

单片机 C 语言程序设计实训 100 例

——基于 AVR+Proteus 仿真

彭 伟 编著

北京航空航天大学出版社



内 容 简 介

基于 AVR Studio+WinAVR(GCC)组合环境和 Proteus 硬件仿真平台,精心安排了 100 个 AVR 单片机 C 程序设计案例。全书提供了所有案例完整的 C 语言源程序,各案例设计了难易适中的实训目标。

基础设计类案例涵盖 AVR 单片机最基本的端口编程、定时/计数器应用、中断程序设计、A/D 转换、比较器程序设计、EEPROM、Flash、USART 及看门狗程序设计;硬件应用类案例涉及单片机存储器扩展、接口扩展、译码、编码、驱动、光电、机电、传感器、I²C/TWI 及 SPI 接口器件、MMC、红外等器件;综合设计类案例涉及消费类电子产品、仪器仪表及智能控制设备相关技术,相关案例涉及 485 及 RTL8019 的应用。

本书适合用作大专院校学生学习实践 AVR 单片机 C 语言程序设计技术的参考书,也可用作电子工程技术人员、单片机技术爱好者的学习参考书。

图书在版编目(CIP)数据

单片机 C 语言程序设计实训 100 例:基于
AVR + Proteus 仿真/彭伟编著. —北京: 北京航空航天
大学出版社, 2010. 5

ISBN 978 - 7 - 5124 - 0068 - 9

I. ①单… II. ①彭…… III. ①单片微型计算机—
C 语言—程序设计 IV. ①TP368. 1②TP312

中国版本图书馆 CIP 数据核字(2010)第 068448 号

版权所有,侵权必究。

单片机 C 语言程序设计实训 100 例——基于 AVR + Proteus 仿真

彭 伟 编著

责任编辑 冯 颖

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(邮编 100191) <http://www.buaapress.com.cn>

发行部电话:(010)82317024 传真:(010)82328026

读者信箱:bhpress@263.net 邮购电话:(010)82316936

北京时代华都印刷有限公司印装 各地书店经销

*

开本: 787×1 092 1/16 印张:36 字数:922 千字

2010 年 5 月第 1 版 2010 年 5 月第 1 次印刷 印数: 4 000 册

ISBN 978 - 7 - 5124 - 0068 - 9 定价:65.00 元

前言

目前,各高校电类专业都将 C 语言作为专业基础课程纳入教学计划。由于 C 语言功能强大、便于模块化开发、所带库函数非常丰富、编写的程序易于移植,因此,它成为单片机应用系统开发最快速高效的程序设计语言。仅具有 C 语言基础知识但不熟悉单片机指令系统的读者也能很快掌握单片机 C 程序设计技术,C 语言在单片机应用系统设计上的效率优势已经远远高于汇编、BASIC 等开发语言。

单片机 C 程序设计不同于通用计算机应用程序设计,它必须针对具体的微控制器及外围电路来完成。为便于学习单片机应用程序设计和系统开发,很多公司推出了单片机实验箱、仿真器和开发板等,这些硬件设备可用于验证单片机程序、开发和调试单片机应用系统。但由于这些设备价格不菲,它们阻碍了普通读者对单片机技术的学习和研究。令人高兴的是,英国 Labcenter 公司推出了具有单片机系统仿真功能的 Proteus 软件,单片机系统开发通常是基于上位机加目标系统进行的,Proteus 的出现使读者仅用一台 PC 在纯软件环境中完成系统设计与调试成为可能。目前 Proteus 支持 8051、AVR、PIC 等多种单片机,系统库中包含有大量的模拟、数字、光电和机电类元器件,系统还提供了多种虚拟仪器,用 AVR Studio + WinAVR (GCC) 开发的程序可以在用 Proteus 设计的仿真电路中调试和交互运行。这无疑为读者学习和提高 AVR 单片机 C 程序设计技术,为单片机应用系统高水平工程师的成长提供了理想平台。

为帮助读者快速提高 AVR 单片机 C 程序设计水平,本书基于 AVR Studio + WinAVR (GCC) 组合开发环境和 Labcenter 公司的 Proteus 仿真平台,精心安排了 100 个 AVR 单片机 C 程序设计案例,各案例同时给出了难易适中的实训目标。

前 2 章分别对 AVR - GCC 程序设计和 Proteus 操作基础作了概述。第 3 章基础程序部分给出的案例涵盖 AVR 单片机端口编程、定时/计数器应用、A/D 转换、模拟比较器程序设计、中断程序设计、EEPROM、Flash、USART 及看门狗程序设计,各案例分别对相关知识和技术要点作了阐述与分析,源程序中也给出了丰富的注释信息。第 4 章硬件应用部分针对 AVR 单片机的存储器扩展、接口扩展、译码、编码、驱动、光电、机电、传感器、I²C/TWI 及 SPI 接口器件、MMC、红外等器件给出了数十个案例,对案例中涉及的硬件技术资料亦进行了有针对性的分析,以便于读者快速理解相关代码的编写原理。第 5 章的案例综合应用了单片机内部资源和外部扩展硬件,通过对这些案例的独立分析研究与调试运行,读者用 C 语言开发 AVR 单片机应用系统的能力会得到大幅提升。

本书是单片机 C 语言程序设计实训仿真系列 8051 版之后的第 2 册。为使本书能早日与读者见面,笔者坚持挤出时间不懈耕耘。在编写过程中,刘静、张力、王魏参与了案例的调试与校稿工作,在此对他们深表感谢!本书从选题、撰稿到出版的全过程中,学院领导、学院科研处及高教研究所对本选题始终给予大力支持,并提供项目资助,教务处和信息技术系也一直关注本书的编

写与进展情况,在此一并对学院和部门领导的关心与支持表示由衷感谢!

本书提供完整的案例压缩包,需要的读者可到北京航空航天大学出版社网站 <http://www.buaapress.com.cn> 的“下载中心”免费下载。

由于编者水平有限,加之时间仓促,书中错漏之处在所难免,在此真诚欢迎读者对本书多多提出宝贵意见(笔者的邮箱是:pw95aaa@foxmail.com)。

至此,本套单片机 C 语言程序设计实训仿真系列的 8051 版与 AVR 版已经编写完成,PIC 版正在后续编撰之中,笔者将努力争取使之早日出炉,以飨读者。

彭伟

2010 年 3 月于武昌



录

第 1 章 AVR 单片机 C 语言程序设计概述	1
1.1 AVR 单片机简介	1
1.2 AVR Studio+WinAVR 开发环境安装及应用	4
1.3 AVR-GCC 程序设计基础	7
1.4 程序与数据内存访问	14
1.5 I/O 端口编程	14
1.6 外设相关寄存器及应用	16
1.7 中断服务程序	31
1.8 GCC 在 AVR 单片机应用系统开发中的优势	33
第 2 章 Proteus 操作基础	35
2.1 Proteus 操作界面简介	35
2.2 仿真电路原理图设计	37
2.3 元件选择	39
2.4 仿真运行	44
2.5 Proteus 与 AVR Studio 的联合调试	45
2.6 Proteus 在 AVR 单片机应用系统开发中的优势	46
第 3 章 基础程序设计	48
3.1 闪烁的 LED	48
3.2 左右来回的流水灯	50
3.3 花样流水灯	52
3.4 LED 模拟交通灯	54
3.5 单只数码管循环显示 0~9	57
3.6 8 只数码管滚动显示单个数字	59
3.7 8 只数码管扫描显示多个不同字符	61
3.8 K1~K4 控制 LED 移位	62



3.9 数码管显示 4×4 键盘矩阵按键	65
3.10 数码管显示拨码开关编码	68
3.11 继电器控制照明设备	70
3.12 开关控制报警器	72
3.13 按键发音	74
3.14 INT0 中断计数	76
3.15 INT0 与 INT1 中断计数	79
3.16 TIMER0 控制单只 LED 闪烁	83
3.17 TIMER0 控制流水灯	85
3.18 TIMER0 控制数码管扫描显示	87
3.19 TIMER1 控制交通指示灯	90
3.20 TIMER1 与 TIMER2 控制十字路口秒计时显示屏	94
3.21 用工作于计数方式的 T/C0 实现 100 以内的脉冲或按键计数	98
3.22 用定时器设计的门铃	100
3.23 报警器与旋转灯	103
3.24 100000 s 以内的计时程序	106
3.25 用 TIMER1 输入捕获功能设计的频率计	109
3.26 用工作于异步模式的 T/C2 控制的可调式数码管电子钟	113
3.27 TIMER1 定时器比较匹配中断控制音阶播放	117
3.28 用 TIMER1 输出比较功能调节频率输出	120
3.29 TIMER1 控制的 PWM 脉宽调制器	123
3.30 数码管显示两路 A/D 转换结果	126
3.31 模拟比较器测试	128
3.32 EEPROM 读/写与数码管显示	130
3.33 Flash 程序空间中的数据访问	136
3.34 单片机与 PC 机双向串口通信仿真	141
3.35 看门狗应用	147
第 4 章 硬件应用	150
4.1 74HC138 与 74HC154 译码器应用	150
4.2 74HC595 串入并出芯片应用	153
4.3 用 74LS148 与 74LS21 扩展中断	157
4.4 62256 扩展内存实验	160
4.5 用 8255 实现接口扩展	163
4.6 可编程接口芯片 8155 应用	168
4.7 可编程外围定时/计数器 8253 应用	173
4.8 数码管 BCD 解码驱动器 7447 与 4511 应用	178
4.9 8×8 LED 点阵屏显示数字	181
4.10 8 位数码管段位复用串行驱动芯片 MAX6951 应用	183

4.11	串行共阴显示驱动器 MAX7219 与 7221 应用	188
4.12	16 段数码管演示	193
4.13	16 键解码芯片 74C922 应用	196
4.14	1602 LCD 字符液晶测试程序	199
4.15	1602 液晶显示 DS1302 实时时钟	205
4.16	1602 液晶工作于 4 位模式实时显示当前时间	211
4.17	2×20 串行字符液晶演示	214
4.18	LGM12864 液晶显示程序	217
4.19	PG160128A 液晶图文演示	226
4.20	TG126410 液晶串行模式显示	247
4.21	用带 SPI 接口的 MCP23S17 扩展 16 位通用 I/O 端口	257
4.22	用 TWI 接口控制 MAX6953 驱动 4 片 5×7 点阵显示器	262
4.23	用 TWI 接口控制 MAX6955 驱动 16 段数码管显示	266
4.24	用 DAC0832 生成多种波形	270
4.25	用带 SPI 接口的数/模转换芯片 MAX515 调节 LED 亮度	273
4.26	正反转可控的直流电机	276
4.27	正反转可控的步进电机	279
4.28	DS18B20 温度传感器测试	282
4.29	SPI 接口温度传感器 TC72 应用测试	293
4.30	SHT75 温、湿度传感器测试	299
4.31	用 SPI 接口读/写 AT25F1024	309
4.32	用 TWI 接口读/写 24C04	318
4.33	MPX4250 压力传感器测试	326
4.34	MMC 存储卡测试	329
4.35	红外遥控发射与解码仿真	340
第 5 章	综合设计	348
5.1	多首电子音乐的选播	348
5.2	电子琴仿真	353
5.3	普通电话机拨号键盘应用	357
5.4	1602 LCD 显示仿手机键盘按键字符	363
5.5	数码管模拟显示乘法口诀	369
5.6	用 DS1302 与数码管设计的可调电子钟	372
5.7	用 DS1302 与 LGM12864 设计的可调式中文电子日历	380
5.8	用 PG12864LCD 设计的指针式电子钟	393
5.9	高仿真数码管电子钟	401
5.10	1602 LCD 显示的秒表	409
5.11	用 DS18B20 与 MAX6951 驱动数码管设计的温度报警器	413
5.12	用 1602 LCD 与 DS18B20 设计的温度报警器	421



5.13	温控电机在 L298 驱动下改变速度与方向运行	431
5.14	PG160128 中文显示日期时间及带刻度显示当前温度	439
5.15	液晶屏曲线显示两路 A/D 转换结果	447
5.16	用 74LS595 与 74LS154 设计的 16×16 点阵屏	452
5.17	用 8255 与 74LS154 设计的 16×16 点阵屏	457
5.18	8×8 LED 点阵屏仿电梯数字滚动显示	461
5.19	用内置 EEPROM 与 1602 液晶设计的带 MD5 加密的电子密码锁	466
5.20	12864LCD 显示 24C08 保存的开机画面	480
5.21	12864LCD 显示 EPROM27C256 保存的开机画面	488
5.22	I ² C—AT24C1024×2 硬字库应用	491
5.23	SPI—AT25F2048 硬件字库应用	498
5.24	带液晶显示的红外遥控调速仿真	505
5.25	能接收串口信息的带中英文硬字库的 80×16 点阵显示屏	511
5.26	用 AVR 与 1601 LCD 设计的计算器	523
5.27	电子秤仿真设计	531
5.28	模拟射击训练游戏	537
5.29	PC 机通过 485 远程控制单片机	546
5.30	用 IE 访问 AVR+RTL8019 设计的以太网应用系统	550
	参考文献	568



AVR 单片机 C 语言程序设计概述

1997 年,美国 Atmel 公司将其先进的 Flash 技术与 8051 单片机结合起来,推出了 8 位 AVR 单片机。与传统的采用复杂指令系统(CISC)的 8051 单片机不同的是,AVR 单片机采用了更适合中高档电子产品和嵌入式系统应用需求的精简指令系统(RISC)。

为适应不同层次与不同场合的应用需要,Atmel 公司推出了多个系列的 AVR 单片机,它们广泛应用于工控系统、计算机外部设备、通信设备、仪器仪表及应用类电子产品。本书所有案例使用的是 ATmega 系列单片机,涉及的型号有 ATmega8515/8535、ATmega8/16/32/64/128。

作为 Atmel 公司免费推出的 AVR 单片机开发平台,AVR Studio 提供了编写和调试 AVR 单片机应用程序的集成开发环境。它为源程序编写、设备编程、仿真及片上调试提供一个项目管理工具、源代码编辑器、模拟器、集成环境和前端。AVR Studio 支持全部的 Atmel AVR 工具集,每个版本总是包含有最新的 AVR 器件和工具支持。

但是,当前版本的 AVR Studio 仅提供汇编语言编译器,并未提供 C/C++ 程序编译器。要在 AVR Studio 集成环境中开发 AVR 单片机 C 程序,显然还需要安装配置第三方提供的 C/C++ 语言程序编译器。本书案例开发所使用的 WinAVR 是一套用于 Atmel AVR 系列 RISC 微处理器程序开发的开源的软件开发工具集,它以著名的自由软件 GCC 为 C/C++ 编译器。

使用 AVR Studio 提供的程序开发与调试前端及 WinAVR 提供的 AVR-GCC 编译程序组合搭建的 AVR 单片机 C 语言程序开发平台,可大大降低开发成本,缩短开发周期,大幅提高开发效率,程序可读性好且易于移植。

本书的编写,旨在进一步提高读者的 AVR 单片机 C 语言程序开发能力,全书的 100 个案例全部在 AVR Studio+WinAVR 环境下编写并调试通过,同时给出了完整 Proteus 仿真电路。当前版本的 AVR Studio 已经支持内嵌 Proteus 进行跟踪调试与仿真。

阅读使用本书之前要求读者已经学习了基本的 AVR 单片机 C 程序设计技术,本章仅介绍使用 C 语言设计单片机应用系统必须参考和重点掌握的技术内容,这些内容会对阅读、调试、研究本书案例及进行设计实训提供重要参考。

1.1 AVR 单片机简介

本书案例使用的 AVR 单片机有 ATmega8515/8535、ATmega8/16/32/64/128,图 1-1 给出了全书案例中出现最多的 ATmega16(L) 单片机的不同封装形式及引脚分布,本节仅对该单片机端口及引脚作简要介绍。

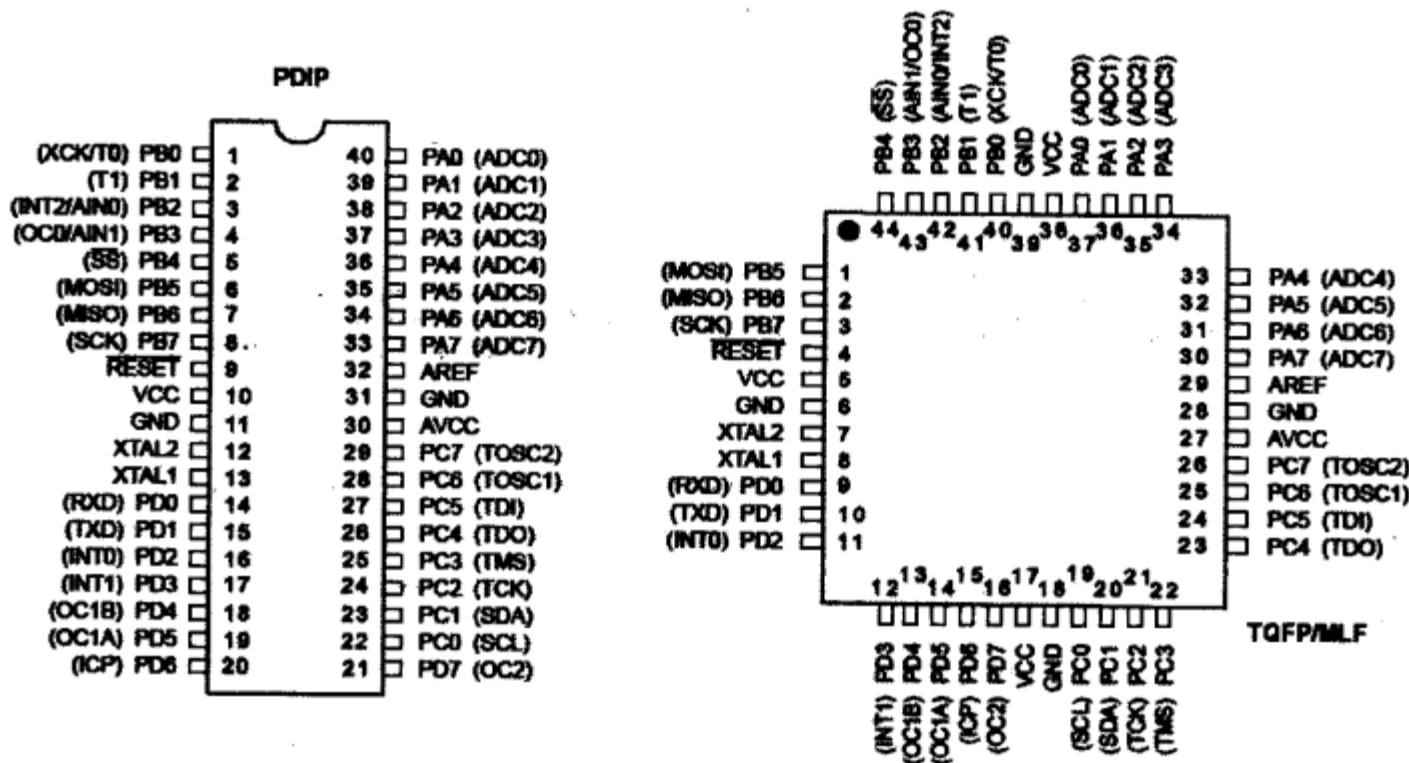


图 1-1 ATmega16(L) 单片机不同封装形式及引脚

ATmega16(L)的主要特征如下：

(1) 非易失性程序与数据存储器(Nonvolatile Program and Data Memories)

- 16 KB 的系统内可编程 Flash, 可擦写 10000 次, 支持 ISP(在系统编程)和 IAP(在应用编程)。
- 单片机内部数据存储器 SRAM 空间达 1 KB。
- 512 字节的 EEPROM 可擦写 100000 次, 可以在系统掉电时保存用户数据信息。

(2) 端口及外设特征(Peripheral Features)

- 32 个可编程 I/O 口分为 PA~PD 共 4 组, 每组 8 位。
- 2 个具有独立预分频器和比较器功能的 8 位定时/计数器, 1 个具有预分频器、比较功能和捕获功能的 16 位定时/计数器。
- 具有独立振荡器的实时计数器 RTC。
- 4 通道 PWM。
- 内置 8 通道 10 位精度的逐次逼近式 A/D 转换器(ADC), 支持单端和双端差分信号输入, 内含增益可编程运算放大器。
- 片内模拟比较器。
- 面向字节的两线式串行接口 TWI(Two-Wire serial Interface)。
- 可工作于主机/从机模式的串行接口 SPI (Serial Peripheral Interface)。
- 2 个可编程的串行 USART。
- 具有独立片内振荡器的可编程看门狗定时器。

(3) 工作电压与速度等级

- ATmega16L: 2.7 ~ 5.5 V; ATmega16: 4.5 ~ 5.5 V。
- ATmega16L: 0 ~ 8 MHz; ATmega16: 0 ~ 16 MHz。

在了解 ATmega16(L)单片机的主要特征以后,再来看一下单片机的引脚说明:

(1)4 个通用 I/O 端口

PA~PD 端口都是 8 位的双向 I/O 端口,具有可编程的内部上拉电阻,其输出缓冲器具有对称的驱动特性,可以输出和吸收较大电流。作为输入端口使用时,如果使能内部上拉电阻,端口被外部电路拉低时将输出电流。

(2)PA~PD 端口的第二功能

PA 端口(PA7~PA0)可作为 A/D 转换器的模拟输入端 ADC7~ADC0。

PB 端口(PB7~PB0)的第二功能如下:

- PB7:SCK (SPI 总线的串行时钟);
- PB6:MISO (SPI 总线的主机输入/从机输出信号);
- PB5:MO SI (SPI 总线的主机输出/从机输入信号);
- PB4:SS (SPI 从机选择引脚);
- PB3:AIN1 (模拟比较负输入)/OC0 (T/C0 输出比较匹配输出);
- PB2:AIN0 (模拟比较正输入)/INT2 (外部中断 2 输入);
- PB1:T1 (T/C1 外部计数器输入);
- PB0:T0 (T/C0 外部计数器输入)/XCK (USART 外部时钟输入/输出)。

PC 端口(PC7~PC0)引脚第二功能如下:

- PC7:TOSC2 (定时振荡器引脚 2);
- PC6:TOSC1 (定时振荡器引脚 1);
- PC5:TDI (JTAG 测试数据输入);
- PC4:TDO (JTAG 测试数据输出);
- PC3:TMS (JTAG 测试模式选择);
- PC2:TCK (JTAG 测试时钟);
- PC1:SDA (两线串行总线数据输入/输出线);
- PC0:SCL (两线串行总线时钟线)。

PD 端口(PD7~PD0)的第二功能如下:

- PD7:OC2 (T/C2 输出比较匹配输出);
- PD6:ICP1 (T/C1 输入捕获引脚);
- PD5:OC1A (T/C1 输出比较 A 匹配输出);
- PD4:OC1B (T/C1 输出比较 B 匹配输出);
- PD3:INT1 (外部中断 1 输入引脚);
- PD2:INT0 (外部中断 0 输入引脚);
- PD1:TXD (USART 输出引脚);
- PD0:RXD (USART 输入引脚)。

(3)其他引脚

RESET:复位输入引脚,持续时间超过最小门限时间的低电平将引起系统复位。

XTAL1:反向振荡放大器与片内时钟操作电路的输入端。

XTAL2:反向振荡放大器的输出端。

AVCC:端口 A 与 A/D 转换器的电源,不使用 ADC 时,该引脚应直接与 VCC 连接,使用



ADC 时应通过一个低通滤波器与 VCC 连接。

AREF: A/D 的模拟基准输入引脚。

VCC/GND: 数字电路的电源/地。

Proteus 仿真时电源已默认连接, 仿真电路图中电源连接全部默认。AVCC 在未使用 ADC 的多数电路中没有连接 VCC, 在实物电路设计时应注意连接 VCC。

1.2 AVR Studio+WinAVR 开发环境安装及应用

AVR Studio 是 Atmel 官方针对 AVR 系列单片机推出的集成开发环境, 它集开发调试于一体, 有很好的用户界面与很好的稳定性。由于 AVR Studio 仅支持编译汇编语言程序, 不支持对 C 语言程序的编译, 要基于 AVR Studio 搭建 AVR 单片机 C 语言程序开发环境, 除免费下载安装 AVR Studio 以外, 还需要下载安装 C/C++ 编译器。本书通过下载安装 WinAVR 来提供 AVR-GCC 编译器。

搭建基于 AVR Studio+WinAVR 的案例开发环境时, 所使用的安装包分别为 WinAVR-20090313-install.exe 及 AvrStudio416Setup.exe, 这两个安装包都可以从网上免费下载。在实际应用中, 可根据需要随时获取最新版本的安装包。

当前版本的 AVR Studio 对中文路径支持得还不够好, 在创建新项目时建议用英文或数字等非全角字符命名文件夹或文件。图 1-2 所示窗口中创建的 LCD_Test 项目保存于 E:\my_avr_c 文件夹下, 单击 Next 按钮后可进一步选择设备(Device)。

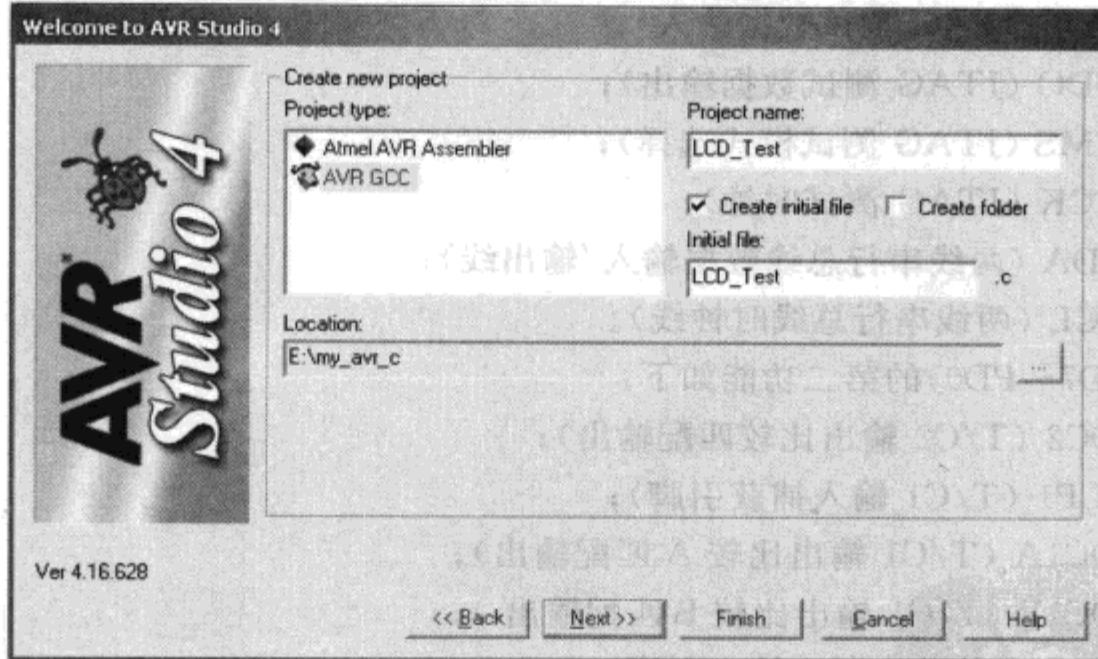


图 1-2 创建新项目

图 1-3 所示为 AVR Studio 主窗体, 左边的 AVR GCC 窗口以树形方式列出了当前项目的所有相关文件, 该窗口中除初始时创建的 LCD_Test.c 文件以外, 还添加了 LCD1602.c。编写完该项目 C 程序后, 在开始编译并生成 HEX、ELF、EEP 等文件之前, 要先单击工具栏上的编辑当前配置选项(Edit Current Configuration Options)按钮。该按钮在图 1-3 所示窗口第 2 条工具栏的最右边。

图 1-4 是当前项目选项设置窗口, 在常规(General)选项中可以看到当前项目的默认输出文件夹是 default。如果还没有设置当前 AVR 单片机选型, 可在设备(Device)下拉框中选

取,当前选择的是 ATmega16。设备下面是时钟频率(Frequency)设置文本框,当前设置的频率是 4 MHz。编译优化(Optimization)选项当前选择是“-Os”,它优化所输出的可执行程序文件的大小(Optimize for size)。有关该窗口中配置选项的更多细节,可单击帮助按钮查看。

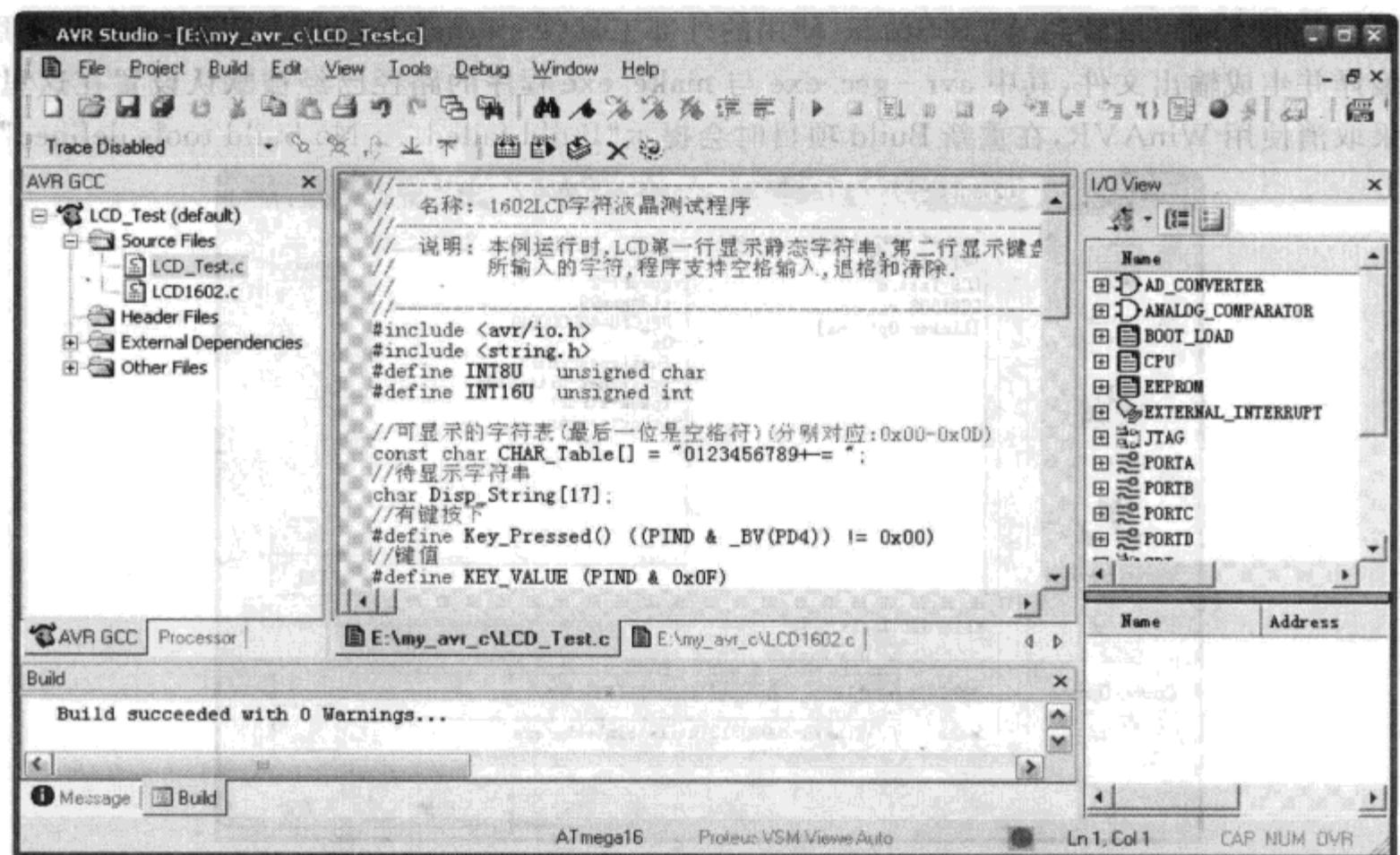


图 1-3 AVR Studio 主窗体

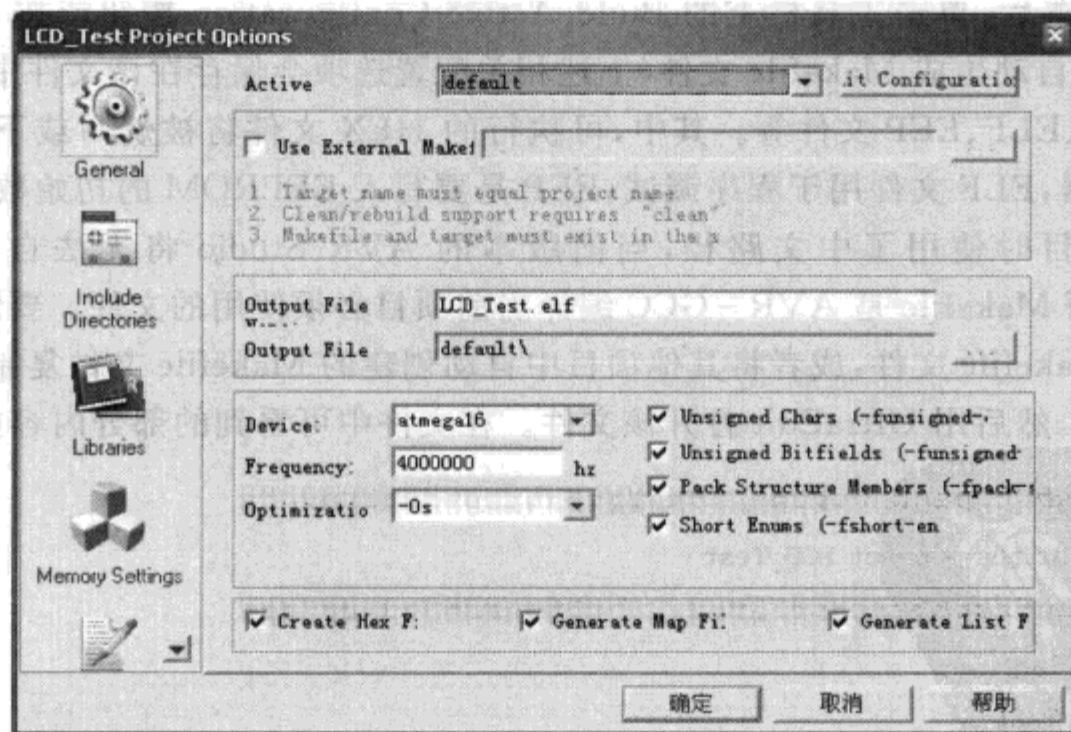


图 1-4 当前项目的常规配置选项窗口

该窗口的 Unsigned Chars(-funsigned char)设置选项应引起重视,此选项是默认选中的,它将程序中的 char 类型默认为无符号类型,这一点与 Keil C 是不一样的。假如程序中出现 -50~125 °C 范围内的正负温度比较,编写程序时就需要使用 signed char 类型定义变量,



而不能直接使用 char 类型进行定义,除非取消该设置。

该窗口中的其他配置选项可保持为默认值,不作任何改动。

当打开项目选项中的自定义选项(Custom Options)时可看到图 1-5 所示的配置窗口,在该窗口的最下端可以看到 AVR Studio 使用的外部工具(External Tools)是 WinAVR,它被用来编译并生成输出文件,其中 avr-gcc.exe 与 make.exe 程序的路径已经被默认设置在这里。如果取消使用 WinAVR,在重新 Build 项目时会提示“Build failed... No build tools defined”。

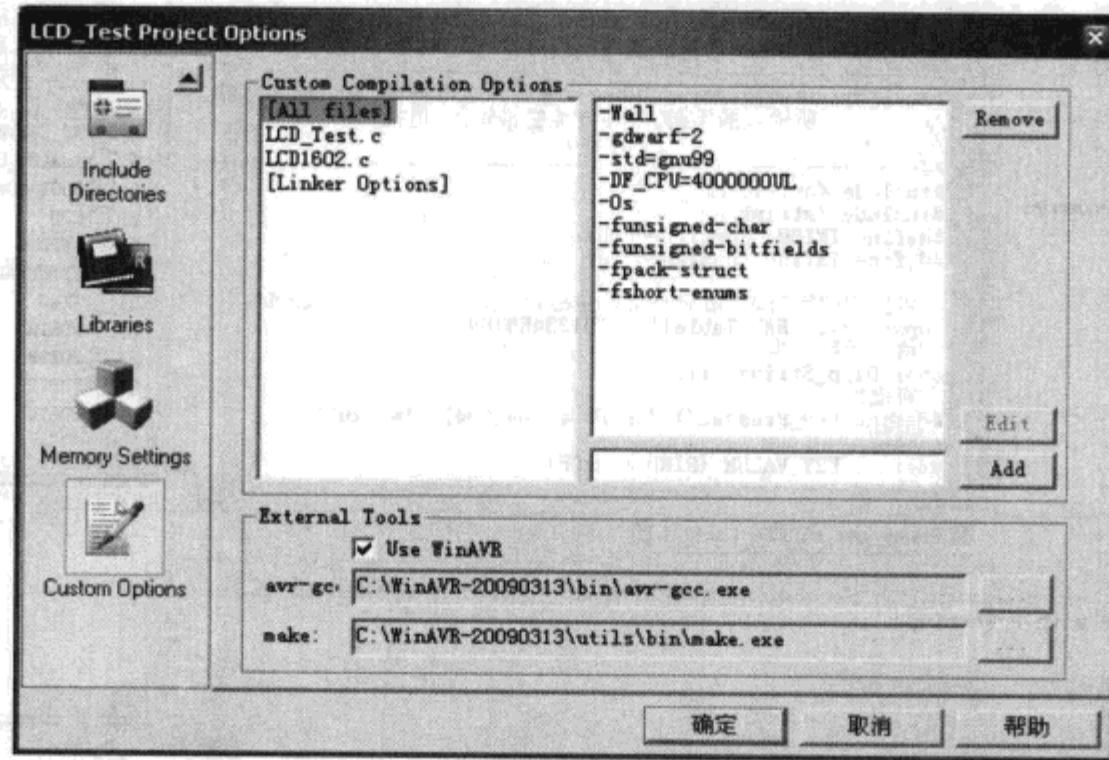


图 1-5 自定义选项配置窗口

完成所有设置后,单击工具栏上的 Build Active Configuration 按钮或按下 F7,这时 default 文件夹下会自动生成 Makefile 文件(上述相关配置选项会保存在该文件中),并生成输出文件,包括 HEX、ELF、EEP 文件等。其中,可执行的 HEX 文件将被烧写或下载到单片机的 Flash 程序存储器,ELF 文件用于程序调试,EEP 是要写入 EEPROM 的初始数据文件。

如果创建项目时使用了中文路径,当前版本的 AVR Studio 将无法自动创建或修改 Makefile 文件,而 Makefile 是 AVR-GCC 编译当前项目必须使用的文件。要解决这一问题,可以手动创建 Makefile 文件,或者将其他项目中自动创建的 Makefile 文件复制到当前项目的 default 文件夹下,然后用 UltraEdit 打开该文件。在文件中可看到的部分内容如下:

```
#####
# Makefile for the project LCD_Test
#####
## General Flags
PROJECT = LCD_Test
MCU = atmega16
TARGET = LCD_Test.elf
CC = avr-gcc
.....
CFLAGS += ..... -DF_CPU = 4000000UL -Os -funsigned -char .....
```

```

## Objects that must be built in order to link
OBJECTS = LCD_Test.o LCD1602.o
.....
## Compile
LCD_Test.o: .../LCD_Test.c
    $(CC) $(INCLUDES) $(CFLAGS) -c $<
LCD1602.o: .../LCD1602.c
    $(CC) $(INCLUDES) $(CFLAGS) -c $<
.....

```

该文件中的以下相关项目要在 UltraEdit 内根据需要进行修改并保存：

PROJECT——设置输出的项目名称，这需要根据当前项目名称进行修改；

MCU——设置当前项目所选用微控制器；

TARGET——设置输出目标调试文件(*.ELF)；

DF_CPU——设置所选择的时钟频率；

OBJECTS——列出 build 时所使用的目标文件(*.o)，有多个目标文件时中间用空格隔开；

Compile——列出了当前项目中的所有源程序文件名(*.c)及生成的目标文件名(*.o)，根据当前新创建的项目文件中的源程序文件名称及文件个数，这里需要进行相应的增删或修改。

通过 UltreaEdit 修改 Makefile 以后，处于中文路径下的 AVR 单片机 GCC 程序项目仍然可以正常编译并生成输出文件。

有关在 AVR Studio 中通过 Debug 菜单选择设备及调试平台(Select Device and Debug Platform)，对当前生成的程序文件进行跟踪调试的方法将在第 2 章介绍。

1.3 AVR – GCC 程序设计基础

AVR – GCC 是一款优秀的 AVR 编译软件，是 GUN C 编译器在 AVR 上的移植，支持多种操作系统。本节不准备全面讲述 AVR – GCC 程序设计的所有基础内容，下面仅列举部分编写调试本书案例程序时需要引起注意的部分和在编写过程中容易出现错误的部分。本节最后还列出了部分 AVR – GCC 对标准 C 语言的扩展功能。

1. 基本数据类型、有符号与无符号数应用、数位分解、位操作

在讨论本节内容之前需要先熟悉 GCC 基本数据类型，本书 C 程序中所使用的部分基本数据类型如表 1 – 1 所列。表中的第 2 列与第 3 列分别是 GCC 头文件<stdint.h>及本书所使用的各种数据类型的重定义。要使用精确定义的数据宽度，建议引入头文件<stdint.h>，使用该文件所定义的类型，这些类型都以“_t”结尾。另外，如果要在程序中使用布尔类型(bool，取值为 true/false)，可在程序中引入头文件<stdbool.h>。



表 1-1 AVR-GCC 部分常用基本数据类型

数据类型	stdint.h 定义	本书定义	长度/位	取值范围
signed char (char)	int8_t	INT8	8	-128~127
unsigned char	uint8_t	INT8U	8	0~255
char			8	取值范围为上述两者之一,具体范围由配置选项决定
int (signed int)	int16_t	INT16	16	-32768~32767
unsigned int	uint16_t	INT16U	16	0~65535
long (signed long)	int32_t	INT32	32	-2147483648~2147483647
unsigned long	uint32_t	INT32U	32	0~4294967295
float/double			32	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$

本书大多数案例使用的都是无符号数,对于 0~255 以内的整数,本书全部定义为 INT8U 类型,相当于字节类型 BYTE 或 <stdint.h> 定义的 uint8_t 类型;对于 0~65535 以内的整数,本书定义为 INT16U 类型,相当于字类型 WORD 或 <stdint.h> 定义的 uint16_t 类型。另外,GCC 不支持 float 类型,它将 float 直接解释为 double 类型。

如果涉及正负数的处理,在定义类型时要注意使用 signed 类型。例如温度控制程序中有正负温度,传感器实际上可处理的温度范围为 -55~125 °C。为使程序能对温度值进行正确比较,程序中将温度类型定义为 INT8 类型(即 signed char 或 int8_t 类型,其取值范围为 -128~127)。在 Keil C 中 char 类型默认为 signed char,但在 AVR Studio 的配置窗口中, char 类型默认为 unsigned char 类型,在定义可能出现正负值的温度变量时,不能定义使用 char temp,除非修改配置选项,将 char 默认为 signed char。

本书大量案例要将整数或浮点数显示在数码管上,这需要对待显示数据各数位进行分解,例如:

```
INT8U d = 227;
INT8U c[3];
c[0] = d/100;
c[1] = d/10 % 10;
c[2] = d % 10;
```

又如:

```
float x = 123.45;
```

如果要得到 x 的各个数位,可以先将 x 乘以 100,然后再分解各数位:

```
INT16U y = x * 100;
INT8U c[5], i;
for (i = 4; i != 0xFF; i--)
{
    c[i] = y % 10; y /= 10;
}
```

上面 for 循环中的循环条件一般都会写成 $i >= 0$ 的,但由于当 $i = 0$ 时,如果将 i 再减 1, i 将变为 0xFF,这个无符号数仍被认为大于等于 0,这样就不能保证 5 次循环了,因此要改写成 $i != 0xFF$ 。如果将 i 定义成 INT8 类型(signed char)而不是 INT8U 类型(unsigned char),使用 $i >= 0$ 时才能得到正确结果。

使用<stdlib.h>提供的函数 itoa 或 utoa 可将有符号或无符号整数转换为字符串,将字符串中各字符编码减去 0x30 或'0'也能分解出其各个数位,其使用方法可参考 1.4 节内容。

上述十进数的数位分解方法常用于数码管显示程序,显示时需要根据各数位提取数码管段码。

在本书案例中位操作也会大量出现,例如在有关 LED 流水灯、数码管位控制、串行收发、键盘扫描等大量案例中,各种位操作符都会频繁使用,例如位左移(<<)、右移(>>)、与(&)、或(|)、取反(~)、异或(^)等。这些位操作符都要熟练掌握和运用。

下面是有关位操作符的几个应用示例:

① 例如,要 PB 端口 PB7~PB0 逐位轮流循环置 1,可先定义变量 i ,并使之在 7~0 范围内循环取值,然后使 $\text{PORTB} = 1 << i$ 或 $\text{PORTB} = _BV(i)$,其中 $_BV(i)$ 等价于 $1 << i$ 。在循环过程中,PORTB 将分别输出 10000000、01000000、00100000、……、00000001,如此往复。

② 如果要 PB7~PB0 逐位轮流循环置 0,可有 $\text{PORTB} = \sim(1 << i)$ 或 $\text{PORTB} = \sim__BV(i)$ 。这类位操作在 LED 流水灯设计或集成式数码管位码扫描显示程序中都会用到。

③ 又如,已知 PD7 外接 LED 或蜂鸣器,要 LED 闪烁或蜂鸣器发声,可先定义:`#define LED_BLINK() PORTD ^= _BV(PD7)` 或 `#define BEEP() PORTD ^= _BV(PD7)`。然后在循环中反复调用 LED_BLINK() 或 BEEP() 即可实现所要求的输出。因为 $_BV(PD7)$ 即 10000000,在异或(^)操作过程中,低 7 位的 0000000 不会使 PORTD 的低 7 位发生任何变化,而最高位的 1 会在每次异或(^)时使 PORTD 的最高位取反,从而在 PD7 引脚实现所要求的输出系列…01010101…。

④ 在 3.9 节的 4×4 键盘矩阵扫描程序中,假设 PB 高 4 位连接矩阵行,低 4 位连接矩阵列,为判断整个键盘中是否有键按下,通常会先在行上发送 4 位扫描码 0000(即 $\text{PORTB} = 0x0F$),然后检查列上是否出现 0,这时使用的位操作语句是:

```
if((PINB & 0x0F) != 0x0F) { //有键按下}
```

因为 != 的优先级高于 &,该语句中的“与”操作表达式要加上括号。

本书中多个案例使用了字符液晶,向连接在 PC 端口的液晶屏发送显示数据时,需要先判断液晶是否忙,其忙标志在读取字节的最高位,因此又有类似语句:

```
if((PINC & 0x80) == 0x80) { //液晶忙}
```

⑤ 本节前面讨论了十进制数的数位分解及应用,对于十六进制数,例如 $k = 0x3E$,如果需要分解出 3 和 E(14),可有:

```
INT8U a, b;  
a = k / 16; b = k % 16;
```

或使用位操作符实现分解,即:

```
a = k >> 4; b = k & 0x0F;
```



2. 数组、字符串、指针

设计 AVR 单片机 C 程序时,常会使用到数组定义,例如在数码管显示程序中,一般总是会给出数码管段码表 SEG_CODE,它定义了七段数码管 0~9 的段码(共阳):

```
const INT8U SEG_CODE[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};
```

字符串类型定义在单片机程序设计中也会大量使用,特别是在有关液晶屏、点阵显示屏程序设计或串口通信程序设计中。下面是几个字符数组定义:

```
char s[20] = "Current Voltage";
char s[20] = {"Current Voltage"};
char s[20] = {'R','e','s','u','l','t',':',':'};
```

这 3 种定义是相同的,它们都占用 20 个字节空间,实际串长为 16 个字符,最后未明确赋值的 4 个字节全部为 0x00(即'\0'),在液晶屏上显示这类字符串时可用以下方法:

```
for(i=0; i<16; i++) { //显示字符 s[i] };
for(i=0; i<strlen(s); i++) { //显示字符 s[i] };
i=0; while (s[i++]!='\0') { //显示字符 s[i] };
```

要注意的是,如果字符串长为 16,而字符数组空间也只固定给出了 16 个字节,那么上述方法中的后两种方法就不可靠了,因为最后一个字符后面不一定会自动分配有字符串结束标志'\0'。

字符串还可以这样定义:

```
char s[] = "Current Voltage";
char * s = "Current Voltage";
```

这两种定义也是相同的,其串长均为 16 个字符,所占用的字节空间均为 17,因为字符串末尾被自动附加了结束标志字节 0x00(即'\0')。

在已知串长时,上述 3 种字符串显示方法均可使用,在字符串长未知时可使用上述的方法中的后 2 种。另外,上述显示方法还可以改写成:

```
for(i=0; i<16; i++) { //显示字符 *(s+i) };
for(i=0; i<strlen(s1); i++) { //显示字符 *(s+i) };
i=0; while (*(s+i) != '\0') { //显示字符 *(s+i); i++; };
```

编写 C 程序时,除使用字符数组(字符串)外,还会使用到字符串数组,例如:

```
char s[][20] = {"Current Voltage", "Counter:      ", "TH:      TL:      "};
```

如果要在液晶上显示“Counter: ”这个字符串,可用以下语句实现:

```
for(i=0; i<strlen(s[1]); i++) { //显示字符 s[1][i] };
```

当要在英文字符液晶上显示数值时,需要将待显示的整型或浮点数据转换为字符串,这时可用此前提到的数位分解方法先分解出各位数字,然后加上 0x30(即'0')得到对应数字的 ASCII 编码,这些 ASCII 编码可直接送 LCD 显示。

除使用上述方法外,还可以使用<stdlib.h>提供的将有符号或无符号整数转换为字符串

的函数 itoa 和 utoa，代码示例如下：

```
int a = -12345;
unsigned int b = 9850;
char sa[7], sb[5];
itoa(a, (char *)sa, 10);
utoa(b, (char *)sb, 10);
```

2 个函数的第 3 个参数用于设置基数(radix)，这里所选择的是 10(十进制)。经转换后，a、b 所对应的字符串保存于 sa 与 sb 中，这 2 个字符串可以直送 LCD 显示。如果要显示在数码管上，除 sa 中的符号位以外，显示其他各位时，可通过执行 sa[i] - '0' 或 sb[i] - '0' 将第 i 个字符转换为数字，然后根据数字 0~9 提取段码再送数码管显示。不过本书数码管显示程序中未采用这种方法分解数位。

另一种将数值转换为字符串的方法是使用 sprintf 函数，示例代码如下：

```
char Buf[10];
float x = -123.45;
sprintf(Buf, "%5.2f", x);
```

遗憾的是，Keil C 支持在 sprintf 函数中使用 %f 占位符，而 AVR-GCC 却不支持，上述最后一行语句要用下面的语句代替：

```
sprintf(Buf, "%d.%2d", (int)x, (int)(fabs(x) * 100) % 100);
```

由于 x 可能为正数，也可能为负数，为适应这种可能性，语句中添加了取绝对值函数 fabs。运行改写后的语句，字符串缓冲 Buf 将被以下字节填充：0x2D, 0x31, 0x32, 0x33, 0x2E, 0x34, 0x35, 0x00, 0x00, 0x00。这些字节就是字符串“-123.45”中各字符的 ASCII 码，Buf 可直接送字符液晶屏显示。

又如，假设已有语句：

```
char Buf[25] = "Result:      ";
```

执行 sprintf(Buf + 7, "%d.%2d", (int)x, (int)(fabs(x) * 100) % 100) 时会使 Buf 中的字符串变为：“Result: -123.45”。使用下面的语句也可以得到相同的结果：

```
char Buf[25];
sprintf(Buf, "Result: %d.%2d", (int)x, (int)(fabs(x) * 100) % 100);
```

显然，与 itoa 和 utoa，包括未举例的 ltoa、ultoa、dtostrf 等函数相比，使用 sprintf 的优点是它能根据要求输出“格式化”的字符串，而前者仅将数值直接转换为字符串，在实际应用中大家可根据具体需要进行选择。

另外，GCC 还提供了多个与字符串/数字操作有关的函数，例如 atoi、atol、atof、strtod、strtol、strtoul。其中部分函数将在 5.26 节“计算器”和 5.27 节“电子秤”的案例中使用，这两个案例都涉及数字字符串输入及数据运算与显示。在程序设计中涉及数据输入/输出及运算与显示时可恰当使用这些函数。

指针是 C 语言的重要特色之一，对于语句：



```
INT8U a[10] = {1,2,3,4,5,6,7,8,9,10};  
INT8U * ptr = a;
```

其中 ptr 指向数组 a 的第 0 个字节,显示数组内容可使用下面的代码:

```
for(i = 0; i < 10; i++) { //输出 a[i]、*(ptr + i)、*ptr++ 或 *(a + i) };
```

但是不能使用下面的代码:

```
for(i = 0; i < 10; i++) { //输出 *a++ };
```

因为数组名 a 虽然也是数组中第 0 个字节的地址,但它不能在运行过程中改变。也正是因为数组名同样也是第 0 个元素的指针;因此某些函数定义中的形参为数组,调用函数时给出的实参常为指向同类型数据的指针;反之形参为指针,实参为数组名也很常见。

前面字符串示例中也出现了指针应用,对它们同样要能熟练掌握。

3. 程序流程控制

用 C 语言开发 AVR 单片机程序时,流程控制语句 if、switch、for、while、do while、goto 都可能频繁使用,下面仅对单片机程序中几个不同于常规的流程控制语句作简要说明。例如程序中可能会有如下代码(有关端口寄存器的设置可参阅 1.5 节):

```
DDRB = 0x00; PORTB = 0xFF;  
if (PINB != 0xFF) { //执行相应操作}
```

这段程序会让初学者感到奇怪,这里 if 语句中的条件什么时候会成立呢?前面没有任何语句设置 PINB 的值呀?

实际情况是:PB 端口外接一组按键,各按键一端连接 PB 端口,另一端接地,PB 端口设置了内部上拉;没有按键按下时,从 Proteus 中观察到 PB 端口引脚都是红色的,读取的 PINB 为 0xFF;如果按键中有一个或多个被按下,则对应引脚将变为蓝色,从 PINB 读取的输入信号也会发生变化。可见,用 C 语言设计单片机程序时,某些寄存器的值不同于程序中某些变量的值,它们的值会随时因外部事件(包括外部操作、中断、定时/计数等)影响而改变。

如果编写的程序中要用 if 语句进行多路平行判断,在这种情况下多用 switch 语句编写代码。使用 switch 语句时要注意各 case 后的 break 语句,恰当地使用 break 和省略 break 可以使分支独立,或者使多个 case 分支共用某个代码块。

对于 for 循环,AVR - GCC 中允许将其控制变量直接定义在循环内,例如:

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

在 AVR 单片机程序的 main 函数中还会经常看到这样的代码块:

```
while (1)  
{  
    //循环体;  
}
```

用标准 C 语言编写程序时,这段代码的循环体内必定有退出循环的语句存在,但是编写单片机程序时,几乎多数主程序中类似语句内都找不到退出循环的语句,这是因为单片机应用系统不同于普通的软件系统,它一旦开始运行就会一直持续下去,对外部的操作或状态变化作

出实时响应或处理,除非系统关闭或出现其他故障。

类似的,很多案例中的主程序最后有一行代码:“while(1);”或“for(;;);”,这显然是两个死循环,在出现该语句的案例中,外部事件的处理工作必定被放在中断服务程序(ISR, Interrupt Service Routine)内。主程序完成若干初始化工作并使能中断以后就不再执行其他操作,它一直等待在 while 或 for 循环所在的语句行;一旦中断发生即保存现场,进入中断服务程序进行处理,完成后恢复现场并返回,直到有后续中断继续发生。

4. AVR - GCC 对标准 C 语言的部分扩展

AVR - GCC 支持 C99 规范,对标准 C 语言进行多项扩展,下面列举部分供大家在编写 AVR 单片机 C 程序时参考使用。

(1) switch 语句

标准的 C 语言程序中,各 case 分支只能适配一个常量,而 GCC 则允许一个 case 匹配多个常量。例如:

```
switch (x)
{
    case 1 ... 3: 语句块 1; break;
    case 4 ... 5: 语句块 2; break;
}
```

注意:在编写上述 switch 语句时,“1 ... 3”及“4 ... 5”中“...”的前后都要空一格。

(2) 函数嵌套定义

标准 C 语言不允许在函数内部定义函数,而 GCC 允许这样定义。例如:

```
double sqr_sum (double a, double b)
{
    double square (double z) { return z * z; }
    return square (a) + square (b);
}
```

上述 sqr_sum 函数中的最后一行代码调用了其内部定义的函数 square。

(3) 数组空间的动态定义

标准 C 语言的数组空间大小必须是常量,不允许在程序中使用变量定义数组空间大小,但 GCC 允许这样定义,不过这些数组只允许是局部数组。例如:

```
void merge_string (char * s1, char * s2)
{
    // 动态分配能容纳 2 个字符串的 str 空间
    char str[strlen (s1) + strlen (s2) + 1];
    strcpy (str, s1); strcat (str, s2);
    // 执行其他操作
}
```

(4) 用变量初始化数组及根据索引初始化数组

标准 C 语言数组初始化时不允许使用变量给元素赋值,而 GCC 则允许这样赋值。例如:

```

void fg_Handle(float f, float g)
{
    float fg[2] = { f - g, f + g };
    //执行其他操作
}

```

而且，在初始化数组时，还可以根据索引给部分元素赋值，例如：

```
int a[6] = { [4] = 29, [2] = 15 };
```

该语句与下面的语句等价：

```
int a[6] = { 0, 0, 15, 0, 29, 0 }.
```

1.4 程序与数据内存访问

ATmega16 单片机的存储器空间分为独立的 Flash 程序存储器、片内 SRAM 和 EEPROM 数据存储器。

16 KB 的 Flash 程序存储器空间分为两个区：引导程序区和应用程序区。在使用 Proteus 软件仿真执行程序时，所生成的 HEX 程序通过设置芯片的 Program File 属性绑定到单片机的应用程序区。

在全书大量案例中，一般变量都被分配在 SRAM 数据存储空间内，例如：

```
unsigned char i; char c; int a[20]; long x; 或 INT8U k, INT32U addr, uint16_t m;
```

有的案例代码内有大量汉字点阵或图像点阵数据，这些数据在程序运行过程不会改变，属于只读数据。由于 SRAM 空间大小有限，如果将这些数据分配于 SRAM 内，可能会因为空间限制导致分配失败，或导致其他自由变量没有空间可以分配。本书案例中，这类数据都被分配到 Flash 存储器空间。

为访问 Flash 存储器中的只读数据，在编写 GCC 程序时要引入头文件 `<avr/pgmspace.h>`。该文件提供了访问 Flash 程序空间中所保存的只读数据的大量函数，例如 `pgm_read_byte` 与 `pgm_read_word` 可分别从 Flash 存储器中指定地址读取字节或字。要将 `char` 或 `unsigned char` 类型数据保存于 Flash 存储器，可使用类型：`prog_char` 和 `prog_uchar`。

编程读/写 EEPROM 数据存储器时，可引入头文件 `<avr/eeprom.h>`。该头文件提供了大量专门用于读/写 EEPROM 的函数，例如 `eeprom_read_byte` 与 `eeprom_write_byte` 函数可分别从 EEPROM 中指定地址读取字节或向指定地址写入字节，在读/写时可通过函数 `eeprom_busy_wait` 执行忙等待。

本书有关以太网应用的案例中使用了实时操作系统 Nut/OS，该操作系统也提供了读/写 EEPROM 存储器的 API，通过调用 Nut/OS 提供的 API 也可以很方便地访问 EEPROM。

1.5 I/O 端口编程

全书所有单片机应用案例都涉及对 I/O 端口的访问，所有 C 程序无一例外地需要引入头文件 `<avr/io.h>`。对于本书案例大量使用的 ATmega16 单片机，`defalut/Makefile` 文件设置

微控制器为 ATmega16，即 MCU=atmega16，AVR-GCC 在编译程序时定义宏 __AVR_AT-mega16__，<avr/io.h> 通过判断定义 __AVR_ATmega16__ 而引入头文件 <avr/iom16.h>，该头文件给出了 ATmega16 单片机的所有寄存器及大量引脚定义。

对于 PA~PD 端口，在不涉及第二功能时，其基本 I/O 功能是相同的。与每个端口相关的寄存器有 3 个，它们分别是数据方向寄存器 DDRx(Port x Data Direction Register)、端口数据寄存器 PORTx(Port x Data Register) 及端口输入引脚地址寄存器 PINx(Port x Input Pins Address Register) (其中 x=A、B、C、D)。表 1-2 给出了 ATmega16 单片机 I/O 端口的配置方法。

表 1-2 ATmega16 I/O 端口的配置方法

DDRx0~7	PORTx0~7	I/O	上 拉	备 注
0	0	输入	关闭	三态(高阻)
0	1	输入	打开	提供弱上拉，被外部电路拉低时输出电流
1	0	输出	关闭	输出 0
1	1	输出	关闭	输出 1

下面是端口的几个配置示例。

示例一：

```
DDRA = 0x00;           //PA 端口全部设为输入
PORTA = 0xFF;          //PA 端口全部设内部上拉
x = PINA;              //读取 PA 端口的输入信号
```

如果将 PORTA=0xFF 改成 PORTA=0x00 则不设内部上拉，无输入时处于高阻状态。

示例二：

```
DDRB = 0x0F;           //将 PB 端口的高 4 位设为输入，低 4 位设为输出
PORTB = 0xF0;          //打开 PB 端口高 4 位内部上拉电阻，低 4 位则直接输出 0000
```

示例三：

```
DDRC = 0xFF;           //PC 端口全部设为输出
PORTC = 0xF0;          //PC 端口输出 11110000
```

除了上述对端口的整体操作以外，在实际编程时还会遇到大量对某个引脚的输入或输出操作，例如 PD3 引脚外接按键，按键另一端接地，为判断按键是否按下。常见的代码如下：

```
DDRD &= 0B11110111;    //PD3 引脚设为输入
PORTD |= 0B00001000;    //PD3 内部上拉
```

有了上述配置后，为判断 PD3 外接按键是否按下，可编写代码：

```
If ((PIND & 0B00001000) == 0x00) //如果按键按下
{
    .....
}
```

为简化设计，增加可读性，上述代码中的 0B00001000 可改写成 _BV(PD3)，而



0B11110111 则可改写成 `~_BV(PD3)`。

又如,要在 PD7 引脚输出 010101……序列,可先定义 PD7 为输出:

```
DDRD |= _BV(PD7);
```

然后反复调用:

```
PORTE ^= _BV(PD7);
```

如果要通过某引脚(例如 PB2)串行输出一字节数据 dat,且要求先发送高位,后发送低位。常见的代码如下:

```
//因为写 I/O 的代码出现很频繁,程序中常使用它们的宏定义
#define w_1() PORTB |= _BV(PB2)
#define w_0() PORTB &= ~_BV(PB2)
//.....
DDRB |= _BV(PB2); //PB2 设为输出
for(i = 0; i < 8; i++) //通过 PB2 引脚串行输出 8 位,先发送高位,后发送低位
{
    if(dat & 0x80) w_1(); else w_0();
    dat <<= 1;
}
```

上述 for 循环还可以改写成:

```
for(i = 0x80; i != 0x00; i >>= 1)
{
    if(dat & i) w_1(); else w_0();
}
```

在涉及按键控制的程序实例中,常需要判断某引脚所连接的按键是否按下,例如 PB3 外接按键 K1,在程序中可以定义:

```
#define K1_DOWN() (PINB & _BV(PB3)) == 0x00
```

除了通过“与”操作(&)判断按键状态以外,还可以使用<avr/io.h>提供的宏 bit_is_clear 判断按键状态,等价的定义语句如下:

```
#define K1_DOWN() bit_is_clear(PINB, PB3)
```

如果要将 PB3 外接开关 S1 拨至高电平定义为 ON,类似的可有如下定义:

```
#define S1_ON() (PINB & _BV(PB3)) 或
```

```
#define S1_ON() bit_is_set(PINB, PB3)
```

1.6 外设相关寄存器及应用

本书大量案例都会涉及外部中断、定时/计数器、USART、TWI、SPI、ADC、比较器等外设程序设计,使用 AVR-GCC 开发程序时大量使用这些外设的相关控制、状态或数据寄存器。

本节列出有关这些寄存器的技术内容。若需要完整的 ATmega16 单片机技术资料,可参阅其 PDF 技术手册文件。

(1) 状态寄存器——SREG (Status Register)

I	T	H	S	V	N	Z	C
---	---	---	---	---	---	---	---

SREG 的最高位 I 为全局中断使能(Global Interrupt Enable)位。单独的中断使能由其他独立的控制寄存器控制。如果清零位 I, 则不论单独中断标志置位与否, 中断都不会产生。

本书置位和清零位 I 时, 多使用 AVR-GCC 所定义的宏 sei() 和 cli(), 部分程序使用:

`SREG |= 0x80` 或 `SREG &= 0x7F`

(2) 通用中断控制寄存器——GICR(General Interrupt Control Register)

INT1	INT0	INT2	—	—	—	IVSEL	IVCE
------	------	------	---	---	---	-------	------

GICR 中的最高 3 位为 INT1、INT0、INT2, 在将它们设为 1 且状态寄存器 SREG 中的 I 置位时, 即可使能相应引脚的外部中断。

以 INT0 为例, 置位和清零语句可分别写成:

`GICR |= _BV(INT0)` 和 `GICR &= ~_BV(INT0)`

置位 SREG 位 I 的语句中不能使用 `_BV(I)`, 因为 `<iom16.h>` 中没有定义 SREG 的位 I。

(3) 通用中断标志寄存器——GIFR (General Interrupt Flag Register)

INTF1	INTF0	INTF2	—	—	—	—	—
-------	-------	-------	---	---	---	---	---

INTF0~INTF2(External Interrupt Flag 0~2)分别为 INT0~INT2 的外部中断标志位。INT0~INT2 引脚的逻辑电平或边缘变化将触发中断请求, 并置位相应的中断标志位。如果 SREG 的位 I 为 1 且 GICR 中的相应位使能, MCU 即跳转到相应的中断向量地址, 进入中断服务服务后该标志自动清零。

(4) MCU 控制寄存器——MCUCR(MCU Control Register)

SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00
-----	----	-----	-----	-------	-------	-------	-------

MCUCR 的中断触发控制(Interrupt Sense Control)位 ISC01/ISC00 与 ISC11/ISC10 分别用于设置 INT0 与 INT1 的中断触发方式, 两者各自的 4 种取值组合及对应的触发方式如下:

- 00——INT0/1 为低电平时产生中断请求;
- 01——INT0/1 引脚上任意的逻辑电平变化都将引发中断;
- 10——INT0/1 的下降沿产生异步中断请求;
- 11——INT0/1 的上升沿产生异步中断请求。

对于 INT2, 其中断控制方式由 MCUCSR(MCU 控制与状态寄存器)中的 ISC2 位控制, ISC2 取 0/1 时, 分别对应于下降沿和上升沿中断触发方式。有关 MCUCR 及 MCUCSR 寄存器相关位的更详细说明可参阅技术手册文件。

3.14 节“INT0 中断计数”、3.15 节“INT0 及 INT1 中断计数”及第 4、5 章大量案例程序设计中使用上述寄存器中的 SREG、GICR、MCUCR。



(5) T/C 中断屏蔽寄存器——TIMSK(Timer/Counter Interrupt Mask Register)

OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
-------	-------	--------	--------	--------	-------	-------	-------

TOIE0~TOIE2(T/C0~2 Overflow Interrupt Enable)分别是 T/C0~T/C2 的溢出中断使能位。

TICIE1(T/C1 Input Capture Interrupt Enable)是 T/C1 输入捕获中断使能位。

OCIE0、OCIE2(Output Compare Match Interrupt Enable 0、2)分别是 T/C0 与 T/C2 的输出比较匹配中断使能位, OCIE1A、OCIE1B 分别是 T/C1 输出比较 A 匹配与输出比较 B 匹配中断使能位。

使能 T/C0 溢出中断时可使用语句 `TIMSK |= _BV(TOIE0)`, 要同时使能 T/C0 与 T/C1 溢出中断, 所使用的语句为: `TIMSK |= _BV(TOIE0) | _BV(TOIE1)`。其他 T/C 中断的使能设置与此类似。

本书有数十个案例中出现了 TIMSK 寄存器, 例如 3.22 节“用定时器设计的门铃”、3.35 节“看门狗应用”案例及第 4、5 章的大量相关案例。

(6) T/C 中断标志寄存器——TIFR (Timer/Counter Interrupt Flag Register)

OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
------	------	------	-------	-------	------	------	------

当 T/C0~T/C2 溢出中断时, 定时/计数器溢出标志(Timer/Counter Overflow Flag)位 TOV0~TOV2 置位, 且 SREG 的位 I 置位时, 溢出中断服务器程序得以执行。

当 T/C0、T/C2 输出比较匹配中断时, 输出比较匹配标志(Output Compare Flag)位 OCF0、OCF2 置位, 且 SREG 的位 I 置位时, 中断服务器程序得以执行。

外部引脚 ICP1 出现捕获事件时 T/C1 输入捕获标志(Input Capture Flag)位 ICF1 置位。此外, 当 ICR1 作为计数器的 TOP 值时, 一旦计数器值达到 TOP, ICF1 也置位。

当 TCNT1 与 OCR1A 匹配成功时, T/C1 输出比较 A 匹配标志(Output Compare A Match Flag)位 OCF1A 被设为 1。当 TCNT1 与 OCR1B 匹配成功时, T/C1 输出比较 B 匹配标志(Output Compare B Match Flag)位 OCF1B 被设为 1。

3.17 节“TIMER0 控制流水灯”及 3.32 节“EEPROM 读/写与数码管显示”案例中使用了 TIFR 寄存器。

(7) T/C0 控制寄存器——TCCR0(Timer/Counter Control Register)

FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
------	-------	-------	-------	-------	------	------	------

FOC0(Force Output Compare 0)为强制输出比较位。

WGM01/00(Waveform Generation Mode 01/00)为波形产生模式位。

COM01/0(Compare Match Output Mode 1/0)为比较匹配输出模式位。

对它们的详细设置可参阅技术手册文件。

寄存器最后 3 位的 CS02~CS00(Clock Select02~00)为时钟选择位, 其设置如下:

- 取值为 000 时, 无时钟, T/C0 不工作;
- 取值为 001、010、011、100、101 时, 分别对应于 1、8、64、256、1024 分频;
- 取值为 110、111 时表示时钟由 T0 引脚输入, 两者分别对应于下降沿/上升沿触发。

(8) T/C0 定时/计数器寄存器——TCNT0(T/C0 Timer/Counter Register)

TCNT0 在预分频时钟或 T0 引脚输入脉冲驱动下计数, TCNT0 的 8 位数据可以直接进行读/写访问。

3.21 节“用工作于计数方式的 T/C0 实现 100 以内的脉冲或按键计数”、3.26 节“用工作于异步模式的 T/C2 控制的可调式数码管电子钟”案例及其他章节共计 20 多个案例中使用了 TCCR0 及 TCNT0 寄存器。

(9) T/C0 输出比较寄存器——OCR0(Output Compare Register)

输出比较寄存器包含一个 8 位的数据, 它不间断地与 TCNT0 进行比较, 匹配事件可以用来产生输出比较中断, 或者用来在 OC0 引脚上产生波形。

(10) T/C1 控制寄存器 A——TCCR1A(Timer/Counter1 Control Register A)

COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
--------	--------	--------	--------	-------	-------	-------	-------

COM1A1 : A0 与 COM1B1 : B0 是通道 A、B 的比较输出模式(Compare Output Mode)。 FOC1A、FOC1B 分别为通道 A、B 的强制输出比较。

WGM11 : 10 为波形发生模式, 这两位与位于 TCCR1B 寄存器的 WGM13 : 12 相结合, 用 WGM13~WGM10 共 4 位控制计数器的计数序列——计数器计数的上限值和确定波形发生器的工作模式。所支持的模式有普通模式(计数器), 比较匹配时清零定时器(CTC)模式及 3 种脉宽调制(PWM)模式。详细设置可参阅技术手册文件。

3.28 节“用 TIMER1 输出比较功能调节频率输出”及 3.29 节“TIMER1 控制的 PWM 脉宽调制器”及其他章节等多个案例中出现了 TCCR1A 寄存器。

(11) T/C1 控制寄存器 B——TCCR1B(Timer/Counter1 Control Register B)

ICNC1	ICES1	—	WGM13	WGM12	CS12	CS11	CS10
-------	-------	---	-------	-------	------	------	------

ICNC1(Input Capture Noise Canceler 1)置位时使能输入捕获噪声抑制功能, 此时外部引脚 ICP1 的输入被滤波。其作用是从 ICP1 引脚(Input Capture Pin)连续进行 4 次采样, 如果 4 个采样值都相等, 那么信号送入边沿检测器。因此, 使能该功能使得输入捕获被延迟了 4 个时钟周期。

ICES1(Input Capture Edge Select)为输入捕获触发沿选择位, 选择使用 ICP1 上的哪个边沿触发捕获事件。ICES1 为 0 时选择下降沿触发输入捕获, ICES1 为 1 时选择上升沿触发输入捕获。按照 ICES1 的设置捕获到一个事件后, 计数器的数值被复制到输入捕获寄存器 ICR1。捕获事件还会置位 ICF1。如果此时中断使能, 输入捕获中断即被触发。

WGM13 : 12 为波形发生模式, 与 TCCR1A 中的 WGM11 : 10 配合使用。

寄存器最后 3 位的 CS12~CS10 为时钟选择位, 其设置如下:

- 取值为 000 时, 无时钟, T/C1 不工作;
- 取值为 001、010、011、100、101 时, 分别对应于 1、8、64、256、1024 分频;
- 取值为 110、111 时表示时钟由 T1 引脚输入, 两者分别对应于下降沿/上升沿触发。

3.24 节“100 000 s 以内的计时程序”及 5.18 节“8×8 LED 点阵屏仿电梯数字滚动显示”等多个案例中出现了 TCCR1B 寄存器。

(12) T/C1 定时/计数器寄存器——TCNT1(T/C1 Timer/Counter Register)

T/C1 的 16 位的数据寄存器 TCNT1 由高 8 位的 TCNT1H 与低 8 位的 TCNT1L 组成, 在单片机 GCC 程序中, 可以通过对两者独立进行访问来读/写 TCNT1, 也可以直接单独访问



TCNT1。

3.20 节“TIMER1 与 TIMER2 控制十字路口秒计时显示屏”及 5.28 节“模拟射击训练游戏”等多个案例中出现了 TCNT1 寄存器。

(13) 输出比较寄存器 1A——OCR1A(Output Compare Register 1 A)

T/C1 的 16 位的输出比较寄存器 OCR1A 由高 8 位的 OCR1AH 与低 8 位的 OCR1AL 组成, 它与 TCNT1 寄存器中的计数值进行连续比较, 一旦数据匹配即产生一个输出比较中断, 或在 OC1A 引脚生成波形输出。在 AVR-GCC 程序中, 可以通过访问 2 个 8 位的寄存器来访问 OCR1A, 也可以单独直接访问 OCR1A。

3.27 节“TIMER1 定时器比较匹配中断控制音阶播放”、3.29 节“TIMER1 控制的 PWM 脉宽调制器”及 5.2 节“电子琴仿真”、5.24 节“带液晶显示的红外遥控调速仿真”等案例使用了 OCR1A 寄存器。

(14) 输出比较寄存器 1B——OCR1B(Output Compare Register 1 B)

T/C1 的 16 位输出比较寄存器 OCR1B 由高 8 位的 OCR1BH 与低 8 位的 OCR1BL 组成, OCR1B 与 TCNT1 寄存器中的计数值进行连续比较, 一旦数据匹配即产生一个输出比较中断, 或在 OC1B 引脚生成波形输出。该寄存器的访问方法与 OCR1A 类似。

(15) 输入捕获寄存器 1——ICR1(Input Capture Register 1)

16 位的 ICR1 由高 8 位的 ICR1H 与低 8 位的 ICR1L 组成, 当外部引脚 ICP1 有输入捕获触发信号产生时, 计数寄存器 TCNT1 中的值将写入 ICR1, 模拟比较器的输出也可用来触发 T/C1 的输入捕获功能。ICR1 的设定值可作为计数器的 TOP 值。3.25 节“用 TIMER1 输入捕获功能设计的频率计”案例中使用了 ICR1 寄存器。

(16) T/C 控制寄存器——TCCR2(Timer/Counter Control Register 2)

FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
------	-------	-------	-------	-------	------	------	------

FOC2 为强制输出比较位, WGM21 : 20 设置波形产生模式, COM21 : 20 设置比较匹配输出模式。CS22 : 20 为时钟选择位, 取值为 000 时无时钟, T/C2 不工作; 当取值为 001~111 时, 分别对应于 1、8、32、64、128、256、1024 分频。

(17) T/C2 定时/计数器寄存器——TCNT2(T/C2 Timer/Counter Register)

通过 TCNT2 可以直接对 T/C2 计数器的 8 位数据进行读/写访问。对 TCNT2 寄存器的写访问将在下一个时钟阻止比较匹配。在计数器运行的过程中修改 TCNT2 的数值有可能丢失一次 TCNT2 和 OCR2 的比较匹配。

5.9 节“高仿真数码管电子钟”及 5.10 节“1602 LCD 显示的秒表”等多个案例中使了 TCCR2 及 TCNT2 寄存器。

(18) 输出比较寄存器 2——OCR2(Output Compare Register 2)

OCR2 包含一个 8 位的数据, 它与计数器数值 TCNT2 持续进行比较, 匹配事件用来产生输出比较中断, 或者用来在 OC2 引脚产生波形。

(19) 异步状态寄存器——ASSR(Asynchronous Status Register)

—	—	—	—	AS2	TCN2UB	OCR2UB	TCR2UB
---	---	---	---	-----	--------	--------	--------

AS2(Asynchronous T/C2) 为 0 时, T/C2 由 I/O 时钟 $\text{clk}_{\text{I/O}}$ 驱动, AS2 为 1 时, T/C2 由连接到 TOSC1 引脚的晶体振荡器驱动。改变 AS2 有可能破坏 TCNT2、OCR2 与 TCCR2 的内

容。ASSR 寄存器其他位的有关说明可参阅技术手册文件。

3.26 节“用工作于异步模式的 T/C2 控制的可调式数码管电子钟”及 5.9 节“高仿真数码管电子钟”等数个案例中使用了 ASSR 寄存器。

(20) SPI 状态寄存器——SPSR(SPI Status Register)

SPIF	WCOL	—	—	—	—	—	SPI2X
------	------	---	---	---	---	---	-------

SPIF(SPI Interrupt Flag)为 SPI 中断标志位,串行发送结束后 SPIF 置位。

WCOL 为写碰撞标志位(Write Collision flag),在发送当中对 SPI 数据寄存器 SPDR 写数据将置位 WCOL。WCOL 可以通过先读 SPSR,再紧接着访问 SPDR 来清零。

SPI2X 为 SPI 倍速位(Double SPI Speed Bit)。置位后 SPI 的速度加倍。若为主机,则 SCK 频率可达 CPU 频率的一半。若为从机,则只能保证 $f_{osc}/4$ 。

(21) SPI 控制寄存器——SPCR(SPI Control Register)

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
------	-----	------	------	------	------	------	------

SPIE 为 SPI 中断使能位(SPI Interrupt Enable),SPE 为 SPI 使能位,进行任何 SPI 操作之前必须置位 SPE。

DORD 为数据次序位(Data Order),置位时先发送 LSB,否则先发送 MSB。

MSTR 为主/从选择位(Master/Slave Select Bit),置位时选择主机模式,否则为从机模式。

CPOL 设置时钟极性(Clock Polarity),置位表示空闲时 SCK 为高电平,否则为低电平。

CPHA 设置时钟相位(Clock Phase),它决定数据是在 SCK 的起始沿采样还是结束沿采样。

最后 2 位 SPR1 与 SPR0 为 SPI 时钟速率选择位(SPI Clock Rate Select Bit),确定主机的 SCK 速率。SPR1 和 SPR0 对从机没有影响。当 SPR1、SPR0 取值 00、01、10、11 时,SCK=时钟频率 $f_{osc}/4$ 、 $/16$ 、 $/64$ 、 $/128$ 。如果 SPI 状态寄存器 SPSR 中的倍速位 SPI2X 置位,则 SCK = 时钟频率 $f_{osc}/2$ 、 $/8$ 、 $/32$ 、 $/64$ 。

本书有关 SPI 的案例中,将 SPI 初始化为主机模式的常见函数语句如下:

```
SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR1) | _BV(SPR0);
```

(22) SPI 数据寄存器——SPDR (SPI Data Register)

8 位的 SPI 数据寄存器 SPDR 为读/写寄存器,用来在通用寄存器和 SPI 移位寄存器之间传输数据。写 SPDR 将启动数据传输,读 SPDR 将读取接收缓冲器。

4.21 节“用带 SPI 接口的 MCP23S17 扩展 16 位通用 I/O 端口”、4.31 节“用 SPI 接口读/写 AT25F1024”等多个案例中使用了上 SPSR、SPCR、SPDR 寄存器。

(23) TWI 数据寄存器——TWDR (TWI Data Register)

TWI 即两线串行接口(Two-Wire Serial Interface)。在发送模式,TWDR 寄存器包含了要发送的字节;在接收模式,TWDR 寄存器包含了接收到的数据。

(24) TWI(从机)地址寄存器——TWAR (TWI Address Register)

TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
------	------	------	------	------	------	------	-------

TWAR 的 TWA6~TWA0 为从机地址,工作于从机模式时,TWI 将根据这个地址进行

响应,主机模式不需要此地址。在多主机系统中,TWAR 需要进行设置以便其他主机访问自己。TWGCE(TWI General Call Recognition Enable)位使能 TWI 广播识别,置位后 MCU 可以识别 TWI 总线广播。

(25) TWI 状态寄存器——TWSR (TWI Status Register)

TWS7	TWS6	TWS5	TWS4	TWS3	—	TWPS1	TWPS0
------	------	------	------	------	---	-------	-------

TWS7~TWS3 为 TWI 状态位,用来反映 TWI 逻辑和总线的状态。从 TWSR 读出的值包括 5 位状态值与 2 位预分频值,检测状态位时应屏蔽预分频位。

AVR-GCC 的头文件<util/twi.h>文件定义了所有的 TWI 状态,形如 TW_MT_xxx 与 TW_MR_xxx 的符号分别表示主机发送与接收,形如 TW_ST_xxx 与 TW_SR_xxx 的符号分别表示从机发送与接收。另外,在<util/twi.h>还有如下定义:

```
#define TW_STATUS_MASK \
(_BV(TWS7) | _BV(TWS6) | _BV(TWS5) | _BV(TWS4) | _BV(TWS3))
```

即 TW_STATUS_MASK=0B11111000,或 TW_STATUS_MASK=0xF8,将 TWI 状态寄存器 TWSR 与 TW_STATUS_MASK 进行“与”(&)操作即可获取高 5 位的状态码。<util/twi.h>所定义的 TWI 状态宏如下:

```
#define TW_STATUS (TWSR & TW_STATUS_MASK)
```

要在 C 程序中获取 TWI 状态,可直接读取 TW_STATUS。

TWSR 的最低 2 位 TWPS1、TWPS0 为 TWI 预分频位(TWI Prescaler Bits),可以读/写,用于控制比特率预分频因子。取值为 00、01、10、11 时,分别对应于 1、4、16、64 分频。

(26) TWI 控制寄存器——TWCR (TWI Control Register)

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	—	TWIE
-------	------	-------	-------	------	------	---	------

TWI 控制寄存器 TWCR 用来控制 TWI 操作。它用来使能 TWI,通过施加 START 到总线上来启动主机访问、产生接收器应答、产生 STOP 状态以及在写入数据到 TWDR 寄存器时控制总线的暂停等。这个寄存器还可以给出在 TWDR 无法访问期间,试图将数据写入到 TWDR 而引起的写入冲突信息。

TWINT 为 TWI 中断标志位(TWI Interrupt Flag Bit),当 TWI 完成当前工作,希望应用程序介入时 TWINT 置位。若 SREG 的 I 标志以及 TWIE 标志也置位,则 MCU 执行 TWI 中断例程。

当 TWINT 置位时,SCL 信号的低电平被延长。TWINT 标志的清零必须通过软件写 1 来完成。执行中断时硬件不会自动将其改写为 0。要注意的是,只要这一位被清零,TWI 立即开始工作。因此,在清零 TWINT 之前一定要首先完成对地址寄存器 TWAR、状态寄存器 TWSR 以及数据寄存器 TWDR 的访问。

TWEA(TWI Enable Acknowledge)控制应答脉冲的产生。若 TWEA 置位,则以下条件出现时接口发出 ACK 脉冲:

- ① 器件的从机地址与主机发出的地址相符合;
- ② TWAR 的 TWGCE 置位时接收到广播呼叫;

③ 在主机/从机接收模式下接收到一个字节的数据。

将 TWEA 清零可以使器件暂时脱离总线。置位后器件重新恢复地址识别。

TWSTA 为 TWI 启始状态位(TWI START Condition Bit)。当 CPU 希望自己成为总线上的主机时需要置位 TWSTA。TWI 硬件检测总线是否可用。若总线空闲,接口就在总线上产生 START 状态。若总线忙,接口就一直等待,直到检测到一个 STOP 状态,然后产生 START 以声明自己希望成为主机。发送 START 之后软件必须清零 TWSTA。

TWSTO 为 TWI 停止状态位(TWI STOP Condition Bit)。在主机模式下,如果置位 TWSTO,TWI 接口将在总线上产生 STOP 状态,然后 TWSTO 自动清零。在从机模式下,置位 TWSTO 可以使接口从错误状态恢复到未被寻址的状态。此时总线上不会有 STOP 状态产生,但 TWI 返回一个定义好的未被寻址的从机模式且释放 SCL 与 SDA 为高阻态。

TWWC 为 TWI 写碰撞标志(TWI Write Collision Flag)。当 TWINT 为低时写数据寄存器 TWDR 将置位 TWWC。当 TWINT 为高时,每一次对 TWDR 的写访问都将更新此标志。

TWEN 位于使能 TWI 操作(TWI Enable)及激活 TWI 接口。当 TWEN 位置为 1 时,TWI 将 I/O 引脚切换到 SCL 与 SDA 引脚,使能波形斜率限制器与尖峰滤波器。如果该位清零,TWI 接口模块将被关闭,所有 TWI 传输将被终止。

TWIE 位于使能 TWI 中断(TWI Interrupt Enable),当 SREG 的 I 以及 TWIE 置位时,只要 TWINT 为 1,TWI 中断即被激活。

以下是本书所有 TWI 接口器件的通用操作宏定义:

```
#define Wait() while ((TWCR & _BV(TWINT)) == 0)
#define START() {TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN); Wait();}
#define STOP() (TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN))
#define WriteByte(x) {TWDR = (x); TWCR = _BV(TWINT) | _BV(TWEN); Wait();}
#define ACK() (TWCR |= _BV(TWEA))
#define NACK() (TWCR &= ~_BV(TWEA))
#define TWI() (TWCR = _BV(TWINT) | _BV(TWEN); Wait();)
```

除 WriteByte(x)以外的所有宏定义全部与 TWI 控制寄存器 TWCR 相关。

4.22 节“用 TWI 接口控制 MAX6953 驱动 4 片 5×7 点阵显示器”、4.32 节“用 TWI 接口读/写 24C04”等数个案例中使用了与上述 TWI 接口操作相关的寄存器。

(27) TWI 比特率寄存器——TWBR (TWI Bit Rate Register)

8 位的 TWI 比特率寄存器用于设置比特率发生器分频因子。比特率发生器是一个分频器,在主机模式下产生 SCL 时钟频率,其计算公式如下:

$$\text{SCL 频率} = \text{CPU 时钟频率} / [16 + 2(\text{TWBR}) \times 4^{\text{TWPS}}]$$

其中 TWPS 是状态寄存器 TWSR 中由 TWPS1、TWSP0 设置的预分频值。本书案例中未设置 TWBR,其默认值为 0x00,TWSR 中的 TWPS1、TWSP0 也默认为 00,因此有:

$$\text{SCL 频率} = \text{CPU 时钟频率} / 16$$

(28) USART 波特率寄存器——UBRR(USART Baud Rate Registers)

URSEL	-	-	-	UBRR[11 : 8]
UBRR[7 : 0]				

16 位的 UBRR 由 UBRRH 与 UBRL 构成,其中 UBRRH 包含了 USART 波特率的高 4



位,UBRRL 包含了低 8 位。波特率的改变将造成正在进行的数据传输受到破坏,写 UBRRL 将立即更新波特率分频器。

UBRRH 的最高位 UBSEL 为寄存器选择位,通过该位选择访问 UCSRC 寄存器或 UBRRH 寄存器,因为 UBRRH 与 UCSRC 具有相同的地址。读 UBRRH 时该位为 0,写 UBRRH 时,URSEL 要置为 0。

异步正常模式的 UBRR 计算公式为:

$$\text{UBRR} = f_{\text{osc}} / 16/\text{BAUD}-1;$$

异步倍速模式(U2X=1)的 UBRR 计算公式为:

$$\text{UBRR} = f_{\text{osc}} / 8/\text{BAUD}-1;$$

同步主机模式的 UBRR 计算公式为:

$$\text{UBRR} = f_{\text{osc}} / 2/\text{BAUD}-1;$$

(29) USART I/O 数据寄存器——UDR (USART Data Registers)

USART 发送数据缓冲寄存器和 USART 接收数据缓冲寄存器共享相同的 I/O 地址,称为 USART 数据寄存器或 UDR。将数据写入 UDR 时实际操作的是发送数据缓冲器存器(TXB),读 UDR 时实际返回的是接收数据缓冲寄存器(RXB)的内容。

(30) USART 控制和状态寄存器 A——UCSRA(USART Control and Status Register A)

RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
-----	-----	------	----	-----	----	-----	------

RXC 与 TXC 分别为 USART 接收结束(Receive Complete)和发送结束(Transmit Complete)标志位,可分别用于产生接收结束中断和发送结束中断。

UDRE 为 USART 数据寄存器空标志(USART Data Register Empty flag),用于指出发送缓冲器(UDR)是否准备好接收新数据。UDRE 为 1 说明缓冲器为空,已准备好进行数据接收。UDRE 标志可用来产生数据寄存器空中断。复位后 UDRE 置位,表明发送器已经就绪。

U2X 为倍速发送位(Double the USART Transmission Speed Bit),这一位仅对异步操作有影响。使用同步操作时将此位清零。此位置 1 可将波特率分频因子从 16 降到 8,将异步通信模式的传输速率倍增。

其他寄存器位的说明可参考 ATmega16 的技术手册文件。

(31) USART 控制和状态寄存器 B——UCSRB(USART Control and Status Register B)

RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
-------	-------	-------	------	------	-------	------	------

RXCIE 与 TXCIE 分别为接收结束中断使能位(RX Complete Interrupt Enable Bit)与发送结束中断使能位(TX Complete Interrupt Enable Bit)。

UDRIE 为 USART 数据寄存器空中断使能位(USART Data Register Empty Interrupt Enable Bit)。

RXEN 与 TXEN 分别使能接收与使能发送。

UCSZ2 为字符长度位 2(Character Size Bit2),它与 UCSRC 寄存器的 UCSZ1:0 结合在一起设置数据帧所包含的数据位数(字符长度)。

RXB8 为接收数据位 8,对 9 位串行帧进行操作时,RXB8 是第 9 个数据位。读取 UDR 包含的低位数据之前首先要读取 RXB8。

TXB8 为发送数据位 8,对 9 位串行帧进行操作时,TXB8 是第 9 个数据位。写 UDR 之前

首先要对它进行写操作。

(32) USART 控制和状态寄存器 C——UCSRC(USART Control and Status Register C)

URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
-------	-------	------	------	------	-------	-------	-------

UCSRC 寄存器与 UBRRH 寄存器共用相同的 I/O 地址, 寄存器选择位 URSEL 用于选择访问 UCSRC 寄存器或 UBRRH 寄存器。读 UCSRC 时该位为 1, 当写 UCSRC 时 URSEL 要置为 1。

UMSEL 为 USART 模式选择位(USART Mode Select Bit), 用于选择同步/异步工作模式。UMSEL 默认为 0, 选择异步工作模式, 置为 1 时选择同步模式。在同步模式下, PB0(XCK)引脚的数据方向寄存器 DDR_XCK 决定时钟源是由内部产生(主机模式)还是由外部产生(从机模式), XCK 仅在同步模式下有效。

UPM1、UPM0 用于设置 USART 奇偶校验模式(USART Parity Mode)。

USBS 为停止位设置, 取值为 0、1 时, 停止位分别为 1、2。USBS 默认为 0, 停止位为 1 位。

UCSRC 中的 UCSZ1、UCSZ0 与 UCSRB 中 UCSZ[2:0]用于设置数据帧包含的数据位数。取值为 000~011 时, 分别设置长度为 5、6、7、8 位, 100~110 为保留值, 取值 111 时设置长度为 9 位。其默认值为 011, 设置长度为 8 位。

本书 USART 初始化函数中的常见语句如下:

```
UCSRB = _BV(RXEN) | _BV(TXEN) | _BV(RXCIE);           //允许接收和发送,接收中断使能
UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0);         //8 位数据位,1 位停止位(此行可省)
UBRRL = (F_CPU/9600/16 - 1) * 256;                   //这两行根据 UBRR 的计算公式
UBRRH = (F_CPU/9600/16 - 1)/256;                     //设置波特率 9 600
```

最后两行还可以改写成:

```
UBRRL = (F_CPU/9600/16 - 1) & 0xFF;
UBRRH = (F_CPU/9600/16 - 1) >> 8;
```

AVR-GCC 不允许直接设置 UBRR 寄存器, UBRR=F_CPU/9600/16-1 这种“最简写法”是不允许的。

对于 UCSRA、UCSRB、UCSRC 及 URBB 寄存器, 上述语句设置了除 UCSRA 以外的其他几个寄存器, 其中第二行语句对 UCSRC 的设置与默认值相同, 该行语句可以省略。如果要进一步设置异步倍速模式, 则还需要设置 UCSRA 中的 U2X 位, 即:

```
UCSRA |= _BV(U2X); //异步倍速模式
```

在 USART 程序中, PutChar 函数通过 UDR 发送一个字节数据后, 接着执行语句:

```
while(!(UCSRA & _BV(UDRE)));
```

它循环检查 UCSRA 寄存器中的空标志位 UDRE, 直到数据缓冲器 UDR 为空(Empty)。

3.34 节“单片机与 PC 机双向串口通信仿真”、4.17 节“2×20 串行字符液晶演示”及 5.25 节“能接收串口信息的带中英文硬字库的 80×16 多汉字点阵显示屏”、5.29 节“PC 机通过 485 远程控制单片机”等多个案例使用了上述 USART 相关寄存器。

通用同步/异步串行接收器和转发器(USART)是一个高度灵活的串行通信设备, 当前版



本的 Proteus 中,ATmega16 仅仅能仿真通用异步串行收发器(UART),但对奇偶校验及倍速模式还不支持。对于通用同步串行收发器(USART)则还完全不能仿真,后续版本会将会实现对同步模式的仿真。

(33) ADC 多工选择寄存器——ADMUX (ADC Multiplexer Selection Register)

REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
-------	-------	-------	------	------	------	------	------

REFS1 与 REFS0(Reference Selection Bits1~0)为参考电压选择位,这两位的默认值为 00,对应选择 AREF 引脚参考电压,内部 V_{REF} 关闭。

ADLAR(ADC Left Adjust Result)为 ADC 转换结果左对齐选择位,它影响 ADC 转换结果在 ADC 数据寄存器中的存放形式,其取值默认为 0,为右对齐方式,ADLAR 置位时则选择左对齐方式。ADLAR 的改变将立即影响 ADC 数据寄存器的内容,不论是否有转换正在进行。

MUX4~MUX0 为模拟通道与增益选择位(Analog Channel and Gain Selection Bits),通过这 5 位可以选择连接到 ADC 的模拟输入通道,也可对差分通道增益进行选择:

MUX4~MUX0 取值为 00000~00111 可分别选择单端输入通道 ADC0~ADC7。本书 AVR-GCC 程序在选择单端模拟输入通道时,直接使用 $ADMUX = 0x00 \sim 0x07$,或使用 $ADMUX = ch$,其中 INT8U 类型的变量 ch 取值为 8 个 ADC 通道之一(0~7)。

MUX4~MUX0 取值为 01000~11100 用于设置差分通道增益。相关设置可参阅技术手册文件。

(34) ADC 控制和状态寄存器 A——ADCSRA (ADC Control and Status Register A)

ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
------	------	-------	------	------	-------	-------	-------

ADEN(ADC ENable)为 ADC 使能位,置位即可启动 ADC,否则 ADC 功能关闭。在转换过程中关闭 ADC 将立即中止正在进行的转换。

ADSC(ADC Start Conversion)为 ADC 开始转换位,在单次转换模式下,ADSC 置位将启动一次 ADC 转换。在连续转换模式下,ADSC 置位将启动首次转换。第一次转换(在 ADC 启动之后置位 ADSC,或者在使能 ADC 的同时置位 ADSC)需要 25 个 ADC 时钟周期,而不是正常情况下的 13 个。第一次转换执行 ADC 初始化的工作。

在转换进行过程中读取 ADSC 的返回值为 1,直到转换结束。ADSC 清零不产生任何动作。

ADATE(ADC Auto Trigger Enable)为 ADC 自动触发使能位,其置位将启动 ADC 自动触发功能。触发信号的上跳沿启动 ADC 转换。触发信号源通过 SFIOR 寄存器的 ADC 触发信号源选择位 ADTS 设置。

ADIF(ADC Interrupt Flag)为 ADC 中断标志,在 ADC 转换结束且数据寄存器被更新后,ADIF 置位。如果 ADIE 及 SREG 中的全局中断使能位 I 也置位,ADC 转换结束中断服务程序即得以执行,同时 ADIF 硬件清零。此外,还可以通过向此标志写 1 来清 ADIF。要注意的是,如果对 ADCSRA 进行读/修改/写操作,那么待处理的中断会被禁止。

ADIE(ADC Interrupt Enable)为 ADC 中断使能位,若 ADIE 及 SREG 的位 I 置位,ADC 转换结束中断即被使能。

ADPS2~ADPS0(ADC Prescaler Select Bits2~0)为 ADC 预分频器选择位,用来确定

XTAL 与 ADC 输入时钟之间的分频因子。取值 000~111 时, 分别对应于分频因子: 2、2、4、8、16、32、64、128。其中默认值 000 与取值 001 所设置的分频因子都是 2。

(35) ADC 数据寄存器——ADCL 及 ADCH (ADC Data Register-ADCL and ADCH)

当 ADLAR=0 时, ADCL 与 ADCH 的数据默认为右对齐, 其格式如下:

—	—	—	—	—	—	ADC9	ADC8
ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0

当 ADLAR=1 时, ADCL 与 ADCH 的数据选择左对齐, 其格式如下:

ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
ADC1	ADC0	—	—	—	—	—	—

ADC 转换结束后, 转换结果存放于这 2 个寄存器之中。如果采用差分通道, 结果由 2 的补码形式表示。读取 ADCL 之后, ADC 数据寄存器一直要等到 ADCH 也被读出才可以进行数据更新。因此, 如果转换结果为左对齐, 且要求的精度不高于 8-bit, 那么仅须读取 ADCH 就足够了。否则必须先读出 ADCL 再读 ADCH。

对于单端通道, 转换结果为: $ADC = V_{IN} \times 1024 / V_{REF}$ 。

其中 V_{IN} 为被选中通道的输入引脚电压, V_{REF} 为参考电压。

在差分模式下, 结果为: $ADC = (V_{POS} - V_{NEG}) \times GAIN \times 512 / V_{REF}$ 。

其中, V_{POS} 为输入引脚正电压, V_{NEG} 为输入引脚负电压, GAIN 为选定的增益因子。

在 AVR-GCC 程序中, 可以使用下面的语句读取转换结果:

```
Result = (INT16U)(ADCL + (ADCH << 8));
```

也可以直接读取 ADC 的结果, 即“Result=ADC;”, 其中 Result 为 16 位的整型变量。

3.30 节“数码管显示两路 A/D 转换结果”、4.33 节“MPX4250 压力传感器测试”、5.27 节“电子秤仿真设计”等数个案例中使用了上述 ADC 相关寄存器。

(36) 模拟比较器控制和状态寄存器——ACSR (Analog Comparator Control and Status Register)

ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
-----	------	-----	-----	------	------	-------	-------

