“十五”重点计算机普及出版物规划项目

单片机应用系统开发典型实例系列

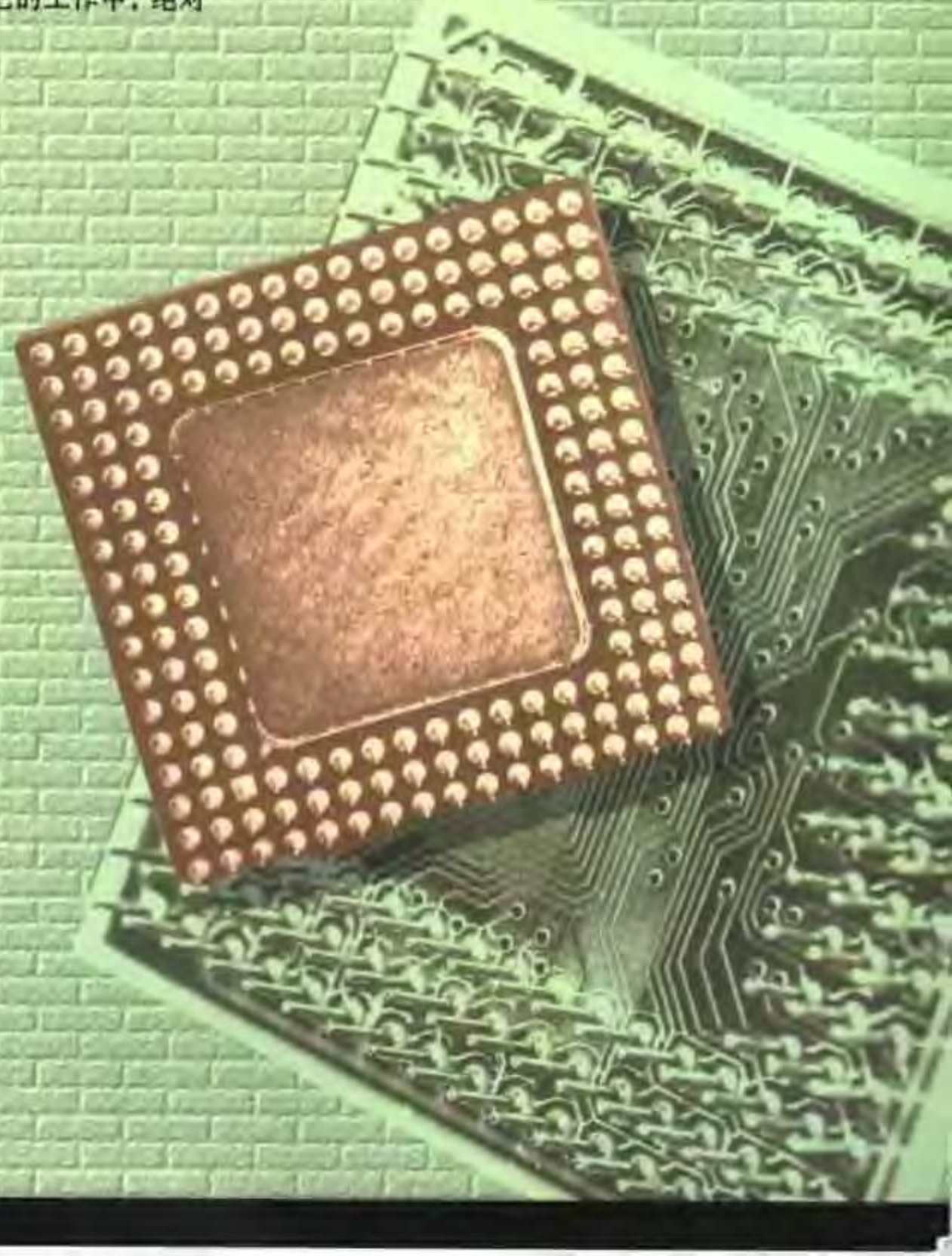
# MSP430单片机

## 应用系统开发典型实例

秦 龙 编著

随书附赠1CD，内含书中7个实例的电路图、C语言源程序，读者稍加修改，便可应用于自己的工作中，绝对物超所值！

- 温度采集报警系统
- MSP430F1XX 实现的数据采集系统
- 日历系统
- MODEM 数据传输的通信系统
- 大数据量本地存储系统
- 语音录放系统
- 短信息收发系统



中国电力出版社
www.infopower.com.cn

## 《研发电子丛书》火辣出击销售热点
## 引领嵌入式/单片机应用开发图书阅读新感觉!

本系列丛书分为嵌入式和单片机两个系列，覆盖了ARM编程、单片机编程、FPGA开发、DSP开发等多项电子方面的技术，填补了目前市场缺乏有深度的实例引导型图书的空白，受到了众多专家与学者的高度赞扬，华南理工大学电子与信息学院院长韦岗教授更是亲自为本丛书作序，大力向读者推荐。本丛书针对初、中级开发人员和相关专业学生，偏重于实用性，具有很强的工程实践指导性。书中所有的例子都是作者本人独自或主要负责完成的，调试通过并且部分进入商品化，读者稍为修改便可以直接应用于实际工作中，非常超值。

**《嵌入式应用系统开发典型实例系列》**

《ARM嵌入式应用系统开发典型实例》

《DSP嵌入式应用系统开发典型实例》

《FPGA嵌入式应用系统开发典型实例》

**《单片机应用系统开发典型实例系列》**

《PIC单片机应用系统开发典型实例》

《51单片机应用系统开发典型实例》

《AVR单片机应用系统开发典型实例》

《MSP430单片机应用系统开发典型实例》

**《嵌入式应用开发白金手册系列》**

《ARM嵌入式应用开发白金手册》

《CPLD/FPGA嵌入式应用开发白金手册》

- 高级研发工程师的倾情力作，凝聚了作者多年的开发经验与心得体会。
- 以典型实例开发为主线，详细介绍了MSP430单片机应用系统开发的流程、方法、技巧以及设计理念。
- 实例丰富，涉及不同的行业领域和方向，具有很强的工程性、实用性和指导性。
- 适合计算机、自动化、电子及硬件等相关专业的学生进行学习，同时也可供从事单片机开发的科研设计人员参考使用。

ISBN 7-5083-3275-X
9 787508 332758 >

策　　划/装红义
　　姚贵胜
责任编辑/李富颖
封面设计/王红柳

ISBN 7-5083-3275-X
定价：39.00元（含1CD）

“十五”重点计算机普及出版物规划项目

单片机应用系统开发典型实例系列

# MSP430单片机

## 应用系统开发典型实例

秦 龙 编著

中国电力出版社
www.infopower.com.cn

# 内 容 简 介

本书通过典型实例的形式，详细介绍了MSP430单片机应用系统开发的流程、方法、技巧以及设计理念。全书共分10章：第1章和第2章简单介绍了MSP430F1XX系列单片机的基础知识；第3章介绍了MSP430单片机开发的C语言基础；第4～10章为本书的重点，通过几个典型的实例，详细介绍了MSP430单片机的开发和使用技术，这些实例都具有典型代表性和应用广泛性，每个例子都有具体的硬件电路设计和程序设计。

全书语言简洁、层次清晰，适合计算机、自动化、电子及硬件等相关专业的学生进行学习，同时也可供从事单片机开发的科研设计人员参考使用。

图书在版编目（CIP）数据

MSP430单片机应用系统开发典型实例／秦龙编著. 北京：中国电力出版社，2005
（单片机应用系统开发典型实例系列）
ISBN 7-5083-3275-X

Ⅰ.M… Ⅱ.秦… Ⅲ.单片微型计算机，MSP430－系统开发 Ⅳ.TP368.1

中国版本图书馆CIP数据核字（2005）第028351号

## 版权声明



策　　划：裴红义
　　姚贵胜
责任编辑：李富颖
责任校对：崔燕菊
责任印制：李志强

丛 书 名：单片机应用系统开发典型实例系列
书　　名：MSP430单片机应用系统开发典型实例
编　　著：秦 龙
出版发行：中国电力出版社
　　地址：北京市三里河路6号　邮政编码：100044
　　电话：（010）88515918　传真：（010）88518169
印　　刷：利森达印务有限公司
开本尺寸：185 × 260　　印　张：21.75
书　　号：ISBN 7-5083-3275-X
版　　次：2005年7月北京第1版
印　　次：2005年7月第1次印刷
印　　数：1~5000
定　　价：39.00元

# 《单片机/嵌入式应用系统开发典型实例系列》
# 丛书编委会名单

# 丛 书 序

中国加入 WTO 之后，越来越多的国际著名 IT 企业都将生产部门转到中国，部分企业将研究、开发部门也逐步转到中国。同时，中国的企业也正越来越多地参与全球市场竞争。经济全球化越加剧，产品的竞争就越激烈，而产品的竞争最终是人才的竞争。中国能否培养出更多的优秀工程师，已经成为中国电子行业迅猛发展的一个必不可少的因素了。

市场决定技术的发展。在这样的环境下，电子类产品的开发已经成为当今的热点。本套丛书就是在这种条件下，为满足广大读者的需要应运而生的。

首先声明一点：下面的意见，仅仅是我个人对该套图书的内容与质量的理解和看法，其他读者完全可以在阅读本套丛书之后，提出不同的意见。

**1. 丛书覆盖范围**

本套丛书覆盖了 ARM 编程、FPGA 开发、DSP 开发、单片机编程、USB 接口等多种技术。

**2. 基本形式**

（1）内容结构：首先简要介绍了基础知识（例如硬件基本内部结构、开发工具和方法、基本指令、开发流程等），然后对应用系统项目开发实例进行了详细的讲解。

（2）表现形式：以技术性强的热门实例介绍为主线，全书基本遵照电子系统开发的基本步骤和思路进行详细讲解，讲解中穿插了经验、小技巧与注意事项。

**3. 实例的安排**

在本套丛书中，每本书都以案例为核心向读者介绍和传递相关的技术，所选用的大多数案例都具有代表性、技术领先性以及应用广泛性，是每一位作者多年开发经验的推广与总结。

每本书附带一张光盘，内容包括书上所介绍的案例的源程序和电路图。这样安排的目的是方便读者在实际工作中充分借鉴，进一步加深对该项电子技术的理解，提高读者应用开发的能力。

**4. 本书作者的优势**

本套丛书的作者全部都具有多年的电子产品开发和编程经验，有的在全国电子设计大赛中获过奖，在公司中担任项目开发部经理或技术骨干；有的是大学实验室的指导老师，从事过许多科研项目的设计、开发，在专业报刊上发表过许多学术论文，在学术和实际开发中都积累了很多经验。正是这些作者高水平的实际开发能力与丰富的经验积累，保证了本套丛书的质量。

### 5. 读者对象

本套丛书面向高校计算机、电子、自动化及相关硬件专业的在校大学生以及从事电子开发的科研人员。

科研人员通过学习，可以提高工作中的开发能力，解决和完善实际工作方案；对于在校大学生，光盘中附有丰富的实例硬件原图文件和程序源代码，只要稍加修改，便可应用于自己的学习中，或者完成自己的课题（毕业设计），物超所值。

### 6. 个人对本套丛书的期望和评价

本套丛书主要偏重于实用性，具有很强的工程实践指导性。所有的例子都是作者本人独自或主要负责完成调试通过并且大部分已进入商品化。衷心地希望本套丛书能够使广大读者受益非浅，并受到广大科研人员以及相关专业大学生的青睐。

韦　岗

2005 年 4 月

## 韦岗简历：

1963 年 1 月出生，现任华南理工大学电子与信息学院院长　历任副教授、教授、博士生导师（1996 年 5 月），享受国务院政府特殊津贴（1997 年）

长期从事电子信息领域的教学与研究。研究领域包括：数字无线通信、多媒体信息处理等。先后主持过国家部委、广东省、广州市及企业各级科研项目等 30 多项，获得国家专利 9 项，在国内外著名刊物上发表论文 50 多篇，包括 4 篇 IEEE 汇刊全文论文，被三大索引收录 30 多篇次。获得国家教委、广东省及广州市各级科研多项奖励。1999 年获广东省“五一”劳动奖章及首届广东省“五四”青年奖章。

担任职务：被聘为国家自然科学基金电子与信息学科评委、国际学术刊物“Real Time Systems”（美国）、国家一级学报《电子学报》、《通信学报》、“Control Theory and Applications”及《控制理论与应用》编委、中国电子学会集成电路系统设计委员会副主任等。

被聘为广东省电子政务专家组成员、广东省产业政策咨询委员会委员、广东省电子类正高职称评审委员会委员、广州市发展信息产业专家组组长、广州市天河软件园（国家火炬计划软件产业基地）专家组组长、广州市电子行业协会副会长。

# 前　言

MSP430F1XX 系列单片机是一种 16 位的单片机。它具有集成度高、外围设备丰富、超低功耗等优点，因此在除超低功耗外等许多领域内得到了广泛的应用。特别是它的超低功耗特性，是目前其他单片机不可比拟的。由于 MSP430F1XX 系列单片机的最高频率可以工作到 8MHz，指令执行的时间只需要 4 个机器周期，是 51 系列单片机远远达不到的，因此该系列单片机具有非常强的处理能力，最高可以达到 2MIPS，非常适合一些对处理要求比较高的嵌入式系统。

MSP430F1XX 系列单片机支持采用汇编语言和 C 语言进行开发。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 MSP430F1XX 系列单片机非常方便，而且适用于 MSP430F1XX 系列单片机的 C 语言与标准 C 语言兼容度高，IAR 公司提供的 Embedded Workbench 集成开发环境人机界面友好，能对 C 语言开发进行很好的支持，因此本书的程序都是采用 C 语言进行开发的。

本书主要通过典型实例的形式，详细介绍了 MSP430F1XX 系列单片机应用系统开发的流程、方法、技巧以及设计思想。本书共分为 10 章：第 1 章介绍了 MSP430F1XX 系列单片机的基本结构和管脚，读者通过学习将对 MSP430F1XX 系列单片机有个基本的认识；第 2 章介绍了单片机的 CPU 和外设，读者通过学习可以了解该系列单片机的硬件知识；第 3 章介绍了 MSP430 单片机开发的 C 语言基础，对开发 MSP430 系列单片机有着非常重要的作用。第 4 章～第 10 章重点介绍了一些具体的开发实例，包括开发步骤、方法、技巧与注意事项等，这些实例都具有代表性和应用广泛性，每个例子都有具体的硬件电路设计和程序设计，并且基本上每一章后面的附录里都给出了具体的程序代码，读者可以直接借鉴和使用。

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的框架，读者可以根据自己的情况，举一反三，丰富程序功能，以实现自己的更为完整的系统。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。通过对本书典型实例的学习，相信读者能循序渐进地掌握 MSP430F1XX 系列单片机的开发技术，并且能做到触类旁通，在自己的开发设计中进行灵活运用。

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 MSP430F1XX 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，取得过不错的成绩。另外，本

书作者先后在《声学学报》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。本书的实例是作者多年开发经验的推广与总结。

本书主要由秦龙编写，另外参与编写的人还有：王渝梅、张晓平、田莉、金成江、尹才华、钱林杰、刘轶、刘卓、徐桂生、穆雍、孟庆慈、李潇、王宁、张纪奎、麻晓波、黄华、屈秋林、唐清善、邱宝良、周克足、刘斌、李亚捷、李永怀、周卫东等，他们在资料收集与整理、硬件设计与程序调试及技术支持等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，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

2004 年 12 月

# 目　录

# 第 1 章　MSP430F1XX 单片机的介绍

本章主要介绍 MSP430F1XX 系列单片机的基本结构，并对各个单片机管脚的功能进行讲解，通过本章的介绍，读者将能对 MSP430F1XX 系列单片机有一个基本的认识。

## 1.1　概述

MSP430F1XX 系列单片机是一种超低功耗的混合信号控制器，它根据不同的应用提供不同的具体型号的单片机，以满足不同用户的需求。它们具有 16 位 RSIC 结构，CPU 中的 16 个寄存器和常数产生器使 MSP430 微控制器能达到最高的代码效率。单片机通过采用不同的时钟源工作可以使器件满足不同的功耗要求，适当选择时钟源，可以让器件的功耗达到最小，满足一些采用电池供电的系统。当器件处于低功耗的模式下，数字控制的振荡器（DCO）可以使器件从低功耗的模式下迅速唤醒，能够在少于 6μs 的时间内从低功耗模式转到激活工作模式。

MSP430F1XX 系列单片机具有丰富的外设，且功耗很低，有非常广阔的应用范围，它主要有以下特点。

- 低电压、超低功耗。MSP430F1XX 系列单片机在 1.8V～3.6V 的电压、1MHz 的时钟频率下运行，耗电电流在 0.1μA～400μA 之间，这个和不同的工作模式有关。MSP430F1XX 系列单片机有 16 个中断源，并且可以嵌套使用，使用中断请求将 CPU 从低功耗模式下唤醒只要 6μs 的时间，这样就可以编写出实时性很高的程序。根据具体的处理情况可以将 CPU 处于低功耗模式，在需要的时候通过中断来唤醒 CPU，从而实现系统的低功耗要求。
- 强大的处理能力。MSP430F1XX 系列单片机为 16 位的 RSIC 结构，具有丰富的寻址方式、简洁的指令、大量的寄存器以及片内的数据存储器都可以参加多种运算，还有高效的查表处理方法，有较高的处理速度，在 8MHz 晶体下运算能力达到 1MIPS（每秒种运算 100 万条指令），是传统 51 单片机远远达不到的。这些特点使该系列单片机采用 C 语言开发仍能有很高的效率，从而可以提高开发的周期，也可以实现程序的可移植性。
- 系统工作稳定。MSP430F1XX 系列单片机在上电复位后，首先由 DCOCLK 启动 CPU，保证程序从正确的位置开始执行，同时也保证晶体振荡器有足够的起振及稳定时间。在完成上述工作后，软件可以设置特定的寄存器的控制位来确定最后的系统工作时钟频率。在 CPU 运行中，如果 MCLK 发生故障，DCO 会自动启动，以保证系统正常工作，如果程序出错的话，可以通过设置看门狗来解决。在程序跑飞的时候，看门狗会出现溢出的情况，这时看门狗产生复位信号，使系统重新启动，从而保证系统运行的稳定性。

- 丰富的外设资源。MSP430F1XX 系列单片机根据不同型号提供了不同的外设资源，主要的外设资源有定时器、看门狗、比较器、串口、硬件乘法器、ADC 模块和丰富的端口资源。MSP430F1XX 系列单片机的定时器具有捕获/比较功能，可以用于事件记数、时序产生、PWM 波形产生等。看门狗可以在程序跑飞的时候重新启动系统，保证系统的稳定运行。比较器可以进行模拟电压的比较，与定时器结合使用可以设计成 A/D 转换器。串口资源可以实现多机通信。硬件乘法器增强了单片机的运算处理能力。集成 ADC 模块可以满足大多数的数据采集应用场合，这样也可以减小系统设计的复杂度，同时减小 PCB 版的面积。丰富的端口资源使单片机具有更加丰富的接口功能，并且该系列单片机的某些端口还具有中断功能，进一步丰富了中断资源，也更加有利于写多任务操作的程序。由于 MSP430F1XX 系列单片机有如此丰富的外设资源，这样就提供了更多的单片机解决方案。
- 方便的调试功能。由于目前的 MSP430F1XX 系列单片机一般是基于 FLASH 型的，这样单片机可以实现写入和擦除，加上 MSP430F1XX 系列单片机提供了 JTAG 口，这样单片机就能实现很好的在线调试仿真功能。通过集成的 IDE 开发环境，使用户很容易调试程序。开发工具能很好支持 C 语言开发，这样能缩短程序开发的时间，也保证程序的可移植性。
- 代码保护功能。虽然 MSP430F1XX 系列单片机基本上是 FLASH 型的，但该系列单片机具有代码保护功能，通过使用代码保护技术，就可以防止程序被读出来进行拷贝，从而起到保护知识产权的作用。

经过这部分介绍，让读者了解了 MSP430F1XX 系列单片机的基本特色，下面将简单介绍该系列单片机的硬件结构。

## 1.2 MSP430F1XX 单片机的结构

MSP430F1XX 系列单片机主要包括 MSP430F11X 系列、MSP430F12X 系列、MSP430F13X 系列、MSP430F14X 系列、MSP430F15X 系列和 MSP430F16X 系列。下面就各个系列进行具体的介绍。

### 1.2.1 MSP430F11X 系列单片机

该系列的单片机主要有 MSP430F1101A、MSP430C1101、MSP430F1111A、MSP430C-1111、MSP430F1121A、MSP430F1122、MSP430C1122、MSP430F1132 和 MSP430C1132 等几种单片机，其中含有字母“C”为 ROM 类型的，含有字母“F”为 FLASH 类型的。该系列单片机主要有以下特点。

- 具有很低的供电电压。单片机的供电电压最低可以低到 1.8V，单片机的供电电压范围是 1.8V～3.6V。
- 超低的功耗。这是目前其他单片机没有的特色。它在休眠的条件下工作的电流只有 0.8μA，就是在 2.2V、1MHz 条件下工作的电流只有 200μA。
- 快速的唤醒时间。从休眠方式唤醒只需要 6μs 的时间。
- 快速的指令执行时间。它采用的是 16 位的 RISC 结构，指令的执行时间只需要 150ns

的时间，是传统单片机不能比拟的。

- 具有灵活的时钟设置。主要有 6 种方式：可变的内部电阻设置方式、单个外部电阻方式、32kHz 的晶体方式、高频率晶体方式、谐振器方式和外部时钟源方式。这样可以根据功耗要求和速度要求进行灵活的时钟设置。
- 16 位的定时器 Timer_A 带有 3 个捕获/比较寄存器。
- 代码保护功能。单片机的安全熔丝能对程序的代码进行保护，从而可以对知识产权进行保护。
- 具有 JTAG 仿真调试接口，这样非常便于软件的调试。

为了对 MSP430F11X 系列有比较全面清楚的认识，下面给出了该系列单片机的结构框图，如图 1-1 所示。

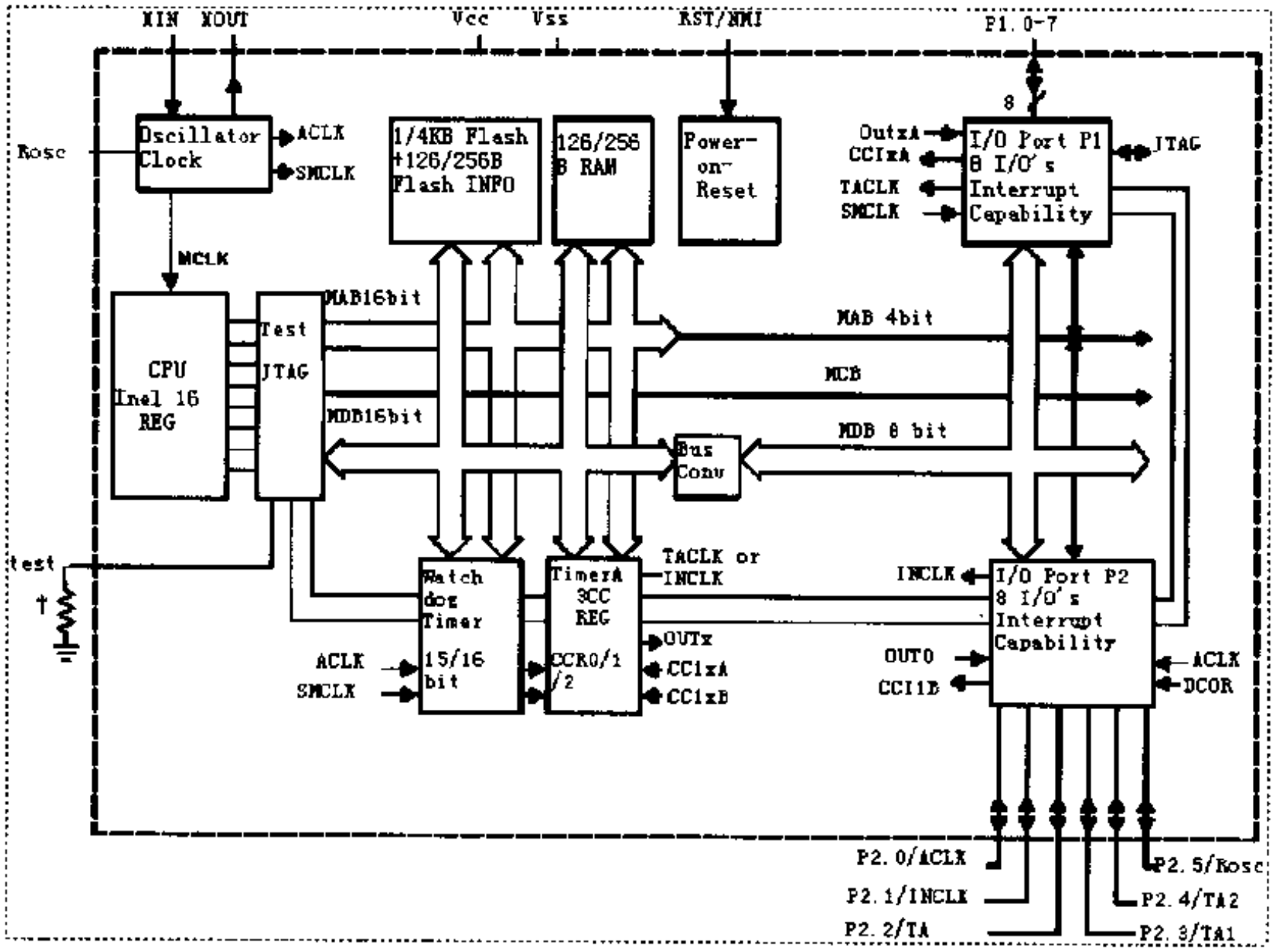



图 1-1　MSP430F11X 系列单片机的结构框图

通过图 1-1 能大致了解该系列单片机的各个功能模块。为了让读者对该系列单片机有进一步的认识，在此特意介绍一下该系列单片机各个管脚。图 1-2 为该系列单片机的管脚图。该图所示的只是该系列单片机的基本型号，也有可能在具体不同型号时有一点小的差别，详细介绍请参看具体型号单片机的数据手册。

下面具体介绍 MSP430F11X 单片机的各个管脚的功能，使读者能够通过了解单片机的管脚功能来完成单片机的硬件设计。对于硬件设计来说，这是必须的而且也是非常重要的。

- TEST：用于端口 1 的 JTAG 管脚的测试方式选择。
- $V_{CC}$：电源端。

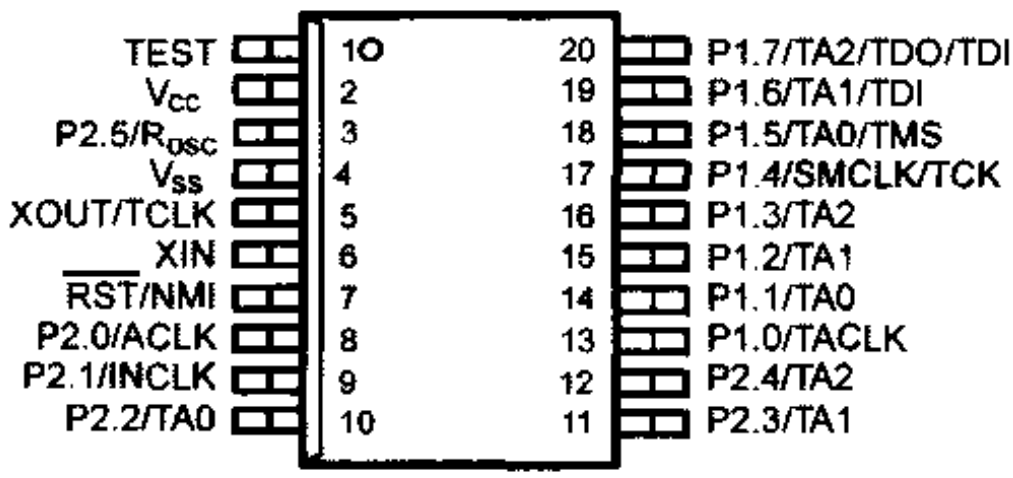



图 1-2　MSP430F11X 系列单片机的管脚图

- P2.5/$R_{OSC}$：通用数字 I/O 管脚/作为外接电阻管脚，通过接一电阻来确定 DCO 的工作频率。
- $V_{SS}$：电源地。
- XOUT/TCLK：晶体振荡器输出端 / 测试时钟输入端。
- XIN：晶体振荡器连接端。
- $\overline{RST}$/NMI：复位信号输入端 / 不可屏蔽中断输入端。
- P2.0/ACLK：通用数字 I/O 管脚 / ACLK 输出端。
- P2.1/INCLK：通用数字 I/O 管脚 / Timer_A，INCLK 时钟信号。
- P2.2/TA0：通用数字 I/O 管脚 / Timer_A，捕获：CCI0B 输入，比较：OUT0 输出。
- P2.3/TA1：通用数字 I/O 管脚 / Timer_A，捕获：CCI0B 输入，比较：OUT1 输出。
- P2.4/TA2：通用数字 I/O 管脚 / Timer_A，比较：OUT2 输出。
- P1.0/TACLK：通用数字 I/O 管脚 / Timer_A，TACLK 时钟输入信号。
- P1.1/TA0：通用数字 I/O 管脚 / Timer_A，捕获：CCI0A 输入，比较：OUT0 输出。
- P1.2/TA1：通用数字 I/O 管脚 / Timer_A，捕获：CCI1A 输入，比较：OUT1 输出。
- P1.3/TA2：通用数字 I/O 管脚 / Timer_A，捕获：CCI2A 输入，比较：OUT2 输出。
- P1.4/SMCLK/TCK：通用数字 I/O 管脚 / SMCLK 信号输出 / 测试时钟，用于器件编程和测试时的时钟输入端。
- P1.5/TA0/TMS：通用数字 I/O 管脚 / Timer_A，比较：OUT0 输出 / 测试方式选择，器件编程和测试输入端。
- P1.6/TA1/TDI：通用数字 I/O 管脚 / Timer_A，比较：OUT1 输出 / 测试数据输入端。
- P1.7/TA2/TDO/TDI：通用数字 I/O 管脚 / Timer_A，比较：OUT2 输出 / 测试数据输出端 / 编程时数据输入端。

### 1.2.2　MSP430F12X 系列单片机

该系列的单片机主要有 MSP430F122、MSP430F123、MSP430F1222 和 MSP430F1232 等几种型号。该系列单片机主要有以下特点。

- 具有很低的供电电压。单片机的供电电压最低可以低到 1.8V，单片机的供电电压范围是：1.8V～3.6V。
- 超低的功耗。这是目前其他单片机没有的特色。它在休眠的条件下工作的电流只有 0.8μA，就是在 2.2V、1MHz 条件下工作的电流只有 200μA。

- 快速的唤醒时间。从休眠方式唤醒只需要 6μs。
- 快速的指令执行时间。它采用的是 16 位的 RISC 结构，指令的执行时间只需要 150ns，是传统单片机不能比拟的。
- 具有灵活的时钟设置。主要有 6 种方式：可变的内部电阻设置方式、单个外部电阻方式、32kHz 的晶体方式、高频率晶体方式、谐振器方式和外部时钟源方式。这样可以根据功耗要求和速度要求进行灵活的时钟设置。
- 16 位的定时器 Timer_A 带有 3 个捕获/比较寄存器。
- 片内模拟信号比较器或者单斜边 A/D 转换器。
- 串口通信模块：USART0。该串口可以通过软件选择设置成 UART 方式或者 SPI 方式。
- 与 MSP430F11X 系列单片机相比，该系列单片机具有更多的 I/O 口，因此外围资源更加丰富。
- 有 5 种节能模式。
- 代码保护功能。单片机的安全熔丝能对程序的代码进行保护，从而可以对知识产权进行保护。
- 具有 JTAG 仿真调试接口，这样非常便于软件的调试。

为了对 MSP430F12X 系列有比较全面清楚的认识，下面给出了该系列单片机的结构框图，如图 1-3 所示。

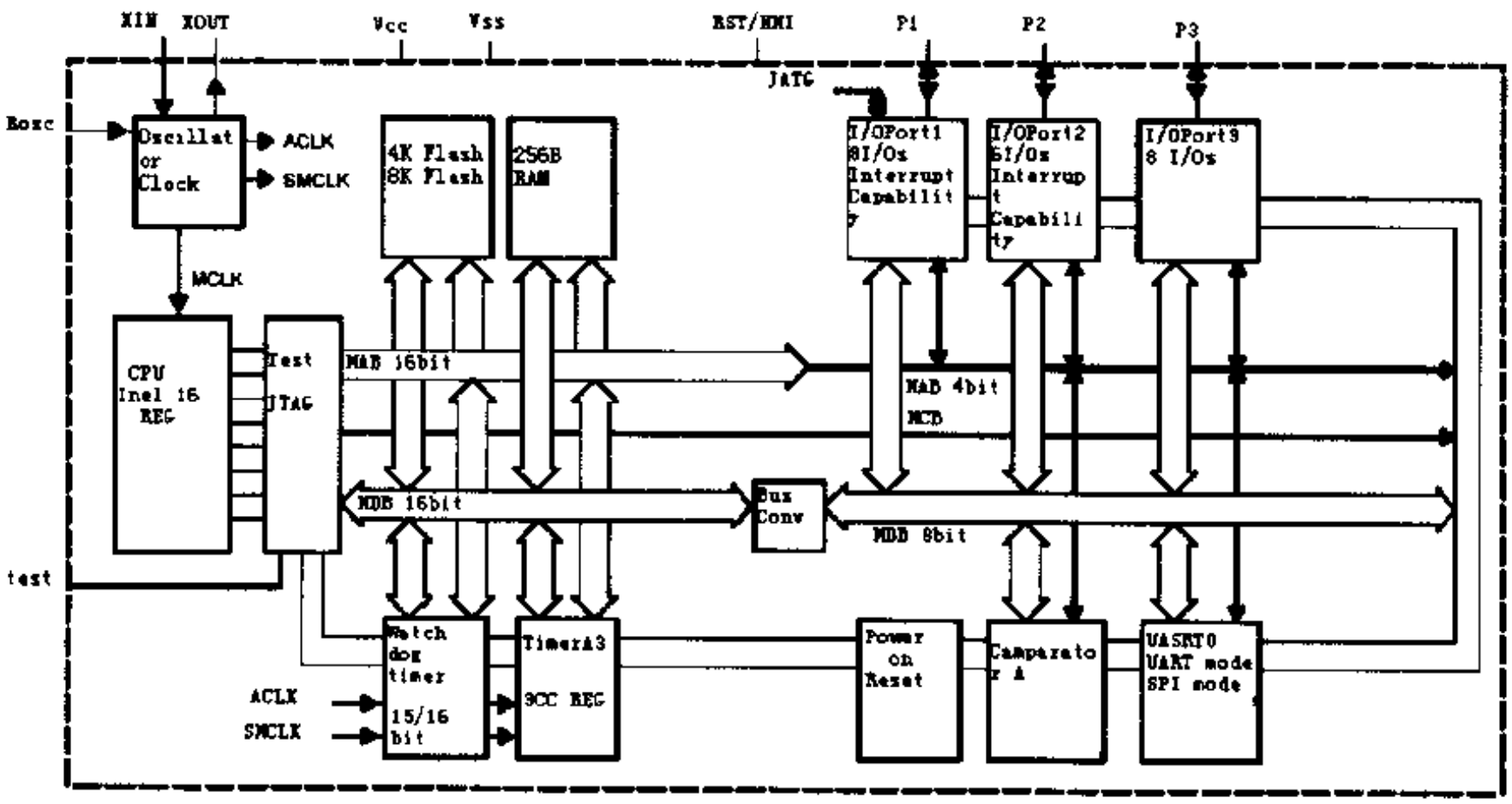



图 1-3　MSP430F12X 系列单片机的结构框图

通过图 1-3 能大致了解该系列单片机的各个功能模块。为了能让读者对该系列单片机有进一步的认识，在此特意介绍一下该系列单片机各个的管脚。图 1-4 为该系列单片机的管脚图。该图只是该系列单片机的基本型号的图，也有可能在具体不同型号时有一点小的差别，详细介绍请参看具体型号单片机的数据手册。

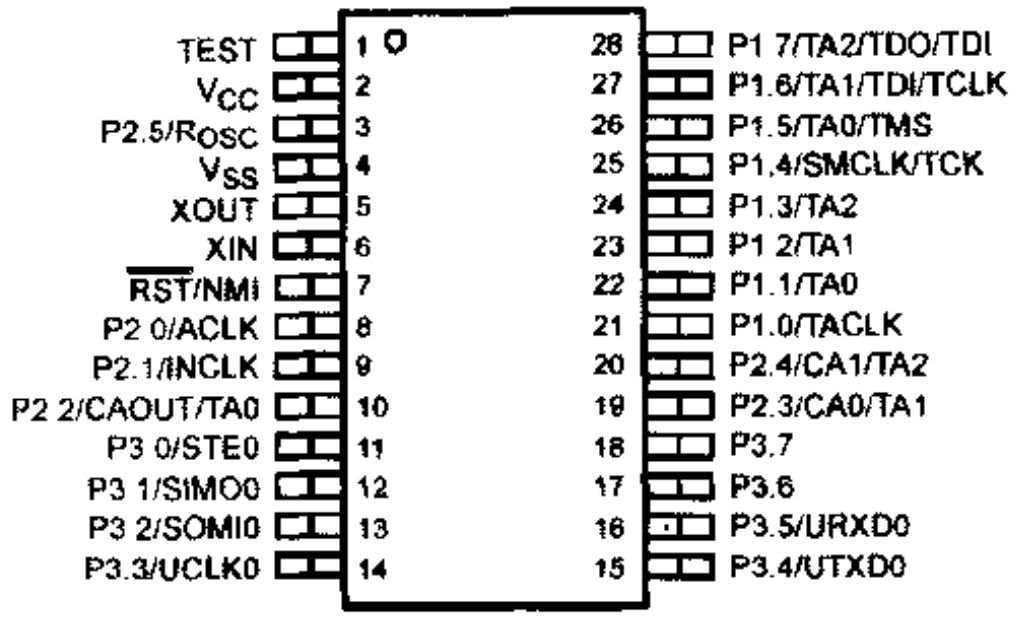



图 1-4　MSP430F12X 系列单片机的管脚图

下面具体介绍 MSP430F12X 系列单片机的各个管脚的功能，使读者能够通过了解单片机的管脚功能来完成单片机的硬件设计，对于硬件设计来说，这是必须的而且也是非常重要的。

- TEST：用于端口 1 的 JTAG 管脚的测试方式选择。
- $V_{CC}$：电源端。
- P2.5/ Rosc：通用数字 I/O 管脚 / 作为外接电阻管脚，通过接一电阻来确定 DCO 的工作频率。
- $V_{SS}$：电源地。
- XGUT/TCLK：晶体振荡器输出端 / 测试时钟输入端。
- XIN：晶体振荡器连接端。
- $\overline{\text{RST}}$/NMI：复位信号输入端 / 不可屏蔽中断输入端。
- P2.0/ACLK：通用数字 I/O 管脚 / ACLK 输出端。
- P2.1/INCLK：通用数字 I/O 管脚 / Timer_A，INCLK 时钟信号。
- P2.2/TA0：通用数字 I/O 管脚 / Timer_A，捕获：CCI0B 输入，比较：OUT0 输出。
- P3.0/STE0：通用数字 I/O 管脚 / 从传输使能：USART0 / SPI 模式。
- P3.1/SIMO0：通用数字 I/O 管脚 / USART0 / SPI 模式下的从输入或者主输出。
- P3.2/SOMI0：通用数字 I/O 管脚 / USART0 / SPI 模式下的从输出或者主输入。
- P3.3/UCLK0：通用数字 I/O 管脚 / 外部时钟输入——USART0 / UART 或 SPI 模式，时钟输出——USART0 / SPI 模式。
- P3.4/UTXD0：通用数字 I/O 管脚 / 发送数据输出——USART0 / SPI 模式。
- P3.5/URXD0：通用数字 I/G 管脚 / 发送数据输入——USART0 / SPI 模式。
- P3.6：通用数字 I/O 管脚。
- P3.7：通用数字 I/O 管脚。
- P2.3/CA0/TA1：通用数字 I/O 管脚 / Timer_A，捕获：CCI0B 输入，比较：OUT1 输出。
- P2.4/CA1/TA2：通用数字 I/O 管脚 / Timer_A，比较：OUT2 输出。
- P1.0/TACLK：通用数字 I/G 管脚 / Timer_A，TACLK 时钟输入信号。
- P1.1/TA0：通用数字 I/O 管脚 / Timer_A，捕获：CCI0A 输入，比较：OUT0 输出。
- P1.2/TA1：通用数字 I/O 管脚 / Timer_A，捕获：CCI1A 输入，比较：OUT1 输出。
- P1.3/TA2：通用数字 I/O 管脚 / Timer_A，捕获：CCI2A 输入，比较：OUT2 输出。

- P1.4/SMCLK/TCK：通用数字 I/O 管脚 / SMCLK 信号输出 / 测试时钟，用于器件编程和测试时的时钟输入端。
- P1.5/TA0/TMS：通用数字 I/O 管脚 / Timer_A，比较：OUT0 输出 / 测试方式选择，器件编程和测试输入端。
- P1.6/TA1/TDI/TCLK：通用数字 I/O 管脚 / Timer_A，比较：OUT1 输出 / 测试数据输入端。
- P1.7/TA2/TDO/TDI：通用数字 I/O 管脚 / Timer_A，比较：OUT2 输出 / 测试数据输出端 / 编程时数据输入端。

### 1.2.3　MSP430F13X 系列单片机

该系列的单片机主要有MSP430F133、MSP430C1331、MSP430F135和MSP430C1351等几种型号，其中含有字母“C”为ROM类型的，含有字母“F”为FLASH类型的。该系列单片机主要有以下特点。

- 具有很低的供电电压。单片机的供电电压最低可以低到 1.8V，单片机的供电电压范围是：1.8V～3.6V。
- 超低的功耗。这是目前其他单片机没有的特色。它在休眠的条件下工作的电流只有 0.8μA，就是在 2.2V、1MHz 条件下工作的电流只有 280μA。
- 快速的唤醒时间。从休眠方式唤醒只需要 6μs。
- 快速的指令执行时间。它采用的是 16 位的 RISC 结构，指令的执行时间只需要 150ns，是传统单片机不能比拟的。
- 片内有 12 位的 A/D 转换器，片内提供参考电压。A/D 转换器具有采样保持和自动扫描等特点。
- 16 位的定时器 Timer_B 带有 7 个捕获/比较寄存器。
- 片内提供温度传感器。
- 具有灵活的时钟设置。主要有以下几种方式：32kHz 的晶体方式、高频率晶体方式、谐振器方式和外部时钟源方式。这样可以根据功耗要求和速度要求进行灵活的时钟设置。
- 16 位的定时器 Timer_A 带有 3 个捕获/比较寄存器。
- 片内提供模拟信号比较器。
- 串口通信模块：USART0。该串口可以通过软件选择设置成 UART 方式或者 SPI 方式。
- 片内提供较多的存储器，MSP430F133 提供的片内的 FLASH 为 8KB，MSP430F135 提供的片内的 FLASH 为 16KB，同时片内还提供较多的 RAM，以便进行运算处理。
- 提供 P1.0～P6.0 共 6 个数据端口，能为用户提供更多的处理功能。在提供的外围数据端口中，有两个端口具有中断功能，这样能丰富硬件系统的中断资源，也为实现多任务系统提供方便。
- 代码保护功能。单片机的安全熔丝能对程序的代码进行保护，从而可以对知识产权进行保护。
- 具有 JTAO 仿真调试接口，这样非常便于软件的调试。

为了对 MSP430F13X 系列有比较全面清楚的认识，下面给出了该系列单片机的结构框

图，如图 1-5 所示。

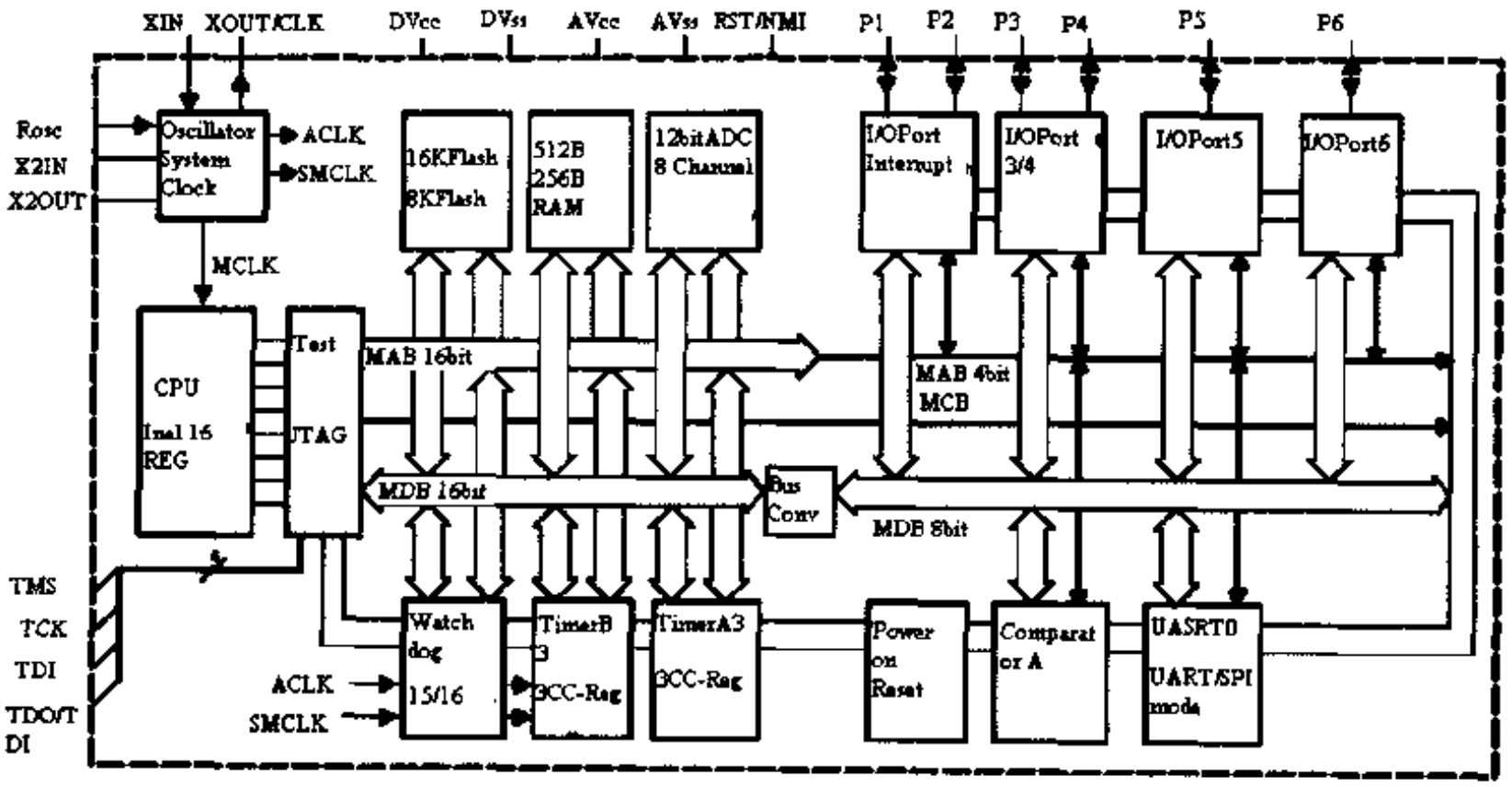



图 1-5　MSP430F13X 系列单片机的结构框图

通过图 1-5 能大致了解该系列单片机的各个功能模块。为了让读者对该系列单片机有进一步的认识，在此特意介绍一下该系列单片机的各个管脚。图 1-6 为该系列单片机的管脚图。该图所示的只是该系列单片机的基本型号，也有可能在具体不同型号时有一点小的差别，详细介绍请参看具体型号单片机的数据手册。

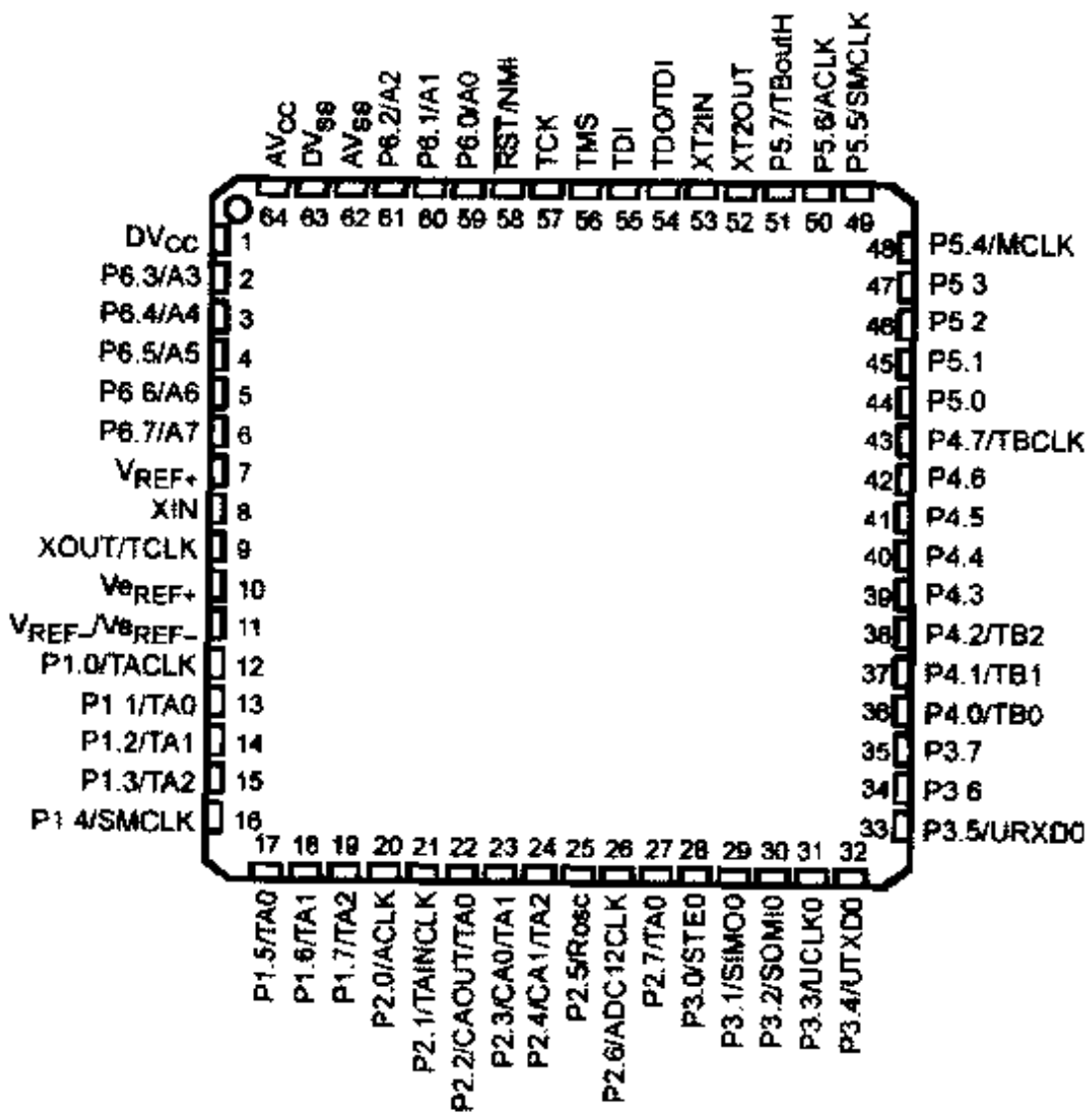



图 1-6　MSP430F13X 系列单片机的管脚图

下面具体介绍 MSP430F13X 系列单片机的各个管脚的功能，使读者能够通过了解单片机的管脚功能来完成单片机的硬件设计，对于硬件设计来说，这是必须的而且也是非常重要的。通过图 1-6 可以看出，该系列单片机具有更多的端口，这样能使用户实现更为复杂的系统，同时也可以减少系统实现的复杂性。这样将许多的功能集成到一片芯片上，增加了系统的可靠性，同时也可以减小硬件的 PCB 板的尺寸。

- P1.0/TACLK：通用数字 I/O 管脚 / Timer_A，TACLK 时钟输入信号。
- P1.1/TA0：通用数字 I/O 管脚 / Timer_A，捕获：CCI0A 输入，比较：OUT0 输出。
- P1.2/TA1：通用数字 I/O 管脚 / Timer_A，捕获：CCI1A 输入，比较：OUT1 输出。
- P1.3/TA2：通用数字 I/O 管脚 / Timer_A，捕获：CCI2A 输入，比较：OUT2 输出。
- P1.4/SMCLK：通用数字 I/O 管脚 / SMCLK 信号输出。
- P1.5/TA0：通用数字 I/O 管脚 / Timer_A，比较：OUT0 输出。
- P1.6/TA1：通用数字 I/O 管脚 / Timer_A，比较：OUT1 输出。
- P1.7/TA2：通用数字 I/O 管脚 / Timer_A，比较：OUT2 输出。
- P2.0/ACLK：通用数字 I/O 管脚 / ACLK 输出端。
- P2.1/TAINCLK：通用数字 I/O 管脚 / Timer_A，INCLK 时钟信号。
- P2.2/CAOUT/TA0：通用数字 I/O 管脚 / Timer_A，捕获：CCI0B 输入，比较：OUT0 输出。
- P2.3/CA0/TA1：通用数字 I/O 管脚 / Timer_A，捕获：CCI0B 输入，比较：OUT1 输出。
- P2.4/CA1/TA2：通用数字 I/O 管脚 / Timer_A，比较：OUT2 输出。
- P2.5/Rosc：通用数字 I/O 管脚 / 作为外接电阻管脚，通过接一电阻来确定 DCO 的工作频率。
- P2.6/ADC12CLK：通用数字 I/O 管脚 / 12 位 A/D 转换器的转换时钟。
- P2.7/TA0：通用数字 I/O 管脚 / Timer_A，比较：OUT0 输出。
- P3.0/STE0：通用数字 I/O 管脚 / 从传输使能：USART0 / SPI 模式。
- P3.1/SIMO0：通用数字 I/O 管脚 / USART0 / SPI 模式下的从输入或者主输出。
- P3.2/SOMI0：通用数字 I/O 管脚 / USART0 / SPI 模式下的从输出或者主输入。
- P3.3/UCLK0：通用数字 I/O 管脚 / 外部时钟输入——USART0 / UART 或 SPI 模式，时钟输出——USART0 / SPI 模式。
- P3.4/UTXD0：通用数字 I/O 管脚 / 发送数据输出——USART0 / SPI 模式。
- P3.5/URXD0：通用数字 I/O 管脚 / 发送数据输入——USART0 / SPI 模式。
- P3.6：通用数字 I/O 管脚。
- P3.7：通用数字 I/O 管脚。
- P4.0/TB0：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI0A 或者 CCI0B 输入，比较：OUT0 输出。
- P4.1/TB1：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI1A 或者 CCI1B 输入，比较：OUT1 输出。
- P4.2/TB2：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI2A 或者 CCI2B 输入，比较：OUT2 输出。

- P4.3：通用数字 I/O 管脚。
- P4.4：通用数字 I/O 管脚。
- P4.5：通用数字 I/O 管脚。
- P4.6：通用数字 I/O 管脚。
- P4.7/TBCLK：通用数字 I/O 管脚 / 定时器 B_3 的输入时钟 TBCLK。
- P5.0：通用数字 I/O 管脚。
- P5.1：通用数字 I/O 管脚。
- P5.2：通用数字 I/O 管脚。
- P5.3：通用数字 I/O 管脚。
- P5.4/MCLK：通用数字 I/O 管脚 / 主系统时钟 MCLK 输出。
- P5.5/SMCLK：通用数字 I/O 管脚 / 子系统时钟 SMCLK 输出。
- P5.6/ACLK：通用数字 I/O 管脚 / 辅助时钟 ACLK 输出。
- P5.7/TBoutH：通用数字 I/O 管脚 / 切换所有的 PWM 数字输出口为高阻抗——定时器 B_3 TB0～TB3。
- P6.0/A0：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 0。
- P6.1/A1：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 1。
- P6.2/A2：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 2。
- P6.3/A3：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 3。
- P6.4/A4：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 4。
- P6.5/A5：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 5。
- P6.6/A6：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 6。
- P6.7/A7：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 7。
- $DV_{CC}$：数字电源端。
- $AV_{CC}$：模拟电源端。
- $AV_{SS}$：模拟电源地。
- $DV_{SS}$：数字电源地。
- $V_{REF}$：A/D 转换器内部基准电压的正输出端。
- XIN：晶体振荡器 XT1 的输入口。
- XOUT/TCLK：晶体振荡器 XT1 的输出端 / 测试时钟输入端。
- $Ve_{REF}$：A/D 转换器外部基准电压。
- $V_{REF-}/Ve_{REF}$：A/D 转换器内部基准电压或者外部基准电压负端。
- XT2IN：晶体振荡器 XT2 的输入口。
- XT2OUT：晶体振荡器 XT2 的输出端。
- $\overline{RST}$/NMI：复位信号输入端 / 不可屏蔽中断输入端。
- TCK：测试时钟，用于器件编程和测试时的时钟输入端。
- TMS：测试方式选择，器件编程和测试输入端。
- TDI：测试数据输入端。
- TDO/TDI：测试数据输出端 / 编程时数据输入端。

## 1.2.4　MSP430F14X 系列单片机

该系列的单片机主要有MSP430F147、MSP430F1471、MSP430F148、MSP430F1481、MSP430F149和MSP430F1491等几种型号。该系列单片机主要有以下特点。

- 具有很低的供电电压。单片机的供电电压最低可以低到 1.8V，单片机的供电电压范围是：1.8V～3.6V。
- 超低的功耗。这是目前其他单片机没有的特色。它在休眠的条件下工作的电流只有 0.8μA，就是在 2.2V、1MHz 条件下工作的电流只有 280μA。
- 快速的唤醒时间。从休眠方式唤醒只需要 6μs。
- 快速的指令执行时间。它采用的是 16 位的 RISC 结构，指令的执行时间只需要 150ns，是传统单片机不能比拟的。
- 片内有 12 位的 A/D 转换器，片内提供参考电压。A/D 转换器具有采样保持和自动扫描特点。
- 16 位的定时器 Timer_B 带有 7 个捕获/比较寄存器。
- 片内提供温度传感器。
- 具有灵活的时钟设置。主要有以下几种种方式：32kHz 的晶体方式、高频率晶体方式、谐振器方式和外部时钟源方式。这样可以根据功耗要求和速度要求进行灵活的时钟设置。
- 16 位的定时器 Timer_A 带有 3 个捕获/比较寄存器。
- 片内提供模拟信号比较器。
- 串口通信模块：USART0、USART1。两个串口都可以通过软件选择设置成 UART 方式或者 SPI 方式，由于该系列单片机提供了两个串口，因此能为用户进行多机通信设计提供方便。
- 片内提供较多的存储器，MSP430F147 提供的片内的 FLASH 为 32KB，MSP430F149 提供的片内的 FLASH 为 60KB，同时片内还提供较多的 RAM，以便进行运算处理。
- 提供 P1.0～P6.0 共 6 个数据端口，能为用户提供更多的处理功能。在提供的外围数据端口中，有两个端口具有中断功能，这样能丰富硬件系统的中断资源，也为实现多任务系统提供方便。
- 代码保护功能。单片机的安全熔丝能对程序的代码进行保护，从而可以对知识产权进行保护。
- 具有 JTAG 仿真调试接口，这样非常便于软件的调试。

为了对 MSP430F14X 系列有比较全面清楚的认识，下面给出了该系列单片机的结构框图，如图 1-7 所示。

通过图 1-7 能大致了解该系列单片机的各个功能模块。为了能让读者对该系列单片机有进一步的认识，在此特意介绍一下该系列单片机的各个管脚。图 1-8 为该系列单片机的管脚图。该图所示的只是该系列单片机的基本型号，也有可能在具体不同型号时有一点小的差别，详细介绍请参看具体型号单片机的数据手册。

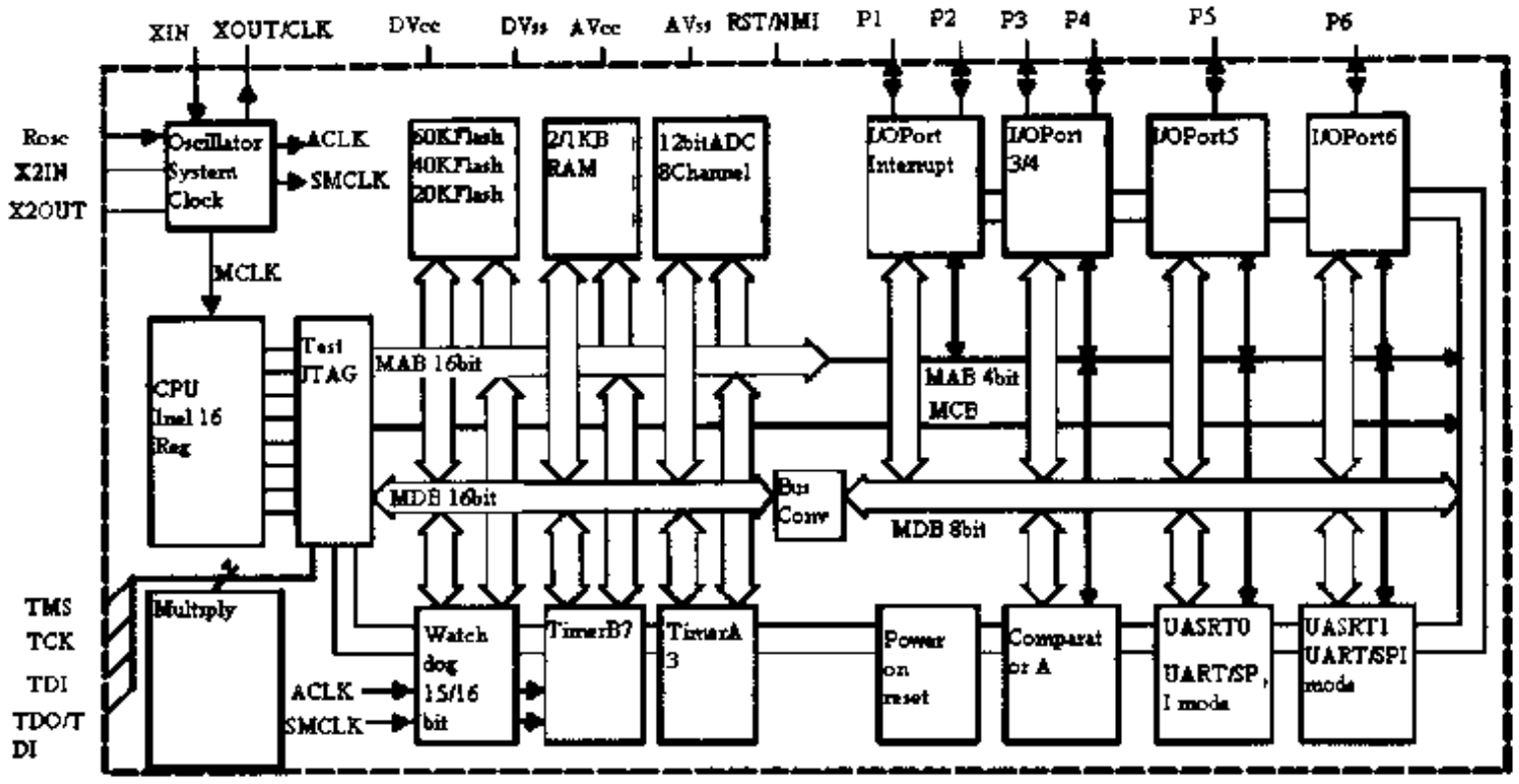



图 1-7　MSP430F14X 系列单片机的结构框图

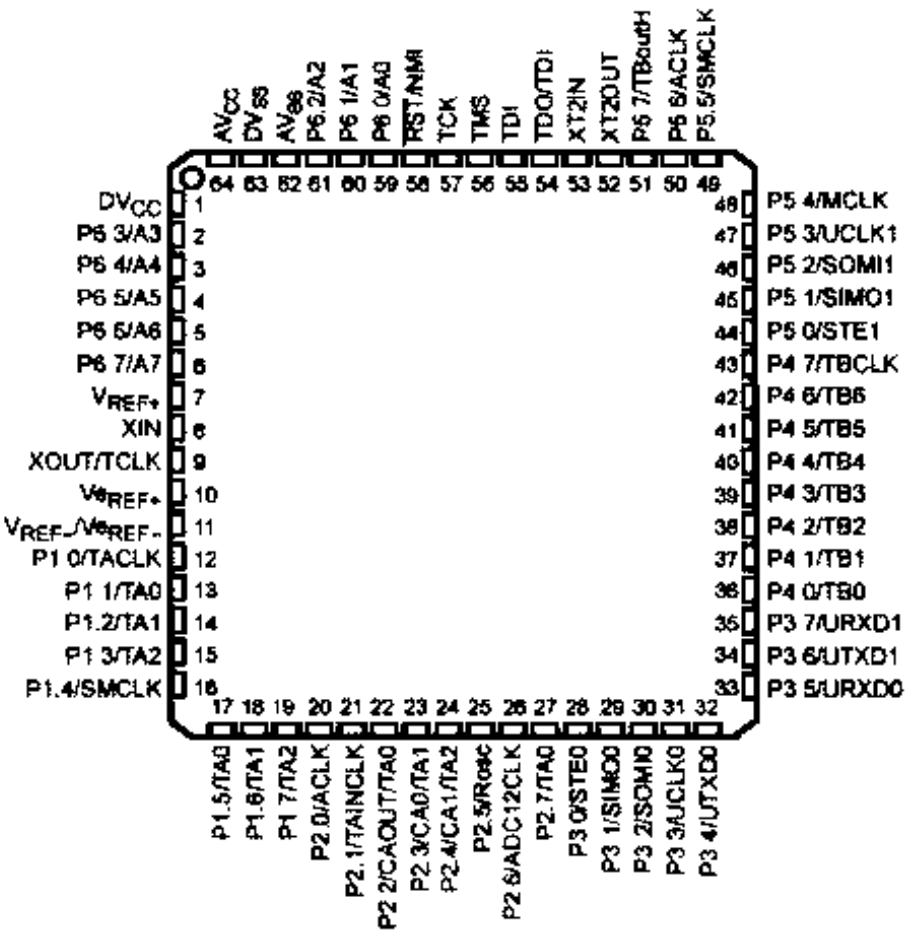



图 1-8　MSP430F14X 系列单片机的管脚图

下面具体介绍单片机的各个管脚的功能，使读者能够通过了解单片机的管脚功能来完成单片机的硬件设计，对于硬件设计来说，这是必须的而且也是非常重要的。通过图 1-8 可以看出，该系列单片机具有更多的端口，这样能使用户实现更为复杂的系统，同时也可以减少系统实现的复杂性。这样将许多的功能集成到一片芯片上，增加了系统的可靠性，同时也可以减小硬件的 PCB 板的尺寸。

- P1.0/TACLK：通用数字 I/O 管脚 / Timer_A，TACLK 时钟输入信号。
- P1.1/TA0：通用数字 I/O 管脚 / Timer_A，捕获：CCI0A 输入，比较：OUT0 输出。
- P1.2/TA1：通用数字 I/O 管脚 / Timer_A，捕获：CCI1A 输入，比较：OUT1 输出。
- P1.3/TA2：通用数字 I/O 管脚 / Timer_A，捕获：CCI2A 输入，比较：OUT2 输出。

- P1.4/SMCLK：通用数字 I/O 管脚 / SMCLK 信号输出。
- P1.5/TA0：通用数字 I/O 管脚 / Timer_A，比较：OUT0 输出。
- P1.6/TA1：通用数字 I/O 管脚 / Timer_A，比较：OUT1 输出。
- P1.7/TA2：通用数字 I/O 管脚 / Timer_A，比较：OUT2 输出。
- P2.0/ACLK：通用数字 I/O 管脚 / ACLK 输出端。
- P2.1/TAINCLK：通用数字 I/O 管脚 / Timer_A，INCLK 时钟信号。
- P2.2/CAOUT/TA0：通用数字 I/O 管脚 / Timer_A，捕获：CCI0B 输入，比较：OUT0 输出。
- P2.3/CA0/TA1：通用数字 I/O 管脚 / Timer_A，捕获：CCI0B 输入，比较：OUT1 输出。
- P2.4/CA1/TA2：通用数字 I/O 管脚 / Timer_A，比较：OUT2 输出。
- P2.5/Rosc：通用数字 I/O 管脚 / 作为外接电阻管脚，通过接一电阻来确定 DCO 的工作频率。
- P2.6/ADC12CLK：通用数字 I/O 管脚 / 12 位 A/D 转换器的转换时钟。
- P2.7/TA0：通用数字 I/O 管脚 / Timer_A，比较：OUT0 输出。
- P3.0/STE0：通用数字 I/O 管脚 / 从传输使能：USART0 / SPI 模式。
- P3.1/SIMO0：通用数字 I/O 管脚 / USART0 / SPI 模式下的从输入或者主输出。
- P3.2/SOMI0：通用数字 I/O 管脚 / USART0 / SPI 模式下的从输出或者主输入。
- P3.3/UCLK0：通用数字 I/O 管脚 / 外部时钟输入——USART0 / UART 或 SPI 模式，时钟输出——USART0 / SPI 模式。
- P3.4/UTXD0：通用数字 I/O 管脚 / 发送数据输出——USART0 / SPI 模式。
- P3.5/URXD0：通用数字 I/O 管脚 / 发送数据输入——USART0 / SPI 模式。
- P3.6/ UTXD1：通用数字 I/O 管脚 / 发送数据输出——USART1 / SPI 模式。
- P3.7/ URXD1：通用数字 I/O 管脚 / 发送数据输入——USART1 / SPI 模式。
- P4.0/TB0：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI0A 或者 CCI0B 输入，比较：OUT0 输出。
- P4.1/TB1：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI1A 或者 CCI1B 输入，比较：OUT1 输出。
- P4.2/TB2：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI2A 或者 CCI2B 输入，比较：OUT2 输出。
- P4.3/TB3：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI3A 或者 CCI3B 输入，比较：OUT3 输出。
- P4.4/TB4：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI4A 或者 CCI4B 输入，比较：OUT4 输出。
- P4.5/TB5：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI5A 或者 CCI5B 输入，比较：OUT5 输出。
- P4.6/TB6：通用数字 I/O 管脚 / 定时器 Timer_B，捕获：CCI6A 或者 CCI6B 输入，比较：OUT6 输出。
- P4.7/TBCLK：通用数字 I/O 管脚 / 定时器 Timer_B 的输入时钟 TBCLK。
- P5.0/ STE1：通用数字 I/O 管脚 / 从传输使能：USART1 / SPI 模式。
- P5.1/ SIMO1：通用数字 I/O 管脚 / USART1 / SPI 模式下的从输入或者主输出。

- P5.2/ SOMI1：通用数字 I/O 管脚 / USART1 / SPI 模式下的从输出或者主输入。
- P5.3/ UCLK1：通用数字 I/O 管脚 / 外部时钟输入——USART1 / UART 或 SPI 模式，时钟输出——USART1 / SPI 模式。
- P5.4/MCLK：通用数字 I/O 管脚 / 主系统时钟 MCLK 输出。
- P5.5/SMCLK：通用数字 I/O 管脚 / 子系统时钟 SMCLK 输出。
- P5.6/ACLK：通用数字 I/O 管脚 / 辅助时钟 ACLK 输出。
- P5.7/TBoutH：通用数字 I/O 管脚 / 切换所有的 PWM 数字输出口为高阻抗——定时器 B_3 TB0～TB3。
- P6.0/A0：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 0。
- P6.1/A1：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 1。
- P6.2/A2：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 2。
- P6.3/A3：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 3。
- P6.4/A4：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 4。
- P6.5/A5：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 5。
- P6.6/A6：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 6。
- P6.7/A7：通用数字 I/O 管脚 / 12 位的 A/D 转换器模拟输入通道 7。
- $DV_{CC}$：数字电源端。
- $AV_{CC}$：模拟电源端。
- $AV_{SS}$：模拟电源地。
- $DV_{SS}$：数字电源地。
- $V_{REF}$：A/D 转换器内部基准电压的正输出端。
- XIN：晶体振荡器 XT1 的输入口。
- XOUT/TCLK：晶体振荡器 XT1 的输出端 / 测试时钟输入端。
- $Ve_{REF}$：A/D 转换器外部基准电压。
- $V_{REF-}/Ve_{REF}$：A/D 转换器内部基准电压或者外部基准电压负端。
- XT2IN：晶体振荡器 XT2 的输入口。
- XT2OUT：晶体振荡器 XT2 的输出端。
- $\overline{RST}$/NMI：复位信号输入端 / 不可屏蔽中断输入端。
- TCK：测试时钟，用于器件编程和测试时的时钟输入端。
- TMS：测试方式选择，器件编程和测试输入端。
- TDI：测试数据输入端。
- TDO/TDI：测试数据输出端 / 编程时数据输入端。

以上介绍了 MSP430F1XX 系列单片机主要型号的结构，并对各个系列的单片机的管脚进行了介绍。为了丰富 MSP430 系列产品的功能和应用领域，TI 公司还提供了 MSP430F15X、MSPF16X、MSP4303XX 和 MSP430F4XX 等单片机，这些系列单片机主要是提供了各种新功能以满足新的需要，比如新增的 $I^2C$ 功能和与液晶接口的功能都是为用户进行设计时提供更多的选择空间，但是万变不离其宗，只要掌握了 MSP430 单片机的基本结构和基本技术的开发，就很容易过渡到其他型号的单片机，因此本书主要通过对 MSP430F1XX 系列单片机的开发介绍来使读者达到融会贯通的地步。

# 第 2 章　MSP430F1XX 的 CPU 与外设

通过第 1 章对 MSP430F1XX 系列单片机结构的介绍，读者对 MSP430F1XX 系列单片机已经有了基本的认识。为了进一步增加对 MSP430F1XX 的了解，本章将详细介绍该系列单片机的 CPU 和外设，同时也为下面关于 MSP430F1XX 单片机的具体开发做好良好的准备。从第 1 章可以知道：MSP430F1XX 单片机主要包括 CPU 和外设，主要的外设有存储器、时钟模块、定时器、比较器、USART 和 A/D 转换器等模块。下面就各个部分进行具体的介绍。

## 2.1　MSP430 的 CPU

MSP430 的内核 CPU 结构是按照精简指令集和高透明的宗旨来设计的。MSP430 系列采用的是“冯-诺依曼”结构，ROM 和 RAM 在同一地址空间，使用同一组地址数据总线。MSP430 系列单片机采用的是 16 位结构的 CPU，它采用了精简的、高透明的、高效率的正交设计，它包括一个 16 位的算术逻辑单元（ALU）、16 个寄存器和一个指令控制单元。16 个寄存器中有 4 个特殊的功能寄存器和通用寄存器。4 个特殊的功能寄存器分别是：程序计数器（PC）、堆栈指针（SP）、状态寄存器和常数发生器。程序计数器是用来表示下一条即将执行指令的地址，也就是说程序执行的地方，采用 C 语言写程序的时候可以不必关心该寄存器。堆栈指针主要用在系统调用子程序或者进入中断服务程序的时候对程序计数器的保护，就是保护程序的现场情况，采用 C 语言写程序时不必关心该寄存器。常数发生器主要用来产生常数，采用 C 语言写程序时也不必关心该寄存器。状态寄存器用来设置某些位来控制 CPU 的行为或者通过某些位来反映 CPU 的状态，在写 C 语言或者汇编语言时都需要用户设置适当的位或者读取出适当的位，从而控制 CPU 的运行行为，因此该寄存器非常重要，图 2-1 给出该寄存器的结构。

| 15~9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| 保留 | V | SCG1 | SCG0 | OscOff | CPUOff | GIE | N | Z | C |

图 2-1　状态寄存器的结构

图 2-1 给出了状态寄存器的具体位分配，为了理解每个位的意义，下面对各个位的含义进行详细的介绍。

- C：进位标志位。当运算结果产生进位的时候 CPU 将该位设置为 1，否则该位的内容为 0。
- Z：零标志位。当运算结果为 0 的时候设置该位为 1，否则该位的内容为 0。
- N：负标志位。当运算结果为负的时候 CPU 将该位设置为 1，否则该位的内容为 0。

- GIE：中断控制位。该位为中断允许位，将该位设置为 1 的时候，允许中断；如果该位设置为 0 的时候，禁止所有的中断。该位需要用户在写程序的时候根据自己的需要进行设置。
- CPUOff：CPU 控制位。如果将该位设置为 1 的话，则 CPU 进入关闭模式，这时除了 RAM 内容、端口和寄存器保持外，CPU 处于停止状态，可以通过某种中断将 CPU 唤醒。如果将该位设置为 0 的话，则 CPU 处于工作状态。
- OscOff：晶体控制位。如果将该位设置为 1 的话，晶体振荡器处于停止状态；如果将该位设置为 0 的话，则晶体处于工作状态。如果要设置该位为 1 的话，还必须同时设置 CPUOff 为 1。
- SCG0：该位与 SCG1 结合使用，用来控制系统时钟的 4 种状态。如表 2-1 所示。
- SCG1：该位与 SCG0 结合使用，用来控制系统时钟的 4 种状态。如表 2-1 所示。
- V：溢出标志位。如果运算结果超出范围，则 CPU 将该位设置为 1，如果没有超出范围，则该位的内容为 0。

**表 2-1　系统时钟的状态**

| SCG0 | SCG1 | 系统时钟的状态 |
|---|---|---|
| 0 | 0 | SMCLK，ACLK |
| 1 | 0 | SMCLK，ACLK |
| 0 | 1 | ACLK |
| 1 | 1 | ACLK |

由表 2-1 可以看出，适当设置 SCG1 和 SCG0 两个位可以控制系统时钟的状态。

## 2.2　存储器组织结构

由于 MSP430 系列单片机采用的是“冯-诺依曼”结构，因此它的 ROM 和 RAM 都在同一地址空间，虽然不同型号的单片机具有不同容量的存储器，但是它们都分配在 0000H～FFFFH 范围内，存储器是按线性方式组织的。在 0000H～FFFFH 范围内从低到高分别是：特殊功能寄存器、外围模块寄存器、数据存储器、程序存储器和中断向量表。为了有一个直观的认识，下面给出几种单片机的存储器组织图。

图 2-2 为 MSP430F11X 系列单片机的存储器结构，通过图 2-2 可以看出，00H～0FH 为特殊功能寄存器，主要包括 4 个特殊功能寄存器和通用寄存器。10H～FFH 为宽为 8 位的寄存器，这些寄存器是 8 位外围模块使用的寄存器。100H～1FFH 为宽为 16 位的寄存器，这些寄存器是 16 位外围模块使用的寄存器。从 200H 到 C00H 这一地址范围是 RAM 存储器，不同型号的单片机具有不同的 RAM 空间，像 MSP430F110 的 RAM 空间为：200H～27FH，而 MSP430F112 的 RAM 空间为：200H～2FFH。C00H～FFFH 这一地址范围内为片内 ROM 存储器。FFE0H～FFFFH 这一地址范围为中断向量表的空间，每一种单片机都有相同的中断向量表的位置。其他空间是单片机的程序空间，像 MSP430F110 的程序空间为：1080H～10FFH 和 FC00H～FFDFH 这两段，而 MSP430F112 的程序空间为：1000H～10FFH 和 F000H～FFDFH 这两段，片内程序存储器根据不同型号的单片机有不同的程序空间，也有不同的开始地址。

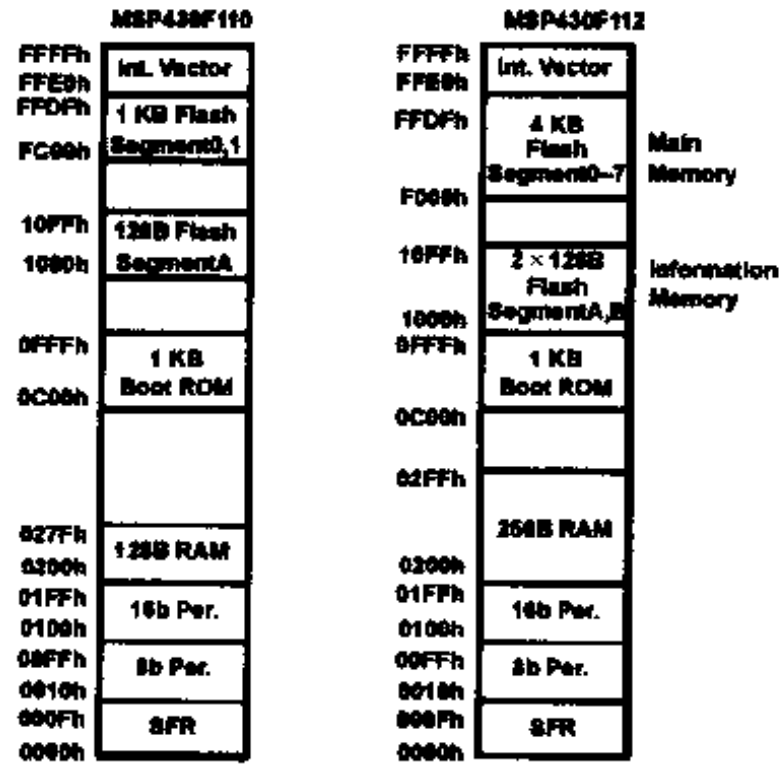



图 2-2　MSP430F11X 系列单片机的存储器的结构

图 2-3 为 MSP430F12X 系列单片机的存储器结构，通过图 2-3 可以看出，00H～0FH 为特殊功能寄存器，主要包括 4 个特殊功能寄存器和通用寄存器。10H～FFH 为宽为 8 位的寄存器，这些寄存器是 8 位外围模块使用的寄存器。100H～1FFH 为宽为 16 位的寄存器，这些寄存器是 16 位外围模块使用的寄存器。从 200H 到 C00H 这一地址范围是 RAM 存储器，不同型号的单片机具有不同的 RAM 空间，像 MSP430F122 和 MSP430F123 的 RAM 空间为：200H～2FFH。C00H～FFFH 这一地址范围内为片内 ROM 存储器。FFE0H～FFFFH 这一地址范围为中断向量表的空间，每一种单片机都有相同的中断向量表的位置。其他空间是单片机的程序空间，像 MSP430F122 的程序空间为：1000H～10FFH 和 F000H～FFDFH 这两段，而 MSP430F123 的程序空间为：1000H～10FFH 和 E000H～FFDFH 这两段，片内程序存储器根据不同型号的单片机有不同的程序空间，也有不同的开始地址。

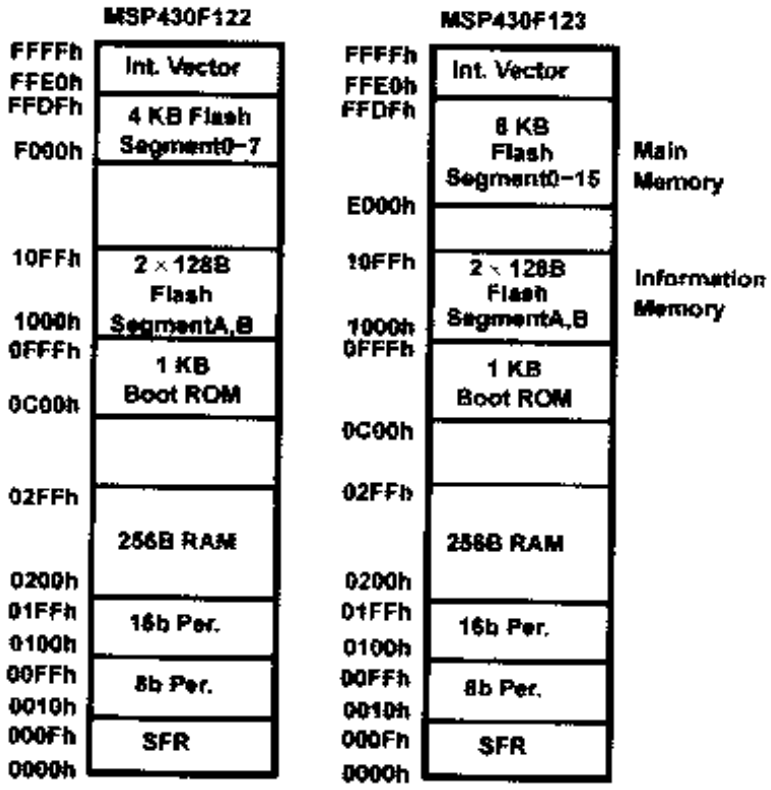



图 2-3　MSP430F12X 系列单片机的存储器的结构

图 2-4 为 MSP430F13X 和 MSP430F14X 系列单片机的存储器结构，通过图 2-4 可以看出，

00H～0FH 为特殊功能寄存器，主要包括 4 个特殊功能寄存器和通用寄存器。10H～FFH 为宽为 8 位的寄存器，这些寄存器是 8 位外围模块使用的寄存器。100H～1FFH 为宽为 16 位的寄存器，这些寄存器是 16 位外围模块使用的寄存器。从 200H～C00H 这一地址范围是 RAM 存储器，不同型号的单片机具有不同的 RAM 空间，像 MSP430F133 RAM 空间为：200H～2FFH 和 MSP430F49 的 RAM 空间为：200H～9FFH。C00H～FFFH 这一地址范围内为片内 ROM 存储器。FFE0H～FFFFH 这一地址范围为中断向量表的空间，每一种单片机都有相同的中断向量表的位置。其他空间是单片机的程序空间，像 MSP430F133 的程序空间为：1000H～10FFH 和 E000H～FFDFH 这两段，而 MSP430F149 的程序空间为：1000H～10FFH 和 1100H～FFDFH 这两段，片内程序存储器根据不同型号的单片机有不同的程序空间，也有不同的开始地址。

| | | MSP430F133 | MSP430F135 | MSP430F147/ MSP430F1471 | MSP430F148/ MSP430F1481 | MSP430F149/ MSP430F1491 |
|---|---|---|---|---|---|---|
| Memory<br>Main: interrupt vector<br>Main: code memory | Size<br>Flash<br>Flash | 8KB<br>0FFFFh – 0FFE0h<br>0FFFFh – 0E000h | 16KB<br>0FFFFh – 0FFE0h<br>0FFFFh – 0C000h | 32KB<br>0FFFFh – 0FFE0h<br>0FFFFh – 08000h | 48KB<br>0FFFFh – 0FFE0h<br>0FFFFh – 04000h | 60KB<br>0FFFFh – 0FFE0h<br>0FFFFh – 01100h |
| Information memory | Size<br>Flash | 256 Byte<br>010FFh – 01000h | 256 Byte<br>010FFh – 01000h | 256 Byte<br>010FFh – 01000h | 256 Byte<br>010FFh – 01000h | 256 Byte<br>010FFh – 01000h |
| Boot memory | Size<br>ROM | 1KB<br>0FFFh – 0C00h | 1KB<br>0FFFh – 0C00h | 1KB<br>0FFFh – 0C00h | 1KB<br>0FFFh – 0C00h | 1KB<br>0FFFh – 0C00h |
| RAM | Size | 256 Byte<br>02FFh – 0200h | 512 Byte<br>03FFh – 0200h | 1KB<br>05FFh – 0200h | 2KB<br>09FFh – 0200h | 2KB<br>09FFh – 0200h |
| Peripherals | 16-bit<br>8-bit<br>8-bit SFR | 01FFh – 0100h<br>0FFh – 010h<br>0Fh – 00h | 01FFh – 0100h<br>0FFh – 010h<br>0Fh – 00h | 01FFh – 0100h<br>0FFh – 010h<br>0Fh – 00h | 01FFh – 0100h<br>0FFh – 010h<br>0Fh – 00h | 01FFh – 0100h<br>0FFh – 010h<br>0Fh – 00h |

图 2-4　MSP430F13X 和 MSP430F14X 系列单片机的存储器的结构

通过以上几种单片机的存储器可以发现：MSP430F1XX 系列单片机的存储器的组织结构有相同的地方也有不同的地方。相同的地方主要有：

- 所有单片机的中断向量表具有相同的地址空间，都在 FFE0H～FFFFH 这一地址范围内。
- 所有单片机的 8 位外围模块使用的寄存器具有相同的地址空间，都在 10H～FFH 这一地址范围内。
- 所有单片机的 16 位外围模块使用的寄存器具有相同的地址空间，都在 100H～1FFH 这一地址范围内。
- 所有单片机的特殊功能寄存器都有相同的地址空间，都在 00H～0FH 这一地址范围内。
- 所有单片机的数据存储器具有相同的起始地址，都从 200H 出开始。
- 所有单片机的存储器都在 0～FFFFH 范围内。

不同的地方主要有：

- 不同的单片机有不同的程序存储器。
- 不同的单片机有不同的数据存储器。

经过上面的介绍，对 MSP430 系列单片机的存储器组织结构有一定的认识，为了增加对 MSP430 系列单片机的存储器进一步认识，下面分别从数据存储器（RAM）和程序存储器（ROM）做一下具体的介绍。

### 2.2.1　数据存储器 RAM

MSP430 系列单片机的数据存储器位于起始地址为 200H 的储器地址空间。数据存储器既作为数据的保存，也作为堆栈，同时也是数学运算的场所，在某些场合还可以作为程序存储器。数据存储器可以按字操作，也可以按字节操作，在 C 语言程序里，声明字变量和字节变量就可以实现字操作和字节操作。另外，FLASH 型的单片机里的还有信息存储区，也可以作为数据 RAM 使用，并且因为它是 FLASH 的，因此在掉电后数据不会丢失，这样可以用这部分存储器来保存重要的参数，比如单片机系统里的配置数据。

### 2.2.2　程序存储器 ROM

程序空间 ROM 可以存放指令和数据表格。程序存储器主要有 3 种：中断向量表、用户代码区和系统 BOOT 区。其中系统 BOOT 区不是每一种单片机都有的，一般 FLASH 型的单片机有。用户代码区用来存放用户的程序。中断向量表是中断服务程序的首地址，用户在写中断程序的时候，某一中断程序的入口必须是中断向量表里固定的位置，而不能由用户进行分配。不同的器件有不同的中断资源，它们的中断向量表也有所不同。下面表 2-2 给出几种单片机的中断向量表。

表 2-2　MSP430F11X 系列单片机的中断向量表

| 中断源 | 中断标志 | 系统中断 | 字地址 | 优先级 |
|---|---|---|---|---|
| 上电、复位、看门狗 | WDTIFG、KEYV | 复位 | FFFEH | 15 |
| NMI、振荡器出错、FLASH 访问非法 | NMIIFG、OFIFG、ACCVIFG | 不可屏蔽 | FFFCH | 14 |
| 看门狗定时器 | WDTIFG | 可屏蔽 | FFF4H | 10 |
| Timer_A | TACCRO、CCIFG | 可屏蔽 | FFF2H | 9 |
| Timer_A | TACCR1、CCIFG 等 | 可屏蔽 | FFF0H | 8 |
| P2 口 | P2IFG.0～P2IFG.7 | 可屏蔽 | FFE6H | 3 |
| P1 口 | P1IFG.0～P1IFG.7 | 可屏蔽 | FFE4H | 2 |

由表 2-2 可以看出：中断具有不同的中断向量地址，也有不同的中断优先级，优先级按数字的大小从高到低，优先级高的先进入中断。另外通过表 2-2 也可以看出：同一个中断地址有不同的中断源，这样在写程序处理的时候一定要加以区分，否则在进行中断处理时会出现错误。

表 2-3 为 MSP430F12X 系列单片机的中断向量表。

表 2-3　MSP430F12X 系列单片机的中断向量表

| 中断源 | 中断标志 | 系统中断 | 字地址 | 优先级 |
|---|---|---|---|---|
| 上电、复位、看门狗 | WDTIFG、KEYV | 复位 | FFFEH | 15 |
| NMI、振荡器出错、FLASH 访问非法 | NMIIFG、OFIFG、ACCVIFG | 不可屏蔽 | FFFCH | 14 |
| 比较器 | CAIFG | 可屏蔽 | FFF6H | 11 |

续表

| 中断源 | 中断标志 | 系统中断 | 字地址 | 优先级 |
|---|---|---|---|---|
| 看门狗定时器 | WDTIFG | 可屏蔽 | FFF4H | 10 |
| Timer_A | TACCRO、CCIFG | 可屏蔽 | FFF2H | 9 |
| Timer_A | TACCR1、CCIFG 等 | 可屏蔽 | FFF0H | 8 |
| USART0 接收 | URXIFG0 | 可屏蔽 | FFEEH | 7 |
| USART0 发送 | UTXIFG0 | 可屏蔽 | FFECH | 6 |
| P2 口 | P2IFG.0～P2IFG.7 | 可屏蔽 | FFE6H | 3 |
| P1 口 | P1IFG.0～P1IFG.7 | 可屏蔽 | FFE4H | 2 |

