AVR系列单片机

C语言编程与应用实例

金春林　邱慧芳　张皆喜　编著

清华大学出版社

AVR系列单片机

C语言编程与应用实例

内容提要

本书针对AT90LS8535单片机，以ICCAVR C语言为工具，详细介绍了ATMEL公司的RISC(精简指令集)系列单片机的内部结构、指令系统、C程序基础及应用系统的开发。本书适合大专院校电子、机电专业师生以及从事单片机开发工作的技术人员阅读。

本书特色

- 深入浅出，从最基本的概念开始，循序渐进地讲解单片机的应用开发。
- 列举了大量实例，使读者能从实际应用中掌握单片机的开发与应用技术。
- 本书系统地介绍了AVR单片机应用系统的开发，涵盖了整机设计中从硬件到软件编程的多个方面。

ISBN 7-302-07457-7

9 787302 074571 >

定价：30.00元

新书查询及技术支持：http://www.epress.cn

读者服务邮箱：service@wenyuan.com.cn

AVR 系列单片机
C 语言编程与应用实例

金春林　邱慧芳　张皆喜　编著

清华大学出版社
北　京

内 容 简 介

本书针对 Atmel 公司的 AVR 系列单片机和 ImageCraft 公司的 ICC AVR 开发环境，详细地介绍了 AT90LS8535 的 C 语言程序设计。全书共有 13 章，其内容既涉及到了单片机的结构原理、指令系统、内部资源和外部功能扩展，又包含了单片机的编程工具——ICC AVR C 编译器的数据类型、控制流、函数和指针等。

本书的特点是：深入浅出，从最基本的概念开始，循序渐进地讲解单片机的应用开发；列举了大量实例，使读者能从实际应用中掌握单片机的开发与应用技术。

本书适合作为从事单片机开发人员的参考用书。

图书在版编目（CIP）数据

AVR 系列单片机 C 语言编程与应用实例/金春林，邱慧芳，张皆喜编著.—北京：清华大学出版社，2003
ISBN 7-302-07457-7

Ⅰ.A… Ⅱ.①金…②邱…③张… Ⅲ.①单片微型计算机，AVR—程序设计②C 语言—程序设计
Ⅳ①TP368.1②TP312

中国版本图书馆 CIP 数据核字(2003)第 094372 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969
组稿编辑：张　瑜
文稿编辑：刘　颖
封面设计：陈刘源
印 装 者：北京国马印刷厂
发 行 者：新华书店总店北京发行所
开　　本：185×260　**印张**：21　**字数**：496 千字
版　　次：2003 年 11 月第 1 版　2003 年 11 月第 1 次印刷
书　　号：ISBN 7-302-07457-7/TP・5506
印　　数：1～5000
定　　价：30.00 元

前　言

自从 1976 年 Intel 公司推出 MCS-48 系列单片机以来，单片机技术得到了迅速的推广，已被广泛用于自动测量、智能仪表、工业控制及家用电器等各个方面。除了应用范围的扩大之外，单片机本身的发展也愈加迅速，如今，单片机的家族越来越庞大，系列产品越来越多，技术上也越来越先进。

AVR 系列单片机是 Atmel 公司生产的一种具有双总线结构的 RISC(Reduced Instruction Set Computer 的缩写，即精简指令系统计算机)单片机，它相对于传统的 CISC(Complicated Instruction Set Computer 的缩写，即复杂指令系统计算机)单片机而言，具有较短的指令周期和较快的运行速度。此外，AVR 单片机还具有片内 Flash、片内看门狗定时器、片内同步/异步串行接口、片内定时/计数器、片内模数转换器等多种内部资源。上述这些特点使采用 AVR 单片机的应用系统不仅具有运行速度快、结构简单、功能强大的特点，而且具有高可靠性和良好的经济性。

C 语言是一种编译型的结构化程序设计语言，具有简单的语法结构和强大的处理功能，具有运行速度快、编译效率高、移植性好和可读性强等多种优点，可以实现对系统硬件的直接操作。C 语言支持自顶向下的结构化程序设计方法，支持模块化程序设计结构。因此，用 C 语言来编写目标系统软件，可以大大缩短开发周期，且明显地增加软件的可读性，便于改进和扩充，从而开发出大规模、高性能的应用系统。

综上所述，以 AVR 单片机为基础，以 C 语言为工具开发单片机应用系统已成为单片机应用中的一个趋势。鉴于此，本书首先介绍了 AVR 单片机——AT90LS8535 的基本结构；然后介绍了 C 语言的数据类型、函数和指针等基础知识；最后结合大量实例，讲解了如何用 C 语言开发 AVR 系列单片机。因此，本书特别适用于单片机的初学者学习使用。

本书共 13 章。第一章为单片机简介；第 2 章为 AT90LS8535 单片机的基础知识；第 3 章介绍 AVR 单片机的 C 编程基础；第 4 章介绍 C 语言的数据结构；第 5 章介绍 C 的控制流；第 6 章介绍 C 的函数结构；第 7 章介绍 C 语言的指针类型；第 8 章介绍 C 的组合数据类型；第 9 章介绍 8535 的内部资源编程；第 10 章介绍 8535 的人机接口技术；第 11 章为 8535 单片机的扩展编程；第 12 章介绍 8535 的通讯接口；第 13 章介绍了系统设计中的程序处理方法。

由于作者的学识水平有限，书中难免有错误或不妥之处，恳请广大读者批评指正。

作者

2003 年 8 月于北京

目　录

第 1 章　单片机系统概述

单片机就是包括了中央处理器 CPU(Central Processor Unit)、随机存储器 RAM(Random Access Memory)、只读存储器 ROM(Read Only Memory)和各种输入/输出单元的单芯片微机系统。

单片微型计算机也称为单片机，目前已被广泛地应用于自动测量、智能仪表、工业控制及家用电器等各个方面。自从 1976 年 9 月美国 Intel 公司的 MCS-48 单片机问世以来，世界各大厂商已相继研制出了 60 多个系列、上千个品种的单片机产品。随着半导体集成技术的提高和发展，单片机正向着高主频、低功耗、低成本、多接口等方向发展。

1.1　AVR 系列单片机的特点

AVR 单片机是 Atmel 公司 1997 年推出的 RISC 单片机。RISC(精简指令系统计算机)是相对于 CISC(复杂指令系统计算机)而言的。RISC 并非只是简单地去减少指令，而是通过使计算机的结构更加简单合理而提高运算速度的。RISC 优先选取使用频率最高的简单指令，避免复杂指令；并固定指令长度，减少指令格式和寻址方式的种类，从而缩短指令周期，提高运行速度。由于 AVR 采用了 RISC 的这种结构，使 AVR 系列的单片机都具备了 1MIPS/MHz(百万条指令每秒/兆赫兹)的高速处理能力。

AVR 单片机吸收了 DSP 双总线的特点，采用 Harvard 总线结构，因此单片机的程序存储器和数据存储器是分离的，并且可对具有相同地址的程序存储器和数据存储器进行独立的寻址。

在 AVR 单片机中，CPU 执行当前指令时取出将要执行的下一条指令放入指令寄存器中，从而可以避免传统 MCS51 系列单片机中多指令周期的出现。

传统的 MCS51 系列单片机所有的数据处理都是基于一个累加器的，因此累加器与程序存储器、数据存储器之间的数据交换就成了单片机的瓶颈；在 AVR 单片机中，寄存器由 32 个通用工作寄存器组成，并且任何一个寄存器都可以充当累加器，从而有效地避免了累加器的瓶颈效应，提高了系统的性能。

AVR 单片机具有良好的集成性能。AVR 系列的单片机都具备在线编程接口，其中的 Mega 系列还具备 JTAG 仿真和下载功能；都含有片内看门狗电路、片内程序 Flash、同步串行接口 SPI；多数 AVR 单片机还内嵌了 AD 转换器、EEPROM、模拟比较器、PWM 定时计数器等多种功能；AVR 单片机的 I/O 接口还具有很强的驱动能力，灌电流可直接驱动继电器、LED 等器件，从而省去驱动电路，节约系统成本。

AVR 单片机采用低功率、非挥发的 CMOS 工艺制造，除具有低功耗、高密度的特点外，还支持低电压的联机 Flash、EEPROM 写入功能。

AVR 单片机还支持 Basic、C 等高级语言编程。采用高级语言对单片机系统进行开发是单片机应用的发展趋势。对单片机用高级语言编程可很容易地实现系统的移植，并加快软件的开发过程。

AVR 单片机具有多个系列，包括 ATtiny、AT90 和 ATmega。每个系列又包括多个产品，它们在功能和存储器容量等方面有很大的不同，但基本结构和原理都类似，而且编程方法也相同。表 1.1 为 AVR 各类型单片机的功能表。

表 1.1 AVR 单片机功能表

	Flash(KB)	Vcc (min)	EEPROM(B)	SRAM(B)	SPI	Watchdog	Clock (MHz)	RTC	On chip Oscillator	Interrupts	External	UART	8-bit Timer	16-bit Timer	PWM	10-bit A/D Channels	Instructions	ISP	Self Program Memory	Analog Comparat
ATtiny11L	1	2.7				Y	2		Y	4	1		1				90	2		Y
ATtiny11	1	4.0				Y	6		Y	4	1		1				90	2		Y
ATtiny12	1	1.8	64			Y	1		Y1	5	1		1				90	Y		Y
ATtiny12L	1	2.7	64			Y	4		Y1	5	1		1				90	Y		Y
ATtiny12	1	4.0	64			Y	8		Y1	5	1		1				90	Y		Y
ATtiny15L	1	2.7	64			Y	1.6		Y1	8	1		2		1	4	90	Y		Y
ATtiny28	2	1.8				Y	1		Y1	5	2		1				90	N		
ATtiny28L	2	2.7				Y	4		Y1	5	2		1				90	N		
AT90S1200	1	2.7	64			Y	12		Y	3	1		1				89	Y		Y
AT90S2313	2	2.7	128	128		Y	10			10	2	1	1	1	1		120	Y		Y
AT90LS2323	2	2.7	128	128		Y	4			2	1		1				120	Y		
AT90S2323	2	4.0	128	128		Y	10			2	1		1				120	Y		
AT90LS2343	2	2.7	128	128		Y	4		Y	2	1		1				120	Y		
AT90S2343	2	4.0	128	128		Y	10		Y	2	1		1				120	Y		
AT90LS4433	4	2.7	256	128	1	Y	4			14	2	1	1	1	1	6	120	Y		Y
AT90S4433	4	4.0	256	128	1	Y	8			14	2	1	1	1	1	6	120	Y		Y
AT90S8515	8	2.7	512	512	1	Y	8			11	2	1	1	1	2		120	Y		Y
AT90LS8535	8	2.7	512	512	1	Y	4	Y		15	2	1	2	1	3	8	120	Y		Y
AT90S8535	8	4.0	512	512	1	Y	8	Y		15	2	1	2	1	3	8	120	Y		Y
ATmega161L	16	2.7	512	1 K	1	Y	4	Y		20	3	2	2	1	4		130	Y	Y	Y
ATmega161	16	4.0	512	1 K	1	Y	8	Y		20	3	2	2	1	4		130	Y	Y	Y
ATmega163L	16	2.7	512	1 K	1	Y	4	Y	Y1	17	2	1	2	1	3	8	130	Y	Y	Y
ATmega163	16	4.0	512	1 K	1	Y	8	Y	Y1	17	2	1	2	1	3	8	130	Y	Y	Y

续表

	Flash(KB)	Vcc (min)	EEPROM(B)	SRAM(B)	SPI	Watchdog	Clock (MHz)	RTC	On chip Oscillator	Interrupts	External Interrupts	UART	8-bit Timer	16-bit Timer	PWM	10-bit A/D Channels	Instructions	ISP	Self Program Memory	Analog Comparat
ATmega323L	32	2.7	1	2 K	1	Y	4	Y	Y1	19	3	1	2	1	4	8	130	Y	Y	Y
ATmega323	32	4.0	1	2 K	1	Y	8	Y	Y1	19	3	1	2	1	4	8	130	Y	Y	Y
ATmega103L	128	2.7	4	4 K	1	Y	4	Y		16	8	1	2	1	4	8	121	Y		Y
ATmega103	128	4.0	4	4 K	1	Y	6	Y		16	8	1	2	1	4	8	121	Y		Y
ATmega8	8	4.5	512	1 K	1	Y	16	Y	Y1	18	2	1	2	1	3	8	130	Y	Y	Y
ATmega8L	8	2.7	512	1 K	1	Y	8	Y	Y1	18	2	1	2	1	3	8	130	Y	Y	Y
ATmega128	11	4.5	4	4 K	1	Y	16	Y	Y1		8	2	2	2	6	8	133	Y	Y	Y
ATmega128L	11	2.7	4	4 K	1	Y	8	Y	Y1		8	2	2	2	6	8	133	Y	Y	Y
ATmega16	16	4.5	5.2	1 K	1	Y	16	Y	Y	20	2	1	2	1	3	8	130	Y	Y	Y
ATmega16L	16	2.7	512	1 K	1	Y	8	Y	Y	20	2	1	2	1	3	8	130	Y	Y	Y
ATmega64	64	4.5	2	4 K	1	Y	16	Y	Y1		8	2	2	2	6	8	133	Y	Y	Y
ATmega64L	64	2.7	2	4 K	1	Y	8	Y	Y1		8	2	2	2	6	8	133	Y	Y	Y

① 高精度(5%)可编程内部 RC 振荡器。

② 编程时需给 RESET 管脚提供 12V 信号。

1.2 AT90 系列单片机简介

AT90 系列单片机是带片内 Flash 的在线可编程 8 位微控制器。

AT90 系列单片机是 AVR 单片机中的中档产品，内部具有的 Flash 程序存储器的容量介于 ATtiny 和 ATmega 之间，同时还具备较丰富的 I/O 接口，是 AVR 单片机中性价比较高的一个系列。AT90 与 ATtiny 和 ATmega 系列的原理相同，只是在功能上略有差异。AT90 系列的单片机的特点如下：

- 采用 120 条指令的 CPU 内核，而 ATtiny 是 90 条，Atmega 是 130 条。
- 具有可在线编程的 Flash 存储器，可反复擦写 1000 次以上。
- 片内含有少量字节的 EEPROM，可以在线编程，也可在程序运行期间由程序写入，即便断电也不会丢失信息。
- 带有看门狗电路，能保证在系统出错时，程序有效复位。
- 具有较多的 I/O 接口，且每个 I/O 接口都有很强的驱动能力，能直接驱动继电器、LED 等器件。
- 带有片内的定时/计数器，除实现定时、计数功能之外，有些还具备比较匹配输出

和输入捕获功能。

- 多数单片机都含有同步串行接口 SPI 和异步串行接口 UART，便于实现系统与计算机或系统与系统间的通信。
- 部分单片机含有 AD 转换器，如 AT90S8535、AT90S4433，可直接对引脚输入的模拟电压信号进行转换。
- 部分产品具备 PWM(Pulse Width Modulation)输出功能，此输出既可作为控制信号使用，也可经滤波作为 D/A 转换器使用。
- 具有较多的中断源，便于程序的设计开发。
- 具有较为丰富的开发语言。除了汇编语言之外，还可以采用 Basic、C 等高级语言进行系统开发。
- 具有 DIP、TQFP 和 PLCC 多种封装形式。

第 2 章　AT90LS8535 单片机的基础知识

AT90 系列单片机是由 Atmel 公司开发的一系列基于 RISC 结构的 8 位微控制器。其中的 AT90LS8535 是这一系列中功能最全的产品，它不仅包含了通用的 I/O 功能，而且还具备一个 10 位的 AD 转换器，一个全双工的串行接口，一个具有计时功能的实时时钟和 3 个可编程的多功能定时/计数器。本章将通过对 AT90LS8535 单片机结构的叙述，来说明 AVR 单片机的系统结构，为后面章节的学习打好基础。

2.1　AT90LS8535 单片机的总体结构

AT90LS8535 单片机的总体结构如图 2.1 所示。各主要组成部分介绍如下：

- 算术逻辑运算单元 ALU。算术逻辑单元 ALU 能对数据进行算术运算和逻辑运算，是单片机的核心。ALU 只能运算，运算的操作数必须预先放在 32 个通用寄存器中，而运算的结果可以送到任何一个可寻址单元中。ALU 在完成运算后还会将本次运算的特征信息送到状态寄存器中。
- 内部程序存储器。AT90LS8535 内部含有 8 KB 的程序 Flash。Flash 中可以存放程序，也可存放变量的初始化值和表格数据。AT90LS8535 的程序存储器可通过编程逻辑单元的控制多次改写。
- 内部数据存储器。AT90LS8535 具有 512B 的内部数据存储单元，可用于存放变量数据。
- 定时及控制单元。这部分确定了系统的时钟和复位逻辑。AT90LS8535 既可以采用芯片内 RC 振荡器提供的时钟信号，也可以采用外部时钟电路提供的时钟信号。AT90LS8535 的复位逻辑支持 3 种复位源——上电复位、外部复位和看门狗复位。
- 并行 I/O 接口。AT90LS8535 具有 4 个 8 位的 I/O 接口(PA、PB、PC、PD)。各 I/O 接口可设置成输入口，也可设置成输出口。当设置成输出口时，输出缓冲器能吸收约 20 mA 的电流，可直接驱动 LED、继电器等部件。复位后，各 I/O 接口均为高阻状态。
- 中断单元。AT90LS8535 有 16 个中断源，每个中断源都有独立的中断程序入口地址，所有的中断事件都有自己的使能位，可以根据使用需要屏蔽或使能。
- 模数转换器。AT90LS8535 具有 8 通道 10 位精度的逐次逼近式模数转换器。AD 转换部分通过 8 通道模拟转换器与 A 口的 8 个输入端相连，从而实现 8 通道的 AD 转换功能。
- 定时/计数器。AT90LS8535 有两个可分频的 8 位的定时/计数器器和一个可分频的 16 位定时/计数器。其中的 1 个 8 位定时/计数器具有比较输出和 PWM 功能；而

16 位的定时/计数器不仅具有比较输出和 PWM 功能，还可实现输入捕获。

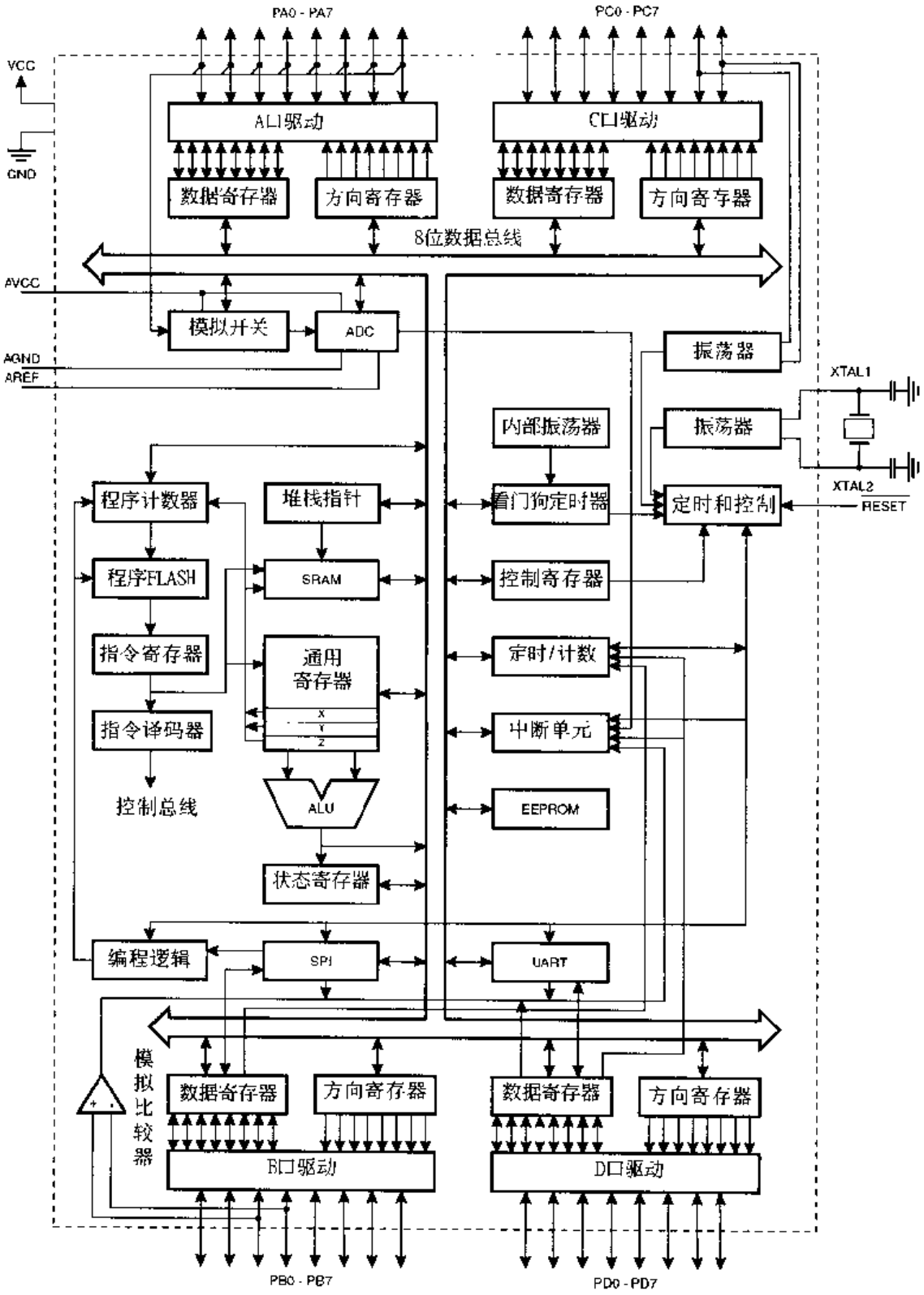


图 2.1 AT90LS8535 单片机的总体结构

2.1.1　AT90LS8535 单片机的中央处理器

AT90LS8535 的中央处理器包括 32 个通用寄存器、一个算术逻辑运算单元和相应的控制逻辑部分。

32 个 8 位的通用寄存器可实现单周期访问，其中的 6 个寄存器可以构成 3 个 16 位的用于数据寻址的指针 X、Y、Z，以提高 ALU 的地址运算能力。

AT90LS8535 同所有的 AVR 单片机一样，都采用了程序和数据总线分离的 HARVARD 结构。这种结构可以保证 ALU 在执行当前指令的同时去取下一条指令，从而使绝大多数指令在一个时钟内完成。

AT90LS8535 单片机的中央处理器结构如图 2.2 所示。

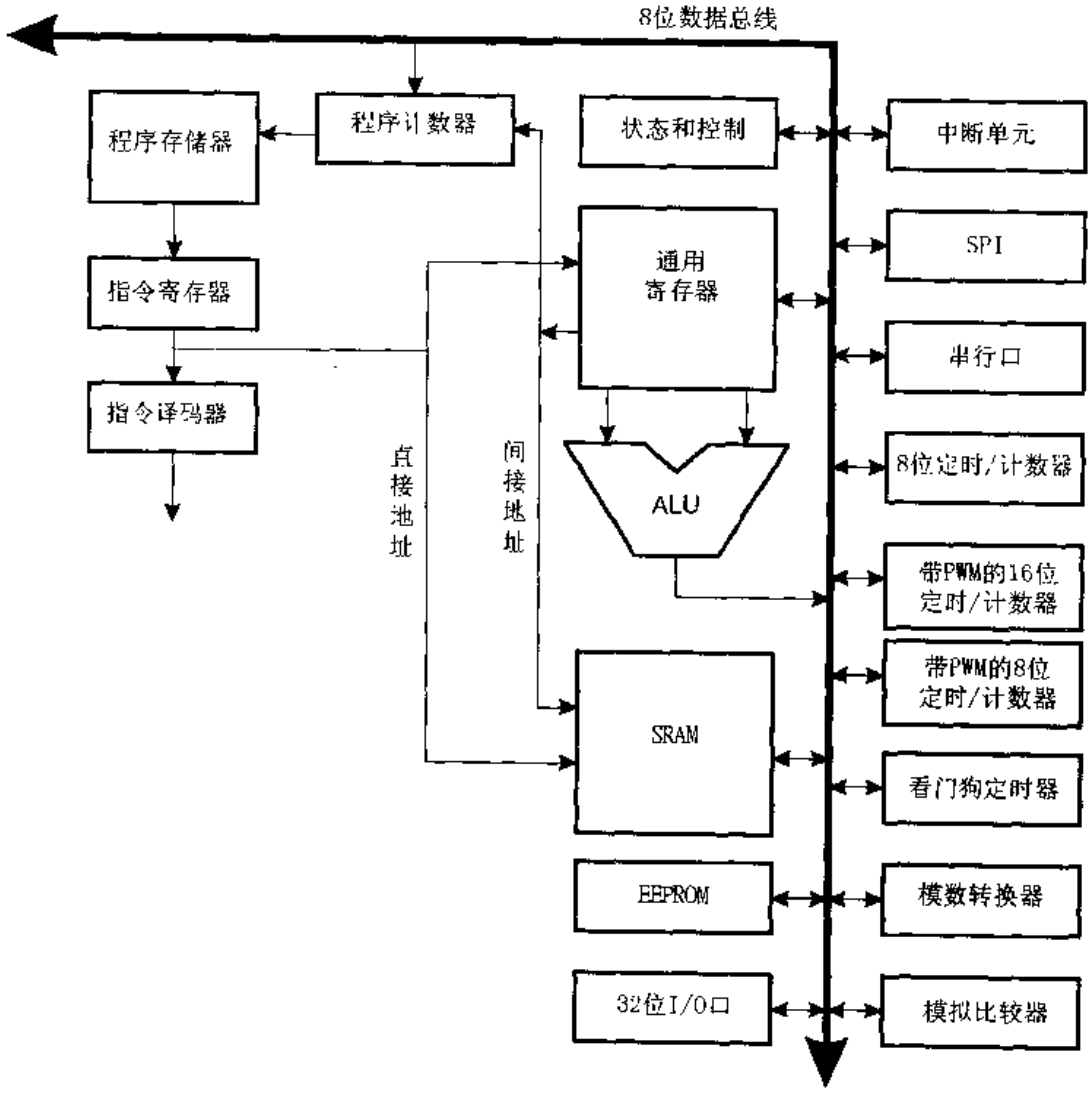


图 2.2　AT90LS8535 单片机的中央处理器结构

- 512 B 的 SRAM 用于存放程序中的变量，并且在中断或子程序调用时生成堆栈，以便中断或子程序返回后，系统能返回到复位前的状态。

- 中断单元由各中断控制器和状态寄存器的中断使能位构成，每个中断源都有一个独立的中断矢量入口地址。各中断源在相应使能位为 1，且全局中断使能位为 1 的时候，产生中断请求。
- ALU 是算术和逻辑运算的核心。它以 32 个通用寄存器中的值为操作数，并把运算的结果送到预定的寄存器或存储器中。

2.1.2 AT90LS8535 单片机的存储器组织

AT90LS8535 单片机的存储器包括程序 Flash、片内 EEPROM 和片内数据存储器 3 个部分，这 3 个部分都具有独立的寻址机构和寻址方式。因此对于重复的地址段，系统通过不同的寻址指令来加以区分。

AT90LS8535 具有 8KB 的程序 FLASH，它可通过在线编程的方式擦除或改写。Flash 中的某些特定的单元是保留出来供系统使用的，如$000 是系统复位后的入口地址，$001～$010 是中断服务程序的入口地址，因此用户所编写的程序不应进入这些区域。

AT90LS8535 含有 512B 的片内 EEPROM，用户程序可通过对其相关的控制寄存器、地址寄存器和数据寄存器的访问实现对 EEPROM 的读写。EEPROM 可实现至少 100 000 次的无故障擦写循环。

AT90LS8535 的片内数据存储器实际上包括通用寄存器、I/O 寄存器和 SRAM 这 3 个部分，但这 3 个部分是统一编址的，因此片内的数据存储器总共就占有 608 个地址。其中，通用寄存器的地址为$0000～$001F，I/O 寄存器的地址为$0020～$005F，内部 SRAM 的地址为$0060～$025F。AT90LS8535 的 I/O 寄存器分布如表 2.1 所示。

表 2.1 AT90LS8535 的 I/O 寄存器分布表

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$3F ($5F)	SREG	I	T	H	S	V	N	Z	C
$3E ($5E)	SPH	-	-	-	-	-	-	SP9	SP8
$3D ($5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
$3C ($5C)	Reserved								
$3B ($5B)	GIMSK	INT1	INT0	-	-	-	-	-	-
$3A ($5A)	GIFR	INTF1	INTF0						
$39 ($59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	-	TOIE0
$38 ($58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	-	TOV0
$37 ($57)	Reserved								
$36 ($56)	Reserved								
$35 ($55)	MCUCR	-	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00
$34 ($54)	MCUSR	-	-	-	-	-	-	EXTRF	PORF
$33 ($53)	TCCR0	-	-	-	-	-	CS02	CS01	CS00
$32 ($52)	TCNT0	Timer/Counter0 (8 Bits)							
$31 ($51)	Reserved								
$30 ($50)	Reserved								
$2F ($4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	PWM11	PWM10
$2E ($4E)	TCCR1B	ICNC1	ICES1	-	-	CTC1	CS12	CS11	CS10
$2D ($4D)	TCNT1H	Timer/Counter1 – Counter Register High Byte							
$2C ($4C)	TCNT1L	Timer/Counter1 – Counter Register Low Byte							
$2B ($4B)	OCR1AH	Timer/Counter1 – Output Compare Register A High Byte							
$2A ($4A)	OCR1AL	Timer/Counter1 – Output Compare Register A Low Byte							
$29 ($49)	OCR1BH	Timer/Counter1 – Output Compare Register B High Byte							
$28 ($48)	OCR1BL	Timer/Counter1 – Output Compare Register B Low Byte							
$27 ($47)	ICR1H	Timer/Counter1 – Input Capture Register High Byte							
$26 ($46)	ICR1L	Timer/Counter1 – Input Capture Register Low Byte							
$25 ($45)	TCCR2	-	PWM2	COM21	COM20	CTC2	CS22	CS21	CS20

续表

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$24 ($44)	TCNT2	Timer/Counter2 (8 Bits)							
$23 ($43)	OCR2	Timer/Counter2 Output Compare Register							
$22 ($42)	ASSR	-	-	-	-	AS2	TCN2UB	OCR2UB	TCR2UB
$21 ($41)	WDTCR	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0
$20 ($40)	Reserved								
$1F ($3F)	EEARH								EEAR8
$1E ($3E)	EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
$1D ($3D)	EEDR	EEPROM Data Register							
$1C ($3C)	EECR	-	-	-	-	EERIE	EEMWE	EEWE	EERE
$1B ($3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
$1A ($3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
$19 ($39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
$18 ($38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
$17 ($37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
$16 ($36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
$15 ($35)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
$14 ($34)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
$13 ($33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
$12 ($32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
$11 ($31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
$10 ($30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
$0F ($2F)	SPDR	SPI Data Register							
$0E ($2E)	SPSR	SPIF	WCOL	-	-	-	-	-	-
$0D ($2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
$0C ($2C)	UDR	UART I/O Data Register							
$0B ($2B)	USR	RXC	TXC	UDRE	FE	OR	-	-	-
$0A ($2A)	UCR	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8
$09 ($29)	UBRR	UART Baud Rate Register							
$08 ($28)	ACSR	ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
$07 ($27)	ADMUX	-	-	-	-	-	MUX2	MUX1	MUX0
$06 ($26)	ADCSR	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0
$05 ($25)	ADCH	-	-	-	-	-	-	ADC9	ADC8
$04 ($24)	ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
$03 ($23)	Reserved								
$02 ($22)	Reserved								
$01 ($21)	Reserved								
$00 ($20)	Reserved								

2.1.3　AT90LS8535 单片机的 I/O 接口

AT90LS8535 单片机具有 4 个 8 位的并行 I/O 接口，分别为 PA、PB、PC 和 PD。实际上它们都被归入专用的 I/O 寄存器之列，并可像访问通用寄存器一样进行寻址。在作为普通的数字 I/O 使用时，这 32 个 I/O 引脚的结构和特性是相同的，它们都可以通过相关控制寄存器定义为输入或输出口。

AT90LS8535 的 I/O 控制寄存器包括数据方向寄存器(DDRA～DDRD)、数据寄存器(PORTA～PORTD)和输入引脚“寄存器”(PINA～PIND)。输入引脚“寄存器”并不是真正意义上的寄存器，它只是一个可寻址的地址，该地址允许单片机直接读取 I/O 引脚上的逻辑值。

I/O 接口的口线逻辑电路如图 2.3 所示(以 PA 口为例)。整个电路由两个数据寄存器、3 个三态数据缓存器和相应的输入输出元件构成。

当 A 口的方向寄存器 DDA 的第 n 位置为 1 时，PAn 定义为输出。由电路的逻辑原理可知：DDAn 的输出 Q 为 1 时，上拉 MOS 管截止，PORTAn 的三态门导通，从而使 PAn 的电平取决于 PORTAn 的输出 Q。

当 A 口的方向寄存器 DDA 的第 n 位置为 0 时，PAn 定义为输入。当 PORTAn 也为 0

时，上拉 MOS 管截止，PAn 三态输入；当 PORTAn 为 1 时，上拉 MOS 管导通，PAn 带上拉电阻作为输入。

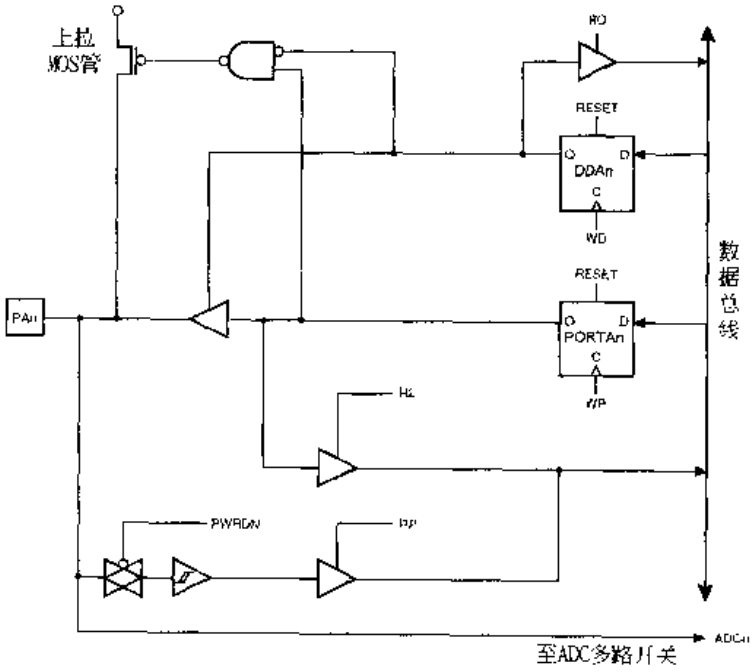


图 2.3 I/O 接口原理图

系统复位后，寄存器 DDRA～DDRD 和寄存器 PORTA～PORTD 均被清 0，即 PA～PD 口都被置为高阻状态。

所有的 I/O 接口在作为输出使用时，其输出缓冲器都可以吸收 20 mA 的电流，可以直接驱动继电器和 LED 等器件。

AT90LS8535 的 32 根 I/O 接口线中有 26 根具有第二功能，关于这些口线的第二功能将在以后的几章中介绍。

2.1.4 AT90LS8535 单片机的内部资源

AT90LS8535 是 AT90 系列单片机中功能最强、性能最好的一款。它具有一个通用串行接口、3 个定时/计数器、16 个中断源、8 通道 AD 转换等多种功能。对于一些简单的应用系统，甚至不需要外围辅助电路就能实现。

1. 通用串行接口

AT90LS8535 单片机内部具有一个全双工的串行接口。数据的发送和接收都是通过对 UART I/O 数据寄存器——UDR 的访问来实现的。UDR 占用内部寄存器的一个地址 $0C($2C)，但在单片机内部，UDR 是由两个物理上分离的寄存器——发送缓冲器和接收缓冲器组成的，因此对 UDR 的写入是针对发送缓冲器的，而对 UDR 的读取则是针对接收缓冲器的。UDR 的这种结构可以保证系统可以同时进行发送和接收操作。

2. 定时/计数器

AT90LS8535 共有 3 个可编程的定时/计数器，其中 T/C0 和 T/C2 是两个 8 位的定时/计数器，T/C1 是一个 16 位的定时/计数器。AT90LS8535 的定时/计数器除具备定时和计数功能外，有些还具有输入捕获、比较匹配和脉宽调制等功能。

AT90LS8535 的 T/C0 和 T/C1 可从同一个 10 位的预分频定时器中取得分频时钟，而 T/C2 则具备独立的预分频器，可以采用外部异步时钟作为定时器的驱动信号。

定时/计数器可以在规定的触发条件发生时向 CPU 发出中断申请，从而实现预期的定时控制功能。由于 AVR 单片机对不同的定时/计数器分配了不同的中断向量，因此，AVR 系统的定时中断处理程序不需要对中断源进行查询。

T/C 控制寄存器(TCCR0～TCCR2)用来负责定时/计数器的启动和停止，而 T/C 的中断控制则由中断屏蔽寄存器 TMISK 和中断标志寄存器 TIFR 管理。

3. 中断源

AT90LS8535 的中断系统支持 16 个独立的中断源，每个中断源都有一个独立的中断矢量和一个独立的中断使能位。各中断源定义及中断矢量如表 2.2 所示。

表 2.2　复位与中断向量定义

向量号	程序地址	中断源	中断定义
1	$000	RESET	硬件引脚上电复位或看门狗复位
2	$001	INT0	外部中断 0
3	$002	INT1	外部中断 1
4	$003	TIMER2 COMP	定时/计数器 2 比较匹配
5	$004	TIMER2 OVF	定时/计数器 2 溢出
6	$005	TIMER1 CAPT	定时/计数器 1 捕获事件
7	$006	TIMER1 COMPA	定时/计数器 1 比较匹配 A
8	$007	TIMER1 COMPB	定时/计数器 1 比较匹配 B
9	$008	TIMER1 OVF	定时/计数器 1 溢出
10	$009	TIMER0 OVF	定时/计数器 0 溢出
11	$00A	SPI，STC	SPI 串行传输完成
12	$00B	UART，RX	UART 接收完成
13	$00C	UART，UDRE	UART 数据寄存器空
14	$00D	UART，TX	UART 发送完成
15	$00E	ADC	ADC 转换结束
16	$00F	EE_RDY	EEPROM 准备好
17	$010	ANA_COMP	模拟比较器

AT90LS8535 具有 2 个中断屏蔽寄存器——通用中断屏蔽寄存器 GMISK 和定时/计数中断屏蔽寄存器。任何一个有效中断产生后，全局中断使能自动清 0，因此用户程序如需中断嵌套，则必须在中断服务程序中重新将全局中断使能位置 1。

4. 8通道AD转换

AT90LS8535是具有8通道10位精度的逐次逼近式A/D转换器。ADC通过一个8通道的模数转换器同PA口相连，并从PA口的引脚中获得需转换的模拟电压。ADC的原理图如图2.4所示。

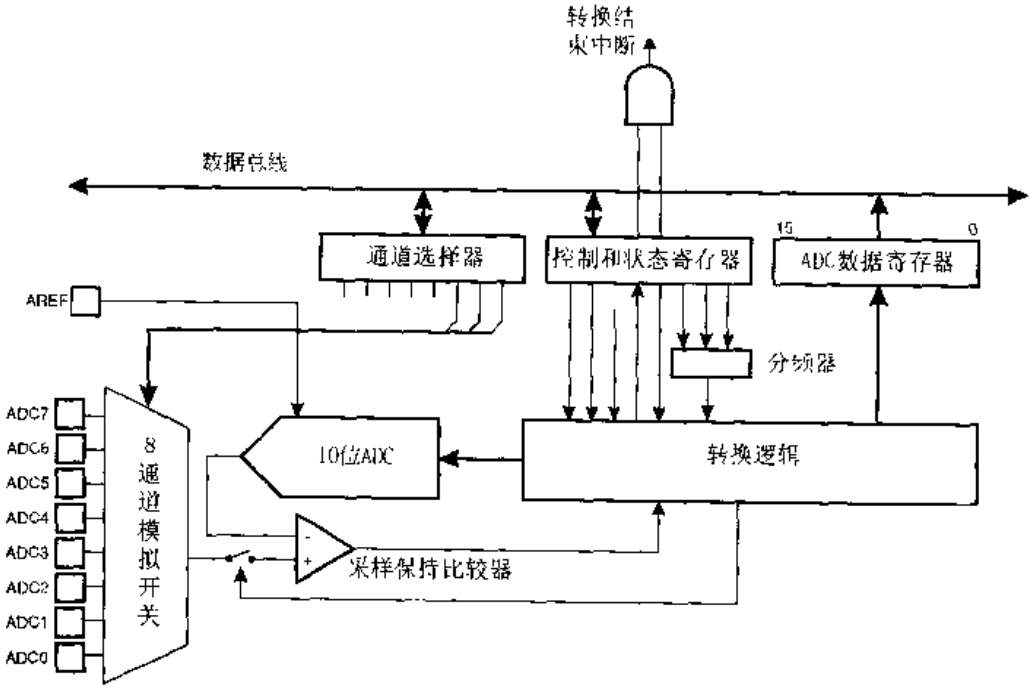


图2.4 ADC原理图

ADC具有两个模拟电压输入端AVcc和AGND以及一个外部参考电源输入端AREF。其中AVcc与Vcc的电压差应在0.3 V以内，AGND应与数字地GND相连，AREF应介于2 V~AVcc之间。

2.1.5 AT90LS8535单片机的时钟电路

单片机的时钟用于产生工作所需的时序，AT90LS8535单片机可以采用两种时钟信号，分别如图2.5和图2.6所示。

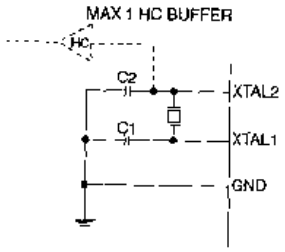


图2.5 晶振电路

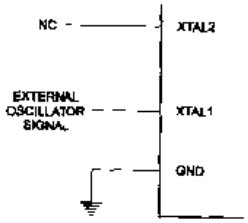


图2.6 外接时钟电路

如图2.5所示，AT90LS8535的芯片内有一个高增益的反相放大器，通过在其输入端

XTAL1 和输出端 XTAL2 跨接晶体振荡器和微调电容可以产生单片机所需的时序。

AT90LS8535 还可以采用外部时钟信号，如图 2.6 所示，外部时钟信号由 XTAL1 输入，而 XTAL2 端悬空。

2.1.6　AT90LS8535 单片机的系统复位

复位的主要功能就是把程序计数器 PC 初始化为$000，从而使单片机重新开始执行程序。AT90LS8535 具有 3 个复位源，如图 2.7 所示。

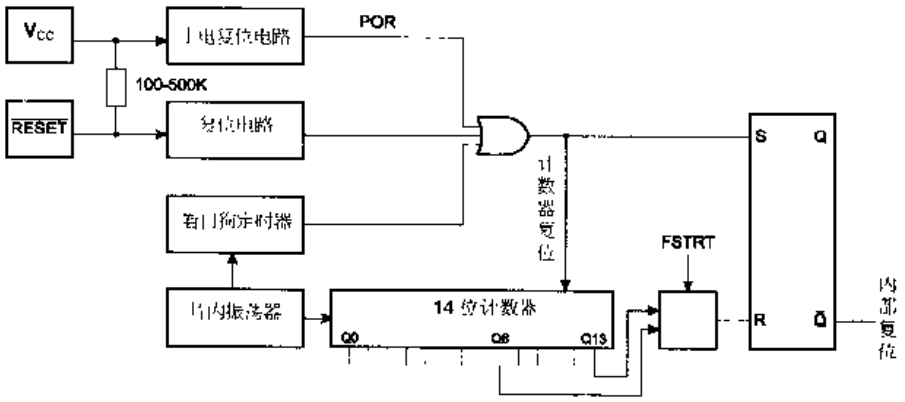


图 2.7　AT90LS8535 的复位逻辑

1. 上电复位

当单片机的 V_{CC} 和 GND 之间的电压低于上电复位门限电压 V_{POT} 时，单片机复位。上电复位(Power On Reset)用于确保单片机在上电时的可靠复位。如图 2.8 所示，一个由看门狗时钟驱动的内部定时器可以保证系统 V_{CC} 在到达门限电压 V_{POT} 一定时间后才启动。

如果在给单片机加 V_{CC} 电压的同时，保持 RESET 引脚为低电平，则可延长复位周期，如图 2.9 所示。

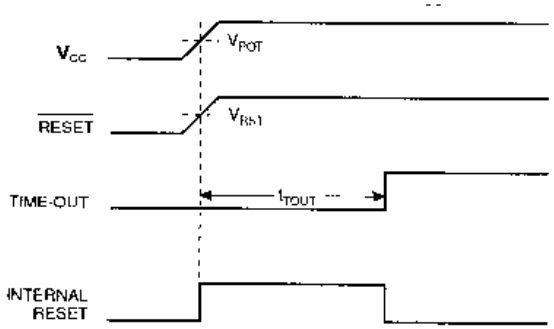


图 2.8　RESET 引脚和 VCC 相连时，单片机的复位时序

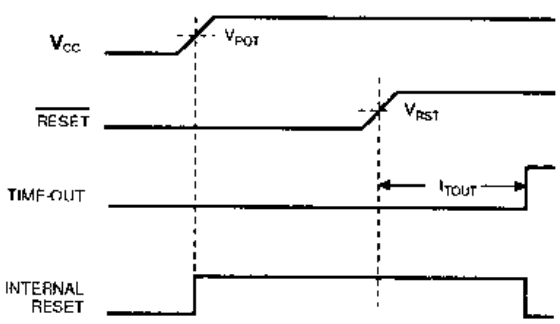


图 2.9　RESET 引脚由外部控制时，单片机的复位时序

2. 外部复位

当在单片机的 RESET 引脚上施加超过 50 ns 低电平时，单片机复位。RESET 引脚上的

复位脉冲必须大于 50 ns，较短的复位脉冲信号将不能保证单片机的可靠复位。外加复位信号由低电平到达复位门限电压 V_{RST} 的上升沿时，单片机延时一个 t_{TOUT} 周期后复位，如图 2.10 所示。

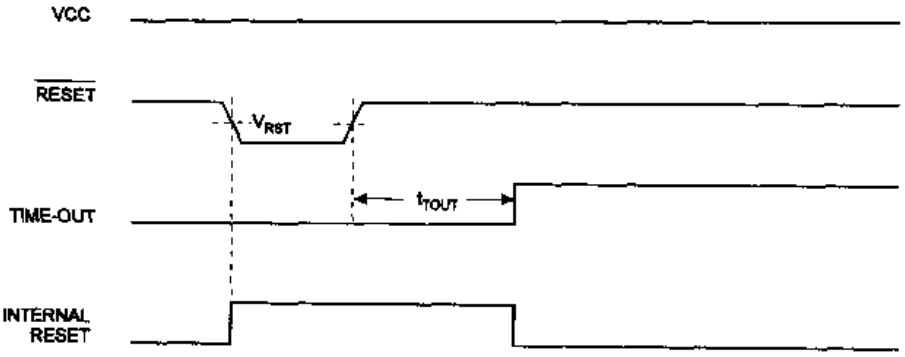


图 2.10 外部复位时序图

3. 看门狗复位

当看门狗功能为开启状态，并且看门狗定时器超时时，单片机复位。看门狗计数器溢出时，将产生一个晶振周期的复位脉冲，在该脉冲的下降沿，单片机延时一个 t_{TOUT} 周期后复位，如图 2.11 所示。

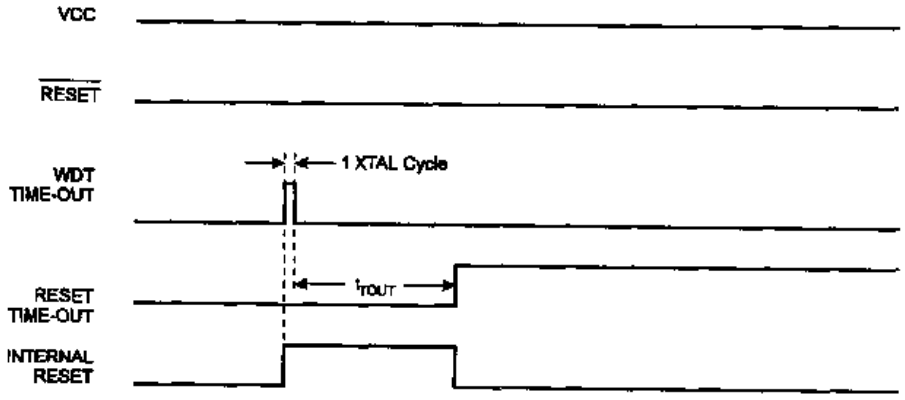


图 2.11 看门狗复位时序图

2.1.7 AT90LS8535 单片机的节电方式

AT90LS8535 单片机具有 3 种节电工作方式：掉电模式、空闲模式和休眠模式。

- 掉电模式：当单片机处于掉电模式时，只有复位和外部中断可以使单片机恢复到正常工作状态。
- 空闲模式：为使单片机进入空闲模式，SM 位必须清 0，且程序必须执行 SLEEP 指令。当系统发生外部中断、定时器溢出中断和看门狗复位时，单片机返回正常工作模式。
- 休眠模式：为使单片机进入休眠模式，SE 位必须置 1，且程序必须执行 SLEEP

指令。系统发生的任何一种中断和复位将使单片机恢复到正常工作状态。

2.1.8　AT90LS8535 单片机的芯片引脚

AT90LS8535 有 PDIP、PLCC、TQFP 和 MLF 这 4 种封装型式，引脚分配如图 2.12 所示。

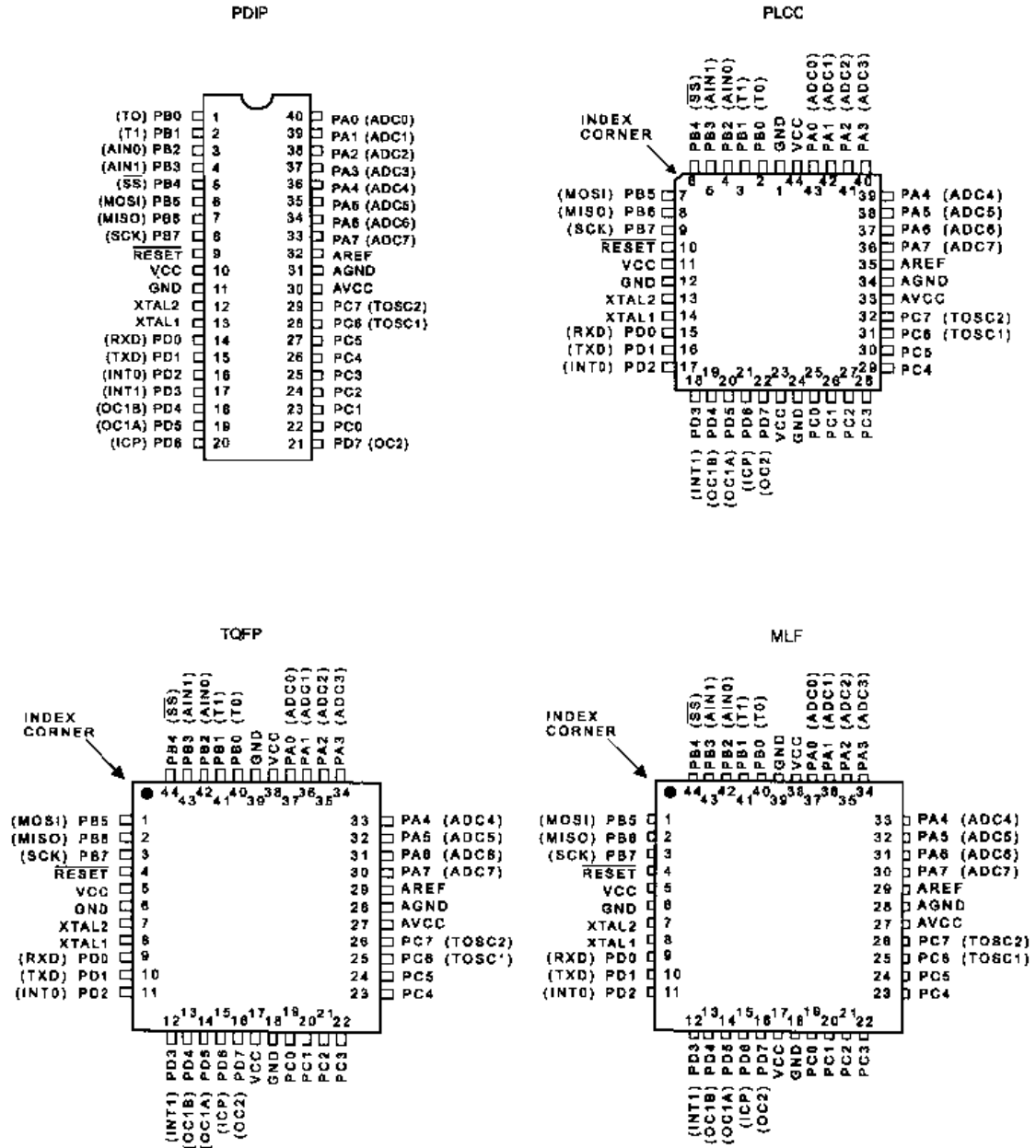


图 2.12　AT90LS8535 的引脚图

各引脚功能如下：

- VCC、GND：电源输入端。
- AVCC、AGND：模数转换器的电源输入端。
- AREF：模数转换器的参考输入端。
- XTAL1：振荡器的输入端。

- XTAL2：振荡器的输出端。
- RESET：复位信号输入端。
- PA～PD：双向I/O接口。当PA～PD作为第二功能使用时，请参看以后各章。

2.2 AT90LS8535单片机的指令系统

指令是指计算机所能执行的命令，它是由生产厂商定义的。一台计算机所能执行的指令集合就是它的指令集。由于指令系统并没有通用性，不同类型的计算机都有不同类型的指令集，因此采用低级语言开发的程序也没有通用性，不能直接移植。

机器语言就是计算机指令的二进制代码，它可以直接在计算机中运行；而汇编语言则是机器语言的符号表示形式，它只是机器语言的助记符集合，因此由汇编语言编写的程序必须经专门的工具程序转化为相应的二进制代码后才能供计算机执行。

AVR单片机的指令系统属于RISC结构的精简指令系统。AT90LS8535的RISC指令集共有118条指令，分为以下4个大类：

- 算术运算和逻辑运算指令(25条)
- 转移指令(31条)
- 数据传送指令(31条)
- 位和位测试指令(31条)

表2.3所示为AT90LS8535的指令系统表。

表2.3 AT90LS8535的指令系统表

助记符	操作数	说明	操作	标志	周期
算术运算和逻辑运算指令					
ADD	Rd, Rr	相加	Rd ←Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	带进位的相加	Rd ←Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl, K	字加立即数	Rdh:Rdl ←Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	相减	Rd ←Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	减立即数	Rd ←Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	带进位相减	Rd ←Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	带进位减立即数	Rd ←Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl, K	字减立即数	Rdh:Rdl ←Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	与	Rd ←Rd · Rr	Z,N,V	1
ANDI	Rd, K	与立即数	Rd ←Rd·K	Z,N,V	1
OR	Rd, Rr	或	Rd ←Rd v Rr	Z,N,V	1
ORI	Rd, K	或立即数	Rd ←Rd v K	Z,N,V	1
EOR	Rd, Rr	异或	Rd ←Rd ⊕ Rr	Z,N,V	1
COM	Rd	取反	Rd ←$FF-Rd	Z,C,N,V	1
NEG	Rd	取补	Rd ←$00-Rd	Z,C,N,V,H	1

续表

助记符	操作数	说明	操作	标志	周期
SBR	Rd, K	置寄存器的某位	Rd ← Rd v K	Z,N,V	1
CBR	Rd, K	清寄存器的某位	Rd ← Rd ·($FF　K)	Z,N,V	1
INC	Rd	加 1	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	减 1	Rd ← Rd -1	Z,N,V	1
TST	Rd	测试	Rd ← Rd ⊕ Rd	Z,N,V	1
CLR	Rd	清寄存器	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	置寄存器	Rd ← $FF	无	1
转移指令					
RJMP	k	相对转移	PC ← PC + k + 1	None	2
IJMP		间接转移	PC ← Z	None	2
RCALL	k	相对调用	PC ← PC + k + 1	None	3
ICALL		间接调用	PC ← Z	None	3
RET		程序返回	PC ← STACK	None	4
RETI		中断返回	PC ← STACK	I	4
CPSE	Rd, Rr	相等跳转	if (Rd = Rr) PC ← PC + 2 或 3	None	1/2/3
CP	Rd, Rr	比较	Rd - Rr	Z,N,V,C,H	1
CPC	Rd, Rr	带进位的比较	Rd - Rr - C	Z,N,V,C,H	1
CPI	Rd, K	与立即数比较	Rd - K	Z,N,V,C,H	1
SBRC	Rr, b	寄存器某位为 0 时跳转	if (Rr(b) = 0) PC ← PC + 2 或 3	None	1/2/3
SBRS	Rr, b	寄存器某位为 1 时跳转	if (Rr(b) = 1) PC ← PC + 2 或 3	None	1/2/3
SBIC	P, b	I/O 寄存器某位为 0 时跳转	if (P(b) = 0) PC ← PC + 2 或 3	None	1/2/3
SBIS	P, b	I/O 寄存器某位为 1 时跳转	if (P(b) = 1) PC ← PC + 2 或 3	None	1/2/3
BRBS	s, k	状态寄存器置 1 跳转	if (SREG(s) = 1) then PC ←PC + k + 1	None	1/2
BRBC	s, k	状态寄存器为 0 跳转	if (SREG(s) = 0) then PC ←PC + k + 1	None	1/2
BREQ	k	相等跳转	if (Z = 1) then PC ← PC + k + 1	None	1/2
BRNE	k	不等跳转	if (Z = 0) then PC ← PC + k + 1	None	1/2
BRCS	k	C=1 跳转	if (C = 1) then PC ← PC + k + 1	None	1/2
BRCC	k	C=0 跳转	if (C = 0) then PC ← PC + k + 1	None	1/2
BRSH	k	大于等于跳转	if (C = 0) then PC ← PC + k + 1	None	1/2
BRLO	k	小于跳转	if (C = 1) then PC ← PC + k + 1	None	1/2
BRMI	k	负跳转	if (N = 1) then PC ← PC + k + 1	None	1/2
BRPL	k	正跳转	if (N = 0) then PC ← PC + k + 1	None	1/2

续表

助记符	操作数	说明	操作	标志	周期
BRGE	k	大于等于跳转	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1/2
BRLT	k	小于跳转	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1/2
BRHS	k	H 为 1 跳转	if (H = 1) then PC ← PC + k + 1	None	1/2
BRHC	k	H 为 0 跳转	if (H = 0) then PC ← PC + k + 1	None	1/2
BRTS	k	T 为 1 跳转	if (T = 1) then PC ← PC + k + 1	None	1/2
BRTC	k	T 为 0 跳转	if (T = 0) then PC ← PC + k + 1	None	1/2
BRVS	k	V 为 1 跳转	if (V = 1) then PC ← PC + k + 1	None	1/2
BRVC	k	V 为 0 跳转	if (V = 0) then PC ← PC + k + 1	None	1/2
BRIE	k	I 为 1 跳转	if (I = 1) then PC ← PC + k + 1	None	1/2
BRID	k	I 为 0 跳转	if (I = 0) then PC ← PC + k + 1	None	1/2
数据传送指令					
MOV	Rd, Rr	寄存器间传送	Rd ← Rr	None	1
LDI	Rd, K	导入立即数	Rd ← K	None	1
LD	Rd, X	导入 X 间接地址数	Rd ← (X)	None	2
LD	Rd, X+	导入 X+1 的间接地址数	Rd ← (X), X ← X + 1	None	2
LD	Rd, -X	导入 X-1 的间接地址数	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	导入 Y 间接地址数	Rd ← (Y)	None	2
LD	Rd, Y+	导入 Y+1 的间接地址数	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, -Y	导入 Y-1 的间接地址数	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd, Y+q	导入 Y+q 的变址地址数	Rd ← (Y + q)	None	2
LD	Rd, Z	导入 Z 间接地址数	Rd ← (Z)	None	2
LD	Rd, Z+	导入 Z+1 的间接地址数	Rd ← (Z), Z ← Z + 1	None	2
LD	Rd, -Z	导入 Z-1 的间接地址数	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	导入 Z+q 的变址地址数	Rd ← (Z + q)	None	2
LDS	Rd, k	从 SRAM 中导入	Rd ← (k)	None	2
ST	X, Rr	以 X 为间接地址存数	(X)=← Rr	None	2
ST	X+, Rr	以 X-1 为间接地址存数	(X)=← Rr, X ← X + 1	None	2
ST	-X, Rr	以 X+1 为间接地址存数	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	以 Y 为间接地址存数	(Y) ← Rr	None	2
ST	Y+, Rr	以 Y+1 为间接地址存数	(Y) ← Rr, Y ← Y + 1	None	2
ST	-Y, Rr	以 Y-1 为间接地址存数	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q, Rr	以 Y+q 为变址地址存数	(Y + q) ← Rr	None	2
ST	Z, Rr	以 Z 为间接地址存数	(Z) ← Rr	None	2

续表

助记符	操作数	说明	操作	标志	周期
ST	Z+, Rr	以 Z+1 为间接地址存数	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	以 Z−1 为间接地址存数	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q, Rr	以 Z+q 为变址地址存数	(Z + q) ← Rr	None	2
STS	k, Rr	存数据于 SRAM	(k) ← Rr	None	2
LPM		由程序区导入	R0 ← (Z)	None	3
IN	Rd, P	从 I/O 输入	Rd ← P	None	1
OUT	P, Rr	输出置 I/O	P ← Rr	None	1
PUSH	Rr	压栈	STACK ← Rr	None	2
POP	Rd	出栈	Rd ← STACK	None	2
位和位测试指令					
SBI	P, b	置 I/O 中的位	I/O(P,b) ← 1	None	2
CBI	P, b	清 I/O 中的位	I/O(P,b) ← 0	None	2
LSL	Rd	左移	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V	1
LSR	Rd	右移	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V	1
ROL	Rd	带进位位的循环左移	Rd(0)←C, Rd(n+1)←Rd(n), C←Rd(7)	Z,C,N,V	1
ROR	Rd	带进位位的循环右移	Rd(7)←C, Rd(n)←Rd(n+1), C←Rd(0)	Z,C,N,V	1
ASR	Rd	算术右移	Rd(n) ← Rd(n+1), n = 0..6	Z,C,N,V	1
SWAP	Rd	高低 4 位交换	Rd(3..0) ← Rd(7..4), Rd(7..4) ← Rd(3..0)	None	1
BSET	s	置 SREG 位	SREG(s) ← 1	SREG(s)	1
BCLR	s	清 SREG 位	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	指定寄存器的位送 T	T ← Rr(b)	T	1
BLD	Rd, b	T 送指定寄存器的位	Rd(b) ← T	None	1
SEC		置 C	C ← 1	C	1
CLC		清 C	C ← 0	C	1
SEN		置 N	N ← 1	N	1
CLN		清 N	N ← 0	N	1
SEZ		置 Z	Z ← 1	Z	1
CLZ		清 Z	Z ← 0	Z	1
SEI		置 I	I ← 1	I	1
CLI		清 I	I← 0	I	1
SES		置 S	S ← 1	S	1
CLS		清 S	S ← 0	S	1
SEV		置 V	V ← 1	V	1

续表

助记符	操作数	说明	操作	标志	周期
CLV		清V	V ← 0	V	1
SET		置T	T ← 1	T	1
CLT		清T	T ← 0	T	1
SEH		置H	H ← 1	H	1
CLH		清H	H ← 0	H	1
NOP		空操作		None	1
SLEEP		休眠	(参见休眠功能的详细说明)	None	1
WDR		看门狗复位	(参见看门狗复位的详细说明)	None	1

2.2.1 汇编指令格式

指令格式是指指令的表示方式。通常一条指令是由操作码和操作数两部分组成的：操作码用来指明指令的具体操作过程，而操作数则用来指明指令的操作对象。由表2.3可见，指令系统中的操作数可以是一个具体的常数(立即数)，也可以是一个表示数据所在位置的地址或符号。

AT90LS8535单片机的汇编程序中的每条程序通常都包含标号、指令助记符或伪指令、操作数和注释4个部分。每条程序的输入形式为：

标号：指令(伪指令) 操作数1，操作数2，…；注释

如：BEGIN：ADD R20，R22；寄存器值相加

START：.equ lable＝$0001 ；初始化

注意：标号和注释可以省略，操作数的个数需和指令操作的对象个数相对应。

2.2.2 寻址方式

寻址是寻找操作数所在地址单元的过程。由于绝大多数指令在执行时都需要使用操作数，且操作数一般又不以立即数的形式在程序中给定，因此计算机必须采用一定的方式(寻址)从内存单元中取得操作数。

不同的计算机具有不同的寻址方式，同一计算机对于不同指令的需要也会有不同的寻址方式。通常来说，寻址能力越强，寻址方式越多的计算机具有更好的操作性，但其指令系统势必更加复杂。因此在设置寻址方式时，必须综合考虑各种因素。

AVR单片机具有以下几种寻址方式，分别介绍如下。

1. 直接单寄存器寻址

直接单寄存器寻址是通过指令中一些特定字段的内容来定位操作数的。操作数的地址由指令中的后5位确定，因此这种寻址方式共可指定工作寄存器的32个内存单元。

2. 直接双寄存器寻址

直接双寄存器寻址也是通过指令中的一些特定字段的内容来定位操作数的。所不同的是，它有两组 5 位的寄存器地址单元，因此可寻址 32 个工作寄存器中两个内存单元。

3. 直接 I/O 寻址

直接 I/O 寻址通过指令中的 D0～D5 位制定 I/O 位置，同时它还可寻址一个寄存器单元作为目的地址或源地址。

4. 直接数据寻址

数据直接寻址用于对 RAM 的操作，该寻址方式的指令中，一个字为操作数在 RAM 中的地址，一个字为指令(其中包括目的或源寄存器地址)。

5. 带偏移量的间接数据寻址

带偏移量的间接数据寻址以 Y 或 Z 寄存器中的变址和指令中的偏移量地址之和来决定操作数在 RAM 中的地址。指令中还包含有目的或源寄存器的地址。

6. 间接数据寻址

指令中给出的寄存器内容为操作数的地址，而不是操作数本身，即以寄存器为地址指针的寻址方式就是间接数据寻址。

7. 预先减 1 的间接数据寻址

这种寻址方式同间接数据寻址类似，所不同的是：操作数的地址为指定寄存器的内容值减 1，同时再把这个减 1 后的值赋给指定寄存器。

8. 预先加 1 的间接数据寻址

这种寻址方式同间接数据寻址类似，所不同的是：操作数的地址为指定寄存器的内容值加 1，同时再把这个加 1 后的值赋给指定寄存器。

9. 程序存储器常量寻址

程序存储器常量寻址是访问程序存储器中常量的直接寻址方式，常量的地址由 Z 寄存器的内容确定。

10. 程序存储器间接寻址

以 Z 寄存器的内容为地址继续执行程序的寻址方式就是程序存储器的间接寻址方式。这种寻址方式的指令执行后，Z 寄存器的内容会代替 PC 值。

11. 程序存储器相关寻址

以当前 PC 值和指令中所包含的相关地址 K 值之和为地址继续执行程序的寻址方式就是程序存储器的相关寻址。指令执行后，PC 值与 K 值相加，并把二者之和送至 PC。

2.2.3 伪指令

伪指令是汇编器用的指令，它用于确定程序在存储器中的位置，并进行一定的初始化。AVR 单片机的伪指令主要有以下几种：

1. BYTE 伪指令

BYTE 伪指令可为变量预留一段内存空间，BYTE 伪指令前应加有标号以定位被预留空间的位置。BYTE 伪指令还应有一个表示被预留字节数的参数。

例如：var1：.BYTE 1
table：.BYTE tab_size

2. CSEG 伪指令

CSEG 伪指令用于定义代码段的起始地址。在一个源程序中，可以多次使用 CSEG，但经汇编后，所有的代码段都被连接成一个代码段。

例如：.CSEG
const：.DW 2

3. DB 伪指令

DB 伪指令的功能是将一个字节表存放到从标号地址开始的 EEPROM 或程序存储器中。DB 伪指令须有标号，且表项中的数据必须用逗号分隔。

例如：consts：.DB 0, 255, 0b01010101, -128, 0xaa

4. DEF 伪指令

DEF 伪指令允许使用符号代替寄存器地址。一个预定义的符号可以在程序中用以指定某个寄存器，同一个寄存器可以使用多个符号表示。

例如：.DEF temp=R16
.DEF ior=R0

5. DEVICE 伪指令

用户可通过 DEVICE 伪指令通知编译器按芯片的执行环境来编译程序。如果程序中没有 DEVICE 伪指令，则编译器默认芯片可以执行所有的指令。

例如：.DEVICE AT90S1200

6. DSEG 伪指令

DSEG 伪指令用于定义数据段的起始地址。在一个源程序中，可以多次使用 DSEG，但经汇编后，所有的数据段都被连接成一个数据段。

例如：.DSEG
var1：.BYTE 1
table：.BYTE tab_size

7. DW 伪指令

DW 伪指令的功能是将一个字节表存放到从标号地址开始的 EEPROM 或程序存储器中。DW 伪指令也须有标号，且表项中的数据必须用逗号分隔。与 DB 伪指令不同的是，DW 伪指令用于定义 16 位的地址表。

例如：varlis：.DW　0,0xffff,0b1001110001010101,-32768,65535

8. ENDMACRO 伪指令

ENDMACRO 伪指令用于定义 MACRO 定义的结束。该指令没有任何参数。

例如：.ENDMACRO

9. EQU 伪指令

EQU 伪指令给一个标号赋予一个值。被 EQU 伪指令定义的标号可在程序中作为一个常数使用。

例如：.EQU　io_offset = 0x23

10. ESEG 伪指令

ESEG 伪指令用于定义 EEPROM 段的起始地址。在一个源程序中，可以多次使用 ESEG，但经汇编后，所有的 EEPROM 段都被连接成一个 EEPROM 段。

例如：.ESEG

　　eevar：.DW　0xff0f

11. EXIT 伪指令

EXIT 伪指令用于通知编译器结束编译。通常，编译器会在文件结束时结束编译。如文件中有 EXIT 伪指令，则编译器从 INCLUDE 伪指令所在行继续编译。

例如：.EXIT

12. INCLUDE 伪指令

INCLUDE 指令通知编译器从指定的文件读取要汇编的指令行。INCLUDE 伪指令可以嵌套使用。

例如：.INCLUDE　"iodefs.asm"

13. LIST 伪指令

LIST 伪指令通知编译器打开列表文件。LIST 伪指令通常和 NOLIST 伪指令一起使用，以便有选择地生成汇编源文件中的列表文件。

例如：.LIST

14. LISTMAC 伪指令

LISTMAC 伪指令通知编译器，当宏被调用时，宏的展开式须在列表文件中显示出来。

例如：.LISTMAC

15. MACRO 伪指令

MACRO 伪指令通知编译器一个宏的开始。MACRO 伪指令以宏的名称为参数。定义宏之后，如后面的程序中出现了宏名，则宏程序在该位置执行一次。

例如：.MACRO MACX

16. NOLIST 伪指令

同 LIST 伪指令相反，NOLIST 伪指令通知编译器关闭列表文件。

例如：.NOLIST

17. ORG 伪指令

ORG 伪指令用于设置位置计数器的绝对值。ORG 伪指令可对代码段、数据段和 EEPROM 的计数器分别进行设置。

例如：.DSEG

```
.ORG   0x67
variable: .BYTE   1
.ESEG
.ORG   0x20
eevar:  .DW   0xfeff
```

18. SET 伪指令

SET 伪指令给一个标号赋予一个值。被 SET 伪指令定义的标号值可在程序中改变。

例如：.SET io_offset = 0x23

2.2.4 指令类型及数据操作方式

AT90LS8535 的指令系统可分为：算术运算和逻辑运算指令、转移指令、数据传送指令以及位和位测试指令。

1. 算术运算和逻辑运算指令

AT90LS8535 的算术运算和逻辑运算指令共 25 条，其中包括加、减、与、或、异或、取反、取补等。

- 不带进位的加法(ADD)
  说明：把两个寄存器相加，结果放至目标寄存器 Rd。
  格式：ADD Rd，Rr($0 \le d \le 31$，$0 \le r \le 31$)
- 带进位的加法(ADC)
  说明：把两个寄存器和进位 C 相加，结果放至目标寄存器 Rd。
  格式：ADC Rd，Rr ($0 \le d \le 31$，$0 \le r \le 31$)
- 字加立即数(ADIW)
  说明：把一个立即数(0～63)和一对寄存器相加，结果放至一对寄存器中。
  格式：ADIW Rd，K ($d \in \{24, 26, 28, 30\}$，$0 \le K \le 63$)

- 加 1 指令(INC)
  说明：使寄存器 Rd 加 1，结果放至目标寄存器 Rd 中。
  格式：INC Rd (0 ≤ d ≤ 31)
- 不带进位的减法(SUB)
  说明：把两个寄存器相减，结果放至目的寄存器 Rd 中。
  格式：SUB Rd，Rr (0 ≤ d ≤ 31，　0 ≤ r ≤ 31)
- 带进位的减法(SBC)
  说明：把两个寄存器相减，并把结果减去进位 C，结果放至目的寄存器 Rd 中。
  格式：SBC Rd，Rr (0 ≤ d ≤ 31，　0 ≤ r ≤ 31)
- 不带进位减立即数(SUBI)
  说明：把寄存器和常数相减，结果放至目的寄存器 Rd 中。
  格式：SUBI Rd，K (16 ≤ d ≤ 31，　0 ≤ K ≤ 255)
- 带进位减立即数(SBCI)
  说明：把寄存器和立即数相减，并把结果减去进位 C，结果放至目的寄存器 Rd 中。
  格式：SBCI Rd，K (16 ≤ d ≤ 31，　0 ≤ K ≤ 255)
- 字减立即数(SBIW)
  说明：把一对寄存器与立即数(0～63)相减，结果放至一对寄存器中。
  格式：SBIW Rd，K (d ∈ {24，26，28，30}，　0 ≤ K ≤ 63)
- 减 1 指令(DEC)
  说明：把寄存器 Rd 减 1，结果放至目的寄存器 Rd 中。
  格式：DEC Rd (0 ≤ d ≤ 31)
- 寄存器相与(AND)
  说明：把两个寄存器的内容相与，结果放至目的寄存器 Rd 中。
  格式：AND Rd，Rr (0 ≤ d ≤ 31，　0 ≤ r ≤ 31)
- 与立即数(ANDI)
  说明：把寄存器和立即数相与，结果放至目的寄存器 Rd 中。
  格式：ANDI Rd，K (16 ≤ d ≤ 31，　0 ≤ K ≤ 255)
- 寄存器相或(OR)
  说明：把两个寄存器相或，结果放至目的寄存器 Rd 中。
  格式：OR Rd，Rr (0 ≤ d ≤ 31，　0 ≤ r ≤ 31)
- 或立即数(ORI)
  说明：把寄存器和立即数相或，结果放至目的寄存器 Rd 中。
  格式：ORI Rd，K (16 ≤ d ≤ 31，　0 ≤ K ≤ 255)
- 寄存器异或(EOR)
  说明：两个寄存器相异或，结果放至目的寄存器 Rd 中。
  格式：EOR Rd，Rr (0 ≤ d ≤ 31，　0 ≤ r ≤ 31　PC ← PC + 1)
- 寄存器取反(COM)
  说明：对寄存器的内容取反，结果放至目的寄存器 Rd 中。
  格式：COM Rd (0 ≤ d ≤ 31)

- 寄存器取补(NEG)
  说明：对寄存器的内容求补，结果放至目的寄存器 Rd 中。
  格式：NEG Rd (0 ≤ d ≤ 31)
- 置寄存器的位(SBR)
  说明：把寄存器的指定位置 1，结果放至目的寄存器 Rd 中。
  格式：SBR Rd，K (16 ≤ d ≤ 31， 0 ≤ K ≤ 255)
- 清寄存器的位(CBR)
  说明：清除寄存器中的指定位，结果放至目的寄存器 Rd 中。
  格式：CBR Rd，K (16 ≤ d ≤ 31， 0 ≤ K ≤ 255)
- 测试(TST)
  说明：判断寄存器等于 0 或小于 0，结果放至目的寄存器 Rd 中。
  格式：TST Rd (0 ≤ d ≤ 31)
- 清寄存器(CLR)
  说明：使寄存器的所有位为 0。
  格式：CLR Rd (0≤d≤31)
- 置寄存器(SER)
  说明：使寄存器的所有位为 1。
  格式：SER Rd (16≤d≤31)

2. 转移指令

转移指令用来实现程序的分支控制。AT90LS8535 单片机的转移指令包括无条件转移指令和条件转移指令：无条件转移指令是没有测试条件的转移指令，条件转移指令是需要对条件进行测试的转移指令。

- 相对转移(RJMP)
  说明：程序转移到 PC-2K~PC+2K 地址范围内的某个地址。
  语法：RJMP k (-2K ≤ k < 2K)
- 间接转移(IJMP)
  说明：程序转移至 Z 寄存器所指向的地址。
  语法：IJMP
- 相对调用(RCALL)
  说明：在 PC+1-2K 至 PC+1+2K 地址范围围内调用子程序。
  语法：RCALL k (-2K ≤ k < 2K)
- 间接调用(ICALL)
  说明：调用 Z 寄存器所指向的子程序。
  语法：ICALL
- 返回指令(RET 和 RETI)
  说明：RET 由子程序返回，RETI 由中断返回。
  语法：RET(RETI)
- 相等跳转(CPSE)

说明：若两个寄存器 Rd 和 Rr 相等，则程序跳行执行。

语法：CPSE Rd，Rr (0 ≤ d ≤ 31，　0 ≤ r ≤ 31)

- 比较指令(CP、CPC 和 CPI)

  说明：CP 对两个寄存器进行比较，CPC 是 CP 的带进位的比较，CPI 是寄存器与立即数比较。

  语法：CP/CPC/CPI Rd，Rr (0 ≤ d ≤ 31，　0 ≤ r ≤ 31)

- 寄存器指定位的位跳转(SBRC 和 SBRS)

  说明：当寄存器某位为 0 时，SBRC 指令使程序跳行执行(SBRS 指令为 1)。

  语法：SBRC/SBRS Rr，b (0 ≤ r ≤ 31，　0 ≤ b ≤ 7)

- I/O 寄存器指定位的位跳转(SBIC 和 SBIS)

  说明：当 I/O 寄存器某位为 0 时，SBIC 指令使程序跳行执行(SBIS 指令为 1)。

  语法：SBIC/SBIS A，b (0 ≤ A ≤ 31，　0 ≤ b ≤ 7)

- 状态寄存器指定位的位跳转(BRBS 和 BRBC)

  说明：当状态寄存器某位为 0 时，BRBC 指令使程序跳转至 PC+1+K(BRBS 指令为 1)。

  语法：BRBS/BRBC s，k (0 ≤ s ≤ 7，−64 ≤ k ≤ +63)

- Z 标志位的位跳转(BREQ 和 BRNE)

  说明：当 Z＝0 时，BREQ 指令使程序跳转至 PC+1+K(BRNE 指令为 1)。

  语法：BREQ/BRNE k (−64 ≤ k ≤ +63)

- C 标志位的位跳转(BRCS 和 BRCC)

  说明：当 C＝0 时，BRCC 指令使程序跳转至 PC+1+K(BRCS 指令为 1)。

  语法：BRCS/BRCC k (−64 ≤ k ≤ +63)

- C 标志位的位跳转(BRSH 和 BRLO)

  说明：这两条指令的功能同 BRCS 和 BRCC。

  语法：BRSH/BRLO k (−64 ≤ k ≤ +63)

- S 标志位的位跳转(BRGE 和 BRLT)

  说明：当 S＝0 时，BRGE 指令使程序跳转至 PC+1+K(BRLT 指令为 1)。

  语法：BRGE/BRLT k (−64 ≤ k ≤ +63)

- H 标志位的位跳转(BRHS 和 BRHC)

  说明：当 H＝0 时，BRHC 指令使程序跳转至 PC+1+K(BRHS 指令为 1)。

  语法：BRHS/BRHC k (−64 ≤ k ≤ +63)

- T 标志位的位跳转(BRTS 和 BRTC)

  说明：当 T＝0 时，BRTC 指令使程序跳转至 PC+1+K(BRTS 指令为 1)。

  语法：BRTS/BRTC k (−64 ≤k ≤ +63)

- V 标志位的位跳转(BRVS 和 BRVC)

  说明：当 V＝0 时，BRVC 指令使程序跳转至 PC+1+K(BRVS 指令为 1)。

  语法：BRVS/BRVC k (−64≤ k ≤ +63)

- 中断标志跳转(BRIE 和 BRID)

说明：当I=0时，BRID指令使程序跳转至PC+1+K(BRIE指令为1)。

语法：BRIE/BRID k (-64 ≤ k ≤ +63)

3. 数据传送指令

传送指令有多种形式，它们的目的是实现数据的复制，并保证不破坏地址中的数据。

- 寄存器间的数据传递(MOV)

  说明：将一个寄存器的内容复制到另一个寄存器。

  语法：MOV Rd，Rr (0 ≤ d ≤ 31，0 ≤ r ≤ 31)

- 间接地址数导入指令(LD)

  说明：以X、Y、Z寄存器的值或寄存器的值减1、加1值为间接地址导入数据。

  语法：LD Rd，X/Y/Z (0 ≤ d ≤ 31)

  LD Rd，X/Y/Z+(0 ≤ d ≤ 31)

  LD Rd，-X/Y/Z(0 ≤ d ≤ 31)

- 立即数导入(LDI)

  说明：将一个8位的立即数复制至寄存器。

  语法：LDI Rd，K (16 ≤ d ≤ 31，0 ≤ K ≤ 255)

- 变址地址数导入指令(LDD)

  说明：以Y+q或Z+q为变址地址导入数据。

  语法：LDD Rd，Y/Z+q

- SRAM中数的导入指令(LDS)

  说明：把SRAM中1个自己的数据复制到寄存器。

  语法：LDS Rd，k (0 ≤ d ≤ 31，0 ≤ k ≤ 65535)

- 导出数据至间接地址(ST)

  说明：将寄存器数据复制至以X/Y/Z寄存器的值或其减1、加1值为间接地址的内存单元。

  语法：ST X/Y/Z，Rr (0 ≤ r ≤ 31)

  ST X/Y/Z+，Rr (0 ≤ r ≤ 31)

  ST -X/Y/Z，Rr (0 ≤ r ≤ 31)

- 导出数据至变址地址(STD)

  说明：将寄存器数据复制至以Y+q或Z+q为变址地址的内存单元。

  语法：STD Y/Z+q，Rr (0 ≤ r ≤ 31，0 ≤ q ≤ 63)

- 导出数据至SRAM(STS)

  说明：将寄存器的数据复制至SRAM中。

  语法：STS k，Rr (0 ≤ r ≤ 31，0 ≤ k ≤ 65535)

- 由程序存储器导入数据的指令(LPM)

  说明：将Z寄存器指向的一个字节复制到R0寄存器。

  语法：LPM

- I/O指令(IN和OUT)

  说明：实现I/O接口的数据和寄存器间的数据传递。

语法：IN Rd，P
　　OUT P，Rd

- 压栈、出栈指令(PUSH 和 POP)
  说明：将数据压入堆栈或将其从堆栈中弹出。
  语法：PUSH Rr
  　　POP Rd

4. 位和位测试指令

- 高低位交换指令(SWAP)
  说明：将寄存器中的高 4 位和低 4 位互换。
  语法：SWAP Rd ($0 \le d \le 31$)
- 状态寄存器的位操作指令(BSET 和 BCLR)
  说明：BSET 指令将状态寄存器指定位置位，BCLR 指令的作用与其相反。
  语法：BSET/BCLR s ($0 \le s \le 7$)
- T 的位操作指令(BST 和 BLD)
  说明：BST 指令将寄存器中的指定位复制到标志 T，BLD 指令的作用与其相反。
  语法：BST/BLD Rd，b ($0 \le d \le 31$，$0 \le b \le 7$)
- 标志位的置位指令(SEC、SEN、SEZ、SEI、SES、SEV、SEH)
  说明：分别用于标志位 C、N、Z、I、S、V 和 H 的置位操作。
  语法：SEC/SEN/SEZ/SEI/SES/SEV/SEH
- 清标志位指令(CLC、CLN、CLZ、CLI、CLS、CLV、CLH)
  说明：分别用于标志位 C、N、Z、I、S、V 和 H 的清 0 操作。
  语法：CLC/CLN/CLZ/CLI/CLS/CLV/CLH
- 空操作指令(NOP)
  说明：空操作，占有一个周期时间长度。
  语法：NOP
- 休眠指令(SLEEP)
  说明：使单片机进入休眠模式。
  语法：SLEEP
- 看门狗复位指令(WDR)
  说明：复位看门狗定时器。
  语法：WDR

2.3　应用程序设计

2.3.1　程序设计方法

程序设计是系统设计中的一个重要环节，它直接关系到系统的性能。好的程序不仅能

提高系统的稳定性和可靠性，而且还能降低系统的硬件成本，增加系统的功能。

那么，怎样才能编写出好的程序来呢？一般来说，系统的程序设计可按图 2.13 所示的步骤进行。

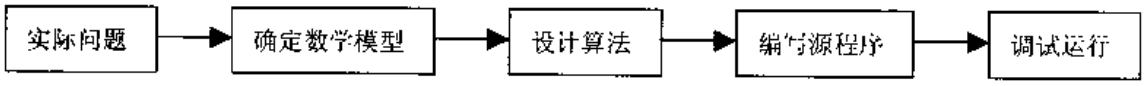


图 2.13 软件设计流程

作为程序设计人员，最关键的一步就是设计算法。算法一般是用流程图表示。如果算法正确，则再将它转换为任何一种编程语言(汇编语言或 C 语言)，这一过程也就是编写源程序的过程。单片机应用系统的软件设计过程也是类似的，下面是它的步骤。

1. 分析问题

根据实际要求，全面而系统地描述、分析问题，并找出解决问题的关键所在。这一步骤要求程序设计人员从整体上把握应用程序应具备的功能。

2. 设计算法

在设计算法时，一般可以采用以下 3 种方法：

- 自上而下
  这种方法先对整体进行设计，整体设计完成之后，再进入下一层的设计。采用这种设计方法可以在总体上考虑程序的功能，不致于顾此失彼、头重脚轻。
- 模块化
  这种方法将一个大的任务分解成若干个较小的模块，每个模块都有一定的功能，能完成系统的一部分任务。各模块可以独立地进行调试，便于逐步地完成复杂的任务。
- 功能细化
  将各个模块要解决的问题分解成单个的、可以很容易实现的单元，实现功能的逐步细化。

3. 编写源程序

这个过程其实就是把已经设计好的算法，用编程语言来表达的过程。

4. 调试运行

程序设计人员编写的程序最后需下载到应用系统，这是检验程序是否正确的一个重要步骤。通常来说，第一次编写的程序不可避免的会有一些问题，因此需要程序设计人员再重复以上几个步骤直至运行的程序达到预期的目的。

2.3.2 应用程序举例

1. I/O 接口数据输出

功能：循环输出 0～255 至 PB 口。

```
; ***************I/O 接口数据输出程序***************
I/O_test:
    ldi r21,$ff          ; 设循环初值
LOOP:
    inc r21              ; 加 1 运算
    ldi r20,$ff          ; 设置 PB 口为输出口
    out ddrb,r20
    out portb,r21        ; PB 口输出寄存器 r21 的值
    sjmp LOOP            ; 循环
```