ACD(Analog Comparator Disable)为模拟比较器禁用位, ACD 置位时, 模拟比较器的电源被切断。可以在任何时候设置此位来关掉模拟比较器, 这可以减少器件工作模式及空闲模式下的功耗。改变 ACD 位时, 必须清零 ACSR 寄存器的 ACIE 位来禁止模拟比较器中断。否则 ACD 改变时可能会产生中断。

ACBG(Analog Comparator Bandgap Select)选择模拟比较器的能隙基准源, ACBG 置位后, 模拟比较器的正极输入由能隙基准源所取代。否则, AIN0 连接到模拟比较器的正极输入。

ACO(Analog Comparator Output)为模拟比较器输出位, 模拟比较器的输出经过同步后直接连到 ACO。同步机制引入了 1~2 个时钟周期的延时。

ACI(Analog Comparator Interrupt Flag)为模拟比较器中断标志位, 当比较器的输出事件触发了由 ACIS1 及 ACIS0 定义的中断模式时, ACI 置位。如果 ACIE 和 SREG 寄存器的全局中断标志 I 也置位, 那么模拟比较器中断服务程序即得以执行, 同时 ACI 被硬件清零。



ACI 也可以通过写 1 来清零。

ACIE(Analog Comparator Interrupt Enable)为模拟比较器中断使能位,当 ACIE 置 1 且状态寄存器中的全局中断标志 I 也被置位时,模拟比较器中断被激活。否则中断被禁止。

ACIC(Analog Comparator Input Capture Enable)为模拟比较器输入捕捉使能位,ACIC 置位后允许通过模拟比较器来触发 T/C1 的输入捕捉功能。此时比较器的输出被直接连接到输入捕捉的前端逻辑,从而使得比较器可以利用 T/C1 输入捕捉中断逻辑的噪声抑制器及触发沿选择功能。ACIC 为 0 时模拟比较器及输入捕捉功能之间没有任何联系。为了使比较器可以触发 T/C1 的输入捕捉中断,定时器中断屏蔽寄存器 TIMSK 的 TICIE1 必须置位。

ACIS1、ACIS0(Analog Comparator Interrupt Mode Select1、0)为模拟比较器中断模式选择位,这两位确定触发模拟比较器中断的事件,所有取值组合如下:

00——比较器输出变化即可触发中断;

01——保留;

10——比较器输出的下降沿产生中断;

11——比较器输出的上升沿产生中断。

3.31 节“模拟比较器测试”案例中使用了 ACSR 寄存器及下面的 SFIOR 寄存器。

(37) 特殊功能 I/O 寄存器——SFIOR(Special Function I/O Register)

ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSR2	PSR10
-------	-------	-------	---	------	-----	------	-------

ADTS2~ADTS0 为 ADC 自动触发源(ADC Auto Trigger Source)选择位,若 ADCSRA 寄存器的 ADATE 置位,ADTS 的值将确定触发 ADC 转换的触发源,否则 ADTS 的设置无任何意义。被选中的中断标志在其上升沿触发 ADC 转换。从一个中断标志清零的触发源切换到中断标志置位的触发源会使触发信号产生一个上升沿。如果此时 ADCSRA 寄存器的 ADEN 为 1,ADC 转换即被启动。切换到连续运行模式(ADTS[2:0]=0)时,即使 ADC 中断标志已经置位也不会产生触发事件。

下面列出的是所有的 ADC 自动触发源选择组合:

000——连续转换模式(默认组合);

001——模拟比较器;

010——外部中断请求 0;

011——定时/计数器 0 比较匹配;

100——定时/计数器 0 溢出;

101——定时/计数器比较匹配 B;

110——定时/计数器 1 溢出;

111——定时/计数器 1 捕捉事件。

ACME 为模拟比较器多路复用器使能(Analog Comparator Multiplexer Enable)位。该位置位且 ADC 处于关闭状态时(ADCSRA 寄存器的 ADEN 为 0),ADC 多路复用器为模拟比较器选择负极输入。当此位为 0 时,AIN1 连接到比较器的负极输入端。

PUD(Pull - Up Disable)为禁用上拉电阻位。置位时,即使将寄存器 DDxn 和 PORTxn 配置为使能上拉电阻 ($\{DD_{xn}, PORT_{xn}\} = 0b01$),I/O 端口的上拉电阻也被禁止。

PSR2(PreScaler Reset Timer/Counter2)用于复位预分频 T/C2 预分频器。该位置 1 时

T/C2 预分频器复位。操作完成后，该位被硬件清零，该位写 0 无效。若内部 CPU 时钟作为 T/C2 时钟，该位读为 0。当 T/C2 工作在异步模式时，直到预分频器复位该位保持为 1。

PSR10(PreScaler Reset Timer/Counter1 and Timer/Counter0)用于复位 T/C1 与 T/C0 预分频器，置位时 T/C1 与 T/C0 的预分频器复位。操作完成后这一位由硬件自动清零。写入零时不会引发任何动作。T/C1 与 T/C0 共用同一预分频器，且预分频器复位对 2 个定时器均有影响。该位总是读为 0。

(38) 看门狗定时器控制寄存器——WDTCR(WatchDog Timer Control Register)

—	—	—	WDTOE	WDE	WDP2	WDP1	WDP0
---	---	---	-------	-----	------	------	------

WDTOE(WatchDog Turn - Off Enable)为看门狗关闭使能位，清零 WDE 时必须置位 WDTOE，否则不能禁止看门狗。一旦置位，硬件将在紧接的 4 个时钟周期之后将其清零。

WDE(WatchDog Enable)看门狗使能位，WDE 为 1 时使能看门狗，否则看门狗将被禁止。只有在 WDTOE 为 1 时 WDE 才能清零。

看门狗定时器由独立的 1 MHz 片内振荡器驱动，WDP2~WDP0(Watchdog Timer Prescaler 2~0)用于设置看门狗定时器预分频器，当 VCC=5 V 时，WDP2~WDP0 取值 000~111 所设置的超时值为 16.3 ms、32.5 ms、65 ms、0.13 s、0.26 s、0.52 s、1.0 s、2.1 s。

3.35 节“看门狗应用”案例中，C 程序引入了看门狗头文件<avr/wdt.h>，该头文件提供了以下用于看门狗控制的宏定义：

- ① `wdt_reset()` 复位看门狗，它调用看门狗复位汇编指令 WDR(watchdog reset)。
- ② `wdt_disable()` 禁止看门狗，它将 WDTCR 寄存器的 WDE 位清零。
- ③ `wdt_enable(value)` 配置超时值，并置位 WDE，使能看门狗。其中 value 的取值影响 WDTCR 寄存器的低 3 位 WDP2~WDP0，取值为符号常量 WDTO_15MS、WDTO_30MS、WDTO_60MS、WDTO_120MS、WDTO_250MS、WDTO_500MS、WDTO_1S、WDTO_2S，它们分别对应于 0~7(即 000~111)。这 8 个符号常量分别将超时值设为 15 ms~2 s 的近似值，符号定义中 WDTO 表示看门狗超时值(WatchDog Timeout)。

由于<avr/wdt.h>提供了上述宏定义与超时值符号常量，在 AVR-GCC 程序中不需要直接访问看门狗定时器控制寄存器 WDTCR。

本书案例使用了本节列出的大部分寄存器，通过 AVR 单片机的 C 语言程序设计实训，读者要进一步熟练掌握这些寄存器的应用。

在 AVR Studio 开发环境中，图 1-3 中右边的 I/O 窗口(I/O View)会列出当前 MCU 的所有寄存器及各寄存器位，当用鼠标指针指向寄存器或寄存器位时，会出现相应的提示信息。

另外，在 ATmega 系列各种型号单片机的技术手册文件中，其最后附录有一张寄存器摘要表(Register Summary)，其中 ATmega16(L)的寄存器摘要如表 1-3 所列，表中列出了所有寄存器及各寄存器位。该表可进一步方便大家全面了解 ATmega 系列单片机的所有寄存器，以及对各寄存器相关技术细节的阅读与研究。

表 1-3 中的寄存器和大部分寄存器位定义在头文件<iom16.h>中，在访问这些寄存器时，恰当使用所定义的“位”名称可大大提高程序的可读性。



表 1-3 ATmega16 寄存器摘要表

名称	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	手册页码
SREG	I	T	H	S	V	N	Z	C	7
SPH	—	—	—	—	—	SP10	SP9	SP8	10
SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	10
OCR0	T/C0 比较输出寄存器								80
GICR	INT1	INT0	INT2	—	—	—	IVSEL	IVCE	46, 66
GIFR	INTF1	INTF0	INTF2	—	—	—	—	—	67
TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	80, 107, 123
TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	80, 108, 123
SPMCR	SPMIE	RWWBSB	—	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN	238
TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	—	TWIE	169
MCUCR	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	30, 65
MCUCSR	JTD	ISC2	—	JTRF	WDRF	BORF	EXTRF	PORF	39, 66, 218
TCCR0	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	78
TCNT0	T/C0(8 位)								80
OSCCAL	振荡器校准寄存器								27
OCDR	片上调试寄存器								214
SFIOR	ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSR2	PSR10	55, 82, 124, 189, 207
TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	103
TCCR1B	ICNC1	ICES1	—	WGM13	WGM12	CS12	CS11	CS10	106
TCNT1H	T/C1—计数器寄存器高字节								106
TCNT1L	T/C1—计数器寄存器低字节								106
OCR1AH	T/C1—输出比较寄存器 A 高字节								106
OCR1AL	T/C1—输出比较寄存器 A 低字节								106
OCR1BH	T/C1—输出比较寄存器 B 高字节								106
OCR1BL	T/C1—输出比较寄存器 B 低字节								106
ICR1H	T/C1—输入捕获寄存器高字节								106
ICR1L	T/C1—输入捕获寄存器低字节								106
TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	120
TCNT2	T/C2(8 位)								120
OCR2	T/C2 输出比较寄存器								121
ASSR	—	—	—	—	AS2	TCN2UB	OCR2UB	TCR2UB	122
WDTCR	—	—	—	WDTOE	WDE	WDP2	WDP1	WDP0	40
UBRRH	URSEL	—	—	—	UBRR[11 : 8]				156
UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	154
EEARH	—	—	—	—	—	—	—	EEAR8	17
EEARL	EEPROM 地址寄存器低字节								17

续表 1-3

名称	位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0	手册页码
EEDR	EEPROM 数据寄存器								17
EECR	—	—	—	—	EERIE	EEMWE	EEWE	EERE	17
PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	63
DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	63
PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	63
PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	63
DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	63
PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	63
PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	64
DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	64
PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	64
PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	64
DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	64
PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	64
SPDR	SPI 数据寄存器								131
SPSR	SPIF	WCOL	—	—	—	—	—	SPI2X	131
SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	129
UDR	USART I/O 数据寄存器								152
UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	152
UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	153
UBRRL	USART 波特率寄存器低字节								156
ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	190
ADMUX	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	204
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	206
ADCH	ADC 数据寄存器高字节								207
ADCL	ADC 数据寄存器低字节								207
TWDR	两线串行接口数据寄存器								170
TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	171
TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	—	TWPS1	TWPS0	170
TWBR	两线串行接口位率寄存器								169

1.7 中断服务程序

本书案例中使用最频繁的 ATmega16(L) 单片机具有 20 个中断源和 1 个复位中断。表 1-4 给出了该单片机完整的中断向量表，处于低地址的中断源具有更高的中断优先级，所有中断源都有独立的中断使能位 (Interrupt Enable Bit)，当相应的中断使能位和全局中断使



能位(SREG 寄存器中的 I 位)置 1 时才允许触发中断,相应的中断服务器程序才能被自动调用。

在 AVR-GCC(4.3.2 及以上版本)开发环境中,中断向量表通过预先确定的名称指向预先定义的中断服务程序,通过中断向量名称可使得中断发生时相应的 ISR 被调用。表 1-4 列出了 ATmega16(L)的所有中断向量名,每个中断向量名都以_vect 结尾(vector,向量)。

表 1-4 ATmega16(L)的中断向量表

向量号	程序地址	中断向量名称	中断定义
1	\$000	无	外部引脚电平引发的复位、上电复位、掉电测试复位、看门狗复位以及 JTAG AVR 复位
2	\$002	INT0_vect	外部中断请求 0
3	\$004	INT1_vect	外部中断请求 1
4	\$006	TIMER2_COMP_vect	定时/计数器 2 比较匹配
5	\$008	TIMER2_OVF_vect	定时/计数器 2 溢出
6	\$00A	TIMER1_CAPT_vect	定时/计数器 1 事件捕获
7	\$00C	TIMER1_COMPA_vect	定时/计数器 1 比较匹配 A
8	\$00E	TIMER1_COMPB_vect	定时/计数器 1 比较匹配 B
9	\$010	TIMER1_OVF_vect	定时/计数器 1 溢出
10	\$012	TIMER0_OVF_vect	定时/计数器 0 溢出
11	\$014	SPI_STC_vect	SPI 串行传输结束
12	\$016	USART_RXC_vect	USART, Rx 结束
13	\$018	USART_UDRE_vect	USART 数据寄存器空
14	\$01A	USART_TXC_vect	USART, Tx 结束
15	\$01C	ADC_vect	ADC 转换结束
16	\$01E	EE_RDY_vect	EEPROM 就绪
17	\$020	ANA_COMP_vect	模拟比较器
18	\$022	TWI_vect	两线串行接口
19	\$024	INT2_vect	外部中断请求 2
20	\$026	TIMER0_COMP_vect	定时/计数器 0 比较匹配
21	\$028	SPM_RDY_vect	保存程序存储器内容就绪

在本书所使用的 AVR-GCC 版本下,所有的中断服务程序都使用宏 ISR() 定义,其中 ISR 即中断服务程序(Interrupt Service Routine),也称为中断例程。

下面是某中断服务程序示例:

```
#include <avr/interrupt.h>
ISR (XXX_vect)
{
    //中断发生后要执行的代码
}
```

中断发生后，在执行中断服务程序之前，单片机硬件将清除 SREG 中的全局中断使能标志位 I，后续中断将被禁止，直到该处理程序完成中断处理后退出，从而许可后续中断。显然，此时的中断函数是不能嵌套调用的，而且很多中断本来就不允许嵌套，以免出现无限递归，例如 UART 中断和设为电平触发 (level-triggered) 方式的外部中断。

在某些情况下，为了避免延误中断处理或为了响应更高优先级的中断事件，可在中断服务程序内最前面使用 sei() 打开中断，置位 SREG 寄存器的位 I。

使用中断函数内置位 I 的方法，编译程序仍会先生成禁止全局中断的代码。为了不阻塞后续中断的处理，另一种方法是按下面的格式编写中断服务程序：

```
ISR (XXX_vect, ISR_NOBLOCK)
{
    //中断发生后要执行的代码
}
```

其中 ISR_NOBLOCK 表示不阻塞中断处理。

当前版本的 AVR-GCC 还支持共享中断服务程序，编写示例如下：

```
ISR (XXX0_vect)
{
    //中断发生后要执行的代码
}
ISR (XXX1_vect, ISR_ALIASOF(XXX0_vect));
```

第 2 个中断服务程序没有语句体，它通过中断别名宏 ISR_ALIASOF 实现对 XXX0_vect 中断服务程序的共享。

第 3 章有关案例中编写了外部中断、定时器溢出中断、比较匹配中断、串行异步接收中断、模拟比较中断、输入捕获中断等中断的服务程序，程序中提供了很详细的注释语句，对这些中断程序的设计要熟练掌握。

1.8 GCC 在 AVR 单片机应用系统开发中的优势

GCC 编译器是一种被广泛使用的 C 编译器，它本是 GNU 项目的一个组成部分。用于开发 AVR 单片机的 GCC 编译器称为 AVR-GCC（也称为 gcc-avr）。

目前，AVR-GCC 可运行于多种主流操作系统，各种不同操作系统平台的开发人员都可以用 AVR-GCC 开发 AVR 单片机 C 程序。AVR-GCC 支持绝大部分的 AVR 单片机，而且受支持的单片机数目还在不断扩展。

使用 AVR-GCC 开发单片机应用程序的优势如下：

- 在不了解 AVR 单片机指令系统，仅熟悉单片机存储结构及相关寄存器时就可以开发单片机应用程序；
- 寄存器分配和不同存储器寻址及数据类型等细节可由编译器管理；
- 程序可分解为多个不同函数，便于进行结构化程序设计，所编写的程序可读性强；
- avr-libc 提供了丰富的库函数，数据处理能力、存储器访问能力很强；



- 程序编写及调试时间大大缩短,开发效率远高于汇编语言;
- 通用的 GCC 程序模块很容易植入新的程序项目,可进一步提高开发效率;
- 免费的开发环境可大大降低项目的开发成本。

安装好 WinAVR 以后即可在 Windows 操作系统下开始基于 AVR-GCC 的单片机应用程
序开发,WinAVR 提供 Programmers Notepad 用于源程序编辑,Mfile 用于创建或编辑 Make-
file 文件。当然,更为方便的是利用 AVR Studio+ WinAVR 组合平台开发 AVR 单片机 GCC
程序,这一平台实际上是 AVR Studio 与 AVR-GCC 的组合。在该平台下,Makefile 文件的创
建与配置由 AVR Studio 自动完成,程序的编译、调试、下载、仿真等可在—个理想的集成环
境中完成。

本书提供了 100 个 AVR 单片机 GCC 程序设计案例,各案例均有完整的 Proteus 仿真电
路图及 GCC 程序源代码,通过对这些案例的学习研究、调试、仿真及实训设计,AVR 单片机的
C 程序开发能力一定会得到大幅提高。

第 2 章

Proteus 操作基础

Proteus 是英国 Labcenter 公司开发的电路分析与实物仿真及印制电路板设计软件, 可以仿真、分析各种模拟电路与集成电路。系统提供了大量模拟与数字元器件及外部设备, 各种虚拟仪器, 例如电压表、电流表、示波器、逻辑分析仪、信号发生器等, 特别是它具备对单片机及外围电路组成的综合应用系统的交互仿真功能。

目前, Proteus 仿真系统支持的主流单片机有 ARM7(LPC21xx)、8051/52 系列、AVR 系列、PIC10/12/16/18/24 系列、HC11 系列等, 它支持的第三方软件开发、编译和调试环境有 Keil μ Vision2/3、MPLAB、AVR Studio 等。

2.1 Proteus 操作界面简介

Proteus 主要由 ISIS 和 ARES 两部分组成, ISIS 的主要功能是原理图设计、与电路原理图的交互仿真, ARES 主要用于印制电路板设计。

ISIS 提供的 Proteus VSM(Virtual System Modelling)实现了混合式的 SPICE(Simulation Program with Integrated Circuit Emphasis)电路仿真, 它将虚拟仪器、高级图表应用、单片机仿真、第三方程序开发与调试环境有机结合, 在搭建硬件模型之前即可在 PC 上完成原理图设计、电路分析与仿真以及单片机程序实时仿真、测试及验证。

图 2-1 是 Proteus ISIS 7.5 操作界面, 窗体左边是由 3 个部分组成的模式选择工具栏, 主要包括主模式图标、部件模式图标和二维图形模式图标。下面给出了这些模式图标功能的简要说明。

(1) 主模式图标

选择模式(Selection Mode)——在选取仿真电路图中的元件等对象时使用该图标模式;

元器件模式(Component Mode)——用于打开元件库选取各种元器件;

连接点模式(Junction Dot Mode)——用于在电路中放置连接点;

连线标签模式(Wire Label Mode)——用于放置或编辑连线标签;

文本脚本模式(Text Script Mode)——用于在电路中输入或编辑文本;

总线模式(Buses Mode)——用于在电路中绘制总线;

子电路模式(Subcircuit Mode)——用于在电路中放置子电路框图或放置子电路元器件。

(2) 部件模式图标

终端模式(Terminals Mode)——提供各种终端, 如输入、输出、电源、地等;

设备引脚模式(Device Pins Mode)——提供 6 种常用的元件引脚;



图形模式(Graph Mode)——列出可供选择的各种仿真分析所需要的图表,如模拟分析图表、数字分析图表、频率响应图表等;

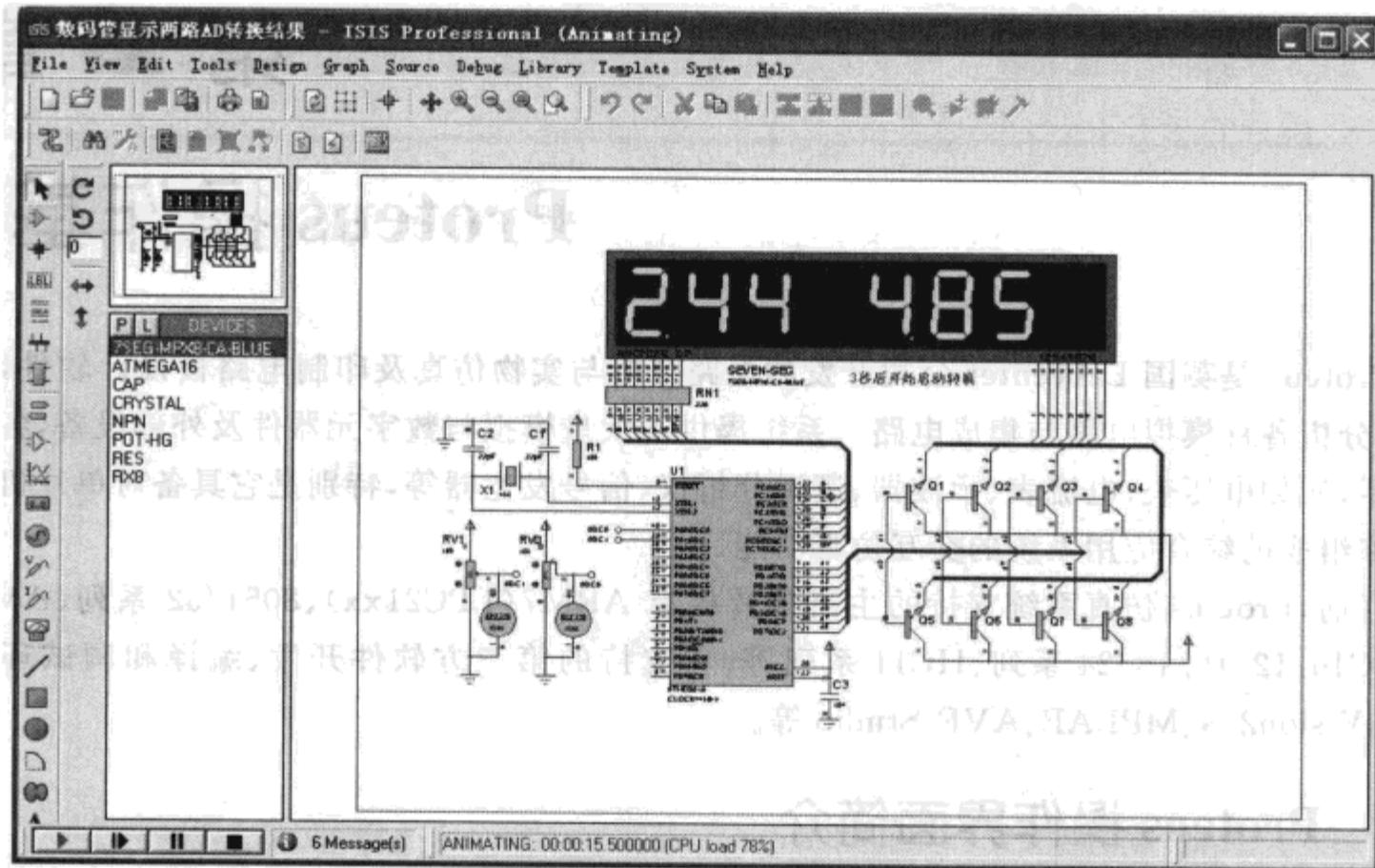


图 2-1 Proteus ISIS7.5 操作界面

磁带记录器模式(Tape Recorder Mode)——对原理图分析分割仿真时用来记录前一步的仿真输出,作为下一步仿真的输入;

发生器模式(Generator Mode)——用于列出可供选择的模拟和数字激励源,例如正弦波信号、数字时钟信号及任意逻辑电平序列等;

电压探针模式(Voltage Probe Mode)——用于记录模拟或数字电路中探针处的电压值;

电流探针模式(Current Probe Mode)——用于记录模拟电路中探针处的电流值;

虚拟仪器(Virtual Instruments Mode)——提供的虚拟仪器有示波器、逻辑分析仪、虚拟终端、SPI 调试器、I²C 调试器、直流与交流电压表、直流与交流电流表。

(3) 二维图形模式图标

直线模式(2D Graphics Line Mode)——用于在创建元件时绘制直线,或者直接在原理图中绘制直线;

框线模式(2D Graphics Box Mode)——用于在创建元件时绘制矩形框,或者直接在原理图中绘制矩形框;

圆圈模式(2D Graphics Circle Mode)——用于在创建元件时绘制圆圈,或者直接在原理图中绘制圆圈;

封闭路径模式(2D Graphics Close Path Mode)——用于在创建元件时绘制任意多边形,或者直接在原理图中绘制多边形;

文本模式(2D Graphics Text Mode)——用于在原理图中添加说明文字;

符号模式(2D Graphics Symbol Mode)——用于从符号库中选择各种元件符号;

标记模式(2D Graphics Markers Mode)——用于在创建或编辑元器件、符号、终端、引脚时产生各种标记图标。

以上介绍了 Proteus 模式工具栏中的各种操作模式图标,紧挨着模式工具栏的另一工具栏上有对象旋转与镜像按钮,其右边的两个小窗口分别是预览窗口和对象选择窗口。预览窗口显示的是当前仿真电路的缩略图;对象选择窗口中列出的是当前仿真电路中用到的所有元件、可用的所有终端、所有虚拟仪器等,当前所显示的可选择对象与当前所选择的操作模式图标对应。

Proteus 主窗体右边的大面积区域是仿真电路原理图(Schematic)编辑窗口,仿真电路原理图的设计与编辑将在 2.2 节中介绍。Proteus 主窗体最下面还有仿真运行、暂停及停止等控制按钮。

2.2 仿真电路原理图设计

本书案例以 ATmega 系列单片机为核心,在设计原理图时根据当前电路复杂程度和特定要求,可在 Proteus 提供的模板中选择恰当的模板进行设计。打开模板时可单击“文件”→“新建设计”(File→New Design)菜单命令打开“创建新设计”(Create New Design)对话框,然后选择相应模板。直接单击工具栏上的“新文件”(New File)按钮时,Proteus 会以默认模板建立原理图文件,调整图纸大小或样式时可单击“系统”→“设置图纸尺寸”(System→Set Paper Size)菜单进行设置,默认图纸背景是灰色的,如果需要改成本书所有案例使用的白色,可单击“模板”→“设置设计默认值”(Template→Set Design Default)菜单命令,将对话框中的“图纸颜色”(Paper Colour)改成白色。

创建空白文件后,建议在开始后续操作之前先将 DSN 文件保存到指定位置,然后向图纸中添加元件。单击模式工具栏上的元件模式(Component Mode)图标,对象选择窗口上会显示出设备 DEVICE。对于空白 DSN 文件,对象选择器中不会显示任何元件。这时可单击“P”(Pick)按钮打开图 2-2 所示的元件选择窗口,在元件库中选择各种模拟元件、数字芯片、微控制器、光电元件、机电元件、显示器件等,2.3 节会给出元件的分类介绍。

放置在图纸中的所有元件旁边都会出现 TEXT,单击“模板/设置设计默认值”菜单,在打开的窗口中取消选择“显示隐藏文本”(Show hidden text?)可快速隐藏所有<TEXT>。

放置元件后,用左键或右键单击都可以选中元件,在元件上双击可打开元件属性窗口,先单击右键再单击左键也可以打开属性窗口,右键双击则会删除元件。主工具栏上还提供了在当前电路图内块复制(Block Copy)、块移动(Block Move)元件或子电路的红绿色相间的工具按钮、对于选取的块电路通过右键菜单“复制到剪贴板”(Copy to Clipboard)可以很方便地将部分或全部电路或元件复制到其他 DSN 文件中。

放置元件后即可开始连线。当鼠标指向连线的起始引脚时,在起始引脚上会出现红色小方框,这时单击,然后移动鼠标指向终点引脚再单击,连线即成功完成。如果连线过程中要按自己的要求拐弯时,只需在移动鼠标的路径上单击要拐弯的地方即可。移动鼠标时还可以配合 CTRL 按键,这样的连线会保持水平或垂直。

如果电路中并行的连线较多或连接线路较长,这时可以使用模式工具栏中的总线模式

(Buses Mode)图标绘制总线。绘制总线后,将起点出发的连线和到终点的连线都连接到总线上。需要注意的是,这样连线时必须给各连线加上标签(Label),标有同名标签的连线被认为是连通的,加标签时可直接在连线上右击,选“Place Wire Label”命令,或先单击模式工具栏中的标签模式(Label Mode)图标,然后用鼠标指向连线,在连线上出现“×”号时单击,在弹出的对话框中输入标签即可。

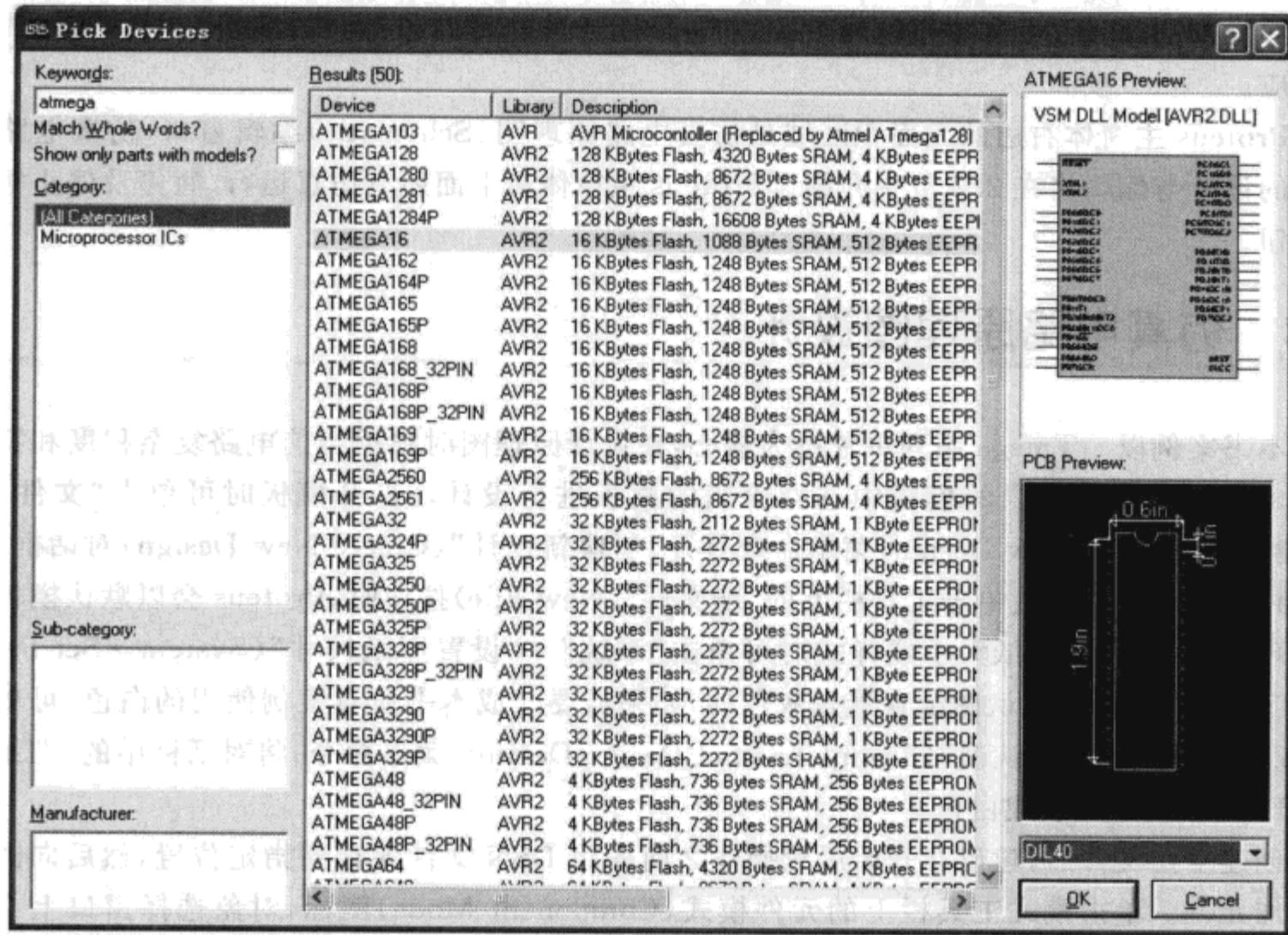


图 2-2 元件选择窗口

对于连接到总线的同样长度与形状的连线,可先绘制好其中一条,绘制其他连线时只需要双击新的起点即可。

本书大量案例电路使用了总线,对于连接到总线的双方要进行同名对等标记,如果这些标记全部用逐个添加 Label 的方法完成,将会浪费很多时间。为了实现快速标记,Proteus 提供了专门的属性赋值工具(Property Assignment Tool)。操作方法如下:

按下键盘 A 键或单击菜单“Tools/ Property Assignment Tool”打开图 2-3 所示窗口,在 String 文本框中输入“NET=D#”,Count 默认为 0,Increment 默认为 1,然后单击 OK。接下来将鼠标指向连接到总线的任意一条连线时,指针旁边将出现绿色的“=”号,依次单击这些连线时,它们会被分别标上 D0、D1、D2…。显然,D# 中的“#”号初值为 Count,在单击过程中不断递增 1。

有的案例中与总线的连线太多且连线距离较长,电路显得非常复杂,通过属性赋值工具逐一单击输入 Label 的工作量也很大,例如 5.25 节“能接收串口信息的带中英文硬字库的 80×16 点阵显示屏”案例。为简化连线并快速标记,该案例使用了大量的默认连接端子(TERMI-

NALS/DEFAULT)。假设某 8 个端子要赋值为 R0~R7, 可先选中这 8 个连接端子, 然后打开属性赋值工具窗口输入“NET=R#”, Count 与 Increment 保持为默认值, 然后单击 OK, 这 8 个端子的名称即可实现一次性快速批量标记。如果要赋值为 R8~R15, Count 应设为 8。如果一组端子标记为 C0~C7, 而显示出来的标记为 C7~C0, 这时可将 Count 设为 7, 然后将 Increment 设为 -1, 当前版本的 Proteus 不支持根据圈选方向自动设置递增方向。

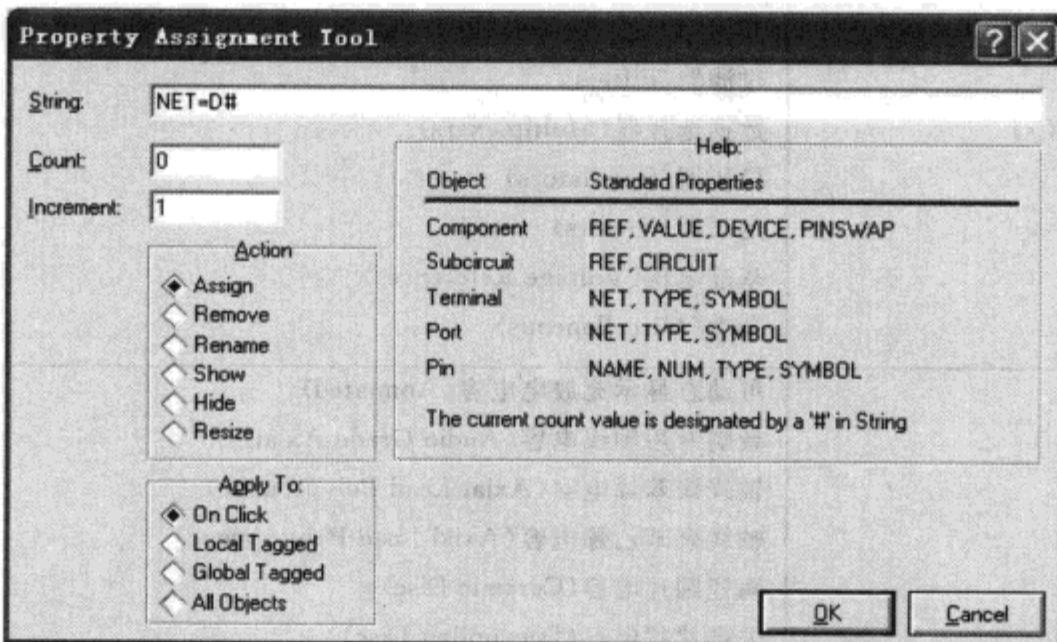


图 2-3 属性赋值窗口

在布线过程还可能会遇到这样的问题: 将一个 DSN 文件中的部分元件或子电路复制到另一文件时, 粘贴进来的部分无法与电路中已有的元件连线。这是因为两者在绘图时设置的网格分辨率不一样。遇到该问题时, 可打开“查看”(View)菜单, 在 10th 到 0.5in 之间选择不同的吸附(Snap)分辨率, 分辨率越小越便于绘制密集的线条, Proteus 的默认设置是 Snap 100th。

在设计电路原理图过程中, 可能会有元件加入 DSN 文件但电路中没有使用该元件, 或者曾经使用过, 但随后又将其删除了。如果要将这些元件从文件中彻底清除, 可执行菜单命令“编辑”→“清理文件中没有用的器件”(Edit→Tidy)。另外, 执行“工具”→“材料清单”(Tools → Materials List)命令可以很方便地生成当前案例的所有元件清单。

2.3 元件选择

设计仿真电路时要从元件库中选择所需要的元件, 在图 2-2 所示窗口中输入元件名全称或者部分名称, 元件拾取窗口会随即进行快速模糊查询。为便于选取元件, 表 2-1 给出了 Proteus 提供的所有元件分类与子类, 其中 CMOS 系列与 TTL 系列多数子类是相同的, 本表将它们列在同一行中。



表 2-1 Proteus 提供的所有元件分类及子类

元件分类	元件子类
所有分类(All Categories)	无子类
模拟芯片(Analogy ICs)	放大器(Amplifiers) 比较器(Comparators) 显示驱动器(Display Drivers) 过滤器(Filters) 数据选择器(Multiplexers) 稳压器(Regulators) 定时器(Timers) 基准电压(Voltage References) 杂类(Miscellaneous)
电容(Capacitors)	可动态显示充放电电容(Animated) 音响专用轴线电容(Audio Grade Axial) 轴线聚苯烯电容(Axial Lead Polypropene) 轴线聚苯乙烯电容(Axial Lead Polypropene) 陶瓷圆片电容(Ceramic Disc) 去耦片状电容(Decoupling Disc) 普通电容(Generic) 高温径线电容(High Temp Radial) 高温轴线电解电容(High Temperature Axial Electrolytic) 金属化聚酯膜电容(Metallised Polyester Film) 金属化聚烯电容(Metallised Ploypropene) 金属化聚烯膜电容(Metallised Ploypropene Film) 小型电解电容(Miniture Electrolytic) 多层金属化聚酯膜电容(Multilayer Metallised Polyester Film) 聚脂膜电容(Mylar Film) 镍栅电容(Nickel Barrier) 无极性电容(Non Polarised) 聚脂层电容(Polyester Layer) 径线电解电容(Radial Electrolytic) 树脂蚀刻电容(Resin Dipped) 钽珠电容(Tantalum Bead) 可变电容(Variable) VX 轴线电解电容(VX Axial Electrolytic)
连接器(Connectors)	音频接口(Audio) D 型接口(D - Type) 双排插座(DIL) 插头(Header Blocks) PCB 转接器(PCB Transfer) 带线(Ribbon Cable) 单排插座(SIL) 连线端子(Terminal Blocks) 杂类(Miscellaneous)

续表 2-1

元件分类	元件子类
所有分类(All Categories)	无子类
数据转换器(Data Converters)	模/数转换器(A/D Converters) 数/模转换器(D / A Converters) 采样保持器(Sample & Hold) 温度传感器(Temperature Sensors)
调试工具(Debugging Tools)	断点触发器(Breakpoint Triggers) 逻辑探针(Logic Probes) 逻辑激励源(Logic Stimuli)
二极管(Diodes)	整流桥(Bridge Rectifiers) 普通二极管(Generic) 整流管(Rectifiers) 肖特基二极管(Schottky) 开关管(Switching) 隧道二极管(Tunnel) 变容二极管(Varicap) 齐纳击穿二极管(Zener)
ECL 10000 系列 (ECL 10000 Series)	各种常用集成电路
机电(Electromechanical)	各类直流和步进电机
电感(Inductors)	普通电感(Generic) 贴片式电感(SMT Inductors) 变压器(Transformers)
拉普拉斯变换(Laplace Primitives)	一阶模型(1st Order) 二阶模型(2st Order) 控制器(Controllers) 非线性模式(Non-Linear) 算子(Operators) 极点/零点(Poles/Zones) 符号(Symbols)
存储芯片(Memory ICs)	动态数据存储器(Dynamic RAM) 电可擦除可编程存储器(EEPROM) 可擦除可编程存储器(EPROM) I ² C 总线存储器(I ² C Memories) SPI 总线存储器(SPI Memories) 存储卡(Memory Cards) 静态数据存储器(Static Memories)



续表 2-1

元件分类	元件子类
所有分类(All Categories)	无子类
微处理器芯片(Microprocessor ICs)	68000 系列(68000 Family) 8051 系列(8051 Family) ARM 系列(ARM Family) AVR 系列(AVR Family) Parallax 公司微处理器(BASIC Stamp Modules) HCF11 系列(HCF11 Family) PIC10 系列(PIC10 Family) PIC12 系列(PIC12 Family) PIC16 系列(PIC16 Family) PIC18 系列(PIC18 Family) Z80 系列(Z80 Family) CPU 外设(Peripherals)
杂项(Miscellaneous)	含天线、ATA/IDE 硬盘驱动模型、单节与多节电池、串行物理接口模型、晶振、动态与通用保险、模拟电压与电流符号、交通信号灯
建模源(Modelling Primitives)	模拟(仿真分析)(Analogy(SPICE)) 数字(缓冲器与门电路)(Digital(Buffers & Gates)) 数字(杂类)(Digital(Miscellaneous)) 数字(组合电路)(Digital(Combinational)) 数字(时序电路)(Digital(Sequential)) 混合模式(Mixed Mode) 可编程逻辑器件单元(PLD Elements) 实时激励源(Realtime(Actuators)) 实时指示器(Realtime(Indictors))
运算放大器(Operational Amplifiers)	单路运放(Single) 二路运放(Dual) 三路运放(Triple) 四路运放(Quad) 八路运放(Octal) 理想运放(Ideal) 大量使用的运放(Macromodel)
光电子类器件(Optolectronics)	七段数码管(7-Segment Displays) 英文字字符与数字符号液晶显示器(Alphanumeric LCDs) 条形显示器(Bargraph Displays) 点阵显示屏(Dot Matrix Displays) 图形液晶(Graphical LCDs) 灯泡(Lamp) 液晶控制器(LCD Controllers) 液晶面板显示(LCD Panels Displays) 发光二极管(LEDs) 光耦元件(Optocouplers) 串行液晶(Serial LCDs)

续表 2-1

元件分类	元件子类
所有分类(All Categories)	无子类
可编程逻辑电路与现场可编程门阵列 (PLD & FPGA)	无子分类
电阻(Resistors)	0.6 W 金属膜电阻(0.6 W Metal Film) 10 W 绕线电阻(10 W Wirewound) 2 W 金属膜电阻(2 W Metal Film) 2 W 金属膜电阻(2 W Metal Film) 3 W 金属膜电阻(3 W Metal Film) 7 W 金属膜电阻(7 W Metal Film) 通用电阻符号(Generic) 高压电阻(High Voltage) 负温度系数热敏电阻(NTC) 排阻(Resistor Packs) 滑动变阻器(Variable) 可变电阻(Varistors)
仿真源(Simulator Primitives)	触发器(Flip - Flops) 门电路(Gates) 电源(Sources)
扬声器与音响设备 (Speakers & Sounders)	无子分类
开关与继电器 (Switchers & Relays)	键盘(Keypads) 普通继电器(Generic Relays) 专用继电器(Specific Relays) 按键与拨码开关(Switches)
开关器件(Switching Devices)	双端交流开关元件(DIACs) 普通开关元件(Generic) 可控硅(SCRs) 三端可控硅(TRIACs)
热阴极电子管(Thermionic Valves)	二极真空管(Diodes) 三极真空管(Triodes) 四极真空管(Tetrodes) 五极真空管(Pentodes)
转换器(Transducers)	压力传感器(Pressure) 温度传感器(Temperature)
晶体管(Transistors)	双极性晶体管(Bipolar) 普通晶体管(Generic) 绝缘栅场效应管(IGBT/Insulated Gate Bipolar Transistors) 结型场效应晶体管(JFET) 金属-氧化物半导体场效应晶体管(MOSFET) 射频功率 LDMOS 晶体管(RF Power LDMOS) 射频功率 VDMOS 晶体管(RF Power VDMOS) 单结晶体管(Unijunction)



续表 2-1

元件分类	元件子类
所有分类(All Categories)	无子类
CMOS 4000 系列 (CMOS 4000 series)	加法器(Adders) 缓冲器/驱动器(Buffers & Drivers)
TTL 74 系列 (TTL 74 Series)	比较器(Comparators) 计数器(Counters) 解码器(Decoders) 编码器(Encoders)
TTL 74 增强型低功耗肖特基系列 (TTL 74ALS Series)	触发器/锁存器(Flip - Flop & Latches) 分频器/定时器(Frequency Dividers & Timers)
TTL 74 增强型肖特基系列 (TTL 74AS Series)	门电路/反相器(Gates & Inverters)
TTL 74 高速系列 (TTL 74F Series)	数据选择器(Multiplexers) 多谐振荡器(Multivibrators)
TTL 74HC 系列/CMOS 工作电平 (TTL 74HC Series)	振荡器(Oscillators) 锁相环(Phase - Locked - Loops, PLL)
TTL 74HCT 系列/TTL 工作电平 (TTL 74HCT Series)	寄存器(Registers) 信号开关(Signal Switches)
TTL 74 低功耗肖特基系列 (TTL 74LS Series)	收发器(Transceivers)
TTL 74 肖特基系列 (TTL 74S Series)	杂类逻辑芯片(Misc. Logic)

2.4 仿真运行

完成单片机系统仿真电路图设计后,给案例中的 AVR 单片机绑定程序文件即可开始仿真运行。本书所有案例的 C 程序保存在 AVR-C 文件夹下。为运行生成的 HEX 程序文件,可双击单片机,打开单片机属性窗口(也可以先在单片机上右击,再单击,或者选中单片机后按下 $CTRL+E$),在 Program Files 项中选择对应的 HEX 文件。

对于需要为外围芯片绑定映象文件(Image file)的案例(例如第 5 章综合设计中 5.20 节“12864LCD 显示 24C08 保存的开机画面”等),可打开相应芯片的属性窗口,在“Image file”中选择对应的 BIN 文件,BIN 文件的创建方法将在相关案例中讨论。给 AVR 单片机的 EEPROM 绑定初始文件时,可先打开单片机属性窗口,找到增强属性(Advanced Properties)下拉框中的 EEPROM 初始内容(Initial Contents of EEPROM),然后选择对应的 BIN 文件。对于编译生成的 ELF 文件,它不能像 BIN 文件那样直接绑定到单片机的 EEPROM。

在仿真电路和程序都没有问题时,单击 Proteus 主窗体下的“运行”(Play)按钮即可仿真运行 AVR 单片机系统,运行过程中可如同在硬件环境下一样与单片机交互。

观察单片机 SRAM、EEPROM 内存数据、24C0X、温度寄存器、时钟芯片等内部数据时,可在案例运行时单击“单步”(Step)或“暂停”(Pause)按钮,然后在“调试”(Debug)菜单中打开相应目标设备。

如果要观察仿真电路中某些位置的电压或波形等,可向电路中添加虚拟仪器(Virtual Instruments),例如直流电压表(DC voltmeter)、示波器(Oscilloscope)等。如果添加的虚拟仪器(例如虚拟示波器)在系统运行时没有任何显示,可在“调试”菜单中将它们打开。

2.5 Proteus 与 AVR Studio 的联合调试

对于较为复杂的程序,如果运行没有达到预期效果,这时可能需要对 Proteus 与 AVR Studio 进行联合调试。

Proteus 与 AVR Studio 的联合调试和 Proteus 与 Keil μ Vision 的联合调试不同,后者是同时运行两个软件,两者之间通过套接字进行通信,前者则只需要运行 AVR Studio 4.16 及以上版本,打开当前 AVR-GCC 项目程序,这里以“1602 液晶显示 DS1302 实时实钟.aps”为例,然后执行菜单命令“调试”→“选择平台与设备”(Debug→Select Platform and Device),在如图 2-4 所示窗口中选择调试平台为“Proteus VSM Viewer”,在设备中选择“ATmega16”,然后确定。这时 AVR Studio 主窗体中会出现 Proteus VSM 窗口,如图 2-5 左边窗口所示。

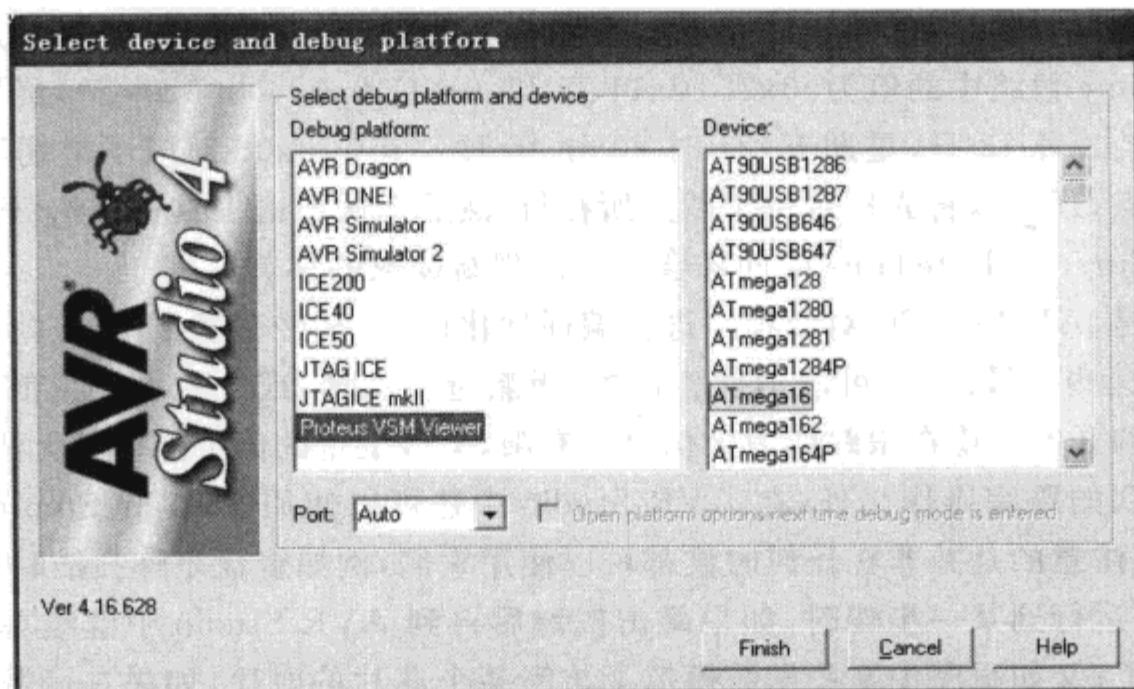


图 2-4 选择调试平台与设备

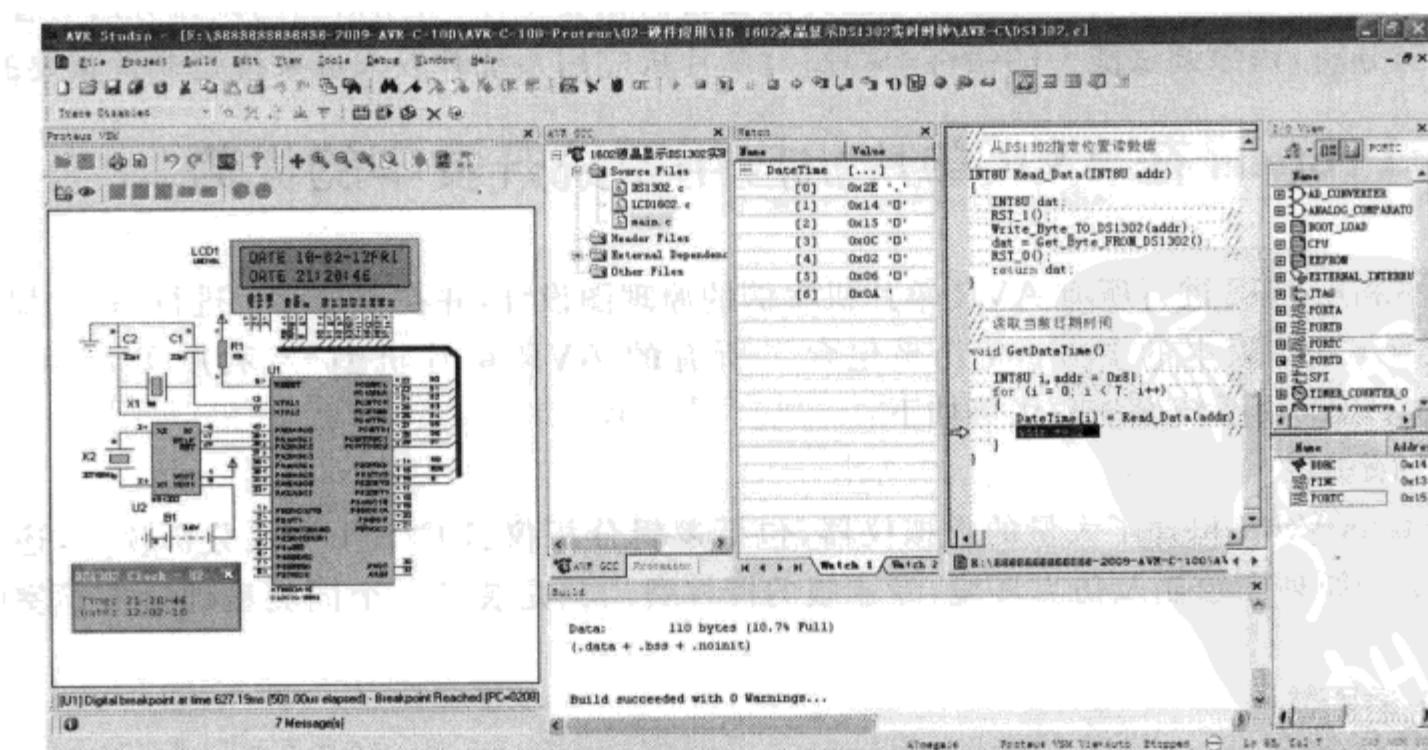


图 2-5 Proteus 与 AVR Studio 联合调试



Proteus VSM Viewer 提供了嵌入式软件开发与电路原理图之间的连接,它使 Proteus 虚拟电路模型可以作为 AVR Studio 的一个插件。该插件使得在 AVR Studio 平台进行软件开发、编译、调试以及与电路原理图进行联合仿真、实现无隙的设计流程成为可能。

单击 Proteus VSM Viewer 中的打开按钮,调入相应的 Proteus 仿真电路图文件(*.DSN),图 2-5 中调入的是“1602 液晶显示 DS1302 实时实钟.DSN”。调入仿真电路以后,单击工具栏上的“开始调试”按钮(Start Debugging),该按钮为绿色三角形。如果要全速运行系统,可接着单击“运行”按钮(Run)或按下快捷键 F5。如果需要观察某些变量的值,可设置断点,在相应变量上右击,选择“添加观察”(Add Watch)。

在图 2-5 所示窗口中,为观察依次读取的 7 个字节:秒、分、时、日、月、星期、年,可首先在 DateTime[i] = Read_Data(addr) 的下一行设置断点,设置时先将光标定位于这一行,然后右击选择“设置断点”(Toggle Breakpoint),或直接单击工具栏上设置断点的红色圆形按钮。除使用鼠标设置断点外,还可以使用快捷键 F9。设置断点后,单击“运行”按钮(Run),在 7 次到达断点后,DateTime 数组中的值为:0x2E、0x14、0x15、0x0C、0x02、0x06、0x0A,经转换后它们对应于:(20)10 年 2 月 12 日,星期五,21 时 20 分 46 秒。其中 0x06 对应于星期五(0x06 - 1)。如果不设置断点,还可以将光标定位于断点所在行,然后 7 次单击工具栏上的“运行到光标处”按钮(Run to Cursor),DateTime 中同样会逐个出现新读取的字节。

要注意的是,AVR-GCC 对源程序进行编译优化以后,某些变量在调试过程中无法观察。如果仍要观察这些变量的值,可取消优化设置,重新进行编译,或在相应变量前添加 volatile,由于当前版本的开发环境在跟踪能力上仍显得有限,某些变量的值可能仍然无法观察。

除以上提到的跟踪操作以外,在 AVR Studio 中还可以使用 Step in、Step out、Step over 等进行跟踪,要注意的是并非任何时候都可以使用它们,例如键盘矩阵扫描时就不能用单步跟踪,因为程序运行到某一步骤时,如果敲击按键后再到 AVR Studio 中继续单步跟踪,这时按键早已释放了;又如程序中某些函数模拟了访问某个芯片的时序,如果在函数内单步跟踪,这样也会失去对芯片时序的仿真模拟,跟踪也是达不到效果的。

在联合调试过程中,要注意综合考虑外部设备的相关特性,运用恰当的调试方法对程序进行跟踪与分析,程序调试能力的锻炼与提高对单片机应用系统开发人员来说是非常重要的。

2.6 Proteus 在 AVR 单片机应用系统开发中的优势

本书利用 ISIS 进行所有 AVR 单片机案例的原理图设计,并在原理图上进行 GCC 程序调试与仿真。当前版本的 Proteus 几乎包含了所有的 AVR 单片机型号,利用 Proteus 进行 AVR 单片机应用系统设计的优势如下:

(1) 廉价性

Proteus VSM 包含了大量的虚拟仪器,包括逻辑分析仪、I²C/SPI 协议分析仪等,还包括通用的电路原理图绘制及仿真环境,专业版的授权费用只是装备一个同类型硬件实验室的一小部分。

(2) 适用性

由于所有的工作在软件环境中完成,对原理图的重新布线、对硬件的修改及重新测试,这些都只需要很少的时间。如果要优化设计或对软硬件进行试验,这都可以很快地完成。并且

在这样的透明环境中,设计者所作的修改效果可以立即观察到,对硬件的修改如同对软件的修改、验证一样简单和快捷。

(3) 独特征

Proteus VSM 包括大量不能够或不容易在硬件环境中实现的特征,例如:诊断消息(Diagnostic messaging)功能允许访问系统器件,获取所有与组件、外部电路及系统其他部分交互的动态报告文本。

Proteus 仿真引擎可监视整个仿真过程,能够自动给出硬件和软件的错误警告,包括系统器件之间的时序与逻辑冲突、写非法内存地址或破坏硬件堆栈。

Proteus 与系统硬件的交互及对系统测试非常容易且效果明显,例如要测试系统中的温度传感器代码,可简单地调整外围温度并检查硬件程序响应,并将所获取的结果与等效的外围硬件原型环境温度进行比较。

(4) 高效性

利用 Proteus 开发的 AVR 单片机应用系统将非常易于测试、分析与调试,易于修改与校正,从而快速改进系统设计,实现高效开发。

第3章

基础程序设计

通过对前 2 章的学习,大家进一步熟悉了 AVR 单片机的基本硬件结构与内部资源,归纳了用 C 语言开发单片机程序必须重点掌握的技术内容,同时还熟悉了 Proteus 的基本操作、Proteus 与 AVR Studio 的联合调试技术等。这为本章及后续章节中 Proteus 环境下 AVR 单片机的 C 语言程序设计案例的学习调试与研究打下了基础。

本章案例涉及 AVR 单片机内部资源的程序设计以及基本外围元件的应用,案例包括 3 个部分:

第一部分包括 3.1~3.13 号案例,涉及基本 I/O 控制,内容包括 LED、数码管、按键、开关与继电器、蜂鸣器等程序设计;

第二部分包括 3.14~3.29 号案例,主要涉及外部中断与定时/计数器程序设计;

第三部分包括 3.30~3.35 号案例,内容涉及模/数转换、模拟比较器、EEPROM 与 Flash 数据访问、看门狗应用及串口控制等。

通过对这些案例的学习研究与跟踪调试,通过对各案例中提出实训目标的独立实践,可以全面掌握 AVR 单片机的 C 语言基础程序设计技术,熟练使用 C 语言控制和运用单片机内部资源,为 AVR 单片机扩展资源的应用以及 AVR 单片机系统的综合设计打下良好的基础。

3.1 闪烁的 LED

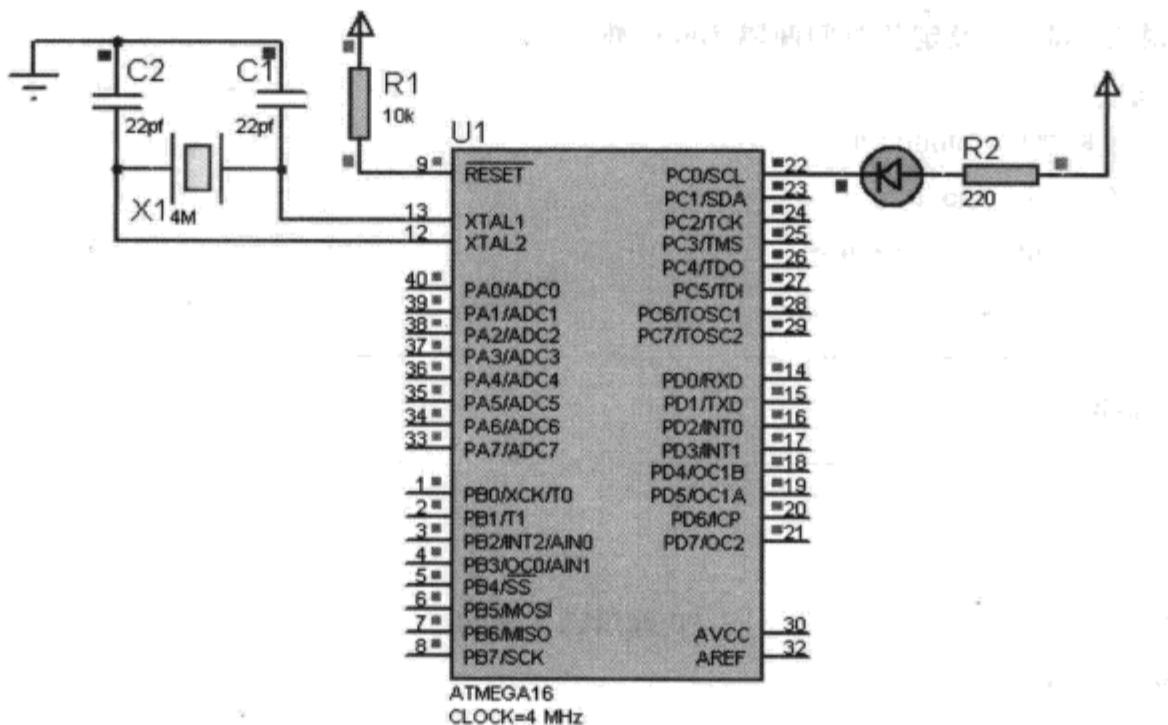
本例单片机 PC0 引脚连接 LED,程序按设定的时间间隔在 PC0 引脚不断输出 010101...,使 LED 按固定时间间隔持续闪烁。通过这个案例可以掌握最基本的单片机 C 程序设计与仿真调试方法,以及在后续案例中大量出现的延时函数的用法。案例电路如图 3-1 所示。注意仿真电路中限流电阻 R2 的阻值,未设置或设置过大都可能导致 LED 无法闪烁。

1. 程序设计与调试

本例及后续所有程序都必须添加头文件<avr/io.h>,缺少该头文件时将会显示如下编译提示:

```
Build started 10.11.2009 at 09:56:50
avr-gcc.exe -mmcu=atmega16-Wall-gdwarf-2
-DF_CPU=4000000UL -O1-fsigned-char-MD-MP-MT LED.o-MF dep/LED.o.d -c ../LED.c
../LED.c: In function 'main':
../LED.c:15: error: 'DDRC' undeclared (first use in this function)
../LED.c:15: error: (Each undeclared identifier is reported only once
../LED.c:15: error: for each function it appears in.)
```

```
..../LED.c:19: error: 'PORTC' undeclared (first use in this function)
..../LED.c:19: warning: implicit declaration of function '_BV'
..../LED.c:19: error: 'PC0' undeclared (first use in this function)
make: * * * [LED.o] Error 1
```



注：按国标 应为 ，下同。

图 3-1 闪烁的 LED

编译显示 DDRC、PORTC、PC0、_BV 均未描述，编译出错。

在 8051 版的案例中，各程序都单独编写了延时函数，由于 AVR-GCC 提供了便于使用的延时函数，本例及后续程序中的延时都可用 _delay_ms 和 _delay_us 实现。

查看 _delay_ms 和 _delay_us 延时函数的细节时，可单击 AVRStudio 的菜单 Help/avr-libc Reference Manual(即帮助菜单下的 avr 库函数参考手册)，在打开的帮助首页中单击链接 Library Reference，然后在所列出的所有模块(modules)中单击 <util/delay.h>：Convenience functions for busy-wait delay loops 即可。

需要注意的是：对于不同的版本的 WinAVR，相关头文件的存放位置可能会改变；在使用不同版本的 WinAVR 时需要自行修改相关头文件的路径。

本例电路中，为使单只 LED 持续闪烁，程序首先通过 DDRC 将 PC 端口设为输出（本例将该端口各引脚全部设成了输出），然后对 PORTC 与 _BV(PC0) 不断进行“异或”操作，其中 _BV(PC0) 即 00000001(PC0 对应的最低位为 1，其他位为 0)，当用 00000001 与 PORTC 执行异或操作时，由于 0 与任何数位异或时都会使对方的值保持不变，1 与任何数位异或时都会使对方取反，这使得 PORTC 的前 7 位始终保持不变，而最低位将不断被取反，从而在该引脚输出 010101…。

另外，本书所有主函数返回值类型均设为 int，主函数声明为 int main()，但主函数末尾的 return 0 或 return 1 等可以省略。

2. 实训要求

- ① 修改延时参数，改变 LED 的闪烁频率。
- ② 修改仿真电路与 C 程序，实现对不同端口多个 LED 的闪烁控制。



3. 源程序代码

```
01 //-----  
02 // 名称：闪烁的 LED  
03 //-----  
04 // 说明：LED 按设定的时间间隔不断闪烁  
05 //-----  
06 #define F_CPU 4000000UL  
07 #include <avr/io.h>  
08 #include <util/delay.h>  
09  
10 //-----  
11 // 主程序  
12 //-----  
13 int main()  
14 {  
15     DDRC = 0xFF;           //PC 端口设为输出  
16     while (1)  
17     {  
18         PORTC ^= _BV(PC0); //通过异或操作对 PC0 引脚反复取反，输出…01010101  
19         _delay_ms(120);    //延时  
20     }  
21 }
```

3.2 左右来回的流水灯

本例连接 PA 端口的 8 只 LED 左右来回循环滚动点亮，形成走马灯效果。案例电路及部分运行效果如图 3-2 所示。本例重点在于左右移位控制。

1. 程序设计与调试

案例中的 8 只 LED 连接在 PA 端口，根据本例 LED 的连接方向，PA 端口对应位输出 1 时 LED 点亮，反之则熄灭。

对于程序中定义的移动位数变量 b：当 0x01 左移 b 位（即 $00000001 \ll b$ ）时，对应的 LED 位被点亮；当 0x80 右移 b 位（即 $10000000 \gg b$ ）时，对应的 LED 位亦被点亮。无论左移还是右移，当 b 由 0 递增到 7 时即完成一趟单向移动，在 b 等于 8 时即可改变方向。本例中方向由变量 direction 控制，通过修改移动方向并配合应用左移（ \ll ）与右移（ \gg ）操作符即实现了 LED 左右回来的滚动显示效果。

AVR-GCC 的头文件<stdint.h>中有如下定义：

typedef signed char	int8_t
typedef unsigned char	uint8_t
typedef signed int	int16_t
typedef unsigned int	uint16_t

```

typedef signed long int          int32_t
typedef unsigned long int        uint32_t
typedef signed long long int     int64_t
typedef unsigned long long int   uint64_t

```

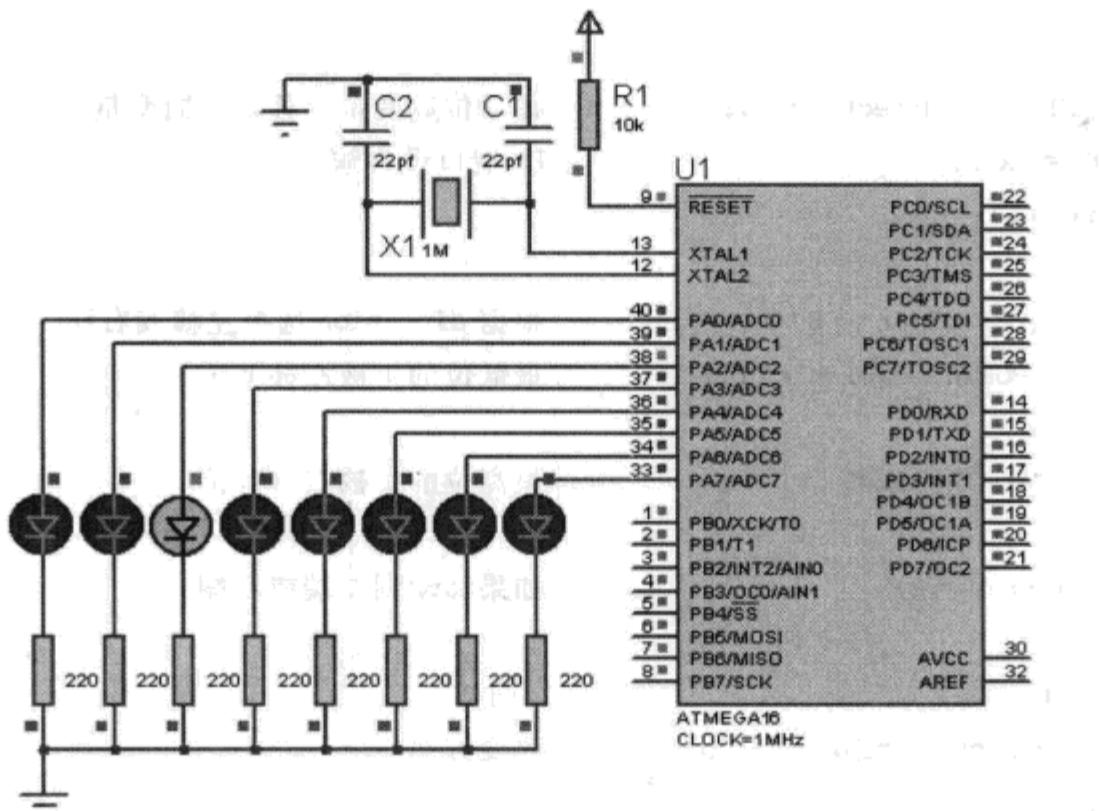


图 3-2 左右来回的流水灯

如果添加头文件<stdint.h>, 则本例中变量 b 和 direction 可定义为 int8_t 类型, 即“int8_t b, direction;”。

本书案例没有引用该头文件, 所有代码中的无符号字符类型(即字节类型 BYTE)与无符号整数类型(即字类型 WORD)分别重新定义为 INT8U 与 INT16U。

2. 实训要求

① 程序中将两行 b=0 的语句均改为 b=1 亦可实现左右来回滚动效果, 但效果稍有差异, 比较差异并思考原因。

② 修改电路并重新编写程序, 使 8 只 LED 分成左右两组(每组 4 个), 分别实现左右来回滚动显示。

3. 源程序代码

```

01 //-----
02 // 名称: 左右来回的流水灯
03 //-----
04 // 说明: LED 按设定的时间左右来回滚动显示
05 //
06 //-----
07 #include <avr/io.h>
08 #include <util/delay.h>
09 #define INT8U unsigned char

```



```
10 #define INT16U unsigned int
11 //-----
12 // 主程序
13 //-----
14 int main()
15 {
16     INT8U b = 0, direction = 0;          //移动位数变量及移动方向变量
17     DDRA = 0xFF;                      //PA 端口设为输出
18     while (1)
19     {
20         if (direction == 0)            //根据 direction 选择左移或右移
21             PORTA = 0x01 << b;        //最低位的 1 被左移 b 位
22         else
23             PORTA = 0x80 >> b;        //最高位的 1 被右移 b 位
24
25         if( ++b == 8)                //如果移动到左端或右端
26         {
27             b = 0;                  //b 归 0
28             direction = !direction; //改变方向
29         }
30         _delay_ms(60);
31     }
32 }
```

3.3 花样流水灯

在 3.2 节的案例中,LED 只能按某种单调的规律显示,无法实现复杂多变的显示花样。本例中两组 LED 连接在 PC 和 PD 端口,它们按自定义的预设花样变换显示。本例电路如图 3-3 所示。

1. 程序设计与调试

本例将设计的花样预设在两个数组中,它们分别与两组 LED 对应,各数组中的每个字节对应一种显示组合,程序循环读取数组中的显示组合并送往端口,实现自定义花样的自由显示。

2. 实训要求

① 将本例 2 个 INT8U 类型的花样数组合并成 1 个 INT16U 类型的数组,请改写程序,要求仍实现本例运行效果。

② 重新设计规划花样数组内容,改变数组大小,实现自定义的花样显示。

③ 在学习调试本章 Flash 内存数据访问案例后,重新将花样数组存放于 Flash 存储器,编程实现同样的显示效果。

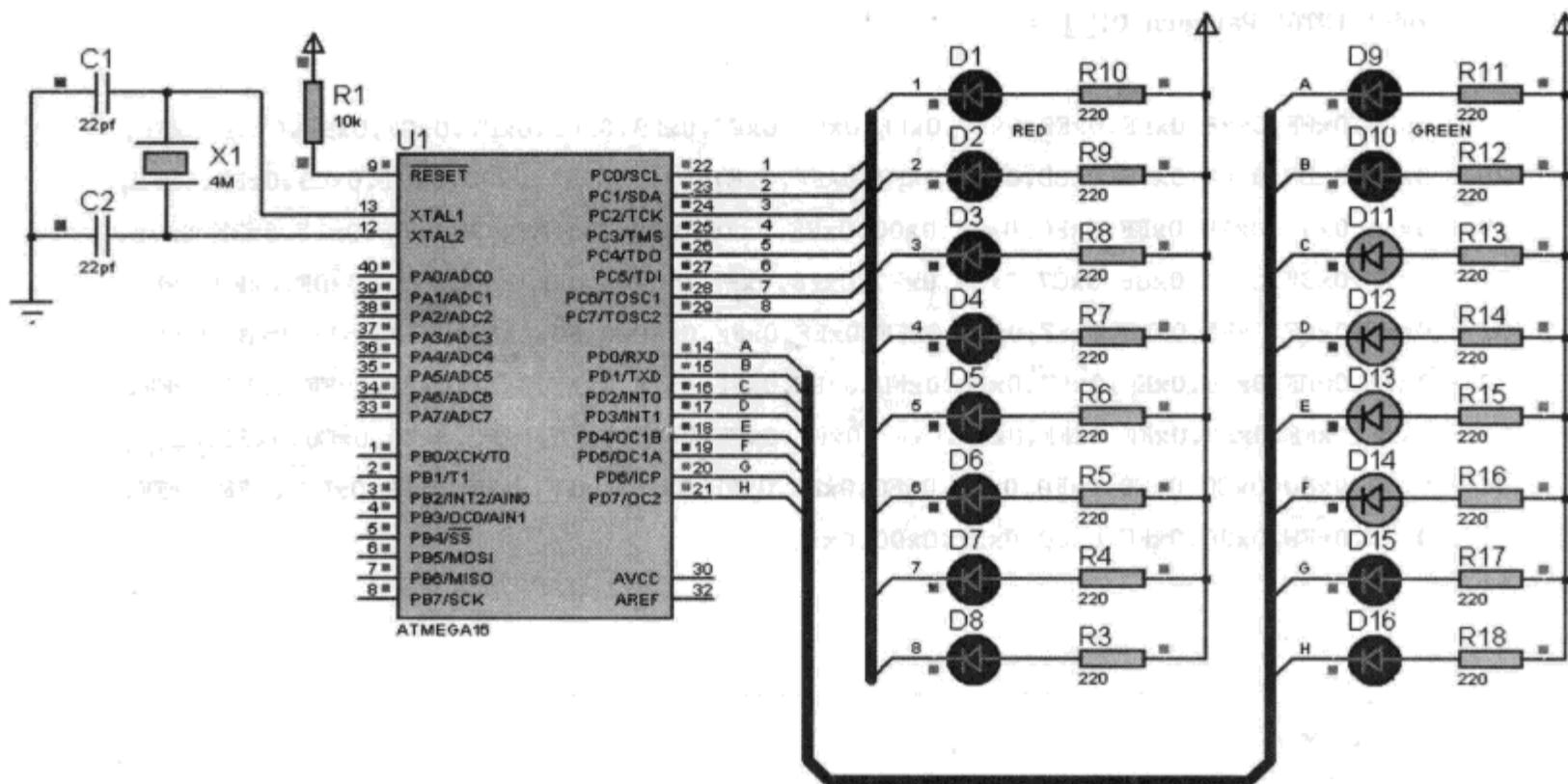


图 3-3 花样流水灯

3. 源程序代码

```

01 //-----
02 // 名称：花样流水灯
03 //-----
04 // 说明：两组 LED 按预设的花样数组变化显示
05 //-----
06 #define F_CPU 4000000UL
07 #include <avr/io.h>
08 #include <util/delay.h>
09 #define INT8U unsigned char
10
11 //两组花样定义
12 const INT8U Pattern_P0[] =
13 {
14     0xFC,0xF9,0xF3,0xE7,0xCF,0x9F,0x3F,0x7F,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
15     0xE7,0xDB,0xBD,0x7E,0xBD,0xDB,0xE7,0xFF,0xE7,0xC3,0x81,0x00,0x81,0xC3,0xE7,0xFF,
16     0xAA,0x55,0x18,0xFF,0xF0,0x0F,0x00,0xFF,0xF8,0xF1,0xE3,0xC7,0x8F,0x1F,0x3F,0x7F,
17     0x7F,0x3F,0x1F,0x8F,0xC7,0xE3,0xF1,0xF8,0xFF,0x00,0x00,0xFF,0xFF,0x0F,0xF0,0xFF,
18     0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x0F,0x00,
19     0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x7F,0xBF,0xDF,0xEF,0xF7,0xFB,0xFD,0x0F,0x00,
20     0xFE,0xFC,0xF8,0xF0,0xE0,0xC0,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
21     0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
22     0x00,0xFF,0x00,0xFF,0x00,0xFF,0x00,0xFF
23 };
24

```



```
25 const INT8U Pattern_P1[] =
26 {
27     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE, 0xFC, 0xF9, 0xF3, 0xE7, 0xCF, 0x9F, 0x3F, 0xFF,
28     0xE7, 0xDB, 0xBD, 0x7E, 0xBD, 0xDB, 0xE7, 0xFF, 0xE7, 0xC3, 0x81, 0x00, 0x81, 0xC3, 0xE7, 0xFF,
29     0xAA, 0x55, 0x18, 0xFF, 0xF0, 0x0F, 0x00, 0xFF, 0xF8, 0xF1, 0xE3, 0xC7, 0x8F, 0x1F, 0x3F, 0x7F,
30     0x7F, 0x3F, 0x1F, 0x8F, 0xC7, 0xE3, 0xF1, 0xF8, 0xFF, 0x00, 0x00, 0xFF, 0xFF, 0x0F, 0xF0, 0xFF,
31     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F,
32     0x7F, 0xBF, 0xDF, 0xEF, 0xF7, 0xFB, 0xFD, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
33     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE, 0xFC, 0xF8, 0xF0, 0xE0, 0xC0, 0x80, 0x00,
34     0x00, 0x80, 0xC0, 0xE0, 0xF0, 0xF8, 0xFC, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
35     0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF, 0x00, 0xFF
36 };
37
38 //-----
39 // 主程序
40 //-----
41 int main()
42 {
43     INT8U i;
44     DDRC = 0xFF;    PORTC = 0xFF;          //配置端口
45     DDRD = 0xFF;    PORTD = 0xFF;
46     while (1)
47     {
48         for(i = 0; i<136; i++)        //循环显示全部花样字节
49         {
50             PORTC = Pattern_P0[i];    //第一组发送给 PORTC 端口
51             PORTD = Pattern_P1[i];    //第二组发送给 PORTD 端口
52             _delay_ms(80);
53         }
54     }
55 }
```

3.4 LED 模拟交通灯

本例中的 12 只 LED 分成东西向和南北向两组,各组指示灯均有相向的 2 只红色、2 只黄色与 2 只绿色的 LED,程序运行时模拟了十字形路口交通信号灯的红绿灯切换显示及黄灯闪烁显示效果。本例电路如图 3-4 所示。

1. 程序设计与调试

本例源程序最前面用“与”(&)、“或”(|)、“异或”(^)操作符对各路指示灯的开、关、闪烁分别进行操作定义,这样可提高主程序中各指示灯操作代码的可读性。阅读调试本例时重点研究操作的切换方法及延时控制。

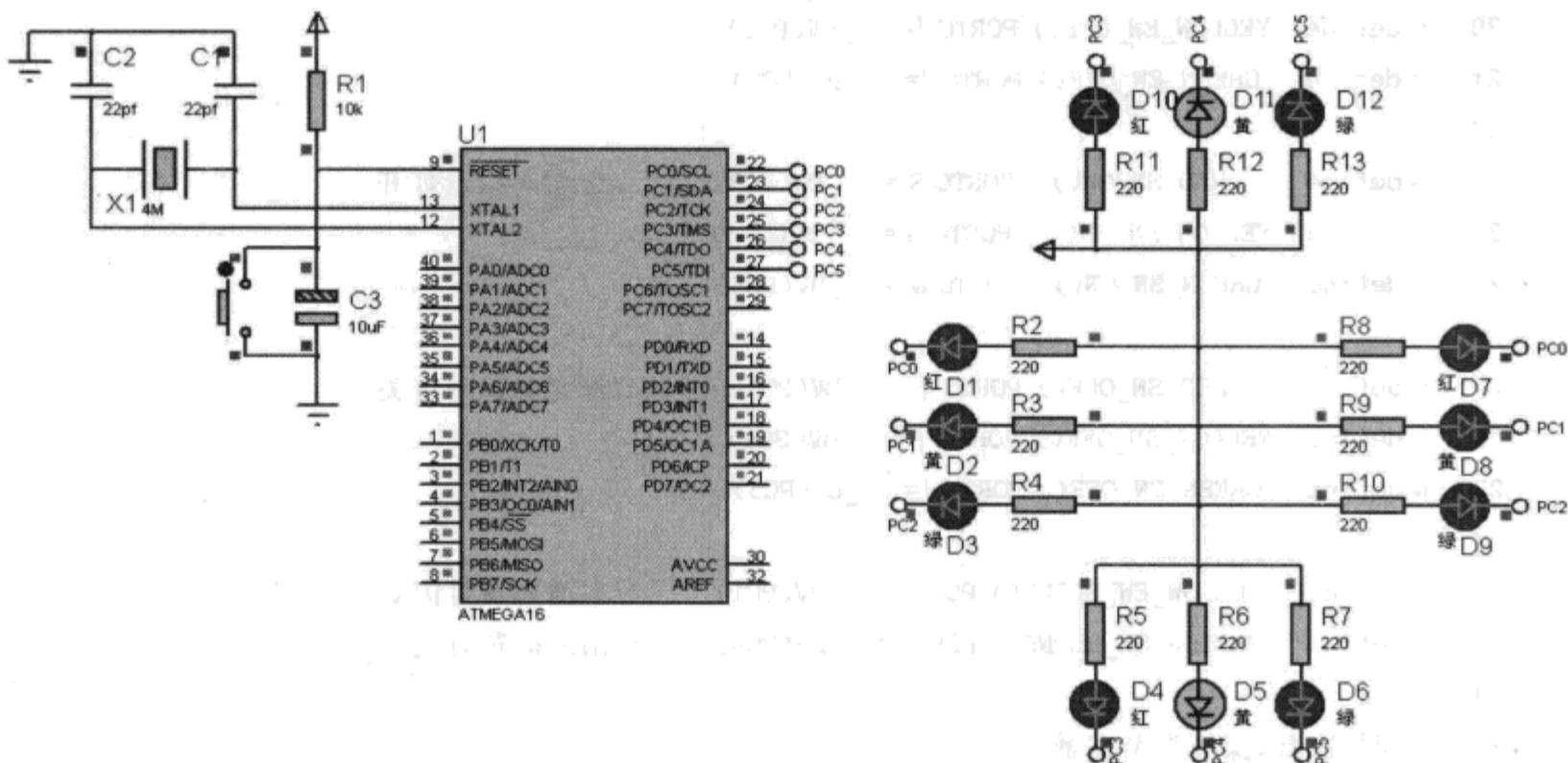


图 3-4 LED 模拟交通灯

2. 实训要求

① 本例将交通指示灯切换时间设置得较短,这样可在调试的时候快速观察到运行效果。在调试运行本例后,重新修改本例程序,模拟实际应用中的交通指示灯切换效果。

② 观察不同路口的指示灯切换效果,重新设计电路与程序进行仿真。

3. 源程序代码

```

01 //-----
02 // 名称: LED 模拟交通灯
03 //-----
04 // 说明: 东西向绿灯亮若干秒后,黄灯闪烁,闪烁 5 次后亮红灯
05 // 红灯亮后,南北向由红灯变为绿灯,若干秒后南北向黄灯闪烁
06 // 闪烁 5 次后亮红灯,东西向绿灯亮,如此往复
07 // 本例将切换时间设得较短,以便快速观察运行效果
08 //
09 //-----
10 #include <avr/io.h>
11 #include <util/delay.h>
12 #define INT8U unsigned char
13 #define INT16U unsigned int
14
15 #define RED_EW_ON() PORTC &= ~_BV(PC0) //东西向指示灯开
16 #define YELLOW_EW_ON() PORTC &= ~_BV(PC1)
17 #define GREEN_EW_ON() PORTC &= ~_BV(PC2)
18
19 #define RED_EW_OFF() PORTC |= _BV(PC0) //东西向指示灯关

```



```
20 # define YELLOW_EW_OFF() PORTC |= ~_BV(PC1)
21 # define GREEN_EW_OFF() PORTC |= ~_BV(PC2)
22
23 # define RED_SN_ON() PORTC &= ~_BV(PC3) //南北向指示灯开
24 # define YELLOW_SN_ON() PORTC &= ~_BV(PC4)
25 # define GREEN_SN_ON() PORTC &= ~_BV(PC5)
26
27 # define RED_SN_OFF() PORTC |= _BV(PC3) //南北向指示灯关
28 # define YELLOW_SN_OFF() PORTC |= _BV(PC4)
29 # define GREEN_SN_OFF() PORTC |= _BV(PC5)
30
31 # define YELLOW_EW_BLINK() PORTC ^= _BV(PC1) //东西向黄灯闪烁
32 # define YELLOW_SN_BLINK() PORTC ^= _BV(PC4) //南北向黄灯闪烁
33
34 //闪烁次数,操作类型变量
35 INT8U Flash_Count = 0, Operation_Type = 1;
36 //-----
37 // 交通灯切换子程序
38 //-----
39 void Traffic_Light()
40 {
41     switch (Operation_Type)
42     {
43         case 1: //东西向绿灯与南北向红灯亮
44             RED_EW_OFF(); YELLOW_EW_OFF(); GREEN_EW_ON();
45             RED_SN_ON(); YELLOW_SN_OFF(); GREEN_SN_OFF();
46             _delay_ms(300); //延时
47             Operation_Type = 2; //下一操作
48             break;
49
50         case 2: //东西向黄灯开始闪烁,绿灯关闭
51             _delay_ms(300);
52             YELLOW_EW_BLINK();
53             GREEN_EW_OFF();
54             //闪烁 5 次
55             if ( ++ Flash_Count != 10) return;
56             Flash_Count = 0;
57             Operation_Type = 3; //下一操作
58             break;
59
60         case 3: //东西向红灯与南北向绿灯亮
61             RED_EW_ON(); YELLOW_EW_OFF(); GREEN_EW_OFF();
62             RED_SN_OFF(); YELLOW_SN_OFF(); GREEN_SN_ON();
```

```

63 //南北向绿灯亮若干秒后切换
64 _delay_ms(300);
65 Operation_Type = 4;           //下一种操作类型
66 break;
67
68 case 4: //南北向黄灯开始闪烁
69     _delay_ms(300);
70     YELLOW_SN_BLINK();
71     GREEN_SN_OFF();
72     //闪烁 5 次
73     if (++Flash_Count != 10) return;
74     Flash_Count = 0;
75     Operation_Type = 1;       //回到第一种操作
76 }
77 }
78
79 //-----
80 // 主程序
81 //-----
82 int main()
83 {
84     DDRC = 0xFF; PORTC = 0xFF;
85     while(1) Traffic_Light();
86 }

```

3.5 单只数码管循环显示 0~9

本例运行时,电路中的单只共阴数码管循环显示数字 0、1、2、…、7、8、9。学习调试本例时,要首先掌握共阴/共阳数码管的段码设计。案例电路及部分运行效果如图 3-5 所示。

1. 程序设计与调试

本例的单只共阴数码管连接在 PC 端口,当 PORTC 某位设为 1 时,对应数码管段将被点亮。程序中预设了数字 0~9 的共阴数码管段码,0~9 的段码按固定时间间隔由 PORTC 循环输出,形成数字循环显示效果。

本例及后续大量案例中均使用数码管显示数据,数码管段码是相对固定的。本例提供的数码管段码表 SEG_CODE 将在后续案例中继续使用。

2. 实训要求

- ① 仍使用本例提供的共阴段码表,在单只共阳数码管上滚动显示数字 0~9。
- ② 将本例段码表改为共阳数码管段码表,改写本例程序,仍实现相同功能。
- ③ 在 PD 端口再连接 1 只数码管,通过 2 只独立数码管组合实现 00~99 的循环显示。

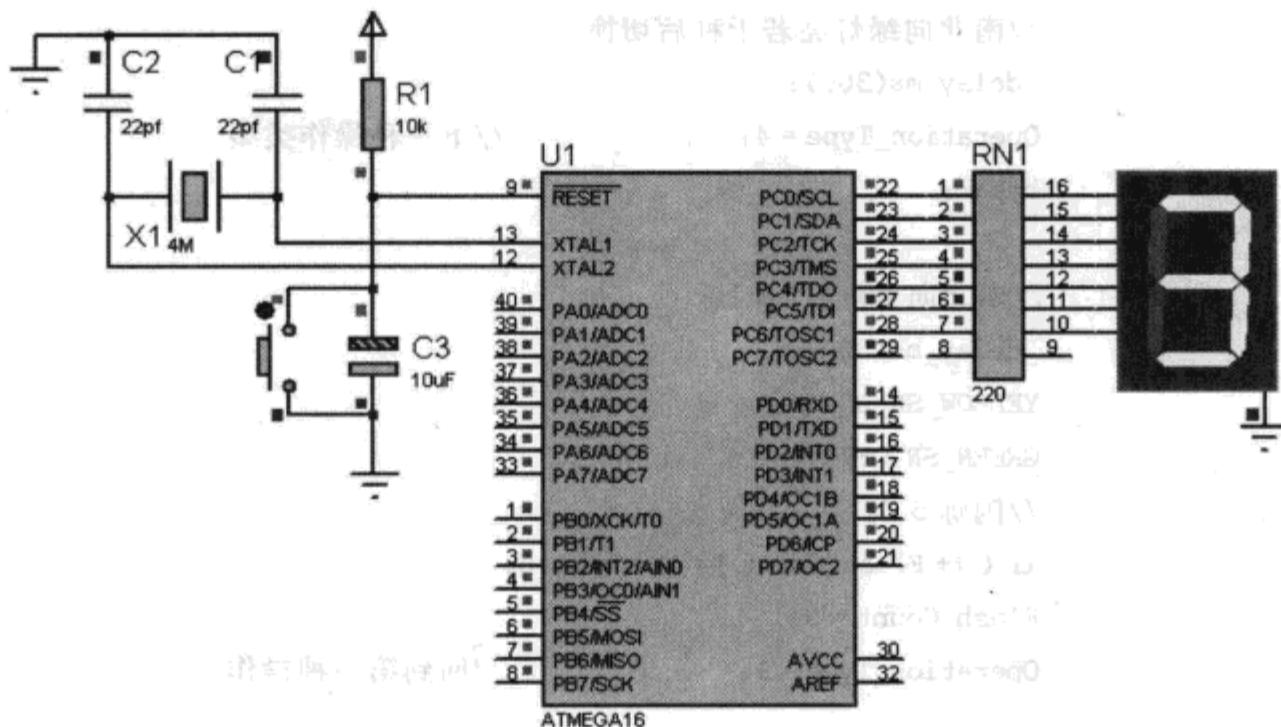


图 3-5 单只数码管循环显示 0~9

3. 源程序代码

```

01 //-----
02 // 名称：单只数码管循环显示 0~9
03 //-----
04 // 说明：主程序中的循环语句反复将 0~9 的段码送 PC 口，形成数字 0~9 的
05 //       循环显示
06 //
07 //-----
08 #include <avr/io.h>
09 #include <util/delay.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int
12
13 //0~9 的共阴数码管段码
14 const INT8U SEG_CODE[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
15 //-----
16 // 主程序
17 //-----
18 int main()
19 {
20     INT8U i = 0;
21     DDRC = 0xFF;           //PC 端口设为输出
22     while (1)
23     {
24         PORTC = SEG_CODE[i];    //发送数字段码
25         i = (i + 1) % 10;      //数字在 0~9 以内循环
26         _delay_ms(200);

```

27 }
28 }

3.6 8只数码管滚动显示单个数字

本例运行时,单个数字0~7显示在8只集成式数码管的相应位置上。通过本例调试研究后,要熟悉集成式数码管的内部构造与工作原理,为下一案例设计打下基础。本例电路及部分运行效果如图3-6所示。

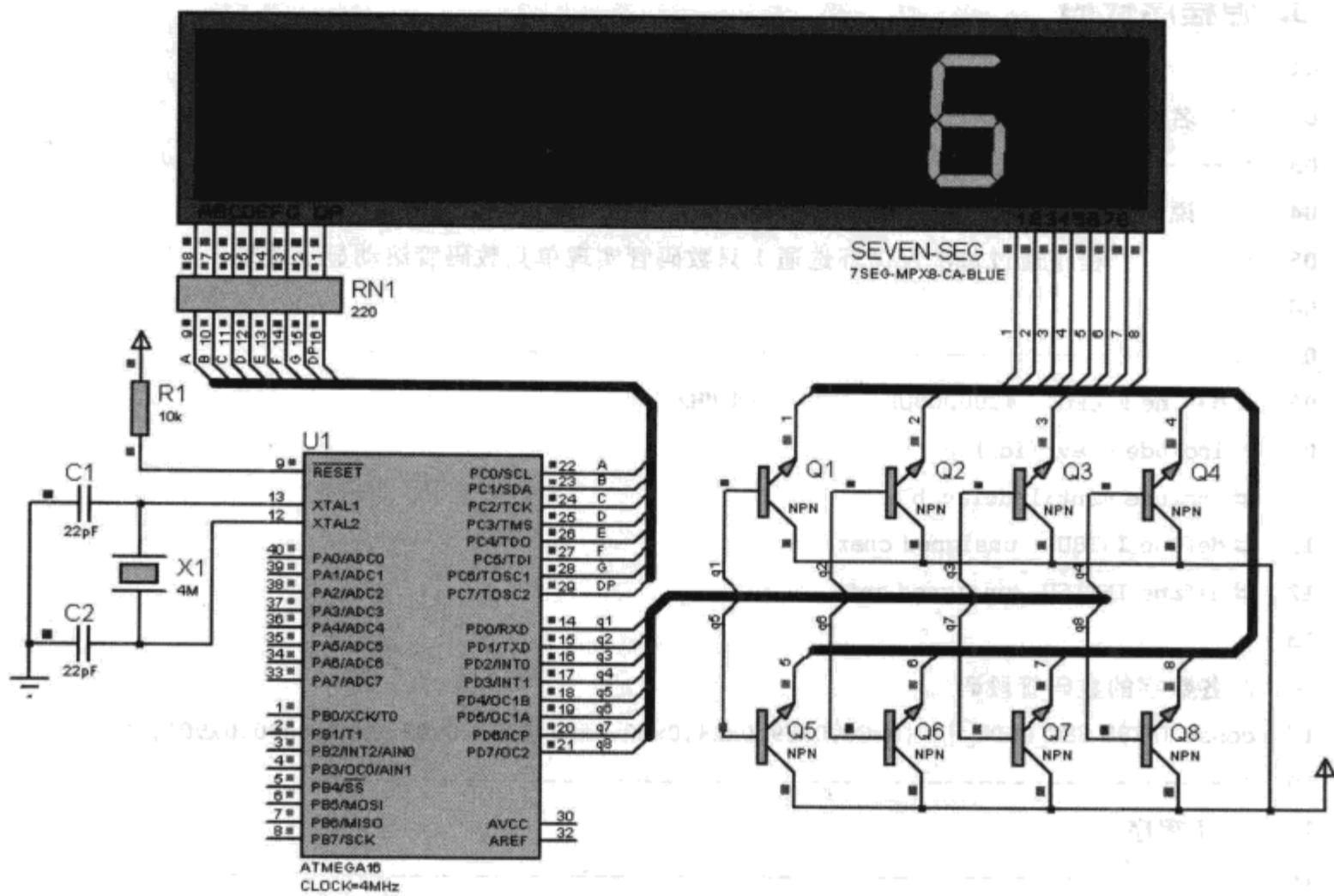


图3-6 8只数码管滚动显示单个数字

1. 程序设计与调试

本例使用了8位集成式七段蓝色共阳数码管(SEG-MPX8-CA-BLUE),CA表示共阳(Common Anode),若为CC则为共阴(Common Cathode),所有数码管a引脚并联在一起,b、c、d、e、f、g、dp亦分别并联。任何时候发送的段码都会传送给所有数码管的各段,集成式数码管各位共阳极1~8是独立的,共阳极分别与8只NPN三极管射极相连,本例程序运行时,任一时刻仅允许一只数码管的位引脚(共阳极)连接+5V。当PD端口输出段码时,相应数字就只会显示在某一只数码管上,在依次循环选中8只数码管之一时,即可形成滚动显示效果。

例如要在最左边数码管上显示数字,对于本例中的共阳数码管,其位引脚1要设为1(+5V),由于使用的是NPN三极管,在PD0为1,即PD端口输出00000001时,第一只三极管导通,对应数码管共阳极连接+5V。又如,要在第3只数码管上显示数字,PD端口必需输出00000100。



本例 for 循环中通过 PORTD=_BV(i)发送位码,当 i 取值 0~7 时,PORTD 分别输出位码 00000001、00000010、00000100、……、100000000;在输出数字 i 时,第 i 只共阳数码管被选通,数字 i 即显示在第 i 只数码管上。

2. 实训要求

- ① 重新修改代码,使单个数字从右向左滚动显示。
- ② 改用集成式共阴数码管实现同样的显示效果。
- ③ 尝试改用 7407 驱动数码管,仍实现本例显示效果。

3. 源程序代码

```
01 //-----  
02 // 名称: 8 只数码管滚动显示单个字符  
03 //-----  
04 // 说明: 数码管从左到右依次滚动显示 0~7  
05 // 程序通过每次仅循环选通 1 只数码管实现单只数码管滚动显示效果  
06 //-----  
07 //-----  
08 #define F_CPU 4000000UL //4 MHz  
09 #include <avr/io.h>  
10 #include <util/delay.h>  
11 #define INT8U unsigned char  
12 #define INT16U unsigned int  
13  
14 //各数字的数码管段码  
15 const INT8U SEG_CODE[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};  
16 //-----  
17 // 主程序  
18 //-----  
19 int main()  
20 {  
21     INT8U i;  
22     DDRC = 0xFF;    DDRD = 0xFF;      //PC、PD 端口均设为输出  
23     while (1)  
24     {  
25         for(i = 0; i<8; i++)  
26         {  
27             PORTD = _BV(i);      //发送数码管位码  
28             PORTC = SEG_CODE[i]; //发送数字 i 的段码  
29             _delay_ms(240);  
30         }  
31     }  
32 }
```

3.7 8 只数码管扫描显示多个不同字符

不同于上一案例的是本例在集成式数码管上同时显示了多个不同字符。有了 3.6 节案例的基础,掌握本例集成式数码管的扫描显示方法就很容易了。本例电路及运行效果如图 3-7 所示。

