第1章 单片嵌入式系统概述

在各种不同类型的嵌入式系统中，以单片微控制器(Microcontroller)作为系统的主要控制核心所构成的单片嵌入式系统（国内通常称为单片机系统）占据着非常重要的地位。本书将介绍以 AVR 系列单片微控制器为核心的单片嵌入式系统的原理、硬软件设计、调试等应用方法。

单片嵌入式系统的硬件基本构成可分成两大部分：单片微控制器芯片和外围的接口与控制电路。其中单片微控制器是构成单片嵌入式系统的核心。

单片微控制器又被称为单片微型计算机(Single-Chip Microcomputre 或 One-Chip Microcomputre)，或者嵌入式微控制器（Embedded Microcontroller)。而在国内普遍采用的名字为“单片机”。尽管单片机的“机”的含义并不十分恰当，比较模糊，但考虑到多年来国内习惯了单片机的叫法，为了符合我国的实际情况，本书仍采用单片机的名称。

所谓的单片微控制器－即单片机，它的外表通常只是一片大规模集成电路芯片。但在芯片的内部却集成了中央处理器单元(CPU)，各种存储器(RAM、ROM、EPROM、E^2PROM 和 FlashROM 等)，各种输入/输出接口(定时器/计数器、并行 I/O、串行 I/O 以及 A/D 转换接口等)，等众多的功能部件。因此，一片芯片就构成了一个基本的微型计算机系统。

由于单片机芯片的微小体积，极低的成本和面向控制的设计，使的它作为智能控制的核心器件被广泛地应用于嵌入到工业控制、智能仪器仪表、家用电器、电子通信产品等各个领域中的电子设备和电子产品中。可以说，由单片机为核心构成的单片嵌入式系统已成为现代电子系统中最重要的组成部分。

1.1 嵌入式系统简介

1.1.1 嵌入式计算机系统

计算机的出现首先是应用于数值计算。随着计算机技术的不断发展，计算机的处理速度越来越快，存储容量越来越大，外围设备的性能越来越好，满足了高速数值计算和海量数据处理的需要，形成了高性能的通用计算机系统。

1. 什么是嵌入式系统

以往我们按照计算机的体系结构、运算速度、结构规模、适用领域，将其分为大型计算机、中型机、小型机和微型计算机，并以此来组织学科和产业分工，这种分类沿袭了约 40 年。近 20 年来，随着计算机技术的迅速发展，以及计算机技术和产品对其它行业的广泛渗透，使得以应用为中心的分类方法变得更为切合实际。具体的说，就是按计算机的非嵌入式应用和嵌入式应用将其分为通用计算机系统和嵌入式计算机系统。

通用计算机具有计算机的标准形态，通过装配不同的应用软件，以类同面目出现，并应用在社会的各个方面。现在我们在办公室里、家庭中，最广泛普及使用的 PC 机就是通用计算机其最典型的代表。

而嵌入式计算机则是以嵌入式系统的形式隐藏在各种装置、产品和系统中的。在许多的应用领域中，如工业控制、智能仪器仪表、家用电器、电子通信设备等电子系统和电子产品中，对计算机的应用有着不同的要求。这些要求的主要特征为：

(1) 面对控制对象。面对物理量传感器变换的信号输入；面对人机交互的操作控制；面对对象的伺服驱动和控制。

(2) 嵌入到应用系统。体积小、低功耗、价格低廉，可方便地嵌入到应用系统和电子产品中。

(3) 能在工业现场环境中可靠运行。

(4) 优良的控制功能。对外部的各种模拟和数字信号能及时地捕捉，对多种不同的控制对象能灵活地进行实时控制。

可以看出，满足上述要求的计算机系统与通用计算机系统是不同的。换句话讲，能够满足和适合以上这些应用的计算机系统与通用计算机系统在应用目标上有巨大的差异。

我们将具备高速计算能力和海量存储，用于高速数值计算和海量数据处理的计算机称为通用计算机系统。而将面对工控领域对象，嵌入到各种控制应用系统、各类电子系统和电子产品中，实现嵌入式应用的计算机系统称之为嵌入式计算机系统，简称嵌入式系统(Embedded System)。

特定的环境、特定的功能，要求计算机系统与所嵌入的应用环境成为一个统一的整体，并且往往要满足紧凑、高可靠性、实时性好、低功耗等技术要求。对于这样一种面向具体专用应用目标的计算机系统的应用，以及系统的设计方法和开发技术，构成了今天嵌入式系统的重要内涵，也是嵌入式系统发展成为一个相对独立的计算机研究和学习领域的原因。

2. 嵌入式系统的特点与应用

因此，嵌入式系统就是指用于实现独立功能的专用计算机系统。它由包括微处理器、微控制器、定时器、传感器等一系列微电子芯片与器件，以及嵌入在存储器中的微型操作系统或控制系统软件组成，完成诸如实时控制、监测管理、移动计算、数据处理等各种自动化处理任务。

嵌入式系统是以应用为核心、以计算机技术为基础、软件硬件可裁剪、适应应用系统对功能、可靠性、安全性、成本、体积、重量、功耗、环境等方面有严格要求的专用计算机系统。嵌入式系统将应用程序和操作系统与计算机硬件集成在一起，简单讲就是系统的应用软件与系统的硬件一体化。这种系统具有软件代码小，高度自动化，响应速度快等特点，特别适应与面向对象的要求实时的和多任务的应用。

嵌入式计算机系统在应用数量上远远超过了各种通用计算机系统，一台通用计算机系统，如 PC 机的外部设备中就包含了 5-10 个嵌入式系统：键盘、鼠标、软驱、硬盘、显示卡、显示器、Modem、网卡、声卡、打印机、扫描仪、数字相机、USB 集线器等均是由嵌入式处理器控制的。在制造工业、过程控制、通讯、仪器、仪表、汽车、船舶、航空、航天、军事装备、消费类产品等方面均是嵌入式计算机的应用领域。

通用计算机系统和嵌入式计算机系统形成了计算机技术的两大分支。与通用计算机系统相比，嵌入式系统最显著的特性是面对工控领域的测控对象。工控领域的测量对象都是一些物理量，如压力、温度、速度、位移等；控制对象则包括马达、电磁开关等。嵌入式计算机系统对这些参量的采集、处理、控制速度是有限的，而对控制方式和能力的要求则是多种多样的。显然，这一特性形成并决定了嵌入式计算机系统和通用计算机系统在系统结构、技术、学习、开发和应用等诸方面的差别，也使得嵌入式系统成为计算机技术发展中的一个重要分支。

嵌入式计算机系统以其独特的结构和性能，越来越多地应用的国民经济的各个领域。

1.1.2 单片嵌入式系统

嵌入式计算机系统的构成，根据其核心控制部分的不同可分为几种不同的类型：

a. 各种类型的工控机

b. 可编程逻辑控制器 PLC
c. 以通用微处理器或数字信号处理器构成的嵌入式系统
d. 单片嵌入式系统

采用上述不同类型的核心控制部件所构成的系统都实现了嵌入式系统的应用，成为嵌入式系统应用的庞大家族。

以单片机作为控制核心的单片嵌入式系统大部分应用于专业性极强的工业控制系统中。其主要特点是：结构和功能相对单一、存储容量较小、计算能力和效率比较低，简单的用户接口。由于这种嵌入式系统功能专一可靠、价格便宜，因此在工业控制、电子智能仪器设备等领域有着广泛的应用。

作为单片嵌入式系统的核心控制部件单片机，它从体系结构到指令系统都是按照嵌入式系统的应用特点专门设计的，它能最好地满足面对控制对象、应用系统的嵌入、现场的可靠运行和优良的控制功能要求。因此，单片嵌入式应用是发展最快、品种最多、数量最大的嵌入式系统，也有着广泛的应用前景。由于单片机具有嵌入式系统应用的专用体系结构和指令系统，因此在其基本体系结构上，可衍生出能满足各种不同应用系统要求的系统和产品。用户可根据应用系统的各种不同要求和功能，选择最佳型号的单片机。

作为一个典型的嵌入式系统－－单片嵌入式系统，在我国大规模应用已有几十年的历史。它不但是在中、小型工控领域、智能仪器仪表、家用电器、电子通信设备和电子系统中最重要的工具和最普遍的应用手段，同时正是由于单片嵌入式系统的广泛应用和不断发展，也大大推动了嵌入式系统技术的快速发展。因此对于电子、通信、工业控制、智能仪器仪表等相关专业的学生来讲，深入学习和掌握单片嵌入式系统的原理与应用，不仅能对自己所学的基础知识进行检验，而且能够培养和锻炼自己的问题分析、综合应用、和动手实践的能力，掌握真正的专业技能和应用技术。同时，深入学习和掌握单片嵌入式系统的原理与应用，也为更好的掌握其它嵌入式系统的打下重要的基础，这个特点尤其表现在硬件设计方面。

1.1.3 单片机的发展历史

1970 年微型计算机研制成功后，随后就出现了单片机。美国 Inter 公司在 1971 年推出了 4 位单片机 4004；1972 年推出了雏形 8 位单片机 8008。特别是在 1976 年推出 MCS-48 单片机以后的三十年中，单片机的发展和其相关的技术经历了数次的更新换代。其发展速度大约每三四年要更新一代、集成度增加一倍、功能翻一番。

尽管单片机出现的历史并不长，但以 8 位单片机的推出为起点，那么，单片机的发展大致可分为四个阶段。

第一阶段（1976 年-1978 年）：初级单片机阶段。以 Inter 公司 MCS-48 为代表。这个系列的单片机内集成有 8 位 CPU、I/O 接口、8 位定时器/计数器，寻址范围不大于 4K 字节，简单的中断功能，无串行接口。

第二阶段（1978 年-1982 年）：单片机完善阶段。在这一阶段推出的单片机其功能有较大的加强，能够应用于更多的场合。这个阶段的单片机普遍带有串行 I/O 口、有多级中断处理系统、16 位定时器/计数器，片内集成的 RAM、ROM 容量加大，寻址范围可达 64K 字节。一些单片机片内还集成了 A/D 转换接口。这类单片机的典型代表有 Inter 公司的 MCS-51、Motorola 公司的 6801 和 Zilog 公司的 Z8 等。

第三阶段（1982 年-1992 年）：8 位单片机巩固发展及 16 位高级单片机发展阶段。在此阶段，尽管 8 位单片机的应用已广泛普及，但为了更好满足测控系统的嵌入式应用的要求，单片机集成的外围接口电路有了更大的扩充。这个阶段单片机的代表为 8051 系列。许多半导体公司和生产厂以 MCS-51 的 8051 为内核，推出了满足各种嵌入式应用的多种类型和型号

的单片机。其主要技术发展有：

（1） 外围功能集成。满足模拟量直接输入的 ADC 接口；满足伺服驱动输出的 PWM；保证程序可靠运行的程序监控定时器 WDT（俗称看门狗电路）。

（2） 出现了为满足串行外围扩展要求的串行扩展总线和接口，如 SPI、I^2C Bus、单总线（1-Wire）等。

（3） 出现了为满足分布式系统，突出控制功能的现场总线接口，如 CAN Bus 等。

（4） 在程序存储器方面广泛使用了片内程序存储器技术，出现了片内集成 EPROM、EEPROM、FlashROM 以及 MaskROM、OTPROM 等各种类型的单片机，以满足不同产品的开发和生产的需要，也为最终取消外部程序存储器扩展奠定了良好的基础。

与此同时，一些公司面向更高层次的应用，发展推出了 16 位的单片机，典型代表有 Inter 公司的 MCS-96 系列的单片机。

第四阶段（1993 年-现在）：百花齐放阶段。现阶段单片机发展的显著特点是百花齐放、技术创新，以满足日益增长的广泛需求。其主要方面有：

（1）单片嵌入式系统的应用是面对最底层的电子技术应用，从简单的玩具、小家电；到复杂的工业控制系统、智能仪表、电器控制；以及发展到机器人、个人通信信息终端、机顶盒等。因此，面对不同的应用对象，不断推出适合不同领域要求的，从简易性能到多全功能的单片机系列。

（2）大力发展专用型单片机。早期的单片机是以通用型为主的。由于单片机设计生产技术的提高、周期缩短、成本下降，以及许多特定类型电子产品，如家电类产品的巨大的市场需求能力，推动了专用单片机的发展。在这类产品中采用专用单片机，具有低成本、资源有效利用、系统外围电路少、可靠性高的优点。因此专用单片机也是单片机发展的一个主要方向。

（3）致力于提高单片机的综合品质。采用更先进的技术来提高单片机的综合品质，如提高 I/O 口的驱动能力；增加抗静电和抗干扰措施；宽（低）电压低功耗等。

1.1.4 单片机的发展趋势

综观三十年的发展过程，作为单片嵌入式系统的核心——单片机，正朝着多功能、多选择、高速度、低功耗、低价格、扩大存储容量和加强 I/O 功能等方向发展。其进一步的发展趋势是多方面的。

（1） 全盘 CMOS 化。CMOS 电路具有许多优点，如极宽的工作电压范围；极佳的低功耗及功耗管理特性等。CMOS 化已成为目前单片机及其外围器件流行的半导体工艺。

（2） 采用 RISC 体系结构。早期的单片机大多采用 CISC 结构体系，指令复杂，指令代码、周期数不统一；指令运行很难实现流水线操作，大大阻碍了运行速度的提高。如 MCS-51 系列单片机，当外部时钟为 12MHz 时，其单周期指令运行速度也仅为 1MIPS。采用 RISC 体系结构和精简指令后，单片机的指令绝大部分成为单周期指令，而通过增加程序存储器的宽度（如从 8 位增加到 16 位），实现了一个地址单元存放一条指令。在这种体系结构中，很容易实现并行流水线操作，大大提高了指令运行速度。目前一些 RISC 结构的单片机，如美国 ATMEL 公司的 AVR 系列单片机已实现了一个时钟周期执行一条指令。与 MCS-51 相比，在相同的 12MHz 外部时钟下，单周期指令运行速度可达 12MIPS。一方面可获得很高的指令运行速度，另一方面，在相同的运行速度下，可大大降低时钟频率，有利于获得良好的电磁兼容效果。

（3） 多功能集成化。单片机在内部已集成了越来越多的部件，这些部件不仅包括一般常用的电路，如：定时/计数器，模拟比较器，A/D 转换器，D/A 转换器，串行通信接口，WDT 电路，LCD 控制器等，还有的单片机为了构成控制网络或形成局部网，内部含有局部网络控

制模块 CAN 总线，以方便地构成一个控制网络。为了能在变频控制中方便使用单片机，形成最具经济效益的嵌入式控制系统。有的单片机内部设置了专门用于变频控制的脉宽调制控制电路 PWM。

（4） 片内存储器的改进与发展。目前新型的单片机一般在片内集成两种类型的存储器：随机读写存储器 SRAM，做为临时数据存储器存放工作数据用；只读存储器 ROM，做为程序存储器存放系统控制程序和固定不变的数据。片内存储器的改进与发展的方向是扩大容量、ROM 数据的易写和保密等。

- 片内存储容量的增加。新型的单片机一般在片内集成的 SRAM 在 128 字节至 1K 字节，ROM 的容量一般为 4K 字节至 8K 字节。为了适应网络、音视频等高端产品的需要，高挡的单片机在片内集成了更大容量的 RAM 和 ROM 存储器。如 ATMEL 公司的 ATmega16，片内的 SRAM 为 1K 字节，FlashROM 为 16K 字节。而该系列的高端产品 ATmega256，片内集成了 8K 字节的 SRAM，256K 字节的 FlashROM 和 4K 字节的 EEPROM。
- 片内程序存储器由 EPROM 型向 FlashROM 发展。早期的单片机在片内往往没有程序存储器或片内集成 EPROM 型的程序存储器。将程序存储器集成在单片机内可以大大提高单片机的抗干扰性能、提高程序的保密性、减少硬件的设计的复杂性和空间等许多优点，因此片内集成程序存储器已成为新型单片机的标准方式。但由于 EPROM 需要使用 12v 高电压编程写入、紫外线光照擦除、重写入次数有限等缺点，这给使用带来了不便。新型的单片机则采用 FlashROM 以及 MaskROM、OTPROM 做为片内的程序存储器。FlashROM 在通常电压（如 5v/3v）下就可以实现编程写入和擦除操作，重写次数在 10000 次以上，并可实现在线编程写入 ISP 技术的优点，为使用带来了及大的方便。采用 MaskROM 的微控制器称为掩模芯片，它是在芯片制造过程中就将程序“写入”了，并永远不能改写。采用 OTPROM 的微控制器，其芯片出厂时片内的程序存储器是“空的”，它允许用户将自己编写好的程序一次性的编程写入，之后便再也无法修改了。后两种类型的单片机适合于大批量产品生产的使用，而前两种类型的微控制器则适合产品的设计开发、批量生产以及学习培训的应用。
- 程序保密化。一个单片嵌入式系统的系统程序是系统的最重要的部分，是知识产权保护的核心。为了片内的程序防止被非法读出复制，新型的单片机往往采用对片内的程序存储器采用加锁保密。系统程序编程写入片内的程序存储器后，可以再对加密保护单元编程，使芯片加锁。加锁加密后，从芯片的外部则无法读取片内的系统程序代码，若将加密单元擦除，则片内的程序也同时擦除掉，这样便达到了程序保密的目的。

（5） ISP、IAP 及基于 ISP、IAP 技术的开发和应用。ISP(In System Programmable)称为在线系统可编程技术。随着微控制器在片内集成 EEPROM、FlashROM 的发展，导致了 ISP 技术在单片机中的应用。首先实现了系统程序的串行编程写入（下载），使得不必将焊接在 PCB 印刷电路板上的芯片取下，就可直接将程序下载到单片机的程序存储器中，淘汰了专用的程序下载写入设备。其次，基于 ISP 技术的实现，使模拟仿真开发技术重新兴起。在单时钟、单指令运行的 RISC 结构的单片机中，可实现 PC 机通过串行电缆对目标系统的在线仿真调试。在 ISP 技术应用的基础上，又发展了 IAP(In Application Programmable)技术，也称在应用可编程技术。利用 IAP 技术，实现了用户可随时根据需要对原有的系统方便的在线更新软件、修改软件，还能实现对系统软件的远程诊断、远程调试和远程更新。

（6） 实现全面功耗管理。采用 CMOS 工艺后，单片机具有极佳的低功耗和功耗管理功能。它包括：

- 传统的 CMOS 单片机的低功耗运行方式，既闲置方式（Idle Mode）、掉电方式（Power Down Mode）。
- 双时钟技术。配置有高速（主）和低速（子）两个时钟系统。在不需要高速运行时，则转入子时钟控制下，以节省功耗。

- 片内外围电路的电源管理。对集成在片内的外围接口电路实行供电管理，当该外围电路不运行时，关闭其供电。
- 低电压节能技术。CMOS 电路的功耗与电源电压有关，降低系统的供电电压，能大幅度减少器件的功耗。新型的单片机往往具有宽电压（3V－5V）或低电压（3V）运行的特点。低电压低功耗是手持便携式系统重要的追求目标，也是绿色电子的发展方向。

（7） 以串行总线方式为主的外围扩展。目前，单片机与外围器件接口技术发展的一个重要方面是由并行外围总线接口向串行外围总线接口的发展。采用串行总线方式为主的外围扩展技术具有方便、灵活、电路系统简单、占用 I/O 资源少等特点。采用串行接口虽然比采用并行接口数据传输速度慢，但随着半导体集成电路技术的发展，大批采用标准串行总线通信协议（如：SPI、I^2C、1-Wire 等）的外围芯片器件的出现，串行传输速度也在不断提高（可达到 1M－10M 的速率），片内集成程序存储器而不必外部并行扩展程序存储器，加之单片嵌入式系统有限速度的要求，使得以串行总线方式为主的外围扩展方式能够满足大多数系统的需求，成为流行的扩展方式，而采用并行接口的扩展技术则成为辅助方式。

（8） 单片机向片上系统 SOC 的发展。SOC(System On Chip)是一种高度集成化、固件化的芯片级集成技术，其核心思想是把除了无法集成的某些外部电路和机械部分之外的所有电子系统电路全部集成在一片芯片中。现在一些新型的单片机（如 AVR 系列单片机）已经是 SOC 的雏形，在一片芯片中集成了各种类型和更大容量的存储器，更多性能更加完善和强大的功能电路接口，这使得原来需要几片甚至十几片芯片组成的系统，现在只用一片就可以实现。其优点不仅是减小了系统的体积和成本，而且也大大提高了系统硬件的可靠性和稳定性。

1.2 单片嵌入式系统的结构与应用领域

1.2.1 单片嵌入式系统结构

仅由一片单片机芯片是不能构成一个应用系统的。系统的核心控制芯片，往往还需要与一些外围芯片、器件和控制电路机构有机的连接在一起，才构成了一个实际的单片机系统，进而再嵌入到应用对象的环境体系中，作为其中的核心智能化控制单元而构成典型的单片嵌入式应用系统，如洗衣机、电视机、空调、智能仪器、智能仪表等等。

单片嵌入式系统的结构如图 1-1 所示，通常包括三大部分：既能实现嵌入式对象各种应用要求的单片机、全部系统的硬件电路和应用软件。

1．单片机：单片机是单片嵌入式系统的核心控制芯片，由它实现对控制对象的测控、系统运行管理控制和数据运算处理等功能。

2．系统硬件电路：根据系统采用单片机的特性以及嵌入对象要实现的功能要求而配备的外围芯片、器件所构成的全部硬件电路。通常包括以下几部分：

- 基本系统电路。提供和满足单片机系统运行所需要的时钟电路、复位电路、系统供电电路、驱动电路、扩展的存储器等。
- 前向通道接口电路。这是应用系统面向对象的输入接口，通常是各种物理量的测量传感器、变换器输入通道。根据现实世界物理量转换成电量输出信号的类型，如模拟电压电流、开关信号、数字脉冲信号等的不同，接口电路也不同。常见的有传感器、信号调理器、模/数转换器 ADC、开关输入、频率测量接口等。
- 后向通道接口电路。这是应用系统面向对象的输出控制电路接口。根据应用对象伺服和控制要求，通常有数/模转换器 DAC、开关量输出、功率驱动接口、PWM 输出控制等。
- 人机交互通道接口电路。人机交互通道接口是满足应用系统人机交互需要的电路，有键盘、拨动开关、LED 发光二极管、数码管、LCD 液晶显示器、打印机等多种输入输出接口

电路。

- 数据通信接口电路。数据通信接口电路是满足远程数据通信或构成多机网络应用系统的接口。通常有 RS232、PSI、I2C、CAN 总线、USB 总线等通信接口电路。

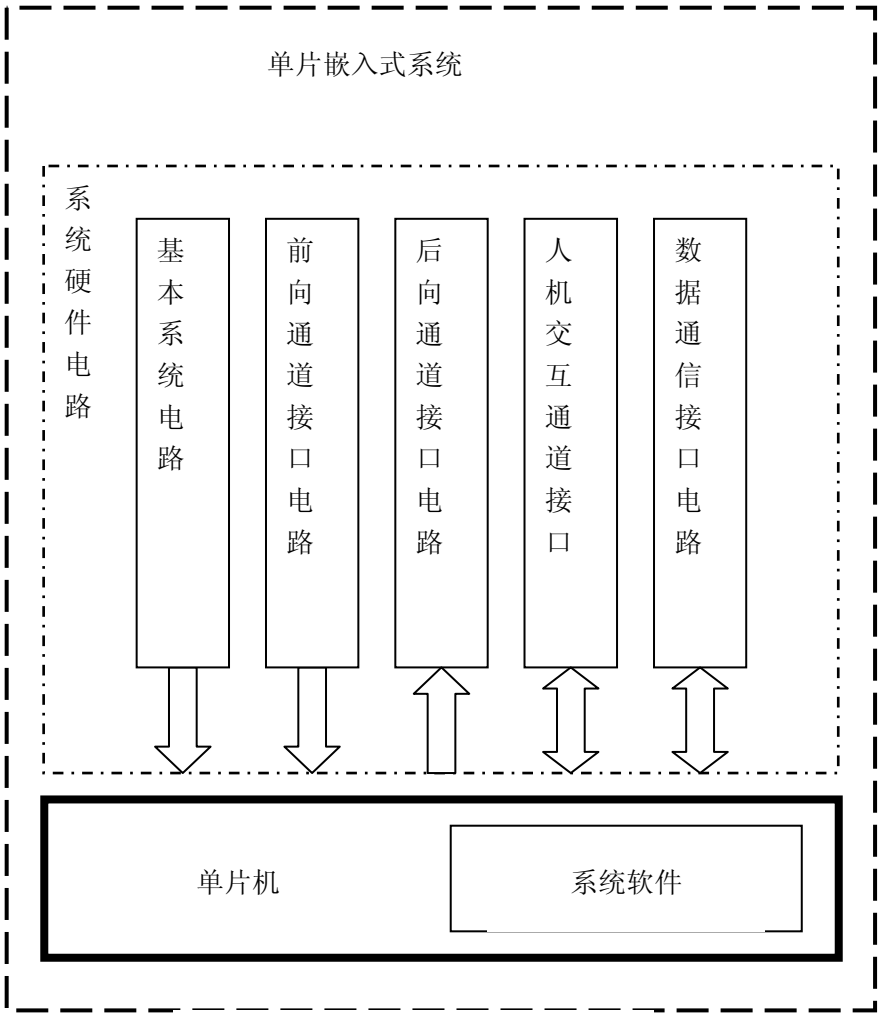

图 1-1 单片嵌入式系统结构

3．系统的应用软件：系统应用软件的核心就是下载到单片机中的系统运行程序。整个嵌入式系统全部硬件的相互协调工作、智能管理和控制都由系统运行程序决定。它可认为是单片嵌入式系统核心的核心。一个系统应用软件设计的好坏，往往也决定了整个系统性能的好坏。

系统软件是根据系统功能要求设计的，一个嵌入式系统的运行程序实际上就是该系统的监控与管理程序。对于小型系统的应用程序，一般采用汇编语言编写。而对于中型和大型系统的应用程序，往往采用高级程序设计语言如 C 语言、Basic 语言来编写。

编写嵌入式系统应用程序与编写其它类型的软件程序(如基于 PC 的应用软件设计开发)有很大的不同，嵌入式系统应用程序更加面向硬件低层和控制，而且还要面对有限的资源(如有限的 RAM)。因为嵌入式系统的应用软件不仅要直接面对单片机和与它连接的各种不同种类和设计的外围硬件电路编程，还要面对系统的具体应用和功能编程。整个运行程序常常是输入、输出接口设计，存储器，外围芯片，中断处理等多项功能交织在一起。因此，除了硬件系统的设计，系统应用软件的设计也是嵌入式系统开发研制过程中重要和困难的任务。

需要强调说明的是，针对单片嵌入式系统的硬件设计和软件设计两者之间的关系是十分紧密，互相依赖和制约的。因此，通常要求嵌入式系统的开发人员即要具备扎实的硬件设计

能力，同时也要具备相当优秀的软件程序设计能力。

1.2.2 单片嵌入式系统的应用领域

以单片机为核心构成的单片嵌入式系统已成为现代电子系统中最重要的组成部分。在现代的数字化世界中，单片嵌入式系统已经大量地渗透到我们生活的各个领域，几乎很难找到哪个领域没有单片机的踪迹。导弹的导航装置，飞机上各种仪表的控制，计算机的网络通信与数据传输，工业自动化过程的时实控制和数据处理，生产流水线上的机器人，医院里先进的医疗器械和仪器，广泛使用的各种智能 IC 卡，小朋友的程控玩具和电子宠物都是典型的单片嵌入式系统应用。

由于单片机芯片的微小体积，极低的成本和面向控制的设计，使的它作为智能控制的核心器件被广泛地用于嵌入到工业控制、智能仪器仪表、家用电器、电子通信产品等各个领域中的电子设备和电子产品中，主要的应用领域有以下几个方面。

（1） 智能家用电器。俗程带“电脑”的家用电器，如电冰箱、空调、微波炉、电饭锅、电视机、洗衣机等。传统的家用电器中嵌入了单片机系统后使产品性能特点都得到很大的改善，实现了运行智能化、温度的自动控制和调节、节约电能等。

（2） 智能机电一体化产品。单片机嵌入式系统与传统的机械产品相结合，使传统的机械产品结构简化，控制智能化，构成新一代的机电一体化产品。这些产品已在纺织、机械、化工、食品等工业生产中发挥出巨大的作用。

（3） 智能仪表仪器。用单片机嵌入式系统改造原有的测量、控制仪表和仪器，能促使仪表仪器向数字化、智能化、多功能化、综合化、柔性化发展。由单片机系统构成的智能仪器仪表可以集测量、处理、控制功能与一体，赋予传统的仪器仪表以崭新的面貌。

（4） 测控系统。用单片机嵌入式系统可以构成各种工业控制系统、适应控制系统、数据采集系统等。例如，温室人工气候控制、汽车数据采集与自动控制系统。

1.3 AVR单片机简介

1.3.1 ATMEL公司的单片机产品

ATMEL 公司是世界上著名的生产高性能、低功耗、非易失性存储器和各种数字模拟 IC 芯片的半导体制造公司。在单片微控制器方面，ATMEL 公司有基于 8051 内核、基于 AVR 内核和基于 ARM 内核的三大系列单片机产品（确切的讲，最后一款应称为嵌入式微处理器）。ATMEL 公司在它的单片机产品中，融入了先进的 EEPROM 电可擦除和 Flash ROM 闪速存储器技术，使得该公司的单片机具备了优秀的品质，在结构、性能和功能等方面都有明显的优势。

ATMEL 公司把 8051 内核与其擅长的 Flash 存储器技术相结合，是国际上最早推出片内集成可重复擦写 1000 次以上 Flash 程序存储器、采用低功耗 CMOS 工艺的 8051 兼容单片机的生产商之一。市场上家喻户晓的 AT89C51、AT89C52、AT89C1051、AT89C2051 就是 ATMEL 公司生产的基于 8051 内核系列单片机中的典型产品（现在已升级换代为 AT89Sxx 系列，采用 ISP 在线编程技术）。该系列单片机一直在我国的单片机市场上占有相当大的份额。

8051 结构的单片机采用复杂指令系统 CISC(Complex Instruction Set Computer)体系。由于 CISC 结构存在指令系统不等长，指令数多，CPU 利用效率低，执行速度慢等缺陷，已不能满足和适应设计中高档电子产品和嵌入式系统应用的需要。ATMEL 公司发挥其 Flash 存储器技术的特长，于 1997 年研发和推出了全新配置采用精简指令集 RISC(Reduced Instruction Set CPU)结构的新型单片机，简称 AVR 单片机。

精简指令集 RISC 结构是 20 世纪 90 年代开发出来的一种综合了半导体集成技术和提高软件性能的新结构，是为了提高 CPU 运行的速度而设计的芯片体系。它的关键技术在于采用流水线操作(Pipelining)，和等长指令体系结构，使一条指令可以在一个单独操作中完成，从而实现在一个时钟周期里完成一条或多条指令。同时 RISC 体系还采用了通用快速寄存器组的结构，大量使用寄存器之间的操作，简化了 CPU 中处理器、控制器和其它功能单元的设计。因此，RISC 的特点就是通过简化 CPU 的指令功能，使指令的平均执行时间减少，从而提高 CPU 的性能和速度。在使用相同的晶片技术和相同的运行时钟下，RISC 系统的运行速度是 CISC 的 2～4 倍。正由于 RISC 体系所具有的优势，使得它在高端系统得到了广泛的应用。例如，ARM 以及大多数 32 位的处理器都采用 RISC 体系结构。

ATMEL 公司的 AVR 是 8 位单片机中第一个真正的 RISC 结构的单片机。它采用了大型快速存取寄存器组、快速的单周期指令系统以及单级流水线等先进技术，使得 AVR 单片机具有高达 1MIPS / MHz 的高速运行处理能力。

AVR 采用流水线技术，在前一条指令执行的时候，就取出现行的指令，然后以一个周期执行指令。大大提高了 CPU 的运行速度。而在其它的 CISC 以及类似的 RISC 结构的单片机中，外部振荡器的时钟被分频降低到传统的内部指令执行周期，这种分频最大达 12 倍（8051）。

另外一点，传统的基于累加器的结构单片机（如 8051），需要大量的程序代码来完成和实现在累加器和存储器之间的数据传送。而在 AVR 单片机中，由于采用 32 个通用工作寄存器构成快速存取寄存器组，用 32 个通用工作寄存器代替了累加器，从而避免了在传统结构中累加器和存储器之间数据传送造成的瓶颈现象，进一步提高了指令的运行效率和速度。

随着电子产品更新换代的周期缩短以及不断向高端发展，为了加快产品进入市场的时间和简化系统的设计、开发、维护和支持，对于以单片机为核心所组成的高端嵌入式系统来说，用高级语言编程已成为一种标准设计方法。AVR 单片机采用 RISC 结构，其目的就是在于能够更好地采用高级语言（例如 C 语言、BASIC 语言）来编写嵌入式系统的系统程序，从而能高效地开发出目标代码。

AVR 单片机采用低功率、非挥发的 CMOS 工艺制造，内部分别集成 Flash、EEPROM 和 SRAM 三种不同性能和用途的存储器。除了可以通过使用一般的编程器（并行高压方式）对 AVR 单片机的 Flash 程序存储器和 EEPROM 数据存储器进行编程外，大多数的 AVR 单片机还具有 ISP 在线编程的特点以及 IAP 在应用编程的特点。这些优点为使用 AVR 单片机开发设计和生产产品提供了及大的方便。在产品的设计生产中，可以“先装配后编程”，从而缩短了研发周期、工艺流程，并且还可以节约购买开发仿真编程器的费用。同样，对于学习和使用 AVR 单片机的用户来说，也不必购买昂贵的开发仿真硬件设备，只需要具备一套好的 AVR 开发软件平台，就可以从事 AVR 单片机系统的学习、设计和开发工作了。

1.3.2 AVR单片机的主要特点

AVR 单片机吸取了 PIC 及 8051 等单片机的优点，同时在内部结构上还作了一些重大改进，其主要的优点如下：

● 程序存储器为价格低廉、可擦写 1 万次以上、指令长度单元为 16 位(字)的 FlashROM(即程序存储器宽度为 16 位，按 8 位字节计算时应乘 2)。而数据存贮器为 8 位。因此 AVR 还是属于 8 位单片机。

● 采用 CMOS 技术和 RISC 架构，实现高速(50ns)、低功耗(μA)、具有 SLEEP(休眠)功能。AVR 的一条指令执行速度可达 50ns(20MHz)，而耗电则在 1uA～2.5mA 间。AVR 采用 Harvard 结构，以及一级流水线的预取指令功能，即对程序的读取和数据的操作使用不同的数据总线，因此，当执行某一指令时，下一指令被预先从程序存储器中取出，这使得指令可以在每一个

时钟周期内被执行。

● 高度保密。可多次烧写的 Flash 且具有多重密码保护锁定(LOCK)功能，因此可低价快速完成产品商品化，且可多次更改程序(产品升级)，方便了系统调试，而且不必浪费 IC 或电路板，大大提高了产品质量及竞争力。

● 工业级产品。具有大电流 10～20mA(输出电流)或 40mA(吸电流)的特点，可直接驱动 LED、SSR 或继电器。有看门狗定时器(WDT)安全保护，可防止程序走飞，提高产品的抗干扰能力。

● 超功能精简指令。具有 32 个通用工作寄存器(相当于 8051 中的 32 个累加器)，克服了单一累加器数据处理造成的瓶颈现象。片内含有 128-4K 字节 SRAM，可灵活使用指令运算，适合使用功能很强的 C 语言编程，易学、易写、易移植。

● 程序写入器件时，可以使用并行方式写入(用编程器写入)，也可使用串行在线下载(ISP)、在应用下载（IAP）方法下载写入。也就是说不必将单片机芯片从系统板上拆下拿到万用编程器上烧录，而可直接在电路板上进行程序的修改、烧录等操作，方便产品升级，尤其是对于使用 SMD 表贴封装器件，更利于产品微型化。

● 通用数字 I/O 口的输入输出特性与 PIC 的 HI/LOW 输出及三态高阻抗 HI-Z 输入类同，同时可设定类同与 8051 结构内部有上拉电阻的输入端功能，便于作为各种应用特性所需(多功能 I/O 口)，AVR 的 I/O 口是真正的 I/O 口，能正确反映 I/O 口的输入/输出的真实情况。

● 单片机内集成有模拟比较器，可组成廉价的 A/D 转换器。

● 像 8051 一样，有多个固定中断向量入口地址，可快速响应中断，而不是像 PIC 一样所有中断都在同一向量地址，需要以程序判别后才可响应，这会浪费且失去控制时机的最佳机会。

● 同 PIC 一样，带有可设置的启动复位延时计数器。AVR 单片机内部有电源上电启动计数器，当系统 RESET 复位上电后，利用内部的 RC 看门狗定时器，可延迟 MCU 正式开始读取指令执行程序的时间。这种延时启动的特性，可使 MCU 在系统电源、外部电路达到稳定后再正式开始执行程序，提高了系统工作的可靠性，同时也可节省外加的复位延时电路。

● 具有多种不同方式的休眠省电功能和低功耗的工作方式。

● 许多 AVR 单片机具有内部的 RC 振荡器，提供 1/2/4/8MHz 的工作时钟，使该类单片机无需外加时钟电路元器件即可工作，非常简单和方便。

● 有多个带预分频器的 8 位和 16 位功能强大的计数器/定时器（C/T），除了实现普通的定时和计数功能外，还具有输入捕获、产生 PWM 输出等更多的功能。

● 性能优良的串行同/异步通讯 USART 口，不占用定时器。可实现高速同/异步通信。

● Mega8515 及 Mega128 等芯片具有可并行扩展的外部接口，扩展能力达 64KB。

● 工作电压范围宽 2.7V～6.0V，具有系统电源低电压检测功能，电源抗干扰性能强。

● 有多通道的 10 位 A/D 及实时时钟 RTC。许多 AVR 芯片内部集成了 8 路 10 位 A/D 接口，如：mega8、mega16、mega8535 等。

● AVR 单片机还在片内集成了可擦写 10 万次的 EEPROM 数据存储器，等于又增加了一个芯片，可用于保存系统的设定参数、固定表格和掉电后的数据的保存。即方便了使用，减小了系统的空间，又大大提高了系统的保密性。

1.3.3 AVR系列单片机简介

ATMEL 公司的 AVR 单片机有三个系列的产品。为满足不同的需求和应用，ATMEL 公司对 AVR 单片机的内部资源进行了相应的扩展和删减，推出了 tinyAVR、low power AVR 和 megaAVR，分别对应低、中、高三个不同档次数十种型号的产品(表 1.1)。

表 1.1 AVR 单片机分类表

8 位 AVR 单片机 RISC 结构		存储器配备（Bytes）		
系列	封装	Flash	SRAM	E^2PROM
tinyAVR	8-32 pin	1-2K	up to 128	up to 128
low power AVR	8-44 pin	1-8K	up to 1K	up to 512
megaAVR	28-64 pin	8-128K	up to 4K	up to 4K

三个系列的所有型号的 AVR 单片机，其内核都是相同的，指令系统兼容。只是在内部资源的配备（存储器容量的大小等）、以及片内集成的外围接口的数量和功能上有所不同。这些不同型号 AVR 单片机的封装形式也不一样，引脚数从 8 到 100 脚，价格从几元到几十元，可以满足不同场合、不同应用的需求，用户可以根据需要选择。表 1.2 至 1.4 为 AVR 三个系列单片机的选型表。

自 2002 年以来，ATMEL 公司对 AVR 单片机产品线进行了调整，逐步停止了性能重叠的中档 low power AVR 单片机中 AT90s 系列的生产，而用性能更加优越的 mega 系列代替。如停止 AT90S4414、AT90S8515 等芯片的生产，用 ATmega8515 替代 AT90S8515，用 ATmega8535 替代 AT90S8535，用 ATmega8 代替 AT90S4433 等。由于 mega 系列单片机的性能更加完美，使用更加方便，功能更加强大，因此，ATMEL 公司今后将以 mega 系列作为 AVR 单片机的主流产品，逐步减少和停止中档 AVR 单片机（AT90SXXXX）的生产。从表 1.2 至 1.4 中可以看出，目前 tinyAVR 和 mega 系列的单片机已成为了 AVR 的主流。

tinyAVR 系列的 AVR 内部的资源相对少一些，引脚也少。适合应用在家用电器、简单的控制方面的应用，如：空调、冰箱、微波炉、烟雾报警器等(见表 1.2)。

mega 系列单片机的性能不仅优越，同时也有非常好的性能价格比。引脚数最少(28 个引脚)的 ATmega8，目前在我国国内市场上的价格不超过 10 元人民币，却有 1K 的 SRAM、8K 的 Flash、512 个字节的 E^2PROM，2 个 8 位和 1 个 16 位共 3 个超强功能的定时器/计数器，以及 USART、SPI、8 路 10 位 ADC、WDT、RTC、ISP、IAP、TWI（I^2C）、片内高精度 RC 振荡器等多种功能的接口和特性(表 1.3)。

ATmega2560 是目前 AVR 中配置最全、功能最强的一款。它的引脚数最多(100 个引脚)，在片内集成了 8K 字节的 SRAM、256K 字节的 Flash、4K 字节的 EEPROM，支持 64K 空间的外部并行扩展，2 个 8 位和 4 个 16 位共 6 个超强功能的定时器/计数器，以及 4 路 USART、SPI、多路 10 位 ADC、WDT、RTC、ISP、IAP、TWI（I^2C）、片内高精度 RC 振荡器等多种功能的接口和特性，适合高档电子产品的应用（表 1.4）。

表 1.2 tinyAVR系列单片机(部分)

	ATtiny11	ATtiny12	ATtiny13	ATtiny15L	ATtiny24	ATtiny26	ATtiny28L	ATtiny85	ATtiny2313
Flash(KB)	1	1	1	1	2	2	2	8	2
E^2PROM(B)	-	64	64	64	128	128	-	512	128
快速寄存器	32	32	32	32	32	32	32	32	32
SRAM(B)	0	0	0	0	128	128	0	512	128
I/O Pins	6	6	6	6	12	16	11	6	18
中断数目	4	5	9	8	17	11	5	15	8
外部中断口	1	1	6	1(+5)	12	1	2(+8)	7	2
SPI	-	-	-	-	USI	USI	-	USI	USI
USART	-	-	-	-	-	-	-	-	1
TWI	-	-	-	-	USI	-	-	USI	-
硬件乘法器	-	-	-	-	-	-	-	-	-
8 位定时器	1	1	1	2	1	2	1	2	1
16 位定时器	-	-	-	-	1	-	-	-	1
PWM通道	-	-	2	1	4	2	-	4	4
模拟比较器	Y	Y	Y	Y	Y	Y	Y	Y	Y
10 位A/D通道	-	-	4	4	8	11	-	4	-
掉电检测BOD	-	Y	Y	Y	Y	Y	-	Y	Y
Watchdog	Y	Y	Y	Y	Y	Y	Y	Y	Y
片内系统时钟	Y	Y	Y	Y	Y	Y	Y	Y	Y
在线编程ISP	-	Y	Y	Y	Y	Y	-	Y	Y
自编程SPM	-	-	Y	-	-	-	-	-	Y
debugWIRE	-	-	Y	-	-	-	-	-	Y
Vcc(V)(最低) (最高)	2.7 5.5	1.8 5.5	1.8 5.5	2.7 5.5	1.8 5.5	2.7 5.5	2.7 5.5	1.8 5.5	1.8 5.5
系统时钟(M)	0-6	0-8	0-20	1.6	0-20	0-8	0-4	0-1	0-20
封装形式	PDIP8 SOIC8	PDIP8 SOIC8	PDIP8 SOIC8	PDIP8 SOIC8	PDIP14 SOIC14 MLF20	PDIP20 SOIC20 MLF32	PDIP28 TQFP32 MLF32	PDIP8 SOIC8 MLF20	PDIP20 SOIC20 MLF32

表 1.3 megaAVR系列单片机(低配置部分)				
	ATmega8	ATmega48	ATmega88	ATmega168
Flash(KB)	8	4	8	16
E^2PROM(B)	512	256	512	512
快速寄存器	32	32	32	32
SRAM(B)	1K	512	1K	1K
Max I/O Pins	23	23	23	23
中断数目	18	26	26	26
外部中断口	2	26	26	26
SPI	1	1+USART	1+USART	1+USART
UART	1	1	1	1
TWI	1	1	1	1
硬件乘法器	Y	Y	Y	Y
8 位定时器	2	2	2	2
16 位定时器	1	1	1	1
PWM通道	3	3	3	3
实时时钟RTC	Y	Y	Y	Y
10 位A/D通道	8	8	8	8
模拟比较器	Y	Y	Y	Y
掉电检测BOD	Y	Y	Y	Y
Watchdog	Y	Y	Y	Y
片内系统时钟	Y	Y	Y	Y
debugWIRE	-	Y	Y	Y
在线编程ISP	Y	Y	Y	Y
自编程SPM	Y	Y	Y	Y
Vcc(v)(最低) (最高)	2.7 5.5	1.8 5.5	1.8 5.5	1.8 5.5
系统时钟(MHz)	0-16	0-20	0-20	0-20
封装形式	PDIP28 MLF32 TQFP32	PDIP28 MLF32 TQFP32	PDIP28 MLF32 TQFP32	PDIP28 MLF32 TQFP32

在AVR系列单片机中,ATmega16是一款中档功能的AVR芯片,它的引脚数为40(44 TQFP),在片内集成了1K字节的SRAM、16K字节的Flash、512个字节的EEPROM,2个8位、1个16位共3个超强功能的定时器/计数器,以及USART、SPI、多路10位ADC、WDT、RTC、ISP、IAP、TWI（I^2C)、片内高精度RC振荡器等多种功能的接口和特性,较全面的体现了AVR的特点,不仅适合对AVR了解和使用的入门起步学习,同时也满足一般的普通应用,在产品中得到了大量的使用。

在本书中,我们将以AVR的ATmega16 为主线,逐步介绍AVR单片机的内部结构,以及各功能部件的使用方法。同时我们与www.ouravr.com网站合作,共同研制开发制作了“AVR-51多功能实验板”与本书配套。书中的实验均可在该板上实现。该实验板具有非常高的性/价比,不仅能够配合AVR单片机使用,同时也能完全适合8051类型的单片机使用,非常适合初学者使用学习和实验。读者可以通过访问www.ouravr.com网站购买。

表 1.4 megaAVR系列单片机(中高部分)

	ATmega8515	ATmega8535	ATmega16	ATmega32	ATmega64	ATmega162	ATmega165	ATmega169	ATmega128	ATmega2560L
Flash(KB)	8	8	16	32	64	16	16	16	128	256
E^2PROM(B)	512	512	512	1K	2K	512	512	512	4K	4K
快速寄存器	32	32	32	32	32	32	32	32	32	32
SRAM(B)	512	512	1K	2K	4K	1K	1K	1K	4K	8K
I/O Pins	35	32	32	32	53	35	54	54	53	86
中断数目	16	20	20	19	34	28	23	23	34	57
外部中断口	3	3	2	3	8	3	17	17	8	32
SPI	1	1	1	1	1	1	1+USI	1+USI	1	1+USART
SUART	1	1	1	1	2	2	1	1	2	4
TWI	-	Y	Y	Y	Y	-	Y	Y	Y	Y
硬件乘法器	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
8 位定时器	1	2	2	2	2	2	2	2	2	2
16 位定时器	1	1	1	1	2	2	1	1	2	4
PWM通道	3	4	3	4	8	4	4	4	8	16
实时时钟RTC			Y	Y	Y	Y	Y	Y	Y	Y
10 位A/D通道	-	8	8	8	8	-	8	8	8	16
模拟比较器	-	Y	Y	Y	Y	Y	Y	Y	Y	Y
掉电检测BOD	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Wacthdog	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
片内系统时钟	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
JTAG接口	-	-	Y	Y	Y	Y	Y	Y	Y	Y
在线编程ISP	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
自编程SPM	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Vcc(v)(最低) (最高)	2.7 5.5	2.7 5.5	2.7 5.5	2.7 5.5	2.7 5.5	2.7 5.5	2.7 5.5	2.7 5.5	2.7 5.5	1.8 5.5
系统时钟 (MHz)	0-16	0-16	0-16	0-16	0-16	0-16	0-16	0-16	0-16	0-16
封装形式	TQFP44 PDIP40 MLF44 PLCC44	TQFP44 PDIP40 MLF44 PLCC44	PDIP40 MLF44 TQFP44	PDIP40 MLF44 TQFP44		PDIP40 MLF44 TQFP44			TQFP64 MLF64	TQFP100

1.3.4 AVR与51单片机

在单片机发展的历程中，51 单片机做出了非常重要的贡献。

今天的所谓的51单片机实际上是一个总概念,泛指所有的采用了 Inter 公司的MCS-51内核结构，或称为与 MCS-51 兼容的那些单片机。典型的代表为 Inter 公司生产的 8051 系列单片机。目前国际上仍有许多半导体公司和生产厂以 MCS-51 为内核，推出了经过改进和扩展的，满足各种嵌入式应用的多种类型和型号的 51 兼容单片机。所以 51 单片机还是在单片嵌入式系统的应用中占有着非常主要的地位。

国内高校中的单片机系统课程与教学，20 几年来基本上都是以 51 类型的单片机来作为构成单片机系统的典型控制芯片介绍的，培养出大批熟悉、了解以及掌握 51 单片机的工

程师和技术人员，出版了大批与 51 单片机相关的教材和应用参考书。因此直到现在，国内的 51 单片机还是有着相当大的用户，在大部分学校的教学中，还是采用 51 类型的单片机作为学习的典型芯片。

从应用和市场的角度看，51 单片机仍旧能够满足许多应用系统的需求，并且有着价格最低廉、最多的参考资料和例程等许多优点。除此之外，现在许多半导体公司和生产厂也不断的在继续推出多种类型和型号的，以 MCS-51 为内核、经过比较大的改进和扩展的 51 兼容 SOC 单片机，其性能比标准的 8051 单片机要高的多，能够满足许多更高需求应用。

但是由于 MCS-51 本身的内核结构的局限性，51 单片机，尤其是标准 51 架构的单片机，在性能、技术和硬软件设计理念等多方面已经落后了。尤其是标准 51 架构的单片机，从技术角度看，已经跟不上单片机流行和发展的趋势了。

随着单片机系统技术的发展，目前市场上出现了许多新型的 8 位芯片。其中 ATMEL 公司的 AVR 发展尤为引人注目。AVR 采用了 RISC 结构，其在速度、内存容量、外围接口的集成化程度、以及向串行扩展，更适合使用高级语言编程的等众多的特性，以及其所使用的开发技术和防真调试技术等方面，都充分体现出和代表了当前单片嵌入式系统发展的趋势。也正是由于这些显著特点，和具有极高的性价比，使得 AVR 得到广泛的应用，在短时间内成为市场上的主流芯片之一。

因此，从教育的长远和发展眼光出发，我们的教学与学习的目标应该更高些，要相应的改变教学内容、教学方式和学习方式，充分体现和融入新的技术、新的硬软件系统设计理念和方法，为培养适应当今技术发展的嵌入式系统工程师和打好坚实的基础，以满足社会对高水平人才的需求。

思考与练习

1. 什么是通用计算机系统？什么是嵌入式计算机系统？两种系统在应用领域和技术构成等方面有那些相同点和区别？
2. 嵌入式计算机系统有哪几种类型？通过网络、杂志与广告了解各种可以构成嵌入式系统的核心部件的性能、价格与应用领域。
3. 为什么说单片机系统是典型的嵌入式系统？列举几个你所知道的单片嵌入式系统的产品和应用。
4. 通过网络、杂志与广告了解国内外主要的单片机生产商，以及它们的产品型号、主要性能和特点，以及相应的开发系统和工具。
5. 什么是单片机？单片机有何特点？
6. 单片机的主要技术发展方向是什么？
7. 简述单片嵌入式系统的系统结构，并以具体实例（产品）为例，说明系统结构中各个部分的具体构成与功能。
8. ATmega 系列单片机有那些特点？这些特点是否符合单片机的主要发展方向？
9. 将程序存储器集成到单片机内有和优点和不足？片内集成 EEPROM、FlashROM 以及 MaskROM、OTPROM 的单片机各有什么特点？
10. 大多数的 AVR 单片机内部都含有 RAM、FlashROM、EEPROM，请给出它们的用途、性能和特点，并举例说明如何使用。
11. 什么是 ISP 技术？采用 ISP 技术的单片机有什么优点？
12. 什么是 IAP 技术？IAP 与 ISP 的本质区别是什么？说明其主要用途。
13. 以串行总线方式为主的外围扩展方式有什么优点？
14. 在单片机中集成了那些常用的硬件接口电路？简单举例说明其功能和作用。

本章参考文献：

1.《AVR快速导引》（英文，CDROM），ATMEL，www.atmel.com

第2章 AVR单片机的基本结构

单片机是构成单片机嵌入式系统的核心器件。本章首先将介绍一般单片机的基本结构和组成，使大家对单片机芯片的内部硬件有基本了解和认识。掌握了单片机的基本结构和组成，对学习、了解任何一种类型单片机的工作原理，编写单片机的系统软件以及和设计外围电路都是非常重要的。

AVR 是美国 ATMEL 公司推出的一款采用 RISC 指令的 8 位高速单片机。本章将以 ATmega16 为主线，介绍和讲述 AVR 单片机内核的基本结构、引脚功能、工作方式等。深入的理解和掌握 AVR 的基本结构，对后续章节的学习、以及对实际的应用 AVR 单片机都是非常重要的。

2.1 单片机的基本组成

2.1.1 单片机的基本组成结构

单片机嵌入式系统的核心部件是单片机，其结构特征是将组成计算机的基本部件集成在一块晶体芯片上，构成一片具有特定功能的单芯片计算机—单片机。一片典型单片机芯片内部的基本组成结构如图 2-1 所示。

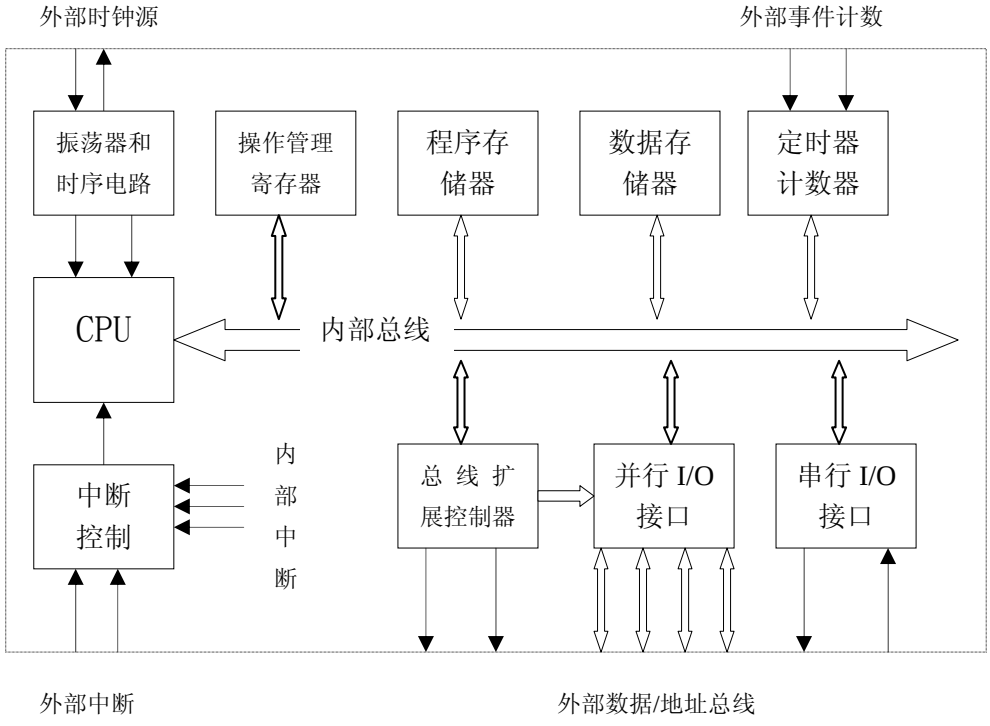

图 2-1 典型单片机的基本组成结构

从单片机的基本组成可以看出，在一片（单片机）芯片中，集成了构成一个计算机系统的最基本的单元：如 CPU、程序（指令）存储器、数据存储器、各种类型的输入/输出接口等。CPU 同各基本单元通过芯片内的内部总线（包括数据总线、地址总线和控制总线）连接。

一般情况下，内部总线中的数据总线宽度(或指 CPU 的字长)也是标定该单片机等级的一个重要指标。一般讲，低档单片机的内部数据总线宽度为 4 位（4 位机），普通和中档单片机的内部数据总线宽度一般为 8 位（8 位机），高档单片机内部数据总线宽度为 16 或 32 位。内部数据总线宽度越宽，单片机的处理速度也相应的提高，功能也越强。

2.1.2 单片机基本单元与作用

下面分别对单片机芯片中所集成的各个组成部分予以简要介绍。

1. MCU 单元(Microcontroller Unit)

MCU 单元部分包括了 CPU、时钟系统、复位、总线控制逻辑等电路。CPU 是按照面向测控对象、嵌入式应用的要求设计的，其功能有进行算术、逻辑、比较等运算和操作，并将结果和状态信息与存储器以及状态寄存器进行交换（读/写）。时钟和复位电路实现上电复位、信号控制复位，产生片内各种时钟及功耗管理等。总线控制电路则产生各类控制逻辑信号，满足 MCU 对内部和外部总线的控制。其中，内部总线用以实现片内各单元电路的协调操作和数据传输，而外部总线控制用于单片机外围扩展的操作管理。

2. 片内存储器

单片机的存储器一般分成程序存储器和数据存储器，它们往往构成相互独立的两个存储空间，分别寻址，互不干扰。在这一点上，与通用计算机系统的结构是不同的。通用计算机系统通常采用“Von-Neumann”结构，在这种结构体系中采用了单一的数据总线用于指令和数据的存取，因此数据和指令是存放在同一个存储空间中的，CPU 使用同一条数据总线与数据和程序进行交换,如在计算机原理课程中介绍的 8086/8088。而单片机的内部结构通常使用“Harvard”体系结构，在这种体系中采用分开的指令和数据总线，以及分开的指令和数据地址空间。单片机采用 Harvard 双（多）总线结构的优点是，指令和数据空间完全分开，分别通过专用的总线同 CPU 交换，可以实现对程序和数据的同时访问，提高了 CPU 的执行速度和数据的吞吐率。

早期的单片机，如典型的 8031 单片机，在片内只集成少量的数据存储器 RAM（128/256 字节），没有程序存储器。因此程序存储器和大容量的数据存储器需要进行片外的扩展，增加外围的存储芯片和电路，这给构成嵌入式系统带来了麻烦。后期的单片机则在片内集成了相当数量的程序存储器，如与 8031 兼容的 AT89S51、AT89S52 在片内集成了 4K/8K 的 Flash 程序存储器。而现在新型的单片机，则在片内集成了更多数量和更多类型的存储器。如 AVR 系列的 ATmega16 在片内就集成了 16K 字节的 Flash 程序存储器,1K 字节的 RAM 数据存储器，以及 512 字节的 EEPROM 数据存储器，这就大大方便了应用。

3. 程序存储器

程序存储器用于存放嵌入式系统的应用程序。由于单片机嵌入式系统的应用程序在开发调试完成后不需要经常改变，因此单片机的程序存储器多采用只读型 ROM 存储器，用于永久性的存储系统的应用程序。为适应不同产品、用户和不同场合的需要，单片机的程序存储器有以下几种不同形式：

①ROMLess 型。该种形式的单片机片内没有集成程序存储器，使用时必须在单片机外部扩展一定容量的 EPROM 器件。因此，使用这种类型的单片机就必须使用并行扩展总线，增加芯片，增加了硬件设计的工作量。

②EPROM 型。单片机片内集成了一定数量的 EPROM 存储器用于存放系统的应用程序。这类单片机芯片的上部开有透明窗口，可通过约 15 分钟的紫外线照射来擦除存储器中的程序，再使用专用的写入装置写入程序代码和数据，写入次数一般为几十次。

③MaskROM 型。使用种类型的单片机时，用户要将调试好的应用程序代码交给单片机的生产厂家，生产商在单片机芯片制造过程的掩膜工艺阶段将程序代码掩膜到程序存储器中。这种单片机便成为永久性专用的芯片，系统程序无法改动，适合于大批量产品的生产。

④OTPROM 型。这种类型的单片机与 MaskROM 型的单片机有相似的特点。生产商提供新的单片机芯片中的程序存储器可由用户使用专用的写入装置一次性编程写入程序代码，写入后也无法改动了。这种类型的单片机也是适用于大批量产品的生产。

⑤FlashROM 型。这是一种可供用户多次擦除和写入程序代码的单片机。它的程序存储器采用快闪存储器（FlashMemory）,现在可实现大于 1 万次的写入操作。

内部集成 FlashROM 型单片机的出现，以及随着 Flash 存储器价格的下降，使得使用 FlashROM 的单片机正在逐步淘汰使用其它类型程序存储器的单片机。由于 FlashROM 可多次擦除（电擦除）和写入的特性，加上新型的单片机又采用了在线下载 ISP 技术（In System Program--既无需将芯片从系统板上取下，直接在线将新的程序代码写入单片机的程序存储器中。），不仅为用户在嵌入式系统的设计、开发和调试带来了极大的方便，而且也适用于大批量产品的生产，并为产品的更新换代提供了更广阔的空间。

4. 数据存储器

单片机在片内集成的数据存储器一般有两类：随机存储器 RAM 和电可擦除存储器 EEPROM。

①随机存储器 RAM。在单片机中，随机存储器 RAM 是用来存储系统程序在运行期间的工作变量和临时数据的。一般在单片机内部集成一定容量（32 字节至 512 字节或更多）的 RAM。这些小容量的数据存储器以高速 RAM 的形式集成在单片机芯片内部，作为临时的工作存储器使用，可以提高单片机的运行速度。

在单片机中，常把内部寄存器(如工作寄存器、I/O 寄存器等)在逻辑上也划分在 RAM 空间中，这样即可以使用专用的寄存器指令对寄存器进行操作，也可将寄存器当做 RAM 使用，为程序设计提供了方便和灵活性。

对一些需要使用大容量数据存储器的系统，就需要在外部扩展数据存储器。这时，单片机就必须具备并行扩展总线的功能，同时外围也要增加 RAM 芯片和相应的地址锁存、地址译码等电路。这不仅增加了硬件设计的工作量，产品的成本，同时降低了系统的可靠性。

目前许多新型单片机片内集成的 RAM 容量越来越大。片内集成的 RAM 容量增加，不仅减少了在片外扩展 RAM 的必要性，提高了系统的可靠性，而且更重要的是，使得单片嵌入式系统的软件设计思想和方法有了许多的改变和发展，给编写系统程序带来很大的方便，更加有利于结构化、模块化的程序设计。

②电可擦除存储器 EEPROM。一些新型的单片机，在芯片中还集成了电可擦除存储器型 EEPROM 的数据存储器。这类数据存储器用于存放一些永久或比较固定的系统参数，如放大倍率、电话号码、时间常数等。EEPROM 的寿命大于 10 万次，具有掉电后不丢失数据的特点，并且通过系统程序可以随时修改,这些特性都给用户设计开发产品带来极大的方便和想象空间。

5. 输入/输出（I/O）端口

为了满足嵌入式系统“面向控制”的实际应用需要，单片机提供了数量众多、功能强、使用灵活的输入/输出端口，简称 I/O。端口的类型可分为以下几种类型：

① 并行总线输入/输出端口（并型 I/O 口）。用于外部扩展和扩充并行存储器芯片或并行 I/O 芯片等使用，包括数据总线、地址总线和读写控制信号等。

② 通用数字 I/O 端口。用于外部电路逻辑信号的输入和输出控制。

③ 片内功能单元的输入/输出端口。如：定时器/计数器的计数脉冲输入，外部中断源信号的输入等。

④ 串行 I/O 通信口。用于系统之间或与采用专用串行协议的外围芯片之间的连接和交换数据。如：UART 串行接口（RS-232），I^2C 串行接口，SPI 串行接口，USB 串行口等。

⑤ 其它专用接口。一些新型的单片机还在片内集成了某些专用功能的模拟或数字的 I/O 端口，如 A/D 输入、D/A 输出接口，模拟比较输入端口，脉宽调制（PWD）输出端口等。更有的单片机还将 LCD 液晶显示器的接口也集成到单片机芯片中了。

为了减少芯片引脚的数量，又能提供更多性能的 I/O 端口给用户使用，大多数的单片机

都采用了 I/O 端口复用技术，既某一端口，它即可作为一般通用的数字 I/O 端口使用，也可作为某个特殊功能的端口使用，用户可根据系统的实际需要来定义使用。这样就为设计开发提供了方便，大大拓宽了单片机的应用范围。

6. 操作管理寄存器。

操作管理寄存器也是单片机芯片中的重要组成部分之一。它的功能是管理、协调、控制和操作单片机芯片中的各功能单元的使用和运行。这类寄存器的种类有：状态寄存器、控制寄存器、方式寄存器、数据寄存器等等。各种寄存器的定义、功能、状态、相互之间的关系和应用相对比较复杂，而且往往同相应的功能单元的使用紧密相关，因此，用户应非常熟悉各个寄存器的作用以及如何与不同的功能单元的配合使用，这样才能通过程序指令对其编程操作，以实现对单片机芯片中各种功能的正确使用，充分发挥单片机的所有特点和性能，设计和开发出高性能、低成本的电子产品。可以这样讲，当你对某个单片机芯片中各个操作管理寄存器的作用、功能、定义非常透彻的掌握了，那么你已经完全精通和能够熟练使用该单片机了。

2.2 ATmega16 单片机的组成

ATMEL 公司的 AVR 单片机是一种基于增强 RISC 结构的、低功耗、CMOS 技术、8 位微控制器(Enhanced RISC Microcontroller)，目前有 Tiny、Mega 两个系列 50 多种型号。它们的功能和外部的引脚各有不同，小到 8－12 个引脚，多到 100 个引脚，但它们内核的基本结构是一样的，指令系统相容。本书将以性能适中的 ATmega16 为主线，介绍和讲述 AVR 单片机的组成，以及如何应用在嵌入式系统中。在正式的产品开发与设计时，设计者可根据系统的实际需要选择合适型号的 AVR 单片机。

2.2.1 AVR单片机的内核结构

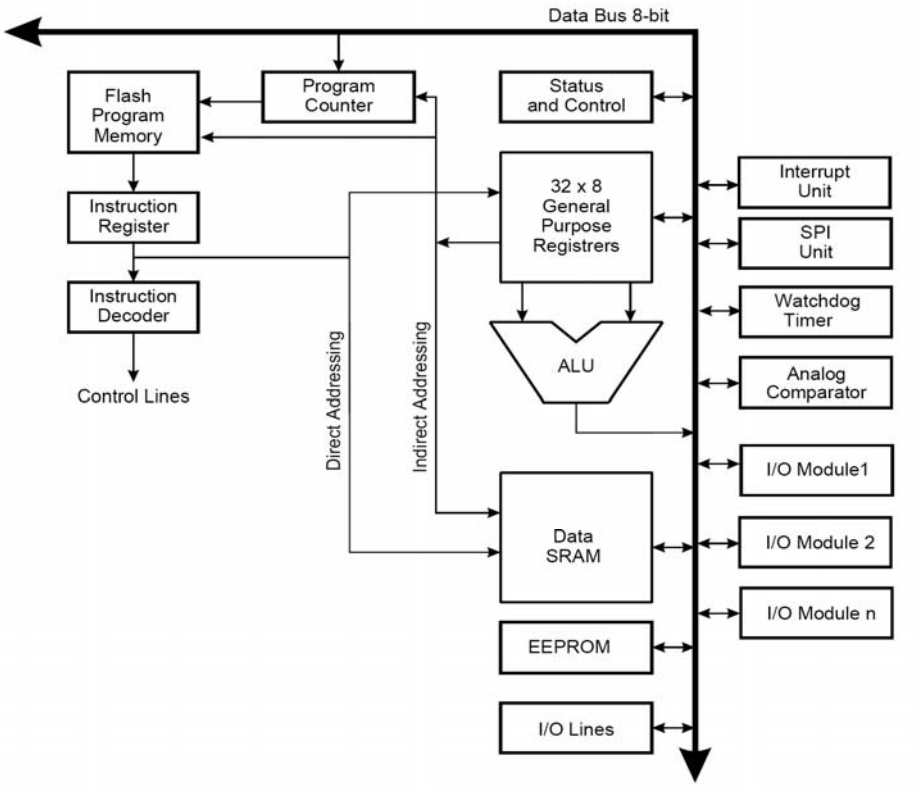

图 2-2 AVR 单片机的内核结构示意图

尽管 AVR 单片机系列有几十种的型号，但它们有着相同的内核结构，指令兼容。图 2-2 为典型的 AVR 单片机的内核结构图。

为了提高 MCU 并行处理的运行效率，AVR 单片机采用了程序存储器和数据存储器使用不同的存储空间和存取总线的 Harvard 结构。算术逻辑单元（ALU）使用单级流水线操作方式对程序存储器进行访问，在执行当前一条指令的同时，也完成了从程序存储器中取出下一条将要执行指令的操作，因此执行一条指令仅需要一个时钟周期。

在 AVR 的内核中，由 32 个访问操作只需要一个时钟周期的 8 位通用工作寄存器组成了“快速访问寄存器组”。“快速访问”意味着在一个时钟周期内执行一个完整的 ALU 操作。这个 ALU 操作中包含三个过程：从寄存器组中取出两个操作数，操作数被执行，将执行结果写回目的寄存器中。这三个过程是在一个时钟周期内完成的，构成一个完整的 ALU 操作。

在 32 个通用工作寄存器中，有 6 个寄存器可以合并成为 3 个 16 位的，用于对数据存储器空间进行间接寻址的间接地址寄存器（存放地址指针），以实现高效的地址计算。这 3 个 16 位的间接地址寄存器称为：X 寄存器，Y 寄存器和 Z 寄存器。其中 Z 寄存器还能作为间接寻址程序存储器空间的地址寄存器，用于在 Flash 程序存储器空间进行查表等操作。

AVR 的算术逻辑单元（ALU）支持寄存器之间，立即数与寄存器之间的算术与逻辑运算功能，以及单一寄存器操作。每一次运算操作的结果将影响和改变状态寄存器（SREG）的值。使用条件转移、无条件转移和调用指令，可以直接访问全部 Flash 程序存储器空间以及控制程序的执行顺序。大部分 AVR 指令为单一 16 位格式，只有少数指令为 32 位格式。因此，AVR 的程序存储器单元为 16 位，即每个程序地址(两字节地址)单元存放一条单一的 16 位指令字。而一条 32 位的指令字，则要占据 2 个程序存储器单元。

ATmega16 单片机的 Flash 程序存储器空间可以分成两段：引导程序段（Boot program section）和应用程序段（Application program section）。两个段的读写保护可以分别通过设置对应的锁定位（Lock bits）来实现。在引导程序段内驻留的引导程序中，可以使用 SPM 指令，实现对应用程序段的写操作（实现在应用自编程 IAP 功能，使系统能够自己更新系统程序）。

在响应中断服务和子程序调用过程时，程序计数器 PC 中的返回地址将被存储于堆栈之中。堆栈空间将占用数据存储器（SRAM）中一段连续的地址。因此，堆栈空间的大小仅受到系统总的数据存储器（SRAM）的大小以及系统程序对 SRAM 的使用量的限制。用户程序应在系统上电复位后，对一个 16 位的堆栈指针寄存器 SP 进行初始化设置（或在子程序和中断程序被执行之前）。

在 AVR 中，所有的存储器空间都是线性的。数据存储器（SRAM）可以通过 5 种不同的寻址方式进行访问。

AVR 的中断控制由 I/O 寄存器空间的中断控制寄存器和状态寄存器中的全局中断允许位组成。每个中断都分别对应一个中断向量（中断入口地址）。所有的中断向量构成了中断向量表，该中断向量表位于 Flash 程序存储器空间的最前面。中断的中断向量地址越小，其中断的优先级越高。

I/O 空间为连续的 64 个 I/O 寄存器空间，它们分别对应 MCU 各个外围功能的控制和数据寄存器地址，如控制寄存器、定时器/计数器、A/D 转换器及其他的 I/O 功能等。I/O 寄存器空间可使用 I/O 寄存器访问指令直接访问，也可将其映射为通用工作寄存器组后的数据存储器空间，使用数据存储器访问指令进行操作。I/O 寄存器空间在数据存储器空间的映射地址为 $020～$05F。

AVR 单片机的性能非常强大，所以它的内部结构相对 8031 结构的单片机要复杂。对于刚开始接触和学习单片机的人员，以及了解 8051 结构单片机的人来讲，在这里尽管不会马上理解 AVR 内核的全部特点，但通过以后的逐步学习，应逐渐深入的体会和掌握它的原理，

这对于熟练的应用 AVR 设计开发产品，以及将来学习使用更新的单片机都会有很大的帮助。技术是在不断的发展的。

2.2.2 典型AVR芯片ATmega16 特点

AVR 系列单片机中比较典型的芯片是 ATmega16。这款芯片具备了 AVR 系列单片机的主要的特点和功能，不仅适合应用于产品设计，同时也方便初学入门。其主要特点有：

（1）采用先进RISC结构的AVR内核

131条机器指令，且大多数指令的执行时间为单个系统时钟周期；

32个8位通用工作寄存器；

工作在16MHz时具有16MIPS的性能。

配备只需要2个时钟周期的硬件乘法器

（2）片内含有较大容量的非易失性的程序和数据存储器

16K字节在线可编程（ISP）Flash程序存储器（擦除次数>1万次），采用Boot Load技术支持IAP功能；

1K字节的片内SRAM数据存储器，可实现3级锁定的程序加密；

512个字节片内在线可编程EEPROM数据存储器（寿命>10万次）；

（3）片内含JTAG接口

支持符合JTAG标准的边界扫描功能用于芯片检测；

支持扩展的片内在线调试功能

可通过JTAG口对片内的Flash、EEPROM、配置熔丝位和锁定加密位实施下载编程；

（4）外围接口

2个带有分别独立、可设置预分频器的8位定时器/计数器；

1个带有可设置预分频器、具有比较、捕捉功能的16位定时器/计数器；

片内含独立振荡器的实时时钟RTC；

4路PWM通道；

8路10位ADC

面向字节的两线接口TWI（兼容I^2C硬件接口）；

1个可编程的增强型全双工的，支持同步/异步通信的串行接口USART；

1个可工作于主机/从机模式的SPI串行接口（支持ISP程序下载）；

片内模拟比较器；

内含可编程的，具有独立片内振荡器的看门狗定时器WDT；

（5）其它的特点

片内含上电复位电路以及可编程的掉电检测复位电路BOD；

片内含有1M/2M/4M/8M，经过标定的、可校正的RC振荡器，可作为系统时钟使用；

多达21个各种类型的内外部中断源；

有6种休眠模式支持省电方式工作；

（6）宽电压、高速度、低功耗

工作电压范围宽：ATmega16L 2.7—5.5v，ATmega16 4.5—5.5v；

运行速度：ATmega16L 0—8M，ATmega16 0—16M；

低功耗：ATmega16L工作在1MHz、3v、25度时的典型功耗为，正常工作模式 1.1mA，空闲工作模式 0.35mA，掉电工作模式 <1uA；

（7）芯片引脚和封装形式

ATmega16共有32个可编程的I/O口（脚），芯片封装形式有40引脚的PDIP、44引脚的TQFP和44引脚的MLF封装。

2.2.3 外部引脚与封装

ATmega16 单片机有三种形式的封装：40 脚双列直插 PDIP、44 脚方形的 TQFP 和 MLF 形式（贴片形式）。其外部引脚封装如图 2-3 所示。

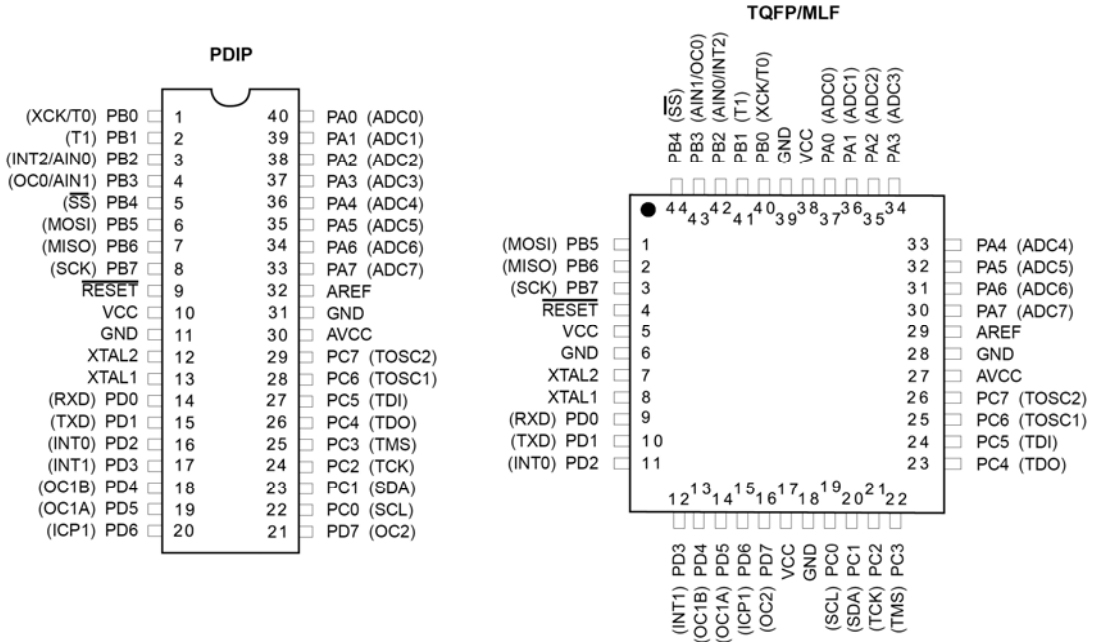

图 2-3　ATmage16 外部引脚与封装示意图

其中，各个引脚的功能如下:

（1）电源、系统晶振、芯片复位引脚

Vcc: 芯片供电（片内数字电路电源）输入引脚，使用时连接到电源正极。

AVcc：为端口 A 和片内 ADC 模拟电路电源输入引脚。不使用 ADC 时，直接连接到电源正极；使用 ADC 时，应通过一个低通电源滤波器与 Vcc 连接。

AREF：使用 ADC 时，可作为外部 ADC 参考源的输入引脚。

GND: 芯片接地引脚，使用时接地。

XTAL2：片内反相振荡放大器的输出端。

XTAL1：片内反相振荡放大器和内部时钟操作电路的输入端。

RESET：RESET 为芯片复位输入引脚。在该引脚上施加（拉低）一个最小脉冲宽度为 1.5us 的低电平,将引起芯片的硬件复位（外部复位）。

（2）32 根 I/O 引脚，分成 PA、PB、PC 和 PD 四个 8 位端口，他们全部是可编程控制的双（多）功能复用的 I/O 引脚（口）。

四个端口的第一功能是通用的双向数字输入/输出（I/O）口，其中每一位都可以由指令设置为独立的输入口，或输出口。当 I/O 设置为输入时，引脚内部还配置有上拉电阻，这个内部的上拉电阻可通过编程设置为上拉有效或上拉无效。

如果 AVR 的 I/O 口设置为输出方式工作，当其输出高电平时，能够输出 20mA 的电流，而当其输出低电平时，可以吸收 40mA 的电流。因此 AVR 的 I/O 口驱动能力非常强，能够直接驱动 LED 发光二极管、数码管等。而早期单片机 I/O 口的驱动能力只有 5mA，驱动 LED 时，还需要增加外部的驱动电路和器件。

芯片 Reset 复位后，所有 I/O 口的缺省状态为输入方式，上拉电阻无效，即 I/O 为输入高阻的三态状态。

以上我们简单介绍了 ATmega16 单片机的主要特性以及引脚封装。可以看出，小小的一

块芯片，其内部的组成结构却是相当复杂的。也正式这种复杂，加上多样的程序，才使得单片机在实际应用中变化无穷。

下面，我们从 ATmega16 的内部结构出发，逐步的介绍它的工作原理和使用方法。

2.3 ATmega16 内部结构

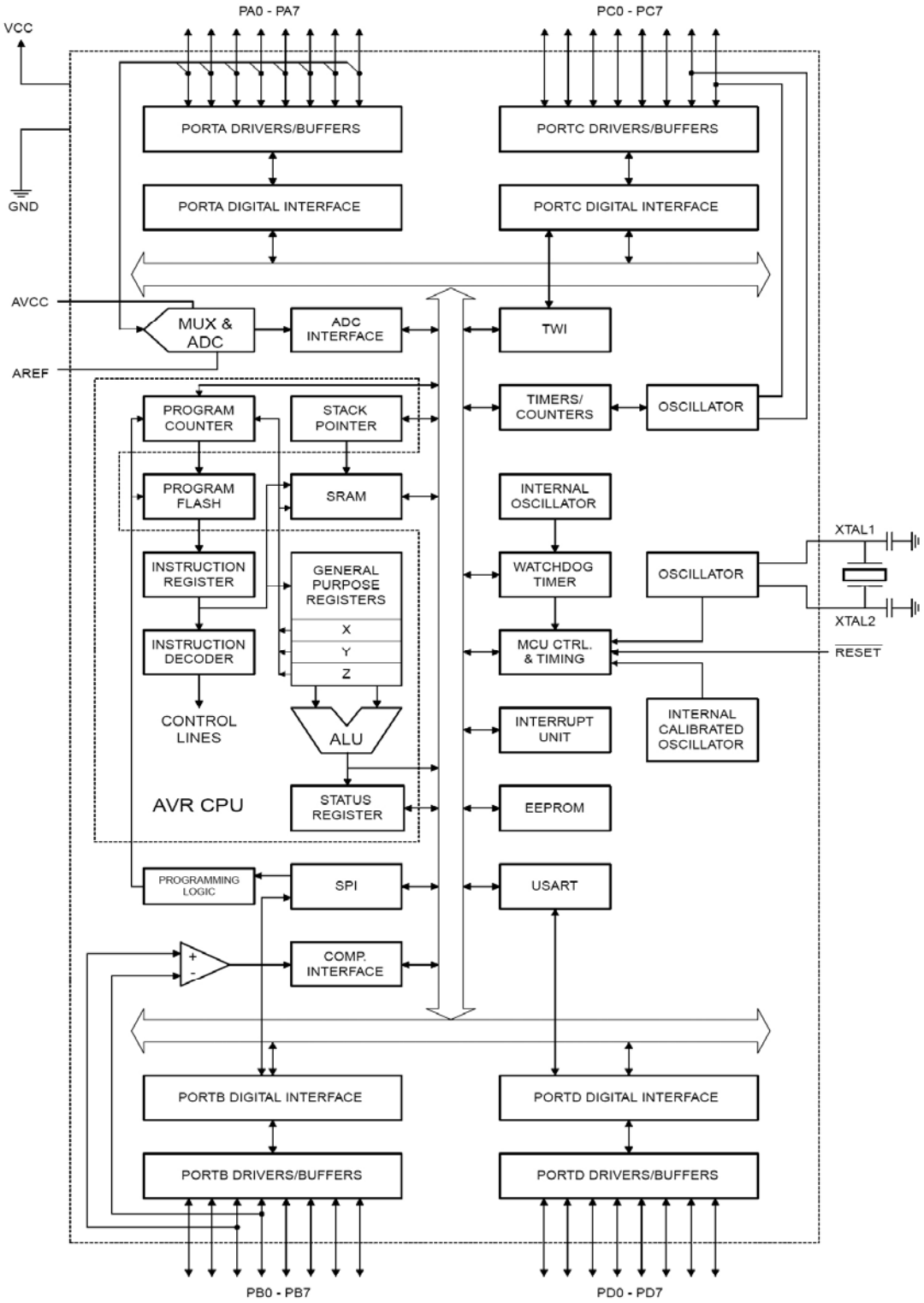

图 2-4 ATmage16 的结构框图

图 2-4 是 ATmage16 的结构框图。它是在 AVR 内核（图 2-3）的基础上，具体化的一个实例。从图中可以看出，ATmega16 内部的主要构成部分有：

- AVR CPU 部分。包括：ALU 运算逻辑单元、32 个 8 位快速访问通用寄存器组（寄存器文件）、程序计数器 PC、指令寄存器、指令译码器。
- 程序存储器 Flash。
- 数据存储器 RAM 和 EEPROM。
- 各种功能的外围接口、I/O，以及与它们相关的数据、控制、状态寄存器等。

2.3.1 AVR中央处理器CPU

AVR CPU 是单片机的核心部分，它由运算逻辑单元 ALU、程序计数器 PC、指令寄存器、指令译码器等部件组成。

1. 运算逻辑单元 ALU

运算逻辑单元 ALU 的功能是进行算术运算和逻辑运算，可对半字节（4 位）、单字节等数据进行操作。如能完成加、减、自动加 1、自动减 1、比较等算术运算和与、或、异或、求补、循环移位等逻辑操作。操作结果的状态，如产生进位、结果为零等状态信息将影响到状态寄存器 SREG 相应的标志位。

运算逻辑单元 ALU 还包含一个布尔处理器，用来处理位操作。它可执行置位、清零、取反等操作。

ATmega16 的 ALU 还能实现无符号数、有符号数以及浮点数的硬件乘法操作。一次硬件乘法操作的时间为 2 个时钟周期。

2. 程序计数器 PC、指令寄存器和指令译码器

程序计数器 PC 用来存放下一条需要执行指令在程序存储器空间的地址（指向 Flash 空间）。取出的指令存放在指令寄存器中，然后送入指令译码器产生各种控制信号，控制 CPU 的运行（执行指令）。

AVR 一条指令的长度大多数为 16 位，还有少部分为 32 位，因此 AVR 的程序存储器结构实际上是以字（16 位）为一个存储单元的。ATmega16 的程序计数器为 13 位，正好满足了对片内 8K 字（即手册上的 16K 字节）的 Flash 程序存储器空间直接寻址的需要，因此就不能（不支持）在外部扩展更多的程序存储器。

AVR CPU 在译码执行一条指令的同时，就将 PC 中指定的 Flash 单元中的指令取出，放入指令寄存器中（图中的 Instruction Register），构成了一级流水线运行方式。AVR 采用一级流水线技术，在当前指令执行的时候，就取出下一条将要执行的指令，加上大多数 AVR 指令的长度是一个字，就使得 AVR CPU 实现了一个时钟周期执行一条指令。采用这种结构，减少了取指令的次数，大大提高了 CPU 的运行速度，同时也提高了取指令操作的（系统的）可靠性。而在其它的 CISC 以及类似的 RISC 结构的单片机中，外部振荡器的时钟被分频降低到传统的内部指令执行周期，这种分频最大达 12 倍（例如，标准 8031 结构的单片机）。

三. 通用工作寄存器组

在 AVR 中，由命名为 R0～R31 的 32 个 8 位通用工作寄存器构成一个“通用快速工作寄存器组”，图 2-5 为通用快速工作寄存器组的结构图。

AVR CPU 中的 ALU 与这 32 个通用工作寄存器组直接相连，为了使 ALU 能够高效和灵活地对寄存器组进行访问操作，通用寄存器组提供和支持 ALU 使用 4 种不同的数据输入/输出的操作方式：

- 提供一个 8 位源操作数，并保存的一个 8 位结果
- 提供两个 8 位源操作数，并保存的一个 8 位结果
- 提供两个 8 位源操作数，并保存的一个 16 位结果

- 提供一个 16 位源操作数，并保存的一个 16 位结果

因此，AVR 大多数操作工作寄存器组的指令都可以直接访问所有的寄存器，而且多数这样的指令的执行时间是一个时钟周期。例如，从寄存器组中取出两个操作数，对操作数实施处理，处理结果回写到目的寄存器中。这三个过程是在一个时钟周期内完成的，构成一个完整的 ALU 指令操作。

在传统的基于累加器结构的单片机中（如 8051），则需要大量的程序代码来完成和实现在累加器和存储器之间的数据传送。如上面所介绍的操作过程就需要三条指令来实现：第一条完成从寄存器中取出源操作数；第二条完成对操作数实施处理；第三条将处理结果回写。这样就构成了累加器和存储器之间数据传送的瓶颈，影响了指令运行效率。

而在 AVR 单片机中，由于采用了 32 个通用工作寄存器构成快速存取寄存器组，相当于用 32 个通用工作寄存器代替了累加器，从而避免了在传统结构中的那种由于累加器和存储器之间频繁的数据传送交换而形成的瓶颈现象，又进一步提高了指令的运行效率和速度。

在 AVR 中，通用寄存器组与片内的数据存储器 SRAM 处在相同的空间，32 个通用寄存器被直接映射到用数据空间的前 32 个地址，如图 2-5 所示。虽然寄存器组的物理结构与 SRAM 不同，但是这种内存空间的组织方式为访问工作寄存器提供了极大的灵活性，如可以利用地址指针寄存器 X、Y 或 Z 实现对通用寄存器组的间接寻址操作。

寄存器名	RAM 空间地址	
R0	$0000	
R1	$0001	
R2	$0002	
……	……	
R14	$000E	
R15	$000F	
R16	$0010	
……		
R26	$001A	X 寄存器低位字节
R27	$001B	X 寄存器高位字节
R28	$001C	Y 寄存器低位字节
R29	$001B	Y 寄存器高位字节
R30	$001D	Z 寄存器低位字节
R31	$001F	Z 寄存器高位字节

图 2-5 通用工作寄存器组在 RAM 空间的地址分配图

2.3.2 系统时钟部件

1. 系统时钟

ATmega16 的片内含有 4 种频率（1/2/4/8M）的 RC 振荡源，可直接作为系统的工作时钟使用。同时片内还设有一个由反向放大器所构成的 OSC（Oscillator）振荡电路，外围引脚 XTAL1 和 XTAL2 分别为 OSC 振荡电路的输入端和输出端，用于外接石英晶体等，构成高精度的或其它标称频率的系统时钟系统。

系统时钟为控制器提供时钟脉冲，是控制器的心脏。系统时钟的频率是单片机的重要性能指标之一。系统时钟频率越高，单片机的执行节拍就越快，处理速度也越快。ATmega16 最高的工作频率为 16M（16MIPS），在 8 位单片机中算是佼佼者。但并不是系统时钟频率越快就越好，因为当时钟频率越高时，其耗电量也越大，也容易受到干扰（或干扰别人）。因此，在具体设计时，应根据实际产品的需要，尽量采用较低的系统时钟频率，这样不仅能降低了功耗，同时也提高了系统的可靠性和稳定性。

为 ATmega16 提供系统时钟源时，有三种主要的选择：（1）直接使用片内的 1/2/4/8M 的 RC 振荡源；（2）在引脚 XTAL1 和 XTAL2 上外接由石英晶体和电容组成的谐振回路，配合片内的 OSC（Oscillator）振荡电路构成的振荡源；（3）直接使用外部的时钟源输出的脉冲信号。方式 2 和方式 3 的电路连接见图 2-6(a)和 2-6(b)。

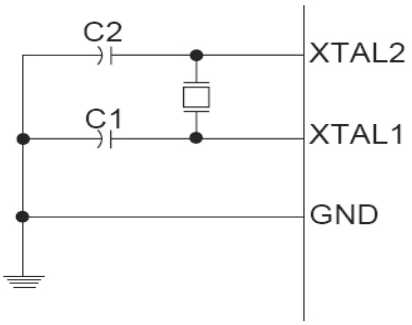

图 2-6(a)　外部接晶体的时钟电路

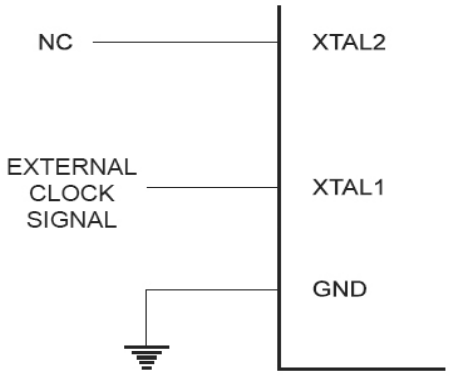

图 2-6(b)　直接使用外部时钟源

方式 2 是比较常用的方法，由于采用了外接石英晶体作为振荡的谐振回路，因此可以提供比较灵活的频率（由使用晶体的谐振频率决定）和稳定精确的振荡。在 XTAL1 和 XTAL2 引脚上加上由石英晶体和电容组成的谐振回路，与内部振荡电路配合就能产生系统需要时钟信号了。最常采用的晶体元件为一个石英晶体和两个电容组成谐振电路。晶体可在 0-16MHz 之间选择，电容值在 20pF-30pF 之间（最好与所选用的晶体相匹配）。

当对系统时钟电路的精度要求不高的话，可以使用第 1 种方式，即使用片内可选择的 1/2/4/8M 的 RC 振荡源作为系统时钟源，可以节省外接器件，此时 XTAL1 和 XTAL2 引脚悬空。

系统时钟电路产生振荡脉冲不经过分频将直接作为系统的主工作时钟 φ，同时它还作为芯片内部的各种计数脉冲，以及各种串口定时时钟等使用（可由程序设定分频比例）。

使用 AVR 时要特别注意：AVR 单片机有一组专用的，与芯片功能、特性、参数配置相关的可编程熔丝位。其中有几个专门的熔丝位（CKSEL3..0）用于配置芯片所要使用的系统时钟源的类型。

新芯片的缺省配置设定为使用内部 1M 的 RC 振荡源作为系统的时钟源。因此当第一次使用前，必须先正确的配置熔丝位，使其与使用的系统时钟源类型相匹配。另外，在配置其它熔丝位时，或进行程序下载时，千万不要对 CKSEL3..0 这几个熔丝位误操作，否则会组成芯片表面现象上的“坏死”，因为没有系统时钟源，芯片不会工作的。

关于 ATmega16 重要熔丝位的配置、使用方式、以及注意事项请参考附录 A。

2.　内部看门狗时钟

在 AVR 片内还集成了一个 1MHz 独立的时钟电路，它仅供片内的看门狗定时器（WDT）使用。因此，AVR 片内的 WDT 是独立硬件形式的看门狗，使用 AVR 可以省掉外部的 WDT 芯片。使用 WDT 可以有效的提高系统的可靠行。

2.3.3 CPU的工作时序

AVR CPU 的工作是由系统时钟（φ）直接驱动的，在片内不再进行分频。图 2-7 所示为 Harvard 结构和快速访问寄存器组的并行指令存取和指令执行时序。CPU 在启动后第一个时钟周期 T1 取出第一条指令，在 T2 周期便执行取出的指令，并同时又取出第二条指令，依次

进行。这种基于流水线形式的取指方式，使 AVR 可以以非常高的速度执行指令，获得高达 1MIPS / MHz 的效率。

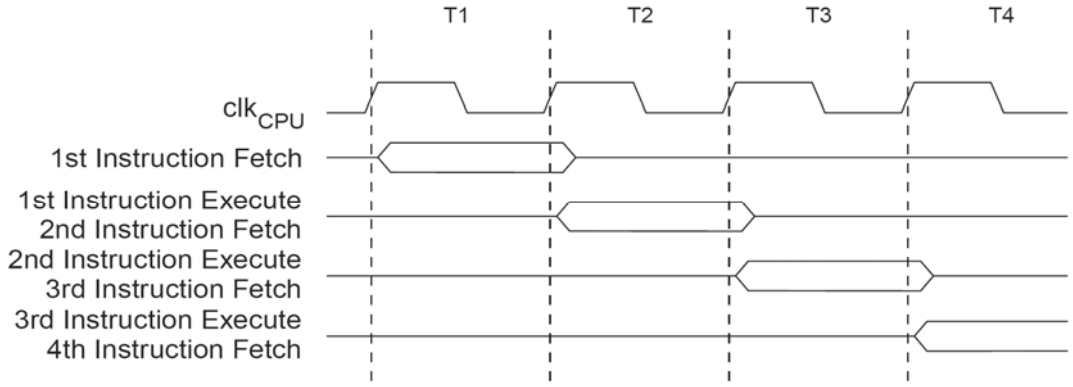

图 2-7 并行指令存取和指令执行

图 2-8 所示为 ALU 与寄存器堆操作单周期指令的执行时序。在单一时钟周期内，由 2 个寄存器提供操作数，ALU 执行相应的操作，最后将操作结果回送到目的寄存器中。

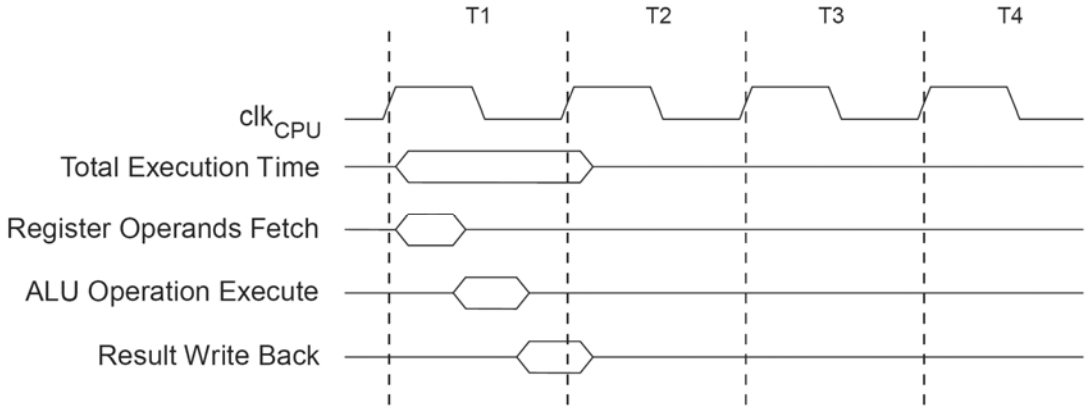

图 2-8 单周期 ALU 操作

AVR 对片内 SRAM 存储器的访问需要 2 个时钟周期。图 2-9 所示，在 2 个系统时钟周期内，ALU 完成对内部数据存储器 SRAM 访问的操作时序。

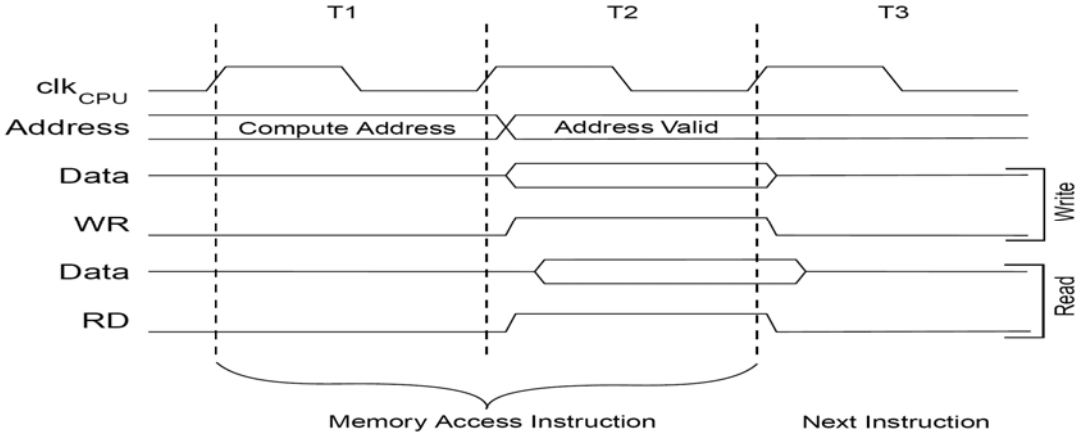

图 2-9 片内数据 SRAM 访问时序

2.3.4 存储器

AVR 单片机在片内集成了 Flash 程序存储器、SRAM 数据存储器和 EEPROM 数据存储器。三个存储器空间互相独立，物理结构也不同。程序存储器为闪存存储器 Flash，以 16 位（字）为一个存储单元，作为数据读取时，以字节为单位，而擦除、写入则是以页为单位的（不同型号 AVR 单片机一页的大小也不同）。SRAM 数据存储器是以 8 位（字节）为一个存储单元，编址方式采用与工作寄存器组、I/O 寄存器和 SRAM 统一寻址的方式。EEPROM 数据存储器也是以 8 位（字节）为一个存储单元，对其的读写操作都以字节为单位。有关存储器结构的详细介绍将在下一节叙述。

2.3.5 I/O端口

ATmega16 有四个 8 位的双向 I/O 端口 PA、PB、PC、PD，它们对外对应 32 个 I/O 引脚，每一位都可以独立地用于逻辑信号的输入和输出。在 5 伏工作电压下，输出时每个引脚可供出达 20mA 的驱动电流。而输入时，每个引脚可吸纳最大为 40mA 的电流，可直接驱动发光二极管 LED（一般 LED 的驱动电流为 10mA 左右）和小型继电器。

AVR 大部分的 I/O 端口都具备双重功能，分别同片内的各种不同功能的外围接口电路组合成一些可以完成特殊功能的 I/O 口，如定时器、计数器、串行接口、模拟比较器、捕捉器等。实际上，学习单片机的主要任务，就是了解、掌握单片机 I/O 端口的功能，以及如何正确设计这些端口与外围电路的连接构成一个嵌入式系统，并编程、管理和运用它们完成各种各样的任务。有关这些内容，将在后面的章节中逐步学习。

2.4 存储器结构和地址空间

2.4.1 支持ISP的Flash程序存储器

AVR 单片机包括 1K～128K 字节的片内支持 ISP 的 Flash 程序存储器。由于 AVR 所有指令为 16 位字或 32 位双字，故 Flash 程序存储器的结构为（512B～64Kb）x 16 位。Flash 存储器的使用寿命最少为 1 万次写/擦循环。ATmega16 单片机的程序存储器为 8K x 16（16K x 8），程序计数器 PC 宽为 13 位，以此来对 8K 字程序存储器地址进行寻址。

程序存储器的地址空间与数据存储器的地址空间是分开的，地址空间从$000 开始。如要在程序存储器中使用常量表，则常量表可以被设定在整个 Flash 地址空间中。

2.4.2 数据存储器SRAM空间

图 2-10 给出 ATmega16 单片机 SRAM 数据存储器的组织结构。全部共 1120 个数据存储器地址为线性编址，前 96 个地址为寄存器组（32 个 8 位通用寄存器），I/O 寄存器（64 个 8 位 I/O 寄存器），分配在 SRAM 数据地址空间的$0000～$001F，$0020～$005F。接下来的 1024 个地址是片内数据 SRAM，地址空间占用$0060～$045F。

CPU 对 SRAM 数据存储器的寻址方式分为 5 种：直接寻址、带偏移量的间接寻址、间接寻址、带预减量的间接寻址和带后增量的间接寻址。在寄存器堆中，寄存器 R26～R31 具有间接寻址指针寄存器的特性。ALU 可使用直接寻址的方式对整个存储器空间寻址操作。带偏移量的间接寻址方式可以寻址由寄存器 Y 和 Z 给出的基本地址附近的 63 个地址。当使用自动预减量和后增量的间接寻址方式时，3 个 16 位的地址寄存器 X、Y 和 Z 都可作为间接寻址的地址指针寄存器，寄存器中的地址指针值将根据操作指令的不同，自动被增加或减小。

32 个通用工作寄存器，64 个 I/O 寄存器，以及 ATmega16 单片机中 1024 个字节的数据 SRAM，都可通过上述的寻址方式进行访问操作。

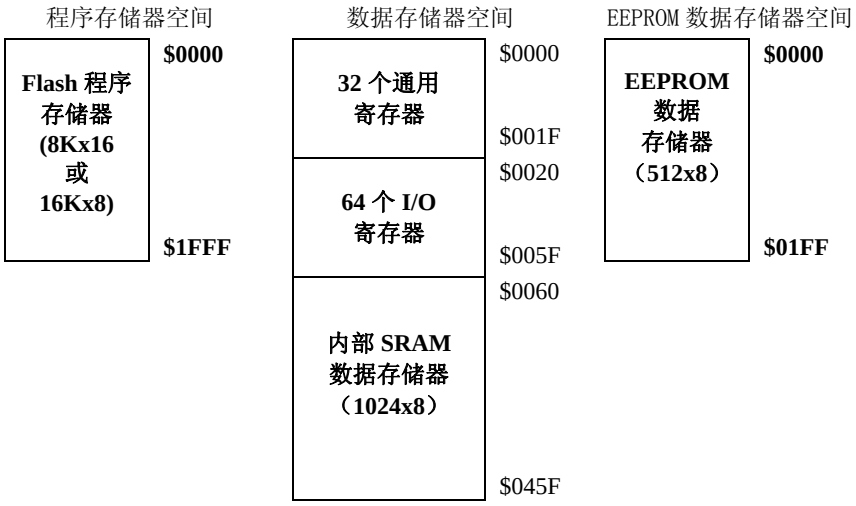

图 2-10 ATmeag16 存储器结构

ATmega16 不支持外部 SRAM 扩展。

2.4.3 内部EEPROM 存储器

AVR 系列单片机还包括 64B～4K 字节的 EEPROM 数据存储器。它们被组织在一个独立的数据空间中。这个数据空间采用单字节读写方式。EEPROM 的使用寿命至少为 10 万次写/擦循环。ATmega16 的 EEPROM 容量是 512 字节，地址范围为$0000～$01FF。EEPROM 数据存储器可用于存放一些需要掉电保护，而且比较固定的系统参数、表格等。

2.5 通用寄存器组与I/O寄存器

全面熟练地理解、掌握 AVR 的通用寄存器组与 I/O 寄存器的性能、特点、功能、设置和使用，是精通和熟练使用 AVR 单片机的关键。由于 AVR 有 32 个通用寄存器和 64 个 I/O 寄存器，其功能、特点及使用方法涉及到整个 AVR 单片机的全部功能和特性，因此相对复杂。学习者不可能一下就能全部掌握它们的应用，只有通过边学习、边实践，逐步深入，加深理解。本节仅给一个总体的介绍，各个不同寄存器具体的使用将在以后相关的章节中加以详细描述。

2.5.1 通用寄存器组

图 2-11 为 AVR 单片机中 32 个通用寄存器的结构图。在 AVR 指令集中，所有的通用寄存器操作指令均带有方向的，并能在单一时钟周期中访问所有的寄存器。

用户在使用汇编语言编写程序时，应注意如何正确使用 AVR 中 32 个通用的寄存器。因为这 32 个通用寄存器的功能还是有一定的区别。尤其是 R16～R31 后 16 个寄存器能实现的操作比 R0～R15 要多，如 SBCI、SUBI、CPI、ANDI、ORI 以及直接装入常数到寄存器 LDI，和乘法指令仅适用于寄存器组中后半部分的寄存器（R16～R31）。另外，R26～R31 还构成 3 个 16 位的地址指针寄存器 X、Y、Z，所以一般情况下不要作为它用。具体指令的介绍见第三章。

如图 2-11 所示，每个通用寄存器还被分配在 AVR 的数据存储器空间中，他们直接映射到数据空间的前 32 个地址，因此也可以使用访问 SRAM 的指令对这些寄存器进行访问，但此时在指令中应使用该寄存器在 SRAM 空间的映射地址。通常情况下，最好是使用专用的寄存器访问指令对通用寄存器组进行操作，因为这类寄存器专用操作指令不仅功能强大，而且执行周期也短。

寄存器名称	对应 SRAM 地址	附加功能
R0	$0000	
R1	$0001	
R2	$0002	
……		
R13	$000D	
R14	$000E	
R15	$000F	
……		
R26	$001A	X-寄存器低字节
R27	$001B	X-寄存器高字节
R28	$001C	Y-寄存器低字节
R29	$001D	Y-寄存器高字节
R30	$001E	Z-寄存器低字节
R31	$001F	Z-寄存器高字节

图 2-11 通用寄存器组结构图

AVR 寄存器组最后的 6 个寄存器 R26～R31 具有特殊的功能，这些寄存器每两个合并成一个 16 位的寄存器，作为对数据存储器空间（使用 X、Y、Z）以及程序存储器空间（仅使用 Z 寄存器）间接寻址的地址指针寄存器。这三个间接寄存器 X、Y、Z 由图 2-12 定义。在不同指令的寻址模式下，利用地址寄存器可实现地址指针的偏移、自动增量和减量（参考不同的指令）等不同形式的间址寻址操作。

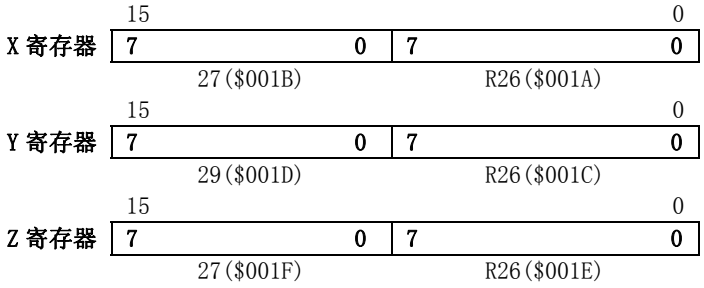

图 2-12 X、Y、Z 寄存器

2.5.2 I/O寄存器

表 2.1 列出了 ATmega16 单片机的 I/O 寄存器的地址空间分配、名称和功能。

表 2.1 ATmega16 I/O 寄存器空间分配表

十六进制地址	名 称	功 能
$00 ($0020)	TWBR	TWI 波特率寄存器
$01 ($0021)	TWSR	TWI 状态寄存器
$02 ($0022)	TWAR	TWI 从机地址寄存器
$03 ($0023)	TWDR	TWI 数据寄存器
$04 ($0024)	ADCL	ADC 数据寄存器低字节
$05 ($0025)	ADCH	ADC 数据寄存器高字节
$06 ($0026)	ADCSRA	ADC 控制和状态寄存器
$07 ($0027)	ADMUX	ADC 多路选择器
$08 ($0028)	ACSR	模拟比较控制和状态寄存器
$09 ($0029)	UBRRL	USART 波特率寄存器低 8 位
$0A ($002A)	UCSRB	USART 控制状态寄存器 B
$0B ($002B)	UCSRA	USART 控制状态寄存器 A

$0C ($002C)	UDR	USART I/O 数据寄存器
$0D ($002D)	SPCR	SPI 控制寄存器
$0E ($002E)	SPSR	SPI 状态寄存器
$0F ($002F)	SPDR	SPI I/O 数据寄存器
$10 ($0030)	PIND	D 口外部输入引脚
$11 ($0031)	DDRD	D 口数据方向寄存器
$12 ($0032)	PORTD	D 口数据寄存器
$13 ($0033)	PINC	C 口外部输入引脚
$14 ($0034)	DDRC	C 口数据方向寄存器
$15 ($0035)	PORTC	C 口数据寄存器
$16 ($0036)	PINB	B 口外部输入引脚
$17 ($0037)	DDRB	B 口数据方向寄存器
$18 ($0038)	PORTB	B 口数据寄存器
$19 ($0039)	PINA	A 口外部输入引脚
$1A ($003A)	DDRA	A 口数据方向寄存器
$1B ($003B)	PORTA	A 口数据寄存器
$1C ($003C)	EECR	EEPROM 控制寄存器
$1D ($003D)	EEDR	EEPROM 数据寄存器
$1E ($003E)	EEARL	EEPROM 地址寄存器低 8 位
$1F ($003F)	EEARH	EEPROM 地址寄存器高 8 位
$20 ($0040)	UBRRH UCSRC	USART 波特率寄存器高 4 位 USART 状态寄存器 C
$21 ($0041)	WDTCR	看门狗定时控制寄存器
$22 ($0042)	ASSR	异步模式状态寄存器
$23 ($0043)	OCR2	定时器/计数器 2 输出比较寄存器
$24 ($0044)	TCNT2	定时器/计数器 2（8 位）
$25 ($0045)	TCCR2	定时器/计数器 2 控制寄存器
$26 ($0046)	ICR1L	定时器/计数器 1 输入捕捉寄存器低 8 位
$27 ($0047)	ICR1H	定时器/计数器 1 输入捕捉寄存器高 8 位
$28 ($0048)	OCR1BL	定时器/计数器 1 输出比较寄存器 B 低 8 位
$29 ($0049)	OCR1BH	定时器/计数器 1 输出比较寄存器 B 高 8 位
$2A ($004A)	OCR1AL	定时器/计数器 1 输出比较寄存器 A 低 8 位
$2B ($004B)	OCR1AH	定时器/计数器 1 输出比较寄存器 A 高 8 位
$2C ($004C)	TCNT1L	定时器/计数器 1 寄存器低 8 位
$2D ($004D)	TCNT1H	定时器/计数器 1 寄存器高 8 位
$2E ($004E)	TCCR1B	定时器/计数器 1 控制寄存器 B
$2F ($004F)	TCCR1A	定时器/计数器 1 控制寄存器 A
$30 ($0050)	SFIOR	特殊功能 I/O 寄存器
$31 ($0051)	OSCCAL OCDR	内部 RC 振荡器校准值寄存器 在线调试寄存器
$32 ($0052)	TCNT0	定时器/计数器 0（8 位）
$33 ($0053)	TCCR0	定时器/计数器 0 控制寄存器
$34 ($0054)	MCUCSR	MCU 控制和状态寄存器
$35 ($0055)	MCUCR	MCU 控制寄存器
$36 ($0056)	TWCR	TWI 控制寄存器
$37 ($0057)	SPMCR	程序存储器写控制寄存器
$38 ($0058)	TIFR	定时器/计数器中断标志寄存器
$39 ($0059)	TIMSK	定时器/计数器中断屏蔽寄存器
$3A ($005A)	GIFR	通用中断标志寄存器
$3B ($005B)	GICR	通用中断控制寄存器
$3C ($005C)	OCR0	T/C0 计数器输出比较寄存器
$3D ($005D)	SPL	堆栈指针寄存器低 8 位
$3E ($005E)	SPH	堆栈指针寄存器高 8 位
$3F ($005F)	SREG	状态寄存器

AVR 系列单片机所有 I/O 口及外围接口的功能和配置均通过 I/O 寄存器的进行设置和使

用。CPU 访问 I/O 寄存器可以使用两种不同的方法，使用对 I/O 寄存器访问的 IN、OUT 专用指令，以及使用对 SRAM 访问的指令。

所有的 I/O 寄存器可以通过 IN(I/O 口输入)和 OUT(输出到 I/O 口)指令访问，这些指令是在 32 个通用寄存器与 I/O 寄存器空间之间传输交换数据，指令周期为 1 个时钟周期。此外，I/O 寄存器地址范围在$00-$1F 之间的寄存器（前 32 个）还可通过指令实现 bit 位操作和 bit 位判断跳转。SBI(I/O 寄存器中指定位置 1)和 CBI(I/O 寄存器中指定位清零)指令可直接对 I/O 寄存器中的每一位进行位操作。使用 SBIS(I/O 寄存器中指定位为 1 跳行)和 SBIC(I/O 寄存器中指定位为 0 跳行)指令能够对这些 I/O 寄存器中的每一位的值进行检验判断，实现跳过一条指令执行下一条指令的跳转。

