

- 使用全球开放的 2.4GHz 频段，125 个频道，满足多频及跳频需要。
- 最高速率达 1Mbps，具有高数据吞吐量。
- 发送功率、工作频率等所有工作参数全部通过软件设置完成。
- 低功耗应用，供电电压为 1.9V～3.6V，满足低功耗设计需要。

- 芯片内部设置了专门的稳压电路，使用各种电源均有很好的通信效果。
- 每个芯片可以通过软件设置达 40 位地址，只有收到本机地址时才会输出数据（提供一个中断指示），编程很方便。
- nRF2401 内置了 CRC 纠错、检错硬件电路和协议，软件开发相当简单。
- nRF2401 的 DuoCeiver 技术可以同时接收两个不同频道的数据。

nRF2401 内置了地址解码器、先入先出堆栈区、解调处理器、时钟处理器、GFSK 滤波器、低噪声放大器、频率合成器，功率放大器等功能模块。如图 29-1 所示为 nRF2401 芯片的框图。

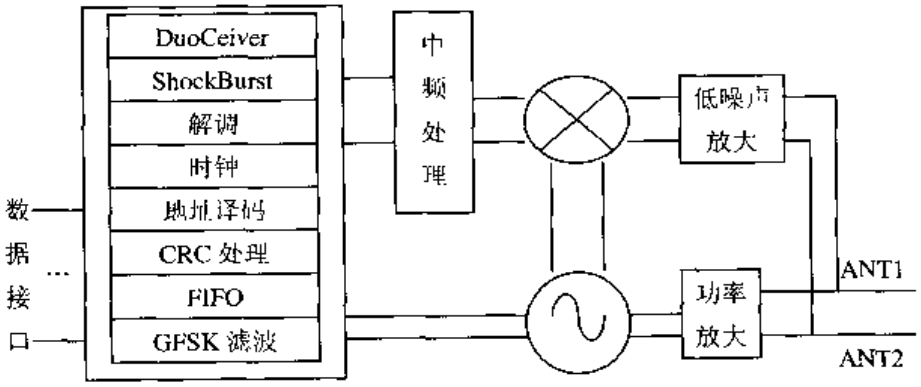


图 29-1　nRF2401 芯片的框图

为了便于后面的硬件设计，下面给出 nRF2401 芯片的管脚图。如图 29-2 所示。

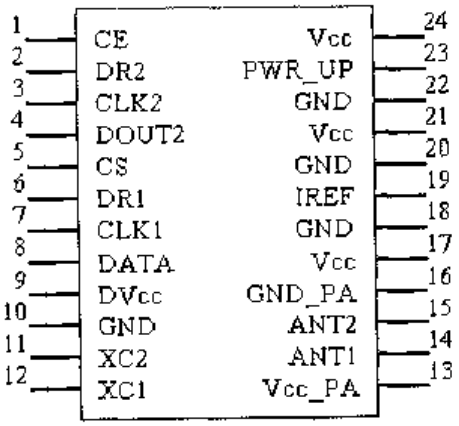


图 29-2　nRF2401 芯片管脚图

由图 29-2 可以看出，nRF2401 芯片共有 24 个管脚，下面对具体的管脚进行介绍。

- CE：接收或者发送选择管脚。
- DR2、DR1：两个频道的接收数据准备好管脚。
- CLK2、CLK1：两个频道的数据时钟管脚。

- DOUT2：频道 2 的数据输出管脚。
- DATA：频道 1 的数据输入输出管脚。
- CS：配置模式的片选信号管脚。
- DVcc：数字电压的输出管脚。
- GND：接地管脚。
- XC2、XC1：晶体管脚。
- Vcc_PA：功率放大器的供电管脚。
- ANT1、ANT2：天线接口管脚。
- GND_PA：接地管脚。
- Vcc：供电管脚。
- IREF：模拟输入的外部参考管脚。
- PWR_UP：芯片激活管脚。

经过对 nRF2401 芯片的简单介绍，对该款芯片有了基本的认识，下面介绍硬件电路的具体设计。

29.1.2　接口电路

硬件的接口电路主要是 MSP430 单片机与 nRF2401 芯片的接口设计。nRF2401 通过外接晶体为它提供工作所用的时钟。nRF2401 还必须有天线电路，这样才能进行数据的收发。nRF2401 通过 SPI 口与 MSP430 单片机进行连接，由于 nRF2401 的 SPI 口只有 1 个数据管脚（DATA 管脚），所以该管脚直接与 MSP430 单片机的 SPI 口的数据收发管脚（MOSI、MISO）进行连接。nRF2401 的外围电路非常简单。如图 29-3 所示为 nRF2401 芯片接口电路图。

由图 29-3 可以看出，nRF2401 芯片的 CE、CS、DR1 和 PWR_UP 管脚分别与 MSP430 单片机的 P1.0、P1.1、P1.2 和 P1.3 管脚进行连接。P1.0、P1.1 和 P1.3 管脚只是用来控制 nRF2401 芯片的工作状态，而 P1.2 管脚用来指示 nRF2401 是否接收到数据，由于 P1 口的管脚有中断功能，因此可以将 P1.2 管脚设置成中断方式。由于 nRF2401 的 SPI 口只有 1 个数据管脚（DATA），因此将 DATA 管脚分别与 MSP430 单片机的 SPI 口的数据收发管脚（MOSI、MISO）进行连接。

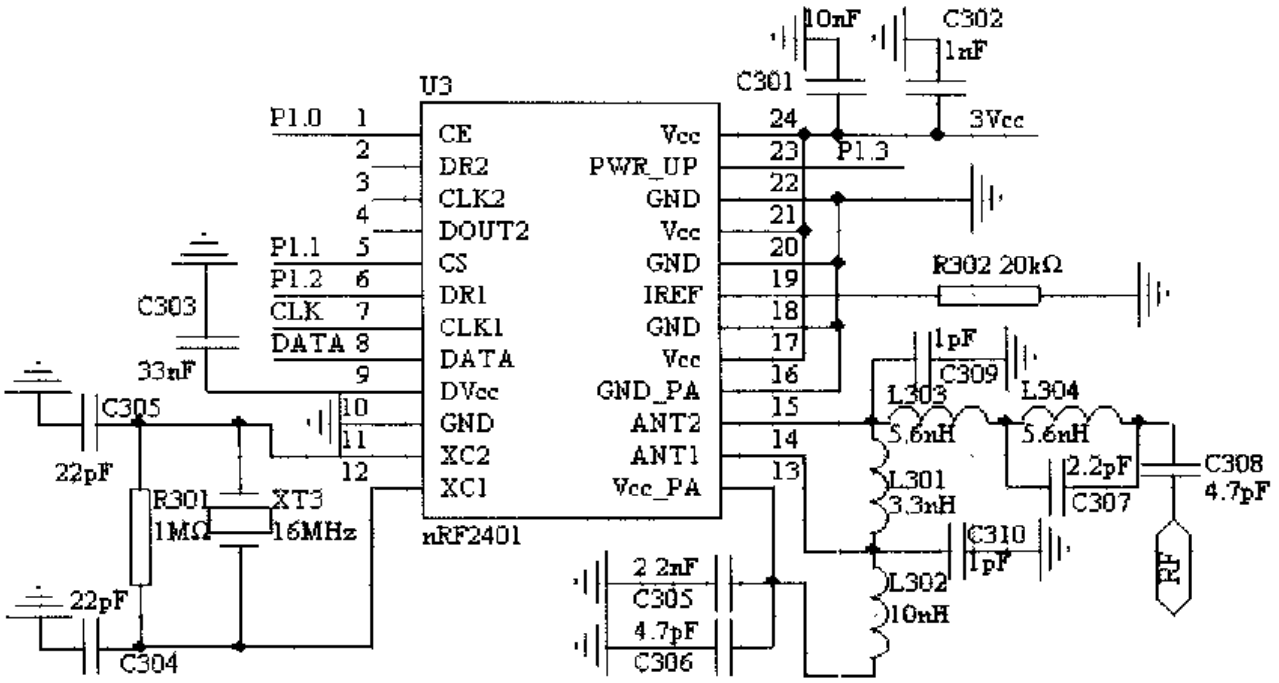


图 29-3　nRF2401 芯片接口电路图

29.1.3　单片机电路

前面的接口电路已经大致介绍了单片机电路，这里只强调一下 SPI 口接口和中断管脚。由于 SPI 口的 2 个数据管脚都连接到 nRF2401 芯片的 DATA 管脚，因此 MSP430 单片机的 SPI 口的数据管脚都需要串接 10kΩ的电阻。由于 DR1 是高电平有效，因此需要将 P1.2 管脚拉低。如图 29-4 所示为单片机电路的示意图。