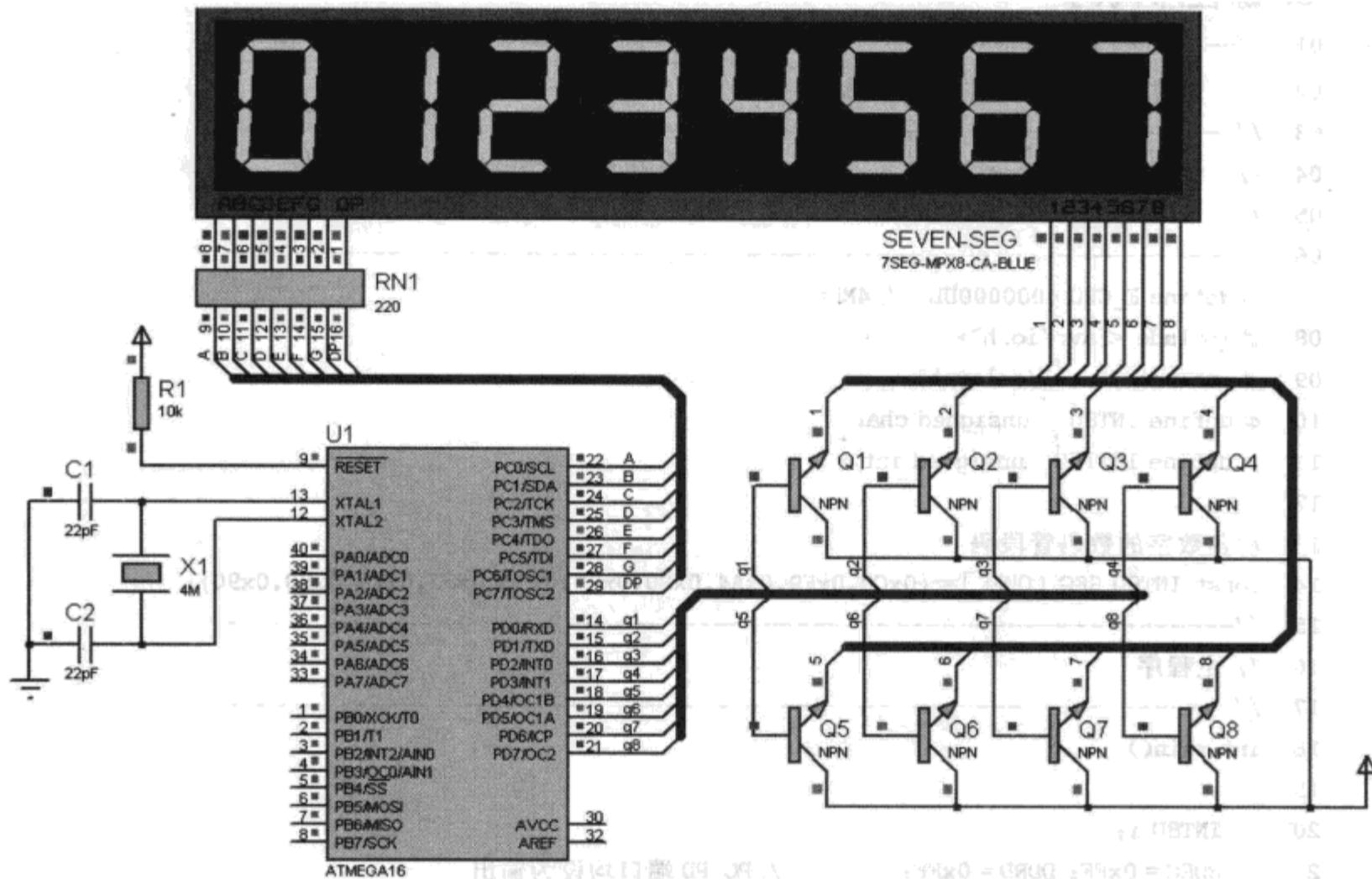


图 3-7 8 只数码管显示多个不同字符

1. 程序设计与调试

前面已经讨论过,对于集成式数码管,任何时候发送的段码都会被所有数码管收到。如果本例中所有共阳数码管的位码均为 1(0xFF),则所有数码管都会显示同一字符。

为使不同数码管显示不同字符,本例使用的是集成式多位数码管显示常用的动态扫描显示技术,它利用了人的视觉暂留特征。在选通第 1 只数码管时,发送第 1 个数字的段码,选通第 2 只数码管时发送第 2 个数字的段码,选通第 3 只数码管时发送第 3 个数字的段码,以此类推。每次仅选通 1 只数码管发送对应的段码,在切换选通下一数码管并发送相应段码的时间间隔非常短,视觉惰性使人感觉不到字符是一个接一个显示在不同数码管上的,你会觉得到所有字符是很稳定地同时显示在不同数码管上的。

可见这种设计方法和上一案例类似的是仍在数码管不同位置逐个显示不同字符,只是切换速度大大增加了。在控制切换延时的时候,要注意全屏的扫描频率要高于视觉暂留频率 16~20 Hz。电影胶片正是采取了每秒 24 张的播放速度,观众才会觉察不到人物或景色是一帧一帧地显示出来,而是觉得画面非常连贯,没有任何抖动或闪烁感。



2. 实训要求

① 将代码中的最后一行语句 _delay_ms(4) 的参数修改为 10、20 或 100 并编译运行, 观察会出现什么样的效果。

② 本例显示的是有规律的数字 0~7, 在调试运行本例后重新设计程序, 实现任意数字串的显示, 例如“23—57—39”, 其中“—”的段码可在原码表的后面添加。

3. 源程序代码

```
01 //-----  
02 // 名称: 8 只数码管扫描显示多个字符  
03 //-----  
04 // 说明: 本例运行时, 数字 0~7 同时显示在 8 位集成式数码管上.  
05 //-----  
06 //-----  
07 #define F_CPU 4000000UL //4MHz  
08 #include <avr/io.h>  
09 #include <util/delay.h>  
10 #define INT8U unsigned char  
11 #define INT16U unsigned int  
12  
13 //各数字的数码管段码  
14 const INT8U SEG_CODE[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};  
15 //-----  
16 // 主程序  
17 //-----  
18 int main()  
19 {  
20     INT8U i;  
21     DDRC = 0xFF; DDRD = 0xFF;           //PC、PD 端口均设为输出  
22     while (1)  
23     {  
24         for(i = 0; i<8; i++)  
25         {  
26             PORTC = 0xFF;                  //先暂时关闭段码  
27             PORTD = _BV(i);              //发送位码  
28             PORTC = SEG_CODE[i];        //发送段码  
29             _delay_ms(4);  
30         }  
31     }  
32 }
```

3.8 K1~K4 控制 LED 移位

运行本例时, 按下独立按键 K1~K4 可分别控制连接在 PC、PD 端口的 LED 上下移位显示。通常本例学习与调试, 要熟练掌握最基本的按键输入检测方法。本例电路及部分运行效果如图 3-8 所示。

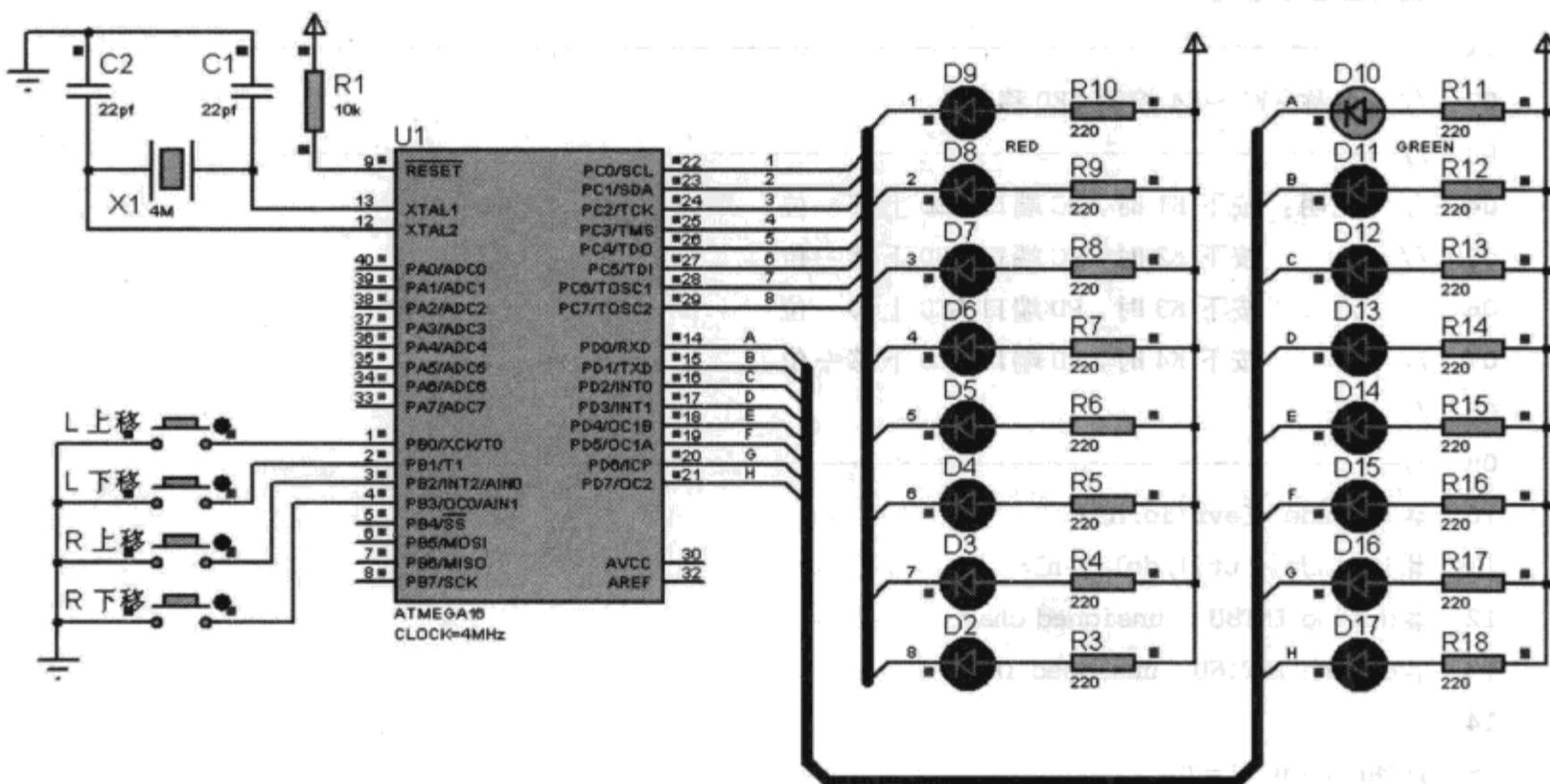


图 3-8 K1~K4 控制 LED 移位

1. 程序设计与调试

由于 K1~K4 连接在 PB 端口的低 4 位,本例在识别按键时将 PB 端口读取的值 PINB 分别与 0x01、0x02、0x04、0x08 进行“与”操作(&),如果与其中之一相与后结果为 0x00,则表明对应按键按下。这 4 个数的低 4 位分别是:0001(1)、0010(2)、0100(4)、1000(8)。

案例运行时,每当有键按下时都会立即导致 LED 移位显示,但按键未释放时不会形成 LED 连续移位显示,这是因为按键后 Recent_Key 保存了 PB 端口的按键状态信息。在下一趟循环中,如果 PB 端口的按键尚未释放,则 PINB 与 Recent_Key 相等,if 语句内的代码不会执行,Move_LED 函数不会被调用,LED 不会继续出现移位显示。

每当按键释放时,PINB 变为 0xFF,此时 PINB 与 Recent_Key 不相等,if 语句内的代码又再次执行,Recent_Key 也变为 0xFF,Move_LED 函数也被调用,但由于 Move_LED 函数内部 PINB 和 0x01、0x02、0x04、0x08 执行与操作时均不等于 0,因此不会导致移位显示。

当再次有键按下时,由于 PINB 不等于值为 0xFF 的 Recent_Key,故 LED 继续移位显示。整个程序的执行如此往复。

本例中的函数 Move_LED() 内的语句:

```
i = ( i - 1 ) & 0x07; i = ( i + 1 ) & 0x07; j = ( j - 1 ) & 0x07; j = ( j + 1 ) & 0x07;
```

将 i 和 j 的增减限制在 0~7 以内,然后通过语句 PORTC = ~ (1<<i) 与 PORTD = ~ (1<<j) 移位输出,两组各 8 只 LED 中将总是仅有 1 只 LED 被点亮。

2. 实训要求

- ① 将 K1~K4 改接在 PB 端口的高 4 位,重新修改程序实现同样的功能。
- ② 在单只数码管显示 0~9 的案例中添加按键,使按键每次按下时切换数字显示。



3. 源程序代码

```
01 //-----  
02 // 名称: K1~K4 控制 LED 移位  
03 //-----  
04 // 说明: 按下 K1 时, PC 端口 LED 上移一位  
05 //       按下 K2 时, PC 端口 LED 下移一位  
06 //       按下 K3 时, PD 端口 LED 上移一位  
07 //       按下 K4 时, PD 端口 LED 下移一位  
08 //-----  
09 //-----  
10 #include <avr/io.h>  
11 #include <util/delay.h>  
12 #define INT8U unsigned char  
13 #define INT16U unsigned int  
14  
15 INT8U i = 0, j = 0;  
16 //-----  
17 // 根据 PB 口的按键移动 LED  
18 //-----  
19 void Move_LED()  
20 {  
21     if ((PINB & 0x01) == 0x00) i = (i - 1) & 0x07; //K1  
22     else if((PINB & 0x02) == 0x00) i = (i + 1) & 0x07; //K2  
23     else if((PINB & 0x04) == 0x00) j = (j - 1) & 0x07; //K3  
24     else if((PINB & 0x08) == 0x00) j = (j + 1) & 0x07; //K4  
25  
26     PORTC = ~(1 << i);           //PC 端口第 i 位 LED 点亮  
27     PORTD = ~(1 << j);           //PD 端口第 j 位 LED 点亮  
28 }  
29  
30 //-----  
31 // 主程序  
32 //-----  
33 int main()  
34 {  
35     INT8U Recent_Key = 0x00;      //最近按键  
36     DDRB = 0x00; PORTB = 0xFF;      //PB 端口按键输入, 内部上拉  
37     DDRC = 0xFF; PORTC = 0xFE;      //PC 端口输出, 初始输出 0xFE  
38     DDRD = 0xFF; PORTD = 0xFE;      //PD 端口初值, 初始输出 0xFE  
39     while (1)  
40     {  
41         if (PINB != Recent_Key)
```

```

42     {
43         Recent_Key = PINB;           //保存最近按键
44         Move_LED();                //LED 移位显示
45         _delay_ms(10);             //延时消抖
46     }
47 }
48 }

```

3.9 数码管显示 4×4 键盘矩阵按键

当按键较多时会占用更多的控制器端口,为减少对端口的占用,本例使用了 4×4 键盘矩阵,这样大大减少了端口占用,但识别按键的代码比独立按键的代码要复杂一些。本例运行过程中,按下不同按键时,其对应的键值将显示在数码管上。本例电路及部分运行效果如图 3-9 所示。

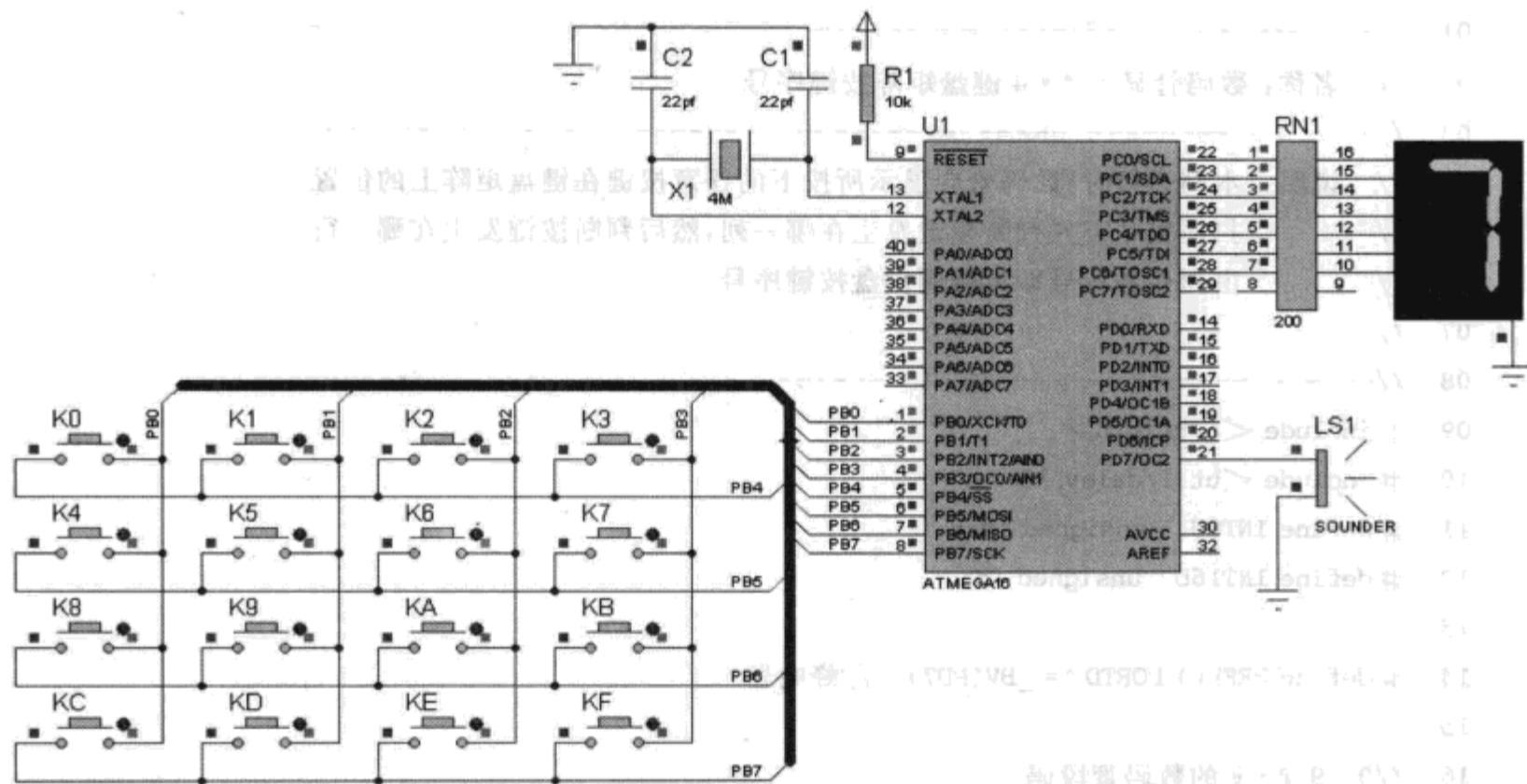


图 3-9 数码管显示 4×4 键盘矩阵按键

1. 程序设计与调试

本例键盘矩阵行线连接 PB4~PB7,列线连接 PB0~PB3,扫描过程如下:

① 程序首先判断是否有键按下。为判断 16 个按键中是否有键按下,第 29 行的函数 KeyMatrix_Down 首先通过方向寄存器 DDRB 设置 PB 端口高 4 位连接的行线为输出,低 4 位连接的列线为输入,PORTB=0x0F 在行线上先放置 4 个 0,并将列线设为内部上拉,以便从低 4 位对应的列线上读取数据。这时 PB 端口的初值为 0x0F,如果键盘矩阵中有任一按键按下,则 4 条列线上必有一位为 0,PINB 读取的将不再是 0x0F 了。因此,通过判断 PINB 是否保持为 0x0F 即可知道是否有键按下。

② 在判断有键按下后,这时 PINB 读取的值会有 4 种可能,即由原来的 00001111 变为



00001110、00001101、00001011、00000111 这 4 者之一,通过这 4 个值就可以分别判断出按键发生在 0、1、2、3 列中的哪一列了。

③ 得出当前按键所在的列以后,还需要判断出按键处于哪一行。代码中第 55 行重新设置了 PB 端口的 I/O 配置,DDRB=0x0F 将高 4 位设为输入,低 4 位设为输出,PORTB=0xF0 将高 4 位设为内部上拉,低 4 位先输出 4 个 0,这时 PB 端口的数据线上出现 11110000,所合上的按键将使高 4 位中出现一个 0,由于这个 0 所出现的位置有 4 种(也就是出现在 4 行中的某一行),11110000 被改变为 01110000、10110000、11010000、11100000 这 4 者之一。在得到所在行后,由于每行按键键值的起点分别是 0、4、8、12,将此值与列号相加即得到最终键值了。

2. 实训要求

① 修改键盘行线与列线连接方法,将高 4 位连接列线,低 4 位连接行线,重新编写程序实现键盘矩阵扫描。

② 设计 4×5 矩阵键盘,编程实现按键扫描与键值显示。

3. 源程序代码

```
01 //-----  
02 // 名称: 数码管显示 4 * 4 键盘矩阵按键序号  
03 //-----  
04 // 说明: 本例运行时,数码管会显示所按下的任意按键在键盘矩阵上的位置  
05 // 扫描程序首先判断按键发生在哪一列,然后判断按键发生在哪一行  
06 // 由列号和行号即可得到键盘按键序号  
07 //-----  
08 //-----  
09 #include <avr/io.h>  
10 #include <util/delay.h>  
11 #define INT8U unsigned char  
12 #define INT16U unsigned int  
13  
14 #define BEEP() PORTD ^= _BV(PD7) //蜂鸣器  
15  
16 //0~9,A~F 的数码管段码  
17 const INT8U SEG_CODE[] =  
18 {  
19     0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07,  
20     0x7F, 0x6F, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71  
21 };  
22  
23 //当前按键序号,该矩阵中序号范围为 0~15,16 表示无按键  
24 INT8U KeyNo = 16 ;  
25  
26 //-----  
27 // 判断键盘矩阵是否有键按下  
28 //-----
```

```

29 INT8U KeyMatrix_Down()
30 {
31     //高 4 位输出,低 4 位输入,高 4 位先置 0,放入 4 行
32     DDRB = 0xF0; PORTB = 0x0F; _delay_ms(1);
33     return PINB != 0x0F ? 1 : 0;
34 }
35
36 //-----
37 // 键盘矩阵扫描子程序
38 //-----
39 void Keys_Scan()
40 {
41     //在判断是否有键按下的函数 KeyMatrix_Down 中,
42     //高 4 位输出,低 4 位输入,高 4 位先置 0,放入 4 行
43     //按键后 00001111 将变成 0000XXXX,X 中有 1 个为 0,3 个仍为 1
44     //下面判断按键发生于 0~3 列中的哪一列
45     switch (PINB)
46     {
47         case 0B00001110: KeyNo = 0; break;
48         case 0B00001101: KeyNo = 1; break;
49         case 0B00001011: KeyNo = 2; break;
50         case 0B00000111: KeyNo = 3; break;
51         default: KeyNo = 0xFF;
52     }
53
54     //高 4 位输入,低 4 位输出,低 4 位先置 0,放入 4 列
55     DDRB = 0x0F; PORTB = 0xF0; _delay_ms(1);
56
57     //按键后 11110000 将变成 XXXX0000,X 中 1 个为 0,3 个仍为 1
58     //下面对 0~3 行分别附加起始值 0、4、8、12
59     switch (PINB)
60     {
61         case 0B11100000: KeyNo += 0; break;
62         case 0B11010000: KeyNo += 4; break;
63         case 0B10110000: KeyNo += 8; break;
64         case 0B01110000: KeyNo += 12; break;
65         default: KeyNo = 0xFF;
66     }
67 }
68
69 //-----
70 // 蜂鸣器子程序
71 //-----

```



```
72 void Beep()
73 {
74     INT8U i;
75     for (i = 0; i<100; i++)
76     {
77         _delay_ms(1); BEEP();
78     }
79 }
80
81 //-----
82 // 主程序
83 //-----
84 int main()
85 {
86     //配置数码管显示端口
87     DDRC = 0xFF; PORTC = 0x00;
88     while (1)
89     {
90         //如果键盘矩阵有键按下则扫描键值
91         if (KeyMatrix_Down()) Keys_Scan(); else continue;
92         if (KeyNo<16) //得到有效键值
93         {
94             PORTC = SEG_CODE[KeyNo]; //显示键值
95             Beep(); //响铃
96         }
97         while (KeyMatrix_Down()); //如果按键未释放则等待
98     }
99 }
```

3.10 数码管显示拨码开关编码

拨码开关常用于配置编码或设置状态,例如某些多媒体教室常用的硬件广播卡就是用拨码开关来配置编码的。本例用数码管显示当前拨码开关所设定的编码,系统运行过程中如果改动拨码设置,新的编码会立即显示在数码管上。本例电路及运行效果如图 3-10 所示。

1. 程序设计与调试

本例直接读取连接在 PB 端口的拨码开关编码值,然后将其分解为 3 个数位并显示在数码管上。语句 `PORTD=～_BV(i+1)` 用于设置共阴数码管位码,当 `i` 取值为 0、1、2 时,`～_BV(i+1)` 的值为 11111101、11111011、11110111,它们分别与 4 位数码管的 2、3、4 位对应(即数码管的右 3 位),所读取的编码值将会显示在这 3 位数码管上。

在系统运行时调整拨码开关,新的编码会立即显示在数码管上。

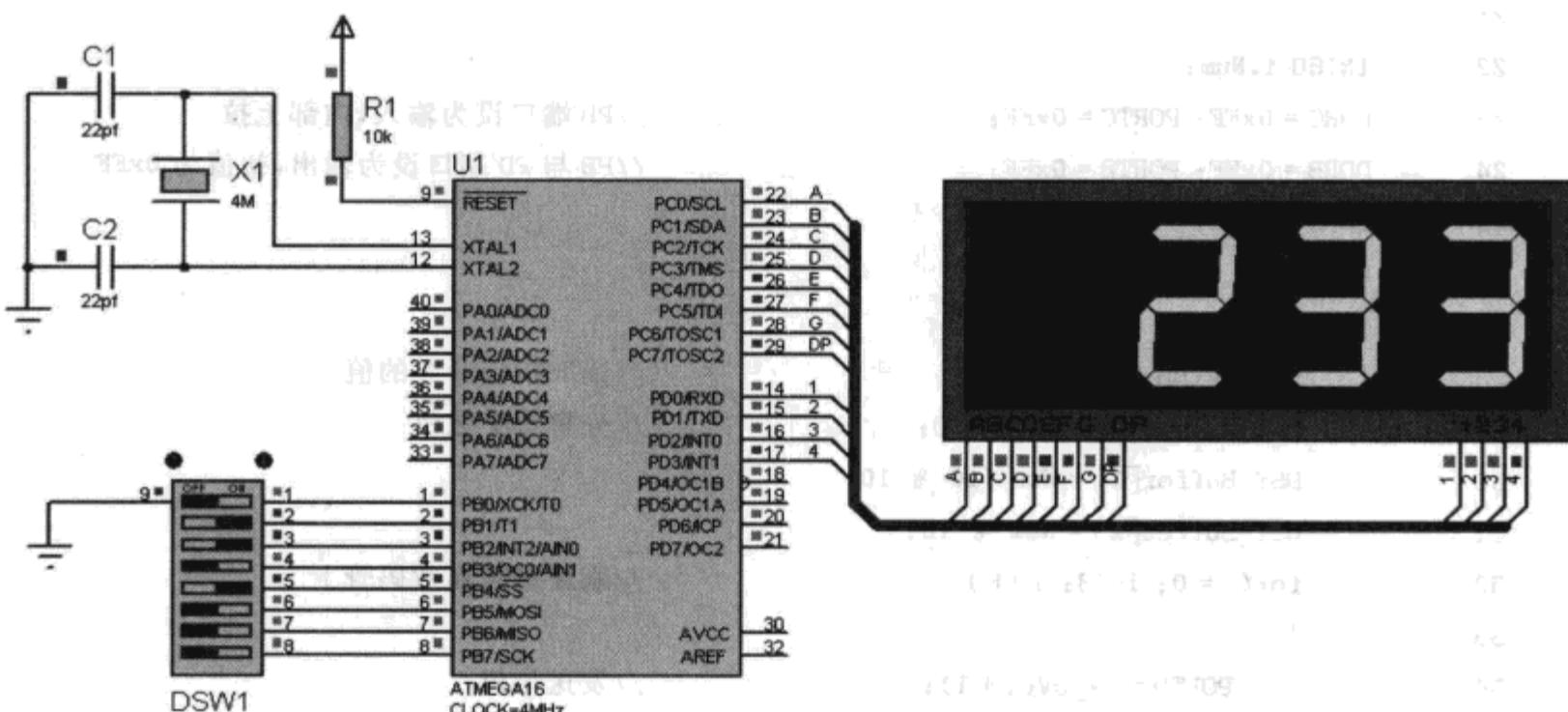


图 3-10 数码管显示拨码开关编码

2. 实训要求

- ① 重新修改本例程序,使 3 个数位从数码管左边开始显示,并将高位无效的 0 屏蔽。
- ② 将本例拨码开关改为 4 位的拨码开关,然后预备 16 种花样数组,重新编写程序,要求根据不同的编码设置显示不同的自定义花样流水灯。

3. 源程序代码

```

01 //-----
02 // 名称: 数码管显示拨码开关编码
03 //-----
04 // 说明: 系统显示拨码开关所设置的编码 000~255
05 //
06 //-----
07 # include <avr/io.h>
08 # include <util/delay.h>
09 # define INT8U unsigned char
10 # define INT16U unsigned int
11
12 //各数字的数码管段码
13 const INT8U SEG_CODE[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
14
15 //显示缓冲
16 INT8U DSY_Buffer[3] = {0,0,0};
17 //-----
18 // 主程序
19 //-----
20 int main()

```

```

21 {
22     INT8U i,Num;
23     DDRC = 0xFF; PORTC = 0xFF; //PB 端口设为输入,内部上拉
24     DDRB = 0xFF; PORTB = 0xFF; //PB 与 PD 端口设为输出,初值为 0xFF
25     DDRD = 0x00; PORTD = 0xFF;
26     while(1)
27     {
28         Num = PINB; //读取拨码开关的值
29         DSY_Buffer[0] = Num / 100; //分解 3 个数位
30         DSY_Buffer[1] = Num / 10 % 10;
31         DSY_Buffer[2] = Num % 10;
32         for(i = 0; i < 3; i++) //刷新显示在数码管上
33         {
34             PORTD = ~_BV(i + 1); //发送位码
35             PORTC = SEG_CODE[ DSY_Buffer[i] ]; //发送段码
36             _delay_ms(8);
37         }
38     }
39 }

```

3.11 继电器控制照明设备

本例用继电器控制照明设备。运行本例时,按下 K1 可点亮灯泡,再次按下时则关闭。本例电路及运行效果如图 3-11 所示。

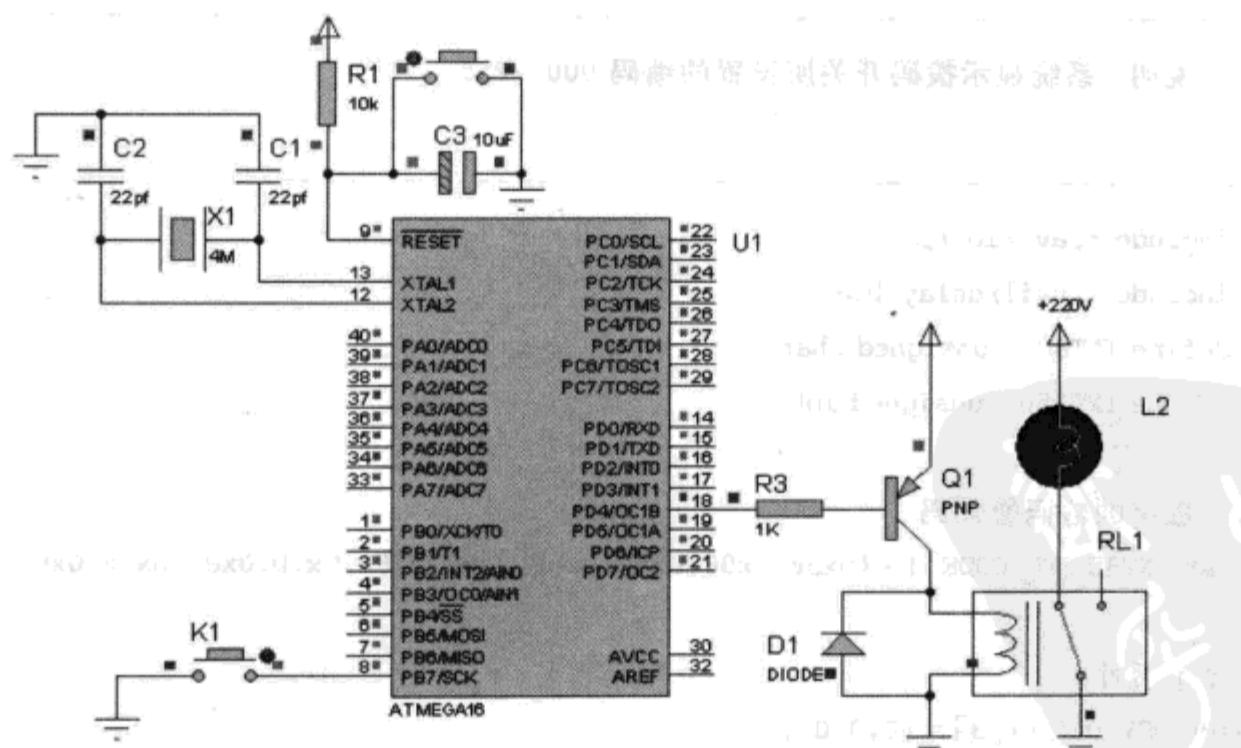


图 3-11 继电器控制照明设备

1. 程序设计与调试

本例用继电器控制外部设备,程序中将继电器定义为由 PD4 引脚控制,每次按下 K1 并释放时对 RELAY 取反。当 RELAY 为 0 时,NPN 三极管导通,继电器吸合,灯泡点亮;反之则三极管截止,继电器断开,灯泡熄灭。本例的按键定义与继电器开关定义与此前有关案例中的定义类似,大家可自行对比研究。

2. 实训要求

- ① 改用 PNP 型三极管控制继电器,并实现对外部直流电机的启停控制。
- ② 使用多个按键和多个继电器实现对多路设备的开关控制。

3. 源程序代码

```

01 //-----
02 // 名称: 继电器控制照明设备
03 //-----
04 // 说明: 按下 K1 时灯泡点亮,再次按下时灯泡熄灭
05 //
06 //-----
07 #include <avr/io.h>
08 #include <util/delay.h>
09 #define INT8U unsigned char
10 #define INT16U unsigned int
11
12 #define K1_DOWN() (PINB & _BV(PB7)) //K1 按键定义
13 #define RELAY_SWITCH() PORTD ^= _BV(PD4) //继电器开关切换控制
14 //-----
15 // 主程序
16 //-----
17 int main()
18 {
19     DDRD = 0xFF; //PD 端口输出
20     PORTD = 0xFF; //关闭继电器
21     DDRB = 0x00; //PB 端口输入
22     PORTB = 0xFF; //PB 端口内部上拉
23     while(1)
24     {
25         if (K1_DOWN()) //K1 按下
26         {
27             while (K1_DOWN()); //等待释放 K1
28             RELAY_SWITCH(); //切换继电器开关
29             _delay_ms(20);
30         }
31     }
32 }
```



3.12 开关控制报警器

本例运行过程中,将开关拨至高电平时系统发出报警声音。案例电路如图 3-12 所示。

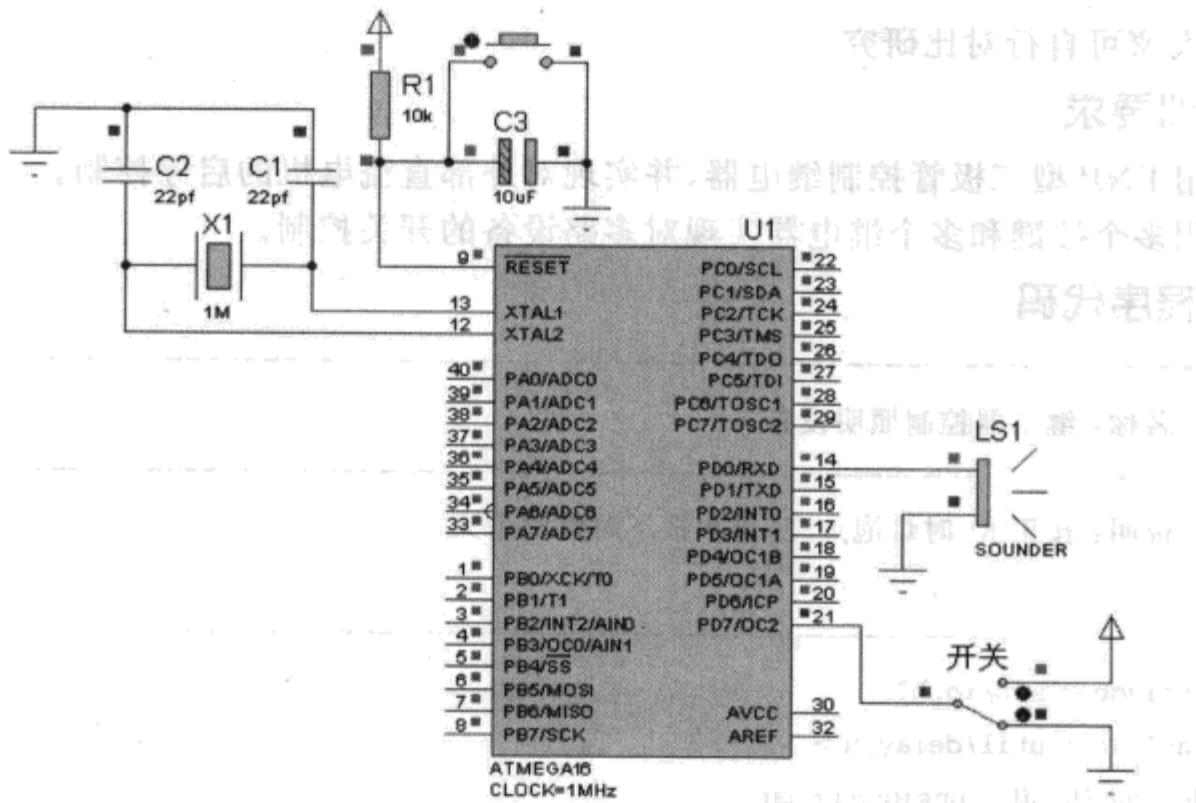


图 3-12 开关控制报警器

1. 程序设计与调试

本例代码编写关键在于 Alarm 函数的设计,函数中 SPK() 定义为:

```
SPK() (PORTD ^= _BV(PD0))
```

_BV(PD0) 即 00000001,它将 PD0 对应的第 0 位设为 1。在 PORTD 与 00000001“异或”时,前 7 位 0000000 不会使 PORTD 的前 7 位任意值发生任何变化;而最低位的 1 与 PORTD 的最低位“异或”时,如果 PORTD 最低位为 1,“异或”操作会使之变为 0。反之,如果 PORTD 的最低位为 0,“异或”操作会使之变为 1。

Alarm 函数中循环调用 SPK() 时,PD0 引脚将持续输出 1010101010…序列,形成的脉冲使 SOUNDER 发出声音。如果 SPK() 语句的执行时间间隔相等,系统会发出单调的声音,不会模拟出报警效果。

本例 Alarm 函数的双重 for 循环中,内层的 for 循环使用了参数 t。不同的 t 值使 SPK() 在不同的延时间隔后被调用,因此形成了不同的输出频率。主程序中的 Alarm(3)和 Alarm(50)使 SOUNDER 循环发出 2 种不同频率的声音,模拟出很逼真的报警器效果。

2. 实训要求

- ① 使用虚拟示波器,观察 PD0 引脚的输出波形。
- ② 重新修改参数 3 与 50,看能够听到什么样的声音效果。
- ③ 进一步修改程序,使系统模拟出其他的报警声音效果输出。

3. 源程序代码

```

01 //-----
02 // 名称：开关控制报警器
03 //-----
04 // 说明：本例用开关 S1 控制报警器，程序控制 PD0 输出 2 种不同频率的声音
05 // 模拟了很逼真的报警效果
06 //
07 //-----
08 #include <avr/io.h>
09 #include <util/delay.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int
12
13 #define S1_ON() ((PIND & _BV(PD7)) == 0x80) //S1 接高电平
14 #define SPK() (PORTD ^= _BV(PD0)) //蜂鸣器
15 //-----
16 // 发声子程序
17 //-----
18 void Alarm(INT8U t)
19 {
20     INT8U i;
21     for(i = 0; i < 200; i++)
22     {
23         SPK(); _delay_us(t); //由参数 t 控制形成不同的频率输出
24     }
25 }
26
27 //-----
28 // 主程序
29 //-----
30 int main()
31 {
32     DDRD = 0x7F; //PD 端口最高位输入，其他为输出
33     PORTD = 0xFF; //PD 端口先置高电平，其中 PD7 设为内部上拉
34     while (1)
35     {
36         if( S1_ON() ) //如果开关 S1 接高电平
37         {
38             Alarm(3); //输出高频声音
39             Alarm(50); //输出低频声音
40         }
41     }
}

```



3.13 按键发音

本例运行时,按下不同按键会听到不同频率的声音。程序中对按键操作给出了 2 种判断定义。案例电路如图 3-13 所示。

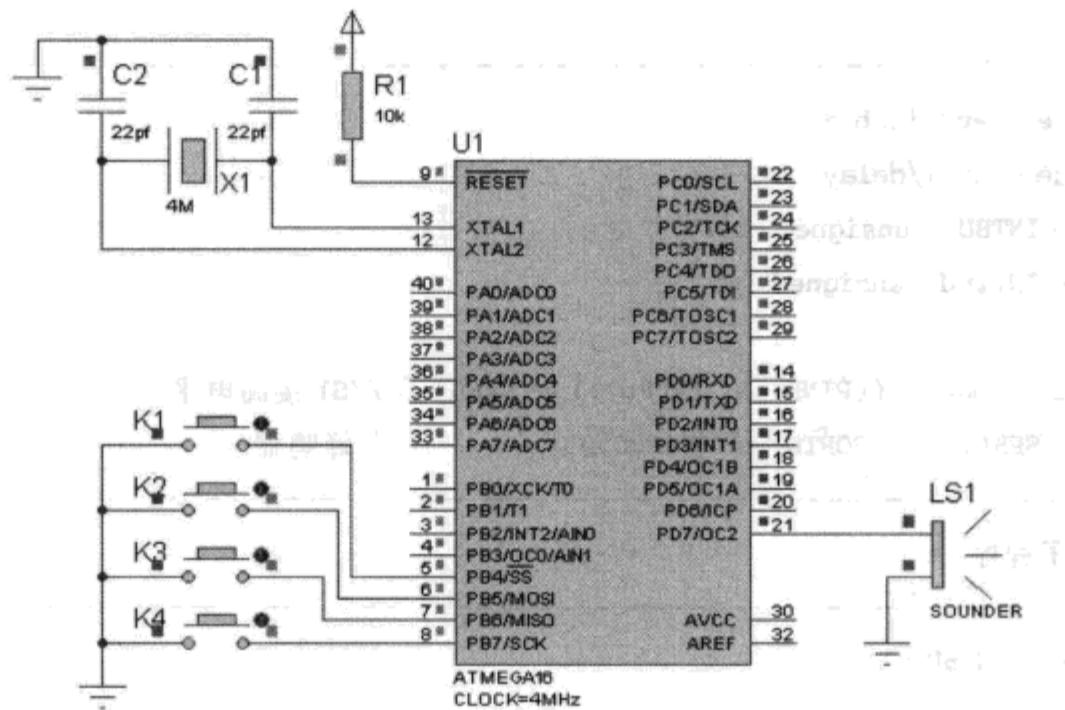


图 3-13 按键发音

1. 程序设计与调试

本例要点在于 Play 函数的编写,在按下不同按键时传给所调用的函数 Play 的参数值不同,Play 函数调用 BEEP()的_delay_us 延时间隔也不同,从而产生不同的频率输出。

对于按键操作定义,本例除了通过与操作(&)判断按键状态以外,还使用宏 bit_is_clear。宏调用格式为 bit_is_clear(sfr, bit),其中 sfr 是 AVR 单片机的特别功能寄存器,bit 是寄存器的某一位,它用于判断 sfr 中的某一位是否清零。类似地,宏 bit_is_set 用于判断 sfr 中的第 bit 位是否置位。

2. 实训要求

① 在电路中放置 7 个按键,使各键按下时可分别输出 DO、RE、ME、FA、SO、LA、XI 的声音。

② 在电路中添加一组 LED,当输出声音频率越高时点亮的 LED 越多,反之点亮的 LED 越少。

3. 源程序代码

```
01 //-----  
02 // 名称: 按键发音  
03 //-----  
04 // 说明: 本例运行时,按下不同的按键会使 SOUNDER 发出不同频率的声音。  
05 // 本例使用延时子程序实现不同频率的声音输出,后续类似案例使用
```

```

06 //      的是定时器技术
07 //
08 //-----
09 #include <avr/io.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //蜂鸣器定义
15 #define BEEP() (PORTD ^= 0x80)           //蜂鸣器
16 //K1~K4 按键定义
17 #define K1_DOWN() ((PINB & 0x10) == 0x00)    //K1 按键按下
18 #define K2_DOWN() ((PINB & 0x20) == 0x00)    //K2 按键按下
19 #define K3_DOWN() ((PINB & 0x40) == 0x00)    //K3 按键按下
20 #define K4_DOWN() ((PINB & 0x80) == 0x00)    //K4 按键按下
21
22 /* 按键 K1~K4 还可以用以下语句定义
23 #define K1_DOWN() bit_is_clear(PINB, PB4)    //K1 按键按下
24 #define K2_DOWN() bit_is_clear(PINB, PB5)    //K2 按键按下
25 #define K3_DOWN() bit_is_clear(PINB, PB6)    //K3 按键按下
26 #define K4_DOWN() bit_is_clear(PINB, PB7)    //K4 按键按下
27 */
28 //-----
29 // 按周期 t 发音
30 //-----
31 void Play(INT8U t)
32 {
33     INT8U i,
34     for(i = 0; i<100; i++)
35     {
36         BEEP(); _delay_ms(t);
37     }
38 }
39
40 //-----
41 // 主程序
42 //-----
43 int main()
44 {
45     INT8U Pre_Key = 0xFF;
46     DDRB = 0x00; PORTB = 0xFF;           //PB 端口输入, 内部上拉
47     DDRD = 0xFF;                      //PD 端口输出
48     while (1)

```

```

49      {
50          while (Pre_Key == PINB);           //如果按键状态未改变则等待
51          Pre_Key = PINB;                //保存新按键
52          if (K1_DOWN()) Play(1);        //根据不同按键输出不同频率声音
53          if (K2_DOWN()) Play(2);
54          if (K3_DOWN()) Play(3);
55          if (K4_DOWN()) Play(4);
56      }
57  }

```

3.14 INT0 中断计数

本例用 3 只七段数码管显示按键计数值。前面数个案例已多次使用过按键。本例清零键与前面某些案例一样单独作了定义,但计数键未作任何定义,该按键的识别使用了新的外部中断技术。本例使用的多只数码管是独立的,因而不存在数码管位码控制及刷新显示问题。案例电路及部分运行效果如图 3-14 所示。

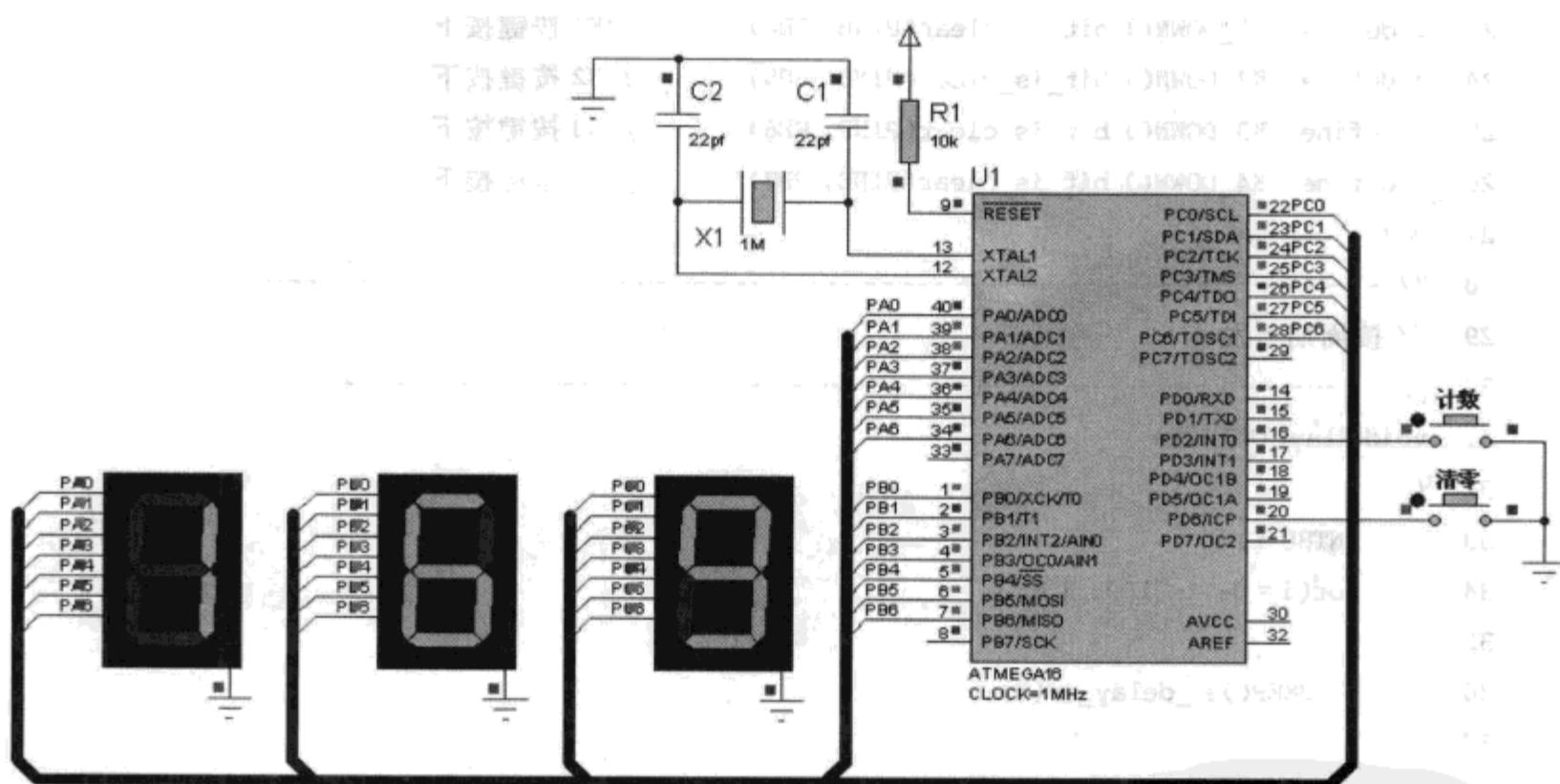


图 3-14 INT0 中断计数

1. 程序设计与调试

与此前的其他案例相比,本例新增了中断头文件<avr/interrupt.h>。

本例中的计数按键连接在单片机的 PD2 引脚(INT0),代码核心部分如下:

```

MCUCR = 0x02;           //INT0 为下降沿触发
GICR = 0x40;             //INT0 中断使能
SREG = 0x80;              //使能中断,也可使用 sei() 开中断

```

其中,MCU 控制寄存 MCUCR 低 4 位由高到低分别是:ISC1[1:0]、ISC0[1:0]。其中前

2位控制INT1中断触发方式,后2位控制INT0中断触发方式。程序中设MCUCR为0x02,即设置ISC0[1:0]为10,它将INT0设为下降沿触发异步中断。

通用中断控制寄存器GICR的高3位分别为INT1、INT0、INT2,语句GICR=0x40将INT0置位,允许INT0中断。

状态寄存器SREG的最高位I为全局中断标志位(Global Interrupt Flag Bit),设为0时中断被禁止,设为1时中断被开放,语句SREG=0x80将其最高位置位。

上述语句中,GICR=0x40还可以写成GICR=_BV(INT0),后者可读性更好,且容易编写。SREG=0x80还可用函数sei()代替。

通过以上设置,PD2(INT0)引脚上由高到低的跳变会触发INT0中断。如果按下后没有释放,则中断不会持续触发。只有在释放按键后再次按下时,才会因为又出现了高电平到低电平的跳变而再次触发中断,这样设置会使计数值仅在计数键每次重新按下时累加,不会在未释放或来不及释放时不停累加。

在本书使用的WinAVR版本下,中断服务例程格式为:

```
ISR(中断向量名称)
{
    //中断发生后要执行的语句
}
```

其中ISR即中断服务例程(Interrupt Service Routine),在查找不同芯片的中断向量名称时,可在AVRStudio的Help菜单中单击avr-libc Reference Manual(AVR库函数参数手册),打开Library Reference中有关Interrupt.h的参考资料,向下找到中断向量表,查找不同芯片和不同中断的中断向量名。本例中断向量名为INT0_vect。

本例中2个按键的识别完全不同:

- 计数键是通过中断触发来识别的,每次中断触发时即表示计数键按下,第65行的中断例程ISR(INT0_vect)将被自动调用,计数变量Count随之累加。
- 清零键是通过主程序中的while循环来轮询判断的,它持续不断地查看PD6是否变为0,如果变为0则表示清零键按下。

2. 实训要求

①修改电路和代码,用查询方式判断计数键,用中断方式控制清零键,实现相同的运行效果。

②将两键分别连接PD2(INT0)和PD3(INT1),添加INT1中断子程序实现计数清零。

3. 源程序代码

```
01 //-----
02 // 名称: INT0 中断计数
03 //-----
04 // 说明: 每次按下计数键时触发 INT0 中断, 中断程序累加计数,
05 //          计数值显示在 3 只数码管上, 按下清零键时数码管清零
06 //
07 //-----
08 #include <avr/io.h>
```



```
09 # include <avr/interrupt.h>
10 # define INT8U unsigned char
11 # define INT16U unsigned int
12
13 //清零键按下
14 # define KEY_CLEAR_ON() ((PIND & _BV(PD6)) == 0x00)
15 //0~9 的数字编码,最后一位为黑屏(索引为 10)
16 const INT8U DSY_CODE[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00};
17 //计数值
18 INT8U Count = 0;
19 //计数值分解后的各待显示数位
20 INT8U Display_Buffer[3] = {0,0,0};
21 //-----
22 // 在数码管上显示计数值
23 //-----
24 void Show_Count_ON_DSY()
25 {
26     Display_Buffer[2] = Count/100;           //分解 3 个数位
27     Display_Buffer[1] = Count % 100/10;
28     Display_Buffer[0] = Count % 10;
29
30     if(Display_Buffer[2] == 0)             //高位为 0 时不显示
31     {
32         Display_Buffer[2] = 10;
33         //高位为 0 时,如果第二位为 0 则同样不显示
34         if(Display_Buffer[1] == 0 ) Display_Buffer[1] = 10;
35     }
36
37     PORTA = DSY_CODE[Display_Buffer[2]];    //3 只数码管独立显示
38     PORTB = DSY_CODE[Display_Buffer[1]];
39     PORTC = DSY_CODE[Display_Buffer[0]];
40 }
41
42 //-----
43 // 主程序
44 //-----
45 int main()
46 {
47     DDRA = 0xFF; PORTA = 0xFF;           //PA、PB、PC 端口设为输出
48     DDRB = 0xFF; PORTB = 0xFF;
49     DDRC = 0xFF; PORTC = 0xFF;
50     DDRD = 0x00; PORTD = 0xFF;          //PD 端口设为输入,内部上拉
51     MCUCR = 0x02;                      //INT0 为下降沿触发
```

```

52     GICR = 0x40;           //INT0 中断使能
53     SREG = 0x80;          //使能总中断
54     //sei();              //或者使用 sei() 开中断
55     while(1)
56     {
57         if (KEY_CLEAR_ON()) Count = 0;      //清零
58         Show_Count_ON_DSY();               //持续刷新显示
59     }
60 }
61
62 //-----
63 // INT0 中断函数
64 //-----
65 ISR (INT0_vect)
66 {
67     Count++;                  //计数值递增
68 }

```

3.15 INT0 与 INT1 中断计数

本例同时允许 INT0 和 INT1 中断,连接 PD2 和 PD3 的 2 个按键触发中断时,对应的中断例程会分别进行计数,两组计数值将分别显示在左右各 3 只数码管上,另外 2 个按键分别用于两组计数的清零操作,对它们的判断仍使用查询法。本例电路及部分运行效果如图 3-15 所示。

1. 程序设计与调试

通过上一案例调试研究,大家已经熟悉了通用中断控制寄存器 GICR 的作用。本例同时允许 INT0 和 INT1 中断,允许 INT0 与 INT1 中断的语句 GICR=0xC0 还可以改写成:

```
GICR = _BV(INT0) | _BV(INT1);
```

主程序中 MCUCR=0x0A(00001010)将 2 个 INT0 和 INT1 中断的触发方式均设为下降沿触发,其设置方法可参考第 1 章有关中断部分的内容及 3.14 节案例中有关中断触发方式的说明。

由于主程序 while 内有对显示计数函数 Show_Counts 的循环调用,持续刷新计数值的显示,因此中断例程不需要管理计数值的显示,只需要完成累加计数。

显示计数函数 Show_Counts 首先完成 2 个计数值的数位分解,计数器 Count_A 分解后放入 Buffer_Counts 的低 3 位,Count_B 分解后放入 Buffer_Counts 的高 3 位。这 6 个数位分别显示在 6 只数码管上。

对于集成式的数码管,它们的位选择通常使用两种方法:

① 使用 _BV(i) 或 ~_BV(i) 选通共阳或共阴数码管的第 i 位。

② 使用位码表,例如本例中的 Scan_BITs。由于本例使用的是共阳数码管,数码管位控



制引脚连接 PC 端口,因此 PC 端口对应位引脚输出 1 时表示相应数码管选通,所提供的位码表正是这样设计的,每个字节中仅有位为 1,其他均为 0,每次发送 1 个位码字节时,只有 1 只数码管选通,程序用 for 循环逐个发送位码和段码,实现数码管的动态扫描显示。

本例使用的是第二种数码管扫描方法。

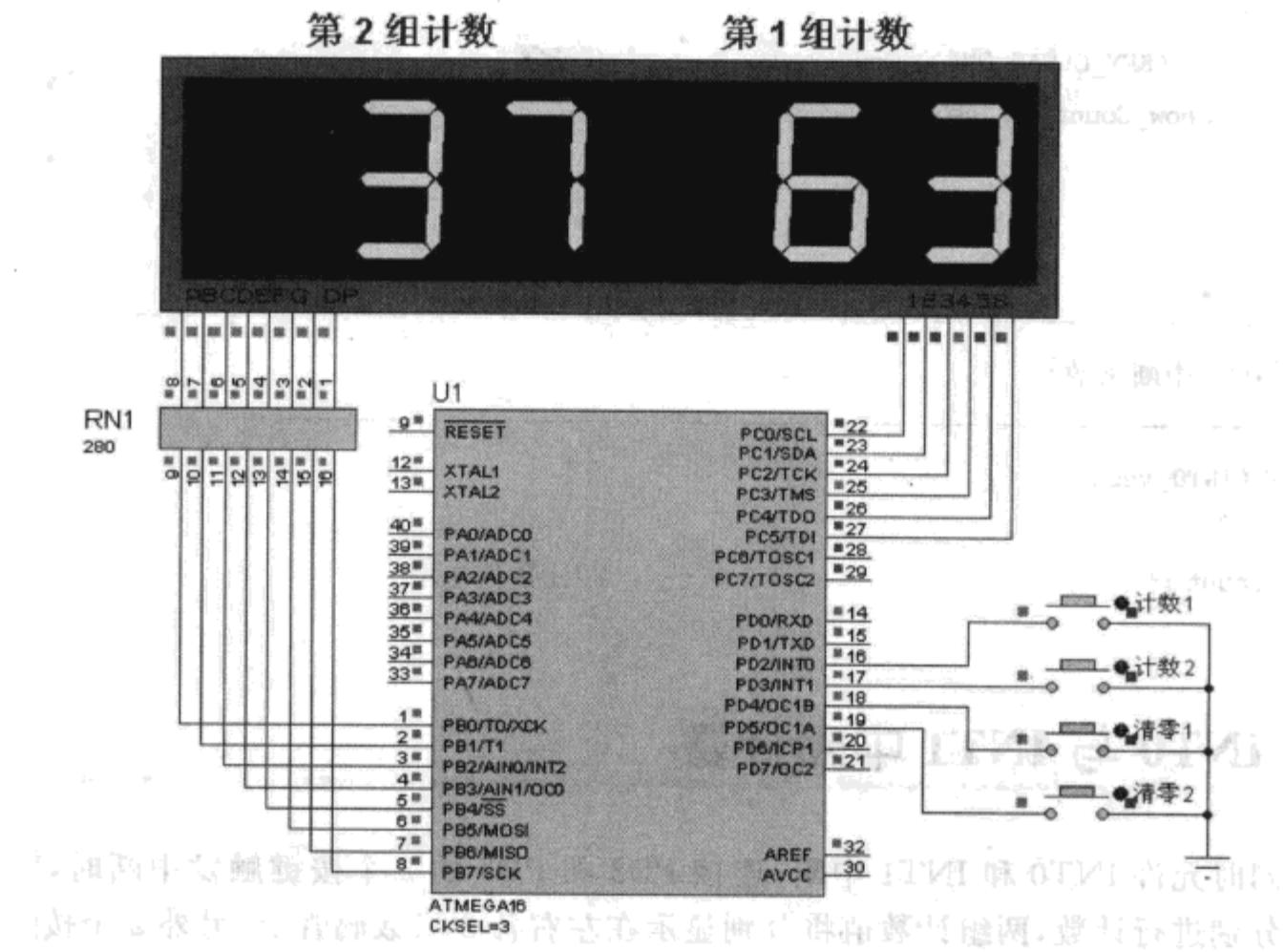


图 3-15 INT0 与 INT1 中断计数

2. 实训要求

- ① 修改代码,按第一种动态扫描显示方法实现数码管 6 个数位的显示。
- ② 本例 2 个清零键全部用查询法控制。完成本例调试后,去掉一个清零键,另一个则改接 PB2(INT2)引脚,占用 PB 端口的段码输出线改接在 PA 端口,重新编写程序,用 INT2 中断实现对两组计数统一清零。
- ③ 重新设计本例,实现篮球比赛的电子计分牌功能,每次可以计 1 分、2 分或 3 分,且具备撤消最近一次分数输入的功能,使用户可以取消误操作。
- ④ 在完成后续有关 EEPROM 案例调试后,继续改进上述程序,为避免意外掉电而丢失当前记分成绩,系统能将当前两组成绩保存于 EEPROM,重新开机后能在原有成绩上继续累加。

3. 源程序代码

```
01 //-----  
02 // 名称: INT0 与 INT1 中断计数  
03 //-----  
04 // 说明: 每次按下第 1 个计数键时第 1 组计数值累加并显示在右边的 3 只管上  
05 // 每次按下第 2 个计数键时第 2 组计数值累加并显示在左边的 3 只管上
```

```

06 //      后 2 个按键分别清零
07 //
08 //-----
09 # include <avr/io.h>
10 # include <avr/interrupt.h>
11 # include <util/delay.h>
12 # define INT8U    unsigned char
13 # define INT16U   unsigned int
14
15 # define K1_CLEAR_ON() ((PIND & 0x10) == 0x00) //清零键 1 按键按下
16 # define K2_CLEAR_ON() ((PIND & 0x20) == 0x00) //清零键 2 按键按下
17
18 //0~9 的段码表,最后一个为黑屏(索引为 10)
19 const INT8U SEG_CODE[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xFF};
20 //数码管位扫描码
21 const INT8U SCAN_BITs[] = {0x20,0x10,0x08,0x04,0x02,0x01};
22 //2 组计数的显示缓冲,前 3 位为一组,后 3 位为另一组
23 INT8U Buffer_Counts[] = {0,0,0,0,0,0};
24 //2 个计数值
25 INT16U Count_A = 0,Count_B = 0;
26 //-----
27 // 数据显示
28 //-----
29 void Show_Counts()
30 {
31     INT8U i;
32     //分解计数值 Count_A
33     Buffer_Counts[2] = Count_A/100;
34     Buffer_Counts[1] = Count_A % 100/10;
35     Buffer_Counts[0] = Count_A % 10;
36
37     if(Buffer_Counts[2] == 0)           //高位为 0 时不显示
38     {
39         Buffer_Counts[2] = 10;
40         if(Buffer_Counts[1] == 0) Buffer_Counts[1] = 10;
41     }
42
43     //分解计数值 Count_B
44     Buffer_Counts[5] = Count_B/100;
45     Buffer_Counts[4] = Count_B % 100/10;
46     Buffer_Counts[3] = Count_B % 10;
47
48     if(Buffer_Counts[5] == 0)           //高位为 0 时不显示

```



```
49      {
50          Buffer_Counts[5] = 10;
51          if(Buffer_Counts[4] == 0) Buffer_Counts[4] = 10;
52      }
53
54      //数码管显示
55      for(i = 0; i<6; i++)
56      {
57          PORTC = SCAN_BITs[i];                      //位码
58          PORTB = SEG_CODE[Buffer_Counts[i]] ;        //段码
59          _delay_ms(1);
60      }
61  }
62
63 //-----
64 // 主程序
65 //-----
66 int main()
67 {
68     DDRB = 0xFF; PORTB = 0xFF;                  //配置端口
69     DDRC = 0xFF; PORTC = 0xFF;
70     DDRD = 0x00; PORTD = 0xFF;
71     MCUCR = 0x0A;                            //INT0、INT1 中断下降沿触发
72     GICR = 0xC0;                            //INT0、INT1 中断许可
73     sei();                                //开中断
74     while(1)
75     {
76         if(K1_CLEAR_ON()) Count_A = 0;
77         if(K2_CLEAR_ON()) Count_B = 0;
78         Show_Counts();
79     }
80 }
81
82 //-----
83 // INT0 中断服务程序
84 //-----
85 ISR (INT0_vect)
86 {
87     Count_A++;
88 }
89
90 //-----
91 // INT1 中断服务程序
```

```

92 //-----
93 ISR (INT1_vect)
94 {
95     Count_B++;
96 }

```

3.16 TIMER0 控制单只 LED 闪烁

前面已有案例设计了单只或多只 LED 的闪烁，这些案例都是使用延时函数使 LED 按一定时间间隔开关显示，形成闪烁效果。本例对 LED 的闪烁延时使用了新的定时器技术。案例电路如图 3-16 所示。

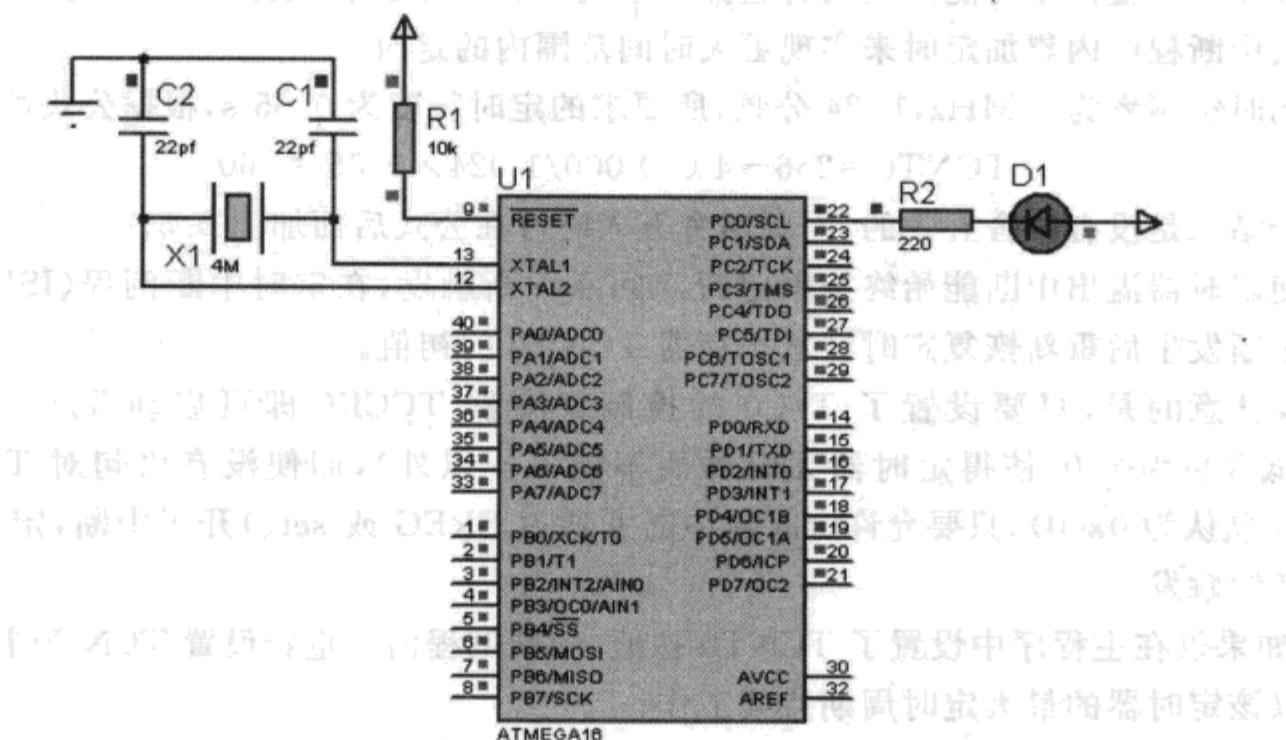


图 3-16 TIMER0 控制单只 LED 闪烁

1. 程序设计与调试

本例使用的 8 位定时器 TIMER0(T/C0)工作于定时方式，其核心计数寄存器 TCNT0 的计数时钟由系统时钟经预分频后提供，TCNT0 溢出时触发中断，中断服务程序控制 LED 点亮或关闭，形成闪烁效果。使 T/C0 工作于定时溢出中断方式时需要完成以下几项工作：

- ① 设置预分频比(TCCR0 的 CS02、CS01、CS00 位)，选择 1、8、64、256、1024 分频比；
- ② 设置定时/计数寄存器初值(TCNT0)，设置范围为 0~255；
- ③ 设置 T/C0 中断屏蔽寄存器，允许溢出中断(TIMSK 的最低位 TOIE0 用于允许或禁止 T/C0 溢出中断)；
- ④ 编写定时溢出中断服务程序 ISR(本例 T/C0 溢出中断向量为：TIMER0_OVF_vect)；
- ⑤ 开中断(设置 SREG 或使用 sei())。
- ⑥ 编写定时器溢出中断例程，当 TCNT0 计数溢出时触发中断，所编写的相应中断例程将被自动调用。

8 位 T/C0 定时器的计数寄存器 TCNT0 累加计数，从设定的初值累加到全 1(即 255)时，

如果再增加 1 即溢出, TCNT0 变为全 0。如果需要在计数 n 以后溢出, 则 TCNT0 的初值应设为 $TCNT0 = 256 - n$ 。

如果要 T/C0 实现 x 秒(s) 定时, 需将 x 秒(s) 换算成所需要的计数次数 n , 若系统时钟频率为 $F_{CPU}(\text{Hz})$, 在分频后 TCNT0 的实际计数时钟 $t_{clk} = F_{CPU}/\text{分频}$, t_{clk} 即 1 s 时间内 TCNT0 的计数次数, 如果需要定时 0.05 s, 则所需要计数次数 $n = t_{clk} \times 0.05$, 由于 TCNT0 是累加计数, 因此还需要用 256 减去 n , 即 $256 - t_{clk} \times 0.05$ 。

由以上分析, 可得 T/C0 定时器的计数寄存器 TCNT0 的初值计算公式为:

$$TCNT0 = 256 - F_{CPU}/\text{预分频} * \text{定时长度}(s)$$

当然, 此公式中减号后面的部分(也就是上面的 n 值)不得大于 255, 否则是不能实现预期的定时效果的。由公式也可以看出, 在固定时钟频率下, 如果需要实现尽可能大的定时长度, 可将预分频比设置得尽可能大些或者选择较小的时钟频率。在实际应用中, 还可以通过在定时器溢出中断程序内累加定时来实现更大时间范围内的定时。

本例时钟频率为 4 MHz, 1024 分频, 所要求的定时长度为 0.05 s, 根据公式可得:

$$TCNT0 = 256 - 4\ 000\ 000/1\ 024 \times 0.05 \approx 60$$

这个结果是没有四舍五入的, 需要四舍五入时可在公式后面加上 0.5。

为使定时器溢出中断能始终按固定时间间隔不断触发, 在定时中断例程 (ISR) 内还需要在每次中断发生后重新恢复定时计数寄存器 TCNT0 的初值。

需要注意的是, 只要设置了 T/C0 的控制寄存器 TCCR0 即可启动 T/C0 定时器(除 TCCR0 低 3 位为全 0, 使得定时器无时钟而不能工作以外), 即使没有语句对 TCNT0 赋值 (TCNT0 默认为 0x00), 只要允许定时器中断并通过 SREG 或 sei() 开了中断, 定时器中断程序仍会持续触发:

① 如果仅在主程序中设置了 TCNT0 初值, 中断例程内未重新设置 TCNT0 初值, 则中断程序将以该定时器的最大定时周期持续工作;

② 如果在中断程序内重新设置了 TCNT0 初值, 主程序中未设置 TCNT0 初值, 则第一次触发的周期是最长的, 以后会以中断程序中 TCNT0 指定的定时周期持续触发。

③ 如果主程序和中断程序中都没有指定 TCNT0 的初值, 则定时器自启动的时候起, 中断将一直以最大定时周期被持续触发。

另外, 本例定时器每隔 0.05 s 触发中断, 为实现 0.25 s 的定时长度, 定时中断程序内使用累加全局变量来实现更长的计时。本例使用 T_Count 累加实现了 5 倍于定时器计时周期 (0.05 s) 的更长计时周期 (0.25 s)。如果其他函数不需要使用 T_Count 变量, 可将该变量定义成中断例程内的静态变量。

2. 实训要求

- ① 用定时器 0 计时溢出中断控制单只数码管按一定时间间隔滚动显示数字 0~9。
- ② 在电路中改用蜂鸣器, 用定时器实现某频率声音的输出。

3. 源程序代码

```

01 //-----
02 // 名称: 定时器 0 控制单只 LED 闪烁
03 //-----

```

```

04 // 说明：LED 在 T0 定时器溢出中断控制下不断闪烁
05 //
06 //-----
07 #define F_CPU 4000000UL
08 #include <avr/io.h>
09 #include <avr/interrupt.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int
12
13 #define LED_BLINK() (PORTC ^= 0x01) //LED 闪烁
14 INT16U T_Count = 0; //用于延时累加的变量
15 //-----
16 // 主程序
17 //-----
18 int main()
19 {
20     DDRC = 0x01; //PC0 引脚设为输出
21     TCCR0 = 0x05; //预分频：1024
22     TCNT0 = 256 - F_CPU/1024.0 * 0.05; //晶振 4 MHz, 0.05 s 定时
23     TIMSK = 0x01; //使能 T0 中断
24     sei(); //开中断
25     while (1);
26 }
27
28 //-----
29 // T0 定时器溢出中断服务程序
30 //-----
31 ISR (TIMER0_OVF_vect)
32 {
33     TCNT0 = 256 - F_CPU/1024.0 * 0.05; //重装 0.05 s 定时
34     if ( ++ T_Count != 5 ) return; //不到 0.05 * 5 = 0.25 s 时返回
35     T_Count = 0;
36     LED_BLINK(); //每 0.25 s 开或关一次 LED, 形成闪烁效果
37 }

```

3.17 TIMER0 控制流水灯

本例使用定时器 TIMER0 控制 P0 端口和 P2 端口的两组 LED 滚动显示。本案例中定时器工作于查询方式，3.16 节案例中的定时器工作于中断方式。本例电路及部分运行效果如图 3-17 所示。

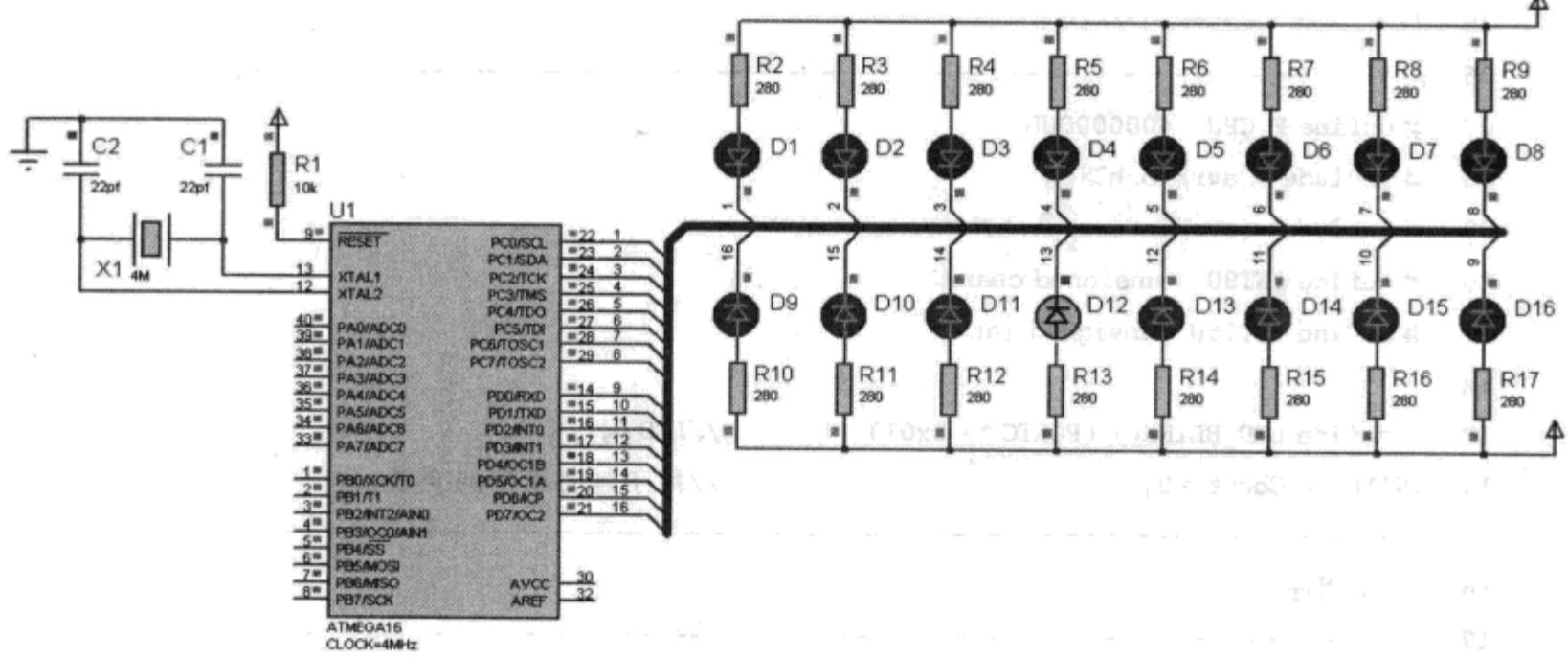


图 3-17 TIMER0 控制流水灯

1. 程序设计与调试

主程序首先将 Pattern 设为 0xFFFF，Pattern 的 16 位中只有最低位为 0。程序中第 21、22 行完成设置后定时器随即启动，定时/计数开始。

本例控制 LED 滚动时没有使用定时溢出中断程序，主程序中第 23 行的 while 循环用来不断检测定时/计数器中断标志寄存器 TIFR 的最低位 TOV0 是否为 1(注意：TOV0 是 T/C0 Timer Overflow 的缩写)，在溢出时 TOV0 被置位，通过写入 1 可将其清零。30~34 行完成 16 只 LED 的循环滚动显示，每次定时溢出时滚动一位。

2. 实训要求

- ① 仍使用本例的定时器溢出查询方式，实现单只数码管滚动显示单个数字。
- ② 重新使用定时器溢出中断方式改写本例代码，实现相同的运行效果。

3. 源程序代码

```

01 //-----
02 // 名称：TIMER0 控制流水灯
03 //-----
04 // 说明：定时器控制 PC,PD 端口的 LED 滚动显示，本例定时器工作于查询方式，
05 // 上一案例中的定时器工作于中断方式
06 //
07 //-----
08 #define F_CPU 4000000UL
09 #include <avr/io.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int
12
13 INT16U Pattern = 0xFFFF; //16 只 LED 的显示初值
14 //-----

```

```

15 // 主程序
16 //——
17 int main()
18 {
19     DDRC = 0xFF; DDRD = 0xFF;           //PC,PD 端口均设为输出
20     PORTC = 0xFF; PORTD = 0xFF;         //初始时关闭所有 LED
21     TCCR0 = 0x05;                      //预分频:1024
22     TCNT0 = 256 - F_CPU/1024.0 * 0.05; //晶振 4 MHz,0.05 s 定时初值
23     while (1)
24     {
25         while( !(TIFR & _BV(TOVO)) );    //等待 T0 溢出标志 TOVO 置位
26         TIFR = _BV(TOVO);                //通过对 TOVO 写 1 实现软件清零
27
28         TCNT0 = 256 - F_CPU/1024.0 * 0.05; //重装 T0 定时初值
29
30         PORTC = (INT8U)Pattern;          //PC 端口对应 Patter 的低 8 位
31         PORTD = (INT8U)(Pattern >> 8); //PD 端口对应 Patter 的高 8 位
32
33         Pattern = Pattern << 1 | 0x0001; //16 位 Pattern 中唯一的 0 左移 1 位
34         if (Pattern == 0xFFFF) Pattern = 0xFFE;
35     }
36 }

```

3.18 TIMER0 控制数码管扫描显示

对于集成式数码管的多位显示,一般采用的是动态扫描刷新显示方法,在发送段码与位码完成一位数码显示后,调用 `_delay_ms` 函数,在短暂延时后显示下一位数码。如此不停地快速刷新显示,用户将感觉不到数码管的任何显示抖动或闪烁,而会觉得所有数位是同时呈现在数码管上的。

本例改用了新的动态显示方法,数码管动态刷新程序放在 T/C0 定时器溢出中断函数内,案例同样实现了多位集成式数码管的动态显示。本例电路及部分运行效果如图 3-18 所示。

1. 程序设计与调试

本例技术要点在于数码管的动态刷新显示过程由定时器溢出中断子程序完成的,位间延时不再通过调用延时函数 `_delay_ms` 来完成,而此前的集成式多位数码管则是使用循环和延时子程序来控制数码管持续刷新显示的。

定时器 T/C0 的计数寄存器 TCNT0 初值选择很重要,设置不当将导致数码管显示闪烁、亮度不足或字符抖动。对于本例的 8 位数码管,程序将其设为每隔 4 ms 切换显示下一位字符。由于人的视觉惰性,在快速切换显示时感觉不到它们是逐个出现并在 4 ms 后消失的,而是觉得所有字符是同时稳定地显示在所有数码管上。

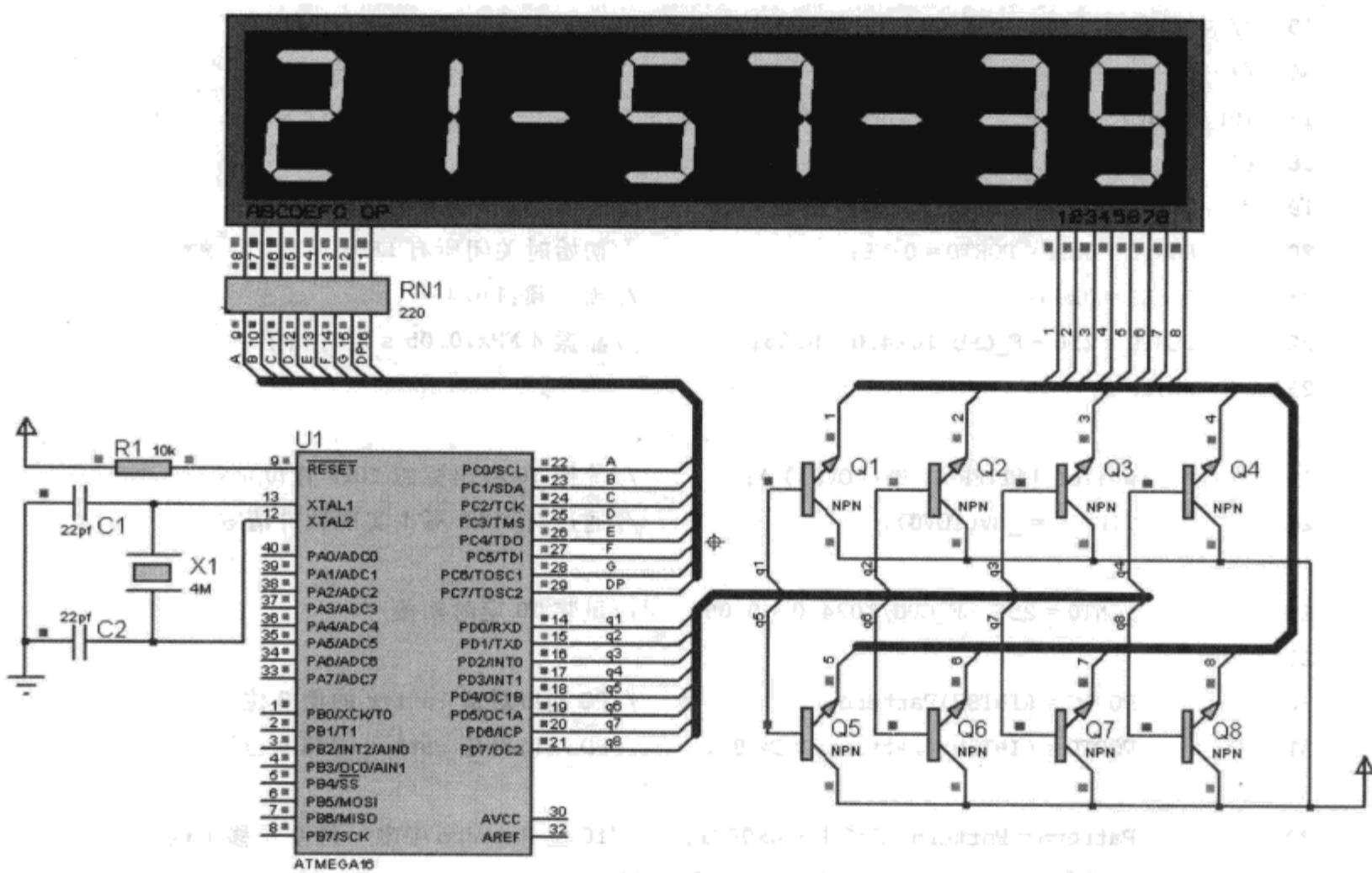


图 3-18 TIMER0 控制数码管扫描显示

中断子程序内控制的变量 i, j 分别是二维数组 Table_OF_Digits 的行/列索引, 定时器中断每隔 4 ms 触发, 数组第 i 行 j 列字符被显示, 同时 j 递增, 4 ms 后中断再次触发时, 下一个字符被显示, 依次下去, 第 i 行的 8 个字符会被反复循环刷新显示在 8 位数码管上。

二维数组中一行 8 个字符的持续刷新显示时间由变量 t 控制, 时间近似等于 $t \times 4$ ms, 增加 t 值会延长一行字符保持显示在数码上的时间长度。本例中 t 取值为 350, 要注意将 t 定义为 INT16U 类型。在一行 8 个字符保持显示若干时间后, i 的增加会使数码管显示出下一行字符。

如果细心的话, 大家可能会发现, 每一趟刷新需要显示 8 个字符, 在 t 值为 350 时开始切换到下一行, 由于 $43 \times 8 = 344$, t 增加到 350 时, 刚刚显示完的是第 44 趟第 6 个字符, 在第 44 趟还剩 2 个字符未显示时, 变化 i 值而切换到另一行, 这样会不会出现显示错误呢?