2. EEPROM 读写程序

功能：通过将参数传递到指定的寄存器，并调用相关的子程序，实现 EEPROM 的随机读、随机写、顺序读和顺序写 4 种功能。

```
; ******************读写 EEPROM 程序**********************
; 随机存取
.def    EEdwr   =r16     ; EEPROM 的数据(写)
.def    EEawr   =r17     ; EEPROM 的地址(写)

EEWrite:                 ; 随机写 EEPROM 子程序
    sbic    EECR,EEWE    ; 如果 EEWE 不为 0
    rjmp    EEWrite      ; 等待 EEWE 为 0
    out EEAR,EEawr       ; 输出 EEPROM 地址
    out EEDR,EEdwr       ; 输出数据
    sbi EECR,EEWE        ; 设置 EEPROM 写锁存
    ret                  ; 返回

.def    EEdrd   =r0      ; EEPROM 的数据(读)
.def    EEard   =r16     ; EEPROM 的地址(读)

EERead:                  ; 随机读 EEPROM 子程序
    sbic    EECR,EEWE    ; 如果 EEWE 不为 0
    rjmp    EERead       ; 等待 EEWE 为 0
    out EEDR,EEard       ; 输出 EEPROM 地址
    sbi EECR,EERE        ; 设置 EEPROM 读锁存
    sbi EECR,EERE        ; 再次设置 EEPROM 读锁存
    in  EEdrd,EEDR       ; 取得数据
    ret                  ; 返回

; 顺序存取
.def    EEwtmp  =r0      ; 临时的 EEPROM 地址
.def    EEdwr_s =r16     ; 要写入 EEPROM 的数据

EEWrite_seq:             ; 顺序写 EEPROM 子程序
    sbic    EECR,EEWE    ; 如果 EEWE 不为 0
    rjmp    EEWrite_seq  ; 等待 EEWE 为 0
    in  EEwtmp,EEAR      ; 得到当前地址
    inc EEwtmp           ; 地址加 1
    out EEAR,EEwtmp      ; 输出 EEPROM 地址
    out EEDR,EEdwr_s     ; 输出数据
```

```
    sbi EECR,EEWE          ; 设置 EEPROM 写锁存
    ret                    ; 返回

.def    EErtmp  =r0        ; 临时的 EEPROM 地址
.def    EEdrd_s =r1        ; 要读入 EEPROM 的数据

EERead_seq:                ; 顺序读 EEPROM 子程序
    in  EErtmp,EEAR        ; 取得当前地址
    inc EErtmp             ; 地址加 1
    out EEAR,EErtmp        ; 输出 EEPROM 地址
    sbi EECR,EERE          ; 设置 EEPROM 读锁存
    sbi EECR,EERE          ; 再次设置 EEPROM 读锁存
    in  EEdrd_s,EEDR       ; 取得数据
    ret                    ; 返回
```

第 3 章　AT90LS8535 单片机的 C 编程

软件编程是开发单片机应用系统中的一个重要环节。好的程序不仅能扩充单片机的功能，而且还能提高系统的可靠性，便于系统的改进和功能扩充。

C 语言是一种结构化的编程语言，相对于其他编程语言而言，C 具有编译效率高、移植性好、可读性强等优点。因此，本章将重点介绍一下 AVR 系列单片机的 C 编译器以及编译文件的下载操作。

3.1　支持高级语言编程的 AVR 系列单片机

对于 AVR 系列单片机，既可以采用汇编语言编程，也可以采用 BASCOM 和 C 等高级语言编程。由于用汇编语言编写的系统程序的可读性和移植性都比较差，因此系统的调试工作比较困难，产品的开发周期也较长。为了提高系统应用程序的编写效率，缩短产品的开发周期，采用 Basic 和 C 等高级语言进行单片机应用程序设计已经成为软件开发的一个主流，它不仅会大大缩短开发周期，而且可以显著地增加软件的可读性，从而便于研制开发规模更大的系统。实践证明，采用高级语言进行单片机系统程序开发的效率要比使用汇编语言高几十倍。

AVR 系列单片机是基于精简指令集(RISC)结构的单片微控制器。RISC 结构是针对高级语言编程而设计的精简指令系统，因此同 MCS51 系列单片机相比，AVR 单片机更适宜用高级语言编程。目前，支持 AVR 单片机的高级语言有：BASCOM-AVR、ImageCraftICCAVR 和 IAR 等。

BASCOM-AVR 是 AVR 系列单片机的 Basic 开发平台。Basic 是一种简单易学的高级语言。但是 Basic 是逐行解释的，即 Basic 的每一行语句在执行时都要转换成机器语言，因此，系统应用程序的编程效率较低，执行时间较长。此外，Basic 的所有运算都是以浮点形式进行，变量也都以浮点形式存储，因而即便对于编译型的 Basic 语言，也不能有效地解决系统运行速度较低的问题。因此，BASCOM-AVR 只适用于程序简单且对实时性要求较低的场合。

C 语言是一种编译型的结构化程序设计语言，它具有简单而强大的处理功能，具有运行速度快、编译效率高、移植性好和可读性强等多种优点，并且可以实现对系统硬件的直接操作。C 语言支持自顶向下的结构化程序设计方法，并且支持模块化程序设计结构。因此，采用 C 语言开发单片机系统程序是单片机软件开发的首选。相对于其他编程语言，C 语言具有如下几个优点：

- 编译效率高，运行速度快。
- 系统维护、调试容易。

- 可读性强，移植性好。
- 结构简单，易于实现模块化。
- 具有强大的库函数，编程效率高。
- 对硬件的操作简单，不需了解指令的操作时序。

C 语言作为一种通用的开发语言，它不依赖于特定的单片机系统，因而可以很方便实现不同类型单片机系统间的替换、升级。

尽管C语言有上述众多优点，但其在实时性方面并不如精简的汇编语言，因此在一些对速度要求特别高的场合，可采用C和汇编混合编程的方法。

3.2 AVR 的 C 编译器

AVR 系列单片机作为一种新型的 RISC 单片机，从一开始就得到了各种高级语言的支持。由于C编译器产生的代码具有效率高、运行速度快等众多优点，因此在各种高级语言当中，C语言得到了最为广泛地应用。下面就几种C编译器进行简单地介绍。

1. ICC AVR C 编译器

ICC AVR 是 ImageCraft 公司针对 AVR 单片机而开发的一个 C 语言编译器，它具有以下几个特点：

- 支持不带 SRAM 的单片机器件。
- 带嵌入式的应用程序编译器。
- 带全局优化器。
- 支持在线编程(STK200/300)。

2. IAR C 编译器

IAR C 编译器嵌入在 IAR Embedded Workbench 中，IAR Embedded Workbench 的 C 编译器不仅适用于对 AVR 单片机的开发，而且还可开发 ARM、PIC 等器件，它的主要特点如下：

- 同时支持 C 和 C++两种编程方式
- 具备状态可视工具。
- 具备 IAR 应用程序编译器。

3. GNU C 编译器

GNU C 编译器是一个免费使用的编译器，因此厂家并不提供技术支持，并且只有下一代的 AVR Studio 才支持 GNU 编译器的输出格式。

由于 ICC AVR 的 C 编译器功能适合，使用简单方便，具有良好的技术支持且有很多功能模块可以调用，因此本书主要采用 ICC AVR 的 C 编译器来开发 AVR 系列单片机的应用系统。

3.3　ICC AVR 介绍

ICC AVR 是由 ImageCraft 公司开发的一种针对 AVR 系列单片机的开发工具。它采用符合 ANSI 标准的 C 语言来开发单片机程序，并具有以下几个方面的特点：

- ICC AVR 是一个运行于 Windows 9X/NT 的集成开发环境(IDE)，它包括单片机软件开发所需要的编辑器、工程管理器和 C 编译器。
- ICC AVR 采用了工程的组织形式。程序的所有源文件全部以工程的一个要素保存在工程中，并且程序的编辑和编译也都在这个环境中进行。工程管理器最后还能生成 HEX 格式的烧录文件和 COFF 格式的仿真文件。
- ICCAVR 还支持长文件名。

3.3.1　安装 ICC AVR

ImageCraft 公司的主页上提供 ICC AVR 程序的下载服务，但是这只是一个可以免费使用 30 天的未注册版。对于正式版，用户还须进行申请注册，得到了注册码以后，才可以无时间限制地使用它。

ICC AVR 的安装过程如下：

(1) 运行 SETUP.EXE 程序进行安装

打开安装程序所在的路径，双击文件 SETUP.EXE，进入安装画面，如图 3.1 所示。

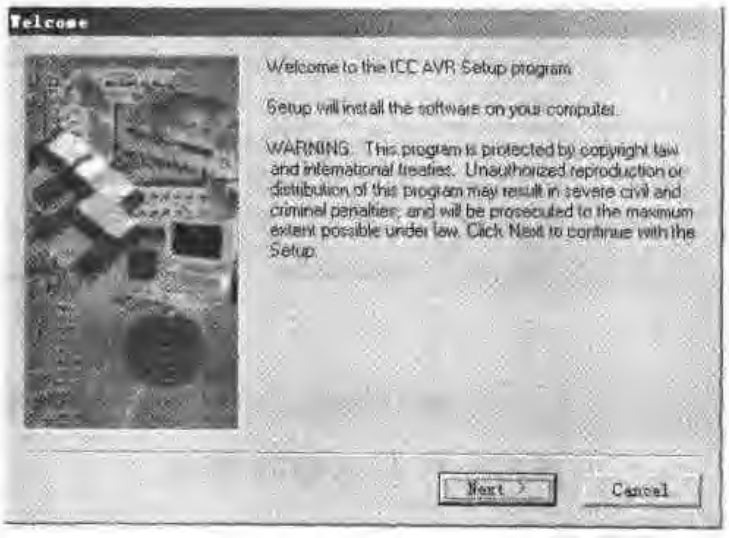


图 3.1　进入安装界面

(2) 选择安装路径

按照屏幕提示，选择程序的安装位置，如图 3.2 所示。

图 3.2 选择安装目录

(3) 完成安装

按程序的安装提示完成 ICC AVR 的安装后，就可以在程序菜单中找到 ImageCraft Development Tools 这个文件夹。打开文件夹，进入子菜单，就可以找到 ICC AVR 程序和帮助文件等。单击 ICC AVR 启动整个集成开发环境(IDE)，如图 3.3 所示。这个集成开发环境包括菜单栏、工具栏、编辑区、工程区和编译显示区 5 个部分。

图 3.3 ICC AVR 的集成开发环境

(4) 对软件进行注册

软件按使用时间是否超过 30 天，采取不同的注册方式。

- 对于使用期未超过 30 天的用户，按下述方法注册。
  - 启动 ICC AVR 集成开发环境。
  - 将附带的注册软盘插入软盘驱动器，单击 Help | Importing a License from a Floppy Disk 命令。

- ICCAVR 软件通过读取注册软盘中的信息，自动进行注册。当注册完成后提示重新启动 ICCAVR 以完成注册过程。

- 对于使用期已超过 30 天的用户，则按下述步骤进行注册。
  - 这时用户已经不能进入集成开发环境，系统在一进入时就出现注册对话框，这时选择 YES。
  - 当系统出现一个注册对话框时，将附带的注册软盘插入软盘驱动器，单击 Importing a icense from a Floppy Disk 按钮。
  - ICCAVR 软件通过读取注册软盘中的信息，自动进行注册。当注册完成后提示重新启动 ICCAVR 以完成注册过程。

3.3.2　设置 ICC AVR

应用系统的源程序在用 ICC AVR 编译和连接前，必须要对编译器的一些属性进行设置。通过单击 Project | Options...命令，可以打开系统的属性设置对话框。每个工程都有其自身特定的属性，这些特定的属性可以保存下来，并作为新建工程的默认属性。系统的属性设置对话框(Compiler Options)有 3 个选项卡，分别是：路径(Paths)、编译器(Compiler)和目标(Target)。本节将着重对书中例子所涉及到的属性进行介绍。

1. 路径属性

如图 3.4 所示，在 Paths 选项卡中，可以定义头部文件、库文件和输出文件的所在目录。头文件和库文件的目录在系统安装完毕时应分别设置为“安装盘:\安装目录\include”和“安装盘:\安装目录\lib”。系统的输出文件目录是编译器用于放置与工程相关文件和输出文件的地方，除非人为地定义输出文件的目录，否则它与工程所在目录一致。如果在 Output Directory 文本框中输入“objs”，则系统将以“c:\icctest\example\objs”作为输出文件的目录。

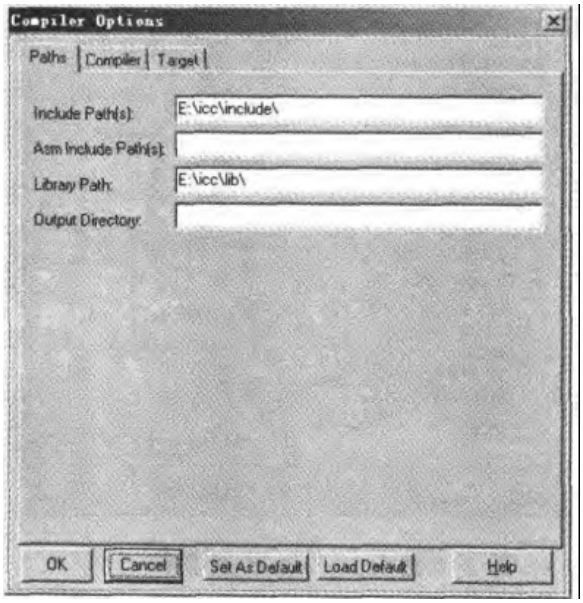


图 3.4　Paths 选项卡

2. 编译器属性

如图 3.5 所示，在对话框的编译器 Compiler 选项卡中，可以定义一些影响编译器操作的属性，例如在 Output Format 下拉列表框中，可以选择是输出 INTEL HEX 格式的烧录文件，还是 COFF 格式的仿真文件。可以通过选中 Accept Extensions(C++ Comments…)复选框来接受 C++格式的程序标注；可以通过选中 Strict ANSI C Checkings 复选框来进行严格的标准 C 语法检查。此外，在这个选项卡中还可以进行宏定义等操作。

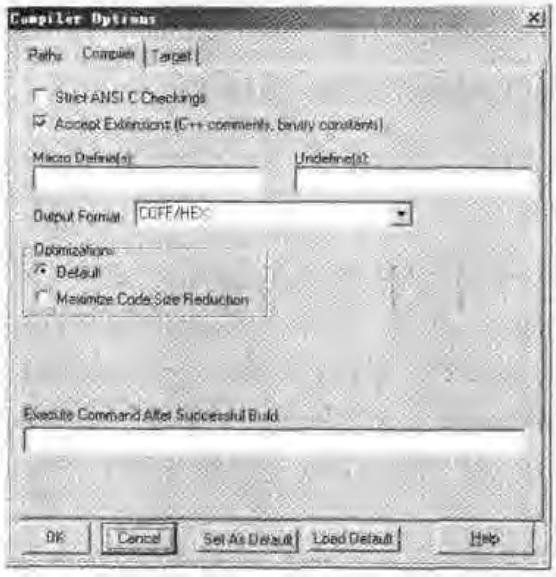


图 3.5 Compiler 选项卡

3. 目标属性

如图 3.6 所示，在对话框的目标 Target 选项卡中，可以定义适合目标器件的属性设置。

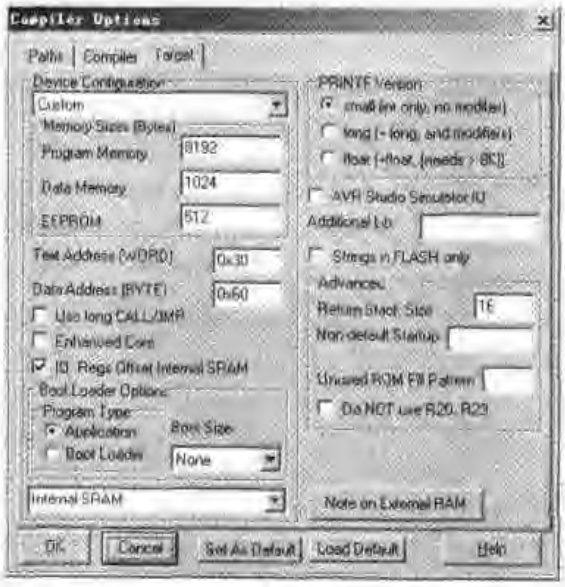


图 3.6 Target 选项卡

例如：目标器件为 AT90LS8535 时，为使编译器生成的输出文件在目标器件上正确运行，可在 Device Configuration 下拉列表框中，选择 AT90LS8535。为了适应新推出的 AVR

单片机产品，ICC AVR 还允许用户设定 Program Memory、Data Memory 和 EEPROM 的大小(该值由芯片的程序存储器、数据存储器和 EEPROM 的大小决定)。目标选项卡中的其他一些设置如下：

- Text Adress(WORD)文本框——预留的中断向量地址大小。
- Enhanced Core 复选框——是否启用增强核功能(只对部分 mega 系列有效)。
- Program Type 选项组——应用程序还是启动引导。
- AVR Studio Simulator IO——AVR Studio 仿真器接口。

注意：其他选项一般采用默认值。

3.4　用 ICC AVR 编写应用程序

当 ICC AVR 的集成开发环境设定好属性后，就可以编写源程序文件了。本节将通过编写一个简单的输入输出程序来了解用 ICC AVR 开发单片机应用程序的过程，电路的连接如图 3.7 所示。

1. 建立工程项目

单击 Project | New 命令，出现 Save New Project As...对话框。这个对话框可以设置新建工程项目的文件名和存放目录。选择好工程目录，在【文件名】文本框中输入工程名称，再单击【保存】按钮就可以自动新建工程项目了，如图 3.8 所示。

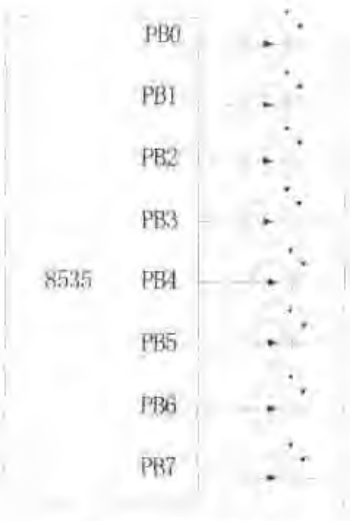


图 3.7　电路连接图

图 3.8　Save New Project As...对话框

2. 建立源文件

单击 File | New 命令，或者单击工具栏上的【新建】图标可以建立源文件。在新出现的窗口中输入源程序如下所示：

```
#include <iov8535.h>

//函数中明
```

```
void initialization(void);
void delay(void);

//初始化
void initialization(void)
{
    DDRB = 0xff;          // 设置B口为输出
    TCCR0 = 0x05;
}

//延时子程序
void delay(void)          //延时
{   while (!(TIFR&0x02));
    TIFR = 0x02;          //清除溢出标志
}

//主程序
void main (void)
{
    initialization();   //初始化
    while(1)
    {   PORTB++;          //增量输出
    delay();          //调用延时
    }
}
```

整个程序分为初始化、延时和主程序循环 3 个部分。在初始化部分，PORTB 被设置为输出，并且定时计数器 0 被用来记录主时钟信号的 1024 分频；在延时部分，程序等待定时计数器 0 的溢出标志位，当溢出标志位为 1 时，程序先把溢出标志位清 0，然后跳出循环；在主程序循环中，PORTB 的输出值做增 1 运算，而后再调用延时子程序。

程序输入完成后，单击 File | Save 命令，或者单击工具栏上的【保存】图标可以把源程序保存到磁盘中的指定位置。如图 3.9 所示，在 Save File As...对话框中指定源文件的目录，输入源文件的名称，再单击【保存】按钮将源文件保存到磁盘。

图 3.9 Save File As...对话框

3. 向工程中添加文件

源文件编写完成以后，还需要添加到工程文件中才能被系统编译。为把文件添加到工程中，首先单击 Project | Add File(s)命令，在弹出的如图 3.10 所示的 Add Files 对话框中，

打开源文件文件所在的目录，并选择需要加入到工程中的源文件，再单击【打开】将文件添加到工程。

图 3.10　Add Files 对话框

4. 编译代码

编译是将高级语言编写的程序转换为单片机可执行的机器代码的过程。单击 Project | Make Project 命令，或者单击工具栏上的【编译工程】按钮都可以对工程进行编译，编译过程中的错误信息会显示在编译窗口中，而单片机的可执行文件则为工程所在目录下的 HEX 文件。

3.5　下载程序文件

系统的应用程序必须转移到单片机的程序 FLASH 上，才能被单片机执行，这一过程就被称作“下载”。AVR 单片机的程序下载可以采取多种方式，其中以串行下载的方式最为普遍。为了得到较快的下载速度，AVR 单片机也支持并行下载的方式。本书只介绍最为常用的串行下载方式。

注意：部分 mega 系列的 AVR 单片机还支持 JTAG 下载编程。

为把应用程序下载到 AVR 单片机，需要使用串行下载软件。PonyProg2000 是一种通用的单片机串行下载软件，它支持 AT89 系列、PIC 系列和 AVR 系列等多种单片机的程序下载。PonyProg2000 的使用非常简单，用户只需把下载线的一端接在 PC 机的 RS232C 串行口，另一端接单片机的在线编程口(ISP 口)，就可以实现程序的下载。下载电缆的连接方式参见附录。

PonyProg2000 是一个可免费使用的软件，用户可直接到它的公司主页上下载(http://www.LancOS.com)。当用户完成 PonyProg2000 的下载和安装以后，可按如下步骤实现单片机的编程。

(1) 启动 PonyProg2000 选择芯片类型。

启动 PonyProg2000 程序，连接好 PC 到单片机的下载电缆，并在 Device 菜单中选择合适的芯片类型，如图 3.11 所示。

图 3.11 选择芯片类型

(2) 导入下载文件。

单击 File | Open Program(FLASH)File...命令，选择要导入的下载文件，如图 3.12 所示。导入到 PonyProg2000 中的下载文件还可以在窗口中进行编辑，以满足用户的特殊要求。

(3) 设置编程步骤。

单击 Command | Program Options...命令，设置单片机的编程步骤，如图 3.13 所示。通常单片机的编程步骤包括芯片擦除、程序写入和程序校验 3 个步骤，用户还可根据自身的要求定义芯片的编程操作。

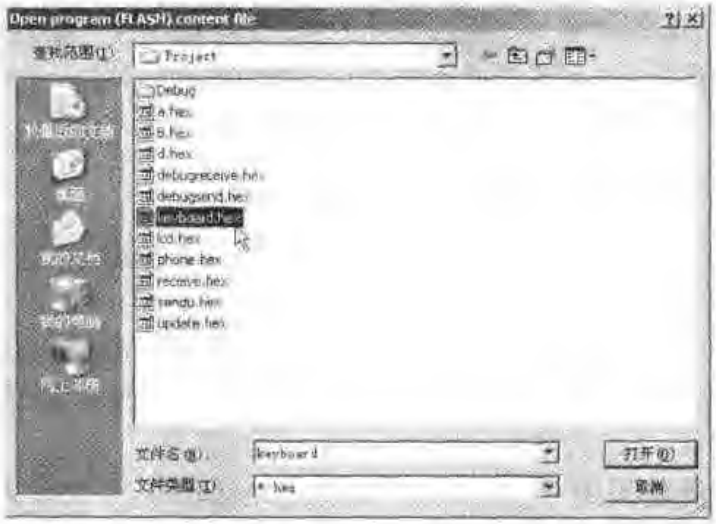


图 3.12 选择下载文件

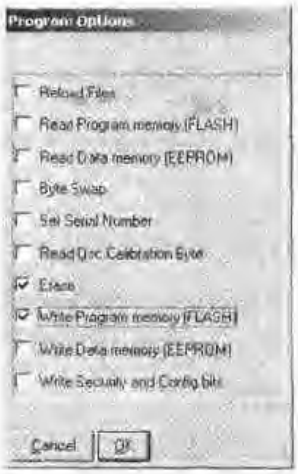


图 3.13 设置编程步骤

(4) 串行编程。

单击 Command | Program 命令对单片机进行编程。系统会按用户设置的编程步骤自动完成芯片擦除、写入、校验等操作。

第 4 章　数据类型、运算符和表达式

任何一种程序设计语言都有对数据的描述和对数据处理的描述。在 C 语言中，系统对数据的描述，是以数据类型的形式出现的；对数据处理的描述，是以运算符和表达式的形式出现的。本章将介绍 C 语言的数据类型、运算符和表达式，从而使读者为学习 C 语言进行单片机编程打好基础。

4.1　ICC AVR 支持的数据类型

数据类型就是数据的不同表示形式，它们按照被描叙数据的性质及其所占存储空间的大小来划分。ICC AVR 的 C 编译器可以支持 4 种数据类型，它们分别是：基本类型、构造类型、指针类型和空类型。具体支持的数据格式如图 4.1 所示。

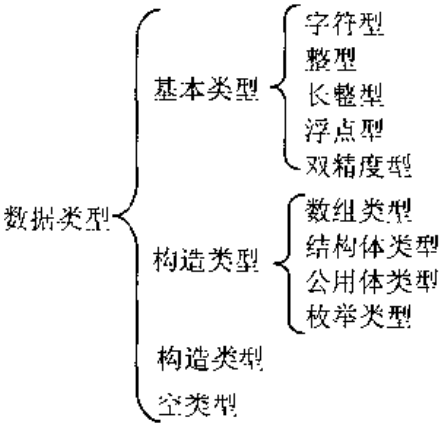


图 4.1　ICC AVR 支持的数据类型

1. 基本数据类型

基本数据类型包括字符型(Unsigned Char 和 Signed Char)、整型(Unsigned Int 和 Signed Int)、长整型(Unsigned Long 和 Signed long)、浮点型(Float)和双精度型(Double)。基本数据类型不只有再分性，它不可以再分解为其他的数据类型。

2. 构造数据类型

构造数据类型包括数组型(Array)、结构体类型(Struct)、公用体类型(Union)和枚举类型(Enum)。构造数据类型由若干个基本数据类型采用构造的方法组成，因此，每个构造数据类型都可以分解成若干个基本数据类型。

3. 指针类型

指针是一种用来专门存放所指对象地址的变量，它指向计算机内的一个内存单元。指针的处理具有较高的灵活性，使用它可以有效地表示复杂的数据结构，有效地使用内存空间，方便地处理数组，因此，它是C语言中一种非常重要的数据类型。

4. 空类型

空类型也就是Void类型。在调用函数时，函数通常都会返回一个函数值，这个返回的函数值的数据类型可以在函数定义和函数说明中由用户指定。但有一些函数，它只进行一定的处理，在调用后并不需要返回某个函数值，那么这种函数可以定义为“空类型”。

当一个计算的几个运算分量的数据类型不相同时，要根据一些规则把它们转换成某个共同的数据类型，这种转换可自动进行，也可以人工强制转换。通常来说，编译器会把“窄”的数据类型自动转换成“宽”数据类型，以保证计算信息不被丢失。

4.2 常量与变量

在程序系统中，数据通常分为常量和变量两种。在程序执行过程中，值不能改变的数据称为常量，值可以改变的数据称为变量。常量和变量都有不同的数据类型。按照数据类型的不同，常量可分为字符常量、整型常量、浮点常量和枚举常量。变量可分为字符变量、整型变量、浮点变量、结构变量和指针变量。在C语言中规定，变量在使用前必须先定义，而常量则可以直接使用。

4.2.1 常量

常量是在程序执行期间不能改变的量。按数据类型的不同，常量可以分为字符常量、整型常量、浮点常量和枚举常量4种。

1. 字符常量

字符常量就是一个字符。在C语言中，字符常量用被单引号括起来的字符表示。

2. 整型常量

整型常量包括Unsigned Int、Signed Int、Unsigned Long和signed Long这4种，它们都可以使用八进制、十进制和十六进制的表示形式。

- 八进制整常数。八进制整常数以0开头，数字序列的取值范围为0～7。
- 十进制整常数。十进制整常数没有前缀，数字序列的取值范围为0～9。
- 十六进制整常数。十六进制整常数的前缀为0x，其数码取值为0～9和a～f。

例：

123　　十进制整常数
0123　　八进制整常数
0x12a　　十六进制整常数

3. 浮点常量

浮点常量也称作实型常量，它有小数表示法和指数表示法两种形式。

- 小数表示法。由十进制小数构成，如 5.6、3.1 都是小数表示的浮点常量。
- 指数表示法。指数表示形式由十进制数、字母 E 以及幂指数 n 组成，如 3E2，其值为 300。

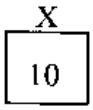
4. 枚举常量

枚举型常量是整常数集合中的一个元素，它表示该元素在集合中的位置。

4.2.2　变量

变量在程序的执行过程中可以改变。每个变量都有一个名字和它对应，这个名字就称为变量名或标识符。变量在内存中会占据一定的存储单元，该内存空间用于存储变量的值。

变量的变量名和变量值是两个不同的概念，它们之间的关系如下所示：



其中，X 为变量名，它只是变量的名称，而变量的值则是内存空间中可以改变的数据。

变量在使用之前还必须先定义，以决定变量的取值范围、运算种类和存储方式。变量定义的一般形式为：

类型标识符 变量名

例：

```
int a;
float sum;
```

同常量一样，变量按其数据类型的不同也可以分成不同的类型，它们分别为字符变量、整型变量、浮点变量、结构变量和指针变量。

1. 字符变量

字符型变量通过关键字 char 说明。

例：

```
char a,b
```

一个字符型变量实际上占据一个 8 位的内存空间，因此它的存储范围为-127～+127。

2. 整型变量

整型变量同过关键字 int 或 long 来定义说明。由 int 定义的整型量占据 16 位的内存空间，由 long 定义的整型量占据 32 位的内存空间。

例：

```
int memory;
long data;
```

3. 浮点变量

浮点型变量也称为实型变量，实型变量可以分为单精度和双精度两类。单精度型的浮点型变量采用关键字 float 来说明，占据 4 个字节的内存空间；双精度浮点型变量采用 double 关键字说明，占用 8 个字节的内存空间。

例：

```
float x,y;        x,y 定义为单精度实型
double a,b;       a,b 定义为双精度实型
```

4. 结构变量

用结构体的类型名定义的变量就是结构体变量，它所占据的内存空间取决于结构体的结构成员个数和类型。它的定义形式为：

```
struct 结构体名
{
    结构成员说明
}变量名;
```

例：

```
struct x
{
    int a;
    double b;
    char c
}sem;
```

5. 指针变量

指针变量是被用来指向变量的指针的变量，它被用来存放地址。它的定义形式为：

类型标识符*指针变量名

例：

```
int *a;
double *next;
```

4.3 AT90LS8535 的存储空间

AVR 单片机是哈佛结构的微处理器，它的程序存贮器和数据存贮器相互独立，并分别占据不同的地址空间。AVR 单片机的这种结构允许它可以比传统结构的单片机访问更多的存贮单元。如 Atmega 系列的 AVR 产品允许 CPU 寻址超过 64KB 的程序存贮器和 64KB 的数据存贮器，这样尽管单片机的程序计数器仍是 16 位，但它可以使用到更多的程序存贮空间。

AT90LS8535 的存储空间在物理上分为以下两部分。

- 程序存储器
- 数据存储器

其结构如图 4.2 所示。

其中的程序存储器用于存放单片机的执行程序和数据表格，而数据存储器则用于存放程序执行过程中的变量值。

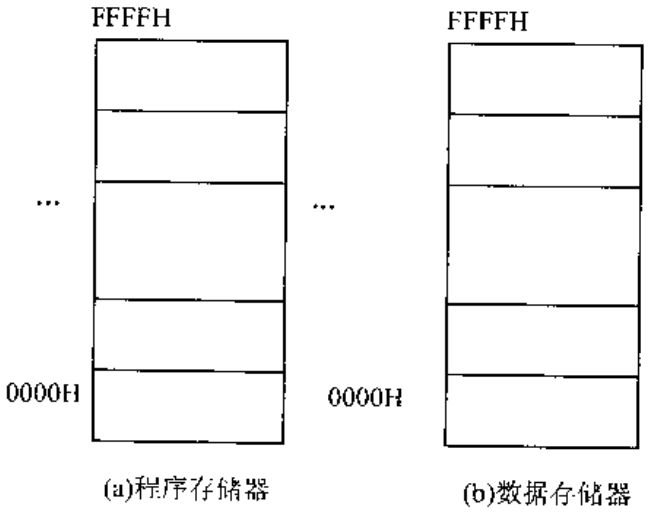


图 4.2　AT90LS8535 的存储空间

注意：　对于 AT90LS8535 来说，它的数据存储器实际上还包括了 I/O 地址空间和 32 个通用寄存器组。

4.4　算术和赋值运算

4.4.1　算术运算符和算术表达式

1. C 语言中的基本算术运算符

- “+”(加法运算符)：加法运算符为双目运算符，具有右结合性。“+”也可作为正值运算符。
- “-”(减法运算符)：减法运算符为双目运算符，具有左结合性。“-”也可作为负值运算符。
- “*”(乘法运算符)：乘法运算符为双目运算符，具有左结合性。
- “/”(除法运算符)：除法运算符为双目运算符，具有左结合性。当两个操作数都为整型数时，运算的结果省略小数部分，也为整型数；若两个操作数中有一个是实型数，则运算的结果为双精度的实型数。
- %(求余运算符)：求余运算符也为双目运算符，具有左结合性。该运算符要求两个操作数均为整形数，其结果为两操作数相除后的余数。

例：若变量说明为

```
int a=3,b=7;
float c=2.0;
```

则以下算术表达式的值为：

b/a 2(表达式的值省略小数部分)

c*-a-6.000000(因为 c 为实型数，所以表达式的值也是实型)

c/a 0.666667(因为 c 为实型数，所以表达式的值也是实型)

b%a 1(b 和 a 相除后的余数)

2. 增 1 和减 1 运算符

对于运算 i=i+1 或 i=i-1 运算，C 语言提供两种更简洁的运算符——增 1 运算符和减 1 运算符，它们分别是：

```
++i, i++; --i, i--
```

其中++i 和 i++都表示变量 i 加 1，即 i=i+1；--i 和 i--都表示变量 i 减 1，即 i=i-1。增 1 运算符和减 1 运算符都是右结合性。

对于增 1 和减 1 运算符，使用时必须注意以下几点：

- 增 1 和减 1 运算只适合与整型变量和指针变量，其他的数据类型都不能使用这种运算符。
- 对于++I、i++或--i、i--，它们的操作并不相同，如果++、--在操作数之前，则程序在使用操作数之前，先使操作数增 1 或减 1；如果++、--在操作数之后，则操作数先使用操作数，再使其增 1 或减 1。

例：

```
main()
{
    int i=1,j=1;
    printf("i=%d\n",i++);
    printf("ix=%d\n",i);
    printf("j=%d",++j);
    printf("jx=%d\n",j);
}
```

运行结果为：

```
i=1
ix=2
j=2
jx=2
```

3. 算术表达式、优先级与结合性

算术表达式是指用算术运算符将常量、变量、函数等对象连接起来的式子。每个算术表达式中的运算符都有一个执行的先后次序，这种执行的先后次序就被称作优先级。所谓结合性则是指当运算对象两侧的运算符具有相同的优先级时运算的执行顺序。算术表达式的求值就是按照运算符的优先级和结合性来执行的。

- 算术表达式——由运算对象(常量、变量、函数等)、算术运算符和括号组成的表

达式。

例：

```
a+b
a+b+c*die/c
a*(b+c)-(d+e)/f
sin(x)+cos(x)
36.89*x-31*y
++i-j++
```

- 优先级——指在算术表达式中，不同运算符之间的执行次序。在C语言中，当一个操作数两边都有运算符时，程序按运算符的优先级高低执行运算，优先级较高的先于优先级较低的进行运算。
- 结合性——指当一个操作数两侧的运算符的优先级相同时的运算顺序。C语言中的结合性分为左结合性和右结合性两种。左结合性采用从左至右的结合顺序。C中的算术运算符就是左结合性的。例如：对于 a+b+c，程序先执行 a+b 的操作，而后再把 a+b 的结果和 c 相加。而右结合性则采用从右至左的结合顺序。在C的运算符中，赋值运算符就是采用右结合性的。

4.4.2　赋值运算符和赋值表达式

1. 赋值运算符

赋值运算符“=”用于改变内存空间的值，实现变量的赋值操作。赋值运算符是双目运算符，它的一般形式为：

变量=表达式

例：

```
a=3*y
```

这条语句的作用就是将右边表达式 3*y 的值赋给左边的变量 a 所在的存储单元。

2. 复合赋值运算符

复合赋值运算符是在赋值符“=”之前加上其他二目运算符所构成的赋值符，它的一般形式为：

变量 双目运算符=表达式

如：+=，-=，*=，/=，%=，<<=，>>=，&=，^=，|=。

复合赋值运算符的优先级和赋值运算符相同，也具有右结合性，因此它实际上等价于以下这种形式：

变量=变量 运算符 表达式

例：

```
x+=y          等价于 x=x+y
b*=5          等价于 b=b*5
x%=y+3        等价于 x=x%(y+3)
```

C 语言提供的这种复合赋值符，能使表达式更为简洁、程序更为精练，同时它也有利于提高程序的编译效率，以便产生质量较高的目标代码。

3. 赋值运算符中的类型转换

当赋值运算符两边运算对象的类型不相同，将发生类型转换，系统会自动把赋值运算符右边的数据类型转换成左边的数据类型。具体转换如下：

- 浮点型与整型。浮点型转化为整型时，浮点数的小数部分会被舍弃，只保留整数部分；整型转化为浮点型时，数值不变，而只将类型改为浮点型。
- 单、双精度浮点型。float 型转化为 double 型时，只是在尾部添 0；double 型化为 float 型时，通过四舍五入的截断操作实现。
- 字符型和短整形。字符型转化为短整型时，值保持不变，只是数据的内部表现形式改变；短整型转化为字符型时，只保留最低 8 位，高 8 位舍弃。
- 短整型与长整型。短整型转化为长整型时，其值保持不变，而内部形式变为长整型；长整型转化为短整型时，长整型的高 16 位舍弃，低 16 位传给短整型。

4. 赋值表达式

利用赋值运算符将变量和表达式连接起来的式子就是赋值表达式。其一般形式为：

变量=表达式

例如，a=2*7+y 就是一个赋值表达式。

赋值表达式也有一个确定的值和类型，它的值是被赋值后的变量值。它的执行过程是：程序先求赋值运算符右边表达式的值，然后把这个值赋给左边的变量。

4.5 逻辑运算

逻辑运算用于在运算过程产生一个逻辑量，C 语言中没有专门的逻辑值，其真假值分别用 1、0 表示。实际上，当判断一个逻辑值为非 0 时，认为逻辑值为“真”；当判断一个逻辑值为 0 时，认为逻辑值为“假”。C 语言提供 3 种逻辑运算，它们分别是：

- 逻辑与 & &(AND)
- 逻辑或||(OR)
- 逻辑非！(NOT)

其中逻辑与和逻辑或是双目运算，它需要有两个操作数；逻辑非是单目运算符，它只需要一个操作数。

由逻辑运算符及其操作数所构成的表达式称为逻辑表达式，逻辑表达式的值是一个表示“真”、“假”的逻辑量。C 语言中由逻辑运算符构成的逻辑表达式有以下 3 种形式：

- 操作数 & &操作数
- 操作数||操作数
- !操作数

逻辑表达式中的操作数可以是常量、变量和表达式。

对于逻辑与，只有当逻辑运算符两边的操作数均为“真”时，结果才为“真”；否则

结果就为“假”。逻辑与的这个运算特点也就是：若逻辑运算符的左边操作数为“假”，则无需再判断其他操作数的值就可断定表达式的值为“假”；而只有左边的操作数不为“假”时，才有必要继续判断其他操作数。

对于逻辑或，只有当逻辑运算符两边的操作数均为“假”时，结果才是“假”；否则结果就是“真”。利用逻辑或的这种运算特点，可以判断：若左边的操作数为“真”，则无需再判断其他操作数就可断定表达式的结果为“真”；而只有当右边操作数为“假”时，才判断右边的的操作数。表 4.1 为逻辑运算的真值表。

表 4.1　逻辑运算真值表

	操作数 1	操作数 2	运算结果
逻辑与	0	0	0
	0	1	0
	1	0	0
	1	1	1
逻辑或	0	0	0
	0	1	1
	1	0	1
	1	1	1
逻辑非	0	无	1
	1	无	0

例：

(a&&b)&&x　其中 a=0。因为 a=0，所以可以断定该逻辑表达式的值为“假”；
(a||b)||x　其中 a=1。因为 a=1，所以可以断定该逻辑表达式的值为“真”。

4.6　关 系 运 算

所谓关系运算是指对两个操作数进行比较的一种运算。同逻辑运算一样，关系运算的结果也只有“真”和“假”两个值。

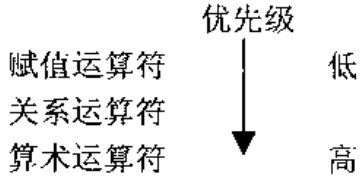
1. 关系运算符

C 语言共有 6 种关系运算符，它们分别是：

- < 小于
- <= 小于或等于
- > 大于
- >= 大于或等于
- == 等于
- != 不等于

在这 6 个关系运算符中，<、<=、>、>=的优先级相同，后 2 种的优先级也相同；且前

4 种的优先级高于后 2 种的优先级。关系运算符与其他运算符的优先级别关系为：



关系运算符都是双目运算符，每个运算符都需要两个操作数。

2. **关系表达式**

用关系运算符将两个表达式(可以是算术表达式、赋值表达式或关系表达式)连接起来的表达式就称为关系表达式，它的一般形式为：

表达式 1 关系运算符 表达式 2

例：

```
a>b+c
a==b>c
```

关系表达式采用自左向右的结合方式。

关系表达式的值是一个逻辑量，它只有“真”和“假”两个值，分别用“1”和“0”表示。

例：若 a=1，b=2，求 c=a>b。

```
main()
{
    int a=1,b=2,c;
    c=a>b;
    printf("%d",c);
}
```

运行结果为：0

4.7 位 操 作

位操作就是指以二进制位为单位进行的操作。在应用系统中，经常需要对位进行运算或处理。C 语言提供了类似汇编语言的位操作功能，这使得它与其他高级语言相比具有了更大的优越性。

ICC AVR 的 C 编译其具有以下 6 种位操作运算符：

- & 按位与
- | 按位或
- ^ 按位异或
- ~ 按位取反
- << 左移
- >> 右移

4.7.1 位逻辑运算

C 语言的位逻辑运算包括按位与运算、按位或运算、按位异或运算和按位取反运算 4 种。

1. 按位与

按位与运算符&是双目运算符，也就是参加运算的操作数必须要两个。

运算规则：按位与运算把参与运算的两个操作数的对应二进位相与，若两者的对应二进位均为 1 时，结果位就为 1，否则结果位就为 0。

0&0=0

0&1=0

1&0=0

1&1=1

例：若 a=7,b=8,求 a&b。

a=00000111B

b=00001000B

把 a 和 b 的对应位相与可得：a&b=0。

2. 按位或

按位或运算符|是双目运算符。

运算规则：按位或运算把参与运算的两个操作数的对应二进位相或，只要两者的对应的任一个二进位有为 1，结果位就为 1，否则结果位就为 0。

0|0=0

0|1=1

1|0=1

1|1=1

例：若 a=7,b=8,求 a|b。

a=00000111B

b=00001000B

把 a 和 b 的对应位相或可得：a|b=15。

3. 按位异或

按位异或运算符^是双目运算符。

运算规则：按位异或运算把参与运算的两个操作数的对应二进位相异或，若两者对应位的二进位相异，则结果位为 1，否则结果位就为 0。

0|0=0

0|1=1

1|0=1

1|1=1

例：若 a=7，b=8，求 a|b。

a=00000111B

b=00001000B

把 a 和 b 的对应位相或可得：a|b=12。

4. 按位取反

按位取反运算符~是单目运算符，它的操作数只有一个。

运算规则：按位取反运算对其操作数的各二进位进行求反。

!0=1

!1=0

例：若 a=7，求!a。

a=00000111B

把 a 的各二进制位取反可得：!a=0xF8。

4.7.2 移位运算

C 语言中的移位运算只有两个：左移运算和右移运算。

1. 左移运算

左移运算符<<是双目运算符。

运算规则：左移运算把运算符左边的操作数按运算符右边的操作数给定的数值向左移若干位，从左边移出的高位部分将被舍弃，右边空出的低位部分补 0。

例：若 a=01101011B，则将 a 左移 3 位的表达式为 a<<3，结果如下：

01101011B

01101011B　　左移 3 位

01011000B　　右边空出低位补 0

表达式 a<<3 的值为 0x58。

由上例可见，左移运算实际上实现了乘法的功能，左移一位相当于乘以 2，左移两位相当于乘以 4。由于左移运算比乘法运算快得多，因此在一些实时性要求较高的场合，通常也采用这种左移运算来实现一个数和 2^n 相乘。

2. 右移运算

右移运算符>>也是双目运算符。

运算规则：右移运算把运算符左边的操作数按运算符右边的操作数给定的数值向右移若干位，从右边移出的低位部分将被舍弃，对于无符号数来说，左边空出的高位部分补 0；对于符号数来说，若符号位为 0，则左边空出的高位部分补 0。若符号位为 1，则左边空出的部分针对不同的计算机系统，或者补 0，或者补 1。

例：若 a=11101010B，则将 a 右移 2 位的表达式为 a<<2，结果如下：

11101010B

11101010B

000111010B

表达式 a>>3 的值为 0x3a。

右移运算相当为左移运算的逆运算，它实际上实现的是除法的功能，右移一位相当于除以 2，右移两位相当于除以 4。由于右移运算比除法运算快，因此也可采用这种运算形式来实现一个数和 2^n 的除法。

3. 移位运算的应用

例：利用移位操作实现循环右移功能。

```
//移位操作子程序
unsigned char move(unsigned char j,int k)
{
    unsigned char mid;
    mid=(j>>n)|(j<<(8-n));
    return(mid);
}

//主程序
main()
{
    unsigned char a;
    int n;
    printf("请输入一个整数\n");
    scanf("%d",&a);
    printf("请输入要移位的位数\n");
    scanf("%d",&n);
    printf("移位后的结果为%d",move(a,n));
}
```

4.8 逗 号 运 算

C 语言中的逗号运算符就是“，”，它可以把两个或多个表达式连接起来构成一个新的表达式，这种用逗号连接起来的表达式就称为逗号表达式。逗号表达式的一般形式如下：

表达式 1，表达式 2，…，表达式 n

逗号表达式按照从左至右的顺序求出表达式 1 至表达式 n 的值，而整个表达式的值则取决于最后一个表达式。

使用逗号表达式必须注意以下两点：

- 逗号表达式允许嵌套，也就是说表达式 1，表达式 2，…，表达式 n 也可以是逗号表达式。
- 并不是所有出现逗号的地方都是逗号表达式。例如在函数的参数表中，在变量的定义中，逗号都只是表示各变量之间的间隔。

例：

```
main()
```

```
{
    int a,b;
    a=100;
    b=(a=a/10,a/2);
    printf("a=%d",b);
}
```

本例中，对于语句“b=(a=a/10,a/2);”，程序先执行 a=a/10，求得 a=10，然后再求 a/2，结果为 5，最后再把 5 作为整个逗号表达式的值传递给标量 b。

注意： 例中逗号表达式的逗号不可以省略，因为逗号运算符是所有运算符中优先级最低的。

第5章 控 制 流

C语言的程序控制流程可以分为3种基本结构，即顺序结构、选择结构和循环结构。通过这3种基本的程序结构可以构成各种复杂的程序。本章将主要介绍这些基本程序结构的语句实现，包括if语句、switch语句、while语句和for语句的语法格式和语意内容，以及各语句在程序控制中的应用，并为后面章节的学习打好基础。

5.1 C语言的结构化程序设计

C语言是一种结构化的编程语言。从程序流程的角度来看，结构化的程序包含3种基本结构，即顺序结构、选择结构和循环结构。这3种基本的程序结构又可以相互组合、嵌套，从而构成各种复杂程序。

5.1.1 顺序结构

顺序结构是一种最基本的程序结构。在这种控制结构中，程序按照程序的存放地址由低向高顺序执行。如图5.1所示，程序先执行语句1，再执行语句2，两者按顺序关系执行。

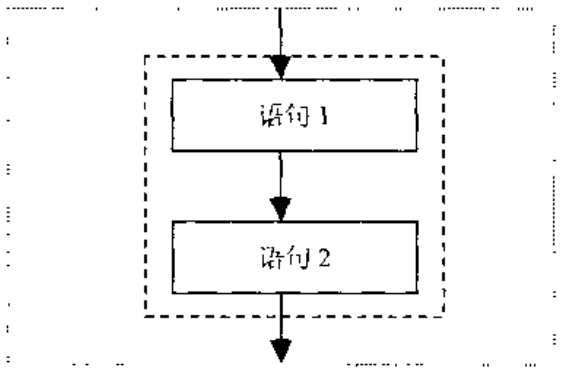


图5.1 顺序结构流程

例：已知圆柱体的底半径为radius，高为high，求其体积。

源程序为：

```
#include<math.h>
main()
{
    float radius,high,vol,pi=3.1415926;
    printf("Please input two numbers!\n");
    scanf("%f%f",&radius,&high);
```

```
    volume=pi*radius*radius*high;
    printf("radius=%7.3f, high=%7.3f,
    volume=%7.23\n",radius,high,volume);
}
```

5.1.2 选择结构

选择结构使计算机具有了决策和选择的能力。如图 5.2 所示，在选择控制结构中，程序先是对某个条件语句进行判断，如果条件为“真”，则执行语句 1；如果条件为“假”，则执行语句 2，语句 1 和语句 2 最后汇集到一起并从一个程序出口中退出。由选择结构的上述功能可以看出，选择语句用于对计算机的某个事件处理提供多条通道，从而使计算机具有选择控制能力。

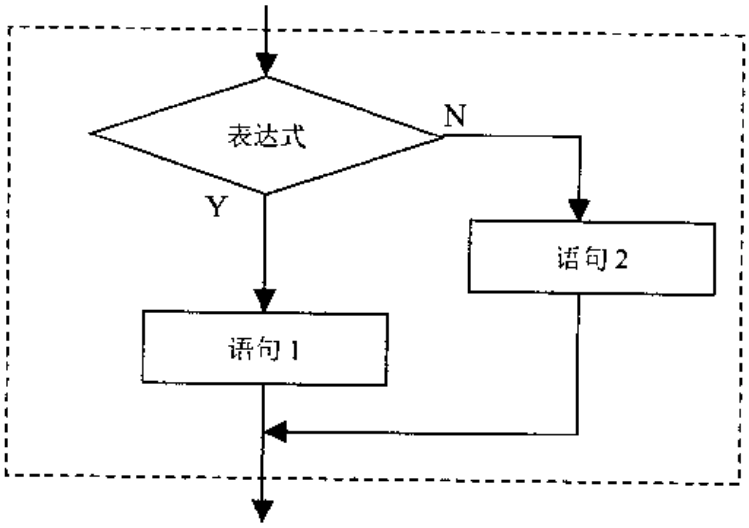


图 5.2 选择结构流程

选择结构程序的设计，需要考虑两个方面的问题。其一是条件的表达，其二是选用何种选择结构语句来实现。对于 C 语言来说，一般用关系表达式或逻辑表达式表示选择条件，而选择结构的实现则常用 if 或 switch 语句。

例：输入两个整数，输出其中的较大值。

```
main()
{
    int a,b;
    printf("Please input two integers!\n");
       scanf("%d,%d",&a,&b);
       if(a>b)
            printf("%d\n",a);
        else
         printf("%d\n",b);
}
```

5.1.3　循环结构

循环结构用于重复执行一组有规律的指令集合，它可分为无条件循环和条件循环两种。由于无条件循环会导致程序的结构混乱，并使程序的可读性变低，因此，在程序设计中一般不采用无条件循环结构，本书也不再做进一步的讨论。对于条件循环结构，程序在给定条件成立时会反复执行某条语句，直至条件不再成立为止，如图 5.3 所示。C 语言提供了 for、while 等多种循环语句，它们可以相互组成各种不同形式的循环结构。

C 语言中的循环结构又可以分为当型循环结构和直到型循环结构。在当型循环结构中，程序首先判断循环条件，如果为“真”，则程序重复执行循环体中的语句；如果为“假”，则退出循环。在直到型循环结构中，程序首先执行的是循环体中的语句，然后再判断循环条件，如果条件为“真”，则再重复执行语句，否则就退出循环。

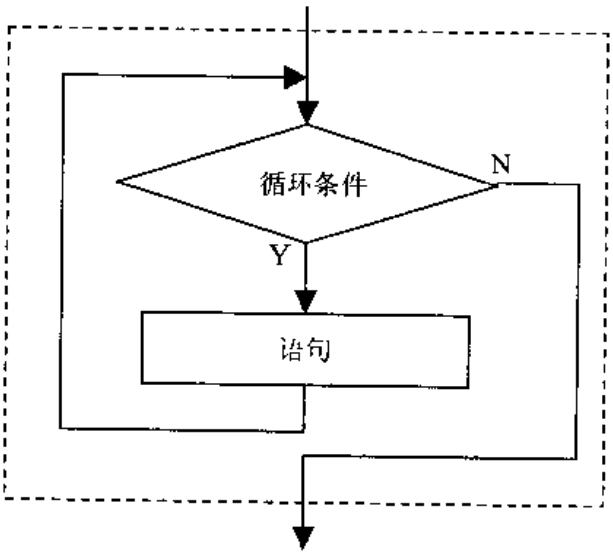


图 5.3　循环结构的流程

5.2　选 择 语 句

选择语句用来判断给定的条件是否满足要求，并根据判断的结果选择程序的执行分支，它构成了计算机的判断决策基础，是模块化程序的重要组成部分。

C 语言中常用的选择语句有：if 语句和 switch 语句。其中 if 语句具有 if…else、if 和 else if 三种形式。

5.2.1　if 语句

1. if…else 分支

if…else 语句的基本结构形式为：

```
if (表达式)
```

```
    语句1;        //当表达式为“真”时
else
    语句2;        //当表达式为“假”时
```

语句说明：在这种结构中，如果表达式的值为“真”，则程序执行语句1；如果表达式的值为“假”，则程序执行语句2。其执行过程如图5.4所示。

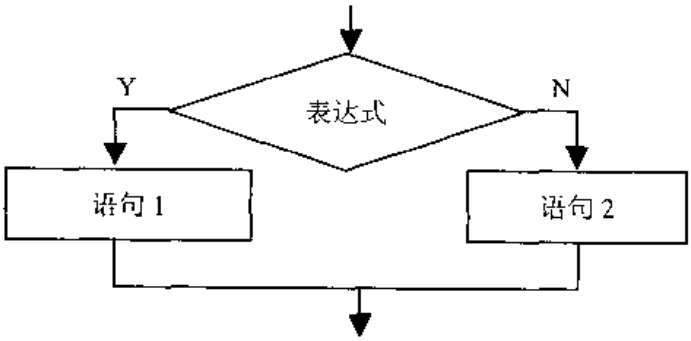


图5.4 if…else语句流程

结构中的判断表达式可以为逻辑表达式或关系表达式，也可以为任意的数值类型。语句1和语句2可以是一条语句也可以是由大括号构成的复合语句。

例：输入一个数，如果为1，则输出“OK”；否则输出“ERROR”。

```
main()
{
    int a;
    printf("Please input one intergers!\n");
    scanf("%d",&a);
    if(a==1)
        printf("OK\n");
    else
    printf("ERROR");
}
```

2. if分支

if分支是if…else语句的简单形式，当if…else语句缺省else部分时，就形成了if结构的分支，其基本结构形式为：

```
if(表达式)
    语句;     //当表达式为“真”时
```

语句说明：if分支中，如果表达式的值为“真”，则程序执行语句；如果表达式的值为“假”，则程序不执行任何操作。语句的执行流程如图5.5所示。

例：输入两个整数，输出其最大值。

```
main()
{
int a,b;
    printf("Please input two integers!\n");
    scanf("%d, %d",&a,&b);
    if (a<b)
```

```
a=b;
    printf("%d",a);
}
```

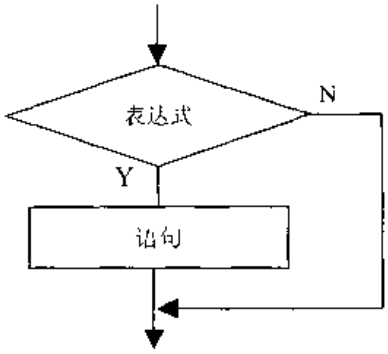


图 5 5　if 语句流程

3. else if 分支

if…else 和 if 语句都只能提供两个分支供程序选择。但在实际应用中，两个分支往往是不够的，例如：对三分段函数，相应的处理分支也需要 3 个。在这种情况下，可以采用 else if 的多分支结构。该结构的基本形式为：

```
if(表达式 1)  语句 1;
else if(表达式 2)语句 2;
else if(表达式 3)语句 3;
…
else if(表达式 n-1)  语句 n-1;
else    语句 n;
```

语句说明：else if 语句为一个多分支语句。语句执行时，程序依次判断各表达式，如果表达式为“真”，则程序执行该表达式所对应的语句，随后跳出 else if 分支继续执行程序；如果所有的表达式均为“假”，则执行 else 后的语句 n。else if 语句的执行过程如图 5.6 所示。

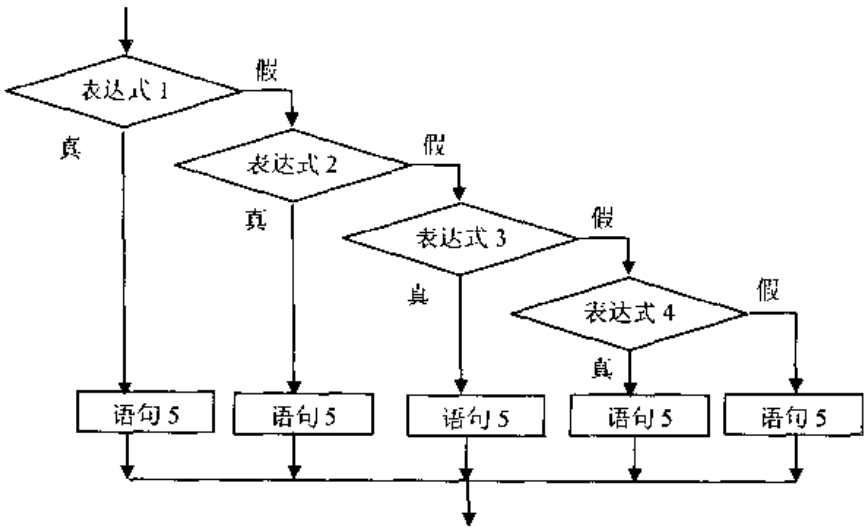


图 5.6　else if 语句流程

例：将百分制成绩转换成优、良、中、及格和不及格5个等级。

```
main()
{
    int score;
    printf("Please input a scroe!\n");
    scanf("%d ",&score);
    if(0<=score&&score<60)
        printf("Rank=%c",'E');
    if(60<=score&&score<70)
        printf("Rank=%c",'D');
    if(70<=score&&score<80)
        printf("Rank=%c",'C');
    if(80<=score&&score<90)
        printf("Rank=%c",'B');
    else
        printf("Rank=%c",'A');
}
```

说明：使用if结构还必须注意以下几点：

- 在if语句中的表达式一般是逻辑表达式或关系表达式，但也可以是其他表达式，甚至也可以是一个变量。
- 在if语句中，判断执行分支的条件表达式必须用括号括起来，不加分号；而语句之后则必须加分号。
- 在3种形式的if语句中，分支语句为单个语句。如果要想在分支中执行多个语句，则必须用大括号括起来组成一个复合语句。

例：

```
if(a>b)
{
    t=a;a=c;c=t;
}
    else
{
    t=a;a=b;b=t
}
```

5.2.2 switch分支

if…else语句能从两条分支中选择一条。然而很多情况下，我们希望程序从多个分支中选择一个分支来执行。虽然else if也可实现“多选一”，但这种实现方法的效率较低，代码量也较大。为此，C语言提供了另一种专用于多分支选择的switch语句，它的一般形式为：

```
switch(表达式){
case 常量表达式1:   语句1;
case 常量表达式2:   语句2;
```

```
...
case 常量表达式 n:    语句 n;
default:              语句 n+1;
}
```

语句说明：程序首先计算表达式的值，然后至上而下寻找与表达式的值对应匹配的 case 关键字后面的常量表达式，如果相互匹配，程序就执行其后的语句；若表达式的值与所有 case 后的常量表达式均不相同时，则执行语句 n+1。

当执行完某一分支语句后，程序还会继续执行后续的 case 语句。如果希望程序只执行一条分支，则可利用 break 语句跳出 switch 结构。

例：用 switch 调用不同的输出。

```
switch (a)
{
    case 1:
        printf("china");
        break;
    case 2:
        printf("is");
        break;
    case 3:
        printf("a");
        break;
    case 4:
        printf("great");
        break;
    case 5:
        printf("country");
        break;
    case 6:
        printf("!");
        break;
    default:
        printf("OK");
}
```

说明：使用 switch 结构还必须注意以下几点：

- switch 后的执行部分，必须用大括号括起来。
- 在同一个 switch 结构中，不能有相同的 case 常量表达式出现。
- 在 case 后的多个语句，可以不用大括号括起来。
- 为实现程序执行一个分支，可采用 break 语句来跳出 switch 结构的方式实现。

5.2.3　选择语句的嵌套

当一个选择语句中又包含了一个或多个选择语句时，就构成了选择语句的嵌套形式。if 语句和 switch 语句都允许嵌套，它们的嵌套格式也基本相同，因此，本书主要以 if 语句来说明选择语句的嵌套形式。选择嵌套的一般形式如下：

```
if(表达式1)
    if(表达式2);
        语句1;
    else
        语句2;
else
    if(表达式3)
        语句3;
    else
        语句4;
```

由于if语句中的else部分是选用的，若嵌套内的语句不带else，那么这将会出现else与if匹配的问题。为解决这个问题，C语言中规定，else总是与它最近的if配对。

例：

```
main()
{
    int a,b,c;
    printf("Please input two integers!\n");
    scanf("%d,%d",&a,&b);
    if(a<b)
    if(b<c)
        c=a;
    else
        c=b;
}
```

5.3 循环语句

在实际应用问题中，经常会碰到一些需要重复处理的问题，这就要求能有一种结构可使计算机循环地执行一组语句。循环结构就是这样的一种结构。其特点是，在给定判定条件为“真”时，程序反复执行某一条或某一组语句，直到判定条件为“假”。

循环结构中的循环语句一般由循环体和循环判定条件两步分组成，其中被重复执行的那部分语句被称为循环体，而循环条件则用于判断是否重复执行循环体部分。

C语言中有3条语句可实现循环功能，它们是：

- while语句
- do…while语句
- for语句

5.3.1 while语句

while语句的一般形式为：

```
while(表达式)
    语句;
```

语句中的表达式为循环判定，它是 while 循环是否继续的条件；语句为循环体，它是执行重复操作的语句。

语句说明：程序首先计算表达式的值，当值为“真”时，程序执行循环体；当值为“假”时，则终止 while 循环。while 语句的执行过程如图 5.7 所示。

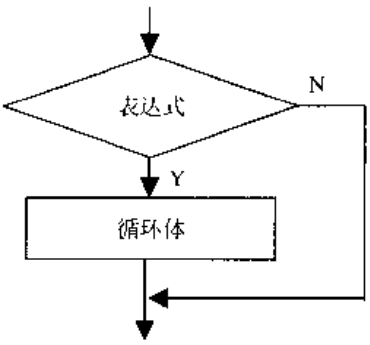


图 5.7　while 语句流程

例：求 1 到 100 的和。

```
main()
{
    int n=1,sum=0;
    while(n<=100)
    {
        sum=sum+n;
        n++;
    }
}
```

说明：使用 while 语句应注意以下几点：

- while 循环结构的循环判定条件在循环体之前，因此在执行循环操作之前必须先进行循环判定条件的测试，如果条件不成立，则循环体内的操作一次也不能执行。
- while 语句中的表达式一般是关系表达或逻辑表达式，但也可以是其他表达式，甚至可以是变量或常量。只要表达式的值为“真”，while 语句就会循环执行。
- 循环体如包括有一个以上的语句，则必须使用大括号括起来，构成一个复合语句。
- while 循环中，应用使循环趋于结束的语句，否则将构成死循环。

5.3.2　do…while 语句

do…while 循环与 while 循环的不同之处在于：while 循环先测试循环判定条件，而 do…while 循环则先执行循环体中的语句，然后再测试循环判定条件。如果条件为“真”，则继续执行循环体；如果条件为“假”，则终止循环。因此，不管循环条件是真是假，do…while 循环至少都会执行一次循环。do-while 语句的一般形式为：

```
do{
    语句;
while(表达式);
```

语句的执行过程如图 5.8 所示。

例：求 1 到 100 的和。

```
main()
{
    int n=1,sum=0;
    do{
```

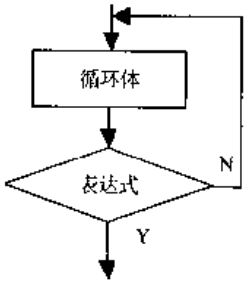


图 5.8　do…while 语句流程

```
        sum=sum+n;
        n++;
    }while(i<=100);
}
```

do…while 循环语句把循环判定条件的测试放在了循环的结尾处，这样就使得循环至少也能够循环一次，因此常用于执行至少一次循环的情况下。

5.3.3 for 语句

在 C 语言中，for 语句是使用最为灵活的一种循环结构。for 循环语句由 3 个部分构成。其中第一个部分为初始化表达式，它用于在开始执行 for 循环时进行循环变量的初始化；第二部分是循环判断表达式，它可以提供循环的终止条件；第三部分是尺度增量表达式，用于在循环之后执行某种操作。如果 for 语句的第一部分和第三部分被省略了，那么 for 语句就退化成了 while 语句；如果用于测试循环条件的第二部分被省略，那么循环就一直进行，这种方式下必须借用其他手段(如 break 语句或 return 语句)终止循环。

for 循环语句的一般形式为：

```
for(表达式 1；表达式 2；表达式 3)
    语句;
```

语句的执行过程为：

(1) 先初始化表达式 1。

(2) 测试循环判定表达式 2，若表达式为“真”，则程序执行 for 语句循环中的循环体；若表达式为“假”，则终止循环。

(3) 求解表达式 3，返回第 2 步继续判断循环表达式。

(4) 结束循环。

其流程如图 5.9 所示。

例：求 1 到 100 的和。

```
main()
{
    int n,sum;
    sum=0;
    for(n=0;n<=100;n++)
    {
        sum=sum+n;
    }
        printf("sum=%d\n",sum);
}
```

从上例可以看出，C 语言中的 for 语句也可以简单地表示如下：

```
for(循环变量赋初值；循环条件表达式；循环变量增量表达式)
    语句;
```

循环变量赋初值为一个赋值语句；循环条件表达式是一个关系表达式，它决定了循环

执行的条件；循环变量增量表达式则定义了循环控制变量每循环一次的变化。

上例中，程序先给 n 赋初值 1，然后判断 n 是否小于等于 100，若是则执行语句 sum=sum+n，随后循环变量增 1，再重新循环，直到 n 不小于等于 100 为止。

由此可见，可以用下面的 while 循环结构来改写 for 循环语句。

```
表达式 1;
while(表达式 2){
    语句;
    表达式 3;
}
```

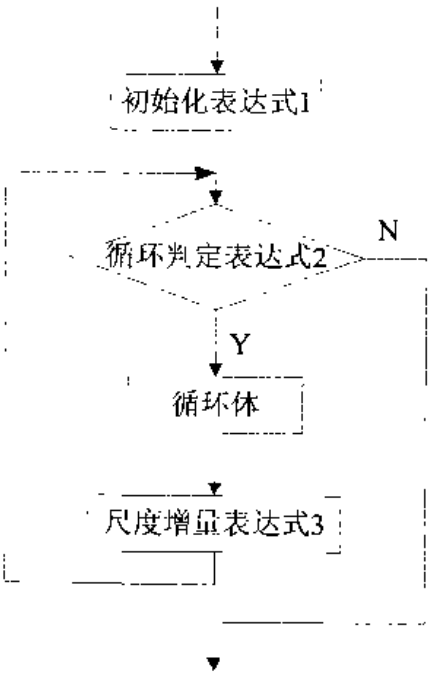


图 5.9　for 语句流程

说明： C 语言中的 for 语句具有如下几个必须注意的地方：

- for 语句中的 3 个表达式都可以省略，但其中的分号不能省。
- 表达式 1 和表达式 3 可以是一个简单的表达式，也可以是一个逗号表达式。
- 如果要使 for 语句中的循环体不止一个语句，必须用大括号把多个语句括起来形成一个复合语句。

5.3.4　循环语句嵌套

当一个循环结构中又包含一个或多个循环结构时，就构成了循环嵌套。C 语言中的 3 种循环结构都允许循环嵌套，并且不同的循环语句之间也可以相互嵌套。

例：

```
main()
{
    int a=1,b=1,c=1;
```

```
    while(a<10)
    {
        do{
            for(;c<10;c++)
                printf("%d%d%d\n",a,b,c);
            b++;
        }while(b<10);
        a++;
    }
}
```

5.3.5 break 语句和 continue 语句

在循环语句执行过程中，有时需要在循环判定条件仍为“真”的时候直接从循环体中退出来，这时就需要用到 break 语句和 continue 语句。break 语句适用于 3 种循环语句，break 语句执行后，程序会立即从最内层的循环语句跳出。continue 语句和 break 语句相似，continue 语句一旦执行，程序就立即开始下一次循环。

1. break 语句

break 语句通常用在循环语句中，它可使程序跳出循环，而执行循环后的语句。在循环结构中 break 语句一般与 if 语句联用，当条件满足时便跳出循环。其流程如图 5.10 所示。

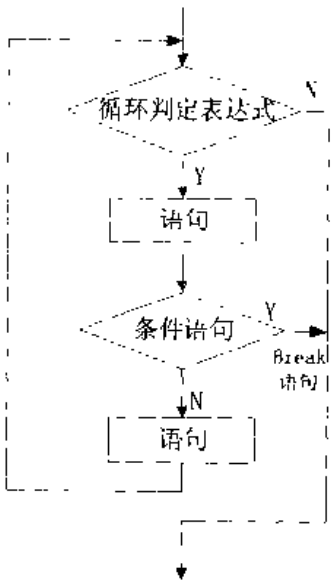


图 5.10 break 语句的执行流程

break 语句也可用于分支语句中，当 switch 结构中插入 break 语句时，程序会跳出 switch 结构，而执行分支语句以后的程序。

例： 打印输出 1~100。

```
main()
{
    int i;
```

```
    do{
        printf("%d\t",i);
        if(i--100)
        break;
        i++;
    }while(1);
}
```

在这个例子中，程序并不是通过 do…while 语句中的循环判定表达式来结束循环，而是通过循环体中插入的条件表达式和 break 语句来跳出循环。当 i<100 时，break 语句不执行，程序循环；当 i=100 时，break 语句被执行，循环终止。

2. continue 语句

continue 语句为循环继续语句，它的作用是将流程转到下一轮循环的入口，强行执行下一次循环。continue 语句只能用于 for、while 和 do…while 这 3 个循环结构中，而不能用在 switch 的分支结构中。continue 语句大都和 if 语句联用，用来在某些特殊情况下，使本轮循环停止执行而进入下一轮循环。其执行流程如图 5.11 所示。

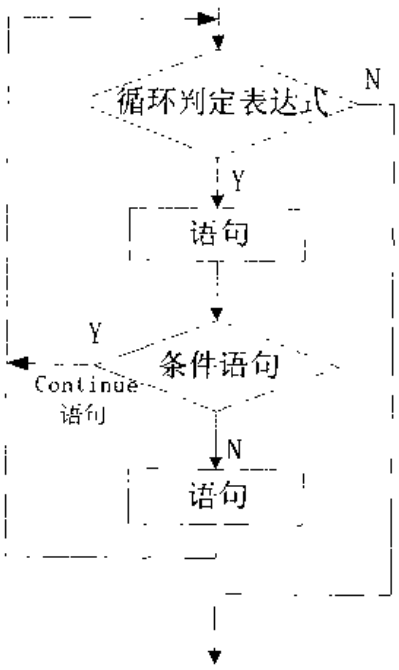


图 5.11　continue 语句的执行流程

例：处理数组 a 中的非负元素。

```
main()
{
    int a[10],i;
    for (i =0; i < 10; i++)
    {
        if (a[i] < 0)
            continue;
```

```
        a[i] = a[i] * 10;
        printf("%d\n",a[i]);
        }
}
```

这个例子中，程序通过continue语句跳过了数组中小于0的元素。与break语句不同，continue语句执行后循环并没有终止，它只是跳转到了下一轮循环的入口，继续进行处理。

第6章　函　　数

在求解一个复杂问题时，是把一个大问题分解成若干个比较容易求解的小问题，然后分别求解。在设计一个复杂的应用程序时，也是把整个程序划分为若干功能较为单一的程序模块，然后分别进行编程实现。

在C语言中，函数是程序的基本组成单位，因此可以很方便地用函数作为程序模块来实现C语言程序。C语言中的函数相当于其他高级语言的子程序。C语言不仅提供了极为丰富的库函数，还允许用户建立自己定义的函数。用户可把自己的算法编成一个个相对独立的函数模块，然后用调用的方法来使用函数。由于采用了函数模块式的结构，C语言易于实现结构化程序设计。使程序的层次结构清晰，便于程序的编写、阅读、调试。

6.1　函数的定义

使用函数的第一步就是定义函数，函数的定义涉及到函数定义的一般形式、函数的参数和函数的值。函数的定义形式有两种，分别是无参函数的定义和有参函数的定义。而函数的参数有形参和实参两种类型。

6.1.1　函数的定义的一般形式

根据函数的形式我们可以把函数分为无参函数和有参函数，所以函数的定义也有两种形式。

1. 无参函数的定义的一般形式

```
类型标识符 函数名()
{
    声明部分语句
}
```

其中类型标识符指明了本函数的类型，函数的类型实际上是函数返回值的类型。该类型标识符与前面介绍的各种说明符相同。函数名是由用户定义的标识符，函数名后有一个空括号，其中无参数，但括号不可少。

{}中的内容称为函数体。在函数体中声明部分，是对函数体内部所用到的变量的类型说明。例如：

[例6.1]：一个简单的无参函数调用的例子。

```
main()
{
```

```
    print_hello();
}
int print_hello()
{
    printf("Hello! World!");
    return 1;
}
```

运行结果：

```
Hello! World!
```

在很多情况下都不要求无参函数有返回值，此时函数类型符可以写为 void。可以将函数定义改为：

```
void print_hello()
{
    printf("Hello! World!");
}
```

这里，只把 int 改为 void 作为函数类型，将 return 语句删去。Hello 函数是一个无参函数，当被其他函数调用时，也是输出"Hello! World!"字符串。

2. 有参函数定义的一般形式

```
类型标识符 函数名(形式参数表列)
{
    声明部分语句
}
```

有参函数比无参函数多了一个内容，即形式参数。在形参表中给出的参数称为形式参数，它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。形参既然是变量，必须在形参表中给出形参的类型说明。

[例 6.2]：定义一个函数，用于求两个数中的大数。

```
main()
{
    int x,y,z;
    x=1;
    y=2;
    z=max(x,y);
    printf("%d",z);
}
int max(int a1,int a2)
{
    if(a1>a2)
        return a1;
    else
        return a2;
}
```

运行结果：

```
2
```

max 函数是一个整型函数，其返回的函数值是一个整数。形参为 a1、a2，均为整型量。a1、a2 的具体值是由主函数在调用时传送过来的。在 max 函数体中的 return 语句是把 a1(或 a2)的值作为函数的值返回给主函数。非 void 型的函数中至少应有一个 return 语句。

在 C 程序中，一个函数的定义可以放在任意位置，既可放在主函数 main 之前，也可放在 main 之后。可把 max 函数置在 main 之后，也可以把它放在 main 之前。可以将程序修改成如下所示：

[例 6.3]：

```
int max(int a1,int a2)
{
    if(a1>a2)
        return a1;
    else
        return a2;
}
main()
{
    int max(int a1,int a2);
    int x,y,z;
    printf("input two numbers:\n");
    scanf("%d,%d",&x,&y);
    z=max(x,y);
    printf("%d",z);
}
```

运行结果：

```
1，2↙
2
```

从函数定义、函数声明及函数调用来分析整个程序。程序的前 7 行为 max 函数定义。进入主函数后，因为准备调用 max 函数，所以先对 max 函数进行说明。函数定义和函数声明并不是一回事，在后面还要专门讨论。可以看出函数声明与函数定义中的函数头部分相同，但是末尾要加分号，它应被看作一条语句。程序第 14 行为调用 max 函数，并把 x、y 中的值传送给 max 的形参 a1，a2。max 函数执行的结果(a1 或 a2)将返回给变量 z。最后由主函数输出 z 的值。

6.1.2　函数的参数

函数的参数分为形参和实参两种。一般情况下，在调用有参函数时，主函数和被调函数之间存在着数据传递关系。函数定义时填入的参数称为形式参数，简称形参。形参的定义是在函数名之后和函数开始的花括号之前。有参函数被调用时填入的参数，称为实际参数，简称实参。必须确认所定义的形参与调用函数的实际参数类型和出现的次序一致。如果不一致，将产生意料不到的错误。

形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。实参出现在主调函数中，进入被调函数后，实参变量也不能使用。形参和实参的功能是作为数据传送。发生函数调用时，主函数把实参的值传送给被调函数的形参，从而实现主函数向被调函数的数据传送。

函数的形参和实参具有以下特点：

- 形参只在函数内部有效。形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，函数调用结束返回主调函数后则不能再使用该形参变量。
- 实参应预先用赋值，使用前实参必须获得确定值。实参可以是常量、变量、表达式或函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形参。
- 形参的类型在定义时必须确定。
- 实参和形参在数量上、类型上、顺序上应严格一致，否则会发生意想不到的错误。
- 函数调用中发生的数据传送是单向的。即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

[例 6.4]：求两数之和。

```
main()
{
    int a1,a2,a3;
    printf("input two numbers\n");
    scanf("%d,%d",&a1,&a2);
    a3=sum(a1,a2);
    printf("a3=%d\n",a3);
}
int sum(int x,int y)
{
    int z;
    z=x+y;
    return z;
}
```

运行结果：

```
10,20↙
a3=30↙
```

本程序中定义了一个函数sum，该函数的功能是求和。在主函数中输入两个整数，并作为实参，在调用时传送给sum函数的形参量x，y。从运行情况看，输入a1值为10，输入a2值为20，即实参a1、a2的值分别为10、20。把此值传给函数x、y时，形参的初值也为10、20，在执行函数过程中，z的值变为30。返回主函数之后，a3的值为30。

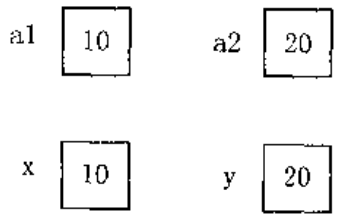


图 6.1 变量的值

6.1.3 函数的值

所有的函数，除了空值类型外，都返回一个数值。函数的值是指函数被调用时，执行函数体中的程序段所取得的并返回给主函数的值。例如调用的 max 函数取得的最大数等。该数值由返回语句确定；无返回语句时，返回值是 0 。对函数的值有以下一些说明：

函数的值只能通过 return 语句返回主调函数。

return 语句的一般形式为：

```
return 表达式;
```

或者为：

```
return (表达式);
```

该语句的功能是计算表达式的值，并将被调函数中的一个确定值带回给主函数中去。如果需要从被调函数返回一个函数值，被调函数中必须包含 return 语句；如果不需要返回值，可以不要 return 语句。在函数中允许有多个 return 语句，但每次调用只能有一个 return 语句被执行，因此只能返回一个函数值。

有的函数有返回值，则这个函数值一定属于某种确定的数据类型，这个函数值的类型在函数定义时就确定下来了。例如：

```
int sum(x,y)
float max(x,y)
char c(c1,c2)
```

上面 3 个函数的类型分别为整型、实型和字符型。

函数值的类型和函数定义中函数的类型应保持一致。如果两者不一致，则以函数类型为准，自动进行类型转换。

[例 6.5]：求两数之和。

```
main()
{
    int a1,a2,a3;
    printf("input two numbers\n");
    scanf("%d,%d",&a1,&a2);
    a3=sum(a1,a2);
    printf("a3=%d\n",a3);
}
int sum(x,y)
{
    float x,y;
    float z;
    z=x+y;
    return z;
}
```

运行结果：

```
10.1,20.5↙
    a3=30↙
```

函数 sum 被定义为整型，而 return 语句的返回变量为实型，二者不一致。首先将 z 转换为整型，然后 sum 函数返回一个整型数 30。

不返回函数值的函数，可以定义为“空类型”，类型说明符为 void。例如：

```
void a(int i)
{
.....
}
```

一旦函数被定义为空类型后，就不能在主调函数中使用被调函数的函数值了。例如，在定义函数 a 为空类型后，在主函数中写下述语句

```
a1=a(i);
```

就是错误的。为了使程序有良好的可读性并减少出错，凡不要求返回值的函数都应定义为空类型。如果函数值为整型，在函数定义时可以省去类型说明。

6.2 函数的调用

函数的定义完成后，就可以调用函数了，本节也将从 3 个方面进行说明，首先介绍函数调用的一般形式和函数的声明等内容，然后介绍函数调用的特殊形式：递归调用和嵌套调用。

6.2.1 函数的一般调用

1. 函数调用的一般形式

函数体的执行是通过对函数的调用实现的，函数的执行过程与其他语言的子程序调用相似。

函数调用的一般形式为：

函数名(实参列表)

调用无参函数时则无实参列表，但括号不能省略。实参列表中的参数可以是常数，变量或其他类型的数据和表达式。实参和形参的类型必须一致，个数必须相同，各参数的次序也应保持一致。各实参之间用逗号分隔。

2. 函数调用的方式

在C语言中，按照函数在程序中出现的位置，函数的调用方式可以分为以下几种形式：

- 函数表达式

  函数可以作为表达式中的一部分出现在表达式中，以函数的返回值作为表达式的一项。这种方式要求函数有一个确定的返回值参加表达式的运算。例如：

z=100-sum(x,y)是一个赋值表达式，把 100-sum(x,y)的返回值赋予变量 z。

- 函数语句

  可以把函数调用作为一个语句，函数调用的一般形式加上分号即构成函数语句。例如：

```
printf("%s",c1);
scanf("%d",&a1);
```

  都是以函数语句的方式调用函数。

- 函数实参

  函数可以作为另一个函数的参数的位置出现，作为函数的实参。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数也是必须有返回值。例如：

```
printf("%d",sum(x,y));
```

  此语句是把 sum 调用的返回值又作为 printf 函数的实参来使用的。

对实参列表求值的顺序并不是确定的，有的系统是自左至右，有的系统是自右至左。在函数调用中应该注意的一个问题就是求值顺序。我们用前面介绍的 printf 函数说明这一点。

[例 6.6]：

```
main()
{
    int i=5;
    printf("%d\n%d\n%d\n%d\n",--i,++i,i--,i++);
}
```

如果按照从右至左的顺序求值，运行结果应为：

```
5
6
6
5
```

如果按照从右至左的顺序求值，运行结果应为：

```
4
5
5
4
```

应特别注意的是，无论是从左至右求值，还是自右至左求值，其输出顺序都是不变的，即输出顺序总是和实参表中实参的顺序相同。

3. 函数的声明

在主函数中调用某函数之前应对该被调函数进行声明，这与使用变量之前要先进行变量声明是一样的。在主函数中对被调函数进行声明的目的是使编译系统知道被调函数返回值的类型，以便在主函数中按此种类型对返回值做相应的处理。

在一个函数调用另一个函数时，被调用函数必须是已经存在的函数。如果被调函数是

用户自定义的函数，而且两函数属于同一个文件，一般应在主调函数中对被调函数的值的类型进行声明，其一般形式为：

```
类型说明符 被调函数名(类型 形参，类型 形参.....);
```

或者为：

```
类型说明符 被调函数名(类型，类型.....);
```

括号内给出了形参的类型和形参名，或只给出形参类型。

[例 6.7]：

```
main()
{
    int sum(int,int);
    int a1,a2,a3;
    printf("input two numbers\n");
    scanf("%d,%d",&a1,&a2);
    a3=sum(a1,a2);
    printf("a3=%d\n",a3);
}
int sum(int x,int y)
{
    int z;
    z=x+y;
    return z;
}
```

运行结果：

```
intput two numbers↙
1，2↙
a3=3↙
```

程序中的第三行int sum(int,int)就是对被调用函数的声明。这是一个很简单的函数调用，函数 sum 的作用是求出两个整型数的和。函数的定义和函数的声明是两个完全不同的概念，函数的定义是对函数功能的确立，包括函数名、函数类型和函数体等。而函数的声明仅仅是在主调函数中对被调函数的返回值的类型进行说明。

如果使用库函数，一般应在文件开头使用#include 命令将有关库函数包含在本文件中。例如：

```
#include"math.h"
```

其中 math.h 是一个头文件，.h 是头文件所用的后缀，标识头文件。

在 C 语言中并不是所有的情况都需要进行函数声明，在以下几种情况时可以不必在主调函数中对被调函数的函数进行声明。

- 如果被调函数的返回值的类型是整型或字符型时，可以不对被调函数进行声明，而直接调用。这时系统将自动对被调函数返回值按整型处理。【例 6.2】的主函数中未对函数 s 进行说明而直接调用即属此种情形。
- 当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函

数再声明而直接调用。例如【例 6.3】中，函数 max 的定义放在 main 函数之前，因此可在 main 函数中省去对 max 函数的函数说明。

- 如果在所有函数定义之前，在函数外预先说明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数进行说明。例如：

```
int f1(int a);
float f2(int b);
main()
{
……
}
int f1(int a)
{
……
}
float f2(int b)
{
……
}
```

程序的第一、二行对 f1 函数和 f2 函数预先进行了声明，因此在以后各函数中不需要对 f1 和 f2 函数再进行声明就可直接调用。

6.2.2　函数的递归调用

函数的递归调用就是在调用一个函数的过程中又直接或者间接地调用函数本身。在递归调用中，主调函数同时又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层。C 语言允许函数的递归调用，例如：

```
int f(int a)
{
    int b,c;
    ……
    b=2*f(c)
    ……
    return b;
}
```

上面的程序表示在调用函数 f 的过程中，又要调用它本身，这样称为直接调用本函数。

还有一种函数的递归调用情况是间接调用，假设有两个自定义函数 f1、f2，在调用 f1 函数的过程中调用 f2 函数，而在调用 f2 函数的过程中又要调用 f1 函数。例如：

```
main()
{
    ……
}
int f1(int a1)
{
```

```
    int b1,c1;
    ……
    b1=f2(c1);
    ……
    return b1;
}
int f2(int a2)
{
    int b2,c2;
    ……
    b2=f1(c2);
    ……
    return b2;
}
```

前面两个简单的例子中的函数都是一个递归函数。但是运行该函数将无休止地递归调用，这样就像一个死循环，显然是不正确的。为了防止递归调用无休止地进行，必须在函数内加上终止递归调用的手段。常用的办法是加条件判断，满足某种条件后就不再递归调用，然后逐层返回。