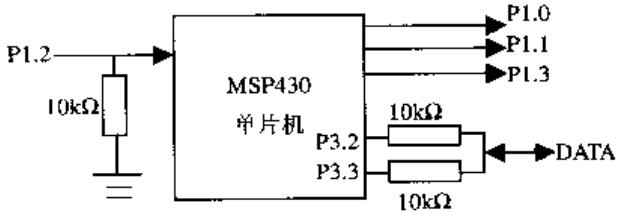


图 29-4　单片机电路示意图

以上只是给出了接口部分的示意图，至于其他振荡管脚、仿真管脚等，相信读者应该很熟悉，在此就不再介绍了。

经过上面对硬件的介绍，对系统的工作有了一定的了解，下面介绍系统的软件设计。

29.2　软件设计

系统软件设计主要是实现单片机与 nRF2401 芯片之间的通信，从而实现数据的无线传输。在介绍具体的程序实现前，先对 nRF2401 芯片的操作进行简单的介绍。

29.2.1　nRF2401 芯片操作

nRF2401 有 4 种工作模式，即收发模式、配置模式、空闲模式和关机模式。nRF2401 的工作模式由 PWR_UP、CE 和 CS 这 3 个管脚决定。表 29-1 中给出了 nRF2401 芯片的 4 种工作模式。

表 29-1　nRF2401 芯片的 4 种工作模式

工作模式	PWR_UP	CE	CS
收发模式	1	1	0
配置模式	1	0	1
空闲模式	1	0	0
关机模式	0	x	x

由表 29-1 可以看出，由 PWR_UP、CE 和 CS 这 3 个管脚就可以决定芯片的工作模式。下面对收发模式和配置模式进行简单介绍。

1．收发模式

nRF2401 芯片的收发模式有 ShockBurst 收发模式和直接收发模式两种，这里只介绍 ShockBurst 收发模式。收发模式由器件配置字决定，具体配置在下面的配置模式中介绍。

在 ShockBurst 收发模式下，主要有以下特点：

- 使用片内的 FIFO，数据从单片机低速送入，由 nRF2401 高速发送，这样可以尽量节能。
- 射频协议由芯片内部进行处理，这样非常方便进行程序开发。
- nRF2401 自动处理数据帧的头和 CRC 校验码，在接收数据时，自动把数据帧的头和 CRC 校验码移去；在发送数据时，自动加上数据帧的头和 CRC 校验码。

下面介绍在 ShockBurst 模式下的收发流程。

（1）ShockBurstTM 发送流程

在发送状态下，主要通过 CE、CLK1 和 DATA 这 3 个管脚来实现数据的发送。具体

流程如下：

1 当单片机有数据要发送时，先把 CE 置高，使 nRF2401 进入发送模式。

2 单片机把地址和要发送的数据通过 SPI 口写入 nRF2401 的 FIFO。

3 单片机将 CE 置低，使 nRF2401 开始以 ShockBurst 模式发送数据。

（2）ShockBurstTM 接收流程

在接收状态下，主要通过 CE、CLK1、DR1 和 DATA 这 4 个管脚来实现数据的接收。具体流程如下：

1 单片机将 CE 置高，进入接收状态。

2 延时 200μs 后，nRF2401 进入监视状态，等待数据包的到来。

3 当接收到正确的数据包后，nRF2401 自动把数据帧的头、地址和 CRC 校验位去掉。

4 nRF2401 将 DR1 置高来通知单片机有数据到来。

5 单片机将数据从 nRF2401 读出。

6 所有数据读完后，nRF2401 把 DR1 置低。如果 CE 继续设置为高，则继续等待下一个数据包；如果 CE 为低，开始其他工作流程。

2. 配置模式

nRF2401 芯片的配置是通过 CS、CLK1 和 DATA 这 3 个管脚来完成的。ShockBurst 收发模式需要 15 字节的配置字，而直接收发模式只需要 2 字节的配置字。这里只介绍 ShockBurst 收发模式的配置。ShockBurst 的配置字使 nRF2401 能够处理射频协议，在配置完成后，在 nRF2401 工作的过程中，只需改变其最低一个字节中的内容，以实现接收模式和发送模式之间的切换。表 29-2 给出了 nRF2401 芯片的配置字节信息。

表 29-2 nRF2401 芯片的配置字节信息

位的位置	位 数	名 字	功 能
143~120	24	TEST	保留
119~112	8	DATA2_W	接收频道 2 的数据的长度
111~104	8	DATA1_W	接收频道 1 的数据的长度
103~64	40	ADDR2	接收频道 2 的地址
63~24	40	ADDR1	接收频道 1 的地址
23~18	6	ADDR_W	接收频道地址的位数
17	1	CRC_L	8 位或 16 位的 CRC
16	1	CRC_EN	CRC 使能
15	1	RX2_EN	频道 2 使能
14	1	CM	通信模式设置（ShockBurst 收发模式还是直接收发模式）
13	1	RFDR_SB	速率设置

续表

位的位置	位　数	名　字	功　能
12~10	3	XO_F	晶体频率
9~8	2	RF_PWR	RF 输出功率
7~1	7	RF_CH	频道设置
0	1	RXEN	接收或者发送模式

由表 29-2 可以看出，只要向 nRF2401 输入适当的配置信息，就可以设置 nRF2401 的相应工作状态和工作参数。

29.2.2　软件设计

整个软件设计包括初始化程序、管脚模拟程序和数据收发程序，下面分别给出具体的程序代码。

1．初始化程序

初始化程序主要是对管脚和 SPI 口进行初始化，设置相应的状态，下面为具体的程序代码。

```
void Port_Init(void)
{
    P1DIR = 0;
    //设置 CE 为输出管脚
    P1DIR |= BIT0;
    //设置 CS 为输出管脚
    P1DIR |= BIT1;
    //设置 PWR_UP 为输出管脚
    P1DIR |= BIT3;
    //将中断寄存器清零
    P1IE = 0;
    P1IES = 0;
    P1IFG = 0;
    //管脚 P1.2 使能中断
    P1IE |= BIT2;
    //对应的管脚由低到高电平跳变使相应的标志置位
    P1IES &= ~(BIT2);

    //将 P3 口所有的管脚设置为一般 I/O 口
    P3SEL = 0;
    //P3.1、P3.2、P3.3 被分配为 SPI 口
    P3SEL = BIT3 + BIT2 + BIT1;
    //P3.3 作为输出管脚
    P3DIR |= BIT3;
```

```
    //P3.1 作为输出管脚
    P3DIR |= BIT1;
    return;
}

void Init_SPI (void)
{
    //SPI0 模块允许
    ME1 |= USPIE0;
    //将寄存器的内容清零
    U0CTL = 0X00;
    //数据为 8 比特，选择 SPI 模式，单片机为主机模式
    U0CTL |= CHAR + SYNC + MM;

    //将寄存器的内容清零
    U0TCTL = 0X00;
    // 时钟源为 SMCLK，选择 3 线模式
    U0TCTL = CKPH + SSEL1 + SSEL0 + STC;

    //传输时钟为 SMCLK / 800
    UBR0_0 = 0X20;
    UBR1_0 = 0X03;
    //调整寄存器，没有调整
    UMCTL_0 = 0X00;

    //发送中断允许
    IE1 |= UTXIE0;
}
```

2. 管脚模拟程序

管脚模拟程序主要是对某个管脚产生高电平或者低电平。具体程序代码如下：

```
void CE_HI(void)
{
    P1OUT |= BIT0;
    return;
}
void CE_LO(void)
{
    P1OUT &= ~(BIT0);
    return;
}
void CS_HI(void)
{
    P1OUT |= BIT1;
    return;
}
```

```
void CS_LO(void)
{
    P1OUT &= ~(BIT1);
    return;
}
void PWR_UP_HI(void)
{
    P1OUT |= BIT3;
    return;
}
void PWR_UP_LO(void)
{
    P1OUT &= ~(BIT3);
    return;
}
```

3. 数据收发程序

根据前面对 nRF2401 芯片工作模式的介绍知道，只有配置好 nRF2401，才可以实现数据的收发。下面分别为配置程序和数据收发程序。

```
void Init_RF2401(void)
{
    int i;
    nDR = 0;
    //激活 nRF2401
    PWR_UP_HI();
    //延迟
    Delay_ms(4);
    CS_HI();
    Delay_us(100);
    CE_LO();
    Delay_us(100);
    //发送配置信息
    for(i = 0;i < 15;i++)
    {
        UART0_TX_BUF[i] = rxConfig[i];
    }
    nTX0_Len = 15;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;
}

void TransmitPacket(unsigned char nVal)
{
    unsigned char i;
```

```
    CS_HI();
    Delay_us(10);
    TXBUF0 = 0x05;
    CS_LO();
    Delay_us(300);

    CE_HI();
    Delay_us(10);
    for(i = 0;i < ADDR_COUNT;i++)
    {
        UART0_TX_BUF[i] = rxConfig[ADDR_INDEX + i];
    }
    UART0_TX_BUF[i] = nVal;
    nTX0_Len = ADDR_COUNT + 1;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;
    CE_LO();
    //延迟
    Delay_us(300);
}
char ReceivePacket()
{
    char nVal;

    CS_HI();
    Delay_us(10);
    TXBUF0 = 0x04;
    CS_LO();
    Delay_us(300);

    CE_HI();
    while(true)
    {
        if(nDR == 1)
        {
            break;
        }

    }
    TXBUF0 = 0x0;
    while ((IFG1 & UTXIFG0) == 0) ;
    nVal = RXBUF0;
    CE_LO();
    //延迟
    Delay_us(300);
    return nVal;
}
```