实际结果是不管 t 的上限是否能整除 8, 显示结果都是正常的。在 $t = 350$ 时, t 归 0, 数码管上前 6 个字符仍是数组当前行的, i 值变更后, 后续显示的将是新行的第 7、8 个字符, 这时数码管上前 6 个字符是一行的, 后 2 个字符是另一行的, 这样显然会出现 2 行混合出现的情况, 但由于每个字符仅停留 4 ms 即被刷新, 前面 6 个异常的字符会在极短的时间内在第 45 趟(或称为新开始的第 0 趟)被刷新为新行的前 6 个字符, 因此根本看不到这种混合出现的情况。

如果希望切换显示新行时不出现可能的瞬间混合显示现象, 要么将 t 上限取为可与 8 除尽, 或者直接在变更 i 值后, 再添加语句:

```
j = 0;
```

这会使得任何时候切换到新行时, 数据都从新行第 0 个元素开始重新刷新显示。

2. 实训要求

- ① 在本例第 18 行的二维数组中再添加一行待显示数据，实现 3 组数据的切换显示。
- ② 重新编写程序，使每组数据能从左向右滚动进入显示屏，保持刷新显示一段时间后再切换显示下一组数据。

3. 源程序代码

```

01 //-----
02 // 名称：T0 定时器控制数码管扫描显示
03 //-----
04 // 说明：8 只数码管分两组动态显示年月日与时分秒，本例的位显示延时
05 //       用定时器实现，未使用前面案例中常用的延时函数
06 //
07 //-----
08 #define F_CPU 4000000UL
09 #include <avr/io.h>
10 #include <avr/interrupt.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //0~9 的数码管段码，最后一位是“-”的段码，索引为 10
15 const INT8U SEG_CODE[] =
16 { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF };
17
18 //待显示数据 09 - 12 - 25 与 21 - 57 - 39(分为两组显示)
19 const INT8U Table_OF_Digits[][][8] =
20 {
21     {0,9,10,1,2,10,2,5},
22     {2,1,10,5,7,10,3,9}
23 };
24
25 INT8U i = 0, j = 0;
26 INT16U Keep_Time = 0; //保持显示时长
27 //-----
28 // 主程序
29 //-----
30 int main()
31 {
32     DDRC = 0xFF; PORTC = 0x00; //配置输出端口
33     DDRD = 0xFF; PORTD = 0x00;
34     TCCR0 = 0x03; //预设分频:64
35     TCNT0 = 256 - F_CPU/64.0 * 0.004; //晶振 4 MHz,4 ms 定时初值
36     TIMSK = 0x01; //允许 T0 定时器溢出中断

```

```

37     sei();                                //开中断
38     while (1);
39 }
40
41 //-----
42 // T0 定时器溢出中断程序(控制数码管扫描显示)
43 //-----
44 ISR (TIMER0_OVF_vect )
45 {
46     TCNT0 = 256 - F_CPU/64.0 * 0.004;      //位间延时 4 ms
47     PORTC = 0xFF;                          //先关闭段码
48     PORTC = SEG_CODE[ Table_OF_Digits[i][j]]; //输出数码管段码
49     PORTD = _BV(j);                      //输出位扫描码
50
51     //数组第 i 行的下一字节索引(0 ~ 7)
52     j = (j + 1) % 8;                     //或使用 j = (j + 1) & 0x07;
53     //每组的 8 个字符位保持刷新一段时间
54     if( ++Keep_Time != 350 ) return;
55     Keep_Time = 0;
56     //刷新若干遍数后切换
57     //数组行 i = 0 时显示年月日,i = 1 时显示时分秒
58     i = (i + 1) % 2;                     //或使用 i = (i + 1) & 0x01;
59 }

```

3.19 TIMER1 控制交通指示灯

Proteus 内置了交通指示灯组件,本例用 T/C1 定时器控制交通指示灯按一定时间间隔切换显示。为了能快速观察到红绿灯切换及黄灯闪烁的效果,本例调短了切换时间间隔。本例电路如图 3-19 所示。

1. 程序设计与调试

本例使用的 T/C1 定时器工作于定时溢出中断方式,实现对交通指示灯所有切换过程的控制,因为指示灯切换有 4 种不同操作,程序中用变量 Operation_Type 表示当前的操作类型,取值 1~4 对应的操作分别如下:

- ① 东西向绿灯与南北向红灯亮 5 s;
- ② 东西向绿灯灭,黄灯闪烁 5 次;
- ③ 东西向红灯与南北向绿灯亮 5 s;
- ④ 南北向绿灯灭,黄灯闪烁 5 次。

在第④项操作后回到第①项操作继续重复。

本例所使用的 T/C1 定时器是 16 的定时/计数器,定时控制寄存器 TCCR1B 的低 3 位用于设置分频比,TCCR1B=0x03 设置 64 分频,本例设置 T/C1 定时/计数寄存器 TCNT1 初值

的计算公式如下：

$$TCNT1 = 65536 - F_{CPU}/分频 \times \text{定时长度}(s)$$

上述公式中减号后面的部分计算出来的值必须在 0~65535 以内。

本例其他设计类似于 3.4 节 LED 模拟交通灯的案例，大家可以自行比对阅读。

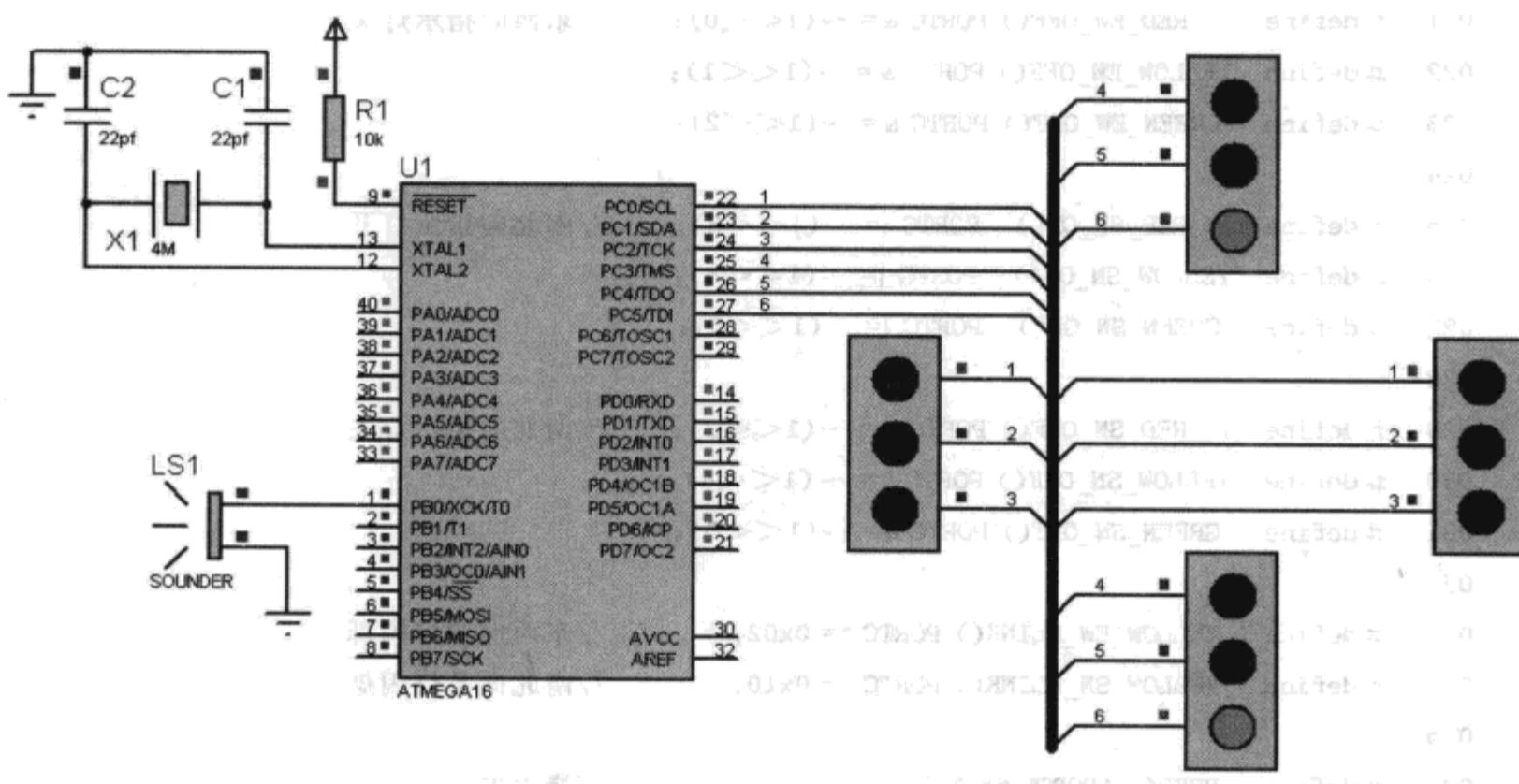


图 3-19 TIMER1 控制交通指示灯

2. 实训要求

- ① 用 T0 定时器重新设计本例，实现相同的显示效果。
- ② 重新调整切换与闪烁时间，实现完整的交通指示灯仿真效果。

3. 源程序代码

```

001 //-----
002 // 名称：定时器 T1 控制交通指示灯
003 //-----
004 // 说明：东西向绿灯亮 5 s 后，黄灯闪烁，闪烁 5 次后亮红灯
005 // 红灯亮后，南北向由红灯变为绿灯，5 s 后南北向黄灯闪烁
006 // 闪烁 5 次后亮红灯，东西向绿灯亮，如此往复。
007 // 本例将时间设得较短是为了调试的时候能较快地观察到运行效果
008 //
009 //-----
010 #define F_CPU 4000000UL
011 #include <avr/io.h>
012 #include <avr/interrupt.h>
013 #include <util/delay.h>
014 #define INT8U unsigned char
015 #define INT16U unsigned int
016

```



```
017 #define RED_EW_ON() PORTC |= (1<<0); //东西向指示灯开
018 #define YELLOW_EW_ON() PORTC |= (1<<1);
019 #define GREEN_EW_ON() PORTC |= (1<<2);
020
021 #define RED_EW_OFF() PORTC &= ~(1<<0); //东西向指示灯关
022 #define YELLOW_EW_OFF() PORTC &= ~(1<<1);
023 #define GREEN_EW_OFF() PORTC &= ~(1<<2);
024
025 #define RED_SN_ON() PORTC |= (1<<3); //南北向指示灯开
026 #define YELLOW_SN_ON() PORTC |= (1<<4);
027 #define GREEN_SN_ON() PORTC |= (1<<5);
028
029 #define RED_SN_OFF() PORTC &= ~(1<<3); //南北向指示灯关
030 #define YELLOW_SN_OFF() PORTC &= ~(1<<4);
031 #define GREEN_SN_OFF() PORTC &= ~(1<<5);
032
033 #define YELLOW_EW_BLINK() PORTC ^= 0x02; //东西向黄灯闪烁
034 #define YELLOW_SN_BLINK() PORTC ^= 0x10; //南北向黄灯闪烁
035
036 #define BEEP() (PORTB ^= 0x01) //蜂鸣器
037
038 //延时倍数,闪烁次数,操作类型变量
039 INT8U Time_Count = 0, Flash_Count = 0, Operation_Type = 1;
040 //-----
041 // 主程序
042 //-----
043 int main()
044 {
045     DDRB = 0xFF; PORTC = 0xFF; //配置输出端口
046     DDRC = 0xFF; PORTC = 0x00;
047     TCCR1B = 0x03; //T1 预设分频:64
048     TCNT1 = 65536 - F_CPU/64.0 * 0.5; //晶振 4 MHz,0.5 s 定时初值
049     TIMSK = _BV(TOIE1); //允许 T1 定时器溢出中断
050     sei(); //开中断
051     while (1);
052 }
053
054 //-----
055 // 黄灯警报声音输出
056 //-----
057 void Yellow_Light_Alarm()
058 {
```

```

059     INT8U i;
060     for (i = 0; i < 100; i++)
061     {
062         BEEP(); _delay_us(380);
063     }
064 }
065
066 //-----
067 // T1 定时器溢出中断服务程序(控制交通指示灯切换显示)
068 //-----
069 ISR (TIMER1_OVF_vect)
070 {
071     TCNT1 = 65536 - F_CPU/64.0 * 0.5;           //重装 0.5 s 定时初值
072     switch (Operation_Type)
073     {
074         case 1: //东西向绿灯与南北向红灯亮,5 s 后绿灯灭
075             RED_EW_OFF(); YELLOW_EW_OFF(); GREEN_EW_ON();
076             RED_SN_ON();  YELLOW_SN_OFF(); GREEN_SN_OFF();
077             //5 s 后切换操作 (0.5 s * 10 = 5 s)
078             if (++Time_Count != 10) return;
079             Time_Count = 0;
080             Operation_Type = 2;           //下一操作
081             break;
082
083         case 2: //东西向绿灯灭,黄灯开始闪烁
084             Yellow_Light_Alarm();
085             GREEN_EW_OFF();
086             YELLOW_EW_BLINK();
087             //闪烁 5 次
088             if (++Flash_Count != 10) return;
089             Flash_Count = 0;
090             Operation_Type = 3;           //下一操作
091             break;
092
093         case 3: //东西向红灯与南北向绿灯亮
094             RED_EW_ON();  YELLOW_EW_OFF(); GREEN_EW_OFF();
095             RED_SN_OFF(); YELLOW_SN_OFF(); GREEN_SN_ON();
096             //南北向绿灯亮 5 s 后切换(0.5 s * 10 = 5 s)
097             if (++Time_Count != 10) return;
098             Time_Count = 0;
099             Operation_Type = 4;           //下一种操作类型
100            break;

```



```
101
102     case 4: //南北向绿灯灭,黄灯开始闪烁
103         Yellow_Light_Alarm();
104         GREEN_SN_OFF();
105         YELLOW_SN_BLINK();
106         //闪烁 5 次
107         if ( ++Flash_Count != 10) return;
108         Flash_Count = 0;
109         Operation_Type = 1; //回到第一种操作
110     }
111 }
```

3.20 TIMER1 与 TIMER2 控制十字路口秒计时显示屏

本例运行时,东西向蓝色数码管与南北向红色数码管同步倒计时,若干秒后交换,如此往复。在倒计时过程中,如果仅剩下 5 s 时,系统会发现报警提示声音。本例同时启用了 2 个定时器 T/C1 和 T/C2,其中 16 位的 T/C1 定时器负责递减秒数及切换方向,8 位的 T/C2 定时器负责刷新数码管显示。本例电路及部分运行效果如图 3-20 所示。

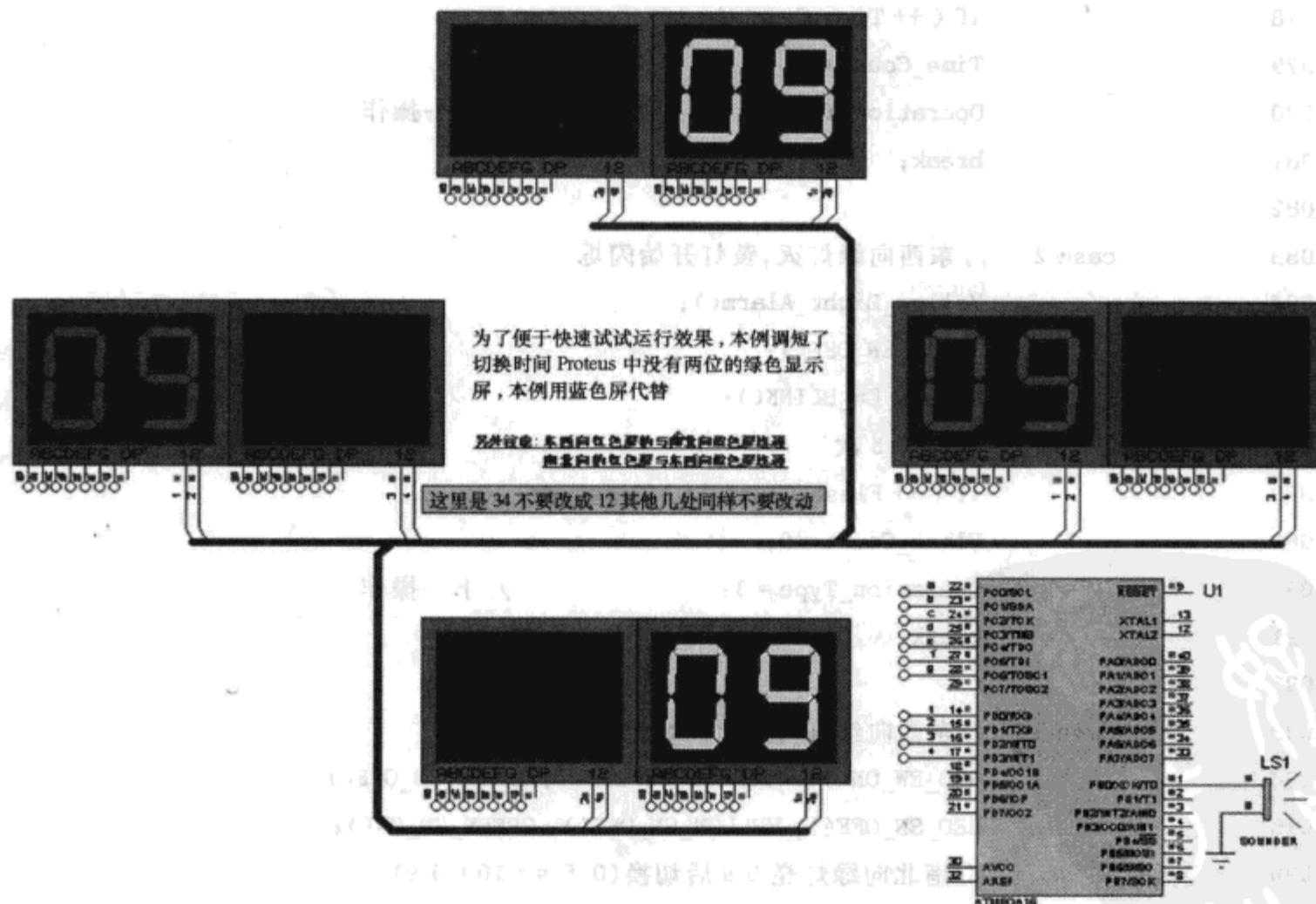


图 3-20 TIMER1 与 TIMER2 控制十字路口秒计时显示屏

1. 程序设计与调试

本例同时启用 16 位的 T/C1 和 8 位的 T/C2 定时器。以下代码分别设置了 T/C1 与 T/C2 的分频比及定时初值, 定时中断屏蔽寄存器 TIMSK 设置为同时允许 T/C1 和 T/C2 定时溢出中断。

```
TCCR1B = 0x03;                                //T1 预设分频:64
TCNT1 = 65536 - F_CPU/64.0 * 1.0;              //晶振 4 MHz, 1 s 定时初值
TCCR2 = 0x04;                                  //T2 预设分频:64
TCNT2 = 256 - F_CPU/64.0 * 0.004;               // 4 MHz 系统时钟, 0.004 s 定时初值
TIMSK = _BV(TOIE1) | _BV(TOIE2);                //同时允许 T1、T2 定时器溢出中断
```

T/C1 定时溢出中断每秒触发一次, 中断程序控制秒数递减, 并在仅剩下 5 s 时开始发出警报声音。T/C2 定时溢出中断程序以 4 ms 周期刷新数码管显示。该中断程序中根据方向的切换, 控制(0,1)位或(2,3)位数码管的刷新显示。

2. 实训要求

- ① 进一步完善本例设计, 仿真十字路口计时屏的切换效果。
- ② 重新用 T/C0 和 T/C2 定时器改编本例。

3. 源程序代码

```
001 //-----
002 // 名称: TIMER1 与 TIMER2 控制十字路口秒计时显示屏
003 //-----
004 // 说明: 本例运行时, 东西向蓝色数码管与南北向红色数码管同步倒计时,
005 // 若干秒后交换, 如此反复。
006 // 本例使用的两个定时器中, T/C1 定时器负责递减秒数及切换方向,
007 // T/C2 定时器负责刷新显示数码管
008 //
009 //-----
010 #define F_CPU 4000000UL
011 #include <avr/io.h>
012 #include <avr/interrupt.h>
013 #include <util/delay.h>
014 #define INT8U unsigned char
015 #define INT16U unsigned int
016
017 #define BEEP() (PORTB ^= 0x01)
018
019 //设置最大秒数为 12 s, 每 12 s 将切换通行方向
020 //这里为了能尽快观察到运行效果而将该值设得较小
021 #define MAX_SECOND 12
022
023 //通行方向类型(东西/南北)
024 enum TRAFFIC_DIRECTION {EW, SN} Current_Direct;
```



```
025
026 //0~9 的数码管段码
027 const INT8U SEG_CODE[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};
028
029 //当前剩余秒数(两位)及秒显示缓冲
030 int Remain_Second;
031 INT8U Second_Display_Buffer[] = {0,0};
032 //-----
033 // 根据剩余秒数 Remain_Second 刷新秒显示缓冲
034 //-----
035 void Refresh_Second_Display_Buffer()
036 {
037     Second_Display_Buffer[0] = Remain_Second/10;
038     Second_Display_Buffer[1] = Remain_Second % 10;
039 }
040
041 //-----
042 // 警报声输出函数
043 //-----
044 void Alarm()
045 {
046     INT8U i;
047     for (i = 0 ; i < 80; i++)
048     {
049         BEEP(); _delay_us(300);
050     }
051 }
052
053 //-----
054 // 主程序
055 //-----
056 int main()
057 {
058     Current_Direct = EW; //初始通行方向设为东西方向
059     Remain_Second = MAX_SECOND; //初始剩余秒数为最大秒数
060     Refresh_Second_Display_Buffer(); //刷新秒显示缓冲
061
062     TCCR1B = 0x03; //T1 预设分频:64
063     TCNT1 = 65536 - F_CPU/64.0 * 1.0; //晶振 4 MHz, 1 s 定时初值
064     TCCR2 = 0x04; //T2 预设分频:64
065     TCNT2 = 256 - F_CPU/64.0 * 0.004; //晶振 4 MHz, 0.004 s 定时初值
066     TIMSK = _BV(TOIE1) | _BV(TOIE2); //允许 T1、T2 定时器溢出中断
067     DDRB = 0xFF; //配置输出端口
```

```

068     DDRC = 0xFF;
069     DDRD = 0xFF;
070     sei();                                //开中断
071     while (1);
072 }
073
074 //-----
075 // T1 定时器溢出中断程序,控制倒计时
076 //-----
077 ISR (TIMER1_OVF_vect)
078 {
079     //重装 1 s 定时初值
080     TCNT1 = 65536 - F_CPU/64.0 * 1.0;    "
081     //计时值递减,递减到终点后重新从最大秒数 MAX_SECOND 开始,
082     //同时切换通行方向
083     if (--Remain_Second == -1 )
084     {
085         Remain_Second = MAX_SECOND;
086         Current_Direct = Current_Direct == EW ? SN : EW;
087     }
088     //刷新秒显示缓冲
089     Refresh_Second_Display_Buffer();
090     //剩余时间在 5 s 以内时输出声音
091     if (Remain_Second <= 5) Alarm();
092 }
093
094 //-----
095 // T2 定时器溢出中断程序,控制数码管扫描显示
096 //-----
097 ISR (TIMER2_OVF_vect)
098 {
099     //当前待显示位索引(注意设为静态变量)
100     static INT8U i = 0;
101     //位间延时 4 ms
102     TCNT2 = 256 - F_CPU/64.0 * 0.004;
103     //先关闭段码
104     PORTC = 0xFF;
105     //输出数码管段码
106     PORTC = SEG_CODE[ Second_Display_Buffer[ i ] ];
107     //根据当前方向输出(0,1)位或(2,3)位的扫描码
108     if (Current_Direct == EW)
109         PORTD = _BV(i);
110     else

```



```
111     PORTD = _BV(i + 2);  
112     //i 在 0,1 之间切换(扫描输出的数位将是 0,1 或 2,3)  
113     i = (i == 0) ? 1 : 0;  
114 }
```

3.21 用工作于计数方式的 T/C0 实现 100 以内的脉冲或按键计数

本例 T/C0 时钟由 T0(PB0)引脚输入。利用这一特点,本例实现了按键和脉冲计数及显示功能,学习调试要点在于掌握 T/C0 工作于计数方式的程序设计方法。案例电路及部分运行效果如图 3-21 所示。

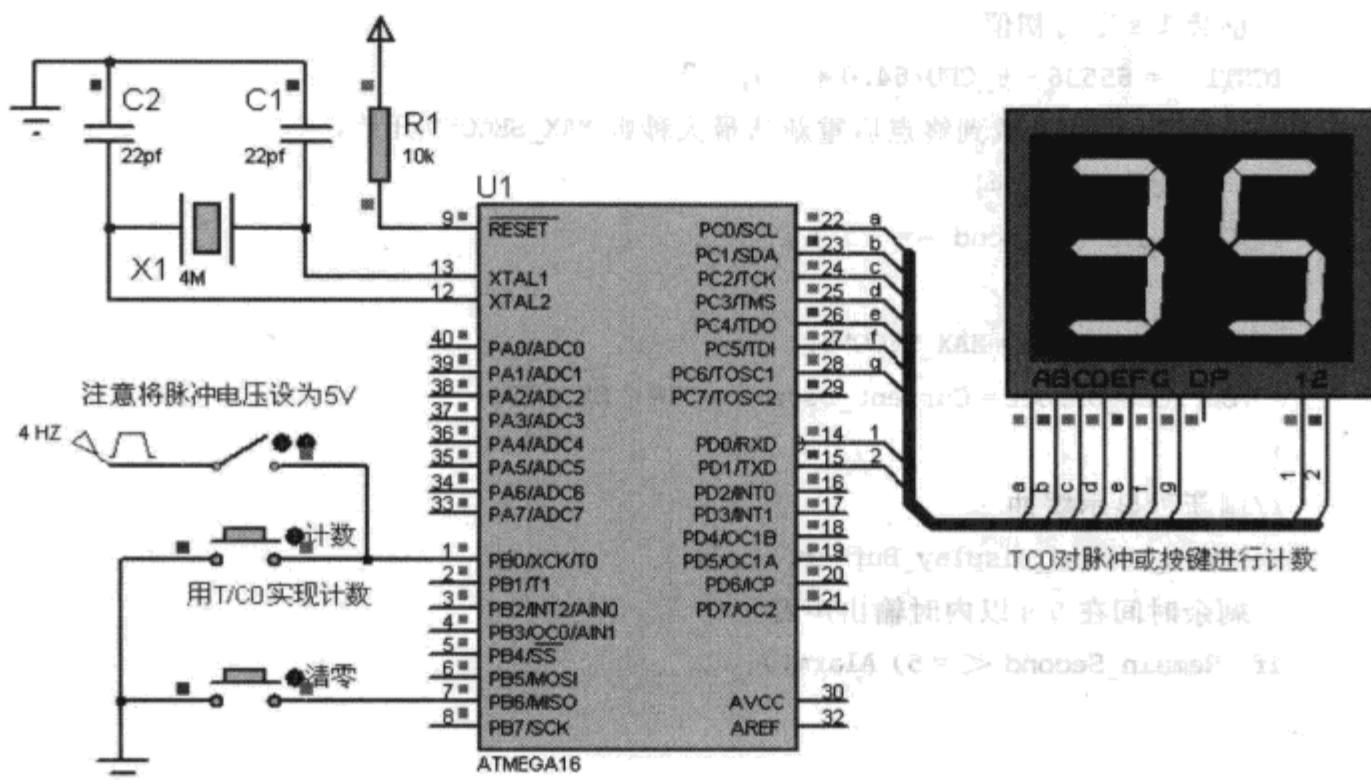


图 3-21 用工作于计数方式的 T/C0 实现 100 以内的脉冲或按键计数

1. 程序设计与调试

此前案例中使用的 T/C0、T/C1、T/C2 均工作于定时方式,本例中 T/C0 工作于计数方式,其区别在于此前 TCNTx 的计数时钟由系统时钟分频后提供。所提供的时钟具有固定频率(周期),因而 TCNTx 的计数可换算成计时。

本例中的 T/C0 工作于计数方式,TCNT0 的计数时钟不再由系统时钟分频提供,而是由来自 T0(PB0)引脚的外部信号提供,这些信号可能是无固定周期的(例如本例的按键计数操作),也可能是有固定周期的(例如本例中的外部输入计数脉冲)。

主程序中 TCCR0=0x06,该行代码将 TCCR0 的低 3 位(CS02、CS01、CS00)设为 110,它使得 TCNT0 的计数时钟来自于 T0(PB0)引脚,且为下降沿触发。主程序中设置 TCCR0 后,T0(PB0)引脚每次由高电平到低电平的跳变都将使 TCNT0 递增 1。

主程序中的 while 循环完成了清零控制,显示上限控制,数码管刷新显示控制等操作。

2. 实训要求

- ① 利用 T/C0 计数溢出中断实现计数,每次按键或脉冲输入信号的下降沿使计数变量累

加, 程序中注意要将 TCNT0 初值设为 255, 这样才能使得每次下降沿累加 TCNT0 都会导致溢出, 触发 T/C0 计数溢出中断, 在中断程序中完成对计数变量的累加。显然, 如果将 TCNT0 设为 254 的话, 每 2 次脉冲或按键输入才会实现 1 次计数变量累加。

② 改用 4 位集成式数码管, 实现更大范围的计数操作。

3. 源程序代码

```

01 //-----
02 // 名称: 用工作于计数方式的 TCO 实现 100 以内的脉冲或按键计数
03 //-----
04 // 说明: TCO 工作于计数器方式且设为下降沿触发, 外部的每次按键或脉冲
05 //       出现的下降沿都将导致 TCNT0 累加计数一次, TCNT0 的计数值将被实时
06 //       刷新并显示在两位数码管上
07 //
08 //-----
09 #define F_CPU 4000000UL           //4 MHz 晶振
10 #include <avr/io.h>
11 #include <util/delay.h>
12 #define INT8U unsigned char
13 #define INT16U unsigned int
14
15 //清除键定义
16 #define Clear_Key_DOWN() ((PINB & 0x40) == 0x00)
17
18 //0~9 的数字编码
19 const INT8U SEG_CODE[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};
20 //-----
21 // 在两位数码管上显示计数值
22 //-----
23 void Show_Count_ON_DSY()
24 {
25     PORTD = 0xFF;
26     PORTC = SEG_CODE[TCNT0/10];
27     PORTD = 0xFE;
28     _delay_ms(2);
29     PORTD = 0xFF;
30     PORTC = SEG_CODE[TCNT0 % 10];
31     PORTD = 0xFD;
32     _delay_ms(2);
33 }
34
35 //-----
36 // 主程序

```



```
37 //-----  
38 int main()  
39 {  
40     DDRC = 0xFF; PORTD = 0xFF;           //配置输出端口  
41     DDRD = 0xFF; PORTD = 0xFF;  
42     DDRB = 0x00; PORTB = 0xFF;           //配置输入端口  
43  
44     TCCR0 = 0x06;                      //T0 工作于计数方式,下降沿触发  
45     TCNT0 = 0x00;                      //设置计数初值  
46     while(1)  
47     {  
48         if(Clear_Key_DOWN()) TCNT0 = 0; //如果按下清零键则将 TCNT0 重新置 0  
49         if (TCNT0 >= 100) TCNT0 = 0;    //计数值限制在 100 以内  
50         Show_Count_ON_DSY();       //持续刷新显示  
51     }  
52 }
```

3.22 用定时器设计的门铃

本例用定时器控制蜂鸣器模拟发出“叮咚”的门铃声，其中“叮”的声音用较短定时形成较高频率，“咚”的声音用较长定时形成较低频率。仿真电路中加入了虚拟示波器，按下按键时除听到门铃声外，还会从示波器中观察到 2 种不同频率的波形。本例电路及输出的部分波形如图 3-22 所示。

1. 程序设计与调试

主程序控制变量 soundDelay 分别取值 -700 与 -1 000，它影响定时器溢出中断程序中 TCNT1 的取值，从而控制输出 2 种不同频率的声音。

定时器初始定时为 $700 \mu\text{s}$ ，按键使能定时器溢出中断后，在前 300 ms 时间内的每次中断触发时间间隔都是 0.7 ms(初值为 -700)，这使得 DoorBell() 可以输出 $1000/(0.7 \times 2) \approx 714 \text{ Hz}$ 的声音频率，在后 500 ms 内的每次中断触发时间间隔为 1 ms(初值为 -1 000)，DoorBell() 输出 $1000/(1.0 \times 2) = 500 \text{ Hz}$ 声音频率。按下按键后的“叮咚”声正是这样产生的。

在一次“叮咚”声音输出完成后，TMISK = 0x00 使定时器溢出中断被禁止，声音输出也随即停止。

设置 T/C1 定时器初值时所使用的公式是：

$$\text{TCNT1} = 65536 - F_{\text{CPU}}/\text{分频} \times \text{定时长度(s)}$$

本例最初定时长度设为 $700 \mu\text{s}$ ，即 0.0007 s，代入公式可得： $\text{TCNT1} = 65536 - 1000000/1 \times 0.0007 = 65536 - 700$ 。

对于语句 $\text{TCNT1} = 65536 - 700$ ，在编写程序时还可直接写成 $\text{TCNT1} = -700$ ，因为 65536 即 17 位二进制数 10000000000000000，其最高位为 1，其余 16 位为 0。对于 16 位的寄存器，65536 与 0 是相等的，因此有 $\text{TCNT1} = 0 - 700 = -700$ ，实际存入 TCNT1 是 -700 的补

码,将 700 进行二进制数分解可得 0000001010111100,将其取反加 1 得到 1111110101000100,即 64836,而 $65536 - 700$ 也等于 64836。

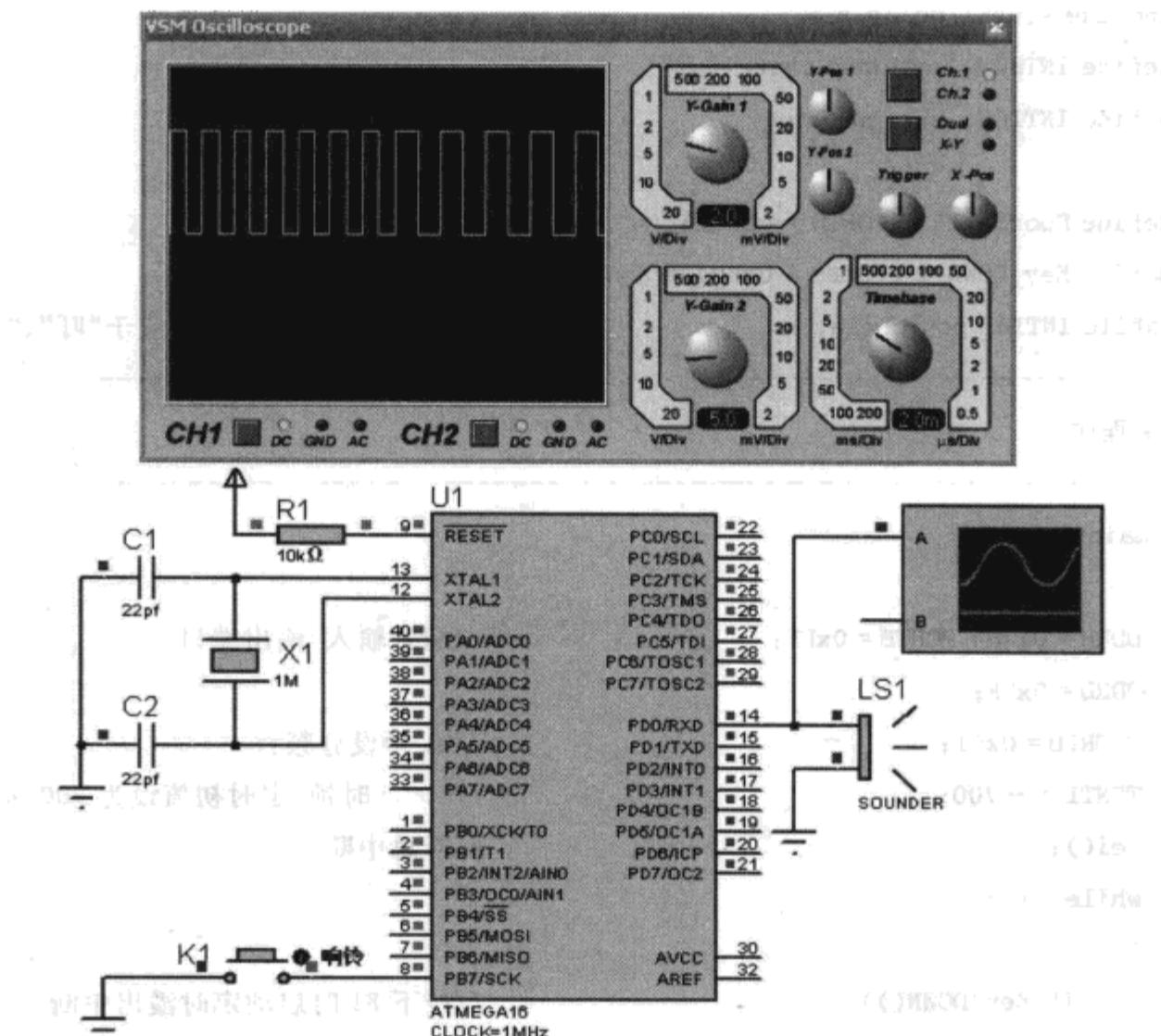


图 3-22 用定时器设计的门铃

可见,对于 16 位的 T/C1 定时器,在 1 MHz 时钟频率、分频为 1 时,要实现 x 微秒(μs)的延时,可直接使用语句 $TCNT1 = -x$,当然, x 的取值应在 0~65535 范围之内。同样,对于 8 位的 T/C0 与 T/C2 定时器,在同样时钟与分频比下,设置延时初值时也可以有 $TCNT0 = -x$ 和 $TCNT2 = -x$ 。当然,这里 x 取值应在 0~255 范围之内。

2. 实训要求

- ① 修改定时器初值—700 和—1000,并改变 2 种频率声音的输出时长,再观察输出效果。
- ② 重新编写程序,用定时器控制输出另一种包含 3 个不同频率的门铃声音效果。

3. 源程序代码

```

01 //-----
02 // 名称: 用定时器设计的门铃
03 //-----
04 // 说明: 按下按键时蜂鸣器发出叮咚的门铃声
05 //
06 //-----
07 #define F_CPU 1000000UL           //1 MHz 晶振

```

```

08 # include <avr/io.h>
09 # include <avr/interrupt.h>
10 # include <util/delay.h>
11 # define INT8U unsigned char
12 # define INT16U unsigned int
13
14 # define DoorBell() (PORTD ^= 0x01)          //门铃定义
15 # define Key_DOWN() ((PINB & 0x80) == 0x00)    //按键定义
16 volatile INT16U soundDelay;                  //2个不同取值分别对应于“叮”、“咚”
17 //-----
18 // 主程序
19 //-----
20 int main()
21 {
22     DDRB = 0x00; PORTB = 0xFF;                //配置输入/输出端口
23     DDRD = 0xFF;
24     TCCR1B = 0x01;                           //T1 预设分频:1
25     TCNT1 = -700;                            //1 MHz 时钟,定时初值设为 700 μs
26     sei();                                  //开总中断
27     while(1)
28     {
29         if(Key_DOWN())                      //按下 K1 时启动定时溢出中断
30         {
31             TIMSK = _BV(TOIE1);            //允许 T1 定时器溢出中断
32             soundDelay = -700;           //700 μs 延时对应输出“叮”的声音
33             _delay_ms(400);             //声音保持 400 ms
34             soundDelay = -1000;          //1000 μs 延时对应输出“咚”的声音
35             _delay_ms(600);             //声音保持 600 ms
36             TIMSK = 0x00;              //禁止 T1 定时器溢出中断
37         }
38     }
39 }
40
41 //-----
42 // T/C1 定时器中断程序控制门铃声音输出
43 //-----
44 ISR(TIMER1_OVF_vect )
45 {
46     DoorBell();                            //门铃声音输出
47     TCNT1 = soundDelay;                  //按主程序设定的 -700 或 -1000 设置延时
48 }

```

3.23 报警器与旋转灯

本例运行时,按下报警开启按键后系统将发出逼真的警报声音,连接 PC 端口的 LED 中相邻的 3 只将随之不断循环滚动点亮,按下关闭键时警报输出停止,所有 LED 熄灭。本例同时启用了 3 个中断程序。案例电路及部分运行效果如图 3-23 所示。

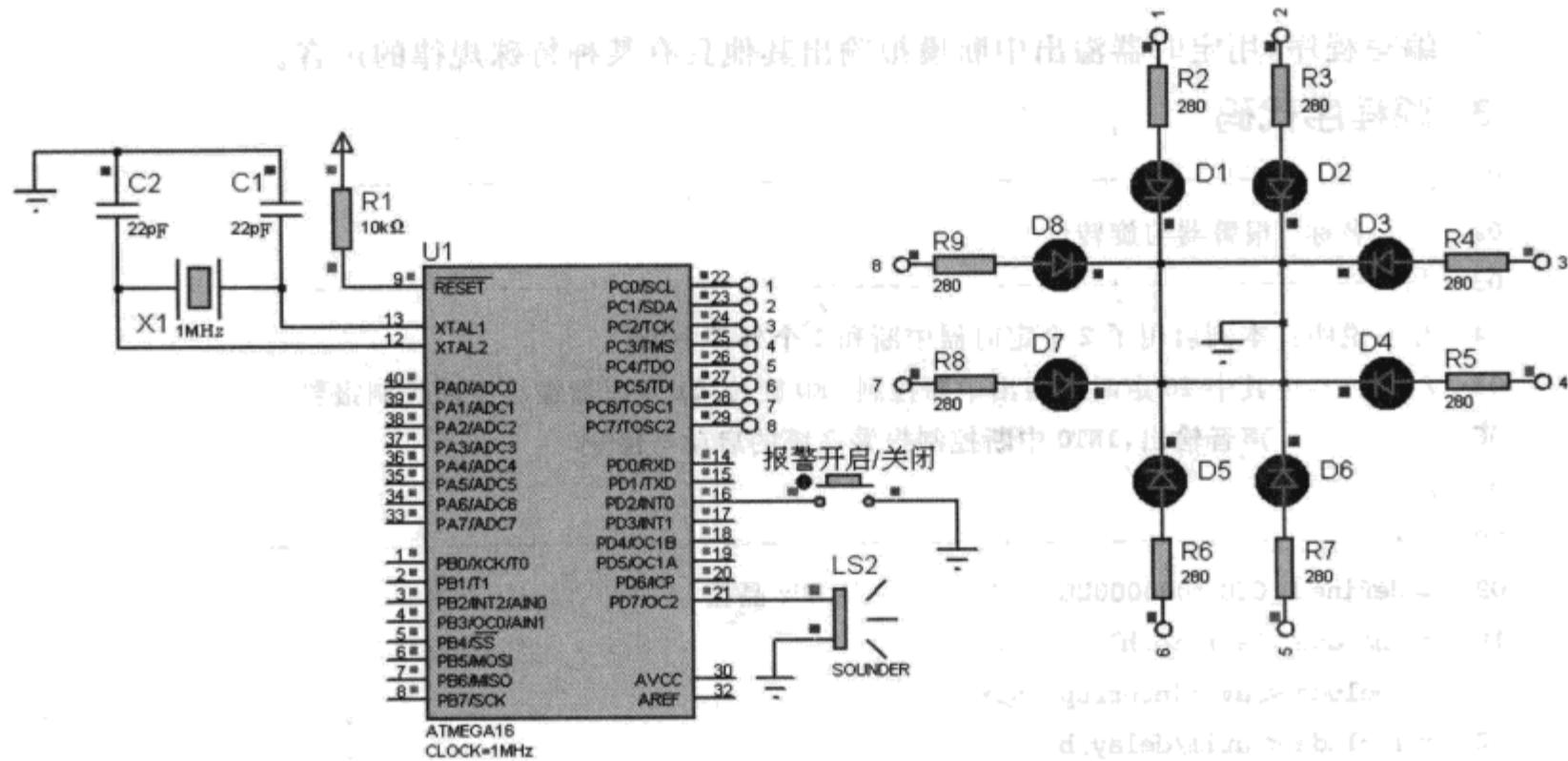


图 3-23 报警器与旋转灯

1. 程序设计与调试

本例同时启用了 INT0 中断、T/C0 溢出中断、T/C1 溢出中断,它们分别负责系统启停控制、LED 旋转滚动显示与警报声音输出。本例的报警声音非常逼真,它模拟了声音频率均匀拉高、还原、再拉高的过程。

本例中 16 位的 T/C1 定时/计数寄存器 $TCNT1=0xFE00+FRQ$,其中 FRQ 由主程序控制在 $0x00\sim0xFF(0\sim255)$ 之间反复递增循环取值,由于 FRQ 的变化由 main 函数控制,T/C1 溢出中断程序中需要使用该变量,因此要注意在定义 FRQ 时添加 volatile 关键字。

本例程序中 T/C1 溢出中断程序输出的频率范围计算如下:

① TCNT1 计数寄存器取值范围为 $0xFE00\sim0xFEFF$,即 $65024\sim65279$;

② 延时值范围为 $(65536-65024)\sim(65536-65279)$,即 $512\sim257$;

③ 对于 1 MHz 的时钟,其输出频率范围为 $1000000/(512\times2)\sim1000000/(257\times2)Hz$,即 $976\sim1945 Hz$ 。

由于 FRQ 类型为 INT8U,主程序中的 while 循环控制 FRQ 变量由 $0x00$ 持续递增,FRQ 将在 $0x00\sim0xFF$ 范围内反复循环取值,而 T/C1 中断程序内的 TCNT1 寄存器在每次中断触发时都将获取这个不断变化的 FRQ 值,使得 TCNT1 不断由 $0xFE00$ 向 $0xFEFF$ 循环递增,每次中断触发都使下一次的中断触发时间变得更短,T/C1 中断的触发频率越来越高,中断程序输出的…01010101…序列的频率也越来越高,从而形成了 $976\sim1945 Hz$ 频率的平滑递增

输出,案例输出的报警器声音效果非常逼真。

2. 实训要求

- ① 修改主程序中 while 循环内 _delay_ms 函数参数为 1、2、3 或 4 等,重新编译并运行程序,试听输出警报声音的急促程度是否会发生变化。
- ② 第 5 章有与射击游戏相关的程序,参考该游戏程序修改本例代码,模拟出枪支射击的声音。
- ③ 编写程序,用定时器溢出中断模拟输出其他具有某种特殊规律的声音。

3. 源程序代码

```

01 //-----
02 // 名称: 报警器与旋转灯
03 //-----
04 // 说明: 本例启用了 2 个定时器中断和 1 个外部中断
05 //       其中 T0 定时器溢出中断控制 LED 旋转,T1 定时器溢出中断控制报警
06 //       声音输出,INT0 中断控制报警系统的启动与停止
07 //
08 //-----
09 #define F_CPU 1000000UL           //1 MHz 晶振
10 #include <avr/io.h>
11 #include <avr/interrupt.h>
12 #include <util/delay.h>
13 #define INT8U   unsigned char
14 #define INT16U  unsigned int
15
16 //蜂鸣器输出定义
17 #define SPK() (PORTD ^= _BV(PD7))
18
19 volatile INT8U FRQ = 0x00;        //定时初值循环递增控制频率循环递增(volatile 不可省略)
20 INT8U ON_OFF = 0;                //开关变量
21 INT8U Pattern = 0xE0;          //旋转灯端口花样初值 11100000
22 //-----
23 // 主程序
24 //-----
25 int main()
26 {
27     DDRC = 0xFF;                  //配置 LED 输出端口
28     DDRD = ~_BV(PD2); PORTD = _BV(PD2); //配置按键输入与蜂鸣器输出端口
29     TCCR0 = 0x05;                 //T0 预设分频:1024
30     TCNT0 = 256 - F_CPU/1024.0 * 0.1; //1 MHz 时钟,0.1 s 定时初值
31     TCCR1B = 0x01;                //T1 预设分频:1
32     MCUCR = 0x02;                //INT0 为下降沿触发
33     GICR = 0x40;                 //INT0 中断使能

```

```

34     sei();           //开中断
35     while (1)
36     {
37         //定时初值循环递增控制频率循环递增
38         //FRQ 在超过 255 溢出后从 0 开始再继续递增
39         FRQ++;
40         //改变延时参数可调整报警声音输出的急促程度(例如 1、2、3、4)
41         _delay_ms(1);
42     }
43 }
44
45 //-----
46 // 外部中断 0,启停报警器声音和 LED 旋转
47 //-----
48 ISR (INT0_vect)
49 {
50     ON_OFF = !ON_OFF;      //启停切换
51     if(ON_OFF)
52     {
53         TIMSK |= 0x05;    //开启 2 个定时器中断,分别控制报警器和 LED
54         Pattern = 0xE0;   //11100000,开 3 个灯旋转
55     }
56     else
57     {
58         TIMSK = 0x00;      //关闭所有定时器中断
59         PORTC = 0x00;      //关闭所有 LED
60         PORTD &= ~_BV(PD7); //在蜂鸣器连接的 PD7 引脚输出低电平
61     }
62 }
63
64 //-----
65 // T0 定时器中断程序控制 LED 旋转
66 //-----
67 ISR (TMR0_OVF_vect)
68 {
69     //重装 0.1 s 计时初值
70     TCNT0 = 256 - F_CPU/1024.0 * 0.1;
71     //以下两行实现 111 的循环左移(高位为 1 时左移后右端补 1,否则直接左移)
72     if (Pattern & 0x80) Pattern = (Pattern << 1) | 0x01;
73     else                  Pattern <<= 1;
74     PORTC = Pattern;      //LED 显示
75 }
76

```



```
77 //-----  
78 // T1 定时器中断控制报警器声音输出  
79 //-----  
80 ISR (TIMER1_OVF_vect)  
81 {  
82     TCNT1 = 0xFE00 + FRQ;      //主程序中 FRQ 的递增导致输出频率递增  
83     SPK();  
84 }
```

3.24 100 000 s 以内的计时程序

本例程序运行时,首次按下 K1 即开始启动精度为 0.1 s 的计时,计时刷新显示在 6 位数码管上,再次按下 K1 时暂停计时,当前计时值保持显示在数码管上,第三次按下 K1 时计时值归 0。本例最大计时为 99 999.9 s。案例电路及部分运行效果如图 3-24 所示。

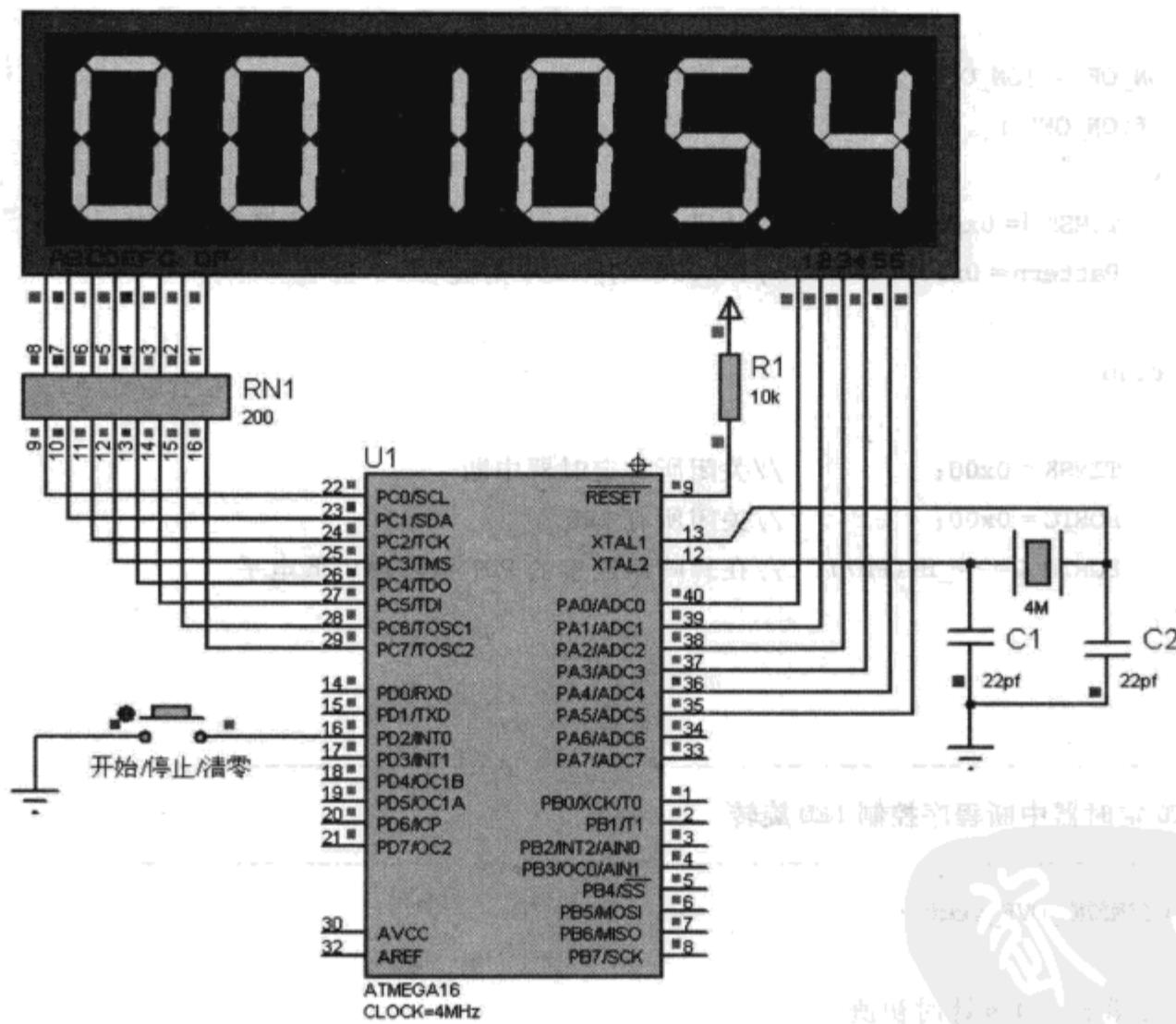


图 3-24 100 000 s 以内的计时程序

1. 程序设计与调试

本例设计与调试要点如下:

- ① 本例计数(计时)方法与此前案例不同,此前案例的计数值一般是保存在全局变量中,或者是保存在计数寄存器内,需要显示在数码管上时,再将待显示数据分解为多个数位。本例

直接定义了含 6 个元素的数组 Digits_Buffer, 定时中断子程序每 0.1 s 对 Digits_Buffer [0] 累加, 累加到 10 时即进位到下一元素, 以此类推。这样设计后的计数值在显示时不需要再用整除 (/) 和取余 (%) 进行分解。

② 对于同一按键上实现的 3 种操作, 程序中使用变量 KeyOperation 进行标识, 每次出现按键中断时递增 KeyOperation, 根据不同的 KeyOperation 值完成不同的操作。变量 KeyOperation 可以定义为全局变量, 也可以在函数内部定义为静态变量, 本例使用的是后一种定义方式, 这样可读性更好一些。注意: 编写调试程序不可忽略了 static 关键字。

③ 由于主程序中第 47 行已设置 TIMSK 允许定时器溢出中断, 按键中断函数中第 89 行只需要设置 TCCR1B 为非 0 的分频比(本例设为 8 分频)即可开始启动计时并每隔 0.1 s 触发中断。停止或清零计时时, 第 91 行也需要将 TCCR1B 设置为 0 分频(无时钟, T/C1 不工作)即可, 虽然 TIMSK 仍然允许定时器溢出中断, 但由于 T/C1 已无时钟, 溢出中断也亦不会发生, 0.1 s 的计时亦停止。

④ 案例使用了 6 位集成式七段数码管, 在完成 0.1 s 精度计时的同时完成数码管的刷新显示, 并实现对小数点的显示控制。由于 Digits_Buffer [0]~Digits_Buffer [5] 分别存放的依次是小数位、个位、十位一直到最高位, 在 6 位数码管上显示该数组时, 循环控制变量 i=0~5, 如果用 PORTC=~_BV(i) 来输出位码, 这会将小数位显示在数码管最左边, 而最高位却显示在最右边, 因此输出位码的语句应为 PORTC=~_BV(5-i), 凡是共阴数码管均需要添加“~”来输出位码, 而_BV 的参数 5-i 显然是起到了将顺序读取的 0~5 号数组元素在数码管上逆向显示的作用。

2. 实训要求

① 修改程序, 将计时精度设为 0.01 s, 并重新进行调试与仿真运行。

② 在本例中实现两段计时, 第一次暂停后再次按下 K1 时, 可在前一段时间的基础上再继续计时, 最后按下 K1 时才将计时清零。

3. 源程序代码

```

01 //-----
02 // 名称: 100000 s 以内的计时程序
03 //-----
04 // 说明: 在 6 只数码管上完成 00000.0~99999.9 s 计时
05 //
06 //-----
07 #define F_CPU 4000000UL           //4 MHz 晶振
08 #include <avr/io.h>
09 #include <avr/interrupt.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //共阴数码管 0~9 的数字段码
15 const INT8U SEG_CODE[] =

```



```
16 { 0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F };
```

```
17
```

```
18 //6 只数码管上显示的数字缓冲
```

```
19 INT8U Digits_Buffer[] = {0,0,0,0,0,0};
```

```
20 //-----
```

```
21 // 主程序
```

```
22 //-----
```

```
23 void Show_Count_ON_DSY()
```

```
24 {
```

```
25     INT8U i;
```

```
26     for (i = 0; i <= 5; i++)
```

```
27     {
```

```
28         PORTC = 0x00;                      //暂时关闭段码
```

```
29         PORTA = ~_BV(5 - i);              //输出位码
```

```
30         PORTC = SEG_CODE[ Digits_Buffer[i] ]; //输出段码
```

```
31         if (i == 1) PORTC |= 0x80;          //在个位数上加小数点
```

```
32         _delay_ms(3);                   //位间延时
```

```
33     }
```

```
34 }
```

```
35
```

```
36 //-----
```

```
37 // 主程序
```

```
38 //-----
```

```
39 int main()
```

```
40 {
```

```
41     DDRA = 0xFF; PORTA = 0xFF;           //配置端口
```

```
42     DDRC = 0xFF; PORTC = 0xFF;
```

```
43     DDRD = 0x00; PORTD = 0xFF;
```

```
44     MCUCR = 0x02;                      //INT0 为下降沿触发
```

```
45     GICR = 0x40;                       //INT0 中断使能
```

```
46     TCNT1 = 65536 - F_CPU/8 * 0.1;    //0.1 s 定时(T1 预设 8 分频在 INT0 中断中完成)
```

```
47     TIMSK = _BV(TOIE1);               //允许 T1 定时器溢出中断
```

```
48     sei();                            //开中断
```

```
49     while(1) Show_Count_ON_DSY();      //持续刷新显示
```

```
50 }
```

```
51
```

```
52 //-----
```

```
53 // T1 定时器溢出中断实现计时
```

```
54 //-----
```

```
55 ISR (TIMER1_OVF_vect )
```

```
56 {
```

```
57     INT8U i;
```

```

58     TCNT1 = 65536 - F_CPU/8 * 0.1;           //重设 0.1 s 定时初值
59     Digits_Buffer[0]++;                      //0.1 s 位累加
60     for (i = 0; i <= 5; i++)                 //进位处理
61     {
62         if(Digits_Buffer[i] == 10)
63         {
64             Digits_Buffer[i] = 0;
65             //如果是 0~4 位则分别向高一位进位
66             if(i != 5) Digits_Buffer[i + 1]++;
67         }
68         //循环过程中如果某个低位没有进位，则循环可提前结束
69         else break;
70     }
71 }
72
73 //-----
74 // INT0 中断函数完成 K1 按键的 3 种操作
75 //-----
76 ISR (INT0_vect)
77 {
78     INT8U i;
79     //按键操作标识:0 停止, 1 开始, 2 暂停
80     static INT8U KeyOperation = 0;
81     //每次按键时,操作标识在 0,1,2 中循环选择
82     if (++KeyOperation == 3) KeyOperation = 0;
83
84     switch (KeyOperation)
85     {
86         case 0: TCCR1B = 0x00;                  //停止(清零)
87             for (i = 0; i < 6; i++) Digits_Buffer[i] = 0;
88             break;
89         case 1: TCCR1B = 0x02;                  //开始计时(提供 8 分频时钟)
90             break;
91         case 2: TCCR1B = 0x00;                  //暂停计时
92             break;
93     }
94 }

```

3.25 用 TIMER1 输入捕获功能设计的频率计

本例 T/C1 工作于一般模式下的定时器方式, 外部被测信号从 ICP(PD6)输入, 程序利用 T/C1 的输入捕获(Input Capture)功能, 在按下 K1 按键时, 通过检测 2 次捕获的计数差值计



算出被测信号频率，并显示在 4 位数码管上。本例电路及部分运行效果如图 3-25 所示。

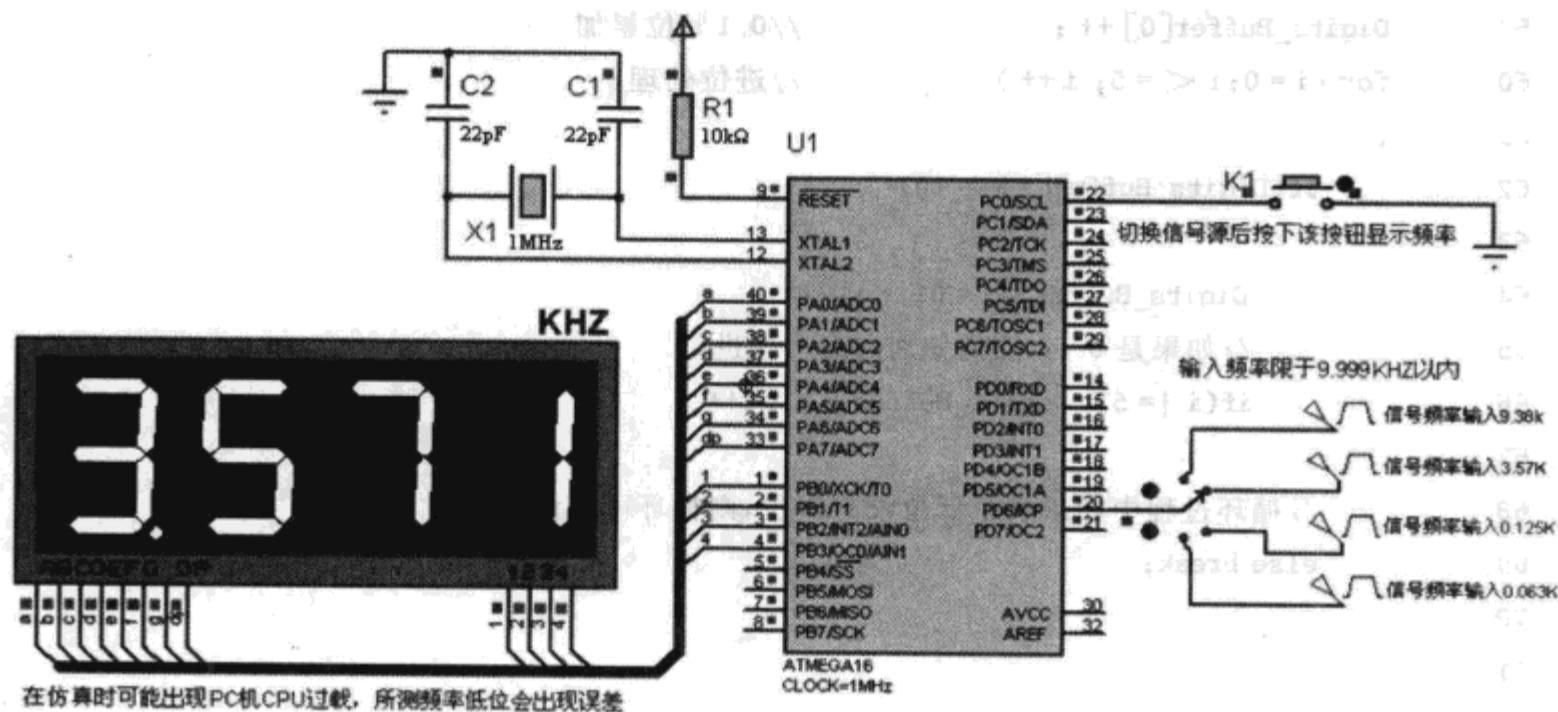


图 3-25 用 TIMER1 输入捕获功能设计的频率计

1. 程序设计与调试

本例 T/C1 工作于定时器方式，TCNT1 计数脉冲由系统时钟分频后提供，本例系统时钟为 1 MHz，TCCR1B 设置分频比为 1，计数时钟仍为 1 MHz，在该计数时钟下，每 1 μs 时间 TCNT1 计数 1 次。

主程序通过设置 TCCR1B = _BV(ICNC1) | _BV(ICES1)，设置了输入捕获噪音消除 (ICNC1=1) 位和 ICP 上升沿触发输入捕获 (ICES1=1) 位。由于 PD6 (ICP) 输入捕获引脚输入信号的每次上升沿都将触发捕获，在捕获发生时，当前计数寄存器 TCNT1 的计数值被复制到输入捕获寄存器 ICR1 中；在连续 2 次 ICP 引脚上升沿触发捕获中断时，中断服务程序通过计算 2 次所读取的 ICR1 的差值，即可得出相邻 2 次 TCNT1 的计数差值。本例的输入捕获中断向量名称为 TIMER1_CAPT_vect。

在本例配置下，TCNT1 的每次计数为 1 μs，如果差值为 8，则该输入信号周期为 8 μs，倒数处理后即可得到信号频率。

因本例仿真电路中放入了多路外部信号源，Proteus 仿真时可能会提示 PC 机 CPU 过载，仿真未能在实时模式下运行，这会影响所检测频率的精度。在运行本例检测外部信号频率时，所显示出来的频率低位数可能会出现误差。

2. 实训要求

① 进一步改进本例，使每次按下 K1 按键后能重复进行 6 次捕获，求得 3 个捕获差值后计算平均频率，以提高系统检测结果的精度。

② 设置 T/C1 工作于计数方式，重新编写本例程序实现频率检测。

3. 源程序代码

```
01 //-----  
02 // 名称：用 TIMER1 输入捕获功能设计的频率计  
03 //-----
```

```

04 // 说明：本例运行时，切换不同的频率输入，然后按下 K1 按键，数码管上将
05 // 显示当前频率值。2 次捕获的时间差值即为当前输入频率的周期，
06 // 周期倒数即可得到当前频率
07 //
08 //-----
09 #define F_CPU 1000000UL //1 MHz 晶振
10 #include <avr/io.h>
11 #include <avr/interrupt.h>
12 #include <util/delay.h>
13 #define INT8U unsigned char
14 #define INT16U unsigned int
15
16 //共阴数码管 0~9 的数字编码，最后一位为黑屏
17 const INT8U SEG_CODE[] =
18 {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00};
19
20 //分解后的待显示数位
21 INT8U Display_Buffer[] = {0,0,0,0};
22
23 //连续 2 次捕获计数变量
24 INT16U CAPi = 0,CAPj = 0;
25 //-----
26 // 数码管显示频率
27 //-----
28 void Show_FRQ_ON_DSY()
29 {
30     INT8U i = 0;
31     for (i = 0; i<4; i++)
32     {
33         PORTA = 0x00; //先暂时关闭段码
34         PORTB = ~_BV(i); //发送扫描码
35         PORTA = SEG_CODE[ Display_Buffer[i] ]; //发送数字段码
36         if (i == 0) PORTA |= 0x80; //最高位加小数点
37         _delay_ms(2);
38     }
39 }
40
41 //-----
42 // 主程序
43 //-----
44 int main()
45 {
46     INT8U LastKey = 0xFF; //最近按键状态

```



```
47     DDRA = 0xFF;                                //配置输出端口
48     DDRB = 0xFF;
49     DDRC = 0x00; PORTC = 0xFF;
50     DDRD = 0x00; PORTD = 0xFF;
51     //输入捕获噪音消除, ICP 上升沿触发输入捕获,
52     //分频系数:1(1 MHz, 每 1 μs 计数 1 次)
53     TCCR1B = _BV(ICNC1) | _BV(ICES1);           //初始时无分频,按下 K1 后提供 1 分频
54     sei();                                       //开中断
55     while(1)
56     {
57         if(LastKey != PINC)                      //PC 端口有键按下(K1 按下)
58         {
59             TIMSK = _BV(TICIE1);                 //使能 TC1 输入捕获中断
60             TCCR1B |= 0x01;                      //提供 1 分频计数时钟
61             LastKey = PINC;                     //保存最近按键状态
62         }
63         Show_FRQ_ON_DSY();                   //数码管显示频率
64     }
65 }
66
67 //-----
68 // T1 输入捕获中断子程序
69 //-----
70 ISR (TIMER1_CAPT_vect)
71 {
72     INT8U i;
73     if (CAPi == 0) CAPi = ICR1;                //第 1 次捕获
74     else                                         //第 2 次捕获
75     {
76         CAPj = ICR1 - CAPi;                    //2 次相减得到周期(μs)
77         CAPj = 1000000UL/CAPj;                 //周期倒数后乘以 1000 000 得到频率
78         TIMSK = 0x00;                          //第 2 次捕获后禁止输入捕获中断
79         TCCR1B &= 0xFC;                        //关闭计数时钟
80         for (i = 3; i != 0xFF; i--)
81         {
82             Display_Buffer[i] = CAPj % 10;
83             CAPj /= 10;
84         }
85         TCNT1 = CAPi = CAPj = 0;              //相关变量和寄存器清零
86     }
87 }
```

3.26 用工作于异步模式的 T/C2 控制的可调式数码管电子钟

本例所使用的 T/C2 定时器工作于异步模式,由 PB6(TOSC1)与 PB7(TOSC1)外接 32768 Hz 晶振提供时钟,利用晶振提供的钟表时钟,本例设计了可调式数码管电子钟。案例电路及部分运行效果如图 3-26 所示。

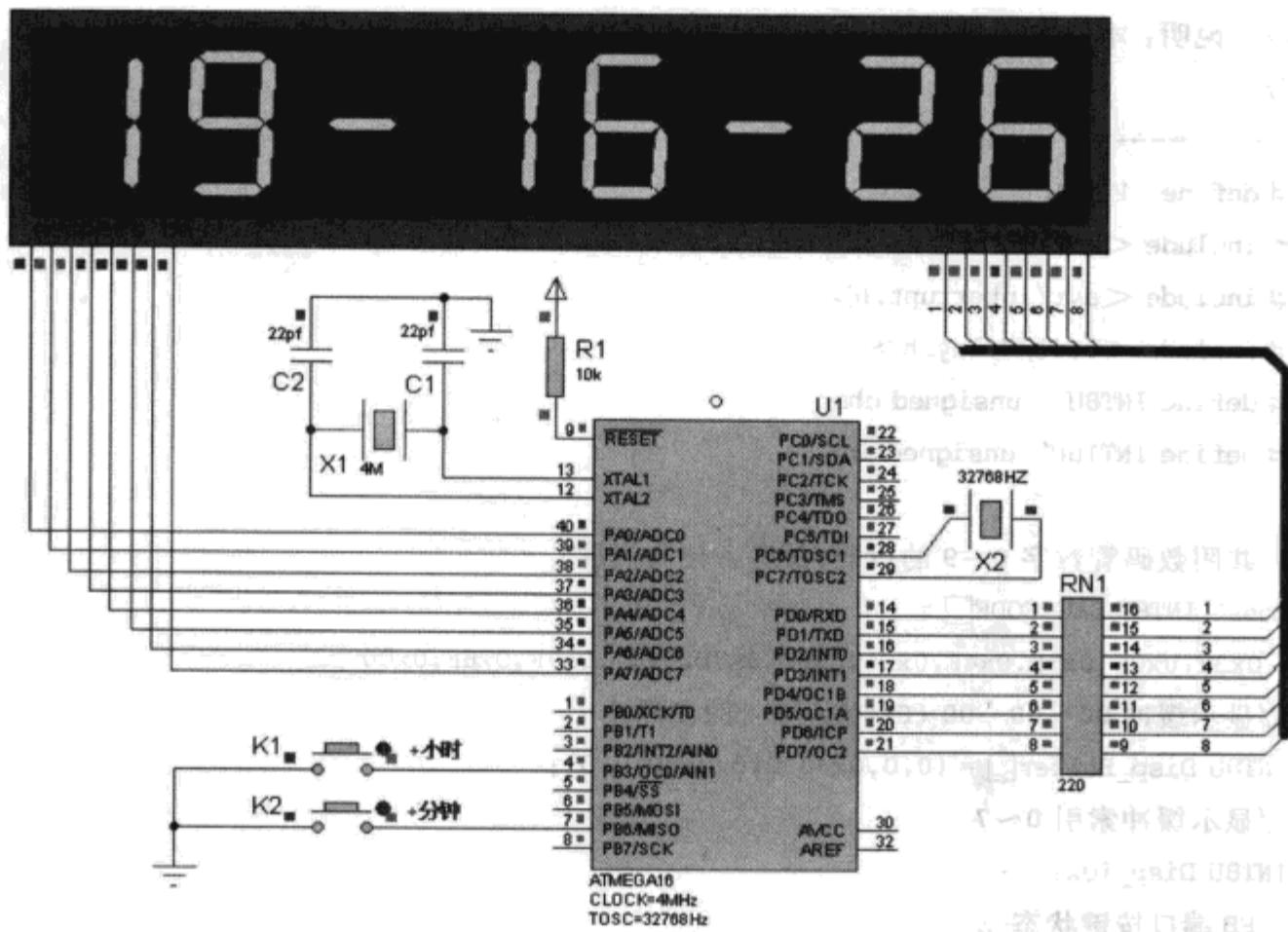


图 3-26 用工作于异步模式的 T2 控制的可调式数码管电子钟

1. 程序设计与调试

对于 T/C2 定时/计数器,通过设置异步状态寄存器 ASSR 可选择 T/C2 的时钟源,程序中 72~74 行对 T/C2 进行了相应设置,ASSR 寄存器中的 AS2 位是 T/C2 的时钟选择位,其中 72 与 73 行:

```
ASSR |= _BV(AS2);      // 用于选择外部时钟
TCCR2 = 0x04;           // 将分频系数设为 64
```

由这两行设置可得 T/C2 计数时钟频率为 $32768 \text{ Hz} / 64 = 512 \text{ Hz}$ 。

第 74 行将 8 位定时/计数器 T/C2 的计数寄存器 TCNT2 初值设为 0,计数 256 次后溢出,在 512 Hz 时钟频率下耗时 0.5 s,T/C2 溢出中断程序将每隔 0.5 s 被调用,中断程序利用 0.5 s 实现时分秒分隔标志“-”的闪烁显示,在每遇到第 2 个 0.5 s 时递增秒数,并进行秒分时的进位处理。

本例数码管的刷新显示则由 T/C0 定时器每隔 4 ms 刷新完成。

2. 实训要求

- 修改电路,使用 3 组 2 位集成数码管,分别显示时分秒,每组数码管之间用 2 位 LED



实现闪烁。

② 将 T/C2 分频系数改为 32,重新修改代码实现本例功能。

3. 源程序代码

```
001 //-----  
002 // 名称：用工作于异步模式的 T2 控制的可调式数码管电子钟  
003 //-----  
004 // 说明：本例 T2 使用外部 32768 Hz 时钟，K1、K2 分别用来调整小时和分钟  
005 //-----  
006 //-----  
007 #define F_CPU 4000000UL //1 MHz 晶振  
008 #include <avr/io.h>  
009 #include <avr/interrupt.h>  
010 #include <util/delay.h>  
011 #define INT8U unsigned char  
012 #define INT16U unsigned int  
013  
014 //共阴数码管数字 0~9 的段码(最后一位为黑屏)  
015 const INT8U SEG_CODE[] =  
016 { 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F, 0x00 } ;  
017 //显示缓冲 00 - 00 - 00 (0x40 为"-"的段码)  
018 INT8U Disp_Buffer[] = { 0, 0, 0x40, 0, 0, 0x40, 0, 0 } ;  
019 //显示缓冲索引 0~7  
020 INT8U Disp_Idx;  
021 //PB 端口按键状态  
022 INT8U Key_State = 0xFF;  
023 //时分秒  
024 INT8U h,m,s;  
025 //-----  
026 // 小时处理函数  
027 //-----  
028 void Increase_Hour()  
029 {  
030     if( ++ h > 23 ) h = 0;  
031     Disp_Buffer[0] = SEG_CODE[h/10];  
032     Disp_Buffer[1] = SEG_CODE[h % 10];  
033 }  
034  
035 //-----  
036 // 分钟处理函数  
037 //-----  
038 void Increase_Minute()  
039 {
```

```

040     if( ++m > 59)
041     {
042         m = 0; Increase_Hour();
043     }
044     Disp_Buffer[3] = SEG_CODE[m/10];
045     Disp_Buffer[4] = SEG_CODE[m % 10];
046 }
047
048 //-----
049 // 秒处理函数
050 //-----
051 void Increase_Second()
052 {
053     if( ++s > 59)
054     {
055         s = 0; Increase_Minute();
056     }
057     Disp_Buffer[6] = SEG_CODE[s/10];
058     Disp_Buffer[7] = SEG_CODE[s % 10];
059 }
060
061 //-----
062 // 主程序
063 //-----
064 int main()
065 {
066     DDRA = 0xFF; PORTA = 0xFF;           //配置端口
067     DDRD = 0xFF; PORTD = 0xFF;
068     DDRB = 0x00; PORTB = 0xFF;
069
070     TCCR0 = 0x03;                      //预设分频:64
071     TCNT0 = 256 - F_CPU/64.0 * 0.004; //晶振4 MHz,4 ms定时初值
072     ASSR = 0x08;                      //异步时钟使能
073     TCCR2 = 0x04;                      //预设分频:64,32768 Hz/64 = 512 Hz
074     TCNT2 = 0;                         //T2计时初值
075     TIMSK = _BV(TOIE2) | _BV(TOIE0); //允许T0、T2定时器中断
076
077     h = 12;    m = s = 0;
078     //将初始时分秒段码放入显示缓冲
079     Disp_Buffer[0] = SEG_CODE[h/10];
080     Disp_Buffer[1] = SEG_CODE[h % 10];
081     Disp_Buffer[3] = SEG_CODE[m/10];
082     Disp_Buffer[4] = SEG_CODE[m % 10];

```

```

083     Disp_Buffer[6] = SEG_CODE[s/10];
084     Disp_Buffer[7] = SEG_CODE[s % 10];
085
086     sei();                                //开中断
087     while (1)
088     {
089         if(PINB ^ Key_State)                //如果按键状态变化
090         {
091             _delay_ms(10);                 //延时消抖
092             if(PINB ^ Key_State) .        //再次判断按键状态是否变化
093             {
094                 Key_State = PINB;       //获取当前按键状态
095                 if(!(Key_State & _BV(PB3))) //K1
096                     Increase_Hour();    // + 小时
097                 else
098                     if(!(Key_State & _BV(PB6))) //K2
099                     Increase_Minute();   // + 分钟
100            }
101        }
102    }
103 }
104
105 //-----
106 // T0 定时器溢出中断程序(控制数码管扫描显示)
107 //-----
108 ISR (TIMERO_OVF_vect)
109 {
110     static INT8U Disp_Idx = 0;              //显示数位索引
111     TCNT0 = 256 - F_CPU/64.0 * 0.004;    //位间延时 4 ms
112     PORTA = 0x00;                         //先关闭段码(共阴)
113     PORTA = Disp_Buffer[Disp_Idx];         //输出数码管段码
114     //输出位码(本例使用的是共阴数码管,注意将_BV 取反)
115     PORTD = ~_BV(Disp_Idx);
116     Disp_Idx = (Disp_Idx + 1) & 0x07;       //数码管位索引在 0~7 内循环
117 }
118
119 //-----
120 // T2 中断控制时钟运行
121 //-----
122 ISR (TIMER2_OVF_vect)
123 {
124     //由于 TCNT2 溢出时自动归 0,因此不需要在中断函数中重装初值
125     //TCNT2 = 0;

```

```

126 //T2 时钟为 512 Hz, TCNT2 由 0 计数到 256 时溢出, 故每 0.5 s 中断一次
127 if (Disp_Buffer[2] == 0x40)
128 {
129     Disp_Buffer[2] = Disp_Buffer[5] = 0x00; //前 0.5 s 关闭“-”显示
130 }
131 else
132 {
133     Disp_Buffer[2] = Disp_Buffer[5] = 0x40; //后 0.5 s(即 1 s) 打开“-”显示
134     Increase_Second(); //秒递增
135 }
136 }

```

3.27 TIMER1 定时器比较匹配中断控制音阶播放

本例运行时,按下 K1 按键将输出一段有 14 个音符的音阶,音符输出由定时器控制完成。在输出声音时,通过连接的虚拟示波器可观察到脉宽逐步缩小,频率不断升高。如果 PC 机 CPU 因连接虚拟示波器而过载,导致声音播放失真,这时可断开示波器再播放。本例电路及部分波形如图 3-27 所示。

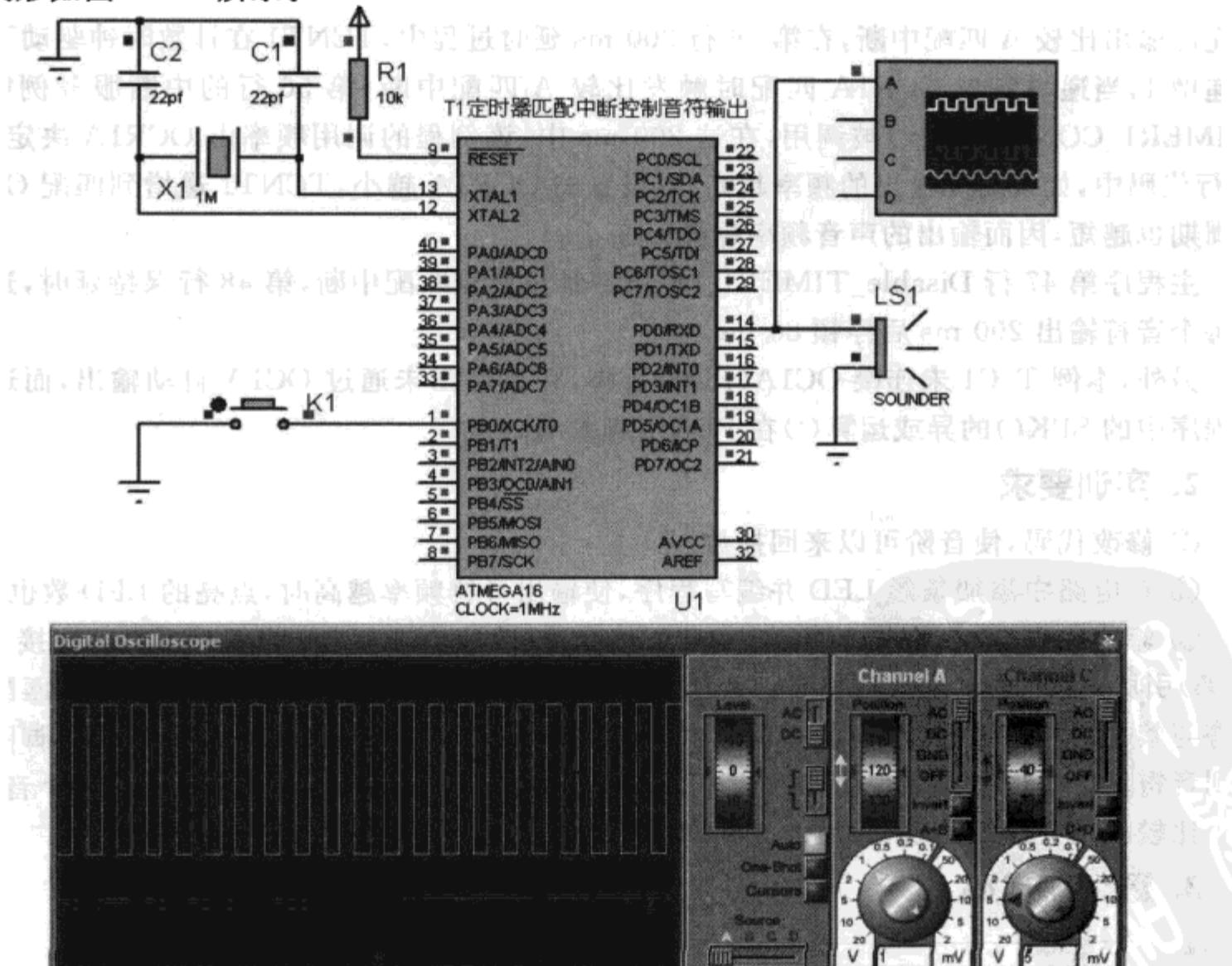


图 3-27 TIMER1 定时器比较匹配中断控制音阶播放



1. 程序设计与调试

本例程序运行时将输出 DO、RE、ME、……的声音，下面是其中前 7 个音符的频率：

简 谱	1	2	3	4	5	6	7
音 符	C5	D5	E5	F5	G5	A5	B5
频 率	523	587	659	698	784	880	987

以上频率的计时初值可根据下面的关系式推出：

- ① 根据频率可得方波周期： $t = 1 / \text{频率} \times 1000000$ (单位为 μs)；
- ② 由于所输出的 t (μs) 周期方波中，高/低电平各占 50%，因此定时器定时(计数)长度为 $\text{Count} = t/2$ ，即 $\text{Count} = 1000000/2/\text{频率}$ 。

主程序中第 32、33 两行将 TCCR1A 与 TCCR1B 分别设为 0x00 与 0x09，它们共同将 WGM1[3:0] 设为 0100，使 T/C1 工作于 CTC 模式(即 OCR1A/B 与 TCNT1 比较匹配时清零 T/C1)，TCCR1B=0x09 还将 TCNT1 计数时钟设为使用 1 分频的系统时钟，即 F_CPU。

根据公式 $\text{Count} = 1000000/2/\text{频率}$ ，可设置输出比较寄存器 OCR1A：

$$\text{OCR1A} = \text{F_CPU}/2/\text{TONE_FRQ}[i]$$

本例 F_CPU 为 1 MHz，由此设定的 OCR1A 决定了相应的输出频率。

设置 OCR1A 寄存器的下一行将 TCNT1 设为 0，随后的第 45 行 Enable_TIMER1_OCIE() 允许输出比较 A 匹配中断，在第 46 行 200 ms 延时过程中，TCNT1 在计数时钟驱动下每微秒递增 1，当递增到与 OCR1A 匹配时触发比较 A 匹配中断，第 56 行的中断服务例程 ISR(TIMER1_COMPA_vect) 被调用，在这 200 ms 中，该例程的调用频率由 OCR1A 决定，在第 43 行代码中，如果需要输出的频率越高，则设置的 OCR1A 越小，TCNT1 递增到匹配 OCR1A 的周期也越短，因而输出的声音频率越高。

主程序第 47 行 Disable_TIMER1_OCIE() 禁止比较匹配中断，第 48 行保持延时，这样可使每个音符输出 200 ms 后停顿 80 ms。

另外，本例 T/C1 未连接 OC1A(PD5) 引脚，频率输出未通过 OC1A 自动输出，而通过中断程序中的 SPK() 的异或运算(^) 在 PD0 引脚输出。

2. 实训要求

- ① 修改代码，使音阶可以来回播放。
- ② 在电路中添加条形 LED 并编写程序，使输出音符频率越高时，点亮的 LED 数也越多。
- ③ 修改代码 32 行的 TCCR1A=0x00，将 0x00 改为 0x40，这样设置后 T/C1 连接 OC1A(PD5) 引脚，在每次比较匹配时 OC1A 引脚会自动取反。通过该设置，程序中的比较匹配中断允许与禁止语句可删除，中断服务程序也可以删除，将蜂鸣器改接至 OC1A(PD5) 引脚同样可听到音符输出。根据以上说明修改代码进行调试时需要注意的是，在输出完最后一个音符后，由于比较匹配会继续出现，最后的高频率音符将持续输出，这个问题要注意解决。

3. 源程序代码

```
01 //-----  
02 // 名称：TIMER1 定时器比较匹配中断控制音阶播放  
03 //-----
```

```

04 // 说明：本例运行时，按下 K1 将在定时器控制下演奏一段音阶 1,2,3,4,5,6,7...
05 //      本例使用了 T1 的定时器比较匹配中断实现不同频率音符输出
06 //
07 //-----
08 #define F_CPU 1000000UL
09 #include <avr/io.h>
10 #include <avr/interrupt.h>
11 #include <util/delay.h>
12 #define INT8U unsigned char
13 #define INT16U unsigned int
14
15 #define K1_DOWN() ((PINB & _BV(PB0)) == 0x00)          //按键定义
16 #define SPK()      (PORTD ^= _BV(PD0))                //蜂鸣器定义
17 //TC1 输出比较 A 匹配中断使能开关
18 #define Enable_TIMER1_OCIE() (TIMSK |= _BV(OCIE1A))
19 #define Disable_TIMER1_OCIE() (TIMSK &= ~_BV(OCIE1A))
20
21 //C 调 15 个音符频率表
22 const INT16U TONE_FRQ[] =
23 { 0,262,294,330,349,392,440,494,523,587,659,698,784,880,988,1046 };
24 //-----
25 // 主程序
26 //-----
27 int main()
28 {
29     INT8U i;
30     DDRB = 0x00; PORTB = 0xFF;                         //配置端口
31     DDRD = 0xFF; PORTD = 0xFF;
32     TCCR1A = 0x00;                                     //TC1 与 OC1A 不连接，禁止 PWM 功能
33     TCCR1B = 0x09;                                     //TC1 预设分频：1
34                                         //CTC 模式（比较匹配时 TC1 自动清零）
35     sei();                                            //开中断
36     while(1)
37     {
38         while (!K1_DOWN());                            //未按键等待
39         while(K1_DOWN());                           //等待释放
40
41         for( i = 1; i<16; i++)
42         {
43             OCR1A = F_CPU/2/TONE_FRQ[i];           //根据频率计算延时，设置 OCR1A
44             TCNT1 = 0;                             //在播放新的音符之前将 TCNT1 清零
45             Enable_TIMER1_OCIE();                 //允许 TC1 比较匹配中断，播放当前音符
46             _delay_ms(200);                      //播放延时

```

```

47     Disable_TIMER1_OCIE();           //禁止 TC1 比较匹配中断,停止播放
48     _delay_ms(80);                 //停顿延时
49   }
50 }
51 }
52
53 //-----
54 // T1 定时器比较匹配中断程序,控制音符频率输出
55 //-----
56 ISR (TIMER1_COMPA_vect)
57 {
58     SPK();
59 }

```

3.28 用 TIMER1 输出比较功能调节频率输出

本例运行时,通过按下 K1~K4 按键可分别调整输出频率的千位、百位、十位与个位数,当前频率将显示在 4 只数码管上。案例电路图及部分运行效果如图 3-28 所示。

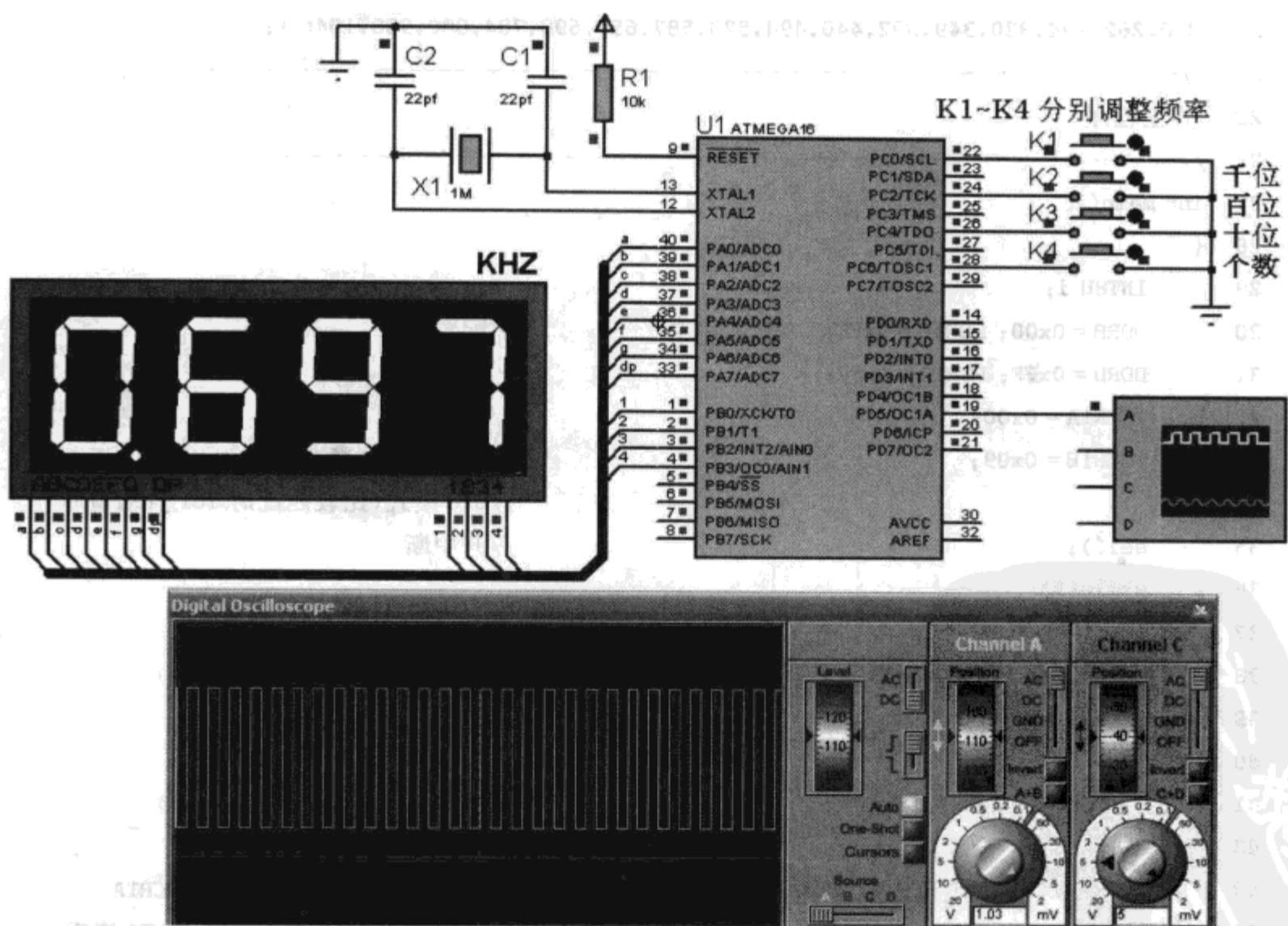


图 3-28 用 TIMER1 输出比较功能调节频率输出

1. 程序设计与调试

本例仍然使用 T/C1 的 CTC(比较匹配清零计数器)模式, 相关设置与上一案例很相似。在阅读调试本例时, 有 2 个要点:

① 上一案例中 T/C1 与 OC1A 不连接, 本例 T/C1 连接了 OC1A 引脚(PD5), 所生成的频率由该引脚输出。

② 本例程序中对按键的扫描未使用“未释放则等待的语句”, 这是因为数码管刷新显示与按键扫描都由主程序中的 while(1) 循环控制完成, 在此循环中, 如果对按键使用 while (按键未释放); 这样的语句来等待释放, 在按下某按键调整频率时, 如果未及时释放则会出现数码管不能被连贯快速扫描而出现缺位的现象。

2. 实训要求

① 本例电路中各按键只能对相应频率数位进行递增循环调节, 完成本例调试仿真后修改仿真电路及程序, 使按键可以任意增减输出频率。

② 将频率输出引脚改为 OC1B(PD6), 重新修改程序, 使用比较匹配中断或非中断方式在该引脚输出所设定的频率。

3. 源程序代码

```

01 //-----
02 // 名称: 用 TIMER1 比较输出功能调节频率输出
03 //-----
04 // 说明: 本例运行过程中, 通过 K1~K4 这 4 个不同按键分别调节频率值的
05 // 千位、百位、十位、个位, 通过虚拟示波器可以观察到不同的频率输出
06 //
07 //-----
08 #define F_CPU 1000000UL           //1 MHz 晶振
09 #include <avr/io.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //定义按键
15 #define K1 (INT8U)(~_BV(PC0))
16 #define K2 (INT8U)(~_BV(PC2))
17 #define K3 (INT8U)(~_BV(PC4))
18 #define K4 (INT8U)(~_BV(PC6))
19
20 //共阴数码管 0~9 的数字编码
21 const INT8U SEG_CODE[] =
22 {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
23 //分解后的待显示频率数位(初值为 100 MHz)
24 INT8U FRQ_DATA[] = {0,1,0,0};
25 //-----

```



```
26 // 数码管显示频率
27 //-----
28 void Show_FRQ_ON_DSY()
29 {
30     INT8U i = 0;
31     for (i = 0; i < 4; i++)
32     {
33         PORTB = ~_BV(i);           //发送扫描码
34         PORTA = SEG_CODE[ FRQ_DATA[i] ]; //发送数字段码
35         if (i == 0) PORTA |= 0x80;
36         _delay_ms(2);
37     }
38 }
39
40 //-----
41 // 频率设置
42 //-----
43 void Set_Frequency()
44 {
45     INT16U f;
46     f = FRQ_DATA[0] * 1000 +           //根据 FRQ_DATA 数组中的各数位
47         FRQ_DATA[1] * 100 +           //计算出频率
48         FRQ_DATA[2] * 10 +
49         FRQ_DATA[3];
50     OCR1A = F_CPU/2.0/f;           //由频率计算出输出比较寄存器 OCR1A 初值
51 }
52
53 //-----
54 // 主程序
55 //-----
56 int main()
57 {
58     INT8U i = 0, Key_State = 0xFF;
59     DDRA = 0xFF; PORTA = 0xFF;           //配置端口
60     DDRB = 0xFF; PORTB = 0xFF;
61     DDRD = 0xFF; PORTD = 0xFF;
62     DDRC = 0x00; PORTC = 0xFF;
63     TCCR1A = 0x40;           //TC1 连接 OC1A 引脚,每次比较匹配时 OC1A 取反
64     TCCR1B = 0x09;           //CTC 模式(比较匹配时清零计数器),分频:1
65     TCNT1 = 0;               //清除定时器值
66     Set_Frequency();          //设置频率
67     while(1)
68     {
```

```

69     if(PINC ^ Key_State)           //如果按键状态变化
70     {
71         Key_State = PINC;        //获取当前按键状态
72         if(Key_State != 0xFF)    //如果有键按下
73         {
74             switch (Key_State)    //根据不同按键分别调整千、百、十、个位
75             {
76                 case K1: i = 0; break;
77                 case K2: i = 1; break;
78                 case K3: i = 2; break;
79                 case K4: i = 3; break;
80             }
81             //修改频率数组的第 i 位(千、百、十、个位)
82             FRQ_DATA[i] = (FRQ_DATA[i] + 1) % 10;
83             Set_Frequency();      //设置频率
84         }
85     }
86     Show_FRQ_ON_DSY();          //数码管显示频率
87 }
88 }
```

3.29 TIMER1 控制的 PWM 脉宽调制器

本例运行时,调节可变电阻,系统程序进行模/数转换后,根据不同的转换结果调节输出不同占空比波形,驱动电机以不同速度转动。在仿真运行过程中,连接虚拟示波器以后如果提示PC机CPU过载,这会使电机转速不正常,这时可删除示波器再运行仿真。案例电路及部分运行效果如图3-29所示。

1. 程序设计与调试

T/C1可工作于一般模式(定时/计数)、CTC模式(比较匹配时清零计数器),以及快速PWM、相位可调PWM及相位频率可调的PWM模式,通过外部运放可构成8位、9位、10位或16位的D/A转换器。

本例主程序中TCCR1A=0x83(10000011)、TCCR1B=0x02(00000010)完成如下设置:

① 0x83的低2位将TCCR1A寄存器最低2位的波形发生器模式(Waveform Generation mode)选择位WGM1[1:0]设为11,0x02的中间2位00将WGM1[3:2]设为00,因此WGM1[3:0]为0011,根据这4位WGM的设置可知T/C1工作模式为10位PWM,相位可调。

② 0x83的高4位1000对于TCCR1A的高4位COM1A[1:0]和COM1B[1:0],它们用于设置T/C1的比较输出模式(Compare Output Modulation:COM)。

由TCCR1A与TCCR1B共同配置的4位波形发生模式位WGM1[3:0]及由TCCR1A高4位COM1A[1:0]与COM1B[1:0]的取值可知比较输出模式为:加1计数与OCR1A比较匹配时将OC1A引脚清零,减1计数与OCR1A比较匹配时将OC1A引脚置1,程序中将该行注



释为：10 位正向 PWM。

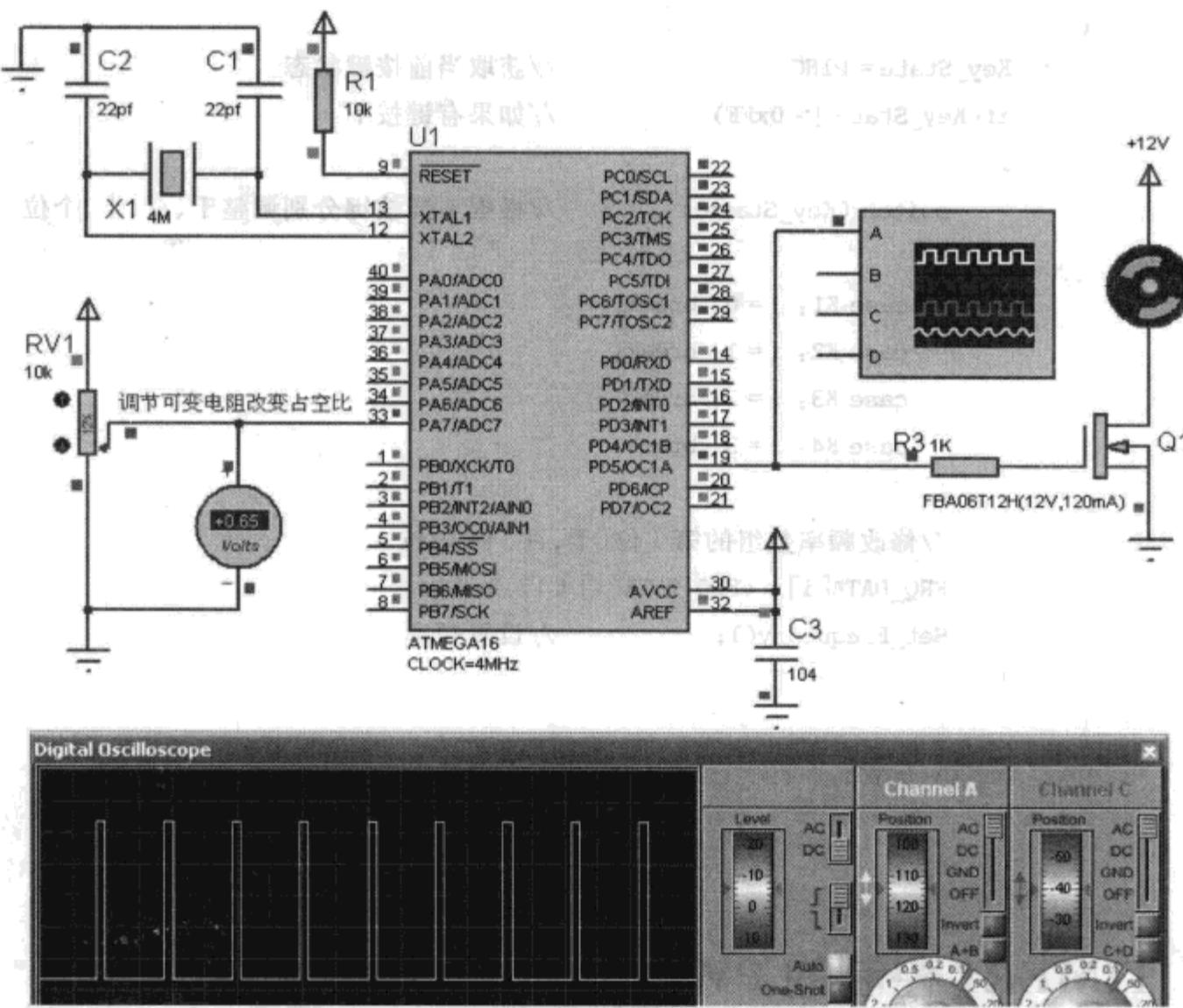


图 3-29 TIMER1 控制的 PWM 脉宽调制器

对于本例 10 位的 PWM，其上限 TOP 值为 1023，主程序中 10 位的模/数转换结果为 0~1023，通过设置 OCR1A 为模/数转换的值，可使 OCR1A 取值范围在 0~1023 之间，OCR1A 的取值决定了 OC1A(PD5)引脚输出脉冲的起始相位和脉宽。在正向 PWM 模式下，OCR1A 取值越小时，占空比越低；OCR1A 取值越大时，占空比越高。但对于 2 个极值 0 与 1023 例外。

在本例相位可调的 PWM 模式下，PWM 波形频率由以下公式确定：

$$F_{OC1A} = F_{CPU}/N/2/TOP \quad (\text{其中 } F_{CPU} \text{ 为系统时钟}, N \text{ 为分频设置})$$

根据本例配置数据可得 OC1A 引脚输出 PWM 波形频率为：

$$F_{OC1A} = 4000000/8/2/1023 \approx 244 \text{ Hz}$$

2. 实训要求

- ① 修改程序，改用 OC1B(PD4)引脚控制电机转速。
- ② 本例 T/C1 配置为相位可调的 PWM 模式，在调试本例后重新将 T/C1 配置为快速 PWM 模式，在 OC1A 或 OC1B 引脚输出占空比可调的 PWM 波形。

3. 源程序代码

```
01 //-----  
02 // 名称：TIMER1 控制的 PWM 脉宽调制器
```

```

03 //-----  

04 // 说明：本例运行过程中，调节 RV1 可改变输出波形的占空比  

05 //-----  

06 //-----  

07 #define F_CPU 4000000UL  

08 #include <avr/io.h>  

09 #include <util/delay.h>  

10 #define INT8U unsigned char  

11 #define INT16U unsigned int  

12  

13 //-----  

14 // 对通道 CH 进行模/数转换  

15 //-----  

16 INT16U ADC_Convert(INT8U CH)  

17 {  

18     int Result;  

19     ADMUX = CH;                                //ADC 通道选择  

20     Result = (INT16U)(ADCL + (ADCH << 8));    //读取转换结果  

21     return Result;  

22 }  

23  

24 //-----  

25 // 主程序  

26 //-----  

27 int main()  

28 {  

29     INT16U x = 0, PRE_ADC_Result = 0;  

30     //float Duty = 1.0;                         //占空比  

31     DDRA = 0x00; PORTA = 0xFF;                  //配置端口  

32     DDRD = 0xFF; PORTD = 0xFF;  

33     DDRC = 0xFF;  

34     ADCSRA = 0xE6;                            //10 位 ADC 转换置位,启动转换,64 分频  

35     _delay_ms(3000);                          //延时等待系统稳定  

36     TCCR1A = 0x83;                            //10 位 PWM(1023),正向 PWM  

37     TCCR1B = 0x02;                            //时钟 8 分频,PWM 频率:F_CPU/8/2046  

38     while(1)  

39     {  

40         x = ADC_Convert(7);  

41         if (x != PRE_ADC_Result)                //如果模/数转换值变化则修改 OCR1A  

42         {  

43             PRE_ADC_Result = x;                 //保存最后一次模/数转换结果  

44             //Duty = x/1023.0;                  //由模/数转换结果计算占空比 Duty  

45             //x = 1023 * Duty;                  //根据占空比求 OCR1A

```



```

46 //上述两行未改变 x 的值,因此可注销
47     if (x == 1023) x = 0;           //如果调节到极值时将极值交换
48     else if (x == 0) x = 1023;
49     OCR1A = x;                  //设置输出比较寄存器 OCR1A(0~1023)
50 }
51 }
52 }

```

3.30 数码管显示两路 A/D 转换结果

本例单片机对 PA 端口输入的多路模拟量进行 A/D 转换,最大转换精度为 10 位。在本例中,ADC0(PA0)、ADC1(PA1)引脚外接两组可变电阻,分别调节 RV0 与 RV1 时,对应两组的模拟电压在进行数字转换后将显示在 8 位数码管上。本例电路及部分运行效果如图 3-30 所示。

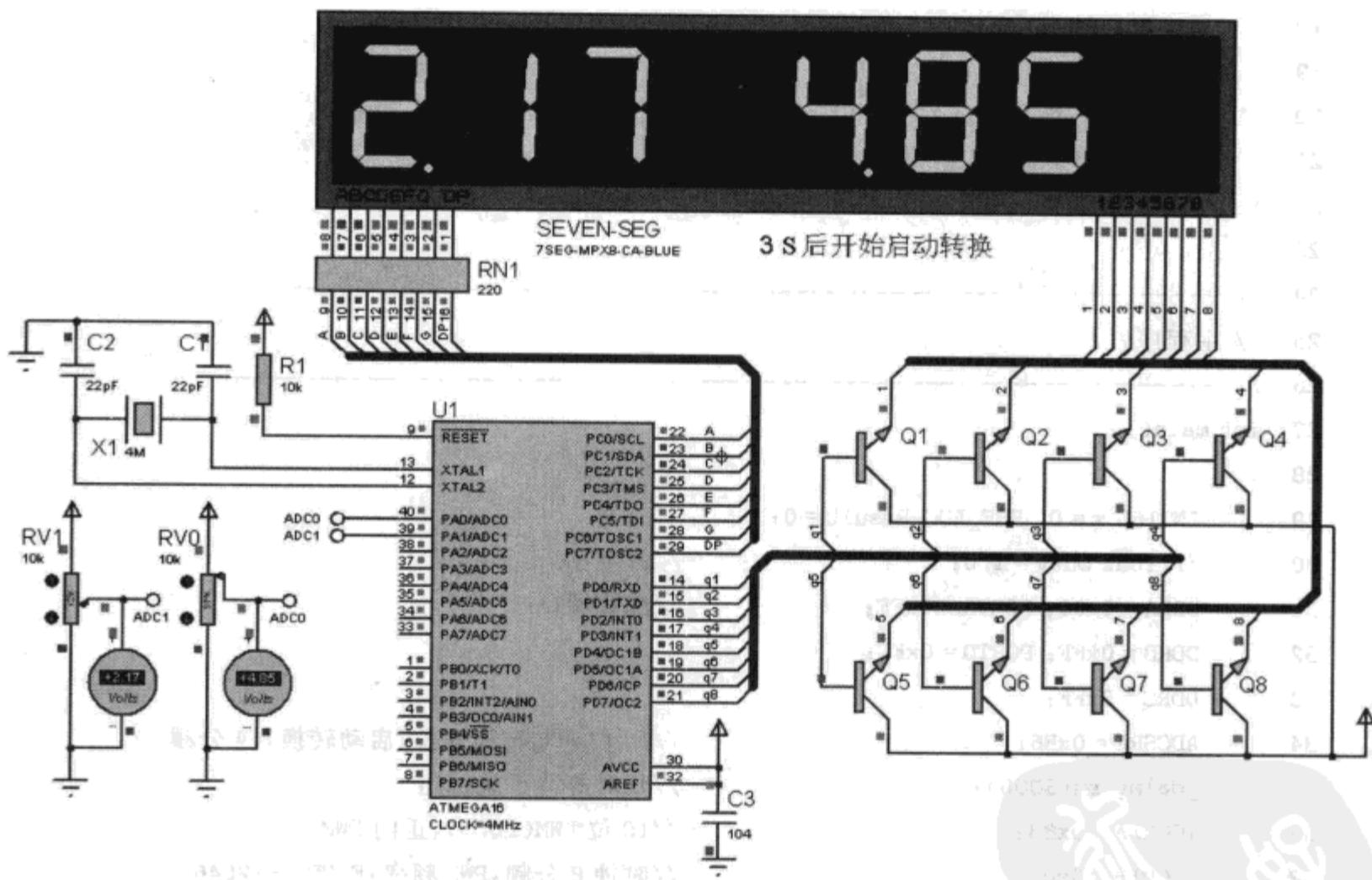


图 3-30 数码管显示两路 A/D 转换结果

1. 程序设计与调试

本例程序设计要点在于:

ADC 控制与状态寄存器 A——ADCSRA, 程序中其取值为 0xE6(11100110), 其最高位 ADEN 置位启动 ADC, ADSC 置位开始转换, ADATE 置位启动 ADC 自动触发功能。ADCSRA 的低 3 位 ADPS[2 : 0]设置为 110, 分频比设为 64。

多路复用选择寄存器——ADMUX，其低 4 位 MUX3~MUX0 为模拟通道选择位，用于选择模拟通道，取值 0000~0111 对应于 ADC0~ADC7。

ADC 数据寄存器——ADC(ADCH/ADCL)，转换结束后数据存放于该寄存中。在读取 ADC 中的数据时，本例 26 行和 27 行提供了分 ADCL 与 ADCH 读取的语句和直接通过 ADC 读取的语句，经编译测试，两种写法都能获取正确的转换结果。

本例程序中单独编写了 ADC 转换函数，参数为待转换通道 CH，函数中 ADMUX 直接等于通道参数 CH。CH 分别取 0 和 1 时，它相当于将低 4 位 MUX3~MUX0 分别设为 0000 和 0001，选择 ADC0 通道和 ADC1 通道进行转换。第 26 行将转换结果除以 1023 再乘以 500，可将 10 位的模/数转换结果 0x0000~0x03FF(即 0~1023)转换为 000~500 之间的待显示数据值(它们对应于电压 0.00~5.00 V)，在显示时两组数值的高位后面单独附加小数点。