在 I/O 寄存器专用指令 IN、OUT、SBI、CBI、SBIS 和 SBIC 中使用 I/O 寄存器地址$00～$3F。

当以 SRAM 方式寻址 I/O 寄存器时，必须将该其地址加上$0020，映射成在数据存储器空间的地址。本书中 I/O 寄存器地址均给出了两种地址表示：I/O 寄存器空间地址以及在数据存储器空间中的映射地址（在圆括号中）。

2.5.3 状态寄存器和堆栈指针寄存器

以下面我们首先介绍 2 个在 AVR 中起着非常重要作用的 I/O 寄存器，它们是状态寄存器 SREG，和堆栈指针寄存器 SP。

1．状态寄存器—SREG

状态寄存器 SREG 是一个 8 位标志寄存器，用来存放指令执行后的有关状态和结果的标志。SREG 中各位状态通常是在指令的执行过程中自动形成，但也可以由用户根据需要用专用指令加以改变。

状态标志位的作用很大，每一位都代表着不同含义。许多指令的运行将对寄存器中的某些位置 1 或清零，它反映了 CPU 运算、操作结果的状态。与 SREG 中的位操作有关的指令有置 1、清零、为 1 转移、为 0 转移等，共有 36 条指令与状态寄存器 SREG 相关联。由此可见它的重要性。

AVR 的状态寄存器 SREG 在 I/O 空间的地址为$3F($005F)，其各标志位的意义如下：

位	7	6	5	4	3	2	1	0	
$3F($005F)	I	T	H	S	V	N	Z	C	SREG
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	

● 位 7—I：全局中断使能位

该标志位为 AVR 中断总控制开关，当 I 位被置位（“1”）时，表示 CPU 可以响应中断请求，而当 I 位被清另（“0”），则所有的中断被禁止，CPU 不响应任何的中断请求。除了该标志位用于 AVR 中断的总控制，各个单独的中断触发控制还由其所在的中断屏蔽寄存器（GIMSK、TIMSK）中控制。如果全局中断触发寄存器被清另（“0”），则全局（所有的）中断被禁止，但单独的中断触发控制在 GIMSK 和 TIMSK 中的值保持不变。在中断发生后，I 位由硬件清除，并由 RETI(中断返回)指令置位，从而允许子序列的中断响应。

● 位 6—T：位复制存储

位复制指令 BLD 和 BST 使用 T 标志位作为源和目标。通用寄存器组中任何一个寄存器中的一位可以通过 BST 指令被复制到 T 中，而用 BLD 指令则可将 T 中的位值复制到通用寄存器组中的任何一个寄存器的一位中。

● 位 5—H：半进位标志位

半进位标志位 H 表示在一些运算操作过程中有无半进位(低四位向高四位进、借位)的产生，该标志对于 BCD 码的运算和处理非常有用。

● 位 4—S：符号标志位，S = N⊕V

S 位是负数标志位 N 和 2 的补码溢出标志位 V 两者异或值。在正常运算条件下（V=0，不溢出）S=N，即运算结果最高位作为符号是正确的。而当产生溢出时 V=1，此时 N 已不能正确指示运算结果的正负，但 S=N⊕V 还是正确的。对于单（或多）字节有符号数据说，执行减法或比较操作后，S 标志能正确指示参与相减或比较的两个数的大小。

● 位 3—V：2 补码溢出标志位

2 的补码溢出标志位 V，支持 2 的补码运算，为模 2 补码加、减运算溢出标志。溢出表示运算结果超过了正数（或负数）所能表示的范围。加法溢出表现为正+正=负，或负+负=正；减法溢出表现为正-负=负，或负-正=正。溢出时，运算结果最高位（N）取反才是真正的结果符号。

● 位 2—N：负数标志位

负数标志位直接取自运算结果的最高位，N=1 时表示运算结果为负，否则为正。但发生溢出时不能表示真实的结果（见上面对溢出标志位的说明）。

● 位 1—Z：零值标志位

零值标志位表明在 CPU 运算和逻辑操作之后，其结果是否为零，当 Z=1 表示结果为零。

● 位 0—C：进/借位标志

进位标志位表明在 CPU 的运算和逻辑操作过程中有无发生进/借位。

以上这些标志位非常重要，对运算结果的判断处理，要以相应的标志位为依据。标志位也是分支、循环控制的依据。采用汇编编写程序时，要注意指令对标志位的影响，以及正确的使用判断指令。

2．堆栈指针寄存器—SP

堆栈是数据结构中所使用的专用名词，它是由一块连续的 SRAM 空间和一个堆栈指针寄存器组成，主要应用于快速便捷的保存临时数据、局部变量和中断调用或子程序调用的返回地址。堆栈在系统程序的设计和运行中起者非常重要的作用，只要程序中使用了中断和子程序调用，就必须正确的设置堆栈指针寄存器 SP，在 SRAM 空间建立堆栈区。

堆栈是一种特殊的线性数据结构，数据的进出在堆栈的顶部进行，并遵循后进先出（LIFO）的原则。堆栈指针实际上就是堆栈顶部的地址，它随着堆栈中数据的进出而变化。堆栈指针寄存器 SP 中保存着堆栈指针，即堆栈顶部的地址。

处在 I/O 地址空间的$3E($005E)和$3D($005D)的两个 8 位寄存器构成了 AVR 单片机的 16 位堆栈指针寄存器 SP。AVR 单片机复位后堆栈寄存器的初始值为 SPH=$00、SPL=$00，因此建议用户程序必须首先对堆栈指针寄存器 SP 进行初始化设置。

位	15	14	13	12	11	10	9	8	
$3E($005E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
$3D($005D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
位	7	6	5	4	3	2	1	0	
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	
初始化值	0	0	0	0	0	0	0	0	

AVR 的堆栈区是建立在 SRAM 空间的，16 位的 SP 寄存器可以寻址的空间为 64K。但在实际应用中，还必须考虑所使用 AVR 芯片 SRAM 空间的实际情况和所配备的 SRAM 容量的大小。

首先，堆栈区应该避开寄存器区域所对应的 SRAM 空间，防止堆栈操作时改变了寄存器的设置。由于 AVR 的堆栈是向下增长的，即新数据进入堆栈时栈顶指针的数据将减小（注意：这里与 51 不同，51 的堆栈是向上增长的，即进栈操作时栈顶指针的数据将增加），所以尽管原则上堆栈可以在 SRAM 的任何区域中，但通常初始化时将 SP 的指针设在 SRAM 最高处。

对于具体的 ATmega16 芯片，堆栈指针必须指向高于$0060 的 SRAM 地址空间，因为低于$0060 的区域为寄存器空间。ATmega16 片内集成有 1K 的 SRAM，不支持外部扩展 SRAM，所以堆栈指针寄存器 SP 的初始值应设在 SRAM 的最高端：$045F 处（参考图 2-10）。

AVR 的堆栈有自动硬件进栈（执行调用指令、响应中断），自动硬件出栈（执行调用返回指令 RET、执行中断返回指令 RETI）和人工进出栈（进栈 PUSH、出栈 POP）等指令。AVR 单片机堆栈采用 SP-1 或 SP-2 的进栈操作，具体见第三章中有关指令介绍部分。

根据上面所讲述，AVR 的 SP 堆栈指针寄存器指示了在数据 SRAM 中堆栈区域的栈顶地址，一些临时数据、局部变量，以及子程序返回地址和中断返回地址将被放置在堆栈区域中。在数据 SRAM 中，该堆栈空间的顶部地址必须在系统程序初始化时由初始化程序定义和设置。

当执行 PUSH 指令，一个字节的数据被压入堆栈，堆栈指针（SP 中的数据）将自动减 1；当执行子程序调用指令 CALL 或 CPU 响应中断时，硬件会自动把返回地址（16 位数据）压入堆栈中，同时将堆栈指针自动减 2。反之，当执行 POP 指令，从堆栈顶部弹出一个字节的数据，堆栈指针将自动加 1；当执行从子程序 RET 返回或从中断 RETI 返回指令时，返回地址将从堆栈顶部弹出，堆栈指针自动加 2

2.6 ATmega16 单片机的工作状态

对采用单片机所构成的嵌入式电子系统来讲，单片机芯片构成了嵌入式系统的心脏，整个系统的正常工作是由单片机来控制、指挥和协调的。可以这样简单的理解：将一块单片机，加上必要的外围电路（如发光二极管、显示器、继电器、按键键盘，等等），以及根据一定功能要求和硬件电路编写的系统运行程序，三者有机的结合，就能组成各种类型，各种功能，千姿百态的电子系统和产品。如果我们把电子产品比喻为人，那么单片机就是人的大脑和心脏，外围电路就如同人的五官和四肢，电路板上的路线如同人的神经系统，而系统运行程序就代表人的知识、思维、判断和反应。外围电路通过各种不同的传感器，测量和获取到现实世界的状态（如温度、转速、压力），这些状态值经过线路传送到单片机中，由单片机中的程序进行计算和判断，然后再发出控制信号到外围电路，外围的控制机构产生正确的动作——一个新的电子产品诞生了。

AVR 单片机的工作状态通常包括：复位状态、正常程序执行工作状态、休眠节电工作状态、程序运行代码下载的编程，以及熔丝位的配置。用户必须非常熟悉和了解 AVR 的这些工作状态和它们之间的转换关系。

2.6.1 AVR单片机最小系统

一个单片嵌入式系统的核心，其实就是一个单片机最小系统。它仅仅由一片单片机芯片、两个电阻、一个石英晶体和两个电容构成，见图 2-13。

图 2-13 虚线框里几个器件所构成的最小系统，就是一颗单片嵌入式系统完整的心脏和大脑，可以工作了。当然，没有相应的外围电路，我们还是不能直观的了解它的工作情况的。因此图中还有一个简单的外围电路：一个发光二极管和一个限流保护电阻。我们可以编写一个简单的程序，其功能让发光二极管每间隔 1 秒闪烁一次，循环往复。把程序的运行代码下

载到 ATmega16 的程序存储器中，一个秒节拍输出显示装置就诞生了。只要一接通电源，ATmega16 就以每秒 4 兆的工作频率运行，驱动发光二极管每间隔 1 秒闪烁一次（具体实现见第五章）。

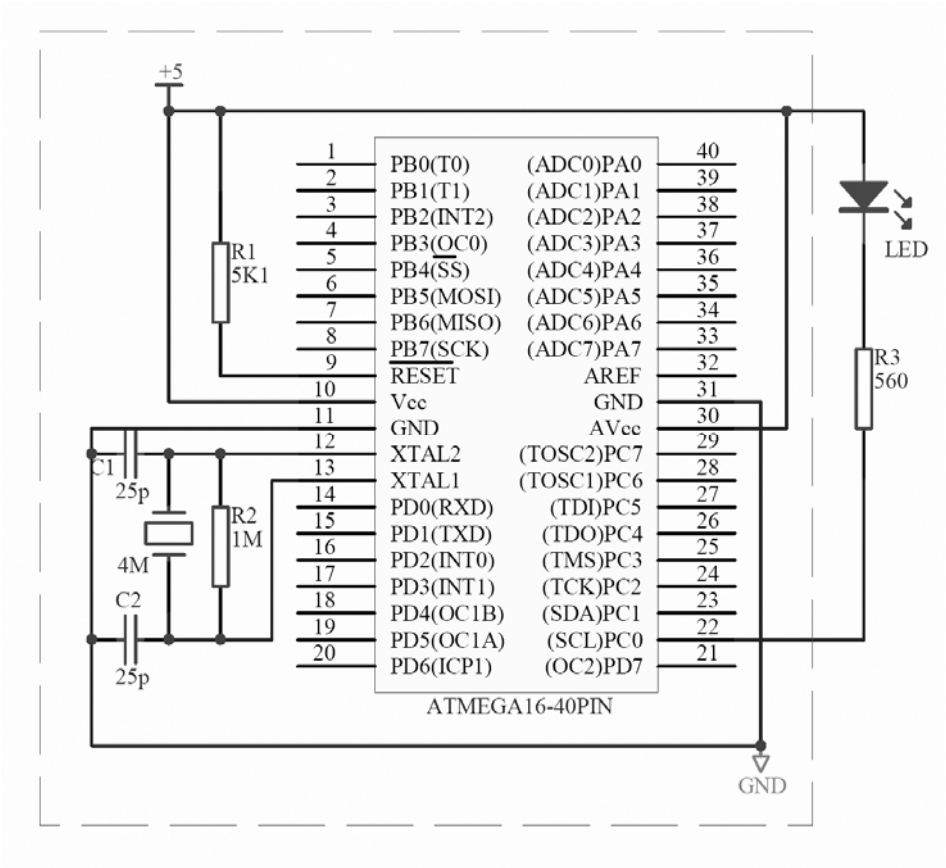

图 2-13 ATmega16 最小系统电路图

在图 2-13 中，采用了在 ATmega16 引脚 XTAL1 和 XTAL2 上外接由石英晶体和电容组成的谐振回路，并配合片内的 OSC（Oscillator）振荡电路构成的振荡源作为系统时钟源的。更简单的电路是直接使用片内的 4M 的 RC 振荡源，这样就可以将 C1、C2、R2 和 4M 晶体省掉，引脚 XTAL1 和 XTAL2 悬空，当然此时系统时钟频率精准度不如采用外部晶体的方式，而且也易受到温度变化的影响。对于图中电阻 R1、R3 的作用和阻值的选取将在第五章中详细分析。

2.6.2 AVR的复位源和复位方式

复位是单片机芯片本身的硬件初始化操作，例如，单片机在上电开机时都需要复位，以便 CPU 以及其它内部功能部件都处于一个确定的初始状态，并从这个初始状态开始工作。

AVR 单片机的复位操作，其主要功能是把程序计数器 PC 初始化为$0000（指非 BOOT LOAD 方式启动），使单片机从$0000 单元开始执行程序。同时决大部分的寄存器（通用寄存器和 I/O 寄存器）也被复位操作清零，有关各个寄存器的复位初始化值请注意书中对各寄存器的详细说明。

除了系统上电的正常复位初始化之外，当系统程序在运行中出现错误或受到电源的干扰，出现错误时，也可通过外部引脚 RESET 进行人工复位，或由芯片内部看门狗定时器 WDT 自动复位，或由芯片内部掉电检测 BOD 来使系统自动进入复位初始化操作。

图 2-14 的电路给出了 ATmega16 系统的复位逻辑，表 2.2 给出了复位电特性的参考值。

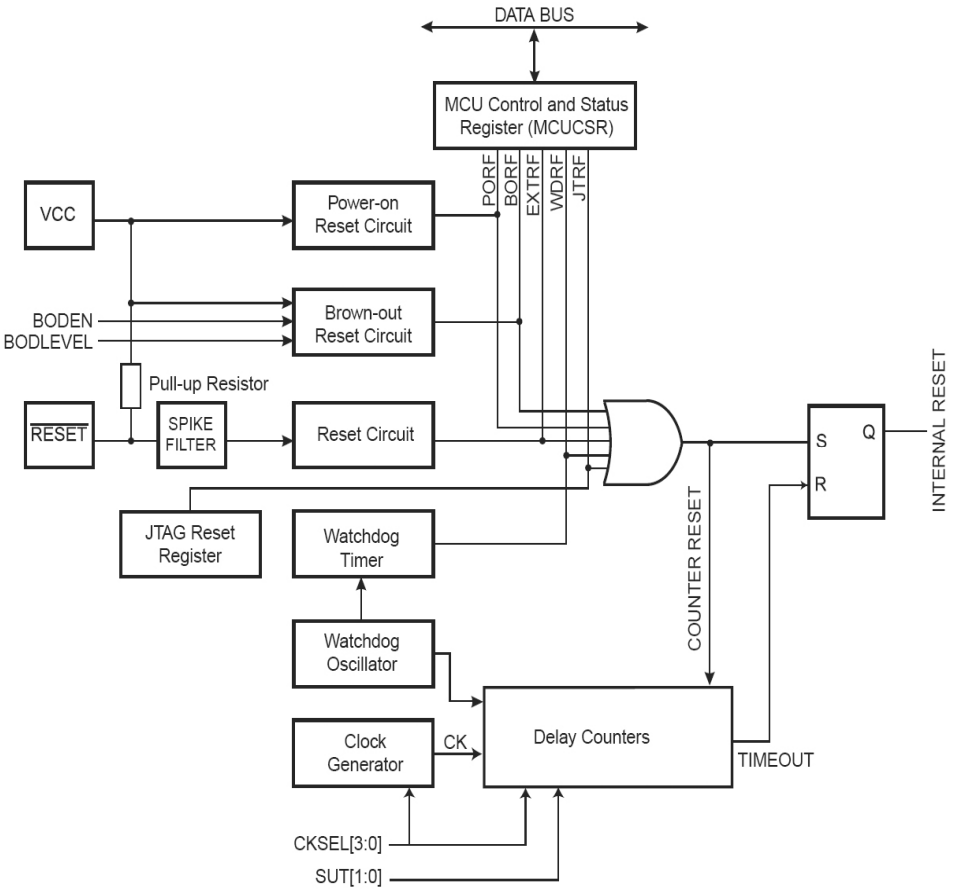

图 2-14 ATmega16 系统复位逻辑图

ATmega16 单片机共有 5 个复位源，它们是：

- 上电复位。当系统电源电压低于上电复位门限 Vpot 时，MCU 复位。
- 外部复位。当外部引脚 RESET 为低电平，且低电平持续时间大于 1.5us 时，MCU 复位。
- 掉电检测（BOD）复位。BOD 使能时，且电源电压低于掉电检测复位门限（4.0v 或 2.7v）时，MCU 复位。
- 看门狗复位。WDT 使能时，并且 WDT 超时溢出时，MCU 复位。
- JTAG AVR 复位。当使用 JTAG 接口时，可由 JTAG 口控制 MCU 复位。

表 2.2　系统复位电参数

符　号	参　数	条　件	最小值	典型值	最大值	单位
V_{POT}	上电复位门限电压（电源电压上升时）			1.4	2.3	V
	上电复位门限电压（电源电压下降时）			1.3	2.3	V
V_{RST}	RESET 门限电压		0.1Vcc		0.9Vcc	V
t_{RST}	RESET 最小复位脉冲宽度				1.5	μs
V_{BOT}	BOD 复位门限电压	BODLEVEL=1	2.5	2.7	3.2	V
		BODLEVEL=0	3.6	4.0	4.5	
t_{BOD}	BOD 检测的低电压最小宽度	BODLEVEL=1		2		μs
		BODLEVEL=0		2		
V_{HYST}	BOD 检测迟滞电压			50		mV

当任何一个复位信号产生时，AVR 将进行复位操作。复位操作过程并不需要时钟源处于运行工作状态（AVR 采用异步复位方式，提高了可靠性）。在 MCU 复位过程中，所有的 I/O 寄存器被设为初始值，程序计数器 PC 置 0。当系统电压高于上电复位门限 Vpot（或 BOD 复位门限电压）时，复位信号撤消，硬件系统 Delay Counters 将启动一个可设置的计数延时过程（延时时间为 Ttout，由一组溶丝位 SUT、CKSEL 确定）。经过一定的延时后，AVR 才进行系统内部真正的复位启动（Internal Reset）。采用这种形式的复位启动过程，能够保证电源电压在达到稳定后单片机才进入正常的指令操作。

AVR 复位启动后，由于程序计数器 PC 置为$0000，因此 CPU 取出的第一条指令就是在 Flash 空间的$0000 处，即复位后系统程序从地址$0000 处开始执行（指非 BOOT LOAD 方式启动）。通常在$0000 地址中放置的指令为一条相对转移指令 RJMP 或 JMP 指令，跳到主程序的开始。这样，系统复位启动后，首先执行$0000 处的跳转指令，然后转到执行主程序的指令。

由此可见，AVR 的复位过程考虑的非常周到，也是非常可靠的。AVR 单片机采用 5 个复位源，异步复位操作，以及内部可设置的延时启动，大大提高了芯片的抗干扰能力和整个系统的可靠性，这在工业控制中非常重要，同时也是 AVR 单片机的优点之一。与此同时，AVR 内部的 MCU 控制和状态寄存器 MCUCSR 还将引起复位的复位源进行了记录，用户程序启动后，可以读取 MCUCSR 中的标记，查看复位是由于何种情况造成的，是正常复位还是异常复位，从而根据实际情况执行不同的程序，实现不同的处理。这对于实现高可靠的系统控制，掉电保护处理、故障处理等应用非常有用，具体请参考 AVR 的器件手册说明。

1．上电复位

AVR 内部含有上电复位 POR(Power_on Reset)电路。POR 确保了只有当 Vcc 超过一个安全电平时，器件才开始工作，如图 2-15、2-16 所示。

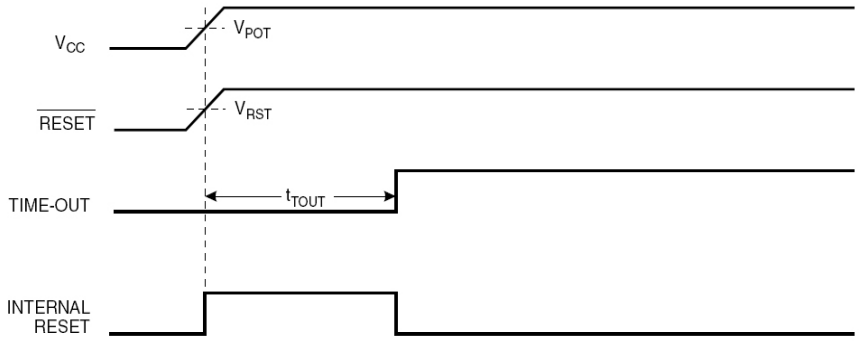

图 2-15 MCU 上电复位启动，RESET 连到 Vcc

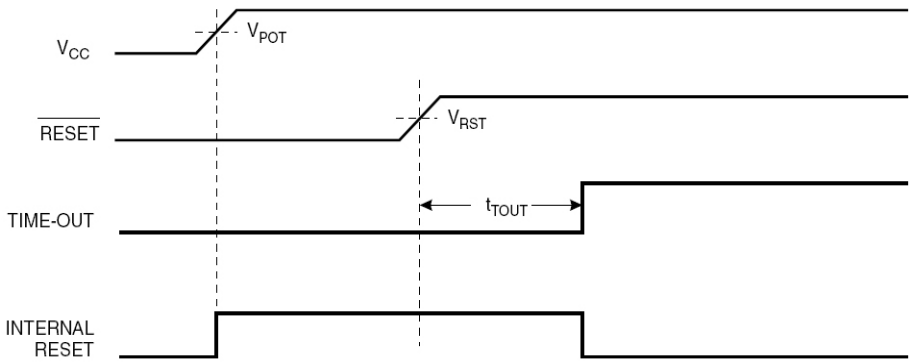

图 2-16 MCU 上电复位启动，RESET 由外部控制

无论何时，只要 Vcc 低于检测电平 Vpot 时，器件进入复位状态。一旦当 Vcc 超过门限电压 Vpot，而 RESET 电压也达到 Vrst 时，将启动芯片内部的一个可设置的延时计数器（图 2-14 中的 Delay Counters）。在延时计数器溢出之前，器件一直保持复位状态（Internal Reset 保持高电平）。经过 Ttout 时间后，延时计数器溢出，将内部复位信号（Internal Reset）拉低，CPU 才开始正式工作。

2．外部复位

外部复位是由外加在 RESET 引脚上的低电平将产生的。当 RESET 引脚被拉低于 Vrst 的时间大于 1.5μs 时既触发复位过程，见图 2-17。当 RESET 引脚电平高于 Vrst 后，将启动内部可设置的延时计数器（图 2-14 中的 Delay Counters）。在延时计数器溢出之前，器件一直保持复位状态（Internal Reset 保持高电平）。经过 Ttout 时间后，延时计数器溢出，将内部复位信号（Internal Reset）拉低，CPU 才开始正式工作。

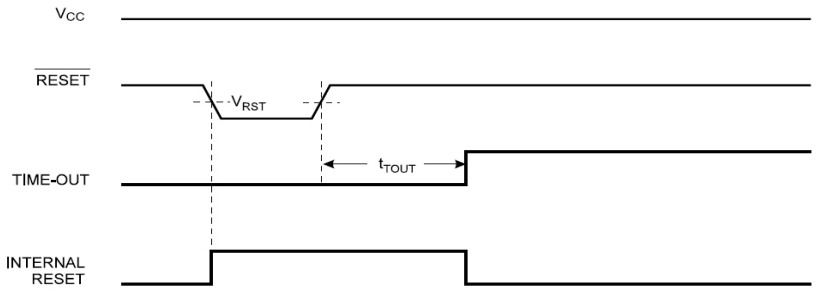

图 2-17 外部 RESET 复位

3．掉电检测（BOD）复位

ATmega16 有一个片内的 BOD（ Brown-out Detection）电源检测电路，用于在系统运行时对系统电压 V_{CC} 的检测，并同一个固定的阈值电压相比较。BOD 检测阈值电压可以通过 BODLEVEL 熔丝位设定为 2.7V 或 4.0V。BOD 检测阈值电压有迟滞效应，以避免系统电源的尖峰毛刺误触发 BOD 检测器。阈值电平的迟滞效应可以理解为：上阈值电压 $V_{BOT+} = V_{BOT} + V_{HYST}/2$，下阈值电压 $V_{BOT-} = V_{BOT} - V_{HYST}/2$。

BOD 检测电路可以通过编程 BODEN 熔丝位来置成有效或者无效。当 BOD 被置成有效，并且 V_{CC} 电压跌到下阈值电压 V_{BOT-}（如图 2-18 所示）以下时既触发复位过程，CPU 进入复位状态。当 V_{CC} 回升，而且超过上阈值电压 V_{BOT+} 后，再经过设定的启动延时时间，CPU 重新启动运行。注意只有当 V_{CC} 电压低于阈值电压并且持续 t_{BOD} 后，BOD 电路才启动延时计数器计数。

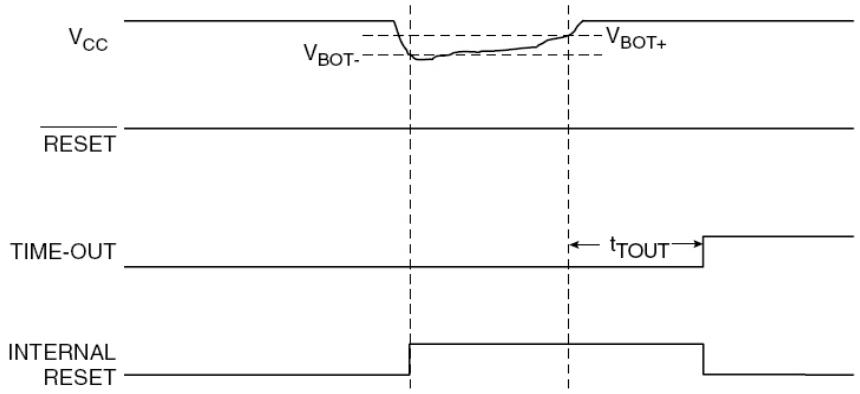

图 2-18 掉电检测 BOD 复位

4．看门狗复位

ATmega16 片内还集成一个独立的看门狗定时器 WDT。该 WDT 由片内独立的 1M 振荡器提

供时钟信号，并且可用专用的熔丝位或由用户通过指令控制 WDT 的启动和关闭，以及设置和清零计数值。当 WDT 启动计数后，一旦发生计数溢出，它将触发产生一个时钟周期宽度的复位脉冲。脉冲的上升沿将使器件进入复位状态，脉冲的下降沿启动延时计数器计数，经过设定的启动延时时间，CPU 重新开始运行（图 2-19）。使用 WDT 功能，可以防止系统受到干扰而引起的程序运行紊乱和跑飞，提高了系统的可靠性。

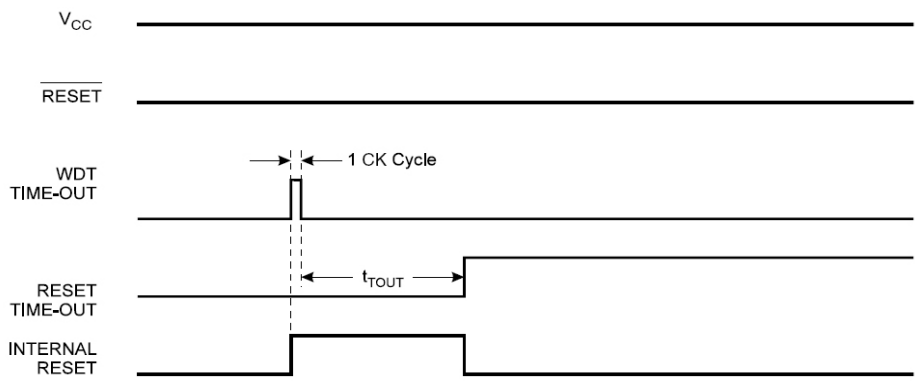

图 2-19 WDT 溢出复位

2.6.3 对AVR的编程下载

现在，单片机系统程序的编写、开发和调试都是借助于通用计算机 PC 完成的。用户首先在 PC 机上通过使用专用单片机开发软件平台，编写由汇编语言或高级语言构成的系统程序(源程序)，再由编译系统将源程序编译成单片机能够识别和执行的运行代码(目标代码)。运行代码的本身是一组二进制的数据，在 PC 中对于纯二进制码的数据文件一般是采用 BIN 格式保存的，以“bin”作为文件的扩展名。但是实际使用中，通常使用的是一种带定位格式的二进制文件：HEX 格式的文件，一般以“hex”作为文件的扩展名（HEX 文件格式说明见附录？）。

对单片机的编程操作，通常也称为程序下载，是指以特殊手段和软硬件工具，对单片机进行特殊的操作，以实现下面的 3 种功能：

- 将在 PC 机上生成的该单片机系统程序的运行代码写入单片机的程序存储器中。
- 用于对片内的 Flash、EEPROM 进行擦除、数据的写入（包括运行代码）、和数据的读出。
- 实现对 AVR 配置溶丝位的设置；芯片型号的读取；加密位的锁定等。

AVR 单片机支持多种形式的编程下载方式：

- 高压并行编程方式。

对于外围引脚数大于 20 的 AVR 芯片，一般都支持这种高压并行编程方式。这种编程方式也是最传统的单片机的程序下载方式，其优点是编程速度快。但使用这种编程方式需要占用芯片众多的引脚和 12V 的电压，所以必须采用专用的编程器单独对芯片操作。这样 AVR 芯片必须从 PCB 板上取下来，不可以实现芯片在线（板）的编程操作，因此这种方式不适合系统调试过程以及产品的批量生产需要。

- 串行编程方式（ISP）。

串行编程方式是通过 AVR 芯片本身的 SPI 或 JTAG 串行口实现的，由于编程时只需要占用比较少的外围引脚，所以可以实现芯片的在线编程（In System Programmable），不需要将芯片从 PCB 板上取下来，所以串行编程方式也是最方便和最常用的编程方式。

串行编程方式还细分成 SPI、JTAG 方式，前者表示通过芯片的 SPI 串口实现对 AVR 芯片

的编程操作，后者则是通过 JTAG 串口来实现的。AVR 的许多芯片都同时集成有 SPI 和 JTAG 两种串口，因此可以同时支持 SPI 和 JTAG 的编程。使用 JTAG 方式编程的优点是，通过 JTAG 口还可以实现系统的在片实时仿真调试（On Chip Debug），缺点是需要占用 AVR 的 4 个 I/O 引脚。而采用 SPI 方式编程，只需要一跟简单的编程电缆，同时可以方便的实现 I/O 口的共用，因此是最常使用的方式。其不足之处是不能实现系统的在片实时仿真调试。

- 其它编程方式

一些型号的 AVR 还支持串行高压编程方式和 IAP(In Application Programmable)在运行编程方式。串行高压编程是替代并行高压编程的一种方式，主要针对 8 个引脚的 Tiny 系列的 AVR 使用。IAP 在运行编程方式则是采用了 ATMEL 称为自引导加载（Boot Load）技术实现的,往往在一些需要进行远程修改更新系统程序,或动态改变系统程序的应用中才采用。

ATmega16 片内集成了 16K 字节的支持系统在线可编程（ISP）和在应用可编程（IAP）的 Flash 程序存储器，以及 512 个字节的 EEPROM 数据存储器。另外在它的内部，还有一些专用的可编程单元－熔丝位，用于加密锁定和对芯片的配置等。对 ATmega16 编程下载操作，就是在片外对上述的存储器和熔丝单元进行读/写（烧入）以及擦除的操作。

由于 ATmega16 片内含有 SPI 和 JTAG 口，所以对 ATmega16 能使用 3 种编程的方式：高压并行编程、串行 SPI 编程、串行 JTAG 编程。在本书中将主要介绍和采用串行 SPI 编程方式。

2.6.4 ATmega16 的熔丝位

在 AVR 内部有多组与器件配置和运行环境相关熔丝位，这些熔丝位非常重要，用户可以通过设定和配置熔丝位，使 AVR 具备不同的特性，以更加适合实际的应用。下面只介绍在开始学习和使用 ATmega16 时，需要特别注意和关心的重要熔丝位的使用配置，全部熔丝位的定义和使用配置见附录 A。

ATmega16 单片机在售出时，片内的 Flash 存储器和 EEPROM 存储器阵列是处在擦除的状态（即内容 = $FF），且可被编程。同时其器件配置熔丝位的缺省值为使用内部 1M 的 RC 振荡源作为系统时钟！

1．存储器加密锁定位

ATmage16 有 2 个加密锁定位 LB1 和 LB2，用于设定对片内存储器的加密方式，用户可在编程方式下，对 LB1、LB2 不编程(1)，或编程(0)，从而获得对片内存储器不同的加密保护方式，见表 2.3。

表 2.3 加密锁定位保护方式

加密锁定位			保护方式
模式	LB2	LB1	
1	1	1	无锁定方式（无加密），出厂状态
2	1	0	禁止对 Flash、EEPROM、溶丝位的再编程
3	0	0	禁止对 Flash、EEPROM、加密锁定位、溶丝位的再编程和校验，禁止对加密锁定位、溶丝位的再编程

需要进一步说明是：

- 在 AVR 的器件手册中，使用已编程（Programmed）和未编程（Unprogrammed）定义加密位和熔丝位的状态。“Unprogrammed”表示熔丝状态为“1”（禁止），“Programmed”表示熔丝状态为“0”（允许），即

  1：未编程

0：编程

- AVR 的加密位和熔丝位可多次编程，不是 OPT 熔丝。
- AVR 芯片加密锁定后（LB2/LB1 = 1/0，0/0），在外部不能通过任何方式读取芯片内部 Flash 和 EEPROM 中的数据，但熔丝位的状态仍然可以读取，不能修改配置。
- 需要重新下载程序时，或芯片被加密锁定后，或发现熔丝位配置不对，都必须先在编程状态使用芯片擦除命令，清除芯片内部存储器中的数据，同时解除加密锁定。然后重新下载运行代码和数据，修改和配置相关的熔丝位，最后再次配置芯片的加密锁定位。
- 编程状态的芯片擦除命令是将 Flash 和 EEPROM 中的数据清除，并同时将两位锁定位状态配置成无锁定状态（LB2/LB1 = 1/1）。但芯片擦除命令并不改变其它熔丝位的状态。
- 下载编程的正确的操作程序是：在芯片无锁定状态下，下载运行代码和数据，配置相关的熔丝位，最后配置芯片的加密锁定位。

2．系统时钟类型的配置

ATmega16 可以使用多种类型的系统时钟源，最常用的为 2 种：使用内部的 RC 振荡源（1M/2M/4M/8M）和外接晶体（晶体可在 0-16MHz 之间选择）配合内部振荡放大器构成的振荡源。具体系统时钟类型的配置由 CKOPT 和 CKSEL[3-0]共 5 个熔丝设定，表 2.4、表 2.5 给出了具体的配置值。用户在使用中，首先要根据实际使用情况进行正确的设置，而且千万注意不要对这些熔丝位误操作！

AVR 提供了用户更多的灵活选择系统时钟的可能性，以满足和适合实际产品的需要。

ATmega16 在片内集成有内部可校准的 RC 振荡器，能提供固定的 1/2/4/8MHZ 的系统时钟，这些频率是在 5V，25℃时的标称数值。CKOPT 和 CKSEL 熔丝按表 2.4 编程配置时，可以选择 4 种内部 RC 振荡源之一作为系统时钟使用，此时将不需要外部的元件。

表 2.4 系统时钟类型为使用内部 RC 振荡源

CKOPT	CKSEL[3..0]	工作频率范围 (MHz)
1	0001	1.0 (出厂设定)
1	0010	2.0
1	0011	4.0
1	0100	8.0

当产品对系统时钟的精度要求比较高，或需要使用一些特殊频率的系统时钟场合时，如使用了 USART 通信接口，系统时钟频率需要使用 4.6080M/7.3728M/11.0592M 时，就要采用第 2 种方式来组成系统时钟源：使用外接晶体（晶体可在 0-16MHz 之间选择）配合内部振荡放大器构成振荡源，具体连接电路为图 2-6(a)。此时需要将 CKOPT 和 CKSEL 熔丝按表 2.5 编程配置。

表 2.5 使用外部晶体与片内振荡放大器构成的振荡源

熔丝位		工作频率范围 (MHz)	C1、C2 容量(pF) (仅适用石英晶振)
CKOPT	CKSEL[3..0]		
1	101x	0.4-0.9	仅适合陶瓷振荡器
1	110x	0.9-3.0	12-22 (应与使用晶体配合)
1	111x	3.0-8.0	12-22 (应与使用晶体配合)
0	101x,110x,111x	≥1.0	12-22 (应与使用晶体配合)

在表 2.5 中，当 CKOPT = 0 时，振荡器的输出振幅较大，容易起振，适合在干扰大的场合以及使用的晶体超过 8M 时的情况下使用。而 CKOPT = 1 时，振荡器的输出振幅较小，这样可以减小对电源的消耗，对外的电磁幅射也较小。

2.6.5 AVR单片机的工作状态

当 AVR 芯片的 Vcc 与系统电源接通后，根据 RESET 引脚的电平值的不同，单片机将进入不同的状态：复位状态、常规工作状态、编程状态。

1. RESET 引脚电平为高

通常情况下，RESET 引脚通过一个上拉电阻接系统电源，为高电平“1”，见图 2-13。在此条件下，一旦接通电源，AVR 将进入上电复位状态。经过短暂的内部的复位操作后，芯片便进入了常规的工作状态（BOD 和 WDT 引起的复位类同）。

AVR 处在常规工作状态时，有两种工作方式：正常程序执行工作方式和休眠节电工作方式。

- 正常程序执行工作方式

正常程序执行工作方式是单片机的基本工作方式。由于硬件的复位操作将程序计数器置为零（PC=$0000），因此程序的执行总是从 Flash 地址的$0000 开始的（指非 BOOT LOAD 方式启动）。

对于 ATmega16 来讲，Flash 地址的$0002 到$0028 是中断向量区（详见第六章），所以真正实际要开始运行的程序代码一般放在从$002A 以后的程序地址空间中。标准的做法是在 Flash 的$0000 单元中放置一条转移指令 JMP 或 RJMP，使得 CPU 在复位重新启动后，首先执行该转移指令，跳过中断向量区，转到执行实际程序的开始处。典型的程序结构如下：

Flash 空间地址		指 令 字	说　明
$0000		jmp RESET	;复位中断向量
…		…	
…		…	;向量区
…		…	
$002A	RESET:	ldi r16,high(RAMEND)	;主程序开始
……		……	;……

- 休眠节电工作方式

休眠节电工作方式是使单片机处于低功耗节电的一种工作方式。当单片机需要处于长时间等待外部触发信号，待有外部触发后才做相应的处理，或每隔一段时间才需要做处理的情况时，可以使用休眠节电工作方式，以减小对电源的消耗。CPU 处于等待的时候（待机状态）可进入休眠节电工作方式，此时 CPU 暂停工作，不执行任何指令。在休眠节电工作方式中，只有部分单片机的电路处于工作状态，而其它的电路停止工作，这样就可节省单片机的对电源消耗，形成系统的省电待机状态。一旦有外部的触发信号，或等待时间到，CPU 从休眠状态中被唤醒，重新进入正常程序执行工作方式。

ATmega16 有 6 种不同的休眠模式，每一种模式对应的电源消耗也不同，被唤醒的方式也有多种类型，用户可以根据实际的需要进行选择。

休眠节电工作方式对使用电池供电的系统非常重要，AVR 提供了更多的休眠模式，更加符合和适应实际的需要。如 ATmega16 处在掉电休眠模式状态，其本身的耗电量小于 1μA。

2. RESET 引脚电平为低

AVR 通电后，如果 RESET 脚的电平被外部拉为低电平“0”，则芯片将进入和处在复位状态，见图 2-16 和图 2-17。通常情况下，该复位状态一直延续到 RESET 脚的低电平被撤消。一旦 RESET 恢复了高电平，AVR 将重新启动，进入常规工作状态。利用该特点可以实现对 AVR

系统的人工复位或外部强制复位操作。

尤其需要说明的是，一旦 RESET 脚的电平被外部拉低，当满足某些特殊条件后，芯片将进入编程状态。例如，如果芯片带有 SPI 接口，支持 SPI 串行编程，则通过以下方式将使芯片进入 SPI 编程状态：

- 外部将 SPI 口的 SCK 引脚拉低，然后外部在 RESET 引脚上施加一个至少为 2 个系统周期以上低电平脉冲；
- 延时等待 20ms 后，由外部通过 AVR 的 SPI 口向芯片下发允许 SPI 编程的指令；

在AVR的器件手册的存储器编程（Memory Programming）一章中串行下载（Serial Downloading）一节里，详细介绍了利用AVR的SPI接口实现ISP编程的硬件连接、编程方式状态的进入过程和串行编程的命令等。

一旦芯片进入编程状态，就可以通过 SPI 口将运行代码写入 AVR 的程序存储器，对片内的 Flash、EEPROM 进行擦除、数据的写入（包括运行代码）、和数据的读出，以及实现对 AVR 配置溶丝位的设置、芯片型号的读取和加密位的锁定等操作了。

2.6.6 支持ISP编程的最小系统设计

在本小节中给出一个最基本的、典型的支持 ISP 编程的 AVR 最小系统硬件图。尽管 ATmega16 的 SPI 和 JTAG 口都可以实现 ISP 在线编程，但采用 SPI 口实现 ISP 在线编程是最常用的方式，因为这样不会造成 AVR 的 I/O 口浪费。

作为图 2-13 以 ATmega16 芯片构成的 AVR 最小系统中，并没有考虑如何实现对 AVR 的编程。如果完全按图 2-13 完成硬件系统后，要对 AVR 编程时，就必须将芯片从 PCB 板上取下，放到专用的编程设备上将才能系统的执行代码下载到芯片中，然后再将芯片插回到 PCB 上，对于系统调试和生产都非常不方便。

图 2-20 在图 2-13 的基础上增加了一个 ISP 编程下载口，该口的 2、3、4、5 脚同芯片 SPI 接口的 MOSI（PB5）、MISO（PB6）、SCK（PB7）和 RESET 引脚连接。当需要改动 AVR 的熔丝位配置，或将编译好的运行代码烧入的 AVR 的 FlashROM 中时，就不需要将芯片从 PCB 板上取下了。只要将一根简单的编程线插在该编程下载口上，利用 PC 机就可以方便的实现上面的操作了。

如 2.6.5 中所介绍，当 PC 机对 AVR 编程时，需要先将 SCK 和 RESET 引脚拉低，使 AVR 芯片进入 SPI 编程状态，然后通过 SPI 口进行下载操作。所以，在设计 AVR 系统硬件时，如考虑使用 SPI 口实现 ISP 的功能，图中的 R1 电阻不可省略。此时 R1 起到了隔离作用，正是有了 R1，才能使用户在外部能够对 RESET 脚施加低电平（0 伏）。当编程下载完成后，外部一旦释放掉 RESET，该引脚通过 R1 又被拉成高电平，AVR 就直接进入了正常运行工作状态。R1 的阻值在 5k-10k 之间，太大和太小都不合适。如果系统不需要支持 ISP 功能话，则可将 R1 省掉，把 RESET 引脚与 Vcc 连接。

AVR 的 PB5、PB6、PB7 与编程下载口连接，在编程状态时这 3 个引脚用于下载操作。编程完成拔掉下载线，芯片进入正常工作后，PB5、PB6、PB7 仍可作为普通的 I/O 口或 AVR 的 SPI 口使用，受 AVR 的控制，这是使用 SPI 口实现 ISP 功能的优点之一。需要注意的是，如果系统中使用了这 3 个引脚，PCB 板上这 3 个引脚已经与外围器件连接在一起的情况下，就需要对外围的连接情况进行分析。如果外围连接在上电情况时表现为强上拉或强下拉（最极端情况为接高电平或 GND），那么为了保证 AVR 的 SPI 功能的正常工作，应该如图 2-20 中所示，串入 3 个隔离电阻，阻值在 2K 左右。

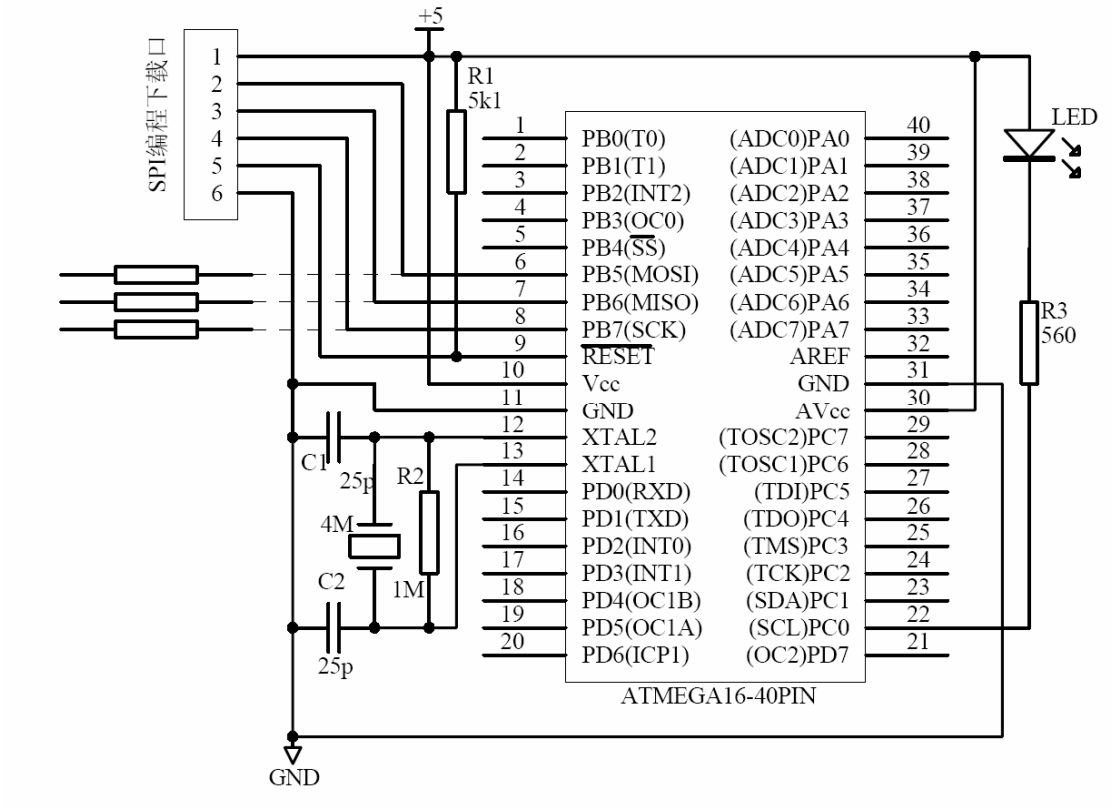

图 2-20 支持 ISP 编程的最小系统设计

对于不同的 AVR 芯片，使用 SPI 方式进行下载编程的硬件连接口，操作命令和时序请参考与该器件相关器件手册中详细说明，会有所变化（如 ATmega128），但基本方式相同。同使用其它类型的单片机（如 8051）一样，可以采用专用的写入设备对进行编程下载，但 AVR 提供了更方便的在线（ISP）串行下载的方法，用户只要制作一个简单的带隔离电路的下载线，就可直接使用个人电脑 PC 机的打印机口实现 AVR 的 Flash、EEPROM 以及熔丝配置位进行编程操作，在第 5 章中将给出利用 ISP 功能的编程下载具体操作实际过程与相关的资料等。

2.7 AVR单片机内部资源的扩展和删减

以上我们以 ATmega16 为典型介绍了 AVR 单片机的基本结构、性能和特点。ATMEL 公司为满足不同的需求，在 AVR 基本内核的基础上，对芯片的内部资源进行了相应的扩展和删减，形成了 AVR 单片机几十种型号的芯片。尽管它们的功能和外部的引脚定义和封装各有不同，小到 8-12 个引脚，多到 100 个引脚，但它们的内核结构相同，指令相容，用户可根据实际需要进行选择。

2.8 思考与练习

1. 典型单片机由哪几部分组成？每部分的基本功能和作用是什么？
2. 了解 AVR 单片机的主要特点和性能。
3. 熟悉 ATmega16 的外围引脚名称和基本作用。
4. AVR 系列单片机内部有哪些主要的逻辑部件？
5. AVR 系列单片机的 CPU 是由哪些部件组成？它们的具体作用是什么？如何协同工作？

6. AVR 单片机系统有 2 个时钟系统，他们如何构成？其作用如何？
7. AVR 单片机是如何实现每 MHz 高达 1MIPS 的处理能力的？
8. 说明 AVR 单片机通用寄存器组的作用和功能？
9. 说明 AVR 单片机 I/O 寄存器的作用和功能？
10. ATmega16 单片机的存储器有几种类型？它们是如何构成和组织的？有何作用？
11. ATmega16 单片机的数据存储器的地址空间是如何分布的？
12. AVR 单片机的 SRAM 存储器和 EEPROM 存储器有和区别？用途是什么？
13. AVR 单片机熔丝位的作用和用途使什么，你认为本章中所介绍的那些熔丝位非常重要，并说明原因。
14. AVR 单片机的 I/O 寄存器空间是如何寻址？
15. 简述状态寄存器（SREG）各个控制位的作用。
16. 熟悉堆栈指针寄存器（SP）和堆栈的作用，并说明 AVR 单片机堆栈是如何工作的。
17. 使用电子电路版图设计软件（如：Protel）画出用 ATmega16 构成最小系统的电路图，并说明各个元器件的作用。
18. AVR 单片机有几种复位方式？它们是如何复位的？这些复位方式有何区别，在实际应用中应该如何使用这些复位方式？复位以后 AVR 单片机怎样开始工作？
19. 在设计 AVR 系统程序时，程序存储器最低几十个存储单元一般应放置什么指令？为什么？
20. AVR 有几种复位方式？仔细理解和分析它们的复位条件。并说明 AVR 复位系统所具备的优点。
21. AVR 系统上电后，一旦外部把 AVR 的 RESET 引脚拉低，使 AVR 进入复位状态后，接下来变化将如何？
22. 对AVR编程下载主要的目的是什么？对AVR的编程下载的方式有哪些？各有何优点和缺点，各适合在那些情况下使用？
23. 阅读 ATmega16 的英文版器件手册，找到其中与本章所介绍内容相关的部分，更仔细的进行学习和理解。
24. 在图 2-20 支持 ISP 功能的最小系统中，R1 的值取 10 欧姆或 10M 可以吗？请分析原因并说明理由。
25. 对于图 2-20 支持 ISP 功能的最小系统，可以使用外部晶体构成系统时钟，也可使用芯片内部 4M 的 RC 振荡源。两种方式各有何优点和缺点？熔丝位应该如何配置？
26. 对于图 2-20 支持 ISP 功能的最小系统，如果 AVR 的熔丝位配置为使用外部晶体，但图中的晶体、C1、C2、R2 并没有使用，此时会产生什么情况，如何解决？
27. 对于图 2-20 支持 ISP 功能的最小系统，如果 PB5、PB6、PB7 这 3 个引脚与外围器件连接，且为强上拉或强下拉，那么为什么要串入 3 个隔离电阻？请分析 3 个电阻的作用，并说明阻值在 2K 左右为合适的。
28. 用 ATmega16 构成一个简单的系统，该系统可以分别控制 8 个半导体发光二极管 LED 闪烁，设计并画出系统的电原理图，并说明使用元器件的作用。
29. 如何用 ATmega16 构成一个简单的嵌入式系统，该系统可以控制 1 个 LED 七段数码管，设计并画出系统的电原理图，并讲述设计思路。
30. 一般嵌入式系统经常采用 6-8 个 LED 数码管作为显示输出，如时钟显示、温度显示等。如何用 ATmega16 构成一个系统，该系统可以控制 8 个 LED 七段数码管，设计并画出系统的电原理图，并说明所使用芯片各个引脚的作用和各个元器件的数值、作用和设计思路。

本章参考文献：

1.《ATmega16 数据手册》（英文，CDROM），ATMEL，www.atmel.com
2.《AVR硬件设计要点》（英文，CDROM），ATMEL，www.atmel.com
3.《EMC设计要点》（英文，CDROM），ATMEL，www.atmel.com

第 3 章　AVR的指令与汇编系统

传统的 8 位单片机（如最典型的 8051 结构的单片机）大都采用复杂指令 CISC(Complex Instruction Set Computer) 系统体系。由于 CISC 结构存在指令系统不等长，指令数多，CPU 利用效率低，执行速度慢等缺陷，已不能满足和适应设计高档电子产品和嵌入式系统应用的需要。

作为 8 位的 AVR 单片机来讲，除了其具备比较完善和功能强大的硬件结构和组成外，其更重要的是它的内核和指令系统为先进的 RISC 体系结构，采用了大型快速存取寄存器组（32 个通用工作寄存器）、快速的单周期指令系统以及单级流水线等先进技术。因此，AVR 内核核指令系统的显著特点有：

1. 16/32 位定长指令

AVR 的一个指令字为 16 位或 32 位，其中大部分的指令为 16 位。采用定长指令，不仅使取指操作简单，提高了取指令的速度；同时也降低了在取指操作过程中的错误，提高了系统的可靠性。

2. 流水线操作

AVR 采用流水线技术，在前一条指令执行的时候，就取出现行的指令，然后以一个周期执行指令。大大提高了 CPU 的运行速度。

3. 大型快速存取寄存器组

传统的基于累加器的结构单片机（如 8051），需要大量的程序代码来完成和实现在累加器和存储器之间的数据传送。而在 AVR 单片机中，采用 32 个通用工作寄存器构成大型快速存取寄存器组，用 32 个通用工作寄存器代替了累加器（相当有 32 个累加器），从而避免了传统结构中累加器和存储器之间数据传送造成的瓶颈现象。

由于 AVR 单片机采用 RISC 结构，使得它具有高达 1MIPS / MHz 的高速运行处理能力。同时也能更好地适合采用高级语言（例如 C 语言、BASIC 语言）来编写系统程序，高效地开发出目标代码，以加快产品进入市场的时间和简化系统的设计、开发、维护和支持。

3.1　ATmega16 指令综述

指令是 CPU 用于控制各功能部件完成某一指定动作或操作的指示和命令。指令不同，CPU 和各个功能部件完成的动作也不一样，指令的功能也不同。程序员根据系统的要求，选用不同功能指令的有序组合就构成的程序。CPU 执行不同的程序，就能完成不同的任务。

CPU 指令的的集合或全体称为指令系统。指令系统是 CPU 的重要性能指标之一，也是学习以及使用单片机的重要内容。由于 CPU 结构的不同，每一种 CPU 的指令和功能也不同，因此学习 AVR，就必须要了解它的指令结构、功能和特点。只有在此基础上，才能更清楚的了解 AVR 的硬件使用，编写出好的系统程序。

AVR 单片机指令系统是 RISC 结构的精简指令集，是一种简明、易掌握、效率高的指令系统。ATmega16 单片机完全兼容 AVR 的指令系统，具有高性能的数据处理能力，能对位、

半字节、字节和双字节数据进行各种操作，包括算术和逻辑运算、数据传送、布尔处理、控制转移和硬件乘法等操作。

ATmega16 共有 131 条指令，按功能可分为五大类，它们是：

- 算术和逻辑运算指令（28 条）；
- 比较和跳转指令（36 条）；
- 数据传送指令（35 条）；
- 位操作和位测试指令（28 条）；
- MCU 控制指令（4 条）。

在本章中将对 ATmega16 全部的 131 条指令，包括字节数、功能、对标志位的影响以及执行周期数等进行简单的描述。

3.1.1 指令格式以及三种表示方式

指令格式是指指令码的结构形式。通常，指令可分为操作码和操作数两部分。其中操作码部分比较简单，操作数部分则比较复杂，而且随 CPU 类型的不同，寻址方式的不同有较大的变化。

AVR 的指令的一般格式为：

操作码	第 1 操作数或操作数地址	第 2 操作数或操作数地址

其中，操作码用于指示 CPU 执行何种操作，是加法操作还是减法操作，是数据传送还是数据移位等。第 1 操作数或操作数地址用于表示参与操作的第 1 个操作数，或该操作数在内存的地址，同时该地址也将作为操作结果存放的地址。第 2 操作数或操作数地址（如果有的话）用于表示参与操作的第 2 个操作数，或该操作数在内存的地址。需要注意的是，在 AVR 的指令中，有相当一部分只有操作码，或只有操作码和第 1 操作数或操作数地址，前者在操作码中隐含了操作数或操作数的地址。

指令的表示方式是指采用何种形式描述指令，也是人们用于编写和阅读程序的基础。通常指令采用二进制、十六进制和助记符三种表示方式。

指令的二进制表示形式，是一种可以直接为 CPU 识别和执行的形式，故称为指令的机器码或汇编语言的目标代码，下载到 AVR 中的代码必须是可执行的目标代码。但二进制形式的代码具有难读、难写、难记忆和难修改等缺点，因此人们通常不用它来编写程序。

指令的十六进制形式是二进制形式的变型，只是将二进制代码 4 位一组用十六进制的形式描述。十六进制的形式虽然比二进制形式读写方便些，但还是不易被人们识别和修改，所以通常也不被用于编写程序，只是在某些场合，如调试环境中的指令字的显示，或调试程序、修改调整个别指令代码时作为输入程序的辅助手段。

指令的助记符形式又称为指令的汇编形式或汇编语句，是一种用英文单词或缩写字母以及数字来表征指令功能的形式。这种形式不仅容易为人们识别和读写，也方便记忆和交流，因此也是人们用于进行程序设计的一种常用的形式。

由于 CPU 可以直接识别和执行的指令形式必须是二进制表示形式的，因此不管使用十六进制的形式还是汇编形式构成的程序，都需要通过人工或机器把它们翻译成二进制机器码的形式，才能下载到芯片中被 CPU 执行。

现在决大多数单片机都提供相应的，能够在 PC 上工作的的开发平台，其最基本的功能就是提供用户编写汇编代码的源程序，并能将汇编源程序翻译成二进制的机器码，生成可

下载的目标代码文件。

3.1.2 AVR指令系统中使用的符号

在本章所列出 ATmega16 所有的 131 条指令中，给出了全部指令的汇编助记符、操作数、操作说明、相应的操作、操作数的范围、对标志位的影响、以及指令的执行周期。

在指令描述中，除了操作码采用了助记符表示外，还在指令的操作数的描述说明中采用了一些符号代码，下面对所使用符号的意义进行简单的说明。

1．状态寄存器与标志位

SREG： 8 位状态寄存器，其中每一位的定义为：

C： 进位标志位

Z： 结果为零标志位

N： 结果为负数标志位

V： 2 的补码溢出标志位

S： N⊕V，用于符号测试的标志位

H： 操作中产生半进位的标志位

T： 用于和 BLD、BST 指令进行位数据交换的位

I： 全局中断触发/禁止标志位

2．寄存器和操作码

Rd： 目的（或源）寄存器，取值为 R0～R31 或 R16～R31（取决于指令）。

Rr： 源寄存器，取值为 R0～R31。

A： I/O 寄存器，取值为 0～63 或 0～31（取决于指令）。

b： I/O 寄存器中的指定位，常数（0～7）。

s： 状态寄存器 SREG 中的指定位，常数（0～7）。

K： 立即数，常数（0～255）。

k： 地址常数，取值范围取决于指令。

q： 地址偏移量常数（0～63）。

X、Y、Z：地址指针寄存器（X=R27:R26；Y=R29:R28；Z=R31:R30）。

3．堆栈

STACK： 作为返回地址和压栈寄存器的堆栈

SP： 堆栈 STACK 的指针

3.1.3 AVR指令的寻址方式和寻址空间

指令的一个重要组成部分是操作数。指令给出参与运算数据的方式称为寻址方式。CPU 执行指令时，首先要根据地址获取参加操作的操作数，然后才能对操作数进行操作，操作的结果还要根据地址保存在相应的储存器或寄存器中。因此 CPU 执行程序实际上是不断的寻找操作数并进行操作的过程。通常，指令的寻址方式有多种，寻址方式越多，指令的功能也就越强。

AVR 单片机指令操作数的寻址方式有以下几种：单寄存器直接寻址、双寄存器直接寻址、I/O 寄存器直接寻址、数据存储器直接寻址、数据存储器间接寻址、带后增量的数据存储器间接寻址、带预减量的数据存储器间接寻址、带位移的数据存储器间接寻址、程序

存储器取常量寻址、程序存储器空间直接寻址、程序存储器空间间接寻址、程序相对寻址等。

1. 单寄存器直接寻址（图 3-1）

由指令指定一个寄存器的内容作为操作数，在指令中给出寄存器的直接地址，这种寻址方式称为单寄存器直接寻址。单寄存器寻址的地址范围限制为通用工作寄存器组中的 32 个寄存器 R0～R31，或后 16 个寄存器 R16～R31（取决于不同指令）。

例：INC Rd；操作：Rd←Rd+1。

INC R5；将寄存器 R5 内容加 1 回放。

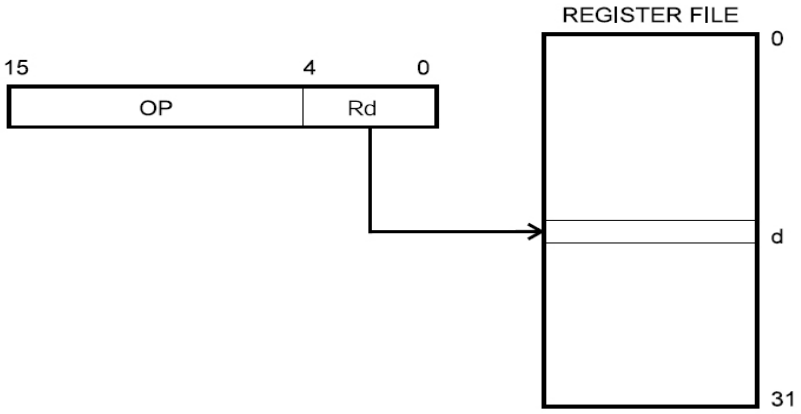

图 3-1 单寄存器直接寻址

2. 双寄存器直接寻址（图 3-2）

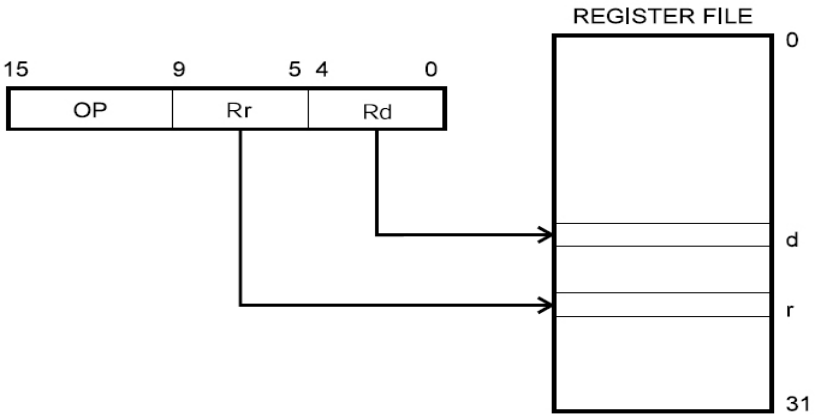

图 3-2 双寄存器直接寻址

双寄存器直接寻址方式与单寄存器直接寻址方式相似，它是将指令指出的两个寄存器 Rd 和 Rr 的内容作为操作数，而结果存放在 Rd 寄存器中。指令中同时给出两个寄存器的直接地址，这种寻址方式称为双寄存器直接寻址。双寄存器寻址的地址范围限制为通用工作寄存器组中的 32 个寄存器 R0～R31，或后 16 个寄存器 R16～R31，或后 8 个寄存器 R16～R23（取决于不同指令）。

例：ADD Rd，Rr；操作：Rd←Rd+Rr。

ADD R0，R1；将 R0 和 R1 寄存器内容相加，结果回放 R0。

3. I/O 寄存器直接寻址（图 3-3）

由指令指定一个 I/O 寄存器的内容作为操作数。在指令中直接给出 I/O 寄存器的地址，这种寻址方式称为 I/O 寄存器直接寻址。I/O 寄存器直接寻址的地址使用 I/O 寄存器空间的地址$00～$3F，共 64 个，取值为 0～63 或 0～31（取决于指令）。

例：IN Rd，P；操作：Rd←P。

IN R5，$3E；读 I/O 空间地址为$3E 寄存器（SPH）的内容，放入寄存器 R5。

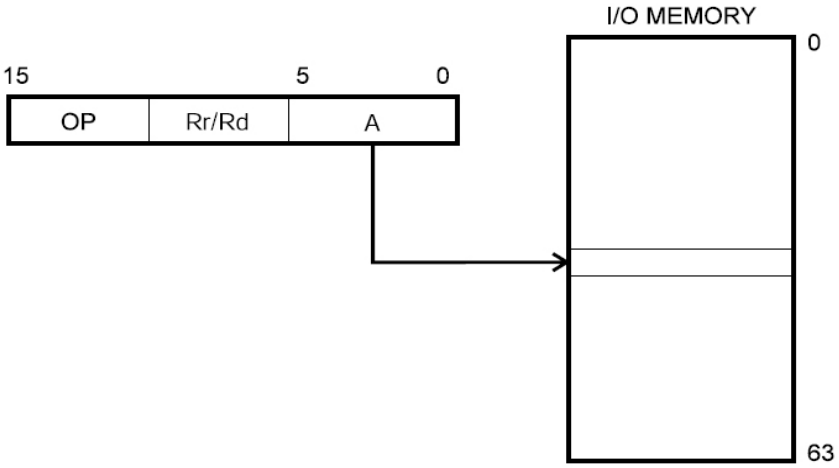

图 3-3 I/O 寄存器直接寻址

4．数据存储器空间直接寻址（图 3-4）

数据存储器空间直接寻址方式用于直接 CPU 从 SRAM 存储器中存取数据。数据存储器空间直接寻址为双字指令，在指令的低字中指出一个 16 位的 SRAM 地址。

例：LDS Rd，K；操作：Rd←（K）。

LDS R18，$100；读地址为$100 的 SRAM 中内容，传送到 R18 中。

指令中 16 位 SRAM 的地址字长度限定了 SRAM 的地址空间为 64K 字节，该地址空间实际包含了 32 个通用寄存器和 64 个 I/O 寄存器。因此，也可使用数据存储器空间直接寻址的方式读取通用寄存器或 I/O 寄存器中的内容，此时应使用这些寄存器在 SRAM 空间的映射地址，而且其效率也比使用寄存器直接寻址的方式要低。原因在于数据存储器空间直接寻址的指令为双字指令，指令周期为 2 个系统时钟。

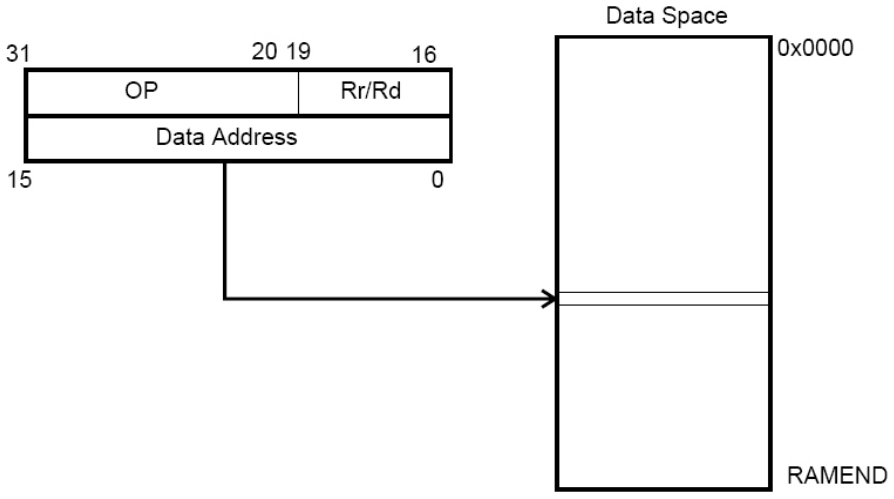

图 3-4 数据存储器空间直接寻址

5．数据存储器空间的寄存器间接寻址（图 3-5）

由指令指定某一个 16 位寄存器的内容作为操作数在 SRAM 中的地址，该寻址方式称为数据存储器空间的寄存器间接寻址。AVR 单片机中使用 16 位寄存器 X、Y 或 Z 作为规定的地址指针寄存器，因此操作数的 SRAM 地址在间址寄存器 X、Y 或 Z 中。

例：LD Rd，Y；操作：Rd ←（Y），把以 Y 为指针的 SRAM 的内容送 Rd。

LD R16，Y；设 Y=$0567，即把 SRAM 地址为$0567 的内容传送到 R16 中。

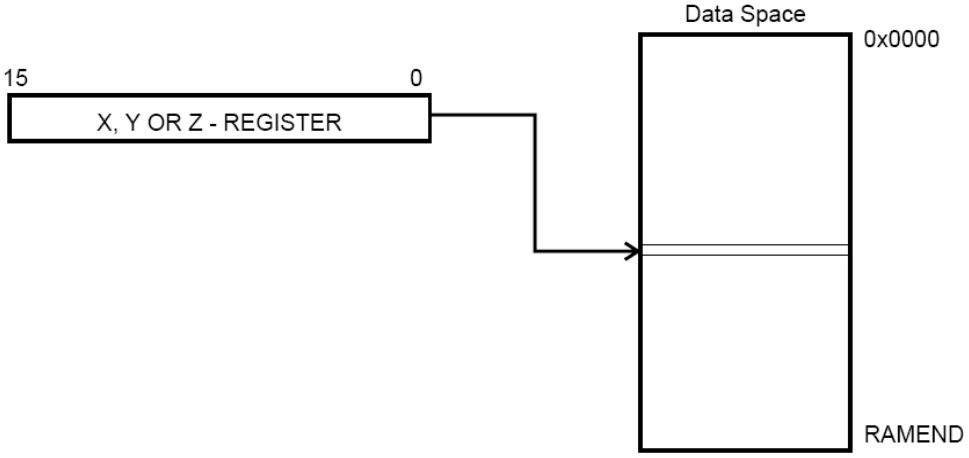

图 3-5 数据存储器空间的寄存器间接寻址

6．带后增量的数据存储器空间的寄存器间接寻址（图 3-6）

这种寻址方式类似于数据存储器空间的寄存器间接寻址方式，间址寄存器 X、Y、Z 中的内容仍为操作数在 SRAM 空间的地址，但指令在间接寻址操作后，再自动把间址寄存器中的内容加 1。这种寻址方式特别适用于访问矩阵、查表等应用。

例：LD Rd，Y+；操作：Rd←（Y），Y=Y+1，先把以 Y 为指针的 SRAM 的内容送 Rd，再把 Y 增 1。

LD R16，Y+；设原 Y=$0567，指令把 SRAM 地址为$0567 的内容传送到 R16 中，再将 Y 的值加 1，操作完成后 Y=$0568。

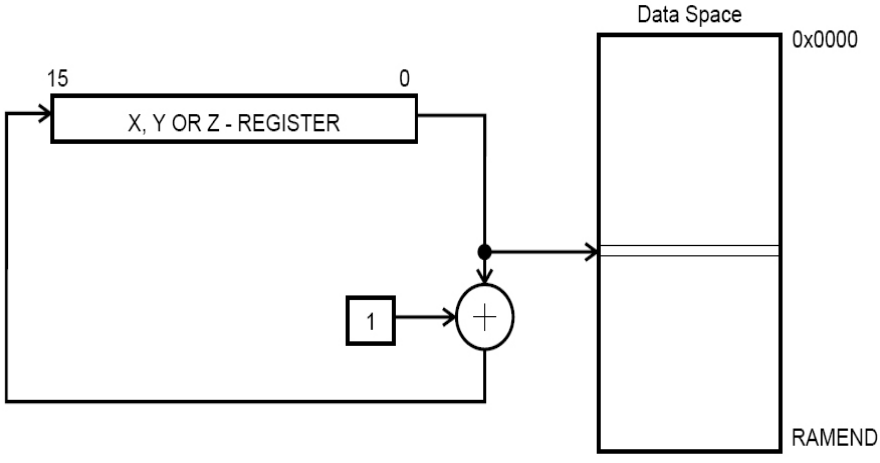

图 3-6 带后增量的数据存储器空间的寄存器间接寻址

7．带预减量的数据存储器空间寄存器间接寻址（图 3-7）

这种寻址方式类似于数据存储器空间的寄存器间接寻址方式，间址寄存器 X、Y、Z 中的内容仍为操作数在 SRAM 空间的地址，但指令在间接寻址操作之前，先自动将间址寄存器中的内容减 1，然后把减 1 后的内容作为操作数在 SRAM 空间的地址。这种寻址方式也特别

适用于访问矩阵、查表等应用。

例：LD Rd，-Y；操作：Y=Y-1，Rd←（Y），先把 Y 减 1，再把以 Y 为指针的 SRAM 的内容送 Rd。

LD R16，-Y；设原 Y=$0567，指令即先把 Y 减 1，Y=$0566，再把 SRAM 地址为$0566 的内容传送到 R16 中。

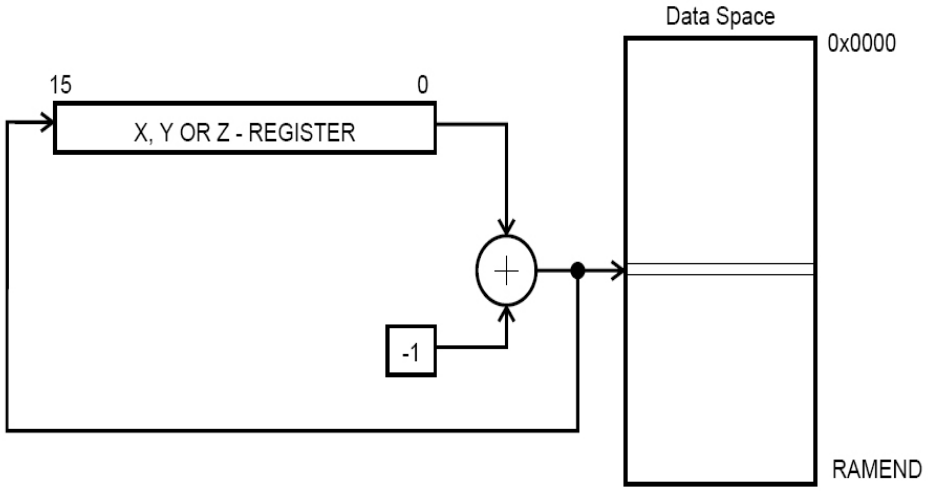

图 3-7 带预减量的数据存储器空间寄存器间接寻址

8．带位移的数据存储器空间寄存器间接寻址（图 3-8）

带位移的数据存储器空间寄存器间接寻址方式是：由间址寄存器（Y 或 Z）中的内容和指令字中给出的地址偏移量共同决定操作数在 SRAM 空间的地址，偏移量的范围为 0～63。

例：LDD Rd，Y+q；操作：Rd←（Y+q），其中 0≤q≤63，即把以 Y+q 为地址的 SRAM 的内容送 Rd，而 Y 寄存器的内容不变。

LDD R16，Y+31；设 Y=$0567，把 SRAM 地址为$0598 的内容传送到 R16 中，Y 寄存器的内容不变。

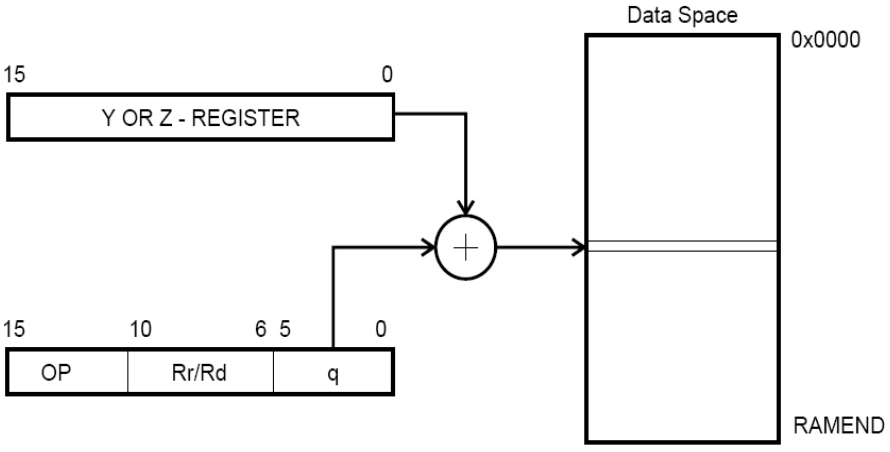

图 3-8 带位移的数据存储器空间寄存器间接寻址

9．程序存储器空间取常量寻址（图 3-9）

程序存储器空间取常量寻址主要从程序存储器 Flash 中读取常量，此种寻址方式只用于指令 LPM。程序存储器中常量字节的地址由地址寄存器 Z 的内容确定。Z 寄存器的高 15 位用于选择字地址（程序存储器的存储单元为字），而 Z 寄存器的最低位 Z(d0)用于确定字

地址的高/低字节。若 d0=0，则选择字的低字节；d0=1，则选择字的高字节。

例：LPM；操作：R0 ←（Z），即把以 Z 为指针的程序存储器的内容送 R0。

若 Z=$0100，即把地址为$0080 的程序存储器的低字节内容送 R0。

若 Z=$0101，即把地址为$0080 的程序存储器的高字节内容送 R0。

例：LPM R16，Z；操作：R16←（Z），即把以 Z 为指针的程序存储器的内容送 R16。

若 Z=$0100，即把地址为$0080 的程序存储器的低字节内容送 R16。

若 Z=$0101，即把地址为$0080 的程序存储器的高字节内容送 R16。

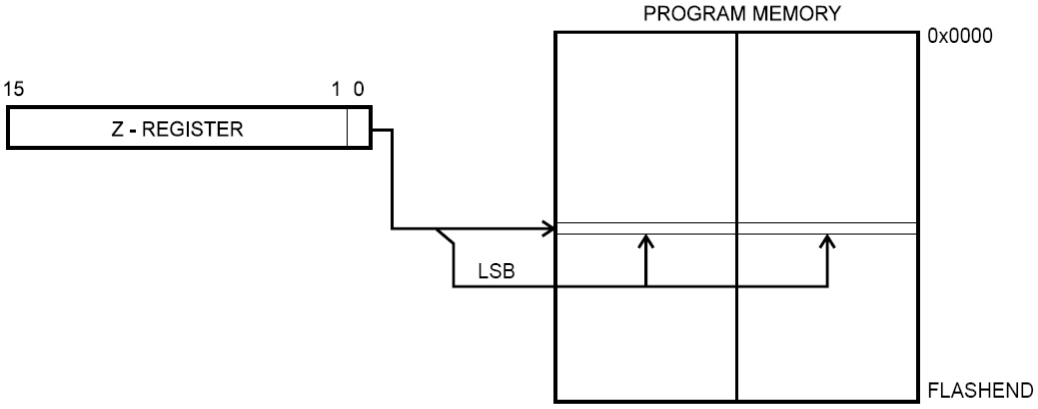

图 3-9 程序存储器空间取常量寻址

10．带后增量的程序存储器空间取常量寻址（图 3-10）

带后增量的程序存储器空间取常量寻址主要从程序存储器 Flash 中取常量，此种寻址方式只用于指令 LPM Rd，Z+。程序存储器中常量字节的地址由地址寄存器 Z 的内容确定。Z 寄存器的高 15 位用于选择字地址（程序存储器的存储单元为字），而 Z 寄存器的最低位 Z（d0）用于确定字地址的高/低字节。若 d0=0，则选择字的低字节；d0=1，则选择字的高字节。寻址操作后，Z 寄存器的内容加 1。

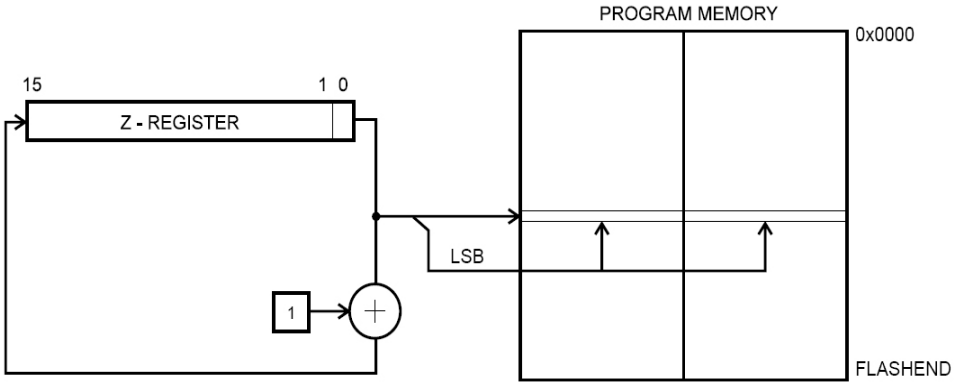

图 3-10 带后增量的程序存储器空间取常量寻址

例：LPM R16，Z+；操作：R16←（Z）；Z←Z+1，即把以 Z 为指针的程序存储器的内容送 R16，然后 Z 的内容加 1。

若 Z=$0100,即把地址为$0080 的程序存储器的低字节内容送 R16,完成后 Z=$0101。

若 Z=$0101,即把地址为$0080 的程序存储器的高字节内容送 R16,完成后 Z=$0102。

11．程序存储器空间写数据寻址（图 3-9）

程序存储器空间写数据寻址主要用于可进行在系统自编程的 AVR 单片机，此种寻址方式只用于指令 SPM。该指令将寄存器 R1 和 R0 中的内容组成一个字 R1:R0，然后写入由 Z 寄

存器的内容作为地址（Z 寄存器的最低位必须为 0）的程序存储器单元中。（注意：实际是写入到 Flash 的页缓冲区中）。

例：SPM；操作：（Z）←R1:R0，把 R1:R0 内容写入以 Z 为指针的程序存储器单元。

12．程序存储器空间直接寻址（图 3-11）

程序存储器空间直接寻址方式用于程序的无条件跳转指令 JMP、CALL。指令中含有一个 16 位的操作数，指令将操作数存入程序计数器 PC 中，作为下一条要执行指令在程序存储器空间的地址。JMP 类指令和 CALL 类指令的寻址方式相同，但 CALL 类的指令还包括了返回地址的压进堆栈和堆栈指针寄存器 SP 内容减 2 的操作。

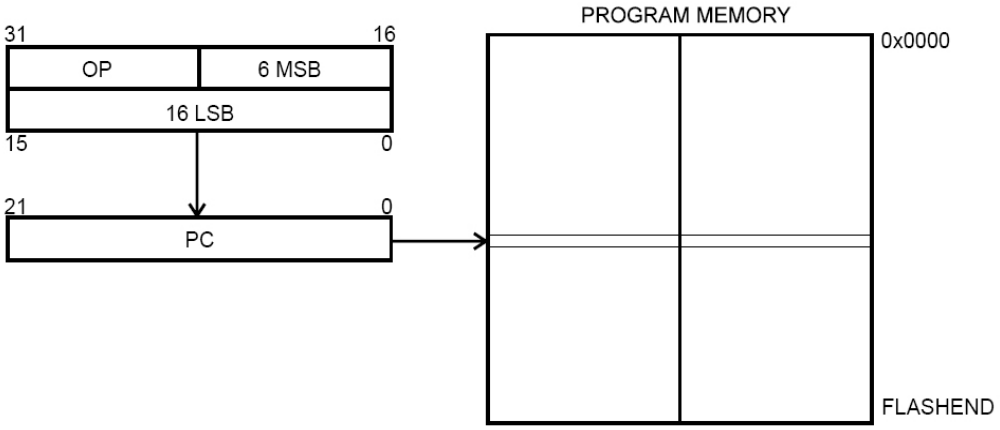

图 3-11 程序存储器空间直接寻址

例：JMP $0100；操作：PC←$0100。程序计数器 PC 的值设置为$0100，接下来执行程序存储器$0100 单元的指令代码。

例：CALL $0100；操作：STACK←PC+2；SP←SP-2；PC←$0100。先将程序计数器 PC 的当前值加 2 后压进堆栈（CALL 指令为 2 个字长），堆栈指针计数器 SP 内容减 2，然后 PC 的值为$0100，接下来执行程序存储器$0100 单元的指令代码。

13．程序存储器空间 Z 寄存器间接寻址（图 3-12）

程序存储器空间间接寻址方式是使用 Z 寄存器存放下一步要执行指令代码程序地址，程序转到 Z 寄存器内容所指定程序存储器的地址处继续执行，即用寄存器 Z 的内容代替 PC 的值。此寻址方式用于 IJMP、ICALL 指令。

例：IJMP；操作：PC←Z，即把 Z 的内容送程序计数器 PC。若 Z=$0100，即把$0100 送程序计数器 PC，接下来执行程序存储器$0100 单元的指令代码。

例：ICALL；操作：STACK←PC+1；SP←SP-2；PC←Z。若 Z=$0100，先将程序计数器 PC 的当前值加 1 后压进堆栈，堆栈指针计数器 SP 内容减 2，然后 PC 的值为$0100，接下来执行程序存储器$0100 单元的指令代码。

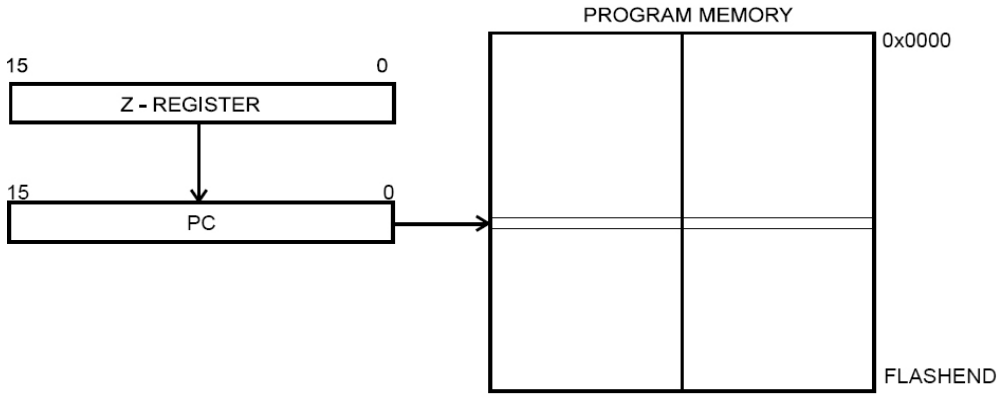

图 3-12 程序存储器空间 Z 寄存器间接寻址

14．程序存储器空间相对寻址（图 3-13）

在程序存储器空间相对寻址方式中，在指令中包含一个相对偏移量 k，指令执行时，首先将当前程序计数器 PC 值加 1 后再与偏移量 k 相加，作为程序下一条要执行指令的地址。此寻址方式用于 RJMP、RCALL 指令。

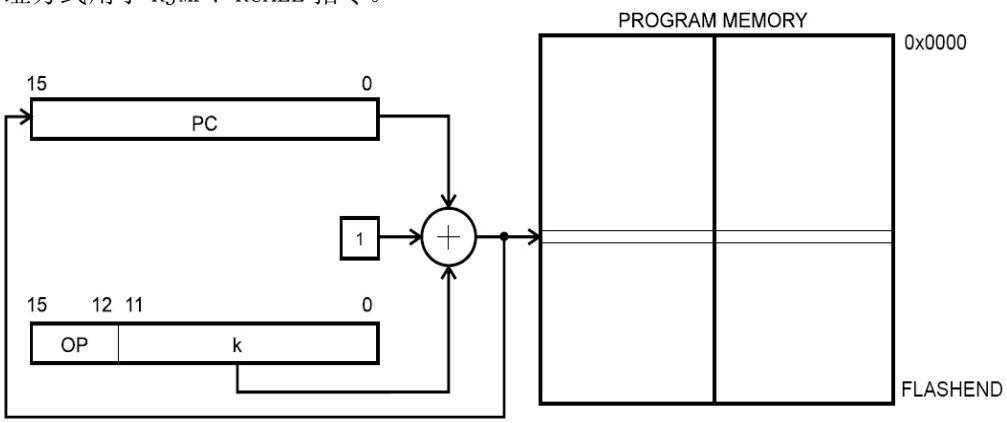

图 3-13 程序存储器空间相对寻址

例：RJMP $0100；操作：PC←PC+1+$0100。若当前指令地址为$0200（PC=$0200），即把$0301 送程序计数器 PC，接下来执行程序存储器$0301 单元的指令代码。

例：RCALL $0100；操作：STACK←PC+1；SP←SP-2；PC←PC+1+$0100。若当前指令地址为$0200（PC=$0200），先将程序计数器 PC 的当前值加 1 后压进堆栈，堆栈指针计数器 SP 内容减 2，然后 PC 的值为$0301，接下来执行程序存储器$0301 单元的指令代码。

15．数据存储器空间堆栈寄存器 SP 间接寻址（图 3-14）

数据存储器空间堆栈寄存器 SP 间接寻址，是将 16 位的堆栈寄存器 SP 的内容作为操作数在 SRAM 空间的地址，此寻址方式用于 PUSH、POP 指令。

例：PUSH R0；操作：STACK←R0；SP←SP-1。若当前 SP=$10FF，先把寄存器 R0 的内容送到 SRAM 的$10FF 单元，再将 SP 内容减 1，即 SP=$10FE。

例：POP R1；操作：SP←SP+1；R1←STACK。若当前 SP=$10FE，先将 SP 内容加 1，再把 SRAM 的$10FF 单元内容送到寄存器 R1，此时 SP=$10FF。

此外，在 CPU 响应中断和执行 CALL 一类的子程序调用指令（SP = SP-2），以及执行中断返回 IRET 和子程序返回 RET 一类的子程序返回指令中（SP = SP+2），都隐含着使用堆栈寄存器 SP 间接寻址的方式。

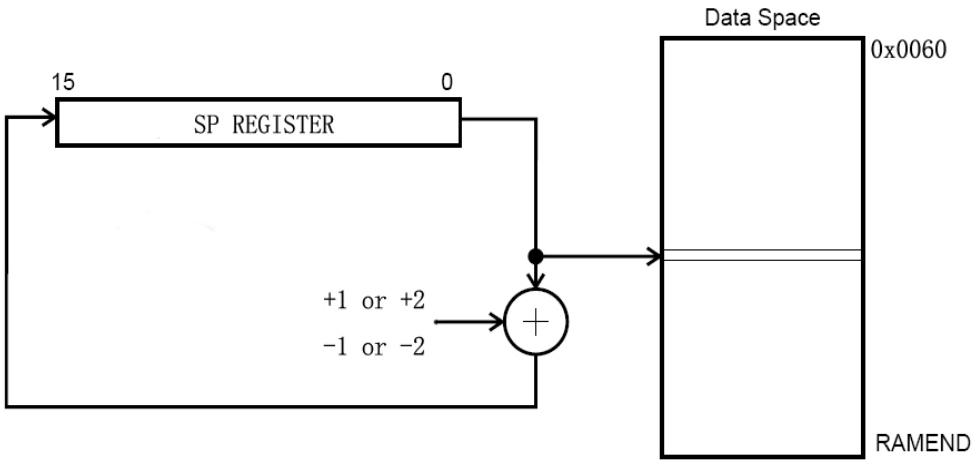

图 3-14 数据存储器空间堆栈寄存器 SP 间接寻址

3.1.4 AVR指令操作结果对标志位的影响

在AVR的指令中，一类指令执行后要影响到状态寄存器中某些标志位的状态，即不论指令执行前标志位状态如何，指令执行后总是根据执行结果的定义形成新的状态标志。另一类指令在执行后不会影响标志位，标志位原来什么状态，指令执行后标志状态还是保持不变。

由于AVR的CPU响应中断时，硬件不保护状态寄存器，所以在编写中断服务处理程序时，应注意将状态寄存器进行保护（如用PUSH指令压入到堆栈），在中断返回前还要恢复状态寄存器在进入中断前时的标志状态（如用POP指令从堆栈中弹出）。

3.2 算术和逻辑指令

AVR 的算术运算指令有加、减、乘法、取反、取补、比较指令、增量和减量指令。逻辑运算指令有与、或和异或指令等。

3.2.1 加法指令

1．不带进位位加法

ADD Rd，Rr　　0≤d≤31，0≤r≤31

说明：两个寄存器不带进位 C 标志相加，结果送目的寄存器 Rd。

操作：Rd←Rd+Rr　　PC←PC+1　　机器码：0000 11rd dddd rrrr

对标志位的影响：H S V N Z C

2．带进位位加法

ADC Rd，Rr　　0≤d≤31，0≤r≤31

说明：两个寄存器和 C 标志的内容相加，结果送目的寄存器 Rd。

操作：Rd←Rd+Rr+C　PC←PC+1　　机器码：0001 11rd dddd rrrr

对标志位的影响：H S V N Z C

3．字加立即数

ADIW Rdl,K　　dl 为：24、26、28、30，0≤K≤63

说明：寄存器对（一个字）同立即数（0～63）相加，结果放到寄存器对。

操作：Rdh:Rdl←Rdh:Rdl+K　　PC←PC+1　　机器码：1001 0110 KKdd KKKK

对标志位的影响：S V N Z C

注意：dl 只能取 24、26、28、30，即仅用于最后 4 个寄存器对。K 为 6 位二进制无符号数（0～63）。

4．增 1 指令

INC Rd　　0≤d≤31

说明：寄存器 Rd 的内容加 1，结果送目的寄存器 Rd 中。该操作不改变 SREG 中的 C 标志，所以 INC 指令允许在多倍字长计算中用作循环计数。当对无符号数操作时，仅有 BREQ（相等跳转）和 BRNE（不为零跳转）指令有效。当对二进制补码值操作时，所有的带符号跳转指令都有效。

操作：Rd←Rd+1　　PC←PC+1　　机器码：1001 010d dddd 0011

对标志位的影响：S V N Z

3.2.2　减法指令

1．不带进位位减法

SUB Rd，Rr　　0≤d≤31，0≤r≤31

说明：两个寄存器相减结果送目的寄存器 Rd 中。

操作：Rd←Rd-Rr　　PC←PC+1　　机器码：0001 10rd dddd rrrr

对标志位的影响：H S V N Z C

2．减立即数（字节）

SUBI Rd，K　　16≤d≤31，0≤K≤255

说明：一个寄存器和常数相减，结果送目的寄存器 Rd。该指令工作于寄存器 R16～R31 之间，非常适合 X、Y 和 Z 指针的操作。

操作：Rd←Rd-K　　PC←PC+1　　机器码：0101 KKKK dddd KKKK

对标志位的影响：H S V N Z C

3．带进位位减法

SBC Rd，Rr　　0≤d≤31，0≤r≤31

说明：两个寄存器带着 C 标志相减，结果放到目的寄存器 Rd 中。

操作：Rd←Rd-Rr-C　　PC←PC+1　　机器码：0000 10rd dddd rrrr

对标志位的影响：H S V N Z C

4．带进位位减立即数（字节）

SBCI Rd，K　　16≤d≤31，0≤K≤255

说明：寄存器和立即数带着 C 标志相减，结果放到目的寄存器 Rd 中。

操作：Rd←Rd-K-C　　PC←PC+1　　机器码：0100 KKKK dddd KKKK

对标志位的影响：H S V N Z C

5．减立即数（字）

SBIW Rdl，K　　dl 为 24、26、28、30，0≤K≤63

说明：寄存器对（字）与立即数 0～63 相减，结果放入寄存器对。

操作：Rdh:Rdl←Rdh:Rdl-K　　PC←PC+1　　机器码：1001 0111 KKdd KKKK

注意：dl 只能取 24、26、28、30，即仅用于最后 4 个寄存器对。K 为 6 位二进制无符号数（0～63）。

对标志位的影响：S V N Z C

6．减 1 指令

DEC Rd　　0≤d≤31

说明：寄存器 Rd 的内容减 1，结果送目的寄存器 Rd 中。该操作不改变 SREG 中的 C 标志，所以 DEC 指令允许在多倍字长计算中用作循环计数。当对无符号值操作时，仅有 BREQ（不相等跳转）和 BRNE（不为零跳转）指令有效。当对二进制补码值操作时，所有的带符号跳转指令都有效。

操作：Rd←Rd-1　PC←PC+1　　机器码：1001 010d dddd 1010

对标志位的影响：S V N Z

3.2.3　取反码指令

COM Rd　　0≤d≤31

说明：该指令完成对寄存器 Rd 的二进制反码操作。

操作：Rd←$FF-Rd　　PC←PC+1　　机器码：1001 010d dddd 0000

对标志位的影响：S N Z V(0) C(1)

3.2.4　取补码指令

NEG Rd　　0≤d≤31

说明：寄存器 Rd 的内容转换成二进制补码值。

操作：Rd←$00-Rd　　PC←PC+1　　机器码：1001 010d dddd 0001

对标志位的影响：H S V N Z C

3.2.5　比较指令

1．寄存器比较

CP Rd，Rr　　0≤d≤31，0≤r≤31

说明：该指令完成两个寄存器 Rd 和 Rr 相比较操作，而寄存器的内容不改变，该指令后能使用所有条件跳转指令。

操作：Rd-Rr　　PC←PC+1　　机器码：0001 01rd dddd rrrr

对标志位的影响：H S V N Z C

2．带进位比较

CPC Rd，Rr　0≤d≤31，　0≤r≤31

说明：该指令完成寄存器 Rd 的值和寄存器 Rr 加 C 相比较操作，而寄存器的内容不改变，该指令后能使用所有条件跳转指令。

操作：Rd-Rr-C　　PC←PC+1　　机器码：0000 01rd dddd rrrr

对标志位的影响：H S V N Z C

3．与立即数（字节）比较

CPI Rd，K　　16≤d≤31，0≤K≤255

说明：该指令完成寄存器 Rd 和常数的比较操作，寄存器的内容不改变，该指令后能使用所有条件跳转指令。

操作：Rd-K　　PC←PC+1　　机器码：0011 KKKK dddd KKKK

对标志位的影响：H S V N Z C

3.2.6 逻辑与指令

1．寄存器逻辑与

AND Rd，Rr　　0≤d≤31，0≤d≤31

说明：寄存器 Rd 和寄存器 Rr 的内容逻辑与，结果送目的寄存器 Rd。

应用：清 0 某位，用 0 去同该位逻辑与；保留某位值，用 1 去逻辑与；代替硬件与门。

操作 Rd←Rd • Rr　　PC←PC+1　　机器码：0010 00rd dddd rrrr

对标志位的影响：S V(0) N Z

2．与立即数（字节）

ANDI Rd，K　　16≤d≤31，0≤K≤255

说明：寄存器 Rd 的内容与常数逻辑与，结果送目的寄存器 Rd。

应用：清 0 某位时，用 0 去同该位逻辑与；保留某位的值，用 1 去逻辑与；代替硬件与门。

操作：Rd←Rd • K　　PC←PC+1　　机器码：0111 KKKK dddd KKKK

对标志位的影响：S V(0) N Z

3．寄存器位清零

CBR Rd，K　　16≤d≤31，0≤K≤255

说明：清除寄存器 Rd 中的指定位，利用寄存器 Rd 的内容与常数表征码 K 的补码相与，其结果放在寄存器 Rd 中。

操作：Rd←Rd •（$FF-K）　PC←PC+1　　机器码：0111 ~~KKKK~~ dddd ~~KKKK~~（~~KKKK~~ 为 KKKK 的补码）

对标志位的影响：S V(0) N Z

4．测试寄存器为零或负

TST Rd　　0≤d≤31

说明：测试寄存器为零或负，实现寄存器内容自己同自己的逻辑与操作，而寄存器内容不改变。

操作：Rd←Rd • Rd　　PC←PC+1　　机器码：0010 00dd dddd dddd

对标志位的影响：S V(0) N Z

3.2.7 逻辑或指令

1．寄存器逻辑或

OR Rd，Rr　　0≤d≤31，　0≤r≤31

应用：置数，使某位为 1，用 1 去或；保留，用 0 去逻辑或；代替硬件或门。

说明：完成寄存器 Rd 与寄存器 Rr 的内容逻辑或操作，结果送目的寄存器 Rd 中。

操作：Rd←Rd∨Rr　　PC←PC+1　　机器码：0010 10rd dddd rrrr

对标志位的影响：S V(0) N Z

2．或立即数(字节)

ORI Rd，K　　16≤d≤31，　0≤K≤255

说明：完成寄存器 Rd 的内容与常量 K 逻辑或操作，结果送目的寄存器 Rd 中。

操作：Rd←Rd∨K　　PC←PC+1　　机器码：0110 KKKK dddd KKKK

对标志位的影响：S V(0) N Z

3．置寄存器位

SBR Rd，K　　16≤d≤31，　0≤K≤255

说明：用于对寄存器 Rd 中指定位置位。完成寄存器 Rd 和常数表征码 K 之间的逻辑或操作，结果送目的寄存器 Rd。

操作：Rd←Rd∨K　　PC←PC+1　　机器码：0110 KKKK dddd KKKK

对标志位的影响：S V(0) N Z

4．置寄存器为$FF

SER Rd　　16≤d≤31

说明：直接装入$FF 到寄存器 Rd。

操作：Rd←$FF　　PC←PC+1　　机器码：1110 1111 dddd 1111

对标志位的影响：无

3.2.8　逻辑异或指令

1．寄存器异或

EOR Rd，Rr　　0≤d≤31，　0≤r≤31

说明：完成寄存器 Rd 和寄存器 Rr 的内容逻辑异或操作，结果送目的寄存器 Rd。

操作：Rd←Rd⊕Rr　　PC←PC+1　　机器码：0010 01rd dddd rrrr

对标志位的影响：S V(0) N Z

2．寄存器清零

CLR Rd　　0≤d≤31

说明：寄存器清零。该指令采用寄存器 Rd 与自己的内容相异或实现的寄存器的所有位都被清零。

操作：Rd←Rd⊕Rd　　PC←PC+1　　机器码：0010 01dd dddd dddd

对标志位的影响：S(0) V(0) N(0) Z(1)

3.2.9　乘法指令

1．无符号数乘法

MUL Rd，Rr　　0≤d≤31，　0≤r≤31

说明：该指令完成将寄存器 Rd 和寄存器 Rr 的内容作为两个无符号 8 位数的乘法操作，结果为 16 位的无符号数，保存在 R1:R0 中，R1 为高 8 位，R0 为低 8 位。如果操作数为寄存器 R1 或 R0，则结果会将原操作数覆盖。

操作：R1:R0=Rd×Rr　　PC←PC+1　　机器码：1001 11rd dddd rrrr

对标志位的影响：Z C

2．有符号数乘法

MULS Rd，Rr　　16≤d≤31，　16≤r≤31

说明：该指令完成将寄存器 Rd 和寄存器 Rr 的内容作为两个 8 位有符号数的乘法操作，

结果为 16 位的有符号数，保存在 R1:R0 中，R1 为高 8 位，R0 为低 8 位。源操作数为寄存器 R16～R31。

操作：R1:R0=Rd×Rr　　PC←PC+1　　机器码：0000 0010 dddd rrrr

对标志位的影响：Z C

3．有符号数与无符号数乘法

MULSU Rd，Rr　　16≤d≤23，16≤r≤23

说明：该指令完成将寄存器 Rd（8 位，有符号数）和寄存器 Rr（8 位，无符号数）的内容相乘操作，结果为 16 位的有符号数，保存在 R1:R0 中，R1 为高 8 位，R0 为低 8 位。源操作数为寄存器 R16～R23。

操作：R1:R0=Rd×Rr　　PC←PC+1　　机器码：0000 0011 0ddd 0rrr

对标志位的影响：Z C

4．无符号定点小数乘法

FMUL Rd，Rr　　16≤d≤23，16≤r≤23

说明：该指令完成将寄存器 Rd（8 位无符号数）和寄存器 Rr（8 位无符号数）的内容相乘操作，结果为 16 位的无符号数，并将结果左移一位后保存在 R1:R0 中，R1 为高 8 位，R0 为低 8 位。源操作数为寄存器 R16～R23。

操作：R1:R0=Rd×Rr　（unsigned（1.15）=unsigened（1.7）×unsigened（1.7））

PC←PC+1　　机器码：0000 0011 0ddd 1rrr

对标志位的影响：Z C

注：（n.q）表示一个小数点左边有 n 个二进制数位、小数点右边有 q 个二进制数位的小数。以（n1.q1）和（n2.q2）为格式的两个小数相乘，产生格式为（(n1+n2)．(q1+q2)）的结果。

对于要有效保留小数位的处理应用，输入的数据通常采用（1.7）的格式，产生的结果为（2.14）格式。因此将结果左移一位，以使高字节的格式与输入的相一致。FMUL 指令的执行周期与 MUL 指令相同，但比 MUL 指令增加了左移操作。

被乘数 Rd 和乘数 Rr 是两个包含无符号定点小数的寄存器，小数点固定在第 7 位和第 6 位之间。结果为 16 位无符号定点小数，其小数点固定在第 15 位和第 14 位之间。

5．有符号定点小数乘法

FMULS Rd，Rr　　16≤d≤23，16≤r≤23

说明：该指令完成寄存器 Rd（8 位带符号数）和寄存器 Rr（8 位带符号数）的内容相乘操作，结果为 16 位的带符号数，并将结果左移一位后保存在 R1:R0 中，R1 为高 8 位，R0 为低 8 位。源操作数为寄存器 R16～R23。

操作：R1:R0=Rd×Rr　（signed（1.15）=sigened（1.7）×sigened（1.7））

PC←PC+1　　机器码：0000 0011 1ddd 0rrr

对标志位的影响：Z C

注：（n.q）表示一个小数点左边有 n 个二进制数位、小数点右边有 q 个二进制数位的小数。以（n1.q1）和（n2.q2）为格式的两个小数相乘，产生格式为（(n1+n2)．(q1+q2)）的结果。

对于要有效保留小数位的处理应用，输入的数据通常采用（1.7）的格式，产生的结果为（2.14）格式。因此将结果左移一位，以使高字节的格式与输入的相一致。FMULS 指令的执行周期与 MULS 指令相同，但比 MULS 指令增加了左移操作。

被乘数 Rd 和乘数 Rr 是两个包含带符号定点小数的寄存器，小数点固定在第 7 位和第 6 位之间。结果为 16 位带符号的定点小数，其小数点固定在第 15 位和第 14 位之间。

6．有符号定点小数和无符号定点小数乘法

FMULSU Rd，Rr　　16≤d≤23，16≤r≤23

说明：该指令完成寄存器 Rd（8 位带符号数）和寄存器 Rr（8 位无符号数）的内容相乘操作，结果为 16 位的带符号数，并将结果左移一位后保存在 R1:R0 中，R1 为高 8 位，R0 为低 8 位。源操作数为寄存器 R16～R23。

操作：R1:R0=Rd×Rr　（signed（1.15）=sigened（1.7）×unsigened（1.7））

PC←PC+1　　机器码：0000 0011 1ddd 1rrr

对标志位的影响：Z C

注：（n.q）表示一个小数点左边有 n 个二进制数位、小数点右边有 q 个二进制数位的小数。以（n1.q1）和（n2.q2）为格式的两个小数相乘，产生格式为（(n1+n2)．(q1+q2)）的结果。对于要有效保留小数位的处理应用，输入的数据通常采用（1.7）的格式，产生的结果为（2.14）格式。因此将结果左移一位，以使高字节的格式与输入的相一致。FMULSU 指令的执行周期与 MULSU 指令相同，但比 MULSU 指令增加了左移操作。

被乘数 Rd 为一个包含带符号定点小数的寄存器，乘数 Rr 是一个包含无符号定点小数的寄存器，小数点固定在第 7 位和第 6 位之间。结果为 16 位带符号的定点小数，其小数点固定在第 15 位和第 14 位之间。

3.3 跳 转 指 令

3.3.1 无条件跳转指令

1．相对跳转

RJMP k　　-2048≤k≤2047

说明：相对跳转到 PC-2048～PC+2047 字范围内的地址，在汇编程序中，用目的地址的标号替代相对跳转字 k。

操作：PC←（PC+1）+ k　　机器码：1100 kkkk kkkk kkkk

对标志位的影响：无

汇编语言中，只要使用欲转向的标号即可。

例：

```
        RJMP ABC
             ……
             ……
ABC:         ……
```

2．间接跳转

IJMP

说明：间接跳转到 Z 指针寄存器指向的 16 位地址。Z 指针寄存器是 16 位宽，允许在当前程序存储器空间 64K 字（128K 字节）内跳转。

IJMP 间接跳转优点：跳转范围大；缺点：作为子程序模块，移植时需修改跳转地址，所以一般在子程序中不要使用。

操作：PC←Z（15～0）　　机器码：1001 0100 0000 1001

对标志位的影响：无

3．直接跳转

JMP k　　0≤k≤4194303

说明：直接跳转到 k 地址处，在汇编程序中，用目的地址的标号替代跳转字 k。

操作：PC←k　　机器码：1001 010k kkkk 110k kkkk kkkk kkkk kkkk

对标志位的影响：无

汇编语言中，只要使用欲转向的标号即可。

例：

```
    JMP ABC
    ……
ABC：……
```

3.3.2 条件跳转指令

条件跳转指令是依照某种特定的条件跳转的指令，条件满足则跳转；条件不满足时则顺序执行下面的指令。

1．测试条件符合跳转指令

（1）状态寄存器中位为“1”跳转

BRBS s,k　　0≤s≤7，-64≤k≤63

说明：执行该指令时，PC 先加 1，再测试 SREG 的 s 位，如果该位被置位，则跳转 k 个字，k 为 7 位带符号数，最多可向前跳 63 个字，向后跳 64 个字；否则顺序执行。在汇编程序中，用目的地址的标号替代相对跳转字 k。

操作：If SREG（s）=1，then PC←（PC+1）+k，else PC←PC+1

机器码：1111 00kk kkkk ksss

对标志位的影响：无

（2）状态寄存器中位为“0”跳转

BRBC s,k　　0≤s≤7，-64≤k≤63

说明：执行该指令时，PC 先加 1，再测试 SREG 的 s 位，如果该位被清零，则跳转 k 个字，k 为 7 位带符号数，最多可向前跳 63 个字，向后跳 64 个字；否则顺序执行。在汇编程序中，用目的地址的标号替代相对跳转字 k。

操作：If SREG（s）=0，then PC←（PC+1）+k，else PC←PC+1

机器码：1111 01kk kkkk ksss

对标志位的影响：无

（3）相等跳转

BREQ k　　-64≤k≤63

说明：条件相对跳转，测试零标志位 Z，如果 Z 位被置位，则相对 PC 值跳转 k 个字。如果在执行 CP、CPI、SUB 或 SUBI 指令后，立即执行该指令，且当寄存器 Rd 中数与寄存器 Rr 中数相等时，将发生跳转。这条指令相当于指令 BRBS 1，k。

操作：If Rd=Rr（Z=1），then PC←（PC+1）+k；else PC←PC+1

机器码：1111 00kk kkkk k001

对标志位的影响：无

（4）不相等跳转

BRNE k　　-64≤k≤63

说明：条件相对跳转，测试零标志位 Z，如果 Z 位被清零，则相对 PC 值跳转 k 个字。

这条指令相当于指令 BRBC 1，k

操作：If Rd≠Rr（Z=0），then PC←（PC+1）+k；else PC←PC+1

机器码：1111 01kk kkkk k001

对标志位的影响：无

（5）进位标志位 C 为“1”跳转

BRCS k　　-64≤k≤63

说明：条件相对跳转，测试进位标志 C，如果 C 位被置位，则相对 PC 值跳转 k 个字。这条指令相当于指令 BRBS 0，k。

操作：If C=1 then PC←（PC+1）+k ；else PC←PC+1

机器码：1111 00kk kkkk k000

对标志位的影响：无

（6）进位标志位 C 为“0”跳转

BRCC k　　-64≤k≤63

说明：条件相对跳转，测试进位标志 C，如果 C 位被清除，则相对 PC 值跳转 k 个字。这条指令相当于指令 BRBC 0，k。

操作：If C=0 then PC←（PC+1）+k　else PC←PC+1

机器码：1111 01kk kkkk k000

对标志位的影响：无

（7）大于或等于跳转（对无符号数）

BRSH k　　-64≤k≤63

说明：条件相对跳转，测试进位标志 C，如果 C 位被清零，则相对 PC 值跳转 k 个字。如果在执行 CP、CPI、SUB 或 SUBI 指令后，立即执行该指令，且当在寄存器 Rd 中无符号二进制数大于或等于寄存器 Rr 中无符号二进制数时,将发生跳转。该指令相当于指令 BRBS 0，k。

操作：If Rd≥Rr（C=0） then PC←（PC+1）+k else PC←PC+1

机器码：1111 01kk kkkk k000

对标志位的影响：无

（8）小于跳转（对无符号数）

BRLO k　　-64≤k≤63

说明：条件相对跳转，测试进位标志 C，如果 C 位被置位，则相对 PC 值跳转 k 个字。如果在执行 CP、CPI、SUB 或 SUBI 指令后立即执行该指令，且当在寄存器 Rd 中无符号二进制数小于在寄存器 Rr 中无符号二进制数时，将发生跳转。该指令相当于指令 BRBS 0，k。