在上面的程序中，通过 DR1 管脚来判断是否有数据到达。由于 nRF2401 的 DR1 管脚与单片机的 P1.2 管脚进行连接，所以采用中断方式实现。具体的中断服务程序代码如下：

```
interrupt [PORT1_VECTOR] void DR_ISR(void)
{
    if(P1IFG & BIT2)
    {
        nDR = 1;
        // 清除中断标志位
        P1IFG &= ~(BIT2);
    }
}
```

29.3　实例总结

本章简单介绍了 nRF2401 芯片的硬件设计和软件设计。nRF2401 通过 ShockBurstTM 收发模式进行无线数据收发，收发可靠。另外，nRF2401 芯片外形尺寸小，需要的外围元器件也少，因此，在工业控制、消费电子等各个领域都具有广阔的应用前景。在实际应用中，PCB 设计对 nRF2401 的整体性能影响很大，PCB 设计是在 nRF2401 收发系统的开发过程中主要的工作之一，所以在进行 PCB 设计时，必须考虑到各种电磁干扰，注意调整电阻、电容和电感的位置，以便得到比较好的效果。

第六篇

控制应用

第 30 章

基于 MSP430 单片机的步进电机控制器的设计与实现

在单片机应用中，很多控制领域都涉及到电机驱动。本章介绍基于 MSP430 系列单片机实现的步进电机控制器。首先介绍步进电机控制器的电路设计，然后介绍它的软件设计。

30.1 控制器电路设计

该系统的硬件系统相对很简单，主要有电源电路、复位监控电路、电机驱动电路、串口通信电路和单片机电路，其中复位监控电路与前面第 2 章的图 2-3 一样，这里就不再进行介绍了。下面对其他的电路分别进行具体介绍。

30.1.1 电机驱动电路

电机驱动电路主要由驱动芯片组成，本系统使用的电机驱动芯片为 UC3717A。UC3717A 芯片使用非常简单，它通过 3 个输入管脚（Phase、I1 和 I0）接收输入的参数，在 2 个输出管脚（Aout 和 Bout）上输出相应的控制信号。由于 UC3717A 包含一个 H 桥，因此电机驱动电路由两片 UC3717A 构成完整的驱动电路。如图 30-1 所示为部分电机驱动电路图。

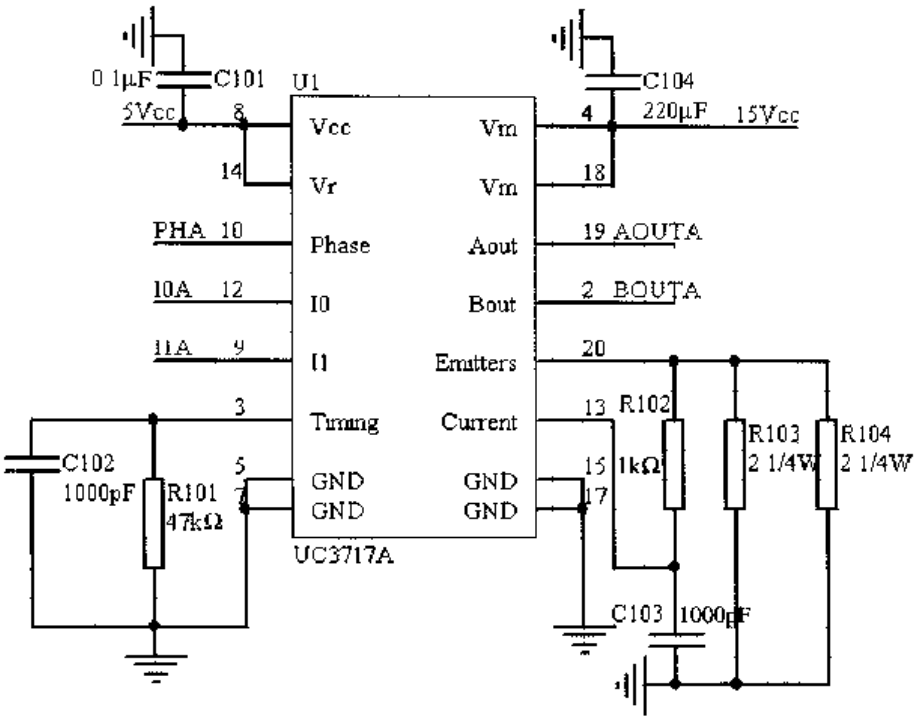


图 30-1　部分电机驱动电路

因为 UC3717A 芯片只有一个 H 桥，因此完成的驱动电路由 2 个如图 30-1 所示的电路组成。在该接口电路中，Phase、I1 和 I0 三个管脚分别与单片机进行连接。Aout 和 Bout 管脚与电机进行连接。

由图 30-1 可以看出，该芯片的工作电压为 5V，而单片机的工作电压为 3.3V，但是根据 UC3717A 芯片的数据手册可以看出，该芯片的数字管脚可以与单片机的 I/O 口进行连接，因此 Phase、I1 和 I0 三个管脚直接与单片机的 I/O 口进行连接，不需要进行电平转换。

30.1.2　串口通信电路

在本系统中，PC 机可以向单片机发送命令信息来控制驱动器的工作，单片机和 PC 机的数据传输采用串口通信来实现。由于单片机的接口电平和 PC 机串口的接口电平不一致，所以需要采用串口芯片，本系统中采用的是 SP3220 芯片来完成接口电平的转换。如图 30-2 所示为串口通信电路图。

由图 30-2 可以看出，串口通信电路相对简单。通过一个上拉电阻将 $\overline{\text{SHDN}}$ 管脚拉高，使该芯片一直处于工作状态；如果系统需要处于低功耗状态，也可以通过单片机来控制该管脚。工作的时候将该管脚设置为低电平，在需要处于低功耗的时候将该管脚设置为高电平，这样很容易实现芯片工作状态的控制。在管脚 C1+、C1−、C2+、C2−、V+和 V−处分

别放置 0.1μF 的电容实现充电作用，以满足相应充电泵的要求。管脚 T1OUT、T1IN、R1OUT 和 R1IN 分别是 232 转换的输入输出脚，实现单片机的 TTL 电平与上位机的接口电平的转换。为了减小输入端受到的干扰，还需要在芯片的电源输入管脚处加一个 0.1μF 的电容来实现滤波。

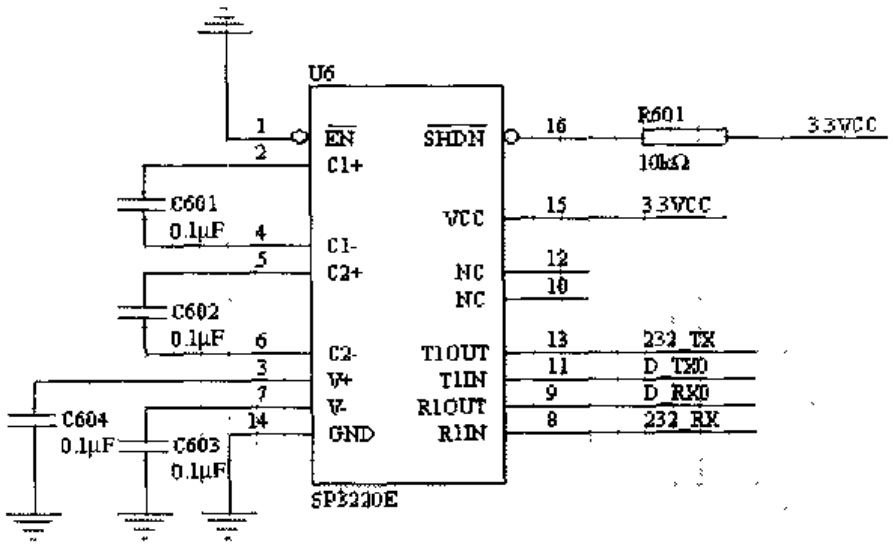


图 30-2　串口通信电路

30.1.3　单片机电路

单片机电路主要完成与 UC3717A 芯片的接口，并且能够与上位机进行通信。单片机与 UC3717A 芯片的接口主要通过单片机的一般 I/O 口与 UC3717A 芯片的 3 个输入管脚（Phase、I1 和 I0）进行连接。虽然单片机与 UC3717A 芯片的供电电压不同，但是由于 UC3717A 芯片的输入高电平最小为 2V，而 MSP430 单片机的输出高电平大于 2V，因此在接口时不需要进行电平转换。单片机与上位机通信通过单片机的串口 0（UART0）实现，由于单片机与上位机的接口电平不一致，所以需要通过串口芯片（SP3220）完成接口电平的转换。另外，单片机还利用 P1 口设计了几个按键，通过使用按键实现一定的控制功能。在设计按键时，利用 P1 口管脚的中断功能来实现，并且选用低电平触发中断的方式。整个系统的单片机电路相对来说十分简单，如图 30-3 所示为单片机电路图。