2. 实训要求

- ① 选择其他 ADC 通道进行模/数转换并显示结果。
- ② 重新编写本例程序，利用 ADC 中断程序完成转换结果显示。

3. 源程序代码

```

01 //-----
02 // 名称：数码管显示两路 A/D 转换结果
03 //-----
04 // 说明：调节 RV1 和 RV2 时，两路模拟电压将显示在 8 只集成式数码管上
05 //
06 //-----
07 #define F_CPU 4000000UL          //4 MHz
08 #include <avr/io.h>
09 #include <util/delay.h>
10 #define INT8U   unsigned char
11 #define INT16U  unsigned int
12
13 //各数字的数码管段码，最后一位为空白
14 const INT8U SEG_CODE[] =
15 {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xFF};
16 //两路模拟转换结果显示缓冲，显示格式为：X.XX X.XX，第 4 位和第 8 位不显示
17 INT8U Display_Buffer[] = {0,0,0,10,0,0,0,10};
18 //-----
19 // 对通道 CH 进行模/数转换
20 //-----
21 void ADC_Convert(INT8U CH)
22 {
23     int Result;
24     ADMUX = CH;                  //ADC 通道选择
25     //读取转换结果，并转换为电压值
26     Result = (int)((ADCL + (ADCH << 8)) * 500.0/1023.0);
27     //或使用语句：Result = (int)(ADC * 500.0/1023.0);

```

```

28
29 //ADC0 的结果放入数组 0、1、2 单元,ADC1 的结果放入数组 4、5、6 单元
30 Display_Buffer[CH * 4] = Result/100;
31 Display_Buffer[CH * 4 + 1] = Result/10 % 10;
32 Display_Buffer[CH * 4 + 2] = Result % 10;
33 }
34
35 //-----
36 // 主程序
37 //-----
38 int main()
39 {
40     INT8U i;
41     DDRA = 0xFC;                                //配置 A/D 转换端口 ADC0、ADC1 为输入
42     DDRC = 0xFF; PORTC = 0x00;                   //配置数码管显示端口
43     DDRD = 0xFF; PORTD = 0x00;
44     ADCSRA = 0xE6;                                //ADC 转换置位,启动转换,64 分频
45     _delay_ms(3000);                            //延时等待系统稳定
46     while(1)
47     {
48         ADC_Convert(0); ADC_Convert(1);          //对 2 个通道进行 A/D 转换
49         for(i = 0; i < 8; i++)
50         {
51             PORTC = 0xFF;                         //先关闭段码
52             PORTD = _BV(i);                      //发送数码管位码
53             PORTC = SEG_CODE[Display_Buffer[i]]; //发送数字段码
54             if(i == 0 || i == 4) PORTC &= 0x7F;    //对整数位加小数点
55             _delay_ms(4);
56         }
57     }
58 }

```

3.31 模拟比较器测试

本例程序运行时,模拟比较器的正极 AN0 与负极 AN1 所输入的模拟电压将进行比较,如果 AN0 上的电压高于 AN1 上的电压时,LED0 点亮,否则 LED1 点亮。本例电路及部分运行效果如图 3-31 所示。

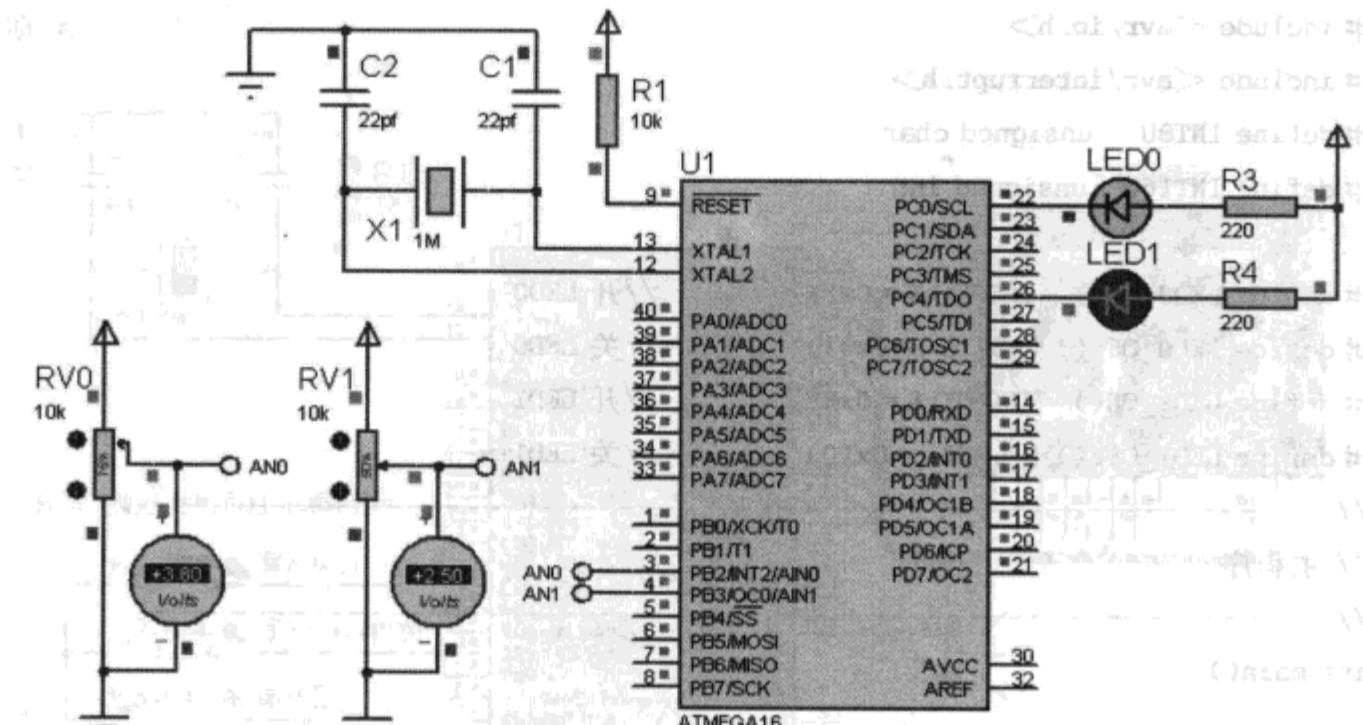


图 3-31 模拟比较器测试

1. 程序设计与调试

本例程序要点在于 SFIOR(Special Function IO Register)与 ACSR(Analog Comparator Control and Status Register)寄存器的设置：

① 主程序将特殊功能 I/O 寄存器 SFIOR 中的 ACME 位清零,使 AN1 连接比较器的负极输入端,在模/数转换状态寄存器 ADCSRA 中的 ADEN 为 0 时,如果将 ACME 置位,模拟比较器将使用 ADC 的多路输入作为负极输入端。

② 将 SFIOR 中的 PUD 置位,禁用内部上拉电阻。

③ 将模拟比较器控制及状态寄存器 ACSR 中的 ACIE 位置位,以允许模拟比较器中断。

在完成上述设置并开中断后,如果 AN0 上的电压高于 AN1 上的电压时,LED0 点亮,否则 LED1 点亮。

2. 实训要求

① 重新配置 25 行的特殊功能 I/O 寄存器 SFIOR 的 ACME 位,并将模/数转换控制与状态寄存器 ADCSRA 的 ADEN 位置 0,利用 ADC 多路输入选择器 ADMUX 将 ADC0~ADC7 中的某一路模拟电压作为模拟比较器的反相(即“-”端)输入源,完成上述模拟比较器测试。

② 在 AN1 引脚提供 1.5 V 电压,编程检测 AN0 引脚模拟电压向上穿越 1.5 V 电压的次数。

3. 源程序代码

```

01 //-----
02 // 名称：模拟比较器测试
03 //-----
04 // 说明：当 AN0 上的电压高于 AN1 时，模拟比较器置位，LED1 点亮，反之 LED2 点亮
05 //
06 //-----
07 #define F_CPU 1000000UL           //1 MHz 晶振

```



```
08 # include <avr/io.h>
09 # include <avr/interrupt.h>
10 # define INT8U unsigned char
11 # define INT16U unsigned int
12
13 # define LED0_ON() (PORTC &= 0xFE) //开 LED0
14 # define LED0_OFF() (PORTC |= 0x01) //关 LED0
15 # define LED1_ON() (PORTC &= 0xEF) //开 LED1
16 # define LED1_OFF() (PORTC |= 0x10) //关 LED1
17 //-----
18 // 主程序
19 //-----
20 int main()
21 {
22     DDRB = 0x00; //PB2,PD3(AIN0/AIN1)设置为输入(无内部上拉)
23     DDRC = 0xFF; //PC 端口设置为输出(外接 LED)
24     SFIOR &= ~_BV(ACME); //AIN1 连接比较器的负极输入端
25     SFIOR |= _BV(PUD); //禁用内部上拉电阻
26     ACSR = _BV(ACIE); //允许模拟比较器中断
27     sei(); //开总中断
28     while(1);
29 }
30
31 //-----
32 // 模拟比较器中断服务程序
33 //-----
34 ISR (ANA_COMP_vect)
35 {
36     if (ACSR & _BV(ACO)) //检查 ACO 位,判断 AN0 电压是否大于 AN1 电压
37     {
38         LED0_ON(); LED1_OFF();
39     }
40     else
41     {
42         LED0_OFF(); LED1_ON();
43     }
44 }
```

3.32 EEPROM 读/写与数码管显示

AVR 单片机内部存储器有 Flash、SRAM、EEPROM 这 3 种。本例程序演示了 EEPROM 数据存储空间透明地址数据和不透明地址数据的读/写与显示。案例电路及部分运行效果如

图 3-32 所示。

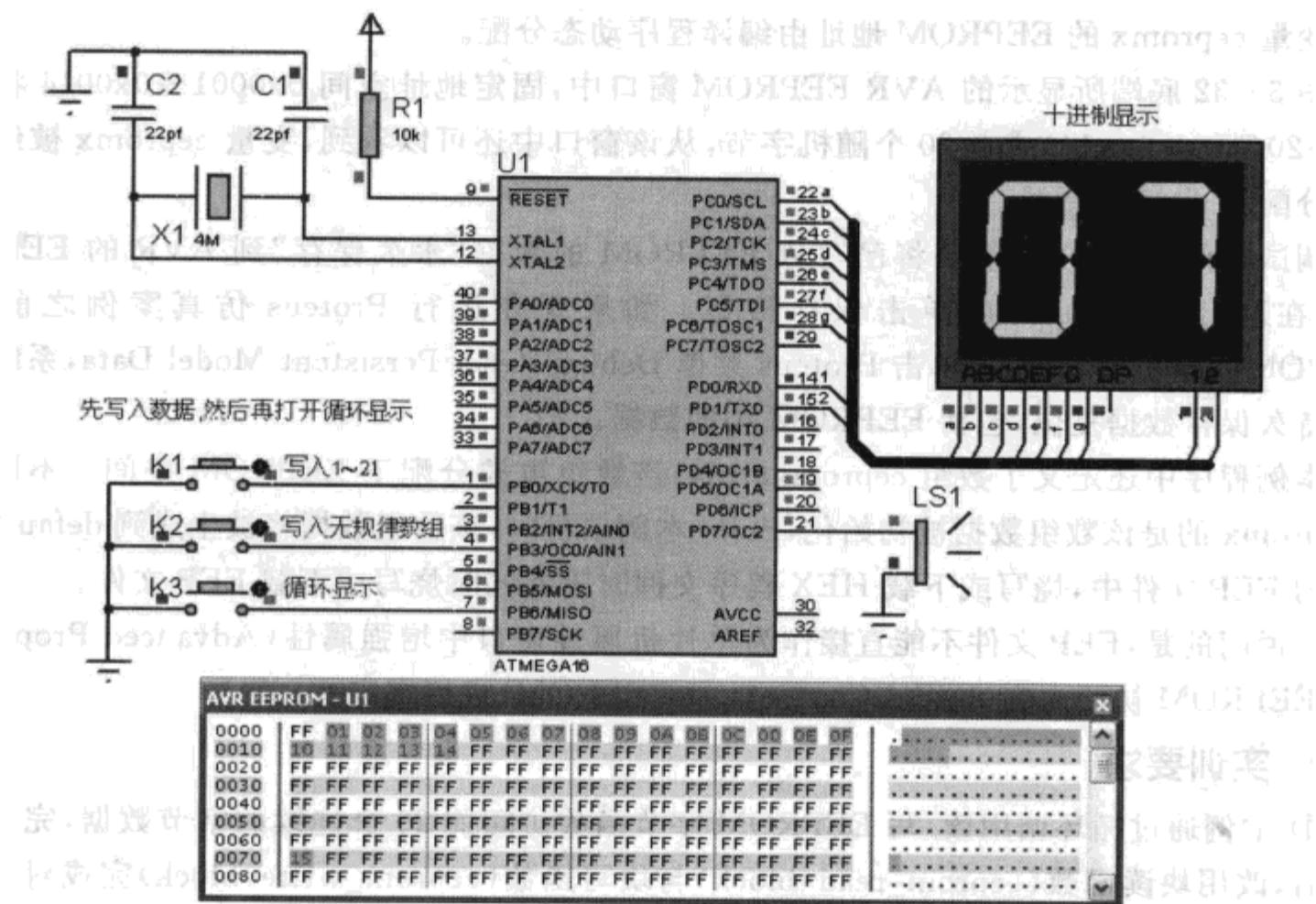


图 3-32 EEPROM 读/写与数码管显示

1. 程序设计与调试

AVR-GCC 提供了专门用于访问 EEPROM 的函数, 这使得 EEPROM 的读/写变得非常简单。在编写本例程序时, 需要添加头文件 `<avr/eeprom.h>`。打开 AVRStudio 帮助菜单中的 `avr-libc` 参考手册, 然后打开 Library Manual, 可找到 `avr/eeprom.h`。该头文件给出了 EEPROM 操作的重要函数:

```

eprom_is_ready()           // EEPROM 就绪
eprom_busy_wait()          // EEPROM 忙等待
eprom_read_byte(地址)      // 从指定地址读取并返回一字节数据
eprom_write_byte(地址, 一字节数据) // 向指定地址写入一字节数据

```

在参考手册中, 还可以看到对字数据进行读/写的函数。

本例读/写的 21 个字节数据中, 前 20 个数据地址是固定的, 其地址空间为 `0x0001~0x0014`, 第 21 个字节数据(对应于 `eepromx`)的地址是编译程序分配的, 字节变量 `eepromx` 通过以下语句申明:

```
INT8U eepromx __attribute__((section("eeprom")));
```

该语句还可以写成:

```
INT8U eepromx EEMEM;
```

其中 `EEMEM` 即 `__attribute__((section("eeprom")))`, 它定义在头文件 `<avr/eeprom.h>`



里面。

变量 eepromx 的 EEPROM 地址由编译程序动态分配。

图 3-32 底端所显示的 AVR EEPROM 窗口中,固定地址空间 0x0001~0x0014 将被写入 1~20(0x01~0x14)或者 20 个随机字节,从该窗口中还可以看到,变量 eepromx 被编译器动态分配到 0x0070 地址。

调试本例程序时,如果要将已写入 EEPROM 的数据“永久保存”到 AVR 的 EEPROM 空间,在退出 Proteus 时应单击保存按钮。如果要在运行 Proteus 仿真案例之前清除 EEPROM 中原有的数据,可单击 Proteus 菜单 Debug/Reset Persistent Model Data,系统会将所有持久保存数据复位,包括 EEPROM 中的数据。

本例程序中还定义了数组 eeprom_array,该数组也被分配于 EEPROM 空间。不同于变量 eepromx 的是该数组数据被初始化,Build 本例项目时,该数组数据将被生成到 default 文件夹下的 EEP 文件中,烧写或下载 HEX 程序文件时需要同时烧写或下载 EEP 文件。

要说明的是,EEP 文件不能直接作为单片机属性窗口中增强属性(Advanced Properties)下的 EEPROM 初始内容(Initial Contents of EEPROM)进行绑定。

2. 实训要求

① 本例通过循环调用读/写 EEPROM 字节函数访问前 20 个连续的字节数据,完成本例调试后,改用块读函数(eeprom_read_block)与块写函数(eeprom_write_block)完成对它们的读/写操作。

② 编程向 EEPROM 空间中先写入 20 个字数据(word data),然后读取显示。

③ 编程向 eeprom_array 数组中写入新的数据,然后读取并显示。

3. 源程序代码

```
001 //-----  
002 // 名称: EEPROM 读/写与数码管显示  
003 //-----  
004 // 说明: 本例运行时,按下 K1 将向 EEPROM 中写入 1~21,按下 K2 时写入无规律  
005 // 的 21 个数,按下 K3 时读取 EEPROM 中的 21 个数并循环显示。  
006 // 所读/写的 21 个数中,前 20 个在 EEPROM 中的地址是透明的(0x0001~0x0014),  
007 // 最后的第 21 个数其地址是不透明的  
008 //-----  
009 //-----  
010 #define F_CPU 4000000UL //4 MHz 晶振  
011 #include <avr/io.h>  
012 #include <avr/eeprom.h>  
013 #include <util/delay.h>  
014 #include <stdlib.h>  
015 #define INT8U unsigned char  
016 #define INT16U unsigned int  
017  
018 //按键定义
```

```

019 #define Write_1_21_Key_DOWN() ((PINB & 0x01) == 0x00) //写 1~21(0x01~0x15)
020 #define Write_Random_Key_DOWN() ((PINB & 0x08) == 0x00) //写随机数
021 #define Loop_Show_Key_DOWN() ((PINB & 0x40) == 0x00) //循环显示
022 //蜂鸣器定义
023 #define BEEP() (PORTD ^= 0x80)
024
025 //将字节变量 eepromx 分配于 EEPROM 存储器(地址不透明)
026 INT8U eepromx __attribute__((section("eeprom")));
027
028 //下面的数组也被分配于 EEPROM,编译后生成 EEP 文件
029 //((其中 EEMEM 即 __attribute__((section("eeprom"))))
030 //后续代码未使用该数组
031 INT8U eeprom_array[] EEMEM =
032 {
033     0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F,
034     0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,
035 };
036
037 //0~9 的数字编码,最后一位为黑屏
038 const INT8U SEG_CODE[] =
039 {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00};
040 //分解后的待显示数位
041 INT8U Display_Buffer[] = {0,0};
042 //-----
043 // 数码管显示字节
044 //-----
045 void Show_Count_ON_DSY()
046 {
047     PORTD = 0xFF;
048     PORTC = SEG_CODE[Display_Buffer[1]];
049     PORTD = 0xFE;
050     _delay_ms(2);
051     PORTD = 0xFF;
052     PORTC = SEG_CODE[Display_Buffer[0]];
053     PORTD = 0xFD;
054     _delay_ms(2);
055 }
056
057 //-----
058 // 响铃子程序
059 //-----
060 void Play_BEEP()

```



```
061  {
062      INT16U i;
063      for (i = 0 ; i < 300; i++)
064      {
065          BEEP(); _delay_us(200);
066      }
067  }
068
069 //-----
070 // 主程序
071 //-----
072 int main()
073 {
074     INT8U Current_Data,LOOP_SHOW_FLAG = 0;
075     INT16U i,Current_Read_Addr = 0x0001;
076     DDRC = 0xFF; PORTD = 0xFF;           //配置输出端口
077     DDRD = 0xFF; PORTD = 0xFF;
078     DDRB = 0x00; PORTB = 0xFF;           //配置输入端口
079     srand(200);                      //设置随机种子
080     while(1)
081     {
082         start:
083         //K1:循环显示-----
084         if(Loop_Show_Key_DOWN())
085         {
086             Current_Read_Addr = 0x0001;    //设为从地址 0x0001 开始显示
087             eeprom_busy_wait();
088             Current_Data = eeprom_read_byte((INT8U *)Current_Read_Addr);
089
090             //因为写入的数据都不超过 100,如果读 0x0001 时出现 0xFF(255),
091             //则说明还未写入过新的数据,这时循环显示标志 LOOP_SHOW_FLAG 仍为关闭
092             //否则才打开循环显示标志
093             if (Current_Data != 0xFF) LOOP_SHOW_FLAG = 1;
094             Play_BEEP();
095             while (Loop_Show_Key_DOWN()); //等待释放
096         }
097         //K2:写入 1~21(0x01~0x15)-----
098         if (Write_1_21_Key_DOWN())
099         {
100             LOOP_SHOW_FLAG = 0;
101             for (i = 1; i <= 20; i++) //根据 Atmel 公司建议,不使用 0 地址
102             {
103                 eeprom_busy_wait();
```

```

104         eeprom_write_byte((INT8U *)i,(INT8U)i);
105     }
106     //前 20 个数的 EEPROM 地址透明,第 21 个数的 EEPROM 地址不透明
107     eeprom_busy_wait();
108     eeprom_write_byte(&EEPROMx,0x15);
109     Play_BEEP();
110     while (Write_1_21_Key_DOWN()); //等待释放
111 }
112 //K3:写入 21 个随机数-----
113 if (Write_Random_Key_DOWN())
114 {
115     LOOP_SHOW_FLAG = 0;
116     for (i = 1; i <= 20; i++)
117     {
118         eeprom_busy_wait();
119         eeprom_write_byte((INT8U *)i,rand() % 100);
120     }
121     //前 20 个数地址透明,第 21 个数地址不透明
122     eeprom_busy_wait();
123     eeprom_write_byte(&EEPROMx,rand() % 100);
124
125     Play_BEEP();
126     while (Write_Random_Key_DOWN()); //等待释放
127 }
128 if (LOOP_SHOW_FLAG)//-----
129 {
130     eeprom_busy_wait();
131     if (Current_Read_Addr != 21) //前 20 个数从透明地址读取
132         Current_Data = eeprom_read_byte((INT8U *)Current_Read_Addr);
133     else //第 21 个数从不透明地址读取
134         Current_Data = eeprom_read_byte(&EEPROMx);
135
136     Display_Buffer[1] = Current_Data/10;
137     Display_Buffer[0] = Current_Data % 10;
138
139     //每个数显示保持一段时间
140     for (i = 0 ; i<160; i++)
141     {
142         Show_Count_ON_DSY();
143         //显示过程中如果某个写入键按下则停止显示
144         if (Write_1_21_Key_DOWN() || Write_Random_Key_DOWN())
145         {
146             LOOP_SHOW_FLAG = 0;

```



```
147             PORTD = 0xFF;
148             goto start;
149         }
150     }
151     //地址循环递增(1~21)
152     Current_Read_Addr = Current_Read_Addr % 21 + 1;
153 }
154 }
155 }
```

3.33 Flash 程序空间中的数据访问

对于在程序运行过程不会发生变化,而且占用空间较大的数据块,本例将其保存到 Flash 程序空间内,并演示了对这些数据的读取与显示。由于本例通过串口发送数据到虚拟终端显示,程序还应用了异步串行接口程序设计技术。本例电路及部分运行效果如图 3-33 所示。

1. 程序设计调试

AVR-GCC 提供了访问 Flash 程序内存空间的相关函数,在编写本例程序时,需要添加头文件<avr/pgmspace.h>,相关细节说明可参考 avr-libc 参考手册。

程序中第 19 行和 22 行分别用 prog_int8_t 类型和 prog_int16_t 类型将含有 320 个字节的数组 Flash_Byte_Array 及含有 60 个字的 Flash_Word_Array 数组保存到 Flash 程序内存中,这大大节省了对 AVR 单片机 RAM 空间的占用。

本例分别使用了 pgm_read_byte 和 pgm_read_word 函数从 Flash 程序内存中读取字节数据与字数据。

为显示所读取的数据,仿真电路中虚拟终端的 RXD 引脚连接单片机的 TXD 引脚,从单片机串口发送的字节数据和字数据将显示到虚拟终端上。

在应用串口发送数据时,需要先初始化串口,初始化步骤如下:

① 设置异步串行通信波特率,收发双方的设置要完全一致,否则会出现收发失败或出现乱码。

② 确定 USART 字符帧结构,包括数据位位数、奇偶校验类型及停止位个数等。

③ 使能发送或接收。

本例的 Init_USART 函数中编写了如下语句:

```
UCSRB = _BV(TXEN);           //允许发送
UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0); //8 位数据位,1 位停止位
UBRRH = (F_CPU/9600/16 - 1) % 256;          //波特率:9600
UBRRH = (F_CPU/9600/16 - 1)/256;
```

前两行设置了 USART 的控制与状态寄存 UCSRB 与 UCSRC:

第 1 行将 UCSRB 中的 TXEN 置位,允许串口数据发送,如果要允许接收,可再将 RXEN 置位。

第 2 行 UCSRC 寄存器中的 UCSZ1,UCSZ0 位与第一行 UCSRB 寄存器中的 UCSZ2 位,即

UCSZ[2:0]共3位,共同设置发送或接收字符帧中的数据位位数大小。UCSZ[2:0]3位的取值为000、001、010、011、111时,字符帧数据位位数大小分别为5、6、7、8、9。本例的设置为011,即数据位位数为8位。另外,由于UCSRC中的停止位USBS位未置位,取值为0,表示停止位为1位,如果将USBS置位则停止位为2位。由于该行的设置与默认值相同,故此行可以省略。

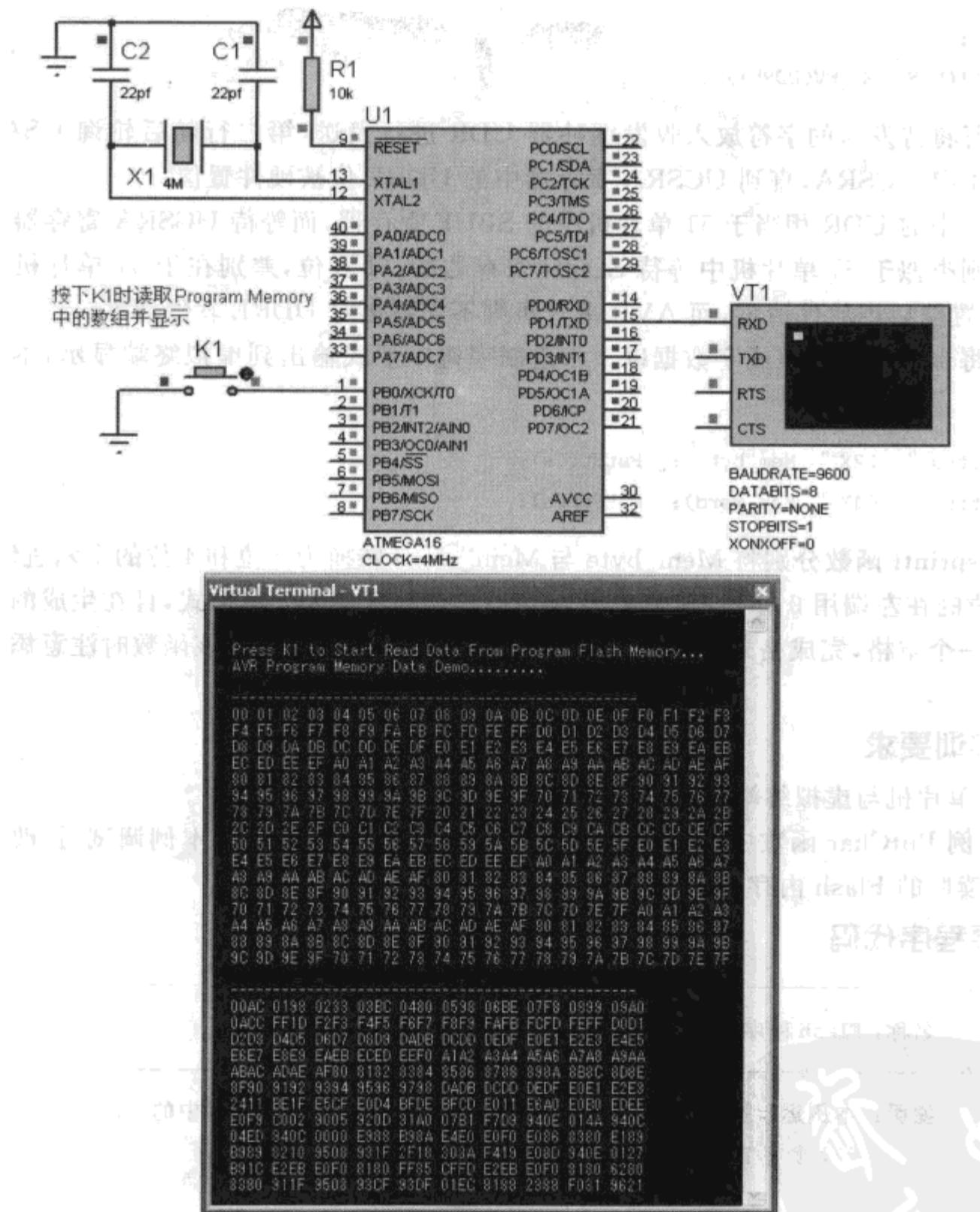


图 3-33 Flash 程序空间中的数据访问

初始化程序最后还需要设置波特率,不同于51单片机是,AVR单片机含有独立的高精度波特率发生器,不需要像51单片机那样占用一个定时/计数器。

波特率寄存器UBRR由UBRRH与UBRRL构成,其中UBRRH的低4位与UBRRL的8位共12位用于保存波特率设置值,UBRRH的低4位是12位波特率的高4位,UBRRL中



存放的则是 12 位波特率的低 8 位。

初始化程序中给出了根据波特率设置 UBRRL 与 UBRRH 的公式,本例将波特率设置为 9600。公式计算结果与精确值之间会存在一定误差,一般误差在 5% 以内是允许的。

完成上述步骤后,就可以进入第③步,利用串口收发数据。本例仅应用串口进行数据发送操作。在第 66 行的 PutChar 函数中,发送字符 c 的关键语句如下:

```
UDR = c;  
while(!(UCSRA & _BV(UDRE)));
```

第 1 行将待发送的字符放入收发缓冲器 UDR 进行发送,第二行随后轮询 USART 的控制与状态寄存 UCSRA,直到 UCSRA 寄存器中的 UDRE 位被硬件置位。

这两行中的 UDR 相当于 51 单片机中的 SBUF 寄存器,而等待 UCSRA 寄存器的 UDRE 硬件置位则类似于 51 单片机中等待 SCON 寄存器的 TI 置位,差别在于 51 单片机中需要在 TI 被硬件置位后用软件清零,而 AVR 单片机则不需要再对 UDRE 软件清零。

为了将十六进制字节或字数据以十六进制字符串形式输出到虚拟终端显示,本例使用了语句:

```
sprintf(s, "%02X ", Mem_byte); PutStr(s);  
sprintf(s, "%04X ", Mem_word); PutStr(s);
```

其中 sprintf 函数分别将 Mem_byte 与 Mem_word 转换为 2 位和 4 位的十六进制数,不足 2 位或 4 位的在左端用 0 补齐,其中 a~f 与 A~F 全部转换为大写形式,且在生成的字符串后面补充了一个空格,完成格式转换后,直接输出字符串 s 即可。引用该函数时注意添加头文件 <stdio.h>。

2. 实训要求

- ① 为单片机与虚拟终端重新选择另一波特率,完成数据发送。
- ② 本例 PutChar 函数中使用轮询标志的方式发送字符,在完成本例调试后,改用中断方式发送所读取的 Flash 内存数据。

3. 源程序代码

```
001 //-----  
002 // 名称: Flash 程序空间的数据访问  
003 //-----  
004 // 说明: 本例运行时,按下 K1 将读取并显示存放于 Flash 程序内存中的  
005 // 320 个字节数据及 60 个字数据  
006 //-----  
007 //-----  
008 #define F_CPU 4000000UL //4 MHz 晶振  
009 #include <avr/pgmspace.h>  
010 #include <stdio.h>  
011  
012 #define INT8U unsigned char  
013 #define INT16U unsigned int
```

```

014
015 //按键定义
016 #define K1_DOWN() (PINB & _BV(PB0)) == 0x00
017
018 //存放于 Flash 程序内存中的字节数据(16 * 20 = 320 个字节)
019 prog_int8_t Flash_Byte_Array[] =
020 {
021     0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,
022     0xF0,0xF1,0xF2,0xF3,0xF4,0xF5,0xF6,0xF7,0xF8,0xF9,0xFA,0xFB,0xFC,0xFD,0xFE,0xFF,
023     0xD0,0xD1,0xD2,0xD3,0xD4,0xD5,0xD6,0xD7,0xD8,0xD9,0xDA,0xDB,0xDC,0xDD,0xDE,0xDF,
024     0xE0,0xE1,0xE2,0xE3,0xE4,0xE5,0xE6,0xE7,0xE8,0xE9,0xEA,0xEB,0xEC,0xED,0xEE,0xEF,
025     0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF,
026     0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F,
027     0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F,
028     0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F,
029     0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,
030     0xC0,0xC1,0xC2,0xC3,0xC4,0xC5,0xC6,0xC7,0xC8,0xC9,0xCA,0xCB,0xCC,0xCD,0xCE,0xCF,
031     0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,
032     0xE0,0xE1,0xE2,0xE3,0xE4,0xE5,0xE6,0xE7,0xE8,0xE9,0xEA,0xEB,0xEC,0xED,0xEE,0xEF,
033     0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF,
034     0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F,
035     0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F,
036     0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F,
037     0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF,
038     0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F,
039     0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F,
040     0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F
041 };
042
043 //存放于 Flash 程序内存中的字数据(10 * 6 = 60 个字)
044 prog_int16_t Flash_Word_Array[] =
045 {
046     0x00AC,0x0198,0x0233,0x03BC,0x0480,0x0598,0x06BE,0x07F8,0x0899,0x09A0,
047     0x0ACC,0xFF1D,0xF2F3,0xF4F5,0xF6F7,0xF8F9,0xFAFB,0xFCFD,0xFEFF,0xD0D1,
048     0xD2D3,0xD4D5,0xD6D7,0xD8D9,0xDADB,0xDCDD,0xDEDF,0xE0E1,0xE2E3,0xE4E5,
049     0xE6E7,0xE8E9,0xEAEB,0xECED,0xEEF0,0xA1A2,0xA3A4,0xA5A6,0xA7A8,0xA9AA,
050     0xABAC,0xADAE,0xAF80,0x8182,0x8384,0x8586,0x8788,0x898A,0x8B8C,0x8D8E,
051     0x8F90,0x9192,0x9394,0x9596,0x9798,0xDADB,0xDCDD,0xDEDF,0xE0E1,0xE2E3
052 };
053 //-----
054 // USART 初始化
055 //-----
056 void Init_USART()

```



```
057  {
058      UCSRB = _BV(TXEN);                                //允许发送
059      UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0);    //8 位数据位,1 位停止位
060      UBRRL = (F_CPU/9600/16 - 1) % 256;              //波特率:9600
061      UBRRH = (F_CPU/9600/16 - 1)/256;
062  }
063 //-----
064 // 发送一个字符
065 //-----
066 void PutChar(char c)
067 {
068     if(c == '\n') PutChar('\r');
069     UDR = c;                                         //将待发送字符放入收发缓冲器
070     while(!(UCSRA & _BV(UDRE)));                  //等待 UDRE 被硬件置位(发送完毕)
071 }
072 //-----
073 // 发送字符串
074 //-----
075 void PutStr(char * s)
076 {
077     while (* s) PutChar(* s++);
078 }
079
080 //-----
081 // 主程序
082 //-----
083 int main()
084 {
085     INT8U Mem_byte;
086     INT16U Mem_word, i, j = 0;;
087     char s[6];
088
089     Init_USART();                                     //串口初始化
090     PutStr("\n\n Press K1 to Start Read Data From Program Flash Memory... ");
091
092     DDRB = 0x00; PORTB = 0xFF;                         //配置端口
093     DDRD = 0xFF;
094
095     while(1)
096     {
097         if (K1_DOWN())
098         {
099             PutStr("\n  AVR Program Memory Data Demo.....\n  ");
```

```

100 PutStr("\n ----- \n ");
101 //读取所有字节并显示
102 for (i = 0, j = 0; i < sizeof(Flash_Bit_Array); i++)
103 {
104     //从 Flash 中读取 1 字节
105     Mem_byte = pgm_read_byte(&Flash_Bit_Array[i]);
106     //将 1 字节转换为字符串(后带 1 个空格)并送虚拟终端显示
107     sprintf(s, "%02X", Mem_byte); PutStr(s);
108     if (++j == 20) //每行显示 20 个字节
109     {
110         j = 0; PutStr("\n ");
111     }
112 }
113 PutStr("\n ----- \n ");
114 //读取所有字并显示
115 for (i = 0, j = 0; i < sizeof(Flash_Word_Array); i++)
116 {
117     //从 Flash 中读取 1 个字
118     Mem_word = pgm_read_word(&Flash_Word_Array[i]);
119     //将读取的 1 个字整数转换为字符串(后带 1 个空格)并送虚拟终端显示
120     sprintf(s, "%04X", Mem_word); PutStr(s);
121     if (++j == 10) //每行显示 10 个字(整数)
122     {
123         j = 0; PutStr("\n ");
124     }
125 }
126 }
127 }
128 }

```

3.34 单片机与 PC 机双向串口通信仿真

通常情况下,虚拟仿真系统是不能与物理环境交互通信的,但 Proteus 虚拟系统模拟了这种能力,它使 Proteus 仿真环境下的系统能与实际的物理环境直接交互,这种模型被称为物理接口模型(PIM)。Proteus 的 COMPIM 组件是一种串行接口组件,当由 CPU 或 UART 软件生成的数字信号出现在 PC 机物理 COM 端口时,它能缓存所接收的数据,并将它们以数字信号的形式发送给 Proteus 仿真电路。如果不希望使用物理串口而使用虚拟串口,使串口调试助手软件能与 Proteus 仿真系统中的单片机串口直接交互,这时还需要安装虚拟串口驱动软件 Virtual Serial Port Driver,简称 VSPD。

本例设计的系统中,单片机可接收 PC 机的串口调试助手软件所发送的数字串,并逐个显示在数码管上,当按下单片机系统的 K1 按键时,会有一串中文字串由单片机串口发送给串口

调试助手软件并显示在软件接收窗口中。

本例电路如图 3-34 所示,串口调试助手的运行效果如图 3-35 所示。

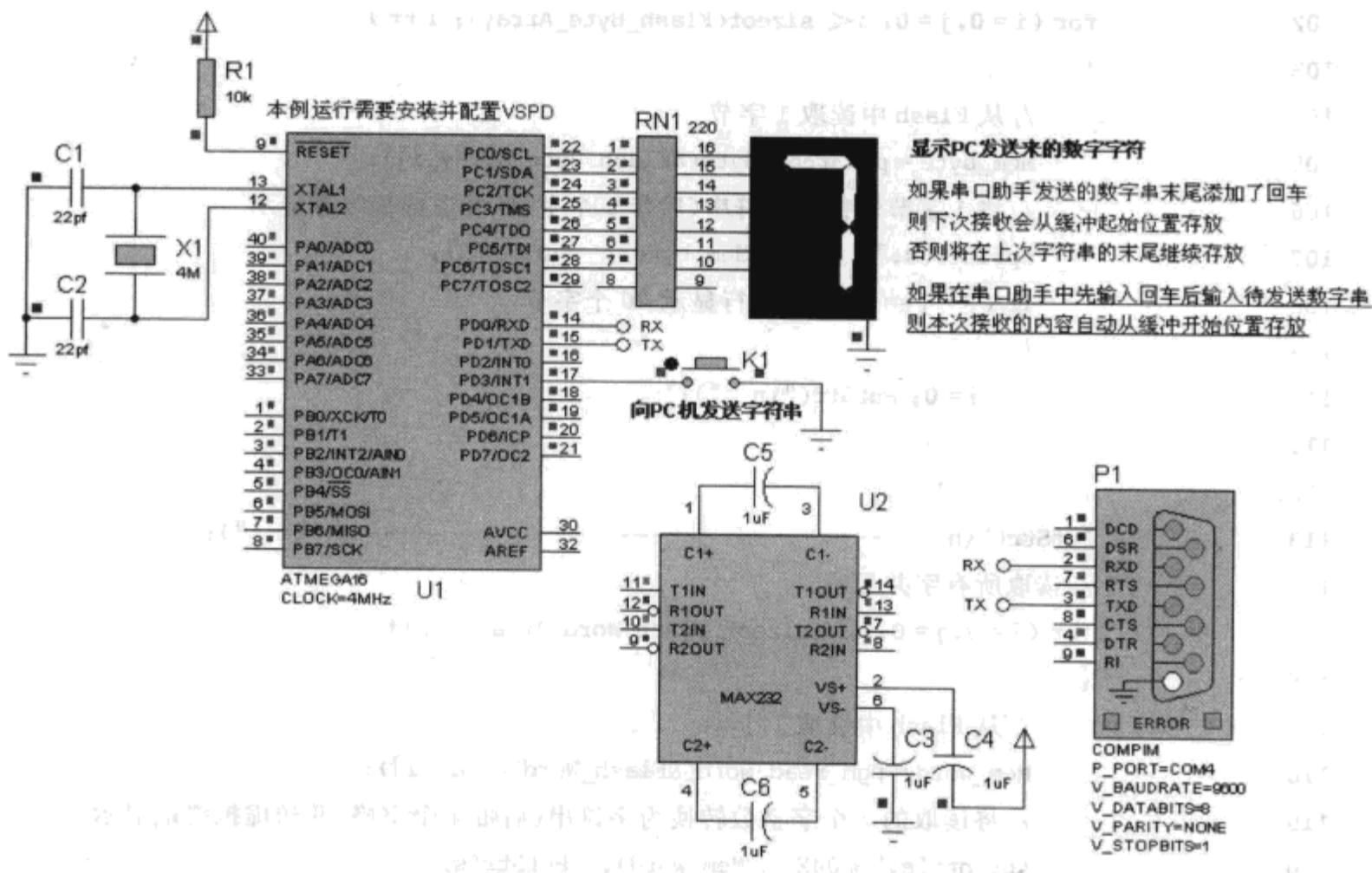


图 3-34 单片机与 PC 机双向串口通信仿真

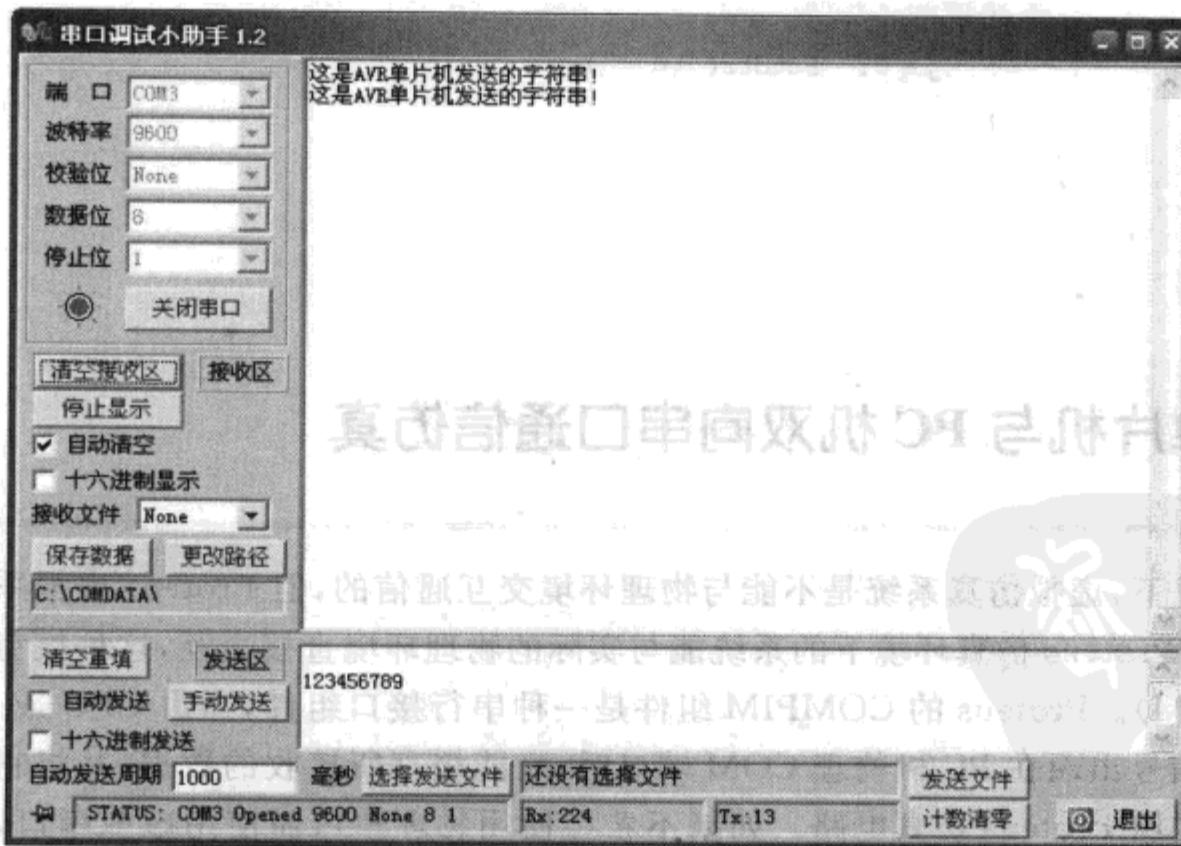


图 3-35 串口调试助手

1. 程序设计与调试

与上一案例中有关串口程序设计代码相比,本例有如下差别:

① 初始化程序中第 45 行添加对 RXEN 与 RXCIE 的置位, 分别允许接收及允许接收中断。

② 对字符的接收, 本例编写了串口接收中断函数 ISR (USART_RXC_vect), 读取所接收的字符时使用语句:c=UDR。

③ 在接收与缓存数字串时, 使用了数据结构中线性表结构形式。

以上这些部分要重点阅读与调试。

因为本例实现的是 PC 机与单片机之间的双向通信, 而且是在纯虚拟仿真环境完成的, 下面重点说明本案例的调试方法。

本例实现的 PC 与单片机通信, 实际上是 PC 机与 Proteus 中单片机仿真系统的通信, 两者的通信通过串口进行, 而串口又有虚拟串口和物理串口两种, 对于本例也就有了以下几种调试方式, 现假设 Proteus 安装在 PC1 中, 如果都使用物理串口, 调试方法有:

方法一: 将串口调试助手软件安装在 PC2, 然后用交叉串口线连接 PC1 与 PC2, 如果两机都是使用的 COM1, 那么在连接好串口线后, 应设置 PC1 中的 COMPIM 属性, 将串口设为 COM1, 波特率等按程序要求设置, 对 PC2 中的串口调试软件也要选择 COM1, 波特率等要设成与 PC1 中的 COMPIM 相同。完成这些设置后, 打开 PC2 中的串口调试软件, 并运行 PC1 中的 Proteus 仿真系统, 这时如果在串口助手软件输入一串数字并单击发送, PC1 中的数码管即会依次显示这些数字, 如果按下 PC1 中单片机系统的 K1 按键, PC2 中的串口调试助手软件会显示:“这是由 AVR 单片机发送的字符串!”并换行, 这样即实现了 PC 机与仿真单片机之间的物理串口通信。当然, 如果两 PC 都使用 COM2 或一个连接 COM1、另一个连接 COM2 也可以, 只是要注意在 COMPIM 组件和串口调试助手 上也要做相应改动。

方法二: 如果希望串口调试软件与单片机仿真系统同在一台 PC 中运行, 假定使用的是 PC1, 如果 PC1 有物理串口 COM1 和 COM2, 这时可以将这两个串口用交叉线连接, 然后仍按上述方法进行调试。不同的是 COMPIM 组件与串口调试软件要分别占用 COM1 和 COM2, 不能占用同一个端口。

上述两种方式均使用的是物理串口, 如果没有找到合适的串口线, 或者使用的 PC 机没有物理串口, 这就需要以虚拟串口软件为桥梁, 实现串口调试助手与 Proteus 仿真单片机系统的串口通信。调试过程如下:

① 安装虚拟串口驱动程序 VSPD (Virtual Serial Port Driver), 安装完成后运行该程序, 在图 3-36 所示窗口的 First Port 中选择 COM3, 在 Second Port 中选择 COM4(当然, 也可以选择 COM5 和 COM6, 除非它们已被占用), 然后单击 Add Ports 按钮, 这两个端口会立即出现在左边的 Virtual Ports 分支下, 且会有蓝色虚线将它们连接起来。如果打开 PC 机的设备管理器, 会发现在其中的端口下多出了两个串口。显示窗口如图 3-37 所示。

② 将这两个串口中的 COM4 分配给 COMPIM 组件使用, COM3 分配给串口助手使用, 由于 COM3 与 COM4 这两个虚拟串口已经由虚拟串口驱动程序 VSPD 虚拟连接, 运行同一台 PC 中的串口调试助手软件和 Proteus 中的单片机仿真系统时, 两者之间就可以进行正常通信了, 这如同使用物理串口连接一样。

2. 实训要求

① 在仿真电路中改用一组 8 位的数码管, 重新编程程序, 将从 PC 机串口助手软件发送的数据滚动显示在数码管上。

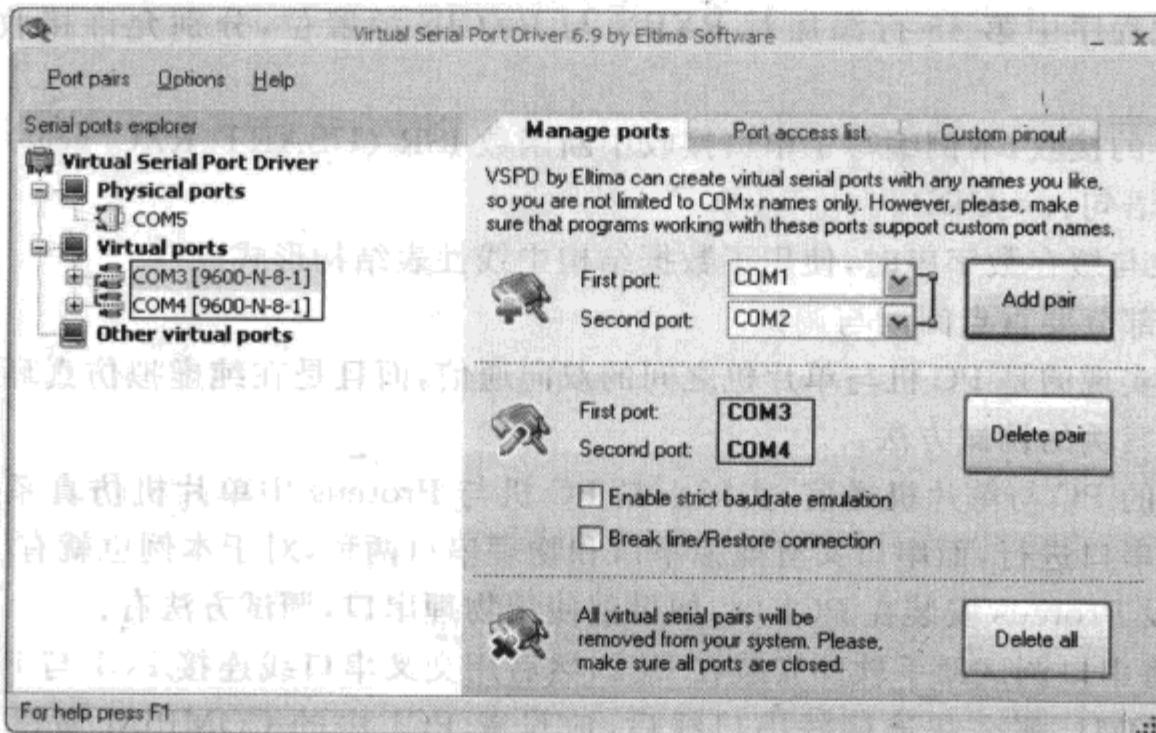


图 3-36 虚拟串口驱动软件

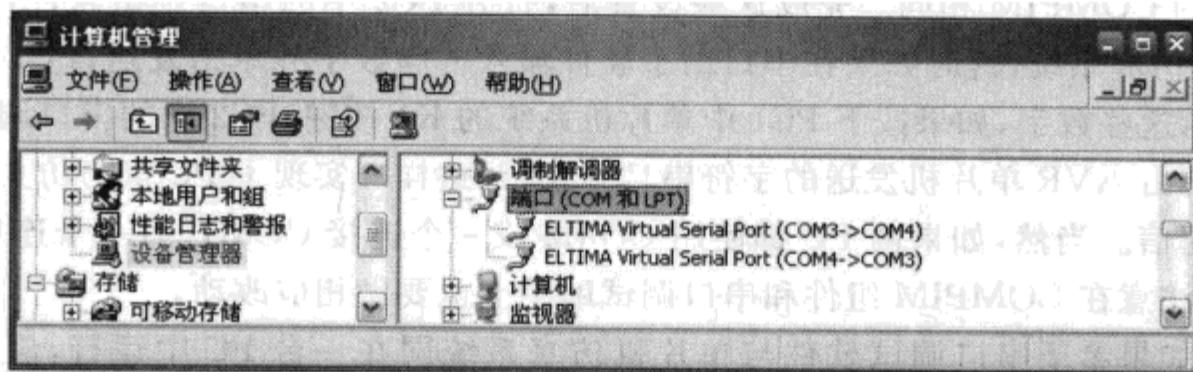


图 3-37 计算机端口管理

② 自编一个上位机 Windows 软件(使用 VB6、VC6、VS. NET 等开发工具),实现对下位单片机系统的控制。在软件中单击“开”按钮时,单片机能控制电机启动;单击“关”按钮时电机停止。调节单片机电路中的可变电阻 RV1 时,模/数转换结果能发送给上位机软件显示。

3. 源程序代码

```

001 //-----
002 // 名称: 单片机与 PC 机双向串口通信仿真
003 //-----
004 // 说明: 单片机可接收 PC 机发送的数字字符,按下单片机 K1 按键时,单片机
005 // 可向 PC 机发送字符串。在 Proteus 环境下完成本实验时,需要先安
006 // 装 Virtual Serial Port Driver 和串口调试助手软件。
007 // 建议在 VSPD 中将 COM3 和 COM4 设为对联端口。Proteus 中设 COM3
008 // 为 COM4,在串口助手中选择 COM3,然后实现单片机程序与 XP 下串口
009 // 助手的通信
010 //
011 // 本例缓冲为 100 个数字字符,如果发送的字符串末尾没有回车符,
012 // 则下次接收的字符串将在上次接收字符串的后面接着存放,否则
013 // 将重新从开始位置存放

```

```

014 //  

015 //      如果本次 PC 发送的数字串是先输入回车符,再输入任意数字串,  

016 //      则本次新接收的数字串也将从缓冲开始位置存放  

017 //  

018 //-----  

019 #define F_CPU 4000000UL          //4 MHz 晶振  

020 #include <avr/io.h>  

021 #include <avr/interrupt.h>  

022 #include <util/delay.h>  

023 #define INT8U unsigned char  

024 #define INT16U unsigned int  

025  

026 //数字串接收缓冲  

027 struct  

028 {  

029     INT8U Buf_Array[100];           //缓冲空间  

030     INT8U Buf_Len;                //当前缓冲长度  

031 } Receive_Buffer;  

032  

033 //清空缓冲标志  

034 INT8U Clear_Buffer_Flag = 0;  

035 //0~9 的数字编码,最后一位为黑屏  

036 const INT8U SEG_CODE[] =  

037 {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00};  

038  

039 char * s = "这是 AVR 单片机发送的字符串! \n", * p;  

040 //-----  

041 // USART 初始化  

042 //-----  

043 void Init_USART()  

044 {  

045     UCSRB = _BV(RXEN) | _BV(TXEN) | _BV(RXCIE);    //允许接收和发送,接收中断使能  

046     UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0);   //8 位数据位,1 位停止位  

047     UBRRRL = (F_CPU/9600/16 - 1) % 256;            //波特率:9600  

048     UBRRRH = (F_CPU/9600/16 - 1)/256;  

049 }
050 //-----  

051 // 发送一个字符  

052 //-----  

053 void PutChar(char c)  

054 {  

055     if(c == '\n') PutChar('\r');  

056     UDR = c;

```



```
057     while(!(UCSRA & _BV(UDRE)));
058 }
059 //-----
060 // 显示所接收的数字字符(数字字符由 PC 串口发送,AVR 串口接收)
061 //-----
062 void Show_Received_Digits()
063 {
064     INT8U i;
065     for (i = 0; i < Receive_Buffer.Buf_Len; i++)
066     {
067         PORTC = SEG_CODE[Receive_Buffer.Buf_Array[i]];
068         _delay_ms(400);
069     }
070 }
071 //-----
072 // 主程序
073 //-----
074 int main()
075 {
076     Receive_Buffer.Buf_Len = 0;
077     DDRB = 0x00; PORTB = 0xFF;           //配置端口
078     DDRC = 0xFF; PORTC = 0x00;
079     DDRD = 0x02; PORTD = 0xFF;
080     MCUCR = 0x08;                     //INT1 中断下降沿触发
081     GICR = _BV(INT1);                //INT1 中断许可
082     Init_USART();                   //串口初始化
083     sei();
084     while(1) Show_Received_Digits(); //显示所接收到数字
085 }
086
087 //-----
088 // 串口接收中断函数
089 //-----
090 ISR (USART_RXC_vect)
091 {
092     INT8U c = UDR;
093     //如果接收到回车换行符则设置清空缓冲标志
094     if (c == '\r' || c == '\n') Clear_Buffer_Flag = 1;
095     if (c >= '0' && c <= '9')
096     {
097         //如果上次曾收到清空缓冲标志,则本次从缓冲开始位置存放
098         if (Clear_Buffer_Flag == 1)
099         {
100             Receive_Buffer.Buf_Len = 0;
```

```

101     Clear_Buffer_Flag = 0;
102 }
103 //缓存新接收的数字
104 Receive_Buffer.Buf_Array[Receive_Buffer.Buf_Len] = c - '0';
105 //刷新缓冲长度(不超过最大长度)
106 if (Receive_Buffer.Buf_Len < 100) Receive_Buffer.Buf_Len++;
107 }
108 }
109
110 //-----
111 // INT1 中断函数(向 PC 发送字符串)
112 //-----
113 ISR (INT1_vect)
114 {
115     INT8U i = 0;
116     while (s[i] != '\0') PutChar(s[i++]); //向 PC 发送字符串
117 }

```

3.35 看门狗应用

单片机的工作常会受到来自外界电磁场的干扰,造成程序跑飞,单片机系统无法继续正常工作。本例演示了启动看门狗,用定时器喂狗以及停止喂狗导致单片机重启的过程。在启动完成后,LED1 熄灭,LED2 开始持续闪烁,一旦停止喂狗则系统自动重启,LED1 在启动时被再次点亮一次,然后熄灭,LED2 再次重新开始闪烁,系统重新进入正常运行状态。本例电路及部分运行效果如图 3-38 所示。

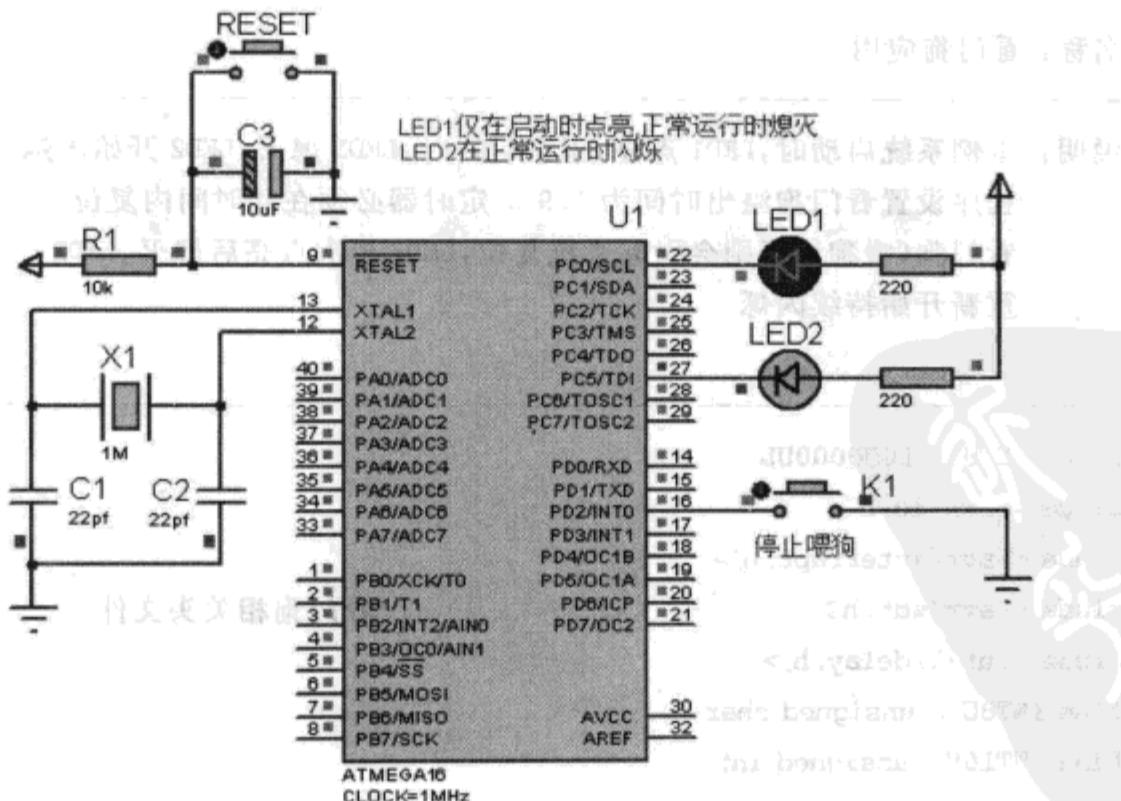


图 3-38 看门狗应用



1. 程序设计与调试

AVR-GCC 提供了看门狗(watchdog)的相关控制函数，在应用看门狗时需要添加头文件：`<avr/wdt.h>`。通过 `wdt.h` 的宏定义 `wdt_enable` 和 `wdt_reset` 可以非常方便地启用和复位看门狗。

电路中 LED1 仅在开始时点亮，完成 T/C1 定时器溢出中断与外部 INT0 中断配置后，通过调用 `wdt_enable` 启动看门狗，喂狗时间设为 2 s(1.9 s)，LED2 熄灭，随后即进入主程序中的 `while(1)` 循环，应用系统要正常处理的事务将在该循环中完成，本例所放置的代码仅控制 LED2 的闪烁，它表示单片机处于正常运行状态下。

当 LED2 开始闪烁时，表示程序开始运行正常，T/C1 定时器溢出中断函数每隔 1.5 s(<1.9 s) 调用 `wdt_reset` 复位看门狗(喂狗)，这样可使系统持续正常运行。当按下 K1 时，它模拟了异常事件导致定时器停止工作。系统出现故障、喂狗停止、程序跑飞的状态，由于喂狗时间超过 2 s，这导致单片机应用系统自动重启。LED1 被再次点亮，然后熄灭，随后 LED2 再次开始持续闪烁，系统重新恢复正常。

在调试运行本例时，按下 K1 停止喂狗可使单片机自动重启。这与按下电路中的热启动键 RESET 所观察到的效果是一样的，区别在于按下 RESET 键是“手动重启”系统，而按下 K1 则模拟了系统遇到故障后“自动重启”的过程。

2. 实训要求

- ① 重新配置喂狗时间为 8 s，并修改相关定时器中断程序，实现上述仿真效果。
- ② 本例将 LED2 闪烁作为正常运行的任务，在完成本例调试后，将数码管显示当前时钟信息作为主程序的正常运行任务，在系统出现故障时能自动重启，然后再重新进入正常运行状态。

3. 源程序代码

```
01 //-----  
02 // 名称：看门狗应用  
03 //-----  
04 // 说明：本例系统启动时，LED1 点亮，正常运行时，LED1 熄灭，LED2 开始闪烁  
05 // 程序设置看门狗溢出时间为 1.9 s，定时器必须在此时间内复位  
06 // 看门狗(喂狗)，否则会引起系统复位，LED1 再次点亮后熄灭，LED2  
07 // 重新开始持续闪烁  
08 //  
09 //-----  
10 #define F_CPU 1000000UL  
11 #include <avr/io.h>  
12 #include <avr/interrupt.h>  
13 #include <avr/wdt.h> //看门狗相关头文件  
14 #include <util/delay.h>  
15 #define INT8U unsigned char  
16 #define INT16U unsigned int  
17  
18 //分别定义 LED1 开/关，LED2 闪烁  
19 #define LED1_ON() (PORTC &= ~_BV(PC0))
```

```

20 #define LED1_OFF() (PORTC |= _BV(PC0))
21 #define LED2_BLINK() (PORTC ^= _BV(PC5))
22 //-----
23 // 主程序
24 //-----
25 int main()
26 {
27     DDRC = 0xFF; PORTC = 0xFF;           //配置端口
28     DDRD = 0x00; PORTD = 0xFF;
29     LED1_ON();                         //LED1 点亮
30     _delay_ms(1600);
31
32     MCUCR = 0x02;                     //INT0 中断下降沿触发
33     GICR = _BV(INT0);                 //INT0 中断许可
34     TCCR1B = 0x03;                   //T1 预设分频:64
35     TCNT1 = 65536 - F_CPU/64.0 * 1.5; //晶振 4 MHz,1.5 s 定时初值
36     TIMSK = 0x04;                   //允许 T1 定时器溢出中断
37     wdt_enable(WDTO_2S);            //启动看门狗(溢出时间 1.9 s,约等于 2.0 s)
38     //WDTCSR = 0x0F;                //用这一行也可以
39     LED1_OFF();                    //LED1 熄灭
40     sei();                         //开中断
41     while(1)
42     {
43         LED2_BLINK();              //LED2 闪烁
44         _delay_ms(200);
45     }
46 }
47
48 //-----
49 // 定时器 1 中断程序负责喂狗(1.9 s 以内)
50 //-----
51 ISR (TIMER1_OVF_vect)
52 {
53     TCNT1 = 65536 - F_CPU/64.0 * 1.5; //1.5 s 定时初值
54     wdt_reset();                   //看门狗复位
55 }
56
57 //-----
58 // INT0 中断函数(按下 K1 时关闭定时器,停止喂狗)
59 //-----
60 ISR (INT0_vect)
61 {
62     TIMSK = 0x00;
63 }

```

第4章

硬件应用

通过对第3章基础案例的学习与调试,大家已经熟悉了AVRStudio+WinAVR开发环境下单片机内部资源的基本程序设计方法,知道如何利用AVR单片机C语言程序设计实现基本的系统功能。本章将在此基础上就单片机的外围硬件扩展提出数十个案例,这些硬件包括大量数字逻辑芯片、驱动芯片、机电器件、显示器件、传感器件等。通过认真的学习研究与跟踪调试,以及对实训要求的认真完成,大家一定会进一步熟悉和掌握单片机外围扩展硬件的应用方法与技巧,积累更多应用经验,进一步提高AVR单片机应用系统的C语言程序开发能力,为单片机系统的综合设计打下基础。

4.1 74HC138与74HC154译码器应用

本例单片机PB与PC端口分别外接3-8译码器与4-16译码器,程序在PB端口低3位输出000、001、010、011、…、111,通过3-8译码器控制8只LED滚动点亮。在PC端口低4位则循环输出0000、0001、0010、0011、…、1111,通过4-16译码器控制16只LED循环滚动显示。本例电路及部分运行效果如图4-1所示。

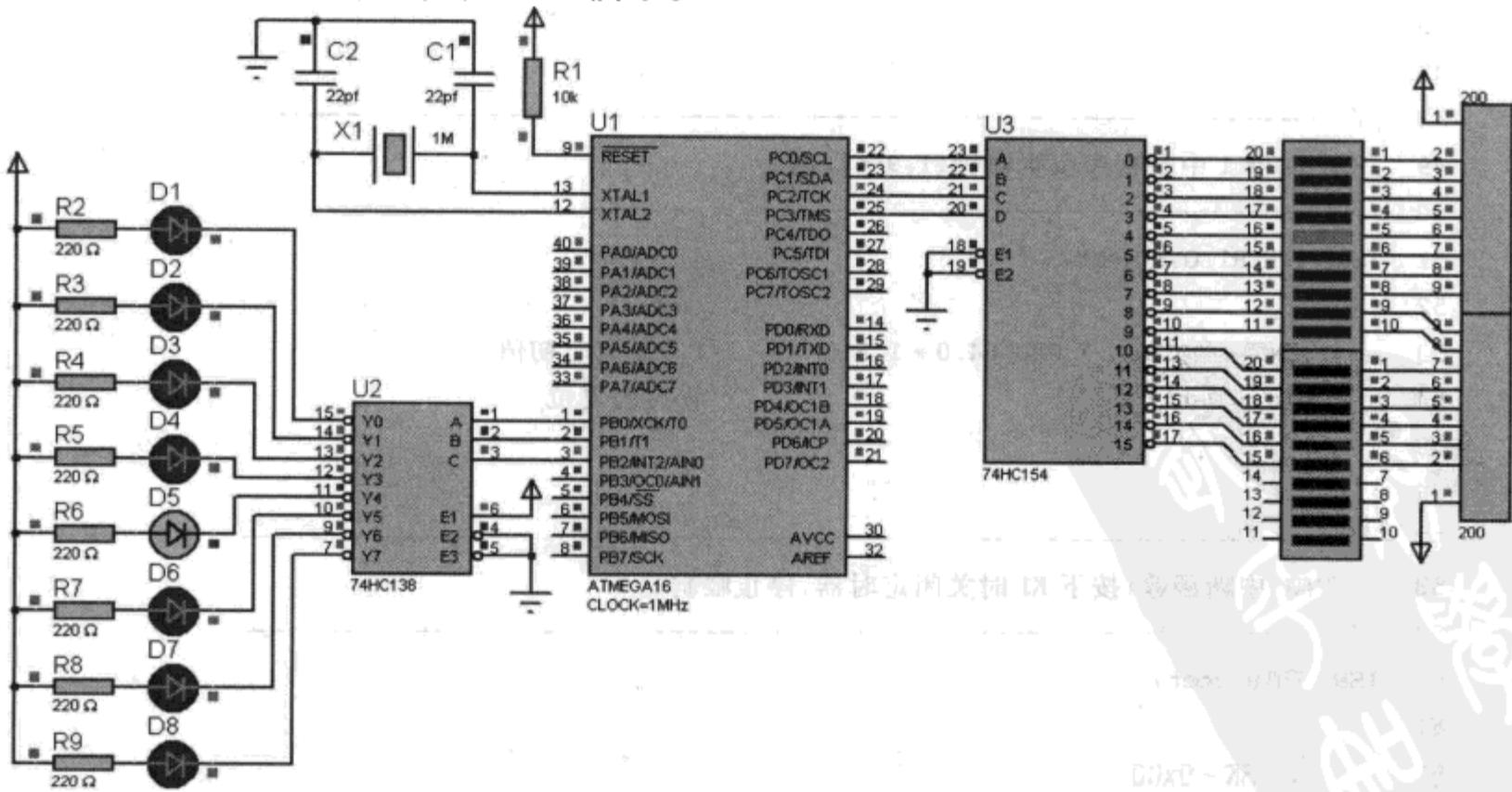


图4-1 74HC138与74HC154译码器应用

1. 程序设计与调试

表 4-1 是 3-8 译码器 74HC138 的真值表。PB 端口低 3 位连接 3-8 译码器的 CBA 输入端,依次输入 000、001、010、011、……、111。根据 3-8 译码器的真值表可知,在向 3-8 译码器输入 000 时,输出端 Y0 引脚为 0;输入 001 时,输出端 Y1 引脚为 0;输入 111 时,输出端 Y7 引脚为 0,这样即形成了 8 只 LED 逐个滚动点亮的效果。

表 4-1 3-8 译码器 74HC138 的真值表

输入					输出							
使能位		选择位			Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
G1	G2*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	L	H	H	H
H	L	H	H	H	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

注: G2=G2A+G2B(本例中 GA=E2+E3)。

控制 3-8 译码器的语句是 PORTB=(PORTB+1) & 0x07。该语句使 PB 端的输出范围为 0~7,即 00000000~00000111,其高 5 位保持为 00000,而低 3 位由 000、001、010、……,一直递增到 111,经译码器译码后即形成 LED 滚动显示的效果。

单片机 PC 端口低 4 位连接 4-16 译码器的 DCBA 输入端,依次输入 0000、0001、0010、0011、……,直到 1111。根据 4-16 译码器的真值表可知,输入 0000 时,输出端引脚 0(Y0)为 0;当输入 0001 时,输出端引脚 1(Y1)为 0;当输入 1111 时,输出端引脚 15(Y15)为 0;16 只 LED 逐个滚动点亮的效果由此形成。表 4-2 给出了 4-16 译码器 74HC154 的真值表。

表 4-2 4-16 译码器 74HC154 的真值表

输入		输出																
G1G2		DCBA	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	L	LLLL	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	
L	L	LLLH	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	
L	L	LLHL	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	
L	L	LLHH	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	
L	L	LHLL	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	
L	L	LHLH	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	



续表 4-2

输入		输出																
G1G2		DCBA	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	L	LHHL	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	
L	L	LHHH	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	
L	L	HLLL	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	
L	L	HLLH	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	
L	L	HLHL	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	
L	L	HLHH	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	
L	L	HHLL	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	
L	L	HHLH	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	
L	L	HHHL	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	
L	L	HHHH	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	
L	H	XXXX	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	
H	L	XXXX	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	
H	H	XXXX	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	

控制 4-16 译码器的语句是 `PORTC=(PORTC+1) & 0x0F`, 它使 PC 端口的输出范围为 0~15(即 00000000~00001111), 其高 4 位保持为 0000, 而低 4 位由 0000、0001、0010、……, 一直递增到 1111, 经译码器译码后使相应的 LED 点亮, 形成 LED 滚动显示的效果。

2. 实训要求

① 删除所有连接 3-8 译码器的 LED, 重新加入 8 位七段数码管, 用 3-8 译码器控制数码管位码, 用 PA 端口控制段码, 实现数码管数据显示。

② 在成功调试 4.9 节有关 LED 点阵屏的案例后, 删除本例中连接 4-16 译码器的条形 LED, 重新加入两片水平并排 8×8 LED 点阵显示屏, 用 4-16 译码器控制列码(两片共 16 列), 行码由 PA 端口控制, 实现 2 片点阵屏的静态或滚动显示效果。这样设计可大大减少对单片机 I/O 端口的占用, 如果直接控制 16 列, 单片机将有 2 个端口被完全占用, 使用译码器后只需要一个端口的 4 只引脚即可。

3. 源程序代码

```

01 //-
02 // 名称: 74HC138 与 74HC154 译码器应用
03 //-
04 // 说明: 本例运行时, PB 与 PC 端口分别循环输出 0x00~0x07, 0x00~0x0F
05 //      两译码器的输出端 Y0~Y7 与 Y0~Y15 分别逐个呈现低电平
06 //      两组 LED 分别循环滚动显示
07 //
08 //-
09 #define F_CPU 1000000UL
10 #include <avr/io.h>

```

```

11 #include <util/delay.h>
12 #define INT8U unsigned char
13 #define INT16U unsigned int
14
15 //-----
16 // 主程序
17 //-----
18 int main()
19 {
20     DDRB = 0xFF; PORTB = 0x00;           //配置 PB,PC 端
21     DDRC = 0xFF; PORTC = 0x00;           //初始输出均为 0x00
22     while(1)
23     {
24         PORTB = (PORTB + 1) & 0x07;       //3-8 译码器输出
25         PORTC = (PORTC + 1) & 0x0F;       //4-16 译码器输出
26         //以上两行还可以改写成:
27         //PORTB = (PORTB + 1) % 8;
28         //PORTC = (PORTC + 1) % 16;
29         _delay_ms(80);                  //延时
30     }
31 }

```

4.2 74HC595 串入并出芯片应用

本例单片机外接一片串入并出芯片 74HC595，该芯片仅占用单片机 PC 端口 3 只引脚，驱动单只数码管实现数字滚动显示。74HC595 芯片在后续有关 LED 点阵显示屏的案例中还会再次用到，通过本例调试要熟练掌握该芯片的程序设计方法。本例电路及部分运行效果如图 4-2 所示。

1. 程序设计与调试

74HC595 的输出端为 Q₀~Q₇，这 8 位并行输出端可以直接控制数码管的 8 个管段（本例数码管没有小数点，仅连接了数码管的 7 个引脚）。Q_{7'} 为级联输出端，它用来连接下一片 595 的串行数据输入端 DS。

74HC595 的控制端说明如下：

① SH_CP(11 脚)用于输入移位时钟脉冲，在上升沿时移位寄存器（Shift Register）数据移位，Q₀→Q₁→Q₂→Q₃→Q₄→Q₅→Q₆→Q₇→Q_{7'}，其中 Q_{7'} 用于 595 的级联。本例中的 595 串行输入函数 Serial_Input_595 使用了 SH_CP 引脚及下面的 DS 引脚。

② DS(14 脚)为串行数据输入引脚，Serial_Input_595 函数通过移位运算符由高位到低位将位数据通过 DS 引脚串行送入 595 芯片，串行发送时由 SH_CP 引脚提供移位时钟。for 循环控制完成 8 次移位即可完成一个字节的串行传送。

③ ST_CP(12 脚)提供锁存脉冲，在上升沿时移位寄存器的数据被传入存储寄存器，由于

\overline{OE} 引脚接地,传入存储寄存器的数据会直接出现在输出端 Q0~Q7。在串行输入函数完成一个字节的传送后,并行输出函数 Parallel_Output_595 在 ST_CP 的上升沿将数据送出。

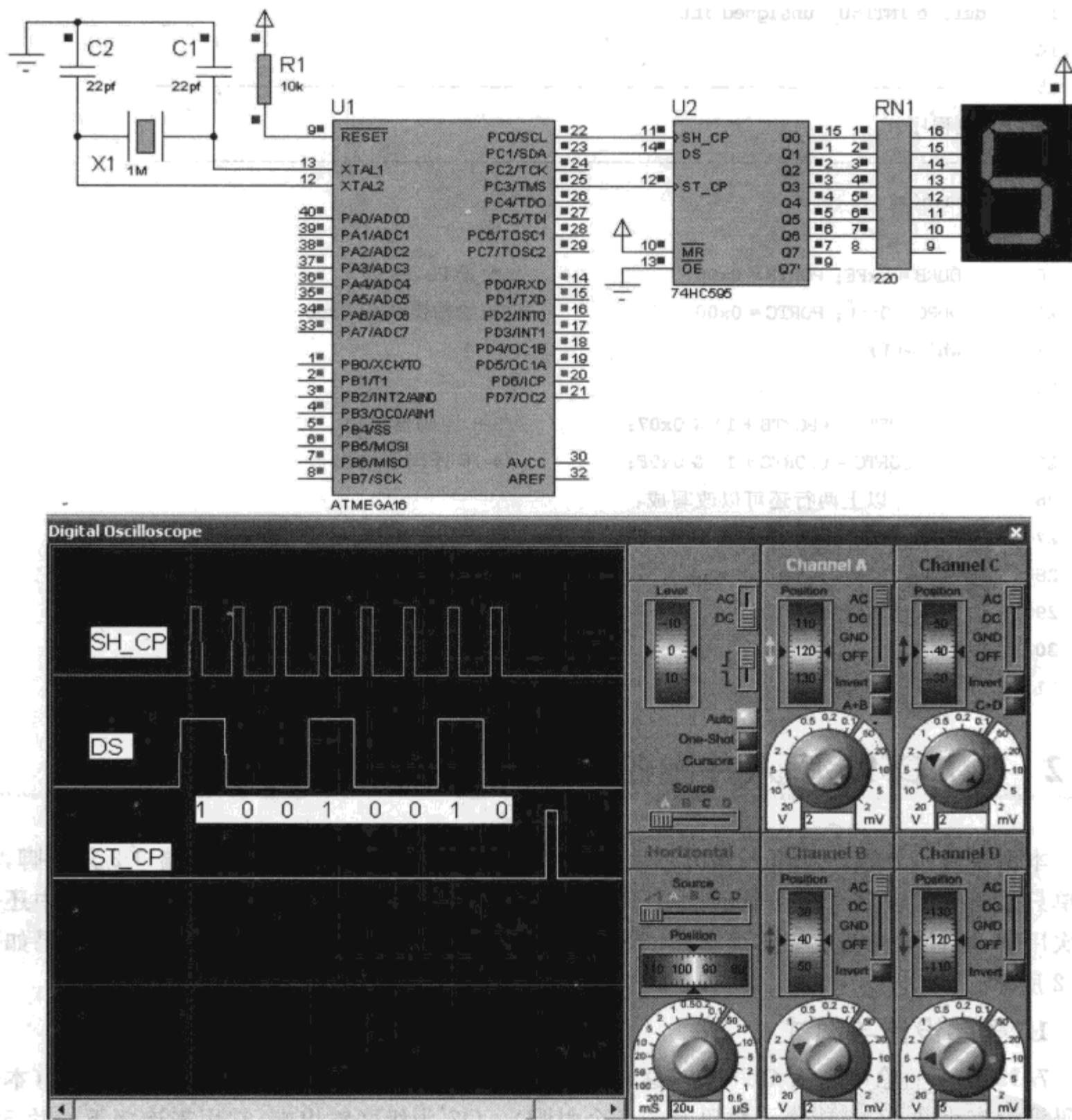


图 4-2 74HC595 串入并出芯片应用

- ④ \overline{MR} (10 脚)在低电平时将移位寄存器数据清零,本例中该引脚直接连接 Vcc。
- ⑤ \overline{OE} (13 脚)在高电平时禁止输出(高阻态),本例中该引脚接地。

75HC595 其主要优点是能锁存数据,在移位过程中输出端的数据保持不变,这有利于使数码管在串行速度较慢的场合不会出现闪烁感。

图 4-2 中所示虚拟示波器的 A、B、C 通道与 SH_CP、DS、ST_CP 引脚对应,当前显示的波形与显示数字“5”(段码为 0x92,即 10010010)的操作时序对应,其中 A、B 通道波形与函数 Serial_Input_595 对应,该函数向 DS 引脚发送数据与并向 SH_CP 引脚输出移位时钟。通道 C 的波形与函数 Parallel_Output_595 对应,在完成一个字节发送后,向 ST_CP 引脚输入锁存

脉冲，在脉冲上升沿将所输入的字节送到输出锁存器。

本例的重要函数 Serial_Input_595 通过 for 循环在 SH_CP 引脚模拟输出 8 个时钟周期，将一个字节由高到低逐位通过 DS 线串行移入 595。该函数的编写模式对其他串行器件的数据写入代码编写都有参考作用，大家要熟练掌握。

2. 实训要求

- ① 思考源程序中第 42 行为什么可以省略，移到 for 循环后面又有什么作用？
- ② 再添加 1 片 74HC595 和 1 只数码管，将 2 片 74HC595 级联，仍使用 PC 端口 3 只引脚，实现对两只独立数码管的显示控制。
- ③ 重新修改本例电路与程序，用两片 595 芯片分别控制 8 位集成式七段数码管的段码与位码，以静态或滚动方式显示指定数据信息。

3. 源程序代码

```

01 //-----
02 // 名称：74HC595 串入并出芯片应用
03 //-----
04 // 说明：74HC595 具有一个 8 位串入并出的移位寄存器和一个 8 位输出锁存器
05 //          本例使用 74HC595，通过串行输入数据来控制数码管显示
06 //
07 //-----
08 #define F_CPU 1000000UL
09 #include <avr/io.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //595 引脚定义
15 #define SH_CP PC0           //移位时钟脉冲
16 #define DS    PC1           //串行数据输入
17 #define ST_CP PC3           //输出锁存器控制脉冲
18
19 //595 引脚操作定义
20 #define SH_CP_0() PORTC &= ~_BV(SH_CP)
21 #define SH_CP_1() PORTC |= _BV(SH_CP)
22 #define DS_0()   PORTC &= ~_BV(DS)
23 #define DS_1()   PORTC |= _BV(DS)
24 #define ST_CP_0() PORTC &= ~_BV(ST_CP)
25 #define ST_CP_1() PORTC |= _BV(ST_CP)
26
27 //数码管段码表
28 const INT8U SEG_CODE[] =
29 {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90},
30 //-----

```



```
31 // 串行输入子程序
32 //-----
33 void Serial_Input_595(INT8U dat)
34 {
35     INT8U i;
36     for(i = 0; i < 8; i++)
37     {
38         if (dat & 0x80) DS_1(); else DS_0(); //发送高位
39         dat <<= 1; //次高位左移到高位
40         SH_CP_0(); _delay_us(2); //移位时钟线拉低
41         SH_CP_1(); _delay_us(2); //放在 DS 线的 0 或 1 在移位时钟脉冲上升沿被移入 595
42         SH_CP_0(); _delay_us(2); //本行可以省略,也可移到 for 循环后面
43     }
44 }
45
46 //-----
47 // 并行输出子程序
48 //-----
49 void Parallel_Output_595()
50 {
51     ST_CP_0(); _delay_us(1);
52     ST_CP_1(); _delay_us(1); //上升沿将数据送到输出锁存器
53     ST_CP_0(); _delay_us(1);
54 }
55
56 //-----
57 // 主程序
58 //-----
59 int main()
60 {
61     INT8U i = 0;
62     DDRC = 0xFF; //PC 端口设为输出
63     while (1)
64     {
65         for(i = 0; i < 10; i++)
66         {
67             //将数字 i 的段码字节串行输入 595
68             Serial_Input_595(SEG_CODE[i]);
69             //595 移位寄存器数据传输到存储寄存器并出现在输出端
70             Parallel_Output_595();
71             _delay_ms(300);
72         }
73     }
74 }
```

4.3 用 74LS148 与 74LS21 扩展中断

本例所选用单片机的 PD2(INT0)、PD3(INT1)、PB2(INT2)用于输入外部中断信号，当需要对更多的外部中断信号作出响应时就需要进行中断扩展了。实现中断扩展的方法较多，本例使用 8-3 编码芯片 74LS148 实现中断扩展，8 路外部中断信号可按优先级进行处理。另外，本例还利用具有双四路输入的与门芯片 74LS21 扩展中断，两者的差别将在程序设计与调试部分中阐述。本例电路及部分运行效果如图 4-3 所示。

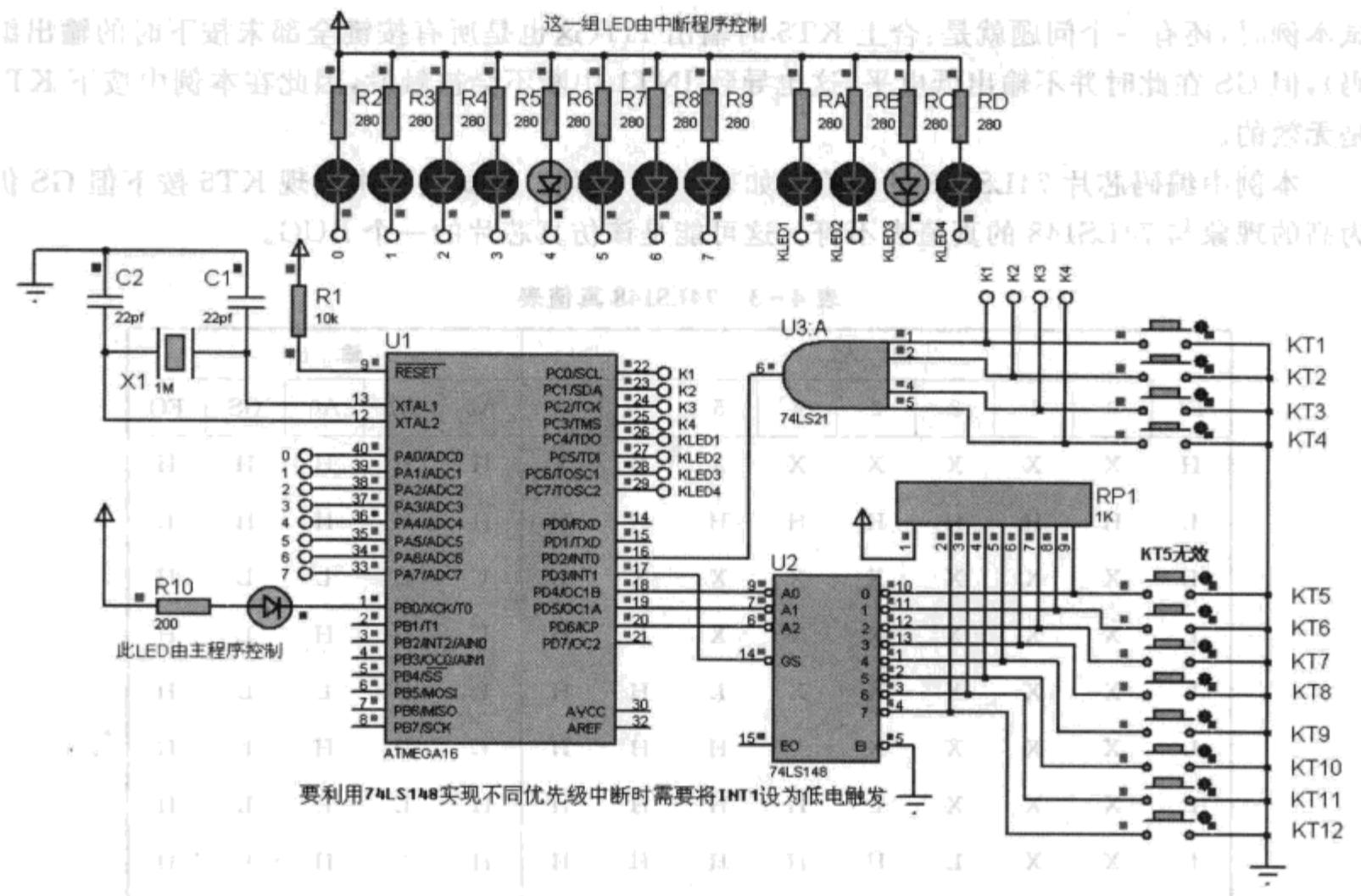


图 4-3 用 74LS148 与 74LS21 扩展中断

1. 程序设计与调试

74LS148 是带优先级的 8-3 编码芯片，对于外部的 8 路数据输入线，只要有 1 路或几路被置为 0，编码芯片即会按由高到低的优先级进行编码，并由 A2~A0 引脚输出 3 位二进制数，而且 GS 引脚会自动变为 0。在没有任何输入、8 路数据线均为高电平时，GS 自动变为 1。

本例将 GS 连接单片机的 PD3(INT1)引脚，当 GS 为 0 时即会触发 INT1 中断，中断程序根据 A2~A0 引脚输入的 3 位二进制编码执行相应操作。

由于 74LS148 是带优先级的，按键 KT12~KT5 模拟的中断级别由高到低，如果单击 KT9 右边的红色双向箭头将 KT9 按下锁住，这时再按下 KT5~KT8 中的任何按键，8-3 编码器的输出都不会发生变化，只有按下 KT10~KT12 时输出的 3 位编码才会变化。

在调试本例时会发现，如果将 KT9 锁住（保持按下状态），这时按下更高级别的按键，虽然输出的 3 位编码会发生变化，但对应的指示 LED 却没有移动，这是因为本例设 INT1 为下降

沿触发,如果希望有低级别按键按下且锁定时,按下高级别按键还能立即触发 INT1 中断,这要通过设 MCUCR=0x02 将 INT1 配置为低电平触发。

这样设置后,即使低级别按键未释放,高级别按键事件也会马上被处理,指示 LED 会立即变化。如果释放高级别按键,LED 会立即回到原位,除非低级别按键也释放了。

在设为低电平触发后,大家又会发现另一个问题,那就是左边由主程序控制的 LED 不能正常闪烁了。这是因为设为低电平触发后,只要有按键没有释放,INT1 的中断程序就会处于无限次调用之中,主程序中控制 LED 闪烁的语句也就没有足够的时间执行了。

对于 8-3 编码器,0~7 号引脚按键按下时,输出编码为 111~000(不是 000~111)。在调试本例时,还有一个问题就是:合上 KT5 时输出 111(这也是所有按键全部未按下时的输出编码),但 GS 在此时并不输出低电平,这也导致 INT1 中断不会被触发,因此在本例中按下 KT5 是无效的。

本例中编码芯片 74LS148 的真值表如表 4-3 所列。调试过程中发现 KT5 按下但 GS 仍为高的现象与 74LS148 的真值表不符。这可能是该仿真芯片的一个 BUG。

表 4-3 74LS148 真值表

输入									输出				
EI	0	1	2	3	4	5	6	7	A2	A1	A0	GS	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	X	X	X	X	X	X	X	X	L	L	L	L	H
L	X	X	X	X	X	X	X	L	H	L	L	L	H
L	X	X	X	X	X	X	L	H	H	L	H	L	H
L	X	X	X	X	L	H	H	H	L	H	H	L	H
L	X	X	X	L	H	H	H	H	H	L	L	L	H
L	X	X	L	H	H	H	H	H	H	L	H	L	H
L	X	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

对于另一组中断扩展,电路中使用了双四路输入的 74LS21 与门芯片(本例电路中只用了“半片”74LS21),按键 KT1~KT4 右端接地,左端接与门输入端,且全部由 PC 端口高 4 位内部上拉(设为输入),显然,KT1~KT4 中任何一个按键按下,与门输出端都会向 PD2(INT0) 引脚输入 0,触发 INT0 中断,INT0 中断程序通过读取 PC 端口(PINC)高 4 位即可知道是哪一个按键触发中断,这种设计不具有 8-3 编码器所具有的优先级,占用的引脚数也更多。

2. 实训要求

- ① 将 INT1 配置为低电平触发,使多路按键按下时能按不同优先级作出响应。模拟多路按键按下时,可先单击一个或多个按键右上角的红色双向箭头将其按下并锁住。
- ② 使用整片 74LS21 实现对外部 8 路中断的处理(输出端占用 INT0 与 INT1)。
- ③ 搜索 Proteus 芯片库,选用其他数字芯片实现中断扩展。

3. 源程序代码

```

01 //-----
02 // 名称：用 74LS148/74LS21 扩展中断
03 //-----
04 // 说明：本例利用 74LS148 扩展外部中断，对于外部的 8 个控制开关，任意
05 // 一个开关合上都将在 GS 引脚输出低电平，触发外部中断，优先级最
06 // 高的是输入引脚 7，最低的是输入引脚 0。中断触发后，中断例程通过
07 // 读取 A2、A1、A0 的输出，判断是哪一路按键触发中断
08 //
09 // 对于 74LS21，任何一个按键都会触发中断，它并没能真正实现中断
10 // 扩展，而是仅利用了 INT0，省去了对多个按键的轮询判断
11 //
12 //-----
13 #include <avr/io.h>
14 #include <avr/interrupt.h>
15 #include <util/delay.h>
16 #define INT8U unsigned char
17 #define INT16U unsigned int
18
19 //此 LED 由主程序控制
20 #define LED_BLINK() PORTB ^= _BV(PB0)
21 //-----
22 // 主程序
23 // 说明：由于 Proteus 中 74LS148 存在问题，与输入引脚 0 对应的开关控制无效
24 //-----
25 int main()
26 {
27     DDRA = 0xFF; PORTA = 0xFF;
28     DDRB = 0xFF; PORTB = 0xFF;
29     DDRC = 0xF0; PORTC = 0xFF;          //PC 端口低 4 位输入，高 4 位输出
30     DDRD = 0x00; PORTD = 0xFF;
31     MCUCR = 0x0A;                      //INT0、INT1 中断下降沿触发
32     GICR = 0xC0;                       //INT0、INT1 中断许可
33     sei();                            //开总中断
34     while(1)
35     {
36         LED_BLINK();                  //主程序控制一只 LED 闪烁
37         _delay_ms(100);             //延时
38     }
39 }
40
41 //-----

```



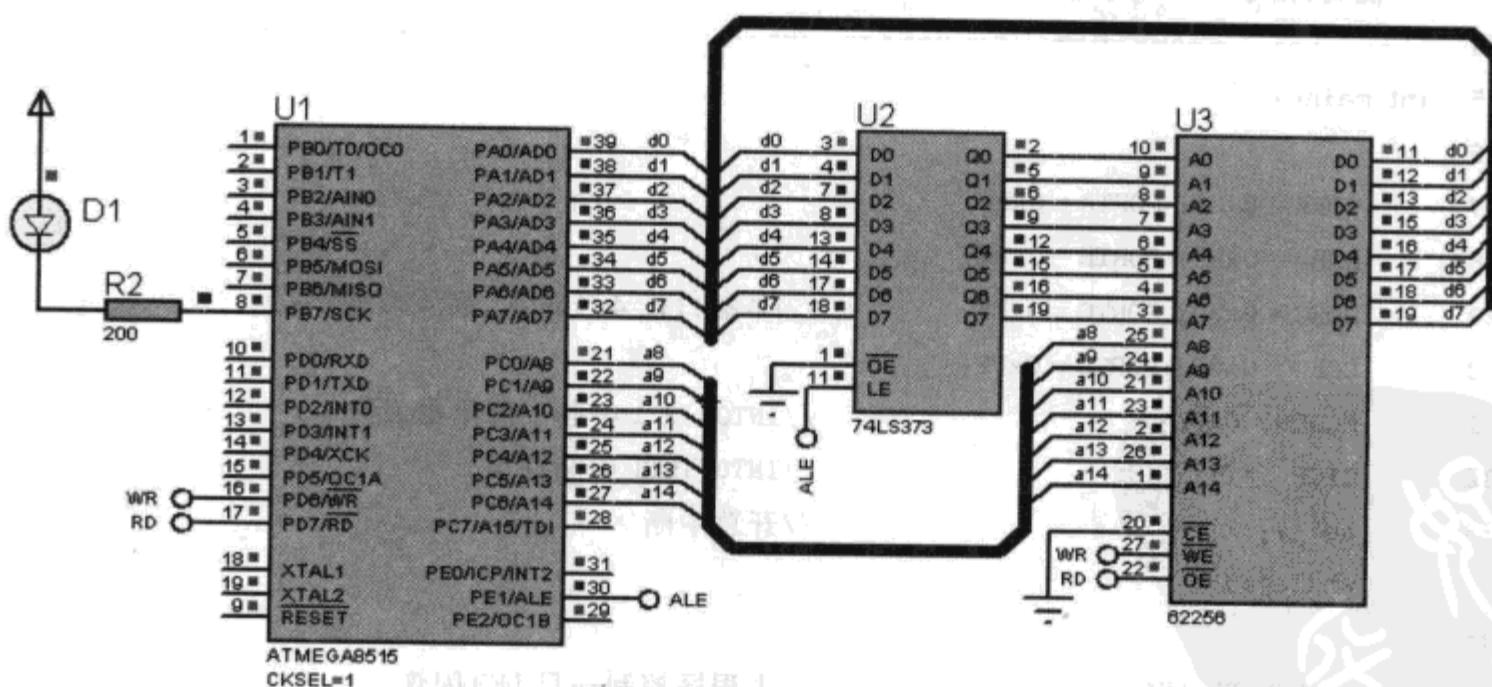
```

42 // INT0 中断服务程序(4 个按键中任何一个按下时都会触发 INT0 中断)
43 //-----
44 ISR (INT0_vect)
45 {
46     PORTC = PINC << 4 | 0x0F;           // "0x0F" 用于保持 PC 低 4 位内部上拉
47 }
48
49 //-----
50 // INT1 中断服务程序(当有按钮按下时,GS 为零,触发 INT1 中断)
51 //-----
52 ISR (INT1_vect)
53 {
54     INT8U bidx = (PIND >> 4) & 0x07;    // 得到按键编号
55     PORTA = ~_BV(bidx);                  // 点亮对应的 LED
56 }

```

4.4 62256 扩展内存实验

本例给出了 ATMEGA8515 单片机外部内存扩展电路。所使用的是 62256SRAM 存储器,该芯片共有地址线 15 根,可提供 $2^{15} = 32K$ 字节空间,提供地址锁存的是 74LS373,它是常用的地址锁存器芯片,其实质是一个带三态缓冲输出的 8D 触发器。本例演示了内存扩展芯片 62256 的读/写实验,这种扩展对学习后续案例中有关接口扩展的案例也有很好的参考作用。案例电路如图 4-4 所示。



注:LED点亮时数据传输开始,闪烁时表示读写完成,
此时可暂停运行,打开DEBUG中的Memory Contents,
查看62256内存数据。

图 4-4 62256 扩展内存实验

1. 程序设计与调试

设计本例仿真电路时,要掌握三总线(CB、AB、DB)的连接方法,本例中控制总线涉及ALE、 \overline{RD} 、 \overline{WR} ;对于由PA、PC端口提供的16位地址总线A0~A15,本例使用了A0~A14,数据总线则复用了PA端口的D0~D7。在程序设计方面,应熟练掌握MCUCR中SRE位的设置及外部内存地址定义等:

① 仿真电路的74LS373、62256与AVR单片机的连接。其中,单片机ALE引脚(Address Latch Enable,地址锁存使能)与74LS373的LE(Latch Enable)引脚的连接,74LS373地址锁存由单片机ALE引脚控制。单片机读/写控制引脚 \overline{RD} 、 \overline{WR} 与62256的OE(Output Enable,输出使能)、 \overline{WE} (Write Enable,写使能)连接。这3条控制总线引脚负责地址锁存及读/写控制。

② 为了访问外部扩展内存,一定要在主程序内将ATMEGA8515单片机MCUCR寄存的最高位SRE(External SRAM/XMEM Enable)置位,这样才能访问外部内存(或访问外部扩展接口地址)。在ATMEGA系列中,8515/64/128等单片机提供了三总线以扩展外部内存或接口,但8/16/32等单片机则没有提供扩展总线。

③ 外部内存地址(或接口地址)访问定义。本例中定义为:

```
#define EXTMEM_ADDR (INT8U *)0x8000
```

62256地址线共有15根,所定义的0x8000超出内部SRAM地址空间,指向某个外部内存地址,0x8000即1000000000000000地址,后面共15个0,它们与62256的15条地址线对应,高位的1与62256无关,当从0x8000地址开始读/写时,实际上是从62256的0地址开始读/写。

④ 主程序中第32行和37行对外部内存进行读/写,语句如下:

```
* (EXTMEM_ADDR + i) = i + 1; //写操作  
* (EXTMEM_ADDR + i + 0x0100) = * (EXTMEM_ADDR + 199 - i); //读/写操作
```

在搞清楚上述内容后,对程序所完成的其他任务就容易理解了。程序运行时首先向62256开始处写入1~200,接着读取这些数,并将其逆向写到62256内存中0x0100开始的位置。

前面已经提到了单片机的 \overline{WR} 与 \overline{RD} 引脚,单片机通过这两只引脚对读/写时序进行自动管理,删除 \overline{WR} 连线时会出现写入失败,删除 \overline{RD} 连线时会导致读取失败。

本例程序完成对外部SRAM的读/写操作后,LED开始闪烁。如果要观察62256芯片内的数据,可按下Pause按钮暂停程序,然后单击Debug菜单,打开Memory Contents即可观察到图4-5所示窗口中显示的内存数据。

2. 实训要求

① 重新编写程序,向62256写入1001~1200共200个整数。这些整数不能用200个单字节来保存,因为它们已经超过了INT8U类型的最大值255,这时所占空间应为400个字节。编程时注意定义指针类型。

② 在Proteus中搜索ROM存储芯片对单片机外部ROM进行扩展,将固定数据绑定到该芯片后,在程序中读取外部ROM中的数据并通过虚拟终端显示。

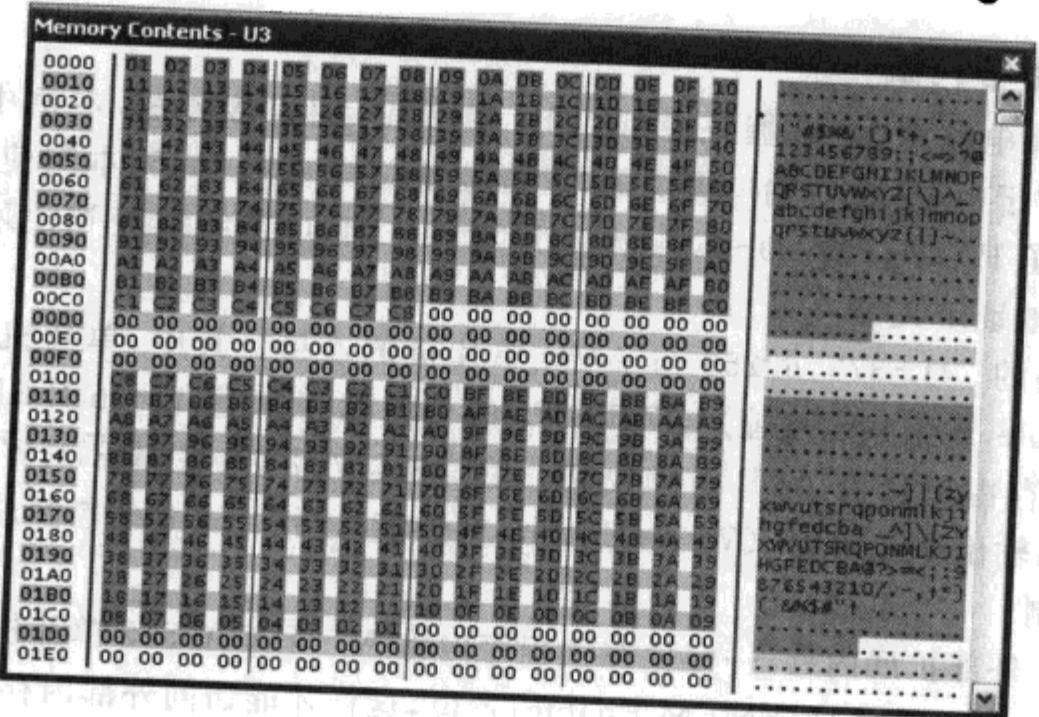


图 4-5 62256 内存内容

3. 源程序代码

```

01 //-----
02 // 名称：用 62256 扩展内存(32 KB)
03 //-----
04 // 说明：程序运行时首先向 62256 开始处写入 1~200，然后读取这些数据，并将
05 // 其逆向写到 62256 内存中 0x0100 开始位置
06 //-----
07 //-----
08 #define F_CPU 1000000UL
09 #include <avr/io.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //外部内存地址定义
15 #define EXTMEM_ADDR (INT8U*)0x8000
16 //LED 控制
17 #define LED_ON() (PORTB &= ~_BV(PB7)) //LED 点亮
18 #define LED_BLINK() (PORTB ^= _BV(PB7)) //LED 闪烁
19 //-----
20 // 主程序
21 //-----
22 int main()
23 {
24     INT8U i;
25     DDRB = 0xFF; PORTB = 0xFF;

```

```

26     LED_ON(); _delay_ms(1000);
27     //允许访问外部存储器
28     MCUCR |= 0x80;
29     //向 62256 的 0x0000 地址开始写入 1~200
30     for (i = 0; i < 200; i++)
31     {
32         *(EXTMEM_ADDR + i) = i + 1;
33     }
34     //将 62256 中的 1~200 逆向拷贝到 0x0100 开始处
35     for (i = 0; i < 200; i++)
36     {
37         *(EXTMEM_ADDR + i + 0x0100) = *(EXTMEM_ADDR + 199 - i);
38     }
39     //扩展内存数据读/写操作完成后 LED 闪烁
40     //这时可暂停 Proteus, 打开菜单 Debug/Memory Contents 查看数据
41     while (1)
42     {
43         LED_BLINK();
44         _delay_ms(200);
45     }
46 }

```

4.5 用 8255 实现接口扩展

本例利用 ATMEGA8515 的三总线,通过 8255 接口扩展芯片控制 8 只集成式七段数码管显示,在 8255 的 PC 端口还添加有 3 个按键,用于调节所显示时间数据。仿真本例时要注意给 8255 单独添加 VDD 引脚。本例电路及部分运行效果如图 4-6 所示。

1. 程序设计与调试

本例的接口扩展电路与上一案例中的数据内存扩展电路非常相似,都使用了地址锁存芯片 74LS373。单片机的控制引脚 ALE、RD、WR 连接方法也与上一案例类似。

正是因为本例的接口扩展电路与上一案例非常类似,上一案例中扩展内存的地址访问方法同样可以应用到本例中的扩展接口的地址访问上。

表 4-4 列出了 8255 的基本操作,通过仔细对比表格与本例电路即可得出 8255 的 3 个 I/O 端口和 1 个命令端口的定义。由于 8255 的接口地址仅需要单片机地址端口的高 8 位控制,这 8 位地址中实际仅使用了低 3 位,它们分别对应 CS、A1、A0,其中 A1 与 A0 地址线可选择 8255 的 4 个端口地址之一。

以 PB 端口为例,由于 A1/A0 为 01,且 CS 为 0,则地址可定义为 11111111 00000001(定义中将未使用的高 8 位地址全部设为 1),由此可得 8255PB 端口地址为 0xFF01。在向 8255PB 端口写入数据时,单片机会自动将 WR 置为低电平,读 8255PB 端口数据时单片机则自动将 RD 置为低电平。

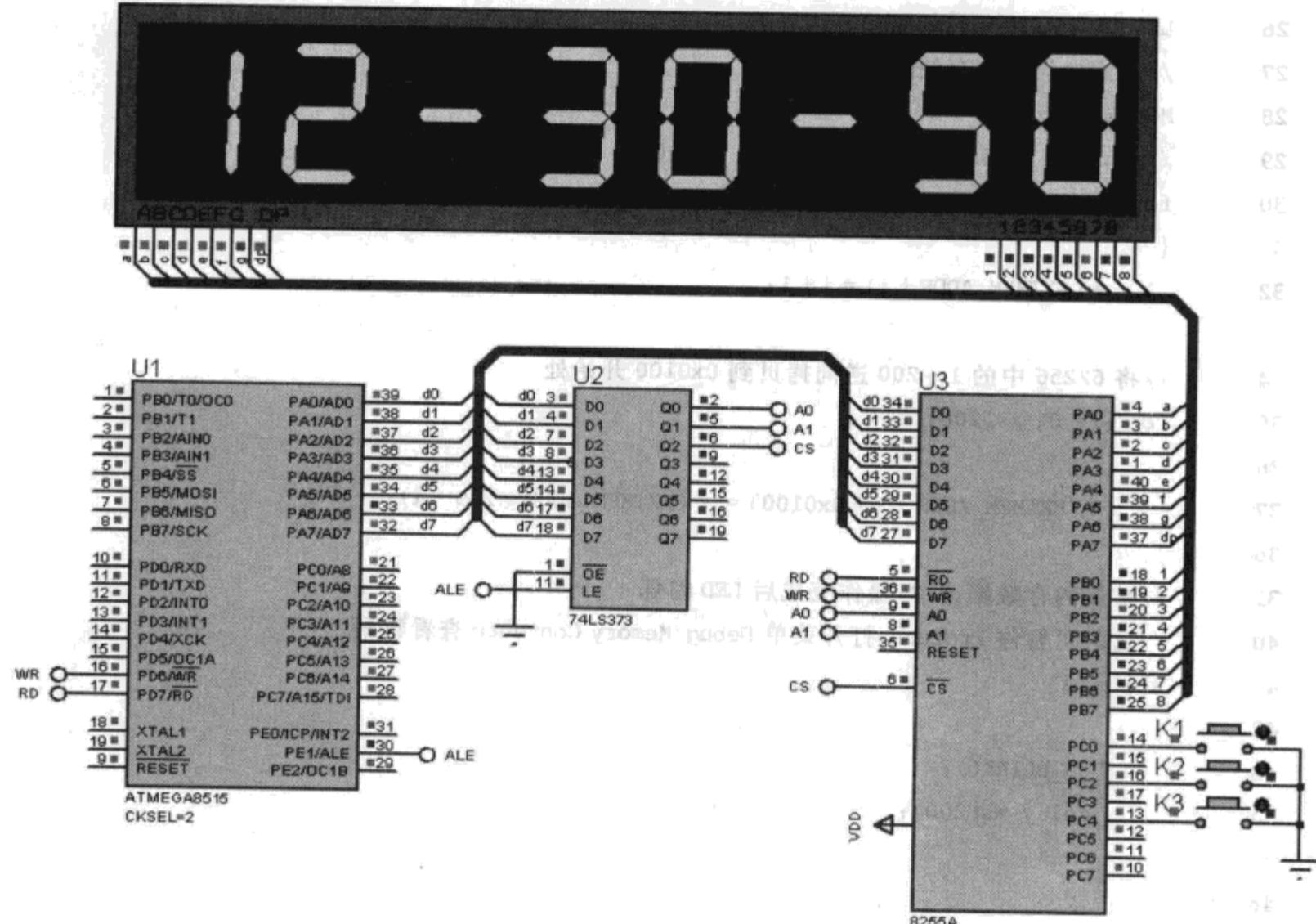


图 4-6 用 8255 实现接口扩展

以下是 8255 的 PA、PB、PC 及命令端口的地址定义：

```
#define PA (INT8U *)0xFF00
#define PB (INT8U *)0xFF01
#define PC (INT8U *)0xFF02
#define COM (INT8U *)0xFF03
```

表 4-4 8255 的基本操作

操作	A1	A0	CS	RD	WR	说明
输入(读)	0	0	0	0	1	PA→数据总线
	0	1	0	0	1	PB→数据总线
	1	0	0	0	1	PC→数据总线
	1	1	0	0	1	控制字→数据总线
输出(写)	0	0	0	1	0	数据总线→PA
	0	1	0	1	0	数据总线→PB
	1	0	0	1	0	数据总线→PC
	1	1	0	1	0	数据总线→控制字

8255 命令口对工作方式的设置可参阅 8255 芯片的技术手册文件。图 4-7 给出了 8255 工作模式字节格式及本例所选择的设置。本例选择模式 0, 8255 工作于基本 I/O 模式, 使用 PA 和 PB 端口输出控制数码管显示, PC 端口则用于读取按键状态并进行相应处理, 各位配置如图 4-7 右半部分所示。

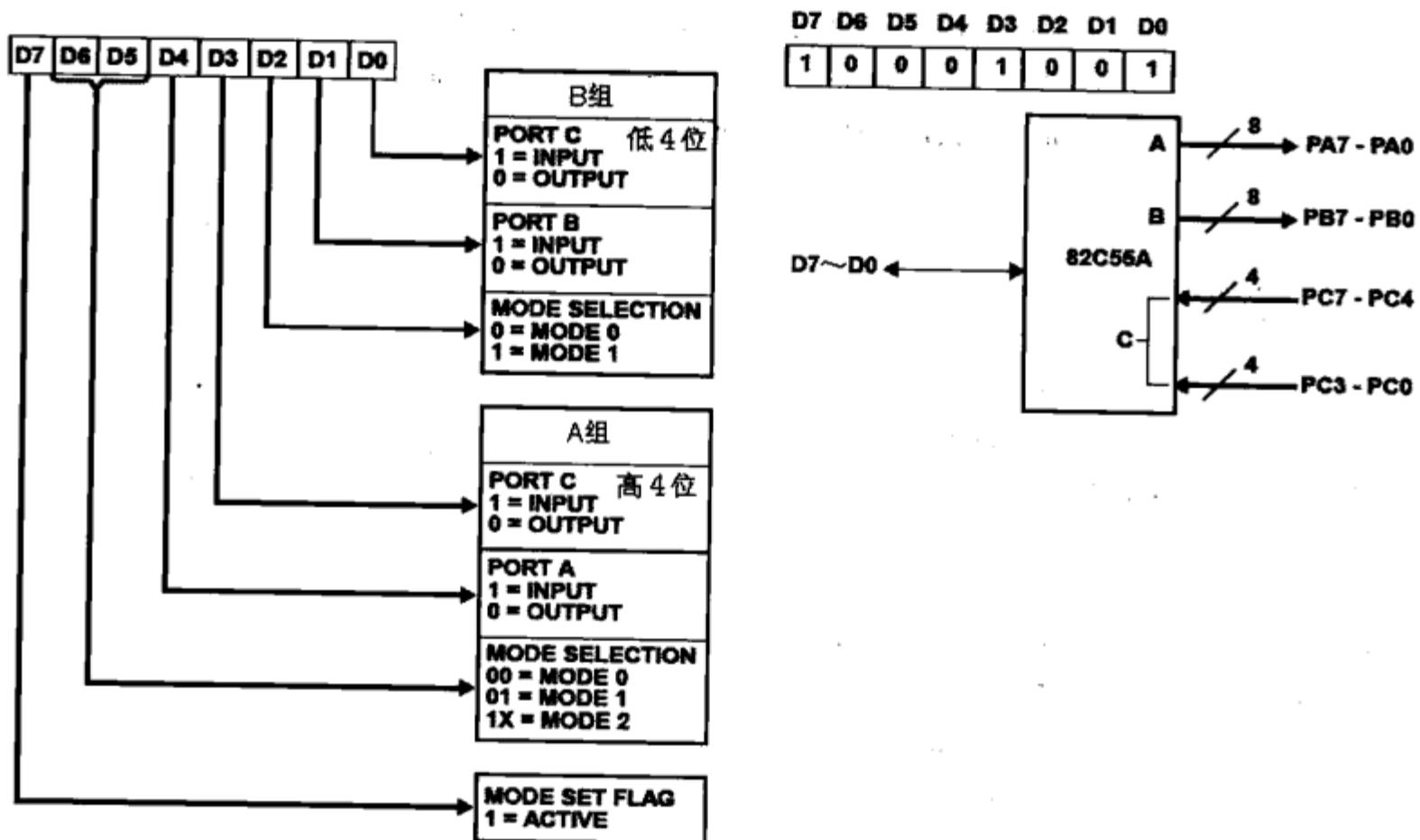


图 4-7 8255 工作模式字节格式(左)及本例设置(右)

在完成相关定义与配置后, 其他操作与扩展内存的操作就很相似了:

① 源程序第 67 行向命令口 COM 写控制字节, 实现对 8255 工作方式的配置:

* COM = 0B10001001;

② 第 73、74 行通过 PB、PA 端口输出位码与段码, 控制数码管扫描显示;

* PB = _BV(i);

* PA = (INT8U)SEG_CODE[Disp_Buffer[i]];

③ 第 34 行读入 8255PC 端口的按键状态, 以便分别进行时分秒的调节:

Key_State = * PC;

2. 实训要求

① 重新设计本例电路, 再加一组相同的 8 位数码管, 用 PA 控制两组数码管段码, PB 与 PC 用于控制 16 位的扫描码。在两组数码管上同时显示出年月日和时分秒信息。

② 保持本例的电路设计, 仅将 PC 端口按键改成 4×4 键盘矩阵, 利用键盘矩阵控制数码管显示、关闭及时分秒调节等自定义功能。

3. 源程序代码

```

01 //-
02 // 名称：用 8255 实现接口扩展
03 //-
04 // 说明：8255 的 PA、PB 端口分别连接 8 位数码管的段码和位码
05 // PC 端口连接 3 只按键，正常运行时数码管显示一组时间值
06 // PC 端口的 3 只按键可对时间值的各部分分别进行调整
07 //
08 //-
09 #define F_CPU 2000000UL
10 #include <avr/io.h>
11 #include <util/delay.h>
12 #define INT8U unsigned char
13 #define INT16U unsigned int
14
15 //PA,PB,PC 端口及命令端口地址定义
16 #define PA (INT8U *)0xFF00
17 #define PB (INT8U *)0xFF01
18 #define PC (INT8U *)0xFF02
19 #define COM (INT8U *)0xFF03
20
21 //0~9 的共阳数码管段码表，最后的 0xBF 表示“-”
22 const INT8U SEG_CODE[] =
23 { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF };
24 //待显示信息缓冲 12 - 30 - 50
25 INT8U Disp_Buffer[] = {1,2,10,3,0,10,5,0};
26 //上次按键状态
27 INT8U Pre_Key_State = 0x00;
28 //-
29 // 8255PC 端口按键处理
30 //-
31 void Key_Process()
32 {
33     INT8U Key_State, t;
34     Key_State = *PC;                                //读 8255PC 端口按键状态
35     if (Key_State == Pre_Key_State) return;
36     Pre_Key_State = Key_State;
37     switch (Key_State)
38     {
39         case (INT8U)~_BV(0):                         //K1: 小时递增

```

```

40     t = Disp_Buffer[0] * 10 + Disp_Buffer[1];
41     if (++t == 24) t = 0;
42     Disp_Buffer[0] = t / 10;
43     Disp_Buffer[1] = t % 10;
44     break;
45 case (INT8U)~_BV(2);           //K2:分钟递增
46     t = Disp_Buffer[3] * 10 + Disp_Buffer[4];
47     if (++t == 60) t = 1;
48     Disp_Buffer[3] = t / 10;
49     Disp_Buffer[4] = t % 10;
50     break;
51 case (INT8U)~_BV(4);           //K3:秒数递增
52     t = Disp_Buffer[6] * 10 + Disp_Buffer[7];
53     if (++t == 60) t = 1;
54     Disp_Buffer[6] = t / 10;
55     Disp_Buffer[7] = t % 10;
56     break;
57 default: break;
58 }
59 }
60
61 //-----
62 // 主程序
63 //-----
64 int main()
65 {
66     INT8U i;
67     MCUCR |= 0x80;          //允许访问外部存储器/接口等
68     *COM = 0B10001001;      //8255 工作方式选择:工作于方式 0,PA、PB 输出,PC 输入
69     while(1)
70     {
71         for(i = 0; i<8; i++)           //数码管显示
72         {
73             *PB = _BV(i);            //向 PB 端口发送位码
74             *PA = (INT8U)SEG_CODE[ Disp_Buffer[i] ]; //向 PA 端口发送段码
75             _delay_ms(2);
76             Key_Process();           //PC 端口按键处理
77         }
78     }
79 }

```

4.6 可编程接口芯片8155应用

可编程接口芯片8155内含256字节RAM存储器、2个可编程的8位并行端口、1个6位并行端口及1个14位的定时/计数器。本例用8155的PA与PB端口控制数码管显示，PC端口连接按键，案例演示了8155控制数码管显示，通过按键调整定时初值、启/停8155定时器，用定时器中断触发蜂鸣器，以及写8155内存等。本例电路及部分运行效果如图4-8所示。

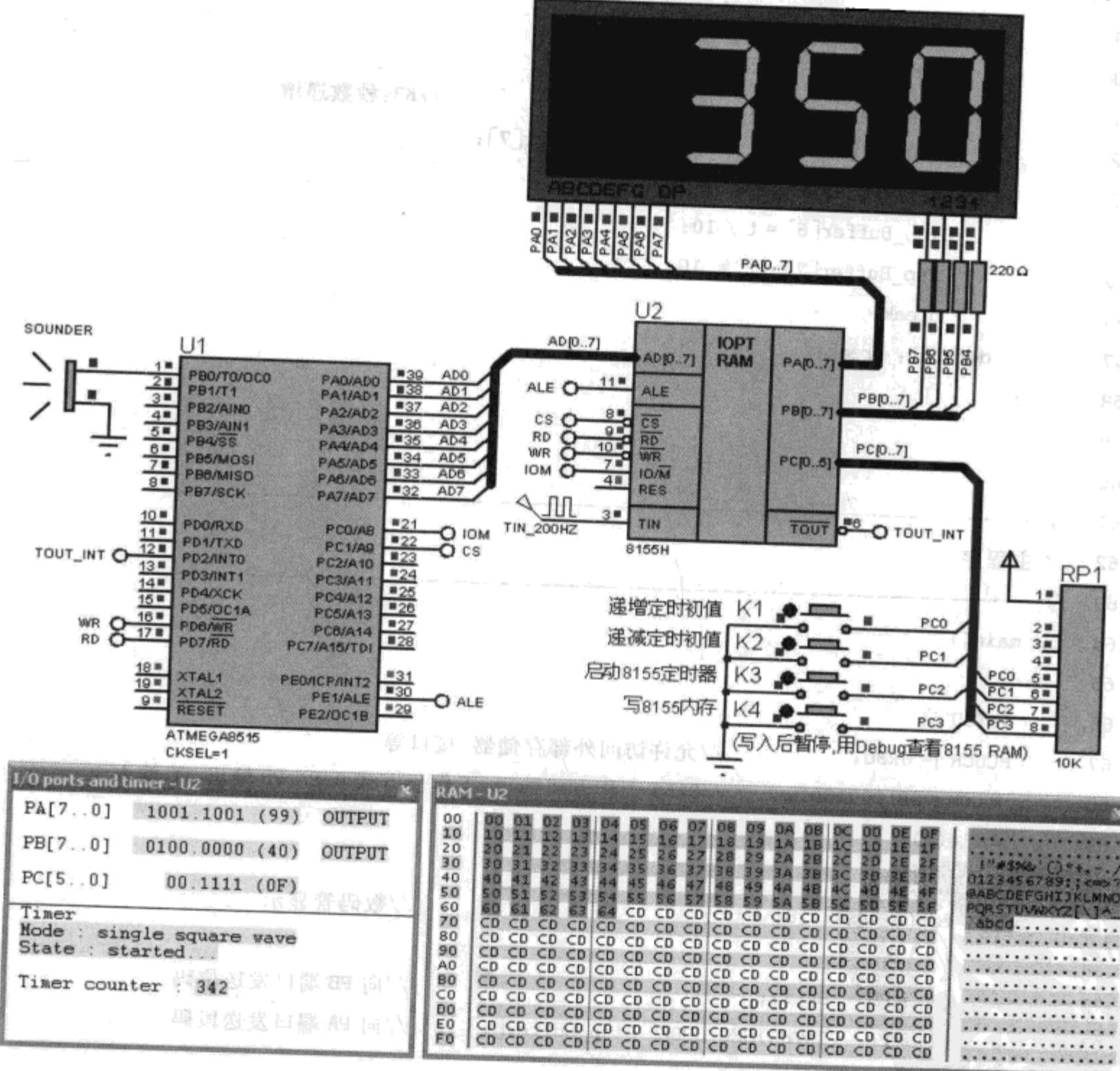


图4-8 可编程接口芯片8155应用

1. 程序设计与调试

图4-8所示电路中，8155的AD[0:7]为三态数据/地址线，TIN是定时/计数器输入引脚，TOUT是定时器输出引脚，可以是方波或脉冲波形。IO/M是I/O与RAM选择线，设为1

时选择 I/O, 设为 0 时选择 RAM。其他引脚与 8255 类似。

本例程序重点在于以下地址定义:

```
#define COMM_8155      (INT8U *)0xFD00    //命令字端口
#define PA_8155         (INT8U *)0xFD01    //PA 端口地址
#define PB_8155         (INT8U *)0xFD02    //PB 端口地址
#define PC_8155         (INT8U *)0xFD03    //PC 端口地址
#define CONT_8155_L8    (INT8U *)0xFD04    //计数器低 8 位地址
#define CONT_8155_H8    (INT8U *)0xFD05    //计数器高 6 位 + 2 位方式地址
#define PMEM_8155       (INT8U *)0xFC00    //8155RAM 地址
```

单片机 PC 端口提供地址的高 8 位, 其中 PC7~PC2 未用, 定义中将它们全部设为 1, PC1 连接的 CS 位设为 0, PC0 对应的 IO/M 分别取 0/1, 因此上述地址高 4 位定义中, 除最后的 PMEM 定义为 0xFC 以外, 其他全部为 0xFD。其他地址定义可根据表 4-5 所列的 8155 内部 I/O 地址表得到。

完成地址定义后, 还需要根据 8155 命令字对端口及定时器进行配置管理, 程序中第 56 行与 123 行对定时/计数器进行设置, 并对端口进行管理。8155 命令字格式如表 4-6 所列。

表 4-5 8155 内部 I/O 地址表

A2	A1	A0	I/O 端口
0	0	0	命令端口
0	0	1	PA 端口
0	1	0	PB 端口
0	1	1	PC 端口
1	0	0	定时器低 8 位
1	0	1	定时器高 6 位及方式

表 4-6 8155 命令字格式

TM2	TM1	IEB	IEA	PC2	PC1	PB	PA

完成地址定义后, 还需要根据 8155 命令字对端口及定时器进行配置管理, 程序中第 56 行与 123 行的设置都向 8155 写入了命令字, 其中:

第 123 行设 *COMM_8155=0B00000011, 其中 TM2/TM1 为 00, 定时器空操作。同时低 4 位 0011 设置 PA 与 PB 端口为输出, PC 端口为输入。

第 56 行设 *COMM_8155=0B11000011, 它将 TM2/TM1 设为 11, 在装入定时器方式和初值后立即启动计数。PA、PB、PC 端口配置不变。

本例运行时:

按下 K4 可向 8155RAM 中写入 0~100(0x00~0x64), 在暂停程序后可通过 Proteus 的 Debug 菜单下的 RAM 菜单查看 8155RAM 数据, 如图 4-8 右下角所示。

按下 K3 时可启动定时器, 程序已经给 14 位的定时器设置了固定初值, 定时溢出时, 8155 的 TOUT 引脚触发单片机 INT0 中断, 输出报警声音, 同时还原定时初值, 使中断能在同样时间后继续触发。

K1 与 K2 按键则用于改变 8155 定时器初值, 在不同定时初值定义下, 中断的触发间隔不同, 这通过报警声音输出的间隔就可以分辨出来。

在暂停程序运行时, 按下 Debug 菜单下的 I/O ports and timer 菜单可查看 I/O 端口与定时器配置及工作状态, 如图 4-8 左下部分窗口所示。



2. 实训要求

① 修改本例程序,对 TIN 引脚输入的脉冲进行计数(改用按键或低频率脉冲),并将计数值显示在 4 位数码管上。

② 在实现计数的基础上进一步修改程序,当计数值每次累加达到 5000 时,将当前计数值累加到 8155RAM 中的指定地址,然后再从 0 开始累加计数。

3. 源程序代码

```
001 //-----  
002 // 名称: 可编程序接口芯片 8155 应用  
003 //-----  
004 // 说明: 本例利用 8155 的 PA、PB 连接数码管, 显示 8155 当前定时初值  
005 // PC 端口连接按键, 分别用于调整定时初值, 启动定时器, 写 8155RAM 等  
006 // 启动定时器后, 在定时溢出时 8155 TOUT 将触发 INT0 中断, 输出提示音  
007 // 在调节的定时初值不同时, 声音输出的间隔也不同  
008 //  
009 //-----  
010 #define F_CPU 2000000UL  
011 #include <avr/io.h>  
012 #include <avr/interrupt.h>  
013 #include <util/delay.h>  
014 #define INT8U unsigned char  
015 #define INT16U unsigned int  
016  
017 //8155 地址定义  
018 #define COMM_8155 (INT8U *)0xFD00 //命令字端口  
019 #define PA_8155 (INT8U *)0xFD01 //PA 端口地址  
020 #define PB_8155 (INT8U *)0xFD02 //PB 端口地址  
021 #define PC_8155 (INT8U *)0xFD03 //PC 端口地址  
022 #define CONT_8155_L8 (INT8U *)0xFD04 //计数器低 8 位地址  
023 #define CONT_8155_H8 (INT8U *)0xFD05 //计数器高 6 位 + 2 位方式地址  
024 #define PMEM_8155 (INT8U *)0xFC00 //8155RAM 地址  
025  
026 //蜂鸣器定义  
027 #define BEEP() PORTB ^= _BV(PB0)  
028 //0~9 的共阳数码管段码表, 最后一位为黑屏幕  
029 const INT8U SEG_CODE[] =  
030 { 0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90, 0xFF } ;  
031 //待显示信息缓冲  
032 INT8U Disp_Buffer[4] = {10, 3, 5, 0};  
033 //8155 定时计数初值
```

```

034 volatile INT16U cnt_8155 = 350;
035 //定时初值递增或递减
036 enum OP_Type {ADD,SUB};
037 //-----
038 // 输出提示音
039 //-----
040 void Sounder()
041 {
042     INT8U i;
043     for (i = 0; i<50; i++)
044     {
045         BEEP(); _delay_us(160);
046     }
047 }
048
049 //-----
050 // 设置 8155 定时初值
051 //-----
052 void Set_8155_TC()
053 {
054     *CONT_8155_L8 = (INT8U)cnt_8155;           //装入定时初值低字节
055     *CONT_8155_H8 = (INT8U)(cnt_8155>>8);    //装入定时初值高字节
056     *COMM_8155 = 0B11000011;                   //设置 PA、PB、PC 端口方式及定时器命令
057 }
058
059 //-----
060 // 8155 定时初值调整
061 //-----
062 void adjust_tCount(enum OP_Type op)
063 {
064     INT8U i;
065     INT16U cnt;
066     cnt_8155 = (op == ADD) ? cnt_8155 + 50; cnt_8155 - 50;
067     if      (cnt_8155 > 500) cnt_8155 = 500;
068     else if (cnt_8155<100) cnt_8155 = 100;
069     cnt = cnt_8155;
070     for (i = 3; i >= 1; i--)
071     {
072         Disp_Buffer[i] = cnt % 10; //从低位开始逐位分解
073         cnt /= 10;
074     }

```



```
075 }
076
077 //-----
078 // 8155PC 端口按键处理
079 //-----
080 void Key_Process()
081 {
082     INT8U i;
083     //上次按键状态
084     static INT8U Pre_Key_State = 0xFF;
085     //读 8255PC 端口按键状态
086     INT8U curr_Key_State = * PC_8155 | 0xF0;
087     //按键状态未改变则返回
088     if (Pre_Key_State == curr_Key_State) return;
089     //保存当前按键状态(用于下一次判断状态是否改变)
090     Pre_Key_State = curr_Key_State;
091     //处理按键操作
092     switch (curr_Key_State)
093     {
094         case (INT8U)~_BV(0): //K1:递增 8155 定时初值,每次递增 50
095             adjust_tCount(ADD);
096             break;
097         case (INT8U)~_BV(1): //K2:递减 8155 定时初值,每次递减 50
098             adjust_tCount(SUB);
099             break;
100         case (INT8U)~_BV(2): //K3:设置并启动 8155 定时器
101             Set_8155_TC();
102             break;
103         case (INT8U)~_BV(3): //K4:写 8155RAM; 0~100
104             for (i = 0 ; i <= 100; i++)
105             {
106                 *(PMEM_8155 + i) = i;
107             }
108             break;
109     }
110 }
111
112 //-----
113 // 主程序
114 //-----
115 int main()
```

```

116 {
117     INT8U i;
118     DDRA = 0xFF;           //配置端口
119     DDRB = 0xFF;
120     DDRC = 0xFF;
121     DDRD = 0x00; PORTD = 0xFF;
122     MCUCR = 0x82;         //允许访问外部存储器/接口等,INT0 中断下降沿触发
123     * COMM_8155 = 0B00000011; //设置 8155 命令字:PA、PB 输出,PC 输入,不影响计数器工作
124     GICR = _BV(INT0);    //INT0 中断使能
125     sei();                //开中断
126     while(1)
127     {
128         for(i = 0; i < 4; i++)          //4 位数码管显示
129         {
130             * PB_8155 = 0x00;          //暂时关闭
131             * PA_8155 = SEG_CODE[Disp_Buffer[i]]; //向 8155 PA 端口发送段码
132             * PB_8155 = _BV(7 - i);    //向 8155 PB 端口发送位码
133             _delay_ms(4);
134             Key_Process();           //8155 PC 端口按键处理
135         }
136     }
137 }
138
139 //-----
140 // INT0 中断子程序
141 //-----
142 ISR (INT0_vect)
143 {
144     Sounder();               //蜂鸣器输出
145     Set_8155_TC();          //重置 8155 TC 初值并启动
146 }

```

4.7 可编程外围定时/计数器 8253 应用

8253 可编程定时/计数器片内含 3 个独立的 16 位计数器,计数器均为递减计数,各计数器的工作方式与初值由软件设置。本例运行时,如果按下 8255 上的“启动 8253TC0”按键,单片机将向 8253 写入随机定时初值,由于定时初值不同,定时溢出中断将以不同时间间隔触发,每次触发时单片机输出报警声音。单片机随机写入 8253 的定时初值由 8255 控制数码管显示。本例电路及部分运行效果如图 4-9 所示。

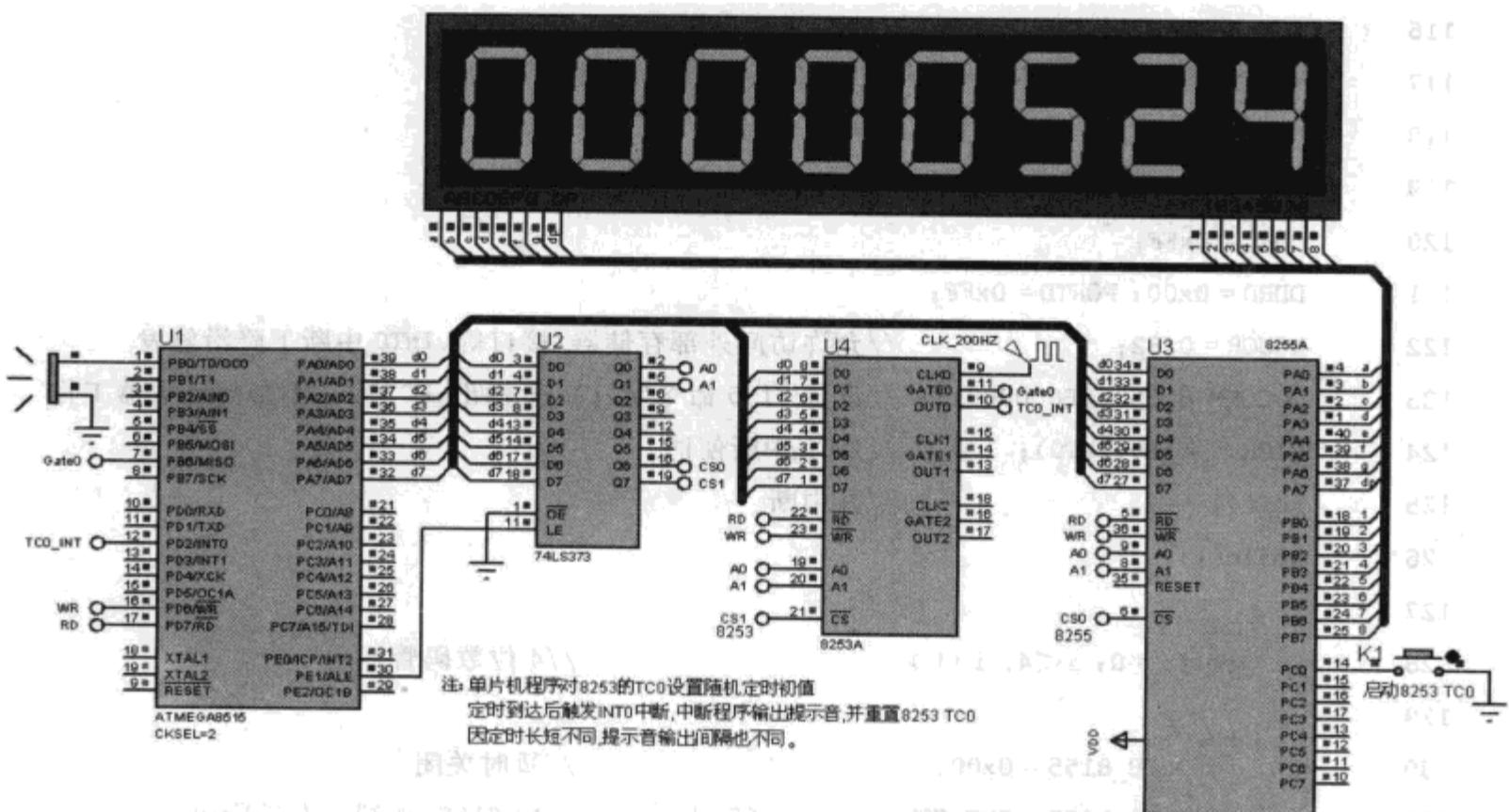


图 4-9 可编程外围定时/计数器 8253 应用

1. 程序设计与调试

本例要点之一在于 8255 与 8253 的接口扩展地址定义：

不同于上一案例中 8255 地址定义的是，本例 8255 地址定义的最前面添加有“*”号，这样定义后，再读/写所定义的地址空间时就不需要在 PA_8255 等符号前面添加“*”。对于 8253 的地址定义则未添加“*”，其用法与上一案例中的 8255 端口操作类似。

由于 8255 与 8253 都没有占用 16 位接口扩展地址的高 8 位（即 PC 端口 A8~A15 不连接 8255 与 8253），因此地址定义中将它们全部设为全 1(FF)。地址低 8 位中的 PA7 与 PA6 分别通过地址锁存器 74LS373 连接 8253 的 CS1 和 8255 的 CS0 引脚，在片选 8253 与 8255 时，它们分别互斥为 0，地址低 8 位中的高 4 位中后 2 位未用，因此 8253 与 8255 低 8 位地址中高 4 位分别为 B(1011) 和 7(0111)，8255 地址的最低 4 位定义可参考前面的 8255 案例。

```
#define PA_8255      * (INT8U *)0xFFB0
#define PB_8255      * (INT8U *)0xFFB1
#define PC_8255      * (INT8U *)0xFFB2
#define COM_8255     * (INT8U *)0xFFB3
```

在 16 位的扩展接口地址中，8253 和 8255 一样都有 A1 与 A0 引脚，其定义也很相似。查阅 8253 的技术手册可知，A1 与 A0 组合为 00、01、10、11 时，分别选择计数器 0、1、2、及控制寄存器。由于本例仅使用了 8253 的 TC0 并需要对其进行命令控制，于是有如下地址定义：

```
#define TC0_8253    (INT8U *)0xFF70
#define COM_8253     (INT8U *)0xFF73
```

下面再来看一下函数 Set_8253_TC0 中的代码：

```

* COM_8253 = 0x3A;           //工作方式为5,先读/写低字节,后读/写高字节
TC0_Count = rand() % 600;    //初值限制于600以内
*(INT16U *)TC0_8253 = (INT8U)TC0_Count; //送低字节
* TC0_8253 = (INT8U)(TC0_Count>>8); //送高字节

```

在阅读这些代码之前要参阅8253的控制字格式,表4-7给出了8253的命令字格式。

表4-7 8253命令字格式

SC1	SC0	RL1	RL0	M2	M1	M0	BCD
-----	-----	-----	-----	----	----	----	-----

其中高2位SC1/0用于选择计数器,取值00、01、10、11分别对应计数器0、1、2、非法。

RL1/0用于设定对计数器的读/写顺序,00、01、10、11分别表示闩锁、只读/写高字节、只读/写低字节、先读/写低字节后读/写高字节。

M2/1/0取值000~101,分别用于选择计数器的工作方式0~5,本例选择的工作方式为5,即硬件触发选通方式。写入方式控制字和计数初值后,输出保持高电平,只有在门控信号GATE的上升沿之后才开始计数,完成最后一个计数后输出一个时钟周期的负脉冲。

最后一位BCD取0~1表示按二进制计数或按BCD码计数。

上述代码中*COM_8253取值0x3A(00111010),设定计数器0工作方式为5,先读/写低字节后读/写高字节。

2. 实训要求

① 修改本例程序,对CLK0与CLK1两路输入脉冲进行计数,计数值分成两组显示在数码管上。

② 修改本例电路,利用8255控制条形LED,利用8253控制扬声器,实现自定义音乐片段输出,根据不同的输出频率,8255控制的条形LED可实现同步闪烁。

3. 源程序代码

```

001 //-----
002 // 名称: 可编程外围定时计数器8253应用
003 //-----
004 // 说明: 本例运行时,按下8255PC端口按键K1可启动8253定时器0,定时器0
005 // 工作于方式5,程序给8253提供随机的定时初值,定时到达后GATE0
006 // 触发INT0中断,中断程序输出提示音,并重置定时器
007 //
008 //-----
009 #define F_CPU 2000000UL
010 #include <avr/io.h>
011 #include <avr/interrupt.h>
012 #include <util/delay.h>
013 #include <stdlib.h>
014 #define INT8U unsigned char
015 #define INT16U unsigned int

```



```
016
017 //8255 PA、PB、PC 端口及命令端口地址定义
018 //（定义前面加 * 可使得后面使用时不用再加 * ）
019 #define PA_8255 * (INT8U *)0xFFB0
020 #define PB_8255 * (INT8U *)0xFFB1
021 #define PC_8255 * (INT8U *)0xFFB2
022 #define COM_8255 * (INT8U *)0xFFB3
023
024 //8253 定时/计数器 0 及命令端口地址定义
025 #define TC0_8253 (INT8U *)0xFF70
026 #define COM_8253 (INT8U *)0xFF73
027
028 //8253 定时/计数器 0/1 的门控制位操作定义
029 #define TC0_G1() PORTB |= _BV(PB6)
030 #define TC0_G0() PORTB &= ~_BV(PB6)
031
032 //蜂鸣器定义
033 #define BEEP() PORTB ^= _BV(PB0)
034 //0~9 的共阳数码管段码表,最后一位为黑屏
035 const INT8U SEG_CODE[] =
036 { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xFF };
037
038 //待显示信息缓冲
039 INT8U Disp_Buffer[8] = {0,0,0,0,0,0,0,0};
040 //上次按键状态
041 INT8U Pre_Key_State = 0x00;
042 //对 TCO 设置的定时/计数初值
043 volatile INT16U TCO_Count = 510;
044 //-----
045 // 输出提示音
046 //-----
047 void Sounder()
048 {
049     INT8U i;
050     for (i = 0; i < 100; i++)
051     {
052         BEEP(); _delay_us(180);
053     }
054 }
055
056 //-----
```

```

057 // 设置 8253 TC0 定时初值
058 //-----
059 void Set_8253_TC0()
060 {
061     TC0_G0();           //先关闭 TC0 门控制位
062     * COM_8253 = 0x3A; //工作方式为 5,先读/写低字节,后读/写高字节
063     TC0_Count = rand() % 600;          //初值限制于 600 以内
064     * (INT16U *)TC0_8253 = (INT8U)TC0_Count; //送低字节
065     * TC0_8253 = (INT8U)(TC0_Count>>8); //送高字节
066     TC0_G1();           //开 TC0 门控制位
067 }
068
069 //-----
070 // 8255 PC 端口按键处理
071 //-----
072 void Key_Process()
073 {
074     INT8U Key_State;
075     Key_State = PC_8255;           //读 8255 PC 端口按键状态
076     if (Key_State == Pre_Key_State) return;
077     Pre_Key_State = Key_State;
078     switch (Key_State)
079     {
080         case (INT8U)~_BV(0): Set_8253_TC0(); //K1:重置 8253 TC0
081                     sei();           //使能总中断
082                     break;
083         //这里可添加 case,用于 8255 PC 端口的其他按键处理
084     default: break;
085     }
086 }
087
088 //-----
089 // 主程序
090 //-----
091 int main()
092 {
093     INT8U i; INT16U cnt;
094     DDRA = 0xFF; DDRB = 0xFF;
095     DDRD = ~(_BV(PD2) | _BV(PD3));
096     srand(87);           //设置随机种子
097

```



```
098     MCUCR |= 0x82;           //允许访问外部存储器/接口等,INT0 中断下降沿触发
099     COM_8255 = 0B10001001;   //8255 工作方式选择:工作于方式 0,PA、PB 输出,PC 输入
100     GICR  = 0x40;          //INT0 中断使能
101     while(1)
102     {
103         cnt = TCO_Count;
104         for(i = 0; i<8; i++)           //数码管显示
105         {
106             Disp_Buffer[i] = cnt % 10;
107             cnt /= 10;
108             PB_8255 = _BV(7 - i);       //向 8255PB 端口发送位码
109             PA_8255 = (INT8U)SEG_CODE[ Disp_Buffer[i] ]; //向 8255PA 端口发送段码
110             _delay_ms(2);
111             Key_Process();           //8255PC 端口按键处理
112         }
113     }
114 }
115
116 //-----
117 // INT0 中断子程序
118 //-----
119 ISR (INT0_vect)
120 {
121     Sounder();                //蜂鸣器输出
122     Set_8253_TCO();           //重置 8253 TCO
123 }
```

4.8 数码管 BCD 解码驱动器 7447 与 4511 应用

此前有关数码管显示的案例中,单片机必须向数码管发送段码。本例使用的七段数码管显示译码器 7447 与 4511 各自仅占用 PB 端口高/低各 4 位引脚,单片机向 7447 与 4511 分别写入 4 位 8421BCD 码,经 2 块芯片译码后再向数码管输出数字段码,实现数码管显示。本例电路及部分运行效果如图 4-10 所示。

1. 程序设计与调试

本例使用了七段数码管显示驱动器 7447 与 4511,它们接收数字 0~9 的 4 位 BCD 编码,译码后输出 0~9 的段码,因此本例代码中没有出现数码管段码表,待显示的数字可以直接输出。

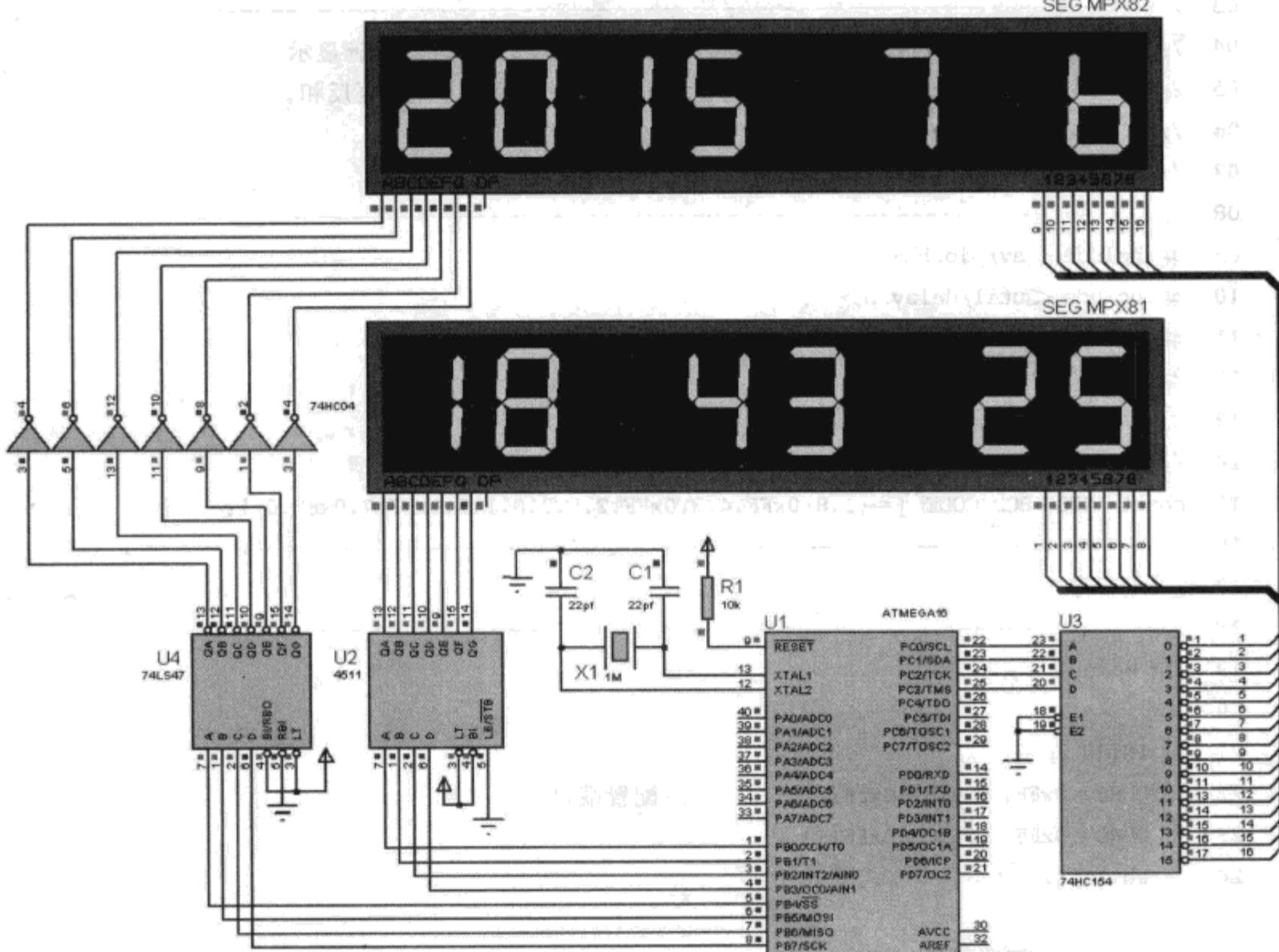


图 4-10 数码管 BCD 解码驱动器 7447 与 4511 应用

传输给 4511 的 4 位 BCD 码只能是 0000~1001，即 0~9 的 BCD 码；超过 1001 的编码会使输出为 00000000，共阴数码管各段均不显示，数码管黑屏。本例中发送给 4511 的 BCD 码为 1、8、0xFF、4、3、0xFF、2、5，数码管显示是 18 43 25。

传输给 7447 的 4 位 BCD 码与 4511 类似,由于 7447 是驱动共阳数码管的,同样的 BCD 码输入后,它的段码输出与 4511 完全相反。

本例两组数码管全部是共阴的，4511 可以直接驱动，但对 7447 则需要在输出端添加非门，如果改用共阳数码管，则输出端不需要添加非门，但本例中由 4-16 译码器控制的扫描码输出端(标号为 9~16)就需要添加非门了。

2. 实训要求

- ① 将两组数码管全部改用 4511 或 7447 驱动显示。
 - ② 在 Proteus 中输入“bcd to 7 – segment”可以找到多种其他七段码数码管译码/驱动器，尝试改用搜索到的其他译码/驱动芯片控制数码管显示。

3. 源程序代码

01 //-----
02 // 名称：数码管 BCD 解码驱动器 7447 与 4511 应用



```
03 //---  
04 // 说明：BCD 码经 7447 或 4511 译码后输出数码管段码，实现数码管显示  
05 //      (7447 驱动共阳数码管，本例用的是共阴数码管，因此需要反相，  
06 //      4511 驱动共阴极数码管)  
07 //---  
08 //---  
09 #include <avr/io.h>  
10 #include <util/delay.h>  
11 #define INT8U unsigned char  
12 #define INT16U unsigned int  
13  
14 //待显示的数字串"18 43 25"和"2015 7 6",其中 00xFF 是不显示的。  
15 const INT8U BCD_CODE[] = {1,8,0xFF,4,3,0xFF,2,5,2,0,1,5,0xFF,7,0xFF,6,};  
16 //---  
17 // 主程序  
18 //---  
19 int main()  
20 {  
21     INT8U i;  
22     DDRB = 0xFF; PORTB = 0xFF;           //配置端口  
23     DDRC = 0xFF; PORTC = 0xFF;  
24     while(1)  
25     {  
26         //4511 解码显示  
27         for(i = 0; i < 8; i++)  
28         {  
29             //译码器输出 0~7 对应的扫描码，控制数码管 7SEG_MPX81  
30             PORTC = i;  
31             //向 4511 输出待显示数字的 BCD 码(非段码)  
32             PORTB = BCD_CODE[i];  
33             _delay_us(500);  
34         }  
35         //7447 解码显示  
36         for(i = 8; i < 16; i++) //或改成 for ( ; i < 16; i++)  
37         {  
38             //译码器输出 8~15 对应的扫描码，控制数码管 7SEG_MPX82  
39             PORTC = i;  
40             //向 7447 输出待显示数字的 BCD 码(非段码)  
41             //因为 7447 的输入端 DCBA 连接 PC 端口高 4 位，故这里需要左移  
42             PORTB = BCD_CODE[i] << 4;  
43             _delay_us(500);  
44         }  
45     }  
46 }
```

4.9 8×8 LED 点阵屏显示数字

本例 8×8 LED 点阵屏的行驱动由 PC 端口控制,列选通由 PD 端口控制,程序运行时,8×8 LED 点阵屏依次循环显示数字 0~9,刷新过程由 T/C0 定时器溢出中断程序控制完成。本例电路及部分运行效果如图 4-11 所示。

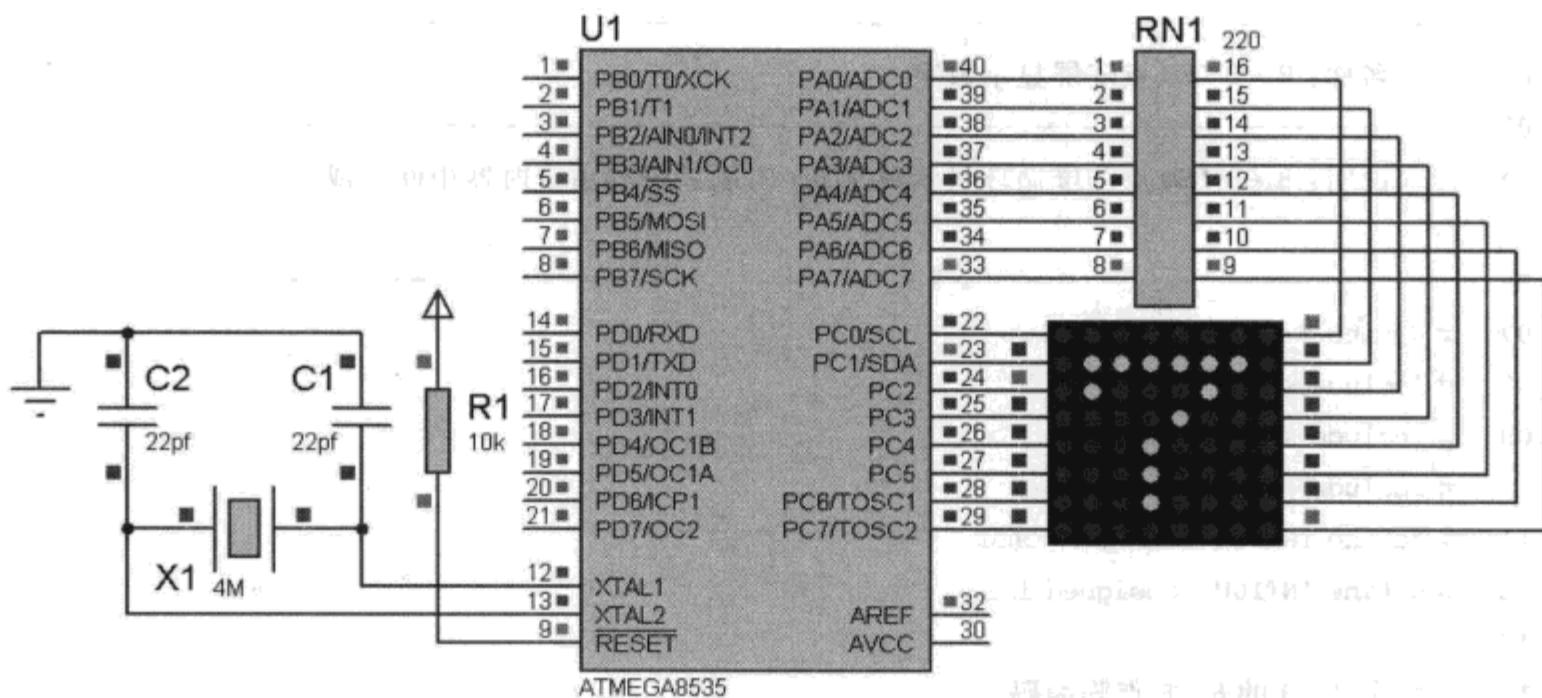


图 4-11 8×8 LED 点阵屏显示数字

1. 程序设计调试

点阵显示屏的动态刷新显示与集成式 8 位数码管的动态刷新显示非常相似,8 位数码管中的一只相当于点阵屏中的一列,数码管的段码相当于点阵屏的行码,位码则相当于点阵屏的列码,两者逻辑结构完全相同,只是外观不一样。由于逻辑结构相似,因此本例点阵屏的中断刷新显示代码与上一案例中的代码也非常相似。

如果点阵屏中每一行 LED 是共阳连接,那么每一列必定是共阴连接;如果将其旋转 90°,则行是共阴连接,列是共阳连接;如果将其旋转 180° 旋转,则逻辑结构没有改变,但在编程控制显示时,点阵的取法或行码与列码字节的发送顺序需要作相应调整。

程序中数组 Table_OF_Digits 共有 64 个字节,每 8 个字节为一个数字的点阵代码,其中每个字节的 8 位对应于一列中的 8 个点,例如数组中第 0 行的 8 个字节 0x00、0x3C、0x66、0x42、0x42、0x66、0x3C、0x00 就是数字 0 第 0~7 列的点阵编码,这类似于数码管一个数字的段码,在点阵屏中就是行码,它们将被分别发送到显示屏的第 0~7 列。各字节的高位对应于列中上面的点还是下面的点,这由 PA 端口与显示屏 8 只行引脚的连接顺序决定。

变量 Num_Index 标明了将要显示的数字,取值范围为 0~9,变量 i 的取值范围为 0~7,表达式 Table_OF_Digits[Num_Index * 8 + i]使程序取得第 Num_Index 个数字的第 i 个字节,因为每个数字的点阵编码由 8 个字节构成,每次取得 0~7 个字节中的一个,通过 PC 端口发送到点阵屏的行引脚上。在发送行码之前,PC 端口先发送相应的列码选通对应列。由语句 PORTC=_BV(i)可以看出,在本例点阵屏连接方式下,各列中的 LED 是共阳的,_BV(i)总是使第 i 列变为高电平,其他列为低电平,这时发送的行码将仅仅显示在第 i 列,这类似于当前发送给 8 位集成式数码管中的段码将仅仅显示在第 i 位上。

2. 实训要求

- ① 应用单片机的4个端口，控制16×16点阵LED显示屏显示。
- ② 利用2片595串入并出芯片分别控制8×8点阵屏的行码与列码，实现数字或字符显示。

3. 源程序代码

```

01 //-----
02 // 名称：8 * 8 LED 点阵屏显示数字
03 //-----
04 // 说明：8 * 8 LED 点阵屏循环显示数字 0~9，刷新过程由定时器中断完成
05 //
06 //-----
07 #define F_CPU 4000000UL
08 #include <avr/io.h>
09 #include <avr/interrupt.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //数字 0~9 的 8 * 8 点阵编码
15 const INT8U Table_OF_Digits[] =
16 {
17     0x00,0x3C,0x66,0x42,0x42,0x66,0x3C,0x00,//0
18     0x00,0x08,0x38,0x08,0x08,0x08,0x3E,0x00,//1
19     0x00,0x3C,0x42,0x04,0x08,0x32,0x7E,0x00,//2
20     0x00,0x3C,0x42,0x1C,0x02,0x42,0x3C,0x00,//3
21     0x00,0x0C,0x14,0x24,0x44,0x3C,0x0C,0x00,//4
22     0x00,0x7E,0x40,0x7C,0x02,0x42,0x3C,0x00,//5
23     0x00,0x3C,0x40,0x7C,0x42,0x42,0x3C,0x00,//6
24     0x00,0x7E,0x44,0x08,0x10,0x10,0x10,0x00,//7
25     0x00,0x3C,0x42,0x24,0x5C,0x42,0x3C,0x00,//8
26     0x00,0x38,0x46,0x42,0x3E,0x06,0x3C,0x00 //9
27 };
28
29 //-----
30 // 主程序
31 //-----
32 int main()
33 {
34     DDRA = 0xFF; PORTA = 0xFF; //配置端口
35     DDRC = 0xFF; PORTC = 0xFF;
36     TCCR0 = 0x03; //预设分频:64
37     TCNT0 = 256 - F_CPU / 64.0 * 0.004; //晶振 4 MHz, 4 ms 定时初值
38     TIMSK = 0x01; //允许 T0 定时器溢出中断

```

```

39     sei();                                //开总中断
40     while (1);
41 }
42
43 //-----
44 // T0 定时器中断控制 LED 点阵屏刷新显示
45 //-----
46 ISR (TIMER0_OVF_vect)
47 {
48     static INT8U i = 0, t = 0, Num_Index = 0;
49     TCNT0 = 256 - F_CPU / 64.0 * 0.004;      //列间延时 4 ms
50     PORTC = _BV(i);                          //列码
51     PORTA = ~Table_OF_Digits[Num_Index * 8 + i]; //行码(用~反相显示)
52     if( ++i == 8) i = 0;                      //每屏一个数字由 8 个字节构成
53     if( ++t == 250)                           //每个数字刷新显示一段时间
54     {
55         t = 0;
56         if( ++Num_Index == 10) Num_Index = 0;    //显示下一个数字
57     }
58 }

```

4.10 8位数码管段位复用串行驱动芯片 MAX6951 应用

Maxim 公司推出的 MAX6950/51 都是串行接口的共阴数码管显示驱动器, 工作电压可低至 2.7 V, 它们可分别驱动 5 位或 8 位的七段数码管。驱动芯片内置十六进制字符译码器 (0~9, A~F)、复用扫描电路、段码和位码驱动器以及用于存储每一位数字的静态 RAM。

使用 MAX6950/51 时可以为每一位数字选择十六进制译码或非译码模式驱动任何七段数码管, 每位数字不需要重写整个显示器即可单独寻址和刷新。该器件具有低功耗关断模式、数字亮度控制电路、扫描范围寄存器(允许用户选择 1~8 位显示数字), 各驱动器可相互保持同步的段闪烁控制以及强制所有 LED 打开的测试模式。本例电路及部分运行效果如图 4-12 所示。

1. 程序设计与调试

在设计 MAX6951 应用电路与应用程序之前, 先简要介绍一下 6951 的引脚:

DIN: 串行数据输入, 在 CLK 的上升沿将数据移入内部 16 位移位寄存器。

CLK: 串行时钟输入, 片选 CS 有效时, 在 CLK 的上升沿, 数据移入内部移位寄存器。

DIGX/SEGX: 位码与段码复用驱动位, DIGX 吸入来自数码管共阴极的电流, SEGX 输出电流, 位码与段码在关断时处于高阻状态。

ISET: 电流设定, 此引脚与 GND 之间串接电阻 R_{SET} , 设置峰值电流, 该电阻器还与电容器 C_{SET} 一起设置多路复用的显示时钟频率。

OSC: 多路复用显示时钟输出。

CS: 片选引脚, 低电平时串行数据移入移位寄存器, 在 CS 的上升沿锁存最后的 16 位数据。

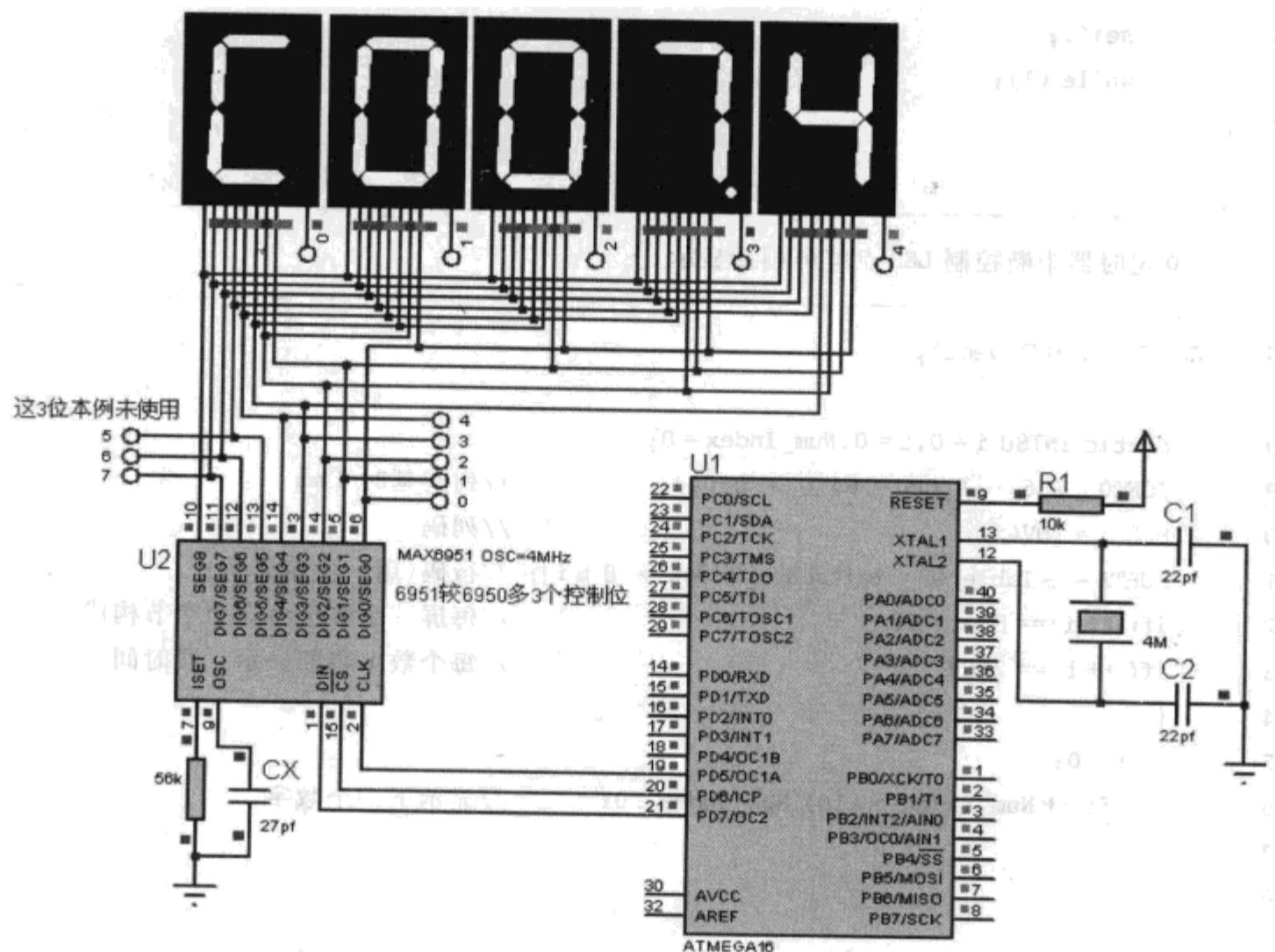


图 4-12 8 位数码管段位复用串行驱动芯片 MAX6951 应用

表 4-8 给出了 6951 与分立式数码管的连接方法,由于使用了段/位复用设计,本例中不能选用集成式数码管。图 4-12 中的 5 只独立数码管正是参照表 4-8 连接的,本例电路中还可以再添加 3 只数码管。

表 4-8 MAX6951 与 8 位分立式数码管的连接方法

	DIG/SEG0	DIG/SEG 1	DIG/SEG 2	DIG/SEG 3	DIG/SEG 4	DIG/SEG 5	DIG/SEG 6	DIG/SEG 7	SEG8
位 0	CC0	DP	G	F	E	D	C	B	A
位 1	DP	CC1	G	F	E	D	C	B	A
位 2	DP	G	CC2	F	E	D	C	B	A
位 3	DP	G	F	CC3	E	D	C	B	A
位 4	DP	G	F	E	CC4	D	C	B	A
位 5	DP	G	F	E	D	CC5	C	B	A
位 6	DP	G	F	E	D	C	CC6	B	A
位 7	DP	G	F	E	D	C	B	CC7	A

MAX6950/51 的 16 位串行数据格式为: D15~D0, 其中高 8 位为地址, 低 8 位为数据。本例函数 Write 完成对 16 位地址与数据字节的串行写入操作。本例中数码管 0~7 的地址分别为 0x60~0x67(本例实际使用了 5 位, 即 0x60~0x64), 主程序中的 3 种演示将分别对这些地址进行写入。

为尽可能充分演示 MAX6950/51 的功能, 主程序完成了全解码演示、部分解码演示、全部

不解码演示。主程序中的 86、90、94 行代码如下：

```
Write(0x01,0B00011111); //解码模式：对 0~4 位全部解码
Write(0x01,0B00010110); //解码模式：对 1,2,4 解码，第 0,3 位不解码
Write(0x01,0B00000000); //解码模式：全部不解码
```

它们分别向 MAX6950/51 的 0x01 地址（即解码模式地址）写入了不同字节，其中置为 1 的对应位为解码，置为 0 的则不解码。以下分别进行说明：

① 在第 1 组演示所使用的解码模式下，只需要向 MAX6950/51 写入字符的 ASCII 码即可，其优点是不需要提供任何字符段码，不足之处是有些特殊字符在这种模式下无法显示，例如温度符号就是其内置编码表中没有的。

② 在第 2 组混合解码模式演示中，发送给 MAX6951 的一部分是字符 ASCII 码，一部分则是字符段码。

③ 在第 3 种模式下，由于全部不解码，因此需要提供所有待发送字符的段码。需要注意的是，这里的段码顺序与此前数码管由 A~DP 进行逆向编码的顺序是不同的，这是因为驱动线的连接不同，具体编码顺序可参考源程序中的相关说明。

阅读 MAX6950/51 初始化函数 Init_MAX695X 时，可进一步参阅 6951 的技术手册文件，该初始化函数分别设置了亮度、扫描范围及非关断模式。

2. 实训要求

① 6951 最多可驱动 8 位独立数码管，完成本例调试后，在电路中再添加 3 只独立数码管，并进一步改写程序，选用不同解码模式显示自定义数据内容。

② 同时使用 6950/51 两块芯片，驱动更多数据信息显示。

3. 源程序代码

```
001 //-----
002 // 名称：8 位数码管段位复用串行驱动芯片 MAX6951 应用
003 //-----
004 // 说明：本例程序仅占用 PD 端口 3 只引脚即实现了多位数码管的显示控制
005 //
006 //-----
007 #include <avr/io.h>
008 #include <util/delay.h>
009 #define INT8U unsigned char
010 #define INT16U unsigned int
011
012 //MAX695X 引脚操作定义
013 #define CLK_1() PORTD |= _BV(PD5)
014 #define CLK_0() PORTD &= ~_BV(PD5)
015 #define DIN_1() PORTD |= _BV(PD7)
016 #define DIN_0() PORTD &= ~_BV(PD7)
017 #define CS_1() PORTD |= _BV(PD6)
018 #define CS_0() PORTD &= ~_BV(PD6)
019
020 //695X 待显示的几组数据-----
```

```

021 //1. 显示 A,C,2,2,0,全解码(直接发送)
022 const INT8U Test1[] = {0x0A,0x0C,0x02,0x02,0x00};
023
024 //2. 显示温度: -32℃, 其中第 0 位 0x01, 第 3 位 0x63 不解码,
025 //它们分别是"-"的段码及"℃"中小圆圈的段码
026 const INT8U Test2[] = {0x01,0x03,0x02,0x63,0x0C};
027
028 //3. 显示 C000.0 递增,全部不解码
029 //显示此数组时要使用 MAX695X 的段码表
030 INT8U Test3[] = {0x0C,0,0,0,0};
031
032 //在非解码模式下 MAX6950/1 对应的段码表,此表不同于直接驱动时所使用的段码表
033 //原来的各段顺序是: DP,G,F,E,D,C,B,A
034 //MAX6950/1 的驱动顺序是:DP,A,B,C,D,E,F,G
035 //除小数点位未改变外,其他位是逆向排列的
036 const INT8U SEG_CODE_695X[] =
037 { 0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,
038   0x7F,0x7B,0x77,0x1F,0x4E,0x3D,0x4F,0x47
039 };
040 void Count_Demo();
041 //-----
042 // 向 MAX695X 写数据
043 //-----
044 void Write(INT8U Addr, INT8U Dat)
045 {
046     INT8U i;
047     CS_0();
048     for(i = 0; i < 8; i++)          //串行写入 8 位地址 Addr
049     {
050         CLK_0();
051         if (Addr & 0x80) DIN_1(); else DIN_0();
052         CLK_1(); _delay_us(20);      //时钟上升沿移入数据
053         Addr <<= 1;
054     }
055     for(i = 0; i < 8; i++)          //串行写入 8 位数据 Dat
056     {
057         CLK_0();
058         if (Dat & 0x80) DIN_1(); else DIN_0();
059         CLK_1(); _delay_us(20);      //时钟上升沿移入数据
060         Dat <<= 1;
061     }
062     CS_1();
063 }
064

```

```

065 //-----
066 // MAX695X 初始化
067 //-----
068 void Init_MAX695X()
069 {
070     Write(0x02,0x07);           //设置亮度:中等亮度
071     Write(0x03,0x05);           //扫描所有数码管
072     Write(0x04,0x01);           //非关断 0x01;关断:0x00
073 }
074
075 //-----
076 // 主程序
077 //-----
078 int main()
079 {
080     INT8U i;
081     DDRD = 0xFF; PORTD = 0xFF;
082     Init_MAX695X();           //695X 初始化
083     while (1)
084     {
085         //1 - 显示 A,C,2,2,0(全解码)-----
086         Write(0x01,0B00011111); //解码模式:对 0~4 位全部解码
087         for(i = 0; i<5; i++) Write( 0x60 | i, Test1[i]);
088         _delay_ms(1000);
089         //2 - 显示温度: -32 ℃ (部分解码)-----
090         Write(0x01,0B00010110); //解码模式:对 1、2、4 解码,第 0、3 位不解码
091         for(i = 0; i<5; i++) Write( 0x60 | i, Test2[i]);
092         _delay_ms(1000);
093         //3 - C000.0 递增演示(全部不解码,发送段码)-----
094         Write(0x01,0B00000000); //解码模式:全部不解码
095         Count_Demo();
096         _delay_ms(2000);
097     }
098 }
099
100 //-----
101 // 数码管数码递增演示 C000.0~C999.9(本例实际演示到 C015.0 时停止)
102 //-----
103 void Count_Demo()
104 {
105     INT8U i,j,k,l;
106     Write( 0x60, SEG_CODE_695X[Test3[0]]); //显示第一个字符 C
107     //以下分别显示 3 个整数位和 1 个小数位
108     for (i = 0; i<10; i++)

```

```

109      {
110          //显示百位数
111          Test3[1] = i; Write( 0x61, SEG_CODE_695X[Test3[1]] );
112          for (j = 0; j<10; j++)
113          {
114              //显示十位数
115              Test3[2] = j;     Write( 0x62, SEG_CODE_695X[Test3[2]] );
116              for (k = 0; k<10; k++)
117              {
118                  //显示个位数,小数点显示个位数旁边
119                  Test3[3] = k; Write( 0x63, SEG_CODE_695X[Test3[3]] | 0x80 );
120                  for (l = 0; l<10; l++)
121                  {
122                      //显示小数位
123                      Test3[4] = l; Write( 0x64, SEG_CODE_695X[Test3[4]] );
124                      _delay_ms(80);
125                      //为提前结束演示,这里添加演示到 15.0 时退出的语句
126                      if (i == 0 && j == 1 && k == 5) return;
127                  }
128              }
129          }
130      }
131  }

```

4.11 串行共阴显示驱动器 MAX7219 与 7221 应用

本例所使用的 MAX7219/7221 是串行集成式共阴显示驱动器,它用来连接单片机与 8 位七段数码管,也可以连接条形 LED 或者 8×8 LED 点阵屏。本例中的两片 MAX7219/7221 驱动两组 8 位七段共阴数码管,两块芯片的串行数据输入线(DIN)与时钟线(CLK)分别共用 PC0 与 PC2 引脚,片选线(\overline{CS})与数据加载线(LOAD)独立。案例电路及部分运行效果如图 4-13 所示。

1. 程序设计与调试

本例用共阴显示驱动芯片 MAX7219/7221 控制数码管显示,每只仅占用单片机 3 只引脚。本例中通过复用串行数据线(DIN)与时钟线(CLK),两者共占用了 PC 端口的 4 只引脚。

除了对单片机端口的占用很少外,使用 MAX7219/7221 最大的优点还在于单片机向它输出所要显示的内容以后,不再需要用动态扫描法高速刷新数码管显示,这显然大大节省了对单片机时间的占用。

表 4-9 给出了 MAX7219/7221 的引脚功能说明,表 4-10 给出了 MAX7219/7221 的串行数据格式(16 位)及寄存器地址表,阅读源程序中的初始化函数 Init_MAX72XX 和 Write 函数时可参考这些表格。

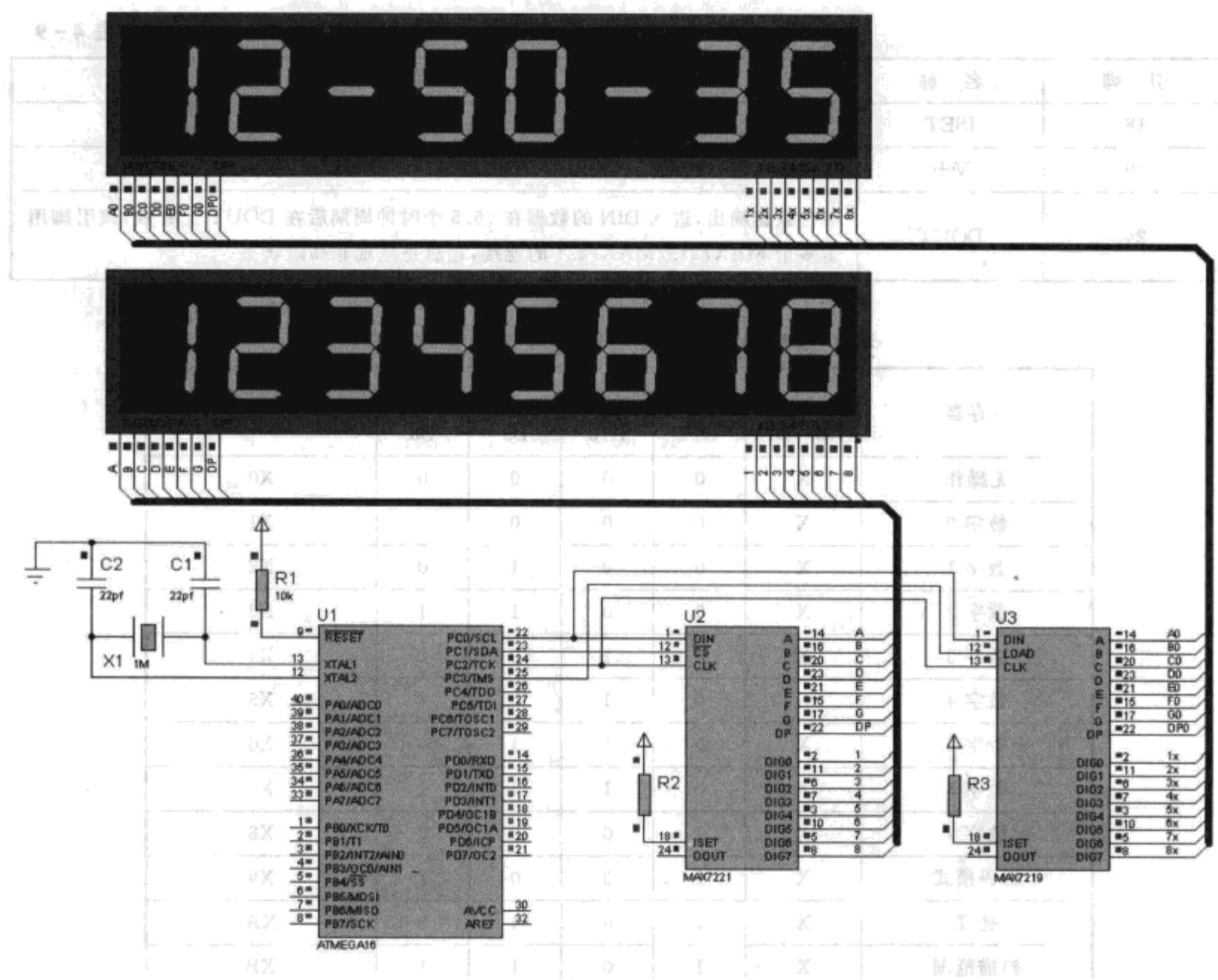


图 4-13 串行共阴显示驱动器 MAX7219 与 7221 应用

表 4-9 MAX7219/MAX7221 引脚功能表

引脚	名称	功 能
1	DIN	串行数据输入, 数据在时钟上升沿进入内部 16 位的移位寄存器
2,3,5~8,10,11	DIG0~DIG7	接入 8 位共阴数码管反向电流的驱动线。在关闭时, MAX7219 将数字输出拉到 V+, 而 MAX7221 的数字驱动线呈现高阻状态
4,9	GND	地端(两个 GND 都必须接地)
12	LOAD (MAX7219)	加载数据输入, 最近的 16 位串行数据在 LOAD 的上升沿锁存
	CS/ (MAX7221)	片选输入, 当 CS 为低电平时, 串行数据加载到移位寄存器, 最近的 16 位串行数据在 CS 的上升沿锁存
13	CLK	串行时钟输入, 最大速率为 10 MHz, 在时钟上升沿时数据移入内部移位寄存器, 下降沿时数据由 DOUT 移出, 对于 MAX7221, 时钟仅在 CS 为低电平时有效
14~17,20~23	SEG A~SEG G, DP	数码管段驱动线(含小数位), 它们为数码管显示提供驱动电流, 在关闭时, MAX7219 段驱动被拉到地, 而 MAX7221 为高阻状态

续表 4-9

引脚	名称	功能
18	ISET	通过电阻连接 VDD(R_{set})以控制最高段电流
19	V+	正电源电压,连接 +5 V
24	DOUT	串行数据输出,进入 DIN 的数据在 16.5 个时钟周期后在 DOUT 上有效,该引脚用于多个 MAX7219/MAX7221 的连接,它总是呈现非高阻状态

表 4-10 MAX7219/MAX7221 寄存器地址表

寄存器	地址					十六进制编码
	D15~D12	D11	D10	D9	D8	
无操作	X	0	0	0	0	X0
数字 0	X	0	0	0	1	X1
数字 1	X	0	0	1	0	X2
数字 2	X	0	0	1	1	X3
数字 3	X	0	1	0	0	X4
数字 4	X	0	1	0	1	X5
数字 5	X	0	1	1	0	X6
数字 6	X	0	1	1	1	X7
数字 7	X	1	0	0	0	X8
解码模式	X	1	0	0	1	X9
亮度	X	1	0	1	0	XA
扫描范围	X	1	0	1	1	XB
关闭	X	1	1	0	0	XC
显示测试	X	1	1	1	1	XF

注:16 个数据位中,D11~D8 位为寄存器地址,D7~D0 为发送的数据。

2. 实训要求

- ① 将 MAX7219 或 7221 设为全部不解码,使数码管可同时显示普通字符与特殊字符。
- ② 重新设计电路,用两片 MAX7219 或 7221 控制两片 8×8 LED 点阵显示屏显示图文信息。
- ③ 重新设计电路,使两片 MAX7219 或 7221 的 CS(或 LOAD)、CLK 分别并联到 PC1 与 PC2 引脚,串行数据通过单片机 PC0 引脚传入第一片的 DIN 引脚,第一片的 DOUT 引脚则连接第二片的 DIN 引脚,重新编程在两片 MAX7219/7221 控制的数码管上显示两组数据信息。
- ④ 在完成上一设计后可再添加多片 MAX7219/7221,控制更多组数码管的显示。显然,采用 DOUT 与 DIN 级联的方法可使多片芯片只占用单片机的 3 只引脚。

3. 源程序代码

```

01 //-----
02 // 名称: 串行共阴显示驱动器 MAX7219/7221 控制数码管显示
03 //-----
04 // 说明: 本例用 MAX7219/7221 控制 8 只数码管动态显示,每组数字输出后

```

```

05 // 不必再高速刷新,该芯片的使用大大减少了对单片机引脚和单片
06 // 机时间的占用
07 //
08 //-----
09 #include <avr/io.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //引脚操作定义
15 #define DIN_1() PORTC |= (INT8U)_BV(PC0)
16 #define DIN_0() PORTC &= ~(INT8U)_BV(PC0)
17 #define CLK_1() PORTC |= (INT8U)_BV(PC2)
18 #define CLK_0() PORTC &= ~(INT8U)_BV(PC2)
19
20 #define CS7221_1() PORTC |= (INT8U)_BV(PC1)
21 #define CS7221_0() PORTC &= ~(INT8U)_BV(PC1)
22 #define CS7219_1() PORTC |= (INT8U)_BV(PC3)
23 #define CS7219_0() PORTC &= ~(INT8U)_BV(PC3)
24
25 //7219 待显示的数字串"2015 9 5"、"12 - 50 - 35"(其中 10 是不显示的)
26 const INT8U Disp_Buffer0[] = {2,0,1,5,10,9,10,5,1,2,10,5,0,10,3,5};
27 //7221 待显示的 1~8,8~1;
28 const INT8U Disp_Buffer1[] = {1,2,3,4,5,6,7,8,8,7,6,5,4,3,2,1};
29 //-----
30 // 向 MAX7221/7219 写数据
31 //-----
32 void Write(INT8U Addr, INT8U Dat, INT8U Clip_N0)
33 {
34     INT8U i;
35     if (Clip_N0 == 1) CS7221_0(); else CS7219_0(); //片选 MAX7219 或 7221
36     for(i = 0; i<8; i++) //串行写入 8 位地址 addr
37     {
38         CLK_0(); if (Addr & 0x80) DIN_1(); else DIN_0();
39         CLK_1(); _delay_us(2);
40         CLK_0(); Addr <<= 1;
41     }
42     for(i = 0; i<8; i++) //串行写入 8 位数据 dat
43     {
44         CLK_0(); if (Dat & 0x80) DIN_1(); else DIN_0();
45         CLK_1(); _delay_us(2);
46         CLK_0(); Dat <<= 1;
47     }
48     if (Clip_N0 == 1) CS7221_1(); else CS7219_1(); //片选禁止
49 }

```



```
50
51 //-----
52 // MAX7221 初始化
53 //-----
54 void Init_MAX72XX(INT8U i)
55 {
56     Write(0x09,0xFF, i);          //解码模式地址 0x09(0x00 为不解码,0xFF 为全解码)
57     Write(0x0A,0x07, i);          //亮度地址 0x0A(0x00~0x0F,0x0F 最亮)
58     Write(0x0B,0x07, i);          //扫描数码管个数地址 0x0B(0x07 为扫描数码管 0~7)
59     Write(0x0C,0x01, i);          //工作模式地址 0x0C(0x00:关闭,0x01:正常)
60 }
61
62 //-----
63 // 主程序
64 //-----
65 int main()
66 {
67     INT8U i;
68     DDRC = 0xFF; PORTC = 0xFF;
69     Init_MAX72XX(0);           //MAX7219 初始化
70     Init_MAX72XX(1);           //MAX7221 初始化
71     while (1)
72     {
73         for(i = 0; i<8; i++)    //显示 Disp_Buffer0 数组的前 8 个数位
74         {
75             Write( i + 1, Disp_Buffer0[i],0);
76         }
77         _delay_ms(200);
78         for(i = 8; i<16; i++)   //显示 Disp_Buffer0 数组的后 8 个数位
79         {
80             Write( i - 7, Disp_Buffer0[i],0);
81         }
82         _delay_ms(200);
83         for(i = 0; i<8; i++)    //显示 Disp_Buffer1 数组的前 8 个数位
84         {
85             Write( i + 1, Disp_Buffer1[i],1);
86         }
87         _delay_ms(200);
88         for(i = 8; i<16; i++)   //显示 Disp_Buffer1 数组的后 8 个数位
89         {
90             Write( i - 7, Disp_Buffer1[i],1);
91         }
92         _delay_ms(200);
93     }
94 }
```

4.12 16段数码管演示

本例 16 段集成式共阴数码管直接由单片机控制显示, 程序设计中最主要的工作在于 16 段数码管的段码编写。案例电路及部分运行效果如图 4-14 所示。

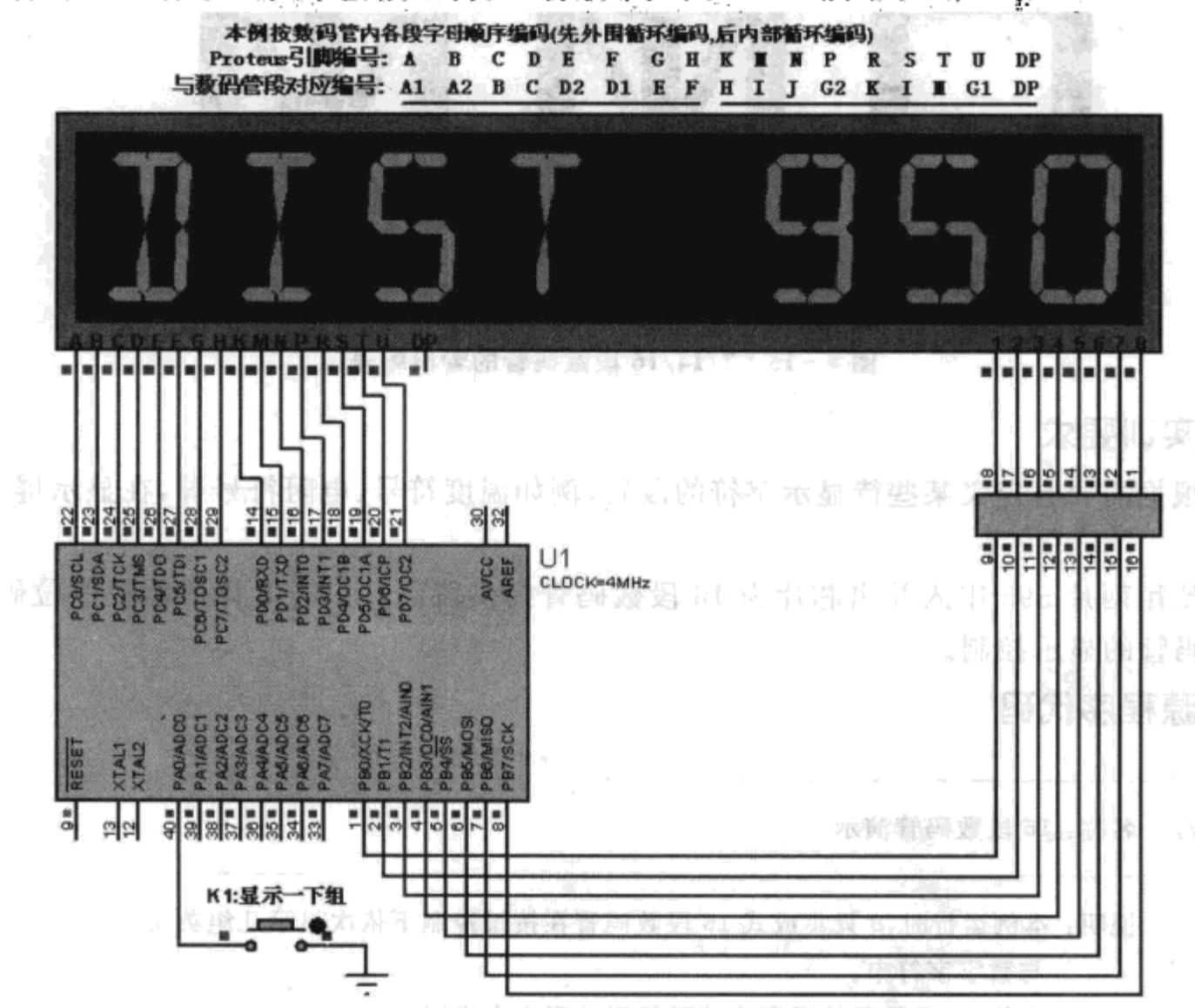


图 4-14 16 段数码管演示

1. 程序设计与调试

驱动 16 段数码管显示时, 需要首先编写 16 段数码管的段码表。图 4-15 给出了 7 段、14 段、16 段数码管各段位的一般排序顺序。图 4-14 中的 16 段集成式数码管 A~DP 引脚与图 4-15 中 16 段数码管的各段对应关系如表 4-11 所列。

表 4-11 PROTEUS 中 16 段数码管引脚 A~DP 与数码管各段的对应关系表

A	B	C	D	E	F	G	H	K	M	N	P	R	S	T	U	DP
a1	a2	b	c	d2	d1	e	f	h	i	j	g2	k	i	m	g1	dp

其中 A~H 引脚与数码管外围“□”由“a1~f”沿顺时钟方向循环一一对应, K~U 与内部“米”字由“h~g1”笔划仍然是沿顺时钟方向循环一一对应。根据以上对应关系可得出所有字符的段码, 16 段数码管的段码设计与 8 段数码管类似。注意: dp 为最高位, a1 为最低位。例如 16 段共阳数码管数字 0~9 的编码为: 0xff00、0xffff3、0x7788、0x77c0、0x7773、0x7744、



0x7704、0xffff0、0x7700、0x7740。

本例程序中提供的是 16 段共阳数码管的段码表，其中“S”与“5”，“O”与“0”在数码管上的显示是相同的，有必要的话可加以修改，以示区别。在电路中共阴数码管上输出时还要注意将各字符的 16 位段码取反(~)。

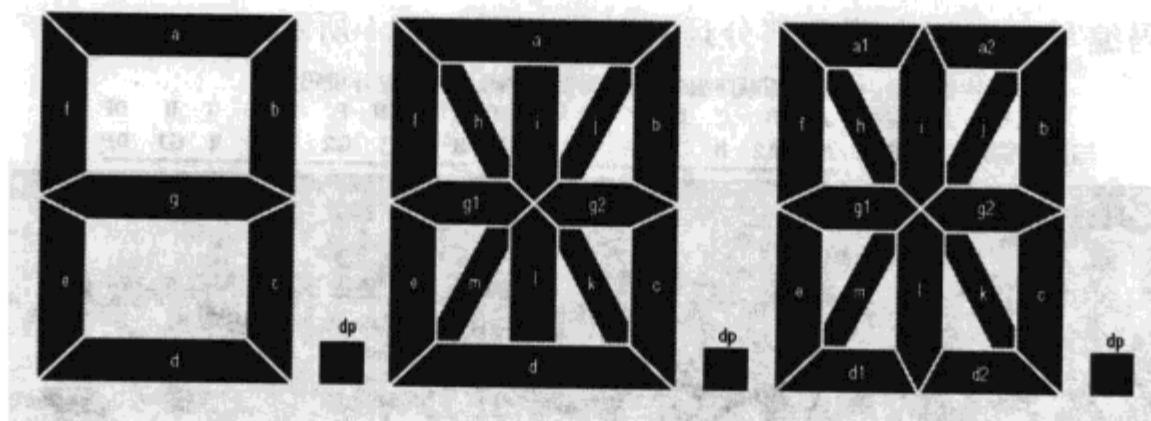


图 4-15 7/14/16 段数码管的管段编号

2. 实训要求

① 根据需要自定义某些待显示字符的段码，例如温度符号、电阻符号等，在显示屏完成显示测试。

② 使用两片 595 串入并出芯片为 16 段数码管提供段码，用 3—8 译码器提供位码，实现 16 段数码管的显示控制。

3. 源程序代码

```
01 //-----  
02 // 名称：16 段数码管演示  
03 //-----  
04 // 说明：本例运行时，8 只集成式 16 段数码管在按键控制下依次显示几组英文  
05 // 与数字字符串。  
06 // 本例 16 段数码管段码表编码规则见程序内说明  
07 //-----  
08 //-----  
09 #define F_CPU 4000000UL  
10 #include <avr/io.h>  
11 #include <util/delay.h>  
12 #include <ctype.h>  
13 #include <string.h>  
14 #include <math.h>  
15 #define INT8U unsigned char  
16 #define INT16U unsigned int  
17  
18 //本例编码按数码管各段字母顺序设计编码(先外框循环,后内部米字循环):  
19 //A1 A2 B C D2 D1 E F H I J G2 K I M G1 DP(编码时注意逆向)  
20 const INT16U SEG_CODE16[] = //16 段共阳数码管段码表(本例用的是共阴数码管,输出时要取反)  
21 { //以下编码中“S”与“5”，“0”与“0”的显示是相同的，必要时可加以修改
```

```

22     0xff00,0xffff3,0x7788,0x77c0,0x7773,0x7744,0x7704,0xffff0,0x7700,0x7740,//0~9
23     0x7730,0x7304,0xff0c,0xddc0,0x770c,0x773c,0xf704,0x7733,0xddcc,0xdd9c,//A~J
24     0x6b3f,0xff0f,0xfa33,0xee33,0xff00,0x7738,0xef00,0x6738,0x7744,0xddfc,//K~T
25     0xff03,0xbb3f,0xaf33,0xaaff,0xdaff,0xbbcc                                //U~Z
26 }
27 //待显示字符串
28 char str_buffer[] = "DIST 950abcdefghijklMNOPlQRSTUVWXYZ 0123456789";
29 //-----
30 // 获取 ASCII 字符的 16 位段码
31 //-----
32 INT16U get_16_segcode(char c)
33 {
34     if (isdigit(c))                                //取得数字段码
35     {
36         c = c - '0';
37         return SEG_CODE16[(INT8U)c];
38     }
39     else if (isalpha(c))                          //取得字母段码
40     {
41         c = toupper(c) - 'A' + 10;
42         return SEG_CODE16[(INT8U)c];
43     }
44     else return 0xFFFF;                           //其他字符返回黑屏段码
45 }
46
47 //-----
48 // 主程序
49 //-----
50 int main()
51 {
52     int i,j,len = strlen(str_buffer);
53     INT8U pre_key = 0xFF;
54     INT16U sCode = 0x0000;
55     DDRA = 0x00; PORTA = 0xFF;                  //配置端口
56     DDRB = 0xFF;
57     DDRC = 0xFF;
58     DDRD = 0xFF;
59     while(1)
60     {
61         i = 0;                                  //组索引,每组显示 8 个字符
62         while ( i<ceil(len / 8.0) )           //用取天花板函数获取组数
63         {
64             for (j = 0; j<8 && 8 * i + j<len; j++)

```



```
65      {
66          PORTB = 0xFF;           //暂时关闭所有数码管
67          sCode = ~get_16_segcode(str_buffer[8 * i + j]); //获取当前字符段码
68          PORTD = sCode >> 8;    //发送段码高 8 位
69          PORTC = sCode & 0x00FF; //发送段码低 8 位
70          PORTB = ~_BV(j);     //发送位扫描码
71          _delay_ms(4);        //延时
72      }
73      if (PIN_A != pre_key) //如果有键按下则显示下一组
74      {
75          if (PIN_A != 0xFF) i++;
76          pre_key = PIN_A;
77      }
78  }
79 }
80 }
81 }
```