由表 2-3 可以看出：中断具有不同的中断向量地址，也有不同的中断优先级，优先级按数字的大小从高到低，优先级高的先进入中断。另外通过表 2-3 也可以看出：同一个中断地址有不同的中断源，这样在写程序处理的时候一定要加以区分，否则在进行中断处理时会出现错误。与 MSP430F11X 相比，MSP430F12X 增加了比较器中断、USART0 接收中断和 USART0 发送中断。

表 2-4 为 MSP430F13X 系列单片机的中断向量表。

表 2-4　MSP430F13X 系列单片机的中断向量表

| 中断源 | 中断标志 | 系统中断 | 字地址 | 优先级 |
|---|---|---|---|---|
| 上电、复位、看门狗 | WDTIFG、KEYV | 复位 | FFFEH | 15 |
| NMI、振荡器出错、FLASH 访问非法 | NMIIFG、OFIFG、ACCVIFG | 不可屏蔽 | FFFCH | 14 |
| Timer_B | TBCCR0 CCIFG | 可屏蔽 | FFFAH | 13 |
| Timer_B | TBCCR1～TBCCR2 等 | 可屏蔽 | FFF8H | 12 |
| 比较器 A | CAIFG | 可屏蔽 | FFF6H | 11 |
| 看门狗定时器 | WDTIFG | 可屏蔽 | FFF4H | 10 |
| USART0 接收 | URXIFG0 | 可屏蔽 | FFF2H | 9 |
| USART0 发送 | UTXIFG0 | 可屏蔽 | FFF0H | 8 |
| ADC12 | ADC12IFG | 可屏蔽 | FFEEH | 7 |
| Timer_A | TACCR0 CCIFG | 可屏蔽 | FFECH | 6 |
| Timer_A | TACCR1 CCIFG 等 | 可屏蔽 | FFEAH | 5 |
| P1 口 | P1IFG.0～P1IFG.7 | 可屏蔽 | FFE8H | 4 |
| USART1 接收 | URXIFG1 | 可屏蔽 | FFE6H | 3 |
| USART1 发送 | UTXIFG1 | 可屏蔽 | FFE4H | 2 |
| P2 口 | P2IFG.0～P2IFG.7 | 可屏蔽 | FFE2H | 1 |

由表 2-4 可以看出：中断具有不同的中断向量地址，也有不同的中断优先级，优先级按数字的大小从高到低，优先级高的先进入中断。另外通过表 2-4 也可以看出：同一个中断地址有不同的中断源，这样在写程序处理的时候一定要加以区分，否则在进行中断处理时会出现错误。与 MSP430F12X 相比，MSP430F13X 增加了 Timer_B 中断、USART1 接收中断、USART1 发送中断和 A/D 转换器中断。

表 2-5 为 MSP430F14X 系列单片机的中断向量表。

表 2-5　MSP430F14X 系列单片机的中断向量表

| 中断源 | 中断标志 | 系统中断 | 字地址 | 优先级 |
|---|---|---|---|---|
| 上电、复位、看门狗 | WDTIFG、KEYV | 复位 | FFFEH | 15 |
| NMI、振荡器出错、FLASH 访问非法 | NMIIFG、OFIFG、ACCVIFG | 不可屏蔽 | FFFCH | 14 |
| Timer_B | TBCCR0 CCIFG | 可屏蔽 | FFFAH | 13 |
| Timer_B | TBCCR1～TBCCR6 等 | 可屏蔽 | FFF8H | 12 |
| 比较器 A | CAIFG | 可屏蔽 | FFF6H | 11 |
| 看门狗定时器 | WDTIFG | 可屏蔽 | FFF4H | 10 |
| USART0 接收 | URXIFG0 | 可屏蔽 | FFF2H | 9 |
| USART0 发送 | UTXIFG0 | 可屏蔽 | FFF0H | 8 |
| ADC12 | ADC12IFG | 可屏蔽 | FFEEH | 7 |
| Timer_A | TACCR0 CCIFG | 可屏蔽 | FFECH | 6 |
| Timer_A | TACCR1 CCIFG 等 | 可屏蔽 | FFEAH | 5 |
| P1 口 | P1IFG.0～P1IFG.7 | 可屏蔽 | FFE8H | 4 |
| USART1 接收 | URXIFG1 | 可屏蔽 | FFE6H | 3 |
| USART1 发送 | UTXIFG1 | 可屏蔽 | FFE4H | 2 |
| P2 口 | P2IFG.0～P2IFG.7 | 可屏蔽 | FFE2H | 1 |

由表 2-5 可以看出：中断具有不同的中断向量地址，也有不同的中断优先级，优先级按数字的大小从高到低，优先级高的先进入中断。另外通过表 2-5 也可以看出：同一个中断地址有不同的中断源，这样在写程序处理的时候一定要加以区分，否则在进行中断处理时会出现错误。与 MSP430F13X 相比，MSP430F14X 的 Timer_B 有更多的中断标志。

以上给出了 MSP430F1XX 系列单片机的中断向量表，在写程序的时候一定要知道向量表，因为这对于写中断服务程序非常重要，如果需要进一步详细了解的话，请参看相关的数据手册。

### 2.2.3　外围模块寄存器和特殊寄存器

通过前面的介绍已经知道：10H～1FFH 地址范围为外围模块寄存器的空间，在这一空间内，有 8 位的外围模块寄存器，也有 16 位的外围模块寄存器。通过这些寄存器的设置可以控制单片机的某一模块的工作。下面具体对特殊寄存器进行介绍，至于外围模块寄存器这一节不进行详细介绍，安排在具体介绍外围模块的时候再进行详细介绍。由于不同系列的单片机的寄存器可能也不同，这样需要根据单片机的不同型号参看相关的数据手册，在这里就不再强调是那一种单片机的寄存器，而是介绍这一系列单片机的寄存器，也就是具体的单片机的寄存器可能是下面介绍的寄存器的子集，有的可能有，有的可能没有。

中断使能 1 寄存器（IE1）：该寄存器主要是使能某些模块的中断功能，设置相应的位可以使相应的模块具有中断功能，如果不设置，则相应的模块不具有中断功能，下面为该寄存器的位分配，如图 2-5 所示。


图 2-5　中断使能 1 寄存器（IE1）的位分配

由图 2-5 可以看出，该寄存器主要有 6 个位，这 6 个位都是可以读写的，这样就可以对该寄存器的相应位进行设置或者读取。为了能够对这 6 个位进行了解，下面分别对这 6 个位进行介绍。

- UTXIE0：USART0 模块的传输中断使能控制位。当该位设置为 1 时，该模块的中断功能使能；如果该位设置为 0 时，则该模块的中断功能关闭。
- URXIE0：USART0 模块的接收中断使能控制位。当该位设置为 1 时，该模块的中断功能使能；如果该位设置为 0 时，则该模块的中断功能关闭。
- ACCVIE：FLASH 存储器非法访问中断使能控制位。当该位设置为 1 时，该中断功能使能；如果该位设置为 0 时，则该中断功能关闭。
- NMIIE：非屏蔽中断使能控制位。当该位设置为 1 时，该中断功能使能；如果该位设置为 0 时，则该中断功能关闭。
- OFIE：晶体出错中断使能控制位。当该位设置为 1 时，该中断功能使能；如果该位设置为 0 时，则该中断功能关闭。
- WDTIE：看门狗中断使能控制位。当该位设置为 1 时，该中断功能使能；如果该位设置为 0 时，则该中断功能关闭。

通过以上各个位的介绍可以知道：如果使某个模块具有中断功能，只需要将相应的位设置为 1。

中断使能 2 寄存器（IE2）：该寄存器主要是使能某些模块的中断功能，设置相应的位可以使相应的模块具有中断功能，如果不设置，则相应的模块不具有中断功能，下面为该寄存器的位分配，如图 2-6 所示。

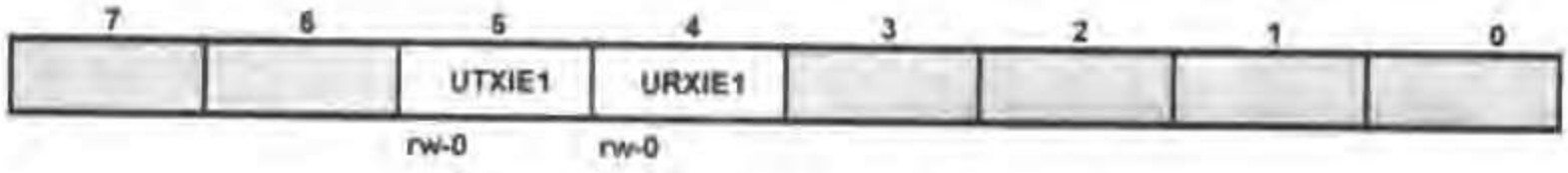



图 2-6　中断使能 2 寄存器（IE2）的位分配

由图 2-6 可以看出，该寄存器主要有两个位，这两个位都是可以读写的，这样就可以对该寄存器的相应位进行设置或者读取。该寄存器是对中断使能 1 寄存器（IE1）的补充，主要是控制 USART1 模块的中断功能。为了能够对这两个位进行了解，下面分别对这两个位进行介绍。

- UTXIE1：USART1 模块的传输中断使能控制位。当该位设置为 1 时，该模块的中断功能使能；如果该位设置为 0 时，则该模块的中断功能关闭。
- URXIE1：USART1 模块的接收中断使能控制位。当该位设置为 1 时，该模块的中断功能使能；如果该位设置为 0 时，则该模块的中断功能关闭。

通过以上两个位的介绍可以知道：如果使串口 1 的接收或者发送具有中断功能，只需要将相应的位设置为 1。

中断标志寄存器 1（IFG1）：该寄存器主要是相应模块的中断标志位，通过设置相应的中断标志，使单片机进入中断。中断标志的设置有的是用户主动设置，有的是单片机完成某种操作后设置，这样也可以读取相应的中断标志位来获取中断标志的状态，下面为该寄存器的位分配，如图 2-7 所示。

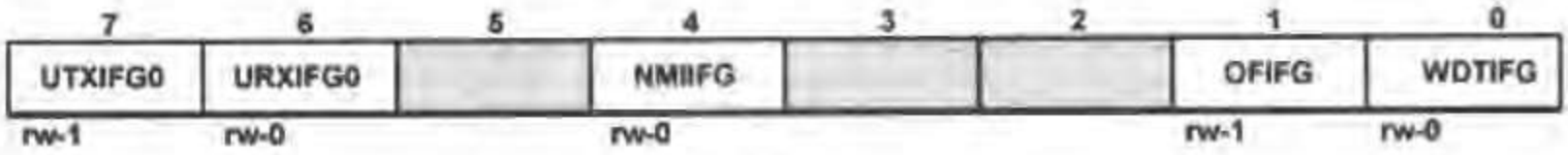



图 2-7　中断标志寄存器 1（IFG1）的位分配

由图 2-7 可以看出，该寄存器主要有 5 个位，这 5 个位都是可以读写的，这样就可以对该寄存器的相应位来判断是否有中断产生，也可以设置某些位，使单片机进入中断。为了能够对这 5 个位进行了解，下面分别对这 5 个位进行介绍。

- UTXIFG0：USART0 传输中断标志位。当设置该位为 1 的时候，则可以使单片机进入发送中断；当该位为 0 时，没有中断产生。
- URXIFG0：USART0 接收中断标志位。当该位为 1 的时候，则有中断产生；当该位为 0 是，没有中断产生。
- NMIIFG：非屏蔽中断标志位。当该位为 1 时有中断产生；当位为 0 时，则没有中断产生。
- OFIFG：晶体出错中断标志。当该位为 1 时有中断产生；当该位为 0 时，则没有中断产生。
- WDTIFG：看门狗中断标志。当该位为 1 时有中断产生；当该位为 0 时，则没有中断产生。当处于看门狗模式下，该中断标志位由软件进行设置或者复位；处于其他模式下，则该标志位自动复位。

通过以上各个位的介绍可以知道：通过以上中断标志位可以判断是否有中断产生，也可以设置某些标志位，使单片机进入中断。

中断标志寄存器 2（IFG2）：该寄存器主要是相应模块的中断标志位，通过设置相应的中断标志，使单片机进入中断，也可以读取中断标志位来判断中断的状态。下面为该寄存器的位分配，如图 2-8 所示。

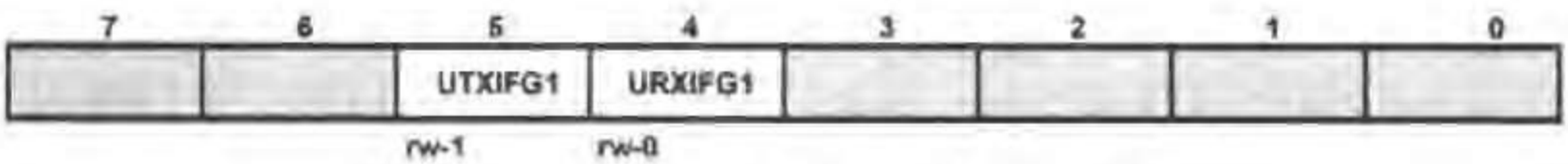



图 2-8　中断标志寄存器 2（IFG2）的位分配

由图 2-8 可以看出，该寄存器主要有两个位，这两个位都是可以读写的，这样就可以对该寄存器的相应位进行设置或者读取。该寄存器是对中断标志寄存器 1（IFG1）的补充，主要是 USART1 模块的中断标志。为了能够对这两个个位进行了解，下面分别对这两个位进行介绍。

- UTXIFG1：USART1 传输中断标志位。当设置该位为 1 的时候，则可以使单片机进入发送中断；当该位为 0 时，没有中断产生。

- URXIFG1：USART1 接收中断标志位。当该位为 1 的时候，则有中断产生；当该位为 0 时，没有中断产生。

由以上两个位的介绍可以知道：通过中断标志位可以判断是否有中断产生，也可以设置 UTXIFG1 标志位，使单片机进入串口的发送中断。

模块使能 1 寄存器（ME1）：该寄存器主要是使能 USART0 模块工作，设置相应的位可以使 USART0 模块工作或者禁止模块工作，下面为该寄存器的位分配，如图 2-9 所示。

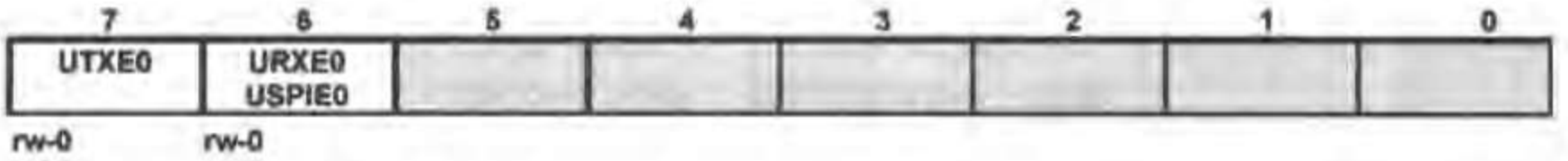



图 2-9　模块使能 1 寄存器（ME1）的位分配

由图 2-9 可以看出，该寄存器主要有两个位，这两个位都是可以读写的，这样就可以对该寄存器的相应位进行设置或者读取。该寄存器主要是使能 USART0 模块。为了能够对这两个位进行了解，下面分别对这两个位进行介绍。

- UTXE0：USART0 的传输使能。当该位设置为 1 时，则传输模块使能；如果该位设置为 0，则 USART0 的传输不工作。
- URXE0/USPIE0：USART0 作为 UART 时，该位控制 UART 的接收使能，当该位设置为 1 时，则接收模块使能，如果该位设置为 0，则 USART0 的接收不工作；USART0 作为 SPI 时，该位控制 SPI 使能，当该位设置为 1 时，SPI 使能，当该位设置为 0 时，则 SPI 不工作。

由以上两个位的介绍可以知道：通过对 UTXE0 和 URXE0 的设置，可以使能 USART0 模块的发送或者接收。当 USART0 处于 SPI 模式下，通过设置 USPIE0 可以使能 SPI 模块。

模块使能 2 寄存器（ME2）：该寄存器主要是使能 USART1 模块工作，设置相应的位可以使 USART1 模块工作或者禁止模块工作，下面为该寄存器的位分配，如图 2-10 所示。

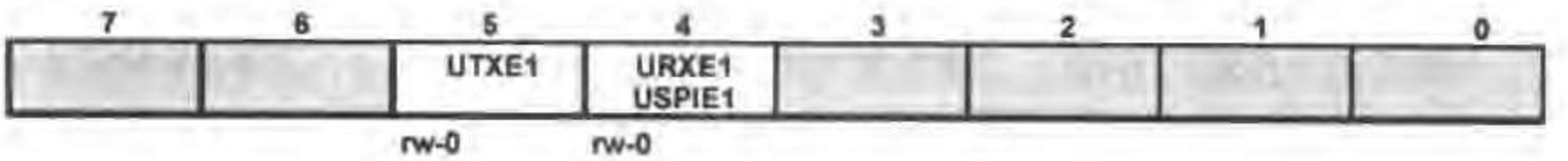



图 2-10　模块使能 2 寄存器（ME2）的位分配

由图 2-10 可以看出，该寄存器主要有两个位，这两个位都是可以读写的，这样就可以对该寄存器的相应位进行设置或者读取。该寄存器主要是使能 USART1 模块。为了能够对这两个位进行了解，下面分别对这两个位进行介绍。

- UTXE1：USART1 的传输使能。当该位设置为 1 时，则传输模块使能；如果该位设置为 0，则 USART0 的传输不工作。
- URXE1/USPIE1：USART1 作为 UART 时，该位控制 UART 的接收使能，当该位设置为 1 时，则接收模块使能，如果该位设置为 0，则 USART1 的接收不工作；USART1 作为 SPI 时，该位控制 SPI 使能，当该位设置为 1 时，SPI 使能，当该位设置为 0 时，则 SPI 不工作。

由以上两个位的介绍可以知道：通过对 UTXE1 和 URXE1 的设置，可以使能 USART1 模块的发送或者接收。当 USART1 处于 SPI 模式下，通过设置 USPIE1 可以使能 SPI 模块。

## 2.3　基础时钟与低功耗

时钟模块是 MSP430 系列单片机不可缺少的模块，时钟模块可以使单片机实现不同的低功耗应用，不同的器件具有不同的时钟模块，一般来说 MSP430F1XX 的时钟模块主要有高速晶体、低速晶体和数字控制振荡器（DCO）等器件构成。高速晶体、低速晶体和 DCO 等器件通过 MSP430F1XX 时钟模块产生 3 个不同的时钟供不同的模块使用，产生的时钟为：辅助时钟（ACLK）、主系统时钟（MCLK）和子系统时钟（SMCLK）。由于时钟模块产生 3 个不同的时钟信号，这样可以采用不同的时钟从而达到低功耗的目的。一般说来，系统的功耗和系统的工作频率成正比关系，这样可以在低功耗应用情况下选用低速晶体。如果系统对运算要求比较高，则可以选用高速晶体产生较高的主系统时钟提供给 CPU，以满足运算要求。如果对系统的实时性要求比较高，则可以采用 ACLK 时钟。总的来说，应该根据不同的应用来选择适当的时钟。

### 2.3.1　低速晶体振荡器

低速晶体支持超低功耗，它在低频模式下使用一个 32768Hz 的晶体。32768Hz 的晶体连接在 XIN 和 XOUT 管脚，不需要任何的电容，在低频的模式下内部提供了集成的电容。低速晶体振荡器在高频模式下也支持高速晶体，连接在 XIN 管脚和 XOUT 管脚之间需要外加电容，每端都必须外加电容，电容的选择需要根据晶体的规范来进行选择。低速晶体振荡器在低频模式和高频模式下可以在 XIN 管脚选择外部时钟信号。当使用外部时钟信号的时候，选择的频率必须满足数据手册中关于选择模式的要求。

### 2.3.2　高速晶体振荡器

高速晶体振荡器作为 MSP430F14XX 的第二晶体振荡器。与低速晶体振荡器不同的是高速晶体振荡器需要的功耗更大。高速晶体振荡器需要外接高速晶体在 XIN2 和 XOUT2 两个管脚，并且必须外接电容，电容的选择必须按照数据手册给出的相关参数进行选择。高速晶体振荡器可以作为 SMCLK 和 ACLK 的时钟源。

### 2.3.3　DCO 振荡器

DCO 是内部集成的 RC 类型的振荡器。DCO 的频率会随温度和电压的变化而变化，并且不同芯片的 DCO 的频率也可能不一样。采用 DCO 方式的时钟信号精度比较差，但是可以通过软件来设置 DCOx、MODx 和 RSELx 等位来调整 DCO 的频率，从而增加 DCO 频率的稳定性。DCO 频率的调整是这样实现的：

- 首先通过外部电阻或者内部电阻来确定一个基准频率。通过设置 DCOR 位来选择是外部电阻还是内部电阻。
- 通过 3 个 RSELx 位来分频。
- 通过 3 个 DCOx 位来选择频率。

- 通过 5 个 MODx 位来选择 DCOx 和 DCOx+1 之间的频率。当 DCOx=07h 时，MODx 位对选择没有效果，因为此时已经达到了最高频率。

DCO 可以工作到很高的频率，内部电阻的正常值大约为 200kΩ，此时 DCO 的工作频率大约为 5MHz。当使用一个100kΩ 的外部电阻时，此时 DCO 的工作频率可以达到 10 MHz。不过需要注意的是 MCLK 不能超过数据手册所规定的最大频率，即使 DCO 可以超过所规定的最大频率。

通过对 3 种振荡器的介绍，对 MSP430F14XX 的时钟模块有了基本的认识，下面具体介绍时钟模块的寄存器，以便能实现对时钟的正确选择。时钟模块的寄存器主要有：DCOCTL、BCSCTL1 和 BCSCTL2，下面对每个寄存器进行详细的介绍。

DCOCTL 寄存器：DCOCTL 寄存器的位分配如图 2-11 所示。

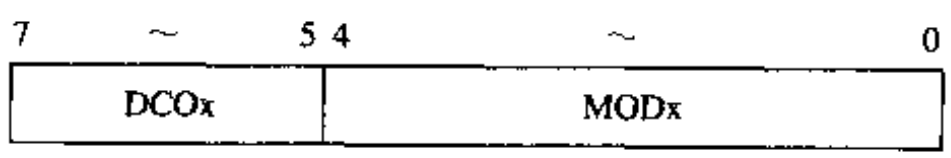



图 2-11　DCOCTL 寄存器

由图 2-11 可以看出，该寄存器主要有 8 个有效位，8 个有效位分成两组，其中 3 个位为 DCOx，5 个位为 MODx。为了能够对这两组位进行了解，下面分别对这两组位进行介绍。

- DCOx：DCO 的频率选择。这些位用来选择由 RSELx 定义的 8 个频率中的一个。
- MODx：调制选择。这些位定义了在 32 个 DCO 周期里插入 $f_{DCO+1}$ 频率，而在剩下的周期中采用 $f_{DCO}$ 频率。

BCSCTL1 寄存器：BCSCTL1 寄存器的位分配如图 2-12 所示。

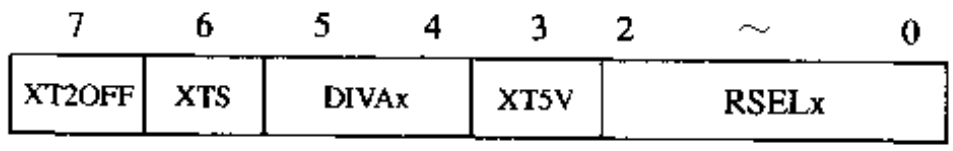



图 2-12　BCSCTL1 寄存器

由图 2-12 可以看出，该寄存器主要有几个有效位组成。为了能够对这些位进行了解，下面分别对这几个位进行介绍。

- XT2OFF：该位用来控制 XT2 振荡器的开启和关闭。当该位设置为 1，则开启 XT2 振荡器；当该位设置为 0，则关闭 XT2 振荡器。
- XTS：该位用来选择低速振荡器是低频模式还是高频模式。当该位设置为 1，为高频模式；当该位设置为 0，为低频模式。
- DIVAx：该位用来选择 ACLK 的分频系数。DIVAx 由两个位组成，因此它的值为 0、1、2、3，用 DIVA 来表示。当 DIVA 的值为 0 时，ACLK 的分频系数为 1；当 DIVA 的值为 1 时，ACLK 的分频系数为 2；当 DIVA 的值为 2 时，ACLK 的分频系数为 4；当 DIVA 的值为 3 时，ACLK 的分频系数为 8。
- XT5V：该位未用，必须选择复位。
- RSELx：电阻的选择。用来选择内部的电阻。当 RSELx 确定的值为 0 时，选择的频

率最低，当 RSELx 确定的值为 7 时，选择的频率最高。

BCSCTL2 寄存器：BCSCTL2 寄存器的位分配如图 2-13 所示。

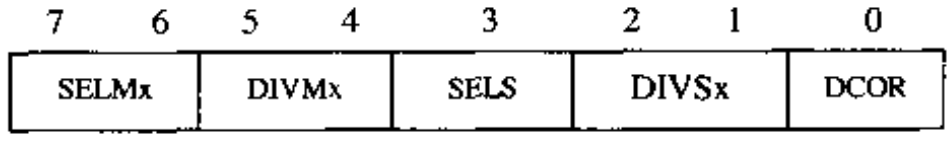



图 2-13　BCSCTL2 寄存器

由图 2-12 可以看出，该寄存器主要有几个有效位组成。为了能够对这些位进行了解，下面分别对这几个位进行介绍。

- SELMx：选择 MCLK 的时钟源。SELMx 由两个位组成，因此它的值为 0、1、2、3，用 SELM 来表示。当 SELM 的值为 0 时，MCLK 的时钟源为 DCOCLK；当 SELM 的值为 1 时，MCLK 的时钟源为 DCOCLK；当 SELM 的值为 2 时，MCLK 的时钟源为高速晶体振荡器；当 SELM 的值为 3 时，MCLK 的时钟源为低速晶体振荡器。
- DIVMx：MCLK 的分频因子。DIVMx 由两个位组成，因此它的值为 0、1、2、3，用 DIVM 来表示。当 DIVM 的值为 0 时，MCLK 的分频系数为 1；当 DIVM 的值为 1 时，MCLK 的分频系数为 2；当 DIVM 的值为 2 时，MCLK 的分频系数为 4；当 DIVM 的值为 3 时，MCLK 的分频系数为 8。
- SELS：选择 SMCLK 的时钟源。当该位设置为 0 时，SMCLK 的时钟源为 DCOCLK；当该位设置为 1 时，SMCLK 的时钟源为高速晶体振荡器。
- DIVSx：SMCLK 的分频因子。DIVSx 由两个位组成，因此它的值为 0、1、2、3，用 DIVS 来表示。当 DIVS 的值为 0 时，SMCLK 的分频系数为 1；当 DIVS 的值为 1 时，SMCLK 的分频系数为 2；当 DIVS 的值为 2 时，SMCLK 的分频系数为 4；当 DIVS 的值为 3 时，SMCLK 的分频系数为 8。
- DCOR：选择外部电阻还是内部电阻。当该位设置为 0 时，选择内部电阻；当该位设置为 1 时，选择外部电阻。

### 2.3.4　基础时钟与低功耗模块

MSP430家族主要是低功耗应用，它可以设置成不同的操作模式。操作模式的设置需要考虑3个不同的需求：低功耗应用、速度和数据处理要求和单个外围设备的最小电流消耗。根据系统的需要可以在某些场合时系统进入低功耗模式，在进入低功耗模式后，系统的时钟会停止，甚至主时钟都可以停止，这样系统的功耗只有μA级。当系统处于低功耗的时候，外部中断可以将系统从低功耗模式唤醒来执行相应的操作。MSP430系列芯片可以在6μs内从低功耗模式迅速进入到活动模式。当系统进入低功耗的时候，所有的I/O断口、RAM和寄存器的内容不会发生变化，并且所有的中断都可以将系统从低功耗模式唤醒，从而进入中断服务程序，进行相应的处理。

MSP430 有一种活动模式和 5 种低功耗模式。MSP430 的工作模式主要通过状态寄存器中的 CPUOFF、OSCFF、SCG0 和 SGC1 等位来设置。设置这些控制位，所选择的工作模式立即有效，那些依赖被禁止的时钟的外围模块也停止工作，直到时钟恢复活动模式后，相应

的外围模块才能继续工作。当然也可以设置外围模块相关寄存器中的相应位来禁止外围模块的工作。为了了解具体模式的选择，表 2-6 给出了具体的位设置来选择相应的模式。

表 2-6　控制位与工作模式

| SCG1 | SCG0 | OSCOFF | CPUOFF | 模式 | CPU 和时钟状态 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 活动 | CPU 激活、所有时钟激活 |
| 0 | 0 | 0 | 1 | LPM0 | CPU、MCLK 停止；SMCLK、ACLK 活动 |
| 0 | 1 | 0 | 1 | LPM1 | CPU、MCLK、DCO 停止；SMCLK、ACLK 活动 |
| 1 | 0 | 0 | 1 | LPM2 | CPU、MCLK、SMCLK、DCO 停止；DC 产生使能；SMCLK、ACLK 活动 |
| 1 | 1 | 0 | 1 | LPM3 | CPU、MCLK、SMCLK、DCO 停止；DC 产生禁止；SMCLK、ACLK 活动 |
| 1 | 1 | 1 | 1 | LPM4 | CPU 所有时钟停止 |

通过表 2-6 可以看出，适当设置控制位，就可以进入相应的工作模式，具体的工作模式选择主要根据系统的需要来确定。

### 2.3.5　时钟系统例子

根据前面部分的介绍，适当设置相应的寄存器就可以获得 SMCLK、MCLK 和 ACLK 时钟信号。下面给出一个时钟系统的例子，来具体说明时钟模块寄存器的设置，具体的程序如下：

```
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
}
```

由上面的程序结合前面介绍时钟模块寄存器就可以知道，只需要适当设置 BCSCTL1 和 BCSCTL2 寄存器的相应位就可以获得需要的 MCLK、SMCLK 和 ACLK 时钟信号。

## 2.4　MSP430F1XX 的端口

MSP430F1XX 系列单片机最多有 6 个 I/O 口：P1～P6，每个端口有 8 个管脚。每个管脚可以单独设置成输入或者输出方向，并且每个管脚都可以进行单独的读或者写。P1 口和 P2 口具有中断功能，P1 口和 P2 口的每个管脚都可以单独设置成中断，并且可以设置成上升沿或者下降沿触发中断。P1 口的所有管脚共用一个中断向量，同样 P2 口的所有管脚也共用一个中断向量。MSP430F1XX 系列单片机的 I/O 口主要有以下特征：

- 每个 I/O 口可以独立编程设置。
- 输入输出可以任意结合使用。
- P1 口和 P2 口的中断功能可以单独设置。
- 有独立的输入输出寄存器。

下面就具体的端口进行详细的介绍。

### 2.4.1　MSP430F1XX 的 P1 口

P1 口的每个管脚都可以设置成输入或者输入方向，并且可以实现任意的输入输出的组合。P1 口具有中断功能，每个管脚都可以单独设置中断方式，中断的触发可以设置成上升沿触发，也可以设置成下降沿触发。P1 口的功能设置是主要设置 P1DIR、P1IE、P1IES、P1IFG、P1IN、P1OUT 和 P1SEL 共 7 个寄存器。下面详细介绍各个寄存器。

P1DIR 寄存器：该寄存器控制 P1 口各个管脚的方向。设置相应的位为 1，则相应的管脚为输出，如果设置相应的位为 0，则相应的管脚为输入。P1DIR 寄存器的位分配如图 2-14 所示。

| P1DIR.7 | P1DIR.6 | P1DIR.5 | P1DIR.4 | P1DIR.3 | P1DIR.2 | P1DIR.1 | P1DIR.0 |
|---|---|---|---|---|---|---|---|

图 2-14　P1DIR 寄存器

由图 2-14 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚的输入输出方向的控制。

P1IE 寄存器：该寄存器控制 P1 口各个管脚的中断使能。设置相应的位为 1，则相应的管脚具有中断功能，如果设置相应的位为 0，则相应的管脚没有中断功能。P1IE 寄存器的位分配如图 2-15 所示。

| P1IE.7 | P1IE.6 | P1IE.5 | P1IE.4 | P1IE.3 | P1IE.2 | P1IE.1 | P1IE.0 |
|---|---|---|---|---|---|---|---|

图 2-15　P1IE 寄存器

由图 2-15 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚的中断功能的控制。

P1IES 寄存器：P1 口的中断触发沿选择寄存器。设置相应的位为 1，选择下降沿触发方式，如果设置相应的位为 0，选择上降沿触发方式。P1IES 寄存器的位分配如图 2-16 所示。

| P1IES.7 | P1IES.6 | P1IES.5 | P1IES.4 | P1IES.3 | P1IES.2 | P1IES.1 | P1IES.0 |
|---|---|---|---|---|---|---|---|

图 2-16　P1IES 寄存器

由图 2-16 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚中断功能触发方式的选择。

P1IFG 寄存器：P1 口的中断标志寄存器。如果 P1 口的某个管脚设置成中断管脚，当该有中断触发时，相应的位为 1，如果没有中断触发，则相应的位为 0。P1IFG 寄存器的位分配如图 2-17 所示。

| P1IFG.7 | P1IFG.6 | P1IFG.5 | P1IFG.4 | P1IFG.3 | P1IFG.2 | P1IFG.1 | P1IFG.0 |
|---|---|---|---|---|---|---|---|

图 2-17　P1IFG 寄存器

由图 2-17 可以看出，该寄存器的每个位可以单独读取，从而判断相应的管脚上是否有中断产生。

P1IN 寄存器：P1 口的输入寄存器。在输入的模式下，读取该寄存器的相应位来获得相应管脚上的数据。P1IN 寄存器的位分配如图 2-18 所示。

| P1IN.7 | P1IN.6 | P1IN.5 | P1IN.4 | P1IN.3 | P1IN.2 | P1IN.1 | P1IN.0 |
|---|---|---|---|---|---|---|---|

图 2-18　P1IN 寄存器

由图 2-18 可以看出，该寄存器的每个位可以单独读取，从而获得相应管脚上的输入数据或者管脚的状态。

P1OUT 寄存器：P1 口的输出寄存器。在输出模式下，如果该寄存器的相应位设置为 1 时，则相应的管脚输出高电平，如果设置该寄存器的相应的位为 0 时，则相应的管脚输出低电平。P1OUT 寄存器的位分配如图 2-19 所示。

| P1OUT.7 | P1OUT.6 | P1OUT.5 | P1OUT.4 | P1OUT.3 | P1OUT.2 | P1OUT.1 | P1OUT.0 |
|---|---|---|---|---|---|---|---|

图 2-19　P1OUT 寄存器

由图 2-19 可以看出，该寄存器的每个位可以单独设置，从而在相应的管脚输出低电平或者高电平。

P1SEL 寄存器：P1 口的功能选择寄存器。该寄存器主要是控制 P1 口的 I/O 管脚作为一般 I/O 口还是外围模块的功能端口。当该寄器的相应位设置为 1 时，则相应的管脚为外围模块的功能管脚，当该寄器的相应位设置为 0 时，则相应的管脚为一般 I/O 管脚。P1SEL 寄存器的位分配如图 2-20 所示。

| P1SEL.7 | P1SEL.6 | P1SEL.5 | P1SEL.4 | P1SEL.3 | P1SEL.2 | P1SEL.1 | P1SEL.0 |
|---|---|---|---|---|---|---|---|

图 2-20　P1SEL 寄存器

由图 2-20 可以看出，该寄存器的每个位可以单独设置，从而对相应的管脚进行一般 I/O

口还是外围模块功能管脚的选择。

## 2.4.2　MSP430F1XX 的 P2 口

P2 口的每个管脚都可以设置成输入或者输入方向，并且可以实现任意的输入输出的组合。P2 口具有中断功能，每个管脚都可以单独设置中断方式，中断的触发可以设置成上升沿触发，也可以设置成下降沿触发。P2 口的功能设置是主要设置 P2DIR、P2IE、P2IES、P2IFG、P2IN、P2OUT 和 P2SEL 共 7 个寄存器。下面详细介绍各个寄存器。

P2DIR 寄存器：该寄存器控制 P2 口各个管脚的方向。设置相应的位为 1，则相应的管脚为输出，如果设置相应的位为 0，则相应的管脚为输入。P2DIR 寄存器的位分配如图 2-21 所示。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| P2DIR.7 | P2DIR.6 | P2DIR.5 | P2DIR.4 | P2DIR.3 | P2DIR.2 | P2DIR.1 | P2DIR.0 |

图 2-21　P2DIR 寄存器

由图 2-21 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚的输入输出方向的控制。

P2IE 寄存器：该寄存器控制 P2 口各个管脚的中断使能。设置相应的位为 1，则相应的管脚具有中断功能，如果设置相应的位为 0，则相应的管脚没有中断功能。P2IE 寄存器的位分配如图 2-22 所示。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| P2IE.7 | P2IE.6 | P2IE.5 | P2IE.4 | P2IE.3 | P2IE.2 | P2IE.1 | P2IE.0 |

图 2-22　P2IE 寄存器

由图 2-22 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚的中断功能的控制。

P2IES 寄存器：P2 口的中断触发沿选择寄存器。设置相应的位为 1，选择下降沿触发方式，如果设置相应的位为 0，选择上降沿触发方式。P2IES 寄存器的位分配如图 2-23 所示。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| P2IES.7 | P2IES.6 | P2IES.5 | P2IES.4 | P2IES.3 | P2IES.2 | P2IES.1 | P2IES.0 |

图 2-23　P2IES 寄存器

由图 2-23 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚中断功能触发方式的选择。

P2IFG 寄存器：P2 口的中断标志寄存器。如果 P2 口的当某个管脚设置成中断管脚时，当该有中断触发时，相应的位为 1，如果没有中断触发，则相应的位为 0。P2IFG 寄存器的位分配如图 2-24 所示。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| P2IFG.7 | P2IFG.6 | P2IFG.5 | P2IFG.4 | P2IFG.3 | P2IFG.2 | P2IFG.1 | P2IFG.0 |

图 2-24　P2IFG 寄存器

由图 2-24 可以看出，该寄存器的每个位可以单独读取，从而判断相应的管脚上是否有中断产生。

P2IN 寄存器：P2 口的输入寄存器。在输入的模式下，读取该寄存器的相应位来获得相应管脚上的数据。P2IN 寄存器的位分配如图 2-25 所示。

| P2IN.7 | P2IN.6 | P2IN.5 | P2IN.4 | P2IN.3 | P2IN.2 | P2IN.1 | P2IN.0 |
|---|---|---|---|---|---|---|---|

图 2-25　P2IN 寄存器

由图 2-25 可以看出，该寄存器的每个位可以单独读取，从而获得相应管脚上的输入数据或者管脚的状态。

P2OUT 寄存器：P2 口的输出寄存器。在输出模式下，如果该寄存器的相应位设置为 1 时，则相应的管脚输出高电平，如果设置该寄存器的相应的位为 0 时，则相应的管脚输出低电平。P2OUT 寄存器的位分配如图 2-26 所示。

| P2OUT.7 | P2OUT.6 | P2OUT.5 | P2OUT.4 | P2OUT.3 | P2OUT.2 | P2OUT.1 | P2OUT.0 |
|---|---|---|---|---|---|---|---|

图 2-26　P2OUT 寄存器

由图 2-26 可以看出，该寄存器的每个位可以单独设置，从而在相应的管脚输出低电平或者高电平。

P2SEL 寄存器：P2 口的功能选择寄存器。该寄存器主要是控制 P2 口的 I/O 管脚作为一般 I/O 口还是外围模块的功能端口。当该寄器的相应位设置为 1 时，则相应的管脚为外围模块的功能管脚，当该寄器的相应位设置为 0 时，则相应的管脚为一般 I/O 管脚。P2SEL 寄存器的位分配如图 2-27 所示。

| P2SEL.7 | P2SEL.6 | P2SEL.5 | P2SEL.4 | P2SEL.3 | P2SEL.2 | P2SEL.1 | P2SEL.0 |
|---|---|---|---|---|---|---|---|

图 2-27　P2SEL 寄存器

由图 2-27 可以看出，该寄存器的每个位可以单独设置，从而对相应的管脚进行一般 I/O 口还是外围模块功能管脚的选择。

### 2.4.3　MSP430F1XX 的 P3 口

P3口的每个管脚都可以设置成输入或者输入方向，并且可以实现任意的输入输出的组合。P3口的功能设置是主要设置P3DIR、P3IN、P3OUT和P3SEL共4个寄存器。下面详细介绍各个寄存器。

P3DIR 寄存器：该寄存器控制 P3 口各个管脚的方向。设置相应的位为 1，则相应的管脚为输出，如果设置相应的位为 0，则相应的管脚为输入。P3DIR 寄存器的位分配如图 2-28 所示。

| P3DIR.7 | P3DIR.6 | P3DIR.5 | P3DIR.4 | P3DIR.3 | P3DIR.2 | P3DIR.1 | P3DIR.0 |
|---|---|---|---|---|---|---|---|

图 2-28　P3DIR 寄存器

由图 2-28 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚的输入输出方向的控制。

**P3IN 寄存器**：P3 口的输入寄存器。在输入的模式下，读取该寄存器的相应位来获得相应管脚上的数据。P3IN 寄存器的位分配如图 2-29 所示。

| P3IN.7 | P3IN.6 | P3IN.5 | P3IN.4 | P3IN.3 | P3IN.2 | P3IN.1 | P3IN.0 |
|---|---|---|---|---|---|---|---|

图 2-29　P3IN 寄存器

由图 2-29 可以看出，该寄存器的每个位可以单独读取，从而获得相应管脚上的输入数据或者管脚的状态。

**P3OUT 寄存器**：P3 口的输出寄存器。在输出模式下，如果该寄存器的相应位设置为 1 时，则相应的管脚输出高电平，如果设置该寄存器的相应的位为 0 时，则相应的管脚输出低电平。P3OUT 寄存器的位分配如图 2-30 所示。

| P3OUT.7 | P3OUT.6 | P3OUT.5 | P3OUT.4 | P3OUT.3 | P3OUT.2 | P3OUT.1 | P3OUT.0 |
|---|---|---|---|---|---|---|---|

图 2-30　P3OUT 寄存器

由图 2-30 可以看出，该寄存器的每个位可以单独设置，从而在相应的管脚输出低电平或者高电平。

**P3SEL 寄存器**：P3 口的功能选择寄存器。该寄存器主要是控制 P3 口的 I/O 管脚作为一般 I/O 口还是外围模块的功能端口。当该寄器的相应位设置为 1 时，则相应的管脚为外围模块的功能管脚，当该寄器的相应位设置为 0 时，则相应的管脚为一般 I/O 管脚。P3SEL 寄存器的位分配如图 2-31 所示。

| P3SEL.7 | P3SEL.6 | P3SEL.5 | P3SEL.4 | P3SEL.3 | P3SEL.2 | P3SEL.1 | P3SEL.0 |
|---|---|---|---|---|---|---|---|

图 2-31　P3SEL 寄存器

由图 2-31 可以看出，该寄存器的每个位可以单独设置，从而对相应的管脚进行一般 I/O 口还是外围模块功能管脚的选择。

### 2.4.4　MSP430F1XX 的 P4 口

P4口的每个管脚都可以设置成输入或者输入方向，并且可以实现任意的输入输出的组合。P4口的功能设置是主要设置P4DIR、P4IN、P4OUT和P4SEL共4个寄存器。下面详细介绍各个寄存器。

**P4DIR 寄存器**：该寄存器控制 P4 口各个管脚的方向。设置相应的位为 1，则相应的管脚为输出，如果设置相应的位为 0，则相应的管脚为输入。P4DIR 寄存器的位分配如图 2-32 所示。

| P4DIR.7 | P4DIR.6 | P4DIR.5 | P4DIR.4 | P4DIR.3 | P4DIR.2 | P4DIR.1 | P4DIR.0 |
|---|---|---|---|---|---|---|---|

图 2-32　P4DIR 寄存器

由图 2-32 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚的输入输出方向的控制。

**P4IN 寄存器**：P4 口的输入寄存器。在输入的模式下，读取该寄存器的相应位来获得相应管脚上的数据。P4IN 寄存器的位分配如图 2-33 所示。

| P4IN.7 | P4IN.6 | P4IN.5 | P4IN.4 | P4IN.3 | P4IN.2 | P4IN.1 | P4IN.0 |
|---|---|---|---|---|---|---|---|

图 2-33　P4IN 寄存器

由图 2-33 可以看出，该寄存器的每个位可以单独读取，从而获得相应管脚上的输入数据或者管脚的状态。

**P4OUT 寄存器**：P4 口的输出寄存器。在输出模式下，如果该寄存器的相应位设置为 1 时，则相应的管脚输出高电平，如果设置该寄存器的相应的位为 0 时，则相应的管脚输出低电平。P4OUT 寄存器的位分配如图 2-34 所示。

| P4OUT.7 | P4OUT.6 | P4OUT.5 | P4OUT.4 | P4OUT.3 | P4OUT.2 | P4OUT.1 | P4OUT.0 |
|---|---|---|---|---|---|---|---|

图 2-34　P4OUT 寄存器

由图 2-34 可以看出，该寄存器的每个位可以单独设置，从而在相应的管脚输出低电平或者高电平。

**P4SEL 寄存器**：P4 口的功能选择寄存器。该寄存器主要是控制 P4 口的 I/O 管脚作为一般 I/O 口还是外围模块的功能端口。当该寄器的相应位设置为 1 时，则相应的管脚为外围模块的功能管脚，当该寄器的相应位设置为 0 时，则相应的管脚为一般 I/O 管脚。P4SEL 寄存器的位分配如图 2-35 所示。

| P4SEL.7 | P4SEL.6 | P4SEL.5 | P4SEL.4 | P4SEL.3 | P4SEL.2 | P4SEL.1 | P4SEL.0 |
|---|---|---|---|---|---|---|---|

图 2-35　P4SEL 寄存器

由图 2-35 可以看出，该寄存器的每个位可以单独设置，从而对相应的管脚进行一般 I/O 口还是外围模块功能管脚的选择。

## 2.4.5　MSP430F1XX 的 P5 口

P5 口的每个管脚都可以设置成输入或者输入方向，并且可以实现任意的输入输出的组合。P5 口的功能设置是主要设置 P5DIR、P5IN、P5OUT 和 P5SEL 共 4 个寄存器。下面详细介绍各个寄存器。

**P5DIR 寄存器**：该寄存器控制 P5 口各个管脚的方向。设置相应的位为 1，则相应的管脚为输出，如果设置相应的位为 0，则相应的管脚为输入。P5DIR 寄存器的位分配如图 2-36 所示。

| P5DIR.7 | P5DIR.6 | P5DIR.5 | P5DIR.4 | P5DIR.3 | P5DIR.2 | P5DIR.1 | P5DIR.0 |
|---|---|---|---|---|---|---|---|

图 2-36　P5DIR 寄存器

由图 2-36 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚的输入输出方向的控制。

**P5IN 寄存器**：P5 口的输入寄存器。在输入的模式下，读取该寄存器的相应位来获得相应管脚上的数据。P5IN 寄存器的位分配如图 2-37 所示。

| P5IN.7 | P5IN.6 | P5IN.5 | P5IN.4 | P5IN.3 | P5IN.2 | P5IN.1 | P5IN.0 |
|---|---|---|---|---|---|---|---|

图 2-37　P5IN 寄存器

由图 2-37 可以看出，该寄存器的每个位可以单独读取，从而获得相应管脚上的输入数据或者管脚的状态。

**P5OUT 寄存器**：P5 口的输出寄存器。在输出模式下，如果该寄存器的相应位设置为 1 时，则相应的管脚输出高电平，如果设置该寄存器的相应的位为 0 时，则相应的管脚输出低电平。P5OUT 寄存器的位分配如图 2-38 所示。

| P5OUT.7 | P5OUT.6 | P5OUT.5 | P5OUT.4 | P5OUT.3 | P5OUT.2 | P5OUT.1 | P5OUT.0 |
|---|---|---|---|---|---|---|---|

图 2-38　P5OUT 寄存器

由图 2-38 可以看出，该寄存器的每个位可以单独设置，从而在相应的管脚输出低电平或者高电平。

**P5SEL 寄存器**：P5 口的功能选择寄存器。该寄存器主要是控制 P5 口的 I/O 管脚作为一般 I/O 口还是外围模块的功能端口。当该寄器的相应位设置为 1 时，则相应的管脚为外围模块的功能管脚，当该寄器的相应位设置为 0 时，则相应的管脚为一般 I/O 管脚。P5SEL 寄存器的位分配如图 2-39 所示。

| P5SEL.7 | P5SEL.6 | P5SEL.5 | P5SEL.4 | P5SEL.3 | P5SEL.2 | P5SEL.1 | P5SEL.0 |
|---|---|---|---|---|---|---|---|

图 2-39　P5SEL 寄存器

由图 2-39 可以看出，该寄存器的每个位可以单独设置，从而对相应的管脚进行一般 I/O 口还是外围模块功能管脚的选择。

### 2.4.6　MSP430F1XX 的 P6 口

P6口的每个管脚都可以设置成输入或者输入方向，并且可以实现任意的输入输出的组合。P6口的功能设置是主要设置P6DIR、P6IN、P6OUT和P6SEL共4个寄存器。下面详细介绍各个寄存器。

**P6DIR 寄存器**：该寄存器控制 P6 口各个管脚的方向。设置相应的位为 1，则相应的管脚为输出，如果设置相应的位为 0，则相应的管脚为输入。P6DIR 寄存器的位分配如图 2-40 所示。

| P6DIR.7 | P6DIR.6 | P6DIR.5 | P6DIR.4 | P6DIR.3 | P6DIR.2 | P6DIR.1 | P6DIR.0 |
|---|---|---|---|---|---|---|---|

图 2-40　P6DIR 寄存器

由图 2-40 可以看出，该寄存器的每个位可以单独设置，从而实现对每个管脚的输入输出方向的控制。

**P6IN 寄存器**：P6 口的输入寄存器。在输入的模式下，读取该寄存器的相应位来获得相应管脚上的数据。P6IN 寄存器的位分配如图 2-41 所示。

| P6IN.7 | P6IN.6 | P6IN.5 | P6IN.4 | P6IN.3 | P6IN.2 | P6IN.1 | P6IN.0 |
|---|---|---|---|---|---|---|---|

图 2-41　P6IN 寄存器

由图 2-41 可以看出，该寄存器的每个位可以单独读取，从而获得相应管脚上的输入数据或者管脚的状态。

**P6OUT寄存器**：P6口的输出寄存器。在输出模式下，如果该寄存器的相应位设置为1时，则相应的管脚输出高电平，如果设置该寄存器的相应的位为0时，则相应的管脚输出低电平。P6OUT寄存器的位分配如图2-42所示。

| P6OUT.7 | P6OUT.6 | P6OUT.5 | P6OUT.4 | P6OUT.3 | P6OUT.2 | P6OUT.1 | P6OUT.0 |
|---|---|---|---|---|---|---|---|

图 2-42　P6OUT 寄存器

由图 2-42 可以看出，该寄存器的每个位可以单独设置，从而在相应的管脚输出低电平或者高电平。

**P6SEL寄存器**：P6口的功能选择寄存器。该寄存器主要是控制P6口的I/O管脚作为一般I/O口还是外围模块的功能端口。当该寄器的相应位设置为1时，则相应的管脚为外围模块的功能管脚，当该寄器的相应位设置为0时，则相应的管脚为一般I/O管脚。P6SEL寄存器的位分配如图2-43所示。

| P6SEL.7 | P6SEL.6 | P6SEL.5 | P6SEL.4 | P6SEL.3 | P6SEL.2 | P6SEL.1 | P6SEL.0 |
|---|---|---|---|---|---|---|---|

图 2-43　P6SEL 寄存器

由图 2-43 可以看出，该寄存器的每个位可以单独设置，从而对相应的管脚进行一般 I/O 口还是外围模块功能管脚的选择。

## 2.4.7　MSP430F1XX 各种端口应用例子

根据前面对 MSP430F1XX 各种端口的介绍，对各个端口的设置、操作有了深入了解。结合前面的介绍，下面讨论各种端口的应用程序。端口的应用主要包括 3 个方面：一般 I/O 口、中断功能口和外围模块特殊功能口。下面就各个方面给出具体的应用例子。

端口作为一般 I/O 口使用的例子：

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

```
    P2DIR = 0;
    P3DIR = 0;

    //将所有的管脚设置为一般I/O口
    P3SEL = 0;
    P2SEL = 0;
    P1SEL = 0;

    //将P2.1和P2.2设置为输出方向
    P2DIR |= BIT1;
    P2DIR |= BIT2;
    //将P3.0和P3.1设置为输出方向
    P3DIR |= BIT0;
    P3DIR |= BIT1;
    //将P2.7设置为输出方向
    P2DIR |= BIT7;
    //输出高电平
    P2OUT |= BIT7;
    //读P1口的数据
    chrTemp = (char)(P1IN);
    return;
}
```

通过以上程序可以看出，作为一般 I/O 口使用时，需要设置好 PxDIR、PxSEL 等寄存器，然后从 PxIN 等寄存器读取输入到管脚的数据，设置 PxOUT 的某一个位或者整个寄存器向外输出数据。

端口作为外围模块的特殊功能管脚：

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

    U1TCTL = 0X00;    //将寄存器的内容清零
    U1TCTL += SSEL1; //波特率发生器选择SMCLK

    UBR0_1 = 0X45;    //波特率为115200
    UBR1_1 = 0X00;
    UMCTL_1 = 0X00;

    ME2 |= UTXE1 + URXE1; //使能UART1的TXD和RXD
    IE2 |= URXIE1;    //使能UART1的RX中断
    IE2 |= UTXIE1;    //使能UART1的TX中断
    P3SEL = 0;        //将所有的管脚设置为一般I/O口
    P3SEL |= BIT6;    //设置P3.6为UART1的TXD
    P3SEL |= BIT7;    //设置P3.7为UART1的RXD
```

```
    P3DIR |= BIT6;    //P3.6为输出管脚
    return;
}
```

通过上面的程序可以看出，需要设置PxSEL的某些位为1，使I/O口作为外围模块的特殊功能管脚。

端口作为中断功能管脚：

```
void Init_IntPort(void)
{
    //初始化
    P1DIR = 0;
    P1SEL = 0;
    //将中断寄存器清零
    P1IE = 0;
    P1IES = 0;
    P1IFG = 0;

    P1IE |= BIT4;     //管脚P1.4使能中断
    P1IES |= BIT4;    //对应的管脚由高到低电平跳变使相应的标志置位
    P1IE |= BIT5;     //管脚P1.5使能中断
    P1IES |= BIT5;    //对应的管脚由高到低电平跳变使相应的标志置位
}
```

通过上面的程序可以看出，需要设置P1IE（或者P2IE）来使能中断功能，设置P1IES（或者P2IES）来选择中断的触发方式。端口的中断可以采用中断服务程序处理，下面为端口的中断服务的程序。

```
interrupt [PORT1_VECTOR] void R_B_ISR(void)
{
    int i;

    if(P1IFG & BIT4)
    {
        //处理中断
        P1IFG &= ~(BIT4);        //清除中断标志位
        for(i = 100;i > 0;i--); //延迟一点时间
    }

    if(P1IFG & BIT5)
    {
        //处理中断
        P1IFG &= ~(BIT5);        //清除中断标志位
        for(i = 100;i > 0;i--); //延迟一点时间
    }
}
```

通过上面的程序可以看出，P1 口或者 P2 口的端口中断使用一个中断向量，这样在中断处理程序中需要判断具体是哪个端口触发的中断，另外也要求软件清除中断标志。

## 2.5　定时器

定时器在单片机系统中是非常重要的部分，它在事件控制与管理方面有着重要的应用。MSP430F1XX 主要有看门狗、定时器 A（Timer_A）和定时器 B（Timer_B）。下面分别介绍各个模块。

### 2.5.1　看门狗

看门狗本质上是一个16位的定时器，它可以用作看门狗也可以用作定时器。看门狗的主要功能就是检测到软件出现问题时重新启动系统。在看门狗设置的时间到时，会产生一个系统的复位信号。借助看门狗的这种功能就可以很容易防止程序出错的情况发生。只需要打开看门狗，设置看门狗溢出的时间间隔，在软件设计的时候估计在看门狗可能会溢出的地方清除看门狗定时器的内容，程序在正常的情况下不会发生看门狗溢出的情况，也就不会产生系统复位信号；当程序出现异常的时候，就没有地方能清除看门狗定时器的内容，看门狗在设置时间到来时就会产生系统复位信号，重新启动系统，从而程序正常运行。由此可见，看门狗可以监测系统，保证系统的正常运行。另外，如果看门狗不设置成看门狗模式时，也可以作为一般的定时器用。MSP430F1XX 系列单片机的看门狗有如下特色：

- 有 4 个可选的时间间隔设置，通过软件来设置。
- 有看门狗模式和定时器模式。
- 访问看门狗控制寄存器受密码保护。
- 控制 $\overline{\text{RST}}$/NMI 管脚。
- 可以选择时钟源。
- 能够停止，以便进一步降低功耗。

看门狗的控制和功能实现主要是通过设置WDTCTL寄存器来实现。下面详细介绍WDTCTL寄存器，该寄存器的位分配如图 2-44 所示。

| 口令字节 | HOLD | NMIES | NMI | TMSEL | CNTCL | SEL | IS.1　IS.0 |
|---|---|---|---|---|---|---|---|

图 2-44　WDTCTL 寄存器

为了增加对看门狗的了解，下面对 WDTCTL 寄存器的各个位进行详细介绍。

口令字节：该口令字节为 WDTCTL 寄存器的高 8 位。作为访问 WDTCTL 寄存器的口令，如果口令不对，就不能访问该寄存器。当读口令字节时，读出的内容总是 69H。如果要设置 WDTCTL 寄存器，则必须输入口令，口令为 5AH，如果口令错误，会产生系统复位信号。

HOLD：该位用于停止看门狗工作。当设置该位为 1 时，则时钟输入禁止，看门狗停止；当设置该位为 0 时，看门狗处于工作状态。

NMIES：看门狗定时器的 NMI 中断触发沿选择。当该位为 1 时，NMI 中断由下降沿触发；当该位为 0 时，NMI 中断由上升沿触发。

NMI：$\overline{\text{RST}}$/NMI 管脚的选择。当该位为 1 时，该管脚作为 NMI 管脚；当该位为 0 时，该管脚作为 $\overline{\text{RST}}$ 管脚。

TMSEL：看门狗的工作模式的选择。当该位为 1 时，选择定时器模式；当该位为 0 时，选择看门狗模式。

CNTCL：清除看门狗的计数器（WDTCNT）。当该位为 1 时，清除 WDTCNT 寄存器的内容；当该位为 0 时，不影响 WDTCNT 寄存器的内容。

SEL：看门狗时钟源的选择。当该位为 1 时，选择 ACLK 作为看门狗的时钟信号；当该位为 0 时，选择 SMCLK 作为看门狗的时钟信号。

IS1、IS0：看门狗时间间隔的选择。表 2-7 给出了 IS1、IS0 位与时间间隔选择的关系。

表 2-7　IS1、IS0 位与时间间隔选择的关系

| IS1 | IS0 | 选择的时间间隔 |
|---|---|---|
| 0 | 0 | 看门狗时钟源/32768 |
| 0 | 1 | 看门狗时钟源/8192 |
| 1 | 0 | 看门狗时钟源/512 |
| 1 | 1 | 看门狗时钟源/64 |

由表 2-7 可以看出，适当设置 IS1 和 IS0 就能获得相应的时间间隔。

关于看门狗的中断使能的设置是设置 IE1 寄存器的相应位，中断标志是在 IFG1 寄存器里，IE1 和 IFG1 已经在前面作了介绍，这里就不在介绍。

## 2.5.2　Timer_A

定时器 A 是一个 16 位的定时器/计数器。它有 3 个捕获/比较寄存器。定时器 A 能支持多个时序控制、多个捕获/比较功能和多个 PWM 输出。定时器 A 有广泛的中断功能，中断可以由计数器溢出产生，也可以由捕获/比较寄存器产生。定时器 A 主要有以下的特点：

- 16 位的计数器/定时器，共有 4 种模式。
- 可以选择设置时钟源。
- 多个捕获/比较寄存器（MSP430F14X 有 3 个）。
- 异步的输入输出锁存。
- 具有中断向量寄存器，能快速译码定时器 A 产生的中断。

用户对定时器 A 的所有操作都是通过操作该模块的寄存器完成的。定时器 A 的寄存器主要有 TACTL、TAR、TAIV、CCTLn 和 CCRn（n 的值可能是 0、1 或者 2）。下面结合具体寄存器的介绍来讲解定时器 A 的功能。

### 1．TACTL 寄存器

TACTL 寄存器是一个 16 位的寄存器。通过设置该寄存器完成对 Timer_A 作为定时器使用的控制，它包含了 Timer_A 作为定时器使用的所有控制位。图 2-45 给出了 TACTL 寄存器的各个位。

| 10~15 未用 | SSEL1 | SSEL0 | ID1 | ID0 | MC1 | MC0 | 未用 | CLR | TAIE | TAIFG |
|---|---|---|---|---|---|---|---|---|---|---|

图 2-45　TACTL 寄存器

由图 2-45 可以看出，该寄存器的 10～15 位目前没有用到。为了增加对 TACTL 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 TACTL 寄存器的各个位进行详细介绍。

SSEL1、SSEL0：定时器 A 的时钟源的选择。表 2-8 给出了 SSEL1、SSEL0 位与时钟源的选择的关系。

表 2-8　定时器 A 的时钟源的选择

| SSEL1 | SSEL0 | 定时器 A 的时钟源 |
|---|---|---|
| 0 | 0 | TACLK（使用外部管脚信号作为输入） |
| 0 | 1 | ACLK |
| 1 | 0 | MCLK |
| 1 | 1 | INCLK（外部输入时钟） |

由表 2-8 可以看出：通过设置 SSEL1 和 SSEL0 的值可以完成对定时器 A 的时钟源的选择。

ID1、ID0：这两个位来选择输入时钟的分频系数。表 2-9 给出了 ID1、ID0 位与输入时钟的分频系数选择的关系。

表 2-9　定时器 A 的分频系数的选择

| ID1 | ID0 | 定时器 A 的分频系数 |
|---|---|---|
| 0 | 0 | 直通，时钟不分频 |
| 0 | 1 | 1/2 分频 |
| 1 | 0 | 1/4 分频 |
| 1 | 1 | 1/8 分频 |

由表 2-9 可以看出：通过设置 ID1 和 ID0 的值可以完成对定时器 A 的输入时钟的分频系数的选择。

MC1、MC0：这两个位来选择 Timer_A 作为定时器的工作模式。表 2-10 给出了 MC1、MC0 位与工作模式选择的关系。

表 2-10　定时器 A 的工作模式的选择

| MC1 | MC0 | 定时器 A 的工作模式 |
|---|---|---|
| 0 | 0 | 停止模式 |
| 0 | 1 | 增模式 |
| 1 | 0 | 连续模式 |
| 1 | 1 | 增减模式 |

由表 2-10 可以看出：通过设置 MC1 和 MC0 的值可以完成对定时器 A 的作为定时器使用时工作模式的选择。

CLR：定时器 A 的清除控制。设置该位为 1 时，TAR 寄存器里的内容、计数器方向等内容被清除。

TAIE：Timer_A 的中断使能。当该位为 1 时，允许 Timer_A 的中断，当该位为 0 时，不

允许 Timer_A 的中断。

TAIFG：Timer_A 的中断标志。当该位为 1 时，有中断产生，当该位为 0 时，没有中断产生。

以上对 Timer_A 的 TACTL 寄存器的各个位进行了介绍，通过介绍知道 Timer_A 作为定时器使用时有 4 种工作模式，下面对定时器的工作模式进行简单的介绍。

停止模式：定时器处于暂停状态。当 Timer_A 处于暂停状态时，TAR 寄存器的内容不受影响。当 Timer_A 重新启动时，计数器从原来的值进行记数，并且记数的方向与原来的方向一致。

增记数模式：计数器工作在增记数模式。该模式用于定时周期小于 65536 的连续记数方式，CCR0 寄存器里面的内容为定时器的记数周期数。计数器的值增加，当计数器的值增加到等于 CCR0 寄存器里面的内容时，计数器就重新从 0 开始记数。在模式选择为增记数的模式时，如果定时器的值大于或者等于 CCR0 寄存器里面的值时，计数器立即从 0 开始记数。当计数器的记数记到 CCR0 时，会设置 CCIFG 中断标志，当定时器从 CCR0 的值转到 0 时，设置 TAIFG 中断标志。

连续记数模式：计数器处于从 0～FFFFH 连续记数模式。该模式的定时周期为 65536 个时钟周期。在该模式下 CCR0 作为一般的捕获/比较寄存器使用。在连续记数模式下，计数器从 0 记数到 FFFFH，然后又回到 0 重新记数。当计数器记数从 FFFFH～0 时，设置 TAIFG 中断标志。连续模式可以用来产生独立的时间间隔并输出频率。每个时间间隔完成就会产生一个中断。在中断到来时，必须在当前中断服务程序里将下一个时间间隔加到 CCRx 寄存器里，这样顺序执行，就可以产生周期性的中断，从而向外输出频率，Timer_A 最多可以使用 3 个捕获/比较寄存器输出 3 个不同的频率。

增/减记数模式：计数器 TAR 的值先增后减。当计数器增记数到 CCR0 的值时，计数器变为减记数，当计数器减记数到 0 时，设置 TAIFG 中断标志位。在增/减记数模式下，记数的周期是 CCR0 寄存器值的两倍。增/减记数模式的计数方向是锁存的，这样可以允许计数器暂停和重新启动，在重新启动后，记数方向不发生变化，如果不是这样的，则可以设置 TACTL 寄存器里面的 CLR 位来清除 TAR 寄存器的内容。在增/减记数模式下，CCIFG 和 TAIFG 中断标志在一个周期内只设置一次。当计数器从 CCR0-1 增记数到 CCR0 时，设置 CCIFG 中断标志；当计数器从 1 减记数到 0 时，设置 TAIFG 中断标志。

**2．TAR 寄存器**

该寄存器是执行记数的单元，是计数器的主体。该寄存器是一个 16 位的寄存器，该寄存器的内容可读可写，但当计数器的时钟不是 MCLK 时，写入该寄存器的时候应该停止计数器的记数，因为它与 CPU 时钟不同步。

**3．TAIV 寄存器**

该寄存器为定时器 A 模块的中断向量寄存器。CCR1 的 CCIFG 中断标志、CCR2 的 CCIFG 中断标志和 TAIFG 中断标志使用一个中断向量，TAIV 寄存器就是用来判断是哪一个中断标志请求。TAIV 是一个 16 位的寄存器，该寄存器的位分配如图 2-46 所示。

| 5～15 为 0 | TAIVx（3 位） | 0 |
|---|---|---|

图 2-46　TAIV 寄存器

由图 2-46 可以看出，TAIV 寄存器使用了 3 位对中断向量进行编码，以便区别是哪一个中断请求，具体的编码如表 2-11 所示。

表 2-11　TAIV 中断向量值

| TAIV 内容 | 中断源 | 中断标志 | 中断优先级 |
|---|---|---|---|
| 00H | 没有中断标志 | — | |
| 02H | 捕获/比较 1 | CCR1 的 CCIFG | 最高 |
| 04H | 捕获/比较 2 | CCR2 的 CCIFG | |
| 06H | 保留将来使用 | — | |
| 08H | 保留将来使用 | — | |
| 0AH | 定时器溢出 | TAIFG | |
| 0CH | 保留将来使用 | — | |
| 0EH | 保留将来使用 | — | 最低 |

由表 2-11 可以看出：只要确定了 TAIV 寄存器的值，就可以很容易判断出是那个中断标志产生，从而进行相应的处理。在进行具体的中断处理时，按照优先级的高低进行处理，即优先级高的先处理。

**4．CCTLn 寄存器**

捕获/比较控制寄存器。该寄存器用来控制 Timer_A 作为捕获/比较模块时的控制功能，该寄存器的位分配如图 2-47 所示。

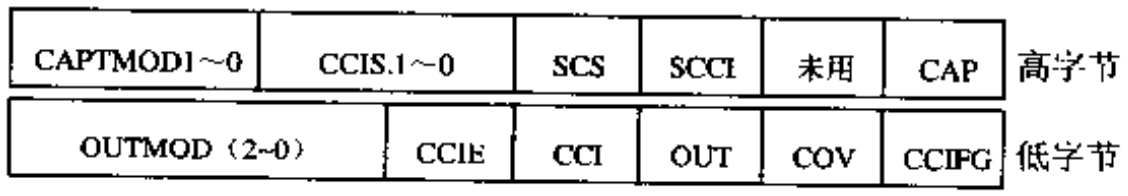



图 2-47　CCTLn 寄存器

为了增加对 CCTLn 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 CCTLn 寄存器的各个位进行详细介绍。

CAPTMOD1、CAPTMOD0：该两位选择捕获模式。表2-12给出了CAPTMOD1、CAPTMOD0位与捕获模式选择的关系。

表 2-12　捕获模式的选择

| CAPTMOD1 | CAPTMOD0 | 捕获模式的选择 |
|---|---|---|
| 0 | 0 | 禁止捕获模式 |
| 0 | 1 | 上升沿捕获模式 |
| 1 | 0 | 下降沿捕获模式 |
| 1 | 1 | 上升沿和下降沿捕获模式 |

由表 2-12 可以看出，适当设置 CAPTMOD1 和 CAPTMOD0 位就可以选择相应的捕获模式。

CCIS1、CCIS0：捕获/比较的输入选择。这些位用来选择 CCRx 的输入信号。这些位只

在捕获模式下使用，在比较模式下不使用。表 2-13 给出了 CCIS1、CCIS0 位对输入信号的选择。

表 2-13　输入信号的选择

| CCIS1 | CCIS0 | 输入信号的选择 |
|---|---|---|
| 0 | 0 | 选择 CCIxA 作为捕获事件的输入信号 |
| 0 | 1 | 选择 CCIxB 作为捕获事件的输入信号 |
| 1 | 0 | 选择 GND 作为捕获事件的输入信号 |
| 1 | 1 | 选择 VCC 作为捕获事件的输入信号 |

由表 2-13 可以看出，适当设置 CCIS1 和 CCIS0 位就可以对捕获模式的输入信号进行相应的选择。

SCS：同步捕获源。该位用来使捕获输入信号与定时器时钟信号同步。当该位为 1 时，为同步捕获；当该位为 0 时，为异步捕获。

SCCI：同步捕获/比较输入。被选择的 CCI 输入信号和 EQUx 锁存，通过 SCCI 读出来。

CAP：捕获模式选择。当该位为 1 时，该模块工作在捕获模式；该位为 0 时，该模块工作在比较模式。

OUTMOD0、OUTMOD1、OUTMOD2：输出模式。表 2-14 给出了 OUTMOD0、OUTMOD1、OUTMOD2 位与输出模式的关系。

表 2-14　输出模式的选择

| OUTMOD2 | OUTMOD1 | OUTMOD0 | 输出模式 |
|---|---|---|---|
| 0 | 0 | 0 | 输出 |
| 0 | 0 | 1 | 置位 |
| 0 | 1 | 0 | PWM 翻转/复位 |
| 0 | 1 | 1 | PWM 置位/复位 |
| 1 | 0 | 0 | 翻转 |
| 1 | 0 | 1 | 复位 |
| 1 | 1 | 0 | PWM 翻转/置位 |
| 1 | 1 | 1 | PWM 复位/置位 |

由表 2-14 可以看出，适当设置 OUTMOD2、OUTMOD1 和 OUTMOD0 位可以获得相应的输出。

CCIE：捕获/比较的中断使能位。当该位为 1 时，则允许中断；当该位为 0 时，则不允许中断。

CCI：捕获/比较的输入信号。选择的输入信号通过该位来读出。

OUT：输出。该位指示输出的状态。对输出模式 0，该位直接控制输出的状态，这时 OUT 为 1 表示输出高电平，OUT 为 0 表示输出低电平。

COV：捕获溢出。该位指示捕获溢出发生。COV 必须软件复位。当该位为 1 时表示有捕获溢出发生；当该位为 0 时表示没有捕获溢出发生。

CCIFG：捕获/比较的中断标志。当该位为 1 时，表示有中断产生；当该位为 0 时，表示没有中断产生。

以上对 Timer_A 的 CCTLn 寄存器的各个位进行了介绍，通过介绍知道 Timer_A 作为捕获/比较模块使用时有捕获模式和比较模式，下面对定时器的这两种工作模式进行简单的介绍。

捕获模式是通过设置 CCTLn 寄存器中的 CAP 为 1 来选定。这时如果在选定的管脚上发生选定的脉冲触发沿（上升沿、下降沿或者任意沿），则 TAR 寄存器中的值将写入到 CCRn 寄存器中。因此常将此模式用于确定事件发生的时间，如速度的计算和时间的测量等。当捕获完成后，将设置 CCIFG 中断标志。如果总的中断位允许，相应的 CCIE 也允许，则将产生中断请求。

比较模式是通过设置 CCTLn 寄存器中的 CAP 为 0 来选定。这时与捕获有关的硬件停止工作，在记数器 TAR 中记数值等于比较器中的值时设置标志位，产生中断请求；也可结合输出单元产生所需的信号。

前面介绍 Timer_A 的 CCTLn 寄存器知道：3 个输出模式位控制模块的输出。捕获/比较模块都有 1 个输出单元，用于产生输出信号。输出的模式主要由：OUTMOD2、OUTMOD1 和 OUTMOD0 确定，共有 8 种输出模式，所有的输出模式定义如下：

- 输出模式 0：输出模式。输出信号 OUT 由每个捕获/比较模块的控制寄存器 CCTLn 中的 OUT 位定义，并在写入该寄存器后立即更新，最终为 OUT 直通。
- 输出模式 1：置位模式。输出信号在 TAR 等于 CCRn 时置位，并保持置位到定时器复位或者选择另一种输出模式为止。
- 输出模式 2：PWM 翻转/复位模式。输出信号在 TAR 的值等于 CCRn 时翻转，当 TAR 的值等于 CCR0 时复位。
- 输出模式 3：PWM 置位/复位模式。输出信号在 TAR 的值等于 CCRn 时置位，当 TAR 的值等于 CCR0 时复位。
- 输出模式 4：翻转模式。输出信号在 TAR 的值等于 CCRn 时翻转，输出周期是定时器周期的两倍。
- 输出模式 5：复位模式。输出信号在 TAR 的值等于 CCRn 时复位，并保持复位到选择另一种输出模式为止。
- 输出模式 6：PWM 翻转/置位模式。输出信号在 TAR 的值等于 CCRn 时翻转，当 TAR 的值等于 CCR0 时置位。
- 输出模式 7：PWM 复位/置位模式。输出信号在 TAR 的值等于 CCRn 时复位，当 TAR 的值等于 CCR0 时置位。

通过以上对 Timer_A 的 CCTLn 寄存器介绍，对捕获/比较模块的工作应该有了基本的认识。

### 5．CCRn 寄存器

捕获/比较寄存器。在捕获/比较模块中，可读可写。在捕获方式下，当满足捕获条件时，硬件自动将计数器 TAR 的数据写入该寄存器。如要测量某窄脉冲的脉冲宽度，可以定义上升沿和下降沿都捕获。在上升沿时，捕获一个定时器数据，这个数据在捕获寄存器中读出；再

等待下降沿到来，在下降沿时又捕获一个定时器数据；那么两次捕获的定时器数据差就是窄脉冲的高电平宽度。

在比较方式时，用户根据需要设置的时间长短并结合定时器的工作方式和定时器的输入时钟信号写入该寄存器相应的数据。比如需要定时 1 秒，定时器工作模式为增模式，输入的时钟信号为 ACLK（32768Hz），则写入该寄存器的值为 32768。

### 2.5.3 Timer_A 使用例子

通过前面对 Timer_A 模块的介绍，应该对 Timer_A 有了基本的认识，下面结合具体的程序来说明 Timer_A 的使用。下面的程序只是使用 Timer_A 作为定时器使用，具体的程序代码如下：

```
//////////////////////////////////////////
//初始定时器模块
void Init_TimerA()
{
    TACTL = TASSEL1 + TACLR;  //选择SMCLK，清除TAR
    CCTL0 = CCIE; //CCR0 中断允许
    CCR0 = 40000; //200Hz
    TACTL |= MC0; //增记数模式

    //初始化端口
    P1DIR = 0;
    P1SEL = 0;
    P1DIR |= BIT0;
    return;
}
//////////////////////////////////////////////////
//定时器中断
interrupt [TIMERA0_VECTOR] void TimerA_ISR(void)
{
    int i;
    //翻转P1.0管脚
    if(P1OUT & BIT0) P1OUT &= ~(BIT0);
    else P1OUT |= BIT0;
    for(i = 100;i > 0;i--) ;
}
```

通过上面的程序可以看出，只要设置适当的寄存器就可以选择 Timer_A 的工作模式，并进行正确的工作。同理，只需要设置 Timer_A 的其他寄存器就可以选择 Timer_A 的其他工作模块，具体的使用这里就不在详细介绍了。

### 2.5.4 Timer_B

定时器 B 是一个 16 位的定时器/计数器。它最多有 7 个（MSP430F14X 有 7 个）捕获/比较寄存器。定时器 B 能支持多个时序控制、多个捕获/比较功能和多个 PWM 输出。定时器 B 有丰富的中断功能，中断可以由计数器溢出产生，也可以由捕获/比较寄存器产生。定时器

B 主要有以下这些特点：

- 16 位的计数器/定时器，共有 4 种模式，有 4 个可以选择的长度。
- 可以选择设置时钟源。
- 3 个或者 7 个捕获/比较寄存器（MSP430F14X 有 7 个）。
- 双缓冲比较锁存。
- 具有中断向量寄存器，能快速译码定时器 B 产生的中断。

Timer_B 和 Timer_A 相比，结构上基本相同，但也存在差异，主要有如下的差异：

- Timer_B 的记数长度为 8、10、12 或者 16，可以软件编程实现。
- Timer_B 的 TBCCRx 寄存器是双缓冲，并且可以分组。
- 所有的 Timer_B 的输出可以设置成高阻状态。
- Timer_B 中不再使用 SCCI 位。

用户对定时器 B 的所有操作都是通过操作该模块的寄存器完成的。定时器 B 的寄存器主要有 TBCTL、TBR、TBIV、TBCCTLn 和 TBCCRn（n 的值可能是 0～6）。下面结合具体寄存器的介绍来讲解定时器 B 的功能。

### 1．TBCTL 寄存器

TBCTL 寄存器是一个 16 位的寄存器。通过设置该寄存器完成对 Timer_B 作为定时器使用的控制，它包含了 Timer_B 作为定时器使用的所有控制位。图 2-48 给出了 TBCTL 寄存器的各个位。

| 未用 | TBCLGRP1~0 | CNTL1~0 | 未用 | TBSSEL1~0 | 高字节 |
|---|---|---|---|---|---|
| ID1~0 | MC1~0 | 未用 | TBCLR | TBIE | TBIFG | 低字节 |

图 2-48　TBCTL 寄存器

由图 2-48 可以看出，该寄存器有 3 个位目前没有用到。为了增加对 TBCTL 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 TBCTL 寄存器的各个位进行详细介绍。

TBCLGRP1、TBCLGRP0：该两位用来决定是单独还是成组装载比较锁存器，而装载信号被相应的捕获/比较控制寄存器中的 CLLDx 位控制如表 2-15 所示。

表 2-15　定时器 B 的装载选择

| TBCLGRP1 | TBCLGRP0 | 定时器 B 的装载选择 |
|---|---|---|
| 0 | 0 | 单独装载，比较锁存器由各自相应的 TBCCTLx 寄存器的 CLLDx 位控制 |
| 0 | 1 | 分为 3 组装载① |
| 1 | 0 | 分为 2 组装载② |
| 1 | 1 | 选择 1 组装载③ |

注　① TBCCTL1+TBCCTL2（TBCCTL1 中的 CLLDx 定义操作模式）；TBCCTL3+TBCCTL4（TBCCTL3 中的 CLLDx 定义操作模式）；TBCCTL5+TBCCTL6（TBCCTL5 中的 CLLDx 定义操作模式）。

② TBCCTL1+TBCCTL2+ TBCCTL3（TBCCTL1 中的 CLLDx 定义操作模式）；TBCCTL4+TBCCTL5+ TBCCTL6（TBCCTL4 中的 CLLDx 定义操作模式）。

③ TBCCTL1 到 TBCCTL6 为一组（TBCCTL1 中的 CLLDx 定义操作模式）。

由表 2-15 可以看出：通过设置 TBCLGRP1 和 TBCLGRP0 的值可以确定定时器 B 的装载选择。

CNTL1、CNTL0：该位用来选择定时器的定时长度。表 2-16 给出了 CNTL1、CNTL0 位与定时长度的关系。

表 2-16　定时器 B 的定时长度

| CNTL1 | CNTL0 | 定时器 B 的定时长度 |
|---|---|---|
| 0 | 0 | 16 位，最大值为 FFFFH |
| 0 | 1 | 12 位，最大值为 FFFH |
| 1 | 0 | 10 位，最大值为 3FFH |
| 1 | 1 | 8 位，最大值为 FFH |

由表 2-16 可以看出：通过设置 CNTL1 和 CNTL0 的值可以完成对定时器 B 的定时长度的选择。

TBSSEL1、TBSSEL0：定时器 B 的时钟源的选择。表 2-17 给出了 TBSSEL1、TBSSEL0 位与时钟源的选择的关系。

表 2-17　定时器 B 的时钟源的选择

| TBSSEL1 | TBSSEL0 | 定时器 B 的时钟源 |
|---|---|---|
| 0 | 0 | TBCLK（使用外部管脚信号作为输入） |
| 0 | 1 | ACLK |
| 1 | 0 | MCLK |
| 1 | 1 | INCLK（外部输入时钟） |

由表 2-17 可以看出：通过设置 TBSSEL1 和 TBSSEL0 的值可以完成对定时器 B 的时钟源的选择。

ID1、ID0：这两个位来选择输入时钟的分频系数。表 2-18 给出了 ID1、ID0 位与输入时钟的分频系数选择的关系。

表 2-18　定时器 B 的分频系数的选择

| ID1 | ID0 | 定时器 B 的分频系数 |
|---|---|---|
| 0 | 0 | 直通，时钟不分频 |
| 0 | 1 | 1/2 分频 |
| 1 | 0 | 1/4 分频 |
| 1 | 1 | 1/8 分频 |

由表 2-18 可以看出：通过设置 ID1 和 ID0 的值可以完成对定时器 B 的输入时钟的分频系数的选择。

MC1、MC0：这两个位来选择 Timer_B 作为定时器的工作模式。表 2-19 给出了 MC1、MC0 位与工作模式选择的关系。

表 2-19　定时器 B 的工作模式的选择

| MC1 | MC0 | 定时器 B 的工作模式 |
|---|---|---|
| 0 | 0 | 停止模式 |
| 0 | 1 | 增模式 |
| 1 | 0 | 连续模式 |
| 1 | 1 | 增减模式 |

由表 2-19 可以看出：通过设置 MC1 和 MC0 的值可以完成对定时器 B 的作为定时器使用时工作模式的选择。

TBCLR：定时器 B 的清除控制。设置该位为 1 时，TBR 寄存器里的内容、计数器方向等内容被清除。

TBIE：Timer_B 的中断使能。当该位为 1 时，允许 Timer_B 的中断，当该位为 0 时，不允许 Timer_B 的中断。

TBIFG：Timer_B 的中断标志。当该位为 1 时，有中断产生，当该位为 0 时，没有中断产生。

以上对 Timer_B 的 TBCTL 寄存器的各个位进行了介绍，通过介绍知道 Timer_B 作为定时器使用时有 4 种工作模式，下面对定时器的工作模式进行简单的介绍。

停止模式：定时器处于暂停状态。当 Timer_B 处于暂停状态时，TBR 寄存器的内容不受影响。当 Timer_B 重新启动时，计数器从原来的值进行记数，并且记数的方向与原来的方向一致。

增记数模式：计数器工作在增记数模式。该模式用于定时周期小于定时器 B 的最大记数值（最大记数值由选择的定时长度确定）的连续记数方式，TBCL0 寄存器里面的内容为定时器的记数周期数。计数器的值增加，当计数器的值增加到等于 TBCL0 寄存器里面的内容时，计数器就重新从 0 开始记数。在模式选择为增记数的模式时，如果定时器的值大于或者等于 TBCL0 寄存器里面的值时，计数器立即从 0 开始记数。当计数器的记数记到 TBCL0 时，会设置 CCIFG 中断标志，当定时器从 TBCL0 的值转到 0 时，设置 TBIFG 中断标志。

连续记数模式：计数器处于从 0 到 TBR 的最大值的连续记数模式。该模式的定时周期为 TBR 的最大值（最大值由选择的定时长度确定）个时钟周期。在该模式下 TBCL0 作为一般的捕获/比较寄存器使用。在连续记数模式下，计数器从 0 记数到 TBR 的最大值，然后又回到 0 重新记数。当计数器记数从 TBR 的最大值到 0 时，设置 TBIFG 中断标志。连续模式可以用来产生独立的时间间隔并输出频率。每个时间间隔完成就会产生一个中断。在中断到来时，必须在当前中断服务程序里将下一个时间间隔加到 TBCLx 寄存器里，这样顺序执行，就可以产生周期性的中断，从而向外输出频率，Timer_B 最多可以使用 7 个捕获/比较寄存器输出 7 个不同的频率。

增/减记数模式：计数器 TBR 的值先增后减。当计数器增记数到 TBCL0 的值时，计数器变为减记数，当计数器减记数到 0 时，设置 TBIFG 中断标志位。在增/减记数模式下，记数的周期是 TBCL0 寄存器值的两倍。增/减记数模式的计数方向是锁存的，这样可以允许计数器暂停和重新启动，在重新启动后，记数方向不发生变化，如果不是这样的，则可以设置 TBCTL 寄存器里面的 TBCLR 位来清除 TBR 寄存器的内容。在增/减记数模式下，CCIFG 和

TBIFG 中断标志在一个周期内只设置一次。当计数器从 TBCL0-1 增记数到 TBCL0 时，设置 CCIFG 中断标志；当计数器从 1 减记数到 0 时，设置 TBIFG 中断标志。

**2．TBR 寄存器**

该寄存器是执行记数的单元，是计数器的主体。该寄存器是一个 16 位的寄存器，但当计数器长度设置为 14 位、12 位或者 8 位时，它只能记数到相应的记数长度。该寄存器的内容可读可写，但必须按字的方式进行读写，写入该寄存器的时候应该停止计数器的记数，这样保证写入的数据可靠。

**3．TBIV 寄存器**

该寄存器为定时器 B 模块的中断向量寄存器。CCR1 的 CCIFG 中断标志、CCR2 的 CCIFG 中断标志等和 TBIFG 中断标志使用一个中断向量，TBIV 寄存器就是用来判断是哪一个中断标志请求。TBIV 是一个 16 位的寄存器，该寄存器的位分配如图 2-49 所示。

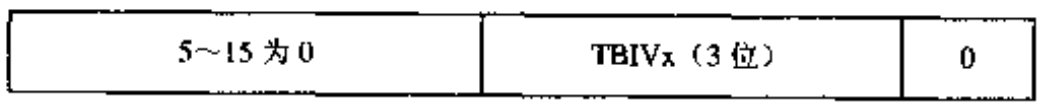



图 2-49　TBIV 寄存器

由图 2-49 可以看出，TBIV 寄存器使用了 3 位对中断向量进行编码，以便区别是哪一个中断请求，具体的编码见表 2-20。

表 2-20　TBIV 中断向量值

| TBIV 内容 | 中断源 | 中断标志 | 中断优先级 |
|---|---|---|---|
| 00H | 没有中断标志 | — | |
| 02H | 捕获/比较 1 | CCR1 的 CCIFG | 最高 |
| 04H | 捕获/比较 2 | CCR2 的 CCIFG | |
| 06H | 保留将来使用 | CCR3 的 CCIFG | |
| 08H | 保留将来使用 | CCR4 的 CCIFG | |
| 0AH | 定时器溢出 | CCR5 的 CCIFG | |
| 0CH | 保留将来使用 | CCR6 的 CCIFG | |
| 0EH | 保留将来使用 | TBIFG | 最低 |

由表 2-20 可以看出：只要确定了 TBIV 寄存器的值，就可以很容易判断出是那个中断标志产生，从而进行相应的处理。

**4．TBCCTLn 寄存器**

捕获/比较控制寄存器。该寄存器用来控制 Timer_B 作为捕获/比较模块时的控制功能，该寄存器的位分配如图 2-50 所示。

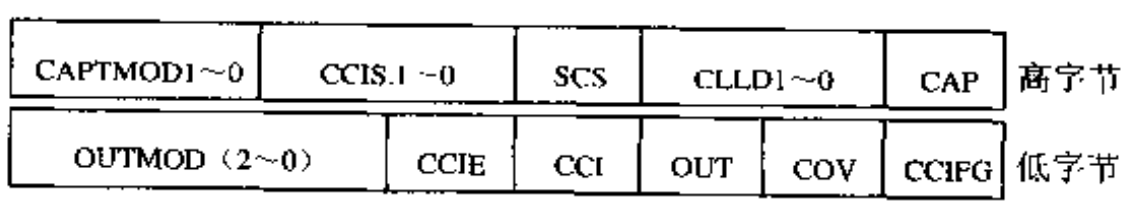



图 2-50　TBCCTLn 寄存器

为了增加对 TBCCTLn 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 TBCCTLn 寄存器的各个位进行详细介绍。

CAPTMOD1、CAPTMOD0：该两位选择捕获模式。表 2-21 给出了 CAPTMOD1、CAPTMOD0 位与捕获模式选择的关系。

表 2-21　捕获模式的选择

| CAPTMOD1 | CAPTMOD0 | 捕获模式的选择 |
|---|---|---|
| 0 | 0 | 禁止捕获模式 |
| 0 | 1 | 上升沿捕获模式 |
| 1 | 0 | 下降沿捕获模式 |
| 1 | 1 | 上升沿和下降沿捕获模式 |

由表 2-21 可以看出，适当设置 CAPTMOD1 和 CAPTMOD0 位就可以选择相应的捕获模式。

CCIS1、CCIS0：捕获/比较的输入选择。这些位用来选择 CCRx 的输入信号。这些位只在捕获模式下使用，在比较模式下不使用。表 2-22 给出了 CCIS1、CCIS0 位对输入信号的选择。

表 2-22　捕获模式的输入信号选择

| CCIS1 | CCIS0 | 输入信号的选择 |
|---|---|---|
| 0 | 0 | 选择 CCIxA 作为捕获事件的输入信号 |
| 0 | 1 | 选择 CCIxB 作为捕获事件的输入信号 |
| 1 | 0 | 选择 GND 作为捕获事件的输入信号 |
| 1 | 1 | 选择 VCC 作为捕获事件的输入信号 |

由表 2-22 可以看出，适当设置 CCIS1 和 CCIS0 位就可以对捕获模式的输入信号进行相应的选择。

SCS：同步捕获源。该位用来使捕获输入信号与定时器时钟信号同步。当该位为 1 时，为同步捕获；当该位为 0 时，为异步捕获。

CLLD1、CLLD0：该两个位确定了比较锁存器的状态方式。表 2-23 给出了 CLLD1、CLLD0 位与捕获模式选择的关系。

表 2-23　锁存器装载方式的选择

| CLLD1 | CLLD0 | 锁存器装载方式的选择 |
|---|---|---|
| 0 | 0 | 立即装载 |
| 0 | 1 | 当定时器 TBR 记数到 0 时，TBCCRx 的数据装载到 TBCLx |
| 1 | 0 | 在增/减记数模式下，当 TBR 记数到 TBCL0 或者 0 时，TBCCRx 的数据装载到 TBCLx；在连续模式下，当 TBR 记数到 0 时，TBCCRx 的数据装载到 TBCLx |
| 1 | 1 | 上升沿和下降沿捕获模式 |

由表 2-23 可以看出，适当设置 CLLD1 和 CLLD0 位就可以选择相应的锁存器装载方式。

CAP：捕获模式选择。当该位为 1 时，该模块工作在捕获模式；该位为 0 时，该模块工

作在比较模式。

OUTMOD0、OUTMOD1、OUTMOD2：输出模式。表 2-24 给出了 OUTMOD0、OUTMOD1、OUTMOD2 位与输出模式的关系。

表 2-24　输出模式的选择

| OUTMOD2 | OUTMOD1 | OUTMOD0 | 输出模式 |
|---|---|---|---|
| 0 | 0 | 0 | 输出 |
| 0 | 0 | 1 | 置位 |
| 0 | 1 | 0 | PWM 翻转/复位 |
| 0 | 1 | 1 | PWM 置位/复位 |
| 1 | 0 | 0 | 翻转 |
| 1 | 0 | 1 | 复位 |
| 1 | 1 | 0 | PWM 翻转/置位 |
| 1 | 1 | 1 | PWM 复位/置位 |

由表 2-24 可以看出，适当设置 OUTMOD2、OUTMOD1 和 OUTMOD0 位可以获得相应的输出。

CCIE：捕获/比较的中断使能位。当该位为 1 时，则允许中断；当该位为 0 时，则不允许中断。

CCI：捕获/比较的输入信号。选择的输入信号通过该位来读出。

OUT：输出。该位指示输出的状态。对输出模式 0，该位直接控制输出的状态，这时 OUT 为 1 表示输出高电平，OUT 为 0 表示输出低电平。

COV：捕获溢出。该位指示捕获溢出发生。COV 必须软件复位。当该位为 1 时表示有捕获溢出发生；当该位为 0 时表示没有捕获溢出发生。

CCIFG：捕获/比较的中断标志。当该位为 1 时，表示有中断产生；当该位为 0 时，表示没有中断产生。

以上对 Timer_B 的 TBCCTLn 寄存器的各个位进行了介绍，通过介绍知道 Timer_B 作为捕获/比较模块使用时有捕获模式和比较模式，下面对定时器的这两种工作模式进行简单的介绍。

捕获模式是通过设置 TBCCTLn 寄存器中的 CAP 为 1 来选定。这时如果在选定的管脚上发生选定的脉冲触发沿（上升沿、下降沿或者任意沿），则 TBR 寄存器中的值将写入到 TBCCRn 寄存器中。因此常将此模式用于确定事件发生的时间，如速度的计算和时间的测量等。当捕获完成后，将设置 CCIFG 中断标志。如果总的中断位允许，相应的 CCIE 也允许，则将产生中断请求。

比较模式是通过设置 TBCCTLn 寄存器中的 CAP 为 0 来选定。这时与捕获有关的硬件停止工作，在记数器 TBR 中记数值等于比较器中的值时设置标志位，产生中断请求；也可结合输出单元产生所需的信号。

前面介绍 Timer_B 的 TBCCTLn 寄存器知道：3 个输出模式位控制模块的输出。捕获/比较模块都有 1 个输出单元，用于产生输出信号。输出的模式主要由：OUTMOD2、OUTMOD1

和 OUTMOD0 确定，共有 8 种输出模式，所有的输出模式定义如下：

- 输出模式 0：输出模式。输出信号 OUT 由每个捕获/比较模块的控制寄存器 TBCCTLn 中的 OUT 位定义，并在写入该寄存器后立即更新，最终为 OUT 直通。
- 输出模式 1：置位模式。输出信号在 TBR 等于 TBCCRn 时置位，并保持置位到定时器复位或者选择另一种输出模式为止。
- 输出模式 2：PWM 翻转/复位模式。输出信号在 TBR 的值等于 TBCCRn 时翻转，当 TBR 的值等于 TBCL0 时复位。
- 输出模式 3：PWM 置位/复位模式。输出信号在 TBR 的值等于 TBCCRn 时置位，当 TAR 的值等于 TBCL0 时复位。
- 输出模式 4：翻转模式。输出信号在 TBR 的值等于 TBCCRn 时翻转，输出周期是定时器周期的两倍。
- 输出模式 5：复位模式。输出信号在 TBR 的值等于 TBCCRn 时复位，并保持复位到选择另一种输出模式为止。
- 输出模式 6：PWM 翻转/置位模式。输出信号在 TBR 的值等于 TBCCRn 时翻转，当 TBR 的值等于 TBCL0 时置位。
- 输出模式 7：PWM 复位/置位模式。输出信号在 TBR 的值等于 TBCCRn 时复位，当 TBR 的值等于 TBCL0 时置位。

通过以上对 Timer_A 的 CCTLx 寄存器介绍，对捕获/比较模块的工作应该有了基本的认识。

**5．TBCCRn 寄存器**

捕获/比较寄存器。在捕获/比较模块中，可读可写。在捕获方式下，当满足捕获条件时，硬件自动将计数器 TBR 的数据写入该寄存器。如要测量某窄脉冲的脉冲宽度，可以定义上升沿和下降沿都捕获。在上升沿时，捕获一个定时器数据，这个数据在捕获寄存器中读出；再等待下降沿到来，在下降沿时又捕获一个定时器数据；那么两次捕获的定时器数据差就是窄脉冲的高电平宽度。

在比较方式时，用户根据需要设置的时间长短并结合定时器的工作方式和定时器的输入时钟信号写入该寄存器相应的数据。比如需要定时 1 秒，定时器工作模式为增模式，输入的时钟信号为 ACLK（32768Hz），则写入该寄存器的值为 32768。

## 2.5.5　Timer_B 使用例子

通过前面对 Timer_B 模块的介绍，应该对 Timer_B 有了基本的认识，下面结合具体的程序来说明 Timer_B 的使用。下面的程序只是使用 Timer_B 作为定时器使用，具体的程序代码如下：