操作：If Rd<Rr（C=1） then PC←（PC+1）+k　else PC←PC+1

机器码：1111 00kk kkkk k000

对标志位的影响：无

（9）结果为负跳转

BRMI k　　-64≤k≤63

说明：条件相对跳转，测试负号标志 N，如果 N 位被置位，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBS 2，k。

操作：If N=1 then　PC←（PC+1）+k　else PC←PC+1

机器码：1111 00kk kkkk k010

对标志位的影响：无

（10）结果为正跳转

BRPL k　　-64≤k≤63

说明：条件相对跳转，测试负号标志 N，如果 N 位被清零，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBC 2，k

操作：If N=0 then PC←（PC+1）+k； else PC←PC+1

机器码：1111 01kk kkkk k010

对标志位的影响：无

（11）大于或等于跳转（带符号数）

BRGE k　　-64≤k≤63

说明：条件相对跳转，测试符号标志 S，如果 S 位被清零，则相对 PC 值跳转 k 个字。如果在执行 CP、CPI、SUB 或 SUBI 指令后立即执行该指令，且当在寄存器 Rd 中带符号二进制数大于或等于寄存器 Rr 中带符号二进制数时,将发生跳转。该指令相当于指令 BRBC 4，k。

操作：If Rd≥Rr （N⊕V=0） then PC←（PC+1）+k，else PC←PC+1

机器码：1111 01kk kkkk k100

对标志位的影响：无

（12）小于跳转（带符号数）

BRLT k　　-64≤k≤63

说明：条件相对跳转，测试符号标志 S，如果 S 位被置位，则相对 PC 值跳转 k 个字。如果在执行 CP、CPI、SUB 或 SUBI 指令后立即执行该指令，且当在寄存器 Rd 中带符号二进制数小于在寄存器 Rr 中带符号二进制数时，将发生跳转。该指令相当于指令 BRBS 4，k。

操作：If Rd＜Rf （N⊕V＝1） then PC←（PC＋1）＋k else PC←PC＋1

机器码：1111 00kk kkkk k100

对标志位的影响：无

（13）半进位标志为“1”跳转

BRHS k　　-64≤k≤63

说明：条件相对跳转，测试半进位标志 H，如果 H 位被置位，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBS 5，k。

操作：If H=1 then PC←（PC+1）+k，else PC←PC+1

机器码：1111 00kk kkkk k101

对标志位的影响：无

（14）半进位标志为“0”跳转

BRHC k　　-64≤k≤63

说明：条件相对跳转，测试半进位标志 H，如果 H 位被清零，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBC 5，k。

操作：If H=0 then PC←（PC+1）+k， else PC←PC+1

机器码：1111 01kk kkkk k101

对标志位的影响：无

（15）T 标志为“1”跳转

BRTS k　　-64≤k≤63

说明：条件相对跳转，测试标志位 T，如果标志位 T 被置位，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBS 6，k。

操作：If T=1 then PC←（PC+1）+k else PC←PC+1

机器码：1111 00kk kkkk k110

对标志位的影响：无

（16）T 标志为“0”跳转

BRTC k　　-64≤k≤63

说明：条件相对跳转，测试 T 标志位，如果标志位 T 被清零，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBC 6，k。

操作：If T=0 then PC←（PC+1）+k　else PC←PC+1

机器码：1111 01kk kkkk k110

对标志位的影响：无

（17）溢出标志为“1”跳转

BRVS k　　-64≤k≤63

说明：条件相对跳转，测试溢出标志 V，如果 V 位被置位，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBS 3，k。

操作：If V=1 then PC←（PC+1）+k，else PC←PC+1

机器码：1111 00kk kkkk k011

对标志位的影响：无

（18）溢出标志为“0”跳转

BRVC k　　-64≤k≤63

说明：条件相对跳转，测试溢出标志 V，如果 V 位被清零，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBC 3，k。

操作：If V=0 then PC←（PC+1）+k，else PC←PC+1

机器码：1111 01kk kkkk k011

对标志位的影响：无

（19）中断标志为“1”跳转

BRIE k　　-64≤k≤63

说明：条件相对跳转，测试全局中断允许标志 I，如果 I 位被置位，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBS 7，k。

操作：If I=1 then PC←（PC+1）+k，else PC←PC+1

机器码：1111 00kk kkkk k111

对标志位的影响：无

（20）中断标志为“0”跳转

BRID k　　-64≤k≤63

说明：条件相对跳转，测试全局中断允许标志 I，如果 I 位被清零，则相对 PC 值跳转 k 个字。该指令相当于指令 BRBC 7，k。

操作：If I=0 then PC←（PC+1）+k, else PC←PC+1

机器码：1111 01kk kkkk k111

对标志位的影响：无

2. 测试条件符合跳行跳转指令

（1）相等跳行

CPSE Rd，Rr　　0≤d≤31，　0≤r≤31

说明：该指令完成两个寄存器 Rd 和 Rr 的比较，若 Rd=Rr，则跳一行执行指令。

操作：If Rd=Rr then PC←PC+2（or 3）　else PC←PC+1

机器码：0001 00rd dddd rrrr

对标志位的影响：无

（2）寄存器位为“0”跳行

SBRC Rr，b　　0≤r≤31，　0≤b≤7

说明：该指令测试寄存器第 b 位，如果该位被清零，则跳一行执行指令。

操作：If Rd（b）=0 then　PC←PC+2（or 3）　else PC←PC+1

机器码：1111 110r rrrr 0bbb

对标志位的影响：无

（3）寄存器位为“1”跳行

SBRS Rr，b　　0≤r≤31，　0≤b≤7

说明：该指令测试寄存器第 b 位，如果该位被置位，则跳下一行执行指令。

操作：If Rr（b）=1 then PC←PC+2（or 3），else PC←PC+1

机器码：1111 111r rrrr 0bbb

对标志位的影响：无

（4）I/O 寄存器位为“0”跳行

SBIC P，b　　0≤P≤31，　0≤b≤7

说明：该指令测试 I/O 寄存器第 b 位，如果该位被清零，则跳一行执行指令。该指令只在低 32 个 I/O 寄存器内操作，地址为 I/O 寄存器空间的 0～31。

操作：If P（b）=0 then PC←PC+2（or 3），else PC←PC+1

机器码：1001 1001 PPPP Pbbb

对标志位的影响：无

（5）I/O 寄存器位为“1”跳行

SBIS P，b　　0≤P≤31，0≤b≤7

说明：该指令测试 I/O 寄存器第 b 位，如果该位被置位，则跳一行执行指令。该指令只在低 32 个 I/O 寄存器内操作，地址为 I/O 寄存器空间的 0～31。

操作：If P（b）=1 then PC←PC+2（or3），else PC←PC+1

机器码：1001 1011 PPPP Pbbb

对标志位的影响：无

3.3.3　子程序调用和返回指令

在程序设计中通常把具有一定功能模块的公用程序段定义为子程序。为了实观调用子程序的功能，指令系统中都有调用子程序指令。调用子程序指令与跳转指令的区别如下：执行调用子程序时把下一条指令地址 PC 值保留到堆栈中，即断点保护，然后把子程序的起始地址置入 PC，子程序执行完毕返回时，将断点由堆栈中弹出到 PC，然后从断点处继续执行原程序；而跳转指令既不保护断点也不返回原程序。在每个子程序中都必须有返回指令，返回指令的功能就是把调用前压入堆栈的断点弹出置入 PC，恢复执行调用子程序前的原程序。

在一个程序中，子程序中还会调用别的子程序，这称为子程序嵌套。每次调用子程序时必须将下条指令地址保存起来，返回时，按后进先出原则依次取出相应的 PC 值。堆栈就是按后进先出规则存取数据的，调用指令和返回指令具有自动保存和恢复 PC 内容的功能，即自动进栈，自动出栈。

1．相对调用

RCALL k　　-2048≤k≤2047

说明：将 PC+1 后的值（RCALL 指令后的下一条指令地址）压入堆栈，然后调用在当前

PC 前或后 k+1 处地址的子程序。

操作：STACK←PC+1，SP←SP-2，PC←（PC+1）+k 机器码：1101 kkkk kkkk kkkk

对标志位的影响：无

2．间接调用

ICALL

说明：间接调用由 Z 寄存器中（16 位指针寄存器）指向的子程序。地址指针寄存器 Z 为 16 位，允许调用在当前程序存储空间 64K 字（128K 字节）内的子程序。

操作：STACK←PC+1，SP←SP-2，PC←Z　　机器码：1001 0101 0000 1001

对标志位的影响：无

3．直接调用

CALL k　　0≤k≤65535

说明：将 PC+2 后的值（CALL 指令后的下一条指令地址）压入堆栈，然后直接调用 k 处地址的子程序。

操作：STACK←PC+2，SP←SP-2，PC←k

机器码：1001 010k kkkk 111k kkkk kkkk kkkk kkkk

对标志位的影响：无

4．从子程序返回

RET

说明：从子程序返回，返回地址从堆栈中弹出。

操作：SP←SP+2，PC←STACK　　机器码：1001 0101 0000 1000

对标志位的影响：无

5．从中断程序返回

RETI

说明：从中断程序中返回，返回地址从堆栈中弹出，且全局中断标志被置位。

操作：SP←SP+2，PC←STACK　　机器码：1001 0101 0001 1000

对标志位的影响：I（1）

注意：（1）主程序应跳过中断向量区，防止给修改补充中断程序带来麻烦。

（2）应在中断向量区中不用的中断入口地址写上 RETI——中断返回指令，有抗干扰作用。

3.4 数据传送指令

数据传送指令是在编程中使用最频繁的一类指令，数据传送指令是否灵活快速对程序的执行速度产生很大影响。数据传送指令执行操作是寄存器与寄存器、寄存器与数据存储器 SRAM、寄存器与 I/O 端口之间的数据传送，另外还有从程序存储器直接取数指令 LPM（ELPM）以及 PUSH 压栈和 POP 出栈的堆栈指令。

所有的传送指令的操作对标志位均无影响。

3.4.1 直接寻址数据传送指令

1．工作寄存器间传送数据

MOV Rd，Rr　　0≤d≤31，0≤r≤31

说明：该指令将一个寄存器内容传送到另一个寄存器中，源寄存器 Rr 的内容不改变，而目的寄存器 Rd 复制了 Rr 的内容。

操作：Rd←Rr　　PC←PC+1　　机器码：0010 11rd dddd rrrr

2．SRAM 数据直接送寄存器

LDS Rd，k　　0≤d≤31，　0≤k≤65535

说明：把 SRAM 中一个存储单元的内容（字节）装入到寄存器，其中 k 为该存储单元的 16 位地址。

操作：Rd←（k）　　PC←PC+2　　机器码：1001 000d dddd 0000 kkkk kkkk kkkk kkkk

3．寄存器数据直接送 SRAM

STS k，Rr　　0≤r≤31，　0≤k≤65535

说明：将寄存器的内容直接存储到 SRAM 中，其中 k 为存储单元的 16 位地址。

操作：（k）←Rr　　PC←PC+1　　机器码：1001 001d dddd 0000 kkkk kkkk kkkk kkkk

4．立即数送寄存器

LDI Rd，K　　16≤d≤31，　0≤K≤255

说明：装入一个 8 位立即数到寄存器 R16～R31 中。

操作：Rd←K　　PC←PC+1　　机器码：1110 KKKK dddd KKKK

3.4.2　间接寻址数据传送指令

1．使用 X 指针寄存器间接寻址传送数据

（1）使用地址指针寄存器 X 间接寻址将 SRAM 内容装入到指定寄存器

①LD Rd，X　　0≤d≤31；将指针为 X 的 SRAM 中的数送寄存器，指针不变。

操作：Rd←（X）　　PC←PC+1　　机器码：1001 000d dddd 1100

②LD Rd，X+　　0≤d≤31；先将指针为 X 的 SRAM 中的数送寄存器，X 指针加 1。

操作：Rd←（X），X←X+1 PC←PC+1　　机器码：1001 000d dddd 1101

③LD Rd，-X　0≤d≤31；X 指针减 1，将指针为 X 的 SRAM 中的数送寄存器。

操作：X←X-1，Rd←（X）　PC←PC+1　　机器码：1001 000d dddd 1110

（2）使用地址指针寄存器 X 间接寻址将寄存器内容存储到 SRAM

①ST X，Rr　　0≤d≤31；将寄存器内容送 X 为指针的 SRAM 中，X 指针不改变。

操作：（X）←Rr　　PC←PC+1　　机器码：1001 001r rrrr 1100

②ST X，Rr　　0≤d≤31；先将寄存器内容送 X 为指针的 SRAM 中，后 X 指针加 1。

操作：（X）←Rr，X←X+1　PC←PC+1　　机器码：1001 001r rrrr 1101

③ST -X，Rr　　0≤d≤31；先将 X 指针减 1，然后将寄存器内容送 X 为指针的 SRAM 中。

操作：X←X-1，（X）←Rr　PC←PC+1　　机器码：1001 001r rrrr 1110

2．使用 Y 指针寄存器间接寻址传送数据

（1）使用地址指针寄存器 Y 间接寻址将 SRAM 中的内容装入寄存器

①LD Rd，Y　　0≤d≤31；将指针为 Y 的 SRAM 中的数送寄存器，Y 指针不变。

操作：Rd←（Y）　　PC←PC+1　　机器码：1000 000d dddd 1000

②LD Rd，Y+　　0≤d≤31；先将指针为 Y 的 SRAM 中的数送寄存器，然后 Y 指针加 1。

操作：Rd←（Y），Y←Y+1　PC←PC+1　　机器码：1001 000d dddd 1001

③LD Rd，-Y　　0≤d≤31；先将 Y 指针减 1，将指针为 Y 的 SRAM 中的数送寄存器。

操作：Y←Y-1，Rd←（Y）　PC←PC+1　　机器码：1001 000d dddd 1010

④LDD Rd，Y+q　　0≤d≤31，0≤q≤63；将指针为 Y+q 的 SRAM 中的数送寄存器，而 Y 指针不改变。

操作：Rd←（Y+q）　　PC←PC+1　　机器码：10q0 qq0d dddd 1qqq

（2）使用地址指针寄存器 Y 间接寻址将寄存器内容存储到 SRAM

①ST Y，Rr　　0≤d≤31；将寄存器内容送 Y 为指针的 SRAM 中，Y 指针不改变。

操作：（Y）←Rr　　PC←PC+1　　机器码：1000 001r rrrr 1000

②ST Y+，Rr　　0≤d≤31；先将寄存器内容送 Y 为指针的 SRAM 中，然后 Y 指针加 1。

操作：（Y）←Rr，Y←Y+1　PC←PC+1　　机器码：1001 001r rrrr 1001

③ST －Y，Rr　　0≤d≤31；先将 Y 指针减 1，然后将寄存器内容送 Y 为指针的 SRAM 中。

操作：Y←Y-1，（Y）←Rr　PC←PC+1　　机器码：1001 001r rrrr 1010

④STD Y+q，Rr　　0≤d≤31，0≤q≤63；将寄存器内容送 Y+q 为指针的 SRAM 中。

操作：（Y+q）←Rr　　PC←PC+1　　机器码：10q0 qq1r rrrr 1qqqq

3．使用 Z 指针寄存器间接寻址传送数据

（1）使用地址指针寄存器 Z 间接寻址将 SRAM 中的内容装入到指定寄存器

①LD Rd，Z　　0≤d≤31；将指针为 Z 的 SRAM 中的数送寄存器，Z 指针不变。

操作：Rd←（Z）　　PC←PC+1　　机器码：1000 000d dddd 0000

②LD Rd，Z+　　0≤d≤31；先将指针为 Z 的 SRAM 中的数送寄存器，然后 Z 指针加 1。

操作：Rd←（Z），Z←Z+1　PC←PC+1　　机器码：1001 000d dddd 0001

③LD Rd，-Z　　0≤d≤31；先将 Z 指针减 1，然后将指针为 Z 的 SRAM 中的数送寄存器。

操作：Z←Z-1，Rd←（Z）　PC←PC+1　　机器码：1001 000d dddd 0010

④LDD Rd，Z+q　　0≤d≤31，0≤q≤63；将指针为 Z+q 的 SRAM 中的数送寄存器，而 Z 指针不改变。

操作：Rd←（Z+q）　　PC←PC+1　　机器码：10q0 qq0d dddd 0qqq

（2）使用地址指针寄存器 Z 间接寻址将寄存器内容存储到 SRAM

①ST Z，Rr　　0≤d≤31；将寄存器内容送 Z 为指针的 SRAM 中，Z 指针不改变。

操作：（Z）←Rr　　PC←PC+1　　机器码：1000 001r rrrr 0000

②ST Z+，Rr　　0≤d≤31；先将寄存器内容送 Z 为指针的 SRAM 中，然后 Z 指针加 1。

操作：（Z）←Rr，Z←Z+1　　PC←PC+1　　机器码：1001 001r rrrr 0001

③ST -Z，Rr　　0≤d≤31；先将 Z 指针减 1，然后将寄存器内容送 Z 为指针的 SRAM 中。

操作：Z←Z-1，（Z）←Rr　PC←PC+1　　机器码：1001 001r rrrr 0010

④STD Z+q，Rr　　0≤d≤31，0≤q≤63；将寄存器内容送 Z+q 为指针的 SRAM 中。

操作：（Z+q）←Rr　　PC←PC+1　　机器码：10q0 qq1r rrrr 0qqq

以上 22 条指令操作之后，X、Y、Z 指针寄存器要么不改变，要么是加 1 或减 1。使用 X、Y、Z 指针寄存器的这些特性特别适用于访问矩阵表和堆栈指针等。

3.4.3　从程序存储器中取数装入寄存器指令

1．从程序存储器中取数装入寄存器 R0

LPM

说明：将 Z 指向的程序存储器空间的一个字节装入寄存器 R0。

操作：R0←（Z）　　PC←PC+1　　机器码：1001 0101 1100 1000

注释：由于程序存储器的地址是以字（双字节）为单位的，因此，16 位地址指针寄存器 Z 的高 15 位为程序存储器的字地址，最低位 LSB 为“0”时，指字的低字节；为“1”时，指字的高字节。该指令能寻址程序存储器空间为一个 64K 字节的页（32k 字）。

2．从程序存储器中取数装入寄存器

LPM Rd，Z　0≤d≤31

说明：将 Z 指向的程序存储器空间的一个字节装入寄存器 Rd。

操作：Rd←（Z）　　PC←PC+1　　机器码：1001 000d dddd 0100

注释：由于程序存储器的地址是以字（双字节）为单位的，因此，16 位地址指针寄存器 Z 的高 15 位为程序存储器的字地址，最低位 LSB 为“0”时，指字的低字节；为“1”时，指字的高字节。该指令能寻址程序存储器空间为一个 64K 字节的页（32k 字）。

3．带后增量的从程序存储器中取数装入寄存器 Rd

LPM Rd，Z+

说明：将 Z 指向的程序存储器空间的一个字节装入 Rd，然后 Z 指针加 1。

操作：Rd←（Z），Z←Z+1　PC←PC+1　　机器码：1001 000d dddd 0101

注释：由于程序存储器的地址是以字（双字节）为单位的，因此，16 位地址指针寄存器 Z 的高 15 位为程序存储器的字地址，最低位 LSB 为“0”时，指字的低字节；为“1”时，指字的高字节。该指令能寻址程序存储器空间为一个 64K 字节的页（32k 字）。

4．从程序存储器中取数装入寄存器 R0(扩展)

ELPM

说明：将 RAMPZ:Z 指向的程序存储器空间的一个字节装入寄存器 R0。

操作：R0←（RAMPZ:Z）　　PC←PC+1　　机器码：1001 0101 1101 1000

注释：由于程序存储器的地址是以字（双字节）为单位的，因此，PAMPZ 寄存器的最低位，加上 16 位地址指针寄存器 Z 的高 15 位，组成 16 位的程序存储器字寻址地址，而 Z 寄存器的最低位 LSB 为“0”时，指字的低字节；为“1”时，指字的高字节。该指令能寻址整个 128K 字节（64K 字）的程序存储器空间。

5．从程序存储器中取数装入寄存器（扩展）

ELPM Rd，Z　0≤d≤31

说明：将 RAMPZ:Z 指向的程序存储器空间的一个字节装入寄存器 Rd。

操作：Rd←（RAMPZ:Z）　　PC←PC+1　　机器码：1001 000d dddd 0110

注释：由于程序存储器的地址是以字（双字节）为单位的，因此，PAMPZ 寄存器的最低位，加上 16 位地址指针寄存器 Z 的高 15 位，组成 16 位的程序存储器字寻址地址，而 Z 寄存器的最低位 LSB 为“0”时，指字的低字节；为“1”时，指字的高字节。该指令能寻址整个 128K 字节（64K 字）的程序存储器空间。

6．带后增量的从程序存储器中取数装入寄存器 Rd（扩展）

LPM Rd，Z+

说明：将 RAMPZ:Z 指向的程序存储器空间的一个字节装入 Rd，然后 RAMPZ:Z 指针加 1。

操作：Rd←（RAMPZ:Z），　RAMPZ:Z←RAMPZ:Z+1　PC←PC+1

机器码：1001 000d dddd 0111

注释：由于程序存储器的地址是以字（双字节）为单位的，因此，PAMPZ 寄存器的最低位，加上 16 位地址指针寄存器 Z 的高 15 位，组成 16 位的程序存储器字寻址地址，而 Z

寄存器的最低位 LSB 为“0”时，指字的低字节；为“1”时，指字的高字节。该指令能寻址整个 128K 字节（64K 字）的程序存储器空间。

3.4.4 写程序存储器指令

SPM

说明：将寄存器对 R1:R0 的内容（16 位字）写入 Z 指向的程序存储器空间。

操作：(Z) ←R1:R0　　PC←PC+1　　机器码：1001 0101 1110 1000

注释：该指令用于具有在应用自编程性能的 AVR 单片机，应用这一特性，单片机系统程序可以在运行中更改程序存储器中的程序，实现动态修改系统程序的功能。由于程序存储器的地址是以字（双字节）为单位的，因此，寄存器 R1:R0 的内容组成一个 16 位的字，其中 R1 为字的高位字节，R0 为字的低位字节。该指令能寻址程序存储器空间为一个 64K 字节的页（32k 字）。具体应用见相关章节的介绍。

3.4.5 I/O口数据传送

1．I/O 口数据装入寄存器

IN Rd，P　　0≤d≤31，　0≤P≤63

说明：将 I/O 空间（口、定时器、配置寄存器等）的数据传送到寄存器区中的寄存器 Rd 中。

操作：Rd←P　　PC←PC+1　　机器码：1011 0PPd dddd PPPP

2．寄存器数据送 I/O 口

OUT P，Rr　　0≤r≤31，　0≤P≤63

说明：将寄存器区中 Rr 的数据传送到 I/O 空间（口、定时器、配置寄存器等）。

操作：P←Rr　　PC←PC+1　　机器码：1011 1PPr rrrr PPPP

3.4.6 堆栈操作指令

AVR 单片机的特殊功能寄存器中有一个 16 位的堆栈指针寄存器 SP，它指出栈顶的位置，在指令系统中有两条用于数据传送的栈操作指令。

1．进栈指令

PUSH Rr　　0≤d≤31

说明：该指令存储寄存器 Rr 的内容到堆栈。

操作 STACK←Rr，SP←SP-1　　PC←PC+1　　机器码：1001 001d dddd 1111

2．出栈指令

POP Rd　　0≤d≤31

说明：该指令将堆栈中的字节装入到寄存器 Rd 中。

操作：SP←SP+1，Rd←STACK　　PC←PC+1　　机器码：1001 000d dddd 1111

3.5 位操作和位测试指令

AVR 单片机指令系统中有四分之一的指令为位操作和位测试指令，这些指令的灵活应用极大地提高了系统的逻辑控制和处理能力。

3.5.1 带进位逻辑操作指令

1．寄存器逻辑左移

LSL Rd　　0≤d≤31

说明：寄存器 Rd 中所有位左移 1 位，第 0 位被清零，第 7 位移到 SREG 中的 C 标志。该指令完成一个无符号数乘 2 的操作。

操作：C←$b_7b_6b_5b_4b_3b_2b_1b_0$←0　　PC←PC+1　　机器码：0000 11dd dddd dddd

对标志位的影响：H S V N Z C

2．寄存器逻辑右移

LSR Rd　　0≤d≤31

说明：寄存器 Rd 中所有位右移 1 位，第 7 位被清零，第 0 位移到 SREG 中的 C 标志。该指令完成一个无符号数除 2 的操作，C 标志被用于结果舍入。

操作：0→$b_7b_6b_5b_4b_3b_2b_1b_0$→C　　PC←PC+1　　机器码：1001 010d dddd 0110

对标志位的影响：S V N（0） Z C

3．带进位位的寄存器逻辑循环左移

ROL Rd　　0≤d≤31

说明：寄存器 Rd 的所有位左移 1 位，C 标志被移到 Rd 的第 0 位，Rd 的第 7 位移到 C 标志。

操作：C←$b_7b_6b_5b_4b_3b_2b_1b_0$←C　　PC←PC+1　　机器码：0001 11dd dddd dddd（参见 ADC 指令）

对标志位的影响：H S V N Z C

4．带进位位的寄存器逻辑循环右移

ROR Rd　　0≤d≤31

说明：寄存器 Rd 的所有位右移 1 位，C 标志被移到 Rd 的第 7 位，Rd 的第 0 位移到 C 标志。

操作：C→$b_7b_6b_5b_4b_3b_2b_1b_0$→C　　PC←PC+1　　机器码：1001 010d dddd 0111

对标志位的影响：S V N Z C

5．寄存器算术右移

ASR Rd　　0≤d≤31

说明：寄存器 Rd 中的所有位右移 1 位，而第 7 位保持原逻辑值，第 0 位被装入 SREG 的 C 标志位。这个操作实现 2 的补码值除 2，而不改变符号，进位标志用于结果的舍入。

操作：b_7→$b_7b_6b_5b_4b_3b_2b_1b_0$→C　　PC←PC+1　　机器码：1001 010d dddd 0101

对标志位的影响：S V N Z C

6．寄存器半字节交换

SWAP Rd　　0≤d≤31

说明：寄存器中的高半字节和低半字节交换。

操作：$b_7b_6b_5b_4$ ←→ $b_3b_2b_1b_0$ PC←PC+1　　机器码：1001 010d dddd 0010

对标志位的影响：无

3.5.2 位变量传送指令

1．寄存器中的位存储到 SREG 中的 T 标志

BST Rr，b　　0≤d≤31，0≤b≤7

说明：把寄存器 Rr 中的位 b 存储到 SREG 状态寄存器中的 T 标志位中。

操作：T←Rr（b）　　PC←PC+1　　机器码：1111 101d dddd 0bbb

对标志位的影响：T

2．SREG 中的 T 标志位值装入寄存器 Rd 中的某一位

BLD Rd，d　　0≤d≤31，0≤b≤7

说明：复制 SREG 状态寄存器的 T 标志到寄存器 Rd 中的位 b。

操作：Rd（b）←T　　PC←PC+1　　机器码：1111 100d dddd 0bbb

对标志位的影响：无

3.5.3 位变量修改指令

1．状态寄存器 SREG 的指定位置“1”

BSET s　　0≤s≤7

说明：置“1”状态寄存器 SREG 的某一标志位。

操作：SREG（s）←1　　PC←PC+1　　机器码：1001 0100 0sss 1000

对标志位的影响：I T H S V N Z C

2．状态寄存器 SREG 的指定位清“0”

BCLR s　　0≤s≤7

说明：清零 SREG 状态寄存器 SREG 中的一个标志位。

操作：SREG（s）←0　　PC←PC+1　　机器码：1001 0100 1sss 1000

对标志位的影响：I T H S V N Z C

3．I/O 寄存器的指定位置“1”

SBI P，b　　0≤P≤31，　0≤b≤7

说明：对 P 指定的 I/O 寄存器的指定位置“1”。该指令只在低 32 个 I/O 寄存器内操作，I/O 寄存器地址为 0～31。

操作：I/O（P，b）←1　　PC←PC+1　　机器码：1001 1010 PPPP Pbbb

对标志位的影响：无

4．I/O 寄存器的指定位置“0”

CBI P，b　　0≤P≤31，0≤b≤7

说明：清零指定 I/O 寄存器中的指定位。该指令只用在低 32 个 I/O 寄存器上操作，I/O 寄存器地址为 0～31。

操作：I/O（P，b）←0　　PC←PC+1　　机器码：1001 1000 PPPP Pbbb

对标志位的影响：无

5．置进位位

SEC　　置位 SREG 状态寄存器中的进位标志 C

操作：C←1　　PC←PC+1　　机器码：1001 0100 0000 1000

对标志位的影响：C（1）

6．清进位位

CLC　　清零 SREG 状态寄存器中的进位标志 C

操作：C←0　　PC←PC+1　　机器码：1001 0100 1000 1000

对标志位的影响：C（0）

7．置位负标志位

SEN　　置位 SREG 状态寄存器中的负数标志 N

操作：N←1　　PC←PC+1　　机器码：1001 0100 0010 1000

对标志位的影响：N（1）

8．清负标志位

CLN 清零 SREG 状态寄存器中的负数标志 N

操作：N←0　　PC←PC+1　　机器码：1001 0100 1010 1000

对标志位的影响：N（0）

9．置零标志位

SEZ　　置位 SREG 状态寄存器中的零标志 Z

操作：Z←1　　PC←PC+1　　机器码：1001 0100 0001 1000

对标志位的影响：Z（1）

10．清零标志位

CLZ　　清零 SREG 状态寄存器中的零标志 Z

操作：Z←0　　PC←PC+1　　机器码：1001 0100 1001 1000

对标志位的影响：Z（0）

11．触发全局中断位

SEI　　置位 SREG 状态寄存器中的全局中断标志 I

操作：I←1　　PC←PC+1　　机器码：1001 0100 0111 1000

对标志位的影响：I（1）

12．禁止全局中断位

CLI　　清除 SREG 状态寄存器中的全局中断标志 I

操作：I←0　　PC←PC+1　　机器码：1001 0100 1111 1000

对标志位的影响：I（0）

13．置 S 标志位

SES　　置位 SREG 状态寄存器中的符号标志 S

操作：　S←1　　PC←PC+1　　机器码：1001 0100 0100 1000

对标志位的影响：S（1）

14．清 S 标志位

CLS　　清零 SREG 状态寄存器中的符号标志 S

操作：S←0　　PC←PC+1　　机器码：1001 0100 1100 1000

对标志位的影响：S（0）

15．置溢出标志位

SEV　　置位 SREG 状态寄存器中的溢出标志 V

操作：V←1　　PC←PC+1　　机器码：1001 0100 0011 1000

对标志位的影响：V（1）

16．清溢出标志位

CLV　　清零 SREG 状态寄存器中的溢出标志 V

操作：V←0　　PC←PC+1　　机器码：1001 0100 1011 1000

对标志位的影响：V（0）

17．置 T 标志位

SET　　置位 SREG 状态寄存器中的 T 标志

操作：T←1　　PC←PC+1　　机器码：1001 0100 0110 1000

对标志位的影响：T（1）

18．清 T 标志位

CLT　　清零 SREG 状态寄存器中的 T 标志

操作：T←0　　PC←PC+1　　机器码：1001 0100 1110 1000

对标志位的影响：T（0）

19．置半进位标志

SHE　　置位 SREG 状态寄存器中的半进位标志 H

操作：H←1　　PC←PC+1　　机器码：1001 0100 0101 1000

对标志位的影响：H（1）

20．清半进位标志

CLH 清零 SREG 状态寄存器中的半进位标志 H

操作：H←0　　PC←PC+1　　机器码：1001 0100 1101 1000

对标志位的影响：H（0）

3.6 MCU控制指令

MCU 控制指令有 3 条，主要用于控制 MCU 的运行方式以及清零看门狗定时器。

1．空操作指令

NOP

说明：该指令完成一个单周期空操作。

操作：无　　PC←PC+1　　机器码：0000 0000 0000 0000

对标志位的影响：无

应用：延时等待、产生方波。抗干扰处理：在空的程序存储器单元中写上空操作，空操作指令最后加一跳转指令，转到$000H。

2．进入休眠方式指令

SLEEP

说明：该指令使 MCU 进入休眠方式运行。休眠模式由 MCU 控制寄存器定义。当 MCU 在休眠状态下由一个中断被唤醒时，在中断程序执行后，紧跟在休眠指令后的指令将被　执行。

操作：PC←PC+1　　MCU 进入由 MCU 控制寄存器定义的休眠方式运行

机器码：1001 0101 1000 1000

应用：省电，尤其对绿色、便携式仪器特别有用。

对标志位的影响：无

3．清零看门狗计数器

WDR

说明：该指令清零看门狗定时器。在允许使用看门狗定时器情况下，系统程序在正常

运行中必须在 WD 预定比例器给出限定时间内执行一次该指令，以防止看门狗定时器溢出，造成系统复位。参见看门狗定时器硬件部分。

操作：清零看门狗定时器　　PC←PC+1　　机器码：1001 0101 1010 1000

对标志位的影响：无

应用：抗干扰、延时、提高系统的稳定性。

3.7 AVR汇编语言系统

用汇编语言编写的程序称为汇编语言程序，或称汇编源程序。显然汇编语言源程序比二进制的机器语言更容易学习和掌握。但是，单片机不能直接识别和执行汇编语言程序，因此需要使用一个专用的软件系统，将汇编语言的源程序“翻译”成二进制的机器语言程序——目标程序（执行代码）。这个专用软件系统就是汇编语言编译软件。

ATMEL 公司提供免费的 AVR 开发平台—AVR Studio 集成开发环境（IDE），其中就包括 AVR Assembler 汇编编译器（HTTP://www.atmel.com）。

用汇编语言编写程序必须按汇编工具软件所规定的语法和规则进行，否则会发生错误。在由汇编语言编译软件所构成的汇编语言系统中，除了允许用户使用规定的汇编指令，还定义了一些用于编写汇编程序的伪指令，以及使用表达式等，以方便用户编写汇编程序。

3.7.1 汇编语言语句格式

汇编语言源程序是由一系列汇编语句组成的。AVR 的汇编语句的标准格式有以下 4 种：

（1）[标号:] 伪指令 [操作数][;注释]

（2）[标号:] 指令 [操作数][;注释]

（3）[;注释]

（4）空行

注意，在汇编语句中是不使用中括号的，格式中的中括号内容表示其可以缺省。

- 标号

标号是语句地址的标记符号，用于引导对该语句的访问和定位。使用标号的目的是为了跳转和分支指令及在程序存储器、数据存储器 SRAM 以及 EEPROM 中定义变量名。有关标号的一些规定如下：

（1）标号一般由 ASCII 字符组成，第一个字符为字母；

（2）同一标号在一个独立的程序中只能定义一次；

（3）不能使用汇编语言中已定义的符号（保留字），如指令字、寄存器名、伪指令字等。

- 伪指令

在汇编语言程序中可以使用一些伪指令。伪指令不属于 CPU 指令集，编译时并不产生实际的目标机器操作代码，只是用于在汇编程序中对地址、寄存器、数据、常量等进行定义说明，以及对编译过程进行某种控制等。AVR 的指令系统不包括伪指令，伪指令通常由汇编编译系统给出。

- 指令

指令是汇编程序中主要的部分，汇编程序中使用 AVR 指令集中给出的全部指令。

- 操作数

操作数是指令操作时所需要的数据或地址。汇编程序完全支持指令系统所定义的操作数格式。但指令系统采用的操作数格式通常为数字形式，在编写程序时使用起来不太方便，因此，在编译器的支持下，可以使用多种形式的操作数，如数字、标识符、表达式等。

- 注释

注释部分仅用于对程序和语句进行说明，帮助程序设计人员阅读、理解和修改程序。只要有“;”符号，后面即为注释内容，注释内容长度不限，注释内容换行时，开头部分还要使用符号“;”。编译系统对注释内容不予理会，不产生任何代码。

- 其它字符

汇编语句中，“:”用于标号之后；空格用于指令字和操作数的分隔；指令有两个操作数时用“,”分隔两个操作数；“;”用于注释之前；“[ ]”中的内容表示可选项。

3.7.2 汇编器伪指令

AVR 的汇编器提供一些伪指令。伪指令并不直接转换生成操作执行代码。它只是用于指示编译生成目标程序的起始地址或常数表格的起始地址，或为工作寄存器定义符号名称，定义外设口地址，SRAM 工作区，规定编译器的工作内容等。因此伪指令有间接产生机器码、控制机器代码空间的分配、对编译器工作过程进行控制等。AVR 汇编系统使用的伪指令共 18 条，如表 3.6 中给出。

表 3-6 伪指令表

序 号	伪 指 令	说 明	序 号	伪 指 令	说 明
1	BYTE	在 RAM 中定义预留存储单元	10	EXIT	退出文件
2	CSEG	声明代码段	11	INCLUDE	包含指定的文件
3	DB	定义字节常数	12	MACRO	宏定义开始
4	DEF	定义寄存器符号名	13	ENDMACRO	宏定义结束
5	DEVICE	指定为何器件生成汇编代码	14	LISTMAC	列表宏表达式
6	DSEG	声明数据段	15	LIST	列表文件生成允许器
7	DW	定义字常数	16	NOLIST	关闭列表文件生成
8	EQU	定义标识符常量	17	ORG	设置程序起始位置
9	ESEG	声名 E^2PROM 段	18	SET	赋值给标识符

1．BYTE—为变量预留字节型存储单元

BYTE 伪指令是从指定的地址开始，在 SRAM 中预留若干字节的存储空间备用。备用存储空间以字节计算，个数由 BYTE 伪指令的参数或表达式的值确定。BYTE 伪指令前应使用一个标号，该标号既作为变量的名称，又作为符号地址，以标记备用存储空间在 SRAM 中的起始位置。该伪指令有一个参数，表示保留存储空间的字节数，但并不能给这些 RAM 单元赋值。BYTE 伪指令仅能用在数据段内（见伪指令 CSEG，DSEG 和 ESEG）。

语法：LABEL：.BYTE 表达式

示例：

```
.DSEG                       ;RAM 数据段(SRAM)
var1:  .BYTE 1              ;保留 1 个字节的存储单元，用 var1 标识
table: .BYTE tab_size       ;保留 tab_size 个字节的存储空间

.CSEG                       ;代码段开始(Flash)
```

```
ldi r30, low(var1)      ;将保留存储单元 var1 起始地址的低 8 位装入 Z
ldi r31, high(var1)     ;将保留存储单元 var1 起始地址的高 8 位装入 Z
ld r1, Z                ;将保留存储单元的内容读到寄存器 R1
```

2．CSEG—声明代码段（Flash）

CSEG 伪指令用于声明代码段的起始（在 Flash 中），既 CSEG 之后是要写入程序存储器中的指令代码、常数表格等。一个汇编程序可包含几个代码段，程序中缺省的段为代码段，这些代码段在编译过程中被连接成一个代码段。每个代码段内部都有自己的字定位计数器。可使用 ORG 伪指令定义该字定位计数器的初始值，作为代码段在程序存储器中的起始位置。CSEG 伪指令不带参数。在代码段中不能使用 BYTE 伪指令。

语法：.CSEG

3．DB—在程序存储器或 E^2PROM 存储器中定义字节常数

DB 伪指令是从程序存储器或 E^2PROM 存储器的某个地址单元开始，存入一组规定的 8 位二进制常数（字节常数）。DB 伪指令只能出现在代码段或 E^2PROM 段中。DB 伪指令前应使用一个标号，以标记所定义的字节常数区域的起始位置。DB 伪指令为一个表达式列表，表达式列表由多个表达式组成，但至少要含有一个表达式，表达式之间用逗号分隔。每个表达式值的范围必须在-128～255 之间。如果表达式的值是负数，则用 8 位 2 的补码表示，存入程序存储器或 E^2PROM 存储器中。如果 DB 伪指令用在代码段，并且表达式表中多于一个表达式，则以两个字节组合成一个字放在程序存储器中。如果表达式的个数是奇数，不管下一行汇编代码是否仍是 DB 伪指令，最后一个表达式的值将单独以字的格式放在程序存储器中。

语法：LABEL：.DB 表达式表

4．DEF—定义寄存器符号名

DEF 伪指令给寄存器定义一个替代的符号名。在后序程序中可以使用定义的符号名来表示被定义的寄存器。可以给一个寄存器定义多个符号名。符号名在后面程序中可以重新定义指定。编译时，凡遇到符号名都以相应被定义的寄存器替代。

语法：.DEF 符号名 = 寄存器

5．DEVICE—指定为何器件生成汇编代码

DEVICE 伪指令告知汇编器为何器件编译产生执行代码。如果在程序中使用该伪指令指定了器件型号，那么在编译过程中，若存在指定器件所不支持的指令，编译器则给出一个警告。如果代码段或 E^2PROM 段所使用的存储器空间大于指定器件本身所能提供的存储器容量，编译器也会给出警告。如果不使用 DEVICE 伪指令，则假定器件支持所有的指令，也不限制存储器容量的大小。

语法：.DEVICE Atmega16

6．DSEG—声明数据段（SRAM）

DSEG 伪指令声明数据段的起始。一个汇编程序文件可以包含几个数据段，这些数据段在汇编过程中被连接成一个数据段。在数据段中，通常是仅由 BYTE 伪指令（和标号）组成。每个数据段内部都有自己的字节定位计数器。可使用 ORG 伪指令定义该字节定位计数器的初始值，作为数据段在 SRAM 中的起始位置。DSEG 伪指令不带参数。

语法：.DSEG

7．DW—在程序存储器或 E²PROM 存储器中定义字常数

DW 伪指令是从程序存储器或 E²PROM 存储器的某个地址单元开始，存入一组规定的 16 位二进制常数（字常数）。DW 伪指令只能出现在代码段或 E²PROM 段。DW 伪指令前应使用一个标号，以标记所定义的字常数区域的起始位置。DW 伪指令为一个表达式列表，表达式列表由多个表达式组成，但至少要含有一个表达式，表达式之间用逗号分隔。每个表达式值的范围必须在-32768～65535 之间。如果表达式的值是负数，则用 16 位 2 的补码表示。

语法：LABEL：.DW 表达式表

8．EQU—定义标识符常量

EQU 伪指令将表达式的值赋给一个标识符，该标识符为一个常量标识符，可以用于后面的指令表达式中，在汇编时凡遇到该标识符都以其等值表达式替代。在编写程序中，只要修改此表达式，就修改了程序中多处涉及该表达式的地方，减少了程序的修改量。但该标识符的值不能改变或重新定义。

语法：.EQU 标号 = 表达式

9．ESEG—声明 E²PROM 段

ESEG 伪指令声明 E²PROM 段的开始。一个汇编文件可以包含几个 E²PROM 段，这些 E²PROM 段在汇编编译过程中被连接成一个 E²PROM 段。在 E²PROM 段中不能使用 BYTE 伪指令。每个 E²PROM 段内部都有自己的字节定位计数器。可使用 ORG 伪指令定义该字节定位计数器的初始值，作为数据段在 E²PROM 中的起始位置。ESEG 伪指令不带参数。

语法：.ESEG

10．EXIT—退出文件

EXIT 伪指令告诉汇编编译器停止汇编该文件。在正常情况下，汇编编译器的编译过程一直到文件的结束，如果中间出现 EXIT，则编译器放弃对其后边语句的编译。

如果 EXIT 出现在所包含文件中，则汇编编译器将结束对包含文件的编译，然后从本文件当前 INCLUDE 伪指令的下一行语句处开始继续编译。

语法：.EXIT

11．INCLUDE—包含指定的文件

INCLUDE 伪指令告诉汇编编译器开始从一个指定的文件中读入程序语句，并对读入的语句进行编译，直到该包含文件结束或遇到该文件中的 EXIT 伪指令，然后再从本文件当前 INCLUDE 伪指令的下一行语句处继续开始编译。在一个包含文件中，也可以使用 INCLUDE 伪指令来包含另外一个指定的文件。

语法：.INCLUDE“文件名”

12．MACRO—宏开始

MACRO 伪指令告诉汇编器一个宏程序的开始。MACRO 伪指令将宏程序名作为参数。当后面的程序中使用宏程序名，则表示在该处调用宏程序。宏有些象子程序或公式，它是一段含形式参数（亚元）的程序。一个宏程序中可带 10 个形式参数，这些形式参数在宏定义中用@0～@9 代表，参数之间用逗号分隔。当调用一个宏程序时，必须以实参替代形参。伪指令 ENDMACRO 定义宏程序的结束。

默认情况下，在汇编编译器生成的列表文件中仅给出宏的调用。如要在列表文件中给出宏的表达式，则必须使用 LISTMAC 伪指令。在列表文件的操作码域中，宏带有 a+ 的记号。

语法：.MACRO 宏名

13．ENDMACRO—宏结束

ENDMACRO 伪指令定义宏定义的结束。该伪指令并不带参数，参见 MACRO 宏定义伪指令。

语法：.ENDMACRO

示例：

```
.MACRO SUBI16                ;宏定义开始，定义一个 16 位的减法
    subi @1,low(@0)          ;减低 8 位字节
    sbci @2,high(@0)         ;带借位减高 8 位字节
.ENDMACRO                    ;宏定义结束
……
……
.CSEG                        ;代码段开始
    SUBI16 0x1234,r16,r17    ;调用宏完成 r17:r16 = r17:r16 - 0x1234
```

14．LISTMAC—列表时输出宏表达式

LISTMAC 伪指令告诉汇编编译器，在生成的列表清单文件中，显示所调用宏的内容。默认情况下，仅在列表清单文件中显示所调用的宏名和参数。

语法：.LISTMAC

15．LIST—启动生成列表清单文件功能

LIST 伪指令告诉汇编编译器在对源文件编译后，产生一个机器语言与源文件相对照的列表清单文件。正常情况下，汇编编译器在编译过程中将生成一个由汇编源代码、地址和机器操作码组成的列表文件。默认时为允许生成列表清单文件。该伪指令可以与 NOLIST 伪指令配合使用，以选择仅使某一部分的汇编源文件产生列清单表文件。

语法：.LIST

16．NOLIST—关闭列表清单文件生成功能

NOLIST 伪指令告诉汇编编译器关闭列表清单文件生成的功能。正常情况下，汇编编译器在编译过程中将生成一个由汇编源代码、地址和机器操作码组成的列表文件，默认情况下为允许生成列表清单文件。当使用该伪指令时，将禁止文件列表清单的产生。该伪指令与 LIST 伪指令配合使用，可以选择使某一部分的汇编源文件产生列表清单文件。

语法：.NOLIST

17．ORG—定义代码起始位置

ORG 伪指令设置定位计数器为一个绝对数值，该数值为表达式的值，作为代码的起始位置。如果 ORG 伪指令出现数据段中，则设定 SRAM 定位计数器；如果该伪指令出现在代码段中，则设定程序存储器计数器；如果该伪指令出现在 E^2PROM 段中，则设定 E^2PROM 定位计数器。如果该伪指令前带标号（在相同的语句行），则标号的定位由 ORG 的参数值定义。代码段和 E^2PROM 段定位计数器的默认值是零；而当汇编器启动时，SRAM 定位计数器的默认值是 32（因为寄存器占有地址为 0～31）。文件中可以出现多处 ORG 伪指令，但后面 ORG

所带的地址不能小于前一个 ORG 所带的地址，也不能落在由前一个 ORG 定位的代码段空间（指同一个存储器空间）。注意，E^2PROM 和 SRAM 定位计数器按字节计数，而程序存储器定位计数器按字计数。

语法：.ORG 表达式

18．SET—设置标识符与一个表达式值相等

与 EQU 作用类似，SET 伪指令将表达式的值赋值给一个标识符。该标识符为一个常量标识符，可以用于后面的指令表达式中，在汇编时凡遇到该标识符都以其等值表达式替代。与 EQU 不同的是，用 SET 伪指令赋值的标识符能在后面使用 SET 伪指令重新设置改变。在 SET 改变之后的代码使用新的等值表达式值，直到遇到下一个重新的 SET 定义为止。

语法：.SET 标号 = 表达式

3.7.3 表达式

在标准指令系统中，操作数通常只能使用纯数字格式，这给程序的编写带来了许多不便。但是在编译系统的支持下，在编写汇编程序时允许使用表达式，以方便程序的编写。AVR 编译器支持的表达式是由操作数、函数和运算符组成。所有的表达式内部都是 32 位的。

1．操作数

AVR 汇编器使用的操作数有以下几种形式：

（1）用户定义的标号，该标号给出了放置标号位置的定位计数器的值。

（2）用户用 SET 伪指令定义的变量。

（3）用户用 EQU 伪指令定义的常数。

（4）整数常数，包括下列几种形式：

十进制数（默认），如：10、255；

十六进制数，如：0x0a、$0a、0xff、$ff；

二进制数，如：0b00001010、0b11111111

（5）PC：程序存储器定位计数器的当前值。

2．函数

AVR 汇编器定义了下列函数：

（1）LOW（表达式）　　返回一个表达式值的最低字节。

（2）HIGH（表达式）　　返回一个表达式值的第二个字节。

（3）BYTE2（表达式）　　与 HIGH 函数相同。

（4）BYTE3（表达式）　　返回一个表达式值的第三个字节。

（5）BYTE4（表达式）　　返回一个表达式值的第四个字节。

（6）LWRD（表达式）　　返回一个表达式值的 0～15 位。

（7）HWRD（表达式）　　返回一个表达式值的 16～31 位。

（8）PAGE（表达式）　　返回一个表达式值的 16～21 位。

（9）EXP2（表达式）　　返回（表达式值）2 次幂的值。

（10）LOG2（表达式）　　返回 Log_2（表达式值）的整数部分。

3．运算符

汇编器提供的部分运算符见表 3-7。优先级数越高的运算符，其优先级也越高。表达式可以用小括号括起来，并且与括号外其他任意的表达式再组合成表达式。

表 3-7　部分运算符表

序　　号	运　算　符	名　　称	优　先　级	说　　明
1	!	逻辑非	14	一元运算符，表达式是 0 返回 1， 表达式是 1 返回 0
2	～	逐位非	14	一元运算符，将表达式的值按位取反
3	-	负号	14	一元运算符，使表达式为算术负
4	*	乘法	13	二进制运算符，两个表达式相乘
5	/	除法	13	二进制运算符，左边表达式除以右边表达式， 得整数的商值
6	+	加法	12	二进制运算符，两个表达式相加
7	—	减法	12	二进制运算符，左边表达式减去右边表达式
8	<<	左移	11	二进制运算符，左边表达式值左移右边表达 式给出的次数
9	>>	右移	11	二进制运算符，左边表达式值右移右边表达 式给出的次数
10	<	小于	10	二进制运算符，左边带符号表达式值小于 右边带符号表达式值，则为 1，否则为 0
11	<=	小于等于	10	二进制运算符，左边带符号表达式值小于或 等于右边带符号表达式值，则为 1，否则为 0
12	>	大于	10	二进制运算符，左边带符号表达式值大于 右边带符号表达式值，则为 1，否则为 0
13	>=	大于等于	10	二进制运算符，左边带符号表达式值大于或 等于右边带符号表达式值，则为 1，否则为 0
14	==	等于	9	二进制运算符，左边带符号表达式值等于 右边带符号表达式值，则为 1，否则为 0
15	!＝	不等于	9	二进制运算符，左边带符号表达式值不等于 右边带符号表达式值，则为 1，否则为 0
16	&	逐位与	8	二进制运算符，两个表达式值之间逐位与
17	^	逐位异或	7	二进制运算符，两个表达式值之间逐位异或
18	|	逐位或	6	二进制运算符，两个表达式值之间逐位或
19	&&	逻辑与	5	二进制运算符，两个表达式值之间逻辑与， 全非 0 则为 1，否则为 0
20	||	逻辑或	4	二进制运算符，两个表达式值之间逻辑或， 非 0 则为 1，全 0 为 0

3.7.4　器件定义头文件“m16def.inc”

在 AVR 的器件手册中，对所有的内部寄存器进行了标称化的定义，如 32 个通用寄存器组用 R0～R31 表示，系统状态寄存器用 SREG 表示，A 口输出 I/O 寄存器用 PORTA 表示等，便于记忆、理解和使用。而当编写程序时，在指令中实际出现的应该是这些寄存器的空间地址，这就给编写程序造成了麻烦。

为了能让程序员编写程序时，不去使用那些不易记忆的寄存器地址，而是直接使用这些寄存器的标称名称，在 AVR 的开发软件平台中都含有各个 AVR 器件的标称定义头文件。在 ATMEL 公司提供的 AVR 开发平台 AVR Studio 中，含有很多的“器件型号.inc”文件，这些 inc 文件，已将该器件所有的 I/O 寄存器、标志位、中断向量地址等进行了标称化的定义。这些定义的标称化符号与硬件结构中寄存器的命名是相同的，同时定义了它们所代表

的地址值，这样在程序中就可直接使用标称化的符号，而不必去记住它的实际地址。如使用 PORTA 来代替 A 口 I/O 输出寄存器的地址$1b。读者可具体查看相关器件的定义文件（在安装 AVR Studio 系统目录的下一级子目录 Appnotes 中）。

作为一个实际的例子，我们看一下 ATmega16 器件定义头文件“m16def.inc”中的部分内容：

```
;***** Specify Device
.device ATmega16

;***** I/O Register Definitions
.equ    SREG = $3f
.equ    SPH  = $3e
.equ    SPL  = $3d
.....
.equ    PORTA= $1b
.equ    DDRA = $1a
.equ    PINA = $19
.....
.def    XL   = r26
.def    XH   = r27
.def    YL   = r28
.def    YH   = r29
.def    ZL   = r30
.def    ZH   = r31

.equ    RAMEND = $45F
.....
```

可以看出，在器件定义文件中，大量使用汇编的伪指令 EQU、DEF 等，定义了各个寄存器标称名所对应的地址值。因此，当我们在编写 AVR 汇编程序时，在程序的开始处，需要先使用伪指令“.INCLUDE”，调用编译系统中的器件标识定义文件“****def.inc”。由于该文件已将该器件所有的 I/O 寄存器、标志位等进行了标称化的符号定义，这样在程序中就可直接使用标称化的符号，而不必去使用它的实际地址了。下面是一个标准的汇编程序的开始部分：

```
.INCLUDE "M16def.inc"                ;引用器件 I/O 标称定义文件
.DEF TEMP1 = r20                     ;定义标识符 TEMP1 代表工作寄存器 R20
.....

.ORG  $0000                          ;代码段起始定位
        jmp RESET                    ;系统上电复位，跳转到主程序

.ORG  $002A                          ;代码段定位，跳过中断向量区

;程序先对器件进行初始化
;设置 ATmega16 的堆栈指针为$045F，RAMEND 在配置文件"M16def.inc"中已定义为$045F
;设置 A 口为输出方式工作
```

```
RESET:  ldi r16,high(RAMEND)        ;取 RAMEND 的高位字节
        out  SPH,r16                ;将 RAMEND 的高位送堆栈寄存器 SP 高位字节中
        ldi  r16,low(RAMEND)        ;取 RAMEND 的低位字节
        out SPL,r16                 ;将 RAMEND 的低位送堆栈寄存器 SP 低位字节中

        ser  temp1                  ;将 temp1 即寄存器 R20 置为$FF
        out  DDRA,temp1             ;R20 值送 DDRA，A 口方向寄存器为$FF,设定为输出
        ……                          ;DDRA 在配置文件"M16def.inc"中已定义为$1A
```

以上对 AVR 单片机的指令结构和汇编系统做了基本的介绍，在本书的第 5 章会结合汇编开发平台 AVR Studio 的使用，给出一些采用汇编编写的程序示例。

如果从具体应用的角度出发，并结合 AVR 单片机的特点，在产品设计开发过程中最好是使用高级语言来编写系统程序。同时，使用高级语言也便于从系统的角度描述介绍、学习和掌握开发设计单片机系统程序的思想、方法和技巧。因此，本书将以高级语言 C 作为主要的工具（使用 CVAVR 软件平台）进行讲述。也由于篇幅的关系，对 AVR 汇编程序的使用不做更详细的学习。但了解和掌握 AVR 汇编，对熟悉 AVR 的功能、系统软件的调试都非常重要。读者可进一步参考《AVR 单片机实用程序设计》一书，该书对 AVR 汇编进行了更为细致的介绍，并给出了大量的、实用的 AVR 汇编参考代码。

本章参考文献：

1.《AVR指令集》（英文，CDROM），ATMEL，www.atmel.com

2.《AVR汇编应用指导》（英文，CDROM），ATMEL，www.atmel.com

3.《AVR 单片机使用程序设计》，张克彦 编著，北京航空航天大学出版社

第 4 章 AVR单片机系统设计与开发工具

在学习和掌握如何应用单片机来设计和开发嵌入式系统时，除了要对所使用的单片机有全面和深入的了解外，配备和使用一套好的开发环境和开发平台也是必不可缺的。在嵌入式系统的设计开发中，选用了好的开发工具和开发平台，往往能加速嵌入式应用系统的研制开发、调试、生产和维修，起到事半功倍的效果。

国内外许多公司根据不同单片机的性能和特点，研制推出了各种类型的用于开发单片嵌入式系统的单片机开发装置和软件开发平台。不同类型的单片机使用的开发系统是不同的。对同一类型的单片机来讲，也有多种类型和功能的开发装置和开发平台。价格便宜、性能适中的系统在几百元，高性能的开发系统则要数千元到上万元，甚至仅仅一套软件开发平台就要上万元。虽然设计开发一个嵌入式系统，可以选用多家公司、多种类型的单片机，但在决定学习和使用哪种单片机时，应对单片机的性能价格，开发装置和开发平台的性能价格，以及是否方便使用等，几方面做一个综合的评估。

由于 AVR 单片机的程序存储器采用的是可多次下载的 Flash 存储器，具有可在线下载（ISP）等的优良特性，给学习和使用都带来极大的方便。

本章将在介绍单片机嵌入式系统设计开发基础知识之后，重点介绍和讲述本书推荐和使用的一套采用 ATMEL 公司的 AVR Studio 配合 C 高级语言的软件开发平台——CodeVisionAVR（简称 CVAVR）所构成的开发软件环境，以及一套简易、开放的，集下载编程、实验和开发一体的多功能 AVR-51 实验板。

4.1 单片机嵌入式应用系统设计

4.1.1 单片机嵌入式系统开发所需的基础知识和技能

在 IT 行业，应用系统设计可以分成两大类，一类用于科学计算、数据处理、企业管理、Internet 网站建立等；另一类用于工业过程检测控制、智能仪表仪器和自动化设备、小型电子系统、通信设备、家用电器等。

对于前一类的应用系统设计，通常都是基于通用计算机系统和网络的系统开发，硬件设备也是通用的，可以从市场购买，而其主要的工作是软件开发，使用的开发平台以 C++、VB、数据库系统、网站建立开发平台等。

而后一类应用系统的设计则同前一类有很大的不同。它涉及的应用系统是一个专用的系统，往往要从零开始。既必须根据实际的需求，从系统硬件的构成设计与实现，到相应的软件设计与实现，两者并重，相辅相成，却一不可。

第二类应用系统的特点是：

- 系统功能、要求、性能的多样性和专用性。
- 硬件电路和软件设计的不可分割和专一性。
- 可靠性高，抗干扰能力强。
- 体积小、重量轻、功耗省、投资少。
- 开发周期短，见效快。

单片机嵌入式应用系统设计归属于第二类应用系统的范畴。因此，对于从事单片机嵌入式系统设计、开发、学习的电子工程师和专业人员来讲，不仅要熟悉各种电子器件和 IC 芯片的使用和特性，具备模拟电路、数字电路等各类硬件电路和硬件系统的设计能力，还必须

具有很强的计算机综合应用和软件编程设计能力。

在今天，单片机嵌入式系统的硬件设计、软件编程、系统仿真调试和程序的编程下载，大都是在个人电脑 PC 的支撑下实现的。因此，单片机嵌入式系统设计开发人员所具备的另一个基本重要的技能就是要熟练掌握和使用个人电脑 PC，应具备熟练使用个人电脑 PC 的基础，具备相应的软件设计编程能力，熟悉相关软件(如 Protel、VHDL)的使用，同时对 PC 机的硬件接口（RS-232 串行通信口、LPT 并行打印机接口、USB 接口等）也要有一定的了解。

早期的单片机系统开发平台是以 PC 的 DOS 操作系统为支撑的，但随着 PC 机的发展，现在的单片机系统开发平台都已经转到以 Windows 平台支撑的 PC 上。Windows 平台具有的多任务、多窗口性能给单片机嵌入式系统的设计开发带来极大的方便。

当你设计研制的单片机嵌入式系统是一个大型管理控制系统的下位机，或要与 Internet 或局网中的数据库联网，那么你除了要熟练掌握与单片机有关的硬件（模拟电路、数字电路、单片机等)和软件开发技术外，你还要具备与整个大的系统有关的基础和技术(如，数据库、Internet 协议、VB、VC 等)。因此，对一个高级电子工程师来讲，他对个人电脑 PC 的熟练掌握程度，以及软件设计和编程的能力，决不亚于计算机专业的人员，在某些方面比计算机专业的人员要求还高，还要全面。

要具备较高的硬件系统设计开发能力和水平，不是在短期内通过理论和书本的学习就能得以实现的。它需要经过一定时间的学习，并且特别注重理论与实际相结合，要亲自独立的动手去做，去实践，才能打下良好的基础。所以说，不亲自动手实践，你是不可能真正掌握设计开发单片机嵌入式系统技术的。有了良好的基础，有了长期的实践经验，加上紧跟世界半导体器件的最新发展，你才能成为一个真正的电子工程师。

4.1.2 单片机嵌入式系统开发过程

对于单片机嵌入式系统的设计与开发来讲，由于涉及对象和要求的多样性和专用性，其硬件和软件结构有很大差异，但系统设计开发的基本内容和主要步骤是基本相同的。图 4-1 是单片机嵌入式系统开发过程示意图。

在一个具体的单片机嵌入式系统的设计时，一般需要作以下几个方面的考虑：

1. 确定系统设计的任务

在进行系统设计之前，首先必须进行设计方案的调研，包括查找资料、进行调查、分析研究。要充分了解对系统的技术要求、使用的环境状况以及使用人员的技术水平。明确任务，确定系统的技术指标，包括系统必须具有那些功能等。这是系统设计的出发点，它将贯串于整个系统设计的全过程，也是产品设计开发工作成败、好坏的关键，因此必须认真做好这项工作。

2. 系统方案设计

在系统设计任务和技术指标确定以后，即可进行系统的总体方案设计，一般包括：

（1）单片机芯片的选择。单片机芯片的选择应适合于应用系统的要求。不仅要考虑单片机芯片本身的性能是否能够满足系统的需要，如：执行速度、中断功能、I/O 驱动能力与数量、系统功耗以及抗干扰性能等，同时还要考虑开发和使用是否方便、市场供应情况与价格、封装形式等其它因素。

（2）外围电路芯片和器件的选择。仅仅一片单片机芯片是不能构成一个完整的嵌入式系统的。一个典型的系统往往由输入部分（按键、A/D、各种类型的传感器与输入接口转换电路），输出部分（指示灯、LED 显示、LCD 显示、各种类型的传动控制部件），存储器（用于系统数据记录与保存），通信接口（用于向上位机交换数据、构成联网应用），电源供电等多个单元组成。这些不同的单元涉及到模拟、数字、弱电、强电以及它们相互之间的协调配合、

转换、驱动、抗干扰等。因此，对于外围芯片和器件的选择，整个电路的设计，系统硬件机械结构的设计，接插件的选择，甚至产品结构、生产工艺等，都要进行全面和细致的考虑。任何一个忽视和不完善，都会给整个系统带来隐患，甚至造成系统设计和开发的失败。

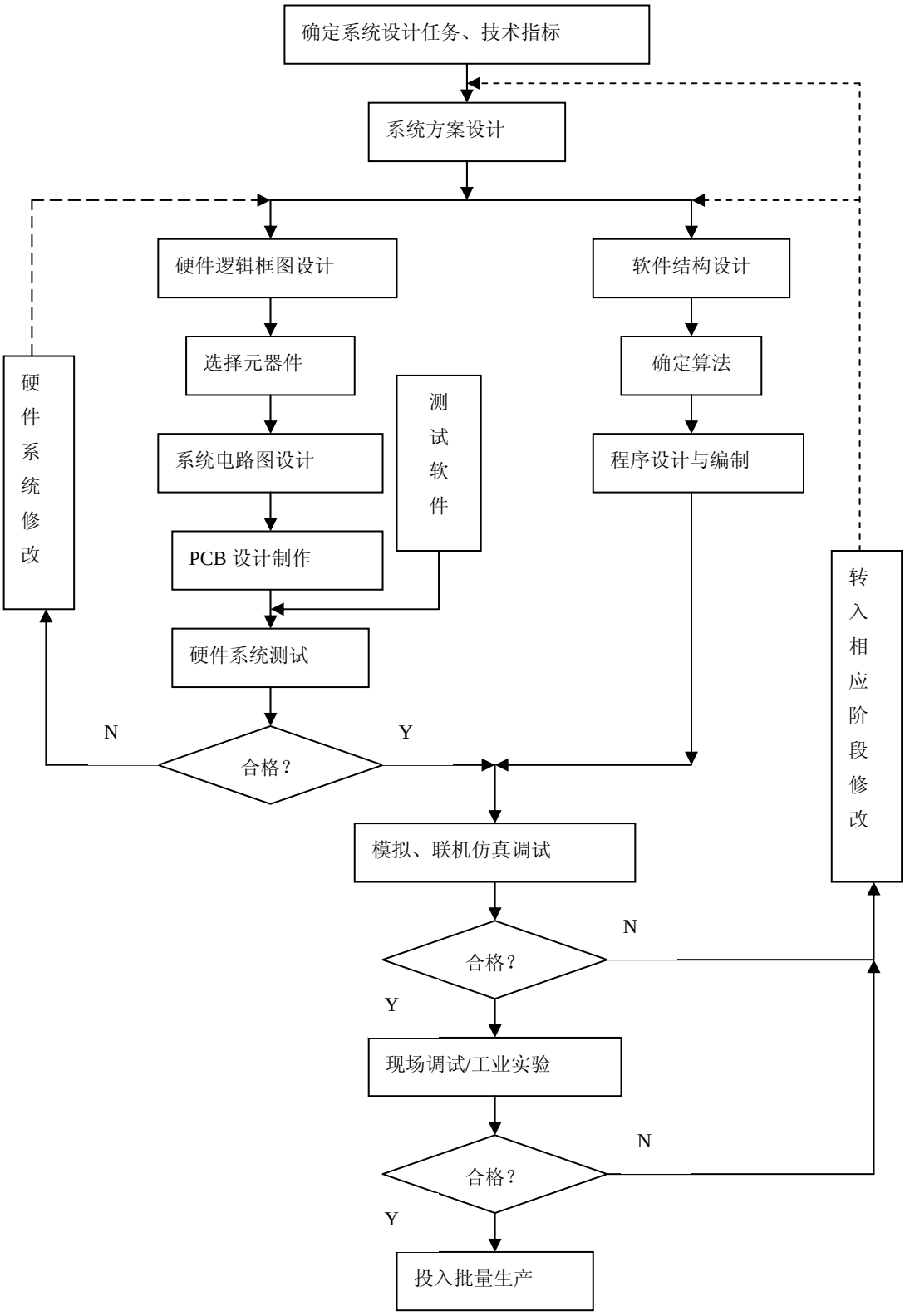

图 4-1 单片机嵌入式系统开发过程

（3）综合考虑软、硬件的分工与配合。单片机嵌入式系统中的硬件和软件具有一定的互换性，有些功能可以用硬件实现，也可以用软件来实现，因此，在方案设计阶段要认真考虑软、硬件的分工和配合。采用软件实现功能可以简化硬件结构，降低成本，但软件系统则相应的复杂化，增加了软件设计的工作量。而用硬件实现功能则可以缩短系统的开周期，使软件设计简单，相对提高了系统的可靠性，但可能提高了成本。在设计过程中，软、硬件的分工与配合需要取得协调，才能设计出好的应用系统。

3. 硬件系统设计

开发人员在全面了解要设计开发系统所具备的功能和要求，制定出整体的系统设计方案后，接下来就是根据具体的需求和设计方案，选择能可靠实现全部功能的单片机芯片和相应的外围电路器件，设计整个系统的电原理图。原理图设计完成后，还要根据实际需要设计相应的印刷板（PCB）图。这个阶段常使用的软件平台是电子电路 CAD 软件，如 PROTEL 等。

单片机嵌入式系统的硬件系统设计是一个综合能力的表现，它全面反映和体现了设计开发人员的所具有的技术水平和创新设计能力。比如说，设计一个具备相同功能的单片机嵌入式系统，如采用传统并行总线扩展外围设备的设计思路，设计出的硬件系统就相对庞大和复杂，因为仅地址线和数据线就有 16+8=24 根，还需要相应的锁存器和地址译码器等器件，稳定性、抗干扰性都相对差一些。如采用新型的单片机，CMOS 器件，选用串行接口的大容量存储器、AD/DA 等器件，就可减少硬件开发的工作量，大大缩短系统设计开发的周期，同时也提高了系统的可靠性。

4. 系统软件设计编写

在硬件系统设计的基础上，则要根据系统的功能要求和硬件电路的结构设计和编写系统软件。作为单片机系统软件设计人员，应该具备扎实的硬件功底，不仅是对系统的功能和要求有深入的了解，而且对实现的硬件系统、使用的芯片和外围电路的性能也要很好的掌握。这样才能设计出可靠的系统程序。

一个嵌入式系统的系统软件实际上就是该系统的监控程序。对于一些小型嵌入式系统的应用程序一般采用汇编语言编写。对于中、大型的嵌入式系统，常采用高级语言(如：C 语言、Basic 语言)来编写。软件设计和编写也是开发嵌入式系统过程中非常重要和困难的任务之一，因为它直接关系到实现系统的功能和系统的性能。

通常在编制程序前应对系统要实现的功能、硬件系统的结构和电路、系统中使用的单片机和外围器件进行全面仔细和深入的了解，对系统软件的结构进行全面和完整的设计，编制程序流程图。系统程序的设计应实现结构化、模块化、子程序化，这不仅便于调试，还便于修改。

要特别注意的是，设计编写嵌入式系统的软件同编写其它类型的软件程序有较大的区别。由于嵌入式系统是直接面对硬件、控制对象的，因此，设计编写嵌入式系统的程序需要考虑更多的硬件细节，要掌握和使用很多软件技巧，要多学习、多实践。如，嵌入式系统程序的设计要仔细地考虑和划分程序存储器、数据存储器；合理定义、安排和使用各种变量；尽量使用字节变量和位标志变量，优化程序，节省内存容量；估算子程序调用和嵌套的最大级数，预留出足够的堆栈的空间等等。

5. 系统调试

当硬件和软件设计好后，就可以进行系统调试了。硬件电路系统调试检查分为静态检查和动态检查。硬件的静态检查主要检查电路制作的正确性，如路线、焊接等。动态检查一般首先要使用仿真系统（对于采用 ISP 技术的系统可直接）输入各种单元部分的系统调试和诊断程序，检查系统的各个部分的功能是否能正常工作。硬件电路调试完成后可进行系统的软

硬件联调。先将各功能模块程序分别调试完毕，然后组合，进行完整的系统运行程序调试。最后还要进行各种工业测试和现场测试，考验系统在实际应用环境中是否能正常可靠的工作，是否达到设计的性能和指标。

系统的调试往往要经过多次的反复。硬件系统设计的不足、软件程序中的漏洞，都可能是造成系统调试出现问题。系统调试要具备相当水平和实践经验，它全面反映了嵌入式系统设计开发者的水平和能力。

以上各个方面的能力是不能够仅仅通过书本的理论学习就能够掌握的，因此，学习和掌握单片嵌入式系统的设计、开发与应用，要非常注重实际的动手练习，要在学习中实践、在实践中加深学习，只有这样才能不断巩固、加强和深入下去，才能真正的掌握这门技术。

4.2 单片嵌入式系统的开发工具与环境

4.2.1 单片嵌入式系统的程序设计语言

在掌握单片机结构和系统设计基础上，根据系统的设计和系统的功能要求就可以编写系统应用程序。掌握程序设计的方法和技术对于嵌入式系统的学习和应用开发具有十分重要的地位。

开发单片机嵌入式系统所用的程序设计语言可分为三类：机器语言、汇编语言和高级语言。

1. 机器语言

机器语言是完全面向芯片的语言，由二进制码“0”和“1”组成。在单片机的程序存储器中存放就是以“0”和“1”构成的二进制序列指令字，它是单片机 CPU 直接识别和执行的语言。用机器语言表示的程序称为机器语言程序或目标程序。如下面一条 AVR 机器语言代码：

0000110000000001

就是将 AVR 单片机内部的寄存器单元 R0 和 R1 的内容相加，结果保存在 R0 中。

采用机器语言编程不仅难学、难记，而且也不易理解和调试，因此人们不直接使用机器语言来编写系统程序，往往使用汇编语言或高级语言编写程序。不过，无论使用汇编语言还是高级语言来编写系统程序，最终都需要使用相应的开发软件系统（一般在软件开发平台中的都提供编译软件系统）将其编译成机器语言，生成目标程序的二进制代码文件（.bin 或.hex），然后再把目标代码写入（编程下载）单片机的程序存储器中，最后由单片机的 CPU 执行。

2. 汇编语言

汇编语言是一种符号化的语言，它使用一些方便记忆特定的助记符（特定的英文字符）来代替机器指令。如“ADD”表示加，“MOV”表示传送等。上面的 AVR 机器指令用汇编语言表示为：ADD R0，R1

用汇编语言编写的程序称为汇编语言程序，显然，它比机器语言易学、易记。但是，汇编语言也是面向机器的，也属于低级语言。由于各种单片机的机器指令不同，每一类单片机的汇编语言也是不同的，如 8051 的汇编语言同 AVR 的汇编语言是完全不一样的。

传统开发单片嵌入式系统主要是用汇编语言编写系统程序。学习和采用汇编语言开发系统程序的优点是：能够全面和深入的理解单片机硬件的功能，充分发挥单片机的硬件特性。但由于汇编语言编写的程序可读性、可移植性（各种单片机的机器指令不同）和结构性都较差，因此采用汇编语言编来开发单片机应用系统程序的比较麻烦，调试和排错也比较困难，产品开发周期长，同时要求软件设计人员要具备相当高的能力和经验。

3. 高级语言

高级语言是一种“基本”不依赖硬件的程序设计语言。这里的“基本”是指编写在通用计算机系统上远行的系统软件。

由于高级语言具有面向问题或过程，其形式类似自然语言和数学公式，结构性、可读性、可移植好的特点，所以为了提高编写系统应用程序的效率，改善程序的可读性和可移植性，缩短产品的开发周期，采用高级语言来开发单片机系统已成为当前的发展趋势。

需要特别注意的是，在设计开发单片嵌入式系统的系统软件过程中，总是要同硬件打交道，而且关联是比较密切的，其软件设计有着自己独特技巧和方法。因此，那些纯软件出身的软件工程师，如果没有硬件的基础，没有经过一定的学习和实践，可能还写不好，甚至写不了单片嵌入式系统的系统软件。

作为一个有经验的单片嵌入式系统开发人员，应能同时掌握和使用汇编语言和高级语言设计系统程序。

概括起来说，基于高级语言开发单片机系统具有语言简洁，使用方便灵活，可移植性好，表达能力强，可进行结构化程序设计等优点。对于开发大型和复杂的嵌入式系统来讲，采用高级程序设计语言进行系统开发的效率比使用汇编语言高几倍甚至几十倍。但对于小型、简易的系统，或有定时精确，高测量精度要求的系统，使用汇编语言则具有优势。在许多情况下，采用高级语言嵌入汇编程序的软件设计技术往往是最有效的方法。

如果你对单片机的内部结构和汇编语言根本不了解，请先不要用 C 语言编程。

如果你对单片机的内部结构和汇编语言根本不了解，也写不出好的单片机的 C 程序。

不管是使用汇编语言还是高级语言来开发单片机系统程序，都需要一个专用的软件平台把软件设计人员编写的源程序“翻译”成二进制的机器指令代码，这个“翻译”过程对汇编语言来讲称为汇编，对高级语言来讲，它包括编译和连接两个过程。因此，一个性能优良的，专门用于开发单片机的软件平台和环境也是必不可少的开发工具。

4.2.2 单片嵌入式系统的开发软件平台

单片嵌入式系统的设计和开发需要一个好的软件开发平台的支持。一个好的单片嵌入式系统的开发软件通常具备以下几个重要的功能：

- 单片机系统程序编写和运行代码的生成。（编辑、编译功能）
  嵌入式系统开发平台支持用户采用专用汇编程序设计语言或高级程序设计语言（C、Basic 等）编写嵌入式系统控制程序的源代码，并将源代码编译连接生成可在单片机中执行的二进制代码（Hex、Bin）。
- 软件模拟仿真。
  提供一个纯软件的仿真环境，在此环境的支持下，单片机的系统程序可以进行模拟的运行，以实现第一步的软件调试和排错功能。
- 在线仿真功能。
  与专用的仿真器配合，提供一个硬件在线的实时仿真调试环境（见下节在线仿真器）。用户将编写好的目标系统运行代码下载到仿真器中，通过开发系统软件控制仿真器中程序的运行，同时观察硬件系统的运行结果，分析、调试和排除系统中存在的问题。

- 程序下载烧入功能。
  与专用的编程器配合或使用 ISP 技术，将二进制运行代码写入到单片机的程序存储器中。

要熟练掌握和应用单片机来设计开发嵌入式系统，除了对所使用的单片机要有全面和深入的了解外，配备和使用一套好的开发环境和开发平台也是必不可缺的。在嵌入式系统的设计开发中，选用了好的开发工具和开发平台，往往能加速嵌入式应用系统的研制开发、调试、生产和维修，起到事半功倍的效果。

国内外许多公司根据不同单片机的性能和特点，研制推出了各种类型的用于开发单片嵌入式系统的单片机开发装置和软件开发平台。不同类型的单片机使用的开发系统是不同的。对同一类型的单片机来讲，也有多种类型和功能的开发装置和开发平台。价格便宜、性能适中的系统在几百元，高性能的开发系统则要数千元到上万元，甚至仅仅一套软件开发平台就要上万元。虽然设计开发一个嵌入式系统，你可以选用多家公司、多种类型的单片机，但你在决定学习和使用哪种单片机时，应对单片机的性能价格，开发装置和开发平台的性能价格，以及是否方便使用等，几方面做一个综合的评估。

4.2.3 单片嵌入式系统的硬件开发工具

在学习和应用单片机来设计开发嵌入式系统的过程中，一般应配备两种硬件设备：仿真器和编程烧入器。仿真器是用于对所设计嵌入式系统的硬软件进行调试的工具，而编程烧入器的作用则是将系统执行代码写入到目标系统中。现在更多的开发设备是将仿真器和编程烧入器合二为一了，同时具备了两者的功能。

调试（Debug）是系统开发过程中必不可少的环节。但是嵌入式系统开发的调试环境和方法同通用计算机系统的软件开发有着明显的差异。通用计算机系统的软件开发基本与硬件无关，而且调试器与被调试程序常常位于同一台计算机上（在相同的 CPU 上运行），如在 Windows 平台上利用 VC、VB 等语言开发在 Windows 上的运行的软件。而对于嵌入式系统的开发，由于开发主机和目标机处于不同的机器中（在不同的 CPU 上运行）：系统程序在开发主机上进行开发，编译生成在另外机器上执行的代码文件，然后需要下装到目标机后才能运行，那么对嵌入式系统的调试方法和过程就比较麻烦和复杂。

目前在嵌入式系统开发过程中，经常采用的调试方法有三种方式：软件模拟仿真调试（Simulator）、实时在板仿真调试（On Board Debug）和实时在片仿真调试（On Chip Debug）。其中软件模拟仿真调试技术和实时在片仿真调试技术发展很快，逐渐成为调试嵌入式系统的主要手段。

1. 软件仿真器

软件仿真器也称为指令集模拟器（ISS），其原理是用软件来模拟 CPU 处理器硬件的执行过程，包括指令系统、中断、定时计数器、外部接口等等。用户开发的嵌入式系统软件，就像已经下装到目标系统硬件一样，载入到软件模拟器中运行，这样用户可以方便对程序运行进行控制，对运行过程进行监视，进而达到实现调试的目的。由于这种调试不是在真正的目标板系统上进行的，而是采用软件模拟方式实现的，所以它是一种非实时性的仿真调试手段。

软件仿真器的一个优点是它可以使嵌入式系统的软件和硬件开发并行开展。只要硬件设计工作完成后，不管硬件实体如何，就可以进行软件程序的编写和调试了。应用程序在结构上、逻辑上的错误能够利用软件仿真器很快的发现和定位。有些与硬件相关的故障和错误也能在软件仿真器中被发现。使用软件仿真器不仅可以缩短产品开发周期，而且非常经济，不需要购买昂贵的实时仿真设备。同时软件仿真器也是学习和加深了解所使用处理器的内部

结构和工作原理的最好工具。

使用软件仿真器的缺点是其模拟的运行速度比真正的硬件慢的多，一般要慢 10～100 倍。另外软件仿真器只能模拟仿真软件的正确性，仿真与时序有关，查找同硬件有关的错误比较困难。

目前推出的比较先进的单片嵌入式系统开发平台一般都内含软件仿真器，如 ATMEL 公司的 AVR Studio 中就包含一个功能非常强大的软件仿真器，能够实现汇编级和高级语言级的软仿真功能。一些针对 AVR 开发的平台，如 IAR、BASCOM 中也都包含自己的软件仿真器。值得一提的是 BASCOM 的软件仿真器提供了模拟实物图形化界面，将一些标准化的外围器件如字符 LCD 模块、键盘模块等作为实物显示在屏幕上，用户能够更加直观的看到系统运行的结果，使用非常方便。另外，目前在市场上有一些专用的软件模拟平台，如 vmlab 等，都可以实现对 AVR 的模拟仿真调试，但一般价格比较昂贵。

2. 实时在板仿真器（ICE）

实时在板仿真器通常称为在线仿真 ICE（In Circuit Emulate），它是最早用于开发嵌入式系统的工具。ICE 是实际是一个特殊的嵌入式系统，一般是由专业公司研制和生产。它的内部含有一个具有“透明性”和“可控性”的 MCU，可以代替被开发系统（目标系统）中的 MCU 工作，既用 ICE 的资源来仿真目标机。因此，ICE 实际上是内部电路仿真器，它是一个相对昂贵的设备，用于代替微处理器，并植入微处理器与总线之间的电路中，允许使用者监视和控制微处理器所有信号的进出。因此，这种仿真方式和设备，更准确的讲应该称为实时在板仿真（On Board Debug）器。

ICE 仿真器一般使用串行口（COM 口或 USB 接口）或并行口（打印机口）同 PC 机通信，并提供一个与目标机系统上的 MCU 芯片引脚相同的插接口（仿真口）。使用时，将目标机上的 MCU 取下，插上仿真器的仿真口，仿真器的通信口与 PC 连接（图 4-2）。

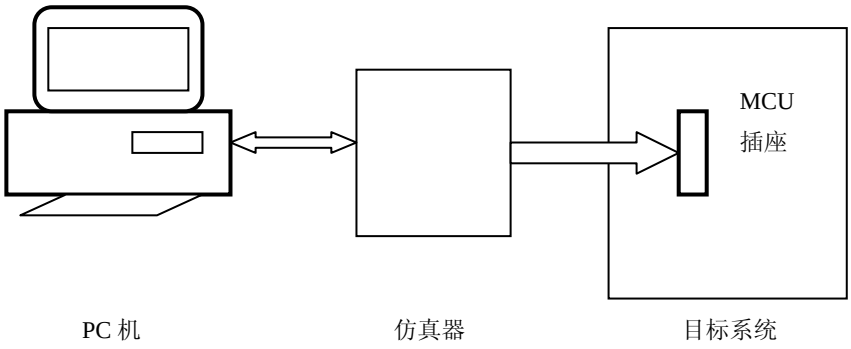

图 4-2　仿真器的连接与使用示意图

仿真器上所提供的 MCU 称为仿真 MCU，它与目标系统上使用的 MCU 是相同系列，或具备相同的功能和特性，其控制作用和工作过程与被仿真的 MCU 几乎完全一样。使用者将编写好的目标系统的软件下载到仿真器中，然后将目标机上的 MCU 取下，插上仿真器的仿真口，仿真器的通信口与 PC 连接（图 4-2）。

在 PC 上需要安装与该仿真器配套使用的专用调试系统软件，用户在该调试系统中，就可以通过 PC 机来控制仿真器中程序的运行，同时观察系统外围器件和设备的运行结果，分析、调试和排除系统中存在的问题。这种运行调试方法称为在线（板）仿真。

为了能实现 MCU 的仿真功能，仿真开发系统通常具有的一些基本功能为：

- 可控性。可以根据调试的需要，控制目标程序的运行方式，如单步、连续、带断点等多种运行方式。

- 透明性。能对 MCU 的各个部分进行监控，如查看和设置内存单元、寄存器、I/O 的数据。

仿真开发系统都必须配备一套在 PC 机上运行的专用仿真开发软件系统，用以配合和实现仿真器的在线仿真调试工作。因此嵌入式系统的开发人员，除了要掌握单片机和嵌入式系统的应用和设计能力，还应熟练地掌握和使用仿真器和仿真系统软件。

实时在板仿真器(ICE)虽然具备实时的跟踪能力，但它最大的缺点是价格昂贵(如 ATMEL 公司的 AVR 在线仿真调试器 ICE50 的价格为 2.5 万人民币左右)，同时与目标板的对接比较困难。尤其面对采用贴片技术，高速的 MCU 构成的系统时，就显得非常不方便。所以 ICE 在过去一般用在低速系统中。

随着软件和芯片技术的发展，实时在板仿真器和相应的调试方法正在逐渐被软件仿真器、实时在片仿真调试（On Chip Debug）方法和实时在片仿真器等其它的形式所替代了。

3. 实时在片仿真器

为了解决实时仿真的困难，新型的芯片在片内集成了硬件调试接口。最常见的就是符合 IEEE1149.1 标准的 JTAG 硬件调试接口。JTAG 硬件调试接口的基本原理，是采用了一种原应用于对集成电路芯片内部进行检测的“边界扫描”技术实现的。使用该技术，当芯片在工作时，可以将集成电路内部的各个部分的状态以及数据，组成一个串行的移位寄存器链，并通过引脚送到芯片的外部。所以通过 JTAG 硬件调试接口，用户就能了解芯片在实际工作过程中，各个单元的实际情况和变化，进而实现跟踪和调试。JTAG 硬件调试接口采用 4 线的串行方式传送数据，占用 MCU 的引脚比较少。

与实时在板仿真器系统一样，采用 JTAG 硬件调试接口进行仿真调试也是实时的在线调试。不同的是，采用这种方式的调试不需要将芯片取下，用户得到的运行数据就是芯片本身运行的真实数据，所以这种调试手段和方式称为实时在片调试（On Chip Debug），并正在替代传统的实时再板仿真调试（On Board Debug）技术。

实现实时在片调试的首要条件，是芯片本身要具备硬件调试接口。除此之外，同实时再板仿真调试一样，也需要一个专用的实时在片仿真器（采用 JTAG 硬件调试口的，称为 JTAG ICE），不过同实时再板仿真器相比，它的价格就便宜了。例如一台应用于 AVR 的 JTAG 仿真器 JTAGICE mkII，其原装价格仅在两千元左右，而国内推出的 JTAG ICE 仅数百元。

使用实时在片仿真器进行系统调试时，其系统的组成和连接方式与使用实时再板仿真器类似，见图 4-2。JTAG 仿真器一般也是使用串行口（COM 口或 USB 接口）或并行口（打印机口）同 PC 机通信，不同之处在于，另一端的接口是直接与目标机系统上 MCU 芯片的 JTAG 引脚连接，不需要将芯片从系统上取下。

在 PC 上也需要安装与相应的 JTAG 仿真器配套使用的专用调试系统软件。在目标板上的 MCU 运行时，用户可以通过 PC 机来读取和跟踪 MCU 的运行数据和过程，并通过仿真器控制 MCU 的运行，同时观察系统外围器件和设备的运行结果，分析、调试和排除系统中存在的问题。由于这种运行调试方法过程中，直接获得的为真实的 MCU 的数据和状态，所以称为实时在片仿真调试技术。

在 AVR 中，大部分 Mage 系列芯片都支持 JTAG 硬件调试口。而对于引脚数少的 tiny 芯片，则使用了一种新的单线硬件调试接口技术，“debug-ware”。顾名思义，它只使用了一根线，就能将芯片内部各个部件的工作状态和数据传送到外部，比 JTAG 使用了更少的接口引脚。

4. 编程烧入器

编程烧入器也称为程序烧入器或编程器，它的作用是将开发人员编写生成的嵌入式系统的二进制运行代码下载（写入）到单片机的程序存储器中。高档的编程器一般称作万用编程器，它不仅可以下载运行代码到多种类型和型号的单片机中，还可以对 EPROM、PAL、GAL

等多种器件进行编程。

目前，性能较好的仿真器也都具备了对其可仿真的 MCU 的编程功能，这样就可以不用专门购置编程器设备。当单片机芯片具备 ISP 功能时，程序的下载更加简单了，一般通过 PC 的串行口或并行口，使用简单的软件就可将编译生成的嵌入式系统的运行代码直接下载到 MCU 中。

现在一些新型的单片机内部集成了一种标准的串行接口 JTAG，专门用于在线仿真调试和程序下载。使用 JTAG，可以简化仿真器（无需使用专用的仿真 MCU）和编程器的结构，甚至可以淘汰专用仿真器和编程器，而将 PC 直接同系统板连接（一般经过简单的隔离），利用系统板上的 MCU 直接实现在线的仿真调试，这为嵌入式系统的设计提供了更为有效和方便的开发手段和方法。当系统使用贴片封装或 BGA 封装的小体积芯片和器件时，它的优点尤为突出。

4.2.4 AVR单片嵌入式系统的软件开发平台

ATMEL 公司为开发使用AVR单片机提供了一套免费的集成开发平台：AVR Studio（http://www.atmel.com）。该软件平台支持AVR汇编程序的编辑、编译、连接以及生成目标代码。同时该软件还内嵌AVR GCC高级语言接口，内含AVR软件模拟器，其仿真调试平台还可以配合ATMEL公司设计推出的多种类型的仿真器，如实时在板仿真器ICE40、ICE50，实时在片仿真器JTAG ICE、JTAGICE mkII等，以实现系统的在线的硬件仿真调试功能和目标代码的下载功能。

此外，一些第三方公司也推出了许多采用高级语言编程的开发平台，用于 AVR 单片机系统的开发。

采用高级程序语言 C 的开发平台有：

- ICCAVR（http://www.imagecraft.com/software）
- CodeVision AVR（http://www.hpinfotech.ro）
- IAR Systems（http://www.iar.com）
- AVR GCC（http://www.avrfreaks.net）

采用高级程序语言 BASIC 的开发平台有：

- BASIC AVR（http://www.digimok.com）
- FastAVR Basic（http://www.fastavr.com）
- BASCOM-AVR（http://www.mcselec.com）。

其中 AVR Studio 和 AVR GCC 是完全免费的软件，而 ICCAVR、CodeVision AVR、IAR System、BASCOM-AVR 等均为商业软件，但它们都有提供给用户试用的 DEMO 版软件（在功能上、时间或代码量上有限制），可以从网上免费下载。在学习单片机嵌入式开发的起步阶段，完全可以使用这些 DEMO 版的开发平台。

本书将介绍 ATMEL 公司提供的 AVR Studio 4.12 的使用。并将以高级程序设计语言 C 为设计手段的 CodeVision AVR(简称 CVAVR)作为本书使用的开发软件。因为采用高级程序设计语言开发嵌入式系统已成为当前的发展趋势。

由于 AVR 单片机具有 ISP 性能，其程序存储器可多次编程、在线下载的优点；加上采用高级程序设计语言来开发单片机系统具有语言简洁，使用方便灵活，表达能力强，可进行结构化程序设计等优点；再配合软件模拟仿真调试；使得我们可以不必购买价格在几千元的仿真器和编程器，就能够很好的学习和掌握 AVR 单片机嵌入式系统的设计和开发。

为配合本书的学习，我们专门设计了一套半开放式的，性能良好，方便学习，制作简便的“AVR-51 多功能单片机系统学习实践开发板”。它不仅便于处学者的学习和实践的使用，

同时也适用工程师作为产品设计和开发的前期使用。建议有条件的学习者，按本书提供的设计和指导，自己动手 DIY 制作，配合本书用于 AVR 单片机的学习和实践使用，这是一个真正的起步。

1. 汇编语言开发平台

ATMEL 公司提供免费的 AVR 汇编语言编译器。在 AVR STUDIO 中已经将 AVR 汇编语言编译器集成在一起，你可以在 AVR STUDIO 中完成 AVR 汇编代码的编辑，编译和连接，生成可下载的运行代码。

由于 AVR 的指令与 C 语言有很强的对应性，再加上 AVR 汇编语言编译器有强大的预编译能力，如宏，表达式计算能力等，所以使用 AVR 汇编语言写出的代码可读性也是很强的。如果你不想话很多的钱在您的编译工具上的话，AVR STUDIO 是一个不错的选择。

另外在 AVR STUDIO 中还提供一个纯软件的软件模拟仿真环境，在此软件环境的支持下，单片机的系统程序可以在 PC 上进行模拟的运行（完全脱离硬件环境），以实现第一步的软件调试和排错功能。

部分第三方的高级语言开发平台不具备软件模拟仿真环境和在线实时仿真的功能，但他们都能够生成在 AVR STUDIO 中可以使用的，用于仿真的文件，这样高级语言的开发平台与 AVR STUDIO 配合使用，就能构成和实现一个基于高级语言的软仿真和在线实时仿真调试的开发环境。

2. 高级语言开发台

由于 AVR 单片机自身的优势，吸引了大量的第三方厂商为 AVR 单片机编写开发出各种各样的 AVR 高级语言编译器和开发软件平台。很少有一个 8 位单片机能有这么众多的编译器以及开发平台可供选择。根据高级语言的种类，AVR 有 C、BASIC、PASCAL，ADA 等多种语言的开发平台。如果您对其中的一种语言比较熟悉的话，那您就不必重新学习另一种语言，而直接选择您熟悉的进行开发。而且这些编译器的厂商在其网站上都提供了免费试用版本的下载，用户可以在试用了一段时间，在比较其之间的优缺点之后，选择购买。

下面就对其中的几种高级语言编译器和开发软件平台进行一个简略的介绍。

（1） IAR Systems 的 Embedded Workbench 编译器

IAR Systems 是非常著名的嵌入式系统的编译工具的提供商。如果您访问其网站，您就会发现它几乎为所有的 8 位、16 位、32 位的单片机和微处理器提供 C 编译器，由此可见也其在业界的地位。正因为如此，当初 ATMEL 在开发和设计 AVR 时，决定咨询 IAR Systems 的编译器设计工程师，商讨如何设计 AVR 的结构，使其对使用高级语言时的编译效率更高。此后，IAR Systems 与 ATMEL 一直保持着良好的而又紧密的合作关系，这使其设计出来的编译器的编译效率也是同类中最高的。但是价格较高。

IAR Systems 的 Embedded Workbench 集成了一个集成环境包括编译器和图形化的调试工具，能够完成系统的设计，测试和文档工作。您可以在其中完全无缝的完成新建项目，编辑源文件，编译，链接和调试等工作。可以同时打开多个项目。很容易扩展集成诸如代码分析等外部工具。

其 C 编译器和汇编编译器支持几乎所有 AVR 芯片，具备以下特点：

- C 编译器支持 ISO/ANSI C 的标准 C 和可选的 Embedded C++编译器。
- 所有代码都可重入。
- 有多种存储器模型和指针类型，以充分利用存储器。
- 内建针对 AVR 优化的选项，多重的代码大小和执行速度的优化控制。

- 针对 AVR 的语言扩展以适应嵌入式编程。
- 新增的强大全局优化器。
- 可以直接在 C/C++中写快速易用的中断处理函数。
- 高效的 32 位和 64 位的 IEEE 兼容的浮点运算。
- 扩展的 C 和 EC++的函数库，并对数学和浮点运算。

IAR Systems的网站地址为http://www.iar.com。

(2) IMAGE CRAFT 的 ICCAVR 编译器

这是 IMAGE CRAFT 提供一款低成本高性能的 C 语言编译器，其包括了 C 编译器和 IDE 集成编译环境，简称 ICCAVR。

其支持除 AT90S1200 外的所有 AT90 系列和 ATmega 系列，Tiny26 和 AT94KFPSLIC 器件，自动生成对 I/O 寄存器操作的 I/O 指令。其编译器是对 LCC 通用 C 编译器的移植，完全支持标准的 ANSI C，支持 32 位的长整数和 32 位的单精度浮点数运算，支持在线汇编，同时也能和单独的汇编模块进行接口。拥有包括 printf，存储器分配，字符串和数学函数的 ANSI C 库函数的子集库函数和针对特定目标访问片上 EEPROM 和各种片上外设的库函数。可以生成用于 AVR STUDIO 源码级调试的目标文件。在其 IDE 中包含了对项目的管理，源文件的编辑，编译和链接源选的设置，还有内嵌的 ISP 编程界面。

但是由于其源自通用 C 编译器，其几乎完全不支持位寻址。

ImageCraft的网站地址为http://www.imagecraft.com，提供 30 天的试用版下载。国内广州双龙公司是ICCAVR的代理商。

(3) HP Info Tech 的 CodeVision AVR 编译器

CodeVision AVR 是 HP Info Tech 专门为 AVR 设计的一款低成本的 C 语言编译器，它产生的代码非常严密，效率很高。它不仅包括了 AVR C 编译器，同时也是一个集成 IDE 的 AVR 开发平台，简称 CVAVR。

CVAVR 支持所有片内含有 RAM 的 AVR 芯片，具备以下特点：

- 支持 bit、char、short、int、long、float 以及指针等多种数据类型，充分利用存储器。
- 内建针对 AVR 优化的多种选项。
- 支持内嵌汇编。
- 扩展的一些标准的外部器件支持和接口函数，如：标准字符 LCD 显示器、I2C 接口、SPI 接口、延时、BCD 码与格雷码转换等。
- 可以直接在 C/C++中写快速易用的中断处理函数。
- 高效的 32 位和 64 位的 IEEE 兼容的浮点运算。
- 扩展的 C 和 EC++的函数库，并对数学和浮点运算。

HP Info Tech的网站地址为http://www.hpinfotech.ro，提供试用板（2K代码限制）的下载。清华大学出版社出版的《嵌入式C编程与Atmel AVR》一书中，对CVAVR的使用和程序设计给出了全面和详细的介绍。本书也采用CVAVR作为主要开发语言平台。

(4) GNU GCC AVR

GNU GCC AVR 是著名的自由软件编译器的 GNU GCC 的 AVR 平台的移植。其包括两部分，编译和链接的命令行程序包和针对 AVR Libc 函数库。如同其他 GNU 协议下的软件一样，所有这些都是以源程序的形式发布，用户可以根据其自身的计算机平台进行配置编译，生成适合用户自身计算机平台的可执行版本的 GNU GCC AVR。对于 WINDOWS 用户，也有已经预先编译好的二进制版本可供下载。

GCCAVR 的特点为：

- 所有源代码都是向用户开放，完全免费。

- GCCAVR 本身支持 ANSI C/C++/EMBEDDED C++。
- GCCAVR 本身的编译效率和稳定性，编译后代码执行效率仅次于 IAR Systems 的 Embedded Workbench。
- 支持几乎所有的 AVR 器件。
- 包括兼容 ANSI C 的部分标准函数库和针对 AVR 的各个外设的函数库。
- 缺乏专业的技术支持，缺乏图形的集成编辑环境（IDE），所有程序都是命令行执行的。

用户可以在http://www.avrfreaks.net上获得最新的GNU GCC AVR软件包。

（5）几种 C 语言开发平台的对比

表 4.1 给出上述 4 种 C 语言开发平台的性能价格对比。

表 4.1 AVR 四种 C 语言开发平台的比较

	IAR	Imagecraft	CodeVision	GNU GCC
代码效率	+++	++	++	++
价格	$$	$	$	Free
易用性	++	+++	+++	+
与 AVR Studio 集成度	++	+++	+++	++
技术支持	+	+++	+++	-

(6) BASCOM-AVR

BASCOM-AVR 是荷兰 MCS Electronics 公司设计的一款针对 AVR 系列单片机的 BASIC 编译器，其软件包由 BACIS 编译器和 IDE 集成编辑环境组成。IDE 集成编辑环境支持对源代码的高亮显示，提供上下文提示，以提高编码效率。IDE 集成编辑环境还包含了一系列工具，图形化的模拟仿真环境，无需连结硬件，你就可以通过它对 LCD，LED，UART，和 PIO 端口进行仿真。此外，你还可以在 IDE 集成环境中对目标板进行 ISP 编程。其主要特点有：

- 采用可带语句标示符的结构型 BASIC 高级程序设计语言编程，程序语句和 Microsoft VB/QB 高度兼容。
- 结构化的 IF-THEN-ELSE-ENDIF、DO-LOOP、WHILE-WEND、SELECT-CASE 程序设计。
- 变量名和语句标示符长达 32 个字符。
- 支持位(Bit)、字节(Byte)、整型(Integer)、字(Word)、长型(Long)，字符串(String)多种类型的变量。
- 编译产生的运行代码可在所有带内部存储器的 AVR 微控制器微中运行。
- 为标准 LCD 显示器，I^2C 芯片和单总线协议芯片等扩充了大量的专用语句。
- 内置模拟终端和程序下载功能。
- 自带内置的图形软件仿真平台，并同时支持和采用 AVR STUDIO 作为其软件模拟仿真器。
- 完善的连机帮助功能和大量的例程。

BSACOM-AVR是采用结构型BASIC作为程序设计语言，简单易学，尤其适合中学生、大中专学生学习使用，以及开发一些相对简单的小系统使用。用户可以到MCS Electronics的网站http://www.mcselec.com下载试用版（2KB代码限制）。清华大学出版社出版的《AVR单片机BASIC语言编程及开发》一书中，对BASCOM-AVR的使用和程序设计给出了全面和详细的介绍，读者可以参考学习。

4.2.5 AVR实验开发板

1. STK500 系列开发板

STK500 系列开发板是 ATMEL 公司推出的一套基于 90 系列和 mega 系列的 AVR 开发评估板和相应的适配板，以使用户能快速入门和了解使用 AVR 芯片。用户在产品设计过程也可以在这些开发板上完成初步的验证，而免去了自己制板的成本与风险。STK500 系列评估板包括 STK500 主板和 STK501、STK502、STK503、STK504、STK505、STK520 等子板。

STK500 是 ATMEL 推出的主要针对 40 脚及 40 脚以下的 DIP 封装的 90 系列和 mega 系列单片机的评估开发板。其具有高压并行和 ISP 编程功能以及 JTAG 仿真接口，同时还配备了一些 LED 和按键，它们可以通过扁平线和单片机的端口连接，用于观察端口的电平变化或者手动触发端口电平的变化，这在没有仿真的情况下是非常有用的。在板上除了一个用于和下载程序的 RS232 接口外，还有一个 RS232 接口，通过跳线可以和单片机的 UART 连接，完成与 PC 机进行通信的任务。另外，板上还有一个振荡电路，您可以根据自己的需求选择不同的时钟源驱动您的单片机。图 4-3 所示为 STK500 开发板。

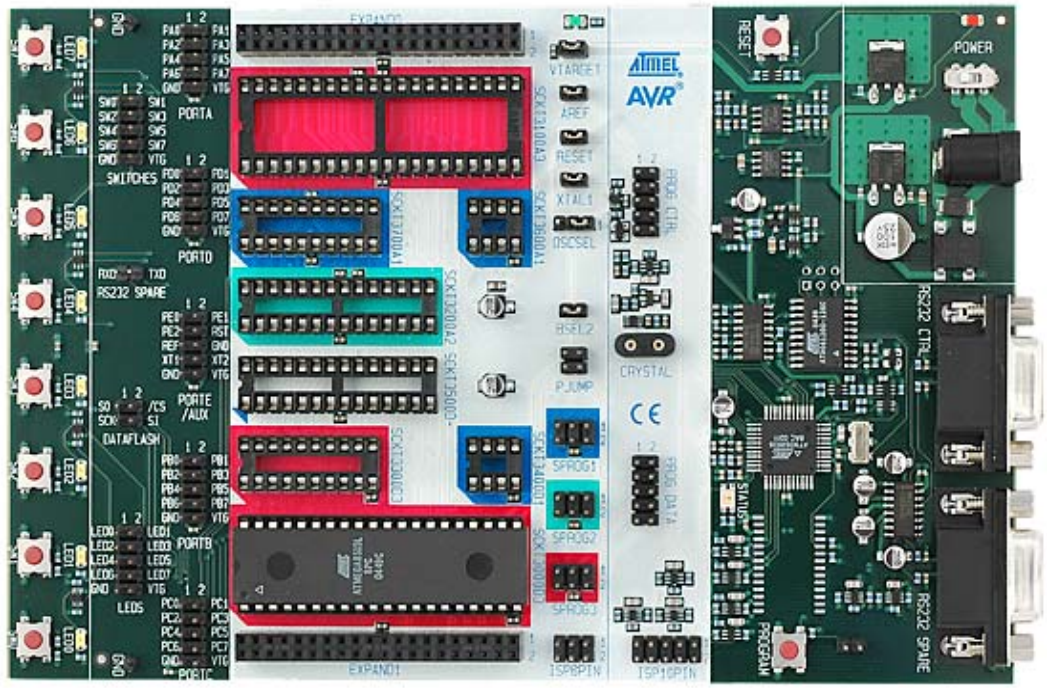

图 4-3 STK500 开发板

由于许多 AVR 芯片采用 TQFP 的贴片封装，不能直接在 STK500 上使用，所以 STK500 还配有多种不同形式的顶置模块子板，以适合各种封装形式的 AVR 使用。图 4-4 为顶置模块子板 STK501，专门应用于 TQFP64 脚的 AVR（ATmega103/ATmega64/ATmega128）使用。

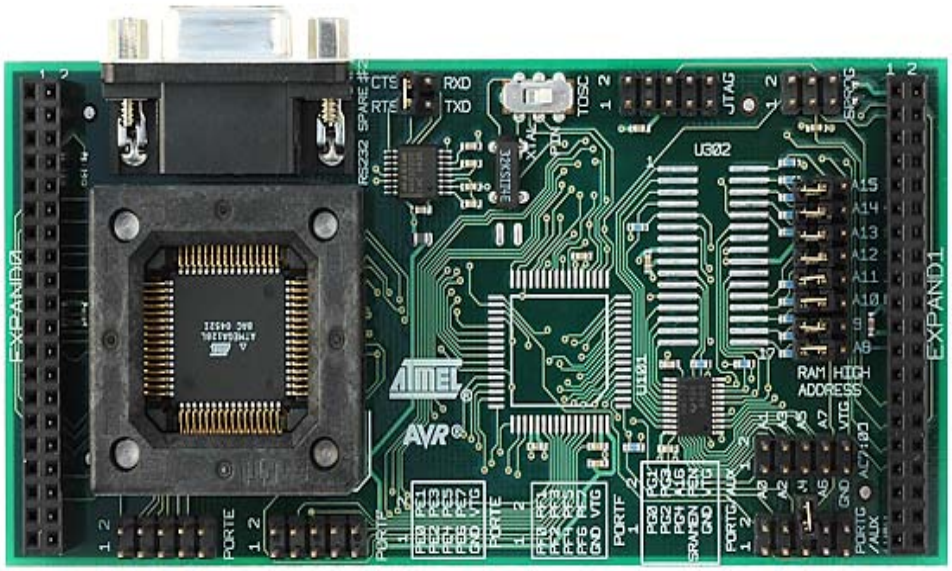

图 4-4 STK501 子板

STK501 作为 STK500 的子板，配有安装 ATmega103/ATmega128 的 ZIF 插座和 PCB 封装。它需要安装在 STK500 上才能完成对 ATmega128 开发功能（如图 4-5）。此外，由于 ATmega128 有两个 USART 口，所以在 STK501 上还扩展了一个额外的 RS232 口，以及 32kHz 的 RTC 振荡器。

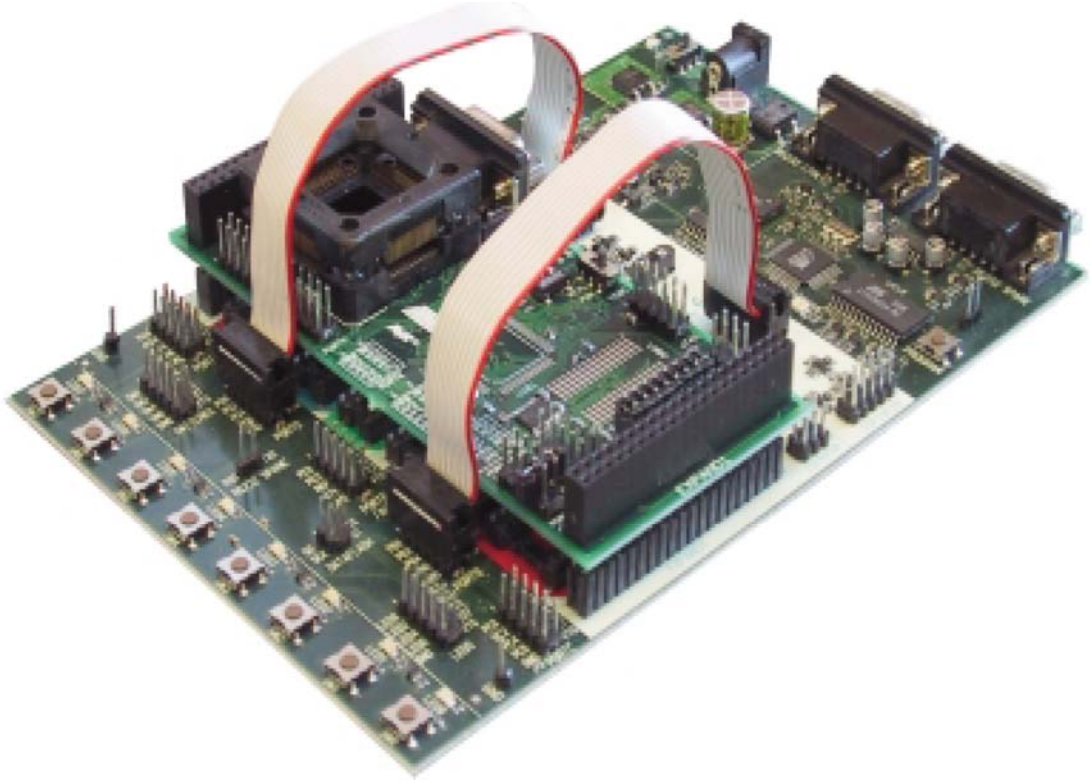

图 4-5 使用 STK500 + STK501 开发 ATmega128

STK500 配备的顶置模块子板有 STK501、STK502、STK503、STK504、STK505、STK520 等。它们都必须与 STK500 配合，以适合各种封装形式和不同型号的 AVR 使用。用户在 AVR Studio 软件环境的的在线帮助中，可以了解到它们的具体特点和使用方法。

2．AVR-51 多功能实验开发板

AVR-51 多功能实验开发板是华东师范大学电子科学技术系 AVR 实验室基于“模块独立、开放、灵活”的思想而设计的，适用于初学者使用的多功能实验开发板。

传统的单片机实验系统透明度不高，实验板上的芯片、接口都是固定连接的，硬件只需很少连接，甚至不需要做硬件的连接，就直接进行编程设计，这就把单片机以硬件为主，软硬结合的训练变相的转化成了纯软件的训练。

单片机系统具有“硬件决定软件，程序基于硬件”的特点。在外表上实现相同功能的系统，其硬件设计往往有多种形式，系统软件的设计也不同。硬件设计的不同，如接口一旦改变，口地址就会变化，程序也就要跟着变。

“AVR-51 多功能实验开发板”采用“模块独立、开放、灵活”的设计思想，本着能够使学习者在硬件设计和软件设计全面得到真正的训练和学习的目，一改以往的固定线路、固定接口、固定芯片的模式，对开发板上的硬件资源采用了全部开放的结构，同时还提供比较丰富的接口，以及构成单片机系统最常使用的同时也是最基本的外围功能电路。板上的单片机引脚全部开放，同时将在单片机系统中最常使用的显示、按键、键盘、等都作为独立的开放单元模块，其连接信号接口和电源接口也是开放的。这样用户就可以非常灵活的，根据需要构建自己的系统。

“AVR-51 多功能实验开发板”不仅可以用于配合本书教学实验，而且也适合单片机系统的设计开发人员作为产品开发前期使用的开发板。该板由国内“我们的 AVR”网站批量生产（http://www.ouravr.com），读者可以从该网站邮购。

实验板分成三大部分：系统电源、MCU 座和外围功能模块单元。其中外围功能模块单元分成 A—O 共 15 个区域。图 4-6 为“AVR-51 多功能实验开发板”的实物图，附录 B 中给出了该板详细的电原理图。有兴趣和能力的读者也可以参考该电路，自己制作实验板。

下面对该板的功能和特点进行一些介绍，以方便今后的使用。

1）系统电源。在 8-12V 输入电压范围内提供高稳定的 5V/1A 的系统电源，配有电源指示灯、极性保护电路及开关。一般工作时，输入电压为 9V，系统电源部分使用 7805 为实验板工作提供需要的 5V/1A 电源。同时板上有多个高频和低频的电源滤波电容。

2）MCU 座。实验板的中间部分有两个 40PIN 的锁紧插座，供插入 MCU 使用。

- ✓ 2 个 40PIN 锁紧插座引脚全部是开放的，与外围没有任何的连接。
- ✓ 左边 40PIN 锁紧插座与 JU1 短路排（6 组）、X2 和 X1 短路排（2 组）、JU2-GND 和 JU3-Vcc（2 组）配合，构成与通用 51 系列 40PIN 单片机引脚兼容的方式，适用于 AT89S5x 系列的单片机以及 AVR 的 ATmega8515 单片机使用。
- ✓ 右边 40PIN 锁紧插座与 JU4 短路排（10 组）、JU7 短路排（3 组）、JU8（4 组）配合，适用于 AVR 的 ATmega16、ATmega8535、ATmega32 单片机。
- ✓ 左边 40PIN 锁紧插座顶部的 2*5 针的插座为 ISP 下载接口，用于配合与 STK200/300 兼容的 ISP 下载线实现对 AT89S5x 和 AVR 单片机进行程序下载和熔丝配置。
- ✓ 右边 40PIN 锁紧插座的右下角处的 2*5 针的插座为 JTAG 接口，用于配合使用专用的 JTAG 仿真器实现对含有 JTAG 接口的 AVR 单片机进行在片（On Chip Debug）实时仿真调试，程序下载以及熔丝位的配置。
- ✓ 由于 2 个 40PIN 锁紧插座引脚全部是开放的，因此当 2 个锁紧插座中的任何一个插入单片机后，另一个就可作为扩展插座使用，如插入 DIP 封装的 EEPROM 芯片 24C02 做 I2C 通信实验。
- ✓ 原则上该板可以适合任何 DIP 封装，引脚在 40PIN 以内的单片机的使用。但此时需要用户使用更多的连接线进行必要的连接，同时还要考虑匹配适当的编程器，以及该单片机 I/O 口的驱动能力问题（>10mA）。

