

普通高等院校电子信息类系列教材

C51 单片机技术教程

田希晖 薛亮儒 编著

人 民 邮 电 出 版 社

北 京

图书在版编目 (CIP) 数据

C51 单片机技术教程 / 田希晖, 薛亮儒编著. —北京: 人民邮电出版社, 2007.10
(普通高等院校电子信息类系列教材)

ISBN 978-7-115-16444-5

I. C... II. ①田...②薛... III. 单片微型计算机, C51—高等学校—教材 IV. TP368.1
中国版本图书馆 CIP 数据核字 (2007) 第 091878 号

内 容 提 要

本书以单片机的基本组成构造为主, 以外围器件、相关电路的扩展为辅, 向读者介绍了单片机软硬件的初步知识和基本的系统扩展。

本书分 9 章, 循序渐进地对 C51 单片机展开讲解, 内容涉及单片机的硬件、软件及系统扩展等方面, 并介绍了一些当前比较新的单片机扩展及应用。本书使用的典型实例都是精挑细选后确定的, 基本覆盖了单片机的主要应用技术。通过学习本书的实例, 读者除可以掌握单片机的基本知识外, 还可获得如何针对一个具体的项目需求设计初步的解决方案以及如何运用单片机的关键技术满足项目需求。

本书基础性和实用性较强, 对于单片机初学者和进行实际项目开发都具有非常高的参考价值。本书适合普通高等院校电子信息科学与技术、电子信息工程、通信工程、自动控制、电子技术、计算机科学与技术及网络工程等专业单片机等相关课程教学使用, 同时也适合单片机初学者、开发人员和系统设计人员阅读和参考。

普通高等院校电子信息类系列教材 C51 单片机技术教程

-
- ◆ 编 著 田希晖 薛亮儒
责任编辑 蒋 亮
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京隆昌伟业印刷有限公司印刷
新华书店总店北京发行所经销
 - ◆ 开本: 787×1092 1/16
印张: 19.5
字数: 473 千字 2007 年 10 月第 1 版
印数: 1—3 000 册 2007 年 10 月北京第 1 次印刷

ISBN 978-7-115-16444-5/TP

定价: 28.00 元

读者服务热线: (010) 67170985 印装质量热线: (010) 67129223

第 1 章 C51 单片机的组成及结构

重点与难点

本章重点是掌握 C51 单片机引脚功能、单片机存储器结构和单片机工作方式。难点是数据存储器结构及专用存储器（SFR）的工作原理和方式。

基本要求

1. 熟悉 C51 单片机 40 个引脚（DIP 封装）及其功能
2. 掌握数据存储器结构，熟悉专用存储器（SFR）的工作原理和方式
3. 理解 C51 单片机工作时序及其基本概念，时钟电路的结构和单片机的工作过程
4. 对 C51 单片机复位、程序执行、单步执行、掉电保护、低功耗以及 EPROM 编程和校验 6 种工作方式有所了解，为以后学习指令系统和编写程序打下基础

本章内容

1. C51 单片机的内部结构
2. C51 单片机的 CPU 结构
3. C51 单片机的存储器空间
4. C51 单片机的并行输入/输出端口
5. C51 单片机的引脚信号和 CPU 时序
6. C51 单片机的工作方式

1.1 C51 单片机的内部结构

所谓单片机，即把组成微型计算机的各个功能部件，如中央处理器（CPU）、随机存储器（RAM）、只读存储器（ROM）、输入/输出接口电路、定时器/计数器以及串行通信接口等集成在一块芯片中，构成一个完整的微型计算机。因此，单片机早期的含义为单片微型计算机（single chip microcomputer），直接译为单片机，并一直沿用至今。

C51 单片机和微型计算机（以下称作微机）一样，是由运算器、控制器、存储器、输入设备、输出设备 5 个基本部分组成，如图 1-1 所示。

C51 单片机的主要资源如下。

- （1）8 位 CPU。能进行 8 位二进制数或代码的运算。
- （2）片内振荡器。其振荡频率 $f_{osc}=1.2\text{MHz}\sim 12\text{MHz}$ ，C51 芯片的内部有时钟电路，但

石英晶体和微调电容需外接，时钟电路为单片机产生时钟脉冲序列，系统允许的最高晶振频率为 12MHz。

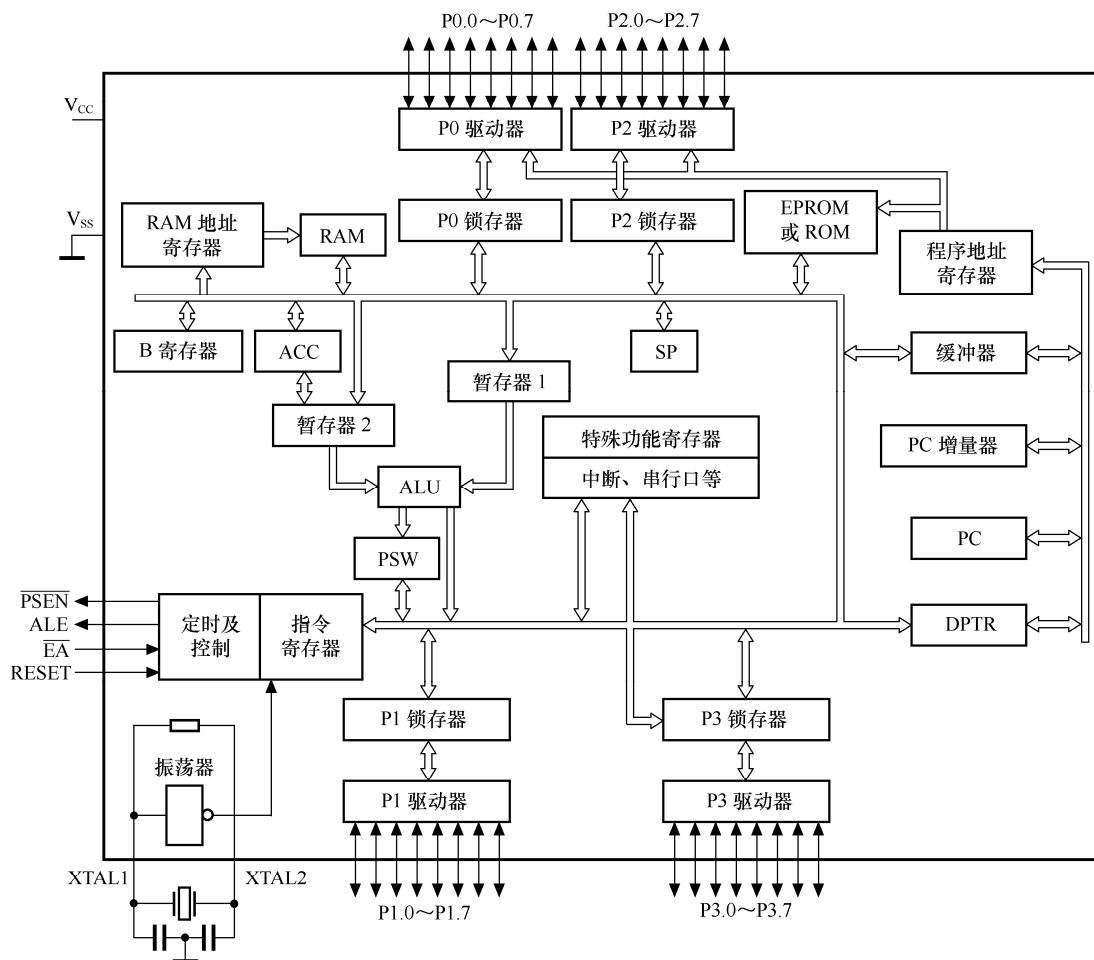


图 1-1 C51 单片机内部结构框图

(3) 片内数据存储器（RAM）。128byte 的片内数据存储器，低 128byte 用以存放可以读/写的数，高 128byte 被特殊功能寄存器占用，只有前 128byte 能作为寄存器供用户使用。

(4) 片内程序存储器（ROM）。C51 有 4KB 的片内程序存储器，用以存放程序、一些原始表格，简称为内部 ROM。但也有些单片机内部不带 ROM/EPROM，如 8031、8032、80C31 等；一些单片机内部 ROM 容量大一些，如 C52 有 8KB 的内容 ROM。

(5) 程序存储器的寻址范围为 64KB。

(6) 片外数据存储器（RAM）寻址范围为 64KB。

(7) 21 个特殊功能寄存器（SFR）。

(8) 4 个 8 位并行双向 I/O 接口：P0，P1，P2，P3。

(9) 1 个全双工串行 I/O 接口（UART），可多机通信。

(10) 2 个 16 位定时器/计数器 T0/T1，C52 还多 1 个 16 位定时器/计数器 T2。

(11) 中断系统有 5 个中断源，即外中断源 2 个，内部中断源 3 个，内部中断中分 2 个定时/计数中断和 1 个串行中断，全部中断分高级和低级共 2 个优先级别。

(12) 111 条指令，含乘法、除法指令。

(13) 位操作功能强（位操作指令 17 条）。

(14) 片内采用单总线结构。

(15) 使用+5V 单一电源。

1.2 C51 单片机的 CPU 结构

CPU 是单片机的核心部件，它由运算器和控制器等部件组成。运算器还包含一个专门进行位数据操作的布尔处理机。

1.2.1 运算器

运算部件以算术逻辑单元（ALU）为核心，包括累加器（ACC）、寄存器 B、暂存器、程序状态字（PSW）等部件。它能实现数据的算术逻辑运算、位变量处理和数据传输操作。

1. 算术逻辑单元（ALU）与累加器（ACC）、寄存器 B

算术逻辑单元不仅能完成 8 位二进制的加、减、乘、除、加 1、减 1 及 BCD 加法的十进制调整等算术运算，还能对 8 位变量进行逻辑“与”、“或”、“异或”、循环移位、求补、清零等逻辑运算，并具有数据传输、程序转移等功能。累加器（ACC，简称累加器 A）为一个 8 位寄存器，它是 CPU 中使用最频繁的寄存器。进入 ALU 做算术和逻辑运算的操作数多来自于累加器 A，运算结果也常送回累加器 A 保存。

寄存器 B 是为 ALU 进行乘除法运算而设置的。若不做乘除运算时，则可作为通用寄存器使用。

2. 程序状态字

程序状态字（PSW）是一个 8 位的标志寄存器，它保存指令执行结果的特征信息，以供程序查询和判别。其各位的功能、标志符号和位地址如表 1-1 所示。

表 1-1 PSW 寄存器各位功能、标志符号、位地址

标志位功能	标志符号	位地址	标志位功能	标志符号	位地址
进位标志位	C 或 CY	PSW.7	工作寄存器选择位	RS0	PSW.3
辅助进位标志位	AC	PSW.6	溢出标志位	OV	PSW.2
用户标志位	F0	PSW.5	保留位	...	PSW.1
工作寄存器选择位	RS1	PSW.4	奇偶标志位	P	PSW.0

进位标志位 C 或 CY（PSW.7）：在执行某些算术操作类、逻辑操作类指令时，可被硬件或软件置位或清零；它表示运算结果是否有进位或借位，如果在最高位有进位（加法时）或

有借位（减法时），则 $C=1$ ，否则 $C=0$ 。

辅助进位（或称半进位）标志位 AC ($PSW.6$)：它表示两个 8 位数运算，低 4 位有无进位（借）位的状况；当低 4 位相加（或相减）时，若 $D3$ 位向 $D4$ 位有进位（或借位），则 $AC=1$ ，否则 $AC=0$ ；在 BCD 码运算的十进制调整中要用到该标志。

用户自定义标志位 $F0$ ($PSW.5$)：用户可根据自己的需要对 $F0$ 赋予一定的含义，通过软件置位或清零，并根据 $F0=1$ 或 0 来决定程序的执行方式，或反映系统某一种工作状态。

工作寄存器组选择位 $RS1, RS0$ ($PSW.4, PSW.3$)：可用软件置位或清零，用于选定当前使用的 4 个工作寄存器组中的某一组。

溢出标志位 OV ($PSW.2$)：做加法或减法时，由硬件置位或清零，以指示运算结果是否溢出。 $OV=1$ 反映运算结果超出了累加器的数值范围（无符号数的范围为 $0\sim255$ ，以补码形式表示一个有符号数的范围为 $-128\sim+127$ ）。进行无符号数的加法或减法时， OV 的值与进位标志位 C 的值相同；进行有符号数的加法时，如最高位、次高位之一有进位，或做减法时，如最高位、次高位之一有借位， OV 被置位，即 OV 的值为最高位和次高位的异或 ($C7\oplus C6$)。执行乘法指令也会影响 OV 标志，积 >255 时 $OV=1$ ，否则 $OV=0$ 。执行除法指令也会影响 OV 标志，如 B 中所放除数为 0， $OV=1$ ，否则 $OV=0$ 。

奇偶标志位 P ($PSW.0$)：在执行指令后，单片机根据累加器 A 中 1 的个数的奇偶自动给该标志置位或清零。若累加器 A 中 1 的个数为奇数，则 $P=1$ ，否则 $P=0$ 。该标志对串行通信的数据传输非常有用，通过奇偶校验可检验传输的可靠性。

算术逻辑单元 (ALU) 主要完成下列功能。

(1) 算术运算：加、带进位加、减、带借位减、乘、除、加 1、减 1 及 BCD 加法的十进制调整。

(2) 逻辑运算：与、或、异或、求反、清 0。

(3) 移位功能：对累加器 ACC 或带进位位 C 进行逐位的循环左、右移位。

微处理器指令系统的优劣决定了运算器功能的强弱，C51 系列单片机较之 48 系列单片机功能更为强大，其原因在于它的运算器功能更全面、更丰富。

1.2.2 布尔处理机

布尔处理（即位处理）机是 C51 单片机 ALU 所具有的一种功能。单片机指令系统中的位处理指令集（17 条位操作指令），存储器中的位地址空间，以及借用程序状态寄存器 PSW 中的进位标志 CY 作为位操作“累加器”，构成了 C51 单片机内的布尔处理机。它可对直接寻址的位 (bit) 变量进行位处理，如置位、清零、取反、测试转移以及逻辑“与”、“或”等位操作，使用户在编程时可以利用指令完成原来单凭复杂的硬件逻辑所完成的功能，并可方便地设置标志等。

1.2.3 控制部件及振荡器

控制部件是单片机的神经中枢，它包括定时和控制电路、指令寄存器、译码器数据地址指针 (DPTR)，程序计数器 (PC)，堆栈指针 (SP)，以及 RAM 地址寄存器，16 位地址缓冲器等部件。

单片机以主振频率为基准发出 CPU 的时序，对指令进行译码，然后发出各种控制信号，完成一系列定时控制的微操作，用来协调单片机内部各功能部件之间的数据传送、数据运算等操作，并对外发出地址锁存 ALE、外部程序存储器选通 PSEN，以及通过 P3.6 和 P3.7 发出数据存储器读 (RD)、写 (WR) 等控制信号，并且接收处理外接的复位和外部程序存储器访问控制 EA 信号。

单片机的定时控制功能是用片内的时钟电路和定时电路来完成的，而片内的时钟产生有两种方式，即内部振荡方式和外部振荡方式。

内部振荡方式，即在引脚 XTAL1 和 XTAL2 外接晶体振荡器（简称晶振），如图 1-2 所示。由于单片机内部有一个高增益运算放大器，当外接晶振后，就构成了自激振荡器并产生振荡时钟脉冲。图 1-2 中，电容器 C_1 、 C_2 起稳定振荡频率、快速起振的作用。晶体振荡器一般可以在 1.2MHz~12MHz 之间任选，电容值一般为 5pF~30pF，对时钟频率有微调作用。

外部振荡方式，即把已有的时钟信号引入单片机。这种方式适用于使单片机的时钟与外部信号保持一致。外部振荡方式接法如图 1-3 所示。

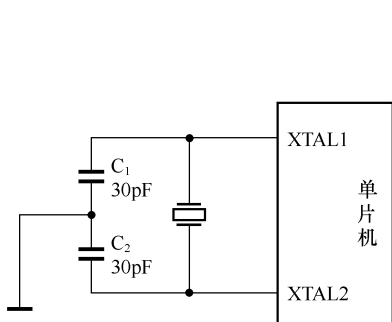


图 1-2 单片机内部振荡方式接法

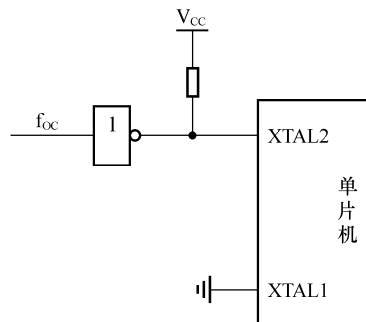


图 1-3 单片机外部振荡方式接法

至于 16 位的程序计数器 (PC) 和 8 位的堆栈指针 (SP) 等，其作用和一般微机是相同的。CPU 利用程序计数器 (PC) 指出下一条指令字节所在程序存储器地址，而用堆栈指针 (SP) 在片内 RAM 的 128 个字节中开辟栈区，并随时跟踪栈顶地址。C51 的堆栈是向上生成的（即先入后出），即将一字节数据压入堆栈时，堆栈指针 (SP) 增 1，将一字节数据弹出堆栈时，堆栈指针 (SP) 减 1，如图 1-4 所示。初始化时，单片机栈底地址（即 SP 的内容）为 07H。由于 C51 系列单片机存储器组织将程序存储器和数据存储器分开，所以又增设了一个 16 位的地址指针 (DPTR)，用以指示外部数据存储器或 I/O 口的地址。通过后面结合指令系统介绍，对它们的作用将会有更深的体会。

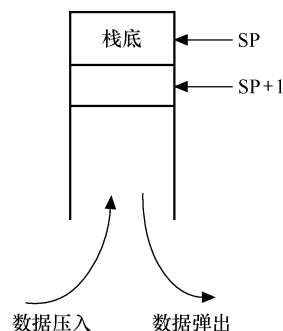


图 1-4 C51 单片机堆栈

1.3 C51 单片机的存储器空间

1.3.1 存储器的基本知识

存储器的常用单位和术语如下。

1. 位的含义

开关的闭合或电位器电平的高低，可以代表两种状态：0 和 1。因此，可以把一个开关或电位器称之为“位”，用 bit 来表示。位是计算机中所能表示的最基本和最小的数据单位。

2. 字节（byte）的含义

一个开关可以表示“0”或“1”，两个开关可以表示“00”、“01”、“10”、“11”四种状态，也就是可以表示 0~3，计算机中通常用 8 位同时计数，就可以表示 0~255。这 8 位就称之为一个字节（byte），即 1 byte=8bit。

在微机中，数据是以字节为单位进行存放的。通常把 1024 个字节称为 1KB（即 1KB=1 024byte），1MB=1 024×1 024byte=1 024KB，1GB=1 024MB，1TB=1 024GB。

3. 字（word）

通常由两个字节构成一个字。

4. 字长

字长是指计算机能一次处理二进制数码位数的多少。如 C51 单片机是 8 位机，字长为 8 位。以 80586 为微处理器的计算机是 32 位机，其字长为 32 位。

1.3.2 C51 单片机的存储器

计算机的存储器结构有两种：一种结构称为哈佛结构，即程序存储器和数据存储器分开，相互独立；另一种结构称为普林斯顿结构，即程序存储器和数据存储器是统一的，地址空间统一编址。

C51 单片机属于哈佛结构，程序存储器和数据存储器分开。C51 单片机的存储器在物理结构上分为程序存储空间和数据存储空间，共有 4 个存储空间：片内程序存储空间和片内数据存储空间以及片外程序存储空间和片外数据存储空间，如图 1-5 所示。

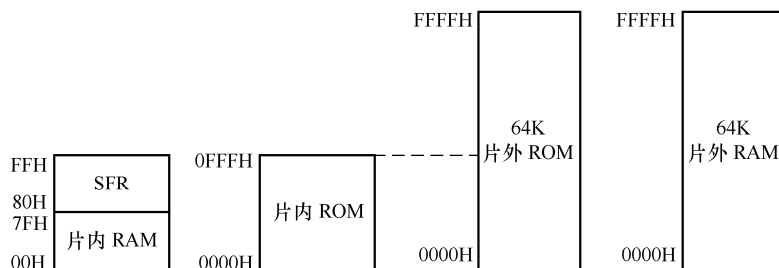


图 1-5 C51 单片机的存储空间分布图

从用户使用的角度看，C51 单片机的存储器地址空间分为 3 类：

- （1）片内片外统一编址 0000H~FFFFH 的 64KB 程序存储器地址空间（用 16 位地址）；
- （2）64KB 片外数据存储器地址空间，地址 0000H~FFFFH（用 16 位地址）；

(3) 256B 片内数据存储器地址空间（用 8 位地址）。

上述三个存储空间地址是重叠的, 如何区别这三个不同的逻辑空间呢? C51 单片机的汇编指令系统设计了不同的数据传送指令符号: CPU 访问片内、片外 ROM, 指令用 MOVC; 访问片外 RAM, 指令用 MOVX; 访问片内 RAM, 指令用 MOV。C51 单片机的 C 编程中设计了不同的数据存储类型: CPU 访问片内、片外 ROM, 数据类型用 code; 访问片外 RAM, 数据类型用 pdata 和 xdata; 访问片内 RAM, 数据类型用 data、bdata 和 idata。其具体应用在后续章节中说明。

1.3.2.1 数据存储器

1. 编址与访问

C51 单片机片内、片外数据存储器是两个独立的地址空间, 应分别单独编址。片内数据存储器除 RAM 块外, 还有特殊功能寄存器 (SFR) 块。对于 C51 子系列, 前者有 128byte, 其编址为 00H~7FH; 后者有 128byte, 其编址为 80H~FFH; 二者连续而不重叠。由于访问它们所用的指令不同, 并不会引起混乱。片外数据存储器一般是 16 位编址。数据存储器的编址如图 1-5 所示。

2. 片内数据存储器

如图 1-6 所示, 片内 RAM 位于片内寄存器的低 128 位, 包括工作寄存器区、位寻址区、数据缓冲区、堆栈区和特殊功能寄存器 (SFR)。

BYTE (MSB)				(LSB)							
7FH									位 寻 址 区	通 用 存 储 区	
2FH	7F	7E	7D	7C	7B	7A	79	78			
2EH	77	76	75	74	73	72	71	70			
2DH	6F	6E	6D	6C	6B	6A	69	68			
2CH	67	66	65	64	63	62	61	60			
2BH	5F	5E	5D	5C	5B	5A	59	58			
2AH	57	56	55	54	53	52	51	50			
29H	4F	4E	4D	4C	4B	4A	49	48			
28H	47	46	45	44	43	42	41	40			
27H	3F	3E	3D	3C	3B	3A	39	38			
26H	37	36	35	34	33	32	31	30			
25H	2F	2E	2D	2C	2B	2A	29	28			
24H	27	26	25	24	23	22	21	20			
23H	1F	1E	1D	1C	1B	1A	19	18			
22H	17	16	15	14	13	12	11	10			
21H	0F	0E	0D	0C	0B	0A	09	08			
20H	07	06	05	04	03	02	01	00			
1FH									工作寄存器区		
18H	寄存器 3 组										
17H	寄存器 2 组										
10H	寄存器 1 组										
0FH	寄存器 0 组										
08H											
07H											
00H											

图 1-6 C51 片内 RAM 地址空间

(1) 工作寄存器区

如图 1-6 所示, 00H~1FH 单元为工作寄存器区。工作寄存器也称通用寄存器, 用于临时寄存 8 位信息。工作寄存器分成 4 组, 每组都有 8 个寄存器, 用 R0~R7 来表示。程序中每次只用 1 组, 其他各组不工作。使用哪一组寄存器工作由程序状态字 (PSW) 中的 PSW.3 (RS0) 和 PSW.4 (RS1) 两位来选择, 其对应关系如表 1-2 所示。通过软件设置 RS0 和 RS1 两位的状态, 就可任意选一组工作寄存器工作。这个特点使 C51 单片机具有快速现场保护功能, 对于提高程序效率和响应中断的速度是很有利的。

表 1-2 工作寄存器组的选择

PSW.4 (RS1)	PSW.3 (RS0)	当前工作寄存器组 R0~R7
0	0	0 组 (00H~07H)
0	1	1 组 (08H~0FH)
1	0	2 组 (10H~17H)
1	1	3 组 (18H~1FH)

(2) 位寻址区

如图 1-6 所示, 20H~2FH 单元是位寻址区。这 16 个单元 (共计 $16 \times 8 = 128$ 位) 的每一位都赋予了一个位地址, 位地址范围为 00H~7FH。位寻址区的每一位都可当作软件触发器, 由程序直接进行位处理。通常可以把各种程序状态标志、位控制变量存于位寻址区内。

(3) 数据缓冲区

如图 1-6 所示, 30H~7FH 是数据缓冲区, 即用户 RAM 区, 共 80 个单元。

由于工作寄存器区、位寻址区、数据缓冲区统一编址, 使用同样的指令访问, 这 3 个区的单元既有自己独特的功能, 又可统一调度使用。因此, 前两个区未使用的单元也可作为用户 RAM 单元使用, 使容量较小的片内 RAM 得以充分利用。

(4) 堆栈和堆栈指针

堆栈是一种数据结构, 所谓堆栈就是只允许在其一端进行数据插入和数据删除操作的线性表。数据写入堆栈称为插入运算 (PUSH), 也叫入栈, 数据从堆栈中读出称之为删除运算 (POP), 也叫出栈; 堆栈的最大特点是“后进先出”的数据操作规则, 常把后进先出写为 LIFO (Last In First Out)。这里所说的进与出就是数据的入栈和出栈, 即先入栈的数据由于存放在栈的底部, 因此后出栈; 而后入栈的数据存放在栈的顶部, 因此先出栈。这跟往弹仓中压入子弹和从弹仓里弹出子弹的情形相类似。

堆栈具有如下特点。

- ① 堆栈是一个符合“先进后出、后进先出”的 RAM 区域;
- ② SP 总是指向堆栈的顶部 (保存有数据);
- ③ 堆栈可以设在内部 RAM 中的任意区域, 一般开辟在 30H~7FH 中。

堆栈的功能如下。

- ① 保护断点: 保护从主程序转向子程序、中断时的断点, 发生转移时自动完成。
- ② 保护现场: 对子程序、中断程序中要用到的、现场的某些寄存器的内容进行保护, 以保证返回时准确恢复。还可以通过软件指令方式实现。

③ 数据的临时存放。

(5) 专用寄存器

内部数据存储器的低 128 单元是为专用寄存器提供的，因此称为专用寄存器区，其单元地址为 80H~FFH，用于存放相应功能部件的控制命令、状态或数据。因为这些寄存器的功能已作专门的规定，故而称为专用寄存器（SFR），有时也称为特殊功能寄存器。C51 有 22 个专用寄存器，其中 21 个为可寻址。它包括程序计数器（PC）、累加器（ACC）、B 寄存器、程序状态字（PSW）、堆栈指针（SP）、数据指针（DPTR）、端口 P0~P3、串行数据缓冲器（SBUF）、串行口控制与状态寄存器（SCON）、定时器/计数器、定时器工作方式寄存器（TMOD）、定时器/计数器控制寄存器（TCON）、中断允许寄存器（IE）和中断优先级寄存器（IP）等，如表 1-3 所示。其中带*号的是可以位寻址的，即可以用“寄存器号、位”来表示。

表 1-3 专用寄存器表示符、名称、地址

符 号	名 称	地 址	符 号	名 称	地 址
*ACC	累加器	0E0H	*IE	中断允许控制寄存器	0A8H
*B	B 寄存器	0F0H	TMOD	定时/计数器工作方式寄存器	89H
*PSW	程序状态字	0D0H	*TCON	定时控制寄存器	88H
SP	栈指针	81H	TH0	定时器 0（高字节）	8CH
DPL	数据指针低 8 位	82H	TL0	定时器 0（低字节）	8AH
DPH	数据指针高 8 位	83H	TH1	定时器 1（高字节）	8DH
*P0	P0 口锁存寄存器	80H	TL1	定时器 1（低字节）	8BH
*P1	P1 口锁存寄存器	90H	*SCON	串行口控制寄存器	98H
*P2	P2 口锁存寄存器	0A0H	SBUF	串行数据缓冲器	99H
*P3	P3 口锁存寄存器	0B0H	PCON	电源控制及波特率选择寄存器	97H
*IP	中断优先级控制寄存器	0B8H			

① 程序计数器（PC）

PC 是一个 16 位的计数器，其内容为将要执行的指令地址，寻址范围达 64KB。PC 有自动加 1 功能，从而实现程序的顺序执行。PC 没有地址，是不可寻址的，因此用户无法对它进行读写。但可以通过转移、调用、返回等指令改变其内容，以实现程序的转移。

② 累加器（ACC）

ACC 为 8 位寄存器，是最常用的专用寄存器，功能较多，地位重要。它既可用于存放操作数，也可用来存放运算的中间结果。C51 单片机中大部分单操作数指令的操作数就取自 ACC，许多双操作数指令中的一个操作数也取自 ACC。

③ B 寄存器

B 寄存器是一个 8 位寄存器，主要用于乘除运算。乘法运算时，B 是乘数。乘法操作后，乘积的高 8 位存于 B 寄存器中。除法运算时，B 是除数。除法操作后，余数存于 B 寄存器中。此外，B 寄存器也可作为一般数据寄存器使用。

④ 程序状态字 (PSW)

PSW 是一个 8 位寄存器，用于寄存程序运行的状态信息。其中有些位状态是根据程序执行结果由硬件自动设置，而有些位状态则使用软件方法设定。PSW 的位状态可以用专门指令进行测试，也可以用指令读出。一些条件转移指令将根据 PSW 有关位的状态进行程序转移。

PSW 各位的定义如下。

位 序	PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
位 标 志	CY	AC	F0	RS1	RS0	OV	/	P

- 进位标志 CY 或 C (PSW.7)

其功能有二，一是存放算术运算的进位标志；二是在位操作中，作累加位使用，在位的传送、位与、位或等位操作中，都要使用进位标志。在执行某些算术操作类指令时，可被硬件或软件置位或清零。例如在 8 位加法运算时，若运算结果最高位 D7 有进位，则 C=1，否则 C=0；又如在 8 位减法运算时，若运算结果的最高位 D7 有借位，则 C=1，否则 C=0。半数以上的位操作指令与 C 有关，可见位处理时，它起着“位累加器”的作用。

- 辅助进位标志 AC (PSW.6)

8 位加法运算时，如果低半字节的最高位 D3 有进位，则 AC=1，否则 AC=0；在 8 位减法运算时，如果低半字节的最高位 D3 有借位，则 AC=1，否则 AC=0。AC 在作为 BCD 码运算而进行二至十进制调整时有用。

- 软件标志位 F0 (PSW.5)

这是用户定义的一个状态标志。可通过软件对它置位、清零；在编程时，也常测试其是否建起而进行程序分支。

- 工作寄存器组选择位 RS1、RS0 (PSW.4、PSW.3)

可借助软件置位或清零，以选定 4 个工作寄存器中的一个组投入工作，如表 1-4 所示。

表 1-4 工作寄存器组选择

RS1	RS0	寄存器组	RAM	RS1	RS0	寄存器组	RAM
0	0	第 0 组	00H~07H	1	0	第 2 组	10H~17H
0	1	第 1 组	08H~0FH	1	1	第 3 组	18H~1FH

- 溢出标志位 OV (PSW.2)

作为符号数加法、减法时由硬件置位或清零，以指示运算结果是否溢出，运算结果放累加器，OV=1，反映它已超出了累加器以补码形式表示一个有符号数的范围（-128~+127）。在乘法运算中，OV=1，表示乘积超过 255，即乘积分别在 B 与 A 中；反之，OV=0，表示积只在 A 中。在除法运算中，OV=1，表示除数为 0，除法不能进行；反之，OV=0，表示除数不为 0，除法可正常进行。

- 奇偶标志 P (PSW.0)

每执行一条指令，单片机都能根据 A 中 1 的个数的奇偶自动令 P 置位或清零，奇为 1，偶为 0。此标志对串行通信的数据传输非常有用，通过奇偶校验可检验传输的可靠性。

- 堆栈指针 (SP)

SP 是一个专用的 8 位寄存器, 用于指明堆栈顶部在内部 RAM 中的位置, 可用软件来设置初始值, 系统复位后 SP 为 07H, 使得堆栈从 08H 开始执行, 但在实际中, SP 指针一般设置在 30H~FFH 得范围内, 通常为 5FH。存取数据先进后出, 后进先出。

- 数据指针 (DPTR)

DPTR 为 16 位寄存器, 它是 C51 中唯一的一个 16 位寄存器。编程时, DPTR 既可以按 16 位寄存器使用, 也可以按两个 8 位寄存器分开使用, 即

DPH: DPTR 高位字节

DPL: DPTR 低位字节

在对 64KB 的外部数据寄存器空间寻址时, DPTR 通常可以作为间址寄存器使用, 访问程序存储器时可以作为基址寄存器使用。由于外部数据存储器的寻址范围为 64KB, 故把 DPTR 设计为 16 位。

- 端口 P0~P3

P0~P3 端口作为并行的 8 位 I/O 端口使用, 它们都是内部 RAM 的一个单元。4 个 8 位并行的 I/O 端口的锁存器 P0, P1, P2, P3 分别是专用寄存器 P0~P3, I/O 端口和内部 RAM 是统一编址的。

- 串行数据缓冲器 (SBUF)

C51 单片机的串行口是个全双工串行口, 可以用来发送和接收串行数据用作异步通信, 位于串行口内部的数据缓冲器由两个独立的寄存器组成, 一个为接收缓冲器, 另一个为发送缓冲器, 地址都是 99H, 写入 SBUF 的数据放在发送寄存器, 用于串行发送, 而当接收中断标志 RI=1, 从 SBUF 读取的数据来自接收缓冲器, 读取的是刚接收的数据。

- 串行口控制与状态寄存器 (SCON)

SCON 主要用来选择串行通信的工作方式, 设置状态标志。

- 定时器/计数器

C51 有 2 个 16 位的定时器/计数器, 即 T0 和 T1, 它们各自包含两个独立的 8 位加法计数器, TH0, TL0, TH1, TL1, 其分别是 T1, T2 的高 8 位与低 8 位加法计数器。

- 定时器工作方式寄存器 (TMOD)

TMOD 的作用是控制定时器/计数器 T0 和 T1 是作为定时器使用还是作为外部事件计数器使用, 以及控制定时器的工作方式, 并确定外部中断请求引脚是否参与 T0 和 T1 的操作控制。

- 定时器/计数器控制寄存器 (TCON)

TCON 的作用是控制定时器/计数器 T0 和 T1 的启停操作和定时器溢出中断标志处理, 另外用于对 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$ 两个外部中断源触发信号进行设置, 并对这两个中断源的脉冲触发中断申请标志进行处理。单片机复位时, 它们的每一位都被清零。

- 中断允许寄存器 (IE)

C51 有 5 个中断源, 中断允许寄存器的作用是控制 CPU 是否对各中断源的中断请求给予响应。可以根据要求用指令对 IE 中的各位状态进行置位和清零。单片机复位时, 它们中的每一位都被清零。

- 中断优先级寄存器 (IP)

IP 用于控制 CPU 对 5 个中断源优先响应的级别，它的低 5 位为各中断源优先级别的控制位，可以通过编程来置位或清零，单片机复位时 IP 中的各位将被清零，各中断源均为低优先级中断源。

3. 片外数据存储区

在进行数据采集和处理数据量较大时，片内的数据存储器空间往往不够使用，这时需要扩展数据存储区域，C51 具有扩展 64KB 的片外数据存储器的能力，具体的扩展在后续章节中介绍。

1.3.2.2 程序存储器

1. 编址与访问

计算机的工作是按照事先编制好的程序命令序列一条条顺序执行的，程序存储器就是用来存放这些已编好的程序和表格常数，它由只读存储器 ROM 或 EPROM 组成。计算机为了有序地工作，设置了一个专用寄存器——程序计数器 (PC)，用以存放将要执行的指令地址。每取出指令的 1 个字节后，其内容自动加 1，指向下一字节地址，使计算机依次从程序存储器取出指令予以执行，以完成某种程序操作。由于 C51 单片机的程序计数器为 16 位，因此可寻址的地址空间为 64KB，片内、片外统一编址 0000H~FFFFH，通过单片机引脚 EA 来判断是读取片内还是片外 ROM，EA 接地读片外 ROM，EA 接电源读片内 ROM。

2. 程序的特殊入口地址

0000H~0002H 是系统的启动单元。因为系统复位后，(PC) = 0000H。单片机从 0000H 单元开始取指令执行程序。使用时应当在这 3 个单元中存放一条无条件转移指令，以便直接转去执行指定的程序。

而 0003H~002AH 共 40 个单元被均匀地分为 5 段，每段 8 个单元，分别作为 5 个中断源的中断地址区。具体划分如表 1-5 所示。

表 1-5 C51 单片机复位、中断入口地址

地 址	说 明	地 址	说 明
0000H~0002H	系统复位	0013H~001AH	外部中断 1 中断地址区
0003H~000AH	外部中断 0 中断地址区	001BH~0022H	定时器/计数器 1 中断地址区
000BH~0012H	定时器/计数器 0 中断地址区	0023H~002AH	串行中断地址区

1.4 C51 单片机的并行输入/输出端口

C51 单片机有 4 个 8 位并行 I/O 口，P0~P3，共 32 根端口线。每个端口都包括锁存器 (即 SFR: P0~P3)、输出驱动器、两个三态缓冲器以及控制电路。4 个 I/O 端口都是准双向的并行 I/O 端口。

1.4.1 P0 口

1. P0 口结构

P0 口是一个三态双向口，可作为地址/数据分时复用口，也可作为通用 I/O 接口。其 1 位的内部结构如图 1-7 所示。P0 口由 8 个这样的电路组成。锁存器起输出锁存作用，8 个锁存器构成了特殊功能寄存器 P0；场效应管（FET）V1、V2 组成输出驱动器，以增大带负载能力；三态门 1 是引脚输入缓冲器；三态门 2 用于读锁存器端口；与门 3、反相器 4 及模拟转换开关构成了输出控制电路。

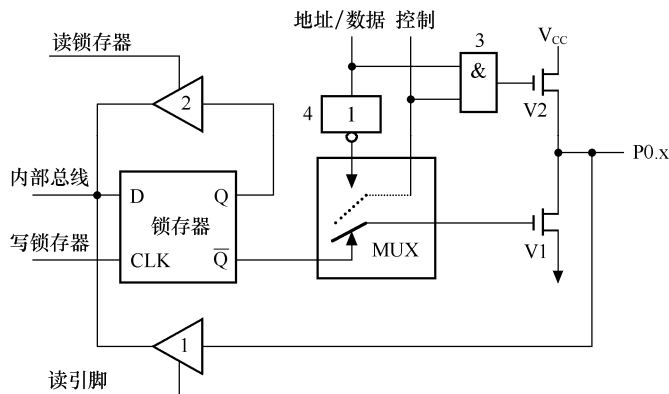


图 1-7 P0 口结构图

2. 地址/数据分时复用功能

当 P0 口作为地址/数据分时复用总线时，可分为两种情况：一种是从 P0 口输出地址或数据，另一种是从 P0 口输入数据。

在访问片外存储器而需从 P0 口输出地址或数据信号时，控制信号应为高电平“1”，使转换开关 MUX 把反相器 4 的输出端与 V1 接通，同时把与门 3 打开。当地址或数据为“1”时，经反相器 4 使 V1 截止，而经与门 3 使 V2 导通，P0.x 引脚上出现相应的高电平“1”；当地址或数据为“0”时，经反相器 4 使 V1 导通而 V2 截止，引脚上出现相应的低电平“0”。这样就将地址/数据的信号输出。

3. 通用 I/O 接口功能

当 P0 口作为通用 I/O 口使用，在 CPU 向端口输出数据时，对应的控制信号为 0，转换开关把输出级与锁存器 Q 端接通，同时因与门 3 输出为 0 使 V2 截止，此时，输出级是漏极开路电路。当写脉冲加在锁存器时钟端 CLK 上时，与内部总线相连的 D 端数据取反后出现在 Q 端，又经输出 V1 反相，在 P0 引脚上出现的数据正好是内部总线的数据。当要从 P0 口输入数据时，引脚信息仍经输入缓冲器进入内部总线。

(1) 在输出数据时，由于 V2 截止，输出级是漏极开路电路，要使“1”信号正常输出，必须外接上拉电阻。

P0=P0&A	; (P0) ← (P0) & (A)
P0=P0 #data	; (P0) ← (P0) data
P0=P0-1	; (P0) ← (P0) -1

综上所述，P0 口在有外部扩展存储器时被作为地址/数据总线口，此时，其是一个真正的双向口；在没有外部扩展存储器时，P0 口也可作为通用的 I/O 接口，但此时只是一个准双向口。另外，P0 口的输出级具有驱动 8 个 LSTTL 负载的能力，即输出电流不大于 800μA。

当 P1 口用作输入口时, 也应先用软件使输出锁存器置 1, 使 FET 截止, 处于高阻状态,



然后再通过缓冲器进行输入操作。

1.4.3 P2 口

P2 口为准双向口，其 1 位的内部结构如图 1-9 所示。它可作输入输出使用，还可作为高 8 位地址总线口使用，扩展系统中，作为高 8 位地址总线，与 P0 口的低 8 位共同组成 16 位地址总线。P0 口，P2 口作为数据/地址总线使用时，不能再作为通用 I/O 使用。当作为准双向通用 I/O 口使用时，控制信号使转换开关接向左侧，锁存器 Q 端经反相器 3 接 V1，其工作原理与 P1 口相同，也具有输入、输出、端口操作三种工作方式，负载能力也与 P1 口相同。

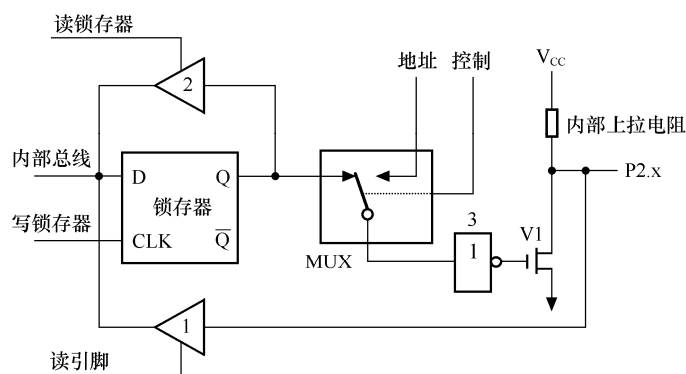


图 1-9 P2 口内部结构图

当作为外部扩展存储器的高 8 位地址总线使用时，控制信号使转换开关接向右侧，由程序计数器（PC）来的高 8 位地址 PCH，或数据指针（DPTR）来的高 8 位地址 DPH 经反相器 3 和 V1 原样呈现在 P2 口的引脚上，输出高 8 位地址 A8H~A15H。在上述情况下，端口锁存器的内容不受影响，所以，取指或访问外部存储器结束后，由于转换开关又接至左侧，使输出驱动器与锁存器 Q 端相连，引脚上将恢复原来的数据。

1.4.4 P3 口

图 1-10 所示为 P3 口的 1 位内部结构图，它具有第二输出功能，既可作为输入/输出口使用，又能涉及串口、外部中断、定时器等。扩展外部数据存储器时，WR 和 RD 作为控制线使用。

当 P3 口作为通用 I/O 口使用时，第二输出功能端应保持高电平，打开与非门，使锁存器输出 Q 端的状态能顺利通过与非门送至引脚上。输入时，先置输出锁存器为 1，使 FET 截止，再通过三态缓冲器读引脚信息。

当 P3 口作为第二输出功能使用时，应先将输出锁存器置 1，使与非门畅通。输出时，第二输出功能端的信息通过与非门送至引脚上。输入时，也先置输出锁存器为 1，使 FET 截止，引脚上的第二输入功能信号经第一个缓冲器输入。不论作为输入口使用还是第二功能信号输



引 脚	第 二 功 能	
P3.0	RXD	串行口输入端
P3.1	TXD	串行口输出端
P3.2	INT0	外部中断 0 请求输入端，低电平有效
P3.3	INT1	外部中断 1 请求输入端，低电平有效
P3.4	T0	定时器/计数器 0 计数脉冲输入端
P3.5	T1	定时器/计数器 1 计数脉冲输入端
P3.6	WR	外部数据存储器写选通信号输出端，低电平有效
P3.7	RD	外部数据存储器读选通信号输出端，低电平有效

- (1) 如果单片机内部有程序存储器, 不需要扩展外部存储器和 I/O 接口, 单片机的 4 个口均可作 I/O 口使用。
- (2) 四个口在作为输入口使用时, 应先对其写 “1”, 以避免误读。
- (3) P0 口作为 I/O 口使用时, 应外接 10k Ω 的上拉电阻, 其他口则可不接。
- (4) P2 口的某几根线作为地址使用时, 剩下的线不能作为 I/O 口线使用。
- (5) P3 口的某些口线作为第二功能时, 剩下的口线可以单独作为 I/O 口线使用。

C51 单片机有 44 个引脚的方形封装形式和 40 个引脚的双列直插式封装形式，其中最常

用的是 40 个引脚封装，如图 1-11 所示。

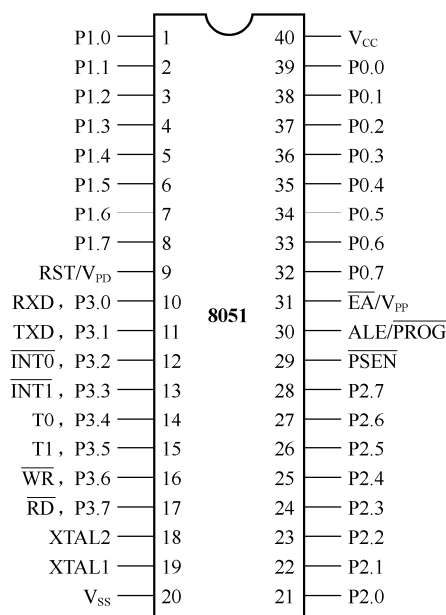


图 1-11 C51 单片机引脚图

1.5.1 C51 单片机引脚功能说明

1. 主电源引脚

V_{CC} (40 脚): 接+5 V 电源正端。

V_{SS} (20 脚): 接+5 V 电源地端。

2. 外接石英晶体引脚

XTAL1 (19 脚) 和 XTAL2 (18 脚): 使用内部振荡电路时, 用来接石英晶体和电容; 使用外部时钟时, 用来输入时钟脉冲。

3. 输入/输出引脚

(1) P0 口 (39~32 脚): P0.0~P0.7 统称为 P0 口。在不接片外存储器与不扩展 I/O 口时, 可作为准双向输入/输出口; 在接有片外存储器或扩展 I/O 口时, P0 口分时复用为低 8 位地址总线和双向数据总线。

(2) P1 口 (1~8 脚): P1.0~P1.7 统称为 P1 口, 可作为准双向 I/O 口使用。

(3) P2 口 (21~28 脚): P2.0~P2.7 统称为 P2 口, 一般可作为准双向 I/O 口使用; 在接有片外存储器或扩展 I/O 口且寻址范围超过 256byte 时, P2 口用作高 8 位地址总线。

(4) P3 口 (10~17 脚): P3.0~P3.7 统称为 P3 口, 除作为准双向 I/O 口使用外, 还可以将每一位用于第二功能, 而且 P3 口的每一条引脚均可独立定义为第一功能的输入输出或第一功能。

4. 控制线

(1) $\overline{\text{ALE/PROG}}$ (30 脚): 地址锁存有效信号输出端。ALE 在每个机器周期内输出两个脉冲。在访问片外程序存储器期间, 下降沿用于控制锁存 P0 输出的低 8 位地址; 在不访问片外程序存储器期间, 可作为对外输出的时钟脉冲或用于定时目的。但要注意, 在访问片外数据存储器期间, ALE 脉冲会跳空一个, 此时作为时钟输出就不妥了 (详见 CPU 时序)。

对于片内含有 EPROM 的机型, 在编程期间, 该引脚用作编程脉冲 PROG 的输入端。

(2) $\overline{\text{PSEN}}$ (29 脚): 片外程序存储器读选通信号输出端, 低电平有效。在从外部程序存储器读取指令或常数期间, 每个机器周期该信号两次有效, 以通过数据总线 P0 口读回指令或常数。在访问片外数据存储器期间, $\overline{\text{PSEN}}$ 信号将不出现。

(3) RST/V_{PD} (9 脚): RST 即为 RESET, V_{PD} 为备用电源。该引脚为单片机的上电复位或掉电保护端。当单片机振荡器工作时, 该引脚上出现持续两个机器周期的高电平, 就可实现复位操作, 使单片机回复到初始状态。上电时, 考虑到振荡器有一定的起振时间, 该引脚上高电平必须持续 10ms 以上才能保证有效复位。

当 V_{CC} 发生故障, 降低到低电平规定值或掉电时, 该引脚可接上备用电源 V_{PD} (+5V) 为内部 RAM 供电, 以保证 RAM 中的数据不丢失。

(4) $\overline{\text{EA/V}}_{\text{PP}}$ (31 脚): EA 为片外程序存储器选用端。该引脚有效 (低电平) 时, 只选用片外程序存储器, 否则单片机上电或复位后选用片内程序存储器。

对于片内含有 EPROM 的机型, 在编程期间, 此引脚用作 21 V 编程电源 V_{PP} 的输入端。

综上所述, C51 系列单片机的引脚特点, 可归纳为以下两点。

(1) 单片机功能多, 引脚数少, 因而许多引脚都具有第二功能。

(2) 单片机对外呈现 3 总线形式, 由 P2 口, P0 口组成 16 位地址总线; 由 P0 口分时复用为数据总线; 由 ALE, $\overline{\text{PSEN}}$, RST, $\overline{\text{EA}}$ 与 P3 口中的 $\overline{\text{INT0}}$, $\overline{\text{INT1}}$, T0, T1, $\overline{\text{WR}}$, $\overline{\text{RD}}$ 共 10 个引脚组成控制总线。由于是 16 位地址线, 因此可使片外存储器的寻址范围达到 64KB。

1.5.2 复位电路

1. 复位状态

复位操作使单片机的片内电路初始化, 使单片机从一种确定的状态开始运行。

当 C51 系列单片机的复位引脚 RST 出现两个机器周期以上的高电平时, 单片机就完成了复位操作。如果 RST 持续为高电平, 单片机就处于循环复位状态。

复位操作通常有如下 2 种基本形式:

上电复位: 上电复位要求接通电源后, 自动实现复位;

开关复位: 开关复位要求在电源接通的条件下, 在单片机运行期间, 用按钮开关操作使单片机复位。

单片机在 RST 引脚高电平的控制下, 专用寄存器和程序计数器 (PC) 复位后的状态如表 1-7 所示。

表 1-7 复位后专用寄存器的初始状态

SFR 名称	初 始 状 态	SFR 名称	初 始 状 态
ACC	00H	TMOD	00H
B	00H	TCON	00H
PSW	00H	TH0	00H
SP	07H	TL0	00H
DPL	00H	TH1	00H
DPH	00H	TL1	00H
P0~P3	FFH	SBUF	不确定
IP	XXX00000B	SCON	00H
IE	0XX00000B	PCON	0XXXXXXXB

(PSW)=00H, 由于 RS1 (PSW.4)=0, RS0 (PSW.3)=0, 复位后单片机选择工作寄存器 0 组。

(SP)=07H, 复位后堆栈在片内 RAM 的 08H 单元处建立。TH1, TL1, TH0, TL0 的内容为 00H, 定时器/计数器的初值为 0。

(TMOD)=00H, 复位后定时器/计数器 T0, T1 为定时器方式 0, 非门控方式。

(TCON)=00H, 复位后定时器/计数器 T0, T1 停止工作, 外部中断 0、1 为电平触发方式。

(SCON)=00H, 复位后串行口工作在移位寄存器方式, 且禁止串行口接收。

(IE)=00H, 复位后屏蔽所有中断。

(IP)=00H, 复位后所有中断源都设置为低优先级。

P0 口~P3 口锁存器都是全 1 状态, 说明复位后 4 个并行接口设置为输入口。

2. 复位电路

与其他计算机一样, C51 单片机系统常常有上电复位和按钮复位两种方法, 如图 1-12 所示。所谓上电复位, 是指计算机加电瞬间, 要在 RST 引脚上出现大于 10ms 的正脉冲, 使单片机进入复位状态。按钮复位是指用户按下“复位”按钮, 使单片机进入复位状态。

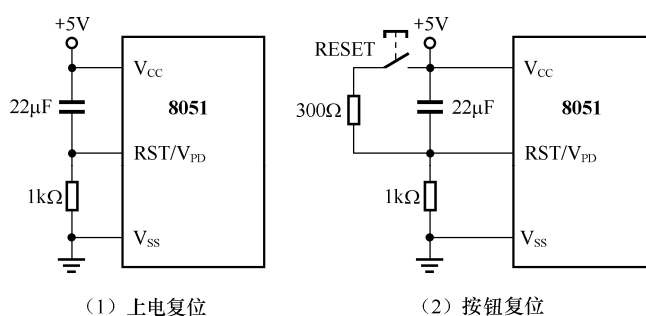


图 1-12 C51 单片机复位电路

复位是靠外部电路实现的。图 1-13 所示为上电复位及按钮复位的一种实用电路。

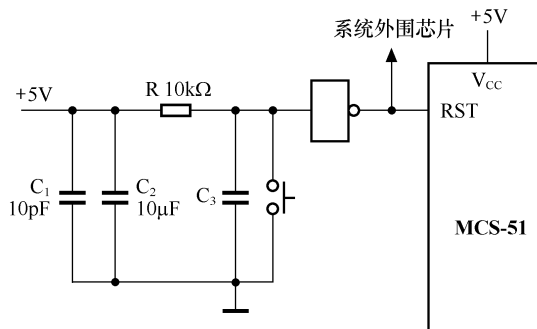


图 1-13 C51 单片机上电复位及按钮复位的一种实用电路

1.5.3 CPU 时序

1. 振荡电路和时钟电路

(1) 内部时钟信号的产生

通过外接晶振和电容，利用单片机内部电路产生时钟信号。

(2) 引入外部脉冲信号

在由多片单片机组成的系统中，为了各单片机之间时钟信号的同步，应当引入唯一的公用外部脉冲信号作为各单片机的振荡脉冲。

2. 时序定时单位

(1) 拍节与状态

把振荡脉冲的周期定义为拍节（用“P”表示）。振荡脉冲经过二分频后，就是单片机的时钟信号，把时钟信号的周期定义为状态（用“S”表示）。这样，一个状态就包含两个拍节，其前半周期对应的拍节叫拍节 1（P₁），后半周期对应的拍节叫拍节 2（P₂）。

(2) 机器周期

C51 单片机采用同步控制方式，因此它呈固定的机器周期。规定一个机器周期的宽度为 6 个状态，并依次表示为 S₁~S₆。由于一个状态又包括两个拍节，因此一个机器周期总共有 12 个拍节，分别记作 S₁P₁, S₁P₂, ..., S₆P₂。由于一个机器周期共有 12 个振荡脉冲周期，因此机器周期就是振荡脉冲的十二分频。

当振荡脉冲频率为 12MHz 时，一个机器周期为 1μs。当振荡脉冲频率为 6MHz 时，一个机器周期为 2μs。

(3) 指令周期

指令周期是最大的时序定时单位。指令周期、机器周期、时钟周期和振荡周期的关系如图 1-14 所示。执行一条指令所需要的时间称之为指令周期。指令周期以机器周期的数目来表示，C51 单片机的指令周期根据指令的不同，可包含有 1、2、3 或 4 个机器周期。

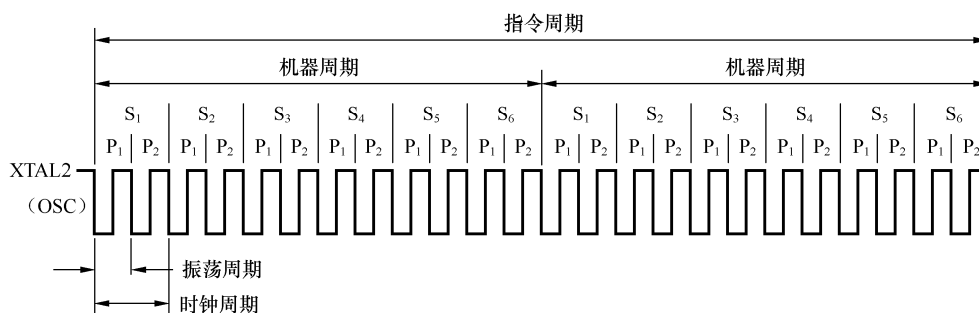


图 1-14 C51 单片机指令周期组成图

3. 典型指令时序

C51 单片机共有 111 条指令，全部指令按其长度可分为单字节指令、双字节指令和三字节指令。图 1-15 所示为单周期指令时序。

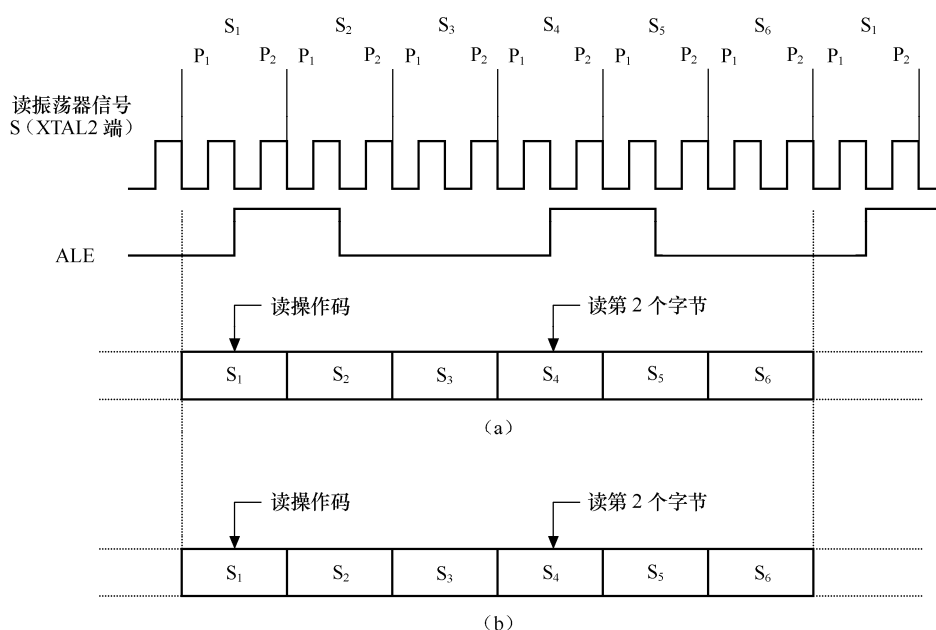


图 1-15 C51 单片机单周期指令时序

(1) 单字节单周期指令（例如：寄存器 A 增 1 指令）

由于是单字节指令，因此只需进行一次读指令操作。当第二个 ALE 有效时，由于 PC 没有加 1，所以读出的还是原指令，属于一次无效的操作。

(2) 双字节单周期指令（例如：加法指令）

这种情况下对应于 ALE 的两次读操作都是有效的，第 1 次是读指令操作码，第 2 次是读指令第 2 字节（本例中是立即数）。

(3) 单字节双周期指令（例如：地址指针 DPTR 增 1 指令）

两个机器周期共进行 4 次读指令的操作，但其中后 3 次的读操作全是无效的。

(4) 双字节双周期指令（例如：读外部存储器类指令）。

1.6 C51 单片机的工作方式

C51 单片机共有复位、程序执行、单步执行、掉电保护、低功耗以及 EPROM 编程和校验 6 种工作方式，本节介绍其中的 4 种。

1.6.1 复位方式

(1) 复位操作：复位是单片机的初始化操作，其主要功能是把 PC 初始化为 0000H，使单片机从 0000H 单元开始执行程序。

(2) 复位信号及其产生：RST 引脚是复位信号的输入端，复位信号对高电平有效。

(3) 复位方式：上电自动复位和按键手动复位两种方式。

1.6.2 程序执行方式

程序执行方式是单片机的基本工作方式。由于复位后 PC=0000H，因此程序执行总是从地址 0000H 开始。但一般程序并不是真正从 0000H 开始，为此就得在 0000H 开始的单元中存放一条无条件转移指令，以便跳转到实际程序的入口去执行。

单片机所执行的程序可以放在内部 ROM、外部 ROM 或者同时放在内外 ROM 中。程序存储到 ROM 中，一般有两种形式：一种是使用 EPROM（可擦除的程序存储器）作为程序存储器时，利用编程器将程序写入；另一种是在批量生产时，由生产厂家直接将程序固化在 ROM 中。

单步执行方式是指单片机在某个按键的控制下一条一条地执行用户程序中指令的方式，即按一次按键就执行一条用户指令。这种方式常用于系统开发过程中用户程序的调整。

1.6.3 掉电保护方式

单片机系统在运行过程中，如发生掉电故障，将会使系统数据丢失，其后果有时是很严重的。为此，C51 单片机设置有掉电保护措施，进行掉电保护处理。其具体做法是：先把有用数据转存，然后再启用备用电源维持供电。

(1) 数据转存

所谓数据转存，是指当电源出现故障时，应立即将系统的有用数据转存到内部 RAM 中。数据转存是通过中断服务程序完成的，即通常所说的“掉电中断”。

因为单机电源端 (V_{CC}) 都接有滤波电容，掉电后电容蓄存的电能尚能维持有效电压达几个 ms 之久，足以完成一次掉电中断操作。为此应在单片机系统中设置一个电压检测电路，一旦检测到电源电压下降，立即通过 $\overline{INT0}$ 或 $\overline{INT1}$ 产生外部中断请求，中断响应后执行中断服务程序，把有用数据送到内部 RAM 中保护起来。

(2) 接通备用电源

为了保存转存后的有用数据，掉电后应给内部 RAM 供电。为此，系统应预先装有备用

电源，并在掉电后立即接通备用电源。备用电源由单片机的 RST/VPD 引脚接入。为了在掉电时能及时接通备用电源，系统中具有备用电源与 V_{CC} 电源的自动切换电路。

1.6.4 C51 单片机的低功耗方式

C51 单片机有两种低功耗方式，即待机方式和掉电保护方式。

(1) 待机方式

如果使用指令使 PCON 寄存器 IDL 位置“1”，则 C51 即进入待机方式。这时振荡器仍然工作，并向中断逻辑、串行口和定时器/计数器电路提供时钟，但向 CPU 提供时钟的电路被阻断，因此 CPU 不能工作，与 CPU 有关的如 SP, PC, PSW, ACC 以及全部通用寄存器也都被“冻结”在原状态。

在待机方式下，中断功能应继续保留，以便采用中断方法退出待机方式。为此，应引入一个外中断请求信号，在单片机响应中断的同时，PCON.0 位被硬件自动清“0”，单片机就退出待机方式而进入正常工作方式。其实在中断服务程序中只需安排一条 RETI 指令，就可以使单片机恢复正常工作后返回断点继续执行程序。

(2) 掉电保护方式

PCON 寄存器的 PD 位控制单片机进入掉电保护方式，因此对 C51 单片机，在检测到电源故障时，除进行信息保护外，还应把 PCON.1 位置“1”，使之进入掉电保护方式。此时单片机一切工作都停止，只有内部 RAM 单元的内容被保存。

习题和思考题

1-1 在下列情况中， $\overline{EA}$ 引脚应接何种电平？

(1) 只有片内 ROM, $\overline{EA}$ = _____; (2) 只有片外 ROM, $\overline{EA}$ = _____;

(3) 有片内、片外 ROM, $\overline{EA}$ = _____; (4) 有片内 ROM 但不用，而用片外 ROM, $\overline{EA}$ = _____。

1-2 10 根地址线可选 _____ 个存储单元，32KB 存储单元需要 _____ 根地址线。

1-3 C51 单片机复位后，R4 所对应的存储单元的地址为 _____，因上电复位时，PSW = _____。这时当前的工作寄存器区是 _____ 组工作寄存器区。

1-4 内 RAM 中，位地址为 30H 的位，该位所在字节的字节地址为 _____。

1-5 C51 单片机的存储空间分为哪几部分？它们的寻址范围是多少？

1-6 决定程序执行顺序的寄存器是哪一个？为什么 C51 单片机都是从程序存储器的 0000H 单元开始执行？

1-7 若 C51 单片机的晶振频率 $f_{OSC} = 12\text{MHz}$ ，则时钟周期、状态周期、机器周期各是多少？

1-8 什么叫指令周期？什么叫指令字节？

第2章 C51 单片机的指令系统

重点与难点

本章重点是掌握 C51 系列单片机的寻址方式、数据传送类指令、算术运算类指令、逻辑运算与移位类指令、控制转移类指令、位操作类指令。本章难点是操作指令对标志位的影响。

基本要求

1. 掌握 C51 系列单片机的寻址方式
2. 掌握数据传送类指令的格式、功能和使用方法
3. 掌握算术运算类指令的格式、功能和使用方法
4. 掌握逻辑运算与移位类指令的格式、功能和使用方法
5. 掌握控制转移类指令的格式、功能和使用方法
6. 掌握位操作类指令的格式、功能和使用方法

本章内容

1. C51 单片机的指令系统概述
2. C51 单片机的寻址方式
3. C51 单片机的指令说明
4. C51 单片机的伪指令
5. C51 单片机的程序设计举例

2.1 C51 单片机的指令系统概述

1. 指令、指令系统的概念

指令是使计算机内部执行某一种操作而提供给用户编程使用的一种命令。由构成计算机的电子器件特性所决定。计算机只能识别二进制代码。以二进制代码来描述指令功能的语言，称之为机器语言。由于机器语言不便被人们识别、记忆、理解和使用，因此给每条机器语言指令赋予助记符号来表示，这就形成了汇编语言。也就是说，汇编语言是便于人们识别、记忆、理解和使用的一种指令形式，它和机器语言指令一一对应，也是由计算机的硬件特性所决定的。

计算机能够执行的全部操作所对应的指令集合，称为这种计算机的指令系统。从指令是

反映计算机内部的一种操作来看，指令系统全面展示出了计算机的操作功能，也就是它的工作原理；从用户使用的角度来看，指令系统是提供给用户使用计算机功能的软件资源。要让计算机处理问题，首先要编写程序，编写程序实际上是从指令系统中挑选一个指令子集的过程。因此，学习指令系统既要从编程使用的角度掌握指令的使用格式及每条指令的功能；又要掌握每条指令在计算机内部的微观操作过程，即工作原理，从而进一步加深对硬件组成原理的理解。

指令一般有功能、时间和空间三种属性。功能属性是指每条指令都对应一个特定的操作功能；时间属性是指一条指令执行所用的时间，一般用机器周期来表示；空间属性是指一条指令在程序存储器中存储所占用的字节数。这3种属性在使用中最重要的是功能属性，但时间属性和空间属性在有些场合也要用到。如一些实时控制应用程序中，有时需要计算一个程序段的确切执行时间，或编写软件延时程序，都要用到每条指令的时间属性；在程序存储器的空间设计，或相对转移指令的偏移量计算时就要用到指令的空间属性。

指令的描述形式有两种，即机器语言形式和汇编语言形式，这样每条指令有如下两种不同的表达形式：

(1) 二进制代码（机器码）指令

(2) 助记符（汇编语言）指令

助记符指令： **MOV A, # 30H**

二进制代码指令： **01110100 00110000**

助记符指令必须转换成二进制代码指令才能存入存储器。

本章仅介绍助记符指令格式。

在实际应用中主要采用汇编语言形式描述计算机指令系统。采用机器语言编写的程序称之为目标程序。采用汇编语言编写的程序称之为源程序。计算机能够直接识别并执行的只有机器语言。汇编语言程序不能被计算机直接识别并执行，必须经过一个中间环节把它翻译成机器语言程序，这个中间过程叫做汇编。汇编有两种方式：机器汇编和手工汇编。机器汇编是用专门的汇编程序，在计算机上进行翻译；手工汇编是程序员把汇编语言指令逐条翻译成机器语言指令。现在主要使用机器汇编，但有时也用到手工汇编。

2. 指令的格式

指令系统中的指令描述了不同的操作，不同操作对应不同的指令。但结构上，每条指令通常由操作码和操作数两部分组成。操作码表示计算机执行该指令将进行何种操作，操作数表示参加操作的数本身或操作数所在的地址。C51 单片机的指令有无操作数、单操作数、双操作数3种情况。汇编语言指令有如下的格式：

[标号:] 操作码助记符 [目的操作数][, 源操作数] [; 注释]

标号是由字母开头的字母数字序列。在C51单片机指令中一条指令占一行，每行都可以有一个标号，每个子程序的第一条语句都必须有一个标号，以标识该子程序的入口地址，也充当该子程序的名字。在程序跳转时，有时也需要一个标号，标识程序跳至的入口。

操作码用来指定指令的功能，而操作数则是指令操作的对象。比如做加法运算，“ADD A, R0”就表示将寄存器A和R0中的数据相加然后将结果存放到寄存器A中。在这里ADD即为操作码，A和R0即为操作数。指令有定长和不定长之分，定长指令其操作码的位数为

一定值，不定长指令其操作码为变动的，一般使用频率最高的是采用最短的操作码。单片机一般采用的是不定长指令格式，根据指令的长短又可将指令分为一字节指令、二字节指令、三字节指令。一字节指令即在程序存储器中需要一个字节的单元来存储；二字节指令即在程序存储器中需要二个字节的单元来存储；三字节指令即在程序存储器中需要三个字节的单元来存储。

注释是对指令和程序添加的一些说明和帮助，对提高程序的可读性有很大作用。每条指令后面都可以有注释。在程序中的‘;’后的内容就是注释，汇编程序在编译程序时会忽略注释。

3. C51 单片机的指令

C51 单片机指令系统具有功能强、指令短、执行快等特点，共有 111 条指令。从功能上可划分成数据传送、算术运算、逻辑运算、程序控制转移、位操作等五大类。从空间属性上分为单字节指令（49 条）、双字节指令（46 条）和最长的三字节指令（只有 16 条）。从时间属性上可分成单机器周期指令（64 条）、双机器周期指令（45 条）和 4 个机器周期的指令（只有乘法和除法两条）。可见，C51 单片机指令系统在存储空间和执行时间方面具有较高的效率。