```
//////////////////////////////////////
//初始定时器模块
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;                  //增记数模式

    //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--) ;
}
```

通过上面的程序可以看出，只要设置适当的寄存器就可以选择 Timer_B 的工作模式，并进行正确的工作。同理，只需要设置 Timer_B 的其他寄存器就可以选择 Timer_B 的其他工作模块，具体的使用这里就不在详细介绍了。

## 2.6 比较器

MSP430F1XX 系列单片机中的大部分型号都带有比较器模块（比较器 A）。比较器 A 支持 A/D 转换、电压监控和外部模拟信号的监控。比较器 A 主要有以下特点：

- 反向和非反向的终端输入复用器。
- 比较器输出有软件选择的 RC 滤波器。
- 比较器的输出可以作为定时器 A 的捕获输入。
- 端口输入缓冲由软件控制。
- 具有中断功能。
- 可选择参考电压的产生。
- 比较器和参考电压产生可以关闭。

用户对比较器 A 的所有操作都是通过操作该模块的寄存器完成的。比较器 A 的寄存器主要有 CACTL1、CACTL2 和 CAPD。下面结合具体寄存器的介绍来讨论比较器 A 的功能。

### 1．CACTL1 寄存器

CACTL1 是一个 8 位的寄存器，该寄存器包含比较器的大部分控制位。该寄存器的位分配如图 2-51 所示。

| CAEX | CARSEL | CAREF1~0 | CAON | CAIES | CAIE | CAIFG |
|---|---|---|---|---|---|---|

图 2-51　CACTL1 寄存器

为了增加对 CACTL1 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 CACTL1 寄存器的各个位进行详细介绍。

CAEX：交换比较器的输入端，也控制比较器的输出是否反向，多用于测量与补偿比较器 A 的偏压。

CARSEL：比较器的参考选择。该位与 CAEX 位结合起来选择内部参考电源。表 2-25 给出了 CAEX 和 CARSEL 两个位与选择内部参考电源的关系。

表 2-25　内部参考电源的选择

| CAEX | CARSEL | 内部参考电源的选择 |
|---|---|---|
| 0 | 0 | 内部参考电源接正端 |
| 0 | 1 | 内部参考电源接负端 |
| 1 | 0 | 内部参考电源接负端 |
| 1 | 1 | 内部参考电源接正端 |

由表 2-25 可以看出，适当设置 CAEX 和 CARSEL 位就可以完成内部参考电源的选择。

CAREF1、CAREF0：该两位用来选择参考电源。表 2-26 给出了 CAREF1 和 CAREF0 两个位与参考电源的关系。

表 2-26　参考电源的选择

| CAREF1 | CAREF0 | 参考电源的选择 |
|---|---|---|
| 0 | 0 | 内部参考电源关闭，使用外部参考电源 |
| 0 | 1 | 选择 0.25Vcc 作为参考电压 |
| 1 | 0 | 选择 0.5Vcc 作为参考电压 |
| 1 | 1 | 选择二极管电压作为参考电压 |

由表 2-26 可以看出，适当设置 CAREF1 和 CAREF0 位就可以完成参考电源的选择。

CAON：比较器的开关控制。当位为 1 时，打开比较器；当该位为 0 时，关闭比较器。比较器关闭可以用来降低功耗。

CAIES：比较器中断的触发沿选择。当该位为 1 时，下降沿触发；当该位为 0 时，上升沿触发。

CAIE：该位用于控制中断的使能。当该位为 1 时，允许中断；当该位为 0 时，中断不允许。

CAIFG：该位为中断标志位。当该位为 1 时，有中断产生；当该位为 0 时，没有中断产生。

通过以上位的介绍可以知道比较器具有中断功能。设置 CAIE 位使能中断，可以设置 CAIES 选择中断出发沿，当有中断产生的时候，硬件会设置 CAIFG 中断标志位，然后进入中断服务程序进行处理，硬件会自动清除中断标志。

### 2. CACTL2寄存器

CACTL2是一个8位的寄存器，该寄存器包含比较器的控制位，主要是一些与输入输出有关的信息。该寄存器的位分配如图2-52所示。

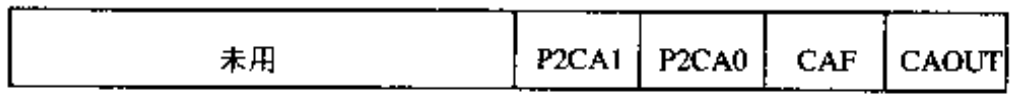



图2-52 CACTL2寄存器

由图2-52可以看出，CACTL2的高4位未用，只使用了低4位。为了增加对CACTL2寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对CACTL2寄存器的各个位进行详细介绍。

P2CA1：该位用来选择CA1管脚。当该位为1时，管脚连接到CA1；当该位为0时，管脚不连接到CA1。

P2CA0：该位用来选择CA0管脚。当该位为1时，管脚连接到CA0；当该位为0时，管脚不连接到CA0。

CAF：比较器的输出滤波选择。当该位为1时，比较器的输出进行滤波处理；当该位为0时，比较器的输出不进行滤波处理。

CAOUT：比较器的输出。该位是只读的，对该位写操作无效。

### 3. CAPD寄存器

该寄存器用来控制比较器输入的缓冲功能。MSP430F1XX系列单片机的I/O口都有缓冲功能，当比较器使用该管脚的时候，可以关闭输入缓冲功能，这一功能由CAPD寄存器控制。当该寄存器的某个位为1时，则相应的管脚关闭输入缓冲功能；当该寄存器的某个位为0时，则相应的管脚不关闭输入缓冲功能。

通过以上对比较寄存器的介绍，对比较器有了基本的了解。下面结合具体的程序来说明比较器的使用，具体的程序代码如下：

```
void main(void)
{
    int nCount_Comp;
    int i;
    int nAlarm_flag;
    nCount_Comp = 0;
    nAlarm_flag = 0;
    Init_Comp();
    for(;;)
    {
        nCount_Comp += 1;
        if(nCount_Comp >= 5000)
        {
            nCount_Comp = 0;
            CACTL1 |= CAON;       //打开比较器
            for(i = 20;i > 0;i--) ;
            if(CAOUT & CACTL2) nAlarm_flag = 0;
```

```
            else nAlarm_flag = 1;
            CACTL1 &= ~(CAON);   //关闭比较器

        }
        for(i = 65536;i > 0;i--) ;
    }
}
//初始化比较器
void Init_Comp(void)
{
    //设置寄存器1
    CACTL1 = 0X00;
    //设置寄存器2
    CACTL2 = 0x00;
    CACTL2 |= P2CA1;    //外部信号连接到比较器A
    CACTL2 |= P2CA0;    //外部信号连接到比较器A
    CACTL2 |= CAF;      //比较器输出经过RC低通滤波器
}
```

通过上面的程序可以看出，只要设置 CACTL2 和 CACTL1 寄存器就可以使比较器进行正确的工作。

## 2.7　MSP430F1XX 的 FLASH 模块

MSP430F1XX 系列单片机的 FLASH 模块可以按位、字节和字访问，并且可以进行编程和擦除。FLASH 模块有一个集成控制器用来控制编程和擦除操作。该集成控制器有 3 个寄存器，用来产生编程和擦除的时序，也用来提供编程和擦除的电压。FLASH 模块的确省模式是读操作。在读模式下，不能进行编程操作和擦除操作。FLASH 模块也提供编程和擦除操作，从而可以实现对在 FLASH 里进行数据保存。用户对 FLASH 模块的所有操作都是通过操作该模块的寄存器完成的。FLASH 模块的寄存器主要有 FCTL1、FCTL2 和 FCTL3。下面结合具体寄存器的介绍来介绍 FLASH 模块的操作。

### 1．FCTL1 寄存器

FCTL1 寄存器是一个 16 位的寄存器。该寄存器主要定义了 FLASH 模块的擦除操作和编程操作的控制位。该寄存器的高 8 位为安全键值，该字段读出内容总是 96H，写入时必须写入 5AH，否则不能进行操作，该字段主要提供保护功能。图 2-53 给出了 FCTL1 寄存器的各个位。

| 安全键值（15~8） | BKWRT | WRT | 未用 | 未用 | 未用 | MERAS | ERASE | 未用 |
|---|---|---|---|---|---|---|---|---|

图 2-53　FCTL1 寄存器

由图 2-53 可以看出，该寄存器只使用了 4 个有效的位。为了增加对 FCTL1 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 FCTL1 寄存器的各个位进行详细介绍。

BKWRT：按块写模式位。在进行块写操作时，WRT 位也必须设置，当设置 EMEX 时，BKWRT 自动复位。当该位为 1 时，打开块写模式；该位为 0 时，关闭块写模式。

WRT：写模式位。该位用来选择任一个写模式。当设置 EMEX 时，WRT 自动复位。当该位为 1 时，打开写模式；该位为 0 时，关闭写模式。

MERAS、ERASE：该两个位用来控制擦除操作。表 2-27 给出了 MERAS、ERASE 两个位与擦除操作的关系。

表 2-27　MERAS、ERASE 两个位与擦除操作的关系

| MERAS | ERASE | 擦除操作 |
|---|---|---|
| 0 | 0 | 不擦除 |
| 0 | 1 | 只擦除单个段 |
| 1 | 0 | 擦除所有的主程序段 |
| 1 | 1 | 擦除所有的主程序段和信息段 |

由表 2-27 可以看出，适当设置 MERAS、ERASE 两个位就可以选择适当的擦除操作。

### 2．FCTL2 寄存器

FCTL2 寄存器是一个 16 位的寄存器。该寄存器主要定义了 FLASH 模块的擦除操作和编程操作所序时序的时钟定义。该寄存器的高 8 位为安全键值。图 2-54 给出了 FCTL2 寄存器的各个位。

| 安全键值（15~8） | SSEL1 | SSEL0 | FN5 | FN4 | FN3 | FN2 | FN1 | FN0 |
|---|---|---|---|---|---|---|---|---|

图 2-54　FCTL2 寄存器

由图 2-54 可以看出，该寄存器主要定义了 8 个有效的位。为了增加对 FCTL2 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 FCTL2 寄存器的各个位进行详细介绍。

SSEL1、SSEL0：该两个位定义了 FLASH 模块控制器的时钟源的选择。表 2-28 给出了 SSEL1、SSEL0 两个位与时钟源的选择关系。

表 2-28　FLASH 模块控制器时钟源的选择

| SSEL1 | SSEL0 | FLASH 模块控制器的时钟源的选择 |
|---|---|---|
| 0 | 0 | ACLK |
| 0 | 1 | MCLK |
| 1 | 0 | SMCLK |
| 1 | 1 | SMCLK |

由表 2-28 可以看出，适当设置 SSEL1、SSEL0 两个位就可以为 FLASH 模块控制器选择适当的时钟源。

FN5～FN0：这 6 个位定义了分频系数。分频系数为 FN5～FN0 的值加上 1，比如当 FN5～FN0 的值为 1 时，则分频系数是 2。

### 3．FCTL3 寄存器

FCTL3 寄存器是一个 16 位的寄存器。该寄存器主要定义一些标志位。该寄存器的高 8 位为安全键值。图 2-55 给出了 FCTL3 寄存器的各个位。

| 安全键值（15~8） | 未用 | 未用 | EMEX | LOCK | WAIT | ACCIFG | KEYV | BUSY |
|---|---|---|---|---|---|---|---|---|

图 2-55　FCTL3 寄存器

由图 2-55 可以看出，该寄存器使用了 6 个有效的位。为了增加对 FCTL3 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 FCTL3 寄存器的各个位进行详细介绍。

EMEX：紧急退出位。当该位为 1 时，则立即停止对 FLASH 的操作。

LOCK：锁。当该位设置为 1 时，表示加锁，这时不能对 FLASH 进行写和擦除操作；当该位设置为 0 时，表示不加锁，这时可以对 FLASH 进行写和擦除操作。

WAIT：等待指示位。该位显示 FLASH 正在进行写。当该位为 1 时，表示 FLASH 准备好下一次写操作；当该位为 0 时，表示 FLASH 还没有准备好下一次写操作。

ACCIFG：非法访问中断标志。当该位为 1 时，表示有非法访问标志；当该位为 0 时，表示没有有非法访问标志。

KEYV：安全键值除出错标志。当该位为 1 时，表示安全键值不正确；当该位为 0 时，表示安全键值正确。

BUSY：忙标志。该位显示 FLASH 模块时序产生的状态。当该位为 1 时，表示忙；当该位为 0 时，表示不忙。

通过以上对 FLASH 模块控制器的寄存器的介绍，对 FLASH 模块有了基本的了解。下面结合具体的程序来说明 FLASH 模块的写和擦除操作，具体的程序代码如下：

```
void FLASH_ww(int *pData,int nValue)
{
    FCTL3 = 0xA500;  //LOCK = 0;
    FCTL1 = 0xA540;  //WRT = 1;

    *pData = nValue;
}
void FLASH_wb(char *pData,char nValue)
{
    FCTL3 = 0xA500;  //LOCK = 0;
    FCTL1 = 0xA540;  //WRT = 1;

    *pData = nValue;
}
void FLASH_clr(int *pData)
{
    FCTL1 = 0xA502;  //ERASE = 1;
    FCTL3 = 0xA500;  //LOCK = 0;
```

```
    *pData = 0;
}
```

以上主要是对 FLASH 进行按字写、按字节写和擦除操作，通过上面的程序可以看出，只要设置 FCTL1、FCTL2 和 FCTL3 寄存器就可以对 FALSH 模块进行写、擦除等操作。

# 2.8 MSP430F1XX 的 USART

在单片机系统中，串口通信是一个非常重要的部分，通过串口通信实现与其他模块进行通信。MSP430F1XX 系列单片机除了 MSP430F11X 系列单片机没有串口通信外，其他型号的单片机都有串口通信模块。MSP430F1XX 系列单片机里提供的串口通信模块为 USART。该模块即可以作为 UART 使用，提供异步通信功能，也可以作为 SPI 使用，提供同步通信功能。下面具体从 USART 的结构、UASRT 的寄存器和工作模式进行介绍，并给出具体的应用例子。

## 2.8.1 USART 的结构

USART 硬件模块主要包括波特率部分、接收部分、发送部分和接口部分等。USART 的硬件结构图如图 2-56 所示。

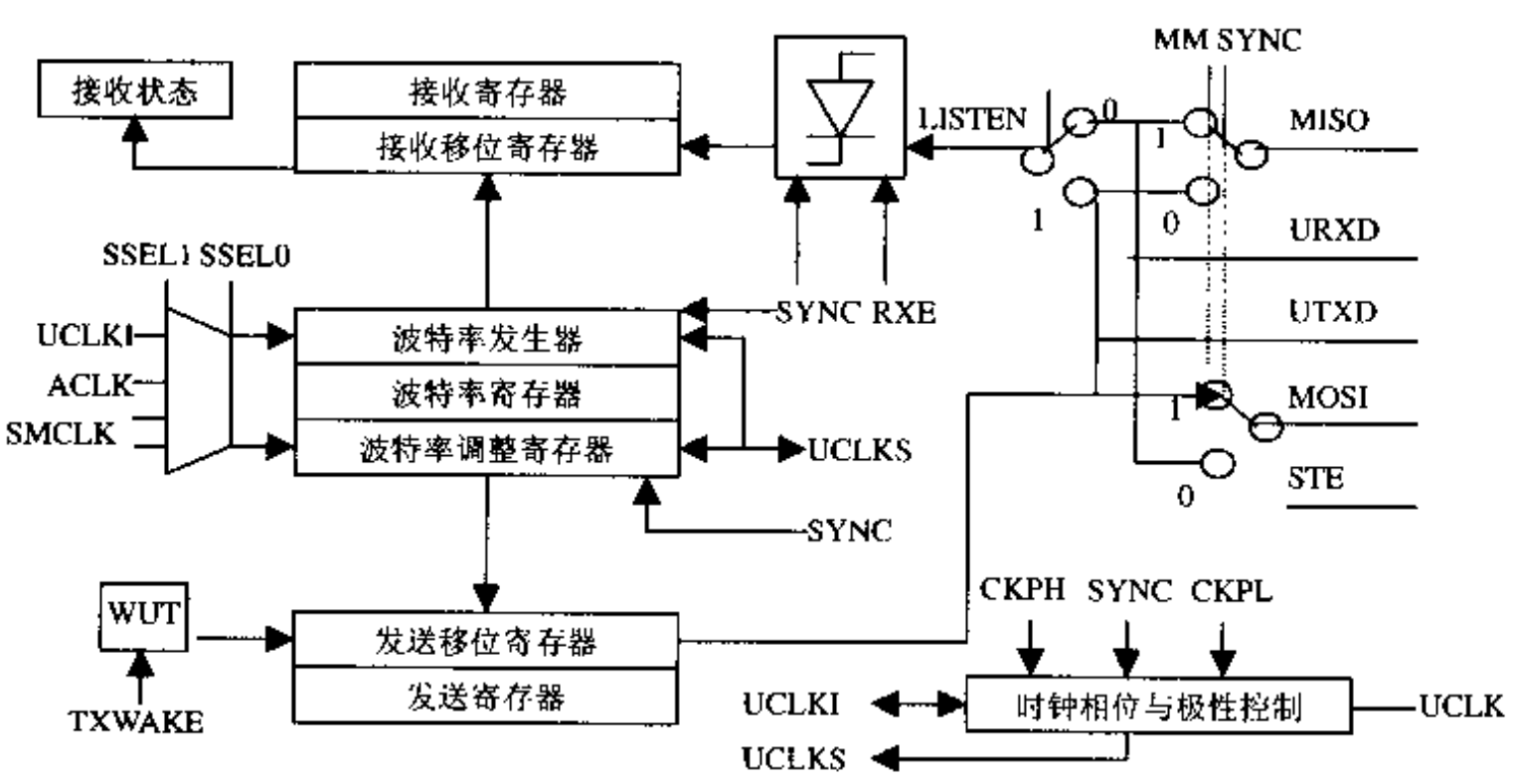



图 2-56 USART 硬件结构图

由图 2-56 可以看出，USART 的接收部分主要包括接收寄存器、接收移位寄存器以及控制模块组成，它在接收的时候产生一些状态信息，并设置相应的中断标志位。USART 的发送部分主要包括发送寄存器、发送移位寄存器以及控制模块组成，它在发送的时候产生一些状态信息，并可以设置发送中断标志位。USART 的波特率产生部分主要包括时钟的选择、波特率的产生以及波特率的调整部分组成，它通过设置波特率寄存器和波特率调整寄存器来获得需要的波特率。另外 USART 模块还包括一个控制模块，通过控制模块可以选择相应的工作模式，同时设置相应的管脚。用户对 USART 模块的所有操作都是通过操作该模块的寄

存器完成的，下面将介绍 USART 模块的寄存器。

## 2.8.2　USART 的寄存器和工作模式

USART 模块主要是对外进行通信，它可以实现异步通信（UART），也可以实现同步通信（SPI）。用户对 USART 模块的所有操作都是通过操作该模块的寄存器实现的。USART 模块的寄存器主要有 UxCTL、UxTCTL、UxRCTL、UxMCTL、UxBR0、UxBR1、UxRXBUF 和 UxTXBUF 等寄存器，其中 x 的值可能是 0 或者 1，因为 MSP430F1XX 系列单片机可能有 1 个 USART 模块，也可能有 2 个 USART 模块，这样 x 的值可能是 0 也可能是 1。下面结合具体寄存器来介绍 USART 模块的功能。

### 1．UxCTL 寄存器

UxCTL 寄存器是一个 8 位的寄存器。UASRT 模块的基本操作由该寄存器的控制位确定的，它包含了通信协议、通信模式和校验位等的选择。图 2-57 给出了 UxCTL 寄存器的各个位。

| PENA | PEV | SPB | CHAR | LISTEN | SYNC | MM | SWRST |
|---|---|---|---|---|---|---|---|

图 2-57　UxCTL 寄存器

由图 2-57 可以看出，UxCTL 寄存器主要包括 8 个有效的控制位。为了增加对 UxCTL 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 UxCTL 寄存器的各个位进行详细介绍。

PENA：校验使能位。当该位为 0 时，不允许校验；当该位为 1 时，允许校验。如果允许校验，则发送时产生校验位，在接收时希望接收到校验位。当在地址位多机模式中，地址位包括在校验计算中。

PEV：奇偶校验位。当该位为 0 时，进行奇校验；当该位为 1 时，进行偶校验。

SPB：停止位。该位用来选择发送时停止位的个数，但接收时停止位只有一个。当该位为 0 时，发送时只有 1 个停止位；当该位为 1 时，发送时有 2 个停止位。

CHAR：字符长度位。该位用来选择发送时数据的长度。当该位为 0 时，发送的数据为 7 位；当该位为 1 时，发送的数据为 8 位。

LISTEN：监听使能位。该位用来选择反馈模式。当该位为 0 时，没有反馈；当该位为 1 时，有反馈，发送的数据被送到接收器，这样可以进行自环测试。

SYNC：该位用于同步模式选择和异步模式选择。当该位为 0 时，USART 模块为异步通信（UART）模式；当该位为 1 时，USART 模块为同步通信（SPI）模式。

MM：多机模式选择位。当该位为 0 时，多机模式选择线路空闲多机协议；当该位为 1 时，多机模式选择地址位多机协议。

SWRST：软件复位使能位。当该位为 0 时，UASRT 模块被允许；当该位为 1 时，UASRT 模块被禁止。

通过以上对 UxCTL 寄存器的各个位的介绍，可以完成对通信模式和通信数据格式等的选择。

### 2. UxTCTL寄存器

UxTCTL寄存器是一个8位的寄存器。该寄存器控制USART模块的发送操作。图2-58给出了UxTCTL寄存器的各个位。

| CKPH | CKPL | SSEL1 | SSEL0 | URXSE | TXWAKE | STC | TXEPT |
|---|---|---|---|---|---|---|---|

图2-58 UxTCTL寄存器

由图2-58可以看出，UxTCTL寄存器主要包括8个有效的控制位。为了增加对UxTCTL寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对UxTCTL寄存器的各个位进行详细介绍。

CKPH：该位用于在SPI模式下控制UCLK时钟的相位。当该位为0时，使用正常的UCLK时钟；当该位为1时，UCLK时钟信号被延迟半个周期。

CKPL：时钟极性选择。在UART模式下，该位控制UCLKI信号的极性；在SPI模式下，该位控制UCLK信号的极性。当该位为0时，在UART模式下UCLKI的极性和UCLK的极性一致；在SPI模式下，数据在UCLK的上升沿输出，输入数据在UCLK的上升沿被锁存。当该位为1时，在UART模式下UCLKI的极性和UCLK的极性相反；在SPI模式下，数据在UCLK的下降沿输出，输入数据在UCLK的下降沿被锁存。

SSEL1、SSEL0：时钟源选择。这两个位确定波特率发生器的时钟源。表2-29给出了SSEL1、SSEL0位与时钟源选择的关系。

表2-29 时钟源的选择

| SSEL1 | SSEL0 | 时钟源的选择 |
|---|---|---|
| 0 | 0 | 选择外部时钟UCLK |
| 0 | 1 | 选择辅助时钟ACLK |
| 1 | 0 | 选择子系统时钟SMCLK |
| 1 | 1 | 选择子系统时钟SMCLK |

由表2-29可以看出，适当设置SSEL1和SSEL0位就可以选择相应的时钟源。

URXSE：UART的接收触发沿。该位用于控制UART的接收触发沿。当该位为0时，没有接收到数据；当该位为1时，接收到数据，请求接收中断服务。

TXWAKE：发送器唤醒。当该位为0时，下一字节是数据；当该位为1时，下一字节是地址。

STC：STE管脚选择位。当该位为0时，STE管脚使能，选择SPI的4线模式；当该位为0时，STE管脚禁止，选择SPI的3线模式。

TXEPT：发送器空标志。该位为0时，表示发送缓冲区（UTXBUF）有数据。当位为1时，表示表示发送缓冲区（UTXBUF）空。

### 3. UxRCTL寄存器

UxRCTL寄存器是一个8位的寄存器。该寄存器控制USART模块的接收操作。图2-59给出了UxRCTL寄存器的各个位。

| FE | PE | OE | BRK | URXEIE | UTXWIE | RXWAKE | RXERR |
|---|---|---|---|---|---|---|---|

图 2-59　UxTCTL 寄存器

由图 2-59 可以看出，UxRCTL 寄存器主要包括 8 个有效的控制位。为了增加对 UxRCTL 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 UxRCTL 寄存器的各个位进行详细介绍。

FE：帧出错标志。当该位为 0 时，表示没有错误；当该位为 1 时，标志有错误，在同步模式下表示总线上有冲突。

PE：校验出错标志位。当该位为 0 时，表示校验正确；当该位为 1 时，表示校验出错。

OE：溢出标志位。当该位为 0 时，表示没有溢出发生；当该位为 1 时，表示有溢出发生。

BRK：打断检测位。当该位为 0 时，表示没有被打断；当该位为 1 时，表示被打断。

URXEIE：接收出错中断允许位。当该位为 0 时，不允许该中断；当该位为 1 时，允许该中断。

URXWIE：接收唤醒中断允许位。当该位为 0 时，接收到每一个字符都使标志位 URXIFGx 置位；当该位为 1 时，只有接收到地址字符才能设置 URXIFGx 标志位。

RXWAKE：接收唤醒标志。当该位为 0 时，接收到的字符是数据；当该位为 1 时，接收到的字符是地址。

RXERR：接收出错标志。当该位为 0 时，没有接收错误；当该位为 1 时，有接收错误发生。

### 4．UxBR0、UxBR1 和 UxMCTL 寄存器

UxBR0、UxBR1 和 UxMCTL 寄存器用来确定波特率。UxBR0 寄存器和 UxBR1 寄存器用来确定波特率的整数部分，其中 UxBR0 为低字节，UxBR1 为高字节，UxBR1 和 UxBR0 两个字节结合起来为 1 个 16 位的字，成为 UBR。UxMCTL 寄存器用来确定波特率的小数部分，如果波特率不是整数，带有小数，则小数部分由调整控制寄存器 UxMCTL 的内容反应。波特率的计算公式如下：

波特率=BRCLK/（UBR+（M7+M6+…M0）/8）

通过上面的计算公式设置 UxBR0、UxBR1 和 UxMCTL 寄存器可以得到需要的波特率。其中 M7、M6 和 M0 为 UxMCTL 寄存器的相应位，当需要设置的时候，相应位为 1，否则为 0。

### 5．UxRXBUF 和 UxTXBUF 寄存器

UxRXBUF 寄存器是用来接收数据的寄存器，当有数据的时候，从该寄存器里读出数据。UxTXBUF 寄存器是发送数据的寄存器，当有数据需要发送的时候，将数据写入到该寄存器里就可以了。

通过上面对 USART 模块的寄存器的介绍，对 USART 模块的操作有了具体的了解。通过上面对 USART 模块的介绍，知道 USART 模块有两种工作模式：异步模式和同步模式。关于 USART 模式的选择主要设置 UxCTL 寄存器，关于同步模式和异步模式的操作主要参看相应寄存器的某些控制位。

### 2.8.3 USART的应用例子

通过前面对USART模块的介绍，应该对USART有了基本的认识，下面结合具体的程序来说明USART的使用。下面的程序只是使用USART模块作为UART模块，具体的程序代码如下：

```
#include <msp430x14x.h>

void Init_UART0(void)
{
    U0CTL = 0X00;                    //将寄存器的内容清零
    U0CTL += CHAR;                   //数据位为8bit

    U0TCTL = 0X00;                   //将寄存器的内容清零
    U0TCTL += SSEL1;                 //波特率发生器选择SMCLK

    UBR0_0 = 0X45;                   //波特率为115200
    UBR1_0 = 0X00;

    UMCTL_0 = 0X49;                  //调整寄存器

    ME1 |= UTXE0 + URXE0;            //使能UART0的TXD和RXD
    IE1 |= URXIE0;                   //使能UART0的RX中断
    IE1 |= UTXIE0;                   //使能UART0的TX中断

    P3SEL |= BIT4;                   //设置P3.4为UART0的TXD
    P3SEL |= BIT5;                   //设置P3.5为UART0的RXD

    P3DIR |= BIT4;                   //P3.4为输出管脚

    return;
}
```

以上主要是对UART模块进行初始化操作，通过上面的程序可以看出，只要设置UART模块的相应寄存器就可以完成对UART模块的设置。

通过上面的介绍知道，USART模块的数据的收发主要是操作UxTXBUF寄存器和UxRXBUF寄存器。下面给出数据收发的程序。

```
interrupt [UART0RX_VECTOR] void UART0_RX_ISR(void)
{
    UART0_RX_BUF[nRX0_Len_temp] = RXBUF0;//接收来自的数据

    nRX0_Len_temp += 1;
    if(UART0_RX_BUF[nRX0_Len_temp - 1] == 13)
    {
        nRX0_Len = nRX0_Len_temp;
```

```
        nRev_UART0 = 1;
        nRX0_Len_temp = 0;
    }
}
interrupt [UART0TX_VECTOR] void UART0_TX_ISR(void)
{
    if(nTX0_Len != 0)
    {
        nTX0_Flag = 0;   //表示缓冲区里的数据没有发送完

        TXBUF0 = UART0_TX_BUF[nSend_TX0];
        nSend_TX0 += 1;
        Delay_us(5);
        if(nSend_TX0 >= nTX0_Len)
        {
            nSend_TX0 = 0;
            nTX0_Len = 0;
            nTX0_Flag = 1;
        }
    }
}
```

上面的程序主要完成数据的收发，数据的收发主要是采用中断程序进行处理，考虑到中断程序与主程序进行数据交互，这样需要设置一些全局的变量，也需要全局的缓冲区来交互数据。关于中断的使能在前面已经做了介绍，这里就不再叙述了。通过以上的程序可以看出，数据的发送主要就是往 UxTXBUF 寄存器里写数据，数据的接收主要是从 UxRXBUF 寄存器里读取数据。

## 2.9　MSP430F1XX 的 ADC 模块

在 MSP430F1XX 系列单片机里，有的型号的单片机（比如 MSP430F13X 和 MSP430F14X）有 ADC 模块，在该系列单片机里，ADC 模块为 12 位的 ADC 模块，叫做 ADC12。ADC12 模块支持快速的 12 位 A/D 转换。ADC12 模块应用了 12 位的 SAR 核、采样选择控制、参考产生和 16 位的转换控制缓冲区。转换控制缓冲区可以支持多达 16 个 ADC 采样转换存储。ADC12 模块主要有以下特点。

- 采样速度快。
- 在采样周期可以编程的情况下，采样保持的时间可以由软件或者定时器控制。
- 转换开始可以由软件、定时器 A 和定时器 B 实现。
- 片内参考电压的产生可以由软件编程选择，也可以由软件选择内部参考还是外部参考。
- 每个信道可以单独选择正极性或者负极性的参考源。
- 可以选择的转换时钟源。

- 具有单通道单次转换、单通道多次转换、序列通道单次转换和序列通道多次转换 4 种转换模式。
- ADC 转换核和参考电压能够单独关断以节省功耗。
- 具有中断矢量寄存器，这样可以快速解码 ADC 的各个不同中断。
- 16 位的转换结果存储寄存器。

用户对 ADC12 模块的所有操作都是通过操作该模块的寄存器完成的。ADC12 模块的寄存器比较多，大致可以分为 4 类：转换控制类、中断控制类、存储控制类和存储器类。其中转换控制类的寄存器有：ADC12CTL0 和 ADC12CTL1；中断控制类的寄存器有：ADC12IFG、ADC12IE 和 ADC12IV；存储控制类的寄存器有：ADC12MCTL0~ADC12MCTL15；存储器类寄存器有：ADC12MEM0~ADC12MEM15。下面结合具体的寄存器来介绍 ADC12 模块。

**1．ADC12 模块的转换控制类寄存器**

转换控制类寄存器主要包括 ADC12CTL0 和 ADC12CTL1 两个寄存器。这两个寄存器主要控制 ADC12 模块的工作。下面对这两个寄存器分别进行介绍。

ADC12CTL0 寄存器的位分配如图 2-60 所示。

| SHT1x(3...0) | | | | SHT0x(3...0) | | | | 高字节 |
|---|---|---|---|---|---|---|---|---|
| MSC | REF2_5V | REFON | ADC12ON | ADC12OVIE | ADC12TOVIE | ENC | ADC12SC | 低字节 |

图 2-60　ADC12CTL0 寄存器

由图 2-60 可以看出，ADC12CTL0 是一个 16 位的寄存器。为了增加对 ADC12CTL0 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 ADC12CTL0 寄存器的各个位进行详细介绍。

SHT1x（3...0）：采样保持时间。该 4 位定义了保存在转换结果寄存器 ADC12MEM8~ADC12MEM15 中的转换采样时序。采样周期是 ADC12CLK 周期乘 4 的整数倍，即当 SHT1x（3...0）的值为 2 时，采样周期为 ADC12CLK 周期的 8 倍。

SHT0x（3...0）：采样保持时间。该 4 位定义了保存在转换结果寄存器 ADC12MEM0~ADC12MEM7 中的转换采样时序。采样周期是 ADC12CLK 周期乘 4 的整数倍，即当 SHT1x（3...0）的值为 2 时，采样周期为 ADC12CLK 周期的 8 倍。

MSC：多次采样转换位。只有对序列采样或者多次转换有效。当该位为 0 时，采样定时器需要 SHI 信号的上升沿触发；当该位为 1 时，首次转换由 SHI 信号的上升沿触发采样定时器，后面的采样转换将在前一次转换完成后立即进行，不需要 SHI 信号的上升沿触发。

REF2_5V：内部参考电压的选择，这个时候 REFON 也必须设置。当该位为 0 时，内部参考电压为 1.5V；当该位为 1 时，内部参考电压为 2.5V。

REFON：内部参考电压控制位。当该位为 0 时，内部参考电压关闭；当该位为 1 时，内部参考电压打开。

ADC12ON：ADC12 模块的控制位。当该位为 0 时，关闭 ADC12 模块，不能进行转换；当该位为 1 时，打开 ADC12 模块，可以进行转换。

ADC12OVIE：ADC12MEMx 溢出中断使能。当该位为 0 时，溢出中断不使能；当该位为 1 时，溢出中断使能。

ADC12TOVIE：ADC12 模块的转换时间溢出中断使能。当该位为 0 时，转换时间溢出中断不使能；当该位为 1 时，转换时间溢出中断使能。

ENC：转换允许位。当该位为 0 时，转换不允许；当该位为 1 时，转换允许。

ADC12SC：开始转换位。当该位为 0 时，没有采样转换开始；当该位为 1 时，采样转换开始。

ADC12CTL1 寄存器的位分配如图 2-61 所示。

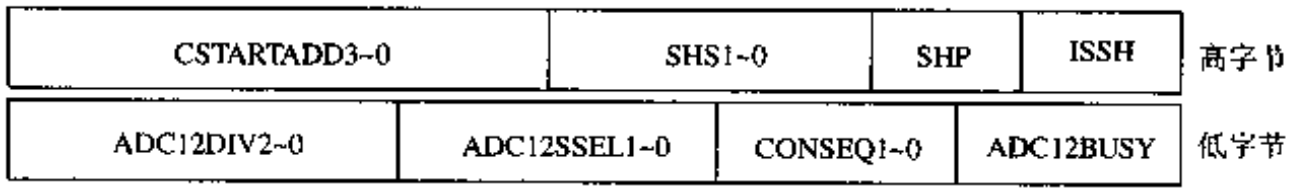



图 2-61　ADC12CTL0 寄存器

由图 2-61 可以看出，ADC12CTL1 是一个 16 位的寄存器。为了增加对 ADC12CTL1 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 ADC12CTL1 寄存器的各个位进行详细介绍。

CSTARTADD3～0：转换起始地址。该 4 位确定了单次转换的地址或者序列转换的首地址。该 4 位定义的二进制数 0～15 分别对应 ADC12MEM0～ADC12MEM15。

SHS1～0：采样保持信号源的选择。表 2-31 给出了 SHS1、SHS0 位与采样保持信号源的选择。

表 2-31　采样保持信号源的选择

| SHS1 | SHS0 | 时钟源的选择 |
|---|---|---|
| 0 | 0 | 选择 ADC12SC |
| 0 | 1 | 选择 Timer_A 的 OUT1 |
| 1 | 0 | 选择 Timer_B 的 OUT0 |
| 1 | 1 | 选择 Timer_B 的 OUT1 |

由表 2-31 可以看出，适当设置 SHS1 和 SHS0 位就可以选择相应的采样保持信号源。

SHP：采样保持脉冲模式选择。该位选择采样信号源（SAMPCON）。当该位为 0 时，SAMPCON 直接源自采样输入信号；当该位为 1 时，SAMPCON 直接源自采样定时器。

ISSH：采样信号反向控制位。当该位为 0 时，采样信号不反向；当该位为 1 时，采样信号反向。

ADC12DIV2～0：ADC12 时钟源分频因子。分频因子的值为该 3 位的值加 1。

ADC12SSEL1～0：ADC12 时钟源的选择。表 2-32 给出了 ADC12SSEL1、ADC12SSEL0 位与 ADC12 时钟源的选择。

表 2-32　ADC12 时钟源的选择

| ADC12SSEL1 | ADC12SSEL0 | ADC12 时钟源的选择 |
|---|---|---|
| 0 | 0 | 选择 ADC12 内部时钟源：ADC12OSC |
| 0 | 1 | 选择 ACLK |
| 1 | 0 | 选择 SMCLK |
| 1 | 1 | 选择 SMCLK |

由表 2-32 可以看出，适当设置 ADC12SSEL1 和 ADC12SSEL0 位就可以选择相应的 ADC12 时钟源。

CONSEQ1～0：转换模式选择位。表 2-33 给出了 CONSEQ1、CONSEQ0 位与转换模式选择的关系。

表 2-33　转换模式的选择

| CONSEQ1 | CONSEQ0 | 转换模式的选择 |
|---|---|---|
| 0 | 0 | 单通道单次转换模式 |
| 0 | 1 | 序列通道单次转换模式 |
| 1 | 0 | 单通道多次转换模式 |
| 1 | 1 | 序列通道多次转换模式 |

由表 2-33 可以看出，适当设置 CONSEQ1 和 CONSEQ0 位就可以选择相应的转换模式。

ADC12BUSY：ADC12 忙的标志。该位显示是否有采样或者转换操作。当该位为 0 时，没有任何操作；当该位为 1 时，有采样或者转换等操作。

通过以上对 ADC12CTL0 和 ADC12CTL1 寄存器的介绍可以知道：该寄存器主要控制 ADC12 模块的操作。

**2．ADC12 模块的中断控制类寄存器**

ADC12 模块的中断控制类寄存器主要对相应的中断进行处理。中断控制类的寄存器有：ADC12IFG、ADC12IE 和 ADC12IV，下面对这 3 个寄存器分别进行介绍。

ADC12IFG 寄存器为 ADC12 模块的中断标志寄存器，该寄存器为一个 16 位的寄存器。图 2-62 为该寄存器的位分配。

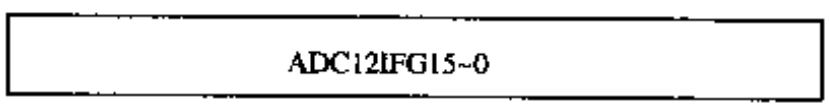



图 2-62　ADC12IFG 寄存器

由图 2-62 可以看出 ADC12IFG 寄存器共有 16 个中断标志位。它的某个位对应相应的 ADC12MEMx 寄存器。在转换结束后，转换结果装入转换存储器 ADC12MEMx 后置位，当 ADC12MEMx 存储器被访问后复位。

ADC12IE 寄存器为 ADC12 模块的中断允许寄存器，该寄存器为 16 位的寄存器。该寄存器与 ADC12IFG 寄存器一样对应于 16 个转换存储寄存器 ADC12MEMx。该寄存器的各个位使能相应的中断。

ADC12IV 寄存器是 ADC12 模块的中断向量寄存器。该寄存器是一个 8 位的寄存器，该

寄存器的位分配如图 2-63 所示。


图 2-63　ADC12IV 寄存器

由图 2-63 可以看出 ADC12IV 寄存器有 5 个有效的控制位，该 5 个控制位确定了 ADC12 模块的中断向量表，如表 2-34 所示。

**表 2-34　ADC12 模块的中断向量表**

| ADC12IV 的值 | 中断源 | 中断标志 | 优先级 |
|---|---|---|---|
| 02H | ADC12MEMx 溢出 | — | 最高 |
| 04H | 转换时间溢出 | — | |
| 06H | ADC12MEM0 中断 | ADC12IFG0 | |
| 08H | ADC12MEM1 中断 | ADC12IFG1 | |
| 0AH | ADC12MEM2 中断 | ADC12IFG2 | |
| 0CH | ADC12MEM3 中断 | ADC12IFG3 | |
| 0EH | ADC12MEM4 中断 | ADC12IFG4 | |
| 10H | ADC12MEM5 中断 | ADC12IFG5 | |
| 12H | ADC12MEM6 中断 | ADC12IFG6 | |
| 14H | ADC12MEM7 中断 | ADC12IFG7 | |
| 16H | ADC12MEM8 中断 | ADC12IFG8 | |
| 18H | ADC12MEM9 中断 | ADC12IFG9 | |
| 1AH | ADC12MEM10 中断 | ADC12IFG10 | |
| 1CH | ADC12MEM11 中断 | ADC12IFG11 | |
| 1EH | ADC12MEM12 中断 | ADC12IFG12 | |
| 20H | ADC12MEM13 中断 | ADC12IFG13 | |
| 22H | ADC12MEM14 中断 | ADC12IFG14 | |
| 24H | ADC12MEM15 中断 | ADC12IFG15 | 最低 |

由表 2-34 可以看出，只要根据 ADC12IV 寄存器的值就可以解析出中断，从而进行相应的处理。

### 3．ADC12 模块的存储控制类寄存器

ADC12 模块的存储控制类寄存器主要控制各个转换存储器的转换条件。中断控制类的寄存器有：ADC12MCTL0～ADC12MCTL15，下面对这几个寄存器进行介绍。

ADC12MCTLx（x 的值为 0～15）寄存器为 8 位的寄存器。该寄存器的位分配如图 2-64 所示。

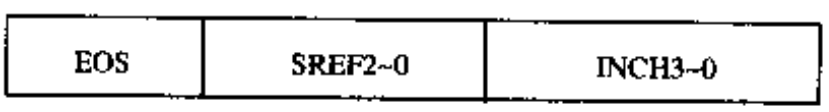



图 2-64　ADC12MCTLx 寄存器

由图 2-64 可以看出 ADC12MCTLx 寄存器有 3 个有效的控制字段。为了增加对 ADC12MCTLx 寄存器的了解，知道怎样对该寄存器进行正确的设置，下面对 ADC12MCTLx 寄存器的各个位进行详细介绍。

EOS：序列结束位。该位表明是序列中的最后一次转换。

SREF2～0：参考电压选择位。表 2-35 给出了 SREF2、SREF1、SREF0 位与参考电压的选择关系。

表 2-35　参考电压的选择

| SREF2 | SREF1 | SREF0 | 参考电压的选择 |
|---|---|---|---|
| 0 | 0 | 0 | $V_{R+}=AV_{CC}$；$V_{R-}=AV_{SS}$ |
| 0 | 0 | 1 | $V_{R+}=V_{REF+}$；$V_{R-}=AV_{SS}$ |
| 0 | 1 | 0 | $V_{R+}=Ve_{REF+}$；$V_{R-}=AV_{SS}$ |
| 0 | 1 | 1 | $V_{R+}=Ve_{REF+}$；$V_{R-}=AV_{SS}$ |
| 1 | 0 | 0 | $V_{R+}=AV_{CC}$；$V_{R-}=V_{REF-}/Ve_{REF-}$ |
| 1 | 0 | 1 | $V_{R+}=V_{REF+}$；$V_{R-}=V_{REF-}/Ve_{REF-}$ |
| 1 | 1 | 0 | $V_{R+}=Ve_{REF+}$；$V_{R-}=V_{REF-}/Ve_{REF-}$ |
| 1 | 1 | 1 | $V_{R+}=Ve_{REF+}$；$V_{R-}=V_{REF-}/Ve_{REF-}$ |

由表 2-35 可以看出，适当设置 SREF2、SREF1 和 SREF0 位就可以选择相应的参考电压。

INCH3～0：选择模拟通通。该 4 位所表示的值为所选择的模拟输入通道。表 2-36 给出了这 4 个位与输出通道选择的关系。

表 2-36　模拟输入通道的选择

| INCH3～0 的值 | 模拟输入通道的选择 |
|---|---|
| 0～7 | A0～A7 |
| 8 | $Ve_{REF+}$ |
| 9 | $V_{REF-}/Ve_{REF-}$ |
| 10 | 片内温度传感器的输出 |
| 11～15 | $(AV_{CC}-AV_{SS})/2$ |

由表 2-36 可以看出，适当设置 INCH3、INCH2、INCH1 和 INCH0 位就可以选择相应的模拟输入通道。

### 4．ADC12 模块的存储器类寄存器

ADC12 模块的存储类寄存器主要用来存储转换得到的数据。存储类寄存器有：ADC12MEM0～ADC12MEM15。这 16 个寄存器为 16 位的寄存器，在存储数据的时候，只使用了低 12 位，因此数据的有效位为 12 位。

通过以上对 ADC12 模块寄存器的介绍，对 ADC12 模块有了基本的了解，下面给出 ADC12 模块的应用程序，具体的程序代码如下：

```
#include <msp430x14x.h>

void Init_ADC()
{
    P6SEL = 0X07;          //设置P6.0～P6.3为模拟输入通道

    ADC12CTL0 &= ～(ENC); //设置ENC为0，从而修改ADC12寄存器的值
    ADC12CTL1 |= CSTARTADD_0;//转换的起始地址为：ADCMEM0

    ADC12MCTL0 = INCH_0;//设置参考电压分别为AVss和AVcc，输入通道为A0
    ADC12MCTL1 = INCH_1;//设置参考电压分别为AVss和AVcc，输入通道为A1
    ADC12MCTL2 = INCH_2;//设置参考电压分别为AVss和AVcc，输入通道为A2
    ADC12MCTL3 = INCH_3 + EOS;//转换通道的结束处

    ADC12CTL0 |= ADC12ON;
    ADC12CTL0 |= MSC;

    ADC12CTL1 |= CONSEQ_1;//转换模式为：多通道、单次转换
    ADC12CTL1 |= ADC12SSEL_1;//SMCLK
    ADC12CTL1 |= ADC12DIV_0;//时钟分频为1
    ADC12CTL1 |= (SHP);//采样脉冲由采用定时器产生

    ADC12CTL0 |= ENC;//使能ADC转换
    return;
}
void Init_TimerA()
{
    TACTL = TASSEL1 + TACLR; //选择SMCLK，清除TAR
    CCTL0 = CCIE;          //CCR0 中断允许
    CCR0 = 40000;          //200Hz
    TACTL |= MC0;          //增记数模式
}
interrupt [TIMERA0_VECTOR] void TimerA_ISR(void)
{
    int results[3];

    ADC12CTL0 &= ～ENC;//关闭转换

    results[0] = ADC12MEM0;//读出转换结果
    results[1] = ADC12MEM1;//读出转换结果
    results[2] = ADC12MEM2;//读出转换结果
    results[3] = ADC12MEM3;//读出转换结果

    ADC12CTL0 |= ENC + ADC12SC;//开启转换
}
```

通过以上的程序可以看出，只要设置好相应的寄存器就可以控制 ADC12 模块的工作，

通过读取 ADC12MEMx 寄存器获得采集得到的数据。上面的程序是通过定时器 A 来进行采集的启动和停止，在定时器 A 的中断服务程序里进行处理的。

通过本章对 MSP430F1XX 系列单片机的 CPU 和外设进行的介绍，相信读者对 MSP430F1XX 系列单片机有了较为深入的了解。

# 第 3 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：可以大大提高软件开发的工作效率；可以提高程序的可靠性、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。

## 3.1　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。

### 3.1.1　标识符与关键字

**1．标识符**

C 语言中的标识符可以作为变量名、函数名、数组名、类型名以及文件名。它可以是一个字符，也可以是多个字符。标识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，标识符要求区分大小写，也就是说大写和小写的标识符被当作不同的标识符。例如：Index 和 index 是两个不同的标识符，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识符。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，unsinged，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的含义，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。

### 3.1.2 数据的基本类型

标准C语言中主要有整型、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。不同的整型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3-1给出了MSP430的C语言支持的几种整数类型。

表3-1 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，有两种常用的整数形式：十进制和十六进制。十六进制与十进制不同的是在数值的前面加上 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 这几种类型。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 类型。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’，这里需要强调一下的是：字符是以单引号表示。

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，通常使用转义字符表示 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。

### 3.1.3 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

算术运算主要是进行一些加、减、乘、除等运算。表 3-2 给出了算术运算符。

表 3-2 算术运算符

| 算术运算符 | 含义 | 说明 |
|---|---|---|
| ++ | 一元加 | 只有一个操作数 |
| -- | 一元减 | 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。表 3-3 给出了关系运算符。

表 3-3　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 3-3 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。表 3-4 给出了逻辑运算符。

表 3-4　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 3-4 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。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，其含义和上面的相同，所以可以触类旁通。

**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。下面通过例子来进行理解。

```
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，可能一个计算包括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 3-5 给出了运算的优先级。

表 3-5　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 3-5 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 3-5 所列出的优先级来进行运算。

### 3.1.4　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。这样选择结构就是包括多个分支，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 3 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2:
        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知道的情况下，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3是循环计数器的改变表达式，表达式2是判断表达式，当该表达式的值为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,m;
    sum = 0;
    i = 0;
    for(; i < 100; )
    {
        sum += i;
        i+= 1;
}
}
```

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 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。

### 3.1.5 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;
    return;
}
```

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 的话，可以不用返回值，其他类型必须返回函数值，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**

在引入了函数概念后，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，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，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
    a=8;
    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，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**

函数定义时的参数称为形式参数，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，传入的参数称为实际参数，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。

### 3.1.6　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：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，也可以赋部分值。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访问差不多。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：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。

### 3.1.7　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：

一种按变量名存取。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”。

另一种方式是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“*”表示是指针类型的变量。类型表示指针所指向的变量的类型。

例：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，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，即通过指向数组元素的指针所需的元素。指向数组元素的指针可以定义一个指针，然后指向它的首地址。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。

### 3.1.8 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**

结构型变量的引用格式为：

结构型变量名.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;
```

### 3.1.9 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 标识符 常量表达式
```

其中 define 是关键字，它表示该命令为宏，标识符是宏符号名，它的含义是后面的常量表达式。标识符最好大写，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

上面介绍了简单宏的定义，宏在定义的时候也可以带参数，带参数的宏的定义如下：

```
#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）

声明函数原型，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：**

```
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，也可以由标识符确定，下面给出两种具体的格式。

```
#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。

条件还有另外一种定义，格式如下：

```
#ifdef 标识符
程序段1
#else
程序段2
#endif
```

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

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

## 3.2　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。

### 3.2.1　MSP430 的 C 语言扩展概述

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、本征函数以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：

- I/O 访问有关的关键字：sfrb、sfwb。
- 非易失 RAM 关键字：no_init。
- 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 bitfield = default
#pragma bitfield = 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_SYSTEM_ICC | IAR C 编译器的标识符 |
| __LINE__ | 当前源程序行号 |
| __STDC__ | ANSI-C 编译器的标识符 |
| __TID__ | 目标标识符 |
| __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 |

## 3.2.2　MSP430 的 C 语言的关键字扩展

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;                         //增记数模式

    //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：函数可以有返回值，也可以没有返回值。

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

下面举例说明该关键字的用法。下面的例子演示在测试标志时禁止中断。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。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 标识符=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 标识符=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。

### 3.2.3 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得默认存储顺序如下：

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

实际结构如图 3-1 所示。

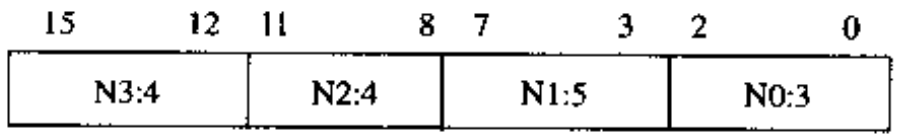



图 3-1 默认存储顺序

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
```

图 3-2 为翻转位域的存储顺序。

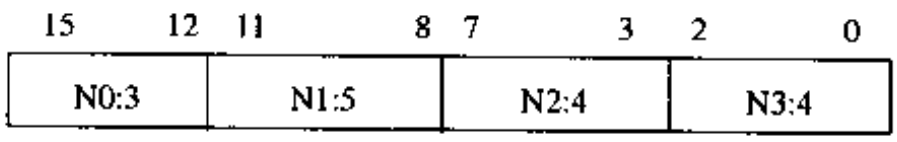



图 3-2 翻转存储顺序

通过图 3-1 和图 3-2 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：下例把代码段定义为 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，下面声明外部函数 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，下面的函数执行期间，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，下面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**

默认情况下，将常数放入所命名的段。

语法：#pragma memory = constseg(段名)

说明：默认情况下，将常数放入所命名的段。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，下面将常量数组 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**

默认情况下，将变量放入所命名的段。

语法：#pragma memory = dataseg(段名)

说明：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 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**

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，下面将变量 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以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。

### 3.2.4 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_SYSTEM_ICC**

IAR C 编译器标识符。

语法：__IAR_SYSTEM_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__**

当前源文件行号。

语法：__LINE__

说明：返回当前在编译的源文件的行号。

**5. __STDC__**

标准 C 编译器标识符。

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__**

目标标识符。

语法：__TID__

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

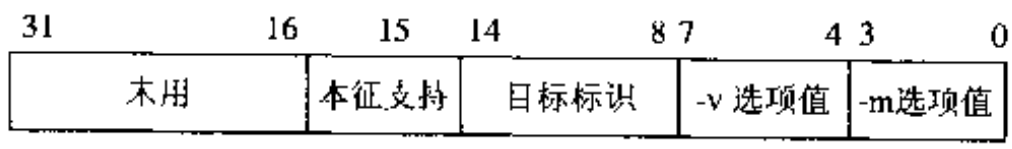



图 3-3 目标标识符

**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，下面测试版本号是否是 3.33

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

### 3.2.5　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，如表 3-6 所示。

表 3-6　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 3-6 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，具体的类型如表3-7所示。

表3-7 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3-7可以知道函数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：

```
//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：

```
//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：

```
_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，下面延迟一点时间

```
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。

### 3.2.6 MSP430 的段定义

通过前面一章，我们知道 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 3-4 所示。

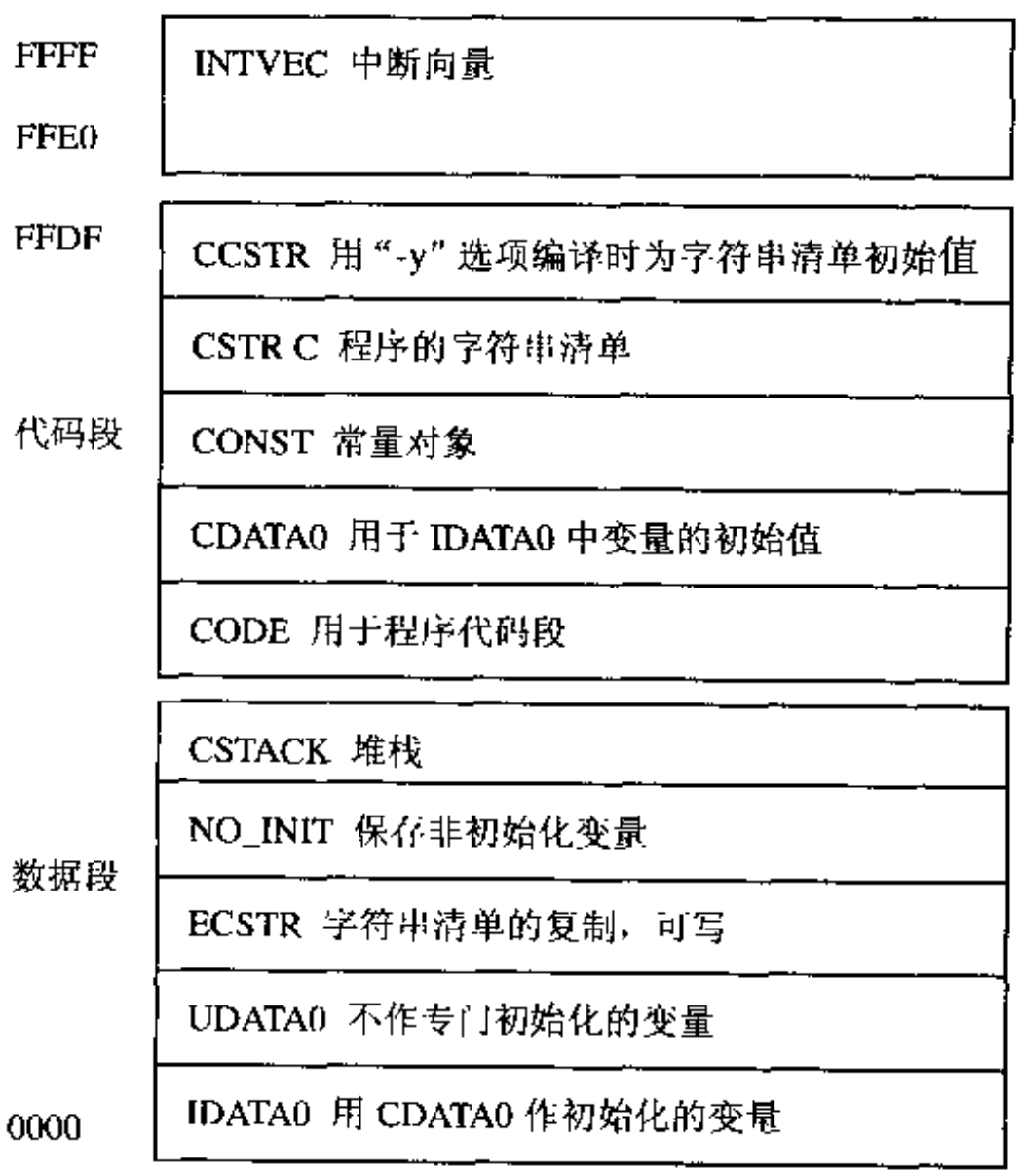



图 3-4　存储器段的示意图

图 3-4 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 3-4 可以对存储器的分段有一个宏观的认识，下面就对具体的各个段加以说明。

**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（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。该段不作显式初始化，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，读者应该对采用 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一个坚实的基础。

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，另外也提供了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：相关头文件

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

#ifndef __IN430_INCLUDED
#define __IN430_INCLUDED

#if __TID__ & 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 __TID__ & 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 1.7976931348623158e+308
#endif
#endif

#define __EDOM_VALUE    HUGE_VAL

/* PI, PI/2, PI/4, 1/PI, 2/PI */
#define __PI            3.14159265358979323846264643
#define __PIO2          1.57079632679489661923
#define __PIO4          .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         2.41421356237309504880
#define __SQ2M1         .41421356237309504880
#define __SQRTO2        0.70710678118654752440

/* LN(10), TWO-LOG(e), LN(2) e */
#define __LN10          2.30258509299404568402
#define __LOG2E         1.44269504088896340735992468
#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 __IAR_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
```

# 第 4 章　温度采集报警系统的实现

湿度采集广泛应用于人们的生产和生活中，使用温度计来采集温度，这样不但采集精度低、实时性差，而且操作人员的劳动强度大。本章介绍一种采用集成温度传感器 MAX6613 作为检测元件，MSP430F149 作为 CPU 的温度监控系统，该系统可以方便地实现温度实时监控。

## 4.1　原理简介

温度采集系统主要通过温度传感器MAX6613采集得到温度数据，MSP430F149 作为CPU 从温度传感器读取数据，将得到的数据进行判断然后做相应的处理，比如显示或者报警。温度传感器通过某种关系的换算，就可以得到温度与输出电压的关系，对于 MAX6613 来说，其输出的电压与温度的关系如图 4-1 所示。

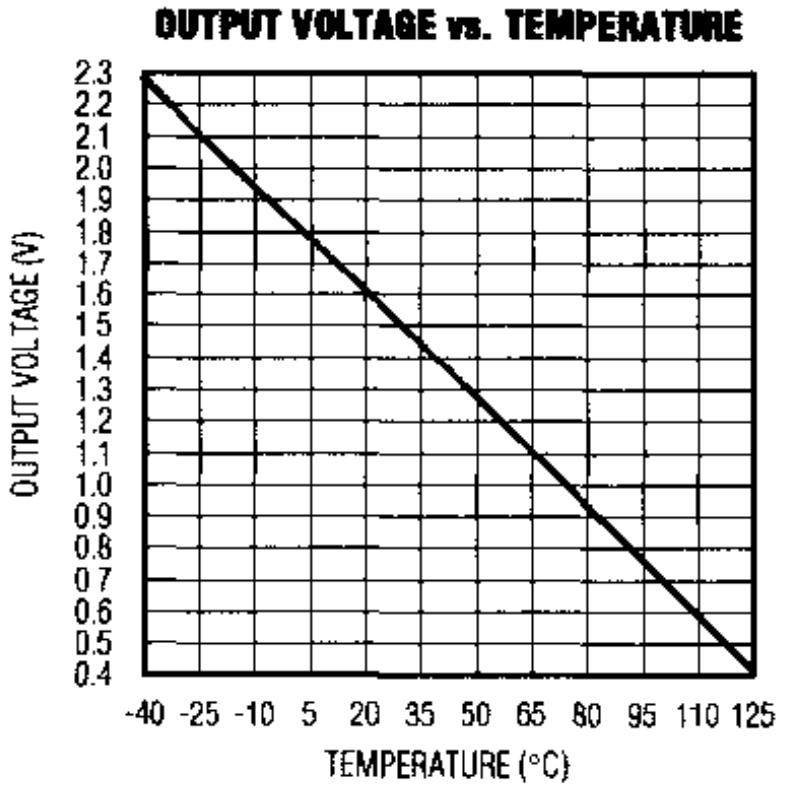



图 4-1　MAX6613 传感器输出的电压与温度的关系

为了能够便于计算，得到温度与电压的转换等式为：

$$V_{OUT} = -0.0000022 \times T^2 - 0.01105 \times T + 1.8455V \tag{4-1}$$

但是在大多数情况下，采用下面的线性关系式也可以完成转换运算。

$$V_{OUT} = -0.01123 \times T + 1.8455V \tag{4-2}$$

这样单片机通过模拟口采集得到传感器输出的电压，通过设置的参考电压就可以得到传

感器的输入电压，再通过上面式（4-2）就可以获得温度参数，将得到的温度参数进行分析后进行相应的处理，比如显示或者报警。由于 MSP430F149 片内集成了 A/D 转换通道，这样可以直接将单片机的 A/D 输入通道与传感器的模拟电压输出通道相连接。另外系统通过键盘输入来完成对报警温度的上限和下限的设置，通过显示电路将得到的数据显示出来，当温度超过上限和下限的时候，系统进行报警，报警是通过驱动一个蜂鸣器来实现。下面一部分将具体介绍系统的构成。

## 4.2 系统功能描述

系统主要由键盘输入模块、传感器采集模块、显示模块、报警模块和 CPU 处理模块等组成，整个系统的原理框图如图 4-2 所示。

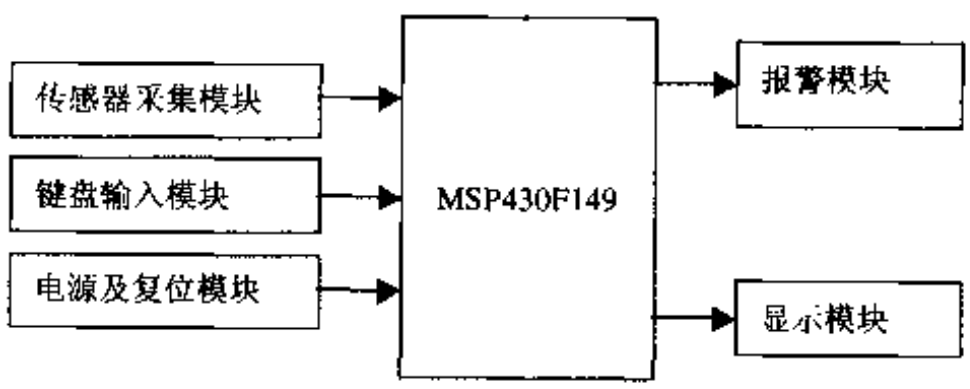



图 4-2 系统的原理框图

由图 4-2 可以看出，整个系统具有结构简单等特点。传感器模块与单片机的 A/D 通道进行连接，这样可以简化模拟采集的设计，从而可以减小设计的复杂性，增加系统的可靠性，也同时减小了 PCB 的面积。键盘输入模块是通过单片机的 P1 口来实现的，由于 P1 口具有中断功能，所以实现起来非常容易，并且也非常适合软件编程。电源及复位模块主要是为整个系统提供可靠的电源，另外考虑到系统工作需要有复位功能，因此也为系统提供复位信号。报警模块主要是单片机在检测到报警条件时，给一个报警信号，从而驱动蜂鸣器实现报警功能。显示模块主要是为了将得到数据显示出来，这样便于实时观察。下面一节将详细介绍各个功能模块的硬件设计。

## 4.3 系统硬件设计

通过上面的介绍，知道整个系统包括：键盘输入模块、传感器采集模块、显示模块、报警模块、CPU 处理模块和电源及复位模块，下面就具体的电路进行介绍。

**1．电源电路**

整个系统采用3.3V供电，考虑到硬件系统对电源要求具有稳压功能和纹波小等特点，另外也考虑到硬件系统的低功耗等特点，因此该硬件系统的电源部分采用TI公司的TPS76033芯片实现，该芯片能很好满足该硬件系统的要求，另外该芯片具有很小的封装，因此能有效节约PCB板的面积。电源电路具体如图4-3所示。

为了使输出电源的纹波小，在输出部分用了一个 2.2μF 和 0.1μF 的电容，另外在芯片的输入端也放置一个 0.1μF 的滤波电容，减小输入端受到的干扰。

### 2．复位电路

在单片机系统里，单片机需要复位电路，复位电路可以采用 R-C 复位电路，也可以采用复位芯片实现的复位电路，R-C 复位电路具有经济性，但可靠性不高，用复位芯片实现的复位电路具有很高的可靠性，因此为了保证复位电路的可靠性，该系统采用复位芯片实现的复位电路，该系统采用 MAX809 芯片。复位电路如图 4-4 所示。

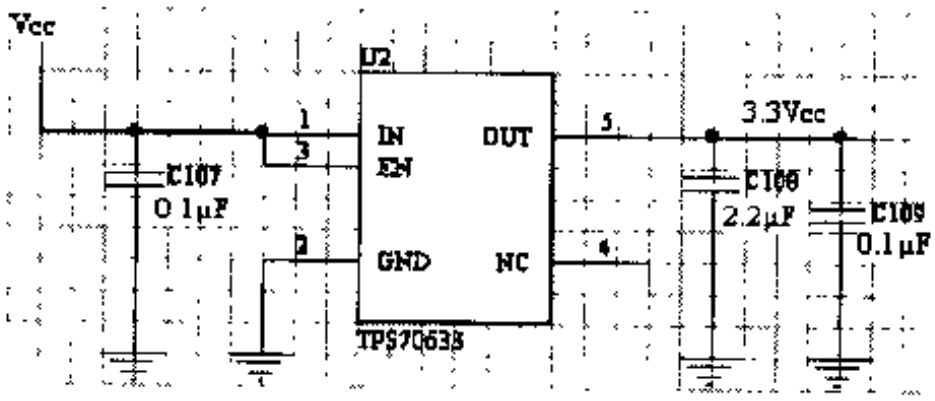



图 4-3　电源电路

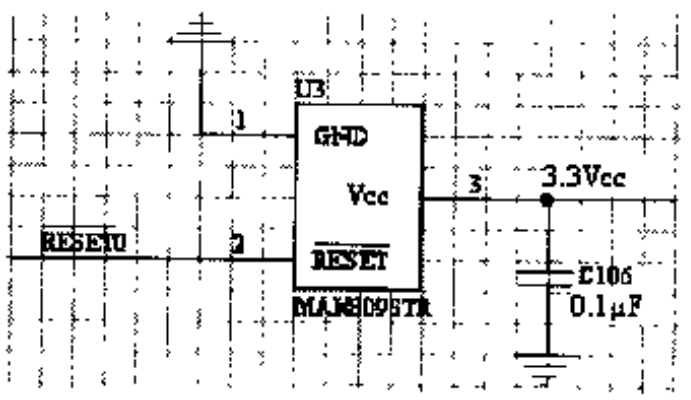



图 4-4　复位电路

为了减小电源的干扰，还需要在复位芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 3．键盘输入电路

键盘电路主要是用来输入数据，从而实现人机交互。该系统的键盘设计是采用扫描方式实现的矩阵键盘。键盘的电路图如图 4-5 所示。

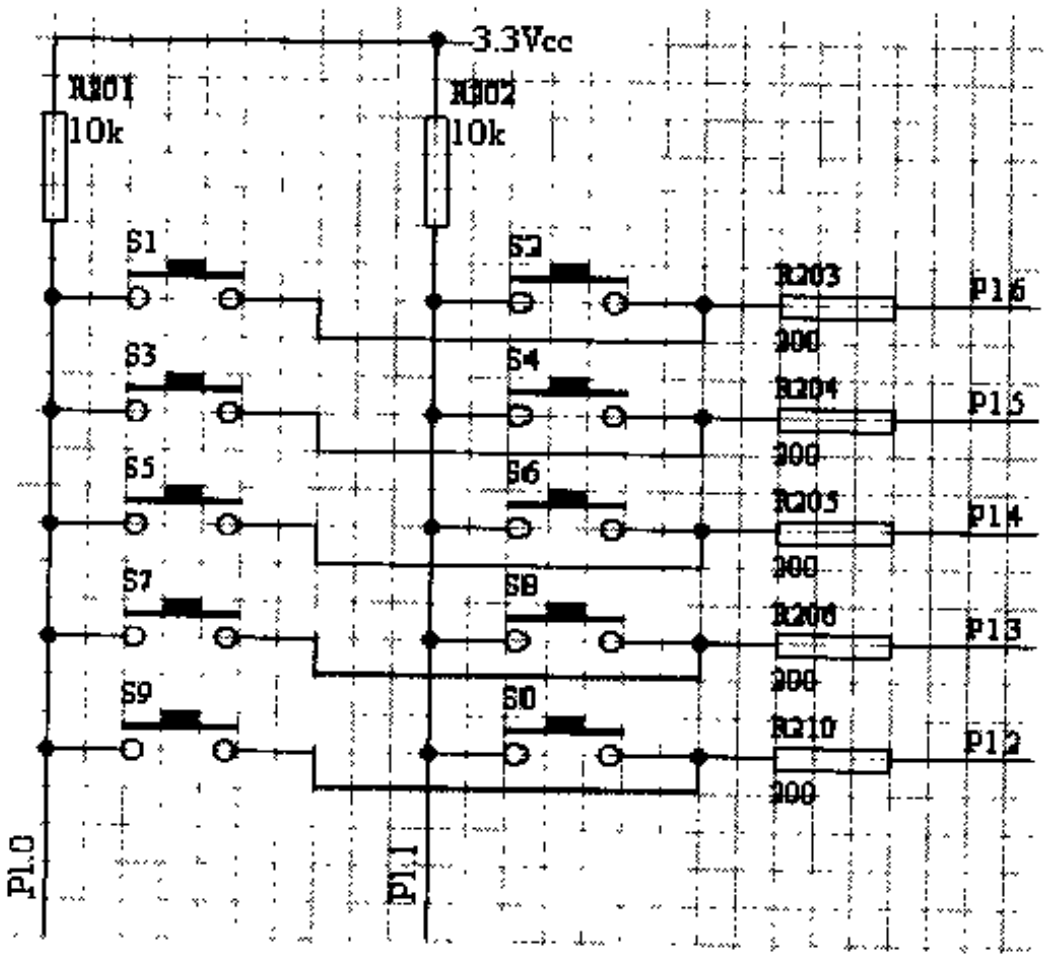



图 4-5　键盘输入电路

由图 4-5 可以看出该矩阵扫描键盘由行线和列线组成，P1.0 和 P1.1 构成了键盘的列线，P1.2、P1.3、P1.4、P1.5 和 P1.6 构成了键盘的行线。键盘的行线作为键盘的控制输出端，键盘的列线作为键盘的输入端。在设计时为了程序设计的方便性，键盘的列线采用的是 P1.0 和 P1.1，这样可以利用该管脚的中断功能。键盘的列线 P1.0 和 P1.1 通过上拉电路将该两个管脚

拉高，这样在没有按键按下的情况下，该两个管脚的电平为高电平，如果有按键按下时，则相应的列线管脚为低电平，这时通过设置 P1.0 和 P1.1 为低电平触发中断方式，低电平就触发中断而进入中断服务程序，从而获得输入的数据。具体分析一下键盘的工作原理，首先将 P1.3、P1.4、P1.5 和 P1.6 设置为输出，将 P1.0 和 P1.1 设置为输入，并将 P1.0 和 P1.1 设置成低电平中断触发方式；将 P1.6 设置为低电平，如果该行上有按键按下的话，则 P1.0 或者 P1.1 上为低电平，就会触发中断，进入中断服务程序，获得输入的数据。如果没有键按下的话，则 P1.0 和 P1.1 均为高电平，不会进入中断服务程序。依次将 P1.5、P1.4、P1.3 和 P1.2 设置为低电平来判断该行是否有输入，如果没有输入的话，P1.0 和 P1.1 均为高电平，如果有输入的话，P1.0 或者 P1.1 上为低电平，就会触发中断，进入中断服务程序，获得输入的数据。键盘的扫描时间时很短的，仅仅几微秒的时间，然而按键的时间一次至少需要几十毫秒，所以只要有键按下的话是都可以被扫描到的，但是按键按下时有一定的时间抖动，因此一定要考虑键盘的抖动处理。

**4．显示电路**

系统的显示电路采用的是简单的 LED 显示方式，这样的方式能满足该系统的要求，也可以减低系统的成本。图 4-6 为该系统的显示电路。

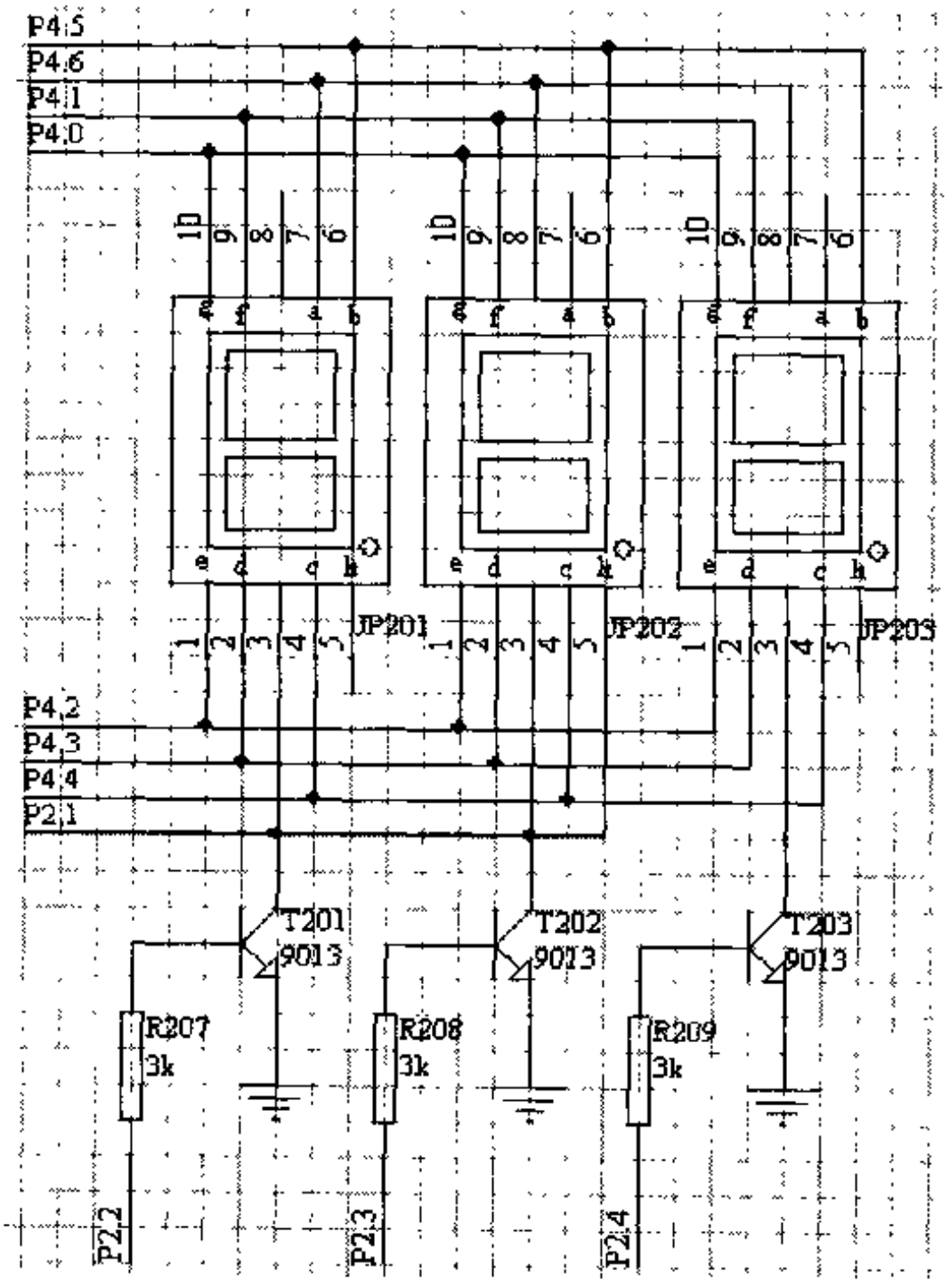



图 4-6　显示电路

通过图4-6可以看出，该显示电路直接与单片机的数据I/O口进行连接，由于MSP430F149具有丰富的 I/O 口资源，这样采用并行的接口方式非常容易，减小了系统设计的复杂度，也可以增加系统的可靠性。P4.0～P4.6 是用来显示数据，P2.1 是用来控制小数点的显示，P2.2、P2.3 和 P2.4 是用来控制数码管的选通状态，比如要在 DIS0 上显示，则要在 P2.2 管脚上给出高电平，通过三极管选通数码管进行显示。

**5．温度采集电路**

该系统的温度采集部分是采用 MAX6613 温度传感器来采集温度数据，作为单片的温度传感器，该芯片具有以下特点：

- 宽电压供电。该芯片的供电电压为 1.8V～5.5V。
- 较高的精度。该芯片的精度为 1.3℃。
- 大的测量范围。该芯片的温度范围为-55℃～130℃。
- 较小的封装。该芯片具有很小的封装，采用的是 SC70 封装。

为了对 MAX6613 有一个清楚的认识，图 4-7 给出了它的管脚图。

通过图 4-7 可以看出，该芯片只有 5 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。

- NC：该管脚在设计的时候不连。
- GND：电源地。
- VCC：电源输入端。
- OUT：采集的温度输出端。

由于该系统采用的是 MAX6613 作为温度采集传感器，这样该系统的这部分电路将非常简单，图 4-8 为该系统的温度采集电路。

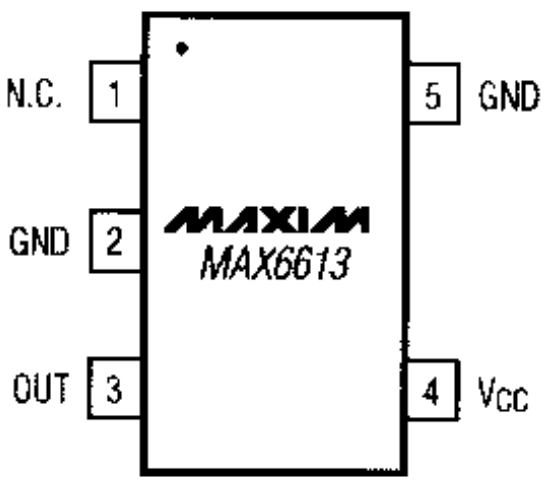



图 4-7　MAX6613 的管脚图

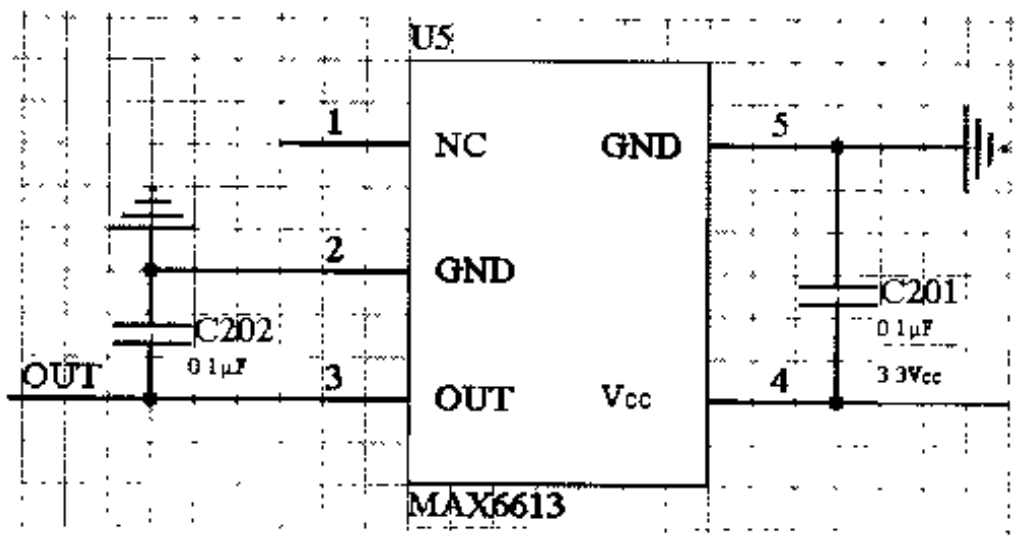



图 4-8　温度采集电路

通过 4-8 图可以看出，该采集电路具有简单、实用等特点。为了减少电源的输入纹波对采集电路的影响，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。

**6．报警电路**

该部分电路主要是驱动一个蜂鸣器，这样只需要将蜂鸣器的一端接地，另外一端与单片机进行相接就可以了，考虑到 MSP430F149 的驱动能力，需要加一个放大电路。图 4-9 为报警电路的实现。

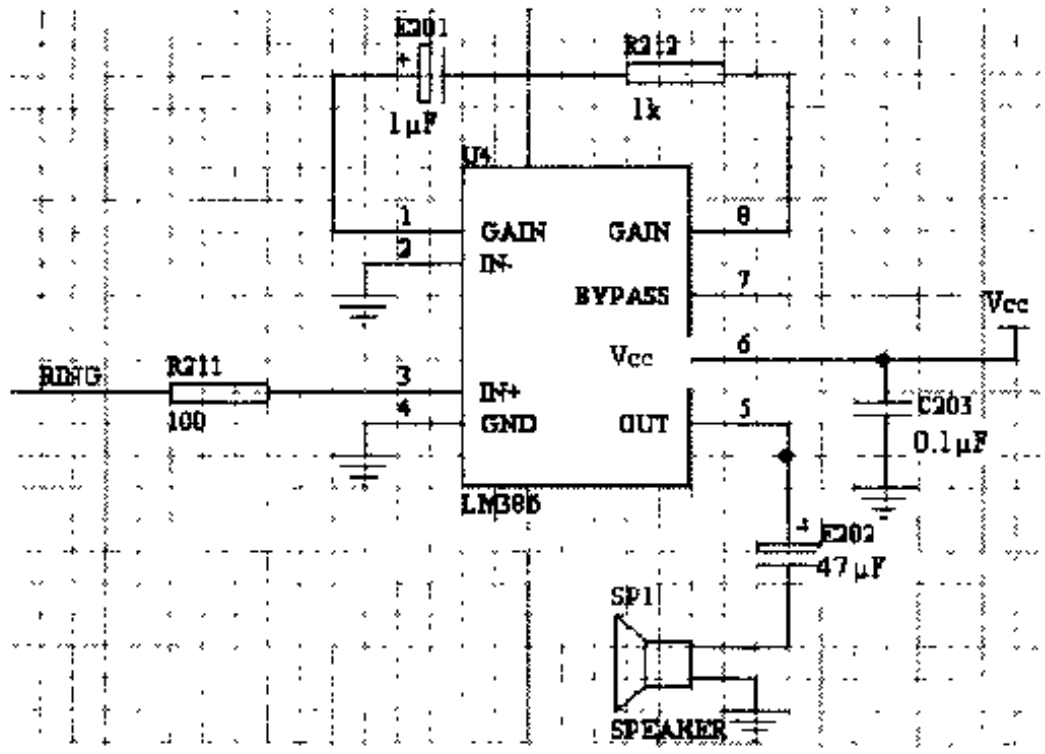



图 4-9　报警电路

通过图 4-9 可以看出，该报警电路具有简单、实用等特点。为了减少电源的输入纹波对放大电路的影响，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。

**7．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，从而获得数据进行处理，将处理的结果采用某种方式表示出来，比如显示或者报警。图 4-10 为单片机电路。

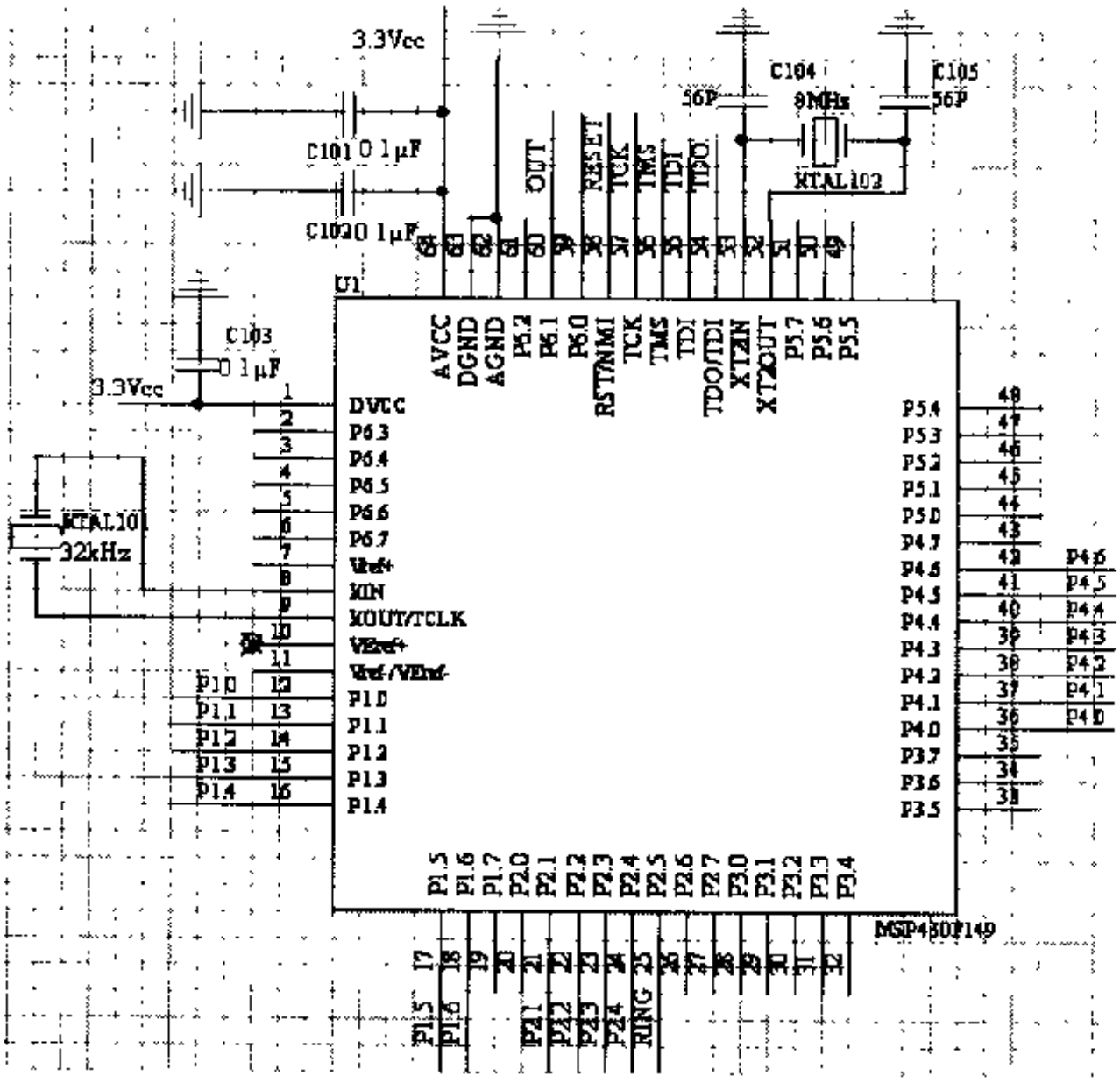



图 4-10　单片机电路

通过图 4-10 可以看出，单片机的接口电路非常简单，分别采用单片机的一般 I/O 口实现与其他电路的接口，在单片机的时钟设计上与其他单片机有一定的区别，MSP430F149 单片机采用两个时钟输入，一个 32kHz 的时钟信号，一个 8MHz 的时钟信号。该系统的时钟部分都是采用晶体振荡器实现的。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。另外单片机还有模拟电源的输入端，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。

经过该节的介绍，对系统的硬件系统有了清楚的认识，下一节介绍系统的软件设计。

## 4.4　系统软件设计

经过前面对系统硬件的了解，这一节介绍系统的软件设计。系统的软件主要包括采集模块、输入模块、显示模块、报警模块和主处理模块。下面就具体的各个模块进行介绍。

**1．采集模块**

采集模块重要是获得 MAX6613 温度传感器的数据，该部分主要是通过 MSP430F149 片内的 A/D 转换来完成数据的采集任务，采集的模拟参考电压采用片内的参考电压。数据采集的间隔时间通过定时器 A 来完成，就是在每次定时器 A 中断到来时读取 A/D 转换采集得到的数据，在读数据之前先停止 A/D 转换，在读取数据完毕后启动 A/D 转换，如果得到数据，则设置一个标志位通知主程序，告诉主程序已经得到新的数据。整个模块采用的是中断服务程序的结构完成。图 4-11 为该模块的程序流程图。

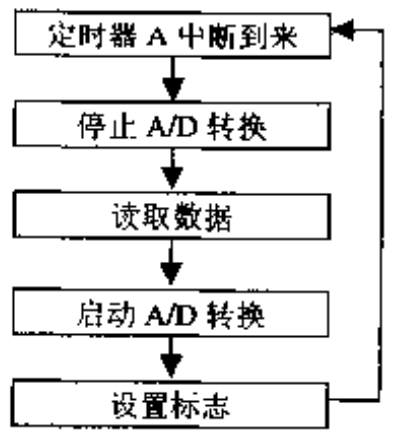



图 4-11　采集模块程序流程图

该模块主要涉及到 A/D 转换和定时器 A 的操作。下面分析该模块的程序设计。

初始化部分：该部分主要完成 A/D 转换和定时器 A 的初始化功能。下面为初始化部分的代码程序。

A/D 初始化程序如下。