✓ 该板最适合 40PIN 以内 DIP 封装的 AVR 单片机和 AT89S5x 系列的 51 单片机使用。因为它们使用相同的 ISP 编程技术，配合与 STK200/300 兼容的 ISP 下载线都能实现程序的下载，不必另外配置专用的编程器。同时他们的 I/O 驱动能力都大于 20mA，可以直接驱动 LED 显示。

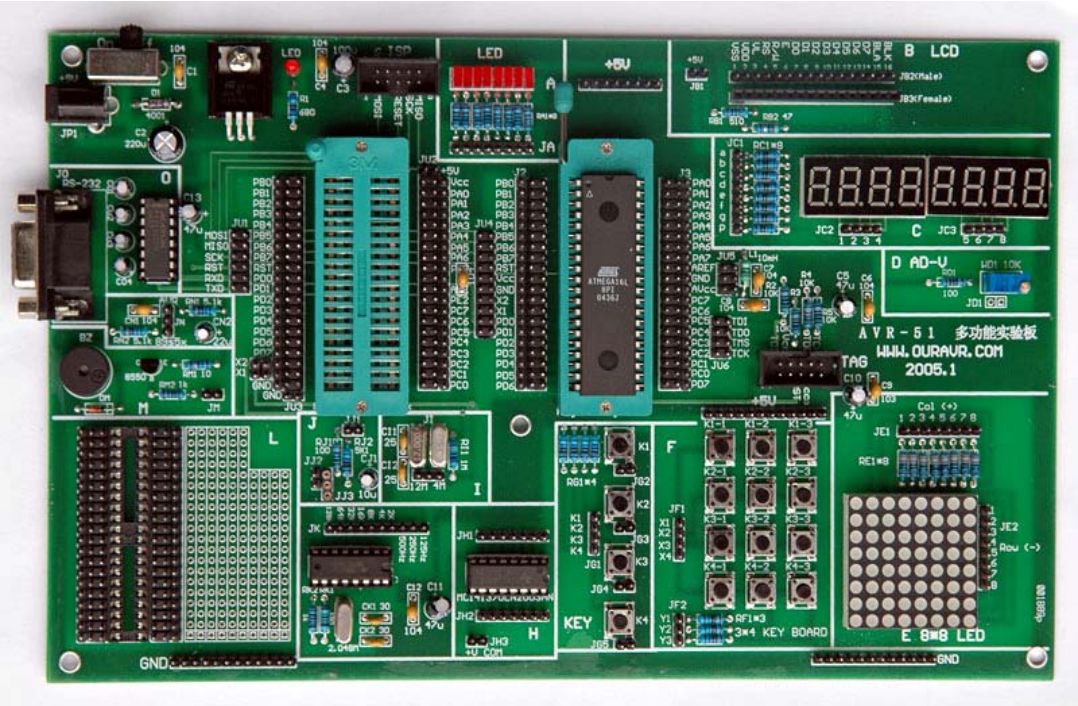

图 4-6 AVR-51 多功能实验开发板

3）使用 ATmega16 的缺省连接。在“AVR-51 多功能实验开发板”上对使用 ATmega16 采用缺省的短路片连接，构成最小系统。

将 ATmega16 插入右边的 40PIN 插座中锁紧。将 JU4 短路排（10 组）、JU7 短路排（3 组）、JU8（4 组）共 17 组短路排用短路片短路。同时将 JN 短路跳针的中心与上方的 AVR 短接。17 组短路排的作用见表 4.2。

表 4.2 使用 ATmega16 时短路片的缺省连接

	连　接		说　明
JU4	ISP MOSI	PB5	与 ISP 接口连通，用于 ISP 方式的下载程序和配置熔丝位。如在实验中 PB5、PB6、PB7 还作为 I/O 使用时，注意不要直接与 5V 电源或地连接，防止与 ISP 发生冲突，造成无法进行程序下载
	ISP MISO	PB6	
	ISP SCK	PB7	
	ISP RESET	RST	
	+5V	Vcc	提供芯片的工作电源
	GND	GND	芯片接地引脚
	外部晶体	X2	当使用芯片内部 RC 振荡源时，X2、X1 的短路片可以不用。
	外部晶体	X1	
	MAX202 R2out	PD0	当不使用芯片的 USART 功能时，这 2 个短路片可以不用，

	MAX202 T2in	PD1	PD0、PD1 可作为普通 I/O 使用
JU7	C7	AREF	ADC 参考电压输入端。确省接 C7 滤波电容，使用内部参考
	GND	GND	芯片接地引脚
	+5V	AVcc	提供芯片 A 口（ADC）的工作电源，通过 L1、C8 滤波。
JU8	JTAG TDI	PC5	与 JTAG 接口连通，用于 JTAG 方式的在线仿真调试，以及下载程序和配置熔丝位。如不使用 JTAG 方式，这 4 个短路片可以不用，此时 PC5、PC4、PC3、PC2 作为普通 I/O 使用
	JTAG TDO	PC4	
	JTAG TMS	PC3	
	JTAG TCK	PC2	
JN	**JN 的中心针与 AVR 针短接**		**AVR 采用低电平复位，正常工作 RESET 接高电平**

4) 外围功能模块单元。外围功能模块单元分成 A—O 共 15 个区域。

A 区，8 路 LED 发光二极管，用于输出显示。

B 区，标准 2*16 字符的 LCD 液晶显示器接口，同时还兼容 3310 图形液晶显示器接口。3310 图形液晶显示器可以显示 84*48 点阵图形，能显示 3 行中文，每行 7 个汉字。

C 区，8 位 8 段 LED 数码管显示器，采用共阴接法，动态扫描方式点亮。

D 区，这是一个由精密电位器与电源组成的 0～5V 的可调直流电压源，可提供 0～5V 可调直流电压信号，作为 ADC 的输入电压源。用于实现 AD 转换、直流电压表、模拟温度计等实验。

E 区，8*8LED 点阵显示模块，用于做点阵字符、小型广告屏，电梯运行指示器等实验。

F 区，4*3 矩阵键盘。

G 区，4 个独立按键。用于按键输入、外部中断输入等。使用 JG1-4 短路片可以将一位 I/O 口对地短路。连接 RESET 线，可以实现人工复位。

H 区，7 路 300mA 功率驱动。用于驱动小型步进电机、继电器等。

I 区，系统时钟选择。当单片机使用外部晶体振荡时，可选择 4M 或 11.0592M 的晶体。使用 11.0592M 晶体可以产生高精度的 RS-232 通信波特率。

J 区，该区电路可变化为 2 种应用。其一，作为 RC 滤波电路用于对 PWM 输出的平滑。其二用于使用 DS1802 数字温度传感器做单总线实验。

K 区，该区模块使用一个 2.048MHz 的晶体振荡器，经过 CD4060 的分频，提供 125Hz～128KHz 之间，占空比 50%的 10 种频率方波脉冲信号，可作为频率、周期测量实验的输入信号，外部计数、外部中断等输入信号。

L 区，40PIN 窄型扩展座和标准 PCB 板。用于扩展和插入其它外围芯片。

M 区，该区提供一个无源蜂鸣器，由 MCU 的某一引脚输出一定频率的方波，就可以发出声响，可以作为一个简单的外设。

N 区，单片机外部复位电路选择。由于 AVR 系列单片机采用低电平复位，而 51 系列单片机为高电平复位，因此需要根据使用单片机的类型正确的选择复位方式。选择通过 JN 短路片确定。

O 区，RS-232 串行接口单元。通过 MAX202 电平转换，连接单片机的 USART（UART）口，实现与 PC 的异步通信。

综上所述，整个实验开发系统板提供了一下功能：

- ✓ 系统的资源与能源

电源供电系统单元、可调直流电压单元（D）和脉冲信号发生器单元（K）为实验提供了必要的条件和手段。电源供电系统为实验板提供了高精度的电源；D 单元为 A/D 转换实验提供信号；K 单元则为计数、频率测量等提供了信号源。

✓ 基本的输入输出设备或接口

这部分主要为基本实验提供必要的外围设备：4 键按键单元（G）、4×3 键盘单元（F）、无源蜂鸣器单元（M）、8×8 LED 点阵式显示单元（E）、8 位 LED 数码管显示单元（C）、2×16 字符型液晶显示单元（B）、LED×8 显示单元（A），功率驱动单元（H）以及 RS-232（O）等。

✓ 系统的扩展和多功能

单片机引脚的全部开放，采用 2 个 40PIN 的锁紧插座，扩展插座(L)、系统时钟选择(I)、外部复位选择（N）等使得实验板可以非常灵活的扩展、组合，同时也适合多种类型以及不同引脚数的单片机使用。用户能够根据需要，采用不同的连接方式，构成新的系统电路或新的 MCU 系统，以兼容更多的实验和应用的需要。

由于 I2C 总线、SPI 总线、单总线（1 WIRE）通信接口均是串行通信方式，使用连接线少，因此在做这些实验时，只要将相关的外围芯片插入扩展座中，再使用几根连接线联上电源、地，和接口引脚就可以了。

对于 AVR 单片机所有的基本功能和单元实验，如：I/O 使用、ADC、时钟、中断、PWM，键盘、按键、LED、LED 数码、LCD 显示、测频率、测周期（利用板上的 125Hz-128K 的方波源）、功率驱动、蜂鸣器、RS-232 等，都可以在板上实现。

如果将这些单元有机的组合，就可以完成和实现一些实际电子产品的设计，如带音乐报时的实时时钟、秒表、频率计、速度表、电话拨号器、电压表、LED 广告屏，计算器.....。

4.3 自制ISP下载电缆

在本节中，我们将介绍一个与 STK200/300 兼容的 ISP 下载电缆。该下载电缆支持所有使用 ISP 技术的 AVR 芯片，同时也支持 ATMEL 公司 51 系列兼容芯片 AT89S51、AT89S52、AT89S53、89S8252。图 4-7 为 AVR ISP 下载电缆电原理图。

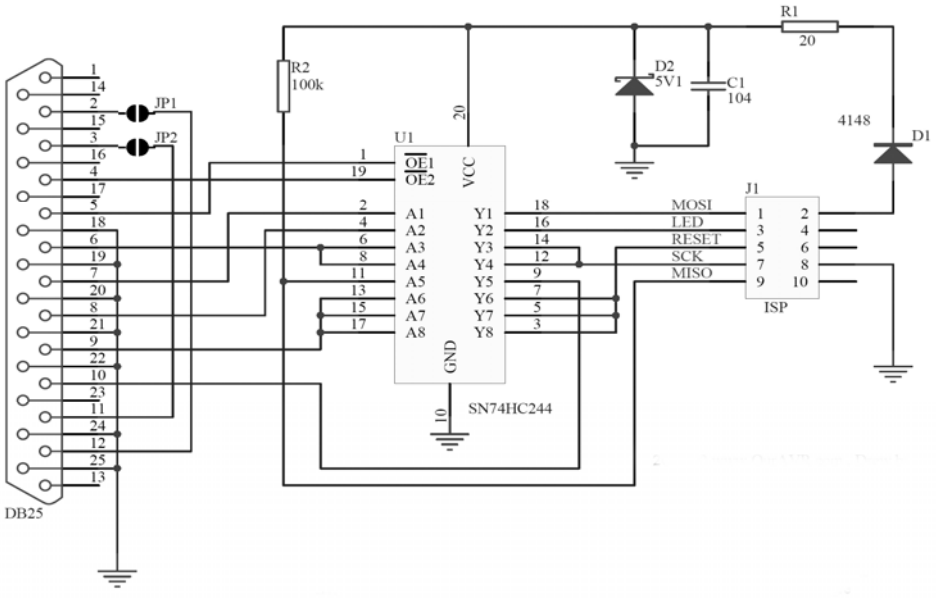

图 4-7 AVR ISP 下载电缆原理图

ISP 编程的原理是使用 PC 机的并行口来驱动 ISP 编程所需的信号波形，实现对 AVR 的程序下载和熔丝位的配置编程。出于安全的考虑，为了防止使用中误操作而损坏 PC 机的并行口，图中使用高速器件 74HC244 作为缓冲，以保护计算机的并行口。74HC244 由目标板供电，VTG 经过 D1（极性保护）和 D2（5.1 V 限压保护）使 74HC244 工作在+3~+5V。R1 为 MISO 信号线的上拉电阻。

制作 AVR ISP 的成本非常低，采用贴片封装器件时整个电路板可以按装在一个普通 DB25 的接口盒中（图 4-8）。配备这样一根下载电缆，一台配备了相应开发软件的 PC 机，再加上开发目标系统板，一套基本的 AVR 软硬件开发环境就建立起来了。接下来需要的，就是发挥你的聪明才智和锻炼你的实际动手能力了。

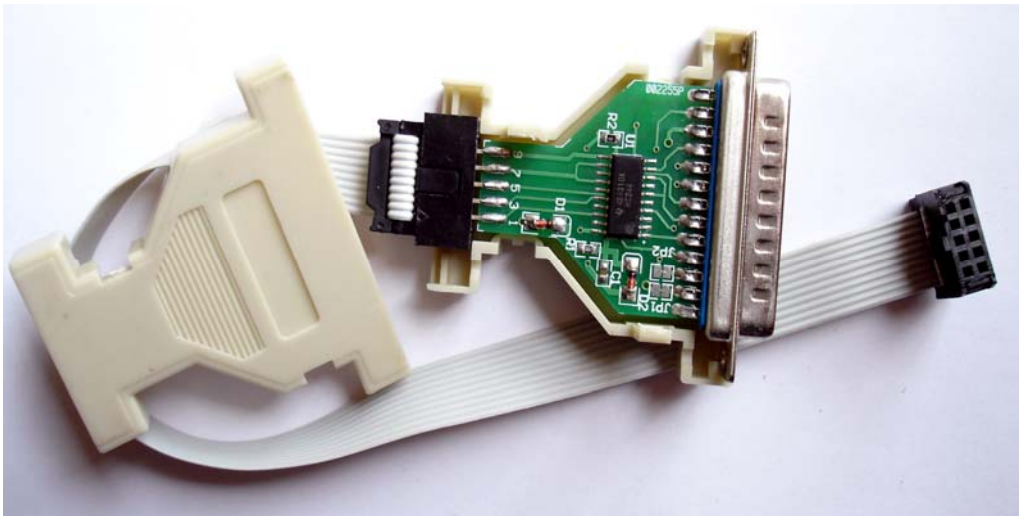

图 4-8 AVR ISP 下载电缆实物图

要使用 AVR ISP 编程电缆下载程序还需要相应的编程软件支持。本节介绍的 AVR ISP 下载电缆与 STK200/300 完全兼容，在 ICCAVR、CVAVR、BASCOM-AVR 等开发平台的程序下载单元中，都直接支持使用 STK200/300 的下载编程（可惜，ATMEL 的 AVR STUDIO 不支持使用 STK200/300 下载电缆）。此外，也可以在网上下载一些专用的免费编程软件使用。

（1） 在 ICCAVR、CVAVR、BASCOM-AVR 中的使用

在 ICCAVR、CVAVR、BASCOM-AVR 等 AVR 高级语言开发平台中，都有内置的程序编程下载功能模块，支持使用多种不同形式的 ISP 下载电缆（编程器）实现对 AVR 的编程操作。STK200/300 是他们支持的下载电缆之一。在下载程序前，你只要在相应的 Programmer 的选项栏中选定使用 STK200/300，就可在程序正确编译后直接将运行代码下载到 AVR 芯片中了，非常方便。

ICCAVR、CVACR 的编程速度比 BASCOM-AVR 快一些。但在 BASCOM-AVR 中对 AVR 芯片熔丝位的编程配置界面更加体现了以人为本的原则（图 4-9），它提供简洁的说明选择，而其它编程软件对 AVR 芯片熔丝位的编程配置界面使用不方便，需要用户仔细查阅手册后作出选择。因此，建议用户对 AVR 芯片熔丝位进行编程配置时，使用 BASCOM-AVR 进行选择和操作。

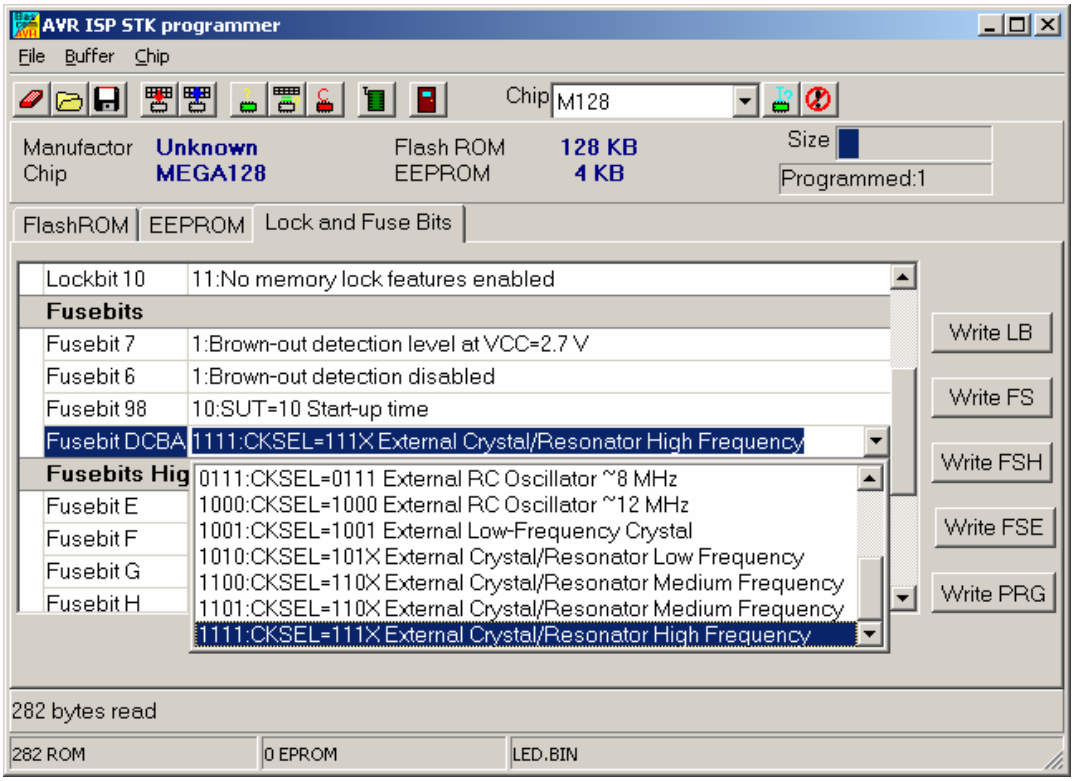

图 4-9 BASCOM-AVR 熔丝位编程配置窗口

（2） 免费通用编程软件 PonyProg。

PonyProg 是一个通用串行器件编程软件，配合的特定的编程硬件（电缆），它支持对 I2C 总线（24 系列），Microwire 总线（93 系列），SPI 总线（25 系列）的 EEPROM 存储器，ATMEL 的 AVR 单片机（支持使用 STK200/300 下载电缆），MICROCHIP 的 PIC 单片机的进行读写编程。

你可以在http://www.lancos.com/prog.html下载最新的PonyProg。同时该网站上还提供其支持的编程器的硬件电路图，你可以参考，并制作一个多功能的编程器。

（3） 免费编程软件 SLISP

SLISP是国内广州双龙公司推出的免费ISP编程软件，它也可以配合STK200/300 实现对AVR的下载编程，同时也能对AT89S5x编程。由于该软件为国内公司推出的，提供了中文板的操作界面，非常适合国内用户的使用，也是本书推荐使用的软件之一。该公司的网站地址为：http://www.sl.com.cn。

4.4 AVR开发环境的建立

单片嵌入式系统应用和开发的学习，不能仅仅依赖于从书本上学习原理和理论，更主要的学习环节是动手实践。单片嵌入式系统的应用开发技术是一门实战（践）性很强的学科，也是一门综合性的学科，最好的学习方法是边学习、边实践、边总结和归纳。

在学习了 AVR 单片机的基本结构和汇编指令以后，我们对 AVR 单片机已经有了一个理论上的认识和初步的了解。进一步的学习和使用 AVR 单片机就应该与实践相结合，动手实践是真正学好和掌握单片机与嵌入式系统设计和开发的必要途径。

4.4.1 AVR单片机实验开发环境

一个 AVR 单片机的基本学习和开发环境由以下几部分组成：

1. PC 机一台，运行 Windows 操作系统

个人电脑 PC 是 AVR 嵌入式系统设计开发的主要工具之一。一般的讲，一台 586 以上，运行 Windows98/Windows2000/Windows XP 操作系统的个人电脑就可以了。

2. AVR 软件开发平台

一般需要选择一个或两个 AVR 软件开发平台。如采用汇编语言来开发 AVR 的系统程序，则首选 ATMEL 公司免费提供的 AVR Studio。喜欢和习惯采用高级语言开发系统程序，可以选取 C 或 BASIC 的开发平台。

3. AVR 实验开发板（Developing Kit）

AVR 实验开发板是系统实现的硬件环境。在开发板上，除了具有可供使用的 AVR 芯片，还提供电源，基本的外围器件（LED 发光管、LED 数码管、LCD 显示器、按键……），通信接口器件，通信连接线，用于执行代码程序下载的连接线等等。它方便用户实际动手学习、调试和检验自己的设计。因此，一块开发板对于学习是不可缺少的。

4. 其它辅助工具、设备和软件

在 AVR 嵌入式系统的开发过程中，一些必要的辅助工具和设备有：万用表，示波器，信号源，频率计等。工具软件有：串口调试软件，执行代码程序的下载编程软件等。

以上我们没有提到 AVR 的在线实时仿真器。由于 AVR 单片机具备 ISP 功能，以及大多数的 AVR 软件开发平台都与 AVR Studio 配合，使用 AVR Studio 中的软件模拟仿真功能，因此对于学习以及开发一些普通的系统，基本可以不必购买价格比较昂贵的 AVR 在线仿真器。当然，在开发一个比较复杂的系统时，手头配备一台 AVR 在线仿真器也是有必要的。

4.4.2 本书中采用的AVR单片机实验开发环境的建立

在本书中，我们推荐和使用以下的软硬件环境作为开始学习和使用 AVR 单片机的手段和工具，开发环境有以下部分构成：

- PC 机一台，运行 Windows 2K 或 Windows XP 操作系统；
- AVR Studio 4.12(http://www.atmel.com，Free)。汇编开发，软件模拟调试。
- AVR高级C语言开发平台CVAVR（DEMO版）(http://www.hpinfotech.ro，Free)。C 语言开发。
- BASCOM-AVR（DEMO版）(http://www.mcselec.com，Free)。用于熔丝位的配置编程和程序下载。
- SLISP（http://www.sl.com.cn)。用于熔丝位的配置编程和程序下载。
- 串口调试精灵。用于 PC 机与单片机之间的 RS-232 通信，软件调试等。
- AVR-51 多功能实验开发板一套。目标实验板。

以上软件都是免费软件，用户可以网上下载，或从本书附带的光盘中获取。软件的安装都比较简单，用户可以根据安装提示进行安装。

思考与练习

1. 学习单片机嵌入式系统的原理与应用开发，应具备和掌握哪些方面的基础知识和技能，为什么？
2. 为什么仅通过书本和课堂是不能学好和掌握单片机嵌入式系统的原理与应用开发的？
3. 简单讲述单片机嵌入式系统的开发过程和步骤，并说明在开发过程中要使用的主要硬件和软件工具是什么。
4. 硬件仿真器和程序烧入器的作用是什么？
5. 一个好的单片机软件开发平台应具备那些必要的功能？
6. 使用汇编语言和高级程序设计语言编写系统程序各有何优点和不足？
7. 通过网络、杂志与广告了解国内外主要的单片机生产商，它们的单片机产品型号，以及相应的开发系统和工具的名称和价格。
8. 本书推荐的学习 AVR 嵌入式系统开发的实验开发环境包含哪些硬件与软件？有何特点？
9. 建立一个学习 AVR 嵌入式系统开发的实验开发环境，熟悉 AVR-51 多功能实验版的硬件电路图与实际的连接，下载和安装软件开发系统和工具软件。
10. 仔细阅读各个软件的使用说明(Online-Help)，熟悉软件的使用环境和主要功能。
11. 在本书附带的光盘中有 AVR Studio、CVAVR 等软件使用参考的中文翻译电子文档，可以作为辅助参考资料。但最好阅读英文原文，可以得到更详细和准确的帮助。

本章参考文献：

1.《AVR Studio在线帮助》(中文，CDROM)，ATMEL，www.atmel.com

第 5 章 实战练习一

本章的实战练习将以一个最简单的设计为例，指导读者完成以下的实践：

- 如何使用 AVR 汇编语言进行系统程序设计与系统实现。
- 初步掌握使用 AVR 免费开发平台 AVR Studio。在该开发平台的支持下，完成汇编源程序的编写，以及程序的软件模拟调试等开发的过程。
- 掌握 AVR-51 多功能实验板使用方法。完成实现硬件系统电路的连接，如何使用 ISP 下载线配置 AVR 的熔丝位，以及运行代码下载。
- 初步掌握 CVAVR 高级 C 语言开发软件的使用。

作为动手实践的一个起步，学习者通过该示例的完成和实现，可以对使用汇编程序语言开发以及 C 语言开发单片机嵌入式系统的过程与特点，以及相关的硬件和软件工具有一个基本的了解。

5.1 秒节拍显示器系统的设计

5.1.1 秒节拍显示器硬件设计

在第 2 章的 2.6.6 中，给出了一个使用 ATmega16 构成的 AVR 简单的系统。这个系统就是一个简易的“秒节拍显示器”。这个秒节拍显示器的功能非常简单，就是用 AVR 单片机控制一个 LED 发光二极管，让它亮一秒钟，暗一秒种，不间断的闪烁，构成一个简单的秒节拍显示器。图 5-1 是它的电原理图。

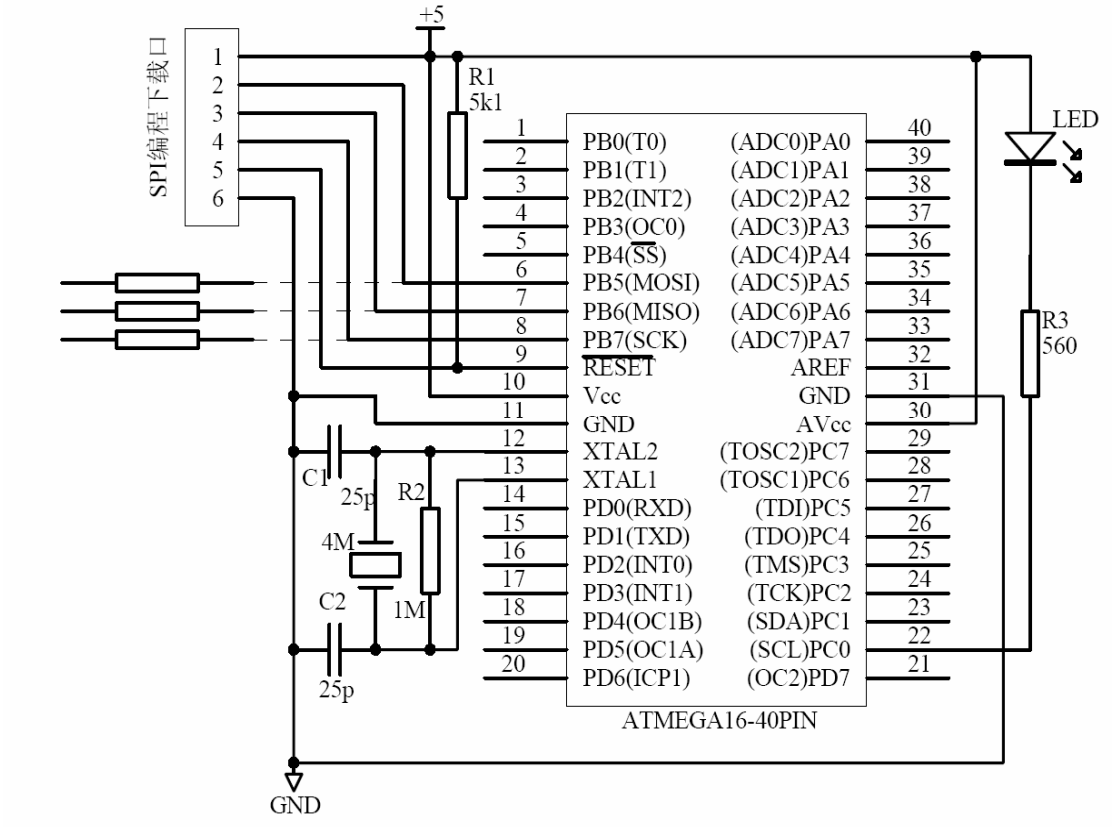

图 5-1 简单的秒节拍显示器电原理图

秒节拍显示器的硬件电路组成非常简单，图中使用一个 AVR 芯片和 LED 发光二极管作为秒信号的显示。当 ATmega16 的 I/O 引脚 PC0 口输出为“0”时，LED 导通发光；输出为“1”时，LED 截止熄灭。560 欧姆电阻起保护限流作用，控制 LED 的导通电流在 5～10mA。适当调整 R3 的阻值，可以调节 LED 的亮度，并限制流过 LED 和 PC0 口的电流，保护其不被大电流烧毁。

在虚线框中，是最小系统的构成。其中 R1 为 RESET 脚的上拉电阻，保证了 RESET 脚可靠的高电平。系统采用外接 4M 晶体和芯片内部的振荡电路组成时钟电路，产生 4M 频率的脉冲作为系统的时钟信号，此时单条指令的执行时间为 0.25us。电容 C1 和 C2 应同具体使用的石英晶体配合（参考具体生产厂的说明），一般在 20p-30p 之间，改变 C1、C2 的值，可以对 4M 频率进行微调。R2 与晶体并连，其作用是稳定晶体的阻抗，提高振荡电路的稳定性。

图中的 ISP 编程下载口的 2、3、4、5 脚同芯片 SPI 接口的 MOSI（PB5）、MISO（PB6）、SCK（PB7）和 RESET 引脚连接。当需要改动 AVR 的熔丝位配置，或将编译好的运行代码烧入的 AVR 的 FlashROM 中时，就不需要将芯片从 PCB 板上取下了。只要将一根简单的编程线插在该编程下载口上，利用 PC 机就可以方便的实现上面的操作了。

如 2.6.5 中所介绍，当 PC 机对 AVR 编程时，需要先将 SCK 和 RESET 引脚拉低，使 AVR 芯片进入 SPI 编程状态，然后通过 SPI 口进行下载操作。所以，在设计 AVR 系统硬件时，如考虑使用 SPI 口实现 ISP 的功能，图中的 R1 电阻不可省略。此时 R1 起到了隔离作用，正是有了 R1，才能使用户在外部能够对 RESET 脚施加低电平（0 伏）。当编程下载完成后，外部一旦释放掉 RESET，该引脚通过 R1 又被拉成高电平，AVR 就直接进入了正常运行工作状态。R1 的阻值在 5k-10k 之间，太大和太小都不合适。

由于 ATmega16 内部集成了 1/2/4/8M 四种频率的 RC 振荡源，因此图 5-1 还可以简化。我们可以使用片内 4M 的 RC 振荡电路作为系统时钟源。这样就可以省掉 C1、C1、R2 和晶体四个元件，使 AVR 的最小系统更加简单，只需要一个 R1 就可以了。

需要注意的是，用户首先必须正确的设置 ATmega16 的 4 个熔丝配置位 CKSEL3..0，使它们的组合设置与你实际使用的系统时钟类型相配合。

5.1.2 秒节拍显示器软件设计思路

图 5-2 为秒节拍显示器的系统软件流程图。可以看出，秒节拍显示器的软件设计重点是一个一秒钟的延时子程序。系统程序每隔一秒（调用一秒延时子程序）将 PC0 口的输出电平取反，同时也控制 LED 的亮与暗。

作为一个简单的入门例子，在这里我们给出一个采用汇编语言设计编写的通用软件延时子程序，每调用一次该子程序，其运行的时间为 1 秒钟，每隔 1 秒钟，控制 PC0 口的输出逻辑取反。这样 LED 就会亮 1s，灭 1s，实现了秒节拍的显示。

实际上，在实际应用中尽量不要使用软件的方式进行延时，因为 CPU 执行大量的无具体工作的指令，这样会减低 CPU 的效率。正确的方法是使用 AVR 的定时器来产生延时，这些将在后续的章节中介绍。

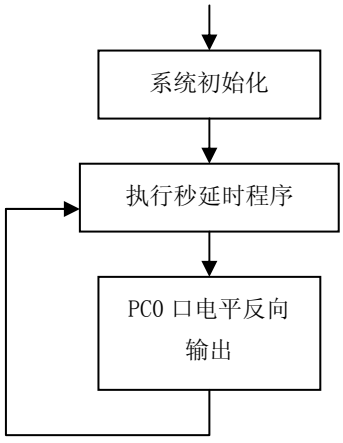

图 5-2 秒节拍系统软件流程图

5.1.3 秒节拍显示器汇编源程序代码

下面是秒节拍发生器的汇编源程序。程序中先初始设置堆栈指针寄存器 SP 的值，然后将 PC 口定义为输出。其主程序部分为一个 LOOP 无限（死）循环：先设置 PC0 口输出“0”，点亮 LED；调用延时子程序 delay 延时一秒；再设置 PC0 口输出“1”，使 LED 熄灭；然后又一次调用延时子程序 delay 延时一秒；最后转入下一次的循环。因此，程序的运行效果是每隔一秒钟后，控制 PC 口的第 0 位输出“1”或“0”，使 LED 亮一秒钟，暗一秒钟，形成秒节拍显示指示。

```
;AVR汇编程序实例:Demo_5_1.asm
.include "m16def.inc"              ;包括器件配置定义文件，不能缺少
.def temp1=r20                     ;定义寄存器R20用临时变量名temp1代表

.org $0000                         ;上电复位启始地址
rjmp reset                         ;转上电复位后的初始化程序执行
                                   ;中断向量区
.org $002A                         ;跳过中断向量区
reset:    ldi r16,high(RAMEND)     ;取内部RAM最高地址的高位字节
          out sph,r16              ;放入SP的高位
          ldi r16,low(RAMEND)      ;取内部RAM最低地址的低位字节
          out spl,r16              ;放入SP的低位，SP中的值见器件配置文件“m16def.inc”
          ser temp1                ;置temp1（R20）为0XFF
          out ddrc,temp1           ;定义PC口为输出
          out portc,temp1          ;PC口输出全“1”，LED不亮
          ldi r16,197              ;设置一秒延时参数
loop:     cbi portc,0              ;值PORTC.0位为“0”，LED亮
          rcall delay              ;调用延时子程序，延时一秒
          sbi portc,0              ;值PORTC.0位为“1”，LED灭
          rcall delay              ;调用延时子程序，延时一秒
          rjmp loop                ;循环跳转到loop继续执行
;通用延时子程序
delay:    push r16                 ;压栈(2t)
del1:     push r16                 ;压栈(2t)
del2:     push r16                 ;压栈(2t)
del3:     dec r16                  ;r16 = r16 - 1,(1t)
          brne del3                ;不为0跳转移,为0顺序执行(2t/1t)
          pop r16                  ;出栈(2t)
          dec r16                  ;r16 = r16 - 1,(1t)
          brne del2                ;不为0跳转移,为0顺序执行(2t/1t)
          pop r16                  ;出栈(2t)
          dec r16                  ;r16 = r16 - 1,(1t)
          brne del1                ;不为0跳转移,为0顺序执行(2t/1t)
          pop r16                  ;出栈(2t)
          ret                      ;子程序返回(4t)
```

5.1.4 通用延时子程序分析

在上面程序中，我们使用了一个通用软件延时的子程序来实现延时。当然采用软件延时并不能得到准确的定时，要产生精确的定时一般应采用定时器。在本例中使用软件延时的主要目的是能够让初学者掌握使用汇编语言开发系统程序的过程，了解和使用AVR Studio集成开发环境编写、编译以及调试系统程序。同时也可以体会编写一个好的、优化的汇编程序也需要相当的软件设计基础和能力。

本例中的通用延时子程序仅使用了一个寄存器R16，采用二次嵌套循环，并多次利用堆栈交换数据。其程序代码短，但能够产生长达2秒的延时（4M系统）。参照该子程序的方法，采用三次嵌套循如，能够产生140秒的延时（使用4MHz晶振）。而采用一般的延时子程序的编写方法，在相同的代码长度时，是不能达到如此长的延时，而且还要占用更多的寄存器。

在子程序代码中，给出了每条指令执行所需要的机器周期数，在使用4MHz晶振时，AVR的每个机器周期t = 0.25us。通过分析程序的执行，可以得到调用该二次嵌套循环子程序执行所需要的总的机器周期数T为：

$$T = 11 + 7x - 1 + \sum_{i=1}^{x}(7i-1) + \sum_{i=1}^{x}\sum_{j=1}^{i}(3j-1)$$

式中x的值为R16的初始设置值，第一项数值11为调用子程序指令rcall、第一条压栈指令push、最后一条出栈指令pop和子程序返回指令ret需要的机器周期数（3+2+2+4）。7x-1为del1循环（外围循环指令）需要的机器周期数。后面两项分别为内循环del2和del3需要的机器周期数。总的延时时间：Delay_Time = T * 0.25us。表5.1给出了几个典型的延时时间。

表5.1 通用延时子程序控制常数与延时周期和时间

使用4MHz晶振，t = 0.25us 在R16中的控制常数 x = 1… 255,0，延时时间范围 6.25us — 2.17s					
x	T	Time	x	T	Time
1	25	6.25us	51	78550	19.64ms
2	52	13.0us	52	83002	20.75ms
3	94	23.5us	71	202360	50.59ms
4	154	38.5us	90	401860	100.465ms
5	235	58.75us	114	800404	200.101ms
6	340	85.0us	131	1202590	300.648ms
7	472	118us	144	1587754	396.939ms
8	634	158.5us	156	2009290	502.323ms
9	829	207.25us	166	2412820	603.205ms
10	1060	265us	175	2819260	704.815ms
13	1999	500us	183	3216784	804.196ms
17	3937	984us	191	3650020	912.505ms
18	4564	1.14ms	197	3999307	999.827ms
20	6010	1.5ms	249	8000629	2.000157s
22	7732	1.933ms			
23	8704	2.176ms	0(256)	8686090	2.1715225s
26	12100	3.025ms			
29	16279	4.07ms			
31	19540	4.885ms			
32	21322	5.33ms			
40	39610	9.9ms			
41	42445	10.61ms			
46	58660	14.67ms			

5.2 AVR Studio—汇编语言集成开发环境使用

AVR Studio 集成开发环境(IDE)是 ATMEL 公司推出的，专门用于开发该公司 AVR 单片机的开发软件平台，它是一个完全免费的，基于 AVR 汇编语言的集成开发环境。AVR Studio 包括了 AVR Assembler 编译器；AVR Studio 软件模拟调试功能；AVR Prog 串行下载和 JTGA 下载功能；JTGA ICE 在线仿真调试等功能。要使用该软件的下载功能和在线仿真调试功能，还需要购买该软件支持的专用的仿真下载硬件设备，如 JTGA ICE 仿真器等。

5.2.1 AVR Studio的安装和其它辅助工具的安装

在本书附带的光盘中，有 AVR Studio、BASCOM-AVR（Demo）、CVAVR(Demo)的系统安装软件。用户也可以通过 Internet 到 ATMEL 网站（http://www.atmel.com）下载最新的版的 AVR Studio，以及其它相关的工具软件。本书中使用的是 AVR Studio4.13 版（2007 年 2 月）。

- 安装AVR Studio4.13

AVR Studio4.12的安装非常简单，用户只要执行从网上下载的aStudio4b528.exe文件，就可以按照提示进行AVR Studio系统的安装了。按照安装过程中的提示，我们将AVR Studio集成开发环境的系统文件安装在目录C:\Atmel\AVR Tools\下面。

使用 Windows NT/2000/XP 的用户请注意，安装 AVR Studio 软件时，必须使用管理员身份的（administrator）权限登陆，这是 Windows 系统限定只有管理员才可以安装新器件。

- 安装BASCOM-AVR软件开发平台

BSACOM-AVR是采用结构型BASIC作为程序设计语言，简单易学，尤其适合中学生、大中专学生学习使用，以及开发一些相对简单的系统使用。用户可以到MCS Electronics的网站 http://www.mcselec.com下载试用版（仅 2KB执行代码限制，其它功能可正常使用）。

由于BASCOM-AVR中的ISP编程功能界面非常友好和直观，它能支持使用STK200/STK300型简易下载线，通过PC的打印机接口，对AVR芯片的熔丝位配置编程，或将一个生成的AVR执行代码程序(HEX/BIN格式)，下载烧入到AVR芯片的程序存储器中。同时，使用该软件的ISP功能对AVR熔丝进行配置时给出了比较清楚的提示，用户不容易出错，所以本书中采用BASCOM-AVR中的ISP功能对AVR芯片的熔丝位编程。

5.2.2 系统工程文件与AVR汇编源程序文件的建立、编译

1. 建立一个新的工程项目管理project文件

AVR Studio采用一个project工程项目管理文件（.APR）保存、记录、管理用户在系统软件开发中所使用和生成的各种文件，以及保存用户的开发环境配置参数和设置情况等。

- 新建工程项目。AVR Studio启动后，你将看到一个欢迎对话框。现在可以创建一个新的项目，点击“New Project”按钮。另外在主窗口中选择 Proiect→Project Wizard，也会出现图5-3所示的欢迎对话框。

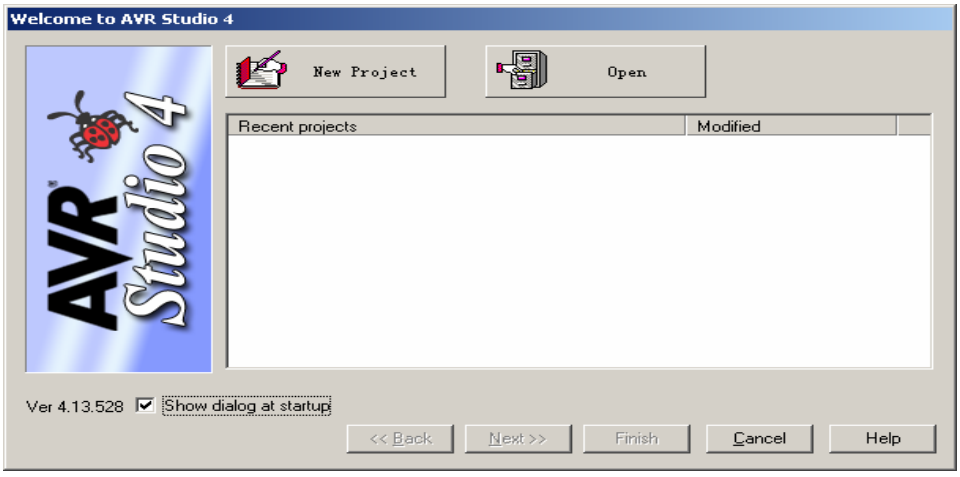

图5-3 启动后的欢迎对话框

- 进入新工程建立窗口（图5-4），配置项目参数。这个步骤包括选择你要创建什么类型的项目，设定名称以及存放的路径等。这个过程包括五个步骤：
  - ✓ 在对话框左边选中Atmel AVR Assembler，表明你要创建一个AVR汇编工程项目。
  - ✓ 输入项目的名称。项目的名称由用户定义，在例子中我们用了“demo_5_1”。demo_5_1作为新建的工程项目文件名，其扩展名.APR可缺省，默认为.APR。
  - ✓ 由AVR Studio自动产生一个空的汇编文件，在例子中我们用了“demo_5_1”。其默认的扩展名为.asm。
  - ✓ 选择新建项目存放的路径，例子中存放在“F:\1015\demo\5-1\”下。
  - ✓ 检查所有的选项，确认之后，按“Next”按钮。

AVRStudio有可能对中文支持不好（不支持UNICODE编码），所以目录和文件名中尽量不要使用中文字符。

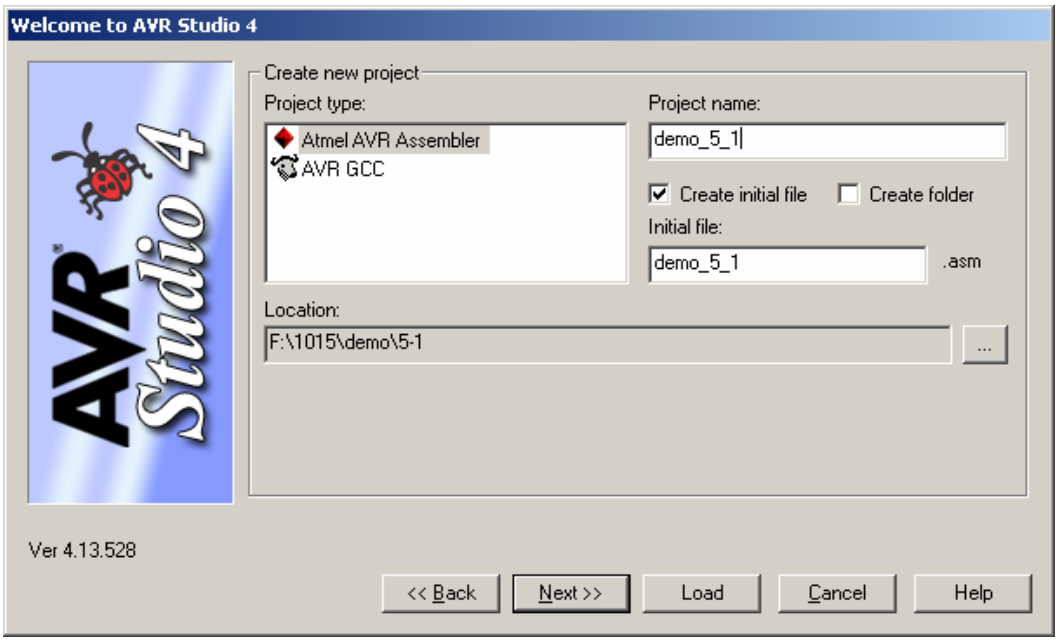

图5-4 新工程建立对话框

- 选择调试平台和使用的芯片型号（图5-5）。

- ✓ AVR Studio 4允许用户选择多种开发调试工具，在这里我们选用软件模拟的带有仿真功能的AVR Simulator，其它选择均为在线仿真功能的开发调试工具，需要相应的硬件设备配合。
- ✓ 芯片我们选用ATmega16。
- ✓ 检查所有的选项，确认之后，按“Finish”按钮。

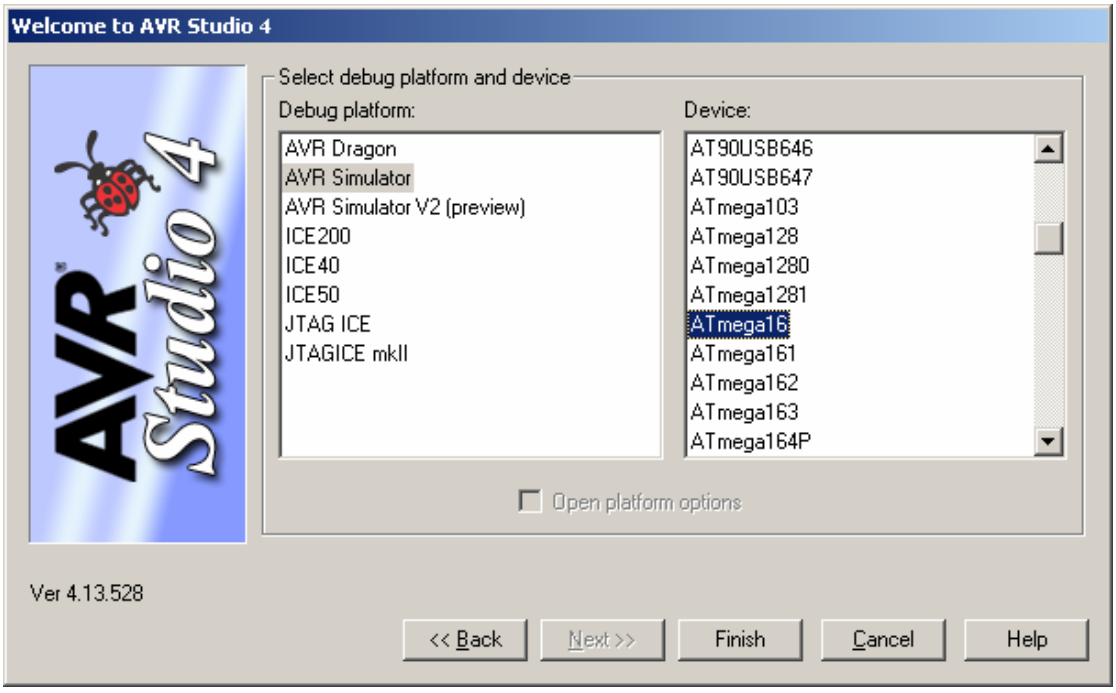

图5-5 调试平台和芯片型号选择对话框

2. 汇编源文件的建立

经过上面的步骤，AVR Studio打开了一个空的文件，文件的名字是demo_5_1.asm。可能你注意到demo_5_1.asm这个文件还是一个空的文件（图5-6）。

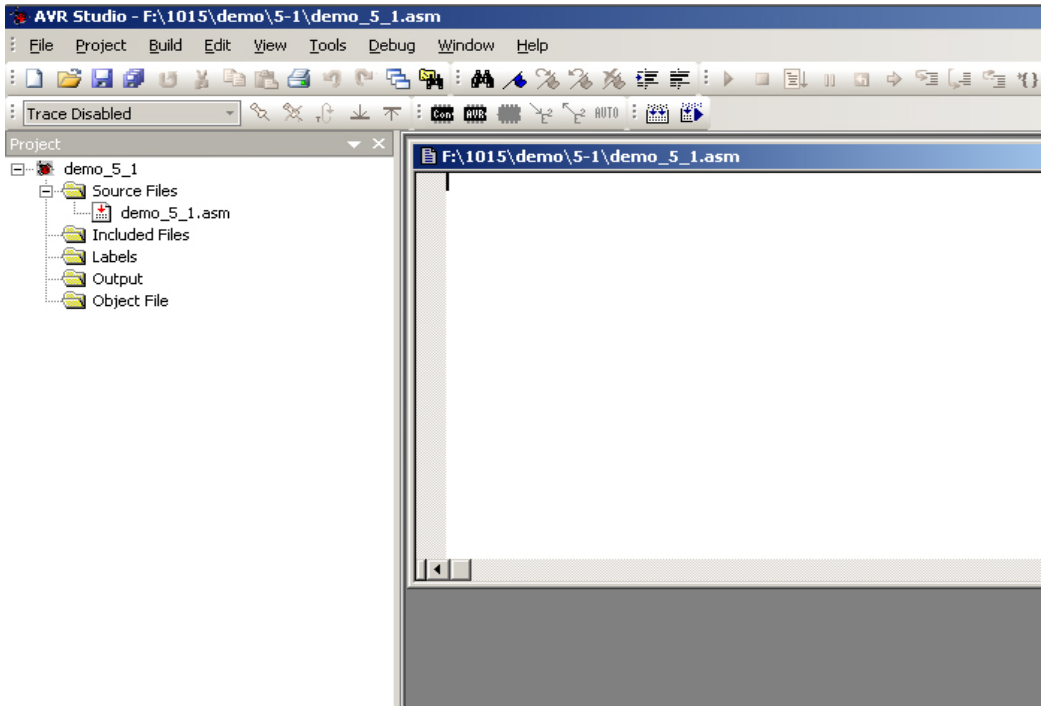

图5-6 AVR Studio主工作界面(一)

在新的源文件编辑窗口中，用户可以输入和编辑你的汇编源程序代码。图5-7为在汇编源程序编辑窗中输入的秒节拍发生器汇编源程序。

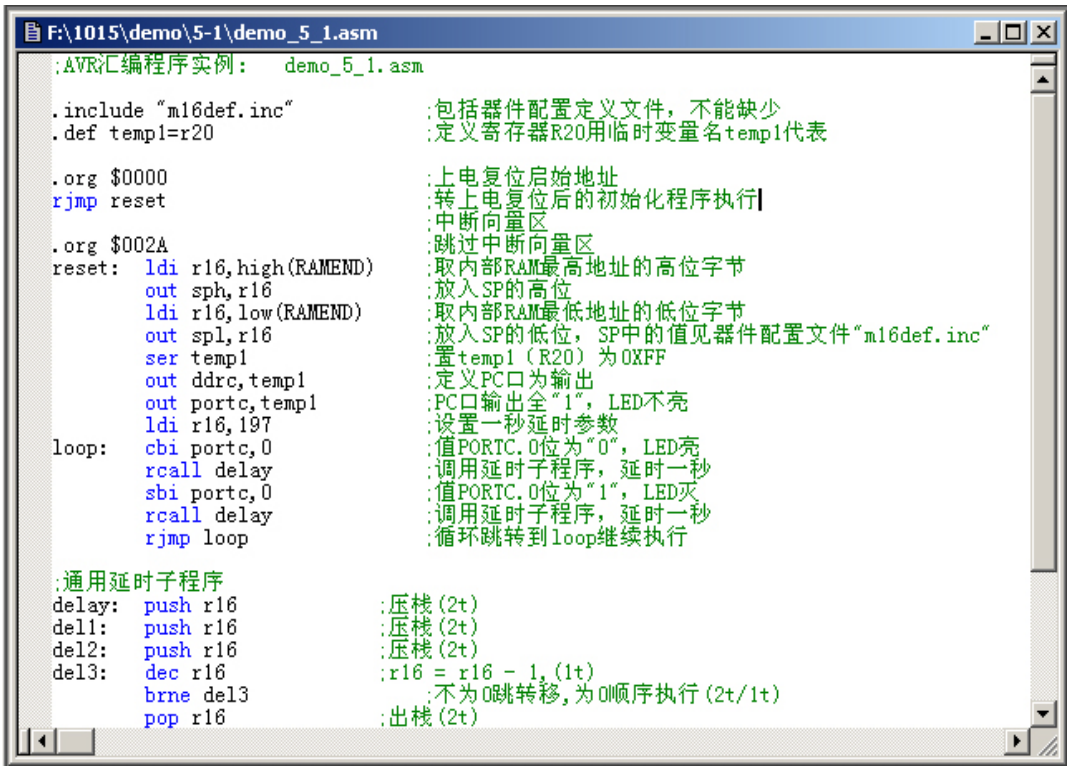

图5-7 输入汇编源程序

3. 汇编源文件的编译

- 选择菜单项Build→Build(或使用快捷键F7，或工具栏上对应工具按钮)对汇编源文件进行编译（图5-8）。

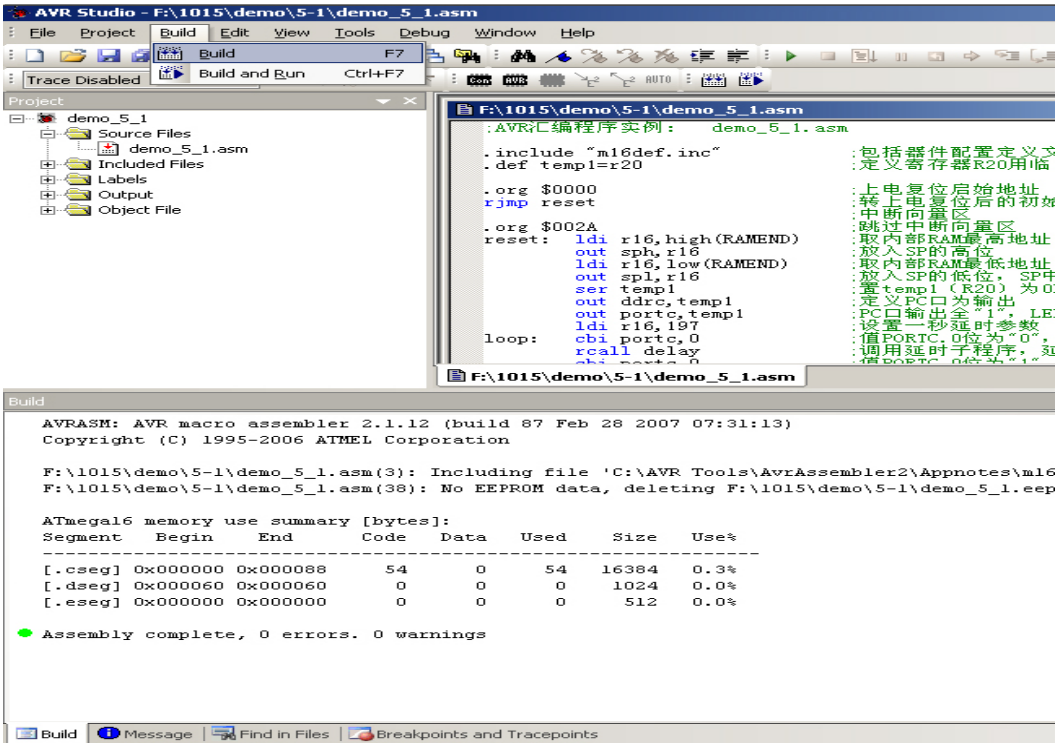

图5-8 编译源文件

- 编译结束后，Studio将在Build的信息提示窗口里显示编译结果。如果发现提示中给出了编译过程中产生的语句错误，用户应该对源程序进行改正后重新编译，一直到编译结果正确为止。

5.2.3 使用软件仿真调试程序

在 AVR Studio 集成开发环境中，可以使用其中的软件模拟仿真调试工具对汇编源文件或第三方支持开发的源程序（C、BASIC 等）进行纯软件环境的模拟仿真调试。这样在没有硬件的情况下，用户可以对自己编写的程序进行调试和排错。软件模拟仿真调试是一种新的逐步推广的有效的调试技术，能够大大节省软件人员的调试时间，节约人力和物力。除了 AVR Studio 能够实现对 AVR 进行纯软件环境的模拟仿真调试外，BASCOM-AVR 本身也具备图形化的更加直观的软件模拟仿真调试功能。另外，还有商业的模拟仿真软件平台 Proteus 和 Vmalb 都能实现对 AVR 的软件模拟仿真调试。

AVR Studio 的仿真调试提供 DBUG 调试、排错、设置断点、单步、自动单步、触发、查看、选项、窗口、帮助等操作。在调试中，可打开多种窗口，如：I/O 窗、源文件窗、汇编机器代码窗、Processor 窗，记录窗，数据窗等（见图 5-9，AVR 集成软件调试窗口图示）。因此，用户可以灵活的使用各种方法，跟踪程序的运行，检查 MCU 中各个寄存器、存储器以及工作单元的变化，检查一条或一段语句和指令执行的时间等。

在本实践过程中，我们以简易秒节拍发生器为例，给出一些基本和简单的调试方法。这些调试方法，与使用硬件实时仿真也是相同的。因此，希望读者在实际操作中以及在以后的学习中，逐步熟练掌握这些模拟调试的技术。

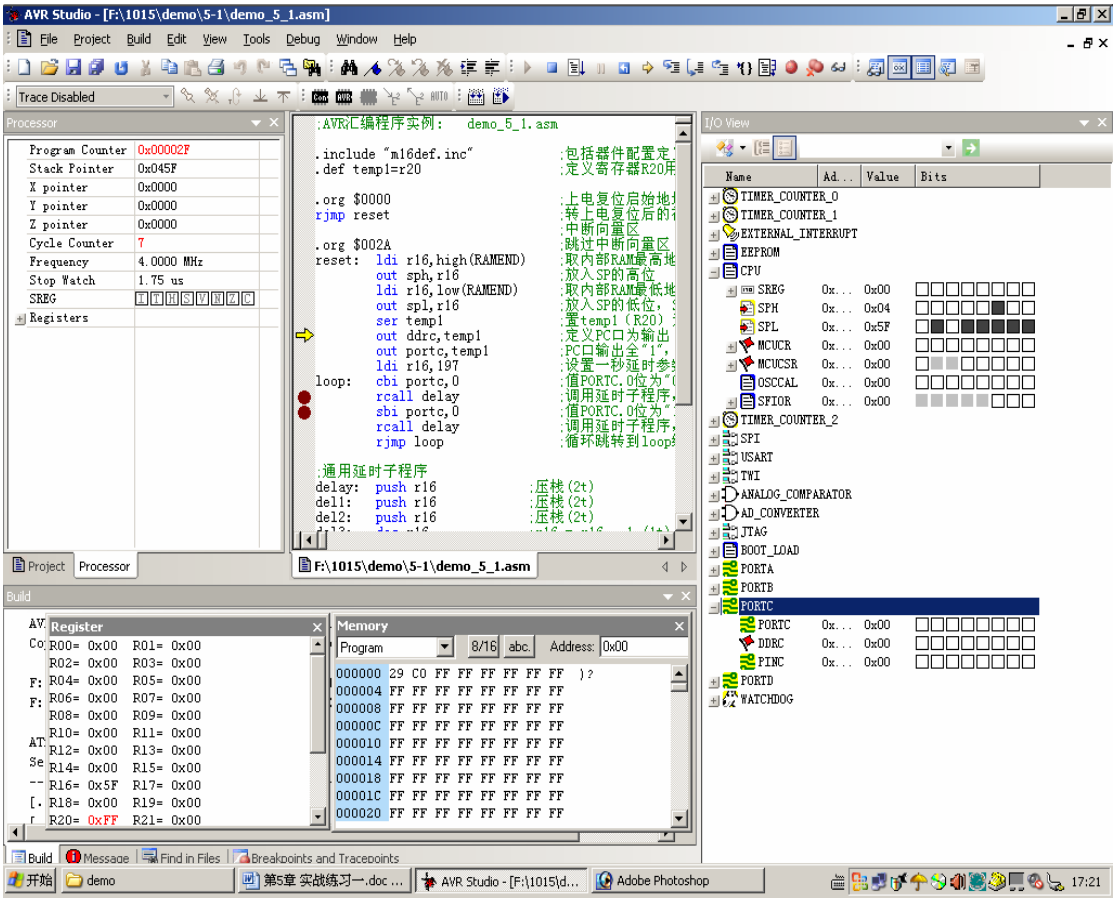

图 5-9 DEBUG 调试和观察

1. 启动进入 DBUG 调试
   - 程序编译无误后，选择菜单 Debug→Start Debugging，启动进入 Debug 调试。
   - 打开相应的观察窗口（图 5-9）。其中主要的窗口有：
     - ✓ Processor 窗口。在该窗口中，用户可以查看 MCU 的主要部分情况，如 PC、Stack、32 个工作寄存器组等。在该窗口中，用户还可以查看和统计指令执行的周期数，以及执行时间等
     - ✓ I/O 查看窗口。这个窗口是最主要的调试窗口，在该窗口中，用户可以查看 MCU 内部集成的功能部件，如 T/C、ADC、USART 等的 I/O 寄存器变化情况和状态。用户不但可以进行观察，同时也可以使用鼠标点击相应的标志位，模拟触发信号的产生，如模拟触发某个中断标志位，从而使 MCU 响应中断，执行中断服务程序，实现对中断程序的调试。
     - ✓ Register 窗口。用于观察 32 个寄存器的变化，如果最后执行的指令改变了寄存器的数值，将已红色显示。在调试过程中，用户也可以根据调试的需要，人为设置和改变寄存器的数值。
     - ✓ Memory 窗口。Memory 窗口用于观察 RAM、EEPROM、FLASH 三个不同空间单元的变化。在调试过程中，用户也可以根据调试的需要，人为设置和改变这些存储单元的数值。图 5-9 中，看到的是 RAM 空间高端地址的单元，这样在调用子程序时，可以观察到堆栈的变化（本程序的堆栈空间设置在 RAM 的高端）。
   - 几个重要的标记和工具栏按钮
     - ✓ 在程序代码的窗口左边，有一个黄色的尖头，它表示下一步要执行的指令（该指令还没有执行，但马上要执行）。
     - ✓ Debug 工具中各种用于调试的按钮，如：单步执行 Step Into（F11），分段执行 Step Over（F10，通常使用一次执行完整个子程序的调用），全速执行程序 Run（F5，只有遇到断点才停止）等。在调试中可以灵活使用。
     - ✓ 程序断点的设置与取消。在调试过程中，用户可以根据需要在程序中设置多处断点。程序在全速运行中，遇到断点就会暂停，此时可以观察 MCU 的变化，进行必要的设置，而后再继续（使用 F5）从断点处向下运行。断点的设置和取消方法非常方便：将光标定位于需要设置为断点的语句行，单击 Toggle Breakpoint 按钮（F9），该语句设置为断点，在左面出现红色园点标记，表示断点。如将光标定位于已设置为断点语句行，单击 Toggle Breakpoint 按钮（F9），将取消该语句设置的断点，左面的红色园点标记消失。
     - ✓ 使用 Processor 窗口中的 Stop Watch 功能，可以检查一条或一段语句和指令执行的时间。
2. 单步运行，观察 MCU 内部资源的变化
   - 按 F11 键单步执行程序，观察指令的执行和查看 CPU 各种资源的变化情况。图 5-9 显示了单步执行 6 条指令后，堆栈寄存器 SPH、SPL 的情况，以及变量 temp1(定义为工作寄存器 R20)的值。在 R20 中其数值为 0xFF，为红色，表示刚执行的指令“ser temp1”将 R20 置为全 1。
   - 在 Processor 观察窗口中，可观察 MCU 的指令计数器 PC 的值为 0x002F（当前指令地址），Stack Pointer 的植为 0x045F(堆栈指针值，与堆栈寄存器 SPH、SPL 的表示对应的)，以及使用了 7 个时钟周期（Cycle Counter），而 Stop Watch 的值为 1.75us（5 * 0.25us + 0.5us，其中 rjmp reset 单条指令执行时间为 2 个时钟周期 0.5us）等等。
3. 调试验证通用延时子程序delay的延时效果

由于软件模拟调试方式是由PC上的软件来模拟AVR的操作过程，因此它不能达到象硬件那样的真实速度，尤其是模拟一个延时程序，需要比较长的时间。下面以本例说明，如何简单和正确的进行延时程序的调试。

- 先将汇编程序中的延时参数197改为3，重新编译后进入调试方式。（更简单的方式是先单步执行初试化部分的指令，当执行完“ldi r16，197”一句后，使用鼠标双击Register窗口中R16寄存器，将R16的值改写为3，这样就不需要重新编译程序了。）
- 使用单步执行的方式执行延时子程序的每一句语句，查看程序的逻辑对不对，能否正确运行，堆栈是如何工作的，SP指针如何变化，各个寄存器如何变化，PC的变化，RAM中数据的变化。这样即了解了AVR的工作原理，也了解程序设计的技巧（学别人的），或验证程序是否同自己想象的那样正确（自己编的），而且训练了如何熟练使用DEBUG（熟练使用工具也是很重要的一环）。
- 验证了整个延时程序没有逻辑错误后，可以查看延时子程序的延时时间了。
  - ✓ 将延时参数由3改回197，编译后进入调试方式。
  - ✓ 在调用该子程序的语句“rcall delay”处设置一个断点；在接下来的一个语句“sbi portc，0”处设置第二个断点（见图5-9）。
  - ✓ 按F5，全速运行程序。
  - ✓ 当程序在第一个断点处停下时，在Processor的选项中（展开该图标）找到Stop Watch子项，在右键菜单中选择将其清另。
  - ✓ 按F5，从断点处继续全速运行程序（开始调用延时子程序）。
  - ✓ 等大约十几秒或几十秒后（取决于你的PC速度），程序在第二个断点处停下（子程序模拟运行时，AVR Studio下面状态栏中的运行图标为绿色，暂停为黄色）。
  - ✓ 查看Processor的选项中Stop Watch的值(本例中为999826.75us)，它记录下调用子程序返回后的时间，该时间值即为延时子程序的运行时间。

由此验证了延时子程序的执行时间。调节延时参数，可以得到不同的延时时间，通过软件模拟可以精确的得到。这比使用在线实时仿真的手段要方便多了，而直接在目标板上运行，你也不能得到精确的时间。

使用软件模拟仿真是现在调试技术的发展方向。当你了解和熟练掌握使用AVR Studio后，你的设计研发速度会提高，硬件系统和软件编程可以平行开展，当硬件完成了，你的软件也完成了60-70%。

5.3 CVAVR + AVR Studio—高级语言集成开发环境使用

CodeVision AVR 是 HP Info Tech 专门为 AVR 设计的一款低成本的 C 语言编译器，它产生的代码非常严密，效率很高。它不仅包括了 AVR C 编译器，同时也是一个集成 IDE 的 AVR 开发平台，简称 CVAVR。

与其它的 C 语言开发平台相比较，CVAVR 对位（bit）变量的支持，大量扩展的对一些标准的外部器件支持和接口函数（如：标准字符 LCD 显示器、I2C 接口、SPI 接口、延时函数、BCD 码与格雷码转换等），以及方便的对 EEPROM 的操作功能等特点更加适合一般人员的学习和使用。

HP Info Tech的网站地址为http://www.hpinfotech.ro，网站提供试用板（2K代码限制）安装程序的下载。读者可以从本书附带的光盘中找到试用版的安装软件CVAVR_setup.EXE。

读者注意：本书中的 C 源代码程序均是在 CVAVR 系统下实现的。

5.3.1 秒节拍显示器的高级C语言源程序代码

下面是使用高级语言编写的秒节拍发生器的 C 语言源程序。C 的源程序看上去比汇编的简洁，也更加易懂。

在程序的初始化代码中，仅仅对 PORTC 口进行了设置，而没有对 AVR 堆栈指针的初始化设置，这是由于 CVAVR 系统在编译时会首先帮用户自动的设置堆栈指针，方便了用户的使用。与汇编相同的是在 C 主程序中，由 while(1)构成无限（死）循环，循环中调用了 CVAVR 提供的延时函数 delay_ms()，延时 1 秒钟后将 PC0 口的值取反输出，控制点亮和熄灭 LED。因此，程序的运行效果是每隔一秒钟后，控制 PC 口的第 0 位输出“1”或“0”，使 LED 亮一秒钟，暗一秒钟，形成秒节拍显示指示。

```
/*********************************************
Demo_5_2.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/

#include <mega16.h>                 // 包括器件配置定义的头文件，不能缺少
#include <delay.h>                  // 包括延时函数定义的头文件，使用延时函数时不能缺少

void main(void)
{
     // 定义PortC口的工作方式
     PORTC=0x01;                    // PC口的第0位输出“1”，LED不亮
     DDRC=0x01;                     // 定义PC口的第0位为输出方式
     // 主循环
     while (1)
     {
          delay_ms(1000);           // 调用CVAVR提供的毫秒延时函数，延时1s
          PORTC.0 = ~PORTC.0;       // PC口第0位输出取反
     };
}
```

5.3.2 系统工程文件与源程序文件的建立、编译

CVAVR系统的安装也是非常简单的。用户只要执行从网上下载的CVAVR系统安装“setup.exe”文件，就可以按照提示进行CVAVR系统的安装了。按照安装过程中的提示，我们将CVAVR集成开发环境的系统文件安装在目录C:\cvavr\下面。

1. 建立一个新的工程项目管理project文件

CVAVR也采用project工程项目管理文件（.APR）来保存、记录、管理用户在系统软件开

发中所使用和生成的各种文件，以及保存用户的开发环境配置参数和设置情况等。

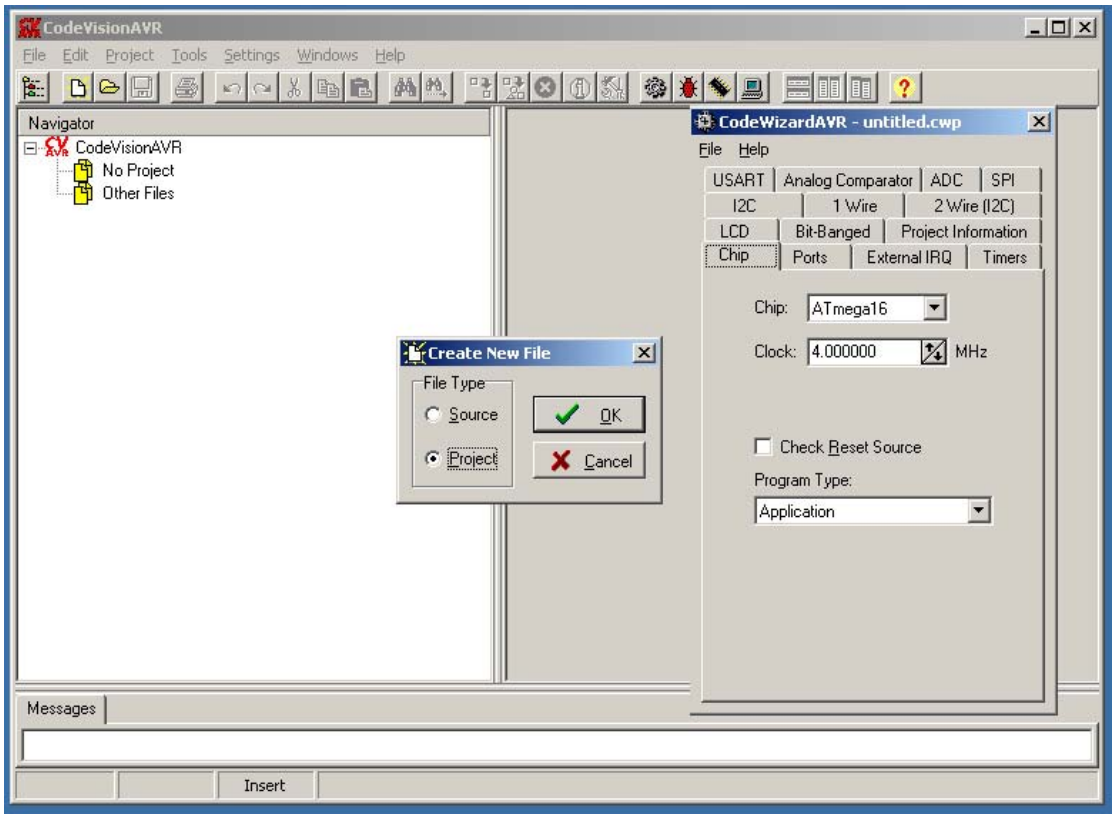

图5-10 在CVAVR中创建新的工程项目

- 新建工程项目。CVAVR启动后，你将看到它的主工作界面。现在可以创建一个新的项目：
  - ✓ 选择菜单栏中“File→New”，出现“Create New File”对话框。
  - ✓ 选择Project选项，表示新建一个工程项目（图5-10），按“OK”按钮确认。
  - ✓ 随后CVAVR出现一个对话框，询问用户是否使用需要在CVAVR系统的程序自动生成向导器的帮助下生成源程序的主结构框架。建议使用该功能，选择“Yes”进入CodeWizardAVR选择对话框。
- 在CVAVR系统的程序自动生成向导器的帮助下生成源程序的主结构框架。CVAVR系统的程序自动生成向导器是一个非常具有特点的功能。用户在它的帮助下，可以非常简单和方便的生成源程序的主结构框架，其中还包括了对AVR各个I/O寄存器初始化的代码。这使得用户不必频繁的查看手册，去确定各个标志的意义，以及计算初始设置值等。读者应逐步掌握和熟练使用该项功能。
  - ✓ 确定使用AVR芯片的型号和系统时钟频率值。本例中，选择ATmega16，系统时钟频率为4M（参见图5-10）。
  - ✓ 确定PORTC口的工作方式。本例中只使用了PORTC口的最低位，为输出方式工作，用于控制LED。图5-11给出对PORTC口初始化配置的界面，用鼠标点击“Bit 0”的方向为输出Out，输出初始值为1。
  - ✓ CodeWizardAVR选择对话框中还有许多对AVR各个功能部件的配置选择，由于本例非常简单，只用到PORTC的第0位，因此配置完成。读者可以仔细游览各个功能配置，配合CVAVR的HELP文件，了解其如何使用，同时也加深对AVR内部资源的熟悉和了解。

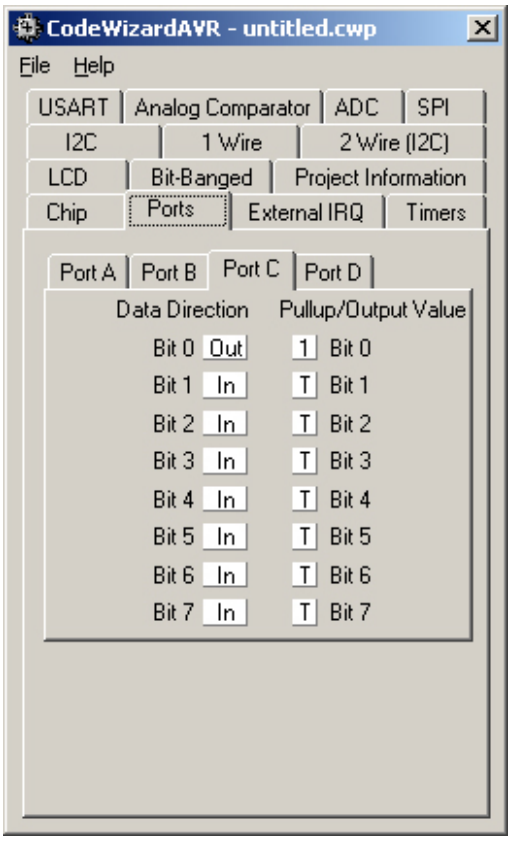

图5-11 PORTC口的初始化配置

2. 建立源文件，输入源代码，编译源代码

- 建立源文件，输入源代码
  - ✓ 在CodeWizardAVR选择对话窗口的菜单栏中选择“File→Generate，Save and Exit”项。
  - ✓ 在弹出的对话框中分别填入源代码文件名xxxxxx.c、工程管理文件名xxxxxx.prj、和器件初始配置文件xxxxxx.cwp，并确认。本例中使用主文件名为demo_5_2。
  - ✓ 在随后出现的CVAVR主工作窗口中，右面的源程序文件中，已经出现了一个根据用户配置而生成的源程序的主结构框架（图5-12）。用户可以在这个主结构框架中的相应的地方输出自己的源代码了。
  - ✓ CodeWizardAVR生成的源程序的主结构框架有许多对AVR各功能I/O寄存器初始化的代码，本例中将一些不影响秒节拍显示器的语句简化了。读者在实际操作中，应该仔细学习CVAVR生成程序的风格和思想，建立良好的程序设计和编写的习惯。

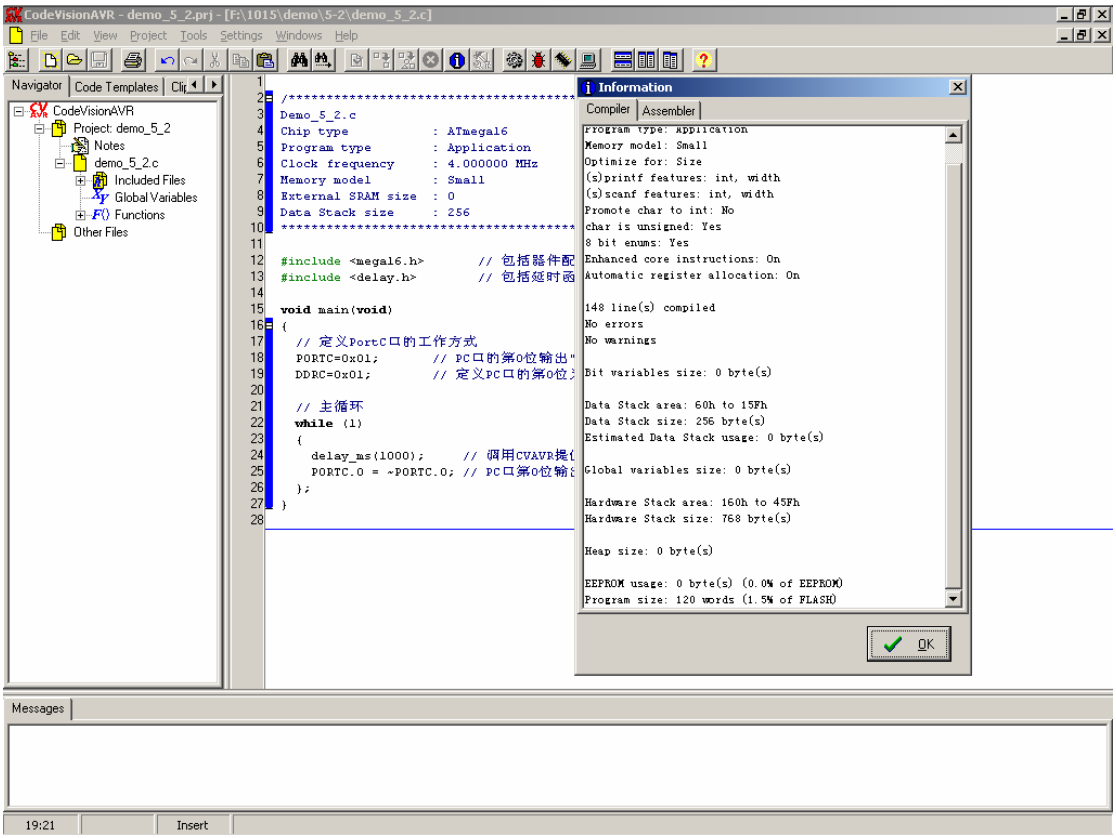

图5-12 CVAVR的主工作窗口

- 编译源文件
  - ✓ 选择菜单项Project→Make(或使用快捷键Shift+F9，或工具栏上对应工具按钮)对C源文件进行编译。
  - ✓ 编译结束后，CVAVR将在Information窗口显示编译结果。如果程序中有语法或连接错误，在左边的Navigator窗口中将给出红色的错误提示详细。发现提示中给出了编译过程中产生的错误，用户应该对源程序进行改正后重新编译，一直到编译结果正确为止。

5.3.3 在CVAVR中使用AVR Studio进行软件仿真调试程序

CVAVR系统本身不带有ICE的调试功能，但它能生成与AVR Studio兼容的调试文件xxxxxx.cof，并通过该文件同AVR Studio实现连接，使AVR Studio成为支持C语言的调试仿真平台。

- 配置CVAVR使用的调试器。
  - ✓ 选择菜单项Settings→Debugger，进入CVAVR的仿真调试器配置对话框。
  - ✓ 设置AVR Studio为CVAVR的仿真调试器，如图5-13所示。

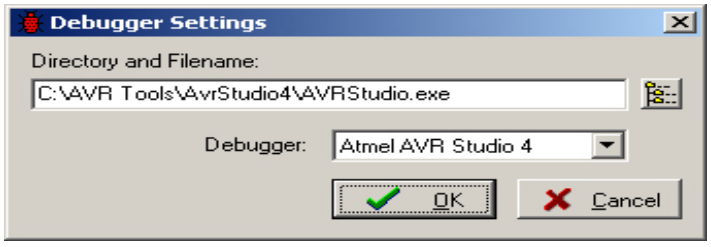

图5-13 设置CVAVR的仿真调试器为AVR Studio

- 使用AVR Studio的软件仿真调试程序进行C代码的模拟调试。
  - ✓ 选择菜单项Tools→Debugger（或使用快捷键Shift+F3，或工具栏上对应工具按钮），启动AVR Studio的仿真调试环境。
  - ✓ 在欢迎对话框中（图5-3）点击“Open ”按钮，在下面的对话框中选择文件xxxxxx.cof（本例为mode_5_2.cof），确认打开。
  - ✓ 选择调试平台和使用的芯片型号（图5-5）。同样，在这里我们选用软件模拟的带有仿真功能的AVR Simulator。
  - ✓ 芯片我们选用Atmega16。检查所有的选项，确认之后，按“Finish”按钮。

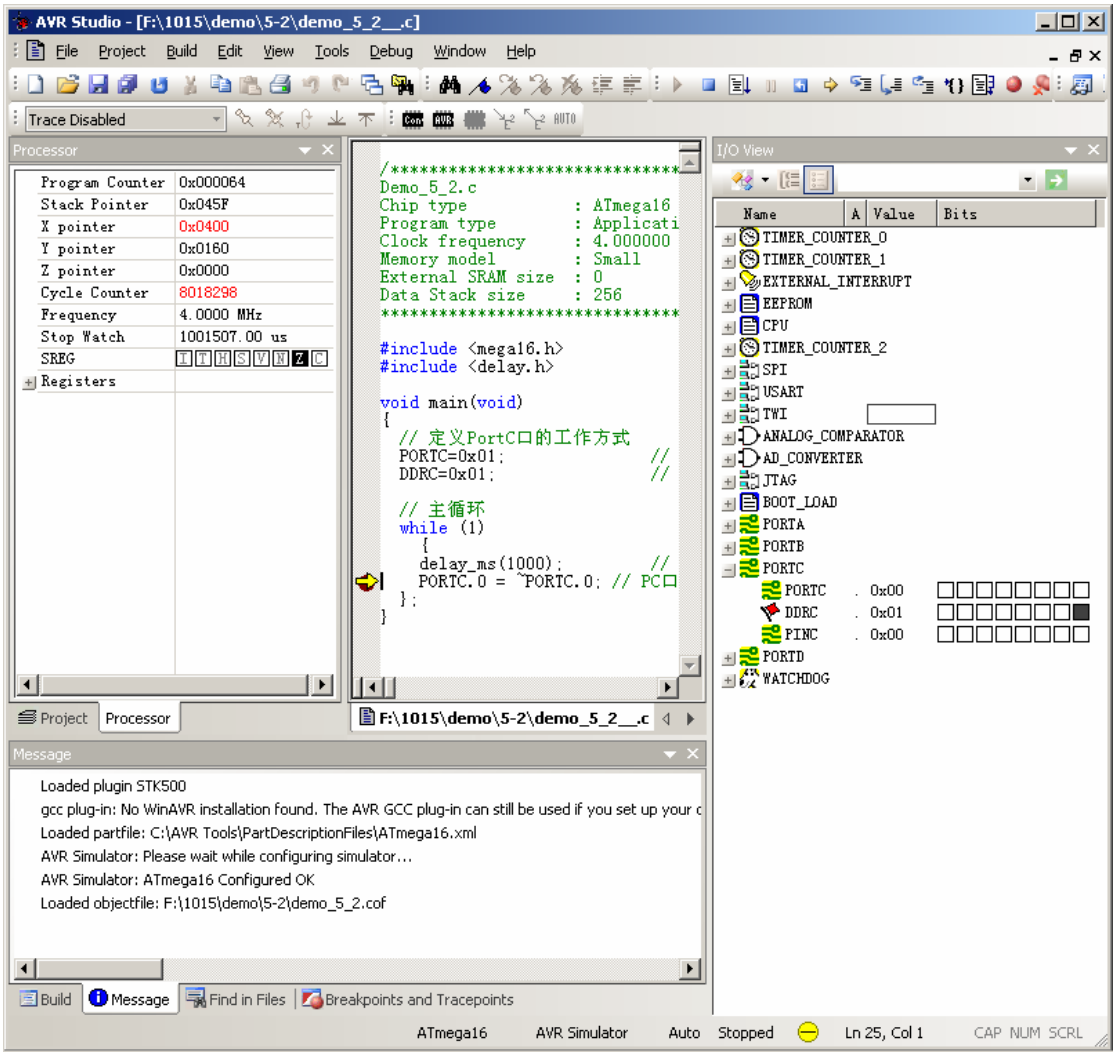

图5-14 在AVR Studio中调试CVAVR的C代码源程序

- CVAVR的C源代码的模拟调试。图5-14就是使用AVR Studio对第三方高级语言代码的调试仿真界面。它同AVR汇编语言程序的调试方法相同，区别在于源代码是高级语言编写的。通过选择菜单项View→Disassembler，用户还可以打开反汇编的代码，以方便进行更加深入的汇编级的调试。

5.4 AVR熔丝位的设置和执行代码下载

通过在AVR Studio中使用软件模拟仿真，可以将程序中的许多问题和BUG找出来，并及时进行修改和调整。模拟仿真完成后，可以将生成AVR的执行代码文件xxxxx.hex下载烧入到

Atmega16芯片中进行实际的运行。如有问题，则需要找出原因，再次进行调试。

5.4.1 AVR-51 多功能板的硬件连接

首先要在“AVR-51多功能实验开发板”上构建一个ATmega16的最小系统。根据第四章表4.2，我们采用短路片进行相关的连接，见图5-15、图5-16。

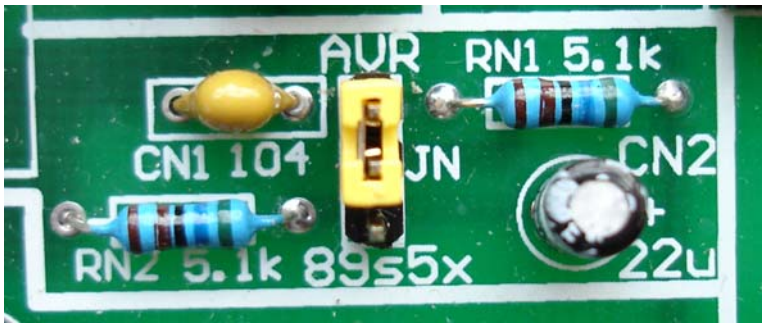

图5-15 用短路片将JN的中心和上端连接

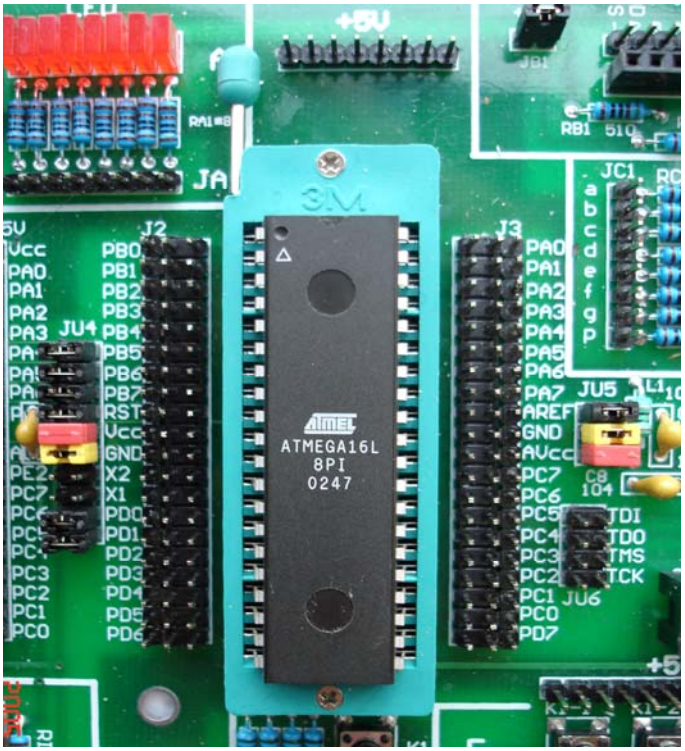

图5-16 使用短路片构成ATmega16最小系统

在图5-15中，使用黄色短路片将JN的中心针与上端标有AVR标记的针短接，构成适合AVR单片机的上电复位电路。

在板上右边的锁紧插座中，放入40PIN的ATmega16，在其两边的JU4、JU7短路排中使用了11个短路片。其中2个红色为连接电源5V，提供芯片工作电源；2个黄色将芯片的地与实验板的地连接；JU4上面4个黑色的连接ISP接口，用于程序下载；JU4下面2个黑色的连接MAX202，构成串行通信接口。

本实验使用ATmega16内部的RC振荡源，并且不使用JTAG口，因此X2、X1和JU8的6个短路排开放。

以上构成了ATmega16的最小系统，最后使用一根连接线将PC0与A区8个LED中的一个连接

（JA），一个秒节拍显示器硬件电路就构成了。

5.4.2 AVR熔丝位的配置

刚出厂ATmega16单片机缺省使用内部1M的RC振荡源作为系统的时钟，而且JTAG口处于允许方式等，因此需要对熔丝位先进行必要的配置。

对于刚开始学习使用AVR的读者，建议改变的熔丝位有：

- ✓ 系统时钟采用内部4M的RC振荡源。其优点是速度适中，且应用于RS-232通信时，分频产生的9600bps速率与标准值的误差最小（0.2%）。
- ✓ 禁止片内的JTAG口功能。不使用JTAG在线仿真，将4个引脚PC2-PC5释放，作为普通的I/O使用。
- ✓ 启用低电压检测复位功能。检测电平设置为4.0V。

在使用AVR单片机时，首先注意要对它的配置熔丝位进行正确的配置编程！

建议采用BASCOM-AVR中的编程下载功能，可以有效的防止错误的发生！

下面给出在BASCOM-AVR环境下设置ATmega16的操作过程。

- 启动BASCOM-AVR，在菜单栏中选择File→New，建立一个空的文件。
- 菜单栏中选择Options→Programmer，进入编程器配置对话窗，选择使用STK200/STK300 Programmer下载线。确认返回（图5-17）。

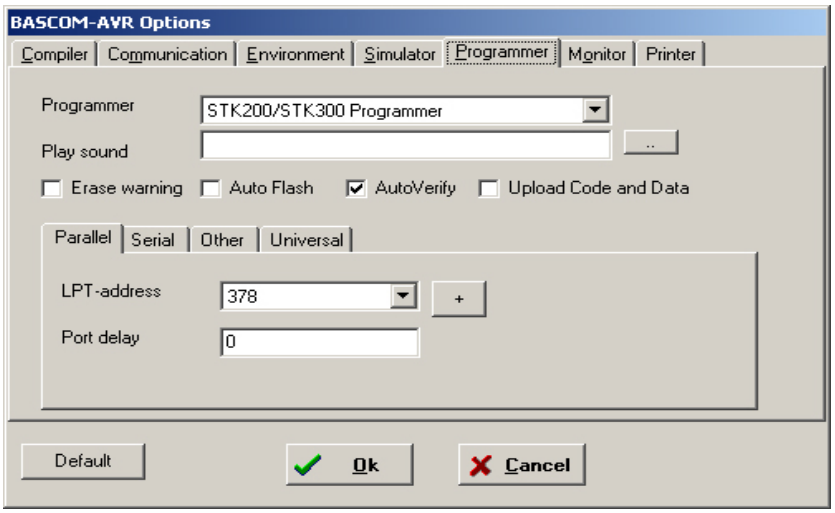

图5-17 BASCOM-AVR编程器选择对话窗

- 将ISP下载线一端与PC的打印口连接，另一端插入实验板的ISP插座中，打开实验板的电源。
- 在菜单栏中选择Program→Send to chip（或使用快捷键F4，或按工具栏中相应的按钮），进入编程功能。
- 跳过2个（由于没有源文件造成）提示信息窗，出现了BASCOM-AVR的编程功能窗口。选择“Lock and Fuse Bits”，可以看到BASCOM-AVR已经读出了芯片的ID号，以及所有熔丝位的状态（图5-18）。

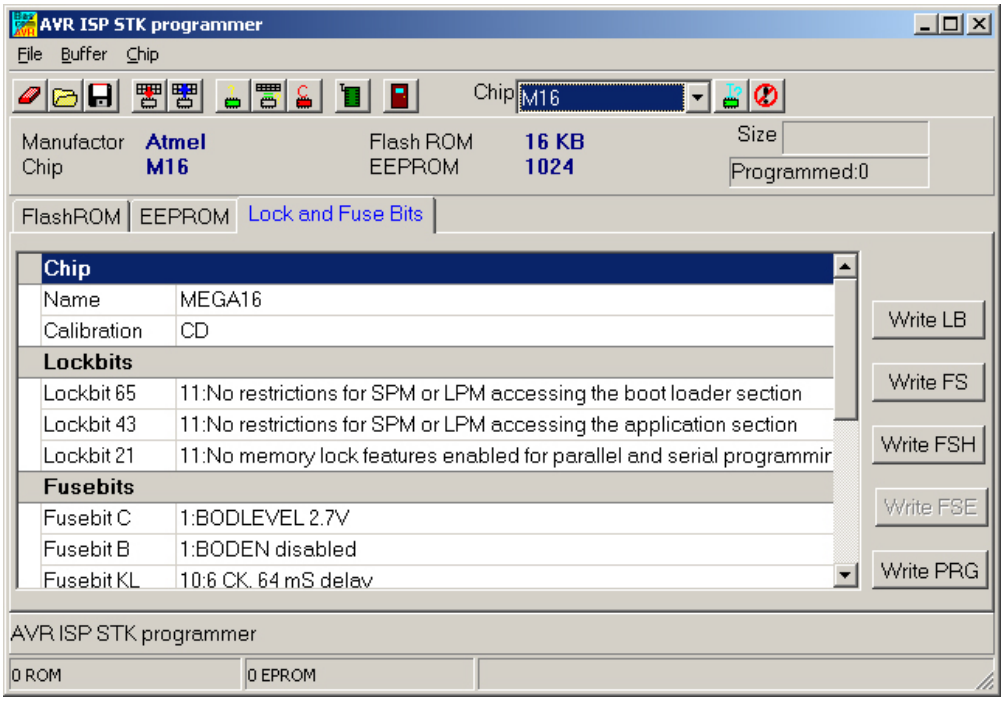

图5-18 熔丝位查看设置窗口

- 在BASCOM-AVR中，对AVR芯片的熔丝位进行了分类，并且提供用户采用下拉菜单式的选择项，每个配置都有简单的解释，非常人性化，有效的防止错误的产生。
- 在本例中，需要重新设置的熔丝有4处（图5-19）。

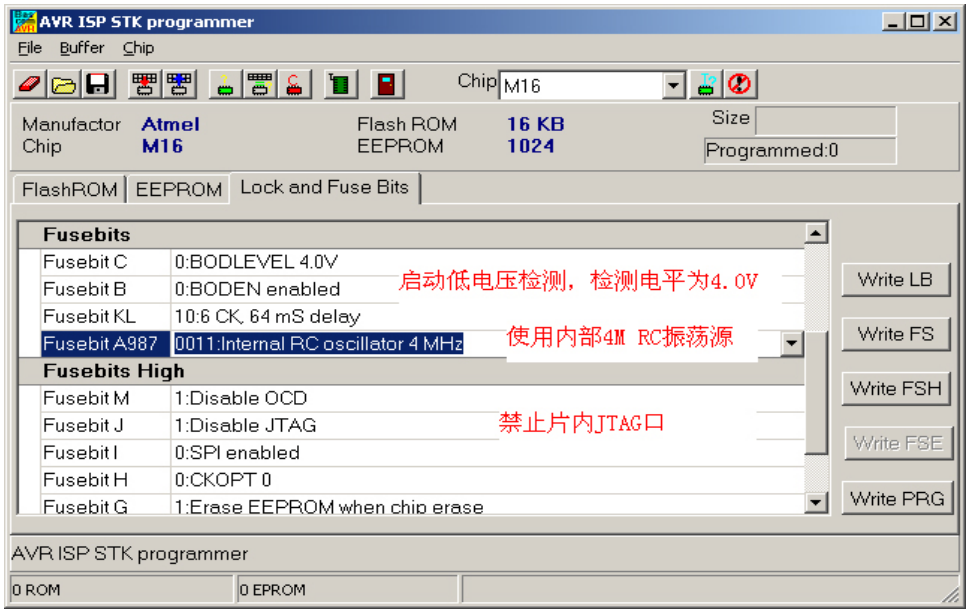

图5-19 熔丝位设置与编程窗口

- ✓ 在Fusebits组中，选定BODEN enabled（允许低电压检测）、BODLEVEL 4.0V（检测电平为4.0v）、Internal RC oscillator 4MHz（使用内部4MHz的RC振荡器为系统时钟源）。检查无误后，按“Write FS”按钮正式改变熔丝的设置状态。
- ✓ 在Fusebits High组中，选定Disable JTAG（禁止片内JTAG口，释放PC2-5为通用I/O）。检查无误后，按“Write FSH”按钮正式改变熔丝的设置状态。

- 由“Lock and Fuse Bits”切换到“Flash ROM”或“EEPROM”，然后在切换回到“Lock and Fuse Bits”，再次回读熔丝位的状态，确认设置是否正确。

- 对于Flash ROM和EEPROM进行擦除或编程下载的操作过程，是不会改变熔丝位的状态的。因此，当熔丝位设置完成后，一般不需要多次的重写，除非必须再次改变熔丝位的配置。

5.4.3 执行代码文件的下载

熔丝位配置完成后，就可以将编译好的运行代码下载烧入到AVR的Flash中，查看系统真实的运行情况了。由于AVR Studio不支持使用STK200/STK300兼容的下载线，因此我们可以在BASCOM-AVR中，将在AVR Studio中使用汇编编写调试过的系统程序，经过编译后生成的执行代码（HEX格式）下载到AVR芯片中。

而CVAVR是直接支持使用STK200/STK300兼容的下载线的，因此在CVAVR中，用户在编写完C源程序后，先应该利用AVR Studio进行模拟的仿真调试。调试完成后，可以直接使用CVAVR中的程序下载功能，通过STK200/STK300将运行代码烧入到AVR中。

1. 用BASCOM-AVR下载烧入执行代码

- 将BASCOM-AVR的编程功能窗口切换显示Flash ROM，在菜单栏中选择Buffer→Load from file，将需要下载的运行代码文件（.hex或.bin格式）读入到PC的内存缓冲区Buffer中（图5-20）。

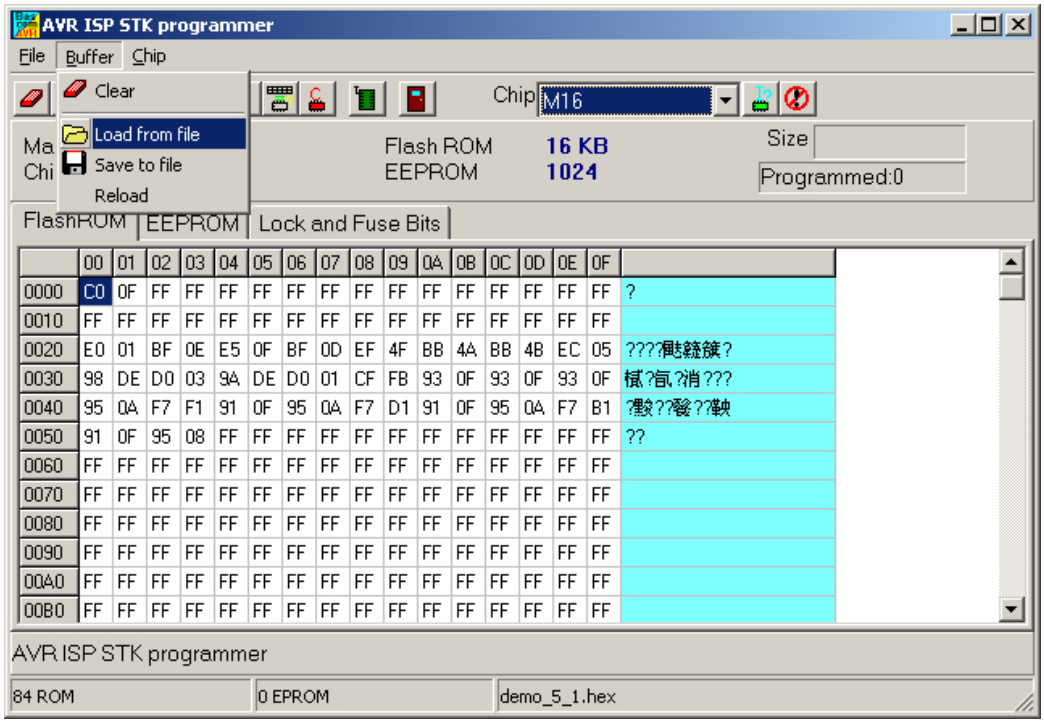

图5-20 选择读入HEX或BIN运行代码文件

- 在菜单栏中选择Chip→Erase（或工具栏上相应的工具按钮），先对AVR芯片进行擦除，将原先的Flash中的程序清除掉。
- 在菜单栏中选择Chip→Write buffer into chip（或工具栏上相应的工具按钮），将Buffer中的运行代码写入到AVR中。

2. 在CVAVR中下载烧入执行代码

- 在CVAVR的主工作窗口中，选择菜单栏Settings→Programmer，进入编程器的选择对话框。选择使用Kanda Systems STK200+/300下载线（图5-21），确认后返回。

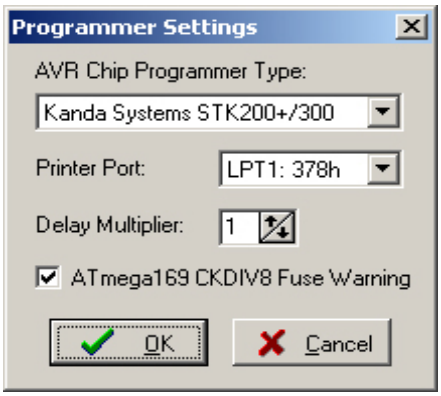

图5-21 选择STK200/300下载线

- 在CVAVR的主工作窗口中，选择菜单栏Tools→Programmer，进入CVAVR的编程功能窗器（图5-21）。

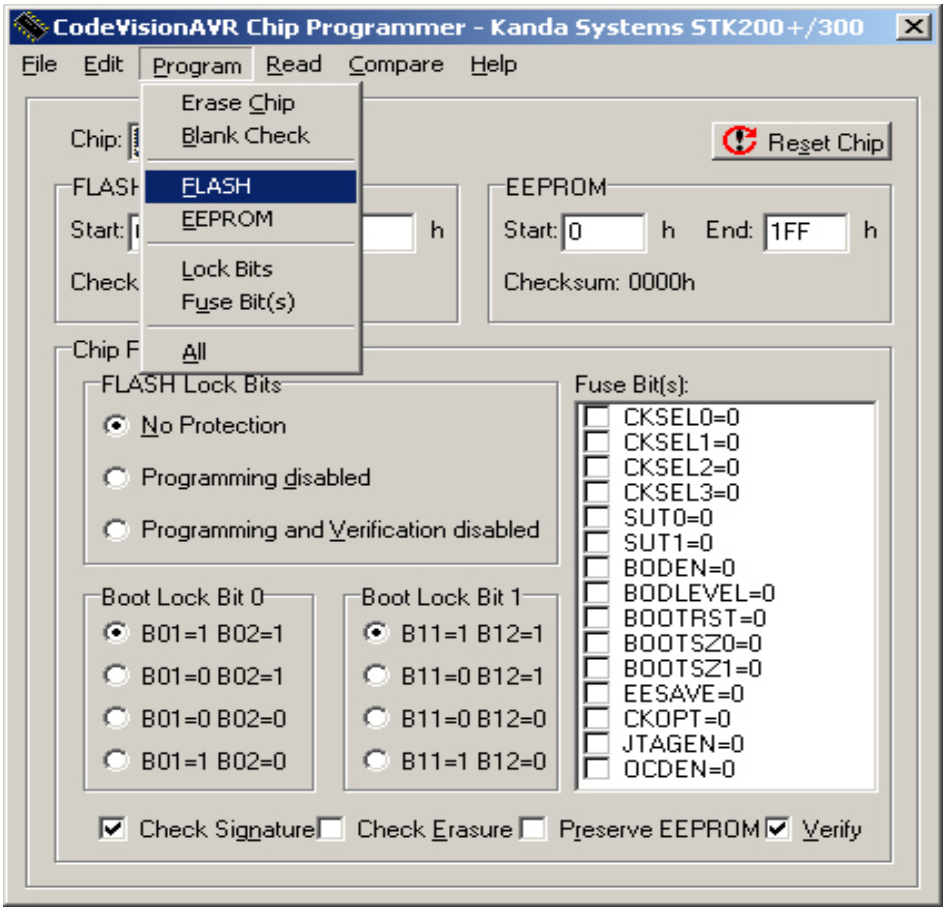

图5-22 CVAVR的编程下载功能窗

- 选择菜单项Program→Erase Chip，先对AVR芯片进行擦除，将原先Flash中的程序清除掉。
- 选择菜单栏Program→FLASH，将Buffer中的运行代码写入到AVR中。

使用CVAVR中的编程下载程序时应特别注意，由于CVAVR编程下载界面初始打开时，大部分熔丝位的初始状态定义为“1”，因此不要使用其编程菜单选项中的“all”选项。此时的“all”选项会以熔丝位的初始状态定义来配置芯片的熔丝位，而实际上其往往并不是用户所需要的配置结果。如果要使用“all”选项，应先使用“read->fuse bits”读取芯片中熔丝位实际状态后，再使用“all” 选项。

尽管在 CVAVR 中也可以配置 AVR 的熔丝位，但它没有给出具体详细的设置意义，因此用户需要仔细的参考和核对器件手册，防止产生错误的配置。

对于刚开始学习使用 AVR，以及对 AVR 的熔丝位配置不是很熟悉的人员，建议使用 BASCOM-AVR 对熔丝进行配置和设置。

以上，我们结合一个简单的例子，讲解了如何使用汇编语言以及高级语言C编写AVR的系统程序；如何使用AVR Studio、CVAVR、BASCOM-AVR等软件；如何进行软件模拟调试；如何配置AVR的熔丝；以及如何下载运行代码程序的整个过程。

上面的介绍都还是一些最基本的操作和过程，希望读者能在以后的学习实践过程中，尽快的熟悉这些软件的功能和使用方法，并能熟练的掌握。

5.5 一个比较复杂的AVR汇编语言实例

在本节中给出一个完整使用 AVR 汇编语言开发编写的一个简单的应用程序。通常使用 ATmega16 构成的系统是比较复杂的系统，因此使用高级程序语言设计编写系统程序更加方便和快捷。本节给出汇编语言程序的目的，主要是对本章内容的总结，同时让用户通过该实例的阅读，能更加了解 AVR 汇编语言的使用，以及对在编写、阅读和调试 ATmega16 的汇编代码时需要注意的一些问题给出讲解。

5.5.1 系统功能与硬件设计

该应用系统为一个带 1/100 秒的简易 24 小时制时钟，它在上电后能够自动从 11 时 59 分 55 秒 00 开始计时和显示时间。图 5-23 为简易时钟系统硬件电路图。

如图所示，系统使用 8 个 LED 数码管作为时钟的显示器，显示时、分、秒、1/100 秒 4 个时段的数字，每个时段占用 2 个 LED。显示方式采用动态扫描方式，ATmega16 的 PA 口输出显示数字的 7 段码（注意：图中省缺了 PA 口连接到 LED 各段的 8 个限流电阻，阻值 800 欧左右），PC 口用于控制 8 个 LED 的位选。ATmega16 使用内部 4MHz 晶振。

系统还使用 ATmega16 片内的计数/定时器 T1，设计 T1 工作在定时溢出中断方式，定时间隔为 2ms，即 T1 每 2ms 产生一次中断。5 次中断得到 10ms 的时间间隔，此时时钟的 1/100 秒加 1，并相应进行时、分、秒的调整。

LED 动态扫描方式的设计如下：在每 2ms 的时间中，点亮 8 个 LED 中的一个，显示该位相应的数字（PC 口的输出只有一位为低电平，选通一个 LED，保持 2ms）。因此 PC 口的输出值为 0b11111110，每隔 2ms 循环右移，到 0b01111111 时 8 个 LED 各点亮一次，时间为 16ms。在 1 秒钟内，循环 8 个 LED 的次数为 62.5（1000/16），是人眼的滞留时间（25 次/秒）的 2.5 倍，保证了 LED 显示亮度均匀，无闪烁。在程序设计中，在各个 LED 转换和 7 段码输出时，关闭位选信号（PC 输出 0b11111111），消除了显示的拖尾现象（消影功能）。

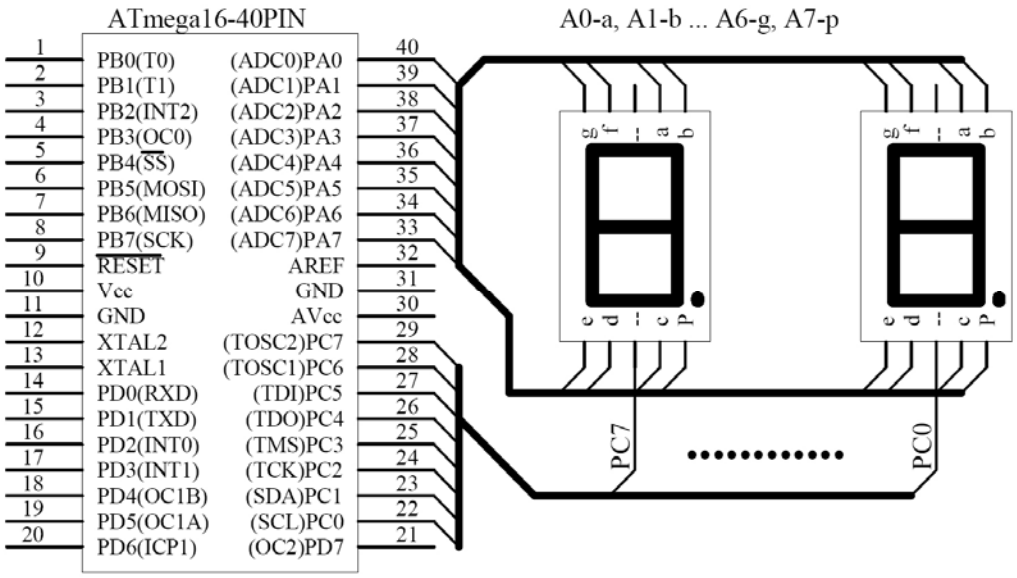

图 5-23　简易 24 小时时钟硬件原理图

T1 的设计：T1 为 16 位定时器，系统时钟为 4M，采用其 64 分频后的时钟作为 T1 的计数信号（寄存器 TCCR1B = 0x03），一个计数周期为 16us，2ms 需要计 125 个（0x007D）。由于 T1 溢出中断发生在 0xFFFF 后下一个 T1 计数脉冲的到来（参见第 8 章关于定时器原理与应用部分），因此 T1 的计数初始值为 0xFF83 = 0xFFFF － 0x007C（65535-124），即寄存器 TCNT1 的初值为 0xFF83。