[例6.8]求$n!$。

可以用递归法计算n的阶乘，即$n!=n*(n-1)!$，$(n-1)!=(n-1)(n-2)!\cdots\cdots 1!=1$。可用下述公式表示：

$$n!=\begin{cases}1 & (n=0,1)\\ n*(n-1)! & (n>1)\end{cases}$$

可编程如下实现阶乘：

```
main()
{
    long fn(int);
    int n;
    long result;
    printf("input a inteager number:\n");
    scanf("%d",&n);
    y=fn(n);
    printf("%d!=%ld",n,result);
}
long fn(int n)
{
    long m;
    if(n<0)
        printf("n<0,input error");
    elseif(n==0||n==1)
        m=1;
    else
        m=fn(n-1)*n;
    return(m);
}
```

程序中给出的函数 fn 是一个递归函数。主函数调用 fn 后即进入函数 fn 执行，如果 $n<0$、$n==0$ 或 $n=1$ 时都将结束函数的执行，否则就递归调用 fn 函数自身。由于每次递归调用的实参为 $n-1$，即把 $n-1$ 的值赋予形参 n，最后当 $n-1$ 的值为 1 时再做递归调用，形参 n 的值也为 1，将使递归终止，然后可逐层退回。

上面阶乘算法还可以用递推方法，即从 1 开始连乘，一直乘到 n。这种方案很容易实现，但是有些问题则只能用递归算法才能实现，典型的问题是 Hanoi 塔问题。

[例 6.9] Hanoi 塔问题。

一块板上有 3 根圆柱 A、B、C。A 柱上套有 64 个大小不等的盘子，大的在下，小的在上。要把这 64 个圆盘从 A 移动到 C 上，每次只能移动一个圆盘，移动可以借助 B 进行。但在任何时候，任何圆柱上的圆盘都必须保持大盘在下、小盘在上，求出移动的步骤。

本题算法进行如下分析：设 A 上有 n 个盘子，如果 $n=1$，则将圆盘从 A 直接移动到 C；如果 $n=2$，则：

(1) 首先将 A 上的 1 个盘子移到 B 上；

(2) 然后将 A 上的另一个圆盘移到 C 上；

(3) 最后将 B 上的 1 个圆盘移到 C 上。

如果 $n=3$，则：

(1) 将 A 上的两个圆盘移到 B，步骤如下：

　　首先将 A 上的 1 个圆盘移到 C 上。

　　然后将 A 上的另一个圆盘移到 B 上。

　　最后将 C 上的 1 个圆盘移到 B 上。

(2) 将 A 上的一个圆盘移到 C。

(3) 将 B 上的两个圆盘移到 C，步骤如下：

　　首先将 B 上的 1 个圆盘移到 A 上。

　　然后将 B 上的另一个盘子移到 C 上。

　　最后将 A 上的 1 个圆盘移到 C 上。

至此，完成了 3 个圆盘的移动过程。

从上面分析可以看出，当 n 大于等于 2 时，移动的过程可分解为 3 个步骤：

(1) 把 A 上的 $n-1$ 个圆盘移到 B 上。

(2) 把 A 上的一个圆盘移到 C 上。

(3) 把 B 上的 $n-1$ 个圆盘移到 C 上；其中第(1)步和第(3)步是雷同的。

当 $n=3$ 时，第(1)步和第(3)步又分解为雷同的 3 步，即把 $n-1$ 个圆盘从一个圆柱移到另一个圆柱上，这里的 $n=n-1$。显然这是一个递归过程，据此算法可编程如下：

```
main()
{
    int i;
    printf("input a number:\n");
    scanf("%d",&i);
    printf("the step to moving %d diskes:\n",i);
    move(i,'a','b','c');
}
```

```
void hanoi(int n,char a,char b,char c)
{
    if(n--1)
        printf("%c->%c\n",a,c);
    else
    {
        move(n-1,a,c,b);
        printf("%c->%c\n",a,c);
        move(n-1,b,a,c);
    }
}
```

运行结果：

```
input a number: ↙
3
the step to moving %d diskes: ↙
a→c
a→b
c→b
a→c
b→a
b→c
a→c
```

从程序中可以看出，hanoi 函数是一个递归函数，它有 4 个形参 *n*、*a*、*b*、*c*。*n* 表示盘子数，*a*、*b*、*c* 分别表示 3 根圆柱。hanoi 函数的功能是把 *a* 上的 *n* 个盘子移动到 *c* 圆柱上。当 *n*==1 时，直接把 *a* 上的圆盘移至 *c* 上，输出 *a*—>*c*。如果 n=1 则可分为 3 步：递归调用 hanoi 函数，把 *n*-1 个圆盘从 *a* 移到 *c*；输出 *a*—>*c*；递归调用 hanoi 函数，把 *n*-*1* 个圆盘从 *b* 移到 *c*。在递归调用过程中 *n*=*n*-1，故 *n* 的值逐次递减，最后 *n*=1 时，终止递归，逐层返回。

6.2.3 函数的嵌套调用

C 语言中不允许进行嵌套的函数定义，各函数的定义是平行的、独立的，不存在上一级函数和下一级函数的问题。

虽然 C 语言不允许嵌套定义，但是 C 语言允许在一个函数的定义中出现对另一个函数的调用，这样就出现了函数的嵌套调用，即在被调函数中又调用其他函数。其关系可表示为图 6.2 所示。

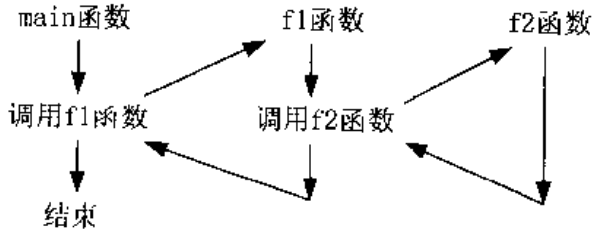


图 6.2 函数调用关系

图 6.2 表示了两层嵌套的情形。其执行过程是：程序首先从 main 开始执行，执行 main 函数时运行到调用 f1 函数的语句时，即转去执行 f1 函数；在 f1 函数中调用 f2 函数时，程序又转去执行 f2 函数，f2 函数执行完毕返回 f1 函数的断点继续执行，f1 函数执行完毕返回 main 函数的断点继续执行，直至结束。

[例 6.10] 计算 $c=(a!)^2+(b!)^2$(a、b 是两个输入的正整数)。

本题可以编写两个函数，一个是用于计算平方值的函数 f1，另一个是用于计算阶乘值的函数 f2。主函数先调用 f1 函数，再在 f1 函数中调用 f2 函数，调用 f2 计算其阶乘值，然后返回 f1，计算平方值，再返回主函数，在循环程序中计算累加和。

```
long f1(int n1)
{
    long r2;
    long f2(int);
    r2=f2(n1)*f2(n1);
    return r2;
}
long f2(int n2)
{
    long r1=1;
    int i;
    for(i=1;i<=n2;i++)
    r1=r1*i;
    return r1;
}
main()
{
    int i;
    int a,b;
    long r;
    printf("intput two numbers:\n");
    scanf("%d,%d",&a,&b);
    r=f1(a)+f2(b);
    printf("result=%ld\n",r);
}
```

运行结果：

```
input two numbers: ✓
2, 3✓
result=40✓
```

6.3　变量的类型及其存储方式

如果从变量的作用域角度来分，变量可以划分为全局变量和局部变量；如果从变量的存在时间的角度来分，变量可以划分为静态存储变量和动态存储变量。

6.3.1 局部变量

所谓局部变量就是在一个函数内部定义的变量，也称之为内部函数。它的作用域是有限的，它只能在本函数内才能使用，也就是说它仅能在函数内有效。例如：

```
int f1(int a,int b)
{
    int c;
    ……
}
/*变量a、b、c有效*/
int f2(int x)
{
    int y,z;
    ……
}
/*变量x、y、z有效*/
main()
{
    int i,j;
    ……
}
/*变量i、j有效*/
```

在函数f1内定义了3个变量，*a*、*b*为形参，*c*为一般变量。在f1的范围内变量*a*、*b*、*c*均有效，或者说变量*a*、*b*、*c*的作用域仅限于函数f1内。同理，变量*x*、*y*、*z*的作用域仅限于f2内，变量*m*、*n*的作用域仅限于main函数内。关于局部变量的作用域还要说明以下几点：

主函数中定义的变量(如*i*、*j*)只能在主函数中使用，不能在其他函数中使用。同时，主函数中也不能使用其他函数中定义的变量。因为主函数也是一个函数，它与其他函数是平行关系，它的特殊之处在于整个程序从它开始执行。

形参变量也是局部变量，它属于被调函数的局部变量，而实参变量属于主调函数的局部变量。

同一变量名在不同的函数中可以重新定义，即C语言允许在不同的函数中使用相同的变量名，它们代表不同的对象，分配不同的地址单元，互不干扰，也不会发生混淆。在复合语句中也可定义局部变量，其作用域只在复合语句范围内，并不是在此复合语句所在的函数内。例如：

```
main()
    {
    int a,r;
    ……
{
    int b;
    b--1;
    r=a*b;
```

```
    ……
{
……
}
```

变量 *b* 就是定义在复合语句内局部变量，它的作用域仅在复合语句内。

[例 6.11]

```
main()
{
    int a=2,b=3,r;
    r=i+j;
    {
    int r=10;
    printf("%d\n",c);
    }
    printf("%d\n",c);
}
```

执行结果：

```
10↙
5↙
```

本程序在 main 中定义了 *a*、*b*、*c* 三个变量，其中 c 未赋初值。而在复合语句内又定义了一个变量 *c*，并赋初值为 8。这两个 *c* 同名变量并不是同一个变量。在复合语句外由 main 定义的 *c* 起作用，而在复合语句内则由在复合语句内定义的 *c* 起作用。因此，第一个 printf 语句中的 *c* 是复合语句中定义的变量，被赋值为 10，所以第一个打印结果为 10。而第二个 printf 函数中的变量 c 是 main 函数定义的，并赋值为 5，所以第二个打印结果是 5。

6.3.2　局部变量的存储方式

内存中供用户使用的存储空间可以分为 3 部分(见图 6.3)：

- 程序区
- 静态存储区
- 动态存储区

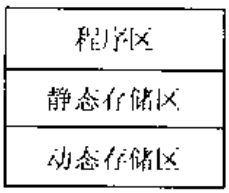


图 6.3　存储空间

数据分别存放在静态存储区和动态存储区中。全局变量存放在静态存储区中，在程序开始执行时给全局变量分配存储区，直到程序执行完成后释放。在程序执行的过程中全局变量占固定的内存区域，而不是在程序的执行过程中分配和释放。而函数中的局部变量一般都是存储在动态存储区中，都是动态分配存储空间。局部变量定义的形式一般都省略了关键字 auto，例如：

```
int a;
```

此语句实际上是语句“auto int a”；的缩写，这类变量又称为自动变量。例如：

```
int f()
```

```
{
    auto int b,c;
    ……
}
```

程序定义了两个整型的自动变量 b、c。关键字 auto 可以省略，如果不写，程序就会默认为定义自动变量。例如：

```
auto int a;
int a;
```

两语句完全等价。

有时，局部变量的值在函数调用结束后还希望保留其原值，即其占用的空间不释放。当函数被再次调用时，该局部变量仍有值，这时称该局部变量为静态局部变量。定义格式为：

```
static a=1;
```

[例 6.12]

```
f(int x)
{
    int y=0;
    static int z=1;
    y=y+1;
    z=z+1;
    return(x+y+z);
}
main()
{
    int x=2,i;
    for(i=0;i<3;i++)
    printf("%d",f(x));
}
```

运行结果：

```
5 6 7
```

在主函数的 for 循环中，第 1 次调用 f 函数，函数开始时，y 的值为 0，z 的值为 1。函数调用完成后，y 的值仍为 0，而 z 的值为 2。第 2 次调用结束后，z 的值为 3，而 y 的值仍为 0，这就是静态局部变量和自动局部变量的区别。

对静态局部变量和自动局部变量的说明：

- 静态局部变量在编译时赋初值，即只赋初值一次；而对自动局部变量赋初值是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。
- 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动局部变量属于动态存储类别，占动态存储空间，函数调用结束后即释放。
- 如果在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值 0 或空字符。而对自动局部变量来说，如果不赋初值则它的值是一个不确定

的值。

- 虽然静态存储变量所占的内存在函数调用完成后不释放，但不允许其他函数引用它。

C 语言还规定，只有在定义静态局部变量和全局变量时才能对数组进行初始化。变量的值一般是存放在内存中的。为了提高程序的效率，C 语言允许将局部变量的值存放在运算器的寄存器中，这种变量被称为寄存器变量，用关键字 register 声明。

[例 6.13]使用寄存器变量。

```
int m(int n)
{
    register int i,r=1;
    for(i=1;i<=n;i++)
        r=r*i
    return(r);
}
main()
{
    int i;
    for(i=0;i<=5;i++)
    printf("%d!=%d\n",i,fac(i));
}
```

说明：

- 只有局部自动变量和形式参数可以作为寄存器变量，全局变量不能定义为寄存器变量。
- 一个计算机系统中的寄存器数目有限，不能定义任意多个寄存器变量。
- 局部静态变量不能定义为寄存器变量。

6.3.3 全局变量

在函数的内部定义的变量是局部变量，在函数外部定义的变量称为外部变量，又称为全局变量。全局变量在函数外定义，因此可以在许多函数中使用。所有通过变量名对全局变量与函数的引用都是引用的同一对象。例如：

```
int a,b;
int f1()
{
  ……
}
float x,y;
float f2()
{
  ……
}
main()
{
  ……
}
```

在上面的程序中，定义了 a、b、x 和 y 共 4 个全局变量，但是它们的作用域不同。在函数 f1 中，仅有 a 和 b 有效。而在 f2 中 4 个全局变量都有效。如果全局变量在文件的开头定义，则它在整个文件的范围内都有效，如果不在文件的开头定义，则其作用域仅限于定义点到文件末尾。如果定义前的函数引用全局变量，则应在函数中用关键字 extern 作为全局变量声明。例如：

```
int a(x,y)
int x,y;
{
……
}
main()
{
    extern int x1,y1;
    ……
    printf("%d",a(x1,y1);
}
int x1=1,y1=1;
```

由于全局变量定义在主函数之后，因此在主函数引用外部函数 x1、y1 前，应该用关键字 extern 进行声明，说明 x1、y1 是全局变量，如果没有此声明，则编译出错。

由于全局变量是可以全局访问的，这就为在函数之间交换数据提供了一种可以代替函数变元与返回值的方法。任何函数都可以用变量名来访问全局变量，只要这个变量名已经做了声明。如果要在函数之间共享大量的变量，那么使用全局变量就会更方便、更有效。然而，这样使用必须充分小心，因为这样可能对程序结构产生不好的影响，而且可能会使程序在各个函数之间产生太多的数据联系。

全局变量的用途还表现在它们比内部变量有更大的作用域和更长的生存期。自动变量只能在函数内部使用，当其所在函数被调用时开始存在，当函数退出时消失。而全局变量在本文件中是永久存在的，它们的值在从一次函数调用到下一次函数调用之间保持不变。因此，如果两个函数必须共享某些数据，而这两个函数都互不调用对方，那么最为方便的方法是，把这些共享数据作成全局变量。

[例 6.14] 输入 10 个同学的年龄，编写一个函数，求出他们年龄的最大值、最小值和平均值。

```
int age_max=0;
int age_min=0;
float average(float a[],int n)
{
    int i;
    float aver;
    float sum=a[0];
    max=min=a[0];
    for(i=1;i<n;i++)
    {
        if(a[i]>max)
            max=a[i];
        else if(a[i]<min)
```

```
            min=a[i];
        sum=sum+a[i];
    }
    aver=sum/n;
    return aver;
}
main()
{
    int age[10];
    float ave;
    int j;
    for(j=0;j<10;j++)
        scanf("%d",&age[i]);
    ave=average(a,10);
    printf("max=%d\nmin=%d\naverage=%6.2f\n",max,min,ave);
}
```

运行结果：

```
21 22 25 19 21 20 24 21 23 2↙
max=25↙
min=19↙
average=21.70
```

当函数的返回值多于一个时(如上例需要 3 个返回值)，就可以利用全局变量。函数 average 仅仅将年龄的平均值作为返回值，而最大最小值是通过全局变量输出的。

虽然全局变量可以解决一些问题，但它也存在一些缺点。由于全局变量在整个程序的执行过程中都占用内存单元，所以比较浪费内存。它同时降低了函数的通用性，如果一个函数中使用了全局变量，此函数如果移到另外一个文件中，就会引起错误。

另外，如果在同一个文件中，全局变量和局部变量同名，则在局部变量的作用域内，全局变量不起作用。

[例 6.15] 全局变量与局部变量同名。

```
int a1=2,a2=3;
max(int a1,int a2)
{
    int b;
    if(a1>a2)
        b=a1;
    else
        b=a2;
    return(b);
}
main()
{
    int a1=6;
    printf("%d\n",max(a,b));
}
```

运行结果：

```
6↙
```

上例中的自定义函数的作用是求出两数中的大者。变量 $a1$ 被定义了两次，一个是全局变量，一个是局部变量。主函数中 printf 语句出在局部变量 $a1$ 的作用域内，所以全局变量 $a1$ 失效，所以程序运行结果是 6，而不是 3。

6.3.4 全局变量的存储方式

全局变量是在函数外部定义的变量，编译时即分配在静态存储区。全局变量可以为程序中各个函数所用。

如果程序有一个源文件组成，使用全局变量的方法前面的章节已经介绍。一个程序还可以由多个源文件组成，那么，一个文件中的函数引用另一个文件中的全局变量有两种情况：

1. 全局变量只被本源文件引用

在一些程序中，一些全局变量仅仅希望被本文件引用而不希望文件引用。这时，只需在全局变量定义语句的前面加关键词 static 加以声明。例如：

```
file1.c
    static int a;
    …
    main()
    {
    …
}

file2.c
    extern int a;
    int f(int c)
    {
        …
        a=b*c;
        …
}
```

上例中在第 1 个文件中定义了全局变量 a，由于有关键词 static 声明，因此该全局变量只能在第 1 个文件中有效。虽然第二个文件中用 extern int a 语句声明了变量 a，但第二个文件还是不能使用第一个文件中的全局变量 a。

2. 全局变量可以被其他源文件引用的情况

如果在一个文件中需要引用另一个文件中定义的全局变量，给全局变量在定义时应用关键词 extern 声明。如下例：

[例 6.16] 设 a 为定值，输入 b 和 c，求出 $a*b$ 和 a^c。

文件 1 的内容：

```
int b;
main()
{
    int p();
    int a=3;
    int b,d,e;
    print("input the number b and c:\n");
    scanf("%d,%d",&b,&c);
    e=a*b;
    printf("%d*%d=%d\n",a,b,e);
    d=p(c);
    printf("%d\n",d);
}
```

文件 2 的内容：

```
extern int b;
p(int n)
{
    int i,y=1;
    for(i=0;i<n;i++)
        y=y*b;
    return y;
}
```

运行结果：

```
intput the number b and c:↙
2,2↙
3*2=6↙
9↙
```

文件 2 的开头有一个 extern 声明，它说明了在本文件中出现的全局变量 b 是一个已经在其他文件中定义过的全局变量。由于在文件 2 中使用关键词 extern 对变量 *b* 进行了声明，所以在此文件中可以直接使用，不需定义，即在文件 2 中引用变量 *b* 不需要再分配内存。

静态局部变量和静态全局变量同属静态存储方式，但两者区别较大：

- 定义的位置不同。静态局部变量在函数内定义，静态全局变量在函数外定义。
- 作用域不同。静态局部变量属于内部变量，其作用域仅限于定义它的函数内；虽然生存期为整个源程序，但其他函数是不能使用它的。静态全局变量在函数外定义，其作用域为定义它的源文件内；生存期为整个源程序，但其他源文件中的函数也是不能使用它的。
- 初始化处理不同。静态局部变量，仅在第 1 次调用它所在的函数时被初始化，当再次调用定义它的函数时，不再初始化，而是保留上 1 次调用结束时的值。而静态全局变量是在函数外定义的，不存在静态内部变量的“重复”初始化问题，其当前值由最近一次给它赋值的操作决定。

6.4 内部函数和外部函数

在一个源文件中，所有的函数都是全局的，它可以被任何其他的函数调用。而相对于多个源文件，函数可以被分为两类：内部函数和外部函数。

6.4.1 内部函数

如果函数只能被本源文件的函数调用，则称此函数为内部函数，又称之为静态函数。在定义内部函数时，给函数定义前面加上关键字static。形式如下：

```
static int f(a,b);
```

这个 f 函数只能在本源文件中使用。有了内部函数的概念后，在不同的源文件中可以有相同的函数名而不会发生冲突。

6.4.2 外部函数

如果函数不仅能被本源文件的函数调用，还可以被其他源文件中的函数调用，则称此函数为外部函数。在定义内部函数时，给函数定义前面加上关键字extern。定义形式如下：

```
extern int f(a,b);
```

这个 f 函数所有源文件中使用。要注意的是，如果在一个源文件中调用另一个源文件中的函数，那么必须在第一个源文件中对要调用的函数进行说明，格式如下：

```
extern int f(a,b);
```

[例6.17] 外部函数应用。

文件1

```
main()
{
    ……
    extern void f2(int,int)
    extern void f3(int,int);
    extern void f4(int,int);
    f2(c2,d2);
    f3(c3,d3);
    f4(c4,d4);
}
```

文件2

```
……
extern void f2(int a2,int b2)        /*定义外部函数*/
    {
```

```
    ……
}
```

文件 3

```
……
extern void f3(int a3,int b3)        /*定义外部函数*/
    {
    ……
}
```

文件 4

```
……
extern void f4(int a4,int b4)        /*定义外部函数*/
    {
    ……
}
```

另外要注意的是在定义外部函数的时候，extern 关键字可以省略。

本章主要介绍了函数的定义、函数的使用。函数可以进行嵌套调用和递推调用，从而可以实现一些复杂的运算。同时，本章还介绍了局部变量和全局变量及其存储方式。

第7章 指　　针

指针是C语言的精华部分，通过利用指针，可以很好地利用内存资源，使其发挥最大的效率。指针是用来操纵地址的特殊类型变量，有了指针技术，我们可以更方便地处理数组，可以作为函数参数，也可以用于内存访问和堆内存操作。总之，指针对于进行C程序设计是至关重要的。

指针的概念比较复杂，使用也比较灵活，因此学习时经常出错，务请在学习时多思考、多练习。

7.1 指针和指针变量

为了说明什么是指针，首先必须弄清楚数据在内存中是如何存储的，又是如何读取的。我们在编程中定义或说明变量，编译系统就为变量分配相应的内存单元，也就是说，每个变量在内存中有固定的大小，有具体的地址。例如，整型变量分配2个字节，实型变量分配4个字节，字符型变量分配1个字节等。内存区的每一个字节有一个编号，这就是“地址”。

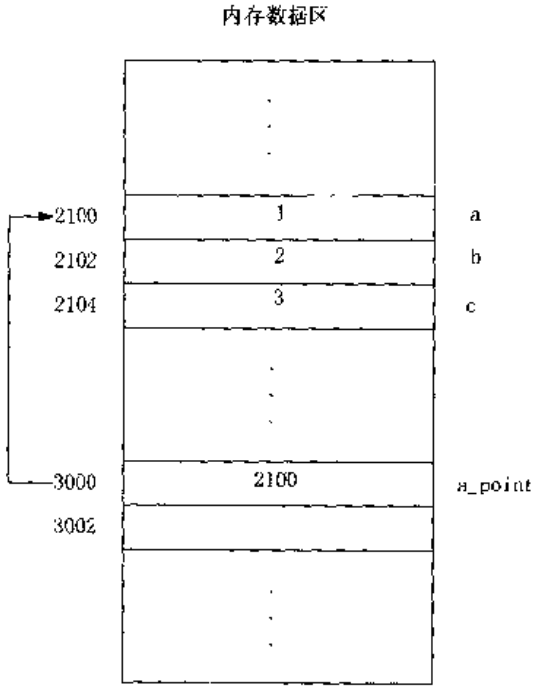


图7.1 内存单元说明

内存单元地址和内存单元的内容是两个完全不同的概念，如图7.1所示。假设程序已

定义了 3 个整型变量 a、b、c，编译时系统分配 2100 和 2101 字节给变量 a，2102 和 2103 字节给变量 b，2104 和 2105 字节给变量 c。在内存中对变量值的存取都是通过地址进行的。在程序的执行过程中，根据变量名与内存地址的对应关系，找到变量的地址，然后从此地址取出存储的数据，即变量的值，这种变量地址存取变量值的方式称为“直接访问”方式。一个地址惟一指向一个内存变量，所以访问变量时，首先应找到其在内存的地址。我们称变量所占内存的首地址为变量的指针。如果将变量的地址保存在内存的特定区域，用变量来存放这些地址，这样的变量就是指针变量，这种通过指针对所指向变量的访问称为“间接访问”方式。如图 7.1 所示，将变量 a 的地址存放在另一个内存单元(3000、3001)，假设定义了变量 i_pointer 是存放整型变量的地址的，它被分配为 3000、3001 字节。可以通过赋值语句 a_point＝&a 将 a 的地址存放到 a_point 中。这时，a_point 的值就是 2100，即变量 a 所用单元的首地址。

为了表示将数值 1 送到变量 a 中，可以有两种表达方法：(1)将 1 送到变量 a 所占的单元中，如图 7.2 上图。(2)将 1 送到变量 a_point 所指向的单元中，如图 7.2 下图。

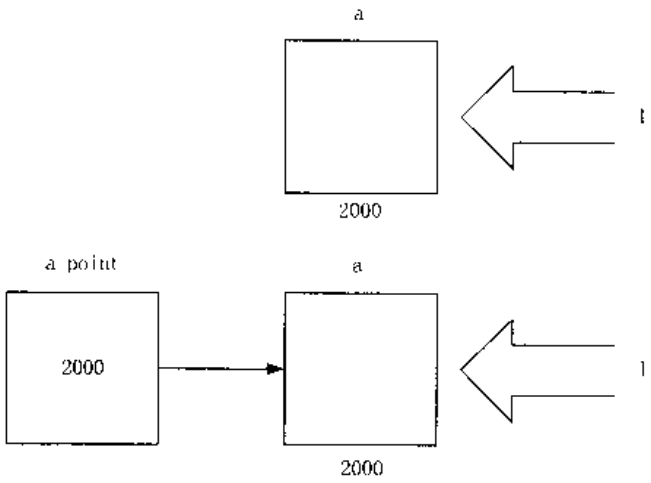


图 7.2　变量的赋值

7.2　指针变量的定义和引用

本节将对指针变量的使用进行说明，首先介绍指针变量的定义，然后对指针变量的引用进行说明，最后对指针变量作为函数参数的情况进行详细介绍。

7.2.1　指针变量的定义

变量的指针就是变量的地址，如果一个变量专门用来存放另一个变量的指针，则称它为指针变量。在 C 程序中，存放地址的指针变量使用前必须定义，指针变量不同于其他类型的变量，它是用来专门存放地址的。

```
int *a1;
```

```
float *a2;
char *a3;
```

表示定义了3个指针变量a1、a2、a3。a1可以指向一个整型变量，a2可以指向一个实数型变量，a3可以指向一个字符型变量，换句话说，a1、a2、a3可以分别存放整型变量的地址、实型变量的地址、字符型变量的地址。

定义了指针变量，我们才可以写入指向某种数据类型的变量的地址，或者说是为指针变量赋初值：

```
a1=&a;
```

上述赋值语句表示将变量a的地址赋给指针变量a1，此时a1就指向a。

定义指针变量的一般形式为：

```
类型标识符    *标识符;
```

标识符就是指针变量的名字。

在定义指针变量时要注意两点：

- 标识符前面的“*”表示该变量为指针变量，指针变量为a1，而不是*a1。
- 一个指针变量只能指向同一个类型的变量。

7.2.2 指针变量的引用

利用指针变量，是提供对变量的一种间接访问形式。对指针变量的引用形式为：

```
*指针变量
```

其含义是指针变量所指向的值。

指针变量中只能存放地址，不能将一个整型量赋给一个指针变量。下面的赋值方式是不合法的：

```
a1=2000;
```

有两个与指针有关的运算符：&和*。&是取地址运算符；*是指针运算符。例如：&a为变量a的地址，*a1为指针变量a1所指向的变量。

[例7.1]

```
main()
{
    int x,y;
    int *ptr1,*ptr2;
    x=1;
    y=2;
    ptr1=&x;
    ptr2=&y;
    printf("%d,%d\n",a,b);
    printf("%d,%d\n",*ptr1,*ptr2);
}
```

运行结果为：

```
1, 2
1, 2
```

程序说明：

程序的开头定义了两个整型变量 x、y 和两个可以指向整型变量的指针变量 ptr1、ptr2，这时这两种变量之间没有任何关系。语句 x=1 和 y=2 实现了对整型变量 x、y 的赋值。程序第 5、6 行“ptr1=&x;”和“ptr2=&y;”是将 x、y 的地址分别赋给 ptr1 和 ptr2。此时，最后一行的*ptr 和*ptr 就是变量 x 和 y，所以最后两个 printf 函数作用是相同的。

7.2.3　指针变量作为函数参数

指针类型是一个较特殊的数据类型，像别的一般的数据类型(如整型、实型、字符型等)的数据一样，指针类型的数据也可以作为函数的参数。由于指针变量被定义为指向其他变量的变量，所以它的作用是将一个变量的地址传送到另一个函数中。

[例 7.2] 输入的两个整数按大小顺序输出(用指针类型的数据作为函数参数)。

```
swap(int *ptr1,int *ptr2)
{
    int temp;
    temp=*p1;
    *ptr1=*ptr2;
    *ptr2=temp;
}
main()
{
    int a1,a2;
    int *p1,*p2;
    scanf("%d,%d",&a1,&a2);
    p1=&a1;p2=&a2;
    if(a1<a2)
    swap(p1,p2);
    printf("\n%d,%d\n",a1,a2);
 }
```

swap 函数是用户自定义的函数，它用来交换两个变量(a1 和 a2)的值。swap 函数的形参 ptr1、ptr2 是指针变量。程序运行时，先执行 main 函数，输入 a1 和 a2 的值。然后将 a1 和 a2 的地址分别赋给指针变量 p1 和 p2，使 p1 指向 a1，p2 指向 a2，如图 7.3 所示。

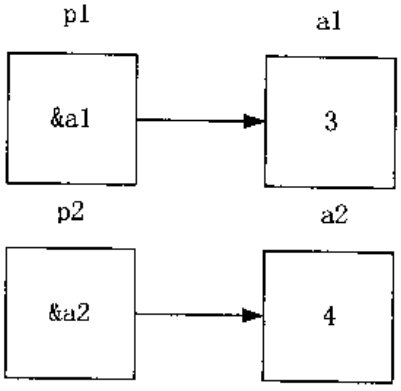


图 7.3　指针赋值

当执行 if 语句，由于 a1＜a2，因此执行 swap 函数。注意实参 p1 和 p2 是指针变量，在函数调用时，将实参变量的值传递给形参变量。采取的依然是“值传递”方式。因此虚实结合后形参 ptr1 的值为&a1，ptr2 的值为&a2。这时 ptr1 和 p1 指向变量 a1，ptr2 和 p2 指向变量 a2，如图 7.4 所示。

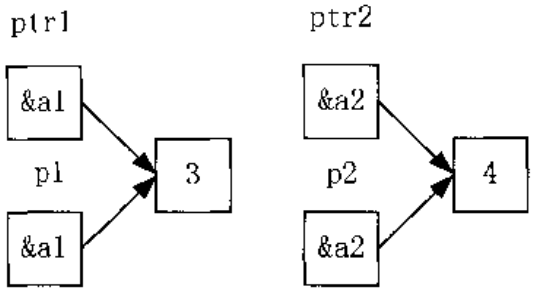


图 7.4 实参向形参的传递

接着执行 swap 函数的函数体使*ptr1 和*ptr2 的值互换，也就是使 a1 和 a2 的值 互换，如图 7.5 所示。

函数调用完成后，ptr1 和 ptr2 不复存在(已释放)，如图 7.6 所示。

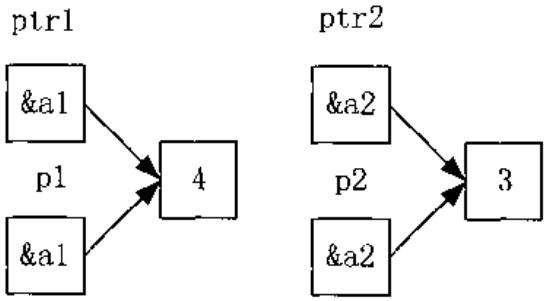


图 7.5 互换后变量的值

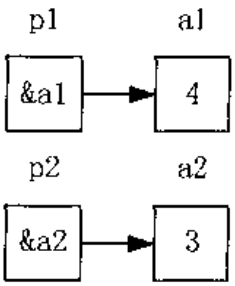


图 7.6 程序完成后各变量的值

最后在通过 printf 函数输出的 a1 和 a2 的值，此时这两个变量的值已交换。

7.3 数组与指针

一个变量有一个地址，一个数组包含若干元素，每个数组元素都在内存中占用存储单元，它们都有相应的地址。所谓数组的指针是指数组的起始地址，数组元素的指针是数组元素的地址。

7.3.1 指向数组元素的指针变量

一个数组是由连续的一块内存单元组成的。每个数组元素按其数据类型的不同占有不同数量的连续内存单元。一个数组元素的首地址就是指它所占有的几个内存单元的首地址。定义一个指向数组元素的指针变量的方法，与前面介绍的指针变量相同。

```
int d[5];
int *ptr;
```

应当注意的是指针变量的数据类型应与数组的数据类型相一致，对指向数组的指针赋值的格式如下式：

```
ptr=&d[0];
```

将数组 d 的首地址赋给指针变量 ptr，也就是指针 ptr 指向数组 d 的第 0 个元素，如图 7.7 所示。

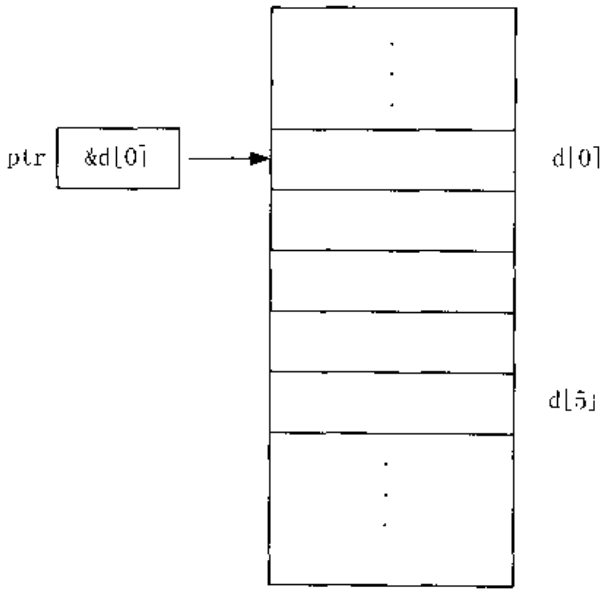


图 7.7　数组指针

由于 C 语言中规定数组名代表数组的首地址，所以下面两个语句等价：

```
ptr=&d[0];
ptr=d;
```

在定义指向数组的指针变量的过程中也可以对其赋初值，如下面两个语句：

```
int a[10];
int *ptr1=&a[0];
```

或者：

```
int a[10];
int *ptr1=a;
```

7.3.2　数组元素的引用(通过指针)

以下 C 语句：

```
int a[10];
int *ptr=&a[0];
*ptr=5;
```

表示对 ptr 当前所指向的整型数组元素赋值为 5。

如果数组元素是整型，每一个数组元素占 2 个字节，数组元素的地址加上 2 就是下一个数组元素的地址。C 语言规定，ptr+1 表示 ptr 的原值加上 2 个字节，这时指针变量 ptr 指向数组元素 a[1]。依此类推，ptr+n 将指向 a[n]数组元素，指针 ptr+n 指向的实际地址就

是 ptr+2n(当数组元素为整型时)。

ptr+n 和 a+n 指向 a[n]，也可以说它们表示 a[n]的地址，*(ptr+n)和*(a+n)是 ptr+n 和 a+n 指向的地址所存的数组元素，如图 7.8 所示。

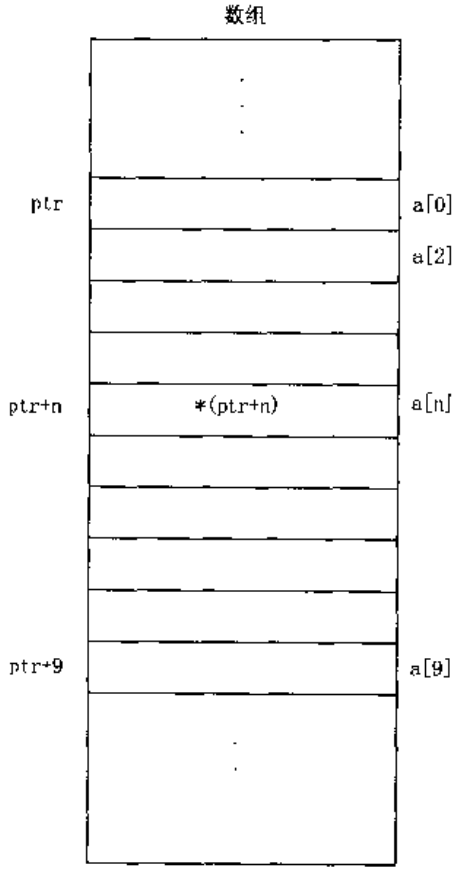


图 7.8 数组指针的表示

[例 7.3] 用指针变量指向数组元素的方法输出数组元素。

```
main()
{
    int number[10];
    int *ptr;
    ptr=number[0]
    for(int i=0;i<10;i++)
    {
        scanf("%d",ptr+i);
    }
    printf("\n");
    for(i=0;i<10;i++)
    {
        printf(%d',*ptr);
        ptr++;
    }
}
```

运行结果：

```
1 2 3 4 5 6 7 8 9 0 ↙
1 2 3 4 5 6 7 8 9 0
```

用指针变量直接指向数组元素的计算速度较快，不必每次重新计算地址。由于指针可以通过自加操作改变它的指向地址，而自加操作比较快，所以能大大提高程序的执行效率。使用指针变量的几点说明：

- 通过[例 7.3]可以看出，指针变量的值可以改变。ptr++语句可以使指针变量 ptr 的值不断改变，在 C 语言中这是合法的。而数组名 a 则是固定的指向数组的首地址，不能改变。
- p 的当前值。如[例 7.3]，当程序执行完成后，ptr 不再指向数组 number 的首地址，而是指向数组的末地址，编写程序时一定要注意这一点。
- 指针变量的运算。

ptr++、ptr 指向下一个元素，即指向 number［1］。

*ptr++、*和++处于同一优先级，因此此语句相当于*(ptr++)。它的作用是先得到指针 ptr 指向变量的值，然后再自加 1。

(*ptr)++表示将 ptr 所指向的变量的值加 1。

(ptr++)与(++ptr)的比较：前者先取*ptr 的值，然后再将 ptr 的值加 1。后者先将 ptr 的值加 1，然后再取*ptr 的值。

指针变量的--运算与++运算同理。

7.3.3　数组名作为函数参数

数组名可以作为函数的实参和形参。如：

```
main()
{
    int a1[10];
    ……
    f(a1,10);
    ……
}
……
f(int a[],int n);
{
    ……
}
```

a1 为实参数组名，a 为形参数组名。

当用数组元素作为函数参数时，数组元素的值并不改变，这种情况与用变量作为函数参数时一样，是“值传递”方式，单向传递数据。用数组名作为函数参数的情况就不一样了，由于数名表示数组的首地址，因此，用数组名作为函数实参，在调用函数时实际上是把数组的首地址传给形参，如图 7.9 所示。此时，实参数组和形参数组共占一个内存。可知，在函数调用的过程中，形参数组始终没有开辟新的内存单元，而是以实参数组的首地址作为形参数组的首地址。所以在函数的调用的过程中，当形参数组的元素值发生改变时，

实参数组的元素值也将随之改变。虽然实参数组的值和形参数组的值同时变化，但并不是将形参数组的值传递给实参数组，数据的传递仍然是单向的，仅仅是因为它们共占同一内存单元。

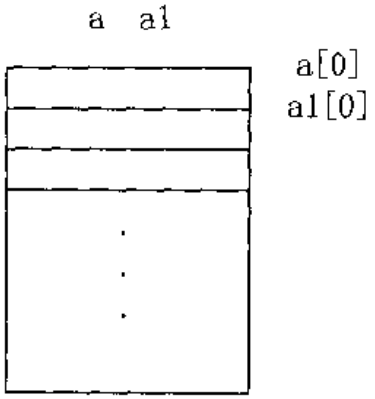


图 7.9 数组首地址传给形参

[例 7.4] 从 15 个数中找出其中的最大值和最小值。

程序如下：

```
int max,min;      /*定义全局变量*/
void value(int a[],int n)
{
    int *p;
    max=min=*a;
    for(int j=0;j<n;j++)
    {
        if(*p>max)
            max=*p;
        elseif(*p<min)
            min=*p;
            p++;
    }
    return;
}
main()
{
    int i,number[15];
    printf("enter 15 integer numbers:\n");
    for(i=0;i<10;i++)
    scanf("%d",&number[i]);
    value(number,10);
    printf("max=%d,min=%d\n",max,min);
 }
```

运行结果：

```
enter 15 integer numbers
1 -3 6 8 -17 9 10 8 32 4 78 150 -19 23 -5↙
max=150,min=-19
```

说明：

- 在函数 value 中求出的最大值和最小值放在 max 和 min 中。由于它们是全局变量，因此在主函数中可以直接使用。
- 函数 max_min_value 中的语句：max=min=*a;a 是数组名，它接收从实参传来的数组 number 的首地址。*a 相当于*(&a [0])。上述语句与“max=min=a[0];”等价。
- 在执行 for 循环时，p 的初值为 a+1，也就是使 p 指向 a[1]。以后每次执行 p++，使 p 指向下一个元素。每次将*p 和 max 与 min 比较，将大者放入 max，小者放入 min，如图 7.10 所示。
- 函数 max_min_value 的形参 array 可以改为指针变量类型。实参也可以不用数组名，而用指针变量传递地址。

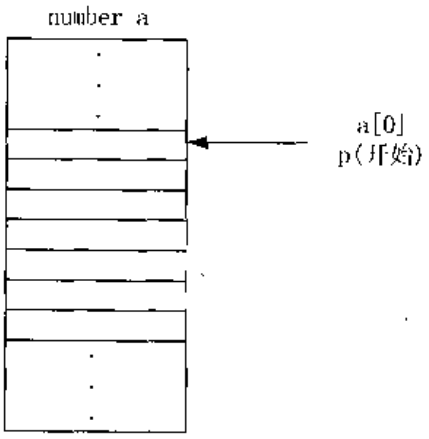


图 7.10　程序开始时指针的值

程序可改为：

```
int max,min;      /*定义全局变量*/
void value(int a[],int n)
{
    int *p;
    max=min=*a;
    for(int j=0;j<n;j++)
    {
        if(*p>max)
            max=*p;
        elseif(*p<min)
            min=*p;
            p++;
    }
    return;
}
main()
{
    int i,number[15],*ptr;
    ptr=number;
    printf("enter 15 integer numbers:\n");
```

```
    for(i=0;i<10;i++)
    scanf("%d",&number[i]);
    value(ptr,10);
    printf("max=%d,min=%d\n",max,min);
}
```

用数组作为函数的实参，如果想在函数中改变此数组的元素的值，实参与形参的对应关系有以下 4 种。

1. 形参和实参都是数组名

```
Main()
{
    int a1[10];
    ……
    f(a1,10);
}
……
f(int a[],int n)
{
    ……
}
```

程序中 a 和 a1 都已定义为数组，分别为函数的形参和实参。它们共用一段内存单元，也可以说它们指的是同一数组，如图 7.11 所示。

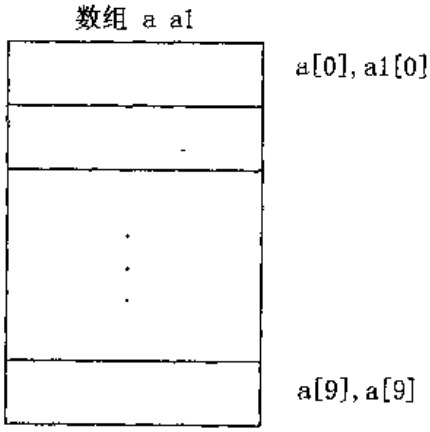


图 7.11 形参和实参的内存分配

2. 实参用数组名，形参用指针变量

```
main()
{
    int a1[10];
    ……
    f(a1,10)
}
……
f(int *a,int n)
{
```

```
    ……
}
```

实参 a1 为数组名，形参 a 为指向整型变量的指针变量。函数开始执行时，x 指向 a[0]，如图 7.12 所示。通过 a 的值的改变，可以使 x 指向数组的任一元素。

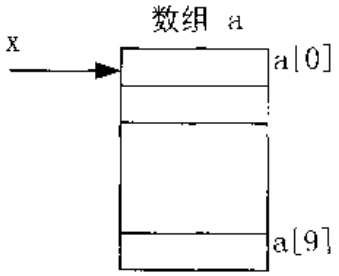


图 7.12　函数开始时的指针的值

3. 实参为指针变量，形参为数组名

```
main()
{
    int a1[10],*p;
    p=a1;
    ……
    f(p,10);
    ……
}
……
f(int a[],int n)
{
    ……
}
```

实参 p 是指向整型变量的指针变量，它的初始值为&a1[0]。形参 a 为数组名。首先使指针变量 p 指向 a[0]，即 p=&a[0]，然后使形参数组名 a 取得 a1 数组的首地址，如图 7.13 所示。

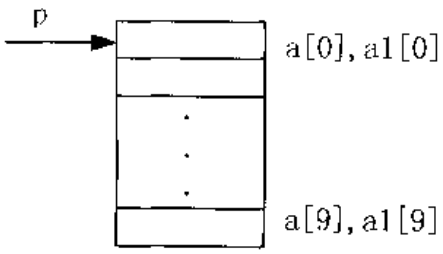


图 7.13　形参和实参之间的传递

4. 实参和形参都用指针变量

```
main()
{
    int a1[10];
```

```
    int *p;
    p=a1;
    ……
    f(p,10);
    ……
}
……
f(int *a,int n)
{
    ……
}
```

实参 p 和形参 a 都是指向整型变量的指针变量。首先将 p 赋值为&a1[0]，然后将 p 的值传给形参指针变量 a，a 的初始值也为&a[0]，如图 7.14 所示。通过改变 a 的值可以使 a 指向数组 a1 的任一个元素。

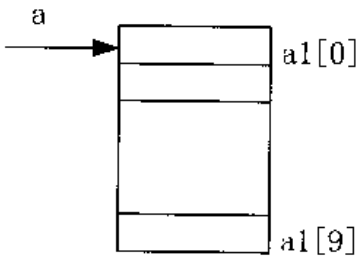


图 7.14 a 的初始值

7.3.4 指向多维数组的元素的指针变量

数组可以分为一维数组和多维数组。指针变量可以指向一维数组，也可以指向多维数组。前面主要介绍了指向一维数组的指针变量，接下来主要介绍指向多维数组的指针变量。为了方便地说明指向多维数组的指针变量，首先应了解多维数组的性质。以二维数组为例，设有一个 4 行 5 列的数组 a，定义为：

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

a 代表整个二维数组的首地址，即第 0 行的首地址。a+1 代表第 1 行的首地址。如果二维数组的首地址为 1000，则 a+1 为 1010，a+2 为 1020，a+3 为 1030。也可以把二维数组看作包含 4 个可以看作一维数组的元素：a[0]、a[1]、a[2]、a[3]，如图 7.15(a)。a[i]所代表的一维数组又包含 5 个元素：a[i][1]、a[i][2]、a[i][3]、a[i][4]、a[i][5]，如图 7.15(b)。

用地址法来表示数组各元素的地址。对元素 a [1] [2]，& a [1] [2]是其地址，a [1] + 2 也是其地址。分析 a [1] + 1 与 a [1] + 2 的地址关系，它们地址的差并非整数 1，而是一个数组元素的所占位置 2，原因是每个数组元素占两个字节。对 0 行首地址与 1 行首地址 a 与 a + 1 来说，地址的差同样也并非整数 1，是 1 行，5 个元素占的字节数为 8。

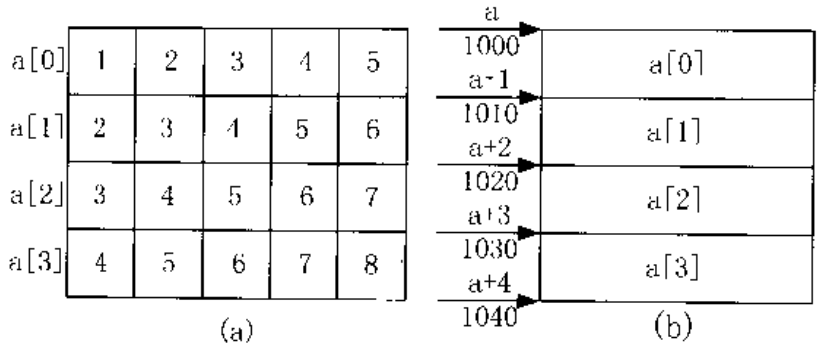


图 7.15　二维数组的地址

由于数组元素在内存的连续存放给指向整型变量的指针传递数组的首地址，则该指针指向二维数组。

```
int *p, a[4][5];
```

若赋值：ptr = a; 则用 p++就能访问数组的各元素。

[例 7.5] 用地址法输入输出二维数组的各个元素。

```
main()
{
    int a[4][5];
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<5;j++)
        {
            scanf("%d",a[i]+j); /*地址法*/
        }
    }
    for(i=0;i<4;i++)
    {
        for(j=0;j<5;j++)
        {
            printf("%4d",*(a[i]+j)); /* *(a[i]+j) 是数组元素*/
        }
        printf("\n");
    }
}
```

运行结果：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20↙
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

[例 7.6] 用指针法输入输出二维数组各元素。

```
main()
```

```
{
    int a[4][5],*p;
    int i,j;
    p=a[0];
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            scanf("%d",p++); /*指针的表示方法*/
        }
    }
    p=a[0];
    for(i=0;i<4;i++)
    {
        for(j=0;j<5;j++)
        {
            printf("%4d",*p++);
        }
        printf("\n");
    }
}
```

运行程序：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20↙
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

7.4 字符串与指针

字符和指针是两个完全不同的数据类型，但字符串占有一个连续的内存单元，而指针变量的值就是内存单元的地址，从而它们之间有了一定的联系。也可以说，有了指针这个数据类型，使对字符串的操作更容易、更方便。

7.4.1 字符串的表示形式

在C语言中，可以用两种方法表示一个字符串。一种是用字符数组实现，另一种是用字符指针实现。

- 字符数组：用数组存放一个字符串，然后输出该字符串。

  [例7.7]

```
main()
{
    char str[]="I am a good boy!";
```

```
    printf("%s\n",str);
}
```

上面程序中的字符数组和前面介绍的数组属性一样，str 是数组名，它代表字符数组的首地址，如图 7.16 所示。

- 字符指针：用字符串指针指向一个字符串。

[例 7.8]

```
main()
{
    char *str="I am a good boy!";
    printf("%s\n",str);
}
```

在这里没有定义字符数组，但 C 语言对字符串常量是按字符数组处理的，实际上在内存开辟了一个字符数组来存放字符串常量。在程序中定义了一个字符指针变量 str，并把字符串首地址赋给它，如图 7.17 所示。字符串指针变量的定义说明与指向字符变量的指针变量说明是相同的，只能按对指针变量的赋值不同来区别。对指向字符变量的指针变量应赋予该字符变量的地址。

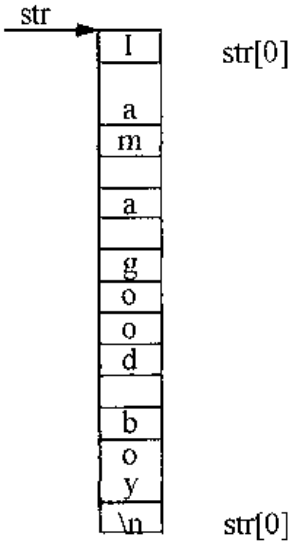


图 7.16　字符数组

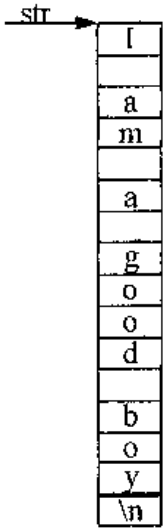


图 7.17　字符指针

如：

```
char c;
char *p=&c;
```

表示 p 是一个指向字符变量 c 的指针变量，而：

```
char *s="C Language";
```

则表示 s 是一个指向字符串的指针变量，把字符串的首地址赋予 s。

上例中，首先定义 str 是一个字符指针变量，然后把字符串的首地址赋予 str (应写出整个字符串，以便编译系统把该串装入连续的一块内存单元)，并把首地址送入 str。程序中的：

```
char *s="C Language";
```

等效于：

```
char *s;
s="C Language";
```

[例 7.9]输出字符串中 10 个字符后的所有字符。

```
main()
{
    char *ps="It is a interesting story";
    int n=10;
    ps=ps+n;
    printf("%s\n",ps);
}
```

运行结果为：

```
teresting story
```

在程序中对 ps 初始化时，即把字符串首地址赋予 ps，当 ps= ps+10 之后，ps 指向字符 t，因此输出为“teresting story”。

[例 7.10]在输入的字符串中查找有无 a 字符。

```
main()
{
    char a[10],*ps;
    int i;
    printf("input a string:\n");
    ps=a;
    scanf("%s",ps);
    for(i=0;i<=10;i++)
    {
        if(ps[i]=='a')
        {
            printf("there is a 'a' in the string\n");
            break;
        }
        if(ps[i]=='\0')
        {
            printf("There is no 'a' in the string\n");
        }
    }
}
```

运行结果：

```
b c e s a e g c k l↙
there is a 'a' in the string↙
```

7.4.2 字符串指针变量与字符数组的区别

用字符数组和字符指针变量都可实现字符串的存储和运算，但是两者是有区别的，在使用时应注意以下几个问题：

- 字符串指针变量本身是一个变量，用于存放字符串的首地址。而字符串本身是存放在以该首地址为首的一块连续的内存空间中并以“\0”作为串的结束。字符数组是由若干个数组元素组成的，它可用来存放整个字符串。
- 定义一个字符指针变量时。给指针变量分配内存单元，在其中放一个地址值，它可以指向一个字符型变量。但如果没有对它赋以一个地址值，则它并不能指向那一个字符型变量。而定义一个数组时，在编译程序时即分配内存单元，有确定的地址。
- 赋值方式。对数组赋值要用 static 存储类别，而对字符指针变量不必加 static。

```
static str[]={"I am a good boy"};
char *ps="I am a good boy";
```

对字符数组只能对各个元素赋值，不能用以下办法对字符数组赋值。

```
char str[5];
str="study";
```

而对字符指针变量，可以采用下面的方法赋值：

```
char *ps;
ps="study";
```

指针变量的值是可以改变的，如：

[例 7.11]

```
main()
{
    char *ps="I am good boy! ";
    ps=ps+10;
    printf("%s",ps);
}
```

运行结果：

```
boy!
```

7.5 函数与指针

可以用指针变量执行各种数据类型的变量，也可以使其指向一个函数。每一个函数都有一个入口地址，这个地址也就是函数的指针，所以也可以通过指针变量调用函数。

7.5.1 函数指针变量

一个函数占用一段连续的内存区，而函数名就是该函数所占内存区的首地址。我们可以定义一个指针变量，把函数的首地址赋予这个指针变量，即使该指针变量指向该函数。然后通过指针变量就可以找到并调用这个函数。我们把这种指向函数的指针变量称为“函数指针变量”。

函数指针变量定义的一般形式为：

```
类型说明符  (*指针变量名)();
```

其中“类型说明符”表示被指函数的返回值的类型。“(* 指针变量名)”表示“*”的后面是定义的指针变量名。最后的空括号表示指针变量所指的是一个函数。例如：

```
int (*ptrf)();
```

表示 ptrf 是一个指向函数的指针变量，该函数的返回值是整型。

［例 7.12］本例用来说明用指针形式实现对函数调用的方法。

```
int min(int a1,int a2)
{
    if(a1<a2)
        return a1;
        else
        return a2;
}
main()
{
    int min(int a1,int a2);
    int (*ptrmin)();
    int x,y,z;
    ptrmin=min;
    printf("input two numbers:\n");
    scanf("%d,%d",&x,&y);
    z=(*ptrmin)(x,y);
    printf("minmum=%d",z);
}
```

运行结果：

```
input two numbers:
1，2↙
maxmum=1
```

从上述程序可以看出用，函数指针变量形式调用函数的步骤如下：

(1) 先定义函数指针变量，如[例 7.13]中“int (*ptrmin)();”定义 ptrmin 为函数指针变量。

(2) 将函数的首地址赋予该函数指针变量，如程序中的“ptrmin=min;”语句。

(3) 用函数指针变量形式调用函数，如程序中的“z=(*ptrmin)(x,y); ”语句。

使用函数指针变量应注意以下两点：

- 函数指针变量与数组指针变量不同，不能进行算术运算。数组指针变量加减一个整数可使指针移动指向后面或前面的数组元素，而函数指针是不能进行此操作的，这种移动也是没有意义的。
- 函数调用中(*指针变量名)的两边的括号是不可少的，其中的“*”不应该理解为求值运算，在此处它只是一种表示符号。

7.5.2　指针型函数

所谓函数类型是指函数返回值的类型。一个函数可以返回各种类型的数据，如整型、字符型、实型等，也可以返回指针型的数据。在C语言中当一个函数的返回值是一个指针(即地址)的，称这种返回指针值的函数为指针型函数。

返回指针值的函数定义的一般形式为：

```
类型说明符 *函数名(形参表)
```

其中函数名之前加了“*”号表明这是一个指针型函数，即返回值是一个指针。类型说明符表示了返回的指针值所指向的数据类型。

如：

```
int *a(int a1,int a2)
{
    ……
}
```

表示 a 是一个返回指针值的指针型函数，它返回的指针指向一个整型变量。调用此函数后能得到一个指向整型数据的指针。

［例 7.13］输入一个 1~7 之间的整数，输出对应的星期名。

```
main()
{
    int i;
    char *search(int n);
    printf("input Day Number:\n");
    scanf("%d",&i);
    printf("The day is %s\n",day_name(i));
}
char *search(int n)
{
    static char *day_name[]={ "Illegal day",
                    "Monday",
                    "Tuesday",
                    "Wednesday",
                    "Thursday",
                    "Friday",
                    "Saturday",
                    "Sunday"};
```

```
    return((n<1||n>7) ? day_name[0] : day_name[n]);
}
```

运行结果：

```
1
The day is Monday
```

本例中定义了一个指针型函数 search，它的返回值指向一个字符串。该函数中定义了一个静态指针数组 day_name。day_name 数组初始化赋值为 8 个字符串，分别表示各个星期名及出错提示。形参 n 表示与星期名所对应的整数。在主函数中，把输入的整数 i 作为实参，在 printf 语句中调用 day_name 函数并把 i 值传送给形参 n。search 函数中的 return 语句包含一个条件表达式，n 值若大于 7 或小于 1，则把 name[0]指针返回主函数输出出错提示字符串“Illegal day”；否则返回主函数输出对应的星期名。

应该特别注意的是函数指针变量和返回指针值函数这两者的区别。如 int(*p)()和 int *p()是两个完全不同的量。int (*p)()是一个变量定义，定义 p 为一个指向函数的指针变量，该函数的返回值是整型量，(*p)的两边的括号不能少。int *p()不是变量的定义，而是函数的定义。定义 p 为一个指针型函数，其返回值是一个指向整型量的指针，*p 两边没有括号，作为函数定义。对于指针型函数定义，int *p()只是函数头部分，一般还应该有函数体部分。

7.6 指向指针的指针

一个指针变量可以指向整型变量、实型变量、字符类型变量，也可以指向指针类型变量。当这种指针变量用于指向指针类型变量时，我们称之为指向指针的指针变量。从图 7.18 可以看到，a 是一个指针数组，它的每一个元素是一个指针型数据。既然 a 是一个数组，它的每一个元素都有相应的地址。数组名 a 代表指针数组的首地址。a+i 是 a [i] 的地址，也是指向指针型数据的指针。

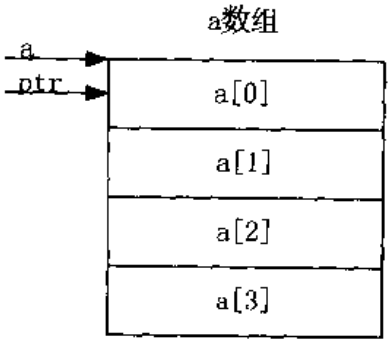


图 7.18 指向指针的指针

定义指向指针的指针变量的一般形式为：

类型标识符* *指针变量名;

例如：int * *ptr;

其含义为定义一个指针变量 ptr，它指向另一个指针变量(该指针变量又指向一个实型

变量)。由于指针运算符“*”是自右至左结合，所以上述定义相当于：

```
float *(*ptr);
```

[例 7.15] 用指向指针的指针变量访问一维和二维数组。

```
main()
{
    int a1[10],a2[3][4];
    int *ptr1,*ptr2,**ptr3,i,j;
    for(i=0;i<10;i++)
        scanf("%d",&a1[i]);
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            scanf("%d",&a2[i][j]);
    for(ptr1=a1,ptr3=&ptr1,i=0;i<10;i++)
        printf("%4d",*(*p3+i));
    printf("\n");
    for(ptr1=a1;ptr1-a1<10;ptr1++)
    {
        ptr3=&ptr1;
        printf("%4d",**ptr3);
    }
    printf("\n");
    for(i=0;i<3;i++)
    {
        ptr2=a2[i];
        ptr3=&ptr2;
        for(j=0;j<4;j++)
        printf("%4d",*(*ptr3+j));
        printf("\n");
    }
    for(i=0;i<3;i++)
    {
        ptr2=a2[i];
        for(ptr2=b[i];ptr2-a2[i]<4;ptr2++)
        {
            ptr3=&ptr2;
            printf("%4d",**ptr3);
        }
        printf("\n");
    }
}
```

运行程序：

```
1 2 3 4 5 6 7 8 9 0↙
1 3 5 7↙
2 4 6 8↙
5 7 9 2↙
1 2 3 4 5 6 7 8 9 0
1 2 3 4 5 6 7 8 9 0
```

```
1 3 5 7
2 4 6 8
5 7 9 2
1 3 5 7
2 4 6 8
5 7 9 2
```

对一维数组 a1 来说，若把数组的首地址即数组名赋给指针变量 ptr1，ptr1 就指向数组 a1，数组的各元素用 ptr1 表示为*(ptr1+i)，也可以简化为*p1+i 表示。如果将 ptr3=&p1，则将 p1 的地址传递给指针变量 ptr3，*ptr3 就是 ptr1。用 ptr3 来表示一维数组的各个元素，只需要将用 ptr1 表示的数组元素*(p1+i)中的 ptr1 换成*ptr3 即可，表示为*(*p3+i)。

同样，对二维数组 a2 来说，a2[i]表示第 i 行首地址，将其传递给指针变量 ptr2，使其指向该行。该行的元素用 ptr2 表示为*(ptr2+i)。若作 ptr3=&ptr2，则表示 ptr3 指向 ptr2，用 ptr3 表示的二维数组第 i 行元素为：*(*p3+i)，这与程序中的表示完全相同。

7.7 有关指针数据类型和运算小结

为了使读者对指针数据类型有一个系统完整的概念，本节将对有关指针数据类型和关于指针运算做一个小结。

7.7.1 有关指针的数据类型的小结

为了便于学习记忆，使读者有一个系统的概念，将关于指针的定义列在一起，如表 7.1 所示。

表 7.1 指针的定义

定 义	含 义
int i;	定义整型变量 i
int *ptr	ptr 为指向整型数据的指针变量
int a[n];	定义整型数组 a，它有 n 个元素
int *ptr[n];	定义指针数组 ptr，它由 n 个指向整型数据的指针元素组成
int (*ptr)[n];	ptr 为指向含 n 个元素的一维数组的指针变量
int f();	f 为带回整型函数值的函数
int *ptr();	ptr 为带回一个指针的函数，该指针指向整型数据
int (*ptr)();	ptr 为指向函数的指针，该函数返回一个整型值
int **ptr;	ptr 是一个指针变量，它指向一个指向整型数据的指针变量

7.7.2 指针运算的小结

现把全部指针运算列出如下：

1. 指针变量加(减)一个整数

例如：p++、p--、p+i、p-i、p+=i、p-=i

一个指针变量加(减)一个整数并不是简单地将原值加(减)一个整数，而是将该指针变量的原值(是一个地址)和它指向的变量所占用的内存单元字节数加(减)。

2. 指针变量赋值

将一个变量的地址赋给一个指针变量。

```
ptr=&a;          (将变量 a 的地址赋给 ptr)
ptr=a;        (将数组 a 的首地址赋给 ptr)
ptr=&a[i];    (将数组 a 第 i 个元素的地址赋给 ptr)
ptr=max;         (max 为已定义的函数，将 max 的入口地址赋给 ptr)
ptr1=ptr2;          (ptr1 和 ptr2 都是指针变量，将 ptr2 的值赋给 ptr1)
```

注意：不能如下：

```
ptr=1000;
```

3. 指针变量可以有空值

该指针变量不指向任何变量

```
ptr=NULL;
```

4. 两个指针变量可以相减

如果两个指针变量指向同一个数组的元素，则两个指针变量值之差是两个指针之间的元素个数。

5. 两个指针变量比较

如果两个指针变量指向同一个数组的元素，则两个指针变量可以进行比较。指向前面的元素的指针变量“小于”指向后面的元素的指针变量。

在本章中介绍了指针的基本概念和初步应用。指针是 C 语言中重要的概念，也是 C 语言的一个特色。但同时指针使用起来太灵活，也容易出错。有时由于指针运用的错误甚至会引起时整个程序遭受破坏。因此，使用指针要十分小心谨慎，要多积累经验。

第 8 章　结构体和共用体

在前面章节中我们已经学习了基本的数据类型(如整型、实型、字符型)的定义和引用，还学习了数组(一维、二维)的定义和应用。我们已经学习的数据类型有一个共同的特点是：当定义某一特定数据类型，就限定该类型变量的存储特性和取值范围。对简单数据类型来说，既可以定义单个的变量，也可以定义数组。而当定义维数组时，数组的全部元素都具有相同的数据类型。

但这些数据类型还不能满足要求，在日常生活中，我们经常会遇到一些需要记录综合信息的表格，如学籍表、成绩单、通信地址等。在这些表中，填写的各种数据是不能用同一种数据类型表达的，在成绩单中我们需要填写姓名、学号、性别、出生年月、分数等项目；在通信地址表中我们要写下姓名、邮编、地址、电话号码、E-mail 等项目。一个表中包括了各种数据类型，我们无法用前面学过的任一种数据类型来描述。如果将一个表中的各个分项目分别定义为不同的简单数据类型，程序将难以表达它们之间的内在联系，因此 C 语言引入一种能集中不同数据类型于一体的数据结构，并称之为结构体。结构体类型的变量可以拥有不同数据类型的成员，是不同数据类型成员的集合。

8.1　结构体的定义和引用

在实际问题中，一组数据往往具有不同的数据类型。例如，在学生登记表中，姓名应为字符型；学号可为整型或字符型；年龄应为整型；性别应为字符型；成绩可为整型或实型。显然不能用一个数组来存放这一组数据。因为数组中各元素的类型和长度都必须一致，以便于编译系统处理。为了解决这个问题，C 语言中给出了另一种构造数据类型——“结构(Structure)”或叫“结构体”。它相当于其他高级语言中的记录。“结构”是一种构造类型，它是由若干“成员”组成的。每一个成员可以是一个基本数据类型或者又是一个构造类型。结构既是一种“构造”而成的数据类型，那么在说明和使用之前必须先定义它，也就是构造它，如同在说明和调用函数之前要先定义函数一样。

8.1.1　结构体类型变量的定义

定义一个结构的一般形式为：

```
struct 结构名
    {成员表列};
```

{成员表列}由若干个成员组成，每个成员都是该结构的一个组成部分。对每个成员也必须进行类型说明，其形式为：

类型说明符 成员名;

成员名的命名应符合标识符的书写规定。例如：

```
struct stu
{
    int num;
    char name[20];
    char sex;
    float score;
};
```

在这个结构定义中，结构名为 stu，该结构由 4 个成员组成。第 1 个成员为 num，整型变量；第 2 个成员为 name，字符数组；第 3 个成员为 sex，字符变量；第 4 个成员为 score，实型变量。应注意在括号后的分号是不可少的。结构定义之后，即可进行变量说明。凡说明为结构 stu 的变量都由上述 4 个成员组成。由此可见，结构是一种复杂的数据类型，是数目固定，类型不同的若干有序变量的集合。

结构体类型变量的定义与其他类型的变量的定义是一样的，但由于结构体类型需要针对问题事先自行定义，所以结构体类型变量的定义形式就增加了灵活性，共计有 3 种形式，分别介绍如下：

1. 先定义结构体类型，再定义结构体类型变量

```
struct stu / *定义学生结构体类型* /
{
    char name[20]; / * 学生姓名* /
    char sex; / * 性别* /
    long num; / *学号* /
    float score[3]; / * 三科考试成绩* /
};
struct stu student1,student2;/ * 定义结构体类型变量* /
```

也可以直接定义结构体变量，省去关键词“struct”，例如：

```
stu student1,student2;
```

用此结构体类型，可以定义更多的该结构体类型变量。如果程序规模比较大，往往将对结构体类型的定义集中放到一个文件(以.h 为后缀的“头文件”)中。哪个源文件需用到此结构类型则可用#include 命令将该头文件包含到本文件中。这样做便于装配、便于修改、便于使用。

2. 定义结构体类型同时定义结构体类型变量

```
struct data
{
    int day;
    int month;
    int year;
} data1,data2;
```

也可以再用此结构体定义如下变量：

```
struct data data3,data4;
```

用此结构体类型，同样可以定义更多的该结构体类型变量。这种形式的定义一般形式为：

```
struct 结构体名
    {
    成员表列
}变量名表列;
```

3. 直接定义结构体类型变量

```
struct
{
    char name[20]; / *学生姓名* /
    char sex; / *性别* /
    long num; / *学号* /
    float score[3]; / *三科考试成绩* /
} student1,student2; / *定义该结构体类型变量* /
```

该定义方法没有结构体的名字，所以无法记录该结构体类型。除直接定义外，不能再定义该结构体类型变量。

8.1.2 结构体类型变量的引用

学习了怎样定义结构体类型和结构体类型变量，怎样正确地引用该结构体类型变量的成员呢？结构体变量的引用形式为：

```
<结构体类型变量名> . <成员名>
```

若我们定义的结构体类型及变量如下：

```
struct stu
{
   int num;
   char name[20];
   char sex;
   float score;
}student1,student2;
```

则变量 student1 和 student2 各成员的引用形式为：student1.num、student1.name、student1.sex、student1.score 及 student2.num、student2.name、student2.sex、student2.score。其结构体类型变量的各成员与相应的简单类型变量使用方法完全相同。

结构体变量的引用方式上面已经做了介绍，但在结构体变量被引用的时候，还应遵守一些规则：

- 不能将一个结构体变量作为一个整体进行输入和输出。例如，已定义 student1 和 student2 为结构体变量并且它们已有值，不能这样引用：

  ```
  scanf("%d,%s,%c,%d,%f,%s\n",&student1);
  ```

  只能对结构体变量中的各个成员分别输出，引用方式为：

结构体变量名.成员名

例如：student1.score 表示 student1 变量中的 score 成员，即 student1 的 score 项。可以对变量的成员赋值，例如：student1.score=80；“.”是成员运算符，它在所有的运算符中优先级最高，因此可以把 student1.num 作为一个整体来看待。

- 如果成员本身又属一个结构体类型，则要用若干个成员运算符，一级一级地找到最低一级的成员。只能对最低的成员进行赋值或存储以及运算。例如，对上面定义的结构体变量 student1，可以这样访问各成员：

```
student1.num
student1.name
student1.birthday.month
student1.birthday.day
student1.birthday.year
```

注意： 不能用 student1.birthday 来访问 student1 变量中的成员 birthday，因为 birthday 本身是一个结构体变量。

- 对成员变量可以像普通变量一样进行各种运算(根据其类型决定可以进行的运算)。例如：

```
student2.score=student1.score;
sum=student1.score+student2.score;
student1.age++;
++student1.age;
```

由于“.”运算符的优先级最高，因此 student1.age++是对 student1.age 进行自加运算，而不是先对 age 进行自加运算。

- 可以引用成员的地址，也可以引用结构体变量的地址。如：

```
scanf("%d",&student1.num);
printf("%o",&student1);
```

结构体变量的地址主要用于作为函数参数，传递结构体的地址。

8.2　结构类型的说明

关于结构体类型的几点要说明：

- 结构体类型与结构体变量是两个完全不同的概念，不要混同。对结构体变量来说，在定义时一般先定义一个结构体类型，然后定义该类型的变量。只能对变量的各个成员赋值、存取或运算，而不能对一个结构体类型赋值、存取或运算。在编译时，对结构体类型是不分配空间的，只对变量分配空间。
- 对结构体中的成员，可以单独使用，它的作用与用法相当于一般变量。关于对成员的引用方法上一节已经介绍。
- 结构体变量的成员也可以是一个结构体变量。

例如：

```
struct birthday
{
    int year;
    int month;
    int day;
};
struct student
{
    char num[10];
    char sex;
    int age;
    struct birthday birthday1;
    char addr[30];
}student1,student2;
```

先定义一个 struct birthday 类型，它代表“出生年月”，包括3个成员：year(年)、month(月)、day(日)。然后在定义 struct student 类型时，成员 birthday1 的类型定义为 struct birthday 类型。struct student 的结构如图 8.1 所示。已定义的类型 struct birthday 与其他类型一样可以用来定义成员的类型。

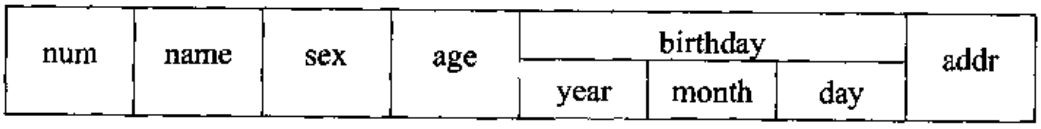


图 8.1 struct student 的结构

成员可以与程序中的变量名相同，二者不代表同一对象。

8.3 结构体变量的初始化和赋值

结构体变量定义完成后，要对其进行赋值。同时，也可以在定义时对其赋值，即初始化。本节将主要针对结构体变量的初始化和赋值进行介绍。

8.3.1 结构体变量的初始化

由于结构体类型变量汇集了各类不同数据类型的成员，所以结构体类型变量的初始化就略显复杂。

结构体类型变量的定义和初始化为：

```
struct stu / *定义学生结构体类型* /
{
    char name[10]; / * 学生姓名* /
    char sex; / * 性别* /
    long num; / *学号* /
    float score[4]; / * 四门考试成绩* /
};
struct stu student1={"xiaoming",'f',9801046,90,97,85.5,80};
```

上述对结构体类型变量的 3 种定义形式均可在定义时初始化。结构体类型变量完成初始化后，即各成员的值分别为：

```
student.name="xiaoming"
student.sex='f'
student.num=9801046
student.score[0]=90
student.score[1]=97
student.score[2]=85.5
student.score[3]=80。
```

[例 8.1]

```
main()
{
    static truct stu
    {   long num;
        char name[10];
        char sex;
        int age;
    }student1={9801046,"dengfang","f",20};
    printf("No.:%ld\nname:%s\nsex:%c\n\nage:%d",student1.num,
            student1.name,student1.sex,student1.age)
}
```

8.3.2　结构体变量的赋值

结构变量的赋值就是给各成员赋值，可用输入语句或赋值语句来完成。

[例 8.2] 给结构变量赋值并输出其值。

```
main()
{
    struct stu
    {
        long num;
        char *name;
        char sex;
        float score;
    } student1,student2;
    student1.num=980101;
    student1.name="Zhang ming";
    printf("input sex and score\n");
    scanf("%c %f",&student1.sex,&student1.score);
    strdent2=student1;
    printf("No.=%d\nName=%s\n",student2.num,student2.name);
    printf("Sex=%c\nScore=%f\n",student2.sex,student2.score);
}
```

运行结果：

```
input sex and score↙
```

```
f 90↙
No.-980101↙
Name-Zhang ming↙
Sex=f↙
Score=90↙
```

本程序中用赋值语句给 num 和 name 两个成员赋值，name 是一个字符串指针变量。用 scanf 函数动态地输入 sex 和 score 成员值，然后把 student1 的所有成员的值整体赋予 student2，最后分别输出 student2 的各个成员值。本例表示了结构变量的赋值、输入和输出的方法。

8.4 结构体数组

数组的元素也可以是结构类型的，因此可以构成结构体数组。结构体数组与一般类型的数组大体上一样，仅仅是结构体数组的元素都是一个结构体类型的数据，他们都分别包括各个成员项。在实际应用中，经常用结构数组来表示具有相同数据结构的一个群体。如一个班的学生的成绩，一个车间职工的档案等。

8.4.1 结构体数组的定义

和定义结构体变量的方法相似，只需说明其为数组即可。例如：

```
struct stu
{
    int num;
    char name[20];
    char sex;
    int age;
    float score;
  };
struct stu student[3];
```

上面定义了一个结构体数组 stu，其元素为 struct student 类型的数据。也可以直接定义一个结构体数组，如：

```
struct stu
{
    int num;
    ……
}stu[3];
```

或

```
struct()
{
    int num;
```

```
    ……
}stu[3];
```

8.4.2　结构体数组的初始化

我们可以对结构体初始化赋值，如下例：

```
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}student[4]={
                {101,"Wang ping","F",50},
                {102,"Zhang ping","M",80.5},
                {103,"Deng fang","F",90.5},
                {104,"Cheng ling","F",89},
            }
```

如图 8.2 所示。

	num	name	sex	score
stu[0]	101	W ang p ing	F	50
stu[1]	102	Zhang p ing	M	80. 5
stu[2]	103	D eng fang	F	90. 5
stu[3]	104	C heng ling	F	89

图 8.2　结构体数组赋值

当对全部元素初始化赋值时，也可不给出数组长度。

［例 8.3］求出 4 个学生的平均成绩和不及格的人数。

```
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}student[4]={
                {101,"Wang ping","F",50},
                {102,"Zhang ping","M",80.5},
                {103,"Deng fang","F",90.5},
                {104,"Cheng ling","F",89},
            };
```

```
main()
{
        int i,c=0;
        float average,sum=0;
        for(i=0;i<5;i++)
        {
        s+=student[i].score;
        if(student[i].score<60) c+=1;
        }
        average=sum/5;
        printf("average=%2f\ncount=%d\n",ave,c);
}
```

运行结果:

```
average=78.50↙
count=1↙
```

本例程序中定义了一个外部结构数组 student，共 4 个元素，并初始化赋值。在 main 函数中用 for 语句逐个累加各元素的 score 成员值存于 sum 之中，如 score 的值小于 60(不及格)即计数器 c 加 1，循环完毕后计算平均成绩，并输出全班平均分和不及格人数。

[例 8.4]建立同学通信录。

```
#include"stdio.h"
struct addr_list
{
        char name[20];
        char phone[10];
};
main()
{
        struct addr_list addr[3];
        int i;
        for(i=0;i<3;i++)
    {
        printf("input name:\n");
        gets(addr[i].name);
        printf("input phone:\n");
        gets(addr[i].phone);
    }
    printf("name\t\t\tphone\n\n");
    for(i=0;i<3;i++)
        printf("%s\t\t\t%s\n",addr[i].name,addr[i].phone);
}
```

本程序中定义了一个结构 addr_list，它有两个成员 name 和 hone 用来表示姓名和电话号码。在主函数中定义 addr 为具有 addr_list 类型的结构数组。在 for 语句中，用 gets 函数分别输入各个元素中两个成员的值。然后又在 for 语句中用 rintf 语句输出各元素中两个成员值。

8.5　指向结构体类型变量的指针

所谓结构体变量的指针就是该结构体变量的首地址。可以定义一个指针变量，使其指向一个结构体变量，此时该变量的值就是所指向的结构体变量的起始地址。指针变量也可以指向结构体数组中的元素。

8.5.1　指向结构体变量的指针

结构指针变量说明的一般形式为：

```
struct 结构名 *结构指针变量名
```

例如，在前面的例题中定义了 stu 这个结构，如要说明一个指向 stu 的指针变量 stu，可写为：

```
struct stu *pstu;
```

当然也可在定义 stu 结构时同时说明 stu。与前面讨论的各类指针变量相同，结构指针变量也必须要先赋值后才能使用。

赋值是把结构变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。如果 student 是被说明为 stu 类型的结构变量，则：

```
pstu=&student
```

是正确的，而：

```
pstu=&stu
```

是错误的。

其访问的一般形式为：

```
(*结构指针变量).成员名
```

或为：

```
结构指针变量->成员名
```

例如：

```
(*pstu).num
```

或者：

```
pstu->num
```

应该注意(*stu) 两侧的括号不可少，因为成员符“.”的优先级高于“*”。如去掉括号写作 *stu.num 则等效于*(stu.num)，这样，意义就完全不对了。

下面通过例子来说明结构指针变量的具体说明和使用方法。

[例 8.5]

```
struct stu
{
    int num;
    char name[20];
    char sex;
    float score;
    }student1,*pstu;
main()
{
    pstu=&student1;
    student1.num=1;
    student1.name="Zhao Ming";
    student1.sex="F";
    student1.score=80.5;
    printf("Number=%d\nName=%s\n",student1.num,student1.name);
    printf("Sex=%c\nScore=%f\n\n",student1.sex,student1.score);
    printf("Number=%d\nName=%s\n",(*pstu).num,(*pstu).name);
    printf("Sex=%c\nScore=%f\n\n",(*pstu).sex,(*pstu).score);
    printf("Number=%d\nName=%s\n",pstu->num,pstu->name);
    printf("Sex=%c\nScore=%f\n\n",pstu->sex,pstu->score);
}
```

运行结果：

```
Number=1↙
Name=Zhao Ming↙
Sex=F
Score=80.5
Number=1↙
Name=Zhao Ming↙
Sex=F
Score=80.5
Number=1↙
Name=Zhao Ming↙
Sex=F
Score=80.5
```

本例程序定义了一个结构 stu，定义了 stu 类型结构变量 student1，并在主函数中初始化赋值，还定义了一个指向 stu 类型结构的指针变量 pstu。在 main 函数中，pstu 被赋予 student1 的首地址，因此 pstu 指向 student1。然后在 printf 语句内用 3 种形式输出 student1 的各个成员值，从运行结果可以看出：

```
结构变量.成员名
(*结构指针变量).成员名
结构指针变量->成员名
```

这 3 种用于表示结构成员的形式是完全等效的。

结构体名称和结构体变量是两个完全不同的概念，不能混淆。结构体名称只能表示一个结构形式，编译系统并不对它分配内存空间。只有当某变量被说明为这种类型的结构时，

才对该变量分配存储空间。因此上面&stu 这种写法是错误的，不可能去取一个结构名的首地址。有了结构指针变量，就能更方便地访问结构变量的各个成员。

8.5.2　指向结构体数组的指针

结构体指针变量可以指向一个结构数组，这时结构体指针变量的值是整个结构体数组的首地址。结构体指针变量也可指向结构体数组的一个元素，这时结构体指针变量的值是该结构体数组元素的首地址。

设 p 为指向结构体数组的指针变量，则 p 也指向该结构数组的 0 号元素，ps+1 指向 1 号元素，ps+i 则指向 i 号元素。这与普通数组的情况是一致的。

[例 8.6]用指针变量输出结构数组。

```
struct stu
{
    long num;
    char *name;
    char sex;
    float score;
}student[5]={
            {9801001,"Zhao ming",'F',59},
            {9801002,"Zhou ping",'M',83.5},
            {9801003,"Liu li",'M',72},
            {9801004,"Deng ling",'F',87},
            {9801005,"Wang gang",'M',48},
    };
main()
{
    struct stu *p;
    int i;
    for(i=0;i<5;i++)
    {
        p=student;
        printf("%ld,%s,%c,%f\n",
                p->num,p->name,p->sex,p->score);
        p++;
    }
}
```

运行结果：

```
9801001,"Zhao ming",'F',59↙
9801002,"Zhou ping",'M',83.5↙
9801003,"Liu li",'M',72
9801004,"Deng ling",'F',87
9801005,"Wang gang",'M',48
```

在程序中，定义了 stu 结构类型的外部数组 student 并初始化赋值。在 main 函数内定义 p 为指向 stu 类型的指针。在循环语句 for 的循环体中，p 被赋予 student 的首地址，然后循环 5 次，输出 student 数组中各成员值。

应该注意的是，一个结构指针变量虽然可以用来访问结构变量或结构数组元素的成员，但是，不能使它指向一个成员，也就是说不允许取一个成员的地址来赋予它。因此，下面的赋值是错误的。

```
p=&student[1].sex;
```

而只能是：

```
p=student;
```

或者是：

```
p=&student[0];
```

8.5.3 指向结构体变量的指针做函数参数

有时需要将一个结构体变量的值传递给另一个函数，有两种方法可以传递：一是用结构体变量的成员作参数；二是用指向结构体变量的指针作参数，将结构体变量的地址传给行参。允许用结构变量作为函数参数进行整体传送。但是这种传送要将全部成员逐个传送，特别是成员为数组时将会使传送的时间和空间开销很大，严重地降低了程序的效率。因此最好的办法就是使用指针，即用指针变量作为函数参数进行传送。这时由实参传向形参的只是地址，从而减少了时间和内存的开销。

[例 8.7]计算一组学生的平均成绩和不及格人数。

```
struct stu
{
    long num;
    char *name;
    char sex;
    float score;
}student[5]={
                {9801001,"Zhao ming",'F',59},
                {9801002,"Zhou ping",'M',83.5},
                {9801003,"Liu li",'M',72},
                {9801004,"Deng ling",'F',87},
                {9801005,"Wang gang",'M',48.5},
            };
void ave(struct stu *p)
{
    int c=0,i;
    float ave,sum=0;
    for(i=0;i<5;i++,p++)
    {
        sum+=p->score;
        if(p->score<60) c+=1;
    }
    ave=sum/5;
    printf("average=%2f\ncount=%d\n",ave,c);
}
```

```
main()
{
    struct stu *p;
    void ave(struct stu *p);
    p=student;
    ave(p);
}
```

运行结果：

```
average=70.00↙
count=2↙
```

本程序中定义了函数 ave，其形参为结构指针变量 p。student 被定义为外部结构数组，因此在整个源程序中有效。在 main 函数中定义说明了结构指针变量 p，并把 student 的首地址赋予它，使 p 指向 student 数组。然后以 p 作为实参调用函数 ave。在函数 ave 中完成计算平均成绩和统计不及格人数的工作并输出结果。

8.6　共　用　体

所谓共用体类型是指将不同的数据项组织成一个整体，它们在内存中占用同一段存储单元。例如，可把一个整型变量、一个字符型变量、一个实型变量放在同一个地址的内存单元中。以上 3 个变量在内存中所占的字节数不同，但都从同一地址开始存放。

8.6.1　共用体的定义

其定义形式为：

```
union 共用体名
{
    成员表列
}; 变量列表;
```

例如：

```
union time
{
    int a;
    float b;
    double c;
    char d;
} time1,time2;
```

也可以将类型与变量定义分开：

```
union time
{
    int a;
    float b;
```

```
    double c;
    char d;
};
union time time1,time2;
```

上面定义了一个共用体数据类型 union time，定义了共用体数据类型变量 time1、time2。共用体数据类型与结构体在形式上非常相似，但其表示的含义及存储方式是完全不同的。先让我们看一个小例子。