```
void Init_ADC(void)
{
    P6SEL = 0X07;                          //设置P6.0为模拟输入通道

    ADC12CTL0 &= ~(ENC);                   //设置ENC为0，从而修改ADC12寄存器的值
```

```
    ADC12CTL1 |= CSTARTADD_0;     //转换的起始地址为：ADCMEM0
    ADC12MCTL0 = INCH_0 + EOS;    //设置参考电压分别为AVSS和AVCC，输入通道为A0

    ADC12CTL0 |= ADC12ON;
    ADC12CTL0 |= MSC;

    ADC12CTL1 |= CONSEQ_1;        //转换模式为：多通道、单次转换

    ADC12CTL1 |= ADC12SSEL_1;     //SMCLK
    ADC12CTL1 |= ADC12DIV_0;      //时钟分频为1
    ADC12CTL1 |= (SHP);           //采样脉冲由采用定时器产生

    ADC12CTL0 |= ENC;             //使能ADC转换
    return;
}
```

通过以上的程序代码可以看出，只要适当设置 A/D 转换的寄存器，就能使 A/D 模块正确的工作。

定时器 A 的初始化程序如下。

```
void Init_TimerA(void)
{
    TACTL = TASSEL1 + TACLR;      //选择SMCLK，清除TAR
    TACTL += ID1;
    TACTL += ID0;                 //1/8 SMCLK
    CCTL0 = CCIE;                 //CCR0 中断允许
    CCR0 = 4000;                  //时间间隔为 250Hz
    TACTL |= MC0;                 //增记数模式

    return;
}
```

通过上面的程序代码可以看出，通过设置 CCR0 就可以设置定时器中断的频率，从而实现采样时间间隔的控制。

定时器 A 处理和 A/D 转换部分：该部分主要完成数据的采集，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实现数据的交互。这部分程序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 转换部分的代码程序。

```
//定时器中断，完成 ADC 转换
interrupt [TIMERA0_VECTOR] void TimerA_ISR(void)
{
    int results;

    ADC12CTL0 &= ~ENC;                    //关闭转换
```

```
    results = ADC12MEM0;                    //读出转换结果
    ADC_BUF[nADC_Count] = results;

    nADC_Count += 1;
    if(nADC_Count == 10)
    {
        nADC_Flag = 1;                      //设置标志
for(int i = 0;i < 10;i++) ADC_BUF_Temp[i] = ADC_BUF[i];
        nADC_Count = 0;
    }

    ADC12CTL0 |= ENC + ADC12SC;             //开启转换
}
```

以上程序使用了全局变量 nADC_Flag，通过 nADC_Flag 变量通知主程序有新的采集数据获得；全局变量 nADC_Count 用来记数处理；ADC_BUF_Temp[]和 ADC_BUF[]全局变量用来存放数据，ADC_BUF_Temp[]作为与主程序交换数据的缓冲区。

**2．输入模块**

系统的输入部分由为矩阵扫描键盘。P1.0 和 P1.1 构成了键盘的列线，P1.2、P1.3、P1.4、P1.5 和 P1.6 构成了键盘的行线。该部分主要完成数据的输入功能，键盘的工作原理是这样的，首先将 P1.3、P1.4、P1.5 和 P1.6 设置为输出，将 P1.0 和 P1.1 设置为输入，并将 P1.0 和 P1.1 设置成低电平中断触发方式。将 P1.6 设置为低电平，如果该行上有按键按下的话，则 P1.0 或者 P1.1 上为低电平，就会触发中断，进入中断服务程序，获得输入的数据。如果没有键按下的话，则 P1.0 和 P1.1 均为高电平，不会进入中断服务程序。依次将 P1.5、P1.4、P1.3 和 P1.2 设置为低电平来判断该行是否有输入，如果没有输入的话，P1.0 和 P1.1 均为高电平，如果有输入的话，P1.0 或者 P1.1 上为低电平，就会触发中断，进入中断服务程序，获得输入的数据。为了防止键盘的扫描而影响其他部分的处理，这里采用定时器 B 来检察是否有按键按下。图 4-12 为该部分的程序流程图。

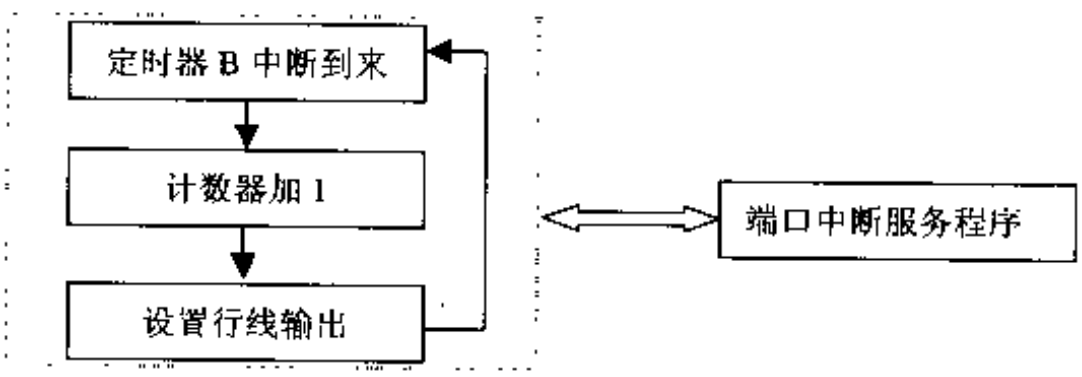



图 4-12　输入模块流程图

该模块主要包括对定时器 B 和端口中断的处理。下面具体分析该模块的程序设计。

初始化部分：该部分主要完成端口的初始化和定时器 B 的初始化。下面为初始化部分的代码程序。

该部分为端口的初始化程序。

```
void Init_INPUTPort(void)
{
    //将管脚在初始化的时候设置为输入方式
    P1DIR = 0;
    //将所有的管脚设置为一般I/O口
    P1SEL = 0;

    //将P1.0和P1.1设置为输入方向
    P1DIR &= ~(BIT1);
    P1DIR &= ~(BIT2);
    //将P1.2~P1.6设置为输出方向
    P1DIR |= BIT2;
    P1DIR |= BIT3;
    P1DIR |= BIT4;
    P1DIR |= BIT5;
    P1DIR |= BIT6;
    //将P1.0和P1.1设置为低电平中断触发方式
    P1IE |= BIT0;      //管脚P1.0使能中断
    P1IES |= BIT0;     //对应的管脚由高到低电平跳变使相应的标志置位
    P1IE |= BIT1;      //管脚P1.1使能中断
    P1IES |= BIT1;     //对应的管脚由高到低电平跳变使相应的标志置位
    return;
}
```

该部分代码为定时器B的初始化代码。

```
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1s

    TBCTL |= MC0;               //增记数模式
}
```

由上面的程序可以看出，通过设置TBCTL可以设置定时器B的时钟源选择，设置TBCCR0来确定时间间隔，用户可以根据自己的需要调整相应寄存器的值以满足自己的要求。

中断处理部分：该部分主要是定时器B中断和端口中断处理两个部分。定时器B中断主要负责设置行线的电平，使系统处于按键输入状态。端口中断主要是负责判断那个列线上有按键输入，从而获得数据。下面给出该部分的程序代码。

该部分代码为定时器B中断服务程序。该部分代码主要完成将相应的行线设置为低电平，使系统处于按键输入状态。

```
interrupt [TIMERB0_VECTOR] void TimerB_ISR(void)
{
    //设置相应的行线为低电平
    switch(PORT_count)
```

```
{
    case 0:
    {
        //设置为高电平
        P1OUT |= BIT3;
        P1OUT |= BIT4;
        P1OUT |= BIT5;
        P1OUT |= BIT6;
        //设置为低电平
        P1OUT &= ~(BIT2);
        break;
    }
    case:1
    {
        //设置为高电平
        P1OUT |= BIT2;
        P1OUT |= BIT4;
        P1OUT |= BIT5;
        P1OUT |= BIT6;
        //设置为低电平
        P1OUT &= ~(BIT3);
        break;
    }
    case 2:
    {
        //设置为高电平
        P1OUT |= BIT3;
        P1OUT |= BIT2;
        P1OUT |= BIT5;
        P1OUT |= BIT6;
        //设置为低电平
        P1OUT &= ~(BIT4);
        break;
    }
    case 3:
    {
        //设置为高电平
        P1OUT |= BIT3;
        P1OUT |= BIT4;
        P1OUT |= BIT2;
        P1OUT |= BIT6;
        //设置为低电平
        P1OUT &= ~(BIT5);
        break;
    }
    case 4:
    {
```

```
            //设置为高电平
            P1OUT |= BIT3;
            P1OUT |= BIT4;
            P1OUT |= BIT5;
            P1OUT |= BIT2;
            //设置为低电平
            P1OUT &= ~(BIT6);
            break;
        }
        default:break;
    }
}
```

在上面的程序中，PORT_count 是所记录的按键按下的顺序号，比如当前按下的是 P1.6 这个按键。

该部分代码为端口中断服务程序模块。该模块主要判断列线上的低电平从而获得输入值，下面为程序代码。

```
//处理来自端口 1 的中断
interrupt [PORT1_VECTOR] void PORT_ISR(void)
{
    Delay_us(100);//消除延时抖动
    if(P1IFG & BIT0) //P1.0列线上有按键输入
    {
    P1IFG &= ~(BIT4);   //清除中断标志位
    Delay_ms(1);
    for(;;)
    {
        if((P1IFG & BIT4) == 0) break;
    }
    //获得输入值
    switch(PORT_count)
    {
        case 0:
        {
            PORT_INPUT = 9;
            PORT_count = 1;
            break;
        }
        case:1
        {
            PORT_INPUT = 7;
            PORT_count = 2;
            break;
        }
        case 2:
        {
```

```
                PORT_INPUT = 5;
                PORT_count = 3;
                break;
            }
            case 3:
            {
                PORT_INPUT = 3;
                PORT_count = 4;
                break;
            }
            case 4:
            {
                PORT_INPUT = 1;
                PORT_count = 0;
                break;
            }
            default:break;
        }
FLAG_PORT = 1;
    }

    if(P1IFG & BIT1)//P1.1列线上有按键输入
    {
    P1IFG &= ~(BIT5);   //清除中断标志位
    Delay_ms(1);
    for(;;)
    {
        if((P1IFG & BIT5) == 0) break;
    }
    //获得输入值
    switch(PORT_count)
    {
        case 0:
        {
            PORT_INPUT = 0;
            PORT_count = 1;
            break;
        }
        case:1
        {
            PORT_INPUT = 8;
            PORT_count = 2;
            break;
        }
        case 2:
        {
            PORT_INPUT = 6;
```

```
                PORT_count = 3;
                break;
            }
            case 3:
            {
                PORT_INPUT = 4;
                PORT_count = 4;
                break;
            }
            case 4:
            {
                PORT_INPUT = 2;
                PORT_count = 0;
                break;
            }
            default:break;
        }
        FLAG_PORT = 1;
    }
}
```

以上两个部分都应用了全局变量 FLAG_PORT、PORT_INPUT 和 PORT_count，通过 FLAG_PORT 通知主程序有新的数据输入；通过 PORT_INPUT 全局变量实现中断程序与主程序进行数据交互；通过 PORT_count 全局变量实现端口中断程序与定时器 B 中断程序进行数据交互，该变量主要用来记录当前按下的按键的编号值，比如当前按下的是 P1.3 这个按键。

下面的代码为上面程序用到的延时程序。

```
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;
}
void Delay_us(unsigned long nValue)//微秒为单位，8MHz为主时钟
{
    int nCount;
    int i;
    int j;
    nCount = 3;
    for(i = nValue;i > 0;i--)
```

```
    {
        for(j = nCount;j > 0;j--) ;
    }
    return;
}
```

### 3. 显示模块

该部分主要完成数据的显示功能。在硬件设计中，显示电路直接与单片机的数据 I/O 口进行连接。P4.0～P4.6 是用来显示数据，P2.1 是用来控制小数点的显示，P2.2、P2.3 和 P2.4 是用来控制数码管的选通状态，比如要在 DIS0 上显示，需要在 P2.2 管脚上给出高电平选通数码管进行显示。显示模块相对比较简单，只是简单的将数据显示在数码管上，该模块主要包括端口初始化和数据显示两个部分，下面根据具体的程序进行介绍。

端口初始化部分：该部分完成 P4.0～P4.6 端口、P2.1～P2.4 端口的初始化工作，设置成相应的输入输出方向。具体的代码如下：

```
void Init_DispPort(void)
{
    //将管脚在初始化的时候设置为输入方式
    P2DIR = 0;
    //将所有的管脚设置为一般I/O口
    P2SEL = 0;

    //将P2.1~P2.4设置为输出方向
    P2DIR |= BIT1;
    P2DIR |= BIT2;
    P2DIR |= BIT3;
    P2DIR |= BIT4;
    //将P4.0~P4.6设置为输出方向
    P4DIR |= BIT0;
    P4DIR |= BIT1;
    P4DIR |= BIT2;
    P4DIR |= BIT3;
    P4DIR |= BIT4;
    P4DIR |= BIT5;
    P4DIR |= BIT6;

    return;
}
```

数据显示部分：该部分主要是将数据显示到数码管上，在显示数据时需要选通不同的数码管。具体的程序如下：

```
void Display(char n1,char n2,char n3)
{
    //数据表
        static char
```

```
    nLed[10]={0x7b,0x42,0x37,0x67,0x4e,0x6d,0x7d,0x43,0x7f,0x6f};
    //选通数码管

    //显示第一个数据
    //选通数码管
    P2OUT |= BIT2;
    P4OUT = nLed[n1];
    //显示第二个数据
    //选通数码管
    P2OUT |= BIT3;
    P4OUT = nLed[n2];
    //显示小数点
    P2OUT |= BIT1;

    //显示后面的一个数据
    //选通数码管
    P2OUT |= BIT4;
    P4OUT = nLed[n3];

    return;
}
```

该程序做了一个数据表，这样要显示某一个数字，只需要根据数组的下标就可以取得数据，然后直接赋值给 P4OUT，从而实现显示。

**4. 报警模块**

报警处理模块相当简单，这里只是简单的在一个 I/O 口上送出数据来驱动缝鸣器，该模块包括初始化端口和数据产生两个部分，下面就各个部分给出具体的程序代码。

初始化部分：该部分将输出端口设置为输出方向。程序代码如下：

```
void Init_AlarmPort(void)
{
    //将P2.5设置为输出方向
    P2DIR |= BIT5;

    return;
}
```

数据产生部分：该部分主要是在输出端口产生数据，这里不是简单的一个高电平或者低电平，而是有一定频率的数据，因为只有是交流信号才可以让蜂鸣器发声。至于不同的周期信号可以得到不同的频率，可以根据信号处理的知识进行分析，这里就不进行讨论。下面给出代码。

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

```
    P2OUT |= BIT5 ;    //高电平
    for(i = 0; i < 200;i++)
        _NOP();

    P2OUT &= ~(BIT5); //低电平
    for(i = 0; i < 200;i++)
        _NOP();

    return;
}
```

在上面得程序中，_NOP()为 MSP430 提供的内联函数。上面的程序只是一个单音频率的数据，可以根据信号处理的知识修改上面的程序，产生出具有丰富频率的数据。

**5．主处理模块**

主处理模块主要是将各个模块进行协调处理和实现数据交互。主处理模块首先完成初始化工作，初始化后进入循环处理，在循环过程中主处理获得采集模块的数据，并将数据进行处理，根据处理后的结果来进行显示或者报警。由于报警的上限和下限需要设置，另外考虑到对数据的保存，因此主程序先检查门限是否在 FLASH 里面有，如果没有则进行等待设置数据，设置完后才进入下一步处理，也就是程序必须在有设置数据的情况下才能正常运行。下面给出主处理的流程图，如图 4-13 所示。

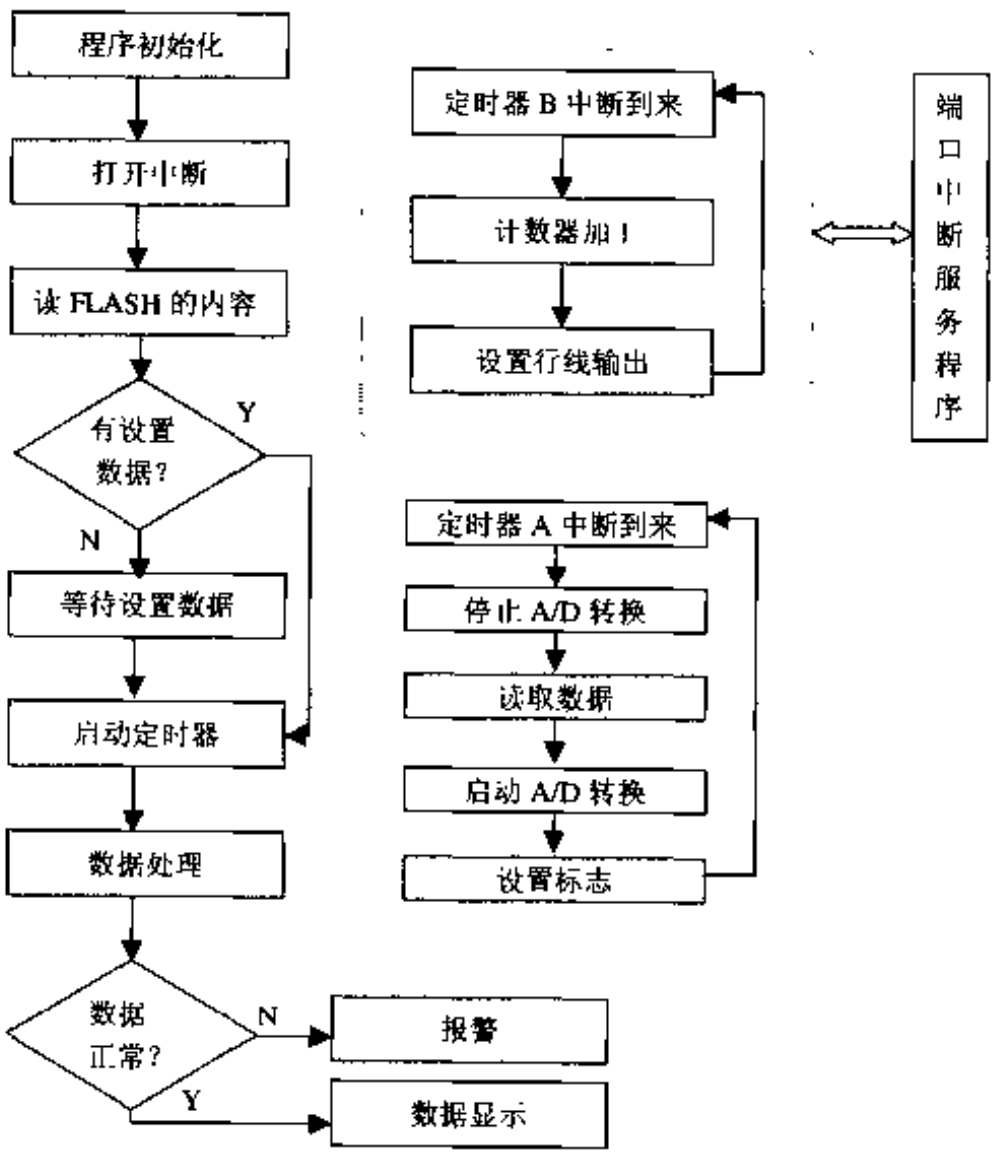



图 4-13　主处理流程图

考虑到需要对设置数据进行读写，这样需要对FLASH进行操作，下面给出FLASH操作的函数。

该函数的功能是将一个WORD类型的数据写入到FLASH里面。

```
void FLASH_ww(int *pData,int nValue)
{
    FCTL3 = 0xA500;       //LOCK = 0;
    FCTL1 = 0xA540;       //WRT = 1;

    *pData = nValue;
}
```

该函数的功能是将一个BYTE类型的数据写入到FLASH里面。

```
void FLASH_wb(char *pData,char nValue)
{
    FCTL3 = 0xA500;       //LOCK = 0;
    FCTL1 = 0xA540;       //WRT = 1;

    *pData = nValue;
}
```

该函数的功能是将FLASH里面的内容擦除掉。

```
void FLASH_clr(int *pData)
{
    FCTL1 = 0xA502;       //ERASE = 1;
    FCTL3 = 0xA500;       //LOCK = 0;

    *pData = 0;
}
```

根据上面的流程图给出简单的程序，下面的程序是简单化的处理，只是将数据乘以100当作整数处理，将得到的结果除以100获得整数部分，该程序忽略小数部分的处理，该程序也是假定上下门限是在0～100之间。以下为具体的程序。

```
#include <MSP430X14X.h>
#include "alarm.h"

#define AT_DATA1   0xef00
#define AT_DATA2   0xf000
#define AT_DATA3   0xf100
#define AT_DATA4   0xf200

//全局变量
int FLAG_PORT;
int PORT_INPUT;
int PORT_count;
int nADC_Flag;
```

```
int nADC_Count;
int ADC_BUF_Temp[10];
int ADC_BUF[10;
int UP1;//上门限整数部分
int UP2;//上门限小数部分
int DOWN1; //下门限整数部分
int DOWN2; //下门限小数部分
void main(void)
{
    int nTemp;
    int *pFlash;
    int nRes;
    int nCount;
    char chrTemp[6];
    float fTemp;

    int m_up1;
    int m_up2;
    int m_down1;
    int m_down2;
    char nTemp1;
    char nTemp2;
    char nTemp3;

    WDTCTL = WDTPW + WDTHOLD;     //关闭看门狗

    _DINT();            //关闭中断

    //初始化变量
    FLAG_PORT = 0;
    PORT_INPUT = 0;
    PORT_count = 0;
    nADC_Flag = 0;
    nADC_Count = 0;
    nRes = 1;
    //初始化
Init_CLK();
    Init_ADC();
    Init_TimerA();
    Init_INPUTPort();
    Init_TimerB();

    Init_DispPort();
    Init_AlarmPort();
```

```
_EINT();              //打开中断
//读取FLASH里面的内容
pFlash= (int*)(AT_DATA1);
UP1 = *pFlash;
if(UP1 == 0x00FF)
{
    nRes = 0;
}
else
{
    UP1 *= 8;
}
pFlash= (int*)(AT_DATA2);
UP2 = *pFlash;
if(UP2 == 0x00FF)
{
    nRes = 0;
}
else
{
    UP1 += UP2;
}
```

```
pFlash= (int*)(AT_DATA3);
DOWN1 = *pFlash;
it(DOWN1 == 0x00FF)
{
    nRes = 0;
}
else
{
    DOWN1 *= 8;
}
pFlash= (int*)(AT_DATA4);
DOWN2 = *pFlash;
if(DOWN2 == 0x00FF)
{
    nRes = 0;
}
else
{
    DOWN1 += DOWN2;
}

if(nRes == 0)
```

```
    {
        nCount = 0;
        //需要读取4个数据
        for(;;) //等待配置数据
        {
            if(FLAG_PORT == 1)
            {
switch(nCount)
                    {
                    case 0:
                    {
                        FLASH_ww(AT_DATA1,PORT_INPUT)
                        break;
                     }
                     case 1:
                     {
                         FLASH_ww(AT_DATA2,PORT_INPUT)
                         break;
                      }
                      case 2:
                      {
                         FLASH_ww(AT_DATA3,PORT_INPUT)
                         break;
                      }
                      case 3:
                      {
                         FLASH_ww(AT_DATA4,PORT_INPUT)
                         break;
                      }
                      default:break;
                }
                FLAG_PORT = 0;
                chrTemp[nCount] = PORT_INPUT;

nCount += 1;
if(nCount >= 4)
{
UP1 = chrTemp[0] * 8 + chrTemp[1];

    DOWN1 = chrTemp[2] * 8 + chrTemp[3];
    break;//数据配置完毕
}

            }
        }
```

```
    }

    for(;;)
    {
        if(nADC_Flag == 1)
        {
            nTemp = ADC_BUF_Temp[0];
            nADC_Flag = 0;
            //这里只是简单的化成整数处理，就是乘以100
            nTemp = 16434 - 29 * nTemp;

            nTemp = (int)((nTemp / 100);//将结果除以100
            nTemp1 = (int)(nTemp / 100);//最高位
            nTemp2 = (nTemp - nTemp1 * 100) / 10;//十位
            nTemp3 = (nTemp - nTemp2 * 10 - nTemp1 * 100);//个位

            //这里只是简单的比较整数部分
            if((nTemp > UP1) || (nTemp < DOWN1))
            {
                //报警
                Ring();
            }
            else
            {
                //显示
                Display(nTemp2,nTemp3,0);
            }

        }
    }
}
```

以上给出了系统的主处理程序，上面的程序只是起简单的演示作用。用户可以根据自己的需要进行修改，丰富系统的功能。

## 4.5 系统调试

前面已经介绍了整个系统的硬件设计和软件设计，该部分结合前面介绍的来讨论整个系统的联调。系统的调试包括硬件调试和软件调试，下面分别进行介绍。

**1．系统硬件调试**

系统的硬件调试很简单，先调试电源电路和复位电路，只要这两个部分能正常工作，再进行单片机的调试，如果单片机的晶振能起振的话，则整个硬件的单片机部分没有问题。关于硬件系统的其他部分则需要结合软件进行调试。

**2. 系统软件调试**

前面分别介绍了各个软件部分，为了能够进行整个系统的联调，需要软件和硬件调试结合起来，对于不同的硬件部分，应该调用不同的软件模块进行测试。经过联合调试，整个系统的软件和硬件能够正确运行。

## 4.6　实例总结

该系统通过一个集成的传感器实现温度采集报警系统具有设计简单、运行可靠等特点。通过软件测试和硬件测试证明该系统能够安全可靠地运行。由于单片机采用的是MSP430F149，该单片机的功耗非常低，另外加上采用的传感器具有很小的封装，这样该系统经过扩展改进可以用于手持设备。该系统的 CPU 采用 MSP430F149 也是考虑将来便于对硬件和软件进行升级扩展。比如，硬件系统中的单片机还有两个 UART 串口资源，因此该硬件系统可以进一步升级，实现可以与上位机进行通信的系统，也可以通过与 MODEM 连接来实现远程温度采集监控与报警系统。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自己功能，实现更为完整的系统。

# 第5章　MSP430F1XX实现的数据采集系统

随着信息化带动工业化进程的逐步深入，电子计算机信息技术的不断发展和完善，数据采集系统的应用越来越多。本章介绍一种采用 MSP430F149 实现的数据采集系统。该系统具有结构简单、功耗低等特点。

## 5.1　系统描述

随着电子计算机信息技术的不断发展和完善，采用单片机实现的数据采集系统的应用越来越多。随着工业化的进步，以前传统的采用人工进行数据记录登记已经远远不能满足现在工业化生产的要求，而采用单片机实现的数据采集系统具有自动化和无人值守等特点，使得它们在许多应用场合得到了广泛的应用。本章介绍一种采用 MSP430F149 实现的数据采集系统具有一定的通用性，它通过片内的 A/D 转换通道与外部的采集传感器进行连接。由于外部与传感器进行连接，获得是标准的电流信号，这样使采集系统具有很大的通用性，只要接不同的传感器就可以采集不同数据源的数据，做到系统与数据源的无关性。由于 MSP430F149 具有很多的 I/O 资源，因此采集系统也可以采集很多的数字开关量。采集系统采集得到数据后，通过 UART 串口将数据送到上位机上去，可以将数据交给上位机进行处理，从而降低采集系统的负担，增加系统处理的灵活性，并且也可以避免采集系统的海量存储器。图 5-1 为采集系统的原理框图。

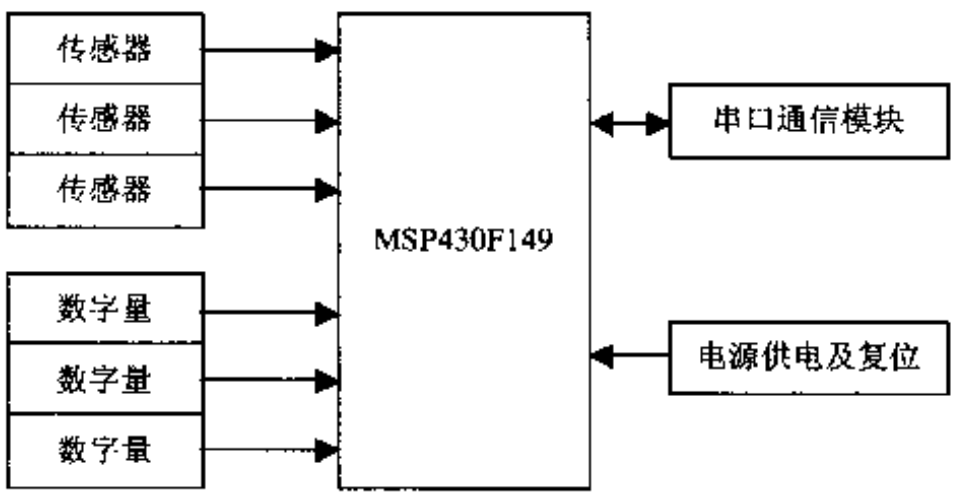



图 5-1　采集系统原理框图

由图 5-1 可以看出，整个采集系统由数字量采集模块、传感器模拟量采集模块、串口通信模块和电源供电及复位模块组成。传感器模块主要是采集模拟量，由于接的是标准的传感器，传感器送过来的是标准的电流信号，因此模拟量采集部分具有一定的通用性，只要接不同的传感器，就可以采集不同信号源的数据。数字量采集模块主要是采集一些开关量，这个部分很简单。串口通信模块主要是与上位机进行通信，由于单片机系统的电平和上位机的电平不同，因此接口的时候需要进行电平转换。电源及复位模块主要是为整个系统提供可靠的

电源，另外考虑到系统工作需要有复位功能，因此也为系统提供复位信号。下面一节将详细介绍各个功能模块的硬件设计。

## 5.2 系统硬件设计

通过上面的介绍，知道整个系统包括：模拟量采集模块、数字开关量采集模块、CPU处理模块、串口通信模块和电源及复位等模块，下面就具体的电路进行介绍。

**1．电源电路**

整个系统采用3.3V供电，考虑到硬件系统对电源要求具有稳压功能和纹波小等特点，另外也考虑到硬件系统的低功耗等特点，因此该硬件系统的电源部分采用TI公司的TPS76033芯片实现，该芯片能很好满足该硬件系统的要求，另外该芯片具有很小的封装，因此能有效节约PCB板的面积。电源电路具体如图5-2所示。

为了使输出电源的纹波小，在输出部分用了一个 2.2μF 和 0.1μF 的电容，另外在芯片的输入端也放置一个 0.1μF 的滤波电容，减小输入端受到的干扰。

**2．复位电路**

在单片机系统里，单片机需要复位电路，复位电路可以采用 R-C 复位电路，也可以采用复位芯片实现的复位电路，R-C 复位电路具有经济性，但可靠性不高，用复位芯片实现的复位电路具有很高的可靠性，因此为了保证复位电路的可靠性，该系统采用复位芯片实现的复位电路，该系统采用 MAX809 芯片。复位电路如图 5-3 所示。

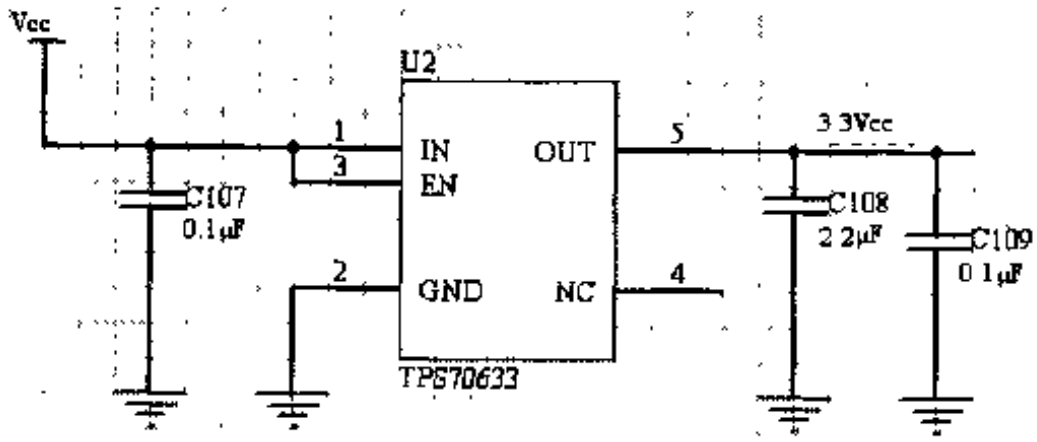



图 5-2　电源电路

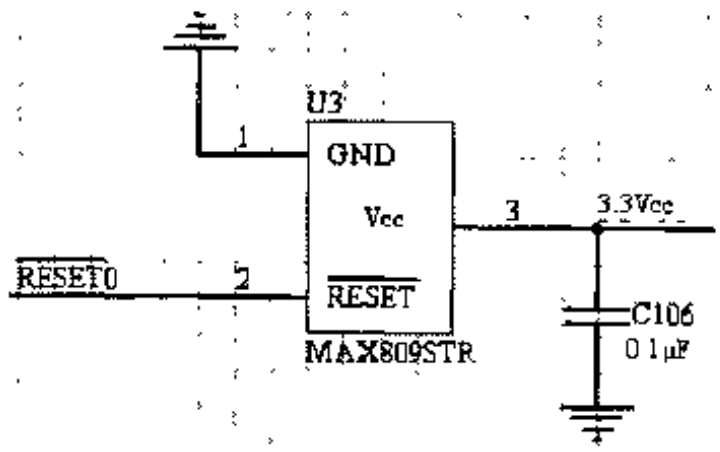



图 5-3　复位电路

为了减小电源的干扰，还需要在复位芯片的电源输入脚加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

**3．模拟量采集电路**

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。图 5-4 为模拟量采集具体的电路。

由图可以看出，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，增加采集电路的抗干扰问题。

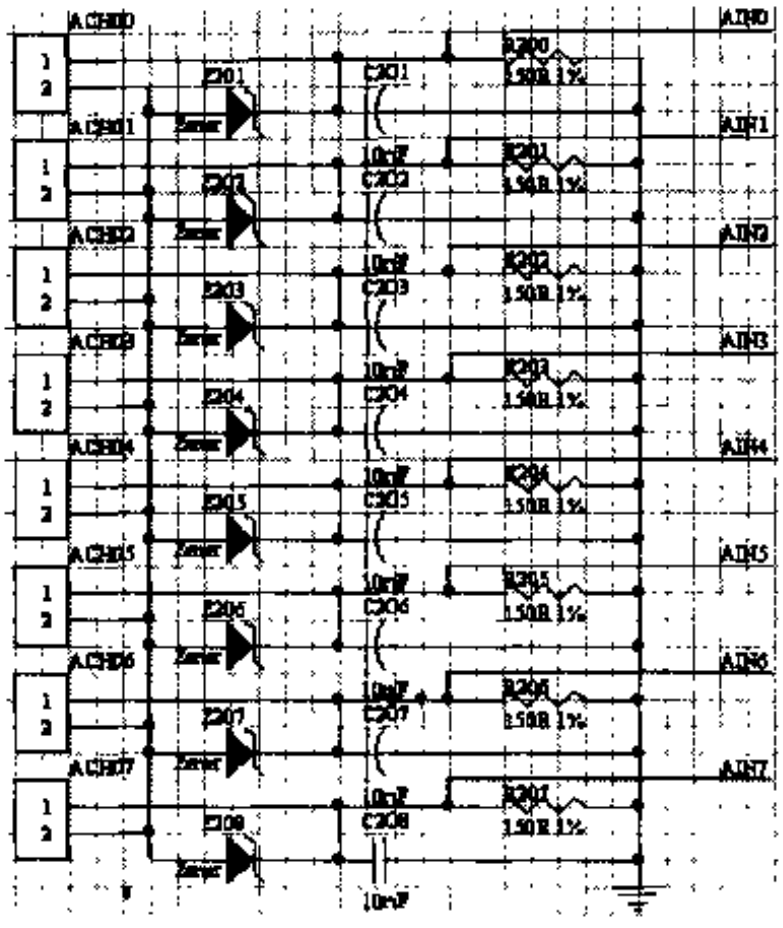


图 5-4　模拟量采集电路

**4．数字量采集电路**

在该系统中还提供数字量采集，数字量采集电路相对简单，图 5-5 为数字量采集具体的电路。

由图 5-5 可以看出，Q200 场效应管作为开关来启动数字量采集或者停止数据量采集。电路中采用二极管进行 ESD 保护。数字量的“1”或者“0”相当于电路的开和关，这样就可以采集在数字采集输入端产生的高低电平，从而实现数字量的采集任务。

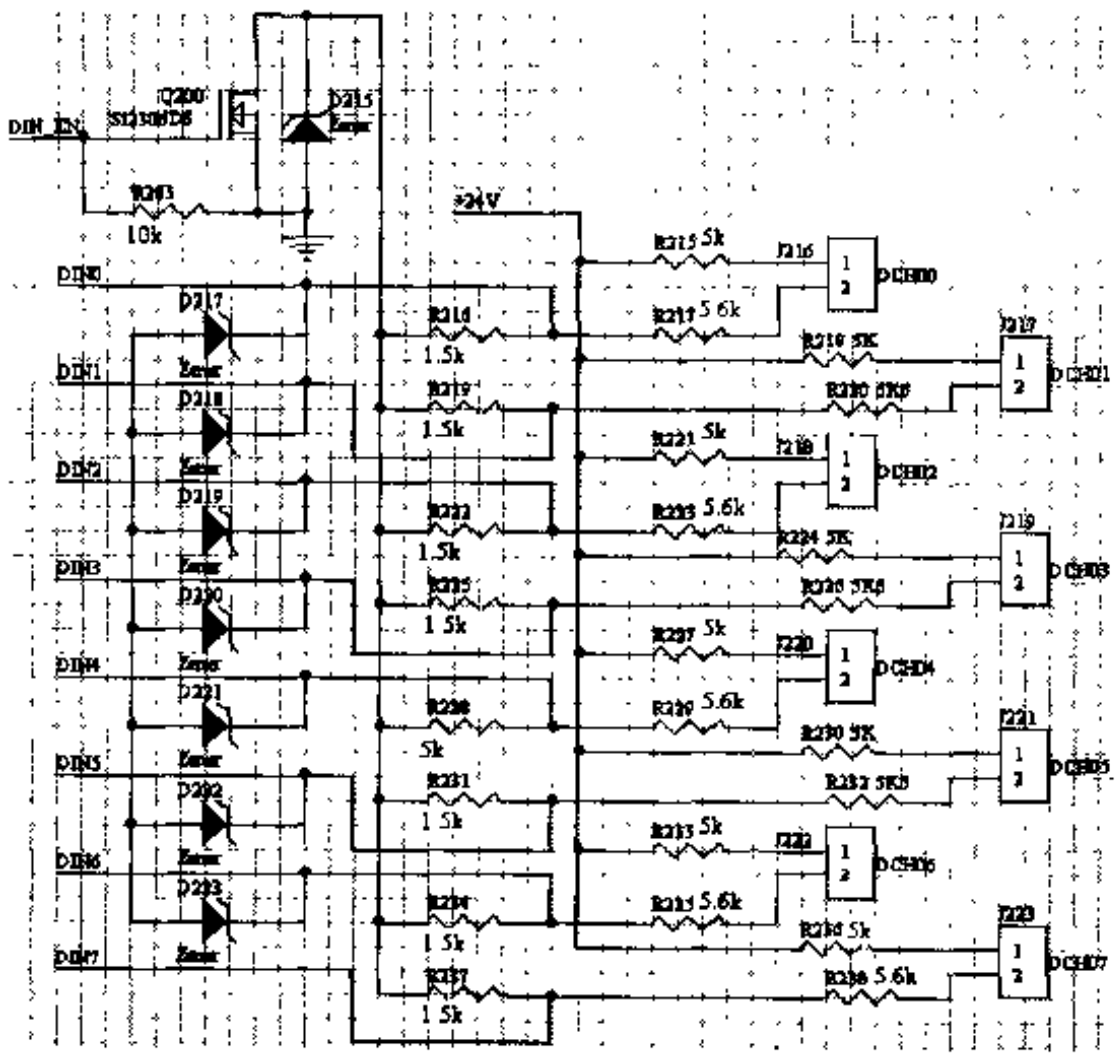


图 5-5　数字量采集电路

### 5．串口通信电路

该系统实现串口模块主要是与上位机进行通信，单片机系统将采到的数据送到上位机进行处理，从而减轻单片机系统的处理负担。由于单片机与上位机进行通信时接口电平不同，因此需要进行接口转换，这里采用 SP3220 芯片来完成接口电平的转换。SP3220 芯片具有功耗低、封装小等特点，在介绍具体电路之前先介绍一下 SP3220 芯片，SP3220 芯片具有以下特点：

- 宽电压供电。供电电压为：3.0V～5.5V。
- 上传速率可以达到 235Kb/s。
- 低功耗的电流为 1μA。
- 增强性 ESD 规范。

SP3220 芯片与一般的 RS232 芯片在使用上基本相同，下面给出该芯片的电路设计图，图 5-6 为串口通信的电路图。

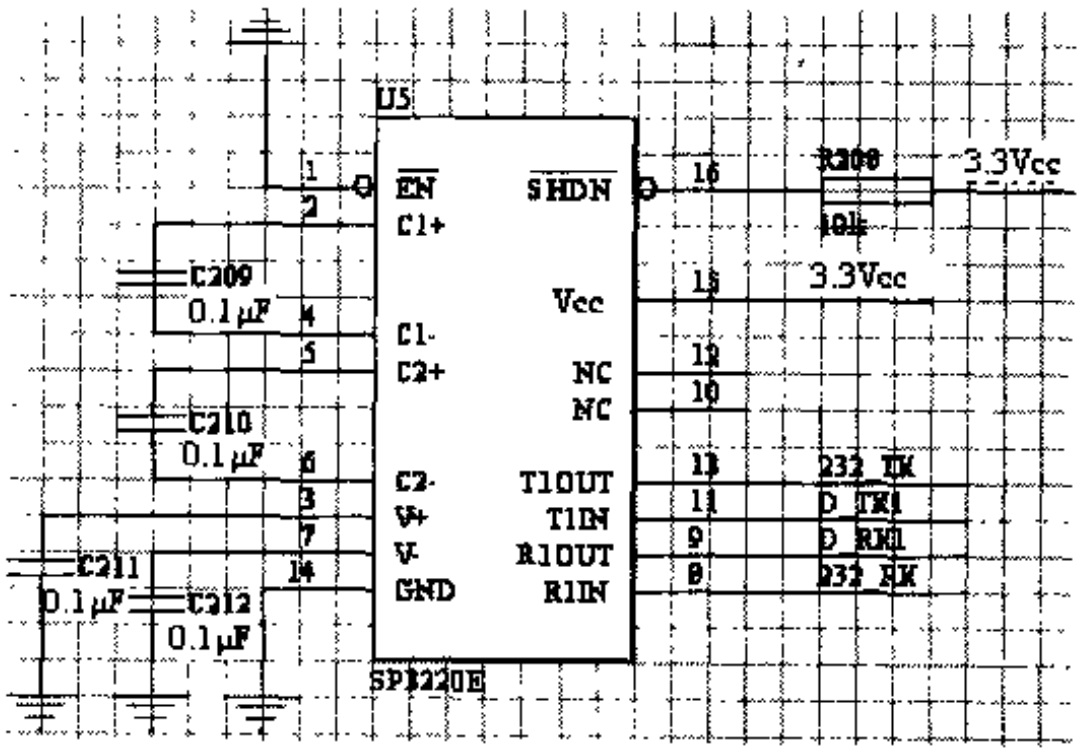



图 5-6　串口通信电路图

由图可以看出，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，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−分别放置 0.1μF 的电容实现充电作用，满足相应的充电泵的要求。管脚 T1OUT、TIN、RIOUT 和 RIN 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。考虑到减小电源的干扰，还需要在芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 6．CPU 处理模块

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和数字开关量进行采集，将采集得到的数据通过串口传给上位机，图 5-7 为单片机电路。

通过图 5-7 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，而且还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，模拟采集的参考电压采用的是片内提供的参考电压。单片机采用一般 I/O 口实现数字量采集电路的接口，并且通过一个一般 I/O 口来控制数字量采集。利用单片机片内的 UART 实现串口通信。在单片机的时钟设计上，考虑到低功耗要求，MSP430F149 单片机采用一个 32kHz 的时钟信号，考虑到串口通信的速率要求，MSP430F149 单片机采用一个 8MHz 的时钟信号。该系统的时钟部分都是采用晶体振荡器实现的。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。另外单片机还有模拟电源的输入端，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。

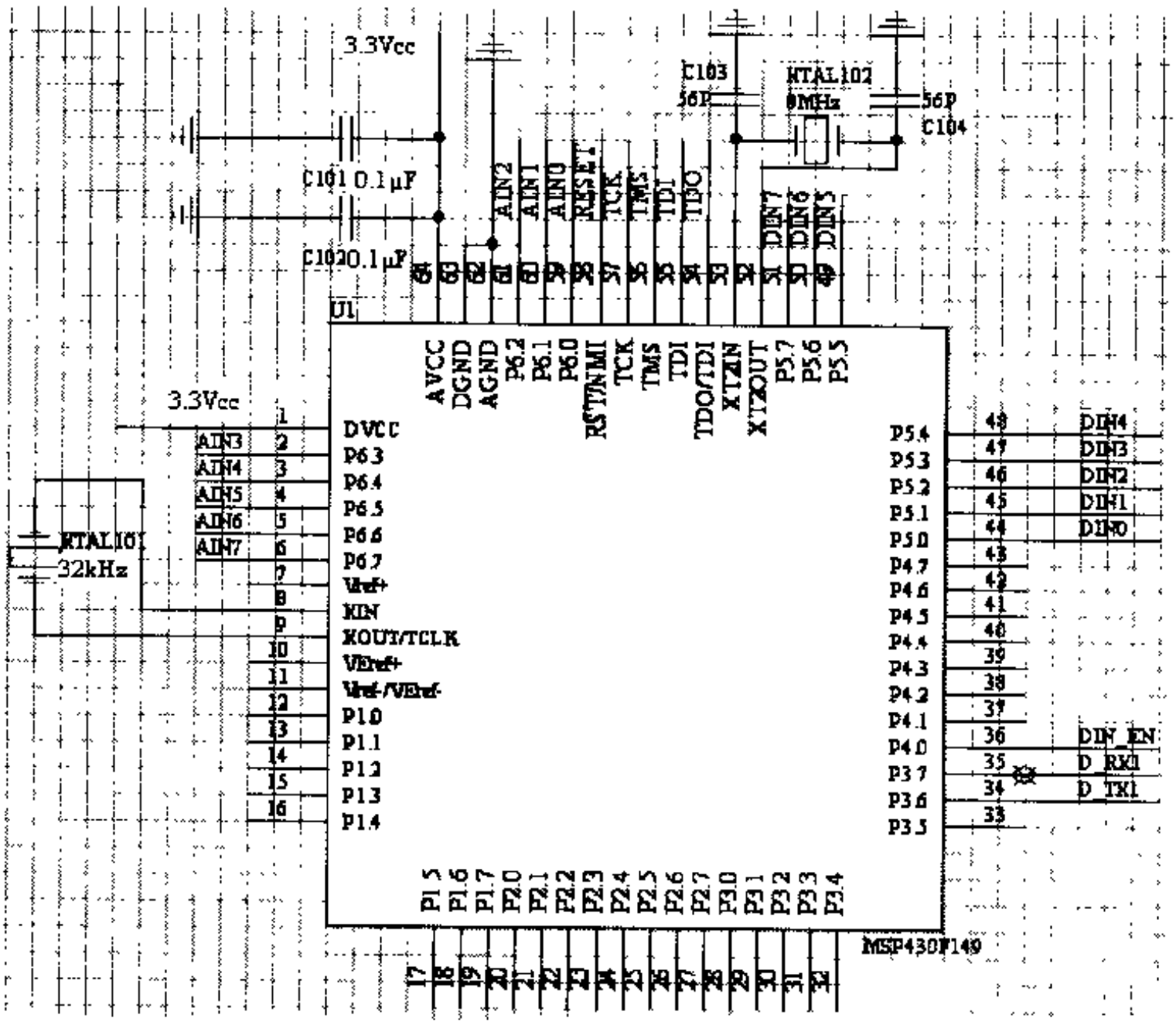



图 5-7　单片机电路

经过该节的介绍，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。

## 5.3　系统软件设计

经过前面对系统硬件的了解，这一节介绍系统的软件设计。系统的软件主要包括模拟量采集模块、数字量采集模块、串口通信模块和主处理模块。下面就具体的各个模块进行介绍。

### 1．模拟量采集模块

模拟量采集模块主要是单片机通过A/D通道采集来自传感器的信号，将信号进行处理。MSP430F149的A/D转换有几种模式，比如序列通道单次转换、序列通道多次转换，考虑到有8路采集，因此选用序列通道单次转换，当然也可以采用序列通道多次转换，关于转换模式的选择主要通过设置相应的A/D转换的寄存器来实现。数据采集的时间间隔通过定时器A来完成，就是在每次定时器A中断到来时读取A/D采集得到的数据，在读数据之前先停止A/D转换，在读取数据完毕后启动A/D转换，如果得到数据，则设置一个标志位通知主程序，告诉主程序已经得到新的数据。整个模块采用的是中断服务程序的结构完成。图5-8为该模块的程序流程图。

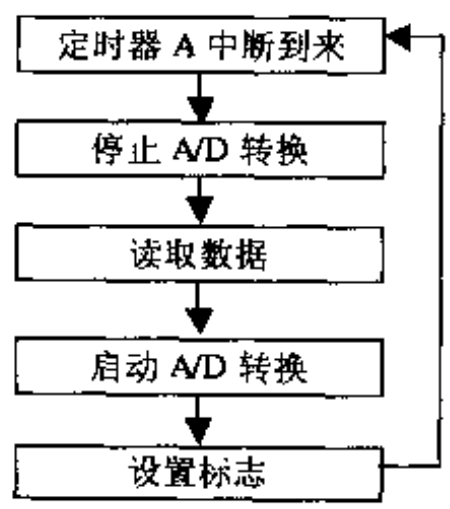



图5-8　模拟量采集模块流程图

该模块主要涉及到A/D转换和定时器A的操作。下面分析该模块的程序设计。

初始化部分：该部分主要完成A/D转换和定时器A的初始化功能。下面为初始化部分的代码程序（该部分程序为A/D初始化程序）。

```
void Init_ADC(void)
{
    P6SEL = 0X07;                          //设置P6.0为模拟输入通道
    ADC12CTL0 &= ~(ENC);                   //设置ENC为0，从而修改ADC12寄存器的值
    ADC12CTL1 |= CSTARTADD_0;              //转换的起始地址为：ADCMEM0

    ADC12MCTL0 = INCH_0;                   //设置参考电压分别为AVSS和AVCC，
                                           //输入通道为A0
    ADC12MCTL1 = INCH_1;                   //设置参考电压分别为AVSS和AVCC，
                                           //输入通道为A1
    ADC12MCTL2 = INCH_2;                   //设置参考电压分别为AVSS和AVCC，
                                           //输入通道为A2
    ADC12MCTL3 = INCH_3;                   //设置参考电压分别为AVSS和AVCC，
                                           //输入通道为A3
    ADC12MCTL4 = INCH_4;                   //设置参考电压分别为AVSS和AVCC，
                                           //输入通道为A4
    ADC12MCTL5 = INCH_5;                   //设置参考电压分别为AVSS和AVCC，
                                           //输入通道为A5
    ADC12MCTL6 = INCH_6                    //设置参考电压分别为AVSS和AVCC，
                                           //输入通道为A6
    ADC12MCTL7 = INCH_7 + EOS;             //设置参考电压分别为AVSS和AVCC，
                                           //输入通道为A6
    ADC12CTL0 |= ADC12ON;
    ADC12CTL0 |= MSC;

    ADC12CTL1 |= CONSEQ_1;                 //转换模式为：多通道、单次转换
```

```
    ADC12CTL1 |= ADC12SSEL_1;          //SMCLK
    ADC12CTL1 |= ADC12DIV_0;           //时钟分频为1
    ADC12CTL1 |= (SHP);                //采样脉冲由采用定时器产生

    ADC12CTL0 |= ENC;                  //使能ADC转换
    return;
}
```

通过以上的程序代码可以看出，只要适当设置 A/D 转换的寄存器，就能使 A/D 模块正确地工作。

```
void Init_TimerA(void)
{
    TACTL = TASSEL1 + TACLR;           //选择SMCLK，清除TAR
    TACTL += ID1;
    TACTL += ID0;                      //1/8SMCLK
    CCTL0 = CCIE;                      //CCR0中断允许
    CCR0 = 4000;                       //时间间隔为250Hz
    TACTL |= MC0;                      //增记数模式
    return;
}
```

通过上面的程序代码可以看出，通过设置 CCR0 就可以设置定时器中断的频率，从而实现采样时间间隔的控制。

定时器 A 处理和 A/D 转换部分：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实现数据的交互。这部分程序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 转换部分的代码程序。

```
//定时器中断，完成 ADC 转换
interrupt [TIMERA0_VECTOR] void TimerA_ISR(void)
{
    int results[8];
    int i;

    ADC12CTL0 &= ~ENC;                 //关闭转换

    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; //读出转换结果
```

```
    nADC_Count += 1;
    if(nADC_Count == 10)
    {
        nADC_Flag = 1;                          //设置标志
        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];
    }

    ADC12CTL0 |= ENC + ADC12SC;             //开启转换
}
```

以上程序使用了全局变量 nADC_Flag，通过 nADC_Flag 变量通知主程序有新的采集数据获得；全局变量 nADC_Count 用来记数处理；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。

**2．数字量采集模块**

数字量采集主要是单片机通过的一般 I/O 口与数字采集电路进行连接，单片机通过一般 I/O 口来简单的读取数字采集量的状态，判断是高电平还是低电平。MSP430F149 的一般 I/O 口只需要设置成输入方向就可以实现读取数字采集电路的状态。数字采集共有 8 路，直接一次读取输入管脚寄存器的值就可以了，这里的这种应用是与硬件的设计有关的。数据采集的间隔时间通过定时器 B 来完成，就是在每次定时器 B 中断到来时启动数字量采集，延时一段时间后读取内容，然后停止数字量采集，继续等待下一次中断的到来，这里数字量采集的控制是通过单片机的一个 I/O 管脚进行控制的。在获得数据后通过设置一个标志位通知主程序，告诉主程序已经得到新的数据。整个模块采用的是中断服务程序的结构完成。图 5-9 为该模块的程序流程图。

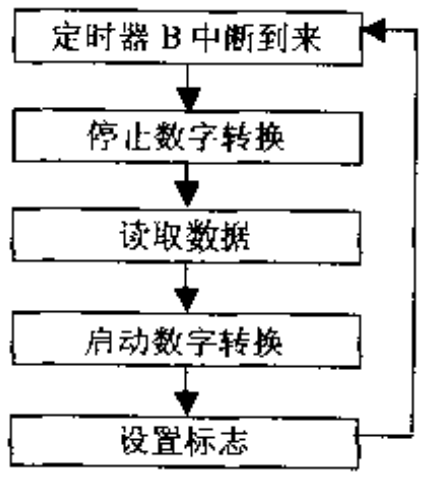



图 5-9　数字量采集流程图

该模块主要涉及到单片机一般 I/O 口操作和定时器 B 的操作。下面分析该模块的程序设计。

初始化部分：该部分主要完成一般 I/O 口和定时器 B 的初始化功能。下面为初始化部分的代码程序。

```
void Init_Port(void)
{
    //将P4口和P5口设置为输入方式
    P4DIR = 0;
    P5DIR = 0;

    //将P4口和P5口设置为一般I/O口
    P4SEL = 0;
    P5SEL = 0;

    //将P4.0设置为输出方向
    P4DIR |= BIT0;

    return;
}
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1s

    TBCTL |= MC0;                     //增记数模式
}
```

通过以上代码可以看出，只需要调整适当的寄存器的值就可以改变设置。定时器 B 选择 ACLK 作为时钟源，通过设置 TBCCR0 来确定中断的时间，设置 TBCCTL0 寄存器的 CCIE 位来允许中断。

定时器 B 处理和数字量采集部分：该部分主要完成 8 通道数字量的采集，并且通过定时器 B 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实现数据的交互。这部分程序采用中断服务程序实现，在定时器 B 里使能数字量采集，延时一段时间后读取数据，读取数据后关闭数字量采集，然后在等待下一次中断的到来。下面为定时器 B 处理和数字量采集的代码程序。

```
//////////////////////////////////////////////
//用于定时器B中断，完成数字量采集
interrupt [TIMERB0_VECTOR] void TimerB_ISR(void)
{
    int i;

    P4OUT |= BIT0;                    //打开采集
```

```
    Delay_us(100);                  //延迟一点时间
    D_BUF[nD_Count] = P5IN;         //读出采集结果

    nD_Count += 1;
    if(nD_Count == 10)
    {
        nD_Flag = 1;                //设置标志
        nD_Count = 0;
        //将数据倒向数据缓冲区
        for(i = 0;i < 10;i++) D_BUF_Temp[i] = D_BUF[i];
    }

    P4OUT &= ~(BIT0);               //关闭采集
}
```

在上面的程序中，通过读取 P5IN 寄存器的内容实现读取 8 路的数字量，当采集到一定的数据量后通过全局缓冲变量 D_BUF_Temp[]和全局标志变量 nD_Flag 实现与主程序的数据交互。其中延时程序的代码如下：

```
void Delay_us(unsigned long nValue)//微秒为单位，8MHz为主时钟
{
    int nCount;
    int i;
    int j;
    nCount = 3;
    for(i = nValue;i > 0;i--)
    {
        for(j = nCount;j > 0;j--);
    }
    return;
}
```

### 3．串口通信模块

串口通信模块主要是完成单片机与上位机的通信，从而将采集得到的数据送到上位机取进行处理。由于 MSP430F149 单片机具有片内的 UART，因此实现串口通信相当容易，只需要设置适当的寄存器就可以使串口工作起来。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。当接收到有数据时，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 5-10 所示。

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，如果有数据到则设置标志通知主程序。下面具体分析程序的代码。

初始化部分：该部分主要完成串口的初始化功能。下面为初始化部分的代码程序。

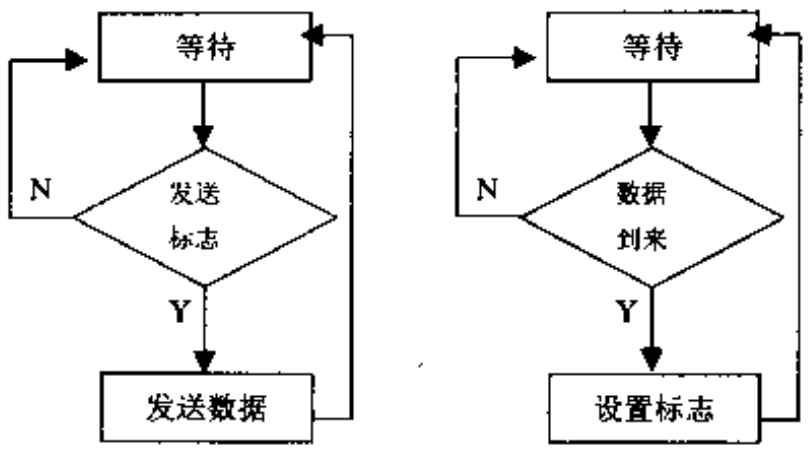



图 5-10　串口通信流程图

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

    U1TCTL = 0X00;             //将寄存器的内容清零
    U1TCTL += SSEL1;           //波特率发生器选择SMCLK

    UBR0_1 = 0X45;             //波特率为115200
    UBR1_1 = 0X00;
    UMCTL_1 = 0X00;

    ME2 |= UTXE1 + URXE1;      //使能UART1的TXD和RXD
    IE2 |= URXIE1;             //使能UART1的RX中断
    IE2 |= UTXIE1;             //使能UART1的TX中断

    P3SEL |= BIT6;             //设置P3.6为UART1的TXD
    P3SEL |= BIT7;             //设置P3.7为UART1的RXD

    P3DIR |= BIT6;             //P3.6为输出管脚
    return;
}
```

通过上面的代码可以看出，要使用串口资源，需要设置 P3SEL 寄存器的相应的位来使能 UART 功能；通过设置 ME2 和 IE2 寄存器的相应的位来打开中断；通过设置 U1CTL 的相应位来设置串口通信的格式，比如 8 位传输，也通过设置 U1TCTL 寄存器来实现时钟源的选择；设置寄存器 UBR0_1 和 UBR1_1 来设置串口通信的波特率，还可以通过设置 UMCTL_1 寄存器来实现波特率误差的调整。

串口中断主要是发送和接收数据，下面为程序代码。

```
//////////////////////////////////////////
//处理来自串口1的接收中断
interrupt [UART1RX_VECTOR] void UART1_RX_ISR(void)
{
```

```
        UART1_RX_BUF[nRX1_Len_temp] = RXBUF1;    //接收来自的数据

        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;
        }
    }
    ////////////////////////////////////////
    //处理来自串口 1 的发送中断
    interrupt [UART1TX_VECTOR] void UART1_TX_ISR(void)
    {
        if(nTX1_Len != 0)
        {
            nTX1_Flag = 0;              //表示缓冲区里的数据没有发送完

            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，很好利用了单片机的资源。

**4．主处理模块**

主处理模块主要是将各个模块进行协调处理和进行数据交互。主处理模块首先完成初始

化工作，初始化后进入循环处理，在循环过程中主处理获得采集的模拟数据和数字量数据，并将得到的数据发送到上位机。下位机也接收来自上位机的数据。整个程序基于中断服务结构，为了实现中断程序与主程序之间的数据交互，通过设置一些全局变量和全局的缓冲区来实现，具体的流程图如图 5-11 所示。

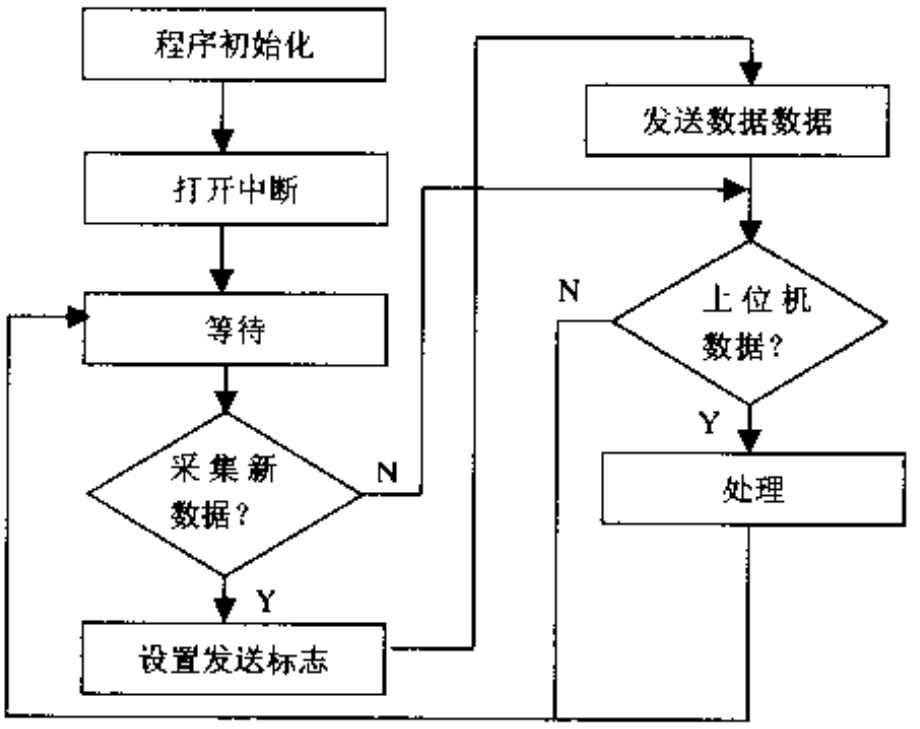



图 5-11　主处理流程图

通过流程图可以看出，主处理只负责简单的标志判断和设置标志，然后从指定的缓冲区读取数据或将数据放到相应的缓冲区，其他的处理由中断来处理，因此主程序主需要和中断程序进行数据交互。下面具体分析程序，由于各个模块在前面已经介绍，这里只是给出一个简单的应用例子，具体的程序还需要用户根据自己的需要进行修改。

程序的参考代码如下：

```
#include <MSP430X14X.h>
#include "init.h"

//定义全局变量
//模拟采集变量
int nADC_Flag;
int nADC_Count;
//数字采集变量
int nD_Count;
int nD_Flag;
//串口发送变量
int nSend_TX1;
static int nTX1_Len;
static char nTX1_Flag;
//串口接收变量
char nRX1_Len_temp;
char nRev_UART1; //串口 1 的接收标志
static char nRX1_Len;
```

```
//定义缓冲区
//数字采集缓冲区
char D_BUF_Temp[10];
char D_BUF[10];
//模拟采集缓冲区
int ADC_BUF_Temp0[10];
int ADC_BUF0[10];
int ADC_BUF_Temp1[10];
int ADC_BUF1[10];
int ADC_BUF_Temp2[10];
int ADC_BUF2[10];
int ADC_BUF_Temp3[10];
int ADC_BUF3[10];
int ADC_BUF_Temp4[10];
int ADC_BUF4[10];
int ADC_BUF_Temp5[10];
int ADC_BUF5[10];
int ADC_BUF_Temp6[10];
int ADC_BUF6[10];
int ADC_BUF_Temp7[10];
int ADC_BUF7[10];
//发送缓冲区
char UART1_TX_BUF[50];
//接收缓冲区
char UART1_RX_BUF[100];
char UART1_RX_TEMP[100]
void main(void)
{
    int i;

    WDTCTL = WDTPW + WDTHOLD; //关闭看门狗

    _DINT();                  //关闭中断
     //初始化
    Init_CLK();

     Init_ADC();
     Init_TimerA();
     Init_Port();
     Init_TimerB();
     //初始化变量
     nADC_Flag = 0;
     nADC_Count = 0;
     nD_Count = 0;
     nD_Flag = 0;
```

```
    nSend_TX1 = 0;
    nTX1_Len = 0;
    nTX1_Flag = 0;
    nRX1_Len_temp = 0;
    nRev_UART1 = 0;
    nRX1_Len = 0;

  _EINT();                          //打开中断
  //开始循环
  for(;;)
  {
      //处理数字量采集并发送
      if(nD_Flag == 1)//数字量数据
      {
          nD_Flag = 0;
          for(i = 0;i < 10;i++)
              UART1_TX_BUF[i] = D_BUF_Temp[i];
          nTX1_Len = 10;  //发送数据的长度
          IFG2 |= UTXIFG1;//设置中断标志
      }
      //处理模拟采集并发送
      if(nADC_Flag == 1)
  {
      nADC_Flag = 0;
      while(1)    //等待缓冲区里的数据发送完毕
      {
          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] = (char)((ADC_BUF_Temp0[i] >> 8) &
                                          0x00ff);
      }
      //设置帧结束标记
      UART1_TX_BUF[20] = 0xaa;
      UART1_TX_BUF[21] = 0xaa;
      nTX1_Len = 22;  //发送数据的长度
      IFG2 |= UTXIFG1;//设置中断标志

      while(1)    //等待缓冲区里的数据发送完毕
      {
          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] = (char)((ADC_BUF_Temp1[i] >> 8) &
                                0x00ff);
}
//设置帧结束标记
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
nTX1_Len = 22;  //发送数据的长度
IFG2 |= UTXIFG1;//设置中断标志

while(1)   //等待缓冲区里的数据发送完毕
{
    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] = (char)((ADC_BUF_Temp2[i] >> 8) &
                                0x00ff);
}
//设置帧结束标记
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
nTX1_Len = 22;  //发送数据的长度
IFG2 |= UTXIFG1;//设置中断标志

while(1)   //等待缓冲区里的数据发送完毕
{
    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] = (char)((ADC_BUF_Temp3[i] >> 8) &
                                0x00ff);
}
//设置帧结束标记
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
nTX1_Len = 22;  //发送数据的长度
```

```
IFG2 |= UTXIFG1;//设置中断标志

while(1)    //等待缓冲区里的数据发送完毕
{
    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] = (char)((ADC_BUF_Temp4[i] >> 8) &
                              0x00ff);
}
//设置帧结束标记
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
nTX1_Len = 22;  //发送数据的长度
IFG2 |= UTXIFG1;//设置中断标志

while(1)    //等待缓冲区里的数据发送完毕
{
    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] = (char)((ADC_BUF_Temp5[i] >> 8) &
                              0x00ff);
}
//设置帧结束标记
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
nTX1_Len = 22;  //发送数据的长度
IFG2 |= UTXIFG1;//设置中断标志

while(1)    //等待缓冲区里的数据发送完毕
{
    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] = (char)((ADC_BUF_Temp6[i] >> 8) &
                              0x00ff);
```

```
}
//设置帧结束标记
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
nTX1_Len = 22;  //发送数据的长度
IFG2 |= UTXIFG1;//设置中断标志

while(1)   //等待缓冲区里的数据发送完毕
{
    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_Temp7[i] & 0x00ff);
    UART1_TX_BUF[2 * i + 1] = (char)((ADC_BUF_Temp7[i] >> 8) &
                                0x00ff);
}
//设置帧结束标记
UART1_TX_BUF[20] = 0xaa;
UART1_TX_BUF[21] = 0xaa;
nTX1_Len = 22;  //发送数据的长度
    IFG2 |= UTXIFG1;//设置中断标志
}

//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，主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nD_Flag标志变量为1的时候，表示有新的数字开关量数据，从D_BUF_Temp[]全局缓冲区里取得采集得到的数字开关量数据。当主程序检测到nADC_Flag标志变量为1时，表示采集得到有新的模拟量数据，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，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。主程序也通过设置标志来通知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的时候，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。关于上面的程序主要是清楚了解相应的标志变量和全局缓冲区。

## 5.4　系统调试

前面已经介绍了整个系统的硬件设计和软件设计，该部分结合前面介绍的来讨论整个系统的联调。系统的调试包括硬件调试和软件调试，下面分别进行介绍。

**1．系统硬件调试**

系统的硬件调试很简单，先调试电源电路和复位电路，只要这两个部分能正常工作，再进行单片机的调试，如果单片机的晶振能起振的话，则整个硬件的单片机部分没有问题。关于硬件系统的其他部分则需要结合软件进行调试。

**2．系统软件调试**

前面分别介绍了各个软件部分，为了能够进行整个系统的联调，需要软件和硬件调试结合起来，对于不同的硬件部分，应该调用不同的软件模块进行测试。经过联合调试，整个系统的软件和硬件能够正确运行。

## 5.5　实例总结

该系统采用 MSP430F149 实现的一个数据采集系统具有设计简单、运行可靠等特点。通过软件测试和硬件测试证明该系统能够安全可靠的运行。本系统采用 MSP430F149 实现的数据采集系统具有一定的通用性，它通过片内的 A/D 转换通道与外部的采集传感器进行连接，由于外部与传感器进行连接，获得是标准的电流信号，这样使采集系统具有很大的通用性，只要接不同的传感器就可以采集不同数据源的数据，做到系统与数据源的无关性。由于 MSP430F149 具有很多的 I/O 资源，因此采集系统也可以采集很多的数字开关量。采集系统采集得到数据后，通过 UART 串口将数据送到上位机上去，可以将数据交给上位机进行处理，从而降低采集系统的负担，并且也可以避免采集系统的海量存储器。虽然本章介绍的系统相对简单，但用户完全可以在该基础上进行硬件和软件的扩展升级，实现自己更加丰富的功能，从而实现更为完整的系统。

# 第 6 章　日历系统的实现

在嵌入式或者单片机的应用中，时钟是一个非常重要的部分。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，实现一个实时时钟是非常有必要的。随着应用场合的增加，需要单片机系统具有实时时钟的需求越来越多。本章介绍一个基于 MSP430F149 实现的日历系统，该系统具有接口简单、运行可靠等特点。

## 6.1　系统描述

实时时钟（RTC）可应用于多种领域：从钟表到时间标记事件，甚至到产生事件。对于通信工程、电力自动化、工业控制等自动化程度高的领域，大多数情况下很多设备都处于无人值守的情况，都希望能把故障发生的时间和相关信息记录下来，以便进行具体分析。目前市面上有很多专用 RTC 器件，这些器件往往灵活性差，系统集成度低。而 MSP430F1XX 系列单片机具有低成本、低电流损耗、使用灵活简单及扩展性好等优点，使之成为专用 RTC 器件在某些特殊场合的理想替代品。

本系统采用了 TI 公司超低功耗 16 位微处理器：MSP430F149，该处理器具有极低功耗特性、极强的抗干扰能力和极高的性价比等特点。整个系统仅用两个普通电池（工作电压为 3V）就可以长期工作，无需其他电源，大大拓宽了应用范围。本系统主要由 MSP430F149 和时钟芯片 S-3505A 构成。S-3505A 具有功耗低的特点。S-3505A 芯片通过 $I^2C$ 接口与单片机进行连接，由于 MSP430F149 没有 $I^2C$ 接口，这样采用两个一般的 I/O 口来模拟 $I^2C$ 口实现。单片机与 S-3505A 连接后从 S-3505A 获得时钟数据，将得到的数据进行简单的显示，显示的部分由简单的 LED 实现。具体的系统框图如图 6-1 所示。

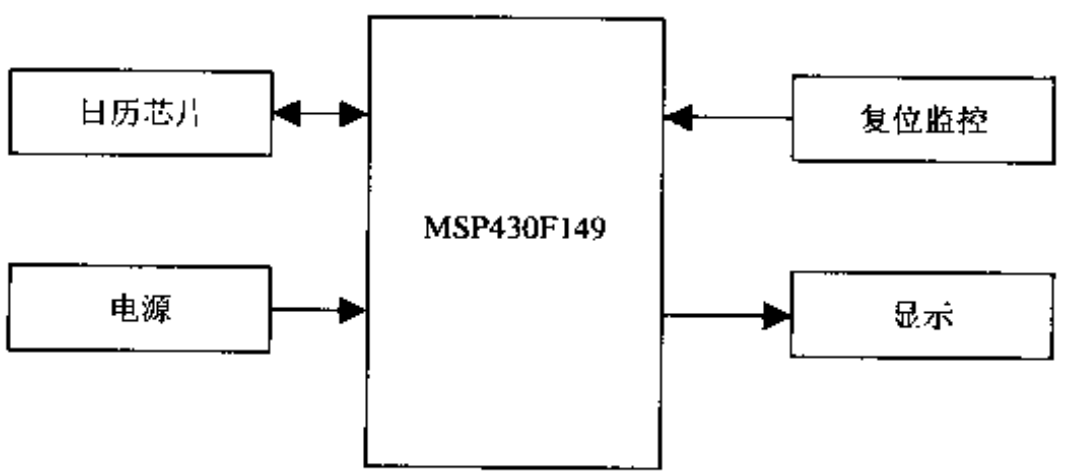



图 6-1　系统框图

由图 6-1 可以看出，整个系统硬件简单、接口方便。电源部分为整个系统提供稳定的 3V 电源。复位监控部分为系统在开机的时候提供复位信号，并监控系统工作的电压是否正常，当不正常的时候使系统复位。日历芯片由单片机进行设置，在设置后就为系统提供日历信息，

单片机可以随时从日历芯片里读取时间信息，将得到的时间信息显示出来。具体各个功能模块的硬件设计在下面一节进行详细介绍。

## 6.2 系统硬件设计

该系统的硬件系统相对很简单，主要有电源模块、复位监控模块、时钟模块、显示模块及单片机处理模块。下面就具体的电路进行介绍。

### 6.2.1 时钟模块介绍

在介绍具体的电路之前，先介绍一下时钟模块的设计。该模块采用的是 S-3505A 芯片来实现。该芯片具有以下特性。

- 低功耗：典型值 0.7 μA （$V_{DD}$=3.0V）。
- 宽工作电压：1.7V～5.5V。
- 年．月．日．星期．时．分．秒的 BCD 码输入 / 输出。
- $I^2C$ 总线接口。
- 自动日历到 2099 年（包括闰年自动换算功能）。
- 内置电源电压检测电路。
- 内置稳压电路。
- 内置上电 / 掉电检测电路。
- 内置报警中断（双系统）。
- 可设固定中断频率/事件。
- 内置 32kHz 石英晶体振荡电路（内部 Cd 外部 Cg）。
- 8 脚 DIP 和 8 脚 SSOP 封装。

为了增加对该芯片的认识，下面给出该芯片的方框图，如图 6-2 所示。由图 6-2 可以看出，芯片内部集成了振荡电路，这样只需要外部集成电容就可以实现振荡。通过对片内寄存器的操作就可以实现日历信息的获得，整个芯片采用 $I^2C$ 与其他芯片进行连接。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 6-3 所示。

由图 6-3 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。

- $\overline{INT1}$：报警中断 1 输出脚，根据中断寄存器与状态寄存器来设置其工作的模式，当定时时间到达时输出低电平或时钟信号。它可通过重写状态寄存器来禁止。
- XIN：晶振连接脚（32768Hz）。
- XOUT：晶体输出管脚。
- $V_{SS}$：电源地。
- $\overline{INT2}$：报警中断 2 输出脚，根据中断寄存器与状态寄存器来设置其工作模式，当定时时间到达时输出低电平或时钟信号。它可通过重写状态寄存器来禁止。
- SCL：串行时钟输出脚，由于在 SCL 上升/下降沿处理信号，要特别注意 SCL 信号的上升/下降升降时间，应严格遵守说明书。

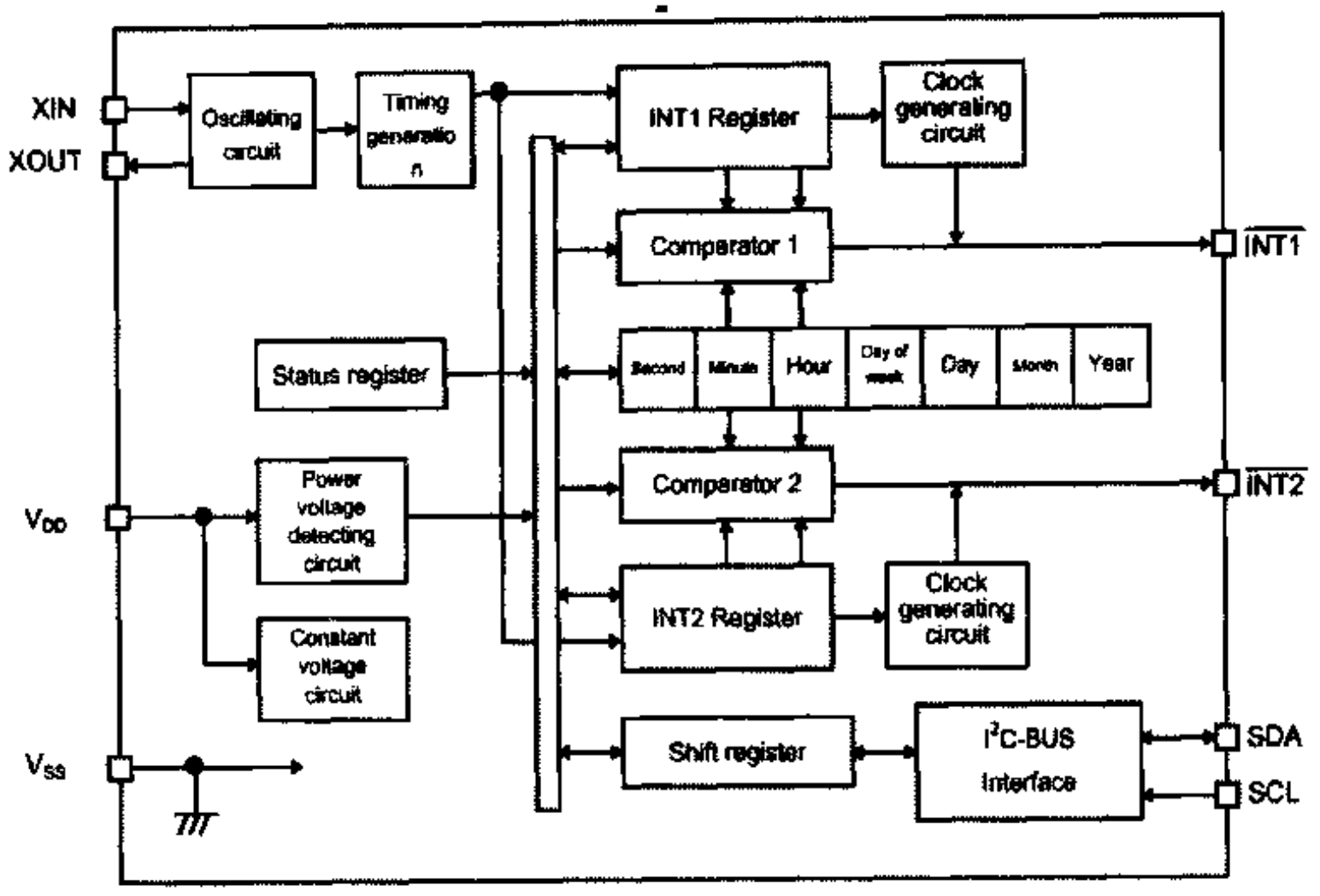



图 6-2　S-3505A 芯片方框图

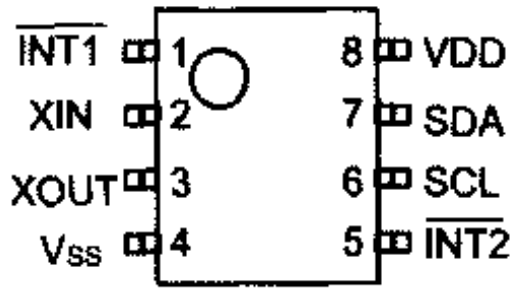



图 6-3　S-3505A 管脚图

- SDA：串行数据输入/输出脚，此管脚通常用 -电阻上拉至 $V_{DD}$，并与其他漏极开路或集电器开路输出的器件通过线或方式连接。
- $V_{DD}$：电源管脚。

经过对时钟芯片的介绍，对时钟模块有了基本的认识，下面介绍整个硬件系统的具体电路。

## 6.2.2　接口设计

整个系统的硬件相对简单，接口设计也非常容易，重要就是 $I^2C$ 接口的设计，但为了对整个硬件系统有全面的认识，该部分不仅仅分析接口电路，也介绍它的其他部分电路。主要电路有时钟模块、显示模块、CPU 处理模块和电源及复位模块。下面就具体的电路进行介绍。

### 1．电源电路

整个系统采用3.0V供电，考虑到硬件系统对电源要求具有稳压功能和纹波小等特点，另外也考虑到硬件系统的低功耗等特点，因此该硬件系统的电源部分采用TI公司的TPS76030芯片实现，该芯片能很好满足该硬件系统的要求，另外该芯片具有很小的封装，因此能有效节约PCB板的面积。电源电路具体如图6-4所示。

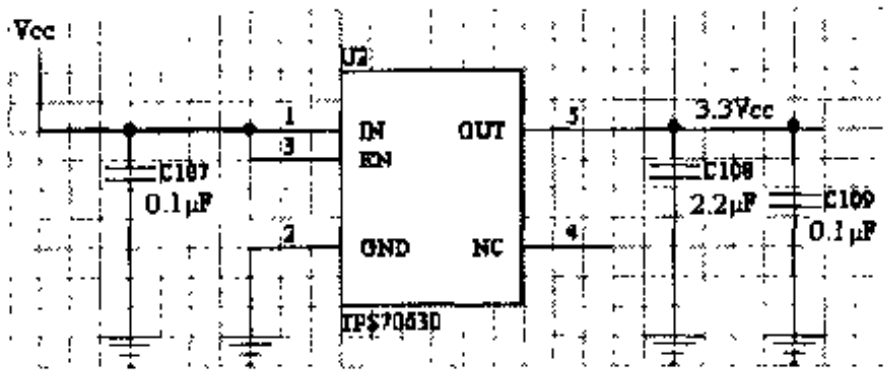



图 6-4　电源电路

为了使输出电源的纹波小，在输出部分用了一个 2.2μF 和 0.1μF 的电容，另外在芯片的输入端也放置一个 0.1μF 的滤波电容，减小输入端受到的干扰。

**2．复位电路**

在单片机系统里，单片机需要复位电路，复位电路可以采用 R-C 复位电路，也可以采用复位芯片实现的复位电路，R-C 复位电路具有经济性，但可靠性不高，用复位芯片实现的复位电路具有很高的可靠性，因此为了保证复位电路的可靠性，该系统采用复位芯片实现的复位电路，该系统采用 MAX809 芯片。复位电路如图 6-5 所示。

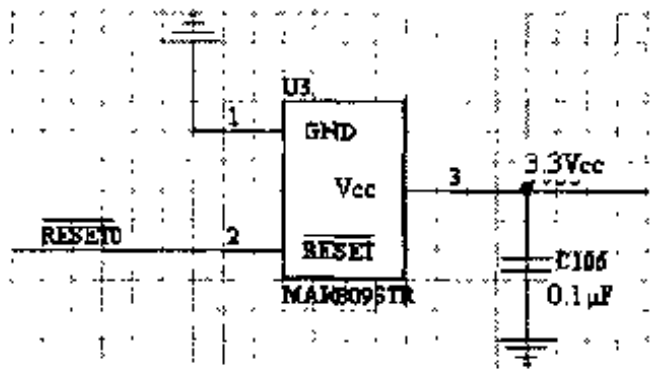



图 6-5　复位电路

为了减小电源的干扰，还需要在复位芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

**3．日历电路**

日历电路采用 S-3505A 实现，前面对 S-3505A 进行了详细的介绍，因此对该部分电路的设计应该不困难。该部分主要是完成日历信息的输出，同时接收单片机的设置信息。该部分主要是通过 $I^2C$ 实现与单片机的连接，具体的电路如图 6-6 所示。

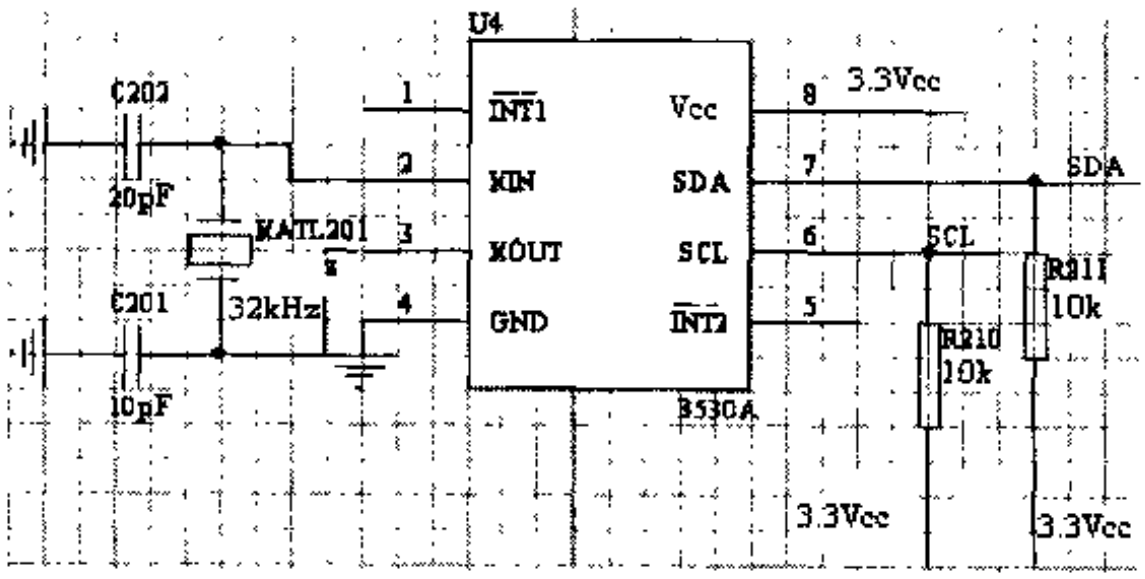



图 6-6　日历电路

由图 6-6 可以看出该电路设计简单。由 32kHz 晶体、20pF 电容和 10pF 电容构成了日历电路的振荡部分，这里电容的值必须严格一致才能保证时间的精度，因此 XIN 管脚必须接 20pF 的电容，XOUT 管脚必须接 10pF 的电容，这两个管脚的电容一定不能接其他值的电容来代替，晶体振荡器采用 32768Hz 的晶体。在设计 $I^2C$ 总线的时候必须将总线的两个管脚拉高。在系统中没有用到的两个报警管脚，只需要将这两个管脚悬空就可以了。为了减小电源的干扰，还需要在日历芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 4．显示电路