5.5.2　AVR汇编源代码

该系统的汇编源代码如下，开发软件平台使用 AVR Studio 4.12。

```
;*******************************************************
;AVR 汇编程序实例:Demo_5_2.asm
;简易带 1/100 秒的 24 小时制时钟
;Mega16  4MHz
;*******************************************************
.include "m16def.inc"                ;引用器件 I/O 配置文件

;定义程序中使用的变量名（在寄存器空间）
.def count      =   r18              ;循环计数单元
.def position   =   r19              ;LED 显示位指针,取值为 0-7
.def p_temp     =   r20              ;LED 显示位选,其值取反由 PC 口输出
.def count_10ms =   r21              ;10ms 计数单元
.def flag_2ms   =   r22              ;2ms 到标志
.def temp       =   r23              ;临时变量
.def temp1      =   r24              ;临时变量
.def temp_int   =   r25              ;临时变量（在中断中使用）

;中断向量区定义,flash 程序空间$000-$029
.org $000
```

```
   rjmp reset           ;复位处理
   nop
   reti      ;IRQ0 Handler
   nop
   reti      ;IRQ1 Handler
   nop
   reti      ;Timer2 Compare Handler
   nop
   reti      ;Timer2 Overflow Handler
   nop
   reti      ;Timer1 Capture Handler
   nop
   reti      ;Timer1 Compare-A Handler
   nop
   reti      ;Timer1 Compare-B Handler
   nop
   rjmp time1_ovf   ;Timer1 Overflow Handler
   nop
   reti      ;Timer0 Overflow Handler
   nop
   reti      ;SPI Transfer Complete Handler
   nop
   reti      ;USART RX Complete Handler
   nop
   reti      ;USART UDR Empty Handler
   nop
   reti      ;USART TX Complete Handler
   nop
   reti      ;ADC Conversion Complete Handler
   nop
   reti      ;E2PROM Ready Handler
   nop
   reti      ;Analog Comparator Handler
   nop
   reti      ;Two-wire Serial Interface Handler
   nop
   reti      ;IRQ2 Handler
   nop
   reti      ;Timer0 Compare Handler
   nop
   reti      ;SPM Ready Handler
   nop

;程序开始
.org $02A
reset:
   ldi r16,high(RAMEND)     ;设置堆栈指针高位
   out sph,r16
   ldi r16,low(RAMEND)      ;设置堆栈指针低位
```

```
   out spl,r16

   ser temp
   out ddra,temp            ;设置 PORTA 为输出,段码输出
   out ddrc,temp            ;设置 PORTC 为输出,位码控制
   out portc,temp           ;PORTC 输出$FF, 无显示

   ldi position,0x00        ;段位初始化为 1/100 秒低位
   ldi p_temp,0x01          ;LED 第 1 位亮

;初始化时钟时间为 11:59:55:00
   ldi xl,low(time_buff)    ;
   ldi xh,high(time_buff)   ;X 寄存器取得时钟单元首指针
   ldi temp,0x00
   st  x+,temp      ;1/100 秒 = 00
   ldi temp,0x55
   st  x+,temp      ;秒 = 55
   ldi temp,0x59
   st  x+,temp      ;分 = 59
   ldi temp,0x11
   st  x,temp       ;时 = 11

   ldi temp,0xff        ;T1 初始化,每隔 2ms 中断一次
   out tcnt1h,temp
   ldi temp,0x83
   out tcnt1l,temp
   clr temp
   out tccr1a,temp
   ldi temp,0x03        ;4M,64 分频 2ms
   out tccr1b,temp
   ldi temp,0x04
   out timsk,temp       ;允许 T1 溢出中断
   sei                  ;全局中断允许

;主程序
main:
   cpi flag_2ms,0x01        ;判 2ms 到否
   brne main                ;No,转 main 循环
   clr flag_2ms             ;到,请 2ms 标志
   rcall display            ;调用 LED 显示时间（动态扫描显示一位）
d_10ms_ok:
   cpi count_10ms,0x05      ;判 10ms 到否
   brne main                ;No,转 main 循环
   clr count_10ms           ;10ms 到,清零 10ms 计数器
   rcall time_add           ;调用时间加 10ms 调整
   rcall put_t2d            ;将新时间值放入显示缓冲单元
   rjmp main                ;转 main 循环

;LED 动态扫描显示子程序,2ms 执行一次,一次点亮一位,8 位循环
```

```
display:
   clr r0
   ser temp                 ;temp = 0x11111111
   out portc,temp           ;关显示,去消影和拖尾作用
   ldi yl,low(display_buff)
   ldi yh,high(display_buff)    ;Y 寄存器取得显示缓冲单元首指针
   add yl,position              ;加上要显示的位值
   adc yh,r0                    ;加上低位进位
   ld temp,y                    ;temp 中为要显示的数字

   clr r0
   ldi zl,low(led_7 * 2)
   ldi zh,high(led_7 * 2)       ;Z 寄存器取得 7 段码组的首指针
   add zl,temp                  ;加上要显示的数字
   adc zh,r0                    ;加上低位进位
   lpm                          ;读对应七段码到 R0 中
   out porta,r0                 ;LED 段码输出

   mov r0,p_temp
   com r0
   out portc,r0      ;输出位控制字,完成 LED 一位的显示

   inc position      ;调整到下一次显示位
   lsl p_temp
   cpi position,0x08
   brne display_ret
   ldi position,0x00
   ldi p_temp,0x01
display_ret:
   ret

;时钟时间调整,加 0.01 秒
time_add:
   ldi xl,low(time_buff)    ;
   ldi xh,high(time_buff)   ;X 寄存器为时钟单元首指针
   rcall dhm3           ;ms 单元加 1 调整
   cpi temp,0x99        ;
   brne time_add_ret    ;未到 99ms 返回
   rcall dhm            ;秒单元加 1 调整
   cpi temp,0x60
   brne time_add_ret    ;未到 60 秒返回
   rcall dhm            ;分单元加 1 调整
   cpi temp,0x60
   brne time_add_ret    ;未到 60 分返回
   rcall dhm            ;时单元加 1 调整
   cpi temp,0x24
   brne time_add_ret    ;未到 24 时返回
   clr temp
   st x,temp            ;到 24 时,时单元清另
```

```
time_add_ret:
   ret

;低段时间清零,高段时间加 1,BCD 调整
dhm:  clr temp          ;当前时段清零
dhm1: st  x+,temp       ;当前时段清零,X 寄存器指针加一
dhm3: ld  temp,x        ;取出新时段数据
      inc temp          ;加一
      cpi temp,0x0A     ;若个位数码未到$0A(10)
      brhs dhm2         ;例如$58+1=$59，不须调整;
      subi temp,0xFA    ;否则做减$FA 调整：例如$49+1-$FA=$50
dhm2: st x,temp         ;并将调整结果送回
      ret

;将时钟单元数据送 LED 显示缓冲单元中
put_t2d:
   ldi xl,low(time_buff)        ;
   ldi xh,high(time_buff)       ;X 寄存器时钟单元首指针
   ldi yl,low(display_buff)
   ldi yh,high(display_buff)    ;Y 寄存器显示缓冲单元首指针
   ldi count,4                  ;循环次数 = 4
loop:
   ld   temp,x+             ;读一个时间单元
   mov   temp1,temp
   swap temp1
   andi temp1,0x0f          ;高位 BCD 码
   andi temp,0x0f           ;低位 BCD 码
   st y+,temp               ;写入 2 个显示单元
   st y+,temp1              ;低位 BCD 码在前,高位在后
   dec   count
   brne loop                ;4 个时间单元->8 个显示单元
   ret

;T1 时钟溢出中断服务
time1_ovf:
   in temp_int,sreg
   push temp_int            ;保护状态寄存器

   ldi temp_int,0xff        ;T1 初始值设定,2ms 中断一次
   out tcnt1h,temp_int
   ldi temp_int,0x83
   out tcnt1l,temp_int

   inc count_10ms           ;10ms 计数器加一
   ldi flag_2ms,0x01        ;置 2ms 标志到

   pop temp_int
   out sreg,temp_int        ;恢复状态寄存器
   reti                     ;中断返回
```

```
.CSEG              ;LED 七段码表,定义在 Flash 程序空间
led_7:             ;7 段码表
.db 0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07
.db 0x7f,0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71

;字 PA7 PA6 PA5 PA4 PA3 PA2 PA1 PA0 共阴极 共阳极
;    h   g   f   E   d   c   b   a
;0   0   0   1   1   1   1   1   1   3FH  C0H
;1   0   0   0   0   0   1   1   0   06H  F9H
;2   0   1   0   1   1   0   1   1   5BH  A4H
;3   0   1   0   0   1   1   1   1   4FH  B0H
;4   0   1   1   0   0   1   1   0   66H  99H
;5   0   1   1   0   1   1   0   1   6DH  92H
;6   0   1   1   1   1   1   0   1   7DH  82H
;7   0   0   0   0   0   1   1   1   07H  F8H
;8   0   1   1   1   1   1   1   1   7FH  80H
;9   0   1   1   0   1   1   1   1   6FH  90H
;A   0   1   1   1   0   1   1   1   77H  88H
;b   0   1   1   1   1   1   0   0   7CH  83H
;C   0   0   1   1   1   0   0   1   39H  C6H
;d   0   1   0   1   1   1   1   0   5EH  A1H
;E   0   1   1   1   1   0   0   1   79H  86H
;F   0   1   1   1   0   0   0   1   71H  8EH

.DSEG              ;定义程序中使用的变量位置（在 RAM 空间）
.ORG     $0060
display_buff:          ;LED 显示缓冲区,8 个字节
.BYTE   0x00           ;LED 1 位显示内容
.BYTE   0x00           ;LED 2 位显示内容
.BYTE   0x00           ;LED 3 位显示内容
.BYTE   0x00           ;LED 4 位显示内容
.BYTE   0x00           ;LED 5 位显示内容
.BYTE   0x00           ;LED 6 位显示内容
.BYTE   0x00           ;LED 7 位显示内容
.BYTE   0x00           ;LED 8 位显示内容

.org   $0068
time_buff:             ;时钟数据缓冲区,4 个字节
.BYTE   0x00           ;1/100s 单元
.BYTE   0x00           ;秒单元
.BYTE   0x00           ;分单元
.BYTE   0x00           ;时单元
```

程序实例采用比较规范标准的设计理念和风格，程序中已给出比较详细的注解。关于程序如何具体完成和实现系统的功能请读者仔细阅读程序，用心体会。下面仅对编写 M16 汇编程序时，在结构和语句使用上一些需要注意的方面加以介绍。

- 将程序中操作最频繁以及需要特殊位处理的变量定义在 AVR 的 32 个工作寄存器空间，因为 MCU 对 R0-R31 的操作仅需要一个时钟周期，而且功能强大。由于 R0-R31

的功能有不同，而且也仅有 32 个，所以程序员应认真考虑和规划这 32 个工作寄存器的使用。如尽量不要将变量放置在 R26-R31 中，因为这 6 个寄存器构成 3 个 16 位的 X、Y、Z 地址指针寄存器，应保留用于各种寻址使用。

- M16 有 21 个中断源，Flash 程序存储器的低段空间为这 21 个中断向量地址。注意：M16 的一个中断向量地址空间为 2 个字长度（4 字节）。在中断向量处可使用长转移指令 jmp（2 字）或 rjmp 转移指令（1 字）跳转到中断服务程序。而有些 AVR 的一个向量地址空间为 1 个字长度，只能使用 rjmp 转移指令。
- 出于提高系统可靠性的设计，对于系统中不使用的中断向量，应填充 2 个中断返回指令 reti（每个 reti 占一个字）。在本程序中，为了程序的理解和阅读方便，使用 rjmp 和 nop，以及 reti 和 nop 指令填充一个 2 个字长度的向量地址空间。
- 程序中使用了 X、Y、Z 三个 16 位的地址指针寄存器，基于他们的一些指令有自动加（减）一的功能，以及先加（减）、后使用，和先使用、后加（减）的区别，在使用中应注意正确和灵活的使用。
- 由于 LED 的七段码对照表是固定不变的，程序中将 LED 的七段码表放置在 Flash 存储器中。对于 Flash 存储器的间址取数只能使用 Z 寄存器。由于程序存储器的地址是以字（双字节）为单位的，因此，16 位地址指针寄存器 Z 的高 15 位为程序存储器的字地址，最低位 LSB 为“0”时，指字的低字节；为“1”时，指字的高字节。程序中使用伪指令 db 定义的七段码为一个字节，他保存在一个字的低字节处。如果定义字，应使用伪指令 dw。
- 本例使用指令 lpm 读取 Flash 中的一个字节，因此在取七段码表的首地址时乘 2（ldi zl,low(led_7 * 2)），将地址左移一位，Z 寄存器的 LSB 为“0”，表示取该字的低位字节。
- 中断服务程序中，必须对 MCU 的标志寄存器 SREG 进行保护。在 T1 的溢出中断服务程序中，还需要对 TCNT1 的初值进行设置，以保证下一次中断仍为 2ms。中断服务程序应尽量短小，因此在中断服务中，只将 2ms 标志置位和 10ms 加一计数，其它处理应尽量放在主程序中。
- 程序中定义了 8 个字节的显示缓冲区和 4 个字节的时钟数据缓冲区，分别存放 8 个 LED 所对应的显示数字和 4 个时间段的时间值（BCD 码），这 12 个单元定义放置在 M16 的 RAM 中。M16 的 RAM 单元应从 0x0060 开始，前面的地址分别对应的是 32 个工作寄存器、I/O 寄存器，因此不要把一般的数据单元定义在小于 0x0060 的空间。
- 与使用 db 或 dw 伪指令在 Flash 空间定义常量不同的是，在 RAM 空间予留变量空间的定义应使用 byte 伪指令。byte 伪指令的功能是定义变量的位置（予留空间），不能定义（填充）变量的值，变量具体的值是需要由程序在运行中写入的。而伪指令 db、dw 具有数据位置和值定义（填充）的功能。

思考与练习

关于硬件设计的讨论：

1. 图5-1中的R1是否可以不用，将RESET悬空或直接与Vcc连接。
2. 如果将图5-1中的LED正端接PC0，由I/O口控制，而负端接R3，R3的另一端接GND，这样的设计可以吗？程序需要做那些调整？这样的设计与图5-1中设计方式哪种好一些？为什么。
3. 图5-1中R3起什么作用？其阻值在什么范围比较合适。
4. 如果要用AVR的3个I/O口控制一个3相的步进马达（5V/300MA），硬件电路应该如何设计。
5. M16的熔丝位如何设置才能与使用的系统时钟源的类型相配合，如果熔丝位的设置与实际系统时钟源的类型不符合，系统会出现那些现象，为什么？如何处理。

6. 讨论M16熔丝位中低电压检测复位（BOD）的作用。在本例中设置DOB的检测电平为4.0V，并起启用了低电压检测复位功能，思考其在实际应用中有和好处。

关于软件的讨论

7. 在本章的例程中，为什么第一句语句都使用了INCLUDE伪指令（语句）？不用可以吗？如果不用，程序的开始应该如何写？
8. 查找汇编中要包含的“M16DEF.INC”文件和C代码中包含“mega16.h”文件在硬盘的何处？两个文件的内容是什么。
9. 在汇编程序中出现的RAMEND（见下题程序段）代表什么，在本章汇编程序中它的值是多少？从哪里来的。
10. demo_5_1.asm开始的语句如下：

```
.org $0000          ;上电复位启始地址
    rjmp reset      ;转上电复位后的初始化程序执行
;中断向量区
.org $002A          ;跳过中断向量区
reset:
    ldi r16,high(RAMEND)     ;取内部RAM最高地址的高位字节
    out sph,r16              ;放入SP的高位
    ldi r16,low(RAMEND)      ;取内部RAM最低地址的低位字节
    out spl,r16              ;放入SP的低位，SP中的值见器件配置文件“m16def.inc”
```

请具体分析以上语句的作用。如果后4句不要可以吗，系统运行会出现什么情况，为什么？请在AVR Studio中用软件模拟的方式观察并解答问题。

11. 仔细分析demo_5_1.asm中延时子程序的结构，运行情况，堆栈和堆栈指针的变化，利用AVR Studio中的软件模拟器进行分析。
12. 仔细查看使用AVR Studio和CVAVR开发编写AVR系统软件后所生成的各种类型的文件，以及这些文件的内容，并思考和分析这些文件的作用。

本章参考文献：
1.《AVR Studio在线帮助》（中文，CDROM），ATMEL，www.atmel.com
2.《CVAVR应用入门》（英文，CDROM），ATMEL，www.atmel.com
3.《CVAVR 使用手册》（英文，CDROM）
4.《CVAVR 使用参考》（中文，CDROM）
5.《cvavr 库函数介绍》（中文，CDROM）

第 6 章 通用I/O接口基本结构与输出应用

从本章开始，将从 AVR 单片机的基本功能单元入手，讲解其各个外围功能部件的基本组成和特性，以及它们的应用。

ATmega16 芯片有 PORTA、PORTB、PORTC、PORTD（简称 PA、PB、PC、PD）4 组 8 位，共 32 路通用 I/O 接口，分别对应于芯片上 32 根 I/O 引脚。所有这些 I/O 口都是双（有的为 3）功能复用的。其中第一功能均作为数字通用 I/O 接口使用，而复用功能则分别用于中断、时钟/计数器、USRAT、I2C 和 SPI 串行通信、模拟比较、捕捉等应用。这些 I/O 口同外围电路的有机组合，构成各式各样的单片机嵌入式系统的前向、后向通道接口，人机交互接口和数据通信接口，形成和实现了千变万化的应用。

由于刚开始学习 I/O 的应用，读者还没有掌握中断和定时计数器的使用，所以在本章的实例中，调用了 CVAVR 提供的软件延时函数来实现时间延时等待的功能。

需要指出的是，使用软件延时的方式会造成 MCU 效率的下降，而且也不能实现精确的延时，所以在一般情况下应尽量不使用软件延时的方式。在后面的章节里，会逐步介绍如何使用 T/C 和中断实现延时的正确方法。

6.1 通用I/O口的基本结构与特性

6.1.1 I/O口的基本结构

图 6-1 为 AVR 单片机通用 I/O 口的基本结构示意图。从图中可以看出，每组 I/O 口配备三个 8 位寄存器，它们分别是方向控制寄存器 DDRx，数据寄存器 PORTx，和输入引脚寄存器 PINx（x=A\B\C\D）。I/O 口的工作方式和表现特征由这 3 个 I/O 口寄存器控制。

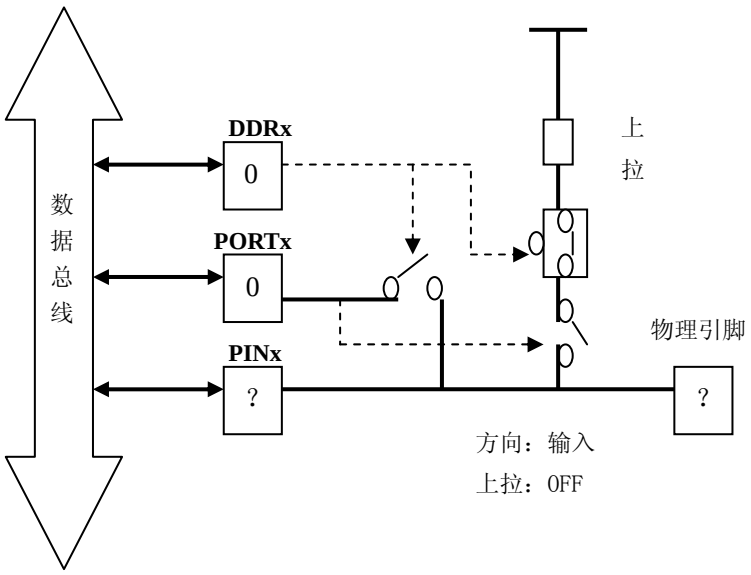

图 6-1 通用 I/O 口结构示意图

方向控制寄存器 DDRx 用于控制 I/O 口的输入输出方向，即控制 I/O 口的工作方式为输出方式还是输入方式。

当 DDRx=1 时，I/O 口处于输出工作方式。此时数据寄存器 PORTx 中的数据通过一个推挽电路输出到外部引脚（图 6-2）。AVR 的输出采用推挽电路提高了 I/O 口的输出能力，当 PORTx=1 时，I/O 引脚呈现高电平，同时可提供输出 20mA 的电流；而当 PORTx=0 时，I/O 引脚呈现低电平，同时可吸纳 20mA 电流。因此，AVR 的 I/O 在输出方式下提供了比较大的驱动能力，可以直接驱动 LED 等小功率外围器件。

当 DDRx=0 时，I/O 处于输入工作方式。此时引脚寄存器 PINx 中的数据就是外部引脚的实际电平，通过读 I/O 指令可将物理引脚的真实数据读入 MCU。此外，当 I/O 口定义为输入时（DDRx=0），通过 PORTx 的控制，可使用或不使用内部的上拉电阻（图 6-3）。

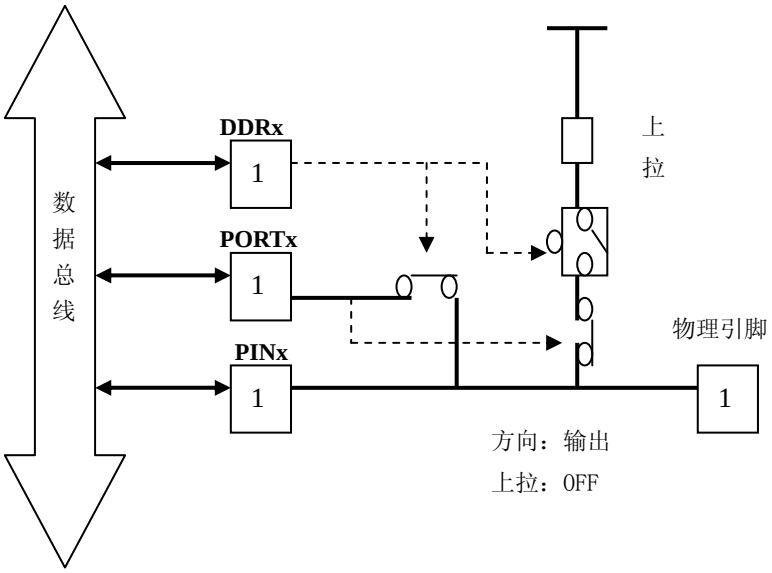

图 6-2　通用 I/O 口输出工作方式示意图

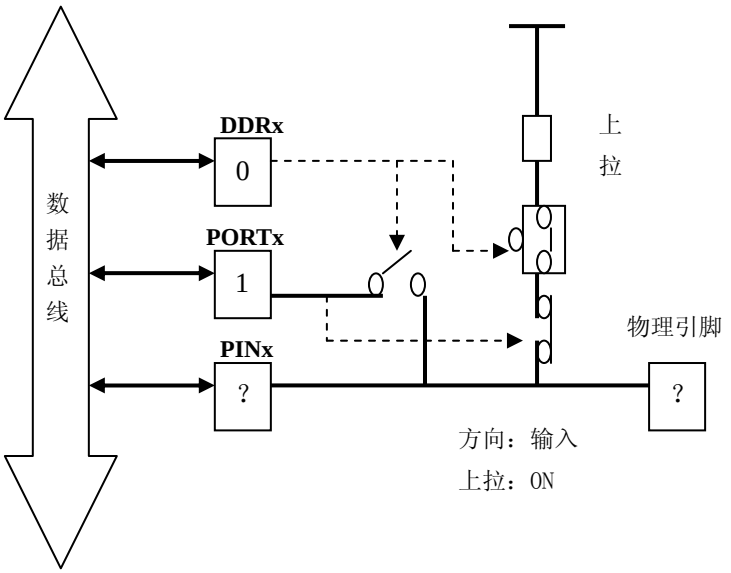

图 6-3　通用 I/O 口输入工作方式示意图（带内部上拉）

表 6.1 是 AVR 通用 I/O 端口的引脚配置情况。

表 6.1 I/O 口引脚配置表

DDRXn	PORTXn	PUD	I/O 方式	内部上拉电阻	引脚状态说明
0	0	X	输入	无效	三态（高阻）
0	1	0	输入	有效	外部引脚拉低时输出电流(uA)
0	1	1	输入	无效	三态（高阻）
1	0	X	输出	无效	推挽 0 输出，吸收电流(20mA)
1	1	X	输出	无效	推挽 1 输出，输出电流(20mA)

表中的 PUD 为寄存器 SFIOR 中的一位，它的作用相当 AVR 全部 I/O 口内部上拉电阻的总开关。当 PUD=1 时，AVR 所有 I/O 内部上拉电阻都不起作用（全局内部上拉无效）；而 PUD=0 时，各个 I/O 口内部上拉电阻取决于 DDRXn 的设置。

AVR 通用 I/O 端口的主要特点为：

✓ 双向可独立位控的 I/O 口

ATmega16 的 PA、PB、PC、PD 四个端口都是 8 位双向 I/O 口，每一位引脚都可以单独的进行定义，相互不受影响。如用户可以在定义 PA 口第 0、2、3、4、5、6 位用于输入的同时定义第 1、7 位用于输出，互不影响。

✓ Push-Pull 大电流驱动（最大 40mA）

每个 I/O 口输出方式均采用推挽式缓冲器输出，提供大电流的驱动，可以输出（吸入）20mA 的电流，因而能直接驱动 LED 显示器。

✓ 可控制的引脚内部上拉电阻

每一位引脚内部都有独立的，可通过编程设置的，设定为上拉有效或无效的内部上拉电阻。当 I/O 口被用于输入状态，且内部上拉电阻被激活（有效）时，如果外部引脚被拉低，则构成电流源输出电流（uA 量级）。

✓ DDRx 可控的方向寄存器。

AVR 的 I/O 端口结构同其它类型单片机的明显区别是，AVR 采用 3 个寄存器来控制 I/O 端口。一般单片机的 I/O 仅有数据寄存器和控制寄存器，而 AVR 还多了一个方向控制器，用于控制 I/O 的输入输出方向。由于输入寄存器 PINx 实际不是一个寄存器，而是一个可选通的三态缓冲器，外部引脚通过该三态缓冲器与 MCU 的内部总线连接，因此，读 PINx 时是读取外部引脚上的真实和实际逻辑值，实现了外部信号的同步输入。这种结构的 I/O 端口，具备了真正的读-修改-写（Read-Modify-Write）特性。

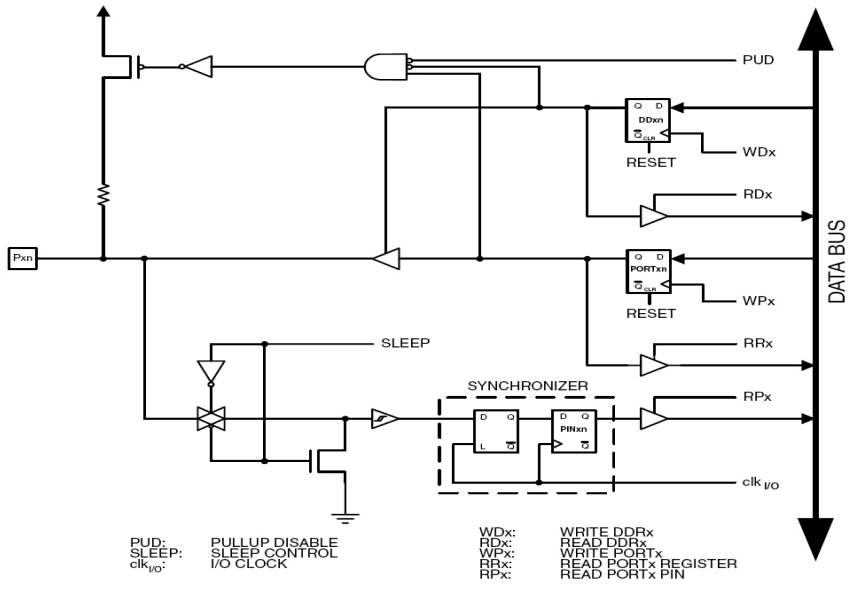

图 6-4 通用 I/O 口逻辑功能示意图

图 6-4 为 AVR 一个（位）通用 I/O 口的逻辑功能图。右上面的两个 D 触发器为方向控制寄存器和数据寄存器。

1） 使用 AVR 的 I/O 口，首先要正确设置其工作方式，确定其工作在输出方式还是输入方式。
2） 当 I/O 工作在输入方式，要读取外部引脚上的电平时，应读取 PINxn 的值，而不是 PORTxn 的值。
3） 当 I/O 工作在输入方式，要根据实际情况使用或不使用内部的上拉电阻。
4） 一旦将 I/O 口的工作方式由输出设置成输入方式后，必须等待一个时钟周期后才能正确的读到外部引脚 PINxn 的值。

上面的第 4 点是由于在 PINxn 和 AVR 内部数据总线之间有一个同步锁存器（图 6-4 中的 SYNCHRONIZER）电路，使用该电路避免了当系统时钟变化的短时间内外部引脚电平也同时变化而造成的信号不稳定的现象，但它有产生大约一个时钟周期（0.5～1.5）的时延。

6.1.2 I/O端口寄存器

ATmega16 的 4 个 8 位的端口都有各自对应的 3 个 I/O 端口寄存器，它们占用了 I/O 空间的 12 个地址（见表 6.2）。

表 6.2 ATmega16 I/O 寄存器地址表

名 称	I/O 空间地址	RAM 空间地址	作 用
PORTA	$1B	0x003B	A 口数据寄存器
DDRA	$1A	0x003A	A 口方向寄存器
PINA	$19	0x0039	A 口输入引脚寄存器
PORTB	$18	0x0038	B 口数据寄存器
DDRB	$17	0x0037	B 口方向寄存器
PINB	$16	0x0036	B 口输入引脚寄存器
PORTC	$15	0x0035	C 口数据寄存器
DDRC	$14	0x0034	C 口方向寄存器
PINC	$13	0x0033	C 口输入引脚寄存器
PORTD	$12	0x0032	D 口数据寄存器
DDRD	$11	0x0031	D 口方向寄存器
PIND	$10	0x0030	D 口输入引脚寄存器

下面是 PA 口寄存器—PORTA、DDRA、PINA 各个位的具体定义，以及其是否可以通过指令读写操作和 RESET 复位后的初始值。其它 3 个口的寄存器的情况与 PA 口相同，只是地址不一样。

位	7	6	5	4	3	2	1	0	
$1B（$003B）	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
复位值	0	0	0	0	0	0	0	0	

位	7	6	5	4	3	2	1	0	
$1A（$003A）	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
复位值	0	0	0	0	0	0	0	0	

位	7	6	5	4	3	2	1	0	
$19（$0039）	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
读/写	R	R	R	R	R	R	R	R	
复位值	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

1） 正确使用 AVR 的 I/O 口要注意：先正确设置 DDRx 方向寄存器，再进行 I/O 口的读写操作。

2） AVR 的 I/O 口复位后的初始状态全部为输入工作方式，内部上拉电阻无效。所以，外部引脚呈现三态高阻输入状态。

3） 因此，用户程序需要首先对要使用的 I/O 口进行初始化设置，根据实际需要设定使用 I/O 口的工作方式（输出还是输入），当设定为输入方式时，还要考虑是否使用内部的上拉电阻。

4） 在硬件电路设计时，如能利用 AVR 内部 I/O 口的上拉电阻，可以节省外部的上拉电阻。

6.1.3 通用数字I/O口的设置与编程

在将AVR 的 I/O 口作为通用数字口使用时,要先根据系统的硬件设计情况,设定各个 I/O 口的工作方式：输入或输出工作方式，既先正确设置 DDRx 方向寄存器，再进行 I/O 口的读写操作。如将 I/O 口定义为数字输入口时，还应注意是否需要将该口内部的上拉电阻设置为有效，在设计电路时，如能利用 AVR 内部 I/O 口的上拉电阻，可以节省外部的上拉电阻。

AVR 汇编指令系统中，直接用于对 I/O 寄存器的操作指令有以下 3 类，全部为单周期指令：

1）IN/OUT

IN/OUT 指令实现了 32 个通用寄存器与 I/O 寄存器之间的数据交换，格式为：

```
IN Rd，A        ；从 I/O 寄存器 A 读数剧到通用寄存器 Rd
OUT A，Rr       ；通用寄存器 Rr 数据送 I/O 寄存器 A
```

2）SBI/CBI

SBI/CBI 指令实现了对 I/O 寄存器（地址空间为 I/O 空间的 0x00-0x31）中指定位的置 1 或清 0，格式为：

```
SBI A，b        ；将 I/O 寄存器 A 的第 b 位置 1
CBI A，b        ；将 I/O 寄存器 A 的第 b 位清 0
```

3）SBIC/SBIS

SBIC/SBIS 指令为转移类指令，它根据 I/O 寄存器（地址空间为 I/O 空间的 0x00-0x31）的指定位的数值实现跳行转移（跳过后面紧接的一条指令，执行后序的第二条指令），格式为：

```
SBIC A，b           ；I/O 寄存器 A 的第 b 位为 0 时，跳行执行
SBIS A，b           ；I/O 寄存器 A 的第 b 位为 1 时，跳行执行
```

ATmega16 的 4 个 8 位的端口共有 12 个 I/O 端口寄存器，它们在 AVR 的 I/O 空间的地址均在前 32 个之中，因此上面 3 类对 I/O 寄存器操作的指令都可以使用。在第 5 章的例程 Demo_5_1.asm 中，使用了 OUT 指令设置 PC 口的工作方式为输出，输出全 1：

```
.def temp1=r20                  ;定义寄存器R20用临时变量名temp1代表
°°°°°°°°°
ser temp1                       ;置temp1（R20）为0XFF
out ddrc,temp1                  ;定义PC口为输出
out portc,temp1                 ;PC口输出全“1”，LED不亮
```

在 CVAVR 中，我们可以直接使用 C 的语句对 I/O 口寄存器进行操作，如：

```
// 定义PortC口的工作方式
PORTC = 0x01;               // PC口的第0位输出“1”，LED不亮
DDRC = 0x01;                // 定义PC口的第0位为输出方式
PORTC.0 = ~PORTC.0;         // PC口第0位输出取反
```

其中 PORTC.0 = 0 (或 PORTC.0 = 1)是 CVAVR 中对 C 的扩展语句，它实现了对寄存器的位操作。这种语句在标准 C 中是没有的，该扩展更加适合编写单片机的系统程序，因为在单片机的系统程序中，是经常需要直接对位进行操作的。

更加标准的 C 程序可以采用以下的写法：

```
#define BIT0 0
#define BIT1 1
#define BIT2 2
#define BIT3 3
#define BIT4 4
#define BIT5 5
#define BIT6 6
#define BIT7 7
……
PORTC = 1<<(BIT0) | 1<<(BIT3);   // PC口的第0位和第3位输出“1”，其它为“0”
```

这里，1<<(BIT0)表示逻辑1左移0位，结果为0b00000001;而1<<(BIT3) 表示逻辑1左移3位，结果为0b00001000。0b00000001在同0b00001000相与，结果为0b00001001。以上的逻辑运算不产生具体的操作指令，是由CVAVR在编译时运算完成，得到结果，最后只是产生将结果赋值到PORTC寄存器的操作指令。

这种表示方式，比直接赋值0b00001001更容易理解程序的作用，如在下面的有关AVR的USART串口的程序中大量使用了这样的描述方式。

```
#define RXB8 1
#define TXB8 0
#define UPE 2
#define OVR 3
#define FE 4
#define UDRE 5
#define RXC 7

#define FRAMING_ERROR (1<<FE)
#define PARITY_ERROR (1<<UPE)
#define DATA_OVERRUN (1<<OVR)
```

```
#define DATA_REGISTER_EMPTY (1<<UDRE)
#define RX_COMPLETE (1<<RXC)
……
char status;
status = UCSRA;
if (( status & ( FRAMING_ERROR | PARITY_ERROR | DATA_OVERRUN )) == 0 )
{……}                          // 接收数据无错误处理过程
else
{……}                          //接收数据产生错误处理过程
```

程序中的 UCSRA 为 ATmega16 的串行接口 USART 的状态寄存器，UPE 是 UCSRA 的第 2 位，当 UPE 为 1 时表示接收到的数据产生了校验错误。程序中采用了定义语句，定义 PARITY_ERROR 为（1 << UPE），实际就是 0b00000100。因此一旦 USART 的值为 PARITY_ERROR 时，表示接受的数据产生了校验错误，使程序的阅读非常明了。

这样的程序编写方式，在 AVR 的汇编中也是可以使用的。

6.2 通用I/O口的输出应用

6.2.1 通用I/O输出设计要点

将通用 I/O 口定义为输出工作方式，通过设置该口的数据寄存器 PORTx，就可以控制对应 I/O 口外围引脚的输出逻辑电平，输出“0”或“1”。这样我们就可以通过程序来控制 I/O 口，输出各种类型的逻辑信号，如方波脉冲，或控制外围电路执行各种动作。

在应用 I/O 口输出时，在系统的软硬件设计上应注意的问题有：

- ✓ 输出电平的转换和匹配。如一般 AVR 系统的工作电源为 5 伏（手持系统往往采用 1.5v－3v 电源），所以 I/O 的输出电平为 5v。当连接的外围器件和电路采用 3v、9v、12v、15v 等与 5v 不同的电源时，应考虑输出电平转换电路。
- ✓ 输出电流的驱动能力。AVR 的 I/O 口输出为“1”时，可以提供 20mA 左右的驱动电流。输出为“0”时，可以吸收 20mA 左右的灌电流（最大为 40mA）。 当连接的外围器件和电路需要大电流驱动或有大电流灌入时，应考虑使用功率驱动电路。
- ✓ 输出电平转换的延时。AVR 是一款高速单片机，当系统时钟为 4M 时，执行一条指令的时间为 0.25us，这意味着将一个 I/O 引脚置“1”，再置“0”仅需要 0.25us，既输出一个脉宽为 0.25us 高电平脉冲。在一些应用中，往往需要较长时间的高电平脉冲驱动，如步进马达的驱动，动态 LED 数码显示器的扫描驱动等，因此在软件设计中要考虑转换时间延时。对于不需要精确延时的应用，可采用软件延时的方法，编写软件延时的子程序。如果要求精确延时，要使用 AVR 内部的定时器。

6.2.2 LED发光二极管的控制

LED 发光二极管是一种经常使用的外围器件，用于显示系统的工作状态，报警提示等，用大量的发光二极管组成方阵，就是构成一个 LED 电子显示屏，可以显示汉字和各种图形，如体育场馆中的大型显示屏。下面设计一个带有一排 8 个发光二极管的简易彩灯控制系统。

例6.1 简易彩灯控制系统

1） 硬件电路设计：

发光二极管一般为砷化镓半导体二极管，其电路图 6-5 所示。当电压 U1 大于 U2 约 1V 以上时，二极管导通发光。当导通电流大于 5mA 时，人的眼睛就可以明显地观察到二极管的

发光，导通电流越大，亮度越高。一般导通电流不要超过 10mA，否则将导致二极管的烧毁或 I/O 引脚的烧毁。因此在设计硬件电路时，要在 LED 二极管电路中串接一个限流电阻，阻值在 300～1000Ω之间，调节阻值的大小可以控制发光二极管的发光亮度。导通电流与限流电阻之间的关系由下面的计算公式确定：

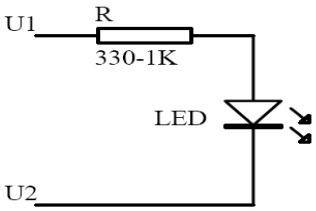

图 6-5 LED 电路

$$I = \frac{U1 - U2 - Vled}{R}$$

式中，Vled 为 LED 的导通电压。

由于 AVR 的 I/O 口输出“0”时，可以吸收最大 40mA 的电流，因此采用控制发光二极管负极的设计比较好。8 个 LED 发光二极管控制系统的硬件电路见图 6-6。

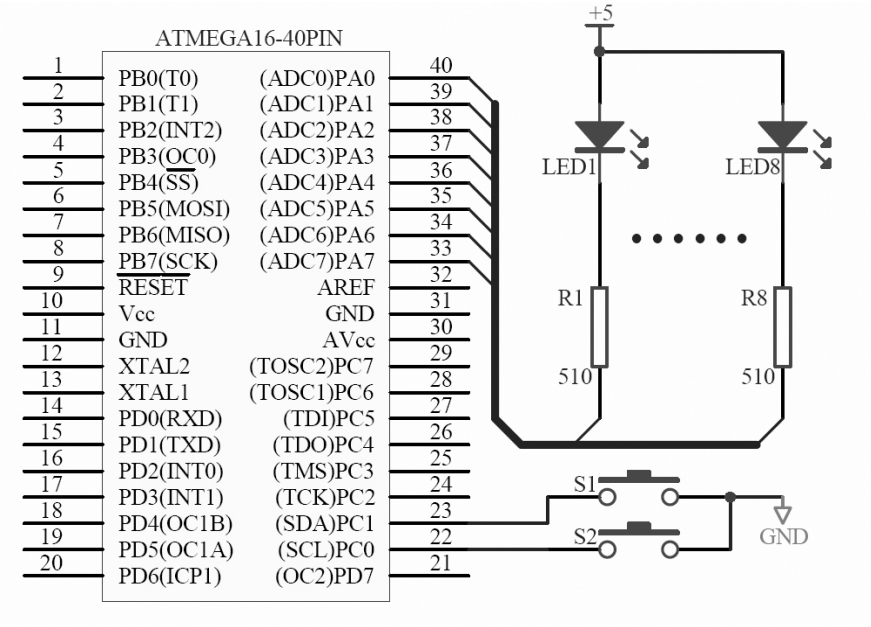

图 6-6 8 路 LED 发光二极管控制电路

在图 6-6 中，ATmega16 的 PA 口工作在输出方式下，8 个引脚分别控制 8 个发光二极管。当 I/O 口输出“0”时 LED 导通发光，输出“1”时 LED 截止熄灭。

2) 软件设计

下面给出一个简单的控制程序，其完成的功能是 8 个 LED 逐一循环发光 1 秒，构成“走马灯”。程序非常简单，请读者自己分析。

```
/*********************************************
file name           : demo_6_1.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
```

```
*******************************************/

#include <mega16.h>
#include <delay.h>

void main(void)
{
    char position = 0;              // position 为控制位的位置

    PORTA=0xFF;                     // PA 口输出全 1，LED 全灭
    DDRA=0xFF;                      // PA 口工作为输出方式

    while (1)
    {
        PORTA = ~(1<<position);     //
        if (++position >= 8) position = 0;
        delay_ms(1000);
    };
}
```

3） 思考与实践

- ✓ 调整程序中的 delay_ms()，延时时间为 1ms，彩灯闪亮有何变化？为什么？
- ✓ 计算并验证当延时时间小于多少毫秒时，“走马灯”的效果变成“全亮”？给出计算方法。
- ✓ 设计一个 4 种闪烁方式交替循环的彩灯，闪烁方式见图 6-7。

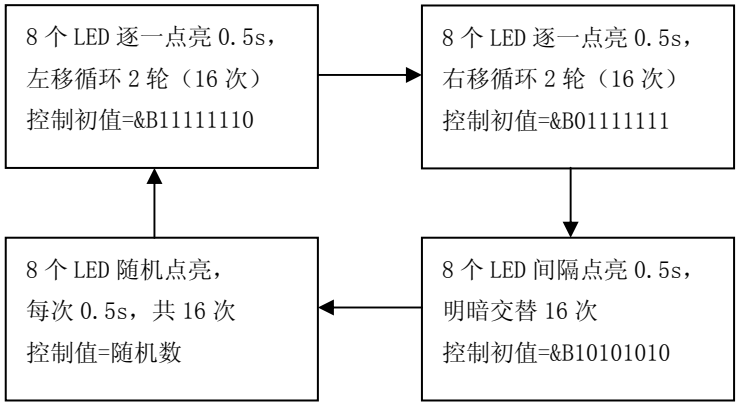

图 6-7 4 种不同控制方式的转换

提示：在 CVAVR 中，提供 int rand (void)和 void srand(int seed)函数，请参考 CVAVR 手册，尝试使用这两个函数。

6.2.3 继电器控制

在工业控制以及许多场合中，嵌入式系统要驱动一些继电器和电磁开关，用于控制马达的开启和关闭，阀门的开启和关闭等。继电器和电磁开关需要功率驱动，驱动电流往往需要几百毫安，超出了 AVR 本身 I/O 口的驱动能力，因此在外围硬件电路中要考虑使用功率驱动

电路。

例6.2　控制恒温箱的加热的硬件电路设

恒温箱的加热源采用 500W 电炉，电炉的工作电压 220v，电流 2.3A。选用 HG4200 继电器，开关负载能力为 5A/AC220V，继电器吸合线圈的工作电压 5v，功耗 0.36W，计算得吸合电流为 0.36/5 = 72mA。因此，要能使继电器稳定的吸合，驱动电流应该大于 80mA。该电流已经超出 AVR 本身 I/O 口的驱动能力，因此外部需要使用功率驱动元件。

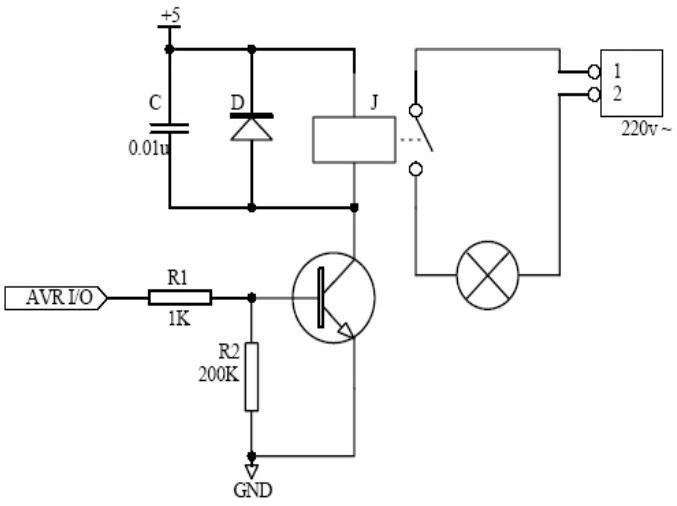

图 6-8 继电器控制电路

设计控制电路如图 6-8 所示，I/O 引脚输出“1”时，三极管导通，继电器吸合，电炉开始加热。I/O 引脚输出“0”时，三极管截止，继电器释放，加热停止。

图中的三极管应采用中功率管，导通电流大于 300mA。电阻 R1 的作用是限制从 I/O 流出的电流太大，保护 I/O 端口，称为限流电阻。注意：三极管集电极的负载继电器吸合线圈在三极管截止时会产生一个很高的反峰电压，在吸合线圈两端并接一个二极管 D，其用途是释放反峰电压，保护三极管和 I/O 口不会被反峰电压击穿，提高系统的可靠性。吸合线圈两端并接的电容 C，能对继电器动作时产生的尖峰电压变化进行有效的过滤，其作用也是为了提高系统的可靠性。在设计 PCB 板时，二极管 D 和电容 C 应该仅靠在继电器的附近。设计中还要考虑系统在上电时的状态。由于 AVR 在上电时，DDRx 和 PORTx 的值均初始化为“0”，I/O 引脚呈高阻输入方式，因此电阻 R2 的作用是确保三极管的基极电位在上电时为“0”电平，三极管截止，保证了加热电炉控制系统上电时不会误动作。

在工业控制中，尤其应认真考虑系统上电初始化时以及发生故障时 I/O 口的状态，应在硬件和软件设计中仔细考虑，否则会产生误动作，造成严重的事故！

在驱动电感性负载时，在硬件上要考虑采取对反峰电压的吸收和隔离，防止对控制系统的干扰和破坏。

6.2.4 步进电机控制

步进电机在自动仪表、自动控制、机器人、自动生产流水线等领域的应用相当广泛，如在打印机、磁盘驱动器、扫描仪中都有步进电机的身影。关于步进电机工作原理请参考有关资料，本节介绍一种普通微型单极 3 相步进电机的控制。

单极 3 相步进电机有三个磁激励相，分别用 A、B、C 表示，每相有一个磁激线圈。通过控制三个磁激线圈电流的通断的先后时间顺序和通断频率就可以改变步进电机的变旋转方向和控制转速，图 6-9 是单极 3 相步进电机的原理图。

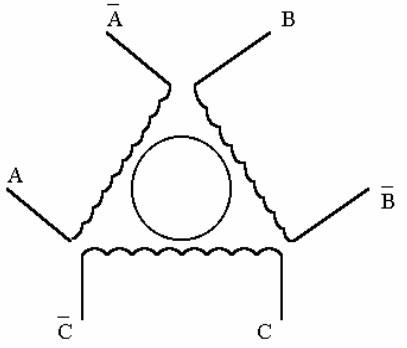

图 6-9 单极 3 相步进电机原理图

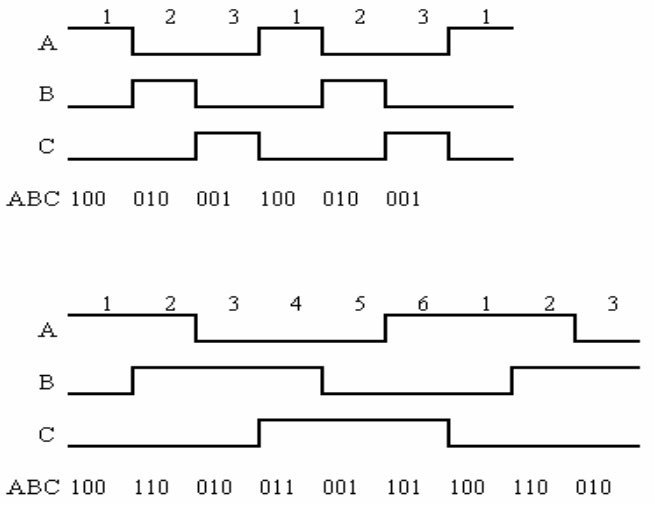

图 6-10 3 相步进电机控制时序图

单极 3 相步进电机有 3 相 3 拍和 3 相 6 拍两种驱动方式，图 6-10 给出它们的控制时序图。3 相 3 拍就是 A、B、C 三相分别通电，正转为 A－B－C－A－B－C，反转为 A－C－B－A－C－B，每拍转动 3°。3 相 6 拍中有三拍是两相同时通电，正转为 A－AB－B－BC－C－CA，反转为 A－AC－C－CB－B－BA，每拍转动 1.5°。

例 6.3 单极 3 相步进电机控制系统

1） 硬件电路设计：

本例使用的3相步进电机，型号为45BC340C，步距角1.5°/3°，相电压12vDC，相电流0.4A，空载启动频率500Hz，控制硬件电路原理图见图6-11。图中采用一片7位达林顿驱动芯片MC1413（Darlington transistor arrays），其驱动电流为0.5A，工作电压达50v。当I/O A输出高电平“1”时，MC1413内部对应Q0的达林顿管导通，电流从电源正极（+12V）流过步进电机A相线圈，经O0端流入地线；输出低电平“0”时，MC1413内部对应O0的达林顿管截止，A相线圈中无电流流过。电阻R1-R3为限流保护电阻。系统上电时，AVR的I/O引脚为高阻，电阻R4-R6将MC1413的3个控制输入端拉低，以保证步进电机在上电时不会产生误动作。

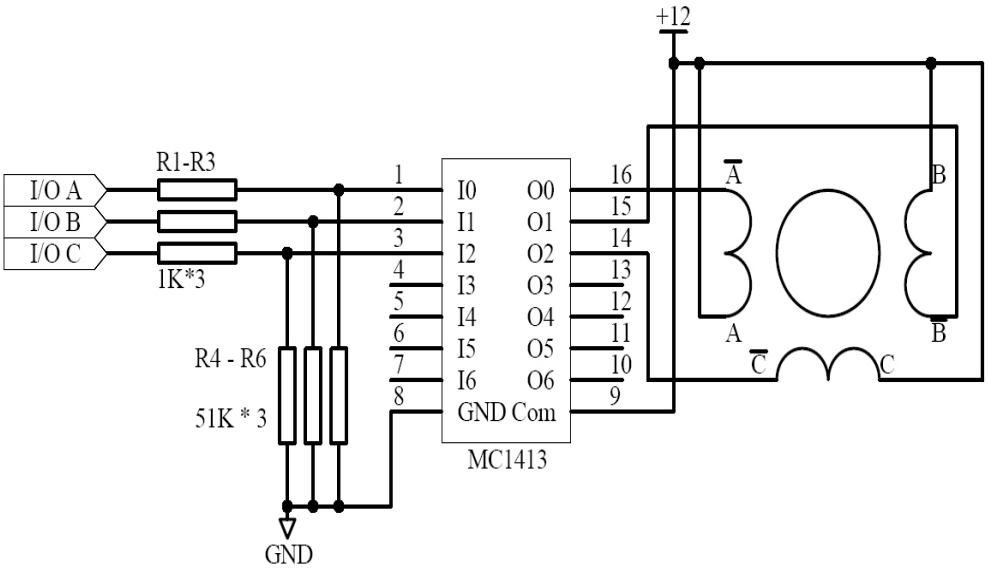

图 6-11 步进电机控制电路

2） 软件设计

```
/*****************************************
file name           : demo_6_3.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

#include <mega16.h>
#include <delay.h>

flash char step_out[6]={0x04,0x06,0x02,0x03,0x01,0x05};

void main(void)
{
    char i = 0;
    int delay = 500;

    PORTA=0x00;
    DDRA=0x07;

    while (1)
    {
        PORTA = step_out[i];
        if (++i >= 6) i = 0;
        delay_ms(delay);
    };
}
```

3） 思考与实践

- ✓ 通过分析 demo_6_3.c，你认为本例中是使用了 AVR 的哪三个 I/O 端口控制步进电机的？采用的是 3 相 3 拍还是 3 相 6 拍的控制方式？电机的转动方向如何？
- ✓ 如果要改变电机的转动方向，仅仅改动硬件路线或只修改软件能够实现吗？各给出一个方案，并检验。
- ✓ 程序中定义的数组“flash char step_out[6]={0x04,0x06,0x02,0x03,0x01,0x05}”的作用是什么？采用如下的定义“char step_out[6]={0x04,0x06,0x02,0x03,0x01,0x05}”可以么？两种定义有何区别？在本例中使用哪种定义比较好，为什么？
- ✓ 程序中变量 Delay 的作用是什么？将 Delay 的数值调大或减小，对电机的转动有何影响？请仔细分析并验证（最好在实际系统上测试）。
- ✓ 本例使用的三相步进电机有一个指标参数“空载启动频率 500Hz”，该参数在软件设计中需要考虑吗？

- ✓ 如何设计软件能控制电机转动的圈数和转速？
- ✓ 设计一段步进马达的控制程序（马达拖动打印机的打印头），它能打印一行字，然后返回起始位置。打印过程为：慢速正转 20 圈（2 圈/秒），高速反转 20 圈（4 圈/秒）。

6.3 LED数码显示器的应用

LED 数码显示器是单片机嵌入式系统中经常使用的显示器件。一个“8”字型的显示模块用“a、b、c、d、e、f、g、p” 8 个发光二极管组合而成，如图 6-12a 所示。每个发光二极管称为一字段。LED 数码显示器有共阳极和共阴极两种结构形式，其内部电原理图为图 6-12b 和图 6-12c，在硬件电路设计和软件编写时不要混淆。

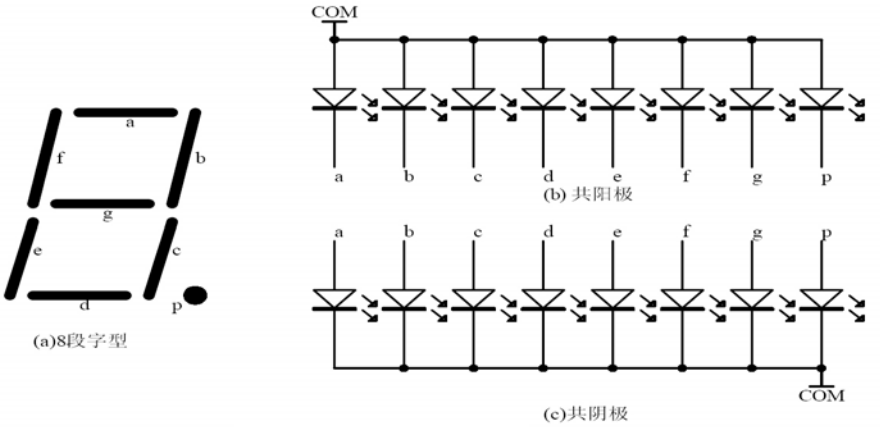

图 6-12 LED 数码显示器

LED 数码管显示的基本控制原理同上面所述发光二极管的控制，但在具体使用时有许多不同的设计和应用电路，软件的设计也各不相同，有许多技巧和变化。

6.3.1 单个LED数码管控制

我们以共阴极的数码管为例，先介绍如何控制一个 8 段数码管显示“0”－“F”16 个十六进制的数字。

例 6.4 单个 LED 数码管字符显示控制

1) 硬件电路设计：

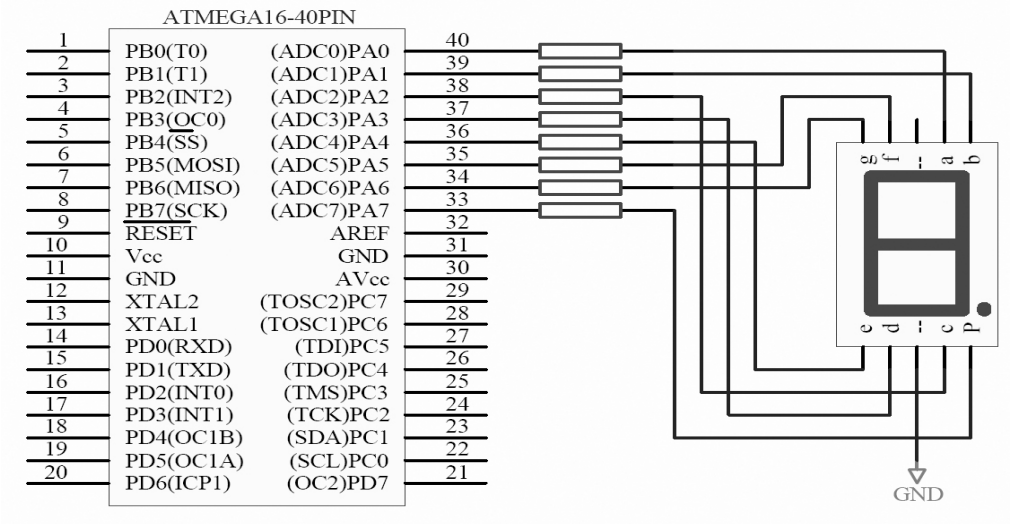

图 6-13 1 位共阴极 LED 数码显示器控制电路

很明显，用 AVR 的一个 I/O 口控制共阴极数码管的 8 个段位，分别置“1”或“0”，让某些段的 LED 发光，其它的熄灭，就可以显示不同的字符和图符号，硬件电路如图 6-13。

2) 软件设计：

为了获得“0”－“F”16 个不同的字型符号，数码管各段所加的电平不同，因此 I/O 口输出的编码也不同。因此首先要建立一个字型与字段 7 段码的编码表，见表 6.3。

表 6.3 8 段 LED 数码管字型字段编码表

显 示 字 型	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	段 码 共阴极	段 码 共阳极
	h	g	f	E	d	c	b	a		
0	0	0	1	1	1	1	1	1	3FH	C0H
1	0	0	0	0	0	1	1	0	06H	F9H
2	0	1	0	1	1	0	1	1	5BH	A4H
3	0	1	0	0	1	1	1	1	4FH	B0H
4	0	1	1	0	0	1	1	0	66H	99H
5	0	1	1	0	1	1	0	1	6DH	92H
6	0	1	1	1	1	1	0	1	7DH	82H
7	0	0	0	0	0	1	1	1	07H	F8H
8	0	1	1	1	1	1	1	1	7FH	80H
9	0	1	1	0	1	1	1	1	6FH	90H
A	0	1	1	1	0	1	1	1	77H	88H
b	0	1	1	1	1	1	0	0	7CH	83H
C	0	0	1	1	1	0	0	1	39H	C6H
d	0	1	0	1	1	1	1	0	5EH	A1H
E	0	1	1	1	1	0	0	1	79H	86H
F	0	1	1	1	0	0	0	1	71H	8EH

注：B、D 字型为小写 b、d，以同数字 8、0 字型区别

有了字型段码对照表，就可以用软件的方式进行 8 段码的译码。如要显示字型“1”，PA 口输出值为 0x06；显示字型“A”，PA 口输出值为 0x77。

在单片机嵌入式系统软件设计中，经常要考虑二进制、十六进制、十进制、BCD 码、压缩 BCD 码、八段码、ASCII 码之间的相互转换问题。人们计数习惯采用十进制，而单片机的计算、存储则为二进制形式最方便。此外传送字符用 ASCII 码，LED 数码显示要转化成相应的 7 段码等等。因此对与各种不同数制的使用和相互转换在软件设计中尤其重要，设计使用得当，可以简化程序设计和优化程序代码。

```
/*****************************************
file name           : demo_6_4.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/
```

```
#include <mega16.h>
#include <delay.h>

flash char led_7[16]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,
                      0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71};
bit point_on = 0;
void main(void)
{
  char i = 0;

  PORTA=0xFF;
  DDRA=0xFF;

  while (1)
   {
      for (i=0;i<=15;i++)
      {
           PORTA = led_7[i];
           if (point_on) PORTA |= 0x80;
           delay_ms(1000);
      }
      point_on = ~point_on;
   };
}
```

在本程序中，数组 led_7[]有 16 个元素，为显示字型“0”－“F”的段码。由于硬件确定后段码值就固定了，不会改变，因此把它定义在 Flash 中，可以节省 RAM 存储器。LED 数码管的小数点则要根据实际情况使用。

3） 思考与实践

- ✓ 如何显示其它特殊符号，如“P”、“q”、“L”、“H”等？
- ✓ 修改程序，控制一个 8 段数码管循环显示“0”－“F”16 个十六进制的数字，每个字符显示 1 秒钟，并且在显示的 1 秒内，数码管的小数点（P 段）要亮 0.5s，灭 0.5s。

6.3.2 多位LED数码管的显示

一个 LED 数码管只能显示一位数字，一般在系统经常要使用多个 LED 数码管，如要显示时间、温度、转速等等。在上面一位数码管控制显示的简单例子中，我们看到，一个数码管要使用 AVR 的 8 个 I/O 口线输出段码（公共端接 GND）。当使用多个数码管时，显然采用这样的控制方式有些问题，因为 AVR 是不能提供太多的 I/O 控制引脚的。比如系统要使用 4 个数码管，按上面例子中的控制方式，ATmega16 的全部 I/O 口将占用，这样其它的外围设备和电路就无法连接了。因此多个数码管的显示驱动系统的实现，有多种不同的方式可以采用，而且在硬件和软件的设计上也是不同的。

多位 LED 数码管显示电路按驱动方式可分为静态显示和动态显示两种方法。

采用静态显示方式时，除了在改变显示数据的时间外，所有的数码管都处于通电发光状态，每个数码管通电占空比为 100%（上例中的显示方式既为静态显示）。静态显示的优点有：显示稳定，亮度高，程序设计相对简单，MCU 负担小。缺点是：占用硬件资源多（如 I/O 口、

驱动锁存电路等），耗电量大。

而所谓动态显示方式，就是一位一位地轮流点亮各个数码管（动态扫描方式）。对于每一位数码管来说，每隔一定时间点亮一次，所以当扫描的时间间隔足够小时，观察者就不会感到数码管的闪烁，看到的现象是所有的数码管一起发光（同看电影的道理一样）。在动态扫描显示方式中，数码管的亮度同 LED 点亮导通时的电流大小，每一位点亮的时间和扫描间隔时间三个因素有关。动态显示的优点有：占用硬件资源少（如 I/O 口、驱动锁存电路等），耗电量小。缺点是：显示稳定性不易控制，程序设计相对复杂，MCU 负担重。

为了减轻 MCU 的负担和编程的复杂性，同时简化外围电路，还可以使用专用的数码管控制器件（见第 12 章）。

1．使用串行传送数据的静态显示接口

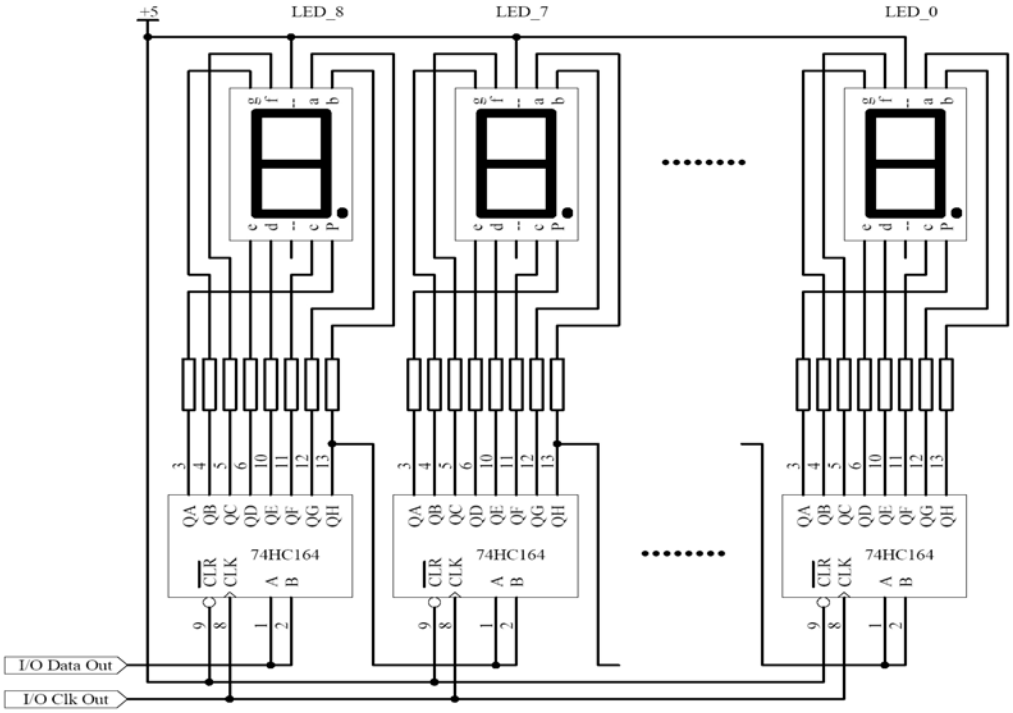

图 6-14 串行数据传送 8 位数码管静态显示接口电路

图 6-14 是一个采用串行传送数据的 8 位数码管静态显示接口。设计中将 8 片八位串行输入/并行输出移位寄存器 74HC164 串接，数码管为共阳极型。MCU 将 8 个要显示字符的段码字准备好，通过 Data Out 引脚，在 Clk Out 引脚产生的 cp 移位脉冲的作用下，一位一位地移入 74HC164 的 QA－QH 端（串行输入）。QA－QH 的输出（并行输出）直接作为数码管的段位控制。由于左边 74HC164 芯片的 QH 引脚（最低位）和右边 74HC164 芯片的数据串入引脚连接，经过 Clk Out 时钟线 64 个 cp 脉冲后，要显示的 8 个字符将会在 8 个数码管上显示，最先发送的显示字符段码将显示在最右边。

在这个电路设计中，硬件上使用了 8 片八位串行输入/并行输出移位寄存器 74HC164 串接，占用 AVR 的 2 个 I/O 口。软件的实现比较简单，MCU 只需要把新的显示内容通过两个 I/O 口线，一次串行输出即可。如果显示内容没有变化时，MCU 是不需要对显示部分进行任何操作的。

2．数码管显示器动态显示设计一

采用数码管动态扫描显示方式，可以节省硬件电路，但软件设计相对比较复杂。下面给出一个采用数码管动态扫描显示方式的设计，使用 6 个数码管组成时钟，两个一组，分别显示时、分、秒。

例 6.5 六位 LED 数码管动态扫描控制显示设计（一）

1）硬件设计电路：

图 6-15 给出硬件接口电路图。图中仅采用了 6 个共阴极的 LED 数码管。所有数码管段位 a 的引脚并接，由 PA0 控制；段 b 并接，由 PA1 控制；以此类推。既仍用 ATmega16 的 PA 口做为段码输出。ATmega16 的 PC0－PC5 分别与 LED0－LED5 的公共端 COM 引脚连接,既 PC 口的低 6 位作为位扫描控制口。

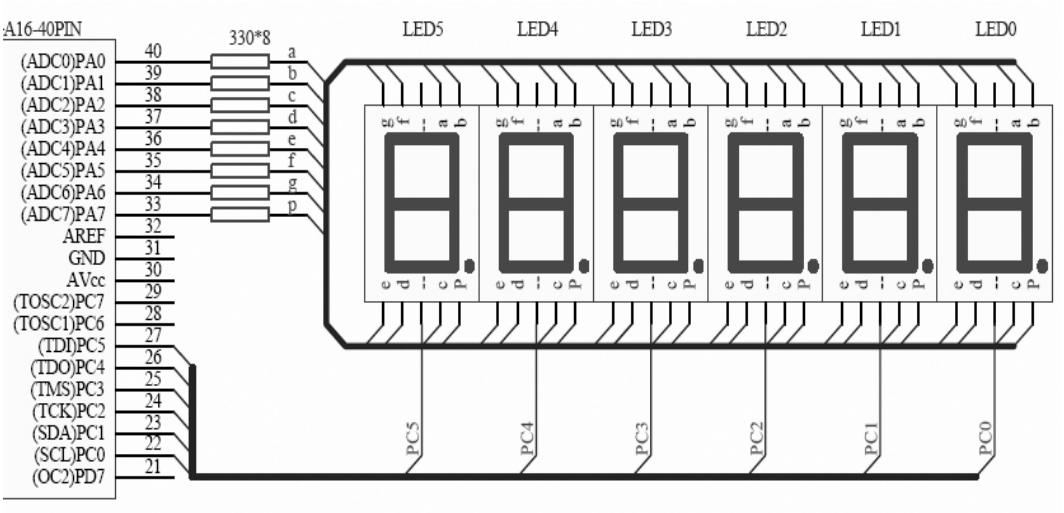

图 6-15 6 位数码管动态扫描显示接口电路

与静态方式的数码管驱动电路相比较，图 6-15 的电路图中没有使用外围器件，但占用了 14 个 I/O 口线（8 位时需要 16 个 I/O 口）。

2）软件设计

根据硬件电路，我们可以看出，在任何一个时刻，PC0-PC5 中只能有一个 I/O 口输出低电平，既只有一位数码管亮。而且，MCU 必须循环轮流控制 PC0-PC5 中的一位使其输出“0”，同时 PA 口要输出该位相应的段码值。即使显示的内容没有变化，MCU 也要进行不停的循环扫描处理。

软件的设计应保证从在外表看数码管显示的效果要连续（即在人眼里各个数码管全部亮），亮度均匀，同时没有拖尾现象。

为了保证各个数码管的显示的效果不产生闪烁情况，表象上全部点亮的话，则首先必须在 1 秒中内循环扫描 6 个数码管的次数应大于 25 次，这里是利用了人眼的影像滞留效应。本例中我们选择 40 次,既每隔 1000/40=25ms 将 6 个数码管循环扫描一遍。第二要考虑的是，在 25ms 时间间隔中，要逐一轮流点亮 6 个数码管，那么每个数码管点亮的持续时间要相同，这样亮度才能均匀。第三个要考虑的要点为每个数码管点亮的持续时间，这个时间长一些的话，数码管的亮度高一些，反之则暗一些。

通常，每个数码管点亮的持续时间为 1-2ms。我们将每个数码管的点亮持续时间定为 2ms，那么 6 个数码管扫描一遍的时间为 12ms，因此 MCU 还有 13ms 的时间处理其它事件，为了简单起见，本例中还是使用了 delay_ms()软件延时函数进行定时，在以后的章节里，将介绍使用 AVR 的定时器产生更精确的秒计时脉冲。

```
/*********************************************
file name          : demo_6_5.c
Chip type           : ATmega16
Program type        : Application
```

```
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

#include <mega16.h>
#include <delay.h>

flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};
char time[3];                   // 时、分、秒计数
char dis_buff[6];               // 显示缓冲区，存放要显示的 6 个字符的段码值
char time_counter;              // 1 秒计数器
bit point_on;                   // 秒显示标志

void display(void)              // 扫描显示函数，执行时间 12ms
{
    char i;
    for(i=0;i<=5;i++)
    {
        PORTA = led_7[dis_buff[i]];
        if (point_on && ( i==2 || i==4 )) PORTA |= 0x80;    // (1)
        PORTC = position[i];
        delay_ms(2);                                        // (2)
        PORTC = 0xff;                                       // (3)
    }
}

void time_to_disbuffer(void)          // 时间值送显示缓冲区函数
{
    char i,j=0;
    for (i=0;i<=2;i++)
    {
        dis_buff[j++] = time[i] % 10;
        dis_buff[j++] = time[i] / 10;
    }
}

void main(void)
{
    PORTA=0x00;        // PORTA 初始化
    DDRA=0xFF;
    PORTC=0x3F;        // PORTC 初始化
```

```
    DDRC=0x3F;

    time[2] = 23; time[1] = 58; time[0] = 55; // 时间初值 23:58:55
    time_to_disbuffer();

    while (1)
    {
        display();                          // 显示扫描，执行时间 12ms
        if (++time_counter >= 40)
        {
            time_counter = 0;               // (4)
            point_on = ~point_on;           // (5)
            if (++time[0] >= 60)
            {
                time[0] = 0;
                if (++time[1] >= 60)
                {
                    time[1] = 0;
                    if (++time[2] >= 24) time[2] = 0;
                }
            }
            time_to_disbuffer();
        }
        delay_ms(13);                   // 延时 13ms，可进行其它处理 (6)
    };
}
```

3）思考与实践

彻底、全面、读懂、理解和体会该段程序，对有（n）注释标记的语句和程序段进行分析，你是否能回答以下问题：

- ✓ 时、分、秒的计算采用何种数制？到数码管的时间显示之间经过了几种数制的转换？为什么要转换（不转换行吗）？怎样转换的？
- ✓ Display()函数是如何工作的？每秒钟执行几次？
- ✓ 说明 time_to_buffer()的功能，每秒执行几次？
- ✓ 说出和深入体会程序中的变量 time_counter、point_on 的作用。
- ✓ 将程序中有（3）注释标记的语句去掉，会产生什么现象，为什么？说明该语句的作用。
- ✓ 将程序中有（4）注释标记的语句去掉，会产生什么现象？
- ✓ 如何调整程序，使数码管的显示亮度有变化？
- ✓ 程序中使用了显示缓冲区，占用了 6 个字节。如果不使用显示缓冲区能否实现时间的显示？而使用显示缓冲区有何优点？
- ✓ 该程序中采用软件延时的方法，其主要的缺点有那些？

3. 数码管显示器动态显示设计二

在数码管显示器动态显示设计一中，尽管没有使用更多的外围器件，但是一共占用了

AVR 的 14 个 I/O 口。由于 AVR 的 I/O 口功能比较多，在实际应用中往往需要留出更多的 I/O 口应用于其它的控制和输入，因此经常采用外部增加少量的器件，以减少数码管显示驱动对 I/O 的占用。

例 6.6 六位 LED 数码管动态扫描控制显示设计（二）

1） 硬件设计

如果在硬件设计上多使用一片 74HC164（八位串行输入/并行输出的移位寄存器），就只要使用 PA 口的 8 根段输出线中的 2 根作为段码输出控制，其它 6 个 I/O 口就可以节省为它用了，电路图见图 6-16。

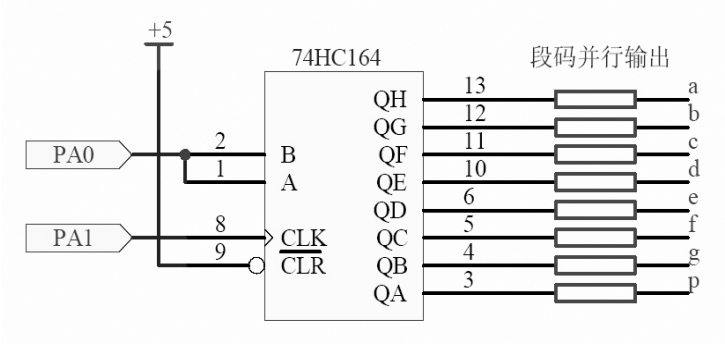

图 6-16 采用 74HC164 串入并出输出段码的接口电路

在图 6-16 中，仅给出的数码管段控制部分的接口电路。位控制仍然同图 6-15，使用 PC 口的 6 个 I/O。这个设计中使用了 AVR 的 PA0 和 PA1 两个 I/O 口串出段码，比图 6-15 中少使用 6 个 I/O 口。

2） 软件设计

根据硬件电路可以知道，数码管显示器的工作方式还是为动态显示的方法。因此，软件中应根据 74HC164 的逻辑真值表（表 6.4），增加一个使用 PA0（data）、PA1（clk）串行输出一个字节的功能函数。

表 6.4 74HC164 逻辑真值表

INPUTS(输入)				OUTPUTS(输出)			
CLR	CLK	A	B	QA	QB	……	QH
L	X	X	X	L	L	……	L
H	↓	X	X	No change			
H	↑	L	X	L	QAn	……	QGn
H	↑	X	L	L	QAn	……	QGn
H	↑	H	H	H	QAn	……	QGn

```
/*********************************************
file name           : demo_6_6.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/
```

```
#include <mega16.h>
#include <delay.h>

#define HC164_data PORTA.0
#define HC164_clk  PORTA.1

flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};
char time[3];                    // 时、分、秒计数
char dis_buff[6];                // 显示缓冲区，存放要显示的 6 个字符的段码值
char time_counter;
bit point_on;

void HC164_send_byte(char byte)
{
     char i;
     for (i=0;i<=7;i++)
     {
          HC164_data = byte & 1<< i;
          HC164_clk = 1;
          HC164_clk = 0;
     }
}

void display(void)
{
     char temp,i;
     for(i=0;i<=5;i++)
     {
          temp = led_7[dis_buff[i]];
          if (point_on && (i==2 || i==4))
               HC164_send_byte(temp | 0x80);
          else
               HC164_send_byte(temp);
          PORTC = position[i];
          delay_ms(2);
          PORTC = 0xff;
     }
}
。。。。。。。。。
```

程序的其它部分同 demo_6_5.c，只是对 display()函数中的段码输出语句进行了修改，变成调用 HC164_send_byte()函数输出段码。在 HC164_send_byte()函数中，程序控制 PA0、PA1，模拟出串行输出数据的时序，将一个字节的数据由低到高串行输出到 74HC164 中。

3） 思考与实践

- ✓ 如果要将一个字节的数据由高到低串行输出到 74HC164 中，HC164_send_byte()函数应该如何改动？硬件电路要如何调整？
- ✓ 在 HC164_send_byte()函数中的 FOR 循环中有 3 句语句，说明“HC164_data = byte & 1<<i;”的作用。此外这 3 句语句的执行顺序可以改动吗？
- ✓ 如果再增加一片芯片的话，还可以使用更少的 I/O 口实现数码管动态扫描显示的接口。请设计硬件电路和相应的软件显示控制程序，能够实现本例的功能（提示：可考虑 74HC164 或 74HC138）。

6.3.3 点阵LED显示控制

点阵 LED 在许多产品中也是经常使用的一种外围设备，如电梯中的运行指示，公交汽车里的站名广告显示，以及大型的电子广告牌等。这种 LED 的优点是可以通过点阵的形式显示汉字、图形等。实际上，PC 的显示屏、手机显示屏等，在上面显示汉字、图形的原理都是点阵显示的方法。

例 6.7 8*8 点阵 LED 显示控制设计

1） 硬件设计

8*8 点阵 LED 一般是一个方型的器件，由 8 行 * 8 列共 64 个 LED 发光二极管组成。图 6-17 是它的内部电原理图（4*4 点阵）。

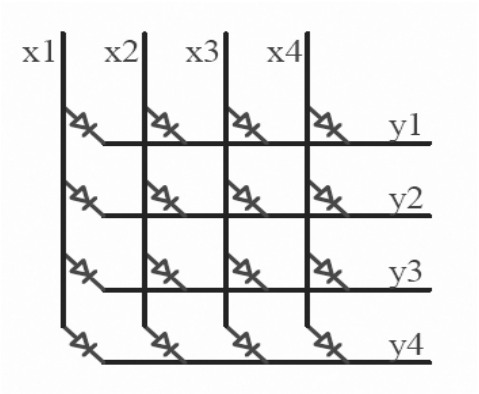

图 6-17 4*4 点阵 LED 原理图

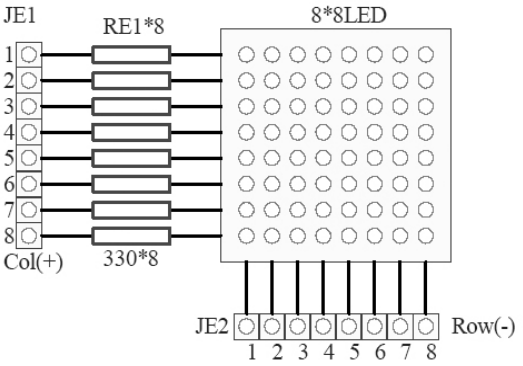

图 6-18 8*8 点阵 LED 模块

图 6-18 为 8*8 点阵 LED 的模块，我们使用 AVR 的 PA 口连接 JE1（Col+），PC 口连接 JE2（Row-），当 PA 输出一个字节数据，PC 口 8 位只有输出为“0”时，模块的其中一行（列）8 个 LED 就会根据 PA 的输出值点亮（熄灭）。

2） 软件设计

表 6.5 上箭头“↑”的码表

		Col+								PA 口输出值
		1 (PA7)	2 (PA6)	3 (PA5)	4 (PA4)	5 (PA3)	6 (PA2)	7 (PA1)	8 (PA0)	
Row(-)	1(PC0)				●					0x10
	2(PC1)			●	●	●				0x38
	3(PC2)		●	●	●	●	●			0x7C
	4(PC3)	●	●	●	●	●	●	●		0xFE
	5(PC4)			●	●	●				0x38
	6(PC5)			●	●	●				0x38
	7(PC6)			●	●	●				0x38
	8(PC7)			●	●	●				0x38

通过电路的分析，可以看出，8*8 点阵 LED 的显示控制方式与 LED 数码管的显示方式类似，也是使用动态扫描的工作方式。下面我们设计实现一个简单的，用于电梯中指示运行的向上运动的箭头“↑”。

首先我们要建立一个向上箭头“↑”的码表，见表 6.5。然后需要一个动态扫描显示的函数 display()。

```
/*****************************************
file name           : demo_6_7.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

#include <mega16.h>
#include <delay.h>

flash char char_7[8]={0x10,0x38,0x7C,0xFE,0x38,0x38,0x38,0x38};

void display(char row)
{
   char i;
   for (i=0;i<=7;i++)
   {
        if (row <= 7)
             PORTA = char_7[row];
        else
             PORTA = 0;
        PORTC = ~(1<<i);
        delay_ms(2);
        PORTC = 0xFF;
        if (++row >= 12 ) row = 0;
   }
}

void main(void)
{
   char time_counter,i = 0;
   PORTA=0x00;
   DDRA=0xFF;
   PORTC=0xFF;
   DDRC=0xFF;
```

```
    while (1)
    {
        display(i);
        delay_ms(9);
        if (++time_counter >= 4)
        {
            time_counter = 0;
            if(++i >= 12) i=0;
        }
    };
}
```

3） 思考与实践

- ✓ 本例程与数码管显示程序有非常多的类似地方，请读者自己分析程序的功能。
- ✓ 请给出程序中的时间分配情况，如何调整箭头移动的速度的快慢，而又不影响正常的显示？
- ✓ 如果将 display()函数中的语句“PORTC = ~(1<<i);”改为“PORTC = ~(1<<(7-i));”，显示有何变化？
- ✓ 设计一个水平移动的 8*8 点阵广告，能够显示“今天 YOU ARE OK？”

6.4 LCD液晶显示器的应用

液晶显示器（LCD）由于体积小、重量轻、耗电小等优点已成为各种嵌入式系统所采用的理想显示器。近年来液晶显示器技术的发展迅猛，大面积的液晶显示器已开始取代 CRT 显示器，在使用电池供电的嵌入式电子产品中，如手机、PDA，以及家电产品，仪器仪表产品等，液晶显示器是首选的显示器。

6.4.1 LCD的特点与分类

1．LCD 的特点

- ✓ 低电压微功耗。工作电压 3-5V，每平方厘米液晶显示屏的耗电量在 uA 级。
- ✓ 平板结构。易大量生产，物理体积小，占用空间少。
- ✓ 寿命长。
- ✓ 光线柔和。液晶显示器是被动发光器件，90%以上是外部物体对光的反射。被动显示适合人的视觉习惯，不会引起疲劳。
- ✓ 无电磁辐射。液晶显示器不会产生电磁辐射，是绿色器件。

2．LCD 显示器的分类

从液晶显示器的使用和显示内容来分，LCD 可分为字段式（笔划式），点阵字符式，点阵图形式三种。

字段式液晶显示器同 LED 数码显示器有些相同点，它是以长条笔划状或一些特殊固定图形与汉字显示象素组成的液晶显示器件，简称段型显示器。段型显示器以七段显示器为常见，特殊图形与字符类的段型液晶显示器一般要到生产厂家定做。段型液晶显示器在数字仪表、计数器，家电产品中应用较多。

点阵字符式液晶显示器一般是一个功能模块，它由小面积的液晶显示屏和驱动电路组合

而成。模块中内置有 192 种字符、数字、字母、标点符号等可显示的字型点阵图形库，并提供可控制的并行或串行接口以及通信协议。市场上常见的有 1 行、2 行、4 行，每行可显示 8、12、16、24、32 个 5x7 点阵字符的通用液晶显示器。

点阵图形式液晶显示器一般显示面积大于点阵式液晶显示器，点阵从 80x32 到 1024x768 不等。点阵图形式液晶显示器的显示灵活性好，自由度大，可以显示各种图形、字符和汉字等。但点阵图形式液晶显示器的控制最复杂，硬件连接线多，占用 MCU 的资源也多。为了适应越来越多的液晶显示器应用，一些高性能的单片机已经将液晶显示器驱动功能集成在片内。目前国内一些厂商将驱动电路、汉字库和点阵液晶显示器屏做成一个组件模块，模块带有与 MCU 通信的并行或串行接口，使用时，只要 MCU 通过通信口下发相应的控制指令就能显示各种信息，方便了使用。

6.4.2 通用点阵字符LCD显示器应用

通用点阵字符液晶显示器是专用于显示数字、字母、图形符号和一些自定义符号的显示器。这类显示器把 LCD 控制器、点阵驱动器、字符存储器全做在一块 PCB 板上，构成便于应用的显示器模块。这类点阵字符液晶显示器模块在国际上已经规范化，一般都采用日立公司的 HD44780 极其兼容电路，如 SED1278、KS0066 等，作为 LCD 的控制器。

HD44780 具有简单而功能较强的指令集，可实现字符移动、闪烁等功能。与 MCU 的数据传输可采用 8 位并行或 4 位并行传输两种方式。可用于驱动 40 * 4，16 * 1，16 * 2，16 * 4，20 * 2，20 * 4，等多种点阵字符液晶显示器。HD44780 对外有 14 根引脚，与 MCU 的接口信号及定义见表 6.6。

表 6.6 HD44780 引脚功能定义表

引脚号	符号	I/O	功　　能
1	Vss		电源负端,接地（或接－5V）
2	Vdd		电源正端,接+5V
3	Vo		LCD 亮度调整电压 0-5V
4	RS	I	寄存器选择:RS=0,选指令寄存器;RS=1,选数据寄存器
5	R/W	I	读/写选择:R/W=0,写数据至 LCD;R/W=1,从 LCD 读数据
6	E	I	输入允许:R/W=0,E↓下降沿打入;R/W=1,E=1 有效
7-10	DB0-DB3	I/O	数据总线：使用 4 位并行传输时仅用(DB4—DB7)4 位
11-14	DB4-DB7	I/O	使用 8 位并行传输时使用(DB0—DB7)8 位
15-16			LCD 背光电源的正极和负极(有些模块没有背光功能)

从零开始编写 HD44780 的控制程序需要了解 HD44780 的内部结构、操作时序、指令集、内部 REM 与字符图形的对应关系和字符代码表等等。编写程序时需要先编写底层的驱动程序，再编写上层的应用接口程序，再加上程序调试时间，通常要花费 3-5 天时间。对于一般的初学者，花费 2-3 个星期也未必能完成软件的设计。但由于这种点阵字符液晶显示器模块在国际上已经规范化，因此在 CVAVR 中扩展提供了一些基本的 LCD 应用接口函数，所以在 CVAVR 平台的支持下，用户使用这类 LCD 点阵字符显示器就比较方便。你只要写几条语句，调用 CVAVR 的 LCD 提供的函数，化几分钟的时间，就能把要显示的信息在 LCD 上显示出来。

在 CVAVR 中，与 LCD 字符显示器有关的功能函数有：

1）void lcd_init(unsigned char lcd_columns)

该函数对 LCD 进行初始化，并清除 LCD 的显示，将显示位置回到第 0 行的第 0 列的起始位置处。函数的参数应是 LCD 显示器的列数（一行能够显示的字符数）。使用 LCD 显示器时，必须先使用该函数对 LCD 显示器进行初始化。

2）void lcd_clear(void)

该函数清除 LCD 的显示，并将显示位置回到第 0 行的第 0 列的起始位置处。

3）void lcd_gotoxy(unsigned char x, unsigned char y)

该函数将显示位置定位于第 x 行的第 y 列的位置处。注意，LCD 的行列定位都是从“0”起始的。

4）void lcd_putchar(char c)

该函数将字符 c 在当前的显示位置上显示出来。

5）void lcd_puts(char *str)

该函数将在从当前的显示位置开始，显示定义在 SRAM 中的字符串（str 为 SRAM 中定义的字符串的指针）。

6）void lcd_putsf(char flash *str)

该函数将在从当前的显示位置开始，显示定义在 Flash 中的字符串（str 为 Flash 中定义的字符串的指针）。

除了上面 6 个 LCD 函数外，在 CVAVR 中还有一些扩展的用于字符 LCD 控制的函数，能提供更多的功能。请读者在具体应用时，仔细阅读 CVAVR 的 Help 和使用手册，掌握这些函数的使用，如学会如何自己设计定义和使用特殊的符号和图形（HD44780 支持用户自己定义最多 8 个 5*8 点阵的字符和图形）。

例 6.8 16*2 标准 LCD 字符显示器应用设计

1） 硬件设计

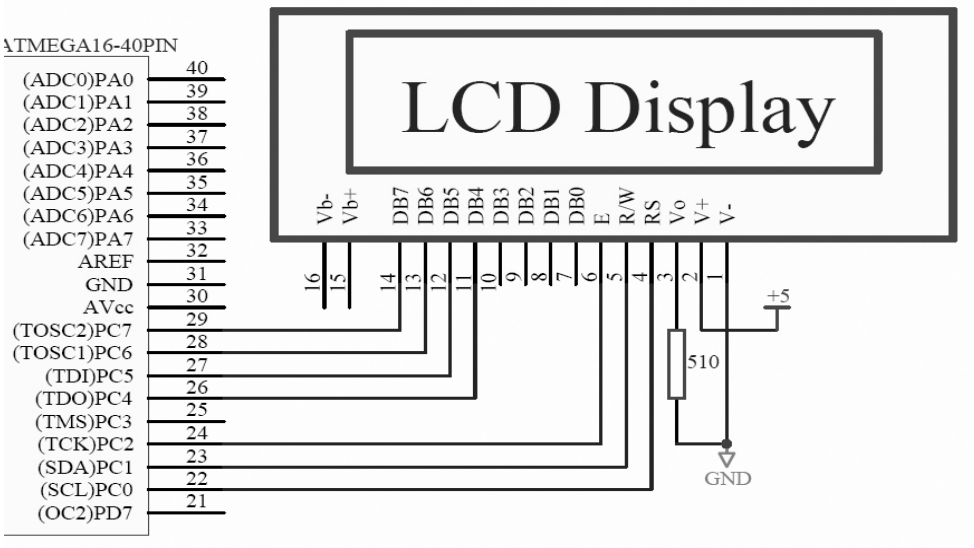

图 6-19 16*2 标准字符型 LCD 接口电路

图 6-19 所示为 16*2 标准字符型 LCD 的连接电路图。由于在 CVAVR 中，必须按照一定的规定连接 LCD 时，才能使用 CVAVR 内部提供的 LCD 函数，所以原理图是按照 CVAVR 的规定设计的。

如要使用 CVAVR 内部提供的 LCD 函数，硬件连接必须按以下要求实现。
1）与 LCD 的连接必须使用 AVR 的同一个 8 位的 I/O 端口，如 PC（或者 PA、PB、PD）。
2）LCD 采用 4 位并行传输方式（既仅用 DB4—DB7，4 位数据总线）。
3）具体连接定义为（以 PC 口为例）：
三根控制线　PC0----RS，PC1----R/W，PC2----E
四根数据线　PC4----DB4，PC5----DB5，PC6----DB6，PC7----DB7

2）软件设计

```
/*****************************************
File name           : demo_6_8.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/
#include <mega16.h>
#include <delay.h>

#asm
.equ __lcd_port=0x15 ; PORTC 数据寄存器地址
#endasm
/*  [LCD]
    1 GND- 9  GND
    2 +5V- 10 VCC
    3 VLC- LCD HEADER Vo
    4 RS - 1  PC0 (M16)
    5 RD - 2  PC1 (M16)
    6 EN - 3  PC2 (M16)
   11 D4 - 5  PC4 (M16)
   12 D5 - 6  PC5 (M16)
   13 D6 - 7  PC6 (M16)
   14 D7 - 8  PC7 (M16) */
#include <lcd.h>
flash char dis_str[]="Hello World! This is a LCD display demo.";
void main(void)
{
  char flash *str;
  str = dis_str;
  lcd_init(16);               // initialize the LCD for 2 lines & 16 columns
  while(1)
```

```
    {
        lcd_clear();                    // clere the LCD
        lcd_putsf("It's demo_6_8.c");   // display the message
        lcd_gotoxy(0,1);                // go on the second LCD line
        lcd_putsf(str);                 // display the message
        if (*str++ == 0) str = dis_str;
        delay_ms(500);
     }
  }
```

该简单的 LCD 显示的演示程序全部调用的是 CVAVR 中的 LCD 函数，程序运行后，在 LCD 的第一行固定显示字符“It's demo_6_8.c”，在第二行滚动显示“Hello World! This is a LCD display demo.”。

在程序的开始部分，嵌入了一句 AVR 汇编的伪指令：“.equ __lcd_port = 0x15”，这也是 CVAVR 规定要使用的，它通知 CVAVR 编译系统，在硬件上 AVR 与 LCD 连接的 I/O 地址为 0x15，这是 PC 口的数据寄存器 PORTC 的地址。此时，用户不必关心 PC 口的初始化问题，在调用 lcd_init(16)函数时，该函数会对 PC 口进行必要的初始化工作。因此，使用标准字符 LCD 显示器时，必须先使用该函数进行初始化工作。

在 CVAVR 中还有一些扩展的用于字符 LCD 控制的函数，能提供更多的功能，如允许用户自己设计定义和使用特殊的符号和图形（HD44780 支持用户自己定义最多 8 个 5*8 点阵的字符和图形）等。下面的演示程序，可以在 LCD 上显示用户定义的简单汉字“天天向上”。

```
/*****************************************
File name           : Demo_6_9.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/
#include <mega16.h>
// Alphanumeric LCD Module functions
#asm
   .equ __lcd_port=0x15
#endasm
#include <lcd.h>

typedef unsigned char byte;

/* table for the user defined character */
flash byte char0[8]={       // 天的字型
0b0011111,
0b0000100,
0b0000100,
0b0011111,
```

```
0b0000100,
0b0000100,
0b0001010,
0b0010001};
flash byte char1[8]={       // 向的字型
0b0000100,
0b0001000,
0b0011111,
0b0010001,
0b0011111,
0b0011011,
0b0011111,
0b0010001};
flash byte char2[8]={       // 上的字型
0b0000100,
0b0000100,
0b0000111,
0b0000100,
0b0000100,
0b0000100,
0b0000100,
0b0011111};

/* function used to define user characters */
void define_char(byte flash *pc,byte char_code)
{
     byte i,a;
     a=(char_code<<3) | 0x40;
     for (i=0; i<8; i++) lcd_write_byte(a++,*pc++);
}

void main(void)
{
     lcd_init(16);          // initialize the LCD for 2 lines & 16 columns

     define_char(char0,0);  // define user character 0
     define_char(char1,1);  // define user character 1
     define_char(char2,2);  // define user character 2
     lcd_clear();
     lcd_putsf("Demo_6_9.c");    // 第一行显示内容
     lcd_gotoxy(0,1);
     lcd_putsf("User define:");  // 第二行显示内容
     lcd_putchar(0);             // 接在后面显示“天天向上”
     lcd_putchar(0);
```

```
        lcd_putchar(1);
        lcd_putchar(2);
        while (1);
    }
```

有兴趣的读者,可以自己尝试和练习编写标准字符型LCD控制芯片HD44780的控制程序,关于 HD44780 的详细资料可以在本书附带的光盘中找到。

思考与练习

1. AVR 单片机 I/O 口三个寄存器的名称和作用是什么？当 I/O 口用于输入和输出时如何设置和应用这三个寄存器？
2. 给出一个 8 位数码管显示器静态显示的硬件和软件设计方案以及一个动态扫描显示的硬件和软件设计方案，并比较这两个方案的优缺点。
3. 全面、仔细、深入分析程序 demo_6_5.c，说明在动态扫描显示设计中，如何保证每个显示器的亮度一致，在系统应用中没有闪烁和熄灭现象。
4. 在动态扫描显示中，如果要调整显示的亮度，请给出三种硬件和软件的设计改动方法，并说明理由。
5. 认真分析和理解本章中所给出的所有示例，并进行实践，思考和回答所有的问题。

本章参考文献：

1.《74hc164.pdf》（英文，CDROM）
2.《74hc138.pdf》（英文，CDROM）
3.《HD44780 器件手册.pdf》（中文，CDROM）

第 7 章　中断系统与基本应用

中断是现代计算机必备的重要功能。尤其在嵌入式系统和单片机系统中，中断扮演了非常重要的角色。因此，全面深入的了解中断的概念，并能灵活掌握中断技术的应用，成为学习和真正掌握单片机应用非常重要的关键问题之一。

7.1　中断的基本概念

中断是指计算机（MCU）自动响应一个“中断请求”信号，暂时停止（中断）了当前程序的执行，转而执行为外部设备服务的程序（中断服务程序），并在执行完服务程序后自动返回原程序执行的过程。

单片机一般都具有良好的中断系统，它的优点有：

- ✓ 实现实时处理。利用中断技术，MCU 可以及时响应和处理来自内部功能模块或外部设备的中断请求，并为其服务，以满足实时处理和控制的要求。
- ✓ 实现分时操作，提高了 MCU 的效率。在嵌入式系统的应用中可以通过分时操作的方式启动多个功能部件和外设同时工作。当外设或内部功能部件向 MCU 发出中断申请时，MCU 才转去为它服务。这样，利用中断功能，MCU 就可以“同时”执行多个服务程序，提高了 MCU 的效率。
- ✓ 进行故障处理。对系统在运行过程中出现的难以预料的情况或故障，如掉电，可以通过中断系统及时向 MCU 请求中断，做紧急故障处理。
- ✓ 待机状态的唤醒。在单片机嵌入式系统的应用中，为了减少电源的功耗，当系统不处理任何事物，处于待机状态时，可以让单片机工作在休眠的低功耗方式。通常，恢复到正常工作方式往往也是利用中断信号来唤醒。

7.1.1 中断处理过程

在中断系统中，通常将 MCU 处在正常情况下运行的程序称为**主程序**，把产生申请中断信号的单元和事件称为**中断源**，由中断源向 MCU 所发出的申请中断信号称为**中断请求信号**，MCU 接受中断申请停止现行程序的运行而转向为中断服务称为**中断响应**，为中断服务的程序称为**中断服务程序或中断处理程序**。现行程序打断的地方称为**断点**，执行完中断处理程序后返回断点处继续执行主程序称为**中断返回**。这一整个的处理过程称为**中断处理过程（图 7-1）**。

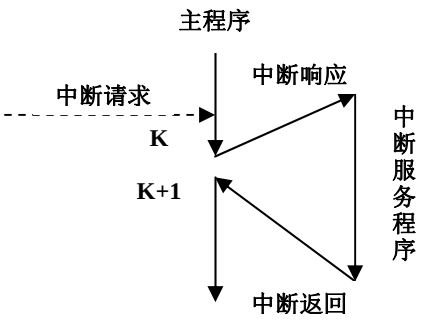

图 7-1 中断过程示意图

在整个中断处理过程中，由于 MCU 执行完中断处理程序后仍然要返回主程序，因此，在执行中断处理程序之前，要将主程序中断处的地址，即断点处（实际为程序计数器 PC 的当前值——即将执行的主程序的下一条指令地址，图 7-1 中的 k+1 点）保存起来，称为**保护断点**。又由于 MCU 在执行中断处理程序时，可能会使用和改变主程序使用过的寄存器、标志位，甚至内存单元，因此，在执行中断服务程序前，还要把有关的数据保护起来，称为**中断现场保护**。在 MCU 执行

完中断处理程序后，则要恢复原来的数据，并返回主程序的断点处继续执行，称为**恢复现场**和**恢复断点**。

在单片机中，断点的保护和恢复操作，是在系统响应中断和执行中断返回指令时由单片机的内部硬件自动实现的，简单的说，就是在响应中断时，MCU 的硬件系统会自动将断点地址压进系统的堆栈保存，而当执行中断返回指令时，硬件系统会自动又将压入堆栈的断点地址弹出到程序计数器 PC 中。

但对于中断现场的保护和恢复，则需要程序员在设计中断处理程序时编程实现。在使用中断时，要认真和仔细考虑中断现场的保护和恢复。

7.1.2 中断源、中断信号、中断向量

1. 中断源

中断源是指能够向 MCU 发出中断请求信号的部件和设备。在一个系统中，往往存在多个中断源。对于单片机讲，中断源一般可分为内部中断源和外部中断源。

在单片机内部集成的许多功能模块，如定时器、串行通讯口、模/数转换器等，它们在正常工作时往往无需 CPU 参与，而当处于某种状态或达到某个规定值需要程序控制时，会通过发出中断请求信号通知 CPU。这一类的中断源位于单片机内部，称作内部中断源。其典型例子有定时器溢出中断、ADC 完成中断等。如 8 位的定时器在正常计数过程中无需 CPU 的干预，一旦计数到达 0xff 产生溢出时便产生一个中断申请信号，通知 CPU 进行必要的处理。内部中断源在中断条件成立时，一般通过片内硬件会自动产生中断请求信号，无须用户介入，使用方便。内部中断是 CPU 管理片内资源的一种高效的途径。

系统中的外部设备也可以用作中断源，这时要求它们能够产生一个中断信号（通常是高（低）电平或者电平跳变的上升（下降）沿），送到单片机的外部中断请求引脚供 CPU 检测。这些中断源位于单片机外部，称为外部中断源。通常用作外部中断源的有输入输出设备、控制对象、以及故障源等。例如，打印机打印完一个字符时可以通过中断请求 CPU 为它送下一个打印字符；控制对象可以通过中断要求 CPU 及时采集参量或者对参数超标做出反应；掉电检测电路发现掉电时可以通过中断通知 CPU，以便在短时间内对数据进行保护。

2. 中断信号

中断信号是指内部或外部中断源产生的中断申请信号，这个中断信号往往是电信号的某种变化形式，通常有以下几种类型：

- ✓ 脉冲的上跳沿或下降沿（上升沿触发型或下降沿触发型）
- ✓ 高电平或低电平（电平触发型）
- ✓ 电平的变化（状态变化触发型）

对于单片机来讲，不同的中断源，产生什么类型的中断信号能够触发申请中断，取决于芯片内部的硬件结构，而且通常也可以通过用户的软件来设定。

单片机的硬件系统会自动对这些中断信号进行检测。一旦检测到规定的信号出现，将会把相应的中断标志位置“1”（在 I/O 空间的控制或状态寄存器中），通知 CPU 进行处理。

3. 中断向量

中断源发出的请求信号被 CPU 检测到之后，如果单片机的中断控制系统允许响应中断，CPU 会自动转移，执行一个固定的程序空间地址中的指令。这个固定的地址称作中断入口地址，也叫做中断向量。中断入口地址往往是由单片机内部硬件决定的。

通常，一个单片机会有若干个中断源，每个中断源都有着自己的中断向量。这些中断向量一般在程序存储空间中占用一个连续的地址空间段，称为中断向量区，如表 7.1 所示。由于一个中断向量通常仅占几个字节或一条指令的长度，所以在中断向量区一般不放置中断服

务程序的。中断服务程序一般放置在程序存储器的其它地方，而在中断向量处放置一条跳转到中断服务程序的指令。这样，CPU 响应中断后，首先自动转向执行中断向量中的转移指令，再跳转执行中断服务程序。

7.1.3 中断优先级和中断嵌套

中断优先级的概念是针对有多个中断源同时申请中断时，MCU 如何响应中断，以及响应哪个中断而提出的。

通常，一个单片机会有若干个中断源，MCU 可以接收若干个中断源发出的中断请求。但在同一时刻，MCU 只能响应这些中断请求中的其中一个。为了避免 MCU 同时响应多个中断请求带来的混乱，在单片机中为每一个中断源赋予一个特定的中断优先级。一旦有多个中断请求信号，MCU 先响应中断优先级高的中断请求，然后再逐次响应优先级次一级的中断。中断优先级也反映了各个中断源的重要程度，同时也是分析中断嵌套的基础。

对于中断优先级的确定，通常是由单片机的硬件结构规定的。一般的确定规则方式为两种：

- ✓ 某中断对应的中断向量地址越小，其中断优先级越高（硬件确定方式）。
- ✓ 通过软件对中断控制寄存器的设定，改变中断的优先级（用户软件可设置方式，注意：AVR 不支持）。

实际上，MCU 在两种情况下需要对中断的优先级进行判断：

第一种情况为同时有两（多）个中断源申请中断。在这种情况下，MCU 首先响应中断优先级最高的那个中断，而将其它的中断挂起。待优先级最高的中断服务程序执行完成返回后，再顺序响应优先级较低的中断。

第二种情况是当 MCU 正处于响应一个中断的过程中。如已经响应了某个中断，正在执行为其服务的中断程序时，此时又产生一个其它的中断申请，这种情况也称作中断嵌套。

对于中断嵌套的处理，不同的单片机处理的方式是不同的，应根据你所使用单片机的特点正确实现中断嵌套的处理。

按照通常的规则，当 MCU 正在响应一个中断 B 的过程中，又产生一个其它的中断 A 申请时，如果这个新产生中断 A 的优先级比正在响应的中断 B 优先级高的话，就应该暂停当前的中断 B 的处理，转入响应高优先级的中断 A，待高优先级中断 A 处理完成后，再返回原来的中断 B 的处理过程。如果新产生中断 A 的优先级比正在处理中断 B 的优先级低（或相同），则应在处理完当前的中断 B 后，再响应那个后产生的中断 A 申请（如果中断 A 条件还成立的话）。

一些单片机（如 8051 结构）的硬件能够自动实现中断嵌套的处理，既单片机内部的硬件电路能够识别中断的优先级，并根据优先级的高低，自动完成对高优先级中断的优先响应，实现中断的嵌套处理。

而另一类的单片机，如本书介绍的 AVR 单片机，其硬件系统不支持自动实现中断嵌套的处理。如果在系统设计中，必须使用中断嵌套处理，则需要由用户编写相应的程序，通过软件设置来实现中断嵌套的功能。

7.1.4 中断响应条件与中断控制

1） 中断的屏蔽

单片机拥有众多中断源，但在某一具体设计中通常并不需要使用所有的中断源，或者在

系统软件运行的某些关键阶段不允许中断打断现行程序的运行，这就需要一套软件可控制的中断屏蔽/允许系统。在单片机的 I/O 寄存器中，通常存在一些特殊的标志位用于控制开放或关闭（屏蔽）MCU 对中断响应处理，这些标志称为中断屏蔽标志位或中断允许控制位。用户程序可以改变这些标志位的设置，在需要的时候允许 MCU 响应中断，而在不需要的时候则将中断请求信号屏蔽（注意：不是取消），此时尽管产生了中断请求信号，MCU 也不会响应中断请求。

因而从对中断源的控制角度讲，中断源还可分成 3 类：

- ✓ 非屏蔽中断。非屏蔽中断是指 MCU 对中断源产生的中断请求信号是不能屏蔽的，也就是说一旦发生中断请求，MCU 肯定响应该中断。在单片机中，外部 RESET 引脚产生的复位信号，就是一个非屏蔽的中断。
- ✓ 可屏蔽中断。可屏蔽中断是指用户程序可以通过中断屏蔽控制标志对中断源产生的中断请求信号进行控制，既允许或禁止 MCU 对该中断的响应。在用户程序中，可以预先执行一条允许中断的指令，这样一旦发生中断请求，MCU 就能够响应中断。反之，用户程序也可以预先执行一条中断禁止（屏蔽）指令，使 MCU 不响应中断请求。因此，可屏蔽中断的中断请求能否可以被 MCU 响应，最终是由用户程序来控制的。在单片机中，大多数的中断都是可屏蔽的中断。
- ✓ 软件中断。软件中断通常是指 CPU 具有相应的软件中断指令，当 MCU 执行这条指令时就能进入软件中断服务，以完成特定的功能（通常用于调试）。但一般的单片机都不具备软件中断的指令，因此不能直接通过软件中断的指令实现软件中断的功能。因此，在单片机系统中，如果必须要使用软件中断的功能，一般要通过间接的方式实现软件中断的功能。

2）中断控制与中断响应条件

综合前面的介绍，我们可以知道，在单片机中，对应每一个中断源都有一个相应的**中断标志位**，该中断标志位将占据中断控制寄存器中的一位。当单片机检测到某一中断源产生符合条件的中断信号时，其硬件会自动将该中断源对应的中断标志位置“1”，这就意味着有中断信号产生了，向 MCU 申请中断。

但中断标志位的置“1”，并不代表 MCU 一定响应该中断。为了合理控制中断响应，在单片机内部还有相关的用于中断控制的**中断允许标志位**。最重要的一个中断允许标志位是**全局中断允许标志位**。当该标志位为“0”，表示禁止 MCU 响应所有的可屏蔽中断的响应。此时不管有否中断产生，MCU 不会响应任何的中断请求。只有全局中断允许标志位为“1”，才为 MCU 响应中断请求打开第一道闸门。

MCU 响应中断请求的第二道闸门是每个中断源所具有的各自**独立的中断允许标志位**。当某个中断允许标志位为“0”时，表示 MCU 不响应该中断的中断申请。

因此，MCU 响应一个可屏蔽中断源（假定为 A 中断）的中断请求的条件是：
响应 A 中断 = 全局中断允许标志 AND 中断 A 允许标志 AND 中断 A 标志

从上面的中断响应条件看出，只有当全局中断允许标志位为“1”（由用户软件设置），中断 A 允许标志位为“1”（由用户软件设置），中断 A 标志位为“1”（符合中断条件时由硬件自动设置或由用户软件设置）时，MCU 才会响应中断 A 的请求信号（如果有多个中断请求信号同时存在的情况下，还要根据中断 A 的优先级来确定）。

用户程序对可屏蔽中断的控制，一般是通过设置相应的中断控制寄存器来实现的。除了设置中断的响应条件，用户程序还需要通过中断控制器来设置中断的其他特性，如：中断触发信号的类型、中断的优先级、中断信号产生的条件等等。

以上我们介绍了中断的基本概念，可以看出中断的控制与使用相对比较复杂。但是正确和熟练掌握中断的应用，是单片机嵌入式系统设计的重要和基本技能之一。单片机的许多功能和特点，以及变化无穷的应用，往往需要中断的巧妙的配合。因此，要正确使用中断，必须全面了解所使用单片机的中断特性，中断服务程序的编写技能，以及中断使用的技巧和设计。因此读者还需要在以后的学习和应用中进一步的深入理解，逐步全面的掌握中断应用的基本技巧。

7.2 ATmega16 的中断系统

与一般 8 位单片机相比，AVR 单片机的中断系统具有中断源品种多、门类全的特点，便于设计实时、多功能、高效率的嵌入式应用系统。但同时由于其功能更为强大，因此相比一般 8 位单片机 AVR 的中断使用和控制也相对复杂些。本节以 ATmega16 为主，讨论 AVR 单片机的中断系统的组成和基本的应用方式。对于各个中断源的具体配置和使用将在相关章节中介绍。

7.2.1 ATmega16 的中断源和中断向量

AVR 一般拥有数十个中断源，每个中断源都有独立的中断向量。缺省情况下，AVR 的程序存储区的最低端，即从 Flash 地址的 0x0000 开始用于放置中断向量，称作中断向量区。

各种型号的 AVR 中断向量区的大小是不同的，由下式决定：

中断向量区大小 = 中断源个数 * 每个中断向量占据字数

对于 Flash 比较小的 AVR 处理器，每个中断向量占据一个字的空间，用于放置一条相对转移指令 rjmp（跳转范围-2k~+2k），而 Flash 较大的 AVR，每个中断向量占据两个字空间，用于放置一条绝对转移指令 jmp，用于跳转到相应中断的中断服务程序的起始地址。

原则上讲，在不使用中断的时候，中断向量区与程序存储区的其它部分没有什么区别，可以用于放置普通的程序。但在正式的系统应用中，为了提高系统的抗干扰能力，通常应该在中断向量的位置上放置一条中断返回指令 RETI（对于中断向量占据两个字空间的处理器，应连续放置两条 RETI）。对于使用了一部分中断的情况，则应在未使用的中断向量处放置这样的指令。在用汇编语言进行开发时应该注意这一点。

ATmega16 共有 21 个中断源，由于 ATmega16 片内的 Flash 为 8K 字，因此每个中断向量占据了两个字（4 个字节），缺省状态下 ATmega16 的中断向量表如下：