[例 8.8]

```
union time
{
    int a;
    float b;
    double c;
    char d;
}time1;
struct stu
{
    int a;
    float b;
    double c;
    char d;
};
main()
{
    struct stu student;
    printf("%d,%d",sizeof(struct stu),sizeof(union time));
}
```

运行结果：

```
15, 8
```

程序的输出说明结构体类型所占的内存空间为其各成员所占存储空间之和，而形同结构体的共用体类型实际占用存储空间为其最长的成员所占的存储空间。

8.6.2 共用体变量的引用

可以引用共用体变量的成员，对共用体的成员的引用与结构体成员的引用相同。但由于共用体各成员共用同一段内存空间，使用时，根据需要使用其中的某一个成员。若定义共用体类型为：

```
union time
{
    int a;
    float b;
    double c;
    char d;
}time1;
```

其成员引用为：time1.a,time1.b,time1.c,time1.d。但是要注意的是，不能同时引用 4 个成员，在某一时刻，只能使用其中之一的成员。

[例 8.9]

```
main()
{
    union t
    {
        int a;
        float b;
        double c;
        char d;
    }t1;
    t1.a =6;
    printf("%d\n",t1.a);
    t1.c=67.2;
    printf("%5.1f\n",t1.c);
    t1.d='W';
    t1.b=34.2;
    printf("%5.1f,%c\n",t1.b,t1.d);
}
```

运行程序输出为：

```
6↙
6 7. 2↙
3 4. 2, =↙
```

程序最后一行的输出是我们无法预料的，其原因是连续做“t1.d='　';”、“t1.b=34.2;”两个连续的赋值语句最终使共用体变量的成员 t1.b 所占 4 字节被写入 34 .2，而写入的字符被覆盖了，输出的字符变成了符号“=”。事实上，字符的输出是无法得知的，由写入内存的数据决定。

[例 8.10]

```
struct time
{
    int year;
    int month;
    int day;
};
union a
{
    struct time date;
    char b[10];
};
```

假定共用体的成员在内存的存储是从地址 2000 单元开始存放，整个共用体类型需占存储空间 6 个字节，即共用体 a 的成员 date 与 b 共用这 6 个字节的存储空间，由于共用体成员 data 包含 3 个整型的结构体成员，各占 2 个字节。date.year 是由 2 个字节组成，用 b 字符数组表示为 b[0]和 b[1]。byte[1]是高字节，byte[0]是低字节。下面用程序实现共用体在内

存中的存储。

```
struct time
{
    int year;
    int month;
    int day;
};
union a
{
    struct time date;
    char b[10];
};
main()
{
    union a u;
    int i;
    printf("enter year:\n");
    scanf("%d",&u.date.year);
    printf("enter month:\n");
    scanf("%d",&u.date.month);
    printf("enter day:\n");
    scanf("%d",&u.date.day);
    printf("year=%d month=%d day=%d\n",unit.data.year,
            unit.data.month, unit.date.day);
    for(i=0;i<6;i++)
        printf("%d,",unit.b[i]);
}
```

运行程序：

```
enter year: ↙
1990↙
enter month: ↙
9↙
enter day: ↙
10↙
year=1990 month=9 day=10
198,7,4,0,10,0
```

从程序的输出结果来看，1990占两个字节，由第0、1字节构成，即7×256+198=1990。9同样占两个字节，由第2、3字节构成，0×256+4=4。10由第4、5字节构成，10=0×256+10。

本章主要介绍了结构体数据类型变量的定义和应用，它可以将一组不同类型的数据组合成一个有机的、互相联系的整体。结构体类型满足了一些需要进行复杂记录的要求，比如学籍卡、住宿登记卡等。

第 9 章　AT90LS8535 的内部资源

AT90LS8535 单片机的内部具有数字 I/O 口、中断、同步/异步串行口、定时/计数器、EEPROM、模数转换器和模拟比较器等多种内部资源。对这些内部资源的编程是开发 AT90LS8535 单片机应用系统的基础，因此，本章将详细介绍如何运用 C 语言对这些内部资源进行编程。

9.1　I/O 口

AVR 单片机的构成，除了 CPU 和存储器以外，还包括输入输出端口。所有的 AVR 单片机的 I/O 端口都具有读、写和修改功能。

9.1.1　端口 A

1. 端口特性

端口 A 是一个 8 位双向 I/O 口。端口 A 包含 3 个 I/O 地址，其中有两个寄存器和一个输入引脚：数据寄存器 PORTA($1B)、数据方向寄存器 DDRA($1A)和输入引脚 PINA($19)，分别如表 9.1、9.2、9.3 所示。数据寄存器 PORTA 和数据方向寄存器 DDRA 是可读可写的，而输入引脚 PINA 仅仅可读不可写。PINA 不是一个寄存器，这个地址用来访问端口 A 的物理值。当读取 PINA 时，读到的是施加于引脚上的逻辑数值。端口 A 的所有引脚都可以单独地选择上拉电阻，并且可以吸收 20 mA 的电流，所以可以直接驱动 LED 指示灯。上拉电阻被激活且引脚被拉低时，则该引脚会输出电流。

表 9.1　数据寄存器(PORTA)

7	6	5	4	3	2	1	0
PORTA7	…	…	…	…	…	…	PORTA0

表 9.2　数据方向寄存器(DDRA)

7	6	5	4	3	2	1	0
DDA7	…	…	…	…	…	…	DDA0

表 9.3　输入引脚(PINA)

7	6	5	4	3	2	1	0
PINA7	…	…	…	…	…	…	PINA0

2. 端口作为通用数据 I/O

当端口 A 作为通用数字输入输出口时，它的 8 个引脚具有完全相同的功能。数据方向寄存器 DDRA 的各位用于引脚方向的选择。当 DDRAn 被置为 1 时，则相应的端口 A 的引脚为输出引脚；反之，为输入引脚。端口 A 的配置如表 9.4 所示。

表 9.4 端口 A 的配置

PORTAn	DDAn	I/O	上 拉
0	0	输入	否
0	1	输入	否
1	0	输出	是
1	1	输出	否

3. 端口的第二功能

端口 A 的第二功能比较单一，可以作为 ADC 的模拟输入端。

4. 原理图(如图 9.1 所示)

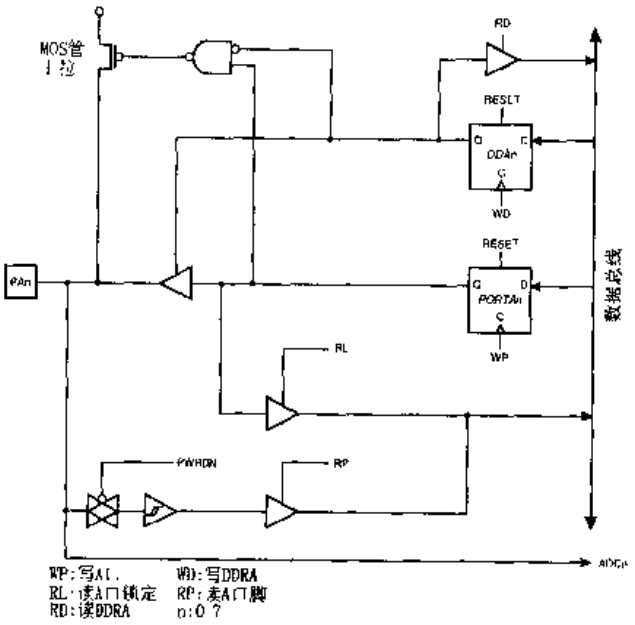


图 9.1 端口 A 的线路图(PA0~PA7)

9.1.2 端口 B

1. 端口特性

端口 B 是一个 8 位双向 I/O 口。端口 B 包含有 3 个 I/O 地址，其中有两个寄存器和一个输入引脚：数据寄存器 PORTB($18)、数据方向寄存器 DDRB($17)和输入引脚 PINB($16)，分别如表 9.5、9.6、9.7 所示。数据寄存器 PORTB 和数据方向寄存器 DDRB 是可读可写的，而输入引脚 PINB 仅仅可读不可写。PINB 不是一个寄存器，这个地址用来访问端口 B 的物

理值。当读取 PINB 时，读到的是施加于引脚上的逻辑数值。端口 B 的所有引脚都可以单独地选择上拉电阻，并且可以吸收 20 mA 的电流，所以可以直接驱动 LED 指示灯。上拉电阻被激活且引脚被拉低时，则该引脚会输出电流。

表 9.5　数据寄存器(PORTB)

7	6	5	4	3	2	1	0
PORTB7	…	…	…	…	…	…	PORTB0

表 9.6　数据方向寄存器(DDRB)

7	6	5	4	3	2	1	0
DDB7	…	…	…	…	…	…	DDB0

表 9.7　输入引脚(PINB)

7	6	5	4	3	2	1	0
PINB7	…	…	…	…	…	…	PINB0

2. 端口作为通用数据 I/O

当端口 B 作为通用数字输入输出口时，它的 8 个引脚具有完全相同的功能。数据方向寄存器 DDRB 的各位用于引脚方向的选择。当 DDRBn 被置为 1 时，则相应的端口 B 的引脚为输出引脚；反之，为输入引脚，端口 B 的配置如表 9.8 所示。

表 9.8　端口 B 的配置

PORTBn	DDBn	I/O	上拉
0	0	输入	否
0	1	输入	否
1	0	输出	是
1	1	输出	否

3. 端口的第二功能

端口 B 的第二功能和端口 A 的第二功能不同，它不像端口 A 那么单一。端口 B 的每个引脚的第二功能都有不同的定义，如表 9.9 所示。

表 9.9　端口 B 的第二功能

引脚	PB0	PB1	PB2	PB3	PB4	PB5	PB6	PB7
第二功能	T0	T1	AIN0	AIN1	SS	MOSI	MISO	SCK

引脚第二功能说明如下：

- T0：T/C0 的外部计数输入。
- T1：T/C1 的外部计数输入。
- AIN0：当该引脚无上拉电阻并且被配置为输入时，为模拟比较器的正输入端。

- AIN1：当该引脚无上拉电阻并且被配置为输入时，为模拟比较器的负输入端。
- SS：该引脚提供从机选择信号。配置为从机时，当此引脚置为输入时，此引脚将激活 SPI；配置为主机时，DDB4 控制它的方向。
- MOSI：SPI 的主机数据输出，从机数据输入。
- MISO：SPI 的主机数据输入，从机数据输出。
- SCK：SPI 的主机时钟输出，从机时钟输入。

4. 原理图

PB0~PB1 端口的原理图分别如图 9.2~图 9.7 所示。

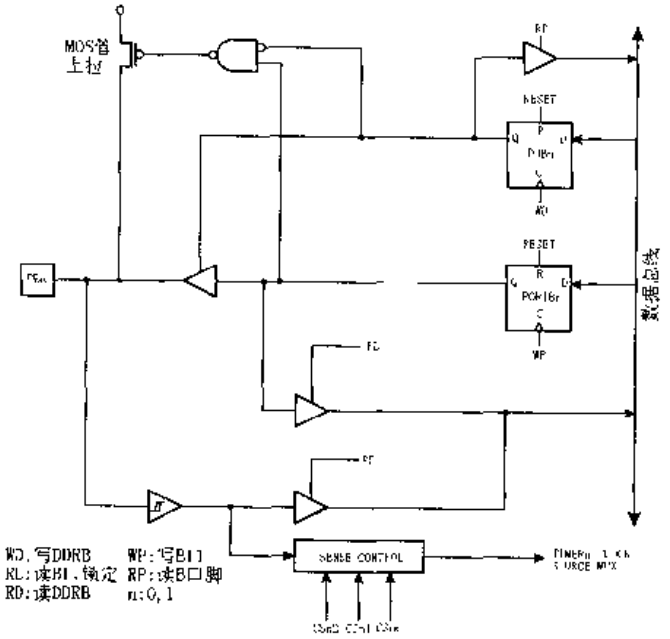


图 9.2 端口 B 的原理图(PB0 和 PB1)

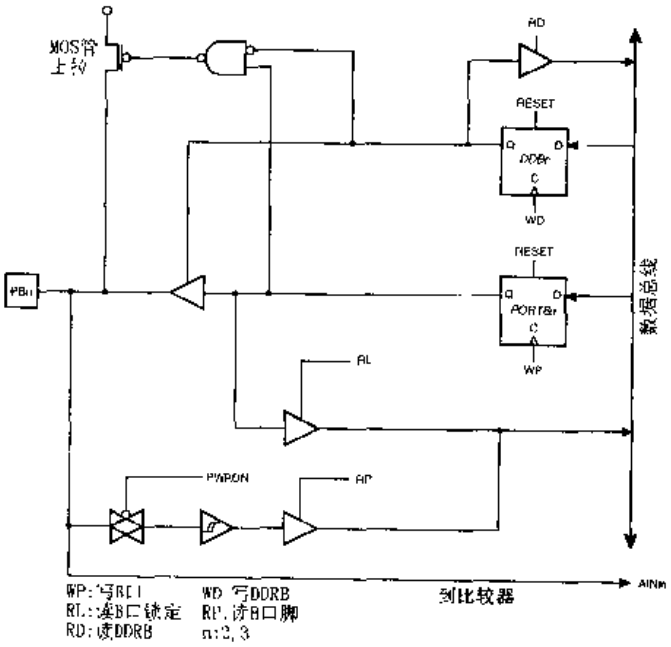


图 9.3 端口 B 的原理图(PB2 和 PB3)

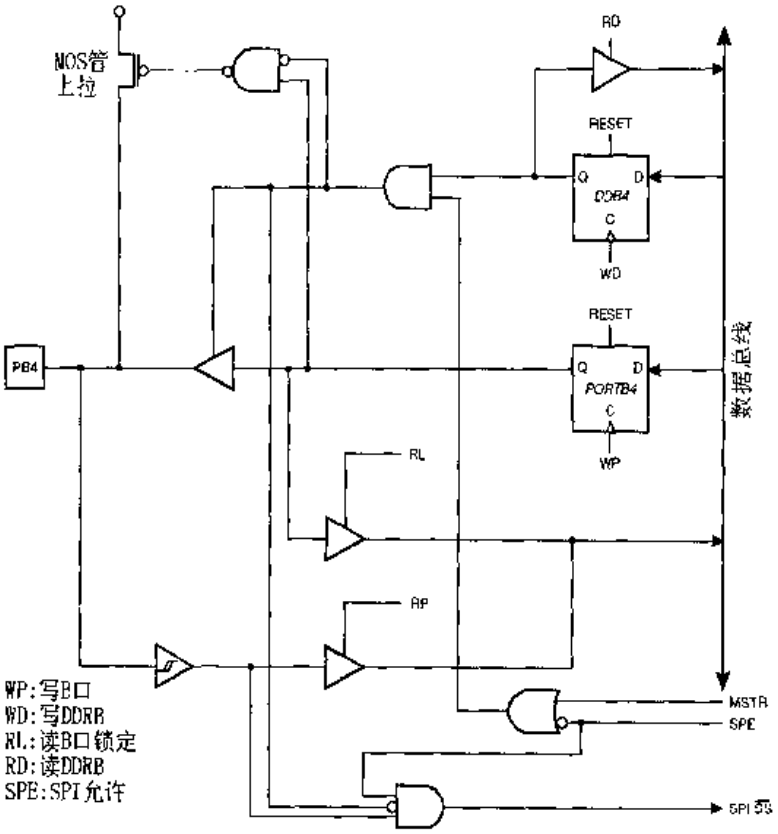


图 9.4　端口 B 的原理图(PB4)

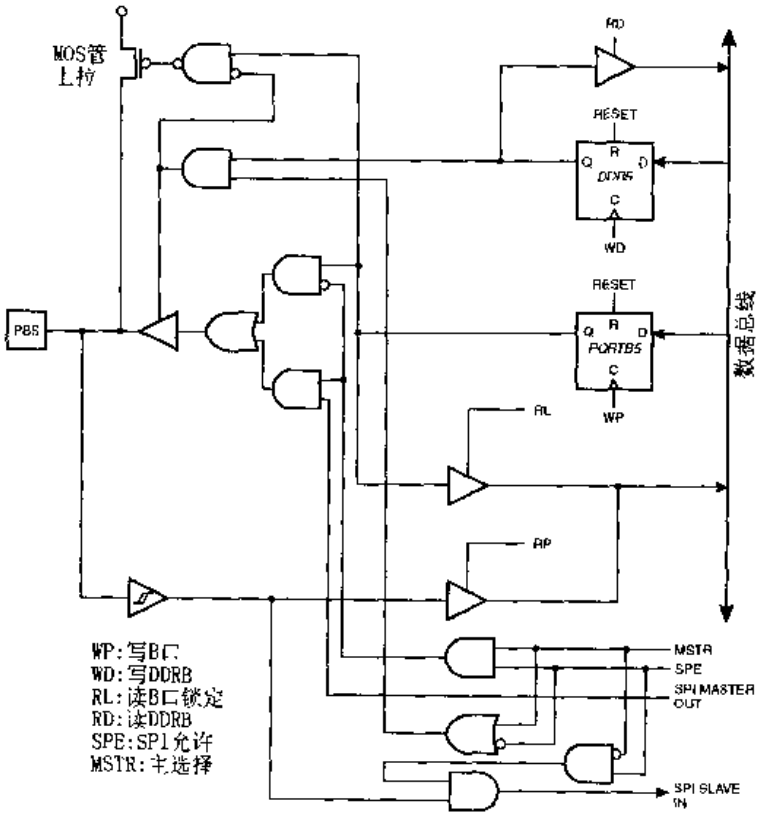


图 9.5　端口 B 的原理图(PB5)

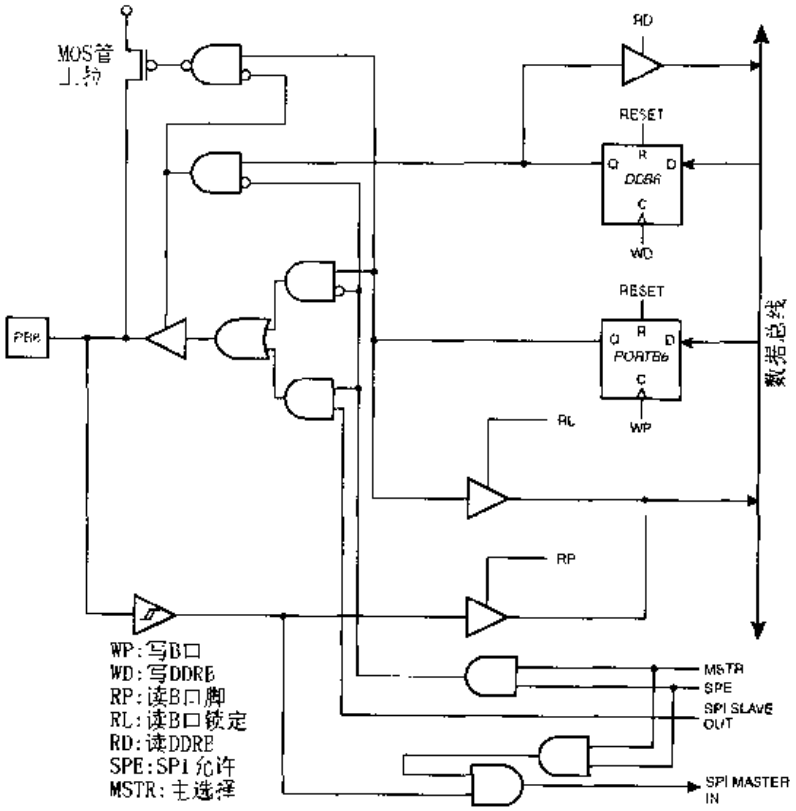


图 9.6 端口 B 的原理图(PB6)

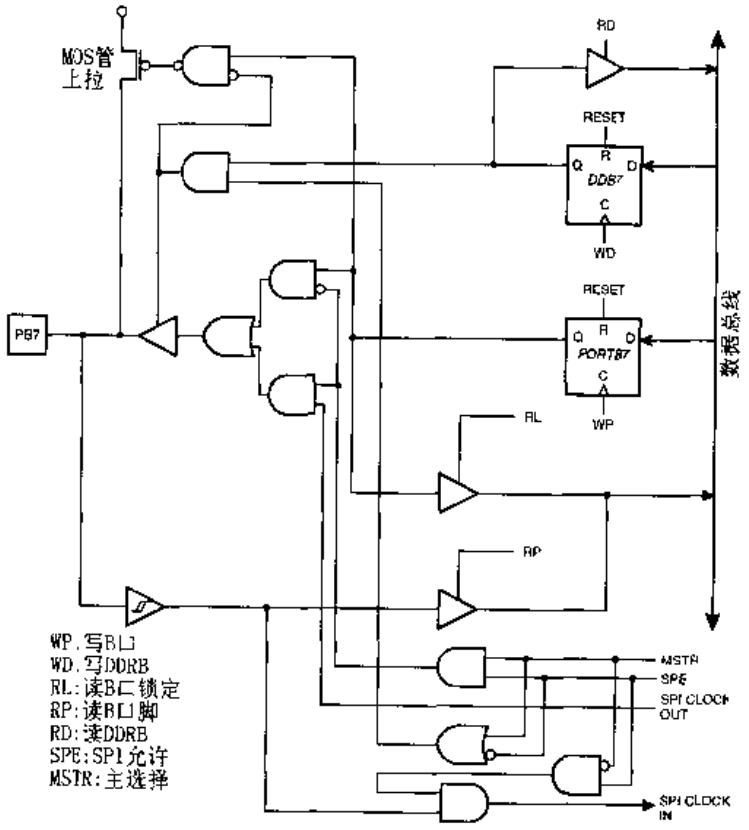


图 9.7 端口 B 的原理图(PB7)

9.1.3　端口 C

1. 端口特性

端口 C 是一个 8 位双向 I/O 口。端口 B 包含有 3 个 I/O 地址，其中有两个寄存器和一个输入引脚：数据寄存器 PORTC($15)、数据方向寄存器 DDRC($14)和输入引脚 PINC($13)，分别如表 9.10、9.11、9.12 所示。数据寄存器 PORTC 和数据方向寄存器 DDRC 是可读可写的，而输入引脚 PINC 仅仅可读不可写。PINC 不是一个寄存器，这个地址用来访问端口 C 的物理值。当读取 PINC 时，读到的是施加于引脚上的逻辑数值。端口 C 的所有引脚都可以单独地选择上拉电阻，并且可以吸收 20 mA 的电流，所以可以直接驱动 LED 指示灯。上拉电阻被激活且引脚被拉低时，则该引脚会输出电流。

表 9.10　数据寄存器(PORTC)

7	6	5	4	3	2	1	0
PORTC7	…	…	…	…	…	…	PORTB0

表 9.11　数据方向寄存器(DDRC)

7	6	5	4	3	2	1	0
DDC7	…	…	…	…	…	…	DDC0

表 9.12　输入引脚(PINC)

7	6	5	4	3	2	1	0
PINC7	…	…	…	…	…	…	PINC0

2. 端口作为通用数据 I/O

当端口 C 作为通用数字输入输出口时，它的 8 个引脚具有完全相同的功能。数据方向寄存器 DDRC 的各位用于引脚方向的选择。当 DDCn 被置为 1 时，则相应的端口 C 的引脚为输出引脚；反之，为输入引脚。端口 B 的配置如表 9.13 所示。

表 9.13　端口 B 的配置

PORTCn	DDCn	I/O	上拉
0	0	输入	否
0	1	输入	否
I	0	输出	是
1	1	输出	否

3. 原理图

PC0~PC7 端口的原理图分别如图 9.8、图 9.9 和图 9.10 所示。

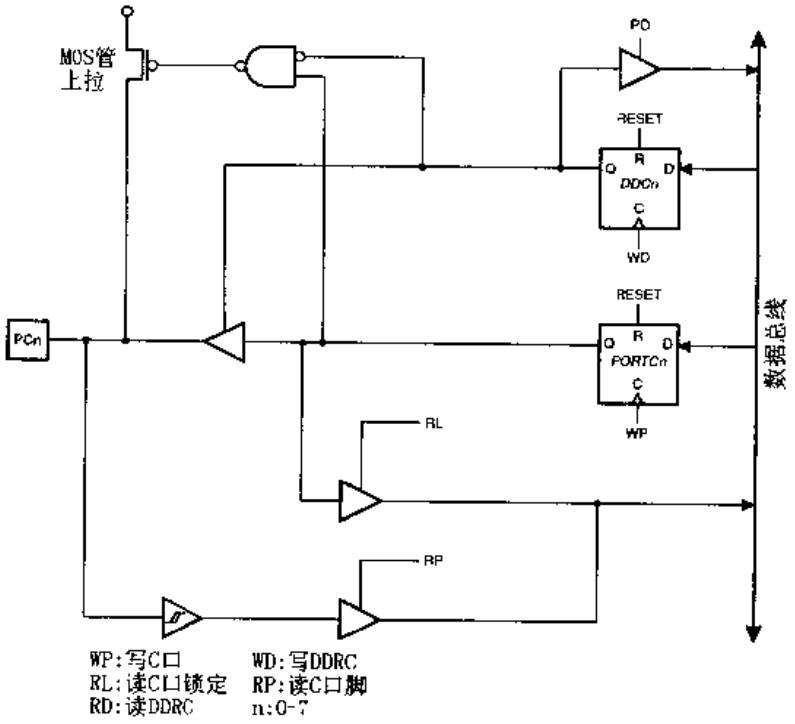


图 9.8 端口 C 的原理图(PC0—PC5)

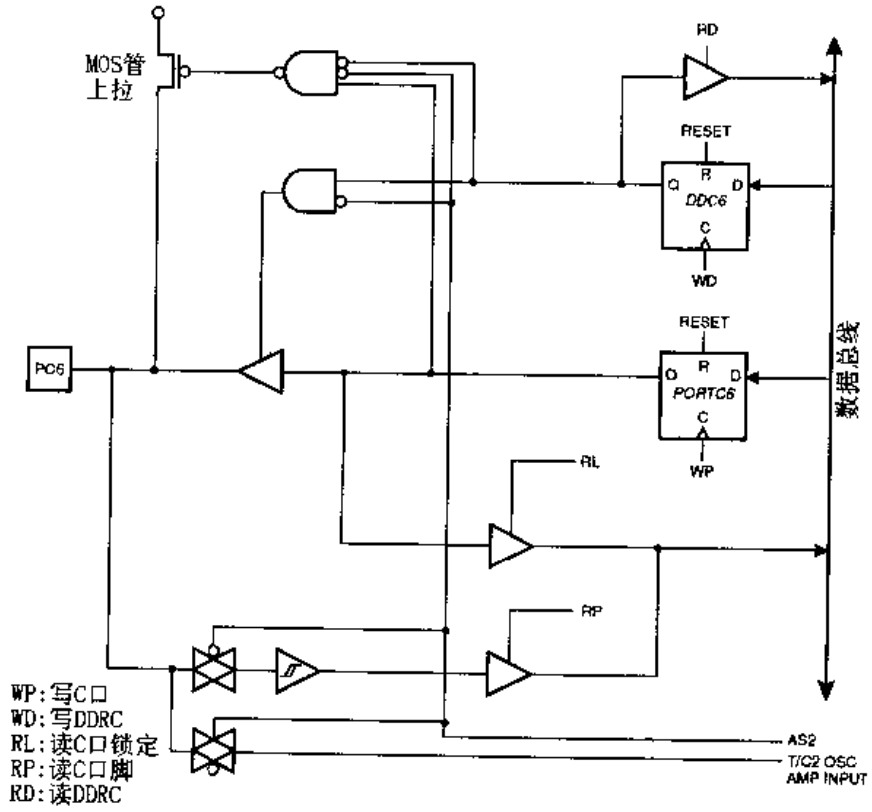


图 9.9 端口 C 的原理图(PC6)

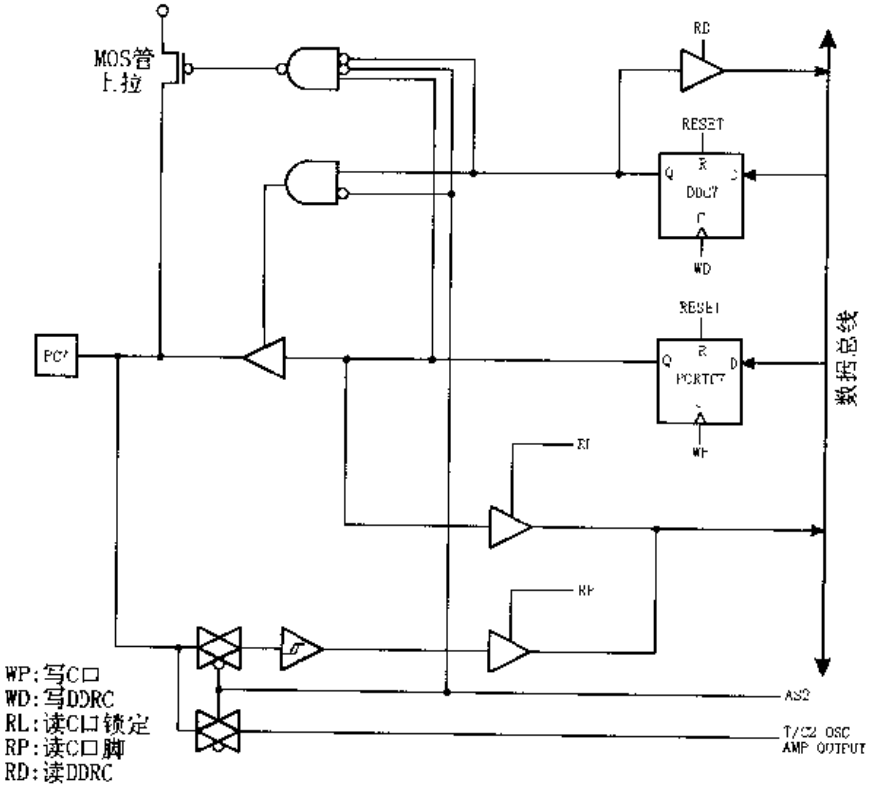


图 9.10　端口 C 的原理图(PC7)

9.1.4　端口 D

1. 端口特性

端口 D 是一个 8 位双向 I/O 口。端口 D 包含有 3 个 I/O 地址，其中有两个寄存器和一个输入引脚：数据寄存器 PORTB($12)、数据方向寄存器 DDRB($11)和输入引脚 PINB($10)，分别如表 9.14、9.15、9.16 所示。数据寄存器 PORTD 和数据方向寄存器 DDRD 是可读可写的，而输入引脚 PIND 仅仅可读不可写。PIND 不是一个寄存器，这个地址用来访问端口 D 的物理值。当读取 PIND 时，读到的是施加于引脚上的逻辑数值。端口 D 的所有引脚都可以单独地选择上拉电阻，并且可以吸收 20 mA 的电流，所以可以直接驱动 LED 指示灯。上拉电阻被激活且引脚被拉低时，则该引脚会输出电流。

表 9.14　数据寄存器(PORTD)

7	6	5	4	3	2	1	0
PORTD7	…	…	…	…	…	…	PORTD0

表 9.15　数据方向寄存器(DDRD)

7	6	5	4	3	2	1	0
DDD7	…	…	…	…	…	…	DDD0

表 9.16 输入引脚(PIND)

7	6	5	4	3	2	1	0
PIND7	…	…	…	…	…	…	PIND0

2. 端口作为通用数据 I/O

当端口 D 作为通用数字输入输出口时，它的 8 个引脚具有完全相同的功能。数据方向寄存器 DDRD 的各位用于引脚方向的选择。当 DDRDn 被置为 1 时，则相应的端口 D 的引脚为输出引脚；反之，为输入引脚。端口 D 的配置如表 9.17 所示。

表 9.17 端口 D 的配置

PORTDn	DDDn	I/O	上 拉
0	0	输入	否
0	1	输入	否
1	0	输出	是
1	1	输出	否

3. 端口的第二功能

端口 D 和端口 B 有些相似，每个引脚的第二功能都有不同的定义，如表 9.18 所示。

表 9.18 端口 B 的第二功能

引脚	PD0	PD1	PD2	PD3	PD4	PD5	PD6	PD7
第二功能	RXD	TXD	INT0	INT1	OC1B	OC1A	ICP	OC2

将端口 D 引脚的第二功能进行如下说明：

- RXD：UART 的数据输入引脚。当 UART 接收器触发时，不论 DDR0 为何值，该引脚自动配置为输入。当 UART 将该引脚强行置 1 时，PORTD 被内部上拉为逻辑 1。
- TXD：RART 的数据接收引脚。当 UART 接收器触发时，不论 DDR1 为何值，该引脚自动配置为输出。
- INT0：外部中断源 0。该引脚可以作为外部中断源的输入引脚，详细内容见关于中断的部分。
- INT1：外部中断源 1。该引脚可以作为外部中断源的输入引脚，详细内容见关于中断的部分。
- OC1B：比较匹配 B 的外部输出。该引脚可以作为 T/C1 的比较匹配输出。此时，该引脚必须配置为输出。同时，它还是 PWM 的输出引脚。
- OC1A：比较匹配 A 的外部输出。该引脚可以作为 T/C1 的比较匹配输出。此时，该引脚必须配置为输出。同时，它还是 PWM 的输出引脚。
- ICP：输入捕获引脚。该引脚可以作为 T/C1 的输入捕获引脚。要实现此功能，必须将它配置为输入。

- OC2：比较匹配的外部输出。该引脚可以作为 T/C2 的比较匹配输出。此时，该引脚必须配置为输出。同时，它还是 PWM 的输出引脚。

4. 原理图

PD0~PD7 的原理图分别如图 9.11~图 9.16 所示。

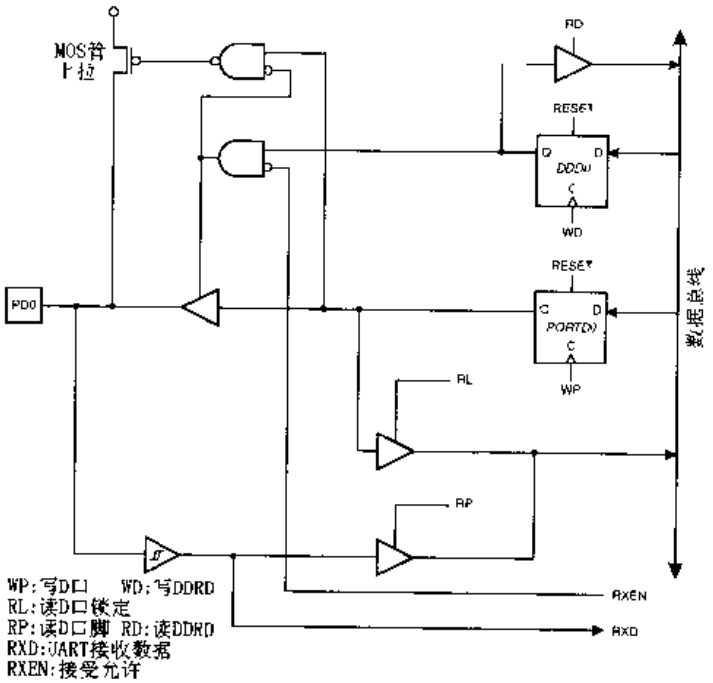


图 9.11　端口 D 的原理图(PD0)

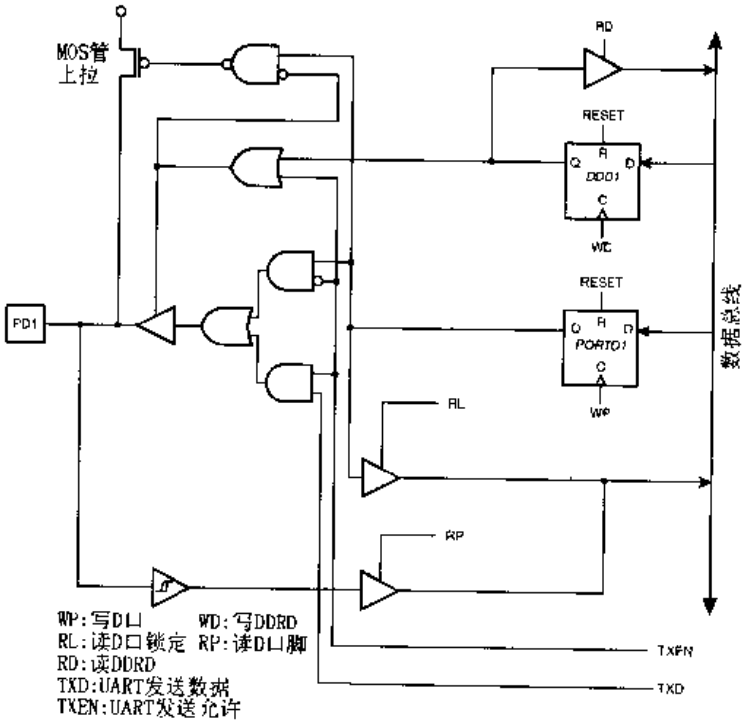


图 9.12　端口 D 的原理图(PD1)

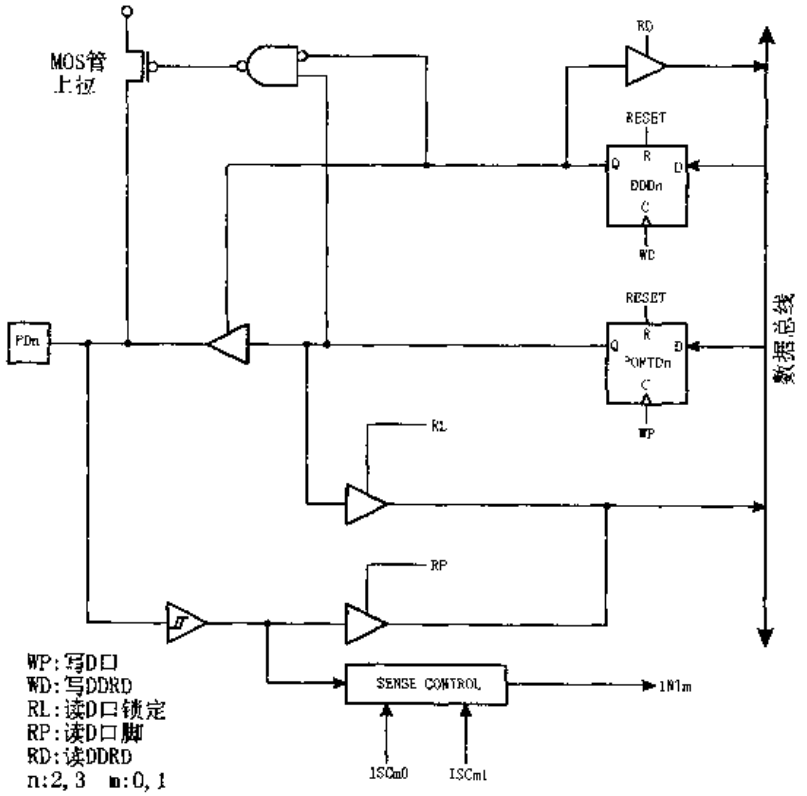


图 9.13 端口 D 的原理图(PD2 和 PD3)

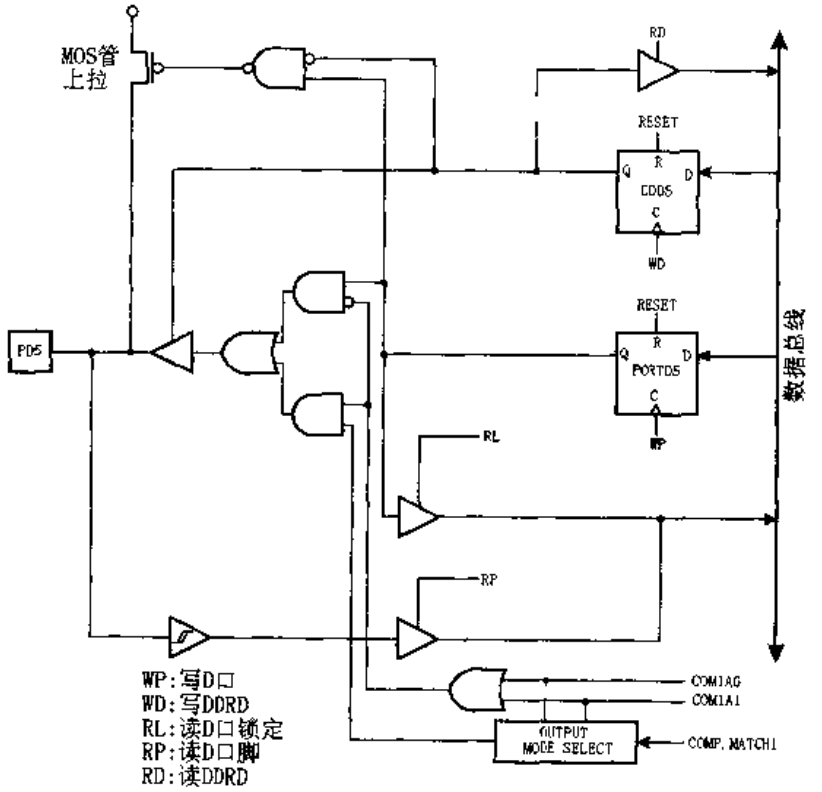


图 9.14 端口 D 原理图(PD4 和 PD5)

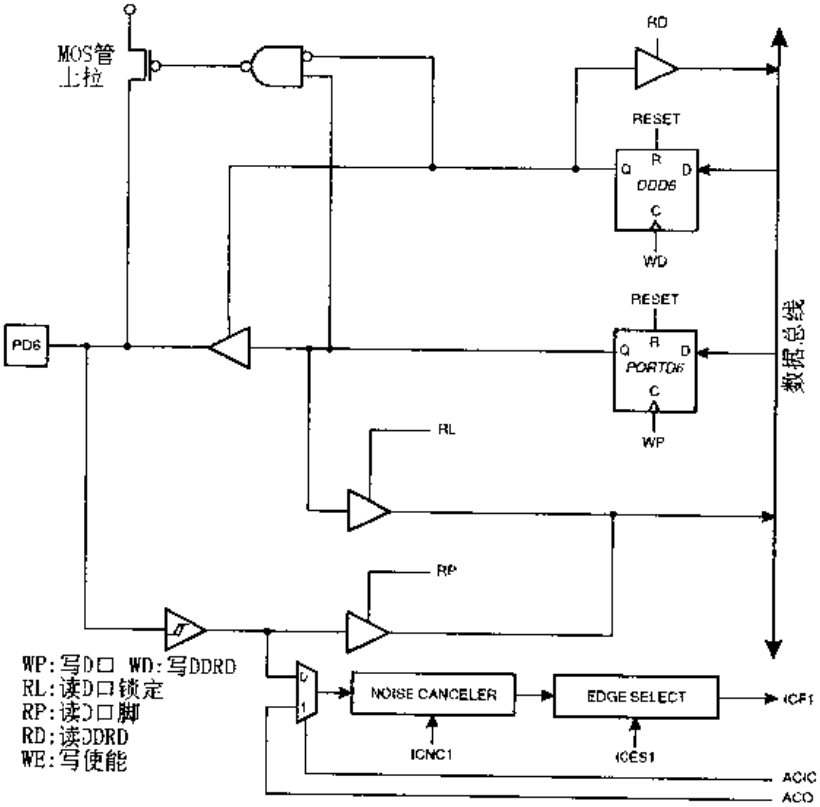


图 9.15　端口 D 的原理图(PD6)

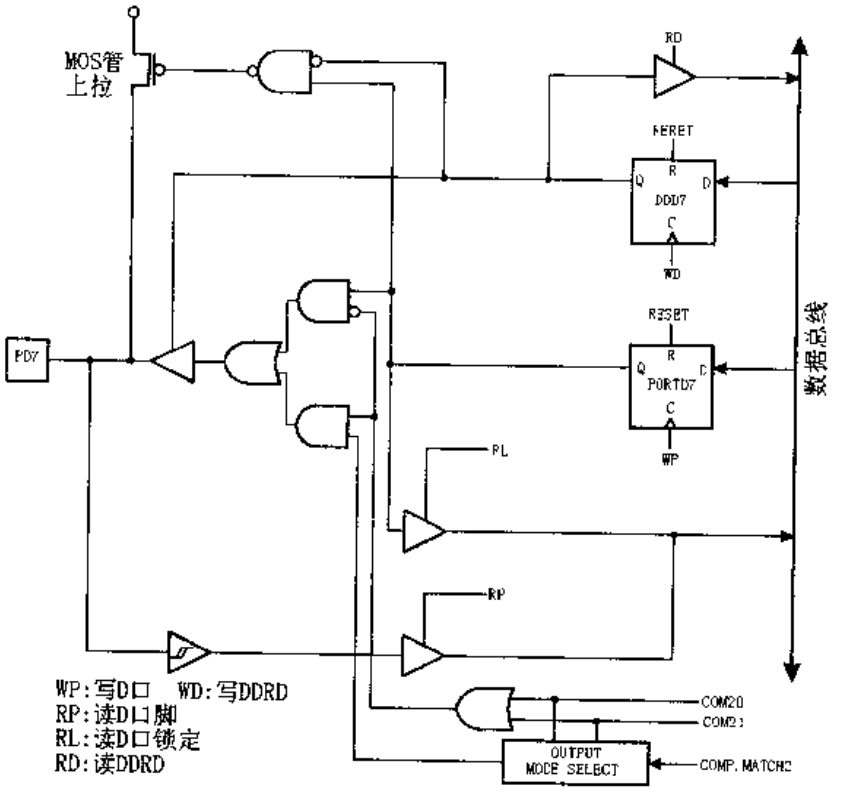


图 9.16　端口 D 的原理图(PD7)

9.1.5 I/O 口的编程

以下程序将 16 位数据从 A、B 口输入，从 C、D 口输出。

源程序为：

```
#include "io8535.h"
#define uchar unsigned char
//主程序
void main()
{
    uchar mid1,mid2;
    DDRA=0x00;        //设置 A、B 口为带上拉的输入口
    DDRB=0x00;
    PORTA=0xff;
    PORTB=0xff;
    DDRC=0xff;        //设置 C、D 口为输出口
    DDRD=0xff;
    mid1=PINA;        //从 A 口输入数据
    mid2=PINB         //从 B 口输入数据
    PORTC=mid1;       //输出 mid1 值至 PORTC
    PORTD=mid2;       //输出 mid2 值至 PORTD

}
```

9.2 中　　断

什么是中断？我们先看一个生活中的例子。当你正在家中看书时，突然电话响了。这时你放下手中的书，去接电话，并和来电话的人交谈，然后你放下电话，回来继续看书。这就是生活中的“中断”现象。计算机的中断与此类似，它是指正在执行正常程序的计算机，由于出现一些需要立即进行处理的情况，CPU 暂停现在正在执行的操作，转而执行中断服务程序，执行完毕后，再返回到原来程序继续执行的过程。

9.2.1 单片机的中断功能

1. 中断的作用

中断技术在单片机上能实现很多的功能，它们主要包括：

- 实时控制功能

  所谓实时控制就是单片机能及时完成对受控对象的测量、计算、分析和控制，从而使受控对象保持良好的工作状态，并达到系统的使用要求。单片机的中断技术使控制参量能随时向计算机发出中断请求，以完成对数据的及时处理，因此它是单片机实现实时控制功能的一个必然要求。
- 实现单片机与低速外设的配合

由于许多外设的速度较慢，无法与单片机实现直接的数据交换，因此，必须采用中断功能来协调单片机与外设的速度。当单片机在执行程序的过程中，如要进行数据的输出/输出操作，则单片机先发一个命令给外设，然后单片机继续执行程序，当外设为数据交换准备好以后，它向单片机发出中断请求。这时，单片机暂时中止正在执行的程序，转而执行中断服务程序中的数据输入/输出程序，数据交换完成后，单片机再返回继续执行原来的程序。由此可见，中断技术实现了单片机与外设的速度配合，提高了单片机的工作效率，也提高了数据交换的效率。

2. 单片机的中断处理过程

单片机检测到外设或内部的中断请求以后，要执行一些特定的操作，然后再转去执行中断服务程序，中断服务完成以后，单片机还必须更改一些寄存器的内容才可以返回到原有程序。详细的中断处理过程如下：

- 现场保护

  由于中断服务程序的执行会破坏单片机内某些寄存器的内容，因此，为了避免中断服务程序完成后影响原有程序的执行，单片机必须要将有关寄存器的内容压入堆栈，也就是必须进行现场保护。
- 中断服务程序

  中断服务程序是对中断进行处理的一个子程序。
- 现场恢复

  中断服务程序完成以后，为了使单片机返回到主程序中继续执行，单片机必须将现场保护时保存下来的内容从堆栈中弹出至相应的寄存器，也即现场恢复。

9.2.2　AT90LS8535 单片机的中断系统

1. 中断源

AT90LS8535 单片机具有 16 个中断源，其中外部中断两个，定时/计数器中断 7 个，串行口中断 4 个，模数转换完成中断、EEPROM 中断和模拟比较器中断各 1 个。

- 外部中断

  AT90LS8535 单片机的外部中断请求由外设发出。

  外设可以通过在 PD2、PD3 上施加低电平、下降沿脉冲或上升沿脉冲触发该中断。
- 定时/计数器中断

  定时/计数器中断发生在单片机内部的 3 个定时/计数器上，当某个定时/计数器的计数值溢出、输出比较匹配或输入捕捉事件发生，且相应的控制寄存器被设置为中断允许时，单片机响应该中断。
- 串行口中断

  串行口中断是为串行数据的发送或接收设置的。每当串行口发送或接收完一个串行数据帧时，其相应的状态位被置位。如果此时该类型的中断为允许，则单片机响应该中断。
- 模数转换完成中断

AT90LS8535 单片机的模数转换完成中断用于标识单片机内部的模数转换器。当模数转换器的一次模数转换及数据更新完成时，ADIF 置“1”，若此时 ADIE(ADC 中断使能位)和全局中断使能位都为“1”，则该中断被响应。

- EEPROM 中断

  EEPROM 中断是为 AT90LS8535 单片机内部的 EEPROM 写操作设置的，当 EEPROM 准备好，且 EERIE(EEPROM 准备好中断使能)和全局中断使能位都为“1”时，该中断被响应。

- 模拟比较器中断

  模拟比较器中断发生在单片机的模拟比较器的输出发生变化时，这种变化可以是一个上升沿，可以是一个下降沿，也可以是一个电平变化，用户可以通过其控制寄存器和状态寄存器(ACSR)来设置。

2. 与中断有关的寄存器

- 通用中断屏蔽寄存器(GIMSK)

  通用中断屏蔽寄存器用于开启和关闭外部中断功能，其格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
INT1	INT0	–	–	–	–	–	–

  INT1(外部中断 1)

  该位置“1”，且全局中断使能位也为“1”时，PD3 引脚上的一个中断触发信号将产生一个外部中断请求 1。

  INT0(外部中断 0)

  该位置“1”，且全局中断使能位也为“1”时，PD2 引脚上的一个中断触发信号将产生一个外部中断请求 0。

  D5～D0(保留位)

- 通用中断标志寄存器(GIFR)

  通用中断标志寄存器用来标识两个外部中断信号，各位功能如下：

D7	D6	D5	D4	D3	D2	D1	D0
INTF1	INTF0	–	–	–	–	–	–

  NTF1(外部中断 1 标志位)

  当 PD3 引脚上的一个信号触发了由 ISC11 和 ISC10 设置的中断触发信号时，该位被置“1”，如果此时全局中断使能位以及通用中断屏蔽寄存器(GIMSK)中的 INT1 都为“1”，则单片机产生一个外部 1 中断。

  当外部中断 1 的服务程序被响应时，INTF1 被置“0”。INTF1 也可通过向该位写入“0”来清除。

  INTF0(外部中断 0 标志位)

  当 PD2 引脚上的一个信号触发了由 ISC01 和 ISC00 设置的中断触发信号时，该位被置“1”，如果此时全局中断使能位以及通用中断屏蔽寄存器(GIMSK)中的 INT0

都为“1”，则单片机产生一个外部 0 中断。

当外部中断 0 的服务程序被响应时，INTF0 被置“0”。INTF0 也可通过向该位写入“0”来清除。

D5～D0(保留位)

- 定时/计数器中断屏蔽寄存器(TIMSK)

定时/计数器中断屏蔽寄存器用于开启和关闭定时/计数器的中断功能，各位功能如下。

D7	D6	D5	D4	D3	D2	D1	D0
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0

OCIE2(定时/计数器 2 输出比较匹配中断使能)

该位为“1”，且全局中断使能位也为“1”时，定时器/计数器 2 的比较匹配事件将触发一个单片机中断。

TOIE2(定时/计数器 2 溢出中断使能)

当该位和全局中断使能位都为“1”时，定时/计数器 2 的溢出中断使能。如果此时定时/计数器 2 产生一个溢出事件，则将触发一个单片机中断。

TICIE1(定时/计数器 1 输入捕获中断使能)

该位为“1”，且全局中断使能位也为“1”时，定时器/计数器 1 的输入捕获事件中断被开启。此时在引脚 20 上的一个输入捕获事件将触发单片机中断。

OCIE1A(定时/计数器 1 输出比较 A 匹配中断使能)

该位为“1”，且全局中断使能位也为“1”时，定时器/计数器 1 的比较 A 匹配事件将触发一个单片机中断。

OCIE1B(定时/计数器 1 输出比较 B 匹配中断使能)

该位为“1”，且全局中断使能位也为“1”时，定时器/计数器 1 的比较 B 匹配事件将触发一个单片机中断。

TOIE1(定时/计数器 1 溢出中断使能)

当该位和全局中断使能位都为“1”时，定时/计数器 1 的溢出中断使能，如果此时定时/计数器 1 产生一个溢出事件，则将触发一个单片机中断。

D1(保留位)

TOIE0(定时/计数器 0 溢出中断使能)

当该位和全局中断使能位都为“1”时，定时/计数器 0 的溢出中断使能，如果此时定时/计数器 1 产生一个溢出事件，则将触发一个单片机中断。

- 定时/计数器中断标志寄存器(TIFR)

定时/计数器中断标志寄存器用来标识定时/计数器中断信号，各位功能如下：

D7	D6	D5	D4	D3	D2	D1	D0
OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	–	TOV0

OCF2(输出比较标志 2)

当定时/计数器 2 和输出比较寄存器 2(OCR2)发生匹配事件时，该标志位置“1”。

单片机响应相应的中断，OCF2 被清除。OCF2 也可通过向该位写入“0”来清除。

TOV2(定时/计数器 2 溢出标志位)

当定时/计数器 2 溢出，该位被置“1”。

单片机响应相应的中断后，TOV2 被硬件清 0。TOV2 也可通过向该位写入“0”来清除。

ICF1(输入捕获中断标志 1)

定时/计数器 1 发生输入捕获事件后，ICF1 位被置“1”。

单片机响应该中断后，ICF1 被硬件清 0。此外，ICF1 也可通过向该位写入“0”来清除。

OCF1A(输出比较标志 1A)

当定时/计数器 1 和输出比较寄存器 1A(OCR1A)发生比较匹配事件时，该标志位置“1”。

单片机响应该中断后，OCF1A 被清除。此外 OCF1A 也可通过向该位写入“0”来清除。

OCF1B(输出比较标志 1B)

当定时/计数器 1 和输出比较寄存器 1B(OCR1B)发生比较匹配事件时，该标志位置“1”。

单片机响应该中断后，OCF1B 被清除。此外 OCF1B 也可通过向该位写入“0”来清除。

TOV1(定时/计数器 1 溢出标志位)

当定时/计数器 1 溢出，该位被置“1”。

单片机响应该中断后，TOV1 被硬件清 0。TOV1 也可通过向该位写入“0”来清除。

D1(保留位)

TOV0(定时/计数器 1 溢出标志位)

当定时/计数器 0 溢出，该位被置“1”。

单片机响应该中断后，TOV0 被硬件清 0。TOV0 也可通过向该位写入“0”来清除。

- MCU 控制寄存器(MCUCR)

MCU 控制寄存器中的低 4 位用于选择外部中断的触发方式，其格式为：

D7	D6	D5	D4	D3	D2	D1	D0
–	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00

D7(保留位)

SE(睡眠使能)

SM1、SM0(睡眠模式选择位)

ISC11、ISC10(中断触发选择位)

ISC11 和 ISC10 用于选择外部中断 1 的中断触发方式，如表 9.19 所示。

表 9.19　外部中断 1 的中断触发方式

ISC11	ISC10	说　明
0	0	INT1 引脚上的低电平产生中断请求
0	1	保留
1	0	INT1 引脚上的下降沿产生中断请求信号
1	1	INT1 引脚上的上升沿产生中断请求信号

ISC01、ISC00(中断触发选择位)

ISC01 和 ISC00 用于选择外部中断 0 的中断触发方式，如表 9.20 所示。

表 9.20　外部中断 0 的中断触发方式

ISC01	ISC00	说　明
0	0	INT0 引脚上的低电平产生中断请求
0	1	保留
1	0	INT0 引脚上的下降沿产生中断请求信号
1	1	INT0 引脚上的上升沿产生中断请求信号

9.2.3　ICC AVR C 编译器的中断操作

ICC AVR 的 C 编译器支持在 C 源程序中直接开发中断程序。当用户使用该功能时，必须在中断服务子程序定义之前用“pragma”语句通知编译器，该子程序是一个中断操作。

用“pragma”语句定义中断服务子程序的格式为：

#pragma interrupt_handler (中断函数名)：(中断向量号)

其中，中断函数名为用户定义的中断服务子程序的名称，而中断向量号则用于表明中断的类型。

通过附加该语句，ICC AVR 的 C 编译器在中断服务子程序后生成 RETI 指令，并且产生保存和恢复在函数中使用的全部寄存器。

例：

```
#pragma interrupt_handler int_0 2
……
void int_0()
{
……
}
```

9.2.4　中断的编程

本例通过外部中断方式实现单片机与低速外设的数据输入/输出操作，电路的原理如图 9.17 所示。

AT90LS8535 的 A 口作为数据线同 74LS377 的 D0～D7 口和 74LS373 的 Q0～Q7 口相

连。当外设为数据输入准备好以后，外设发出一个/STROBE 信号将数据锁存到 74LS373 中，与此同时，/STROBE 信号将使单片机产生一个外部中断 0。单片机响应该中断，并在中断服务程序中读取 373 中锁存的数据，然后再将数据输出至 74LS377 中。AT90LS8535 的中断应用电路如图 9.17 所示。

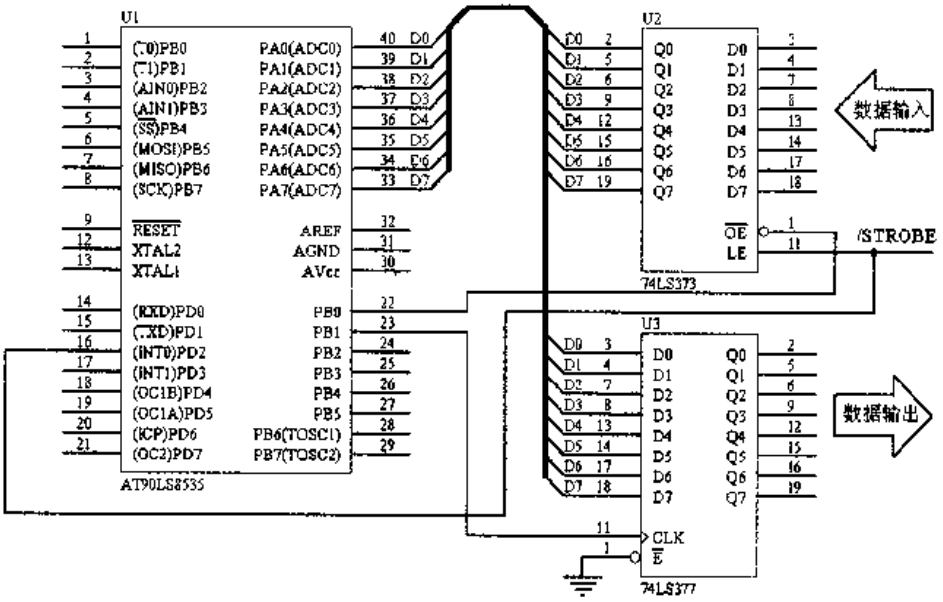


图 9.17 AT90LS8535 的中断应用电路

源程序为：

```
#include "io8535.h"
#define uchar unsigned char

#pragma interrupt_handler Receivedata:2

//中断服务子程序
void  Receivedata(void)
{
    uchar mid;
    DDRA=0;
    PORTB&=~BIT(PB0);    //置 373/OE 为低电平
    mid=PINA;            //数据输入
    PORTB|=BIT(PB0);     //置 373/OE 为高电平
    Senddata(mid);
}

//数据发送子程序
void Senddata(uchar x)
{
    DDRA=0xff;
    PORTA=x;             //输出 x 至端 PA 端口
    PORTB|=BIT(PB1);     //置 377CLK 为高电平
    PORTB&=~BIT(PB1);    //置 377CLK 为低电平
```

```
}
//主程序
void main()
{
    SREG=0x80;          //开中断
    GIMSK=0x40;
    MCUCR=0x02;
    DDRA=0;          //初始化 A、B 端口
    DDRB=0xff;
    PORTB|=BIT(PB0);
    PORTB&=~BIT(PB1);
    do{
        ;
        }while(1);
}
```

9.3　串行数据通信

计算机的数据通信有两种方式，即并行数据通信和串行数据通信。

并行数据通信的各个数据位同时传送，因此传送的速度快。但它需要多根数据线，所以数据通信的成本较高。

串行数据通信的数据按位顺序传送，因此至少只需一根线即可实现数据的交换。但由于串行通信在任一时刻只能发送或接收一个数据位，因此它的速度较慢，只适合低速的应用场合。

串行数据通信又可分为同步串行通信和异步串行通信两种。在单片机应用系统中，同步串行通信主要用于单片机和外设以及单片机与单片机之间的高速数据传送；而异步串行通信则主要用于单片机与外设之间的低速数据传送。

9.3.1　数据通信基础

1. 同步串行通信的基本原理

同步串行通信以字节为单位，即每次传送一个字节的数据。工作于同步串行通信方式的串行口实际上相当于一个同步移位寄存器，这时串行口的一个端口作为数据移位的入口或出口，而另一个端口则提供移位所需要的时钟脉冲。同步串行通信的数据传送格式如图 9.18 所示。

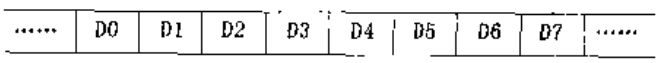


图 9.18　同步串行通信的数据格式

图 9.19 就是一个同步串行通信的电路，数据通过移位脉冲 CP 实现串行口数据的移位输入/输出，从而完成数据的串行传送。

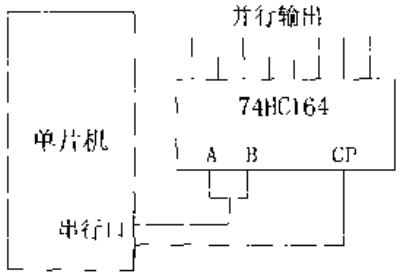


图 9.19 通过同步串行口传送数据

2. 异步串行通行的基本原理

异步串行通信以帧为单位，即每次传送一个数据帧。与同步串行通信不同，异步串行通信并不需要一个时钟脉冲端口，它通过特定的帧格式和内部时钟电路实现数据的正确传送。串行数据通信的帧格式如图 9.20 所示。

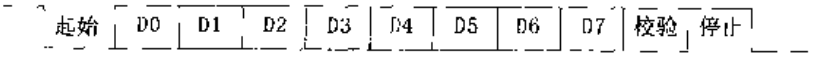


图 9.20 异步串行通信的数据格式

串行通信的数据帧说明：

- 起始位：串行数据发送端通过发送一个起始位而开始一个数据帧的传送。
- 数据位：起始位之后传送的数据就是数据位。串行通信协议规定：数据位的低位在前，高位在后。
- 奇偶校验位：该位用于对数据传送的正确性做出检验。用户可以根据实际需要，对传送的数据做奇校验、偶校验或不做校验。
- 停止位：停止位用于表示一个数据帧传送的结束。停止位的长度也可根据用户的需要设定为 1 位、1.5 位或 2 位。
- 位时间：位时间表示一个位所占据的时间长度。

9.3.2 AT90LS8535 的同步串行接口

AT90LS8535 的片内含有一个同步串行接口(SPI)用于单片机与外设或几个单片机之间进行高速同步数据传送。对于 SPI 主机来说，发送的数据由 MOSI 端口送出，接收的数据由 MISO 端口输入；对于 SPI 从机来说，发送的数据由 MISO 端口送出，接收的数据由 MOSI 端口输入。AT90LS8535 的 SPI 接口具有两个数据寄存器，一个作为移位输出用，一个作为移位输入用，由于单片机对移位输出寄存器只能写入，对移位输入寄存器只能读出，因此它们只占据一个内部 I/O 地址。

1. SPI 接口的内部结构

AT90LS8535 的 SPI 接口如图 9.21 所示。

如图，SCK 是 SPI 主机的时钟输出端和 SPI 从机的时钟输入端，当单片机将数据写入主机 SPI 接口的数据寄存器时，主机的 SPI 时钟启动，从而将数据从主机的 MOSI 口输出至从机的 MOSI 口。在一个字节被移出后，SPI 的时钟立即停止发送，并置传送停止标志

位(SPIF)为“1”，这时，若 SPI 控制寄存器(SPCR)的中断触发控制位(SPIE)为“1”，则单片机产生一个中断请求。/SS 端口用来设置 SPI 接口的主从关系，当/SS=1 时，单片机为 SPI 主机；当/SS=0 时，单片机为 SPI 从机。图 9.22 为主从 SPI 的连接方式.

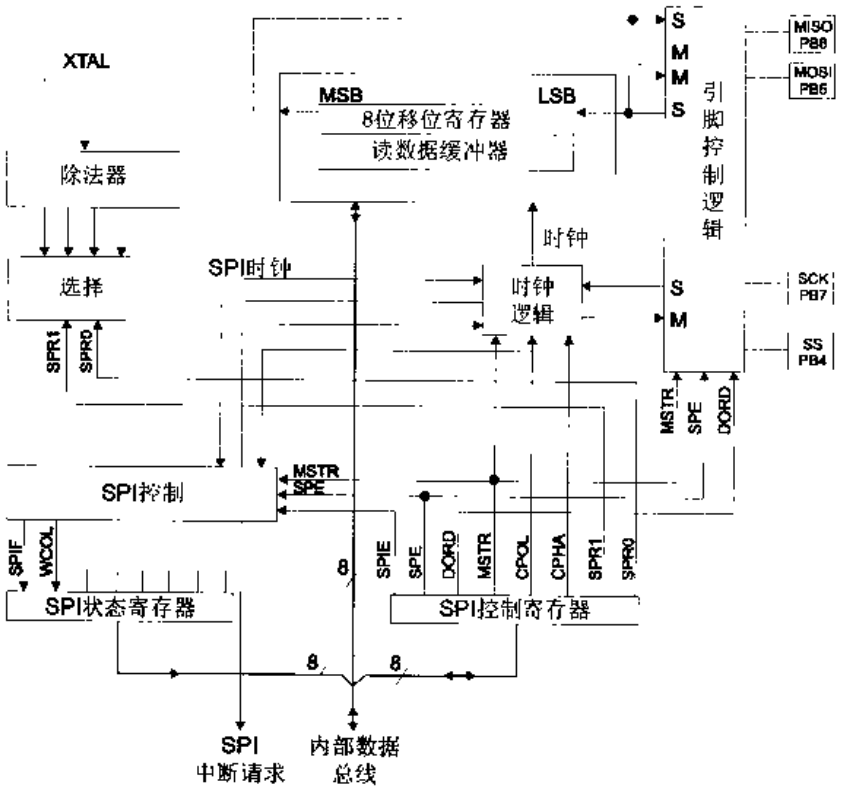


图 9.21　AT90LS8535 的 SPI 接口结构

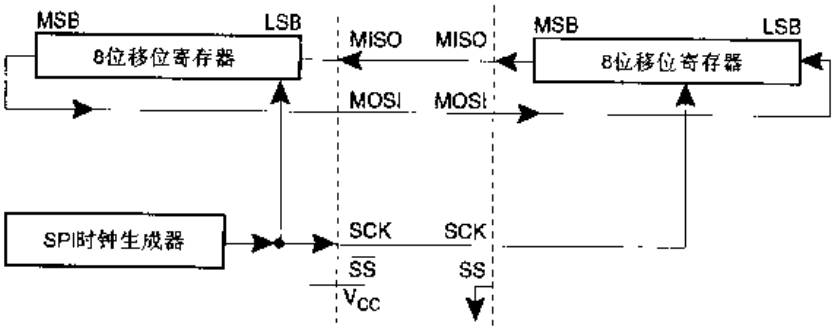


图 9.22　主从 SPI 接口的连接

由图可知，主从 SPI 的两个 8 位移位寄存器实际上构成了一个 16 位的寄存器，这个寄存器的首尾相连，因此当数据从主机移入从机的同时，数据也由从机移入了主机。

注意： AT90LS8535 的 SPI 接口在发送方向上单缓冲，而在接收方向是双缓冲。这表示在所有的移位操作完成前，被发送的字节不能被写入 SPI 的数据寄存器；但 SPI 在接收数据时，已经收到的数据必须在下一个字节被完全移入之前从 SPI 数据寄存器中读出，否则这个字节就会丢失。

2. SPI 接口的数据传输模式

对于串行数据来说，SCK 的相位和极性有 4 种组合，它们由控制位 CPHA 和 CPOL 决定。SPI 的数据传输模式如图 9.23 和图 9.24 所示。

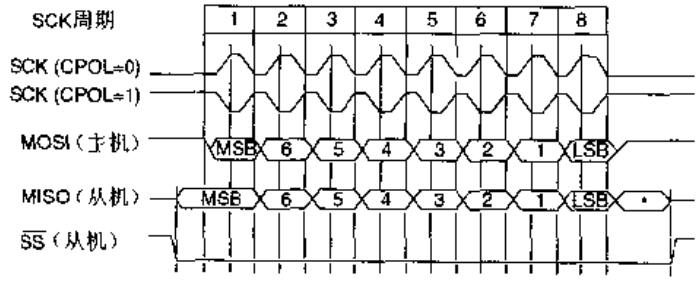


图 9.23 CPHA=1 时的数据传输模式

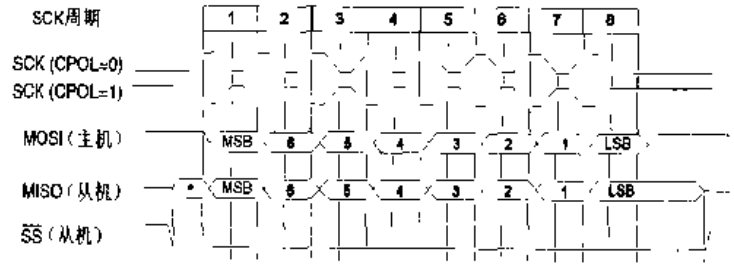


图 9.24 CPHA=0 时的数据传输模式

3. 与 SPI 有关的寄存器

- SPI 控制寄存器(SPCR)

SPI 的控制寄存器 SPCR 用于控制 SPI 接口的工作方式，其格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SPIE(SPI 中断使能)

该位置“1”，且全局中断使能时，SPI 状态寄存器中的 SPIF 置位将产生 SPI 中断。

SPE(SPI 使能)

该位置“1”时，SPI 功能开启。

DORD(数据发送顺序)

该位置“1”，数据的最低位首先被发送。

该位清“0”，数据的最高位首先被发送。

MSTR(主从选择)

该位置“1”，SPI 为主机模式。

该位清“0”，SPI 为从机模式。

注意： 如果/SS 设置为输入方式，且在 MSTR=1 时，它的输入为低电平，则 MSTR

将复位。

CPOL(时钟极性)

该位置“1”，时钟信号在无数据传输时为高电平。

该位清“0”，时钟信号在无数据传输时为低电平。

CPHA(时钟相位)

该位置“1”，时钟信号在数据位的中部为下降沿(CPOL=0 时)。

该位清“0”，时钟信号在数据位的中部为上升沿(CPOL=0 时)。

SPR1、SPR0(SPI 时钟选择)

由 SPR1 和 SPR0 确定的 SPI 时钟如表 9.21 所示。

表 9.21　SPI 的时钟选择

SPR1	SPR0	SCK
0	0	fcl/4
0	1	fcl/16
1	0	fcl/64
1	1	fcl/128

- SPI 状态寄存器(SPSR)

SPI 状态寄存器用于标识当前 SPI 接口的运行情况，其格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
SPIF	WCOL	–	–	–	–	–	–

SPIF(SPI 中断标志)

当串行传输完成时 SPIF 被置位，若此时 SPI 中断使能位 SPIE 为“1”，且全局中断使能，则 SPI 向单片机提出中断请求。

WCOL(写冲突标志)

在数据传输过程中，如果向 SPI 的数据寄存器写入数据则 WCOL 被置“1”。

D5～D0(保留位)

- SPI 数据寄存器(SPDR)

SPI 数据寄存器用于存储 SPI 要发送和接收的数据，其格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

SPI 的数据寄存器是可读/写的，它实际上包括两个数据寄存器：一个用于 SPI 的数据发送(只可写入)，一个用于 SPI 的数据接收(只可读出)。

4. 同步串行口应用举例

本例将同步串行口作为移位串行口使用，其电路原理如图 9.25 所示。AT90LS8535 作为 SPI 主机，其数据输出端口(MOSI)接 74HC164 的移位输入端口(A、B)，时钟输出端口(SCK)

接 74HC164 的移位时钟端(CLK)。

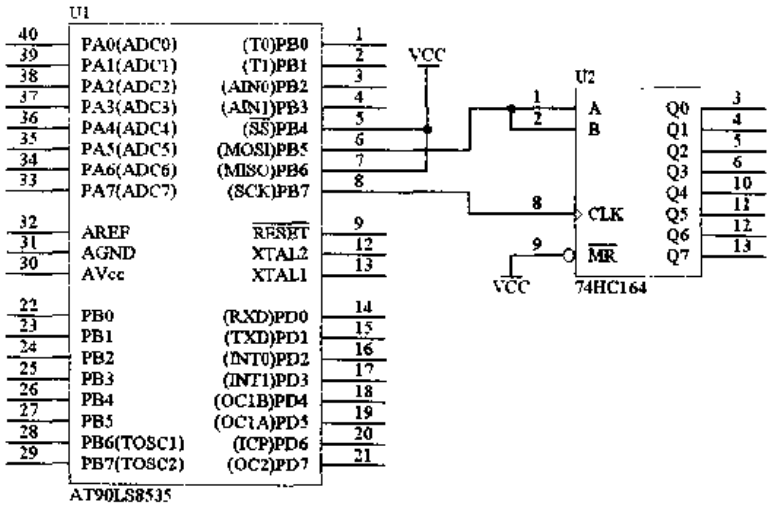


图 9.25 SPI 口的应用电路

源程序为：

```
#include "io8535.h"
#define uchar unsigned char
#define uchar unsigned int

//延时子程序
void delay()
{
    uint i;
    for(i=0;i<0xffff;i++)
    {   ;
}
}

//主程序
void main()
{
    uchar j;
    SPCR=0xf7;
    SPSR=0;
    for(j=0;j<255;j++)
    {
        delay();
        SPDR=j;
    }
}
```

9.3.3 AT90LS8535 的异步串行接口

AT90LS8535 单片机有一个全双工的异步串行接口，这个串行口既可用于单片机与外

设的低速通信，也可用于单片机与单片机、单片机与 PC 的通信。

AT90LS8535 的串行接口实际上可以分为数据发送单元和数据接收单元两个部分。其中，数据发送单元接收来自单片机的并行数据，然后在内部时钟和移位逻辑的控制下，将该数据经 TXD 端串行发出；而数据接收单元则从 RXD 端接收串行数据，并通过内部的数据恢复逻辑和移位寄存器将接收到的串行数据保存到接收缓冲器中。

1. 异步串行口的内部结构

AT90LS8535 的数据发送和数据接收分别由数据发送单元和数据接收单元来完成。

- 数据发送单元

  图 9.26 为数据发送单元的内部结构。

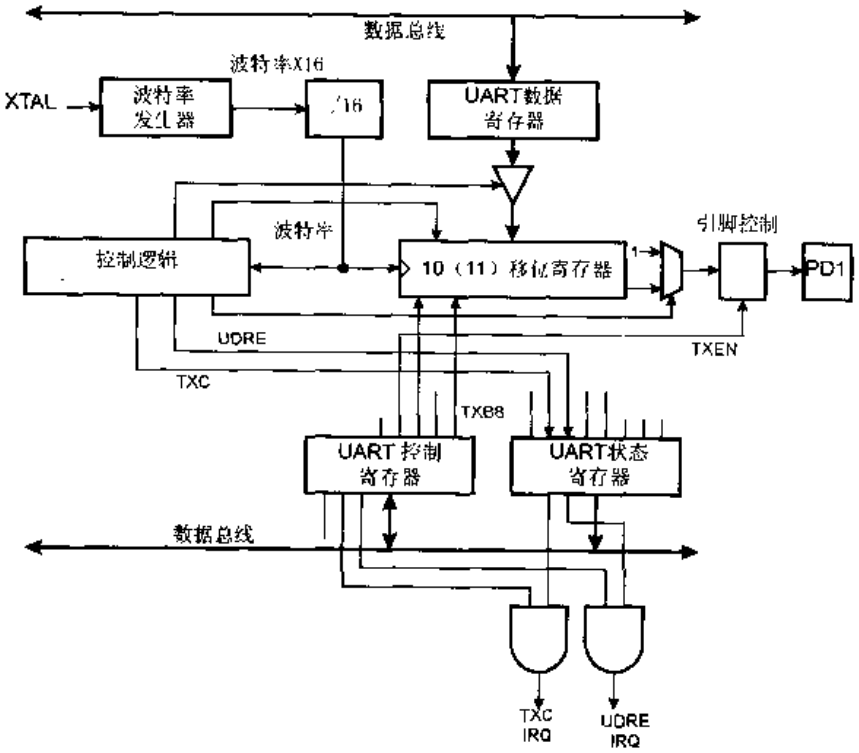


图 9.26　AT90LS8535 单片机的异步发送单元

如上图，AT90LS8535 单片机首先通过将数据写入串口数据寄存器(UDR)启动发送过程，然后在控制逻辑的控制作用下将移位时钟脉冲加载到移位寄存器，随后依次将起始位、数据位、奇偶校验位和停止位输出至 TXD 端口。在一帧数据传输完成后，若又有数据写入串口数据寄存器(UDR)，则该数据立即被装入移位寄存器中，且 UDRE 位置“1”。

当一个字节数据发送完成后，串口状态寄存器(USR)的发送完成状态位被置“1”，此时，若发送完成中断位和全局中断使能位都为“1”，则单片机产生一个中断请求。

注意： 只有当串口控制寄存器(UCR)的发送使能位(TXEN)为“1”时，PD1 才被用于发送串口数据，如果 TXEN 被设置为“0”，则它就只能用作通用 I/O 口。

- 数据接收单元

图 9.27 为数据接收单元的内部结构。

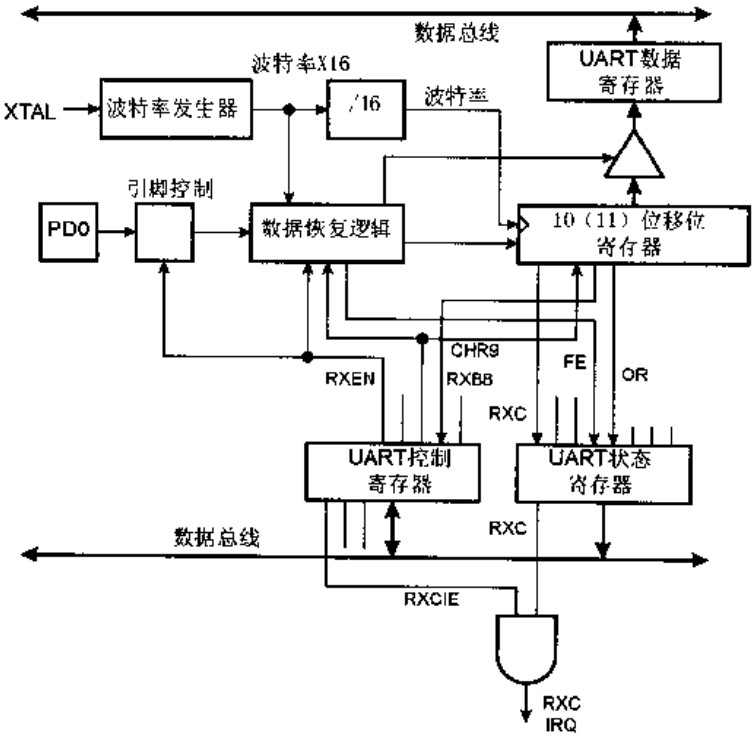


图 9.27 AT90LS8535 单片机的异步接收单元

如图所示，AT90LS8535 的接收逻辑以 16 倍的波特率采样 RXD 引脚。当线路空闲时，1 个逻辑“1”至逻辑“0”的下降沿被接收逻辑认为是串行数据的起始位，随后采样逻辑将 RXD 上的串行数据流送至串行移位寄存器，在完成数据位的采样后，接收逻辑将检测到的一个逻辑“0”到逻辑“1”的跳变认定为停止位，并结束串口接收过程。

当一个字节数据接收完成后，串口状态寄存器(USR)的接收完成状态位被置“1”，此时，若接收完成中断位和全局中断使能位都为“1”，则单片机产生一个中断请求。

2. 与 UART 有关的寄存器

- UART 数据寄存器(UDR)

UART 的数据寄存器用于存储 UART 要发送和接收的数据，其格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

UART 的数据寄存器是可读/写的，它实际上包括两个物理上分离的数据寄存器：一个用于 UART 的数据发送(只可写入)，一个用于 UART 的数据接收(只可读出)，因此它们可以使用同一个 I/O 地址空间。

- UART 状态寄存器(USR)

UART 状态寄存器用于标识当前 UART 接口的运行情况，其格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
RXC	TXC	UDRE	FE	OR	–	–	–

RXC(UART 接收完成)

当接收到的字节从移位寄存器转移到 UART 数据寄存器(UDR)时，RXC 位被置“1”，如果此时 UART 控制寄存器(UCR)中的 UART 接收完成中断使能位(RXCIE)和全局中断使能位都为“1”，则单片机产生一个中断请求。

RXC 在单片机读 UART 数据寄存器(UDR)时被复位。

TXC(UART 发送完成)

当 UART 的移位寄存器中的所有数据(包括停止位)都被移出，且没有新的数据被写入 UART 数据寄存器(UDR)时，TXC 位被置“1”，此时，如果 UART 控制寄存器(UCR)中的 UART 发送完成中断使能位(TXCIE)和全局中断使能位都为“1”，则单片机产生一个中断请求。

当单片机调用相应的中断服务程序时，TXC 位将被置为“0”。

UDRE(UART 数据寄存器空)

当 UART 数据寄存器(UDR)被送至到移位寄存器中时，UDRE 被位被置“1”，若此时 UART 控制寄存器(UCR)中的 UART 数据寄存器空中断使能位(UDRIE)位和全局中断使能位都为“1”，则单片机产生一个中断请求。

UDRE 可通过写入新的数据至 UART 数据寄存器(UDR)来清除。

FE(帧错误)

当串口采样逻辑对停止位的 3 个采样点中，有两个或三个为逻辑“0”时，帧错误位(FE)被置“1”。

当接收到数据的停止位为 1 时，FE 位被清除。

OR(溢出错误)

如果一个串行数据帧的最后一位被接收后，UART 数据寄存器(UDR)还没有被取走，则溢出错误位 OR 被置“1”

当接收到的数据被送至 UART 数据寄存器(UDR)时，OR 位被清除。

D2～D0(保留位)

- UART 控制寄存器(UCR)

UCR 寄存器用于控制 UART 接口的工作方式，其格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8

RXCIE(UART 接收完成中断使能)

该位置“1”，且全局中断使能时，UART 状态寄存器中的 RXC 置位将产生 UART 接收中断。

TXCIE(UART 发送完成中断使能)

该位置“1”，且全局中断使能时，UART 状态寄存器中的 TXC 置位将产生 UART 发送中断。

UDRIE(UART 数据寄存器空中断使能)

该位置“1”，且全局中断使能时，UART 状态寄存器中的 UDRE 置位将产生 UART 数据寄存器空中断。

RXEN(接收使能)

该位置“1”时，UART 接收允许。

TXEN(发送使能)

该位置“1”时，UART 发送允许。

CHR9(9 位字符)

该位置“1”时，UART 发送和接收的数据都是 9 位。其中待发送的第 9 位通过 UART 控制寄存器(UCR)中的 TXB8 设置，接收到的第 9 位数据为 RXB8。

RXB8(接收数据第 8 位)

该位用于存放 UART 接收到的第 9 位数据。

TXB8(发送数据第 8 位)

该位为 UART 待发送的第 9 位数据。

- 波特率寄存器(UBRR)

波特率寄存器用于设置串口发送和接收的速率，其格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

波特率寄存器确定的串口工作波特率如表 9.22 所示。

表 9.22 串口工作波特率

波特率	1 MHz	误差%	1.843MHz	误差%	2MHz	误差%
2400	UBRR=25	0.2	UBRR=47	0.0	UBRR=51	0.2
4800	UBRR=12	0.2	UBRR=23	0.0	UBRR=25	0.2
9600	UBRR=6	7.5	UBRR=11	0.0	UBRR=12	0.2
14400	UBRR=3	7.8	UBRR=7	0.0	UBRR=8	3.7
19200	UBRR=2	7.8	UBRR=5	0.0	UBRR=6	7.5
28800	UBRR=1	7.8	UBRR=3	0.0	UBRR=3	7.8
38400	UBRR=1	22.9	UBRR=2	0.0	UBRR=2	7.8
57600	UBRR=0	7.8	UBRR=1	0.0	UBRR=1	7.8
76800	UBRR=0	22.9	UBRR=1	33.3	UBRR=1	22.9
115200	UBRR=0	84.3	UBRR=0	0.0	UBRR=0	7.8
波特率	2.457 MHz	误差%	3.276 Hz	误差%	3.686 MHz	误差%
2400	UBRR=63	0.0	UBRR=84	0.4	UBRR=95	0.0
4800	UBRR=31	0.0	UBRR=42	0.8	UBRR=47	0.0
9600	UBRR=15	0.0	UBRR=20	1.6	UBRR=23	0.0

续表

波特率	2.457 MHz	误差%	1.843MHz	误差%	3.686 MHz	误差%
14400	UBRR=10	3.1	UBRR=13	1.6	UBRR=15	0.0
19200	UBRR=7	0.0	UBRR=10	3.1	UBRR=11	0.0
28800	UBRR=4	6.3	UBRR=6	1.6	UBRR=7	0.0
38400	UBRR=3	0.0	UBRR=4	6.3	UBRR=5	0.0
57600	UBRR=2	12.5	UBRR=3	12.5	UBRR=3	0.0
76800	UBRR=1	0.0	UBRR=2	12.5	UBRR=2	0.0
115200	UBRR=0	25.0	UBRR=1	12.5	UBRR=1	0.0
波特率	4 MHz	误差%	4.608 MHz	误差%	7.372 MHz	误差%
2400	UBRR=103	0.2	UBRR=119	0.0	UBRR=191	0.0
4800	UBRR=51	0.2	UBRR=59	0.0	UBRR=95	0.0
9600	UBRR=25	0.2	UBRR=29	0.0	UBRR=47	0.0
14400	UBRR=16	2.1	UBRR=19	0.0	UBRR=31	0.0
19200	UBRR=12	0.2	UBRR=14	0.0	UBRR=23	0.0
28800	UBRR=8	3.7	UBRR=9	0.0	UBRR=15	0.0
38400	UBRR=6	7.5	UBRR=7	6.7	UBRR=11	0.0
57600	UBRR=3	7.8	UBRR=4	0.0	UBRR=7	0.0
76800	UBRR=2	7.8	UBRR=3	6.7	UBRR=5	0.0
115200	UBRR=1	7.8	UBRR=2	20.0	UBRR=3	0.0
波特率	8 MHz	误差%	9.216 MHz	误差%	11.059 MHz	误差%
2400	UBRR=207	0.2	UBRR=239	0.0	UBRR=287	-
4800	UBRR=103	0.2	UBRR=119	0.0	UBRR=143	0.0
9600	UBRR=51	0.2	UBRR=59	0.0	UBRR=71	0.0
14400	UBRR=34	0.8	UBRR=39	0.0	UBRR=47	0.0
19200	UBRR=25	0.2	UBRR=29	0.0	UBRR=35	0.0
28800	UBRR=16	2.1	UBRR=19	0.0	UBRR=23	0.0
38400	UBRR=12	0.2	UBRR=14	0.0	UBRR=17	0.0
57600	UBRR=8	3.7	UBRR=9	0.0	UBRR=11	0.0
76800	UBRR=6	7.5	UBRR=7	6.7	UBRR=8	0.0
115200	UBRR=3	7.8	UBRR=4	0.0	UBRR=5	0.0