系统的显示电路采用的是简单的 LED 显示方式，这样的方式能满足该系统的要求，也可以减低系统的成本。图 6-7 为该系统的显示电路。

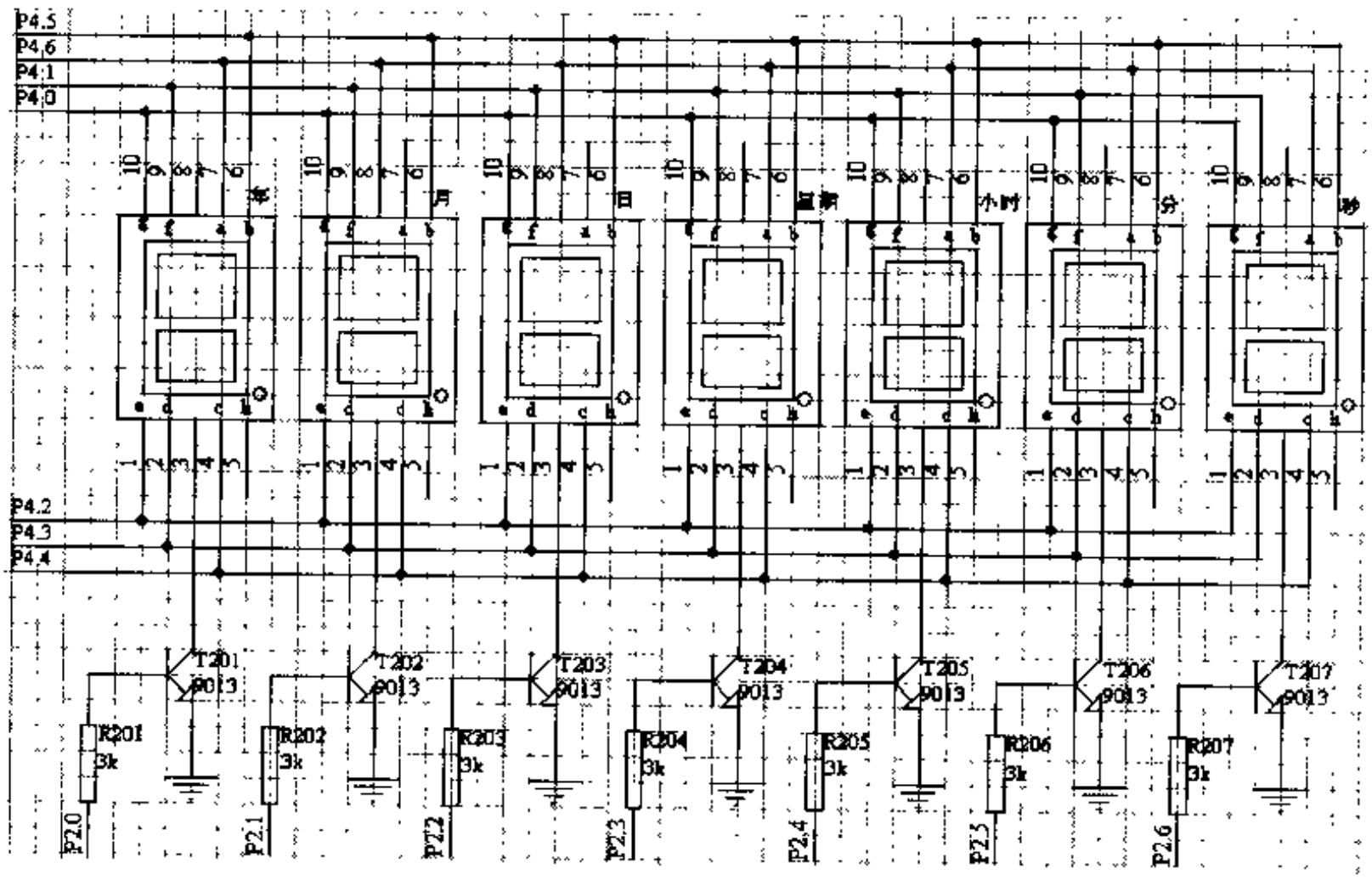



图 6-7　显示电路

通过图 6-7 可以看出，该显示电路直接与单片机的数据 I/O 口进行连接，由于 MSP430F149 具有丰富的 I/O 口资源，这样采用并行的接口方式非常容易，减小了系统设计的复杂度，也可以增加系统的可靠性。P4.0～P4.6 是用来显示数据，P2.0、P2.1、P2.2、P2.3、P2.4、P2.5 和 P2.6 是用来控制数码管的选通状态，比如要在“星期”上显示，则要在 P2.0 管脚上给出高电平，通过三极管选通数码管进行显示。

### 5．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，从而获得日历信息，将获得的日历信息进行显示。图 6-8 为单片机电路。

通过图 6-8 可以看出，单片机的接口电路非常简单，分别采用单片机的一般 I/O 口实现与其他电路的接口，在单片机的时钟设计上与其他单片机有一定的区别，MSP430F149 单片

机采用两个时钟输入，一个 32kHz 的时钟信号，一个 8MHz 的时钟信号。该系统的时钟部分都是采用晶体振荡器实现的。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。另外单片机还有模拟电源的输入端，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。由于 MSP430F149 单片机没有 $I^2C$ 接口，在设计时采用一般 I/O 口 P1.1 和 P1.2 分别作为 $I^2C$ 总线的 SCL 和 SDA 线，采用软件来模拟 $I^2C$ 总线，从而实现与日历芯片进行接口。

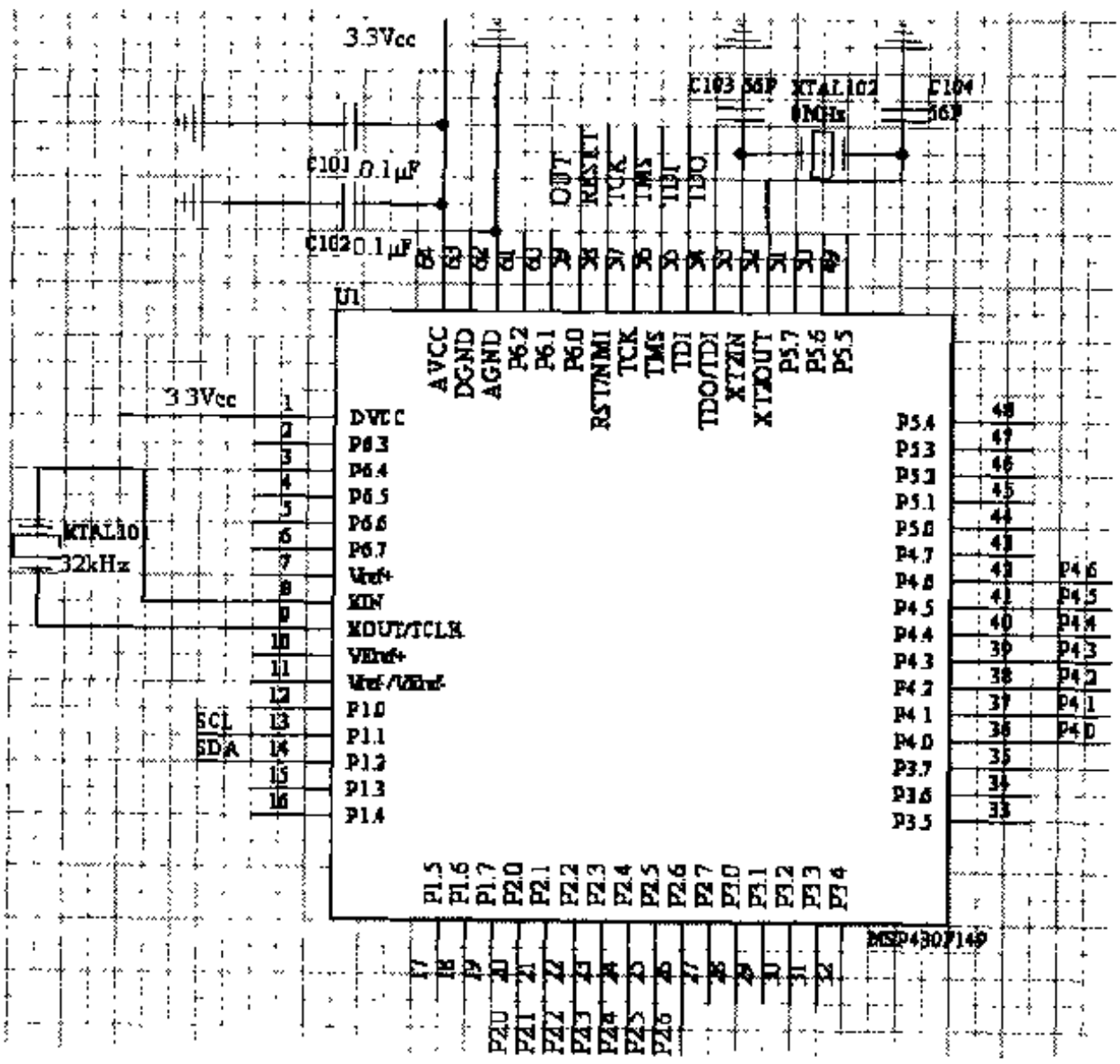



图 6-8　单片机电路

经过该节的介绍，对系统的硬件系统有了清楚的认识，下一节介绍系统的软件设计。

## 6.3　系统软件设计

经过对系统硬件的了解，这一节介绍系统的软件设计。系统的软件主要包括 $I^2C$ 模块、显示模块和主处理模块。下面就具体的各个模块将进行介绍，在具体介绍各个模块之前先介绍一下 $I^2C$ 协议。

### 6.3.1　$I^2C$ 协议介绍

在现代电子系统中，有为数众多的芯片需要进行相互之间以及与外界进行通信。为了提高硬件的效率和简化电路的设计，PHILIPS 开发了一种用于内部芯片控制的简单的双向两线串行总线 $I^2C$。$I^2C$ 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（NMOS、CMOS、双极性）。采用两线：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。每个器件都有一个唯一的识别地址，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唯一的地址和一直存在的简单的主机/从机关系软件设置地址，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100Kb/s，快速模式下可达400Kb/s，高速模式下可达3.4Mb/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（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 VDD 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保持稳定。数据线的高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。下面结合 $I^2C$ 协议对 S-3530A 的工作方式进行描述。

### 1．起始条件和停止条件

在 $I^2C$ 总线中唯一出现的是被定义为起始（S）和停止（P）条件，如图 6-9 所示。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，所有操作均必须由开始条件开始。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。

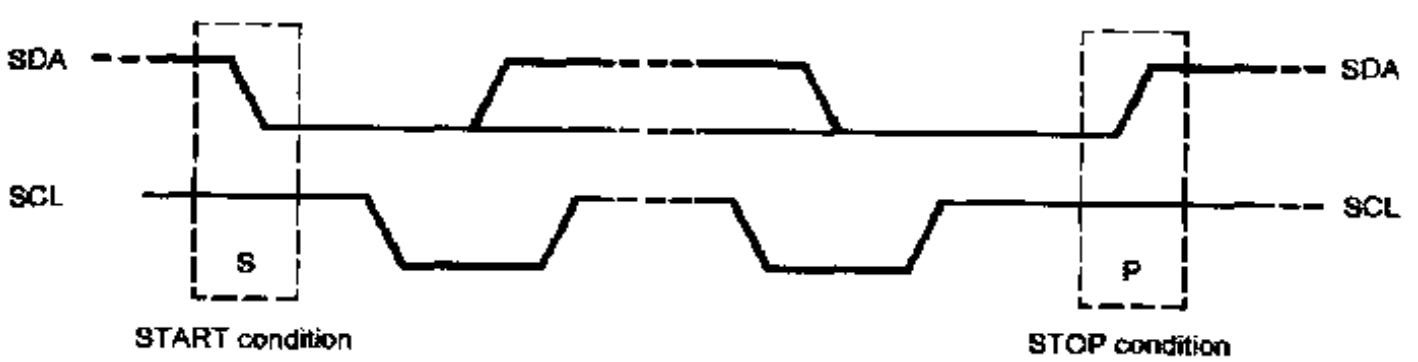



图 6-9　$I^2C$ 的开始条件和停止条件

### 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，数据传输的顺序是首先传输的是数据的最高位 MSB。如果从机要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。$I^2C$ 的数据传输如图 6-10 所示。

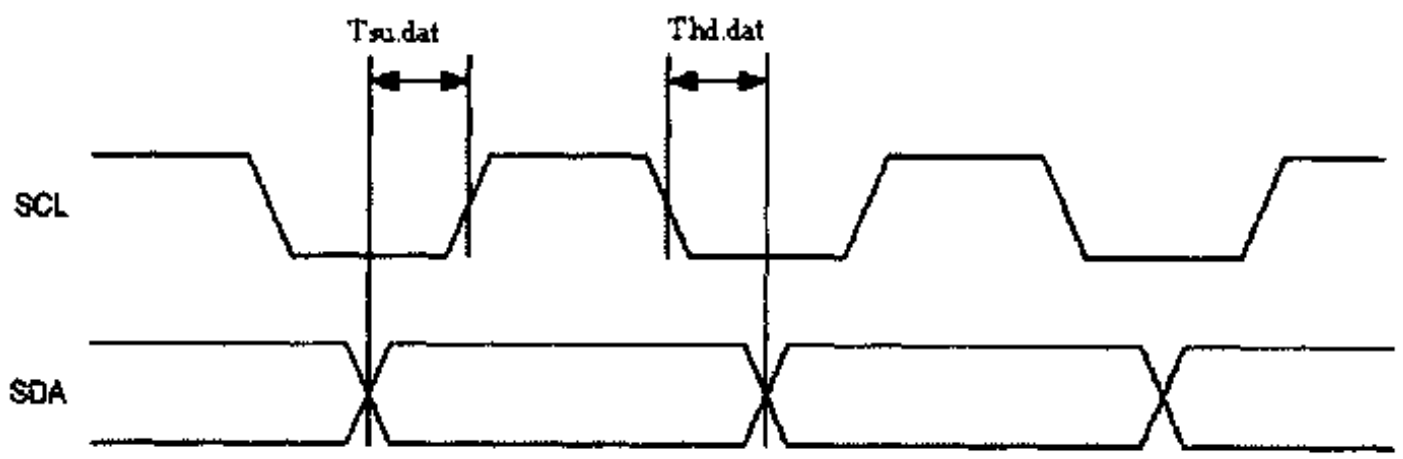



图 6-10　数据传输

### 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，在确认的时钟脉冲期间发送器释放SDA线（高）。在确认的时钟脉冲期间，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，如图6-11所示。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不能接收或发送），从机必须使数据线保持高电平，主机然后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，主机产生一个停止或重复起始条件。如果传输中有主机接收器，它必须通过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或重复起始条件。

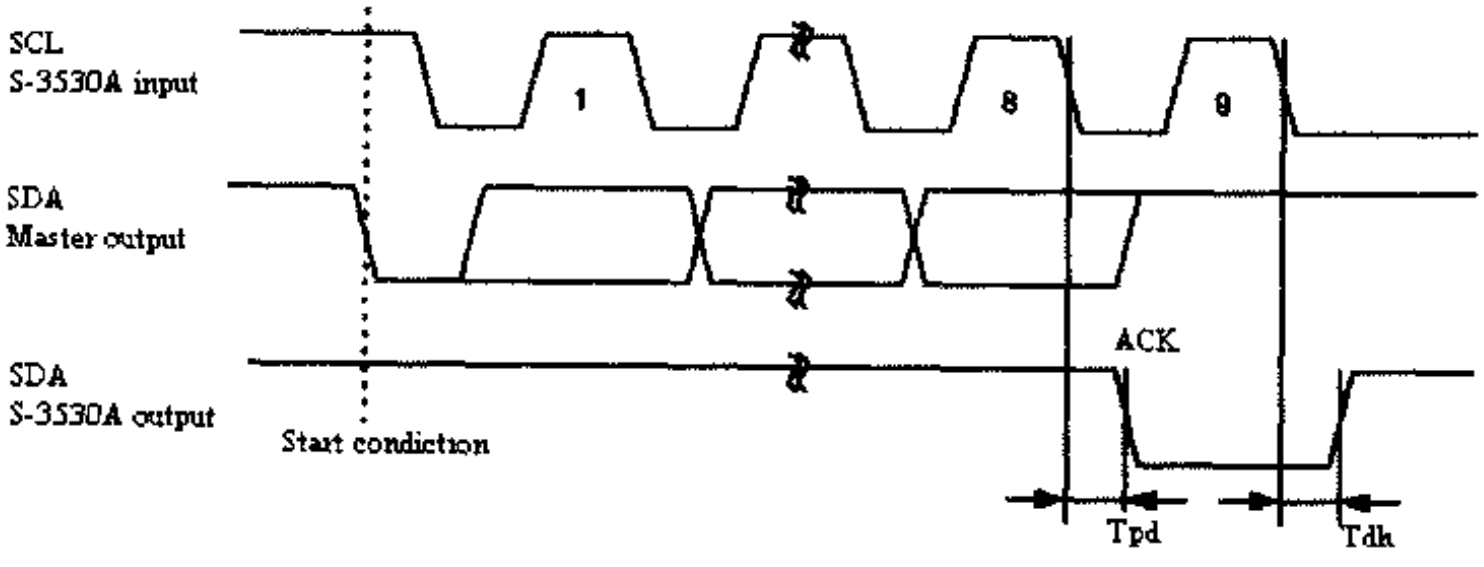



图 6-11　数据确认

前面经过对 $I^2C$ 协议的简单介绍，对 $I^2C$ 有了基本的概念，下面结合 $I^2C$ 协议介绍一下对 S-3530A 芯片的操作。对 S-3530A 芯片的操作主要包括器件寻址、读数据和写数据等操作。

**4．器件寻址**

CPU 发出开始条件给 S-3530A 以建立连接。CPU 通过 SDA 总线连续输出 4 位器件地址，3 位指令和 1 位读/写指令。其中高四位称“器件代码”，它代表器件地址，固定为“0110”。发送数据的格式如图 6-12 所示。

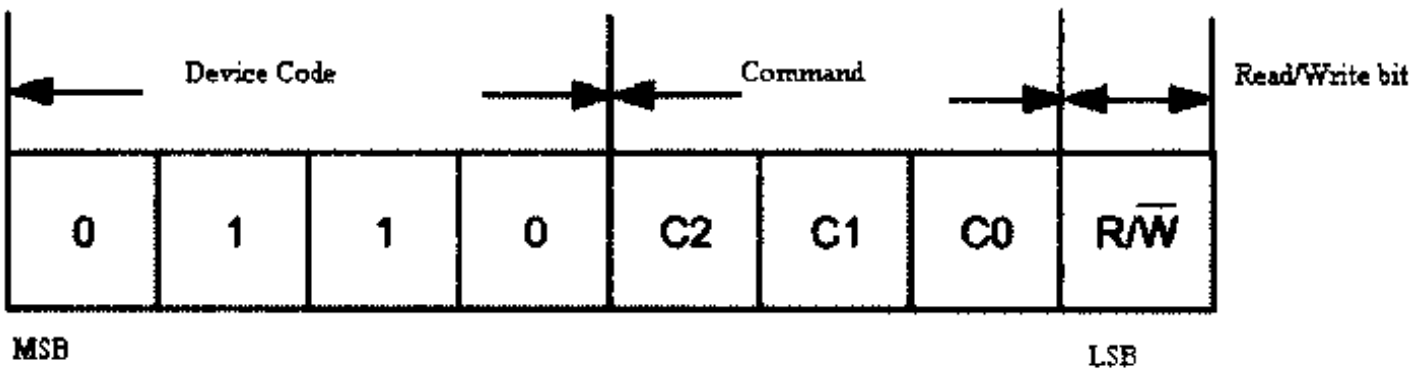



图 6-12　S-3530A 器件寻址

其中 R/$\overline{W}$ 表示读/写操作，当该位为 0 时表示向器件写数据，当该位为 1 时，从器件读取数据。C2、C1 和 C0 三个位表示命令，通过不同的命令来表示不同的操作。表 6-1 为 S-3530A 的操作命令。

**表 6-1　操作命令表**

| C2 | C1 | C0 | 操作 | ACK 数目 |
|---|---|---|---|---|
| 0 | 0 | 0 | 复位 00（年），01（月），01（天），0（星期）00（分），00（秒）（*1） | 1 |
| 0 | 0 | 1 | 状态寄存器存取 | 2 |
| 0 | 1 | 0 | 实时数据 1（从年数据开始）存取 | 8 |
| 0 | 1 | 1 | 实时数据 2（从小时数据开始）存取 | 4 |
| 1 | 0 | 0 | 频率事件设置 1（INT1 脚） | 3 |
| 1 | 0 | 1 | 频率事件 2（INT2 脚） | 3 |
| 1 | 1 | 0 | 测试模式开始（*2） | 1 |
| 1 | 1 | 1 | 测试模式结束（*2） | 1 |

表 6-1 列出了 S-3530A 的所有操作命令，通过这些操作命令就可以完成对 S-3530A 的不同操作。

**5．读数据**

当检验到开始条件后，S-3530A 接收器件代码和命令。当读/写位为“1”时，此时进入实时读取模式或状态寄存器读取模式。无论上述哪一种方式，数据均是从 LSB 依次输出。由表 6-1 可以看出，读数据主要包括读实时数据 1、读实时数据 2 和读取状态寄存器。下面给出 3 种读数据的时序图。

读实时数据 1：该命令为“010”，R/$\overline{W}$ 为“1”，读取时间数据，数据从年开始，图 6-13 为读取数据命令 1 的时序图。

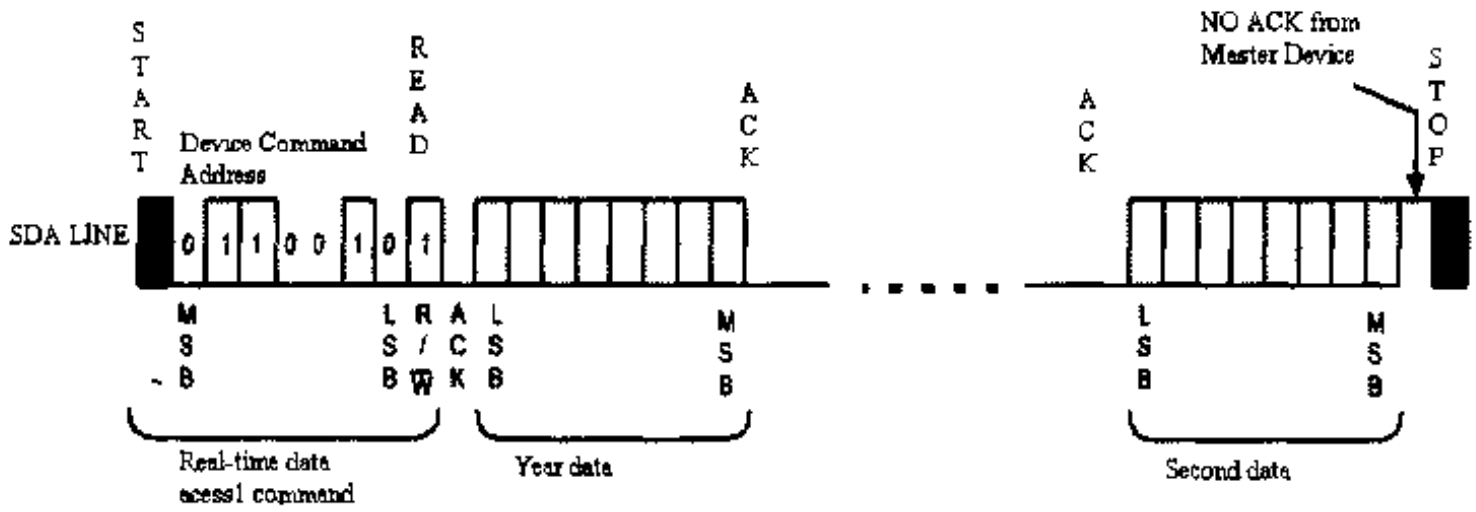



图 6-13　读取数据 1 的时序图

读实时数据 2：该命令为“011”，R/$\overline{W}$ 为“1”，读取时间数据，数据从小时开始，图 6-14 为读取数据命令 2 的时序图。

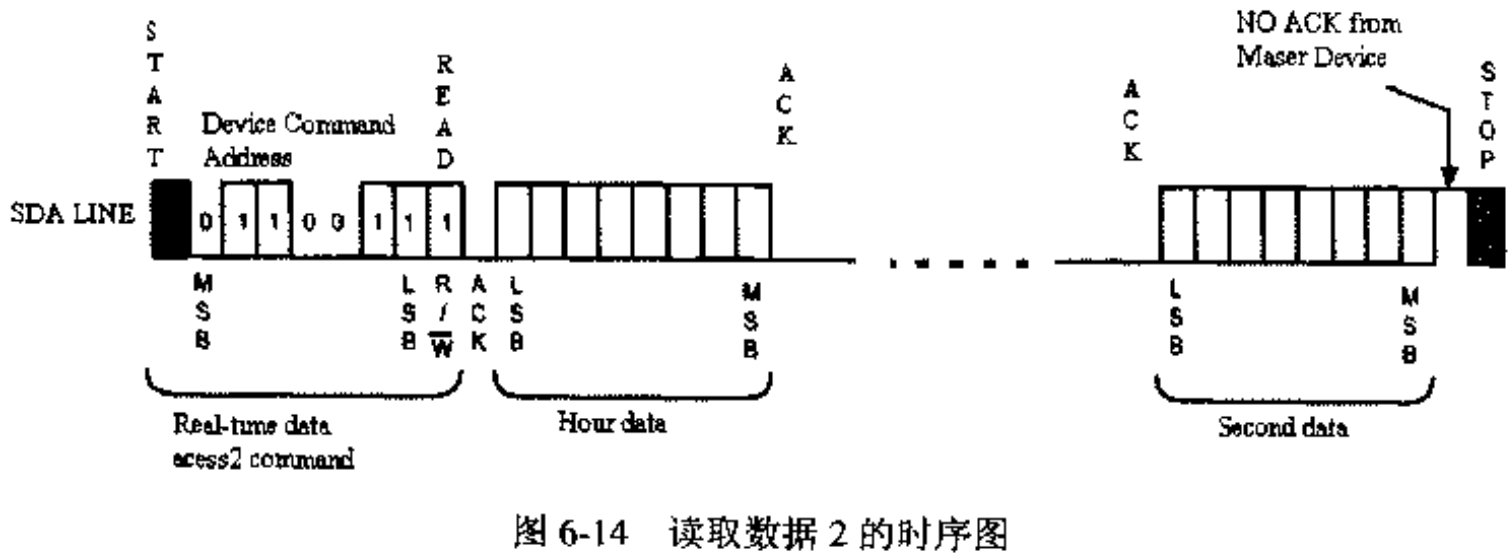



图 6-14　读取数据 2 的时序图

读取状态寄存器：该命令为“001”，R/$\overline{W}$ 为“1”，读取芯片的状态寄存器的内容，图 6-15 为读取状态寄存器命令的时序图。

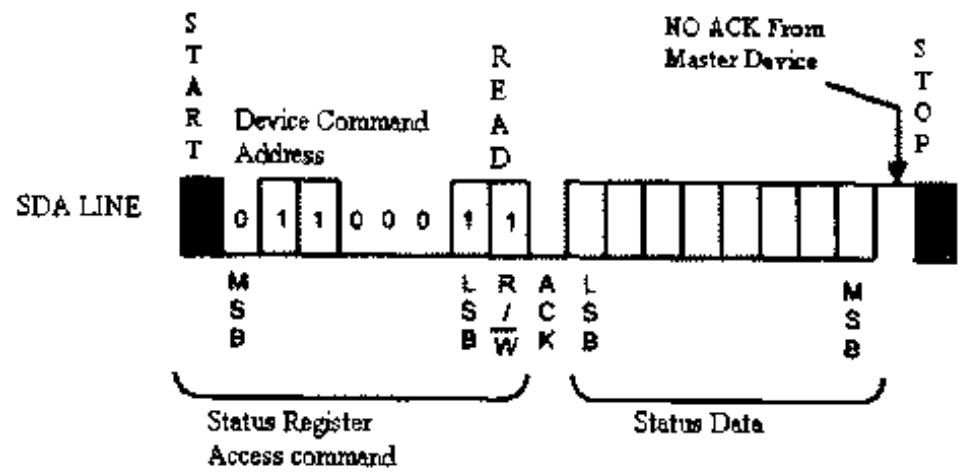



图 6-15　读取状态寄存器时序图

### 6．写数据

当检测到开始条件后，S-3530A 开始接收器件代码和命令。当读/写位为“0”时，此时进入实时数据写模式或其他寄存器写模式，数据必须按顺序从 LSB 位（在实时数据写模式或其他寄存器写模式下）开始依次输入。实时数据写入时，当 ACK 信号紧跟实时数据写命令时，日历与时间计数器的值将被重新设置，并将禁止任何更新操作。继接收完分钟数据，再输入秒数据，此时月末数据将被修正。当接收完秒数据，即发出 ACK 信号，从此开始计时。

写数据主要包括读实时数据 1、读实时数据 2 和读取状态寄存器。下面给出 3 种读数据的时序图。

写实时数据 1：该命令为“010”，R/$\overline{W}$ 为“0”，写时间数据，数据从年开始，图 6-16 为该写数据命令 1 的时序图。

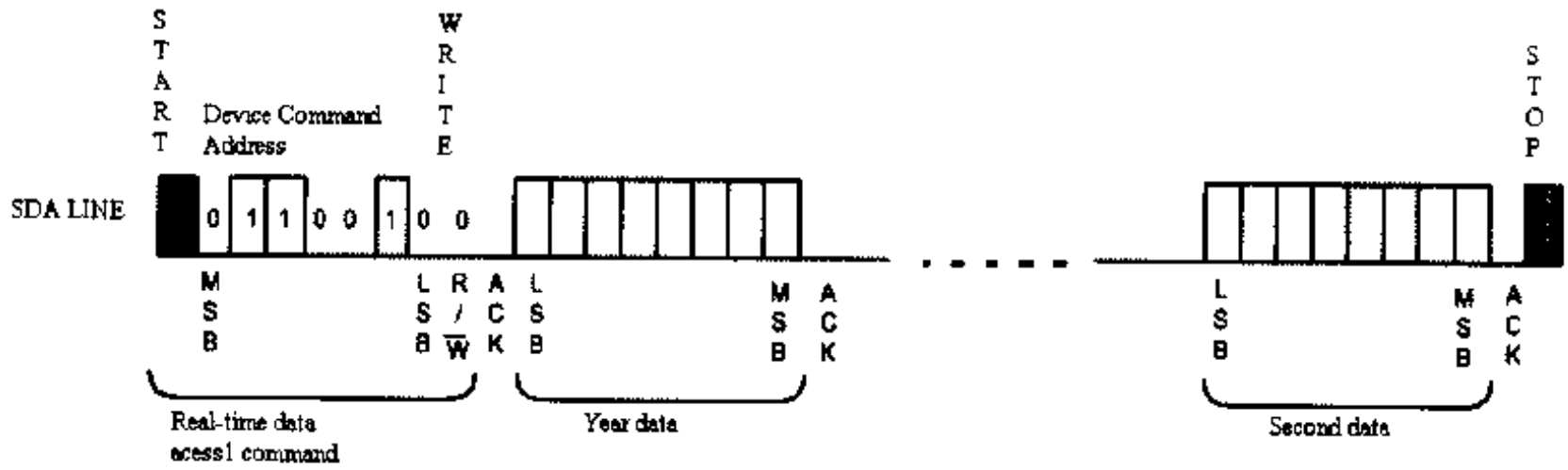



图 6-16　写数据命令 1 时序图

写实时数据 2：该命令为“011”，R/$\overline{W}$ 为“0”，写时间数据，数据从小时开始，图 6-17 为写数据命令 2 的时序图。

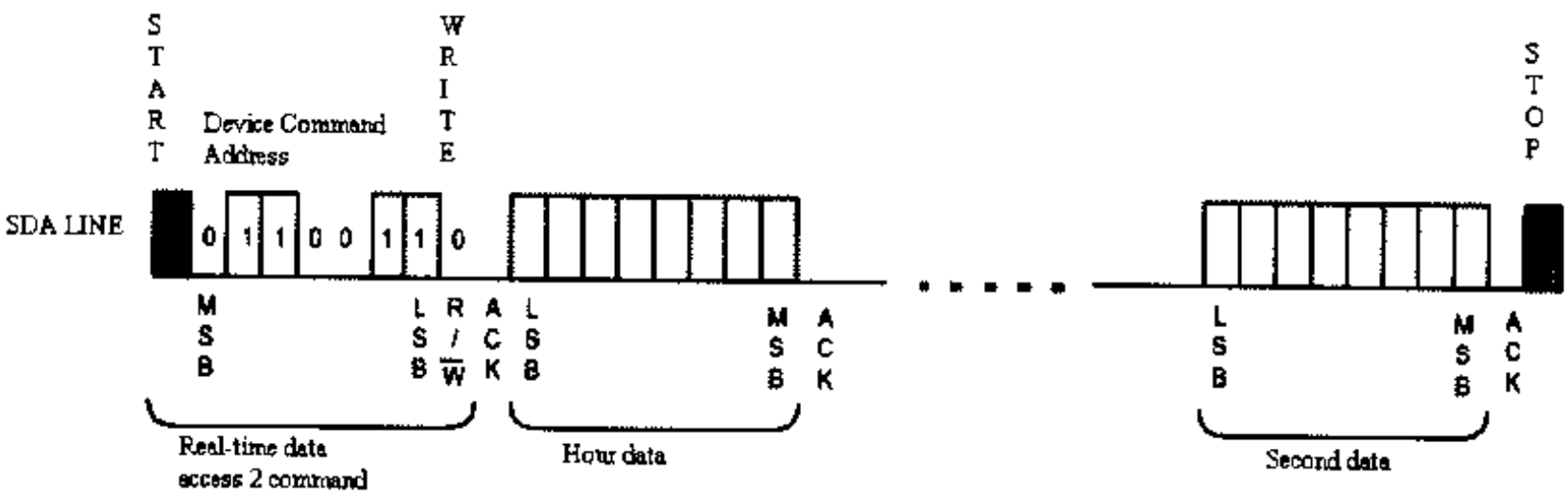



图 6-17　写数据命令 2 时序图

写取状态寄存器：该命令为“001”，R/$\overline{W}$ 为“0”，写芯片的状态寄存器的内容，图 6-18 为写状态寄存器命令的时序图。

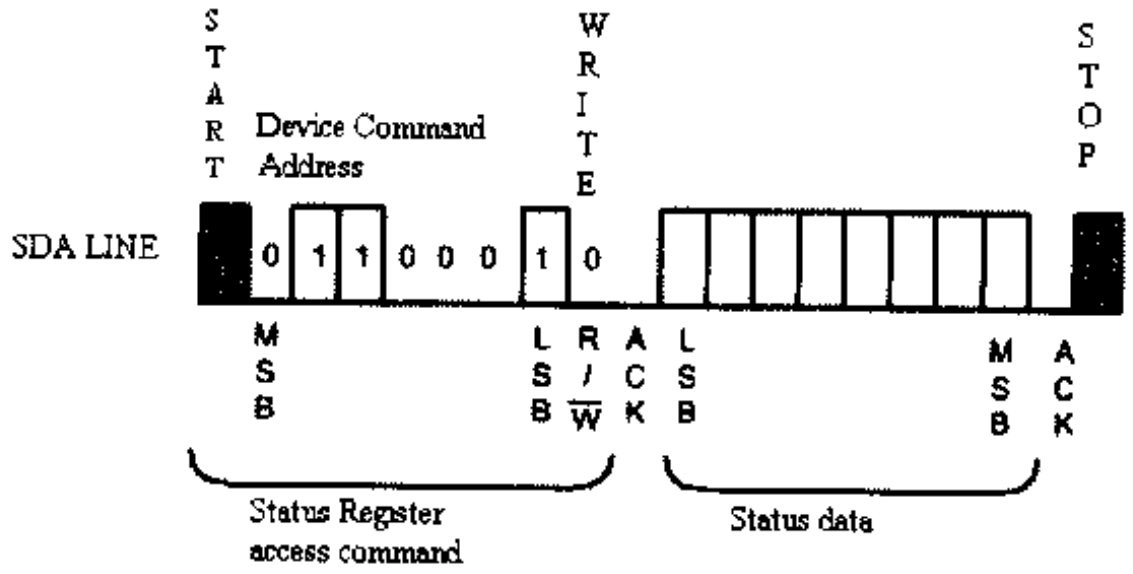



图 6-18　写状态寄存器命令时序图

经过这部分对 $I^2C$ 协议和对 S-3530A 操作的介绍，对软件实现 $I^2C$ 模块有一定的基础，下面具体来分析程序代码。

## 6.3.2 $I^2C$ 模块的实现

经过前面的介绍，对 $I^2C$ 协议有一定的了解，对 S-3530A 的操作也有深入的了解，下面具体分析 $I^2C$ 模块的软件实现，同时给出对 S-3530A 实现的代码。

### 1. SDA 高电平的产生

```
oid I²C_Set_sda_high( void )
{
    P1DIR |= SDA;          //将 SDA 设置为输出模式
    P1OUT |= SDA;          //SDA 管脚输出为高电平

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

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

### 3. SCL 高电平的产生

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

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

### 4. SCL 低电平的产生

```
void I²C_Set_sck_low ( void )
{
    P1DIR |= SCL;          //将 SCL 设置为输出模式
```

```
    P1OUT &= ~(SCL);          //SCL 管脚输出为低电平

    _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;

    I²C_Set_sda_high();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_high();
    for(i = 5;i > 0;i--);
    I²C_Set_sda_low();
    for(i = 5;i > 0;i--);
    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**

```
void I²C_STOP(void)
{
    int i;

    I²C_Set_sda_low();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_low();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_high();
    for(i = 5;i > 0;i--);
    I²C_Set_sda_high();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_low();
    Delay_ms(10);          //延迟一点时间

    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，如果需要精确的话，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。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函数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**

考虑到S-3530A对命令和数据发送的顺序不一致，发送数据的顺序可能是从最高位（MSB）到最低位（LSB），也可能在发送数据的时候要求从最低位到最高位发送数据。下面分别给出这两个操作的程序代码。

数据发送顺序从最高位到最低位的程序代码：

```
void I²C_TxHToL(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;
}
```

数据发送顺序从最低位到最高位的程序代码：

```
void I²C_TxLToH(int nValue)
{
    int i;
    int j;
    for(i = 0;i < 8;i++)
    {
    if(nValue & 0x01)
        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．读字节数据程序

```
//////////////////////////////////////////////
//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;
}
```

在字节数据发送和字节数据接收的基础上可以实现对 S-3530A 的操作函数。比如，设置状态寄存器、读状态寄存器、读时间信息和设置时间信息等函数。下面具体介绍程序代码。

### 9. 读状态寄存器

```
int  I²C_ReadSta(void)
{
    unsigned char nTemp = 0;

    I²C_START();                    //启动数据总线

    I²C_TxHToL(0x63);               //发送读状态寄存器命令
    nTemp = I²C_GetACK();           //等待 ACK

    nTemp = I²C_RxByte();
    I²C_SetACK();


    I²C_STOP();                     //停止总线

    return nTemp;
}
```

### 10. 写状态寄存器

```
void I²C_WriteSta(int nValue)
{
    int nTemp = 0;
    char chrTemp = 0;
    chrTemp = (char)(nValue & 0x00ff);

    I²C_START();                    //启动数据总线

    I²C_TxHToL(0x62);               //发送写状态寄存器命令
    nTemp = I²C_GetACK();           //等待 ACK
    if(nTemp & BIT3) return;
```

```
    I²C_TxLToH(chrTemp);
    nTemp = I²C_GetACK();          //等待 ACK
    if(nTemp & BIT3) return;

    I²C_STOP();                    //停止总线

    return;
}
```

### 11. 读取时间信息

```
//////////////////////////////////////////////
//获取时间信息，成功返回 1，错误返回 0
int  I²C_Read (char *pBuf)
{
    int nTemp = 0;

    I²C_START();                   //启动数据总线

    I²C_TxHToL(0x65);              //发送读时间信息命令

    nTemp = I²C_GetACK();          //等待 ACK

    pBuf[0] = I²C_RxByte();
    I²C_SetACK();

    pBuf[1] = I²C_RxByte();
    I²C_SetACK();

    pBuf[2] = I²C_RxByte();
    I²C_SetACK();

    pBuf[3] = I²C_RxByte();
    I²C_SetACK();

    pBuf[4] = I²C_RxByte();
    I²C_SetACK();

    pBuf[5] = I²C_RxByte();
    I²C_SetACK();

    pBuf[6] = I²C_RxByte();
    I²C_SetACK();

    pBuf[7] = I²C_RxByte();
    I²C_SetACK();
```

```
    I²C_STOP();                          //停止总线
    return nTemp;
}
```

### 12. 写时间信息

```
//////////////////////////////////////////
//写时间信息，成功返回 1，错误返回 0
int  I²C_Write(char *pBuf)
{
    int nTemp = 0;

    I²C_START();                         //启动数据总线

    I²C_TxHToL(0x64);                    //发送写时间信息命令
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[0]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[1]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[2]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[3]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[4]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[5]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[6]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_STOP();                          //停止总线
```

```
    return (nTemp & SDA);
}
```

另外还提供复位函数和时间的 24 小时制的函数，具体如下：

### 13. 复位函数

```
int  I²C_Reset(void)
{
    int nTemp = 0;

    I²C_START();                    //启动数据总线

    I²C_TxHToL(0x60);               //发送复位命令
    nTemp = I²C_GetACK();           //等待 ACK

    I²C_STOP();                     //停止总线
    return (nTemp & SDA);
}
```

### 14. 24 小时制设置函数

```
void Set24TimeMode(void)
{
    int nTemp;

    nTemp = 0;

    nTemp = I²C_ReadSta();
    nTemp |= BIT6;
    I²C_WriteSta(nTemp);
}
```

最后给出确认的函数，具体函数如下：

### 15. 获得 ACK

```
int  I²C_GetACK(void)
{
    int nTemp = 0;
    int j;

    _NOP();
    _NOP();
    I²C_Set_sck_low();
    for(j = 30;j > 0;j--);
    PIDIR &= ~(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);
}
```

### 16. ACK确认

```
void I²C_SetACK(void)
{
    I²C_Set_sck_low();
    I²C_Set_sda_low();
    I²C_Set_sck_high();
    I²C_Set_sck_low();
    return;
}
```

### 17. 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，也实现了S-3530A的操作模块，这样就很容易实现日历信息模块。日历信息模块主要是先设置好S-3530A，然后单片机可以不断的读取日历数据，为了实现定期读取数据，该模块采用定时器B来实现时间间隔控制，图6-19为日历模块的流程图。

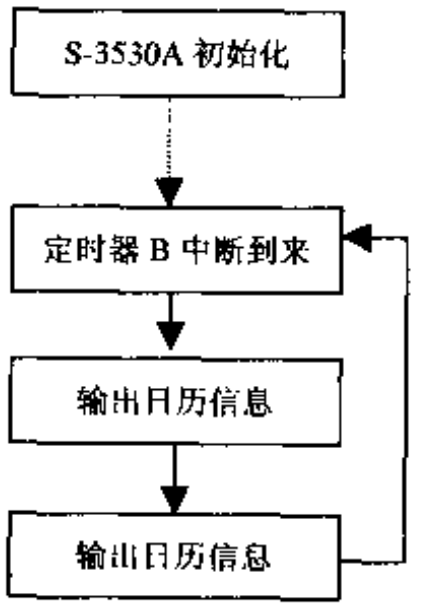



图 6-19　日历模块流程图

由图 6-19 可以看出，这个模块包括 S-3530A 操作处理和定时器 B 操作处理，下面分别介绍程序代码。

初始化模块：该模块主要完成定时器 B 和 $I^2C$ 模块的初始化。下面为具体实现的程序代码。

定时器 B 初始化：

```
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 1s

    TBCTL |= MC0;                   //增记数模式
}
```

通过以上代码可以看出，只需要调整适当的寄存器的值就可以改变设置。定时器 B 选择 ACLK 作为时钟源，通过设置 TBCCR0 来确定中断的时间，设置 TBCCTL0 寄存器的 CCIE 位来允许中断。

$I^2C$ 模块初始化：

```
void I²C_Initial( void )
{
    P1DIR |= BIT2;                  //将 SCL 管脚（P1.1）设置为输出管脚
    I²C_Set_sck_low();
    I²C_STOP();
    Delay_ms(10);
    return;
}
```

上面程序只是简单的设置了 SCL 管脚，由于 SDA 为双向数据通信，因此具体的设置在具体的操作时实现。

中断处理模块：该部分主要是完成定时读取日历信息。下面为具体的程序代码。

```
interrupt [TIMERB0_VECTOR] void TimerB_ISR(void)
{
    I²C_Read(pTime);
    Time_Flag = 1;
}
```

以上程序通过“$I^2C$_Read(pTime);”读取日历信息，读取的时间信息通过全局变量“pTime”实现与主程序之间进行数据交互，通过设置全局变量“Time_Flag”为“1”来通知主程序新的数据获得。

### 6.3.3　显示的实现

该部分主要完成数据的显示功能。在硬件设计中，显示电路直接与单片机的数据 I/O 口进行连接。P4.0～P4.6 是用来显示数据，P2.1、P2.2、P2.3、P2.4、P2.5 和 P2.6 是用来控制数

码管的选通状态，比如要在“星期”上显示，则要在P2.0管脚上给出高电平选通数码管进行显示。显示模块相对比较简单，只是简单的将数据显示在数码管上，该模块主要包括端口初始化和数据显示两个部分，下面根据具体的程序进行介绍。

端口初始化部分：该部分完成P4.0～P4.6端口、P2.0～P2.6端口的初始化工作，设置成相应的输入输出方向。具体的代码如下：

```
void Init_DispPort(void)
{
    //将所有的管脚在初始化的时候设置为输入方式
    P2DIR = 0;
    P4DIR = 0;
    //将所有的管脚设置为一般I/O口
    P2SEL = 0;
    P4SEL = 0;
    //将P2.0～P2.6设置为输出方向
    P2DIR |= BIT0;
    P2DIR |= BIT1;
    P2DIR |= BIT2;
    P2DIR |= BIT3;
    P2DIR |= BIT4;
    P2DIR |= BIT5;
    P2DIR |= BIT6;
    //将P4.0～P4.6设置为输出方向
    P4DIR |= BIT0;
    P4DIR |= BIT1;
    P4DIR |= BIT2;
    P4DIR |= BIT3;
    P4DIR |= BIT4;
    P4DIR |= BIT5;
    P4DIR |= BIT6;

    return;
}
```

数据显示部分：该部分主要是将数据显示到数码管上，在显示数据时需要选通不同的数码管。具体的程序如下：

```
void Display(char *pBuf)
{
    //数据表
    static char nLed[10]={0x7b,0x42,0x37,0x67,0x4e,0x6d,0x7d,0x43,0x7f,0x6f};
    //选通数码管

    //显示“星期”
    //选通数码管
    P2OUT |= BIT0;
```

```
P4OUT = nLed[pBuf[0]];
//显示“小时”
//选通数码管
P2OUT |= BIT3;
P4OUT = nLed[pBuf[1]];
//显示“小时”
//选通数码管
P2OUT |= BIT1;
P4OUT = nLed[pBuf[2]];
//显示“分”
//选通数码管
P2OUT |= BIT4;
P4OUT = nLed[pBuf[3]];

//显示“分”
//选通数码管
P2OUT |= BIT0;
P4OUT = nLed[pBuf[4]];
//显示“秒”
//选通数码管
P2OUT |= BIT3;
P4OUT = nLed[pBuf[5]];
//显示“秒”
//选通数码管
P2OUT |= BIT1;
P4OUT = nLed[pBuf[6]];

return;
}
```

该程序做了一个数据表，这样要显示某一个数字，只需要根据数组的下标就可以取得数据，然后直接赋值给 P4OUT，从而实现显示，数据的下标为传进来的数据。在该程序中只是简单的显示星期、小时、分、秒等数据。其中星期只显示一位，而小时、分和秒每个数据显示两位。

### 6.3.4　系统软件流程

经过前面对硬件系统的介绍，对软件系统有一个基本的认识，经过前面对各个模块的介绍，实现整个软件系统就比较容易了。整个软件主要包括日历模块的操作和显示的实现。主程序首先初始化端口、定时器 B 和 $I^2C$ 模块等，然后设置好 S-3530A 芯片，使它处于工作状态，启动定时器 B，当每一次定时器中断到来时就读取日历信息，将获得的数据放到全局的缓冲区里去，通过设置一个全局变量来通知主程序获得新的日历数据，主程序将获得的新的日历数据显示在 LED 上，同时清除全局变量的标志。整个系统的软件流程图如图 6-20 所示。

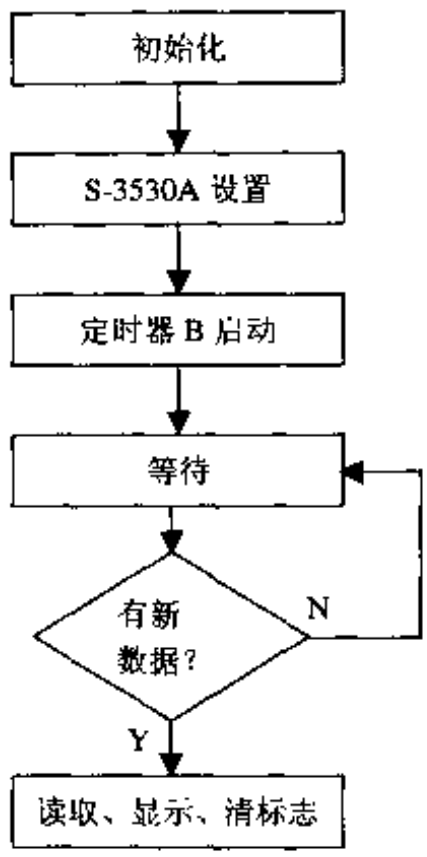



图 6-20　软件流程图

通过图 6-20 可以看出，该软件系统主要就是通过 $I^2C$ 对 S-3530A 操作，将得到的数据显示出来，结合上面的流程图，给出一个简单的程序，具体的程序如下。

```
char Time_Flag;
char pTime[7];
void main(void)
{
    int i;
    char chrTemp[7];
    char chrHi;
    char chrLow;
    char pBuf[7];

    WDTCTL = WDTPW + WDTHOLD;    //关闭看门狗

    _DINT();                     //关闭中断
    //初始化
Init_CLK();
    Init_TimerB();
    Init_DispPort();
    I2C_Initial();
    //初始化变量
    Time_Flag = 0;
    _EINT();                     //打开中断
    ////////////////////////////////
    //设置为 24 小时制
    Set24TimeMode();
    pTime[0] = 4;
```

```
pTime[1] = 1;
pTime[2] = 1;
pTime[3] = 1;
pTime[4] = 1;
pTime[5] = 1;
pTime[6] = 1;
//设置时间
I²C_Write(pTime);
//循环处理
for(;;)
{
 if(Time_Flag == 1)
 {
    Time_Flag = 0;//清楚标志

    for(i = 0;i < 7;i++)
       chrTemp[i] = pTime[i];

    //星期
    pBuf[0] = chrTemp[3];

    //小时
    chrHi = (char)((chrTemp[4] / 16) & 0x0f);
    chrLow = (char)(chrTemp[4] & 0x0f);
    pBuf[1] = chrHi;
    pBuf[2] = chrLow;

    //分
    chrHi = (char)((chrTemp[5] / 16) & 0x0f);
    chrLow = (char)(chrTemp[5] & 0x0f);
    pBuf[3] = chrHi;
    pBuf[4] = chrLow;

    //秒
    chrHi = (char)((chrTemp[6] / 16) & 0x0f);
    chrLow = (char)(chrTemp[6] & 0x0f);
    pBuf[5] = chrHi;
    pBuf[6] = chrLow;
    //显示
    Display(pBuf);
 }
}
return;
}
```

上面的程序从日历芯片里读取的数据是BCD格式，所以需要做相应的处理。

## 6.4 系统调试

前面已经介绍了整个系统的硬件设计和软件设计，该部分结合前面介绍的来讨论整个系统的联调。系统的调试包括硬件调试和软件调试，下面分别进行介绍。

**1．系统硬件调试**

系统的硬件调试很简单，先调试电源电路和复位电路，只要这两个部分能正常工作，再进行单片机的调试，如果单片机的晶振能起振的话，则整个硬件的单片机部分没有问题。关于S-3530A的调试也很简单，只需要用示波器测试振荡电路有没有时钟输出，如果有波形的话，则该部分硬件方面没有问题。关于显示部分需要结合具体的软件进行调试。

**2．系统软件调试**

前面分别介绍了各个软件部分，为了能够进行整个系统的联调，需要软件和硬件调试结合起来，对于不同的硬件部分，应该调用不同的软件模块进行测试。关于软件部分调试需要强调的是$I^2C$模块，这里主要是考虑$I^2C$的工作频率不能超过器件的最大频率。经过联合调试，整个系统的软件和硬件能够正确运行。

## 6.5 实例总结

该系统通过一个日历芯片实现的日历系统系统具有设计简单、运行可靠等特点。通过软件测试和硬件测试证明该系统能够安全可靠地运行。由于单片机采用的是MSP430F149，这样可以很容易实现系统的硬件升级。由于该硬件系统中的单片机还有两个UART串口资源，因此该硬件系统可以进一步升级，实现可以与上位机进行通信的系统，这样就可以通过上位机来配置时钟芯片。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自己功能，实现更为完整的系统。经过这章的介绍，主要是能让读者实现通过一般I/O口来实现$I^2C$模块，为了能使读者全面了解$I^2C$模块，附录给出了整个$I^2C$的程序源代码。

### 附录：$I^2C$程序模块

```
//I²C 模块的头文件
#include <MSP430X14X.h>
//定义管脚
#define SCL BIT1
#define SDA BIT2
//定义函数
void I²C_Initial(void);
void I²C_Set_sda_high(void);
void I²C_Set_sda_low (void);
void I²C_Set_sck_high(void);
void I²C_Set_sck_low (void);
int  I²C_GetACK(void);
void I²C_SetACK(void);
```

```
void I²C_SetNAk(void);
void I²C_START(void);
void I²C_STOP(void);
void I²C_TxHToL(int);
void I²C_TxLToH(int);
int  I²C_RxByte(void);
int  I²C_Read (char *pBuf);
int  I²C_Write(char *pBuf);
int  I²C_Reset(void);
int  I²C_ReadSta(void);
void I²C_WriteSta(int);
void Delay_ms(unsigned long nValue);
void Delay_us(unsigned long nValue);
void Test(unsigned char nVal);
void Set24TimeMode(void);
//I²C.c 文件
#include "I²C.h"

void I²C_Initial( void )
{
    P1DIR |= BIT2;              //将 SCL 管脚 (P1.2)设置为输出管脚
    I²C_Set_sck_low();
    I²C_STOP();
    Delay_ms(10);
    return;
}
void I²C_Set_sda_high( void )
{
    P1DIR |= SDA;               //将 SDA 设置为输出模式
    P1OUT |= SDA;               //SDA 管脚输出为高电平

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

    _NOP();
    _NOP();
    return;
}
void I²C_Set_sck_high( void )
```

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

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

    _NOP();
    _NOP();
    return;
}
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);
}
void I²C_SetACK(void)
{
    I²C_Set_sck_low();
    I²C_Set_sda_low();
    I²C_Set_sck_high();
    I²C_Set_sck_low();
    return;
}
void I²C_SetNAk(void)
```

```
{
    I²C_Set_sck_low();
    I²C_Set_sda_high();
    I²C_Set_sck_high();
    I²C_Set_sck_low();
    return;
}
void I²C_START(void)
{
    int i;

    I²C_Set_sda_high();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_high();
    for(i = 5;i > 0;i--);
    I²C_Set_sda_low();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_low();
    return;
}
void I²C_STOP(void)
{
    int i;

    I²C_Set_sda_low();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_low();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_high();
    for(i = 5;i > 0;i--);
    I²C_Set_sda_high();
    for(i = 5;i > 0;i--);
    I²C_Set_sck_low();
    Delay_ms(10);          //延迟一点时间

    return;
}
void I²C_TxHToL(int nValue)
{
    int i;
    int j;
    //I²C_Set_sck_low();
    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;
}
void I²C_TxLToH(int nValue)
{
    int i;
    int j;
    for(i = 0;i < 8;i++)
    {
    if(nValue & 0x01)
        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;
}
//////////////////////////////////////////////////
//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;
}
////////////////////////////////////////////
//获取时间信息，成功返回 1，错误返回 0
int  I²C_Read (char *pBuf)
{
    int nTemp = 0;

    I²C_START();                    //启动数据总线

    I²C_TxHToL(0x65);               //发送读时间信息命令

    nTemp = I²C_GetACK();           //等待 ACK

    pBuf[0] = I²C_RxByte();
    I²C_SetACK();

    pBuf[1] = I²C_RxByte();
    I²C_SetACK();

    pBuf[2] = I²C_RxByte();
    I²C_SetACK();

    pBuf[3] = I²C_RxByte();
    I²C_SetACK();

    pBuf[4] = I²C_RxByte();
    I²C_SetACK();

    pBuf[5] = I²C_RxByte();
    I²C_SetACK();

    pBuf[6] = I²C_RxByte();
    I²C_SetACK();

    pBuf[7] = I²C_RxByte();
```

```
    I²C_SetACK();

    I²C_STOP();                          //停止总线
    return nTemp;
}
//////////////////////////////////////////'
//写时间信息，成功返回 1，错误返回 0
int  I²C_Write(char *pBuf)
{
    int nTemp = 0;

    I²C_START();                         //启动数据总线

    I²C_TxHToL(0x64);                    //发送写时间信息命令
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[0]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[1]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[2]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[3]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[4]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[5]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_TxLToH(pBuf[6]);
    nTemp = I²C_GetACK();                //等待 ACK
    if(nTemp & BIT3) return 0;

    I²C_STOP();                          //停止总线
    return (nTemp & SDA);
```

```
}
int  I²C_Reset(void)
{
    int nTemp = 0;

    I²C_START();                    //启动数据总线

    I²C_TxHToL(0x60);               //发送复位命令
    nTemp = I²C_GetACK();           //等待 ACK

    I²C_STOP();                     //停止总线
    return (nTemp & SDA);
}
void Delay_ms(unsigned long nValue) //毫秒为单位，8MHz 为主时钟
{
    unsigned long nCount;
    int j;
    unsigned long j;
    nCount = 2667;
    for(i = nValue;i > 0;i--)
    {
     for(j = nCount;j > 0;j--);
    }
    return;
}
void Delay_us(unsigned long nValue)//微秒为单位，8MHz 为主时钟
{
    int nCount;
    int i;
    int j;
    nCount = 3;
    for(i = nValue;i > 0;i--)
    {
     for(j = nCount;j > 0;j--);
    }
    return;
}
void Test(unsigned char nVal)
{
    int nTemp = 0;

    I²C_START();                    //启动数据总线

    I²C_TxHToL(nVal);               //发送复位命令
    nTemp = I²C_GetACK();           //等待 ACK

    I²C_STOP();                     //停止总线
```

```
    return;
}
int  I²C_ReadSta(void)
{
    unsigned char nTemp = 0;

    I²C_START();                    //启动数据总线

    I²C_TxHToL(0x63);               //发送读状态寄存器命令
    nTemp = I²C_GetACK();           //等待 ACK

    nTemp = I²C_RxByte();
    I²C_SetACK();


    I²C_STOP();                     //停止总线

    return nTemp;
}
void I²C_WriteSta(int nValue)
{
    int nTemp = 0;
    char chrTemp = 0;
    chrTemp = (char)(nValue & 0x00ff);

    I²C_START();                    //启动数据总线

    I²C_TxHToL(0x62);               //发送写时间信息命令
    nTemp = I²C_GetACK();           //等待 ACK
    if(nTemp & BIT3) return;

    I²C_TxLToH(chrTemp);
    nTemp = I²C_GetACK();           //等待 ACK
    if(nTemp & BIT3) return;

    I²C_STOP();                     //停止总线
    return;
}
void Set24TimeMode(void)
{
    int nTemp;
    nTemp = 0;
    nTemp = I²C_ReadSta();
    nTemp |= BIT6;
    I²C_WriteSta(nTemp);
}
```

# 第7章　MODEM数据传输的通信系统设计

在单片机应用系统中，与远端进行数据通信已经变得越来越重要，利用电话线进行数据传输是一种非常方便、可靠和经济的传输手段。本章将介绍采用 MODEM 实现的数据传输系统。该系统具有运行可靠、配置灵活等特点。

## 7.1　系统描述

目前，在许多数据采集的应用中需要将数据传输到远端，采用 MODEM 通过电话线传输数据已经在日常生活中得到了广泛的应用（比如 POS 机系统）。通过 MODEM 传输数据，使采集系统的应用空间得到扩伸。公共电话网（PSTN）已普及全国，采用 MODEM 实现的远程监控传输系统与一般的电话线相连接，即可获得简单实用的远程监控系统。另外由于电话线的普及以及有线传输具有可靠性和保密性等特点，因此采用 MODEM 传输数据的系统在诸如电子银行等领域也得到了广泛的应用。

本章将讨论基于单片机实现的 MODEM 数据传输系统。本系统采用 MSP430F149 作为整个系统的 MCU。采用一个嵌入式的 MODEM 作为系统传输数据的 MODEM，MODEM 与单片机通过串口进行连接，考虑到 MODEM 使用 5V 供电，因此需要考虑进行电平转换。为了增加系统使用的灵活性，利用单片机的另外一个片内串口实现一个与上位机进行通信的接口，从而实现整个系统的配置功能。该系统具有运行可靠、接口简单等特点，具体的系统框图如图 7-1 所示。

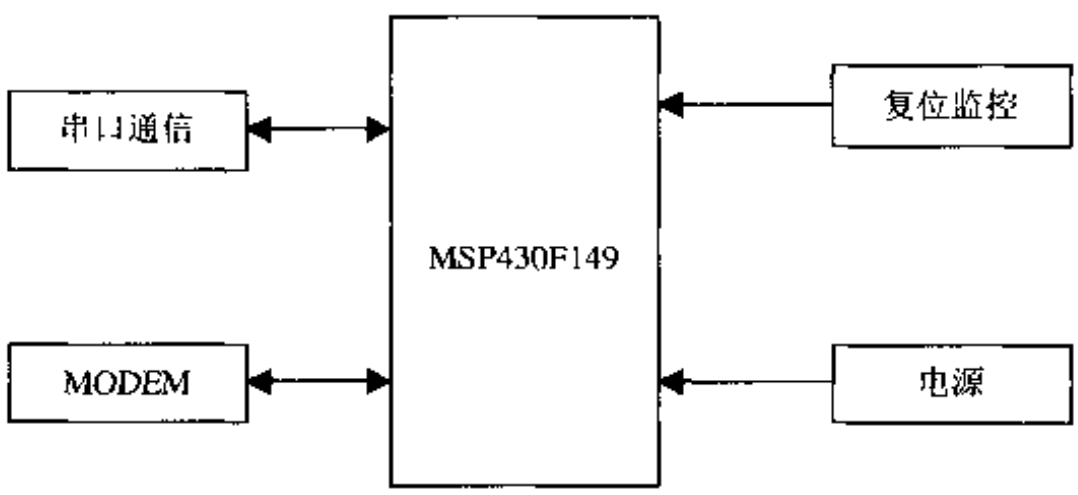



图 7-1　系统框图

由图 7-1 可以看出，整个系统硬件简单、接口方便。电源部分为整个系统提供稳定的 3V 和 5V 电源，其中 3V 是为单片机提供的工作电压，5V 是为 MODEM 提供的工作电压。复位监控部分为系统在开机的时候提供复位信号，并监控系统工作的电压是否正常，当不正常的时候使系统复位。MODEM 通过串口与单片机进行连接，由于单片机系统的工作电压是 3V，

而 MODEM 的工作电压是 5V，这样在接口的时候需要考虑电平转换。串口通信是利用单片机的另外一个片上串口实现的与上位机进行通信的模块，考虑到上位机的接口电平与单片机的接口电平不同，需要进行电平转换。具体各个功能模块的硬件设计在下面一节进行详细介绍。

## 7.2 系统硬件设计

该系统的硬件主要由电源模块、复位监控模块、MODEM 模块、串口通信模块及单片机处理模块组成。下面就具体的电路进行介绍。

### 7.2.1 MODEM 模块介绍

在介绍具体的电路之前，先介绍一下 MODEM 通信模块。该模块采用的是工业级的嵌入式 MODEM（以下简称 eModem），该 MODEM 模块直接通过串口与单片机进行连接。该模块采用 MODEM 专用芯片，支持标准 AT 命令集，具有拨号/自动应答功能。eModem 模块采用专门的结构设计，使用双列 SIP14 插针接口，具有尺寸小，连接方便的特点。eModem 电气接口采用标准串行总线连接方式支持标准 RS232 电平或 TTL 电平，使该模块可以方便地嵌入到各种单片机系统、数据采集系统或其他工业控制系统中。该模块具有以下特性。

- 具有 14.4Kb/s～56Kb/s 传输速率，支持 V.32bis、V.34、V.90 三种标准。
- 支持标准 AT 命令集信号/自动应答功能。
- 采用先进的双面 SMD 制造工艺。
- 采用两列 SIP14 插接口。
- 采用串行总线接口，支持标准 RS232 电平或 TTL 电平。
- 符合国家相关电信标准。
- 具有防雷击保护功能。
- 外形尺寸：74.4mm×37.8mm×13.2mm。

为了增加对 MODEM 的了解，图 7-2 为 MODEM 接口示意图。

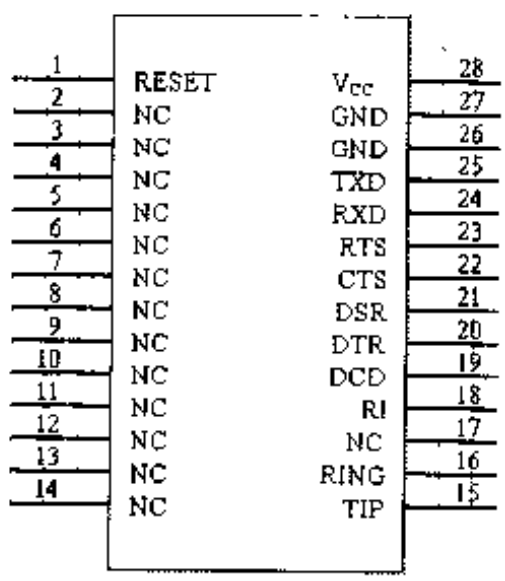



图 7-2 MODEM 接口示意图

由图 7-2 可以看出，MODEM 通过串口与单片机进行连接。为了了解各个管脚的功能，对各个管脚进行详细的介绍。

- RESET：复位信号。低电平有效。
- NC：空管脚。悬空。
- TIP：外接电话线。
- RING：外接电话线。
- RI：振铃信号。
- DCD：电话线上是否有载波的标志。
- DTR：数据终端准备好。DTE 控制该信号线有效。
- DSR：数据设备准备好。eModem 控制该信号向 DTE 报告状态。
- CTS：清除发送。该信号有效表示 eModem 准备接受 DTE 的数据。
- RTS：请求发送。该信号有效表示 DTE 准备发送数据到 eModem。
- RXD：发送数据到 DTE。
- TXD：接收数据从 DTE。
- GND：电源地。
- $V_{CC}$：+5V 电源。

通过上述对管脚的描述，就可以设计出 MODEM 与单片机的接口，由于单片机串口采用的是 UART，因此只需要两线（TXD、RXD）就可以实现通信任务，至于其他控制线可以不连接，下面具体介绍接口设计。

## 7.2.2　接口设计

整个系统的硬件比较简单，接口设计也非常容易，主要是 MODEM 与单片机接口的设计，但为了对整个硬件系统有全面的认识，该部分不仅仅分析 MODEM 与单片机的接口电路，也介绍整个系统的其他部分电路。主要电路有 RS232 电路、MODEM 接口电路、电压升压电路、CPU 处理模块和 3V 电源及复位模块。下面就具体的电路进行介绍。

### 1．3V 电源电路

整个系统除了 MODEM 外都采用 3.0V 供电，考虑到硬件系统对电源要求具有稳压功能和纹波小等特点，另外也考虑到硬件系统的低功耗等特点，因此该硬件系统的电源部分采用 TI 公司的 TPS76330 芯片实现，该芯片的输出电流为 150mA，可以满足系统的电流要求，另外该芯片具有很小的封装，因此能有效节约 PCB 板的面积。电源电路如图 7-3 所示。

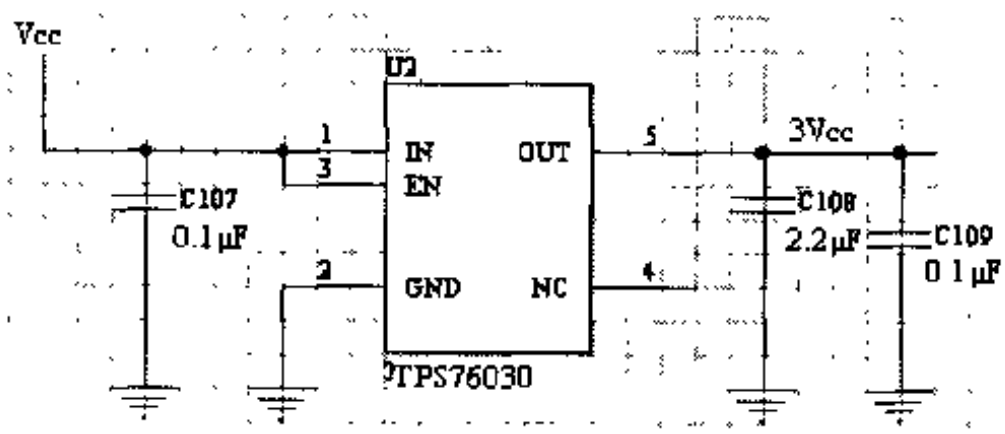



图 7-3　电源电路

为了使输出电源的纹波小，在输出部分用了一个 2.2μF 和 0.1μF 的电容，另外在芯片的输入端也放置一个 0.1μF 的滤波电容，减小输入端受到的干扰。

### 2．复位电路

在单片机系统里，单片机需要复位电路，复位电路可以采用 R-C 复位电路，也可以采用复位芯片实现的复位电路，R-C 复位电路具有经济性，但可靠性不高，用复位芯片实现的复位电路具有很高的可靠性，因此为了保证复位电路的可靠性，该系统采用复位芯片实现的复位电路，该系统采用 MAX809 芯片。复位电路如图 7-4 所示。

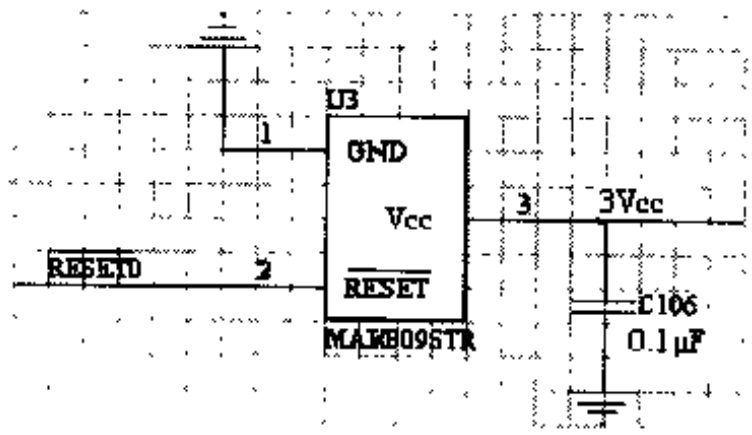



图 7-4　复位电路

为了减小电源的干扰，还需要在复位芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 3．MODEM 接口电路

通过前面的介绍，我们对 MODEM 模块已经有了深入的了解，因此设计它的接口电路就相对比较容易了。虽然 MODEM 提供了许多控制线，但是考虑到设计接口的简单性，并且由于 MODEM 模块与单片机是通过 UART 进行连接，所以只需要采用两线（TXD、RXD）连接，关于 MODEM 通信的控制通过软件来实现，采用软件实现控制具有使用比较灵活的特点，这样也很好避免了过多的硬件信号的检测。在设计的时候需要考虑 MODEM 模块与单片机系统的共地问题。MODEM 的两根线（TIP、RING）直接与电话线连接，由于 MODEM 模块本身考虑了电话线端的处理，因此这里不需要做任何的处理。图 7-5 为 MODEM 模块的接口设计。

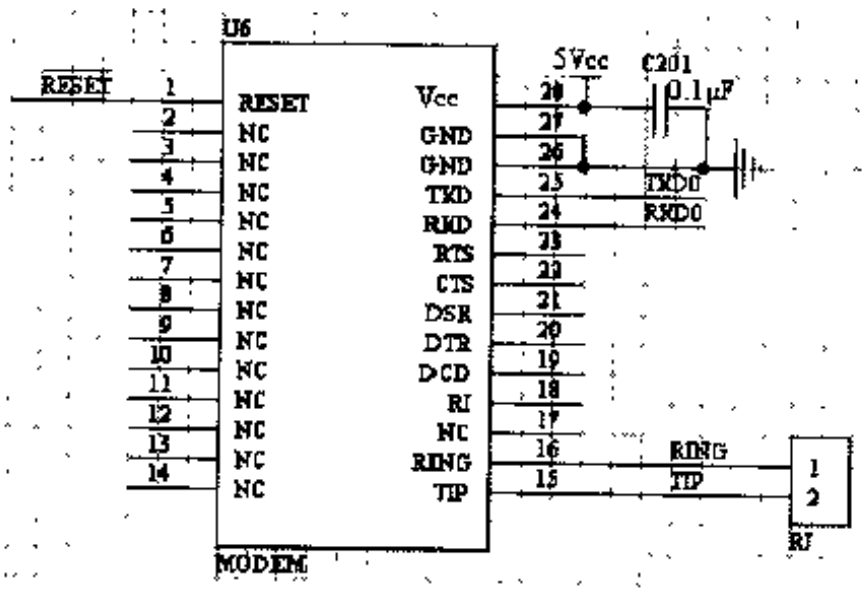



图 7-5　MODEM 模块的接口设计

由图可以看出，MODEM 接口电路设计比较简单。虽然 MODEM 模块的工作电压是 5V，单片机的工作电压是 3V，但由于单片机和 MODEM 模块的串口电平都是 TTL 电平，因此串口线直接与单片机进行连接。电话线接口也直接与电话线进行连接就可以了。考虑到为了减

小电源的干扰，还需要在 MODEM 模块的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 4. RS232 电路

该系统实现的 RS232 电路主要是与上位机进行通信，实现单片机系统与上位机进行通信处理。由于单片机与上位机进行通信时接口电平不同，因此需要进行接口转换，这里采用 SP3220 芯片来完成接口电平的转换。关于 SP3220 芯片在前面给出了介绍，在这里不再进行说明。为了加深对 RS232 电路的了解，下面给出该电路的设计图，图 7-6 为采用 SP3220 芯片实现的 RS232 电路图。

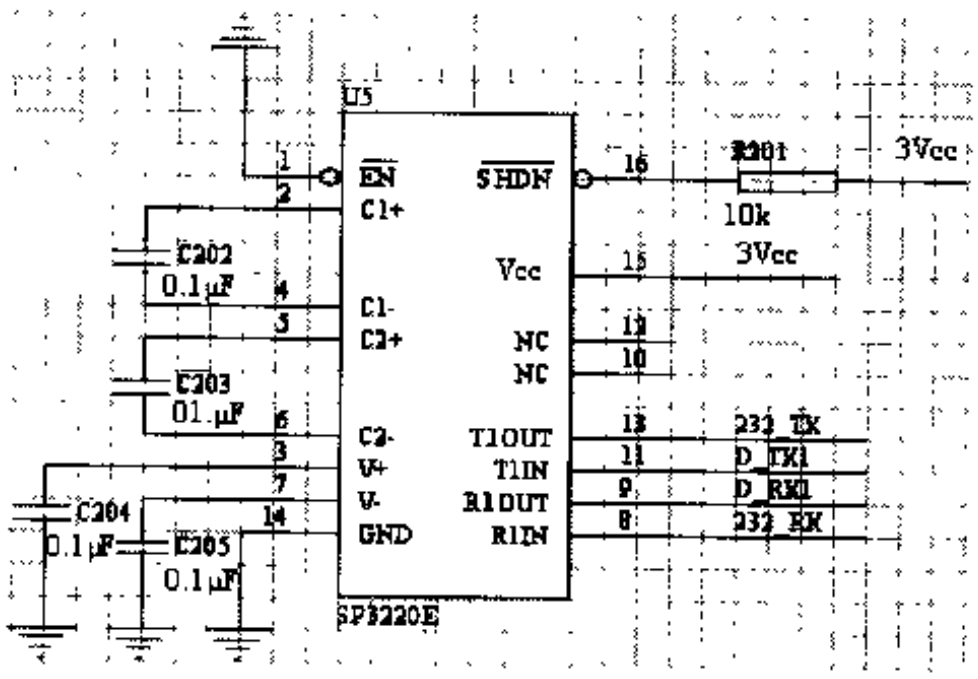



图 7-6　RS232 电路图

由图可以看出，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−分别放置 0.1μF 的电容实现充电作用，满足相应的充电泵的要求。管脚 T1OUT、TIN、RIOUT 和 RIN 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。考虑到减小电源的干扰，还需要在芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 5. 升压电路

由于 MODEM 模块的工作电压是 5V，而系统的电源供电是 3V，这样系统的供电就满足不了 MODEM 模块的要求。考虑到电池应用的场合，这样整个系统采用 3V 供电，正好两节电池。由于整个系统的供电是 3V，因此需要考虑将 3V 电压升压成 5V 电压供 MODEM 模块工作。3V 升到 5V 的升压芯片比较多，但大多数的芯片都需要使用电感，考虑到电感使用没有电容方便，加上电感的贴片器件不如电容的贴片器件容易购买，因此 3V 升到 5V 的升压芯片采用的是 TI 公司的 TPS60130 芯片。通过 TPS60130 芯片来实现将一部分 3V 电压转换成 5V 电压供 MODEM 模块使用。TPS60130 芯片具有价格便宜、使用方便等特点。TPS60130 芯片具有以下特性：

- 宽的输入电压范围，输入电压为 2.7V～5.4V。
- 输出电流最大可以达到 300mA。

- 不需要外接电感。
- 输出电压纹波小。
- 只需要 4 个外部元件。
- 通过 ENABLE 管脚可以进入低功耗状态。
- 芯片具有很小的封装，封装为 20 管脚的 PowerPAD 封装。

为了增加对该芯片的认识，下面给出该芯片的框图，如图 7-7 所示。由图 7-7 可以看出该芯片内部具有控制电路，这样就可以通过 ENABLE 管脚上的信号使芯片进入低功耗状态，经过内部的比较电路实现低电压检测功能，芯片内部的充电泵实现电压的升压功能。充电泵受到启动/关闭电路和控制电路的控制。

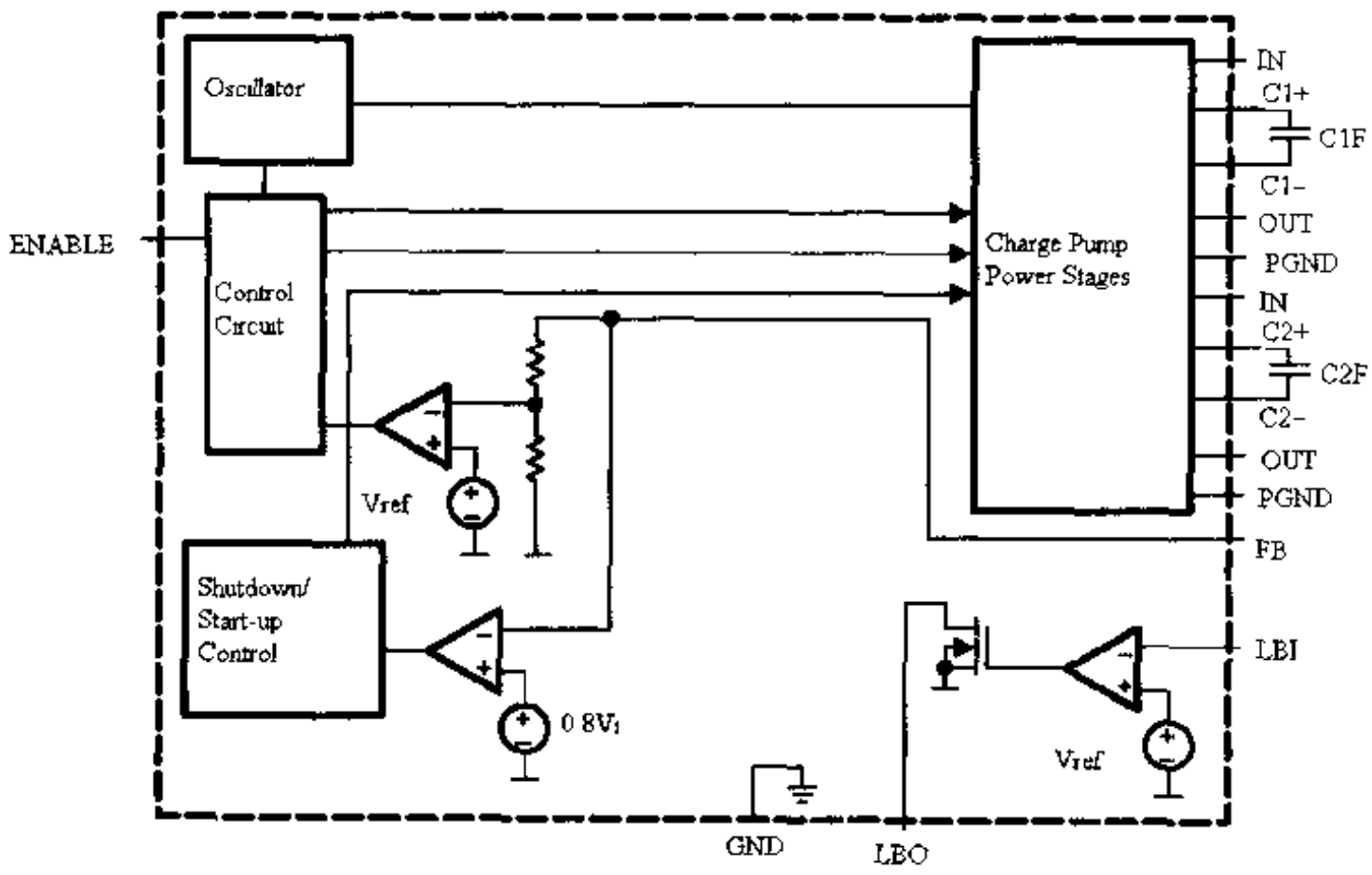



图 7-7　TPS60130 芯片框图

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 7-8 所示。

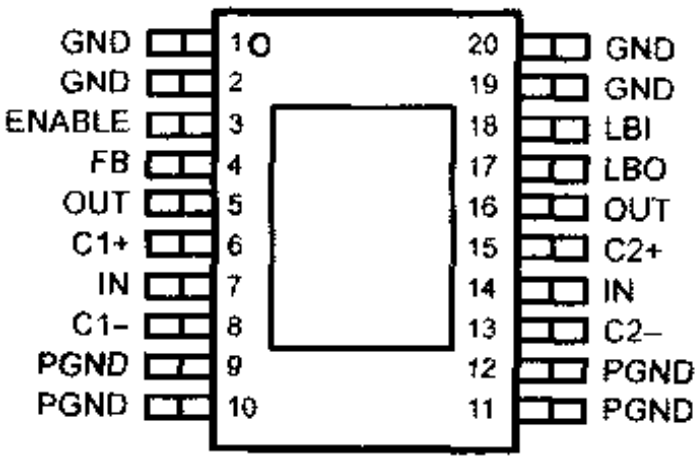



图 7-8　TPS60130 芯片管脚图

由图 7-8 可以看出，该芯片有 20 个管脚，该芯片有很多的地管脚，这样在芯片内部具有丰富的地资源。由于芯片不需要电感，只需要电容，这样该芯片使用起来简单，只需要简单的外围电路，下面对具体的管脚进行介绍。

- GND：地。内部参考电路和控制电路的模拟地。通过短线与 PGND 连接起来。
- PGND：电源地。充电泵的电流流经该管脚。
- ENABLE：使能管脚。当处于一般操作的时候，将该管脚与 IN 管脚连在一起。当该管脚处于低电平的时候，该芯片处于低功耗状态。
- FB：反馈输入。与 OUT 连在一起，线尽可能的短，这样可以获得好的性能。
- OUT：5V 电压输出管脚。将两个 OUT 管脚用短线连接在一起。
- IN：输入管脚。将所有的 IN 管脚用短线连接在一起。
- LBI：低电池检测输入。该管脚的输入电压与内部的 1.21V 参考电压进行比较，如果该管脚不用的话，需要将该管脚与地相连。
- LBO：低电池检测输出。如果该管脚不用的话，该管脚悬空。
- C1+：电容 C1 的正端。
- C1−：电容 C1 的负端。
- C2+：电容 C2 的正端。
- C2−：电容 C2 的负端。

经过对 TPS60130 芯片的框图和管脚的介绍，对该芯片有深入的了解，下面介绍该芯片的具体电路的设计。图 7-9 为升压电路设计图。

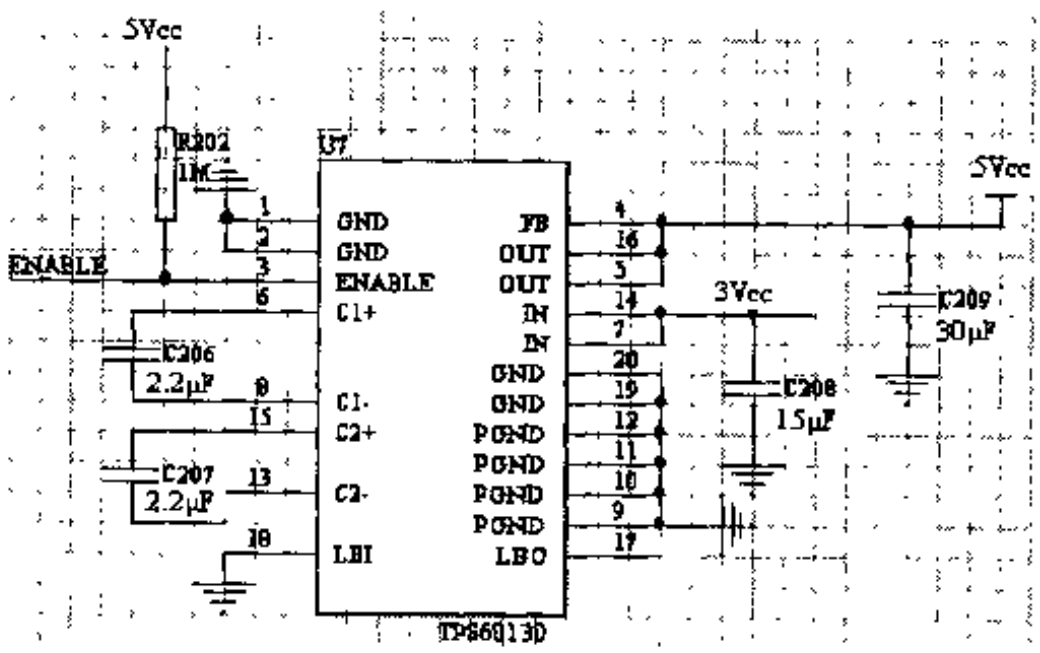



图 7-9　升压电路设计图

由图7-9可以看出，该芯片的电路设计比较简单，只需要4个电容。考虑到减小电源的干扰，还需要在芯片的电源输入管脚加一个15μF的电容来实现滤波，以减小输入端受到的干扰。在电源的输出端需要增加一个33μF的电容来实现滤波，从而减小输出电压的纹波。对于TPS60130的ENABLE管脚，将该管脚与5V电压输出管脚相连，ENABLE管脚是通过一个1MΩ的电阻与5V电压输出管脚相连，同时也与单片机的一个一般I/O口连接，通过单片机来实现对TPS60130芯片的工作状态的控制。因为MODEM的功耗比较大，考虑到某些低功耗应用，这样只要在需要传输数据的时候打开MODEM，使MODEM处于工作状态，平时不传数据的时候将MODEM关闭，处于低功耗状态。由于MODEM本身没有控制其低功耗状态的管脚，因此对MODEM的工作状态的控制是通过对TPS60130芯片的工作状态进行控制的。具体的原理是这样的，当单片机给一个低电平给TPS60130芯片的ENABLE管脚时，TPS60130芯片处于低功耗状态，这时该芯片的OUT管脚没有输出，这样MODEM就不工作，因为这个时候没有

它工作的电压；当单片机给一个高电平给TPS60130芯片的ENABLE管脚时，TPS60130芯片处于工作状态，这时该芯片的OUT管脚有5V电压输出，这样MODEM就可以工作，因为这个时候有它工作的电压。

### 6．单片机处理电路

单片机电路作为整个系统的核心控制部分，主要是完成与 MODEM 的通信，与上位机进行通信。单片机与 MODEM 通信采用单片机的串口 0（UART0）实现，虽然单片机与 MODEM 的供电电压不同，但它们的接口电平都是 TTL 电平，因此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 1（UART1）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。另外单片机还需要通过一个 I/O 管脚来控制升压芯片（TPS60130）的工作，分别在管脚上输出高电平或者低电平来使升压芯片工作或者停止工作。整个系统的单片机电路相对来说十分简单。下面给出具体的单片机电路图，如图 7-10 所示。

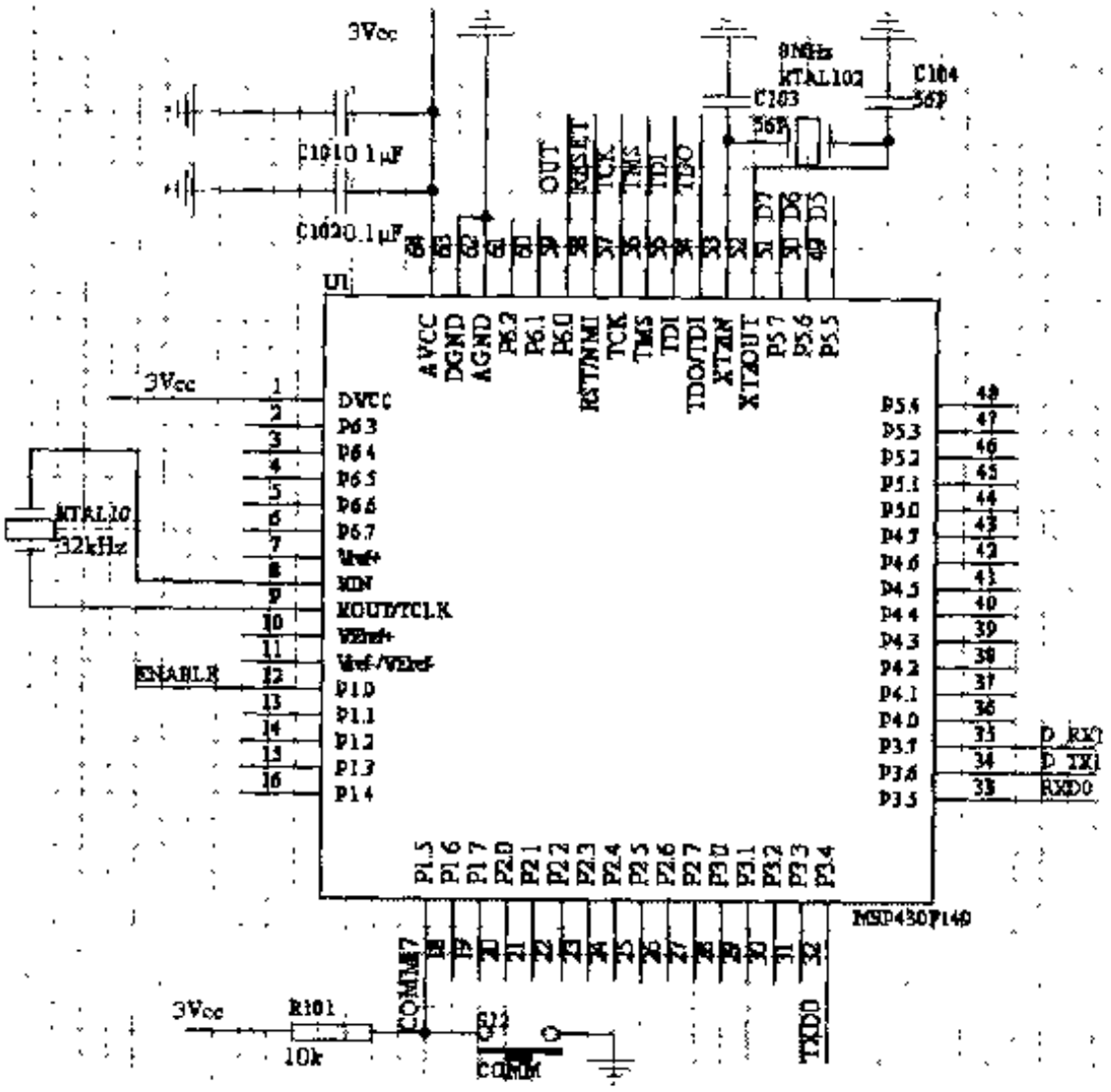



图 7-10　单片机电路

由图 7-10 可以看出，单片机的接口电路非常简单。在单片机的时钟设计上与其他单片机有一定的区别，MSP430F149 单片机采用两个时钟输入（即一个 32kHz 的时钟信号，一个 8MHz 的时钟信号）。该系统的时钟部分都是采用晶体振荡器实现的。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。另外单片机还有模拟电源的输入端，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。利用单片机的串口 0 与 MODEM 接口。为了控制 MODEM 传输数据的时刻，利用单片机的一般 I/O 口 P1.5 来作为启动通信的按键，由于 P1.5 可以作为中断口使用，这里设计成低电平触发方式，

所以需要将该管脚拉高。单片机的串口 1 与上位机进行通信，因此串口 1 与 RS232 芯片进行连接。另外单片机的 P1.0 作为输出口，与升压芯片的 ENABLE 管脚进行连接。

经过该节的介绍，对系统的硬件有了清楚的认识，下一节介绍系统的软件设计。

## 7.3　系统软件设计

通过对前面系统硬件的了解，这一节将介绍系统的软件设计。系统的软件主要包括 MODEM 通信模块、串口通信模块和主处理模块。下面就具体的各个模块进行介绍。

### 7.3.1　UART 模块的实现

UART 模块主要包括两个串口通信模块。串口 0 通信模块主要是完成与 MODEM 模块进行通信，从而实现数据的传输。串口 1 通信模块主要是完成单片机与上位机的通信，从而实现与上位机进行数据交互（比如完成配置管理或者将数据送到上位机）。由于 MSP430F149 单片机具有两个片内的 UART，因此实现两个串口通信相当容易，只需要设置适当的寄存器就可以使串口工作起来。两个串口通信都采用中断机制，两个串口的发送数据和接收数据都分别采用中断方式，当接收到有数据时，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。由于两个串口通信的工作原理是一样的，因此串口的流程也是一样的。下面给出串口通信的流程图，如图 7-11 所示。

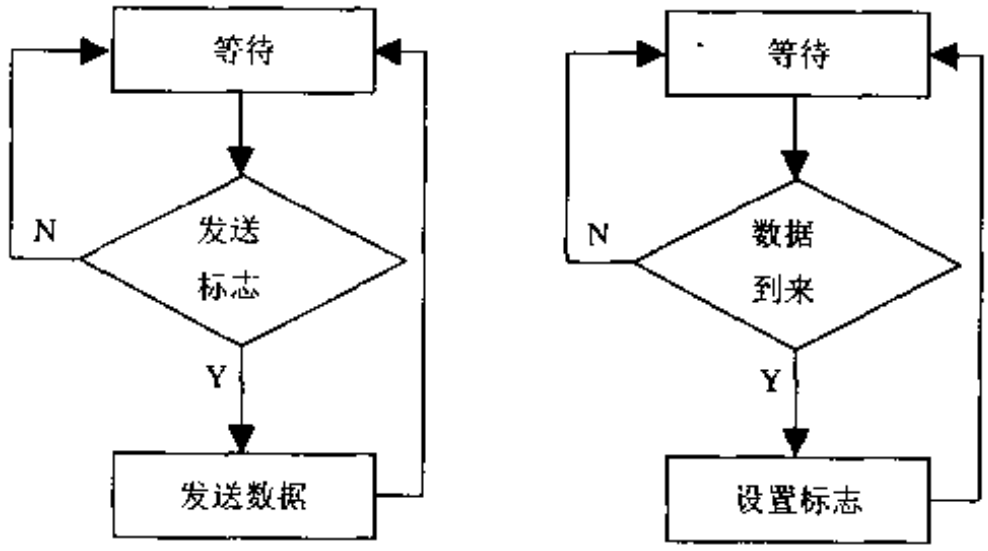



图 7-11　串口通信流程图

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。由于共有两个串口通信模块，因此分别对不同的串口模块采用不同的发送标志和数据到来标志。下面具体分析程序的代码。

初始化部分：该部分主要分别完成串口的初始化功能。以下为初始化部分的代码程序。