4. 指令中有关操作数符号的说明

为便于后面的学习，在这里先对描述指令的一些符号的约定意义作一说明。

(1) Ri 和 Rn: 表示当前工作寄存器区中的工作寄存器，i 取 0 或 1，表示 R0 或 R1；n 取 0~7，表示 R0~R7。

(2) #data: 表示包含在指令中的 8 位立即数。

(3) #data16: 表示包含在指令中的 16 位立即数。

(4) rel: 以补码形式表示的 8 位相对偏移量，范围为-128~127，主要用在相对寻址的指令中。

(5) addr16 和 addr11: 分别表示 16 位直接地址和 11 位直接地址。

(6) direct: 表示直接寻址的地址。

(7) bit: 表示可位寻址的直接位地址。

(8) (X): 表示 X 单元中的内容。

(9) ((X)): 表示以 X 单元的内容为地址的存储器单元内容，即 (X) 作地址，该地址单元的内容用 ((X)) 表示。

(10) “/” 和 “→” 符号: “/” 表示对该位操作数取反，但不影响该位的原值。“→” 表示操作流程，将箭尾一方的内容送入箭头所指另一方的单元中去。

2.2 C51 单片机的寻址方式

所谓寻址就是指寻找操作数的地址。由于大多数指令都需要操作数，而操作数往往存储在存储器中，因此在使用操作数的过程中就存在一个寻找存储单元的问题。在 C51 系列单片

机中共有 7 种寻址方式：立即寻址、寄存器寻址、直接寻址、寄存器间接寻址、变址寻址、相对寻址和位寻址。

2.2.1 立即寻址

操作数直接出现在指令中，紧跟在操作码的后面，作为指令的一部分与操作码一起存放在程序存储器中，可以立即得到并执行，不需要经过别的途径去寻找，故称为立即寻址。汇编指令中，在一个数的前面冠以“#”符号作前缀，就表示该数为立即寻址。

例如： `MOV A, #30H`

指令中 30H 就是立即数。这一条指令的功能是执行将立即数 30H 传送到累加器 A 中的操作。该指令操作码的机器代码为 74H，占用一个字节存储单元，立即数 30H 存放在紧跟在其后的一个字节存储单元中，成为指令代码的一部分。整条指令的机器码为 74H 30H。

2.2.2 寄存器寻址

在指令选定的某寄存器中存放或读取操作数，以完成指令规定的操作，称为寄存器寻址。

寻址对象：A, B, DPTR, R0~R7。

A 可以寄存器寻址又可以直接寻址，直接寻址时写作 ACC，B 仅在乘除法指令中为寄存器寻址，在其他指令中为直接寻址。

例如： `MOV A, R0`

指令中源操作数和目的操作数都是寄存器寻址。该指令的功能是把工作寄存器 R0 中的内容传送到累加器 A 中，如 R0 中的内容为 30H，则执行该指令后 A 的内容也为 30H。

寄存器寻址按所选定的工作寄存器 R0~R7 进行操作，指令机器码的低 3 位的八种组合 000, 001, ..., 110, 111 分别指明所用的工作寄存器 R0, R1, ..., R6, R7。

例如： `MOV A, Rn (n=0~7)`

这 8 条指令对应的机器码分别为 E8H~EFH。

在汇编指令中，累加器 A 作为寄存器寻址的操作数，但在机器码中无需指明，它由操作码隐含。

2.2.3 寄存器间接寻址

由指令指出某一寄存器的内容作为操作数地址的寻址方法，称为寄存器间接寻址。这里需要强调的是：寄存器中的内容不是操作数本身，而是操作数的地址，只有到该地址单元中才能得到操作数，寄存器起地址指针的作用。

例如： `MOV A, @R1`

指令的源操作数是寄存器间接寻址。该指令的功能是将以工作寄存器 R1 中的内容为地址的片内 RAM 单元的数据传送到 A 中去。例如，若 R1 中的内容为 80H，片内 RAM 地址为 80H 的单元中的内容为 2FH，则执行该指令后，片内 RAM 80H 单元的内容 2FH 被送到 A 中。

寄存器间接寻址示意图如图 2-1 所示。

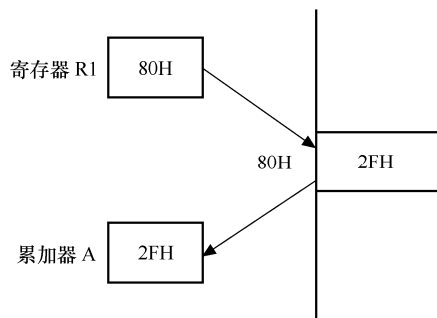


图 2-1 寄存器间接寻址示意图

2.2.4 直接寻址

指令中直接给出操作数所在的存储器地址，以供取数或存数的寻址方式称为直接寻址。

寻址对象：

- (1) 内部数据存储器：使用它的地址；
- (2) 专用寄存器：既可使用它的地址，也可以直接使用寄存器名。

例如： `MOV A, 40H`

指令中的源操作数就是直接寻址，40H 为操作数的地址。该指令的功能是把片内 RAM 地址为 40H 单元的内容送到累加器 A 中。该指令的机器码为 E5H 40H，8 位直接地址在指令操作码中占一个字节。

C51 系列单片机的直接寻址可用于访问片内、外数据存储器，也可用于访问程序存储器。

直接寻址可访问片内 RAM 的低 128 个单元 (00H~7FH)，同时也是用于访问高 128 个单元的特殊功能寄存器 (SFR) 的唯一方法。由于 C52 子系列的片内 RAM 有 256 个单元，其高 128 个单元与 SFR 的地址是重叠的。为了避免混乱，单片机规定：直接寻址的指令不能访问片内 RAM 的高 128 个单元 (80H~FFH)，若要访问这些单元只能用寄存器间接寻址指令，而要访问 SFR 只能用直接寻址指令。另外，访问 SFR 可在指令中直接使用该名寄存器的名字来代替地址，如 `MOV A, 80H`，可以写成 `MOV A, P0`，因为 P0 口的地址为 80H。

直接寻址访问程序存储器的转移、调用指令中直接给出了程序存储器的地址，执行这些指令后，程序计数器 (PC) 的内容将更换为指令直接给出的地址，机器将改为访问以所给地址为起始地址的存储区间，取指令 (或取数)，并依次执行。

2.2.5 变址寻址

以 DPTR 或 PC 寄存器内容为基地址，和累加器 A 的内容相加形成操作数的地址。其中累加器 A 内容是可变的。变址寻址示意图如图 2-2 所示。

例如：MOVC A, @A+DPTR

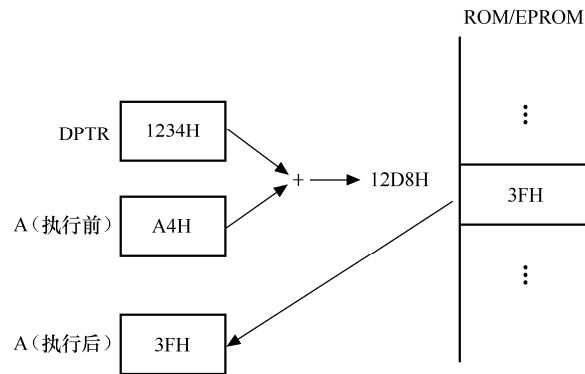


图 2-2 变址寻址示意图

2.2.6 相对寻址

相对寻址是以当前程序计数器（PC）值加上指令中给出的偏移量 *rel* 而构成实际操作数地址的寻址方法。它用于访问程序存储器，常出现在相对转移指令中。

例如：JZ 61H

在使用相对寻址时要注意以下两点。

（1）当前 PC 值是指相对转移指令的存储地址加上该指令的字节数。例如，JZ *rel* 是一条累加器 A 为零就转移的双字节指令。若该指令的存储地址为 2050H，则执行该指令时的当前 PC 值即为 2052H，即当前 PC 值是对相对转移指令取指结束时的值。

（2）偏移量 *rel* 是有符号的单字节数，以补码表示，其值的范围是 -128 ~ +127 (00H ~ FFH)，负数表示从当前地址向前转移，正数表示从当前地址向后转移。所以，相对转移指令满足条件后，转移的地址（目的地址）为：

目的地址 = 当前 PC 值 + *rel* = 指令存储地址 + 指令字节数 + *rel*

2.2.7 位寻址

位寻址是在位操作指令中直接给出位操作数的地址，可以对片内 RAM 中的 128 位和特殊功能寄存器（SFR）中的 93 位进行寻址。

例如：MOV C, 20H; 20H 位的内容送 CY 标志位，C 称为位累加器。

MOV A, 20H; 字节寻址，将内部 RAM 中 20H 单元中的内容送给累加器 A。

以上两条指令均为寻址，究竟是位寻址还是字节寻址，根据两操作数类型一致的原则，由另一个操作数决定。

2.2.8 寻址方式总汇

指令系统的寻址方式和寻址空间如表 2-1 所示。

表 2-1 指令系统的寻址方式和寻址空间

序 号	寻 址 方 式	使用符号和变量	寻 址 空 间
1	直接寻址	Direct 或 data、data16	内部 RAM 128B、SFR 数据存储器（片外 RAM）64KB 访问 SFR 的唯一方法
2	寄存器寻址	R0~R7、A、B、Cy、DPTR	内部 RAM 128B
3	寄存器间接寻址	@R0、@R1、SP	内部 RAM 128B
		@R0、@R1、@DPTR	数据存储器（片外 RAM）64KB
4	立即数寻址	#data、#data16	程序存储器（片内片外）64KB
5	相对寻址	PC+rel（偏移量）	程序存储器（片内片外）64KB
6	变址寻址	@A+DPTR、@A+PC	程序存储器（片内片外）64KB
7	位寻址	bit	内部 RAM（20H-2FH）、SFR

2.3 C51 单片机的指令说明

C51 单片机的指令按功能分为五大类，即数据传送、算术运算、逻辑运算、控制转移和位操作类指令。

2.3.1 数据传送类指令

数据传送类指令就相当于 C 语言中的赋值语句，它的作用就是将某一个操作数复制到其他存储单元中去，其一般格式为：

MOV 目的操作数 源操作数

注意：源操作数在传送前后保持不变。

数据传送类指令如图 2-3 所示。

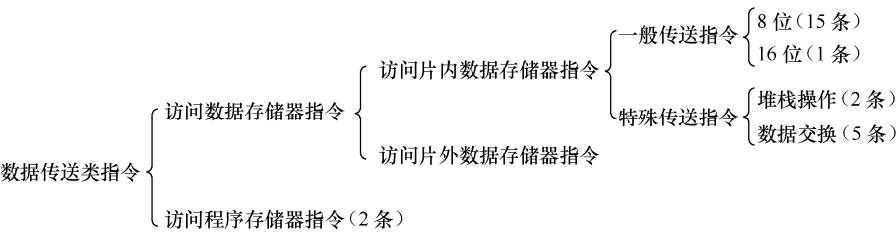


图 2-3 数据传送类指令结构示意图

数据传送类指令一般格式中，目的操作数不能采用立即寻址的方式；目的单元为 A 时影响 P 标志。根据目的操作数和源操作数的不同可以将数据传送指令分为 6 类。

1. 内部 RAM 数据传送指令

内部 RAM 数据传送指令示意图如图 2-4 所示。

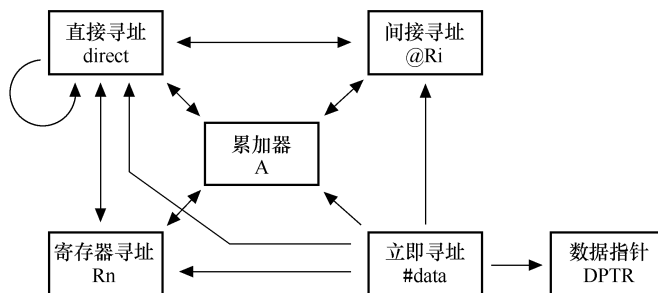


图 2-4 内部 RAM 数据传送指令示意图

(1) 立即数传送指令

立即数传送指令共有 4 条：

```
MOV A, #data ;A←#data
MOV direct, #data ;direct←#data
MOV Rn, #data ;Rn←#data
MOV @Ri, #data ;(Ri)←#data
```

除此以外还有一个很特殊的寄存器寻址指令：

```
MOV DPTR, #data ;DPTR←#data16
```

这 5 条指令均是将立即数传送内部 RAM 中去。

(2) 内部 RAM 的存储单元中的数据传送指令，共 5 条指令：

```
MOV direct2, direct1 ;direct2←(direct1)
MOV direct, Rn ;direct←(Rn)
MOV Rn, direct ;Rn←(direct)
MOV direct, @Ri ;direct←((Ri))
MOV @Ri, direct ;(Ri)←(direct)
```

(3) 通过累加器的数据传送指令，共 6 条指令：

```
MOV A, direct ;A←(direct)
MOV direct, A ;direct←(A)
MOV Rn, A ;Rn←(A)
MOV A, Rn ;A←(Rn)
MOV A, @Ri ;A←((Ri))
MOV @Ri, A ;(Ri)←(A)
```

设 (30H)=40H, (40H)=10H, (P1)=0CAH, 试判断下列程序执行后的结果。

```
MOV R0, #30H ;R0←#30H
MOV A, @R0 ;A←(30H), (A)=40H
MOV R1, A ;R1←(A), (R1)=40H
MOV B, @R1 ;B←(40H), (B)=10H
```

```
MOV    @R1    P1          ; (R1) ← (P1), (40H) = 0CAH
MOV    P2,     P1          ; P2 ← (P1), (P2) = 0CAH
```

结果是: (A) = 40H, (B) = 10H, (40H) = 0CAH, (P2) = 0CAH。

2. 外部 RAM 数据传送指令

外部 RAM 中数据传送均是通过间接寻址的方式来实现的, 共有 4 条指令。

(1) 读指令

```
MOVX   A,      @DPTR      ; A ← ((DPTR))
MOVX   A,      @Ri        ; A ← ((Ri))
```

(2) 写指令

```
MOVX   @DPTR,   A          ; (DPTR) ← (A)
MOVX   @Ri,     A          ; (Ri) ← (A)
```

其中需要注意以下几点:

(1) 这里的地址寄存器只能使用 DPTR 和 Ri, 并且当使用 Ri 时只能访问外部 RAM 的 256 字节;

(2) 与外部 RAM 传送数据只能通过累加器 A 来实现;

(3) 与外部 RAM 传送数据时使用 MOVX 指令。

【例 2-1】 将片外 RAM 1000H 单元中的内容送到 2000H 单元。

方法一:

```
MOV    DPTR     #1000H     ; DPTR ← #1000H
MOVX   A,       @DPTR      ; A ← ((DPTR))
MOV    DPTR,    #2000H     ; DPTR ← #2000H
MOVX   @DPTR,   A          ; (DPTR) ← (A)
```

方法二:

```
MOV    P2,      #10H       ; P2 ← #10H
MOV    R0,      #00H       ; R0 ← #00H
MOVX   A,       @R0        ; A ← ((R0))
MOV    P2,      #20H       ; P2 ← #20H
MOVX   @R0,     A          ; (R0) ← (A)
```

注意: 在两个片外 RAM 单元之间是不能直接进行数据传送的, 必须经过片内的累加器 A 来间接地传送。

3. 程序存储器传送指令 (查表指令)

表格: 程序存储器除存放程序外, 还可存放一些常数, 这种数据的结构称为表格。

访问: 通过两条程序存储器取数指令, 即查表指令来访问, 完成从 ROM 中读数, 并只能送累加器 A。

程序存储器数据传送指令必须使用 MOVC, 并且只能通过累加器 A 来实现, 共两条指令:

```
MOVC   A,       @A+PC      ; PC ← (PC) + 1, A ← ((A) + (PC))
MOVC   A,       @A+DPTR    ; A ← ((A) + (DPTR))
```

其功能是到程序存储器中查表格数据送入累加器 A。

前一条指令将 A 中的内容与 PC 的内容相加得到 16 位表格地址；后一条指令是将 A 中的内容与 DPTR 中的内容相加得到 16 位表格地址。由于程序存储器只能读不能写，因此程序存储器的数据传送都是单向的。

特点：单字节指令，源操作数为变址寻址，执行时，PSEN 会自动有效。

功能：以 DPTR、PC 为基地址，与 A 中的 8 位无符号数相加，得到一个新 16 位地址，将其内容送 A。

(1) 用 DPTR 作为基寄存器

【例 2-2】 设外部 ROM 的 2000H 单元开始的连续 10 个字节中已存放有 0~9 的平方数，要求根据 A 中的内容 (0~9) 来查找对应的平方值。

```
START: MOV     A,      #3
        MOV     DPTR,   #TABLE
        MOVC    A,      @A+DPTR    ;查表
        ...
```

ORG 2000H

结果：A ← (2003H)，(A) = 09H。

特点：可访问整个 ROM 的 64KB 空间，表格可放在 ROM 的任何位置，与 MOVC 指令无必然的关系。

(2) 用 PC 作为基寄存器

【例 2-3】 ORG 1000H

```
1000H    MOV     A,    #30H
1002H    MOVC    A,    @A+PC
```

结果是：A ← (1033H)。

优点：不改变 PC 的状态，根据 A 的内容取表格常数。

缺点：

- ① 表格只能存放在查表指令以下的 256 个单元内；
- ② 当表格首地址与本指令间有其他指令时，须用调整偏移量，调整量为下一条指令的起始地址到表格首址之间的字节数。

【例 2-4】 阅读下列程序，给出运行结果，设 (A) = 3。

```
1000H  ADD     A,    #02H          ; 加调整量
1002H  MOVC    A,    @A+PC         ; 查表
1003H  NOP
1004H  NOP
1005H  TAB: DB  66, 77, 88H, 99H, 'W', '10'
```

结果：(A) = 99H，显然，2 条 NOP 指令没有时，不需调整。

4. 数据交换指令

数据交换主要用在累加器和其他内部 RAM 中的数据交换，数据交换指令分为三类：

(1) 整字节交换，使用指令 XCH，例如：

XCH A, Rn ; (A) $\leftrightarrow$ (Rn)

它表示将 A 内的数据和 Rn 中的数据交换，其中指令中目的操作数只能使用累加器 A，源操作数可以是直接寻址、寄存器寻址、寄存器间接寻址。

(2) 半字节交换，使用指令 **XCHD**，例如：

XCHD A, @Ri ; (A) $\leftrightarrow$ (Ri)

注意半字节交换指令使用这一种方式，它实现的功能是将累加器 A 中的低四位和 (Ri) 中的低四位进行交换。

(3) 累加器高低半字节交换指令

SWAP A

其作用就是将 A 中的高低四位互换。

【例 2-5】 设 (A) = 57H, (20H) = 68H, (R0) = 30H, (30H) = 39H, 求下列指令的执行结果

(1) **XCH A, 20H**

结果: (A) = 68H, (20H) = 57H。

(2) **XCH A, @R0**

结果: (A) = 39H, (30H) = 57H。

(3) **XCH A, R0**

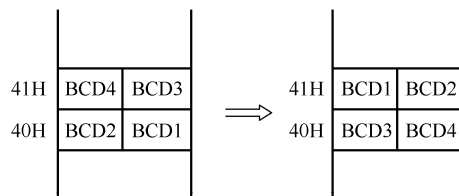
结果: (A) = 30H, (R0) = 57H。

(4) **XCHD A, @R0**

结果: (A) = 59H, (30H) = 37H)。

(5) **SWAP A**

结果: (A) = 75H。



【例 2-6】 设内部 RAM 40H、41H 单元中连续存放有 4 个压缩的 BCD 码数据，试编程序将这 4 个 BCD 码倒序排列。

分析: 流程如图 2-5 所示。

程序如下:

```
MOV    A,    41H
SWAP   A
XCH    A,    40H
SWAP   A
MOV    41H,  A
```

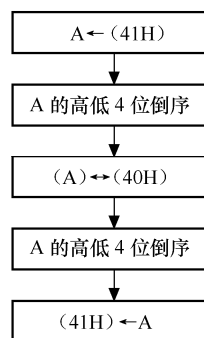


图 2-5 BCD 码倒序排列

5. 堆栈操作指令

堆栈由特殊功能寄存器 SP (81H) 管理，始终指向其栈顶位置，栈底视需要设在内部 RAM 低 128byte 内。

(1) 进栈操作: **PUSH direct**

功能: 先 (SP) $\leftarrow$ (SP) + 1, 再 ((SP)) $\leftarrow$ (direct);

其中: direct 为源操作数; 目的操作数为 @SP, 隐含。

【例 2-7】 已知：(A) = 30H, (B) = 70H

执行：

```
MOV     SP,    #60H    ;设栈底
PUSH    ACC
PUSH    B
```

结果：(61H) = 30H, (62H) = 70H, (SP) = 62H。

(2) 出栈操作： POP direct

功能：先 (direct) $\leftarrow$ ((SP)), 再 SP $\leftarrow$ (SP) - 1

其中：direct 为目的操作数，源操作数为 @SP，隐含。

【例 2-8】 已知：(SP) = 62H, (62H) = 70H, (61H) = 30H

执行：

```
POP     DPH
POP     DPL
```

结果：(DPTR) = 7030H, (SP) = 60H。

2.3.2 算术运算类指令

包括 +、-、 $\times$ 、 $\div$ 、加 1、减 1、十进制调整指令，C51 单片机算术运算类指令共有 24 条，一般影响 PSW 中的 CY, AC, OV, P 标志位。

1. 加法指令

```
ADD  A,    Rn    ;A  $\leftarrow$  (A) + (Rn)
ADD  A,    direct ;A  $\leftarrow$  (A) + (direct)
ADD  A,    @Ri   ;A  $\leftarrow$  (A) + ((Ri))
ADD  A,    #data ;A  $\leftarrow$  (A) + data
ADDC A,    Rn    ;A  $\leftarrow$  (A) + (Rn) + (CY)
ADDC A,    direct ;A  $\leftarrow$  (A) + (direct) + (CY)
ADDC A,    @Ri   ;A  $\leftarrow$  (A) + ((Ri)) + (CY)
ADDC A,    #data ;A  $\leftarrow$  (A) + data + (CY)
```

如果把参加运算的两个操作数看作是无符号数 (0~255)，加法运算对 CY 标志位的影响如下：

(1) 若结果的第 7 位向前有进位 (C7 = 1)，CY = 1；

(2) 若结果的第 7 位向前无进位 (C7 = 0)，CY = 0。

【例 2-9】 设有两个无符号数放在 A 和 R2 中，设 (A) = 0C6H (198)，(R2) = 68H (104)。

执行指令：ADD A, R2，试分析运算结果及对标志位的影响。

分析：写成竖式

(A)	11000110	198
(R2)	+01101000	+104
(A)	1 00101110	302

结果是: (A) = 2EH, CY=1。

2. 加 1 指令

```

INC    A           ;A ← (A) + 1
INC    Rn          ;Rn ← (Rn) + 1
INC    direct       ;direct ← (direct) + 1
INC    @Ri          ; (Ri) ← ((Ri)) + 1
INC    DPTR         ;DPTR ← (DPTR) + 1

```

这组指令的功能是使源操作数的值加 1。

【例 2-10】 设有两个 16 位无符号数，被加数存放在内部 RAM 的 30H（低位字节）和 31H（高位字节）中，加数存放在 40H（低位字节）和 43H（高位字节）中。试写出求两数之和，并把结果存放在 30H 和 31H 单元中的程序。

参考程序为：

```

MOV    R0,    #30H    ; 地址指针 R0 赋值
MOV    R1,    #40H    ; 地址指针 R1 赋值
MOV    A,     @R0     ; 被加数的低 8 位送 A
ADD    A,     @R1     ; 被加数与加数的低 8 位相加，和送 A，并影响 CY 标志
MOV    @R0,   A       ; 和的低 8 位存 30H 单元
INC    R0      ; 修改地址指针 R0
INC    R1      ; 修改地址指针 R1
MOV    A,     @R0     ; 被加数的高 8 位送 A
ADDC   A,     @R1     ; 被加数和加数的高 8 位与 CY 相加，和送 A
MOV    @R0,   A       ; 和的高 8 位存 31H 单元

```

3. 减法指令

(1) 带借位减法指令

```

SUBB   A,     Rn       ;A ← (A) - (Rn) - (CY)
SUBB   A,     direct   ;A ← (A) - (direct) - (CY)
SUBB   A,     @Ri      ;A ← (A) - ((Ri)) - (CY)
SUBB   A,     #data    ;A ← (A) - data - (CY)

```

该组指令的功能是从累加器 A 减去源操作数及标志位 CY，其结果再送累加器 A。CY 位在减法运算中是作借位标志。

SUBB 指令对标志位的影响如下：

- ① 若第 7 位向前有借位 (C7=1)，则 CY=1；
- ② 若第 7 位向前无借位 (C7=0)，则 CY=0，

【例 2-11】 在 A 和 R3 中存有两个无符号数，设 (A) = 98H, (R3) = 6AH, CY=1，执行指令：SUBB A, R3，分析执行结果及对标志位的影响。

分析:	(A)	10011000	98H
	(R3)	-01101010	6AH
	CY	-1	-1
	(A)	00101101	2DH

结果是: (A) = 2DH, CY=0。

(2) 减 1 指令

DEC A ; $A \leftarrow (A) - 1$
 DEC Rn ; $Rn \leftarrow (Rn) - 1$
 DEC direct ; $direct \leftarrow (direct) - 1$
 DEC @Ri ; $(Ri) \leftarrow ((Ri)) - 1$

这组指令的功能是使源操作数的值减 1。

加减法指令形式结构图如图 2-6 所示, 通过该图可以很清楚地掌握加法和减法指令的结构。

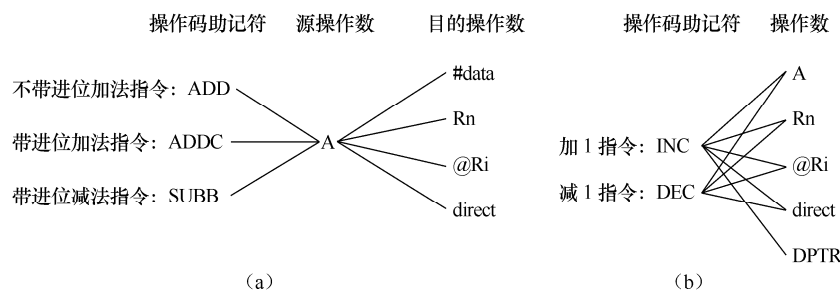


图 2-6 加减法指令形式结构图

4. 十进制调整指令

DA A

该指令用于两个 BCD 码加法运算的加 6 修正, 只影响 CY 位。指令的使用条件如下:

- (1) 只能紧跟在加法指令 (ADD/ADDC) 后进行;
- (2) 两个加数必须已经是 BCD 码;
- (3) 只能对累加器 A 中结果进行调整, 且加 6 修正的依据: 由 CPU 判 CY、AC 是否为 1, A 的高、低 4 位是否大于 9。

【例 2-12】试编制十进制数加法程序 (单字节 BCD 加法), 假定和数为单字节, 要求: (20H) + (21H) → 22H。

分析: 流程如图 2-7 所示。

程序如下:

```
MOV R0, #20H
MOV A, @R0
INC R0
ADD A, @R0
DA A
INC R0
```

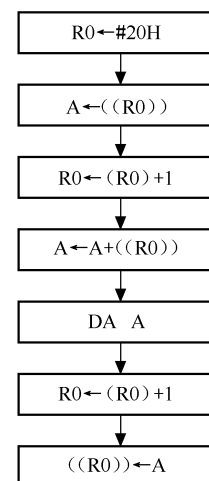


图 2-7 例 2-12 流程图

```
MOV  @R0,  A
```

【例 2-13】十进制减法程序（单字节 BCD 数减法）要求：

$(20H) - (21H) \rightarrow 22H$ 。

分析：考虑到 DA A 只能对加法调整，故必须先化 BCD 减法为加法运算，关键为求两位十进制减数的补码（9AH=减数）。

流程如图 2-8 所示，程序如下：

```
CLR    C
MOV    R0,    #20H
MOV    R1,    #21H
MOV    A,     #9AH
SUBB   A,     @R1    ;求补
ADD    A,     @R0    ;求差
DA     A
INC    R1
MOV    @R1,   A      ;存结果
```

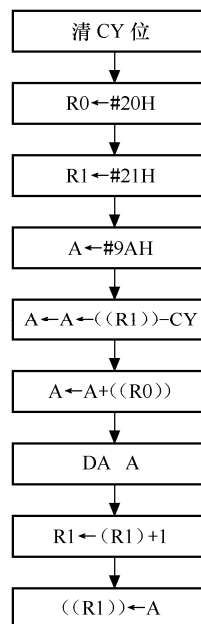


图 2-8 例 2-13 流程图

5. 乘法指令

```
MUL AB ; BA ← (A) × (B)
```

指令的功能是把累加器 A 和寄存器 B 中两个 8 位无符号整数相乘，并把乘积的高 8 位存于寄存器 B 中，低 8 位存于累加器 A 中。

执行乘法运算指令时会对标志位产生影响：CY 标志总是被清 0，即 CY=0；OV 标志则反映乘积的位数，若 OV=1，表示乘积为 16 位数，若 OV=0，表示乘积为 8 位数。

例如：(A) = 50H, (B) = A0H, 执行 MUL AB

结果：(B) = 32H, (A) = 00H, (OV) = 1。

6. 除法指令

```
DIV AB ; A 商, B 余 ← (A) ÷ (B)
```

该指令的功能是把累加器 A 和寄存器 B 中的两个 8 位无符号整数相除，所得商的整数部分存于累加器 A 中，余数存于 B 中。

除法指令执行过程对标志位的影响：CY 位总是被清 0，OV 标志位的状态反映寄存器 B 中的除数情况，若除数为 0，则 OV=1，表示本次运算无意义，否则，OV=0。

例如：(A) = 2AH, (B) = 05H, 执行 DIV AB

结果：(A) = 08H, (B) = 02H, (OV) = 0。

【例 2-14】编写将 A 中的数转换成三位 BCD 码程序，百位放在 20H，十位、个位放在 21H 中。

分析：(A) / 100 → 商（百位）→ (20H)；

余数 / 10 → 商（十位）→ (21H)；

余数（个位）→ (21H)；

流程如图 2-9 所示，程序如下：


```

MOV    B,      #100
DIV     AB
MOV    20H,    A
MOV    A,      B
MOV    B,      #10
DIV     AB
SWAP   A
ADD    A,      B
MOV    21H,    A
SJMP   $

```

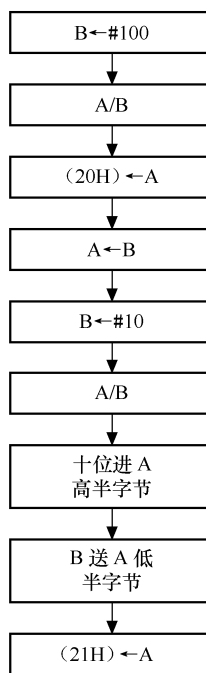


图 2-9 例 2-14 流程图

2.3.3 逻辑运算类指令

在 C51 单片机指令系统中，逻辑运算类指令有 25 条，可实现与、或、异或等逻辑运算操作。逻辑指令形式结构如图 2-10 所示。这类指令有可能会影响 CY 和 P 标志位的状态。

1. 累加器 A 的逻辑操作指令

(1) 累加器 A 清 0

```
CLR  A ; A ← 00H
```

(2) 累加器 A 取反

```
CPL  A ; A ← (A)
```

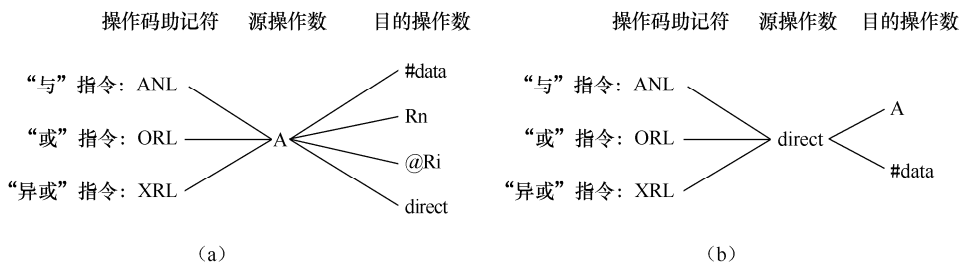


图 2-10 逻辑指令形式结构图

例如：设 (A) = 33H，执行指令 CPL A
结果：(A) = 0CCH。

(3) 累加器 A 循环左移
RL A

利用左移指令，可实现对 A 中的无符号数乘 2 的目的，其指令操作示意图如图 2-11 所示。

【例 2-15】 执行下列指令后，A 中的内容如何变化？

```
MOV A, #11H ; (A) = 11H (17)
RL A ; (A) = 22H (34)
RL A ; (A) = 44H (68)
RL A ; (A) = 88H (136)
RL A ; (A) = 11H (17)
```

(4) 累加器 A 带进位循环左移

累加器 A 带进位循环左移的操作示意图如图 2-12 所示。

(5) 累加器 A 循环右移

对累加器 A 进行循环右移，可实现对 A 中无符号数的除 2 运算，其指令操作示意图如图 2-13 所示。

(6) 累加器 A 带进位循环右移

累加器 A 带进位循环右移的操作示意图如图 2-14 所示。

(7) 累加器 A 半字节交换

该指令的功能是将累加器 A 中内容的高 4 位与低 4 位互换，其指令操作示意图如图 2-15 所示。

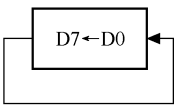


图 2-11 累加器 A 循环左移

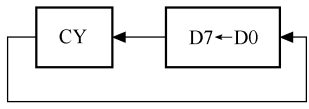


图 2-12 累加器 A 带进位循环左移

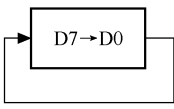


图 2-13 累加器 A 循环右移

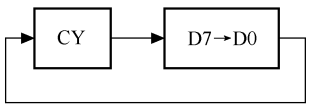


图 2-14 累加器 A 带进位循环右移

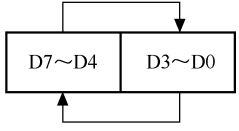


图 2-15 累加器 A 半字节交换

例如：设 (A) = 63H，执行指令
SWAP A
结果：(A) = 36H。

2. 逻辑与指令

```
ANL A, Rn ; A ← (A) ∧ (Rn)
```

```

ANL  A,      direc    ;A ← (A) ∧ (direct)
ANL  A,      @Ri      ;A ← (A) ∧ ((Ri))
ANL  A,      #data    ;A ← (A) ∧ #data
ANL  direct, A        ;direct ← (direct) ∧ (A)
ANL  direct, #data    ;direct ← (direct) ∧ #data

```

该组指令的功能是将目的操作数和源操作数按位进行逻辑与操作，结果送目的操作数。

在程序设计中，逻辑与指令主要用于对目的操作数中的某些位进行屏蔽（清0）。方法是将需屏蔽的位与“0”相与，其余位与“1”相与。

3. 逻辑或指令

```

ORL  A,      Rn      ;A ← (A) ∨ (Rn)
ORL  A,      direct  ;A ← (A) ∨ (direct)
ORL  A,      @Ri     ;A ← (A) ∨ ((Ri))
ORL  A,      #data   ;A ← (A) ∨ #data
ORL  direct, A       ;direct ← (direct) ∨ (A)
ORL  direct, #data   ;direct ← (direct) ∨ #data

```

该组指令的功能是将目的操作数和源操作数按位进行逻辑或操作，结果送目的操作数。

逻辑或指令可对目的操作数的某些位进行置位。方法是将需置位的位与“1”相或，其余位与“0”相或，常用于组合数据。

4. 逻辑异或指令

```

XRL  A,      Rn      ;A ← (A) ⊕ (Rn)
XRL  A,      direct  ;A ← (A) ⊕ (direct)
XRL  A,      @Ri     ;A ← (A) ⊕ ((Ri))
XRL  A,      #data   ;A ← (A) ⊕ #data
XRL  direct, A       ;direct ← (direct) ⊕ (A)
XRL  direct, #data   ;direct ← (direct) ⊕ #data

```

该组指令的功能是将目的操作数和源操作数按位进行逻辑异或操作，结果送目的操作数。

逻辑异或指令可用于对目的操作数的某些位取反，而其余位不变。方法是将要取反的这些位和“1”异或，其余位则和“0”异或。

【例 2-16】 将工作寄存器 R2 中数据的高 4 位和 R3 中的低 4 位拼成一个数，并将该数存入 30H。

```

MOV  R0,     #30H    ;R0 作地址指针
MOV  A,      R2
ANL  A,      #0F0H   ;屏蔽低 4 位
MOV  B,      A       ;中间结果存 B 寄存器
MOV  A,      R3
ANL  A,      #0FH    ;屏蔽高 4 位
ORL  A,      B       ;组合数据

```

MOV @R0, A ;结果存 30H 单元。

【例 2-17】 分析下列程序的执行结果。

```
MOV A,      #77H    ; (A) = 77H
XRL A,      #0FFH   ; (A) = 77H ⊕ FFH = 88H
ANL A,      #0FH    ; (A) = 88H ∧ 0FH = 08H
MOV P1,     #64H    ; (P1) = 64H
ANL P1,     #0F0H   ; (P1) = 64H ∧ F0H = 60H
ORL A,      P1      ; (A) = 08H ∨ 60H = 68H
```

2.3.4 控制转移类指令

程序的顺序执行是由 PC 自动加 1 实现的。要改变程序的执行顺序，实现分支转向，应通过强迫改变 PC 值的方法来实现，这就是控制转移类指令的基本功能。C51 单片机中共有两类转移，即无条件转移和有条件转移。此外，本节还将介绍子程序调用及返回指令。

1. 无条件转移指令组

不规定条件的程序转移称为无条件转移。C51 单片机中共有 4 条无条件转移指令。

(1) 长转移指令

LJMP addr 16 ; PC ← addr16

该指令执行后把 16 位地址 (addr16) 送 PC，从而实现程序转移。该指令的转移范围大，可达 64KB，因此称为“长转移”。长转移指令是三字节指令，依次是操作码、高 8 位地址、低 8 位地址。

(2) 绝对转移指令

AJMP addr 11

该指令是一条二字节指令，其指令格式如图 2-16 所示。

A10	A9	A8	0	0	1	0	0
A7	A6	A5	A4	A3	A2	A1	A0

图 2-16 AJMP 指令格式

指令提供的 11 位地址中，A7~A0 在第二字节。A10~A8 则占据第一字节的高 3 位，而指令操作码只占第一字节的低 5 位。AJMP 指令的功能是构造程序转移的目的地址，实现程序转移，其构造方法是：以指令提供的 11 位地址去替换 PC 的低 11 位内容，形成新的 PC 值，即转移的目的地址。但要注意，被替换的 PC 值是本条指令地址加 2 以后的 PC 值，即指向下一条指令的 PC 值。因此 AJMP 指令的操作过程可表示为：

$PC \leftarrow (PC) + 2$

$PC_{10 \sim 0} \leftarrow \text{addr } 11$

例如，程序中 2070H 地址单元有绝对转移指令：

2070H AJMP 16AH

11 位绝对转移地址为：00101101010B (16AH)。因此，指令代码如图 2-17 所示。

0	0	1	0	0	1	0	0
0	1	1	0	1	0	1	0

图 2-17 绝对转移地址为 16AH 的指令代码

程序计数器 PC 加 2 后的内容为：0010000001110010B（2072H），以 11 位绝对转移地址替换 PC 的低 11 位内容，最后形成的目的地址为 0010000101101010B（216AH）。其中 B 表示二进制数，H 表示十六进制数。addr11 是地址，因此是无符号数，其最小值为 000H，最大值为 7FFH，因此绝对转移指令所能转移的最大范围是 2KB。对于“2070H AJMP 16AH”指令，其转移范围是 2000H~27FFH。

例如指令：

LOOP: AJMP addr 11

假设 addr11 = 00100000000B，标号 LOOP 的地址为 1030H，则执行指令后，程序将转移到 1100H。该指令的机器码为 21H，00H（A10A9A8 为 001），即指令的第一字节为 21H。

（3）短转移指令

SJMP Rel

SJMP 为相对寻址方式转移指令，其中 rel 为相对偏移量，其功能是计算目的地址，并按计算得到的目的地址实现程序的相对转移。计算公式为：

目的地址 = (PC) + 2 + rel

偏移量 rel 是一个带符号的 8 位二进制补码数，因此能实现程序的双向转移。rel 为正数则向前转移；rel 为负数则向后转移。转移的范围是 256byte，因此称为短转移。对于短转移指令的使用可从以下两个方面进行讨论。

① 根据偏移量 rel 计算转移的目的地址

这种情况经常在读目标程序时遇到，是解决往哪儿转移的问题。例如，在 835AH 地址上有 SJMP 指令：

835AH SJMP 35H

源地址为 835AH，rel = 35H 是正数，因此程序向前转移。目的地址 = 835AH + 35H = 8931H，即执行完本指令后，程序转到 8931H 地址去执行。

又例如，在 835AH 地址上 SJMP 指令是：

835AH SJMP 0E7H

rel = 0E7H，是负数 19H 的补码，因此程序向后转移。目的地址 = 835AH + 02H - 19H = 8343H，即执行完本指令后，程序向后转到 8343H 地址去执行。

② 根据目的地址计算偏移量

这是编程时必须解决的问题，也是一项比较麻烦的问题。对于二字节 SJMP 指令，rel 的计算公式如下：

向前转移时，则

rel = 目的地址 - 源地址 - 2 = (地址差) - 2

向后转移时，目的地址小于源地址，rel 应为负数的补码，则

rel = (目的地址 - (源地址 + 2)) 补
= FFH - (源地址 + 2 - 目的地址) + 1

= FEH - (地址差)

为方便起见，在汇编程序中都有计算偏移量的功能。编写汇编源程序时，只需在相对转移指令中直接写上要转向的地址标号就可以了。程序汇编时由汇编程序自动计算和填入偏移量。但手工汇编时，偏移量的值则需程序自己计算。

此外，在汇编语言程序中，为等待中断或程序结束，常有使程序“原地踏步”的需要，对此可使用 SJMP 指令完成。

```
HERE: SJMP HERE
```

或 HERE: SJMP \$

指令机器码为 80FE。在汇编语言中，以“\$”代表 PC 的当前值。

【例 2-18】 执行指令：

```
LOOP: SJMP LOOPI
```

如果 LOOP 的标号值为 0100H (即 SJMP 指令的机器码存于 0100H 和 0101H 两个单元之中)。标号 LOOPI 的值为 0123H，即跳转的目标地址为 0123H，则指令的第二个字节 (相对偏移量) 应为：

rel = 0123H - 0102H = 21H

(4) 变址寻址转移指令

```
JMP @A+DPTR ; PC ← (A) + DPTR
```

该指令是一条一字节转移指令。转移的目的地址由 A 的内容和 DPTR 内容之和来确定，即目的地址 = (A) + (DPTR)。本指令以 DPTR 内容为基址，而以 A 的内容作变址，因此只要把 DPTR 的值固定，而给 A 赋以不同的值，就可实现程序的多分支转移。本指令的一个典型应用是键盘译码程序。

2. 条件转移指令组

所谓条件转移就是程序转移是有条件的。执行条件转移指令时，如指令中规定的条件满足，则进行程序转移；否则程序顺序执行。这种转移相当于高级语言中的 if...go to 语句。条件转移有以下指令。

(1) 累加器判零转移指令

```
JZ rel ; 若 (A) = 0, 则 PC ← (PC) + 2 + rel
        ; 若 (A) ≠ 0, 则 PC ← (PC) + 2
JNZ rel ; 若 (A) ≠ 0, 则 PC ← (PC) + 2 + rel
        ; 若 (A) = 0, 则 PC ← (PC) + 2
```

这两条指令都是二字节指令，是有条件的相对转移指令，且以 rel 为偏移量。

【例 2-19】 将外部数据 RAM 的一个数据块传送到内部数据 RAM 中，两者的首地址分别为 DATA1 和 DATA2，遇到传送的数据为 0 时停止。

分析：外部 RAM 向内部 RAM 的数据传送一定要借助于累加器 A，利用累加器判零转移指令正好可以判别是否要继续传送或者终止传送。

```
MOV R0, #DATA1H ;外部数据块首地址
MOV R1, #DATA2H ;内部数据块首地址
LOOP: MOVX A, @R0 ;外部数据送给 A
```

```

HERE:  JZ      HERE           ;为 0 则终止
        MOV    @R1,    A      ;不为 0 传送内部 RAM 数据
        INC    R0            ;修改地址指针
        INC    R1
        SJMP   LOOP          ;继续循环

```

(2) 数值比较转移指令

数值比较转移指令是把两个操作数进行比较，以比较结果作为条件来控制程序转移，共有 4 条指令。

```

CJNE    A,      # data , rel  ;累加器内容与立即数不等则转移
CJNE    A,      direct , rel  ;累加器内容与内部 RAM 单元内容不等则转移
CJNE    Rn,     # data , rel  ;寄存器内容与立即数不等则转移
CJNE    @Ri,    # data , rel  ;内部 RAM 低 128 单元内容与立即数不等则转移

```

数值比较转移指令是三字节指令，这是 C51 指令系统中仅有的 4 条 3 个操作数的指令，在程序设计中非常有用。这 4 条指令的功能可从程序转移和数值比较两个方面来说明。

① 程序转移

为简单起见，把指令中的两个比较数据分别称为左操作数和右操作数，则指令的转移可按以下 3 种情况说明。

若左操作数 = 右操作数，则

```

程序顺序执行      PC ← (PC) + 3
进位标志位清 “0”  CY ← 0

```

若左操作数 > 右操作数，则

```

程序转移          PC ← (PC) + 3 + rel
进位标志位清 “0”  CY ← 0

```

若左操作数 < 右操作数，则

```

程序转移          PC ← (PC) + 3 + rel
进位标志位置 “1”  CY ← 1

```

② 数值比较

在 C51 中没有专门的数值比较指令，两个数的数值比较可利用这 4 条指令来实现，数值比较可在程序转移的基础上进行，即

```

程序顺序执行，      则      左操作数 = 右操作数
程序转移且 (CY) = 0, 则      左操作数 > 右操作数
程序转移且 (CY) = 1, 则      左操作数 < 右操作数

```

(3) 减 1 条件转移指令

该组指令是一组把减 1 与条件转移两种功能结合在一起的指令，共有 2 条指令。

① 寄存器减 1 条件转移指令

```
DJNZ    Rn,      rel
```

该指令是一条两字节指令，其功能为寄存器内容减 1，如所得结果为 0，则程序顺序执行；如没有减到 0，则程序转移。具体表示如下：

```
Rn ← (Rn) - 1      ; 若 (Rn) ≠ 0, 则 PC ← (PC) + 2 + rel
```

② 直接寻址单元减 1 条件转移指令

该指令是一条三字节指令，其功能为直接寻址单元内容减 1，如所得结果为 0，则程序顺序执行；如没有减到 0，则程序转移。具体表示如下：

上面 2 条指令主要用于控制程序循环。例如，预先把寄存器或内部 RAM 单元赋值循环次数，则利用减 1 条件转移指令，以减 1 后是否为 0 作为转移条件，即可实现按次数控制循环。

	MOV	DPTR,	# 2000H	;源数据区首地址
	PUSH	DPL		;源首址暂存堆栈
	PUSH	DPH		
	MOV	DPTR,	# 3000H	;目的数据区首址
	MOV	R2,	DPL	;目的首址暂存寄存器
	MOV	R3,	DPH	
LOOP:	POP	DPH		;取回源地址
	POP	DPL		
	MOVB	A,	@DPTR	;取出数据
	INC	DPTR		;源地址增量
	PUSH	DPL		;源地址暂存堆栈
	PUSH	DPH		
	MOV	DPL,	R2	;取回目的地址
	MOV	DPH,	R3	
	MOVB	@DPTR,	A	;数据送目的区
	INC	DPTR		;目的地址增量
	MOV	R2,	DPL	;目的地址暂存寄存器
	MOV	R3,	DPH	
	DJNZ	35H,	LOOP	;没完, 继续循环
	RET			;返回主程序

可以看出, 无条件转移指令有直接寻址和相对寻址两种方式, 而条件转移指令则只有相对寻址一种寻址方式。

子程序结构是一种重要的程序结构。在一个程序中经常遇到反复多次执行某程序段的情况，如果重复书写这个程序段，会使程序变得冗长而杂乱。对此，可采用子程序结构，即把重复的程序段编写为一个子程序，通过主程序调用而使用它，这样不但减少了编程工作量，

而且也缩短了程序的长度。

调用和返回构成了子程序调用的完整过程,如图 2-18 所示。为了实现这一过程,必须有子程序调用指令和返回指令。调用指令在主程序中使用,而返回指令则应该是子程序的最后一条指令。执行完这条指令之后,程序返回主程序断点处继续执行。

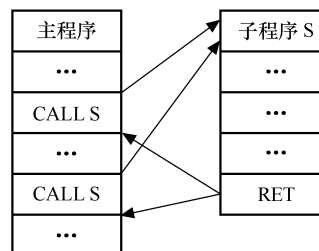


图 2-18 子程序的调用过程

(1) 绝对调用指令

ACALL addr 11

该指令是一条二字节指令,其指令格式如图 2-19 所示。

A10	A9	A8	1	0	1	0	0
A7	A6	A5	A4	A3	A2	A1	A0

图 2-19 ACALL 指令格式

在指令中提供了子程序入口地址的低 11 位,这 11 位地址的 A7~A0 在指令的第二字节中, A10~A8 则占据第一字节的高 3 位。

为了实现子程序调用,该指令共完成两项操作。

① 断点保护

断点保护是通过自动方式的堆栈操作实现的,即把加 2 以后的 PC 值自动送堆栈保存起来,待子程序返回时再送回 PC。

② 构造目的地址

目的地址的构造是在 PC 加 2 的基础上,以指令提供的 11 位地址取代 PC 的低 11 位,而 PC 的高 5 位不变。例如,程序中有绝对调用指令:

8100H ACALL 48FH

Addr11 的高三位 A10~A8 = 100,因此指令第一字节为 91H,第二字节为 8FH,构造的目的地址 PC 加 2 后 (PC) = 8102H,即:

1000000100000010 B (8102H)

指令提供的 11 位地址是 10010001111B。替换 PC 的低 11 位后,PC 的值即形成的目的地址是:

1000010010001111B (848FH)

即被调用子程序入口地址为 848FH,或者说主程序到 848FH 处去调用子程序。

因为指令给出了子程序入口地址的低 11 位,因此本指令的子程序调用范围是 2KB。例如,本调用指令的地址为 8100H,不变的高 5 位是 10000byte,因此本指令的子程序调用范围是 8000H~87FFH。

综上所述,ACALL 指令的操作内容可表示为:

$PC \leftarrow (PC) + 2$

$SP \leftarrow (SP) + 1, (SP) \leftarrow (PC) 7 \sim 0$

$SP \leftarrow (SP) + 1, (SP) \leftarrow (PC) 15 \sim 8$

$PC_{10 \sim 0} \leftarrow \text{addr}_{11}$

(2) 长调用指令

LCALL addr 16

该指令是三字节指令,调用地址在指令中直接给出。指令执行后,断点进栈保存,以 addr16 作地址调用子程序。因此本指令的操作内容可表示为:

$PC \leftarrow (PC) + 3$

$SP \leftarrow (SP) + 1, (SP) \leftarrow (PC) 7 \sim 0$

$SP \leftarrow (SP) + 1, (SP) \leftarrow (PC) 15 \sim 8$

$PC_{10 \sim 0} \leftarrow \text{addr16}$

长调用指令的子程序调用范围是 64KB。Addr16 就是被调用子程序的入口地址,使用比较方便,但三字节指令较长,占用存储空间较多。

例如: $(SP) = 60H$, 标号地址 START 为 0100H, 标号 MIR 为 8100H, 执行指令:

START: LCALL MIR

结果: $(SP) = 62H$, $(61H) = 03H$, $(62H) = 01H$, $(PC) = 8100H$ 。

(3) 返回指令

返回指令共有两条:

RET ; 子程序返回指令

RETI ; 中断服务子程序返回指令

子程序返回指令执行子程序返回功能,从堆栈中自动取出断点地址送给程序计数器(PC),使程序在主程序断点处继续向下执行。因此本指令的操作内容可表示为:

$PC_{15 \sim 8} \leftarrow (SP) - 1, SP \leftarrow (SP) - 1$

$PC_{7 \sim 0} \leftarrow (SP) - 1, SP \leftarrow (SP) - 1$

例如: $(SP) = 62H$, $(62H) = 07H$, $(61H) = 30H$, 执行指令:

RET

结果: $(SP) = 60H$, $(PC) = 0730H$, CPU 从 0730H 开始执行程序。

中断服务子程序返回指令,除具有上述子程序返回指令所具有的全部功能之外,还有清除中断响应时被置位的优先级状态,开放较低级中断和恢复中断逻辑等功能。

4. 空操作指令

NOP ; $PC \leftarrow (PC) + 1$

空操作指令也算一条控制指令,即控制 CPU 不作任何操作,只消耗一个机器周期的时间。空操作指令是单字节指令,因此执行后 PC 加 1,时间延续一个机器周期。NOP 指令常用于程序的等待或时间的延迟。

2.3.5 位操作指令

所谓位处理是以位(bit)为单位进行的运算和操作。位变量也称为布尔变量或开关变量。位操作指令是位处理器的软件资源,它是 C51 单片机指令系统的一个子集,用以进行位的传送、置位、清“0”、取反、位状态判断、位逻辑运算、位输入与输出等位操作。

位处理器的硬件资源包括:

① 运算器中的 ALU,与字节处理合用。

② 程序存储器，与字节处理合用。

③ 位累加器 CY，它是位传送的中心。在字节处理中有一个累加器 A，而位处理需要使用进位标志 (CY) 作为累加器。为区分起见，暂且把 A 称为字节累加器，而把 CY 称为位累加器，在指令中写作 C。

④ 内部 RAM 位寻址区的 128 个可寻址位。

⑤ 专用寄存器中的可寻址位。

⑥ I/O 口的可寻址位。

1. 位传送指令组

位传送操作就是可寻址位与累加位 CY 之间的相互传送，共有 2 条指令：

```
MOV    C,      bit    ;CY ← (bit)
MOV    bit,     C      ;bit ← (CY)
```

例如：

```
MOV    C,      06H
MOV    P1.0,    C
```

由于没有两个可寻址位之间的传送指令，因此它们之间无法实现直接传送。如需要这种传送，应使用这 2 条指令并以 CY 作中介来实现。例如，将 20H 位的内容传送至 5AH 位，程序如下：

```
MOV    10H,     C      ;暂存 CY 内容
MOV    C,      20H     ;20H 位送 CY
MOV    5AH,     C      ;CY 送 5AH 位
MOV    C,      10H     ;恢复 CY 内容
```

2. 位复位指令组

该组指令是对 CY 及可寻址位进行置位或复位操作，共有 4 条指令：

```
SETB   C        ;CY ← 1
SETB   bit       ;bit ← 1
CLR    C        ;CY ← 0
CLR    bit       ;bit ← 0
```

例如：SETB P1.0

3. 位运算指令组

位运算都是逻辑运算，有与、或、非三种，共 6 条指令：

```
ANL    C, bit    ;CY ← (CY) ∨ (bit)
ANL    C, /bit   ;CY ← (CY) ∨ ( $\overline{\text{bit}}$ )
ORL    C, bit    ;CY ← (CY) ∧ (bit)
ORL    C, /bit   ;CY ← (CY) ∧ ( $\overline{\text{bit}}$ )
CPL    C        ;CY ← ( $\overline{\text{CY}}$ )
CPL    bit      ;bit ← ( $\overline{\text{bit}}$ )
```

例如：设 P1 为输入口，P3 为输出口，执行程序：

```
MOV    C,    P1.0
ANL    C,    P1.1
ANL    C,    /P1.2
MOV    P3.0, C
```

结果：P3.0 = P1.0 ∧ P1.1 ∧ /P1.2。

在位操作指令中，没有位的异或运算，如需要时可由上述多条位操作指令实现。例如，E、B、D 代表位地址，进行 E、B 内容的异或操作，结果送 D。可按公式：

$$D = E \oplus B = \overline{E}B + E\overline{B}$$

进行异或运算，实现的程序如下：

```
MOV    C,    B
ANL    C,    /E        ;CY ← B
MOV    C,    C
MOV    C,    E
ANL    C,    /B        ;CY ← E
ORL    C,    D        ;B+
MOV    D,    C        ;异或结果送 D 位
```

此外，通过位逻辑运算还可以对各种组合逻辑电路进行模拟，即用软件方法来获得组合电路的逻辑功能。

多数位操作指令与同类字节操作指令的助记符完全相同，但位操作指令中有“C”作为操作数，可以以此作为区别；而与 CLR BIT 和 CPL BIT 两条指令相对应的只有 CLR A 和 CPL A，不会发生混淆；最后只剩下 SETB 一条指令，这是位操作独有的。

4. 位控制转移指令组

位控制转移指令是以位的状态作为实现程序转移的判断条件，对这些指令的说明如下。

(1) 以 C 状态为条件的转移指令

```
JC     rel    ; (CY) = 1, 转移指令，其转移控制位：
                若 (CY) = 1, 则 PC ← (PC) + 2 + rel
                若 (CY) ≠ 1, 则 PC ← (PC) + 2
JNC    rel    ; (CY) = 0, 转移指令，其转移控制位：
                若 (CY) = 0, 则 PC ← (PC) + 2 + rel
                若 (CY) ≠ 0, 则 PC ← (PC) + 2
```

(2) 以位状态为条件的转移指令

```
JB     bit,    rel    ;位状态为“1”转移
JNB    bit,    rel    ;位状态为“0”转移
JBC    bit,    rel    ;位状态为“1”转移，并使该位清“0”
```

这 3 条指令都是三字节指令，因此如果状态满足，则程序转移：

PC ← (PC) + 3 + rel ;

否则程序顺序执行：

$PC \leftarrow (PC) + 3。$

2.4 C51 单片机的伪指令

伪指令不要求计算机做任何操作，也没有对应的机器码，不产生目标程序，不影响程序的执行，仅仅是一些能够帮助进行汇编的指令。伪指令主要用来指定程序或数据的起始位置，给出一些连续存放数据的地址或为中间运算结果保留一部分存储空间以及表示源程序结束等。不同版本的汇编语言，伪指令的符号和含义可能有所不同，但基本用法是相似的。本节介绍几种常用的伪指令。

2.4.1 程序起始地址伪指令 ORG

格式：[符号:] ORG 地址（十六进制表示）

该伪指令的功能是规定其后面的目标程序或数据块的起始地址。它放在一段源程序（主程序、子程序）或数据块的前面，说明紧跟在其后的程序段或数据块的起始地址就是 ORG 后面给出的地址。例如：

```
ORG 2000H
START: MOV A, #7FH
.....
```

2.4.2 结束汇编伪指令 END

格式：[符号:] END

END 是汇编语言源程序的结束标志，表示汇编结束。在 END 以后所写的指令，汇编程序都不予处理。一个源程序只能有一个 END 命令，否则就有一部分指令不能被汇编。如果 END 前面加标号的话，则应与被结束程序段的起始点的标号一致，以表示结束的是哪一个程序段。

2.4.3 定义字节伪指令 DB

格式：[标号:] DB 项或项表

该伪指令的功能是把项或项表的数据（字符串按字符顺序以 ASCII 码）存入从标号地址开始的连续存储单元中。其中，项或项表是指一个字节数据，用逗号分开的字节数据串，或以引号括起来的字符串。例如：

```
ORG 2000H
TAB1: DB 30H, 8AH, 7FH, 73
      DB '5', 'A', 'BCD'
```

由于 ORG 2000H，所以 TAB1 的地址为 2000H，因此，以上伪指令经汇编后，将对 2000H 开始的连续存储单元赋值：

(2000H) =30H
 (2001H) =8AH
 (2002H) =7FH
 (2003H) =49H ;十进制数 73 以十六进制数存放
 (2004H) =35H ;35H 是数字 5 的 ASCII 码
 (2005H) =41H ;41H 是字母 A 的 ASCII 码
 (2006H) =42H ;42H 是字符串'BCD'中 B 的 ASCII 码
 (2007H) =43H ;43H 是字符串'BCD'中 C 的 ASCII 码
 (2008H) =44H ;44H 是字符串'BCD'中 D 的 ASCII 码

2.4.4 定义字伪指令 DW

格式: [标号:] DW 项或项表

DW 伪指令与 DB 相似,但用于定义字的内容。项或项表是指所定义的一个字(两个字节)或用逗号分开的字串。汇编时,机器自动按高 8 位先存入,低 8 位在后的格式排列。例如:

```

      ORG 1500H
TAB2: DW 1234H, 80H
汇编后, (1500H) =12H, (1501H) =34H, (1502H) =00H, (1503H) =80H。

```

2.4.5 预留存储空间伪指令 DS

格式: [标号:] DS 表达式

该伪指令的功能是从标号地址开始,保留若干个字节的内存空间以备存放数据。保留的字节单元数由表达式的值决定。例如:

```

      ORG 1000H
      DS 20H
      BD 30H, 8FH

```

汇编后从 1000H 开始,预留 32 (20H) 个字节的内存单元,然后从 1020H 开始,按照下一条 DB 指令赋值,即 (1020H) =30H, (1021H) =8FH。

2.4.6 等值伪指令 EQU

格式: 标号: EQU 项

该伪指令的功能是将指令中的项的值赋予 EQU 前面的标号。项可以是常数、地址标号或表达式。例如:

```

TAB1: EQU 1000H
TAB2: EQU 2000H

```

汇编后 TAB1、TAB2 分别具有值 1000H、2000H。

用 EQU 伪指令对某标号赋值后,该标号的值在整个程序中不能再改变。

2.4.7 位地址定义伪指令 BIT

格式：标号： BIT 位地址

该伪指令的功能是将位地址赋予 BIT 前面的标号，经赋值后可用该标号代替 BIT 后面的位地址。例如：

PLG: BIT F0

AI: BIT P1.0

经以上伪指令定义后，在程序中就可以把 FLG 和 AI 作为位地址来使用。

2.5 C51 单片机的程序设计举例

2.5.1 简单程序设计

简单程序是程序结构中最简单、基本的一种程序结构，它按程序的书写顺序依次执行，程序中没有转移指令。

【例 2-21】 将一个双字节数存入片内 RAM。

设该待存双字节数高字节在工作寄存器 R2 中，低字节在累加器 A 中，要求高字节存入片内 RAM 的 36H 单元，低字节存入 35H 单元，则相应的参考程序如下：

```
ORG      3000H
MOV      R0,    #35H    ;R0 作指向片内 RAM 单元的地址指针，先指向 35H 单元
MOV      @R0,   A      ;低字节存入 35H 单元
INC      R0
          ;使 R0 指向 36H 单元
XCH      A,     R2      ;R2 与 A 的内容交换，待存高字节交换到 A 中
MOV      @R0,   A      ;高字节存入 36H 单元，A 的内容未受影响
XCH      A,     R2      ;R2 与 A 的内容再次交换，两者的内容恢复原状
END
```

2.5.2 分支程序设计

分支程序是根据条件的成立与否来决定执行哪个程序段。在程序中条件转移指令常用来实现分支，无条件转移指令用来控制每个分支的转移方向。

【例 2-22】 编写计算函数 $Y=f(X)$ 的程序，实现以下方程：

$$Y = \begin{cases} 2 & \text{当 } X > 0 \text{ 时} \\ 0 & \text{当 } X = 0 \text{ 时} \\ -2 & \text{当 } X < 0 \text{ 时} \end{cases}$$

该程序有三个分支，要做两次判断：第 1 次用累加器判零指令判断 X 是否为零，第 2 次用位条件转移指令判断 $X > 0$ 还是 $X < 0$ 。程序如下：

```

ORG    3000H
        X      EQU    30H    ;设 X 存放在 30H
        Y      EQU    31H    ;设 Y 存放在 31H
        MOV    A,      X      ; (30H) → A
        JZ     LOOP2          ;若 X=0, 则转 LOOP2
        JNB    ACC.7, LOOP1    ;若 X>0, 则转 LOOP1
        MOV    A,      #0FEH  ;若 X<0, 则 A=-2
        LJMP   LOOP2          ;转 LOOP2
LOOP1:   MOV    A,      #02H    ;若 X>0, 则 A=2
LOOP2:   MOV    Y,      A      ;存结果
END

```

【例 2-23】 设计可多达 128 路分支出口的转移程序。

设 128 路出口分别转向 128 段小程序，它们的首地址依次为 addr00, addr01, addr02, addr03, ..., addr7F。要转移到某分支的信息存放在工作寄存器 R2 中，则转移程序如下：

```

ORG    3000H
        MOV    DPTR, #TAB
        MOV    A,      R2
        RL     A              ;将出口分支信息乘 2
        JMP    @A+DPTR
TAB:    AJMP   addr00
        AJMP   addr01
        AJMP   addr02
        .....
        AJMP   addr7F
END

```

AJMP 指令的转移范围不超出所在的 2KB 区间。如果各段小程序较长，在 2KB 范围内无法全部容纳，则应改用 LJMP 指令。每条 LJMP 指令占用 3 个字节，如改用 LJMP 指令，程序需作如下改动：

```

ORG    3000H
        MOV    DPTR, #TAB
        MOV    A,      R2
        MOV    B,      #3
        MUL    AB          ;以上 3 条指令将出口信息乘 3
        XCH    A,      B    ;积的高 8 位交换到 A，低 8 位暂存于 B
        ADD    A,      DPH
        MOV    DPH,    A    ;把积的高 8 位叠加到 DPH
        XCH    A,      B    ;积的低 8 位交换回 A
        JMP    @A+DPTR
TAB:    LJMP   addr00

```



```

        LJMP    addr01
        LJMP    addr02
        .....
        LJMP    addr7F
END

```

2.5.3 循环程序设计

循环程序是将要重复执行的部分做为一个程序段，然后利用指令来实现循环，直到满足要求为止。

【例 2-24】 设 51 单片机的时钟频率为 12MHz，要求设计一个软件程序，延时时间为 100ms。

由题可知单片机的机器周期为 1μs，可以设计一个延时 1ms 的循环程序作为内程序，共循环 100 次，就可以实现 100ms 的延时。程序如下：

```

ORG    3000H
        MOV    R6,  #100           ;外循环 100 次
LOOP1:  MOV    R7,  #200           ;内循环 1ms
LOOP2:  NOP
        NOP
        NOP
        DJNZ   R7,  LOOP2
        DJNZ   R6,  LOOP1
END

```

2.5.4 数据转换程序设计

在单片机应用系统中，数据的输入输出常采用十进制数，直观方便；内部运算时常使用二进制数，运算简便、存储量小。程序中，经常使用数制转换子程序。

【例 2-25】 将某 8 位二进制数转换为 BCD 码。

设 8 位二进制数已在 A 中，转换后存于片内 RAM 的 20H、21H 单元。程序如下：

```

ORG    3000H
        MOV    B,      #100
        DIV    AB              ;该 8 位二进制数除 100，在 A 中得商，也即转换为
                                ;BCD 码后的百位数
        MOV    R0,      #21H   ;R0 指向 21H 单元
        MOV    @R0,     A       ;百位数存入片内 RAM 的 21H 单元
        DEC    R0            ;调整 R0 指向 20H 单元
        MOV    A,      #10
END

```

2.5.5 查表程序设计

查表是单片机应用程序设计中常用的算法,根据不同的表格结构,可以有多种查表算法。

【例 2-26】设计有 256 路分支出口的转移程序。

设出口信息已在 R2 中。查表取得目标地址,如何把该地址送入 PC 以实现转移是程序设计的关键。下面的程序巧妙地用 PUSH 与 RET 指令将目标地址送入 PC,达到转移的目的。程序如下:

```

MOV    DPTR, #TAB
MOV    A,     R2           ;出口信息送 A
CLR    C
RLC    A                ;出口信息乘 2
JNC    LOW        ;出口信息在 0~127 之间则转移
INC    DPH            ;出口信息在 128~255 之间,则 DPTR 加 256
LOW:   MOV    R3,     A        ;出口信息暂存于 R3
MOVC   A,     @A+DPTR    ;查找目标地址的低 8 位
PUSH   A                ;目标地址的低 8 位压栈
MOV    A,     R3          ;A 恢复为出口信息
INC    A
MOVC   A,     @A+DPTR    ;查找目标地址的高 8 位
PUSH   A                ;目标地址的高 8 位压栈
RET

```

```

TAB:   DW    1520H, 3721H, 4246H, 7850H
        DW    3483H, 2657H, 246H, 9837H
        DW    232H, 3278H, 3452H, 9738H
        DW    4325H, 4352H, 8432H, 6943H

```

习题和思考题

2-1 指出下列指令中操作数的寻址方式。

```

MOV    R0,     #60H
MOV    A,     30H
MOV    A,     @R0
MOVC   A,     @A+DPTR
CJNE   A,     #00H,     ONE
CPL    C
MOV    C,     30H

```

2-2 给下面一段程序加上机器码，并说明程序运行后寄存器 A，R0 和内部 RAM 50H，51H、52H 单元中的内容为何值。

```
MOV  50H,  #50H
MOV  A,     50H
MOV  R0,    A
MOV  A,     #30H
MOV  @R0,   A
MOV  A,     #50H
MOV  51H,   A
MOV  52H,   #00H
```

2-3 指令 POP B 的源操作数是_____，它是_____寻址方式，目的操作数是_____，它是_____寻址方式。

2-4 已知 SP=25H，PC=4345H，(24H)=12H，(25H)=34H，(26H)=56H，当执行 RET 指令后，SP=_____，PC=_____。

2-5 试编写一段程序，将内部数据存储器 30H，31H 单元内容传送到外部数据存储器 1000H，1001H 单元中去。

2-6 已知指令 AJMP addr11 的机器码为 41H 和 FFH，指令所在的地址为 0810H，求该转移指令的目的地址。

2-7 求指令 H: SJMP H 中的地址偏移量。

2-8 传送类指令中有 MOV、MOVX、MOVC 助记符形式，它们区别是什么？C51 单片机指令 MOV，MOVX，MOVC 的源操作数的最大地址分别为多少？

2-9 判断以下指令的正误。

- (1) MOV 28H, @R2
- (2) DEC DPTR
- (3) INC DPTR
- (4) CLR R0
- (5) CPL R5
- (6) MOV R0, R1
- (7) PUSH DPTR
- (8) MOV F0, C
- (9) MOV F0, ACC.3
- (10) RLC R0

第3章 单片机的C程序设计

重点与难点

本章重点是掌握 C51 单片机的程序结构, 各种数据类型及变量的定义、存储及运算。C51 单片机程序的流程控制, C51 单片机函数的定义及调用。难点是掌握数据运算的优先级, C51 单片机构造数据类型的存储结构及具体的灵活应用。