3. 异步串行口应用举例

AT90LS8535 单片机的异步串行口主要用于单片机与外设或单片机与 PC 间的数据交换。它可以只作为发送器(接收器)使用，也可以同时作为发送器和接收器使用。

例：循环发送数据 0x55。

单片机通过异步串行口循环发送字符“a”，当单片机的 UART 口和外部设备相连时，

外设可以接收到该字符。此程序可用于检验单片机的串行口是否发送正常。源程序如下：

```
#include "io8535.h"
#include "macros.h"
#include "stdio.h"
#define uchar unsigned char
//串口发送数据
void USART_Transmit(uchar x)
{
    //等待数据寄存器空
    while(!(USR&0x20))
        ;
    UDR=x;
}
//主程序
void main()
{
    UBRRH=0;    //设置波特率 9600
    UBRRL=38;
    UCR=0x08;   //发送使能
    while(1)
    {
        USART_Transmit(0x55);
    }
}
```

例：接收串口数据。

AT90LS8535 单片机的异步串行口也可只工作在接收模式。源程序如下：

```
#include "io8535.h"
#include "macros.h"
#include "stdio.h"
#define uchar unsigned char
//主程序
void main()
{
    uchar rdata
    UBRRH=0;    //设置波特率 9600
    UBRRL=38;
    UCR=0x10;   //接收使能
    while(1)
    {
        if(USR&0x80)
            rdata=getchar();
    }
}
```

例：单片机先接收数据，若接收到的数据为 0xff，则发送数据 0xff。

本例中，单片机的异步串行口既可以接收数据，也可以发送数据。因此，在需要数据交互的应用场合中，多采用这种模式。源程序如下：

```
#include "io8535.h"
#include "macros.h"
#include "stdio.h"
#define uchar unsigned char
//串口发送数据
void USART_Transmit(uchar x)
{
    //等待数据寄存器空
    while(!(USR&0x20))
        ;
    UDR=x;
}
//主程序
void main()
{
    uchar rdata
    UBRRH=0;    //设置波特率 9600
    UBRRL=38;
    UCR=0x18;   //接收、发送使能
    while(1)
    {
        if(USR&0x80)
            rdata=getchar();    //接收数据
        if(rdata==0xff)         //判断接收到的数据是否为 0xff
            USART_Transmit(0xff);
    }
}
```

9.4　定时/计数器

AT90LS8535 单片机的内部具有 3 个定时/计数器。其中，T/C0 和 T/C1 为 8 位的定时/计数器，T/C2 为 16 位的定时/计数器。这 3 个定时/计数器除了具有通用的定时/计数功能外，有些还具备输入捕获、比较匹配和脉宽调制输出等功能。

9.4.1　定时/计数器的分频器

AT90LS8535 单片机的定时/计数器可以选择不同频率的计数源，这些计数源由分频器对主时钟的不同分频构成。

定时/计数器 0 和定时/计数器 1 共用一个分频器，可选的时钟频率为 0、CK/8、CK/64、CK/256、CK/1024 或外部时钟。图 9.28 为定时/计数器 0 和定时/计数器 1 的分频器结构。

定时/计数器 2 拥有自己独立的分频器，如图 9.29 所示。当异步状态寄存器(ASSR)的 AS2 位被清 0 时，分频器和单片机的主时钟相连；当该位置 1 时，则分频器同外部异步时钟源相连。定时/计数器 2 可选的时钟频率为 0、CK/8、CK/32、CK/64、CK/128、CK/256、CK/1024 或外部晶振频率。

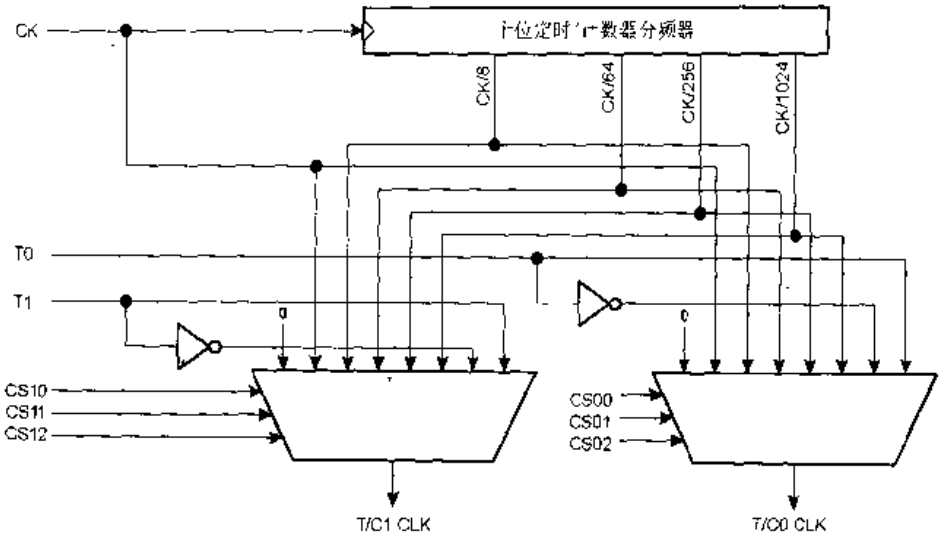


图 9.28 定时/计数器 0 和定时/计数器 1 的分频器结构

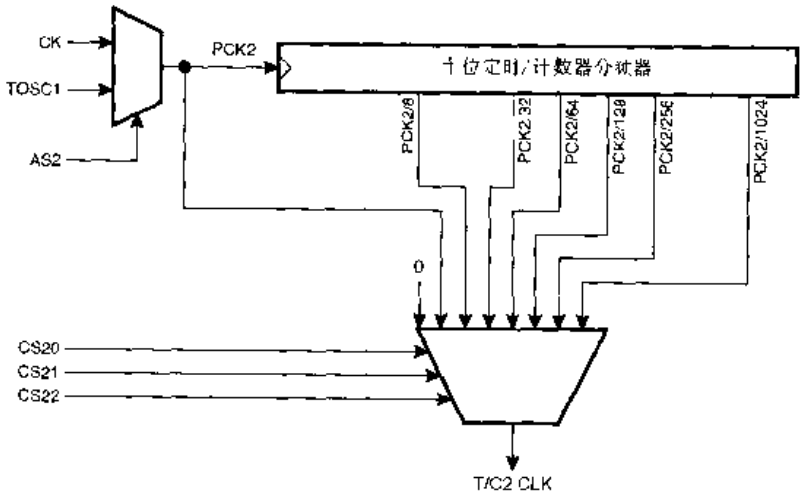


图 9.29 定时/计数器 2 的分频器结构

9.4.2 8 位定时/计数器 0

1. 定时/计数器 0 的结构

定时/计数器 0 的内部结构如图 9.30 所示。

如图，定时/计数器 0 可以采用 CK、CK 的分频或外部时钟作为计数源。此外，它还可以在定时/计数器 0 的控制寄存器(TCCR0)中设置为停用方式。溢出状态标志位在定时/计数器的中断标志寄存器(TIFR) 中，中断使能控制位在定时/计数器的中断屏蔽寄存器(TIMSK)中。

注意： 当定时/计数器 0 对外部脉冲计数时，为了确保外部时钟与 CPU 的时钟频率的同步，必须使两个外部时钟转换之间的最少时间间隔为一个 CPU 的时钟周期。

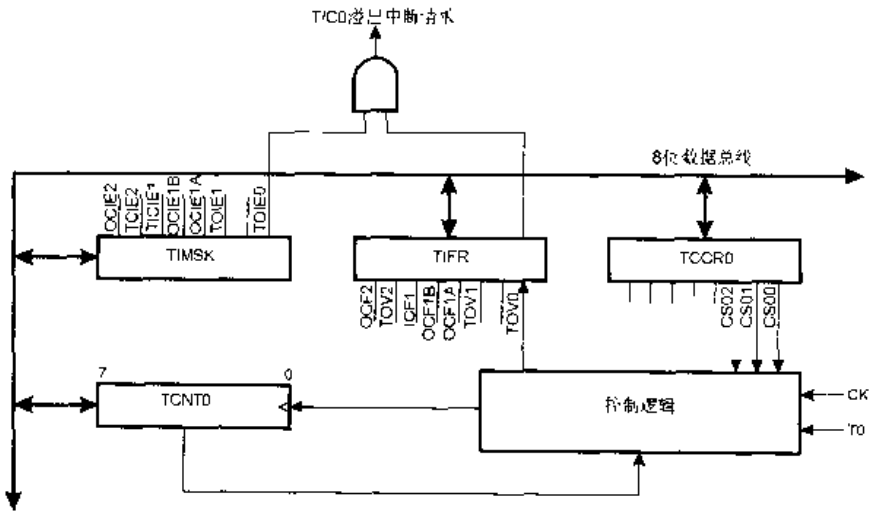


图 9.30　定时计数器 0 的内部结构

2. 与定时/计数器 0 有关的寄存器

- 定时/计数器 0 控制寄存器(TCCR0)

定时/计数器 0 控制寄存器用于控制定时/计数器 0 的工作参数，其格式为：

D7	D6	D5	D4	D3	D2	D1	D0
–	–	–	–	–	CS02	CS01	CS00

D7～D3(保留位)

CS02、CS01、CS00(定时/计数器 0 的时钟源选择位)

CS02、CS01、CS00 用于选择定时/计数器 0 的计数时钟分频比例，如表 9.23 所示。

表 9.23　定时/计数器 0 的分频比例

CS02	CS01	CS00	说　明
0	0	0	停止计数
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	外部下降沿信号
1	1	1	外部上升沿信号

- 定时/计数器 0 计数寄存器(TCNT0)

定时/计数器 0 计数寄存器用于存储当前的计数值，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

该计数寄存器是一个可读/写的加 1 计数器。当计数寄存器的计数值溢出时，定时/计数器中断标志寄存器(TIFR)中的 TOV0 置为“1”，若此时定时/计数器中断屏蔽寄存器(TIMSK)中的 TOIE0 和全局中断使能为都为“1”，则单片机产生一个定时/计数器 0 的溢出中断。

3. 定时/计数器 0 的应用举例

定时/计数器 0 对外部脉冲计数的电路如图 9.31 所示。

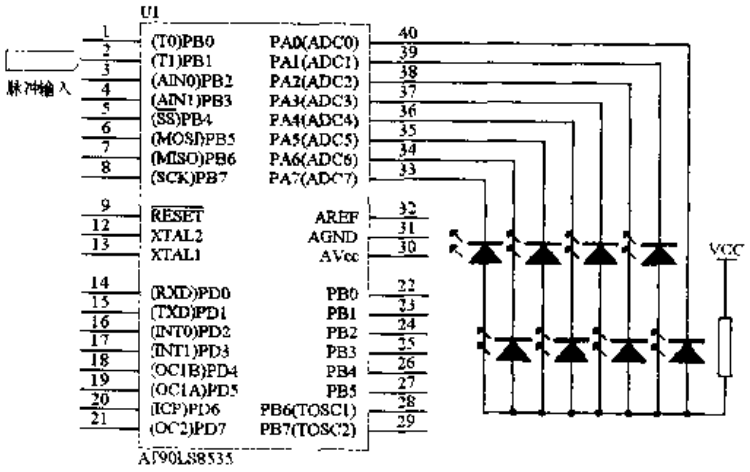


图 9.31 定时/计数器 0 的应用电路

外部脉冲信号通过 PB0(T0)输入，定时/计数器 0 对脉冲个数进行计数，并通过 PA 口的发光二极管输出。

源程序为：

```
#inlucde "io8535.h"
#define uchar unsigned char
//主程序
void main()
{
    uchar mid;
    DDRA=0xff;
    PORTA=0xff;
    TCCR0=0x07;        //设置定时/计数器 0 的工作方式
    TCNT0=0;           //置定时/计数器 0 的初值
    do                 //循环输出计数值
    {
        mid=TCNT0;
        PORTA=~mid;
    }while(1)
}
```

9.4.3 16 位定时/计数器 1

1. 定时/计数器 1 的结构

定时/计数器 1 的内部结构如图 9.32 所示。

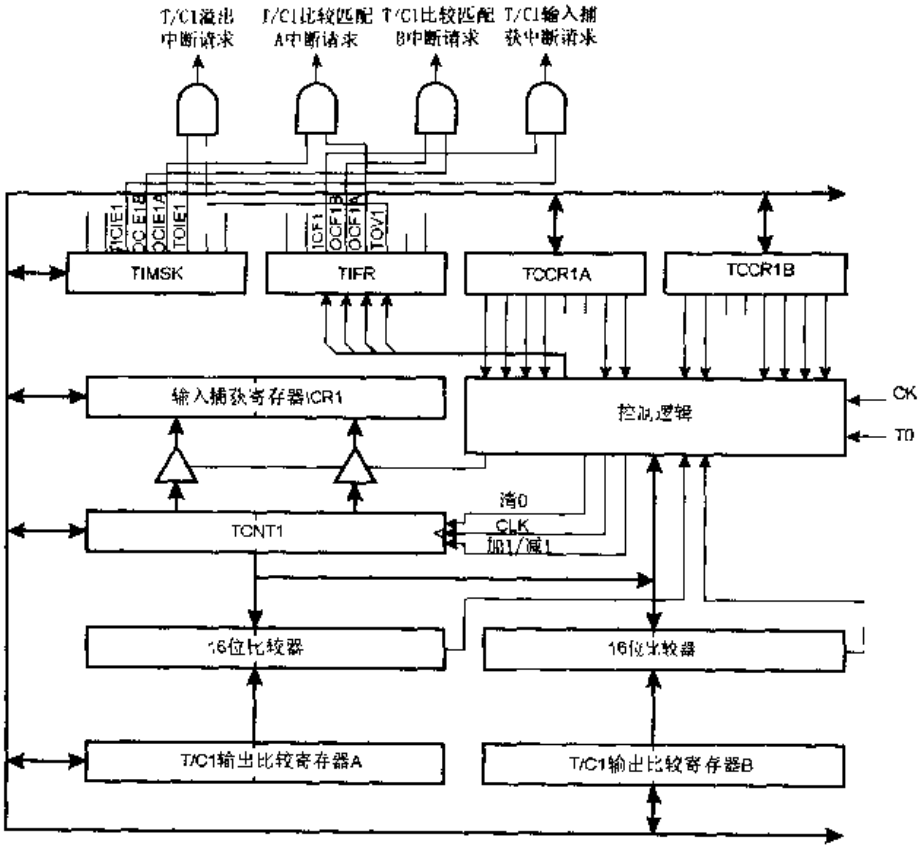


图 9.32　定时计数器 1 的内部结构

16 位定时/计数器 1 采用的计数频率可以是 CK、CK 的分频或外部时钟。同定时/计数器 0 一样，它也可以通过设置定时/计数器 1 的控制寄存器(TCCR1A 和 TCCR1B)来停止定时/计数功能。

除了通用的定时/计数功能外，定时/计数器 1 还提供输出比较、输入捕获和脉宽调制输出功能。

- 输出比较

  定时/计数器 1 提供两次输出比较，输出比较寄存器(OCR1A 和 OCR1B)分别用于存放这两次比较的数据，当定时/计数器 1 的计数值和这两个寄存器中的某个值匹配时，相应的输出比较中断标志位置“1”，如果此时输出比较的中断使能位和全局中断使能位都为“1”，则单片机产生一个输出比较中断。

- 输入捕获

  当单片机的输入捕获引脚(ICP)上产生一个输入捕获信号时，定时/计数器 1 当前的值被传送到输入捕获寄存器 ICR1 中，同时，输入捕获标志位 ICF1 置“1”。如果此时输入捕获的中断使能位和全局中断使能位都为“1”，则单片机产生一个输入捕获中断。

- 脉宽调制输出

  定时/计数器 1 可用作 8 位、9 位或 10 位的脉冲调制器。这种脉宽调制输出经过滤波后，可得到模拟电压信号。

注意：定时/计数器 1 可用作 8 位、9 位或 10 位的脉冲调制器。这种脉宽调制输出经过滤波后，可得到模拟电压信号。

当定时/计数器 1 采用对外部计数时钟时，为了确保外部时钟与 CPU 的时钟频率的同步，必须使两个外部时钟转换之间的最少时间间隔为一个 CPU 的时钟周期。

2. 与定时/计数器 1 有关的寄存器

- 定时/计数器 1 的控制寄存器 A(TCCR1A)

定时/计数器 1 的控制寄存器 A 用于控制定时/计数器 1 的输出比较和脉冲输出参数，其格式为：

D7	D6	D5	D4	D3	D2	D1	D0
COM1A1	COM1A0	COM1B1	COM1B0	–	–	PWM11	PWM10

COM1A1、COM1A0(输出比较 1A)

COM1A1 和 COM1A0 定义定时/计数器 1 的输出比较引脚 OC1A 的电平。由于输出比较是 I/O 口的一个可选功能，因此引脚 OC1A 的方向寄存器必须设置为输出模式。

COM1B1、COM1B0(输出比较 1B)

COM1B1 和 COM1B0 定义定时/计数器 1 的输出比较引脚 OC1B 的电平。由于输出比较是 I/O 口的一个可选功能，因此引脚 OC1B 的方向寄存器必须设置为输出模式。

COM1A1 和 COM1A0(COM1B1 和 COM1B0)的输出比较控制如表 9.24 所示。

表 9.24 输出比较方式

COM1A1(COM1B1)	COM1A0(COM1B0)	说 明
0	0	定时/计数器 1 与 OC1A(OC1B)不相连
0	1	触发 OC1A(OC1B)
1	0	置 OC1A(OC1B)为“0”
1	1	置 OC1A(OC1B)为“1”

D3、D2(保留位)

PWM11、PWM10(脉宽调制选择)

这两位用于选择定时/计数器 1 的脉宽调制方式，如表 9.25 所示。

表 9.25 脉宽调制方式

PWM11	PWM10	说 明
0	0	定时/计数器 1 的 PWM 功能禁止
0	1	定时/计数器 1 为 8 位的 PWM
1	0	定时/计数器 1 为 9 位的 PWM
1	1	定时/计数器 1 为 10 位的 PWM

- 定时/计数器 1 的控制寄存器 B(TCCR1B)

定时/计数器 1 的控制寄存器 B 用于设置比较输入比较器参数并选择定时/计数器 1 的工作时钟，其格式为：

D7	D6	D5	D4	D3	D2	D1	D0
ICNC1	ICES1	–	–	CTC1	CS12	CS11	CS10

ICNC1(输入捕获噪声消除)

该位置“1”时，输入捕获噪声消除功能使能。此时，单片机将连续 4 次采样输入捕获引脚(ICP)，若 4 次采样值都相同，则输入捕获信号被认为有效。

该位置“0”时，输入捕获噪声消除功能禁止。此时，输入捕获功能在 ICP 引脚上的第一个上升(下降沿)被触发。

ICES1(输入捕获 1 边沿选择)

该位置“1”时，ICP 引脚上的一个上升沿触发输入捕获。

该位置“0”时，ICP 引脚上的一个下降沿触发输入捕获。

D5、D4(保留位)

CTC1(清除比较匹配定时/计数器 1)

若该位为“1”，则在发生比较匹配事件后，定时/计数器 1 被清“0”。

CS12、CS11、CS10(定时/计数器 1 的时钟分频选择)

这 3 位用于选择定时/计数器的计数分频，如表 9.26 所示。

表 9.26　定时/计数器 1 的计数分频

CS12	CS11	CS10	说　明
0	0	0	定时/计数器 1 停止
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	外部下降沿信号
1	1	1	外部上升沿信号

- 定时/计数器 1 计数寄存器(TCNT1H 和 TCNT1L)

定时/计数器 1 的计数寄存器用于存储当前的计数值，其格式为：

TCNT1H	D7	D6	D5	D4	D3	D2	D1	D0
	BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8
TCNT1L	D7	D6	D5	D4	D3	D2	D1	D0
	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

该计数寄存器是一个可读/写的 16 位加 1 计数器。为了确保单片机能对该寄存器

的高、低位同时进行读写操作，定时计数器设置了一个 TEMP 寄存器。当数据写入时，单片机先将高位字节写入 TCNT1H 中(实际上，写入的数据被暂时存放在 TEMP 中)，然后单片机再把低位字节写入 TCNT1L 中(此时，TEMP 中的数据与当前的写入数据组合，并同时写入两个寄存器中)；当数据读出时，单片机先读低位寄存器 TCNT1L(高位寄存器 TCNT1H 的内容同时传送至 TEMP)，然后单片机再读取高位寄存器 TCNT1H(此时，单片机读取的是 TEMP 中的内容)。

当该计数寄存器的计数值溢出时，定时/计数器中断标志寄存器(TIFR)中的 TOV1 置为“1”，若此时定时/计数器中断屏蔽寄存器(TIMSK)中的 TOIE1 和全局中断使能为都为“1”，则单片机产生一个定时/计数器 1 的溢出中断。

- 定时/计数器 1 输出比较寄存器 A、B(OCR1AH 和 OCR1AL、OCR1BH 和 OCR1BL)

  定时/计数器 1 的输出比较寄存器 A 和 B 存储定时/计数器 1 相比较的两个 16 位数据，由于 OCR1A 和 OCR1B 也都是 16 位的寄存器，因此单片机对它们的读写操作也需要像操作定时计数器 1 的计数寄存器 TCNT1 一样遵循同样的读写顺序。

  其格式为：

OCR1AH(OCR1BH)	D7	D6	D5	D4	D3	D2	D1	D0
	BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8
OCR1AL(OCR1BL)	D7	D6	D5	D4	D3	D2	D1	D0
	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

- 定时/计数器的输入捕获寄存器(ICR1H 和 ICR1L)

  当输入捕获事件发生时，定时/计数器的输入捕获寄存器存储定时/计数器 1 的计数值，其格式为：

ICR1H	D7	D6	D5	D4	D3	D2	D1	D0
	BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8
ICR1L	D7	D6	D5	D4	D3	D2	D1	D0
	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

ICR1 的读写顺序同 TCNT1、OCR1A 和 OCR1B 相同。

3. 定时/计数器 1 的工作方式

定时/计数器 1 的工作方式包括定时/计数方式、输出比较方式、输入捕获方式和脉宽调制器(PWM)方式，由于前 3 种方式较为简单，且在说明定时/计数 1 的有关寄存器时，也做了详细的介绍，因此，这里只对定时/计数器 1 的 PWM 方式做一些叙述。

定时/计数器 1 可设置为一个双 8 位、9 位或 10 位的脉宽调制器(PWM)，此时，定时/计数器 1 从 0000H 加 1 计数到顶，然后再减 1 计数到 0000H，如此反复循环，并且在 OC1A 和 OC1B 上输出相应信号。

定时/计数器 1 在不同 PWM 分辨率下的计数最大值和频率如表 9.27 所示。

表 9.27　定时/计数器 1 在不同 PWM 分辨率下的计数最大值和频率

PWM 分辨率	定时/计数器的最大计数值	频率
8	255	fTCK1/510
9	511	fTCK1/1022
10	1023	fTCK1/2046

定时/计数器 1 的 PWM 输出控制方式如表 9.28 所示。

表 9.28　定时/计数器 1 的 PWM 输出方式

COM1A1(COM1B1)	COM1A0(COM1B0)	OC1A(OC1B)上的作用
0	0	断开
0	1	断开
1	0	加 1 计数匹配时，清除；减 1 计数时，置位。
1	1	加 1 计数匹配时，置位；减 1 计数时，清除。

定时/计数器 1 的 PWM 输出如图 9.33 所示。

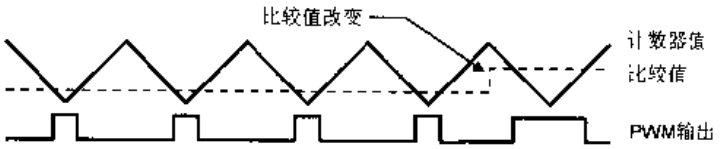


图 9.33　定时/计数器 1 的 PWM 输出

4. 定时/计数器 1 的应用举例

定时/计数器 1 除了通用的定时和计数功能之外，还可实现输出比较、输入捕获和 PWM 功能。

- 定时/计数器 1 的输出比较

  电路如图 9.34 所示，单片机的定时/计数器 1 对外部脉冲信号进行计数。计数值每满 1024，则单片机的 PA0 口反相一次，从而使电动机反向运转。

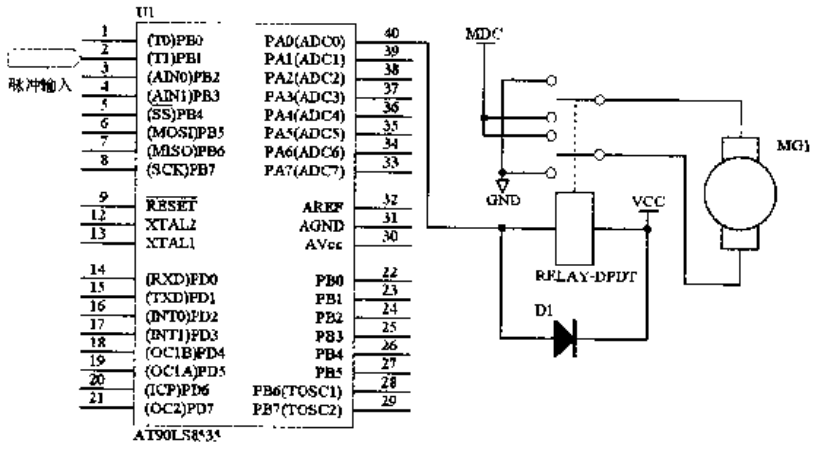


图 9.34　定时/计数器 1 的输出比较应用电路

源程序为：

```
#inlucde "io8535.h"
#pragma interrupt_handler TIME:7

//定时/计数器1中断服务子程序
void TIME()
{
    PORTA^0x01;
}
//主程序
void main()
{
    DDRA=0xff;
    PORTA=0xff;
    TCCR1A=0;          //设置定时/计数器1的工作方式
    TCCR1B=0x0f;
    TCNT1H=0;          //置定时/计数器1的初值
    TCNT1L=0;
    OCR1AH=0x03;
    OCR1AL=0xff;
    TIMSK=0x10;        //开启定时/计数器1的比较匹配中断
    SREG=0x80;         //开中断
    do
    {

        }while(1);
}
```

- 定时/计数器1的输入捕获

应用电路如图9.35所示，外部脉冲信号同定时/计数器1的输入捕获端口ICP相接，液晶显示电路同单片机的PA口相接。

程序采用定时/计数器1的输入捕获中断功能测量每两次脉冲信号之间的计数值，并在液晶屏上显示出来。

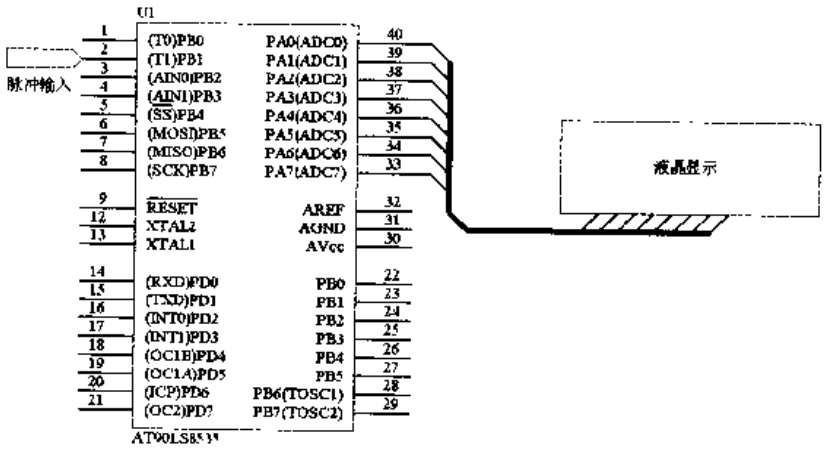


图9.35 定时/计数器1的输入捕获应用电路

源程序为：

```
#inlucde "io8535.h"
#define uchar unsigned char
#define uint unsigned int
#pragma interrupt_handler CAPTURE:6
uchar a,b;
uint mid1=0,mid2=0,mid3=0;
//液晶初始化子程序
void lcd_display()
{
    ……
}
//液晶显示子程序
void lcd_time()
{
    ……
}
//定时/计数器 1 输入捕获中断
void CAPTURE()
{
    a=ICR1L;
    b=ICR1H;
    mid2=mid1;
    mid1=a;
    mid1=mid1<<8;
    mid1=mid1+b;
    if(mid1<=mid2)
        mid3=0xffff-mid2+mid1;
    else
        mid3=mid1-mid2;
    lcd_display(mid3);
}
//主程序
void main()
{
    lcd_init();        //初始化
    TCCR1A=0;          //设置工作方式
    TCCR1B=0xc5;
    TCNT1H=0x00;
    TCNT1L=0x00;
    TIMSK=0x20;        //开启定时/计数器 1 的输入捕获中断
    SREG=0x80;         //开中断
    do
    {
        ;
        }while(1);
}
```

- 定时/计数器 1 的 PWM 输出

定时/计数器 1 的 PWM 输出可以用来控制电机，也可以经滤波电路构成一个简单

的数模转换器。以下为定时/计数器 1 作为 PWM 使用举例。

源程序为：

```
#inlucde "io8535.h"
//主程序
void main()
{
    DDRA|=BIT(PD5);        //设置 OC1A 为输出
    TCCR1A=0x83;           //定义定时/计数器的工作方式
    TCCR1B=0x01;
    OCR1AH=0xff;           //定义 PWM 输出的占空比
    OCR1AL=0x01;
}
```

9.4.4 8 位定时/计数器 2

1. 定时/计数器 2 的结构

定时/计数器 2 的内部结构如图 9.36 所示。

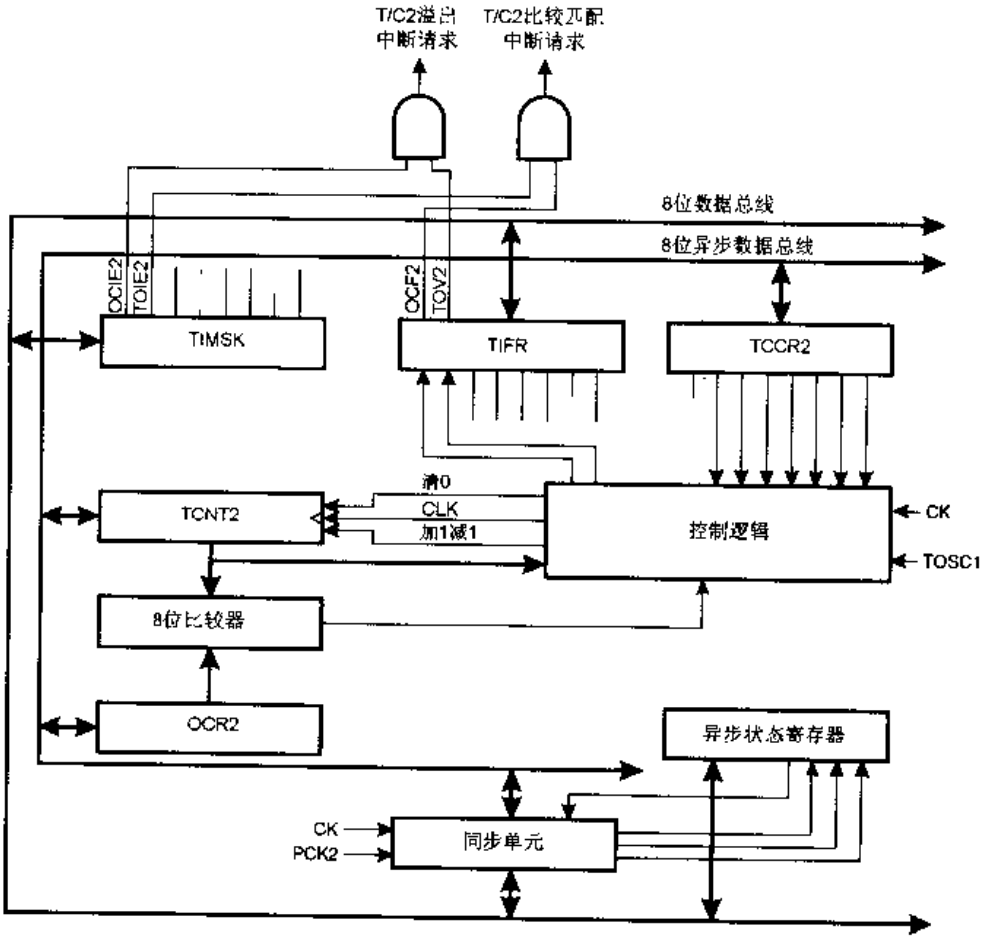


图 9.36 定时/计数器的内部结构

如图，定时/计数器 2 的计数时钟可以是 CK、CK 的分频或异步外部时钟。当异步状态

寄存器(ASSR)的 AS2 位为“1”时，定时/计数器 2 的分频器输入为系统主时钟；当 AS2 位为“0”时，分频器的输入为引脚 PC6 和 PC7 上所加的一个外部异步时钟。

定时/计数器 2 除了定时和计数外，还可实现输出比较和脉宽调制功能。

2. 与定时/计数器 2 有关的寄存器

- 定时/计数器 2 的控制寄存器(TCCR2)

定时/计数器 2 的控制寄存器可设置定时/计数器 0 的工作方式，并选择计数源的分频比例，其格式为：

D7	D6	D5	D4	D3	D2	D1	D0
–	PWM2	COM21	COM20	CTC2	CS22	CS21	CS20

D7(保留位)

PWM2(PWM 使能)

该位置“1”，定时/计数器 2 的 PWM 功能使能。

该位置“0”，定时/计数器 2 的 PWM 功能禁止。

COM21、COM20(输出比较模式)

COM21 和 COM20 定义定时/计数器 2 的输出比较模式，它影响引脚 OC2 的电平。由于输出比较是 I/O 口的一个可选功能，因此引脚 OC2 的方向寄存器必须设置为输出模式。输出比较模式的选择如表 9.29 所示。

表 9.29　定时/计数器 2 的输出比较模式

COM21	COM20	说　明
0	0	定时/计数器和 OC2 引脚断开
0	1	触发 OC2
1	0	置 OC2 为“0”
1	1	置 OC2 为“1”

CTC2(清除比较匹配定时/计数器 2)

若该位为“1”，则在发生比较匹配事件后，定时/计数器 2 被清“0”。

CS22、CS21、CS20(定时/计数器 2 的时钟分频选择)

这 3 位用于选择定时/计数器的计数分频，如表 9.30 所示。

表 9.30　定时/计数器 2 的计数分频

CS22	CS21	CS20	说　明
0	0	0	定时/计数器 2 停止
0	0	1	PCK2
0	1	0	PCK2/
0	1	1	PCK2/
1	0	0	PCK2/
1	0	1	PCK2/128

续表

CS22	CS21	CS20	说　明
1	1	0	PCK2/256
1	1	1	PCK2/1024

- 定时/计数器 2 计数寄存器(TCNT2)

定时/计数器 2 计数寄存器用于存储当前的计数值，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

TCNT2 是一个可读/写的计数器。如果该寄存器被写入，且定时/计数器 2 选择了一个计数时钟源，则寄存器的值不断加 1/减 1(在 PWM 模式时)。

- 定时/计数器 2 的输出比较寄存器(OCR2)

定时/计数器 2 的输出比较寄存器存储与定时/计数器 2 相比较的两个 16 位数据，其格式为：

D7	D6	D5	D4	D3	D2	D1	D0
BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0

- 异步状态寄存器(ASSR)

异步状态寄存器用于标识定时/计数器 2 在异步计数方式下的工作状态，其格式如下：

D7～D4(保留位)

AS2(异步定时/计数器 2)

D7	D6	D5	D4	D3	D2	D1	D0
–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB

AS2 为“1”时，定时/计数器 2 由 TOSC1 引脚获得时钟。

AS2 为“0”时，定时/计数器 2 的计数时钟由系统主时钟获得。

TCNUB(定时/计数器 2 更新忙)

当定时/计数器 2 工作于异步模式时，若 TCNUB 为“1”，则系统对计数寄存器值的更新操作正在进行，用户不可以向 TCNT2 写入数据；若 TCNUB 为“0”，则用户可以对 TCNT2 执行写入操作。

OCR2UB(输出比较寄存器更新忙)

当定时/计数器 2 工作于异步模式时，若 OCR2UB 为“1”，则系统对输出比较寄存器 2 的更新操作正在进行，用户不可以向该寄存器写入数据；若 OCR2UB 为“0”，则用户可以对输出比较寄存器 2 执行写入操作。

TCR2UB(定时/计数器 2 的控制寄存器更新忙)

当定时/计数器 2 工作于异步模式时，若 TCR2UB 为“1”，则系统对输出比较寄存器 2 的更新操作正在进行，用户不可以向该寄存器写入数据；若 OCR2UB 为“0”，

则用户可以对输出比较寄存器 2 执行写入操作。

3. 定时/计数器 2 的工作方式

定时/计数器 2 具有通用定时/计数、输出比较和脉宽调制 3 种工作方式，它们的设置和操作同定时/计数器 1 类似，在此不再介绍。

4. 定时/计数器 2 的应用举例

AT90LS8535 单片机的定时/计数器 2 可以对外部异步时钟计数。外部的时钟源通过 TOSC1(PC6)和 TOSC2(PC7)的外接晶振获得。当外接晶振的频率为 32.768 KHz，且计数源的时钟采用 128 分频时，计数器每计满 256 个数就可以产生一个溢出中断，时间间隔刚好为 1 秒，因此，利用定时/计数器 2 的这个特性，就可以把它用作实时时钟。

图 9.37 为定时/计数器 2 用作实时时钟的电路原理。其中，单片机的 TOSC1 和 TOSC2 引脚接 32.768 KHz 的晶振，PA0 口则外接液晶显示模块以显示当前的时钟。

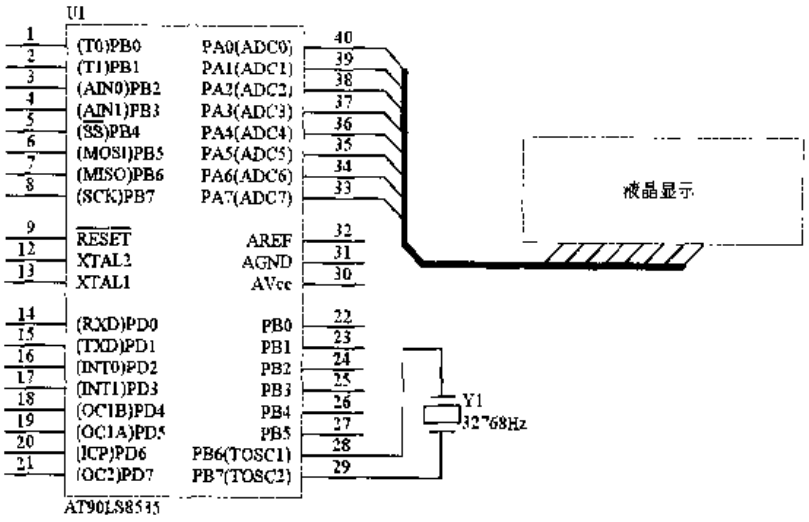


图 9.37　定时/计数器 2 的应用电路

源程序为：

```
#inlucde "io8535.h"
#define uchar unsigned char
#pragma interrupt_handler TIME:5

//设置外部变量
uchar second=0,minute=0,hour=0;

//液晶初始化子程序
void lcd_init()
{
    ......
}

//液晶时钟显示
void lcd_time()
```

```
{
.....
}

//定时/计数器2中断服务子程序
void TIME()
{
    second=second+1;
    if(second==60)
    {
        second=0;
        minute=minute+1;
        if(minute==60)
        {
            minute=0;
            hour=hour+1;
            if(hour==24)
                hour=0;
        }
    }
    lcd_time();
}

//主程序
void main()
{
    lcd_init();
    ASSR=0x08;        //启动异步计数功能
    TCCR2=0x05  //计数器采用128分频
    TIMSK=0x40;       //开启定时/计数器2溢出中断
    TCNT2=0x00;       //定时/计数器2置初值0
    SREG=0x80;        //开中断
    do
    {   ;
        }while(1);
}
```

9.5 EEPROM

AT90LS8535单片机的片内含有512个字节的EEPROM，用户可以通过操作其控制寄存器实现对EEPROM中各个存储单元的访问。

9.5.1 与EEPROM有关的寄存器

1. EEPROM地址寄存器(EEARH和EEARL)

EEPROM地址寄存器用于指定某个EEPROM单元的地址，其格式为：

EEARH	D7	D6	D5	D4	D3	D2	D1	D0
	–	–		–	–	–	–	EEAR8
EEARL	D7	D6	D5	D4	D3	D2	D1	D0
	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0

2. EEPROM 数据寄存器(EEDR)

对于 EEPROM 的写入操作，EEDR 存储的是待写入 EEPROM 的 8 位数据；对于 EEPROM 的读取操作，EEDR 存储的是从 EEPROM 的指定地址中读取的 8 位数据。

3. EEPROM 的控制寄存器(EECR)

EEPROM 的控制寄存器能对 EEPROM 的读写操作进行控制，其格式为：

D7	D6	D5	D4	D3	D2	D1	D0
–	–	–	–	EERIE	EEMWE	EEWE	EERE

D7～D4(保留位)

EERIE(EEPROM 准备好中断使能)

当 EERIE 和全局中断使能位都为“1”时，若 EEWE 置“0”，则单片机产生一个相应的中断。

EEMWE(EEPROM 主写使能)

当 EEMWE 为“1”时，设置 EEWE 为“1”将把 EEDR 中的数据写入 EEAR 所选择的地址空间中。如果 EEMWE 为“0”，则 EEWE 无效。

EEMWE 在被用户置“1”后的 4 个时钟周期后，被硬件清“0”。

EEWE(EEPROM 写使能)

当 EEPROM 的地址和数据准备好以后，用户必须设置 EEWE 为“1”，才能将数据写入 EEPROM 中。在置 EEWE 为“1”之前，EEMWE 必须置“1”，否则写入操作无效。EEPROM 数据的写入操作如下：

- 等待 EEWE 为“0”
- 把 EEPROM 的地址写入 EEAR
- 把 EEPROM 的数据写入 EEDR
- 置 EEMWE 为“1”
- 在置 EEMWE 为“1”的 4 个时钟周期内，向 EEWE 中写入“1”

EERE(EEPROM 读使能)

EERE 用于对 EEPROM 的数据读取，当 EEAR 中设置了 EEPROM 的读取地址后，EERE 的置“1”操作，将使该单元中的数据送至 EEDR 寄存器中，此时 EERE 位被硬件自动清“0”。

9.5.2　EEPROM 读/写操作

1. ICCAVR 的 C 编译器对 EEPROM 的编程

AT90LS8535 单片机的 EEPROM 可以使用库函数访问。在调用这些库函数之前，用户

必须把头文件 eeprom.h 加入到源文件中。

用户可以使用的库函数包括：

- unsigned char EEPROMread(int addr)
- void EEPROMwrite(int addr,unsigned char data)
- void EEPROMReadBytes(int addr, void *ptr, int size)
- void EEPROMWriteBytes(int addr, void *ptr, int size)

其中 addr 为 EEPROM 的读/写地址；data 为待写入的数据字节；*ptr 为指向数据缓冲区的指针；size 为待写入的数据个数。

2. 注意事项

当单片机的供电电压较低，或者 CPU 的指令被错误执行时，EEPROM 的数据读/写将会发生错误。为了确保 EEPROM 中数据正确性，以下两点必须注意：

- 使单片机的复位端在电源电压较低时保持高电平。
- 使 AVR 的 CPU 在电源电压较低时工作在睡眠模式。

9.5.3 EEPROM 的应用举例

利用 ICCAVR C 的库函数，实现对 EEPROM 的访问。

源程序为：

```
#include "io8535.h"
#include "eeprom.h"
#define uchar unsigned char
#define uint unsigned int
//主程序
void main()
{   uchar wdata=0x55;
    uchar rdata;
    uint wdatas=0x5555;
    uint rdatas;
    EEPROM_WRITE(0x00,wdata);
    EEPROM_READ(0x00,rdata);
    EEPROM_WRITE(0x10,wdatas);
    EEPROM_READ(0x10,rdatas);
}
```

9.6 模拟量输入接口

由于单片机只能处理数字信号，因此外部模拟信号必须经过转换，变成数字信号之后才能输入到单片机中。模数转换器就是一种将模拟信号转换成数字信号的器件。AT90LS8535 单片机的片内包含一个模数转换器。本节将首先介绍一下这个模数转换器的基本原理，然后再对接口编程进行详细的说明。

9.6.1　模数转换器的结构

AT90LS8535 单片机的内部具有一个 8 通道的 10 位 AD 转换器，其内部结构如图 9.38 所示。

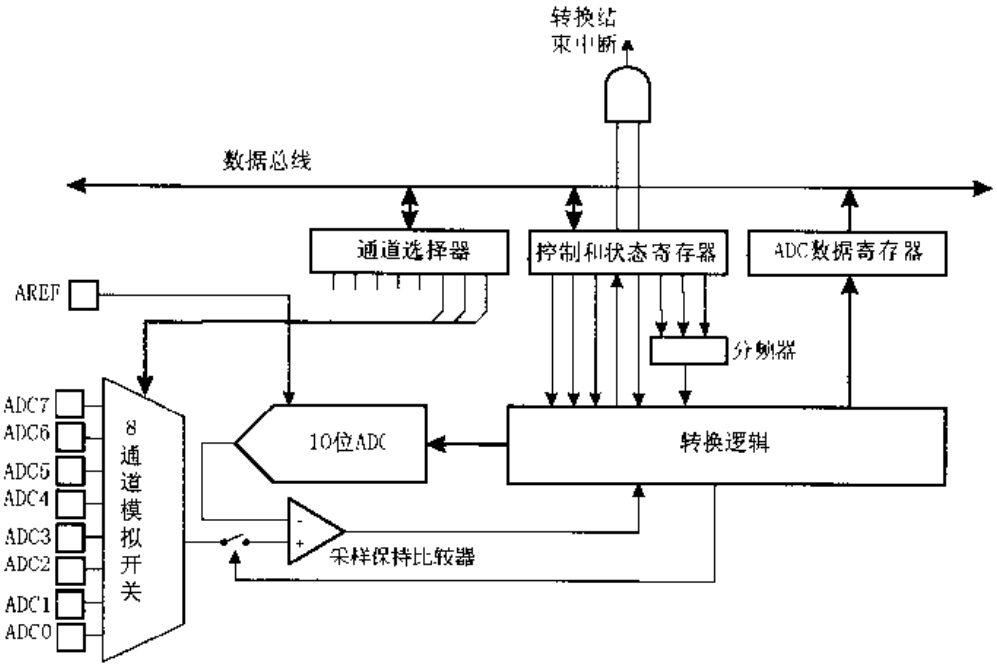


图 9.38　AT90LS8535 单片机的模数转换器结构

如图，AT90LS8535 单片机的模数转换单元包括一个 8 通道的模拟开关、一个采样保持比较器、一个转换逻辑和 3 个控制/状态寄存器。其中，A 口同 8 通道模拟开关的输入相连，用于输入模拟信号，模拟开关的输出则接至采样保持比较器的输入上；采样保持比较器可以确保模数转换逻辑的输入在转换过程中保持不变，它的输出接至模数转换逻辑。

9.6.2　ADC 的使用

ADC 可以将输入的模拟电压信号转换成一个 10 位的数字量信号。输入模拟电压的范围介于 AGND 和 AVcc 之间，输入模拟通道通过 ADMUX 寄存器选择。

AT90LS8535 单片机的 ADC 模块由 ADSCR 寄存器中的 ADEN 位使能。当 ADEN 为“1”时，ADC 功能有效，并且输入通道同模拟电压的输入引脚相连。此时，若 ADSC 置“1”，则 ADC 启动一个模数转换过程，这个模数转换过程用于初始化 ADC(转换结果无效)。

当 ADC 模块被启动以后，用户可以通过 ADFR 位选择 ADC 的两种转换模式，即单次转换模式和自由转换模式。若 ADFR 为“0”时，则 ADC 工作在单次转换模式，此时，每个转换过程都需要置位 ADSC；若 ADFR 为“1”时，则 ADC 工作在自由转换模式，此时，ADC 连续采样模拟输入端并将转换得到的数据输出至 ADC 的数据寄存器。单次转换模式

的时序如图 9.39 所示，自由转换模式的时序如图 9.40 所示。

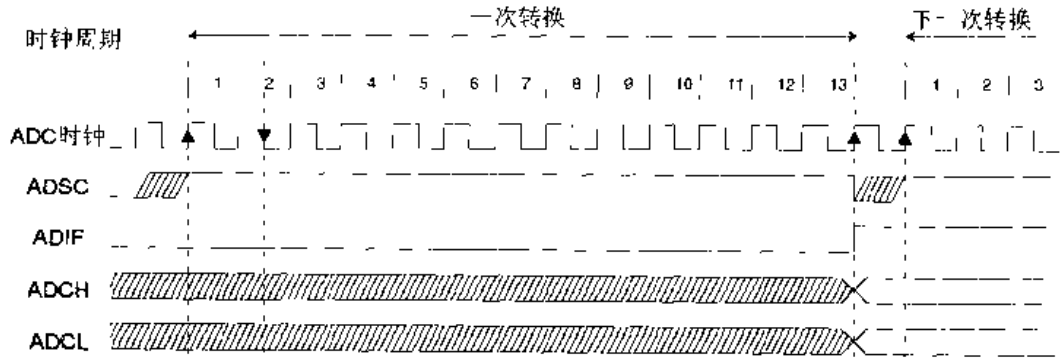


图 9.39 单次转换模式的时序逻辑

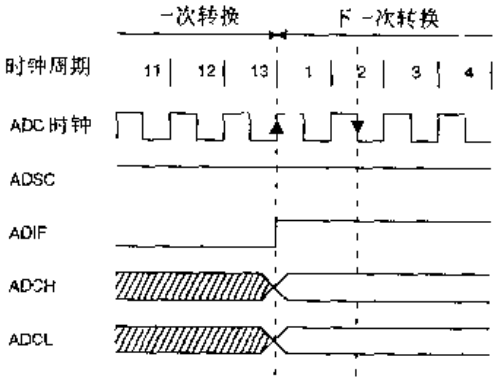


图 9.40 自由转换模式的时序逻辑

ADC 的转换结果存储在 ADCH 和 ADCL 两个寄存器中。为了确保数据读取的正确性，ADCL 寄存器的内容应当首先被读取，一旦用户开始对 ADCL 读取，ADC 对数据寄存器的写操作就被禁止，这就意味着，如果用户读取了 ADCL，那么即便另一次 ADC 转换过程在读 ADCH 之前结束了，两个数据寄存器中的内容也不会被更新。当用户对 ADCH 的读操作完成后，ADC 才可以更新 ADCH 和 ADCL。

当一次转换过程结束后，ADIF 位被置“1”，此时，若 ADIE 和全局中断使能位都为“1”，则单片机产生一个 ADC 中断。

9.6.3 与模数转换器有关的寄存器

- ADC 通道选择寄存器(ADMUX)

  ADC 通道选择寄存器用于选择需要进行模数转换的模拟通道，其格式为：

  D7～D3(保留位)

  MUX2、MUX1、MUX0(模拟通道选择)

D7	D6	D5	D4	D3	D2	D1	D0
–	–	–	–	–	MUX2	MUX1	MUX0

这 3 位的值选择接入模数转换逻辑的模拟通道(ADC7～ADC0)。

- ADC 控制和状态寄存器(ADCSR)

ADC 控制和状态寄存器可以实现 ADC 的控制和状态显示，其格式为：

D7	D6	D5	D4	D3	D2	D1	D0
ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADEN(模数转换使能)

ADEN 为“1”时，单片机的模数转换使能；否则禁止。

ADSC(模数转换启动)

当 ADC 工作于单次转换模式时，该位必须写入“1”才能启动每次转换过程；当 ADC 工作与自由转换模式时，ADSC 也必须在第一次转换时写入“1”。

注意： ADC 在上电后，必须首先进行一次初始化转换，这个转换值无效。

ADFR(ADC 自由运转模式)

ADFR 置“1”时，ADC 工作于自由运转模式。在这种模式下，ADC 不断地采样、更新数据。该位置“0”时，ADC 的自由运转模式被终止。

ADIF(ADC 中断标志)

ADC 的转换完成，并且数据更新后，ADC 中断标志(ADIF)置“1”。此时，若 ADC 的中断使能位(ADIE)和全局中断使能位都为“1”，则单片机产生一个 ADC 完成中断。

当单片机执行相应的中断后，ADIF 被清“0”。ADIF 也可通过写入“0”来清除。

ADIE(ADC 中断使能)

ADIE 为“1”，则 ADC 中断使能；否则，中断禁止。

ADPS2、ADPS1、ADPS0(ADC 分频选择)

这 3 位决定 ADC 分频器的值，如表 9.31 所示。

表 9.31　ADC 的分频值

ADPS2	ADPS1	ADPS0	分频因数
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

- ADC 数据寄存器(ADCL 和 ADCH)

ADC 数据寄存器存储模数转换后的数值，其格式为：

ADCH	D7	D6	D5	D4	D3	D2	D1	D0
	–	–	–	–	–	–	ADC9	ADC8
ADCL	D7	D6	D5	D4	D3	D2	D1	D0
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0

当 ADCL 被读取以后，ADC 数据寄存器的值一直保持不变直到 ADCH 也被读取。因此，单片机每次必须读取 ADC 的两个数据寄存器，并且先读低位，后读高位。

9.6.4 ADC 的噪声消除

AT90LS8535 单片机内部的数字电路所产生的电磁干扰会影响模拟量测量的精度。如果用户需要较高的转换精度，则可以通过以下几种方式减小噪声的影响。

- 系统的模拟部分采用单独的模拟地，这个模拟地和数字地单点连接。
- 使模拟路线的路径尽可能短，并远离数字地。
- 模数转换器的 Avcc 通过一个 LC 网络连接到数字 Vcc 上。
- 使用 ADC 的噪声消除技术减少来自 CPU 的噪声。

9.6.5 ADC 的应用举例

AT90LS8535 单片机的模数转换电路如图 9.41 所示。外部待测模拟电压信号经 PA0(ADC0)端口同单片机内部的模数转换单元相连；AVcc 和 AREF 接至电源电压；AGND 接电源地；模数转换后的结果通过 PB 口上的 LED 输出。由于模数转换的结果是 10 位的数据，而 PB 口是 8 位的，因此程序中还需要进行数据转换，将数据的高 8 位输出。

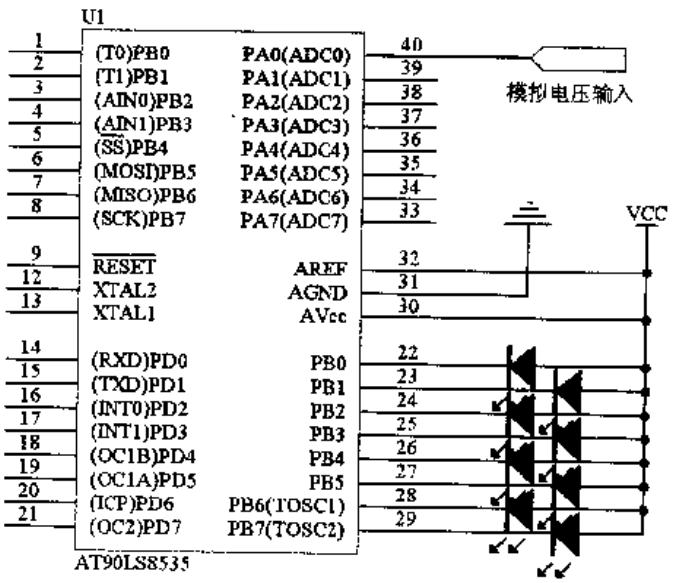


图 9.41 AT90LS8535 单片机的模数转换电路

源程序为：

```
#include "io8535.h"
#define uchar unsigned char

#pragma interrupt_handler ad_handler:15
uchar addata;

//模数转换完成中断
void ad_handler(void)
{
    addata=ADC>>2;
    PORTB=addata;
    ADCSR|=BIT(ADSC);    //启动下一次转换
}

//主程序
void main()
{
    PORTA=0;     //设置 PA 口为输入，无上拉
    DDRA=0;
    PORTB=0xff;
DDRB=0xff;
    ADMUX=0x00;      //选择第 0 通道
    ADCSR=0x86;      //采用单次转换模式，64 分频
    ADCSR|=BIT(ADSC);    //启动一次哑转换
    SREG=0x80;           //开中断
    ADCSR|=BIT(ADSC);    //启动一次转换
    while(1)                 //等待中断
    {
        ;
    }
}
```

9.7　模拟比较器

AT90LS8535 单片机的片内模拟比较器用于对 PB2(AIN0)和 PB3(AIN1)的模拟电压进行比较，当 PB2 上的电压高于 PB3 上的电压时，模拟比较器的输出 ACO 就被置“1”，这个信号可以用于触发一个单独的模拟比较器中断，也可以用于触发一个定时/计数器的输入捕获。

9.7.1　模拟比较器的结构

模拟比较器的内部结构如图 9.42 所示，模拟比较器实际上就是一个带输出单元的运算放大器，外部引脚通过接到运算放大器的两个输入端实现模拟比较功能(PB2 接同相端，PB3 接反相端)。

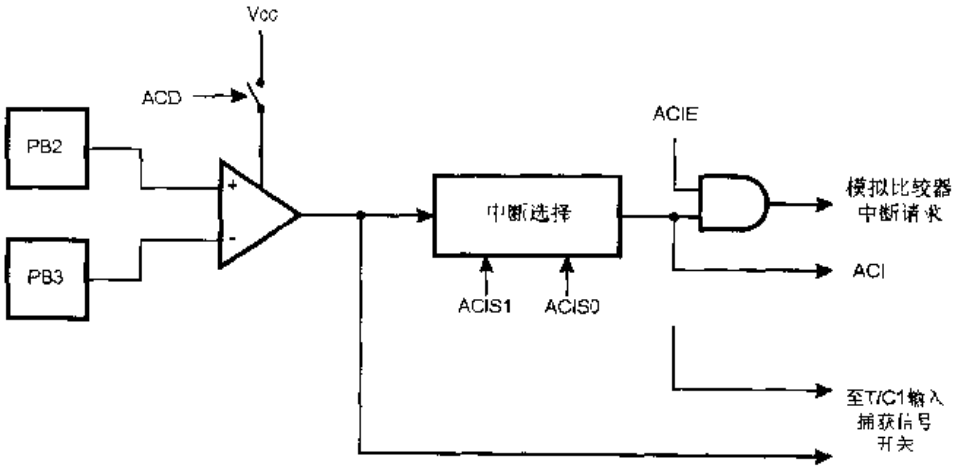


图 9.42 模拟比较器的内部结构

9.7.2 与模拟比较器有关的寄存器

- 模拟比较器控制和状态寄存器(ACSR)

D7	D6	D5	D4	D3	D2	D1	D0
ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

ACD(模拟比较器禁止)

ACD 为 1 时，模拟比较器的电源被切断。用户可以在任何时候开启或关闭模拟比较器的电源，但是在对 ACD 操作时，模拟比较器的中断使能位必须先置为“0”；否则，ACD 的改变将会产生一个中断。

D6(保留位)

ACO(模拟比较器输出)

ACO 直接同运算放大器的输出相连。

ACI(模拟比较器中断标志)

当模拟比较器的输出发生如 ACIS1 和 ACIS0 所定义的变化时，ACI 被置“1”。此时，若模拟比较器中断使能位(ACIE)和全局中断使能位均为“1”，则单片机产生一个模拟比较器中断。

ACIE(模拟比较器中断使能)

ACIE 为“1”时，模拟比较器中断使能。

ACIE 为“0”时，模拟比较器中断禁止。

ACIC(模拟比较器输入捕获使能)

该位置“1”时，模拟比较器的输出将触发定时/计数器 1 的输入捕获功能。当该位置“0”时，模拟比较器的输出和定时/计数器的输入捕获引脚分离。

ACIS1、ACIS0(模拟比较器中断模式选择)

这两位用于设置模拟比较器的中断触发方式，如表 9.32 所示。

表 9.32　模拟比较器的中断触发方式

ACIS1	ACIS0	中断模式
0	0	输出变化触发
0	1	保留位
1	0	输出下降沿触发
1	1	输出上升沿触发

注意：　改变 ACIS1 和 ACIS0 时，模拟比较器的中断要禁止，否则可能会引发一个中断。

9.7.3　模拟比较器的应用举例

使用 AT90LS8535 单片机的模拟比较器可以实现模拟电压信号的比较，应用电路如图 9.43 所示。

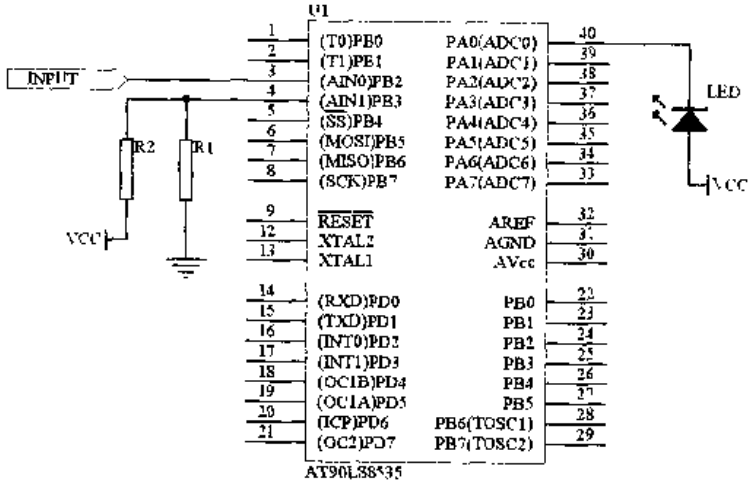


图 9.43　模拟比较器的应用电路

如图，外部模拟电压信号从模拟比较器的 AIN0 端输入，而比较器的 AIN1 端则接至 Vcc 的分压，若取 R1=R2，则 AIN1 上的电压为 0.5Vcc。

当 AIN0 端的电压高于 0.5Vcc，模拟比较器的控制和状态寄存器(ACSR)中的 ACO 置“1”；当 AIN0 端的电压低于 0.5Vcc，ACO 清“0”。由此可见，程序可以通过对状态位 ACO 的查询来判断外部模拟电压信号(AIN0 端的电压)是否大于 0.5Vcc：若大于 0.5Vcc，则 LED 亮；否则 LED 灭。

源程序如下：

```
#include "io8535.h"
#define uchar unsigned char
//主程序
void main()
{
```

```
    uchar mid;
    DDRA=0xff;
    PORTA=0xff;
    ACSR=0x80;  //启动模拟比较器功能
    while(1)
    {
        mid=ACSR&0x20;
        if(mid==0)
            PORTA|=BIT(PA0);     //输出1，灭LED
        else
            PORTA&=~BIT(PA0);    //输出0，亮LED
    }
}
```

第 10 章　AT90LS8535 的人机接口编程

人机交互单元是计算机与用户之间实现信息流通的一个重要渠道。一个完善的计算机系统必须具备恰当的人机交互部分，才能及时反映系统的工作状态，实现用户对系统的监控。这一章将以单片机应用系统中比较常见的键盘接口、LED(LCD)显示接口、语音芯片接口、打印机接口和 IC 卡接口为例，说明各种接口的工作原理和编程特性，从而使读者在以后的开发过程中，能够快速而有效地开发出具有多种接口特性的单片机应用系统。

10.1　键 盘 接 口

键盘是计算机系统中最常用的输入设备，用户可以通过它向计算机输入指令和数据。计算机系统中的键盘按其连接方式的不同，可以分为非矩阵式键盘和矩阵式键盘两类。其中，非矩阵式键盘的结构简单，使用方便，但要占用较多的 I/O 口，因此它适用于按键个数较少的场合；矩阵式键盘的编程较为复杂，但为减少系统 I/O 口的占用，在按键数较多时，一般采用这种键盘接口。

10.1.1　非矩阵式键盘

非矩阵式键盘的每个按键都和单片机的一个 I/O 口相连，这种简单的连接方式使得程序对按键的识别变得非常容易，但当按键个数较多时，它会占用较多的 I/O 口资源，因此它只适合于按键较少的应用场合。

AT90LS8535 与非矩阵式键盘(4 个按键)的连接可以采取两种形式，分别介绍如下：

1. 查询方式

如图 10.1 所示，将每个按键的一端接地，另一端分别接到接到 PA0、PA1、PA2 和 PA3 端口。

键盘处理程序通常采用查询方法来实现按键的识别，这时 CPU 只要一有空闲就调用键盘扫描程序，查询键盘，识别键值，并予以处理。

键盘输入信息的流程如图 10.2 所示。

- 判断是否有键按下

  键盘的一端经电阻接到+5 V 电源上，然后再接入单片机的输入口。由于按键被按下时，相应单片机输入线的电平被拉低。因此，为判断是否有键按下，可先由单片机读入 PA 口的值，若 PA 口的输入寄存器和 0xf0 按位相“或”的值不等于 0xff，则表明有键被按下；否则表明无键按下。
- 确定按下的是哪一个键

单片机将 PA 口的低 4 位译码，就可以得到当前按键的键值。

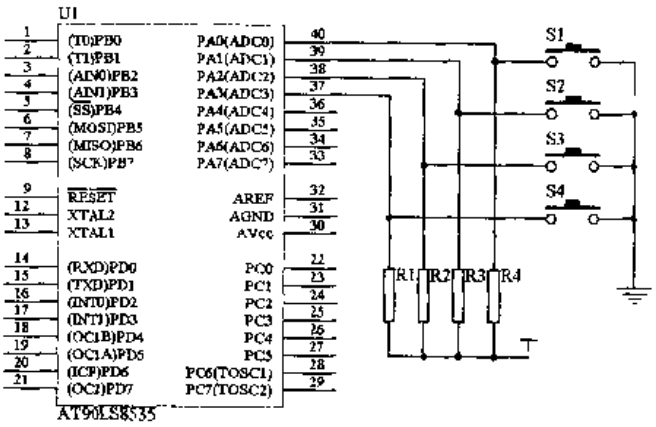


图 10.1 AT90LS8535 与非矩阵式键盘的查询接口

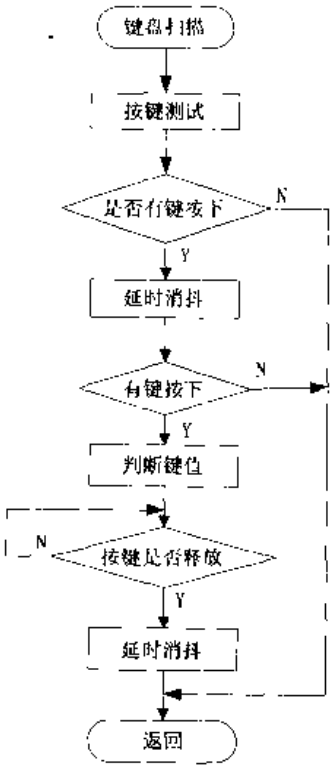


图 10.2 非矩阵式键盘的查询流程

- 等待按键释放

  确定完按键的编码以后，还需要判断按键的释放。按键被释放以后就可以根据键盘扫描程序返回的按键编码调用相应的子程序进行处理。

- 返回键值

  键盘扫描程序将按键的键值返回到主程序，以便主程序执行相应的键盘处理程序。

- 按键的消抖处理

  由于按键是机械触点，因此当用手按动一个键时，在按键的断开和闭合瞬间会出现电压的抖动，如图 10.3 所示。为了保证按键识别的正确，在电压抖动的时候不能进行状态的输入。为此，必须增设消抖处理，使单片机获得按键被按下信息后，并不立即确认按键被按下，而是延时一段时间后再次检测相应端口，如果按键仍为按下状态，则说明按键确实被按下；同样，在单片机检测到按键被释放后，程序同样也延时一段时间，进行后沿的消抖处理，然后再处理键值。

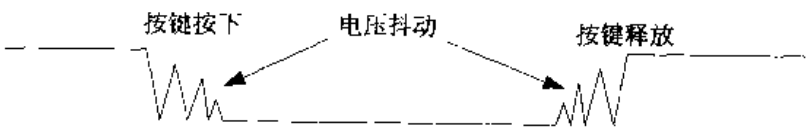


图 10.3 键盘图

该键盘接口的源程序如下：

```
#include "io8535.h"
#define uchar unsigned char
```

```
/*延时子程序*/
void delay(uchar x)
{   uchar j;
    while((x--)!=0)
    {   for(j=0;j<125;j++)
        {;}
    }
}
/*判断是否有键按下,有键按下返回 1，否则返回 0*/
uchar KEYPRESSED(void)
{   uchar key;
    key=PINA;
    key=key|0xf0;
    if(key==0xFF)
        return 0;
    else
        return 1;
}
/*键盘扫描子程序*/
uchar KEYSCAN(void)
{
    uchar key=0;
    delay(10);  /*延时消抖*/
    if((KEYPRESSED())==1)/*如有键按下，查询键值*/
    {   key=PINA;
        key=key|0xf0;
        if(key==0xfe)
            {   key=1;}
        else if(key==0xfd)
            {   key=2;}
        else if(key==0xfb)
            {   key=3;}
        else if(key==0xf8)
            {   key=4;}
        else
            {   key=0;}

        do{
          }while((KEYPRESSED())==1);/*等待键释放*/
        delay(10);                  /*延时消抖*/

     }
    return key; /*返回键值，如为干扰，返回 20*/
}
/*主程序*/
main()
{
    uchar keycode;
    DDRA&=~BIT(PA0);    //设置键盘输入，无上拉
    DDRA&=~BIT(PA1);
```

```
    DDRA&=~BIT(PA2);
    DDRA&=~BIT(PA3);
    do{
        if((KEYPRESSED(void))==1)
        keycode=KEYSCAN(void);
        switch(keycode)
        {
            case 1:
                ......
            case 2:
                ......
        }
    ......
        }while(1);
}
```

2. 中断方式

采用中断方式的键盘连接如图 10.4 所示。将每个按键的一端接地，另一端除了连接到 PA0、PA1、PA2 和 PA3 端口外，还连接到了一个与非门上。当 4 个按键中的任何一个按键按下时，与非门的输出都为高电平，从而使单片机产生中断，这样，单片机就只需要在产生键盘中断的时候才调用键盘处理程序，而不需要不停地查询 PA0～PA3 端口。

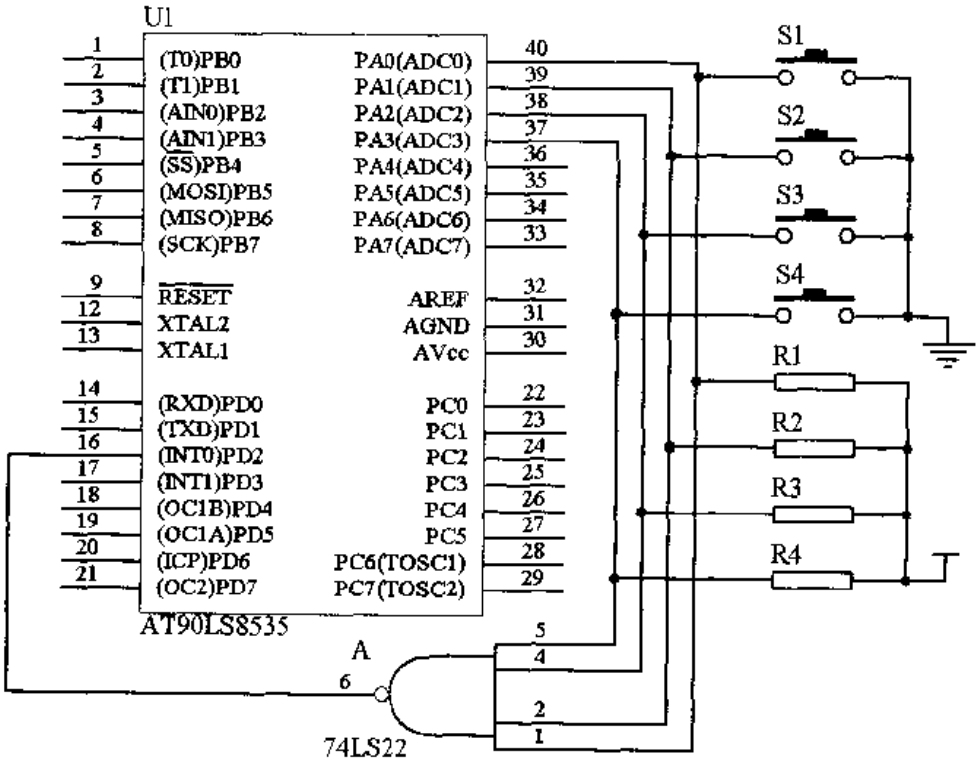


图 10.4 AT90LS8535 与非矩阵式键盘的中断接口

采用中断方式的键盘查询流程如图 10.5 所示，其键盘识别和消抖处理与查询方式相同。该键盘接口的源程序如下：

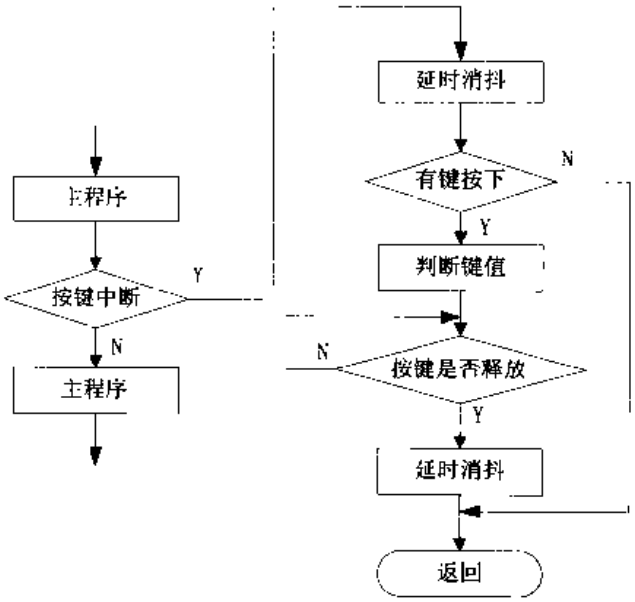


图 10.5　非矩阵式键盘的查询流程

```
#include "io8535.h"
#define uchar unsigned char
#pragma interrupt_handler int0:2
/*延时子程序*/
void delay(uchar x)
{   uchar j;
    while((x--)!=0)
    {   for(j=0;j<125;j++)
        {;}
    }
}
/*判断是否有键按下，有键按下返回 1，否则返回 0*/
uchar KEYPRESSED(void)
{   uchar key;
    key=PINA;
    key=key|0xf0;
    if(key==0xFF)
        return 0;
    else
        return 1;
}

/*键盘扫描子程序*/
uchar KEYSCAN(void)
{
    uchar key=0;
    delay(10);  /*延时消抖*/
    if((KEYPRESSED())==1)/*如有键按下，查询键值*/
    {   key=PINA;
        key=key|0xf0;
        if(key==0xfe)
            {   key=1;}
```

```
        else if(key==0xtd)
            {   key=2;}
        else if(key==0xfb)
            {   key=3;}
        else if(key=-0xf8)
            {   key=4;}
        else
            {   key=0;}

        do{
          }while((KEYPRESSED())==1);/*等待键释放*/
        delay(10);                    /*延时消抖*/

     }
    return key; /*返回键值，如为干扰，返回20*/
}
/*中断服务程序*/
void keyint(void)
{
    uchar keycode;
    if((KEYPRESSED(void))==1)
        keycode=KEYSCAN(void);
    switch(keycode)
        {
            case 1:
                ......
            case 2:
                ......
        }
}
/*主程序*/
main()
{
    DDRA&=~BIT(PA0);    //设置键盘输入，无上拉
    DDRA&=~BIT(PA1);
    DDRA&=~BIT(PA2);
    DDRA&=~BIT(PA3);
GIMSK=0x40;          //开外部中断0
    MCUCR=0x03;      //设置上升沿中断触发
    SREG=0x80;           //开中断
    ......
}
```

10.1.2 矩阵式键盘

为减少系统I/O资源的占用，在按键数较多时，一般采用矩阵式的键盘接口。如图10.6所示，矩阵式键盘由行线和列线组成，而按键则在行线和列线的交叉处用于连接行线和列线，这样一个矩阵式键盘含有的按键个数就等于键盘的行数和列数的乘积。

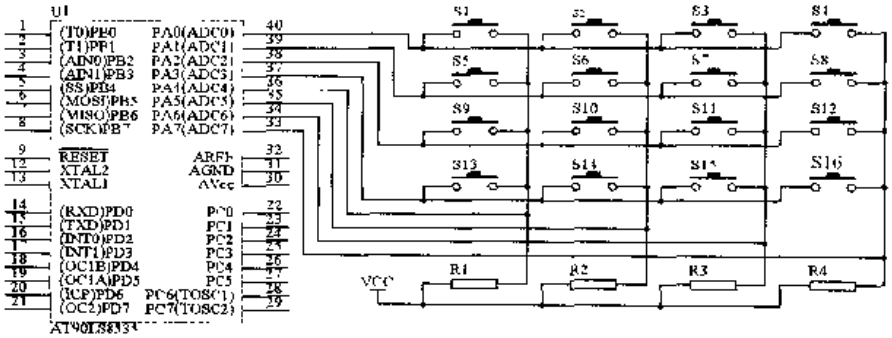


图 10.6　AT90LS8535 与矩阵式键盘的接口

矩阵式键盘适合采用查询的处理方式，其查询流程如图 10.7 所示。

由图 10.7 可见，矩阵式键盘的处理方式同非矩阵式键盘基本相同，但由于矩阵式键盘的每个按键并不独占一个 I/O 口，因此它的键盘识别采用了行扫描法。

行扫描法的基本原理为：程序首先使键盘阵列的一条列线为低电平，如果这条列线上没有键被按下，则各行线的电平也都为高；如果列线上有键被按下，则相应的行线上的电平变为低电平，这样根据行线和列线的编号就可得到按键的键值。

行扫描的过程为：先使 PA 口输出 0xef，然后输入行线的状态，并判断行线状态中是否有低电平，如果没有行线为低电平，再使输出口输出 0xdf，再判断行线状态，直到行线状态中有低电平时，被按键的被找到。

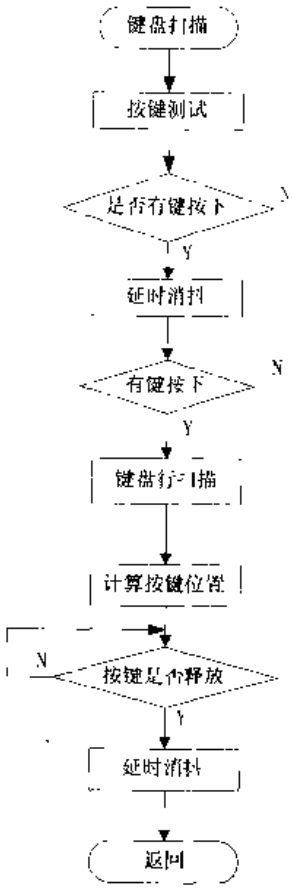


图 10.7　矩阵式键盘的查询流程

该键盘接口的源程序如下：

```
#include "io8535.h"
#define uchar unsigned char
/*判断是否有键按下，有键按下返回 1，否则返回 0*/
uchar KEYPRESSED(void)
{
DDRA=0xFF;  //设置输出模式
    PORTA=0xF0;
    DDRA=0;      //设置输入模式
    if(PINA!=0xF0)
        return 1;
    else return 0;
}
/*扫描并返回所按键的键值*/
uchar KEYSCAN(void)
{
uchar i;
    uchar j=0x7F,keycode=0xff;
    delay(10);  /*延时消抖*/
    if((KEYPRESSED())==1)/*如有键按下，查询键值*/
```

```
    {
do{
j=j<<1;      //进行行扫描
        DDRA=0xFF;  //设置输出模式
        PORTA=j;
        DDRA=0;      //设置输入模式
        keycode=PINA;
        if((keycode&0xF0)==0xF0)
            i=1;
        else i=0;
       }while(i--1);
    do{
      }while((KEYPRESSED())==1);/*等待键释放*/
    delay(5);/*延时消抖*/
    switch(keycode)      //译码
    {
        case 0x7E:
            keycode=1;break;
        case 0xBE:
            ......
    }
 }
    return keycode;
}

/*主程序*/
main()
{    //初始化
    if((KEYPRESSED(void))==1)
        key=KEYSCAN(void);
        switch(keycoce)
        {
            case xxx:
......
                    case xxx:
                ......
        }
......
}
```

10.2 LED 显示输出

LED(Light Emiting Diode)指的是发光二极管。由于 LED 数码显示器具有亮度高、寿命长，价格低廉等众多优点，因此它在单片机系统中得到了广泛的应用。

LED 显示器由 8 个发光二极管构成，因此也称为 8 段数码显示器。LED 数码显示器中的 8 个发光二极管具有两种连接方法，如图 10.8 所示。一种是把 8 个发光二极管的阳极连在一起构成一个公共的阳极，称之为共阳极接法；另一种是把 8 个发光二极管的阴极都连

在一起构成一个公共的阴极，称之为共阴极接法。

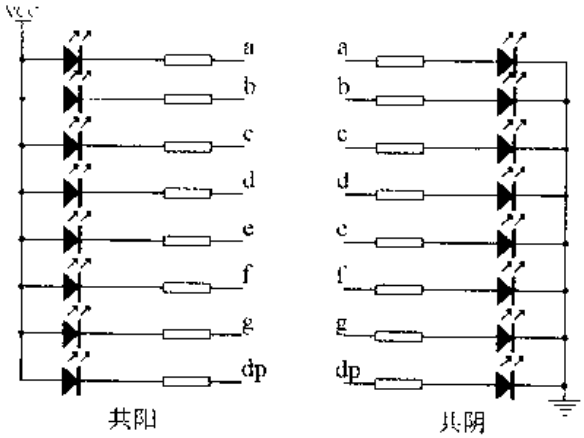


图 10.8　LED 数码显示器

为了在 LED 显示器上显示数字或符号，必须使 LED 的各个发光二极管按给定的组合发光，这就需要为 LED 提供合适的字形代码。若 8 段数码管的各显示段与代码段的对应关系如表 10.1 所示，则用 LED 显示器显示 0～9 这 10 个数字的字形代码如表 10.2 所示。

表 10.1　LED 显示器的代码段和显示段对应关系

代码位	0	1	2	3	4	5	6	7
显示段	a	b	c	d	e	f	g	dp

表 10.2　LED 显示器显示的字形代码

显示数码	共阳字形码	共阴字形码
0	0xc0	0x3f
1	0xf9	0x06
2	0xa4	0x5b
3	0xb0	0x4f
4	0x99	0x66
5	0x92	0x6d
6	0x82	0x7d
7	0xf8	0x07
8	0x80	0x7f
9	0x90	0x6f
灭	0xff	0x00

LED 显示器具有两种显示方法，一种是静态显示，另一种是动态扫描显示。静态显示采用具有锁存器的 I/O 接口电路，单片机只在数据需要更新时，才发送新的字形代码；动态扫描显示则是采用分时的方法，逐个循环地点亮各位显示器。

10.2.1 LED 的静态显示

LED 的静态显示采用具有锁存功能的 I/O 接口电路实现，这种显示电路要求单片机对 LED 数码显示器的各个显示段分别控制，因此它的接口电路较为复杂，适用于显示位数较少的场合。

图 10.9 为一个简单的静态显示电路，AT90LS8535 采用串行方式把要显示的数据输出至移位寄存器 74LS164。每个 74LS164 都可以连接一个 8 段的数码显示器，因此 3 个 74LS164 就可以连接 3 位 LED 显示器。图中 AT90LS8535 的 PA0 为串行的数据输出线，PA1 为串行的移位脉冲。74LS164 为单向 8 位的移位寄存器，可实现串行输入，并行输出。

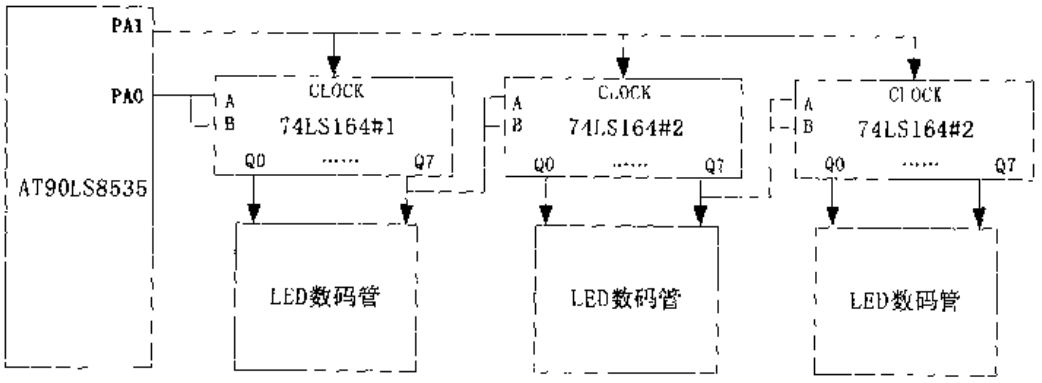


图 10.9 AT90LS8535 的静态显示电路

显示源程序为：

```
#include "io8535.h"
#define uchar unsigned char
void WD164(uchar *disdata)
{
uchar mid;
    int i,j;
    int x;
    PORTA&=~0x02;    //CLOCK 信号置低
    for(j=0;j<3;j++)
    {
        mid=disdata[j];
        for(i=0;i<8;i++)
        {
            x=mid&0x80;
            if(x==0)
                PORTA&=~0x01;
            else
                PORTA|=0x01;
            PORTA|=0x02;     //CLOCK 信号置高
            mid=mid<<1;
            PORTA&=~0xC2;    //CLOCK 信号置低
        }
    }
```

```
}
main()
{
    ddata[3]={1,2,3};
    do{
        WD164(ddata);
    }While(1);
}
```

10.2.2　LED 的动态扫描显示

实际应用的 LED 显示器都具有较多的位数，由于单片机的 I/O 资源并不是很多，因此对于这样的显示器通常都采用动态扫描的显示方式。

为了实现 LED 显示器的动态扫描，除了要把所有显示器的 8 个显示段(a、b、c、d、e、f、g、dp)的同名端连在一起之外，还必须对每一个显示器的公共极 COM 实行独立地 I/O 控制。这样，当所有的显示器接收到 CPU 向显示段输出口送出的字形代码时，还必须要由 COM 端决定显示器是否点亮。所以就可以采用分时的方法，循环地控制各个显示器的 COM 端，使各个显示段轮流被点亮。