```
void Init_UART0(void)
{
    U0CTL = 0X00;     //将寄存器的内容清零
    U0CTL += CHAR;    //数据位为 8bit
```

```
    U0TCTL = 0X00;   //将寄存器的内容清零
    U0TCTL += SSEL1;//波特率发生器选择 SMCLK

    UBR0_0 = 0X45;   //波特率为 115200
    UBR1_0 = 0X00;

    UMCTL_0 = 0X49; //调整寄存器

    ME1 |= UTXE0 + URXE0; //使能 UART0 的 TXD 和 RXD
    IE1 |= URXIE0;   //使能 UART0 的 RX 中断
    IE1 |= UTXIE0;   //使能 UART0 的 TX 中断

    P3SEL |= BIT4;   //设置 P3.4 为 UART0 的 TXD
    P3SEL |= BIT5;   //设置 P3.5 为 UART0 的 RXD

    P3DIR |= BIT4;   //P3.4 为输出管脚
    return;
}
void Init_UART1(void)
{
    U1CTL = 0X00;    //将寄存器的内容清零
    U1CTL += CHAR;   //数据位为 8bit

    U1TCTL = 0X00;   //将寄存器的内容清零
    U1TCTL += SSEL1;//波特率发生器选择 SMCLK

    UBR0_1 = 0X45;   //波特率为 115200
    UBR1_1 = 0X00;
    UMCTL_1 = 0X00;

    ME2 |= UTXE1 + URXE1; //使能 UART1 的 TXD 和 RXD
    IE2 |= URXIE1;   //使能 UART1 的 RX 中断
    IE2 |= UTXIE1;   //使能 UART1 的 TX 中断

    P3SEL |= BIT6;   //设置 P3.6 为 UART1 的 TXD
    P3SEL |= BIT7;   //设置 P3.7 为 UART1 的 RXD

    P3DIR |= BIT6;   //P3.6 为输出管脚
    return;
}
```

通过以上的程序可以看出，要使用串口资源，需要设置 P3SEL 寄存器的相应的位来使能 UART 功能，P3SEL 位 4 和位 5 来使能串口 0，P3SEL 位 6 和位 7 来使能串口 1。通过其 ME1 和 IE1 寄存器的相应的位来打开串口 0 的中断；通过其 ME2 和 IE2 寄存器的相应的位来打开串口 1 的中断。通过设置 U0TCTL 的相应位来设置串口 0 的通信的格式，比如 8 位传输，也

通过设置 U0TCTL 寄存器来实现串口 0 的时钟源的选择；设置寄存器 UBR0_0 和 UBR1_0 来设置串口 0 的通信的波特率，还可以通过设置 UMCTL_0 寄存器来实现串口 0 的波特率误差的调整。通过设置 U1TCTL 的相应位来设置串口 1 的通信的格式，比如 8 位传输，也通过设置 U1TCTL 寄存器来实现串口 1 的时钟源的选择；设置寄存器 UBR0_1 和 UBR1_1 来设置串口 1 的通信的波特率，还可以通过设置 UMCTL_1 寄存器来实现串口 1 的波特率误差的调整。

串口中断程序主要是发送和接收数据，下面为程序代码。

```
//////////////////////////////////////////
//处理来自串口 0 的接收中断
interrupt [UART0RX_VECTOR] void UART0_RX_ISR(void)
{
    UART0_RX_BUF[nRX0_Len_temp] = RXBUF0;    //接收来自的数据
    nRX0_Len_temp += 1;
    if(UART0_RX_BUF[nRX0_Len_temp - 1] == 13)
    {
        nRX0_Len = nRX0_Len_temp;
        nRev_UART0 = 1;
        nRX0_Len_temp = 0;
    }
}
//////////////////////////////////////////
//处理来自串口 0 的发送中断
interrupt [UART0TX_VECTOR] void UART0_TX_ISR(void)
{
    if(nTX0_Len != 0)
    {
     nTX0_Flag = 0;                    //表示缓冲区里的数据没有发送完
     TXBUF0 = UART0_TX_BUF[nSend_TX0];
     nSend_TX0 += 1;
     Delay_us(5);
     if(nSend_TX0 >= nTX0_Len)
     {
         nSend_TX0 = 0;
         nTX0_Len = 0;
         nTX0_Flag = 1;
     }
    }
}
```

上面的程序为串口 0 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，在接收到数据后设置一个标志来通知主程序。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，则进入发送中断服务程序，发送完数据后，发送中断程序等待下一次中断的到来。

```
//////////////////////////////////////////
//处理来自串口 1 的接收中断
interrupt [UART1RX_VECTOR] void UART1_RX_ISR(void)
```

```
{

    UART1_RX_BUF[nRX1_Len_temp] = RXBUF1;    //接收来自的数据

    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;
    }
}
/////////////////////////////////////////
//处理来自串口 1 的发送中断
interrupt [UART1TX_VECTOR] void UART1_TX_ISR(void)
{
    if(nTX1_Len != 0)
    {
     nTX1_Flag = 0;              //表示缓冲区里的数据没有发送完
     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，在接收到数据后设置一个标志来通知主程序。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，则进入发送中断服务程序，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，很好利用了单片机的资源。

MODEM 与单片机的通信是通过串口 0(UART0)实现的，由于单片机需要控制 MODEM 模块的工作状态或者数据传输，这样需要单片机向 MODEM 发送命令来实现的，单片机向 MODEM 发送的是 AT 命令，通过 AT 命令来实现对 MODEM 的控制和实现数据传输，在下面部分具体介绍 AT 命令。

### 7.3.2 AT 命令介绍

AT 命令是一套用于对 MODEM 控制的命令，通常以“AT”开头。单片机通过向 MODEM

发送 AT 命令来实现对 MODEM 的控制。不同的 AT 命令控制 MODEM 的不同动作。通常情况下，AT 命令以字母“AT”开头，以 ASCII 码为 13 的字符结尾。下面介绍几种常用的 AT 命令，下面的命令如果不做特殊说明，都是需要带 ASCII 码为 13 的字符结尾，如果不带结尾符的话，则会对那条命令说明。

**1．回显命令：ATEn**

默认值：1

n=0，关闭回显。

n=1，打开回显。

该命令主要是用于向MODEM发送命令后，MODEM在返回响应结果的时候是否带发送的命令。如果发送的命令为ATE0，则只返回结果，不返回上次发送的命令。如果发送的命令为ATE1，不仅返回结果，而且还返回上次发送的命令。

**2．结果码格式：ATVn**

默认值：1

n=0，以数字形式显示结果码。

n=1，以字符形式显示结果码。

该命令主要是用于向 MODEM 发送命令后，MODEM 在返回响应结果的时候响应结果的具体形式。如果发送的命令为 ATV0，则返回结果为数字形式。如果发送的命令为 ATV1，则返回的结果为字符形式。

**3．结果码控制：ATQn**

默认值：0

n=0，MODEM 返回结果码。

n=1，MODEM 不返回结果码。

该命令主要是用于向 MODEM 发送命令后，控制 MODEM 是否返回结果码。如果发送的命令为 ATQ0，则 MODEM 返回结果码。如果发送的命令为 ATQ1，则 MODEM 不返回结果码。

**4．挂机/摘机命令：ATHn**

默认值：0

n=0，挂机命令。

n=1，摘机命令。

该命令用于在有来话呼叫时摘机或者挂机。当有来话呼叫时，如果发送命令为ATH0，则对呼叫挂机；如果发送命令为ATH1时，则对呼叫摘机应答。

**5．手工应答命令：ATA**

该命令使得 MODEM 立即摘机，并等待来自远端 MODEM 的拨号呼叫和载波信号，试图应答呼叫，而不需要等待呼叫振铃信号。

**6．流量控制命令：&Kn**

默认值：3

n=0，禁止流控。

n=3，允许 RTS/CTS 流控。

n=4，允许 XON/XOFF 流控。

n=5，允许透明 XON/XOFF 流控。

n=6，允许 RTS/CTS 和 XON/XOFF 两种流控。

该命令主要用于进行传输数据的时候进行流量控制。在应用的时候主要是根据不同的应用场合选择不同的参数 n，从而实现相应的流量控制。

**7．拨号命令：ATD**

该命令使 MODEM 立即进入摘机状态，并拨出随后的电话号码，以试图建立连接。该命令分别可以接“T”或者“D”。当该命令后面接“T”时，表示采用音频拨号方式；当该命令后面接“D”时，表示采用脉冲拨号方式。例如，ATDT123 表示采用音频方式，呼叫号码为 123 的电话；ATDP123 表示用脉冲方式，呼叫号码为 123 的电话。

**8．S 寄存器操作命令：ATSn**

ATS0=0，非自动应答方式。

ATS0=2，振铃两次后自动应答。

该命令主要用于设置 MODEM 的应答方式，可以设置成自动应答或者非自动应答。当发送命令 ATS0=0 命令，则 MODEM 处于非自动应答方式，当有来话呼叫时，需要手工应答。如果发送命令为 ATS0=1 时，则 MODEM 处于自动应答方式，这个时候振铃一次后应答来话呼叫。命令 ATS0=n 中的 n 值可以取其他的值，比如 1、2 或 3。

**9．切换到在线命令状态：+++**

该命令使 MODEM 从数据传输状态切换到命令状态。如果 MODEM 处于数据传输阶段时，并且当数据传输完毕时，那么 MODEM 需要再次进入命令状态，这样才有可能实现下次的拨号呼叫等操作，从而实现下次的数据传输。MODEM 从数据传输阶段到命令状态是通过+++命令来实现的。该命令是直接发送“+++”，前面不要带“AT”，并且不需要带 ASCII 码为 13 的字符结尾。

以上的命令只是最普通的几个命令，使用以上几个命令就能完成基本的数据传输任务，如果需要用到复杂的控制，需要查看相应的 AT 命令规范。另外有的 AT 命令是可以结合在一起使用的，比如 ATE0V0Q0S0=1，该命令就表示振铃一次后自动应答，不回显发送的命令，并且以数字的形式返回结果码。

在了解了 AT 命令后，下面将介绍具体的通信实现。

### 7.3.3 通信的流程

单片机通过向 MODEM 发送不同的 AT 命令来控制 MODEM 的工作。由于采用 MODEM 传输数据是基于链路传输的，所以传输过程必须遵守一定的通信流程。一般来说通信的流程主要包括初始化 MODEM、拨号、数据传送和释放链路 4 步。为了让 MODEM 能够有序地工作，在这里设计了一个简单的状态机。图 7-12 为状态机的示意图。

由图 7-12 可以看出，整个 MODEM 通信过程包括初始化，拨号、数据传输和挂断线路 4

个部分组成。在初始化不成功就继续进行初始化工作，直到 MODEM 初始化成功为止。在初始化的过程中将 nBusy 标志设置为 1，表示正在进行初始化；将 COMMAND 设置成 0 表示是初始化阶段；如果初始化不成功，将 nBusy 设置成 0，重新再一次进行初始化。当初始化成功后，进入拨号阶段。进入拨号阶段后将 nBusy 标志设置为 1，表示正在进行拨号；将 COMMAND 设置成 1 表示是拨号和数据传输阶段阶段；如果拨号成功，则将 nDial 设置成 1，表示拨号成功，就进入数据传输阶段；如果拨号不成功，则将 nBusy 设置成 0，COMMAND 也设置成 0，进行重新拨号，直到拨号成功为止。在具体的应用中，可以设置拨号不成功的次数，当拨号不成功的次数达到事先设置的次数后，不再进行拨号处理，也不再继续下一步处理，这些操作主要根据具体的应用和使用的环境来决定。拨号成功后进行数据传输阶段，如果数据传输完成，则将 COMMAND 设置成 2，并发送+++命令，如果收到成功的响应，则进行下一步，如果不成功，则一直发送+++命令，直到成功为止。当发送+++命令成功后，这个时候 MODEM 从数据传输的阶段已经成功进入命令状态，这样就可以发送挂断线路命令。单片机通过发送 ATH0 来挂机，这个时候将 COMMAND 设置成 3，如果成功挂断线路，则进入等待状态，等待下一次传输的到来，如果没有挂断线路的话，则继续发送挂断线路的命令，直到成功挂断为止。整个通信流程包括 MODEM 的操作部分和状态机的实现两个部分，下面具体分析程序的代码。

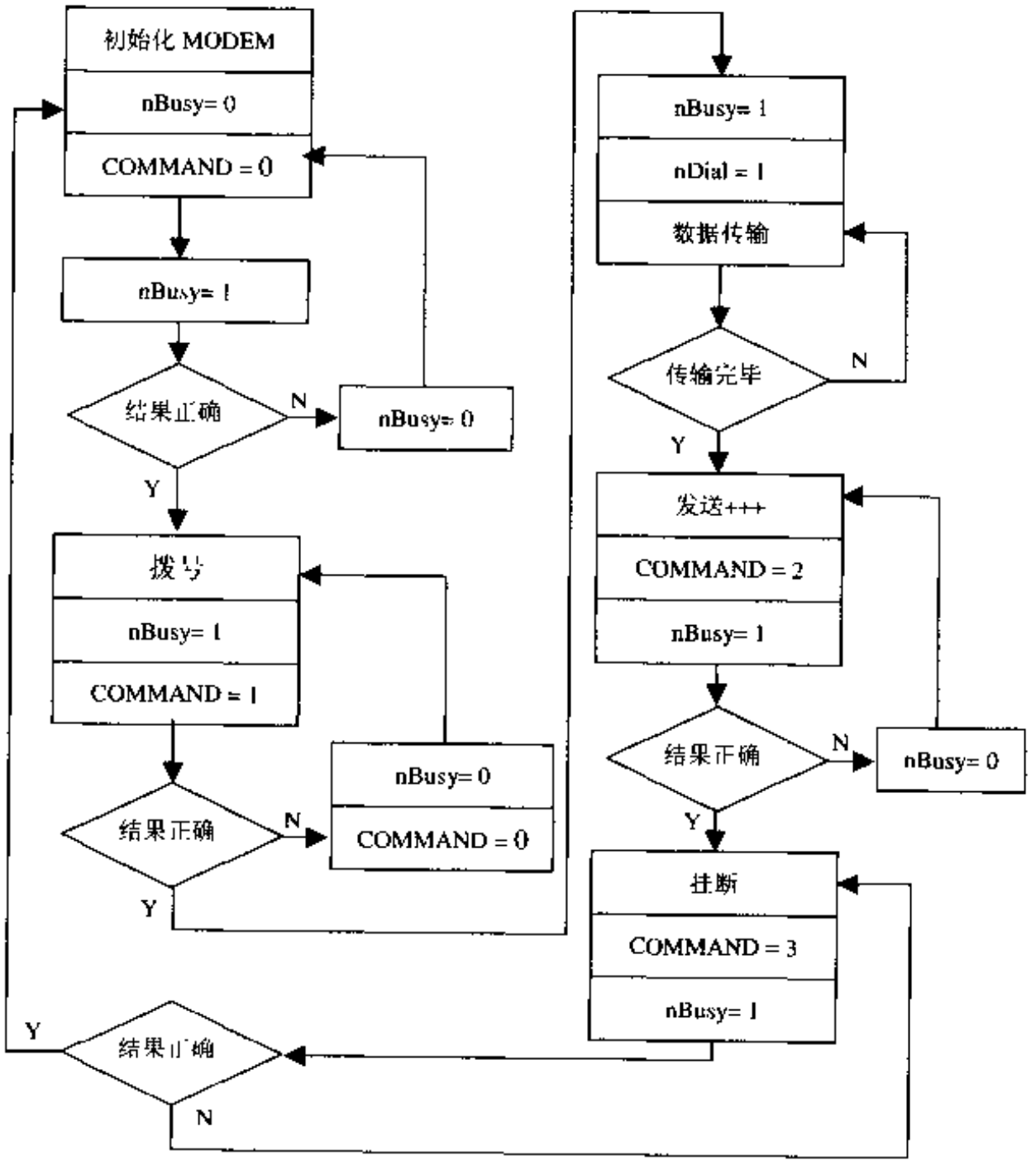



图 7-12　MODEM 通信状态机

### 1．MODEM的操作

MODEM的操作部分主要包括初始化、拨号、数据传输、状态切换和挂断线路几个部分。由于对MODEM的操作是单片机向MODEM发送AT命令来实现的，而数据的发送是通过串口来实现的，所以这里介绍的程序位于串口通信程序之上，这样对MODEM的相关操作主要是按照AT命令进行封装数据。具体的数据传输是由底层的串口通信程序来完成。下面介绍具体实现的代码。

MODEM初始化部分：该部分完成MODEM的初始化，使MODEM处于正确的工作模式。下面为该部分的程序代码。

```
//MODEM初始化设置
int ModemInit(char UART0_TX_BUF[])
{
    int nLen;
    //发送MODEM的初始化命令
    UART0_TX_BUF[0] = 'A';
    UART0_TX_BUF[1] = 'T';
    UART0_TX_BUF[2] = 'E';
    UART0_TX_BUF[3] = '0';
    UART0_TX_BUF[4] = 'V';
    UART0_TX_BUF[5] = '0';
    UART0_TX_BUF[6] = 'Q';
    UART0_TX_BUF[7] = '0';
    UART0_TX_BUF[8] = 'S';
    UART0_TX_BUF[9] = '0';
    UART0_TX_BUF[10] = '=';
    UART0_TX_BUF[11] = '1';

    UART0_TX_BUF[12] = 13;
    nLen = 13;
    return nLen;
}
```

该部分封装了用于初始化MODEM的AT命令：ATE0V0Q0S0=1，发送的数据为ASCII码，命令以ASCII值为13的字符结束。程序返回要向串口发送数据的长度。UART0_TX_BUF[]为串口发送数据的缓冲区。

拨号部分：该部分封装电话号码，实现拨号功能。下面为具体的程序代码。

```
//封装电话号码
int SetPhoneNum(char UART0_TX_BUF[],char nPhone,char PhoneNumber[])
{
    int nLen;
    char i;
    char chrTemp[8];
    //发送拨号命令
    UART0_TX_BUF[0] = 'A';
```

```
    UART0_TX_BUF[1] = 'T';
    UART0_TX_BUF[2] = 'D';
    UART0_TX_BUF[3] = 'T';

    if(nPhone == 1)
    {
        for(i = 0;i < 8;i++) chrTemp[i] = PhoneNumber[i];
        for(i = 0;i < 8;i++) UART0_TX_BUF[4 + i] = chrTemp[i];
        UART0_TX_BUF[12] = 13;

        nLen = 13;
    }
    else if(nPhone == 0)
    {
        for(i = 0;i < 7;i++) chrTemp[i] = PhoneNumber[i];
        for(i = 0;i < 7;i++) UART0_TX_BUF[4 + i] = chrTemp[i];
        UART0_TX_BUF[11] = 13;
        nLen = 12;
    }
    return nLen;
}
```

该程序通过 nPhone 来区分电话号码是 7 位还是 8 位，这里没有考虑区号，即这里是针对本地使用的应用领域。考虑到有的城市电话号码是 7 位，有的城市的电话号码是 8 位，所以使用 nPhone 来区分，如果 nPhone 为 1 的话，电话号码是 7 位，否则为 8 位。程序返回要向串口发送数据的长度。UART0_TX_BUF[]为串口发送数据的缓冲区。

数据传输部分：该部分主要是对需要发送的数据进行封装，按照一定的格式发送。具体的程序代码如下：

```
//封装需要发送的数据
int PackData(char UART0_TX_BUF[],char pBuf[],int nSendLen)
{
    char n;
    int nLen;

    UART0_TX_BUF[0] = 0xaa;
    UART0_TX_BUF[1] = 0xaa;

    for(n = 0;n < nSendLen;n++)
    {
        UART0_TX_BUF[n + 2] = pBuf[n];
    }

    UART0_TX_BUF[nSendLen + 2] = 0xaa;
    UART0_TX_BUF[nSendLen + 3] = 13;
```

```
    nLen = nSendLen + 4;

    return nLen;
}
```

程序封装了头（0xaa 和 0xaa），也封装了尾（0xaa 和 ASCII 值为 13 的字符）。程序返回要向串口发送数据的长度。UART0_TX_BUF[]为串口发送数据的缓冲区。

状态切换：该部分主要用于将 MODEM 从数据传输阶段切换到命令状态。具体的程序代码如下：

```
//发送状态切换命令+++
int DataToCommand(char UART0_TX_BUF[])
{
    int nLen;
    //发送状态切换命令+++
    UART0_TX_BUF[0] = '+';
    UART0_TX_BUF[1] = '+';
    UART0_TX_BUF[2] = '+';
    nLen = 3;

    return nLen;
}
```

该部分程序主要封装+++命令，程序返回要向串口发送数据的长度。UART0_TX_BUF[]为串口发送数据的缓冲区。

挂断线路：该部分主要用于将 MODEM 从线路上挂断。具体的程序代码如下：

```
int SetATH0(char UART0_TX_BUF[])
{
    int nLen;
    //发送挂断 MODEM 命令
    UART0_TX_BUF[0] = 'A';
    UART0_TX_BUF[1] = 'T';
    UART0_TX_BUF[2] = 'H';
    UART0_TX_BUF[3] = '0';
    UART0_TX_BUF[4] = 13;
    nLen = 5;

    return nLen;
}
```

该部分程序主要封装 ATH0 命令，程序返回要向串口发送数据的长度。UART0_TX_BUF[]为串口发送数据的缓冲区。

### 2. 状态机的实现

该部分主要包括状态机的实现和 MODEM 响应命令解析部分，下面就具体的实现分别介绍各自的程序代码。

响应命令解析部分：该部分完成 MODEM 接收命令后返回的命令响应。具体的程序代码如下：

```
int ProcessUART0(char *pBuf,int nLen)
{
    int nTemp = -1;
    int i;

    //处理拨号后的响应
    for(i = 0;i < nLen;i++)
    {
        if(pBuf[i] == 0x37 && pBuf[i + 1] == 0x39)
        {
            nTemp = 1;
            break;
        }
    }
    //处理初始化后的响应和发送+++以及 ATH0 的响应
    if(pBuf[0] == 0x30 && pBuf[1] == 13)
        nTemp = 4;
    //处理拨号后的响应
    if(pBuf[0] == 0x31 && pBuf[1] == 0x32 && pBuf[2] == 13)
        nTemp = 1;
    //处理是否已经在拨
    if(pBuf[0] == 0x33 && pBuf[1] == 13)
        nTemp = 3;
    //处理拨号后的响应
    if(pBuf[0] == 0x33 && pBuf[1] == 0x30 && pBuf[2] == 13)
        nTemp = 1;
    //处理拨号后的响应
    if(pBuf[0] == 0x35 && pBuf[1] == 0x30 && pBuf[2] == 0x20
       && pBuf[3] == 0x20 && pBuf[4] == 0x37 && pBuf[5] == 0x39
       && pBuf[6] == 13)
     nTemp = 1;
    if(pBuf[0] == 0x35 && pBuf[1] == 0x31 && pBuf[2] == 0x20
       && pBuf[3] == 0x20 && pBuf[4] == 0x37 && pBuf[5] == 0x39
       && pBuf[6] == 13)
     nTemp = 1;
    if(pBuf[0] == 0x32 && pBuf[1] == 0x30 && pBuf[2] == 0x20
       && pBuf[3] == 0x20 && pBuf[4] == 0x37 && pBuf[5] == 0x39
       && pBuf[6] == 13)
     nTemp = 1;
    //没有接电话线的情况
    if(pBuf[0] == 0x36 && pBuf[1] == 13)
     nTemp = 5;
```

```
    //对方挂断的情况
    if(pBuf[0] == 0x37 && pBuf[1] == 13)
     nTemp = 6;
    //2表示发送+++，3表示收到+++响应
    if(pBuf[0] == 13)
     nTemp = 7;

    return nTemp;
}
```

该部分程序是针对响应码里面没有命令回显、结果码为数字形式的命令进行解析的，返回的值用来作为状态机的参考值。该部分程序主要针对了初始化、拨号、状态切换和线路挂断几个命令的响应进行处理的。

状态机的实现部分：该部分主要是根据图7-12的状态机实现整个通信过程。具体的程序代码如下：

```
for(;;)
{
//首先初始化MODEM
        if(nComm == 1 && nComm_Command == 0 && nBusy == 0)
        {
//将p1.0设置成高电平
            P2OUT |= BIT0;
        for(i = 0;i < 3;i++)
        {
            nTX0_Len = ModemInit(UART0_TX_BUF);

            nBusy = 1;
            IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序
            Delay_ms(500);
        }
}//if(nComm == 1 && nComm_Command == 0 && nBusy == 0)

//拨号
if(nComm == 1 && nDial == 0 && nComm_Command == 1 && nBusy == 0)
{
    nBusy = 1;

nTX0_Len = SetPhoneNum(UART0_TX_BUF,nPhone,PhoneNumber);
    IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序

}//if(nComm == 1 && nDial == 0 && nComm_Command == 1 && nBusy == 0)
//往MODEM传送数据
if(nComm == 1 && nDial == 1)
{

    for(i = 0;i < 50;i++)
```

```
    {
      for(n = 0;n < 100;n++)
      {
          pBuf0[n] = i;
      }

      while(1) //等待缓冲区里的数据发送完毕
      {
          if(nTX0_Flag == 1) break;
      }
      nTX0_Len = PackData(UART0_TX_BUF,pBuf0,100);
     IFG1 |= UTXIFG0;//设置中断标志，进入发送中断程序
    }
  //数据已经发送完毕
  Delay_ms(100);
  //发送+++命令，从传送数据状态到命令状态
      nTX0_Len = DataToCommand(UART0_TX_BUF);

      nComm_Command = 2;
      nDial = 0;
      IFG1 |= UTXIFG0;      //设置中断标志，进入发送中断程序

  }//if(nComm == 1 && nDial == 1)

  //发送+++命令，从传送数据状态到命令状态
  if(nComm_Command == 2)
  {
      Delay_ms(1000);
      nTX0_Len = DataToCommand(UART0_TX_BUF);
      IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序
      Delay_ms(1000);
  }//if(nComm_Command == 2)
  //发送挂断命令
  if(nComm_Command == 3)
  {
      Delay_ms(1000);
      nTX0_Len = SetATH0(UART0_TX_BUF);
      IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序
      Delay_ms(1000);
  }//if(nComm_Command == 3)
  //处理来自 MODEM 的响应
  if(nRev_UART0 == 1)
  {
      nRev_UART0 = 0;

      for(i = 0;i < nRX0_Len;i++) UART0_RX_Temp[i] = UART0_RX_BUF[i];
      nRes_UART0 = ProcessUART0(UART0_RX_Temp,nRX0_Len,0);
```

```
nRX0_Len = 0;

switch(nRes_UART0)
{
case 0:
{
    break;
}
case 1:
{
    nDial = 1;
    nBusy = 1;
    break;
}
case 2:
{
    break;
}
case 3:
{
    if(nComm_Command == 3)
    {
        nBusy = 0;
     nDial = 0;
     nComm_Command = 0;
    }
    nBusy = 0;
    nBusy = 0;
    break;
}
case 4:
{
    if(nComm_Command == 0)
    {
      //0 表示发送初始化命令，1 表示初始化完毕
     nComm_Command = 1;
     nBusy = 0;
    }
    if(nComm_Command == 2)
    {
      //2 表示发送+++，3 表示收到+++响应
      nComm_Command = 3;
      //数据已经发送完毕
      Delay_ms(1000);
      //发送 ATH0 命令，挂断线路
      nTX0_Len = SetATH0(UART0_TX_BUF);
      //设置中断标志，进入发送中断程序
```

```
        IFG1 |= UTXIFG0;
              Delay_ms(1000);
              break;
        }
        if(nComm_Command == 3)
        {
        nBusy = 0;
        nDial = 0;
        nComm_Command = 0;
        nComm = 0;
        //停止给 MODEM 供电
        for(i = 0;i < 5;i++) P2OUT &= ~(BIT0);
        for(i = 300;i > 0;i--) ;  //延迟一点时间

        break;
        }
}
case 5: //没有接电话线的情况
{
    nBusy = 0;
    nDial = 0;
    nComm_Command = 1;
    break;
}
case 6: //对方挂断的情况
{
    nBusy = 0;
    nDial = 0;
    nComm_Command = 1;
    break;
}
case 7:
{
    //2 表示发送+++，3 表示收到+++响应
    nComm_Command = 3;
    //数据已经发送完毕
    Delay_ms(1000);
    //发送 ATH0 命令，挂断线路
    nTX0_Len = SetATH0(UART0_TX_BUF);
    //设置中断标志，进入发送中断程序
       IFG1 |= UTXIFG0;
        Delay_ms(1000);
        break;
}
case 255:
{
```

```
                nBusy = 0;
                break;
            }
            case -1:
            {
                nBusy = 0;
                break;
            }
            default:
                    break;
            }
        }//if(nRev_UART0 == 1)
    }
```

上面的程序按照状态机实现。单片机通过串口0向MODEM发送命令，这样在需要发送数据的时候，需要设置中断标志“IFG1 |= UTXIFG0;”进入中断，同时需要设置发送长度“nTX0_Len”来确定发送数据的长度，发送的数据放在UART0_TX_BUF[]缓冲区里。单片机通过串口0接收到数据后经过解析后，将得到状态交给主要的程序进行处理。整个处理过程按照图7-12的状态机实现。整个这部分代码需要与中断服务程序进行数据交互，这里实现数据交互是通过全局的标志变量和全局的数据缓冲区来实现的，具体的变量参看具体的程序。

### 7.3.4 系统软件流程

经过上面对AT命令和各个通信流程的介绍，对整个软件系统的实现有一定的了解。现在具体讨论整个系统软件的实现。

整个软件系统主要实现MODEM的数据通信，另外还通过与上位机的通信来实现一些简单的配置管理。经过前面对通信流程的介绍，对MODEM的通信过程应该很了解，整个系统软件在通信流程的基础上实现一个简单的通信系统，包括一些辅助的配置管理，整个程序的流程图如图7-13所示。

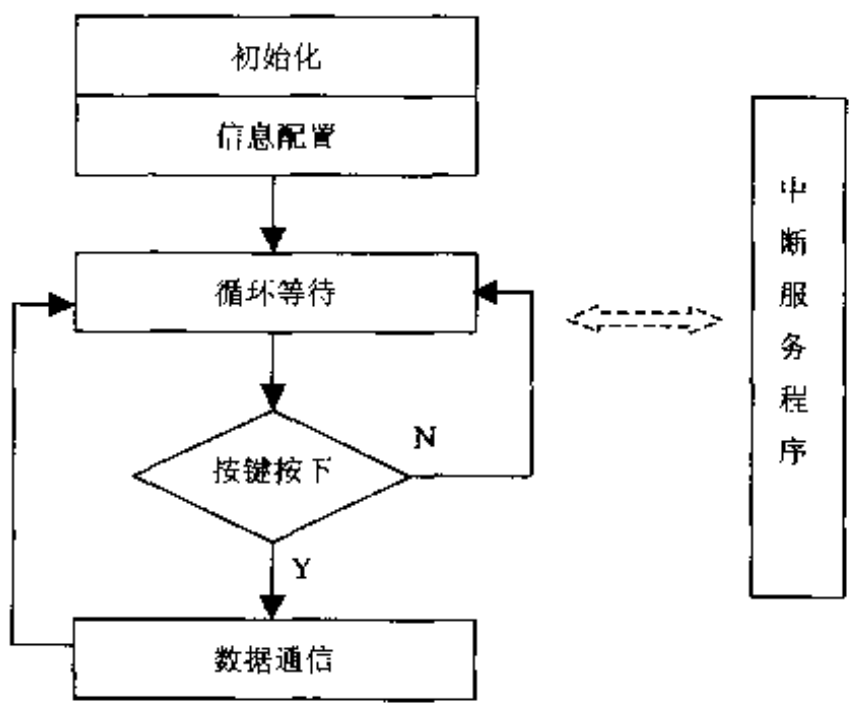



图7-13 系统软件流程图

由图 7-13 可以看出，整个系统软件基于中断服务机制，主处理需要与中断进行数据交互，具体的实现是通过全局的标志变量和全局的数据缓冲区来实现的。图 7-13 中的数据通信部分是按照上面介绍的状态机进行处理的。

整个软件系统包括上位机配置部分、按键处理部分、状态机实现部分和主处理部分。其中状态机实现部分在前面一小节已经做了具体的介绍，这里再进行详细的介绍，下面就其他各个部分进行具体介绍。

**1．上位机配置部分**

上位机配置部分主要是通过单片机的串口实现与上位机的通信，上位机与单片机的通信格式是按照自定义的 AT 命令来实现的，这样可以便于将来软件的扩展。在单片机处理的时候将 AT 指令做成一张表，将不同的命令放到相应的位置，在处理来自上位机的命令的时候只需要查找相应的表，这样做表的目的是可以提高搜索的速度。表的具体作成是采用 4 个关键字（即长度和前面 3 个字母）。为了保证指令的唯一性，在定义指令的时候必须满足长度和前面 3 个字母是唯一的。表的内容存储在单片机片内的 FLASH 里面。这样上位机配置部分主要包括表的作成、命令处理和 FLASH 操作等部分。下面就各个部分具体介绍其实现的程序代码。

表的作成：该部分主要是将一些命令分配到具体的位置。这样作成的表具有一定的伸缩性，可以根据需要增加相应的命令，只是在增加命令的时候必须满足命令在表里的唯一性。下面见具体的程序。

```
void Init_Process(void)
{
    ////////////////////////////////////////
    //为了提高算法的搜索效率，这里采利用 AT
    //命令的长度和 3 个关键字符来识别 AT 命令
    int *pAtLen;
    int *pAtBuf1;
    int *pAtBuf2;
    int *pAtBuf3;

    pAtLen = (int*)AT_LEN;
    FLASH_clr(pAtLen);
    pAtBuf2 = (int*)AT_BUFA2;
    FLASH_clr(pAtBuf2);
    pAtLen = (int*)AT_LEN;
    pAtBuf1 = (int*)AT_BUFA1;
    pAtBuf2 = (int*)AT_BUFA2;
    pAtBuf3 = (int*)AT_BUFA3;
    //0AT 命令的长度，检查通信链路正常命令
    FLASH_ww(pAtLen,3);
    FLASH_ww(pAtBuf1,'A');
    FLASH_ww(pAtBuf2,'T');
    FLASH_ww(pAtBuf3,13);
```

```
//1 读出设置的号码 AT+PHONENUM=?(enter)
pAtLen += 1;
pAtBuf1 += 1;
pAtBuf2 += 1;
pAtBuf3 += 1;
FLASH_ww(pAtLen,14);
FLASH_ww(pAtBuf1,'P');
FLASH_ww(pAtBuf2,'H');
FLASH_ww(pAtBuf3,'O');

//2AT+PHONENUM=Xxxxxxxxx(Enter)
pAtLen += 1;
pAtBuf1 += 1;
pAtBuf2 += 1;
pAtBuf3 += 1;
FLASH_ww(pAtLen,22);
FLASH_ww(pAtBuf1,'P');
FLASH_ww(pAtBuf2,'H');
FLASH_ww(pAtBuf3,'O');

return;
    return;
}
```

该程序依次写入 4 个字的内容，4 个字构成了一条 AT 命令，本部分程序只是简单地写了 3 条命令，用户可以根据自己的需要进行增加相应的命令。

命令处理：该部分是解析来自上位机的 AT 命令，主要是根据介绍到的数据搜索做成的表，如果搜索到满足条件的命令，则返回表中的位置，如果没有搜索到的话，则返回一个错误的指示。下面为该部分实现的代码。

```
int ProcessUART1(char *pBuf,int nLen)
{
    int *pAtLen;
    int *pAtBuf1;
    int *pAtBuf2;
    int *pAtBuf3;
    int nTemp = -1;
    int i;

    if(nLen <= 2) return -1;
    if(nLen == 3)
    {
    if((pBuf[0] == 'A') && (pBuf[1] == 'T') && (pBuf[2] == 13))
        return 0;
    }
```

```
    pAtLen = (int*)AT_LEN;
    pAtBuf1 = (int*)AT_BUFA1;
    pAtBuf2 = (int*)AT_BUFA2;
    pAtBuf3 = (int*)AT_BUFA3;
    for(i = 0;i < 3;i++)
    {
       nATlen = *(pAtLen + i);
     if(nLen == nATlen)
     {
        nBuf1 = *(pAtBuf1 + i);
        nBuf2 = *(pAtBuf2 + i);
        nBuf3 = *(pAtBuf3 + i);
        if((pBuf[3] == nBuf1) && (pBuf[4] == nBuf2) && (pBuf[5] == nBuf3))
        {
         nTemp = i;
        }
     }
    }
    return nTemp;
}
```

该部分程序主要是完成表的搜索，如果有满足要求的话则返回在表中的位置，否则返回“-1”作为错误的指示。

FLASH 操作部分：该部分主要完成 FLASH 的读写和擦除操作。具体的程序如下：

```
void FLASH_ww(int *pData,int nValue)
{
    FCTL3 = 0xA500;//LOCK = 0;
    FCTL1 = 0xA540;//WRT = 1;

    *pData = nValue;
}
void FLASH_wb(char *pData,char nValue)
{
    FCTL3 = 0xA500;//LOCK = 0;
    FCTL1 = 0xA540;//WRT = 1;

    *pData = nValue;
}
void FLASH_clr(int *pData)
{
    FCTL1 = 0xA502;//ERASE = 1;
    FCTL3 = 0xA500;//LOCK = 0;
```

```
    *pData = 0;
}
```

**2．按键处理部分**

该部分程序主要是处理按键。当如果有按键按下的时候，设置标志通知主程序。由于P1.5管脚可以设置成中断方式，因此这里采用中断服务程序的机制，当有按键按下的时候，则进入中断服务程序，中断服务程序设置一个标志通知主程序有按键按下，以便主程序进行通信处理。下面是中断服务程序的程序代码。

```
//////////////////////////////////////
//处理来自端口 1 的中断
interrupt [PORT1_VECTOR] void COMM_ISR(void)
{

    if(P1IFG & BIT5)
    {
     nComm = 1;

     nComm_Command = 0;
     P1IFG &= ~(BIT5);   //清除中断标志位
     Delay_us(100);
    }
}
```

在中断服务程序中，设置“nComm = 1”来通知主程序有按键按下。设置“nComm_Command = 0”使通信流程进入MODEM初始化状态。

**3．主处理部分**

主处理模块主要是将各个模块进行协调处理和实现数据交互。主处理模块首先完成初始化工作，初始化后读取配置信息，如果读到有配置信息，则进入处理循环，如果没有读到配置信息，则等待配置信息，直到配置完后进入处理循环。在循环过程中主处理扫描是否有按键按下，如果有按键按下，则进行通信流程，如果没有按键按下则继续循环。另外主程序还需要与两个串口的发送和接收中断服务程序进行数据交互。整个主程序基于中断服务结构，为了实现中断程序与主程序之间的数据交互，通过设置一些全局变量和全局的缓冲区来实现。下面给出程序的具体实现。

```
#include <MSP430X14X.h>
#include "modem.h"
#include "uart.h"
#include "process.h"
#include "const.h"

//定义全局变量
static char nComm;
int *pFlash;
//定义串口操作变量
```

```
char nRev_UART0;     //串口 0 的接收标志
char nRev_UART1;     //串口 1 的接收标志
char UART0_TX_BUF[112]; //串口 0 的发送缓冲区
char UART0_RX_BUF[20];  //串口 0 的接收缓冲区
char UART1_TX_BUF[112]; //串口 1 的发送缓冲区
char UART1_RX_BUF[50];  //串口 1 的接收缓冲区
char pBuf0[100];
static int nTX1_Len;
static char nRX1_Len;
char nRX1_Len_temp;
static int nTX0_Len;
static int nRX0_Len;
int nRX0_Len_temp;
static char nTX0_Flag;
static char nTX1_Flag;
int nSend_TX0;
int nSend_TX1;
static char nComm_Command;
char nBusy;
char nDial;
void main(void)
{

    int j;
    int n;
    int nTemp;
    char nRes_UART1;
    char nRes_UART0;
    char PhoneNumber[8];
    char UART1_RX_Temp[50];
    char UART0_RX_Temp[30];
    char nPhoneOK;
    int nPhone;

    WDTCTL = WDTPW + WDTHOLD;     //关闭看门狗

    _DINT();                      //关闭中断

    nBusy = 0;
    nDial = 0;
    nComm_Command = 0;
    nSend_TX1 = 0;
    nSend_TX0 = 0;
    nTX1_Flag = 0;
    nTX0_Flag = 0;
```

```
    nTX0_Len = 0;
    nTX1_Len = 0;
    nRX1_Len = 0;
    nRX0_Len = 0;
    nRev_UART1 = 0;
    nRev_UART0 = 0;
    nComm = 0;
    ////////////////////////////////
    //初始化
    Init_CLK();
    Init_Process();
    Init_UART0();
    Init_UART1();
_EINT();                //打开中断
    //读配置信息
    nPhone = ReadPhoneNum(PhoneNumber);
    //等待配置信息
    if(nPhone == 2)
    {
     for(;;)
     {
        if(nRev_UART1 == 1)
        {
    nRev_UART1 = 0;
    for(i = 0;i < nRX1_Len;i++)
        UART1_RX_Temp[i] = UART1_RX_BUF[i];
    nRes_UART1 = ProcessUART1(UART1_RX_Temp,nRX1_Len);
    switch(nRes_UART1)
    {
        case 0:
        {
          SetOK(UART1_TX_BUF,&nTX1_Len);
          nRX1_Len = 0;
               IFG2 |= UTXIFG1; //设置中断标志，进入发送中断程序
          break;
        }
        case 1:
        {
          QueryPhone(UART1_TX_BUF,&nTX1_Len);
          nRX1_Len = 0;
               IFG2 |= UTXIFG1; //设置中断标志，进入发送中断程序
          break;
        }
        case 2:
        {
```

```
            nTemp = 0;
            nTemp = SetPhone(UART1_RX_Temp);
            if(nTemp == 1)
            {
                SetOK(UART1_TX_BUF,&nTX1_Len);

            nPhone = (char)(UART1_RX_Temp[12] - 0x30);
            for(i = 0;i < 8;i++)
                PhoneNumber[i] = UART1_RX_Temp[13 + i];

            nPhoneOK = 1;
            }
            else
            {
                SetError(UART1_TX_BUF,&nTX1_Len);
            }
            nRX1_Len = 0;
                IFG2 |= UTXIFG1; //设置中断标志，进入发送中断程序
            break;
        }
        default:break;
    }
        }
        if(nPhoneOK == 1) break;
    }

}
for(;;)
{

    //首先初始化 MODEM
    if(nComm == 1 && nComm_Command == 0 && nBusy == 0)
    {
        //将 p1.5 设置成高电平
        P2OUT |= BIT0;
        for(i = 0;i < 3;i++)
        {
            nTX0_Len = ModemInit(UART0_TX_BUF);

            nBusy = 1;
            IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序
            Delay_ms(500);
        }
    }//if(nComm == 1 && nComm_Command == 0 && nBusy == 0)
```

```
//拨号
if(nComm == 1 && nDial == 0 && nComm_Command == 1 && nBusy == 0)
{
    nBusy = 1;

 nTX0_Len = SetPhoneNum(UART0_TX_BUF,nPhone,PhoneNumber);
    IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序

}//if(nComm == 1 && nDial == 0 && nComm_Command == 1 && nBusy == 0)
//往MODEM传送数据
if(nComm == 1 && nDial == 1)
{

    for(i = 0;i < 50;i++)
    {
      for(n = 0;n < 100;n++)
      {
          pBuf0[n] = i;
      }

      while(1) //等待缓冲区里的数据发送完毕
      {
          if(nTX0_Flag == 1) break;
      }
      nTX0_Len = PackData(UART0_TX_BUF,pBuf0,100);
      IFG1 |= UTXIFG0;//设置中断标志，进入发送中断程序
    }
 //数据已经发送完毕
 Delay_ms(100);
 //发送+++命令，从传送数据状态到命令状态
    nTX0_Len = DataToCommand(UART0_TX_BUF);

    nComm_Command = 2;
    nDial = 0;
    IFG1 |= UTXIFG0;      //设置中断标志，进入发送中断程序

}//if(nComm == 1 && nDial == 1)

//发送+++命令，从传送数据状态到命令状态
if(nComm_Command == 2)
{
    Delay_ms(1000);
    nTX0_Len = DataToCommand(UART0_TX_BUF);
    IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序
    Delay_ms(1000);
```

```
}//if(nComm_Command == 2)
//发送挂断命令
if(nComm_Command == 3)
{
    Delay_ms(1000);
    nTX0_Len = SetATH0(UART0_TX_BUF);
    IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序
    Delay_ms(1000);
}//if(nComm_Command == 3)
//处理来自 MODEM 的响应
if(nRev_UART0 == 1)
{
    nRev_UART0 = 0;

    for(i = 0;i < nRX0_Len;i++) UART0_RX_Temp[i] = UART0_RX_BUF[i];
    nRes_UART0 = ProcessUART0(UART0_RX_Temp,nRX0_Len,0);
    nRX0_Len = 0;

    switch(nRes_UART0)
    {
     case 0:
     {
        break;
     }
    case 1:
    {
        nDial = 1;
        nBusy = 1;
        break;
    }
    case 2:
    {
        break;
    }
    case 3:
    {
        if(nComm_Command == 3)
        {
            nBusy = 0;
         nDial = 0;
         nComm_Command = 0;
        }
        nBusy = 0;
        nBusy = 0;
        break;
```

```
    }
    case 4:
    {
        if(nComm_Command == 0)
        {
          //0 表示发送初始化命令，1 表示初始化完毕
         nComm_Command = 1;
         nBusy = 0;
        }
        if(nComm_Command == 2)
        {
          //2 表示发送+++，3 表示收到+++响应
          nComm_Command = 3;
          //数据已经发送完毕
          Delay_ms(1000);
          //发送 ATH0 命令，挂断线路
          nTX0_Len = SetATH0(UART0_TX_BUF);
          //设置中断标志，进入发送中断程序
          IFG1 |= UTXIFG0;
          Delay_ms(1000);
          break;
        }
        if(nComm_Command == 3)
        {
         nBusy = 0;
         nDial = 0;
         nComm_Command = 0;
         nComm = 0;
         //停止给 MODEM 供电
         for(i = 0;i < 5;i++) P2OUT &= ~(BIT0);
         for(i = 300;i > 0;i--) ;  //延迟一点时间

         break;
        }
    }
    case 5: //没有接电话线的情况
    {
        nBusy = 0;
        nDial = 0;
        nComm_Command = 1;
        break;
    }
    case 6: //对方挂断的情况
    {
        nBusy = 0;
```

```
            nDial = 0;
            nComm_Command = 1;
            break;
        }
        case 7:
        {
            //2 表示发送+++，3 表示收到+++响应
            nComm_Command = 3;
            //数据已经发送完毕
            Delay_ms(1000);
            //发送 ATH0 命令，挂断线路
            nTX0_Len = SetATH0(UART0_TX_BUF);
            //设置中断标志，进入发送中断程序
               IFG1 |= UTXIFG0;
               Delay_ms(1000);
               break;
        }
        case 255:
        {
            nBusy = 0;
            break;
        }
        case -1:
        {
            nBusy = 0;
            break;
        }
        default:
               break;
        }
    }//if(nRev_UART0 == 1)
  }
}
```

上面为主处理的程序，该部分为一个可伸缩性的框架，可以在此基础上进一步丰富处理的功能。主程序先进行全局变量的初始化，初始化一些函数，然后读取配置信息，如果没有配置信息的话，则需要等到上位机的配置，直到配置完成后才进入处理循环。在处理循环里面检测通信按键是否按下，如果按下就进入通信处理流程，如果没有按下则继续等待。主程序需要和串口通信程序进行数据交互。当需要向串口发送数据的时候，往发送的数据缓冲区里面封装数据，设置发送的数据长度，设置中断标志后就触发中断服务程序，中断服务程序就发送数据，直到发送完毕后清除相应的标志位。主程序还需要检测串口的接收中断服务程序，当串口有新数据到来时，串口接收中断服务程序将收到的数据填入到接收缓冲区里，设置接收到的长度，设置接收标志来通知主程序取数据。主程序与中断服务程序进行数据交互参看具体用到的全局标志变量和全局数据缓冲区。

## 7.4 系统调试

前面已经介绍了整个系统的硬件设计和软件设计，该部分结合前面介绍的来讨论整个系统的联调。系统的调试包括硬件调试和软件调试，下面分别进行介绍。

**1．系统硬件调试**

系统的硬件调试很简单，先调试电源电路和复位电路，只要这两个部分能正常工作，再进行单片机的调试，如果单片机的晶振能起振的话，则整个硬件的单片机部分没有问题。关于硬件系统的串口通信和 MODEM 通信部分则需要结合软件进行调试。

**2．系统软件调试**

前面分别介绍了各个软件部分，为了能够进行整个系统的联调，需要软件和硬件调试结合起来，对于不同的硬件部分，应该调用不同的软件模块进行测试。经过联合调试，整个系统的软件和硬件能够正确运行。

## 7.5 实例总结

该系统通过 MODEM 实现数据传输的系统具有设计简单、运行可靠等特点。通过软件测试和硬件测试证明该系统能够安全可靠的运行。本系统采用 MSP430F149 实现的数据传输系统具有一定的通用性，它通过一个嵌入式的 MODEM 模块与单片机的 UART 进行连接实现数据传输。由于系统可以通过上位机进行配置，这样使该系统在使用上有很大的灵活性。该系统具有很大的伸缩性，单片机与上位机通信只需要扩充通信的 AT 命令就可以实现更加丰富的功能，另外该系统也可以和前面介绍的数据采集系统结合起来组成一个数据采集传输系统。虽然本章介绍的系统相对简单，但用户完全可以在该基础上进行扩展升级实现自己功能，实现更为完整的系统。为了能使读者全面了解程序的其他模块，附录给出了程序里调用的函数的程序源代码。

### 附录：其他程序模块

```
//Modem.c 文件
#include "Modem.h"

//MODEM 初始化设置
int ModemInit(char UART0_TX_BUF[])
{
    int nLen;
    //发送 MODEM 的初始化命令
    UART0_TX_BUF[0] = 'A';
    UART0_TX_BUF[1] = 'T';
    UART0_TX_BUF[2] = 'E';
    UART0_TX_BUF[3] = '0';
    UART0_TX_BUF[4] = 'V';
```

```
    UART0_TX_BUF[5] = '0';
    UART0_TX_BUF[6] = 'Q';
    UART0_TX_BUF[7] = '0';
    UART0_TX_BUF[8] = 'S';
    UART0_TX_BUF[9] = '0';
    UART0_TX_BUF[10] = '=';
    UART0_TX_BUF[11] = '1';

    UART0_TX_BUF[12] = 13;

    nLen = 13;
    return nLen;
}
//封装电话号码
int SetPhoneNum(char UART0_TX_BUF[],char nPhone,char PhoneNumber[])
{
    int nLen;
    char i;
    char chrTemp[8];
    //发送拨号命令
    UART0_TX_BUF[0] = 'A';
    UART0_TX_BUF[1] = 'T';
    UART0_TX_BUF[2] = 'D';
    UART0_TX_BUF[3] = 'T';

    if(nPhone == 1)
    {
        for(i = 0;i < 8;i++) chrTemp[i] = PhoneNumber[i];
        for(i = 0;i < 8;i++) UART0_TX_BUF[4 + i] = chrTemp[i];
        UART0_TX_BUF[12] = 13;

        nLen = 13;
    }
    else if(nPhone == 0)
    {
        for(i = 0;i < 7;i++) chrTemp[i] = PhoneNumber[i];
        for(i = 0;i < 7;i++) UART0_TX_BUF[4 + i] = chrTemp[i];
        UART0_TX_BUF[11] = 13;

        nLen = 12;
    }

    return nLen;
}
//封装需要发送的数据
```

```
int PackData(char UART0_TX_BUF[],char pBuf[],int nSendLen)
{
    char n;
    int nLen;

    UART0_TX_BUF[0] = 0xaa;
    UART0_TX_BUF[1] = 0xaa;

    for(n = 0;n < nSendLen;n++)
    {
        UART0_TX_BUF[n + 2] = pBuf[n];
    }

    UART0_TX_BUF[nSendLen + 2] = 0xaa;
    UART0_TX_BUF[nSendLen + 3] = 13;

    nLen = nSendLen + 4;

    return nLen;
}
//Process.c 文件
#include "Process.h"
#include "flash.h"
#include "const.h"



int nATlen = 0;
int nBuf1 = 0;
int nBuf2 = 0;
int nBuf3 = 0;
void Init_Process(void)
{
    ////////////////////////////////////////
    //为了提高算法的搜索效率，这里采利用 AT
    //命令的长度和 3 个关键字符来识别 AT 命令
    int *pAtLen;
    int *pAtBuf1;
    int *pAtBuf2;
    int *pAtBuf3;

    pAtLen = (int*)AT_LEN;
    FLASH_clr(pAtLen);
    pAtBuf2 = (int*)AT_BUFA2;
    FLASH_clr(pAtBuf2);
```

```
pAtLen = (int*)AT_LEN;
pAtBuf1 = (int*)AT_BUFA1;
pAtBuf2 = (int*)AT_BUFA2;
pAtBuf3 = (int*)AT_BUFA3;
//0 AT 命令的长度,检查通信链路正常命令
FLASH_ww(pAtLen,3);
FLASH_ww(pAtBuf1,'A');
FLASH_ww(pAtBuf2,'T');
FLASH_ww(pAtBuf3,13);


//1 读出设置的号码 AT+PHONENUM=?(enter)
pAtLen += 1;
pAtBuf1 += 1;
pAtBuf2 += 1;
pAtBuf3 += 1;
FLASH_ww(pAtLen,14);
FLASH_ww(pAtBuf1,'P');
FLASH_ww(pAtBuf2,'H');
FLASH_ww(pAtBuf3,'O');

//2 AT+PHONENUM=xxxxxxxxx(Enter)
    pAtLen += 1;
    pAtBuf1 += 1;
    pAtBuf2 += 1;
    pAtBuf3 += 1;
    FLASH_ww(pAtLen,22);
    FLASH_ww(pAtBuf1,'P');
    FLASH_ww(pAtBuf2,'H');
    FLASH_ww(pAtBuf3,'O');

    return;
}
int ProcessUART0(char *pBuf,int nLen)
{
    int nTemp = -1;
    int i;


    //处理拨号后的响应
    for(i = 0;i < nLen;i++)
    {
        if(pBuf[i] == 0x37 && pBuf[i + 1] == 0x39)
        {
            nTemp = 1;
```

```
            break;
        }
    }
    //处理初始化后的响应和发送+++以及ATH0的响应
    if(pBuf[0] == 0x30 && pBuf[1] == 13)
        nTemp = 4;
    //处理拨号后的响应
    if(pBuf[0] == 0x31 && pBuf[1] == 0x32 && pBuf[2] == 13)
        nTemp = 1;
    //处理是否已经在拨
    if(pBuf[0] == 0x33 && pBuf[1] == 13)
        nTemp = 3;
    //处理拨号后的响应
    if(pBuf[0] == 0x33 && pBuf[1] == 0x30 && pBuf[2] == 13)
        nTemp = 1;
    //处理拨号后的响应
    if(pBuf[0] == 0x35 && pBuf[1] == 0x30 && pBuf[2] == 0x20
      && pBuf[3] == 0x20 && pBuf[4] == 0x37 && pBuf[5] == 0x39
      && pBuf[6] == 13)
     nTemp = 1;
    if(pBuf[0] == 0x35 && pBuf[1] == 0x31 && pBuf[2] == 0x20
      && pBuf[3] == 0x20 && pBuf[4] == 0x37 && pBuf[5] == 0x39
      && pBuf[6] == 13)
     nTemp = 1;
    if(pBuf[0] == 0x32 && pBuf[1] == 0x30 && pBuf[2] == 0x20
      && pBuf[3] == 0x20 && pBuf[4] == 0x37 && pBuf[5] == 0x39
      && pBuf[6] == 13)
     nTemp = 1;
    //没有接电话线的情况
    if(pBuf[0] == 0x36 && pBuf[1] == 13)
     nTemp = 5;

    //对方挂断的情况
    if(pBuf[0] == 0x37 && pBuf[1] == 13)
     nTemp = 6;
    //2表示发送+++，3表示收到+++响应
    if(pBuf[0] == 13)
     nTemp = 7;

    return nTemp;
}
////////////////////////////////////////
//正确返回各个命令对应的值，
//错误返回 -1
int ProcessUART1(char *pBuf,int nLen)
```

```
{
    int *pAtLen;
    int *pAtBuf1;
    int *pAtBuf2;
    int *pAtBuf3;
    int nTemp = -1;
    int i;

    if(nLen <= 2) return -1;
    if(nLen == 3)
    {
     if((pBuf[0] == 'A') && (pBuf[1] == 'T') && (pBuf[2] == 13))
        return 0;
    }


    pAtLen = (int*)AT_LEN;
    pAtBuf1 = (int*)AT_BUFA1;
    pAtBuf2 = (int*)AT_BUFA2;
    pAtBuf3 = (int*)AT_BUFA3;
    for(i = 0;i < 3;i++)
    {
        nATlen = *(pAtLen + i);
     if(nLen == nATlen)
     {
         nBuf1 = *(pAtBuf1 + i);
         nBuf2 = *(pAtBuf2 + i);
         nBuf3 = *(pAtBuf3 + i);
         if((pBuf[3] == nBuf1) && (pBuf[4] == nBuf2) && (pBuf[5] == nBuf3))
         {
          nTemp = i;
         }
     }
    }
    return nTemp;
}
//将结果由参数返回
char ReadPhoneNum(char strPhone[8])
{
    int *pFlash;
    int nTemp;
    int i;
    char chrRes;

    pFlash = (int*)PHONE_ADDR;
```

```
    //第一个数用来确定号码的位数
    chrRes = (char)(*pFlash - 0x30);
    //第 个电话号码
    pFlash += 1;
    nTemp = *pFlash;
    strPhone[0] = (char)(nTemp & 0x00ff);
    pFlash += 1;
    nTemp = *pFlash;
    strPhone[1] = (char)(nTemp & 0x00ff);
    pFlash += 1;
    nTemp = *pFlash;
    strPhone[2] = (char)(nTemp & 0x00ff);
    pFlash += 1;
    nTemp = *pFlash;
    strPhone[3] = (char)(nTemp & 0x00ff);
    pFlash += 1;
    nTemp = *pFlash;
    strPhone[4] = (char)(nTemp & 0x00ff);
    pFlash += 1;
    nTemp = *pFlash;
    strPhone[5] = (char)(nTemp & 0x00ff);
    pFlash += 1;
    nTemp = *pFlash;
    strPhone[6] = (char)(nTemp & 0x00ff);
    pFlash += 1;
    nTemp = *pFlash;
    strPhone[7] = (char)(nTemp & 0x00ff);
    for(i = 0;i < 8;i++)
    {
     if(strPhone[i] == 0xff)
     {
        chrRes = 2;
        break;
     }
    }

    return chrRes;
}
//正确返回0，错误返回1
char SetPhone(char UART1_RX_Temp[])
{
    int *pFlash;
```

```
pFlash = (int*)PHONE_ADDR;

FLASH_clr(pFlash);
//第一个电话号码
//写数据到 FLASH 里面
if(UART1_RX_Temp[12] >= 0x30 && UART1_RX_Temp[12] <= 0x39)
    FLASH_ww(pFlash,UART1_RX_Temp[12]);//1 或者 0
else
{
 FLASH_clr(pFlash);
 return 1;
}

pFlash += 1;
if(UART1_RX_Temp[13] >= 0x30 && UART1_RX_Temp[13] <= 0x39)
    FLASH_ww(pFlash,UART1_RX_Temp[13]);
else
{
 FLASH_clr(pFlash);
 return 1;
}

pFlash += 1;
if(UART1_RX_Temp[14] >= 0x30 && UART1_RX_Temp[14] <= 0x39)
    FLASH_ww(pFlash,UART1_RX_Temp[14]);
else
{
 FLASH_clr(pFlash);
 return 1;
}

pFlash += 1;
if(UART1_RX_Temp[15] >= 0x30 && UART1_RX_Temp[15] <= 0x39)
    FLASH_ww(pFlash,UART1_RX_Temp[15]);
else
{
 FLASH_clr(pFlash);
 return 1;
}

pFlash += 1;
if(UART1_RX_Temp[16] >= 0x30 && UART1_RX_Temp[16] <= 0x39)
    FLASH_ww(pFlash,UART1_RX_Temp[16]);
else
{
```

```
FLASH_clr(pFlash);
return 1;
}

pFlash += 1;
if(UART1_RX_Temp[17] >= 0x30 && UART1_RX_Temp[17] <= 0x39)
   FLASH_ww(pFlash,UART1_RX_Temp[17]);
else
{
FLASH_clr(pFlash);
return 1;
}

pFlash += 1;
if(UART1_RX_Temp[18] >= 0x30 && UART1_RX_Temp[18] <= 0x39)
   FLASH_ww(pFlash,UART1_RX_Temp[18]);
else
{
FLASH_clr(pFlash);
return 1;
}

   pFlash += 1;
   if(UART1_RX_Temp[19] >= 0x30 && UART1_RX_Temp[19] <= 0x39)
      FLASH_ww(pFlash,UART1_RX_Temp[19]);
   else
   {
    FLASH_clr(pFlash);
    return 1;
   }

   pFlash += 1;
   if(UART1_RX_Temp[20] >= 0x30 && UART1_RX_Temp[20] <= 0x39)
      FLASH_ww(pFlash,UART1_RX_Temp[20]);
   else
   {
    FLASH_clr(pFlash);
    return 1;
   }

   return 0;
}
void QueryPhone(char UART1_TX_BUF[],int *nLen)
{
   int *pFlash;
```

```
int nTemp;

pFlash = (int*)PHONE_ADDR;
UART1_TX_BUF[0] = '+';
UART1_TX_BUF[1] = 'P';
UART1_TX_BUF[2] = 'H';
UART1_TX_BUF[3] = 'O';
UART1_TX_BUF[4] = 'N';
UART1_TX_BUF[5] = 'E';
UART1_TX_BUF[6] = 'N';
UART1_TX_BUF[7] = 'U';
UART1_TX_BUF[8] = 'M';
UART1_TX_BUF[9] = '=';
//第一个电话号码
//从 FLASH 里面读出数据
nTemp = *pFlash;
UART1_TX_BUF[10] = (char)(nTemp & 0x00ff);
pFlash += 1;
nTemp = *pFlash;
UART1_TX_BUF[11] = (char)(nTemp & 0x00ff);
pFlash += 1;
nTemp = *pFlash;
UART1_TX_BUF[12] = (char)(nTemp & 0x00ff);
pFlash += 1;
nTemp = *pFlash;
UART1_TX_BUF[13] = (char)(nTemp & 0x00ff);
pFlash += 1;
nTemp = *pFlash;
UART1_TX_BUF[14] = (char)(nTemp & 0x00ff);
pFlash += 1;
nTemp = *pFlash;
UART1_TX_BUF[15] = (char)(nTemp & 0x00ff);
pFlash += 1;
nTemp = *pFlash;
UART1_TX_BUF[16] = (char)(nTemp & 0x00ff);
pFlash += 1;
nTemp = *pFlash;
UART1_TX_BUF[17] = (char)(nTemp & 0x00ff);
pFlash += 1;
nTemp = *pFlash;
UART1_TX_BUF[18] = (char)(nTemp & 0x00ff);

UART1_TX_BUF[19] = 13;

*nLen = 20;
```

```
}
void SetOK(char UART1_TX_BUF[],int *nLen)
{
    UART1_TX_BUF[0] = 'O';
    UART1_TX_BUF[1] = 'K';
    UART1_TX_BUF[2] = 13;

    *nLen = 3;
}
void SetError(char UART1_TX_BUF[],int *nLen)
{
    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;
    *nLen = 6;
}
```

# 第 8 章　大数据量本地存储系统设计

第 2 章介绍了 MSP430F1XX 的端口，本章在第 2 章的基础上介绍如何利用一般的 I/O 口来实现大数据量的本地存储系统的设计。一般说来，实现大数据量的存储可以采用硬盘或者 SmartMedia 卡的方式，本章介绍的是基于 SmartMedia 的方式。下面就大数据量本地存储系统的硬件设计和软件设计分别进行详细的介绍。

## 8.1　系统描述

在一些数据采集系统中，既需要将数据传送到相应的服务器，也需要将数据进行本地存储，进行本地存储的好处是可以将得到的数据进行本地再处理，也可以作为数据的备份。由于很多数据采集系统采集的数据量比较大，因此对系统的存储容量就有较高的要求，一般EPROM或者FLASH的容量达不到要求，而采用SmartMedia卡就能很好解决问题。SmartMedia卡具有以下特点：

- 容量大。一般目前的SmartMedia容量是64MB或者128MB，将来也许能达到更大的容量。
- 处理灵活。SmartMedia既可以通过单片机进行操作（读、写、擦除），也可以使用第三方提供的单独的SmartMedia读写器进行操作，给系统增加操作的灵活性，加上SmartMedia是采用座子的形式，SmartMedia可以实现热插拔，进一步增加系统处理的灵活性。
- 接口简单。SmartMedia卡采用的是地址线和数据线复用的方式，也就是通过往SmartMedia卡发不同的命令来完成相应的操作，并且它的地址数据也是通过发不同的字节数（不同容量的SmartMedia卡有不同字节的地址数）来完成寻址操作的。这样SmartMedia卡与单片机的接口非常简单，同时也满足相同的硬件系统支持不同容量的SmartMedia卡，而不必因为不同容量的SmartMedia卡要进行不同的硬件系统的设计。

基于以上特点，采用 SmartMedia 卡来作为存储介质。本章介绍的大数据量本地存储系统选用三星公司的 K9S1208V0M-SSB0 来作为存储器，K9S1208V0M-SSB0 是容量为 64MB 的 SmartMedia，该系统的单片机采用 MSP430F149，系统的设计主要针对大数据量采集存储的应用场合，也适合低功耗的应用场合。下面分别具体介绍系统的硬件设计和软件设计。

## 8.2　系统硬件设计

该系统的硬件设计主要包括电源设计、复位电路设计和单片机与SmartMedia卡接口设计。

由于MSP430F149没有数据总线，只能采用利用一般I/O口来模拟数据总线，MSP430F149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SmartMedia卡进行介绍。

### 8.2.1 SmartMedia 卡介绍

SmartMedia 是采用 NAND 技术实现的 FLASH，它为固态数据存储提供了一种有效的解决方案。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。SmartMedia 的框图如图 8-1 所示。

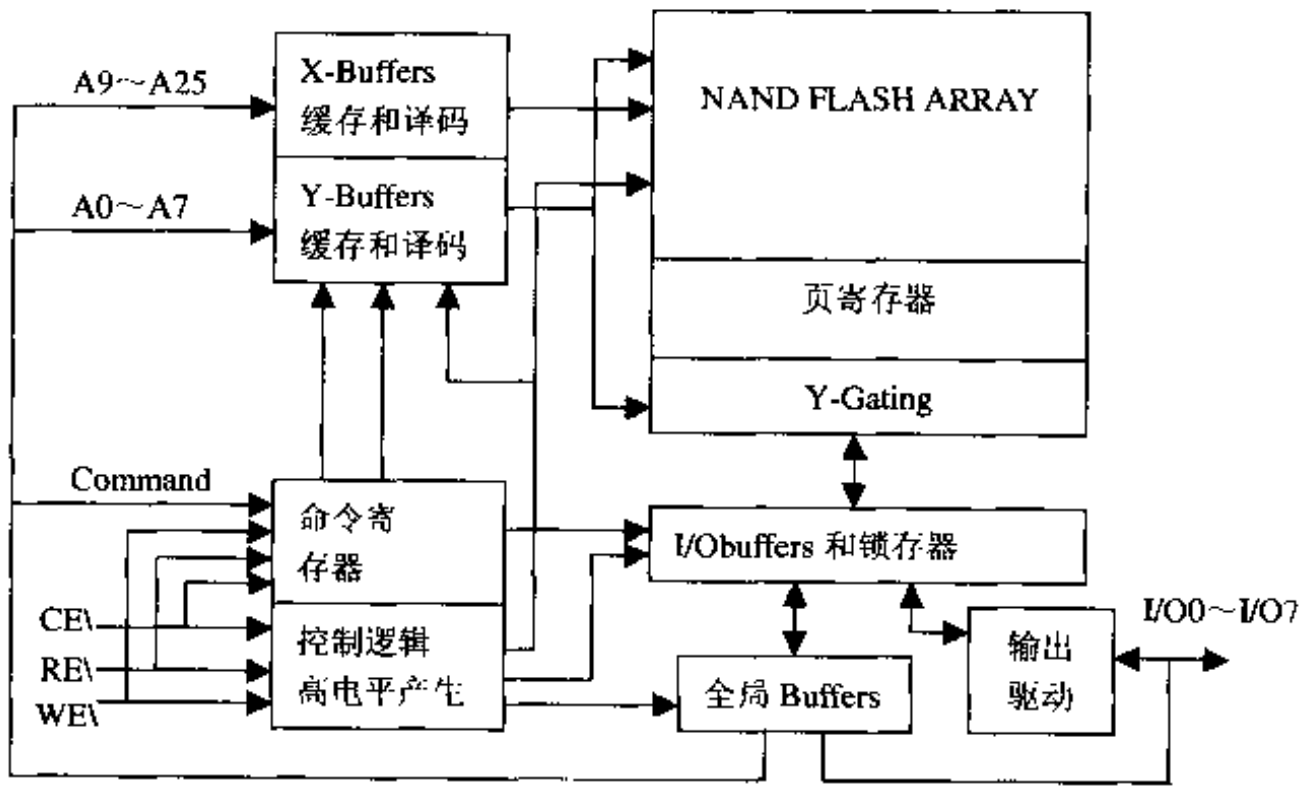



图 8-1　SmartMedia 的框图

由图8-1可以看出，SmartMedia主要有控制逻辑单元、缓存和译码单元、NAND FLASH存储阵列以及输出驱动几个部分组成。为了对SmartMedia有个清楚的认识，下面从SmartMedia管脚定义、存储阵列的组织方式、操作模式、操作命令、读操作、写操作和擦除操作等具体操作进行详细的介绍。

**1．SmartMedia 管脚定义**

SmartMedia 卡与一般芯片在封装上有所不同的是该卡不是采用管脚的形式，而是采用金手指的形式，采用这样的形式使 SmartMedia 卡在实际应用中能够非常方便地插拔，从而提高系统使用的灵活性。图 8-2 给出了 SmartMedia 卡的封装形式和管脚定义。

根据图 8-2 的管脚定义，下面具体对各个管脚进行说明：

- 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。

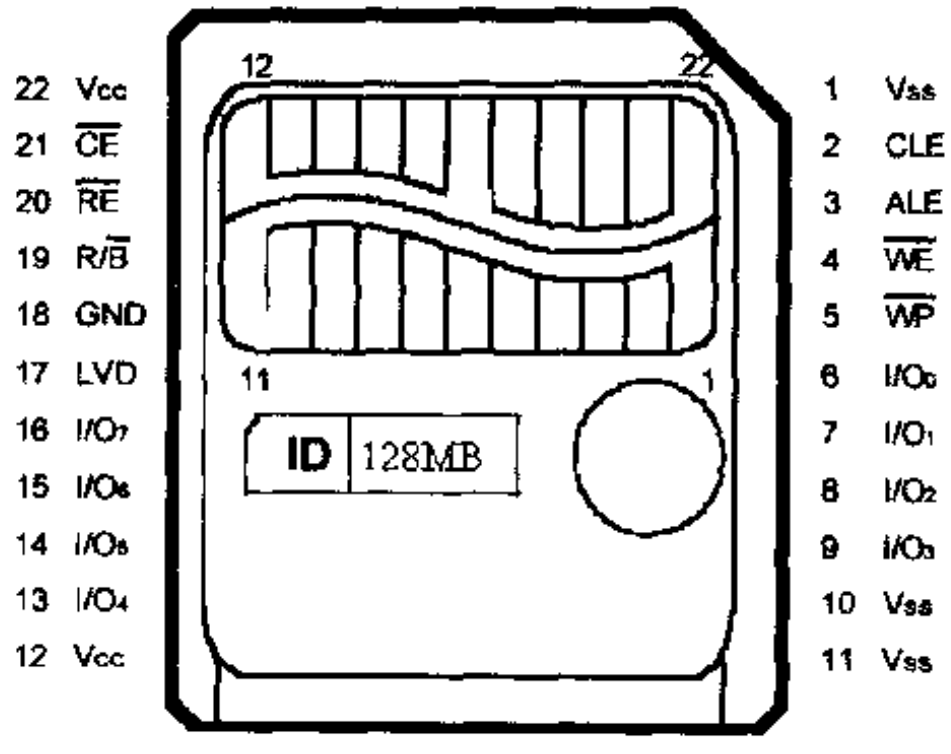



图 8-2　SmartMedia 卡的管脚定义

- CE：SmartMedia 卡选择管脚。该管脚低电平有效。当该管脚为低电平的时候，选通SmartMedia卡，否则SmartMedia 卡不工作。
- RE：SmartMedia卡读使能管脚。管脚低电平有效。当该管脚为低电平的时候，对SmartMedia卡进行读操作。
- WE：SmartMedia卡写使能管脚。管脚低电平有效。当该管脚为低电平的时候，对SmartMedia卡进行写操作。
- I/O口（I/O0～I/O7）：SmartMedia卡的数据线，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SmartMedia 卡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。
- LVD：低电压检测。该管脚用来检测供电电压，如果不用该管脚，该管脚悬空就可以了。

### 2．SmartMedia 的存储阵列的组织方式

由图 8-1 可以知道，SmartMedia 卡的地址分为行地址和列地址，SmartMedia 卡以字节为单位，这样 SmartMedia 卡的存储阵列可以看成一个三维模型。图 8-3 显示了 SmartMedia 卡的存储阵列的组织形式。

通过图8-3可以看出，SmartMedia卡由很多的页（Page）组成，其中32页组成一块（Block），这样整个SmartMedia卡可以看成很多的块组成。SmartMedia 卡的一页由3个区域组成，3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个字节组成，用来存放备注信息。SmartMedia卡通过列地址（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。SmartMedia卡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结合相应的命令就能实现对SmartMedia卡任意地址进行访问。

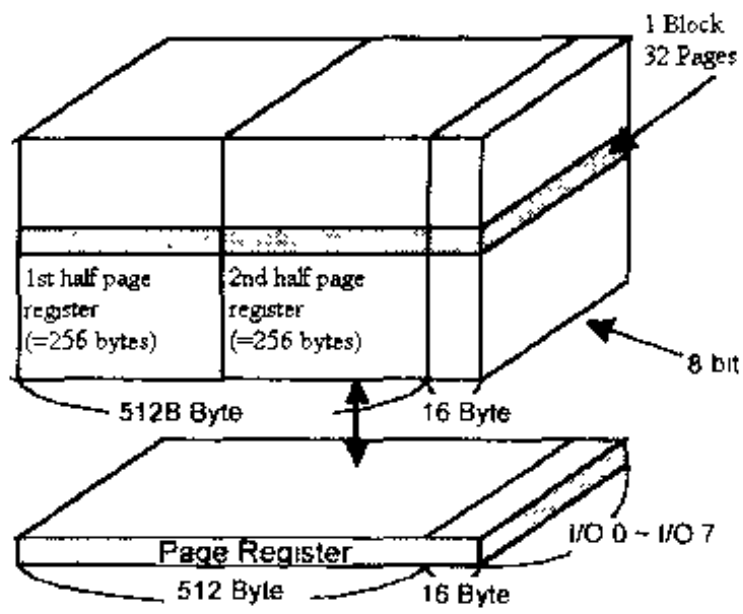



图 8-3　SmartMedia 卡的存储阵列组织形式

### 3．SmartMedia 卡的具体操作模式

SmartMedia 卡只有 8 根数据总线，然而 SmartMedia 卡却需要完成读、写和擦除等不同的操作，因此 SmartMedia 卡提供了不同的控制线，借助这些不同的控制线可以使 SmartMedia 卡有不同的操作模式，SmartMedia 卡有写模式、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 8-1 给出了 SmartMedia 卡的几种工作模式下的控制线的输入情况。

表 8-1　SmartMedia 卡的模式选择

| 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 |
| 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/$V_{CC}$ | 停止工作 |

表 8-1 中的 H 表示逻辑高电平，L 表示逻辑低电平，X 表示输入什么没有关系。通过表 8-1 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。

### 4．SmartMedia 卡的操作命令

对 SmartMedia 卡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 SmartMedia 卡的操作是通过向 SmartMedia 卡发送不同的命令来实现不同的操作的。表 8-2 给出了 SmartMedia 卡操作的全部命令集，除了表中例出来的命令外，其余所有的命令都是非法的，SmartMedia 卡不能识别那些表中以外的命令。

表 8-2　SmartMedia 卡的操作命令

| 功能 | 第一周期 | 第二周期 | 功能 | 第一周期 | 第二周期 |
|---|---|---|---|---|---|
| 读命令 1 | 00/01 | | 多块编程命令 | 0x80 | 0x15 |
| 读命令 2 | 0x50 | | 块擦除命令 | 0x60 | 0xd0 |
| 读 ID 命令 | 0x90 | | 多块擦除命令 | 0x60…0x60 | 0xd0 |
| 复位命令 | 0xff | | 读状态命令 | 0x70 | |
| 页写命令 | 0x80 | 0x10 | 读多块状态命令 | 0x71 | |

表 8-2 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 表示操作页的备注区。将通过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。

### 5．读操作

通过前面对操作模式选择和操作命令的介绍，已经对 SmartMedia 卡的操作有了大概的认识，现结合操作模式选择和操作命令来具体分析读操作命令。读操作命令包括 3 个部分：发送命令码、发送地址数据和接收内容数据。图 8-4 给出了读操作命令的时序图。

由图 8-4 可以看出，首先使 CE 为低电平，让 SmartMedia 卡处于工作状态，使 CLE 为高电平，ALE 为低电平，SmartMedia 卡处于命令状态，这个时候总线上输入读数的操作命令（00 或者 01），输入命令的时候必须给出写时钟信号，输入完命令码后，使 ALE 为高电平，CLE 为低电平，SmartMedia 卡处于发送地址数据状态，结合在写管脚产生写时钟信号，在总线上输入地址数据（共 4 个字节），在输入完地址数据后，SmartMedia 卡会在 R/B 管脚上输出一个低电平脉冲，表示进入输出数据状态，将 CLE 和 ALE 处于低电平，并在 RE 管脚不断产生读信号，这样就在总线上不断输出数据。总线在读操作命令时输出数据的长度和具体的读操作命令是有关的，单字节读操作只输出一个数据，多字节读操作输出多个数据，页读写操作输出 528 个字节的数据。

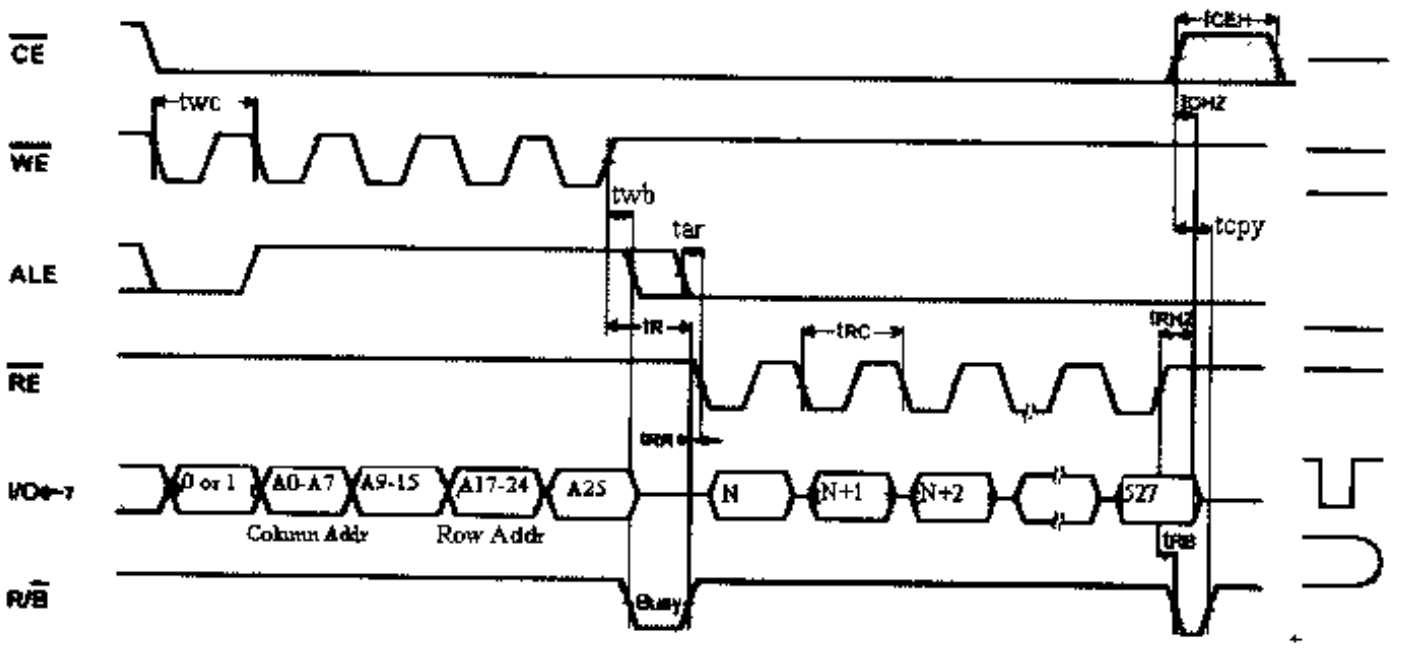



图 8-4　SmartMedia 卡的读操作时序

### 6. 写操作

SmartMedia 卡的写操作和读操作基本上略有不同，写操作包括 5 个部分：发送命令码、发送地址数据、写入内容数据、写确认命令和写状态读取。图 8-5 给出了写操作命令的时序图。

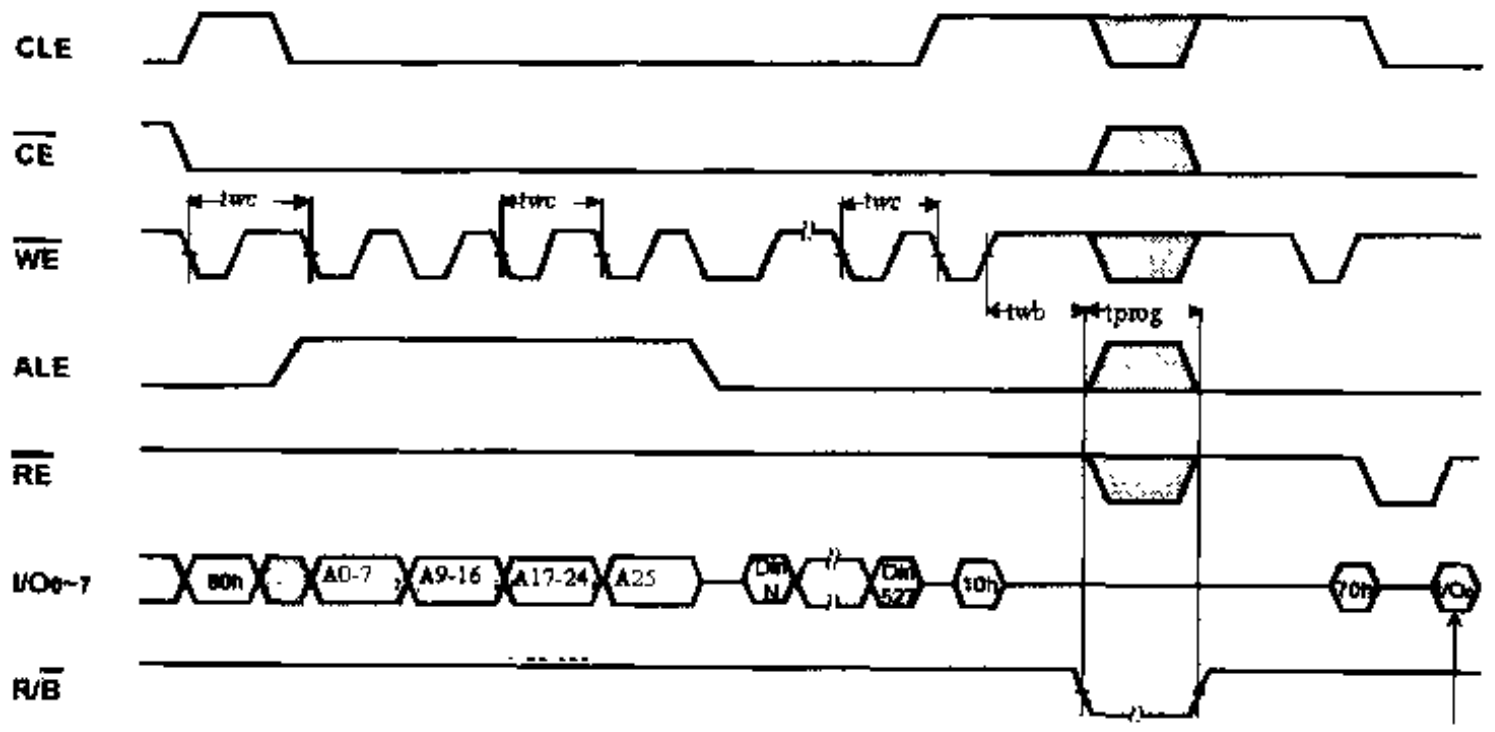



图 8-5　SmartMedia 卡的写操作时序

由图8-5可以看出，首先使CE为低电平，让SmartMedia卡处于工作状态，使CLE为高电平，ALE为低电平，SmartMedia卡处于命令状态，这个时候总线上输入写数的操作命令（0x80），输入命令的时候必须给出写时钟信号，输入完命令码后，使ALE为高电平，CLE为低电平，SmartMedia卡处于发送地址数据状态，结合在写管脚产生写时钟信号，在总线上输入地址数据（共4个字节），在输入完地址数据后，将CLE和ALE处于低电平，在WE管脚不断产生写信号，这样就在总线上不断写入数据。总线在写操作命令时写入数据的长度和具体的写操作命令是有关的，单字节写操作只写入一个数据，多字节写操作写入多个数据，页写操作写入528个字节的数据。在完成写入内容数据后，延时一定时间后将CLE设置为高电平，ALE为低电平，使SmartMedia卡处于命令状态，结合在写管脚产生写时钟信号，这个时候发送写确认命令码（0x10），对写操作进行确认，延迟一段时间后，在R/B管脚上输出一个低电平脉冲，表示写入数据完成，SmartMedia卡会再发送读取状态信息命令（0x70），这时需要将CLE设置为高电平，ALE设置为低电平，并在WE管脚上产生写时钟，发完读状态寄存器命令后，将ALE和CLE设置为低电平，并在RE管脚给出读信号，则在总线上读取状态寄存器的数据，通过从总线上读取的状态寄存器的数据来判断写入操作是否成功。

### 7. 擦除操作

为了能够对 SmartMedia 卡进行正确写数据，必须满足的条件是即将写入的单元的内容必须是 0xFF，SmartMedia 卡提供了擦除操作，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。擦除操作包括 4 个部分：发送命令码、发送块地址数据、擦除确认命令和状态读取。图 8-6 给出了擦除操作命令的时序图。