基本要求

1. 掌握 C51 单片机的程序结构
2. 掌握数据类型及变量的定义、存储及运算
3. 掌握 C51 单片机的构造数据类型及存储结构
4. 掌握 C51 单片机的程序的流程控制
5. 掌握 C51 单片机的函数的定义及调用

本章内容

1. 单片机程序设计概论
2. C51 单片机的数据与运算
3. C51 单片机流程控制语句
4. C51 单片机构造数据类型
5. C51 单片机的函数
6. C51 单片机的库函数

3.1 单片机程序设计概论

C51 系列单片机支持三种高级语言, 即 PL/M、C 和 BASIC。8052 单片机内固化有解释 BASIC 语言的解释程序, BASIC 语言适用于简单编程而对编程效率运行速度要求不高的场合。PL/M 是一种结构化的语言, 很像 PASCAL, PL/M 编译器能像汇编器一样产生紧凑的机器代码, 可以说是高级汇编语言, 但它不支持复杂的算术运算, 没有丰富库函数支持, 学习 PL/M 无异于学习一种新的语言。C 语言是一种通用的程序设计语言, 其代码效率高, 数据类型及运算符丰富, 并具有良好的程序结构, 适用于各种应用的程序设计, 是目前使用较广的单片机编程语言。

3.1.1 单片机的C语言编程特点

应用C51编程具有以下优点：

- (1) C51管理内部寄存器和存储器的分配，编程时，无需考虑不同存储器的寻址和数据类型等细节问题；
- (2) 程序由若干函数组成，具有良好的模块化结构；
- (3) 有丰富的子程序库可直接引用，从而大大减少用户编程的工作量；
- (4) C语言和汇编语言可以交叉使用。汇编语言程序代码短、运行速度快，但复杂运算编程耗时。如果用汇编语言编写与硬件有关的部分程序，用C语言编写与硬件无关的运算部分程序，充分发挥两种语言的长处，可以提高开发效率。

3.1.2 C51单片机的C语言编译器

用C语言编写的应用程序必须经过单片机的C语言编译器（简称C51）转换生成单片机可执行的代码程序。支持C51系列单片机的C语言编译器有很多种，如American Automation、Auocet、BSO/TASKING、DUNFIELD SHAREWARE、Keil/Franklin等。其中，Keil/Franklin以它的代码紧凑和使用方便等特点优于其他编译器。本章针对这种编译器来介绍C51单片机的C语言程序设计。

3.1.3 C51单片机的程序结构

同标准C一样，C51单片机的程序由一个个函数组成，这里的函数和其他语言的“子程序”或“过程”具有相同的意义，其中必须有一个主函数main()。程序的执行从main()函数开始，调用其他函数后返回主函数main()，最后在主函数中结束整个程序而不管函数的排列顺序如何。

C语言程序的组成结构如下所示：

```

全局变量说明                /*可被各函数引用*/
main()                        /*主函数*/
{
    局部变量说明              /*只在本函数引用*/
    执行语句（包括函数调用语句）
}

fun1(形式参数表)              /*函数1*/
形式参数说明
{
    局部变量说明
    执行语句（包括调用其他函数语句）
}

```

```

...
funn (形式参数表)          /*函数 n*/
形式参数说明
{
    局部变量说明
    执行语句
}

```

C 语言的语句规则如下所示:

- (1) 每个变量必须先说明后引用, 变量名英文大小写是有差别的;
- (2) C 语言程序一行可以书写多条语句, 但每个语句必须以 “;” 结尾, 一个语句也可以多行书写;
- (3) C 语言的注释用 “/*.....*/” 表示;
- (4) “{” 必须成对, 位置随意, 可在紧挨函数名后, 也可另起一行; 多个花括号可以同行书写, 也可逐行书写。为了层次分明, 增加可读性, 同一层的 “{” 应对齐, 并采用逐层缩进方式书写。

3.2 C51 的数据与运算

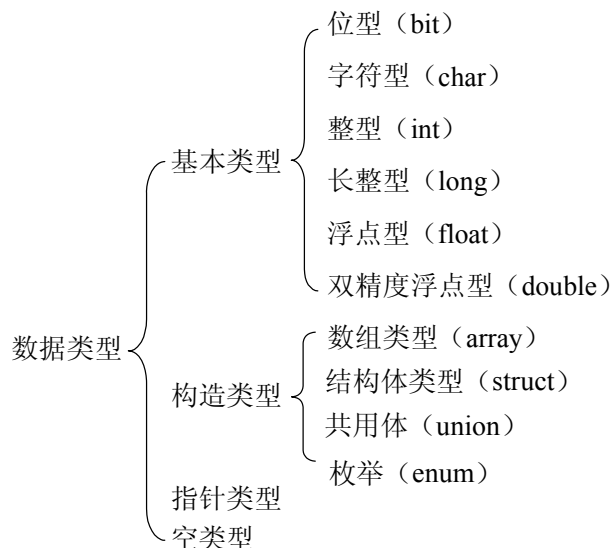
3.2.1 数据与数据类型

数据: 具有一定格式的数字或数值叫做数据。数据是计算机操作的对象。不管使用任何语言、何种算法进行程序设计, 最终在计算机中运行的只有数据流。

数据类型: 数据的不同格式叫作数据类型。

数据结构: 数据按一定的数据类型进行的排列、组合、架构称为数据结构。

C51 提供的数据结构是以数据类型的形式出现的, C51 的数据类型如下所示。



Franklin C51 编译器具体支持的数据类型有：位型（bit）、无符号字符（unsigned char）、有符号字符（signed char）、无符号整型（unsigned int）、有符号整型（signed int）、无符号长整型（unsigned long）、有符号长整型（signed long）、浮点型（float）和指针类型等，如表 3-1 所示。

表 3-1 Franklin C51 的数据类型

数据类型	长度 (bit)	长度 (byte)	值 域
bit	1	...	0, 1
unsigned char	8	1	0~255
signed char	8	1	-128~127
unsigned int	16	2	0~65 535
signed int	16	2	-32 768~32 767
unsigned long	32	4	0~4 294 967 295
signed long	32	4	-2 147 483 648~214 748 364 7
float	32	4	$\pm 1.176\text{E}-38 \sim \pm 3.40\text{E}+38$ (6 位数字)
double	64	8	$\pm 1.176\text{E}-38 \sim \pm 3.40\text{E}+38$ (10 位数字)
一般指针	24	3	存储空间 0~65 535

编译的数据类型（如结构）包含表 3-1 所示的数据类型。由于 C51 系列单片机是 8 位机，因而不存在字节校准问题，这意味着数据结构成员是顺序放置的。

当计算结果隐含着另外一种数据类型时，数据类型可以自动进行转换，例如，将一个位变量赋给一个整型变量时，位型值自动转换为整值，有符号变量的符号也能自动进行处理。这些转换也可以用 C 语言的标准指令进行人工转换。

3.2.2 常量与变量

C51 的数据有常量和变量之分。

常量是指在程序运行中其值不变的量，可以为字符、十进制数或十六进制数（用 0x 表示）。

常量分为数值型常量和符号型常量，如果是符号型常量，需用宏定义指令（#define）对其进行定义（相当于汇编的“EQU”伪指令）。例如：

```
#define PI 3.1415
```

那么程序中只要出现 PI 的地方，编译程序都编译为 3.1415。

变量是指在程序运行中其值可以改变的量。

一个变量由变量名和变量值构成，变量名即是存储单元地址的符号表示，而变量的值就是该单元存放的内容。定义一个变量，编译系统就会自动为它安排一个存储单元，具体的地址值用户不必在意。

常见的变量如下所示。

(1) 位变量 (bit)

与 C51 硬件特性操作有关的可以定义成位变量。位变量必须定位在 C51 单片机片内 RAM 的位寻址空间中。

(2) 字符变量 (char)

字符变量的长度为 1 字节，即 8 位。这很合适 C51 单片机，因为 C51 单片机每次可处理 8 位数据。无符号变量 (unsigned char) 的值域范围是 0~255。对于有符号字符变量 (signed char) 来说，最具有重要意义的位是最高位上的符号标志位 (msb)，此位为 1 代表“负”，为 0 代表“正”。有符号字符变量和无符号字符变量在表示 0~127 的数值时，其含义是一样的，都是 0~0x7F。负数一般用补码表示，即用 11111111 表示 -1，用 11111110 表示 -2……。当进行乘除法运算时，符号问题就变得十分复杂，而 C51 编译器会自动地将相应的库函数调入程序中来解决这个问题。

(3) 整型变量 (int)

整型变量的长度为 16 位。与 8080 和 8086 CPU 系列不同，C51 系列单片机将 int 型变量的高位字节数存放在低地址字节中，低位字节数存放在高地址字节中。有符号整型变量 (signed int) 也使用 msb 位作符号标志位，并使用二进制补码表示数值。可直接使用几种专用的机器指令来完成多字节的加、减、乘、除运算。整型变量值 0x1234 以图 3-1 所示的方式存放在内存中，而长整型变量值 0x12345678 以图 3-2 所示的方式存放在内存中。

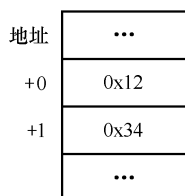


图 3-1 整型数的存储结构



图 3-2 长整型变量的存储结构

(4) 浮点型变量 (float)

浮点型变量为 32 位，占 4byte，许多复杂的数学表达式都采用浮点变量数据类型。其应用符号位表示数的符号，用阶码和尾数表示数的大小。

用它们进行任何数学运算都需要使用由编译器决定的各种不同效率等级的库函数。Franklin C51 的浮点变量数据类型的使用格式与 IEEE-754 标准有关，具有 24 位精度，尾数的高位始终为“1”，因而不保存，位的分布如下：

- (1) 1 位符号位；
- (2) 8 位指数位；
- (3) 23 位尾数。

符号位是最高位，尾数为低 23 位，内存中字节存储顺序如下所示：

地 址	+0	+1	+2	+3
内 容	MMMM MMMM	MMMM MMMM	EMMM MMMM	SEEE EEEE

其中，S 为符号位，1 表示负，0 表示正；E 为阶码；M 为 23 位尾数，最高位为“1”。

浮点变量值-12.5的十进制为：0xC1480000，它按图3-3所示方式存于内存中。

在编程时，如果只强调运算速度而不进行负数运算时，最好采用无符号（unsigned）格式。

无论何时，应尽可能使用无符号字符变量，因为它能直接被C51所接受。基于同样的原因，也应尽量使用位变量。有符号字符变量虽然也只占用一个字节，但需要进行额外的操作来进行测试代码的符号位。这无疑会降低代码效率。

使用简化形式定义数据类型的方法是在源程序开头使用#define语句自定义简化的类型标识符。例如：

```
#define uchar unsigned char
```

```
#define uint unsigned int
```

这样，在编程中，就可以用uchar代替unsigned char，用uint代替unsigned int来定义变量。

地址 +0	0x00
+1	0x00
+2	0x48
+3	0xC1
	...

图3-3 浮点数的存储结构

3.2.3 C51 数据的存储类型与存储器结构

1. 数据的存储类型

C51是面向8XX51系列单片机及硬件控制系统的开发语言，它定义的任何变量必须以一定存储类型的方式定位在8XX51的某一存储区中，否则便没有意义。因此在定义变量类型时，还必须定义它的存储类型，C51的变量的存储类型如表3-2所示，存储类型及其数据长度和值域如表3-3所示。

表3-2 C51 的变量的存储类型

存 储 类 型	与存储空间的对 应 关 系
data	直接寻址片内数据存储区，访问速度快（128byte）
bdata	可位寻址片内数据存储区，允许位与字节混合访问（16byte）
idata	间接寻址片内数据存储区，可访问片内全部RAM地址空间（256byte）
pdata	分页寻址片外数据存储区（256byte）由MOV @Ri访问（i=0, 1）
xdata	片外数据存储区（64 KB）由MOVX @DPTR访问
code	程序存储器64 KB空间，由MOVC @DPTR访问

表3-3 C51 存储类型及其数据长度和值域

存 储 类 型	长 度（bit）	长 度（byte）	值 域 范 围
data	8	1	0~255
idata	8	1	0~255
pdata	8	1	0~255
xdata	16	2	0~65 535
code	16	2	0~65 535

访问内部数据存储器（idata）比访问外部数据存储器（xdata）相对要快一些，因此，可将经常使用的变量置于内部数据存储器中，而将较大以及很少使用的数据变量置于外部数据存储器中。例如，定义变量 x 语句：`data char x`（等价于 `char data x`）。如果用户不对变量的存储类型定义，则编译器承认默认存储类型，默认的存储类型由编译控制命令的存储模式部分决定。

2. 存储器模式

存储器模式决定了变量的默认存储器类型、参数传递区和无明确存储区类型的说明。C51 的存储器模式有 SMALL、LARGE 和 COMPACT 如表 3-4 所示。

在固定的存储器地址进行变量参数传递是 C51 的一个标准特征，在 SMALL 模式下参数传递是在内部数据存储区中完成的。LARGE 和 COMPACT 模式允许参数在外部存储器中传递。C51 同时也支持混合模式，例如，在 LARGE 模式下生成的程序可将一些函数分页放入 SMALL 模式中，从而加快执行速度。

表 3-4 存储器模式

存 储 模 式	说 明
SMALL	默认的存储类型是 <code>data</code> ，参数及局部变量放入可直接寻址片内 RAM 的用户区中（最大 128byte）。另外所有对象（包括堆栈），都必须嵌入片内 RAM。栈长很关键，因为实际栈长依赖于函数嵌套调用层数
COMPACT	默认的存储类型是 <code>pdata</code> ，参数及局部变量放入分页的外部数据存储区，通过 <code>@R0</code> 或 <code>@R1</code> 间接访问，栈空间位于片内数据存储区中
LARGE	默认的存储类型是 <code>xdata</code> ，参数及局部变量直接放入片外数据存储区，使用数据指针 <code>DPTR</code> 来进行寻址。用此数据指针进行访问效率较低，尤其对两个或多个字节的变量，这种数据类型的访问机制直接影响代码的长度

例如，设 C 语言源程序为 PROR.C，若使程序中的变量类型和参数传递区限定在外部数据存储区，有两种方法。

方法 1：用 C51 对 PROR.C 进行编译时，使用命令 `C51 PROR.C COMPACT`。

方法 2：在程序的第一句加预处理命令 `#pragma compact`。

3. 变量说明举例

```
data char var;           /*字符变量 var 定位在片内数据存储区*/
char code MSG[ ]=" PARAMETER:" ; /*字符数组 MSG[ ]定位在程序存储区*/
unsigned long xdata array[100]; /*无符号长型数组定位在片外 RAM 区，每元素
                                占 4byte*/
float idata x, y, z;      /*实型变量 x, y, z，定位在片内用间址访问的
                            内部 RAM 区*/
bit lock;                /*位变量 Lock 定位在片内 RAM 可位寻址区*/
unsigned int pdata sion;  /*无符号整型变量 sion 定位在分页的外部 RAM*/
unsigned char xdata vector[10][4][4]; /*无符号字符型三维数组，定位在片外 RAM 区*/
sfr P0=0x80;             /*定义 P0 口，地址为 80H*/
```

```
char bdata flags;           /*字符变量 flags 定位在可位寻址内部 RAM 区*/
sbit flag0=flags^0;        /*定义 flag0 为 flags.0 */
```

如果在变量说明时略去存储器类型标志符，编译器会自动选择默认的存储器类型。默认的存储器类型由控制指令 SMALL、COMPACT 和 LARGE 限制。例如，如果声明 char var，则默认的存储器模式为 SMALL，var 放在 data 存储区；如果使用 COMPACT 模式，var 放入 idata 存储区；在使用 LARGE 模式的情况下，var 被放入外部数据存储区（xdata 存储区）。

4. 指针变量说明举例

```
long xdata *px;             /*指针 px 指向 long 型 xdata 区（每个数
                             据占四个单元），指针自身在默认存储器
                             （如不指定编译模式在 data 区），指针长
                             度为 2byte*/

char xdata *data pd;        /*指针 pd 指向字符型 xdata 区，自身在
                             data 区，长度为 2byte*/

data char xdata *pd;        /*与上例等效*/
data int *pn;               /*和 int *data pn 及 intr*pn 等效，定义一
                             个类型为 int 型的通用型指针，指针自
                             身在 data 区长度为 3byte*/
```

在上例的指针声明中包含如下几个内容：

（1）指针变量名（如 px）前面冠以“*”，表示 px 为指针型变量，此处*不带取内容之意。

（2）指针指向的存储类型，即指向哪个存储区，它决定了指针本身的长度。存储类型声明的位置在数据类型和指针名（如*px）之间，如无此项声明，则此指针型变量为通用型。

（3）指针指向的存储区的数据类型，即被指向的存储区以多少个单元作一个数据单位。当程序通过指针对该区操作时，将按此规定的单元个数的内容作为一个数据操作。

（4）指针变量自身的存储类型，即指针处于什么区与自身的长度无关，该声明可位于声明语句的开头，也可在“*”和变量名之间。此项由编译模式放在默认区，如无规定编译模式，通常在 data 区。

3.2.4 特殊功能寄存器（SFR）及其定义

C51 单片机中，除了程序计数器（PC）和 4 组工作寄存器组外，其他所有的寄存器均为特殊功能寄存器（SFR），分散在片内 RAM 区的高 128 字节中，地址范围为 80H~0FFH。SFR 中有 11 个寄存器具有位寻址能力，它们的字节地址都能被 8 整除，即字节地址是以 8 或 0 为尾数的。

为了能直接访问这些 SFR，Franklin C51 提供了一种自主形式的定义方法，这种定义方法与标准 C 语言不兼容，只适用于对 C51 系列单片机进行 C 语言编程。特殊功能寄存器 C51 定义的一般语法格式如下：

```
sfr sfr-name = int constant;
```

“sfr”是定义语句的关键字，其后必须跟一个 MCS-51 单片机真实存在的特殊功能寄存器名；“=”后面必须是一个整型常数，不允许是带有运算符的表达式，它是特殊功能寄存器“sfr-name”的字节地址，这个常数值的范围必须在 SFR 地址范围内，位于 0x80~0xFF。

例如：

```
sfr  SCON=0x98;          /* 串口控制寄存器地址 98H */
sfr  TMOD=0x89;          /* 定时器/计数器方式控制寄存器地址 89H */
```

C51 系列单片机的特殊功能寄存器的数量与类型不尽相同，因此建议将所有特殊的“sfr”定义放入一个头文件中，该文件应包括 C51 单片机系列机型中的 SFR 定义。C51 编译器的“reg51.h”头文件就是这样一个文件。

在新的 C51 系列产品中，SFR 在功能上经常组合为 16 位值，当 SFR 的高字节地址直接位于低字节之后时，对 16 位 SFR 的值可以直接进行访问。例如，52 子系列的定时器/计数器 2 就是这种情况。为了有效地访问这类 SFR，可使用关键字“sfr16”来定义，其定义语句的语法格式与 8 位 SFR 相同，只是“=”后面的地址必须用 16 位 SFR 的低字节地址，即低字节地址作为“sfr16”的定义地址。

例如：

```
sfr16  T2 = 0xCC;          /*定时器/计数器 2: T2 低 8 位地址为 0CCH, T2 高 8 位地址为 0CDH*/
```

这种定义适用于所有新的 16 位 SFR，但不能用于定时器/计数器 0 和 1。

对于位寻址的 SFR 中的位，C51 的扩充功能支持特殊位的定义，像 SFR 一样不与标准 C 兼容，需使用“sbit”来定义位寻址单元。

(1) 第 1 种格式：sbit bit-name = sfr-name^int constant;

“sbit”是定义语句的关键字，后跟一个寻址位符号名（该位符号名必须是 C51 单片机中规定的位名称），“=”后的“sfr-name”必须是已定义过的 SFR 的名字，“^”后的整常数是寻址位在特殊功能寄存器“sfr-name”中的位号，必须是 0~7 范围中的数。例如：

```
sfr  PSW=0xD0;          /* 定义 PSW 寄存器地址为 D0H */
sbit  OV=PSW^2;          /* 定义 OV 位为 PSW.2, 地址为 D2H */
sbit  CY=PSW^7;          /* 定义 CY 位为 PSW.7, 地址为 D7H */
```

(2) 第 2 种格式：sbit bit-name = int constant^int constant;

“=”后的 int constant 为寻址地址位所在的特殊功能寄存器的字节地址，“^”符号后的 int constant 为寻址位在特殊功能寄存器中的位号。例如：

```
sbit  OV=0XD0^2;          /* 定义 OV 位地址是 D0H 字节中的第 2 位 */
sbit  CY=0XD0^7;          /* 定义 CY 位地址是 D0H 字节中的第 7 位 */
```

(3) 第三种格式：sbit bit-name = int constant;

“=”后的 int constant 为寻址位的绝对位地址。例如：

```
sbit  OV=0XD2;          /* 定义 OV 位地址为 D2H */
sbit  CY=0XD7;          /* 定义 CY 位地址为 D7H */
```

特殊功能位代表了一个独立的定义类，不能与其他位定义和位域互换。

C51 也建立了一个头文件 `reg51.h` (增强型为 `reg52.h`)，在该文件中对所有的特殊功能寄存器进行了 `sfr` 定义，对特殊功能寄存器的有位名称的可寻址位进行了 `sbit` 定义，因此，只要程序中包含语句 `#include<reg51.h>`，就可以直接引用特殊功能寄存器名，或直接引用位名称。

要特别注意：在引用时特殊功能寄存器或者位名称必须大写。下面给出了特殊功能寄存器的一览表，如表 3-5 所示。

表 3-5 特殊功能寄存器一览表

SFR	MSB 位地址/位定义 LSB								字节地址
B									F0H
ACC									E0H
PSW	D7	D6	D5	D4	D3	D2	D1	D0	D0H
	CY	AC	F0	RS1	RS0	OV	F1	P	
IP	BF	BE	BD	BC	BB	BA	B9	B8	B8H
	/	/	/	PS	PT1	PX1	PT0	PX0	
P3	B7	B6	B5	B4	B3	B2	B1	B0	B0H
	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0	
IE	AF	AE	AD	AC	AB	AA	A9	A8	A8H
	EA	/	/	ES	ET1	EX1	ET0	EX0	
P2									A0H
SBUF									99H
SCON	9F	9E	9D	9C	9B	9A	99	98	98H
	SM0	SM1	SM2	REN	TB8	RB8	TI	RI	
P1									90H
TH1									8DH
TH0									8CH
TL1									8BH
TL0									8AH
TMOD	GATE	C/T	M1	M0	GATE	C/T	M1	M0	89H
TCON	8F	8E	8D	8C	8B	8A	89	88	88H
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	
PCON	SMOD	/	/	/	GF1	GF0	FD	IDL	87H
DPH									83H
DPL									82H
SP									81H
P0									80H

3.2.5 并行接口及其定义

C51 系列单片机并行 I/O 接口除了芯片上的 4 个 I/O 口 (P0~P3) 外，还可以在片外扩

展 I/O 口。C51 单片机 I/O 口与数据存储器统一编址，即把一个 I/O 口当作数据存储器中的一个单元来看待。

使用 C51 进行编程时，C51 片内的 I/O 口与片外扩展的 I/O 口可以统一在一个头文件中定义，也可以在程序中（一般在开始的位置）进行定义，其定义方法如下。

对于 C51 片内 I/O 口按特殊功能寄存器方法定义。例如：

```
sfr P0=0x80;          /* 定义 P0 口，地址为 80H */
sfr P1=0x90;          /* 定义 P1 口，地址为 90H */
```

对于片外扩展 I/O 口，则根据硬件译码地址，将其视作为片外数据存储器中的一个单元，使用#define 语句进行定义。例如：

```
#include <absacc.h>
#define PORTA XBYTE [0xFFC0]
```

absacc.h 是 C51 中绝对地址访问函数的头文件，程序将 PORTA 定义为外部 I/O 口，地址为 FFC0H，长度为 8 位。

一旦在头文件或程序中对这些片外 I/O 口进行定义后，在程序中就可以自由使用变量名与其实际地址进行联系，以便使程序员能用软件模拟 C51 的硬件操作。

3.2.6 位变量（BIT）及其定义

C51 对位变量的定义有 5 种方法。

1. 位变量 C51 定义。使用 C51 编程时，定义位变量后，就可以用定义的变量来表示 C51 的位寻址单元。位变量的 C51 定义的一般语法格式如下：

位类型标识符 (bit) 位变量名；

例如：

```
bit direction_bit;      /* 把 direction_bit 定义为位变量 */
bit look_pointer;       /* 把 look_pointer 定义为位变量 */
```

2. 采用字节寻址变量·位的方法。

例如：

```
bdata int ibase;         /*ibase 定义为整型变量*/
sbit mybit=ibase^15;     /*mybit 定义为 ibase 的 D15 位*/
```

这里位是运算符“^”相当于汇编中的“·”，其后的最大取值依赖于该位所在的字节寻址变量的定义类型，如定义为 char 最大值只能为 7。

3. 函数可包含类型为“bit”的参数，也可以将其作为返回值。

例如：

```
bit func(bit b0, bit b1) /* 变量 b0, b1 作为函数的参数 */
{
    return (b1);         /* 变量 b1 作为函数的返回值 */
}
```

注意，使用（#pragma disable）或包含明确的寄存器组切换（using n）的函数不能返回位值，否则编辑器将会给出一个错误信息。

4. 对特殊功能寄存器的位的定义

方法 1: 使用头文件及 sbit 定义符, 多用于无位名的可寻址位。

例如: #include <reg51.h>

```
sbit P1_1=P1^1;      /*P1-1 为 P1 口的第 1 位*/
sbit ac=ACC^7;       /*ac 定义为累加器 A 的第 7 位*/
```

方法 2: 使用头文件 reg51.h, 再直接用位名称。

例如: #include <reg51.h>

```
RS1=1;
RS0=0;
```

方法 3: 用字节地址位表示。

例如: sbit OV=0xD0^2;

方法 4: 用寄存器名·位定义。

```
sfr PSW=0xd0;      /*定义 PSW 地址为 d0H*/
sbit CY=PSW^7;     /*CY 为 PSW·7*/
```

5. 对位变量定义的限制。位变量不能定义成一个指针, 例如不能定义: bit *bit_pointer; 不存在位数组, 例如不能定义: bit b_array[]。

在位定义中, 允许定义存储类型, 位变量都被放入一个位段, 此段总位于 C51 片内的 RAM 区中。因此, 存储类型限制为 data 和 idata, 如果将位变量的存储类型定义成其他存储类型都将编译出错。

例如, 先定义变量的数据类型和存储类型:

```
bdata int ibase;      /* 定义 ibase 为 bdata 整型变量 */
bdata char bary[4];   /* bary[4]定义为 bdata 字符型数组 */
```

然后可使用“sbit”定义可独立寻址访问的对象位:

```
sbit mybit0 = ibase^0; /* mybit0 定义为 ibase 的第 0 位 */
sbit mybit15 = ibase^15; /* mybit15 定义为 ibase 的第 15 位 */
sbit Ary07 = bary[0]^7; /* Ary07 定义为 abry[0]的第 7 位 */
sbit Ary37 = bary[3]^7; /* Ary37 定义为 abry[3]的第 7 位 */
```

对象 ibase 和 bary 也可以进行字节寻址:

```
Ary37=0;              /* bary[3]的第 7 位赋值为 0 */
bary[3]='a';          /* 字节寻址, bary[3] 赋值为'a' */
```

sbit 定义的位寻址对象所在字节基址对象的存储类型为“bdata”, 否则只有绝对的特殊位定义(sbit)是合法的。“^”操作符后的最大值依赖于指定的基类型, 对于 char/uchar 而言是 0~7, 对于 int/uint 而言是 0~15, 对于 long/ulong 而言是 0~31。

3.2.7 C51 运算符、表达式及其规则

C 语言的运算符范围很宽, 把除了控制语句和输入输出以外的几乎所有的基本操作都作为运算符处理, 例如将赋值符“=”作为赋值运算符, 方括号作为下标运算符等。c 的运算符有以下几类: 算术运算符, 关系运算符, 逻辑运算符和位运算符。

1. 算术运算符和算术表达式

(1) 基本的算术运算符

- ① + (加法运算符, 或正值运算符。例如 3+5、+3)
- ② - (减法运算符, 或负值运算符。例如 5-2、-3)
- ③ * (乘法运算符。例如 3*5)
- ④ / (除法运算符。例如 5/3)
- ⑤ % (模运算符, 或称求余运算符, %两侧均应为整型数据, 例如 7%4 的值为 3)。

(2) 算术表达式和运算符的优先级与结合性

用算术运算符和括号将运算对象 (也称操作数) 连接起来的, 符合 C 语法规则的式子, 称 **c** 算术表达式。运算对象包括常量、变量、函数等。例如, 下面是一个合法的 **c** 算术表达式:

`a*b/c-1.5+'a'`

C 语言规定了运算符的优先级和结合性。在表达式求值时, 先按运算符的优先级别高低次序执行, 例如先乘除后加减。如表达式 `a-b*c`, `b` 的左侧为减号, 右侧为乘号, 而乘号优先于减号, 因此相当于 `a-(b*c)`。如果在一个运算对象两侧的运算符的优先级别相同, 如 `a-b+c`, 则按规定的“结合方向”处理。

C 规定了各种运算符的结合方向 (结合性), 算术运算符的结合方向为“自左至右”, 即先左后右, 因此 `b` 先与减号结合, 执行 `a-b` 的运算, 再执行加 `c` 的运算。“自左至右的”结合方向”又称“左结合性”, 即运算对象先与左面的运算符结合。以后可以看到有些运算符的结合方向为“自右至左”, 即右结合性 (例如, 赋值运算符)。关于“结合性”的概念在其他一些高级语言中是没有的, 是 C 的特点之一, 希望能弄清楚。

如果一个运算符的两侧的数据类型不同, 则会先自动进行类型转换, 使二者具有同一种类型, 然后进行运算。

(3) 类型转换

一种是自动 (缺省) 类型转换, 即在程序编译时由 C 编译自动进行数据类型转换。

如图 3-4 所示为自动数据类型转换规则。图中横向向左箭头表示必定的转换, 如 `char`, `int` 变量同时存在时, 必定将 `char` 转换成 `int` 类型; 当 `float` 与 `double` 类型共存时, 在运算时一律先转换成 `double` 类型, 以提高运算精度。图中纵向箭头表示当运算对象为不同类型时的转换方向。例如, `int` 与 `long` 型数据进行运算时, 先将较低类型 `int` 转成较高的类型 `long`, 然后再进行运算, 结果为 `long` 类型。一般来说, 当运算对象的数据类型不相同, 先将较低的数据类型转换成较高的数据类型, 运算的结果为较高的数据类型。

另一种数据类型的转换方式为强制类型转换, 需要使用强制类型转换运算符, 其形式为: (类型名)(表达式)

例如:

- `(double)a` 将 `a` 强制转换成 `double` 类型。
- `(int)(x+y)` 将 `x+y` 的值强制转换成 `int` 类型。

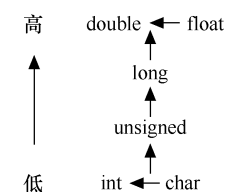


图 3-4 自动类型转换规则

(float)(5%3) 将模运算 5%3 的值强制转换成 float 类型。

使用强制转换类型运算符后，运算结果被强制转换成规定的类型。

例如：

```
unsigned char x, y;
unsigned int z;
z=x-(unsigned int)y;
z=(unsigned int)x*y;
```

这样的加法和乘法才能保证结果超过一字节时仍正确。

(4) 自增、自减运算符

作用是使变量的值增 1 或减 1，例如：

```
++i, --i (在使用 i 之前，先使 i 的值加 (减) 1)
i++, i-- (在使用 i 之后，使 i 的值加 (减) 1)
```

粗略地看，++i 和 i++ 的作用相当于 i=i+1。但 ++i 和 i++ 不同之处在于 ++i 是先执行 i=i+1 后，再使用 i 的值；而 i++ 是先使用 i 的值后，再执行 i=i+1。如果 i 的原值等于 3，则执行下面的赋值语句：

```
j = ++i; (i 的值先变成 4，再赋给 j，j 的值为 4)
j = i++; (先将 i 的值 3 赋给 j，j 的值为 3，然后 i 变为 4)
```

注意：

① 自增运算符 (++) 和自减运算符 (--) 只能用于变量，而不能用于常量或表达式，如 5++ 或 (a+b)++ 都是不合法的。因为 5 是常量，常量的值不能改变；(a+b)++ 也不可能实现，假如 a+b 的值为 5，那么自增后得到的 6 放在什么地方呢？无变量可供存放。

② ++ 和 -- 的结合方向是“自右至左”。前面已提到，算术运算符的结合方向为“自左而右”，这是大家所熟知的。如果有 -i++，i 的左面是负号运算符，右面是自加运算符。如果 i 的原值等于 3，若按左结合性，相当于 (-i)++，而 (-i)++ 是不合法的，因为对表达式不能进行自加自减运算。负号运算符和 “++” 运算符同优先级，而结合方向为“自右至左”（右结合性），即它相当于 -(i++)，如果有 -i++，则先取出 i 的值 3，输出 -i 的值 -3，然后 i 增值为 4。注意 -(i++) 是先用 i 的原值 3 加上负号输出 -3，再对 i 加 1，不要认为先加完 1 后再加负号，输出 -4，这是不对的。

自增（减）运算符常用于循环语句中，使循环变量自动加 1，也用于指针变量，使指针指向下一个地址。这些将在以后的章节中介绍。

2. 关系运算符和关系表达式

关系运算是逻辑运算中比较简单的一种。所谓“关系运算”实际上是“比较运算”，将两个值进行比较，判断其比较的结果是否符合给定的条件。例如，a>3 是一个关系表达式，大于号 (>) 是一个关系运算符，如果 a 的值为 5，则满足给定的“a>3”条件，因此关系表达式的值为“真”（即“条件满足”）；如果 a 的值为 2，不满足“a>3”条件，则称关系表达式的值为“假”。

(1) 关系运算符及其优先次序

C 语言提供 6 种关系运算符：

优先级相同（高）： $<$ （小于）， $<=$ （小于或等于）， $>$ （大于）， $>=$ （大于或等于）

优先级相同（低）： $==$ （等于）， $!=$ （不等于）

下面对关系运算符的优先次序进行讨论。

① 前 4 种关系运算符（ $<$ ， $<=$ ， $>$ ， $>=$ ）的优先级别相同，后两种也相同。前 4 种高于后 2 种。例如，“ $>$ ”优先于“ $==$ ”，而“ $>$ ”与“ $<$ ”优先级相同。

② 关系运算符的优先级低于算术运算符。

③ 关系运算符的优先级高于赋值运算符。

例如：

$c > a + b$	等效于 $c > (a + b)$
$a > b == c$	等效于 $(a > b) == c$
$a == b < c$	等效于 $a == (b < c)$
$a = b > c$	等效于 $a = (b > c)$

(2) 关系表达式

用关系运算符将两个表达式（可以是算术表达式、关系表达式、逻辑表达式、赋值表达式、字符表达式）连接起来的式子，称关系表达式。例如，下面都是合法的关系表达式：

$a > b$, $a + b > b + c$, $(a = 3) > (b = 5)$, $'a' < 'b'$, $(a > b) > (b < c)$

关系表达式的值是一个逻辑值，即“真”或“假”。例如，关系表达式“ $5 == 3$ ”的值为“假”，“ $5 > 0$ ”的值为“真”。C 语言以 1 代表“真”，以“0”代表“假”。例如， $a = 3$, $b = 2$, $c = 1$ ，则：

关系表达式“ $a > b$ ”的值为“真”，表达式的值为 1。

关系表达式“ $(a > b) == c$ ”的值为“真”（因为 $a > b$ 的值为 1，等于 c 的值），表达式的值为 1。

关系表达式“ $b + c < a$ ”的值为“假”，表达式的值为 0。

如果有以下赋值表达式：

$d = a > b$ ，则 d 的值为 1。

$f = a > b > c$ ，则 f 的值为 0（因为“ $>$ ”运算符是自左至右的结合方向，先执行“ $a > b$ ”得值为 1，再执行关系运算“ $1 > c$ ”，得值 0，赋给 f ）。

3. 逻辑运算符和逻辑表达式

(1) 逻辑运算符及其优先次序

C 语言提供三种逻辑运算符：

① $\&\&$ ，逻辑与（相当于其他语言中的 AND）

② $\|$ ，逻辑或（相当于其他语言中的 OR）

③ $!$ ，逻辑非（相当于其他语言中的 NOT）

“ $\&\&$ ”和“ $\|$ ”是“双目（元）运算符”，它要求有两个运算量（操作数），如 $(a > b) \&\& (x > y)$, $(a > b) \| (x > y)$ 。“ $!$ ”是“一目（元）运算符”，只要求有一个运算量，如 $(a > b)$ 。

逻辑运算举例如下：

$a \&\& b$ 若 a 、 b 为真，则 $a \&\& b$ 为真。

$a||b$ 若 a 、 b 之一为真，则 $a||b$ 为真。

$!a$ 若 a 为真，则 $!a$ 为假。

在一个逻辑表达式中如果包含多个逻辑运算符，如

$!a \&\& b || x > y \&\& c$

则按以下的优先次序进行运算：

1) $!(非) \rightarrow \&\&(与) \rightarrow (或)$ ，即“ $!$ ”为三者中最高的。

2) 逻辑运算符中的“ $\&\&$ ”和“ $||$ ”低于关系运算符，“ $!$ ”高于算术运算符。逻辑运算符优先级如图 3-5 所示。

例如：

$(a > b) \&\& (x > y)$ 可写成 $a > b \&\& x > y$

$(a = b) || (x = y)$ 可写成 $a = b || x = y$

$(!a) || (a > b)$ 可写成 $!a || a > b$

(2) 逻辑表达式

用逻辑运算符将关系表达式或逻辑量连接起来的式子就是逻辑表达式。如前所述，逻辑表达式的值应该是一个逻辑量“真”或“假”。C 语言编译系统在给出逻辑运算结果时，以数值 1 代表“真”，以 0 代表“假”，但在判断一个量是否为“真”时，以 0 代表“假”，以非 0 代表“真”。即将一个非零的数值认作为“真”。例如：

① 若 $a=4$ ，则 $!a$ 的值为 0。因为 a 的值为非 0，被认作“真”，对它进行“非”运算，得“假”，“假”以 0 代表。

② 若 $a=4$ ， $b=5$ ，则 $a \&\& b$ 的值为 1。因为 a 和 b 均为非 0，被认为是“真”，因此 $a \&\& b$ 的值也为“真”，值为 1。

③ a ， b 值同前， $a||b$ 的值为 1。

④ a ， b 值同前， $!a||b$ 的值为 1。

⑤ $4 \&\& 0 || 2$ 的值为 1。

通过这几个例子可以看出，由系统给出的逻辑运算结果不是 0 就是 1，不可能是其他数值。而在逻辑表达式中作为参加逻辑运算的运算对象（操作数），可以是 0（“假”）或任何非 0 的数值（按“真”对待）。如果在一个表达式中不同位置上出现数值，应区分哪些是作为数值运算或关系运算的对象，哪些作为逻辑运算的对象。

在逻辑表达式的求解中，并不是所有的逻辑运算符都被执行，只是在必须执行下一个逻辑运算符才能求出表达式的解时，才执行该运算符。例如：

1) $a \&\& b \&\& c$ 只有 a 为“真”（非 0）时，才需要判别 b 的值，只有 a 和 b 都为“真”的情况下才需要判别 c 的值。只要 a 为“假”，就不必判别 b 和 c （此时整个表达式已确定为“假”）。如果 a 为“真”， b 为“假”，不判别 c 。判别过程如图 3-6 所示。

2) $a||b||c$ 只要 a 为“真”（非 0），就不必判断 b 和 c ；只有 a 为“假”，才判别 b ； a 和 b 都为“假”才判别 c 。判别过程如图 3-7 所示。

也就是说，对“ $\&\&$ ”运算符来说，只有 $a \neq 0$ ，才继续进行右面的运算。对运算符 $||$ 来说，只有 $a=0$ ，才继续进行其右面的运算。因此，如果有下面的逻辑表达式：

$(m=a > b) \&\& (n=c > d)$

当 $a=1$ ， $b=2$ ， $c=3$ ， $d=4$ ， m 和 n 的原值为 1 时，由于“ $a > b$ ”的值为 0，因此 $m=0$ ，而

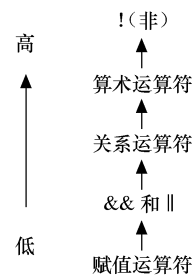


图 3-5 运算符优先级

“ $n=c>d$ ”不被执行，因此 n 的值不是 0 而仍保持原值 1。这点请读者注意。

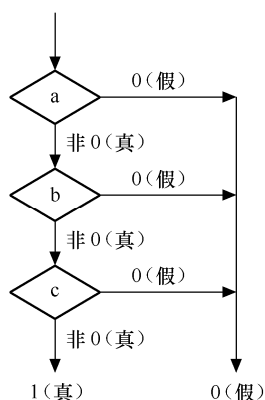


图 3-6 $a \& b \& c$ 的判别过程

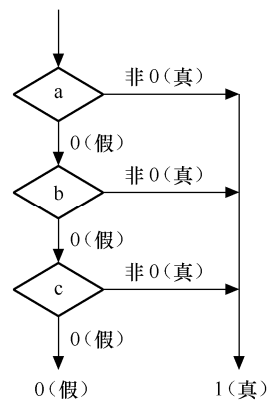


图 3-7 $a||b||c$ 的判别过程

熟练掌握 C 语言的关系运算符和逻辑运算符后，可以巧妙地用一个逻辑表达式来表示一个复杂的条件。

4. 位运算符和位运算

(1) 按位与运算符 (&)

参加运算的两个数据，按二进位进行“与”运算。如果两个相应的二进位都为 1，则该位的结果值为 1，否则为 0。即

$$0 \& 0 = 0; \quad 0 \& 1 = 0; \quad 1 \& 0 = 0; \quad 1 \& 1 = 1;$$

例如：3 & 5 并不等于 8，应该是按位与。

$$\begin{array}{r} 00000011 \text{ (十进制数 3)} \\ \& \quad 00000101 \text{ (十进制数 5)} \\ \hline 00000001 \text{ (十进制数 1)} \end{array}$$

因此，3 & 5 的值为 1。如果参加 & 运算的是负数（如 -3 & -5），则以补码形式表示为二进制数，然后按位进行“与”运算。

(2) 按位或运算符 (|)

两个相应的二进位中只要有一个为 1，该位的结果值为 1。即

$$0|0=0; \quad 0|1=1; \quad 1|0=1; \quad 1|1=1。$$

例如：060|017

将八进制数 60 与八进制数 17 进行按位或运算。

$$\begin{array}{r} 00110000 \text{ (八进制数 60)} \\ | \quad 00001111 \text{ (八进制数 17)} \\ \hline 00111111 \text{ (八进制数 77)} \end{array}$$

低 4 位全为 1。如果想使一个数 a 的低 4 位改为 1，只需将 a 与 017 进行按位或运算即可。

按位或运算常用来对一个数据的某些位定值为 1。例如： a 是一个整数（16 位），有表达式 $a|0377$ ，则低 8 位全置为 1，高 8 位保留原值。

(3) 异或运算符 (^)

异或运算符 ^ 也称 XOR 运算符。它的规则是若参加运算的两个二进位同号，则结果为 0 “假”；异号则为 1 “真”。即

$0 \wedge 0 = 0$; $0 \wedge 1 = 1$; $1 \wedge 0 = 1$; $1 \wedge 1 = 0$;

例如: $57 \wedge 42$

```

      00111001 (十进制数 57)
    ^ 00101010 (十进制数 42)
    -----
      00010011 (十进制数 19)
  
```

即 $071 \wedge 052$, 结果为 023 (八进制数)。

异或的意思是判断两个相应的位值是否为“异”，为“异”（值不同）就取“真”（1），否则为“假”（0）。

(4) 取反运算符 (~)

~ 是一个单目（元）运算符，用来对一个二进制数按位取反，即将 0 变 1，1 变 0。例如: ~025 是对八进制数 25（即二进制数 00010101）按位求反。

```

    ~ 00000000010101 (八进制数 025)
    -----
    11111111101010 (八进制数 0177752)
  
```

结果即八进制数 177752。因此，~025 的值为八进制数 177752，不要以为 ~025 的值是 -025。

(5) 左移运算符 (<<)

用来将一个数的各二进制全部左移若干位。例如:

$a = a << 2$

将 a 的二进制数左移 2 位，右补 0。若 $a=15$ ，即二进制数 00001111，左移 2 位得 00111100，即十进制数 60（为简单起见，用 8 位二进制数表示十进制数 15，如果用 16 位，结果是一样的）。

高位左移后溢出，舍弃不起作用。

左移 1 位相当于该数乘以 2，左移 2 位相当于该数乘以 $2^2=4$ 。上面举的例子 $15 << 2 = 60$ ，即乘了 4。但此结论只适用于该数左移时被溢出舍弃的高位中不包含 1 的情况。例如，假设以一个字节（8 位）存一个整数，若 a 为无符号整型变量，则 $a=64$ 时，左移一位时溢出的是 0，而左移 2 位时，溢出的高位中包含 1。左移比乘法运算快得多，有些 C 编译程序自动将乘 2 的运算用左移一位来实现，将乘 2^n 的幂运算处理为左移 n 位。

(6) 右移运算符 (>>)

$a >> 2$ 表示将 a 的各二进制右移 2 位。移到右端的低位被舍弃，对无符号数，高位补 0。例如 $a=017$ 时:

a 为二进制数 00001111， $a >> 2$ 为 00000011|11 后二位舍弃。

右移一位相当于除以 2，右移 n 位相当于除以 2^n 。在右移时，需要注意符号位问题。对无符号数，右移时左边高位移入 0。对于有符号的值，如果原来符号位为 0（该数为正），则左边也是移入 0，如同上例表示的那样。如果符号位原来为 1（即负数），则左边移入 0 还是 1，要取决于所用的计算机系统。有的系统移入 0，有的移入 1。移入 0 的称为“逻辑右移”，即简单右移。移入 1 的称为“算术右移”。例如，a 的值为八进制数 0113755。

a: 1001011111101101

a>>1: 0100101111110110 (逻辑右移时)

a>>1: 1100101111110110 (算术右移时)

3.3 C51 流程控制语句

3.3.1 C 语言程序的基本结构及其流程图

C 语言是一种结构化编程语言。这种结构化语言有一套不允许交叉的程序流程存在的严格规定。结构化语言的基本元素是模块，它是程序的一部分，只有一个入口和一个出口，不允许有偶然的中途插入或以模块的其他路径退出。结构化编程语言在没有妥善保护或恢复堆栈和其他相关的寄存器之前，不应随便跳入或跳出一个模块。因此使用这种结构化语言进行编程，当要退出中断时，堆栈不会因为程序使用了任何可以接受的命令而崩溃。

结构化程序由若干模块组成，每个模块中包含着若干个基本结构，而每个基本结构中可以有若干条语句。

归纳起来，C 语言有三种基本结构：

- (1) 顺序结构；
- (2) 选择结构；
- (3) 循环结构。

1. 顺序结构及其流程图

顺序结构是一种最基本，最简单的编程结构。在这种结构中，程序由低地址向高地址顺序执行指令代码。顺序结构流程图如图 3-8 所示，程序先执行 A 操作，再执行 B 操作，两者是顺序执行的关系。

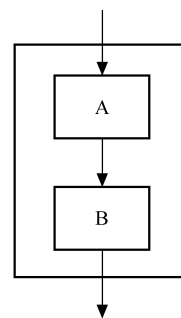


图 3-8 顺序结构流程图

2. 选择结构及其流程图

如果计算机只能做像顺序结构那样简单的基本操作，它的用途十分有限。计算机功能强大的原因就在于它具有决策或者说具有选择能力。例如：

- (1) 依靠条件选择开关，打开或关闭水泵。
- (2) 如果上面的这种操作重复执行了 22 次的话，那么继续执行下面另一个操作。
- (3) 连续监测一个信号，这个信号指示语言芯片可以接收下一个字的代码。

以上这些都是人们通常要求计算机作出选择（决策）的例子。依靠选择（决策）测试，程序可以进行循环（自己返回）或分支操作（在几个可能的方向上选择一个进入方向）。

在选择结构中，程序首先对一个条件语句进行测试。当条件为“真”（True）时，执行一个方向上的程序流程；当条

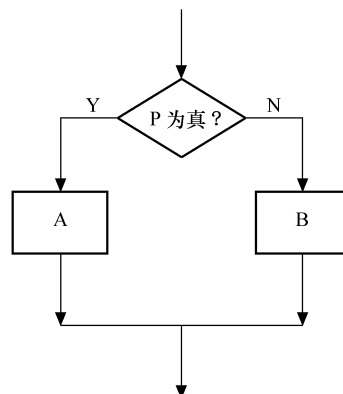


图 3-9 选择程序结构图

件为“假”(False)时,执行另一个方向上的程序流程。如图3-9所示, p 代表一个条件,当 p 条件成立“为真”时,执行A操作,否则执行B操作。但两者只能选择其一,两个方向上的程序流程最终将汇集到一起从一个出口中退出。

常见的选择语句是if语句。

由选择结构可以派生出另一种基本结构,即多分支结构。在多分支结构中又分为串行多分支和并行多分支两种情况。

(1) 串行多分支及其流程图

串行多分支结构流程图如图3-10所示,在串行多分支结构中,以单选择结构中的某一分支方向作为串行多分支方向(假设,以条件为“真”为串行方向)继续进行选择结构的操作。若条件为“假”时,则执行另外的操作。最终程序在若干种选择之中选出一种操作来执行,并从一个共用的出口退出。

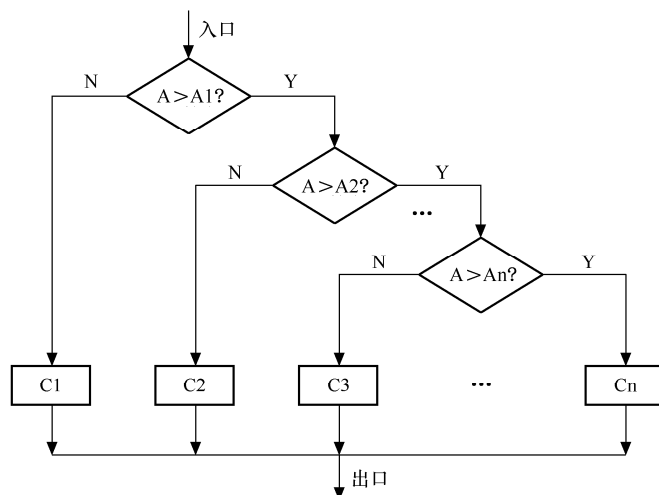


图 3-10 串行多分支结构流程图

这种串行多分支结构由若干条if语句嵌套构成。

(2) 并行多分支结构及其流程图

并行多分支结构流程图如图3-11所示,在并行多分支结构中,根据K值的不同选择A1, A2, ..., An等不同操作中的一种来执行。

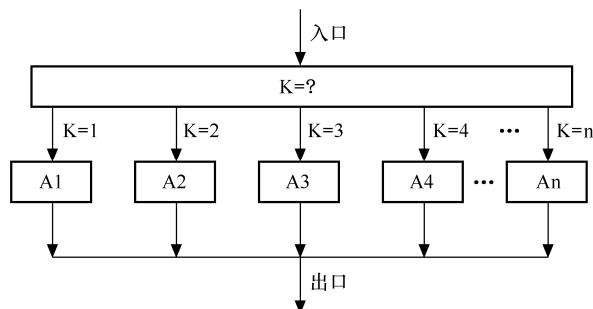


图 3-11 并行多分支结构流程图

用于构成并行多分支结构的语句是 switch 语句。

3. 循环结构及其流程图

所有的分支结构都使程序流程一直向前执行（除非使用了某种 goto 语句），而使用循环结构则可使分支流程重复地进行。循环结构分成“当”（while）型循环结构和“直到”（do...while）型循环结构两种。

（1）“当”（while）型循环结构及其流程图

while 循环结构如图 3-12 所示，在这种结构中，当判断条件 p 成立（为“真”）时，反复执行操作 A，直到 p 条件不成立（为“假”）时，才停止循环。

（2）“直到”（do...while）型循环结构及其流程图

do...while 循环结构如图 3-13 所示，这种结构中，先执行操作 A，再判断条件 p，若 p 成立（为“真”）则再次执行 A，反复执行直到 p 为假为止。

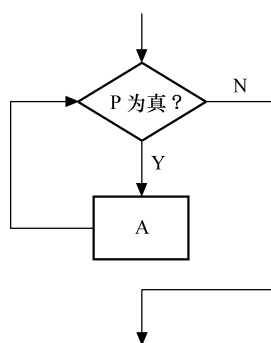


图 3-12 while 循环结构

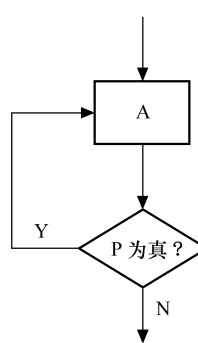


图 3-13 do while 循环结构

构成循环结构的常见语句主要有：while，do while 和 for 等。

可以证明，由以上基本结构组成的程序，能够处理任何复杂的问题。换句话说，任何复杂的程序都是由以上三种基本结构组成的。

在进行编程之前，应该先画出程序的流程图。程序的流程图不是一个程序的细节，而只是这个程序关于解决问题的方法概述。一般来说，一个程序模块的代码长度应该由不多于一页的程序段组成，所以相应地的程序流程图也应画在一张单页纸上。一幅长达数页，含有许多路径的程序流程图是令人望而生畏的，虽然从技术的角度看它们是正确无误的，但这样的程序流程图是不能帮助人来弄懂这段程序究竟是在干什么。为了使程序的流程图便于理解，当它们复杂到不能放在一页中的时候，可以用把它们分解成若干段子程序的方法对它们进行简化。这些子程序的细节可以在其他页上的程序流程图中加以适当的展开。这样主流程图总是展示程序各个段的概貌，如果需要了解各分段程序解决问题的细节，请查阅各分段程序流程图及其相应的子程序。

所有流程图应使用功能命名而不应以特定

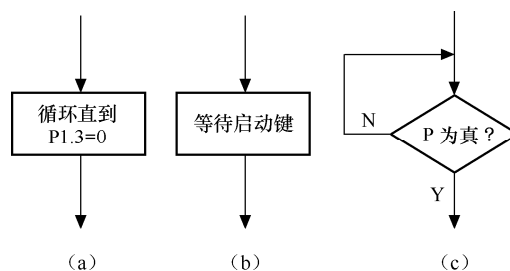


图 3-14 流程图表示法

的变量名作参考，例如，等待某键按下启动，可采用图 3-14 所示的几种表示方法。

其中，b 表示方法最好，c 表示方法次之。

3.3.2 选择语句

1. 选择语句 if

if 语句是用来判定所给定的条件是否满足，然后根据判定的结果决定执行给出的两种操作之一。

(1) if 语句的 3 种形式

C 语言提供了 3 种形式的 if 语句。

① if(表达式) 语句

例如：

```
if(x>y) printf("%d", x);
```

② if(表达式)语句 1 else 语句 2

例如：

```
if(x>y) printf("%d", x);
```

```
else printf("%d", y);
```

③ if(表达式 1) 语句 1

else if(表达式 2) 语句 2

else if(表达式 3) 语句 3

...

else if(表达式 m) 语句 m

else 语句 n

例如：

```
if(number>500) cost=0.15;
```

```
else if(number>300) cost=0.10;
```

```
else if(number>100) cost=0.075;
```

```
else if(number>50) cost=0.05;
```

```
else cost=0;
```

说明：

① 3 种形式的 if 语句中 if 后面都有“表达式”，一般为逻辑表达式或关系表达式。在执行 if 语句时先对表达式求解，若表达式的值为 0，按“假”处理，若表达式的值为非 0，按“真”处理，然后执行指定的语句。例如 if 语句：

```
if(3) i=5;
```

是合法的，因为表达式的值为 3，按“真”处理。由此可见，表达式的类型不限于逻辑表达式，可以是任意的数值类型（包括整型、实型、字符型、指针型数据）。例如，下面的 if 语句也是合法的。

```
if('a') b='a';
```

执行结果：b 等于 ‘a’ 的 ASCII 码值 97。

② 第 2 种、第 3 种形式的 if 语句中，在每个 else 前面有一分号，整个语句结束处有一分号。例如：

```
if(x>0)
    a=x;
else
    a=-x;
```

执行语句后各有一个分号，这是由于分号是 C 语句中不可缺少的部分，这个分号是 if 语句中的内嵌语句所要求的。如果无此分号，则出现语法错误。但应注意，不要误认为上面是两个语句（if 语句和 else 语句），它们都属于同一个 if 语句。else 子句不能作为语句单独使用，它必须是 if 语句的一部分，与 if 配对使用。

③ 在 if 和 else 后面可以只含一个内嵌的操作语句（如上例），也可以有多个操作语句，此时用花括号 “{ }” 将几个语句括起来成为一个复合语句。例如：

```
if(a+b>c && b+c>a && c+a>b)
{
    s=0.5*(a+b+c);
    area=sqrt(S*(S-a)*(S-b)*(S-c));
}
else s=1.5*(a+b+c);
```

注意在第 5 行的花括号 “}” 外面不需要再加分号。因为 { } 内是一个完整的复合语句，不需另附加分号。

(2) if 语句的嵌套

在 if 语句中又包含一个或多个 if 语句称为 if 语句的嵌套。一般形式如下：

```
if()
    if()语句 1
    else 语句 2
else
    if()语句 3
    else 语句 4
```

应当注意 if 与 else 的配对关系，else 总是与它上面的最近的 if 配对。假如写成：

```
if()
    if()语句 1
else
    if()语句 2
    else 语句 3 内嵌 if
```

编程序者把 else 写在与第一个 if（外层 if）同一列上，希望 else 与第一个 if 对应，但实际上 else 是与第二个 if 配对，因为它们相距最近。因此最好使内嵌 if 语句也包含 else 部分，这样 if 的数目和 else 的数目相同，从内层到外层一一对应，不致出错。

如果 if 与 else 的数目不一样，为实现程序设计者的意图，可以加花括弧来确定配对关系。

例如：

```
if()
```

```
{ if()语句 1}
```

```
else 语句 2(内嵌 if)
```

这时 {} 限定了内嵌 if 语句的范围，因此 else 与第一个 if 配对。

【例 3-1】 有一函数： $y=-1(x<0)$ ， $y=0(x=0)$ ， $y=1(x>0)$ ，编一程序，输入一个 x 值，输出相应的 y 值。

可以先写出算法：

输入 x

若 $x < 0$ ， $y = -1$ ；

否则：

若 $x = 0$ ， $y = 0$ ；若 $x > 0$ ， $y = 1$ ；

输出 y 。

2. switch 语句

switch 语句是多分支选择语句，用来实现如图 3-11 所示的多分支选择结构。if 语句只有两个分支可供选择，而实际问题中常常需要用到多分支的选择。例如，学生成绩分类（90 分以上为 A 等，80~89 分为 B 等，70~79 分为 C 等.....）；人口统计分类（按年龄分为老、中、青、少、儿童）；工资统计分类；银行存款分类.....。

当然这些都可以用嵌套的 if 语句来处理，但如果分支较多，则嵌套的 if 语句层数多，程序冗长而且可读性降低。C 语言提供 switch 语句直接处理多分支选择，它相当于 Pascal 语言中的 case 语句。它的一般形式如下：

```
switch(表达式)
```

```
{ case 常量表达式 1: 语句 1
```

```
case 常量表达式 2: 语句 2
```

```
...
```

```
case 常量表达式 n: 语句 n
```

```
default: 语句 n+1
```

```
}
```

说明：

(1) ANSI 标准允许 switch 后面括弧内的“表达式”为任何类型。

(2) 当表达式的值与某一个 case 后面的常量表达式的值相等时，就执行此 case 后面的语句；若所有 case 中常量表达式的值都没有与表达式的值匹配，就执行 default 后面的语句。

(3) 每一个 case 的常量表达式的值必须互不相同，否则就会出现互相矛盾的现象（对表达式的同一个值，有两种或多种执行方案）。

(4) 各个 case 和 default 的出现次序不影响执行结果。

(5) 执行完一个 case 后面的语句后，流程控制转移到下一个 case 继续执行。“case 常量表达式”只是起语句标号作用，并不是在该处进行条件判断。在执行 switch 语句时，根据 switch 后面表达式的值找到匹配的入口标号，就从此标号开始执行下去，不再进行判断。

因此,应该在执行一个 case 分支后,使流程跳出 switch 结构,即终止 switch 语句的执行。可以采用一个 break 语句来达到此目的。将上面的 switch 结构改写如下:

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

3.3.3 循环语句

在许多实际问题中,需要进行具有规律性的重复操作,例如求累加和,数据块的搬移等。而计算机的基本特性之一就是具有重复执行一组语句的能力,即循环能力。利用这种循环能力,程序员只要编写一个包含重复执行语句的简短程序,就能执行所需的成千上万次的重复操作,几乎所有的实用程序都包含有循环结构。循环结构是结构化程序设计的三种基本结构之一,它和顺序结构、选择结构一起共同作为各种复杂程序的基本构造单元。因此熟练地掌握和运用循环结构的概念是程序设计,尤其是 C 程序设计最基本的要求。

作为构成循环结构的循环语句,一般是由循环体及循环终止条件两部分组成的。一组被重复执行的语句称为循环体,能否继续重复执行下去则取决于循环终止条件。

在 C 语言中用来实现循环的语句有以下 3 种:

- (1) while 语句;
- (2) do-while 语句;
- (3) for 语句。

1. while 语句

while 语句的一般形式为:

```
while(表达式)
{语句/*循环体*/}
```

其中,表达式是 while 循环能否继续的条件,而语句部分则是循环体,是执行重复操作的部分。只要表达式为“真”,就重复执行循环体内的语句;反之,则终止 while 循环。执行循环之外的下一行语句。while 循环语句的语法流程图如图 3-12 所示。

while 循环结构的最大特点在于:其循环条件测试处于循环体的开头,要想执行重复操作首先必须进行循环条件测试,若条件不成立则循环体内的重复操作一次也不能执行。例如:

```
while((P1 & 0x10) == 0)
{ }
```

这个语句的作用是等待来自用户或外部硬件的某些信号的变化。该语句对 C51 的 P1 口的第五位(bit4)进行测试。如果第五位电平为低(<0),则由于循环体无实际操作语句,故继续测试下去(等待);一旦 P1 口的第五位电平变高,则循环终止。

注意：“=”号与“==”号的区别。“=”是赋值运算符，而“==”是测试相等运算符，它只是对该符号两边的表达式进行测试和比较，而不进行赋值。

例如：

```
while((P1&0x10)>0&&y++<=5)
{
    a=1;
    b=45;
    x=P1;
}
```

语句中表达式所列出的测试条件是由两个分测试条件相“与”构成的。

第1个分测试条件是 $(P1 \& 10H) > 0$ ，它的作用是对 8031 P1 口的第 5 管脚 (P1.4) 进行测试，如果它等于 1 则条件为“真”(1)，否则为“假”(0)。在本句中使用大于 0 而不是等于 10H 来进行测试较为安全。因为使用 $= 10H$ 进行测试 $(P1 \& 10H)$ ，会将屏蔽数值和相等测试混淆，这样将会潜伏着某种隐患。

第2个分测试条件是 $y++ \leq 5$ ，它的含义是当 $y < 5$ 时条件为“真”(1)，否则为“假”(0)。 $y++ \leq 5$ 中先测试 $y < 5$ ，然后再执行 y 自加 1 的运算。

把上面两个分测试条件相“与”，则这个 while 循环的功能是：只有当 P1.4 位电平为高，并持续一段时间（由 $y++ \leq 5$ 来控制），则执行花括号 {} 中语句： $a=1$ ， $b=45$ ， $x=P1$ 。否则退出此 while 循环体，执行下一条语句。

注意：

(1) 在 while 循环体内有 3 条语句，应使用花括号 {} 括起来，表示这是一个语句块。当循环体内只有一条语句时，可以不使用花括号，但此时使用花括号将使程序更加安全可靠，特别是在进行 while 循环的多重嵌套时，使用花括号来分隔循环体将提高程序的可读性和可靠性。

(2) 在 while 循环体中，应有使循环趋向于结束的语句。在本例中，当 P1.4 等于 0 或 y 大于 5 时，while 循环体结束。若无此种语句，则循环将无休止地继续下去。

2. do-while 语句

上节讨论的 while 循环语句是在循环执行之前检测循环结束条件的。如果条件不成立，则该循环不会被执行。在程序设计中，有时需要在循环体的结尾处，而不是在循环体的开始处检测循环结束条件。do-while 循环语句可以满足这种要求。

do-while 语句的格式为：

```
do
{语句; }/*循环体*/
while(表达式);
```

do-while 循环语句的执行过程如下：首先执行循环体语句，然后执行圆括号中的表达式。如果表达式的结果为“真”(1)，则循环继续，并再一次执行循环语句。只有当表达式的结果为假(0)时，循环才会终止，并以正常方式执行程序后面的语句。

do-while 循环语句把 while 循环语句作了移位，即把循环条件测试的位置从起始处移至

循环的结尾处。该语句大多用于执行至少一次以上的循环的情况。do-while 的程序流程图如图 3-13 所示。

【例 3-2】 do-while 循环例程。

```
int sum=0, i;
```

```
do
```

```
{
```

```
    sum+=i;
```

```
    i++;
```

```
}
```

```
while(i<=10);
```

程序的执行结果：变量 sum 值为 55。

再如：

```
do
```

```
{
```

```
    a=1;
```

```
    b=45;
```

```
    x=pl;
```

```
}
```

```
while(x>0&& y++=5);
```

该程序先执行循环体内语句，对 a, b, x 变量赋值，然后再进行循环条件测试：
while(x>0&& y++=5)。

3. for 循环语句

在 C 语言中 for 循环语句是循环语句中最为灵活也是最为复杂的一种。它不仅可以用于循环次数已经确定的情况，而且可以用于循环次数不确定但已经给出循环条件的情况。它既可以包含一个索引计数变量，也可以包含任何一种表达式。除了被重复的循环指令体外，表达式模块由 3 个部分组成。第 1 部分是初始化表达式，对 C 语言而言，任何表达式在开始执行时都应该做一次初始化。第 2 部分是对结束循环进行测试，对 C 语言而言，它可以是任何一种测试，一旦测试为假，就会结束循环。第 3 部分是尺度增量，对 C 语言而言，任何指定的操作、或在测试之后在进入之前将要执行的表达式都可以放在这里。如果为了有效和保密，可以使用这种结构让 C 去做别人几乎不能理解的事情。

for 循环语句的一般形式为：

```
for(表达式 1; 表达式 2; 表达式 3)
```

```
{语句; }/*循环体*/
```

for 循环的语句执行过程如下：

(1) 先对表达式 1 赋初值，进行初始化。

(2) 判断表达式 2 是否满足给定的循环条件：若满足，则执行循环体内语句，然后执行下面第 3 步；若不满足循环条件，则结束循环，转到第 5 步。

(3) 若表达式 2 为真，则在执行指定的循环语句后，求解表达式 3。

(4) 回到第2步继续执行。

(5) 退出for循环，执行下面一条语句。

【例 3-3】 for 循环例程。

```
int i, sum;
sum=0;
for(i=0; i<=10; i++)
sum+=i;
运算结果: sum 值为 55。
```

在程序中，for 循环表达式 1 是 $i=0$ ，其作用是给 i 赋初值。表达式 2 是 $i\leq 10$ ，其作用是对循环条件进行测试，当 i 值小于等于 10，表达式 2 为“真”，则执行循环体内语句 $\text{sum}+=i$ （即 $\text{sum}=\text{sum}+i$ ），然后执行表达式 3 ($i++$)，进入下一轮循环。若 i 值大于 10 时，表达式 2 为“假”，则终止循环。

在上面程序中，循环体为单一语句。如果在循环中执行的不是一条程序语句，而是一组程序语句，则应当用一对花括号 $\{\}$ 将这些语句括起来，这样系统才会把这一组语句当作一个循环实体来看待。否则，除了第一句外，系统会将其余的语句当作循环之外的语句。一般来说，C 程序中凡是允许一个语句出现的地方，都能使用一个语句块，但该语句块一定要包括在一对花括号 $\{\}$ 中。

【例 3-4】 for 循环例程。

```
for(y=0; y<=99; y=y+3)
{
    delay(33);
    px=~px;
}

for(da=start; status==busy; leds=~leds)
{
    delay(33);
}
```

本例中，for 循环的表达式对预先用 `#define` 语句指定的口和位进行赋值操作。这些口和位可以启动 A/D 转换器 ($\text{da}=\text{start}$)，并等待转换完成 ($\text{status}=\text{busy}$)，也可以用延时的方法按预设的速率触发脉冲信号 ($\text{leds}=\sim\text{leds}$)。在这个 for 循环中，除了延时之外，似乎没有做别的事情，而实际上则恰恰相反。对 A/D 转换器来说，循环之后的下一个动作应该是完整地读出转换结果。

下面，对 for 循环语句的几种特例情况进行说明。

(1) for 语句中的小括号内的三个表达式全部为空。

```
例如: for(;;)
{.....}/*循环体*/
```

在小括号内只有两个分号，无表达式。这意味着没有设初值，无判断条件，循环变量为增值，这将导致一个无限循环。一般在编写 C51 单片机监控程序需要无限循环时，就采用这

种形式的 for 循环语句。

(2) for 语句三个表达式中，有一个或两个缺省。

① 表达式 1 缺省

例如：for(; i<=100; i++)sum=sum+i;

表示不对 i 设初值。

② 表达式 2 缺省

例如：for(i=1; ; i++) sum=sum+i;

表示不判断循环条件，认为表达式始终为“真”。循环将无休止地进行下去。它相当于如下程序：

```
i=1;
while(1)
{
    sum=sum+i;
    i++;
}
```

③ 表达式 1, 3 省略

例如：for(; i<=100;)

```
{
    sum=sum+i;
    i++;
}
```

它等效于如下程序：

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

(3) 没有循环体的 for 语句。

例如：int sum=2000;

for(t=0; t<sum; t++);