在 LED 显示器的扫描显示过程中，任意时刻只有一位显示器被点亮，但由于人眼的视觉暂留效应，在显示刷新很快的时候，可以认为全部显示器持续点亮。

图 10.10 是一个 3 位 LED 显示器的接口电路。

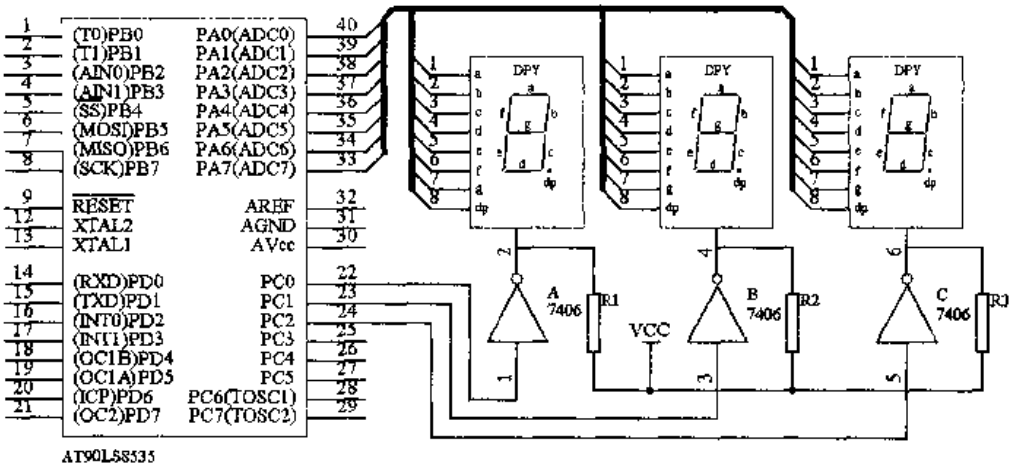


图 10.10　AT90LS8535 组成的 3 位动态扫描显示电路

其中 PC0、PC1 和 PC2 为位控口，由于 8 段数码显示器全亮时的驱动电流约为 40 mA~60 mA，因此在 PC0~PC2 口加了集电极开路的 7406 来提高驱动能力，然后再接到 3 个数码管的 COM 端。

PA 口为段控口输出，由于 AT90LS8535 的 I/O 口具有 20 mA 的驱动电流，而段控线的驱动电流约为 8 mA，因此 PA 口可直接同段控线相连。

显示子程序如下：

```
void LEDSCAN(*disdata)
```

```
{

    PORTC|=0xC1;     //显示第一位数据
    PORTA=discata[0];
    delay(10);       //延时 1 ms
    PORTC&=~0x01;    //关闭第一位显示

    PORTC|=0xC2;     //显示第二位数据
    PORTA=discata[1];
    delay(10);       //延时 1 ms
    PORTC&=~0x02;    //关闭第二位显示

    PORTC|=0xC4;     //显示第三位数据
    PORTA=disdata[2];
    delay(10);       //延时 1 ms
    PORTC&=~0x04;    //关闭第三位显示
}
```

程序说明：

- 显示子程序必须不断地被调用，才能保证持续不断地显示。
- 在动态扫描过程中，必须调用延时子程序。

以上的这个例子可以实现 LED 数码显示器的动态扫描显示，但并不太实用，因为这种连接方式，要求单片机循环地调用扫描显示程序，以实现 LED 的持续显示，这显然会影响单片机对其他事件的实时处理。为了解决这个问题，单片机系统的动态扫描显示可以采用专用芯片来实现。

10.2.3 动态扫描显示专用芯片 MC14489

MC14489 的芯片引脚如图 10.11 所示，它可以直接驱动 5 位 LED 数码显示器。片内包括一个 24 位的移位寄存器、一个锁存器、一个译码输出器和一个时钟逻辑电路。其译码输出器将单片机输入的数据转换成相应的字形码后送至 LED 数码显示器的各段和小数点。此外，片内时钟电路产生的 5 个控制信号，经驱动电路后送至 5 条位控线(BANK1～BANK5)，以实现动态扫描显示。

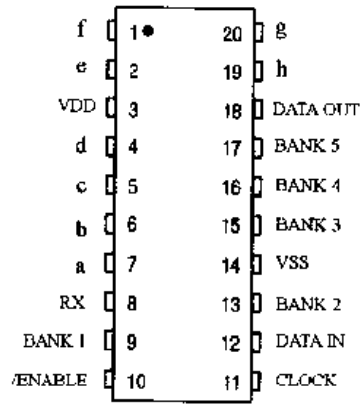


图 10.11 MC14489 的引脚图

MC14489 芯片的各引脚功能如下：

- DATA IN：串行数据输入。数据格式可以为 1 个字节的控制字或 3 个字节的显示数据。
- DATA OUT：串行数据输出。24 位移位寄存器的输出。
- BANK 1～BANK 5：5 个位控驱动信号。
- a～h：8 段显示输出。
- CLOCK：时钟输入端。时钟信号由低到高的跳变实现数据的移位操作。

- /ENABLE：低电平使能端。
- Rx：外接电流设置电阻。
- MC14489 可以接收 8 位的控制字或者是 24 位的显示数据。其控制字的输入时序如图 10.12 所示，其显示数据的输入时序如图 10.13 所示。

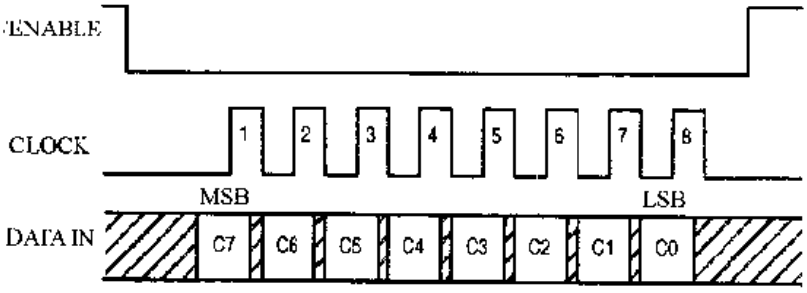


图 10.12　MC14489 的控制字输入时序

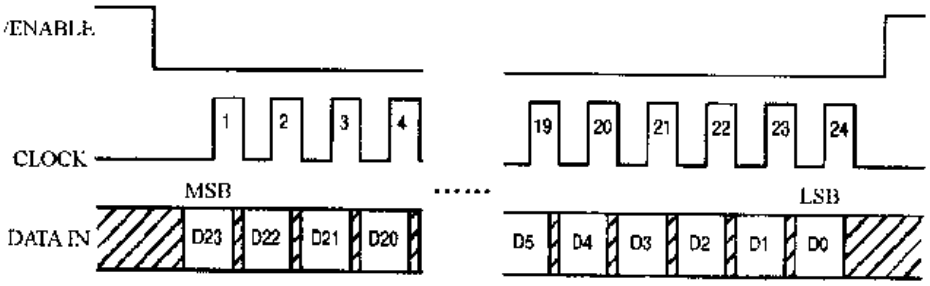


图 10.13　MC14489 的显示数据输入时序

C7、C6：L＝不译码；H＝特殊译码方式。

C5、C4：L＝十六进制译码；H＝取决于 C7。

C3、C2、C1：L＝十六进制译码；H＝取决于 C6。

C0：L＝低功耗模式；H＝正常模式。

D23：L＝暗；H＝亮。

D22～D20：小数点位置。

D19～D16：BANK 5 的显示数据。

D15～D12：BANK 4 的显示数据。

D11～D8：BANK 3 的显示数据。

D7～D4：BANK 2 的显示数据。

D3～D0：BANK 1 的显示数据。

MC14489 的数据显示格式如表 10.3 所示。

表 10.3　MC14489 的数据显示格式

BANK1～5 的显示数据				十六进	特殊
MSB			LSB	制译码	译码
L	L	L	L	0	
L	L	L	H	1	C

续表

BANK1～5的显示数据				十六进制译码	特殊译码
L	L	H	L	2	H
L	L	H	H	3	H
L	H	L	L	4	J
L	H	L	H	5	L
L	H	H	L	6	N
L	H	H	H	7	O
H	L	L	L	8	P
H	L	L	H	9	R
H	L	H	L	A	U
H	L	H	H	B	U
H	H	L	L	C	Y
H	H	L	H	D	-
H	H	H	L	E	-
H	H	H	H	F	O

由于MC14489具有锁存器，所以每一个串行数据流输入之后，都会被保存在芯片中，供5位LED数码显示器使用，并一直持续到用户再输入新的显示数据为止。也就是说，单片机只需要提供更新数据，而数据的动态扫描显示则由芯片的硬件自动完成。

一片MC14489可以同时驱动5位LED数码显示器，当需要显示更多位时，可以采用多片MC14489相级联的方式。如图10.14为两片MC14489级联组成的10位LED数码显示器。

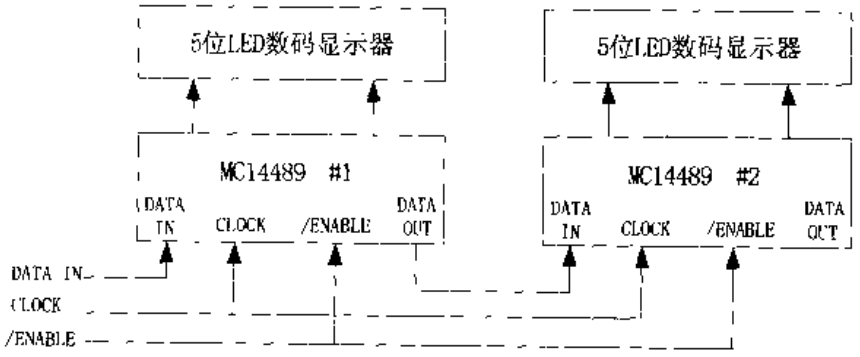


图10.14 MC14489的级联方式

由于MC14489是一种串行输入的芯片，因此它与单片机的接口非常简单。MC14489的控制线只有数据输入(DATA IN)、时钟(CLOCK)和片选3根，因此可采用AT90LS8535的任意3个I/O口和该芯片相连。如图10.15所示，数据输入端接PA0，以输入显示和控制数据；时钟信号线和PA1相连，用于产生数据输入脉冲；片选信号则和PA2相连，用于产生使能信号。

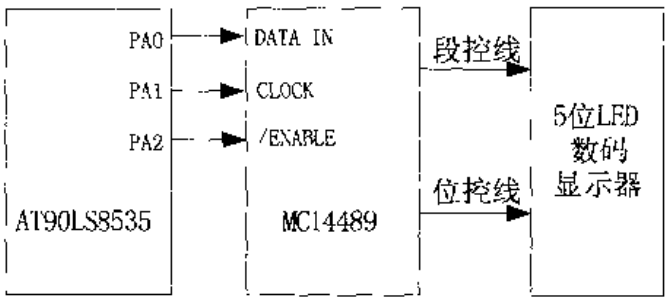


图 10.15　MC14489 与 AT90LS8535 的连接

注意：　PA0、PA1 和 PA2 的输出信号必须符合 MC14489 所要求的时序关系。

MC14489 的显示源程序为：

```
#define uchar unsigned char
/*写指令至 MC14489*/
void WCdisplay(uchar comm)
{   int i;
    int x;
    PORTA&=~0x04;   //片选信号置低
    PORTA&=~0x02;   //CLOCK 信号置低
    for(i=0;i<8;i++)
    {
        x=comm&0x80;
        if(x==0)
            PORTA&=~0x01;
        else
            PORTA|=0x01;
        PORTA|=0x02;    //CLOCK 信号置高
        comm=comm<<1;
        PORTA&=~0x02;   //CLOCK 信号置低
    }
    PORTA|=0x04;    //片选信号置高
}

/*写数据至 MC14489*/
void WDdisplay(uchar *disdata)
{   uchar mid;
    int i,j;
    int x;
    PORTA&=~0x04;   //片选信号置低
    PORTA&=~0x02;   //CLOCK 信号置低
    for(j=0;j<5;j++)
    {
        mid=disdata[j];
        for(i=0;i<8;i++)
        {
            x=mid&0x80;
            if(x==0)
                PORTA&=~0x01;
            else
```

```
                PORTA|=0x01;
            PORTA|=0x02;     //CLOCK 信号置高
            mid=mid<<1;
            PORTA&=~0x02;    //CLOCK 信号置低
        }
    }
    PORTA|=0x04;     //片选信号置高
}

main()
{   uchar ddata[5]=[1,2,3,4,5];
    WCdisplay(0x0FF);   //设置显示方式
    WDdisplay(ddata);   //写入显示数据
}
```

10.3 LCD 显示输出

液晶显示器 LCD(Liquid Crystal Display)具有体积小、重量轻、功耗低等优点，在各种仪表和工业应用系统中，得到了越来越广泛的应用。下面将介绍 LCD 接口的相关技术。

10.3.1 字符型 LCD

字符型 LCD 是一种 8 段的数码显示器件，本节以 HT1621 控制器来说明字符型 LCD 的接口及其应用编程。

1. HT1621 控制器概述

HT1621 是一种 4×8 段的字符型 LCD 控制器，它采用串行接口同微处理器相连，因此很容易构成多模块的 LCD 应用系统。HT1621 的内部结构如图 10.16 所示。

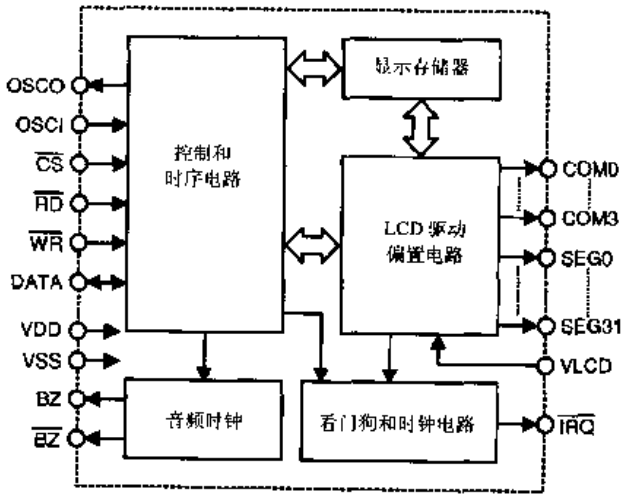


图 10.16 HT1621 的内部结构

- /CS：片选信号。

- BZ、/BZ：音频输出。
- /WR、/RD、DATA：串行接口。
- COM0~COM3、SEG0~SEG31：LCD 输出。
- /IRQ：看门狗和时钟中断输出。

注意： 不同类型的 LCD 模块接口形式略有不同。

该控制器的主要性能如下：

- 工作电压：2.4 V~5.2 V。
- 振荡器：内置 256 KHz 的 RC 振荡器。
- LCD 驱动方式：片内 LCD 驱动频率发生电路。
- 接口电路：3 线串行接口。
- 工作环境：-25℃~+75℃。

2. HT1621 的指令系统

HT1621 通过两种不同的指令格式设置芯片的工作方式，并实现显示数据的传输。设置 HT1621 的指令称为模式指令，其指令识别号(ID)为 100；实现 MCU 和 HT1621 之间数据传送的指令称为数据指令，它又可分为读、写、读—修改—写操作。表 10.4 为数据指令和模式指令的指令识别号。

表 10.4　HT1621 控制器的指令识别号

操作	模式	指令识别号
读	数据指令	110
写	数据指令	101
读—修改—写	数据指令	101
设置	模式指令	100

HT1621 的写时序如图 10.17 所示。

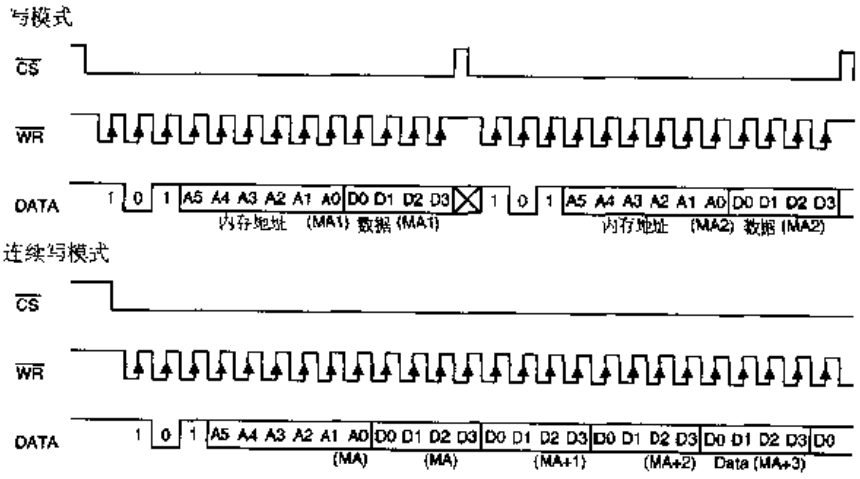


图 10.17　HT1621 控制器的写时序

3、MCU 与 HT1621 的接口和编程

AT90LS8535 与 HT1621 显示模块的接口方式如图 10.18 所示。

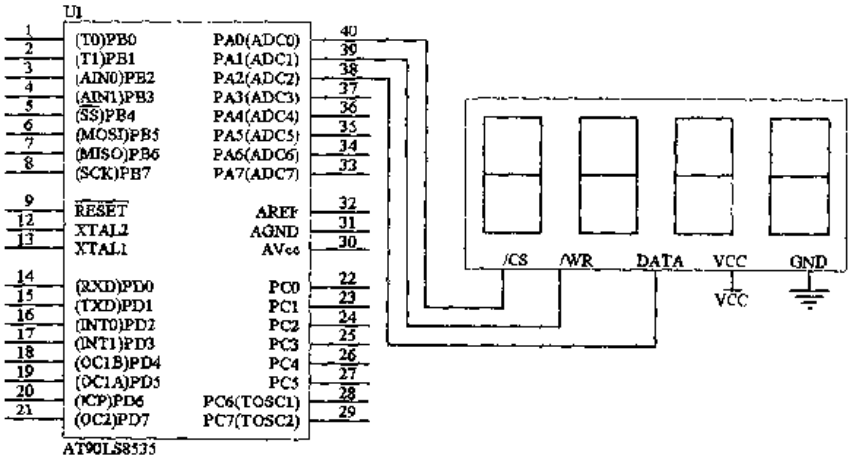


图 10.18 AT90LS8535 与 HT1621 显示模块的接口电路

下面是字符显示的源程序。

```
#include "io8535.h"
const char number[21][2]={
{0xB0,0xe0},{0x00,0x60},{0x70,0xc0},{0x50,0xe0},{0xc0,0x60},
{0xd0,0xa0},{0xf0,0xa0},{0x00,0xe0},{0xf0,0xe0},{0xd0,0xe0},
{0xB0,0xf0},{0x00,0x70},{0x70,0xd0},{0x50,0xf0},{0xc0,0x70},
{0xd0,0xb0},{0xf0,0xb0},{0x00,0xf0},{0xf0,0xf0},{0xd0,0xf0},};
//延时子程序
void delay(void)
{   unsigned char i;
    for(i=0;i<100;i++)
    ;
}
//写指令子程序
void wcommand(unsigned char command)
{   unsigned char i,mid;
    mid=command;
    PORTA&=~BIT(PA0);   //片选使能

    PORTA|=BIT(PA2); delay();       //发送 100
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    PORTA&=~BIT(PA2);delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    PORTA&=~BIT(PA2);delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    for(i=0;i<8;i++)    //发送指令代码
    {
```

```
        if((mid&0x80)==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        delay();PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();
        mid=mid<<1;
    }

    PORTA&=~BIT(PA2);delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();
    PORTA|=BIT(PA0);    // /cs=1

}
    //显示数据写入子程序
void wdata(unsigned address,unsigned char j)
{
    unsigned char i,mid;
    switch(address)
    {
        case 1:
            address=0;
            break;
        case 2:
            address=8;
            break;
        case 3:
            address=0x10;
            break;
        default:
            address=0x18;
            break;
    }
    PORTA&=~BIT(PA0);   //片选使能

    PORTA|=BIT(PA2); delay();       //发送 101
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    PORTA&=~BIT(PA2);delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    PORTA|=BIT(PA2); delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    mid=address;
    for(i=0;i<6;i++)
    {
        if((mid&0x80)==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        delay();PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();
```

```
        mid=mid<<1;
    }

    mid=number[j][0];
    for(i=0;i<4;i++)
    {
        if((mid&0x80)==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        delay();PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();
        mid=mid<<1;
    }

    PORTA|=BIT(PA0);    // /cs=1
    delay();
    PORTA&=~BIT(PA0);   //片选使能
        PORTA|=BIT(PA2); delay();       //发送101
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    PORTA&=~BIT(PA2);delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    PORTA|=BIT(PA2); delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    mid=address|4;
    for(i=0;i<6;i++)
    {
        if((mid&0x80)==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        delay();PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();
        mid=mid<<1;
    }

    mid=number[j][1];
    for(i=0;i<4;i++)
    {
        if((mid&0x80)==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        delay();PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();
        mid=mid<<1;
    }
    PORTA|=BIT(PA0);    // /cs=1
}

//液晶初始化程序
```

```
void lcd_init(void)
{
    unsigned char i,j,mid;
    PORTA|=BIT(PA0);    // /cs=1
    PORTA|=BIT(PA1);
    PORTA|=BIT(PA2);
    wcommand(1);
    wcommand(3);
    wcommand(0x29);

    PORTA&=~BIT(PA0);   //片选使能

    PORTA|=BIT(PA2); delay();       //发送 101
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    PORTA&=~BIT(PA2);delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    PORTA|=BIT(PA2); delay();
    PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();

    mid=0x20;
    for(i=0;i<6;i++)
    {
        if((mid&0x80)==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        delay();PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();
        mid=mid<<1;
    }

    mid=0;
    for(j=0;j<24;j++)
    {
        for(i=0;i<4;i++)
        {
            if((mid&0x80)==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        delay();PORTA&=~BIT(PA1);delay();PORTA|=BIT(PA1); delay();
        mid=mid<<1;
        }
    }

    PORTA|=BIT(PA0);    // /cs=1
    delay();
}

//主程序
```

```
main()
{
......
lcd_init();
wdata(0,6);
......
}
```

10.3.2 点阵型 LCD

本节以 HS12864I 液晶显示器为例，来说明点阵型 LCD 显示模块的应用。

1. HS12864I 显示模块概述

HS12864I 是一种图形点阵液晶显示模块，它主要由行驱动器/列驱动器和 128×64 全点阵液晶显示器组成，它除了可以显示 8×4(16×16 点阵)的汉字外，还可完成图形显示功能。

HS12864I 的主要技术参数如下：

- 电源：DC+5V，模块内自带用于 LCD 驱动的负压电路。
- 显示内容：128×64 全屏幕点阵。
- 指令系统：7 种指令。
- 接口形式：与控制器采用 8 位数据总线和 8 条控制线相连。
- 工作环境：-10℃~+50℃。

2. HS12864I 显示模块的外部接口说明

HS12864I 显示模块的外部接口如表 10.5 所示。

表 10.5 HS12864I 显示模块的外部引脚功能

引脚号	引脚名称	电平	引脚功能说明
1	VSS	0	电源
2	VDD	5V	电源
3	V0	H/L	液晶显示器驱动电压
4	D/I	H/L	D/I=“L”，表示 DB7~DB0 为显示数据； D/I=“H”，表示 DB7~DB0 为显示指令
5	R/W	H/L	R/W=“L”，E=“H”数据由控制器输出至 DB7~DB0 R/W=“H”，E 的下降沿，数据由 DB7~DB0 输入至控制器
6	E	H/L	R/W=“L”，E 的下降沿锁存 DB7~DB0； R/W=“H”，E 为高电平时，数据由控制器输出至 DB7~DB0
7	DB0	H/L	数据线
8	DB1	H/L	数据线
9	DB2	H/L	数据线
10	DB3	H/L	数据线
11	DB4	H/L	数据线

12	DB5	H/L	数据线
13	DB6	H/L	数据线
14	DB7	H/L	数据线
15	CS1	H/L	右边液晶块芯片的片选信号(高电平选择)
16	CS2	H/L	左边液晶块芯片的片选信号(高电平选择)
17	RET	H/L	复位信号(低电平复位)
18	VOUT	－10V	LCD 负压驱动电压
19	LED＋	－	显示模块背光电源
20	LED－	－	显示模块背光电源

3. HS12864I 显示模块的硬件构成说明

HS12864I 的硬件电路如图 10.19 所示。其中 IC1 和 IC2 为列驱动器，IC3 为行驱动器。IC1、IC2 和 IC3 主要含指令寄存器、数据寄存器、忙标志位、显示控制触发器、XY 地址计数器、显示数据 RAM 和 Z 地址计数器 7 个部分。各部分的功能如下：

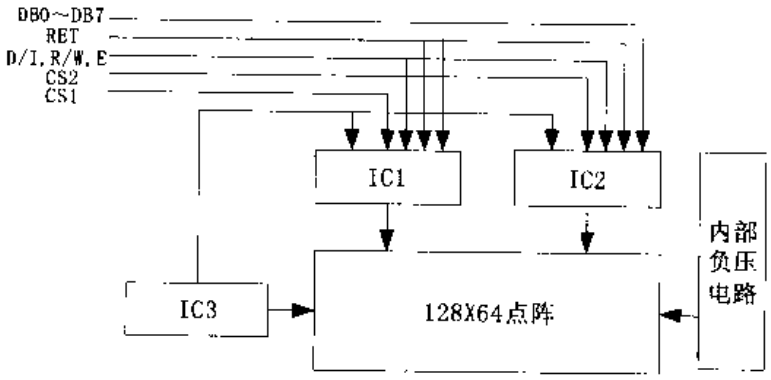


图 10.19　HS12864I 的硬件组成

- 指令寄存器(IR)
  指令寄存器用于控制指令的输入，它与数据寄存器相对应。当 D/I＝“0”时，E 信号的下降沿将数据总线上的数据锁存至该寄存器。
- 数据寄存器(DR)
  数据寄存器用于寄存显示数据，它与指令寄存器相对应。当 D/I=“1”时，E 信号的下降沿将数据总线上的图形显示数据写入数据寄存器；或在 E 信号的高电平作用下由数据寄存器输出至 DB7～DB0。数据寄存器和 DDRAM 之间的数据传输由模块自动完成。
- 忙标志位(BF)
  忙标志位提供芯片的工作情况。BF＝“1”表明模块正在进行操作，此时模块不接收指令和数据。当 BF＝“0”时，模块为准备状态，这时模块可接收指令和数据。
- 显示控制触发器(DFF)
  显示控制触发器是用于开关模块屏幕显示用的。DFF＝“1”为开显示，DDRAM 中的数据就可以显示在屏幕上。DFF＝“0”为关显示。

- XY地址计数器

  XY地址计数器是一个9位的计数器。其中高3位为X地址计数器，低6位为Y地址计数器。XY地址计数器实际上是作为DDRAM的地址指针，X地址计数器为DDRAM的页地址指针，Y地址为DDRAM的Y地址指针。

  X地址计数器是没有计数功能的，它只能用指令的形式设置。

  Y地址计数器具有循环计数功能，各显示数据写入后，Y地址可以自动加1。

- 显示数据RAM(DDRAM)

  DDRAM是存储图形数据的。数据为“1”表示显示相应点，数据为“0”表示不显示。DDRAM的地址与显示位置的关系如表10.6所示。

表10.6 DDRAM地址表

CS1=1						CS2=1					
Y=	0	1	…	62	63	0	1	…	62	63	行号
	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	0
X=0	…	…	…	…	…	…	…	…	…	…	…
	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	7
…	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	0
	…	…	…	…	…	…	…	…	…	…	…
X=7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	7
	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	0
	…	…	…	…	…	…	…	…	…	…	…
	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	7

- Z地址计数器

  Z地址计数器是一个6位的计数器，同Y计数器一样，这个计数器也具有循环计数的功能，它用于显示行扫描同步。当一行的扫描完成完成后，此计数器自动加1，指向下一行的扫描数据。

Z地址计数器可以用设置显示起始行的指令设置，因此，就可以设置DDRAM的数据从哪一行开始，显示在屏幕的第一行。

4. HS128641显示模块的指令说明

- 显示开关控制

  代码：

R/W	D/I	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	1	1	1	1	D

  指令说明：D=“1”，显示模块开，此时可对显示器进行各种操作。D=“0”，显示模块关，此时不可对显示器进行操作。

- 设置显示起始行

  代码：

R/W	D/I	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	1	A5	A4	A3	A2	A1	A0

指令说明：该指令将 A5～A0 的 6 位地址送入 Z 地址计数器，以确定屏幕的显示起始行和 DDRAM 中数据的对应关系。

- 设置页地址

  代码：

R/W	D/I	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	0	1	1	1	A2	A1	A0

指令说明：所谓页地址就是 DDRAM 的行地址，8 行为 1 页。HS12864I 显示模块共有 64 行即 8 页。读写数据对页地址没有影响，页地址通过本指令或复位信号设置。

- 设置 Y 地址

  代码：

R/W	D/I	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	A5	A4	A3	A2	A1	A0

指令说明：该指令将 A5～A0 送入 Y 地址计数器，作为 DDRAM 的 Y 地址指针。在对 DDRAM 进行读写操作后，Y 地址指针自动加 1，指向下一个 DDRAM 单元。

- 读状态

  代码：

R/W	D/I	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	BUSY	0	ON/OFF	RET	0	0	0	0

指令说明：当 R/W＝“1”，D/I＝“0”时，在 E 信号的高电平作用下，模块的状态信号输出到数据总线。

BUSY 标志表示模块忙，不能处理外部送来的指令和数据。

RST 表示显示模块内部正在初始化，此时，模块也不能处理外部的指令和数据。

- 写显示数据

  代码：

R/W	D/I	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	D7	D6	D5	D4	D3	D2	D1	D0

指令说明：这条指令把数据总线上的 D7～D0 写入相应的 DDRAM 单元。指令执行后，Y 地址指针自动加 1。

- 读显示数据

  代码：

R/W	D/I	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	D7	D6	D5	D4	D3	D2	D1	D0

指令说明：此指令把 DDRAM 中的数据 D7~D0 输出至数据总线上。指令执行后，Y 地址指针加 1。

5. HS12864I 显示模块的操作时序。

图 10.20 为读操作时序，图 10.21 为写操作时序。

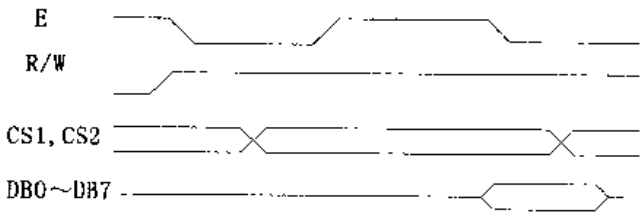


图 10.20 HS12864I 的读操作时序

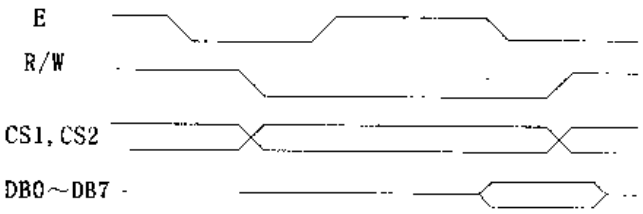


图 10.21 HS12864I 的写操作时序

6. 与 HS12864I 显示模块的接口和编程

AT90LS8535 与 HS12864I 显示模块的接口方式如图 10.22 所示。

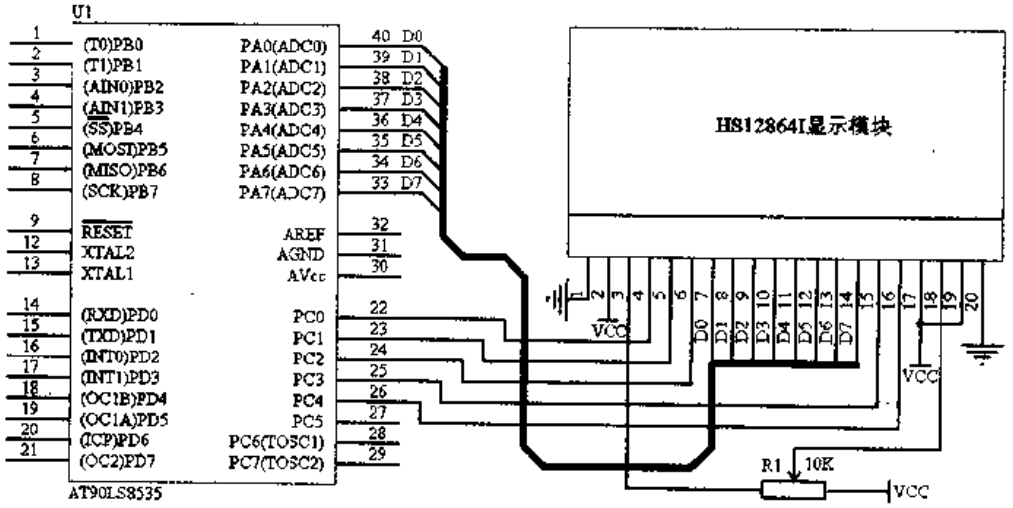


图 10.22 AT90LS8535 与 HS12864I 模块的接口电路

以下是显示字符的源程序：

```
#include "io8535.h"
```

```
#define uchar unsigned char

/*预设 LCD DDRAM 页地址和 Y 地址*/
#define PREPAGE 0xB8
#define PRECOL 0x40

char status;
/*字符串列表*/
const char string[15]={"HELLO WORLD?\n"};

/*西文字符库列表*/
const char chartable[96][8]={
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x4F,0x00,0x00,0x00,0x00},//"!"
{0x00,0x00,0x07,0x00,0x07,0x00,0x00,0x00},//""
{0x00,0x14,0x7f,0x14,0x7F,0x14,0x00,0x00},//" #"
{0x00,0x24,0x2A,0x7F,0x2A,0x12,0x00,0x00},//"$"
{0x00,0x23,0x13,0x08,0x64,0x62,0x00,0x00},//"%"
{0x00,0x36,0x49,0x55,0x22,0x00,0x00,0x00},//"&"
{0x00,0x00,0x05,0x03,0x00,0x00,0x00,0x00},//"'"
{0x00,0x00,0x1C,0x22,0x41,0x00,0x00,0x00},//"( "
{0x00,0x00,0x41,0x22,0x1C,0x00,0x00,0x00},//")
{0x00,0x14,0x08,0x3E,0x08,0x14,0x00,0x00},//"*"
{0x00,0x00,0x08,0x3E,0x08,0x08,0x00,0x00},//"+"
{0x00,0x00,0x50,0x30,0x00,0x00,0x00,0x00},//";"
{0x00,0x08,0x08,0x08,0x08,0x08,0x00,0x00},//"—"
{0x00,0x00,0x60,0x60,0x00,0x00,0x00,0x00},//"."
{0x00,0x20,0x10,0x08,0x04,0x02,0x00,0x00},/"/"
{0x00,0x3E,0x51,0x49,0x45,0x3E,0x00,0x00},//"0"
{0x00,0x00,0x42,0x7f,0x40,0x00,0x00,0x00},//"1"
{0x00,0x42,0x61,0x51,0x49,0x46,0x00,0x00},//"2"
{0x00,0x21,0x41,0x45,0x4B,0x31,0x00,0x00},//"3"
{0x00,0x18,0x14,0x12,0x7F,0x10,0x00,0x00},/"4"
{0x00,0x27,0x45,0x45,0x45,0x39,0x00,0x00},//"5"
{0x00,0x3C,0x4A,0x49,0x49,0x30,0x00,0x00},//"6"
{0x00,0x01,0x01,0x79,0x05,0x03,0x00,0x00},//"7"
{0x00,0x36,0x49,0x49,0x49,0x36,0x00,0x00},//"8"
{0x00,0x06,0x49,0x49,0x29,0x1E,0x00,0x00},//"9"
{0x00,0x00,0x36,0x36,0x00,0x00,0x00,0x00},//": "
{0x00,0x00,0x56,0x36,0x00,0x00,0x00,0x00},//";"
{0x00,0x08,0x14,0x22,0x41,0x00,0x00,0x00},//"<"
{0x00,0x14,0x14,0x14,0x14,0x14,0x00,0x00},//"="
{0x00,0x00,0x41,0x22,0x14,0x08,0x00,0x00},//">"
{0x00,0x02,0x01,0x51,0x09,0x06,0x00,0x00},//"? "
{0x00,0x32,0x49,0x79,0x41,0x3E,0x00,0x00},//"@"
{0x00,0x7E,0x11,0x11,0x11,0x7E,0x00,0x00},//"A"
{0x00,0x41,0x7f,0x49,0x49,0x36,0x00,0x00},//"B"
{0x00,0x3E,0x41,0x41,0x41,0x22,0x00,0x00},//"C?
{0x00,0x41,0x7f,0x41,0x41,0x3E,0x00,0x00},//"D"
{0x00,0x7F,0x49,0x49,0x49,0x49,0x00,0x00},//"E"
{0x00,0x7f,0x09,0x09,0x09,0x01,0x00,0x00},//"F "
```

```
{0x00,0x3E,0x41,0x41,0x49,0x7A,0x00,0x0C},//"G"
{0x00,0x7f,0x08,0x08,0x08,0x7f,0x00,0x00},//""
{0x00,0x00,0x41,0x7F,0x41,0x00,0x00,0x00},//"I "
{0x00,0x20,0x40,0x41,0x3F,0x01,0x00,0x00},//"J'
{0x00,0x7F,0x08,0x14,0x22,0x41,0x00,0x00},/"K"
{0x00,0x7F,0x40,0x40,0x40,0x40,0x00,0x00},//"L"
{0x00,0x7F,0x02,0x0c,0x02,0x7f,0x00,0x00},//"M"
{0x00,0x7f,0x06,0x08,0x30,0x7f,0x00,0x00},//"N"
{0x00,0x3E,0x41,0x41,0x41,0x3E,0x00,0x00},// "O"
{0x00,0x7f,0x49,0x09,0x09,0x06,0x00,0x00},//"P"
{0x00,0x3E,0x41,0x51,0x21,0x5e,0x00,0x00},//"Q"
{0x00,0x7F,0x09,0x19,0x29,0x46,0x00,0x00},//"R"
{0x00,0x26,0x49,0x49,0x49,0x32,0x00,0x00},//"S"
{0x00,0x01,0x01,0x7F,0x01,0x01,0x00,0x00},//"T "
{0x00,0x3f,0x40,0x40,0x40,0x3F,0x00,0x00},//"U"
{0x00,0x1F,0x20,0x40,0x20,0x10,0x00,0x00},//"V"
{0x00,0x7f,0x20,0x18,0x20,0x7F,0x00,0x00},//"W'
{0x00,0x63,0x14,0x08,0x14,0x63,0x00,0x00},//"X"
{0x00,0x07,0x08,0x70,0x08,0x07,0x00,0x00},//"Y''
{0x00,0x61,0x51,0x49,0x45,0x43,0x00,0x00},//"Z"
{0x49,0x00,0x7F,0x41,0x00,0x00,0x00,0x00},//"["
{0x00,0x02,0x04,0x08,0x10,0x20,0x00,0x00},//""
{0x00,0x00,0x41,0x41,0x7f,0x00,0x00,0x00},//"]"
{0x00,0x04,0x02,0x01,0x02,0x04,0x00,0x00},//"^"
{0x00,0x40,0x40,0x40,0x40,0x40,0x00,0x00},//"_"
{0x00,0x01,0x02,0x04,0x00,0x00,0x00,0x00},/"'"
{0x00,0x20,0x54,0x54,0x54,0x78,0x00,0x00},//"a"
{0x00,0x7f,0x48,0x44,0x44,0x38,0x00,0x00},//"b"
{0x00,0x38,0x44,0x44,0x44,0x28,0x00,0x00},//" c"
{0x00,0x38,0x44,0x44,0x48,0x7F,0x00,0x00},//" d"
{0x00,0x38,0x54,0x54,0x54,0x18,0x00,0x00},//"e"
{0x00,0x00,0x08,0x7E,0x09,0x02,0x00,0x00},//"f "
{0x00,0x0C,0x52,0x52,0x4C,0x3E,0x00,0x00},//"g"
{0x00,0x7f,0x08,0x04,0x04,0x78,0x00,0x00},//"/"
{0x00,0x00,0x44,0x7D,0x40,0x00,0x00,0x00},//"i"
{0x00,0x20,0x40,0x44,0x3D,0x00,0x00,0x00},//"j "
{0x00,0x00,0x7F,0x10,0x28,0x44,0x00,0x00},//"k"
{0x00,0x00,0x41,0x7F,0x40,0x00,0x00,0x00},//"l"
{0x00,0x7C,0x04,0x78,0x04,0x78,0x00,0x00},//"m"
{0x00,0x7C,0x08,0x04,0x04,0x78,0x00,0x00},//"n"
{0x00,0x38,0x44,0x44,0x44,0x38,0x00,0x00},//"o"
{0x00,0x7E,0x0C,0x12,0x12,0x0C,0x00,0x00},//"p"
{0x00,0x0C,0x12,0x12,0x0C,0x7E,0x00,0x00},//"q"
{0x00,0x7C,0x08,0x04,0x04,0x08,0x00,0x00},//"r"
{0x00,0x58,0x54,0x54,0x54,0x64,0x00,0x00},//"s"
{0x00,0x04,0x3F,0x44,0x40,0x20,0x00,0x00},//"t"
{0x00,0x3C,0x40,0x40,0x3C,0x40,0x00,0x00},//"u"
{0x00,0x1C,0x20,0x40,0x20,0x1C,0x00,0x00},//"v"
{0x00,0x3C,0x40,0x30,0x40,0x3C,0x00,0x00},//"w"
{0x00,0x44,0x28,0x10,0x28,0x44,0x00,0x00},//"x"
{0x00,0x1C,0xA0,0xA0,0x90,0x7C,0x00,0x00},//"y"
```

```
{0x00,0x44,0x64,0x54,0x4C,0x44,0x00,0x00},//"z"
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}};

void wcodeL(uchar c)/*左边液晶块写指令*/
{
    DDRA=0xff;  //A 口输入
    while(1){
            PORTC&=~BIT(PC0);   //设置成指令模式
            PORTC|=BIT( PC1);   //设置读模式
            PORTC|=BIT( PC4);   //选择左半屏
            PORTC|=BIT( PC2);   //数据输出至数据总线
            status=PINA;
            status=status&0x80;
        if(status==0) break;
       }
    DDRA=0;      //设置 A 口输出
    PORTC&=~BIT(PC1);   //设置写模式
    PORTA=c;
    PORTC&=~BIT(PC2);   //数据写入 DDRAM
}

void wdataL(uchar d)/*左边液晶块写数据*/
{
    DDRA=0xff;  //A 口输入
    while(1){
            PORTC&=~BIT(PC0);   //设置成指令模式
            PORTC|=BIT( PC1);   //设置读模式
            PORTC|=BIT( PC4);   //选择左半屏
            PORTC|=BIT( PC2);   //数据输出至数据总线
            status=PINA;
            status=status&0x80;
            if(status==0) break;
          }
    DDRA=0;      //设置 A 口输出
    PORTC|=BIT(PC0);    //设置成数据模式
    PORTC&=~BIT(PC1);   //设置写模式
    PORTA=d;
    PORTC&=~BIT(PC2);   //数据写入 DDRAM
}

void wcodeR(uchar c)/*右边液晶块写指令*/
{
    DDRA=0xff;  //A 口输入
    while(1){
            PORTC&=~BIT(PC0);   //设置成指令模式
            PORTC|=BIT( PC1);   //设置读模式
            PORTC|=BIT( PC3);   //选择右半屏
            PORTC|=BIT( PC2);   //数据输出至数据总线
            status=PORTA;
            status=status&0x80;
            if(status==0) break;
```

```
        }
    DDRA=0;      //设置A口输出
    PORTC&=~BIT(PC1);   //设置写模式
    PORTA=c;
    PORTC&=~BIT(PC2);   //数据写入DDRAM
}

void wdataR(uchar d)/*右边液晶块写数据*/
{
    DDRA=0xff;  //A口输入
    while(1){
            PORTC&=~BIT(PC0);   //设置成指令模式
            PORTC|=BIT( PC1);   //设置读模式
            PORTC|=BIT( PC3);   //选择右半屏
            PORTC|=BIT( PC2);   //数据输出至数据总线
            status=PINA;
            status=status&0x80;
            if(status==0) break;
           }
    DDRA=0;      //设置A口输出
    PORTC|=BIT(PC0);    //设置成数据模式
    PORTC&=~BIT(PC1);   //设置写模式
    PORTA=d;
    PORTC&=~BIT(PC2);   //数据写入DDRAM
}

/*LCD初始化子程序*/
void lcdinit(void)
{
wcodeL(0x0c0);/*0xc0设置显示起始行从第一行开始*/
    wcodeR(0x0c0);
    wcodeL(0x3f);/*开显示*/
    wcodeR(0x3f);
}

/*清屏子程序，已指定字符写满整个屏幕*/
void clear(uchar symbol)
{
uchar page=0x00;
    uchar col=0x00;
    for(page=0x00;page<0x08;page++)
    {
wcodeL(PREPAGE+page);
        for(col=0x00;col<0x40;col++)
        {
wcodeL(PRECOL+col);
            wdataL(symbol);
        }
    }
    for(page=0x00;page<0x08;page++)
    {
```

```
wcodeR(PREPAGE+page);
        for(col=0x00;col<0x40;col++)
        {
wcodeR(PRECOL+col);
            wdataR(symbol);
        }
    }
}

/*设置设置 LCD 中 DDRAM 的页地址和 Y 地址*/
/*左边液晶块*/
void addressL(uchar page,uchar col)
{
wcodeL(PREPAGE+page);
    wcodeL(PRECOL+col);
}
/*右边液晶块*/
void addressR(uchar page,uchar col)
{
    wcodeR(PREPAGE+page);
    wcodeR(PRECOL+col);
}

/*显示子程序，通过调用 display(x,y,*string)显示，string 为一个字符数组*/
void displaystring(uchar x,uchar y,uchar *onechar)
{
    uchar i=0,j=0;
    if(y>63)
    {
        y=y-64;
        addressR(x,y);
        while((*onechar)!='\n')
        {
            for(i=0;i<8;i++)
            {   wdataR(chartable[*onechar-0x20][i]);    }
            onechar++;
        }
    }
    else{
        addressL(x,y);
        addressR(x,0);
        while((*onechar)!='\n')
            {
                for(i=0;i<8;i++)
                {   wdataL(chartable[*onechar-0x20][i]);}
                onechar++;
                j++;
                if(j>7) break;
            }
        while((*onechar)!='\n')
            {
```

```
                for(i=0;i<8;i++)
                {   wdataR(chartable[*onechar-0x20][i]);    }
                onechar++;
            }
        }
}

/*主程序*/
main()
{
        ......
    lcdinit(void);
    clear(0x00);
    displaystring(0,0,string);
        ......
}
```

10.4 ISD2500 系列语音芯片的编程

ISD2500 芯片是美国 ISD 公司生产的一种录/放语音芯片，它可以采用单芯片的控制方式，也可以采用微处理器的控制方式。按录放时间的不同，ISD2500 系列芯片可以分为 ISD2560、2575、2590 和 25120 这 4 个品种。

同 1400 系列的语音电路一样，由 ISD2500 系列芯片构成的语音系统也具有音质好、使用方便等优点。它的特点在于 ISD2500 系列提供有 10 个地址输入端(1400 系列仅为 8 个)，可寻址的范围可达 1024 位，最多能分 600 段；芯片还设有便于多个器件级联的 OVF(溢出)端。此外，ISD2500 的片内 EEPROM 为 480 KB(1400 系列为 128 KB)，所以它具有更长的录放时间。

10.4.1 ISD2500 的片内结构和引脚

图 10.23 是 ISD2500 的内部结构图。

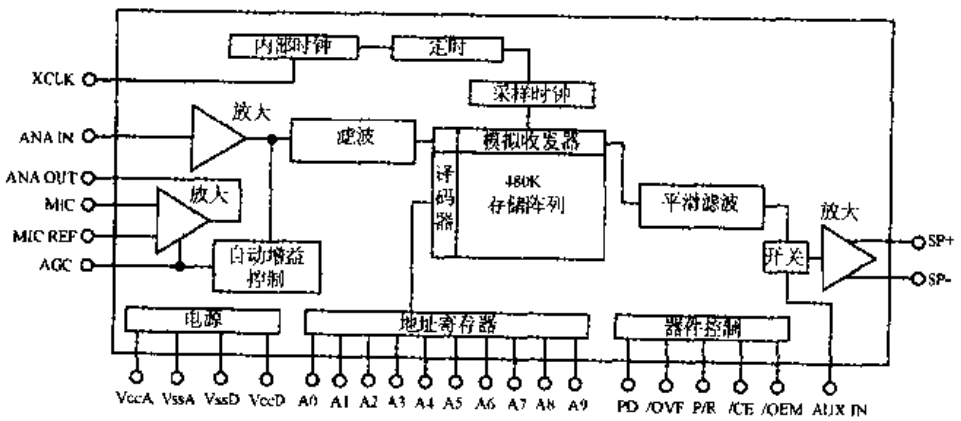


图 10.23 ISD2500 的内部结构

- 480 K 存储阵列：ISD2500 系列采用非易失性的存储器，因此即便系统掉电，存储在芯片中的信息也不会丢失。
- 微处理器接口：ISD2500 系列除了提供单芯片的处理模式外，还提供了一种便于操作的微处理接口，该接口可接收 CPU 的控制信号，输出反应 ISD2500 内部状态的信号，并控制语音信号的合成。
- 时钟电路：用于产生芯片工作所需要的时序和采样时钟。

图 10.24 是 ISD2500 的引脚图，各引线端功能介绍如下：

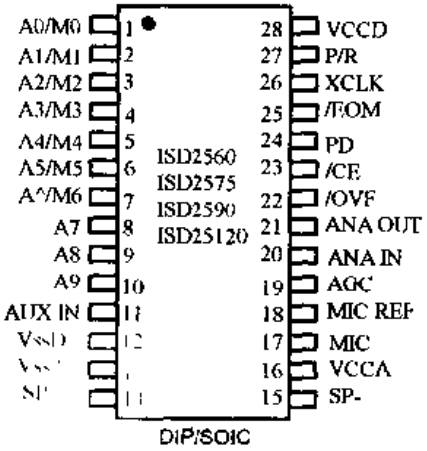


图 10.24　ISD2500 的引脚图

- A0/M0～A6/M6：地址。
- A7～A9：地址。
- AUX IN：辅助输入。
- VSSD、VSSA：数字和模拟地。
- SP+、SP−：扬声器输出。
- VCCA 、VCCD 模拟、数字信号电源正极。
- MIC、MIC REF：麦克风输入和输入参考端。
- AGC：自动增益控制。
- ANA IN、OUT：模拟信号输入和输出。
- /OVF：溢出。
- /CE：片选(低电平允许芯片工作)。
- PD：芯片低功耗状态控制。
- /EOM：录放音结束信号输出。
- XCLK：外部时钟。
- P//R：录/放控制选择。

10.4.2　ISD2500 的操作

1. 操作模式

当最高位地址(MSB)A8、A9 都为高电平时，地址端就作为操作模式选择端(高电平有

效)，表 10.7 为操作模式表。

表 10.7 ISD2500 的操作模式

模式控制	功 能	典型应用	可同时执行的操作
M0	信息检索	快速检索信息	M4，M5，M6
M1	删除 EOM 标志	在全部语音录放结束时，给出 EOM 标志	M3，M4，M5，M6
M2	未用	当工作模式操作时，此端应接低电平	N/A
M3	循环放音	从 0 地址开始连续重复放音	M1，M5，M6
M4	连续寻址	可录放连续的多段信息	M0，M1，M5
M5	CE 电平触发	允许信号中止	M0，M1，M3，M4
M6	按钮控制	简化器件接口	M0，M1，M3

注意：

- 所有操作模式下的操作都是从 0 地址开始，以后的操作根据模式的不同，而从相应的地址开始工作。当电路中录音转放音或进入省电状态时，地址计数器复位为 0。
- 操作模式位不加锁定，可以在 MSB(A8、A9)地址位为高电平时，/CE 电平变低的任何时间执行操作模式操作。如果下一片选周期 MSB(A8、A9)地址位中有一个(或两个)变为低电平，则执行信息地址，即从该地址录音或放音，原来设定的操作模式状态丢失。

2. 分段录/放音

2500 系列最多可分为 600 段，只要在分段录/放音之前，给地址 A0~A9 赋值，录音和放音就都可以从设定的起始地址开始，录音结束由停止键操作决定，芯片内部自动在该段的结束位置插入结束标志(EOM)；而放音时芯片遇到 EOM 标志即自动停止放音，如表 10.8 所示。

表 10.8 ISD2560/90/120P 的地址功能

	地址状态												功能状态
DIP 开关	1	2	3	4	5	6	7	8	9	10	11	12	(ON=0，OFF=1)
地址位	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	P/R	CE	(1 为高电平，0 为低电平，*为高或低电平)
	0	0	0	0	0	0	0	0	0	0			一段式最长 60 s 录放音，从首地址开始
	1	0	0	0	0	0	0	0	0	0			以十位二进制表示地址，每个地址代表 100 ms
地址模式	0	0	0	0	0	0	0	0	1	0			一段从 A6 地址开始的 12 s 录放音
	*	*	*	*	*	*	*	*	*	0			只要 A8、A9 有一位是 0，就处于地址模式
	0	0	0	1	0	0	0	0	1	1			循环放音操作，按住 CE 键不放循环放音第一段

续表

	地址状态										功能状态
操作模式	0	0	0	0	1	0	0	0	1	1	按顺序连续分段录放音，录音时压住 CE 键不放，放音时每触发一次 CE 键即放音一段，按 PD 键复位。每段语音长度不限
按钮模式	0	0	0	0	0	0	1	0	1	1	

注意： ISD2500 系列芯片的 ROM 地址中，地址 0~599 作为分段语音数据区，地址 600~767 未使用，地址 768~1023 作为工作模式的控制寄存器。

10.4.3　ISD2500 和单片机的接口及编程

1. 典型应用电路

ISD2500 可以构成单芯片应用系统，如图 10.25 所示，其中 S1 为【启动/暂停】键，S2 为【停止/恢复】键，S3 为【录放】键。系统所有的功能通过按键的输入后，通过 ISD2500 自动进行处理。

在很多场合下，由于语音数据的录/放需要由其他信号自动控制，因此单芯片的 ISD2500 系统并不能满足使用的要求，这时就必须采用微控制器的语音系统。

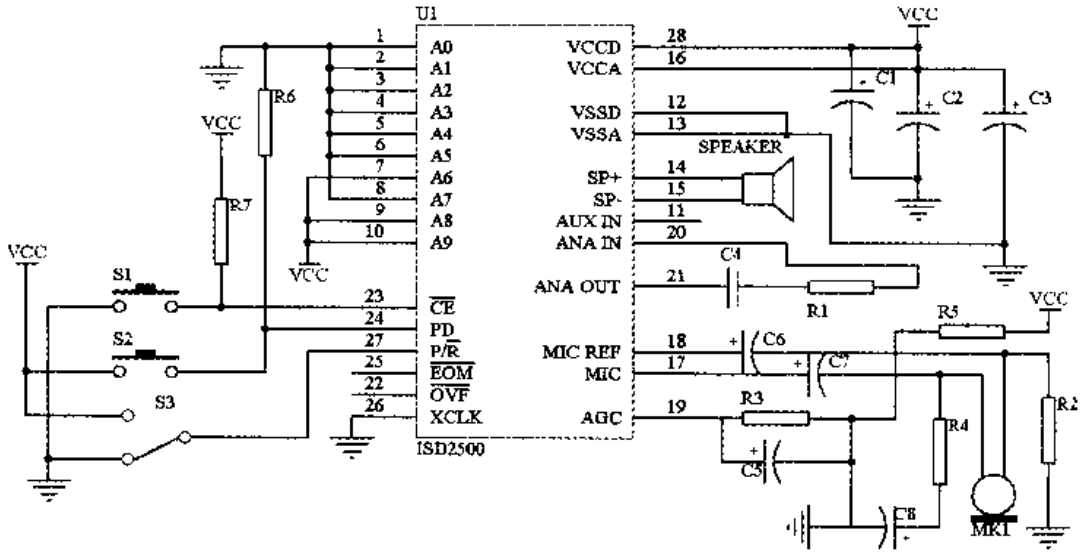


图 10.25　ISD2500 构成的单芯语音电路

2. 和微处理器配合使用电路

ISD2500 可以与各种微控制构成智能的语音系统。图 10.26 所示为 AT90LS8535 与 ISD2500 构成的一个简单电路。

其中 S1 为【录音】键，S2 为【播放】键，S3 为【段】操作键。

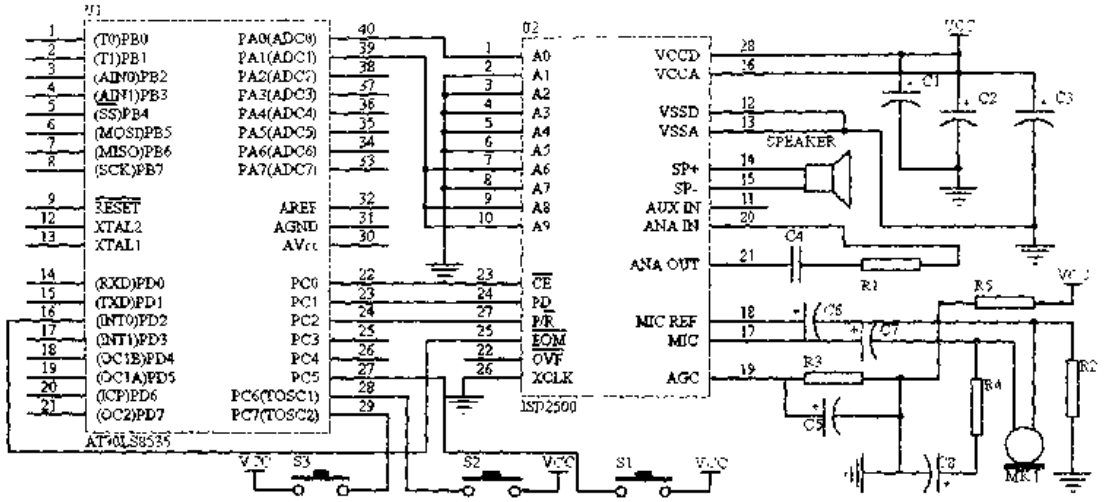


图 10.26 AT90LS8535 与 ISD2500 的接口电路

源程序：

```
#include "io8535.h"
#pragma interrupt_handler int0:2

//中断服务子程序
void int0(void)
{
    PORTC|=BIT(PC1);    //置 PD 为 1
    PORTC&=~BIT(PC0);   //停止
    PORTC|=BIT(PC0);

}
//播放子程序
void PLAY(void)
{
    PORTC&=~BIT(PC1);   //置 PD 为 0
    PORTC|=BIT(PC2);    //设置播放模式
    PORTC&=~BIT(PC0);   //播放
    PORTC|=BIT(PC0);
}
//录音子程序
void RECORD(void)
{
    PORTC&=~BIT(PC1);   //置 PD 为 0
    PORTC|=BIT(PC2);    //设置录音模式
    PORTC&=~BIT(PC0);   //录音
    PORTC|=BIT(PC0);
}

//跳转子程序
void NEXTMSG(void)
{
```

```
    PORTC|=BIT(PC1);     //停止播放
    PORTA|=BIT(PA1);     //设置 ISD2500 为按键模式
    PORTC&=~BIT(PC0);
    PORTC|=BIT(PC0);

    PORTA|=BIT(PA0);     //跳转至下一节
    PORTC&=~BIT(PC0);
    PORTC|=BIT(PC0);

    PORTA&=~BIT(PA1);    //设置 ISD2500 为正常模式
    PORTC&=~BIT(PC0);
    PORTC|=BIT(PC0);
}
//主程序
main(void)
{
    //初始化
    ......
    GIMSK=0x40;      //开中断
    MCUCR=0x02;
    PORTC|=BIT(PC0);     //置片选信号为高
    do{
        mid=PORTC&0xe0;
        switch(mid);
        {
            case 0x20:  //按键 1 按下,调用播放子程序
                PLAY();
                break;
            case 0x40:
                RECORD();    //按键 2 按下，调用录音子程序
                break;
            case 0x80:
                NEXTMSG();       //按键 3 按下，调用段子程序
                break;       }
    while(1);
}
```

10.5　TP-μP 微型打印机

TP-μP 系列打印机是单片机应用系统中的一种重要输出设备，它采用与标准打印机的接口兼容的接口电路，既具有传统微打的指令系统，又有 ESC/P 的标准打印控制命令。TP-μP 一般都采用 EPSON M150II、M160、M164、M180、M183 等型号的点阵式打印头，可打印出 16×16、24×24 点阵的符号和汉字。

10.5.1　TP-μP 打印机的接口和逻辑时序

TP-μP 打印机的并行口与 Centronic 并行接口兼容，也都采用以/STB、BUSY、ACK 和

8 位数据总线为特征的信号形式。这些信号的时序逻辑可以概括为：单片机先查询打印机的 BUSY 信号线的状态，如果 BUSY 为 0(打印机空闲)，则输出数据至数据总线，然后产生选通脉冲信号/STB，将数据总线上的数据存入打印机的缓冲区，在打印机处理数据期间，BUSY 信号为高，此时单片机不可再发送数据，打印机对数据的处理完毕后，BUSY 信号又变成低电平，并发送应答脉冲信号/ACK，表示又可以接收新的数据。信号的时序如图 10.27 所示。

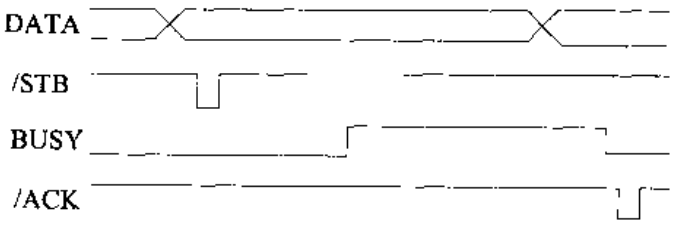


图 10.27 TP-μP 打印机的时序逻辑

10.5.2 P-μP 打印机的打印命令和字符代码

打印机的打印命令规定了打印机的功能，它们可以指导打印机的换行、回车、制表等操作；字符代码就是打印机能打印的字符的编码。由于打印机的打印命令格式对于不同的打印机略有不同，因此本书只对字符代码加以说明。

TP-μP 打印机内固化了 96 个 ASCII 字符和 352 个其他的字符。对于这些字符，TP-μP 只需要接收到相对应的字符代码就可以打印输出。表 10.9 是标准 ASCII 代码表。

表 10.9 ASCII 字符代码表

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

10.5.3 AT90LS8535 与 TP-μP 系列打印机的接口及编程

TP-μP 系列打印机与单片机相连时，既可以采用并行接口的方式，也可以采用串口的方式。TP-μP 的串行接口与 RS-232C 串行接口标准兼容，打印机采用+5V 供电，而机内则通过升压和降压电路实现 EIA RS-232C 所要求的电平。由于并行接口的编程简单，且 TP-μP 的打印机又都提供并行接口和串行接口，因此以下仅以 TP-μP 打印机的并行接口为例说明 AT90LS8535 与微打的连接及编程。图 10.28 所示为硬件电路图。

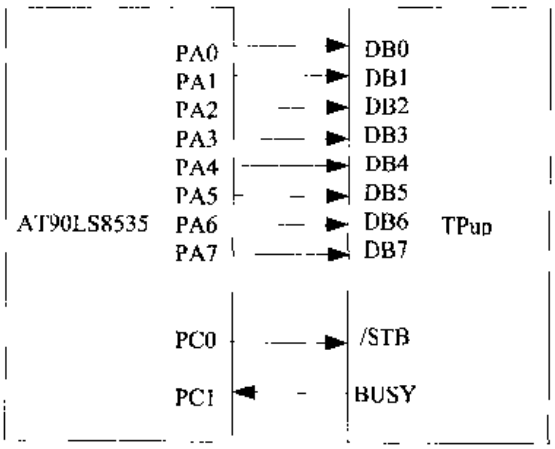


图 10.28　AT90LS8535 与 TP-μP 系列打印机的接口电路

由 TP-μP 系列打印机的时序逻辑图可知，单片机的程序流程应如图 10.29 所示。

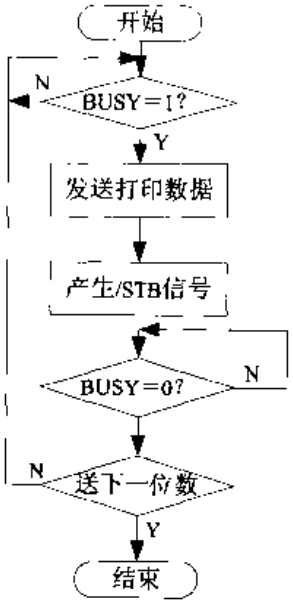


图 10.29　TP-μP 系列打印机的控制流程

下面是利用上述电路打印“hello world”字符串的源程序。

```
#include "io8535.h"
#define uchar unsigned char

//"hello world"的ASCII代码
const char printchar[11]={0x68,0x65,0x6c,0x6c,0x6f,0x20,
0x77,0x6f,0x72,0x6c,0x64};
//延时子程序
void delay(void)
{
    uchar i,j;
    for(i=1;i<100;i++)
    {
        for(j=1;j<200;j++)
```

```
        {
            ;
        }
    }
}
//打印一个ASCII字符
void tpprint(uchar x);
{
    uchar mid;
    mid=PORTC;
    mid=mid&0x02;
    while(mid==1)    //查询BUSY位
    {
        mid=PINC;
        mid=mid&0x02;
    }

    PORTA=x;
    PORTC&=~BIT(PC0);    //产生/STB脉冲
    delay();
    PORTC|=BIT(PC0);
    delay();    //延时等待/ACK信号有效
    delay();
    delay();
}

//主程序
void main(void)
{
    uchar i;
    DDRA=0xff;        //设置A口输出
    DDRC|=BIT(PC0); //设置/STB为输出
    DDRC&=~BIT(PC1);    //设置BUSY为输入
    for(i=0;i<11;i++)
        tpprint(printchar[i]);
}
```

10.6 IC卡

IC卡是携带应用信息和数据的媒体。在使用IC卡之前，必须对IC卡系统进行初始化编程，写入系统数据信息和个人的应用数据。所有这些操作都需要IC卡读写装置和相应的开发软件。

10.6.1 IC卡读写装置

IC卡的读写设备由IC卡连接器件、单片机、数据存储器、程序存储器、EEPROM和外围辅助电路组成。

1. IC 卡连接器件

IC 卡的连接器件用于连接 IC 卡和单片机系统。它提供 IC 卡与应用系统之间的接口电路，同时提供插卡检测、过流保护等辅助功能。

2. 单片机和存储器

IC 卡读写装置中的单片机运行存储器中的程序，执行对 IC 卡的读写，完成 PC 与 IC 卡之间的通信。

3. 外围辅助电路

IC 卡读写装置一般都通过标准并行或串行口与 PC 机联机工作，并利用 PC 机的键盘和显示器实现数据的输入输出。

10.6.2　IC 卡软件

IC 卡软件用于对应用系统中的 IC 卡进行规划设计，完成 IC 卡的个人化进程。这一过程通过对 IC 卡的读写编程实现。

对 IC 卡的读写操作在 IC 卡的读写器上完成，此时用户需与读写器进行交互式处理。首先，系统核对 IC 卡的厂商代码和卡型，如果正确，则在空白卡上写入发行商的代码，并确定此卡有效。然后应用系统应提供交互界面，让用户从键盘输入自己的用户密码。在密码写入后，系统在提供用户对密码核实和再修改的机会。确认无误后，软件发出熔断命令，熔断熔丝，完成 IC 卡的个人化进程。随后，系统再对 IC 卡的数据区写入初始化数据。

由 AT90LS8535 单片机构成的 IC 卡读卡器如图 10.30 所示。

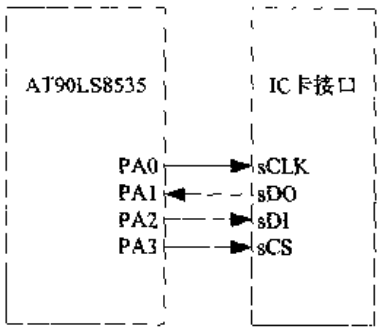


图 10.30　IC 卡读卡器

图中单片机通过 PA1 将数据按按位送出至 sD0 口，通过 PA2 口将数据按位从 sDI 输入，其中的 sCLK 和 sCS 分别接 PA0 和 PA3。

以下是 IC 卡的子程序。

```
sCLK 接 PA0
sDO 接 PA1
sDI 接 PA2
sCS 接 PA3
#define uchar unsigned char
#define uint  unsigned int
```

```
#define ulong unsigned long
//文件目录结构
typedef struct
{   uchar FileName[8];  // 文件名称
    uchar FileExt[3];       // 文件扩展名
    uchar FileAttr;          // 文件属性
    uchar a[10];               // 系统保留
    uint UpdateTime;
    uint UpdateDate;
    uint FirstPage;     // 文件首页号
    ulong FileSize;             // 文件字节数
} ICDirInfo;
//信息结构
typedef struct
{   uchar VersionName[8];  //版本名称
    uchar Version[3];        //版本号
    uchar ICName[10];       //IC卡名称
    ulong ICPages;
    uint PageBytes;
    uint DirStart;          //根文件目录开始页
    uint DirLength;          //根文件目录表页长度
    uint FatStart;
    uint FatLength;
    uint DataStart;
    uint DataLength;
} _ICSysInfo;
//文件分配表中页标记含义
    const uint code csFree = 0x00C0;          //未用的页
    const uint code csReserved = 0xfff0;      //系统保留的页
    const uint code csError = 0xfff7;          //坏页标记
    const uint code csFileEnd = 0xfff8;        //文件中最后的页
    const uint code csNextMin = 0x0003;      //下一页标记最小值
    const uint code csNextMax = 0xffef;       //下一页标记最大值
    const uint code csEnd = 0xffff;            //文件分配表结束
    const uint code SysInfoPage = 0;         //保存系统信息的页
    const uint code DirStartPage = 2;         //文件目录区首页
    ICSysInfo  ICSysInfo;
//AT45D041卡的操作
typedef enum
{   PageRead=0x52,        //直接读页，缓存不变
    Buf1Read=0x54,         //缓存1读
    Buf2Read=0x56,       //缓存2读
    PageToBuf1=0x53,         //页读至缓存1
    PageToBuf2=0x55,         //页读至缓存2
    PageCmpBuf1=0x60,    //页与缓存1比较，结果在状态寄存器第6位
    PageCmpBuf2=0x61,    //页与缓存2比较，结果在状态寄存器第6位
    Buf1Write=0x84,      //缓存1写
    Buf2Write=0x87,      //缓存2写
    Buf1ToPageWithErase=0x83,        //将缓存1写入页
    Buf2ToPageWithErase=0x86,      //将缓存2写入页
    Buf1ToPageWithoutErase=0x88,  //将缓存1写入页
```

```
    Buf2ToPageWithoutErase=0x89,  //将缓存 2 写入页
    PageWriteThroughBuf1=0x82,    //数据先写入缓存 1，再写入页
    PageWriteThroughBuf2=0x85,    //数据先写入缓存 2，再写入页
    PageRewriteThroughBuf1=0x58,  //页读入缓存 1，然后缓存 1 写入页
    PageRewriteThroughBuf2=0x59,  //页读入缓存 2，然后缓存 2 写入页
    StatusRead=0x57                 //读状态寄存器
} OpCode;

uchar FileExists(uchar *FileName)
{   uchar i=1;
    char c;
    ICDirInfo tempDir;
    uchar *p=(uchar*)(&tempDir);
    do
    {   ICReadDirInfo(i,&tempDir);
        c=strncmp(p,FileName,sizeof(tempDir.FileName));
        if(c==0)
            return(i);
        else
            i++;
        }while(i<128);
    return (0);
}
//创建文件
uchar FileCreate(uchar *FileName)
{   uchar i ;
    ICDirInfo tempDir;
    uchar *p=(uchar*)(&tempDir);
    i=FileExists(FileName);
    if(i!=0)
    {   FileDelete(FileName);
    }
    do
    {   ICReadDirInfo(i,&tempDir);
        if(*p == 0x00)
        {   ICWriteDirInfo(i,&tempDir);
            return(i);
        }
    } while(i<128);
    return (0);
}
//打开文件
uchar FileOpen(uchar *FileName)
{   return(FileExists(FileName));
}
//关闭文件
void FileClose(uchar *Handle)
{   *Handle=0x00;
}
//读文件
int FileRead(uchar Handle, uchar *Buffer, int Start, int Count)
```

```
{   int iResult=0;
    uint iTemp,curPage,oldPage=0,numPage=0;
    ulong filelen;
    ICDirInfo tempDir;
    if (Handle==0x00)   //判断文件句柄是否有效
        return(-1);
    ICReadDirInfo(Handle,&tempDir);//读取文件目录信息
    filelen=tempDir.FileSize;
    curPage=tempDir.FirstPage;
    while(Count>0 && Start<filelen)//读取文件字节数据
    {   do
        {   iTemp=(numPage+1)*ICSysInfo.PageBytes;
            if(Start>iTemp)
            {   curPage=ICReadPageSign(curPage);
                numPage++;
            }
            else
                break;
        } while(1);
        if (oldPage!=curPage)   //判断是否在同一页
        {
    *Buffer++=ICReadPageByte(curPage,(Start%ICSysInfo.PageBytes));
            oldPage = curPage;
        }
        else
        {   *Buffer++ = ICReadNextByte();
        }
        Start++;
        Count--;
        iResult++;      //读取字节数加1
        }
    return(iResult);
}
//写文件
int FileWrite(uchar Handle, char *Buffer, int Start, int Count)
{   int iResult=0;
    uint iTemp,curPage,oldPage=0,numPage=0;
    ulong filelen;
    ICDirInfo tempDir;
    if (Handle==0x00)   //判断文件句柄是否有效
        return(-1);
    ICReadDirInfo(Handle,&tempDir);//读取文件目录信息
    filelen=tempDir.FileSize;
    curPage=tempDir.FirstPage;
    while(Count>0)  //读取文件字节数据
    {   do
        {   iTemp=(numPage+1)*ICSysInfo.PageBytes;
            if(Start>iTemp)
            {   curPage=ICReadPageSign(curPage);
                numPage++;
            }
```

```
            else
            {   break;
            }
        }while(1);
        if(oldPage!=curPage)    //判断是否在同一页
        {   if(oldPage!=0)
            {   ICBuf1ToPage(oldPage);
            }
            ICPageToBuf1(curPage);
            ICWriteBuf1Byte(Start%ICSysInfo.PageBytes,*Buffer++);
            oldPage=curPage;
        }
        else
        {   ICWriteNextByte(*Buffer++);
        }
        Start++;
        Count--;
        iResult++;      //读取字节数加 1
        }
    return (iResult);
}
//删除文件
uchar FileDelete(uchar* FileName)
{   uchar i;
    ICDirInfo tempDir;
    uchar *p=(uchar *)(&tempDir);
    uint w,w1;
    i=FileExists(FileName);
    if(i!=0)
    {   ICReadDirInfo(i,&tempDir);
        w=tempDir.FirstPage;
        w1=ICReadPageSign(w);
        while(w1!=csFileEnd)
        {   ICWritePageSign(w,csFree);
            w=w1;
            w1=ICReadPageSign(w);
        }
        ICWritePageSign(w,csFree);
        ICWritePageSign(w1,csFree);
        *p=0x00;
        ICWriteDirInfo(i,&tempDir);
    }
    return (1);
}
//格式化 IC 卡
ucharICFormat()
{   uchar b;
    ICWriteSysInfo(&ICSysInfo,SysInfoPage);     //写入系统信息
    ICDeleteAll();
    return(b);
}
```

```
//清除 IC 卡数据
uchar ICDeleteAll()
{   uchar b=1;
    uint i,j;
//初始化写入文件目录表
    for(i=ICSysInfo.DirStart;i<=ICSysInfo.DirLength;i++)
        {   ICWriteBuf1Byte(i,0);
            for(j=1;j<ICSysInfo.PageBytes;j++)
                ICWriteNextByte(0);
            ICBuf1ToPage(i);
            ICWaitReady();
            if (!ICBuf1CmpPage(i))
            {   b=0;
                break;
            }
        }
    if(!b) return(b);
//初始化写入文件分配表
    for(i=ICSysInfo.FatStart;i<=ICSysInfo.FatLength;i++)
        {   ICWriteBuf1Byte(i,0);
            for(j=1;j<ICSysInfo.PageBytes;j++)
                ICWriteNextByte(0);
            ICBuf1ToPage(i);
            ICWaitReady();
            if (!ICBuf1CmpPage(i))
            {   b = 0;
                break;
            }
        }
    return (b);
}
//获取可用空间字节数
ulong ICFreeSize()
{   uint CurPage=0;
    ulong lResult=0;
    do
{   CurPage=ICNextFreePage(CurPage);
        if (CurPage>0)
            lResult++;
        else
            break;
    }while(1);
    lResult *=ICSysInfo.PageBytes;
    return(lResult);
}
//获取已使用空间字节数
ulong ICUsedSize()
{   ulong lResult;
    lResult=ICTotalSize()-ICFreeSize();
    return(lResult);
}
```

```
//获取总空间字节数
ulong ICTotalSize()
{   ulong lResult;
    lResult=ICSysInfo.DataLength;
    lResult *=ICSysInfo.PageBytes;
    return(lResult);
}
//读取目录信息
void ICReadDirInfo(uchar IndexOfDir, ICDirInfo *DirInfo)
{   uchar i;
    uchar *p;
    uint Page,ByteAddr;
    ICDirIndexToPageByteAddr(IndexOfDir,&Page,&ByteAddr);
    p=(uchar *)DirInfo;
    *p++=ICReadPageByte(Page,ByteAddr);
    for(i=1;i<sizeof(ICDirInfo);i--)
        *p++=ICReadNextByte();
}
//写入目录信息
uchar ICWriteDirInfo(uchar IndexOfDir, ICDirInfo *DirInfo)
{   uchar b;
    uint Page,ByteAddr;
    uchar *p = (uchar*) DirInfo;
    ICDirIndexToPageByteAddr(IndexOfDir,&Page,&ByteAddr);
    b=ICWriteToPage(p,(uchar)sizeof(ICDirInfo),Page,ByteAddr);
    return(b);
}
//读取 IC 卡系统信息
void ICReadSysInfo(_ICSysInfo *ICInfo,uint SysPage)
{   uchar i;
    uchar *p=(uchar *)ICInfo;
    *p++=ICReadPageByte(SysPage,0);
    for (i=1;i<sizeof(ICSysInfo);i++)
        *p++=ICReadNextByte();
}
//写入 IC 卡系统信息
uchar ICWriteSysInfo(_ICSysInfo *ICInfo, uint SysPage)
{   uchar b;
    uchar *p=(uchar *)ICInfo;
    b=ICWriteToPage(p,(uchar)sizeof(_ICSysInfo),SysPage,0);
    return(b);
}
//读页标记
uint  ICReadPageSign(uint Page)
{   uint wPage,wByteAddr,wResult,i;
    i=ICSysInfo.PageBytes/sizeof(Page);
    wPage=(Page/i)+ICSysInfo.FatStart;
    wByteAddr=(Page%i)*sizeof(Page);
    i=ICReadPageByte(wPage,wByteAddr);
    wResult=i;
    wResult=wResult<<8;
```

```
    i=ICReadNextByte();
    wResult+=i;
    return(wResult);
}
//写页标记
uchar  ICWritePageSign(uint Page,uint Sign)
{   uchar b;
    uint wPage,wByteAddr,i;
    i=ICSysInfo.PageBytes / sizecf(Page);
    wPage=(Page/i)+ICSysInfo.FatStart;
    wByteAddr=(Page%i)*sizeof(Page);
    ICPageToBuf2(wPage);
    ICWaitReady();
    ICWriteBuf2Byte(wByteAddr,(uchar)(Sign>>8));
    ICWriteNextByte((uchar)(Sign));
    ICBuf2ToPage(wPage);
    ICWaitReady();
    b=ICBuf2CmpPage(wPage);
    return (b);
}
//寻找第一个空闲的页
uint  ICFirstFreePage()
{   return (ICNextFreePage,0);
}
//寻找下一个空闲的页
uint  ICNextFreePage(uint StartPage) /**/
{   uint w;
    do
    {   w=ICReadPageSign(StartPage);
        if(w==csFree)
            break;
        if(w==csEnd)
        {   w=0x0000;
            break;
        }
        StartPage++;
        }while(1);
    return(w);
}

void ICDirIndexToPageByteAddr(uchar DirIndex, uint *Page, uint *ByteAddr)
{
    uchar i;
    i=ICSysInfo.PageBytes/sizeof(ICDirInfo);
    *Page=(DirIndex/i)+ICSysInfo.DirStart;
    *ByteAddr=(DirIndex%i)*sizeof(ICDirInfo);
}

uchar ICWriteToPage(uchar*p, uchar Count, uint Page, uint ByteAddr)
{   uchar b;
```

```
    uchar i;
    ICPageToBuf2(Page);
    ICWaitReady();
    ICWriteBuf2Byte(ByteAddr, *p++);          //写入字符串至缓存 2
    for(i=1;i<Count;i++)
        ICWriteNextByte(*p++);
    ICBuf2ToPage(Page);      //缓存 2 写入页
    ICWaitReady();
    b=ICBuf2CmpPage(Page);       //比较写入是否成功
    return (b);
}

uchar ICReadPageByte(uint Page, uint ByteAddr)
{   uchar c;
    c=ICCommand(PageRead,Page,ByteAddr);
    return(c);
}

void ICWriteBuf1Byte(uint ByteAddr,uchar b)
{   ICCommand(Buf1Write,0,ByteAddr);
    ICWriteNextByte(b);
}

void ICWriteBuf2Byte(uint ByteAddr,uchar b)
{   ICCommand(Buf2Write,0,ByteAddr);
    ICWriteNextByte(b);
}

uchar ICReadBuf1Byte(uint ByteAddr)
{   uchar c;
    c=ICCommand(Buf1Read,0,ByteAddr);
    return(c);
}

uchar ICReadBuf2Byte(uint ByteAddr)
{   uchar c;
    c=ICCommand(Buf2Read,0,ByteAddr);
    return(c);
}

uchar ICBuf1CmpPage(uint Page)
{   uchar b;
    uchar c;
    ICCommand(PageCmpBuf1,Page,0);
    ICWaitReady();
    c=ICStatus();
    b=c&0x40;
    return(b);
}
```

```
uchar ICBuf2CmpPage(uint Page)
{   uchar b;
    uchar c;
    ICCommand(PageCmpBuf2,Page,0);
    ICWaitReady();
    c=ICStatus();
    b=c&0x40;
    return(b);
}

void  ICBuf1ToPage(uint Page)
{   ICCommand(Buf1ToPageWithErase,Page,0);
}

void  ICBuf2ToPage(uint Page)
{   ICCommand(Buf2ToPageWithErase,Page,0);
}

void  ICPageToBuf1(uint Page)
{   ICCommand(PageToBuf1,Page,0);
}

void  ICPageToBuf2(uint Page)
{   ICCommand(PageToBuf2,Page,0);
}

//等待IC卡准备好
void  ICWaitReady()
{   uchar c;
    while (1)
    {   c=ICStatus();
        if (c&0x80)
            break;
    }
}

//读IC卡状态
uchar ICStatus()
{   uchar ucResult;
    ucResult=ICCommand(StatusRead,0,0);
    return(ucResult);
}

//IC卡命令选择执行
unsigned char ICCommand(OpCode cmd, uint Page, uint ByteAddr)
{   unsigned char ucResult;
    ICOutOpCode(cmd);
    if(cmd!=StatusRead)
    {   ICOutXBits(4);
        ICOutPage(Page);
```

```
        ICOutByteAddr(ByteAddr);
    }
    switch (cmd)
    {   case PageRead:              //直接读页，缓存不变
        {   ICOutXBits(32);
            ucResult=ICReadNextByte();
            break;
        }
        case Buf1Read:              //缓存 1 读
        case Buf2Read:               //缓存 2 读
        {   ICOutXBits(8);
            ucResult=ICReadNextByte();
            break;
        }
        case PageToBuf1:            //页读至缓存 1
        case PageToBuf2:            //页读至缓存 2
        {   ICOutXBits(8);
            break;
        }
        case StatusRead:            //读状态寄存器
            ucResult = ICReadNextByte();
            break;
        default : break;
        }
    return ucResult;
}
//连续写下一个字节
void  ICWriteNextByte(uchar b)
{   uchar i,mid1,mid2;
    mid1=b;
    for(i=0;i<8;i++)
    {   mid2=mid1&0x80;
        if(mid2==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        mid1=mid1<<1;
        PORTA&=~BIT(PA0);  PORTA|=BIT(PA0);
    }
}
//连续读下一个字节
uchar ICReadNextByte()
{   uchar i,mid1;
    for(i=0;i<8;i++)
    {   PORTA&=~BIT(PA0);  PORTA|=BIT(PA0);
        mid=(PORTA&0x02)<<7;
        mid>>1;
    }
    return (mid);
}
```

```
//发送命令字
void  ICOutOpCode(OpCode cmd)
{   uchar b=(uchar)cmd;
    ICWriteNextByte(b);
}

//发送页面地址
void  ICOutPage(uint Page)
{   uchar i,mid1,mid2;
    mid1=(uchar)(Page>>8);
    mid2=mid1&0x04;
    for(i=0;i<3;i++)
    {   if(mid2==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        mid1=mid1<<1;
        PORTA&=~BIT(PA0);  PORTA|=BIT(PA0);
    }
    mid1=(uchar)(Page%0xff);
    for(i=0;i<8;i++)
    {   mid2=mid1&0x80;
        if(mid2==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        mid1=mid1<<1;
        PORTA&=~BIT(PA0);  PORTA|=BIT(PA0);
    }
}
//发送字节地址
void  ICOutByteAddr(uint ByteAddr)
{   uchar i,mid1,mid2;
    mid1=(uchar)(ByteAddr>>8);
    mid2=mid1&0x01;
    if(mid2==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
    mid1=(uchar)(ByteAddr%0xff);
    PORTA&=~BIT(PA0);  PORTA|=BIT(PA0);
    for(i=0;i<8;i++)
    {   mid2=mid1&0x80;
        if(mid2==0)
            PORTA&=~BIT(PA2);
        else
            PORTA|=BIT(PA2);
        mid1=mid1<<1;
        PORTA&=~BIT(PA0);  PORTA|=BIT(PA0);
    }
```

```
}
//发送 X 个任意字符
void  ICOutXBits(uchar X)
{   PORTA&=~BIT(PA2)
    while (X>0)
    {   PORTA&=~BIT(PA0);
        PORTA|=BIT(PA0);
        N--;
    }
}
```

第 11 章　AT90LS8535 的外围扩展

在很多应用系统中，单片机的自身资源并不能满足应用系统的要求，这时就必须要利用单片机已有的 I/O 资源进行系统扩展，以扩大其应用范围，增强其使用功能。

单片机的外围扩展包括数字 I/O 口的扩展、数模转换器的扩展、定时/计数器的扩展以及特定功能芯片的扩展。这些扩展都拓宽了单片机的应用范围，并使得以单片机为基础的应用系统兼顾了通用性和灵活性的特点。

本章将着重讲述了单片机外围扩展芯片的功能特点和接口编程，并用实例的方法使读者掌握外围芯片扩展中的接口编程。

11.1　简单 I/O 扩展芯片

在单片机应用系统中，由于很多 I/O 口都采用分时使用的方式，因此在构成输出口时，接口芯片必须具有锁存功能；构成输入口时，还应根据数据的有效时间，确定接口芯片应该采用三态缓冲方式还是锁存选通方式。

11.1.1　用 74LS377 扩展数据输出接口

74LS377 为 8 位的 D 触发器，它有 8 个数据输入端(D)、8 个数据输出端(Q)、一个锁存允许输入端和一个时钟信号输入端。74LS377 的功能如表 11.1 所示。