在图 30-3 中，单片机的 P1.0、P1.1、P1.2 和 P1.3 管脚分别接按键电路，具体的按键电路如图 30-4 所示。

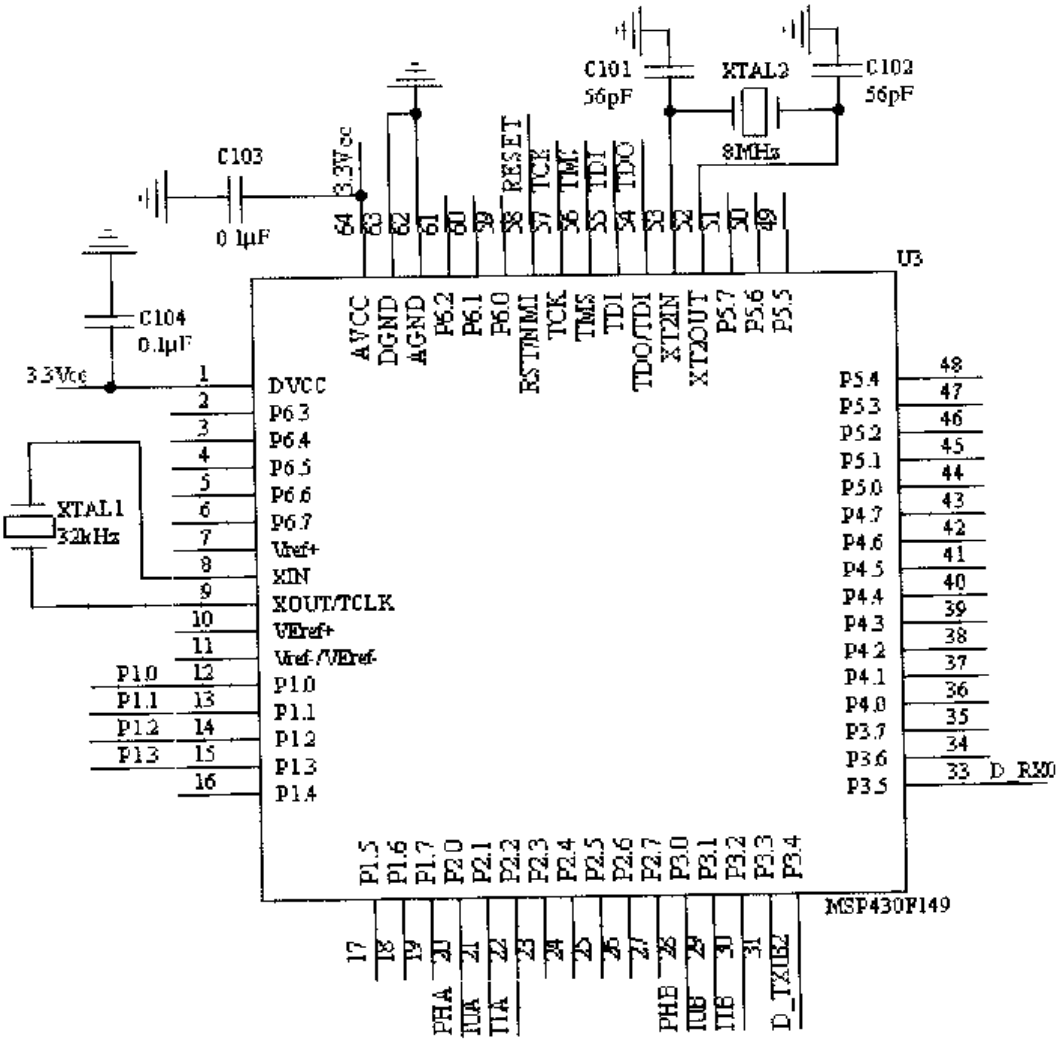


图 30-3　单片机电路

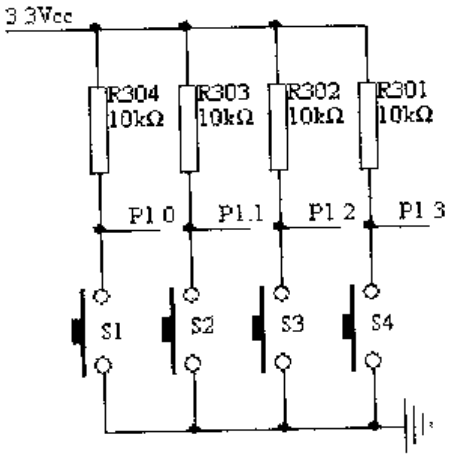


图 30-4　按键电路

在如图 30-4 所示的按键电路中，单片机的每个一般 I/O 口都对应一个按键输入，这样电路实现比较简单，只需要分别从不同的管脚读位就可以获得相应的输入值，程序实现起来非常简单。

30.1.4 电源电路

在该系统中，由于 UC3717A 芯片的 Vm 管脚的电压为 15V，因此整个板的电压为 15V。该系统的供电电压有 15V、5V 和 3.3V，因此需要将 15V 电压转换成 5V 电压，再将 5V 电压转换为 3.3V 电压。由于 5V 电压转换成 3.3V 电压已经在第 2 章中作了介绍，这里只介绍 15V 电压转换成 5V 电压的电路。如图 30-5 所示为 15V 电压转换成 5V 电压的电源电路图。

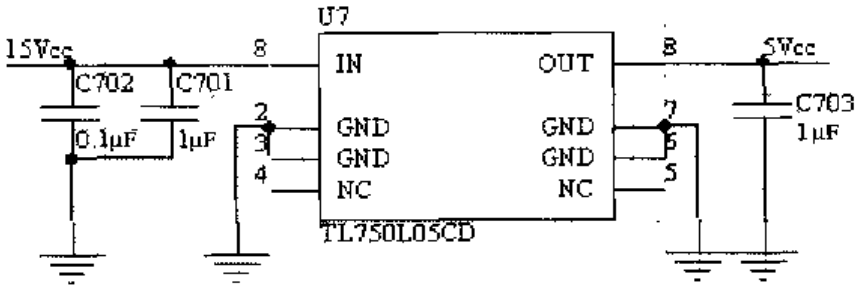


图 30-5 电源电路

为了使输出电源的纹波小，在输出端用了一个 1μF 的电容进行滤波处理。另外，在芯片的输入端也放置两个分别为 0.1μF 和 1μF 的滤波电容，以减小输入端受到的干扰。

30.2 控制器软件设计

本系统的软件主要包括初始化模块、电机驱动模块、串口通信模块和主处理模块。下面分别对各个模块的程序进行具体的分析。

30.2.1 初始化模块

初始化模块主要是初始化端口管脚、初始化时钟、定时器等，下面为具体的初始化程序。

```
void sys_init(void)
{
    // 设置下降沿触发中断
    P1IES = 0x0f;
```

```
    P1IFG = 0x00;
    P1IE = 0x0f;

    // 设置 P2.3、P2.2、P2.1、P2.0 为输出
    // 设置 P3.3、P3.2、P3.1、P3.0 为输出
    P2OUT = 0x00;
    P3OUT = 0x00;
    P2DIR |= 0x0f;
    P3DIR |= 0x0f;

    // 设置 DCO
    Set_DCO(DCO_FREQ);

    // 初始化 Timer A
    timerA_Init();

    // 初始化 UART0
    uart0_Init();

    // 初始化 WDT
    wdt_Init();
}
```

在上面的程序中，调用了几个函数，例如“Set_DCO”，这些函数分别完成设置系统时钟、初始化定时器等功能，这里限于篇幅原因不再进行详细介绍。

30.2.2　电机驱动模块

电机驱动模块主要是通过定时器和看门狗实现在相应的管脚输出不同的状态逻辑来驱动电机驱动芯片，以使电机进行相应的工作。该模块主要使用中断服务程序来进行相应的处理，其中定时器中断服务程序主要根据相应的状态向电机驱动芯片输出相应的值，从而实现电机的不同工作；看门狗中断服务程序主要是根据 P1 口的输入来确定相应的状态，并进行状态的切换。