表 7.1 ATmega16 的中断向量区

向量号	Flash 空间地址（中断向量）	中断源	中断定义说明
1	$000	RESET	外部引脚电平引发的复位、 上电复位、掉电检测复位、 看门狗复位、JTAG AVR 复位
2	$002	INT0	外部中断请求 0
3	$004	INT1	外部中断请求 1
4	$006	TIMER2 COMP	定时计数器 2 比较匹配
5	$008	TIMER2 OVF	定时计数器 2 溢出
6	$00A	TIMER1 CAPT	定时计数器 1 事件捕捉
7	$00C	TIMER1 COMPA	定时计数器 1 比较匹配 A

8	$00E	TIMER1 COMPB	定时计数器 1 比较匹配 B
9	$010	TIMER1 OVF	定时计数器 1 溢出
10	$012	TIMER0 OVF	定时计数器 0 溢出
11	$014	SPI STC	SPI 串行传输结束
12	$016	USART RXC	USART，Rx 结束
13	$018	USART UDRE	USART, 数据寄存器空
14	$01A	USART TXC	USART，Tx 结束
15	$01C	ADC	AD 转换结束
16	$01E	EE_RDY	EEPROM 就绪
17	$020	ANA_COMP	模拟比较器
18	$022	TWI	两线串行接口
19	$024	INT2	外部中断请求 2
20	$026	TIMER0 COMP	定时计数器 0 比较匹配
21	$028	SPM_RDY	保存程序存储器内容就绪

在这 21 个中断中，包含 1 个非屏蔽中断（RESET）3 个外部中断（INT0、INT1、INT2）和 17 个内部中断，它们的具体意义和使用方法将在相应的章节中详述，这里仅做大概的简单介绍。

系统复位 RESET 中断，也被称作系统复位源。RESET 是一个特殊的中断源，是 AVR 中唯一的不可屏蔽的中断。当 ATmega16 由于各种原因被复位后，程序将跳到复位向量（缺省为 0x0000）处，在该地址处通常放置一条跳转指令，跳转到主程序继续执行（见 2.6.5 节）。

INT0、INT1 和 INT2 是 3 个外部中断源，它们是分别由芯片外部引脚 PD2、PD3、PB2 上的电平的变化或状态触发的。通过对控制寄存器 MCUCR 和控制与状态寄存器 MCUCSR 的配置，外部中断可以定义为由 PD2、PD3、PB2 引脚上的电平的下降沿、上升沿、逻辑电平变化，或者低电平（INT2 仅支持电平变化的边沿触发）触发，这为外部硬件电路和设备向 AVR 申请中断服务提供了很大方便。

TIMER2 COMP、TIMER2 OVF、TIMER1 CAPT、TIMER1 COMPA、TIMER1 COMPB、TIMER1 OVF、TIMER0 OVF、TIMER0 COMP 这 8 个中断是来自于 ATmega16 内部的 3 个定时计数器触发的内部中断。定时计数器处在不同的工作模式下时，这些中断的发生条件和具体意义是不同的，它们的应用将在有关定时计数器介绍的章节中进行详述。

USART RXC、USART TXC、USART UDRE 是来自于 ATmega16 内部的通用同步/异步串行接收和转发器 USART 的 3 个内部中断。当 USART 串口完整接收一个字节、成功发送一个字节以及发送数据寄存器为空时，这 3 个中断会分别被触发。

还有其它 6 个中断也是来自 ATmega16 内部，它们分别由芯片内部集成的各个功能模块产生，其中：SPI STC 为内部 SPI 串行接口传送结束中断，ADC 为 ADC 单元完成一次 A/D 转换的中断，EE_RDY 是片内的 EEPROM 就绪（对 EEPROM 的操作完成）中断，ANA_COMP 是由内置的模拟比较器输出引发的中断，TWI 为内部两线串行接口的中断，SPM_RDY 是对片内的 Flash 写操作完成中断。在本书的相关章节中，会对这些中断的使用进行介绍。

7.2.2 ATmega16 的中断控制

1） 中断优先级的确定

在 AVR 单片机中，一个中断在中断向量区中的位置决定了它的优先级，位于低地址的中断优先级高于位于高地址的中断。因此，对于 ATmega16 来说，复位中断 RESET 具有最高优先级，外部中断 INT0 次之，而 SPM_RDY 中断的优先级最低。也就是说，当与其他中断同时

发生时，SPM_RDY 将最后得到响应。

AVR 单片机采用固定的硬件优先级方式，不支持通过软件对中断优先级的重新设定。因此中断优先级的作用仅体现在当同一时刻有两（多）个中断源向 MCU 申请中断的情况中。在这种情况下，MCU 将根据中断的优先级的不同，把低优先级的中断挂起，首先响应中断优先级最高的那个中断。待优先级最高的中断服务程序执行完成返回后，再顺序响应优先级较低的中断。

2） 中断标志

AVR 有两种机制不同的中断：带有中断标志的中断（可挂起）和不带中断标志的中断（不能挂起）。

在 AVR 中，大多数的中断都属于带中断标志的中断。所谓的中断标志，是每个中断源在其 I/O 空间寄存器中具有自己的一个中断标志位。AVR 的硬件系统在每个时钟周期内都会检测（接受）外部（内部）中断源的中断条件。一旦中断条件满足，AVR 的硬件就会将置位相应的中断标志位（置为“1”），表示向 MCU 提起中断请求。

中断标志位一般在 MCU 响应该中断时，由硬件自动清除，或在中断服务程序中通过读/写专门数据寄存器的方式自动清除。中断标志位除了由硬件自动清除外也可以使用软件指令清除，注意：如用软件方法清除，清除的方法是对其写“1”。

当中断被禁止，或 MCU 不能马上响应中断，则该中断标志将会一直保持，直到中断允许并得到响应之止。已建立的中断标志，实际就是一个中断的请求信号，如果暂时不能被响应，该中断标志会一直保留（除非被用户软件清除掉），此时该中断被“挂起”。如果有多个中断被挂起，一旦中断允许后，各个被挂起的中断将按优先级依次得到中断响应服务。应该注意的是，AVR 在退出中断后至少要再执行一条指令才能去响应其它被挂起的中断。

在 AVR 中，还有个别的中断不带（不设置）中断标志，如配置为低电平触发的外部中断即为此类型的中断。这类中断只要中断条件满足（外部输入低电平），便会一直向 MCU 发出中断申请。这种外部低电平中断有其特殊性，它不产生中断标志，因此不能悬挂记忆。如果由于等待时间过长而得不到响应，可能会因中断条件结束（低电平取消）而失去一次服务机会。另一方面，如果这个低电平维持时间过长，则会使中断服务完成返回后再次响应，使 MCU 重复响应同一中断的请求，进行重复服务。因此，在这在类中断的服务程序中，应该有破坏中断条件产生的操作，例如，在低电平中断的服务程序中，使用相应的操作以释放外部器件加在 INT 引脚上低电平。

低电平中断的重要应用是唤醒处于休眠工作模式的 MCU。因为当 MCU 休眠时，其系统时钟往往处于停止工作状态，使用低电平中断可以将 MCU 唤醒。而这一功能边沿中断是不能代替的，因为边沿信号的检测需要系统时钟。

3）中断屏蔽与管理

为了能够灵活地管理中断，AVR 对中断采用两级控制方式。所谓两级控制是指 AVR 有一个中断允许的总控制位 I（既 AVR 标志寄存器 SREG 中的 I 标志位“SREG.7”），通常称为全局中断允许控制位。同时 AVR 为每一个中断源都设置了独立的中断允许位，这些中断允许位分散位于各中断源所属模块的控制寄存器中。

AVR 指令中 SEI 和 CLI 指令专门用于对全局中断允许位进行置位和清零。当设置 I = 0，表示关闭全局中断，此时 AVR 所有的中断源（除 RESET 外）的中断请求全部被屏蔽，MCU 不响应任何的中断，因此 CLI 也称为关中断指令。当设置 I = 1，表示允许全局中断，此时 AVR 总的中断请求被开放，MCU 可以响应任何的中断请求，但中断请求最终是否能为 MCU 响应，还要取决各个中断源相应的中断允许位的设置。需要注意的是，当使用 CLI 指令关闭全局中断时，中断禁止将立即生效，没有中断可以在执行 CLI 指令再之后发生，即便中断请求是在执行 CLI 指令时产生的也不会得到响应。而在使用 SEI 指令开放全局中断后，要等 CPU 再执

行一条指令之后，中断允许才会有效。也就是说，紧跟 SEI 之后的第一条指令一定会先于任何中断而被首先执行。

因此，AVR 响应一个可屏蔽中断源（假定为 A 中断）的中断的条件是：
响应 A 中断 = 全局中断允许标志 AND 中断 A 允许标志 AND 中断 A 标志
AVR 复位后，各个中断允许位以及全局中断允许位均被清零，这保证了程序在开始执行时（一般程序开头是对芯片内部以及外围系统的初始化配置）不会受到中断的干扰。
因此，在 AVR 复位后的用户初始化程序中，需要先对需要使用的中断源进行必要的配置。待系统初始化过程结束后再置位 I，使系统进入正常的工作状态，开始响应中断请求。

4） 中断嵌套

由于 AVR 在响应一个中断的过程中通过硬件将 I 标志位自动清零，这样就阻止了 MCU 响应其它中断。因此通常情况下，AVR 是不能自动实现中断嵌套的。如要系统中必须要实现中断嵌套的应用，用户可在中断服务程序中使用指令将全局中断允许位开放，通过间接的方式实现中断的嵌套处理。

请读者注意，滥用中断嵌套会造成程序流程的不确定性。因此建议只有当某中断确实需要得到实时响应时才考虑使用中断嵌套处理，一般情况下尽量不要采用中断嵌套，因为 AVR 本身是高速单片机，它运行速度是能足够快速的将中断服务程序执行完的。当然，用户编写中断服务程序时，应遵循尽量短小的原则（关于中断服务程序编写的要求在后面还会介绍）。

7.2.3 AVR的中断响应过程

通常，当某个中断条件成立后，硬件会自动将该中断的标志位置“1”，表示中断产生，同时也作为申请中断服务的请求信号。如果该中断的允许位为“1”，同时 AVR 的全局中断允许标志位 I 也是“1”时，那么 MCU 在执行完当前一条指令之后就会响应该中断。

1） 中断响应的过程

AVR 在响应中断请求时，MCU 会使用 4 个时钟周期自动顺序的完成以下任务：

- ✓ 清零状态寄存器 SREG 中的全局中断允许标志位 I，禁止响应其他中断。
- ✓ 将被响应中断的标志位清零（注意：仅对于部分中断有此操作）。
- ✓ 将中断断点的地址（即当前程序计数器 PC 的值）压入堆栈，并将 SP 寄存器中的堆栈指针减二。
- ✓ 自动将相应的中断向量地址压入程序计数器 PC，即强行转入执行中断入口地址处的指令。

因此，AVR 中断响应时间最少为 4 个时钟周期，就是说 4 个时钟周期后，CPU 便会跳到中断向量处开始执行中断入口的指令，这个中断响应的过程全部由硬件自己实现的，不需要用户程序的干预（当然，中断入口处的指令是用户放置的）。

从执行中断入口的指令开始，便进入了用户编写的中断服务程序的执行了。所以，中断服务要完成什么任务，是由用户的程序决定的。我们已经知道，在中断入口处通常放置的是一条转移指令，这条转移指令应该使 MCU 再次跳到真正的中断服务程序开始处的。因此，当 MCU 响应中断，跳到中断向量处执行转移指令，又要花费 2-3 个时钟周期。固从 MCU 开始响应中断，到真正执行中断服务程序的第一条指令，至少需要 6-7 个时钟周期。

2） 中断返回的过程

AVR 一旦执行中断返回 RETI 指令，MCU 便开始了中断返回的过程。AVR 在中断返回过程中，也是使用 4 个时钟周期自动按顺序完成以下任务：

- ✓ 从栈顶弹出 2 个字节的数据，将这两个数据压入程序计数器 PC 中，并将 SP 寄存器中的堆栈指针加 2。
- ✓ 置位状态寄存器 SREG 中的全局中断允许标志位 I，允许响应其他中断。

因此，AVR 的中断返回过程同样需要 4 个时钟周期才能完成，同样中断返回过程也是硬件自动完成的，或者说，是中断返回指令 RETI 的全部操作过程。

在此之后，被中断打断的程序将继续执行。如果存在其它被挂起的中断，则 AVR 在中断返回后还需执行一条指令，被挂起的中断才会得到响应。

3）中断现场的保护

从上面介绍的中断响应和返回过程可以看出，AVR 的中断响应和返回过程主要都是由硬件自动完成的，而在整个过程中用户程序的作用在于：

- ✓ 中断入口处的指令。用于指引 MCU 转移到中断服务程序。
- ✓ 中断服务程序。完成中断服务的功能。
- ✓ 中断返回指令。指引 MCU 从中断服务程序中返回。

应该尤其引起注意和需要提醒的是：为了提高中断响应的实时性，AVR 在中断响应和返回过程中，硬件上的处理仅仅保护和恢复了中断的断点（PC 值）。而对中断现场没有采取任何处理。

因此，中断现场的保护工作需要用户在自己编写的中断服务程序中通过软件完成，以保证主程序在被打断时所使用的标志位和临时寄存器等不会被中断服务程序改变，例如对状态寄存器 SREG 的保护等！

保护和恢复中断现场通常会利用堆栈进行，同时在中断程序中的其它地方也可能要用到堆栈。使用堆栈保护数据时应该特别谨慎，必须保证数据进栈和出栈的数量一致，以确保在进入中断程序时和中断程序结束时（即将执行 RTEI 指令时）堆栈指针的值相同。因为中断返回时，CPU 会认为此时栈顶保存的值是被保护的断点，将其放入 PC 中。如果堆栈使用不当，将造成程序流程的严重错误。

7.3 中断服务程序的编写

在了解了 AVR 单片机的中断系统之后，本节说明如何利用汇编语言和 C 语言正确的编写 AVR 单片机的中断服务程序，以及编写中断服务程序的基本原则和需要注意的问题。

7.3.1 汇编语言AVR中断程序的编写

使用汇编编写带有中断服务的 AVR 系统程序，其基本的程序框架如下：

```
;*************************************
;ATmega16 使用中断的汇编程序框架
;*************************************
.include "m16def.inc"

;第一部分中断向量区配置,FLASH 空间$000~$028
.org  $000
jmp RESET          ; 复位处理
jmp EXT_INT0       ; IRQ0 中断向量,跳转到外部中断 0 的中断服务程序
```

```
    reti              ; IRQ1 中断向量
    reti
    reti              ; Timer2 比较中断向量
    reti
    reti              ; Timer2 溢出中断向量
    reti
    reti              ; Timer1 捕捉中断向量
    reti
    reti              ; Timer1 比较 A 中断向量
    reti
    reti              ; Timer1 比较 B 中断向量
    reti
    reti              ; Timer1 溢出中断向量
    reti
    reti              ; Timer0 溢出中断向量
    reti
    reti              ; SPI 传输结束中断向量
    reti
    reti              ; USART RX 结束中断向量
    reti
    reti              ; UDR 空中断向量
    reti
    reti              ; USART TX 结束中断向量
    reti
    reti              ; ADC 转换结束中断向量
    reti
    reti              ; EEPROM 就绪中断向量
    reti
    reti              ; 模拟比较器中断向量
    reti
    reti              ; 两线串行接口中断向量
    reti
    reti              ; IRQ2 中断向量
    reti
    reti              ; 定时器 0 比较中断向量
    reti
    reti              ; SPM 就绪中断向量
    reti
    ;第二部分,主程序部分
    .org  $02A
RESET:                ; 主程序开始
    ;堆栈指针的设置,设置堆栈指针为 RAM 的顶部
    ldi r16,high(RAMEND)
    out SPH,r16 ;
```

```
    ldi r16,low(RAMEND)
    out SPL,r16
    ;…….
    ;中断源的初始化
    ldi r20,0x02
    out mcucr,r20          ;将 INT0 设置为下降沿触发
    ldi r20,0x40
    out gifr,r20           ;清除可能存在的 INT0 中断标志
    out gicr,r20           ;开放 INT0 中断允许标志

    ;…….
    sei ; 开放全局中断

    ;正常程序开始
WAIT:
    ;…….                  ;主程序
    rjmp wait
    ;…….
    ;第三部分,中断服务程序
EXT_INT0:
    push r21
    in r21,sreg
    push r21               ;中断现场保护
    ;…….                  ;中断服务
    pop r21
    out sreg,r21           ;中断现场恢复
    pop r21
    reti ;中断返回
```

这个基本的程序框架大体分为三个部分：中断向量区部分、主程序部分和中断服务程序部分。

1） 中断向量区部分

缺省情况下，AVR 的中断向量区在 Flash 程序储存器的最低端。最开始的 0x0000 是不可屏蔽的复位上电的中断向量，此处必然应该放置一条转移到主程序开始处的跳转指令。具体应该按如下方法进行编写：

- ✓ Flash 较小的 AVR，每个中断向量占据一个字的空间。对于使用到的中断，在中断向量处放置一条相对转移指令 RJMP（单字指令），用于跳至相应的中断服务程序；在复位向量处放置一条 RJMP 指令，用于跳转到主程序入口处；对于程序中不使用的中断，为了增加程序的抗干扰性，应在该中断向量处放置一条 RETI 指令。
- ✓ 象 ATmega16 这样 Flash 较大的 AVR，每个中断向量占据了两个字的空间。对于使用到的中断，在中断向量处放置一条绝对转移指令 JMP（双字指令），用于跳至相应的中断服务程序；在复位向量处放置一条 JMP 指令，用于跳转到主程序入口处；对于程序中不使用的中断，为了增加程序的抗干扰性，应在中断向量处连续放置两条 RETI 指令。

在有些系统中,如CVAVR,对于程序中不使用的中断,则是在中断向量处放置rjmp 0x0000或 jmp 0x0000 代替中断返回指令 reti，这种编写方式也是可以采用的。

2）主程序部分

作为单片机嵌入式系统的主程序,在开始阶段通常要对整个系统以及芯片本身进行初始化设置，然后才能进入正常的工作处理流程中。对芯片初始化时的一些必要的设置有：

- ✓ 堆栈指针的初始化。在系统程序中，当调用子程序和中断响应时，硬件系统都是利用堆栈来保护程序返回的断点。由于 AVR 在复位上电过程中自动将堆栈指针寄存器 SP 清零，而且 AVR 堆栈的进栈操作是 SP-1 或 SP-2，所以应该首先对堆栈指针寄存器 SP 进行设置。通常将堆栈指针初始化设置成 SRAM 的最高端，这样可以将 SRAM 空间最大限度的保证和提供给用户程序使用。
- ✓ 中断源的设置。对于系统程序使用到的中断源也要进行必要的设置，设置内容包括中断源的工作方式，中断的产生（触发）条件等。
- ✓ 开放全局中断。AVR 在复位上电过程中自动将全局中断允许标志 I 清零的，因此在初始化完成后，应该开放全局中断，允许 MCU 响应中断。
- ✓ 各中断源相应的中断允许设置。开放了全局中断允许，并不意味者 MCU 就一定能够响应中断了，因为各个中断还有第二级的控制—各自的中断的允许标志。用户的程序还应该根据实际的需要，或在芯片初始化阶段，或当程序运行到需要使用中断的地方，将中断源本身的中断允许开放。

尤其注意的是，在开放中断源本身的中断允许位之前，最好先使用指令将该中断的中断标志位清除，然后马上将中断允许位置“1”。

在开放中断前清除可能存在的中断标志，保证了中断开放后不会形成一次“多余”的中断，这个“多余”的中断有时会造成致命的错误。因为在对中断源进行设置过程中，或中断源对应的硬件模块在工作中都有可能改变中断标志位。

在 AVR 中，通常采用指令对中断标志位清“0”的操作是向该标志位写“1”!

3）中断服务程序

由于在中断向量处通常放置一条转移指令，用于再次跳转到中断服务程序的开始处，所以中断服务程序可以放置在 Flash 空间的任何地方。AVR 汇编语言的中断服务程序通常具有如下形式：

```
lable:                  ;标号，便于定位
    push Rd             ;中断现场保护
    in Rd, sreg
    push Rd
    ……                  ;中断服务程序
    ……
    pop Rd              ;中断现场恢复
    out sreg, Rd
    pop Rd
    reti
```

我们可以看到，汇编的中断服务程序的结构同一般的汇编子程序相同，唯一不同的区别是：中断服务程序的返回指令为 RETI，而一般子程序的返回指令是 RET，所以也可将中断服务程序称为中断服务子程序。

子程序通常使用一个 lable 标号作为开始，该标号即是子程序的名字，也用于程序调用

转移的定位。接下来是子程序的主体部分，而作为中断服务的子程序的结束必须使用一条 RETI（一般子程序为 RET）指令，当程序执行到 RETI 时，表示该中断服务程序的结束，执行中断返回的处理。

具体中断服务程序的编写与其它程序的编写没有区别，但有几个非常关键的地方，在编写中断服务程序时需要高度的重视。

在中断服务程序中，应该首先考虑中断现场的保护和恢复问题。因为中断的产生和响应是随机的，而且在中断服务程序中经常要使用一些寄存器，或对 SRAM 中的变量进行操作，也会有判断和跳转的操作（指令操作会改变 SREG 中标志位!），所以必须确保当从中断服务程序返回时，被中断服务程序改变的现场全部正确的恢复，这样当中断返回后，主程序才能正确继续运行下去。

现场的保护和恢复问题，在编写一般的子程序时也是要首先考虑的。

在编写中断服务程序和子程序时，有大量的问题会出现在现场的保护和恢复问题上，而且比较难于调试，往往需要花费许多时间才能找到原因。因此在编写中断服务程序和子程序时，千万牢记要仔细全面认真考虑中断现场的保护和恢复处理。

确定需要现场保护的原则是保证中断服务程序不会干扰返回后主程序的运行。在编写程序时，应根据具体情况，仔细分析确定需要保护的中断现场。通常状态寄存器 SREG（包含重要的标志位）必须进行保护，同时两部分程序共享的其他寄存器也应进行保护。一般情况下，在中断子程序的一开始就对 SERG 进行保护（在保护状态寄存器 SREG 之前，要避免使用影响标志位的指令，以免破坏寄存器原有的值），因为许多指令的操作都会隐型的改变 SREG 中的标志。现场的保护和恢复通常采用堆栈来完成，因为堆栈操作是最快的方法。因此，在中断服务程序的一开始，通常就是使用 PUSH 指令将需要保护的寄存器入栈（见上面的中断服务程序的基本形式）。

中断服务功能完成之后，在执行 RETI 指令返回主程序之前，还要进行中断现场的恢复处理工作，即使用 POP 指令从堆栈中将保存的现场弹出，重新赋给各个寄存器。由于堆栈遵循的是“先进后出，后进先出”的原则，因此在恢复现场时 POP 的顺序应该与保护现场时 PUSH 的顺序相反，即将首先弹出的值赋给最后保存的寄存器，而将最后弹出的值赋给最先保存的寄存器，依次类推，最后恢复 SREG 寄存器（最先保护 SREG）。

考虑好对中断现场进行保护和恢复之后，就可以开始进行为中断服务所需要完成的功能了。这部分的编写同一般的汇编语言程序编写没有区别，此处不做详述，但要遵循编写中断服务程序的另一个基本原则：中断服务程序应尽可能的短。

编写中断服务程序的两个基本原则：

- ✓ **全面、仔细考虑中断现场的保护和恢复。**
- ✓ **中断服务程序应尽可能的短**

许多人在编写系统程序时，喜欢把所有需要处理工作放在中断服务程序中完成，这不是一个好的程序设计方法。其实，除了一些必须在中断中完成的工作外，对于那些不需要马上处理的工作，如键盘处理、扫描显示等应该放在主程序中完成，也就是说，中断服务程序应尽可能的短。中断服务程序短，不是单纯的看程序指令的多少，重要的是指执行一次中断服务程序所需要的时间短。尽量减少中断服务程序的执行时间有以下的优点：

- ✓ 可以不必采用中断嵌套的技术。AVR 的硬件不支持自动的中断嵌套处理，因此中断

执行时间短的话能够尽快的响应其它被挂起中断，提高系统总体的实时响应速度。

- ✓ 能够防止丢失周期性中断或其它短时中断的丢失。例如，当系统有一个 1ms 产生一次的周期型中断源，你的中断服务程序就必须在 1ms 完成，否则将会造成下一个中断的“丢失”。

有很多情况下，中断仅仅表示外围设备或内部功能部件的工作过程已经达到某种状态，但不需要马上去处理，或者允许在一个比较充裕的限定时间内处理，这就可以将它们的处理工作放到主程序中完成。在这种情况下，最好的方式是定义和使用信号量或标志变量，在中断服务程序中只是简单的对这些信号量或标志量进行必要的设置，不做其它处理就马上返回主程序，由主程序中根据这些信号量或标志量的值进行和完成处理工作。

这样做的另一个好处是，可以大大减少中断服务程序中的对中断现场保护和恢复的工作，从而又减少了中断程序的执行时间，同时也节省了堆栈空间和 Flash 空间（代码少了）。

7.3.2 CodeVision 中断程序的编写

在 AVR 的高级语言开发环境中，都扩展和提供了相应编写中断服务程序的方法，但不同高级语言开发环境中对编写中断服务程序的语法规则和处理方法是不同的。用户在编写中断服务程序前，应对所使用开发平台，中断程序的编写方法，中断的处理方法等有较好的了解。

使用 ICCAVR、CVAVR、BASCOM-AVR 等高级语言编写中断服务程序时，用户通常不必考虑中断现场保护和恢复的处理，这是由于编译器在编译中断服务程序的源代码时，会在生成的目标代码中自动加入相应的中断现场保护和恢复的指令，同时自动采用 RETI 指令作为中断服务的返回指令。

在 CVAVR 中，中断服务程序必须定义成一个特殊的函数，称为中断服务函数。中断服务函数按以下格式定义：

```
interrupt [中断向量号] void 函数名 (void)
{
      ……            //函数体
}
```

其中，关键字 interrupt 声明了该函数为中断服务函数，用以区别于一般软件调用的函数。方括号和方括号中的中断向量号则进一步说明该函数是哪一个中断的服务函数。由于中断函数是 MCU 响应中断时通过硬件自动调用，因此中断函数的返回值和参数均为 void（不能返回函数值，以及带参数）。中断函数的命名规则与一般函数相同。

按照上面的格式，外部中断 INT0（2 号中断）的中断服务函数可以定义如下：

```
interrupt [EXT_INT0] void ext_int0_isr (void)
{
   ……             //函数体
}
```

其中 EXT_INT0 是在头文件 mega16.h 中定义的宏，等同于数字 2。为了便于中断服务程序的编写和阅读，CVAVR 对各个型号 AVR 单片机的中断源都定义了类似的宏，详细定义请参考各个型号 AVR 单片机的头文件（包含于 CVAVR 的 inc 子目录下）。

了解了如何在 CVAVR 中编写中断服务函数后，我们就可以给出一个在 CVAVR 开发平台支

持下，用高级语言 C 编写系统程序框架：

```
#include <mega16.h>

// 第一部分，中断服务函数
interrupt [EXT_INT1] void ext_int1_isr(void)  // 外部中断 INT1 的中断服务函数
{
    ……;            // 中断服务函数
}

// 主程序
void main(void)
{
    ……;
    // 中断源的初始化
    GICR|=0x80;        // External Interrupt(s) initialization
    MCUCR=0x08;        // INT0: Off
    MCUCSR=0x00;       // INT1: On
    GIFR=0x80;         // INT2: Off

    // 开放全局中断
    #asm("sei")

    // 正常程序开始
    while (1)
     {
        ……;
     };
}
```

由于 CVAVR 在编译过程中，会自动帮助用户产生正确的中断向量处的以及初始化堆栈指针的代码，同时在中断服务程序中自动生成中断现场保护和恢复以及使用 RETI 指令返回，因此，按照 CVAVR 的规范编写中断服务函数还是比较方便的。

只要正确定义了中断函数，编译器便能够生成可以保证相应的中断发生时，该函数被自动调用，以及完成了中断现场保护和恢复工作的 AVR 汇编代码，用户只需按照一般的函数编写方法设计函数体即可。

但在主程序的初始化部分中，仍然要对中断源进行必要的设置，以及在进入正常工作程序前注意控制中断允许位的打开和关闭。各个单独的中断允许位的设置是通过寄存器赋值语句实现的，但对全局中断允许位的操作在 CVAVR 中使用的是 C 内嵌汇编指令的方式：用#asm("sei")开放全局中断，用#asm("cli")关闭全局中断。

尽管在 CVAVR 中编写中断服务函数与使用 AVR 汇编相比更加方便，但“中断服务程序应尽可能的短”的编程原则还是非常重要，需要认真分析和考虑。

中断服务函数在编写和使用时还要注意：

1．中断服务函数只能在中断发生时由硬件自动调用，不能像其他函数一样可以通过软件调用。同时，由于程序中不会出现调用语句，因此中断服务函数只需要定义语句，不需要进行说明。

2．CVAVR 在缺省方式下，在生成中断服务函数时，会自动把 R0、 R1、 R15、 R22、 R23、R24、 R25、 R26、 R27、 R30、 R31、SREG，以及用户程序中使用的所有通用寄存器保护起来。如果用户要编写效率更高或特殊的中断服务程序，可以采用关闭编译系统的自动产生中断现场保护和恢复代码功能，嵌入汇编代码等方式自己编写相关的程序。此时需要程序员对 CVAVR 开发环境有更深的了解和掌握，并具备较高的软件设计能力。读者可以通过下面一个简单的例子，体会如何在 CVAVR 中编写效率更高的中断程序。

```
#pragma savereg-        // 关闭自动生成现场保护和恢复代码的功能

interrupt [EXT_INT0]  void my_irq(void)
{
  // 仅保护在本中断服务程序中使用的寄存器,如 R30、 R31 和 SREG
  #asm
      push r30                        ; 中断现场保护部分
      push r31
      in   r30,SREG
      push r30
  #endasm

  /* place the C code or AVR code here */
  /* .... */

  // 恢复保护现场 SREG、R31 和 R30
  #asm
      pop r30                    ; 中断现场恢复
      out SREG,r30
      pop r31
      pop r30
  #endasm
}

#pragma savereg+        // 重新开放自动生成现场保护和恢复代码的功能
```

在上面的中断 INT0 的服务函数中，先使用 CVAVR 中的“#pragma savereg-”编译控制命令将系统自动生成现场保护和恢复代码的功能关闭，这样在编译过程中将不会生成中断现场保护和恢复的代码。因此，中断现场保护和恢复的代码则必须由用户根据中断服务的实际情况自己编写。例子中采用了嵌入 AVR 汇编的方式，在中断服务程序的开始和结束部分插入了中断现场保护和恢复的代码。在这里只将 R30、R31 和 SREG 进行了保护，那就必须保证在该中断服务程序中，所有代码的执行仅仅只影响到这三个被保护的寄存器。

7.4 ATmega16 的外部中断

ATmega16 有 INT0、INT1 和 INT2 是 3 个外部中断源，分别由芯片外部引脚 PD2、PD3、PB2 上的电平的变化或状态作为中断触发信号。

7.4.1 外部中断触发方式和特点

INT0、INT1、INT2 的中断触发方式取决于用户程序对 MCU 控制寄存器 MCUCR 以及 MCU 控制与状态寄存器 MCUCSR 的设定。其中，INT0 和 INT1 支持 4 种中断触发方式，INT2 支持 2 种。

表 7.2　外部中断的 4 种中断触发方式

	INT0	INT1	INT2	说 明
上升沿触发	Yes	Yes	Yes(异步)	
下降沿触发	Yes	Yes	Yes(异步)	
任意电平变化触发	Yes	Yes	—	
低电平触发	Yes	Yes	—	无中断标志

表 7.2 给出了 4 种具体的触发方式，其中的任意电平变化触发表示只要引脚上有逻辑电平的变化就会产生中断申请（不管是上升沿还是下降沿都引起中断触发）。在这 4 种触发方式中，还有以下的一些不同的特点：

- ✓ 低电平触发是不带中断标志类型的，即只要中断输入引脚 PD2 或 PD3 保持低电平，那么将一直会产生中断申请。
- ✓ MCU 对 INT0 和 INT1 的引脚上的上升沿或下降沿变化的识别（触发），需要 I/O 时钟信号的存在（由 I/O 时钟同步检测），属于同步边沿触发的中断类型。
- ✓ MCU 对 INT2 的引脚上的上升沿或下降沿变化的识别(触发)，以及低电平的识别(触发）是通过异步方式检测的，不需要 I/O 时钟信号的存在。因此，这类触发类型的中断经常作为外部唤醒源，用于将处在 Idle 休眠模式，以及处在各种其它休眠模式的 MCU 唤醒。这是由于除了在空闲（Idel）模式时，I/O 时钟信号还保持继续工作，在其它各种休眠模式下，I/O 时钟信号均是处在暂停状态的。
- ✓ 如果使用低电平触发方式的中断作为唤醒源，将 MCU 从掉电模式（Power-down）中唤醒时，电平拉低后仍需要维持一段时间才能将 MCU 唤醒，这是为了提高了 MCU 的抗噪性能。拉低的触发电平将由看门狗的时钟信号采样两次（在通常的 5V 电源和 25℃时，看门狗的时钟周期为 1μs）。如果电平拉低保持 2 次采样周期的时间，或者一直保持到 MCU 启动延时（start-up time）过程之后，MCU 将被唤醒并进入中断服务。如果该电平的保持时间能够满足看门狗时钟的两次采样，但在启动延时（start-up time）过程完成之前就消失了，那么 MCU 仍将被唤醒，但不会触发中断进入中断服务程序。所以，为了保证既能将 MCU 唤醒，又能触发中断，中断触发电平必须维持足够长的时间。
- ✓ 如果设置了允许响应外部中断的请求，那么即便是引脚 PD2、PD3、PB2 设置为输出方式工作，引脚上的电平变化也会产生外部中断触发请求。这一特性为用户提供了使用软件产生中断的途径。

7.4.2 与外部中断相关的寄存器和标志位

在 ATmega16 中，除了寄存器 SREG 中的全局中断允许标志位 I 外，与外部中断有关的寄存器有 4 个，共有 11 个标志位。其作用分别是 3 个外部中断各自的中断标志位，中断允许控制位，和用于定义外部中断的触发类型。

1） MCU 控制寄存器—MCUCR

MCU 控制寄存器 MCUCR 的低 4 位为 INT0（ISC01、ISC00）和 INT1（ISC11、ISC10）中断触发类型控制位，中断触发方式见表 7.3 中定义

位	7	6	5	4	3	2	1	0	
$35($0055)	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	

表 7.3 INT0、INT1 的中断触发方式

ISCn1	ISCn0	中 断 触 发 方 式
0	0	INTn 的低电平产生一个中断请求
0	1	INTn 的下降沿和上升沿都产生一个中断请求
1	0	INTn 的下降沿产生一个中断请求
1	1	INTn 的上升沿产生一个中断请求

MCU 对 INT0、INT1 引脚上电平值的采样在边沿检测前。如果选择脉冲边沿触发或电平变化中断的方式，那么在 INT0、INT1 引脚上的一个脉宽大于一个时钟周期的脉冲变化将触发中断，过短的脉冲则不能保证触发中断。如果选择低电平触发中断，那么低电平必须保持到当前指令执行完成才触发中断。如果是低电平触发方式的话，中断请求将一直保持到引脚上的低电平消失为止。

2）MCU 控制和状态寄存器—MCUCSR

MCU 控制和状态寄存器 MCUCSR 中的第 6 位（ISC2）为 INT2 的中断触发类型控制位，中断触发方式见表 7.4 中定义

位	7	6	5	4	3	2	1	0	
$34($0054)	JTD	ISC2	—	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	5 个 RESET 复位标志					

表 7.4 INT2 的中断触发方式

ISC2	中 断 触 发 方 式
0	INT2 的下降沿产生一个异步中断请求
1	INT2 的上升沿产生一个异步中断请求

3） 通用中断控制寄存器—GICR

通用中断控制寄存器 GICR 的高 3 位为 INT0、INT1 和 INT2 的中断允许控制位，如果 SREG 寄存器中的全局中断 I 位为“1”，以及 GICR 寄存器中相应的中断允许位被置为“1”，当外部引脚 INT0（或 INT1、或 INT2）上的电平变化时，MCU 将会响应相应的中断请求。

位	7	6	5	4	3	2	1	0	
$3B($005B)	INT1	INT0	INT2	—	—	—	IVSEL	IVCE	GICR
读/写	R/W	R/W	R/W	R	R	R	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	

4） 通用中断标志寄存器—GIFR

通用中断标志寄存器 GIFR 的高 3 位为 INT0、INT1 和 INT2 的中断标志位。

位	7	6	5	4	3	2	1	0	
$3A($005A)	INTF1	INTF0	INTF2	—	—	—	—	—	GIFR
读/写	R/W	R/W	R/W	R	R	R	R	R	
初始化值	0	0	0	0	0	0	0	0	

当 INT2..0 引脚上的有效事件满足中断触发条件后，INTF2..0 位会变成“1”。如果此时 SREG 寄存器中 I = 1，以及 GICR 寄存器中的 INTn 被置为“1”，MCU 将响应中断请求，跳至相应的中断向量处开始执行中断服务程序，同时硬件自动将 INTFn 标志位清零。

注意：用户可以使用指令将 INTFn 清除，清除的方式是写逻辑“1”到 INTFn，将标志清零。另外，当 INT0（INT1）设置为低电平触发方式时，标志位 INTF0（INTF1）始终为“0”，这并不意味着不产生中断请求，而是低电平触发方式是不带中断标志类型的中断触发。在低电平触发方式时，中断请求将一直保持到引脚上的低电平消失为止。

通过以上的介绍可以看出，ATmega16 外部中断有多种类型的触发方式，MCU 对中断的检测方式也是不同的。因此，用户在使用外部中断时，还要根据实际的需要，采用合适的外部中断以及选择好中断触发方式。

在系统程序的初始化部分中对外部中断进行设置时（定义或改变触发方式），应先将 GICR 寄存器中该中断的中断允许位清零，禁止 MCU 响应该中断后再设置 ISCn 位。

而在开放中断允许前，一般应通过向 GIFR 寄存器中的中断标志位 INTFn 写入逻辑“1”，将该中断的中断标志位清除，然后开放中断。这样可以防止在改变 ISCn 的过程中误触发中断。

7.5 外部中断应用实例

例 7.1 用按键控制的一位 LED 数码管显示系统

1） 硬件电路

图 7-2 为硬件原理图。其中 LED 数码管的控制显示连接与例 6.4 相同，PA 口工作于输出方式，作为 LED 数码管的段码输出，LED 数码管的位信号接地，因此这个一位的 LED 数码管工作于静态显示方式。图中使用了两个按键 K1、K2，按键的一端分别与 PD2（INT0）、PD3（INT1）连接。INT0 和 INT1 作为外部中断的输入，采用电平变化的下降沿触发方式，当 K1（K2）按下时，会在 PD2（PD3）引脚上产生一个高电平到低电平的跳变，触发 INT0 或 INT1 中断。

系统的功能还是控制一个 8 段数码管显示“0”－“F”16 个十六进制的数字。当系统

上电时，显示“0”。K1 键的作用是加“1”控制键：按 1 次 K1 键，显示数字加 1，依次类推。当第 15 次按 K1 键时，显示“F”，第 16 次按 K1 键，显示又从“0”开始。K2 键的作用是减 1 控制键：按 1 次 K1 键，显示数字减 1，减到“0”后，再从“F”开始。

该电路可以在配套的实验板上通过连线轻松实现。

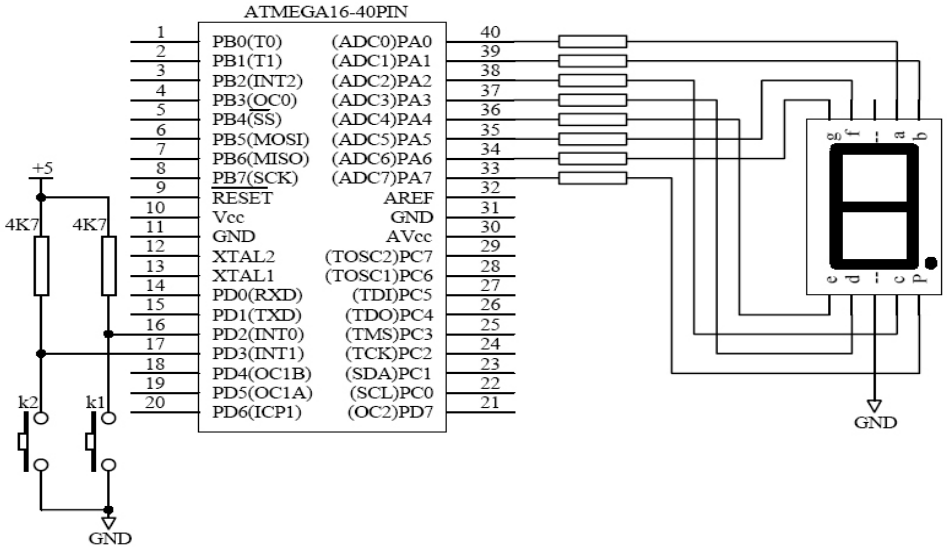

图 7-2 按键中断方式控制一位 LED 数码管显示

2） 软件设计

本实例的显示控制非常简单，主要是说明如何编写中断服务程序和掌握外部中断的基本使用方法。下面各给出一个采用汇编和 C 语言编写的系统程序。

首先是在 CVAVR 中使用 C 语言编写的程序。再次建议读者使用 CVAVR 中的程序生成向导功能来帮助你建立整个程序的框架，以及芯片的初始化部分的语句，可以省掉你过多的查看器件手册和考虑寄存器的设置值等。图 7-3 为 CVAVR 中利用程序生成向导功能配置产生外部中断初始化部分的界面。

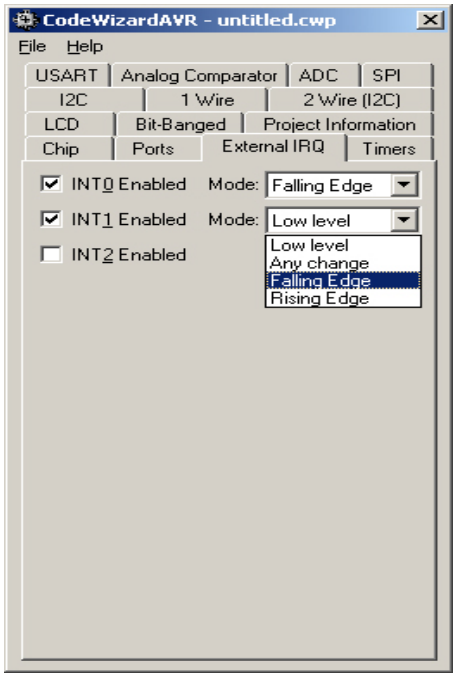

图 7-3 使用 CVAVR 的程序生成向导配置外部中断的界面

```
/*********************************************
File name           : Demo_7_1.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/
#include <mega16.h>

flash char led_7[16]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,
                      0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71};
char counter;

// INT0 中断服务程序
interrupt [EXT_INT0] void ext_int0_isr(void)
{
    if (++counter>=16) counter = 0;
}

// INT1 中断服务程序
interrupt [EXT_INT1] void ext_int1_isr(void)
{
    if (counter) --counter;
    else counter = 15;
}

void main(void)
{
    PORTA=0xFF;
    DDRA=0xFF;

    GICR|=0xC0;         // 允许 INT0、INT1 中断
    MCUCR=0x0A;         // INT0、INT1 下降沿触发
    GIFR=0xC0;          // 清除 INT0、INT1 中断标志位

    counter = 0;        // 计数单元初始化为 0

    #asm("sei")         // 全局中断允许

    while (1)
    {
        PORTA = led_7[counter];     // 显示计数单元
```

```
    };
}
```

上面的程序，就是先利用 CVAVR 的程序生成向导功能进行配置，然后在它生成的程序框架基础上完成的。程序中定义了一个计数变量 counter，执行一次 INT0 中断服务程序，counter 加 1，而执行一次 INT1 中断服务程序，counter 减 1。在主程序中只是显示 counter 的值。INT0、INT1 初始化为电平变化的下降沿触发。

下面给出使用 AVR 汇编编写的系统程序。汇编程序的思想方法同 C 语言的程序相同，读者可以参考本章 7.3.1 中汇编程序框架和 demo_7_1.c，仔细体会使用汇编编写系统程序时，如何正确的编写中断向量区、中断初始化设置以及中断服务程序。

```
;*********************************************************
;AVR 汇编程序实例:Demo_7_1.asm
;使用 INT0、INT1 控制 LED 数码管显示
;Mega16  4MHz
;*********************************************************
.include "m16def.inc"
.def temp     =    r23              ;临时变量
.def counter  =    r24              ;计数变量

;中断向量区配置，FLASH 空间$000~$028
.org  $000
jmp  RESET          ; 复位处理
jmp  EXT_INT0       ; IRQ0 中断向量
jmp  EXT_INT1       ; IRQ1 中断向量
reti                ; Timer2 比较中断向量
nop
reti                ; Timer2 溢出中断向量
nop
reti                ; Timer1 捕捉中断向量
nop
reti                ; Timer1 比较 A 中断向量
nop
reti                ; Timer1 比较 B 中断向量
nop
reti                ; Timer1 溢出中断向量
nop
reti                ; Timer0 溢出中断向量
nop
reti                ; SPI 传输结束中断向量
nop
reti                ; USART RX 结束中断向量
nop
reti                ; UDR 空中断向量
nop
reti                ; USART TX 结束中断向量
```

```
nop
reti              ; ADC 转换结束中断向量
nop
reti              ; EEPROM 就绪中断向量
nop
reti              ; 模拟比较器中断向量
nop
reti              ; 两线串行接口中断向量
nop
reti              ; IRQ2 中断向量
nop
reti              ; 定时器 0 比较中断向量
nop
reti              ; SPM 就绪中断向量
nop

.org  $02A
RESET:                      ; 上电初始化程序
     ldi r16, high(RAMEND)
     out SPH, r16
     ldi r16, low(RAMEND)
     out SPL, r16           ; 设置堆栈指针为 RAM 的顶部

     ser temp
     out ddra, temp         ; 设置 PORTA 为输出,段码输出
     out porta,temp         ; 设置 PORTA 输出全 1

     ldi temp, 0x0a
     out mcucr, temp        ; INT0、INT1 下降沿触发
     ldi temp, 0xc0
     out gicr, temp         ; 允许 INT0、INT1 中断
     out gifr, temp         ; 清除 INT0、INT1 中断标志位

     clr counter
     sei                    ; 使能中断

MAIN:
     clr r0
     ldi zl, low(led_7 * 2)
     ldi zh, high(led_7 * 2)     ; Z 寄存器取得 7 段码组的首指针
     add zl,counter              ; 加上要显示的数字
     adc zh,r0                   ; 加上低位进位
     lpm                         ; 读对应七段码到 R0 中
     out porta, r0               ; LED 段码输出
```

```
    rjmp MAIN                   ; 循环显示

EXT_INT0:
    in temp, sreg
    push temp               ; 中断现场保护
    inc counter             ; 计数单元加 1
    cpi counter, 0x10       ; 与 16 比较
    brne EXT_INT0_RET       ; 小于 16 转中断返回
    clr counter             ; 计数单元清 0
EXT_INT0_RET:
    pop temp
    out sreg, temp          ; 中断现场恢复
    reti                    ; 中断返回

EXT_INT1:
    in temp, sreg
    push temp               ; 中断现场保护
    dec counter             ; 计数单元减 1
    cpi counter, 0xFF       ; 与 255 比较
    brne EXT_INT1_RET       ; 未到 255 转中断返回
    ldi counter, 0x0F       ; 计数单元设置为 15
EXT_INT1_RET:
    pop temp
    out sreg, temp          ; 中断现场恢复
    reti                    ; 中断返回

led_7:                      ;7 段码表
.db 0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07
.db 0x7f,0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71
```

在这两段例程中，都定义了一个计数单元变量 counter，在主程序中只是完成将该计数变量的值转换成 7 段码后输出送显示。外部中断 INT0 和 INT1 都采用引脚上电平变化的下降沿触发中断，INT0 的中断服务程序中将 counter 加一处理，而 INT1 的中断服务程序中则是将 counter 做减一处理。

3） 思考与实践

- ✓ 平稳的按下 K1 和 K2，观察显示的变化。
- ✓ 修改程序，将 INT0、INT1 的中断触发方式分别改成电平变化的上升沿触发中断，以及电平变化触发中断和低电平触发中断，然后运行程序，使显示数据加一或减一变化，体会与使用电平变化的下降沿触发中断的情况做比较，有何不同？
- ✓ 不管使用哪种中断触发方式，经常会产生按键控制不稳定的现象，如显示为“5”时，按下 K1 键一次，应该加一显示“6”，但显示“7”或“8”，甚至更多，这是为什么？
- ✓ 查看在 CVAVR 开发环境中编写编译 demo_7_1.c 的过程中，CVAVR 都生成建立了那些文件？这些文件的内容是什么？文件的扩展名是什么？有何作用？
- ✓ 查看 CVAVR 生成的 demo_7_1.lst 的内容，回答以下问题：a)CVAVR 对中断向量是如何处理的。b)counter 变量分配在什么地方。c)CVAVR 中，如何对中断现场进行保护和恢

复的，是使用硬件堆栈进行保护的吗。

- ✓ 整理出编译 demo_7_1.c 生成的汇编代码，体会宏的使用，并与 demo_7_1.asm 进行比较。
- ✓ CVAVR 在编译 demo_7_1.c 生成的汇编代码中还做了哪些工作？
- ✓ 参考 CVAVR 的 HELP，尝试采用在 CVAVR 中用嵌入汇编的方式编写 INT0 和 INT1 的中断服务程序。

例 7.2 采用外部中断方式，用外部振荡源为基准的时钟系统

1) 硬件电路

在第六章中的例 6.5“六位 LED 数码管动态扫描控制显示设计（一）”中，使用调用 CVAVR 中软件延时函数的方法给出了一个使用 6 个数码管组成时钟系统。采用软件延时，时钟是不准确的，因为一旦系统中使用了中断，就可能打断延时程序的执行，使得延时时间的变化。下面给出以外部振荡源为基准，采用外部中断方式实现的时钟系统的设计。

在“AVR-51 多功能实验开发板”上，有一个采用 CD4060 和 2.048M 晶体构成的 50%占空比、0-5v 的标准方波振荡源。我们将它 10 个标准频率中 500Hz 的输出端与 ATmega16 的 PD3 脚连接，作为外部输入信号，INT1 采用下降沿方式触发中断，那么 500 次中断就是 1 秒钟了。此时，外部 500Hz 的振荡源就是时钟系统的计时基准，这样的时钟系统比使用软件延时的方法要准确的多。

时钟系统显示部分的硬件电路是同图 6-15 相同的，只需要使用一根连线，将板上标准方波振荡源的 500Hz 输出端与 ATmega16 的 PD3 脚连接在一起。

2) 软件设计

下面是一个采用 C 编写的系统源程序。

```
/*********************************************
File name           : Demo_7_2.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/
#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

char time[3];                   // 时、分、秒计数单元
char dis_buff[6];               // 显示缓冲区，存放要显示的 6 个字符的段码值
int time_counter;               // 中断次数计数单元
char posit;                     // （1）
```

```
bit point_on, time_1s_ok;

void display(void)                          // 6 位 LED 数码管动态扫描函数
{
  PORTC = 0xff;                             //  (2)
  PORTA = led_7[dis_buff[posit]];
  if (point_on && (posit==2||posit==4)) PORTA |= 0x80;
  PORTC = position[posit];
  if (++posit >=6 ) posit = 0;              //  (3)
}

// 外部中断 INT1 服务函数
interrupt [EXT_INT1] void ext_int1_isr(void)
{
  display();                                // 调用 LED 扫描显示
  if (++time_counter>=500)
  {
       time_counter = 0;                    //  (4)
       time_1s_ok = 1;                      //  (5)
  }
}

void time_to_disbuffer(void)                // 时钟时间送显示缓冲区函数
{
  char i,j=0;
  for (i=0;i<=2;i++)
  {
       dis_buff[j++] = time[i] % 10;
       dis_buff[j++] = time[i] / 10;
  }
}
```

```
void main(void)
{
  PORTA=0x00;                                  // 显示控制 I/O 端口初始化
  DDRA=0xFF;
  PORTC=0x3F;
  DDRC=0x3F;

  time[2] = 23; time[1] = 58; time[0] = 55; // 设时间初值 23:58:55
  posit = 0;
  time_to_disbuffer();

  GICR|=0x80;                                  // INT1 中断允许
  MCUCR=0x08;                                  // INT1 下降沿触发
  GIFR=0x80;                                   // 清 INT1 中断标志
  #asm("sei")                                  // 全局中断允许

  while (1)
  {
      if (time_1s_ok)                          // 1 秒到
      {
          time_1s_ok = 0;                      // （6）
          point_on = ~point_on;
          if (++time[0] >= 60)                 // 以下时间调整
          {
              time[0] = 0;
              if (++time[1] >= 60)
              {
                  time[1] = 0;
                  if (++time[2] >= 24) time[2] = 0;
              }
          }
```

```
                time_to_disbuffer();              // 新调整好的时间送显示缓冲区
            }
        };
    }
```

3) 思考与实践

该程序同第六章的例 6.5 有许多地方相同或类似，请读者彻底、全面、读懂、理解和体会该段程序，对有（n）注释标记的语句和程序段进行分析，你是否能回答以下问题：

- ✓ Display()函数是如何工作的？与例 6.5 中的 display()函数的不同点在何处？该函数每秒钟执行几次？每执行一次的时间是多少？
- ✓ 在 INT1 的中断服务程序的一开始就调用一次 display()函数，那么整个中断服务程序的执行时间是多少？
- ✓ 可以看出，LED 数码管的显示还是动态扫描方式，那么每一位 LED 数码管在一次扫描过程中点亮的时间是多少？在 1 秒中内循环扫描 6 个数码管的次数是多少？数码管的显示会闪烁吗？
- ✓ 将 500Hz 的输入换接为 GND（0Hz）、125Hz、250Hz、2K、4K 等输入，显示有何变化？显示的时间有何变化？说明原因。
- ✓ 将 500Hz 的输入换接成 64K 的输入，观察显示。然后将 display()中第一条语句“PORTC = 0xff;”去掉，再运行程序（输入保持 64K），观察显示同原来有何区别？说明为什么，并给出“PORTC = 0xff;”语句的作用。
- ✓ 深入体会程序中的变量 time_counter、posit、time_1s_ok 的作用，以及在程序中使用的地方。如果将程序中带有标识的语句（3）、（4）、（5）、（6）分别去掉，或随便改变语句出现的位置时，程序能完成正常的功能吗？说明原因。
- ✓ 将原程序中 INT1 的中断触发方式改为电平变化触发方式，时钟系统有和变化？说明原因。
- ✓ 同第六章的例 6.5 程序 demo_6_5.c 比较，有那些优点和缺点？MCU 的利用有何变化？
- ✓ 有能力的读者请尝试修改 demo_7_2.c 程序，在主程序中加上休眠功能。

例 7.3 利用外部中断实现系统断电保护的实例

在一些实际的控制设备和系统中，当系统电源故障时（电源掉电、电源供电波动），需要将系统中的一些重要数据保存起来，当再次上电时或电源稳定后，系统将保存的重要数据读出来，然后可以从掉电前的状态继续运行下去。这如同我们使用的 PC 所具有的掉电保护功能，它能在掉电前一刻把用户正在使用的程序和界面保存在硬盘中，下次开机后，自动返回用户在掉电前的程序和界面。

要实现这个应用，需要考虑以下几个问题：

- ✓ 重要数据保存的介质。ATmega16 片内集成有 512 个字节的 EEPROM，因此重要数据可以保存在片内的 EEPROM 中。但由于 EEPROM 写入次数有限（>10 万次），因此不能在程序正常运行中频繁的写 EEPROM。为保证 EEPROM 不会过早失效，重要数据应在断电前一刻写入。
- ✓ 写 EEPROM 的时间比较长，需要 8ms/字节，因此在硬件设计上需要有一个掉电预检测电路，它能预先检测出掉电的可能性，并通知单片机进行掉电保护处理。
- ✓ 在掉电预检测电路发出掉电信号后，系统的电源供电不能马上停止，还需要提供系统（尤其是 MCU）一段时间足够的电源。时间的长短根据需要保存的数据多少确定，一般约几

百毫秒左右，让 MCU 有时间将数据写入 EEPROM 中。

✓ MCU 如何能尽快响应掉电信号，如何实现掉电处理。

下面是一个利用外部中断 INT0 实现系统断电保护的设计实例。在这个实例中，系统程序运行的重要标志和数据共计有 30 个字节（需要保护的数据），这些数据不能在程序正常运行中写入 EEPROM，这会很快使 EEPROM 失效。因此，数据写入 EEPROM 的时间应该是在断电的前一刻。写入 EEPROM 一个字节的时间为 8ms，那么写入 30 字节的时间至少需要 240ms。

系统的硬件设计上有一个掉电预检测电路，它能在 MCU 真正的掉电停止工作前 300ms 通知 MCU 进行掉电保护处理，硬件电路见图 7-4。

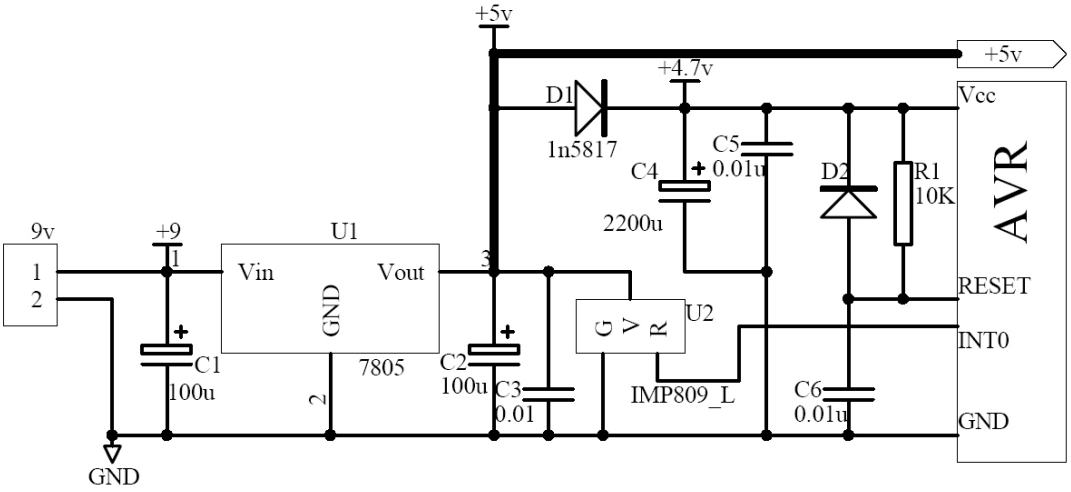

图 7-4 掉电保护电路

在图中，外部 9V 电源通过 7805 稳压到 5V，作为系统电源使用。而 AVR 的工作电源则是单独提供的，由 5v 系统电源通过低压差肖特极二极管 1N5817 后得到。IN5817 的正向压降为 0.3v，因此，AVR 的工作电压为 4.7v。电源监控芯片 IMP809-L 的监控电压为 4.63V，当系统电源的电压低于 4.63V 时，在 R 脚上产生由高电平到低电平的变化，使 AVR 进入 INT0 中断。

该电路的工作原理为：首先通过配置 AVR 的熔丝位，设置 BOD 掉电检测电压门限为 2.7V，并允许 BOD 检测。因此，当 AVR 的 Vcc 电压掉到 2.7v 以下时，AVR 就停止工作（掉电检测功能是 AVR 片内的功能之一，见第二章的 2.6.2 AVR 的复位源和复位方式）。电源监控芯片 IMP809-L 检测电压门限为 4.63v，用于检测系统电源的电压。当系统电源大于 4.63v 时，IMP809-L 的 R 端输出高电平，整个系统正常工作。当系统电源的电压跌到 4.63v 以下时，IMP809-L 的 R 脚输出低电平，作为 AVR 外部中断 INT0 的申请。INT0 设计为掉电处理中断，其主要任务是备份系统运行的重要数据到 EEPROM 中。

在提供 AVR 工作的电源系统中，大容量的电解电容 C5 作为储能电容，一旦系统电源电压下降，二极管 1N5817 截止，此时 AVR 可以靠 C5 提供的电储可以继续工作一段时间。C5 容量应足够大，在系统电源掉电过程中，IMP809-L 的 R 端输出低电平（下降到 4.63v）时，要能够保证维持 AVR 的工作电压 Vcc 从 4.7v 降到 2.7V 的时间超过 300ms，使 AVR 有时间做紧急处理和备份数据。AVR 写 EEPROM 大约需要 50-100mA 的电流，所以电容 C5 的值应该在 1000u～4700u，需要保存的数据越多，C5 的容量应该越大。

INT0 是 AVR 优先级最高的中断，采用外部电平变化的下降沿触发方式。一旦 IMP809-L 的 R 脚电平由正常的高电平变为低电平时，将触发 INT0 中断，进入 INT0 掉电中断服务程序。

在 INT0 掉电保护中断服务程序中，应按以下的步骤和过程处理：

A) 紧急处理，关闭所有外部器件的工作，或将外部状态设置到安全模式，如关闭马达、

开关等，保证系统不出事故。

B) 将 AVR 所有 I/O 设置为输入方式，最大程度的减少 AVR 芯片对电源的消耗。

C) 将重要数据写入到 EEPROM 中。

D) 循环检测 INT0 引脚是否恢复高电平。如为高电平则转到下一步 E 执行；如果 INT0 电平一直为低，程序将在此循环，直到完全停止运行（因为储能电容 C5 的电压低于 2.7v 后，AVR 的 BOD 起作用，产生内部复位，AVR 停止运行程序）。

E) 软件延时一段时间。

F) 再次检测 INT0 引脚电平。为低电平时转回 D 再次循环检测；为高电平时继续向下执行（这种情况表示系统电源受到干扰或短时掉电，现已经恢复正常）。

G) 恢复外部器件工作（此时尽管进入了掉电保护程序，但 AVR 在 C5 的维持下，一直正常工作，所有的数据并没有破坏，可以继续进行工作）；

H) 中断返回。

在实际应用中，系统断电保护的设计是一个比较难的问题，实现的方法和手段也有不同。这个设计主要是作为一个使用外部中断的例子，让读者可以从中体会到如何合理和正确的使用外部中断。

思考与练习

1. 什么是中断？计算机采取中断有什么好处？说明中断的作用和用途。
2. 什么叫中断源？ATmega16 有那些中断源？各有什么特点？
3. 什么是中断系统？中断系统的功能是什么？
4. 中断优先级有什么作用？AVR 的中断优先级如何确定的？
5. 什么叫断点？什么叫中断现场？断点和中断现场的保护和恢复有什么意义？
6. 在 AVR 中，中断断点和中断现场的保护是如何实现的？
7. 什么是中断向量？AVR 的中断系统是如何构成的，它完成哪些功能？
8. 对于不使用的中断，它的中断向量处应如何处理？试比较放入一条“JMP 0000”和放入一条“RETI”的区别，哪个更好些？为什么？
9. AVR 对中断采用两级控制方式，它是如何控制的？
10. AVR 响应中断是有条件的，请说出这些条件是什么。
11. 请详细说明 AVR 中断响应的全过程。在这个过程中，硬件完成了哪些工作，软件完成了哪些工作？
12. 以 AVR 外部中断使用为例，说明在中断初始化程序中需要设置那些内容，以及如何正确的编写中断的初始化程序？
13. 说明全局中断允许标志位、各个独立中断允许标志位和中断标志位的作用。如何清除中断标志位？
14. 系统上电时，AVR 的中断允许标志位、各个独立中断允许标志位和中断标志位的初始值是什么？全局中断允许标志位只是通过指令来置位或清除吗？那么各个独立中断允许标志位呢？
15. 在编写中断服务程序时，应该注意那些问题？有那些原则需要注意的？
16. AVR 中断响应的最短时间为多少？如何估计和计算中断服务的时间？
17. 全面分析总结中断服务程序和一般的子程序的相同点和不同点。
18. 可以在程序中使用语句直接调用中断服务程序吗？
19. 比较用汇编语言和 C 语言编写中断服务程序的优劣，在哪些情况下应考虑采用汇编（或嵌入部分汇编）编写中断服务程序？

20. AVR 的外部中断有几种触发方式？适合那些应用场合？
21. 使用低电平触发方式的外部中断时应该注意那些问题？
22. 在 AVR 中，如果要使 A 中断能够打断 B 中断而形成中断嵌套，而其它中断不允许中断嵌套，试描述 A 的中断服务程序应做怎样的处理。

本章参考文献：

1.《ATmega16 数据手册》（英文，CDROM），ATMEL，www.atmel.com

第 8 章　定时计数器的结构与应用

定时计数器（Timer/Counter）是单片机中最基本的接口之一，它的用途非常广泛，常用于计数、延时、测量周期、频率、脉宽、提供定时脉冲信号等。在实际应用中，对于转速，位移、速度、流量等物理量的测量，通常也是由传感器转换成脉冲电信号，通过使用定时计数器来测量其周期或频率，再经过计算处理获得。

相对于一般 8 位单片机而言，AVR 不仅配备了更多的定时计数器接口，而且还是增强型的，如通过定时计数器与比较匹配寄存器相互配合，生成占空比可变的方波信号，即脉冲宽度调制输出 PWM 信号，用于 D/A、马达无级调速控制、变频控制等，功能非常强大。

ATmega16 一共配置了 2 个 8 位和 1 个 16 位，共 3 个定时计数器，它们是 8 位的定时计数器 T/C0、T/C2 和 16 位的定时计数器 T/C1。本章将着重对 AVR 的 8 位定时计数器的结构、功能和应用进行讲解，并介绍基本的使用设计方法。

8.1　定时计数器的结构

在单片机内部，一般都会集成由专门硬件电路构成的可编程定时计数器。定时计数器最基本的功能就是对脉冲信号“自动”进行计数。这里所谓的“自动”，指计数的过程是由硬件完成的，不需要 MCU 的干预。但 MCU 可以通过指令设置定时计数器的工作方式，以及根据定时计数器的计数值或工作状态做必要的处理和响应。

学习和使用定时计数器时，必须注意以下的基本要素：

- ✓ 定时计数器的长度。定时计数器的长度是指计数单元的位长度，一般为 8 位（一个字节）或 16 位（2 个字节）。
- ✓ 脉冲信号源。脉冲信号源是指输入到定时计数器的计数脉冲信号。通常用于定时计数器计数的脉冲信号可以由外部输入引脚提供，也可以由单片机内部提供。
- ✓ 计数器类型。计数器类型是指计数器的计数运行方式，可分为加一（减一）计数器，单程计数或双向计数等。
- ✓ 计数器的上下限。计数器的上下限指计数单元的最小值和最大值。一般情况下，计数器的下限值为零，上限值为计数单元的最大计数值，即 255（8 位）或 65535（16 位）。需要注意的是，当计数器工作在不同模式下时，计数器的上限值并不都是计数单元的最大计数值 255 或 65535，它将取决于用户的配置和设定。
- ✓ 计数器的事件。计数器的事件指计数器处于某种状态时的输出信号，该信号通常可以向 MCU 申请中断。如当计数器计数到达计数上限值 255 时，产生“溢出”信号，向 MCU 申请中断。

8.1.1　8 位定时计数器T/C0 的结构

ATmega16 中有两个 8 位的定时计数器：T/C0、T/C2，它们都是通用的多功能定时计数器，其主要特点是：

- ✓ 单通道计数器。
- ✓ 比较匹配时清零计数器（自动重装特性，Auto Reload）。
- ✓ 可产生无输出抖动（glitch-free）的，相位可调的脉宽调制（PWM）信号输出。
- ✓ 频率发生器。

- ✓ 外部事件计数器（仅 T/C0）。
- ✓ 带 10 位的时钟预分频器。
- ✓ 溢出和比较匹配中断源（TOV0、OCF0 和 TOV2、OCF2）。
- ✓ 允许使用外部引脚的 32kHz 手表晶振作为独立的计数时钟源（仅 T/C2）。

T/C0、T/C2 的主要结构和大部分的功能是相同或类似的，因此，我们先介绍 T/C0 的结构和应用。

1．T/C0 的组成结构

图 8-1 为 8 位 T/C0 的硬件结构框图。图中给出了 MCU 可以操作的寄存器以及相关的标志位。在 T/C0 中，有两个 8 位的寄存器：计数寄存器 TCNT0 和输出比较寄存器 OCR0。其它相关的寄存器还有 T/C0 的控制寄存器 TCCR0，中断标志寄存器 TIFR 和定时器中断屏蔽寄存器 TIMSK。T/C0 的计数器事件输出信号有两个，计数器计数溢出 TOV0 和比较匹配相等 OCF0。这两个事件的输出信号都可以申请中断，中断请求信号 TOV0、OCF0 可以在定时器中断标志寄存器 TIFR 中找到，同时在定时器中断屏蔽寄存器 TIMSK 中，可以找到与 TOV0、OCF0 对应的两个相互独立的中断屏蔽控制位 TOIE0、OCIE0。

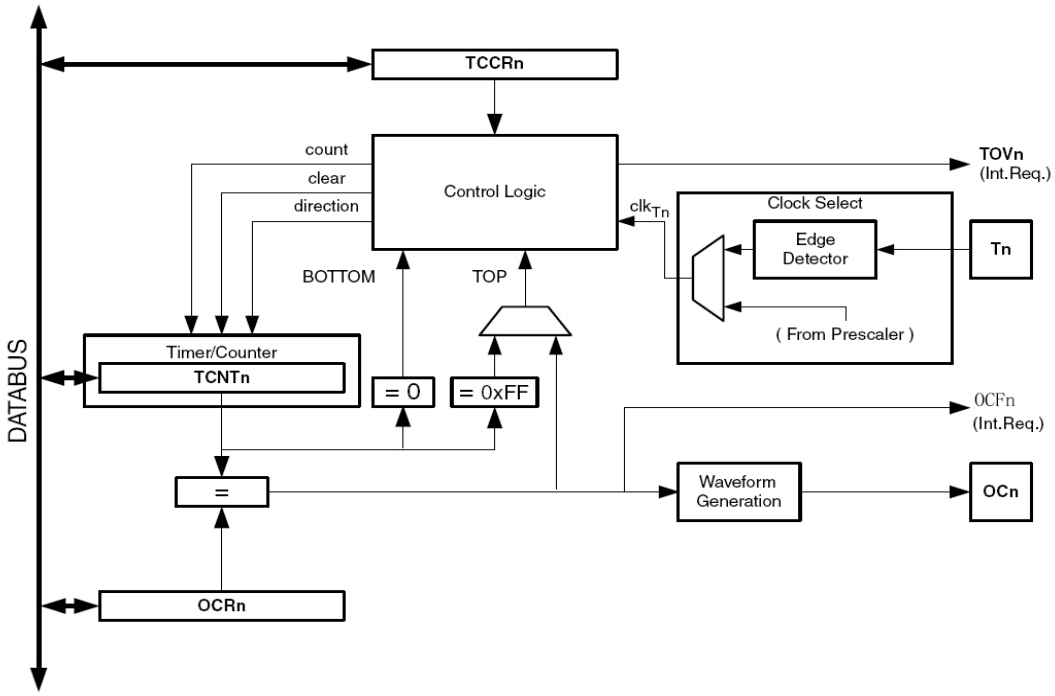

图 8-1　8 位 T/C0 的结构框图

（1）T/C0 的时钟源

T/C0 的计数时钟源可由来自外部引脚 T0 的信号提供，也可来自芯片的内部。图 8-2 为 T/C0 时钟源部分的内部功能图。

- T/C0 计数时钟源的选择

T/C0 的时钟源的选择由 T/C0 的控制寄存器 TCCR0 中的 3 个标志位 CS0[2:0]确定，共有 8 种选择。其中包括无时钟源（停止计数），外部引脚 T0 的上升沿或下降沿，以及内部系统时钟经过一个 10 位预定比例分频器分频的 5 种频率的时钟信号（1/1、1/8、1/64、1/256、1/1024）。T/C0 与 T/C1 共享一个预定比例分频器，但它们时钟源的选择是独立的。

- 使用系统内部时钟源

当定时计数器使用系统内部时钟作为计数源时，通常作为定时器和波形发生器使用。因为系统时钟的频率是已知的，这样通过计数器的计数值就可以知道时间值。

AVR 在定时计数器和内部系统时钟之间增加了一个预定比例分频器，分频器对系统时钟信号进行不同比例的分频，分频后的时钟信号提供定时计数器使用。利用预定比例分频器，定时计数器可以从内部系统时钟获得几种不同频率的计数脉冲信号，使用非常灵活。

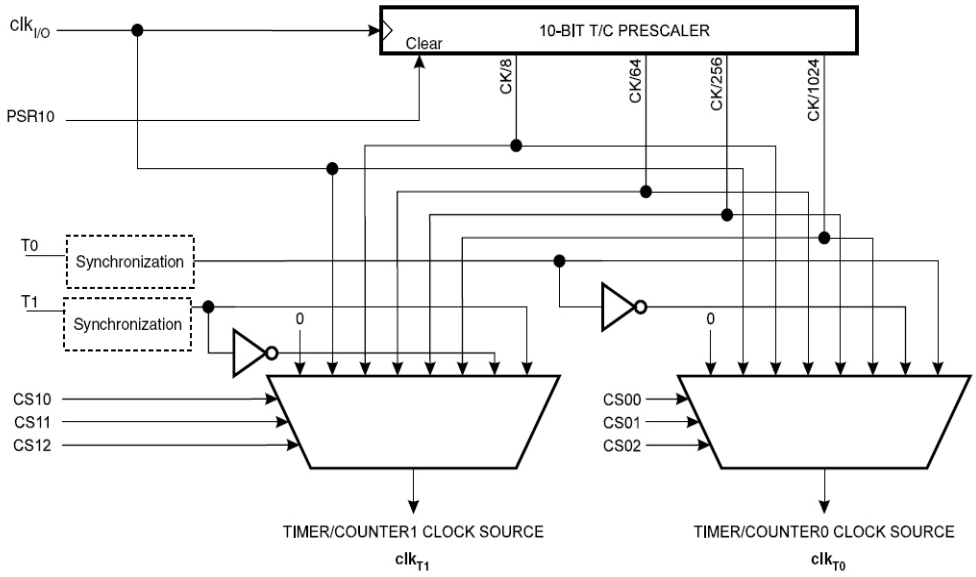

图 8-2 T/C0 的时钟源与 10 位预定比例分频器

- 使用外部时钟源

当定时计数器使用外部时钟作为计数源时，通常作为计数器使用，用于记录外部脉冲的个数。图 8-3 为外部时钟源的检测采样逻辑功能图。

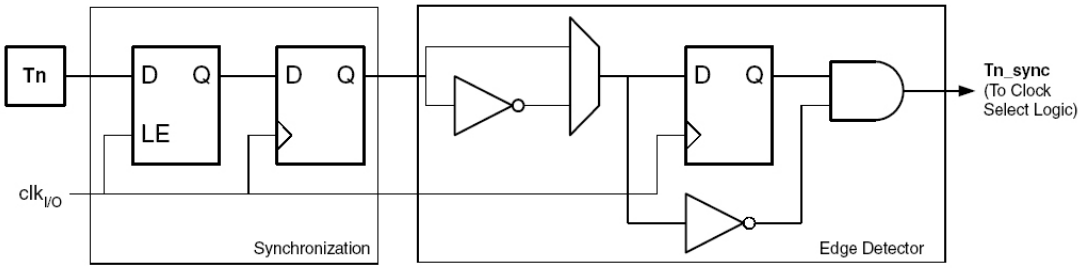

图 8-3 T/C0 外部时钟检测采样逻辑功能图

外部引脚 T0（PB0）上的脉冲信号可以作为 C/T0 的计数时钟源。PB0 引脚内部有一个同步采样电路（Synchronization），它在每个系统时钟周期都对 T0 引脚上的电平进行同步采样，然后将同步采样信号送到边沿检测器（Edge Detector）中。同步采样电路在系统时钟的上升沿将引脚信号电平打入寄存器，因此当系统的时钟频率大大高于外部引脚上电平变化的频率时，同步采样寄存器可以看作是透明的。边沿检测电路对同步采样的输出信号进行边沿检测，当检测到一个正跳变（CS0[2:0]=7）或负跳变（CS0[2:0]=8）时产生一个计数脉冲 CLK_{T0}。

由于引脚内部的同步采样和边沿检测电路的存在，引脚电平的变化需要经过 2.5～3.5 个系统时钟后才能在边沿检测的输出端上反映出来。因此，要使外部时钟源能正确的被引脚的检测采样，外部时钟源的最高频率不能大于 $f_{clk\_I/O}/2.5$，脉冲宽度也要大于一个系统时钟周期。另外，外部时钟源是不进入预定比例分频器进行分频的。

（2）T/C0 的计数单元

T/C0 的计数单元是一个可编程的 8 位双向计数器，图 8-4 是它的逻辑功能图，图中符号所代表的意义如下：

- 计数（count）　　TCNT0 加 1 或减 1。
- 方向（direction）　　加或减的控制。
- 清除（clear）　　清零 TCNT0。
- 计数时钟（clk_{T0}）　　C/T0 时钟源
- 顶部值（TOP）　　表示 TCNT0 计数值到达上边界。
- 底部值（BOTTOM）　　表示 TCNT0 计数值到达下边界（零）。

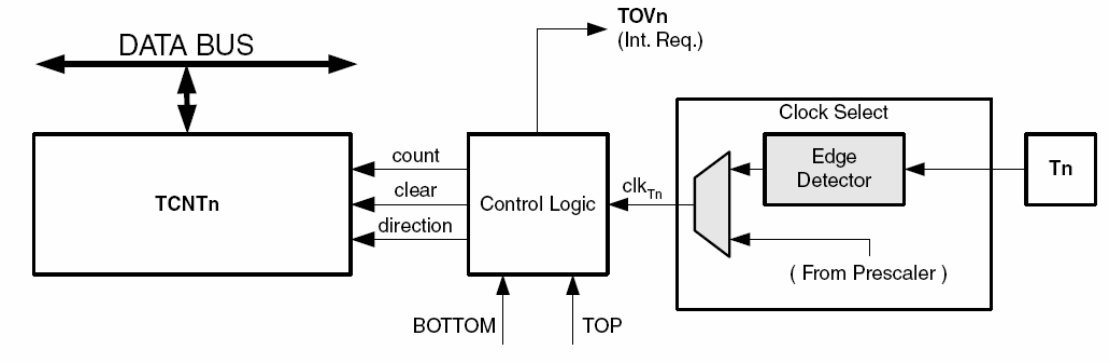

图 8-4　T/C0 计数单元逻辑功能图

T/C0 根据计数器的工作模式，在每一个 clk_{T0} 时钟到来时，计数器进行加 1、减 1 或清零操作。clk_{T0} 的来源由标志位 CS0[2:0]设定。当 CS0[2:0]=0 时，计数器停止计数（无计数时钟源）。

T/C0 的计数值保存在 8 位的寄存器 TCNT0 中，MCU 可以在任何时间访问（读/写）TCNT0。MCU 写入 TCNT0 的值将立即覆盖其中原有的内容，同时也会影响到计数器的运行。

计数器的计数序列取决于寄存器 TCCR0 中标志位 WGM0[1:0]的设置。WGM0[1:0]的设置直接影响到计数器的计数方式和 OC0 的输出，同时也影响和涉及 T/C0 的溢出标志位 TOV0 的置位。标志位 TOV0 可以用于产生中断申请。

（3）比较匹配单元

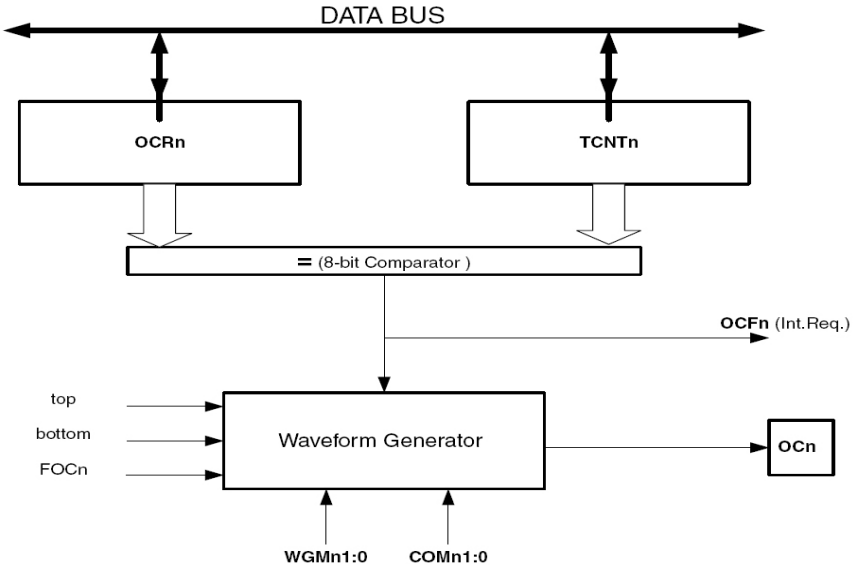

图 8-5　T/C0 比较匹配单元逻辑功能图

图 8-5 为 T/C0 的比较匹配单元逻辑功能图。在 T/C0 运行期间，比较匹配单元一直将寄存器 TCNT0 的计数值同寄存器 OCR0 的内容进行比较（硬件进行自动比较处理）。一旦两者相

等，在下一个计数时钟脉冲到达时置位 OCF0 标志位。标志位 OCF0 也可以用于产生中断申请。根据 WGM0[1:0]和 COM0[1:0]的不同设置，可以控制比较匹配单元产生和输出不同类型的脉冲波形。

寄存器 OCR0 实际上配置有一个辅助缓存器。当 T/C0 工作在非 PWM 模式下时，该辅助缓存器处于被禁止使用状态，此时 MCU 直接访问和操作寄存器 OCR0。当 T/C0 工作在 PWM 模式时，该辅助缓存器投入使用，这时 MCU 对 OCR0 的访问操作，实际上是对 OCR0 的辅助缓存器操作。一旦计数器 TCNT0 的计数值达到设定的最大值（TOP）或最小值（BOTTOM）时，辅助缓存器中的内容将同步更新比较寄存器 OCR0 的值。这将有效防止产生奇边非对称的 PWM 脉冲信号，使输出的 PWM 波中没有杂散脉冲。

- 强制输出比较

在非 PWM 波形发生模式下，写 1 到强制输出比较位（FOC0）时，将强制比较器产生一个比较匹配输出信号。强制比较输出信号不会置 OCF0 标志位或重新装载/清零计数器，但是会像真的发生了比较匹配事件一样更新 OC0 输出引脚输出。

- 通过写 TCNT0 寄存器屏蔽比较匹配事件

任何 MCU 对 TCNT0 寄存器的写操作都会屏蔽在下一个定时器时钟周期中的发生的比较匹配事件，即使在定时器暂停时。这一特性使 OCR0 可以被初始化为与 TCNT0 相同的值，而不会在定时计数器被使能时触发中断。

- 使用输出比较单元

由于在任何工作模式下，写 TCNT0 寄存器都会使得输出比较匹配事件被屏蔽一个定时器时钟周期，因此可能会影响比较匹配输出的正确性。例如，写入一个与 OCR0 相同的值到 TCNT0 时，将丢失一次比较匹配事件，从而引起发生不正确的波形。同样，当定时器是向下计数时，不要将下边界的值写入 TCNT0。

外部引脚 OC0 的设置必须在设置该端口引脚（PB3）为输出之前。设置 OC0 的值最简单的方法是在通常模式下使用 FOC0 来设置，这是因为在改变工作模式时，OC0 寄存器将保持其原来的值。

需要注意的是，COM0[1:0]是无缓冲的，改变 COM0[1:0]位的设置，会立即影响 T/C0 的工作方式。

（4）比较匹配输出单元

标志位 COM0[1:0]有两个作用：定义 OC0 的输出状态，以及控制外部引脚 OC0 是否输出 OC0 寄存器的值。图 8-6 为比较匹配输出单元的逻辑图。

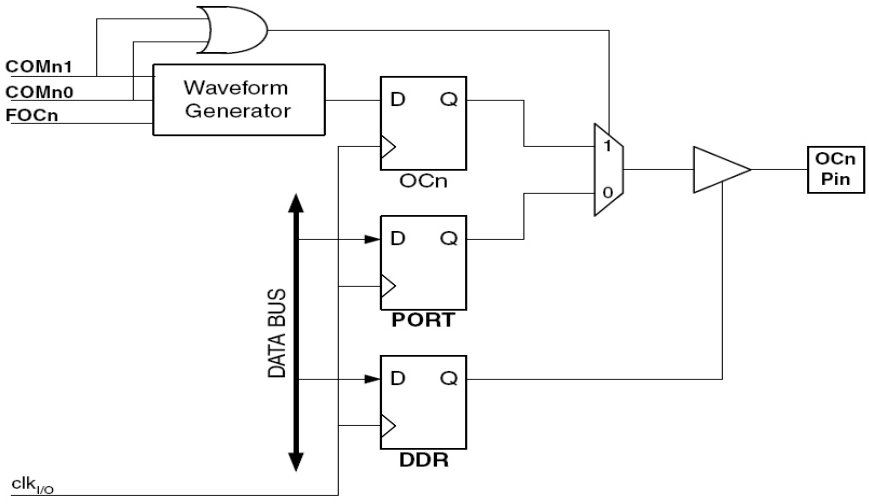

图 8-6　T/C0 比较匹配输出单元逻辑图

当标志位 COM0[1:0]中任何一位为“1”时，波形发生器的输出 OC0 取代引脚 PB3 原来的 I/O 功能，但引脚的方向寄存器 DDRB3 仍然控制 OC0 引脚的输入/输出方向。如果要在外部引脚 PB3 上输出 OC0 的逻辑电平，应设定 DDRB3 定义该引脚为输出脚。采用这种结构，用户可以先初始化 OC0 的状态，然后再允许其由引脚 PB3 输出。

（5）比较输出模式和波形发生器

T/C0 有四种工作模式，根据 COM0[1:0]的不同设定，波形发生器将产生各种不同的脉冲波形，如 PWM 波形的产生和输出。但只要 COM0[1:0]=0，波形发生器对 OC0 寄存器没有任何作用。

2．与 8 位 T/C0 相关的寄存器

（1）T/C0 计数寄存器—TCNT0

位	7	6	5	4	3	2	1	0	
$32($0052)									TCNT0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	

TCNT0 是 T/C0 的计数值寄存器，该寄存器可以直接被 MCU 读写访问。写 TCNT0 寄存器将在下一个定时器时钟周期中阻塞比较匹配。因此，在计数器运行期间修改 TCNT0 的内容，有可能将丢失一次 TCNT0 与 OCR0 的匹配比较操作。

（2）输出比较寄存器—OCR0

位	7	6	5	4	3	2	1	0	
$3C($005C)									OCR0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	

8 位寄存器 OCR0 中的数据用于同 TCNT0 寄存器中的计数值进行匹配比较。在 T/C0 运行期间，比较匹配单元一直将寄存器 TCNT0 的计数值同寄存器 OCR0 的内容进行比较。一旦 TCNT0 的计数值与 OCR0 的数据匹配相等，将产生一个输出比较匹配相等的中断申请，或改变 OC2 的输出逻辑电平。

（3）定时计数器中断屏蔽寄存器—TIMSK

位	7	6	5	4	3	2	1	0	
$39($0059)	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	

- 位 7（位 1）— OCIE2（OCIE0）：T/C2（T/C0）输出比较匹配中断允许标志位

当 OCIE2（OCIE0）被设为“1”，且状态寄存器中的 I 位被设为“1”时，将使能 T/C2（T/C0）的输出比较匹配中断。若在 T/C2（T/C0）上发生输出比较匹配，即 OCF2=1（OCF0=1）时，则执行 T/C2（T/C0）输出比较匹配中断服务程序。

- 位 6（位 0）— TOIE2（TOIE0）：T/C2（T/C0）溢出中断允许标志位

当 TOIE2（TOIE0）被设为“1”，且状态寄存器中的 I 位被设为“1”时，将使能 T/C2（T/C0）溢出中断。若在 T/C2（T/C0）上发生溢出，即 TOV2=1（TOV0=1）时，则执行 T/C2（T/C0）溢出中断服务程序。

（4）定时计数器中断标志寄存器—TIFR

位	7	6	5	4	3	2	1	0	
$38($0058)	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	

- 位 7（位 1）— OCF2（OCF0）：T/C2（T/C0）比较匹配输出的中断标志位

当 T/C2（T/C0）输出比较匹配成功，即 TCNT2=OCR2（TCNT0=OCR0）时，OCF2（OCF0）位被设为“1”。当转入 T/C2(T/C0)输出比较匹配中断向量执行中断处理程序时，OCF2(OCF0)由硬件自动清零。写入一个逻辑“1”到 OCF2(OCF0)标志位将清除该标志位。当寄存器 SREG 中的 I 位、OCIE2（OCIE0）以及 OCF2（OCF0）均为“1”时，T/C2（T/C0）的输出比较匹配中断被执行。

- 位 6（位 0）— TOV2（TOV0）：T/C2（T/C0）溢出中断标志位

当 T/C2（T/C0）产生溢出时，TOV2（TOV0）位被设为“1”。当转入 T/C2（T/C0）溢出中断向量执行中断处理程序时，TOV2（TOV0）由硬件自动清零。写入一个逻辑“1”到 TOV2（TOV0）标志位将清除该标志位。当寄存器 SREG 中的 I 位、TOIE2(TOIE0)以及 TOV2(TOV0)均为“1”时，T/C2（T/C0）的溢出中断被执行。在 PWM 模式中，当 T/C2（T/C0）计数器的值为 0x00 并改变计数方向时，TOV2（TOV0）自动被置为“1”。

（5）T/C0 控制寄存器—TCCR0

位	7	6	5	4	3	2	1	0	
$33($0053)	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	

8 位寄存器 TCCR0 是 T/C0 的控制寄存器，它用于选择计数器的计数源，工作模式和比较输出的方式等。

- 位 7—FOC0：强制输出比较

FOC0 位只在 T/C0 被设置为非 PWM 模式下工作时才有效，但为了保证同以后的器件兼容，在 PWM 模式下写 TCCR0 寄存器时，该位必须被写零。当将一个逻辑“1”写到 FOC0 位时，会强加在波形发生器上一个比较匹配成功信号，使波形发生器依据 COM0[1:0]位的设置而改变 OC0 输出状态。注意：FOC0 的作用仅如同一个选通脉冲，而 OC0 的输出还是取决于 COM0[1:0]位的设置。

一个 FOC0 选通脉冲不会产生任何的中断申请，也不影响计数器 TCNT0 和寄存器 OCR0 的值。一旦一个真正的比较匹配发生，OC0 的输出将根据 COM0[1:0]位的设置而更新。

- 位 3，6—WGM0[1:0]：波形发生模式

这两个标志位控制 T/C0 的计数和工作方式，计数器计数的上限值，以及确定波形发生器的工作模式（见表 8.1）。T/C0 支持的工作模式有：普通模式，比较匹配时定时器清零（CTC）模式，以及两种脉宽调制（PWM）模式。

表 8.1 T/C0 的波形产生模式

模　　式	WGM01	WGM00	T/C0 工作模式	计数上限值	OCR0 更新	TOV0 置位
0	0	0	普通模式	0xFF	立即	0xFF
1	0	1	PWM，相位可调	0xFF	0xFF	0x00
2	1	0	CTC 模式	OCR0	立即	0xFF
3	1	1	快速 PWM	0xFF	0xFF	0xFF

- 位 5，4—COM0[1:0]：比较匹配输出方式

这两个位用于控制比较输出引脚 OC0 的输出方式。如果 COM0[1:0]中的任何一位或两位被置“1”，OC0 的输出将覆盖 PB3 引脚的通用 I/O 端口功能，但此时 PB3 引脚的数据方向寄存器 DDRB3 位必须置为输出方式。当引脚 PB3 作为 OC0 输出引脚时，其输出方式取决于 COM0[1:0]和 WGM0[1:0]的设定。

表 8.2 给出了在 WGM0[1:0]的设置为普通模式和 CTC 模式（非 PWM）时，COM0[1:0]位的功能定义。表 8.3 给出了在 WGM0[1:0]的设置为快速 PWM 模式时，COM0[1:0]位的功能定义。表 8.4 给出了在 WGM0[1:0]设置为相位可调的 PWM 模式时，COM0[1:0]位的功能定义。

表 8.2 比较输出模式，非 PWM 模式（WGM = 0、2）

COM01	COM00	说　　明
0	0	PB3 为通用 I/O 引脚（OC0 与引脚不连接）
0	1	比较匹配时触发 OC0（OC0 为原 OC0 的取反）
1	0	比较匹配时清零 OC0
1	1	比较匹配时置位 OC0

表 8.3 比较输出模式，快速 PWM 模式（WGM = 3）

COM01	COM00	说　　明
0	0	PB3 为通用 I/O 引脚（OC0 与引脚不连接）
0	1	保留
1	0	比较匹配时清零 OC0，计数值为 0xFF 时置位 OC0
1	1	比较匹配时置位 OC0，计数值为 0xFF 时清零 OC0

表 8.4 比较输出模式，相位可调 PWM 模式（WGM = 1）

COM01	COM00	说　　明
0	0	PB3 为通用 I/O 引脚（OC0 与引脚不连接）
0	1	保留
1	0	向上计数过程中比较匹配时清零 OC0 向下计数过程中比较匹配时置位 OC0
1	1	向上计数过程中比较匹配时置位 OC0 向下计数过程中比较匹配时清零 OC0

- 位 2，0—CS0[2:0]：T/C0 时钟源选择

这 3 个标志位被用于选择设定 T/C0 的时钟源，见表 8.5。

表 8.5 T/C0 的时钟源选择

CS02	CS01	CS00	说　　明
0	0	0	无时钟源（停止 T/C0）
0	0	1	clk_{T0S}（不经过分频器）

0	1	0	clk_{T0S}/8（来自预分频器）
0	1	1	clk_{T0S}/64（来自预分频器）
1	0	0	clk_{T0S}/256（来自预分频器）
1	0	1	clk_{T0S}/1024（来自预分频器）
1	1	0	外部 T0 脚，下降沿驱动
1	1	1	外部 T0 脚，上升沿驱动

8.1.2 8 位T/C0 的工作模式

T/C0 的控制寄存器 TCCR0 的标志位 WGM0[1:0]和 COM0[1:0]的组合构成 T/C0 的四种工作模式以及 OC0 不同方式的输出。

（1）普通模式（WGM0[1:0]=0）

普通模式是 T/C0 最简单和基本的一种工作方式。T/C0 工作在普通模式下时，计数器为单向加 1 计数器，一旦寄存器 TCNT0 的值到达 0xFF（上限值），在下一个计数脉冲到来时便恢复为 0x00，并继续单向加 1 计数。当 TCNT0 由 0xFF 转变为 0x00 的同时，溢出标志位 TOV0 置位为“1”，用于申请 T/C0 溢出中断。一旦 MCU 响应 T/C0 的溢出中断，硬件则将自动把 TOV0 清零。

考虑到 T/C0 在正常的计数过程中，当 TCNT0 值由 0xFF 返回 0x00 时能将标志位 TOV0 置“1”（注意：不能清零），而且当 MCU 响应 T/C0 的溢出中断时，硬件会自动把 TOV0 清零。因此溢出标志位 TOV0 也可作为计数器的第 9 位使用，使 T/C0 变成 9 位的计数器。但这种提高定时器的分辨率的方法，需要通过软件配合实现。

与其它工作模式相比，T/C0工作在普通模式时，不会产生任何其它的特殊状态，用户可以随时改变计数器TCNT0的数值。

在普通模式中，同样可以使用比较匹配功能产生定时中断，但最好不要在普通模式下使用输出比较单元来产生 PWM 波形输出，因为这将占用过多的 MCU 的时间。

（2）比较匹配清零计数器 CTC 模式（WGM2[1:0]=2）

T/C0 工作在 CTC 模式下时，计数器为单向加 1 计数器，一旦寄存器 TCNT0 的值与 OCR0 的设定值相等（此时寄存器 OCR0 的值为计数上限值），就将计数器 TCNT0 清零为 0x00，然后继续向上加 1 计数。通过设置 OCR0 的值，可以方便地控制比较匹配输出的频率，也方便了外部事件计数的应用。图 8-7 为 CTC 模式的计数时序图。

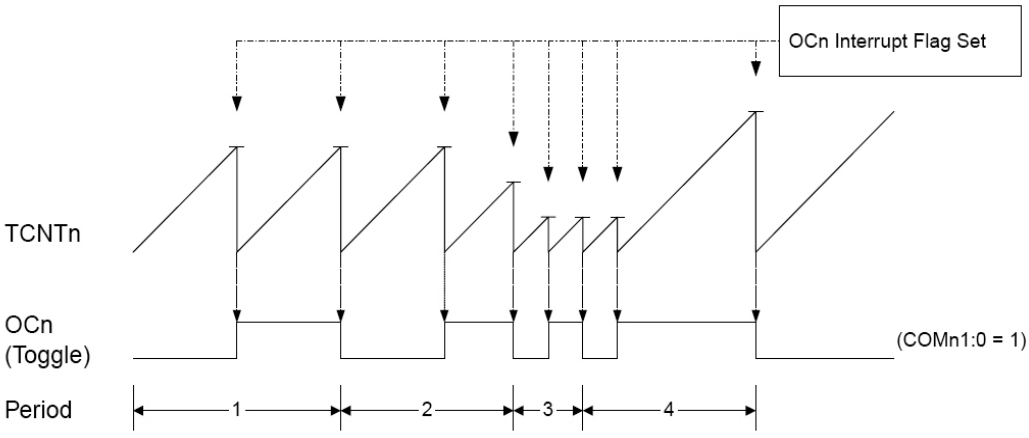

图 8-7 T/C0 的 CTC 模式计数时序

在 TCNT0 与 OCR0 匹配的同时，置比较匹配标志位 OCF0 为“1”。标志位 OCF0 可以用于申请中断。一旦 MCU 响应比较匹配中断，用户在中断服务程序中可以修改 OCR0 的值。

修改 OCR0 的值时需要注意，当 T/C0 的计数时钟频率比较高时，如果写入 OCR0 的值与

0x00 接近，可能会丢失一次比较匹配成立条件。例如：当 TCNT0 的值与 OCR0 匹配相等，此时 TCNT0 被硬件清零并申请中断；在中断服务中重新改变设置 OCR0 为 0x05；但中断返回后 TCNT0 的计数值已经为 0x10 了，因此便丢失了一次比较匹配成立条件。此时计数器将继续加 1 计数到 0xFF，然后返回 0x00，当再次计数到 0x05 时，才能产生比较匹配成功。

在 CTC 模式下利用比较匹配输出单元产生波形输出时，应设置 OC0 的输出方式为触发方式（COM0[1:0]=1）。OC0 输出波形的最高频率为 $f_{OC0}=f_{clk\_I/O}/2$（OCR0=0x00）。其他的频率输出由下式确定，式中 N 的取值为 1、8、64、256 或 1024。

$$f_{OC\,0}=\frac{f_{\mathrm{clk\_I/O}}}{2N(1+OCR\ 0)}$$

除此之外，与普通模式相同，当 TCNT0 计数值由 0xFF 转到 0x00 时，标志位 TOV0 置位。

（3）快速 PWM 模式（WGM0[1:0]=3）

T/C0 工作在快速 PWM 模式可以产生较高频率的 PWM 波形。当 T/C0 工作在此模式下时，计数器为单程向上的加 1 计数器，从 0x00 一直加到 0xFF（上限值），在下一个计数脉冲到来时便恢复为 0x00，然后再从 0x00 开始加 1 计数。在设置正向比较匹配输出（COM0[1:0]=2）方式中，当 TCNT0 的计数值与 OCR0 的值相同匹配时清零 OC0，当计数器的值由 0xFF 返回 0x00 时置位 OC0。而在设置反向比较匹配输出（COM0[1:0]=3）方式中，当 TCNT0 的计数值与 OCR0 的值相同匹配时置位 OC0，当计数器的值由 0xFF 返回 0x00 时清零 OC0。图 8-8 为快速 PWM 工作时序图。

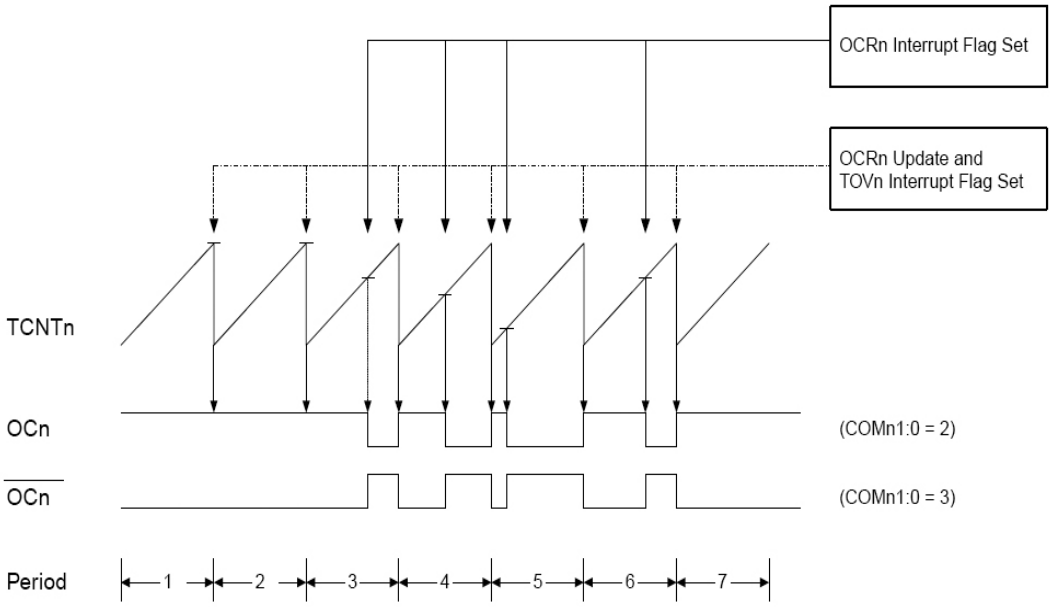

图 8-8　T/C0 快速 PWM 工作时序

由于快速 PWM 模式采用单程计数方式，所以它可以产生比相位可调 PWM 模式高一倍频率的 PWM 波。因此快速 PWM 模式适用于电源调整、DAC 等应用。

在 TCNT0 的计数值到达 0xFF 时，置溢出标志位 TOV0 为“1”。标志位 TOV0 可以用于申请中断。一旦 MCU 响应 TOV0 中断，用户可以在中断服务程序中修改 OCR0 的值。

OC0 输出的 PWM 波形的频率输出由下式确定，式中 N 的取值为 1、8、64、256 或 1024。

$$f_{OC0PWM}=\frac{f_{\mathrm{clk\_I/O}}}{256N}$$

通过设置寄存器 OCR0 的值，可以获得不同占空比的脉冲波形。OCR0 的一些特殊值，会产生极端的 PWM 波形。当 OCR0 的设置值为 0x00 时，会产生周期为 MAX+1 的窄脉冲序列。而

设置 OCR0 的值为 0xFF 时，OC0 的输出为恒定的高（低）电平。

当 OC0 的输出方式为触发方式时(COM0[1:0]=1),T/C0 将产生占空比为 50%的 PWM 波形。此时设置 OCR0 的值为 0x00 时，T/C0 将产生占空比为 50%的最高频率 PWM 波形，频率为 $f_{OC0}=f_{clk\_I/O}/2$。

（4）相位可调 PWM 模式（WGM0[1:0]=1）

相位可调 PWM 模式可以产生高精度相位可调的 PWM 波形。当 T/C0 工作在此模式下时，计数器为双程计数器：从 0x00 一直加到 0xFF，在下一个计数脉冲到达时，改变计数方向，从 0xFF 开始减 1 计数到 0x00。设置正向比较匹配输出（COM0[1:0]=2）方式：在正向加 1 过程中，TCNT0 的计数值与 OCR0 的值相同匹配时清零 OC0；在反向减 1 过程中，当计数器 TCNT0 的值与 OCR0 相同时置位 OC0。设置反向比较匹配输出（COM0[1:0]=3）方式：在正向加 1 过程中，TCNT0 的计数值与 OCR0 的值相同匹配时置位 OC0；在反向减 1 过程中，当计数器 TCNT0 的值与 OCR0 相同时清零 OC0。图 8-9 为相位可调 PWM 工作时序图。

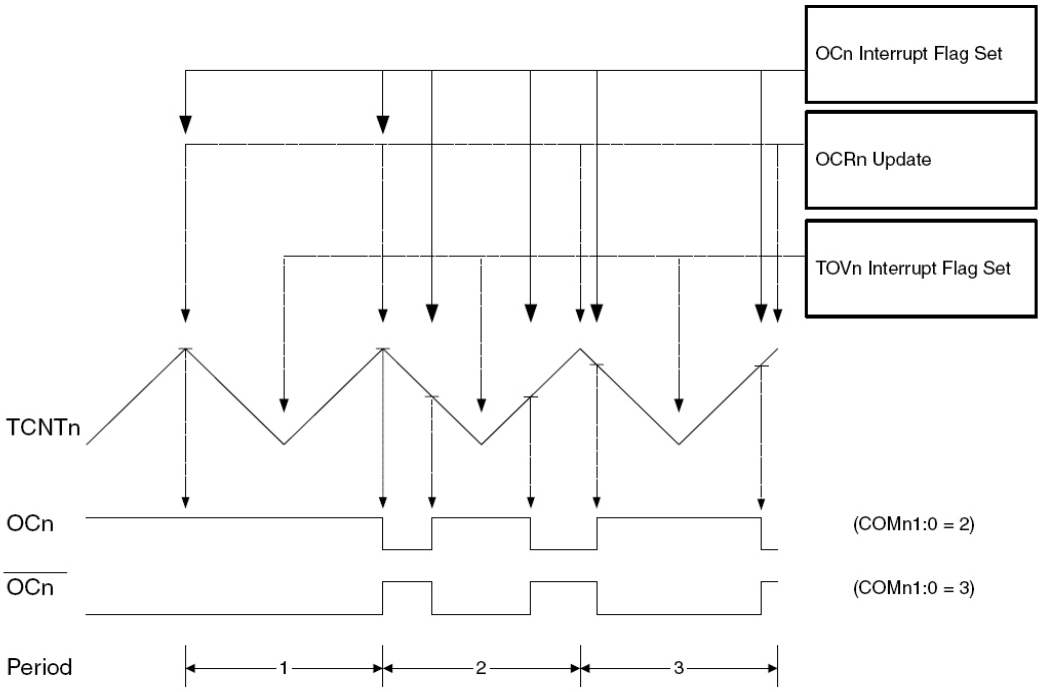

图 8-9　T/C0 相位可调 PWM 工作时序

由于相位可调 PWM 模式采用双程计数方式，所以它产生的 PWM 波的频率比快速 PWM 低。其相位可调的特性（即 OC0 逻辑电平的改变不是固定在 TCNT0=0x00 处），适用于马达控制一类的应用。

在 TCNT0 的计数值到达 0x00 时，置溢出标志位 TOV0 为“1”。标志位 TOV0 可以用于申请溢出中断。

在相位可调 PWM 模式下，OC0 输出的 PWM 波形频率由下式确定，式中 N 的取值为 1、8、64、256 或 1024。

$$f_{OC0PCPWM}=\frac{f_{clk\_I/O}}{510N}$$

通过设置寄存器 OCR0 的值，可以获得不同占空比的脉冲波形。OCR0 的一些特殊值，会产生极端的 PWM 波形。当 COM0[1:0]=2 且 OCR0 的值为 0xFF 时，OC0 的输出为恒定的高电平；而 OCR0 的值为 0x00 时，OC0 的输出为恒定的低电平。

8.1.3 8 位T/C0 的计数工作时序

图 8-10、图 8-11、图 8-12 和图 8-13 给出了 T/C0 在同步工作情况下的各种计数时序，同时给出标志位 TOV0 和 OCF0 的置位条件。图中 MAX=0xFF，BOTTOM=0x00，TOP=[OCRn]。

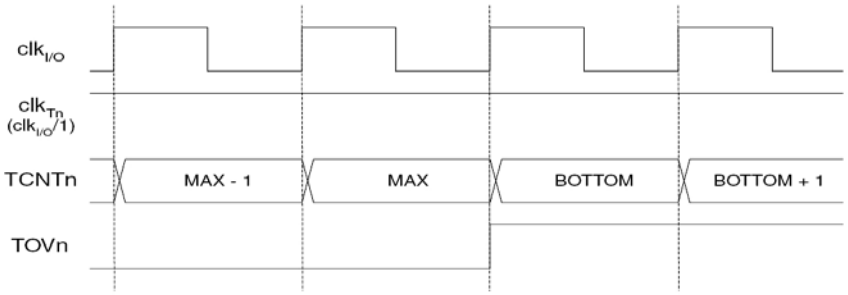

图 8-10 T/C0 计数时序（无预分频）

图 8-10 是 T/C0 对外部时钟或直接对内部时钟（无分频）计数工作的时序图。从图中看出，T/C0 的计数是同系统时钟同步的（在系统时钟上升沿）。当 TCNT0 的值到达 MAX（0xFF）后，在下一个系统时钟的上升沿处把 TCNT0 的值清为 BOTTOM（0x00），同时置位 TOV0 申请中断。然而 T/C0 的计数过程并没有停止，重新从 0x00 开始继续加 1 计数。

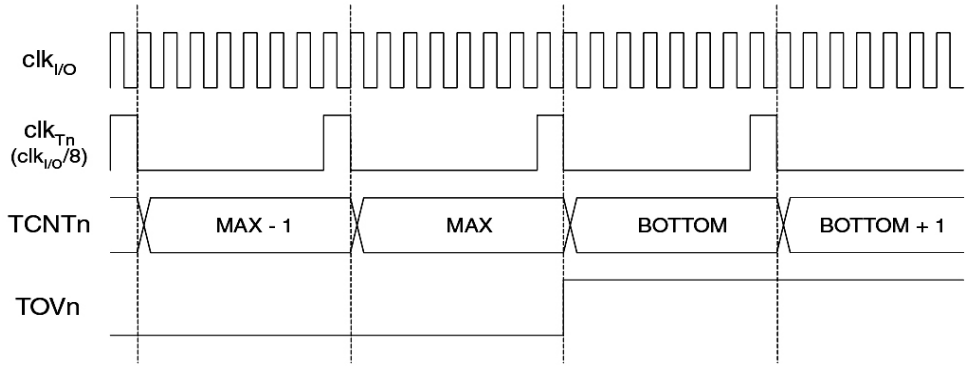

图 8-11 T/C0 计数时序，带 1/8 预分频

图 8-11 是 T/C0 对经过预分频器的内部时钟（8 分频）计数工作的时序图。从图中看出，T/C0 的计数是同系统时钟同步的（每隔 8 个系统时钟的上升沿）。当 TCNT0 的值到达 MAX（0xFF）后，在接下来第 8 个系统时钟的上升沿处将 TCNT0 的值清为 BOTTOM（0x00），同时置位 TOV0 申请中断。然而 T/C0 的计数过程并没有停止，重新从 0x00 开始继续加 1 计数。

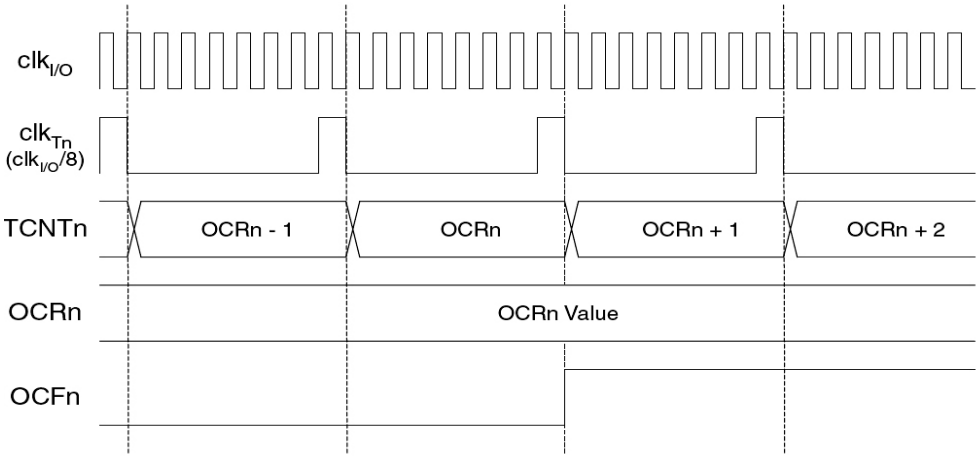

图 8-12 T/C0 计数时序，OCFn 置位，带 1/8 预分频（CTC 模式除外）

图 8-12 给出了 T/C0 工作在各种模式（除 CTC 模式外）时，比较匹配输出的标志位 OCF0 的置位情况。在 T/C0 对经过预分频器的内部时钟（8 分频）计数过程中，比较匹配单元将寄存器 TCNT0 中的计数值和比较匹配寄存器 OCR0 中的值进行比较。一旦两者相等，在下一个计数脉冲到达时置位 OCF0 标志位，申请中断，然而 T/C0 的计数过程并没有停止，继续加 1 向上计数。

图 8-13 是 T/C0 工作在 CTC 模式时的比较匹配输出标志位 OCF0 的置位情况。在 T/C0 对经过预分频器的内部时钟（8 分频）计数过程中，比较匹配单元将寄存器 TCNT0 中的计数值和比较匹配寄存器 OCR0 中的值进行比较。一旦两者相等（此时 OCR0 的值是计数器的上限值 TOP），在下一个计数脉冲到达时置位 OCF0 标志位，申请中断，并同时将 TCNT0 的值清为 BOTTOM（0x00）。然而 T/C0 的计数过程并没有停止，重新从 0x00 开始继续加 1 计数。

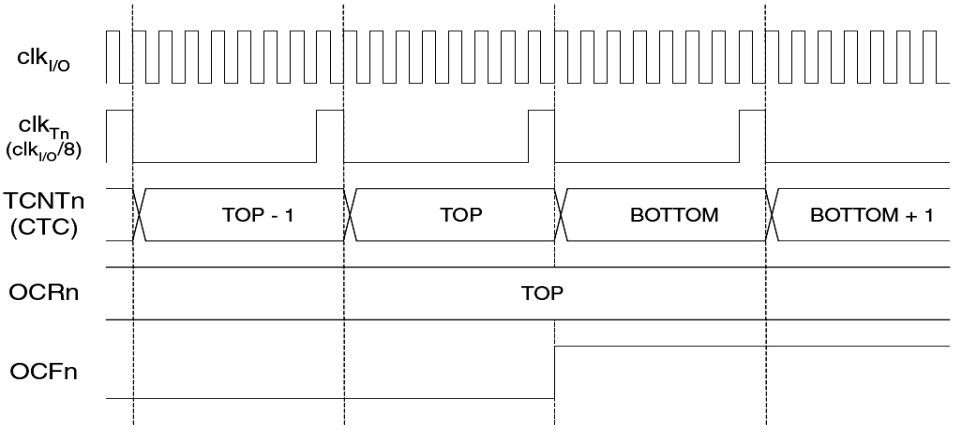

图 8-13　T/C0 计数时序，OCFn 置位，带 1/8 预分频（CTC 模式）

8.2　8 位定时计数器T/C0 的应用

使用定时计数器进行系统设计是非常灵活的，用户需要对实际的情况做仔细的分析，充分考虑利用定时计数器的特点，采用不同的方式来实现。

尽管定时计数器的基本工作原理比较简单，其基本特点为：

- ✓ 对一个序列的脉冲信号进行计数，而且计数过程由硬件自己完成，不需要软件干预。
- ✓ 一旦计数值到达某个值，通常是 MAX（0xFF）、BOTTOM（0x00）或 TOP 时，可以产生中断申请，通知 MCU 进行处理。

但在实际使用中，如果能巧妙的结合定时计数器各种不同的工作模式，则会产生多种变化。因此用户在使用定时计数器进行设计时，应该注意以下几个要点。

- ✓ 仔细确定使用哪个定时计数器。ATmega16 一共配置了 2 个 8 位和 1 个 16 位，共 3 个定时计数器，不仅长度不同，而且其功能也不同。要选择适合的定时计数器使用。
- ✓ 脉冲信号源。脉冲信号源是指输入到定时计数器的计数脉冲信号。用于定时计数器计数的脉冲信号可以由外部输入引脚提供，也可以由单片机内部提供。当使用内部计数脉冲信号时，应选择合适的分频比例与计数值的配合（建议使用 CVAVR 系统中的程序生成器功能）。
- ✓ 计数器的工作模式和触发方式的选择。
- ✓ 中断服务程序的正确设计。定时计数器的使用通常都是与中断相结合一起使用的，因此要非常清楚中断的产生条件，以及在中断服务程序中正确的进行中断处理以及相关的设置。

本节将给出一些定时计数器基本的应用和设计，以便读者能在学习和理解这些基本使用方法的过程中，更好的掌握定时计数器的特点，进而能达到能真正的在实际系统中灵活的使用定时计数器的目的。

8.2.1 外部事件计数器

T/C0 作为外部事件计数器使用时，是指其计数脉冲信号来自外部的引脚 T0（PB0）。

注意，外部引脚 T0 输入的脉冲信号是不通过 ATmega16 内部的预分频器的。

通常对外部输入的脉冲信号的基本处理有两种：

- ✓ 对外部的脉冲信号进行计数，即记录脉冲的个数，一旦记录的脉冲个数到达一设定值时进行必要的处理。
- ✓ 对外部的脉冲信号的频率（周期）进行测定。

在本小节中，我们只介绍前一种的应用。关于对外部的脉冲信号的频率（周期）进行测定的设计，将在本书后面的相关章节中介绍。

例 8.1 2N 分频系统设计一

1） 硬件电路

2N 分频系统要实现的功能是对 T0 脚输入的方波信号进行偶数次的分频，以获得频率低于 T0 输入的方波信号。

本设计的硬件电路非常简单，将实验板上的 250Hz 的方波信号输出与 ATmega16 的 T0 脚连接，作为 T/C0 计数器的外部输入。另外将 PA0 作为分频后的脉冲输出脚，用 PA0 控制一个 LED 的显示，通过 LED 的亮暗变化可以简单的观察方波的频率。当然最好的方式是使用示波器观察 PA0 的输出。

2） 软件设计

首先考虑使用 T/C0 的普通模式（WGM0[1:0]=0），采用 T0 上升沿触发（CS0[2:0]=111），并设置 TCNT0 的初值为 0xFF。当 T0 引脚输入电平出现一个上跳变时，T/C0 的 TCNT0 回到 0x00，并产生溢出中断（参考图 8-10），在溢出中断服务中重新设置 TCNT0 为 0xFF，并改变 PA0 口的输出电平（取反输出）。在 T0 引脚输入电平出现第二个上跳变时，又会产生中断，在中断服务程序中再次改变 PA0 的输出，这样在 PA0 上就得到 T0 的 2 分频输出信号。同理，如果将 TCNT0 的初值设置为 0xFE，则在 PA0 上得到 T0 的 4 分频输出信号，0xFD→6 分频，0xFC—>8 分频……，而当 TCNT0 的初值设置为 0x00 时，可实现最大 512 分频的输出。

下面，我们给出在 PA0 上输出 1Hz 的方波(LED 亮 0.5s，暗 0.5s)的设计和程序。由于 T0 输入的频率为 250Hz，所以分频系数为 250，因此 TCNT0 的初值=255-124（0x83），即 T/C0 计数 125 次时 PA0 的电平改变一次。

再次建议读者使用 CVAVR 中的程序生成向导功能来帮助你建立整个程序的框架，以及芯片的初始化部分的语句，可以省掉你过多的查看器件手册和考虑寄存器的设置值等。

图 8-14 是在 CVAVR 的程序生成向导中设置 T/C0 的对话窗口。选择 T/C0 的计数时钟源为 T0 的上升沿，工作方式为普通模式，允许溢出中断，TCNT0 的初值为 0x83。

利用 CVAVR 的程序生成向导，在它的帮助下生成一个程序框架后，然后再加入自己的程序和进行必要的修改。

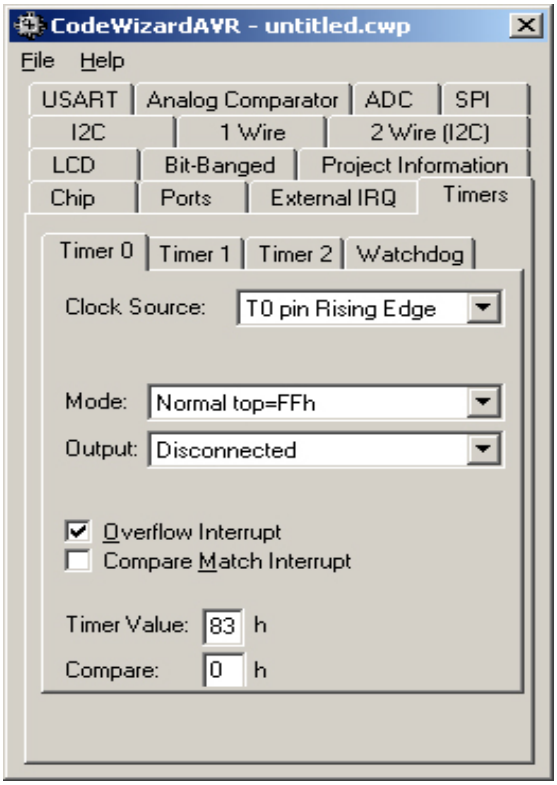

图 8-14 在程序生成向导中设置 T/C0

```
/*****************************************
File name           : Demo_8_1.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

#include <mega16.h>

// Timer 0 溢出中断服务
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
    TCNT0=0x83;                 // 重新设置 TCNT0 的初值
    PORTA.0 = ~PORTA.0;         // PA0 取反输出
}

void main(void)
{
    PORTA=0x01;
    DDRA=0x01;                  // 设置 PA0 输出方式
```

```
    PORTB=0x01;
    DDRB=0x00;                  // 设置 PB0(T0)为输入方式

    // T/C0 初始化
    TCCR0=0x07;                 // T/C0 工作于普通模式，T0 上升沿触发
    TCNT0=0x83;
    OCR0=0x00;

    TIMSK=0x01;                 // 允许 T0 溢出中断

    #asm("sei")                 // 开放全局中断

    while (1)
    {
        // Place your code here
    };
}
```

上面的程序，就是先利用 CVAVR 的程序生成向导功能进行配置，然后在它生成的程序框架基础上完成的。

在程序中，主程序对 T/C0 进行初始化后进入一个无限的循环。在 T/C0 的中断服务程序中，重新设置 TCNT0 的初值，并将 PA0 取反输出。这时 PA0 上可以获得 1Hz 的方波输出。

3） 思考与实践

- ✓ 为什么在中断服务程序中要重新设置 TCNT0 的初值?
- ✓ 如何计算 TCNT0 的初值，使得 PA0 输出 0.5Hz 的方波。

例 8.2 2N 分频系统设计二

1）硬件电路

同 2N 分频系统设计一。

2）软件设计

2N 分频系统设计一中使用了 T/C0 的普通模式,因此在中断服务程序中必须重新对 TCNT0 进行初始化。其实，更方便的方法是使用 T/C0 的 CTC 模式，利用 T/C0 的自动重装特性。当 T/C0 工作在 CTC 模式时，计数器 TCNT0 的值与 OCR0 的值比较，一旦相等，在下一次计数脉冲到来时，清零 TCNT0，并产生 T/C0 的比较匹配中断（参考图 8-13）。此时在比较匹配中断服务程序中改变 PA0 的输出即可。

```
/*****************************************
File name            : demo_8_2.c
Chip type            : ATmega16
Program type         : Application
Clock frequency      : 4.000000 MHz
Memory model         : Small
External SRAM size   : 0
Data Stack size      : 256
*****************************************/
```

```
#include <mega16.h>

// Timer 0 比较匹配中断服务
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
       PORTA.0=~PORTA.0;       // PA0 取反输出
}

void main(void)
{
    PORTA=0x01;
    DDRA=0x01;

    PORTB=0x01;
    DDRB=0x00;

    // T/C0 初始化
    TCCR0=0x0F;              // T/C0 工作于 CTC 模式，T0 上升沿触发
    TCNT0=0x00;
    OCR0=0x7C;               // 设置 OCR0 的比较值为 124（0x7C）

    TIMSK=0x02;              // 允许 T/C0 的比较匹配中断
    #asm("sei")              // 开放全局中断

    while (1)
    {
          // Place your code here
    };
}
```

3） 思考与实践

- ✓ 比较 demo_8_1.c 和 demo_8_2.c 中 T/C0 两种方式的特点。
- ✓ 如何利用 T/C0 实现 N 分频?