表 11.1　74LS377 功能说明

输入			输出
/E	CLK	D	Q
H	–	·	Q0
L	↑	H	H
L	↑	L	L
–	L	–	Q0

图 11.1 为 74LS377 与 AT90LS8535 单片机的接口电路，74LS377 的 D0~D7 和单片机的 PA0~PA7 相连，/E 和 PB0 相连，CLK 和 PB1 相连。当 PB0 输出低电平时，PB1 的一个上升沿脉冲将 PA 口输出的数据锁存至 373 的数据输出口。

AT90LS8535 的执行程序如下：

```
#include "io8535.h"
#define uchar unsigned char
```

```
//数据发送子程序
void Senddata(uchar x)
{
    PORTB&=~BIT(PB0);    //置/G 为低电平
    PORTA=x;             //输出 x 至 PA 端口
    PORTB|=BIT(PB1);     //产生一个正脉冲信号
    PORTB&=~BIT(PB1);
    PORTB|=BIT(PB0);     //置/G 为高电平
}
//主程序
void main()
{
    DDRA=0xff;           //初始化 A、B 端口
    DDRB=0xff;
    PORTB|=BIT(PB0);
    PORTB&=~BIT(PB1);
    Senddata(0x55);
}
```

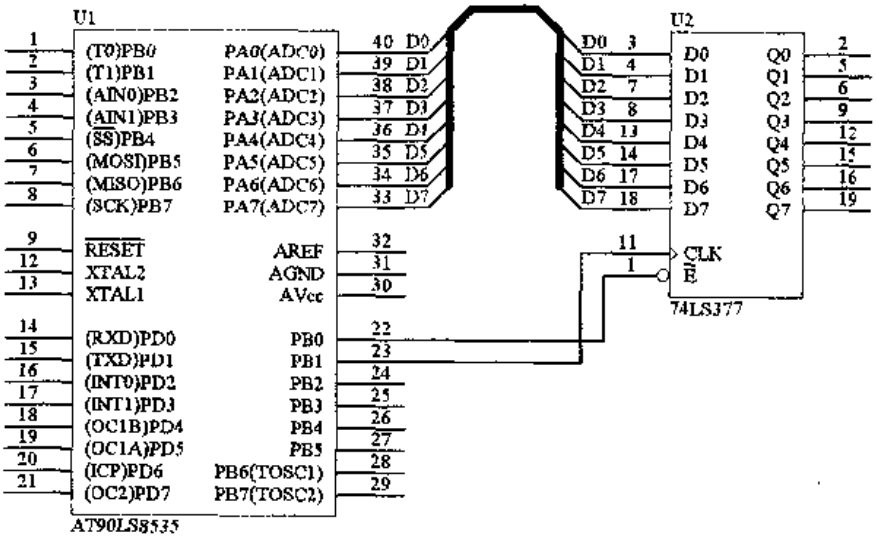


图 11.1　74LS377 与 AT90LS8535 的接口电路

11.1.2　数据输入接口

在单片机应用系统中，数据的输入接口主要有两种，即用缓冲器构成的输入接口和用锁存器构成的输入接口。

1. 用缓冲器作为输入接口扩展

缓冲器的作用就是当输入设备被选通时，使外设输入数据直接同系统的数据总线相连；而当外设处于非选通状态时，把数据源和数据总线隔离。这种总线扩展方式适合与常态数据的输入扩展。

74LS244 的芯片内部含有 2 组 4 位的三态门电路，/G1 和/G2 分别为各组三态门的选通

信号。74LS244 的功能如表 11.2 所示。

表 11.2 74LS244 的功能说明

输入		输出
/G	A	Y
L	L	L
L	H	H
H	-	高阻

使用 74LS244 进行输入口扩展时，其连接方式如图 11.2 所示。AT90LS8535 的 PA0～PA3 接 74LS244 的 1Y1～1Y4，PA4～PA7 接 2Y1～2Y4，PB0 接 1/G，PB1 接 2/G。

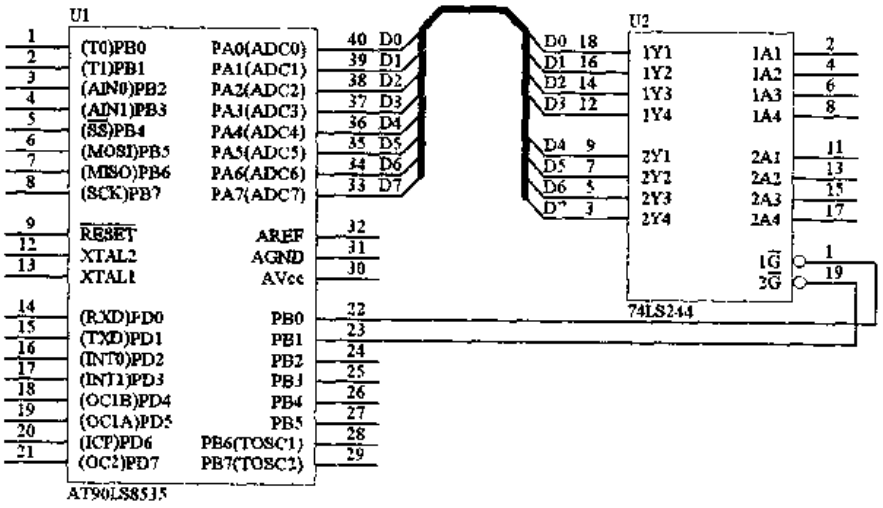


图 11.2 74LS244 与 AT90LS8535 的接口电路

利用 74LS244 输入数据的源程序如下：

```
#include "io8535.h"
#define uchar unsigned char
//接收数据
uchar Receivedata(void)
{
    uchar mid;
    PORTB&=~BIT(PB0);   //置 1/G 为低电平
    PORTB&=~BIT(PB1);   //置 2/G 为低电平
    mid=PINA;           //数据输入
    return  mid;
}
//主程序
void main()
{
    uchar data;
    DDRA=0;          //初始化 A、B 端口
    DDRB=0xff;
```

```
    PORTB|=BIT(PB0);
    PORTB|=BIT(PB1);
    data=Receivedata();
}
```

2. 用锁存器扩展并行输入口

当外设的数据变化很快时，使用缓冲器构成的数据输入通道可能会错误地输入数据，这时就需要采用具有数据锁存功能的数据输入接口。74LS373 是一个带三态门的 8D 触发器，它具有一个输出允许输入端(/OE)、一个锁存信号端(LE)、8 个数据输入端(D)和 8 个数据输出端(Q)。74LS373 的功能如表 11.3 所示。

表 11.3　74LS373 的功能说明

输入			输出
/OE	LE	D	Q
L	H	H	H
L	H	L	L
L	L	-	Q0
H	-	-	高阻

AT90LS8535 同 74LS373 的接口电路如图 11.3 所示。

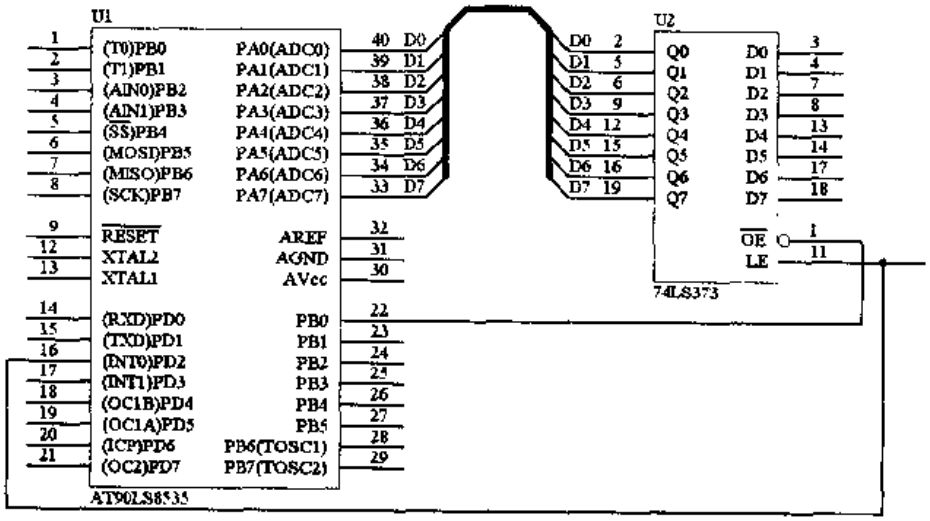


图 11.3　74LS373 与 AT90LS8535 的接口电路

该接口电路的原理是：当外设将数据准备好以后，发出一个控制信号到 74LS373 的锁存信号输入端(LE)以将数据锁存输出至 373 的 Q 端。与此同时外设发出的控制信号在单片机上产生中断，单片机在中断服务程序中，实现数据的输入。

数据输入的源程序为：

```
#include "io8535.h"
#define uchar unsigned char
```

```
#pragma interrupt_handler Receivedata:2
uchar data;
//中断服务子程序
void  Receivedata(void)
{
    uchar mid;
    PORTB&=~BIT(PB0);    //置/OE 为低电平
    mid=PINA;            //数据输入
    PORTB|=BIT(PB0);     //置/OE 为高电平
    return  mid;
}
//主程序
void main()
{
    GIMSK=0x40;
    MCUCR=0x02;
    DDRA=0;          //初始化 A、B 端口
    DDRB=0xff;
    PORTB|=BIT(PB0);
}
```

11.2 模拟量输出

在单片机系统中，经常要用到模拟量输出，数模转换器就是一种将数字信号转换成模拟信号的器件。本节将首先介绍一下数模转换器的基本原理和主要技术指标，然后对数模转换器件——DAC0832 的结构和接口编程进行详细的介绍。

11.2.1 D/A 转换器简介

D/A 转换器用来将数字量转换成模拟量，它们通过接口电路同微处理器相连，可以实现计算机系统的模拟量输出。

1. D/A 转换器的原理

D/A 转换器要实现的基本要求就是输出电压 Vout 和输入数字量 Vdigital 的正比关系，即：

Vout=Vdigital×Vbit

其中：$Vdigital=Dn\times2^{n-1}+Dn\text{-}1\times2^{n-2}+\cdots+D0\times2^{0}$

Vbit 为一位数字量所代表的电压。

由此可见，D/A 转换器就是把输入数字量中每位都按其权值分别转换成模拟量，然后通过运算放大器求和相加，最后得到模拟输出量。因此，D/A 转换器内部必须要有一个求和网络，以实现按权值分别进行模拟量的转换。

数模转换器的类型很多，其中最为常用的就是采用 T 型求和网络的数模转换器。这种电路包括电阻网络、模拟电子开关和求和放大器构成。模拟电子开关受数字量控制，当数

字量为 1 时，模拟开关接到内部电源上，当数字量为 0 时，模拟开关则接地。如图 11.4 所示，现以 4 位 D/A 转换器为例说明该 T 型网络的转换过程。

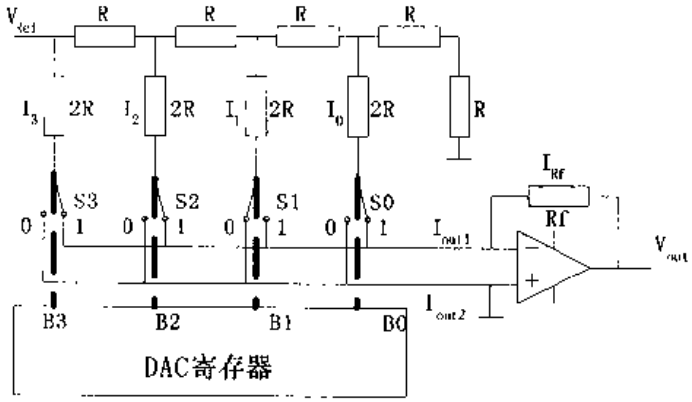


图 11.4　DAC 的 T 型网络转化原理

设 V_{ref}=-10 V，4 位 DAC 寄存器的值为 1101，则：

$I_3=V_{ref}/2R$，$I_2=I_3/2$ ，$I_1=I_3/4$ ，$I_0=I_3/8$，

$I_{out1}=I_0+I_2+I_3$

$=Vref\times(1/8R+1/2R+1/R)$

$=13I_3/8$

因为 $I_{Rf}=-I_{out}$，若 Rf=R，则

$Vout=I_{Rf}\times R=8.125V$

从而实现数模转换器的数字量到模拟量的转换要求。

2. D/A 转换器的性能指标

DAC 的性能指标是选用 DAC 芯片类型的依据，也是衡量芯片质量的重要参数。DAC 的性能指标主要主要有以下 5 个：

- 分辨率

  分辨率是指 DA 转换器的最小输出模拟电压增量与最大输出模拟电压的比值，它取决于 DA 转换器的位数。一个 n 位的 DA 转换器的分辨率就等于 2^{-n}，例如：10 位的 DAC 芯片的分辨率为 2^{-10}。

- 转换精度

  转换精度是指满量程时 DAC 的实际模拟输出值和理论值的接近程度。它实际上包含了非线性误差、温度漂移误差等综合误差。它与分辨率的不同在于：分辨率是指最低一位的数字量所能引起的转换结果的变化；而转换精度则是表示实际转换值与理论值的接近程度。

- 建立时间

  建立时间是描述 DAC 转换速度的一个指标，它是指从输入数字量变化到输出达到终值误差所需的时间。对于理想的数模转换器来说，其输入的数字量变化时，它输出的模拟量也会立即发生变化，即建立时间为 0。但是在实际应用的 DAC 中，由于电路中的电容、电感等效应会引起转换的延时。

- 线性度
  线性度是描述 DAC 的实际转换特性曲线和理想直线之间的最大偏差指标。
- 输出电压范围
  不同型号的 DAC 具有不同的输出电压范围。对于这个应指标实际需要的电压范围予以选择。

11.2.2 8 位数模转换器 DAC0832

DAC0832 是 8 位分辨率的数模转换芯片，它具有两个输入数据缓冲器，能直接同微处理器相连，其特性参数如下：

- 8 位分辨率。
- 单缓冲、双缓冲或直接数字量输入。
- 电流稳定时间 1μs。
- 单一电源供电。

1. DA0832 的引脚及内部结构

图 11.5 为 DAC0832 的内部逻辑结构。

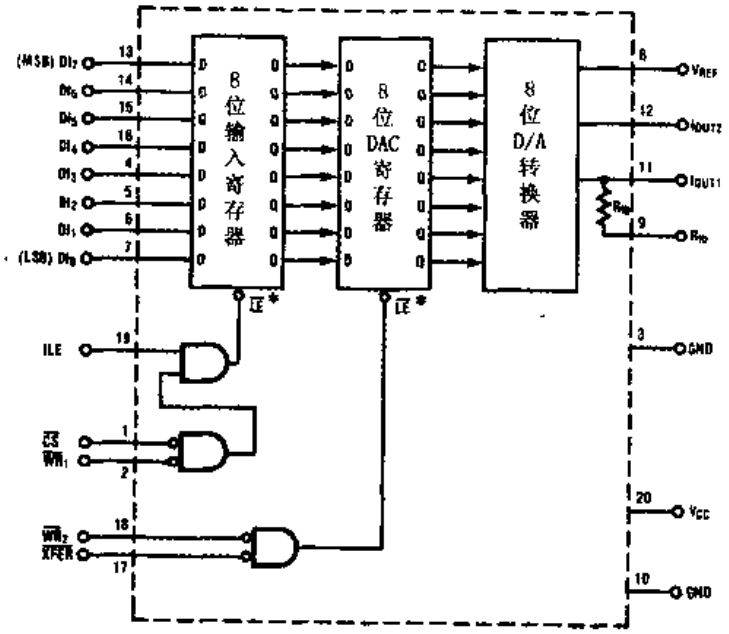


图 11.5 DAC 的内部逻辑结构

如图所示，DAC0832 的内部由 8 位输入锁存器、8 位 DAC 寄存器和 8 位 D/A 转换器 3 部分组成。其中，8 位输入锁存器用于锁存微控制器送来的数字量，使输入数字量得到缓冲和锁存，由/LE(1)控制；8 位 DAC 寄存器用于存放待转换的数字量，由/LE(2)控制；8 位 D/A 转换器则由 8 位 T 型求和网络组成，它能输出与数字量成正比的模拟电流，因此 DAC0832 必须要外接运算放大器才能得到模拟电压输出。

DAC0832 为 20 引脚的双列直插式封装形式，其引脚排列如图 11.6 所示。

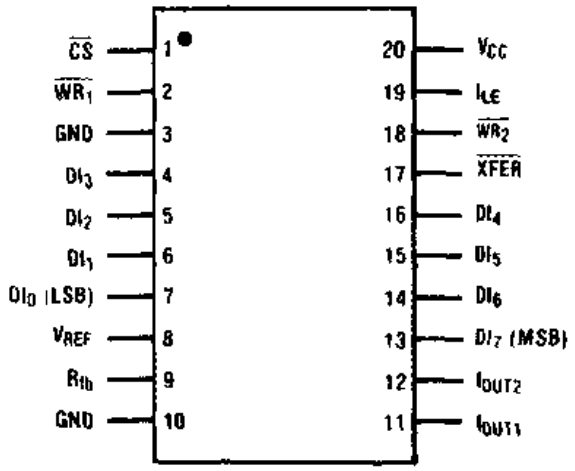


图 11.6　DAC0832 的引脚

各引脚功能如下：

- /CS：输入寄存器选择信号。
- /WR1：输入寄存器的写信号。
- /WR2：DAC 寄存器写信号。
- Vref：参考电压输入。
- Rfb：反馈信号输入。
- /XFER：数据传送信号
- Iout1、Iout2：模拟电流输出。
- ILE：允许数字量输入信号。
- DI0~DI7：数据总线。
- Vcc、GND：电源输入。

11.2.3　8 位数模转换器与单片机的接口及编程

DAC0832 与单片机有两种连接方式，即单缓冲器方式和双缓冲器方式。

1. 单缓冲器方式

使 DAC0832 的两个输入寄存器中的一个处于直通方式时，DAC0832 就工作在单缓冲器方式。这种方式一般应用在只有一路模拟两路输出的应用系统中。

单缓冲器方式的接口形式如图 11.7 所示。

如图，由于 DAC0832 的/WR2=0，/XFER＝0，因此数模转换器的 DAC 寄存器处于直通方式，DAC0832 为单缓冲器方式。

以下为利用 DAC0832 输出锯齿波的源程序：

```
#include "io8535.h"
#define uchar unsigned char

//主程序
void main()
```

```
{
    uchar i;
    DDRA=0xff;              //初始化 A、B 端口
    DDRB=0xff;
    PORTB|=BIT(PB0);
    PORTB|=BIT(PB1);
    do
    {
        PORTB&=~BIT(PB0);   //置/CS 为低电平
        PORTA=i;            //输出 DA 转换数值
        PORTB&=~BIT(PB1);   //产生写脉冲信号
        PORTB|=BIT(PB1);
        PORTB|=BIT(PB0);    //置/CS 为高电平
        i++;
    }while(1);
}
```

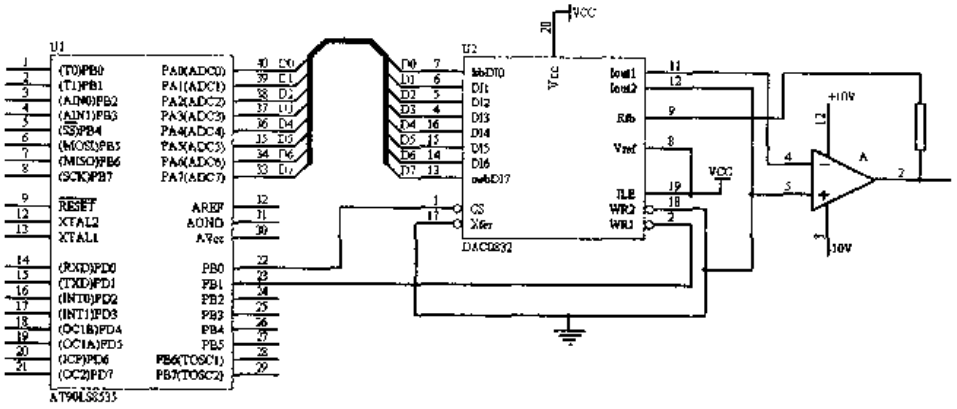


图 11.7 DAC0832 的单缓冲器模式

2. 双缓冲器方式

使 DAC0832 的两个锁存器都接成受控锁存方式时，DAC0832 就工作在双缓冲器方式。这种方式多用于多路数模转换系统中，以实现多路模拟信号的同步输出。

双缓冲器方式的接口形式如图 11.8 所示。

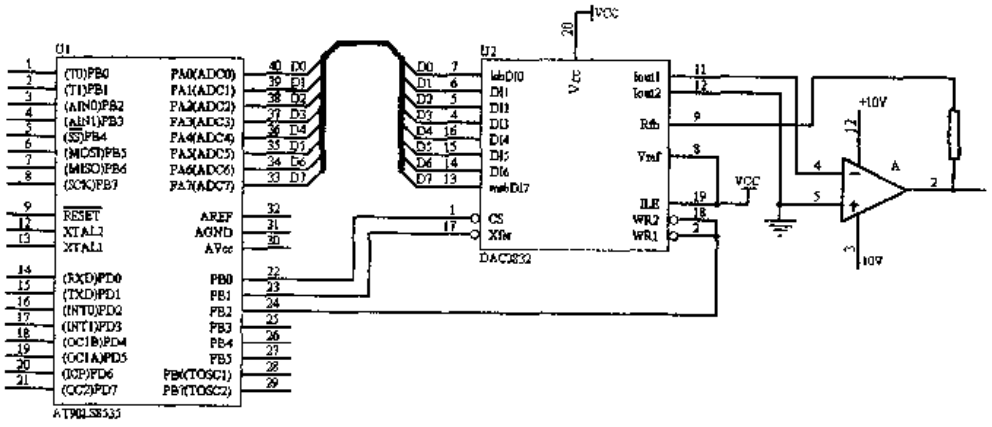


图 11.8 DAC0832 的双缓冲器模式

如图，为了实现对两个锁存器的控制，单片机采用了 PB0 和 PB1 两个端口分别连接 DAC0832 的输入寄存器和 DAC 寄存器的控制端口/CS 和/XFER。由于每个进入数模转换器的数据都要经过两次锁存，因此在单片机的程序中必须运行两个步骤才能完成一次数模转换。

以下为采用双缓冲方式的 DAC0832 输出锯齿波的源程序：

```
#include "io8535.h"
#define uchar unsigned char

//主程序
void main()
{
uchar i;
    DDRA=0xff;          //初始化 A、B 端口
    DDRB=0xff;
    PORTB|=BIT(PB0);
    PORTB|=BIT(PB1);
    PORTB|=BIT(PB2);
    //循环向 DAC 发送数据
do
    {
        PORTB&=~BIT(PB0);   //置/CS 为低电平
        PORTA=i;            //输出 DA 转换数值
        PORTB&=~BIT(PB2);   //产生写脉冲信号
        PORTB|=BIT(PB2);
        PORTB|=BIT(PB0);    //置/CS 为高电平
        PORTB&=~BIT(PB1);   //置/XFER 为低电平
        PORTB&=~BIT(PB2);   //产生写脉冲信号
        PORTB|=BIT(PB2);
        PORTB|=BIT(PB1);    //置/XFER 为高电平
        i++;
    }while(1);
}
```

11.2.4　12 位数模转换器 DAC1230

DAC1230 是与微处理器兼容的 12 位数模(D/A)转换芯片，这个系列的类似产品还有 DAC1231 和 DAC1232。其特性参数如下：

- 12 位分辨率。
- 8 位数据总线接口。DAC1230 系列的低 4 位数据线在片内和高 4 位数据线相连，在片外表现为 8 位数据线的接口形式。
- 电流稳定时间 1 μs。
- 工作电压范围为+5 V～+15 V。
- 参考电压 Vref=−10 V～+10 V。

1. DA1230 的引脚及内部结构

DAC1230 为 20 引脚的双列直插式封装形式，其引脚排列如图 11.9 所示。

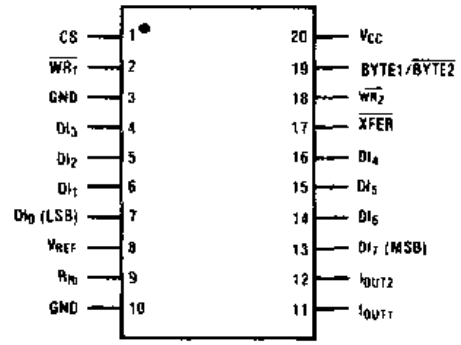


图 11.9 DAC1230 的引脚

各引脚功能如下：

- /CS：输入寄存器选择信号。
- /WR1：输入寄存器的写信号。
- /WR2：DAC 寄存器写信号。
- Vref：参考电压输入。
- Rfb：反馈信号输入。
- /XFER：数据传送信号。
- Iout1、Iout2：模拟电流输出。
- BYTE1\BYTE2：数据输入寄存器选择信号。
- DI0~DI7：数据总线(12 位)。
- Vcc、GND：电源输入。

如图 11.10 所示为 DAC1230 的内部逻辑结构。

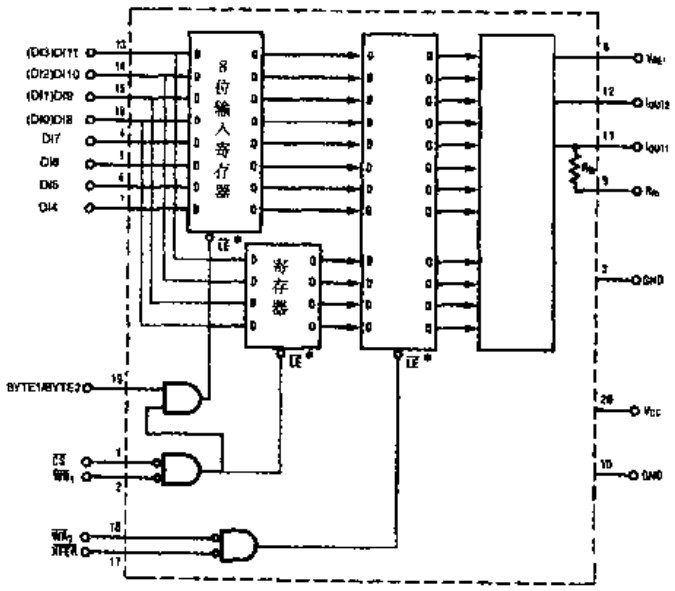


图 11.10 DAC1230 的内部逻辑结构

DAC1230 的内部包括一个 8 位的输入锁存器、一个 4 位的输入锁存器、一个 12 位的 DAC 寄存器和一个 12 位的 D/A 转换器。其中，低 4 位的输入锁存器同高 8 位的输入锁存器一同构成了 12 位长度的数据输入，它们由字节选择信号线(BYTE1\/BYTE2)控制，以从 8 位数据总线上获取 12 位数据；12 位 DAC 寄存器由/XFER 和/WR2 共同控制用于存放待转换的数字量；12 位 D/A 转换器先输出与数字量成正比的模拟电流，再经运算放大器得到模拟电压输出。

11.2.5　12 位数模转换器与单片机的接口及编程

DAC1230 与单片机的接口中，最重要的就是实现 12 位数据的分时传送。为了实现这一目的，接口电路中增加了一条控制线用于在数据输出时选择不同的 DAC 输入寄存器。当 DAC1230 的数据输入寄存器选择信号为“1”时，单片机输出高 8 位数据；当该信号线为“0”时，输出低 4 位数据。

AT90LS8535 与 DAC1230 的接口如图 11.11 所示。DAC1230 的工作采用双缓冲方式。在进行数据传送时，单片机应先输出高 8 位数据，然后再输出低 4 位数据，而不能按相反的顺序传送。这是因为在输出高 8 位数据时，低 4 位寄存器也是打开的，如果先送低 4 位，后送高 8 位，结果就不正确。在 12 位数据都发送到数模转换器的输入寄存器后，单片机就可以打开 DAC 寄存器，以完成数模转换。

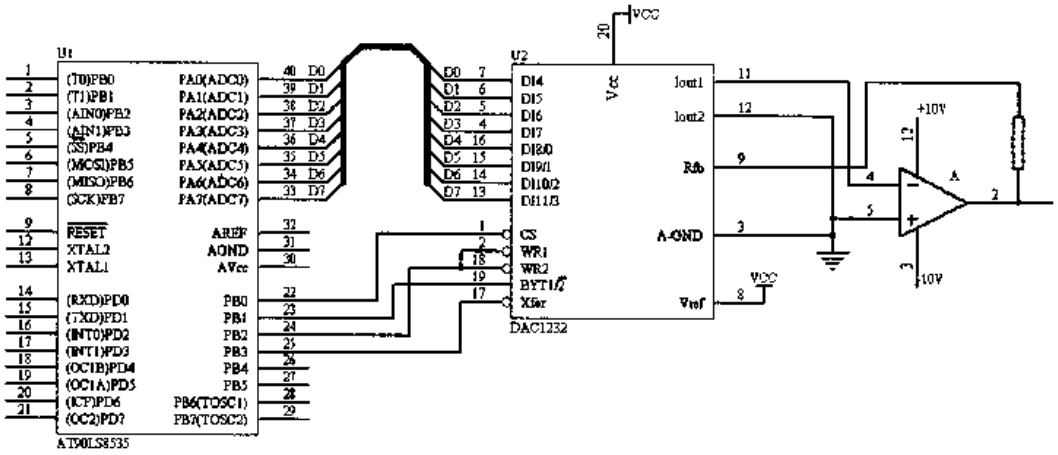


图 11.11　DAC1230 与单片机的硬件接口

DAC1230 输出锯齿波的源程序如下：

```
#include "io8535.h"
#define uint unsigned int
#define uchar unsigned char

//主程序
void main()
{
    uint i;
    uchar x;
    DDRA=0xff;              //初始化 A、B 端口
```

```
    DDRB=0xff;
    PORTB|=BIT(PB0);
    PORTB|=BIT(PB1);
    PORTB|=BIT(PB2);
    PORTB|=BIT(PB3);
    do
    {
if(i==0xfff)
          i=0;
        x=(i>>4)&0xff;
        PORTB&=~BIT(PB0);    //置/CS为低电平
        PORTB|=BIT(PB1);     //选择高8位寄存器
        PORTA=x;             //输出高8位数值
        PORTB&=~BIT(PB2);    //产生写脉冲信号
        PORTB|=BIT(PB2);
        PORTB&=~BIT(PB1);    //选择低4位寄存器
        x=i&0xf;
        PORTA=x;             //输出低4位数据
        PORTB&=~BIT(PB2);    //产生写脉冲信号
        PORTB|=BIT(PB2);
        PORTB|=BIT(PB0);     //置/CS为高电平
        PORTB&=~BIT(PB3);    //置/XFER为低电平
        PORTB&=~BIT(PB2);    //产生写脉冲信号
        PORTB|=BIT(PB2);
        PORTB|=BIT(PB3);     //置/XFER为高电平
        i++;
    }while(1);
}
```

11.3 可编程I/O扩展芯片8255A

8255A是Intel公司生产的通用可编程并行输入/输出接口芯片，它具有3个并行的8位I/O口，可以通过程序改变其功能，因此通用性强，使用灵活。

11.3.1 8255A的引脚和内部结构

1. 8255的引脚

8255A的引脚如图11.12所示。

引脚说明：

- /CS：片选信号输入端。
- /RD：读信号输入端。
- /WR：写信号输入端。
- D0～D7：双向数据总线。
- A0、A1：地址选择线，用于选择A口、B口、C口和控制寄存器。

- PA0～PA7：A 口 I/O 线。
- PB0～PB7：B 口 I/O 线。
- PC0～PC7：C 口 I/O 线。
- RESET：复位信号输入端。
- VCC、GND：电源输入端。

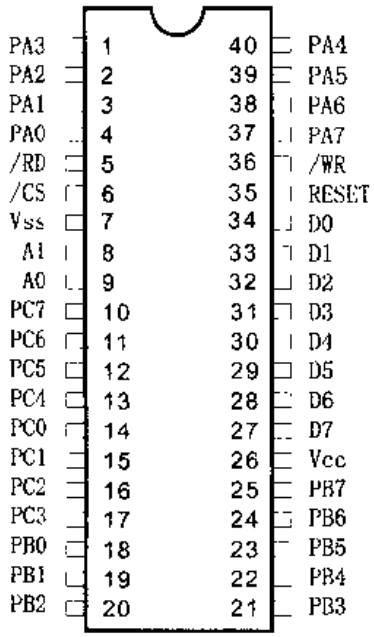


图 11.12　8255A 的引脚

2. 8255A 的内部结构

8255A 的内部结构如图 11.13 所示。

8255A 的内部由 4 部分组成，它们分别是 3 个并行的输入/输出口、口控制逻辑、数据缓冲器和控制逻辑。

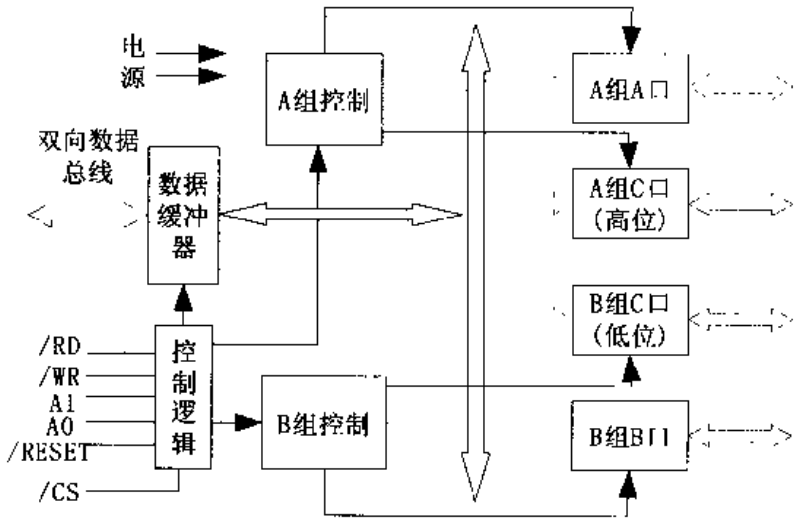


图 11.13　8255A 的内部结构

- 并行输入/输出端口

  A 口、B 口、C 口均为 8 位的并行 I/O 口，但结构上略有差别。其中 A 口由一个 8 位的数据输出锁存/缓冲器和一个 8 位的数据输入锁存/缓冲器构成；B 口由一个 8 位的数据输出锁存/缓冲器和一个 8 位的数据输入缓冲器构成；而 C 口则是由一个 8 位的数据输出锁存/缓冲器和一个 8 位的数据输入缓冲器(无输入数据锁存器)构成。

  PA、PB 和 PC 三个端口都可和外设相连，分别传送外设的输入/输出信息。但通常情况下，都将 A 口和 B 口作为数据的输入/输出口，而将 C 口作为控制/状态信息端口。

- 口控制逻辑

  口控制逻辑是两组根据微控制器的命令字控制 8255A 工作方式的电路结构。它们通过控制寄存器接受微控制输出的命令，然后分别决定两组端口的的工作方式。两组电路控制的端口如下：

  A 组控制电路控制端口 A 和端口 C 的上半部(PC7～PC4)。

  B 组控制电路控制端口 B 和端口 C 的下半部(PC3～PC0)。

- 数据缓冲器

  数据缓冲器是一个三态双向的 8 位缓冲器，它是 8255A 与微控制器数据进行交换的接口。

- 控制逻辑

  8255A 的控制逻辑可以接收微控制器发出的读/写命令和选口地址，从而对 8255A 工作状态进行管理。控制逻辑的控制线包括/CS、/RD、/WR、A0、A1 和 RESET，它们可实现的功能如表 11.4 所示。

表 11.4 8255A 控制信号的控制功能

/CS	A1	A0	/RD	/WR	端口	工作状态
1	-	-	-	-	-	-
0	0	0	0	1	A	A 口数据→数据总线
0	0	0	1	0	A	数据总线→A 口
0	0	1	0	1	B	B 口数据→数据总线
0	0	1	1	0	B	数据总线→B 口
0	1	0	0	1	C	C 口数据→数据总线
0	1	0	1	0	C	数据总线→C 口
0	1	1	1	0	控制端口	写控制命令

11.3.2 8255A 的工作方式

8255A 具有 3 种基本的工作方式。

1. 方式 0：基本输入/输出

8255A 的 A 口、B 口和 C 口均可工作在方式 0，并可根据需要设定各端口为输入或输出方式。在模式 0 时，微控制器和 8255A 以无条件方式传送数据，但也可以设定端口中的某几位为外设的状态输入口，这样，CPU 就可以对这些状态位进行查询以实现数据的异步传送。

2. 方式 1：选通输入/输出

方式 1 为选通输入或选道输出方式。在这种方式下，3 个端口分为 A 组和 B 组，每个组包括一个 8 位的数据端口和一个 4 位的状态/控制端口。A 口和 B 口通常用于传送和它们相连的外设数据，而 C 口则用作 A 口和 B 口的控制联络信号，以实现中断方式的 I/O 数据传送。C 口的各位功能在输入和输出方式的功能如下：

- 输入方式

  控制/状态信号的连接如图 11.14 所示。

  /STB：选通输入。由外设发送的输入选通信号，用来将输入数据锁存到输入锁存器。

  IBF：输入缓冲器满信号，表示数据已经输入，它由/STB 的下降沿置位，由/RD 信号的上升沿复位。

  INTR：中断请求信号。由 8255A 向微控制器发送，中断请求由/RD 的下降沿复位。

- 输出方式

  控制/状态信号的连接如图 11.15 所示。

  /OBF：输出缓冲器满信号。由 8255 发送给外设的控制信号，表示输出数据已发送到指定端口，外设可将数据取走。它由/WR 信号的上升沿置“0”，由/ACK 信号的下降沿置“1”。

  /ACK：外设响应信号。表示 CPU 输出给 8255 的数据已被外设取走。

  INTR：中断请求信号。表示数据已被外设取走，控制器可以继续输出数据。中断请求信号由/WR 的下降沿复位。

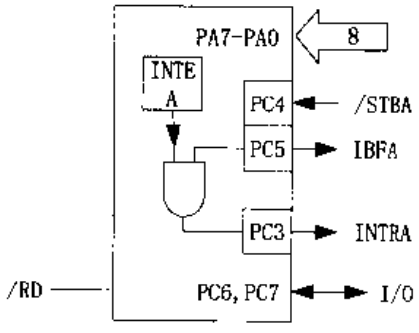


图 11.14　方式 1 的控制/状态信号(输入)

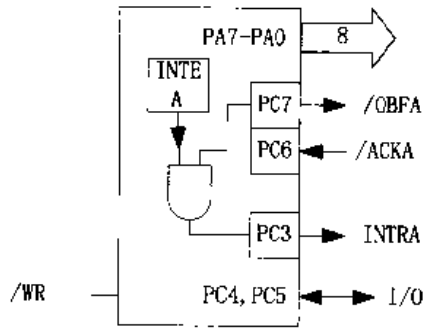


图 11.15　方式 1 的控制/状态信号(输出)

C 口各位与握手信号的对应关系如表 11.5 所示。

表 11.5 C 口各位的功能

C 口	输入方式	输出方式
PC0	INTRB	INTRB
PC1	IBFB	/OBFB
PC2	/STBB	/ACKB
PC3	INTRA	INTRA
PC4	/STBA	I/O
PC5	IBFA	I/O
PC6	I/O	/ACKA
PC7	I/O	/OBFA

3. 方式 2：双向传送

只有 A 口才可设定为方式 2，此时 A 口为双向的 I/O 口，当作为输入总线时，A 口受/STB 和 IBF 控制，其工作过程和方式 1 下的输入方式相同；当作为输出总线时，A 口受/ACK 和/OBF 的控制，其工作过程和方式 1 下的输出方式相同。

其控制信号如图 11.16 所示。

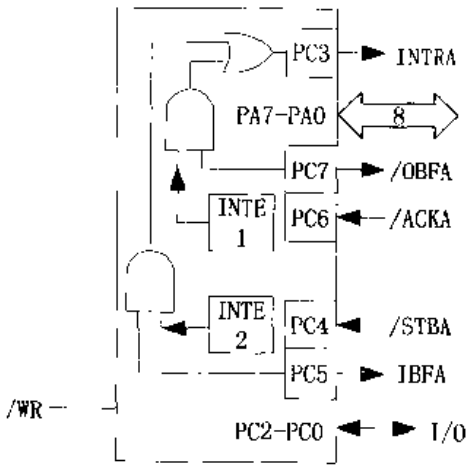


图 11.16 方式 2 的控制信号

控制/状态联络信号的功能如下。

- INTR(PC3)：中断请求信号。
- /STB(PC4)：选通信号。由外设发出，用来将数据锁存至输入锁存器。
- IBF(PC5)：输入缓冲器满信号。表示数据已输入缓冲器。
- /ACK(PC6)：外设应答信号。可使 A 口的输出缓冲器输出数据。
- /OBF(PC7)：输出缓冲器满信号。由 8255A 输出给外设的信号，表示数据已经输出到 A 口，外设可以将数据取走。

方式 2 适合需要进行双向数据传送的 I/O 口。当 A 口工作在这种方式下时，它使用了 C 口的高 5 位作为控制和状态信号端。C 口剩余的低 3 位可以同 B 口构成选通输入或选通输出 I/O 口，也可当作简单的 I/O 使用。

11.3.3　8255A 的控制字

8255A 的控制字包括 C 口的位控控制字和方式控制字，位控控制字的格式如图 11.17 所示。

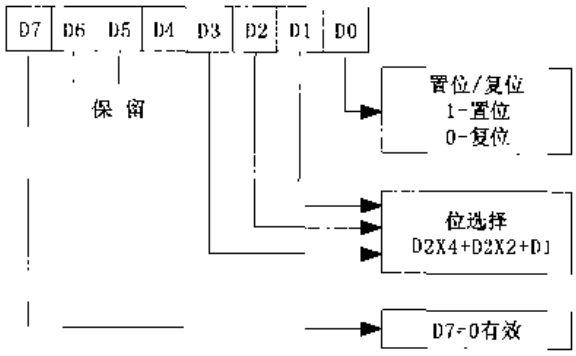


图 11.17　C 口的位操作控制字

注意： C 口的位控功能可以通过方式控制字将任意一位置 1 或清 0。

8255A 的工作方式取决与微控制器写入的方式控制字，方式控制字的格式如图 11.18 所示。其中，D7 为方式标志位，若 D7=1，则控制字为方式控制字：D6～D3 位为 A 组控制位；D2～D0 为 B 组控制位。

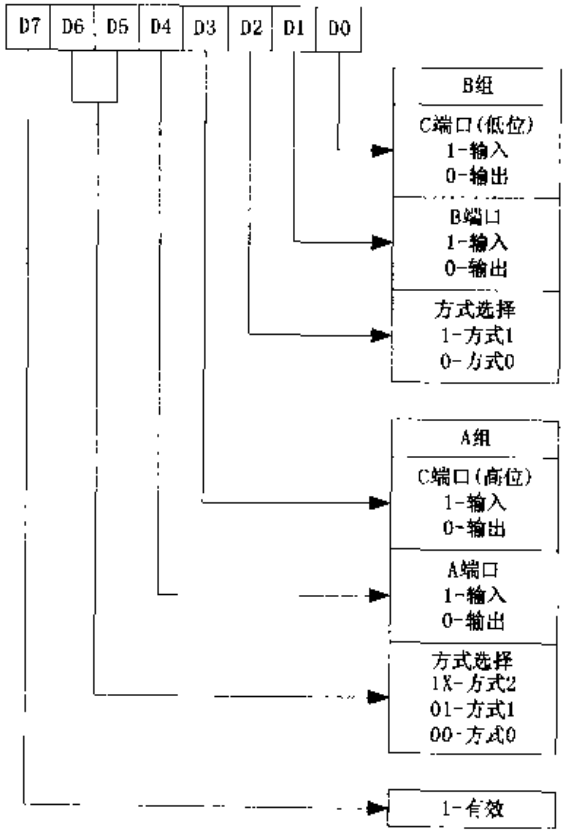


图 11.18　8255A 的控制字格式

11.3.4 AT90LS8535 和 8255A 的接口

AT90LS8535 和 8255A 的接口如图 11.19 所示。

如图，8255A 的 D0～D7 接单片机的 PA 口、地址选择线 A0 和 A1 分别接 PB0 和 PB1，片选信号/CS 接 PB2，读信号/RD 接 PB3，写信号/WR 接 PB4，复位端 RESET 接至 PB5 上。

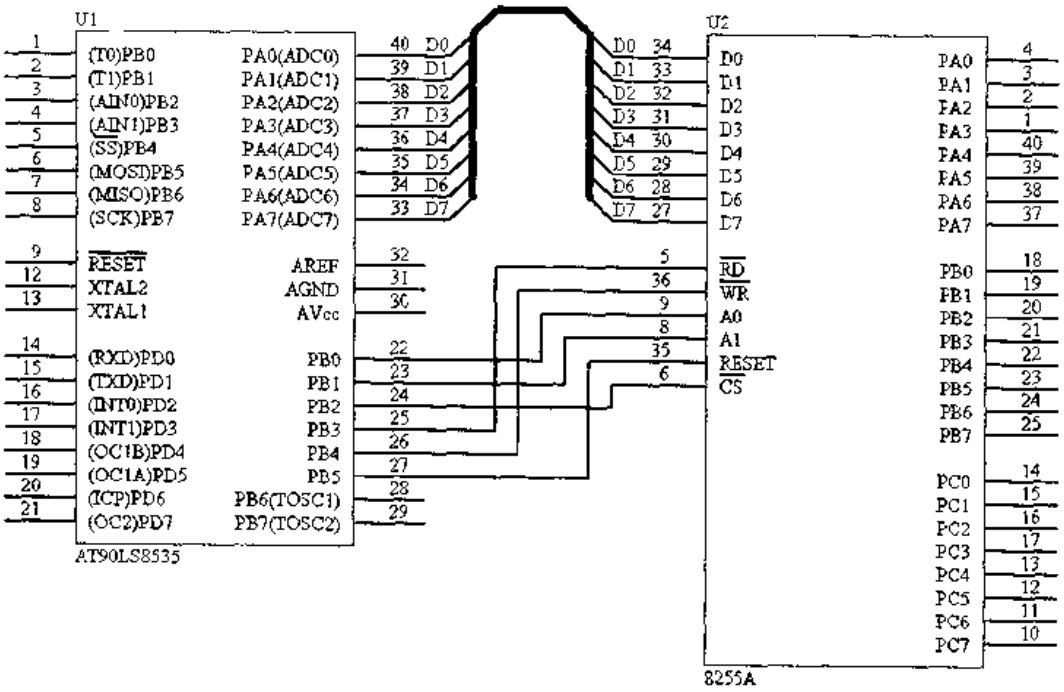


图 11.19 AT90LS8535 和 8255A 的接口电路

以下是采用 8255A 扩展 I/O 口的的源程序。

1. 设定 8255A 的工作方式为 0，A 口输入，B 口和 C 口输出

单片机程序为：

```
#include "io8535.h"
#define uchar unsigned char

//主程序
void main()
{
    uchar outdata1=0x55,outdata2,indata1;
    DDRB=0xff;
    PORTB=0xff;
    for(i=0;i<100;i++)  //复位延时
        ;
    //初始化，并设置方式控制字
    PORTB&=~BIT(PB5);   //置 8255A 为正常模式
    DDRA=0xff;
    PORTB&=~BIT(PB2);   //置/CS 为低电平
```

```
    PORTB|=BIT(PB1);     //置 A0 和 A1 为 11
    PORTB|=BIT(PB2);
    PORTA=0x90;          //设置方式 0，A 口输入，B 口、C 口输出
    PORTB&=~BIT(PB4);    //发送写脉冲
    PORTB|=BIT(PB4);
    PORTB|=BIT(PB2);     //置/CS 为高电平
    //读 8255A 的 A 口数据
    PORTB&=~BIT(PB2);    //置/CS 为低电平
    PORTB&=~BIT(PB1);    //置 A0 和 A1 为 00
    PORTB&=~BIT(PB2);
    DDRA=0;
    PORTB&=~BIT(PB3);    //发送读脉冲
    indata1=PINA;
    PORTB|=BIT(PB3);
    PORTB|=BIT(PB2);     //置/CS 为高电平
    //向 8255A 的 B 口写数据
    PORTB&=~BIT(PB2);    //置/CS 为低电平
    PORTB|=BIT(PB1);     //置 A0 和 A1 为 01
    PORTB|=BIT(PB2);
    DDRA=0xff;
    PORTA=outdata1;
    PORTB&=~BIT(PB4);    //发送写脉冲
    PORTB|=BIT(PB4);
    PORTB|=BIT(PB2);     //置/CS 为高电平
    //向 8255A 的 C 口写数据
    PORTB&=~BIT(PB2);    //置/CS 为低电平
    PORTB|=BIT(PB1);     //置 A0 和 A1 为 10
    PORTB|=BIT(PB2);
    DDRA=0xff;
    PORTA=outdata2;
    PORTB&=~BIT(PB4);    //发送写脉冲
    PORTB|=BIT(PB4);
    PORTB|=BIT(PB2);     //置/CS 为高电平
}
```

2. 对 8255A 的 C 口进行位控

单片机程序为：

```
#include "io8535.h"
#define uchar unsigned char

//主程序
void main()
{
    DDRB=0xff;
    PORTB=0xff;
    for(i=0;i<100;i++)
        ;
    PORTB&=~BIT(PB5);
    DDRA=0xff;
    //置位 C7 位
```

```
    PORTB&=~BIT(PB2);    //置/CS 为低电平
    PORTB|=BIT(PB1);     //置 A0 和 A1 为 11
    PORTB|=BIT(PB2);
    PORTA=0x0f;
    PORTB&=~BIT(PB4);    //发送写脉冲
    PORTB|=BIT(PB4);
    PORTB|=BIT(PB2);     //置/CS 为高电平
    //C6 位复位
    PORTB&=~BIT(PB2);    //置/CS 为低电平
    PORTB|=BIT(PB1);     //置 A0 和 A1 为 11
    PORTB|=BIT(PB2);
    PORTA=0x0C;
    PORTB&=~BIT(PB4);    //发送写脉冲
    PORTB|=BIT(PB4);
    PORTB|=BIT(PB2);     //置/CS 为高电平
}
```

11.4 带片内 RAM 的 I/O 扩展芯片 8155

8155 也是 Intel 公司生产的可编程并行 I/O 扩展芯片，它的内部包含 2 个 8 位、1 个 6 位的可编程 I/O 口、1 个 14 位的定时/计数器和 256 B 的 RAM。

11.4.1 8155 的引脚和内部结构

1. 8155 的引脚

8155 的引脚排列如图 11.20 所示。

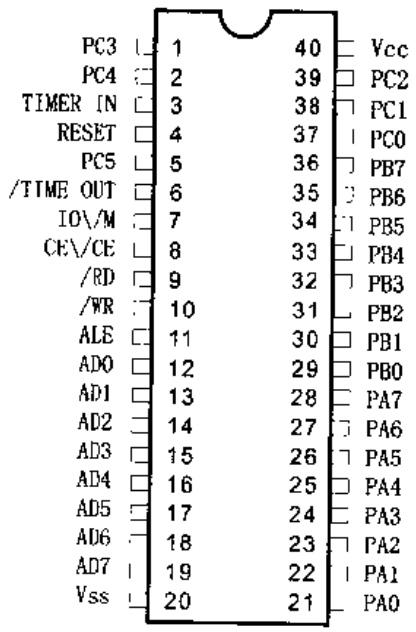


图 11.20 8155 的引脚

引脚说明：

- /CE：片选信号。
- /RD：读信号。
- /WR：写信号。
- ALE：地址锁存信号。
- IO\M：I/O 和存储器选择信号。8155 的内部 I/O 和 RAM 是分开编址的，因此需要使用独立的控制信号加以区分。该管脚为“0”时，控制器对 RAM 进行操作；该管脚为“1”时，控制器对 I/O 口读写。
- TIMER IN：定时/计数器输入。
- /TIME OUT：定时/计数器输出。
- RESET：复位信号。
- AD0～AD7：地址/数据总线。
- PA0～PA7：A 口 I/O 线。
- PB0～PB7：B 口 I/O 线。
- PC0～PC5：C 口 I/O 线(C 口还可作为 A、B 口的握手信号线)。
- Vss、Vdd：电源输入。

2. 8155 的内部结构

8155 的内部结构如图 11.21 所示。

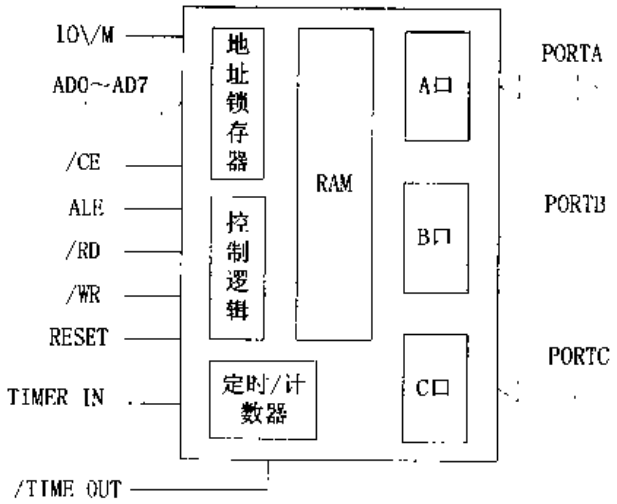


图 11.21　8155 的内部结构

8155 内部由地址锁存器、RAM 存储器、I/O 接口单元、定时/计数器和控制逻辑 5 部分组成。各部分电路的功能如下：

- 地址锁存器

  地址锁存器用于锁存由微控制器送来的 RAM 单元地址或 I/O 端口地址。

- RAM 存储器

  8155 的 RAM 存储器共有 256 字节的容量，它们主要用于存放微控制的临时数据。存储单元的地址通过地址锁存信号 ALE 从地址/数据总线上获得。

- I/O 接口单元

  8155 具有 PA、PB 和 PC 三个 I/O 端口。其中，A 口和 B 口为 8 位，用于进行与外设数据的输入和输出；C 口只有 6 位，它用于简单的 I/O 操作或传递 A、B 口的命令和状态信号。

- 定时/计数器

  8155 具有一个 14 位结构的减 1 计数器。计数器的初值由微控制器通过程序设定。定时/计数器通过 TIMER IN 输入计数脉冲，每当计数器满溢出时，/TIME OUT 上可产生一个终止脉冲。

- 控制逻辑

  控制逻辑用于接收微控制器传来的命令，并管理 8155 的工作方式，其控制功能如表 11.6 所示。

表 11.6 8155 控制信号的控制功能

IO\/M	A7	A6	A5	A4	A3	A2	A1	A0	功能
0	-	-	-	-	-	-	-	-	访问相应存储器单元
1	-	-	-	-	-	0	0	0	问命/状寄存器
1	-	-	-	-	-	0	0	1	访问 A 口
1	-	-	-	-	-	0	1	0	访问 B 口
1	-	-	-	-	-	0	1	1	访问 C 口
1	-	-	-	-	-	1	0	0	访问定时器低 8 位
1	-	-	-	-	-	1	0	1	访问定时器高 6 位和定时器方式位

11.4.2 8155 的 I/O 口工作方式

8155 具有 A 口、B 口和 C 口 3 个 I/O 口，其中 A 口和 B 口是 8 位的通用输入/输出口，主要用于数据的 I/O 操作，它们只能传递数据，因此只有输入和输出两种工作方式。C 口只有 6 位，它们既可以作为数据口使用，也可以作为传送控制信号和状态信息的控制口使用，因此 C 口具有 4 种工作状态，即输入方式、输出方式、A 口控制端口方式和 B 口控制端口方式。

当进行无条件数据传输时，A 口、B 口和 C 口都可以作为数据 I/O 口使用。但当 A 口或 B 口需要以中断方式传送数据时，C 口就必须为 A 口和 B 口提供控制/状态信号了，其中，C0～C2 为 A 口提供，C3～C5 为 B 口提供。C 口的各位功能如表 11.7 所示。

表 11.7 C 口各位功能

C 口	选通 I/O 方式作为 A 口的控制口	作为 A 口和 B 口的控制口
C0	AINTR	AINTR
C1	ABF	ABF
C2	/ASTB	/ASTB

续表

C 口	选通 I/O 方式作为 A 口的控制口	作为 A 口和 B 口的控制口
C3	输出	BINTR
C4	输出	BBF
C5	输出	/BSTB

注意：/STB：选通输入。
IBF：输入缓冲器满信号。
INTR：中断请求信号。

11.4.3　8155 的定时/计数器

8155 中具有一个 14 位的定时/计数器，它可用来定时或计数。微控制器可以通过修改计数寄存器来选择计数器的计数方式和计数长度。计数寄存器的格式如表 11.8。

表 11.8　8155 的计数器寄存器

	D7	D6	D5	D4	D3	D2	D1	D0
TH	输出方式		计数器高 6 位					
TL	计数器低 8 位							

8155 的定时/计数器具有 4 种工作方式，它们通过 TH 的 D7 和 D6 位设置。定时器方式及相应的输出波形如图 11.22 所示。

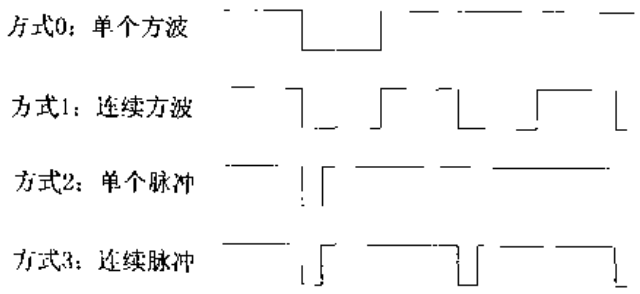


图 11.22　8155 定时器的输出波形

11.4.4　8155 的命令和状态字

8155 具有一个命令/状态寄存器，实际上这是两个不同的寄存器，但由于对命令寄存器只能进行写操作，对状态寄存器只能进行读操作，因此把它们编为一个地址。

1. 8155 的命令字

8155 的命令字用于设定 8155 的工作方式，并实现对中断和定时/计数器的控制。方式控制字的格式如图 11.23 所示。

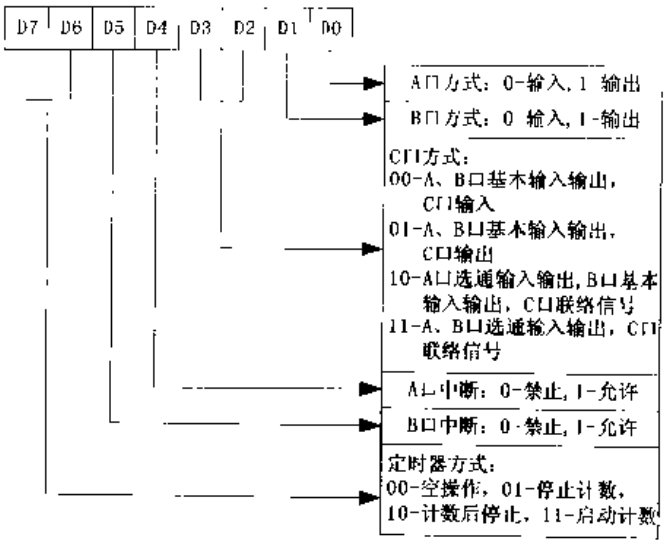


图 11.23 8155 的方式控制字

其中，D7 和 D6 是定时/计数器的方式控制位；D5 和 D4 是 A 口和 B 口的中断使能位；D3 和 D2 是 C 口的方式控制位；D1 是 A 口方式控制位；D0 是 B 口的方式控制位。

2. 8155 的状态字

8155 状态字为 7 位，用于存放 8155 的当前工作状态，其格式如图 11.24 所示。

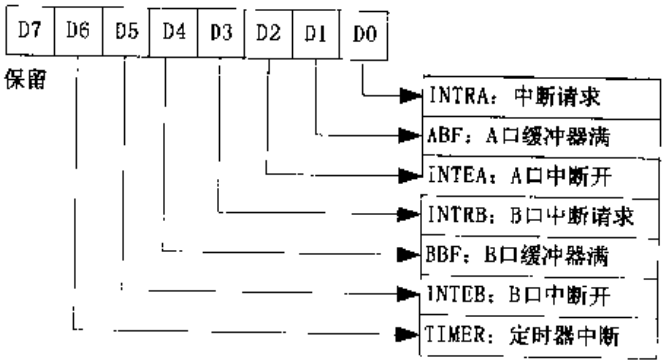


图 11.24 8155 的状态字

11.4.5 AT90LS8535 与 8155 的接口及编程

AT90LS8535 和 8155 的接口如图 11.25 所示。

单片机通过 PA0～PA7 与 8155 的 AD0～AD7 相连，ALE 与 PB0 相连。AT90LS8535 首先输出地址至 PA 口，然后通过 PB0 将地址锁存至 8155 中，随后单片机再进行数据的输出或输入操作。

单片机的驱动程序为：

```
#include "io8535.h"
#define uchar unsigned char

//主程序
```

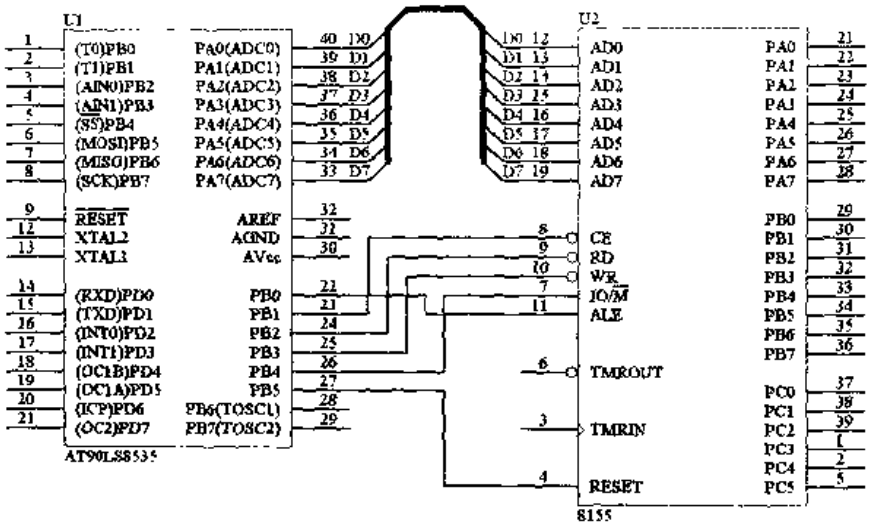


图 11.25　AT90LS8535 与 8155 的接口电路

```
void main()
{
    uchar dataA,dataB=0x55;
    DDRB=0xff;
    PORTB=0xff;
    //复位 8155
    for(i=0;i<100;i++)  //延时
    {
;
}
    PORTB&=~BIT(PB5);
    DDRA=0xff;

    //设置 8155 的工作方式
    PORTB|=BIT(PB4);     //选择 IO
    PORTB&=~BIT(PB1);    //置/CS 为低电平
    PORTB|=BIT(PB0);     //锁存低位地址
    PORTA=0x04;          //指向计数器低位地址
    PORTB&=~BIT(PB0);

    PORTA=0x10;          //输出计数常数
    PORTB&=~BIT(PB3);    //发送写脉冲
    PORTB|=BIT(PB3);

    PORTB|=BIT(PB0);     //锁存低位地址
    PORTA=0x05;          //指向计数器高位地址
    PORTB&=~BIT(PB0);

    PORTA=0x40;          //设定计数器方波输出
    PORTB&=~BIT(PB3);    //发送写脉冲
    PORTB|=BIT(PB3);

    PORTB|=BIT(PB0);     //锁存低位地址
    PORTA=0;             //指向命令状态口
```

```
    PORTB&=~BIT(PB0);

    PORTA=0xc2;          //设定A口基本输入,B口基本输出，并启动计数器
    PORTB&=~BIT(PB3);    //发送写脉冲
    PORTB|=BIT(PB3);
    //读A口
    PORTB|=BIT(PB0);     //锁存低位地址
    PORTA=1;             //指向A口
    PORTB&=~BIT(PB0);

    DDRA=0;              //设置A口为输入
    PORTB&=~BIT(PB2);    //发送读脉冲
    dataA=PINA;
    PORTB|=BIT(PB2);
    //写B口
    PORTB|=BIT(PB0);     //锁存低位地址
    PORTA=2;             //指向B口
    PORTB&=~BIT(PB0);

    DDRA=0xff;           //设置A口为输出
    PORTA=dataB;
    PORTB&=~BIT(PB3);    //发送写脉冲
    PORTB|=BIT(PB3);

    //将数据dataA写RAM中的00H单元
    PORTB&=~BIT(PB4);    //选择存储器
    PORTB|=BIT(PB0);     //锁存低位地址
    PORTA=0;             //指向RAM的00H单元
    PORTB&=~BIT(PB0);

    DDRA=0xff;           //设置A口为输出
    PORTA=dataA;
    PORTB&=~BIT(PB3);    //发送写脉冲
    PORTB|=BIT(PB3);

    //将数据从RAM中的01H单元读至dataB变量
    PORTB|=BIT(PB0);     //锁存低位地址
    PORTA=1;             //指向RAM的01H单元
    PORTB&=~BIT(PB0);

    DDRA=0;              //设置A口为输入
    PORTB&=~BIT(PB2);    //发送读脉冲
    dataB=PINA;
    PORTB|=BIT(PB2);
}
```

11.5 定时/计数器芯片 8253

8253 是可编程的计数器芯片。它的内部含有 3 个计数器单元，计数频率为 0~2 MHz。

11.5.1　8253 的信号引脚和逻辑结构

8253 是一个 24 引脚的双列直插式芯片，它的引脚排列如图 11.26 所示。

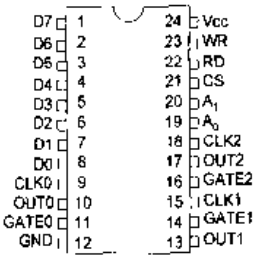


图 11.26　8253 的引脚

引脚说明：

- /CS：片选信号输入端。
- /RD：读信号输入。
- /WR：写信号输入。
- CLK0～CLK2：计数器的时钟信号输入。
- GATE0～GAYT2：门控信号。GATE 信号用于启动计数器(GATE 信号启动计数的方式随计数器工作方式的不同也略有不同)。
- OUT0～OUT2：计数器输出信号。当计数寄存器减到 0 时，OUT 端产生一个输出信号。
- D0～D7：数据总线。
- A0、A1：地址选择线，用于选择 8253 的内部寄存器。
- Vcc、GND：电源输入端。

如图 11.27 所示，8253 具有总线锁存器、控制逻辑、控制寄存器和计数器 4 个逻辑部分。

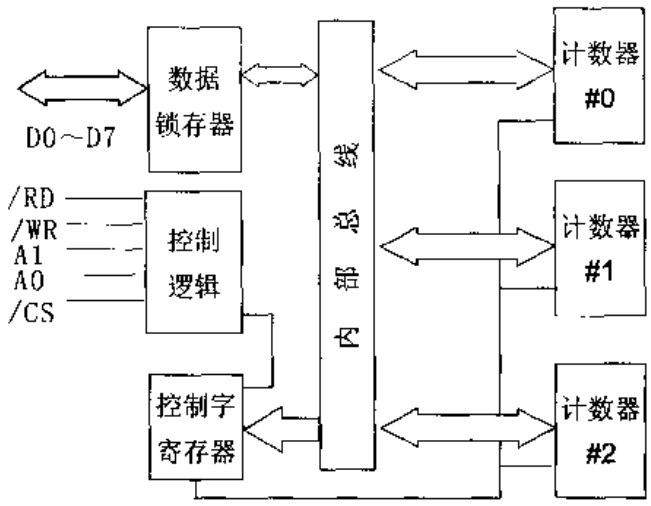


图 11.27　8253 的内部逻辑结构

- 总线锁存器

  8253 的总线锁存器是一个双向的 8 位锁存器。它可以同系统的数据总线相连，微处理器通过它将控制命令字和计数值写入 8253。

- 控制逻辑

  控制逻辑用于接收来自微处理器的控制信号，并完成对各个计数寄存器的读/写操作。这些控制信号包括读/写信号、片选信号和地址选择信号，其控制功能如

表 11.9 所示。

表 11.9 8253 控制信号的控制功能

/CS	/WR	/RD	A1	A2	功能
0	0	1	0	0	写入计数器 0 的计数初值
0	0	1	0	1	写入计数器 1 的计数初值
0	0	1	1	0	写入计数器 2 的计数初值
0	0	1	1	1	写控制寄存器
0	1	0	0	0	读计数器 0 的计数初值
0	1	0	0	1	读计数器 0 的计数初值
0	1	0	1	0	读计数器 0 的计数初值
0	1	0	1	1	无操作
1	-	-	-		无操作

- 控制寄存器
  控制寄存器用来存储微处理器写入的控制字，以决定各计数器的工作方式和计数初值。
- 计数器
  8253 由 3 个计数器单元构成，每个计数器单元都有时钟输入信号 CLK、门控信号 GATE 和一个计数器输出信号 OUT。8253 采用减 1 方式计数，即：每个输入脉冲都使计数寄存器内的计数值减 1，直至为 0，此时，8253 在输出端上产生一个输出信号。

11.5.2 8253 的工作方式

8253 的每个计数器有 6 种可选择的工作方式。对于每个工作方式，CLK 都用于时钟信号的输入，GATE 信号都用于启动/停止计数器，而 OUT 信号则用于在计数结束时输出计数完成标志。

1. 方式 0：计数结束中断方式

这种方式采用/WR 信号或 GATE 信号启动计数器。在计数寄存器计数到 0 时，通过 OUT 端口产生中断信号。

2. 方式 1：可编程单稳态输出方式

这种方式用于产生负脉冲输出，脉冲宽度为计数初值。

3. 方式 2：比率发生器方式

采用这种工作方式时，OUT 先输出一个时钟宽度的负脉冲，然后在输出一个宽度为(计数初值-1)个时钟宽度的正脉冲。

4. 方式 3：方波发生器方式

这种计数器工作方式可产生方波信号。若计数值 n 为偶数，则计数器输出端在 n/2 个计数周期时输出高电平，在另 n/2 个计数周期输出低电平；若计数值 n 为奇数，则在(n+1)/2 个计数周期时输出高电平，在(n-1)/2 个计数周期时输出低电平。

5. 方式 4：软件触发选通方式

这种方式下的计数操作是从写入计数值开始的，在方式控制字写入以后，OUT 就输出高电平，并且在计数到最后一个数时，输出一个负脉冲选通信号，又变为高电平。

6. 方式 5：硬件触发选通方式

在这种方式下，计数器在写入方式控制字后并不启动计数，而只有在 GATE 信号的上升沿到来之后才启动计数器。OUTT 信号在 GATE 输入之前一直保持高电平，直至计数结束，才输出一个时钟宽度的负脉冲。

11.5.3　8253 的控制字

8253 是可编程的计数器芯片，微处理器通过写入控制命令字来对 8253 进行编程，并完成 8253 工作方式的设定。

8253 的命令字为 8 位，用于定义各个计数器的动作方式。命令字的格式如图 11.28 所示，其中，D7 和 D6 寻址 3 个计数器；D5 和 D4 设定计数器的计数寄存器的读/写方式；D3、D2 和 D1 设定计数器的 6 种工作方式；D0 位则定义计数寄存器中值的表示形式。

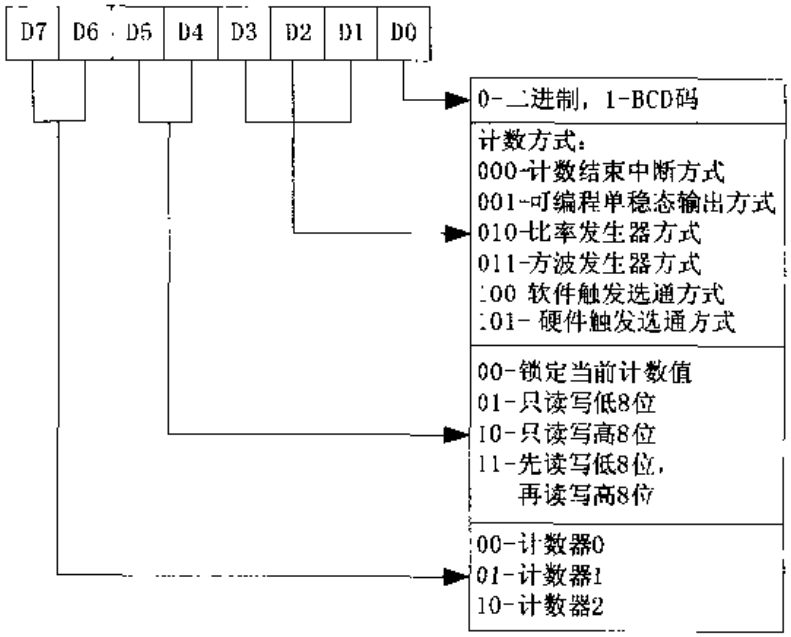


图 11.28　8253 的控制字

11.5.4 AT90LS8535 与 8253 的接口及编程

AT90LS8535 和 8253 的接口如图 11.29 所示。

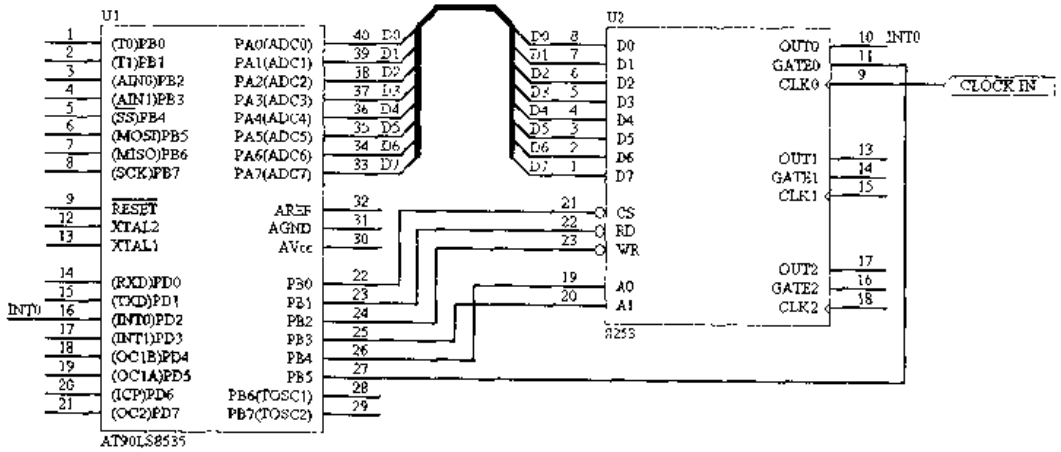


图 11.29 AT90LS8535 和 8253 的接口电路

如图，该电路用于对外部脉冲进行计数，计数满 1000 后，输出供单片机处理的中断信号。其中，外部脉冲信号由 8253 的 CLK0 端输入。AT90LS8535 的 PA0～PA7 接 8253 的 D0～D7 端，PB0 接/CS，PB1 接/RD，PB2 接/WR，PB3 接 A1，PB4 接 A0，PB5 接 GATE0，INT0 接 OUT0。

源程序为：

```
#include "io8535.h"
#define uchar unsigned char
        //定义中断
#pragma interrupt_handler INT0:2
uchar data;

//中断服务子程序
void  INT0(void)
{
    //中断服务
}

//主程序
void main()
{
    GIMSK=0x40;          //开中断，上升沿触发
    MCUCR=0x03;
    DDRA=0xff;           //初始化 A、B 端口
    DDRB=0xff;
    PORTB=0xff;
    PORTB&=~BIT(PB0);    //置/CS 为低电平
    PORTB|=BIT(PB5);     //置 GATE 为高电平
    PORTB|=BIT(PB3);     //置 A0、A1 为 11
```

```
    PORTB|=~BIT(PB4);
    PORTA=0x31;          //采用方式 0 对外界计数
    PORTB&=~BIT(PB2);    //发送写脉冲
    PORTB|=BIT(PB2);
}
```

11.6　实时时钟芯片 DS1302

DS1302 是涓流式充电的时间管理芯片(Trickle Charge Timekeeping Chip)，它通过一个简单的串行接口与微处理机通信，可提供秒、分、时、星期、日、月和年等信息。对于少于 31 天的月份，芯片会自动予以校正。

DS1302 的时钟运行可以采用 24 小时制或以 AM/PM 标识的 12 小时制。它与微处理机的通信仅需要 3 条线，即/RST(复位端)、I/O(数据线)和 SCLK(串行时钟)就可以进行。它的片内还具有 31 个字节的静态 RAM，微处理器对 RAM 中的数据存取可以是一次一个字节，也可以是一次 31 个字节。

DS1302 的特性如下：

- 实时时钟包含秒、分、时、星期、日、月和闰年调整(达到 2100 年)。
- 31×8 个字节的 SRAM。
- 串行式 I/O。
- 电压操作范围：2.0 V～5.5 V。
- 在 2.0 V 时，电流小于 300 nA。
- 单一字节或多字节(连续模态)的数据存取。
- 3 线的外部接口。

11.6.1　DS1302 的引脚和内部结构

1. DS1302 的引脚

DS1302 的引脚排列如图 11.30 所示。

引脚功能说明：

- X1、X2：晶振输入。
- Vcc2、GND：电源输入端。
- Vcc1：备用电源输入。
- I/O：数据输入/输出。
- /RST：复位信号输入。
- SCLK：串行时钟信号输入。

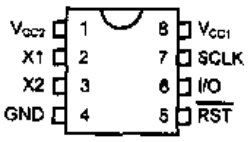


图 11.30　DS1302 的引脚排列

2. DS1302 的内部结构

DS1302 的内部结构如图 11.31 所示。

如图，DS1302 的主要部份包括：移位寄存器、控制逻辑、振荡器、实时时钟和 RAM。

其中，移位寄存器和 RAM 完成同微处理器的数据交换关系，控制逻辑用于管理芯片的工作过程，而实时时钟和振荡器则实现时钟的计数。

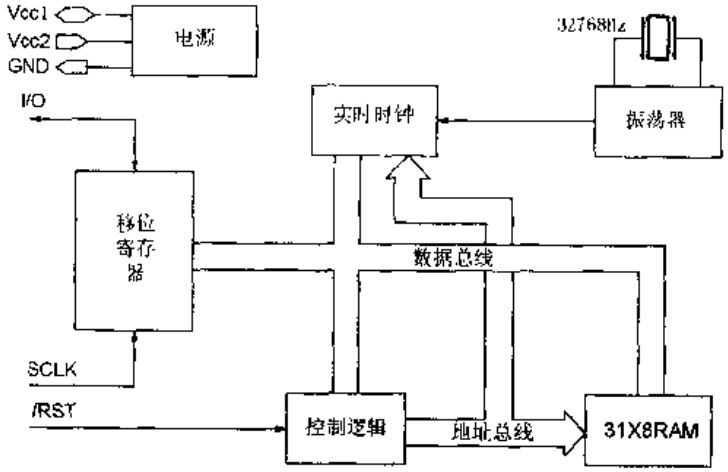


图 11.31 DS1302 的内部结构

11.6.2 DS1302 的控制方式

1. 操作方法

为了初始化数据的存取，/RST 引脚必须置为高电平，并将地址和命令数据加载到移位寄存器中。串行数据在 SCLK 的上升沿时输入。其中，串行数据的第一个字节指出 40 个字节中的哪一个将被操作，在第一个 8 位时钟脉冲产生以后，命令字就加载到移位寄存器，随后的时钟脉冲就开始进行数据的传送。

2. 命令字节(COMMAND BYTE)

命令字节如表 11.10 所示。

表 11.10 DS1302 的命令字

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
1	RAM/CK	A4	A3	A2	A1	A0	RD/W

DS1302 的每一次数据传送都是以命令字节的初始化开始的。最高位(Bit7)必须为“1”，如果为“0”，则表示禁止写 DS1302。Bit6 为“0”指定时钟/日历的数据，为“1”指定 RAM 数据的地址。Bit1～Bit5 用于指定特定的寄存器，最小位(Bit0)指定读操作或写操作，如果为“0”表示是要写入，如果为“1”表示要读出。

3. 复位和时钟控制

所有数据的传送都是以置/RST 引脚为高电平开始的。/RST 输入有两种功能。首先，/RST 可以打开控制逻辑，允许命令字节输入到移位寄存器中；其次，/RST 引脚的信号可以终止数据的传送。在数据输入时，数据必须在时钟的上升沿有效，而数据位在时钟的下降沿输

出。如果/RST 变为“0”，则终止所有的数据传送，并且 I/O 引脚也将进入高阻抗状态。数据传送的时序如图 11.32 所示。

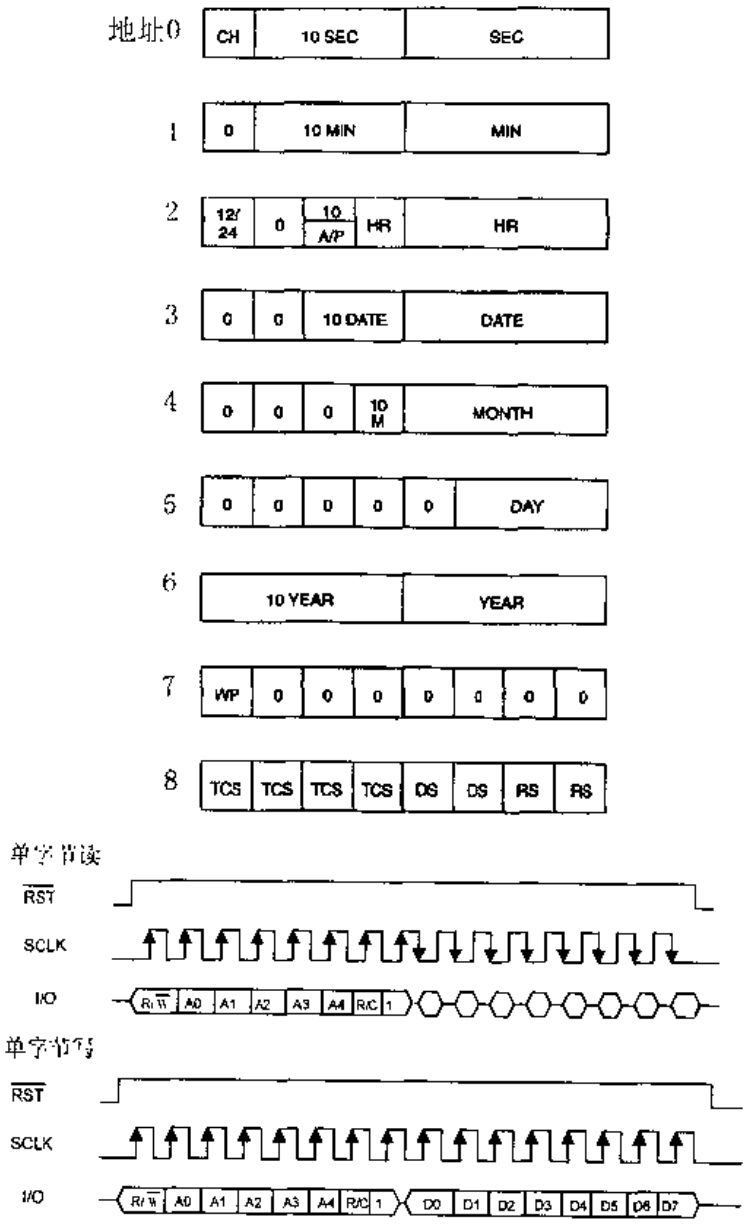


图 11.32　DS1302 的数据传送时序

11.6.3　AT90LS8535 与 DS1302 的接口与编程

AT90LS8535 与 DS1302 的接口如图 11.33 所示。

如图，DS1302 的复位端(/RST)与单片机的 PA0 相连，数据端口(I/O)与 PA1 相连，时钟输入端口(CLK)接 PA2。

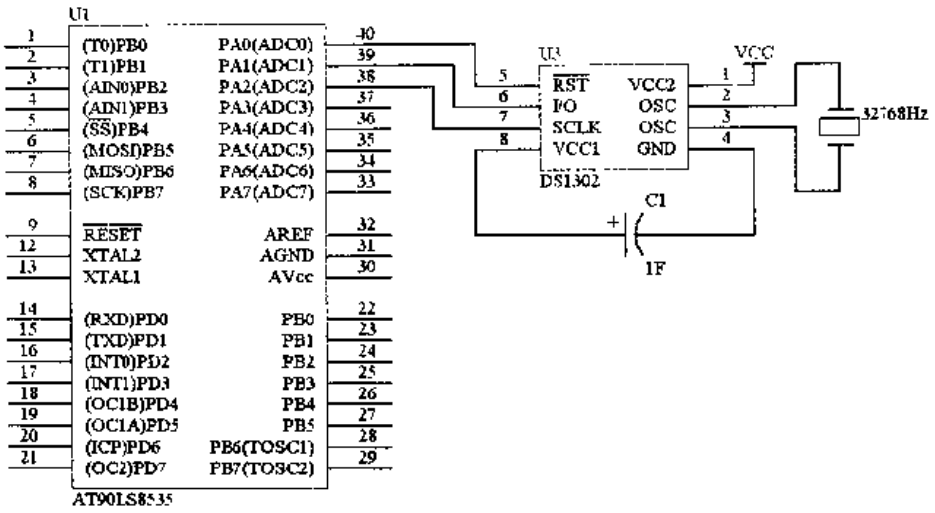


图 11.33 AT90LS8535 与 DS1302 的接口电路

注意： 电路中加入了一个 1 F 的电容，以保证 DS1302 在主电源掉电后继续工作。

以下为 DS1302 的驱动源程序：

```
#define uchar unsigned char
//向 DS1302 写入 1Byte 数据
void InputByte(uchar CD)
{
    uchar i,mid;
    mid=CD&0xC1;
    for(i=8;i>0;i--)
    {
DDRA=0xff;
        if(mid==0)
            PORTA&=~BIT(PA1);
        else
            PORTA|=BIT(PA1);

        PORTA|=BIT(PA2);
        PORTA&=~BIT(PA2);
        CD=CD>> 1;
    }
}

//从 DS1302 读取 1Byte 数据
uchar CutputByte(void)
{
    uchar i,mid;
    for(i=8;i>0;i--)
    {
        DDRA=0;
        mid=mid>>1;
        if((PINA&0x02)==0)
            mid&=~BIT(PA7);
```

```
        else
            mid|=BIT(PA7);
    DDRA=0xff;
    PORTA|=BIT(PA2);
    PORTA&=~BIT(PA2);
    }
    return(mid);
}

//往 DS1302 写入数据
void v_W1302(uchar ucAddr, uchar ucDa)
{
    PORTA&=~BIT(PA0);
    PORTA&=~BIT(PA2);
    PORTA|=BIT(PA0);
    InputByte(ucAddr); /* 地址，命令 */
    InputByte(ucDa); /* 写 1 字节数据*/
    PORTA|=BIT(PA2);
    PORTA&=~BIT(PA0);
}

//读取 DS1302 某地址的数据
uchar uc_R1302(uchar ucAddr)
{
    uchar ucDa;
    PORTA&=~BIT(PA0);
    PORTA&=~BIT(PA2);
    PORTA|=BIT(PA0);
    InputByte(ucAddr); /* 地址，命令 */
    ucDa = OutputByte(); /* 读 1 字节数据 */
    PORTA|=BIT(PA2);
    PORTA&=~BIT(PA0);
    return(ucDa);
}

//往 DS1302 写入时钟数据(多字节方式)
//时钟数据地址格式为：秒 分 时 日 月 星期 年
void v_BurstW1302T(uchar *pSecDa)
{
    uchar i;
    v_W1302(0x8e,0x00); /* 控制命令，WP=0,写操作?*/
    PORTA&=~BIT(PA0);
    PORTA&=~BIT(PA2);
    PORTA|=BIT(PA0);
    InputByte(0xbe); /* 0xbe:时钟多字节写命令 */
    for (i=8;i>0;i--) /*8 字节 = 7 字节时钟数据 + 1 字节控制*/
    {
        InputByte(*pSecDa);/* 写 1 字节数据*/
        pSecDa++;
    }
    PORTA|=BIT(PA2);
```

```
    PORTA&=~BIT(PA0);
}

//读取 DS1302 时钟数据
//时钟数据地址格式为: 秒 分 时 日 月 星期 年
void v_BurstR1302T(uchar *pSecDa)
{
    uchar i;
    PORTA&=~BIT(PA0);
    PORTA&=~BIT(PA2);
    PORTA|=BIT(PA0);
    InputByte(0xbf); /* 0xbf:时钟多字节读命令 */
    for (i=8; i>0; i--)
    {
        *pSecDa = OutputByte(); /* 读 1 字节数据 */
        pSecDa++;
    }
    PORTA|=BIT(PA2);
    PORTA&=~BIT(PA0);
}

// 往 DS1302 寄存器数写入数据(多字节方式)
void v_BurstW1302R(uchar *pReDa)
{
    uchar i;
    v_W1302(0x8e,0x00); /* 控制命令,WP=0,写操作?*/
    PORTA&=~BIT(PA0);
    PORTA&=~BIT(PA2);
    PORTA|=BIT(PA0);
    InputByte(0xfe); /* 0xbe:时钟多字节写命令 */
    for (i=31;i>0;i--) /*31 字节寄存器数据 */
    {
        InputByte(*pReDa); /* 写 1 字节数据*/
        pReDa++;
    }
    PORTA|=BIT(PA2);
    PORTA&=~BIT(PA0);
}

//读取 DS1302 寄存器数据
void v_BurstR1302R(uchar *pReDa)
{
    uchar i;
    PORTA&=~BIT(PA0);
    PORTA&=~BIT(PA2);
    PORTA|=BIT(PA0);
    InputByte(0xff); /* 0xbf: 时钟多字节读命令 */
    for (i=31; i>0; i--) /*31 字节寄存器数据 */
    {
        *pReDa=OutputByte(); /*读 1 字节数据 */
        pReDa++;
```

```
    }
    PORTA|=BIT(PA2);
    PORTA&=~BIT(PA0);
}

//设置初始时间
//初始时间格式为: 秒 分 时 日 月 星期 年
void v_Set1302(uchar *pSecDa)
{
    uchar i;
    uchar ucAddr=0x80;
    v_W1302(0x8e,0x00); /* 控制命令, WP=0, 写操作*/
    for(i =7;i>0;i--)
    {
        v_W1302(ucAddr,*pSecDa); /* 秒 分 时 日 月 星期 年 */
        pSecDa++;
        ucAddr+=2;
    }
    v_W1302(0x8e,0x80); /* 控制命令, WP=1, 写保护*/
}

//读取 DS1302 当前时间
// 时间格式为: 秒 分 时 日 月 星期 年
void v_Get1302(uchar ucCurtime[])
{
    uchar i;
    uchar ucAddr = 0x81;
    for (i=0;i<7;i++)
    {
        ucCurtime[i]=uc_R1302(ucAddr);
        ucAddr+= 2;
    }
}
```