1．定时器中断服务程序

```
interrupt [TIMERA0_VECTOR] void TimerA_ISR(void)
{
    unsigned char index;
    unsigned char p2 = 0;
    unsigned char p3 = 0;

    // 判断步进速率是否需要改变
    if( change_rate_flag )
```

```
    {
        TACCR0 = rate;
        change_rate_flag = 0;
    }

    // 判断状态
    switch( (state & 0x3) )
    {
    case 0x00:  // 整步、逆时钟方向
        index = stepIndex & 0x03;
        p2 |= fullStepA[index];
        p3 |= fullStepB[index];
        P2OUT = p2;
        P3OUT = p3;
        ++stepIndex;
        break;
    case 0x01:  // 整步、顺时钟方向
        index = stepIndex & 0x03;
        p3 |= fullStepA[index];
        p2 |= fullStepB[index];
        P3OUT = p3;
        P2OUT = p2;
        ++stepIndex;
        break;
    case 0x02:  // 半步、逆时钟方向
        index = stepIndex & 0x07;
        p2 |= HalfStepA[index];
        p3 |= CcwHalfStepB[index];
        P2OUT = p2;
        P3OUT = p3;
        ++stepIndex;
        break;
    case 0x03:  // 半步、顺时钟方向
        index = stepIndex & 0x07;
        p3 |= CwHalfStepB[index];
        p2 |= HalfStepA[index];
        P3OUT = p3;
        P2OUT = p2;
        ++stepIndex;
        break;
    default: break;
    }

    // 如果在单步状态下，禁止定时器中断
    if( state & MOTION_MASK )
    {
        TACCTL0 &= ~CCIE;
    }
}
```

下面的定时器中断服务程序主要根据步进电机的工作状态，输出相应的控制序列给电机的驱动芯片。输出的序列是根据步进电机的状态进行查表得到的。

2. 看门狗中断服务程序

```
interrupt [WDT_VECTOR] void WDT_ISR(void)
{
    unsigned char sw_state;
    static unsigned char one_sec_flag = 0;

    // 获得 P1 口的输入
    sw_state = ~P1IN & 0x0f;

    // 判断是否有键按下
    if( sw_state == 0x00 )
    {
        // 禁止看门狗中断
        IE1 &= ~WDTIE;

        // 判断 S2 的激活状态是否小于 1 秒
        if( !one_sec_flag && (SW[1] >= DEBOUNCE_CNT) )
        {
            toggle_motion();
        }

        // 复位状态计数器
        SW[0] = 0;
        SW[1] = 0;
        SW[2] = 0;
        SW[3] = 0;

        // 复位标志
        one_sec_flag = 0;

        // 使能 P1 口的中断功能
        P1IFG = 0x00;
        P1IE = 0x0f;
    }
    else
    {
        // 检查是否是 S1 状态
        if( sw_state & 0x01 )
        {
            if( SW[0] < ONE_SEC_CNT )
            {
                // 增加状态计数器
                ++SW[0];
```

```
        }

        if( SW[0] == DEBOUNCE_CNT )
        {
            toggle_direction();
        }
    }
    else
    {
        SW[0] = 0;
    }

    // 检查是否是 S2 状态
    if( sw_state & 0x02 )
    {
        if( SW[1] < ONE_SEC_CNT )
        {
            // 增加状态计数器
            ++SW[1];
        }

        if( SW[1] == ONE_SEC_CNT )
        {
            toggle_stepping_mode();
            one_sec_flag = 1;
            SW[1] = 0;
        }
    }
    else
    {
        // 判断 S2 的激活状态是否小于 1 秒
        if( !one_sec_flag && (SW[1] >= DEBOUNCE_CNT) )
        {
            toggle_motion();
        }

        one_sec_flag = 0;
        SW[1] = 0;
    }

    // 检查是否是 S3 状态
    if( sw_state & 0x04 )
    {
        // 检查是否是连续模式
        if( (state & MOTION_MASK) == 0 )
        {
            if( SW[2] < ONE_SEC_CNT )
            {
```

```
                    // 增加状态计数器
                    ++SW[2];
                }

                if( SW[2] == DEBOUNCE_CNT )
                {
                    increase_stepping_rate();
                }
            }
            else // 单步模式
            {
                // 增加状态计数器
                ++SW[2];

                if( (SW[2] % DEBOUNCE_CNT) == 0 )
                {
                    increase_stepping_rate();
                }
            }
        }
        else
        {
            SW[2] = 0;
        }

        // 检查是否是 S4 状态
        if( sw_state & 0x08 )
        {
            if( SW[3] < ONE_SEC_CNT )
            {
                // 增加状态计数器
                ++SW[3];
            }

            if( SW[3] == DEBOUNCE_CNT )
            {
                decrease_stepping_rate();
            }
        }
        else
        {
            SW[3] = 0;
        }
    }
}
```

通过上面的两个中断服务程序就可以完成电机的控制功能。

在看门狗中断服务程序里，需要处理 P1 口的输入状态，看门狗中断是由 P1 口的中断服务程序来启动的。P1 口的中断服务程序如下：

```
interrupt [PORT1_VECTOR] void PORT1_ISR(void)
{
    // 禁止端口 1 的中断
    P1IE = 0x00;

    // 清除端口 1 的中断标志
    P1IFG = 0x00;

    // 使能看门狗中断
    IE1 |= WDTIE;
}
```

从 P1 口的中断服务程序可以看出，它只是清除中断标志，然后启动看门狗中断程序，并将具体的输入交给看门狗中断服务程序处理。

在上面的中断服务程序中，数据及状态信息的交互都是通过全局变量来实现的。另外，输出的电机控制信号是通过查表取得的。步进电机的状态表是根据步进电机的步进序列做成的，这里限于篇幅原因不再进行详细介绍，读者可以查阅步进电机的相关资料。

30.2.3 串口通信模块

串口通信模块主要是完成数据的发送和接收。在该软件系统中，采用中断服务程序实现串口通信，下面为具体的程序代码。

```
interrupt [UART0RX_VECTOR] void UART0_RX_ISR(void)
{
    //接收来自串口的数据
    UART0_RX_BUF[nRX0_Len_temp] = RXBUF0;

    nRX0_Len_temp += 1;

    if(nRX0_Len_temp >= 2)
    if(UART0_RX_BUF[nRX0_Len_temp - 2] == '\r' &&
       UART0_RX_BUF[nRX0_Len_temp - 1] == '\n')
    {
        // 过滤掉回车换行(\r\n)
        if(nRX0_Len_temp == 2)
        {
            nRX0_Len_temp = 0;
        }
        else if(nRX0_Len_temp > 2)
        {
```

```
            nRX0_Len = nRX0_Len_temp;
            nRev_UART0 = 1;
            nRX0_Len_temp = 0;
        }
    }

}
interrupt [UART0TX_VECTOR] void UART0_TX_ISR(void)
{
    if(nTX0_Len != 0)
    {
        // 表示缓冲区里的数据没有发送完
        nTX0_Flag = 0;

        TXBUF0 = UART0_TX_BUF[nSend_TX0];
        nSend_TX0 += 1;
        if(nSend_TX0 >= nTX0_Len)
        {
            nSend_TX0 = 0;
            nTX0_Len = 0;
            nTX0_Flag = 1;
        }
    }
}
```

在上面的中断服务程序中，主要通过全局变量和全局的缓冲区与主程序或者其他的模块进行数据交互。在接收程序里，一旦接收到回车换行，就设置全局变量“nRev_UART0”为“1”来通知其他程序已经接收到一帧数据。

30.2.4　主处理模块

主处理模块主要是接收来自 PC 机的命令信息，并给出响应。下面给出主程序的具体代码。

```
void main(void)
{
    // 停止WDT
    WDTCTL = WDTPW + WDTHOLD;
    sys_init();

    _EINT();

    for(;;)
    {
        if(nRev_UART1 == 1)
```

```
        {
            nRev_UART1 = 0;
            for(i = 0;i < nRX1_Len;i++) UART1_RX_Temp[i] = UART1_RX_BUF[i];
            if((UART1_RX_Temp[0] == 'A') && (UART1_RX_Temp[1] == 'T'))
            {
                UART0_TX_BUF[0] = 'O';
                UART0_TX_BUF[1] = 'K';
                UART0_TX_BUF[2] = 13;
                nTX0_Len = 3;
                switch(UART1_RX_Temp[2])
                {
                case 'D': // 方向
                    toggle_direction();
                    break;
                case 'C': // 运动模式
                    toggle_motion();
                    break;
                case 'M': // 步进模式
                    toggle_stepping_mode();
                    break;
                case 'F': // 增加速率
                    increase_stepping_rate();
                    break;
                case 'S': // 降低速率
                    decrease_stepping_rate();
                    break;
                default: break;
                }
            }
            else
            {
                UART0_TX_BUF[0] = 'E';
                UART0_TX_BUF[1] = 'R';
                UART0_TX_BUF[2] = 'O';
                UART0_TX_BUF[3] = 'R';
                UART0_TX_BUF[4] = 'R';
                UART0_TX_BUF[5] = 13;
                nTX0_Len = 6;
            }
            // 设置中断标志，进入发送中断程序
            IFG1 |= UTXIFG0;
        nRX1_Len = 0;
        }
    }
}
```