例 8.3 N 分频系统设计

1）硬件电路

同 2N 分频系统设计，但在 PA0 上得到 N 分频的输出。

2）软件设计

实际上，利用 T/C0 的比较匹配的特点，可以实现 N 分频的系统。

```
/*****************************************
File name           : demo_8_3.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
```

```
Data Stack size      : 256
*****************************************/

#include <mega16.h>

// Timer 0 溢出中断服务
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
       TCNT0=0xFB;             // 重新设置 TCNT0 的初值
       PORTA = ~PORTA;         // PA0 取反输出
}

// Timer 0 比较匹配中断服务
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
       PORTA = ~PORTA;         // PA0 取反输出
}

void main(void)
{
       PORTA=0x00;
       DDRA=0x01;

       PORTB=0x01;
       DDRB=0x00;

       // T/C0 初始化
       TCCR0=0x07;             // T/C0 工作于普通模式，T0 上升沿触发
       TCNT0=0xFB;
       OCR0=0xFD;              // 设置 OCR0 的比较值，>TCNT0 的初始值，<0xFF

       TIMSK=0x03;             // 允许 T/C0 的溢出和比较匹配中断
       #asm("sei")             // 开放全局中断

       while (1)
      {
           // Place your code here
      };
}
```

程序 demo_8_3.c 设置 T/C0 工作在普通模式，并结合比较匹配的特性，在比较匹配中断和溢出中断中都改变 PA0 的输出，在 PA0 上获得 5 分频的脉冲信号。

3） 思考与实践

- ✓ 请读者自己分析程序实现 N 分频的原理。
- ✓ 如果要实现 11 分频的输出，TCNT0 的初值应该如何计算，为什么同例 8.1 不同？

- ✓ 在 N 分频的系统中，OCR0 的值应该如何设置？
- ✓ 在 PA0 上输出的方波序列的占空比同例 8.1 和例 8.2 有何不同？
- ✓ 利用这个方法，能否在 PA0 上获得占空比可调的 PWM 波？

8.2.2 定时器应用设计

实际上不管定时计数器是作为计数器使用还是作为定时器使用，其根本的工作原理并没有改变，都是对一个脉冲系列信号进行计数。通常所谓的定时器，更多的情况是指其计数脉冲信号来自芯片本身的内部。由于内部的计数脉冲信号的频率（周期）是已知的或固定的，因此用户可以根据需要来设定计数器脉冲计数的个数，以获得一个等间隔的定时中断。利用定时中断，可以方便的实现系统定时访问外设或处理事物，以及获得更加准确的延时等等。

同其他一些单片机类似，AVR 的定时计数器的计数脉冲可以来自外部的引脚，也可以从内部系统时钟获得，但 AVR 的定时计数器在内部系统时钟和计数单元之间增加了一个可设置的预分频器，利用这个预分频器，定时计数器可以从内部系统中获得不同频率的计数脉冲信号。表 8.6 给出了系统时钟为 4MHz 时，ATmega16 芯片本身能够提供给 T/C0 的计数脉冲信号的最高计时精度和时宽范围。

表 8.1 T/C0 计时精度和时宽（系统时钟 4MHz）

分频系数	计时频率	最高计时精度 （TCNT0=255）	最宽时宽 （TCNT0=0）
1	4MHz	0.25us	64us
8	500KHz	2us	512us
64	62.5KHz	16us	4.096ms
256	15.625KHz	64us	16.384ms
1024	3906.25Hz	256us	65.536ms

从表中看出，在系统时钟为 4Mh 条件下，8 位的 T/C0 最高计时精度为 0.25us，而最长的时宽可达到 65.536ms。而如果使用 16 位的定时计数器 T/C1 时，不需要使用辅助软件计数器，就可以非常方便的设计一个时间长达 16.777216 秒（精度为 256us）的定时器，这是其它的 8 位单片机所做不到的。

AVR 单片机的每一个定时计数器都配备独立的、多达 10 位的预分频器，由软件设定分频系数，与 8/16 位定时计数器配合，可以提供多种档次的定时时间。使用时可选取最接近的定时档次，即选 8/16 位定时计数器与分频系数的最优组合，减少了定时误差。所以，AVR 定时计数器的显著特点之一是：高精度和宽时范围，使得用户应用起来更加灵活和方便。

例 8.4 采用 T/C0 硬件定时器的时钟系统

1) 硬件电路

在第六章中的例 6.5,“六位 LED 数码管动态扫描控制显示设计(一)”中,使用调用 CVAVR 中软件延时函数的方法给出了一个使用 6 个数码管组成时钟系统。采用软件延时，时钟是不准确的，因为一旦系统中使用了中断，就可能打断延时程序的执行，使延时时间发生变化。另外使用软件延时的方法，也降低了 MCU 的效率。

而在第七章中的例 7.2，“采用外部中断方式，用外部振荡源为基准的时钟系统”中，系统时钟的基准信号来自外部的标准方波信号源，这样尽管定时时间比采用软件延时方式要准确的多，但由于采用外部标准方波信号源而增加了系统的成本。

实际上更加方便和简单的方式是采用系统本身的时钟信号，配合 T/C0 产生时钟系统的定时信号。下面给出采用 T/C0 硬件定时器实现的时钟系统设计。时钟系统的硬件电路仍旧

与图 6-15 相同。

2) 软件设计

下面是一个采用 C 编写的系统源程序。

```
/*********************************************
File name           : demo_8_4.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/

#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

char   time[3];                  // 时、分、秒计数单元
char   dis_buff[6];              // 显示缓冲区，存放要显示的 6 个字符的段码值
int    time_counter;             // 中断次数计数单元
char   posit;
bit    point_on, time_1s_ok;

void display(void)               // 6 位 LED 数码管动态扫描函数
{
       PORTC = 0xff;
       PORTA = led_7[dis_buff[posit]];
       if (point_on && (posit==2||posit==4)) PORTA |= 0x80;
       PORTC = position[posit];
       if (++posit >=6 ) posit = 0;
}

// Timer 0 比较匹配中断服务
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
       display();                // 调用 LED 扫描显示
       if (++time_counter>=500)
       {
            time_counter = 0;
            time_1s_ok = 1;
       }
}

void time_to_disbuffer(void)     // 时钟时间送显示缓冲区函数
```

```
{
        char i,j=0;
        for (i=0;i<=2;i++)
        {
            dis_buff[j++] = time[i] % 10;
            dis_buff[j++] = time[i] / 10;
        }
}

void main(void)
{
        PORTA=0x00;                 // 显示控制 I/O 端口初始化
        DDRA=0xFF;
        PORTC=0x3F;
        DDRC=0x3F;
        // T/C0 初始化
        TCCR0=0x0B;                 // 内部时钟，64 分频（4M/64=62.5KHz），CTC 模式
        TCNT0=0x00;
        OCR0=0x7C;                  // OCR0 = 0x7C(124),(124+1)/62.5=2ms
        TIMSK=0x02;                 // 允许 T/C0 比较匹配中断

        time[2] = 23; time[1] = 58; time[0] = 55;     // 设时间初值 23:58:55
        posit = 0;
        time_to_disbuffer();

        #asm("sei")                 // 开放全局中断

        while (1)
        {
            if (time_1s_ok)                     // 1 秒到
            {
                time_1s_ok = 0;
                point_on = ~point_on;
                if (++time[0] >= 60)            // 以下时间调整
                {
                    time[0] = 0;
                    if (++time[1] >= 60)
                    {
                        time[1] = 0;
                        if (++time[2] >= 24) time[2] = 0;
                    }
                }
                time_to_disbuffer();            // 新调整好的时间送显示缓冲区
            }
```

```
        };
    }
```

该程序中的 LED 动态扫描，时间调整与例 7.2 相同，所不同的是使用了 T/C0 硬件定时。T/C0 工作在 CTC 模式，采用系统时钟经过 64 分频的信号作为计数器的计数脉冲。4M 系统时钟经过 64 分频后为 62.5KHz，周期为 16us。T/C0 的比较寄存器 OCR0 的值为 124（0x7C），因此 T/C0 每计数 125 次产生一次比较匹配中断，中断的间隔时间为 16*125=2ms。

在 T/C0 的比较匹配中断服务中，中断服务的内容同例 7.2，首先进行 LED 的扫描，即每位 LED 的扫描间隔（点亮时间）为 2 毫秒，然后中断次数计数器 time_counter 加 1。当 time_counter 加到 500，则置位秒标志 time_1s_ok，表示 1 秒时间到。

主程序中循环检测秒标志 time_1s_ok，当秒标志置位，则进行时间的调整，然后将新的时间值送到显示缓冲区中。

3) 思考与实践

该程序同第六章的例 6.5、第七章的例 7.2 有许多地方相同或类似，读者可以对三个程序进行全面的分析和比较，例如它们各自的优点和缺点以及 MCU 的利用率等。

例 8.5 基于 T/C2 硬件方式的 2N 分频方波发生器

1） 硬件电路

在本章的例 8.1 和 8.2 中，分别利用 T/C0 两种不同工作模式实现了 2N 分频的功能。在本例中，我们给出一个基于 ATmega16 本身系统时钟，利用 T/C2 构成的硬件方式的 2N 分频方波发生器的设计。

在这个设计中，T/C2 工作在 CTC 模式下，并利用其比较匹配输出的性能，直接由 OC2（PD7）输出 2N 分频的方波。具体工作原理参见本章对 T/C 结构中比较匹配输出的介绍和表 8.2。

在 CTC 模式下利用比较匹配输出单元产生波形输出时，应设置 OC2 的输出方式为触发方式（COM2[1:0]=1）。OC2 输出波形的最高频率为 $f_{OC2}=f_{clk\_I/O}/2$（OCR2=0x00）。其它频率输出由下式确定，式中 K 的取值为 1、8、32、64、128、256 或 1024。

$$f_{OC2}=\frac{f_{clk\_I/O}}{2K(1+OCR2)}$$

2） 软件设计

```
/*****************************************
File name           : demo_8_5.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

#include <mega16.h>

void main(void)
{
    DDRD=0x80;                  // 设置 PD7 为输出方式，即 OC2 方波输出脚
```

```
    // T/C2 初始化
    TCCR2=0x19;           //内部时钟，1 分频（4M），CTC 模式，OC2 的输出方式为触发方式
    TCNT2=0x00;
    OCR2=52;

    while (1)
    { };
}
```

上面的程序非常简单，仅在初始化时对 C/T2 进行设置，并设置 PD7 脚作为 OC2 的输出，然后进入一个无限的循环。该程序没有使用任何中断，在对 C/T2 做了必要的初始化设置后，就不再对 T/C2 做其它的处理，但程序运行后，在 PD7 的引脚上输出了一个 50%的 37.736KHz 的方波序列信号。因此，这种方式的波形发生器是基于 T/C2 硬件方式的，不需要软件的干预。通过选择不同的计数频率的时钟信号，以及配合 OCR2 不同的值就可以获得更多频率的方波输出信号。

3） 思考与实践

- ✓ 将该程序同例 8.1 和 8.2 进行比较，读者可以对三个程序进行全面的分析，例如它们各自的优点和缺点以及 MCU 的利用率等。
- ✓ 如果系统需要一个频率为 50Hz 左右 50%占空比的方波信号，那么应该 T/C2、OCR2 如何设置和计算？理论上的输出频率是多少？与 50Hz 的误差是多少（假定系统时钟频率为 4M）。

8.3 PWM脉宽调制波的产生和应用

8.3.1 PWM脉宽调制波

PWM 是脉冲宽度调制的简称。实际上，PWM 波也是一个连续的方波，但在一个周期中，其高电平和低电平的占空比是不同的。一个典型 PWM 的波形如图 8-15 所示。

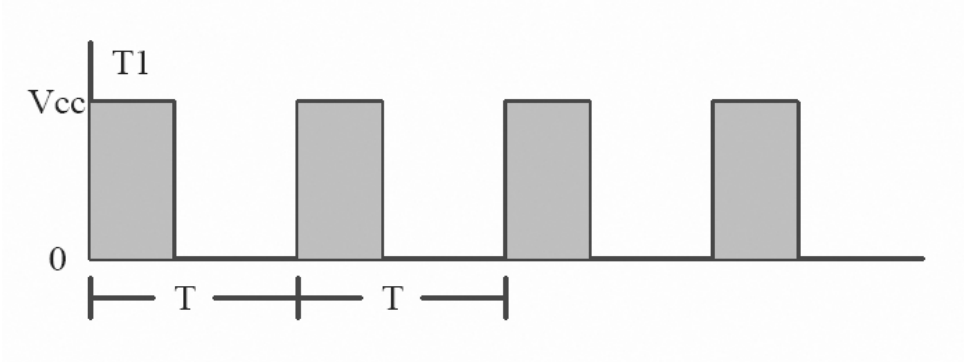

图 8-15 典型的 PWM 波

在图中，T 是 PWM 波的周期，T1 是高电平的宽度，Vcc 是高电平值。当该 PWM 波通过一个积分器后（低通滤波器）后，我们可以得到其输出的平均电压为：

$$V = \frac{Vcc * T1}{T}$$

式中，T1/T 称为 PWM 波的占空比。控制调节和改变 T1 的宽度，即改变 PWM 的占空比，就可以的到不同的平均电压输出。因此在实际应用中，常利用 PWM 波的输出，实现 D/A 转换，调节电压或电流控制改变马达的转速，实现变频控制等功能。

一个 PWM 方波的参数有频率、占空比和相位（在一个 PWN 周期中，高低电平转换的起始时间），其中频率和占空比为主要的参数。图 8-16 为 3 个占空比都为 2/3 的 PWM 波形，尽管他们输出的平均电压是一样的，但其中（b）的频率比（a）高一倍，相位相同；而（c）与(a)的频率相同，但相位不同。

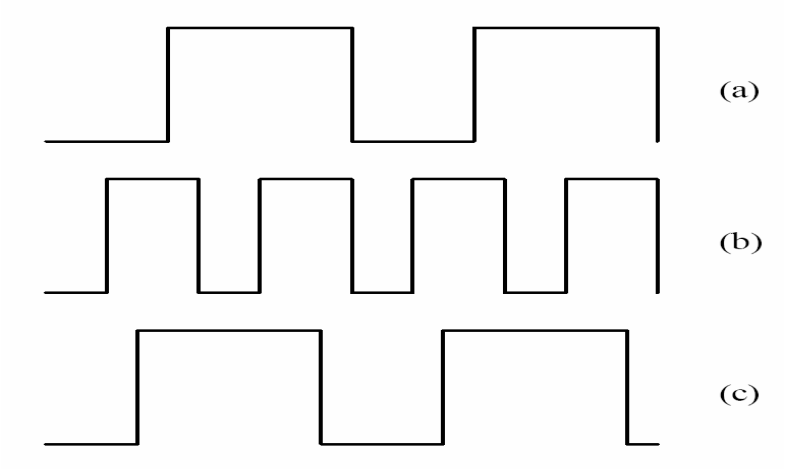

图 5.9 占空比相同，频率与相位不同的 PWM

在实际应用中，除了要考虑如何正确的控制和调整 PWM 波的占空比，获得达到要求的平均电压的输出外，还需要综合的考虑 PWM 的周期、PWM 波占空比调节的精度（通常用 BIT 位表示）、积分器的设计等。而且这些因素相互之间也是互相牵连的。根据 PWM 的特点，在使用 AVR 定时计数器设计输出 PWM 时应注意以下几点：

- ✓ 首先应根据实际的情况，确定需要输出的 PWM 波的频率范围。这个频率与控制的对象有关。如输出的 PWM 波用于控制灯的亮度，由于人眼不能分辨 42Hz 以上的频率，所以 PWM 的频率应高于 42Hz，否则人眼会察觉到灯的闪烁。PWM 波的频率越高，经过积分器输出的电压也越平滑。
- ✓ 同时还要考虑占空比的调节精度。同样，PWM 波占空比的调节精度越高，经过积分器输出的电压也越平滑。但占空比的调节精度与 PWM 波的频率是一对矛盾，在相同的系统时钟频率时，提高占空比的调节精度，将导致 PWM 波频率的降低。
- ✓ 由于 PWM 波的本身还是数字脉冲波，其中含有大量丰富的高频成分，因此在实际使用中，还需要一个好的积分器电路，如采用有源低通滤波器，或多阶滤波器等，能将高频成分有效的去除掉，从而获得比较好的模拟变化信号。

ATmega16 的 T/C0 和 T/C2 都具备产生 PWM 波的功能，由于它们计数器的长度为 8 位，所以是固定 8 位精度的 PWM 波发生器。即 PWM 的调节精度为 8 位，因此 PWM 波的周期只能取决于系统时钟和分频系数（即作为计数器计数脉冲的频率）。T/C0、T/C2 的工作模式中有快速 PWM 模式（WGMn[1:0]=3）和相位可调 PWM 模式（WGMn[1:0]=1）两种（参见图 8-8 和图 8-9）。

快速 PWM 模式可以的到比较高频率、相位固定的 PWM 输出，适合一些要求输出 PWM 频率较高，频率相位固定的应用。快速 PWM 模式中，计数器仅工作在单程正向计数方式，计数器的上限值决定 PWM 的频率，而比较匹配寄存器 OCRn 的值决定了占空比的大小。快速 PWM 频率的计算公式为：

PWM 频率 = 系统时钟频率/（分频系数 * 256）

相位可调 PWM 模式的输出频率比较低，此时计数器工作在双向计数方式。同样计数器的上限值决定了 PWM 的频率，比较匹配寄存器的值决定了占空比的大小。PWM 频率的计算公式为：

PWM 频率 = 系统时钟频率/（分频系数 * 510）

相位调整 PWM 模式适合在要求输出 PWM 频率较低，频率固定应用中。由于计数器工作在双向模式，当调整占空比时（改变 OCR0 的值），PWM 的相位也相应的跟着变化（Phase Correct），但由于其产生的 PWM 波形是对称的，更加适合在电机控制中使用。

T/C0、T/C2 两种 PWM 模式都是固定 8 位的 PWM，计数器 TCNTn 的上限值为固定的 0xFF（8 位 T/C），而比较匹配寄存器 OCRn 的值与计数器上限值之比即为占空比。

8.3.2 基于比较匹配输出的脉冲宽度调制PWM

利用 ATmega16 定时计数器的 PWM 模式，与比较匹配寄存器相配合，能直接生成占空比可变的方波信号，即脉冲宽度调制输出 PWM 信号。快速 PWM 模式工作的基本原理是：定时计数器在计数过程中，内部硬件电路会将计数值（TCNTn）和比较寄存器（OCRn）中的值进行比较，当两个值相匹配（相等）时，能自动置位（清零）一个固定引脚的输出电平（OCnx），而当计数器的值到达最大值时，则自动将该引脚的输出电平（OCnx）清零（置位）（参考图 8-8）。因此，在程序中改变比较寄存器中的值（通常在溢出中断服务中），定时计数器就能自动产生不同占空比的方波（PWM）信号输出了。

例 8.6 利用 T/C0 的 PWM 功能产生正弦波

1） 硬件电路

在本例中将 T/C0 的 PWM 方式来产生一个 1KHz 左右的正弦波，正弦波的幅度为 0-Vcc。

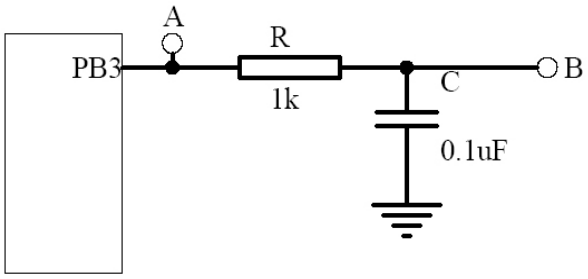

图 8-16 采用 T/C0 的 PWM 功能产生正弦波

图 8-16 为电原理图。PB3 为 ATmega16 的 T/C0 匹配输出脚 OC0，该脚上的 PWM 波通过由电阻 R 和电容 C 构成的简单滤波电路滤掉高频进行平滑后，在 B 点获得模拟的正弦波信号。

2） 软件设计

首先按照下面的公式建立一个正弦波样本表，样本表将一个正弦波周期分为 128 个点，每点按 8 位量化（1 对应最低幅度，255 对应最高幅值 Vcc）：

$$f(x) = 128 + 127 * \sin(2\pi x/127) \quad x \in [0 \cdots 127]$$

如果在一个正弦波周期中采用 128 个样点，那么对应 1KHz 的正弦波 PWM 的频率为 128KHz。实际上，根据采样频率至少为信号频率的 2 倍的采样定理来计算，PWM 的频率的理论值应大于 2KHz。考虑到尽量提高 PWM 的输出精度，实际设计中使用的 PWM 频率为 16KHz，即一个正弦波周期中输出 16 个正弦波样本值。这意味着在 128 点的正弦波样本表中，要每隔 8 点取出一点作为 PWM 的输出。

程序中使用 ATmega16 的 8 位 T/C0，工作模式为快速 PWM 模式输出，系统时钟为 4MHz，分频系数为 1，其可以产生 PWM 波的频率为：4000000Hz / 256 = 15625Hz。每 16 次输出构成一个周期正弦波，那么正弦波的频率为 15625/16=976.56Hz。PWM 由 OC0（PB3）引脚输出。参考程序如下。

```
/*********************************************
File name            : demo_8_6.c
```

```
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

#include <mega16.h>

flash char auc_SinParam[128] = {
128,134,140,147,153,159,165,171,177,182,188,193,199,204,209,213,
218,222,226,230,234,237,240,243,245,248,250,251,253,254,254,255,
255,255,254,254,253,251,250,248,245,243,240,237,234,230,226,222,
218,213,209,204,199,193,188,182,177,171,165,159,153,147,140,134,
128,122,116,109,103,97,91,85,79,74,68,63,57,52,47,43,
38,34,30,26,22,19,16,13,11,8,6,5,3,2,2,1,
1,1,2,2,3,5,6,8,11,13,16,19,22,26,30,34,
38,43,47,52,57,63,68,74,79,85,91,97,103,109,116,122}      // 128 点正弦波样本值

char x_SW = 8,X_LUT = 0;

// T/C0 溢出中断服务
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
      X_LUT += x_SW;                  // 新样点指针
      if (X_LUT > 127) X_LUT -= 128;  // 样点指针调整
      OCR0 = auc_SinParam[X_LUT];     // 取样点指针到比较匹配寄存器
      // OCR0+=1;
}

void main(void)
{
      DDRB=0x08;                      // PB3 输出方式，作为 OC0 输出 PWM 波
      // Timer/Counter 0 initialization
      // Clock source: System Clock
      // Clock value: 4000.000 kHz
      // Mode: Fast PWM top=FFh
      // OC0 output: Non-Inverted PWM
      TCCR0=0x69;
      OCR0=128;

      TIMSK=0x01;                     // 允许 T/C0 溢出中断
      #asm("sei")                     // 开放全局中断
```

```
        while (1)
        {};
    }
```

程序中，在每次的计数器溢出中断的服务中取出一个正弦波的样点值到比较匹配寄存器中，用于调整下一个 PWM 的脉冲宽度，这样在 PB3 引脚上输出了按正弦波调制的 PWM 方波。当 PB3 的输出通过低通滤波器后，便得到一个 976.56Hz 的正弦波了。

如要得到更精确的 1KHz 的正弦波，可使用 ATmega16 的 T/C1，选择工作模式 10，设置 ICR1=250 为计数器的上限值（将在以后的章节中给出）。

图 8-17 为程序运行后使用示波器在 B 点测量的实际波形。B 点的波形轮廓非常接近正弦波，频率为 976.7Hz，幅度在 0-5v 之间。由于滤波电路由非常简单的 RC 电路构成，因此还能够看出在正弦波的轮廓上的高频成分。如果用示波器在 A 点测量的话，可以观察到一个占空比变化的序列方波。

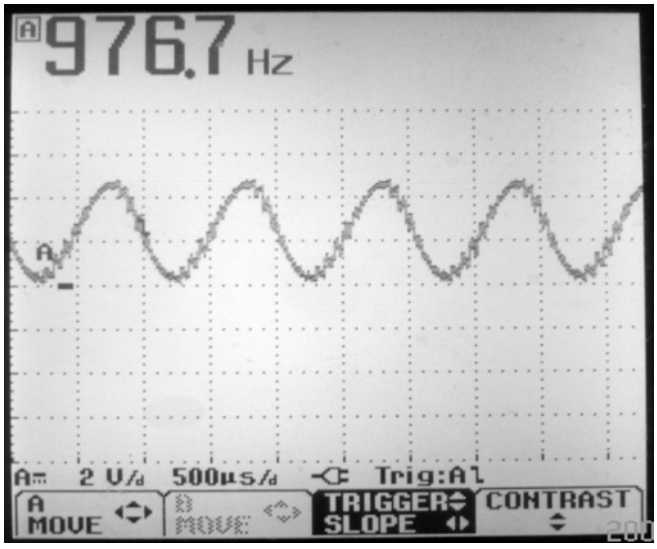

图 8-17 实际测量到的正弦波输出波形

3） 思考与实践

- ✓ 根据 T/C0 的 PWM 模式，再结合本例，详细描述快速 PWM 的工作原理。
- ✓ 减小或增加程序中变量 x_SW 的值时，B 点输出的波形有何变化？如何进行理论的计算？
- ✓ 如果将 T/C0 中断服务程序中的语句清除到，用 OCR0+=1 代替时，分析 B 点输出的波形，以及频率，并用示波器观察。

8.4 16 位定时计数器T/C1 的应用

ATmega16 的 T/C1 是一个 16 位的多功能定时计数器，图 8-18 为该 16 位定时计数器的结构框图。其主要特点有：

- ✓ 真正的 16 位设计。
- ✓ 2 个独立的输出比较匹配单元。
- ✓ 双缓冲输出比较寄存器。

- ✓ 一个输入捕捉单元。
- ✓ 输入捕捉躁声抑制。
- ✓ 比较匹配时清零计数器（自动重装特性，Auto Reload）。
- ✓ 可产生无输出抖动（glitch-free）的，相位可调的脉宽调制（PWM）信号输出。
- ✓ 周期可调的 PWM 波形输出。
- ✓ 频率发生器。
- ✓ 外部事件计数器。
- ✓ 带 10 位的时钟预分频器。
- ✓ 4 个独立的中断源（TOV1、OCF1A、OCF1B、ICF1）。

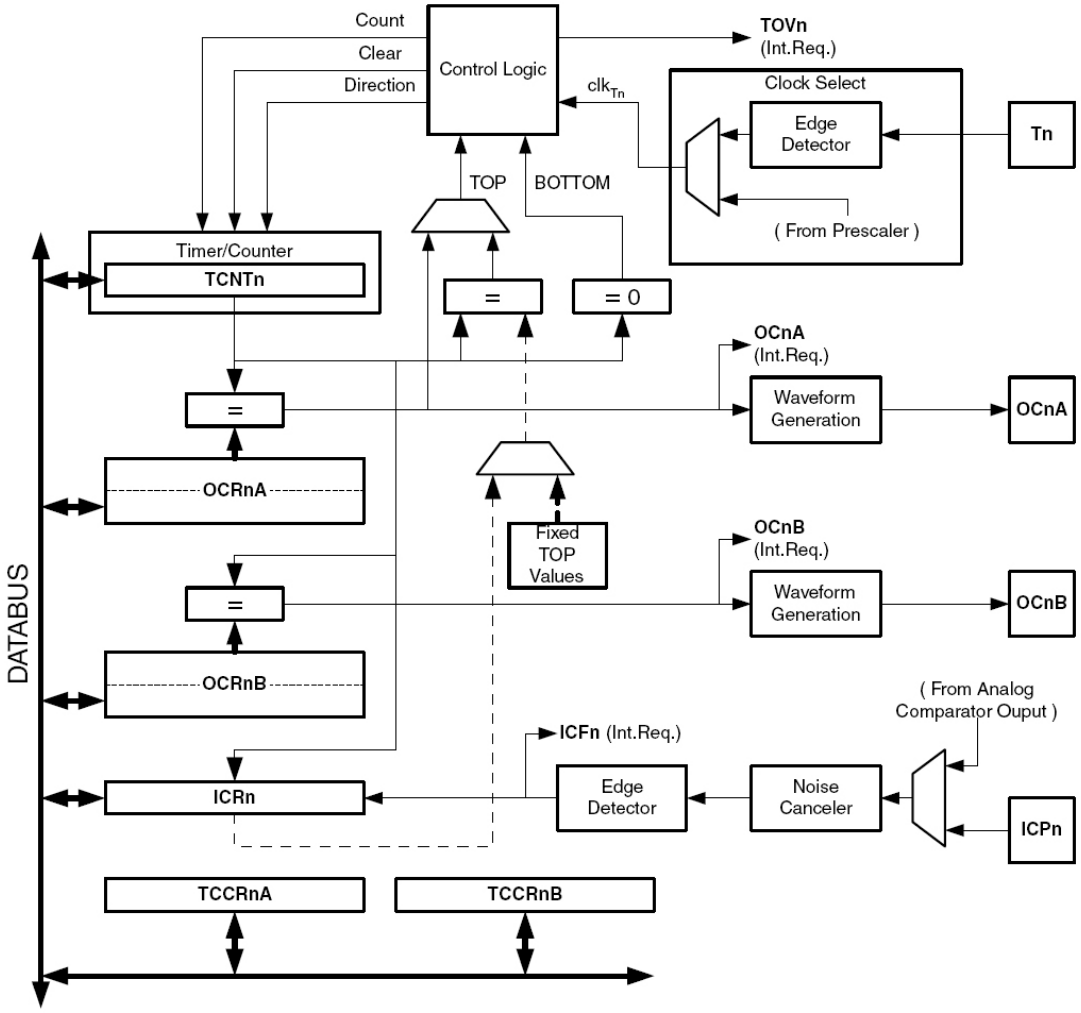

图 8-18 T/C1 结构图（图中 n 为 1）

图中给出了 MCU 可以操作的寄存器以及相关的标志位，其中计数器寄存器 TCNT1、输出比较寄存器 OCR1A、OCR1B 和输入捕捉寄存器 ICR1 都是 16 位的寄存器。T/C1 所有的中断请求信号 TOV1、OC1A、OC1B、ICF1 可以在定时计数器中断标志寄存器 TIFR 找到，而在定时器中断屏蔽寄存器 TIMSK 中，可以找到与它们对应的 4 个相互独立的中断屏蔽控制位 TOIE1、OCIE1A、OCIE1B 和 TICIE1。TCCR1A、TCCR1B 为 2 个 8 位寄存器，是 T/C1 的控制寄存器。

T/C1 的时钟源的选择由 T/C1 的控制寄存器 TCCR1B 中的 3 个标志位 CS1[2:0]确定，共有 8 种选择。其中包括无时钟源（停止计数），外部引脚 T1 的上升沿或下降沿，以及内部系

统时钟经过一个 10 位预定比例分频器分频的 5 种频率的时钟信号（1/1、1/8、1/64、1/256、1/1024）。

T/C1 基本的工作原理和功能与 8 位定时计数器相同，常规的使用方法也是类同的。但与 8 位的 T/C0、T/C2 相比，T/C1 不仅位数增加到 16 位，其功能也更加强大。由于篇幅有限，本节着重介绍 T/C1 一些增强的功能和基本应用。

8.4.1 16 位T/C1 增强功能介绍

与 8 位 T/C0、T/C2 相比，T/C1 的功能增强主要表现在以下几个方面。

1．16 位的计数器

由于 T/C1 是 16 位的计数器，因此它的计数宽度、计时长度大大增加，配合一个独立的 10 位预定比例分频器，在系统时钟为 4Mh 条件下，16 位的 T/C1 最高计时精度为 0.25us，而最长的时宽可达到 16.777216 秒（精度为 256us），这是其它的 8 位单片机所做不到的。

需要注意的是，AVR 的内部有许多 16 位的寄存器，这些寄存器都是由两个 8 位的寄存器组成的。如 16 位的寄存器 TCNT1 实际由 2 个 8 位寄存器 TCNT1H、TCNT1L 组成的。对这些 16 位寄存器的读写操作需要遵循以下特定的步骤。

2．16 位寄存器的读写操作步骤

由于 AVR 的内部数据总线为 8 位，因此读写 16 位的寄存器需要分两次操作。为了能够同步读写 16 位寄存器，每一个 16 位寄存器分别配有一个 8 位的临时辅助寄存器(Temporary Register)，用于保存 16 位寄存器的高 8 位数据。要同步读写这些 16 位的寄存器，读写操作应遵循以下特定的步骤：

- 16 位寄存器的读操作

当 MCU 读取 16 位寄存器的低字节（低 8 位）时，16 位寄存器低字节内容被送到 MCU，而高字节（高 8 位）内容在读低字节操作的同时被置于临时辅助（TEMP）寄存器中。当 MCU 读取高字节时，读到的是 TEMP 寄存器中的内容。因此，要同步读取 16 位寄存器中的数据，应先读取该寄存器的低位字节，再立即读取其高位字节。

- 16 位寄存器的写入操作

当 MCU 写入数据到 16 位寄存器的高位字节时，数据是写入到 TEMP 寄存器中。当 MCU 写入数据到 16 位寄存器的低位字节时，写入的 8 位数据与 TEMP 寄存器中的 8 位数据组合成一个 16 位数据，同步写入到 16 位寄存器中。因此，要同步写 16 位寄存器时，应先写入该寄存器的高位字节，再立即写入它的低位字节。

用户编写汇编程序时，如要对 16 位寄存器进行读写操作，应遵循以上特定的步骤。此外，在对 16 位寄存器操作时，最好将中断响应屏蔽，防止在主程序读写 16 位寄存器的两条指令之间插入一个含有对该寄存器操作的中断服务。如果这种情况发生，那么中断返回后，寄存器中的内容已经改变，会造成主程序中对 16 位寄存器的读写失误。下面是读写 16 位寄存器的汇编子程序示例。

```
;汇编代码：TIME16_Read_Write_TCNT1:
;保存寄存器 SREG
in r18,SREG
;禁止中断
cli
;读 TCNT1 到 r17:r16
in r16,TCNT1L
in r17,TCNT1H
;置 TCNT1 为 0x01FF
```

```
ldi r17,0x01
ldi r16,0xFF
out TCNT1H,r17
out TCNT1l,r16
;恢复寄存器 SREG
out SREG,r18
ret
```

采用C等高级语言编写程序则可以直接对16位的寄存器进行操作，因为这些高级语言的编译系统会根据16位寄存器的操作步骤生成正确的执行代码。

3. 更加强大和完善的 PWM 功能

T/C1 配备了 2 个比较匹配输出单元 OC1A、OC1B 和比较匹配寄存器 OCR1A、OCR1B。同时它的 PWM 模式，有多种不同的计数器上限(TOP)值可供选择，因此 T/C1 的 PWM 功能具备以下特点：

- 可产生频率、相位均可调整的 PWM 波。

T/C1 有 15 种工作模式，除了常规的计数、CTC 模式外，还可以产生频率可调、相位可调、频率相位均可调的多种形式的 PWM 波。其中频率可调的 PWM 波利用 8 位定时计数器是不能实现的。T/C1 的频率调整范围可以达到 16 位的精度，它是通过改变计数器的上限值实现的。

- 可同时产生 2 路不同占空比的 PWM 波

由于 T/C1 配备了 2 个比较匹配输出单元 OC1A、OC1B 和比较匹配寄存器 OCR1A、OCR1B，因此使用一个计数器就可以得到相同频率，不同占空比的 2 路 PWM 输出。2 路 PWM 波的占空比的确定和调整分别由寄存器 OCR1A、OCR1B 确定，分别在 OC1A、OC1B 上输出。

4. 输入捕捉功能

T/C1 的输入捕捉功能是 AVR 定时计数器的另一个非常有特点的功能。T/C1 的输入捕捉单元（如图 8-19 所示）可应用于精确捕捉一个外部事件的发生，记录事件发生的时间印记（Time-stamp)。捕捉外部事件发生的触发信号由引脚 ICP1 输入，或模拟比较器的 ACO 单元的输出信号也可作为外部事件捕获的触发信号。

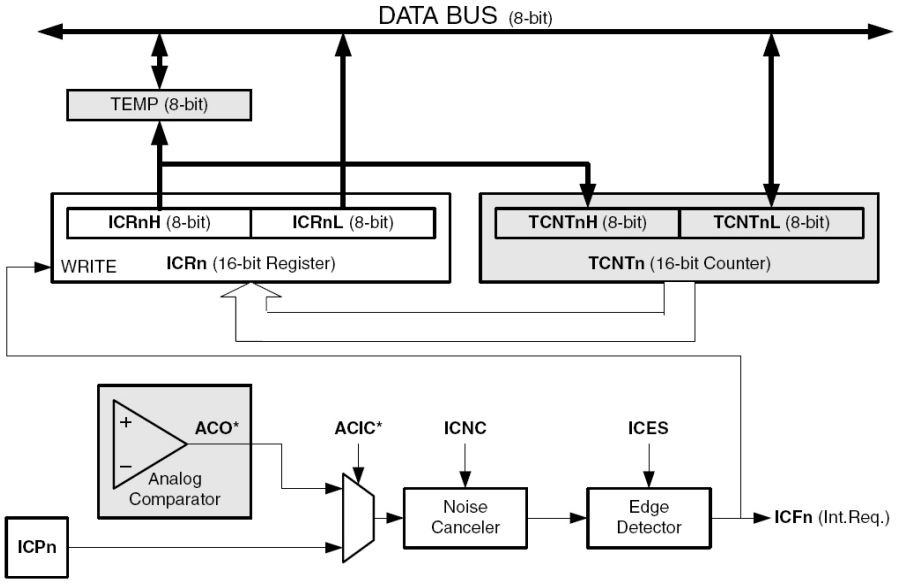

图 8-19　T/C1 的外部事件输入捕捉单元（n 为 1）

当一个输入捕捉事件发生，如外部引脚ICP1上的逻辑电平变化时，或者模拟比较器输出电平变化（事件发生）时，此时T/C1的计数器TCNT1中的计数值被写入输入捕捉寄存器ICR1中，并置位输入捕获标志位ICF1，并产生中断申请。输入捕捉功能可用于频率和周期的精确测量。

置位标志位 ICNC 将使能对输入捕捉触发信号的噪声抑制功能。噪声抑制电路是一个数字滤波器，它对输入触发信号进行 4 次采样，当 4 次采样值相等才确认此触发信号。因此使能输入捕捉触发信号的噪声抑制功能可以对输入的触发信号的噪声实现抑制，但确认触发信号比真实的触发信号延时了 4 个系统时钟周期。噪声抑制功能是通过寄存器 TCCR1B 中的输入捕捉噪声抑制位(ICNC)来使能。如果使能了输入噪声抑制功能，捕捉输入信号的变化到 ICR1 寄存器的更新延迟四个时钟周期。噪声抑制功能使用的系统时钟，与预分频器无关。

输入捕捉信号触发方式的选择由寄存器 TCCR1B 中的第 6 位 ICES1 决定。当 ICES1 设置为“0”时，输入信号的下降沿将触发输入捕捉动作；当 ICES1 为“1”时，输入信号的上升沿将触发输入捕捉动作。一旦一个输入捕捉信号的逻辑电平变化触发了输入捕捉动作时，T/C1 计数器 TCNT1 中的计数值被写入输入捕获寄存器 ICR1 中，并置位输入捕获标志位 ICF1，申请中断处理。

寄存器 ICR1 由两个 8 位寄存器 ICR1H、ICR1L 组成，当 T/C1 工作在输入捕捉模式时，一旦外部引脚 ICP1 或模拟比较器有输入捕捉触发信号产生，计数器的 TCNT1 中的计数值写入寄存器 ICR1 中。T/C1 工作在其它模式时，如 PWM 模式，ICR1 的设定值可作为计数器计数上限（TOP）值。此时 ICP1 引脚与计数器脱离，将禁止输入捕获功能。

输入捕捉事件发生后产生的中断申请标志 ICF1，以及相应的中断屏蔽控制位 TICIE1 可以在定时计数器中断标志寄存器 TIFR 和定时器中断屏蔽寄存器 TIMSK 中找到。

在本节中仅对 ATmega16 的 16 位定时计数器的结构和基本功能做了简单的介绍。读者可以结合对本章前几节 8 位定时计数器学习的基础上，来体会和掌握 16 位定时计数器的原理和使用。在本书的匹配的光盘中，给出了使用定时计数器实现双音频拨号的应用设计参考(avr_app_314.pdf)，读者可以从中学习到如何更好的设计和使用 PWM 的功能。在下面一节中，将会给出一个 T/C1 的设计应用例子。而在后面的第 11 章中，还将对 T/C1 输入捕捉功能的设计使用进行详细的介绍。

8.4.2 16 位T/C1 应用示例

例 8.7 利用 T/C1 设计实现的单音手机音乐播放器

在实际的应用中，经常需要利用单片机系统产生各种音乐用于报警和提示等，如手机的来电铃声，儿童玩具，时钟的音乐报时等。原则上讲，用单片机产生各种音乐发声的原理很简单，就是由 I/O 引脚输出不同频率的脉冲信号，再将信号放大，推动发声器件发声（这里是指在要求不高的情况下，用不同频率的脉冲方波替代正弦波）。

在编写程序前，通常要先知道各不同的音符所对应的振荡频率，也就是确定各音符所对应的脉冲输出频率值。另外就是该音符发生的长度(节拍)值。接下来是根据所要产生曲谱的音调和节拍，建立相对应发声数据组（参考手机上自编音乐的功能）。然后才能编写发生控制程序。

表 8-2 是一个简单的 8 位音符的频率(周期)对应表。在表中，低音音符“1”的频率为 523Hz，既 1 秒钟脉冲为 523 个，周期为 1/523=1912us，半周期为 1912us。假定把音乐的 1 个节拍的时间长度为 0.4 秒，那么 1/4 节拍的时间长度为 0.1 秒。

因此，如果要发出 1/4 节拍长度的低音“1”，I/O 输出的脉冲频率应该为 523Hz，输出脉冲的个数为 52.3 个。

我们以 1/4 拍为 1 个基本节拍单位，则 2 个基本单位表示 1/2 拍，4 个基本单位为 1 拍，

依次类推。

注意，表中最后的两行为音符所对应的半周期数和 1/4 基本节拍时间单位中输出半周期的个数。做这样的换算处理，并取整数，主要是方便在程序中使用。

表 8-2 8 位音符的频率(周期)对应表

音符	1	2	3	4	5	6	7	i(高音)
数字表示	1	2	3	4	5	6	7	8
频率(个)	523	578	659	698	784	880	988	1046
周期 us	1912	1730.1	1517.5	1432.7	1275.5	1136.4	1012.1	956
1/4 节拍	52.3	57.8	65.9	69.8	78.4	88	98.8	104.6
*半周期 us	956	865	759	716	638	568	506	470
*1/4 节拍	105	116	132	140	157	176	198	209

根据表 8-2，再按儿童歌曲“我爱北京天安门”的乐谱编制出发音数据组，每一个音符的发生用一组数据来表示。其中第一个数据表示发音的音符，第二个数据是节拍基本单位的倍率值，它表示音符的发音长度。如乐谱中第一个音符“5”为 1/2 拍，则其对应的数据表示为（5，2）；第 2 个音符是 1/2 拍的高音“i”，其数据表示为（8，2）；最后一个音符为 2 拍的“5”，其数据表示为（5，8）。

5 i 5 4 | 3 2 1 | 1 1 2 3 | 3 1 3 4 | 5- |

5,2 8,2 5,2 4,2 3,2 2,2 1,4 1,2 1,2 2,2 3,2 3,3 1,2 3,2 4,2 5,8

1） 硬件电路

图 8-20 为电原理图。PD5 为 ATmega16 的 T/C1 匹配输出脚 OC1A。编写程序在该脚上输出脉冲波，输出经三极管放大驱动蜂鸣器发声。图中的发声部分位于 AVR-51 多功能实验板的 M 区。BZ 为一个无源蜂鸣器，作为发声器件（有源蜂鸣器的发声频率固定，不能使用），I/O 口的脉冲输出经三极管放大驱动蜂鸣器发声。

，

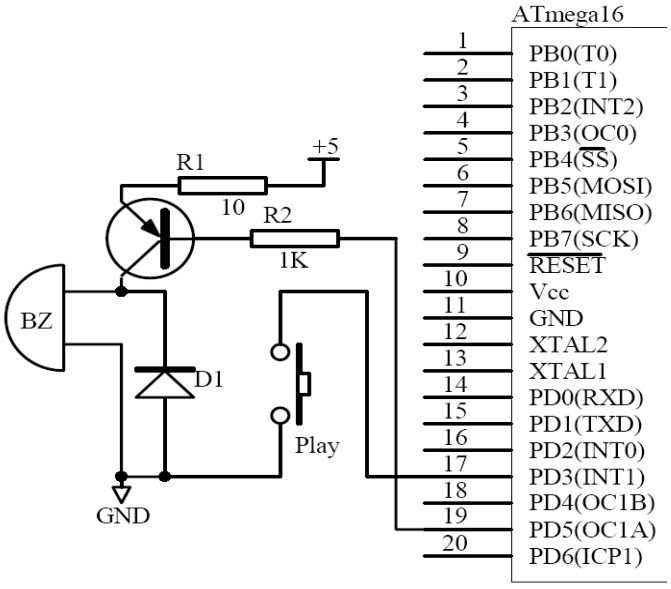

图 8-20 采用 T/C0 的 PWM 功能产生正弦波

图中在 PD3 上接了一个按键，作为播放音乐的控制，每按一下，播放一遍乐曲。电阻 R1、R2 起到限流和保护作用，电阻 R1 的值应在 5-10 欧姆之间，太大会降低 BZ 发声功率。D1 为反峰保护二极管，因为三极管驱动的为 BZ 内部的电感线圈。

2） 软件设计

在一些教课书或参考书中都可看到类似的设计例子，不足的是，这些示例往往是一个独立的例子，或采用软件定时和延时的方法，或使用多个硬件功能部件，如 2 个定时器等。本例给出的设计方法充分利用了 AVR 的 T/C1 的特点，仅使用一个定时计数器，配合它的比较匹配功能，结合中断实现了不需要前台管理的音乐全程播放功能。而且这种功能的设计实现，可以非常方便的组合在一个复杂系统中使用。参考例程如下。

```
/*****************************************************
File name           : demo_8_7.c
Chip type           : ATmega16L
Program type        : Application
Clock frequency     : 1.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************************/
#include <mega16.h>

flash unsigned int t[9] = {0,956,865,759,716,638,568,506,470};
flash unsigned char d[9] = {0,105,116,132,140,157,176,198,209};
#define Max_note    32
flash unsigned char music[Max_note] =
             {5,2,8,2,5,2,4,2,3,2,2,2,1,4,1,2,1,2,2,2,3,2,3,2,1,2,3,2,4,2,5,8};

unsigned char note_n;
unsigned int int_n;
bit play_on;

interrupt [EXT_INT1] void ext_int1_isr(void)  // INT1 中断服务程序
{
    if (!play_on)
    {
        TCCR1B = 0x09;                        // 启动 T/C1，播放音乐
    }
}

interrupt [TIM1_COMPA] void timer1_compa_isr(void) // T/C1 比较匹配中断服务
{
    if (!play_on)
    {
        note_n = 0;
        int_n = 1;
```

```
        play_on = 1;              // 开始重头播放音乐
    }
    else
    {                             // 播放一个音符
        if (--int_n == 0)
        {                                       // 一个音符播放完成
            TCCR1B = 0x08;                      // 停止 T/C1 工作
            if (note_n < Max_note)
            {                                   // 音乐未播完
                OCR1A = t[music[note_n]];       // 取下一个音符
                int_n = d[music[note_n]];       // 取该音符的基本节拍单位
                note_n++;
                int_n = int_n * music[note_n];  // 计算该音符的节拍值
                note_n++;
                TCCR1B = 0x09;                  // 启动 T/C1，播放 1 个音符
            }
            else
                play_on = 0;                    // 整个音乐播放完成
        }
    }
}

void main(void)
{
    PORTD=0x08;         // PD5 匹配输出方式，脉冲输出，驱动发声；
    DDRD=0x20;          // PD3 按键(INT1)输入口，输入方式，且上拉电阻有效

     TCCR1A=0x40;           // T/C1 初始化代码。T/C1 工作于 CTC 方式，
     TCCR1B=0x08;           // OCR1A 为比较寄存器，OC1A 为触发取反方式
     TIMSK=0x10;            // 允许比较匹配 OC1A 中断

     GICR|=0x80;            // 外部中断初始化代码
     MCUCR=0x08;            // 允许 INT1 中断
     MCUCSR=0x00;           // INT1 中断采用下降沿出发方式
     GIFR=0x80;
     #asm("sei")            // 开放全局中断标志

     while (1)              // 主程序
    { }
}
```

3） 软件设计说明

系统软件采用 T/C1 的 CTC 比较匹配模式。系统时钟为 1MHz，则一个时钟周期为 1us。寄存器 OCRA1 中为音符的半周期值，所以 2 次匹配中断的匹配比较输出在 OC1A 上输出一个

完整的方波（注意，OC1A 为触发取反方式）。变量 int_n 记录了中断的次数，用于控制该音符脉冲输出的个数，实际上就是音符输出的时间，代表了节拍的长度。

在 T/C1 的 OC1A 中断服务中会自动判别整个音乐是否全部播放完成，如果音乐没有全部播完，将取出下一个音符的音调和节拍，继续播放。数组 t[9]、d[9]为音符对应的半周期的 us 数和基本 1/4 节拍应产生的半周期数。music 数组中为整个音乐的发声数据。

程序中巧妙的通过利用设置选定 T/C1 的计数脉冲源的方法来启动和停止 T/C1 的工作。设置 T/C1 无计数脉冲源输入(TCCR1B = 0x08)就是停止了 T/C1 的工作，不会产生脉冲输出。而 TCCR1B = 0x09 是设置系统时钟 1 分频后（周期 1us）作为计数脉冲输入，这样就启动了 T/C1 工作计数，一旦计数值与 OCR1A 相同，不仅在 OC1A 脚上产生电平的变化（脉冲输出发音），而且也产生了中断申请。

在这个例程中，主程序没有做任何的工作，一旦 play 按键按下，产生 INT1 中断。在 INT1 中断中，初始化音乐的播放条件并启动 T/C1 工作，然后音乐播放的一切控制由 T/C1 的 OC1A 中断处理了。这种程序设计思想是 T/C1 和中断巧妙结合使用的巧妙例子，它大大提高了 MCU 的效率。

在本章中，我们仅对 ATmega16 定时计数器的结构、工作原理和基本功能应用做了稍微详细的介绍。希望读者能通过本章的学习和实践，全面体会和熟练掌握定时计数器的原理，为以后灵活多变的应用设计打好基础。在以后的章节中，将会看到定时计数器与中断配合使用，是一个重要的功能部件和设计方法被经常使用的。

思考与练习

1. 简述定时计数器的基本工作原理，它是如何实现定时器和计数器功能的？
2. AVR 的 8 位定时计数器有几种工作方式？每种工作方式的基本用途是什么？
3. AVR 定时计数器的计数脉冲源有那些种类和方式？预分频器的作用是什么？
4. AVR 定时计数器配备的比较寄存器的作用是什么？由于它的存在，使定时计数器的基本功能得到哪些扩展？
5. 当定时计数器工作在普通模式和 CTC 模式时，都可以产生一个固定的定时中断。如果要求精确的定时中断，采用那种模式比较好？为什么？
6. 本章例程 8.4 与第六章的例 6.5、第七章的例 7.2 实现了相同的功能，在结构上有许多地方相同或类似，请对三个程序进行全面的分析和比较，说明那种方式最好，为什么？
7. 认真分析和理解本章中所给出的所有示例，并进行实践，思考和回答所有的问题。
8. 参考 demo_8_6.c，利用 T/C0 的 PWM 方式，设计能产生频率为 500Hz 左右，幅值为 0～5V 的三角波发生器。
9. 参考 demo_8_6.c，利用 T/C1 的 PWM 方式，设计能产生更精确的，频率为 1KHz，幅值为 0～5V 的正弦波（提示：使用 T/C1，选择工作模式 10，设置 ICR1=250 为计数器的上限值）。
10. 将第 6 章中所有采用软件延时的例程进行修改，不使用软件延时方式，用使用 T/C 加中断的定时方式代替。

本章参考文献：

1.《ATmega16 数据手册》（英文，CDROM），ATMEL，www.atmel.com
2. AVR应用笔记AVR314《avr_app_314.pdf》（英文，CDROM），ATMEL，www.atmel.com

第 9 章 键盘输入接口与状态机编程

在前面的章节中，已经详细介绍了 AVR 单片机通用数字 I/O 口的特性以及应用于输出方式的基本使用方法，并给出了一些与中断、定时计数器相结合的输出控制应用和应用实例。本章将进一步讨论 AVR 通用 I/O 口用于按键和键盘输入接口，以及基于状态机的软件设计思想和实现。

9.1 通用I/O数字输入接口设计

假如把一个单片机嵌入式系统比做一个人的话，那么单片机就相当于人的心脏和大脑，而输入接口就好似人的感官系统，用于获取外部世界的变化、状态等各种信息，并把这些信息输送进人的大脑。嵌入式系统的人机交互通道、前向通道、数据交换和通信通道的各种功能都是由单片机的输入接口及相应的外围接口电路实现的。

对于一个电子系统来讲，外部现实世界各种类型和形态的变化和状态都需要一个变换器将其转换成电信号，而且这个电信号有时还需要经过处理，使其成为能被 MCU 容易识别和处理的数字逻辑信号，这是因为单片机常用的输入接口通常都是数字接口（A/D 接口，模拟比较器除外，他们属于模拟输入口，而是在芯片内部将模拟信号转换成数字信号的）。上述的所为“变换器”和“转换处理”从专业的角度讲就是“传感器技术”和“信号调理电路”。因此，一个单片机嵌入式系统的设计和开发人员要具备这些专业知识和技能，不仅要熟悉一些常用传感器的特性和应用，以及相关的信号调理、转换、接口电路，还要跟踪国际上新技术的发展，将新型传感器器件和新型电路元器件应用于系统设计中。采用新型传感器器件和新型电路元器件，可以大大提高嵌入式系统设计的效率，简化系统的硬件结构和软件设计难度，缩短开发周期，提高系统的性能和可靠型。

9.1.1 I/O输入接口硬件设计要点

根据系统外围电路输入的电信号形式，可以把输入信号分为以下几种形式：

(1) 模拟信号和数字信号。

传感器将某个外部参数（如温度、转速）的变化转换成电信号（电压或电流）。如果传感器输出电信号的幅度变化特征代表了外部参数的变化，如电压的升高下降（电流增大减小）表示温度高低的变化，那么这个传感器就是模拟传感器，它产生是模拟信号。由于 MCU 是数字化的，因此模拟信号要转换成数字信号才能由 MCU 处理。这个转换电路称为摸数转换“A/D”。A/D 变换是嵌入式系统重要的外围接口电路之一，用途广泛。在系统硬件设计中可以选取专用的 A/D 变换芯片作为模拟传感器和单片机之间的接口，也可以选取片内带 A/D 转换功能的单片机以简化硬件电路的设计。大多数型号的 AVR 单片机在片内都集成有 A/D 接口，关于 A/D 接口将在以后的章节中进行专门介绍。

有些外部参数的变化可以采用数字式传感器器件直接将其变化转换成数字信号。如采用光栅和光电开关器件将位移和转动圈数转换成脉冲信号，用以测量位移或转速。还有一些新型的传感器器件，把模拟传感器、A/D 变换和数字接口集成在一片芯片内，构成了智能数字传感器，如 AD 公司和 MAXIM 公司的数字温度传感器器件等，这些器件的推出，方便了嵌入式系统的硬件设计。

(2) 电压信号和电流信号。

单片机 I/O 接口的逻辑是数字电平逻辑，既以电压的高和低电平作为逻辑“1”和“0”，因此进入单片机的信号要求是电压信号。一些传感器的输出是电流信号，甚至是微小的电流，那么在进入单片机前还需要将电流信号放大、并把电流转换成电压的信号调理电路。

在一些长远距离的应用中，考虑到电压信号的抗干扰能力差，长线衰减等因素，往往在一端把电压信号变成电流信号，在长线中传送电流，而在另一端再把电流信号再转换成电压信号，这样大大提高了信号传输的可靠性，如 RS-485 通信等。另外，为了防止外部强电信号对嵌入式系统的冲击而使用的光电隔离技术，也是电流/电压变换的应用（图 9-1）。

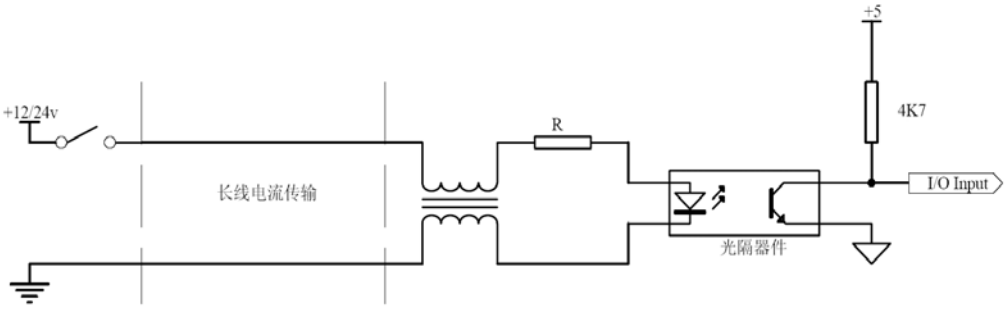

图 9-1 长线电流传输和光电隔离

(3) 单次信号和连续信号。

间隔时间较长单次产生的脉冲信号，以及较长时间保持电平不变化的信号称为单次信号。常见的单次信号一般是由按键、限位开关等人为动作或机械器件产生的信号。而连续信号一般指连续的脉冲信号，如计数脉冲信号，数据通信传输等。

单次信号要注意信号的纯净和抗干扰，如消除按键的抖动，外部的干扰等。在图 9-1 中，外部状态开关与系统之间采用电流传输方式，在系统入口串接磁阻线圈。当外部开关闭合后，回路中有大电流通过，光电器件导通输出“0”；而在开关断开后，回路中无电流流通，光电器件不会导通输出“1”。空间的电磁信号会在传输线上产生高频的干扰信号，磁阻线圈则对高频信号起到阻碍作用，使电流不能突变。另外空间电磁干扰往往能产生较高的干扰电压，但不会产生大的干扰电流（<mA），而没有毫安级的电流在回路中流过，光电器件是不会导通的。所以采用上面的电路设计，就能有效提高系统的抗干扰性。此外，采用光电隔离设计后，当外部有强信号冲击时，只能把隔离器件损坏，有效地保护了弱电系统本身。

连续信号往往在数据交换和通信通道中使用，其特点是对时间定位、捕捉、时序的要求较高，通常要对信号的边沿（上升沿或下降沿）进行检测，或由信号的边缘触发。此时外围器件的选择应符合频率的要求，同时还要求程序员熟悉信号的时序以及相关的通信接口协议等。AVR 单片机提供了丰富的接口，如外部中断、定时计数器、USART、I2C、SPI，可以方便的对连续脉冲信号进行检测、计数，以及支持多种协议的数据交换和通信等。

9.1.2 I/O输入接口软件设计要点

根据不同的硬件接口电路和嵌入式系统功能需求的不同，输入接口软件的设计也是千百万化、丰富多彩的。设计开发一个好的嵌入式系统产品，不仅要求软件设计人员要具备很强的硬件能力，还要有相当高的软件设计编程能力和经验。这是开发嵌入式系统的特点：硬件软件不可分割，同等重要。在以后的学习中，我们会逐渐体会和深入学习各种接口的软件设计技巧。

在本章以下的几节中，将主要讲解通用数字 I/O 口的一个基本和重要应用：按键和键盘接口的软硬件设计与实现。按键和键盘是单片机嵌入式系统中一个重要的组成部分，虽然在

硬件电路上的连接实现非常简单，但由于按键其本身的特殊性（如需要考虑消除抖动，确认释放等）和功能的多样性（如连续按键，多功能等），它的软件接口程序的设计和实现要相对复杂些。所以读者在学习本章的内容时，更重要的是掌握系统软件设计编写的方法和技术，采用模块化的思想实现功能模块的单一性、独立性，以及学习基于状态机思想的程序设计方法。

AVR 通用 I/O 口的结构以及相关的寄存器已经在本书的第六章中做了介绍，同时给出了通用 I/O 口的输出应用实例。但要将 AVR 单片机的 I/O 接口用做数字输入口使用时，请千万注意 AVR 单片机同其它类型单片机的 I/O 口的不同，即 AVR 的通用 I/O 口是有方向的。在程序设计中，如果要将 I/O 口作为输入接口时，不要忘记应先对 I/O 进行正确的初始化和设置。

1） 正确使用 AVR 的 I/O 口要注意：先正确设置 DDRx 方向寄存器，再进行 I/O 口的读写操作。
2） AVR 的 I/O 口复位后的初始状态全部为输入工作方式，内部上拉电阻无效。所以，外部引脚呈现三态高阻输入状态。
3） 因此，用户程序需要首先对要使用的 I/O 口进行初始化设置，根据实际需要设定使用 I/O 口的工作方式（输出还是输入），当设定为输入方式时，还要考虑是否使用内部的上拉电阻。
4） 在硬件电路设计时，如能利用 AVR 内部 I/O 口的上拉电阻，可以节省外部的上拉电阻。
5） I/O 口用于输出时，应设置 DDRx = 1 或 DDRx.n = 1，输出值写入 PORTx 或 PORTx.n 中。
6） I/O 口用于输入时，应设置 DDRx = 0 或 DDRx.n = 0。读取外部引脚电平时，应读取 PINx.n 的值，而不是 PORTx.n 的值。此时 PORTx.n = 1 表示该 I/O 内部的上拉电阻有效，PORTx.n = 0 表示不使用内部上拉，外部引脚呈现三态高阻输入状态。
7） 一旦将 I/O 口的工作方式由输出设置成输入方式后，必须等待一个时钟周期后才能正确的读到外部引脚 PINx.n 的值。

最后一点是由于在 PINx.n 和 AVR 内部数据总线之间有一个同步锁存器（图 6-4 中的 SYNCHRONIZER）电路，使用该电路避免了当系统时钟变化的短时间内外部引脚电平也同时变化而造成的信号不稳定的现象，但它将产生大约一个时钟周期（0.5～1.5）的时延。

9.2 按键输入接口设计

在单片机嵌入式系统中，按键和键盘是一个基本和常用的接口，它是构成人机对话通道的一种常用的方式。按键和键盘能实现向嵌入式系统输入数据、传输命令等功能，是人工干预、设置和控制系统运行的主要手段。

9.2.1 简单的按键输入硬件接口与分析

键盘是由一组按键组合构成的，所以我们先讨论简单的单个按键的输入。

图 9-2 是简单按键输入接口硬件连接电路图，图中单片机的三个 I/O 口 PC7、PC6、PC5 作为输入口（输入方式），分别与 K3、K2、K1 三个按键连接。其中 K2 是标准的连接方式，当没有按下 K2 时，PC6 的输入为高电平，按下 K2 输入为低电平。PC6 引脚上的电平值反映了按键的状态。

按键 K1 是一种经济的接法，它充分利用了 AVR 单片机 I/O 口的内部上拉特点。在 K1 的连接中，除了把 PC5 定义为输入方式时（DDRC.5=0），同时设置 PC5 口的上拉电阻有效(PORTC.5=1)，这样当 K1 处在断开状态时，PC5 引脚在内部上拉电阻的作用下为稳定的高电平（如果上拉电阻无效，则 PC5 处在高阻输入态，PC5 的输入易受到干扰，不稳定），按下 K1 输入为低电平。与 K2 连接方式比较，K1 连接电路中省掉了一个外部上拉电阻，而在 K2 的连接方法中，由于外部使用了上拉电阻，所以只要设置 PC6 口为输入方式即可，该口内部的上拉电阻有效与否则不必考虑了。电阻 R1 不仅起到上拉的作用，还有限流的作用，通常在 5K-50K 之间。

而对于 K3 的连接方式，我们不提倡使用，因当 K3 按下闭合时，PC7 口直接与 Vcc 接通了，有可能会造成大的短路电流流过 PC7 引脚，从而把端口烧毁。

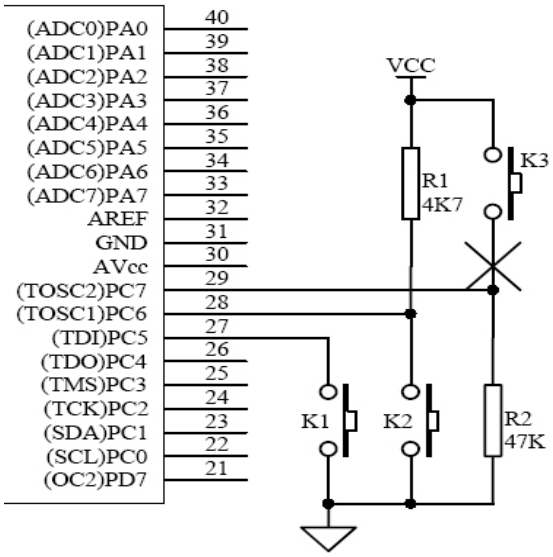

图 9-2 简单按键输入电路图

根据按键连接电路可知，按键状态的确认就是判别按键是否闭合，反映在输入口的电平就是和按键相连的 I/O 引脚呈现出高电平或低电平。如果输入高电平表示断开的话，那么低电平则表示按键闭合，所以简单的讲，在程序中通过检测引脚电平的高低，便可确认按键是否按下。

但对于实际的按键确认并不是象上面描述的那么简单。首先要考虑的是按键消抖的问题。通常，按键的开关为机械弹性触点开关，它是利用机械触点接触和分离实现电路的通、断。由于机械触点的弹性作用，加上人们按键时的力度、方向的不同，按键开关从按下到接触稳定要经过数毫秒的弹跳抖动，既在按下的几十毫秒时间里会连续产生多个脉冲。释放按键时，电路也不会一下断开，同样会产生抖动（图 9-3）。这两次抖动的时间分别为 10-20ms 左右，而按键的稳定闭合期通常大于 0.3-0.5 秒。因此，为了确保 MCU 对一次按键动作只确认一次，在确认按键是否闭合时，必须要进行消抖处理。否则，由于 MCU 软件执行的速度很快，非常可能将抖动产生的多个脉冲误认为多次的按键。在第七章的例 7.1 中，采用了简单的中断输入按键接口，没有消抖动的功能，所以出现了按键输入控制不稳定

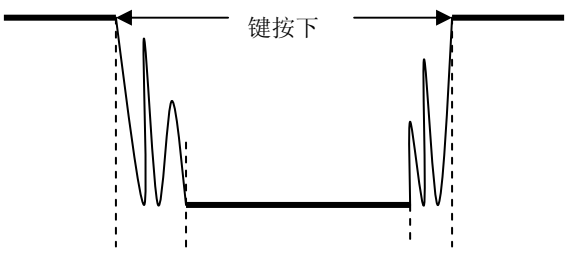

图 9-3 按键操作波形

的现象。

消除按键的抖动既可采用硬件方法，也可采用软件的方法。使用硬件消抖的方式，需要在按键连接的硬件设计上增加硬件消抖电路，如采用 R-S 触发器或 RC 积分电路等。采用硬件消抖方式增加了系统的成本，而利用软件方式消抖则是比较经济的做法，但增加了软件设计的复杂性。

软件方式消抖的基本原理是在软件中对按键进行两次测试确认，既在第一次检测到按键按下后，间隔 10ms 左右再次检测该按键是否按下，只有在两次都测到按键按下时才最终确认有键按下，从而消除了抖动的影响。

在按键接口软件中，除了要考虑按键消抖外，一般还要判别按键的释放，只有检测到按键释放以后，才能确定为一次完整的按键动作完成。

9.2.2 基于状态机的按键输入软件接口设计

一般的教课书中给出的按键输入软件接口程序通常非常简单，在程序中一旦检测到按键输入口为低电平时（图 9-2），便采用（调用）软件延时程序延时 10ms。然后再次检测按键输入口，如果还是低电平则表示按键按下，转入执行按键处理程序。如果第二次检测按键输入口为高电平，则放弃本次按键的检测，从新开始一次按键检测过程。这种方式实现的按键输入接口，作为基础学习和一些简单的系统中可以采用，但在多数的实际产品设计中，这种按键输入软件的实现方法有很大的缺陷和不足。

上面所提到简单的按键检测处理方法，不仅是由于采用了软件延时而使的 MCU 的效率降低，而且也不容易同系统中其它功能模块协调工作，系统的实时性也差。另外，由于在不同的产品系统中对按键功能的定义和使用方式也会不同，而且是多变的，加上在测试和处理按键的同时，MCU 还要同时处理其它的任务（如显示、计算、计时等），因此编写键盘和按键接口的处理程序需要掌握有效的分析方法，具备较高的软件设计能力和程序编写的技巧。

读者可以先仔细观察一下实际产品中各种按键的功能和使用。如一般的电子手表上只有 2-3 个的按键，却要实现时间、日期、闹钟时间的设置和查看显示等多种功能，因此这些按键是多功能（或复用）的，在不同的状态下，按键的功能也不同。更典型的是手机的键盘，就拿手机键盘上的数字键“2”来将，当手机用于打电话需要拨出电话号码时，按“2”键代表数值“2”。而使用手机发短信用于输入短信文字信息时（英文输入），第一次按下“2”键为字母“A”，紧接着再次按下为字母“B”，连续短时间按下该键，它的输入代表的符号不同，但在同一个位置，而稍微等待一段时间后，光标的位置就会右移，表示对最后输入字符的确认。

因此，按键输入接口设计和实现的核心，更多的体现在软件接口处理程序的设计中。下面将以此为例，介绍有限状态机的分析设计原理，以及基于状态机思想进行程序设计的基本方法与技巧。

一、有限状态机分析设计的基本原理

对于电子技术和电子工程类的读者，最先接触和使用到状态机应该是在数字逻辑电路课程里，状态机的思想和分析方法被应用于时序逻辑电路设计。其实，有限状态机（FSM）是实时系统设计中的一种数学模型，是一种重要的、易于建立的、应用比较广泛的、以描述控制特性为主的建模方法，它可以应用于从系统分析到设计（包括硬件、软件）的所有阶段。

很多实时系统，特别是实时控制系统，其整个系统的分析机制和功能与系统的状态有相当大的关系。有限状态机由有限的状态和相互之间的转移构成，在任何时候只能处于给定数目的状态中的一个。当接收到一个输入事件时，状态机产生一个输出，同时也可能伴随着状态的转移。

一个简单的有限状态机在数学上可以描述为：

（1） 一个有限的系统状态的集合 $S_i(t_k)=\{S_1(t_k),S_2(t_k),\cdots,S_q(t_k)\}$

其中(i = 1，2，…，q)。该式表示系统可能存（处）在的状态有 q 个，而在时刻 Tk，系统的状态为其中之一 Si（唯一性）。

（2） 一个有限的系统输入信号的集合 $I_j(t_k)=\{I_1(t_k),I_2(t_k),\cdots,I_m(t_k)\}$

其中(j = 1，2，…，m)，表示系统共有 m 个输入信号。该式表示在时刻 Tk，系统的输入信号为输入集合的全集或子集（集合性）。

（3） 一个状态转移函数 F： $S_{i'}(t_{k+1})=S_i(t_k)\times I_j(t_k)$

状态转移函数也是一个状态函数，它表示对于时刻 Tk，系统在某一状态 Si 下，相对给定输入 Ij 后，FSM 转入该函数产生的新状态，这个新状态就是系统在下一时刻（K+1）的状态。这个新的状态也是唯一确定的（唯一性）。

（4） 一个有限的输出信号集合 $O_l(S_i(t_k))=\{O_1(S_i(t_k)),O_2(S_i(t_k)),\cdots,O_n(S_i(t_k))\}$

其中(l = 1，2，…，n)，表示系统共有 n 个输出信号。该式表示对于在时刻 Tk，系统的状态为 Si 时，其输出信号为输出集合的全集或子集（集合性）。这里需要注意的是，系统的输出只与系统所处的状态有关。

（5） 时间序列 $T=\{t_0,t_1,\cdots,t_k,t_{k+1},\cdots\}$

在状态机中，时间序列也是非常重要的一个因素，从硬件的角度看，时间序列如同一个触发脉冲序列或同步信号，而从软件的角度看，时间序列就是一个定时器。状态机由时间序列同步触发，定时检测输入，以及根据当前的状态输出相应的信号，并确定下一次系统状态的转移。在时间序列进入下一次触发时，系统的状态将根据前一次的状态和输入情况发生状态的转移。其次，作为时间序列本身也可能是一个系统的输入信号，影响到状态的改变，进而也影响到系统的输出。所以对于时间序列，正确分析和考虑选择合适的时间段的间隔也是非常重要的。间隔太短的话，对系统的速度、频率响应要求高，并且可能减低系统的效率；间隔太长时，系统的实时性差，响应慢，还有可能造成外部输入信号的丢失。一般情况下，时间序列的时间间隔的选取，应稍微小于外部输入信号中变化最快的周期值。

通常主要有两种方法来建立有限状态机，一种是“状态转移图”，另一种是“状态转移表”，分别用图形方式和表格方式建立有限状态机。实时系统经常会应用在比较大型的系统中，这时采用图形或表格方式对理解复杂的系统具有很大的帮助。

总的来说，有限状态机的优点在于简单易用，状态间的关系能够直观看到。应用在实时系统中时，便于对复杂系统进行分析。

下面将给出两个按键与显示相结合的应用设计实例，结合设计的例子，讨论如何使用有限状态机进行系统的分析和设计，以及如何在软件中进行描述和实现。

二、基于状态机分析的简单按键设计（一）

我们把单个按键作为一个简单的系统，根据状态机的原理对其动作和确认的过程进行分析，并用状态图表示出来，然后根据状态图编写出按键接口程序。

把单个按键看成是一个状态机话，首先需要对一次按键操作和确认的实际过程进行分析，根据实际的情况和系统的需要确定按键在整个过程的状态，每个状态的输入信号和输出信号，以及状态之间的转换关系。最后还要考虑时间序列的间隔。

采用状态机对一个系统进行分析是一项非常细致的工作，它实际上是建立在对真实系统有了全面深入的了解和认识的基础之上，进行综合和抽象化的模型建立的过程。这个模型必

须与真实的系统相吻合，既能正确和全面的对系统进行描述，也能够适合使用软件或硬件方式来实现。

在一个嵌入式系统中，按键的操作是随机的，因此系统软件对按键需要一直循环查询。由于按键的检测过程需要进行消抖处理，因此取状态机的时间序列的周期为 10ms 左右，这样不仅可以跳过按键抖动的影响，同时也远小于按键 0.3-0.5 秒的稳定闭合期（图 9-3），不会将按键操作过程丢失。很明显，系统的输入信号是与按键连接的 I/O 口电平，“1”表示按键处于开放状态，“0”表示按键处于闭合状态（图 9-2）。而系统的输出信号则表示检测和确认到一次按键的闭合操作，用“1”表示。

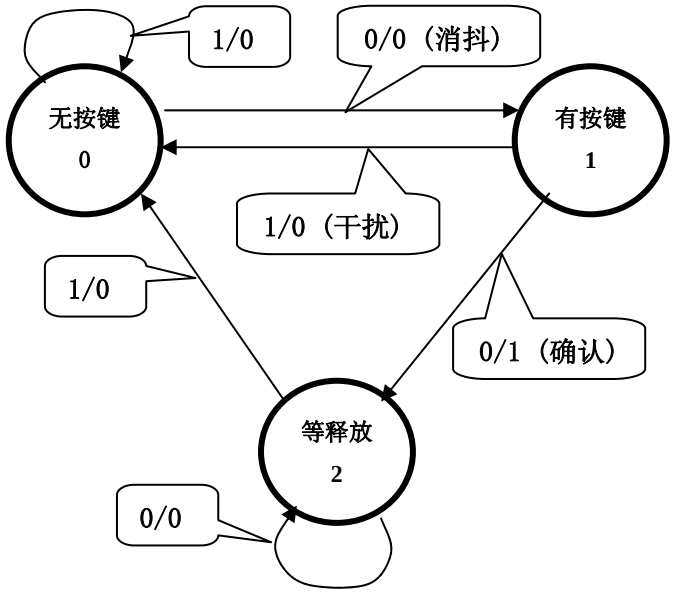

图 9-4 简单按键状态机的状态转换图

图 9-4 给出了一个简单按键状态机的状态转换图。在图中，将一次按键完整的操作过程分解为 3 个状态，采用时间序列周期为 10ms。下面对该图做进一步的分析和说明，并根据状态图给出软件的实现方法。

首先，读者要充分体会时间序列的作用。在这个系统中，采用的时间序列周期为 10ms，它意味着，每隔 10ms 检测一次按键的输入信号，并输出一次按键的确认信号，同时按键的状态也发生一次转换。

图中“状态 0”为按键的初始状态，当按键输入为“1”时，表示按键处于开放，输出“0”（1/0），下一状态仍旧为“状态 0”。当按键输入为“0”，表示按键闭合，但输出还是“0”（0/0）（没有经过消抖，不能确认按键真正按下），下一状态进入“状态 1”。

“状态 1”为按键闭合确认状态，它表示了在 10ms 前按键为闭合的，因此当再次检测到按键输入为“0”时，可以确认按键被按下了（经过 10ms 的消抖），输出“1”表示确认按键闭合（0/1），下一状态进入“状态 2”。而当再次检测到按键的输入为“1”时，表示按键可能处在抖动干扰，输出为“0”（1/0），下一状态返回到“状态 0”。这样，利用状态 1，实现了按键的消抖处理。

“状态 2”为等待按键释放状态，因为只有等按键释放后，一次完整的按键操作过程才算完成。

从对图 9-4 的分析中可以知道，在一次按键操作的整个过程，按键的状态是从“状态 0”->“状态 1”->“状态 2”，最后返回到“状态 0”的。并且在整个过程中，按键的输出信号仅在“状态 1”时给出了唯一的一次确认按键闭合的信号“1”（其它状态均输出“0”）。所以上面状态机所表示的按键系统，不仅克服了按键抖动的问题，同时也确保在一次按键整个的过程中，系统只输出一次按键闭合信号（“1”）。换句话讲，不管按键被按下的时间保持多

长，在这个按键的整个过程中都只给出了一次确认的输出，因此在这个设计中，按键没有“连发”功能，它是一个最简单和基本的按键。

一旦有了正确的状态转换图，就可以根据状态转换图编写软件了。在软件中实现状态机的方法和程序结构通常使用多分支结构（IF-ELSEIF-ELSE、CASE 等）实现。下面是根据图 9-4、基于状态机方式编写的简单按键接口函数 read_key()。

```
#define key_input       PIND.7              // 按键输入口
#define key_state_0     0
#define key_state_1     1
#define key_state_2     2

char read_key(void)
{
    static char key_state = 0;
    char key_press, key_return = 0;

    key_press = key_input;                  // 读按键 I/O 电平
    switch (key_state)
    {
        case key_state_0:                   // 按键初始态
            if (!key_press) key_state = key_state_1;  // 键被按下，状态转换到键确认态
            break;
        case key_state_1:                   // 按键确认态
            if (!key_press)
            {
                key_return = 1;             // 按键仍按下，按键确认输出为“1”
                key_state = key_state_2;    // 状态转换到键释放态
            }
            else
                key_state = key_state_0;    // 按键已抬起，转换到按键初始态
            break;
        case key_state_2:
            if (key_press) key_state = key_state_0;   //按键已释放，转换到按键初始态
            break;
    }
    return key_return;
}
```

该简单按键接口函数 read_key()在整个系统程序中应每隔 10ms 调用执行一次，每次执行时将先读取与按键连接的 I/O 的电平到变量 key_press 中，然后进入用 switch 结构构成的状态机。switch 结构中的 case 语句分别实现了 3 个不同状态的处理判别过程，在每个状态中将根据状态的不同，以及 key_press 的值（状态机的输入）确定输出值（key_return），和确定下一次按键的状态值（key_state）。

函数 read_key()的返回参数提供上层程序使用。返回值为 0 时，表示按键无动作；而返回 1 表示有一次按键闭合动作，需要进入按键处理程序做相应的键处理。

在函数 read_key()中定义了 3 个局部变量，其中 key_press 和 key_return 为一般普通

的局部变量，每次函数执行时，key_press 中保存着刚检测的按键值。key_return 为函数的返回值，总是先初始化为 0，只有在状态 1 中重新置 1，作为表示按键确认的标志返回。变量 key_state 非常重要，它保存着按键的状态值，该变量的值在函数调用结束后不能消失，必须保留原值，因此在程序中定义为“局部静态变量”，用 static 声明。如果使用的语言环境不支持 static 类型的局部变量，则应将 key_state 定义为全局变量（关于局部静态变量的特点请参考相关介绍 C 语言程序设计的书籍）。

例 9.1　单按键的实时时钟秒校时设置设计（一）

1）　硬件电路

在前面的章节中曾几次给出了简易实时时钟的设计例子，但都没有加入按键，不能实现时钟校时的设置。下面结合上面的按键接口的设计，实现对时钟的校时设置。在该例子中，只是实现了秒单元的校时设置，其重点是让读者体会和实践按键输入接口和处理的实现。

时钟系统的硬件电路与图 6-15 基本相同，仅在 I/O 口 PD7 上连接一个按键 K1，该按键的功能为秒加 1，既每按下一次，秒加 1，到 60 秒时分加 1，秒回到为 0。图 9-5 为电原理图。

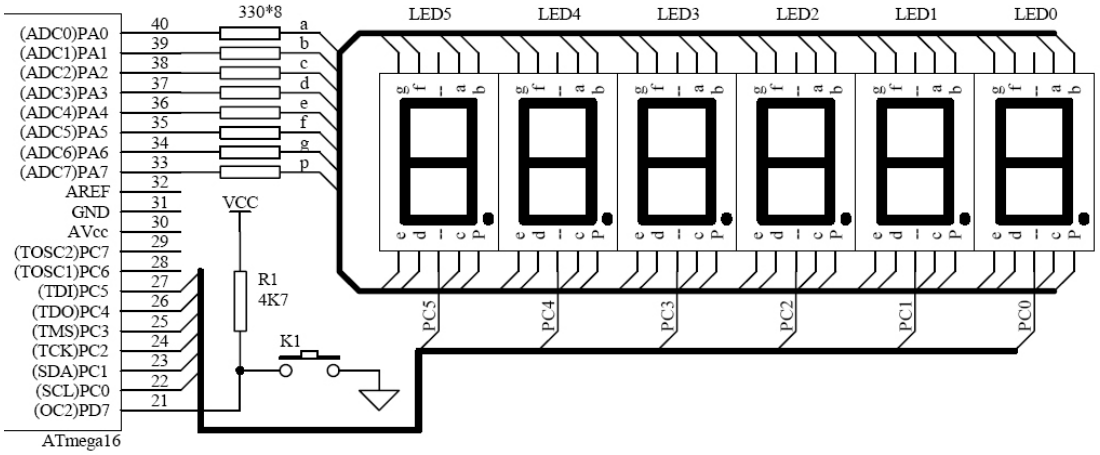

图 9-5 可实现秒校时的简单时钟电路

2）　软件设计

以下是采用一个简单按键实现秒单元的校时设置系统的程序。程序中，给出了采用一个简单按键实现时钟校时（仅秒位）设置功能的基本设计，时钟本身的计时显示方法用秒闪烁表示（每秒种 LED 数码管的小数点段闪烁一次）。

```
/*****************************************
File name          : demo_9_1.c
Chip type          : ATmega16
Program type       : Application
Clock frequency    : 4.000000 MHz
Memory model       : Small
External SRAM size : 0
Data Stack size    : 256
*****************************************/

#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
```

```
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

char time[3];                    // 时、分、秒计数单元
char dis_buff[6];                // 显示缓冲区，存放要显示的 6 个字符的段码值
char time_counter,key_stime_counter;      // 时间计数单元，
char posit;
bit  point_on, time_1s_ok,key_stime_ok;

void display(void)                    // 6 位 LED 数码管动态扫描函数
{
     PORTC = 0xff;
     PORTA = led_7[dis_buff[posit]];
     if (point_on && (posit==2||posit==4)) PORTA |= 0x80;
     PORTC = position[posit];
     if (++posit >=6 ) posit = 0;//  (3)
}

// Timer 0 比较匹配中断服务，2ms 定时
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
     display();                           // LED 扫描显示
     if (++key_stime_counter >=5)
     {
          key_stime_counter = 0;
          key_stime_ok = 1;               // 10ms 到
          if (++time_counter >= 100)
          {
               time_counter = 0;
               time_1s_ok = 1;            // 1s 到
          }
     }
}

void time_to_disbuffer(void)              // 时钟时间送显示缓冲区函数
{
     char i,j=0;
     for (i=0;i<=2;i++)
     {
          dis_buff[j++] = time[i] % 10;
          dis_buff[j++] = time[i] / 10;
     }
}

#define key_input  PIND.7                 // 按键输入口
```

```
#define key_state_0    0
#define key_state_1    1
#define key_state_2    2

char read_key(void)
{
    static char key_state = 0;
    char key_press, key_return = 0;

    key_press = key_input;              // 读按键 I/O 电平
    switch (key_state)
    {
        case key_state_0:               // 按键初始态
            if (!key_press) key_state = key_state_1;  // 键被按下，状态转换到键确认态
            break;
        case key_state_1:               // 按键确认态
            if (!key_press)
            {
                key_return = 1;             // 按键仍按下，按键确认输出为"1" (1)
                key_state = key_state_2;    // 状态转换到键释放态
            }
            else
                key_state = key_state_0;    // 按键已抬起，转换到按键初始态
            break;
        case key_state_2:
            if (key_press) key_state = key_state_0;   //按键已释放，转换到按键初始态
            break;
    }
    return key_return;
}

void main(void)
{
    PORTA = 0x00;                   // 显示控制 I/O 端口初始化
    DDRA = 0xFF;
    PORTC = 0x3F;
    DDRC = 0x3F;
    DDRD = 0x00;                    // PD7 为输入方式
    // T/C0 初始化
    TCCR0 = 0x0B;           // 内部时钟，64 分频（4M/64=62.5KHz），CTC 模式
    TCNT0 = 0x00;
    OCR0 = 0x7C;            // OCR0 = 0x7C(124),(124+1)/62.5=2ms
    TIMSK = 0x02;           // 允许 T/C0 比较匹配中断
```

```
    time[2] = 23; time[1] = 58; time[0] = 55; // 设时间初值 23:58:55
    posit = 0;
    time_to_disbuffer();

    #asm("sei")             // 开放全局中断

    while (1)
    {
        if (time_1s_ok)                 // 1 秒到
        {
            time_1s_ok = 0;
            point_on = ~point_on;       // 秒闪烁标志
        }
        if (key_stime_ok)               // 10ms 到，键处理
        {
            key_stime_ok = 0;
            if (read_key())             // 调用按键接口程序
            {                           // 按键确认按下
                if (++time[0] >= 60)    // 秒加 1，以下为时间调整
                {
                    time[0] = 0;
                    if (++time[1] >= 60)
                    {
                        time[1] = 0;
                        if (++time[2] >= 24) time[2] = 0;
                    }
                }
            }
            time_to_disbuffer();        // 新调整好的时间送显示缓冲区
        }
    };
}
```

该程序中 LED 数码管动态扫描等部分与前面介绍的例子相同，T/C0 的工作于 CTC 方式，每 2ms 产生中断。在 T/C0 中断服务中增加了 10ms 到的标志变量 key_stime_ok，作为按键状态机的时间触发序列信号。在主程序中，每隔 10ms 调用 read_key()按键接口程序，当函数返回值为 1 时，说明产生了一次按键操作过程，秒位加 1，然后进行时间调整。

3） 思考与实践

- ✓ 仔细分析 read_key()函数的调用和执行情况，说明为什么该按键接口程序可以正确的处理一次按键的操作过程？
- ✓ 如果将 read_key()中标有（1）的语句 key_return = 1 去掉，而将状态 2 的处理程序改为：

```
case key_state_2:
    if (key_press) key_state = key_state_0;    //按键已释放，转换到按键初始态
    else  key_return = 1;
```

```
        break;
```

或：

```
case key_state_2:
     if (key_press)
     {     key_state = key_state_0;   //按键已释放，转换到按键初始态
           key_return = 1;
     }
     break;
```

时，按键处理会产生什么变化？为什么？

三、基于状态机分析的简单按键设计（二）

在上面的例子中，对秒的校时设置操作还是不方便，例如，如果将秒从 00 设置为 59，需要按键 59 次。如果将按键设计成一个具有“连发”功能的系统，就能很好的解决这个问题。

现在考虑这样的一个按键系统：当按键按下后在 1 秒内释放了，此时秒计时加 1，而当按键按下后在 1 秒内没有释放，那么以后每隔 0.5 秒，秒计时就会自动加上 10，直到按键释放为止。这样的按键系统就具备了一种“连发”功能，其中时间也成为系统的一个输入参数了。

参考图 9-4，根据状态机的分析方法，可以得到具有上述“连发”功能的按键系统状态转换图（图 9-6）。

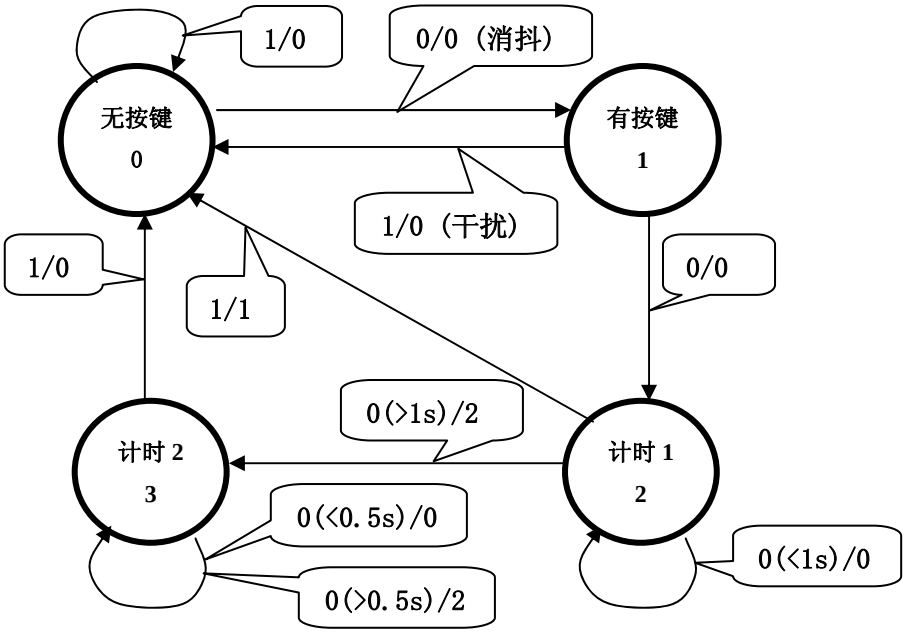

图 9-6 具有连发功能按键状态机的状态转换图

在图 9-6 所示的图中，按键系统输出“1”表示按键在 1 秒内释放了，输出“2”表示按键产生一次“连发”的效果。根据图 9-4，下面是基于状态机方式编写的带“连发”功能按键接口函数 read_key_n()。

```
#define key_input       PIND.7              // 按键输入口
#define key_state_0     0
#define key_state_1     1
#define key_state_2     2
#define key_state_3     3
```

```
char read_key_n(void)
{
    static char key_state = 0, key_time = 0;
    char key_press, key_return = 0;

    key_press = key_input;                  // 读按键 I/O 电平
    switch (key_state)
    {
        case key_state_0:                   // 按键初始态
            if (!key_press) key_state = key_state_1;  // 键被按下，状态转换到键确认态
            break;
        case key_state_1:                   // 按键确认态
            if (!key_press)
            {
                key_state = key_state_2;    // 按键仍按下，状态转换到计时 1
                key_time = 0;               // 清另按键时间计数器
            }
            else key_state = key_state_0;   // 按键已抬起，转换到按键初始态
            break;
        case key_state_2:
            if (key_press)
            {
                key_state = key_state_0;    // 按键已释放，转换到按键初始态
                key_return = 1;             // 输出“1”
            }
            else if (++key_time >= 100)     // 按键时间计数
            {
                key_state = key_state_3;    // 按下时间>1s，状态转换到计时 2
                key_time = 0;               // 清按键计数器
                key_return = 2;             // 输出“2”
            }
            break;
        case key_state_3:
            if (key_press) key_state = key_state_0; //按键已释放，转换到按键初始态
            else
            {
                if (++key_time >= 50)       // 按键时间计数
                {
                    key_time = 0;           // 按下时间>0.5s，清按键计数器
                    key_return = 2;         // 输出“2”
                }
            }
            break;
```

```
    }
    return key_return;
}
```

函数 read_key_n()与前面的 read_key()结构非常类似，设计为每隔 10ms 被调用执行一次，在构成状态机的 switch 结构中使用了局部静态变量 key_time 作为按键时间计数器，记录按键按下的时间值。

函数 read_key_n()的返回参数提供上层程序使用。返回值为 0 时，表示按键无动作；返回 1 表示一次按键的“单发”（<1s）动作；返回 2 表示一次按键的“连发”动作，提供按键处理程序做相应的键处理。

例 9.2　单按键的实时时钟秒校时设置设计（二）

1） 硬件电路

硬件电路还是图 9-6，在 I/O 口 PD7 上连接一个按键 K1，该按键为具备“连发”功能，按键按下后并在 1 秒内释放，此时秒计时加 1；而当按键按下后在 1 秒内没有释放，那么每隔 0.5 秒，秒计时就会自动加上 10，直到按键释放为止。

2） 软件设计

以下是采用具备“连发”功能按键实现秒单元的校时设置系统的程序，程序中的显示等部分与 demo_9_1.c 相同，仅仅在主程序中做了相应的改变。

```
/*********************************************
File name          : demo_9_2.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/

#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

char time[3];                       // 时、分、秒计数单元
char dis_buff[6];                   // 显示缓冲区，存放要显示的 6 个字符的段码值
char time_counter,key_stime_counter;
char posit;
bit  point_on, time_1s_ok,key_stime_ok;

void display(void)                      // 6 位 LED 数码管动态扫描函数
{
    ……
}

// Timer 0 比较匹配中断服务,2ms 定时
```

```
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
    。。。。。。
}

void time_to_disbuffer(void)              // 时钟时间送显示缓冲区函数
{
    。。。。。。
}

#define key_input       PIND.7            // 按键输入口
#define key_state_0     0
#define key_state_1     1
#define key_state_2     2
#define key_state_3     3

char read_key_n(void)
{
    。。。。。。
}

void main(void)
{
    PORTA = 0x00;                              // 显示控制 I/O 端口初始化
    DDRA = 0xFF;
    PORTC = 0x3F;
    DDRC = 0x3F;
    DDRD = 0x00;
    // T/C0 初始化
    TCCR0=0x0B;             // 内部时钟，64 分频（4M/64=62.5KHz），CTC 模式
    TCNT0=0x00;
    OCR0=0x7C;           // OCR0 = 0x7C(124),(124+1)/62.5=2ms
    TIMSK=0x02;             // 允许 T/C0 比较匹配中断

    time[2] = 23; time[1] = 58; time[0] = 55; // 设时间初值 23:58:55
    posit = 0;
    time_to_disbuffer();

    #asm("sei")             // 开放全局中断

    while (1)
    {
        if (time_1s_ok)
        {
```

```
            time_1s_ok = 0;
                point_on = ~point_on;            // 1 秒到，秒闪烁标志取反
        }
        if (key_stime_ok)
        {
            key_stime_ok = 0;                // 10ms 到
            switch (read_key_n())
            {
                case 1:
                    ++time[0];
                    break;
                case 2:
                    time[0] += 10;
                    break;
            }
            if (time[0] >= 60)      // 按键确认按下，秒加 1，以下时间调整
            {
                time[0] -= 60;
                if (++time[1] >= 60)
                {
                    time[1] = 0;
                    if (++time[2] >= 24) time[2] = 0;
                }
            }
            time_to_disbuffer();        // 新调整好的时间送显示缓冲区
        }
    };
}
```

主程序中，每隔 10ms 调用 read_key_n()按键接口程序，并根据其返回值对秒位的时间进行加 1 或加 10 的处理。

9.3 键盘输入接口设计

在上节中按键的连接方法采用的是独立式按键接口方式。独立式按键就是各按键相互独立，每个按键各占用一位 I/O 的口线，它们之间状态是独立的，相互之间没有影响，只要单独测试口线电平的高低就能判断键的状态。独立式按键电路简单、配置灵活，软件结构也相对简单。此种接口方式适用于系统需要按键数目较少的场合。在按键数量较多的情况下，如系统需要 12 或 16 个按键的键盘时，采用独立式接口方式就会占用太多的 I/O 口，因此一般适用于按键数量多的硬件连接方式往往使用矩阵式键盘接口。

9.3.1 矩阵键盘的工作原理和扫描确认方式

当键盘中按键数量较多时，为了减少对 I/O 口的占用，通常将按键排列成矩阵形式，也称为行列键盘，这是一种常见的连接方式。矩阵式键盘接口见图 9-7 所示，它由行线和列线组成，按键位于行、列的交叉点上。当键被按下时，其交点的行线和列线接通，相应的行线

或列线上的电平发生变化，MCU 通过检测行或列线上的电平变化可以确定哪个按键被按下。

图 9-7 为一个 4 x 3 的行列结构，可以构成 12 个键的键盘。如果使用 4 x 4 的行列结构，就能组成一个 16 键的键盘。很明显，在按键数量多的场合，矩阵键盘与独立式按键键盘相比可以节省很多的 I/O 口线。

矩阵键盘不仅在连接上比单独式按键复杂，它的按键识别方法也比单独式按键复杂。在矩阵键盘的软件接口程序中，常使用的按键识别方法有行扫描法和线反转法。这两种方法的基本思路是采用循环查循的方法，反复查询按键的状态，因此会大量占用 MCU 的时间，所以较好的方式也是采用状态机的方法来设计，尽量减少键盘查询过程对 MCU 的占用时间。

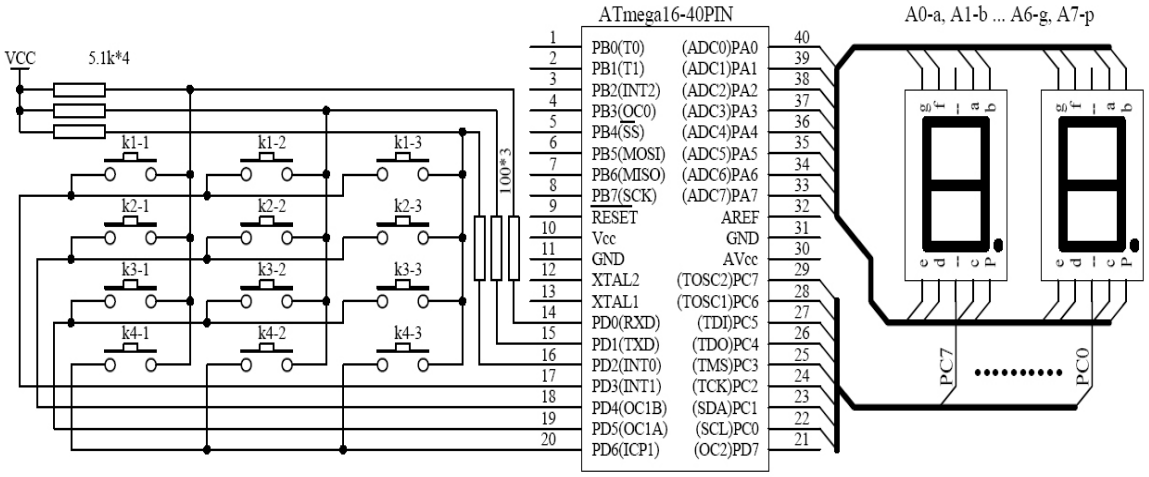

图 9-7 4*3 矩阵键盘的组成

下面以图 9-7 为例，介绍采用行扫描法对矩阵键盘进行判别的思路。图 9-7 中，PD0、PD1、PD2 为 3 根列线，作为键盘的输入口（工作于输入方式）。PD3、PD4、PD5、PD6 为 4 根行线，工作于输出方式，由 MCU（扫描）控制其输出的电平值。行扫描法也称为逐行扫描查询法，其按键识别的过程如下。

- ✓ 将全部行线 PD3－PD6 置低电平输出，然后读 PD0－PD2 三根输入列线中有无低电平出现。只要有低电平出现，则说明有键按下（实际编程时，还要考虑按键的消抖）。如读到的都是高电平，则表示无键按下。
- ✓ 在确认有键按下后，需要进入确定具体哪一个键闭合的过程。其思路是：依次将行线置为低电平，并检测列线的输入（扫描），进而确认是具体的按键位置。如当 PD5 输出低电平时（PD3=1、PD4=1、PD5=0、PD6=1），测到 PD1 的输入为低电平（PD0=1、PD1=0、PD2=1），则可确认按键 K3-2 处于闭合状态。通过以上分析可以看出，MCU 对矩阵键盘的按键识别，是采用扫描方式控制行线的输出和检测列线输入的信号相配合实现的。
- ✓ 矩阵按键的识别仅仅是确认和定位了行和列的交叉点上的按键，接下来还要考虑键盘的编码，即对各个按键进行编号。在软件中常通过计算的方法或查表的方法对按键进行具体的定义和编号。

在单片机嵌入式系统中，键盘扫描只是 MCU 的工作内容之一。MCU 除了要检测键盘和处理键盘操作之外，还要进行其他事物的处理，因此，MCU 如何响应键盘的输入需要在实际系统程序设计时认真考虑。

键盘扫描处理的设计原则是：既要保证 MCU 能及时的判别按键的动作，处理按键输入的操作，又不能过多占用 MCU 的工作时间，让它有充裕的时间去处理其它的操作。

通常，完成键盘扫描和处理的程序是系统程序中的一个专用子程序，MCU 调用该键盘扫描子程序对键盘进行扫描和处理的方式有三种：程序控制扫描、定时扫描和中断扫描。

- ✓ 程序控制扫描方式。在主控程序中的适当位置调用键盘扫描程序，对键盘进行读取和处理。
- ✓ 定时扫描方式。在该方式中，要使用 MCU 的一个定时器，使其产生一个 10ms 的定时中断，MCU 响应定时中断，执行键盘扫描，当在连续两次中断中都读到相同的按键按下（间隔 10ms 作为消抖处理），MCU 才执行相应的键处理程序。
- ✓ 中断方式。使用中断方式时，键盘的硬件电路要做一定的改动，增加一个按键产生中断信号的输入线，当键盘有按键按下时，键盘硬件电路产生一个外部的中断信号，MCU 响应外部中断，进行键盘处理。

下面我们介绍基于状态机并采用定时键盘扫描的键盘处理系统的设计方法。

9.3.2 定时扫描方式的键盘接口程序

根据图 9-7，下面的键盘接口函数 read_keyboaed()完成了对 4*3 键盘的扫描识别和键盘的编码。编码键盘的定义使用 define 语句定义，键盘的形式类似电话和手机键盘，如图 9-8 所示。