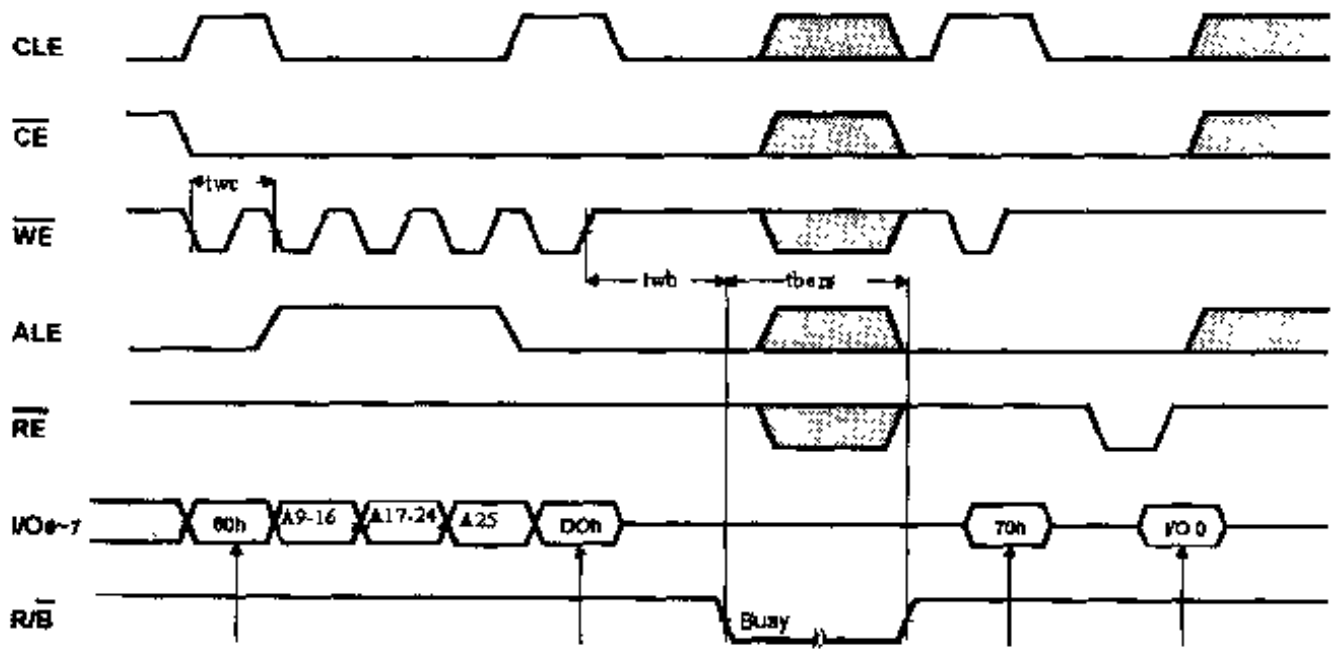



图 8-6　SmartMedia 卡的擦除操作时序

由图8-6可以看出，首先使CE为低电平，让SmartMedia卡处于工作状态，使CLE为高电平，ALE为低电平，SmartMedia卡处于命令状态，这个时候总线上输入擦除的操作命令（0x60），输入命令的时候必须给出写时钟信号，输入完命令码后，使ALE为高电平，CLE为低电平，SmartMedia 卡处于发送地址数据状态，结合在写管脚产生写时钟信号，在总线上输入地址数据（共3个字节），这里与前面读和写不同的是，擦除操作是针对块为单位，因此输入的是块地址，这样地址字节总共只有3个字节。在输入完地址数据后，需要将CLE设置为高电平，ALE为低电平，使SmartMedia卡处于命令状态，结合在写管脚产生写时钟信号，发送写确认命令码（0xd0），完成擦除操作后，SmartMedia 卡会在R/B管脚上输出一个低电平脉冲，表示擦除操作完成，延时一定时间后再发送读取状态信息命令（0x70），这时需要将CLE设置为高电平，ALE设置为低电平，并在WE管脚上产生写时钟，发完读状态寄存器命令后，将ALE和CLE设置为低电平，并在RE管脚给出读信号，则在总线上读取状态寄存器的数据，通过从总线上读取的状态寄存器的数据来判断擦除操作是否成功。

经过这部分对 SmartMedia 卡的具体介绍，相信对 SmartMedia 卡已经有了深刻的认识，下面就系统的硬件进行具体的介绍。

## 8.2.2　系统硬件接口设计

整个系统的硬件相对简单，接口设计也非常容易。为了对整个硬件系统有全面的认识，该部分不仅仅分析接口电路，也介绍它的其他部分电路。系统的硬件接口主要包括电源电路、复位电路和 SmartMedia 卡与 MSP430F149 的接口设计 3 个部分。下面就各个部分进行具体分析。

### 1．电源电路

整个系统采用 3.3V 供电，考虑到硬件系统对电源要求具有稳压功能和纹波小等特点，另外也考虑到硬件系统的低功耗等特点，因此该硬件系统的电源部分采用 TI 公司的 TPS76033 芯片实现，该芯片能很好满足该硬件系统的要求，另外该芯片具有很小的封装，因此能有效节约 PCB 板的面积。电源电路具体如图 8-7 所示。

为了使输出电源的纹波小，在输出部分用了一个 2.2μF 和 0.1μF 的电容，另外在芯片的输入端也放置一个 0.1μF 的滤波电容，减小输入端受到的干扰。

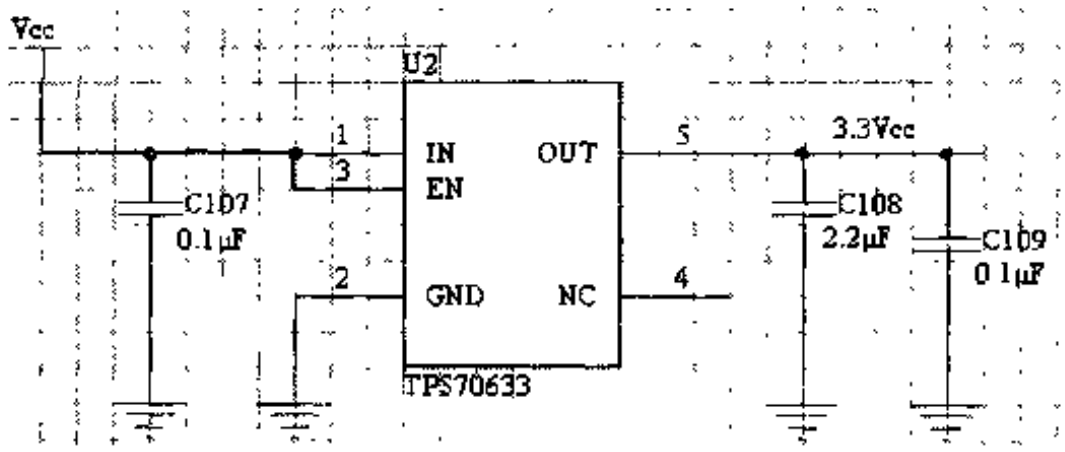



图 8-7　电源电路

### 2．复位电路

在单片机系统里，单片机需要复位电路，复位电路可以采用 R-C 复位电路，也可以采用复位芯片实现的复位电路，R-C 复位电路具有经济性，但可靠性不高，用复位芯片实现的复位电路具有很高的可靠性，因此为了保证复位电路的可靠性，该系统采用复位芯片实现的复位电路，该系统采用 MAX809 芯片。复位电路如图 8-8 所示。

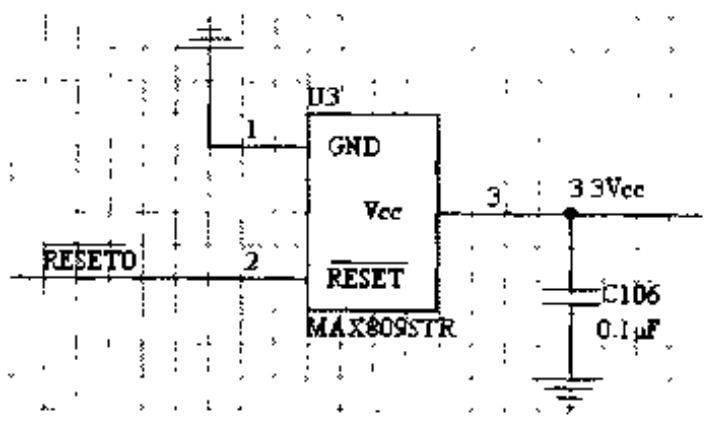



图 8-8　复位电路

为了减小电源的干扰，还需要在复位芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 3．SmartMedia 卡与 MSP430F149 的接口设计

由于 MSP430F149 没有数据总线，因此利用 MSP430F149 的一般 I/O 口来模拟总线。MSP430F149 能通过端口的方向寄存器来设置端口的输入输出方向，因此能很好的完成总线的数据读写功能。MSP430F149 的一般 I/O 口与 SmartMedia 卡的相应控制线接口，完成相应的控制功能。图 8-9 给出了 MSP430F149 与 SmartMedia 卡的接口。

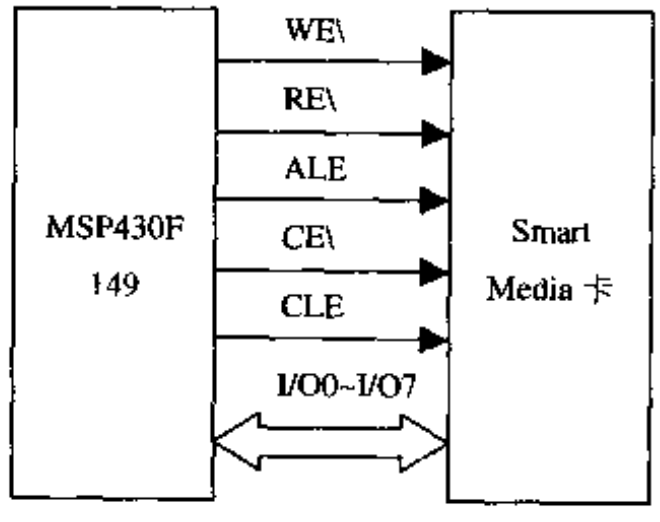



图 8-9　MSP430F149 与 SmartMedia 卡的接口

从图 8-9 可以看出：MSP430F149 与 SmartMedia 卡的接口非常简单。将 MSP430F149 的 P5 口与数据总线相连，P4 口与一些控制线相连。

通过上面对 SmartMedia 卡和系统硬件设计的介绍，对本地大数据量存储系统有了全面的认识，下面将根据系统的硬件设计介绍来系统的软件设计。

## 8.3　系统软件设计

根据前面的硬件设计介绍知道，单片机是通过一般 I/O 口来实现与 SmartMedia 卡进行接口的，因此软件必须要能实现单片机的端口操作。由于 MSP430F149 没有数据总线，因此系统软件还必须通过一般 I/O 口来模拟总线，并根据 SmartMedia 卡的不同操作来模拟不同的时序。系统的软件主要包括以下部分：控制线模拟、读操作、写操作和擦除操作。下面就各个部分进行具体的介绍。

### 8.3.1　控制线模拟

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MSP43F149的一般I/O口也可以作为第二功能腿使用，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．ALE 控制线的模拟**

该部分主要作用是为控制信号线ALE设置正确的输入输出方向，产生高电平或者低电平。具体的程序如下：

```
void ALE_Enable(void)
```

```
{
    P4OUT |= BIT2; //产生高电平
    return;
}
void ALE_Disable(void)
{
    P4OUT &= ~(BIT2); //产生低电平
    return;
}
```

**3. CLE 控制线的模拟**

该部分主要作用是为控制信号线 CLE 设置正确的输入输出方向，产生高电平或者低电平。具体的程序如下：

```
void CLE_Enable(void)
{
    P4OUT |= BIT0; //产生高电平

    return;
}
void CLE_Disable(void)
{
    P4OUT &= ~(BIT0); //产生低电平
    return;
}
```

**4. WE\ 控制线的模拟**

该部分主要作用是为控制信号线 WE\ 设置正确的输入输出方向，产生高电平或者低电平。具体的程序如下：

```
void WE_Enable(void)
{
    P4OUT &= ~(BIT4); //产生低电平

    return;
}
void WE_Disable(void)
{
    P4OUT |= BIT4; //产生高电平

    return;
}
```

**5. RE\ 控制线的模拟**

该部分主要作用是为控制信号线 RE\设置正确的输入输出方向，产生高电平或者低电平。具体的程序如下：

```
void RE_Enable(void)
{
    P4OUT &= ~(BIT3); //产生低电平

    return;
}
void RE_Disable(void)
{
    P4OUT |= BIT3; //产生高电平

    return;
}
```

**6. CE\ 控制线的模拟**

该部分主要作用是为控制信号线 CE\ 设置正确的输入输出方向，产生高电平或者低电平。具体的程序如下：

```
void CE_Enable(void)
{
    P4OUT &= ~(BIT1); //产生低电平

    return;
}
void CE_Disable(void)
{
    P4OUT |= BIT1; //产生高电平
    return;
}
```

## 8.3.2　读操作

根据前面对SmartMedia卡读操作时序的介绍，只要通过MSP430F149正确模拟SmartMedia卡的读操作时序就能完成读操作。图8-10给出了读操作的流程图。

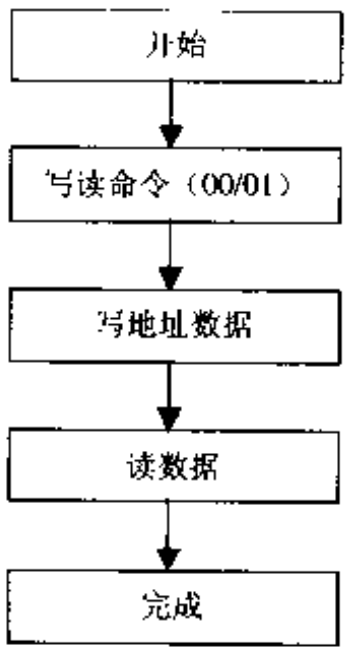



图 8-10　SmartMedia 读操作流程图

根据图 8-10 给出的流程图，下面给出具体的程序：

```
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，读也可以按照字节进行读，与页不同的是，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还是备注区。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，后面的附录里给出来详细的程序代码。

### 8.3.3　写操作

根据前面对SmartMedia卡写操作时序的介绍，只要通过MSP430F149正确模拟SmartMedia卡的写操作时序就能完成读操作。图8-11给出了写操作的流程图。

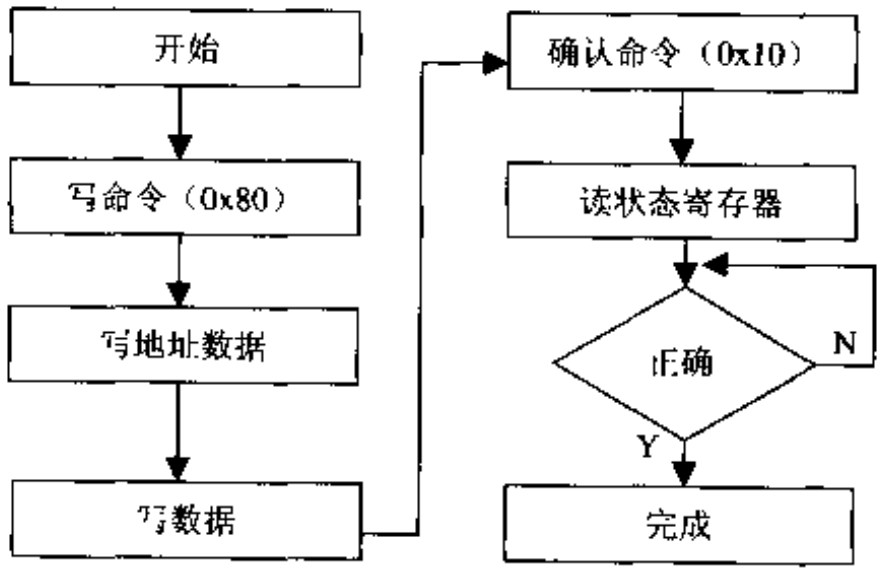



图 8-11　SmartMedia 写操作流程图

根据图 8-11 给出的流程图，下面给出具体的程序：

```
//////////////////////////////////////////////
//正确返回 1，错误返回 0
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，写操作也可以按照字节进行，与页不同的是，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还是备注区。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，后面的附录里给出来详细的程序代码。

### 8.3.4　擦除操作

根据前面对SmartMedia卡擦除操作时序的介绍，只要通过MSP430F149正确模拟SmartMedia卡的擦除操作时序就能完成擦除操作。与读写操作不同的是，擦除是针对块的，一次擦除一块的内容。图8-12给出了擦除操作的流程图。

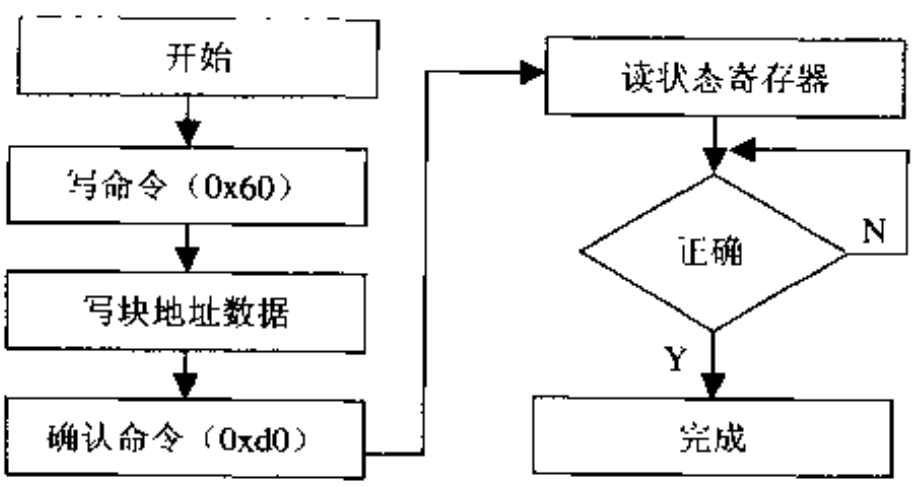



图 8-12　SmartMedia 擦除操作流程图

根据图 8-12 给出的流程图，下面给出具体的程序：

```
////////////////////////////////////////
//成功返回1，错误返回0
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; //输出块擦出命令
    WE_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;
}
```

# 8.4　系统调试

前面已经介绍了整个系统的硬件设计和软件设计，该部分结合前面介绍的来讨论整个系统的联调。系统的调试包括硬件调试和软件调试，下面分别进行介绍。

## 8.4.1　系统硬件调试

系统的硬件调试很简单，先调试电源电路和复位电路，只要这两个部分能正常工作，再进行单片机的调试，如果单片机的晶振能起振的话，则整个硬件系统没有问题。关于 SmartMedia 卡需要注意的是，如果 R/B 管脚不使用的话，则应该通过一个上拉电阻将该管脚拉为高电平，一般选用 10k 的上拉电阻就可以了。

## 8.4.2　系统软件调试

前面分别介绍了各个软件部分，为了能够测试软件的正确性，下面给出测试代码来进行测试软件的正确性。为了能够增加对卡操作的灵活性，提供了对字节、页进行操作的读写程序。下面给出测试代码。

```
//包括头文件
#include <MSP430X14X.h>
#include <stdio.h>
#include "SM.h"
void Init_CLK(void);//时钟初始化函数
int main(void)
{
```

```
unsigned long nAddr;
int j;
int n;
int nTemp;
char pBuf0[528];
char pBuf1[528];
int nCount_Err;
char chrTemp0;
char chrTemp1;
int nLen;

WDTCTL = WDTPW + WDTHOLD;    //关闭看门狗

_DINT();                     //关闭中断

//初始化
Init_CLK();
SM_Port_Init();

_EINT();                     //打开中断
nTemp = ReadID(device);      //读取设备参数

//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
nCount_Err = 0;
for(n = 0;n < 32;n++)
{
 for(j = 0;j < 528;j++) pBuf0[j] = 0;//将内容清零
 PageRead(0,n + 1024,pBuf0);//按页读操作
 for(j = 0;j < 528;j++)
 {
    if(pBuf0[j] != 0xFF)
    {
     nCount_Err += 1;
     break;
    }
 }
}
for(j = 0;j < 528;j++) pBuf0[j] = j;
PageWrite(0,1024,pBuf0);//按页写入数据，地址为：1024
```

```
    //延迟一点时间
    for(j = 1000;j > 0;j--) ;
    PageRead(0,1024,pBuf1);//按页读操作
    //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] != pBu1[j])
     {
        nCount_Err += 1;
        break;
     }
    }
    //按字节写入和读出
    chrTemp0 = 0xaa;
    //往第 1025 页第 一半区的第 12 列写入一个字节
    WriteByte(0,12,1025,chrTemp0);
    //从第 1025 页第 半区的第 12 列写入 个字节
    chrTemp0 = ReadByte(0,12,1025);

    return 0;
}
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
}
```

经过测试，该系统的软件对块擦除操作、读操作（按页读和按字节读）以及写操作（按页写和按字节写）正确可靠。

## 8.5 实例总结

本章详细介绍了大数据量本地存储系统的硬件设计和软件设计，并给出了系统的软件和测试软件，经过测试表明：该系统的软件运行可靠。在本章的基础上，利用 MSP430F14x 本身提供的 ADC 通道，用户可以加上数据采集部分，从而实现一个大数据量采集系统。由于该系统具有存储量大、功耗低、使用方便等特点，因此该系统能够在很多采集领域中得到大量的应用。通过本章的介绍，使用户能够掌握整个系统的软件和硬件设计，也使用户可以按照自己的要求适当修改硬件和软件就能实现自己采集存储系统。

### 附录：系统软件包

```
//SM.h 头文件
//函数声明
//端口初始化
void SM_Port_Init(void);
//控制线操作
void CLE_Enable(void);
void CLE_Disable(void);
void CE_Enable(void);
void CE_Disable(void);
void ALE_Enable(void);
void ALE_Disable(void);
void WE_Enable(void);
void WE_Disable(void);
void RE_Enable(void);
void RE_Disable(void);
//卡操作函数
int ReadID(unsigned char pBuf[2]);
int PageWrite(int nCol,unsigned long nRow,char *pBuf);
int PageRead(int nCol,unsigned long nRow,char *pBuf);
int WriteByte(int nCommand,int nCol,unsigned long nRow,char nValue);
char ReadByte(int nCommand,int nCol,unsigned long nRow);
int PageRead(int nCol,unsigned long nRow,char *pBuf);
int BlockErase(unsigned long nAddr);

//SM.c 文件
#include <MSP430X14X.h>
#include "SM.h"

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;
}
void CLE_Enable(void)
{
    P4OUT |= BIT0;
    return;
}
void CLE_Disable(void)
{
    P4OUT &= ~(BIT0);
    return;
}
void CE_Enable(void)
{
    P4OUT &= ~(BIT1);
    return;
}
void CE_Disable(void)
{
    P4OUT |= BIT1;
    return;
}
void ALE_Enable(void)
{
    P4OUT |= BIT2;
    return;
}
void ALE_Disable(void)
{
    P4OUT &= ~(BIT2);
    return;
}
void WE_Enable(void)
{
    P4OUT &= ~(BIT4);
    return;
}
void WE_Disable(void)
```

```
{
    P4OUT |= BIT4;
    return;
}
void RE_Enable(void)
{
    P4OUT &= ~(BIT3);
    return;
}
void RE_Disable(void)
{
    P4OUT |= BIT3;
    return;
}
////////////////////////////////////////
//正确返回 1，错误返回 0
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();
    //发送第一个行地址
    WE_Enable();
```

```
P5OUT = nADD1;
WE_Disable();
//发送第二个行地址
WE_Enable();
P5OUT = nADD2;
WE_Disable();
//发送第三个行地址
WE_Enable();
P5OUT = nADD3;
WE_Disable();
ALE_Disable();
//写如数据
for(j = 0;j < 528;j++)
{
 WE_Enable();
 P5OUT = pBuf[j];
 WE_Disable();
}
//发送写确认命令
CLE_Enable();
WE_Enable();
P5OUT = 0x10;
WE_Disable();
CLE_Disable();

//延迟一点时间，等待 R/B 低电平
for(i = 100;i > 0;i--);
//读状态寄存器
CLE_Enable();
WE_Enable();
P5OUT = 0x70;
WE_Disable();
CLE_Disable();

P5DIR = 0x00; //设置 P5 口为输入方向
for(i = 1000;i > 0;i--)
{
 RE_Enable();
 nTemp = P5IN;
 RE_Disable();
 if(nTemp == 0xc0) break;
}

if(nTemp == 0xc0) return 1;
else return 0;
```

```
}
////////////////////////////////////////////
//正确返回 1，错误返回 0
int WriteByte(int nCommand,int nCol,unsigned long nRow,char nValue)
{
    int nTemp = 0;
    int i;
    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 = (unsigned char)(nCommand);  //命令
    WE_Disable();
    CLE_Disable();

    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();
    //发送第一个行地址
    WE_Enable();
    P5OUT = nADD1;
    WE_Disable();
    //发送第二个行地址
    WE_Enable();
    P5OUT = nADD2;
    WE_Disable();
    //发送第三个行地址
    WE_Enable();
```

```
    P5OUT = nADD3;
    WE_Disable();
    ALE_Disable();
    //写入一个字节的内容
    WE_Enable();
    P5OUT = nValue;
    WE_Disable();

    //发送写确认命令
    CLE_Enable();
    WE_Enable();
    P5OUT = 0x10;
    WE_Disable();
    CLE_Disable();

    //延迟一点时间，等待 R/B 低电平
    for(i = 100;i > 0;i--);
    //读状态寄存器
    CLE_Enable();
    WE_Enable();
    P5OUT = 0x70;
    WE_Disable();
    CLE_Disable();

    P5DIR = 0x00; //设置 P5 口为输入方向
    for(i = 1000;i > 0;i--)
    {
     RE_Enable();
     nTemp = P5IN;
     RE_Disable();
     if(nTemp == 0xc0) break;
    }

    if(nTemp == 0xc0) return 1;
    else return 0;
}
////////////////////////////////////////
//正确返回 1，错误返回 0
char ReadByte(int nCommand,int nCol,unsigned long nRow)
{
    int i;
    char chrLow = 0;
    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 = (unsigned char)(nCommand);  //输出读命令代码
WE_Disable();
CLE_Disable();
//发送列地址
ALE_Enable();
WE_Enable();
P5OUT = (unsigned char)(nCol);
WE_Disable();
//发送第一个行地址
WE_Enable();
P5OUT = (unsigned char)(nADD1);
WE_Disable();
//发送第二个行地址
WE_Enable();
P5OUT = (unsigned char)(nADD2);
WE_Disable();
//发送第三个行地址
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
//读取一个字节的内容
RE_Enable();
chrLow = P5IN;
RE_Disable();

CE_Disable();
return chrLow;
```

```
}

//////////////////////////////////////////
//正确返回 1，错误返回 0
int PageRead(int nCol,unsigned long nRow,char *pBuf)
{
    int nTemp = 0;
    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();
    //发送列地址
    ALE_Enable();
    WE_Enable();
    P5OUT = (unsigned char)(nCol);
    WE_Disable();
    //发送第一个行地址
    WE_Enable();
    P5OUT = (unsigned char)(nADD1);
    WE_Disable();
    //发送第二个行地址
    WE_Enable();
    P5OUT = (unsigned char)(nADD2);
    WE_Disable();
    //发送第三个行地址
    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
    //读取一页的数据
    for(j = 0;j < 528;j++)
    {
     RE_Enable();
     pBuf[j] = P5IN;

     RE_Disable();
    }

    CE_Disable();
    return nTemp;
}
////////////////////////////////////////
//正确返回 1，错误返回 0
int ReadID(unsigned char pBuf[2])
{
    int nTemp = 0;
    int nMaker = 0;
    int nDevice = 0;

    CE_Enable(); //使能片选信号
    ALE_Disable();
    CLE_Disable();
    RE_Disable();
    WE_Disable();
    P5DIR = 0XFF; //设置P5口为输出方向

    CLE_Enable();
    WE_Enable();
    P5OUT = 0x90; //输出命令代码0x90
    WE_Disable();
    CLE_Disable();

    ALE_Enable();
    WE_Enable();
    P5OUT = 0x00; //地址周期
    WE_Disable();
    ALE_Disable();

    P5DIR = 0x00; //设置P5口为输入方向
```

```
    //读取内容
    RE_Enable();
    nMaker = P5IN;
    RE_Disable();

    RE_Enable();
    nDevice = P5IN;
    RE_Disable();

    CE_Disable();

    if(nMaker == 0xec)
    {
     if(nDevice == 0x76 || nDevice == 0x79)
     {
        nTemp = 1;
        pBuf[0] = (unsigned char)(nMaker);
        pBuf[1] = (unsigned char)(nDevice);
     }
    }
    return nTemp;
}
////////////////////////////////////////
//成功返回 1，错误返回 0
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; //输出块擦出命令
    WE_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;
}
```

# 第 9 章　语音录放系统的实现

在单片机系统中，语音应用越来越广泛。在一些银行查询系统中需要用到语音提示，同样在一些无人值守的系统中同样需要语音回答。鉴于语音应用场合越来越广泛，本章介绍一种语音录放系统，该系统采用语音芯片 ISD4004 实现，具有录音时间长、功耗低、语音质量高等特点。下面就整个系统的硬件和软件分别进行详细介绍。

## 9.1　系统描述

在一些远程无人值守系统中需要实现人机对话，这样就需要无人值守端能用语音进行应答；在一些金融应用系统中，对客户的查询也需要语音回答。许多类型的语音录放应用要求具备信息管理的功能，能够随意地录、放、删除任意一段信息。而许多语音录放系统并不能很好地满足这种要求（如磁带录音系统）。本系统通过单片机控制语音芯片 ISD4004 实现语音录放系统。本系统的单片机采用 TI 公司的 MSP430F149，MSP430F149 具有片内的 SPI 接口，ISD4004 语音录放芯片也提供了 SPI 微控制器接口，这样 MSP430F149 与 ISD4004 很容易接口，从而使得语音录放的信息管理成为可能。

与普通的录音/重放芯片相比，ISD4004 具有如下特点：首先，记录声音没有段长度限制，并且声音记录不需要 A/D 转换和压缩；其次，将快速闪存作为存储介质，无需电源即可保存数据长达 100 年，并且能重复记录 1 万次以上；此外，ISD4004 具有记录时间长（可达 16 分钟，本文采用的为 8 分钟的 ISD4004 语音芯片）的优点；最后，ISD4004 的开发应用具有外围电路简单等优点。ISD4004 语音录放芯片工作电压为 3V。芯片设计使得所有操作必须由微控制器控制，操作命令可通过串行通信接口 SPI 送入。芯片采用多电平直接模拟量存储技术，每个采样值直接存储在片内 FLASH 存储器中，因此能够非常真实、自然地再现语音、音乐、音调和效果声。采样频率可为 4.0kHz、5.3kHz、6.4kHz、8.0kHz，频率越低，录放时间越长，但音质有所下降。该系统具有运行可靠、接口简单等特点，具体的系统框图如图 9-1 所示。

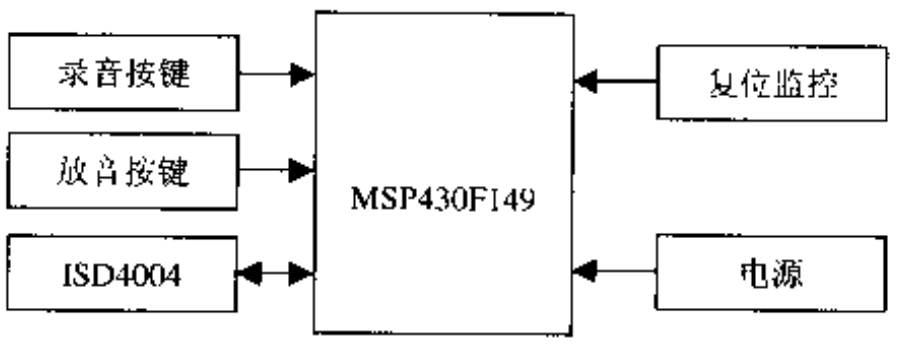



图 9-1　系统原理框图

由图 9-1 可以看出，该系统硬件简单、接口方便。整个系统的供电电压为 3V，采用 3V 供电也可以满足电池应用场合。复位监控部分为系统在开机的时候提供复位信号，并监控系

统工作的电压是否正常，当不正常的时候使系统复位。ISD4004 通过 SPI 口与单片机进行连接，由于单片机具有片内 SPI 口，因此 ISD4004 与单片机直接进行连接而不用考虑任何转换。具体各个功能模块的硬件设计在下面一节进行详细介绍。

## 9.2 系统硬件设计

该系统的硬件系统相对较简单，主要有电源模块、复位监控模块、语音录放模块及单片机处理模块。下面就具体的电路进行介绍。

### 9.2.1 语音芯片的介绍

在介绍具体的硬件设计之前，先介绍语音录放芯片 ISD4004。ISD4004 语音芯片是由美国 ISD4004 公司推出的产品，该芯片主要具有以下特征。

- 单芯片提供录音和放音功能。
- 单电压供电，供电电压为 3V。
- 功耗低。在录音的时候电流为 23mA；放音的时候电流为 15mA；当处于停止状态时，电流为 1μA。
- 单芯片的语音存储时间为 8 分钟、10 分钟、12 分钟和 16 分钟，这可以根据对语音质量的要求来选择语音存储的时间。
- 高质量的自然语音/音频信号的重构。
- 提供 SPI 接口或者微波接口。
- 完全可编程地址可以处理几个消息。
- 信息保留时间长，一般可达 100 年。
- 片内时钟选择，这样使用上有很大的灵活性。
- 提供 PDIP、SOIC 和 TSOP 等封装形式，满足不同的应用要求。

为了增加对该芯片的认识，下面给出该芯片的框图，如图 9-2 所示。

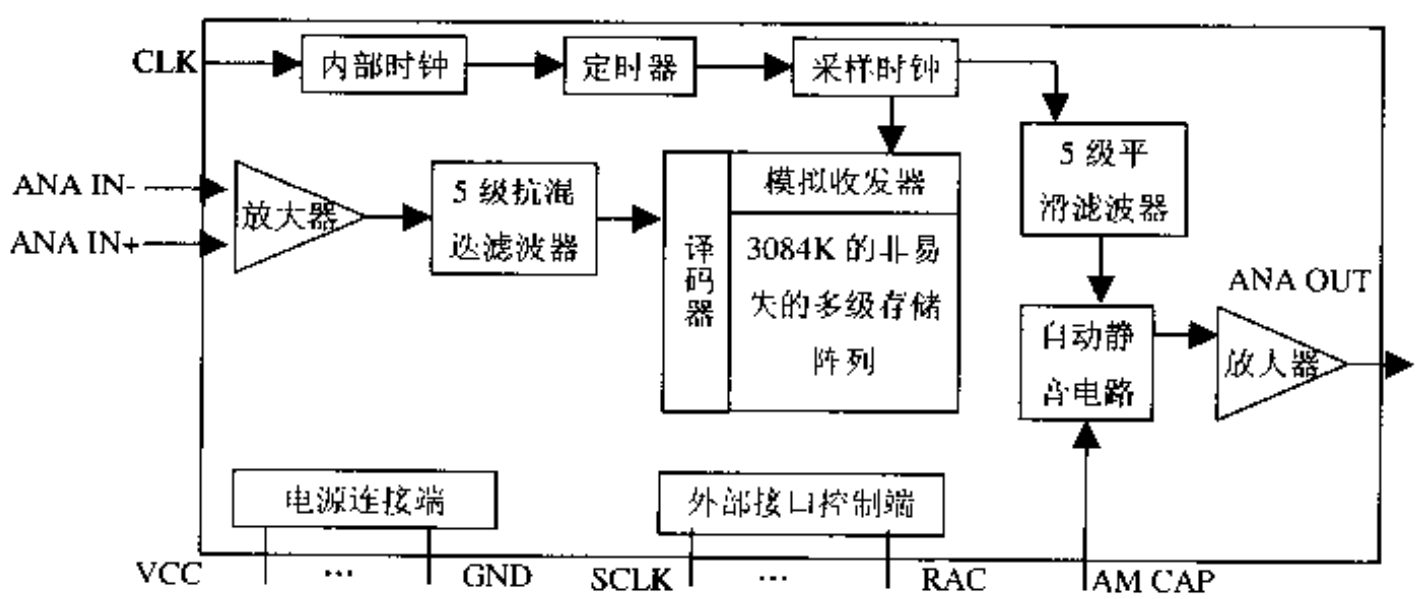



图 9-2 ISD4004 芯片内部结构框图

由图 9-2 可以看出，芯片内部有存储器，录音数据就放到存储器里。芯片提供内部时钟和采样时钟，可以设置语音信号的采样频率。另外芯片内部提供了平滑滤波器，从而可以进

一步提高输出语音的语音质量。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 9-3 所示。

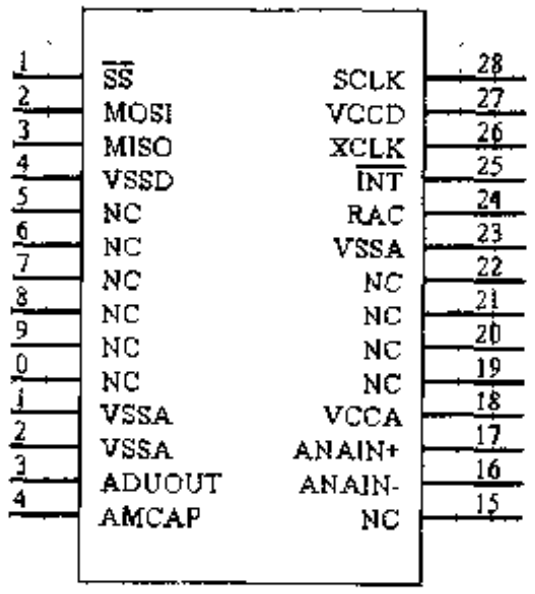



图 9-3　ISD4004 芯片的管脚

由图 9-3 可以看出，该芯片有 28 个管脚，下面对具体的管脚进行介绍。

- VCCA：模拟电源管脚。
- VCCD：数字电源管脚。
- VSSA：模拟地管脚。
- VSSD：数字地管脚。
- ANAIN+：同相模拟输入管脚。这是录音信号的同相输入端，输入放大器可用单端或差分驱动。单端输入时，信号由耦合电容输入，最大幅度为峰值 32mV，耦合电容和本端的 3kΩ电阻输入阻抗决定了芯片频带的低端截止频率。差分驱动时，信号最大幅度为峰值 16mV。
- ANAIN−：反相模拟输入管脚。差分驱动，这是录音信号的反相输入端，信号通过耦合电容输入，最大幅度为峰值 16mV。
- AUDOUT：音频输出管脚。提供音频输出，可驱动 5kΩ的负载。
- $\overline{SS}$：片选管脚。此端为低，即向该 ISD4004 芯片发送指令，两条指令之间为高电平。
- MOSI：串行输入管脚。此端为串行输入端，主控制器应在串行时钟上升沿之前半个周期将数据放到本端，供芯片输入。
- MISO：串行输出管脚。此端为串行输出端。芯片未选中时，本端呈高阻状态。
- SCLK：串行时钟管脚。芯片的时钟输入端，由主控制器产生，用于同步 MOSI 和 MISO 的数据传输。数据在 SCLK 上升沿锁存到 ISD4004，在下降沿移出 ISD4004。
- $\overline{INT}$：中断管脚。本端为漏极开路输出。ISD4004 在任何操作（包括快进）中检测到 EOM 或 OVF 时，本端变低并保持。中断状态在下一个 SPI 周期开始时清除。中断状态也可用 RINT 指令读取。OVF 标志——指示 ISD4004 的录、放操作已到达存储器的末尾。EOM 标志——只在放音中检测到内部的 EOM 标志时，此状态位才置 1。
- RAC：行地址时钟管脚。每个 RAC 周期表示 ISD4004 存储器的操作进行了一行（ISD4004 系列中的存储器共 2400 行）。该信号 175ms 保持高电平，低电平为 25ms。快进模式下，RAC 的 218.75μs 为高电平，31.25μs 为低电平。该端可用于存储管理技术。

- XCLK：外部时钟输入管脚。该管脚不用时必须接地。
- AUTCAP：自动静噪管脚。该管脚用于当没有信号时自动减少噪音，大信号不衰减时，静音衰减6Db。
- NC：空管脚。该管脚直接悬空。

经过对语音存储芯片的介绍，读者应该对该芯片的硬件接口电路的设计有了基本的认识，下面介绍具体的电路。

### 9.2.2 接口设计

整个系统的硬件相对简单，接口设计也非常容易。主要就是ISD4004与单片机的接口设计，但为了对整个硬件系统有全面的认识，我们不仅仅分析接口电路，也介绍其他部分电路，主要电路有ISD4004接口模块、CPU处理模块和电源及复位模块，下面就具体的电路进行介绍。

**1．电源电路**

整个系统采用3.0V供电，考虑到硬件系统对电源要求具有稳压功能和纹波小等特点，另外也考虑到硬件系统的低功耗等特点，因此该硬件系统的电源部分采用TI公司的TPS76030芯片实现，该芯片能很好地满足该硬件系统的要求，另外该芯片具有很小的封装，因此能有效节约PCB板的面积。电源电路具体如图9-4所示。

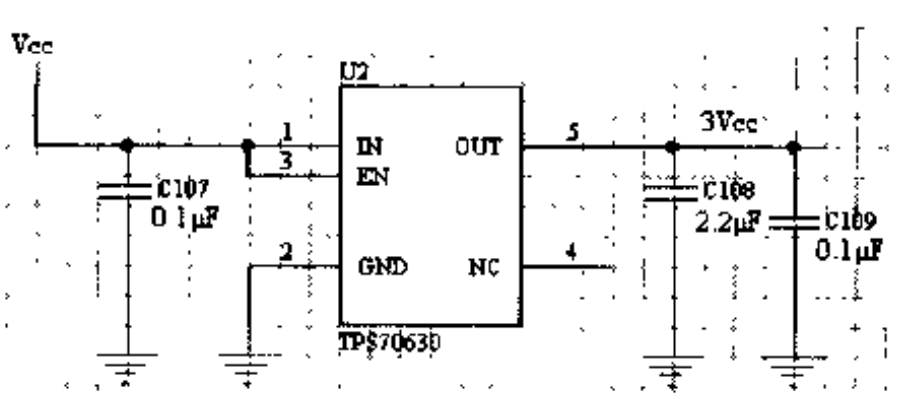



图9-4 电源电路

为了使输出电源的纹波小，在输出部分用了一个2.2μF和0.1μF的电容，另外在芯片的输入端也放置一个0.1μF的滤波电容，减小输入端受到的干扰。

**2．复位电路**

在单片机系统里，单片机需要复位电路，可以采用R-C复位电路，也可以采用复位芯片实现的复位电路。R-C复位电路很经济，但可靠性不高，用复位芯片实现的复位电路具有很高的可靠性，因此为了保证复位电路的可靠性，该系统采用复位芯片实现的复位电路，该系统采用MAX809芯片。复位电路如图9-5所示。

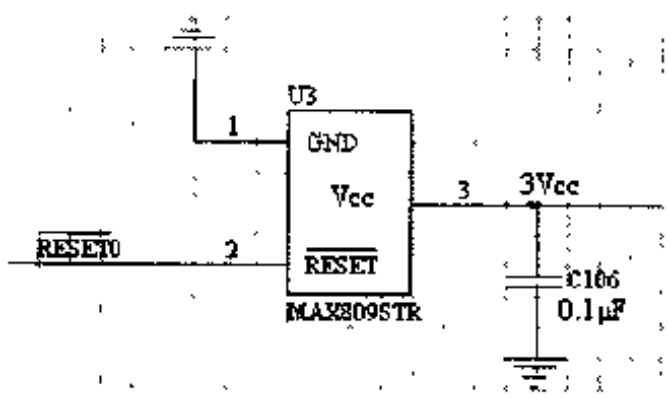



图9-5 复位电路

为了减小电源的干扰，还需要在复位芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 3．ISD4004 接口模块

语音电路采用 ISD4004 芯片实现，前面对 ISD4004 芯片进行了详细的介绍，因此对该部分电路的设计应该不困难。该部分主要是完成录音和放音，同时与单片机进行通信。ISD4004 电路通过 SPI 口实现与单片机的连接，具体的电路如图 9-6 所示。

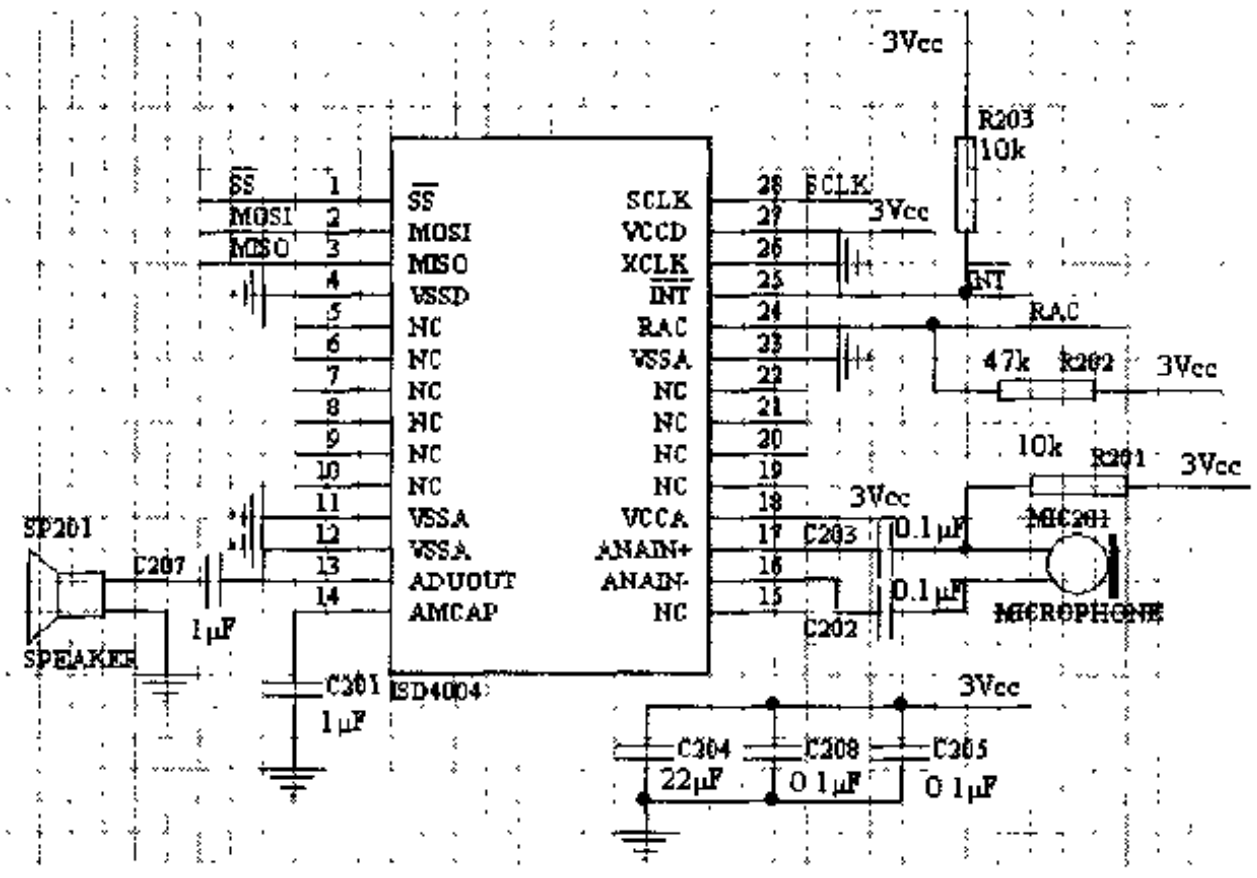



图 9-6　ISD4004 电路设计图

由图 9-6 可以看出，该电路设计很简单。ISD4004 的 SPI 口直接与单片机的 SPI 口进行连接。ISD4004 的 ADUOUT 直接与音频输出连接，在音频输出管脚上放一个电容以实现提高语音质量。ISD4004 的音频输出采用差分的方式与 MICPHONE 进行连接。ISD4004 的中断输出通过一个电阻拉高，然后与单片机进行连接。ISD4004 的 RAC 管脚与单片机的一般 I/O 口进行连接，这样通过单片机来控制该管脚的状态。在图 9-6 的下端有 3 个滤波电容，加滤波电容的目的是为了减小电源的干扰。在芯片的模拟电源和数字电源管脚端分别放一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰，为了进一步减小干扰，放一个 22μF 的电容对模拟电源端进行滤波处理。在该芯片的电源连接中，芯片的模拟电源和数字电源连在一起，在 PCB 布线的时候，电源之间连接的线要尽可能的短；同样模拟地和数字地也连在一起，在 PCB 布线的时候，地之间连接的线也要尽可能的短。

### 4．CPU 处理模块

单片机电路作为整个系统的核心控制部分，主要是完成与语音电路的接口，从而实现语音的录放功能。图 9-7 为单片机电路。

通过图 9-7 可以看出，单片机的接口电路非常简单。单片机与语音芯片 ISD4004 通过 SPI 口进行连接，设计的时候考虑到可能用到三线模式也可能用到四线模式，这样通过跳线器来选择是三线模式还是四线模式，跳线器主要是选择 $\overline{SS}$ 信号，当选择三线模式时，单片机的 P1.2 产生片选信号，单片机的 SPI 口的其他 3 个管脚直接与 ISD4004 芯片的 SPI 口的其他 3

个管脚连接；当选择四线模式时，单片机的 SPI 口的 4 个管脚直接与 ISD4004 芯片的 SPI 口的 4 个管脚进行连接。为了实现控制录音和放音的开始，利用单片机的一般 I/O 口 P1.5 和 P1.4 来作为录音和放音的按键。P1.5 和 P1.4 可以作为中断口使用，在这里设计成低电平触发方式，需要将该管脚拉高。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。另外单片机还有模拟电源的输入端，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。

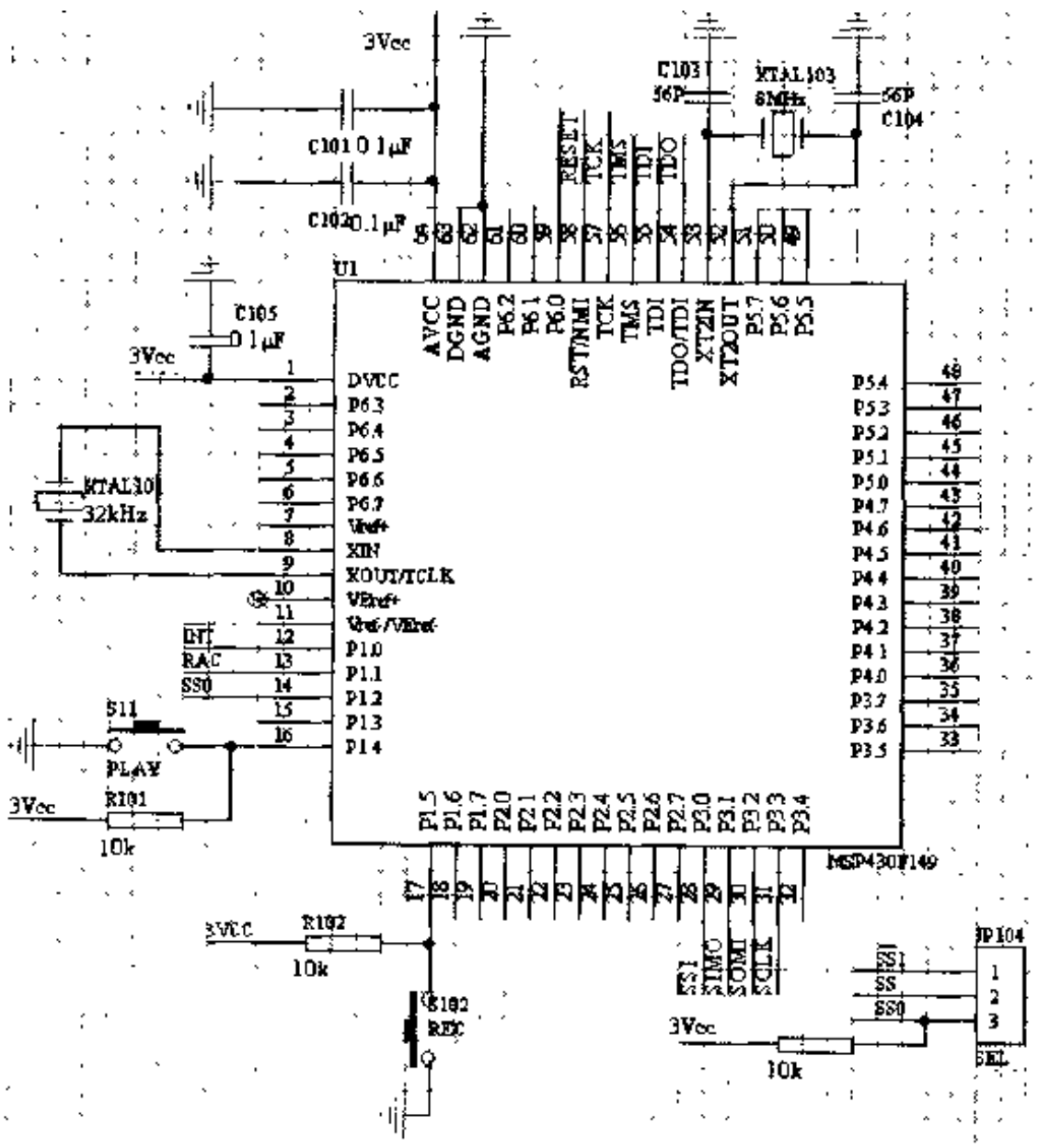



图 9-7　单片机电路

经过该节的介绍，读者应该对系统的硬件系统有了清楚的认识，下一节介绍系统的软件设计。

## 9.3　系统软件设计

系统的软件主要包括 SPI 口模块、语音录放模块和主处理模块。下面就具体的各个模块进行介绍。在具体介绍各个模块之前先介绍一下语音芯片的操作。

### 9.3.1　语音芯片的操作介绍

ISD4004 将语音数据存放到片内存储器里面。ISD4004 内部有 3840KB 的存储单元，它以行为单位，每行有 1600 个存储单元，共有 2400 行。

ISD4004 器件可以进行多段录放操作，每一段称为一个信息段。一个信息段由起始地址指针指定、记录数据和信息结束标志（EOM）3 部分组成。一个信息段占用一行或多行存储空间，可以包含多个地址单元；一个地址单元最多只能作为一个独立的段。因此，ISD4004 最多可以分为 2400 段。

ISD4004 系列芯片可由开发人员或用户任意录制、播放需要的一段或几段语音。在应用中，可以将用户预存的多段语音选择顺序连续播放，将字或词素组合成一句话甚至一段话播放出来，从而实现最准确、定量的语义表达，例如“欢迎使用四级语音查账系统”、“您的余额为 5 元”等。

ISD4004 是采用模拟存取技术集成的可反复录放一定时间语音数据的语音芯片，ISD4004 系列芯片的不同型号具有不同的存储容量，表 9-1 给出了 ISD4004 系列中的几种不同型号的存储时间。

表 9-1　ISD4004 的产品概括

| 产品号 | 时间（分钟） | 采样率（kHz） | 滤波器带宽（kHz） |
|---|---|---|---|
| ISD4004-8M | 8 | 8.0 | 3.4 |
| ISD4004-10M | 10 | 6.4 | 2.7 |
| ISD4004-12M | 12 | 5.3 | 2.3 |
| ISD4004-16M | 16 | 4.0 | 1.7 |

ISD4004 在掉电后语音数据不丢失，语音数据最多可分 2400 段，最小每段语音长度为 400ms，每段语音都可由地址线控制输出，每 400ms 为一个地址，由 A0～A10 的地址位控制。用户录制的语音每一段结束后芯片自动设有段结束标志（EOM），芯片录满后设有溢出标志（OVF）。如果用单片机等控制电路按某一段的起始地址进行放音操作，遇到段结束标志即自动停止放音，单片机收到段结束标志就开始触发下一段语音的起始地址，这样可以将很多、不同段的语音组合在一起成一句话放音出来，实现语音的自动组合。

ISD4004 芯片的控制方式有较强的通用性和方便性，它不需要事先规定每段语音的时间长度、总段数，甚至不需要知道每段语音在 ISD4004 芯片上的具体地址，只要用户记住录入语音的段顺序即可控制各段语音的自由组合。

ISD4004 的各种操作是由外部的单片机进行控制的。单片机的 SPI 口与 ISD4004 语音芯片的 SPI 口进行连接，单片机作为主机，ISD4004 语音芯片作为从机。单片机通过 SPI 口向 ISD4004 发送命令和接收响应，实现对 ISD4004 语音芯片的控制。在具体的实现的时候，就是通过单片机向语音芯片发送不同的命令码来实现控制语音芯片完成不同的操作。具体的操作码如表 9-2 所示。

表 9-2　ISD4004 芯片的操作码

| 指令 | 操作码 | | 描述 |
|---|---|---|---|
| | 地址（A0～A15） | 控制位（XXXC0C1C2C3C4） | |
| POWERUP | XX…XXX | XXX00100 | 上电 |
| SETPLAY | A0～A15 | XXX00111 | 初始回放，地址为 A0～A15 |
| PLAY | XX…XXX | XXX01111 | 回放当前地址的语音 |

续表

| 指令 | 操作码 | | 描述 |
|---|---|---|---|
| | 地址（A0～A15） | 控制位（XXXC0C1C2C3C4） | |
| SETREC | A0～A15 | XXX00101 | 初始录音，地址为 A0～A15 |
| REC | XX…XXX | XXX01101 | 在当前地址录音，直到 OVF 标志或者发送停止命令 |
| SETMC | A0～A15 | XXX10111 | 初始消息提示，地址为 A0～A15 |
| MC | XX…XXX | XXX11111 | 在当前地址处消息提示 |
| STOP | XX…XXX | XXX011X0 | 停止当前操作 |
| STOPPWRDN | XX…XXX | XXXX10X0 | 停止当前操作，进入暂停模式 |
| RINT | XX…XXX | XXX011X0 | 读中断状态位，OVF 或者 EOM |

由表 9-2 可以看出，ISD4004 提供了加电、录音、回放和停止等操作码，通过这些操作命令码就可以完成相应的操作，从而实现录音和回放的功能。命令是由单片机通过 SPI 口向 ISD4004 语音芯片发送的。单片机的输入输出数据分别通过 MOSI 和 MISO 两根数据线来传输。对于 MOSI 来说，先发送地址信息，然后发送控制位，数据的发送从最低位（LSB）开始发送。对于 MISO 来说，最低位（LSB）先移出，开始移出 OVF 状态位，然后是 EOM 状态位，接着是行指针寄存器等数据。图 9-8 为 SPI 口数据的发送顺序和相应的控制位以及状态位。

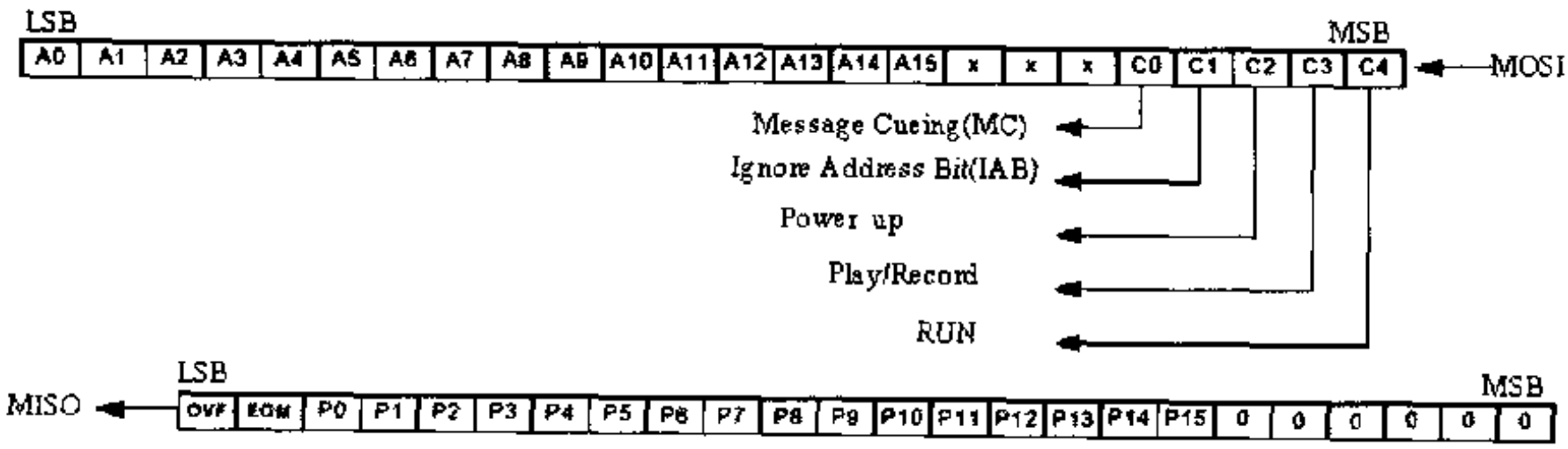



图 9-8　SPI 口发送接收数据顺序

通过图 9-8 了解了 ISD4004 语音芯片的 SPI 口数据的发送和接收顺序，下面介绍 SPI 口的时序。对于 ISD4004 来说，$\overline{SS}$ 为芯片的片选信号，这样只有 $\overline{SS}$ 管脚为低电平的时候，SPI 口才进行工作。在该系统中，由于 ISD4004 语音芯片作为从机，所以需要外部提供时钟信号，在保证 $\overline{SS}$ 为低电平并且单片机为 ISD4004 语音芯片提供了时钟信号的情况下，ISD4004 语音芯片就可以通过 MOSI 和 MISO 实现数据的接收和发送了，图 9-9 给出了 ISD4004 语音芯片的 SPI 口工作的一般时序图。

通过图 9-9 可以看出，数据在时钟信号的高电平进行采样取得，具体的各个电平的保持时间和过渡时间参看 ISD4004 语音芯片的数据手册。

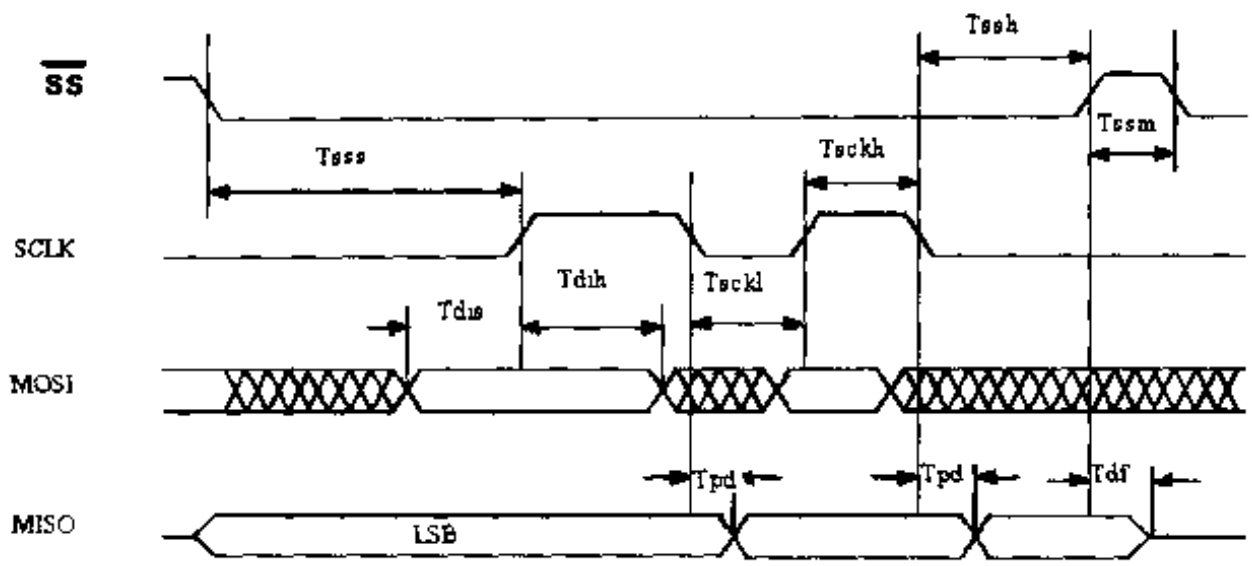



图 9-9　SPI 口的时序图

由表 9-2 和图 9-8 可以看出，单片机向 ISD4004 语音芯片先发送地址信息，再发送控制信息，ISD4004 语音芯片再向单片机发送相应数据。针对不同的操作只是发送不同的地址信息（有的命令不需要发送地址信息）和命令码就可以完成相应的操作，但是对于不同的操作命令有相同的时序形式。针对是否发送地址信息，SPI 的数据有两种格式，即 8 位格式和 16 位格式，具体的时序如图 9-10 和图 9-11 所示。

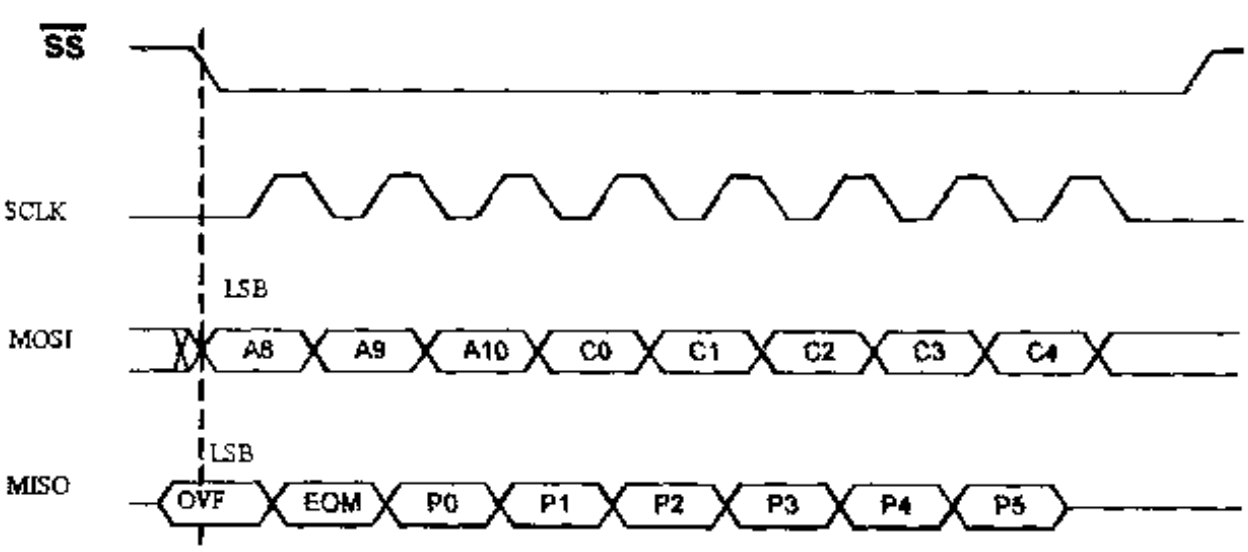



图 9-10　8 位命令格式

由图 9-10 可以看出，对于 8 位命令格式，单片机只是发送控制位 C0、C1、C2、C3 和 C4，同时发送地址字节的 A8、A9 和 A10 三位。ISD4004 向单片机返回 OVF 和 EOM 状态位，只返回 P0、P1、P2、P3 和 P4 行地址行指针寄存器。

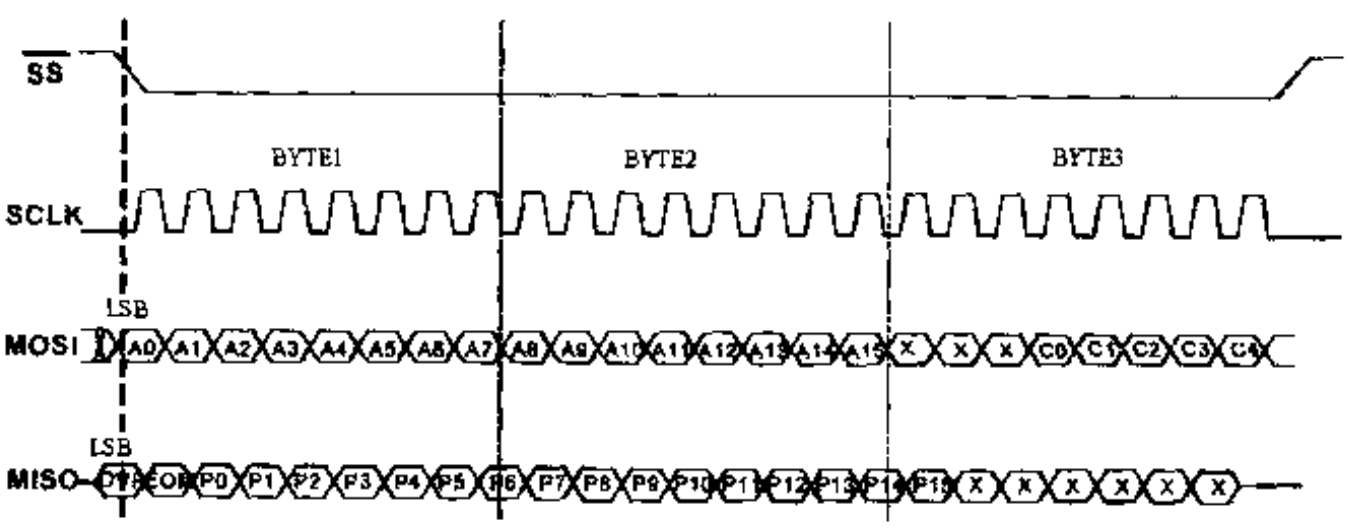



图 9-11　16 位命令格式

由图 9-11 可以看出，对于 16 位命令格式，单片机先是发送 A0～A15 的地址数据，共两个字节的地址信息。然后单片机发送一个字节的命令码，其中最低的三位与操作码无关，为任意的数据，然后接着是控制位 C0、C1、C2、C3 和 C4。ISD4004 向单片机返回 OVF 和 EOM 状态位，返回 P0～P15 行地址行指针寄存器。

通过单片机向 ISD4004 发送控制命令就可以实现语音的录音和回放的功能，图 9-12 为 ISD4004 语音芯片的录音与回放的时序图。

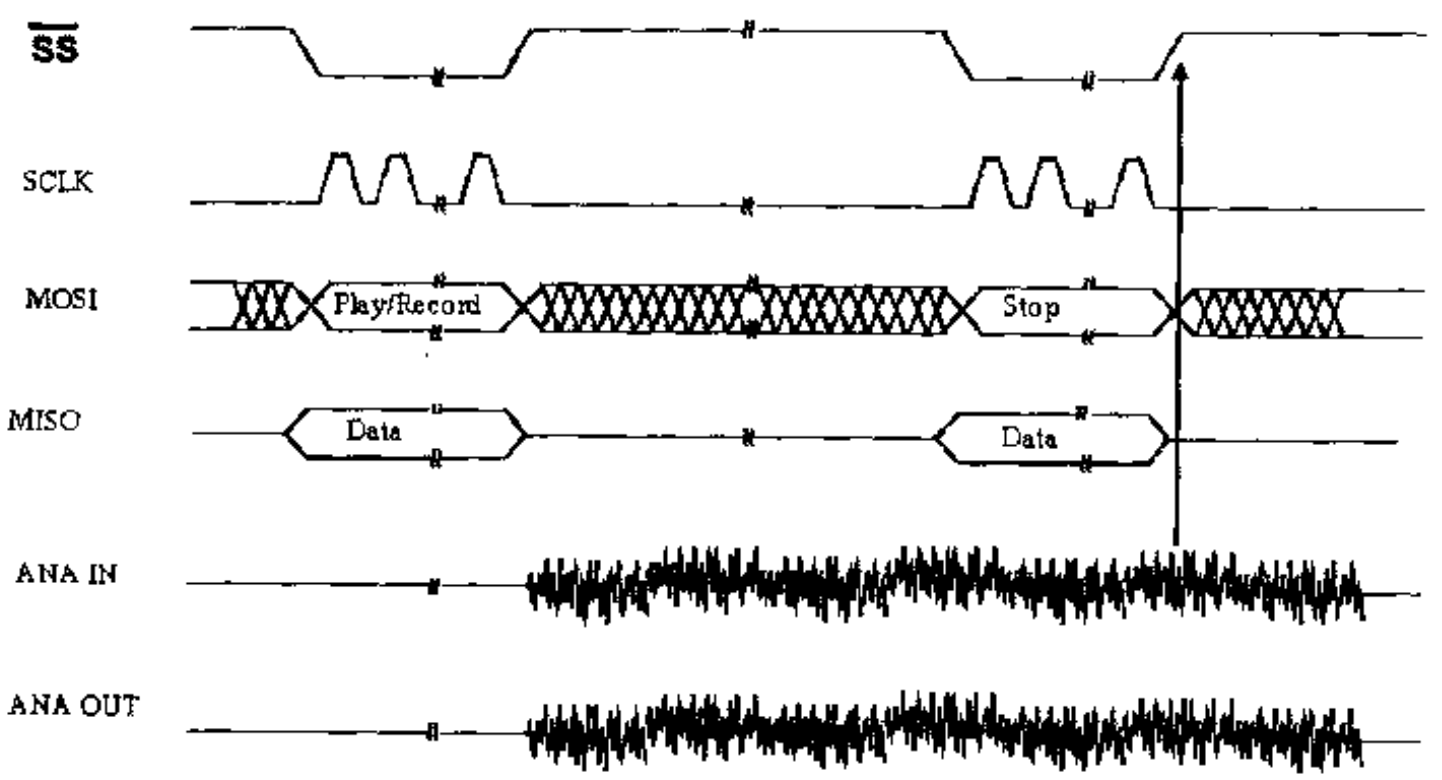



图 9-12　ISD4004 的录音与回放时序图

由图 9-12 可以看出，单片机向 ISD4004 发送录音命令，ISD4004 的 ANA IN 管脚就将语音数据存储到 ISD4004 指定的内部存储器里，单片机向 ISD4004 发送放音命令，ISD4004 就从指定的内部存储器里读出数据通过 ANA OUT 管脚输出语音数据。

根据前面的介绍，读者应当对 ISD4004 的命令操作码、SPI 口、数据发送接收时序有了一定的了解，下面结合操作码介绍 ISD4004 几种具体的操作，ISD4004 的操作主要有录音和放音等。

录音操作：录音操作主要是将外部输入的语音存储到片内的存储器里。录音操作的具体流程如图 9-13 所示。

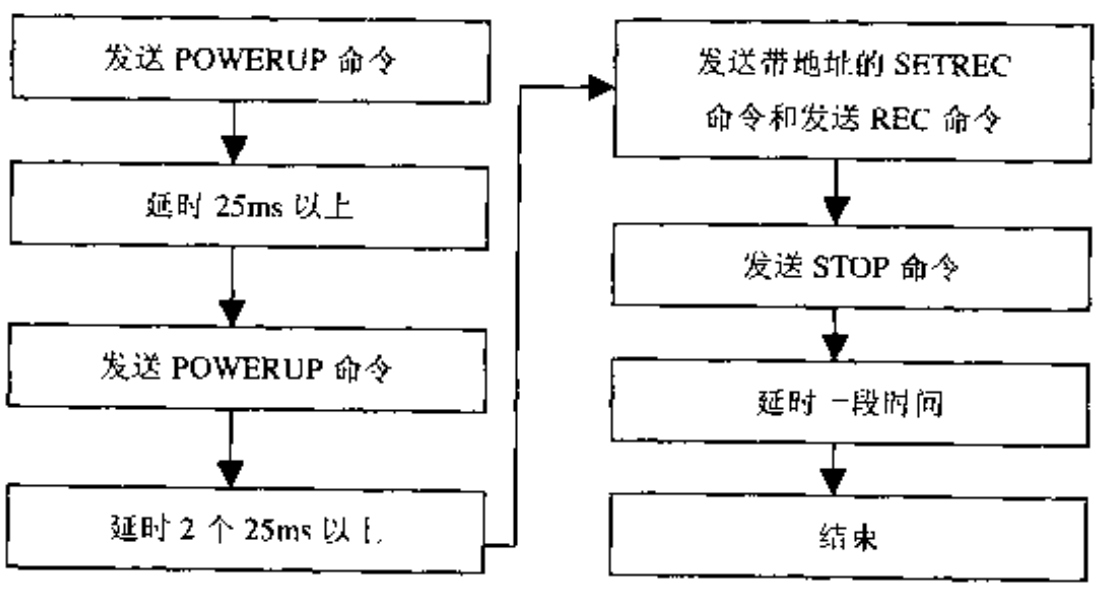



图 9-13　录音操作流程图

由图 9-13 可以看出录音操作的具体实现。首先发送 POWERUP 命令，然后延时 25ms，再发送 POWERUP 命令，延时 2 个 25ms 的时间。在进行完上述操作后，发送录音命令，如果需要指定地址的时候，发送 SETREC 命令，如果是当前地址，则发送 REC 命令，则开始录音操作。当需要停止的时候，就发送 STOP 命令，发送完 STOP 命令后需要延时一段时间，如果不发送 STOP 命令，ISD4004 会自动停止录音操作。

放音操作：放音操作主要是将存储到内部的语音数据回放到输出管脚。放音操作的具体流程如图 9-14 所示。

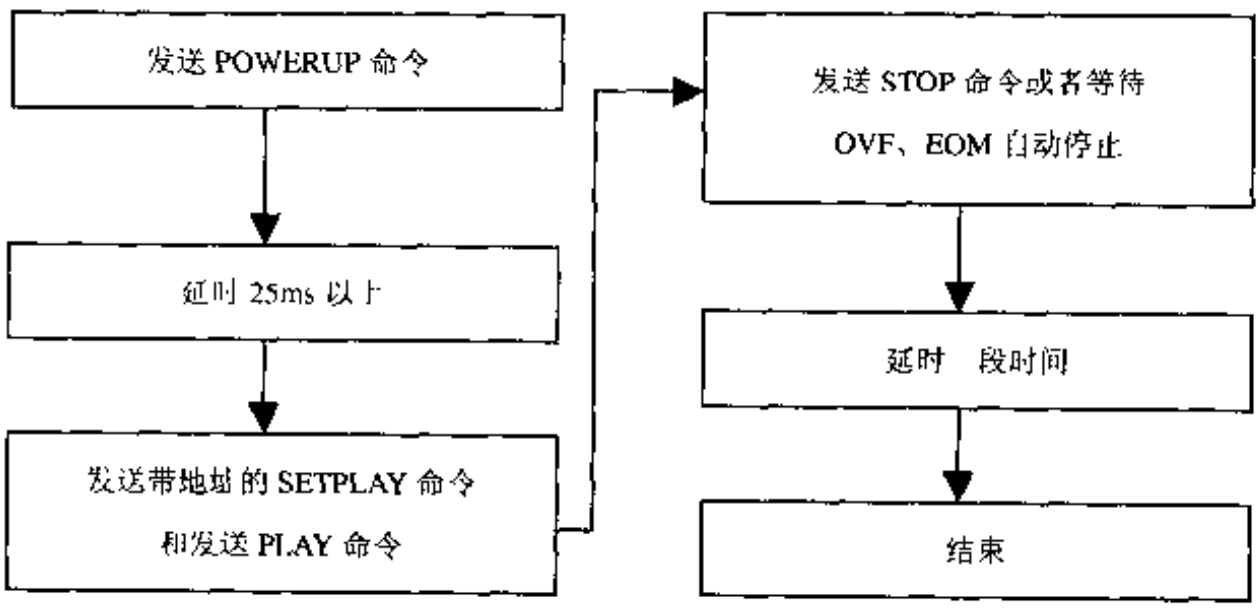



图 9-14　放音操作流程图

由图 9-14 可以看出放音操作的具体实现。首先发送 POWERUP 命令，然后延时 25ms。在进行完上述操作后，发送放音命令，如果需要指定地址的时候，发送 SETPLAY 命令，如果是当前地址，则发送 PLAY 命令，则开始放音操作。当需要停止的时候，就发送 STOP 命令，发送完 STOP 命令后需要延时一段时间，如果不发送 STOP 命令，ISD4004 会自动停止放音操作。

通过以上对 ISD4004 语音芯片的命令操作码、SPI 口数据发送接收、录音操作和放音操作等的介绍，读者应该对 ISD4004 实现语音的录放有了基本的概念，下面详细讨论程序模块的实现。

### 9.3.2　语音录放模块的实现

通过前面的介绍，读者应该对 ISD4004 语音芯片的操作有了基本认识，下面结合前面的介绍具体讨论 ISD4004 语音芯片操作的程序实现。ISD4004 语音芯片的操作主要有 SPI 口通信模块、ISD4004 语音芯片的录音操作和 ISD4004 语音芯片的放音操作等，下面就各个部分进行具体分析。

**1．SPI 口通信模块**

SPI 口通信主要是完成单片机与 ISD4004 语音芯片之间的数据通信的任务。SPI 口的初始化程序主要是设置适当的寄存器，使 SPI 口工作起来。中断服务程序主要是完成数据的接收和发送任务。中断服务程序需要与主程序之间进行数据交互，交互的方式是通过采用全局的标志变量进行通知，当接收到有数据时，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。需要交互的数据存放到全局

的缓冲区里，这样要发送数据时，主程序将需要发送的数据放到全局的发送缓冲区里。发送中断服务程序从发送缓冲区里取出数据进行发送，当接收中断服务程序接收到数据后，也放到接收的全局缓冲区里，主程序从接收的缓冲区里取出数据进行相应的分析。下面给出了中断服务程序的流程图，如图 9-15 所示。

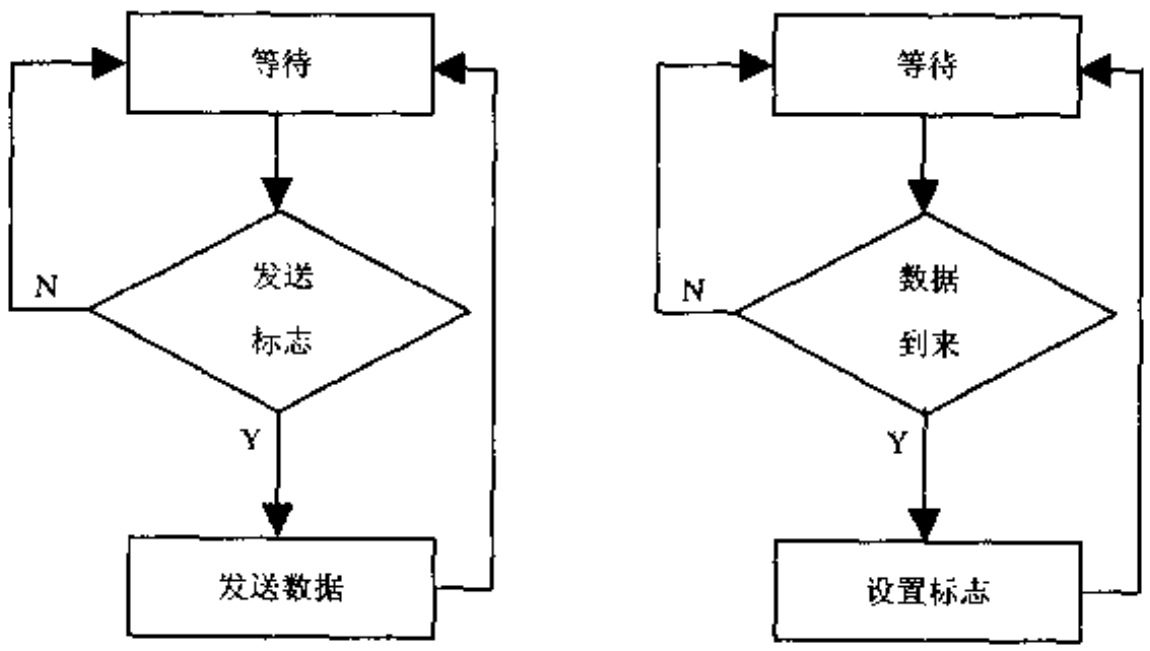



图 9-15　SPI 口通信模块的流程图

下面就 SPI 模块的程序进行介绍。SPI 口通信模块主要包括初始化和中断服务程序。

初始化部分：该部分主要完成 SPI 口的初始化功能。下面为初始化部分的代码程序。

```
void SPI_Init(void)
{
    //P3.1 P3.2 P3.3 作为 SPI 的管脚
    P3SEL = BIT3 + BIT2 + BIT1;
    //P3.1 作为输出
    P3DIR |= BIT1;
    //P3.0 作为输出
    P3DIR |= BIT0;
    //P3.3 作为输出
    P3DIR |= BIT3;
    //P3.0 输出高电平
    P3OUT = BIT0;
    //P3.1 输出高电平
    P3OUT = BIT1;
    //P3.3 输出高电平
    P3OUT = BIT3;

    //以下设置 SPI 口的参数
    U0CTL = 0X00;    //将寄存器的内容清零
    //数据为 8 位，选择 SPI 模式，单片机为主机模式
    U0CTL |= CHAR + SYNC + MM;

    U0TCTL = 0X00;   //将寄存器的内容清零
    //时钟源为 SMCLK，选择 3 线模式
```

```
    U0TCTL = CKPH + SSEL1 + SSEL0 + STC;

    UBR0_0 = 0X45;  //波特率为 115200
    UBR1_0 = 0X00;
    UMCTL_0 = 0X49; //调整寄存器

    ME1 = USPIE0;  //SPI0 模块允许
    IE1 |= URXIE0; //接收中断允许
    IE1 |= UTXIE0; //发送中断允许

}
```

通过上面的代码可以看出，要使用串口资源，需要设置 P3SEL 寄存器相应的位来使能 SPI 功能；通过 ME1 寄存器来使能 SPI 模块；通过 IE1 寄存器的相应的位来打开接收中断和发送中断；通过设置 U0CTL 的相应位来设置串口通信的格式，比如 8 位传输，也通过设置 U0TCTL 寄存器来实现时钟源的选择；设置寄存器 UBR0_0 和 UBR1_0 来设置串口通信的波特率，还可以通过设置 UMCTL_0 寄存器来实现波特率误差的调整。

串口中断主要是发送和接收数据，下面为程序代码。

```
//////////////////////////////////////////
//处理来自串口 0 的接收中断
interrupt [UART0RX_VECTOR] void SPI0_RX_ISR(void)
{
    UART0_RX_BUF[nRX0_Len_temp] = RXBUF0;//接收数据

    nRX0_Len_temp += 1;
    //接收满 10 个设置标志
    if(nRX0_Len_temp >= 10)
    {
        nRX0_Len = nRX0_Len_temp;
        nRev_UART0 = 1;
        nRX0_Len_temp = 0;
    }
}
//////////////////////////////////////////
//处理来自串口 0 的发送中断
interrupt [UART0TX_VECTOR] void SPI0_TX_ISR(void)
{
    if(nTX0_Len != 0)
    {
    nTX0_Flag = 0;  //表示缓冲区里的数据没有发送完

    TXBUF0 = UART0_TX_BUF[nSend_TX0];
    nSend_TX0 += 1;
    Delay_us(5);
```

```
        if(nSend_TX0 >= nTX0_Len)
        {
            nSend_TX0 = 0;
            nTX0_Len = 0;
            nTX0_Flag = 1;
        }
    }
}
```

上面的程序为SPI口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，在接收到10个数据后设置一个标志来通知主程序。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，则进入发送中断服务程序，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，很好利用了单片机的资源。

### 2. ISD4004录音操作

经过前面一节的介绍，读者应该对ISD4004语音芯片的录音操作有了清楚的认识。下面结合前面介绍的录音的流程图来介绍录音的具体程序的实现。

```
//录音操作
void Record(int nAddr)
{
    int i;
    unsigned char code;

    //发送加电指令
    PowerUp();
    //延时25ms以上
    Delay_ms(30);
    //再次发送加电指令
    PowerUp();
    //延时25ms以上
    Delay_ms(30);
    //延时25ms以上
    Delay_ms(30);

    //发送地址信息
    SendAddr(nAddr);

    //发送SETREC命令
    code = 0x05;
    SendOpCode(code);
```

```
//发送 REC 命令
code = 0x0D;
SendOpCode(code);

//处理结束
for(;;)
{
    //如果发生溢出的时候则结束录音
    if(nOVF == 1)
{
    nOVF = 0;//清除中断标志变量
    //发送停止命令
        Stop();
        //延迟一点时间
        for(i = 0;i < 100;i++) ;

        break;
}
//如果第 2 次按下录音键，则结束录音
   if((nRec == 0) && (nRec_count == 2))
{
    nRec_count = 0;//清除按键次数记录变量
    //发送停止命令
        Stop();
        //延迟一点时间
        for(i = 0;i < 100;i++) ;

        break;
 }

}

//发送掉电指令
PowerDown();
//延迟一段时间
Delay_ms(30);

return;
}
```

通过上面的程序可以看出，程序是完全按照图 9-13 给出的录音操作流程图实现的。首先发送 POWERUP 命令，然后延时 30ms，再发送 POWERUP 命令，延时 2 个 30ms 的时间。在进行完上述操作后，先需要确定录音存放的位置以及发送地址的信息。由于 SPI 口发送数据的顺序是从最高位（MSB）开始发送的，而 ISD4004 语音芯片接收地址信息是从最低位

（LSB）开始的，因此需要将地址的高字节和低字节的位顺序分别进行变换。发送完地址字节后，发送 SETREC 命令，通过 SETREC 命令来指明即将录音的起始地址。在确定好录音的起始地址后，发送 REC 命令，则开始录音操作。在录音的过程中需要处理录音结束的事件，当第二次按下录音键的时候，就认为录音结束，这样就发送 STOP 命令；如果在录音时发生了 ISD4004 的中断，也认为是录音结束，这样就发送 STOP 命令。发送完 STOP 命令后需要延时一段时间，然后发送掉电命令，延迟一段时间后，整个录音操作就结束。

### 3．ISD4004 放音操作

经过前面一节的介绍，读者应该对 ISD4004 语音芯片的放音操作有了清楚的认识。下面结合前面介绍的放音的流程图来介绍放音的具体程序的实现。

```
//放音操作
void Play(int nAddr)
{
    int i;
    unsigned char code;

    //发送加电指令
    PowerUp();
    //延时 25ms 以上
    Delay_ms(30);

    //发送地址信息
    SendAddr(nAddr);

    //发送 SETPLAY 命令
    code = 0x07;
    SendOpCode(code);

    //发送 PLAY 命令
    code = 0x0F;
    SendOpCode(code);

    //处理结束
    for(;;)
    {
        //如果发生溢出的时候则结束录音
        if(nOVF == 1)
        {
        nOVF = 0;//清除中断标志变量
        //发送停止命令
        Stop();
        //延迟一点时间
        for(i = 0;i < 100;i++) ;
```

```
            break;
        }
        //如果第2次按下放音键，则结束放音
        if((nPlay == 0) && (nPlay_count == 2))
        {
            nPlay_count = 0;//清除按键次数记录变量
            //发送停止命令
            Stop();
            //延迟一点时间
            for(i = 0;i < 100;i++) ;

            break;
        }

    }

    //发送掉电指令
    PowerDown();
    //延迟一段时间
    Delay_ms(30);

    return;
}
```

通过上面的程序可以看出，程序是完全按照图9-14给出的放音操作流程图实现的。首先发送POWERUP命令，然后延时25ms。在进行完上述操作后，先要确定放音开始的位置，就需要发送地址信息，由于SPI口发送数据的顺序是从最高位（MSB）开始发送的，而ISD4004语音芯片接收地址信息是从最低位（LSB）开始的，因此需要将地址的高字节和低字节的位顺序分别进行变换。发送完地址字节后，发送SETPLAY命令，通过SETPLAY命令来指明即将放音的起始地址。在确定好放音的起始地址后，发送PLAY命令，则开始放音操作。在放音的过程中需要处理放音结束的事件，当第二次按下放音键的时候，就认为放音结束，这样就发送STOP命令；如果在放音时发生了ISD4004的中断，也认为是放音结束，这样就发送STOP命令。发送完STOP命令后需要延时一段时间，然后发送掉电命令，延迟一段时间后，整个放音操作就结束。

### 9.3.3　系统软件流程

通过前面介绍的SPI口通信模块、录音操作和放音操作程序就可以实现一个简单的语音录放系统。系统通过录音按键来实现对录音的控制，在具体实现的时候，分别采用一个记数器，当按第一次按键的时候，认为是开始录音，当按第二次按键的时候，则认为是停止录音。同样系统对放音也是采用与录音相似的控制方法。下面就给出系统流程的简单实现，如图9-16所示。

由图9-16可以看出，系统程序比较简单，只是简单地处理按键响应，并根据按键的情况进行相应处理。当录音按键第一次按下的时候，则中断程序设置标志，通知主程序，主程序

在捕获到录音中断标志的时候，则进行录音操作，如果第二次按下录音按键，则停止录音操作，并清除录音按键的计数器。当放音按键第一次按下的时候，则中断程序设置标志，通知主程序，主程序在捕获到放音中断标志的时候，则进行放音操作，如果第二次按下放音按键，则停止放音操作，并清除录放音按键的计数器。系统软件程序主要包括按键中断处理和主处理程序，下面具体给出主处理程序。