该例在程序中起延时作用。

循环的基本用途之一是用嵌套循环产生时间延迟，执行的指令消磨一段已知的时间。这种延时方式是依靠一定数量的时钟周期来计时的，所以延时依赖于晶振的振荡频率。C51 单片机的数据手册中列出了每一条机器指令所需要的时钟周期数，使用 12MHz 的晶振，12 个振荡周期的指令花费 1 μ s，而 24 个振荡周期指令花费 2 μ s。所有 3 字节指令每条指令用 2 μ s，所有分支指令每一条用 2 μ s，所有 1 到 2 字节逻辑和算术指令每一条用 1 μ s，而寄存器到寄存器传送 (mov) 用 2 μ s。mul 和 div 指令是唯一的例外，每条指令要花费 4 μ s 时间。当需要执行其他操作时，用软件编程的方法来获得延时效率是很低的。软件延时使控制器在延时循环时接收不到其他的输入，解决这个问题的是使用中断。

【例 3-5】 下面是一个一次延时 1ms 的延时程序。这个例子通过传递一个 50 的数值可以产生约 50 000 μ s 即 50 ms 延时。

```
void msec(unsigned int x)
{
    unsigned char j;
    while(x--)
    {
        for(j=0; j<125; j++)
        {; }
    }
}
```

这个程序可以用整型值产生较长的延时。根据汇编代码进行的分析表明，用 j 进行的内部循环大约延时 8 μ s，程序编写得近似正确，但并不精确。不同的编译器会产生不同的延时，因此 j 的上限值 125 应根据实验进行补偿调整。

3.4 C51 构造数据类型

前面讲述的字符型（char）、整型（int）和浮点型（float）等数据，它们都属于基本数据类型。C 语言还提供了一些扩展的数据类型，它们是对基本数据类型的扩展称为构造数据类型。这些按一定规则构成的数据类型有数组、结构、指针、共用体、枚举等。

3.4.1 数组

C 语言具有定义一组有序数据项的能力。这组有序的数据即数组。数组是一组具有固定数目和相同类型成分分量的有序集合。其成分分量的类型为该数组的基本类型。例如整型变量的有序集合称为整型数组，字符型变量的有序集合称为字符型数组。这些整型或字符型变量是各自所属数组的成分分量，称为数组元素。

构成一个数组的各元素必须是同一类型的变量，而不允许在同一数组中出现不同类型的变量。

数组数据是用同一个名字的不同下标访问的，数组的下标放在方括号中，它是从 0 开始到 n 的一组有序整数。例如数组 a[i]，当 i=0, 1, 2, ..., n 时，a[0], a[1], ..., a[n] 分别是数组 a[i] 的元素（或成员）。数组有一维、二维、三维、多维数组之分，常用的有一维、二维数组和字符数组。

1. 一维数组

（1）一维数组的定义

一维数组的定义方式为：

类型说明符 数组名[常量表达式];

例如：

```
int a[10];
```

它表示数组名为 a，此数组有 10 个元素。

说明：

① 数组名定名规则和变量名相同，遵循标识符定名规则。

② 数组名后是用方括弧括起来的常量表达式，不能用圆括弧。

③ 常量表达式表示元素的个数，即数组长度。例如，在 a[10] 中，10 表示 a 数组有 10 个元素，下标从 0 开始，这 10 个元素是 a[0]，a[1]，a[2]，a[3]，a[4]，a[5]，a[6]，a[7]，a[8]，a[9]。注意：不能使用数组元素 a[10]。

④ 常量表达式中可以包括常量和符号常量，不能包含变量。也就是说，C 语言不允许对数组的大小作动态定义，即数组的大小不依赖于程序运行过程中变量的值。例如，下面这样定义数组是不行的：

```
int n;
```

```
int a[n];
```

(2) 一维数组的初始化

对数组元素的初始化可以用以下方法实现：

① 在定义数组时对数组元素赋以初值。例如：

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

即将数组元素的初值依次放在一对花括弧内。经过上面的定义和初始化之后，a[0]=0，a[1]=1，a[2]=2，a[3]=3，a[4]=4，a[5]=5，a[6]=6，a[7]=7，a[8]=8，a[9]=9。

② 可以只给一部分元素赋值。例如：

```
int a[10]={0, 1, 2, 3, 4};
```

定义 a 数组有 10 个元素，但花括弧内只提供 5 个初值，这表示只给前面 5 个元素赋初值，后 5 个元素值为 0。

③ 如果想使一个数组中全部元素值为 0，可以写成：

```
int a[10]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

但不能写成：

```
int a[10]={0*10};
```

④ 在对全部数组元素赋初值时，可以不指定数组长度。例如：

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

可以写成：

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

在第二种写法中，花括弧中有 5 个数，系统就会据此自动定义 a 数组的长度为 5。

但若被定义的数组长度与提供初值的个数不相同，则数组长度不能省略。例如，想定义数组长度为 10，就不能省略数组长度的定义，而必须写成：

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

只初始化前 5 个元素，后 5 个元素为 0。

(3) 一维数组的引用

数组必须先定义，然后使用。C 语言规定只能逐个引用数组元素而不能一次引用整个

数组。

数组元素的表示形式为：数组名[下标]，下标可以是整型常量或整型表达式。例如：

$a[0]=a[5]+a[7]-a[2*3]$

【例 3-6】 由开关控制指示灯的程序举例。

由 C51 控制指示灯的电路，如图 3-15 所示。此电路从与 P0 口相连的 8 个开关读入控制量信息，并将最近读入的十次信息存储在 10 字节数组 `int array[10]` 中，并把最后一次读入的信息显示在与 P2 口相连的 8 位 LED 指示灯上。每次读入显示信息间隔 100ms，由延时程序 `msec(100)` 完成。

相应的 C 语言程序流程图如图 3-16 所示。实际的 C 语言程序如下：

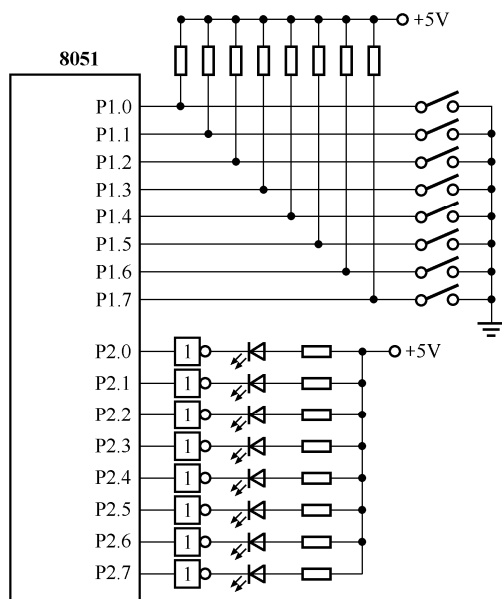


图 3-15 开关控制指示灯电路图

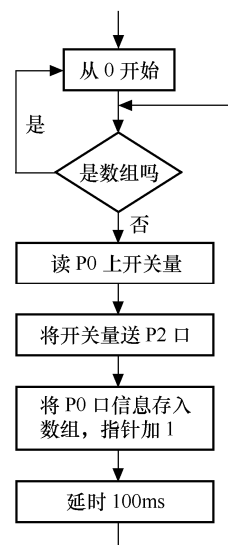


图 3-16 开关控制指示灯流程图

```

#include<reg51.h>          /*item 1*/
void msec(unsigned int):    /*item 2*/
void main()
{
    unsigned char array[10];
    unsigned char i;
    while(1)                /*item 3*/
    {
        for(i=0; i<=9; i++) /*item 4*/
        {
            array[i]=P2=P0;  /*item 5*/
            msec(100);        /*item 2*/
        }
    }
}
  
```

```

    }
}
void msec(unsigned int x)
{
    unsigned char j;
    while((x--)!=0)
    {
        for(j=0, j<125; j++)
            {; }
    }
}

```

item1 在本例中为了定义两个 I/O 口的名称必须使用#include 语句引用 reg51.h 头文件, 该文件定义了 C51 内部所有的寄存器和 I/O 口。

item2 这里显示了一个 ms 级延时程序的原型。这意味着实际函数存在于其他模块中, 而这里只是简单地告诉编译器, 其细节必须调用实际的子程序。

item3 在 C 语言中, 可以用 while (1) 或 for (;;) 表示一个无结束的无限循环。

item4 的 for 程序块显示, i 从 0 开始 (在初始化时这个 0 使 i 指向数组的第一个元素), 当 i 不大于 9 时, 程序一直保持循环 (测试条件是小于或等于 9)。在循环的底部, i 每经过一次循环就加 1。

item5 的 array[i]=P2=P0 中两个等号是有效的, 因为等号是从右向左赋值的。也就是说, 在 P2 口将其值送到数组 array[i]之前, P0 口已将其值赋给了 P2 口。

2. 二维数组

(1) 二维数组的定义

二维数组定义的一般形式为:

类型说明符 数组名[常量表达式][常量表达式];

例如: float a[3][4], b[5][10];

定义 a 为 3×4 (3 行 4 列) 的数组, b 为 5×10 (5 行 10 列) 的数组。注意不能写成:

float a[3, 4], b[5, 10];

C 语言对二维数组采用这样的定义方式, 可以把二维数组看作是一种特殊的一维数组: 它的元素又是一个一维数组。例如, 可以把 a 看作是一个一维数组, 它有 3 个元素: a[0]、a[1]、a[2], 每个元素又是一个包含 4 个元素的一维数组。可以把 a[0]、a[1]、a[2]看作是 3 个一维数组的名字。上面定义的二维数组可以理解为定义了 3 个一维数组, 即相当于 float a[0][4], a[1][4], a[2][4]。

此处把 a[0], a[1], a[2]看作一维数组名。C 语言的这种处理方法在数组初始化和用指针表示时显得很方便, 这在以后会体会到。

C 语言中, 二维数组中元素排列的顺序是: 按行存放, 即在内存中先顺序存放第一行的元素, 再存放第二行的元素。

C 允许使用多维数组。有了二维数组的基础, 再掌握多维数组是不困难的。例如, 定义

三维数组的方法是：

```
float a[2][3][4];
```

多维数组元素在内存中的排列顺序：第一维的下标变化最慢，最右边的下标变化最快。

例如，上述三维数组的元素排列顺序为：

```
a[0][0][0]→a[0][0][1]→a[0][0][2]→a[0][0][3]→
a[0][1][0]→a[0][1][1]→a[0][1][2]→a[0][1][3]→
a[0][2][0]→a[0][2][1]→a[0][2][2]→a[0][2][3]→
a[1][0][0]→a[1][0][1]→a[1][0][2]→a[1][0][3]→
a[1][1][0]→a[1][1][1]→a[1][1][2]→a[1][1][3]→
a[1][2][0]→a[1][2][1]→a[1][2][2]→a[1][2][3]
```

(2) 二维数组的初始化

可以用下面的方法对二维数组初始化。

① 分行给二维数组赋初值。例如：

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

这种赋初值方法比较直观，把第1个花括弧内的数据给第1行的元素，第2个花括弧内的数据赋给第2行的元素……即按行赋初值。

② 可以将所有数据写在一个花括弧内，按数组排列的顺序对各元素赋初值。例如：

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

效果与前相同。但第1种方法为好，一行对一行，界限清楚。用第2种方法如果数据多，写成一大片，容易遗漏，也不易检查。

③ 可以对部分元素赋初值，例如：

```
int a[3][4]={{1}, {5}, {9}};
```

它的作用是只对各行第1列的元素赋初值，其余元素值自动为0。赋初值后数组各元素为：

```
1  0  0  0
5  0  0  0
9  0  0  0
```

也可以对各行中的某一元素赋初值，例如：

```
int a[3][4]={{1}, {0, 6}, {0, 0, 11}};
```

初始化后的数组元素如下：

```
1  0  0  0
0  6  0  0
0  0 11  0
```

这种方法对非0元素少时比较方便，不必将所有的0都写出来，只需输入少量数据。也可以只对某几行元素赋初值，例如：

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

数组元素为：

```
1  0  0  0
5  6  0  0
0  0  0  0
```

第 3 行不赋初值。也可以对第 2 行不赋初值，例如：

```
int a[3][4]={ {1}, {}, {9}};
```

④ 如果对全部元素都赋初值（即提供全部初始数据），则定义数组时对第一维的长度可以不指定，但第二维的长度不能省。例如：

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

与下面的定义等价：

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

系统会根据数据总个数分配存储空间，一共 12 个数据，每行 4 列，当然可确定为 3 行。在定义时也可以只对部分元素赋初值而省略第一维的长度，但应分行赋初值。例如：

```
int a[][4]={ {0, 0, 3}, {}, {0, 10}};
```

这样的写法，能通知编译系统数组共有 3 行。数组各元素为：

```
0  0  3  0
0  0  0  0
0 10  0  0
```

可见，C 语言在定义数组和表示数组元素时采用 `a[][]` 这种两个方括弧的方式，对数组初始化时十分有用，它使概念清楚，使用方便，不易出错。

(3) 二维数组的引用

二维数组的元素的表示形式为：

数组名[下标][下标]

例如 `a[2][3]`。下标可以是整型表达式，如 `a[2-1][2*2-1]`。

数组元素可以出现在表达式中，也可以被赋值，例如：

```
b[1][2]=a[2][3]/2
```

在使用数组元素时，应该注意下标值应在已定义的数组大小的范围内。例如：

```
int a[3][4];
```

```
...
```

```
a[3][4]=3;
```

定义 `a` 为 3×4 的数组，它可用的行下标值最大为 2，列下标值最大为 3，采用 `a[3][4]` 超过了数组的范围。

请读者严格区分在定义数组时用的 `a[3][4]` 和引用元素时的 `a[3][4]` 的区别。前者 `a[3][4]` 用来定义数组的维数和各维的大小，后者 `a[3][4]` 中的 3 和 4 是下标值，`a[3][4]` 代表某一个元素。

3. 字符数组

用来存放字符数据的数组是字符数组。字符数组中的一个元素存放一个字符。

(1) 字符数组的定义

定义方法与前面介绍的类似。例如：`char c [10]`，定义 `c` 为字符数组，包含 10 个元素。

(2) 字符数组的初始化

字符数组置初值的最直接的方法是将各字符逐个赋给数组中的各个元素。例如：

```
char a[10]='B', 'E', 'T', '\n', 'J', 'T', 'N', 'G', '\0';
```

定义了一个字符型数组 `a`，有 10 个数组元素，并且将 9 个字符（其中包括一个字符串结束

标志'\0') 分别赋给了 a[0]~a[8], 剩余的 a[9]被系统自动赋予空格字符。其状态如图 3-17 所示。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
B	E	I		J	I	N	G	\0	

图 3-17 a[10]赋值后的状态

C 语言还允许用字符串直接给字符数组置初值, 其方法有以下两种形式:

```
char a[10]={ "BEI JING" };
```

```
char a[10]= "BEI JING".
```

用双引号 “ ” 括起来的一串字符, 称为字符串常量。比如 “Happy”。C 编译器会自动地在字符末尾加上结束符 '\0' (NULL)。

用单引号 ' ' 括起来的字符为字符的 ASCII 码值, 而不是字符串。比如 'a' 表示 a 的 ASCII 码值 97, 而 “a” 表示一个字符串, 它由两个字符 'a' 和 '\0' 组成。

一个字符串可以用一维数组来装入, 但数组的元素数目一定要比字符多一个。以便 C 编译器自动在其后面加入结束符 '\0'。

若干个字符串可以装入一个二维字符数组中, 称为字符数组。数组的第一个下标是字符串的个数, 第二个下标定义每个字符串的长度, 该长度应当比这批字符串中最长的串多一个字符, 用于装入字符串的结束符 '\0'。比如 char a[60][81], 定义了一个二维字符数组 a, 它可容纳 60 个字符串, 每串最长可达 80 个字符。

【例 3-7】 字符数组实例。

```
uchar code msg[ ][17]=
{
    { "This is a test", \n},
    { "message 1", \n},
    { "message 2", \n}
};
```

这是一个二维数组, 第二个下标必须给定, 因为它不能从数据表中得到, 第一个下标可缺省由数据常量表决定。

4. 查表

数组的一个非常有用的用途之一就是查表。

在许多嵌入式控制系统应用中, 人们更愿意采用表格而不是数学公式计算, 特别是那些对于传感器的非线性转换需要进行补偿的场合 (例如水泵的流量传感器的非线性补偿), 使用查表法 (如有必要再加上线性插值法) 将比采用复杂的曲线拟合所需要的数学方法有效得多, 因为表格查找执行起来速度更快, 所用代码较少。表可以事先计算好后装入 EPROM 中, 使用内插法可以增加查表值的精度, 减少表的长度。数组的使用非常适合于这种查表方法。

【例 3-8】 下面举一个将摄氏温度转换成华氏温度的例子来说明这个问题。

```
#define uchar unsigned char
uchar code tempt[]={32, 34, 36, 37, 39, 41}; /*数组, 设置在 EPROM 中, 长度为实际输入的数值数*/
```

```

uchar ftoc(uchar degc)
{
    return tempt[degc];           /*返回华氏温度值*/
}
main()
{
    x=ftoc(5);                   /*得到 5℃ 的华氏温度值*/
}

```

在程序的开始处，`uchar code tempt[]={32, 34, 36, 37, 39, 41}`；定义了一个无符号字符型数组 `tempt[]`，并对其初始化将摄氏温度 0, 1, 2, 3, 4, 5 对应的温度 32, 34, 36, 37, 39, 41 赋予数组 `tempt[]`，类型代码 `code` 指定编译器将此表定位在 EPROM 中。

在主程序 `main()` 中调用函数 `ftoc(uchar degc)` 从 `tempt[]` 数组中查表获取相应的温度转换值。`x=ftoc(5)`；执行后，`x` 的结果为 5℃ 相应的华氏温度 41°。

5. 数组与存储空间

当程序中设定了一个数组时，C 编译器就会在系统的存储空间中开辟一个区域用于存放该数组的内容。数组就包含在这个由连续存储单元组成的存储体内。对字符数组而言，它占据了内存中一串连续的字节位置。对整型（`int`）数组而言，它们将在存储区中占据一串连续的字节对的位置。对长型（`long`）或浮点型（`float`）数组，一个成员将占用 4byte 的存储空间。对于多维数组来说，一个 $10 \times 10 \times 10$ 的三维浮点数组需要大约 4KB 的存储空间，而一个 $25 \times 25 \times 25$ 的三维浮点数组就需要大于 64KB 的存储空间（C51 单片机的最大可寻址空间只有 64KB）。

当数组，特别是多维数组中大多数元素没有被有效地利用时，就会浪费大量的存储空间。C51 单片机这样的嵌入式控制器，不像复用式系统那样拥有大型的存储区，其存储资源极为有限，因此无论如何不能被不必要地占用。因此在进行 C51 编程开发时要仔细地根据需求来选择数组的大小。

3.4.2 指针

指针是 C 语言中的一个重要的概念，也是 C 语言的一个重要特色。正确而灵活地运用它，可以有效地表示复杂的数据结构；能动态地分配内存；能方便地使用字符串；有效而方便地使用数组；在调用函数时能得到多于 1 个的值；能直接处理内存地址等，这对设计系统软件是很必要的。掌握指针的应用，可以使程序简洁、紧凑、高效。每一个学习和使用 C 语言的人，都应当深入地学习和掌握指针。可以说，不掌握指针就是没有掌握 C 语言的精华。

1. 指针的概念

为了了解指针的基本概念，必须了解数据在内存中是如何存储和读取的。前面已经讲过，一旦程序中定义了一个变量，C 语言编译器在编译时就给这个变量在内存中分配相应的内存空间。通常 C 语言系统对一个整型（`int`）变量分配两字节内存单元，对一个字符型变量分配

一字节 (char) 内存单元, 对一个浮点型变量分配 4byte 内存单元。

对于变量要弄清两个概念, 一个是变量名, 一个是变量值。前者是一个数据的标号, 后者是一个数据的内容。

对于内存单元, 也要弄清两个概念, 一个是内存单元的地址, 一个是内存单元的内容。前者是对该内存单元的编号, 它表示着该单元在整个内存中的位置; 后者指的是在该内存单元中存放的数据。

在变量与内存单元的对应关系中, 变量的变量名与内存单元的地址相对应; 变量的变量值与内存单元的内容相对应。

假设程序中定义了三个整型变量 a、b、c, 它们的值分别为 6、8、10, 而 C 语言编译系统将地址为 1000H 和 1001H 的两个字节内存单元分配给了变量 a, 将地址为 1002H 和 1003H 的两字节内存单元分配给了变量 b, 将地址为 1004H 和 1005H 的两个字节的内存单元分配给了变量 c, 则变量 a、b、c 与地址为 1000H~1005H 的内存单元之间的对应关系如图 3-18 所示。

在内存中变量名 a、b、c 是不存在的, 对变量值的存取是通过地址进行的。存取的方式有两种:

第一种是“直接访问”方式。例如: printf(“%d”, a), 其执行过程是这样的: 根据变量名与内存单元地址的对应关系, 找到变量 a 在内存中的位置, 即地址 1000H,

然后由地址 1000H 开始的两个字节中取出变量 a 的值 6, 最后把它用 printf() 语句按一定格式输出。这种访问方式就是直接访问方式。

第二种访问方式是间接访问方式。例如: 要存取变量 a 的值时, 可以将变量 a 的地址放在另一个内存单元中 (如放在 2010H 和 2011H 中), 访问时先找到存放变量 a 的地址的内存单元的地址 (2010H-2011H) 从中取出变量 a 的地址 (1000H), 然后从地址为 1000H 和 1001H 的两字节内存单元中取出变量 a 的值 6。这种访问方式就是间接访问方式。在这种访问方式中就使用了指针。

为了使用指针进行间接访问, 必须弄清关于指针的两个基本概念, 即变量的指针和指向变量的指针变量 (简称指针变量)。

变量的指针就是变量的地址。对于上面提到的变量 a 而言, 其指针就是 1000H。

若有一个变量专门用来存放另一个变量的地址 (即指针), 则该变量称为指向变量的指针变量 (简称指针变量)。指针变量的值是指针。对于上面提到的地址为 2010H 的内存单元, 如果定义一个变量 ap, 并使其定位在地址为 2010H 的这个内存单元上, 则 ap 就是一个指针变量。因为 ap 中 (即 2010H 址址单元中) 存放着变量 a 的地址 1000H。

请务必区分“指针”和“指针变量”这两个概念。例如, 可以说变量 a 的指针 (地址) 为 1000H, 不能说 a 的指针变量是 1000H。变量 a 的指针变量应该是 ap, ap 的指针是 2010H。

(1) 指针变量的定义

C 语言规定, 所有的变量在使用之前必须定义, 以确定其类型。指针变量也不例外, 由于它是用来专门存放地址的, 因此必须将它定义为指针类型。

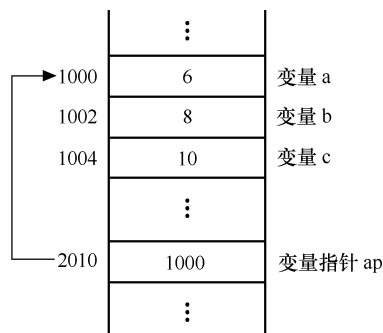


图 3-18 变量与内存单元的对应关系

指针定义的一般形式为：
类型识别符 *指针变量名；
例如：
int *ap;
float *pointer;

注意：指针变量名前面的“*”号表示该变量为指针变量。但指针变量名应该是 ap 和 pointer，而不是*ap 和*pointer。

(2) 指针变量的引用

弄清指针和指针变量的概念，掌握了指针变量的定义后，接下来就可以使用指针来进行前面所述的间接访问了。

例如：
int a, b, c; /*定义整型变量 a, b, c*/
int *ap; /*定义指针变量 ap*/
int *bp; /*定义指针变量 bp*/
int *cp; /*定义指针变量 cp*/

当进行完变量、指针变量定义之后，如果对这些语句进行编译的话，C 语言编译器就会给每一个变量和指针变量在内存中安排相应的内存单元。然而，这些单元的地址除非使用特殊的调试程序，否则是看不到的。为了能清楚地说明问题，假设 C 语言编译器对这些变量的地址定位如图 3-19 所示。

C 语言编译器将地址为 1000H, 1001H 的两字节内存单元指定给变量 a 使用，将地址 1002H, 1003H 的两字节内存单元指定给变量 b 使用，将地址为 1004H 和 1005H 的两字节内存单元指定给变量 c 使用。同理指针变量 ap 的地址为 2010H，指针变量 bp 的地址为 2012H，指针变量 cp 的地址为 2014H。注意，到现在为止还没有对上述所有的变量和指针变量进行赋值，故这些变量（指针变量）所对应的内存单元均为空白。

下面使用赋值语句对变量 a、b、c 进行赋值：

a=6;
b=8;
c=10;

通过编译，C 语言编译器就会在变量 a、b、c 对应的地址单元中装入初值 6、8、10。变量赋值后内存单元的具体情形如图 3-20 所示。

到现在为止，仍然没有对指针变量 ap、bp 和 cp 赋值，所以它们所对应的内存地址单元仍然为空白，即它们虽然已被定义为指针变量，但仍然没有被装入指针，它们谁也没有指向。

为了使空白的指针变量指向某一个具体的变量，就必须执行指针变量的引用操作。指针变量的引用是通过取地址运算符“&”来实现的。使用取地址运算符“&”和赋值运算符“=”就可以使一个指针变量指向一

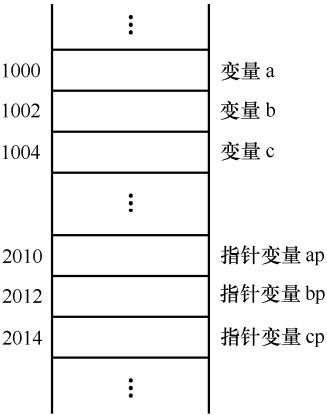


图 3-19 变量和指针变量的地址定位



图 3-20 变量赋值后的内存单元

个变量。例如：

```
ap=&a;
bp=&b;
cp=&c;
```

通过这些取地址运算和赋值运算操作后，指针 **ap** 就指向了变量 **a**，即指针变量 **ap** 所对应的内存地址单元中就装入了变量 **a** 所对应的内存单元的地址 **1000H**，指针变量 **bp** 就指向了变量 **b**，即指针变量 **bp** 所对应的内存地址单元中就装入了变量 **b** 所对应的内存单元的地址 **1002H**，而指针变量 **cp** 就指向了变量 **c**，即指针变量 **cp** 所对应的内存地址单元中装入了变量 **c** 所对应的内存单元的地址 **1004H**。指针变量赋值后内存单元具体情形如图 3-21 所示。

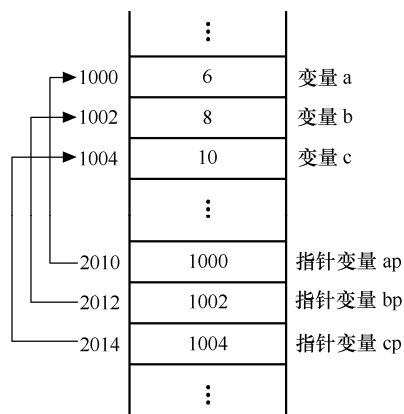


图 3-21 指针变量赋值后的内存单元

在完成了变量、指针变量的定义以及指针变量的引用之后，接下来就可以通过指针和指针变量来对内存进行间接访问了。这时要用到指针运算符（又称间接运算符）“*”。

例如，要将整型变量 **a** 的值赋给整型变量 **x**。

- ① 若使用直接访问方式，则用 “**x=a;**”;
- ② 若使用指针变量 **ap** 进行间接访问，则用 “**x=*ap;**”。

此时程序先从指针变量 **ap** 中取出 **a** 变量的指针（地址）**1000H**，然后从地址为 **1000H** 的内存单元中取出 **a** 变量的值 **6** 赋给变量 **x**。

应当特别注意的是，“*”在指针变量定义时和在指针运算时所代表的含义是不同的。在进行指针变量定义时，***ap** 中的*是指针变量类型说明符。在进行指针运算时，**x=*ap** 中的*是指针运算符。同样的“*”号，同样的***ap**在不同的场合下，含义截然不同。

在实际的编程和运算过程中，变量的地址和指针变量的地址是不可见的。变量、指针变量和内存单元地址这三者之间的对应关系完全由 C 语言编译器来确定。程序设计者只能通过取地址运算符“&”和指针运算符“*”来使指针变量与变量建立起联系，因此初学者难免会觉得指针运算比较抽象，难于掌握，故有必要对取地址运算符“&”和指针运算符“*”再做如下说明。

如果已经完成了指针变量的定义和引用，如：

```
int *ap, int a;
ap=&a;
```

则在进行指针运算时，有：

- ① ***ap** 与 **a** 是等价的，即***ap**就是 **a**。
- ② 由于***ap**与 **a** 等价，则**&*ap**与**&a** 等价。
- ③ 由于 **ap** 与**&a** 等价，则***&a**与***ap** 等价，即***&a**与 **a** 等价。
- ④ ***ap++**相当于 **a++**。

例如：

```
#define uchar unsigned char
uchar count;
```

```

uchar *x;           /*x 指向字符型变量*/
uchar xdata *y;      /*存在外部数据 RAM*/
uchar data *z;       /*存在内部数据 RAM*/
uchar code *w;       /*存在程序代码空间*/
uchar data *xdata zz; /*外部 RAM 指针指向内部 RAM 数据*/
x=&cowrit            /*得变量的地址即指针*/
*x=0xfe;            /*指针变量赋值*/

```

2. 数组指针和指向数组的指针变量

指针既然可以指向变量，当然也可以指向数组。所谓数组的指针，就是数组的起始地址。若有一个变量，用来存放一个数组的起始地址（指针），则称它为指向数组的指针变量。

（1）指向数组的指针变量的定义、引用和赋值

首先定义一个数组 `a[10]` 和一个指向数组的指针变量 `aap`：

```

int a[10];    /*定义 a 为包含 10 个整型元素的数组*/
int *app;     /*定义 app 为指向整型数据的指针*/

```

当未对指针变量 `app` 进行引用时，`app` 与 `a[10]` 毫不相干，即此时指针变量 `app` 并未指向数组 `a[10]`。为了将指针变量 `app` 指向数组 `a[10]`，需要对 `app` 进行引用，有两种引用方法：

① `app=&a[0]`

此时数组 `a[10]` 的第一个元素 `a[0]` 的地址就赋给了指针变量 `app`，也就是将指针变量 `app` 指向数组 `a[10]` 的第 0 号元素 `a[0]`。

② `app=a`

此种引用的方法与①的作用完全相同，但形式上更简单。C 语言规定，数组名可以代表数组的首地址，即第一个元素的地址，因此下面两个语句是等价的：

```

app=&a[0];
app=a;

```

也可以把指针变量的定义和引用放在一个语句中完成。

例如：

```

int *app=&a[0];
int *app=a;

```

其作用是定义一个指针变量 `app`，并将数组 `a[10]` 的首地址赋给指针变量 `app`（而不是 `*app`），如图 3-22 所示。

（2）通过指针引用数组元素

引用数组元素，可以使用数组下标法，如 `a[3]`，也可以使用指针法。与数组下标法相比，使用指针法引用数组元素能使目标程序代码效率高（占用内存少，运行速度快）。

通过指针引用数组元素的方法如下。

① `app+i` 和 `a+i` 就是数组元素 `a[i]` 的地址，即它们指向数组 `a[10]` 的第 `i` 个元素。由于 `a` 代表数组的首地址，则 `a+i` 与 `app+i` 等价。

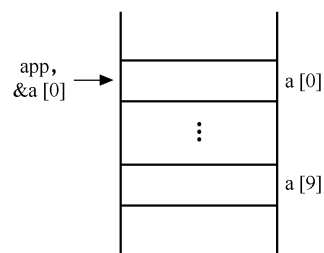


图 3-22 指针变量 `app` 的赋值

② $*(app+i)$ 或 $*(a+i)$ 是 $app+i$ 或 $a+i$ 所指向的数组元素，即 $a[i]$ 。

③ 指向数组的指针变量可以带下标，如 $app[i]$ 与 $*(app+i)$ 等价。

通过指针引用数值元素的示意图如图 3-23 所示。

C 语言规定： $app+1$ 指向数组首地址的下一个元素，而不是将指针变量 app 的值简单地加 1。例如，若数组的类型是浮点型（float），每个数组元素占 4 个字节，则对于浮点型指针变量 app 来说， $app+1$ 意味着使 app 的原值（地址）加 4 个字节，以使它指向下一个元素。一般来说， $app+i$ 所代表的地址实际上是 $app+i \times d$ ， d 是一个数组元素在内存中所占字节数（由元素的类型来确定：对于整型 $d=2$ ，对实型 $d=4$ ，对字符型 $d=1$ ）。

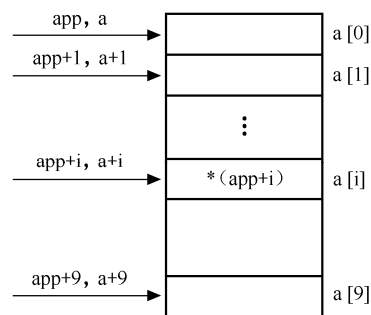


图 3-23 通过指针引用数组元素的示意图

【例 3-9】 设一个整型数组 a ，有十个元素。要求输出全部元素的值。

分析：要输出数组的全部元素的值有三种方法：

① 下标法。

```
main()
{
    int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    for(i=0; i<10; i++)
        printf( "%d", a[i]);
}
```

② 通过数组名计算数组元素的地址，找出元素的值。

```
main()
{
    int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    for(i=0; i<10; i++)
        printf( "%d", *(a+i));
}
```

③ 用指针变量指向数组元素。

```
main()
{
    int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *P;
    for(p=a; p<(a+10); p++)
        printf( "%d", *p);
}
```

以上三个程序的运行结果均为：

1 2 3 4 5 6 7 8 9 10

在这个例子中第①、②种方法执行的效率相同。C 语言编译器在进行编译时，就是将 $a[i]$ 转换成 $*(a+i)$ 来处理。由于要进行元素的地址计算，因此第①、②种方法找数组元素很费时间。而第③种方法由于用指针变量直接指向数组元素，不必每次都重新计算地址。因此像 $p++$ 这样的自加计算比第①、②种方法要快得多，能大大提高运算的效率。

(3) 关于指针变量的运算

若先使指针变量 p 指向数组 $a[10]$ (即 $p=a;$)，则：

① $p++$ (或 $P+=1$)

将使指针变量 p 指向下一个数组元素，即 $a[1]$ 。若再执行 $x=*P$ ，则将取出 $a[1]$ 的值，将其赋给变量 x 。

② $*p++$

由“++”与“*”运算符优先级相同，而结合方向为自右向左，因此 $*p++$ 等价于 $*(p++)$ ，其作用是先得到 p 指向的变量的值 (即 $*p$)，然后再执行 p 自加运算。

例如：for($i=0$; $i<10$; $i++$)

```
{
    p++;
    printf( "%d", *p);
}
```

等价于：

```
for( $i=0$ ;  $i<10$ ;  $i++$ )
    printf( "%d", p++);
```

③ $*p++$ 与 $*++p$ 作用不同

前者先取 $*p$ 值，后使 p 自加 1；后者先使 p 自加 1 再取 $*p$ 值。若 p 的初值为 $\&a[0]$ ，则执行 $x=*p++$ 时， x 值为 $a[0]$ 的值；而执行 $x=*++p$ 后，则 x 值等于 $a[1]$ 的值。

④ $(*P)++$

表示 p 所指向的元素值加 1。要注意的是，元素值加 1 而不是指针变量值加 1。若指针变量 p 指向 $\&a[0]$ ，且 $a[0]=3$ ，则 $(*p)++$ 等价于 $(a[0])++$ ，此时 $a[0]$ 值增为 4。

⑤ 若 p 当前指向数组中第 i 个元素，则

$*(p--)$ 与 $a[i--]$ 等价，相当于先执行 $*p$ ，然后再使 p 自减；

$*(++p)$ 与 $a[++i]$ 等价，先执行 p 自加，再执行 $*p$ 运算；

$*(--p)$ 与 $a[--i]$ 等价，先执行 p 自减，再执行 $*p$ 运算。

【例 3-10】 指针运算实例 1。

```
#define uint unsigned int
uint xdata a[];      /*方式 1*/
a[22]=0xff;
uint xdata *a;       /*方式 2，方式 1 和方式 2 等同*/
*(a+22)=0xff;
```

【例 3-11】 指针运算实例 2。

```
#include<reg51.h>
```

```

#define uchar unsigned char
uchar code m1[]={ "This is a test" };
uchar code m2[]={ "You failed" };
uchar code m3[]={ "You passed  " };
uchar code *code fail[]={&m1[0], &m2[0], 0};
uchar code *code pass[]={&m1[0], &m3[0], 0};
void display(uchar code *message)
{
    uchar code *m;
    for(; *message!=0; message++)
    {
        for(m=*message; *m!=0; m++)
        {
            P1=*m;
        }
    }
}
main()
{
    display(&pass[0]);
}

```

3. 指向多维数组的指针和指针变量

下面以二维数组为例来说明指向多维数组的指针和指针变量的使用方法。

首先定义一个整型的三行四列的两维数组 `a[3][4]`。同时定义一个这样的指针变量 `(*p)[4]`，即 `P` 是一个指针变量，它指向一个包含 4 个元素的一维数组。

下面使指针变量 `P` 指向数组 `a[3][4]` 的首址：`p=a`；则此时 `P` 和 `a` 等价，均指向数组 `a[3][4]` 的第 0 行首址；`p+1` 和 `a+1` 等价，均指向数组 `a[3][4]` 的第一行首址；`p+2` 和 `a+2` 等价，均指向数组 `a[3][4]` 的第二行首址；以下依此类推。而 `*(p+1)+3` 与 `&a[1][3]` 等价，指向 `a[1][3]` 的地址。`*(*(p+1)+3)` 与 `a[1][3]` 等价，表示 `a[1][3]` 的值。一般来讲，对于数组元素 `a[i][j]`：`*(p+i)+j` 就相当于 `&a[i][j]`，表示数组第 `i` 行第 `j` 列的元素的地址。`*(*(p+i)+j)` 就相当于 `a[i][j]`，表示数组第 `i` 行第 `j` 列元素的值。

【例 3-12】 输出二维数组中任一行、列元素的值。

```

main()
{
    int a[3][4]={ {1, 3, 5, 7}, {9, 11, 13, 15}, {17, 19, 21, 23} };
    int (*p)[4], i, j;
    p=a;
    i=2;

```

```

j=2;
printf( "a[%d, %d]=%d\n", i, j, *((*p+i)+j));
}

```

运行结果:

a[2, 2]=21

4. 关于 Franklin C51 的指针类型

Franklin C51 支持“基于存储器”的指针和“一般”指针两种指针类型。

基于存储器的指针类型由 C 源代码中存储器类型决定，并在编译时确定，用这种指针可以高效访问对象且只需 1 至 2 个字节。

一般指针需占用 3 个字节：1 个字节为存储器类型，2 个字节为偏移量。存储器类型决定了对象所用的 C51 存储空间，偏移量指向实际地址。一般指针可以访问任何变量而不管它在 C51 存储器空间中的位置。这样就允许一般函数，如 `memcpy()` 等将数据由任意一个地址拷贝到另一个地址空间。

(1) 基于存储器的指针

基于存储器的指针以存储器类型为参量，它在编译时才被确定。因此，为指针选择存储器的方法可以省掉，以便这些指针的长度可为 1 个字节（idata *, data *, pdata*）或 2 个字节（code*, xdata*）。在编译时，这类操作一般被“行内”（inline）编码，而无需进行库调用。

基于存储器的指针定义举例：

```
char xdata *px;
```

在 xdata 存储器中定义了一个指向字符类型（char）的指针。指针自身在默认存储区（决定于编译模式），长度为 2byte（值为 0~0xFFFF）。

```
char xdata *data pdx;
```

除了明确定义指针位于 C51 内部存储区（data）中外，其他与上例相同，它与编译模式无关。

```
data char xdata *pdx;
```

本例与上例完全相同。存储器类型定义既可以放在定义的开头，也可以直接放在定义的对象名之前。这种形式与早期的 C51 编译器版本相兼容。

```

struct time
{
    char hour;
    char min;
    char sec;
    struct time xdata * pxtime;
}

```

在结构 struct time 中（关于结构的细节详见 3.4.3 节），除了其他结构成员外，还包含一个具有和 struct time 相同的指针 pxtime，time 位于外部存储器（xdata），指针 pxtime 长度为 2byte。

```
struct time idata *ptime;
```


这个声明定义了一个位于默认存储器中的指针，它指向结构 `time`，`time` 位于 `idata` 存储器中，结构成员可以通过 C51 的 `@R0` 或 `@R1` 进行间接访问，指针 `ptime` 长度为 `lbyte`。

```
ptime->ptime->hour=12;
```

使用上面的关于 `struct time` 和 `struct time idata *ptime` 的定义，指针 `ptime` 被间接调用，它指向位于 `xdata` 存储器中的 `time` 结构。结构成员 `hour` 被赋值 12。

上面的例子阐明了 Franklin C51 指针的一般定义及使用方法，它们与 Franklin C51 所有的数据类型和 C51 单片机的存储器类型相关。所有用于一般指针的操作同样可以用于基于存储器的指针。

(2) 一般指针

一般指针包括 3 个字节：2 字节偏移和 1 字节存储器类型，即

地 址	+0	+1	+2
内 容	存储器类型	偏移量高位	偏移量低位

其中，第一个字节代表了指针的存储器类型，存储器类型编码如下所示。

存储器类型	idata	xdata	pdata	data	code
值	1	2	3	4	5

注意，使用其他类型值可能会导致不可预测的程序动作。

例如，以 `xdata` 类型的 `0x1234` 地址作为指针可以表示如下：

地 址	+0	+1	+2
内 容	0x02	0x12	0x34

当用常数作指针时，必须注意正确定义存储类型和偏移。例如，将常数值 `0x41` 写入地址为 `0x8000` 的外部数据存储器。

```
#define XBYTE((char*)0x20000L)
```

```
XBYTE[0x8000]=0x41;
```

其中，`XBYTE` 被定义为 `(char*)0x20000L`，`0x20000L` 为一般指针，其存储类型为 2，偏移量为 `0x0000`，这样 `XBYTE` 成为指向 `xdata` 零地址的指针，而 `XBYTE[8000]` 则是外部数据存储器的 `0x8000` 绝对地址。

注意，绝对地址被定义为“long”型常量，低 16 位包含偏移量，而高 8 位表明了存储器类型。为了表示这种指针，必须用长整数来定义存储类型。

Franklin C51 编译器不检查指针常数，用户必须选择有实际意义的值。

3.4.3 结构

C 语言中的结构，就是把多个不同类型的变量结合在一起形成的一个组合形变量，称为结构变量，简称结构。这些不同类型的变量可以是基本类型、枚举类型、指针类型、数组类型或其他结构类型的变量。这些构成一个结构的各个变量称为结构元素（或成员），它们的定义规则与变量名相同。

1. 结构的定义和引用

(1) 定义结构的类型

一个结构类型定义的一般形式为：

```
struct 结构名
{
    结构成员说明
};
```

结构成员说明的格式为：

类型标识符 成员名；

注意，在同一结构中不同分量不可同名。例如，定义一个名为 **date** 的结构类型：

```
struct date
{
    int month;
    int day;
    int year;
};
```

struct date 表示这是一个“结构类型”，其中 **struct** 是关键字，不能省略，**date** 为结构名。它包含了三个结构成员：**month**，**day**，**year**，这三个结构成员的数据类型都是整型（**int**），当然也可以根据实际需要选用各种不同类型的变量作为结构的成员。特别要指出的是，**struct date** 是程序员自己定义的结构类型，它和系统定义的标准类型（如 **int**，**char**，**float** 等）一样可以用来定义变量的类型。

(2) 定义结构类型变量

上面定义的 **struct date** 只是结构体的类型名，而不是结构体的变量名。为了在程序中正常地执行结构操作，除了定义了结构的类型名之外，还需要进一步定义该结构类型的变量名。

定义一个结构的变量的方法有 3 种。

① 先定义结构的类型，再定义该结构的变量名。

例如：

```
struct date
{
    int month;
    int day;
    int year;
};
date date1, date2; /*定义结构的变量名*/
```

上例中，在定义了结构的类型 **struct date** 之后，使用 **date** 来定义 **date1**、**date2** 为 **date** 类型的结构变量，即定义 **date1**、**date2** 为具有 **struct date** 类型的结构变量。

② 在定义结构类型的同时定义该结构的变量。

这种定义方法的一般形式为：

struct 结构名

{

 结构成员说明

}变量名 1, 变量名 2, ..., 变量名 n;

例如:

struct date

{

 int month;

 int day;

 int year;

}date1, date2; /*定义结构变量名*/

③ 直接定义结构类型变量。

这种定义方法的一般形式为:

struct

{

 结构成员说明

}变量名 1, 变量名 2,...,变量名 n;

下面对结构作几点说明。

① 结构体类型和结构体变量是两个不同的概念,不能混淆。对于一个结构变量来说,在定义时一般先定义一个结构类型,然后再定义该结构变量为该种结构体类型。

② 结构体的成员也可以是一个结构变量。

例如:

struct date

{

 int month;

 int day;

 int year;

}

struct clerk

{

 int num;

 char name[20];

 char sex;

 int age;

 struct date birthday; /*birthday 是结构类型 date 的变量*/

 float wages;

}clerk1, clerk2;

上面程序中先定义了一个 struct date,它代表“日期”,包括“年”,“月”,“日”三个成员,然后将结构 birthday 定义为 struct date 类型并作为结构成员加入到 struct clerk 结构中。

③ 结构的成员可以与程序中的其他变量名相同，但两者代表不同的对象。例如，在程序中可以另行定义一个 `name` 变量，但它与 `struct clerk` 中的成员 `name` 不是一回事，互不相干。

④ 如果在程序中所用到的结构数目多，规模大，可以将它们集中定义在一个头文件（以 `.h` 为后缀）中，然后用宏指令 `#include` 将该头文件包含在需要它们的源文件中。这样做的好处是便于装配、修改和使用。

（3）结构类型变量的引用

就结构而言，可操作的对象是结构类型变量，而不是结构类型。也就是说，当对结构进行引用时，只能对结构类型变量进行赋值、存取和运算，而不能对结构类型进行赋值、存取和运算。这是因为在编译时，C 语言编译器不对抽象的结构类型分配内存空间，它只对具体的结构类型变量分配内存空间。

对结构类型变量的引用应当遵守如下规则。

① 结构不能做为一个整体参加赋值、存取和运算，也不能整体地作为函数的参数，或函数的返回值。

对结构所执行的操作，只能用 “&” 运算符取结构的地址，或对结构变量的成员分别加以引用。引用的方式为：

结构变量名.成员名；

例如：`date1.year=1996;`

其中，“.” 是成员运算符，它在所有的运算符中优先级最高，因此可以把 `date.year` 作为一个变量来看待。上面赋值语句作用是将年号 1996 赋给 `struct date` 类型的结构变量 `date1` 中的成员 `year`。

② 如果结构类型变量的成员本身又属于一个结构类型变量，则要用若干个成员运算符 “.”，一级级地找到最低一级的成员，只有最低一级的成员才能参加赋值、存取或运算。“->” 符号和 “.” 符号等同。一般情况下，多级引用时，最后一级用 “.” 符号，高的级别用 “->” 符号。

例如：`clerk1.birthday.year=1957;`

注意，不能用 `clerk1.birthday` 来访问 `clerk1` 变量的成员 `birthday`，因为 `birthday` 本身也是一个结构类型变量。

③ 结构类型变量的成员可以像普通变量一样进行各种运算。例如：

`float sum=clerk1.wages+clerk2.wages;`

【例 3-13】 结构类型变量引用例程。

```
#define uchar unsigned char
```

```
#define uint unsigned int
```

```
struct stateform
```

```
{
```

```
    unsigned long s;
```

```
    uint t;
```

```
    uchar done;
```

```
};
```

```
struct stateform state;
```

```
state.t=321;
```

2. 结构数组

在前面讲到 `date` 结构类型时，虽然只定义了两个具有该类型的结构变量 `date1` 和 `date2`，但在使用时已经感到了对它们的引用的麻烦，因为尽管这两个变量结构相同，具有同样的成员项，但当使用 `printf()` 语句打印它们时，必须分别使用两个 `printf()` 语句。假若有若干个这样的结构变量，要将它们的内容全部打印出来，将是何等麻烦！为了避免上述情况的发生，可以将具有同样结构类型的若干个结构变量定义成结构数组，这样就可以使用循环语句对它们进行引用，从而大大提高效率。

当数组中的每个元素都具有相同的结构类型的结构变量，则称该数组为结构数组。结构数组与变量数组的不同之处在于：结构数组的每一个元素都是具有同一个结构类型的结构变量，并且都含有相同的成员项。

结构数组定义与结构变量的定义方法相似，只需将结构变量改成结构数组即可。

【例 3-14】 定义一个有 10 个元素的结构数组 `date1[10]`。

```
struct date
{
    int month;
    int day;
    int year;
};
struct date date1[10]; /*定义结构数组变量*/
```

也可以这样定义：

```
struct date
{
    int month;
    int day;
    int year;
}date1[10]; /*定义结构数组变量*/
```

或者按照如下方式定义：

```
struct
{
    int month;
    int day;
    int year;
}date1[10]; /*定义结构数组变量*/
```

3. 指向结构类型数据的指针

一个指向结构类型数据的指针，就是该数据在内存中的首地址。这里也可以设一个指针变量，使它指向一个结构数组，此时该指针变量的值就是结构数组的起始地址。

(1) 指向结构变量的指针变量

指向结构体变量的指针变量的一般形式为：

struct 结构类型名 *指针变量名；

或：

struct

{

结构成员说明

}*指针变量名；

指向结构变量的指针变量的一个实际应用是对信息进行传送，一个任务使用“传送一条信息”的方法与另一任务进行通信。这个传送操作包括传送信息结构变量的指针变量，这个指针变量是向接收任务传递的信息或指针。

【例 3-15】 指向结构类型数据的指针实例。

```
define uint unsigned int
```

```
define uchar unsigned char
```

```
struct msg1
```

```
{
```

```
    uint lnk;
```

```
    uchar len, fig, nod, sdt, cmd, stuff;
```

```
};
```

```
struct msg1 *msg;
```

```
void rqsendmessage(struct msg1 *m);    /* “传递信息” 函数*/
```

```
main()
```

```
{
```

```
    uchar stuff;
```

```
    msg->len=8;
```

```
    msg->fig=0;
```

```
    msg->nod=0;
```

```
    msg->sdt=0x12;
```

```
    msg->cmd=0;
```

```
    msg->stuff=stuff;
```

```
    rqsendmessage(msg);
```

```
}
```

在程序中，函数 `rqsendmessage()` 便是用于在两任务之间进行结构变量指针传送的专用函数，而 `struct msg1 *msg` 正是所要传递的结构变量的指针变量。主程序首先对信息结构赋值，然后调用 `rqsendmessage()` 函数将指针变量 `msg` 发送出去。

(2) 指向结构数组的指针变量

指向结构数组的指针变量的一般形式为：

struct 结构数组名 *结构数组指针变量名；

或：

```
struct
```

```
{
```

```
    结构成员说明
```

```
}*结构数组指针变量名[];
```

下面可以将前面关于指向结构变量的指针变量的例程作适当修改，使 `rqsendmessage()` 函数传送一个指向结构数组的指针变量。

```
#define uint unsigned int
```

```
#define uchar unsigned char
```

```
struct
```

```
{
```

```
    uint lnk;
```

```
    uchar len, flg, nod, sdt, cmd, stuff;
```

```
}msg1[4];
```

```
struct msg1 *p;
```

```
void rqsendmssage(struct msg1 *m);
```

```
main()
```

```
{
```

```
    uchar stuff=0;
```

```
    struct msg1 *p;
```

```
    for(p=msg1; p<msg1+4; p++)
```

```
    {
```

```
        p->len=8;
```

```
        p->flg=0;
```

```
        p->nod=0;
```

```
        p->cmd=0;
```

```
        p->stuff=stuff;
```

```
        rqsendmessage(p);    /*传递结构数组指针变量*/
```

```
    }
```

```
}
```

在本程序中，定义了一个具有 4 个元素的结构数组 `struct msg1[4]`，并定义了一个指向该结构数组的指针变量 `p(struct msg1 *p)`。主程序 `main()` 中使用 `for` 循环语句和 `p` 指针对结构数组 `msg1[4]` 的所有元素中的每个成员进行初始化赋值，并且每初始化一个 `msg1` 结构数组的成员，就将其指针变量用 `rqsendmessage(p)` 函数发送出去供接收任务进行相应的操作处理。在 `for` 循环中 `p=msg1` 意味着结构数组指针 `p` 的初值指向结构数组 `msg1[0]` 的首地址。在第 1 次循环中，`rqsendmessage()` 输出的是结构数组元素 `msg1[0]` 的首地址，然后 `p++` 使指针 `p` 自加 1，`p+1` 意味着 `p` 所增加的地址值为结构数组 `msg1[]` 中一个结构元素的所有成员所占有的字节数（在本例中为 8 字节），即 `p++` 使 `p` 指向 `msg1[1]` 的首地址。在第 2 次循环中，`rqsendmessage(p)` 传送的是 `msg1[1]` 的首地址……当 `p++` 使 `p` 值变为 `p+4` 时，根据循环判断条件 `p<p+4`，循环终止。

最后对 p 指针的运算作以下说明。如果指针 p 指向结构数组 msg1[0]的首地址，则：

① (*p).fig 与 p->fig 及 msg1[0].fig 三者完全等价，即(*p).成员名与 p->成员名及结构数组元素成员名是等价的。

② p+1 使指针 p 指向结构数组 msg1[0]的下一个元素 msg1[1]的首地址。

③ 由于指向运算符“->”的优先级高于自加运算符“++”，则(++p)->fig 先使 p 自加 1 指向 msg1[1]的首地址，然后得到它指向的 msg1[1].fig 成员值。

(p++)->fig: p 自加 1 指向 msg1[1]的首址。

p->fig++是先得到 msg1[0].fig 成员值，使用完后使 msg1[0].fig 的值加 1。

++p->fig 是先将 msg1[0].fig 成员值加 1，然后再使用。

3.4.4 共用体

无论任何数据，在使用前必须定义其数据类型。只有这样，在编译时 C 语言编译器才会根据其数据类型，在内存中为其指定相应长度的内存单元，供其使用。不同数据类型的数据占据各自拥有的内存空间，彼此互不“侵犯”。那么是否存在某种数据类型，使 C 语言编译器在编译时为其指定一块内存空间，并允许各种类型的数据共同使用呢？回答是肯定的，这种数据类型就是共用体或称联合。

共用体是 C 语言的构造类型数据结构之一，它与数组和结构等一样，也是一种比较复杂的构造数据类型。

共用体与结构类似，它也可以包含多个不同数据类型的元素，但其变量所占有的内存空间并不是各成员所需存储空间之和，而是在任何时候，其变量至多只能存放该类型所包含的一个成员，即它所包含的各个成员只能分时共享同一存储空间。这是共用体与结构的区别所在。

定义共用体类型的一般格式为：

union 共用体类型标识符

```
{
    类型说明符变量名;
```

```
};
```

说明共用体变量的一般格式为：

union 共用体类型标识符 共用体变量名表；

例如：

```
union int_or_char
```

```
{
    int i;
    char ch;
};
```

以上定义了一个名为 int_or_char 的共用体类型，该类型包含两个不同类型的元素：一个是 int 型，另一个是 char 型。

```
Union int_or_char cnvt;
```

上句定义一个 int_or_char 类型共用体变量 cnvt，它能使一个整型变量 cnvt.i 和一个字

符型变量 `cnvt.ch` 分时共享同一存储空间。

与结构变量一样，也可以在定义共用体类型的同时，定义共用体变量。

例如：

```
union int_or_char
{
    int i;
    char ch;
}cnvt;      /*定义共用体变量*/
```

或：

```
union
{
    int i;
    char ch;
}cnvt;      /*定义共用体变量*/
```

对于共用体变量，系统只给该变量按其各共用体成员中所需空间最大的那个成员的长度分配一个内存空间。例如上面的 `cnvt` 共用体变量，它共有两个元素：一个为 `int` 类型，需要 2byte 内存空间；另一个是 `char` 类型，只需 1byte 空间，所以 C 语言编译器只给共用体变量 `cnvt` 分配两字节内存空间。这样在共用体中，每时每刻它只能保存共用体类型中的一个成员，而且此时也只能访问该成员。由此可知，共用体变量可以在不同时间内保存不同类型和长度的数据，从而提供了在同一存储单元中可以分时操作不同类型数据的功能。

例如：

```
#include<reg51.h>
#define uint unsigned int
#define uchar unsigned char
union u
{
    uint word;
    struct{uchar hi; uchar lo; }bytes
};
union u newcount;
uint oldcount;
newcount.bytes.hi=TH1;
newcount.bytes.lo=TL1;
oldcount=newcount.word;
```

这样，定时器的计数值既可以按字节使用，也可以按字使用。

3.4.5 枚举

在 C 语言中，用作标志的变量通常只能被赋予 `True` (1) 或 `False` (0) 两个值中的一个。

但由于疏忽，有时会将一个在程序中作为标志使用的变量，赋予了除 True(1)或 False(0)以外的值。另外，这些变量通常被定义成 int 数据类型，从而使它们在程序中的作用模糊不清。如果可以定义标志类型的数据变量，然后指定这种被说明的数据变量只能赋值为 True 或 False，不能赋予其他值，就可以避免上述情况的发生。枚举数据类型正是因这种需要而产生的。

(1) 枚举的定义和说明

枚举数据类型是一个有名字的某些整数型常量的集合。这些整数型常量是该类型变量可取的所有合法值。枚举定义应当列出该类型变量的所有可取值。

一个完整的枚举定义说明语句的一般格式为：

```
enum 枚举名 {枚举值列表} 变量列表;
```

枚举的定义和说明也可以分成两个语句完成：

```
enum 枚举名 {枚举值列表};
```

```
enum 枚举名 变量列表;
```

例如：

```
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat} d1, d2;
```

或：

```
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

```
enum day d1, d2;
```

只有在建立了枚举类型的原型 enum day，将枚举名与枚举值列表联系起来，并进一步说明该原型的具体变量 d1、d2 后，C 语言编译器才会给 d1、d2 分配存储空间。这些变量才可以具有与所定义枚举列表中的值。

(2) 枚举变量的取值

在枚举列表中，每一项符号代表一个整数值。在默认情况下，第一项符号取值为 0，第二项取值为 1，第三项取值为 2，……，依次类推。此外，也可以通过初始化指定某些项的符号值。某项符号初始化后，该项后续各项符号值随之依次递增，例如：

```
enum direct {up, down, left=10, right};
```

则 C 语言编译器将 up 赋值为 0，down 赋值为 1，由于 left 被初始化为 10，则 right 值为 11。

例如：将颜色为红、绿、蓝的三个球作全排列，共有几种排法？打印出每种组合的三种颜色。

程序如下：

```
main()
{
    enum color {red, green, blue};
    enum color i, j, k, st;
    int n=0, lp;
    for(i=red; i<=blue; i++)
        for(j=red; j<=blue; j++)
        {
```

```
for(k=red; k<=blue; k++)
{
    n=n+1;
    printf( "%-4d", n);
    for(lp=1; lp<=3; lp++)
    {
        switch(lp)
        {
            case 1: st=i; break;
            case 2: st=j; break;
            case 3: st=k; break;
            default: break;
        }
        switch(st)
        {
            case red: printf( "%-10s", "red" ); break;
            case green: printf( "%-10s", "green" ); break;
            case blue: printf( "%-10s", "blue" ); break;
            default: break;
        }
    }
    printf( "\n" );
}
printf( "\n total: %5d\n", n)
}
```

3.5 函数

在高级语言中，函数和另外两个名词“子程序”、“过程”用来描述同样的事情，它们都含有以同样的方法重复地去做某件事的意思。在 C51 中使用“函数”这个术语。主程序(main())可以根据需要用来调用函数。当函数执行完毕时，就发出返回(return)指令，而主程序 main()用后面的指令来恢复主程序流的执行。同一个函数可以在不同的地方被调用，并且函数可以重复使用。

C 语言程序是由一个个函数构成的。在构成 C 语言程序的若干个函数中，必有一个是主函数 main()。下面给出了 C 语言程序的一般组成结构：

 全程变量说明；

```

main()/*主函数*/
{
    局部变量说明;
    执行语句;
}
function_1(形式参数表)/*函数 1*/
形式参数说明;
{
    局部变量说明;
    执行语句;
}
function_n(形式参数表)/*函数 n*/
形式参数说明;
{
    局部变量说明;
    执行语句;
}

```

所有函数在定义时都是相互独立的，一个函数中不能再定义其他函数，即函数不能嵌套定义，但可以互相调用。函数调用的一般规则是：主函数可以调用其他普通函数；普通函数之间也可以互相调用，但普通函数不能调用主函数。

一个 C 语言程序的执行是从 `main()` 函数开始，调用其他函数后返回到主函数 `main()` 中，最后在主函数 `main()` 中结束整个 C 语言程序的运行。

3.5.1 函数的分类

从 C 语言程序的结构上划分，C 语言函数分为主函数 `main()` 和普通函数两种。从用户使用的角度划分，普通函数又分为两种：一种是标准库函数，一种是用户自定义函数。

(1) 标准库函数

标准库函数是由 C 语言编译系统的函数库提供的，早在 C 语言编译系统设计过程中，系统的设计者事先将一些独立的功能模块编写成公用函数，并将它们集中存放在系统的函数库中，供系统的使用者在设计应用程序时使用，故把这种函数称为库函数或标准库函数。C 语言系统一般都具有功能强大、资源丰富的标准函数库。因此，作为系统的使用者，在进行程序设计时，应该善于充分利用这些功能强大、内容丰富的标准库函数资源，以提高效率，节省时间。

(2) 用户自定义函数

用户自定义函数，顾名思义，它是用户根据自己的需要编写的函数。

从函数定义的形式上划分，普通函数可以分为 3 种形式：无参数函数、有参数函数和空函数。

(1) 无参数函数

这种函数在被调用时，既无参数输入，也不返回结果给调用函数。它是为完成某种操作

而编写的。

(2) 有参数函数

在调用这种函数时，必须提供实际的输入参数，它在被调用时，必须说明与实际参数一一对应的形式参数，并在函数结束时返回结果供调用它的函数使用。

(3) 空函数

这种函数体内无语句，是空白的。调用这种空函数时，什么工作也不做，不起任何作用。而定义这种函数的目的并不是为了执行某种操作，而是为了以后程序功能的扩充。

在程序的设计过程中，往往根据需要确定若干模块，分别由一些函数来实现。而在程序设计的第一阶段，往往只设计最基本功能模块的函数，其他模块的功能函数，则可以在以后补上。为此先将这些非基本模块的功能函数定义成空函数，先占好位子，以后再用一个编好的函数代替它。这样做的好处是，程序的结构清楚，可读性好，以后扩充新功能也方便。

3.5.2 函数的定义

上节讨论了从函数定义的形式上划分，函数有3种形式：无参数函数、有参数函数、空函数。下面讨论这3种函数的具体定义方法。

1. 无参数函数的定义方法

无参数函数的定义形式为：

返回值类型标识符 函数名()

{函数体语句}

无参数函数一般不带返回值，因此函数返回值类型识别符可以省略。

【例 3-16】 函数定义实例。

```
#include<stdio.h>
```

```
func()
```

```
{
```

```
    printf( "Function In func respond the call of Main\n" );
```

```
}
```

```
main()
```

```
{
```

```
    printf( "Function In Main Calls A Function in func\n" );
```

```
    func();
```

```
}
```

上面程序中，实际定义了两个函数：`main()`和`func()`，它们都是无参数函数。因此它们的返回值标识符可以省略，默认值是 `int` 类型。

上面程序中函数 `func()`放在主函数 `main()`之前，这是经典的 C 语言程序写法。但是标准 C (ANSI C) 则要求用另一种格式进行规范化书写：首先，即使是无参数函数，其返回值类型标识符也要注明“`void`”关键字；而主函数 `main()`则放在文件的前面，被调用的函数应

在开头进行原型声明。上面的程序若按 ANSI C 写法应改为：

```
#include<stdio.h>
void func(void); /*进行函数原型声明*/
int main(void)
{
    printf( "Function In Main Calls A Function in func\n" );
    func();
}
void func(void)
{
    printf( "Function In func respond the call of main\n" );
}
```

2. 有参数函数的定义方法

有参数函数的定义形式为：

返回值类型识别符 函数名（形式参数列表）

形式参数说明；

{函数体语句;}

【例 3-17】 求两个数的最大公约数

```
#include<stdio.h>
int gcd(u, v)
int u, v;
{
    int temp;
    while(v!=0)
    {
        temp=u%v;
        u=v;
        v=temp;
    }
    return(u);
}
main()
{
    int result, a=150, b=-35;
    printf( "a=%d, b=%d", a, b);
    result=gcd(a, b);
    printf( "The gcd of %d and %d is %d\n", a, b, result);
}
```

程序运行结果：

a=150 b=35

The gcd of 150 and 35 is 5

在本程序中，`int gcd(u, v)`就是一个典型的有参数函数。其中，`int`为函数返回值类型标志符，`gcd`为函数名，而括号中的 `u, v` 则函数的输入形式参数。在 `gcd` 函数的结尾处有一个返回语句 `return(u)`，其中 `u` 为函数的返回变量。

3. 空函数的定义方法

空函数的定义形式为：

返回值类型说明符 函数名()

{ }

例如：

`float dummy()`

{ }

调用此函数时，什么工作也不做，没有任何实际作用。在主调函数中写上“`dummy();`”表明“这里要调用一个函数”，而现在这个函数没有起作用，等以后扩充函数功能时补充上。

3.5.3 函数的参数和函数值

C语言采用函数之间的参数传递方式，使一个函数能对不同的变量进行功能相同的处理，从而大大提高了函数的通用性与灵活性。

函数之间的参数传递，是通过主调函数的实际参数与被调用函数的形式参数之间进行数据传递来实现。被调用函数的最后结果由被调用函数的 `return` 语句返回给调用函数。

1. 形式参数和实际参数

在定义函数时，函数名后面括号中的变量名称为形式参数，简称形参。在函数调用时，主调函数名后面括号中的表达式称为实际参数。

以前面提到的求两个数的最大公约数程序为例。程序中函数说明语句 `int gcd(u, v)` 括号中的变量 `u` 和 `v` 即为该函数的被调用函数的形式参数；而在主函数 `main()` 中的 `result=gcd(a, b)` 语句括号中的变量 `a` 和 `b` 则是调用函数的实际参数。该语句在调用 `gcd()` 函数的同时，将已赋值的实际参数 `a` 和 `b` 传递给 `gcd(u, v)` 函数的形式参数 `u` 和 `v`，由 `gcd` 函数用 `u` 和 `v` 进行运算。

在C语言的函数调用中，实际参数与形式参数之间的数据传递是单向进行的。只能由实际参数传递给形式参数，而不能由形式参数传递给实际参数。

实际参数与形式参数的类型必须一致，否则会发生类型不匹配的错误。被调用函数的形式参数在函数未被调用之前，并不占用实际内存单元。只有当函数调用发生时，被调用函数的形式参数才被分配给内存单元，此时内存中调用函数的实际参数和被调用函数的形式参数位于不同的单元中。在调用结束后，形式参数所占有的内存被系统释放，而实际参数所占有的内存单元仍然保留并维持原值。

2. 函数的返回值

上面讲到主调用函数 `main()` 在调用有参数函数 `gcd()` 的时候, 将实际参数 `a` 和 `b` 传递给被调用函数的形式参数 `u` 和 `v`。然后, 被调用函数 `gcd()` 使用形式参数 `u` 和 `v` 作为输入变量进行运算, 所得结果通过返回语句 `return(u)` 返回给主函数, 并在主函数的 `result=gcd(a, b)` 语句中通过等号赋值给变量 `result`。这个 `return(u)` 中 `u` 变量值就是被调用函数的返回值, 简称函数的返回值。

函数调用时, 主调用函数与被调用函数之间参数传递及函数值返回的全部过程示意图如图 3-24 所示。

函数的返回值是通过函数中的 `return` 语句获得的。一个函数可以有一个以上的 `return` 语句, 但多于一个的 `return` 语句必须在选择结构 (`if` 或 `do/case`) 中使用, 因为被调用函数一次只能返回一个变量值。

`return` 语句中的返回值也可以是一个表达式, 例如, 可以使用冒号选择表达式:

```
return(x>y? x:y)
```

当 `x>y` 为“真”, 则返回 `x` 值, 否则返回 `y` 值。

函数返回值的类型一般在定义函数时, 采用返回类型标识符来指定。例如:

`int gcd(u, v)` 中在函数名 `gcd` 之前的 `int` 指定函数的返回值为整型 (`int`); `char letter(a, b)` 指定函数的返回值的类型为字符 (`char`)。

C 语言规定, 凡不加返回类型标识符说明的函数, 都按整型 (`int`) 来处理。如果函数返回值的类型说明和 `return` 语句中表达式的变量类型不一致, 则以函数返回类型标识符为标准进行强制类型转换。

另外, 有时为了明确表示被调用函数不带返回值, 可以将该函数定义为“`void`”(无类型)。

例如:

```
void print_message()
{.....}
```

在许多 C 语言程序中, 为了减少程序出错, 保证正确调用, 凡不要求返回结果的函数, 一般都被定义为 `void` 函数。

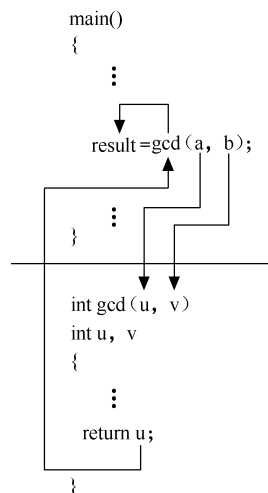


图 3-24 参数传递及函数值返回示意图

3.5.4 函数的调用

1. 函数调用的一般形式

函数调用的一般形式为:

函数名 (实参表列);

如果是调用无参函数, 则“实参表列”可以没有, 但括弧不能省略。如果实参表列包含多个实参, 则各参数间用逗号隔开。实参与形参的个数应相等, 类型应一致。实参与形参按

顺序对应，一一传递数据。但应说明，如果实参表列包括多个实参，对实参求值的顺序并不是确定的，有的系统按自左至右顺序求实参的值，有的系统则按自右至左顺序求实参的值。许多C版本（例如 Turbo C 和 MS C）是按自右至左的顺序求值。

【例 3-18】 函数调用实例。

```
main()
{
    int i=2, p;
    p=f(i, ++i);    /* 函数调用 */
    printf( "%d", p);
}
int f(int a, int b)    /* 函数定义 */
{
    int c;
    if(a>b)  c=1;
    else if (a==b)  c=0;
    else c=-1;
    return(c);
}
```