通过上面的程序可以看出，单片机主要接收来自 PC 机以“AT”开头的命令，如果命

令正确，则响应“OK”；如果命令错误，则响应“EORROR”。同时单片机根据接收到的命令进行相应的处理，比如改变方向、改变速率等操作。

30.3　实例总结

本章介绍了步进电机控制器的硬件和软件设计。通过本章的介绍，读者对步进电机的工作应该有一个初步的了解；读者也可以进一步加深对中断服务程序的认识，并且在此基础上了解单片机的多任务操作。虽然本章介绍的系统相对比较简单，但读者完全可以进一步改进和扩展硬件和软件功能，设计出满足自己系统需要的步进电机控制系统。为了能使读者全面了解程序的其他模块，附录中给出了程序中使用的步进电机的状态序列表，并且也给出了程序里所调用到的函数的程序源代码。

附录 A　其他程序模块

```
// 整步状态表 A
static const unsigned char fullStepA[] =
{
    0x00,
    0x00,
    0x01,
    0x01
};
//  半步状态表 B
static const unsigned char fullStepB[] =
{
    0x01,
    0x00,
    0x00,
    0x01
};
// 半步状态表 A
static const unsigned char HalfStepA[] =
{
    0x01,   // 001   1
    0x06,   // 110   2
    0x00,   // 000   3
    0x00,   // 000   4
    0x00,   // 000   5
    0x07,   // 111   6
    0x01,   // 001   7
    0x01    // 001   8
```

```
};
// 逆时钟、半步状态表 B
static const unsigned char CcwHalfStepB[] =
{
    0x01,   // 001   1
    0x01,   // 001   2
    0x01,   // 001   3
    0x06,   // 110   4
    0x00,   // 000   5
    0x00,   // 000   6
    0x00,   // 000   7
    0x07    // 111   8
};

// 顺时钟、半步状态表 B
static const unsigned char CwHalfStepB[] =
{
    0x00,   // 000   1
    0x00,   // 000   2
    0x00,   // 000   3
    0x07,   // 111   4
    0x01,   // 001   5
    0x01,   // 001   6
    0x01,   // 001   7
    0x06    // 110   8
};

void increase_stepping_rate(void)
{
    unsigned int new_rate;

    // 检查是否是连续模式
    if( (state & MOTION_MASK) == 0 )
    {
        new_rate = rate >> 1;

        if( new_rate >= max_rate )
        {
            rate = new_rate;
            change_rate_flag = 1;
        }
    }

    //使能定时器 A 的中断
    TACCTL0 |= CCIE;
}
void decrease_stepping_rate(void)
{
```

```
    // 检查是否是连续模式
    if( (state & MOTION_MASK) == 0 )
    {
        if( rate <= (min_rate >> 1) )
        {
            rate <<= 1;
            change_rate_flag = 1;
        }
    }

    // 使能定时器 A 的中断
    TACCTL0 |= CCIE;
}
void toggle_stepping_mode(void)
{
    // 切换步进模式
    state ^= STEP_MASK;

    // 检查是否是半步模式
    if( state & STEP_MASK )
    {
        // 从整步模式切换到半步模式
        // 定时器的频率加倍
        rate = (rate >> 1);
        change_rate_flag = 1;
        max_rate = (MAX_RATE >> 1);
        min_rate = (MIN_RATE >> 1);
    }
    else // 整步模式
    {
        // 从半步模式切换到整步模式
        // 定时器的频率减半
        rate = (rate << 1);
        change_rate_flag = 1;
        max_rate = MAX_RATE;
        min_rate = MIN_RATE;
    }
}

void toggle_motion(void)
{
    state ^= MOTION_MASK;

    // 检查是否是连续步进模式
    if( (state & MOTION_MASK) == 0 )
    {
        // 使能定时器中断
        TACCTL0 |= CCIE;
```

```
    }
}
void toggle_direction(void)
{
    state ^= DIR_MASK;
}
```

第 31 章

基于 MSP430 单片机实现的 CAN 通信系统

在工业控制中，CAN 总线技术是解决工业控制现场数据通信的最佳方案。本章介绍采用 MSP430 单片机实现的 CAN 通信系统，由于 MSP430 单片机不带 CAN 通信接口，因此需要外加 CAN 驱动芯片。本章首先介绍 CAN 通信系统的硬件设计，然后介绍它的软件设计。

31.1 硬件设计

CAN 总线是一种串行数据通信协议，其通信接口中集成了 CAN 协议的物理层和数据链路层功能，可完成对数据的成帧处理。CAN 总线支持全双工通信。CAN 协议的一个最大特点是废除了传统的站地址编码，而代之以对通信数据块编码。采用这种编码方法可使网络内节点个数在理论上不受限制，还可使不同的节点同时收到相同的数据。

CAN 总线能有效支持分布式控制或实时控制的串行通信网络，它可以将连接在总线上的设备连接成网络系统，从而实现基本的控制、传输等功能。如图 31-1 所示为 CAN 总线基本结构图。

在图 31-1 中，CAN 节点处理数据链路层的功能，CAN 收发器处理物理层功能。关于 CAN 总线协议的具体规范限于篇幅原因不进行介绍，读者可以参看相关的协议规范。下面介绍具体的芯片。

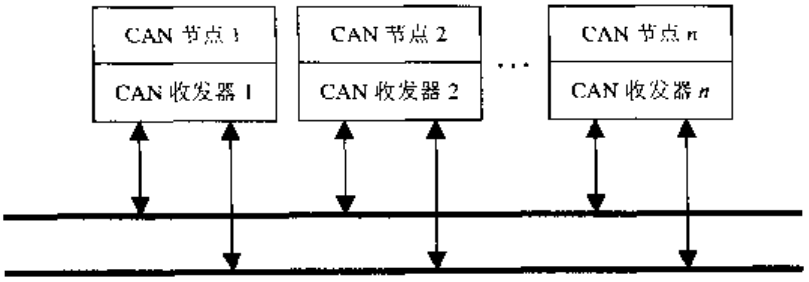


图 31-1　CAN 总线基本结构图

31.1.1　MCP2510 芯片

本章使用的 CAN 控制器为 MCP2510 芯片。MCP2510 是由 Microchip 公司生产的一款 CAN 协议控制器，该芯片完全支持 CAN 总线的 V2.0A/B 技术规范。MCP2510 芯片支持 CAN1.2、CAN2.0A、主动和被动 CAN2.0B 等版本的协议，能够发送和接收标准及扩展报文；还同时具备验收过滤及报文管理功能。该芯片包含三个发送缓冲器和两个接收缓冲器，减少了单片机的管理负担。该芯片与单片机的接口是通过标准串行外设接口（SPI）来实现的，其数据传输速率高达 5 Mbps。如图 31-2 所示为 MCP2510 芯片的内部框图。

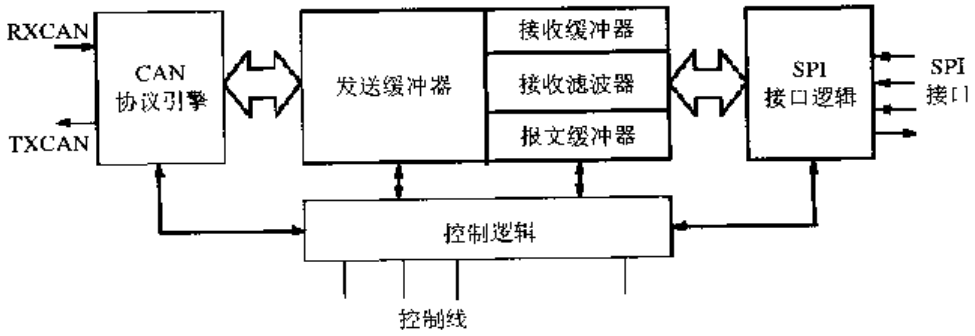


图 31-2　MCP2510 芯片内部框图

为了便于硬件设计，下面给出 MCP2510 芯片的管脚示意图，如图 31-3 所示。

由图 31-3 可以看出，该芯片有 18 个管脚，下面对每个管脚进行简单介绍。

- TXCAN：连接到 CAN 总线的发送输出引脚。
- RXCAN：连接到 CAN 总线的接收输入引脚。
- $\overline{\text{TX0RTS}}$：发送缓冲器 TXB0 请求发送或通用数字输入引脚。内部上拉电阻。
- $\overline{\text{TX1RTS}}$：发送缓冲器 TXB1 请求发送或通用数字输入引脚。内部上拉电阻。
- $\overline{\text{TX2RTS}}$：发送缓冲器 TXB2 请求发送或通用数字输入引脚。内部上拉电阻。