4.13 16 键解码芯片 74C922 应用

第 3 章中已经设计调试了 4×4 键盘矩阵扫描程序, 键盘矩阵占用了单片机的一整个 8 位端口。本例电路使用了 16 键专用解码芯片, 该芯片的使用将使程序设计变得非常简单。本例电路及部分运行效果如图 4-16 所示。

1. 程序设计与调试

16 键解码芯片 74C922 采用 CMOS 工艺技术制造, 工作电压为 $3 \sim 15$ V, 具有二键锁定功能, 编码为三态输出, 可与单片机直接相连, 内部振荡器完成 4×4 键盘矩阵扫描, 外接电容用于消抖, 键盘矩阵的 4 行分别连接解码芯片 X1~X4 引脚, 4 列分别连接 Y1~Y4 引脚。当有按键按下时, 解码芯片的 DA 引脚向单片机 PA7 引脚输出高电平, 同时封锁其他按键, 片内锁存器将保持当前按键的 4 位编码。

本例单片机 PA3~PA0 分别与 74C922 的 D~A 这 4 位引脚连接。在检测到有键按下时, DA 为高电平, 主程序检测到连接 DA 的 PA7 为高电平时, 即可读取这 4 位按键编码, 得到 0000~1111, 即 0~15 号按键的键值。

2. 实训要求

① 将解码芯片 DA 引脚连接至 PD2(INT0)或 PD3(INT1)中断输入引脚, 用中断程序响应按键操作, 在配置 INT0 或 INT1 时, 注意选择上升沿触发中断。

② 74C923 是 20 键的解码芯片, 当前版本的 Proteus 中未提供该组件, 大家可查阅芯片资料, 设计应用电路并编写程序, 在实物硬件上进行测试。

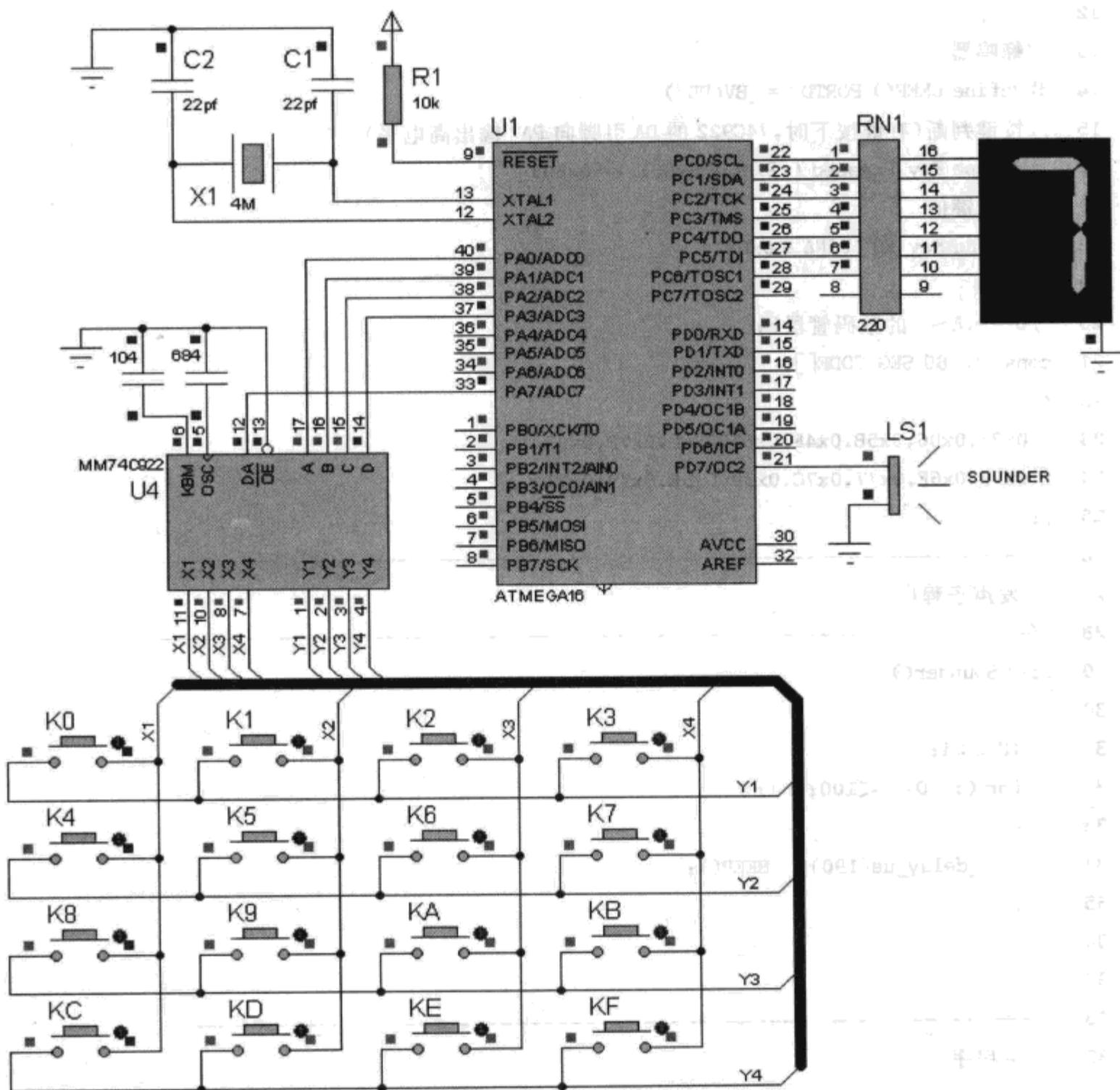


图 4-16 16 键解码芯片 74C922 应用

3. 源程序代码

```

01 //-----
02 // 名称：16 键解码芯片 74C922 应用
03 //-----
04 // 说明：本例因为使用了 74C922 解码芯片，使得程序代码非常简单
05 //          在按下不同按键时，数码管将显示对应键值
06 //
07 //-----
08 #include <avr/io.h>
09 #include <util/delay.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int

```

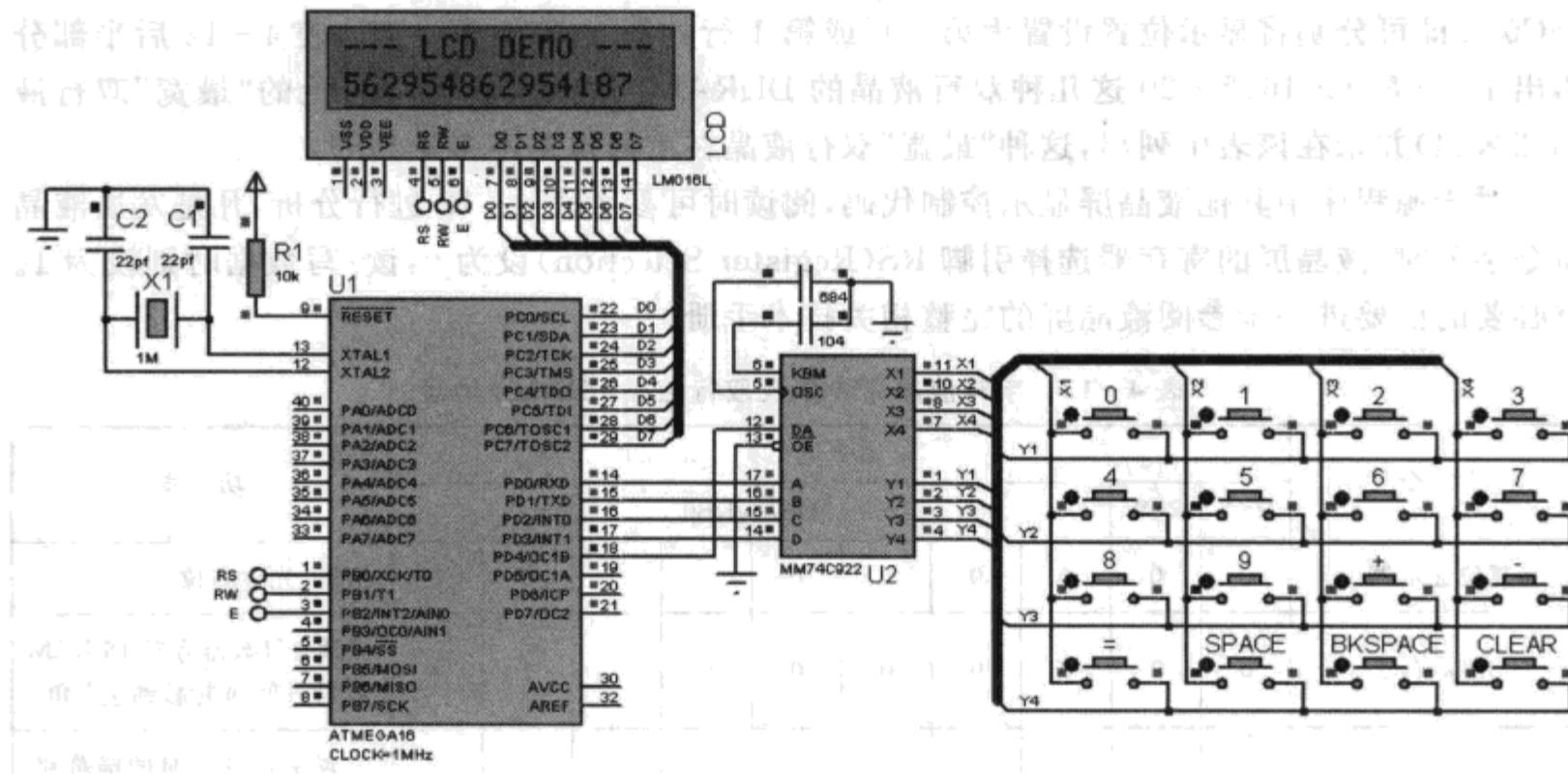
```

12
13 //蜂鸣器
14 #define BEEP() PORTD ^= _BV(PD7)
15 //按键判断(有键按下时,74C922的DA引脚向PA7输出高电平)
16 #define Key_Pressed ((PINB & 0x80) == 0x80)
17 //获取键值
18 #define Key_NO (PINB & 0x0F)
19
20 //0~9,A~F的数码管段码
21 const INT8U SEG_CODE[] =
22 {
23     0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07,
24     0x7F, 0x6F, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71
25 };
26 //-----
27 //发声子程序
28 //-----
29 void Sounder()
30 {
31     INT8U i;
32     for (i = 0; i < 100; i++)
33     {
34         _delay_us(190); BEEP();
35     }
36 }
37
38 //-----
39 //主程序
40 //-----
41 int main()
42 {
43     DDRA = 0x00; PORTA = 0xFF;           //配置端口
44     DDRC = 0xFF; PORTC = 0x00;
45     DDRD = 0xFF; PORTD = 0xFF;
46     while(1)
47     {
48         if (Key_Pressed)                //有键按下
49         {
50             PORTC = SEG_CODE[Key_NO];   //根据键值Key_NO显示按键
51             Sounder();
52         }
53     }
54 }

```

4.14 1602 LCD 字符液晶测试程序

本例使用了基于 HD44780 控制芯片的 1602 液晶显示屏。程序运行时,当前按键值将显示在 1602 液晶屏上。本案例将通用的 LCD 显示控制代码编写在独立的 LCD1602.c 文件中,这样设计可便于以后移植到其他案例。本例电路及部分运行效果如图 4-17 所示。





根据该地址命令格式可知,DDRAM 的最大显示地址范围为 0B10000000~0B11111111,即 0x80~0xFF,最多可设置的字符地址个数为 128 个。对于“最宽”的双行液晶,每行可分配的最大地址个为 64 个,因而可得出如下地址划分:

第 0 行:0x80~0xBF (64 个字符);

第 1 行:0xC0~0xFF (64 个字符)。

由此划分可知,上下两行的起始地址分别为:0x80 与 0xC0,上述代码中的 $0x80 \mid x$ 和 $0xC0 \mid x$ 即可分别将显示位置设置为第 0 行或第 1 行的第 x 个字符位置。表 4-12 后半部分给出了 2×8 、 2×16 、 2×20 这几种双行液晶的 DDRAM 显示地址,刚才讨论的“最宽”双行液晶(2×64)并未在该表中列出,这种“最宽”双行液晶在市面也是不多见到的。

对于源程序中其他液晶屏显示控制代码,阅读时可参考表 4-12 进行分析,凡是发送液晶命令字节时,液晶屏的寄存器选择引脚 RS(Register Selection)设为 0,读/写数据时则设为 1。有必要的话要进一步参阅液晶屏的完整相关技术手册。

表 4-12 字符液晶命令集及双行液晶 DDRAM 地址

命 令	命令位										功 能		
	RS*	R/W	DB7~DB0										
复位显示器	0	0	0	0	0	0	0	0	0	1	清屏,光标归位		
光标归位	0	0	0	0	0	0	0	0	1	*	设地址计数器清零,DDRAM 数据不变,光标移到左上角		
字符进入模式	0	0	0	0	0	0	0	1	I/D	S	设置字符进入时的屏幕移位方式		
显示开关	0	0	0	0	0	0	1	D	C	B	设置显示开关,光标开关,闪烁开关		
显示光标移位	0	0	0	0	0	1	S/C	R/L	*	*	设置字符与光标移动		
功能设置	0	0	0	0	1	DL	N	F	*	*	设置 DL,显示行数,字体		
设置 CGRAM 地址	0	0	0	1	CGRAM 地址						设置 6 位的 CGRAM 地址以读/写数据		
设置 DDRAM 地址	0	0	1	DDRAM 地址						设置 7 位的 DDRAM 地址以读/写数据			
忙标志/地址计数器	0	1	BF	由最后写入的 DDRAM 或 CGRAM 地址设置指令设置的 DDRAM/CGRAM 地址						读忙标志及地址计数器			
CGRAM/DDRAM 写数据	1	0	写入一字节数据(先设置 RAM 地址)						向 CGRAM/DDRAM 写入一字节数据				
CGRAM/DDRAM 读数据	1	1	读取一字节数据(先设置 RAM 地址)						从 CGRAM/DDRAM 读取一字节数据				

续表 4-12

命 令	命令位								功 能								
	RS*	R/W	DB7~DB0														
RS* 为寄存器选择位 RS=0 时选择命令寄存/状态寄存器, RS=1 时选择数据寄存器																	
I/D=1 递增, I/D=0 递减																	
S=0 时显示屏不移动, S=1 时, 如果 I/D=1 且有字符写入时显示屏左移, 否则右移																	
D=1 显示屏开, D=0 显示屏关																	
C=1 时光标出现在地址计数器所指的位置, C=0 时光标不出现																	
B=1 时光标出现闪烁, B=0 时光标不闪烁																	
S/C=0 时, RL=0 则光标左移, 否则右移																	
S/C=1 时, RL=0 则字符和光标左移, 否则右移																	
DL=1 时数据长度为 8 位, DL=0 时为使用 D7~D4 共 4 位, 分两次发送一个字节																	
N=0 为单行显示, N=1 时为双行显示																	
F=1 时为 5×10 点阵字体, F=0 时为 5×7 点阵字体																	
BF=1 时 LCD 忙, BF=0 时 LCD 就绪																	
双行液晶的 DDRAM 地址																	
2×20 LCD DDRAM(80~93/C0~E3)																	
2×16 LCD DDRAM(80~8F/C0~CF)																	
2×8 LCD DDRAM(80~87/C0~C7)																	
80	81	82	83	84	85	86	87	88	89								
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9								
								CA	CB								
								CC	CD								
								CE	CF								
								D0	D1								
								D2	D3								
									90 91 92 93								

2. 实训要求

① 第 3 章中有利用工作于异步模式的 T/C2 控制的可调式数码管电子钟的案例, 在完成本例调试后, 将该案例中的显示器件改用液晶显示屏, 实现相同的显示效果。

② 重新设计第 3 章有关 A/D 转换的案例, 将转换结果显示在 1602 液晶显示屏上。

3. 源程序代码

```

01 //----- LCD1602.C -----
02 // 名称: LCD1602 液晶控制与显示程序
03 //-----
04 #include <avr/io.h>
05 #include <util/delay.h>
06 #define INT8U unsigned char
07 #define INT16U unsigned int
08
09 //LCD 控制引脚定义
10 #define RS PB0 //寄存器选择
11 #define RW PB1 //读/写
12 #define E PB2 //使能
13
14 //LCD 控制端口定义
15 #define LCD_CRTL_PORT PORTB

```

16

```

17 //LCD 数据端口定义
18 #define LCD_PORT  PORTC           //发送 LCD 数据端口
19 #define LCD_PIN   PINC           //读取 LCD 数据端口
20 #define LCD_DDR   DDRC           //LCD 数据端口方向
21

22 //LCD 控制引脚操作定义
23 #define RS_1() LCD_CRTL_PORT |= _BV(RS)
24 #define RS_0() LCD_CRTL_PORT &= ~_BV(RS)
25 #define RW_1() LCD_CRTL_PORT |= _BV(RW)
26 #define RW_0() LCD_CRTL_PORT &= ~_BV(RW)
27 #define EN_1() LCD_CRTL_PORT |= _BV(E)
28 #define EN_0() LCD_CRTL_PORT &= ~_BV(E)
29 //---

30 // LCD 忙等待
31 //---

32 void LCD_BUSY_WAIT()
33 {
34     RS_0();  RW_1();           //读状态寄存器
35     LCD_DDR = 0x00;          //将端口设为输入
36     EN_1();  _delay_us(10);
37     loop_until_bit_is_clear(LCD_PIN,7); //LCD 忙等待,直到 LCD_PIN 最高位为 0
38     EN_0();                //还原 LCD 端口为输出
39
40 }

41 //---

42 //写 LCD 命令寄存器
43 //---

44 void Write_LCD_Command(INT8U cmd)
45 {
46
47     LCD_BUSY_WAIT();
48     RS_0();  RW_0();           //写命令寄存器
49     LCD_PORT = cmd;          //发送命令
50     EN_1();  EN_0();          //写入
51 }

52 //---

53 //写 LCD 数据寄存器
54 //---

55 void Write_LCD_Data(INT8U dat)
56 {
57
58     LCD_BUSY_WAIT();

```

```

59     RS_1();  RW_0();           //写数据寄存器
60     LCD_PORT = dat;         //发送数据
61     EN_1();  EN_0();         //写入
62 }
63
64 //-----
65 // LCD 初始化
66 //-----
67 void Initialize_LCD()
68 {
69     Write_LCD_Command(0x38); _delay_ms(15); //置功能,8位,双行,5x7
70     Write_LCD_Command(0x01); _delay_ms(15); //清屏
71     Write_LCD_Command(0x06); _delay_ms(15); //字符进入模式:屏幕不动,字符后移
72     Write_LCD_Command(0x0C); _delay_ms(15); //显示开,光标
73 }
74
75 //-----
76 // 显示字符串
77 //-----
78 void LCD_ShowString(INT8U x, INT8U y, char * str)
79 {
80     INT8U i = 0;
81     //设置显示起始位置
82     if(y == 0) Write_LCD_Command(0x80 | x); else
83     if(y == 1) Write_LCD_Command(0xC0 | x);
84     //输出字符串
85     for ( i = 0; i < 16 && str[i] != '\0'; i++ )
86         Write_LCD_Data(str[i]);
87 }

01 //----- main.c -----
02 // 名称: 1602LCD 字符液晶测试程序
03 //-----
04 // 说明: 本例运行时,LCD 第一行显示静态字符串,第二行显示键盘矩阵
05 //       所输入的字符,程序支持空格输入、退格和清除
06 //
07 //-----
08 #include <avr/io.h>
09 #include <string.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int
12
13 //可显示的字符表(最后一位是空格符)(分别对应:0x00~0x0D)

```



```
14 const char CHAR_Table[] = "0123456789+-= ";
15 //待显示字符串
16 char Disp_String[17];
17 //有键按下
18 #define Key_Pressed() ((PIND & _BV(PD4)) != 0x00)
19 //键值
20 #define KEY_VALUE (PIND & 0x0F)
21
22 //液晶相关函数
23 extern void Initialize_LCD();
24 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
25 //-----
26 // 主程序
27 //-----
28 int main()
29 {
30     char c;    INT8U sLen;
31     DDRB = 0xFF;                                //配置 LCD 与键盘端口
32     DDRD = 0x00; PORTD = 0xFF;
33     Initialize_LCD();
34     LCD_ShowString(0,0," --- LCD DEMO --- ");
35     while (1)
36     {
37         if (Key_Pressed())
38         {
39             sLen = strlen(Disp_String);
40             if (KEY_VALUE <= 0x0D)          //处理按键 0x00 - 0x0D
41             {
42                 c = CHAR_Table[KEY_VALUE]; //获取新输入的字符
43                 if (sLen < 16)
44                 {
45                     Disp_String[sLen] = c;
46                     Disp_String[sLen + 1] = '\0';
47                 }
48             }
49             else                          //处理按键 0x0E、0x0F(退格和清除)
50             {
51                 switch (KEY_VALUE)
52                 {
53                     case 0x0E:
54                         if (sLen > 0) Disp_String[sLen - 1] = '\0';
55                         break;
56                     case 0x0F:
57                 }
58             }
59         }
60     }
61 }
```

```

57     Disp_String[0] = '\0';
58
59 }
60 } //LCD 清除并重新显示
61 LCD_ShowString(0,1,"");
62 LCD_ShowString(0,1,Disp_String);
63 while (Key_Pressed());
64 }
65 }
66 }

```

4.15 1602 液晶显示 DS1302 实时时钟

DS1302 是 DALLAS 公司推出的一种高性能、低功耗、带 RAM 的实时时钟芯片，它可以对年、月、日、周、时、分、秒进行计时，具有闰年补偿功能，最大有效年份可达 2100 年。运行本例时，单片机不断从 DS1302 读取当前日期时间信息并刷新显示在 1602 液晶显示屏上。本例电路及运行效果如图 4-18 所示。

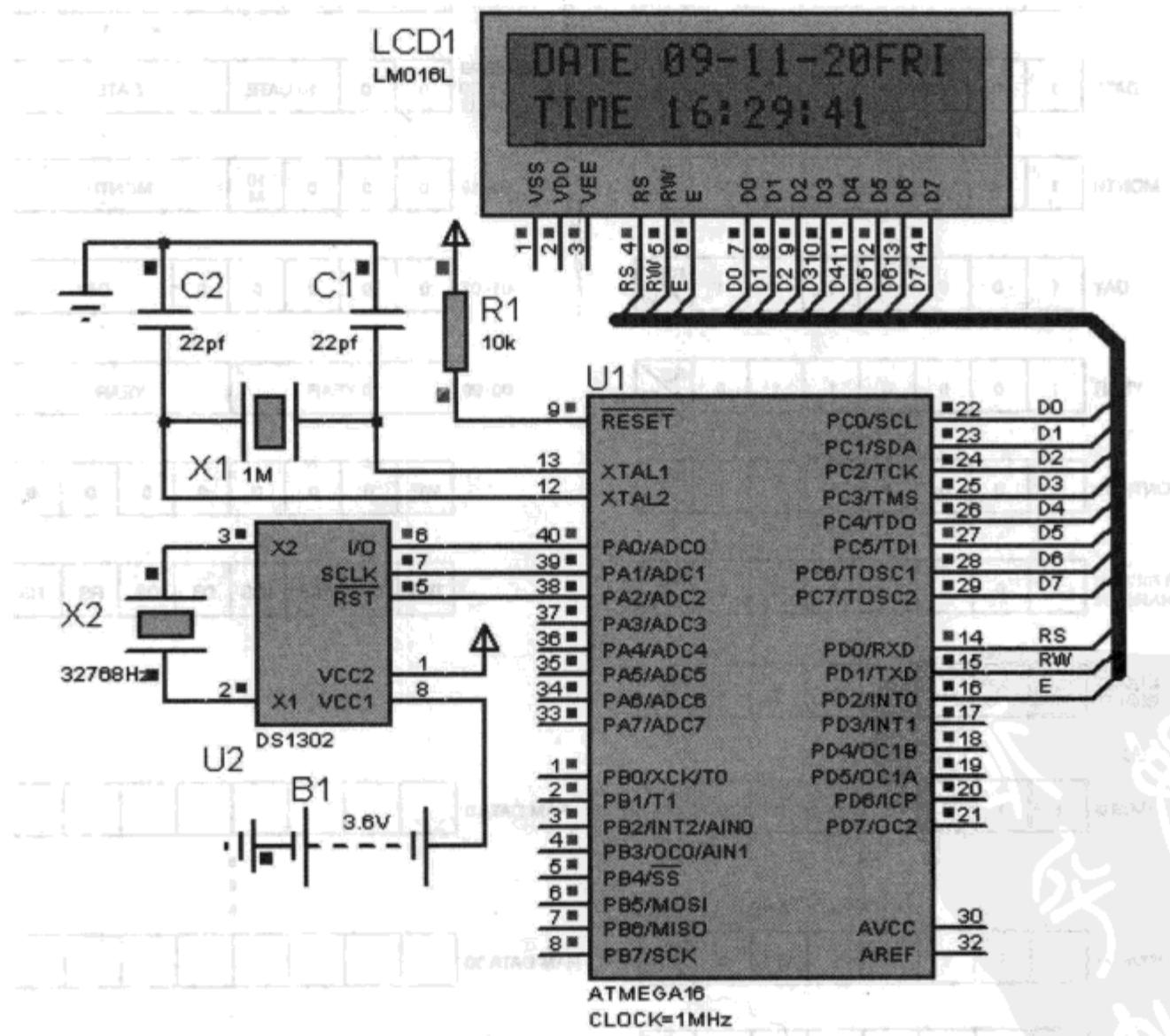


图 4-18 1602 液晶显示 DS1302 实时时钟

1. 程序设计与调试

本例的液晶显示驱动程序与上一案例相似，使用时仅需要修改数据与控制端口定义即可。

DS1302 实时时钟程序在后续多个案例中还会使用,与上一案例中的 LCD1602 一样,本例将其单独编写在 DS1302.c 文件中。

在编写 DS1302 读取当前日期时间的函数 GetTime 时,可参考图 4-19 所示的 DS1302 地址/命令字节(ADDRESS/COMMAND BYTE)格式、A. 时钟(CLOCK)与 B. RAM 的寄存器地址(REGISTER ADDRESS)与寄存器定义(REGISTER DEFINITION)。在该函数中,addr

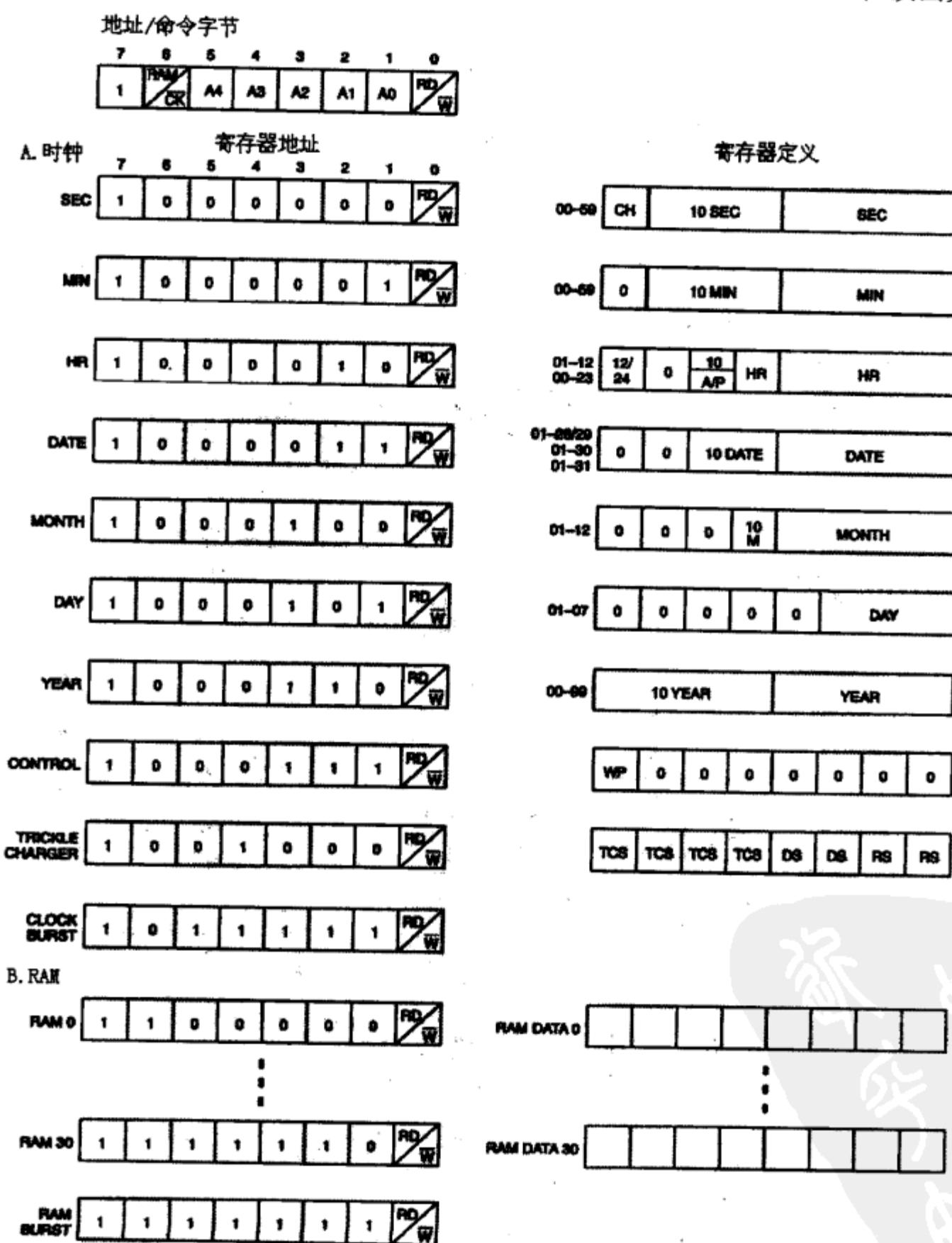


图 4-19 DS1302 时钟及 RAM 寄存器地址与定义

初值为 0x81(即 10000001),最高两位 10 表示要读/写 CLOCK 数据(如果为 11 则表示要读/写 RAM 数据),最后一位为 1 表示读(RD),其余 5 位 A4A3A2A1A0 为 00000,表示访问的是秒(SEC)寄存器,可见该函数将从秒开始读取 7 个字节数据,分别是秒、分、时、日、月、周、年。函数中地址每次递增 2,这是因为 CLOCK 寄存器地址第 0 位为读/写(RD/W)位,在本函数保持为 1,最低地址位从第 1 位开始,该位每次递增 1 时,相当于地址递增 2。

编写 DS1302 字节读/写函数 Get_Byte_FROM_DS1302 与 Write_Byte_TO_DS1302 时,要参考图 4-20 所示的时序图,时序图上半部分是读单字节的时序,下半部分的是写单字节的时序,图中 R/C 即 RAM/CLOCK。根据时序图可知,读/写 DS1302 时要首先写入地址,在写入地址字节时,要由低位到高位逐位写入,读取数据也是由低位到高位逐位读取。

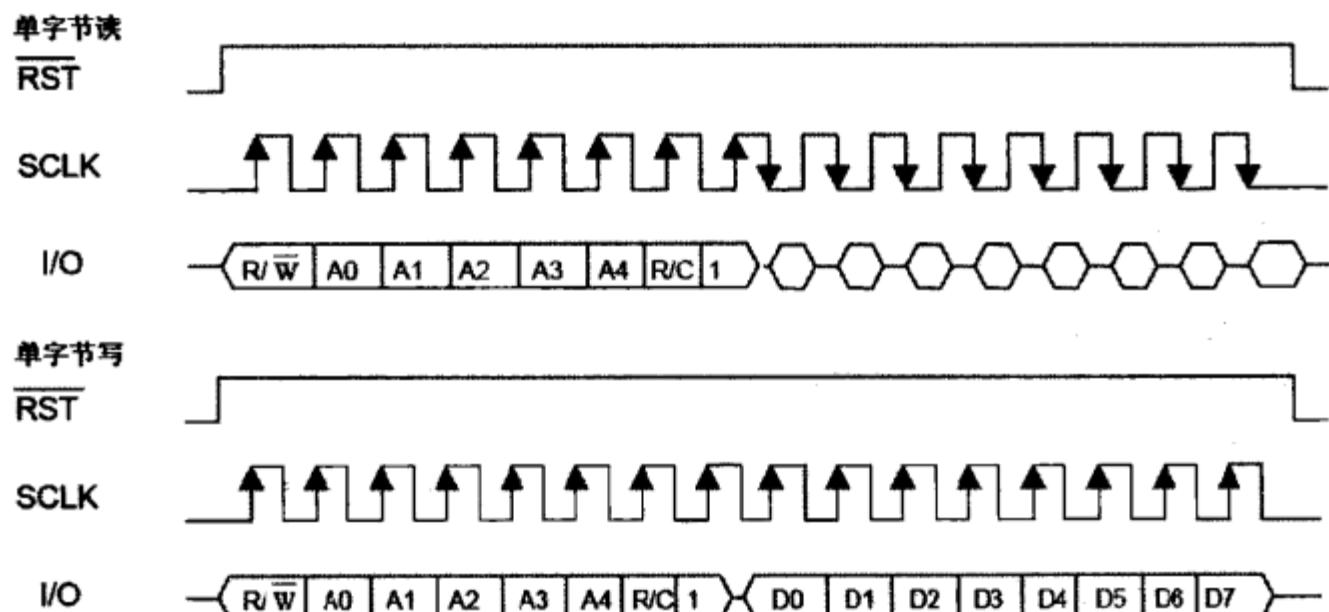


图 4-20 DS1302 单字节读/写时序

另外,还要注意 DS1302 所保存的数据是 BCD 码,在读/写时要注意转换。本例中从 DS1302 读取一字节的函数在返回值之前用表达式 $dat / 16 * 10 + dat \% 16$ 进行转换。

每次运行本例时,LCD 所显示的都是 PC 机时间,这是因为在 DS1302 的属性设置中,默认选中了自动根据 PC 机时间初始化(Automatically Initialize from PC Clock?)选项。如果取消此项(注意:不能在选项中出现问号),再次运行本例时所显示的日期时间将全部为 0。在后续相关案例中将讨论如何将调整后的时间写入 DS1302。

2. 实训要求

① 取消 PC 机时钟初始化设置,编程在显示时间之前先写入某个自定的初始时间,然后由该时间开始显示日期与时间信息。

② 重新设计本例,改用数码管显示当时日期时间信息。

3. 源程序代码

```

01 //----- DS1302.c -----
02 // 名称: DS1302 实时时钟程序
03 //-----
04 #include <avr/io.h>
05 #include <util/delay.h>
06 #define INT8U unsigned char

```



```
07 #define INT16U unsigned int
08
09 //DS1302 引脚定义
10 #define IO PA0
11 #define SCLK PA1
12 #define RST PA2
13
14 //DS1302 端口定义
15 #define DS_PORT PORTA
16 #define DS_DDR DDRA
17 #define DS_PIN PINA
18
19 //DS1302 端口数据读/写(方向)
20 #define DDR_IO_RD() DS_DDR &= ~_BV(IO)
21 #define DDR_IO_WR() DS_DDR |= _BV(IO)
22
23 //DS1302 控制引脚操作定义
24 #define WR_IO_0() DS_PORT &= ~_BV(IO) //DS1302 I/O 线(W/R)
25 #define WR_IO_1() DS_PORT |= _BV(IO)
26 #define RD_IO() (DS_PIN & _BV(IO)) //注意:此行的括号不可省略
27 #define SCLK_1() DS_PORT |= _BV(SCLK) //DS1302 时钟线
28 #define SCLK_0() DS_PORT &= ~_BV(SCLK)
29 #define RST_1() DS_PORT |= _BV(RST) //DS1302 复位线
30 #define RST_0() DS_PORT &= ~_BV(RST)
31
32 //0、1、2、3、4、5、6 分别对应周日、周一~周六
33 char * WEEK[] = {"SUN", "MON", "TUS", "WEN", "THU", "FRI", "SAT"};
34 //所读取的日期时间
35 INT8U DateTime[7];
36 //-----
37 // 向 DS1302 写入 1 字节
38 //-----
39 void Write_Byteto_DS1302(INT8U x)
40 {
41     INT8U i;
42     DDR_IO_WR(); //写 DS1302 I/O 口
43     for(i = 0x01; i != 0x00; i <<= 1) //写 1 字节(上升沿写入)
44     {
45         if (x & i) WR_IO_1(); else WR_IO_0(); SCLK_0(); SCLK_1();
46     }
47 }
48
49 //-----
50 // 从 DS1302 读取 1 字节
```

```

51 //-----
52 INT8U Get_Byte_FROM_DS1302()
53 {
54     INT8U i,dat = 0x00;
55     DDR_IO_RD();                                //读 DS1302 IO 口
56     for(i = 0; i<8 ; i++)                      //串行读取 1 字节(下降沿读取)
57     {
58         SCLK_1(); SCLK_0(); if (RD_IO()) dat |= _BV(i);
59     }
60     return dat / 16 * 10 + dat % 16;           //将 BCD 码转换为十进制数并返回
61 }
62
63 //-----
64 // 从 DS1302 指定位置读数据
65 //-----
66 INT8U Read_Data(INT8U addr)
67 {
68     INT8U dat;
69     RST_1();                                     //将 RST 拉高
70     Write_Bit_TO_DS1302(addr);                  //向 DS1302 写地址
71     dat = Get_Byte_FROM_DS1302();                //从指定地址读字节
72     RST_0();                                     //将 RST 拉低
73     return dat;
74 }
75
76 //-----
77 // 读取当前日期时间
78 //-----
79 void GetDateTime()
80 {
81     INT8U i,addr = 0x81;                         //从读秒地址 0x81 开始
82     for (i = 0; i<7; i++)                      //依次读取 7 字节,分别是秒、分、时、日、月、周、年
83     {
84         DateTime[i] = Read_Data(addr);
85         addr += 2;                             //读日期时间地址依次为 0x81,0x83,0x85...
86     }
87 }

01 //----- main.c -----
02 // 名称: 1602LCD 液晶显示 DS1302 实时实钟
03 //-----
04 // 说明: 本例运行时,程序每隔 0.5 s 读取 DS1302 实时时钟芯片时间数据,
05 //        通过格式转换后显示在 1602LCD 上
06 //

```



```
07 //-----  
08 #define F_CPU 1000000UL  
09 #include <avr/io.h>  
10 #include <util/delay.h>  
11 #include <string.h>  
12 #include <stdio.h>  
13 #define INT8U unsigned char  
14 #define INT16U unsigned int  
15  
16 //液晶相关函数  
17 extern void Initialize_LCD();  
18 extern void LCD_ShowString(INT8U x, INT8U y,char * str);  
19 // DS1302 相关函数与数据  
20 extern void GetTime();  
21 extern INT8U DateTime[];  
22 extern char * WEEK[];  
23 //LCD 显示缓冲  
24 char LCD_DSY_BUFFER[17];  
25 //-----  
26 // 主程序  
27 //-----  
28 int main()  
29 {  
30     DDRA = 0xFF;                      //端口配置  
31     DDRC = 0xFF;  DDRD = 0xFF;  
32     Initialize_LCD();                //初始化 LCD  
33     while (1)  
34     {  
35         GetTime();                  //读取 DS1302 实时时钟  
36         //按格式：“DATE 00-00-00XXX”显示年月日与星期  
37         sprintf(LCD_DSY_BUFFER,"DATE %02d-%02d-%02d%3s",  
38                 DateTime[6],DateTime[4],DateTime[3],WEEK[DateTime[5] - 1]);  
39         LCD_ShowString(0,0,LCD_DSY_BUFFER);  
40  
41         //按格式：“TIME 00:00:00”显示时分秒  
42         sprintf(LCD_DSY_BUFFER,"TIME %02d:%02d:%02d",  
43                 DateTime[2],DateTime[1],DateTime[0]);  
44         LCD_ShowString(0,1,LCD_DSY_BUFFER);  
45  
46         _delay_ms(100);  
47     }  
48 }
```

4.16 1602 液晶工作于 4 位模式实时显示当前时间

本例 1602LCD 仅使用低 4 位数据线与 3 位控制线实现液晶显示，减少了单片机的端口引脚占用，程序运行效果与上一案例相同。本例电路如图 4-21 所示。

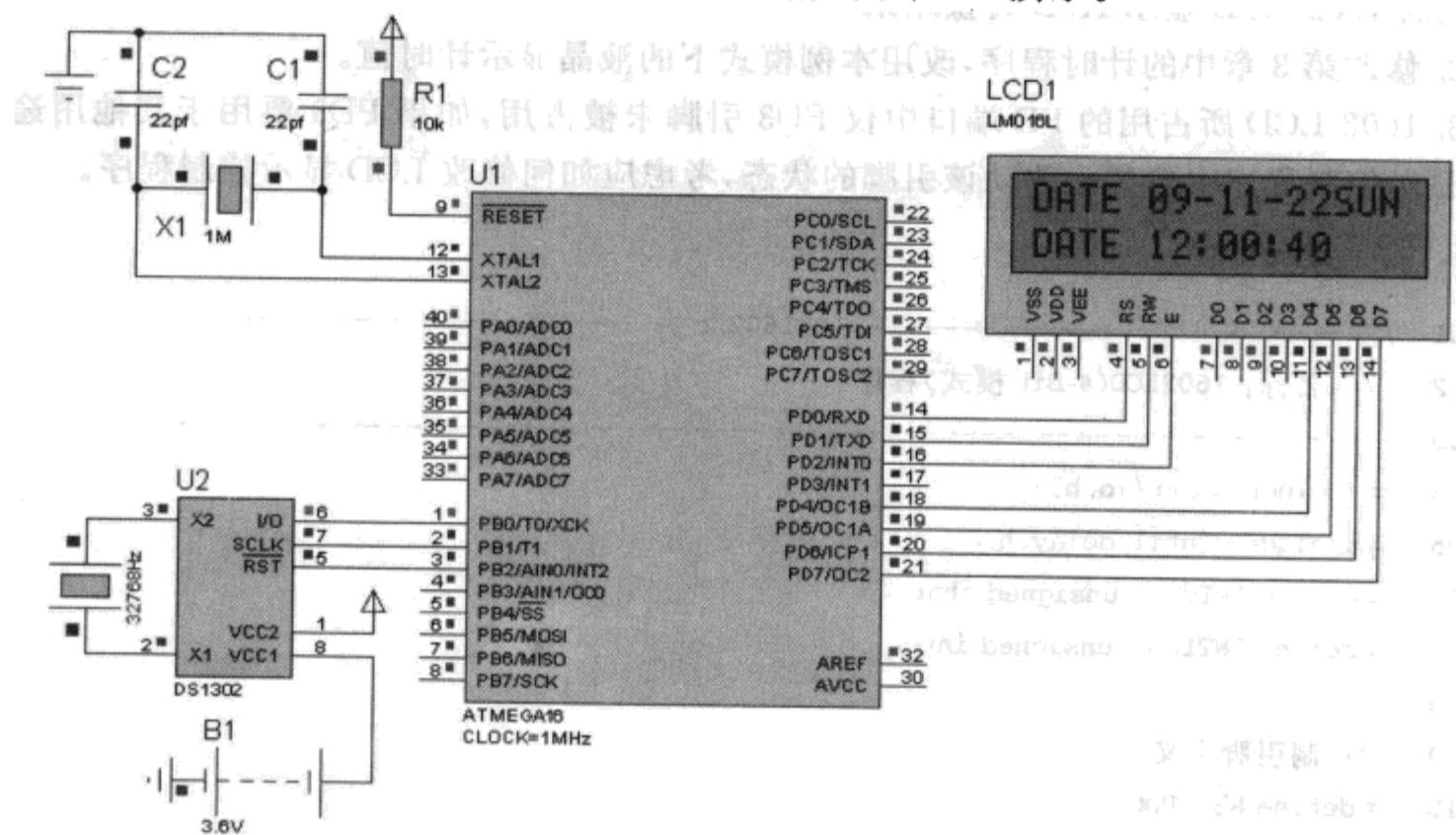


图 4-21 1602 液晶工作于 4 位模式实时显示当前时间

1. 程序设计与调试

本例源程序文件有：LCD1602.c、DS1302.c、main.c。后 2 个源程序文件与上一案例类似，引用 DS1302.c 时只需要对数据与控制引脚定义进行修改即可，在主程序中则需要对端口配置进行相应修改。

本例中 1602 液晶的 8 位数据线中仅使用了低 4 位的 D0~D4，在读/写字节时需要分为两次，先读/写高 4 位，后读/写低 4 位。

表 4-13 单独给出了 4-Bit 模式的设置，在初始程序中需要先给命令/数据端口发送命令 0x20（即 00100000），它将 DL 设为 0，选择 4 位模式，这条命令仅通过一次写入完成，最开始的 4Bit 设置命令是在默认的 8 位模式下进行的。

初始化程序随后又发送了命令 0x28，这发送的仍是功能设置命令，即 00101000，高 4 位没有变，低 4 位设置 N=1、F=0，选择双行、5×7 点阵字体，不同的是发送的 0x28 即其他数据/命令都是分两次发送的，先发高 4 位，后发低 4 位。

表 4-13 字符液晶 4Bit 模式设置

功能设置	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
设置 DL，显示行数，字体	0	0	0	0	1	DL	N	F	*	*

注：DL=0 时使用 D7~D4 共 4 位，分两次发送一个字节，N=0 为单行显示，N=1 时为双行显示。

F=1 时为 5×10 点阵字体，F=0 时为 5×7 点阵字体。

其他所有命令都可参考表 4-12 所列的液晶控制命令集,不同的是本例对这些命令的发送都要分两次完成。

2. 实训要求

- ① 第 3 章中有关 A/D 转换的结果是用数码管显示的,完成本例调试后改成用工作于 4 位模式的 1602 LCD 显示 A/D 转换结果。
- ② 修改第 3 章中的计时程序,改用本例模式下的液晶显示计时值。
- ③ 1602 LCD 所占用的 PD 端口中仅 PD3 引脚未被占用,如果 PD3 要用于其他用途,在 LCD 显示控制程序中要禁止改动该引脚的状态,考虑应如何修改 LCD 显示控制程序。

3. 源程序代码

```
001 //----- LCD1602.c -----  
002 // 名称: 1602LCD(4-Bit 模式)程序  
003 //-----  
004 #include <avr/io.h>  
005 #include <util/delay.h>  
006 #define INT8U unsigned char  
007 #define INT16U unsigned int  
008  
009 //控制引脚定义  
010 #define RS PD0  
011 #define RW PD1  
012 #define EN PD2  
013  
014 //液晶端口定义  
015 #define LCD_PORT PORTD  
016 #define LCD_PIN PIND  
017 #define LCD_DDR DDRD  
018  
019 //控制引脚操作  
020 #define EN_1() LCD_PORT |= _BV(EN)  
021 #define RS_1() LCD_PORT |= _BV(RS)  
022 #define RW_1() LCD_PORT |= _BV(RW)  
023 #define EN_0() LCD_PORT &= ~_BV(EN)  
024 #define RS_0() LCD_PORT &= ~_BV(RS)  
025 #define RW_0() LCD_PORT &= ~_BV(RW)  
026 //-----  
027 // 液晶忙等待  
028 //-----  
029 void LCD_BUSY_WAIT()  
030 {  
031     INT8U Hi,Lo ;  
032     LCD_DDR = 0x0F; //高 4 位设为输入,以便读取忙状态
```

```

033     do
034     {
035         LCD_PORT = 0x00;
036         RW_1(); RS_0();
037         EN_1(); _delay_us(3); Hi = LCD_PIN; EN_0();
038         EN_1(); _delay_us(3); Lo = LCD_PIN; EN_0();
039     } while( Hi & 0x80);
040 }
041
042 //-----
043 // 写液晶命令
044 //-----
045 void Write_LCD_Command (INT8U cmd)
046 {
047     LCD_BUSY_WAIT();
048     LCD_DDR = 0xFF;           //设置方向为输出
049     LCD_PORT = cmd & 0xF0;   //输出高 4 位
050     RW_0(); RS_0(); EN_1(); _delay_us(2); EN_0();
051     LCD_PORT = cmd << 4;    //输出低 4 位
052     RW_0(); RS_0(); EN_1(); _delay_us(2); EN_0();
053 }
054
055 //-----
056 // 向液晶写数据
057 //-----
058 void Write_LCD_Data(INT8U dat)
059 {
060     LCD_BUSY_WAIT();
061     LCD_DDR = 0xFF;           //设置方向为输出
062     LCD_PORT = dat & 0xF0;   //输出高 4 位
063     RW_0(); RS_1(); EN_1(); _delay_us(2); EN_0();
064     LCD_PORT = dat << 4;    //输出低 4 位
065     RW_0(); RS_1(); EN_1(); _delay_us(2); EN_0();
066 }
067
068 //-----
069 // 液晶初始化
070 //-----
071 void Initialize_LCD()
072 {
073     LCD_DDR = 0xFF;           //液晶数据端口设为输出
074     LCD_PORT = 0x20;          //通过功能设置命令设置为 4 位模式
075     EN_1(); _delay_us(2); EN_0(); _delay_us(40);

```



```
076
077 //以下每条命令都需要通过分别发送高 4 位与低 4 位完成
078 Write_LCD_Command(0x28); //功能设置(4-Bit,双行,5 * 7 点阵)
079 Write_LCD_Command(0x08); //关显示
080 Write_LCD_Command(0x01); //清屏
081 Write_LCD_Command(0x06); //模式设置
082 Write_LCD_Command(0x0C); //开显示
083 Write_LCD_Command(0x02); //光标归于左上角
084 }
085
086 //-----
087 // 将光标定位于指定行列
088 //-----
089 void Set_LCD_Pos(INT8U x, INT8U y)
090 {
091     if(y == 0) Write_LCD_Command(0x80 | x); else
092         if(y == 1) Write_LCD_Command(0xC0 | x);
093 }
094
095 //-----
096 // 在指定位置显示字符串
097 //-----
098 void LCD_ShowString(INT8U x, INT8U y,char * str)
099 {
100     INT8U i = 0;
101     //设置显示起始位置
102     Set_LCD_Pos(x,y);
103     //输出字符串
104     for ( i = 0; i<16 && str[i]!='\0';i++)
105         Write_LCD_Data(str[i]);
106 }
```

4.17 2×20 串行字符液晶演示

本例液晶屏同样基于 HD44780 控制芯片,它连接单片机串口,显示单片机串口所发送的字符信息。运行本例时,虚拟终端 VT1—INPUT 中输入的字符也可以显示在串行液晶屏上。本例电路及运行效果如图 4-22 所示。

1. 程序设计与调试

第 3 章的基础程序设计部分已经提供了有关单片机串口程序设计的案例。设计本例时仍要注意单片机、虚拟终端及串行液晶的波特率设置要保持一致(9600,8,1)。

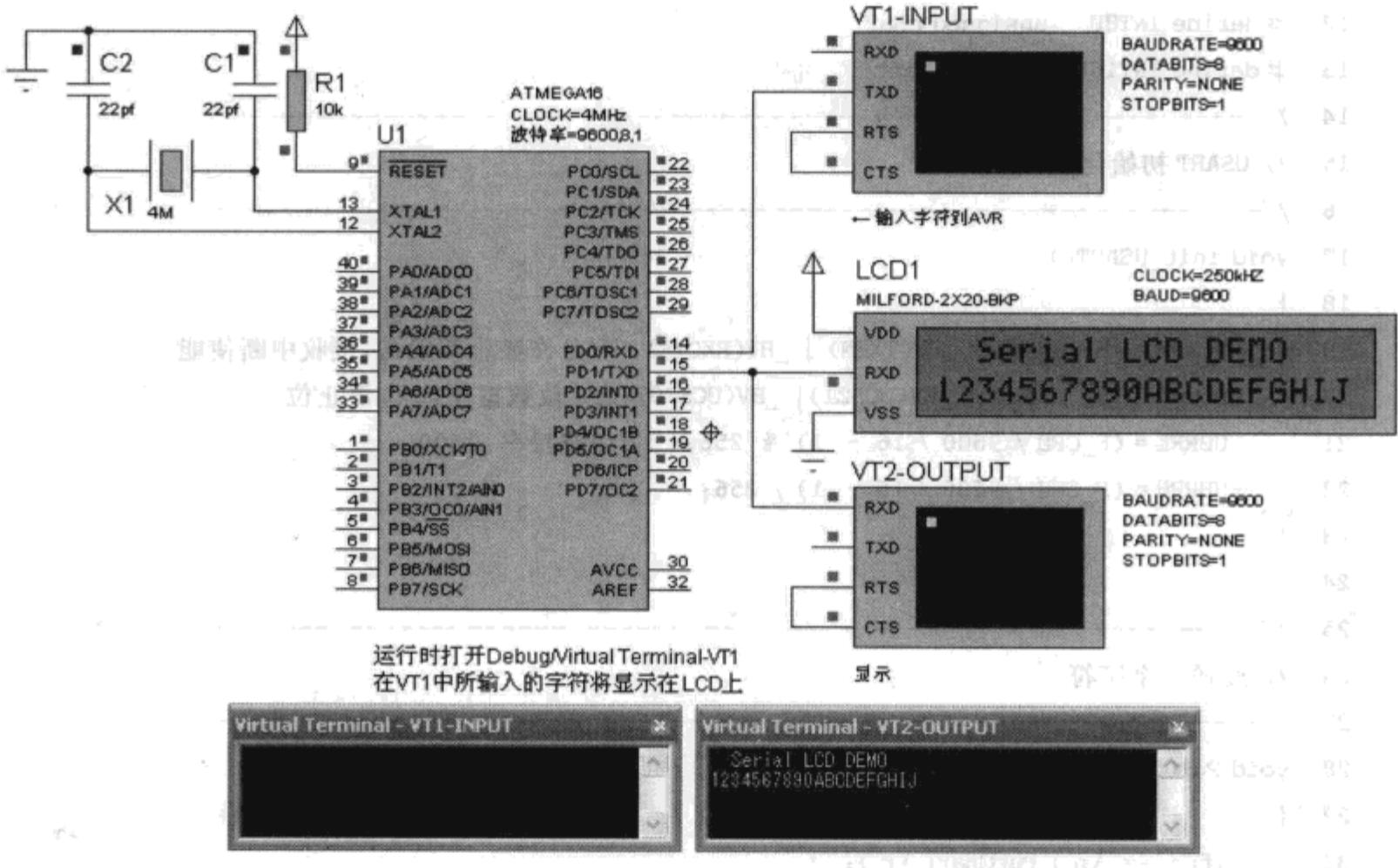


图 4-22 2×20 串行字符液晶演示

阅读该液晶的控制程序时可参考表 4-12 提供的液晶指令集及 2×20 液晶的地址表, 不同的是每次向串行液晶写入命令时需要先写入 0xFE, 然后写入命令字节。查阅 HD44780 技术手册中的字模表(16x16)时会发现, 0xFE 编码未分配给任何可显示字符, 凡以 0xFE 为前导的后续一字节为命令字节, 而非待显示字符的编码字节。

在本例串行液晶上发送字符数据时直接通过 PutChar 写入即可。

2. 实训要求

- ① 参阅液晶命令表, 关闭本例中的光标。
- ② 将以前液晶显示实时时钟案例中的液晶屏改为串行液晶并重新设计, 实现相同功能。

3. 源程序代码

```

01 //-----
02 // 名称: 2 * 20 串行液晶演示
03 //-----
04 // 说明: 程序执行时串行液晶上显示:Serial LCD DEMO
05 //       当光标在第二行闪烁时,虚拟终端中输入的字符将显示在
06 //       LCD 上,按下退格键时光标左移,按下回车键时清屏
07 //-----
08 #define F_CPU 4000000UL          //4 MHz 晶振
09 #include <avr/io.h>
10 #include <avr/interrupt.h>
11 #include <util/delay.h>

```



```
12 #define INT8U unsigned char
13 #define INT16U unsigned int
14 //-
15 // USART 初始化
16 //-
17 void Init_USART()
18 {
19     UCSRB = _BV(RXEN) | _BV(TXEN) | _BV(RXCIE); //允许接收和发送,接收中断使能
20     UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0); //8 位数据位,1 位停止位
21     UBRRH = (F_CPU / 9600 / 16 - 1) % 256;      //波特率:9600
22     UBRRL = (F_CPU / 9600 / 16 - 1) / 256;
23 }
24
25 //-
26 // 发送一个字符
27 //-
28 void PutChar(char c)
29 {
30     if(c == '\n') PutChar('\r');
31     UDR = c;
32     while(!(UCSRA & _BV(UDRE)));
33 }
34
35 //-
36 // 发送字符串
37 //-
38 void PutStr(char * s)
39 {
40     INT8U i = 0;
41     while (s[i] != '\0')
42     {
43         PutChar(s[i++]);
44         _delay_ms(5);
45     }
46 }
47
48 //-
49 // 写 LCD 命令
50 //-
51 void Write_LCD_COMMAND(INT8U comm)
52 {
53     PutChar(0xFE);                      //发送串行液晶命令先导字节 0xFE
54     PutChar(comm);                     //发送命令字节
```

```

55 }
56
57 //-----
58 // 主程序
59 //-----
60 int main()
61 {
62     Init_USART();           //串口初始化
63     sei();                 //开总中断
64     _delay_ms(300);        //等待液晶初始化完成
65     PutStr(" Serial LCD DEMO "); //在 LCD 上显示提示字符串
66     Write_LCD_COMMAND(0xC0); //光标定位到第二行
67     Write_LCD_COMMAND(0x0D); //显示光标
68     while (1);            //等待中断接收并显示
69 }
70
71 //-----
72 // 串口接收中断函数
73 //-----
74 ISR (USART_RXC_vect)
75 {
76     INT8U c = UDR;
77     if ( c == 0x0D )       //按下回车键时 LCD 清屏
78     {
79         Write_LCD_COMMAND(0x01);
80         return;
81     }
82     if ( c == 0x08 )       //按下退格键时光标后移
83     {
84         Write_LCD_COMMAND(0x10);
85         return;
86     }
87     PutChar( c );          //在串行 LCD 上显示输入的字符
88 }

```

4.18 LGM12864 液晶显示程序

本例演示了 LGM12864 液晶屏的汉字显示效果,该液晶屏使用 KS0108 控制芯片。阅读调试本例时需要参考该液晶的指令表及相关技术手册文件。本例电路如图 4-23 所示。

1. 程序设计与调试

本例源程序文件包括 LCD12864.c 与 main.c,前者提供了 LGM12864 液晶显示驱动程



序,main.c 提供了由 Zimo 软件取得的“液晶屏测试程序”这一行汉字的点阵字模,并调用液晶显示程序根据字模数据显示对应的汉字。下面首先谈一下取模方法。

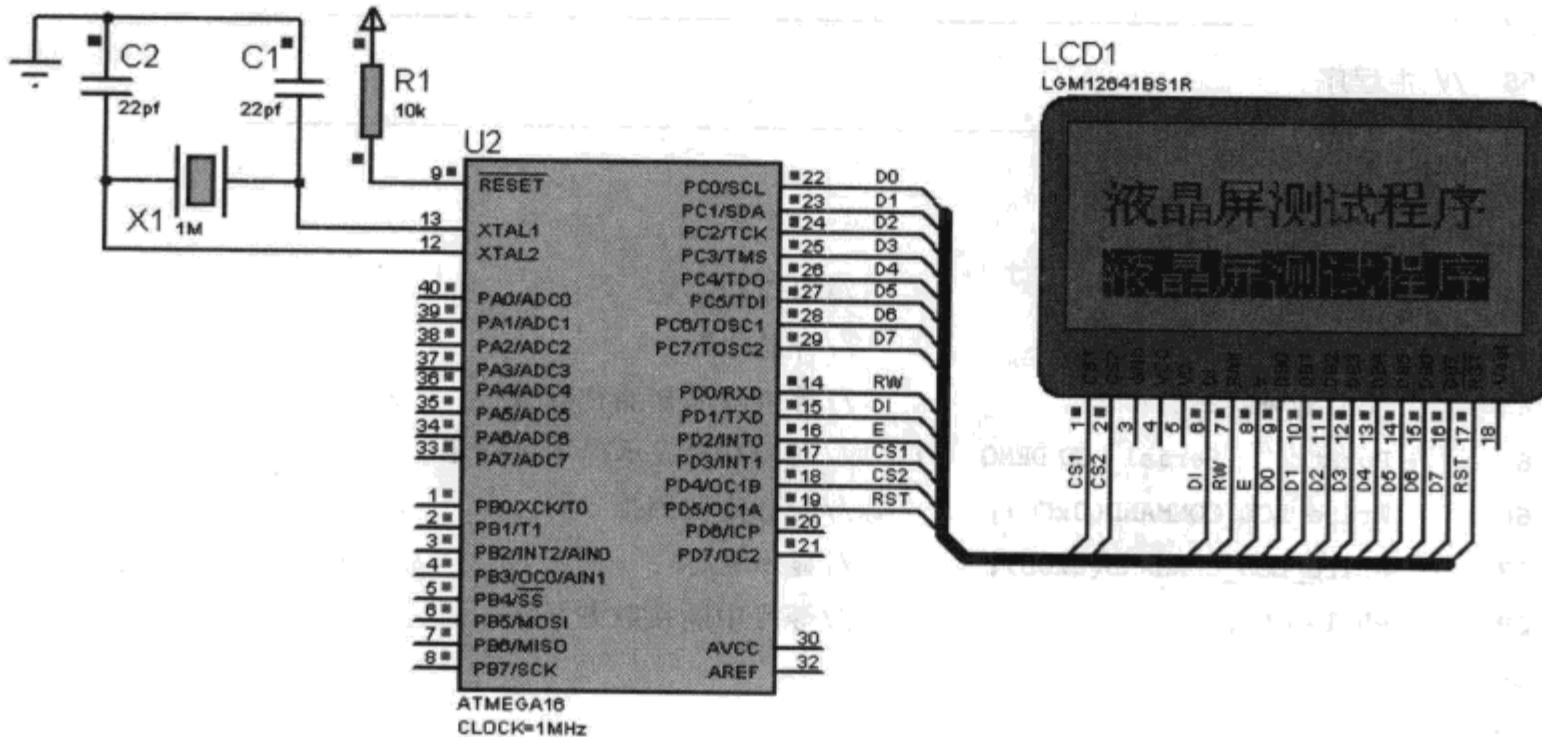


图 4-23 LGM12864 液晶显示程序

根据图 4-24 所示的 LGM12864(KS0108)液晶像素与显示 RAM 的映射关系(左半屏)可知,在显示字符“A”时,首先输出的是它最左边的像素,也就是第一列像素,且高位在下,低位在上,然后再输出第 2 列、第 3 列,每列 8 位(1 字节)。

获取待显示汉字的点阵数据时,本例使用了字模软件 Zimo,为获取本例所用的汉字点阵,首先要在图 4-25 所示窗口中单击“参数设置/文字输入区字体选择”,将字体设为宋体小四号,然后在文字输入区中输入“液晶屏测试程序”并按下 Ctrl+Enter,接着单击图 4-25 所示窗口中的“参数设置/其他选项”,按图 4-26 所示对话框设置取模方式为“纵向取模,字节倒序”,最后单击“取模方式/C51 格式”即可生成汉字点阵数据,生成的数据显示在“点阵生成区”中,这些数据直接复制粘贴到源程序中即可。

下面再来讨论如何编写 LGM12864 液晶驱动程序:

① 显示定位:对照表 4-14 与图 4-25 中 Page0~Page7 可知,设置 X 地址即页地址,在设置页地址后还需要设置列地址(Y),如果不从该页第 0 行开始显示时,还需要设置行地址。根据表 4-14,在 LCD12864.c 的开始部分定义了以下 3 条常用指令:

```
#define LCD_START_ROW      0xC0      //起始行
#define LCD_PAGE             0xB8      //页指令
#define LCD_COL              0x40      //列指令
```

② 通用显示函数 Common_Show 的编写:由于 12864 是由 64×64 左半屏和右半屏构成的,通过设置 CS1 与 CS2 为 10 与 01 可分别选择左半屏与右半屏进行显示操作。通用显示函数的参数为 P、L、W、* r,它们表示从第 P 页(X 地址)开始,在左边距为 L 的位置开始显示 W 个字节,字节缓冲地址为 r。

在上述参数中,P 取值只能为 0~7,L 取值可以为 0~127,在函数内部要将 0~127 分成左右两半屏分别进行控制。该函数内部给出了非常完整的说明,大家可以仔细对比图 4-25

及表 4-14 进行分析。

另外,本例液晶的数据/命令引脚 DI(Data/Instruction)用于选择发送命令字节还是读/写数据字节,DI=0 时选择命令寄存器,DI=1 时则选择数据寄存器,该引脚类似于 1602 LCD 中的寄存器选择引脚 RS。

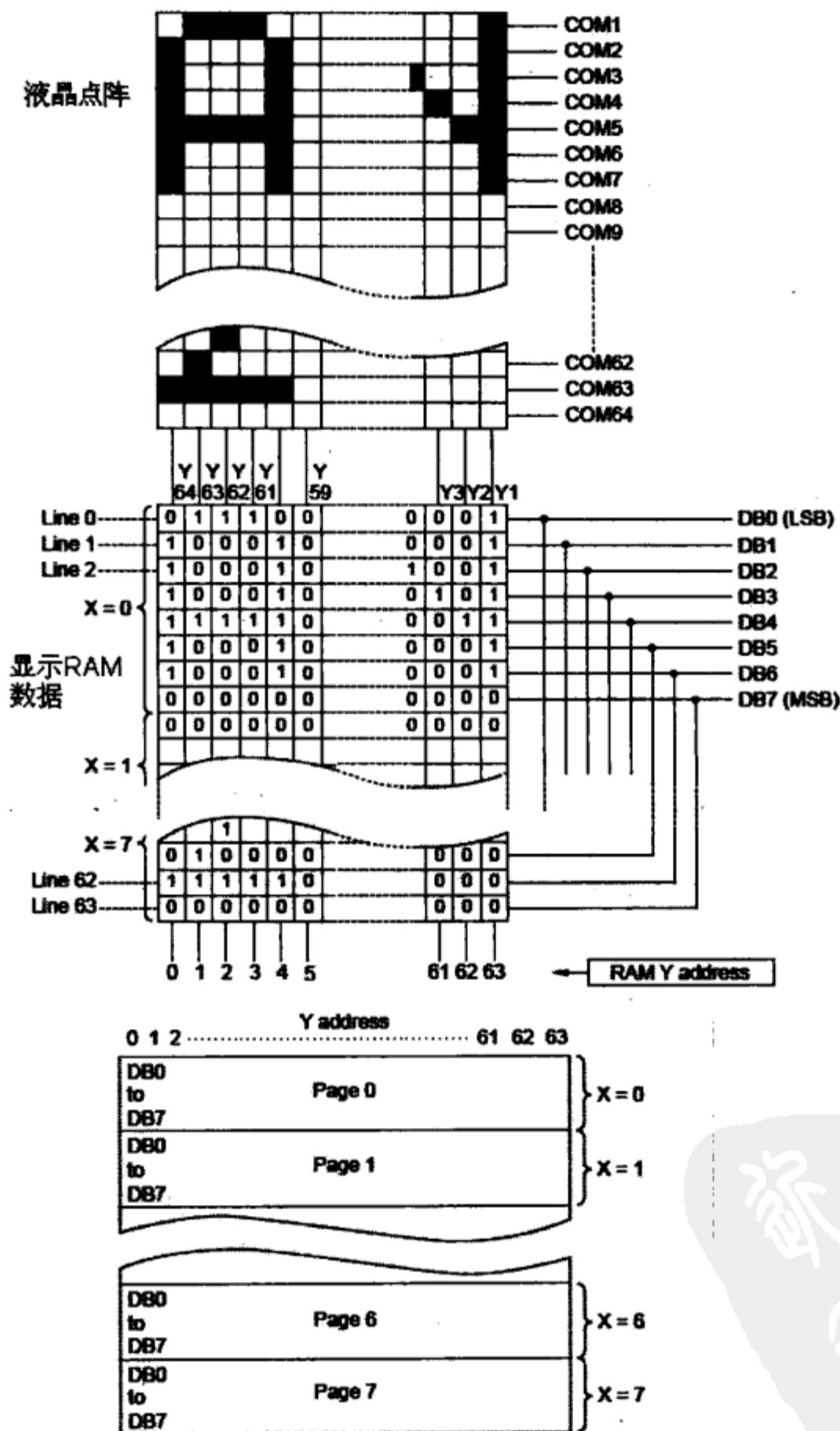


图 4-24 LGM12864(KS0108) 液晶像素与显示 RAM 的映射关系(左半屏)

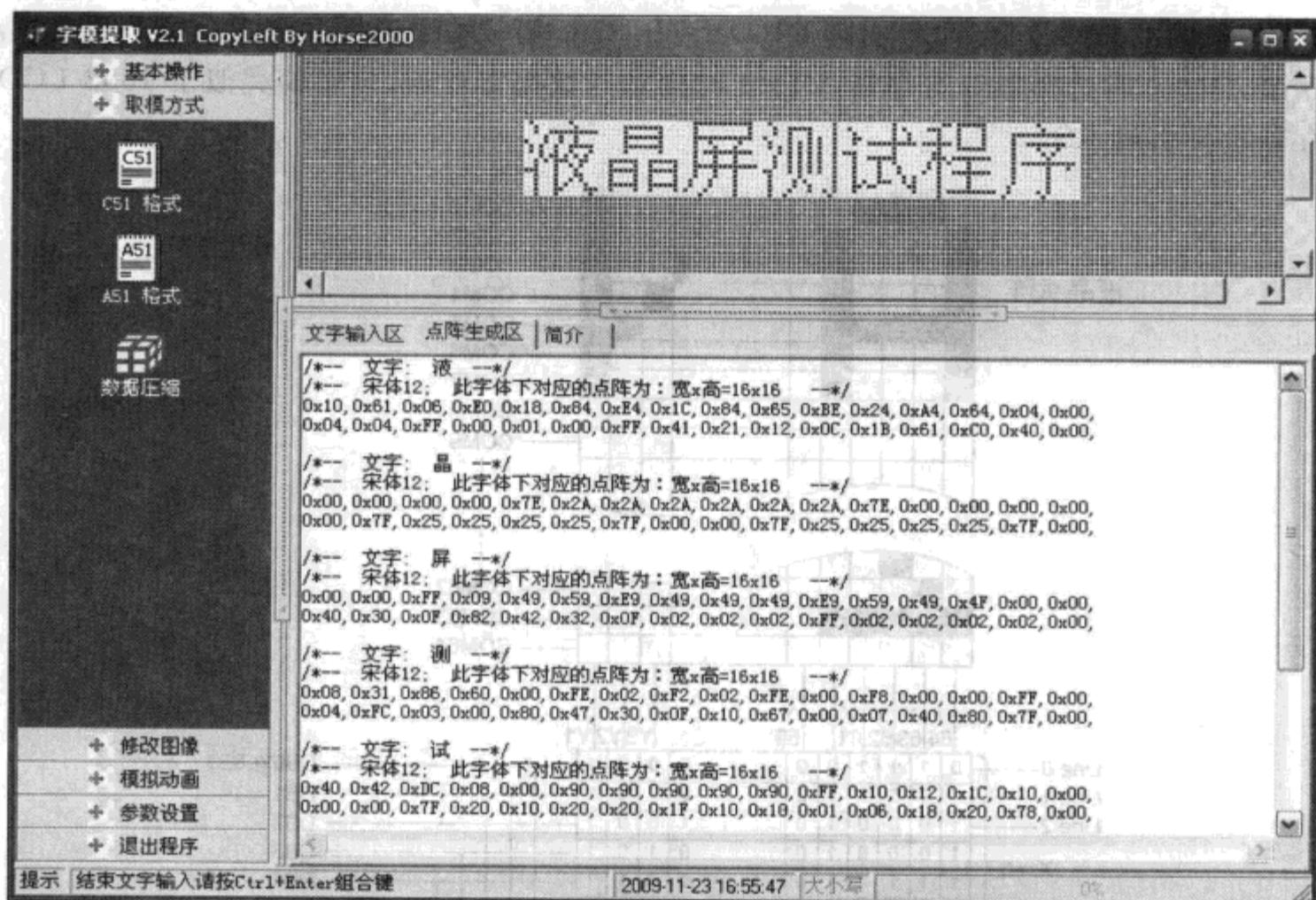


图 4-25 用 Zimo 软件提取汉字字模

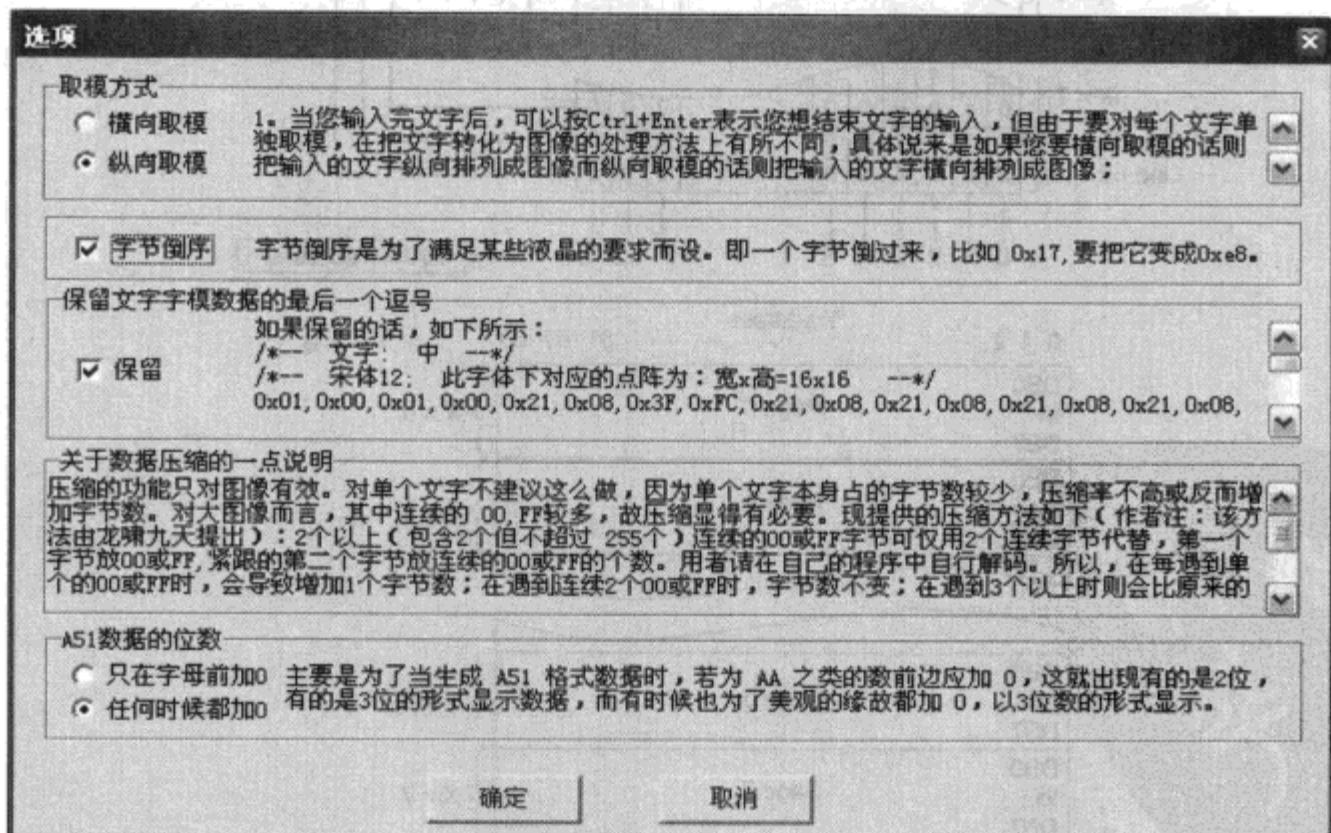


图 4-26 LGM1284 液晶汉字取模方式设置

表 4-14 LGM12864(KS0108) 液晶显示控制命令表

命 令	命令代码										功 能
	R/W	DI	DB7 ~ DB0								
显示开/关	0	0	0	0	1	1	1	1	1	0/1	控制显示开:1/关:0
设置列(Y 地址)	0	0	0	1	Y 地址(0~63)						设置 Y 地址
设置页(X 地址)	0	0	1	0	1	1	1	页(0~7)			设置 X 地址
显示起始行	0	0	1	1	0	显示起始行(0~63)					设置显示开关,光标开关,闪烁开关
读状态	1	0	B	0	On/Off	Rst	0	0	0	0	DB7(1:忙,0:就绪) DB5(显示开关 1:关,0:开) DB4(1:复位 0:正常)
写显示数据	0	1	待写入数据字节								写入后 Y 地址自动递增
读显示数据	1	0	读取数据字节								从显示数据 RAM 读数据

2. 实训要求

- ① 将本例汉字点阵重新保存到 Flash 程序存储器中,然后读取点阵并显示汉字。
- ② 将一幅面小于等于 128×64 点阵的位图显示在 LGM12864 液晶屏上。
- ③ 为本例液晶设计 Pixel、Line、Circle 与 Rectangle 函数。

3. 源程序代码

```

001 //----- LCD12864.C -----
002 // 名称: LGM12864LCD 显示驱动程序(不带字库)
003 //-
004 #include <avr/io.h>
005 #include <util/delay.h>
006 #include <string.h>
007 #define INT8U unsigned char
008 #define INT16U unsigned int
009
010 //LCD 起始行/页/列指令定义
011 #define LCD_START_ROW 0xC0      //起始行
012 #define LCD_PAGE        0xB8      //页指令
013 #define LCD_COL         0x40      //列指令
014
015 //液晶控制引脚
016 #define RW  PD0                //读/写
017 #define DI  PD1                //数据/指令
018 #define E   PD2                //使能
019 #define CS1 PD3                //右半屏选择
020 #define CS2 PD4                //右半屏选择

```

```

021 #define RST PD5           //复位
022
023 //液晶端口
024 #define LCD_PORT        PORTC    //液晶 DB0~DB7
025 #define LCD_DDR         DDRC     //设置数据方向
026 #define LCD_PIN          PINC     //读状态数据
027 #define LCD_CTRL         PORTD    //液晶控制端口
028
029 //液晶引脚操作定义
030 #define RW_1()  LCD_CTRL |= _BV(RW)
031 #define RW_0()  LCD_CTRL &= ~_BV(RW)
032 #define DI_1()  LCD_CTRL |= _BV(DI)
033 #define DI_0()  LCD_CTRL &= ~_BV(DI)
034 #define E_1()   LCD_CTRL |= _BV(E)
035 #define E_0()   LCD_CTRL &= ~_BV(E)
036 #define CS1_1() LCD_CTRL |= _BV(CS1)
037 #define CS1_0() LCD_CTRL &= ~_BV(CS1)
038 #define CS2_1() LCD_CTRL |= _BV(CS2)
039 #define CS2_0() LCD_CTRL &= ~_BV(CS2)
040 #define RST_1() LCD_CTRL |= _BV(RST)
041 #define RST_0() LCD_CTRL &= ~_BV(RST)
042
043 //是否反相显示(白底黑字/黑底白字)
044 INT8U Reverse_Display = 0;
045 //-----
046 // 等待液晶就绪
047 //-----
048 void Wait_LCD_Ready()
049 {
050     Check_Busy:
051     LCD_DDR = 0x00;           //设置数据方向为输入
052     LCD_PORT = 0xFF;         //内部上拉
053     RW_1(); asm("nop"); DI_0(); //读状态寄存器
054     E_1(); asm("nop"); E_0();
055     if (LCD_PIN & 0x80) goto Check_Busy;
056 }
057
058 //-----
059 // 向 LCD 发送命令
060 //-----
061 void LCD_Write_Command(INT8U cmd)
062 {
063     Wait_LCD_Ready();       //等待 LCD 就绪

```

```

064     LCD_DDR = 0xFF;           //设置方向为输出
065     LCD_PORT = 0xFF;          //初始输出高电平
066     RW_0(); asm("nop"); DI_0(); //写命令寄存器
067     LCD_PORT = cmd;          //发送命令
068     E_1(); asm("nop"); E_0(); //写入
069 }
070
071 //-----
072 // 向 LCD 发送数据
073 //-----
074 void LCD_Write_Data(INT8U dat)
075 {
076     Wait_LCD_Ready();          //等待 LCD 就绪
077     LCD_DDR = 0xFF;            //设置方向为输出
078     LCD_PORT = 0xFF;           //初始输出高电平
079     RW_0(); asm("nop"); DI_1(); //写数据寄存器
080     //发送数据,根据 Reverse_Display 决定是否反相显示
081     if ( !Reverse_Display) LCD_PORT = dat; else LCD_PORT = ~dat;
082     E_1(); asm("nop"); E_0(); //写入
083 }
084
085 //-----
086 // 初始化 LCD
087 //-----
088 void LCD_Initialize()
089 {
090     LCD_Write_Command(0x3F); _delay_ms(15); //开显示(0x3E 为关显示)
091 }
092
093 //-----
094 //
095 // 通用显示函数
096 //
097 // 从第 P 页第 L 列开始显示 W 个字节数据,数据在 r 所指向的缓冲
098 // 每字节 8 位是垂直显示的,高位在下,低位在上
099 // 每个 8 * 128 的矩形区域为一页
100 // 整个 LCD 又由 64 * 64 的左半屏和 64 * 64 的右半屏构成
101 //-----
102 void Common_Show(INT8U P, INT8U L, INT8U W, INT8U * r)
103 {
104     INT8U i;
105     //显示在左半屏或左右半屏
106     if( L<64 )

```

```

107     {
108         CS1_1(); CS2_0();
109         LCD_Write_Command( LCD_PAGE + P );
110         LCD_Write_Command( LCD_COL + L );
111         //全部显示在左半屏
112         if( L + W < 64 )
113         {
114             for(i = 0; i < W; i++) LCD_Write_Data(r[i]);
115         }
116         //如果越界则跨越左右半屏显示
117         else
118         {
119             //左半屏显示
120             for(i = 0; i < 64 - L; i++) LCD_Write_Data(r[i]);
121             //右半屏显示
122             CS1_0(); CS2_1();
123             LCD_Write_Command( LCD_PAGE + P );
124             LCD_Write_Command( LCD_COL );
125             for(i = 64 - L; i < W; i++) LCD_Write_Data(r[i]);
126         }
127     }
128     //全部显示在右半屏
129     else
130     {
131         CS1_0(); CS2_1();
132         LCD_Write_Command( LCD_PAGE + P );
133         LCD_Write_Command( LCD_COL + L - 64 );
134         for( i = 0; i < W; i++) LCD_Write_Data(r[i]);
135     }
136 }
137
138 //-----
139 // 显示一个 8 * 16 点阵字符
140 //-----
141 void Display_A_Char_8X16(INT8U P, INT8U L, INT8U * M)
142 {
143     Common_Show( P, L, 8, M ); //显示上半部分 8 * 8
144     Common_Show( P + 1, L, 8, M + 8 ); //显示下半部分 8 * 8
145 }
146
147 //-----
148 // 显示一个 16 * 16 点阵汉字
149 //-----

```

```

150 void Display_A_WORD(INT8U P, INT8U L, INT8U * M)
151 {
152     Common_Show( P, L, 16, M );      //显示汉字上半部分 16 * 8
153     Common_Show( P + 1, L, 16, M + 16 ); //显示汉字下半部分 16 * 8
154 }
155
156 //-----
157 // 显示一串 16 * 16 点阵汉字
158 //-----
159 void Display_A_WORD_String(INT8U P, INT8U L, INT8U C, INT8U * M)
160 {
161     INT8U i;
162     for (i = 0; i < C; i++)
163     {
164         Display_A_WORD(P, L + i * 16, M + i * 32);
165     }
166 }

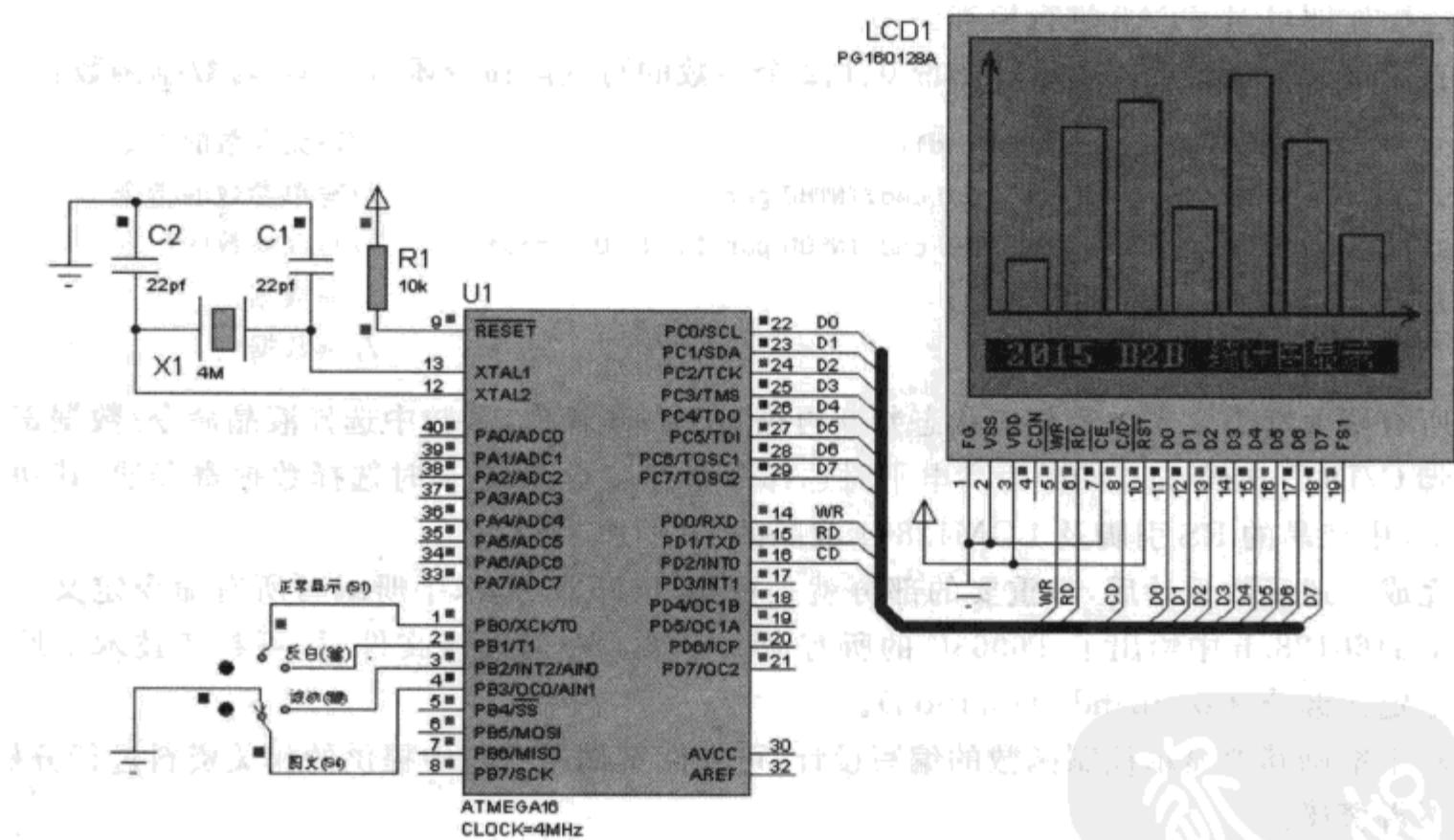
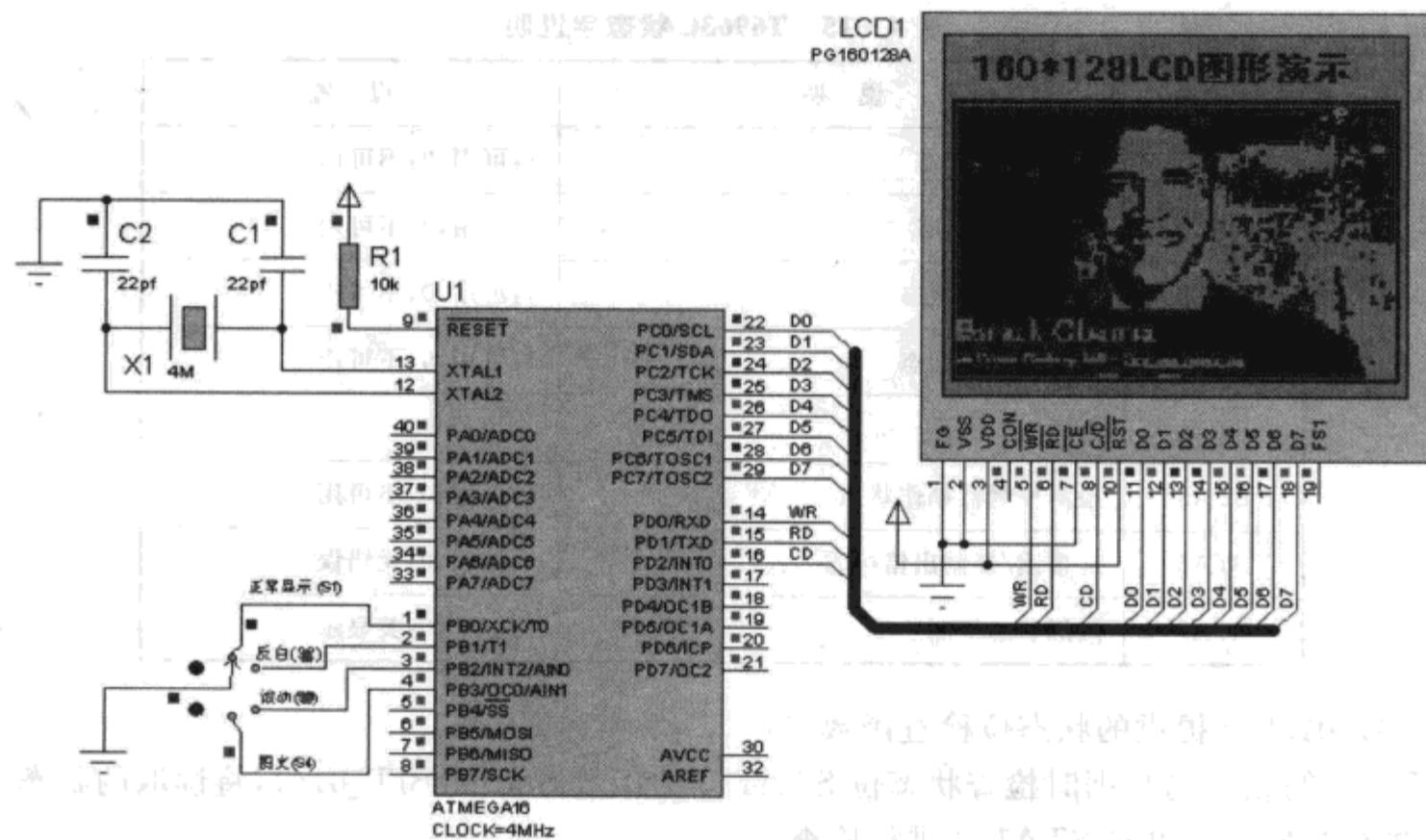
01 //----- main.c -----
02 // 名称：LGM12864 液晶显示程序
03 //-----
04 // 说明：本例运行时，液晶屏上将以正常与反白方式显示两行文字
05 //
06 //-----
07 #include <avr/io.h>
08 #include <util/delay.h>
09 #define INT8U unsigned char
10 #define INT16U unsigned int
11
12 //12864LCD 相关函数与变量
13 extern void LCD_Initialize();
14 extern void Display_A_WORD_String(INT8U P, INT8U L, INT8U C, INT8U * M);
15 extern INT8U Reverse_Display;
16 //待显示汉字点阵(在 Zimo 软件中取模时，设宋体小四号，纵向取模，字节倒序)
17 const INT8U WORD_Dot_Matrix[] =
18 {
19 //液
20 0x10,0x61,0x06,0xE0,0x18,0x84,0xE4,0x1C,0x84,0x65,0xBE,0x24,0xA4,0x64,0x04,0x00,
21 0x04,0x04,0xFF,0x00,0x01,0x00,0xFF,0x41,0x21,0x12,0x0C,0x1B,0x61,0xC0,0x40,0x00,
22 //晶
23 0x00,0x00,0x00,0x00,0x7E,0x2A,0x2A,0x2A,0x2A,0x2A,0x7E,0x00,0x00,0x00,0x00,0x00,
24 0x00,0x7F,0x25,0x25,0x25,0x25,0x7F,0x00,0x00,0x7F,0x25,0x25,0x25,0x25,0x7F,0x00,
25 //屏

```

```
26 0x00,0x00,0xFF,0x09,0x49,0x59,0xE9,0x49,0x49,0xE9,0x59,0x49,0x4F,0x00,0x00,
27 0x40,0x30,0x0F,0x82,0x42,0x32,0x0F,0x02,0x02,0x02,0xFF,0x02,0x02,0x02,0x02,0x00,
28 //测
29 0x08,0x31,0x86,0x60,0x00,0xFE,0x02,0xF2,0x02,0xFE,0x00,0xF8,0x00,0x00,0xFF,0x00,
30 0x04,0xFC,0x03,0x00,0x80,0x47,0x30,0x0F,0x10,0x67,0x00,0x07,0x40,0x80,0x7F,0x00,
31 //试
32 0x40,0x42,0xDC,0x08,0x00,0x90,0x90,0x90,0x90,0x90,0xFF,0x10,0x12,0x1C,0x10,0x00,
33 0x00,0x00,0x7F,0x20,0x10,0x20,0x20,0x1F,0x10,0x10,0x01,0x06,0x18,0x20,0x78,0x00,
34 //程
35 0x10,0x12,0xD2,0xFE,0x91,0x11,0x80,0xBF,0xA1,0xA1,0xA1,0xA1,0xBF,0x80,0x00,0x00,
36 0x04,0x03,0x00,0xFF,0x00,0x41,0x44,0x44,0x44,0x44,0x7F,0x44,0x44,0x44,0x44,0x40,0x00,
37 //序
38 0x00,0x00,0xFC,0x04,0x04,0x14,0x14,0x35,0x56,0x94,0x54,0x34,0x14,0x04,0x04,0x04,0x00,
39 0x80,0x60,0x1F,0x00,0x01,0x01,0x41,0x81,0x7F,0x01,0x01,0x01,0x03,0x01,0x00
40 };
41
42 //-----
43 // 主程序
44 //-----
45 int main()
46 {
47     //配置端口
48     DDRD = 0xFF;    PORTD = 0xFF;
49     //初始化 LCD
50     LCD_Initialize();
51     //从第 2 页开始(即 LCD 的第 16 行),左边距 8,显示 7 个汉字
52     Display_A_WORD_String(2, 8, 7, (INT8U *)WORD_Dot_Matrix),
53     //从第 5 页开始(即 LCD 的第 40 行),左边距 8,反相显示 7 个汉字(黑底白字)
54     Reverse_Display = 1;
55     Display_A_WORD_String(5, 8, 7, (INT8U *)WORD_Dot_Matrix),
56     while (1);
57 }
```

4.19 PG160128A 液晶图文演示

本例 PG160128A 液晶屏使用 T6963C 控制芯片。运行本例时可显示一幅内置图像，图像可滚动与反白显示，将开关拨到“图文”时还可以显示一组条形统计图，本例中显示的条形统计图不是通过获取图形文件的点阵数据来绘制的，而是根据代码中所提供的统计值动态绘制线条来实现的。本例电路及运行效果如图 4-27、图 4-28 所示。



1. 程序设计与调试

本例难点在于 PG160128.c 与 PG160128.h 的编写。T6963C 控制器在执行指令时可以带 0 个、1 个或 2 个参数，每条指令执行时都是先送入 1 个或 2 个参数（除无参以外），然后再发送命令。每次执行操作前需要先检查状态字，表 4-15 给出了 T6963C 的 8 位状态字。由于状态字的作用不一样，在执行不同指令时必须检查不同的状态位。



表 4-15 T6963C 状态字说明

位	说 明	设 置
STA0	指令读/写状态	1:可用,0:不可用
STA1	数据读/写状态	1:可用,0:不可用
STA2	数据自动读状态	1:可用,0:不可用
STA3	数据自动写状态	1:可用,0:不可用
STA4	未用	
STA5	检测控制器操作状态	1:可用,0:不可用
STA6	屏幕读/复制出错标志	1:错误,0:无错误
STA7	闪烁状态检测	1:正常,0:关显示

PG160128.c 提供的状态位检查函数有：

- ① 读/写指令与数据时检查状态位 STA1/0 的函数 Status_BIT_01(), 将读取的状态字和 0x03 进行与操作即可对 STA1/0 进行检查。
- ② 在数据自动读/写时检查状态位 STA3 的函数 Status_BIT_3(), 对读取的状态字和 0x08 进行与操作即可对 STA3 进行检查。

有了这两个函数后即可编写出带 0、1、2 个参数的写液晶命令函数及读/写数据函数：

```

INT8U LCD_Write_Command(INT8U cmd);                                //写无参数的指令
INT8U LCD_Write_Command_P1(INT8U cmd, INT8U para1);                //写单参数的指令
INT8U LCD_Write_Command_P2(INT8U cmd, INT8U para1, INT8U para2);    //写双参数的指令
INT8U LCD_Write_Data(INT8U dat);                                     //写数据
INT8U LCD_Read_Data();                                              //读数据

```

所有函数都需要先进行相应状态判断再进行下一步操作，函数中选择液晶命令/数据寄存的引脚 C/D(Command/Data)为高电平时选择命令寄存器，低电平时选择数据寄存器，其功能与 1602 中液晶的 RS 引脚及 LGM12864 液晶的 DI 引脚功能相同。

完成上述函数设计后，最重要的部分就是根据 T6963C 技术手册编写所有命令定义。头文件 PG160128.h 中给出了 T6963C 的所有命令定义。在阅读该文件时，可参考技术手册中的命令定义部分(Command Definition)。

对于本例其他显示控制函数的编写设计，可参阅案例压缩包中提供的相关资料进行分析，这里不再赘述。

最后说一下本例的汉字与图像取模问题：为取得 12×12 点阵汉字字模，注意先在 Zimo 软件中设置字体字号为宋体小五号，并设置取模方式为“横向取模”、“字节不倒序”，然后输入汉字“统计图表显示”并按下 Ctrl+Enter，最后单击取模按钮获取字模点阵。

在为图像取模时，首先导入案例文件夹下的 BMP 文件，然后同样设置取模方式为“横向取模”、“字节不倒序”，最后单击取模按钮即可得到图像点阵数据。本例的图像点阵数据存放于 PictureDots.h 文件。

本例图像的取模效果如图 4-29 所示。



图 4-29 用 Zimo 提取 PG160128.bmp 点阵数据

2. 实训要求

- ① 重新选择 3 幅不同图像,用 Zimo 软件分别取得图像点阵数据,修改本例程序,通过按钮切换 3 幅不同图像的显示。
- ② 编写程序在本例液晶屏上显示正弦曲线。

3. 源程序代码

```

001 //-----PG160128.c-----
002 // 名称: PG12864LCD 显示驱动程序(T6963C) (不带字库)
003 //-
004 #include <avr/io.h>
005 #include <avr/pgmspace.h>
006 #include <util/delay.h>
007 #include <string.h>
008 #include <math.h>
009 #include <string.h>
010 #include "PG160128.h"
011 #define INT8U unsigned char
012 #define INT16U unsigned int
013
014 //-
015 // 变更 LCD 与 MCU 的连接时,
016 // 只需要修改以下数据端口、控制端口及控制引脚定义
017 //-