在 Turbo C 系统上运行的结果为：0

如果按自左至右顺序求实参的值，则函数调用相当于 f(2, 3)，程序运行结果为“-1”。若按自右至左顺序求实参的值，则它相当于 f(3, 3)，程序运行结果为“0”。读者可以在所用的计算机系统上试一下，以便知道它所处理的方法。由于存在上述情况，使程序通用性受到影响。因此应当避免这种容易引起不同理解的情况。如果本意是按自左而右顺序求实参的值，可以改写为：

```
j=i;
k=++i;
p=f(j, k);
```

如果本意是自右而左求实参的值，可改写为：

```
j=++i;
p=f(j, j);
```

这种情况在 printf 函数中也同样存在，例如：

```
printf( "%d, %d", i, i++);
```

该语句也会发生上述同样的问题，若 i 的原值为 3，在 Turbo C 上运行结果为 4, 3。请读者务必注意，应该避免这种容易混淆的用法。

2. 函数调用的方式

按函数在程序中出现的位置来分，可以有以下三种函数调用方式。

(1) 函数语句

函数调用作为一个语句出现。这时不要求函数返回值，只要求函数完成一定的操作。

(2) 函数表达式

函数出现在一个表达式中，这种表达式称为函数表达式。这时要求函数返回一个确定的值以参加表达式的运算。例如：

```
c=2*max(a, b);
```

函数 `max` 是表达式的一部分，它的值乘 2 再赋给 `c`。

(3) 函数参数

函数调用作为一个函数的实参。例如：

```
m=max(a, max(b, c));
```

其中 `max(b, c)` 是一次函数调用，它的值作为 `max` 另一次调用的实参。`m` 的值是 `a`、`b`、`c` 三者最大的。又如：`printf("%d", max(a, b));` 也是把 `max(a, b)` 作为 `printf` 函数的一个参数。

函数调用作为函数的参数，实质上也是函数表达式形式调用的一种，因为函数的参数本来就要求是表达式形式。

3. 对被调用函数的声明和函数原型

在一个函数中调用另一函数（即被调用函数）需要具备以下条件。

(1) 首先被调用的函数必须是已经存在的函数（库函数或用户自己定义的函数）。但只有这一条件还不够。

(2) 如果使用库函数，一般还应该在本文件开头用 `#include` 命令将调用有关库函数时所需用到的信息“包含”到本文件中来。例如，前几章中已经用过的 `#include<studio.h>`，其中“`studio.h`”是一个“头文件”。在 `studio.h` 文件中放了输入输出库函数所用的一些宏定义信息。如果不包含“`studio.h`”文件中的信息，就无法使用输入输出库中的函数。同样，使用数学库中的函数，应该用 `#include<math.h>`，其中 `.h` 是头文件所用的后缀，用来标志头文件（header file）。

(3) 如果使用用户自己定义的函数，而且该函数与调用它的函数（即主调函数）在同一个文件中，一般还应该在主调函数中对被调用的函数作声明，即向编译系统声明将要调用此函数，并将有关信息通知编译系统。“声明”一词的原文是 `declaration`，过去在许多书中译为“说明”，近年来，愈来愈多的计算机专家提出应称为声明，作者也认为称为“声明”更确切，表意更清楚。

【例 3-19】 对被调用的函数作声明。

```
main()
{
    float add(float x, float y); /*对被调用函数的声明*/
    float a, b, c;
    scanf( "%f, %f", &a, &b);
    c=add(a, b);
    printf( "sum is %f", c);
}

float add(float x, float y) /*函数首部*/
{
    float z;          /* 函数体 */
    z=x+y;
```

```
return(z);
}
```

运行情况如下：

3.6, 6.5

sum is 10.000000

这是一个很简单的函数调用，函数 `add` 的作用是求两个实数之和，得到的函数值也是实型。请注意程序第2行“`float add (float x, float y):`”是对被调用的 `add` 函数作声明。

注意，对函数的“定义”和“声明”不是一回事。“定义”是指对函数功能的确立，包括指定函数名、函数值类型、形参及其类型、函数体等，它是一个完整的、独立的函数单位；而“声明”的作用则是把函数的名字、函数类型以及形参的类型、个数和顺序通知编译系统，以便在调用该函数时系统按此进行对照检查（例如函数名是否正确，实参与形参的类型和个数是否一致）。从程序中可以看到对函数的声明与函数定义中的第1行（函数首部）基本上是相同的。因此可以简单地照写已定义的函数的首部，再加一个分号，就成为了对函数的“声明”。

其实，在函数声明中也可以不写形参名，而只写形参的类型。例如“`float add(float, float)`”。在C语言中，把以上形式的函数声明称为函数原型（`function prototype`）。使用函数原型是C的一个重要特点，它的作用主要是利用它在程序的编译阶段对调用函数的合法性进行全面检查。从上面例子中可以看到 `main` 函数的位置在 `add` 函数的前面，如果没有对函数的声明，当编译到包含函数调用的语句“`c=add(a, b);`”时，编译系统不知道 `add` 是不是函数名，也无法判断实参（`a` 和 `b`）的类型和个数是否正确，因而无法进行正确性的检查。只有在运行时才会发现实参与形参的类型或个数不一致，出现运行错误。但是在运行阶段发现错误并重新调试程序，是比较麻烦的，工作量也较大，因此应当在编译阶段尽可能多地发现错误，随之纠正错误。现在在函数调用之前用函数原型做了函数声明，因此编译系统记下了所需调用的函数的有关信息，在对“`c=add(a, b);`”进行编译时就“有章可循”了。编译系统根据函数的原型对函数的调用的编译时是从上到下逐行进行的，合法性进行全面的检查。和函数原型不匹配的函数调用会导致编译出错，它属于语法错误。用户根据屏幕显示的出错信息很容易发现和纠正错误。

函数原型的一般形式为：

(1) 函数类型 函数名（参数类型1，参数类型2.....）

(2) 函数类型 函数名（参数类型1，参数名1，参数类型2，参数名2.....）

第1种形式是基本的形式。为了便于阅读程序，也允许在函数原型中加上参数名，就成了第2种形式。但编译系统不检查参数名，因此参数名是什么都无所谓。上面程序中的声明也可以写成 `float add(float a, float b);` 参数名不用 `x`、`y`，而用 `a`、`b` 效果完全相同。

应当保证函数原型与函数首部写法上的一致，即函数类型、函数名、参数个数、参数类型和参数顺序必须相同。函数调用时函数名、实参个数应与函数原型一致。实参类型必须与函数原型中的形参类型赋值兼容，按第2章介绍的赋值规则进行类型转换。如果不是赋值兼容，就按出错处理。

下面对被调用户函数的声明作以下说明：

(1) 以前C版本的函数声明方式不是采用函数原型，而只声明函数名和函数类型。例如，例3-19中也可以采用下面的函数声明形式：

```
float add( );
```

该声明形式不包括参数类型和参数个数。系统不检查参数类型和参数个数。新版本也兼容这种用法，但不提倡这种用法，因为它未进行全面的检查。

(2) 实际上，如果在函数调用之前，没有对函数作声明，则编译系统会把第一次遇到的该函数形式（函数定义或函数调用）作为函数的声明，并将函数类型默认为 `int` 型。例如在调用 `max` 函数之前没有进行函数声明，编译时首先遇到的函数形式是函数调用“`max(a, b)`”，由于对原型的处理是不考虑参数名的，因此系统将 `max()` 加上 `int` 作为函数声明，即：

```
int max();
```

因此，不少 C 教材认为，如果函数类型为整型，可以在函数调用前不必作函数声明。但是使用这种方法时，系统无法对参数的类型做检查。若调用函数时参数使用不当，在编译时也不会报错。因此，为了程序清晰和安全，建议都加上声明为好。例如，在 `max` 函数中最好加上以下函数声明：

```
int max(int, int);
```

或：

```
int max(int x, int y);
```

(3) 如果被调用函数的定义出现在主调函数之前，可以不必加以声明。因为编译系统已经先知道了已定义的函数类型，会根据函数首部提供的信息对函数的调用作正确性检查。

如果把例 3-19 改写如下（即把 `main` 函数放在 `add` 函数的下面），那么就不必在 `main` 函数中对 `add` 声明。

```
float add(float x, float y)
{
    float z ;
    z=x+y;
    return(z);
}
main()    /*不必对 add 函数作声明*/
{
    float a, b, c;
    scanf( "%f, %f", &a, &b);
    c=add(a, b);
    printf( "%f", c);
}
```

(4) 如果已在所有函数定义之前，在函数的外部已做了函数声明，则在各个主调函数中不必对所调用的函数再作声明。例如：

```
char letter(char, char);
/*以下 3 行在所有函数之前，且在函数外部*/
float f(float, float);
int i(float, float);
main()
```

```

{...}                /*不必声明它所调用的函数*/
char letter(char c1, char c2)    /*定义 letter 函数*/
{...}
float f(float x, float y)        /*定义 f 函数*/
{...}
int i(float j, float k)          /*定义 i 函数*/
{...}

```

除了以上(2)、(3)、(4)所提到的三种情况外,都应该按上述介绍的方法对所调用函数作声明,否则编译时就会出现错误。用函数原型来声明函数,还能减少编写程序时可能出现的错误。由于函数声明的位置与函数调用语句的位置比较近,因此在写程序时便于就近参照函数原型来书写函数调用,不易出错。

4. 函数的嵌套调用

C语言的函数定义都是互相平行、独立的,也就是说在定义函数时,一个函数内不能包含另一个函数,C语句不能嵌套定义函数,但可以嵌套调用函数,即在调用一个函数的过程中,又调用另一个函数。函数嵌套调用示意图如图3-25所示。

图3-25表示的是两层嵌套(连main函数共3层函数),其执行过程是:

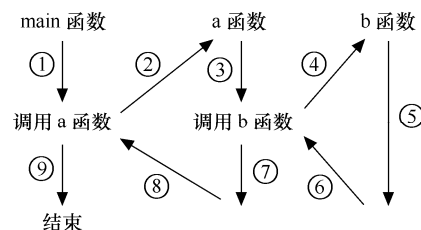


图 3-25 函数嵌套调用示意图

- (1) 执行 main 函数的开头部分;
- (2) 遇函数调用 a 的操作语句, 流程转去 a 函数;
- (3) 执行 a 函数的开头部分;
- (4) 遇调用 b 函数的操作语句, 流程转去函数 b;
- (5) 执行 b 函数, 如果再无其他嵌套的函数, 则完成 b 函数的全部操作;
- (6) 返回调用 b 函数处, 即返回 a 函数;
- (7) 继续执行 a 函数中尚未执行的部分, 直到 a 函数结束;
- (8) 返回 main 函数中调用 a 函数处;
- (9) 继续执行 main 函数的剩余部分直到结束。

5. 函数的递归调用

在调用一个函数的过程中, 又直接或间接地调用该函数本身, 这种情况称为函数的递归调用。

C 语言的强大优势之一就在于它允许函数的递归调用。函数的递归调用通常用于问题的求解, 它可用于计算含嵌套括号的表达式, 还普遍用于检索和分类 trees 和 lists 的数据结构。

下面以计算一个数的阶乘为例来说明函数的递归调用。

【例 3-20】 计算一个数的阶乘。

一般说来, 任何大于 0 的正整数 n 的阶乘等于 n 被 $(n-1)$ 的阶乘乘, 即 $n!=n(n-1)!$ 。用 $(n-1)!$ 的值来表示 $n!$ 的值的表达式就是一种递归调用, 因为一个阶乘的值是以另一个阶乘的值为基

础的。采用递归调用求正数 n 的阶乘的程序如下：

```
int factorial(n)
int n;
{
    int result;
    if(a==0)
        result=1;
    else
        result=n*factorial(n-1);
    return(result);
}
main()
{
    int j, c;
    for(j=0; j<11; j++)
        c=factorial(j);
    printf( "%d!=%d\n", j, c);
}
```

程序运行结果：

```
0!=1
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
8!=40320
9!=362880
10!=3628800
```

在程序 `factorial()` 函数中，包含着对它本身的调用，这使该函数成为递归型函数。下面来看看当函数计算 3 的阶乘时发生递归调用的全过程。

当函数开始工作时，形式参数 n 被指定为 3。因为此值不为 0，故程序将跳过 `if(n==0)` 一句而执行 `result=n*factorial(n-1)`；即：

```
result=3*factorial(2);
```

在这个表达式中 `factorial()` 函数调用了其自身，只不过现在的实际参数变为 2。根据自右至左结合的原则，先计算 `factorial(2)`，而将 3 乘以 2 的阶乘运算放在一边。尽管正再一次调用同一个函数，而实际上却是在调用各自独立的函数。因为每当 C 语言中的任意函数被调用时（无论是否递归），函数都将获得一套属于自己的局部变量和形式参数。当函数执行完毕，

系统会自动将这局部变量释放掉。因此当调用 `factorial()` 函数计算 2 的阶乘时所用的局部变量 `result` 和形式参数 `n` 与调用该函数计算 3 的阶乘时所用的局部变量 `result` 和形式参数 `n` 是不同的，不会发生覆盖。

当调用 `factorial(2)` 时，程序将执行语句：

`result=2*factorial(1)`；这时又会先计算 `factorial(1)` 而将 2 乘 1 的阶乘运算放在一边。

当调用 `factorial(1)` 时，程序将执行语句：

`result=1*factorial(0)`；而将 1 乘 0 的阶乘运算放在一边。

而当调用 `factorial(0)` 时，函数将 `result` 赋值为 1，并返回，然后开始计算那一串悬而未决的表达式。`factorial(0)` 的值 1 将返回到发出调用的 `factorial(1)` 中，在那里 `result` 将被赋值 `1*factorial(0)` 即 `1×1`，然后再返回发出调用的 `factorial(2)` 中，在那里 `result` 将被赋值 `2*1*factorial(1)`，即 `2×2×1`，并返回发出调用的 `factorial(3)` 中，同样在那里 `result` 将被赋值 `3×factorial(2)`，即 `3×2×1×1`，并返回主调用函数。这样就完成了 `factorial()` 函数的全部递归调用运算。

6. 用函数指针变量调用函数

前面在学习指针的概念时，可知可以用指针变量指向变量、字符串、数组。其实，也可以用指针变量指向一个函数，即可以用函数的指针变量来调用函数。

一个函数在编译时，C 语言编译器会给它分配一个入口地址，这个入口地址就称为函数的指针。程序中可以用一个指针变量指向函数，然后通过该指针变量调用此函数。

下面仍以前面讲述的递归调用程序为例，来讨论采用函数指针变量调用函数。

`main` 函数中的 “`c=factorial(j)`；” 中包含了一次函数调用（调用 `factorial()` 函数）。每一个函数都占用一段内存单元，它们有一个起始地址。因此可以定义一个指针变量，把它指向 `factorial()` 函数，并通过该指针变量来访问 `factorial()` 函数。基于这种思想，可以将 `main()` 函数改写如下：

```
main()
{
    int j, c;
    int (*p)();    /*函数指针变量定义*/
    p=factorial;   /*函数指针变量 p 指向 factorial()函数*/
    for(j=0; j<11; j++)
    {
        c=(*p)(j); /*用指针变量 p 调用 factorial 函数*/
        printf( "%2d!=%ld\n", j, c);
    }
}
```

程序中 `int(*p)()` 是函数指针定义语句，说明 `p` 是一个指向函数的指针变量，此函数的返回值类型为整型 `int`。注意，`(*p)()` 不能写成 `*p()`，因为 `*p` 两边的括号不能省略，表示 `p` 先与 `*` 号结合，是指针变量，然后再与 `()` 结合，表示此指针变量指向函数。如果写成 `*p()`，由于 `()` 优先级高于 `*` 号，则表示 `p` 为一个函数，该函数的返回值类型是指针。

语句 `p=factorial` 的作用是将函数 `factorial()` 的入口地址赋给指针变量 `p`，此后调用 `(*p)(j)` 就是调用 `factorial(j)` 函数。

在 `main()` 中赋值语句 `c=(*p)(j)`，包含着函数调用，它和 `c=factorial(j)` 是等价的，这就是采用函数指针变量来调用函数。

通过上面的讨论，对用函数的指针变量调用函数可归纳为如下几点。

(1) 指向函数的指针变量的一般定义形式为：

函数值返回类型(*指针变量名)(函数形参表……)

`int(*p)()` 表示定义一个指向函数的指针变量，若不对它进行赋值，它便不能固定指向某一个函数，而只是表示定义了一个函数指针类型的变量，它是专门用来存放函数的入口地址的。在程序中若把哪个函数的地址赋给它，它就指向哪个函数。在一个程序中，通过不同的赋值，一个函数指针变量可以先后指向不同的函数。

(2) 在给函数指针变量赋值时，只需给出函数名而不必给出参数，例如 `p=factorial`。

(3) 用函数指针变量调用函数时，只需将 `(*p)` 代替函数名即可(`p` 为指针变量)，在 `(*p)` 之后的括号中可根据需要写上实际参数。例如 `c=(*p)(j)` 表示调用由 `p` 指向的函数，实参为 `j`，返回的函数值赋给 `c`。

(4) 对指向函数的指针变量进行诸如 `p+n`，`p++`，`p--` 的运算是没有意义的。

3.5.5 数组/指针作为函数的参数

1. 用数组作为函数的参数

前面已经讨论过如何用变量作为函数的参数，其实也可以用数组元素，乃至整个数组作为函数的参数。

【例 3-21】 编写一个 `min` 函数，以便在含有 10 个整数元素的数组中找出最小值。

程序如下：

```
int min(values)
int values[10];
{
    int minimum, i;
    for(i=1; i<10; i++)
        if(values[i]<minimum)
            minimum=values[i];
    return(minimum);
}
main()
{
    int i, minimum_score;
    int scores[10]={6, 5, 7, 9, 1, 4, 2, 8, 0, 3}
    for(i=0; i<10; i++)
```



```

    printf( "scores[%d]=%d\n", i, scores[i]);
    minimum_score=min(scores);
    printf( "\n minimum_score is %d\n", minimum_score);
}

```

程序运行结果:

scores[0]=6

scores[1]=5

scores[2]=7

scores[3]=9

scores[4]=1

scores[5]=4

scores[6]=2

scores[7]=8

scores[8]=0

scores[9]=3

minimum_score is 0

在程序中, 主函数 `main()` 定义了一个 10 元素的数组 `scores[10]`, 并进行初始化赋值。在语句 `minimum_score=min(scores)` 中调用函数 `min(scores)`, 同时将数组 `scores[]` 作为被调用函数的实际参数。每当 `min()` 被调用时, 实际参数 `scores[10]` 的全部元素传递给被调用函数的形式参数 `value[10]` 数组。`min()` 函数的作用是将数组元素中最小的值找出来, 并作为函数的返回值, 返回给调用函数。

通过以上讨论, 对用数组作为函数的参数可归纳为以下几点。

(1) 当用数组名作函数的参数时, 应该在调用函数和被调用函数中分别定义数组。例如上例中, `values` 是形式参数数组名, `scores` 是实际参数数组名, 应分别在其所在的函数中定义, 不能只在一方定义。只有这样, 作为调用函数的实际参数数组的全部元素, 才能顺利地传递到被调用函数的形式参数数组中。

(2) 实参数组与形参数组的类型应一致, 否则将导致结果出错。

(3) 实参数组与形参数组的大小可以一致, 也可以不一致。C 语言编译器对形参数组的大小不作检查, 只是将实参数组的地址传给被调用函数。如果要求形参数组得到实参数组的全部元素值, 则应使形参数组和实参数组的大小保持一致, 形参数组不应大于实参数组。

2. 用指向函数的指针变量作为函数的参数

函数指针变量常用的用途之一是把指针作为参数传递给其他函数。下面通过一个实例简述其基本原理。

设一个函数 `sub()` 有两个形式参数(`x1`, `x2`), 定义 `x1` 和 `x2` 为指向函数的指针变量。在调用函数 `sub` 时, 实际参数用两个实用函数的函数名 `f1` 和 `f2` 给形式参数传递函数地址。这样在函数 `sub` 中就可以调用 `f1` 和 `f2` 了。例如:

```

main()
{

```

```

.....
sub(f1, f2);    /*f1, f2 为函数名*/
.....
}
sub(x1, x2);
int (*x1)(), (*x2)();
{
    int a, b, i, j;
    a=(*x1)(i);
    b=(*x2)(i, j);
}

```

其中，变量 i 和 j 是函数 f1 和 f2 所要求的参数。函数 sub() 中的函数指针变量 x1, x2 在函数未被调用时并不占内存单元，也不指向任何函数。在调用 sub() 函数时，把实参函数 f1 和 f2 函数的入口地址传给形式参数指针变量 x1 和 x2，使 x1 指向函数 f1，x2 指向函数 f2，这时在函数 sub() 中，就可以使用 *x1 和 *x2 来调用 f1 和 f2 了。(*x1)(i) 等价于 f1(i)，而 (*x2)(i, j) 等价于 f2(i, j)。

【例 3-22】 设一个函数 process，在调用它时每次实现不同的功能。对于主程序的两个变量值 a 和 b，第一次调用 process 时，找出 a, b 中的最大值；第二次找出其中最小者；第三次求 a, b 之和。

程序如下：

```

main()
{
    int max(), min(), add();
    int a=6, b=2;
    printf("a=%d b=%d\n", a, b);
    printf("max=");
    process(a, b, max);
    printf("min=");
    process(a, b, min);
    printf("sum=");
    process(a, b, add);
}

max(x, y)
int x, y
{
    int z;
    if(x>y) z=x;
    else z=y;
    return(z);
}

```

```

}
min(x, y)
int x, y;
{
    int z;
    if(x<y) z=x;
    else z=y;
    return(z);
}
add(x, y)
int x, y;
{
    int z;
    z=x+y;
    return(z);
}
process(x, y, fun)
int x, y;
int(*fun)();
{
    int result;
    result=(*fun)(x, y);
    printf("%d\n", result);
}

```

程序运行结果:

```

a=2 b=6
max=6
min=2
sum=8

```

在程序中, max()、min()和 add()是三个已定义的函数, 分别用来实现求最大、最小、以及求和的功能。

main()函数在第一次调用 process 函数时, 除了将 a、b 作为实际参数把两个变量值传给 process 的形式参数 x、y 之外, 还将函数名 max 作为实际参数, 将其入口地址传送给 process 函数中的形式参数——指向函数的指针变量 fun, 这样在 process()中, 语句(*fun)(x, y)等价于 max(x, y)。同理, 第二次调用 process(x, y, min)函数时, (*fun)(xy)等价于 min(x, y)。第三次调用 process(x, y, add)函数时, (*fun)(x, y)等价于 add(x, y)。

可见, 不论执行 max()或 min(), 还是 add(), 函数 process()中的内容一点没有改变, 只是在调用 process()函数时实际参数——函数名表改变而已。因此, 用指向函数指针变量作为函数的参数方法, 可以使用同一个函数来完成不同的函数功能, 这样就大大增强了函数使用的

灵活性。

3. 用指向结构的指针变量作函数的参数

前面曾经提到，C 语言不允许整体引用结构变量名，那么如何将一个结构变量的值从一个函数传递给另一个函数呢？通常有以下两种方法：

(1) 像用变量作为函数的参数一样，直接引用结构体变量的成员作参数。

(2) 用指向结构变量的指针变量作实际参数，将结构变量的地址传递给被调用函数的形式参数。

下面举例说明第二种方法。

【例 3-23】 在结构变量 record 中，存放着一个记录，内含记录号、记录名称和三项数据。要求在 main() 函数中，对该结构进行赋值，而在另一个函数 show 中将它们打印出来。

程序如下：

```
struct record
{
    int number;
    char name[20];
    float score[3];
};
void show(p)
struct record *p;
{
    int i;
    printf("note.number=%d\n", p->number);
    printf("note.name=%S\n", p->name);
    for(i=1; i<3; i++)
        printf("note.score[%d]=%f\n", i, p->score[i]);
    printf("\n");
}
main()
{
    void show();
    struct record note;
    note.number=3361;
    note.name= "VGK";
    note.score[0]=11.25;
    note.score[1]=15.15;
    note.score[2]=20.05;
    show(&note);
}
```

程序运行结果:

```
note.number=3361
note.name=VGK
note.score[0]=11.25
note.score[1]=15.15
note.score[2]=20.05
```

在主函数main()中,对struct record类型的结构变量note的各个成员进行赋值;而show(&note)则是调用show()函数,并用&note作为实际参数,&note是struct record结构变量note的地址。在调用show(p)函数时,地址&note被传递给show的形式参数p,p是一个指向struct record类型结构变量的指针,这样p就指向note,在show()函数中p指向的结构体变量的各个成员p->number, p->name, p->score[i]就是note的成员note.number, note.name, note.score[i]。

在结构类型成员多的情况下,同一个一个地直接引用结构变量成员作为函数参数传递相比,用指向结构变量的指针变量作为函数的参数,时间省、效率高。

4. 返回指针的函数

一个函数可以返回一个整型(int)值,字符(char)值和浮点(float)值,同样也可以返回一个指针型数据,即返回一个数据的地址。

这种返回指针值的函数的一般定义形式为:

返回值类型 *函数名 (参数表)

例如:

```
int *A(x, y);
```

其中A是函数名,调用它后可以得到一个指向整型数据的指针(地址)。在A的两侧,分别是*运算符和()运算符。由于()优先级高于*,故A先与()结合,A()显然是函数形式。A()前面的*表示此函数是指针型函数(函数的返回值是指针)。

【例 3-24】有一个数据档案记录,每个记录中含有四项数据,要求用返回指针的函数来实现。

程序如下:

```
float *view(pp, j)
float(pp)[4];
int j;
{
    float *pt;
    pt=*(pp+j);
    return(pt);
}
main()
{
    float score[][4]={{8.87, 8.83, 8.64, 8.91},
    {12.11, 12.23, 12.34, 12.20},
```

```

    {30.44, 30.66, 30.58, 30.40},
    {20.12, 20.08, 20.21, 20.66}};
float *view;
float *p;
int i, n=2;
printf("The record of No.%d are: \n", n)
p=view(score, n);
for(i=0; i<4; i++)
printf("%5.2f\t", *(p+i));
}

```

程序运行结果:

```

The record of No.2 are
30.44  30.66  30.58  30.40

```

在上面程序中, 函数 `view` 被定义为指针型函数, 它的形式参数 `pp` 是指向包含 4 个元素的一维数组的指针变量。`pp+1` 指向 `score` 数组的第一行, 而 `*(pp+1)` 指向 `score` 数组的第一行的第 0 列元素。

在主函数 `main()` 中, `p` 是 `float` 型指针变量, 在没有执行调用赋值语句 `p=view(score, n)` 之前, 它没有指向任何数据。而在执行 `p=view(score, n)` 时, 首先程序先调用 `view()` 函数, 并把实际参数 `score[]` 和 `n` 值传递给 `view()` 函数的形式参数 `*pp[4]` 和 `j`, 然后通过语句 `pt=*(pp+j)` 将 `view()` 函数的内部指针 `pt` 指向 `score[][]` 数组的第 `j` 行地址, 并将此指针值返回给主调用函数, 继而该地址被赋给指针变量 `p`。此时, `p` 指针指向 `score[][]` 数组的第 `n` 行, 然后这个记录行的四项记录数据就被打印出来。

3.5.6 局部变量和全局变量

在前面的各个例子中, 变量都是在函数内部定义的, 各个函数内部的变量可以同名而互不影响, 这些变量称为内部变量, 又称局部变量。函数的形式参数也属于局部变量。C 语言程序中, 还允许在函数外面定义变量, 在函数外面定义的变量称为外部变量, 又称全局变量。

全局变量与局部变量的区别在于它们的作用域。每个函数都能使用全局变量; 而局部变量只能被定义它的函数使用, 不能被其他函数使用。也就是说, 全局变量对所有函数都是可见的, 局部变量只对定义它的函数才是可见的。全局变量和局部变量的另一个区别在于: 全局变量被定义时若没有初始化, 其值为 0; 而局部变量被定义时若没有初始化, 则它的值是不确定的。

在一个函数内部, 当一个局部变量与一个全局变量同名时, 全局变量不起作用, 局部变量起作用。

【例 3-25】 分析以下程序。

```

int a=3, b=5;                /*全局变量 a、b*/
int max(int a, int b)        /*局部变量 a、b, 全局变量 a、b 被屏蔽*/
{

```

```

    int c;
    c=a>b? a:b;          /*a=8, b=5, c=8*/
    return c;
}
main()
{
    int a=8;              /*局部变量 a, 全局变量 b=5, 全局变量 a 被屏蔽*/
    printf("%d" max(a, b)); /*输出结果: 8*/
}

```

局部与全局的概念是相对的。在一个函数内部，也有局部与全局之别，例如，可以在复合语句中定义变量，这些变量只在复合语句中有效。

当一个函数要使用一个全局变量，而这个变量是在该函数的后面定义的，应该在函数中对该变量作声明，以向编译程序说明该变量是一个已被定义的外部变量。声明形式为：

extern 类型说明符变量名；

当一个文件中的函数要使用另一个文件中定义的全局变量时，也必须用 **extern** 作外部变量声明。当然，这两个文件都是同一个程序的模块。

【例 3-26】 分析以下程序。

```

int max(int a, int b)      /*局部变量 a、b*/
{
    int c;
    c=a>b? a:b;
    return c;
}
main()
{
    extern int a, b;        /*全局变量 a、b 的申明*/
    printf("%d", max(a, b));
}
int a=13, b=5;             /*全局变量 a、b 的定义*/

```

全局变量提供了函数之间数据交换的途径。但是，由于各函数都能修改全局变量的值，这样会产生隐患。因此，一般不提倡使用全局变量。

3.6 C51 的库函数

C51 编译器提供了丰富的库函数，使用库函数大大提高了编程的效率，用户可以根据需要随时调用。每个库函数都在相应的头文件中给出了函数的原型，使用时只需在源程序的开头用编译预处理命令 **#include** 将相关的头文件包含进来即可。下面就一些常用的 C51 库函数分类做一些介绍。

1. 字符函数库 CTYPE.H

该函数库提供了一组关于字符处理的函数，主要的函数原型和功能如下：

(1) extern bit isalpha(char);

检查参数字符是否为英文字母，是则返回 1，否则返回 0。

(2) extern bit isalnum(char);

检查参数字符是否为英文字母或数字字符，是则返回 1，否则返回 0。

(3) extern bit iscntrl(char);

检查参数字符是否为控制字符，即 ASCII 值为 0x00~0x1f 或 0x7f 的字符，是则返回 1，否则返回 0。

(4) extern bit islower(char);

检查参数字符是否为小写英文字母，是则返回 1，否则返回 0。

(5) extern bit isupper(char);

检查参数字符是否为大写英文字母，是则返回 1，否则返回 0。

(6) extern bit isdigit(char);

检查参数字符是否为数字字符，是则返回 1，否则返回 0。

(7) extern bit isxdigit(char);

检查参数字符是否为 16 进制数字字符，是则返回 1，否则返回 0。

(8) extern char toint(char);

将 ASCII 字符的 0~9、a~f（大小写无关）转换为 16 进制数字。

(9) extern char toupper(char);

将小写字母转换成大写字母，如果字符不在 ‘a’ ~ ‘z’ 之间，则不作转换直接返回该字符。

(10) extern char tolower(char);

将大写字母转换成小写字母，如果字符不在 ‘A’ ~ ‘Z’ 之间，则不作转换直接返回该字符。

2. 标准函数库 STDLIB.H

(1) extern float atof(char *s);

将字符串 s 转换成浮点数值并返回它。参数字符串必须包含与浮点数规定相符的数。

(2) extern long atol(char *s);

将字符串 s 转换成长整型数值并返回它。参数字符串必须包含与长整型数规定相符的数。

(3) extern int atoi(char *s);

将字符串 s 转换成整型数值并返回它。参数字符串必须包含与整型数规定相符的数。

(4) void *malloc(unsigned int size);

返回一块大小为 size 个字节的连续内存空间的指针。如返回值为 NULL，则无足够的内存空间可用。

(5) void free(void *p);

释放由 malloc 函数分配的存储器空间。

(6) void init mempool(void *p, unsigned int size);

清零由 malloc 函数分配的存储器空间。

3. 数学函数库 MATH.H

(1) extern int abs(int val);

extern char abs(char val);

extern float abs(float val);

extern long abs(long val);

计算并返回 val 的绝对值。这四个函数的区别在于参数和返回值的类型不同。

(2) extern float exp(float x);

返回以 e 为底的 x 的幂，即 e^x 。

(3) extern float log(float x);

extern float log10(float x);

log 返回 x 的自然对数，即 $\ln x$ ；log10 返回以 10 为底的 x 的对数，即 $\log_{10} x$ 。

(4) extern float sqrt(float x);

返回 x 的正平方根，即 $\sqrt{x}$ 。

(5) extern float sin(float x);

extern float cos(float x);

extern float tan(float x);

sin 返回值为 $\sin(x)$ ；cos 返回值为 $\cos(x)$ ；tan 返回值为 $\tan(x)$ 。

(6) extern float pow(float x, float y);

返回值为 x^y 。

4. 绝对地址访问头文件 ABSACC.H

(1) #define CBYTE((unsigned char *)0x50000L)

#define DBYTE((unsigned char *)0x40000L)

#define PBYTE((unsigned char *)0x30000L)

#define XBYTE((unsigned char *)0x80000L)

用来对 C51 系列单片机的存储器空间进行绝对地址访问，以字节为单位寻址。CBYTE 寻址 CODE 区；DBYTE 寻址 DATA 区；PBYTE 寻址 XDATA 的 00H~FFH 区域（用指令 MOVX @R0, A）；XBYTE 寻址 XDATA 区（用指令 MOVX @DPTR, A）。

(2) #define CWORD((unsigned int *)0x50000L)

#define DWORD((unsigned int *)0x40000L)

#define PWORD((unsigned int *)0x30000L)

#define XWORD((unsigned int *)0x20000L)

与前面的宏定义相同，只是数据为双字节。

5. 内部函数库 INTRINS.H

(1) unsigned char _crol_(unsigned char val, unsigned char n);

```
unsigned int _lrol_(unsigned int val, unsigned char n);
unsigned long _lrol_(unsigned long val, unsigned char n);
```

将变量 val 循环左移 n 位。

```
(2) unsigned char _rrol_(unsigned char val, unsigned char n);
    unsigned int _rrol_(unsigned int val, unsigned char n);
    unsigned long _rrol_(unsigned long val, unsigned char n);
```

将变量 val 循环右移 n 位。

```
(3) void _nop_(void);
```

该函数产生一个 C51 单片机的 NOP 指令，用于延时一个机器周期。

例如：

```
P10=1;
_nop();    /*等待一个机器周期*/
P10=0;
```

```
(4) bit _testbit_(bit x);
```

测试给定的位参数 x 是否为 1。若为 1，返回 1，同时将该位复位为 0；否则返回 0。

6. 访问 SFR 和 SFR_bit 地址头文件 REGxxx.H

头文件 reg51.h、reg52.h 等文件中定义了 C51 单片机中的 SFR 寄存器名和相关的位变量名。

3.7 编程举例

循环队列是一种先入先出（FIFO）存储结构，在单片机应用程序中经常使用。队列需要队头指针 listhead、队尾指针 listtail、队列长度 listlen、队列空标志 listempty 和队列满 listfull 标志。队头指针 listhead 总指向队列中的第一个数据，队尾指针 listtail 总指向队列当前的空位置。队列可以用数组实现，数组长度为 listlen。listempty 标志若为 1，表示队列中没有数据，即队列为空；若为 0，表示队列有数据，即队列非空。listfull 标志若为 1，表示队列已满；若为 0，表示还能向队列写入数据，即队列非满。

初始时，listhead=0，listtail=0，listempty=1，listfull=0。需要定义两个函数 listwrite() 和 listread()，分别对队列进行读写操作。

listwrite() 函数的操作思路：

```
if(队列满) //退出;
else
{
    //将数据写入 listtail 指向的数组单元;
    listtail++;
    if(listtail==listlen) listtail=0;
    listempty=0;
    if(listhead==listtail) listfull=1;
```

```

}
listread()//函数的操作思路:
if(队列空) //退出;
else
{
    //将 listtail 指向的数组单元的内容读出;
    listhead++;
    if(listhead==listlen) listhead=0;
    listfull=0;
    if(listtail==listhead) listempty=1;
}

```

函数的实现:

```

#define listlen 10;
unsigned char list[listlen];
char listwrite(char x)
{
    if(listfull) return 0;
    list[listtail]=x;
    listtail++;
    if(listhead==listlen) listtail=0;
    listempty=0;
    if(listhead==listtail) listfull=1;
    return 1;
}
char listread(char *x)
{
    if(listempty) return 0;
    *x=list[listhead];
    listhead++;
    if(listtail==listlen) listhead=0;
    listfull=0;
    if(listtail==listhead) listempty=1;
    return 1;
}

```

习题和思考题

3-1 C语言中的类型是怎么分配的? 什么是赋值操作?

3-2 按给定的存储类型和数据类型，写出下列变量的说明形式。

- (1) up, down, 整数, 使用堆栈存储;
- (2) first, last, 浮点小数, 使用外部数据存储器存储;
- (3) cc, ch, 字符, 使用内部数据存储器存储。

3-3 判断下列关系表达式或逻辑表达式的运算结果 (1 或 0)。

- (1) 10==9+1; (2) 0&&0; (3) 10&&8; (4) 8||0;
- (5) !(3+2); (6) 设 x=10, y=9, x>=88&&y<=x。

3-4 设 x=4, y=8, 说明进行下列运算后, x, y 和 z 的值分别是多少?

- (1) z=(x++)*(--y); (2) z=(++x)-(y--);
- (3) z=(++x)*(--y); (4) z=(x++)+(y--)

3-5 分析下列运算表达式运算顺序。

- (1) c=a|| (b); (2) x+=y-z; (3) -b>>2;
- (4) c=++a%b--; (5) !m&n; (6) a<b||c&d。

3-6 C 语言中的 while 和 do...while 的不同点是什么?

3-7 用选择分支编写程序, 把输入的一个数字, 按下列对应关系显示。当输入 ^Z 时, 程序结束。

输入数字	显示
1	A
2	B
3	C
4	D
其他	?

3-8 用三种循环方式分别编写程序, 显示整数 1 到 100 的平方。

3-9 写出二维数组 Data[2][4]的各个元素, 并按它们在内存中存储时的顺序排列。

3-10 设下列运算表达式中 p 是指针, 试分析各表达式的运算顺序。

- (1) b=*p--; (2) x=*p++;
- (3) a[++i]=*p++; (4) y=*--*++p-6。

3-11 使用 union 的目的是什么? 定义 union, 它用于容纳下列数据:

```
int    data[4];
char   ch[8];
float  f。
```

3-12 C 语言中函数有什么特性? 函数的存储类型和数据类型的意义是什么?

3-13 当一个函数需要返回多于一个值时, 可以怎么做?

3-14 编写把十六进制字符串 s 转换成整数值返回的函数 htoi(s)。

3-15 编写把字符串 s 逆转的函数 reverse(s)。

3-16 把上题的 reverse(s)编写成递归函数。

第 4 章 C51 单片机的内部资源

重点及难点

单片机中断和定时与单片机通信的基本概念、单片机的中断系统、单片机的定时/计数器、单片机外部中断源的扩展、数据传递的方式、串行通信控制寄存器、MCS-51 串行通信工作方式及其应用。

基本要求

1. 掌握单片机中断和定时与单片机通信的基本概念
2. 掌握单片机的中断源、中断控制、中断响应过程的基本概念及单片机中断系统的功能和使用方法
3. 掌握单片机的定时/计数器的初值计算、工作方式控制寄存器的初始化、程序的设计方法和步骤
4. 串行通信的基本方式、数据传送的次序、串行通信控制寄存器的每一位的定义、串行通信的四种工作方式
5. 了解单片机外部中断源扩展的常用方法和步骤

教学内容

1. 中断系统
2. 定时/计数器
3. 串行通信接口

4.1 输入/输出控制方式

输入和输出设备（简称 I/O 设备）是微机系统的重要组成部分。各种数据、程序及现场采样信号，都要通过输入设备送入计算机；处理结果及各种控制信号需要输出到输出设备，以便显示、打印和实现各种控制动作。常用的输入设备有键盘、鼠标器、扫描器或 A/D 转换器等；常用的输出设备有显示器、打印机、绘图仪或 D/A 转换器等。输入/输出设备统称为外部设备。外部设备的种类繁多，有机械式、电子式或其他形式，外设信息也具有多种形式，有数字量、模拟量或开关量。由于这些设备向微机输入数据或接收微机输出数据的速度、数据格式不相匹配，两者的电路时序也可能不一样。因此在微机和外设之间必须通过接口电路相连接，以使双方动作协调配合。

通常,微处理器通过总线与接口电路相连接,而接口电路则与外部设备相连接,因此 CPU 总是通过接口电路与外部设备发生联系。从控制的角度看,微机的接口电路是用来控制数据或信息传送的,接口控制即为接口电路对微处理器与外设之间数据传送的控制。一般的接口电路控制数据信息的传送是在中央处理器的监控下实现的。对中央处理器而言,数据传送就是输入/输出操作,常采用三种控制方式实现数据传送,即程序控制方式、中断控制方式和 DMA 方式。单片机常用的是程序控制方式和中断控制方式,本节主要讲一下这两种方法。

4.1.1 程序控制方式

程序控制方式的特点是 CPU 与外部设备交换信息完全在用户程序的控制下进行。什么时候在哪个端口进行输入/输出操作,均在程序中反映出来。程序控制方式有以下两种形式。

1. 无条件传送方式

无条件传送方式是最简单的程序控制方式。在无条件传送时, CPU 无需询问外设是否准备好(即认为外设一定准备好了)便通过程序向指定的端口执行输入/输出操作。采用这种方式所需的硬件和软件较为简单,输入/输出操作完全取决于程序的安排,而不管外设的状态。

这种控制方式在实际应用中还是比较常见的。例如,微处理器或单片机可以通过某一端口(对应的端口有一地址)执行输出操作,来控制诸如发光二极管驱动接口、继电器驱动接口或步进电机功率接口等,使相应的执行部件进行动作。

2. 程序查询方式

上述无条件传送方式虽然比较简单,但使用上有局限性,它不了解外设的实时状态。有的外设不一定与计算机同步,或者不能及时响应计算机的输入/输出操作,从而很难保证操作的正确性。例如,一个键盘输入设备,只有确认键盘上按下了一个键,程序才能从连接键盘的端口读取信息,否则读取的信息可能就是上次按下的键,或者是任意不确定的信息。同样,对于输出操作也有类似的情况,当要输出一个信息时,程序必须确认上次送入端口的信息已经被输出设备取走,输出端口是“空”的,否则就要覆盖上次输出的信息。

程序查询方式是指在执行一个输入/输出传送操作之前, CPU 首先要去查询 I/O 设备的当前状态,获得端口的状态信息,看它是否准备就绪。对输入操作,查询是否有新的数据送入端口,即准备好否;对输出操作,查询是否已完成上次的输出任务,即设备忙否。如果外设未准备好,就进入等待循环;如果已经准备好,就执行数据传送。程序查询控制方式的流程图如图 4-1 所示。当有多个 I/O 设备时,可采用逐个查询的方法。

这种方式的特点是: I/O 操作完全由 CPU 控制,即由 CPU 执行启动、控制以及输入/输出操作。其优点是硬件电路比较简单;缺点是 CPU 要不断检测外设状态,直至外设准备好后才传送数据。这在与慢速外设进行 I/O 传送时,不仅浪费了 CPU 的时间,降低了主机的效率,而且在很多控制过程中也是行不

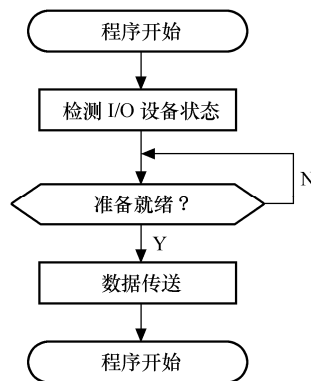


图 4-1 程序查询方式的流程图

通的。

4.1.2 中断控制方式

为了克服程序查询的缺点,可以采用中断控制方式。中断方式可使 CPU 不必定时查询接口状态,而由接口在数据传送完毕或接收数据准备好后通知 CPU, CPU 通过执行一个中断服务程序来完成数据的传送。而当接口没有准备好时, CPU 可以继续执行主程序,从而大大提高了 CPU 的工作效率。

1. 中断的概念

所谓中断是指 CPU 在正常运行程序时,由于 CPU 以及某一事件的发生(例如一个电平的变化、一个脉冲沿的产生或定时/计数器溢出等),引起了 CPU 中断正在运行的程序,而转去执行预先安排的该事件服务程序。这些引起程序中断的事件称为中断源。不同的机器其中断源也是不一样的。中断源要求为其服务的请求称为中断请求(或申请),例如某些接口设备所发出的请求中断服务程序去处理的信号。中断请求信号何时发生不是预先知道的,但它们一旦产生,便会马上通知 CPU,这样 CPU 就无需花费大量的时间去查询这些信号是否存在。当 CPU 接到中断请求信号后,便立即转去执行相应的中断服务程序,服务程序执行完后, CPU 便返回原来所执行程序的中断处(称为断点)继续往下执行。中断流程示意图如图 4-2 所示。

在实际应用中,常常会有多个中断源请求中断服务,因此 CPU 响应这些中断就有个先后次序问题,这称为中断的优先级。CPU 首先响应优先级高的中断。

在执行中断服务程序时,若产生了优先权较高的中断请求,这时 CPU 就暂停正在处理的优先权较低的中断,转去响应优先权较高的中断,等具有较高优先权的中断处理完毕后,再返回继续执行原来的中断处理程序。在执行某一中断服务程序时,如果有低级或同级中断申请, CPU 均不响应。上述这种处理过程称为中断嵌套。具有中断嵌套功能的中断系统称为多级中断系统,否则称为单级中断系统。8051 的五个中断源有两个优先级,可实现二级中断嵌套,二级中断嵌套的流程图如图 4-3 所示。

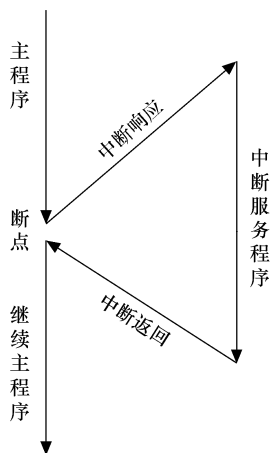


图 4-2 中断流程示意图

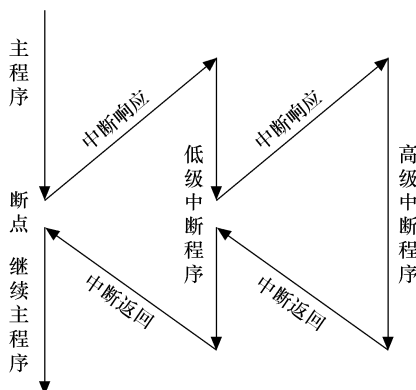


图 4-3 中断嵌套流程图

2. 中断传送方式

一个完整的中断传送方式的处理过程应包括中断请求、中断优先权判别、中断响应、中断处理、中断返回等 5 个步骤。

(1) 中断请求

中断源向 CPU 发出中断申请，这是引起中断的第一步。不同的中断源将根据不同的原因产生中断请求，例如，输入设备输入一个字符后向 CPU 发出中断申请，请求读取数据；输出设备从端口取走一个字符后发出中断申请，请求发送下一个数据等。中断请求信号一经产生，将一直保持着，直到 CPU 响应该中断后，才清除中断请求信号。

(2) 中断优先权判别

由于每个中断源向 CPU 发出的中断信号是随机的，因而有时会出现多个中断源同时向 CPU 发出中断请求，但 CPU 每次只能响应其中的一个中断，CPU 究竟响应哪一个中断，实际上是系统根据中断源的轻重缓急，给每一个中断源规定了一个中断级别，即优先权。一般任务最紧迫、速度最高的设备优先权最高。实现中断优先权的判别一般有硬件判优和软件判优两种方法。

① 软件判优

一般采用查询技术实现软件判优。当 CPU 响应中断后，用软件查询来确定当前提出中断申请的优先权最高的中断源。软件判优的特点是查询的次序就是中断源优先权的次序。采用这种方法可节省硬件开销，但实现速度较慢。

② 硬件判优

一般采用优先权排队电路等硬件方式来实现中断优先权的判别。例如中断控制芯片 8259 等都具有这种功能。

(3) 中断响应

当 CPU 得到经中断判优送出的中断请求信号后，要等到出现适当的条件才能响应中断。这些条件一般包括：当前的一条指令已执行完毕而不是正在执行中，且中断允许寄存器中的有关位置“1”（即 CPU 开中断）。满足上述条件后，CPU 才响应中断，进入中断响应周期。在中断响应周期完成将程序中断处的断点地址压入堆栈保护起来，然后将对应的中断服务程序入口地址装入程序计数器 PC，控制程序转移到相应的中断服务程序去执行。

(4) 中断处理

中断处理是由中断服务子程序来完成的。进入中断服务子程序后，首先要做的就是保护现场，即把断点处的寄存器及标志寄存器的有用内容压入堆栈，以便不影响主程序的运行，然后再执行中断服务程序。在中断处理结束前又要恢复现场，即把堆栈中的内容恢复到各寄存器。

在中断处理进行的过程中可能又有新的中断请求到来，但保护现场和恢复现场的操作是不允许打扰的，否则就会造成原有内容的破坏。为此在保护现场和恢复现场之前要先关闭中断系统，以屏蔽其他中断请求。待保护现场和恢复现场完成后，为了使系统具有中断嵌套的功能，再开放中断系统。

(5) 中断返回

每一个中断服务子程序的末尾都要使用中断返回指令，该指令的功能是恢复断点地

址。待返回指令执行完毕后，便返回到被中断了的程序，从断点地址开始继续向下执行原程序。

4.2 中断系统

4.2.1 中断的基本概念

实现中断功能的硬件系统和软件系统称为中断系统，中断系统是计算机的重要组成部分。实时控制、故障自动处理时往往用到中断系统，计算机与外部设备间传送数据及实现人机联系也常常采用中断方式。

中断系统需要解决以下几个基本问题。

1. 中断源：中断请求信号的来源。包括中断请求信号的产生及该信号怎样被 CPU 有效地识别；而且要求中断请求信号产生一次，只能被 CPU 接收处理一次，即不能一次中断申请被 CPU 多次响应。这就涉及到中断请求信号的及时撤除问题。

2. 中断响应与返回：CPU 采集到中断请求信号后，怎样转向特定的中断服务子程序及执行完中断服务子程序怎样返回被中断的程序继续正确地执行。中断响应与返回的过程中涉及到 CPU 响应中断的条件、现场保护等问题。

3. 优先级控制：一个计算机应用系统，特别是计算机实时测控应用系统，往往有多个中断源，各中断源所要求的处理具有不同的轻重、缓急程度。与人处理问题的思路一样，希望重要紧急的事件先处理，而且如果当前处于正在处理某个事件的过程中，有更重要、更紧急的事件到来，就应当暂停当前事件的处理，转去处理新事件。这就是中断系统优先级控制所要解决的问题。中断优先级的控制形成了中断嵌套。

4.2.2 中断源

中断源指引起正在执行的程序中断，转而执行中断服务程序的设备或事件。中断源可分为硬中断、软中断。C51 单片机共有 5 个中断源，其中外部中断 2 个，内部中断 3 个。单片机中断系统的结构图如图 4-4 所示，中断源具体情况如表 4-1 所示。

表 4-1 单片机中断源

符 号	名 称	中断引起原因
$\overline{\text{INT0}}$	外部中断 0	P3.2 引脚的低电平或下降沿信号
$\overline{\text{INT1}}$	外部中断 1	P3.3 引脚的低电平或下降沿信号
T0	定时器 0 中断	定时计数器 0 计数回零溢出
T1	定时器 1 中断	定时计数器 1 计数回零溢出
TI/RI	串行口中断	串行通信完成一帧数据发送或接收引起中断

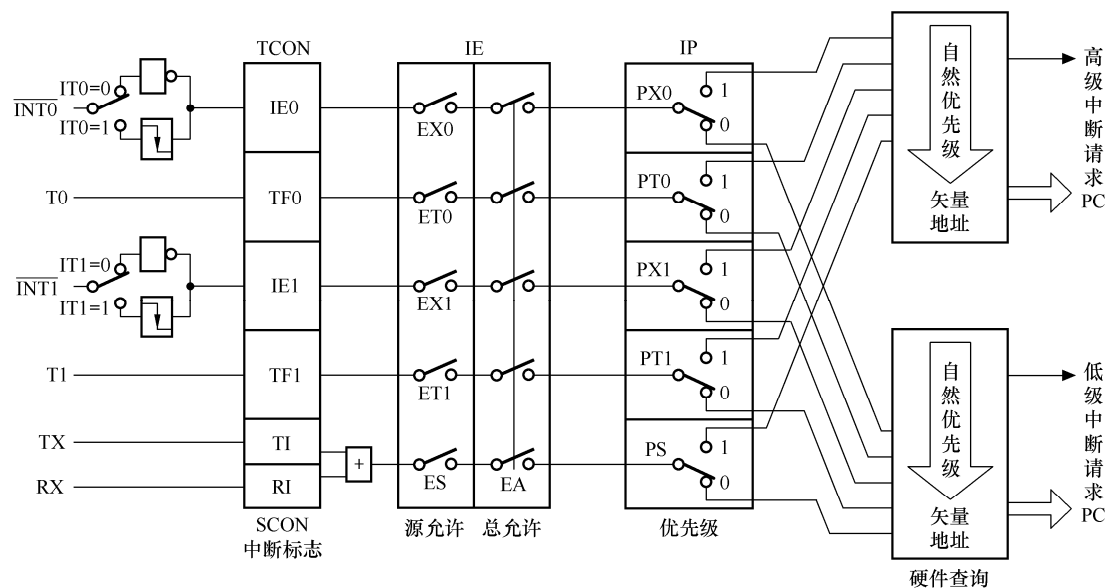


图 4-4 单片机中断系统结构图

1. 外部中断

外部中断是指从单片机外部引脚 $\overline{\text{INT0}}/\text{P3.2}$ 、 $\overline{\text{INT1}}/\text{P3.3}$ 输入中断请求信号的中断，即外部中断源有两个。输入/输出的中断请求、实时事件的中断请求、掉电和设备故障的中断请求都可以作为外部中断源，从引脚 $\overline{\text{INT0}}/\text{P3.2}$ 、 $\overline{\text{INT1}}/\text{P3.3}$ 输入。

外部中断请求 $\overline{\text{INT0}}$ 、 $\overline{\text{INT1}}$ 有两种触发方式：电平触发和跳变（边沿）触发。这两种触发方式可以通过对特殊功能寄存器 TCON 编程来选择。

2. 内部中断

内部中断是单片机芯片内部产生的中断。C51 单片机的内部中断有定时/计数器 T0、T1 的溢出中断以及串行口的发送/接收中断。

4.2.3 中断入口地址

CPU 响应某个中断事件时，将会自动转入固定的地址执行中断服务程序，单片机各个中断源的中断入口地址如表 4-2 所示。

表 4-2 各中断源的入口地址

中 断 源	中断入口地址
外部中断 0 $\overline{\text{INT0}}$	0003H
定时/计数器 0 溢出中断 T0	000BH
外部中断 1 $\overline{\text{INT1}}$	0013H
定时/计数器 1 溢出中断 T1	001BH
串行口中断 TI/RI	0023H

4.2.4 中断控制

1. 定时/计数器控制寄存器 (TCON)

定时/计数器控制寄存器的字节地址为 88H，可进行位寻址。其格式如下所示：

位地址	8FH	8EH	8DH	8CH	8BH	8AH	89H	88H
位符号	TF1	TR1	TF0	TR0	IE1	IT1	IE1	IE0

IT0 (IT1)：外部中断 0 (或 1) 触发方式控制位。IT0 (或 IT1) 被设置为 0，则选择外部中断为电平触发方式；IT0 (或 IT1) 被设置为 1，则选择外部中断为跳变触发方式。

IE0 (IE1)：外部中断 0 (或 1) 的中断请求标志位。当 IT0 (或 IT1) = 0，即电平触发方式时，CPU 在每个机器周期的 S₅P₂ 采样外部中断引脚。若外部中断引脚为低电平，将直接发外部中断。跳变触发方式时，若第一个机器周期采样到外部中断引脚为高电平，第二个机器周期采样到外部中断引脚为低电平时，由硬件置位 IT0 (或 IT1)，并以此向 CPU 请求中断。当 CPU 响应中断转向中断服务程序时由硬件将 IE0 (或 IE1) 清零。

若把外部中断设置为跳变触发方式，CPU 在每个机器周期都采样外部中断引脚。为了保证检测到负跳变，输入到外部中断引脚上的高电平与低电平至少应保持 1 个机器周期。对于电平触发的外部中断，由于 CPU 对外部中断引脚没有控制作用，也没有相应的中断请求标志位，因此需要外接电路来撤除中断请求信号。

TR1、TR0：T1、T0 的启动控制位，置 1 启动，清 0 停止。

TF1、TF0：T1、T0 的溢出标志位。

2. 中断允许控制寄存器 (IE)

C51 单片机中没有专设的开中断和关中断指令，对各中断源的中断开放或关闭是由内部中断允许寄存器 (IE) 的各位来控制的，如果允许该中断源中断则该位置 1，禁止中断则该位置 0。它的字节地址为 A8H，可进行位寻址。IE 各位的定义如下所示：

位地址	0AFH	0AEH	0ADH	0ACH	0ABH	0AAH	0A9H	0ABH
位符号	EA	/	ET2	ES	ET1	EX1	ET0	EX0

EA：中断允许总控位。EA=0，屏蔽所有的中断请求；EA=1，开放中断。EA 的作用是使中断允许形成两级控制，即各中断源首先受 EA 位的控制；其次还要受各中断源自己的中断允许总控位控制。

ET2：定时/计数器 T2 的溢出中断允许位，只用于 52 子系列，51 子系列无此位。ET2=0，禁止 T2 中断；ET2=1，允许 T2 中断。

ES：串行口中断允许位。ES=0，禁止串行口中断；ES=1 允许串行口中断。

ET1：定时/计数器 T1 的溢出中断允许位。ET1=0，禁止 T1 中断；ET1=1，允许 T1 中断。

EX1：外部中断 1 的中断允许位。EX1=0，禁止外部中断 1 中断；EX1=1，允许外部中断 1 中断。

ET0: 定时/计数器 T0 的溢出中断允许位。ET0=0, 禁止 T0 中断; ET0=1, 允许 T0 中断。

EX0: 外部中断 0 的中断允许位。EX0=0, 禁止外部中断 0 中断; EX0=1 允许外部中断 0 中断。

3. 中断优先级控制寄存器 (IP)

C51 单片机的中断源有两个可控的中断优先级, 从而可实现二级中断嵌套。中断系统遵循如下 3 条规则。

(1) 低优先级中断请求不能打断高优先级的中断服务; 但高优先级中断可以中断低优先级的中断服务, 从而实现中断嵌套。

(2) 如果一个中断请求已被响应, 则同级的其他中断服务将被禁止, 即同级不能嵌套。

(3) 如果同级的多个中断请求同时出现, 则按 CPU 查询次序确定哪个中断请求被响应。同级的中断优先级查询次序如表 4-3 所示。

表 4-3 同级的中断优先级

中 断 源	同级的中断优先级
外部中断 0	最高
定时器/计数器 0 中断	次高
外部中断 1	中
定时器/计数器 1 中断	次低
串行口中断	最低

5 个中断源的优先级别由 IP 寄存器管理, 相应位置 1, 则该中断源优先级别高, 置 0 的优先级别低。IP 寄存器的字节地址为 B8H, 可进行位寻址。IP 各位的定义如下所示:

位地址	0BFH	0BEH	0BDH	0BCH	0BBH	0BAH	0B9H	0BBH
位符号	/	/	/	PS	PT1	PX1	PT0	PX0

PS: 串行口的中断优先级控制位。

PT1: 定时/计数器 T1 的中断优先级控制位。

PX1: 外部中断 1 的中断优先级控制位。

PT0: 定时/计数器 T0 的中断优先级控制位。

PX0: 外部中断 0 的中断优先级控制位。

4. 串行口控制寄存器 (SCON)

串行口控制寄存器的字节地址为 98H, 可进行位寻址。SCON 各位的定义如下所示:

位地址	9FH	9EH	9DH	9CH	9BH	9AH	99H	98H
位符号	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

TI: 串行口发送中断请求标志位。

RI: 串行口接收中断请求标志位。

5. 中断控制寄存器汇总

中断控制寄存器汇总如表 4-4 所示。

表 4-4 中断控制寄存器

寄存器名称		D7	D6	D5	D4	D3	D2	D1	D0
定时/计数器控制寄存器	TCON (88H)	TF1	TR1	TF0	TR1	IE1	IT1	IE0	IT0
	位地址	8FH	8EH	8DH	8CH	8BH	8AH	89H	88H
串行口控制寄存器	SCON (98H)	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
	位地址	9FH	9EH	9DH	9CH	9BH	9AH	99H	98H
中断允许寄存器	IE (A8H)	EA	/	/	ES	ET1	EX1	ET0	EX0
	位地址	AFH	AEH	ACH	ACH	ABH	AAH	A9H	A8H
中断优先级寄存器	IP (B8H)	/	/	/	PS	PT1	PX1	PT0	PX0
	位地址	BFH	BEH	BDH	BCH	BBH	BAH	B9H	B8H

4.2.5 中断响应

1. 中断响应的条件

单片机响应中断的条件为中断源有请求（中断允许寄存器 IE 相应位置 1），且 CPU 开中断（即 EA=1）。这样，在每个机器周期的 S₅P₂ 期间，对所有中断源按用户设置的优先级和内部规定的优先级进行顺序检测，并可在 S₆ 期间找到所有有效的中断请求。如有中断请求，且满足下列条件，则在下一个机器周期的 S₁ 期间响应中断，否则将丢弃中断采样的结果。

（1）无同级或高级中断正在处理。

（2）现行指令执行到最后 1 个机器周期且已结束。

（3）若现行指令为 RETI 或访问 IE、IP 的指令时，执行完该指令且紧随其后的另一条指令也已执行完毕。

如有下列情况之一，则中断采样的结果被取消。

（1）CPU 正在处理同级或高级优先级的中断。

（2）现行的机器周期不是所执行指令的最后一个机器周期。

（3）正在执行的指令是 RETI 或访问 IE、IP 指令。CPU 在执行 RETI 或访问 IE、IP 的指令后，至少需要再执行一条其他指令后才会响应中断请求。

2. 中断响应过程

CPU 响应中断后，由硬件自动执行如下的功能操作。

（1）根据中断请求源的优先级高低，对相应的优先级状态触发器置 1。

（2）保护断点，即把程序计数器 PC 的内容压入堆栈保存。

(3) 清内部硬件可清除的中断请求标志位 (IE0、IE1、TF0、TF1)。

(4) 把被响应的中断服务程序入口地址送入 PC，从而转入相应的中断服务程序执行。

中断处理过程分为 4 个阶段：中断请求、中断响应、中断处理、中断返回。C51 系列单片机的中断服务流程如图 4-5 所示。

中断请求、中断响应过程由硬件完成。中断服务程序应根据需要进行编写。程序中要注意保护现场和恢复现场。

中断返回是通过执行一条 RETI 中断返回指令，使堆栈中被压入的断点地址送 PC，从而返回主程序的断点继续执行主程序。另外，RETI 还有恢复优先级状态触发器的作用，因此不能以 RET 指令代替“RETI”指令。

向量中断包括把先前的程序计数指针 (PC) 压入堆栈 (像调用)，中断服务程序很像其他子程序 (有一个返回)。当向量中断发生时，硬件禁止所有中断 (IE 寄存器的 EA 位)，这时表明外部中断或定时溢出的标志位由硬件清除。串行口的标志位不由硬件清除，通常要判别是 RI 或 TI 引起中断，中断服务程序的不同分支取决于中断源。在重新允许全局 CPU 中断 (EA) 之前，必须仔细清除各种标志。标志会引起立即的重复中断。8051 对中断实际上有特殊的返回指令 RETI 而不是 RET，它重新允许系统识别其他中断。因而，没必要在正常使用中断时复位 EA，只要在程序初始化时开中断一次即可。

在处理一个中断时，不能识别同级或更低优先级的其他中断；当高优先级或低优先级 (优先级 0) 程序运行时，其他低优先级中断被屏蔽；当高优先级程序运行时，其他高优先级 (优先级 1) 中断服务程序被屏蔽；仅当 RETI 执行或 EA 设置为 1，中断硬件才对其他中断识别。一旦设置了中断请求标志，则标志保留，不管标志是由硬件或是由软件设置。可以通过设置 TCON 中的 TF0、TF1、IE0 或 IE1，SCON 中的 RI 或 TI 来采用软件引发中断程序。

同样，在向量中断硬件识别之前，可以用软件清除中断请求标志 (通常它们被屏蔽或更高或同优先级的中断正在执行中断服务)。注意，外部电平触发的中断不锁存。

在实时系统中，建立优先级是重要的。如何建立优先级和“任务可等多长时间”有关。例如，包含下列任务的项目：

- (1) 从串行口读字符，这要求在下一个字符到来前完成 (典型 1ms 内)；
- (2) 识别人按键 (在感觉延时前可等 100ms)；
- (3) 一个实时时钟 (以 10ms 速率滴答)，在下一个滴答之前需要重新启动定时器并计数一次。

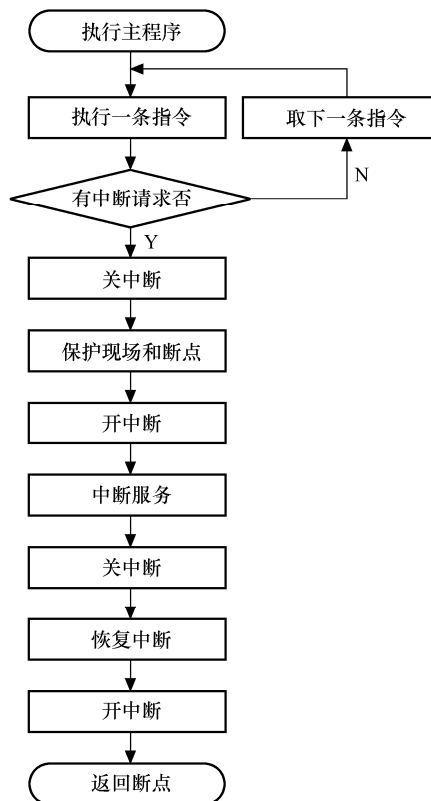


图 4-5 中断服务流程

时钟中断应立即识别, 因为任何延迟会减慢时钟。在中断处理时间中必须平衡中断的紧急程度。当自己编写中断服务程序时, 最好尽可能地短, 这样系统对其他中断保留敏感性。对耗用 CPU 时间, 且能在背景任务中完成的不很紧急的处理可为其设置标志。

对于上述情况, 若定时器必须软件重置初值(如方式 1), 逻辑上应让实时时钟是高优先级中断, 但应编写很短的中断服务程序。串行口输入可以是低优先级, 按键扫描可以是非中断的背景程序。

再者, 上述情况串行接收也可以设成高优先级, 因为它仅用几 μs 。实时时钟可以是低优先级并包括按键扫描。这种决定的影响是有时丢字符或有时漏掉几 μs 。仅特定的应用可决定这种问题的答案。

C51 直接支持仅两级中断优先级和背景非中断级。在同等或更高优先级任务运行时, 任务中断要屏蔽, 因而不同优先级的使用不可避免。运行很快而且很长时间不再使用的任务可放在高优先级。一种情况是步进电机驱动任务, 每几 ms 仅使用几 μs , 若挂起时间太长, 电机看起来似动非动。

为得到更快、更有效的代码, 使用中断服务程序仅能做非常简单的操作, 在中断程序中应避免使用太长的变量类型和算术操作。应该在中断程序中使用标志, 长的处理在背景程序中完成。例如: (1) 让原始的串行字符放入缓冲区, 待以后用于行编辑和命令分析; (2) 收集数据以后再取平均。这种设计方法在实际的实时系统是经常使用的。

3. 中断响应时间

所谓中断响应时间是指 CPU 检测到中断请求信号到转入中断服务程序入口所需要的机器周期数。了解中断响应时间对设计实时测控应用系统有重要指导意义。

C51 单片机响应中断的最短时间为 3 个机器周期。若 CPU 检测到中断请求信号时间正好是一条指令的最后一个机器周期, 则不需等待就可以立即响应。所谓响应中断就是由内部硬件执行一条长调用指令, 需要 2 个机器周期, 加上检测需要 1 个机器周期, 一共需要 3 个机器周期才开始执行中断服务程序。

中断响应的最长时间由下列情况所决定: 若中断检测时正在执行 RETI 或访问 IE 或 IP 指令的第一个机器周期, 这样包括检测在内需要 2 个机器周期(以上三条指令均需两个机器周期); 若紧接着要执行的指令恰好是执行时间最长的乘除法指令, 其执行时间均为 4 个机器周期; 然后再用 2 个机器周期执行一条长调用指令才转入中断服务程序。这样, 总共需要 8 个机器周期。其他情况下的中断响应时间一般为 3~8 个机器周期。

4. 中断请求的撤销

CPU 响应中断后, 应撤除该中断请求标志, 否则会再次引起中断。

对定时计数器 T0、T1 的溢出中断, CPU 响应中断后, 硬件自动清除中断请求标志 TF0 和 TF1。

对边沿触发的外部中断 $\overline{\text{INT1}}$ 和 $\overline{\text{INT0}}$, CPU 响应中断后硬件自动清除中断请求标志 IE0 和 IE1。

对于串行口中断, CPU 响应中断后, 没有用硬件清除中断请求标志 TI、RI, 即这些中断标志不会自动清除, 必须用软件清除, 这是在编串行通信中断服务中应该注意的。

对电平触发的外部中断，CPU 在响应中断时也不会自动清除中断标志，因此，在 CPU 响应中断后应立即撤除 $\overline{\text{INT1}}$ 或 $\overline{\text{INT0}}$ 的低电平信号。

5. 寄存器组切换

当一个特定任务正在执行时，可能有更紧急的事需引起 CPU 注意。在一个具有优先级的系统中，CPU 不是等待第一个任务完成，而是假定前一个任务已完成，立即处理新任务。若程序流程立刻转向新任务，新任务使用的各寄存器就破坏了第一个任务使用的中间信息。当第一个任务重新执行时，寄存器的值可引起错误发生。解决问题的方法是每次发生任务变化时执行一些指令，这就叫做上下文切换。

C51 有 4 个寄存器组，每组 8 个字节，它们位于内部 RAM 的起始位置。可分配 R0~R7 对应的这 8 个字节，其具体位置取决于 PSW（程序状态字）的两位（RS0，RS1）设置。这两位决定给定时间内 R0~R7 对应 RAM 的十六进制地址 0~7，8~F，10~17 或 18~1F。寄存器组使得中断系统有非常快的上下文切换。当中断发生时，典型变化包括由一动作转移到另一动作而不是压入和弹出堆栈（通常其他处理器是），RS0 和 RS1 两位的改变可保存所有 8 个寄存器，这是为了上下文切换。当运行一个中断任务时，中断系统采用不同的寄存器组。一个任务的 8 个字节保留，另一个不同的 8 个字节用在新任务中。通常建议同时把累加器和寄存器 B 的值压入栈。上下文切换一般包括两个进栈指令和一个简单的位变化指令，最坏的情况是要保存 DPH 和 DPL 寄存器。

寄存器组切换，在汇编语言中由编程者选择。但对混合语言编程的连接器，汇编程序使用的组可被特定，因而连接器不能像普通存储器那样分配寄存器组。在 C51 中，寄存器组选择取决于特定的编译器指令。

高优先级中断可以中断正在处理的低优先级程序，因而必须注意寄存器组。除非可以确定未使用 R0~R7（用汇编程序），最好给每种优先级程序分配不同的寄存器组。

中断不允许用于外部函数，它对函数目标代码影响有以下几点。

- (1) 当调用函数时，SFR 中的 ACC、B、DPH、DPL、PSW（当需要时）入栈；
- (2) 如果不使用寄存器组切换，甚至中断函数所需的所有工作寄存器都入栈；
- (3) 函数退出前，所有的寄存器内容出栈；
- (4) 函数由 8051 的指令 RETI 终止。

中断服务程序使用的任何程序也使用同一寄存器组。递归程序可以使用，它们自己调用自己是因为它们依赖堆栈。

4.2.6 中断的程序设计

用户对中断的控制和管理，实际是对 4 个与中断有关的寄存器 IE、TCON、IP、SCON 进行控制或管理。这几个寄存器在单片机复位时是清零的，因此必须根据需要对这几个寄存器的有关位进行预置。在中断程序的编写中应注意以下几点：

- (1) 采用开中断总控开关 EA 置位中断源的中断允许位；
- (2) 对外部中断 $\overline{\text{INT0}}$ 、 $\overline{\text{INT1}}$ 应选择中断触发方式；
- (3) 对于多个中断源中断，应设定中断优先级和预置 IP；

(4) 编写中断服务程序应注意保护现场和恢复现场，以免中断返回时丢失原寄存器、累加器中的信息；

(5) 若要在执行当前中断程序时禁止更高优先级中断，可以采用软件关 CPU 中断或禁止某中断源中断，在中断返回前再开放中断。

C51 编译器支持在 C 源程序中直接开发中断程序，因此减轻了用汇编语言开发中断程序的烦琐过程。使用该扩展属性的函数定义语法如下所示：

返回值函数名 interruptn

n 对应中断源的编号。在 C51 中，外部中断 $\overline{\text{INT0}}$ 为 0，定时/计数器 T0 为 1，外部中断 $\overline{\text{INT1}}$ 为 2，定时/计数器 T1 为 3，串行口中断为 4。

Franklin C51 编译器可以特殊指定寄存器独立的函数。当前工作寄存器由 PSW 中的两位来设置也可使用 using 指定，using 后的变量为一个 0~3 的常整数。

using 不允许用于外部函数，它对函数的目标代码影响有如下几点：

- (1) 函数入口处将当前寄存器组保留；
- (2) 需使用指定的寄存器组；
- (3) 函数退出前寄存器组恢复。

例如：void function(void)using 3{

.....