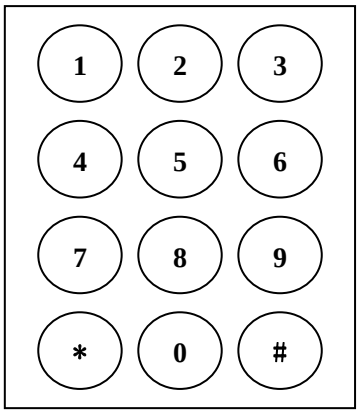

图 9-8 手机键盘

```
#define No_key   255
#define K1_1     1
#define K1_2     2
#define K1_3     3
#define K2_1     4
#define K2_2     5
#define K2_3     6
#define K3_1     7
#define K3_2     8
#define K3_3     9
#define K4_1     10
#define K4_2     0
```

```
#define K4_3     11
#define Key_mask      0b00000111

char read_keyboard()
{
  static char key_state = 0, key_value, key_line;
  char key_return = No_key,i;
  switch (key_state)
  {
       case 0:
            key_line = 0b00001000;
            for (i=1; i<=4; i++)                    // 扫描键盘
            {
                 PORTD = ~key_line;                 // 输出行线电平
                 PORTD = ~key_line;                 // 必须送 2 次!!!(注 1)
                 key_value = Key_mask & PIND;       // 读列电平
                 if (key_value == Key_mask)
                      key_line <<= 1;               // 没有按键，继续扫描
                 else
                 {
                      key_state++;                  // 有按键，停止扫描
                      break;                        // 转消抖确认状态
                 }
            }
            break;
       case 1:
            if (key_value == (Key_mask & PIND))     // 再次读列电平，
            {
                 switch (key_line | key_value)      // 与状态 0 的相同，确认按键
                 {                                  // 键盘编码，返回编码值
                      case 0b00001110:
                           key_return = K1_1;
                           break;
                      case 0b00001101:
                           key_return = K1_2;
                           break;
                      case 0b00001011:
                           key_return = K1_3;
                           break;
                      case 0b00010110:
                           key_return = K2_1;
                           break;
                      case 0b00010101:
                           key_return = K2_2;
```

```
                    break;
                case 0b00010011:
                    key_return = K2_3;
                    break;
                case 0b00100110:
                    key_return = K3_1;
                    break;
                case 0b00100101:
                    key_return = K3_2;
                    break;
                case 0b00100011:
                    key_return = K3_3;
                    break;
                case 0b01000110:
                    key_return = K4_1;
                    break;
                case 0b01000101:
                    key_return = K4_2;
                    break;
                case 0b01000011:
                    key_return = K4_3;
                    break;
            }
            key_state++;                    // 转入等待按键释放状态
        }
        else
            key_state--;                    // 两次列电平不同返回状态 0,（消抖处理）
        break;
    case 2:                                 // 等待按键释放状态
        PORTD = 0b00000111;                 // 行线全部输出低电平
        PORTD = 0b00000111;                 // 重复送一次
        if ( (Key_mask & PIND) == Key_mask)
            key_state=0;                    // 列线全部为高电平返回状态 0
        break;
  }
  return key_return;
}
```

系统主程序应每隔 10ms 调用该键盘接口函数 read_keyboaed()，函数返回值为 255 时表示无按键按下。检测和确认按键按下时，函数返回值为 0 到 11 之间的一个，该返回值已经是经过了键盘编码的值。

键盘接口函数 read_keyboaed()是基于状态机实现的，将键盘扫描处理过程化分成三个状态，每个状态的功能为：

- ✓ 状态 0，键盘扫描检测。控制 PD3-PD6，4 根行线逐行输出低电平，对键盘进行扫描检测。一旦检测到有键按下（key_value），立即停止键盘的扫描，状态转换到状

态 1。注意此时变量 key_value 中保存着读到的列线输入值，而且该行线低电平的输出是保持不变的。

- ✓ 状态 1，消抖处理和键盘编码。再次检测键盘列线的输入，并与状态 0 时的 key_value 比较，不相等则返回状态 0，实现了消抖处理。相等则确认该键的输入，进行键盘编码和设置函数的返回值，状态转化到状态 2。
- ✓ 状态 2，等待按键释放。控制 PD3-PD6，4 根行线全部输出低电平，检测 3 根列线输入全部为高电平（无按键按下）时状态返回到状态 0。

读者在阅读该段程序时，请注意 key_mask、key_value、key_line 的作用和使用，它们不仅与键盘的硬件连接有关系，同时还要注意他们在程序中是如何使用的，其值的保存等等。

通过这个例子也可以看出，在硬件设计的过程中，也需要认真考虑软件的编写。比如，你也可以将 4 根键盘行线与 PD0、PD2、PD3、PD6 连接，列线使用 PD1、PD4、PD5，从硬件的角度看是完全可以的，但会给软件编写造成很多麻烦。所以，一个好的嵌入式系统工程师必须同时具备良好的软件设计编写能力和硬件设计能力。

实际上 read_keyboaed()还是一个比较简单的键盘接口函数，还不能处理类似多键按下的识别、按键“连发”等功能。但当你真正掌握了基本键盘接口的设计思想和方法后，就可以在这个简单的键盘接口函数基础上，设计出功能更加完善的键盘系统了。

（注 1）当 AVR 的 I/O 口处于输入方式工作时，其对应的 PINx 就是外部引脚上的实际电平。为了防止读入 PINx 时数据的不稳定和不确定，（例如在读的期间，正好引脚电平发生改变），AVR 的 I/O 输入端到总线之间加入一个同步锁存器，用以保证读入数据的稳定和确定。同步锁存器在系统 I/O 时钟的作用下，经过 1/2-3/2 个系统时钟周期，将引脚电平锁定，提供MCU读取。也就是说外部引脚的电平变化要经过1/2-3/2个时钟周期才能真正的被读到，见图 9-9。

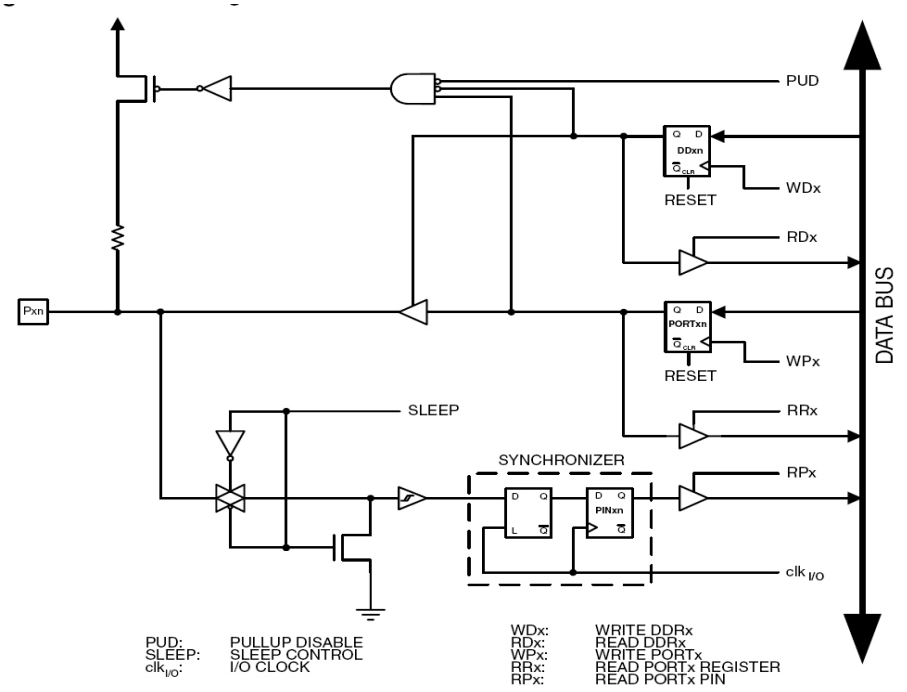

图 9-9 AVR I/O 内部结构图

因此，当编写程序读取 AVR 的 I/O 口 PINx 电平值时应注意：

- 当 I/O 口从输出方式改成输入方式后，应延时一个 CLK 再读。
- 当 I/O 外部引脚电平改变后，也要延时一个 CLK 后再读取。

在本例的键盘扫描程序中，根据扫描条件，首先改变行线输出的电平，如果按键按下的话，那么对应点的列线（输入口）电平也马上改变了，此时需要延时一个 CLK 后读列线的输入电平值才是正确的。程序中采用输出 2 次相同的行电平的方式，第 2 次输出的实际作用是起到延时的作用（其实相当于 2 个 NOP）。当然，也可以输出 1 次行电平，在读 I/O 口时采用读 2 次的方式，只要满足规定的延时时间就可以的。

例 9.3　简单电话拨号键盘的设计

1）　硬件电路

在这个例子中，结合图 9-7 的硬件电路和图 9-8 定义的键盘，实现一个简单的电话拨号键盘。系统由 PA、PC 口控制的 8 个 LED 数码管和 PD 口的 4*3 键盘组成，系统上电时，8 个 LED 数码管显示“--------”8 条横线，每按下一个号码后，原 8 位 LED 数码管的显示内容向左移动一位，最右边一位则显示键盘上刚按下的数字（“*”键用“A”表示，“#”键用“b”表示）。要求：对键盘按键操作的反应迅速而且无误，同时按键操作过程中应保证 LED 的扫描显示均匀、连续不间断。

2）软件设计

```
/*********************************************
File name          : demo_9_3.c
Chip type          : ATmega16
Program type       : Application
Clock frequency    : 4.000000 MHz
Memory model       : Small
External SRAM size : 0
Data Stack size    : 256
*********************************************/

#include <mega16.h>
flash char led_7[13]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,     // 字型码
                      0x7D,0x07,0x7F,0x6F,0x77,0x7C,0x40};
                                    // 后 3 位为 “A”,“b”,“-”
flash char position[8]={0x7f,0xbf,0xdf,0xef,0xf7,0xfb,0xfd,0xfe};

char dis_buff[8];                  // 显示缓冲区，存放要显示的 8 个字符的段码值
char key_stime_counter;
char posit;
bit  key_stime_ok;

void display(void)                 // 8 位 LED 数码管动态扫描函数
{
    PORTC = 0xff;
    PORTA = led_7[dis_buff[posit]];
    PORTC = position[posit];
    if (++posit >=8 ) posit = 0;
}
```

```
// Timer 0 比较匹配中断服务,2ms 定时
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
     display();                              // 调用 LED 扫描显示
     if (++key_stime_counter >=5)
     {
          key_stime_counter = 0;
          key_stime_ok = 1;                  // 10ms 到
     }
}
#define No_key     255
#define K1_1  1
#define K1_2  2
#define K1_3  3
#define K2_1  4
#define K2_2  5
#define K2_3  6
#define K3_1  7
#define K3_2  8
#define K3_3  9
#define K4_1  10
#define K4_2  0
#define K4_3  11
#define Key_mask   0b00000111

char read_keyboard()
{
     static char key_state = 0, key_value, key_line;
     char key_return = No_key,i;
     ……                          // 同上面 read_keyboard()
}

void main(void)
{
     char i, key_temp;

     PORTA = 0x00;                                  // 显示控制 I/O 端口初始化
     DDRA = 0xFF;
     PORTC = 0xFF;
     DDRC = 0xFF;
     PORTD = 0xFF;                                  // 键盘接口初始化
     DDRD = 0xF8;                                   // PD2、PD1、PD0 列线，输入方式，上拉有效
     // T/C0 初始化
     TCCR0=0x0B;              // 内部时钟，64 分频（4M/64=62.5KHz），CTC 模式
```

```
        TCNT0=0x00;
        OCR0=0x7C;              // OCR0 = 0x7C(124),(124+1)/62.5=2ms
        TIMSK=0x02;             // 允许 T/C0 比较匹配中断

        for (i=0; i<8 ;i++)
        {dis_buff[i]= 12;}      // LED 初始显示 8 个“-”
        #asm("sei")             // 开放全局中断

        while (1)
        {
            if (key_stime_ok)
            {
                key_stime_ok = 0;               // 10ms 到
                key_temp = read_keyboard();     // 调用键盘接口函数读键盘
                if (key_temp != No_key)
                {                               // 有按键按下
                    for (i=0; i<7; i++)
                    {dis_buff[i] = dis_buff[i+1];}   // LED 显示左移一位
                    dis_buff[7] = key_temp;          // 最右显示新按下键的键值
                }
            }
        };
    }
```

3）思考与实践

程序中 LED 扫描和定时器的使用与以前的方式相同，每隔 2ms 进行 LED 的扫描，同时每隔 10ms 调用 read_keyboard()键盘接口程序读键盘，并根据返回结果调整 LED 显示的内容。程序本身不复杂，主要体会键盘接口程序的功能和使用，请读者自己分析并实现。

- ✓ 本例中，在 T/C0 的中断服务中进行了 LED 的扫描，而读键盘和键盘处理是在主程序中完成的。如果将读键盘和键盘处理也放在 T/C0 中断中完成是否可以？请深入分析这两种处理方式的优点和缺点，说明原因。
- ✓ 在 read_keyboard()中，行线输出语句为什么重复 2 次？
- ✓ 说明在 read_keyboard()中，key_mask 的作用，另外是否可以将变量 key_line 和 key_value 定义成普通的局部动态变量？为什么？
- ✓ 修改程序，键盘上数字键功能不变，而“#”键的功能为总清除（即清除 LED 上的全部的数字显示，显示复原为 8 个“-”），“*”键的功能为修改键（表示最后输入的数字有误，LED 显示全部左移一位，清除最后输入的数字，最左边一位补入“-”）。

思考与练习

1. 在按键处理过程中，除了要进行消抖处理，还要判别按键的释放，如果不进行按键释放的判别，会发生什么现象？
2. 为什么要使用状态机的程序设计方法？该方法有什么优点？说明其基本原理。
3. 如何使用状态机的程序设计方法来设计嵌入式系统的软件，在程序设计中应注意和掌握

那些原则？

4．说明矩阵式键盘的硬件结构和软件处理思想。

5．认真分析和理解本章中所给出的所有示例，并进行实践，思考和回答所有的问题。

6．简易秒表的设计与实现

使用 6 个数码管组成秒表的显示，两个一组，分别显示分、秒和百分之一秒。使用一个按键，作为秒表的记时控制。秒表初始显示“00.00.00”；第一次按键，秒表启动百分之一秒的记时并显示；第二次按键，秒表停止百分之一秒的记时，显示器显示两次按键之间的相隔时间；第三次按键，秒表复位，显示“00.00.00”。

本章参考文献：

1.《ATmega16 数据手册》（英文，CDROM），ATMEL，www.atmel.com

第 10 章 模拟比较器和ADC接口

模拟比较器和模数转换 ADC 是单片机内部最常见的两种支持模拟信号输入的功能接口。大部分 AVR 都具备这两种类型的接口。本章将以 ATmage16 芯片为例，介绍这两种模拟接口的原理和应用设计方法。

10.1 模拟比较器

ATmega16 的模拟比较器可以实现对两个输入端：正极 AIN0 和负极 AIN1（对应于 ATmage16 的引脚 PB2、PB3）的模拟输入电压进行比较。当 AIN0 上的电压高于 AIN1 的电压时，模拟比较器输出 ACO 被设为“1”。比较器的输出还可以被设置作为定时计数器 1 输入捕获功能的触发信号。此外，比较器的输出可以触发一个独立的模拟比较器中断。用户可以选择使用比较器输出的上升沿、下降沿或事件触发作为模拟比较器中断的触发信号。比较器的方框图和周围电路如图 10-1 所示。

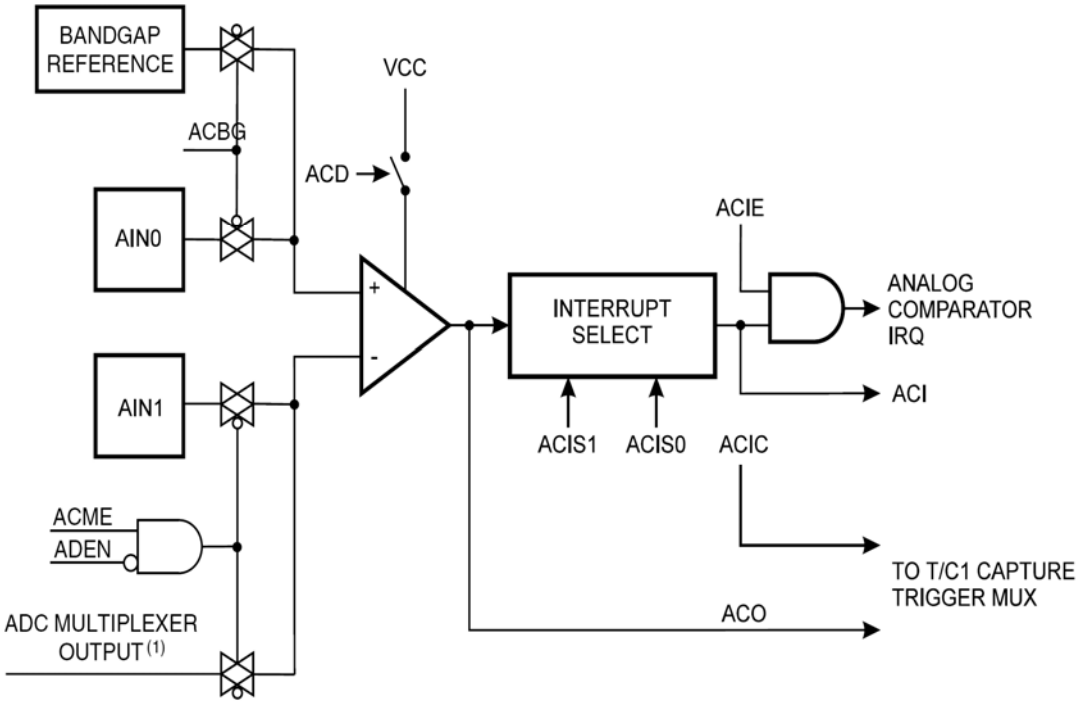

图 10-1 模拟比较器的方框图

10.1.1 与模拟比较器相关的寄存器和标志位

与模拟比较器相关的寄存器是 SFIOR、ACSR。用户通过这两个寄存器的相关位实现对模拟比较器的设置和控制。

1） 特殊功能 IO 寄存器—SFIOR

位	7	6	5	4	3	2	1	0	
$30（$0050）	ADTS2	ADTS1-	ADTS0-	-	ACME	PUD	PSR2	PSR10	SFIOR
读/写	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
复位值	0	0	0	0	0	0	0	0	

寄存器 SFIOR 中的第 3 位 ACME 为模拟比较器多路使能控制位。当该位为逻辑“1”，同时模数转换（ADC）功能被关闭（ADCSRA 寄存器中的 ADEN 使能位为“0”）时，允许使用 ADC 多路复用器选择 ADC 的模拟输入端口作为模拟比较器反向端的输入信号源。当该位为零时，AIN1 引脚的信号将加到模拟比较器反向端。

2） 模拟比较器控制和状态寄存器—ACSR

位	7	6	5	4	3	2	1	0	
$08（$0028）	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
复位值	0	0	N/A	0	0	0	0	0	

ACSR 是模拟比较器主要的控制寄存器，其中各个位的作用如下：

- 位 7—ACD：模拟比较器禁止

当该位设为“1”时，提供给模拟比较器的电源关闭。该位可以在任何时候被置位，从而关闭模拟比较器。在 MCU 闲置模式，且无需将模拟比较器作为唤醒源的情况下，关闭模拟比较器可以减少电源的消耗。要改变 ACD 位的设置时，应该先将寄存器 ACSR 中的 ACIE 位清零，把模拟比较器中断禁止掉。否则，在改变 ADC 位设置时会产生一个中断。

- 位 6—ACBG：模拟比较器的能隙参考源选择

当该位为“1”时，芯片内部一个固定的能隙（Bandgap）参考电源 1.22V 将代替 AIN0 的输入，作为模拟比较器的正极输入端。当该位被清零时，AIN0 的输入仍然作为模拟比较器的正极输入端。

- 位 5—ACO：模拟比较器输出

模拟比较器的输出信号经过同步处理后直接与 ACO 相连。由于经过同步处理，ACO 与模拟比较器的输出之间，会有 1~2 个时钟的延时。

- 位 4—ACI：模拟比较器中断标志位

当模拟比较器的输出事件符合中断触发条件时（中断触发条件由 ACIS1 和 ACIS0 定义），ACI 由硬件置“1”。若 ACIE 位置“1”，且状态寄存器中的 I 位为“1”时，MCU 响应模拟比较器中断。当转入模拟比较中断处理向量时，ACI 被硬件自动清空。此外，也可使用软件方式清零 ACI：对 ACI 标志位写入逻辑“1”来清零该位。

- 位 3—ACIE：模拟比较器中断允许

当 ACIE 位设为“1”，且状态寄存器中的 I 位被设为“1”时，允许模拟比较器中断触发。当 ACIE 被清“0”时，模拟比较器中断被禁止。

- 位 2—ACIC：模拟比较器输入捕获允许

当该位设置为“1”时，定时计数器 1 的输入捕获功能将由模拟比较器的输出来触发。在这种情况下，模拟比较器的输出直接连到输入捕获前端逻辑电路，从而能利用定时器/计数器 1 输入捕获中断的噪声消除和边缘选择的特性。当该位被清零时，模拟比较器和输入捕获功能之间没有联系。要使能比较器触发定时器/计数器 1 的输入捕获中断，定时器中断屏蔽寄存器（TIMSK）中 的 TICIE1 位必须被设置。

- 位 1、0—ACIS1、ACIS0：模拟比较器中断模式选择

这 2 个位决定哪种模拟比较器的输出事件可以触发模拟比较器的中断。不同的设置参见

表 10-1。

表 10-1　模拟比较器中断模式选择

ACIS1	ACIS0	中 断 模 式
0	0	比较器输出的上升沿和下降沿都触发中断
0	1	保留
1	0	比较器输出的下降沿触发中断
1	1	比较器输出的上升沿触发中断

注意：当要改变 ACIS1、ACIS0 时，必须先清除 ACSR 寄存器中的中断允许位，以禁止模拟比较器中断；否则，当这些位被改变时，会发生中断。

3） 模拟比较器的多路输入

用户可以选择 ADC7..0 引脚中的任一路的模拟信号代替 AIN1 引脚，作为模拟比较器的反向输入端。模数转换的 ADC 多路复用器提供这种选择的能力，但此时必须关闭芯片的 ADC 功能。当模拟比较器的多路选择使能位（SFIOR 中的 ACME 位）置“1”，同时 ADC 被关闭时（ADCSRA 中的 ADEN 位置“0”），由寄存器 ADMUX 中的 MUX[2:0]位所确定的引脚将代替 AIN1 作为模拟比较器的反向输入端，如表 10-2 所示。如果 ACME 被清零，或 ADEN 被置 1，则 AIN1 仍将为模拟比较器的反向输入端。

表 10-2　模拟比较器多路输入选**择**

ACME	ADEN	MUX2..0	模拟比较器反向输入端
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6
1	0	111	ADC7

10.1.2 模拟比较器的应用设计

模拟比较器的基本应用就是对两个输入端（AIN0、AIN1）的模拟电压进行比较，例如对系统电源电压的监测等。

例 10.1　系统电源电压的监测

1） 硬件电路

在一些使用电池供电的便携和手持式系统中，系统需要对电源电压进行监测，一旦电压低于某个值时，就要给出警告，提示用户更换电池或对电池进行充电。图 10-2 是一个简单的电源电压监测电路。电源电压经过 R1、R2 分压后，作为监测电压输入端与 PB3（AIN1）连接。模拟比较器的 AIN0 采用芯片内部 1.22V 的固定能隙（Bandgap）参考电源作为比较参考电压。

假定系统正常工作电压范围为 3V-5V，当电源电压低于 3.6V 时就要给出低电压提示。图中使用 PB2 控制一个 LED 发光作为低电压提示。当电源电压高于 3.7V 时，PB3（AIN1）引

脚的电压大于 1.22V，比 AIN0 的 1.22V 高，此时寄存器 ACSR 中的 ACO 为“0”。而当电源电压低于 3.6V 时，PB3（AIN1）引脚的电压降到 1.2V 以下，比 AIN0 的 1.22V 低，此时寄存器 ACSR 中的 ACO 为“1”。因此，ACO 标志位反映了电压高低的情况。

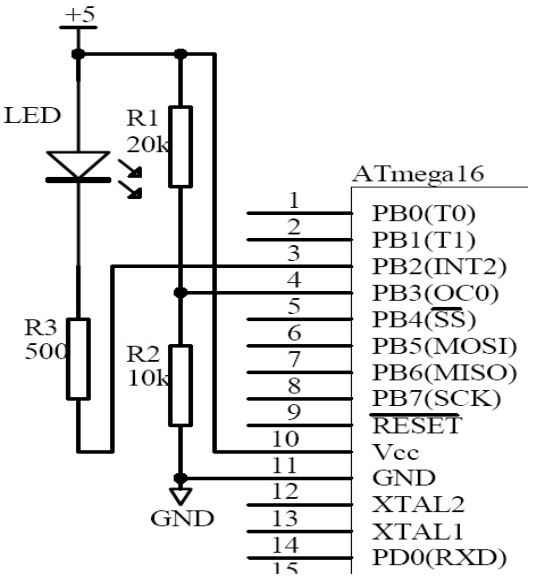

图 10-2 系统电源电压监测电路

2） 软件设计

下面给出一个简单的电源监测程序，程序循环检测 ACO 的值，当 PB3 的电压低于 1.22V 时，PB2 输出低电平，LED 发光，表示低电压报警提示。

```
/*********************************************
File name           : demo_10_1.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/
#include <mega16.h>

void main(void)
{
    PORTB.2 = 1;      //PB2 设置为输出，控制 LED
    DDRB.2 = 1;

    // 模拟比较器初始化：
    ACSR=0x40;        //允许模拟比较器，AIN0 设置为内部 Bandgap 参考电压 1.22V

    while (1)         //循环检测 ACO 位
    {
```

```
            if (ACSR.5)
                PORTB.2 = 0;    // AIN0 < AIN1
            else
                PORTB.2 = 1;    // AIN0 > AIN1,低电压报警
        }
    }
```

在上面简单程序中使用了模拟比较器的功能，实现了对电源电压的监测。在 AVR-51 多功能实验板上模拟时，PB3（AIN1）的输入电压可以通过板上 D 区获得。用连接线直接将 PB3 与 JD1 连接，通过调节电位器 WD1 的阻值，观察 LED 的现象。当 LED 刚发光时，测量 JD1 的电压在 2.2V 左右。

3） 模拟比较器使用注意点

- 芯片 RESET 后，模拟比较器为允许工作状态。如果系统中不使用模拟比较器功能，应将寄存器 ACSR 的 ACD 位置 1，关闭模拟比较器，这样可以减少电源的消耗。
- 使用模拟比较器时，应注意比较器的两个输入端口 PB2、PB3 的设置。当 PB2/PB3 作为模拟输入端使用时，PB2/PB3 应设置为输入工作方式，且上拉电阻无效，这样就不会使 PB2/PB3 上输入的模拟电压受到影响。
- 当 AIN0 设置为使用芯片内部 1.22V 的固定能隙（Bandgap）参考电源时，PB2 口仍然可以作为通用 I/O 端口使用，这样就能节省一个 I/O 引脚。在上面的例子里，AIN0 就是设置为使用芯片内部 1.22V 的固定能隙（Bandgap）参考电源，这样就可将 PB2 口释放出来，作为普通 I/O 口用来驱动 LED 了。

例 10.2　利用模拟比较器构成 ADC

更巧妙的例子是可以利用模拟比较器和一些简单的外围电路，设计构成一个 ADC 转换系统。感兴趣的读者可以参看本章提供的参考文献 avr_app_400.pdf。

10.2 模数转换器ADC

外部的模拟信号量需要转变成数字量才能进一步的由 MCU 进行处理。ATmega16 内部集成有一个 10 位逐次比较（successive approximation）ADC 电路。因此使用 AVR 可以非常方便的处理输入的模拟信号量。

ATmega16 的 ADC 与一个 8 通道的模拟多路选择器连接，能够对以 PORTA 作为 ADC 输入引脚的 8 路单端模拟输入电压进行采样，单端电压输入以 0V（GND）为参考。另外还支持 16 种差分电压输入组合，其中 2 种差分输入方式（ADC1，ADC0 和 ACD3，ADC2）带有可编程增益放大器，能在 A/D 转换前对差分输入电压进行 0dB（1×），20dB（10×）或 46dB（200×）的放大。还有七种差分输入方式的模拟输入通道共用一个负极（ADC1），此时其它任意一个 ADC 引脚都可作为相应的正极。若增益为 1×或 10×，则可获得 8 位的精度。如果增益为 200×，那么转换精度为 7 位。

10.2.1　10 位ADC结构

AVR 的模数转换器 ADC 具有下列特点：

- 10 位精度；
- 0.5LSB 积分非线形误差
- ±2LSB 的绝对精度；
- 13μs~260μs 的转换时间；

- 在最大精度下可达到每秒 15kSPS 的采样速率；
- 8 路可选的单端输入通道；
- 7 路差分输入通道；
- 2 路差分输入通道带有可选的 10×和 200×增益；
- ADC 转换结果的读取可设置为左端对齐（LEFT ADJUSTMENT）；
- ADC 的电压输入范围 0～Vcc；
- 可选择的内部 2.56V 的 ADC 参考电压源；
- 自由连续转换模式和单次转换模式；
- ADC 自动转换触发模式选择；
- ADC 转换完成中断；
- 休眠模式下的噪声抑制器（NOISE CANCELER）。

AVR 的 ADC 功能单元由独立的专用模拟电源引脚 AVcc 供电。AVcc 和 Vcc 的电压差别不能大于±0.3V。ADC 转换的参考电源可采用芯片内部的 2.56V 参考电源，或采用 AVcc，也可使用外部参考电源。使用外部参考电源时，外部参考电源由引脚 ARFE 接入。使用内部电压参考源时，可以通过在 AREF 引脚外部并接一个电容来提高 ADC 的抗噪性能。

ADC 功能单元包括采样保持电路，以确保输入电压在 ADC 转换过程中保持恒定。ADC 方框图如图 10-3 所示。

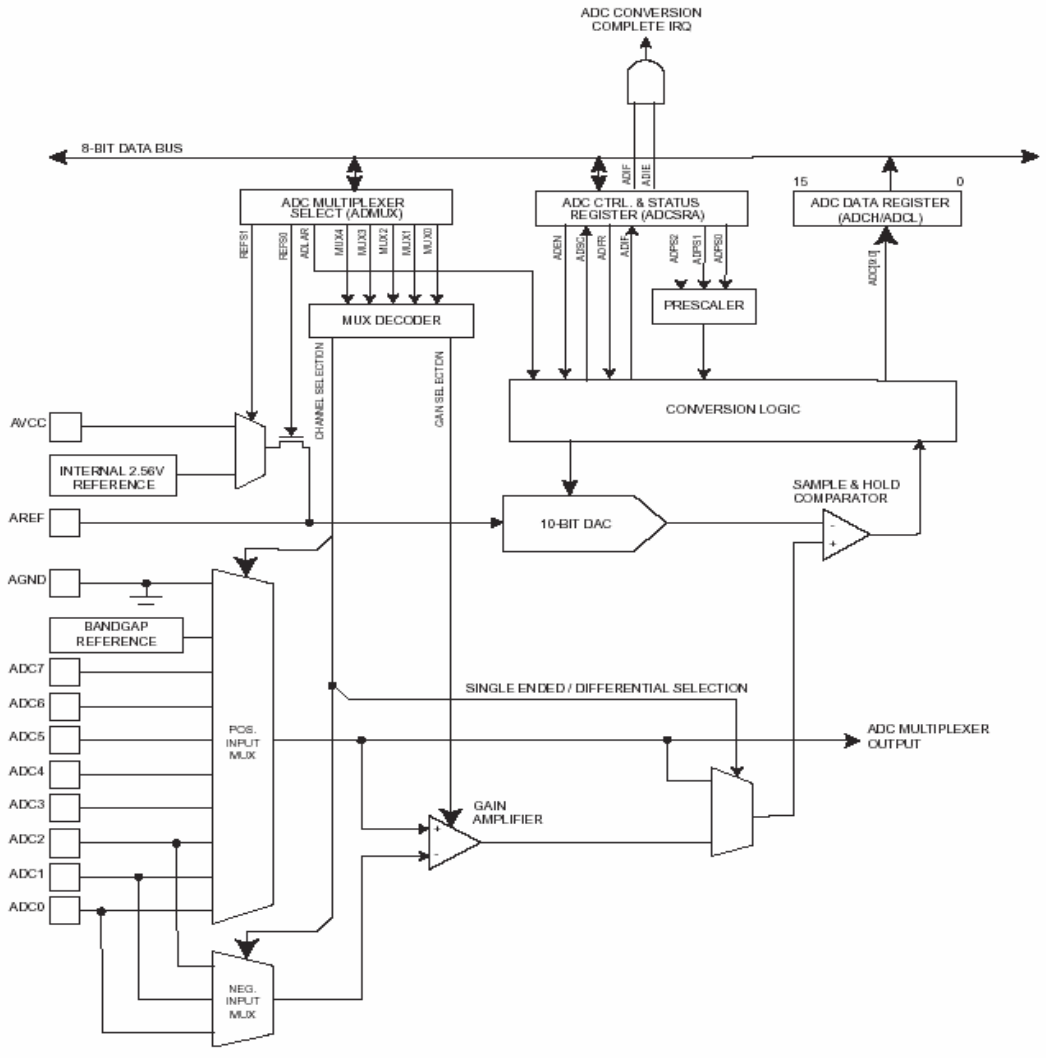

图 10-3 ADC 功能单元方框图

ADC 通过逐次比较（successive approximation）方式，将输入端的模拟电压转换成 10 位的数字量。最小值代表地，最大值为 AREF 引脚上的电压值减 1 个 LSB。可以通过 ADMUX 寄存器中 REFSn 位的设置，选择将芯片内部参考电源（2.56V）或 AVcc 连接到 AREF，作为 A/D 转换的参考电压。这时，内部电压参考源可以通过外接于 AREF 引脚的电容来稳定，以改进抗噪特性。

模拟输入通道和差分增益的选择是通过 ADMUX 寄存器中的 MUX 位设定的。任何一个 ADC 的输入引脚，包括地（GND）以及内部的恒定能隙（fixed bandgap）电压参考源，都可以被选择用来作为 ADC 的单端输入信号。而 ADC 的某些输入引脚则可选择作为差分增益放大器的正、负极输入端。当选定了差分输入通道后，差分增益放大器将两输入通道上的电压差按选定增益系数放大，然后输入到 ADC 中。若选定使用单端输入通道，则增益放大器无效。

通过设置 ADCSRA 寄存器中的 ADC 使能位 ADEN 来使能 ADC。在 ADEN 没有置“1”前，参考电压源和输入通道的选定将不起作用。当 ADEN 位清“0”后，ADC 将不消耗能量，因此建议在进入节电休眠模式前将 ADC 关掉。

ADC 将 10 位的转换结果放在 ADC 数据寄存器中（ADCH 和 ADCL）。默认情况下，转换结果为右端对齐（RIGHT ADJUSTED）的。但可以通过设置 ADMUX 寄存器中 ADLAR 位，调整为左端对齐（LEFT ADJUSTED）。如果转换结果是左端对齐，并且只需要 8 位的精度，那么只需读取 ADCH 寄存器的数据作为转换结果就达到要求了。否则，必须先读取 ADCL 寄存器，然后再读取 ADCH 寄存器，以保证数据寄存器中的内容是同一次转换的结果。因为一旦 ADCL 寄存器被读取，就阻断了 ADC 对 ADC 数据寄存器的操作。这就意味着，一旦指令读取了 ADCL，那么必须紧接着读取一次 ADCH；如果在读取 ADCL 和读取 ADCH 的过程中正好有一次 ADC 转换完成，ADC 的 2 个数据寄存器的内容是不会被更新的，该次转换的结果将丢失。只有当 ADCH 寄存器被读取后，ADC 才可以继续对 ADCL 和 ADCH 寄存器操作更新。

ADC 有自己的中断，当转换完成时中断将被触发。尽管在顺序读取 ADCL 和 ADCH 寄存器过程中，ADC 对 ADC 数据寄存器的更新被禁止，转换的结果丢失，但仍会触发 ADC 中断。

10.2.2 ADC相关的I/O寄存器

1．ADC 多路复用器选择寄存器—ADMUX

位	7	6	5	4	3	2	1	0	
$07 ($0027)	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
复位值	0	0	0	0	0	0	0	0	

- 位 7，6—REFS[1:0]：ADC 参考电源选择

REFS1、REFS2 用于选择 ADC 的参考电压源，见表 10-3。如果这些位在 ADC 转换过程中被改变，新的选择将在该次 ADC 转换完成后（ADCSRA 中的 ADIF 被置位）才生效。一旦选择内部参考源（AVcc、2.56V）为 ADC 的参考电压后，AREF 引脚上不得施加外部的参考电源，只能与 GND 之间并接抗干扰电容。

表 10-3 ADC 参考电源选择

REFS1	REFS0	ADC 参考电源
0	0	外部引脚 AREF，断开内部参考源连接
0	1	AVcc，AREF 外部并接电容
1	0	保留
1	1	内部 2.56V，AREF 外部并接电容

- 位 5—ADLAR：ADC 结果左对齐选择

ADLAR 位决定转换结果在 ADC 数据寄存器中的存放形式。写“1”到 ADLAR 位，将使转换结果左对齐（LEFT ADJUST）；否则，转换结果为右对齐（RIGHT ADJUST）。无论 ADC 是否正在进行转换，改变 ADLAR 位都将会立即影响 ADC 数据寄存器。

- 位 4..0—MUX4:0：模拟通道和增益选择

这 5 个位用于对连接到 ADC 的输入通道和差分通道的增益进行选择设置，详见表 10-4。注意，只有转换结束后（ADCSRA 的 ADIF 是“1”），改变这些位才会有效。

表 10-4　ADC 输入通道和增益选择

MUX[4:0]	单端输入	差分正极输入	差分负极输入	增益
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	N/A	ADC0	ADC0	10×
01001		ADC1	ADC0	10×
01010		ADC0	ADC0	200×
01011		ADC1	ADC0	200×
01100		ADC2	ADC2	10×
01101		ADC3	ADC2	10×
01110		ADC2	ADC2	200×
01111		ADC3	ADC2	200×
10000		ADC0	ADC1	1×
10001		ADC1	ADC1	1×
10010		ADC2	ADC1	1×
10011		ADC3	ADC1	1×
10100		ADC4	ADC1	1×
10101		ADC5	ADC1	1×
10110		ADC6	ADC1	1×
10111		ADC7	ADC1	1×
11000		ADC0	ADC2	1×
11001		ADC1	ADC2	1×
11010		ADC2	ADC2	1×
11011		ADC3	ADC2	1×
11100		ADC4	ADC2	1×
11101		ADC5	ADC2	1×

11110	1.22V(V_{BG})	N/A
11111	0V(GND)	

2．ADC 控制和状态寄存器 A—ADCSRA

位	7	6	5	4	3	2	1	0	
$06（$0026）	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
复位值	0	0	0	0	0	0	0	0	

- 位 7—ADEN：ADC 使能

该位写入“1”时使能 ADC，写入“0”关闭 ADC。如在 ADC 转换过程中将 ADC 关闭，该次转换随即停止。

- 位 6—ADSC：ADC 转换开始

在单次转换模式下，置该位为“1”，将启动一次转换。在自由连续转换模式下，该位写入“1”将启动第一次转换。先置位 ADEN 位使能 ADC，再置位 ADSC；或置位 ADSC 的同时使能 ADC，都会使能 ADC 开始进行第一次转换。第一次 ADC 转换将需要 25 个 ADC 时钟周期，而不是常规转换的 13 个 ADC 时钟周期，这是因为第一次转换需要完成对 ADC 的初始化。

在 ADC 转换的过程中，ADSC 将始终读出为“1”。当转换完成时，它将转变为“0”。强制写入“0”是无效的。

- 位 5—ADATE：ADC 自动转换触发允许

当该位被置为“1”时，允许 ADC 工作在自动转换触发工作模式下。在该模式下，在触发信号的上升沿时 ADC 将自动开始一次 ADC 转换过程。ADC 的自动转换触发信号源由 SFIOR 寄存器中的 ADTS 位选择确定。

- 位 4—ADIF：ADC 中断标志位

当 ADC 转换完成并且 ADC 数据寄存器被更新后该位被置位。如果 ADIE 位（ADC 转换结束中断允许）和 SREG 寄存器中的 I 位被置“1”，ADC 中断服务程序将被执行。ADIF 在执行相应的中断处理向量时被硬件自动清零。此外，ADIF 位可以通过写入逻辑“1”来清零。

- 位 3—ADIE：ADC 中断允许

当该位和 SREG 寄存器中的 I 位同时被置位时，允许 ADC 转换完成中断。

- 位 2，0—ADPS[2:0]：ADC 预分频选择

这些位决定了 XTAL 时钟与输入到 ADC 的 ADC 时钟之间分频数，见表 10-5。

表 10-5　ADC 时钟分频

ADPS[2:0]	分　频　系　数
000	2
001	2
010	4
011	8
100	16
101	32
110	64
111	128

3．ADC 数据寄存器—ADCL 和 ADCH

- ADLAR = 0，ADC 转换结果右对齐时，ADC 结果的保存方式

位	15	14	13	12	11	10	9	8	
$05（$0025）	-	-	-	-	-	-	ADC9	ADC8	ADCH
$04（$0024）	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
位	7	6	5	4	3	2	1	0	
读/写	R	R	R	R	R	R	R	R	
读/写	R	R	R	R	R	R	R	R	
复位值	0	0	0	0	0	0	0	0	
复位值	0	0	0	0	0	0	0	0	

- ADLAR = 1，ADC 转换结果左对齐时，ADC 结果的保存方式

位	15	14	13	12	11	10	9	8	
$05（$0025）	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
$04（$0024）	ADC1	ADC0	-	-	-	-	-	-	ADCL
位	7	6	5	4	3	2	1	0	
读/写	R	R	R	R	R	R	R	R	
读/写	R	R	R	R	R	R	R	R	
复位值	0	0	0	0	0	0	0	0	
复位值	0	0	0	0	0	0	0	0	

当 ADC 转换完成后，可以读取 ADC 寄存器的 ADC0-ADC9 得到 ADC 的转换的结果。如果是差分输入，转换值为二进制的补码形式。一旦开始读取 ADCL 后，ADC 数据寄存器就不能被 ADC 更新，直到 ADCH 寄存器被读取为止。因此，如果结果是左对齐（ADLAR=1），且不需要大于 8 位的精度的话，仅仅读取 ADCH 寄存器就足够了。否则，必须先读取 ADCL 寄存器，再读取 ADCH 寄存器。ADMUX 寄存器中的 ADLAR 位决定了从 ADC 数据寄存器中读取结果的格式。如果 ADLAR 位为“1”，结果将是左对齐；如果 ADLAR 位为“0”（默认情况），结果将是右对齐。

4. 特殊功能 I/O 寄存器—SFIOR

位	7	6	5	4	3	2	1	0	
$30（$0050）	ADTS2	ADTS1	ADTS0	---	ACME	PUD	PSR2	PSR10	SFIOR
读/写	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
复位值	0	0	0	0	0	0	0	0	

- 位 7..5—ADTS[2:0]：ADC 自动转换触发源选择

当ADCSRA寄存器中的ADATE为“1”，允许ADC工作在自动转换触发工作模式时，这3位的设置用于选择ADC的自动转换触发源。如果禁止了ADC的自动转换触发(ADATE为“0”)，这3个位的设置值将不起任何作用。

表 10-6 ADC 自动转换触发源的选择

ADTS[2:0]	触 发 源
000	连续自由转换
001	模拟比较器
010	外部中断 0
011	T/C0 比较匹配
100	T/C0 溢出
101	T/C1 比较匹配 B
110	T/C1 溢出

111	T/C1 输入捕捉

10.2.3 ADC应用设计要点

1．预分频与转换时间

在通常情况下，ADC 的逐次比较转换电路要达到最大精度时，需要 50kHz~200kHz 之间的采样时钟。在要求转换精度低于 10 位的情况下，ADC 的采样时钟可以高于 200kHz，以获得更高的采样率。

ADC 模块中包含一个预分频器的 ADC 时钟源（图 10-4），它可以对大于 100KHz 的系统时钟进行分频，以获得合适的 ADC 时钟提供 ADC 使用。预分频器的分频系数由 ADCSRA 寄存器中的 ADPS 位设置的。一旦寄存器 ADCSRA 中的 ADEN 位置“1”（ADC 开始工作），预分频器就启动开始计数。ADEN 位为“1”时，预分频器将一直工作；ADEN 位为“0”时，预分频器一直处在复位状态。

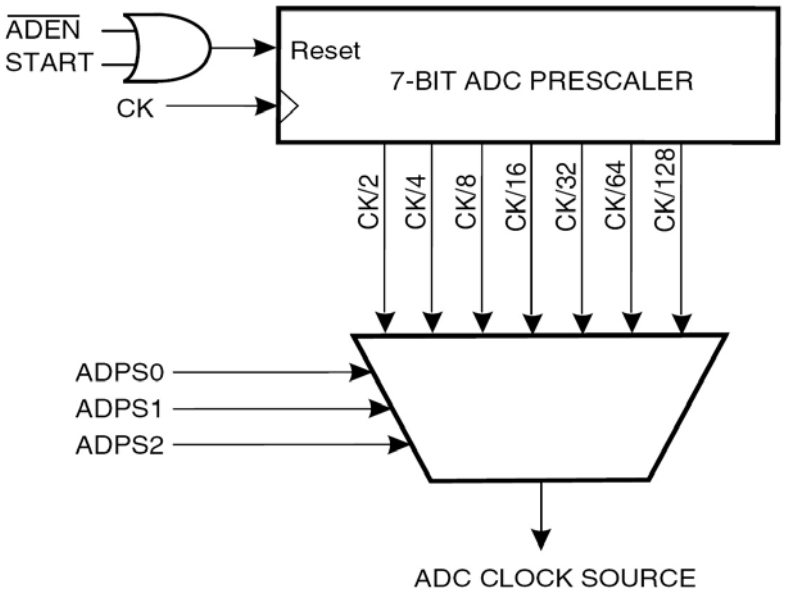

图 10-4　带预分频器的 ADC 时钟源

AVR 的 ADC 完成一次转换的时间见表 10-7。从表中可以看出，完成一次 ADC 转换通常需要 13-14 个 ADC 时钟。而启动 ADC 开始第一次转换到完成的时间需要 25 个 ADC 时钟，这是因为要对 ADC 单元的模拟电路部分进行初始化。

表 10-7　ADC 转换和采样保持时间

转 换 形 式	采样保持时间	完 成 转 换 总 时 间
启动 ADC 后的第一次转换	13.5 个 ADC 时钟	25 个 ADC 时钟
正常转换，单端输入	1.5 个 ADC 时钟	13 个 ADC 时钟
自动触发方式	2 个 ADC 时钟	13.5 个 ADC 时钟
正常转换，差分输入	1.5/2.5 个 ADC 时钟	13/14 个 ADC 时钟

当 ADCSRA 寄存器中的 ADSC 位置位，启动 ADC 转换时，A/D 转换将在随后 ADC 时钟的上升沿开始。一次正常的 A/D 转换开始时，需要 1.5 个 ADC 时钟周期的采样保持时间（ADC 首次启动后需要 13.5 个 ADC 时钟周期的采样保持时间）。当一次 A/D 转换完成后，转换结果写入 ADC 数据寄存器，ADIF（ADC 中断标志位）将被置位。在单次转换模式下，ADSC 也同时被清零。用户程序可以再次置位 ADSC 位，新的一次转换将在下一个 ADC 时钟的上升沿开始。

当 ADC 设置为自动触发方式时，触发信号的上升沿将启动一次 ADC 转换。转换完成的结果将一直保持到下一次触发信号的上升沿出现，然后开始新的一次 ADC 转换。这就保证了使 ADC 每隔一定的时间间隔进行一次转换。在这种方式下，ADC 需要 2 个 ADC 时钟周期的采样保持时间。

在自由连续转换模式下，一次转换完毕后马上开始一次新的转换，此时，ADSC 位一直保持为“1”。

2．ADC 输入通道和参考电源的选择

寄存器 ADMUX 中的 MUXn 和 REFS1、REFS0 位实际上是一个缓冲器，该缓冲器与一个 MCU 可以随机读取的临时寄存器相连通。采用这种结构，保证了 ADC 输入通道和参考电源只能在 ADC 转换过程中的安全点被改变。在 ADC 转换开始前，通道和参考电源可以不断被更新，一旦转换开始，通道和参考电源将被锁定，并保持足够时间，以确保 ADC 转换的正常进行。在转换完成前的最后一个 ADC 时钟周期（ADCSRA 的 ADIF 位置“1”时），通道和参考电源又开始重新更新。注意，由于 A/D 转换开始于置位 ADSC 后的第一个 ADC 时钟的上升沿，因此，在置位 ADSC 后的一个 ADC 时钟周期内不要将一个新的通道或参考电源写入到 ADMUX 寄存器中。

改变差分输入通道时需特别当心。一旦确定了差分输入通道，增益放大器需要 125μs 的稳定时间。所以在选择了新的差分输入通道后的 125μs 内不要启动 A/D 转换，或将这段时间内转换结果丢弃。通过改变 ADMUX 中的 REFS1、REFS0 来更改参考电源后，第一次差分转换同样要遵循以上的时间处理过程。

- 当要改变 ADC 输入通道时，应该遵守以下方式，以保证能够选择到正确的通道：

在单次转换模式下，总是在开始转换前改变通道设置。尽管输入通道改变发生在 ADSC 位被写入“1”后的 1 个 ADC 时钟周期内，然而，最简单的方法是等到转换完成后，再改变通道选择。

在连续转换模式下，总是在启动 ADC 开始第一次转换前改变通道设置。尽管输入通道改变发生在 ADSC 位被写入“1”后的 1 个 ADC 时钟周期内，然而，最简单的方法是等到第一次转换完成后再改变通道的设置。然而由于此时新一次的转换已经自动开始，所以，当前这次的转换结果仍反映前一通道的转换值，而下一次的转换结果将为新设置通道的值。

- ADC 电压参考源

ADC 的参考电压（V_{REF}）决定了 A/D 转换的范围。如果单端通道的输入电压超过 V_{REF}，将导致转换结果接近于 0x3FF。ADC 的参考电压 V_{REF} 可以选择为 AV_{CC} 或芯片内部的 2.56V 参考源，或者为外接在 AREF 引脚上的参考电压源。

AV_{CC} 通过一个无源开关连接到 ADC。内部 2.56V 参考源是由内部能隙参考源（V_{BC}）通过内部的放大器产生的。注意，无论选用什么内部参考电源，外部 AREF 引脚都是直接与 ADC 相连的，因此，可以通过外部在 AREF 引脚和地之间并接一个电容，使内部参考电源更加稳定和抗噪。可以通过使用高阻电压表测量 AREF 引脚，来获得参考电源 V_{REF} 的电压值。由于 V_{REF} 是一个高阻源，因此，只有容性负载可以连接到该引脚。

如果将一个外部固定的电压源连接到 AREF 引脚，那就不能使用任何的内部参考电源，否则就会使外部电压源短路。外部参考电源的范围应在 2.0V 到 AV_{CC}-0.2V 之间。参考电源改变后的第一次 ADC 转换结果可能不太准确，建议抛弃该次转换结果。

3．ADC 转换结果

A/D 转换结束后（ADIF = 1），在 ADC 数据寄存器（ADCL 和 ADCH）中可以取得转换的结果。对于单端输入的 A/D 转换，其转换结果为：

$$ADC = (V_{IN} \times 1024) / V_{REF}$$

其中 V_{IN} 表示选定的输入引脚上的电压，V_{REF} 表示选定的参考电源的电压。0x000 表示输入

引脚的电压为模拟地，0x3FF 表示输入引脚的电压为参考电压值减去一个 LSB。

对于差分转换，其结果为：

$$ADC=(V_{POS}-V_{NEG}) \times GAIN \times 512/V_{REF}$$

例：若差分输入通道选择为 ADC3-ADC2，10 倍增益，参考电压 2.56V，左端对齐（ADMUX=0xED），ADC3 引脚上电压 300mV，ADC2 引脚上电压 500mV。

则 ADCR =（300-500）×10×512 / 2560 = -400 = 0x270

ADCL=0x00，ADCH=0x9C。

若结果为右端对齐时（ADLAR=“0”），则 ADCL=0x70，ADCH=0x02。

10.2.4 ADC的应用设计

例 10.3 简易电压表的实际与实现

1） 硬件电路

利用 AVR-51 多功能实验板，可以实现和完成一个简易电压表的设计，电路图为 10-5 所示。图中 ATmega16 的 PC 口作为 LED 的段码输出口，PA0-PA4 为 4 位 LED 的位控，采用动态扫描方式构成电压表的输出显示。而 PA7（ADC7）口作为模拟电压测量的输入口（ADC 输入）。系统 5v 电源经过 L、C2 滤波后到 AVcc，提高了 AVcc 的稳定性。ADC 的参考电压源采用 AVcc，电容 C2 并接在 AREF 和地之间也进一步的提高参考电压的稳定性。

调节可变电阻 W 的阻值，在 PA7 端可以得到在 0-5v 之间变化的电压值，PA7 为单端输入方式，利用 ATmega16 内部的 ADC 进行转换，转换后的结果换算成测量的电压值在 4 位 LED 上显示。

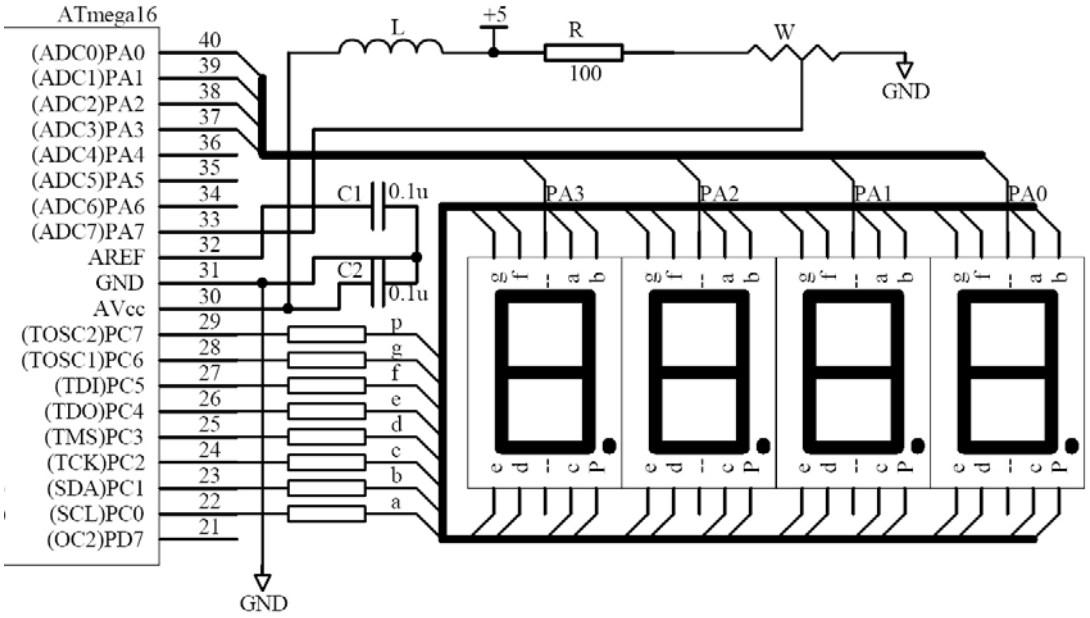

图 10-5 简易 0-5v 电压表电路

2） 软件设计

下面是实现简易电压的系统程序代码。

```
/*****************************************************
File name           : demo_10_3.c
Chip type           : ATmega16L
```

```
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************************/

#include <mega16.h>

flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[6]={0xfe,0xfd,0xfb,0xf7};

unsigned char dis_buff[4]={0,0,0,0},posit;
bit time_2ms_ok;

// ADC 电压值送显示缓冲区函数
void adc_to_disbuffer(unsigned int adc)
{
     char i;
     for (i=0;i<=3;i++)
     {
          dis_buff[i]=adc % 10;
          adc /= 10;
     }
}

// Timer 0 比较匹配中断服务
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
     time_2ms_ok = 1;
}

// ADC 转换完成中断服务
interrupt [ADC_INT] void adc_isr(void)
{
    unsigned int adc_data,adc_v;
    adc_data=ADCW;                    //读取 ADC 置换结果
    adc_v=(unsigned long)adc_data*5000/1024;    //换算成电压值
    adc_to_disbuffer(adc_v);
}

void display(void) // 4 位 LED 数码管动态扫描函数
{
    PORTA |= 0x0f;
```

```
    PORTC = led_7[dis_buff[posit]];
    if (posit==3) PORTC |= 0x80;
    PORTA &= position[posit];
    if (++posit >=4 ) posit = 0;
}
// 系统主程序
void main(void)
{
    DDRA=0x0f;    // !! 见以下说明
    PORTA=0x0f;   // !! 见以下说明
    DDRC=0xff;    // LED 显示控制 I/O 端口初始化
    PORTC=0x00;
    // T/C0 初始化
    TCCR0=0x0B;   // 内部时钟，64 分频（4M/64=62.5KHz），CTC 模式
    TCNT0=0x00;
    OCR0=0x7C;    // OCR0 = 0x7C(124),(124+1)/62.5=2ms
    TIMSK=0x02;   // 允许 T/C0 比较中断

    // ADC 初始化
    ADMUX=0x47;     // 参考电源 AVcc、ADC7 单端输入
    SFIOR&=0x1F;
    SFIOR|=0x60;    // 选择 T/C0 比较匹配中断为 ADC 触发源
    ADCSRA=0xAD;    // ADC 允许、自动触发转换、ADC 转换中断允许、ADCclk=125Kz

    #asm("sei")     // 开放全局中断

    while (1)
    {
        if (time_2ms_ok)
        {
              display();    // LED 扫描显示
              time_2ms_ok = 0;
        }
    }
}
```

程序中采用 T/C0 比较匹配中断，每 2ms 中断一次。该定时中断除了作为 LED 动态扫描的定时外，还作为 ADC 自动触发转换的触发源信号。在 ADC 的初始化代码中，设置 ADC 时钟的分频系数为 32。系统 4M 时钟经过 32 分频产生 125KHz 的 ADC 时钟，满足了逐次比较转换电路达到最大精度时，需要的 50kHz~200kHz 之间的采样时钟的要求。ADC 单端输入转换时间为 13 个 ADC 时钟周期，则一次 ADC 转换的时间为 13/125KHz = 0.11ms。因此，2ms 的固定转换间隔时间远超出完成一次 ADC 的转换时间 0.11ms，不会影响 ADC 的转换过程，同时每秒内完成的 ADC 转换达 500 次。

尽管 ATmega16 的 PA 口的 PA7 作为 ADC 的输入端，PA 口的其它引脚仍可作为普通的数

字 I/O 口使用，本例就是使用 PA0-PA3 作为 4 个 LED 的位控制线使用。但对 PA 口的初始化时需要注意，PA7 要设置成输入方式，且不能使用该口内部的上拉电阻，否则会影响到输入的模拟电压值。

在 ADC 转换完成中断服务中，把 ADC 转换结果换算成电压值，换算采用了整型数计算。为了保证计算产生不溢出，先将 adc_data 强行转换成长型，然后再乘 5000（这里假定 AVcc 参考电压为 5v），最后再除 1024，保证了换算的正确型。

10.2.5 ADC的应用设计的深入讨论

尽管 AVR 内部集成了 10 位的 ADC，但是在实际应用中，要想真正实现 10 位精度，比较稳定的 ADC 的话，并不象上一节中的例子那么简单。需要进一步从硬件、软件等方面进行综合的、细致的考虑。下面介绍一些在 ADC 设计应用中应该考虑的几个要点。

1．AVcc 的稳定性。

AVcc 是提供给 ADC 工作的电源，如果 AVcc 不稳定，就会影响 ADC 的转换精度。在图 10-5 中，系统电源通过一个 LC 滤波后接入 AVcc，这样就能很好的抑制掉系统电源中的高频躁声，提高了 AVcc 的稳定性。另外在要求比较高的场合使用 ADC 时，PA 口上的那些没被用做 ADC 输入的端口尽量不要作为数字 I/O 口使用。因为 PA 口的工作电源是由 AVcc 提供的，如果 PA 口上有比较大的电流波动，也会影响 AVcc 的稳定。

2．参考电压 V_{REF} 的选择确定

在实际应用中，要根据输入测量电压的范围选择正确的参考电压 V_{REF}，以求得到比较好的转换精度。ADC 的参考电压 V_{REF} 还决定了 A/D 转换的范围。如果单端通道的输入电压超过 V_{REF}，将导致转换结果全部接近于 0x3FF，因此 ADC 的参考电压应稍大于模拟输入电压的最高值。

ADC 的参考电压 V_{REF} 可以选择为 AV_{CC}，或芯片内部的 2.56V 参考源，或者为外接在 AREF 引脚上的参考电压源。外接参考电压应该稳定，并大于 2.0V（芯片的工作电压为 1.8V 时，外接参考电压应大于 1.0V）。要求比较高的场合，建议在 AREF 引脚外接标准参考电压源来作为 ADC 的参考电源。

3. ADC 通道带宽和输入阻抗

不管使用单端输入转换还是差分输入转换方式，所有模拟输入口的输入电压应在 AVcc-GNG 之间。基本的接入方式如图 10-6 所示。

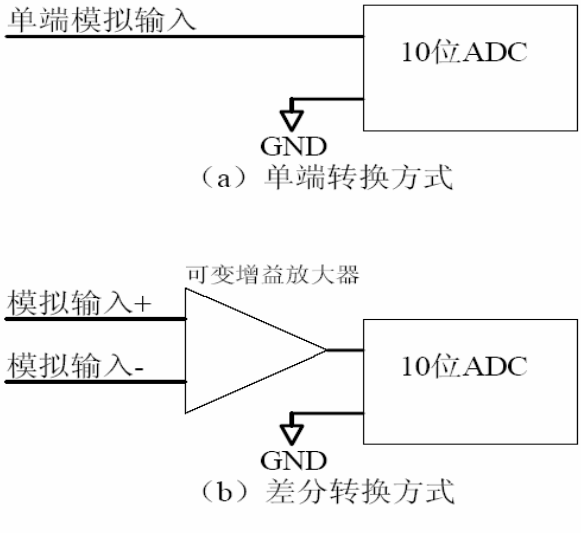

图 5.8 单端 ADC 和差分 ADC 转换输入示意图

在单端 ADC 转换方式时，ADC 通道的输入频率带宽取决于 ADC 转换时钟频率。一次常规的 ADC 转换需要 13 个 ADC 时钟，当 ADC 转换时钟为 1MHz 时，一秒种内 ADC 采样转换的次数约 77K。根据采样定理，此时 ADC 通道的带宽为 38.5KHz。

差分方式 ADC 转换的带宽是由芯片内部的差分放大器的带宽决定，为 4KHz。

AVR 的 ADC 输入阻抗典型值为 100MΩ，为保证测量的准确，被测信号源的输出阻抗要尽可能的低，应在 10K 以下。

4. ADC 采样时钟的选择

通常条件下，AVR 的 ADC 逐次比较电路要达到转换的最大精度，需要一个 50K～200KHz 的采样时钟。一次正常的 ADC 转换过程需要 13 个采样时钟，假定 ADC 采样时钟为 200KHz，那么最高的采样速率为 200K/13=15.384K。因此根据采样定理，理论上被测模拟信号的最高频率为 7.7K！

尽管可以设置 ADC 的采样时钟为 1M，但并不能提高 ADC 转换精度，反而会降低转换精度（受逐次比较硬件电路的限制），因此 AVR 的 ADC 不能完成高速 ADC 的任务。如果所需的转换精度低于 10 位，那么采样时钟可以高于 200KHz，以达到更高的采样频率。

ADC 采样时钟的选择方式为：给出或估计被测模拟信号的最高频率 fs，取采样频率为 fs 的 4-10 倍，再乘上 13 为 ADC 采样时钟频率，该频率应在 50K～200KHz 之间。 如果该频率大于 200KHz，则 ADC 的 10 位精度不能保证。如果该频率小于 50Khz，则可选择 50K～200KHz 之间的数值。

5．模拟噪声的抑制

器件外部和内部的数字电路会产生电磁干扰，并会影响模拟测量的精度。如果 ADC 转换精度要求很高，可以采用以下的技术来降低噪声的影响：

（1）使模拟信号的通路尽可能的短。模拟信号连线应从模拟地的布线盘上通过，并使它们尽可能远离高速开关数字信号线。

（2）AVR 的 AVcc 引脚应该通过 LC 网络（如图 10-5 所示）与数字端电源 Vcc 相连。

（3）采用 ADC 噪声抑制器功能来降低来自 MCU 内部的噪声。

（4）如果某些 ADC 引脚是作为通用数字输出口使用，那么在 ADC 转换过程中，不要改变这些引脚的状态。

6．ADC 的校正

由于 AVD 内部 ADC 部分的放大器非线性等客观原因，ADC 的转换结果会有误差的。如果要获得高精度的 ADC 转换，还需要对 ADC 结果进行校正。具体的方法请参考 AVR 应用笔记 AVR120（avr_app_120.pdf），在这篇应用设计参考中详细介绍了误差的种类，以及校正方案。

7．ADC 精度的提高

在有了上述几点的保证后，通过软件的手段也能适当的提高 ADC 的精度。如采用多次测量取平均，软件滤波算法等。在 AVR 应用笔记 AVR121（avr_app_121.pdf）中介绍了一种使用过采样算法的软件实现，可以将 ADC 的精度提高到 11 位或更高，当然这是在花费更多的时间基础上实现的。

思考与练习

1．正确的使用 AVR 的 ADC 需要在硬件和软件方面做那些考虑？

2．ADC 的转换精度与那些因素相关？如何能真正的提高 ADC 的转换精度？

3．怎样正确的选择 AVR 的 ADC 时钟频率？

4．ADC 的参考电压和转换结果的精度有何关系？

5．参考本章的 ADC 使用与第 8 章的定时计数器的 PWM 输出配合，设计一个控制系统。系统

功能为：输入直流电压范围 0-2v 时，输出直流电压为 4v；输入直流电压范围 2-4v 时，输出直流电压为 3v；输入直流电压范围大于 4v 时，输出直流电压为 2v。输入电压的测量采用 ADC，输出电压采用产生 PWM 波加低通滤波器的方式获得。

本章参考文献：

1.《ATmega16 数据手册》(英文，CDROM)，ATMEL，www.atmel.com
2. AVR应用笔记AVR128《avr_app_128.pdf》(英文，CDROM)，ATMEL，www.atmel.com
3. AVR应用笔记AVR400《avr_app_400.pdf》(英文，CDROM)，ATMEL，www.atmel.com
4. AVR应用笔记AVR120《avr_app_120.pdf》(英文，CDROM)，ATMEL，www.atmel.com
5. AVR应用笔记AVR121《avr_app_121.pdf》(英文，CDROM)，ATMEL，www.atmel.com

第 11 章 综合实践（二）

本章的综合实践将综合前几章的内容，指导读者完成以下的实践：

- 如何实现频率测量和简单频率计的设计实现。
- 使用 T/C1 的输入捕捉功能实现高精度的频率周期测量
- 完成一个比较完善的实时时钟的设计和实现。

11.1 频率测量和简单频率计的设计

11.1.1 频率测量原理

单片机应用系统中，经常要对一个连续的脉冲波频率进行测量。在实际应用中，对于转速，位移、速度、流量等物理量的测量，一般也是由传感器转换成脉冲电信号，采用测量频率的手段实现。

使用单片机测量频率或周期，通常是利用单片机的定时计数器来完成的，测量的基本方法和原理有两种：

测频法：在限定的时间内（如 1 秒钟）检测脉冲的个数。

测周法：测试限定的脉冲个数之间的时间。

这两种方法尽管原理是相同的，但在实际使用时，需要根据待测频率的范围、系统的时钟周期、计数器的长度、以及所要求的测量精度等因素进行全面和具体的考虑，寻找和设计出适合具体要求的测量方法。

在具体频率的测量中，需要考虑和注意的因素有以下几点。

- ✓ 系统的时钟。首先测量频率的系统时钟本身精度要高，因为不管是限定测量时间还是测量限定脉冲个数的周期，其基本的时间基准是系统本身时钟产生的。其次是系统时钟的频率值，因为系统时钟频率越高，能够实现频率测量的精度也越高。因此使用 AVR 测量频率时，建议使用由外部晶体组成的系统的振荡电路，不使用其内部的 RC 振荡源，同时尽量使用频率比较高的系统时钟。
- ✓ 所使用定时计数器的位数。测量频率要使用定时计数器，定时计数器的位数越长，可以产生的限定时间越长，或在限定时间里记录的脉冲个数越多，因此也提高了频率测量的精度。所以对频率测量精度有一定要求时，尽量采用 16 位的定时计数器。
- ✓ 被测频率的范围。频率测量需要根据被测频率的范围选择测量的方式。当被测频率的范围比较低时，最好采用测周期的方法测量频率。而被测频率比较高时，使用测频法比较合适。需要注意的是，被测频率的最高值一般不能超过测频 MCU 系统时钟频率的 1/2，因为当被测频率高于 MCU 时钟 1/2 后，MCU 往往不能正确检测被测脉冲的电平变化了。

除了以上三个因素外，还要考虑频率测量的频度（每秒内测量的次数），如何与系统中其它任务处理之间的协调工作等。频率测量精度要求高时，还应该考虑其它中断以及中断响应时间的影响，甚至需要在软件中考虑采用多次测量取平均的算法等。

在“AVR-51 多功能实验开发板”的 K 区有一个方波信号源。该区模块使用一个 2.048MHz 的晶体振荡器，经过 CD4060 的分频后，提供了占空比为 50%，125Hz～128KHz 之间 10 种不同频率的标准方波脉冲信号。下面我们介绍 2 个基本的频率测量实例，实现对这些不同频率的信号进行测量。

11.1.2 测频法测量频率

测频法的基本思想，就是采用在已知限定的时间内对被测信号输入的脉冲个数进行计数的方法来实现对信号频率的测量。当被测信号的频率比较高时，采用这种方法比较适合，因为在一定时间内，频率越高，计数脉冲的个数也越多，测量也越准确。

例 11.1　采用测频法的频率计设计与实现

1）　硬件电路

硬件电路的显示部分与图 9-7 相同，PA 口为 8 个 LED 数码管的段输出，PC 口控制 8 个 LED 数码管的位扫描。使用 T/C0 对被测信号输入的脉冲个数进行计数，被测频率信号由 PB0（T0）输入。

2）　软件设计

我们首先给出系统程序，然后做必要的说明。

```
/*****************************************
File name          : demo_11_1.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*****************************************/

#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[8]={0x7f,0xbf,0xdf,0xef,0xf7,0xfb,0xfd,0xfe};

char dis_buff[8];                       // 显示缓冲区，存放要显示的 8 个字符的段码值
char posit;
bit  time_1ms_ok,display_ok=0;
char time0_old,time0_new,freq_time;
unsigned int  freq;

void display(void)                      // 8 位 LED 数码管动态扫描函数
{
     PORTC = 0xff;
     PORTA = led_7[dis_buff[posit]];
     if (posit==5) PORTA = PORTA | 0x80;
     PORTC = position[posit];
     if (++posit >=8 ) posit = 0;
}

// Timer 2 output compare interrupt service routine
interrupt [TIM2_COMP] void timer2_comp_isr(void)
{
```

```
    time0_new = TCNT0;                  // 1ms 到，记录当前 T/C0 的计数值
    time_1ms_ok = 1;
    display_ok = ~display_ok;
    if (display_ok) display();
}

void freq_to_disbuff(void)                  // 将频率值转化为 BCD 码并送入显示缓冲区
{
    char i,j=7;
    for (i=0;i<=4;i++)
    {
        dis_buff[j-i] = freq % 10;
        freq = freq / 10;
    }
    dis_buff[2] = freq;
}

void main(void)
{
    char i;
    DDRA=0xFF;                          // LED 数码管驱动
    DDRC=0xFF;
        // T/C0 初始化，外部计数方式
    TCCR0=0x06;                         // 外部 T0 脚下降沿触发计数，普通模式
    TCNT0=0x00;
    OCR0=0x00;
        // T/C2 初始化
    TCCR2=0x0B;                         // 内部时钟，32 分频（4M/32=125KHz），CTC 模式
    TCNT2=0x00;
    OCR2=0x7C;                          // OCR2 = 0x7C(124),(124+1)/125=1ms

    TIMSK=0x80;                         // 允许 T/C2 比较匹配中断

    for (i=0;i<=7;i++) dis_buff[i] = 0;
    time0_old = 0;

    #asm("sei")                         // 开放全局中断

    while (1)
    {
        if (time_1ms_ok)
        {   // 累计 T/C0 的计数值
            if (time0_new >= time0_old) freq = freq + (time0_new - time0_old);
            else freq = freq + (256 - time0_old + time0_new);
```

```
                time0_old = time0_new;
                if (++freq_time >= 100)
                {
                    freq_time = 0;          // 100ms 到，
                    freq_to_disbuff();      // 将 100ms 内的脉冲计数值送显示
                    freq = 0;
                }
                time_1ms_ok = 0;
            }
        };
    }
```

程序中 LED 扫描形式函数 desplay()，以及脉冲计数值转换成 BCD 码并送显示缓冲区函数 freq_to_disbuff()比较简单，请读者自己分析。

在该程序中，使用了两个定时计数器。T/C0 工作在计数器方式，对外部 T0 引脚输入的脉冲信号计数（下降沿触发）。T/C2 工作在 CTC 方式，每隔 1ms 中断一次，该定时时间即作为 LED 的显示扫描，同时也是限定时间的基时。每一次 T/C2 的中断中，都首先记录下 T/C0 寄存器 TCNT0 当前的计数值，因此前后两次 TCNT0 的差值(time0_new - time0_old)或(256 - time0_old + time0_new)就是 1ms 时间内 T0 脚输入的脉冲个数。为了提高测量精度，程序对 100 个 1ms 的脉冲个数进行了累计（在变量 freq 中），即已知限定的时间为 100ms。

读者还应该注意频率的连续测量与 LED 扫描、BCD 码转换之间的协调问题。T/C2 中断间隔为 1ms，因此在 1ms 时间内，程序必须将脉冲个数进行的累计、BCD 码转换和送入显示缓冲区，以及 LED 的扫描工作完成掉，否则就会影响到下一次中断到来后的处理。

在本实例的 T/C2 中断中，使用了 display_ok 标志，将 LED 扫描分配在奇数 ms（1、3、5、7、……），而将 1ms 的 TCNT0 差值计算、累积和转换等处理放在主程序中完成。另外由于计算量大的 BCD 码转换是在偶数 ms（100ms）处理，所以程序中 LED 的扫描处理和 BCD 码转换处理不会同时进行（不会在两次中断间隔的 1ms 内同时处理 LED 扫描和 BCD 码转换），这就保证了在下一次中断到达时，前一次的处理已经全部完成，使频率的连续测量不受影响。

该实例程序的性能和指标为（假定系统时钟没有误差 = 4MHz）：

- ✓ 频率测量绝对误差：±10Hz。由于限定的时间为 100ms，而且 T/C0 的计数值有±1 的误差，换算成频率为±10Hz。
- ✓ 被测最高频率值：255KHz。由于 T/C0 的长度 8 位，所以在 1ms 中，T0 输入的脉冲个数应小于 255 个，大于 255 后造成 T/C0 的自动清另，丢失脉冲个数。
- ✓ 测量频度：10 次/秒。限定的时间为 100ms，连续测量，所以为 10 次/秒。
- ✓ 使用资源：两个定时器，一个中断。

3） 思考与实践

根据上面采用测频法的思路，如何修改程序提高测量精度和被测最高频率？参考提示如下：

- ✓ 延长限定的时间，如采用 1s，可提高频率的测量精度。但测量频度减小，同时注意变量 freq 应定义为长整型变量。
- ✓ 将 T/C0 换成 16 位的 T/C1，可以提高被测最高频率值。注意此时 time0_new、time0_old 应定义为整型变量。

11.1.3 测周法测量频率

测周法的基本思想，就是测量在限定的脉冲个数之间的时间间隔，然后再换算成频率（需要时）。当被测信号的频率比较低时，采用这种方法比较适合，因为频率越低，在限定的脉

冲个数之间的时间间隔也也长，因此定时计数的个数也越多，测量也越准确。

例 11.2　采用测周法的频率计设计与实现

1） 硬件电路

硬件电路的显示部分与图 9-7 相同，PA 口为 8 个 LED 数码管的段输出，PC 口控制 8 个 LED 数码管的位扫描。被测频率信号由 PB0（T0）输入。

2） 软件设计

我们首先给出系统程序，然后做必要的说明。

```
/*********************************************
File name          : demo_11_2.c
Chip type          : ATmega16
Program type       : Application
Clock frequency    : 4.000000 MHz
Memory model       : Small
External SRAM size : 0
Data Stack size    : 256
*********************************************/

#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[8]={0x7f,0xbf,0xdf,0xef,0xf7,0xfb,0xfd,0xfe};

char dis_buff[8];                  // 显示缓冲区，存放要显示的 8 个字符的段码值
char posit;
bit  freq_ok = 0;
char time2_new;
unsigned int  freq;
unsigned long int freq_temp;

void display(void)                 // 8 位 LED 数码管动态扫描函数
{
    PORTC = 0xff;
    PORTA = led_7[dis_buff[posit]];
    if (posit==5) PORTA = PORTA | 0x80;
    PORTC = position[posit];
    if (++posit >=8 ) posit = 0;
}

// T/C0 比较匹配中断服务，250 个计数脉冲中断一次
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
    time2_new = TCNT2;
    TCNT2 = 0;
    TIFR |= 0x02;
    freq_temp = freq;
    freq = 0;
    freq_ok = 1;
}

// T/C2 比较匹配中断服务，500us 一次
```

```
interrupt [TIM2_COMP] void timer2_comp_isr(void)
{
    freq++;
    #asm("sei")        // 开中断，允许中断嵌套，T/C0 中断可打断该中断服务
    display();
}

void freq_to_disbuff(void)       // 频率值转化为 BCD 码送显示缓冲区
{
    char i,j=7;
    for (i=0;i<=7;i++)
    {
        dis_buff[j-i] = freq_temp % 10;
        freq_temp = freq_temp / 10;
    }
}

void main(void)
{
    char i;
    DDRA=0xFF;                   // LED 数码管
    DDRC=0xFF;
        // T/C2 初始化
    TCCR2=0x0A;                  // 内部时钟，8 分频（4M/8=500KHz），CTC 模式，
    TCNT2=0x00;                  // 基时为 2us
    OCR2=0xF9;                   // OCR2 = 0xF9(249),(249+1)/500 = 0.5ms
        // T/C0 初始化
    TCCR0=0x0E;                  // 外部 T0 脚下降沿触发计数，CTC 模式
    TCNT0=0x00;
    OCR0=0xF9;                   // OCR0 = 0xF9(249),(249 + 1) = 250

    TIMSK=0x82;                  // 允许 T/C2、T/C0 比较匹配中断

    for (i=0;i<=7;i++)dis_buff[i]=0;

    #asm("sei")                  // 开放全局中断

    while (1)
    {
        if (freq_ok)
        {
            freq_temp = freq_temp * 250 + time2_new;  // 累计 250 个脉冲的时间间隔
            freq_temp = 12500000000/freq_temp;        // 换算成频率
            freq_to_disbuff();                        // 频率值送显示
            freq_ok = 0;
        }
    };
}
```

程序中 LED 扫描形式函数 desplay()，以及脉冲计数值转换成 BCD 码并送显示缓冲区函

数 freq_to_disbuff()比较简单，请读者自己分析。

在该程序中，同样使用了两个定时计数器。T/C2 仍旧工作在 CTC 方式，每隔 500us 中断一次，该定时时间即作为 LED 的显示扫描，同时也用于时间累计。在每一次 T/C2 的中断中，将累计中断的次数（在 freq 中），然后马上开放全局中断（由于在进入 T/C0 中断时，系统硬件已经自动关闭了全局中断允许），保证系统能及时响应 T/C0 的中断。

该程序的核心是 T/C0 的中断。T/C0 工作在 CTC 方式，它负责对外部 T0 引脚输入的脉冲信号计数（下降沿触发），一旦计数值（限定脉冲个数）到达 250 产生中断。进入 T/C0 中断后，立即记录当前 T/C2 寄存器 TCNT2 的值（在 time2_new 中），然后清零 TCNT2 和 T/C2 的中断标志位，为下一次计时做初始化准备。接下来同样需要把 T/C2 产生中断的次数累计值备份到 freq_temp 中，此时变量 freq_temp 和 time2_new 中的值就是 T0 输入的 250 个限定脉冲之间的时间间隔。

当 T/C0 中断产生后，系统应该立即响应，马上读取 T/C2 的值。由于 T/C2 的计时过程不会停止，所以拖延 T/C0 中断的响应时间就会影响测量的精度。因此需要把 T/C2 的中断服务程序设计成能够支持中断嵌套的方式，使系统尽可能的立即响应 T/C0 中断。

计算 250 个限定脉冲之间的时间间隔是在主程序中完成的。计算公式为：250 个脉冲之间的时间间隔 = T/C2 中断次数 * 250 + T/C2 当前值（计时时基个数）；1 计时时基个数 = 2us（注：T/C2 计时时基 = 4M/8）。换算成频率值：1000000/（250 个脉冲之间的时间间隔 * 2us/250）* 100 = 12500000000/250 个脉冲之间的时间间隔，单位为 Hz。乘上 100 是为了保留 2 位小数。程序中全部使用了整数运算，它比采用浮点数运算的速度要快的多，同时也保证了在 T/C0 两次中断的间隔中，能全部完成频率换算、LED 扫描等处理任务，不造成对频率连续测量的影响。

该实例程序的性能和指标为（假定系统时钟没有误差 = 4MHz）：

- ✓ 周期测量绝对误差为±(2us/250)。如果不考虑中断响应时间的影响，由于 T/C2 的计数值有±1 的误差，所以周期测量绝对误差为±(2us/250)。如果考虑中断响应时间的影响时，周期测量绝对误差在±(2～5/250)us。
- ✓ 被测最低频率值为 8Hz。考虑 freq 的长度为 16 位，最大计数值为 65535，所以可以记录的 250 个脉冲之间的时间间隔最大为 65535*250*2us = 32767500us。那么最长 1 个脉冲周期为 32767500us/250 = 131070us，换算成频率为 1/131070 = 7.63Hz。
- ✓ 测量频度：与被测频率有关。如被测频率为 125Hz，测量频度 = 1 次/2 秒；被测频率为 250Hz，测量频度 = 1 次/秒；被测频率为 2K，测量频度 = 8 次/秒。
- ✓ 使用资源：两个定时器，两个中断，其中一个支持中断嵌套。

下面我们进一步讨论测量的精度问题，在测频法中，由于频率测量的绝对误差是±10Hz，因此被测频率越高（仅受系统时钟限制），测量精度也就越好，这一点是明显的。而在测周法中，由于其周期测量绝对误差是固定的，因此被测频率越低，精度越好。这一特点不容易直接看出，我们以测量 1K 频率和 4K 频率为例，分别计算出它们的精度结果，并进行比较。

首先我们取测周法的周期测量绝对误差为±(2us/250)，即±0.008us。对于 1K 频率，其标准周期为 1000us。考虑测量误差：1000.008us～999.992us，对应频率为：999.992Hz～1000.008Hz，有效位数为 6 位。而对于 4K 频率，其标准周期为 250us。考虑测量误差：250.008us～249.992us，对应频率为：3999.872Hz～4000.128Hz，此时有效位数降为 5 位了。可见，当被测频率越高时，有效位数越少，测量的精度也越差了。

11.1.4 频率测量小结

以上我们介绍了两种频率的测量方法，通过分析我们知道，频率的测量还是比较复杂的。

如果设计制作一个频率计，要能满足在被测频率范围比较宽，变化大时使用的话，单一的使用某一种测量方法都是不能达到需要的。所以，一个完善的频率计，要设计一个智能的测量过程，即其系统程序能够根据每次的测试数据，自动转换使用正确的测量方法，以及能够自动调节限定的时间（测频法），或调节限定脉冲数（测周法），或调整计时的时间基时等。这样经过几次自动的调整后，系统测出的频率达到最高的测量精度。

此外，上面的频率测量方法都必须占用 MCU 的 2 个硬件资源，这也是一般单片机测频所采用的方法（或采用 1 个 T/C 加 1 个外部中断，同样占用 2 个硬件资源）。AVR 单片机的 T/C1 增加了捕捉功能，利用该功能进行频率的测量时，不但只需要使用 1 个硬件资源 T/C1 就能完成周期的测量，而且还能获得更好的测量的精度。

11.2 基于T/C1 捕捉功能实现高精度的周期测量

在第8章第4节中我们介绍了AVR定时计数器的一个非常有特点的功能－－T/C1的输入捕捉功能。该功能可以应用于精确捕捉一个外部事件的发生，记录事件发生的时间印记（Time-stamp）。当一个输入捕捉事件发生，如外部引脚ICP1上的逻辑电平变化时，T/C1计数器TCNT1中的计数值被实时的写入到输入捕捉寄存器ICR1中，并置位输入捕获标志位ICF1，产生中断申请。因此，利用输入捕捉功能可以实现对周期的精确测量。

采用输入捕捉功能进行精确周期测量的基本原理比较简单，实际上就是将被测信号作为 ICP1 的输入，被测信号的上升（下降）沿作为输入捕捉的触发信号。T/C1 工作在常规计数器方式，对设定的已知系统时钟脉冲进行计数。在计数器正常工作过程中，一旦 ICP1 上的输入信号由低变高（假定上升沿触发输入捕捉事件）时，TCNT1 的计数值被同步复制到了寄存器 ICR1 中。换句话将，当每一次 ICP1 输入信号由低变高时，TCNT1 的计数值都会再次同步复制到 ICR1 中。

如果能及时的将两次连续的 ICR1 中数据记录下来，那么 2 次 ICR1 的差值乘上已知的计数器计数脉冲的周期就是输入信号一个周期的时间。由于在整个过程中，计数器的计数工作没有受到任何影响，捕捉事件发生的时间印记也是由硬件自动同步复制到 ICR1 中的，因此所得到的周期值是非常精确的。

下面，我们把“AVR-51 多功能实验开发板”K 区提供有占空比为 50%、125Hz～128KHz 之间 10 种不同频率的标准方波脉冲信号作为被测信号源，给出仅采用一个 T/C1，配合输入捕捉功能的应用，实现一个高精度的周期（频率）测试计的设计应用。

例 11.3　基于 T/C1 捕捉功能的可变量程频率计的设计与实现

1）　硬件电路

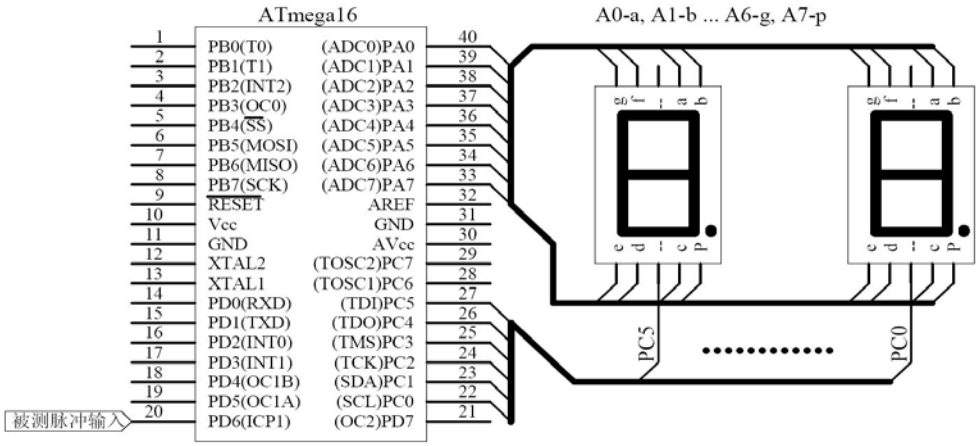

图 11-1 基于 T/C1 捕捉功能的可变量程频率计电路图

本例的硬件电路如图 11-1 所示，PA 口为 6 个 LED 数码管的段输出，PC 口是 6 个 LED 数码管的位扫描控制口。6 位 LED 构成频率计的结果显示。被测脉冲信号由 ICP1（PD6）输

入。需要注意的是，为了提高测量的精度，系统时钟应该采用外部晶体，同时系统时钟频率原则上越高越好。本例中采用外部 4M 晶体，因此系统时钟频率为 4M，周期为 0.25us（图中未画出外部晶体部分的电路和 8 个段限流电阻）。

2） 软件设计

尽管采用输入捕捉功能进行精确周期测量的基本原理比较简单，但是实际实现起来却不是那么简单的。因为系统中需要 LED 扫描显示，频率值的换算也需要大量的计算，而且在系统的运行的过程中，还必须确保 T/C1 每次捕捉中断产生后马上把寄存器 ICR1 中的时间印记读出，以及 T/C1 计数过程是否溢出等等。另外由于被测信号的频率范围在 125Hz 到 128KHz 之间，差比达 128000/125 = 1024 倍，所以还要考虑使用量程的自动转换。

下面首先给出系统程序，然后做必要的说明。

```
/*********************************************
File name           : demo_11_3.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/
#include <mega16.h>
sfrw ICR1=0x26;             // 补充定义 16 位寄存器 ICR1 地址为 0x26(mega16.h 中未定义)

flash char led_7[11]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00};
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

char dis_buff[6];           // 显示缓冲区，存放要显示的 6 个字符的段码值
unsigned int icp_v1,icp_v2;
char icp_n,max_icp;
bit icp_ok,time_4ms_ok,f_2_d,begin_m,full_ok;

void display(void)          // 6 位 LED 数管动态扫描函数
{
    static char posit;
    PORTC = 0xff;
    PORTA = led_7[dis_buff[posit]];
    PORTC = position[posit];
    if (++posit == 6) posit = 0;
}

// Timer 2 比较匹配中断服务，4ms 定时
interrupt [TIM2_COMP] void timer2_comp_isr(void)
{
    #asm("sei")             // 开放全局中断,允许中断嵌套
    display();
```

```
     time_4ms_ok = 1;
}

// Timer 1 溢出中断服务
interrupt [TIM1_OVF] void timer1_ovf_isr(void)
{
    full_ok = 1;
}

// Timer 1 输入捕捉中断服务
interrupt [TIM1_CAPT] void timer1_capt_isr(void)
{
    if (icp_n >= max_icp)   // 第 N 个上升沿到
    {
        icp_v2 = ICR1;      // 记录第 N 个上升沿时间
        TIMSK = 0x80;       // 禁止 T/CI 输入捕捉和溢出中断
        icp_ok = 1;
    }
    else if (icp_n == 0)
    {
        icp_v1 = ICR1;      // 记录第 1 个上升沿时间
    }
    icp_n++;
}

void f_to_disbuf(long v)    // 频率值送显示缓冲区函数
{
     char i;
     for (i=0;i<=4;i++)     // 转换成 6 位 BCD 码送显示缓冲区
     {
        dis_buff[i] = v%10;
        v /= 10;
     }
     dis_buff[5] = v;
     for (i=5;i>0;i--)       // 高位零不显示
     {
          if (dis_buff[i] == 0)
               dis_buff[i] = 10;
          else
               break;
     }
}
void main(void)
{
```

```
unsigned int icp_1,icp_2;
long fv;

DDRA=0xFF;          // LED 段码输出
PORTC=0xFF;
DDRC=0x3F;          // LED 位控输出
PORTD=0x40;         // PD6(icp)输入方式，上拉有效

// T/C2 初始化
TCCR2=0x0C;         // 内部时钟，64 分频（4M/64=62.5KHz），CTC 模式
OCR2=0xf9;//7c;          // OCR2 = 0xf9(249),(249+1)/62.5=4ms
// T/C1 初始化
TCCR1B = 0x41;      // T/C1 正常计数方式，上升沿触发输入捕捉，4M/1 计数时钟
TIMSK = 0xA4;      // 允许 T/C2 比较匹配中断，允许 T/C1 输入捕捉、溢出中断

icp_n = 0;
max_icp =1;
#asm("sei")             // 开放全局中断

while (1)
{
    if (icp_ok == 1)    // 完成一次测量
    {
        if (icp_v2 >= icp_v1)          // 计算 N 个上升沿的时钟脉冲个数，
            icp_2 = icp_v2 - icp_v1;
        else
            icp_2 = 65536 - icp_v1 + icp_v2;

        if (!(icp_v2 >= icp_v1 && full_ok)) // 有溢出，数据无效
        {
            if (icp_2 == icp_1)         // 两次个数相等，测量有效
            {
                fv = 4000000 * (long)max_icp / icp_2;     // 换算成频率值
                f_2_d = 1;              // 允许新频率送显示

                if (fv > 4000)
                    max_icp = 64;       // 如果频率大于 4Khz，N=64
                else
                    max_icp = 1;        // N=1
            }
        }
        Else
            max_icp = 1;    // 有溢出，N=1
        icp_1 = icp_2;
```

```
                icp_ok = 0;
                begin_m = 1;
            }
            if (time_4ms_ok)
            {
                if (f_2_d)
                {
                    f_to_disbuf(fv);   // 新频率送显示
                    f_2_d = 0;
                }
                else if (begin_m)
                {
                    icp_n = 0;      // 开始新的一次测量，
                    full_ok = 0;    // 清除溢出标志
                    TIFR = 0x24;    // 清除可能存在的输入捕捉、溢出中断标志位
                    TIMSK = 0xa4; // 开启 T/C1 输入捕捉、溢出中断允许
                    begin_m = 0;
                }
                time_4ms_ok = 0;
            }
        }
    }
```

程序中 LED 动态扫描函数 desplay()，以及频率值转换成 BCD 码后送显示缓冲区函数 f_to_disbuf()已经在前面多次出现了，请读者自己分析。

在该程序中，使用了两个定时计数器。T/C2 工作在 CTC 方式，每隔 4ms 定时中断一次，定时服务中执行 LED 显示扫描。扫描定时时间的设计考虑了 2 个方面，设置 4ms 为 LED 显示扫描间隔，即能达到每秒 40 次的扫描频率，也尽量减少了 T/C2 中断对 T/C1 中断及时响应的影响。同时为了确保不影响 T/C1 的工作，在 T/C2 的中断服务中还必须再次开放全局中断，实现中断嵌套，使得 T/C1 的中断能及时的得到响应。

本例中仅使用了一个 16 位的 T/C1 进行周期的测量。T/C1 工作在计数器方式，对 4M 系统时钟进行计数，因此每 1 个数的时间为 0.25us。T/C1 设置为引脚 ICP1 的上升沿为外部事件的触发。一旦 ICP1 上出现上跳变，T/C1 的硬件将自动同步的把当前 TCNT1 的值复制到 ICR1 中，并申请捕捉中断。在 T/C1 捕捉中断服务程序中记录下两个 ICR1 的值：一个为第 0 次触发时的 T/C1 值，另一次为第 1(N)次触发时 T/C1 值。当第 2 个值也记录下来后，随即关闭 T/C1 所有的中断，将 2 个记录的数据交给主程序进行有效性的判断和周期频率的换算。程序中还使用了 T/C1 的溢出中断，该中断主要用于判断第二次 ICR1 的值是否比第一次 ICR1 的值超出了 65536 个，如果超出，则需要调整量程。T/C1 的两个中断服务都是非常重要的（等级相同），任何一个一旦发生，都应该立即响应，不能延误。在实际情况中这点是不容易做到的，但应尽量精心设计，尽量做到没有延误，或减少延误。另外，这两个中断服务程序的执行时间也必须越短越好。

在系统主程序中，对每一组的两个 ICR1 值进行判断，将其相减，得到差值，然后判断其是否溢出。这里的溢出不是单指 TCNT1 的值从 65535 变到 0 的溢出，其条件应是当第 2 个 ICR1 值大于第 1 个 ICR1 的值，且 TCNT1 的值出现过从 65536 变到 0 的现象。一旦出现了这种情况，说明两次 ICR1 的差值超出了 16 位 65536 的长度，数据无效，需要改变量程了。

另外，在程序中还采取了连续 2 次有效的差值相等才作为一次真正有效的周期测量的限定，更有效的把受到各种干扰以及由于中断响应不及时造成的错误数据剔除掉了。

另外在系统主程序中，把频率值送显示缓冲区的调用放置在刚刚扫描过一位 LED 数码管后执行，这是由于频率值送入显示缓冲区的工作需要比较长的执行时间，更重要的是要改变显示缓冲区的数值。而在这个期间，f_to_disbuf()函数一旦被 LED 扫描中断打断的话，就会造成个别数字显示的不稳定以及跳动的现象。考虑到 LED 扫描的间隔时间有 4ms，所以把频率值送入显示缓冲区的工作放置在刚刚扫描过一位 LED 数码管后立即执行，就能充分利用 4ms 的间隔，可以使整个函数的执行过程不会被中断打断了（4M 系统时钟条件下，4ms 可以执行约 16000 条指令!）。同时，在主程序的处理中，只有在一次周期测量过程的数据全部处理完成，并将新的转换频率值送显后，才重新开启 T/C1 的中断，开始新的一次周期测量。这就使周期测量和数据处理是完全分开（分时）进行的，两者之间没有相互的干扰，不会形成这边数据还没处理完，那边又来了新的测量数据所造成的数据冲突的现象。

在本例中，周期的测量采用了比较简单的量程自动转换方式。量程的确定受到 T/C1 的长度和计数器的计数脉冲频率和被测频率的制约。对于 125Hz 到 4K 的频率测量，采用的是记录间隔为 1（N = 1）的两个相邻上升沿之间的时间差，也就测量被测信号一个周期的时间。对于 4M 的计数时钟，1/125 的时间内可以记录的脉冲个数为 32000 个，而 1/4000 的时间内可以记录的脉冲个数为 1000 个，均不超出 65536，T/C1 的长度。对于 4KHz 到 128K 的频率测量，采用的是记录间隔为连续 64（N = 64）个上升沿之间的时间差，也就测量被测信号 64 个周期的时间。对于 4M 的计数时钟，64/4000 的时间内可以记录的脉冲个数为 64000 个，而 64/128000 的时间内可以记录的脉冲个数为 2000 个，都不超出 65536，T/C1 的长度。因此，当测量信号的频率值大于 4K 时，自动转换成 N=64 的量程，而一旦频率小于 4K，或出现测量数据溢出情况时，量程自动转换成 N=1。

下面对该实例程序的周期测量性能和指标进行评估（假定系统时钟没有误差 = 4MHz）：

✓ 周期测量绝对误差：(±0.25us)/(±0.25us/64)。需要注意的是，为了能简洁的说明主要的设计思想，本例程做了简化，采用整形数计算处理，所以频率值仅显示到个位的 Hz，小数点后的数值已经丢却了，真正的精度没有体现出来。下面我们以测量 125Hz 频率和 128K 频率为例，分别计算出它们的精度结果，并进行比较。

对于 125Hz 频率，其标准周期为 8000us，测量绝对误差为±0.25us，那么测量误差：8000.25us～7999.75us，对应频率为：124.9961Hz～125.0039Hz，有效位数为 5 位。而对于 128K 频率，其标准周期为 7.8125us，测量绝对误差为±0.25us/64，那么测量误差为：7.81640625us～7.80859375us，对应频率为：127936.0Hz～128064.0Hz，有效位数为 4 位。可见，当被测频率越高时，有效位数越少，测量的精度也越差了。

读者也许会有疑问，这个例子好像没有上面的例子精度高。其实，在例程 11-2 中的评估条件是在不考虑中断影响下进行的，而在本例中，中断处理是不影响精度的。另外在上面的例子中是测量 250 个脉冲的周期，而在本例中只是 1 个和 64 个。

✓ 被测最低频率值：62.5KHz。由于 T/C1 的长度 16 位，1/62.5 时间内可以记录的个数为 64000 个。当频率值再低的话，一个周期内的计数值将超出 65536，造成溢出。

✓ 被测最高频率值：128KHz。可能有的读者认为只要增加 N 的值，就能提高测量频率的上限，这只是在一定条件下才可以这样考虑的。实际上被测频率的上限是由 T/C1 捕捉中断服务程序的执行时间限定的，因为被测信号每一个脉冲的上升沿时都要进入中断处理程序的，而中断处理的时间必须在下一个上升沿到来前完成，否则将会丢失掉一次中断，造成数据不准确。在 4M 时钟系统下，128K 的被测信号每隔 7.8125us 就产生一次中断，而在这个时间内，MCU 最多可以执行 31.25 条指令！考虑到 T/C1 的中断服务还有嵌在 T/C2 的 LED 定时扫描中断中执行的情况，所以每次 T/C1 的中断服务程序执行的指令应该小于 25 条指令！

从这点可以看出，在本例中，最关键的一环是中断服务程序的设计和编写。在真正产品的设计中，这样的中断服务程序建议最好采用汇编编写，当然这就要求程序员具备更高的水平了。

- ✓ 测量频度：40 次/秒。T/C1 计满一次需要 65535 个系统时钟，约 65535＊0.25us = 16.34ms，附加上 2 次 LED 定时时间间隔 8ms 的计算处理时间，一次测量完成时间约为 25ms，所以测量频度为 40 次/秒。
- ✓ 使用资源：一个 16 位定时器，两个中断（T/C2 中断对测周期没有贡献）。

3） 思考与实践

- ✓ 根据上面采用测频法的思路，在 MCU 的设置选择和软件方面如何能提高被测频率的上限值？
- ✓ 参考上面测频法的思路，在软件方面如何能降低被测频率的下限值？（参考提示：T/C1 溢出中断中记录溢出的次数）
- ✓ 参考上面测频法的思路，在软件方面如何能提高测量精度？（参考提示：增加 N 的值，T/C1 溢出中断中记录溢出的次数）
- ✓ 如果程序在一次测量中，得到的两次 ICR1 的有效差值为 234，那么此时的测量的有效位数是多少？相对精度为多少？为什么？

11.3 带校时和音乐报时功能实时时钟的设计与实现

在前面的章节中分别介绍了 I/O 口输入/出的应用、中断的应用、T/C 的应用，以及基于状态机思想的系统分析和系统程序设计方法等。在本节里，将给出一个功能比较完整的“带校时和音乐报时功能的实时时钟”系统，作为上面各种基本应用的综合设计示例。

例 11.4 带校时和音乐报时功能实时时钟的设计与实现

1） 硬件电路

硬件电路如图 11-2 所示，PA 口为 LED 数码管的 8 段码输出，PC0-PC5 共 6 个 I/O 口，作为控制时间显示的 6 个 LED 数码管的位扫描线。PC6、PC7 分别接连接两个按键，用于设置时钟的工作状态和校时时间的设置。图中音乐报时电路部分（未画出）与第 8 章中的图 8-20 相同，由端口 PD5 输出产生音乐的脉冲信号，经三极管驱动蜂鸣器发声。

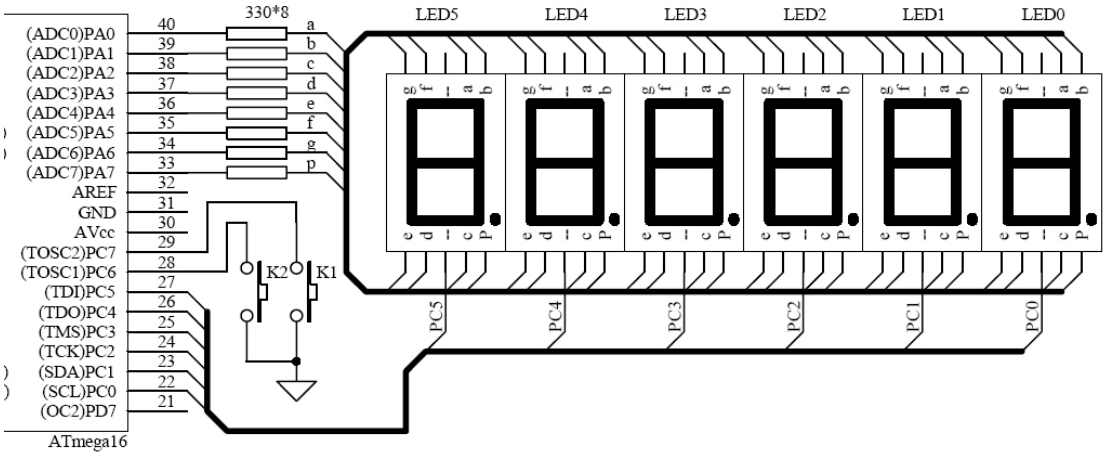

图 11-2 带校时功能的实时时钟电路图

定义两个按键的功能为：K1 用于设置转换时钟工作状态，K2 用于设置校时时间（加 1 操作）。时钟工作状态转换图如图 11-3 所示，具体每个状态的定义和功能如下：

- ✓ 平时时钟工作在时钟显示状态，每按一下 K1 键，时钟依次进入校时时间的设置状态。
- ✓ 时钟由“时钟显示”进入“秒低位设置”时，校时时间的初始值为转换时刻的时钟

值。

- ✓ 时钟由“时高位设置”回到（K1 作用下）“时钟显示”时，时钟时间由校时时间代替，确认完成校时的设置。
- ✓ 当时钟处在时间设置的 6 个状态时，每按一次 K2 键，相应的位上的数值加 1，并且要能根据具体所在的位置自动做相应的调整。如秒高位的数字只能在 0-5 之间，而时高位的数值要限制在 0、1、2（时个位数小于 3 时），或时高位的数值要限制在 0、1（时个位数大于 3 时）。
- ✓ 当时钟处在时间设置的 6 个状态时，在 20 秒内无任何键按下，系统自动返回“时间显示”状态，设置的时间无效，不改变原时钟的计时时间。
- ✓ 在效时时间设置的操作过程中，时钟不停止其前时间的计时过程，除非当时钟由“时高位设置”回到（K1 作用下）“时钟显示”时，时钟的计时时间由确认的校时时间代替而改变。
- ✓ 时钟显示亮度均匀、无闪烁。当设置相应时间位时，该位应闪烁提示。

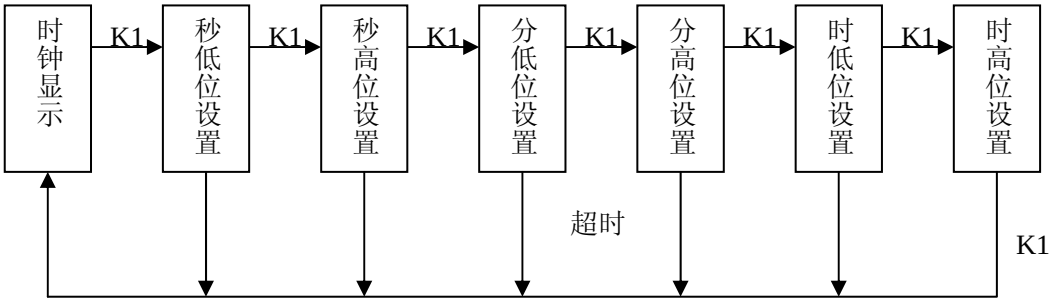

图 11-3 时钟工作状态转换图

2） 软件设计

本示例的程序是在前几章所给例子的综合应用基础上实现的，代码中也给出了相应的解释，因此本节中不再做更多的说明，留给读者去自行分析。希望能在真正掌握了前几章内容的基础上，慢慢的去品味和体会，掌握如何更好的综合使用 AVR 硬件的功能，以及程序设计的方法与技巧。

```
/*********************************************
File name           : demo_11_4.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 1.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/

#include <mega16.h>

flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

flash unsigned int t[9] = {0,956,865,759,716,638,568,506,470};
```

```
flash unsigned char d[9] = {0,105,116,132,140,157,176,198,209};
#define Max_note    32
flash unsigned char music[Max_note] =
        {5,2,8,2,5,2,4,2,3,2,2,2,1,4,1,2,1,2,2,2,3,2,3,2,1,2,3,2,4,2,5,8};

unsigned char note_n;
unsigned int int_n;
bit play_on;

char time[3],time_set[3];            // 时、分、秒计数和设置单元
char dis_buff[6];                    // 显示缓冲区，存放要显示的 6 个字符的段码值
char time_counter,key_stime_counter; // 时间计数单元，
char clock_state = 6,return_time;
bit  point_on,set_on,time_1s_ok,key_stime_ok;

void display(void)                   // 6 位 LED 数管动态扫描函数
{
    static char posit=0;
     PORTC = 0xff;
     PORTA = led_7[dis_buff[posit]];
     if (set_on && (posit==clock_state)) PORTA= 0x00;        // 校时闪烁
     if (point_on && (posit==2||posit==4)) PORTA |= 0x80;    // 秒闪烁
     PORTC = position[posit];
     if (++posit >=6 ) posit = 0;
}

// Timer 0 比较匹配中断服务，2ms 定时
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
     display();                          // LED 扫描显示
     if (++key_stime_counter >=5)
     {
          key_stime_counter = 0;
          key_stime_ok = 1;                              // 10ms 到
          if (!(++time_counter % 25)) set_on = !set_on;  // 设置校时闪烁标志
          if (time_counter >= 100)
          {
               time_counter = 0;
               time_1s_ok = 1;                           // 1s 到
          }
     }
}

// T/C1 比较匹配 A 中断服务
```

```
interrupt [TIM1_COMPA] void timer1_compa_isr(void)
{
    if (!play_on)
    {
        note_n = 0;
        int_n = 1;
        play_on = 1;
    }
    else
    {
        if (--int_n == 0)
        {
            TCCR1B = 0x08;
            if (note_n < Max_note)
            {
                OCR1A = t[music[note_n]];
                int_n = d[music[note_n]];
                note_n++;
                int_n = int_n * music[note_n];
                note_n++;
                TCCR1B = 0x09;
            }
            else
                play_on = 0;
        }
    }
}

void time_to_disbuffer(char *time)                  // 时钟时间送显示缓冲区函数
{
     char i,j=0;
     for (i=0;i<=2;i++)
     {
          dis_buff[j++] = time[i] % 10;
          dis_buff[j++] = time[i] / 10;
     }
}

#define key_input  PINC          // 按键输入口
#define key_mask   0b11000000    // 按键输入屏蔽码
#define key_no     0
#define key_k1     1
#define key_k2     2
#define key_state_0     0
```

```
#define key_state_1     1
#define key_state_2     2

char read_key(void)
{
     static char key_state = 0,key_press;
     char key_return = key_no;

     key_press = key_input & key_mask;     // 读按键 I/O 电平
     switch (key_state)
     {
          case key_state_0:                // 按键初始态
               if (key_press != key_mask) key_state = key_state_1;
               break;                      // 键被按下，状态转换到键确认态
          case key_state_1:                // 按键确认态
               if (key_press == (key_input & key_mask))
               {
                    if (key_press == 0b01000000) key_return = key_k1;
                    else if (key_press == 0b10000000) key_return = key_k2;
                    key_state = key_state_2;     // 状态转换到键释放态
               }
               else
                    key_state = key_state_0;     // 按键已抬起，转换到按键初始态
               break;
          case key_state_2:
               if (key_press == key_mask) key_state = key_state_0;
               break;                           //按键已释放，转换到按键初始态
     }
     return key_return;
}

void main(void)
{
     char key_temp,i;

     DDRA=0xFF;          // LED 段码输出
     PORTC=0xFF;
     DDRC=0x3F;          // LED 位控输出
     DDRD=0x20;          // PD5 音乐播放输出

     // T/C0 初始化
     OCR0 = 0xF9;        // OCR0 = 0xF9(249),(249+1)/125=2ms
     TCCR0 = 0x0A;       // 内部时钟，8 分频（1M/8=125KHz），CTC 模式
     // T/C1 初始化
```

```
TCCR1A=0x40;
TCCR1B=0x08;
TIMSK = 0x12;      // 允许 T/C1 比较匹配 A 中断，允许 T/C0 比较匹配中断

time[2] = 23; time[1] = 58; time[0] = 55; // 设时间初值 23:58:55

#asm("sei")             // 开放全局中断

while (1)
{
    if (time_1s_ok)                 // 1 秒到
    {
        time_1s_ok = 0;
        point_on = ~point_on;       // 秒闪烁标志
        if (++time[0] >= 60)        // 秒加 1，以下为时间调整
        {
            time[0] = 0;
            if (!play_on) TCCR1B = 0x09;    // 1 分钟到，播放音乐
            if (++time[1] >= 60)
            {
                time[1] = 0;
                if (++time[2] >= 24) time[2] = 0;
            }
        }
        if ((++return_time >= 20) && (clock_state != 6)) clock_state = 6;
        if (clock_state == 6) time_to_disbuffer(time);
    }
    if (key_stime_ok)               // 10ms 到，键处理
    {
        key_stime_ok = 0;
        key_temp = read_key();      // 调用按键接口程序
        if (key_temp)               // 确认有键按下
        {
            return_time = 0;
            if (key_temp == key_k1)     // K1 键按下，状态转换
            {
                if (++clock_state >= 7) clock_state = 0;
                if (clock_state == 0)
                {
                    for (i=0;i<=2;i++) time_set[i] = 0;
                    time_to_disbuffer(time_set);
                }
                if (clock_state == 6)
                {
```

```
                    for (i=0;i<=2;i++) time[i] = time_set[i];
                    time_to_disbuffer(time);
                }
            }
            if ((clock_state != 6) && (key_temp == key_k2))     // K2 键按下
            {
                if (clock_state%2) time_set[clock_state/2] += 10;
                else
                {
                    if ((time_set[clock_state/2] % 10) == 9)
                        time_set[clock_state/2] -= 9;
                    else
                        time_set[clock_state/2]+=1;
                }
                if (time_set[0] >= 60) time_set[0]-= 60;  // 以下设置时间调整
                if (time_set[1] >= 60) time_set[1]-= 60;
                if (time_set[2] >= 24) time_set[2]-= 10;
                time_to_disbuffer(time_set);              // 设置时间送显示缓存
            }
        }
    }
}
```

思考与练习

到本章为止，我们已经结束了对 AVR 最基本功能单元的原理、工作方式和应用设计方法的学习。但要真正的达到我们的学习目标还需要不断的实践和练习。实际上基于前十章的学习，读者已经可以自行进行一些简单的应用系统的设计，只有这样，才能得到真正的提高。以下给出几个简单应用系统的设计题目，读者可以根据实际系统功能，定义系统需要实现的功能，并进行设计和实现。

1. 洗衣机或微波炉的控制器。系统具备的基本功能有：时间设置、分段倒计时、电机控制（输出到 LED 显示模拟）、时间到音乐提示等。
2. 出租车计价器。用不同频率的输入脉冲代表不同的转速值，有行驶里程显示、单价显示、总价计算和显示等。
3. 秒表系统设计。设计一个秒表系统，可以同时记录 3 个运动员跑 100M 的时间。
4. 大脑反应测试器。有 8 个 LED 二极管，每次随机点亮其中的一个，时间 0.5 秒。要求被测人员看到 LED 亮后，马上按相对应的按键。系统能记录每次的反应时间，N 次平均为被测人员的反应系数。
5. 简易数字万用表。能测量电阻、电流和电压。
6. 简易电子计算器。
7. 简易信号发生器。能产生不同频率的方波、正弦波、三角波、锯齿波。
8. 简易电子琴。