```

```

018 //LCD 数据端口
019 #define LCD_DATA_PORT PORTC
020 #define LCD_DATA_PIN PINC
021 #define LCD_DATA_DDR DDRC
022 //LCD 控制端口
023 #define LCD_CTRL_PORT PORTD
024 //LCD 控制引脚定义(读,写,命令/数据寄存选择)
025 #define WR PD0
026 #define RD PD1
027 #define CD PD2
028 //-----
029
030 //LCD 控制引脚相关操作
031 #define WR_1() LCD_CTRL_PORT |= _BV(WR)
032 #define WR_0() LCD_CTRL_PORT &= ~_BV(WR)
033 #define RD_1() LCD_CTRL_PORT |= _BV(RD)
034 #define RD_0() LCD_CTRL_PORT &= ~_BV(RD)
035 #define CD_1() LCD_CTRL_PORT |= _BV(CD)
036 #define CD_0() LCD_CTRL_PORT &= ~_BV(CD)
037
038 //ASCII 字模宽度及高度定义
039 #define ASC_CHR_WIDTH 8
040 #define ASC_CHR_HEIGHT 12
041 #define HZ_CHR_HEIGHT 12
042 #define HZ_CHR_WIDTH 12
043 //液晶宽度与高度定义
044 const INT8U LCD_WIDTH = 20; //宽 160 像素(160/8 = 20 个字节)
045 const INT8U LCD_HEIGHT = 128; //高 128 像素
046
047 //下面的英文,数字,标点符号等字符点阵存放于程序 Flash 空间中
048 //使用时要用 pgm_read_byte(INT8U *) 函数读取,该函数在 avr/pgmspace.h 中申明
049 //本例使用的图像点阵也存放于 Flash 中
050 prog_uchar ASC_MSK[96 * 12] = {
051 0x00,0x00,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,0xff,0xff, //<0x20 时
052 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, // ''
053 0x00,0x30,0x78,0x78,0x78,0x30,0x30,0x00,0x30,0x30,0x00,0x00, // '!'
054 0x00,0x66,0x66,0x66,0x24,0x00,0x00,0x00,0x00,0x00,0x00,0x00, // '"'
055 0x00,0x6c,0x6c,0xfe,0x6c,0x6c,0xfe,0x6c,0x6c,0x00,0x00, // '#'
056 0x30,0x30,0x7c,0xc0,0xc0,0x78,0x0c,0x0c,0xf8,0x30,0x30,0x00, // '$'
057 0x00,0x00,0x00,0xc4,0xcc,0x18,0x30,0x60,0xcc,0x8c,0x00,0x00, // '%'
058 0x00,0x70,0xd8,0xd8,0x70,0xfa,0xde,0xcc,0xdc,0x76,0x00,0x00, // '&'
059 0x00,0x30,0x30,0x30,0x60,0x00,0x00,0x00,0x00,0x00,0x00, // ''
060 0x00,0x0c,0x18,0x30,0x60,0x60,0x30,0x18,0x0c,0x00,0x00, // '(

```




```
104 0x00,0xfc,0xb4,0x30,0x30,0x30,0x30,0x30,0x78,0x00,0x00,// 'T'
105 0x00,0xcc,0xcc,0xcc,0xcc,0xcc,0xcc,0xcc,0x78,0x00,0x00,// 'U'
106 0x00,0xcc,0xcc,0xcc,0xcc,0xcc,0xcc,0x78,0x30,0x00,0x00,// 'V'
107 0x00,0xc6,0xc6,0xc6,0xc6,0xd6,0xd6,0x6c,0x6c,0x6c,0x00,0x00,// 'W'
108 0x00,0xcc,0xcc,0xcc,0x78,0x30,0x78,0xcc,0xcc,0x00,0x00,// 'X'
109 0x00,0xcc,0xcc,0xcc,0x78,0x30,0x30,0x30,0x78,0x00,0x00,// 'Y'
110 0x00,0xfe,0xce,0x98,0x18,0x30,0x60,0x62,0xc6,0xfe,0x00,0x00,// 'Z'
111 0x00,0x3c,0x30,0x30,0x30,0x30,0x30,0x30,0x3c,0x00,0x00,// '['
112 0x00,0x00,0x80,0xc0,0x60,0x30,0x18,0x0c,0x06,0x02,0x00,0x00,// '\'
113 0x00,0x3c,0x0c,0x0c,0x0c,0x0c,0x0c,0x0c,0x3c,0x00,0x00,// ']'
114 0x10,0x38,0x6c,0xc6,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,// '^'
115 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0x00,// '_'
116 0x30,0x30,0x18,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,// '<'
117 0x00,0x00,0x00,0x00,0x78,0x0c,0x7c,0xcc,0xcc,0x76,0x00,0x00,// 'a'
118 0x00,0xe0,0x60,0x60,0x7c,0x66,0x66,0x66,0xdc,0x00,0x00,// 'b'
119 0x00,0x00,0x00,0x00,0x78,0xcc,0xc0,0xc0,0xcc,0x78,0x00,0x00,// 'c'
120 0x00,0x1c,0x0c,0x0c,0x7c,0xcc,0xcc,0xcc,0x76,0x00,0x00,// 'd'
121 0x00,0x00,0x00,0x00,0x78,0xcc,0xfc,0xc0,0xcc,0x78,0x00,0x00,// 'e'
122 0x00,0x38,0x6c,0x60,0x60,0xf8,0x60,0x60,0x60,0xf0,0x00,0x00,// 'f'
123 0x00,0x00,0x00,0x00,0x76,0xcc,0xcc,0x7c,0x0c,0xcc,0x78,// 'g'
124 0x00,0xe0,0x60,0x60,0x6c,0x76,0x66,0x66,0x66,0xe6,0x00,0x00,// 'h'
125 0x00,0x18,0x18,0x00,0x78,0x18,0x18,0x18,0x18,0x7e,0x00,0x00,// 'i'
126 0x00,0x0c,0x0c,0x00,0x3c,0x0c,0x0c,0x0c,0x0c,0xcc,0xcc,0x78,// 'j'
127 0x00,0xe0,0x60,0x60,0x6c,0x78,0x6c,0x66,0xe6,0x00,0x00,// 'k'
128 0x00,0x78,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x7e,0x00,0x00,// 'l'
129 0x00,0x00,0x00,0x00,0xfc,0xd6,0xd6,0xd6,0xc6,0x00,0x00,// 'm'
130 0x00,0x00,0x00,0x00,0xf8,0xcc,0xcc,0xcc,0x00,0x00,0x00,// 'n'
131 0x00,0x00,0x00,0x00,0x78,0xcc,0xcc,0xcc,0x78,0x00,0x00,// 'o'
132 0x00,0x00,0x00,0x00,0xdc,0x66,0x66,0x66,0x7c,0x60,0xf0,// 'p'
133 0x00,0x00,0x00,0x00,0x76,0xcc,0xcc,0xcc,0x7c,0x0c,0x1e,// 'q'
134 0x00,0x00,0x00,0x00,0xec,0x6e,0x76,0x60,0x60,0xf0,0x00,0x00,// 'r'
135 0x00,0x00,0x00,0x00,0x78,0xcc,0x60,0x18,0xcc,0x78,0x00,0x00,// 's'
136 0x00,0x00,0x20,0x60,0xfc,0x60,0x60,0x60,0x6c,0x38,0x00,0x00,// 't'
137 0x00,0x00,0x00,0x00,0xcc,0xcc,0xcc,0xcc,0x76,0x00,0x00,// 'u'
138 0x00,0x00,0x00,0x00,0xcc,0xcc,0xcc,0x78,0x30,0x00,0x00,// 'v'
139 0x00,0x00,0x00,0x00,0xc6,0xc6,0xd6,0xd6,0x6c,0x6c,0x00,0x00,// 'w'
140 0x00,0x00,0x00,0x00,0xc6,0x6c,0x38,0x38,0x6c,0xc6,0x00,0x00,// 'x'
141 0x00,0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x3c,0x0c,0x18,0xf0,// 'y'
142 0x00,0x00,0x00,0x00,0xfc,0x8c,0x18,0x60,0xc4,0xfc,0x00,0x00,// 'z'
143 0x00,0x1c,0x30,0x30,0x60,0xc0,0x60,0x30,0x1c,0x00,0x00,// '{'
144 0x00,0x18,0x18,0x18,0x18,0x00,0x18,0x18,0x18,0x00,0x00,// '|'
145 0x00,0xe0,0x30,0x30,0x18,0x0c,0x18,0x30,0xe0,0x00,0x00,// '}'
146 0x00,0x73,0xda,0xce,0x00,0x00,0x00,0x00,0x00,0x00,0x00,// '~'
```

```

147  };
148
149 struct typFNT_GB16 //汉字字模数据结构
150 {
151     char Index[2]; //汉字内码,2字节
152     INT8U Msk[24]; //汉字点阵
153 };
154
155 //取本例汉字 12 * 12 点阵库时,先在 Zimo 软件中设置字体字号为宋体小五号,
156 //取点阵前先设置横向取模,字节不倒序,然后输入汉字并按下 Ctrl + Enter,
157 //最后按下取模按钮获取字模
158 const struct typFNT_GB16 GB_16[] = {
159     {"统"}, {0x21,0x00,0x27,0xE0,0x51,0x00,0xF2,0x00,0x24,0x40,0x47,0xE0,
160                 0xF2,0x80,0x02,0x80,0x32,0xA0,0xC4,0xA0,0x18,0xE0,0x00,0x00},
161     {"计"}, {0x41,0x00,0x21,0x00,0x01,0x00,0x01,0x00,0xCF,0xE0,0x41,0x00,
162                 0x41,0x00,0x41,0x00,0x51,0x00,0x61,0x00,0x41,0x00,0x00,0x00},
163     {"图"}, {0x7F,0xE0,0x48,0x20,0x5F,0x20,0x6A,0x20,0x44,0x20,0x4A,0x20,
164                 0x75,0xA0,0x42,0x20,0x4C,0x20,0x42,0x20,0x7F,0xE0,0x00,0x00},
165     {"显"}, {0x3F,0x80,0x20,0x80,0x3F,0x80,0x20,0x80,0x3F,0x80,0x00,0x00,
166                 0x4A,0x40,0x2A,0x40,0x2A,0x80,0x0B,0x00,0xFF,0xE0,0x00,0x00},
167     {"示"}, {0x00,0x80,0x7F,0xC0,0x00,0x00,0x00,0x00,0xFF,0xE0,0x04,0x00,
168                 0x14,0x80,0x24,0x40,0x44,0x20,0x84,0x20,0x1C,0x00,0x00,0x00}
169 };
170
171 INT8U gCurRow,gCurCol; //当前行、列
172 //-----
173 // LCD 控制相关函数
174 //-----
175 INT8U Status_BIT_01(); //状态位 STA1,STA0 判断(读/写指令和读/写数据)
176 INT8U Status_BIT_3(); //状态位 ST3 判断(数据自动写状态)
177 INT8U LCD_Write_Command(INT8U cmd); //写无参数指令
178 INT8U LCD_Write_Command_P1(INT8U cmd,INT8U para1); //写单参数指令
179 INT8U LCD_Write_Command_P2(INT8U cmd,INT8U para1,INT8U para2); //写双参数指令
180 INT8U LCD_Write_Data(INT8U dat); //写数据
181 INT8U LCD_Read_Data(); //读数据
182
183 void Clear_Screen(); //清屏
184 char LCD_Initialise(); //LCD 初始化
185 void Set_LCD_POS(INT8U row, INT8U col); //设置当前地址
186 void OutToLCD(INT8U Dat, INT8U x, INT8U y); //输出到液晶
187 void Line(INT8U x1, INT8U y1, INT8U x2, INT8U y2, INT8U Mode); //绘制直线
188 void Pixel(INT8U x, INT8U y, INT8U Mode); //绘点
189

```



```
190 //-----  
191 // 读状态  
192 //-----  
193 INT8U Read_LCD_Status()  
194 {  
195     INT8U st;  
196     LCD_DATA_DDR = 0x00;  
197     CD_1(); RD_0(); _delay_us(1);  
198     st = LCD_DATA_PIN; RD_1(); LCD_DATA_DDR = 0xFF;  
199     return st;  
200 }  
201  
202 //-----  
203 // 读数据  
204 //-----  
205 INT8U Read_LCD_Data()  
206 {  
207     INT8U dat;  
208     LCD_DATA_DDR = 0x00;  
209     CD_0(); RD_0(); _delay_us(1);  
210     dat = LCD_DATA_PIN; RD_1(); LCD_DATA_DDR = 0xFF;  
211     return dat;  
212 }  
213  
214 //-----  
215 // 写数据  
216 //-----  
217 void Write_Data(INT8U dat)  
218 {  
219     LCD_DATA_DDR = 0xFF;  
220     CD_0(); LCD_DATA_PORT = dat; WR_0(); _delay_us(2); WR_1();  
221 }  
222  
223 //-----  
224 // 写命令  
225 //-----  
226 void Write_Command(INT8U cmd)  
227 {  
228     LCD_DATA_DDR = 0xFF;  
229     CD_1(); LCD_DATA_PORT = cmd; WR_0(); _delay_us(2); WR_1();  
230 }  
231  
232 //-----
```

```

233 // 状态位 STA1,STA0 判断(读/写指令和读/写数据)
234 //-----
235 INT8U Status_BIT_01()
236 {
237     INT8U i;
238     for(i = 10; i > 0; i--)
239     {
240         if((Read_LCD_Status() & 0x03) == 0x03) break;
241     }
242     return i; //错误时返回 0
243 }
244
245 //-----
246 // 状态位 ST3 判断(数据自动写状态)
247 //-----
248 INT8U Status_BIT_3()
249 {
250     INT8U i;
251     for(i = 10; i > 0; i--)
252     {
253         if((Read_LCD_Status() & 0x08) == 0x08) break;
254     }
255     return i; //错误时返回 0
256 }
257
258 //-----
259 // 写双参数的指令
260 //-----
261 INT8U LCD_Write_Command_P2(INT8U cmd, INT8U para1, INT8U para2)
262 {
263     if(Status_BIT_01() == 0) return 1;
264     Write_Data(para1);
265     if(Status_BIT_01() == 0) return 2;
266     Write_Data(para2);
267     if(Status_BIT_01() == 0) return 3;
268     Write_Command(cmd);
269     return 0; //成功时返回 0
270 }
271
272 //-----
273 // 写单参数的指令
274 //-----
275 INT8U LCD_Write_Command_P1(INT8U cmd, INT8U para1)

```



```
276  {
277      if(Status_BIT_01() == 0) return 1;
278      Write_Data(para1);
279      if(Status_BIT_01() == 0) return 2;
280      Write_Command(cmd);
281      return 0;                                //成功时返回 0
282  }
283
284 //-----
285 // 写无参数的指令
286 //-----
287 INT8U LCD_Write_Command(INT8U cmd)
288 {
289     if(Status_BIT_01() == 0) return 1;
290     Write_Command(cmd);
291     return 0;                                //成功时返回 0
292 }
293
294 //-----
295 // 写数据
296 //-----
297 INT8U LCD_Write_Data(INT8U dat)
298 {
299     if(Status_BIT_3() == 0) return 1;
300     Write_Data(dat);
301     return 0;                                //成功时返回 0
302 }
303
304 //-----
305 // 读数据
306 //-----
307 INT8U LCD_Read_Data()
308 {
309     if(Status_BIT_01() == 0) return 1;
310     return Read_LCD_Data();
311 }
312
313 //-----
314 // 设置当前地址
315 //-----
316 void Set_LCD_POS(INT8U row, INT8U col)
317 {
318     INT16U Pos;
```



```

319     Pos = row * LCD_WIDTH + col;
320     LCD_Write_Command_P2(LC_ADD_POS,Pos % 256, Pos / 256);
321     gCurRow = row;
322     gCurCol = col;
323 }
324
325 //-----
326 // 清屏
327 //-----
328 void Clear_Screen()
329 {
330     INT16U i;
331     LCD_Write_Command_P2(LC_ADD_POS,0x00,0x00); //置地址指针
332     LCD_Write_Command(LC_AUT_WR);           //自动写
333     for(i = 0;i<0x2000; i++)
334     {
335         Status_BIT_3();
336         LCD_Write_Data(0x00);           //写数据
337     }
338     LCD_Write_Command(LC_AUT_OVR);           //自动写结束
339     LCD_Write_Command_P2(LC_ADD_POS,0x00,0x00); //重置地址指针
340     gCurRow = 0;
341     gCurCol = 0;
342 }
343
344 //-----
345 // LCD 初始化
346 //-----
347 char LCD_Initialise()
348 {
349     LCD_Write_Command_P2(LC_TXT_STP,0x00,0x00);      //文本显示区首地址
350     LCD_Write_Command_P2(LC_TXT_WID,LCD_WIDTH,0x00); //文本显示区宽度
351     LCD_Write_Command_P2(LC_GRH_STP,0x00,0x00);      //图形显示区首地址
352     LCD_Write_Command_P2(LC_GRH_WID,LCD_WIDTH,0x00); //图形显示区宽度
353     LCD_Write_Command(LC_CUR_SHP | 0x01);           //光标形状
354     LCD_Write_Command(LC_MOD_OR);                   //显示方式设置
355     LCD_Write_Command(LC_DIS_SW | 0x08);
356     return 0;
357 }
358
359 //-----
360 // ASCII 及汉字显示(wb 表示是否反白显示)
361 //-----

```

```

362 void Display_Str_at_xy(INT8U x, INT8U y, char * Buffer, INT8U wb)
363 {
364     char c1,c2,cData;
365     INT8U i = 0, j, k, uLen = strlen(Buffer);
366     while(i < uLen)
367     {
368         c1 = Buffer[i]; c2 = Buffer[i + 1];
369         Set_LCD_POS(y, x / 8);
370         //ASCII 字符显示
371         if((c1 & 0x80) == 0x00)
372         {
373             if(c1 < 0x20)
374             {
375                 switch(c1)
376                 {
377                     case CR;
378                     case LF: i++; x = 0; //回车或换行
379                     if(y < 112) y += HZ_CHR_HEIGHT;
380                     continue;
381                     case BS: i++; //退格
382                     if(y > ASC_CHR_WIDTH) y -= ASC_CHR_WIDTH;
383                     cData = 0x00;
384                     break;
385                 }
386             }
387             //从 Flash 程序 ROM 中读取字符点阵并显示
388             for(j = 0; j < ASC_CHR_HEIGHT; j++)
389             {
390                 if(c1 >= 0x1F)
391                 {
392                     cData = pgm_read_byte(ASC_MSK + (c1 - 0x1F) * ASC_CHR_HEIGHT + j);
393                     if (wb) cData = ~cData;
394                     Set_LCD_POS(y + j, x / 8);
395                     if( (x % 8) == 0)
396                     {
397                         LCD_Write_Command(LC_AUT_WR);
398                         LCD_Write_Data(cData);
399                         LCD_Write_Command(LC_AUT_OVR);
400                     }
401                     else OutToLCD(cData, x, y + j);
402                 }
403                 Set_LCD_POS(y + j, x / 8);
404             }
}

```

```

405         if(c1 != BS) x += ASC_CHR_WIDTH;
406     }
407     //中文字符显示
408     else
409     {
410         //在字库中查找汉字
411         for(j = 0; j < sizeof(GB_16)/sizeof(GB_16[0]); j++)
412         {
413             if(c1 == GB_16[j].Index[0] && c2 == GB_16[j].Index[1])
414                 break;
415         }
416         //从中文点阵库中读取点阵并显示
417         for(k = 0; k < HZ_CHR_HEIGHT; k++)
418         {
419             Set_LCD_POS(y + k, x / 8);
420             if(j < sizeof(GB_16)/sizeof(GB_16[0]))
421             {
422                 c1 = GB_16[j].Msk[k * 2];
423                 c2 = GB_16[j].Msk[k * 2 + 1];
424             }
425             else c1 = c2 = 0;
426             if((x % 8) == 0)
427             {
428                 LCD_Write_Command(LC_AUT_WR);
429                 if(wb) c1 = ~c1;
430                 LCD_Write_Data(c1);
431                 LCD_Write_Command(LC_AUT_OVR);
432             }
433             else
434             {
435                 if(wb) c1 = ~c1;
436                 OutToLCD(c1, x, y + k);
437             }
438             if(((x + 2 + HZ_CHR_WIDTH / 2) % 8) == 0)
439             {
440                 LCD_Write_Command(LC_AUT_WR);
441                 if(wb) c2 = ~c2;
442                 LCD_Write_Data(c2);
443                 LCD_Write_Command(LC_AUT_OVR);
444             }
445             else
446             {

```

```

448             if (wb) c2 = ~c2;
449             OutToLCD(c2,x + 2 + HZ_CHR_WIDTH / 2,y + k);
450         }
451     }
452     x += HZ_CHR_WIDTH; i++;
453 }
454 i++;
455 }
456 }
457
458 //-----
459 // 输出起点x不是8的倍数时,原字节分成两部分输出到LCD
460 //-----
461 void OutToLCD(INT8U Dat, INT8U x, INT8U y)
462 {
463     INT8U dat1,dat2,a,b;
464     b = x % 8; a = 8 - b;
465     Set_LCD_POS(y,x / 8);
466     LCD_Write_Command(LC_AUT_RD);
467     dat1 = LCD_Read_Data();
468     dat2 = LCD_Read_Data();
469     //将读取的前后两字节分别与待显示字节的前后部分组合
470     dat1 = (dat1 & (0xFF<<a)) | (Dat>>b);
471     dat2 = (dat2 & (0xFF>>b)) | (Dat<<a);
472     LCD_Write_Command(LC_AUT_OVR);
473     Set_LCD_POS(y,x / 8);
474     //输出组合后的两字节
475     LCD_Write_Command(LC_AUT_WR);
476     LCD_Write_Data(dat1);
477     LCD_Write_Data(dat2);
478     LCD_Write_Command(LC_AUT_OVR);
479 }
480
481 //-----
482 // 绘点函数
483 // 参数:点的坐标,模式1/0分别为显示与清除点
484 //-----
485 void Pixel(INT8U x, INT8U y, INT8U Mode)
486 {
487     INT8U start_addr, dat;
488     start_addr = 7 - (x % 8);
489     dat = LC_BIT_OP | start_addr;           //生成位操作命令绘点数据
490     if (Mode) dat |= 0x08;

```

```

491     Set_LCD_POS(y, x / 8);
492     LCD_Write_Command(LC_BIT_OP | dat); //写数据
493 }
494
495 //-----
496 // 两数交换
497 //-----
498 void Exchange(INT8U * a, INT8U * b)
499 {
500     INT8U t;
501     t = * a; * a = * b; * b = t;
502 }
503
504 //-----
505 // 绘制直线函数
506 // 参数:起点与终点坐标,模式为显示(1)或清除(0),点阵不超过 255 * 255)
507 //-----
508 void Line(INT8U x1, INT8U y1, INT8U x2, INT8U y2, INT8U Mode)
509 {
510     INT8U x,y;           //绘点坐标
511     float k,b;          //直线斜率与偏移
512
513     if( fabs(y1 - y2) <= fabs( x1 - x2 ) )
514     {
515         k = (float)(y2 - y1) / (float)(x2 - x1) ;
516         b = y1 - k * x1;
517         if( x1 > x2 ) Exchange(&x1, &x2);
518         for(x = x1;x <= x2; x++)
519         {
520             y = (INT8U)(k * x + b);
521             Pixel(x, y, Mode);
522         }
523     }
524     else
525     {
526         k = (float)(x2 - x1) / (float)(y2 - y1) ;
527         b = x1 - k * y1;
528         if( y1 > y2 ) Exchange(&y1, &y2);
529         for(y = y1;y <= y2; y++)
530         {
531             x = (INT8U)(k * y + b);
532             Pixel( x , y,Mode );
533         }

```

```

534      }
535  }
536
537 //-----
538 // 绘制图像(图像数据来自于 Flash 程序 ROM 空间)
539 //-----
540 void Draw_Image(prog_uchar * G_Buffer, INT8U Start_Row, INT8U Start_Col)
541 {
542     INT16U i,j,W,H;
543     //图像行数控制(G_Buffer 的前两个字节分别为图像宽度与高度)
544     W = pgm_read_byte(G_Buffer + 1);
545     for(i = 0; i < W; i++)
546     {
547         Set_LCD_POS(Start_Row + i, Start_Col);
548         LCD_Write_Command(LC_AUT_WR);
549         //绘制图像每行像素
550         H = pgm_read_byte(G_Buffer);
551         for(j = 0; j < H / 8; j++)
552             LCD_Write_Data(pgm_read_byte(G_Buffer + i * (H / 8) + j + 2));
553         LCD_Write_Command(LC_AUT_OVR);
554     }
555 }
```

01 //----- PG160128.h -----

02 // 名称：PG160128 显示驱动程序头文件

```

03 //-----
04 #include <stdio.h>
05 #include <math.h>
06 #include <string.h>
07 #define STX 0x02
08 #define ETX 0x03
09 #define EOT 0x04
10 #define ENQ 0x05
11 #define BS 0x08
12 #define CR 0x0D
13 #define LF 0x0A
14 #define DLE 0x10
15 #define ETB 0x17
16 #define SPACE 0x20
17 #define COMMA 0x2C
18
19 #define TRUE 1
20 #define FALSE 0
```

```

21
22 #define HIGH 1
23 #define LOW 0
24
25 //T6963C 命令定义
26 #define LC_CUR_POS 0x21 //光标位置设置
27 #define LC_CGR_POS 0x22 //CGRAM 偏置地址设置
28 #define LC_ADD_POS 0x24 //地址指针位置
29 #define LC_TXT_STP 0x40 //文本区首址
30 #define LC_TXT_WID 0x41 //文本区宽度
31 #define LC_GRH_STP 0x42 //图形区首址
32 #define LC_GRH_WID 0x43 //图形区宽度
33 #define LC_MOD_OR 0x80 //显示方式:逻辑或
34 #define LC_MOD_XOR 0x81 //显示方式:逻辑异或
35 #define LC_MOD_AND 0x82 //显示方式:逻辑与
36 #define LC_MOD_TCH 0x83 //显示方式:文本特征
37 #define LC_DIS_SW 0x90 //显示开关:
38 //D0 = 1/0:光标闪烁启用/禁用
39 //D1 = 1/0:光标显示启用/禁用
40 //D2 = 1/0:文本显示启用/禁用
41 //D3 = 1/0:图形显示启用/禁用
42 #define LC_CUR_SHP 0xA0 //光标形状选择:0xA0~0xA7 表示光标占的行数
43 #define LC_AUT_WR 0xB0 //自动写设置
44 #define LC_AUT_RD 0xB1 //自动读设置
45 #define LC_AUT_OVR 0xB2 //自动读/写结束
46 #define LC_INC_WR 0xC0 //数据写,地址加 1
47 #define LC_INC_RD 0xC1 //数据读,地址加 1
48 #define LC_DEC_WR 0xC2 //数据写,地址减 1
49 #define LC_DEC_RD 0xC3 //数据读,地址减 1
50 #define LC_NOC_WR 0xC4 //数据写,地址不变
51 #define LC_NOC_RD 0xC5 //数据读,地址不变
52 #define LC_SCN_RD 0xE0 //读屏屏
53 #define LC_SCN_CP 0xE8 //屏幕拷贝
54 #define LC_BIT_OP 0xF0 //位操作:B0~B2 对应 D0~D7 位;B3:1 置位/0:清除

```

```

//----- PictureDots.h -----
001 //显示在 LCD 上的图像点阵,数组数据存放于程序 Flash 空间中
002 prog_uchar ImageX[] = { //用 Zimo 取本例图像点阵时,注意"横向取模,字节不倒序"
003 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
004 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
.....因篇幅限制,这里省略了待显示图像的大部分点阵数据。
161 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
162 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00

```

```

163  };

001 //----- main.c -----
002 // 名称：PG160128 液晶图形滚动演示
003 //-----
004 // 说明：本例可显示一幅图像，可控制图像滚动，反白，合上“图文”开关时，
005 // 还可以显示一幅条形统计图
006 //
007 //-----
008 #include <avr/io.h>
009 #include <avr/pgmspace.h>
010 #include <util/delay.h>
011 #include <stdio.h>
012 #include "PG160128.h"
013 #include "PictureDots.h"
014 #define INT8U unsigned char
015 #define INT16U unsigned int
016

017 extern void Clear_Screen();           //清屏
018 extern INT8U LCD_Initialise();        //LCD 初始化
019 extern INT8U LCD_Write_Command(INT8U cmd); //写无参数的命令
020                                     //写双参数命令
021 extern INT8U LCD_Write_Command_P2(INT8U cmd, INT8U para1, INT8U para2);
022 extern INT8U LCD_Write_Data(INT8U dat); //写数据
023 extern void Set_LCD_POS(INT8U row, INT8U col); //设置当前地址
024                                     //绘制线条
025 extern void Line(INT8U x1, INT8U y1, INT8U x2, INT8U y2, INT8U Mode); //显示字符串
026                                     //显示字符串
027 extern INT8U Display_Str_at_xy(INT8U x, INT8U y, char * fmt);
028 extern INT8U LCD_WIDTH;
029 extern INT8U LCD_HEIGHT;
030

031 //开关定义
032 #define S1_ON() (PINB & _BV(PB0)) == 0x00 //正常显示
033 #define S2_ON() (PINB & _BV(PB1)) == 0x00 //反白
034 #define S3_ON() (PINB & _BV(PB2)) == 0x00 //滚动
035 #define S4_ON() (PINB & _BV(PB3)) == 0x00 //图文
036

037 //当前操作序号
038 INT8U Current_Operation = 0;
039 //待显示的统计数据
040 INT8U Statistics_Data[] = {20, 70, 80, 40, 90, 65, 30};
041 //-----

```

```

042 // 绘制条形图
043 //-----
044 void Draw_Bar_Graph(INT8U d[])
045 {
046     INT8U i,h;
047     Line(4,2,4,100,1);           //纵轴
048     Line(4,100,158,100,1);      //横轴
049     Line(4,2,1,10,1);          //纵轴箭头
050     Line(4,2,7,10,1);
051     Line(158,100,152,97,1);    //横轴箭头
052     Line(158,100,152,103,1);
053     for (i = 0; i < 7; i++)
054     {
055         h = 100 - d[i];
056         Line(10 + i * 20, h, 10 + i * 20, 100, 1);
057         Line(10 + i * 20, h, 10 + i * 20 + 15, h, 1);
058         Line(10 + i * 20 + 15, h, 10 + i * 20 + 15, 100, 1);
059     }
060 }
061
062 //-----
063 // 主程序
064 //-----
065 int main()
066 {
067     INT8U i,j,m,c = 0;  INT16U k;
068     //配置端口
069     DDRB = 0x00; PORTB = 0xFF;
070     DDRC = 0xFF; PORTC = 0xFF;
071     DDRD = 0xFF; PORTD = 0xFF;
072     //初始化 LCD
073     LCD_Initialise();
074     //从 LCD 左上角开始清屏
075     Set_LCD_POS(0,0);
076     Clear_Screen();
077     while(1)
078     {
079         if (S1_ON()) Current_Operation = 1; //正常
080         else if (S2_ON()) Current_Operation = 2; //反白
081         else if (S3_ON()) Current_Operation = 3; //滚动
082         else if (S4_ON()) Current_Operation = 4; //图文
083         //如果操作类型未改变则仅执行延时
084         if (c == Current_Operation) goto delayx;

```



```
085     c = Current_Operation;
086     switch (Current_Operation)
087     {
088         case 1://正常或反白显示
089         case 2:
090             LCD_Write_Command_P2( LC_GRH_STP,0x00,0x00);
091             //行循环,LCD_HEIGHT = 128
092             for(i = 0;i<LCD_HEIGHT; i++)
093             {
094                 //设置从每行起点开始显示
095                 Set_LCD_POS(i,0);
096                 //写数据
097                 LCD_Write_Command(LC_AUT_WR);
098                 //显示每行中的 160 个像素,LCD_WIDTH = 160/8
099                 for( j = 0; j<LCD_WIDTH; j++)
100                 {
101                     m = pgm_read_byte(ImageX + i * LCD_WIDTH + j);
102                     //如果合上 S2 则反白显示
103                     if (S2_ON()) m = ~m;
104                     //向 LCD 输出图像像素,每次输出 1 字节,8 个像素
105                     LCD_Write_Data(m);
106                 }
107                 LCD_Write_Command(LC_AUT_OVR);
108             }
109             break;
110         case 3://滚动显示
111             //每次向下移动一行 GFXHOME 地址(20 字节),使前面的图像向上滚动出屏幕
112             k = 0;
113             //宽度单位为字节(相当于 8 像素),高度单位为像素
114             while ( k != LCD_WIDTH * LCD_HEIGHT)
115             {
116                 //设置图形区首地址
117                 LCD_Write_Command_P2( LC_GRH_STP, k % 256, k / 256) ;
118                 _delay_ms(20);//延时
119                 k += LCD_WIDTH;
120             }
121             break;
122         case 4://图文显示
123             LCD_Write_Command_P2( LC_GRH_STP,0x00,0x00);
124             Set_LCD_POS(0,0);
125             Clear_Screen();
126             Draw_Bar_Graph(Statistics_Data); //根据统计数据数组显示条形图
127             Display_Str_at_xy(3,110," 2015 B2B 统计图显示 ");//显示统计图标识
```

```

128         break;
129     }
130     delayx: _delay_ms(300);
131 }
132 }

```

4.20 TG126410 液晶串行模式显示

本例 TG126410 液晶使用的控制芯片为 SED1565，程序将其配置成工作于串行模式，开关和按键可控制 LCD 显示不同画面，所显示的画面有的完全由程序控制显示，有的则根据画面图形点阵数据显示。本例电路及部分运行效果如图 4-30 所示。

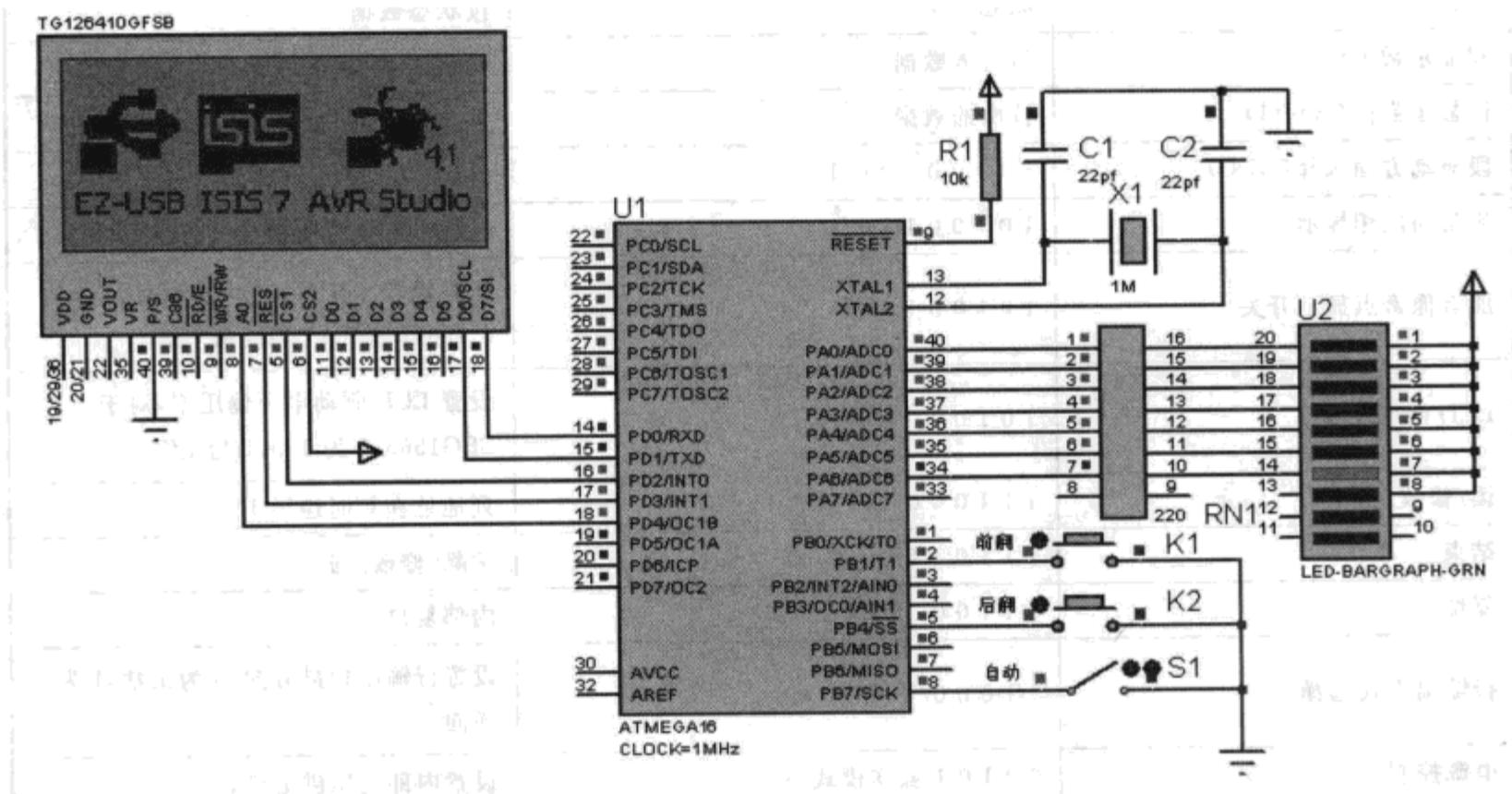


图 4-30 TG126410 液晶串行模式显示

1. 程序设计与调试

本例液晶使用 SED1565 控制芯片，下面对其引脚作简要说明：

① P/S 引脚用于选择并行(Parallel)与串行(Serial)方式，当 P/S 引脚接高电平时选择并行方式，反之则选择串行方式，本例将该引脚接低电平。

② 本例液晶工作于并行方式时，D0~D7 为 8 位双向数据总线；在工作于串行方式时，D7 为串行数据输入线(SI)，D6 为串行时钟线。其他引脚呈高阻状态。

③ A0 引脚用于决定所发送是待显示数据还是控制命令。当 A0=1 时所发送的是数据，当 A0=0 时为命令。

④ CS1 与 CS2 为片选信号输入线，当 CS1=0，CS2=1 时数据与命令 I/O 端口有效。

需要详细了解其他引脚功能时可参阅本书网上资料案例压缩包中“芯片资料”/TG12640—SED1565.pdf 文件。

为编写本例液晶显示程序，还需要给出该液晶的控制命令，表 4-16 是 TG126410(SED1565)

液晶命令集。

表 4-16 TG126410(SED1565) 液晶命令集

命 令	命令代码(在读/写数据 时 A0=1, 在写命令时 A0=0)	功 能
	D7~D0	
显示开关	1 0 1 0 1 1 1 0/1	液晶显示开关(0:关,1:开)
设置显示起始行	0 1 起始行(0x00~0x3F)	设置显示 RAM 的显示起始行
页地址	1 0 1 1 页地址(0~7)	设置页地址
列地址高位	0 0 0 1 列地址高 4 位	
列地址低位	0 0 0 0 列地址低 4 位	分别设置列地址的高 4 位与低 4 位
读状态	状态 0 0 0 0	读状态数据
写显示数据(A0=1)	待写入数据	将数据写入显示 RAM
读显示数据(A0=1)	读取的数据	从显示 RAM 中读数据
段驱动方向选择(ADC)	1 0 1 0 0 0 0 0/1	0 为正常,1 为逆向
正常与反相显示	1 0 1 0 0 1 1 0/1	0 为正常,1 为反相
所有像素点显示开关	1 0 1 0 0 1 0 0/1	所有像素点显示 0 为正常显示,1 为反黑
LCD 偏压设置	1 0 1 0 0 0 1 0/1	设置 LCD 驱动电压偏压率,对于 SEG1565,0 为 1/9,1 为 1/7
读/修改/写	1 1 1 0 0 0 0 0	列地址在写时递增 1
结束	1 1 1 0 1 1 1 0	清除/修改/写
复位	1 1 1 0 0 0 1 0	内部复位
行输出方式选择	1 1 0 0 0/1 * * *	设置行输出扫描方向,0 为正常,1 为 逆向
电源控制	0 0 1 0 1 操作模式	设置内部电压供电模式
V5 电压调节内部电阻率设置	0 0 1 0 0 电阻率	设置内部电阻率模式(Rb/Ra)
电量模式设置	1 0 0 0 0 0 0 1	
电量寄存器设置	* * 电 量 值	设置 V5 输出电压电量寄存器
静态指示器开关	1 0 1 0 1 1 0 0/1	0:关,1:开
静态指示寄存器设置	* * * * * * * 模式	设置刷新模式
无操作命令(NOP)	1 1 1 0 0 0 1 1	无操作
测试命令	1 1 1 1 * * * *	用于 IC 测试
测试模式复位	1 1 1 1 0 0 0 0	进入刷新序列

注:以上标有*的数位是无用的。

图 4-31 给出了 TG126410 液晶 DDRAM(Display DATA RAM, 显示数据 RAM)的页地址、行地址及列地址示意图。另外表 4-16 第 8 行给出了段驱动方向设置命令,通过该命令可颠倒列地址与段输出之间的关系,该命令的最低位设为 0 时,列地址 0x00~0x83 对应于段 0x00~0x83(SEG0~SEG131);反之,如果该位设为 1,则列地址与段输出的对应关系刚好相

反。显然,通过设置该位可“水平翻转”画面。

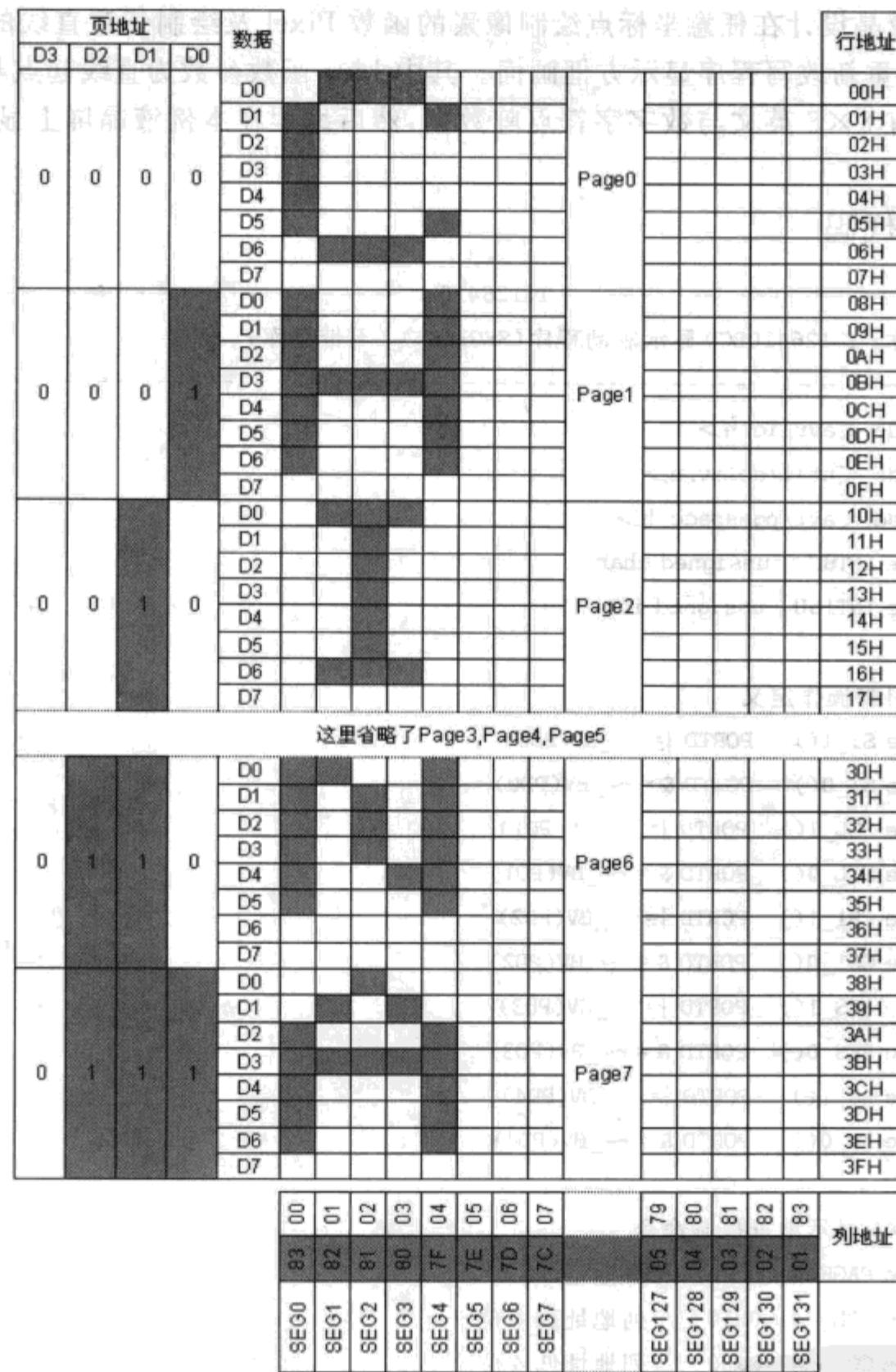


图 4-31 TG126410 液晶的页、行、列地址示意图

另外,如果已经将一幅画面的点阵数据写入 DDRAM,在使用行地址设置命令动态修改显示起始行时会导致画面滚动。例如:起始行设为 30 行时,屏幕最顶端开始显示的将是 DDRAM 中的第 30 行数据,向上滚出的部分将出现在屏幕下方。

在阅读本例全屏显示、全屏不显示、平铺显示 R、显示方框、显示画面等子程序时,可参照图 4-31 及本例液晶命令表进行对比分析。

2. 实训要求

- ① 通过本例液晶命令测试像素反相及画面左右水平翻转的显示效果。

- ② 设置本例液晶工作于并行模式,仍实现与本例类似的显示效果。
- ③ 为本例液晶设计在任意坐标点绘制像素的函数 Pixel 及绘制任意直线的函数 Line,利用所编写的函数重新改写程序显示方框画面。其中 Line 函数参数为直线起点与终点坐标。
- ④ 提取所有 8×8 英文与数字字符点阵数据,然后编程在本例液晶屏上显示任意指定的字符串。

3. 源程序代码

```

001 //-----TG126410.c-----
002 // 名称: TG126410LCD 显示驱动程序(SEG1565) (不带字库)
003 //-----
004 #include <avr/io.h>
005 #include <util/delay.h>
006 #include <avr/pgmspace.h>
007 #define INT8U unsigned char
008 #define INT16U unsigned int
009
010 //液晶引脚操作定义
011 #define SI_1() PORTD |= _BV(PD0)
012 #define SI_0() PORTD &= ~_BV(PD0)
013 #define SCL_1() PORTD |= _BV(PD1)
014 #define SCL_0() PORTD &= ~_BV(PD1)
015 #define CS1_1() PORTD |= _BV(PD2)
016 #define CS1_0() PORTD &= ~_BV(PD2)
017 #define RES_1() PORTD |= _BV(PD3)
018 #define RES_0() PORTD &= ~_BV(PD3)
019 #define A0_1() PORTD |= _BV(PD4)
020 #define A0_0() PORTD &= ~_BV(PD4)
021
022 //SEG1565 显示地址控制命令
023 #define PAGE 0xB0 //页地址
024 #define COL_H4 0x10 //列地址高 4 位
025 #define COL_L4 0x00 //列地址低 4 位
026 #define LINE 0x40 //行地址
027
028 //大写字母 R 的 8x8 点阵(纵向取模,字节倒序)
029 const INT8U R[8] = { 0x00,0xFE,0x12,0x32,0x52,0x8C,0x00,0x00 };
030
031 //案例文件夹下 BMP 位图文件的点阵数据(存放于程序 Flash 空间)-----
032 prog_uchar ICONs_Picture[1024] = { //纵向取模,字节倒序
033 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
034 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
.....限于篇幅,这里省略了图像的大部分点阵数据

```

```

095 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
096 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
097 };
098
099 //-----
100 //写指令
101 //-----
102 void Write_Command(INT8U cmd)
103 {
104     INT8U i;
105     CS1_0(); A0_0(); _delay_us(4);      //A0 设为 0,选择命令寄存器
106     for(i = 0; i<8; i++)                //串行写入一字节命令
107     {
108         SCL_0(); if (cmd & 0x80) SI_1(); else SI_0();
109         cmd <<= 1;
110         SCL_1(); _delay_us(4);          //时钟上升沿写入
111     }
112     _delay_us(4);
113     CS1_1();
114 }
115
116 //-----
117 //写数据
118 //-----
119 void Write_Data(INT8U dat)
120 {
121     INT8U i;
122     CS1_0(); A0_1(); _delay_us(4);      //A0 设为 1,选择数据寄存器
123     for(i = 0; i<8; i++)                //串行写入一字节数据
124     {
125         SCL_0(); if (dat & 0x80) SI_1(); else SI_0();
126         dat <<= 1;
127         SCL_1(); _delay_us(4);          //时钟上升沿写入
128     }
129     _delay_us(4);
130     CS1_1();
131 }
132
133 //-----
134 //LCD 初始化
135 //-----
136 void LCD_Initialize()
137 {

```

```

138     RES_0(); _delay_ms(10); RES_1();
139     Write_Command(0xA2); _delay_ms(10); //设置偏压比为1/7
140     Write_Command(0xA1); _delay_ms(10); //设置段驱动方为逆向
141     Write_Command(0xC8); _delay_ms(10); //设置COM扫描方为逆向
142     Write_Command(0x27); _delay_ms(10); //设置电阻率
143     Write_Command(0x81); _delay_ms(10); //设置电量寄存器
144     Write_Command(0x1B); _delay_ms(10); //设置电量
145     Write_Command(0x2C); _delay_ms(10); //依次打开倍压电路
146     Write_Command(0x2E); _delay_ms(10); //内部电压调整
147     Write_Command(0x2F); _delay_ms(10); //开启偏置电路功能
148     Write_Command(0xA4); _delay_ms(10); //正常显示所有点
149     Write_Command(0xAF); _delay_ms(10); //开显示
150 }
151
152 //-----
153 //全屏显示或全屏不显示(形成全黑色屏幕或底色屏幕)
154 //-----
155 void Full_Displ_ON_OFF(INT8U k)
156 {
157     INT8U i,j;
158     Write_Command(LINE);           //设置显示起始行为第0行(高2位01为命令,低6位全0)
159     for(i = 0; i < 8; i++)        //全屏输出,共8页(0~7)
160     {
161         Write_Command(PAGE + i); //选择第i页
162         Write_Command(COL_H4);   //列地址高4位设为0000
163         Write_Command(COL_L4);   //列地址低4位设为0000
164         for(j = 0; j < 128; j++) //输出第i页的0~127列(列地址自动递增)
165         {
166             if (k == 1) Write_Data(0xFF); //各列全部输出11111111
167             else      Write_Data(0x00); //否则输出00000000
168         }
169     }
170 }
171
172 //-----
173 //显示边框
174 //-----
175 void Disp_Frame()
176 {
177     INT8U i,j;
178     //第0页输出---
179     Write_Command(LINE);           //设置显示起始行
180     Write_Command(PAGE);          //选择第0页

```

```

181     Write_Command(COL_H4);           //设置起始列为第 0 列
182     Write_Command(COL_L4);
183     Write_Data(0xFF);              //垂直输出 1 列,8 个点
184     for(j = 0; j < 126; j++)       //输出 126 列
185     {
186         Write_Data(0x01);          //各列为 00000001,这使最上面一行出现一条横线
187     }
188     Write_Data(0xFF);              //该页第 127 列输出 8 个点
189     //第 1~6 页输出 -----
190     for(i = 1; i < 7; i++)        //输出 1~6 页
191     {
192         Write_Command(PAGE + i);   //选择第 i 页(1~6)
193         Write_Command(COL_H4);    //设置列地址高 4 位
194         Write_Command(COL_L4);    //设置列地址低 4 位
195         Write_Data(0xFF);        //第 0 列输出 8 个点
196         for(j = 0; j < 126; j++) //接下来输出 126 列
197         {
198             Write_Data(0x00);      //输出 00000000,与屏幕底色相同
199         }
200         Write_Data(0xFF);        //第 127 列显示 8 个点
201     }
202     //第 7 页输出 -----
203     Write_Command(PAGE + 7);       //输出第 7 页
204     Write_Command(COL_H4);        //设置列地址高 4 位
205     Write_Command(COL_L4);        //设置列地址低 4 位
206     Write_Data(0xFF);            //第 0 列输出 8 个点
207     for(j = 0; j < 126; j++)    //输出 126 列
208     {
209         Write_Data(0x80);          //各列输出 10000000,这使最下面一行输出一条横线
210     }
211     Write_Data(0xFF);            //最后一页最后一列输出 8 个点
212 }
213
214 //-----
215 // 正显与反显棋盘
216 //-----
217 void Disp_Checker(INT8U k)
218 {
219     INT8U i,j;
220     Write_Command(LINE);          //设置显示起始行
221     for(i = 0; i < 8; i++)        //全屏共 8 页
222     {
223         Write_Command(PAGE + i);  //选择第 i 列

```



```
224     Write_Command(COL_H4);      //输出列地址高 4 位
225     Write_Command(COL_L4);      //输出列地址低 4 位
226     for(j = 0; j < 64; j++)    //每页 64 次输出
227     {
228         if (k == 0)            //k = 0 或 1 时,每次输出 2 字节(2 列) 每页 64 * 2 = 128 列
229         {
230             Write_Data(0xAA); Write_Data(0x55); //正显:10101010 01010101
231         }
232         else
233         {
234             Write_Data(0x55); Write_Data(0xAA); //反显:01010101 10101010
235         }
236     }
237 }
238 }
239
240 //-----
241 // R 字符平铺画面
242 //-----
243 void Disp_R()
244 {
245     INT8U i,j,k;
246     Write_Command(LINE);          //设置显示起始行地址
247     for(i = 0; i < 8; i++)        //全屏共输出 8 页
248     {
249         Write_Command(PAGE + i);   //选择第 i 页
250         Write_Command(COL_H4);    //设置列地址高 4 位
251         Write_Command(COL_L4);    //设置列地址低 4 位
252         for(j = 0; j < 16; j++)   //每页横向显示 16 个 R
253         {
254             //输出一个 R 字符的 8 列点阵数据字节
255             for(k = 0; k < 8; k++) Write_Data(R[k]);
256         }
257     }
258
259 //-----
260 // 显示案例文件夹下的一幅图片
261 //-----
262 void Disp_Picture()
263 {
264     INT8U i,j;
265     Write_Command(LINE);          //设置显示起始行地址
266     for(i = 0; i < 8; i++)        //全屏共输出 8 页
```

```

267  {
268      Write_Command(PAGE + i); //选择第 i 页
269      Write_Command(COL_H4); //设置列地址高 4 位
270      Write_Command(COL_L4); //设置列地址低 4 位
271      //用 pgm_read_byte 从程序 Flash 空间中读取点阵数据
272      for(j = 0; j < 128; j++) //每页显示 128 列,列地址自动递增
273          Write_Data(pgm_read_byte(ICONS_Picture + i * 128 + j));
274  }
275 }

```

```

01 //----- main.c -----
02 // 名称: TG126410 液晶串行模式演示
03 //-
04 // 说明: 本例用按键与开关控制 TG126410 显示不同画面,LCD 工作于串行模式。
05 //      本例所显示的几幅画面常用于对液晶屏进行显示测试
06 //
07 //-
08 #include <avr/io.h>
09 #include <util/delay.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int
12
13 //按键定义
14 #define K1_DOWN() ((PINB & _BV(PB1)) == 0x00) //前翻
15 #define K2_DOWN() ((PINB & _BV(PB4)) == 0x00) //后翻
16 #define K3_DOWN() ((PINB & _BV(PB7)) == 0x00) //自动刷新
17
18 //画面总数及当前画面页索引
19 INT8U MaxPage = 7, CurrentPageIndex = 0;
20 //控制是否继续显示下一幅图像的标识变量
21 enum {FALSE,TRUE} ShowNext = FALSE;
22
23 //12864LCD 显示与屏幕测试相关函数
24 extern void LCD_Initialize();
25 extern void Full_Displ_ON_OFF(INT8U k);
26 extern void Disp_Checker(INT8U k);
27 extern void Disp_Frame();
28 extern void Disp_R();
29 extern void Disp_Clip();
30 extern void Disp_Picture();
31 //-
32 // 按键扫描
33 //-

```



```
34 void Scan_KEYs()
35 {
36     if(K3_DOWN()) //开关合上时自动刷新
37     {
38         ShowNext = TRUE;
39         if (++CurrentPageIndex == MaxPage) CurrentPageIndex = 0;
40         _delay_ms(200);
41     }
42     else if(K1_DOWN()) //前翻
43     {
44         ShowNext = TRUE;
45         if(CurrentPageIndex > 0)
46             CurrentPageIndex--;
47         else
48             CurrentPageIndex = MaxPage - 1;
49     }
50     else if(K2_DOWN()) //后翻
51     {
52         ShowNext = TRUE;
53         if (++CurrentPageIndex == MaxPage) CurrentPageIndex = 0;
54     }
55     PORTA = ~_BV(CurrentPageIndex); //刷新指示 LED
56 }
57
58 //-----
59 // 主程序
60 //-----
61 int main()
62 {
63     DDRA = 0xFF; PORTA = 0xFF; //配置端口
64     DDRB = 0x00; PORTB = 0xFF;
65     DDRD = 0xFF;
66     LCD_Initialize(); _delay_ms(5); //液晶初始化
67     Full_Disp_ON_OFF(0); //全屏不显示
68     _delay_ms(200);
69     Full_Dispatch(1); //全显(形成全黑色屏幕)
70     while (1)
71     {
72         Scan_KEYs(); //键盘扫描
73         if(ShowNext == TRUE)
74         {
75             switch(CurrentPageIndex)
76             {
```

```

77         case 0 : Disp_R();           break; //R字符平铺画面
78         case 1 : Disp_Frame();     break; //方框
79         case 2 : Full_Displ_ON_OFF(1); break; //全显(形成全黑色屏幕)
80         case 3 : Full_Displ_ON_OFF(0); break; //全不显(底色屏幕)
81         case 4 : Disp_Checker(1);   break; //正显棋盘
82         case 5 : Disp_Checker(0);   break; //反显棋盘
83         case 6 : Disp_Picture();   break; //案例文件夹下的一幅图片
84     }
85     ShowNext = FALSE;
86 }
87 }
88 }

```

4.21 用带 SPI 接口的 MCP23S17 扩展 16 位通用 I/O 端口

MCP23017/MCP23S17 都是 16 位的接口扩展芯片,前者通过 I²C 接口与 MPU 连接,后者则通过 SPI 接口连接。本例使用 MCP23S17 进行接口扩展,GPB6 与 GPB7 引脚的按键可以控制 GPA0~7 及 GPB0~1 引脚连接的条形 LED 按不同方向滚动。本例电路及运行效果如图 4-32 所示。

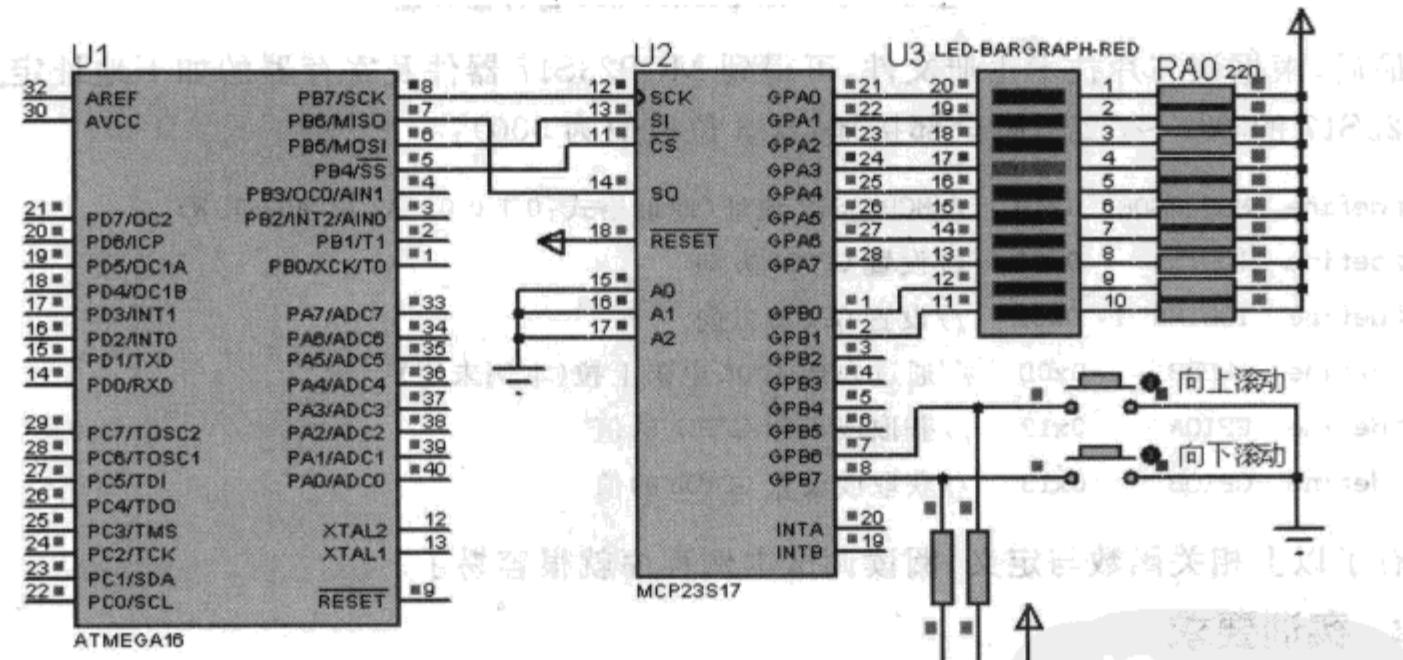


图 4-32 用带 SPI 接口的 MCP23S17 扩展 16 位通用 I/O 端口

1. 程序设计与调试

使用 MCP23S17 扩展接口时,需要首先对 SPI 接口进行初始化,SPI_MasterInit 函数完成了这项工作。除完成端口配置以外,SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR0)将 SPI 接口控制寄存器 SPCR 的 SPE、MSTR、SPR0 置位,其中 SPE 使能 SPI,MSTR 置位选择主机(Master)模式,SPCR 寄存器的最低 2 位为 SPR1、SPR0,它们被设为 01,选择 SCK 频率为系统时钟 16 分频。

为通过 SPI 接口进行数据传输,程序中提供了函数 SPI_Transmit,其中关键语句如下:



```
SPDR = dat;  
while(!(SPSR & _BV(SPIF)));
```

第 1 行通过写 SPI 数据寄存器 SPDR 启动发送, 程序随后开始等待 SPI 状态寄存器中 SPIF 置位, 这 2 行与 USART 程序设计中的以下 2 行语句很相似:

```
UDR = c;  
while(!(UCSRA & _BV(UDRE)));
```

第 1 行将待发送的字符放入 USART 收发缓冲器 UDR 进行发送, 第 2 行随后轮询 USART 控制与状态寄存 UCSRA, 直到 UCSRA 寄存器中的 UDRE 位被硬件置位。

有了 SPI 接口初始化函数 SPI_MasterInit 及 SPI 接口数据传送函数 SPI_Transmit, 为对 MCP23S17 进行访问, 还需要知道 MCP23S17 的 SPI 寄存器寻址格式。根据图 4-33 可知, 访问 MCP23S17 的寄存器时, 需要先发送设备操作码(0100—A2A1A0—R/W, 然后发送寄存器地址, 其中 R/W 位控制读/写 MCP23S17, 根据该图可以很容易编写出读/写 MCP23S17 的函数。

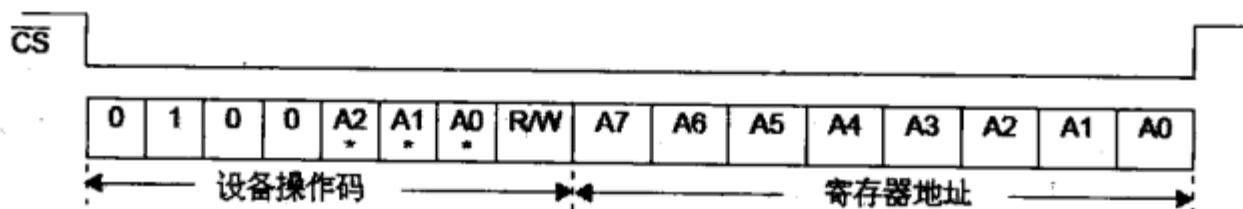


图 4-33 MPC23S17 SPI 寄存器寻址

最后, 根据该芯片技术手册文件, 可得到 MCP23S17 器件及寄存器的如下地址定义(本例 MCP23S17 的 A0 ~ A2 引脚全部接地, 这 3 位取值为 000):

```
#define MCP_ADDR 0x40 //MCP23S17 地址(地址格式:0 1 0 0 - A2 A1 A0 - R/W)  
#define IODIRA 0x00 //设置 GPIOA 方向  
#define IODIRB 0x01 //设置 GPIOB 方向  
#define GPPUB 0x0D //通过内部 100K 电阻上拉(本例未用)  
#define GPIOA 0x12 //获取或设置 GPIOA 的值  
#define GPIOB 0x13 //获取或设置 GPIOB 的值
```

有了以上相关函数与定义, 阅读调试本例程序就很容易了。

2. 实训要求

- ① 重新配置 MCP23S17 的硬地址引脚 A0~A2, 使用多片接口扩展芯片进行扩展实验。
- ② 将本例接口扩展芯片改为兼容 I²C 接口的 MCP23017, 重新编写程序, 实现与本例相同的运行效果。

3. 源程序代码

```
001 //--  
002 // 名称: 用带 SPI 接口的 MCP23S17 扩展 16 位通用 I/O 端口  
003 //--  
004 // 说明: 本程序将 MCP23S17 的 GPIOA 的 8 位及 GPIOB 的低 4 位设为输出端口,  
005 // 将 GPIOB 的高 4 位设为输出端口, 演示了条形 LED 在按键控制的下
```

```

006 //          的滚动效果
007 //
008 //-----
009 #define F_CPU 4000000UL
010 #include <avr/io.h>
011 #include <avr/interrupt.h>
012 #include <util/delay.h>
013 #define INT8U unsigned char
014 #define INT16U unsigned int
015
016 //MCP23S17 器件及寄存器地址定义
017 #define MCP_ADDR 0x40          //MCP23S17 地址(地址格式:0 1 0 0 A2 A1 A0 R/W)
018 #define IODIRA 0x00           //设置 GPIOA 方向
019 #define IODIRB 0x01           //设置 GPIOB 方向
020 #define GPPUB 0x0D             //通过内部 100 kΩ 电阻上拉(本例未用)
021 #define GPIOA 0x12             //获取或设置 GPIOA 的值
022 #define GPIOB 0x13             //获取或设置 GPIOB 的值
023
024 //SPI 使能与禁用
025 #define SPI_EN() PORTB &= ~_BV(PB4)
026 #define SPI_DI() PORTB |= _BV(PB4)
027
028 //当前演示操作序号(0,1)
029 INT8U Demo_OP_No = 0;
030 //-----
031 // SPI 主机初始化
032 //-----
033 void SPI_MasterInit()
034 {
035     //第 4、5、7 位分别为~CS、SI、SCK,设为输出,第 6 位为 MISO,设为输入
036     DDRB = 0B10110000; PORTB = 0xFF;
037     //SPI 使能,主机模式,16 分频
038     SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR0);
039 }
040
041 //-----
042 // SPI 数据传输
043 //-----
044 INT8U SPI_Transmit(INT8U dat)
045 {
046     SPDR = dat;                  //启动数据传输
047     while(!(SPSR & _BV(SPIF))); //等待结束
048     return SPDR;

```



```
049 }
050
051 //-----
052 // 向 MCP23S17 写入器件地址、寄存器地址、命令/数据共 3 个字节
053 //-----
054 void Write_MCP23S17(INT8U Device_addr, INT8U Reg_addr, INT8U CD)
055 {
056     SPI_EN();
057     SPI_Transmit(Device_addr);
058     SPI_Transmit(Reg_addr);
059     SPI_Transmit(CD);
060     SPI_DI();
061 }
062
063 //-----
064 // 根据器件地址、寄存器地址从 MCP23S17 读字节
065 //-----
066 void Read_MCP23S17(INT8U Device_addr, INT8U Reg_addr, INT8U * Dat)
067 {
068     SPI_EN();
069     SPI_Transmit(Device_addr | 0x01); //将 R/W 位设为读
070     SPI_Transmit(Reg_addr);
071     * Dat = SPI_Transmit(0xFF);
072     SPI_DI();
073 }
074
075 //-----
076 // 初始化 MCP23S17
077 //-----
078 void Initialise_MCP23S17()
079 {
080     //设置 I/O 方向(1 为输入,0 为输出)
081     Write_MCP23S17(MCP_ADDR,IODIRA,0x00);
082     Write_MCP23S17(MCP_ADDR,IODIRB,0xF0);
083     //清除 GPIOA,GPIOB 所有位
084     Write_MCP23S17(MCP_ADDR,GPIOA,0x00);
085     Write_MCP23S17(MCP_ADDR,GPIOB,0x00);
086 }
087
088 //-----
089 // 按键处理
090 //-----
091 void Key_Handle()
```

```

092  {
093      INT8U Key_Port_Status;
094      //从 MCP23S17 的 GPIOB 端口读取按键值
095      Read_MCP23S17(MCP_ADDR,GPIOB,&Key_Port_Status);
096      //根据按键改变当前演示操作序号 Demo_OP_No
097      //如果未按键则保持原演示序号
098      if ((Key_Port_Status & 0x80) == 0x00) Demo_OP_No = 0;
099      else
100         if ((Key_Port_Status & 0x40) == 0x00) Demo_OP_No = 1;
101     }
102
103 //-----
104 // 主程序
105 //-----
106 int main()
107 {
108     INT8U i; INT16U Pattern;
109     DDRB = 0xFF;           //配置端口
110     SPI_MasterInit();    //SPI 主机初始化
111     Initialise_MCP23S17(); //MCP23S17 初始化
112     while(1)
113     {
114         if (Demo_OP_No == 0)          //条形 LED 向上滚动演示
115         {
116             Pattern = 0xFFFF;
117             for (i = 0; i < 10; i++)
118             {
119                 Write_MCP23S17(MCP_ADDR,GPIOA,(INT8U)Pattern);
120                 Write_MCP23S17(MCP_ADDR,GPIOB,(INT8U)(Pattern>>8));
121                 Pattern = Pattern << 1 | 0x0001;
122                 Key_Handle();
123                 if (Demo_OP_No != 0) break;
124                 _delay_ms(10);
125             }
126         }
127         else                         //条形 LED 向下滚动演示
128         {
129             Pattern = 0x01FF;
130             for (i = 0; i < 10; i++)
131             {
132                 Write_MCP23S17(MCP_ADDR,GPIOA,(INT8U)Pattern);
133                 Write_MCP23S17(MCP_ADDR,GPIOB,(INT8U)(Pattern>>8));

```

```

134     Pattern = Pattern >> 1 | 0x0200;
135     Key_Handle();
136     if (Demo_OP_No != 1) break;
137     _delay_ms(10);
138 }
139 }
140 }
141 }

```

4.22 用 TWI 接口控制 MAX6953 驱动 4 片 5×7 点阵显示器

MAX6953 是紧凑的行共阴显示驱动器，通过 I²C 兼容的串行接口将微控制器连接至 5×7LED 点阵屏。MAX6953 可驱动多达 4 位单色或 2 位双色的 5×7 点阵屏，6953 包含 104 个 ASCII 字符字模、复用扫描电路、行列驱动器以及用于存储每一位字符和 24 个用户自定义字模数据的静态 RAM。LED 的段电流由内部逐位数字亮度控制电路设定。该器件具有低功耗的关断模式、段闪烁控制以及强制所有 LED 打开的测试模式。

本例运行时，所设定的字符串将在 4 位 LED 点阵显示屏上滚动显示。本例电路及部分运行效果如图 4-34 所示。

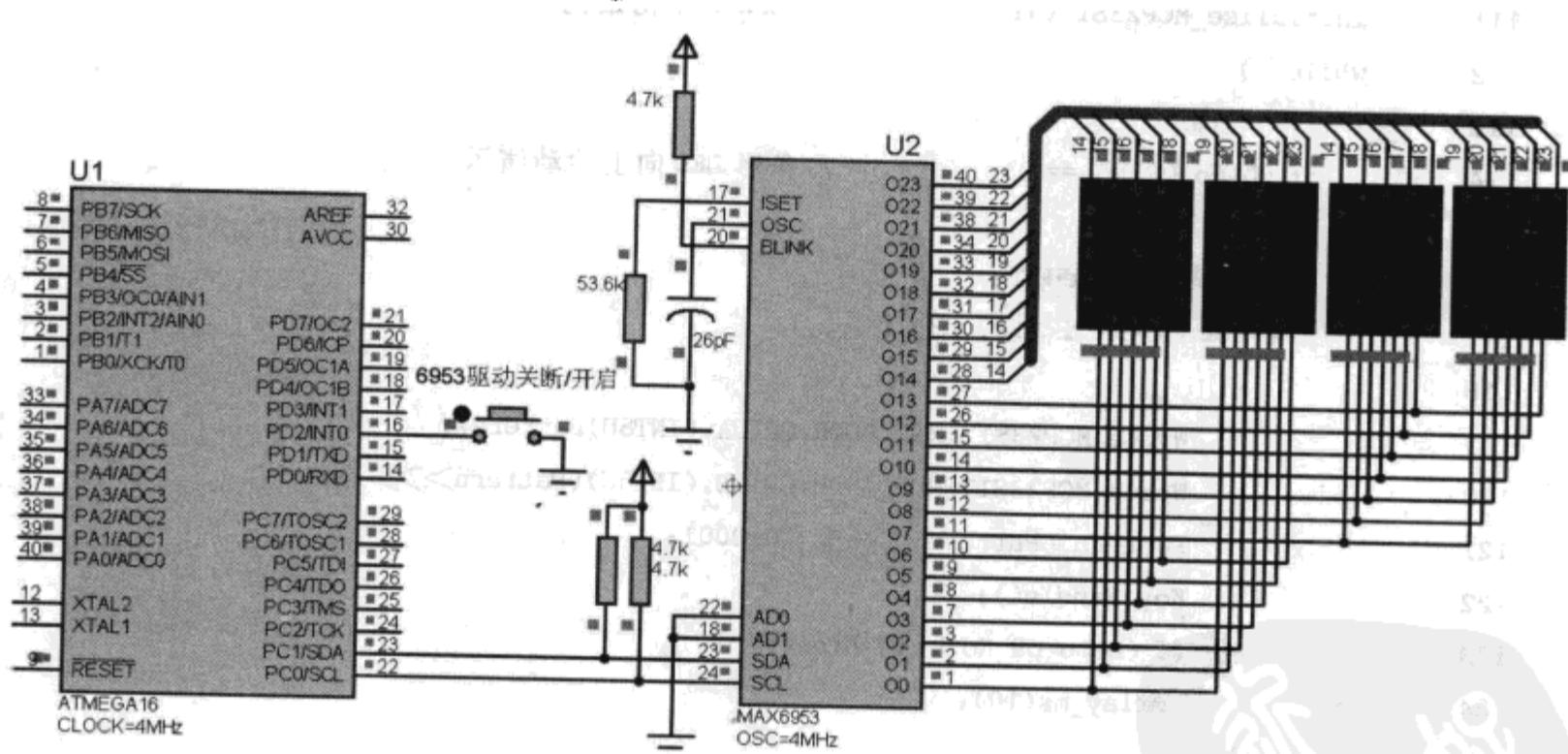


图 4-34 用 TWI 接口控制 MAX6953 驱动 4 片 5×7 点阵显示器

1. 程序设计与调试

本例单片机通过 TWI 接口与兼容 I²C 接口的 MAX6953 相连，在 TWI 通用操作宏定义中，涉及 TWI 控制寄存器 TWCR、状态寄存器 TDSR 和数据寄存器 TWDR，其中：

TWCR 中的 TWINT、TWEA、TWSTA、TWSTO、TWEN 分别为 TWI 中断标志、使能 TWI 应答、TWI START 状态标志、TWI STOP 状态标志、TWI 使能标志。

TWSR 的高 5 位为 TWI 总线状态位, twi.h 给出了读取 TWI 总线状态的宏定义:

```
#define TW_STATUS (TWSR & TW_STATUS_MASK)
```

其中 TW_STATUS_MASK 定义为 0xF8, 它用于获取该寄存器的高 5 位, 不同返回值的含义可参考 ATmega16 技术手册文件, 头文件<util/twi.h>定义了所有状态码。

最后是 8 位的 TWI 数据寄存器 TWDR, 它包含的是接收或发送的字节。

本例所使用的显示驱动芯片 MAX6953 的引脚定义如下:

- ① O0~O13 是共阴 LED 驱动位, 它们从显示器的共阴行上吸入电流。
- ② O14~O23 是 LED 阳极驱动位, 它们向共阳列上输出电流。
- ③ SDA、SCL 分别是 I²C 兼容的串行数据位与串行时钟位。
- ④ AD0、AD1 是地址输入位, 用于设置子器件(或称从器件)地址。
- ⑤ ISET 用于设置段电流, 在 ISET 与地之间串接 RSET 电阻可设置峰值电流。
- ⑥ BLINK 为闪烁控制位, 输出开漏。

本例 MAX6953 的 AD0、AD1 引脚全部接地, 根据技术手册可知从器件地址为 10100000。

通过单片机 TWI 接口控制从器件 6953, 还需要知道其命令寄存器地址。下面给出的部分命令寄存器地址是本例中所用到的:

- 0x00——无操作;
- 0x01——第 1、0 两位的亮度设置;
- 0x02——第 3、2 两位的亮度设置;
- 0x03——设置扫描范围;
- 0x04——配置, 控制关断及闪烁;
- 0x05——用户自定义字体;
- 0x06——出厂设置;
- 0x07——显示测试;

0x20~0x27——P0 显示平面的数位 0~7 的寄存器地址, 本例未启用闪烁, 故而未使用 P1 显示平面。

有关上述各命令寄存器的更多详细资料, 可参考 MAX6953 的技术手册, 根据命令寄存器地址及相关技术手册, 初始化程序及读/写程序就很容易编写了。

2. 实训要求

- ① MAX6953 字符表 16 行 8 列共 128 个字符中, 0x00~0x17 这 24 个编码为自定义字符编码, 完成本例调试后, 设计部分自定义字符点阵数据, 通过自定义字符命令 0x05 创建并编程显示。
- ② 重新设计本例, 用多片 MAX6953 驱动更大幅面的 LED 点阵显示屏, 并实现对闪烁功能的开关控制。
- ③ 重新编程在其他端口某两只引脚上模拟 I²C 时序, 实现与本例相同的显示效果。
- ④ 改用兼容 SPI 接口的 MAX6952 重新设计本例, 实现相同的显示效果。

3. 源程序代码

```

01 //-----
02 // 名称：用 TWI 接口控制 MAX6953 驱动 4 片 5 * 7 点阵显示器
03 //-----
04 // 说明：本例运行时，4 块点阵屏将滚动显示一组信息串，信息串中的字符
05 // 点阵信息由 MAX6953 提供，本例不需要为各字符单独提供字模点阵。
06 // 运行过程中通过按键命令可随时关断或开启 6953
07 //
08 //-----
09 #define F_CPU 4000000UL
10 #include <avr/io.h>
11 #include <avr/interrupt.h>
12 #include <util/twi.h>
13 #include <util/delay.h>
14 #include <string.h>
15 #define INT8U unsigned char
16 #define INT16U unsigned int
17 #define INT32U unsigned long
18
19 //子器件地址
20 #define MAX6953R 0B10100001 //1 = READ
21 #define MAX6953W 0B10100000 //0 = WRITE
22
23 //TWI 通用操作
24 #define Wait() while ((TWCR & _BV(TWINT)) == 0)
25 #define START() {TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN); Wait();}
26 #define STOP() {TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN)}
27 #define WriteByte(x) {TWDR = (x); TWCR = _BV(TWINT) | _BV(TWEN); Wait();}
28 #define ACK() {TWCR |= _BV(TWEA)}
29 #define NACK() {TWCR &= ~_BV(TWEA)}
30
31 //4 块点阵屏滚动显示的信息串
32 char LED_String[] = "LEDSHOW: <---- 0123456789";
33 //-----
34 // 写 MAX6953 子程序
35 //-----
36 INT8U MAX6953_Write(INT8U addr, INT8U dat)
37 {
38     START(); //启动
39     if(TW_STATUS != TW_START) return 0;
40     WriteByte(MAX6953W); //发送器件地址

```

```

41     if(TW_STATUS != TW_MT_SLA_ACK)  return 0;
42     WriteByte(addr);           //发送从器件寄存器地址
43     if(TW_STATUS != TW_MT_DATA_ACK) return 0;
44     WriteByte(dat);           //发送数据
45     if(TW_STATUS != TW_MT_DATA_ACK) return 0;
46     STOP();
47     _delay_ms(2);
48     return 1;
49 }
50
51 //-----
52 // MAX6953 初始化
53 //-----
54 void MAX6953_INIT()
55 {
56     MAX6953_Write(0x01, 0xFF);    //数位 0、1 的亮度设置(最大亮度)
57     MAX6953_Write(0x02, 0xFF);    //数位 2、3 的亮度设置(最大亮度)
58     MAX6953_Write(0x03, 0x03);    //设置扫描位数范围为 0~3(共 4 片点阵屏)
59     MAX6953_Write(0x04, 0x01);    //设置非关断模式
60     MAX6953_Write(0x07, 0x00);    //不进行测试
61 }
62
63 //-----
64 // 主程序
65 //-----
66 int main()
67 {
68     INT8U i,j;
69     DDRD = 0x00; PORTD = 0xFF;    //配置端口
70     MCUCR = 0x02;                //INT0 为下降沿触发
71     GICR = 0x40;                 //INT0 中断使能
72     SREG = 0x80;                  //使能总中断(或使用 sei() 函数)
73     MAX6953_INIT();             //MAX6953 初始化设置
74     while(1)
75     {
76         for (i = 0; i <= strlen(LED_String) - 4; i++)
77         {
78             //将第 i 个字符开始的 4 个字符逐个发送到
79             //MAX6953 各数位地址: 0x20,0x21,0x22,0x23
80             for (j = 0; j<4; j++)
81                 MAX6953_Write(0x20 | j, (INT8U)LED_String[i + j]);
82             _delay_ms(300);
83         }
}

```



```
84     _delay_ms(2000);
85 }
86 }
87
88 //-----
89 // INT0 中断函数控制点阵屏关断或开启
90 //-----
91 ISR (INT0_vect)
92 {
93     static INT8U Shut_Down_6955 = 0x01;
94     Shut_Down_6955 ^= 0x01;
95     MAX6953_Write(0x04, 0x00 | Shut_Down_6955); //关断 0x00,非关断 0x01
96 }
```

4.23 用 TWI 接口控制 MAX6955 驱动 16 段数码管显示

MAX6955 也是一种紧凑型的显示驱动器,兼容 I²C 接口,可驱动多达 16 位 7 段、8 位 14 段、8 位 16 段或 128 个分立的 LED,器件还包括 5 条 I/O 扩展线。器件内部包含全部 14 段和 16 段 104 个 ASCII 字符的字模、7 段显示使用的十六进制字模、多工扫描电路、阳极和阴极驱动器以及用于存储各位显示的静态 RAM。显示位的最大段电流可用单个外部电阻设定,各位的显示亮度可用内部的 16 级数字亮度控制电路独立调节,限斜率段电流驱动器降低 EMI。MAX6955 还包含低功耗关断模式、限制扫描位寄存器、段闪烁控制以及强制所有 LED 点亮的测试模式。

本例电路及运行效果如图 4-35 所示。

1. 程序设计与调试

本例单片机通过 TWI 接口与兼容 I²C 接口的 MAX6955 相连,MAX6955 的引脚定义如下:

- ① P0~P4 是通用的 I/O 端口(GPIO),可配置为逻辑输入或开漏输出。
- ② AD0、AD1 是地址输入位,用于设置子器件地址。
- ③ SDA、SCL 分别是 I²C 兼容的串行数据位与串行时钟位。
- ④ O0~O18 是位码/段码驱动线,当作为位码驱动线时,O0~O7 从数码管共阴极吸人电流,当作为段码驱动线时,O0~O18 向阳极输出电流。
- ⑤ ISET 用于设置段电流,串接到 ISET 与 GND 之间的 R_{SET} 电阻可设置峰值电流。
- ⑥ BLINK 为闪烁时钟输出,输出开漏。
- ⑦ OSC 为多重时钟输入,使用内部振荡器时要将电容 C_{SET} 连接在 OSC 与 GND 之间。使用外部时钟时,要使用 1~8 MHz CMOS 时钟驱动 OSC。
- ⑧ OSC_OUT 为时钟推挽输出。

MAX6955 与上一案例中 MAX6953 的控制命令类似,具体细节请参考技术手册文件。

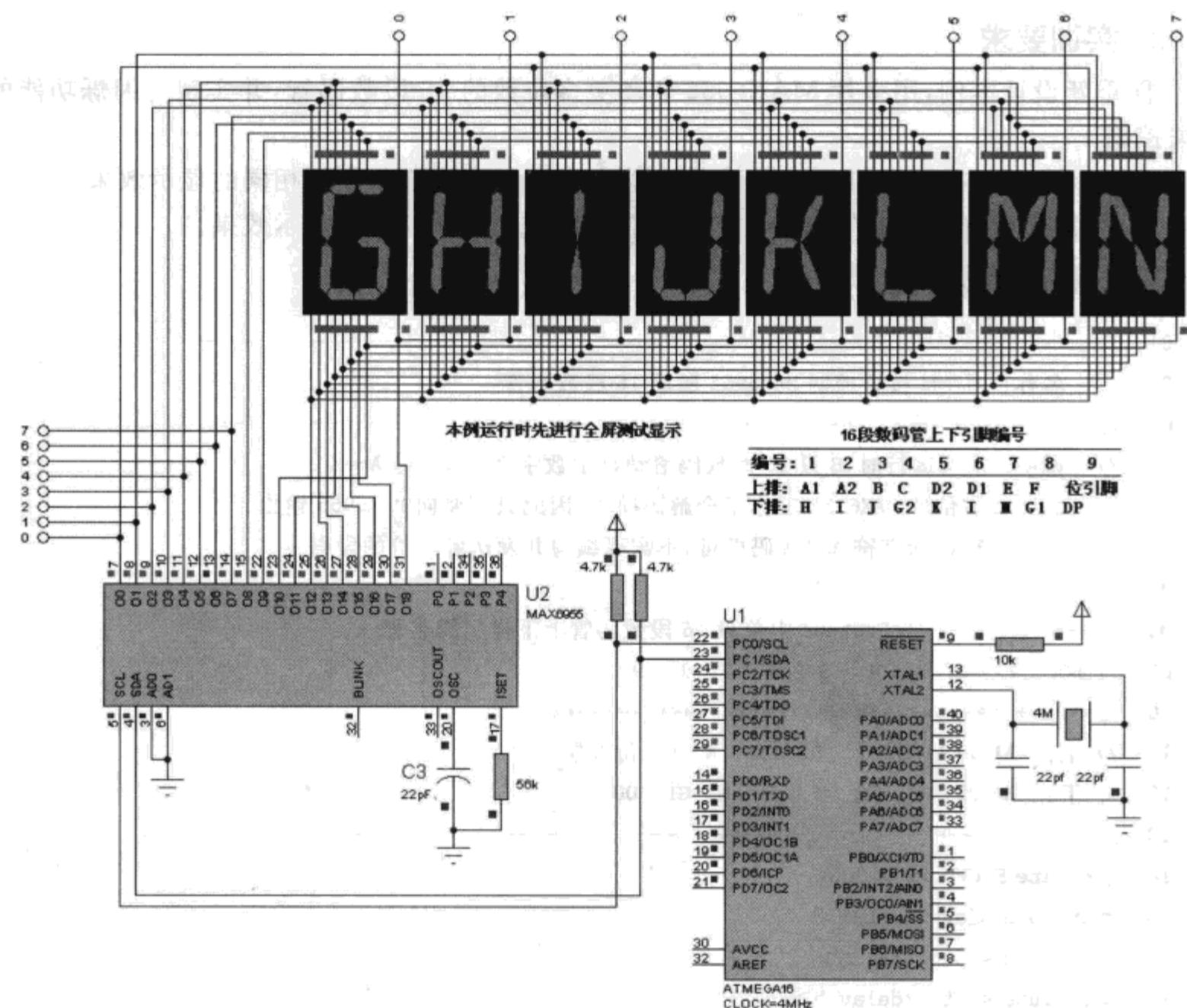


图 4-35 用 TWI 接口控制 MAX6955 驱动 16 段数码管显示

由于 MAX6955 的 O0~O18 引脚中, O0~O7 采用了段/位复用技术, 在设计本例电路时, 要参考表 4-17 来连接分立式 16 段共阴数码管与驱动芯片 MAX6955。表中第 1 行 O0~O7、O8~O18 是 MAX6955 的引脚, C0~C7 对应于 8 位数码管的位引脚。关于表中 a1、a2、b、c 等引脚与 16 段数码管引脚的对应关系, 可参考电路图或源程序代码的说明部分。

表 4-17 MAX6955 与 8 位分立式 16 段数码管的连接

位	O0 ~ O7								O8 ~ O18										
	C0	—	a1	a2	b	c	d1	d2	e	f	g1	g2	h	i	j	k	l	m	dp
0	—	C1	a1	a2	b	c	d1	d2	e	f	g1	g2	h	i	j	k	l	m	dp
1	—	C2	a1	a2	b	c	d1	d2	e	f	g1	g2	h	i	j	k	l	m	dp
2	a1	a2	—	C3	b	c	d1	d2	e	f	g1	g2	h	i	j	k	l	m	dp
3	a1	a2	—	—	C4	b	c	d1	d2	e	f	g1	g2	h	i	j	k	l	dp
4	a1	a2	b	c	—	C5	d1	d2	e	f	g1	g2	h	i	j	k	l	m	dp
5	a1	a2	b	c	—	—	C6	d1	d2	e	f	g1	g2	h	i	j	k	l	dp
6	a1	a2	b	c	d1	d2	—	C7	e	f	g1	g2	h	i	j	k	l	m	dp
7	a1	a2	b	c	d1	d2	—	—	e	f	g1	g2	h	i	j	k	l	m	dp



2. 实训要求

- ① 重新设计本例,用多片 MAX6955 驱动更多位数的 16 段数码管,并实现对闪烁功能的开关控制。
- ② 重新编程在其他端口某 2 只引脚上模拟 I²C 时序,实现与本例相同的显示效果。
- ③ 改用兼容 SPI 接口的 MAX6954 重新设计本例,仍实现本例显示效果。

3. 源程序代码

```
01 //-----  
02 // 名称: 用 TWI 接口控制 MAX6955 驱动 16 段数码管  
03 //-----  
04 // 说明: 本例运行时,8 只 16 段数码滚动显示数字 0~9,字母 A~Z  
05 // 本例使 MAX6955 工作于全解码模式,因此只需要向 MAX6955 输出  
06 // 待显示字符 ASCII 码即可,不需要编写并发送各字符的段码  
07 //-----  
08 //----- Proteus 中单只 16 段数码管上下排引脚名称-----  
09 // NO. 1 2 3 4 5 6 7 8 9  
10 //*****  
11 // 上: A1 A2 B C D2 D1 E F 位控制  
12 // 下: H I J G2 K I M G1 DP  
13 //-----  
14 #define F_CPU 4000000UL  
15 #include <avr/iq.h>  
16 #include <util/twi.h>  
17 #include <util/delay.h>  
18 #include <string.h>  
19 #define INT8U unsigned char  
20 #define INT16U unsigned int  
21  
22 //子器件地址 0xC0,0xC1  
23 #define MAX6955R 0B11000001 //1 = READ  
24 #define MAX6955W 0B11000000 //0 = WRITE  
25  
26 //TWI 通用操作  
27 #define Wait() while ((TWCR & _BV(TWINT)) == 0)  
28 #define START() {TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN); Wait();}  
29 #define STOP() {TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN)}  
30 #define WriteByte(x) {TWDR = (x); TWCR = _BV(TWINT) | _BV(TWEN); Wait();}  
31 #define ACK() (TWCR |= _BV(TWEA))  
32 #define NACK() (TWCR &= ~_BV(TWEA))  
33  
34 //16 段数码管滚动显示的字符串  
35 char SEG_LED_String[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```

36 //-----
37 // 写 MAX6955 子程序
38 //-----
39 INT8U MAX6955_Write(INT8U addr, INT8U dat)
40 {
41     START();
42     if(TW_STATUS != TW_START)      return 0;
43     WriteByte(MAX6955W);
44     if(TW_STATUS != TW_MT_SLA_ACK) return 0;
45     WriteByte(addr);
46     if(TW_STATUS != TW_MT_DATA_ACK) return 0;
47     WriteByte(dat);
48     if(TW_STATUS != TW_MT_DATA_ACK) return 0;
49     STOP();
50     _delay_ms(20);
51     return 1;
52 }
53
54 //-----
55 // MAX6955 初始化
56 //-----
57 void MAX6955_INIT()
58 {
59     MAX6955_Write(0x01, 0xFF);    //解码模式设置(全解码)
60     MAX6955_Write(0x02, 0x03);    //亮度设置
61     MAX6955_Write(0x03, 0x07);    //设置扫描范围 0~7
62     MAX6955_Write(0x04, 0x01);    //控制寄存器设置(非关断模式)
63                         //将 0x01 改为 0x0D 可使数码管以 0.5 s 周期闪烁
64     MAX6955_Write(0x06, 0x00);    //GPIO 设置为输出
65     MAX6955_Write(0x0C, 0x00);    //显示数字类型设置(数位 0~7 为 16 段或 7 段)
66
67     MAX6955_Write(0x07, 0x01);    //显示测试(各数码管 16 段全部点亮)
68     _delay_ms(1000);
69     MAX6955_Write(0x07, 0x00);    //关闭测试
70 }
71
72 //-----
73 // 主程序
74 //-----
75 int main()
76 {
77     INT8U i,j,Len = strlen(SEG_LED_String);
78     DDRD = 0xFF; PORTD = 0x00;

```



```
79     MAX6955_INIT();           //MAX6955 初始化设置
80     while (1)
81     {
82         for (i = 0; i < Len ; i += 8)
83         {
84             //MAX6955 数位 0~7 的地址：0x20~0x27,下面的循环每次发送 8 个字符
85             for (j = 0; j < 8 && i + j < Len; j++)
86                 MAX6955_Write(0x20 | j, (INT8U)SEG_LED_String[i + j]);
87             //如果最后一组不足 8 个字符则补充显示空格将余下部分清空
88             for (; j < 8; j++)
89                 MAX6955_Write(0x20 | j, (INT8U)(' '));
90             _delay_ms(2000);
91         }
92         _delay_ms(4000);
93     }
94 }
```

4.24 用 DAC0832 生成多种波形

DAC0832 是 8 位的 D/A 转换器件,转换结果以电流形式输出,为通过 DAC0832 生成所需要的波形,电路中采用运放 uA741 将电流信号转换为电压信号。案例电路及部分运行效果如图 4-36 所示。

1. 程序设计与调试

DAC0832 输出的是电流信号,本例用 uA741 运放将 DAC0832 经数/模转换后输出的电流信号转换为电压信号,按电路连接的不同可有单极输出和双极输出,单极输出时只有正电压或只有负电压,双极输出时电压在正负数值范围内变化。

本例将 DAC0832 的电流输出端 I_{OUT1} 连接至运放的反相输入端, I_{OUT2} 连接模拟地,得到的输出电压与参考电压 V_{REF} 反相,实现单极性输出,转换后输出的电压公式为 $-D \times V_{REF} / 255$,其中 D 为输入 DAC0832 的数据字节,通过改变输入给 DAC0832 的数据字节即可改变输出波形:

- ① 当输出的字节值由 0x00~0xFF 循环递增时,输出电压值由 5 V 向 0 V 循环递减,从而输出锯齿波效果。
- ② 当输出由 0x00~0xFF 循环递增,然后再由 0xFF~0x00 循环递减时,即形成三角波效果。
- ③ 同样,当使用正弦函数 sin 生成输出值时,即可得到正弦波。

2. 实训要求

- ① 改用 DAC0808 重新设计本例。

- ② 本例在输出正弦波时,单片机承担了大量的运算任务。调试本例后,先用其他编程工具按一定采样频率取得正弦波数据表并存入 Flash,然后编写单片机程序,根据数据表输出正弦波形。

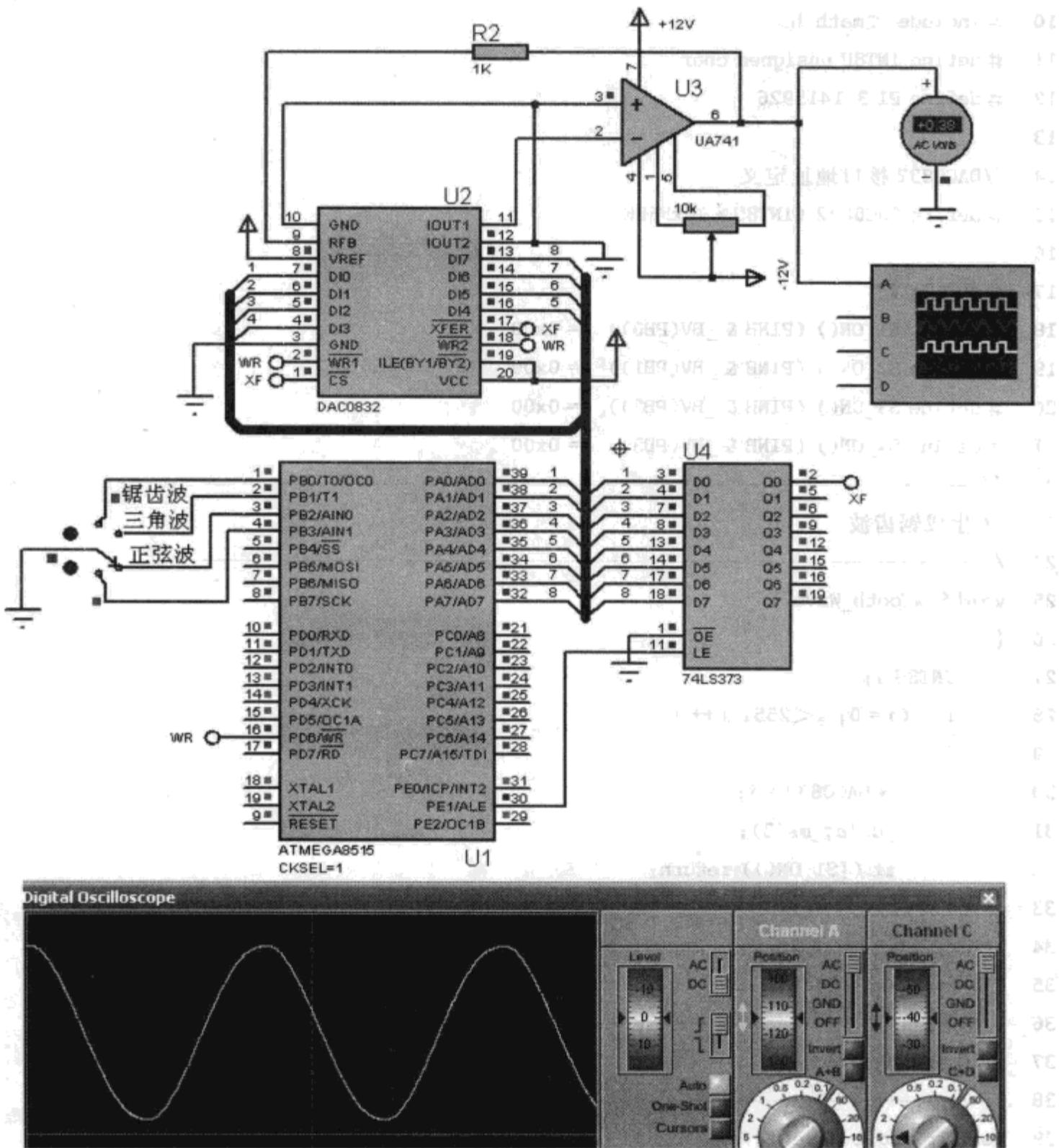


图 4-36 用 DAC0832 生成多种波形

3. 源程序代码

```

01 //-----
02 // 名称：用 DAC0832 生成多种波形
03 //-----
04 // 说明：本例运行时，通过切换开关，可分别输出锯齿波、三角波、正弦波
05 //
06 //-----
07 #define F_CPU 1000000UL
08 #include <avr/io.h>
09 #include <util/delay.h>

```



```
10 # include <math.h>
11 # define INT8U unsigned char
12 # define PI 3.1415926
13
14 //DAC0832 接口地址定义
15 # define DAC0832 (INT8U *)0xFFFF
16
17 //开关定义
18 # define S1_ON() (PINB & _BV(PB0)) == 0x00
19 # define S2_ON() (PINB & _BV(PB1)) == 0x00
20 # define S3_ON() (PINB & _BV(PB2)) == 0x00
21 # define S4_ON() (PINB & _BV(PB3)) == 0x00
22 //-----
23 // 生成锯齿波
24 //-----
25 void SawTooth_Wave()
26 {
27     INT8U i;
28     for (i = 0; i<255; i++)
29     {
30         *DAC0832 = i;
31         _delay_ms(3);
32         if (!S1_ON()) return;
33     }
34 }
35
36 //-----
37 // 生成三角波
38 //-----
39 void Triangle_Wave()
40 {
41     INT8U i;
42     for (i = 0; i<255; i++)
43     {
44         *DAC0832 = i;
45         _delay_ms(3);
46         if (!S2_ON()) return;
47     }
48     for (i = 255; i > 0; i--)
49     {
50         *DAC0832 = i;
51         _delay_ms(3);
52         if (!S2_ON()) return;
```