}

中断服务函数的完整语法如下所示：

返回值 函数名 ([参数]) [模式][重入]interrupt n[using n]

interrupt 后接一个 0~31 的常整数，它不允许使用表达式。

例如：void service_int1()interrupt 2 using 3{

.....

}

大量的 C 语言库程序是可重入的，但使用时需检查一下当中断到来时寄存器有何变化。而且，外部堆栈程序可以使用寄存器加载或卸载值，因而不能假定所有 C 语言编译器设计者考虑了所有中断和寄存器组的影响。某些编译器可能未在中断环境下测试。

【例 4-1】在图 4-6 中 P1.4~P1.7 接有 4 个发光二极管，P1.0~P1.3 接有 4 个开关，消抖电路用于产生中断请求信号，当消抖电路的开关来回拨动一次将产生一个下降沿信号，通过 $\overline{\text{INT0}}$ 向 CPU 申请中断，要求：开始时发光二极管全黑，每中断一次，P1.0~P1.3 所接的开关状态反映到发光二极管上，且要求开关断开的对应发光二极管亮。

Franklin C51 编译器提供定义特定 51 系统成员的寄存器头文件 reg51.h。程序如下：

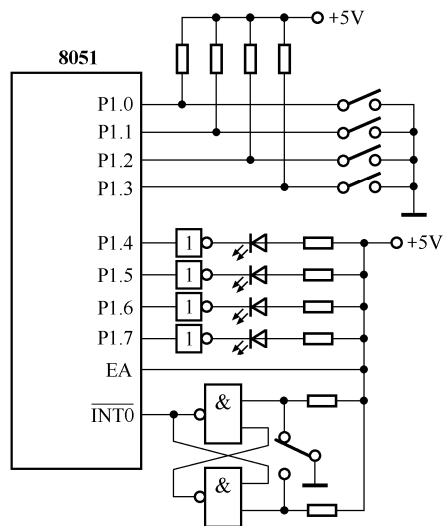


图 4-6 中断编程实例图

```

#include <reg51.h>
unsigned char breaks;
bit flag;
sbit P1_0=P1^0;
sbit P1_1=P1^1;
sbit P1_2=P1^2;
sbit P1_3=P1^3;
sbit P1_4=P1^4;
sbit P1_5=P1^5;
sbit P1_6=P1^6;
sbit P1_7=P1^7;

void service_int1() interrupt 0 using 2
{ /* INT0 中断服务程序，使用第二组寄存器 */
    flag =1;          /* 设置标志 */
    breaks=p1;        /* 存状态 */
}

void main(void)
{
    IP=0x01;          /* 置 INT0 高优先级中断 */
    IE=0x81;          /* INT0 开中断，CPU 开中断 */
    for(;;)
    {
        If(flag)
        { /* 有中断 */
            P1_4=P1_0;
            P1_5=P1_1;
            P1_6=P1_2;
            P1_7=P1_3;
            flag =0;    /* 处理完成清标志 */
        }
    }
}

```

4.3 定时/计数器

51 系列单片机内部有两个十六位定时/计数器：定时器 0（T0）和定时器 1（T1）。

定时/计数器实际上是 16 位加 1 计数器。T0 由 2 个 8 位特殊功能寄存器 TH0 和 TL0 构成，T1 由 2 个 8 位特殊功能寄存 TH1 和 TL1 构成。每个定时器都可由软件设置为定时工作方式或计数工作方式。T0 和 T1 受特殊功能寄存器 TMOD 和 TCON 控制。

对于定时器/计数器来说，不管是独立的定时器芯片还是单片机内的定时器，大都具有以下特点：

(1) 定时/计数器有多种方式，可以是计数方式也可以是定时方式。

(2) 定时/计数器的计数值是可变的，当然计数的最大值是有限的，这取决于计数器的位数。计数的最大值也就限定了定时的最大值。

(3) 在到达设定的定时或计数值时发出中断申请，以便实现定时控制。

两个定时器都有定时或事件计数的功能，可用于定时控制、延时、对外部事件计数和检测等场合。下面对定时/计数器的工作方式进行讨论。

1. 定时工作方式

设置为定时工作方式时，定时器计数的脉冲是由 C51 单片机片内振荡器经 12 分频后产生的。每经过一个机器周期定时器（T0 或 T1）的数值加 1 直至计数满产生溢出。

例如：当 8051 采用 12MHz 晶体时，每个机器周期为 $1\mu\text{s}$ ，计 5 个机器周期即为 $5\mu\text{s}$ ，即定时 $5\mu\text{s}$ 。

2. 计数工作方式

设置为计数工作方式时，通过引脚 T0（P3.4）和 T1（P3.5）对外部脉冲信号计数。

当输入脉冲信号产生由 1 至 0 的下降沿时，定时器的值加 1，在每个机器周期 CPU 采样 T0 和 T1 的输入电平。若前一个机器周期采样值为高，下一个机器周期采样值为低，则计数器加 1。由于检测一个 1 至 0 的跳变需要二个机器周期，故最高计数频率为振荡频率的四分之一。

虽然对输入信号的占空比无特殊要求，但为了确保某个电平在变化之前至少被采样一次，要求电平保持时间至少是一个完整的机器周期。

4.3.1 定时/计数器的控制寄存器

定时/计数器具有两个控制寄存器：定时/计数器控制 TCON（88H）和定时/计数器工作模式寄存器 TMOD（89H）。

1. 计数寄存器 TH 和 TL

定时/计数器的核心是 16 位加法计数器，它采用特殊功能寄存器 TH0、TL0 及 TH1、TL1 表示。TH0、TL0 是定时/计数器 0 加法计数器的高 8 位和低 8 位，TH1、TL1 是定时/计数器 1 加法计数器的高 8 位和低 8 位。

加法计数器的初值可以由程序设定，设置的初值不同，计数值或定时时间就不同。在定时/计数器的工作过程中，加法计数器的内容可用程序读回 CPU。

2. 定时/计数器控制寄存器 (TCON)

TCON 控制寄存器各位定义如下所示:

位地址	8FH	8EH	8DH	8CH	8BH	8AH	89H	88H
位符号	TF1	TR1	TF0	TR0	IE1	IT1	IE1	IE0

TF0 (TF1): T0 (T1) 定时/计数器溢出中断标志位。当 T0 (T1) 计数溢出时, 由硬件置位, 并在允许中断的情况下, 向 CPU 发出中断请求信号, CPU 响应中断转向中断服务程序时, 由硬件自动将该位清零。

TR0 (TR1): T0 (T1) 运行控制位。当 TR0 (TR1) =1 时启动 T0 (T1); TR0 (TR1) =0 时关闭 T0 (T1)。该位由软件进行设置。

3. 定时/计数器方式控制寄存器 (TMOD)

定时/计数器 T0、T1 都有四种工作方式, 可通过程序对 TMOD 设置来选择。TMOD 的低 4 位用于定时/计数器 0, 高 4 位用于定时/计数器 1。其位定义如下所示:

位序	B7	B6	B5	B4	B3	B2	B1	B0
位符号	GATE	C/ $\bar{T}$	M1	M0	GATE	C/ $\bar{T}$	M1	M0
← 定时器/计数器 1 →				← 定时器/计数器 0 →				

(1) C/ $\bar{T}$: 定时或计数功能选择位。当 C/T=1 时为计数方式; 当 C/T=0 时为定时方式。

(2) M1, M0: 定时/计数器工作方式选择位, 其值与工作方式的对应关系如下所示。

M1M0	工作方式	对应计数器
00	方式 0	13 位计数器
01	方式 1	16 位计数器
10	方式 2	初值自动重装 8 位计数器
11	方式 3	2 个 8 位计数器, 仅适用于 T0

(3) GATE: 门控位, 用于控制定时/计数器的启动是否受外部中断请求信号的影响。如果 GATE=1, 定时/计数器 0 的启动受芯片引脚 $\overline{INT0}$ (P3.2) 控制, 定时器/计数器 1 的启动受芯片引脚 $\overline{INT1}$ (P3.3) 控制; 如果 GATE=0, 定时/计数器的启动与引脚 $\overline{INT0}$ 、 $\overline{INT1}$ 无关。一般情况下 GATE=0。

4.3.2 定时/计数器的工作方式

1. 工作方式 0

(1) 电路逻辑结构

方式 0 是 13 位计数结构的工作方式，其计数器由 TH0 全部 8 位和 TL0 的低 5 位构成，TL0 的高 3 位弃之不用。具体逻辑结构如图 4-7 所示。

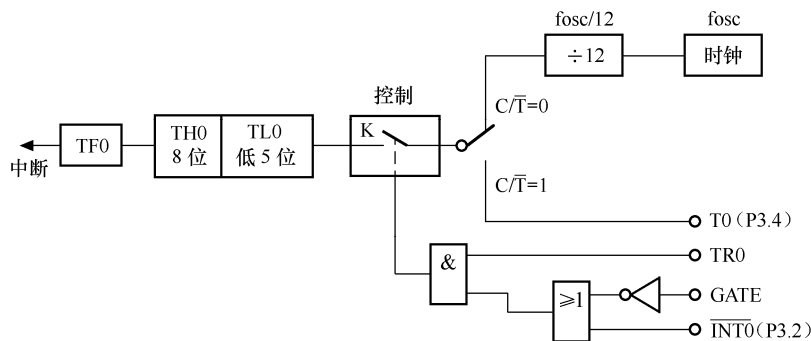


图 4-7 定时/计数器方式 0 的逻辑结构

(2) 定时和计数应用

在方式 0 下，当为计数工作方式时，计数值的范围是 $1 \sim 8192$ (2^{13})；当为定时工作方式时，定时时间的计算公式为 $(2^{13} - \text{计数初值}) \times \text{晶振周期} \times 12$ 或 $(2^{13} - \text{计数初值}) \times \text{机器周期}$ ，其时间单位与晶振周期或机器周期相同 (μs)。

例如，晶振频率为 6 MHz，则最小定时时间为：

$$[2^{13} - (2^{13} - 1)] \times 1/6 \times 10^{-6} \times 12 = 2 \times 10^{-6} = 2 (\mu\text{s})$$

最大定时时间为：

$$(2^{13} - 0) \times 1/6 \times 10^{-6} \times 12 = 16384 \times 10^{-6} = 16384 (\mu\text{s})$$

【例 4-2】设单片机晶振频率为 6MHz，使用定时器 1 以方式 0 产生周期为 500 μs 的等宽正方波连续脉冲，并由 P1.0 输出，以查询方式完成。

① 计算计数初值

要产生 500 μs 的等宽正方波脉冲，只需在 P1.0 端以 250 μs 为周期交替输出高低电平即可实现，为此定时时间应为 250 μs 。使用 6MHz 晶振，则一个机器周期为 2 μs 。方式 0 为 13 位计数结构。设待求的计数初值为 X，则：

$$(2^{13} - X) \times 2 \times 10^{-6} = 250 \times 10^{-6}$$

求解得：X = 8067，二进制数表示为 11111100 00011B；十六进制表示时，高 8 位为 0FCH，低 5 位为 03H。其中，高 8 位放入 TH1，即 TH1 = 0FCH；低 5 位放入 TL1，即 TL1 = 03H。

② TMOD 寄存器初始化

若定时/计数器 1 设定为方式 0，则 M1M0 = 00；为实现定时功能，应使 $C/\overline{T} = 0$ ；为实现定时/计数器 1 的运行控制，则 GATE = 0；定时/计数器 0 不用，有关位设定为 0。因此，TMOD 寄存器应初始化为 00H。

③ 由定时器控制寄存器 (TCON) 中的 TR1 位控制定时的启动和停止

TR1 = 1：启动；

TR1 = 0：停止。

④ 程序设计

main()

```

{
    TMOD=00;
    TH1=0xFC;
    TH0=0x03;
    IE=0x00;
    for()
    {
        TR1=1;
        for(TF1)
        {
            TH1=0xFC;
            TH0=0x03;
            TF1=0x00;
            P1^0=~ P1^0;
        }
    }
}

```

2. 工作方式 1

(1) 电路逻辑结构

方式 1 是 16 位计数结构的工作方式，其计数器由 TH0 全部 8 位和 TL1 全部 8 位构成。具体逻辑结构如图 4-8 所示。

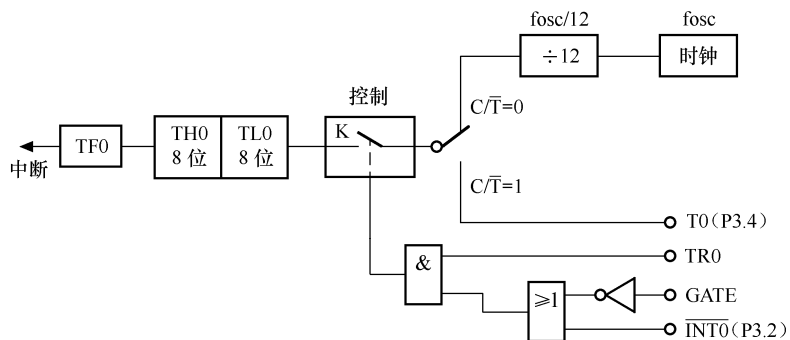


图 4-8 定时/计数器方式 1 的逻辑结构

(2) 定时和计数应用

当为计数工作方式时，计数值的范围是 1~65536 (2¹⁶)；当为定时工作方式时，定时时间的计算公式为 (2¹⁶-计数初值) × 晶振周期 × 12 或 (2¹⁶-计数初值) × 机器周期，其时间单位与晶振周期或机器周期相同 (μs)。

【例 4-3】 题目同【例 4-2】，但以中断方式完成。单片机晶振频率为 6MHz，使用定时器 1 以工作方式 1 产生周期为 500μs 的等宽正方形波连续脉冲，并由 P1.0 输出。

① 计算计数初值: $TH1 = 0FFH$, $TL1 = 83H$ 。

② TMOD 寄存器初始化: $TMOD = 10H$ 。

③ 程序设计

```
main()
{
    TMOD=0x10H;      /*设置 T1 为工作方式*1/
    TH1=0xFFH;       /*设置计数初值*/
    TL1=0x83H;
    EA=1;            /*开中断*/
    ET1=1;           /*定时器 1 允许中断*/
    TR1=1;           /*定时开始*/
    for(;;)          /*等待中断*/
}

void timer() interrupt 2 using 0
{
    TH1=0xFFH;       /*设置计数初值*/
    TL1=0x83H;
    P1^0=~P1^0;      /*输出取反*/
}
```

3. 工作方式 2

(1) 电路逻辑结构

方式 2 是自动重新加载工作方式, 其逻辑结构如图 4-9 所示。

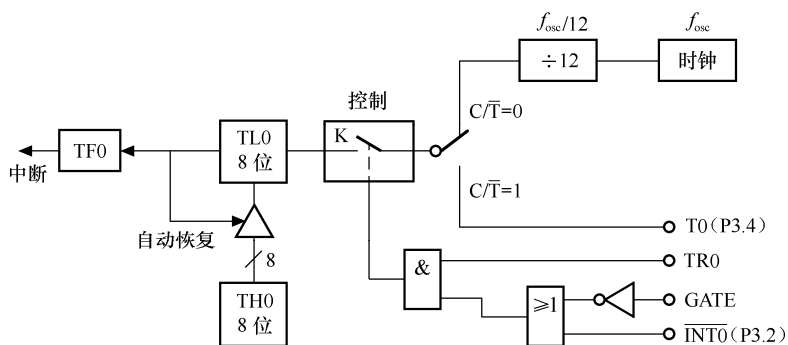


图 4-9 定时/计数器方式 2 的逻辑结构

(2) 定时和计数应用

在这种工作方式下, 把 16 位计数器分为两部分, 即以 TL 作计数器, 以 TH 作预置寄存器。初始化时把计数初值分别装入 TL 和 TH 中; 当计数溢出后, 由预置寄存器 TH 以硬件方法自动给计数器 TL 重新加载, 变软件加载为硬件加载。

【例 4-4】 使用定时器 0 以工作方式 2 产生 $100\mu s$ 定时, 在 P1.0 输出周期为 $200\mu s$ 的连

续正弦波脉冲。已知晶振频率 $f_{\text{OSC}} = 6\text{MHz}$ 。

① 计算计数初值

6MHz 晶振下，一个机器周期为 $2\mu\text{s}$ ，以 TH0 作重装载的预置寄存器，TL0 作 8 位计数器，假设计数初值为 X，则：

$$(2^8 - X) \times 2 \times 10^{-6} = 100 \times 10^{-6}$$

求解得：

$$X = 206\text{D} = 11001110\text{B} = 0\text{CEH}$$

把 0CEH 分别装入 TH0 和 TL0 中：TH0 = 0CEH，TL0 = 0CEH。

② TMOD 寄存器初始化

定时/计数器 0 为工作方式 2，M1M0 = 10；定时功能 $\overline{\text{C/T}} = 0$ ；为实现定时器/计数器 0 的运行 GATE = 0；定时器/计数器 1 不用，有关位设定为 0。因此，TMOD 寄存器的状态应为 02H。

③ 程序设计（查询方式）

```
main()
{
    IE=0x00H;           /*禁止中断*/
    TMOD=0x02H;         /*设置定时器 0 为工作方式 2*/
    TH0=0xCEH;           /*保存计数初值*/
    TL0=0xCEH;           /*设置计数初值*/
    TR0=1;               /*启动定时*/
    for(;;)
    {
        if(TF0)          /*查询计数溢出*/
        {
            P1^0=~P1^0;   /*输出方波*/
            TF0=0;
        }
    }                     /*重复循环*/
}
```

由于方式 2 具有自动重装载功能，因此计数初值只需设置一次，以后不再需要软件重置。

④ 程序设计（中断方式）

```
main()
{
    TMOD=0x02H;         /*设置定时器 0 为工作方式 2*/
    TH0=0xCEH;           /*保存计数初值*/
    TL0=0xCEH;           /*设置计数初值*/
    EA=1;                /*开中断*/
    ET0=1;               /*定时器 0 允许中断*/
}
```



```

SETB=1;                /*定时开始*/
for(;;);                /*等待中断*/
}

void timer() interrupt 2 using 0
{
    P1^0=~ P1^0;        /*输出方波*/
    TF0=0;              /*计数溢出标志位清“0”*/
}

```

4. 工作方式 3

在工作方式 3 下，定时/计数器 0 被拆成两个独立的 8 位计数器 TL0 和 TH0。其中 TL0 既可以计数使用，又可以定时使用，定时/计数器 0 的各控制位和引脚信号全归它使用。

在方式 3 下，T0、T1 的设置和使用是不同的。

(1) T0 方式 3

TL0：使用 T0 原有控制资源，功能与方式 0、1 相同。

TH0：借用 T1 的 TR1、TF1，只能对片内机器周期脉冲计数，作 8 位定时器。

T0 方式 3 时的 T0、T1 电路逻辑结构，如图 4-10 所示。T0 方式 3 时的 T0 初值计算完全同方式 2。

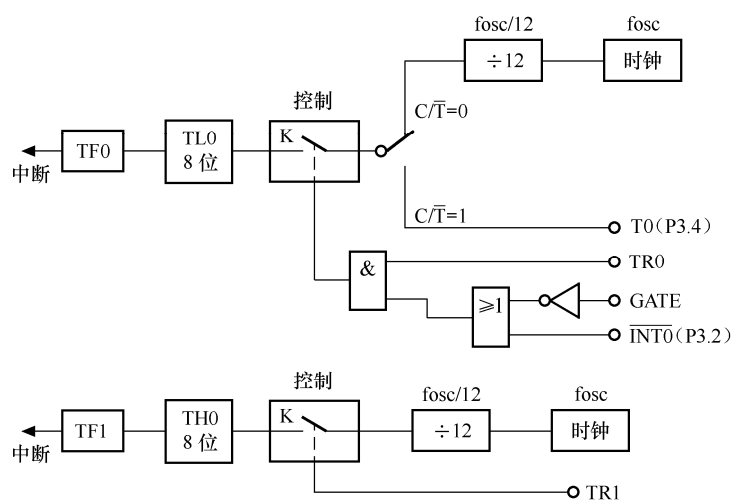


图 4-10 定时/计数器方式 3 的逻辑结构

(2) T0 方式 3 下的 T1

T0 方式 3 时，T1 仍然可工作于方式 0~方式 2。C/T 控制位仍可使 T1 工作在定时器或计数器方式，只是由于其 TR1、TF1 被 T0 的 TH0 占用，因而没有计数溢出标志可供使用，计数溢出时只能将输出结果送至串行口，即用作串行口波特率发生器。T0 方式 3 下的 T1 方式 2，因定时初值能自动恢复，用作波特率发生器更为合适。

4.3.3 定时/计数器的应用实例

在实时系统中，定时通常使用定时器，这与软件循环的定时完全不同，尽管两者最终都依赖系统的时钟。在定时器计数时，其他事件可继续进行，软件定时不允许任何事情发生。

【例 4-5】 设单片机的 $f_{osc}=12\text{ MHz}$ ，要求在 P1.0 脚上输出周期为 2ms 的方波。

分析：周期为 2ms 的方波要求定时间隔 1ms，每次时间到 P1.0 取反。

定时器计数率= $f_{osc}/12$ ，机器周期= $12/f_{osc}=1\mu\text{s}$ ；

每个机器周期定时器计数加 1， $1\text{ms}=1000\mu\text{s}$ ；

需计数次数= $1000/(12/f_{osc})=1000/1=1000$ 。

由于计数器向上计数，为得到 1000 个计数之后的定时器溢出，必须给定时器置初值-1000。

(1) 用定时器 0 的方式 1 编程，采用查询方式，程序名为 time32.c。

```
#include<reg51.h>
sbit P1_0=P1^0;
void main(void)
{
    TMOD=0x01;          /*定时器 0 方式 1*/
    TR0=1;               /*启动 T/C0*/
    for(;;)
    {
        TH0=-(1000/256); /*装载计数初值*/
        TL0=-(1000%256);
        do{}while(!TF0); /*查询等待 TF0 置位*/
        P1_0=!P1_0;      /*定时时间到 P1.0 反相*/
        TF0=0;           /*软件清 TF0*/
    }
}
```

(2) 用定时器 0 的方式 1 编程，采用中断方式，程序名为 time31.c。

```
#include<reg51.h>
sbit p1_0=P1^0;

void timer0(void)interrupt 1 using 1
{
    /*T/C0 中断服务程序入口*/
    P1_0=!P1_0; /*P1.0 取反*/
    TH0=-(1000/256); /*计数初值重载*/
    TL0=-(1000%256);
}
```

```
void main(void)
```

```

{
    TMOD=0x01;          /*T/C0 工作在定时器方式 1*/
    P1_0=0;
    TH0=-(1000/256);     /*预置计数初值*/
    TL0=-(1000%256);
    EA=1;                /*CPU 开中断*/
    ET0=1;               /*T/C0 开中断*/
    TR0=1;               /*启动 T/C0 开始定时*/
    do{}
    while(1);
}

```

【例 4-6】 采用 10MHz 晶振，在 P1.0 脚上输出周期为 2.5s，占空比 20%为脉冲信号。

分析：由于采用 10MHz 晶振，使用定时器最大定时几十 ms。取 10ms 定时，周期 2.5s 需 250 次中断，占空比 20%，高电平应为 50 次中断。

采用 10ms 定时，晶振 $f_{osc}=10\text{MHz}$ ，因此需定时器计数次数 $=10 \times 10^3 \times 10/12=8333$ 。

程序名为 time34.c，具体程序如下所示：

```

#include<reg51.h>
#define uchar unsigned char
uchar time;
uchar period=250;
uchar high=50;

timer0()interrupt 1 using 1
{
    /*T/C0 中断服务程序*/
    TH0=-8333/256;      /*重载计数初值*/
    TL0=-8333%256;
    if(++time==high)P1=0; /*高电平时间到变低*/
    else if(time==period){ /*周期时间到变高*/
        time=0;
        P1=1;
    }
}

main()
{
    TMOD=0x01;          /*定时器 0 方式 1*/
    TH0=-8333/256;      /*预置计数初值*/
    TL0=-8333%256;

```

```

EA=1;                /*开 CPU 中断*/
ET0=1;              /*开 T/C0 中断*/
TR0=1;              /*启动 T/C0*/
do{}
while(1);
}

```

【例 4-7】 设 P1 口的 P1.0 和 P1.1 上有两个开关 S1 和 S2，周期开始它们全关。2s 以后 S1 开，0.1s 后 S2 开，S1 保持开 2.0s，S2 保持开 2.4s，周而复始。采用 10MHz 晶振。

分析：根据要求，P1.1 和 P1.0 上开关顺序为：

（关关）2s 后（关开）0.1s 后（开开）1.9s 后（开关）0.5s 秒后（关关）。

采用 10MHz 晶振，每 10ms 中断一次，0.1s 对应 10 次，开关变化对应的中断次数位置为 0、200、210、400、450；相应的 P1.0 输出 0、1、3、2。

程序名为 time35.c，具体程序如下所示：

```

#include<reg51.h>
#define uchar unsigned char
#define uint unsigned int
uchar i;
uint time;

code struct
{
    int position;
    char pattern;
}next[]=                /* item1 */
{{0, 0x00}, {200, 0x01}, {210, 0x03}, {400, 0x02}, {450, 0xff}};

timer0()interrupt 1 using 1
{
    TH0=-8333/256;
    TL0=-8333%256;
    time++;                /* item3 */
    if(time==next[i].position)
    {
        if(next[i].pattern==0xff)i=time=0;    /* item4 */
        P1=next[i++].pattern;
    }
}

main()
{

```

```

P1=0;                                     /* item5 */
time=0;
i=1;
TMOD=0x01;
TH0=-8333/Z56;
TL0=-8333%256;
TR0=1;
ET0=1;
EA=1;
for(;;){
}

```

在程序中,

item1: 采用结构数组定义时间和输出的对应关系。

item2: 中断服务程序, 每 10ms 中断一次, 重载定时器初值。

item3: 记录中断发生的次数。

hem4: 以 0xff 判断周期的结束。

hem5: 初始化变量和定时器。

4.4 串行通信接口

4.4.1 串行通信基础知识

1. 数据通信的基本概念

通信是指 CPU 与外围设备之间的数据传送、转换和处理。数据通信方式有两种: 并行数据通信和串行数据通信, 如图 4-11 所示。

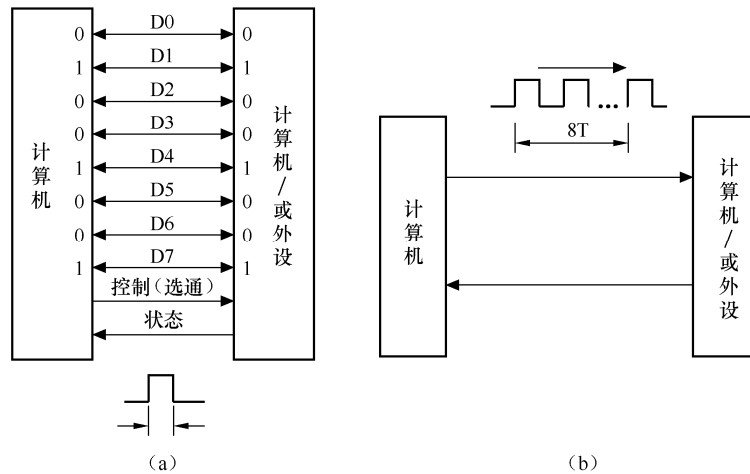


图 4-11 通信的两种基本方式

(a) 并行通信; (b) 串行通信

① 串行数据通信是指数据按位顺序进行传送的通信方式。其特点是数据传送是一位一位进行传送的，最少只需一根传输线即可完成，成本低但速度慢。

② 并行数据通信是指数据的各位同时进行传送的通信方式。其特点是各数据位同时传送，传送速度快、效率高。

2. 异步通信和同步通信

(1) 异步通信

异步串行通信以字符为单位，即一个字符一个字符地传送。图 4-12 所示，就是一个字符的异步串行通信格式。

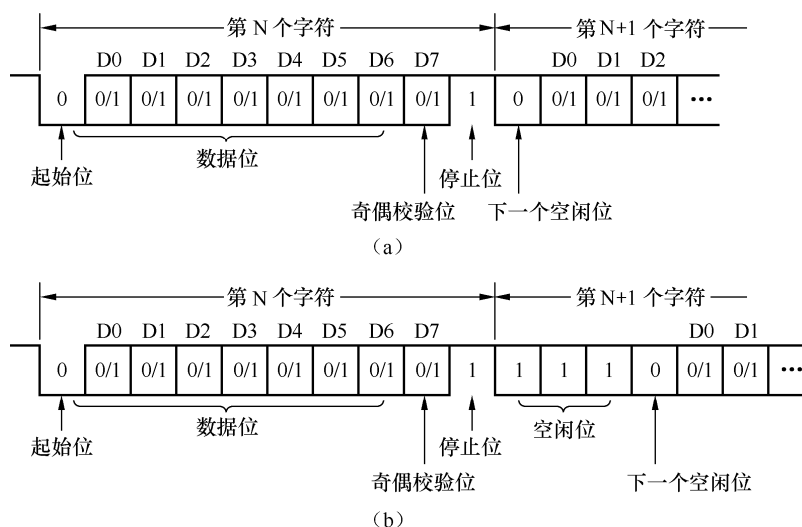


图 4-12 串行异步传送的字符格式

(a) 字符格式; (b) 有空闲位的字符格式

对异步串行通信的字符格式作如下说明。

① 起始位：发送器是通过发送起始位来开始一个字符的传送。起始位使数据线处于“Space”状态。

② 数据位：起始位之后就传送数据位。在数据位中，低位在前（左），高位在后（右）。由于字符编码方式的不同，数据位可以是 5、6、7 或 8 位等。

③ 奇偶校验位：用于对字符传送作正确性检查，因此奇偶校验位是可选择的，共有 3 种可能，即奇校验、偶校验和无校验，其由用户根据需要选定。

④ 停止位：停止位在最后，用以标志一个字符传送的结束，它对应于 Mark 状态。停止位可能是 1、1.5 或 2 位，在实际应用中根据需要确定。

(2) 同步通信

同步通信中，在数据开始传送前用同步字符来指示（常约定 1~2 个），并由时钟来实现发送端和接收端同步，即检测到规定的同步字符后，下面就连续按顺序传送数据，直到通信告一段落。同步传送时，字符与字符之间没有间隙，也不用起始位和停止位，仅在数据块开始时用同步字符 SYNC 来指示。串行同步传送的字符格式如图 4-13 所示。

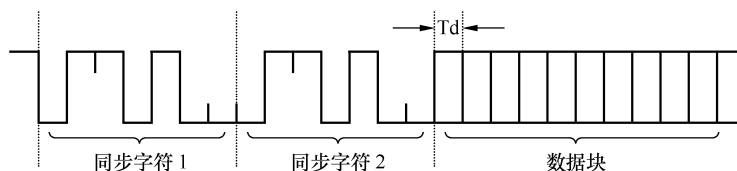


图 4-13 串行同步传送的字符格式

3. 串行通信的数据通路形式

串行数据通信共有以下几种数据通路形式：

(1) 单工 (Simplex) 形式。单工形式的数据传送是单向的，通信双方中一方固定为发送端，另一方则固定为接收端，如图 4-14 (a) 所示。

(2) 全双工 (Full-duplex) 形式。全双工形式的数据传送是双向的，且可以同时发送和接收数据，因此全双工形式的串行通信需要两条数据线，如图 4-14 (c) 所示。

(3) 半双工 (Half-duplex) 形式。半双工形式的数据传送也是双向的，但任何时刻只能由其中的一方发送数据，另一方接收数据。因此，半双工形式既可以使用一条数据线，也可以使用两条数据线，如图 4-14 (b) 所示。

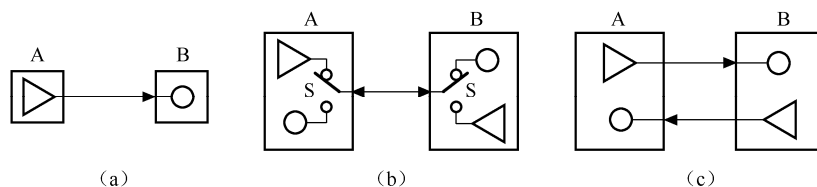


图 4-14 串行通信数据传送的三种方式
(a) 单工方式；(b) 半双工方式；(c) 全双工方式

4. 串行接口的基本任务

在串行传输中，通信的双方都按通信协议进行。所谓通信协议就是通信双方必须共同遵守的一种约定，约定包括数据的格式、同步的方式、传送的步骤、检纠错方式及控制字符的定义等。

串行接口的基本任务主要包括以下几个方面。

(1) 实现数据格式化

因为 CPU 发出的数据是并行数据，接口电路应实现不同串行通信方式下的数据格式化任务，如自动生成起止方式的帧数据格式（异步方式）或在待传送的数据块前加上同步字符等。

(2) 进行串、并转换

在发送端，接口将 CPU 送来的并行信号转换成串行数据进行传送；而在接收端，接口要将接收到串行数据变成并行数据送往 CPU，由 CPU 进行处理。

(3) 控制数据的传输速率

接口应具备对数据传输率—波特率的控制选择能力，即具有波特率发生器。

(4) 进行传送错误检测

在发送时，对传送的数据自动生成校验位或校验码，在接收端能检查校验位或校验码，以确定传送中是否有误码。

5. 波特率和接收/发送时钟

(1) 波特率的概念

波特率，即数据传送速率，表示每秒钟传送二进制代码的位数，它的单位是位/秒 (bps)。波特率对于 CPU 与外界的通信是很重要的。

(2) 接收/发送时钟

① 接收/发送时钟就是用来控制通信设备接收/发送字符数据速度的，该时钟信号通常由微机内部时钟电路产生。

② 在接收数据时，接收器在接收时钟的上升沿对接收数据采样，进行数据位检测；在发送数据时，发送器在发送时钟的下降沿将移位寄存器的数据串行移位输出。

③ 接收/发送时钟频率与波特率有如下关系：

$$\text{收/发时钟频率} = n \times \text{收/发波特率}$$

$$\text{收/发波特率} = \frac{\text{收/发时钟频率}}{n}$$

其中频率系数 $n=1、16、64$ 。串行通信要求发送端与接收端的波特率必须一致。

异步串行通信的传送速率一般为 50~9600 波特，常用于计算机到 CRT 终端和字符打印机之间的通信、直通电报以及无线电通信的数据发送等。

对于同步传送方式，必须取 $n=1$ ，即接收/发送时钟的频率等于收/发波特率。对于异步传送方式， $n=1、16、64$ ，即可以选择的接收/发送时钟频率是波特率的 1.16 或 64 倍。因此，可由要求的传送波特率及所选择的倍数 n 来确定接收/发送时钟的频率。

④ 接收/发送时钟的周期 T_c 与发送的数据位宽 T_d 之间的关系是：

$$T_c = \frac{T_d}{n} \quad (n = 1, 16, 64)$$

4.4.2 C51 串行接口

C51 单片机有一个可编程的全双工异步串行通信接口，它可作 UART 用，也可作同步移位寄存器，其帧格式可有 8 位、10 位或 11 位，并能设置各种波特率，给使用者带来很大的设计灵活性。

1. 串行口寄存器结构

串行接口主要由发送数据缓冲器、发送控制器、输出控制门、接收数据缓冲器、接收控制器、输入移位寄存器、波特率发生器 T1 等组成。

串行口中还有两个特殊功能寄存器 SCON、PCON。特殊功能寄存器 SCON 用来存放串行口的控制和状态信息。定时器/计数器 1 (T1) 与定时器/计数器 2 (T2) 都可构成串行口的波特率发生器，其波特率是否增倍可由特殊功能寄存器 PCON 的最高位控制。

2. 串行接口控制

(1) 数据缓冲器 (SBUF)

发送 SBUF 和接收 SBUF 共用一个地址 99H。

① 发送 SBUF 存放待发送的 8 位数据，写入 SBUF 将同时启动发送。发送指令为：

MOV SBUF, A

② 接收 SBUF 存放已接收成功的 8 位数据，供 CPU 读取。读取串行口接收数据指令为：

MOV A, SBUF

(2) 串行控制寄存器 (SCON)

SCON 用于串行数据通信的控制。其单元地址 98H，位地址 9FH~98H。寄存器的内容及位地址表如下所示：

位地址	9F	9E	9D	9C	9B	9A	99	98
位符号	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

各位功能说明如下。

① SM0、SM1：串行口工作方式选择位

SM0、SM1 和串行口工作方式的具体关系如表 4-5 所示。

表 4-5 串行口的工作方式

SM0	SM1	工 作 方 式	说 明	波 特 率
0	0	方式 0	同步移位寄存器	$f_{osc}/12$
0	1	方式 1	10 位异步收发	由定时器控制
1	0	方式 2	11 位异步收发	$f_{osc}/32$ 或 $f_{osc}/64$
1	1	方式 3	11 位异步收发	由定时器控制

② SM2：多机通信控制位

因多机通信是在方式 2 和方式 3 下进行，因此 SM2 位主要用于方式 2 和方式 3。当串行口以方式 2 或方式 3 接收时，如 SM2=1，则有当接收到的第 9 位数据 (RB8) 为 1，才将接收到的前 8 位数据送入 SBUF，并置位 RI 产生中断请求；否则，将接收到的前 8 位数据丢弃。而当 SM2=0 时，则不论第 9 位数据为 0 还是为 1，都将前 8 位数据装入 SBUF 中，并产生中断请求。在方式 0 时，SM2 必须为 0。

③ REN：允许接收位

REN 位用于对串行数据的接收进行控制：REN=0 禁止接收；REN=1 允许接收。

④ TB8：发送数据位 8

在方式 2 和方式 3 时，TB8 是要发送的第 9 位数据。

⑤ RB8：接收数据位 8

在方式 2 或方式 3 时，RB8 存放接收到的第 9 位数据，代表着接收数据的某种特征。

⑥ TI：发送中断标志

当方式 0 时，发送完第 8 位数据后，该位由硬件置位。

⑦ RI：接收中断标志

当方式 0 时，接收完第 8 位数据后，该位由硬件置位，RI 位由软件清零。

(3) 电源控制寄存器 (PCON)

PCON 主要是为 CHMOS 型单片机的电源控制而设置的专用寄存器。单元地址内容如下所示:

位 序	D7	D6	D5	D4	D3	D2	D1	D0
位 符 号	SMOD	/	/	/	GF1	GF0	PD	IDL

在 CHMOS 的单片机中, 该寄存器中除最高位之外, 其他位都是虚设的。最高位 (SMOD) 是串行口波特率的倍增位, 当 SMOD=1 时, 串行口波特率加倍。系统复位时, SMOD=0。

由于 PCON 寄存器不能进行位寻址, 因此, 表中写了“位序”而不是“位地址”。

3. 串行接口的工作方式

(1) 方式 0

方式 0 为移位寄存器输入/输出方式。串行数据通过 RXD 输入/输出, TXD 则用于输出移位时钟脉冲。波特率固定为 $f_{osc}/12$, 其中 f_{osc} 为单片机外接晶振频率。方式 0 时, 移位数据的发送和接收以 8 位为一帧, 不设起始位和停止位, 低位在前, 高位在后。

其帧格式如下所示:

...	D0	D1	D2	D3	D4	D5	D6	D7	...
-----	----	----	----	----	----	----	----	----	-----

方式 0 下的接收发送时序如图 4-15 所示。

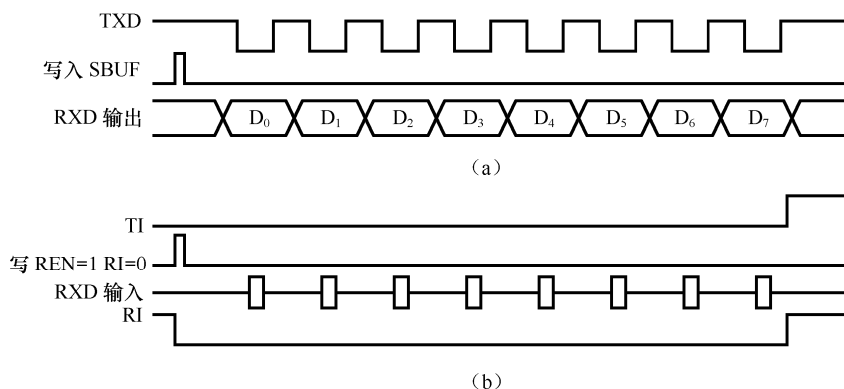


图 4-15 串口方式 0 通信时序

(a) 发送时序 (b) 接收时序

发送是以写 SBUF 寄存器的指令开始的, 8 位输出结束时 TI 被置位。方式 0 接收是在 REN=1 和 RI=0 同时满足时开始的。接收的数据装入 SBUF 中, 结束时 RI 被置位。

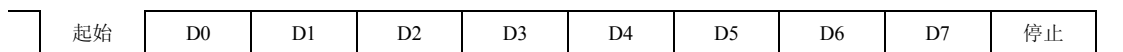
移位寄存器方式在用最小的硬件扩展接口时很有用。串行口外接一片移位寄存器 74LS164 可构成输出接口电路; 串行口外接一片移位寄存器 74LS165 可构成输入接口电路。在典型的 1MHz 时钟下, 8 位加载大约用 $10\mu s$ 。任何数目的移位寄存器可串接用于输出和输入, 这需要通过一系列的 SBUF 的写和读。若移位时的波动不重要或移位寄存器中包含并行

加载锁存, 这样可构成非常经济的 I/O 扩展小系统。

移位寄存器方式的第二种用法是用于两个单片机之间的通信。与通常 9600 波特相比, 1MHz 通信能力对短距离通信具有较强的吸引力。

(2) 方式 1

方式 1 是 10 位异步通信方式, 1 位起始位 (0)、8 位数据位和 1 位停止位 (1)。其中起始位和停止位在发送时是自动插入的。RXD 为接收端, TXD 为发送端。一帧信息由 10 位组成, 方式 1 的波特率可变, 由定时/计数器 1 的溢出速率以及 SMOD (PCON.7) 决定, 且发送波特率与接收波特率可以不同。其帧格式如下所示:



方式 1 下的接收发送时序如图 4-16 所示。

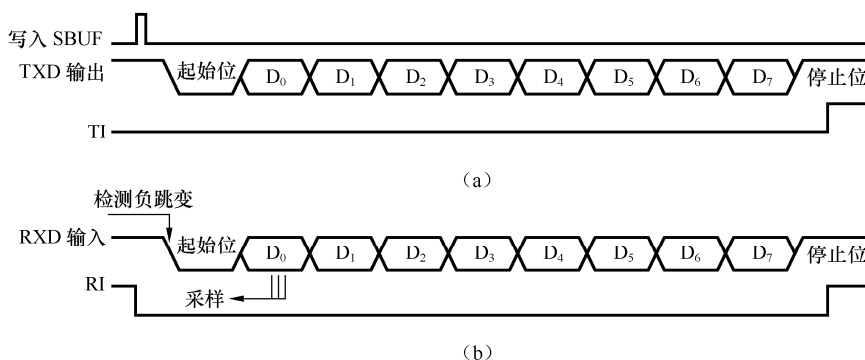


图 4-16 串口方式 1 通信时序

(a) 发送时序 (b) 接收时序

① 发送

CPU 执行一条写 SBUF 指令后便启动了串行口发送, 数据从 TXD 输出, 时序如图 4-16 (a) 所示。

在指令执行期间, CPU 送来“写 SBUF”信号, 将并行数据送入 SBUF, 并启动发送控制器, 经一个机器周期, 发送控制器的 SEND、DATA 相继有效, 通过输出控制门从 TXD 上逐位输出一帧信号。一帧信号发送完毕后, SEND、DATA 失效, 发送控制器硬件置发送中断标志 TI=1, 向 CPU 申请中断。

② 接收

方式 1 的接收时序如图 4-16 (b) 所示。允许接收位 REN 被置 1 接收器就开始工作, 跳变检测器以波特率 16 倍的速率采样 RXD 引脚上电平, 当采样到从 1 到 0 的负跳变时, 启动接收控制器接收数据, 由于发送、接收双方各自使用自己的时钟, 两者的频率总有少许差异。为了避免这种影响, 控制器将 1 位送时间等分成 16 分、位检测器在 7、8、9 三个状态也就是在位信号中央采样 RXD 三次。而且, 三次采样中至少两次相同的值被确认为数据, 这是为了减小干扰的影响。如果起始位接收到的值不是 0, 由于起始位无效, 接收电路复位。如果起始位为 0, 则开始接收本帧其他数据。控制器发出的内部移位脉冲将 RXD 上的数据逐位移入移位寄存器, 当 8 位数据及停止位全部移入后:

- 如果 $RI=0$ 、 $SM2=0$ ，接收控制器发出“装载 SBUF”信号，将 8 位数据装入接收数据缓冲器 SBUF，停止装入 RB8，并置 $RI=1$ ，向 CPU 申请中断。
- 如果 $RI=0$ 、 $SM2=1$ ，那么只有停止位为 1 时才发生上述动作。
- 如果 $RI=0$ 、 $SM2=1$ 且停止位为 0，（通常由传输过程中的干扰所致）所接收的数据应会丢失，不再恢复。
- 如果 $RI=1$ ，则所接收的数据在任何情况下都会丢失。

无论出现哪一种情况，跳变检测器将继续采样 RXD 引脚的负跳变，以便接收下一帧信息。

接收器采用移位寄存器和 SBUF 双缓冲结构，以避免在接收后一帧数据之前，CPU 尚未及时响应中断将前一帧数据取走，造成两帧数据重叠问题。采用双缓冲器后，前、后两帧数据进入 SBUF 的时间间隔至少有 10 个位传送周期。在后一帧数据送入 SBUF 之前，CPU 有足够的时间将前一帧数据取走。

（3）方式 2 和方式 3

串行口工作在方式 2、方式 3 时，为 9 位异步通信接口。发送或接收的一帧信息由 11 位组成。方式 2 与方式 3 仅波特率不同，方式 2 的波特率为 $f_{osc}/32$ （ $SMOD=1$ 时）或 $f_{osc}/64$ （ $SMOD=0$ 时），而方式 3 的波特率由定时/计数器 0 或定时/计数器 1 及 SMOD 决定。

方式 2 和方式 3 下的接收发送时序如图 4-17 所示。

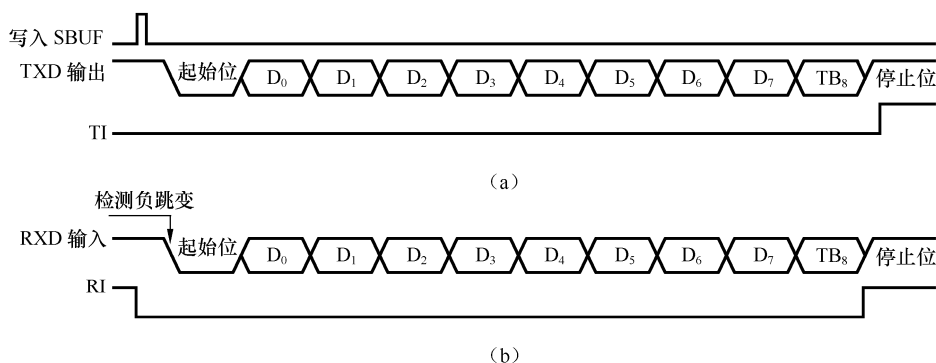


图 4-17 串口方式 2、3 通信时序

（a）发送时序 （b）接收时序

① 发送

方式 2、方式 3 发送时，数据从 TXD 引脚输出，附加的第 9 位数据由 SCON 中的 TB8 提供。CPU 执行一条写入 SBUF 的指令后立即启动发送器发送，发送完一帧信息后由硬件置 $TI=1$ 。

② 接收

与方式 1 相似，REN 置 1 后，跳变检测器不断对 RXD 引脚采样，当采样到负跳变后就启动接收控制器。位检测器对每位数据采集 3 个值，用采 3 取 2 办法确定每位的数值。当第 9 位数据移入移位寄存器后，将 8 位数据装入 SBUF，第 9 位数据装入 RB8，并置 $RI=1$ 。

与方式 1 相同，方式 2、方式 3 中也设置有数据辨别功能，即当 $RI=1$ 或 $SM2=1$ 且第 9 位数据为 0 时所接收的一帧信息被丢失。

4. 波特率的设计

在串行通信中，收发双方对发送或接收的数据速率（即波特率）要有一定的约定，这可通过对串行口编程设定。各种工作模式下其波特率的设置均有所不同，其中方式 0 和方式 2 的波特率是固定的，方式 1 和方式 3 的波特率是可变的，由定时器 T1 的溢出率确定。

(1) 方式 0 的波特率

方式 0 时，其波特率固定为振荡频率的 1/12，并不受 PCON 中 SMOD 位的影响。因而，方式 0 的波特率= $f_{osc}/12$ 。

(2) 方式 2 的波特率

方式 2 的波特率由系统的振荡频率 f_{osc} 和 PCON 的最高位 SMOD 确定，即为 $2SMOD \times f_{osc}/64$ 。在 SMOD=0 时，波特率= $f_{osc}/64$ ；在 SMOD=1 时，波特率= $f_{osc}/32$ 。

(3) 方式 1、3 的波特率

C51 串行口方式 1、3 的波特率由定时器 T1 的溢出率和 SMOD 的值共同确定，即：

方式 1、3 的波特率= $(2SMOD/32) \times$ 定时器 T1 的溢出率。

其中定时器溢出率取决于计数速率和定时器的预置值。计数速率与 TMOD 寄存器中 C/T 的设置有关。当 C/T=0 时，为定时方式，计数速率= $f_{osc}/12$ ；当 C/T=1 时，为计数方式，计数速率取决于外部输入时钟的频率，但不能超过 $f_{osc}/24$ 。

定时器的预置值等于 $M-X$ ， X 为计数初值， M 为定时器的最大计数，其和操作模式有关。（可取 2^{13} 、 2^{16} 、 2^8 ）。如果为了达到很低的波特率，则可以选择 16 位的操作模式，即模式 1 或模式 0，可以利用 T1 中断来实现重装计数值。

为能实现定时器计数初值重装，则通常选择操作模式 2。在模式 2 中，TL1 作计数用，TH1 用于保存计数初值，当 TL1 计满溢出时，TH1 的值自动装到 TL1 中，因此一般选用 T1 工作于模式 2 作波特率发生器。设 T1 的计数初值为 X ，设置 C/T=0 时，那么每过 $256-X$ 个机器周期，定时器 T1 就会产生一次溢出。则 T1 的溢出周期为：

$$\text{溢出周期} = 12/f_{osc} \times (256 - X)$$

溢出率为溢出周期之倒数，所以

$$\text{波特率} = \frac{2^{SMOD}}{32} \times \frac{f_{osc}}{12 \times (256 - X)} = \frac{2^{SMOD} \times f_{osc}}{384 \times (256 - X)}$$

定时器 T1 模式 2 的计数初值由上式可得：

$$X = 256 - \frac{2^{SMOD} \times f_{osc}}{384 \times \text{波特率}}$$

5. 串行口初始化

(1) 串行口波特率

通常使用单片机的串行口时，选用的晶振比较固定，通常为 6MHz、12MHz 或 11.0592MHz。串行口波特率常用于和微机的通信，选用的波特率也相对固定。串行口常用的波特率及相应的设置如表 4-6 所示。

表 4-6

串行口的常用波特率

串行口工作方式	波特率	$f_{osc}=6\text{MHz}$	$f_{osc}=12\text{MHz}$	$f_{osc}=11.059\text{MHz}$
		SMOD TMOD TH1	SMOD TMOD TH1	SMOD TMOD TH1
方式 0	1MHz		× × ×	
方式 2	375k		1 × ×	
	187.5k	1 × ×	0 × ×	
方式 1 或 方式 3	62.5k		1 20 FFH	
	19.2k			1 20 FDH
	9.6k			0 20 FDH
	4.8k		1 20 F3H	0 20 FAH
	2.4k	1 20 F3H	1 20 F3H	0 20 F4H
	1.2k	1 20 FFH	0 20 E6H	0 20 E8H
	600	1 20 E6H	0 20 CCH	0 20 D0H
	300	0 20 CCH	0 20 98H	0 20 A0H
	137.5	1 20 1DH	0 20 1DH	0 20 2EH
	110	0 20 72H	0 10 FEEBH	0 10 FEFH

(2) 初始化步骤

在使用串行口之前，应对它进行编程初始化，主要是设置产生波特率的定时器 1、串行口控制和中断控制。具体步骤如下所示：

- ① 确定定时器 1 的工作方式，即编程 TMOD 寄存器；
- ② 计算定时器 1 的初值，即装载 TH1、TL1；
- ③ 启动定时器 1，即编程 TCON 中的 TR1 位；
- ④ 确定串行口的控制，即编程 SCON；
- ⑤ 串行口在中断方式工作时，须开 CPU 和源中断，即编程 IE 寄存器。

6. 串行口的应用编程

【例 4-8】单片机 $f_{osc}=11.059\text{MHz}$ ，波特率 9 600，各设置 32 个字节的队列缓冲区用于发送接收。设计单片机和终端或另一计算机通信的程序。

分析：单片机串行口初始化成 9 600 波特，中断程序双向处理字符，程序双向缓冲字符。背景程序可以“放入”和“提取”在缓冲区的字符串，而实际传入和传出 SBUF 的动作由中断完成。

Loadmsg 函数加载缓冲数组，标志发送开始。缓冲区分为发 (t) 和收 (r) 缓冲。缓冲区通过两种指示 (进 in 和出 out) 和一些标志 (满 full、空 empty、完成 done) 管理。队列缓冲区 32 字节长为循环队列，由简单的逻辑与 (&) 操作管理，它比取模 (%) 操作运行更快。当 $r_in=r_out$ 接收缓冲 (r_buf) 区满，不能再有字符插入；当 $t_in=t_out$ ，发送缓冲区 (t_buf) 空，发送中断清除，停止 UART 请求。具体程序 sio36.C 如下：

```
#include<reg51.h>
#define uchar unsigned char
uchar xdata r_buf[32];          /*iteml*/
```

```

uchar xdata t_buf[32];
uchar r_in, r_out, t_in, t_out;      /*队列指针*/
bit r_full, t_empty, t_done;         /*item2*/
code uchar m[]={"this is a test program\r\n"};

serial() interrupt 4 using 1
{
    /*item3*/
    if(RI&&~r_full)
    {
        r_buf[r_in]=SBUF;
        RI=0;
        r_in=++r_in&0x1f;
        if(r_in==r_out)r_full=1;
    }
    else if(TI &&~t_empty)
    {
        SBUF=t_buf[t_out];
        TI=0;
        t_out=++t_out&0x1f;
        if(t_out==t_in)t_empty=1;
    }
    else if(TI)
    {
        TI=0;
        t_done=1;
    }
}

void loadmsg(uchar code *msg)
{
    /*item4*/
    while((*msg!=0)&&(((t_in+1)^t_out)&0x1f)!=0))
    {
        /*测试缓冲区满*/
        t_buf[t_in]=*msg;
        msg++;
        t_in=++t_in&0x1f;
        if(t_done)
        {
            TI=1;
            T_empty=t_done=0;
        }
    }
}

```

```

        }
    }
}

/*若完成重新开始*/

void process(uchar ch){return;}    /*item5*/
/*用户定义*/

void processmsg(void)
{
    /*item6*/
    while(((r_out+1)^r_in)!=0)
    {
        /*接收缓冲区非空*/
        process(r_buf[r_out]);
        r_out=++r_out&0x1f;
    }
}

main()
{
    /*item7*/
    TMOD=0x20;    /*定时器 1 方式 2*/
    TH1=0xfd;    /*9600 波特 11.0592 MHz*/
    TCON=0x40;    /*启动定时器 1*/
    SCON=0x50;    /*允许接收*/
    IE=0x90;    /*允许串行口中断*/
    t_empty=t_done=1;
    r_full=0;
    r_out=t_in=t_out=0;
    r_in=1;    /*接收缓冲和发送缓冲置空*/
    for(;;)
    {
        loadmsg(&m);
        processmsg();
    }
}

```

在程序中，

item1: 背景程序“放入”和“提取”字符的队列缓冲区；

item2: 缓冲区状态标志；

item3: 串行口中断服务程序，从 RI、TI 判别接收或发送中断，由软件清除。判别缓冲

区状态（满 full 和空 empty）和全部发送完成（done）；

item4: 此函数把字符串放入发送缓冲，准备发送；

item5: 接收字符的处理程序，实际应用自定义；

item6: 此函数逐一处理接收缓冲区的字符；

item7: 主程序即背景程序，进行串行口的初始化、载入字符串、处理接收的字符串。

习题和思考题

4-1 C51 系列单片机有几个中断源，各中断标志是如何产生的，又是如何清 0 的？CPU 响应中断时，它们的中断矢量地址分别是多少？

4-2 C51 系列单片机的中断系统中有几个优先级？如何设定？

4-3 设 $f_{osc}=6\text{MHz}$ ，外部中断采用电平触发方式，那么中断请求信号的低电平至少应持续多少 μs ？

4-4 8031 芯片的 $\overline{\text{INT0}}$ 、 $\overline{\text{INT1}}$ 引脚分别输入压力超限、温度超限中断请求信号，定时/计数器 0 作定时检测的实时时钟，用户规定的中断优先权排队次序为：压力超限 > 温度超限 > 定时检测，要求确定 IE、IP 的内容，以实现上述要求。

4-5 设 $f_{osc}=12\text{MHz}$ ，8051 定时器 0 的方式 0、方式 1、方式 2 的最大定时间隔分别是多少？

4-6 希望 8051 单片机定时器 0 的定时值以内部 RAM 的 20H 单元的内容为条件而可变：当 (20H)=00H 时，定时值为 10ms；当 (20H)=01H 时，定时值为 20ms。请根据以上要求对定时器 0 初始化。设单片机时钟频率为 12MHz。

4-7 串行接口有哪几种工作方式？各有什么特点？

4-8 什么是波特率？某异步串行通信接口每分钟传送 1800 个字符，每个字符由 11 位组成，请计算出传送波特率。

4-9 若异步通信接口按方式 3 传送，已知每分钟传送 3600 个字节，其波特率是多少？

第 5 章 C51 单片机的系统扩展

重点与难点

单片机系统扩展的基本概念、单片机程序存储器扩展、单片机数据存储器扩展、单片机 I/O 口扩展。

基本要求

1. 掌握单片机系统扩展的基本概念
2. 掌握单片机程序存储器的扩展方法
3. 掌握单片机数据存储器的扩展方法
4. 掌握常用扩展芯片 8255、8253、8155、I²C 接口芯片 24C02、SPI 接口芯片 X5045 和单片机配置 FPGA 的使用方法

本章内容

1. 系统扩展概述
2. 程序存储器的扩展
3. 数据存储器的扩展
4. 可编程外围并行接口 8255
5. 可编程外围定时器 8253
6. 可编程并行接口芯片 8155
7. I²C 接口芯片 AT24C01
8. SPI 接口芯片 X5045
9. 单片机配置 FPGA

5.1 系统扩展概述

5.1.1 C51 单片机的最小系统

1. C51 的最小系统

C51 是片内具有 ROM 的单片机，需要外接晶体振荡器和复位电路就可构成最小系统。该最小系统的特点主要体现在以下几个方面：

- (1) P0, P1, P2, P3 口都可作为 I/O 口使用;
- (2) 内部程序存储器容量仅为 4KB;
- (3) 由于不采用外部程序存储器, $\overline{EA}$ 应接高电平。

2. 8031 的最小系统

由于 8031 是片内无 ROM 的单片机, 因此它必须外接程序存储器, 并使用地址锁存器才能构成一台完整的没有扩展的最小系统。该最小系统的特点主要体现在以下几个方面:

- (1) 由于 P0、P2 口需要提供访问存储器的地址 A0~A15, 因此这两个 8 位口不能作为一般的 I/O 口使用;
- (2) 存储器芯片的容量可根据需要选择;
- (3) P0 口地址线 and 数据线是复用的, 它必须用到地址锁存器;
- (4) 由于采用外部程序存储器, $\overline{EA}$ 应接地 (低电平)。

5.1.2 系统扩展及结构

1. 系统扩展结构

单片机系统的扩展结构如图 5-1 所示, 可见扩展内容包括 ROM、RAM 和 I/O 接口电路等。扩展是通过系统总线进行的, 通过总线把各扩展部件连接起来, 并进行数据、地址和信号的传送, 要实现扩展首先要构造系统总线。

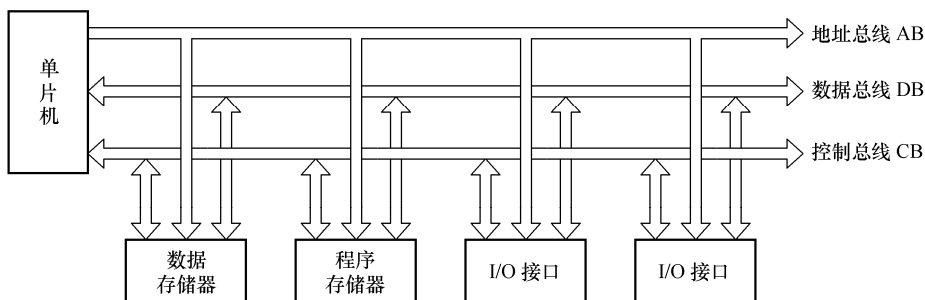


图 5-1 单片机系统的扩展结构

2. 系统总线及总线构造

(1) 系统总线

① 地址总线 (Address Bus, 简写 AB)

地址总线上传送的是地址信号, 用于存储单元和 I/O 端口的选择。地址总线是单向的, 地址信号只能由单片机向外送出。

② 数据总线 (Data Bus, 简写 DB)

数据总线用于在单片机与存储器之间或单片机与 I/O 端口之间传送数据。数据总线是双向的, 可以进行双向的数据传送, 也就是说, 单片机可以向存储器或 I/O 端口传送数据, 存

存储器之间可以传送数据，存储器或 I/O 端口可以向单片机传送数据。

③ 控制总线（Control Bus，简写 CB）

控制总线实际上是一组控制信号线，包括单片机发出的以及从其他部件传送给单片机的控制信号。对于一条具体的控制信号线来说，其传送方向是单向的，但是由不同方向的控制信号线组合的控制总线则表示为双向。

（2）总线构造

单片机与其他微型计算机不同，为了减少芯片封装引脚，单片机芯片并没有提供专用的地址线 and 数据线，而是采用 I/O 口线的复用技术，把 I/O 口线改造为总线。C51 单片机地址线 and 数据线构造情况如图 5-2 所示。“构造”总线的具体方法包括以下几个方面。

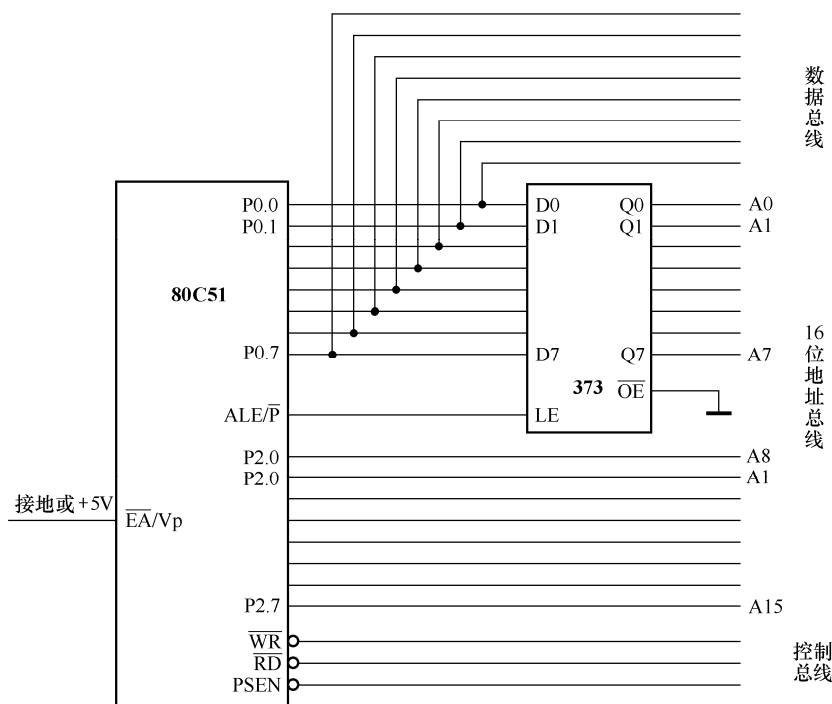


图 5-2 C51 单片机扩展总线构造图

① 以 P0 口的 8 位口线作地址/数据线。

② 以 P2 口的口线作高位地址线。

③ 控制信号线：

- 使用 ALE 作地址锁存的选通信号，以实现低 8 位地址的锁存；
- 以 $\overline{\text{PSEN}}$ 信号作扩展程序存储器的读选通信号；
- 以 $\overline{\text{EA}}$ 信号作为内外程序存储器的选通信号；
- 以 $\overline{\text{RD}}$ 和 $\overline{\text{WR}}$ 作为扩展数据存储器和 I/O 端口的读写选通信号。

（3）单片机的串行扩展技术

串行扩展是通过串行接口实现的，这样可以减少芯片的封装引脚，降低成本，简化系统结构，增加系统扩展的灵活性。

5.1.3 C51 单片机存储器扩展与编址技术

1. 存储器扩展概述

存储器扩展是单片机系统扩展的主要内容，由于扩展是在单片机芯片之外进行的，因此通常把扩展的程序存储器（ROM）称为外部 ROM，把扩展的数据存储器（RAM）称为外部 RAM。

2. 单片机存储器系统

C51 单片机的存储器结构和存储空间分配如图 5-3 所示。

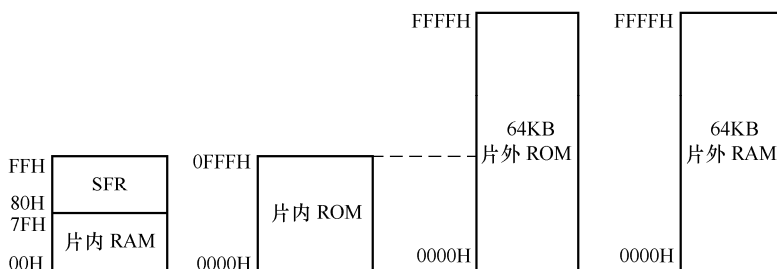


图 5-3 单片机系统的存储器结构和存储空间分配

可见，整个存储器系统包括程序存储器（ROM）和数据存储器（RAM）两部分。扩展 ROM 的地址与芯片内是否有程序存储有关。如果没有片内程序存储器，则扩展 ROM 的地址从 0000H 开始；如果有片内程序存储，则扩展 ROM 的地址从 1000H 开始。而扩展 RAM 的地址，不管容量大小，都是从 0000H 开始。