11.7　数字温度传感器 DS18B20

DS18B20 是 DALLAS 公司生产的单线数字温度传感器，它的特点如下：

- 采用单总线的接口方式

  “单线总线”具有经济性好、抗干扰能力强和使用方便等优点，因此用户可轻松地构建经济的传感器网络。

- 测量温度范围宽，测量精度高

  DS18B20 的测量温度范围为-55℃~+125℃；在-10~+85℃ 范围内，测量精度为±0.5℃，而同类型的单线数字温度传感器 DS1822 的精度仅为± 2℃。

- 供电方式灵活

  DS18B20 可以通过内部寄生电路从数据线上获取电源。因此当数据线上的时序满足一定要求时，可以不接外部电源。

- 测量参数可配置
  DS18B20 的测量分辨率可通过程序设定为 9～12 位。
- 掉电保护功能
  DS18B20 内部含有 EEPROM，在系统掉电以后，它仍可保存分辨率及报警温度的设定值。

11.7.1 DS18B20 的引脚和内部结构

1. DS18B20 的引脚

DS18B20 的引脚排列如图 11.34 所示。
引脚功能说明：

- V_{DD}、GND：电源输入端。
- DQ：数据输入输出/端。

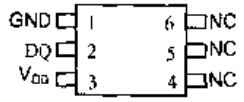


图 11.34 DS18B20 的引脚

2. DS18B20 的内部结构

DS1302 的内部结构如图 11.35 所示。

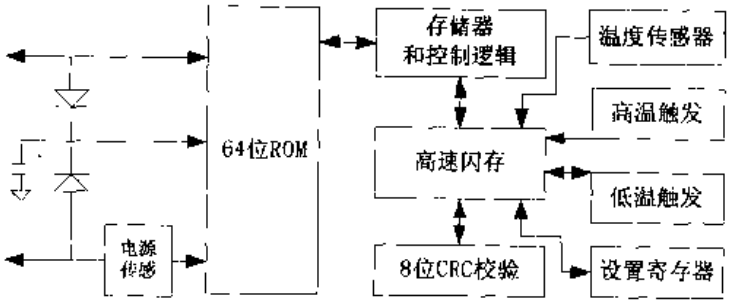


图 11.35 DS1302 的内部结构

如上图，DS18B20 的内部主要包括 64 位的 ROM、温度传感器、非挥发的温度报警触发器(TH、TL)和设置寄存器 4 个部分。

- 64 位 ROM
  64 位的光刻 ROM 包含 3 个部分，其中第一部分(开始 8 位，地址：28H)是产品的类型标号；紧接着的 48 位是 DS18B20 器件的序列号，该序列号是由生产厂商在出厂前被设置好的，并且每个 DS18B20 的序列号都不相同，因此它可以看作是该 DS18B20 的地址序列码；最后 8 位则是前面 56 位数据的循环冗余校验码。由于每一个 DS18B20 的 ROM 中的数据都各不相同，因此微控制器就可以通过单总线对多个 DS18B20 进行寻址，从而实现在单总线上挂接多个 DS18B20 器件。
- 温度传感器
  温度传感器用于完成对温度的测量，它的测量精度可以配置成 9 位、10 位、11 位或 12 位 4 种状态。温度传感器在测量完成后会将测量结果存储在 DS18B20 的两个 8 Bit 的 RAM 中，数据的存储格式如表 11.11 所示(以 12 位转化精度为例)。

表 11.11　DS18B20 的数据存储格式

	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
LSB	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
MSB	S	S	S	S	S	2^6	2^5	2^4

其中，高位字节中的前 5 位是符号位，若测得的温度大于 0，则这 5 位数据也都为“0”，实际温度也就等于测到的数值与 0.0625(0.0625℃/LSB)的乘积；若温度小于 0，则这 5 位数据就都为“1”，实际温度就等于测到的数值取反加 1 后再乘以 0.0625。

例如+125℃的数字输出为 07D0H，+25.0625℃的数字输出为 0191H，-25.0625℃的数字输出为 FF6FH，-55℃的数字输出为 FC90H。

DS18B20 温度传感器中还有一个存储器，该内部存储器包括一个高速暂存 RAM 和一个非易失性的电可擦除的 RAM。后者存放高温度和低温度触发器 TH、TL 和结构寄存器。前者包含了 8 个连续的字节，前两个字节是测得的温度数据，第 1 个字节的内容是温度的低 8 位，第 2 个字节是温度的高 8 位；第 3 个和第 4 个字节是 TH、TL 的易失性拷贝，第 5 个字节是结构寄存器的易失性拷贝，这 3 个字节的内容在每一次上电复位时被刷新；第 6、7、8 个字节用于内部计算；第 9 个字节是冗余检验字节。表 11.12 为 DS18B20 的暂存寄存器分布。

表 11.12　DS18B20 的暂存寄存器分布

寄存器内容	地址
温度的低八位数据	0
温度的高八位数据	1
高温阀值	2
低温阀值	3
保留	4
保留	5
计数剩余值	6
每度计数值	7
CRC 校验	8

- 设置寄存器

  设置寄存器位于高速闪存的第 5 个字节，这个寄存器中的内容被用来确定温度的转换精度。寄存器各位的内容如表 11.13 所示。

表 11.13　DS18B20 的设置寄存器各位内容

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
X	R1	R0	1	1	1	1	1

该寄存器的低 5 位一直都是 1 。TM 是测试模式位，用于设置 DS18B20 在工作模式还是在测试模式。R1 和 R0 则用来设置分辨率，分辨率设置如表 11.14 所示。

表 11.14 分辨率设置

R1	R0	分辨率	温度最大转换时间
0	0	9 位	93.75 ms
0	1	10 位	187.5 ms
1	0	11 位	375 ms
1	1	12 位	750 ms

11.7.2 DS18B20 的温度测量

根据 DS18B20 的协议规定，微控制器控制 DS18B20 完成温度转换必须经过以下 4 个步骤：

(1) 每次读写前对 DS18B20 进行复位初始化。复位要求主 CPU 将数据线下拉 500 ms，然后释放，DS18B20 收到信号后等待 16 ms~60 ms 左右，然后发出 60 ms~240 ms 的存在低脉冲，主 CPU 收到此信号表示复位成功。

(2) 发送一条 ROM 指令，表 11.15 为 DS18B20 的 ROM 指令集。

(3) 发送存储器指令，表 11.16 为 DS18B20 的存储器指令集。

(4) 进行数据通信。

表 11.15 DS18B20 的 ROM 指令集

指令代码	指令名称	指令功能
33H	读 ROM	读器件的 64 位地址
55H	ROM 匹配	访问相对应地址的器件
CCH	搜索 ROM	用于确定总线上的器件个数，并识别各器件的 64 位地址
F0H	跳过 ROM	忽略 64 位地址，直接向器件发送温度变换命令
ECH	警报搜索	该指令执行后，只有温度超过设定的上限和下限报警值时，器件才予以响应

表 11.16 DS18B20 的存储器指令集

指令代码	指令名称	指令功能
4EH	写闪存器	向内部 RAM 中写 TH 和 TL 数据
BEH	读闪存器	读 CRC 校验
48H	复制闪存器	将 RAM 中的 TH 和 TL 复制到 SRAM 中
44H	启动温度转换	启动 DS18B20 的温度转换
B8H	重新调用 SRAM	将 SRAM 中的 TH 和 TL 复制到 RAM 中
B4H	读供电方式	读 DS18B20 的供电方式。外接电源时，DS18B20 发送“1”；否则发送“0”

注意：

- 对 DS1820 进行读写编程时，必须严格的保证读写时序，否则将无法读取测

温结果。

- 对多点测温时，要注意微处理器的总线驱动问题。
- 在 DS1820 测温程序中，向 DS1820 发出温度转换命令后，程序总要等待 DS1820 的返回信号。

11.7.3　AT90LS8535 与 DS18B20 的接口与编程

AT90LS8535 与 DS18B20 的接口如图 11.36 所示。

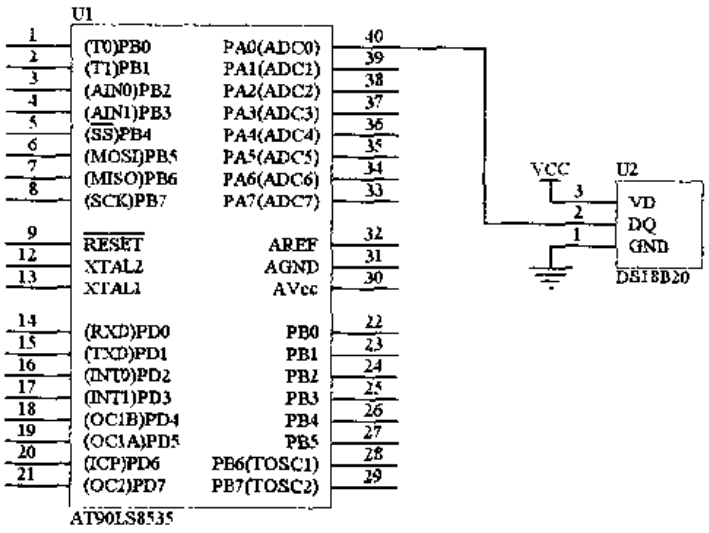


图 11.36　AT90LS8535 与 DS18B20 的接口

如图，DS18B20 通过单总线 DQ 同单片机的 PA0 端相连。

以下为 DS18B20 驱动子程序：

```
#define uchar unsigned char
#define uint  unsigned int

//延时子程序(1μs)
void delay(uint mseconds)
{
    for(;mseconds>0;mseconds--)
        ;
}
//复位子程序
uchar Device_reset(void)
{
    uchar online;
    DDRA|=BIT(PA0);      // 设置 DQ 为输出
    PORTA&=~BIT(PA0);    // DQ 置 0
    delay(480);          // 延时 480μs
    PORTA|=BIT(PA0);     // DQ 置 1
    delay(30);           // 延时等待响应
    DDRA&=~BIT(PA0);     //设置 DQ 为输入
```

```
    online=PORTA&0x01;  // 响应信号
    delay(100);
    return(online);
}

//从单总线上读取一个字节
uchar read_byte(void)
{
    uchar i,mid;
    uchar value=0;
    for (i=8;i>0;i--)
    {
        value>>=1;
        DDRA|=BIT(PA0);      //设置 DQ 为输出
        PORTA&=~BIT(PA0);
        PORTA|=BIT(PA0);
        delay(15);
        DDRA&=~BIT(PA0);     //设置 DQ 为输入
        mid=PINA&0x01;
        if(mid)
            value|=0x80;
        delay(90);
    }
    return(value);
}

//向 单总线上写一个字节
void write_byte(char val)
{
    uchar i,mid;
    for (i=8; i>0; i--)
    {
        DDRA|=BIT(PA0);      //设置 DQ 为输出
        PORTA&=~BIT(PA0);
        mid=val&0x01;
        if(mid)
            PORTA|=BIT(PA0);
        else
            PORTA&=~BIT(PA0);
        delay(90);
        PORTA|=BIT(PA0);
        val=val/2;
    }
    delay(90);
}

//读取温度
int Read_Temperature(void)
{
    char c[2];
    int x;
```

```
    Device_reset();
    write_byte(0xCC);   // 跳过 ROM
    write_byte(0xBE);   // 读闪存
    c[1]=read_byte();
    c[0]=read_byte();
    Device_reset();
    write_byte(0xCC);   // 跳过 ROM
    write_byte(0x44);   // 启动转换
    x=(c1<<8)&c2;
    return x;
}
```

第 12 章　AT90LS8535 的通信编程

随着微机和单片机技术的发展，单机模式的单片机应用系统已经不能满足日益复杂的系统要求，多机联网已成为单片机发展的必然趋势，而多机联网的关键就在于系统间的相互通信。因此本章将重点介绍单片机的各类通信标准，包括串口、I^2C 总线、CAN 总线等多种单片机的双机、多机以及单片机与 PC 机之间的通信技术。

12.1　串 口 通 信

本书在第 9 章中已经对 AT90LS8535 单片机的同步/异步串行口做了介绍，本节将重点讨论如何利用它们实现单片机的双机通信。

12.1.1　异步串口 UART 通信

两个 AT90LS8535 单片机的异步串行通信电路如图 12.1 所示。

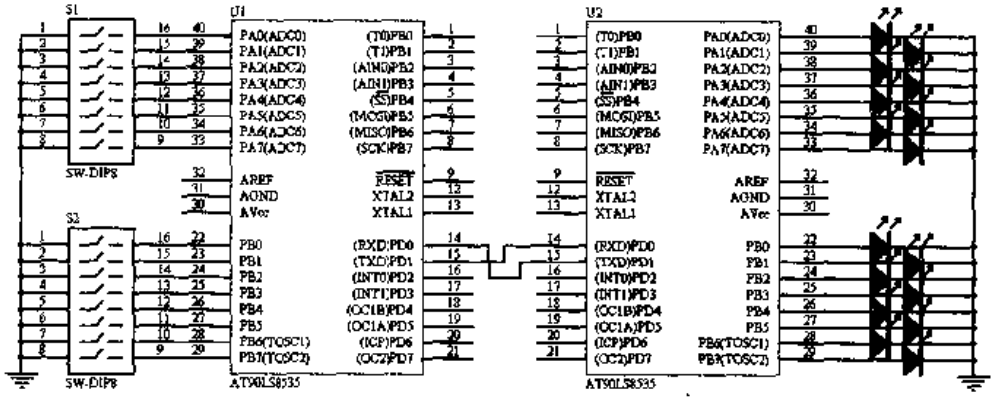


图 12.1　AT90LS8535 单片机双机异步串行通信原理

电路的工作原理：单片机 1 循环输入拨码开关的值，然后通过单片机的串行口输出至 TXD 端口，单片机 1 的 TXD 端再连接到单片机 2 的 RXD 端，这样单片机 2 就能接收到单片机 1 发出的数据，随后，单片机 2 再对片内的串行资源进行操作，并将拨码开关的键值显示出来。

单片机 1 的源程序：

```
#include "io8535.h"
#include "macros.h"
#include "stdio.h"
```

```
#define uchar unsigned char

//串口发送数据
void USART_Transmit( uchar x)
{
    //等待数据寄存器空
    while(!(USR&0x20))
        ;
    UDR=x;
}
//主程序
void main()
{
    uchar mid1,mid2;
    DDRA=0;      //设置A口为输入，带上拉
    PORTA=0xff;
    DDRB=0;      //设置B口为输入，带上拉
    PORTB=0xff;
    UBRRH = 0;  //设置波特率9600
    UBRRL=38;
    UCR=0x08;   //发送使能
    while(1)
    {
        mid1=PINA;
        mid2=PINB;
        USART_Transmit(mid1);
        USART_Transmit(mid2);
    }
}
```

单片机 2 的源程序：

```
#include "io8535.h"
#include "macros.h"
#include "stdio.h"
#define uchar unsigned char

void main()
{
    uchar mid;
    DDRA=0xff;  //设置A口为输出
    DDRB=0xff;  //设置B口为输出
    UBRRH = 0;  //设置波特率9600
    UBRRL=38;
    UCR=0x10;   //接收使能
    while(1)
    {
        if(USR&0x80)
            PORTA=getchar();
        if(USR&0x80)
            PORTB=getchar();
```

```
    }
}
```

12.1.2 同步串口 SPI 通信

两个 AT90LS8535 单片机的同步串行通信电路如图 12.2 所示。

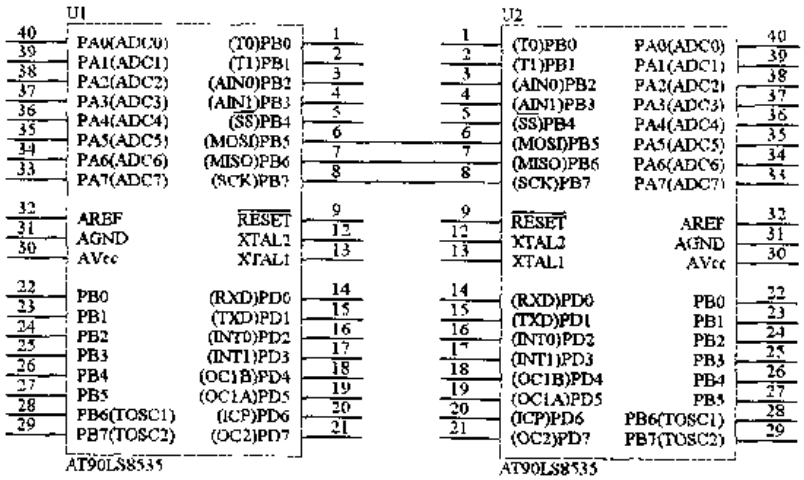


图 12.2 AT90LS8535 单片机双机同步串行通信原理

功能：实现通过 SPI 口相互传递 100 个字节。

单片机 1 为主方式，程序把 1 个字节的数据送至寄存器 SPDR，从而启动 SPI 传送功能。在 SPI 的中断子程序里，单片机 1 把从单片机 2 传来的数据送到接收缓冲区，再把下 1 个数据发送至 SPDR，从而又一次启动 SPI 通信，如此直到所有的数据传输完毕。

单片机 2 为从方式，程序把 1 个字节的数据送至寄存器 SPDR，并等待单片机 1 启动 SPI 传送。在单片机 2 的中断服务程序中，单片机 2 把从单片机 1 传来的数据送至接收缓冲区，再把下 1 个数据发送至 SPDR，然后再等待单片机 1 启动 SPI，如此直到程序完成。

单片机 1 的源程序：

```
#include "io8535.h"
#define uchar unsigned char
#pragma interrupt_handler spi1:11

uchar a[100]={};
uchar b[100];
uchar i=0;

void spi1(void)
{
    b[i]=SPDR;
    SPDR=a[i+1];
    i++;
}

void main()
```

```
{
    SPCR=0xf0;
    SPSR=0;
    SPDR=a[0];
}
```

单片机 2 的源程序：

```
#include "io8535.h"
#define uchar unsigned char
#pragma interrupt_handler spi2:11

uchar a[100]={};
uchar b[100]
uchar i=0;

void spi2(void)
{
    b[i]=SPDR;
    SPDR=a[i+1];
    i++;
}

//主程序
void main()
{
    SPCR=0xe0;
    SPSR=0;
    SPDR=a[0];
}
```

12.2　I²C 总线

I²C 总线，是 INTER-IC 串行总线的缩写。这种串行总线上的各单片机或集成电路模块通过一条串行数据线(SDA)和一条串行时钟线(SCL)进行信息传送。同其他形式的总线相比，I²C 总线具有可靠性好、传输速度快、结构简单等优点，因此也被广泛地应用在单片机应用系统中。

12.2.1　I²C 总线协议

按照 I²C 总线的通信规则，每个总线上的电路模块都有惟一的地址，总线通过这个地址识别连在总线上的器件。每个设备既可以是主控器(能控制总线，完成一次传输过程的初始化，并产生时钟信号及传输终止信号的器件)或被控器(被主控器寻址的器件)，也可以是发送器(在总线上发送信息的器件)或接收器(从总线上接收信息的器件)。

1. I²C 总线的基本结构

采用 I²C 总线标准的单片机或 IC 器件，其内部不仅有 I²C 接口电路，而且还有将内部各单元电路按功能划分的独立模块，它们通过软件寻址实现片选功能，因此减少了器件片选线的连接。总线控制设备不仅能通过指令将某个功能单元电路挂接到总线或摘离总线，还可对该单元的工作状况进行监测，从而实现对硬件系统的简单而又灵活的扩展与控制。I²C 总线接口电路结构如图 12.3 所示。

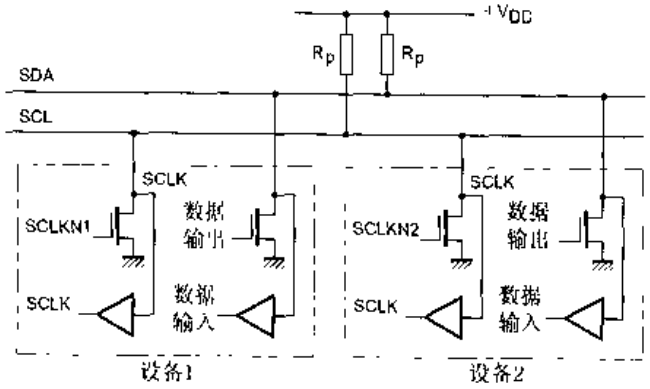


图 12.3 I²C 总线的接口结构

2. I²C 总线的接口特性

传统单片机的串行发送和接收都各用一条线，而 I²C 总线则根据器件的功能，并通过软件程序使其同时工作于发送或接收方式。当某个器件向总线上发送信息时，它就是发送器；而当其从总线上接收信息时，又成为接收器。

I²C 总线上的 SDA 和 SCL 均为双向 I/O 线，它们通过上拉电阻接正电源。当总线空闲时，两根线都是高电平。I²C 总线上的连接器件的输出级都为集电极或漏极开路的形式，这样总线上的数据就能实现线“与”的功能。I²C 总线在传送数据时其速率可达 100 Kb/s，最高速率时可达 400 Kb/s。

I²C 总线在进行信息传输时，若 SCL 为高电平，则 SDA 上的信息必须保持稳定不变；若 SCL 为低电平，则 SDA 上的信息允许变化。SDA 上的每一位数据都和 SCL 上的时钟脉冲相对应。由于 SCL 和 SDA 的线“与”逻辑关系，当 SCL 没有时钟信号，SDA 上的数据也将停止传输。SCL 保持高电平期间，SDA 由高电平向低电平变化这种状态定义为起始信号；而 SDA 由低电平向高电平变化则定义为终止信号。图 12.4 为 I²C 总线的起始信号和终止信号时序。

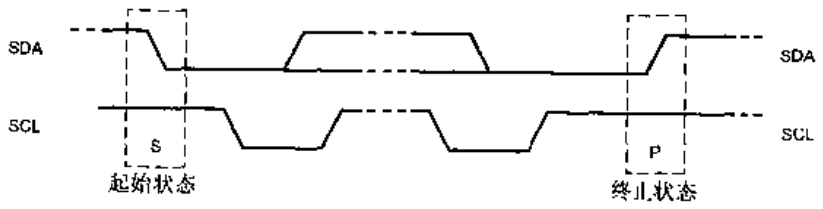


图 12.4 I²C 总线的起始信号和终止信号时序

在 I²C 总线上，SDA 上的数据传输必须以字节为单位(最高位最先传送)，每个传送字节后还必须跟随一个应答位。这个应答信号由发送器发出。整个数据传输过程中，传输的字节数目是没有限制的。但是若数据传输一段时间后，接收器无法继续接收时，主控器也可以终止数据的传送。

3. I²C 总线的竞争仲裁

I²C 总线上可以挂接多个器件，这样，当两个或多个主控器同时想占用总线时，就会产生总线竞争。I²C 总线具有多主控能力，可以对发生在 SDA 线上的总线竞争产生仲裁过程，仲裁是在 SCL 为高电平时，根据 SDA 状态进行的。在总线仲裁期间，如有其他主控器已经在 SDA 上传送低电平，则发送高电平的主控器将会发现此时 SDA 上的电平与它发送的信号不一致，这样，该主控器就自动被裁决失去总线控制权。I²C 总线协议的详细仲裁过程为：当主控器在发送某个字节时，若被仲裁失去主控权时，它的时钟信号继续输出，并直到整个字节发送结束为止。若主控器在寻址阶段被仲裁失去主控权，它就立刻进入被控接收状态，并判断取得主控权的主控器是否正在对它进行寻址。

在仲裁过程中，一旦有个主控器输出低电平时钟信号，则 SCL 也变为低电平状态，它将影响所有有关的主控器，并使它们的计时器复位。如果有一个主控器首先由低电平向高电平转换，这时由于还有其他主控器处于低电平状态，因此它只能处于高电平等待状态，直至所有主控器都结束低电平状态，SCL 才转为高电平。仲裁过程中的时钟同步如图 12.5 所示。

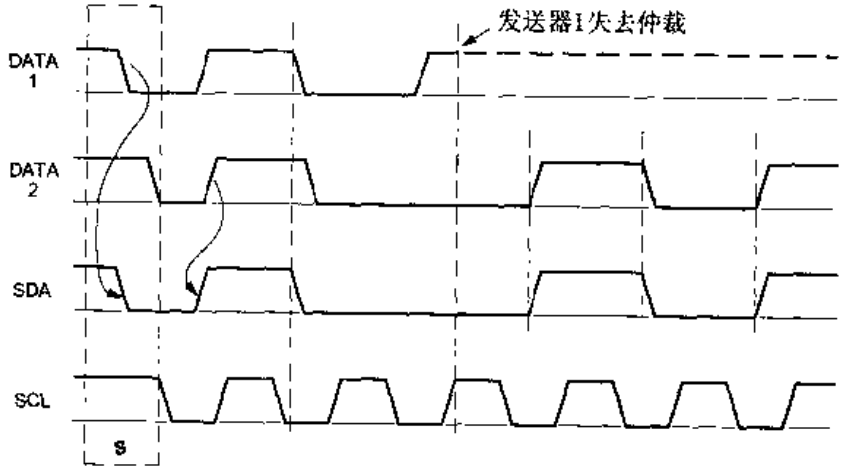


图 12.5　I²C 总线的总裁过程

注意： 在多主竞争仲裁过程时，各主控器向总线输出的时钟频率可以各不相同。这时总线就需要一个统一的时钟脉冲信号。I²C 总线的输出接口采用了线“与”逻辑的双向传输模式。总线的时钟同步就利用这两个特点来实现。

4. I²C 总线的数据传送

I²C 总线上的数据传输主要是以 18 位的字节进行的，其传输过程如图 12.6 所示。图中 1 时刻时字节传送完成，接收器内产生中断信号；2 时刻则为当中断服务处理过程中，接收器保持的低电平信号。

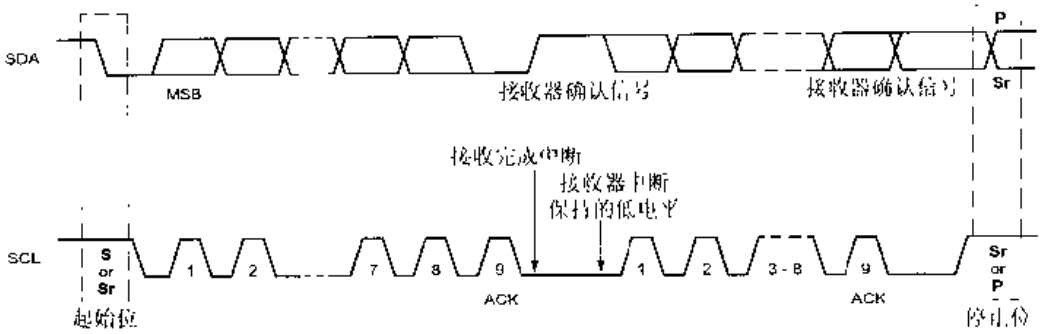


图 12.6 I^2C 总线上的数据传输

在 I^2C 总线上，每个数据的逻辑“0”和逻辑“1”的信号电平取决于相应的正端电压。当 I^2C 总线进行数据传送时，时钟信号的高电平使数据线上的数据保持稳定；而时钟信号在低电平时，数据线上的高电平或低电平状态才允许变化。

在时钟线保持高电平期间，由于数据线由高电平向低电平的变化是一种稳定的状态，所以就将其状态规定为起始条件；而当时钟线保持高电平期间，数据线是由低电平向高电平变化，则规定为停止条件。因此，只有 I^2C 总线中的主控器产生的起始条件和停止条件才能使总线进入“忙”状态或“闲”状态。

在 I^2C 总线上，比特位传送字节的后面都必须有一位应答位，并且数据是以最高有效位首先发出。由于进行数据传输的接收器收到完整的一个数据字节后，有可能还要进行相应的数据处理。因此，接收器也就无法立刻接收下一个字节的数据。为了解决这一问题，I^2C 协议规定：接收器可以通过总线上的时钟保持为低电平，使发送器进入等待状态，直到接收器准备好接收新的数据，并释放时钟线使数据传输继续进行为止。

当一个字节的数据被总线上的另一个已被寻址的接收器接收后，总线上都要求产生一个确认信号，并在这一位时钟信号的高电平期间，使数据保持稳定的低电平状态，从而完成应答确认信号的输出。确认信号通常是指起始信号和停止信号，如果这个信息是一个起始字节，或是总线寻址，则总线上不允许有应答信号产生。如果接收器对被控寻址做出了确认应答，但在数据传输的一段时间以后，又无法继续接收更多的数据，则主控器也将停止数据的继续传送。

图 12.7 为 I^2C 总线的数据传输格式。

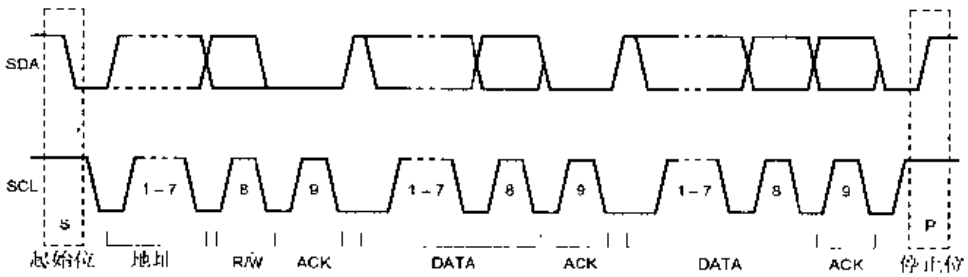


图 12.7 I^2C 总线的数据传输格式

其中 1～7 位为地址位，第 8 位为读/写位；第 9 位为应答位。

I^2C 总线中，数据的传输协议为：

- 起始信号的后面总有一个被控器的地址，被控器的地址一般规定为 7 位的数据。
- 数据的第 8 位是数据的传输方向位，即读/写位。在读/写位中，如果是“0”，则表示主控器向被控机发送数据，也就是执行“写”的功能；如果是“1”，则表示主控器接收被控机发来的数据，也就是执行“读”的功能。
- 数据的传输总是随主控器产生的停止信号而结束。

12.2.2　采用 AT90LS8535 的并行 I/O 口模拟 I^2C 总线

AT90LS8535 单片机只有 32 个 I/O 口，因此在一些复杂的单片机应用系统中，经常需要进行 I/O 的扩展。通常情况下，用户可以采用 8255、8259、74244、74374 等芯片进行 I/O 的外扩，但这样的应用系统较为复杂，且可靠性也不高。

I^2C 标准允许在一条总线上挂接多个设备，因此它可以减少连线、缩小系统体积并降低开发成本。但 AT90LS8535 单片机本身并不具备 I^2C 接口，因此必须通过其并行口上模拟 I^2C 总线，实现对 I^2C 器件的操作。图 12.8 为采用 AT90LS8535 的并行 I/O 口模拟 I^2C 总线的电路原理。

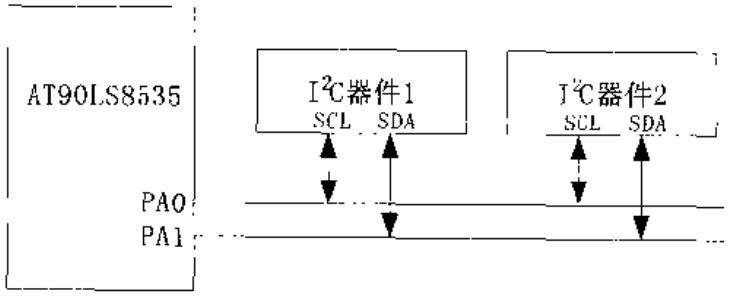


图 12.8　采用 AT90LS8535 的并行 I/O 口模拟 I^2C 总线电路

```
#define uchar unsigned char
//设置系统错误标志(全局变量)
uchar SystemError;

//延时子程序
void delay(void)
{
    uchar i;
    for(i=0;i<100;i++);
    {
;
}
}

//启动通信
//数据线保持高，时钟线从高到低一次跳变，I²C 通信开始
void I2CStart(void)
{
    PORTA|=BIT(PA1);
    PORTA|=BIT(PA0);
```

```
    delay();
    PORTA&=~BIT(PA1);
    delay();
    PORTA&=~BIT(PA0);
}

//停止通信
//数据线保持低，时钟线从低到高一次跳变，I²C通信停止
void I2CStop(void)
{
    PORTA&=~BIT(PA0);
    PORTA&=~BIT(PA1);
    delay();
    PORTA|=BIT(PA0);
    delay();
    PORTA|=BIT(PA1);
}

//等待应答
uchar WaitAck(void)
{
    uchar mid,errtime=255;//因故障接收方无ACK，超时值为255。
    PORTA|=BIT(PA1);
    PORTA|=BIT(PA0);
    SystemError=0x10;
    mid=(PINA&0x02)>>1;
    while(mid)
    {
errtime--;
        if(!errtime)
        {
I2CStop();
            SystemError=0x11;
            return 0;
        }
    }
    PORTA&=~BIT(PA0);
    return 1;
}

//主器件接收，从器件发送(应答信号)
void SendAck(void)
{
    PORTA&=~BIT(PA1);
    delay();
    PORTA|=BIT(PA0);
    delay();
    PORTA&=~BIT(PA0);
}

//主器件接收，从器件发送(非应答信号)
```

```
void SendNotAck(void)
{
    PORTA|=BIT(PA1);
    delay();
    PORTA|=BIT(PA0);
    delay();
    PORTA&=~BIT(PA0);
}

//发送一个字节
void I2CSendByte(uchar x)
{
    uchar i=8,mid;
    while (i--)
    {
        PORTA&=~BIT(PA0);
        delay();
        mid=x&0x80;
        if(mid==0)
            PORTA&=~BIT(PA1);
        else
            PORTA|=BIT(PA1);
        x<<=1;
        delay();
        PORTA|=BIT(PA0);
        delay();
    }
    PORTA&=~BIT(PA0);
}

//接收一个字节
uchar I2CReceiveByte(void)
{
    uchar i=8,data=0,mid;
    PORTA|=BIT(PA1);
    while (i--)
    {
        data<<=1;
        PORTA&=~BIT(PA0);
        delay();
        PORTA|=BIT(PA0);
        delay(); //时钟做一次从低到高的跳变，可以接收数据
        mid=PINA&0x02;
        if(mid==0)
            data|=0;
        else
            data|=1;
    }
    PORTA&=~BIT(PA0);
    return data;
}
```

12.3 CAN 总线

CAN(Controller Area Network)即控制器局域网，是一种用于各种设备检测及控制的串行通信网络。CAN 总线最初是由 BOSCH 公司为汽车的检测、控制系统而设计。由于 CAN 总线在设计过程中采用了许多新的技术和独特的设计思想，这使得它具有了良好实时性、极高的可靠性和很强的现场抗干扰能力。

12.3.1 CAN 总线的特点

具体来讲，CAN 总线有如下特点：

- 结构简单。CAN 为两线形式的现场总线。
- 通信方式灵活。网络上任意一个节点都不分主从，它们之间可以进行点对点、单点对多点及全局广播的方式发送和接收数据。
- 网络上的节点信息具有不同的优先级，用户可以根据实时性的要求予以设置。
- CAN 总线采用短帧格式，每帧字节数最多为 8 个，这样既可满足通常工业领域中的一般要求。同时，每帧数据也不会占用过长的总线时间，从而保证了系统的实时性。
- 通信距离的限制小。CAN 总线的直接通信距离最大可达 10 km，最高通信速率可达 1 Mb/s (此时距离最长为 40 m)；节点数可多达 110 个。
- CAN 总线的通信接口中可完成对通信数据的处理，包括位填充、数据块编码、循环冗余检验、优先级判别等操作。
- CAN 总线采用 CRC 校验方式，并可提供相应的错误处理功能，从而保证了数据通信的可靠。

CAN 总线的以上特点使 CAN 成为工业控制系统中的一种全新的解决方案，CAN 总线已成为最有发展前途的现场总线之一。

12.3.2 CAN 协议的信息格式

CAN 协议支持数据帧、远程帧、出错帧和超载帧 4 种帧格式，其中数据帧和远程帧的发送需要在 CPU 控制下进行，而出错帧和超载帧的发送则是在错误发生或超载发生时由接口芯片自动进行的。因此，用户只需要了解前两个帧的结构形式。数据帧的结构如图 12.9 所示。

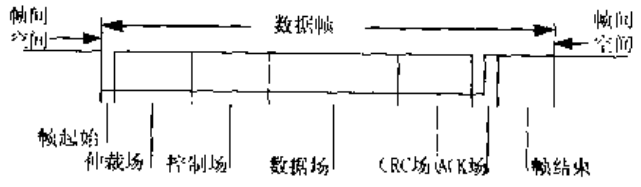


图 12.9 CAN 总线的数据帧结构

一个完整的数据帧格式除仲裁场、控制场和数据场外，其他都是由 CAN 控制器在发送数据时自动加上去的。当采用 SJA1000 作为 CAN 控制器时，写至发送缓冲器的 TXID0 和 TXID1 即设定了相应的仲裁场和控制场。TXID0 为仲裁场的高 8 位，TXID1 的高 3 位为仲裁场的低 3 位。其中 TXID1 的第 5 位为 RTR 位，即远程请求位，该位在数据帧中为“0”；TXID1 的低 4 位用于表示数据场所含字节的个数，称为 DLC。控制场就是由 RTR 与 DLC 共同构成的，而数据场则是由最多不超过 8 个字节的有效数据组成。

远程帧与数据帧的形式差别在于：远程帧没有数据场，除此之外，远程帧中的 RTR 位必须置为“1”，表示请求数据源节点向它的目的节点发送数据，而数据帧的 RTR 位则必须置为“0”。

12.3.3　CAN 控制器 SJA1000

SJA1000 芯片是 Philips 公司生产的 CAN 控制器，它是一种基于内存编址的微控制器。该设备的独立操作是通过像 RAM 一样的片内寄存器的修改来实现的。

1. SJA1000 简介

SJA1000 是早期 CAN 控制器 PCA82C200 的替代品，它具有如下一些特点：

- 完全兼容 PCA82C200 及其工作模式，即 BASIC CAN 模式。
- 有扩展的接收缓冲器，64 字节的 FIFO 结构。
- 支持 CAN2.0B。
- 位速率可达 1Mb/s。
- 支持 peliCAN 模式及其扩展功能。
- 支持与不同微处理器的接口。
- 可编程的 CAN 输出驱动配置。

2. SJA1000 的引脚和内部结构

图 12.10 是 SJA1000 的引脚图。

各引脚功能：

- VDD1：+5 V 电压。
- VDD2：输入比较器的+5 V 电压。
- VDD3：输出驱动的+5 V 电压源。
- VSS1：接地。
- VSS2：输入比较器的接地端。
- VSS3：输出驱动器接地。
- TX0：CAN 输出 0。
- TX1：CAN 输出 1。
- XTAL1、XTAL2：外部振荡信号输入。
- AD0~AD7：地址/数据总线。
- ALE/AS：ALE 输入信号(Intel 模式)/AS 输入

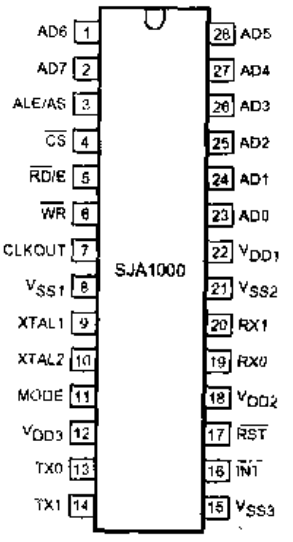


图 12.10　SJA1000 的引脚图

信号(Motorola 模式)。

- /CS：片选信号。
- /RD/E：读信号(Intel 模式)或使能信号(Motorola 模式)。
- /WR：写信号(Intel 模式)或读/写信号(Motorola 模式)。
- CLKOUT：SJA1000 产生的时钟信号。
- MODE：模式选择输入(1=Intel 模式，0=Motorola 模式)。
- RX0、RX1：CAN 输入。
- /RST：复位输入。
- /INT：中断输出。

图 12.11 所示为 SJA1000 的内部结构图。

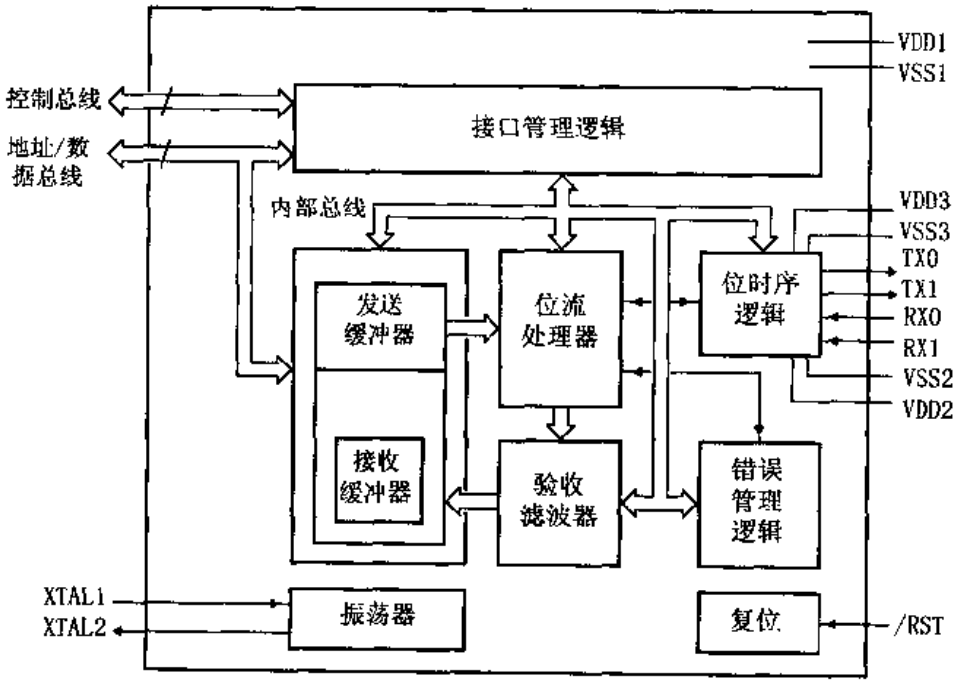


图 12.11 SJA1000 的内部结构

3. SJA1000 控制器各功能模块说明

- 接口管理逻辑(IML)

  SJA1000 的接口管理逻辑用来解释来自单片机的命令，控制 CAN 寄存器的寻址，并向主控制器提供总线的各种状态信息。

- 发送缓冲器(TXB)

  发送缓冲器是单片机和 SJA1000 的位流处理器之间的接口，它能够存储发送到 CAN 总线上的数据流。发送缓冲器具有 13 个字节长度。

- 接收缓冲器(RXB)

  接收缓冲器是 CAN 控制器用于储存从 CAN 总线上接收到的信息的存储器，它是验收滤波器和单片机之间的接口。发送缓冲器具有 13 个字节长度。

- 验收滤波器(ACF)

  验收滤波器可以把存储在其寄存器中的数据和控制器从 CAN 总线上接收到的识

别码的内容做比较，并决定信息是否接收信息。

- 位流处理器(BSP)
  位流处理器是一个在发送缓冲器和 CAN 总线之间的程序装置，它用于控制数据流，并执行错误检测、仲裁填充以及错误处理。
- 位时序逻辑(BTL)
  位时序逻辑用于监视 CAN 总线的时序逻辑，并处理与总线有关的位时序，它提供 CAN 总线位流的硬同步和传送过程中的软同步。
- 错误管理逻辑(EML)
  错误管理逻辑可对位流处理器和接口管理逻辑的错误进行统计。

4. SJA1000 的寄存器结构及地址分配

由于 SJA1000 是一种基于内存编址的微控制器，设备对它的独立操作都是通过修改其片内寄存器来实现的，因此在使用 CAN 总线前，必须设置 SJA1000 的各个寄存器参数。

SJA1000 具有 10 个控制寄存器、10 个发送缓冲寄存器和 10 个接收缓冲寄存器。各部分的功能分别介绍如下：

- 控制寄存器(CR)
  地址：0
  控制寄存器用于改变 CAN 控制器的运行方式，这些位可以被微控制器置位或清 0。微控制器可以对这个寄存器进行读/写操作，控制寄存器的各位功能如表 12.1 所示。

表 12.1　控制寄存器的各位功能

位	符　号	名　称	功　能
CR.7			保留
CR.6			保留
CR.5			保留
CR.4	OIE	溢出中断使能	如果该位为“1”，则接收溢出中断信号； 如果该位为“0”，则不接收溢出中断信号。
CR.3	EIE	错误中断使能	如果该位为“1”，则接收错误中断信号； 如果该位为“0”，则不接收错误中断信号。
CR.2	TIE	发送中断使能	如果该位为“1”，则接收一个发送中断信号； 如果该位为“0”，则不接收发送中断信号。
CR.1	RIE	接收中断使能	如果该位为“1”，则接收一个接收中断信号； 如果该位为“0”，则不接收发送中断信号。
CR.0	RR	复位请求	如果该位为“1”，则 SJA1000 一旦检测到复位请求后,就中止当前发送/接收的信息,进入复位模式 该位为“0”时，复位请求位接收到一个下降沿后，系统回到工作模式。

- 命令寄存器(CMR)

地址：1

微控制器只能对命令寄存器进行写操作。如果对该地址进行读操作的话，系统会返回值 0xff。此外，微控制器对命令寄存器访问的两条命令之间至少还应有一个内部时钟周期的间隔。命令寄存器的各位功能如表 12.2 所示。

表 12.2 命令寄存器的各位功能

位	符 号	名 称	功 能
CMR.7			保留
CMR.6			保留
CMR.5			保留
CMR.4	GTS	睡眠模式	该位为“1”时，若没有总线活动，则进入睡眠模式； 该位为“0”时，总线进入正常工作模式
CMR.3	CDO	数据溢出清除	该位为“1”，则清除数据溢出状态； 该位为“0”，则不做操作
CMR.2	RRB	释放接收缓冲器	该位为“1”时，接收缓冲器释放内存空间； 该位为“0”，则不做操作
CMR.1	AT	中止发送	该位为“1”时，取消等待处理的发送请求； 该位为“0”，则不做操作
CMR.0	TR	发送请求	该位为“1”时，发送当前信息； 该位为“0”，则不做操作

- 状态寄存器(SR)

地址：2

状态寄存器的各位用于反映了 CAN 控制器当前的工作状态，微控制器只能对它进行读操作，状态寄存器的各位功能如表 12.3 所示。

表 12.3 状态寄存器的各位功能

位	符 号	名 称	功 能
SR.7	BS	总线状态	该位为“1”，退出总线活动； 该位为“0”，总线开启
SR.6	ES	出错状态	该位为“1”，表明至少有一个错误计数器超过报警限制； 该位为“0”，表明两个错误计数器都在报警限制以下
SR.5	TS	发送状态	该位为“1”，表明正在传送信息； 该位为“0”，表示系统空闲
SR.4	RS	接收状态	该位为“1”，表明正在接收信息； 该位为“0”，表示系统空闲
SR.3	TCS	发送完毕状态	该位为“1”，表明最近一次发送成功； 该位为“0”，表明当前发送请求还未处理完毕
SR.2	TBS	发送缓冲器状态	该位为“1”时，CPU 可以向发送缓冲器写入信息； 该位为“0”时，CPU 不能访问发送缓冲器

续表

位	符　号	名　称	功　能
SR.1	DOS	数据溢出状态	该位为“1”，表明信息出现溢出错误； 该位为“0”，表明无数据溢出发生
SR.0	RBS	接收缓冲器 状态	该位为“1”，表明 RXFIFO 中有可用信息； 该位为“0”，表明无可用信息

- 中断寄存器(IR)

  地址：3

  中断寄存器使微控制器能够对中断源进行识别。当该寄存器的一位或多位被置位时，芯片的中断引脚向微控制器产生一个中断请求。微控制器读过中断寄存器后，中断寄存器会被复位到 0x00。中断寄存器也只能进行读操作，中断寄存器的各位功能如表 12.4 所示。。

表 12.4　中断寄存器的各位功能

位	符　号	名　称	功　能
IR.7			保留
IR.6			保留
IR.5			保留
IR.4	WUI	唤醒中断	系统退出睡眠模式时此位被置位； 控制器的任何读访问清除此位
IR.3	DOI	数据溢出中断	溢出中断使能时，数据溢出状态位上跳变化，此位被置位； 控制器的任何读访问将清除此位
IR.2	EI	错误中断	错误中断使能时，错误状态位变化，此位被置位； 控制器的任何读访问将清除此位
IR.1	TI	发送中断	发送中断使能时，发送缓冲器状态上跳变化，此位被置位； 微控制器的任何读访问将清除此位
IR.0	RI	接收中断	当接收缓冲器不空，且接收中断使能时，此位被置位； 微控制器的任何读访问将清除此位

- 验收滤波器

  通过验收滤波器，CAN 控制器可以确定哪些数据可以接收，而哪些数据不需要接收。首先 CAN 控制器接收 CAN 总线上的数据，然后将所接收到的识别码同验收滤波器中设定的值相比较，如果符合要求，就将产生信息存放入接收缓冲器，否则就舍弃。验收滤波器包含验收代码寄存器和验收屏蔽寄存器两个部分，各部分的功能如下：

  - 验收代码寄存器(ACR)，其各位功能如表 12.5 所示。

    地址：4
  - 验收屏蔽寄存器(AMR)，其各位功能如表 12.6 所示。

    地址：5

表 12.5 验收代码寄存器的各位功能

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
AC.7	AC.6	AC.5	AC.4	AC.3	AC.2	AC.1	AC.0

表 12.6 验收屏蔽寄存器的各位功能

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
AM.7	AM.6	AM.5	AM.4	AM.3	AM.2	AM.1	AM.0

当验收滤波器和 CAN 总线上的识别码满足以下关系时，数据被接收。

验收代码寄存器的 AC.7～AC.0 和 CAN 总线上的识别码的 ID.10～ID.3 相等且与验收屏蔽寄存器中的 AM.7~AM.0 相“或”为 0xff，即 ID.10～ID.3＝AC.7～AC.0 且 ID.10~ID.3∨AM.7～AM.0＝0xff。

- 发送缓冲区

  地址：10～19

  发送缓冲区用来存储微控制器要发送到 CAN 网络上的信息，它包含描述符区和数据区两个部分，数据描述符用来表示接收信息的 CAN 网络上的结点编号，而数据区则是要在 CAN 网络上传递的数据。发送缓冲器对微控制器是可读/写的，其各位功能如表 12.7 所示。

表 12.7 发送缓冲区的各位功能

CAN 地址	区	名 称	位							
10	描述符	识别码字节 1	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3
11	描述符	识别码字节 2	ID.2	ID.1	ID.0	RTR	DLC.3	DLC.2	DLC.1	DLC.0
12~19	数据	TX 数据	发送数据							

  - 识别码(ID)

    识别码用于标识数据的接收结点，它包含 ID0～ID10 共 11 位数据。其中最高位的 D10 首先被发送到总线上。识别码除了确定访问结点外，它还可以在仲裁期间决定总线访问的优先级顺序。

  - 远程发送请求位(RTR)

    远程发送请求位用于决定当前 CAN 网络要发送的是远程帧还是数据帧。如果此位为“1”，则总线上将要发送的数据为远程帧，如果该位为“0”，则总线要以数据帧格式发送数据。

  - 数据长度码(DLC)

    数据长度码可确定将要发送数据的字节数。

  - 数据区

    数据区用于存放将要传送的数据字节。而发送的字节数则由数据长度码决定。

- 接收缓冲器

  地址：20～29

接收缓冲器用于存放经验收滤波器检验通过的数据字节。

12.3.4　AT90LS8535 与 SJA1000 的接口及编程

图 12.12 是 AT90LS8535 与 SJA1000 的接口电路。通过这个电路，AT90LS8535 不仅可以与 CAN 总线器件进行通信，而且也可与其他单片机进行通信(只需要将两个相同电路通过 CAN 总线连接就可实现两个单片机之间的数据传送)。

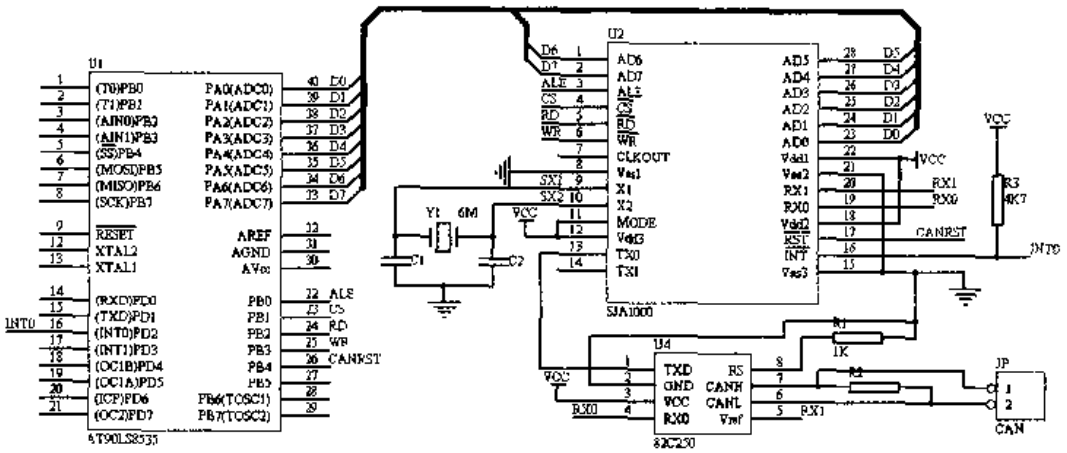


图 12.12　AT90LS8535 与 SJA1000 的接口电路

图中的 82C250 芯片用于实现 CAN 控制器 SJA1000 与 CAN 的物理总线之间的连接。SJA1000 工作于中断模式，每当有数据的接收、发送或数据错误时，控制器向单片机发出中断请求，单片机在随后的中断服务程序中对 CAN 控制器进行操作，从而实现对 CAN 总线的控制。

电路的子程序如下：

```
#define uchar unsigned char
#define uint unsigned int
#define CAN_CR 0x00
#define CAN_CMR 0x01
#define CAN_SR 0x02
#define CAN_IR 0x03
#define CAN_ACR 0x04

#define CAN_AMR 0x05
#define CAN_BTR0 [0x06
#define CAN_BTR1 0x07
#define CAN_OCR 0x08
#define CAN_SendDataBase 0x0A
#define CAN_ReceiveDataBase 0x14

//CAN 总线波特率 1 Mb/s
```

```
#define CAN_BAUDRATE1M 1
//发送完成标志
uchar CAN_SEND_COMPLETE;

//以address为地址，从sja1000中读取一个字节
uchar BYTEREAD(uchar address)
{
    uchar mid;
    PORTB&=~BIT(PB1);    //片选信号有效
    PORTB|=BIT(PB0);     //置ALE为1
    DDRA=0xff;           //设置A口输出
    PORTA=address;       //A口输出地址
    PORTB&=~BIT(PB0);    //置ALE为0
    DDRA=0;              //设置A口为输入
    PORTB&=~BIT(PB2);    //读信号有效
    mid=PINA;
    PORTB|=BIT(PB2);     //读信号置1
    PORTB|=BIT(PB1);     //片选信号置1
    return mid;
}

//以address为地址，向sja1000中写入一个字节
void BYTEWRITE(uchar address,uchar x)
{
    uchar mid;
    PORTB&=~BIT(PB1);    //片选信号有效
    PORTB|=BIT(PB0);     //置ALE为1
    DDRA=0xff;           //设置A口输出
    PORTA=address;       //A口输出地址
    PORTB&=~BIT(PB0);    //置ALE为0
    PORTB&=~BIT(PB2);    //写信号有效
    PORTA=x;             //A口输出数据
    PORTB|=BIT(PB2);     //写信号置1
    PORTB|=BIT(PB1);     //片选信号置1
}

//CAN总线的复位子程序
//Resetstyle=1则启动中断，否则禁止中断
uchar CAN_Reset(uchar ResetStyle)
{
    PORTB&=~BIT(PB4);    //CAN总线复位
    PORTB|=BIT(PB4);
    GIMSK=0;             //禁止中断
    CAN_QUEUE_HEAD = 0;
    CAN_QUEUE_TAIL = 0;
    BYTEWRITE(CAN_CR, 0x21);
    while(((BYTEREAD(CAN_CR))&0x01)==0)
    {
        BYTEWRITE(CAN_CR, 0x21);
    }
    //设置控制寄存器
```

```
    BYTEWRITE(CAN_CMR, 0x0E);
    BYTEWRITE(CAN_ACR, 0x00);
    BYTEWRITE(CAN_AMR, 0xff);
    BYTEWRITE(CAN_BTR0, 0x80);
    BYTEWRITE(CAN_BTR1, 0x23);
    BYTEWRITE(CAN_OCR, 0xfa);
    BYTEWRITE(CAN_CR, 0x1F);
    BYTEWRITE(CAN_CR, 0x1E);
    while(((BYTEREAD(CAN_CR))&0x01)==1)
    {
        BYTEWRITE(CAN_CR,0x1E);
    }
    if(ResetStyle)
    {
        GIMSK=0x40;
    }
    return(1);
}

//设置 CAN 通信参数
uchar CAN_Config(uchar BaudRate, uchar ReceiveCode,uchar ReceiveMask)
{
    BYTEWRITE(CAN_CR, 0x21);
    while(((BYTEREAD(CAN_CR))&0x01)==0)
    {
        BYTEWRITE(CAN_CR, 0x21);
    }
    BYTEWRITE(CAN_ACR, ReceiveCode);
    BYTEWRITE(CAN_AMR, ReceiveMask);
    switch(BaudRate)
    {
        case CAN_BAUDRATE1M:
            BYTEWRITE(CAN_BTR0, 0x80);
            BYTEWRITE(CAN_BTR1, 0x23);
            break;
        default:
            break;
    }
    BYTEWRITE(CAN_CR, 0x1E);
    while(((BYTEREAD(CAN_CR))&0x01)==1)
    {
        BYTEWRITE(CAN_CR, 0x1E);
    }
    return(1);
}

//发送缓冲器中的数据
uchar CAN_SendMessage(uchar *DataPointer, uchar ReturnStyle)
{
    uint i;
    while(CAN_SEND_COMPLETE==0)
```

```
    {

    }
    CAN_SEND_COMPLETE=0;
    BYTEWRITE(CAN_SendDataBase, DataPointer[0]);
    BYTEWRITE(CAN_SendDataBase+1, DataPointer[1]);
    BYTEWRITE(CAN_SendDataBase+2, DataPointer[2]);
    BYTEWRITE(CAN_SendDataBase+3, DataPointer[3]);
    BYTEWRITE(CAN_SendDataBase+4, DataPointer[4]);
    BYTEWRITE(CAN_SendDataBase+5, DataPointer[5]);
    BYTEWRITE(CAN_SendDataBase+6, DataPointer[6]);
    BYTEWRITE(CAN_SendDataBase+7, DataPointer[7]);
    BYTEWRITE(CAN_SendDataBase+8, DataPointer[8]);
    BYTEWRITE(CAN_SendDataBase+9, DataPointer[9]);
    BYTEWRITE(CAN_CMR, 0x01);  //Request to send
    if(ReturnStyle)
    {
        for(i=0;i<1000;i++) //poll for finish
        {
            if(CAN_SEND_COMPLETE)
                break;
        }
    }
    return(i);
}

//CAN中断服务程序
void CAN_Interrupt(void) interrupt 0
{
    unsigned char InterruptValue;
    unsigned char temp;

    InterruptValue=BYTEREAD(CAN_IR);
    if(InterruptValue&0x01) //接收中断
    {
        DataPointer[0]=BYTEREAD(CAN_ReceiveDataBase);
        DataPointer[1]=BYTEREAD(CAN_ReceiveDataBase+1);
        DataPointer[2]=BYTEREAD(CAN_ReceiveDataBase+2);
        DataPointer[3]=BYTEREAD(CAN_ReceiveDataBase+3);
        DataPointer[4]=BYTEREAD(CAN_ReceiveDataBase+4);
        DataPointer[5]=BYTEREAD(CAN_ReceiveDataBase+5);
        DataPointer[6]=BYTEREAD(CAN_ReceiveDataBase+6);
        DataPointer[7]=BYTEREAD(CAN_ReceiveDataBase+7);
        DataPointer[8]=BYTEREAD(CAN_ReceiveDataBase+8);
        DataPointer[9]=BYTEREAD(CAN_ReceiveDataBase+9);
    }   BYTEWRITE(CAN_CMR, 0x0C);
    if(InterruptValue&0x02)//发送中断
    {
        CAN_SEND_COMPLETE=1;
    }
}
```

```
//读取 CAN 寄存器数据
unsigned char CAN_ReadRegister(unsigned char RegIndex)
{
    return(BYTEREAD(RegIndex));
}
```

12.4　AT90LS8535 单片机与 PC 的串行通信

工业控制领域中，经常要涉及到串行通信的问题。为了在 PC 和单片机之间进行串行数据通信，用户可以采用多种方法来实现。如在 DOS 下的汇编语言、Basic 语言和 C 语言，在 Windows 下的 Visual C++ 6.0、Visual Basic 6.0 等。由于 VC++6.0 较其他的编程语言具有较多的优点，本节讨论用 VC++6.0 开发单片机与 PC 的串口通信程序。

12.4.1　基于 VC++6.0 的 PC 串口通信

VC++6.0 是微软公司推出的一种集成开发环境，它具有强大的功能和友好的开发界面，因此被广泛应用于各个领域。应用 VC++开发串行通信目前通常有如下 4 种方法：

- 利用 Windows API 通信函数。
- 利用 VC 的标准通信函数_inp、_inpw、_inpd、_outp、_outpw、_outpd 等直接对串口进行操作。
- 使用 Microsoft Visual C++的通信控件(MSComm)。
- 利用串行通信类。

以上几种方法中第一种的使用面较广，但编程复杂，专业化程度高，使用困难。第二种方法需要了解硬件电路结构原理。第三种方法要使用令人费解的 VARIANT 类，因此也不易使用。第四种方法是只要理解这种类的几个成员函数，使用较为方便。以下给出实现串行通信的几种方法。

1. Windows API 通信函数方法

与串行通信有关的 Windows API 函数有：

- CreateFile()用 COMn 打开串口。
- ReadFile()读串口。
- WriteFile()写串口。
- CloseHandle()关闭串口句柄。初始化时应注意 CreateFile()函数中串口共享方式应设为 0，串口为不可共享设备，其他与一般文件读写类似。以下给出 API 实现的源代码。

  发送源代码：

```
//声明全局变量
HANDLE m_hIDComDev;
OVERLAPPED m_OverlappedRead, m_Over lappedWrite;
//初始化串口
```

```
void CSerialAPIView::OnInitialUpdate()
{
    CView::OnInitialUpdate();
    Char szComParams[50];
    DCB dcb;
    Memset(&m_OverlappedRead, 0, sizeof (OVERLAPPED));
    Memset(&m_OverlappedWrite, 0, sizeof (OVERLAPPED));
    m_hIDComDev = NULL;
    m_hIDComDev = CreateFile("COM2",GENERIC_READ | GENERIC_WRITE, 0,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
NULL);
    if(m_hIDComDev==NULL)
    {
        AfxMessageBox("Can not open serial port!");
        goto endd;
    }
    memset(&m_OverlappedRead, 0, sizeof (OVERLAPPED));
    memset(&m_OverlappedWrite, 0, sizeof (OVERLAPPED));
    COMMTIMEOUTS CommTimeOuts;
    CommTimeOuts. ReadIntervalTimeout=0×FFFFFFFF;
    CommTimeOuts. ReadTotalTimeoutMultiplier = 0;
    CommTimeOuts. ReadTotalTimeoutConstant = 0;
    CommTimeOuts. WriteTotalTimeoutMultiplier = 0;
    CommTimeOuts. WriteTotalTimeoutConstant = 5000;
    SetCommTimeouts(m_hIDComDev, &CommTimeOuts);
    Wsprintf(szComparams, "COM2:9600, n, 8, 1");
    m_OverlappedRead. hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    m_OverlappedWrite. hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    dcb. DCBlength = sizeof(DCB);
    GetCommState(m_hIDComDev, &dcb);
    dcb. BaudRate = 9600;
    dcb. ByteSize= 8;
    unsigned char ucSet;
    ucSet = (unsigned char) ((FC_RTSCTS&FC_DTRDSR) != 0);
    ucSet = (unsigned char) ((FC_RTSCTS&FC_RTSCTS) ! = 0);
    ucSet = (unsigned char) ((FC_RTSCTS&FC_XONXOFF) ! = 0);
    if (!SetCommState(m_hIDComDev, &dcb) ||
        !SetupComm(m_hIDComDev,10000,10000) ||
        m_OverlappedRead. hEvent ==NULL ||
        m_OverlappedWrite. hEvent ==NULL)
    {
        DWORD dwError = GetLastError();
        if (m_OverlappedRead. hEvent != NULL)
            CloseHandle(m_OverlappedRead. hEvent);
        if (m_OverlappedWrite. hEvent != NULL)
            CloseHandle(m_OverlappedWrite. hEvent);
            CloseHandle(m_hIDComDev);
    }
endd:
;
}
```

```
//发送数据
void CSerialAPIView::OnSend()
{
    char szMessage[20] = "hello world";
    DWORD dwBytesWritten;
    for (int i=0; i<sizeof(szMessage); i++)
    {
        WriteFile(m_hIDComDev, (LPSTR)&szMessage[i], 1,
    &dwBytesWritten, &m_OverlappedWrite);
        if(WaitForSingleObject(m_OverlapperWrite, hEvent, 1000))
                dwBytesWritten = 0;
        else{
                GentOverlappedResult(m_hIDComDev, &m_OverlappedWrite,
            &dwBytesWritten, FALSE);
            m_OverlappedWrite. Offset += dwBytesWritten;
        }
        dwBytesWritten++;
    }
}
```

接收源代码：

```
DCB ComDcb; //设备控制块
HANDLE hCom; //global handle
hCom = CreateFile ("COM1",GENERIC_READ| GENERIC_WRITE,0,
NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
if (hCom==INVALID_HANDLE_VALUE)
{
    AfxMessageBox("无法打开串行口");
}
else
{
    COMMTIMEOUTS CommTimeOuts ;
    SetCommMask(hCom, EV_RXCHAR ) ;
    SetupComm(hCom, 4096, 4096 ) ;
    PurgeComm(hCom, PURGE_TXABORT| PURGE_RXABORT |
    PURGE_TXCLEAR| PURGE_RXCLEAR ) ;
    //以下初始化结构变量 CommTimeOuts
    CommTimeOuts.ReadIntervalTimeout = 0xFFFFFFFF ;
    CommTimeOuts.ReadTotalTimeoutMultiplier = 0 ;
    CommTimeOuts.ReadTotalTimeoutConstant = 4000 ;
    CommTimeOuts.WriteTotalTimeoutMultiplier = 0;
    CommTimeOuts.WriteTotalTimeoutConstant = 4000 ;
    SetCommTimeouts(hCom, &CommTimeOuts );
    ComDcb.DCBlength = sizeof( DCB ) ;
    GetCommState( hCom, &ComDcb ) ;
    ComDcb.BaudRate =9600; //设置波特率
    ComDcb.ByteSize = 8; //数据位长度
    ComDcb.Parity = 0;
    SetCommState(hCom, &ComDcb ) ;
```

```
} //设置通信参数
DWORD dRead,dReadNum;
unsigned char buft [200];
dRead=ReadFile(hCom, buff, 100, &dReadNum, NULL); //接收100个字符,
//cReadNum为实际接收字节数
```

2. 利用端口操作函数直接

这种方式主要是采用两个端口函数_inp()和_outp()实现对串口的读写，其中读端口的函数原型为：

```
int _inp(port)
```

该函数可从端口读取一个字节(端口号为0~65535)。

写端口的函数原型为：

```
int _outp(port, data)
```

该函数可向指定端口写入一个字节的数据。

由于这种串口访问方式比较复杂，本书不予以讨论。

3. MSComm控件

MSComm控件是微软开发的专用通信控件，封装了串口的所有功能，使用很方便，但在实际应用中要小心对其属性进行配置。下面详细说明该类应用方法。

- MSComm控件的属性

  CommPort：设置串口号。

  Settings：设置串口通信参数，包括波特率、奇偶性(无校验、偶校验、奇校验)、字节有效位数以及S停止位。

  PortOpen：设置或返回串口状态。

  InputMode：设置从接收缓冲区读取数据的格式。

  Input：从接收缓冲区读取数据。

  InBufferCount：接收缓冲区中的字节数。

  InBufferSize：接收缓冲区的大小。

  Output：向发送缓冲区写入数据。

  OutBufferCount：发送缓冲区中的字节数。

  OutBufferSize：发送缓冲区的大小。

  InputLen：设置或返回Input读出的字节数。

  CommEvent：串口事件。

- 程序示例

```
//串口初始化
if (!m_comm.GetPortOpen())
m_comm.SetPortOpen(TURE); //打开串口
m_comm.SetSettings("9600,n,8,1"); //设置串口参数
m_comm.SetInputMode(0);
```

```
m_comm.SetRthresHold(1);

//接收数据
m_comm.SetInputLen(1);
VARINAT V1=m_comm.GetInput();

//读入字符
m_V1=V1.bstrval;

//发送字符
m_comm.SetOutput(Colevariant ("welcome"); //发送 "welcome"
```

注意： SetOutput 可以发送文本数据或二进制数据。用 SetOutput 方法传输文本数据时，必须首先定义一个包含一个字符串的 Variant 类型；发送二进制数据时，必须传递一个包含字节数组的 Variant 到 Output 属性。通常，如果发送一个字符串到应用程序，可以以文本数据的形式发送。如果发送包含嵌入控制字符、Null 字符等的数据，则要以二进制的形式发送。

4. VC++类 CSerial

- CSerial 简介

  Cserial 类是由一个免费的 VC++类，它可方便地实现对串行通信的支持。以下为该类定义的说明部分。

```
class CSerial
{
    public:
    CSerial();
    ~CSerial();
    BOOL Open( int nPort = 2, int nBaud = 9600 );
    BOOL Close( void );
    int ReadData( void *, int );
    int SendData( const char *, int );
    int ReadDataWaiting( void );
    BOOL IsOpened( void ){ return( m_bOpened ); }
    protected:
    BOOL WriteCommByte( unsigned char );
    HANDLE m_hIDComDev;
    OVERLAPPED m_OverlappedRead, m_OverlappedWrite;
    BOOL m_bOpened;
}
```

- Cserial 类的成员函数

  Cserial::Cserial 是构造函数，用于对类变量进行初始化。

  CSerial::Open 用于打开串行端口，它带有两个参数，一个是串行端口号，另一个是波特率，函数返回一个布尔量。

  CSerial::Close 可关闭串行通信端口。Cserial 的析构函数会自动调用这个函数，所以可不用显式调用该函数。

CSerial::SendData 可把数据从一个缓冲区写到串行端口，它的第一个参数是缓冲区的指针，其中包含要被发送的资料。

CSerial::ReadDataWaiting 用于返回通信端口缓冲区中的数据。

CSerial::ReadData 用于从接收缓冲区中读入数据，它的一个参数是缓冲区指针；另一个参数用于指定缓冲区的大小。

12.4.2 应用实例

串行通信的内容包括如下两部分：单片机向上传送数据和 PC 机发送控制命令。

PC 机源程序：

```
void CCommDlg::OnSend()
{
    CSerial Serial;
    //构造串口类，初始化串行口
    if (Serial.Open(2,9600)) //if-1
    //打开串行口 2，波特率为 9600bps
{
    static char szMessage[]="0";
    //命令码(可定义各种命令码)
    int nBytesSent;
    int count=0;
    resend:
    nBytesSent=Serial.SendData(szMessage,strlen(szMessage));
    //发送命令码
    char rdMessage [20];
    if (Serial.ReadDataWaiting())
    {
        Serial.ReadData(rdMessage,88);
        //rdMessage 定义接收字节存储区，为全局变量//
        if ((rdMessage[0]!=0x7f)&&(count<3))
        {
            count++;
            goto resend
    }
        if(count>=3)
        MessageBox("发送命令字失败");
        }
    else
        MessageBox("接收数据错误");
        }
    else
        MessageBox("串行口打开失败");
}
```

下位机通信程序：

```
#include "io8535.h"
```

```
#include "macros.h"

#include "stdio.h"

#pragma interrupt_handler UART_Receive:12

//接收到的数据
 char rdata=0;

//发送数据标志
char TXFLAG=0;

//串口接收中断
void UART_Receive(void)
{
    rdata=getchar();
    if((rdata=='T') |(rdata=='S'))
        TXFLAG=1;
    if((rdata=='R') |(rdata=='M'))
        TXFLAG=1;
    if(rdata=='D')
        TXFLAG=1;
}

//串口发送数据
void USART_Transmit( int data )
{
    /* 等待寄存器空 */
    while(!( USA & (1<<UDRE)) )
        ;
    /* 把数据传送至发送缓冲器*/
    UDR = (char)(data>>8)&0xff;

    while(!( USR & (1<<UDRE)) )
        ;
    UDR=data&0xff;
    TXFLAG=0;   //清除发送标志
}

//主程序
void main()
{
    UBRRH = 0;              //设置波特率 9600
    UBRRL=38;
      USR=0x98;             //开接收中断，发送，接收使能
    SREG=0x80;              //开中断
    do{
        if(TXFLAG==1){
            if(rdata=='T')
                USART_Transmit(0xffff);
```

```
            if(rdata=='M')
                USART_Transmit(0xeeee);

            if(rdata=='D')
                USART_Transmit();
            if(rdata=='R')
                USART_Transmit(0xdddd);
            if(rdata=='S')
                USART_Transmit(0xcccc);
                }
    }while(1);
}
```

第 13 章　系统设计中的程序处理方法

系统的程序处理方法是在保持单片机系统硬件不变的基础上，通过程序的方法使应用系统获得更高性能的一种手段。通常，在单片机处理模拟信号输入时，存在两个重要的问题：一是信号干扰的排除，一是信号的转换。干扰问题的解决需要数字滤波技术，而模拟信号向数字信号的转换可以通过各种非线性信号处理方法实现。

13.1　数字滤波处理

一般的单片机应用系统要用来处理输入的模拟信号，但是一般单片机应用系统输入的模拟信号都含有各种噪音和干扰。这些噪声和干扰都来源于被测信号本身、各种传感器等等。为了进行精确地控制，必须将信号中的噪音和干扰消除掉。为了消除干扰，即可以通过模拟滤波器对信号实现频率滤波，也可以用软件的方法有效地消除其影响，这就是运用 CPU 的运算、控制功能完成模拟滤波器类似功能的数字滤波技术。

所谓数字滤波，就是通过一定的计算或判断程序减少干扰在有用信号中的比重，实际上是一种程序滤波。数字滤波器和模拟滤波器比较有自己的优点，首先数字滤波是通过程序实现的，不用另外设计硬件电路，降低了设计成本，并且更加可靠、更加稳定。另外模拟滤波器只能对频率相对较高的信号进行滤波，而数字滤波器可以对频率很低的信号进行滤波，克服了模拟滤波器的这一缺点。数字滤波可以根据不同的信号采用不同的滤波方法，所以使用起来更灵活、更方便，因此，数字滤波在单片机应用系统中得到了广泛的应用。

13.1.1　平滑滤波

所谓平滑滤波就是采用求平均值的方法消除随机误差的方法。在很多情况下，掺杂在有用信号中的随机干扰可以认为是白噪声。它的一个统计平均值为 0，因此可以采用平滑滤波的方法消除。

1. 算术平均值法

这种滤波的具体做法是：以第一次采样时刻为基准，向后递推 N 个周期，单片机存储 N 个周期的采样值。当累计采样了 N 个周期的数据后，进行累加运算，并求平均值，将此平均值作为真实采样值送入单片机进行运算。

算术平均值法适用于对一般具有随机干扰的信号进行滤波，这种信号的特点是有一个平均值，信号在某一个数值范围上下波动，此时仅取一个采样值作为依据显然是不准确的，如压力、流量、液平面等信号的测量。但对脉冲性干扰的平滑作用尚不理想，因此不适用于脉冲性干扰比较严重的场合。算术平均值法对信号的平滑滤波程度完全取决于测量次数。

当测量次数较大时，平滑度高，但灵敏度低，即外界信号的变化对测量计算结果Y的影响小；当测量次数较小时，平滑度低，但灵敏度高。应视具体情况选取测量次数，以便既少占用计算时间，又达到最好的效果。

2. 加权平均值法

算术平均值法对每次采样值给出相同的加权系数，即1/N，N为采样次数。但有些场合为了改进滤波效果，提高系统对当前所受干扰的灵敏度，需要增加新采样值在平均值中的比重，即将各采样值取不同的比例，然后再相加，此方法称为加权平均值法。

加权平均值法适用于系统纯滞后时间常数较大、采样周期较短的过程，它给不同的相对采样时间得到的采样值以不同的权系数，以便能迅速反应系统当前所受干扰的严重程度。但采用加权平均值法需要测试不同过程的纯滞后时间，同时要不断计算各权系数，增加了计算量，降低了控制速度。

3. 递推平均值法

以上平均滤波算法有一个共同点，即每计算1次有效采样值必须连续采样N次。对于采样速度较慢或要求数据计算速率较高的实时系统，这些方法是无法使用的。递推平均值法只采样1次，将本次采样值和以前的N-1次采样值一起求平均，得到当前的有效采样值。滑动平均值法把N个采样数据看成一个队列，队列的长度固定为N，每进行一次新的采样，把采样结果放入队尾，而扔掉原来队首的一个数据，这样在队列中始终有N个“最新”的数据。计算滤波值时，只要把队列中的N个数据进行平均，就可得到新的滤波值。递推平均值法对周期性干扰有良好的抑制作用，平滑度高，灵敏度低，但对偶然出现的脉冲性干扰的抑制作用差。不易消除由于脉冲干扰引起的采样值的偏差，因此它不适用于脉冲干扰比较严重的场合，而适用于高频振荡系统。

4. 防脉冲干扰平均值滤波

在脉冲干扰比较严重的场合，若采用一般的平均值法，则干扰将“平均”到计算结果中去，故平均值法不易消除由于脉冲干扰而引起的采样值偏差。防脉冲干扰平均值法先对N个数据进行比较，去掉其中的最大值和最小值，然后计算余下的N-2个数据的算术平均值。

13.1.2 中值滤波

中值滤波是对某一被测参数连续采样N次(一般N取奇数)，然后把N次采样值从小到大或从大到小排队，再取其中间值作为本次采样值。中值滤波对于去掉偶然因素引起的波动或采样器不稳定而造成的误差所引起的脉冲干扰比较有效，对温度、液位等变化缓慢的被测参数采用此法能收到良好的滤波效果，但对流量、速度等快速变化的参数一般不易采用。

13.1.3　程序判断滤波

许多物理量的变化都需要一定的时间，相邻两次采样值之间的变化有一定的限度。程序判断滤波就是根据实践经验确定出相邻两次采样信号之间可能出现的最大偏差，若超出此偏差值，则表明该输入信号是干扰信号，应该去掉；若小于此偏差值，可将信号作为本次采样值。当采样信号由于随机干扰，如大功率用电设备的启动或停止，造成电流的尖峰干扰或误检测，以及变送器不稳定而引起的严重失真等，可采用程序判断法进行滤波。程序判断滤波根据滤波方法的不同，可分为限幅滤波和限速滤波。

1. 限幅滤波

限幅滤波把两次相邻的采样值相减，求出其增量(以绝对值表示)，然后与两次采样允许的最大差值(由被控对象的实际情况决定)进行比较，若小于或等于最大差值，则取本次采样值；若大于最大差值，则仍取上次采样值作为本次采样值。限幅滤波主要用于变化比较缓慢的参数，如温度、物理位置等测量系统。

2. 限速滤波

限速滤波最多可用 3 次采样值来决定采样结果，设顺序采样 3 个时刻的采样值。限速滤波既照顾了采样的实时性，又顾及了采样值变化的连续性。但这种方法也有明显的缺点：一是不够灵活，必须根据现场的情况不断更换新值；二是不能反映采样点数 N>3 时各采样值受干扰的情况，因而其应用受到一定的限制。

除了上述的数字滤波方法外，经常使用数值滤波的还有复合数字滤波、低通滤波等。每种滤波算法都有其各自的特点，在实际应用中，究竟选取哪一种数字滤波算法，应根据具体的测量参数合理的选用。不适当地应用数字滤波，不仅达不到滤波效果，反而会降低控制品质，甚至失控，这点必须予以注意。

13.2　非线性处理

在智能化仪表及测控系统中，特别是用显示仪表进行显示时，总是希望传感器及检测电路的输出和输入特性呈线性关系，使仪表在整个刻度范围内灵敏度一致，以便于读数及对系统进行分析处理。但是，很多检测元件具有不同程度的非线性特性，这使较大范围的动态检测存在着很大的误差。过去在使用模拟电路组成检测回路时，为了进行非线性补偿，通常用硬件电路组成各种补偿回路，但这样不仅增加了电路的复杂性，而且也很难达到理想的补偿。这种非线性补偿完全可以用单片机的软件来完成，其补偿较简单，精确度也很高，又减少了电路的复杂性。软件进行非线性补偿方法有查表法、线性插值法等。查表法是应用最广泛的一种，我们首先介绍这种非线性补偿方法。

13.2.1 查表法

所谓查表法，就是预先将满足一定要求的、表示变量与函数关系的一张表制出，然后把这张表存于单片机程序存储器中，自变量为地址编号，相应函数值为该地址单元内容。在单片机中设有专门的查表指令，可以方便地实现查表法。

[例 13.1] 一个倾角传感器，量程范围为 0~180° 。当角度为 0 时，输出的电压为 0V；当角度为 180° 时，传感器输出电压为 4.883 V。AT90S8535 单片机的 Vref 取 5.120 V。设与电压 E 相对应的值为 X，则：

```
X=1024*E/5.120=500E
```

传感器输出电压每增加 0.005 V，X 增加 1。列表可以取 E 为 0、0.005 V、0.010 V、0.015 V……5.110 V、5.115 V，它们对应的角度值分别为：0、0.18° 、0.36° ……179° 、180° ，可将 A/D 转换结果的数字量用 3 位 LED 显示。程序如下：

```
.include"8535def.inc"
.org $0000
    rjmp reset
.org $000e
    rjmp inter
.def    hledbyte=r19
.def lledbyte=r18
.equ    label=$0100
.equ    tab=$0200
.equ    distime=$38
.def    temp=r16
reset:  ldi temp,$02
    out sph,temp
    ldi temp,$5f
    out spl,temp
    ldi temp,$ff
    out ddrb,temp
    out ddrd,temp
    out portd,temp
    clr temp
    out ddra,temp
    out pina,temp
main:   clt
    sei
    rcall conini     ;调用A/D初始化
wait:   brtc wait    ;等待中断
    clt
    ldi r20,distime
start:  ldi zh,high(label*2)
    mov temp,r19
    rcall outpb
    cbi $12,2
    sbi $12,1
```

```
        sbi $12,0
        rcall delay
        mov temp,r18
        swap temp
        rcall outpb
        cbi $12,1
        sbi $12,2
        sbi $12,0
        rcall delay
        mov temp,r18
        rcall outpb
        cbi $12,0
        sbi $12,1
        sbi $12,2
        rcall delay
        dec r20
        brne start
        rjmp main
inter:  in lledbyte,adcl
        in hledbyte,adch
        cbi adsc,6
        set
        reti
conini: ldi temp,$8d
        out adcsr,temp
        clr temp
        out admux,temp
        sbi adcsr,6
        ret
delay:  ldi r27,$10
delay1: dec r26
        brne delay1
        dec r27
        brne delay1
        ret
outpb:  andi temp,$0f
        ldi zl,low(label*2)
        add zl,temp
        lpm
        out portb,r0
        ret
.cseg
.org $0100
.db $3f,$06,$5b,$4f,$66,$6d,$7d,$07,$7f,$6f
```

13.2.2　线性插值法

对非线性关系的模拟输入可用查表法处理，但也有它的缺点，占用空间大，输入比较麻烦，线性插值法可以克服这些缺点。线性插值法是将整个测量范围分为若干段，每段端

点的值查表求出，段内按直线计算。为了计算方便，一般将测量范围分为 2^n 段，生成 2^n+1 个节点。一般来说，段数分的越多查表误差越小。

当所查的数字量在节点上时，可以直接查表求出所需结果。当数字量在两个节点之间时，将非线性关系近似为线性关系处理，所以首先过两个节点做一条直线，然后算出模拟量对应的数字量。

本章主要介绍模拟信号的处理方法，包括信号干扰、噪音的消除和模拟信号向数字信号转换的方法。消除信号干扰和噪音的方法有平滑滤波、中值滤波和程序判断滤波等；模数信号转换的方法主要介绍了查表法和线性插值法。