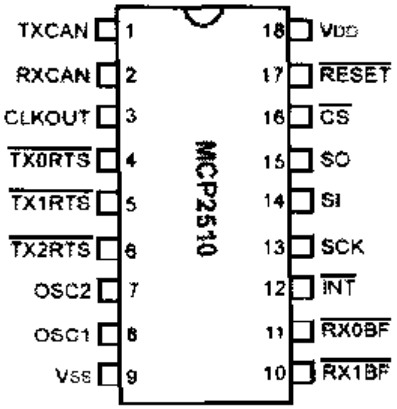


图 31-3　MCP2510 芯片管脚示意图

- $\overline{\text{RX1BF}}$：接收缓冲器 RXB1 的中断引脚或通用数字输出引脚。
- $\overline{\text{RX0BF}}$：接收缓冲器 RXB0 的中断引脚或通用数字输出引脚。
- SCK：SPI 接口时钟输入引脚。
- SI：SPI 接口数据输入引脚。
- SO：SPI 接口数据输出引脚。
- $\overline{\text{CS}}$：SPI 接口片选输入引脚。
- CLKOUT：带可编程预分频器的时钟输出引脚。
- OSC2：振荡器输出。
- OSC1：振荡器输入。
- Vss：接地管脚。
- $\overline{\text{INT}}$：中断输出引脚。
- $\overline{\text{RESET}}$：低电平有效器件复位输入引脚。
- V_{DD}：电源管脚。

经过对 MCP2510 芯片的简单介绍，对该芯片有了基本的认识，下面介绍硬件电路的具体设计。

31.1.2　硬件接口电路设计

硬件接口电路主要是 MCP2510 芯片和 MSP430 单片机通过 SPI 串口进行连接。如图 31-4 所示为具体的硬件接口电路图。

由图 31-4 可以看出，MCP2510 芯片的 OSC2 和 OSC1 外接晶体构成振荡电路，为 MCP2510 芯片提供工作时钟；MCP2510 芯片的 SPI 串口与 MSP430 单片机的 SPI 串口进

行连接，使用的是 SPI 口的 3 线方式，即分别与单片机的 P3.1、P3.2 和 P3.3 进行连接；MCP2510 芯片的片选管脚与单片机的 P1.3 管脚进行连接；MCP2510 芯片的 10、11 和 12 管脚分别与单片机的 P1.2、P1.1 和 P1.0 管脚进行连接，作为可选中断输入，在连接时需要将这 3 个管脚拉高；MCP2510 芯片的管脚 1 和管脚 2 分别为 CAN 控制器的输入与输出，这两个管脚直接与 CAN 收发器进行连接。由于 CAN 收发器电路非常简单，这里就不再讨论了。

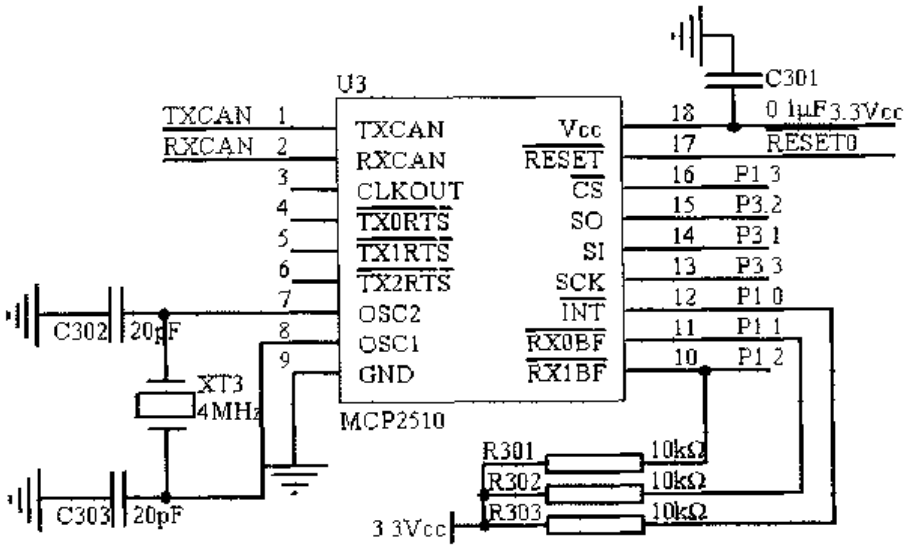


图 31-4　硬件接口电路图

31.2　软件设计

系统软件设计主要是完成单片机与 CAN 控制器芯片 MCP2510 之间的通信，从而实现数据的传输。在介绍具体程序实现前，先对 MCP2510 芯片的操作进行简单的介绍。

31.2.1　MCP2510 芯片操作

MCP2510 主要有 5 种操作模式，即配置模式、正常模式、休眠模式、监听模式和环回模式。在正常运行之前，必须对 MCP2510 进行初始化；只有在配置模式下，才能对其进行初始化。在初始上电或复位时，MCP2510 自动进入配置模式；在完成配置后就可以进入正常模式进行数据的收发了。在正常模式下，MCP2510 主动监视总线上的所有报文，并产生确认位和错误帧等。

对 MCP2510 模式的设置是通过操作 MCP2510 的寄存器来实现的。MCP2510 共有 128

个寄存器，操作相应的寄存器就可以实现对 MCP2510 的相应操作。单片机通过 SPI 口就可以操作 MCP2510 的不同寄存器。单片机在操作 MCP2510 时，需要向 MCP2510 发送不同的命令。如表 31-1 中所示为 MCP2510 的 SPI 口指令。

表 31-1　MCP2510 的 SPI 口指令

指　令	编　码	功 能 描 述
RESET	11000000	复位操作，并设置成配置模式
READ	00000011	从指定地址开始的寄存器中读取数据
WRITE	00000010	向指定地址开始的寄存器中写入数据
RTS	10000nnn	请求发送指令
READ STATUS	10100000	读取状态（包括各种请求和中断标志位等）
BIT MODIFY	00000101	对指定的寄存器进行位修改

单片机通过表 31-1 中的不同命令就可以实现对 MCP2510 的不同操作。

31.2.2　SPI 数据传输模块的实现

SPI 数据传输模块主要负责单片机与 MCP2510 之间的数据传输，该模块是为 MCP2510 操作模块服务的。下面为具体的程序代码。

```
void Init_Port(void)
{
    //将 P3 口所有的管脚在初始化的时候设置为输入方式
    P3DIR = 0;
    //将 P3 口所有的管脚设置为一般 I/O 口
    P3SEL = 0;
    //P3.1、P3.2、P3.3 被分配为 SPI 口
    P3SEL = BIT3 + BIT2 + BIT1;
    //P1.0 作为输出管脚
    P1DIR |= BIT0;
    //P3.3 作为输出管脚
    P3DIR |= BIT3;
    //P3.1 作为输出管脚
    P3DIR |= BIT1;
    //P1.0 输出高电平，MCP2510 不被选通
    P1OUT |= BIT0;
    return;
}
void Init_SPI (void)
{
    //SPI0 模块允许
    ME1 |= USPIE0;
    //将寄存器的内容清零
```

```
    U0CTL = 0X00;
    //数据为 8 比特，选择 SPI 模式，单片机为主机模式
    U0CTL |= CHAR + SYNC + MM;
    //将寄存器的内容清零
    U0TCTL = 0X00;
    // 时钟源为 SMCLK，选择 3 线模式
    U0TCTL = CKPH + SSEL1 + SSEL0 + STC;
    //传输时钟为 SMCLK / 4
    UBR0_0 = 0X02;
    UBR1_0 = 0X00;
    //调整寄存器，没有调整
    UMCTL_0 = 0X00;
    //发送接收中断允许
    IE1 |= UTXIE0;
    IE1 |= URXIE0;
}
```

上面的程序主要完成端口和 SPI 串口的初始化功能。

```
void CS_Enable(void)
{
    //P1.0 输出低电平
    P1OUT &=~(BIT0);
    return ;
}
void CS_Disable(void)
{
    //P1.0 输出高电平
    P1OUT|=BIT0;
    return ;
}
```