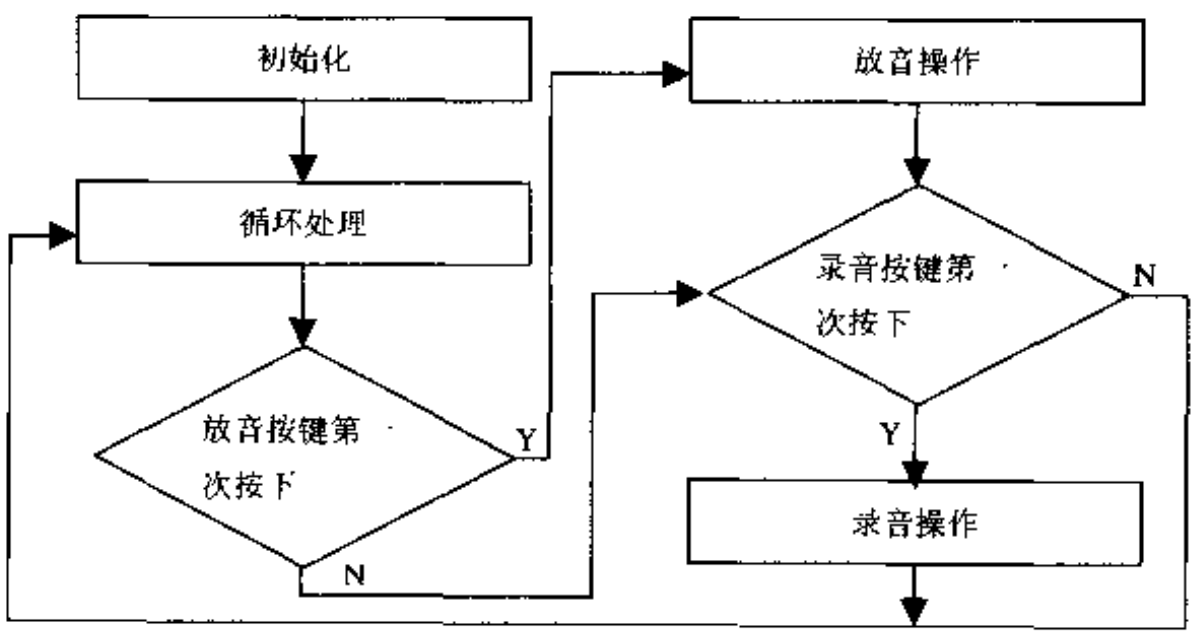



图 9-16　系统程序流程图

按键中断处理程序主要是负责处理检测按键的按下，并设置相应的标志，同时记录按键按下的次数，下面为具体的程序。

```
//////////////////////////////////////
//处理来自端口 1 的中断
interrupt [PORT1_VECTOR] void R_B_ISR(void)
{
    //处理 INT 中断
    if(P1IFG & BIT0)
    {
     nVOF = 1;//设置标志
     P1IFG &= ~(BIT0);//清除中断标志
     Delay_us(100);
    }

    //处理录音按钮
    if(P1IFG & BIT5)
    {
       if(nRec == 1)
     {
        nRec = 0;
        nRec_count += 1;
     }
    else if(nRec == 0)
    {
```

```
        nRec = 1;
        nRec_count += 1;
    }

    P1IFG &= ~(BIT5);   //清除中断标志位
    Delay_us(100);

}

//处理放音按钮
if(P1IFG & BIT4)
{
    if(nPlay == 1)
    {
        nPlay = 0;
        nPlay_count += 1;
    }
    else if(nPlay == 0)
    {
        nPlay = 1;
        nPlay_count += 1;
    }
    P1IFG &= ~(BIT4);   //清除中断标志位
    Delay_us(100);
    }
}
```

由上面的程序可以看出，中断程序需要判断是哪一个管脚产生的中断，如果某个管脚有中断产生的时候，则进行相应的处理。

主处理程序主要是捕获相应的中断标志，进行相应的处理，下面为具体的程序。

```
#include <MSP430X14X.h>
#include "SPI.h"
#include "ISD4004.h"

//定义全局变量
static char nPlay;//PLAY 按键
static char nRec;//REC 按键
static char nPlay_count;//PLAY 按键次数
static char nRec_count;//REC 按键次数
static char nVOF;//ISD4004 INT 中断标志

//定义串口操作变量
char nRev_UART0;//串口 0 的接收标志
unsigned char UART0_TX_BUF[20];//串口 0 的发送缓冲区
unsigned char UART0_RX_BUF[20];//串口 0 的接收缓冲区
```

```
static int nTX0_Len;//发送数据长度
static int nRX0_Len;//接收数据长度
int nRX0_Len_temp;
static char nTX0_Flag;//发送完成标志
int nSend_TX0;//发送记数器


void main()
{
    int nRec_Row;
    int nPlay_Row;
    WDTCTL = WDTPW + WDTHOLD;    //关闭看门狗
    _DINT();//关闭中断
    //初始化
    Init_CLK();
    PORT_Init();
    SPI_Init();

    nPlay = 0;
    nRec = 0;
    nPlay_count = 0;
    nRec_count = 0;
    nRec_Row = 0;
    nPlay_Row = 0;

    _EINT();//打开中断

    for(;;)
    {
     //放音的处理
       if((nPlay == 1) && (nPlay_count == 1))
       {
           //放音
           Play(nPlay_Row);
           nPlay_Row += 1;
           if(nPlay_Row >= 2400) nPlay_Row = 0;
       }
       //录音的处理
       if((nRec == 1) && (nRec_count == 1))
       {
           //录音
           Record(nRec_Row);
           nRec_Row += 1;
           if(nRec_Row >= 2400) nRec_Row = 0;
       }
```

```
        Delay_us(10);
    }

}
```

通过上面的程序可以看出，主程序主要是捕获是否有录音请求或者放音请求，如果有相应的请求，则进行相应的处理，并清除中断标志变量。

## 9.4　系统调试

前面已经介绍了整个系统的硬件设计和软件设计，现在讨论整个系统的调试。系统的调试包括硬件调试和软件调试，下面分别进行介绍。

**1．系统硬件调试**

系统的硬件调试很简单，先调试电源电路和复位电路，只要这两个部分能正常工作，再进行单片机的调试，如果单片机的晶振能起振的话，则整个硬件的单片机部分没有问题。关于硬件系统的 SPI 口通信和 ISD4004 语音芯片操作部分则需要结合软件进行调试。

**2．系统软件调试**

前面分别介绍了各个软件部分，为了能够进行整个系统的联调，需要软件和硬件调试结合起来，对于不同的硬件部分，应该调用不同的软件模块进行测试。经过联合调试，整个系统的软件和硬件能够正确运行。

## 9.5　实例总结

该系统通过 ISD4004 语音芯片实现语音的录放的系统具有设计简单、运行可靠等特点。通过软件测试和硬件测试证明，该系统能够安全可靠的运行。本系统采用 MSP430F149 实现的数据传输系统具有一定的通用性，它通过 ISD4004 语音芯片实现语音的录放系统在许多的应用场合能得到很好的应用。虽然本章介绍的系统相对简单，但用户完全可以在该基础上进行扩展升级实现自己功能，实现更为完整的系统。为了能使读者全面了解程序的其他模块，附录给出了程序里调用的函数源代码。

### 附录：其他程序模块

```
//字节的位顺序高低交换
unsigned char ByteSwap(unsigned char chrIn)
{
    unsigned char chrTemp;
    unsigned char chrOut;
    int i;

    chrOut = 0;
    for(i = 0;i < 8;i++)
```

```
    {
        chrTemp = (chrIn & 0x80) >> 7;
        chrIn <<= 1;
        if(chrTemp == 1)
        {
            chrOut |= (chrTemp << i);
        }
    }
    return chrOut;
}
//片选信号置低
void SS_Enable(void)
{
    //置低电平
    P3OUT &= ~(BIT0);
    _NOP();
    _NOP();
    return;
}
//片选信号置高
void SS_Disable(void)
{
    //置高电平
    P3OUT |= BIT0;
    _NOP();
    _NOP();
    return;
}
//发送上电指令
void PowerUp(void)
{
    unsigned char code;

    //上电命令
    code = 0x04;
    //发送命令码
    SendOpCode(code);

    return;
}
//发送掉电指令
int PowerDown(void)
{
    unsigned char code;
```

```
    //掉电命令
    code = 0x08;
    //发送命令码
    SendOpCode(code);

    return;
}
//发送停止命令
void Stop(void)
{
    unsigned char code;

    //停止命令
    code = 0x0C;
    //发送命令码
    SendOpCode(code);

    return;
}
//放音操作
void Play(int nAddr)
{
    int i;
    unsigned char code;

    //发送加电指令
    PowerUp();
    //延时 25ms 以上
    Delay_ms(30);

    //发送地址信息
    SendAddr(nAddr);

    //发送 SETPLAY 命令
    code = 0x07;
    SendOpCode(code);

    //发送 PLAY 命令
    code = 0x0F;
    SendOpCode(code);

    //处理结束
    for(;;)
    {
        //如果发生溢出的时候则结束录音
```

```
    if(nOVF == 1)
{
    nOVF = 0;//清除中断标志变量
    //发送停止命令
       Stop();
       //延迟一点时间
       for(i = 0;i < 100;i++) ;

       break;
}
//如果第2次按下放音键，则结束录音
   if((nPlay == 0) && (nPlay_count == 2))
{
    nPlay_count = 0;//清除按键次数记录变量
    //发送停止命令
    Stop();
    //延迟一点时间
    for(i = 0;i < 100;i++) ;

    break;
}

}
    }

    //发送掉电指令
    PowerDown();
    //延迟一段时间
    Delay_ms(30);
    nPlay_count = 0;//清除按键次数记录变量
    return;
}
//录音操作
void Record(int nAddr)
{
    int i;
    unsigned char code;

    //发送加电指令
    PowerUp();
    //延时25ms以上
    Delay_ms(30);
    //再次发送加电指令
    PowerUp();
    //延时25ms以上
    Delay_ms(30);
```

```
//延时 25ms 以上
Delay_ms(30);

//发送地址信息
SendAddr(nAddr);

//发送 SETREC 命令
code = 0x05;
SendOpCode(code);

//发送 REC 命令
code = 0x0D;
SendOpCode(code);

//处理结束
for(;;)
{
    //如果发生溢出的时候则结束录音
    if(nOVF == 1)
{
     nOVF = 0;//清除中断标志变量
     //发送停止命令
     Stop();
     //延迟一点时间
     for(i = 0;i < 100;i++) ;

     break;
}
//如果第 2 次按下录音键，则结束录音
   if((nRec == 0) && (nRec_count == 2))
{
     nRec_count = 0;//清除按键次数记录变量
     //发送停止命令
     Stop();
     //延迟一点时间
     for(i = 0;i < 100;i++) ;

     break;
}

}

//发送掉电指令
PowerDown();
//延迟一段时间
```

```
    Delay_ms(30);
    nRec_count = 0;//清除按键次数记录变量
    return;
}
//发送地址信息
void SendAddr(int nAddr)
{
    int i;
    unsigned char chrHi;
    unsigned char chrLow;
    //片选使能
    SS_Enable();
    //分别取出高、低字节
    chrHi = (unsigned char)((nAddr & 0xff00) >> 8);
    chrLow = (unsigned char)(nAddr & 0x00ff);
    for(i = 100;i > 0;i--) ; //延迟一点时间
    //字节的高低顺序变换
    chrHi = ByteSwap(chrHi);
    //字节的高低顺序变换
    chrLow = ByteSwap(chrLow);

    UART0_TX_BUF[0] = chrLow;//低地址字节
    UART0_TX_BUF[1] = chrHi;//高地址字节
    nTX0_Len = 2;
    IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序

    for(i = 100;i > 0;i--) ; //延迟一点时间
    while(1)//等待缓冲区里的数据发送完毕
    {
        if(nTX0_Flag == 1) break;
    }
    //片选禁止
    SS_Disable();
    return;
}
//发送命令码
void SendOpCode(unsigned char code)
{
    //片选使能
    SS_Enable();
    for(i = 100;i > 0;i -) ; //延迟一点时间
    UART0_TX_BUF[0] = code;//发送命令
    nTX0_Len = 1;
    IFG1 |= UTXIFG0; //设置中断标志，进入发送中断程序
```

```
    for(i = 100;i > 0;i--) ; //延迟一点时间
    while(1)//等待缓冲区里的数据发送完毕
    {
        if(nTX0_Flag == 1) break;
    }
    //片选禁止
    SS_Disable();
}
void Delay_ms(unsigned long nValue)//毫秒为单位，主时钟为 8MHz
{
    unsigned long nCount;
    int i;
    unsigned long j;
    nCount = 2667;
    for(i = nValue;i > 0;i--)
    {
     for(j = nCount;j > 0;j--);
    }
    return;
}
void Delay_us(unsigned long nValue)//微秒为单位，主时钟为 8MHz
{
    int nCount;
    int i;
    int j;
    nCount = 3;
    for(i = nValue;i > 0;i--)
    {
     for(j = nCount;j > 0;j--);
    }
    return;
}
```

# 第 10 章　短信息收发系统实现

GSM 系统是目前基于时分多址技术的移动通信体制中比较成熟、完善、应用最广泛的一种系统。主要提供话音、短信息、数据等多种业务。基于 GSM 短信息功能可以做成传输各种检测、监控数据信号和控制命令的数据通信系统，能广泛用于远程监控、定位导航、个人通信终端等。由于 GSM 网络实现了联网和漫游，这样利用 GSM 网络传输数据不须再组建专用通信网络，因此采用短信息方式传输数据的系统应用将会越来越广泛。本章介绍一种通过单片机控制 GSM 模块实现数据传输的系统，该系统采用西门子的 TC35 模块，采用短信息的方式传输数据，该系统具有使用灵活、运行可靠等特点。下面就整个系统的硬件和软件分别进行详细介绍。

## 10.1　系统描述

在一些监控系统中需要将数据传向远端服务器，实现这种应用的主要有有线传输和无线传输两种方式。有线传输具有可靠性高、成本低的特点，但有线传输需要有电话线，这样就很大限度的限制了应用的场合。无线传输有数传电台的方式，由于采用电台的方式受通信距离和入网许可等方面的限制，因此采用电台方式的无线数据传输系统在应用上也有很大的局限性。采用 GSM 网络传输数据具有接入方便、不需要组网等优点，加上短信息的广泛应用，因此采用短信息的方式传输数据将是一个实现远程传输数据的新的切入点。

本章将讨论基于单片机实现的短信息数据传输系统。本系统采用 MSP430F149 作为整个系统的 MCU。采用一个 GSM 模块——TC35 作为传输数据的无线 MODEM，TC35 与单片机通过串口进行连接。考虑到系统使用的灵活性，利用单片机的另外一个片内串口实现一个与上位机进行通信的接口，从而实现整个系统的配置功能。该系统具体的原理框图如图 10-1 所示。

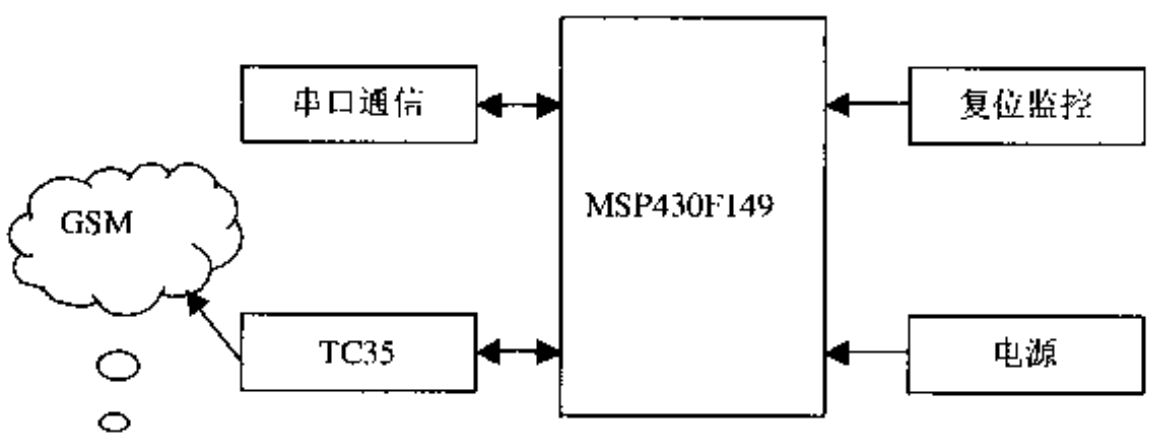



图 10-1　系统原理框图

由图 10-1 可以看出，整个系统硬件简单、接口方便。系统的供电为 5V，单片机等芯片的供电电源为 3V，而 TC35 的供电电源为 3.6V，这样电源部分有两个稳定的电压输出，并

且 3.6V 电压的供电电流必须能达到 2A 以上，这是因为 GSM 模块的峰值电流要求的。复位监控部分为系统在开机的时候提供复位信号，并监控系统工作的电压是否正常，当不正常的时候使系统复位。TC35 通过串口与单片机进行连接，由于单片机系统的工作电压是 3V，而 TC35 的工作电压是 3.6V，TC35 的串口工作电平为 CMOS 电平，这样在接口的时候需要考虑电平转换。串口通信是利用单片机的另外一个片上串口实现的与上位机进行通信的模块，考虑到上位机的接口电平与单片机的接口电平不同，需要进行电平转换。具体各个功能模块的硬件设计在下面一节进行详细介绍。

## 10.2 系统硬件设计

该系统的硬件系统比较简单，主要有电源模块、复位监控模块、TC35 模块、串口通信模块及单片机处理模块。下面就具体的电路进行介绍。

### 10.2.1 西门子 TC35 模块介绍

在介绍具体的电路之前，先介绍一下 TC35 模块。TC35 是 Siemens 公司推出的一种无线通信 GSM 模块，可以快速安全可靠地实现系统方案中的数据传输、语音传输、短信息服务（Short Message Service）和传真服务。模块的工作电压为 3.3V～5.5V，可以工作在 900MHz 和 1800MHz 两个频段，所在频段功耗分别为 2W（900M）和 1W（1800M）。模块有 AT 命令集接口，支持文本和 PDU 模式的短信息、第三组的二类传真以及 2.4k、4.8k、9.6k 的非透明模式。此外，该模块还具有电话簿功能、多方通话、漫游检测等功能。常用工作模式有省电模式、IDLE、TALK 等模式。通过独特的 40 管脚的 ZIF 连接器，实现电源连接、指令、数据、语音信号、及控制信号的双向传输。通过 ZIF 连接器及 50Ω天线连接器，可分别连接 SIM 卡支架和天线。

TC35 模块主要由 GSM 基带处理器、GSM 射频模块、供电模块（ASIC）、闪存、ZIF 连接器和天线接口 6 部分组成。作为 TC35 的核心，基带处理器主要处理 GSM 终端内的语音、数据信号，并涵盖了蜂窝射频设备中的所有的模拟和数字功能。在不需要额外硬件电路的前提下，可支持 FR、HR 和 EFR 语音编码。

TC35 是一个完整的无线 GSM 模块，本身能完成独立的功能。外部通过 40 管脚的 ZIF 连接器对 TC35 模块进行控制，从而实现电源连接、指令、数据、语音信号及控制信号的双向传输。为了便于硬件设计，对 ZIF 连接器的管脚图进行介绍。图 10-2 为 ZIF 连接器的管脚图。

为了了解各个管脚的具体功能，下面对具体的管脚分别进行详细的介绍，为下面的硬件设计打下基础。

- VBATT+：供电管脚。供电的电压在 3.3V～5.5V 之间，该管脚还必须满足峰值电流为 2A。当模块在充电的时候，该管脚还可以作为输出管脚。所有的 VBATT+必须并行连接在一起。
- GND：接地管脚。
- POWER：充电管脚。如果该管脚不用的话，将该管脚悬空。

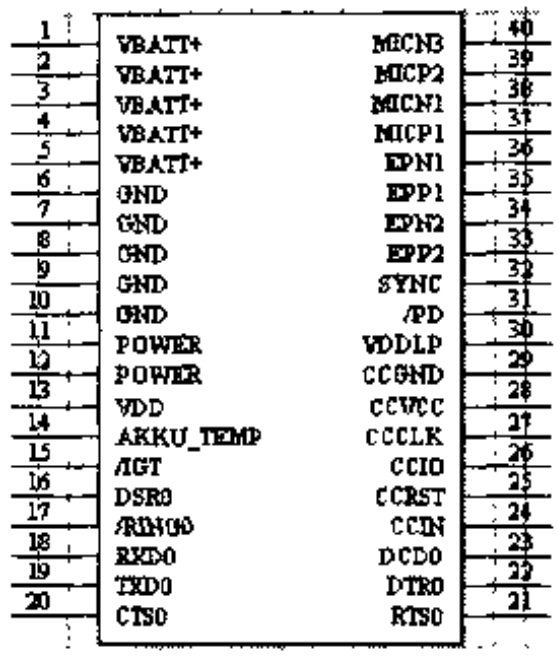



图 10-2　TC35 的 ZIF 连接器管脚图

- VDD：供电管脚。该管脚为外部应用提供电压。如果该管脚不用的话，将该管脚悬空。
- AKKU_TEMP：电池温度管脚。如果该管脚不用的话，将该管脚悬空。
- /IGT：启动管脚。该管脚用来启动 TC35 模块进行工作。该管脚低电平有效。
- DSR0：串口管脚。数据设备准备好。TC35 控制该信号向 DTE 报告状态。
- /RING0：呼叫指示管脚。该管脚用来指示应用有呼叫到来。
- RXD0：发送数据到 DTE。
- TXD0：接收数据从 DTE。
- CTS0：清除发送。该信号有效表示 TC35 模块准备接受 DTE 的数据。
- RTS0：请求发送。该信号有效表示 DTE 准备发送数据到 TC35 模块。如果该管脚不用的话，通过一个 10k 的电阻将该管脚拉高。
- DTR0：数据终端准备好，DTE 控制该信号线有效。如果该管脚不用的话，通过一个 10k 的电阻将该管脚拉高。
- DCD0：电话线上是否有载波的标志。
- CCIN：SIM 卡连接管脚。该管脚用来检测 SIM 卡是否连上，如果连上，该管脚被设置成高电平，如果没有连上，则该管脚被设置成低电平。
- CCRST：SIM 复位管脚。由基带处理器提供。
- CCIG：SIM 卡的串行数据线。输入输出数据。
- CCCLK：SIM 卡时钟线。
- CCVCC：SIM 卡的电源输出管脚。
- CCGND：SIM 卡的接地管脚。

经过这部分对 TC35 模块的介绍，对 TC35 模块有了深入的认识，下面就具体的各个部分的电路进行详细的介绍。

### 10.2.2　接口设计

整个系统的硬件相对简单，接口设计也非常容易，主要是 TC35 模块与单片机接口的设计，但为了对整个硬件系统有全面的认识，该部分不仅仅分析 TC35 模块与单片机的接口电

路，也介绍整个系统的其他部分电路，主要电路有 RS232 电路、TC35 接口电路、CPU 处理模块和电源及复位模块。下面就具体的电路进行介绍。

### 1．电源电路

整个系统采用 5V 供电。除了 TC35 外都采用 3.3V 电压供电，考虑到硬件系统对电源要求具有稳压功能和纹波小等特点，另外也考虑到硬件系统的低功耗等特点，因此该硬件系统的 3.3V 电源部分采用 TI 公司的 TPS76033 芯片实现。电源电路具体如图 10-3 所示。

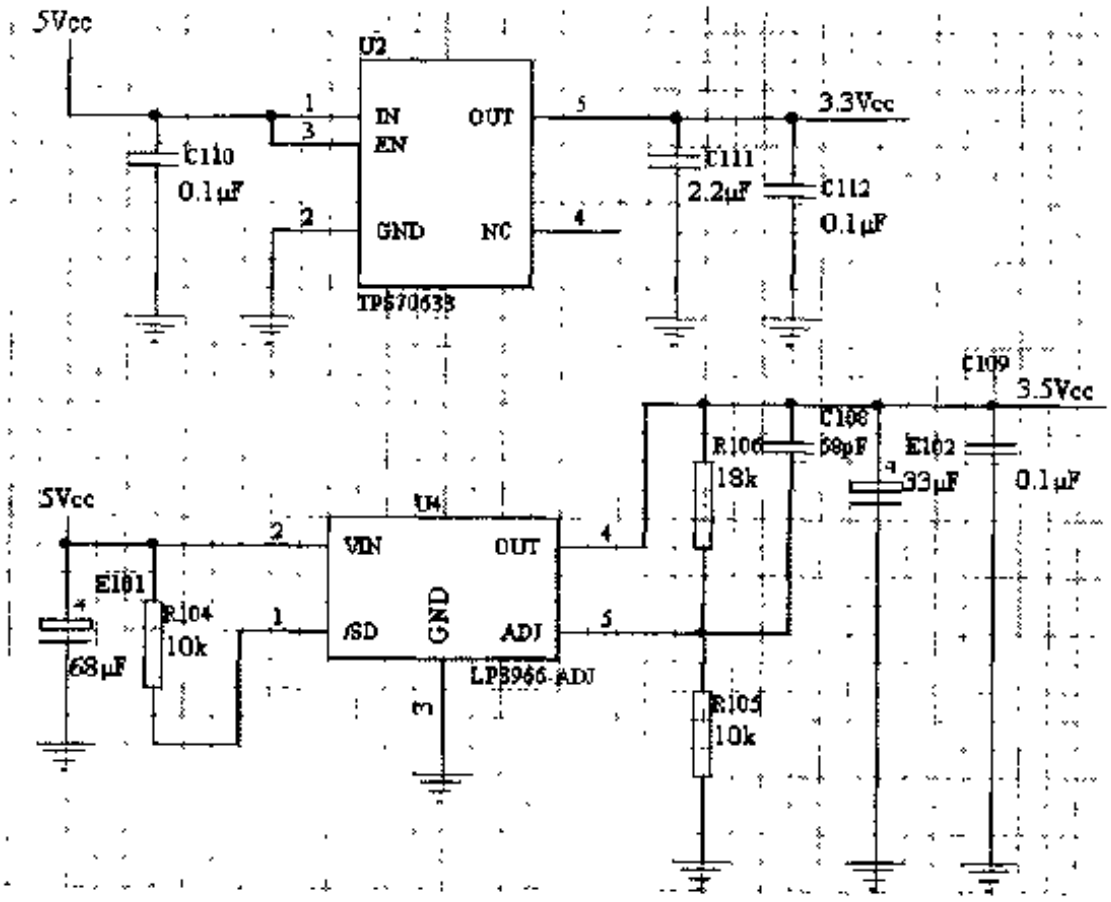



图 10-3　电源电路

为了使 3.3V 输出电源的纹波小，在输出部分用了一个 2.2μF 和 0.1μF 的电容，另外在芯片的输入端也放置一个 0.1μF 的滤波电容，减小输入端受到的干扰。

对于 TC35 模块采用 3.6V 供电，由于该电源部分的输出电流必须满足输出电流能达到 2A，在此采用 NATIONAL 公司的 LP3966-ADJ 芯片。该芯片的管脚 2 为 shutdown 管脚，在设计的时候必须通过一个 10k 的电阻拉高到 5V。该芯片的输出为可调类型，该芯片通过电阻 R2 和 R1 实现输出电压的调节。输出电压调节换算公式为：

$$R2 = R1(\frac{Vout}{1.216} - 1) \qquad (10\text{-}1)$$

通过式（10-1）可以知道：只要给定电阻 R1 和确定输出电压 Vout 的情况下，就可以计算得到电阻 R2 的值。为了使 3.6V 输出电源的纹波小，在输出部分用了一个 68pF、33μF 和 0.1μF 的电容，实现滤波。另外在芯片的输入端也放置一个 68μF 的滤波电容，减小输入端受到的干扰。

### 2．复位电路

在单片机系统里，单片机需要复位电路，复位电路可以采用 R-C 复位电路，也可以采用复位芯片实现的复位电路，R-C 复位电路具有经济性，但可靠性不高，用复位芯片实现的复

位电路具有很高的可靠性，因此为了保证复位电路的可靠性，该系统采用复位芯片实现的复位电路，该系统采用 MAX809 芯片。复位电路如图 10-4 所示。

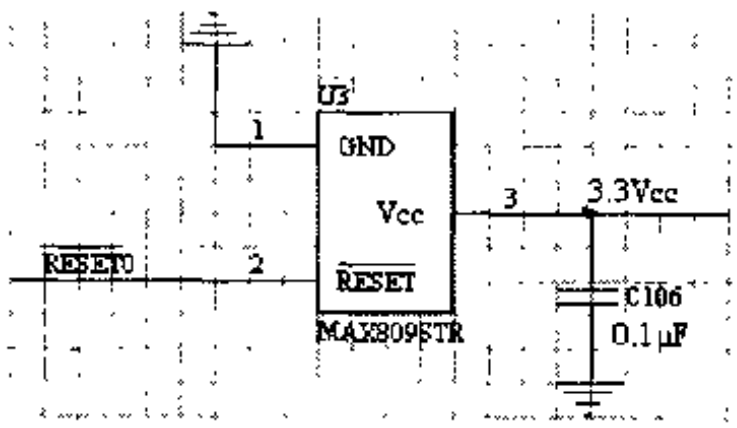



图 10-4　复位电路

为了减小电源的干扰，还需要在复位芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 3．RS232 电路

该系统实现的 RS232 电路主要是与上位机进行通信，实现单片机系统与上位机进行通信处理。由于单片机与上位机进行通信时接口电平不同，因此需要进行接口转换，这里采用 SP3220 芯片来完成接口电平的转换。关于 SP3220 芯片在前面给出了介绍，在这里不再进行说明。为了加深对 RS232 电路的了解，下面给出该电路的设计图，图 10-5 为采用 SP3220 芯片实现的 RS232 电路图。

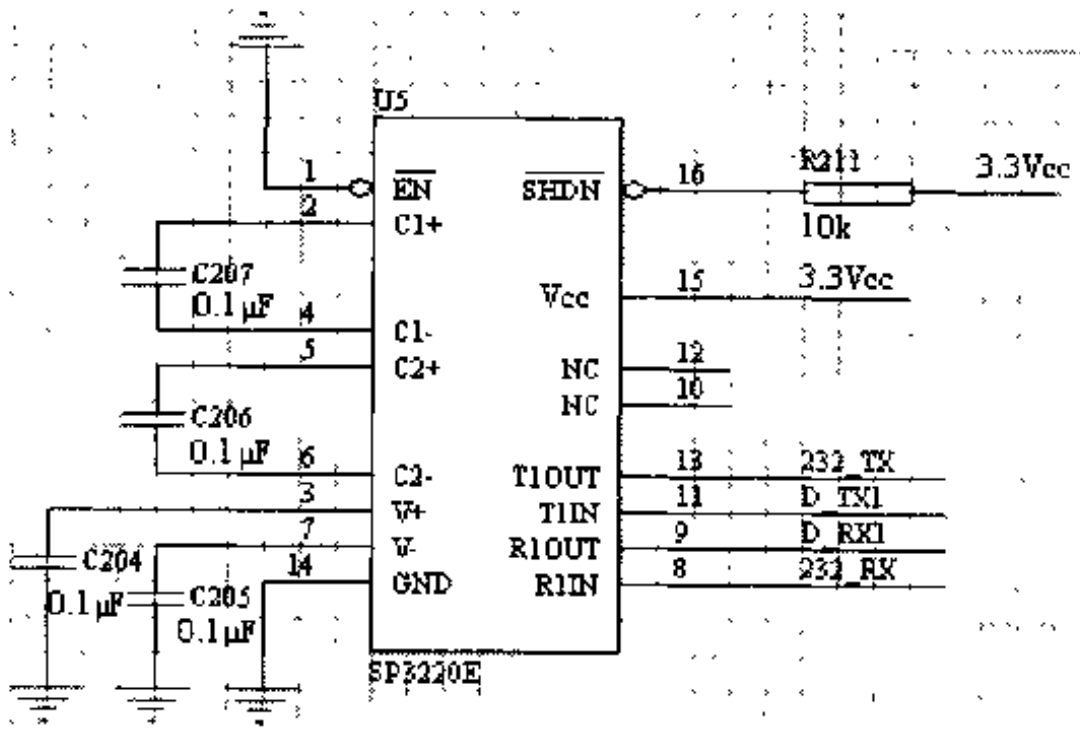



图 10-5　RS232 电路图

由图可以看出，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，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−分别放置 0.1μF 的电容实现充电作用，满足相应的充电泵的要求。管脚 T1OUT、TIN、RIOUT 和 RIN 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。考虑到减小电源的干扰，还需要在芯片的电源输入腿加一个 0.1μF 的电容来实现滤波，以减小输入端受到的干扰。

### 4．TC35 模块接口设计

经过前面对TC35模块的介绍，对TC35模块已经有了深入的了解，因此设计它的接口电路就相对比较容易了。TC35模块主要通过串口与单片机进行连接，从而单片机实现对TC35模块的控制。虽然TC35的串口提供了许多控制线，但由于考虑到设计接口的简单性，并且与单片机的UART进行连接，所以采用两线（TXD、RXD）连接。对TC35模块通信的控制可以通过软件来实现，采用软件实现控制具有使用比较灵活等特点，也能很好避免了过多的硬件信号的检测。对于TC35的其他管脚在不使用的时候，如果该管脚为输出的话，一般让该管脚悬空，如果该管脚为输入管脚，需要将该管脚通过10k的电阻上拉。另外由于/IGT管脚是控制TC35模块工作的管脚，所以需要将该管脚上拉，并且将该管脚与单片机进行连接，从而可以通过单片机来控制TC35模块的工作状态。在设计的时候需要考虑TC35模块的电源管脚并连在一起。由于TC35是一个功能完全的模块，因此这里不需要做任何的信号处理和射频处理。另外TC35模块还需要连接SIM卡座子，这样才能够实现一个完整独立的GSM终端。下面图10-6为TC35模块的接口设计。

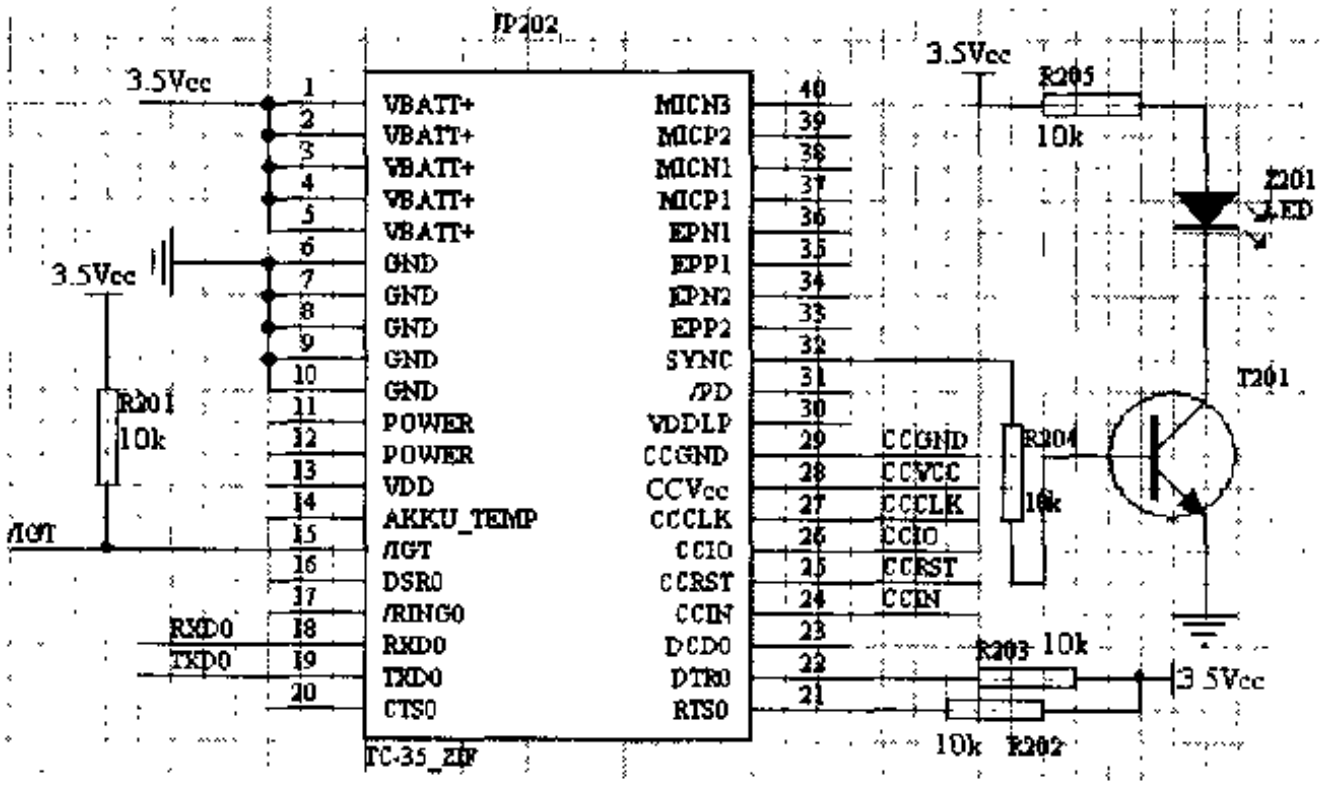



图 10-6　TC35 模块的接口设计

由图10-6可以看出，TC35接口电路设计简单。在进行串口设计时，虽然TC35模块的串口管脚的工作电平是CMOS电平，单片机的串口管脚的工作电平是TTL电平，但由于单片机的高电平和低电平的逻辑判断电平可以实现与TC35的管脚进行连接（具体的可以参看MSP430F149的数据手册），因此TC35模块的串口线直接与单片机的串口进行连接。对于TC35模块的串口管脚中的DTR0和RTS0两个管脚为输入管脚，因此分别通过10k的电阻将这两个管脚拉高。/IGT为TC35模块的工作状态控制管脚，该管脚首先通过一个电阻拉高，平时该管脚为高电平，处于不工作的状态；另外该管脚还同时与单片机的一般I/O口进行连接，这样通过单片机来实现对TC35模块工作状态的控制，当单片机在该管脚送低电平的时候，则TC35模块工作。TC35模块的SYNC管脚用来指示GSM模块的工作状态，连接一个指示灯来指示工作状态。TC35模块的SIM卡座子采用的是Molex座子，该座子有8个管脚，而TC35模块的SIM管脚只有6个管脚，具体的实现电路如图10-7所示。

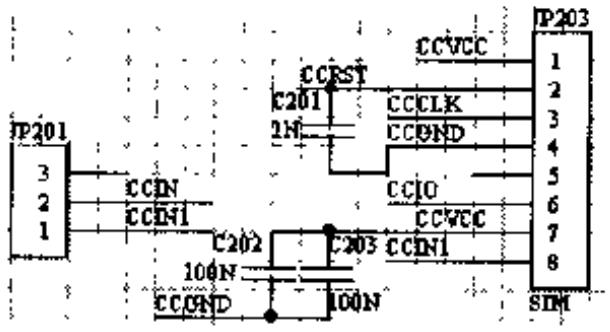



图 10-7　SIM 座子接口设计

由图 10-7 可以看出，SIM 座子只需要直接与 TC35 模块的 ZIF 连接器对应的 SIM 卡管脚进行连接，只是在需要的地方加电容进行滤波处理。对于跳线器 JP 不是必须的，这里加上主要是用来进行仿真模拟使用的。当 SIM 座子的管脚 8 与 TC35 模块的 CCIN 进行连接时，则用来模拟 SIM 卡插入的情况；当 SIM 座子的管脚 8 不与 TC35 模块的 CCIN 进行连接时，则用来模拟 SIM 卡没有插入的情况。

**5．单片机处理电路**

单片机电路作为整个系统的核心控制部分，主要是完成与 TC35 模块的通信，与上位机进行通信。单片机与 TC35 模块的通信采用单片机的串口 0（UART0）实现，虽然单片机与 TC35 模块的供电电压不同，但它们的接口电平可以直接接口，因此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 1（UART1）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。另外单片机还需要通过一个 I/O 管脚来控制 TC35 模块的工作，在该管脚上输出低电平来使 TC35 模块工作。整个系统的单片机电路相对来说十分简单。下面给出具体的单片机电路图，图 10-8 为单片机电路。

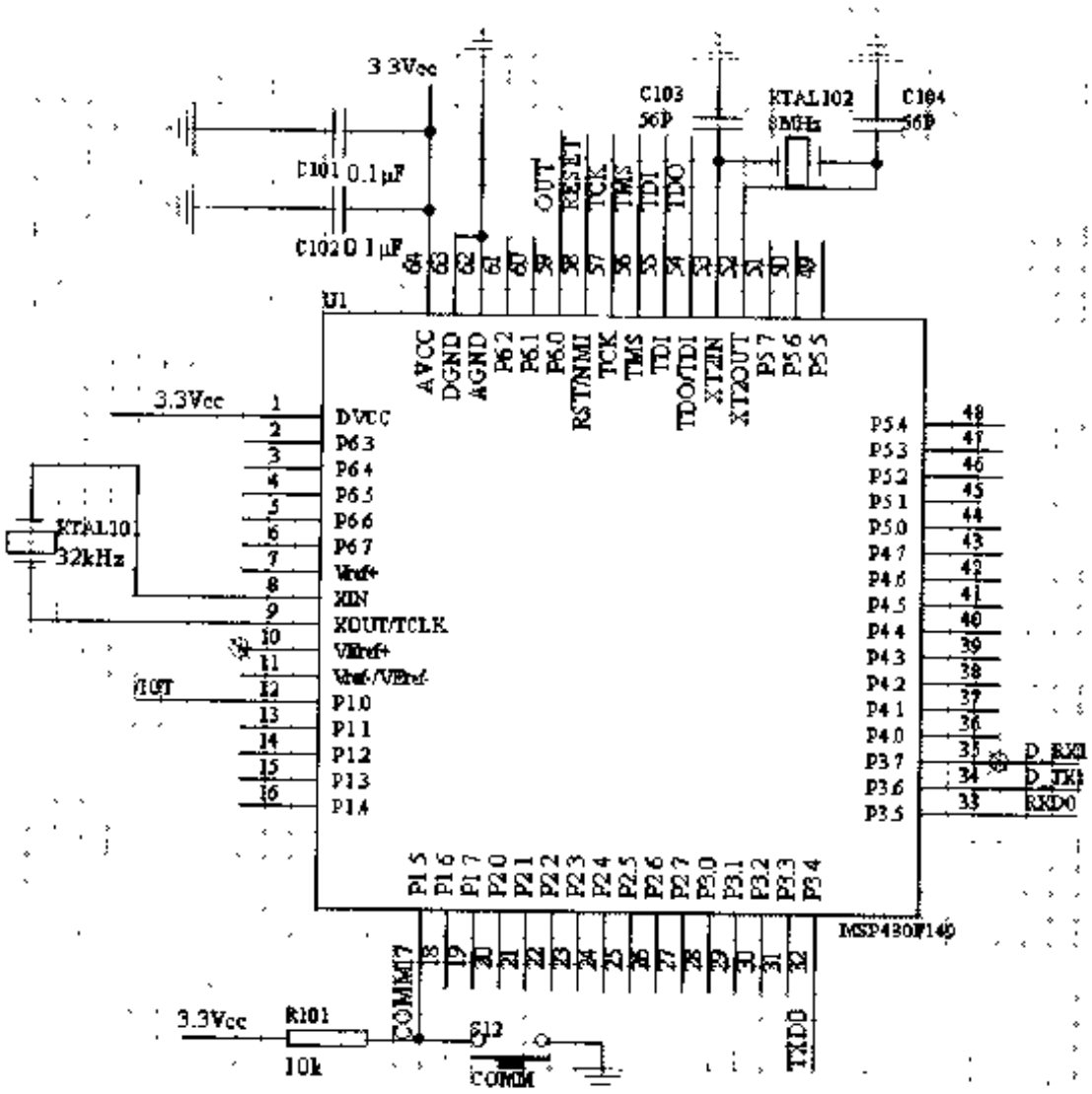



图 10-8　单片机电路

通过图10-8可以看出，单片机的接口电路非常简单。在单片机的时钟设计上与其他单片机有一定的区别，MSP430F149单片机采用两个时钟输入，既一个32kHz的时钟信号，一个8MHz的时钟信号。该系统的时钟部分都是采用晶体振荡器实现的。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。另外单片机还有模拟电源的输入端，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。利用单片机的串口0与TC35模块接口。为了控制单片机控制TC35模块传输数据的时刻，利用单片机的一般I/O口P1.5来作为启动通信的按键，由于P1.5可以作为中断口使用，这里设计成低电平触发方式，需要将该管脚拉高。单片机的串口1与上位机进行通信，因此串口1与RS232芯片进行连接。另外单片机的P1.0作为输出口，与TC35模块的/IGT管脚进行连接，实现控制TC35模块的工作，当/IGT管脚输出高电平的时候，TC35模块不工作；当/IGT管脚输出为低电平的时候，TC35模块工作。当TC35模块启动后，/IGT管脚可以是高电平，具体的时序可以参看TC35的数据手册。

经过该节的介绍，对系统的硬件系统有了清楚的认识，下一节介绍系统的软件设计。

## 10.3 系统软件设计

经过对前面系统硬件的了解，这一节介绍系统的软件设计。系统的软件主要包括TC35通信模块、串口通信模块和主处理模块。下面就具体的各个模块进行介绍。在对各个模块的软件介绍之前，先对短信息的相关信息进行介绍。

### 10.3.1 AT命令介绍

在介绍具体的各个模块之前，先介绍一下AT命令。AT命令是一套用于对TC35控制的命令，通常以“AT”开头。单片机通过向TC35发送AT命令来实现对TC35模块的控制。不同的AT命令控制TC35的不同动作。通常情况下，AT命令以字母“AT”开头，以ASCII码为13的字符结尾。下面介绍几种常用的AT命令，下面的命令如果不做特殊说明，都是需要带ASCII码为13的字符结尾，如果不带的话，则会对那条命令说明。

同一般的有线MODEM一样，TC35的AT指令集支持标准的V.25ter指令集，也支持传真指令集。作为GSM模块，它支持满足规范GSM 07.05和GSM 07.07的指令集。其中GSM 07.05指令集主要是短信息应用，考虑到本章介绍的是短信息应用系统，这里只讨论GSM 07.05指令集。GSM 07.05指令集提供的AT命令主要实现短信息的发送、删除、存储等操作，下面就某些AT命令进行具体的介绍。

**1．短信息发送命令：AT+CMGS**

该命令主要用于短信息的发送，它提供TEXT形式的短信息的发送，也提供PDU格式的短信息的发送。另外它还提供测试指令。具体的指令格式如下：

```
AT+CMGS=?
```

响应为OK。该指令主要是用来进行测试的。

```
AT+CMGS=<da>[,<toda>]<CR>
```

```
text is entered <ctrl-Z'ESC>
```

该命令是用来发送基于TEXT格式的短信息。如果发送成功，响应为：+CMGS: <mr>[,<scts>]；发送失败，响应为：+CMS ERROR: <err>。

在该命令中，<da>为字符串形式的目的地址，指接收短信息的手机号码，它的类型由<toda>来确定。<toda>为地址类型识别号，当<da>的第一个是“+”的时候，<toda>的值为整数值“145”，否则<toda>的整数值为“129”。该条命令在输入完前面的参数后，以回车符号结束，接下来输入短信息的内容，并以字符“ctrl-Z”结束，该字符的ASCII码值为“26”。如果取消发送的话，则以字符“ESC”结束。如果发送成功的话，返回服务中心的时间戳，具体参看响应命令格式。如果发送不成功，则返回错误信息。

```
AT+CMGS=<length><CR>
PDU is given <ctrl-Z/ESC>
```

该命令是用来发送基于PDU格式的短信息。如果发送成功，响应为：+CMGS:<mr>[,<ackpdu>]；发送失败，响应为：+CMS ERROR: <err>。

在该命令中，先发送命令AT+CMGC=<length>，并以回车符号结束，然后等待TC35模块返回“>”字符，当返回“>”字符后，再具体输入PDU的内容，并以字符“ctrl-Z”结束，该字符的ASCII码值为“26”。如果取消发送的话，则以字符“ESC”结束。如果发送成功的话，返回状态信息，具体参看响应命令格式。如果发送不成功，则返回错误信息。在基于PDU格式的短信息中，所有的参数均在PDU数据包里，具体的数据的格式在下面一节给出详细的说明。

**2．删除短信息命令：AT+CMGD**

该命令主要用于短信息的删除，因为一般来说手机的存储量或者 SIM 卡的存储量是有限的，因此有时需要删除已经阅读过的短信息。它具体提供两条指令，具体的指令格式如下：

```
AT+CMGD=? <CR>
```

响应为OK。该指令主要是用来进行测试的。

```
AT+CMGD=INDEX<CR>
```

该命令主要用来删除指定位置的短信息。如果删除成功，响应为OK；如果删除失败，响应为：+CMS ERROR: <err>。

**3．阅读短信息命令：AT+CMGR**

该命令主要用于阅读短信息的内容，它具体提供两条指令，具体的指令格式如下：

```
AT+CMGR=? <CR>
```

响应为OK。该指令主要是用来进行测试的。

```
AT+CMGR=INDEX<CR>
```

该命令主要用来阅读指定位置的短信息。如果操作成功，响应为具体的短信息的内容，针对TEXT模式和PDU模式，有不同的响应内容；如果操作失败，响应为：+CMS ERROR:<err>。关于短信息内容的解析在下一节详细介绍。

**4．选择信息格式命令：AT+CMGF**

该命令主要用于设置短信息的格式，将短信息设置成TEXT格式或者PDU格式，它具体提供3条命令，具体的指令格式如下：

```
AT+CMGF=? <CR>
```

响应为OK。该指令主要是用来进行测试的。

```
AT+CMGF? <CR>
```

该命令主要用来格式读取。如果操作成功，响应为+CMGF：<mode>OK。

```
AT+CMGF=<mode><CR>
```

该命令主要用来设置格式。如果操作成功，响应为+OK。这里mode的值为“0”时，代表的是PDU格式；mode的值为“1”时，代表的是TEXT格式。

**5．短信中心设置命令：AT+CSCA**

该命令主要用于设置短信中心，它具体提供3条命令，具体的指令格式如下：

```
AT+CSCA=? <CR>
```

响应为OK。该指令主要是用来进行测试的。

```
AT+CSCA? <CR>
```

该命令主要用来读取短信中心的地址。如果操作成功，响应为+CSCA：<sca><tosca>OK。响应中的<sca>为短信中心的地址；<tosca>用来表示为地址类型识别号，可以参照AT+CMGS命令中的<toda>的含义。

```
AT+CSCA=<sca>,[<tosca>]<CR>
```

该命令主要用来短信中心的地址。如果操作成功，响应为+OK。<sca>为短信中心的地址；<tosca>用来表示为地址类型识别号，<tosca>可以参照AT+CMGS命令中的<toda>的含义。

以上只列出了短信息操作的一些AT命令，TC35模块还支持其他AT命令，这里不再进行介绍，具体参看AT命令手册。

## 10.3.2 发送短信息的实现

在前面了解了短信息相关的AT命令后，对短信息的一些操作有了基本的概念，这一节具体来介绍短信息相关操作的实现。在该系统中，短信息操作是基于PDU格式的，因此本节只讨论PDU格式的具体实现。短信息的实现主要有短信中心地址的设置、短信息格式的设置、短信息发送、短信息接收、短信息的删除等操作。下面对各个部分的实现进行具体的分析。在下面的介绍中，由于数据的发送是基于串口实现的，这样该部分讨论的软件位于串口发送软件之上，因此这里具体讨论的是数据包的封装和解析。

**1．短信中心地址的设置**

在短信息的发送过程中，源 GSM 终端将短信息发送到另外一个目的 GSM 终端，源 GSM 终端首先将短信息发送到短信中心，由短信中心再转发给目的终端，因此实现的机制是存储转发的机制，这样就必须要正确设置号短信中心的地址。下面为该部分程序的具体代码。

```
//设置短信中心地址
int setCsca(char pBuf[])
{
    pBuf[0] = 'A';
    pBuf[1] = 'T';
    pBuf[2] = '+';
    pBuf[3] = 'C';
    pBuf[4] = 'S';
    pBuf[5] = 'C';
    pBuf[6] = 'A';
    pBuf[7] = '=';
    pBuf[8] = '+';
    pBuf[9] = '8';
    pBuf[10] = '6';
    pBuf[11] = '1';
    pBuf[12] = '3';
    pBuf[13] = '8';
    pBuf[14] = '0';
    pBuf[15] = '0';
    pBuf[16] = '2';
    pBuf[17] = '3';
    pBuf[18] = '0';
    pBuf[19] = '5';
    pBuf[20] = '0';
    pBuf[21] = '0';
    pBuf[22] = ',';
    pBuf[23] = '1';
    pBuf[24] = '4';
    pBuf[25] = '9';
    pBuf[26] = 13;

    return 27;
}
```

在该程序中，封装好命令“AT+CASC=+8613800230500，149”，函数返回的是数据包的长度。命令中的参数必须是字符形式。命令中的“+8613800230500”为短信中心的地址。命令中的“149”为地址识别号，这里地址信息中使用了“+”，因此识别号为“149”。

**2．短信息格式的设置**

在发送短信息的时候，需要选择短信息的格式，短信息的格式有 TEXT 和 PDU 两种格式。在该系统中选择 PDU 格式。下面为具体的程序。

```
//设置短信息格式
int setCmgf(char pBuf[])
{
    pBuf[0] = 'A';
    pBuf[1] = 'T';
    pBuf[2] = '+';
```

```
    pBuf[3] = 'C';
    pBuf[4] = 'M';
    pBuf[5] = 'G';
    pBuf[6] = 'F';
    pBuf[7] = '=';
    pBuf[8] = '0';
    pBuf[9] = 13;

    return 10;
}
```

在该程序中，封装好命令“AT+CMGF=0”，函数返回的是数据包的长度。由于设置成PDU 格式，因此命令里面的参数为“0”。

### 3. 短信息发送

由于采用的 PDU 格式发送短信息，PDU 数据包有具体的帧结构，因此必须按照 PDU 数据包的格式进行封装数据。下面先分析 PDU 数据包的帧结构。图 10-9 为 PDU 数据包的帧格式。

| SMSC | PDU 类型 | MR | DA | PID | DCS | VP | UDL | UD |
|---|---|---|---|---|---|---|---|---|

图 10-9　PDU 数据包的帧结构

在 PDU 数据包的帧结构中，“SMSC”字段为短信息中心的地址；“PDU 类型”来指明数据包的类型；“MR”数据包是表示发出信息；“DA”为目的地址；“PID”为协议识别号；“DCS”为短信息的编码格式，对于数字或者字符采用编码值为“00”，如果内容是汉字的话，则采用的编码值为“08”，采用的是 UNICODE 编码方式，在该系统中采用的字符编码方式；“VP”表示短信息的有效时间；“UDL”表示数据内容的长度；“UD”为具体的短信息内容。这里需要强调的是：在 PDU 数据包里面，所有的数字是以字符形式发送的。

根据前面介绍的短信息发送的 AT 命令，在 PDU 模式下，先发送完长度字节后，必须等待 TC35 模块的响应，当响应为“>”时，则继续发送 PDU 数据包。下面给出 PDU 模式下，短信息发送的流程图，图 10-10 为短信息发送的流程图。

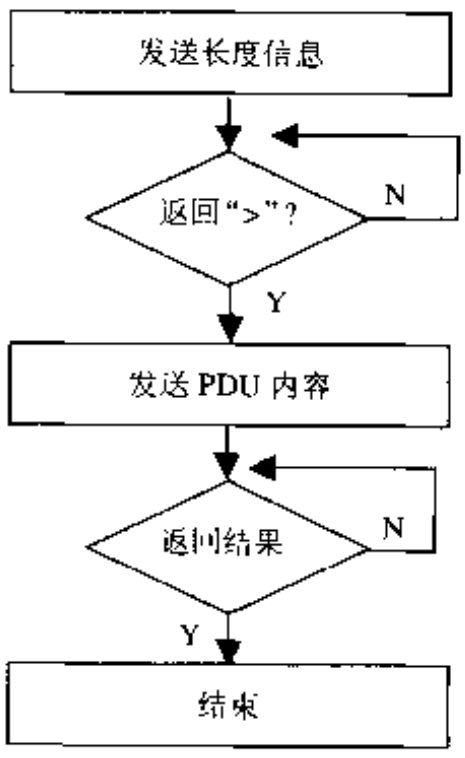



图 10-10　短信息发送流程

下面给出发送短信息的程序，程序如下：

```
//发送短信息
void sendSms(char pPhone[],int phonelen,char pData[],int nLen,
          int *nTXLen1,int *nTXLen2,char pOut1[],char pOut2)
{
   char strHead[18] = {'0','8','9','1','6','8',
                       '3','1','0','8','2','0',
                       '0','3','0','5','F','0'};
   char chrInfo[6] = {'1','1','0','0','0','B'};
   int nLen_temp;
   int nContent_Len;
   int nTempLen;
   int nOff;
   int nOffset;
   char chrTemp[100];
   char chrTmp[100];
   char pBuf[200];
   char phoneTemp[20];
   char nTemp[100];
   char Len[1];
   int i;
   int n;
   nOff = 0;
   nOffset = 0;
   for(i = 0;i < phonelen;i++)
   {
    chrTmp[i] = pPhone[i];
   }
   chrTmp[phonelen] = 'F';
   phonelen += 1;
   //将电话号码按照规范的顺序作成
   n = 0;
   for(i = 0;i < phonelen / 2;i++)
   {
    phoneTemp[n++] = chrTmp[2 * i + 1];
    phoneTemp[n++] = chrTmp[2 * i];
   }
   copy(chrTemp,0,chrInfo,0,6);
   nOff = 6;
   chrTemp[nOff] = '8';
   nOff += 1;
   chrTemp[nOff] = '1';
   nOff += 1;
   //设置电话号码
```

```
copy(chrTemp,nOff,phoneTemp,0,phonelen);
nOff += phonelen;
chrTemp[nOff] = '0';
nOff += 1;
chrTemp[nOff] = '0';
nOff += 1;

nContent_Len = nLen;
//设置编码类型
chrTemp[nOff] = '0';
nOff += 1;
chrTemp[nOff] = '0';
nOff += 1;

Len[0] = nLen;
for(i = 0;i < 10;i++)
{
 chrTmp[i] = 0;
}
ByteToChar(Len,chrTmp,1);
chrTemp[nOff] = 'A';
nOff += 1;
chrTemp[nOff] = 'A';
nOff += 1;
copy(chrTemp,nOff,chrTmp,0,2);
nOff += 2;
nLen_temp = nOff;
nLen_temp += nContent_Len;

//获得长度的字符数组
nTempLen = IntToChar(nLen_temp,chrTmp);
//封装长度信息
nTemp[0] = 'A';
nTemp[1] = 'T';
nTemp[2] = '+';
nTemp[3] = 'C';
nTemp[4] = 'M';
nTemp[5] = 'G';
nTemp[6] = 'S';
nTemp[7] = '=';
nOffset = 8;
//长度
for(i = 0;i < nTempLen;i++)
{
   nTemp[nOffset] = chrTmp[i];
```

```
        nOffset += 1;
    }
    nTemp[nOffset] = 13;
    for(i = 0; i < nOffset;i++)
    {
     pOut1[i] = nTemp[i];
    }
    *nTXLen1 = nOffset;

    //封装内容数据
    copy(pOut2,0,strHead,0,18);
    nOffset = 18;
    copy(pOut2,nOffset,0,chrTemp,nOff);
    nOffset += nOff;
    Encode(pData,pBuf,nLen);
    ByteToChar(pBuf,chrTmp,nLen);
    copy(pOut2,nOffset,chrTmp,0,(2 * nLen));
    nOffset += (2 * nLen);
    pOut2[nOffset] = 26;

    *nTXLen2 = nOffset;
}
```

该程序主要完成对数据的封装，先按照发送短信息的命令封装长度信息，然后封装好 PDU 数据包。pPhone[]为电话号码，字符数组类型，在进行封装的时候必须按照规范进行顺序重新调整。Phonelen 表示电话号码的长度。pData[]为内容数据，即需要发送的短信息的内容，为字节数组。nLen 为内容数据的长度。nTXLen1 表示长度信息包的长度，用来指示串口需要发送数据的长度。nTXLen2 表示 PDU 信息包的长度，用来指示串口需要发送数据的长度。pOut1 为长度信息包， pOut2 为 PDU 信息包。函数里 strHead 用来表示 SMSC 的地址，这里短信中心的号码的顺序进行了调整，以满足规范要求。strHead 的内容为“0891683108200305F0”，其中“08”用来指示号码的长度，以字节为单位；“91”表示短信息中心号码类型，91 是 TON/NPI；“683108200305F0”为短信息中心的号码，它经过十六进制以字节为单位的高低半字节换位处理，号码是奇数的添 F，构成一个 HEX 字节，因此它实际的号码应该是“8613800230500”。函数里面的 chrInfo 的内容为“11000B”，其中“11”表示 PDU SMS 发送的文件头字节，这里“11”指正常发送短信息；“00” 表示信息类型，这里 00 指让手机自动加上主叫号码；“0B”表示被叫号码长度。接着在数据包里面填入信息字节“81”，表示被叫号码类型。在数据包里面封装电话号码信息，电话号码经过十六进制以字节为单位的高低半字节换位处理，号码是奇数的添 F，构成一个 HEX 字节。在填入电话号码信息后，填入协议识别信息“00”和数据编码类型“00”，这里传输的是数据，因此采用的编码类型是 GSM Default Alphabet。编码类型后的“AA”表示短信息被保留的时间为 4 天，具体的算法参看具体的规范。在填完时间信息后，填入长度信息。填入完长度信息后填入具体的数据内容，最后再填入结束符号“ctrl-Z”它的 ASCII 值为 26，这里不需要将 26 转换成字符，直接传输数值 26。所有的数据内容也必须是字符型的，这样需要将字节流转换成字符

流，在将字节流转换成字符流之前需要对字节流按照 GSM Default Alphabet 格式编码。GSM Default Alphabet 编码是将由 7 位 ASCII 码移位组成 8 位十六进制码(octet)，函数 Encode (char in[],char out[],int nLen)完成 GSM Default Alphabet 编码，in[]为需要编码的数据，out[]为编码后得到的数据，nLen 为编码的长度。为了将字节流转换成字符流，提供函数 ByteToChar(char nInPut[],char Out[],int nLen)来实现，nInPut[]为输入的字节流，Out[]为转换后得到的字符流，nLen 为需要转换的长度。

下面给出具体的编码函数和转换函数，具体函数如下：

```
//编码函数
void Encode(char in[],char out[],int nLen)
{
    int nOrigin = 0;
    int nCode = 0;
    while(true)
    {
        if(nOrigin >= nLen) break;
        out[nCode] = in[nOrigin];

        if((nOrigin + 1) >= nLen) break;
        out[nCode] |= (byte)((in[nOrigin + 1] & 0x01) << 7);
        out[nCode + 1] = (byte)((in[nOrigin + 1] >> 1) & 0xff);

        if((nOrigin + 2) >= nLen) break;
        out[nCode + 1] |= (byte)((in[nOrigin + 2] & 0x03) << 6);
        out[nCode + 2] = (byte)((in[nOrigin + 2] >> 2) & 0xff);

        if((nOrigin + 3) >= nLen) break;
        out[nCode + 2] |= (byte)((in[nOrigin + 3] & 0x07) << 5);
        out[nCode + 3] = (byte)((in[nOrigin + 3] >> 3) & 0xff);

        if((nOrigin + 4) >= nLen) break;
        out[nCode + 3] |= (byte)((in[nOrigin + 4] & 0x0f) << 4);
        out[nCode + 4] = (byte)((in[nOrigin + 4] >> 4) & 0xff);

        if((nOrigin + 5) >= nLen) break;
        out[nCode + 4] |= (byte)((in[nOrigin + 5] & 0x1f) << 3);
        out[nCode + 5] = (byte)((in[nOrigin + 5] >> 5) & 0xff);

        if((nOrigin + 6) >= nLen) break;
        out[nCode + 5] |= (byte)((in[nOrigin + 6] & 0x3f) << 2);
        out[nCode + 6] = (byte)((in[nOrigin + 6] >> 6) & 0xff);

        if((nOrigin + 7) >= nLen) break;
        out[nCode + 6] |= (byte)((in[nOrigin + 7] & 0x7f) << 1);
```

```
        nCode += 7;
        nOrigin += 8;
    }
}
```

由程序代码可以看出，编码函数主要通过移位运算来完成编码。

```
//将字节处理成字符串
void ByteToChar(char nInPut[],char Out[],int nLen)
{
    int i;
    char chrTemp;

    for(i = 0;i < nLen;i++)
    {
        //高字节
        chrTemp = (char)((nInPut[i] >> 4) & 0x0f);
        if(chrTemp >= 0 && chrTemp <= 9) chrTemp += 48;
        else chrTemp += 55;
        Out[i] = chrTemp;

        //低字节
        chrTemp = (char)(nInPut[i] & 0x0f);
        if(chrTemp >= 0 && chrTemp <= 9) chrTemp += 48;
        else chrTemp += 55;
        Out[i] = chrTemp;
    }

    return;
}
```

在该函数中，主要需要考虑的是 16 进制的转换，所以当发现是字母的时候需要将值加上 55，而对于数字只需要加上 48。

### 4．短信息接收

根据前面对短信息读取的 AT 命令的分析可以看出，短信息的接收主要包括两个步骤，首先发送读取短信息的命令，然后 TC35 返回响应，对响应进行解析得到短信息的内容。下面给出具体的程序。

首先看短信息的读取命令的实现。

```
//短信息的读取命令
void revSms(char pBuf[],char pOut[])
{
    pBuf[0] = 'A';
    pBuf[1] = 'T';
    pBuf[2] = '+';
    pBuf[3] = 'C';
```

```
    pBuf[4] = 'M';
    pBuf[5] = 'G';
    pBuf[6] = 'R';
    pBuf[7] = '=';
    pBuf[8] = (char)((index >> 8) & 0xff + 0x30);
    pBuf[9] = (char)(index & 0xff + 0x30);
    pBuf[10] = 13;

    return 11;
}
```

在上面的程序封装了“AT+CMGR=index”命令，这里 index 为具体的位置，发送的时候必须将该整数值转换成字符形式。

在发送完了读取短信息的 AT 命令后，TC35 会给出响应，对响应进行解析就可以得到短信息的内容。下面为具体的程序实现。

```
int AnalyseSms(char in[],int nLen,char chrPhone[],char chrMessage[])
{
    char Phone[20];
    int phone_len;
    char chrTemp[200];
    char chrMessage[140];
    int nLen_temp;
    int nTempLen;
    int nOffset;
    int nOff;
    char chrTmp;
    int n;
    int nContent_Len;
    byte content[140];

    //去掉+CMGR:信息
    nLen_temp = nLen - 6;
    nOffset = 6;
    copy(chrTemp,0,in,nOffset,nLen_temp);

    if(nLen_temp < 20) return -1;
    chrTmp = chrTemp[0];
    if(chrTmp == '1')//新信息
    {
        chrTmp = strTemp[3];
        if(chrTemp == '1')//长度为3个字符
        {
        nTempLen = (strTemp[3] - 48) * 100 + (strTemp[4] - 48) * 10
                 + (strTemp[4] - 48);
        if(nLen_temp >= 26)
```

```
{
    nLen_temp -= 26;
    nOffset += 26;
    copy(chrTemp,0,in,nOffset,nLen_temp);
}
else return -1;
}//长度为两个字符
else
{
nTempLen = (strTemp[3] - 48) * 10 + (strTemp[4] - 48);
if(nLen_temp >= 25)
{
    nLen_temp -= 25;
    nOffset += 25;
    copy(chrTemp,0,in,nOffset,nLen_temp);
}
else return -1;
}
//取得电话号码
int nPhone_Len = 0;
if(nLen_temp >= 18)
{
    copy(Phone,0,chrTemp,0,18);
    phone_len = (int)(Phone[1] - 48);
    copy(chrPhone,0,Phone,4,14);
}
else return -1;
//获得电话号码的正确顺序
copy(Phone,0,chrPhone,0,14);
n = 0;
for(i = 0;i < 7;i++)
{
chrPhone[n++] = Phone[2 * i + 1];
chrPhone[n++] = Phone[2 * i];
}

if(nLen_temp >= 20)
{
    nLen_temp -= 20;
nOffset += 20;
copy(chrTemp,0,in,nOffset,nLen_temp);
}
else return -1;
if(nLen_temp >= 16)
{
```

```
    nLen_temp -= 16;
 nOffset += 16;
 copy(chrTemp,0,in,nOffset,nLen_temp);
}
else return -1;
//取出内容的长度
nContent_Len = 0;
if(nLen_temp >= 2)
{
    if((strTemp[1] >= 48) && (strTemp[1] <= 57))
        nContent_Len = (strTemp[0] - 48) * 16 + (strTemp[1] - 48);
    else if(strTemp[1] >= 65 && strTemp[1] <= 70)
        nContent_Len = (strTemp[0] - 48) * 16 + (strTemp[1] - 55);
    else if(strTemp[1] >= 97 && strTemp[1] <= 102)
        nContent_Len = (strTemp[0] - 48) * 16 + (strTemp[1] - 87);
}
else return -1;
if(strTemp.length() >= 2)
{
    nLen_temp -= 2;
 nOffset += 2;
 copy(chrTemp,0,in,nOffset,nLen_temp);
}
        else return -1;

        CharToByte(strTemp,content,nLen_temp);
        nLen_temp /= 2;
        n = Decode(content,chrMessage,nLen_temp);
    }
    return n;

}
```

函数如果成功的话，返回消息内容的长度，如果不成功的话，则返回“-1”。in[]为接收到的响应数据流。nLen 为数据的长度。chrPhone[]为解析得到的电话号码。chrMessage[]为解析得到的消息内容。在函数中，需要对得到的电话号码进行高低字节互换，从而得到电话号码的正确形式。在解析短信息的内容的时候，需要将得到的字符流转换成字节流，然后对字节流进行解码，最后得到短信息的内容。对于接收到的 PDU 的具体帧结构如图 10-11 所示。具体的数据解析参看程序，并参看 PDU 的格式。

| SMSC | PDU 类型 | OA | PID | DCS | SCTS | UDL | UD |
|---|---|---|---|---|---|---|---|

图 10-11　接收短信息的 PDU 的帧结构

在 PDU 数据包的帧结构中，“SMSC”字段为短信息中心的地址；“PDU 类型”来指明数

据包的类型；“OA”为源地址；“PID”为协议识别号；“DCS”为短信息的编码格式，对于数字或者字符采用编码值为“00”，如果内容是汉字的话，则采用的编码值为“08”，采用的是 UNICODE 编码方式，在该系统中采用的字符编码方式；“SCTS”表示短信息到达业务中心的时间；“UDL”表示数据内容的长度；“UD”为具体的短信息内容。在 PDU 数据包里面，所有的内容是以字符形式存在的。

下面给出具体的解码函数和字节流转换字符流函数，具体函数如下：

```
void CharToByte(char in[],byte out[],int nLen)
{
    char chrHi,chrLow;

    for(int i = 0;i < nLen / 2;i++)
    {
        chrHi = in[2 * i];
        if(chrHi >= 48 && chrHi <= 57)
            chrHi = (char)(chrHi - 48);
        else if(chrHi >= 65 && chrHi <= 70)
            chrHi = (char)(chrHi - 55);
        else if(chrHi >= 97 && chrHi <= 102)
            chrHi = (char)(chrHi - 87);
        chrLow = in[2 * i + 1];
        if(chrLow >= 48 && chrLow <= 57)
            chrLow = (char)(chrLow - 48);
        else if(chrLow >= 65 && chrLow <= 70)
            chrLow = (char)(chrLow - 55);
        else if(chrLow >= 97 && chrLow <= 102)
            chrLow = (char)(chrLow - 87);

        out[i] = (byte)(chrHi * 16 + chrLow);
    }

    return;
}
```

上面的程序主要是将字符的 ASCII 值减去相应的值得到字节数据。

```
int Decode(char in[],char out[],int nLen)
{
    int nLenTemp = byteStream.length;
    int nOrigin = 0;
    int nCode = 0;
    nLenTemp = nLen * 8 / 7;

    while(true)
    {
```

```
if(nOrigin >= nLen) break;
out[nCode] = (char)(in[nOrigin] & 0x7f);
if((nOrigin + 1) >= nLen) break;
out[nCode + 1] = (char)((in[nOrigin + 1] & 0x3f) << 1);
out[nCode + 1] += (char)((in[nOrigin] & 0x80) >> 7);
if((nOrigin + 2) >= nLen) break;
out[nCode + 2] = (char)((in[nOrigin + 2] & 0x1f) << 2);
out[nCode + 2] += (char)((in[nOrigin + 1] & 0xc0) >> 6);
if((nOrigin + 3) >= nLen) break;
out[nCode + 3] = (char)((in[nOrigin + 3] & 0x0f) << 3);
out[nCode + 3] += (char)((in[nOrigin + 2] & 0xe0) >> 5);
if((nOrigin + 4) >= nLen) break;
out[nCode + 4] = (char)((in[nOrigin + 4] & 0x07) << 4);
out[nCode + 4] += (char)((in[nOrigin + 3] & 0xf0) >> 4);
if((nOrigin + 5) >= nLen) break;
out[nCode + 5] = (char)((in[nOrigin + 5] & 0x03) << 5);
out[nCode + 5] += (char)((in[nOrigin + 4] & 0xf8) >> 3);
if((nOrigin + 6) >= nLen) break;
out[nCode + 6] = (char)((in[nOrigin + 6] & 0x01) << 6);
out[nCode + 6] += (char)((in[nOrigin + 5] & 0xfc) >> 2);
out[nCode + 7] = (char)((in[nOrigin + 6] & 0xfe) >> 1);
nCode += 8;
nOrigin += 7;
}
return nLenTemp;
}
```

上面的程序主要完成 8 位的字节转换成 ASCII 字符。

**5．短信息的删除**

一般来说，手机的存储量或者 SIM 卡的存储量是有限的，因此有时需要删除已经阅读过的短信息。短信息的删除操作主要是发送删除的 AT 命令来实现，具体的程序如下：

```
//删除短信息
int deleteSms(char pBuf[],short int index)
{
    pBuf[0] = 'A';
    pBuf[1] = 'T';
    pBuf[2] = '+';
    pBuf[3] = 'C';
    pBuf[4] = 'M';
    pBuf[5] = 'G';
    pBuf[6] = 'D';
    pBuf[7] = '=';
    pBuf[8] = (char)((index >> 8) & 0xff + 0x30);
    pBuf[9] = (char)(index & 0xff + 0x30);
```

```
    pBuf[10] = 13;
    return 11;
}
```

在上面的程序封装了“AT+CMGD=index”命令，这里 index 为具体的位置，发送的时候必须将该整数值转换成字符形式。

### 10.3.3　系统软件流程

经过前面对 AT 命令和短信息操作实现的介绍，对整个软件系统的实现有一定的了解。现在具体讨论整个系统软件的实现。

整个软件系统主要实现 TC35 的短信息发送，另外还通过与上位机的通信来实现一些简单的配置管理。整个系统软件实现了一个简单的短信息传输数据系统，另外也包括一些辅助的配置管理，整个程序的流程图如图 10-12 所示。

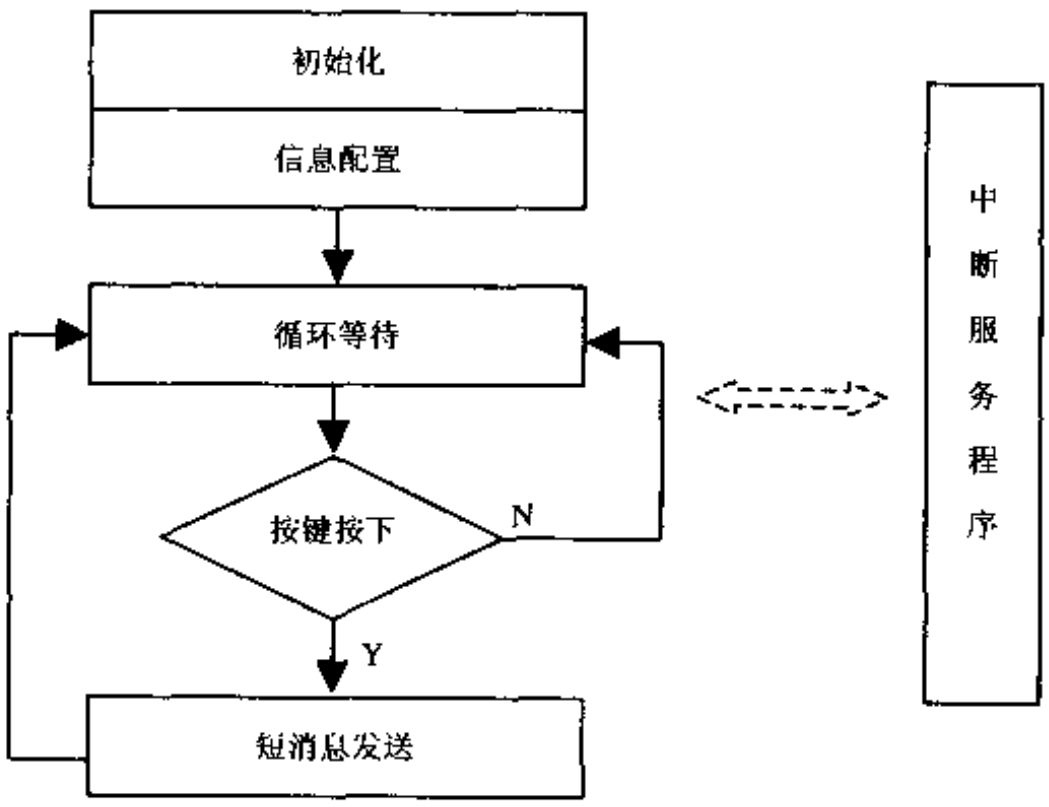



图 10-12　系统软件流程图

由图 10-12 可以看出，整个系统软件基于中断服务机制，主处理需要与中断进行数据交互，具体的实现是通过全局的标志变量和全局的数据缓冲区来实现的。整个软件系统包括上位机通信部分、按键处理部分、短信息操作部分和主处理部分。其中短信息操作部分在前面一小节已经做了具体的介绍，这里再进行详细的介绍，下面就其他各个部分进行具体介绍。

#### 1．上位机通信部分

上位机通信部分主要是完成单片机接收来自上位机的配置数据，利用单片机的串口 1 实现，主要包括串口初始化和串口中断服务程序。下面具体分析程序。

串口 1 的初始化程序：

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

```
    U1TCTL = 0X00;   //将寄存器的内容清零
    U1TCTL += SSEL1;//波特率发生器选择 SMCLK

    UBR0_1 = 0X45;   //波特率为 115200
    UBR1_1 = 0X00;
    UMCTL_1 = 0X00;

    ME2 |= UTXE1 + URXE1; //使能 UART1 的 TXD 和 RXD
    IE2 |= URXIE1;   //使能 UART1 的 RX 中断
    IE2 |= UTXIE1;   //使能 UART1 的 TX 中断

    P3SEL |= BIT6;   //设置 P3.6 为 UART1 的 TXD
    P3SEL |= BIT7;   //设置 P3.7 为 UART1 的 RXD

    P3DIR |= BIT6;   //P3.6 为输出管脚
    return;
}
```

上面的程序主要对串口的几个寄存器进行正确的设置，这样就能使串口工作起来。串口 1 的中断服务程序，主要包括接收和发送的中断服务程序，具体程序如下：

```
//////////////////////////////////////////
//处理来自串口 1 的接收中断
interrupt [UART1RX_VECTOR] void UART1_RX_ISR(void)
{

    UART1_RX_BUF[nRX1_Len_temp] = RXBUF1;   //接收来自的数据

    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;
    }
}
//////////////////////////////////////////
//处理来自串口 1 的发送中断
interrupt [UART1TX_VECTOR] void UART1_TX_ISR(void)
{
    if(nTX1_Len != 0)
    {
     nTX1_Flag = 0;          //表示缓冲区里的数据没有发送完
```

```
        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，在接收到数据后设置一个标志来通知主程序。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，则进入发送中断服务程序，发送完数据后，发送中断程序等待下一次中断的到来。

**2．按键处理部分**

该部分程序主要是处理按键。当如果有按键按下的时候，设置标志通知主程序。由于 P1.5 管脚可以设置成中断方式，因此这里采用中断服务程序的机制，当有按键按下的时候，则进入中断服务程序，中断服务程序设置一个标志通知主程序有按键按下，以便主程序进行短信息发送。下面是中断服务程序的程序代码。

```
//////////////////////////////////////////
//处理来自端口 1 的中断
interrupt [PORT1_VECTOR] void COMM_ISR(void)
{
    if(P1IFG & BIT5)
    {
     nComm = 1;
     P1IFG &= ~(BIT5);   //清除中断标志位
     Delay_us(100);
    }
}
```

在中断服务程序中，设置“nComm = 1”来通知主程序有按键按下。

**3．主处理部分**

主处理模块主要是将各个模块进行协调处理和实现数据交互。主处理模块首先完成初始化工作，初始化后等待配置信息，配置完必须信息后，进入处理循环。在循环过程中主处理扫描是否有按键按下，如果有按键按下，则发送短信息，如果没有按键按下则继续循环。另外主程序还需要与两个串口的发送和接收中断服务程序进行数据交互。整个主程序基于中断服务结构，为了实现中断程序与主程序之间的数据交互，通过设置一些全局变量和全局的缓冲区来实现。下面给出程序的具体实现。

```
#include <MSP430X14X.h>
```

```
#include "uart.h"
#include "TC35.h"


//定义全局变量
static char nComm;
//定义串口操作变量
char nRev_UART0;    //串口0的接收标志
char nRev_UART1;    //串口1的接收标志
char UART0_TX_BUF[200]; //串口0的发送缓冲区
char UART0_RX_BUF[200]; //串口0的接收缓冲区
char UART1_TX_BUF[50];  //串口1的发送缓冲区
char UART1_RX_BUF[50];  //串口1的接收缓冲区
char pBuf0[100];
static int nTX1_Len;
static char nRX1_Len;
char nRX1_Len_temp;
static int nTX0_Len;
static int nRX0_Len;
int nRX0_Len_temp;
static char nTX0_Flag;
static char nTX1_Flag;
int nSend_TX0;
int nSend_TX1;

void main(void)
{

   int j;
   int n;
   int nTemp;
   int nLen1;
   int nLen2;
   char nRes_UART1;
   char nRes_UART0;
   char PhoneNumber[18];
   char UART1_RX_Temp[50];
   char UART0_RX_Temp[20];
   char pOut1[40];
   char pOut2[200];
   char nSend;
   int nPhone;

   WDTCTL = WDTPW + WDTHOLD;   //关闭看门狗
```

```
_DINT();          //关闭中断

nSend_TX1 = 0;
nSend_TX0 = 0;
nTX1_Flag = 0;
nTX0_Flag = 0;
nTX0_Len = 0;
nTX1_Len = 0;
nRX1_Len = 0;
nRX0_Len = 0;
nRev_UART1 = 0;
nRev_UART0 = 0;
nPhone = 0;
nLen1 = 0;
nLen2 = 0;
nComm = 0;
nSend = 0;
////////////////////////////////
//初始化
Init_CLK();
Init_UART0();
Init_UART1();

_EINT();//打开中断

    //TC35 初始化
    tc35_init();
    Delay_ms(100);
    nTX0_Len = setCsca(UART0_TX_BUF);
    IFG1 |= UTX1FG0;//设置中断标志，进入发送中断程序
    Delay_ms(500);
    nTX0_Len = setCmgf(UART0_TX_BUF);
    IFG1 |= UTXIFG0;//设置中断标志，进入发送中断程序
    Delay_ms(500);
    //等待配置
    for(;;)
    {
     if(nRev_UART1 == 1)
     {
            nRev_UART1 = 0;
           for(i = 0;i < nRX1_Len;i++)
            {
                UART1_RX_Temp[i] = UART1_RX_BUF[i];
            }
            //获得电话号码
```

```
            nPhone = SetPhone(UART1_RX_Temp,PhoneNumber,nRX1_Len);
            if(nPhone != 0)
            {
             nTX1_Len = SetOK(UART1_TX_BUF);
             IFG2 |= UTXIFG1;//设置中断标志，进入发送中断程序
             break;
            }
            else
            {
             nTX1_Len = SetError(UART1_TX_BUF);
             IFG2 |= UTXIFG1;//设置中断标志，进入发送中断程序
            }

 }
}
//循环处理
for(;;)
{
 if(nComm == 1)
 {
    nComm = 0;
    for(i = 0;i < 50;i++)
    {
     pBuf[i] = i;
    }
    sendSms(PhoneNumber,nPhone,pBuf,50,&nLen1,&nLen2,pOut1,pOut2);
    for(i = 0;i < nLen1;i++)
    {
     UART0_TX_BUF[i] = pOut1[i];
    }
    nTX0_Len = nLen1;
    IFG1 |= UTXIFG0;//设置中断标志，进入发送中断程序
    //等待">"
    for(;;)
    {
     if(nRev_UART0 == 1)
     {
        nRev_UART0 = 0;
        for(i = 0;i < nRX0_Len;i++)
              {
                  UART0_RX_Temp[i] = UART0_RX_BUF[i];
              }
        if(nRX0_Len >= 2)
        {
         if((UART0_RX_Temp[0] == 62)
```

```
        && (UART0_RX_Temp[1] == 32))
      {
          nSend = 1;
          break;
      }
     }
     else
     {
      nSend = 0;
      break;
     }

 }

}
for(i = 0;i < nLen2;i++)
{
 UART0_TX_BUF[i] = pOut2[i];
}
nTX0_Len = nLen2;
IFG1 |= UTXIFG0;//设置中断标志，进入发送中断程序
        Delay_ms(10000);
      }

    }
}
```

上面为主处理的程序，该部分为一个可伸缩性的框架，可以在此基础上进一步丰富处理的功能。主程序先进行初始化，先初始化时钟和两个串口，然后等待配置信息，直到配置完成后才进入处理循环。在处理循环里面检测通信按键是否按下，如果按下就发送短信息，如果没有按下则继续等到。在发送短信息的时候，先发送头信息数据，然后等待TC35返回“>”，等到“>”返回后则继续发送短信息内容，在发送短信息完后，需要延时10秒以上。一般说来发送的短信息与短信息之间需要有时间间隔，一般都需要几十秒的时间间隔。主程序需要和串口通信程序进行数据交互。当需要向串口发送数据的时候，往发送的数据缓冲区里面封装数据，设置发送的数据长度，设置中断标志后就触发中断服务程序，中断服务程序就发送数据，直到数据发送完毕后清除相应的标志位。主程序还需要检测串口的接收中断服务程序，当串口有新数据到来时，串口接收中断服务程序将收到的数据填入到接收缓冲区里，设置接收到的长度，设置接收标志来通知主程序取数据。主程序与中断服务程序进行数据交互参看具体用到的全局标志变量和全局数据缓冲区。

在该主处理程序中只是一个简单的示范，还需要在此基础上增加对消息发送的进一步处理，也需要增加短信息的阅读、删除等操作，关于这些操作，这里不再进行详细讨论，可以结合前面提供的函数，自行实现短信息的阅读和删除等操作。

## 10.4 系统调试

前面已经介绍了整个系统的硬件设计和软件设计，该部分结合前面介绍的来讨论整个系统的联调。系统的调试包括硬件调试和软件调试，下面分别进行介绍。

**1．系统硬件调试**

系统的硬件调试很简单，先调试电源电路和复位电路，只要这两个部分能正常工作，再进行单片机的调试，如果单片机的晶振能起振的话，则整个硬件的单片机部分没有问题。关于硬件系统的串口通信和 TC35 模块通信部分则需要结合软件进行调试。

**2．系统软件调试**

前面分别介绍了各个软件部分，为了能够进行整个系统的联调，需要软件和硬件调试结合起来，对于不同的硬件部分，应该调用不同的软件模块进行测试。经过联合调试，整个系统的软件和硬件能够正确运行。

## 10.5 实例总结

该系统通过 TC35 模块实现简单的短信息传输的系统具有设计简单、运行可靠等特点。通过软件测试和硬件测试证明该系统能够安全可靠的运行。本系统采用 MSP430F149 实现的数据传输系统具有一定的通用性，它通过一个 TC35 无线 GSM 模块与单片机的 UART 进行连接，由于系统可以通过上位机进行配置，这样使该系统具有很大的通用性。该系统具有很大的伸缩性，单片机与上位机通信部分可以扩充来实现更加丰富的功能。虽然本章介绍的系统相对简单，可以在 TC35 模块基础上增加语音电路的处理，这样还可以实现语音传输，也可以采用数据传输的 AT 命令实现数据传输，这样完全可以在该系统的基础上增加硬件和软件实现更加丰富的功能。在附录中给出前面所调用到的一些函数。

### 附录：其他程序模块

```
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
```

```
    BCSCTL1 &= ~(XT2OFF + XTS);//open XT2, LFTX2 select low frequency
    BCSCTL1 |= RSEL0 + RSEL1 + RSEL2; //DCO Rsel=7(Freq=3200k/25 摄氏度)
    BCSCTL1 |= 0x07;
    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

}
int FindERROR(char in[],int nLen)
{
    int nOffset,i;
    nOffset = -1;if(nLen < 5) return nOffset;
    for(i = 0;i < nLen;i++)
    {
     if((in[i] == 'R') && (in[i - 1] == 'O') && (in[i - 2] == 'R')
       && (in[i - 3] == 'R') && (in[i - 4] == 'E'))
     {
         nOffset = i - 4;
         break;
     }
    }
    return nOffset;
}
int FindCMGR(char in[],int nLen)
{
    int nOffset,i;
    nOffset = -1;
    if(nLen < 5) return nOffset;
    for(i = 0;i < nLen;i++)
    {
     if((in[i] == 'R') && (in[i - 1] == 'G') && (in[i - 2] == 'M')
       && (in[i - 3] == 'C') && (in[i - 4] == '+'))
     {
         nOffset = i - 4;
         break;
     }
    }
    return nOffset;
}
int FindOK(char in[],int nLen)
{
    int nOffset,i;
    nOffset = -1;
    if(nLen < 2) return nOffset;
    for(i = 0;i < nLen;i++)
    {
```

```
        if((in[i] == 'K') && (in[i - 1] == 'K'))
        {
            nOffset = i - 1;
            break;
        }
        }
        return nOffset;
}
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;
}
void Delay_us(unsigned long nValue)//微秒为单位，8MHz为主时钟
{
        int nCount;
        int i;
        int j;
        nCount = 3;
        for(i = nValue;i > 0;i--)
        {
         for(j = nCount;j > 0;j--);
        }
        return;
}
int SetOK(char UART1_TX_BUF[])
{
        UART1_TX_BUF[0] = 'O';
        UART1_TX_BUF[1] = 'K';
        UART1_TX_BUF[2] = 13;

        return 3;
}
int SetError(char UART1_TX_BUF[])
{
        UART1_TX_BUF[0] = 'E';
        UART1_TX_BUF[1] = 'R';
        UART1_TX_BUF[2] = 'O';
```

```
    UART1_TX_BUF[3] = 'R';
    UART1_TX_BUF[4] = 'R';
    UART1_TX_BUF[5] = 13;

    return 6;
}
int SetPhone(char recv[],char phone[],int nLen)
{
    int i;
    char chrHi,chrLo;
    chrHi = recv[0];//长度
    chrLo = recv[1];
    if(recv[1] == ',')
    {
        for(i = 0;i < chrHi;i++)
        {
        phone[i] = recv[i + 2];
        }
    }
    else return 0;

    return chrHi;
}
//将源数组的内容拷贝到目的数组
void copy(char pDest[],int nOrg,char pOrg[],int nStart,int nLen)
{
    int i;
    for(i = 0;i < nLen;i++)
    {
    pDest[nOrg + i] = pOrg[i + nStart];
    }
}
//数的范围为 250 以内
int IntToChar(int n,char Out[])
{
    int i;
    char chrTemp1;
    char chrTemp2;
    char chrTemp3;
    int nLen;
    chrTemp1 = n / 100;
    chrTemp2 = (n - chrTemp1 * 100) / 10;
    chrTemp3 = n - chrTemp1 * 100 - chrTemp2 * 10;
    if(chrTemp1 != 0)
    {
```

```
        nLen = 3;
           Out[0] = chrTemp1 + 0x30;
           Out[1] = chrTemp2 + 0x30;
           Out[2] = chrTemp3 + 0x30;
        }
        else
        {
        if(chrTemp2 != 0)
        {
            nLen = 2;
               Out[0] = chrTemp2 + 0x30;
               Out[1] = chrTemp3 + 0x30;
        }
        else
        {
            nLen = 1;
               Out[0] = chrTemp3 + 0x30;
        }
        }
        return nLen;
    }
```

[General Information]
书名=MSP430单片机就任系统开发典型实例
作者=
页数=334
SS号=0
出版日期=