3. 扩展存储器编址技术

（1）扩展存储器编址概述

存储器编址就是如何使用系统提供的地址线，通过适当连接最终达到一个存储单元只唯一地对应一个地址的要求。芯片内存储单元已经编址，只有扩展存储器才有编址问题存在。

芯片内部存储单元的编址由芯片内的译码电路完成。对设计者来说，只需把存储器芯片的地址引脚与相应的系统地址线直接连接即可，几乎没什么技术可言。由于芯片的选择要由设计者完成，而且比较复杂，因此所谓的存储器编址，实际上主要是研究芯片的选择问题。为了芯片选择的需要，存储器芯片都有片选信号引脚，因此芯片选择的实质就是如何产生芯片的片选信号。

（2）存储器扩展的编址技术

① 线选法

线选法就是直接以系统的地址位作为存储器芯片的片选信号，为此只需把用到的地址线与存储器芯片的片选端直接连接即可。线选法编址的特点是简单明了，且不需要另外增加电路；但这种编址方法对存储空间的使用是断续的，不能够充分有效地利用存储空间，扩充存储容量受限，只适用于小规模单片机系统的存储器扩展。

② 译码法

译码法就是使用译码器对系统的高位地址进行译码，以其译码输出作为存储器芯片的片选信号。这是一种最常用的存储器编址方法，能有效地利用存储空间，适用于大容量多芯片存储器扩展。译码电路可以使用现有的译码器芯片，常用的译码芯片有 74LS139（双 2-4 译码器）和 74LS138（3-8 译码器）等，它们的 CMOS 型芯片分别为 74HC138 和 74HC139。

线选法和译码法的连接图如图 5-4 和图 5-5 所示。

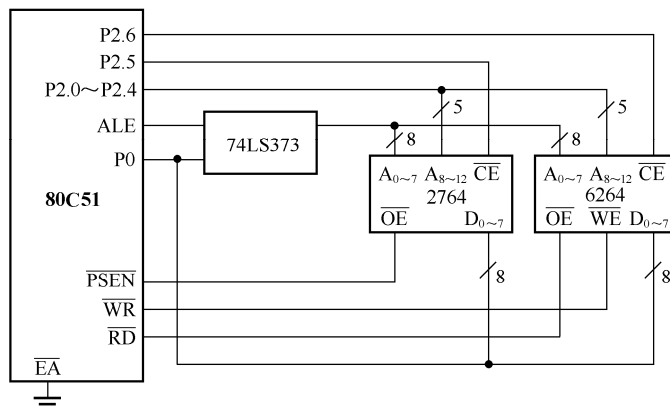


图 5-4 线选法扩展两片 RAM

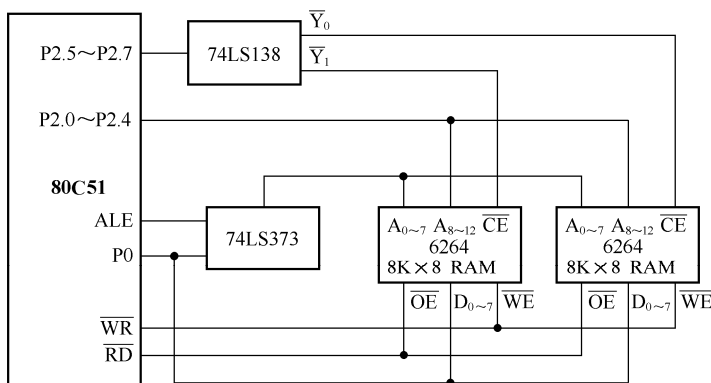


图 5-5 译码法扩展两片 RAM

5.2 程序存储器的扩展

5.2.1 扩展 EPROM

1. 单片程序存储器扩展

EPROM 是单片机最常选用的程序存储器芯片，它是一种紫外线可擦除、电可编程的存储器，最经常使用的是 27C 系列的 EPROM，如 27C16（2K）、27C32（4K）、27C64（8K）、27C128（16K）、27C256（32K），其芯片引脚如图 5-6 所示。除了 27C16 和 27C32 为 24 脚外，其余均为 28 脚。芯片型号的高两位数字 27 是表示 EPROM，低位数字表示存储容量的 Kbit

值，如 2764 表示 64K 个存储位的 EPROM，低位数字除以 8 为芯片的字节存储容量，如 2764 表示字节容量为 8K，可以表示成 $8K \times 8$ 。

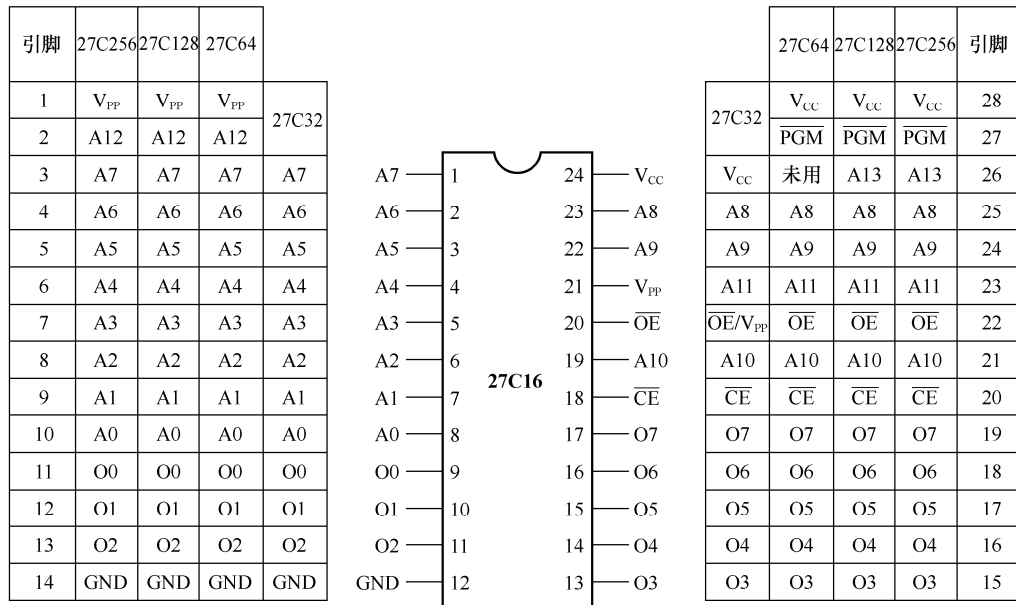


图 5-6 常用 EPROM 芯片的引脚图

这里以单片 27C16 为例，说明程序存储器扩展的有关问题，其扩展连接如图 5-7 所示。

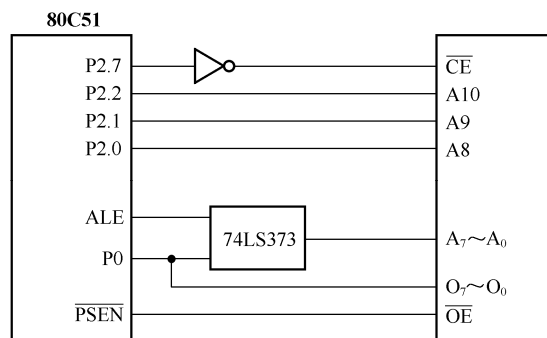


图 5-7 单片程序存储器（ROM）扩展连接

(1) 存储器扩展的主要工作是地址线、数据线和控制信号线的连接。

① 地址线：地址线的连接与存储器芯片的容量有直接关系。采用线选法编址比较方便，因此只需在剩下的高位地址线中取 P2.7 作芯片选择信号与 27C16 的 $\overline{CE}$ 端相连即可。

② 数据线：数据线的连接则比较简单，只要把存储芯片的数据输出引脚与单片机 P0 口线对应连接即可。

③ 控制线：程序存储器的扩展只涉及到 $\overline{PSEN}$ （外部程序存储器读选通）信号，把该信号连接 27C16 的 $\overline{OE}$ 端，以便进行存储单元的读出选通。

(2) 分析存储器在存储空间中占据的地址范围，实际上就是根据地址线连接情况确定其最低地址和最高地址。如把 P2 口中没用到的高位地址线假定为“0”状态，则本例 27C16 芯

号, 当 P2.7=0 时, 选左片, 其地址范围为 0000H~1FFFH; 当 P2.7=1 时, 选择右片, 其地址范围为 8000H~9FFFH。

② 数据线: 把存储器芯片的数据输出引脚与单片机 P0 口线对应相连。

③ 控制线: 把 $\overline{\text{PSEN}}$ 分别与两片 27C64 的 $\overline{\text{OE}}$ 端相连。

对于多片程序存储器的扩展, 可以得出以下几个设计要点:

- (1) 各芯片的低位地址线并行连接。
- (2) 各芯片的数据线并行连接。
- (3) 各芯片的控制信号 $\overline{\text{PSEN}}$ 并行连接。
- (4) 各芯片的片选信号是不同的, 需要分别产生。

5.2.2 扩展 (E^2PROM)

E^2PROM 是一种电可擦除可编程的存储器, 最经常使用的并口 E^2PROM 有高压编程的 28C16 (2K)、28C17 (2K) 和 28C64 (8K), 低压编程的 28C16A (2K)、28C17A (2K) 和 28C64A (8K) 等; 最经常使用的串口 E^2PROM 有 24C01 (1K)、24C02 (2K)、24C04 (4K) 等, 它们将在 5.7 节中进行介绍。

其中, 28C16 和 28C17 都是 2K 的存储器, 它们的区别在于: 一种有 28 个引脚, 另一种只有 24 个引脚; 另外 28C17 增加了一种检测写周期结束的方法, 利用增加的引脚 RDY/ $\overline{\text{BUSY}}$ 来表示写操作何时完成。当写操作开始后, 该引脚输出为低电平, 当写操作结束时, 该引脚输出高电平, 这样通过该引脚的信号就可以指示与其接口的微处理器进行合适的操作。

当使用 E^2PROM 作为存储器使用时, 它与 CPU 的接线和使用 EPROM 的扩展接线非常相似, 也是按照三总线的扩展原则进行接线, 不同之处在于为了在线改写, 单片机的 $\overline{\text{WR}}$ 端与 E^2PROM 的 $\overline{\text{WE}}$ 端相接。

5.3 数据存储器的扩展

随机存储器 (Random Access Memory) 简称为 RAM, 在单片机系统中用于存放可随时修改的数据, 因此在单片机领域中也称为数据存储器。与 ROM 不同, 对 RAM 可以进行读、写两种操作。但 RAM 是易失性存储器, 断电后所存信息立即消失。按照半导体工艺, RAM 分为 MOS 型和双极型两种, MOS 型集成度高、功耗低、价格便宜, 但速度较慢, 而双极型的特点则正好相反。在单片机系统中使用的大多数是 MOS 型的随机存储器, 它们的输入输出信号能与 TTL 相兼容, 所以在扩展中信号连接是很方便的。

数据存储器扩展常使用随机存储器芯片, 用得较多的是 Intel 公司的 6116 和 6264。这里以 6116 为例进行说明, 其引脚如图 5-9 所示。

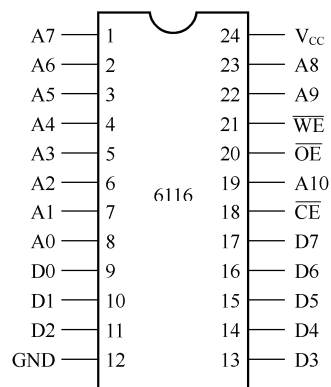


图 5-9 静态 RAM6116 引脚

6116 芯片为引脚双列直插封装。下面对芯片引脚进行一下说明。

A10~A0: 地址线;

D7~D0: 数据线;

$\overline{\text{CE}}$: 片选信号;

$\overline{\text{OE}}$: 数据输出允许信号;

$\overline{\text{WE}}$: 写选通信号;

V_{CC}: 电源 (+5V);

GND: 地。

6116 共有 4 种工作方式, 如表 5-1 所示。

表 5-1 6116 工作方式

状 态	$\overline{\text{CE}}$	$\overline{\text{OE}}$	$\overline{\text{WE}}$	D7~ D0
未 选 中	1	×	×	高 阻
禁 止	0	1	1	高 阻
读 出	0	0	1	数据读出
写 入	0	1	0	数据写入

5.3.1 单片数据存储器扩展

使用一片 6116 实现 2KB RAM 扩展, 其电路连接如图 5-10 所示。

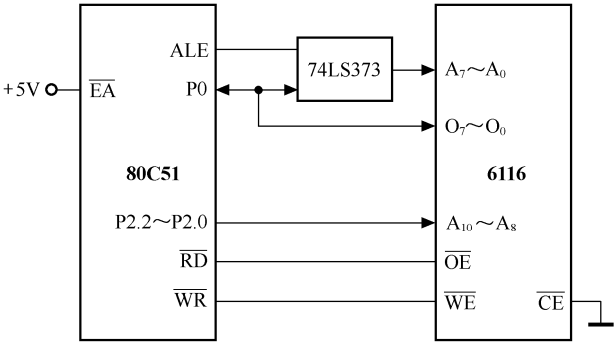


图 5-10 单片 RAM 扩展连接图

扩展连接中, $\overline{\text{RD}}$ 信号接 6116 的 $\overline{\text{OE}}$ 端, 信号 $\overline{\text{WR}}$ 接 6116 的 $\overline{\text{WE}}$ 端, 以进行 RAM 芯片的读写控制。由于假定系统中有一片 6116, 因此没有使用片选信号, 而把 $\overline{\text{CE}}$ 端直接接地。这种情况下, 6116 的地址范围是 0000H~07FFH。

5.3.2 线选法多片数据存储器扩展

用 4 片 6116 实现 8KB 数据存储器扩展, 其电路连接如图 5-11 所示。

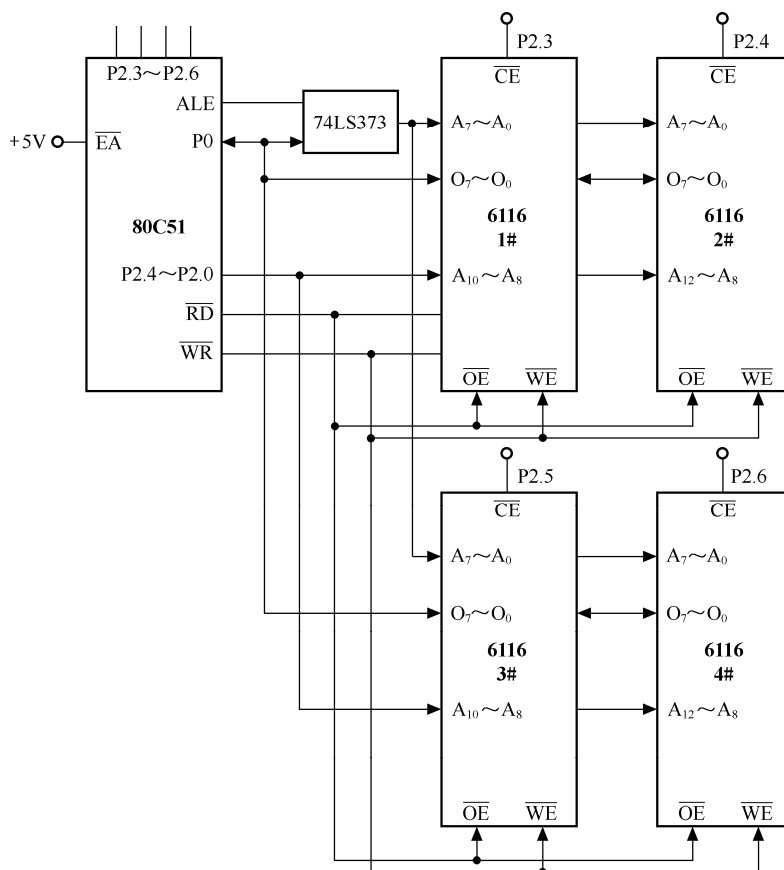


图 5-11 线选法 4 片 RAM 扩展连接图

多片 RAM 扩展时，读写选通信号以及 A10~A0 地址引线的连接与单片 RAM 的扩展相同。特殊之处只在于高位地址线的连接。这时 P2 口的高 8 位地址线中 P2.2~P2.0 已用作 RAM 芯片的高 3 位地址 (A10~A8)，尚余下 5 条地址线，为此把其中的 P2.3、P2.4、P2.5、P2.6 分别作为 4 片 RAM 的片选信号，从而构成一个完整的线选法编址的 8KB RAM 扩展存储器。

线选法存储器扩展系统中各存储芯片的存储映像如表 5-2 所示。

表 5-2 线选法存储器扩展中各存储芯片的存储映像

6116 芯片	P2 口								P1 口								相 应 地 址	
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
I #	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	最低地址	7000H
	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	最高地址	77FFH
II #	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	最低地址	6800H
	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	最高地址	6FFFH

续表																		
6116 芯片	P2 口								P1 口								相 应 地 址	
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
III#	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	最低地址	5800H
	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	最高地址	5FFFH
IV#	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	最低地址	3800H
	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	最高地址	3FFFH

5.3.3 译码法多片数据存储器扩展

同样以 4 片 6116 进行 8KB 数据存储器扩展，不过以译码法实现，其译码电路如图 5-12 所示，其他电路连接与图 5-11 完全相同。

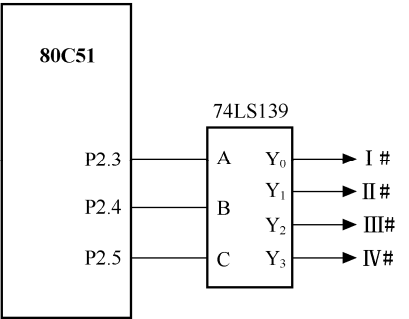


图 5-12 译码法扩展时的译码电路

可见，图中使用 74LS139 作译码器，其译码输出 Y0、Y1、Y2、Y3 依次作为存储器芯片 I#、II#、III#、IV#的片选信号。
译码法存储器扩展系统中各存储芯片的存储映像如表 5-3 所法。

表 5-3 译码法存储器扩展中各存储芯片的存储映像

6116 芯片	P2 口								P1 口								相 应 地 址	
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
I #	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	最低地址	0000H
	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	最高地址	07FFH
II #	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	最低地址	0800H
	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	最高地址	0FFFH
III#	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	最低地址	1000H
	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	最高地址	17FFH

续表																		
6116 芯片	P2 口								P1 口								相 应 地 址	
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
IV#	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	最低地址	1800H
	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	最高地址	1FFFH

5.4 可编程外围并行接口 8255A

5.4.1 8255A 的引脚功能和逻辑结构

8255A 是可编程的并行输入/输出接口芯片，通用性强且使用灵活，常用来实现 C51 系列单片机的并行 I/O 口扩展。它是一个有 40 引脚的双列直插式集成电路芯片，有 A、B、C 三个可编程的 8 位 I/O 接口。

8255A 的引脚如图 5-13 所示，引脚功能如表 5-4 所示。

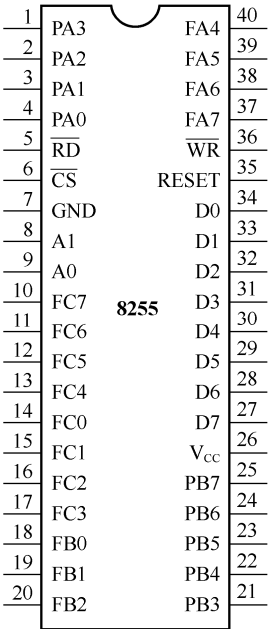


图 5-13 8255A 引脚图

8255A 的内部结构按功能可分为 3 个逻辑电路部分，即口电路、总线接口电路和控制电路，如图 5-14 所示。

表 5-4 8255A 芯片的引脚功能

名 称	引 脚	功 能
RESET	35	复位信号输入端，使扩展字寄存器等各内部寄存清除，置 A、B、C 口为输入

续表

名 称	引 脚	功 能
D7~D0	27~34	双向数据总线
PA7~PA0	37~40, 1~4	A 口的 8 位 I/O 引脚
PB7~PB0	25~18	B 口的 8 位 I/O 引脚
PC7~PC0	10~13, 17~14	C 口的 8 位 I/O 引脚
$\overline{\text{CS}}$	6	片选端
$\overline{\text{RD}}$	5	读信号输入端, 使 8255A 送数据或状态信息到 CPU
$\overline{\text{WR}}$	36	写信号输入, 使 CPU 输出数据或控制字到 8255A
A1、A0	8、9	地址线, 通过 A1、A0 不同的组合来选择控制寄存器或 A、B、C 口中的一个
GND	7	地线
V _{CC}	26	+5V 电源

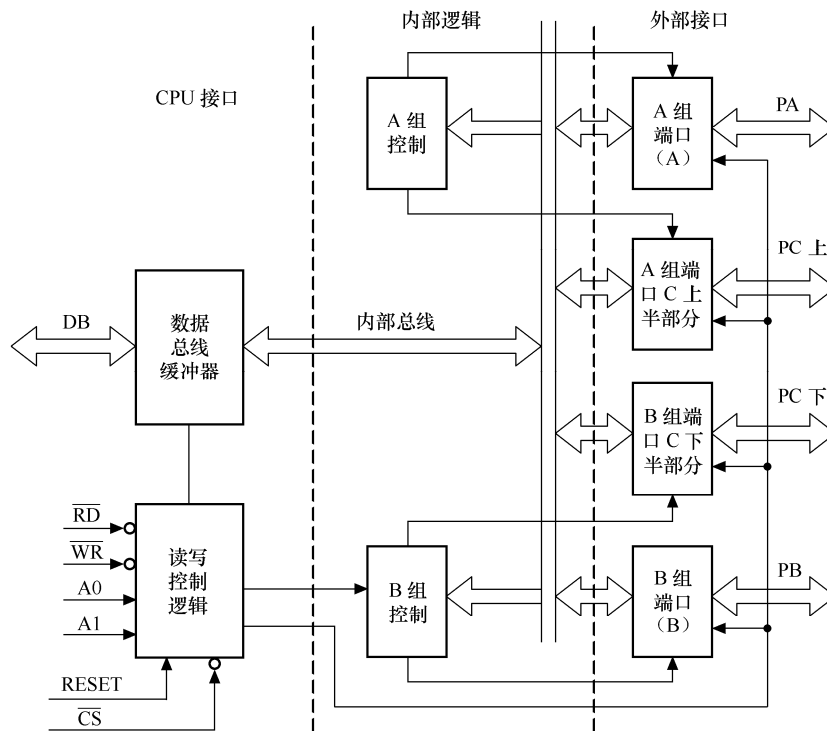


图 5-14 8255A 内部逻辑结构图

1. 口电路

8255A 共有 3 个 8 位口，其中 A 口和 B 口是单纯的数据口，供数据 I/O 使用；而 C 口则既可以作数据口，又可以作控制口使用，用于实现 A 口和 B 口的控制功能。因此在使用中常把 C 口分为两部分，即：C 口高位部分（PC7~PC4）；C 口低位部分（PC3~

PC0)。

数据传送中 A 口所需的控制信号由 C 口高位部分提供，因此把 A 口和 C 口高位部分合在一起称为 A 组；同样，把 B 口和 C 口低位部分合在一起称为 B 组。

2. 总线接口电路

总线接口电路用于实现 8255A 和单片机芯片的信号连接，其中包括：

(1) 数据总线缓冲器

(2) 读/写控制逻辑

$\overline{CS}$ ：片选信号（低电平有效）。

$\overline{RD}$ ：读信号（低电平有效）。

$\overline{WR}$ ：写信号（低电平有效）。

A1、A0：端口选择信号。8255A 共有 4 个可寻址的端口（A 口、B 口、C 口和控制寄存器），用 2 位地址编码就可实现选择。在 I/O 扩展连接时通常把 A0 和 A1 直接与单片机的 P0.0 和 P0.1 相连，以确定口地址。

RESET：复位信号（高电平有效）。复位之后，控制寄存器清除，各端口被置为输入方式。

读/写控制逻辑用于实现 8255A 的硬件管理，其内容包括芯片的选择、口的寻址和规定各端口和单片机之间的数据传送方向。8255A 读写控制逻辑如表 5-5 所示。

表 5-5 8255A 读/写控制

$\overline{CS}$	A1	A0	$\overline{RD}$	$\overline{WR}$	所选端口	操 作
0	0	0	0	1	A 口	读端口 A
0	0	1	0	1	B 口	读端口 B
0	1	0	0	1	C 口	读端口 C
0	0	0	1	0	A 口	写端口 A
0	0	1	1	0	B 口	写端口 B
0	1	0	1	0	C 口	写端口 C
0	1	1	1	0	控制寄存器	写控制字
1	×	×	×	×	/	数据总线缓冲器输出高阻抗

3. 控制逻辑电路

控制逻辑电路包括 A 组控制和 B 组控制，合在一起构成 8 位控制寄存器，用于存放各口的工作方式控制字，具体逻辑电路不再介绍。

5.4.2 8255A 的工作方式及数据 I/O 操作

1. 8255A 的工作方式

(1) 方式 0：基本输入/输出方式

方式 0 下, 可供使用的是两个 8 位口 (A 口和 B 口) 及两个 4 位口 (C 口高位部分和低位部分)。4 个口可以是输入和输出的任何组合。方式 0 适用于无条件数据传送, 也可以把 C 口的某一位作为状态位, 实现查询方式的数据传送。

(2) 方式 1: 选通输入/输出方式

方式 1 下, A 口和 B 口分别用于数据的输入/输出, 而 C 口则作为数据传送的联络信号。方式 1 下 C12 联络信号的具体定义如表 5-6 所示。

表 5-6 C 口联络信号定义

C 口位线	方式 1		方式 2	
	输 入	输 出	输 入	输 出
PC7		$\overline{\text{OBFA}}$		$\overline{\text{OBFA}}$
PC6		$\overline{\text{ACKA}}$	$\overline{\text{ACKA}}$	
PC5	IBFA		IBFA	
PC4	$\overline{\text{STBA}}$		$\overline{\text{STBA}}$	
PC3	INTRA	INTRA	INTRA	INTRA
PC2	$\overline{\text{STBB}}$	$\overline{\text{ACKB}}$		
PC1	IBFB	$\overline{\text{OBFB}}$		
PC0	INTRB	INTRB		

(3) 方式 2: 双向数据传送方式

只有 A 口才能选择这种工作方式, 这时 A 口既能输入数据又能输出数据, 这种方式下需使用 C 口的 5 位口线作控制线, 其信号定义如表 5-6 所示。方式 2 适用于查询或中断方式的双向数据传送。如果把 A 口置于方式 2 下, 则 B 工作于方式 0 下。

2. 数据输入操作

- (1) $\overline{\text{STB}}$ (STroBe): 选通脉冲 (输入), 低电平有效。
- (2) IBF (Input Buffer Full): 输入缓冲器满信号 (输出), 高电平有效。
- (3) INTR (INTerrupt Request): 中断请求信号 (输出), 高电平有效。

3. 输出操作

- (1) $\overline{\text{ACK}}$ (ACKnowledge): 外设响应信号 (输入), 低电平有效。
- (2) $\overline{\text{OBF}}$ (Output Buffer Full): 输出缓冲器满信号 (输出), 低电平有效。
- (3) INTR: 中断请求信号 (输出), 高电平有效。

5.4.3 8255A 控制字及初始化

1. 工作方式控制字

工作方式控制字用于确定各口的工作方式及数据传送方向, 其具体格式如图 5-15 所示。

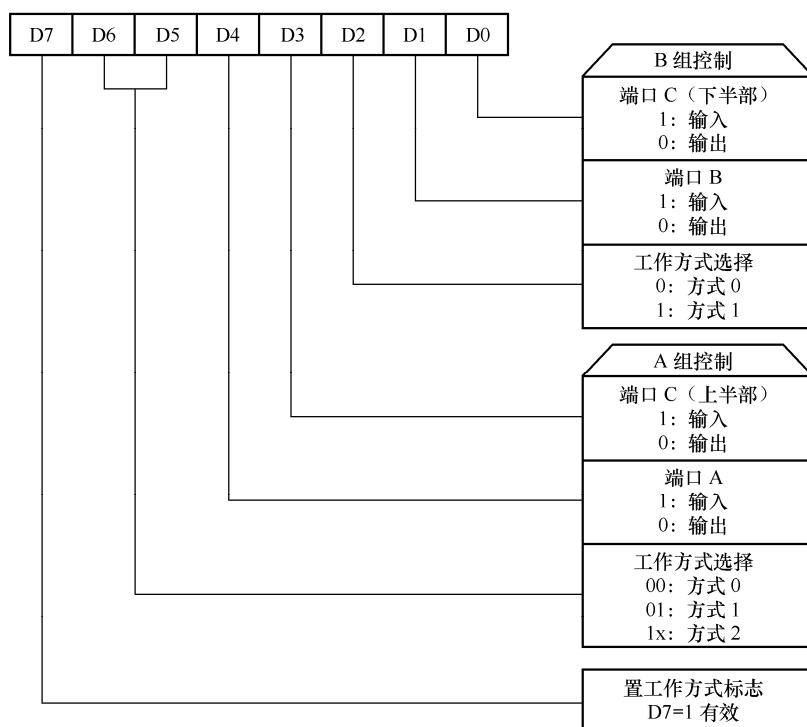


图 5-15 8255A 工作方式控制字格式

- (1) A 口有 3 种工作方式，而 B 口只有 2 种工作方式。
- (2) 在方式 1 或方式 2 下，对 C 口的定义（输入或输出）不影响作为联络线使用的 C 口各位功能。
- (3) 最高位是标志位，其状态固定为“1”，用于表明本字节是方式控制字。

2. C 口位置位/复位控制字

在一些应用情况下，C 口用来定义控制信号和状态信号，因此，C 口的每一位都可以进行置位或复位。对 C 口各位的置位或复位是由位置位/复位控制字进行的。8255A 的位置位/复位控制字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
----	----	----	----	----	----	----	----

D0: 表示对 C 口位的操作，1: 置位，0: 清零。

D1~D3: 位选择，选择要操作的位，如下所示。

操作位 位选择	0	1	2	3	4	5	6	7
D1	0	1	0	1	0	1	0	1
D2	0	0	1	1	0	0	1	1
D3	0	0	0	0	1	1	1	1

D4~D6: 任意。

D7: 对 C 进行位控制有效标志, D7=0 有效。

3. 初始化编程

8255A 初始化的内容就是向控制字寄存器写入工作方式控制字和 C 口位置位/复位控制字, 由于这两个控制字标志位的状态不同, 8255A 自身能加以区别。因此, 两个控制字可按同一地址写入且不受先后顺序限制。

例如, 对 8255A 各口作如下设置: A 口方式 0 输入, B 口方式 1 输出, C 口高位部分为输出, 低位部分为输入。设控制寄存器地址为 003AH。按各口的设置要求, 工作方式控制字为 10010101B, 即 95H。初始化程序段如下所示:

```
MOV    R0,    # 3AH
MOV    A,      # 95H
MOVX   @R0,    A
```

5.4.4 8255A 与 C51 单片机的接口

8255A 与 C51 单片机的接口如图 5-16 所示, 8255A 与 C51 的连接主要包括:

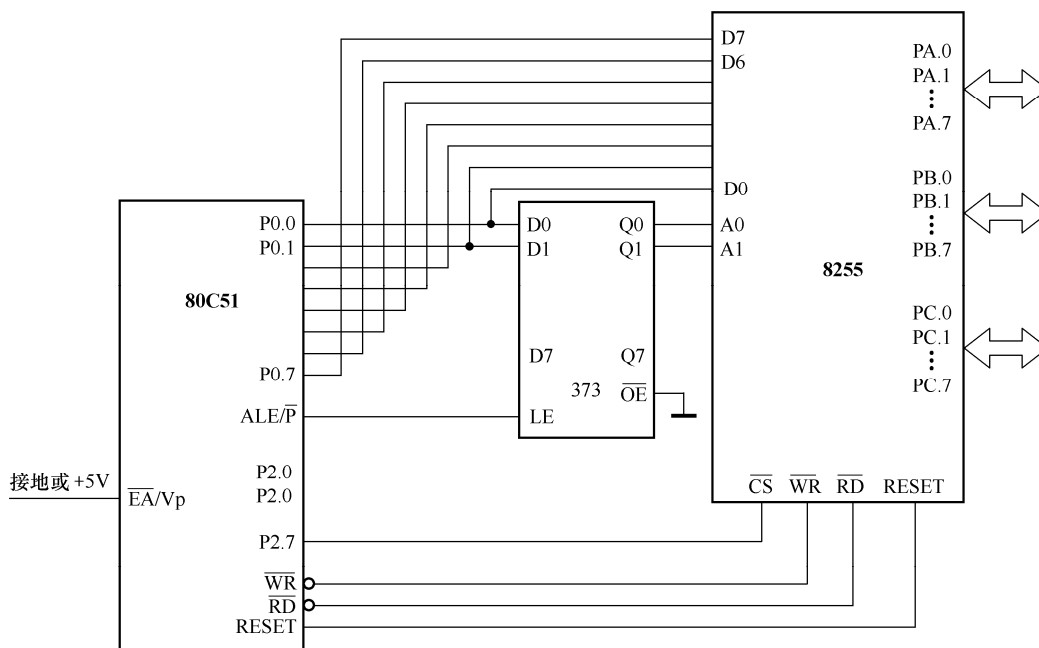


图 5-16 C51 与 8255A 的连接

- (1) 数据总线 D0~D7: 它们分别与 80C51 的 P0.0~P0.7 相连;
- (2) 地址线 A0、A1: 分别与地址锁存器 74LS373 的输出线 Q0、Q1 相连;
- (3) 片选线 CS: 与 P2 口的一条未使用的地址线相连。若为 P2.7, 则 8255A 各口及命令字寄存器的地址如下所示:

PA 口地址为 7FFCH;

PB 口地址为 7FFDH;

PC 口地址为 7FFE H;

控制口地址为 7FFF H。

8255A 与 C51 单片机的接口程序首先定义 8255A 各口和命令字寄存器的地址, 即:

```
#include<absacc.h>
```

```
#define COM8255 XBYTE[0x7FFF]
```

```
#define PA8255 XBYTE[0x7FFC]
```

```
#define PB8255 XBYTE[0x7FFD]
```

```
#define PC8255 XBYTE[0x7FFE]
```

然后在主函数的初始化部分, 对 8255A 的工作方式进行定义:

```
COM8255=0xXX; /*XX 为 8255A 方式字*/
```

这样就可以用 PA8255、PB8255 或 PC8255 对 8255A 的各 I/O 口进行操作了。

5.5 可编程外围定时器 8253

Intel 8253 是可编程定时/计数器, 片内包含有三个独立的通道, 每个通道均为 16 位的计数器, 其计数速率均可达 2.6MHz。

5.5.1 8253 的结构和引脚

8253 的结构框图如图 5-17 所示。

8253 具有三个功能相同的 16 位递减计数器, 每个计数器的工作方式和计数长度分别由软件编程来选择, 8253 可以直接和 C51 接口。

8253 的引脚图如图 5-18 所示, 下面给出具体的引脚定义。

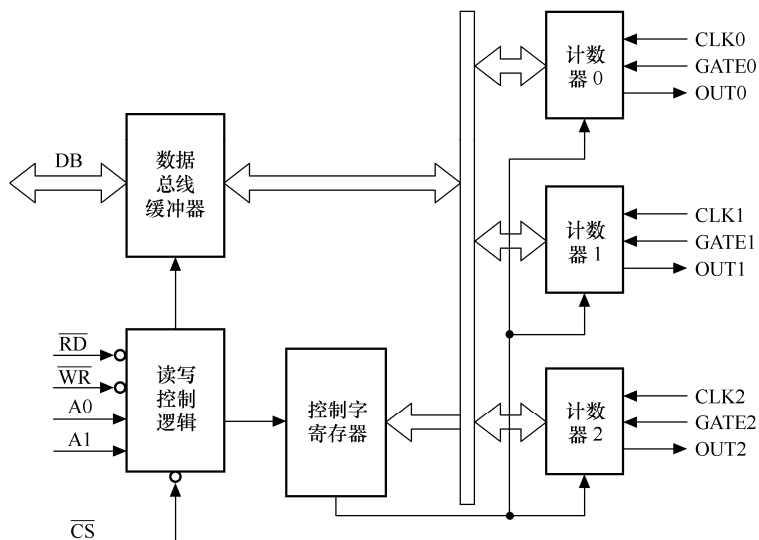


图 5-17 8253 结构框图

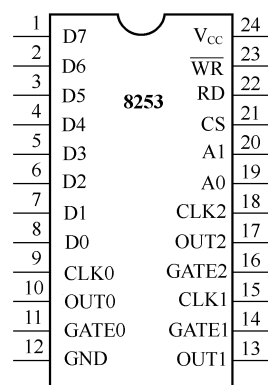


图 5-18 8253 引脚配置

- (1) D7~D0: 双向三态数据总线。
 (2) $\overline{CS}$: 片选信号线。
 (3) $\overline{RD}$ 、 $\overline{WR}$: 读写信号线。
 (4) A1, A0: 地址, 其组合如下所示。

A1	A0	选 择
0	0	计数器 0
0	1	计数器 1
1	0	计数器 2
1	1	控制寄存器

- (5) CLK: 时钟输入线, 这是计数脉冲输入端。
 (6) OUT: 计数器输出信号线, 当计数器减为零时输出相应的信号。
 (7) GATE: 门控信号, 用于启动或禁止计数器的工作。

5.5.2 8253 的工作方式和控制字

8253 计数器工作方式由编程设定, 通过将控制字写入控制寄存器, 来选择每一个计数器的工作方式。控制字的格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RL1	RL0	M2	M1	M0	BCD

- (1) SC1、SC0: 选择计数器, 选择原则如下所示。

SC1	SC0	选 择
0	0	计数器 0
0	1	计数器 1
1	0	计数器 2
1	1	非法

(2) RL1、RL0: 用来设定对计数器的读/写顺序, 计数器的闩锁操作用于计数过程中的读出, 控制原则如下所示。

RL1	RL0	选 择
0	0	计数闩锁操作
0	1	只读/写高字节
1	0	只读/写低字节
1	1	先读/写低字节, 先读/写高字节

(3) BCD: 用来确定计数方式。BCD=0 按二进制计数; BCD=1 按 BCD 码计数。

(4) M2、M1、M0: 用来设定计数器的工作方式。8253 的计数器共有六种工作方式, 如下所示。

M2	M1	M0	工 作 方 式
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5

下面给出各种工作方式的定义。

(1) 方式 0: 这种方式在计数器减为 0 时, 输出线 OUT 升为高电平, 向 CPU 发出中断请求。方式控制字写入后, 输出线 OUT 为低电平, 计数器初值写入后计数器开始计数, 计数期间仍为低电平。

(2) 方式 1: 方式 1 输出单拍负脉冲信号, 脉冲宽度可编程设定。在设定工作方式和写入计数值后, OUT 输出高电平。在门控信号 GATE 上升为高电平时, OUT 输出低电平, 并开始计数。在计数器减为 0 时, 输出变为高电平。

(3) 方式 2: 方式 2 为脉冲发生器方式, 产生连续的负脉冲信号。OUT 输出的负脉冲宽度等于一个时钟周期, 脉冲周期等于写入计数器的计数值和时钟周期的乘积。OUT 受门控信号 GATE 控制。

(4) 方式 3: 方式 3 计数时, 计数器输出为方波。若计数值 N 为偶数, 在前 N/2 计数期间 OUT 输出高电平, 后 N/2 计数期间 OUT 输出低电平。如果 N 为奇数, 高低电平为 (N+1)/2 和 (N-1)/2。其余特性同方式 2。

(5) 方式 4: 方式 4 为软件触发选通方式。方式控制字写入 8253 后, 计数器输出高电平, 再写入计数值之后开始计数。当计数到 0 时输出一个时钟周期的负脉冲。当门控 GATE 输入低电平时, 计数停止。

(6) 方式 5: 方式 5 为硬件触发选通方式。写入方式控制字和计数值后, 输出保持高电平, 只有在门控信号 GATE 上升沿之后才开始计数, 计完最后一个数, 输出一个时钟周期的负脉冲。

5.5.3 8253 与 C51 单片机的接口

8253 与 8031 的接口方法很简单, 其接口电路如图 5-19 所示。

图中, 8253 计数器 0、1、2 的地址为 8000H、8001H、8002H, 控制口地址为 8003H。

【例 5-1】测三个通道脉冲信号的计数率。

采用 8253 可同时测三个通道脉冲信号, 程序 c8253.c 如下所示:

```
#include <absacc.h>
#include <reg51.h>
#define uchar unsigned char
#define uint unsigned int
```

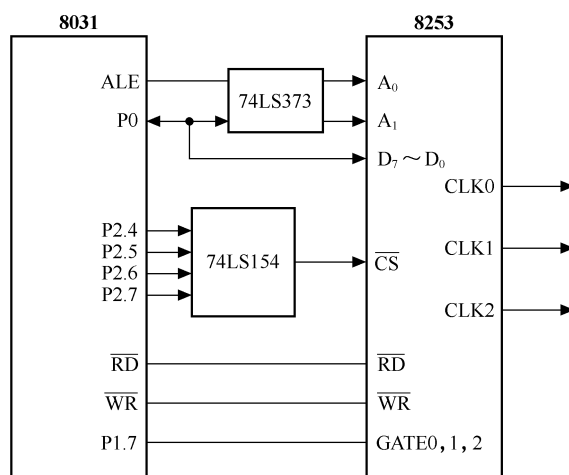


图 5-19 8253 与 8031 的接口电路

```

#define COM8253 XBYTE[0x8003]
#define C1 XBYTE[0x8000]
#define C2 XBYTE[0x8001]
#define C3 XBYTE[0x8002]
#define SNUM 1
sbit P1_7=P1^7; /*门控端*/
uchar it=20;
uchar sn=SNUM;
uint idata BUF8253[3]; /*计数率缓冲区*/
void usto(void) /*使用定时器 0 函数*/
{
    TMOD=TMOD&0xf0+0x01; /*定时器为方式 1*/
    TH0=0x4c; TL0=0x81; /*在 11.0592 MHz 下的 50 ms 定时初值*/
    TR0=1; ET0=1; EA=1; /*启动定时器开中断*/
    P1_7=1; /*开门控*/
}

void toi(void) interrupt 1 /*定时器 0 中断服务*/
{
    TH0=0x4c; TL0=0x81;
    if(--it==0)
    {
        it=20; /*1 秒需 20 次中断*/
        sn--; /*秒减 1*/
    }
}

```

```

}

void cbfp(po, x)                /*计数器取值函数，参数为地址和通道*/
uchar xdata *po;
uchar x;
{
    uchar h, l;
    l=*po; po++; po--;
    h=*po;
    BUF8253[x]=-(h*256+1);      /*因 8253 减计数，应取其负值*/
}

void fcbf(void)                 /*三个通道的取值函数*/
{
    P1_7=0;
    COM8253=0x0a;
    COM8253=0x4a;
    COM8253=0x8a;
    COM8253=0x0a;
    cbfp(&C1, 0);
    COM8253=0x4a;
    cbfp(&C2, 1);
    COM8253=0x8a;
    cbfp(&C3, 2);
}

void init8253(void)             /*初值化 8253 函数*/
(
    COM8253=0x3a;               /*选方式 5*/
    C1=0; C1=0;                 /*设计数初值*/
    COM8253=0x7a;
    C2=0; C2=0;
    COM8253=0xba;
    C3=0; C3=0;
}

void main(void)
{
    init8253();

```

```
    usto();
    while(sn);           /*取 1s 定时到*/
    fcbf();
}
```

5.6 可编程并行接口芯片 8155

5.6.1 8155 的引脚功能和逻辑结构

8155 芯片为 40 引脚双列直插封装，单一的+5V 电源，其引脚排列如图 5-20 所示，其逻辑结构框图如图 5-21 所示。

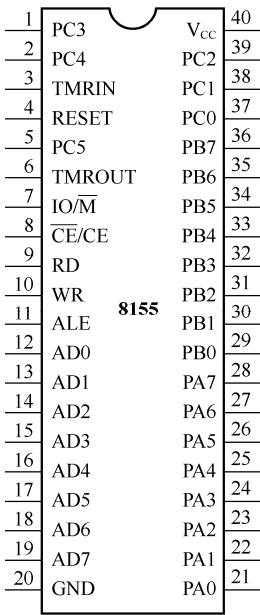


图 5-20 8155 引脚

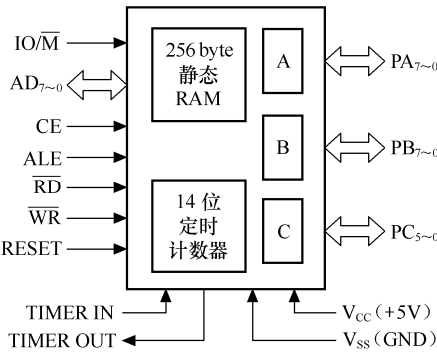


图 5-21 8155 内部逻辑结构

8155 的性能特点：

- (1) 256byte 的 RAM；
- (2) 两个可编程 8 位输入输出口，一个可编程 6 位输入输出口；
- (3) 一个 14 位可编程定时/计数器。

8155 的引脚功能如表 5-7 所示。

表 5-7 8155 芯片的引脚功能

名 称	引 脚	功 能
RESET	4	复位信号输入端，8155 以 600ns 的正脉冲进行复位，复位后 A、B、C 口均置为输入
AD7~AD0	12~19	地址数据复用线

续表

名 称	引 脚	功 能
PA7~PA0	21~28	8 位的通用 I/O 接口, 输入/输出的流向可由程序控制
PB7~PB0	29~36	8 位的通用 I/O 接口, 输入/输出的流向可由程序控制
PC5~PC0		这 6 条引脚可为 6 位的通用 I/O 接口, 也可以用作 PA、PB 的控制信号, 这些可以通过程序控制
PC0	37	A INTR (A 口中断申请线)
PC1	38	A BF (A 口缓冲器满)
PC2	39	A STB (A 口选通)
PC3	1	B INTR (B 口中断申请线)
PC4	2	B BF (B 口缓冲器满)
PC5	5	B STB (B 口选通)
$\overline{CE}$	8	片选信号
$\overline{RD}$	9	读选通信号
$\overline{WR}$	10	写选通信号
ALE	11	地址锁存信号。除进行 AD7~AD0 的地址锁存控制外, 还用于把片选信号 $\overline{CE}$ 和 $\overline{IO/M}$ 等信号进行锁存
$\overline{IO/M}$	7	I/O 与 RAM 选择信号。8155 内部的 I/O 口与 RAM 是分开编址的, 因此要使用控制信号进行区分。 $\overline{IO/M}=0$, 对 RAM 进行读写; $\overline{IO/M}=1$ 对 I/O 口进行读写。该信号是一个特殊信号, 对它的使用要格外注意
TIMER IN	3	定时器输入端
V_{SS}	20	接地
V_{CC}	40	+5V 电源

5.6.2 8155 的工作原理

1. 一般说明

8155 内部具有 256 字节 RAM, 最快存取时间为 400ns。三个 IO 通道, 其中 C 口可编程用以决定另外两个端口 A 和 B 的信息交换方式。8155 还包含一个 14 位定时/计数器。

2. RAM

当 CPU 对 8155 中的 RAM 进行存取时, 应置低 $\overline{IO/M}$ 引脚, 在 A0~A7 引脚上给出单元地址信息。

3. 定时/计数器

在 TMRIN 引脚加入时钟脉冲后, 在 TMROUT 引脚上根据设置的方式可输出四种波形。

4. I/O 口

8155 有 A、B、C 三个输入输出口。A 口和 B 口可以工作于基本输入输出方式或选通输入输出方式；C 口可以工作于基本输入输出方式或选通控制方式（此时 A 口或 B 口为选通输入输出方式）。

5.6.3 8155 的命令/状态字

1. 8155 的命令字

8155 的三组 I/O 口以及定时/计数器有几种工作方式，在使用之前必须设置工作方式。C51 通过 8155 的命令字寄存器内容来设置工作方式，此后 8155 就按设定的方式动作。8155 命令字格式如下所示：

D7	D6	D5	D4	D3	D2	D1	D0
TM2	TM1	IEB	IEA	PC2	PC1	PB	PA

下面给出 8155 命令字相应位的具体定义。

（1）PA：A 口功能，0 表示输入；1 表示输出。

（2）PB：B 口功能，0 表示输入；1 表示输出。

（3）PC2PC1：00 对应方式 1；01 对应方式 2；10 对应方式 3；11 对应方式 4。各种工作方式下，PC 口对应的功能如下所示。

方式	1	2	3	4
PC0	输入口	输出口	A 口 INTR	A 口 INTR
PC1	输入口	输出口	A 口 BF	A 口 BF
PC2	输入口	输出口	A 口 STB	A 口 STB
PC3	输入口	输出口	输出口	B 口 INTR
PC4	输入口	输出口	输出口	B 口 BF
PC5	输入口	输出口	输出口	B 口 STB

（4）IEA：0 表示禁止 A 口中断；1 表示允许 A 口中断。

（5）IEB：0 表示禁止 B 口中断；1 表示允许 B 口中断。

（6）TM2 和 TM1：决定定时器的四种工作方式。

T0~T13 用于控制输出定时，由程序设置计数器的计数长度，14 位长度需两次写入。对定时器四种工作方式的说明如下所示。

（1）方式 1：启动定时器后，定时器只计数一次。在给定的计数长度的前半部分，定时器输出高电平；计数的后半部分输出低电平。当计数值减到零时，定时器输出高电平。

（2）方式 2：启动定时器后，定时器开始连续计数，当计数值减到零时，系统自动重新装入计数初值。在一个计数周期（计数长度）内，前半部分输出高电平，后半部分输出低电平，因此形成周期矩形波。

（3）方式 3：启动定时器后，定时器只计数一次。当计数值减到零时，定时器输出一个低脉冲，宽度为定时器输入时钟周期。

(4) 方式 4: 启动定时器后, 定时器开始连续计数, 当计数值减到零时, 系统自动重新装入计数初值, 并输出一个低脉冲。

8155 定时器对 TMRIN 引脚输入的外部事件的计数脉冲或定时脉冲进行计数, 减法计数器减至零时, TMROUT 引脚输出定时到矩形波或脉冲信号。使用定时器时, 应先将计数长度写入定时器寄存器中, 设置定时器工作方式, 然后再装入命令字并启动定时器。

定时器在计数期间, 可以将新的计数长度值和计数方式写入计数长度寄存器中, 它不影响定时器原来的操作, 只有装入了新的启动控制命令到命令寄存器, 等原来的计数值到零后才按新的工作方式及计数长度进行工作。

2. 8155 的状态寄存器

8155 的状态寄存器用来记录 8155 的当前状态。其格式为:

D7	D6	D5	D4	D3	D2	D1	D0
X	TIMER	INTEB	BFE	INTRB	INTEA	BFA	INTRA

(1) INTR(A): A 口中断请求;

(2) BF(A): A 口缓冲器满/空;

(3) INTE(A): A 口中断允许;

(4) X: 保留。

8155 的命令寄存器只能写入, 不能读出。状态寄存器只能读出, 不能写入。当对 8155 的 RAM 进行存取时, 置 $\text{IO}/\overline{\text{M}}$ 为 0, AD0~AD7 8 位地址即可对所有 RAM 单元进行寻址。

当对 8155 的 I/O 端口、命令字状态字寄存器或定时/计数器进行操作时, 则要区分所操作的对象。首先, 置 $\text{IO}/\overline{\text{M}}$ 为 1, 命令字寄存器、状态字寄存器、计数器高低字节、A 口、B 口和 C 口的地址编码如表 5-8 所示。这时 8155 的片选引脚必须为低电平。

表 5-8 8155 地址编码

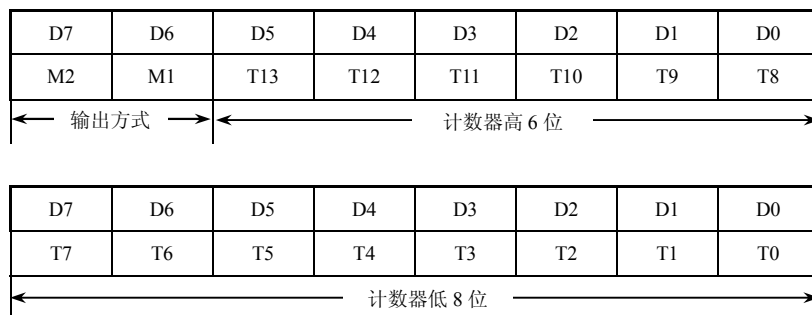
A7	A6	A5	A4	A3	A2	A1	A0	选中的寄存器
×	×	×	×	×	0	0	0	命令/状态寄存器 (写/读)
×	×	×	×	×	0	0	1	A 口
×	×	×	×	×	0	1	0	B 口
×	×	×	×	×	0	1	1	C 口
×	×	×	×	×	1	0	0	定时器低 8 位寄存器
×	×	×	×	×	1	0	1	定时器高 6 位寄存器

注: ×表示任意值, 一般取 1。

5.6.4 8155 的定时/计数器

1. 定时/计数器的计数结构

8155 的定时/计数器是一个 14 位的减法计数器, 它由两个 8 位寄存器构成, 以其中的低 14 位组成计数器, 剩下的两个高位 (M2, M1) 用于定义计数器输出的信号形式。8155 定时/计数器的计数结构如下所示。



2. 定时/计数器的使用

(1) 8155 的定时/计数器是减法计数，而 C51 的定时/计数器却是加法计数器，因此确定计数初值的方法是不同的。

(2) C51 的定时/计数器有多种工作方式，而 8155 的定时器/计数器则只有一种固定的工作方式，即 14 位计数，并通过软件方法进行计数值加载。

(3) C51 的定时/计数器存两种计数脉冲。当定时工作时，由芯片内部按机器周期提供固定频率的计数脉冲；当计数工作时，从芯片外部引入计数脉冲。8155 的定时/计数器，不论是定时工作还是计数工作，都由外部提供计数脉冲，其信号引脚就是 TMER IN。

(4) C51 的定时/计数器，计数溢出时自动置位 TCON 寄存器的计数溢出标志位 (TF)，供用户以查询或中断方式使用；但 8155 的定时/计数器，计数溢出时向芯片外边输出一个信号 (TMROUT)，而且这一信号还有脉冲和方波两种形式，供用户进行选择。具体由 M2、M1 两位定义：M2M1=00 对应单个方波；M2M1=01 对应连续方波；M2M1=10 对应单个脉冲；M2M1=11 对应连续脉冲。

3. 定时/计数器的控制

8155 的定时/计数器的工作方式由命令字中的高两位 D7D6 进行控制，具体说明如下：

- (1) D7D6=00，不影响计数器工作；
- (2) D7D6=01，停止计数，如计数器未启动则无操作，如计数器正运行则停止计数；
- (3) D7D6=10，达到计数值（计数器减为 0）后停止；
- (4) D7D6=11，启动，如计数器没运行，则在装入计数值后开始计数；如计数器已运行，则在当前计数值计满后，再以新的计数值进行计数。

4. 8155 定时/计数器的初始化

要求使用 8155 定时/计数器对计数脉冲进行千分频，即计数 1000 后，TMROUT 端电平状态变化，并重新置数以产生连续方波。此外假定 PA 口为输入方式，PB 口为输出方式，PC 口为输入方式，禁止中断。请编写初始化程序。

解此题共两项任务：计数初值的确定和命令字的确定。由于要求输出连续方波，因此计数器的最高两位 M2M1=01。计数器的其他 14 位装入计数初值。由于 8155 计数器是减法计数，所以计数初值应为十进制数 1000，十六进制数为 03E8H，则计数器高位字节为 43H，计数器低位字节为 0E8H。

按上述要求，8155 的命令字为 0C2H。各位状态如下所示：

计 数 器		B 口	A 口	C 口		B 口	A 口
装入后启动		不允许中断		输入		输出	输入
D7	D6	D5	D4	D3	D2	D1	D0
1	1	0	0	0	0	1	0

由于命令字的高两位 D7D6=11，因此在装入计数值后计数器即开始计数。假定命令/状态寄存器的地址为 0FD00H，初始化程序为：

```

MOV     DPTR,      # 0FD00H      ; 命令/状态寄存器地址
MOV     A,          # 0C2H       ; 命令字
MOVX    @DPTR,      A           ; 装入命令字
MOV     DPTR,      # 0FD04H      ; 计数器低 8 位地址
MOV     A,          # 0E8H       ; 低 8 位计数值
MOVX    @DPTR,      A           ; 写入计数值低 8 位
INC     DPTR
MOV     A,          # 43H        ; 高 8 位计数值
MOVX    @DPTR,      A           ; 写入计数值高 8 位

```

5.6.5 8155 的输入/输出

1. 基本输入/输出方式

以 A 口为例，当预先设定其为基本输入方式后，CPU 就可以用输入指令（MOVX A, @DPTR）从 PA0~PA7 读取 8 位数据。CPU 向 8155 发出的控制信号有 RD=0、IO/ $\overline{M}$ =1 以及 A 口的地址编码。若预先设定 A 口为基本输出方式，CPU 即可用输出指令（MOVX @DPTR, A）向 A 口输出 8 位数据。这时 CPU 要发出的信号是 WR=0、IO/ $\overline{M}$ =1、A 口的地址编码以及要输出的数据。

2. 选通输入方式

若 A 口设置为选通输入方式，则相应地 C 口 PC0、PC1、PC2 就作为 A 口的选通控制引线。此时，PC0 作为 INTR（中断）、PC1 作为 BF（缓冲器满）、PC2 作为 $\overline{STB}$ （选通）。这些引线与 C51 和外设的连接如图 5-22 所示。

A 口的动作过程如下：

- （1）外部设备可提供数据时，就把数据送至 PA0~PA7 引脚，然后发出 $\overline{STB}$ 信号；
- （2）8155 接收到数据后，就将 BF 置高，表示 A 口缓冲器已满，同时，INTR 引脚向 CPU 发出中断请求；
- （3）CPU 响应中断请求，读取 8155A 口数据；
- （4）8155 在数据被取走后，就清零 BF 引脚，恢复到（1）之前的状态。

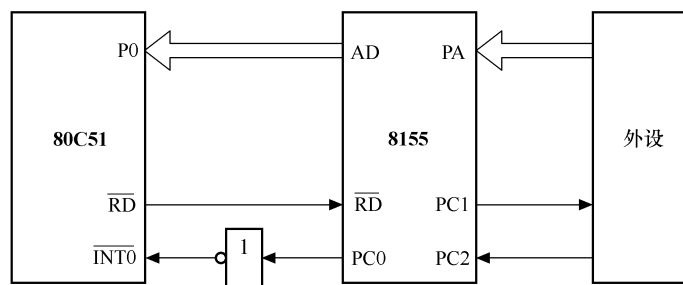


图 5-22 8155 A 口选通输入方式

3. 选通输出方式:

A 工作于选通输出方式时, C 口的 PC0、PC1、PC2 的作用与选通输入方式时相同, 这些引线 with C51 和外设的连接如图 5-23 所示。

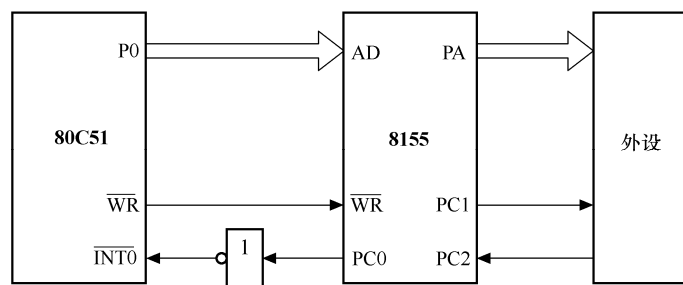


图 5-23 8155 A 口选通输出方式

其工作过程如下所示:

- (1) 当 8155 可接收数据时, 就置高 INTR 引线, 向 CPU 发出中断请求;
- (2) CPU 响应中断, 向 8155 A 口写数据;
- (3) 8155 接收数据后, 就向外设发出 BF 信号;
- (4) 外设若可以接收数据, 就从 8155 读取数据, 并发出 $\overline{STB}$ 信号;
- (5) 8155 在 $\overline{STB}$ 的后沿 (上升沿) 再将 INTR 置高, 即回到 (1) 的状态。

5.6.6 8155 与 C51 单片机的接口

8155 可直接与 C51 系列单片机相连, 不需要任何外加逻辑电路, 如图 5-24 所示。

通常选用 P2 口的高位地址作 8155 的片选信号和 $\overline{IO/\overline{M}}$ 选择信号。C51 的 ALE 与 8155 的 ALE 相连, 利用 C51 ALE 的下降沿把 P0 口的地址锁存在 8155 内的锁存器中。P0.0~P0.7 与 AD0~AD7 相连, $\overline{RD}$ 和 $\overline{WR}$ 也有对应相连端口。

按照图 5-24 所示的连接方法, 8155 的地址分配如下:

0111 1111 1111 1000~0111 1111 1111 1101, 即 7FF8H~7FFDH。

对 8155 内部 RAM 的地址分配如下:

0011 1111 0000 0000~0011 1111 1111 1111, 即 3F00H~3FFFH。

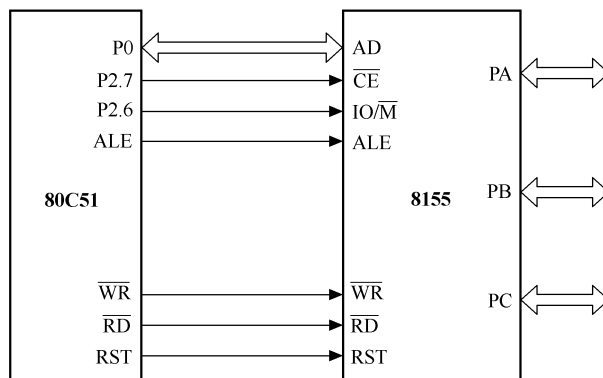


图 5-24 8155 与 C51 单片机的接口电路

对 8155 的操作可描述为以下几个方面：

```
#include <absacc.h>
```

```
#define STATE8155 XBYTE[0x7FF8]
```

```
#define IOA XBYTE[0x7FF9]
```

```
#define IOB XBYTE[0x7FFA]
```

```
define IOC XBYTE[0x7FFB]
```

(1) 写命令字寄存器

```
STATE8155=x;
```

(2) 读状态寄存器

```
x=STATE8155;
```

(3) 写 A 口

```
IOA=x;
```

(4) 读 B 口

```
x=IOB;
```

(5) 写 8155 的 RAM 单元 (20H)

```
XBYTE[0x3F20]=35;
```

(6) 读 8155 的 RAM 单元 (65H)

```
x=XBYTE[0x3F65];
```

5.7 I²C 接口芯片 AT24C01

I²C (Inter-Integrated Circuit) 总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。I²C 总线产生于在 20 世纪 80 年代，最初为音频和视频设备开发，如今主要在服务器管理中使用，其中包括单个组件状态的通信。例如，管理员可对各个组件进行查询，以管理系统的配置或掌握组件的功能状态，如电源和系统风扇；可随时监控内存、硬盘、网络、系统温度等多个参数，增加了系统的安全性，便于管理。

5.7.1 I²C 总线特点

I²C 总线最主要的优点是其简单性和有效性。由于接口直接在组件之上，因此 I²C 总线占用的空间非常小，减少了电路板的空间和芯片引脚的数量，降低了互联成本。总线的长度可高达 25 英尺，并且能够以 10kbit/s 的最大传输速率支持 40 个组件。I²C 总线的另一个优点是它支持多主控（multimastering），其中任何能够进行发送和接收的设备都可以成为主总线。一个主控能够控制信号的传输和时钟频率。当然，在任何时间点上只能有一个主控。

5.7.2 I²C 总线工作原理

1. 总线的构成及信号类型

I²C 总线是一种串行数据总线，只有二根信号线，一根是双向的数据线 SDA，另一根是时钟线 SCL。在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送，最高传送速率 100kbps。各种被控制电路均并联在这条总线上，但就像电话机一样只有拨通各自的号码才能工作，所以每个电路和模块都有唯一的地址，在信息的传输过程中，I²C 总线上并接的每一模块电路既是主控器（或被控器），又是发送器（或接收器），这取决于它所完成的功能。CPU 发出的控制信号分为地址码和控制量两部分，地址码用来选址，即接通需要控制的电路，确定控制的种类；控制量决定该调整的类别（如对比度、亮度等）及需要调整的量。这样，各控制电路虽然挂在同一条总线上，却彼此独立，互不相关。

2. 位的传输

SDA 线上的数据必须在时钟的高电平周期保持稳定，数据线的高或低电平状态只有在 SCL 线的时钟信号是低电平时才能改变，如图 5-25 所示。

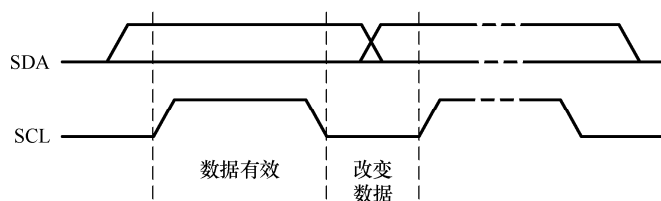


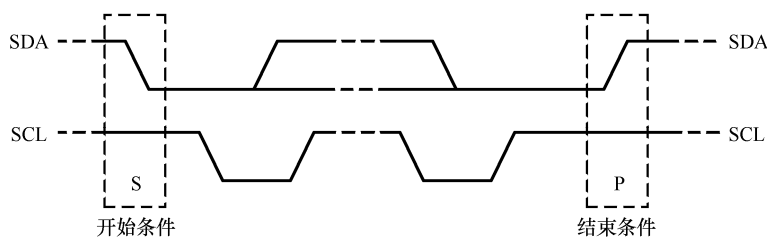
图 5-25 I²C 位传输数据改变时序图

3. 开始信号

SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。

4. 结束信号

SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。I²C 位传输时序如图 5-26 所示。

图 5-26 I²C 位传输时序图

5. 应答信号

接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，则判断受控单元出现故障。

6. 总线基本操作

I²C 规程运用主/从双向通信。器件发送数据到总线上，则定义为发送器，器件接收数据则定义为接收器。主器件和从器件（本书为 AT24C01）都可以工作于接收和发送状态。总线必须由主器件（通常为微控制器 CPU）控制，主器件产生串行时钟（SCL）控制总线的传输方向，并产生起始和停止条件。SDA 线上的数据状态仅在 SCL 为低电平期间才能改变，SCL 为高电平期间，SDA 状态的改变被用来表示起始和停止条件，参见图 5-26 所示的时序。

5.7.3 AT24C01 与 C51 的接口

AT24C 系列串行 E²PROM 具有 I²C 总线接口功能，功耗小，宽电源电压（根据不同型号 2.5V~6.0V），工作电流约为 3mA，静态电流随电源电压不同为 30μA~110μA。

1. AT24C 系列串行 E²PROM 引脚结构

AT24C 系列串行 E²PROM 引脚结构如图 5-27 所示。

其中：A0, A1, A2 为芯片地址线，单片使用时一般接 V_{ss}；SCL 为串行移位时钟；SDA 为串行数据或地址，通过 SDA，CPU 可对芯片写入或读出数据；WP 为写保护，若 WP 接 V_{CC}，芯片为只读。

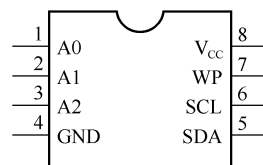


图 5-27 AT24C 系统芯片引脚结构图

2. AT24C 系列 E²PROM 接口及地址选择

由于 I²C 总线可挂接多个串行接口器件，在 I²C 总线中每个器件应有唯一的器件地址。按 I²C 总线规则，器件地址为 7 位数据（即一个 I²C 总线系统中理论上可挂接 128 个不同地址的器件），它和 1 位数据方向位构成一个器件寻址字节，最低位 D0 为方向位（读/写）。器件寻址字节中的最高 4 位（D7~D4）为器件型号地址，不同的 I²C 总线接口器件的型号地址是厂家给定的，如 AT24C 系列 E²PROM 的型号地址皆为 1010，器件地址中的低 3 位为引脚

地址 A2、A1、A0，对应器件寻址字节中的 D3、D2、D1 位，在硬件设计时由连接的引脚电平给定，AT24C 系列地址分配情况如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
1	0	1	0	A2	A1	A0	R/W

对于 E²PROM 的片内地址，容量小于 256 字节的芯片(AT24C01/02)，8 位片内寻址(A0~A7)即可满足要求。然而对于容量大于 256 字节的芯片，则 8 位片内寻址范围不够，如 AT24C16，相应的寻址位数应为 11 位 ($2^{11}=2048$)。若以 256 字节为 1 页，则多于 8 位的寻址视为页面寻址。在 AT24C 系列中对页面寻址位采取占用器件引脚地址(A2、A1、A0)的办法，如 AT24C16 将 A2、A1、A0 作为页地址。凡在系统中引脚地址用作页地址后，该引脚在电路中不得使用，作悬空处理。AT24C 系列串行 E²PROM 的器件地址寻址字节如图 5-28 所示，图中 P0、P1、P2 表示页面寻址位。

		D7	D6	D5	D4	D3	D2	D1	D0
AT24C01/02	1K/2K	1	0	1	0	A2	A1	A0	R/W
AT24C04	4K	1	0	1	0	A2	A1	P0	R/W
AT24C08	8K	1	0	1	0	A2	P1	P0	R/W
AT24C16	16K	1	0	1	0	P2	P1	P0	R/W

图 5-28 AT24C 系列串行 E²PROM 的器件地址寻址字节

3. C51 单片机与 AT24C01 E²PROM 通信的硬件实现

图 5-29 是用 C51 的 P2 口模拟 I²C 总线与 E²PROM 通信的连接电路图(以 AT24C01 为例)，由于 AT24C01 是漏极开路，图中 R1、R2 为上拉电阻(5.1k Ω)。A0~A2 和 Vss 地址引脚均接地。串行时钟 SCL 接 P2.1，串行数据或地址线 SDA 接 P2.0 引脚。其芯片的写地址为 A0H，读地址为 A1H。