上面的程序主要是为 MCP2510 产生相应的片选信号。

在完成上面的处理基础上，MSP430 单片机就可以实现 SPI 串口数据的通信了。SPI 口数据传输采用中断程序来实现，具体程序代码如下：

```
interrupt [UART0TX_VECTOR] void UART0_TX_ISR(void)
{
    if(nTX0_Len != 0)
    {
        // 表示缓冲区里的数据没有发送完
        nTX0_Flag = 0;
        TXBUF0 = UART0_TX_BUF[nSend_TX0];
        nSend_TX0 += 1;
        if(nSend_TX0 >= nTX0_Len)
        {
            nSend_TX0 = 0;
            nTX0_Len = 0;
```

```
                nTX0_Flag = 1;
            }
        }
    }
```

上面的程序为串口发送中断程序。在上面的程序中，主要通过“nTX0_Len”、“nSend_TX0”、“nTX0_Flag”和“UART0_TX_BUF[]”全局变量实现与其他函数进行数据交互。

```
interrupt [UART0RX_VECTOR] void UART0_RX_ISR(void)
{
    //接收来自串口的数据
    UART0_RX_BUF[nRX0_Len_temp] = RXBUF0;

    nRX0_Len_temp += 1;
    if(nRX0_Len_temp >= nRX0_Len)
    {
        nRev_UART0 = 1;
        nRX0_Len_temp = 0;
        nRX0_Len = 0;
    }
}
```

上面的程序为串口接收中断程序。在上面的程序中，主要通过“nRX0_Len”、“nRev_UART0”和“UART0_RX_BUF[]”全局变量实现与其他函数进行数据交互。

31.2.3　MCP2510 操作模块的实现

MCP2510 操作模块主要是完成对 MCP2510 的读写操作，从而实现数据的传输。下面首先介绍基本操作函数，然后介绍复位、读状态寄存器、请求发送数据、位修改、读写数据操作等。具体程序如下：

```
void RessetMcp2510(void)
{
    int i;
    CS_Enable();
    for(i = 10;i > 0;i--) ;
    //复位命令
    UART0_TX_BUF[0] = (char)(0xC0);
    nTX0_Len = 1;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;
    for(i = 100;i > 0;i--) ;
    CS_Disable();
    return;
```

```
}
int GetStatusMcp2510(void)
{
    int i;
    int nStatus = 0;
    CS_Enable();
    for(i = 10;i > 0;i--) ;
    //读状态命令
    UART0_TX_BUF[0] = 0xA0;
    nTX0_Len = 1;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;

    nRX0_Len = 1;
    while(1)
    {
        if(nRev_UART0 == 1)
        {
            nRev_UART0 = 0;
            nStatus = (int)(UART0_RX_BUF[0]);
            break;
        }
    }
    for(i = 100;i > 0;i  ) ;
    CS_Disable();
    return nStatus;
}
void RtsMcp2510(char RTSn)
{
    int i;
    CS_Enable();
    for(i = 10;i > 0;i -) ;
    //请求发送命令
    UART0_TX_BUF[0] = (char)(0x80 | RTSn);
    nTX0_Len = 1;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;
    for(i = 100;i > 0;i--) ;
    CS_Disable();
    return;
}
void BitModiMcp2510(char addr,char mask,char data)
{
    int i;
    CS_Enable();
    for(i = 10;i > 0;i--) ;
    //位修改命令
    UART0_TX_BUF[0] = 0x05;
```

```
    UART0_TX_BUF[0] = addr;
    UART0_TX_BUF[0] = mask;
    UART0_TX_BUF[0] = data;
    nTX0_Len = 4;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;
    for(i = 100;i > 0;i--) ;
    CS_Disable();
    return;
}
void ReadMcp2510(int addr,int n,char outBuf[])
{
    int i;
    CS_Enable();
    for(i = 10;i > 0;i--) ;
    //读命令
    UART0_TX_BUF[0] = 0x03;
    UART0_TX_BUF[0] = addr;
    for(i = 0;i < n;i++)
    {
        UART0_TX_BUF[i + 2] = 0;
    }
    nTX0_Len = 2 + n;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;
    //接收数据
    nRX0_Len = n;
    while(1)
    {
        if(nRev_UART0 == 1)
        {
            nRev_UART0 = 0;
            for(i = 0;i < n;i++)
            {
                outBuf[i] = UART0_RX_BUF[i];
            }
            break;
        }
    }
    for(i = 100;i > 0;i--) ;
    CS_Disable();
    return;
}
void WriteMcp2510(int addr,int n,char inBuf[])
{
    int i;
    CS_Enable();
    for(i = 10;i > 0;i--) ;
```

```
    //写命令
    UART0_TX_BUF[0] = 0x02;
    UART0_TX_BUF[0] = addr;
    for(i = 0;i < n;i++)
    {
        UART0_TX_BUF[i + 2] = inBuf[i];
    }
    nTX0_Len = 2 + n;
    // 设置中断标志，进入发送中断程序
    IFG1 |= UTXIFG0;
    for(i = 100;i > 0;i--) ;
    CS_Disable();
    return;
}
```

在上面的几个基本操作基础上，可以完成模式设置及 MCP2510 的初始化操作。具体程序代码如下：

```
void SetNormal(void)
{
    int nFlag = 1;
    int i;
    char outBuf[10];
    for(i = 0;i < 10;i++)
    {
        outBuf[i] = 0;
    }
    //设置成正常模式
    BitModiMcp2510(CANSTAT,0xe0,0x00);
    do
    {
        ReadMcp2510(CANSTAT,1,outBuf);
        nFlag = outBuf[0] & 0xe0;
    }while(nFlag);

    return;
}
void IsSendComplete(int addr)
{
    int i;
    int nFlag = 1;
    char outBuf[10];
    for(i = 0;i < 10;i++)
    {
        outBuf[i] = 0;
    }

    do
```

```
    {
        ReadMcp2510(addr,1,outBuf);
        nFlag = outBuf[0] & 0x08;
    }while(nFlag);
    return;
}
void InitMcp2510()
{
    char inBuf[10];
    //复位
    ResetMcp2510();
    //设置波特率
    inBuf[0] = 0x02;
    inBuf[1] = 0x90;
    inBuf[2] = 0x07;
    WriteMcp2510(CNF3,3,inBuf);
    //RX0 接收，屏蔽位为 0，过滤器为 0
    inBuf[0] = 0x00;
    inBuf[1] = 0x00;
    WriteMcp2510(RXM0SIDH,2,inBuf);
    inBuf[0] = 0x00;
    inBuf[1] = 0x00;
    WriteMcp2510(RXF0SIDH,2,inBuf);
    //CAN 中断不使能
    inBuf[0] = 0x00;
    WriteMcp2510(CANINTE,1,inBuf);
    //设置成正常模式
    SetNormal();
}
```

在完成初始化配置后，就可以实现 CAN 通信了。下面为数据传输的具体程序代码。

```
void SendMsg(int nDLC,char inBuf[])
{
    int i;
    char sndBuf[16];
    for(i = 0;i < nDLC;i++)
    {
        sndBuf[i] = inBuf[i];
    }
    WriteMcp2510(TXB0D0,nDLC,sndBuf);
    sndBuf[0] = nDLC;
    WriteMcp2510(TXB0DLC,1,sndBuf);
    sndBuf[0] = 0x03;
    sndBuf[1] = nRecID_Hi;
    sndBuf[2] = nRecID_Lo;
    WriteMcp2510(TXB0CTRL,3,sndBuf);
    RtsMcp2510(0x01);
```

```
    IsSendComplete(TXB0CTRL);
    return;
}
int ReceiveMsg(char outBuf[])
{
    int i;
    int nFlag;
    int nDLC;
    char revBuf[16];
    for(i = 0;i < 16;i++)
    {
        revBuf[i] = 0;
    }
    ReadMcp2510(CANINTF,1,revBuf);
    nFlag = revBuf[0] & 0x01;
    if(nFlag == 0)
    {
        return 0;
    }
    BitModiMcp2510(CANINTF,0x01,0x00);
    ReadMcp2510(TXB0SIDH,2,revBuf);
    nRecID_Hi = revBuf[0];
    nRecID_Lo = revBuf[1] & 0xe0;
    ReadMcp2510(RXB0DLC,1,revBuf);
    nDLC = revBuf[0] & 0x0f;
    ReadMcp2510(RXB0D0,nDLC,outBuf);
    return 1;
}
```

31.3 实例总结

本章介绍了 MSP430 单片机与 CAN 控制器芯片 MCP2510 的接口设计，并给出了相应的程序。本章只是介绍了采用 CAN 总线技术实现的一个简单系统，MSP430 单片机和 MCP2510 的很多实用功能在本章中还没有进一步介绍，需要进一步完善。此外，由于 MCP2510 使用标准的 SPI 接口与单片机通信，所以硬件上很容易移植，只需要换上其他单片机就可以，并且只需对程序稍作修改即可。因此，采用该方案几乎可以在任何现有系统上实现 CAN 总线功能。