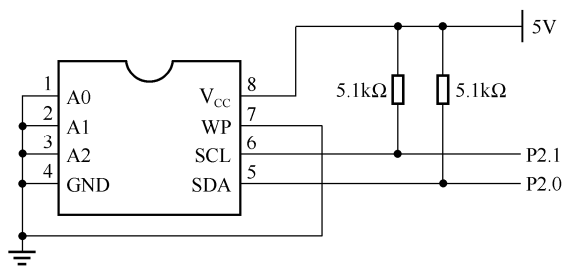


图 5-29 AT24C01 与 C51 单片机接口

4. AT24C 系列 E²PROM 读写操作

对 AT24C 系列 E²PROM 的读写操作完全遵守 I²C 总线的主收从发和主发从收的规则。

单片机对 AT24C 系列 E²PROM 的读写，在串行线 SDA 上遵循以下的数据传送约定步骤。

(1) 单片机发送起始信号，占据串行总线，随后发送 7 位芯片地址和 1 位数据传送方向位 R/W，构成寻址字节，此处为 A0H。

(2) E²PROM 接收到单片机发送的芯片寻址字节后返回给单片机一个响应位（低电平）以做好读、写准备。

(3) 单片机在收到 E²PROM 片内读写单元，随后 E²PROM 返回单片机一个响应位，至此，E²PROM 的读、写准备工作皆已完成。

(4) 单片机对 E²PROM 的读、写方向已在芯片寻址字节中规定好了，若是写，则在 (3) 步骤后由单片机向 E²PROM 发送所写数据字节，每发送一字节数据，片内地址自动加 1，若连续发送数据超过 8 字节，则需换页，每次换页间隔时间至少为 10ms（芯片写入周期）；若是读，则在 (3) 后由单片机接收 E²PROM 发送的指定单元数据。数据读、写终止由单片机发送停止信号而终止。

写操作分为字节写和页面写两种操作，对于页面写，根据芯片一次装载的字节不同有所不同。连续写操作是对 E²PROM 连续装载 n 个字节数据的写入操作，n 随型号不同而不同，一次可装载字节数也不同。例如，AT24C01/02 为 8 字节/每页；AT24C04/08/16 为 16 字节/每页。

【例 5-2】C51 模拟 I²C 总线程序设计。

由于 C51 没有 I²C 接口，与 I²C 总线通信就要用软件模拟 I²C 时序。程序如下所示：

```
#include <reg51.h>
#define WriteDeviceAddress 0xa0
#define ReadDviceAddress 0xa1
sbit SCL=P2^1;
sbit SDA=P2^0;
void DelayMs(unsigned int number)    /*延时程序，延时 number ms*/
{
    .....
}

void Start()                          /*读写开始程序*/
{
    SDA=1;
    SCL=1;
    SDA=0;
    SCL=0;
}
void Stop()                           /*读写停止程序*/
{
    SCL=0;
    SDA=0;
    SCL=1;
```

```
    SDA=1;
}

void Ack()                                /*读一字节响应程序*/
{
    SDA=0;
    SCL=1;
    SCL=0;
    SDA=1;
}
void NoAck()                              /*读结束响应程序*/
{
    SDA=1;
    SCL=1;
    SCL=0;
}
bit TestAck()                             /*写结束响应程序*/
{
    bit ErrorBit;
    SDA=1;
    SCL=1;
    ErrorBit=SDA;
    SCL=0;
    return(ErrorBit);
}
bit Write8Bit(unsigned char input)        /*写一字节程序*/
{
    unsigned char temp;
    for(temp=8;temp!=0;temp--)
    {
        SDA=(bit)(input&0x80);
        SCL=1;
        SCL=0;
        input=input<<1;
    }
}
void Write (unsigned char *Wdata, unsigned char RomAddress, unsigned char number)
{
    /*写程序*/
    Start();
```

```

    Write8Bit(WriteDeviceAddress);
    TestAck();
    Write8Bit(RomAddress);
    TestAck();
    for(;number!=0;number--)
    {
        Write8Bit(*Wdata);
        TestAck();
        Wdata++;
    }
    Stop();
    DelayMs(10);
}

unsigned char Read8Bit()                /*读一字节程序*/
{
    unsigned char temp, rbyte=0;
    for(temp=8;temp!=0;temp--)
    {
        SCL=1;
        rbyte=rbyte<<1;
        rbyte=rbyte|((unsigned char)(SDA));
        SCL=0;
    }
    return(rbyte);
}

void Read (unsigned char *RamAddress, unsigned char RomAddress, unsigned char bytes)
{
    /*读程序*/
    unsigned char temp, rbyte;
    Start();
    Write8Bit(WriteDeviceAddress);
    TestAck();
    Write8Bit(RomAddress);
    TestAck();
    Start();
    Write8Bit(ReadDviceAddress);
    TestAck();
    while(bytes!=1)
    {
        *RamAddress=Read8Bit();

```

```

Ack();
RamAddress++;
bytes--;
}
*RamAddress=Read8Bit();
NoAck();
Stop();
}

```

5.8 SPI 接口芯片 X5045

SPI (Serial Peripheral Interface, 串行外设接口) 总线系统是一种同步串行外设接口, 它可以使 MCU 与各种外围设备以串行方式进行通信以交换信息。如外围设置 FLASH RAM、网络控制器、LCD 显示驱动器、A/D 转换器和 MCU 等。SPI 总线系统可直接与各个厂家生产的多种标准外围器件直接接口, 该接口一般使用 4 条线: 串行时钟线 (SCK)、主机输入/从机输出数据线 (MISO)、主机输出/从机输入数据线 (MOSI) 和低电平有效的从机选择线 (SS)。注意, 有的 SPI 接口芯片带有中断信号线 INT 或 $\overline{\text{INT}}$ 、有的 SPI 接口芯片没有主机输出/从机输入数据线 MOSI。由于 SPI 系统总线一共只需 3~4 位数据线和控制即可实现与具有 SPI 总线接口功能的各种 I/O 器件进行接口, 而扩展并行总线则需要 8 根数据线、8~16 位地址线、2~3 位控制线, 因此, 采用 SPI 总线接口可以简化电路设计, 节省很多常规电路中的接口器件和 I/O 口线, 提高设计的可靠性。

5.8.1 SPI 总线的组成

利用 SPI 总线可在软件的控制下构成各种系统。如 1 个主 MCU 和几个从 MCU、几个从 MCU 相互连接构成多主机系统 (分布式系统)、1 个主 MCU 和 1 个或几个从 I/O 设备所构成的各种系统等。在大多数应用场合, 可使用 1 个 MCU 作为控机来控制数据, 并向 1 个或几个从外围器件传送该数据。从器件只有在主机发命令时才能接收或发送数据。其数据的传输格式是高位 (MSB) 在前, 低位 (LSB) 在后。SPI 总线接口系统的典型结构如图 5-30 所示, SPI 总线读写数据读写时序如图 5-31 所示。

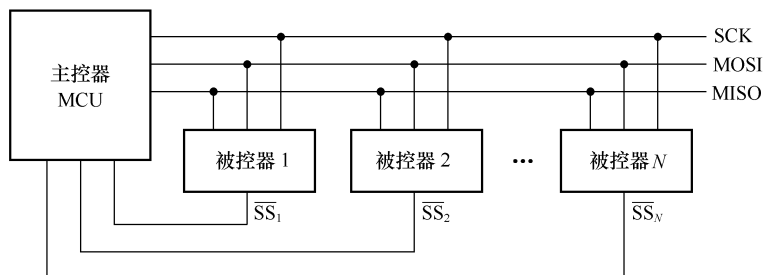


图 5-30 SPI 总线接口系统的典型结构图

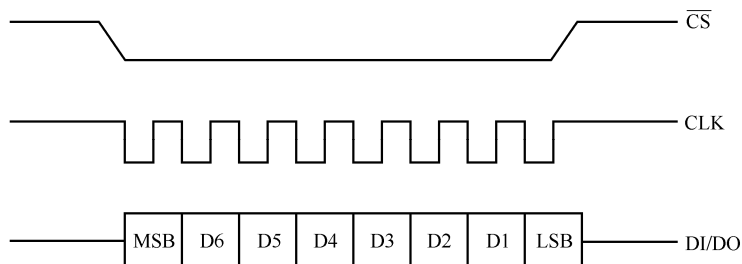


图 5-31 SPI 总线读写数据时序图

在 SPI 与几种不同的串行 I/O 芯片相连时，必须用每片的允许控制端，可用 CPU 的 I/O 端口输出线来实现。此时应特别注意这些串行 I/O 芯片的输入输出特性。

1. 输出芯片的串行数据输出是否有三态控制端。

三态控制端平时未选中时，芯片的输出端应处于高阻态。若没有三态控制端，应外加三态门。否则 CPU 的 MISO 端只能连接一个输入芯片。

2. 输出芯片的串行数据输入是否有允许控制端。

只有在这片芯片允许时，SCK 脉冲才把串行数据移入该芯片；芯片禁止时 SCK 对芯片无影响。若没有允许控制端，应在外部用门电路对 SCK 进行控制后，再加到芯片的时钟输入端，或者 SPI 只连接一个芯片，不能再连接其他输入或输出芯片。

5.8.2 X5045 与 C51 单片机的接口

X5045 是在单片机系统中广泛应用的一种看门狗芯片，它把上电复位、看门狗定时器、电压监控和 E²PROM 四种常用功能组合在单个芯片里，以降低系统成本、节约电路板空间。其看门狗定时器和电源电压监控功能可对系统起到保护作用；512×8 位的 E²PROM 可用来存储单片机系统的重要数据。

1. X5045 芯片的特点与工作原理

(1) 特点

- ① 可编程的看门狗定时器；
- ② 低电压检测和复位信号提供；
- ③ 5 种标准复位端电压；
- ④ 使用特殊编程序列可重复对低 V_{CC} 复位电压的编程；
- ⑤ 低功耗，看门狗开时，最大电流小于 50fA，看门狗关时，最大电流小于 10fA，读数时最大电流小于 2mA，工作电压可以在 118V~316V，217V~515V 或 415V~515V 之间；
- ⑥ 内置 4KB E²PROM，可写入 1 000 000 次；
- ⑦ 使用块保护功能可以保护存入的数据不被意外改写；
- ⑧ 313MHz 时钟速率；

- ⑨ 片内偶然性的写保护，即有写锁存和写保护引脚；
 ⑩ 最小编程时间：16 位页写模式和写周期 5ms（典型）。
 （2）引脚定义及通信协议

X5045 芯片的引脚结构如图 5-32 所示。下面对其引脚功能进行说明。

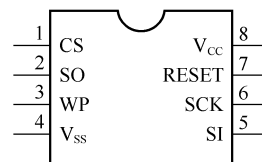


图 5-32 X5045 管脚结构图

SO: 串行数据输出脚，在一个读操作的过程中，数据从 SO 脚移位输出，在时钟的下降沿时数据改变。

SI: 串行数据输入脚，所有的操作码、字节地址和数据从 SI 脚写入，在时钟的上升沿时数据被锁定。

SCK: 串行时钟，控制总线上数据输入和输出的时序。

CS: 芯片使能信号。当其为高电平时，芯片不被选择，SO 脚为高阻态，除非一个内部的写操作正在进行，否则芯片处于待机模式；当引脚为低电平时，芯片处于活动模式，上电后在任何操作之前需要 CS 引脚的一个从高电平到低电平的跳变。

WP: 当 WP 引脚为低电平时，芯片禁止写入，但是其他的功能正常；当 WP 引脚为高电平时，所有的功能都正常。当 CS 为低时，WP 变为低可以中断对芯片的写操作。但是如果内部的写周期已经被初始化后，WP 变低不会对写操作造成影响。

RESET: 复位输出端。

（3）工作原理

X5045 除了作为看门狗芯片使用外，另外一个基本的功能就是作为 E²PROM 数据存储器使用，内部包含 512×8 的串行 E²PROM，以保证系统在掉电后仍可维持重要数据不变。X5045 与 MCU 采用流行的 SPI 总线接口方式，可以和任意一款单片机的 I/O 口直接连接。芯片内部含有一个位指令移位寄存器，该寄存器可以通过 SI 来访问。数据在 SCK 的上升沿由时钟同步输入，在整个工作期内，CS 必须是低电平且 WP 必须是高电平。如果在看门狗定时器预置的溢出时间内没有总线活动（通常指 CS 引脚电平变化），那么 X5045 将提供复位信号输出以保证系统的可靠运行。X5045 内部有一个“写使能”锁存器，在执行写操作之前该锁存器必须被置位，在写周期完成之后，该锁存器自动复位。X5045 还有一个状态寄存器，用来提供 X5045 状态信息以及设置块保护和看门狗的定时周期。对芯片内部寄存器的读写均按一定的指令格式进行，表 5-9 为 X5045 的指令格式。

表 5-9 X5045 的指令寄存器

指令名称	指令格式	操作
WREN	0000 0110	设置写使能锁存器（允许写操作）
WRDI	0000 0100	复位写使能锁存器（禁止写操作）
RSDR	0000 0101	读状态寄存器
WRSR	0000 0001	写状态寄存器
READ	0000 A ₈ 011	从所选地址的存储器阵列读取数据
WRITE	0000 A ₈ 010	把数据写入所选地址的存储器阵列中

数据读写时，MSB（最高位）在前。表 5-9 中的 A₈ 表示内部存储器的高地址位。在实际应用中，往往要对状态寄存器进行读写操作，它是一个 8 位的寄存器，用来标识芯片的忙闲状态、内部 E²PROM 数据块保护范围以及看门狗定时器的定时周期，其内部格式

如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
X	X	WD1	WD0	RL1	RL0	WEL	WIP

2. X5045 芯片与 C51 单片机的硬件连接

对于不带 SPI 串行总线接口的 C51 系列单片机来说, 可以使用软件来模拟 SPI 的操作, 包括串行时钟、数据输入和数据输出。对于不同的串行接口外围芯片, 它们的时钟时序是不同的。对于在 SCK 的上升沿输入 (接收) 数据和在下降沿输出 (发送) 数据的器件, 一般应将其串行时钟输出口 P1.1 的初始状态设置为 1, 而在允许接收后再置 P1.1 为 0。这样, MCU 在输出 1 位 SCK 时钟的同时, 将使接口芯片串行左移, 从而输出 1 位数据至 C51 单片机的 P1.3 口 (模拟 MCU 的 SI 线), 此后再置 P1.1 为 1, 使 C51 系列单片机从 P1.0 (模拟 MCU 的 SO 线) 输出 1 位数据 (先为高位) 至串行接口芯片。至此, 模拟 1 位数据输入输出便宣告完成。此后再置 P1.1 为 0, 模拟下 1 位数据的输入输出……依此循环 8 次, 即可完成 1 次通过 SPI 总线传输 8 位数据的操作。对于在 SCK 的下降沿输入数据和上升沿输出数据的器件, 则应取串行时钟输出的初始状态为 0, 即在接口芯片允许时, 先置 P1.1 为 1, 以便外围接口芯片输出 1 位数据 (MCU 接收 1 位数据), 之后再置时钟为 0, 使外围接口芯片接收 1 位数据 (MCU 发送 1 位数据), 从而完成 1 位数据的传送。

(1) 硬件连接

C51 系列单片机与存储器 X5045 (E²PROM) 的硬件连接图如图 5-33 所示, 图中, P1.0 模拟 MCU 的数据输出端 (SO), P1.1 模拟 SPI 的 SCK 输出端, P1.2 模拟 SPI 的从机选择端, P1.3 模拟 SPI 的数据输入端 (SI)。

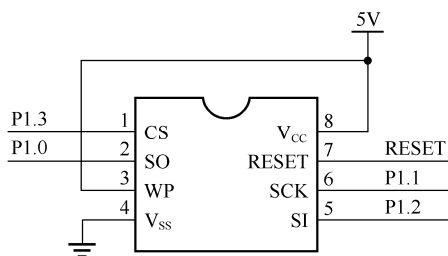


图 5-33 X5045 的指令寄存器

下面介绍用 C51 单片机的汇编语言模拟 SPI 串行输入、串行输出和串行输入/输出的 3 个子程序, 实际上, 这些子程序也适用于在串行时钟的上升沿输入和下降沿输出的其他各种串行外围接口芯片 (如 A/D 转换芯片、网络控制器芯片、LED 显示驱动芯片等)。对于下降沿输入、上升沿输出的各种串行外围接口芯片, 只要改变 P1.1 的输出电平顺序, 即先置 P1.1 为低电平, 之后再次置 P1.1 为高电平, 再置 P1.1 为低电平……则这些子程序也同样用。

(2) 程序设计

操作码:

WREN	0x06	设置写允许位
WRDI	0x04	复位写允许位
RDSR	0x05	读状态寄存器
WRSR	0x01	写状态寄存器
READ	0x03/0x0b	读操作时内部 EEPROM 页地址
WRITE	0x02/0x0a	写操作时内部 EEPROM 页地址

【例 5-3】 C51 模拟 SPI 的程序设计。

程序设计如下所示：

```
#include <reg51.h>
sbit CS= P2^7;
sbit SO= P2^6;
sbit SCK= P2^5;
sbit SI= P2^4;
#define WREN 0x06
#define WRDI 0x04
#define RDSR 0x05
#define WRSR 0x01
#define READ0 0x03
#define READ1 0x0b
#define WRITE0 0x02
#define WRITE1 0x0a
#define uchar unsigned char
uchar ReadByte()                                //从设备中读一个字节
{
    bit bData;
    uchar ucLoop;
    uchar ucData;
    for(ucLoop=0;ucLoop<8;ucLoop++)
    {
        SCK=1;
        SCK=0;
        bData=SO;
        ucData<<=1;
        if(bData)
        {
            ucData |=0x01;
        }
    }
    return ucData;
}
void WriteByte(uchar ucData)                    //写一个字节到设备
{
    uchar ucLoop;
    for(ucLoop=0;ucLoop<8;ucLoop++)
    {
```

```

        if((ucData&0x80)==0)                //高位 (MSB) 先发
            SI=0;
        else
            SI=1;
        SCK=0;
        SCK=1;
        ucData<<=1;
    }
}

uchar ReadReg()                            //读寄存器
{
    uchar ucData;
    CS=0;
    WriteByte(RDSR);
    ucData=ReadByte();
    CS=1;
    return ucData;
}

uchar WriteReg(uchar ucData)              //写寄存器
{
    uchar ucTemp;
    ucTemp=ReadReg();
    if((ucTemp&0x01)==1)                  //设备忙
        return 0;
    CS=0;
    WriteByte(WREN);                      //当写 WREN 时, cs 必须为高电平
    CS=1;
    CS=0;
    WriteByte(WRSR);
    WriteByte(ucData);
    CS=1;
    return 1;
}

void WriteEpm(uchar cData, uchar cAddress, bit bRegion)
/* 写入一个字节, cData 为写入的数, cAddress 为写入地址, bRegion 为页 */
{
    while((ReadReg()&0x01)==1);          //设备忙
    CS=0;

```

```

WriteByte(WREN);
CS=1;
CS=0;
if(bRegion==0)
    WriteByte(WRITE0);           //写页面地址
else
    WriteByte(WRITE1);
WriteByte(cAddress);
WriteByte(cData);
SCK=0;
CS=1;
}
uchar ReadEpm(uchar cAddress, bit bRegion)
/* 读入一个字节, cAddress 为读入地址, bRegion 为页 */
{
    uchar cData;
    while((ReadReg() & 0x01) == 1); //设备忙
    CS=0;
    if(bRegion==0)
        WriteByte(READ0);
    else
        WriteByte(READ1);
    WriteByte(cAddress);
    cData=ReadByte();
    CS=1;
    return cData;
}
main()
{
    WriteReg(0x00);               //置位看门狗时间为 1.4s
    CS=1;
    CS=0;                         //复位看门狗
}

```

5.9 单片机配置 FPGA

Altera FLEX10K 是工业界第一个嵌入式 PLD，由于其具有高密度、低成本、低功率等特点，并具有 ICR (In-Circuit Reconfigurability，在线可重配置) 的特点，受到了广大设计者的

喜爱，目前日趋广泛地应用于数字系统设计中。

基于 SRAM 查找表结构的 FLEX10K 器件，其逻辑功能和互连关系是由存储在 SRAM 中的配置数据决定的，由于 SRAM 的易失性，每次加电期间，都必须对其进行重新配置，将 I/O 引脚和进入了用户状态。FLEX10K 器件配置分为主动配置模式和被动配置模式方式。根据数据线宽度 FLEX10K 器件配置分为 8 位并行配置和串行配置两类。

5.9.1 FLEX10K 的工作状态

FLEX10K 器件有三种工作状态：第一种为用户状态（Usermode），指电路中 FLEX10K 器件正常工作时的状态；第二种称为配置状态（Configuration），指将配置数据装入 FLEX10K 器件；第三种就是初始化状态（Initialization），FLEX10K 器件复位各类寄存器，让 I/O 引脚为器件正常工作做准备。

5.9.2 FLEX10K 器件的配置

FLEX10K 器件的配置引脚可分为专用配置引脚和非专用配置引脚两类。专用配置引脚只在配置时起作用，非专用引脚配置完成后作为普通的 I/O 引脚使用。

专用配置引脚：模式选择引脚 MSEL1 和 MSEL0；配置状态输出引脚 nSTATUS；配置控制引脚 nCONFIG；时钟提供引脚 DCLK；配置状态输入输出引脚 CONF_DONE；使能输入引脚 nCE；使能输出引脚 nCEO。非专用配置引脚包括 nWS、nRS、RDYnBSY、CS、nCS、CLKUSR、DATA[7..0]、DATA0、DATA7、INIT_DONE、DEV_OE、DEV_CLRn。

对 FLEX10K 器件的配置有两种方式，即主动配置和被动配置。

采用主动配置的方法，即在系统对 SRAM 加载配置数据时，由可编程逻辑器件自身控制整个配置过程。FLEX10K 系列常用的主动配置是 AS（主动串行）方式，即将编程文件存放在 Altera 公司的 EPC 系列专用存储器中，在芯片上电时由 FLEX 器件控制整个配置过程，实现将编程文件串行地送到 FLEX 器件的 DATA0 引脚进行配置，并在配置结束后自动进行器件的初始化过程，并进入用户状态。该方法简单方便，不需要其他的外围控制器，由 FLEX 器件自身引导整个配置过程，但是必须采用专用存储器放置配置数据，而对专用存储器的编程也需要专用的编程硬件，相对来说成本较高。

采用被动配置（PS 或 PPA、PPS）的方式，编程文件可以放在通用程序存储器中，如 EPROM、EEPROM 或 FLASH 中，在 FLEX 器件上电后，由芯片外部控制器自动地从通用存储器中读出编程文件并送到 FLEX 器件进行配置，数据传送方式可以为串行，也可以为并行。串行传送时，即 PS 方式，配置数据送至 FLEX 的 DATA0 引脚；并行传送时，即 PPA 或 PPS 方式，配置数据送至 FLEX 器件的 DATA0 到 DATA7 引脚。

5.9.3 主动串行模式配置 FLEX10K 器件

主动串行（Active-Serial，AS）配置模式由 Altera 提供的专用配置器件 EPC（如 EPC1 是一次可编程器件，EPC2 是多次可编程器件）向 FLEX10K 输入串行位流的配置数据。

EPC 中的配置数据是 MAX+PLUS II 软件生成的编程目标文件。配置数据大于单个 EPC 的容量时，可以级连使用多个此类器件（EPC1441 不支持级连，各个器件之间的握手信号由 nCASC 和 nCS 提供）。图 5-34 所示是单个器件配置单个 FLEX10K 的配置电路原理图。

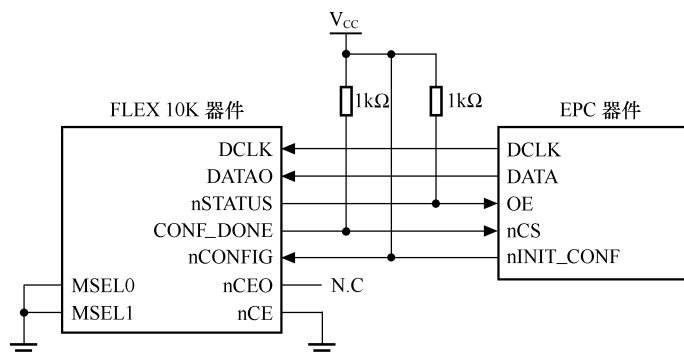


图 5-34 FLEX10K 的配置电路原理图

在配置过程中，如果发生错误，FLEX10K 将 nSTATUS 拉至低电平，复位 EPC 和 FLEX10K。如果在 MAX+PLUS II 软件内，ASSIGN 菜单的 GlobalProjectDeviceOption 对话框中的“Auto-Restart Configuration on Frame Error”选项打开，FLEX10K 器件将在发生错误配置之后重新自动配置；否则，外部系统必须监视 nSTATUS 引脚以检测错误的发生，然后给 nCONFIG 一个脉冲重新配置。当配置完成后，FLEX10K 释放 CONF_DONE，将 EPC 与系统隔离。

5.9.4 基于 C51 的被动串行配置 FLEX10K 器件

1. 配置基本原理

被动串行（Passive Serial, PS）配置模式中，由 Bit-Blaster、FLEX10K 下载电缆或微处理器产生一个由低到高的跳变电平送到 nCONFIG 引脚，然后微处理器或编程硬件将配置数据送到 DATA0 引脚，数据锁存至 CONF_DONE 变为高电平，先将每字节的最低位 LSB 送到 FLEX10K 器件。CONF_DONE 变为高电平后，DCLK 必须再送出 10 个周期来初始化 FLEX10K 器件。

采用被动串行（PS）模式对 FLEX10K 进行配置，基于以下几个方面的考虑：

- （1）PS 方式连线最简单；
- （2）与 Configuration EPROM 方式可以兼容（MSEL0、1 设置不变）；
- （3）与并行配置相比，误操作的几率小，可靠性高；
- （4）只需利用 MCU 的 5 个 I/O 线，就可对 FLEX10K 进行 PS 方式的配置。

2. 配置文件的准备

由开发软件 MAX+PLUS II 产生的编程目标文件（.pof）或 SRAM 目标文件（.sof），文件不能直接用于 CPU 配置 FLEX10K 中，需要进行数据转换才能得到软件可用的配置数据。开

发工具可以生成多种配置文件。在 MAX+PLUS II 编程状态选择文件菜单的 con-vertobjectfiles 命令，指定要转换的文件并将输出文件格式选为.rbf。 .rbf 即为二进制的配置文件，包括所有的配置数据，一个字节的.rbf 文件有 8 位数据，在配置时最低位先装入。

3. FPGA 配置

CPU 仅需要利用 5 个 I/O 引脚与 FPGA 相连，就实现了 PS 方式的硬件连接，如图 5-35 所示。

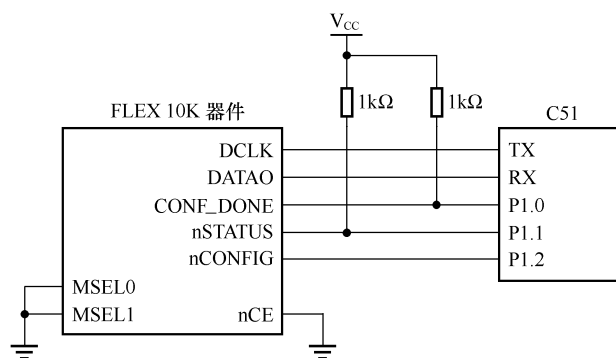


图 5-35 FLEX10K 器件 PS 模式配置图

【例 5-4】 FLEX10K 器件 PS 模式配置。

相应的配置程序如下所示：

```
void load_epld(void)
{
    unsigned char data i;
    unsigned int data j;
    unsigned char xdata * data pt;
    SCON = 0x0;                                /* 设置 8031 工作在方式 0，同步串行方式 */
    CONFIG = 0;                                /* 初始化 FPGA */
    i=4;                                        /* 注意 8031 的工作频率和速度，对于 Xilinx
                                           的 FPGA，初始化的时间要长一些，具体查一
                                           下各个 FPGA 的参数 */

    while(STATUS!=0) {
        i--;
        if(i==0) {
            CONFIG = 0;
            FPGA_Init_Error();                /* FPGA 初始化错误处理程序，自己编制 */
        }
    }

    CONFIG = 1;                                /* 初始化 FPGA 完毕 */
}
```

```

i=10;                                /* 注意 8031 的工作频率和速度，各个 FPGA
                                      的初始化的时间不一样，具体查下各个 FPGA
                                      的参数 */

while(STATUS!=1) {
    i--;
    if(i==0) {
        CONFIG = 0;                  /* 错误时，使 FPGA 处于初始化状态，保证
                                      电路处于安全状态 */

        FPGA_Init_Error();           /* FPGA 初始化错误处理程序，自己编制 */
    }
}

pt=(unsigned char xdata *)EPLD_DATA; /* EPLD_DATA 为 FPGA 的存放地址 */
for(j=0; j < EPLD_Number; j++) {    /* EPLD_Number 为 FPGA 的加载字节数，当
                                      大于 64KB 时，要采用页面切换的方式 */

    SBUF = *pt++;
    while(!TI);                      /* 等待 1 个字节加载完毕 */
    TI=0;
    if(STATUS!=1) {                  /* 加载过程中，检查加载是否正确 */
        CONFIG = 0;                  /* 错误时，使 FPGA 处于初始化状态，保证
                                      电路处于安全状态 */

        FPGA_Load_Error();           /* FPGA 加载错误处理程序，自己编制 */
    }
}
if(CFG_DOWN!=1) {                    /* FPGA 数据加载完毕，检查加载是否正确 */
    CONFIG = 0;                      /* 错误时，使 FPGA 处于初始化状态，保证
                                      电路处于安全状态 */

    FPGA_Done_Error();               /* FPGA 加载错误处理程序，自己编制 */
}
}

```

习题和思考题

5-1 使用 3-8 译码器最多可以扩展多少片 6264，试画出其逻辑扩展图，并且写出每一片的地址范围。

5-2 外部数据 RAM 从 2000H 开始有 100 个数据，现要将它们移动到从 2030H 开始的区域，试编写有关的程序。

5-3 画出 8031 扩展一片 28C16 作为程序存储器的逻辑扩展图，并写出其地址范围。

5-4 与 8031 接口的 8255 片内 4 个端口地址(A 口、B 口、C 口、命令口)分别为 DFFCH ~ DFFFH。对 8255 编程初始化,并由 A 口输出数据 AAH;由 B 口输入十个数到片内 RAM 区;由 PC4 位产生一个负脉冲,低电平宽度为 10 μ s。

5-5 设某个生产过程有六个工序,每道工序的时间分别为 10s、8s、12s、15s、9s 和 6s。设延迟程序 DELAY 的延时为 1s。用单片机通过 8255 的 A 口来进行控制。A 口中的一位就可控制某一工序的起停。试编写有关的程序。

5-6 在顺序控制过程中,有时会遇到一些告警信号,以便在出现不正常情况时进行处理。设单片机通过 8155 来进行控制,A 口输出顺序控制信号。设仍为六道工序,每道工序 10s。C 口的某一位来接收告警信息。出现告警时,单片机中断,然后送出顺序控制信号。同时从 B 口输出告警控制信号,使警铃或灯发出指示(只需 1 位输出)。编写有关的程序。

5-7 用单片机定时器进行定时以产生顺序控制信号。设仍为六道工序,每道工序的时间为 10s。告警信号有两路接收到 C 口,用查询方法来获得告警信息。告警之后从 B 口送出控制信号,分别应为 06H 和 05H。使用 8155 作为接口芯片,试编写有关的程序,包括主程序和中断服务程序。

5-8 分别编写 I²C 口和 SPI 口 RAM 的读写程序。

5-9 单片机配置 FPGA 的方式有哪几种?各有什么特点?

第 6 章 C51 单片机的机间串行通信

重点与难点

本章重点是点对点的串行异步通信、多机通信的基本编程格式以及 PC 和单片机的通信方式。

基本要求

1. 掌握点对点的串行异步通信
2. 掌握多机通信的方法及编程格式
3. 掌握 PC 和单片机之间的通信方法

本章内容

1. 点对点的串行异步通信
2. 多机通信
3. PC 和单片机之间的通信方法

6.1 点对点的串行异步通信

6.1.1 通信双方的硬件连接

如果采用单片机自身的 TTL 电平直接传输信息，其传输距离一般不超过 1.5m。C51 单片机一般采用 RS-232C 标准进行点对点的通信连接。图 6-1 所示为两个 8031 间的连接方法，信号采用 RS-232C 电平传输，电平转换芯片采用 MAX232。

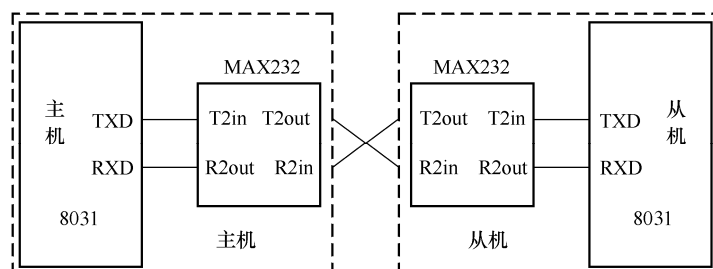


图 6-1 单片机 RS-232C 电平信号的传输

6.1.2 通信双方的约定

按照图 6-1 所示的接口电路,当主机开始发送数据时,先发送一个“AA”信号,从机收到后回答一个“BB”,表示同意接收。当主机收到“BB”后,开始发送数据,每发送一次求“校验和”,假定数据块长度为 16 个字节,数据缓冲区为 buf,数据块发送完后马上发送“校验和”。

从机接收数据并将其转存到数据缓冲区 buf,每接收到一个数据便计算一次“校验和”,当收齐一个数据块后,再接收主机发来的校验和,并将它与从机求出的校验和进行比较。若两者相等,说明接收正确,从机回答 00H;若两者不等,说明接收不正确,从机回答 0FFH,请求重发。

主机收到 00H 的回答后,结束发送。若收到的答复非零,则将数据再重发一次。双方约定的传输波特率若为 1200 波特/秒,查表可知,在双方的 $f_{osc}=11.0592\text{ MHz}$ 下,T1 工作在定时器方式 2,TH1=TL1=0E8H,PCON 寄存器的 SMOD 位为 0。

按照上述约定,发送和接收程序框图如图 6-2 所示。

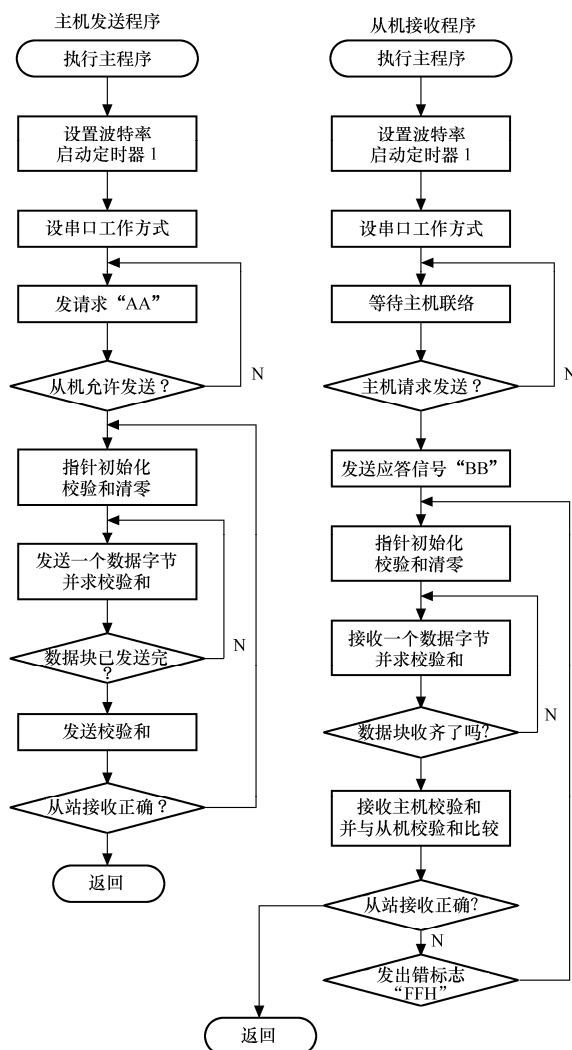


图 6-2 点对点通信的程序框图

6.1.3 点对点通信编程

点对点通信双方基本等同，只是人为规定一个为发送，一个为接收。要求两机串行口的波特率相同，因而发送和接收方串行口的初始化相同。可编写含有初始化函数、发送函数、接收函数的程序，在主函数中根据程序的发送、接收设置 TR，采用条件判别决定使用发送函数还是接收函数。这样点对点通信的双方都可运行此程序，只需在程序运行之前进行人为设置选择 TR，一个令 TR=0，一个令 TR=1，然后分别编译，在两机上分别装入，同时运行。

【例 6-1】 点对点通信编程。

点对点通信的程序 ppcmun.c 如下所示：

```
#include<reg51.h>
#define uchar unsigned char
#define TR 1 /*发送接收差别值 TR=0 发送*/
uchar idata buf[10];
uchar pf;

void init(void) /*串行口初始化*/
{
    TMOD=0x20; /*设 T/C1 为定时方式 2*/
    TH1=0xe8; /*设定波特率*/
    TL1=0xe8;
    PCON=0x00;
    TR1=1; /*启动 T/C1*/
    SCON=0x50; /*串行口工作在方式 1*/
}

void send(uchar idata *d)
{
    uchar i;
    do{
        SBUF=0xaa; /*发送联络信号*/
        while(TI==0); /*等待发送出去*/
        TI=0;
        while(RI==0); /*等待从机回答*/
        RI=0;
    }while((SBUF^0xbb)!=0); /*从机未准备好，继续联络*/
    do{
        pf=0; /*清校验和*/
        for(i=0;i<16;i++){
```

```

        SBUF=d[i];          /*发送一个数据*/
        pf+=d[i];          /*求校验和*/
        while(TI==0);
        TI=0;
    }
    SBUF=pf;                /*发送校验和*/
    while(TI==0);
    TI=0;
    while(RI==0);
    RI=0;                   /*等待从机应答*/
}while(SBUF!=0);          /*回答出错，则重发*/
}
void receive(uchar idata *d)
{
    uchar i;
    do{
        while(RI==0);
        RI=0;
    }while((SBUF^0xaa)!=0); /*判主机请求否*/
    SBUF=0xbb;              /*发应答信号*/
    while(TI==0);
    TI=0;
    while(1)
    {
        pf=0;              /*清校验和*/
        for(i=0;i<16;i++)
        {
            while(RI==0);
            RI=0;
            d[i]=SBUF;      /*接收一个数据*/
            pf+=d[i];       /*求校验和*/
        }
        while(RI==0);
        RI=0;              /*接收 A 机校验和*/
        if((SBUF^pf)==0)
        {                   /*比较校验和*/
            SBUF=0x00;
            break;
        }
    }                       /*校验和相同发“00”*/
}

```

```

else
{
    SBUF=0xff;          /*出错发"FF", 重新接收*/
    while(TI==0);
    TI=0;
}
}
}

void main(void)
{
    init();
    if(TR==0)
    {
        send(buf);
    }
    else
    {
        receive(buf);
    }
}

```

6.2 多机通信

6.2.1 通信接口

图 6-3 所示是在单片机多机系统中经常采用的总线型主从式多机系统。所谓主从式，即在数个单片机中，有一个是主机，其余的为从机，从机要服从主机的调度、支配。单片机的串行口方式 2、方式 3 很适合这种主从式的通信结构。当然，在采用不同的通信标准通信时，

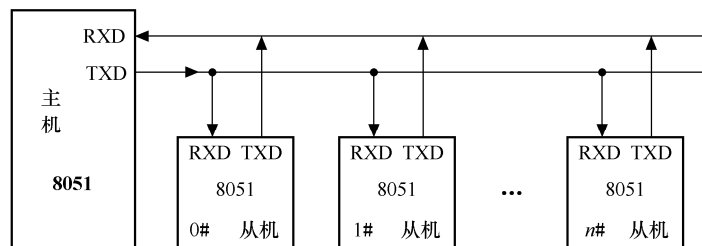


图 6-3 总线型主从式多机系统

还需进行相应的电平转换,也可以对传输信号进行光电隔离。在多机系统中,通常采用 RS-422 或 RS-485 串行标准总线进行数据传输。

6.2.2 通信协议

根据 C51 串行口的多机通信能力,多机通信可以按照以下协议进行。

(1) 首先使所有从机的 SM2 位置 1,使其处于只接收地址帧的状态。

(2) 主机先发送一帧地址信息,其中 8 位地址,第 9 位为地址/数据信息的标志位,该位置 1 表示该帧为地址信息。

(3) 从机接收到地址帧后,各自将接收的地址与本从机的地址比较。对于地址相符的那个从机,使 SM2 位清 0,以接收主机随后发来的所有信息;对于地址不符的从机,仍保持 SM2=1,对主机随后发来的数据不予理睬,直至发送新的地址帧。

(4) 当从机发送数据结束后,发送一帧校验和,并置第 9 位(TB8)为 1,作为从机数据传送结束标志。

(5) 主机接收数据时先判断数据结束标志(RB8),若 RB8=1,表示数据传送结束,并比较此帧校验和,若正确,则回送正确信号 00H,此信号令该从机复位(即重新等待地址帧);若校验和出错,则发送 0FFH,令该从机重发数据。若接收帧的 RB8=0,则送原数据到缓冲区,并准备接收下帧信息。

(6) 若主机向从机发送数据,从机在第(3)步中比较地址相符后,从机令 SM2=0,同时把本站地址发回主机,作为应答之后才能收到主机发送来的数据。其他从机继续监听地址(SM2=1),无法收到数据。

(7) 主机收到从机的应答地址后,确认地址是否相符。如果地址不符,发送复位信号(数据帧中 TB8=1);如果地址相符,则清 TB8,开始发送数据。

(8) 从机收到复位命令后回到监听地址状态(SM2=1),否则开始接收数据和命令。

6.2.3 通信程序

设主机发送的地址联络信号 00H、01H、02H 为从机设备地址,地址 FFH 是命令各从机恢复 SM2 为 1 的状态即复位。主机的命令编码为:

- (1) 01H 请求从机接收主机的数据命令;
- (2) 02H 请求从机向主机发送数据命令;
- (3) 其他都按从机向主机发送数据命令 02H 对待。

从机的状态字节格式为:

D7	D6	D5	D4	D3	D2	D1	D0
ERR	0	0	0	0	0	TRDY	RRDY

(1) RRDY=1: 从机准备好接收主机的数据;

(2) TRDY=1: 从机准备好向主机发送数据;

(3) ERR=1: 从机接收到的命令是非法的。

通常从机以中断方式控制和主机的通信。程序可分成主机程序和从机程序，约定一次传送的数据为 16 个字节，这里以 02H 地址的从机为例。

【例 6-2】 多机通信编程。

1. 主机程序

主机程序流程图如图 6-4 所示。

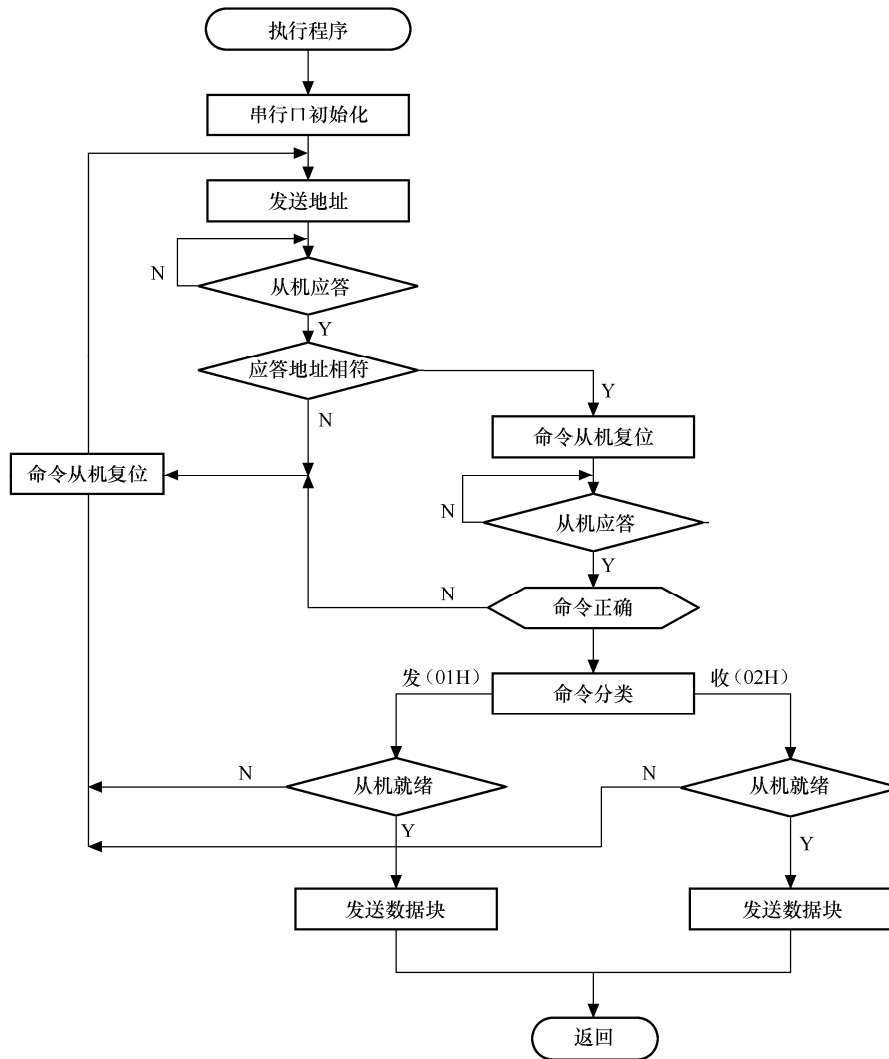


图 6-4 多机通信主机程序流程

主机程序 master.c 如下所示：

```

#include<reg51.h>
#define uchar unsigned char
#define SLAVE 0x02          /*从机地址*/
#define BN 16
  
```



```
uchar idata rbuf[16];
uchar idata tbuf[16]="master transmit";
void err(void)
{
    SBUF=0xff;
    while(TI!=1);
    TI=0;
}
uchar master(uchar addr, uchar command)
{
    uchar aa, i, P;
    while(1)
    {
        SBUF=SLAVE;           /*发呼叫地址*/
        while(TI!=1);
        TI=0;
        while(RI!=1);
        RI=0;                 /*等待从机回答*/
        if(SBUF!=addr)
            err();            /*若地址错，发复位信号*/
        else
        {
            /*地址相符*/
            TB8=0;            /*清地址标志*/
            SBUF=command;     /*发命令*/
            while(TI!=1);
            TI=0;
            while(RI!=1);
            RI=0;
            aa=SBUF;          /*接收状态*/
            if((aa&0x08)==0x08)
            {
                TB8=1;
                err();
            }                /*若命令未被接收，发复位信号*/
            else
            {
                if(command==0x01) /*是发送命令*/
                {
                    if((aa&0x01)==0x01) /*从机准备好接收*/
```

```

{
    do{
        P=0;                /*清校验和*/
        for(i=0;i<BN;i++)
        {
            SBUF=tbuf[i];    /*发送一数据*/
            p+=tbuf[i];
            while(TI!=1);
            TI=0;
        }
        SBUF=p;              /*发送校验和*/
        while(TI==0);
        TI=0;
        while(RI==0);
        RI=0;
    }while(SBUF!=0);        /*接收不正确，重新发送*/
    TB8=1;                  /*置地址标志*/
    return(0);
}
}
else
{
    if((aa&0x02)==0x02)    /*是接收命令，从机准备好发送*/
    {
        while(1)
        {
            P=0;            /*清校验和*/
            for(i=0;i<BN;i++)
            {
                while(RI!=1);
                RI=0;
                rbuf[i]=SBUF; /*接收一数据*/
                p+=rbuf[i];
            }
            while(RI==0);
            RI=0;
            if(SBUF==p)
            {
                SBUF=0x00;    /*校验和相同发"00"*/
            }
        }
    }
}

```

```

        while(TI==0);
        TI=0;
        break;
    }
    else
    {
        SBUF=0xff;        /*校验和不同发"OFF", 重新接收*/
        while(TI==0);
        TI=0;
    }
}
TB8==1;                /*置地址标志*/
return(0);
}
}
}
}
}

void main(void)
{
    TMOD=0x20;           /*T/C1 定义为方式 2*/
    TL1=0xfd;
    TH1=0xfd;            /*置初值*/
    PCON=0x00;
    TRI=1;
    SCON=0xf0;           /*串行口为方式 3*/
    master(SLAVE, 0x01);
    master(SLAVE, 0x02);
}

```

2. 从机程序

从机中断服务程序的流程如图 6-5 所示。

从机程序 slave.c 如下所示：

```

#include<reg51.h>
#define uchar unsigned char
#define SLAVE 0x02
#define BN 16

```

```
uchar idata trbuf[16];
```

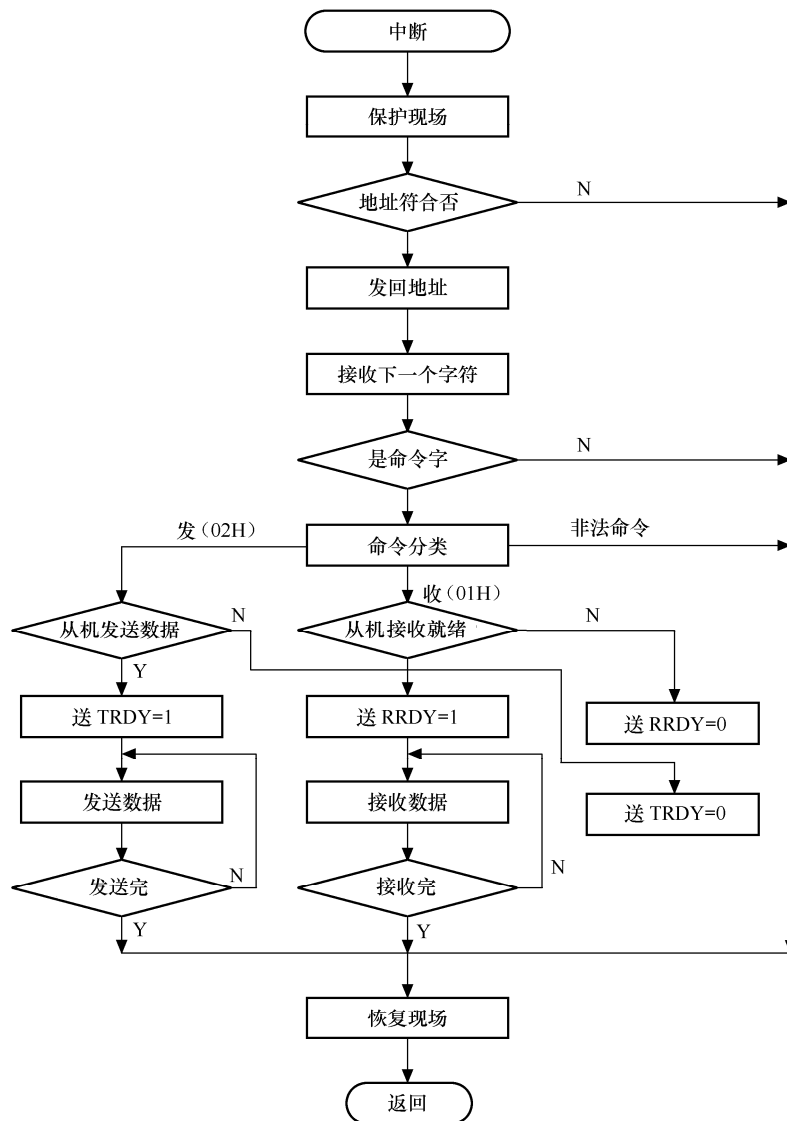


图 6-5 多机通信从机中断程序流程

```
uchar idata rebuf[16];
```

```
bit tready;
```

```
bit rready;
```

```
void main(void)
```

```
{
```

```
    TMOD=0x20;
```

```
    /*T/C1 定义为方式 2*/
```

```
    TL1=0xfd;
```

```
    /*置初值*/
```

```
    TH1=0xfd;
```

```
    PCON=0x00;
```

```
    TR1=1;
    SCON=0xf0;          /*串行口为方式 3*/
    ES=1;               /*开串行口中断*/
    EA=1;
    while(1)
    {
        tready=1;
        rready=1;
    }                    /*假定准备好发送和接收*/
}

void ssio(void) interrupt 4 using 1
{
    void str(void);
    void sre(void);
    uchar a, i;
    RI=0;
    ES=0;               /*关串行口中断*/
    if(SBUF!=SLAVE)
    {
        ES=1;
        goto reti;
    }                    /*非本机地址、继续监听*/
    SM2=0;              /*取消监听状态*/
    SBUF=SLAVE;         /*从机地址发回*/
    while(TI!=1);
    TI=0;
    while(RI!=1);
    RI=0;
    if(RB8==1)
    {
        SM2=1;
        ES=1;
        goto reti;
    }                    /*是复位信号，恢复监听*/
    a=SBUF;             /*接收命令*/
    if(a==0x01)
    {                    /*从机接收主机的数据*/
        if(rready==1)
```

```
        SBUF=0x01;          /*接收准备好发状态*/
    else
        SBUF=0x00;
    while(TI!=1);
    TI=0;
    while(RI!=1);
    RI=0;
    if(RB8==1)
    {
        SM2=1;
        ES=1;
        goto reti;
    }
    sre();                  /*接收数据块*/
}
else
{
    if(a==0x02)
    {                      /*从机向主机发数据*/
        if(tready==1)
            SBUF=0x02;    /*发送准备好发状态*/
        else
            SBUF=0x00;
        while(TI!=1);
        TI=0;
        while(RI!=1);
        RI=0;
        if(RB8==1)
        {
            SM2=1;
            ES=1;
            goto reti;
        }
        str();            /*发送数据块*/
    }
    else
    {
        SBUF=0x80;        /*命令非法，发状态*/
        while(TI!=1);
    }
}
```

```
        TI=0;
        SM2=1;
        ES=1;          /*恢复监听*/
    }
}
reti: ;
}

void str(void)          /*发送数据块*/
{
    uchar P, i;
    tready=0;
    do{
        p=0;            /*清校验和*/
        for(i=0;i<BN;i++)
        {
            SBUF=trbuf[i];    /*发送一数据*/
            p+=trbuf[i];
            while(TI!=1);
            TI=0;
        }
        SBUF=p;            /*发送校验和*/
        while(TI==0);
        TI=0;
        while(RI==0);
        RI=0;
    }while(SBUF!=0);      /*主机接收不正确，重新发送*/
    SM2=1;
    ES=1;
}

void sre(void)          /*接收数据块*/
{
    uchar p, i;
    rready=0;
    while(1)
    {
        p=0;            /*清校验和*/
        for(i=0;i<BN;i++)
```

```

{
    while(RI!=1);
    RI=0;
    rebuf[i]=SBUF;          /*接收一数据*/
    p+=rebuf[i];
}
while(RI!=1);
RI=0;
if(SBUF==p)
{
    SBUF=0x00;
    break;
}                      /*校验和相同发"00"*/
else
{
    SBUF=0xff;           /*校验和不同发"OFF", 重新接收*/
    while(TI==0);
    TI=0;
}
}
SM2=1;
ES=1;
}

```

6.3 PC 和单片机之间的通信方法

在单片机应用系统中,经常需要通过 RS-232 串行口与微机进行通信。目前在各种操作系统中,Microsoft 的 Windows 较为常见,而且大多为 Windows95/98 等 32 位平台。以往在 Windows 平台上的串行通信多使用其提供的 API 函数来实现,这种方法使用起来需要许多底层设置,因而较为烦琐,并且难以理解。Microsoft 推出的 ActiveX 技术提供了另外一种实现串行通信的方法。这种方法不仅相对较为简单,而且非常实用。尤其是在 Visual C++ 这种可视化面向对象的编程环境中,可以真正把串口看作一个对象,编程时只需简单的设置,理解起来也很容易。下面详细讨论 Microsoft 提供的串行通信 ActiveX 控件的使用方法。该控件的相应文件是 MSCOMM32.OCX, 以下简称为 MSCOMM 控件。

6.3.1 MSCOMM 控件

MSCOMM 控件,即 Microsoft Communication Control,是 Microsoft 为简化 Windows 下

串行通信编程而提供的 ActiveX 控件。它提供了一系列标准通信命令的使用接口，利用它可以建立与串口的连接，并可以通过串口连接到其他通信设备（如调制解调器），发出命令、交换数据以及监视和响应串行连接中发生的事件和错误。MSCOMM 控件可用于创建电话拨号程序、串口通信程序和功能完备的终端程序。

1. MSCOMM 控件通信方式

MSCOMM 控件提供了以下两种处理通信的方式。

(1) 事件驱动方式

当通信事件发生时，MSCOMM 控件会触发 OnComm 事件，调用者可以捕获该事件，通过检查其 CommEvent 属性便可确认发生的是哪种事件或错误，从而进行相应的处理。这种方法的优点是响应及时、可靠性高。

(2) 查询方式

在程序的每个关键功能之后，可以通过检查 CommEvent 属性的值来查询事件和错误。如果应用程序较小，这种方法可能更可取。例如，如果写一个简单的电话拨号程序，则没有必要每接收 1 个字符都产生事件，因为唯一等待接收的字符是调制解调器的“确定”响应。

在使用 MSCOMM 控件时，1 个 MSCOMM 控件只能同时对应 1 个串口。如果应用程序需要访问和控制多个串口，那么必须使用多个 MSCOMM 控件。

在 VC++ 中，MSCOMM 控件只对应着 1 个 C++ 类，即 CMSCComm。由于 MSCOMM 控件本身没有提供方法，所以 CMSCComm 类除了 Create() 成员函数外，其他的函数都是 Get/Set 函数对，用来获取或设置控件的属性。MSCOMM 控件也只有 1 个 OnComm 事件，用来向调用者通知有通信事件发生。

2. MSCOMM 控件的属性

MSCOMM 控件有许多很重要的属性，限于篇幅只给出几个较为重要和常用的属性，如下所示。

- (1) CommPort 属性：设定通信端口号。
- (2) HandShaking 属性：设定通信协议。
- (3) Settings 属性：以字符串形式表示的波特率、奇偶校验、数据位。
- (4) PortOpen 属性：通信端口的状态，设置打开或关闭端口。
- (5) Rthreshold 属性：在产生 OnComm 事件之前要接收的字符数。
- (6) Sthreshold 属性：在产生 OnComm 事件之前，设置并返回传输缓冲区允许的最小字符数。
- (7) InputMode 属性：接收数据的类型，0 为文本，1 为二进制。
- (8) InputLen 属性：设置从接收缓冲区读取的字符数。默认值为 0，表示使用 Input 将是 MSCComm 控件读取接收缓冲区的全部内容。
- (9) InBufferSize 属性：设置输入缓冲区大小。
- (10) InBufferCount 属性：可以返回输入缓冲区内等待读取的字节个数。
- (11) Input 属性：表示从缓冲区移走一串字符，将缓冲区中收到的数据读入 VARIANT

变量。

- (12) OutBufferSize 属性：设置发送缓冲区大小。
- (13) OutBufferCount 属性：可以返回发送缓冲区内等待发送的字节个数。
- (14) Output 属性：等待发送缓冲区写数据流。
- (15) CommEvent 属性：其属性值反映错误或事件的类型。

6.3.2 编程实现

MSCOMM 控件可以两种不同的形式接收数据，即文本形式和二进制形式。用 MSCOMM 控件进行字符数据传输的文献和资料可以找到很多，在 Microsoft 的 MSDN (Microsoft Developer Network) 中就可以找到这样的例子，即 VCTERM。可是几乎所有以单片机为核心的测量系统所得到的原始数据都是二进制形式的，因此，以二进制形式传输数据将是最为直接而又简洁的办法。不仅如此，由于 MSCOMM 控件在文本形式下，其传输的是宽字符格式的字符，要想得到有用信息，还要额外处理。因此本书主要讨论在二进制形式下的使用方法。

在 VC++6.0 中，用 APPWizard 可以生成三种应用程序：单文档 (SDI)、多文档 (MDI) 和基于对话框的应用程序。为了说明问题和省去不必要的细节，下面以基于对话框的应用程序为例。

1. 创建一个基于对话框的应用程序

打开 VC++6.0 集成开发环境，选择菜单项 File/New，在出现的对话框中选中 Projects 标签中的 MFC AppWizard (exe)，然后在 Project Name 框中填入 MyCOMM (可根据需要命名)，之后点 OK 按钮。在接着出现的对话框中选中 Dialog Based 项，然后点 NEXT 按钮。以下的各对话框都按照缺省设置，这样即可生成一个基于对话框的应用程序。在资源编程器中会出现其对话框模板。

2. 插入 MSCOMM 控件

选择菜单项 Project/Add to project/Components and Controls...，在弹出的对话框中选择 Registered ActiveX Controls 文件夹下的 Microsoft Communications Control, version 6.0，然后按下 Insert 按钮，接着会弹出一个对话框，提示生成的类名及文件名，按 OK 按钮即可实现控件的插入。这时在对话框的控件工具栏上会多出一个电话机模样的控件图标，Workspace 的 Classview 中也多了一个类 CMSComm。

此时即可将 MSCOMM 控件加入到对话框模板，加入方法与其他控件一样，然后还要在对话框类中相应加入一个成员变量，此处将其命名为 m_comm。成员变量的加入方法是：首先在对话框模板中，用鼠标右键点击该控件，选择 ClassWizard，在出现的对话框的 Member Variables 标签的 Control Ids 项下，选中 IDC_MSCOMM1，然后按 Add Variable...按钮，在出现的对话框的 Member Variable Name 项中输入 m_comm，最后按 OK 按钮即可。

3. 设置属性

通常可以在两个地方对控件的属性进行设置。

(1) 在对话框资源编辑器中。在对话框模板上,用右键单击 **MSCOMM** 控件,然后选择 **Properties...** 菜单项,然后便可设置各项属性。此处只对以下几处进行改动: **Rthreshold: 1**; **InputLen: 1**; **DTREnable: 不选**; **InputMode: 1-Binary**; 其他接受缺省设置。

(2) 在对话框类的 **OnInitDialog()** 函数中。下面是以上设置的具体函数实现。

```

BOOL CMyCOMMDlg:: OnInitDialog ()
{
    CDialog:: OnInitDialog ();

    //TODO:                                //此处为应用框架自动生成代码,不予列出
    //此处添加其他初始化程序

    m_comm.SetCommPort(1);                //使用串口 1
    m_comm.SetSettings("9600, N, 8, 1");
    //波特率为 9600, 无奇偶校验, 8 位数据位, 1 位停止位
    m_comm.SetRThreshold(10);              //每接收 10 个字符就触发 1 次接收事件
    m_comm.SetSThreshold(0);               //不触发发送事件
    m_comm.SetInputLen(10);                //每次读操作从缓冲区中取 10 个字符
    m_comm.SetInputMode(1);                //二进制数据传输形式
    m_comm.SetPortOpen (TRUE);             //打开串口
    return TRUE;                           //除非锁定控件, 否则返回 TRUE
}

```

4. 发送二进制数据

如果需要发送二进制数据,可将数据作如下处理,即具体代码如下所示:

```

CByteArray bytOutArr;
bytOutArr.Add(0x0, data);                //给数组第一个字符赋值
bytOutArr.Add(0x1, data);                //给数组第二个字符赋值
bytOutArr.Add(0x2, data);                //给数组第三个字符赋值
bytOutArr.Add(0x3, data);                //给数组第四个字符赋值
bytOutArr.Add(0x4, data);                //给数组第五个字符赋值
COleVariant varOut;
varOut=COleVariant(bytOutArr);           //将数据转换为变体数据类型
m_comm.SetOutput (varOut);               //发送数据

```

5. 接收二进制数据

当需要接收大量的数据时,最好采用事件驱动方式进行编程。其具体步骤如下。

(1) 响应 **OnComm** 事件。在对话框资源编程器中,双击对话框模板上的 **MSCOMM** 控件,在弹出的对话框中填入所希望的事件响应函数名,此处将其命名为 **OnCommMscomm1()**。

(2) 在事件响应函数中接收和处理数据。接收来的数据为变体数据,所以需要做一些处理,具体代码如下所示:

```

void CMyCOMMDlg:: OnCommMscomm1()

```

```

{
    COleVariant varRcv;
    CByteArray byt;
    int i;
    long num;
    switch (m_comm.GetCommEvent())
    {
        case 1:                                //数据发送事件
            break;
        case 2:                                //数据接收事件
            varRcv=m_comm.GetInput();
            varRcv.ChangeType (VT_ARRAY |VT_UI1);
            BYTE HUGE *pbstr;
            HRESULT hr;
            hr=SafeArrayAccessData (varRcv.parray, (void HUGE *FAR*)&pbstr);
                                                    //获取安全数组指针

            if (FAILED (hr))
            {
                AfxMessageBox("获取数组指针失败！");
                break;
            }
            num=0;
            hr=SafeArrayGetUBound (varRcv.parray, 1, &num);        //获取数组上界
            if (FAILED (hr))
            {
                AfxMessageBox("获取数组上界失败！");
                break;
            }
            for(i=0;i<NUM;i++)
                byt.Add(pbstr [i]);
            SafeArrayUnaccessData (varRcv.parray);
                                                    //此时数据已保存在二进制数组 byt 中，可根据需要进行相关处理

            break;
        default:
            break;
    }
}

```

以上代码中的处理部分可以做成一个单独的函数，然后在此处调用即可。经过以上代码的处理，接收来的数据已存放在二进制数组 `byt` 中，可以根据自己的需要对其进行相关处理，

如保存和显示等。

习题和思考题

6-1 并行数据通信与串行数据通信各有什么特点，分别适用于什么场合？

6-2 串行异步通信的数据帧格式是怎样的？这种通信方式的主要优缺点是什么？

6-3 8031 的串行口工作在方式 2，编一段从机向主机传送 16 字节数据和校验和的程序，传送前发送联络信号。

6-4 设置 8031 串行口为方式 3，通信波特率为 2400 波特/秒，第 9 位数据用作奇偶校验位。在这种情况下，如何编写双工通信的程序？设数据交换采用中断方式，写出有关的程序。

6-5 用 VC 编写一个基于 MSCOMM 控件的程序，实现简单的 PC 与单片机的通信。

第7章 C51 单片机人机交互的 C 编程

重点与难点

本章重点是掌握键盘和数码显示、可编程键盘/显示接口芯片 8279 的原理、LCD 显示模块的原理以及它们的编程实现。

基本要求

1. 键盘和数码显示的原理及 C 程序编程
2. 可编程键盘/显示接口芯片 8279 的原理及 C 程序编程
3. 字符型 LCD 显示模块的原理及 C 程序编程
4. 点阵型 LCD 显示模块的原理及 C 程序编程

本章内容

1. 键盘接口技术
2. LED 数码显示技术
3. 可编程键盘/显示接口芯片 8279
4. 字符型 LCD 显示模块
5. 点阵型 LCD 显示模块

7.1 键盘接口技术

键盘是单片机应用系统最常用的输入设备，操作人员可以通过键盘向单片机系统输入指令、地址和数据，实现简单的人机通信。

7.1.1 独立式键盘与工作原理

键盘的结构形式有两种：独立式键盘和行列式键盘。

独立式按键是各按键互相独立地接通一条输入数据线，各按键的状态互不影响，如图 7-1 所示。这是最简单的键盘结构，该电路采用了中断方式读取键值。由于 C51 系列单片机 P1 口在内部已经有上拉电阻，根据使用经验图 7-1 中的 3 个外部上拉电阻可省掉。

当没有键按下时，与之相连的输入口线为 1（高电平），与门输出为高电平。当任何一个键按下时，与之相连的输入口线被置 0（低电平），与门输出由高变低，产生外中断条件，在

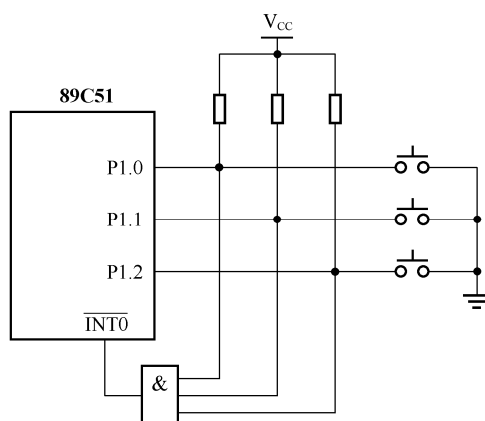


图 7-1 独立式按键与单片机的接口

独立式按键的 C 语言程序如下所示:

```
#include<reg51.h>
#include<absacc.h>
#define uchar unsigned char
#define TRUE 1
#define FALSE 0
extern void delay_10ms(void);    /*延时 10ms 函数*/
bit key_flag;
uchar key_value;
void main(void)
{
    IE=0x81;                    /*中断总控制位 EA、外中断 0 开中断*/
    IP=0x01;                    /*外中断 0 为高中断优先级*/
    key_flag=0;                 /*设置中断标志为 0*/
    do
    {
        if(key_flag)            /*如果按键有效*/
        {
            switch(key_value)    /*根据按键分支*/
            {
                casel:
                .....           /*处理 P1.0 口按键*/
                break;
                case2:
            }
        }
    } while(1);
}
```

```

        .....          /*处理 P1.1 口按键*/
        break;
        case4:
        .....          /*处理 P1.2 口按键*/
        break;
        default:
        break;          /*无效按键，如多个键同时按下*/
    }
    key_flag=0;          /*清按键标志*/
}
} while(TRUE);
}

/******外中断 0 处理程序******/
void int0() interrupt 0
{
    uchar reread_key;
    IE=0x80;             /*屏蔽  $\overline{\text{INT0}}$  中断*/
    key_flag=0;           /*设置中断标志为 0*/
    P1=0xff;             /*锁存器各口置 1*/
    key_value=P1&0x07;    /*读取外部中断源输入，并屏蔽高 5 位*/
    delay_10ms();         /*延时 10ms*/
    reread_key=P1&0x07;   /*再次读取外部中断源输入，并屏蔽高 5 位*/
    if(key_value==reread_key)
    {
        key_flag=1;       /*设置中断标志为 1*/
    }
    IE=0x81;             /*打开  $\overline{\text{INT0}}$  中断*/
}

```

独立式键盘结构的优点是电路简单；缺点是当键数较多时，需要占用较多的 I/O 线。

7.1.2 行列式键盘与工作原理

为了减少键盘与单片机接口时所占用 I/O 口线的数目，在键数较多时，通常都将键盘排列成行列矩阵式。每一水平线（行线）与垂直线（列线）的交叉处不相通，是通过一个按键连通的。利用这种行列矩阵结构只需 N 个行线和 M 个列线即可组成 $M \times N$ 个按键的键盘。图 7-2 所示是 4×4 （16 键）行列式键盘电路。由于 C51 单片机 P1 口在内部已经有上拉电阻，根据使用经验外部上拉电阻可以省掉。

在这种行列矩阵式非编码键盘的单片机系统中，对键的识别通常采用两步扫描判别法。下面以图 7-2 所示的 4×4 键盘为例，说明两步扫描判别法识别是哪一个键被按下的工作过程，

其工作过程如图 7-3 所示。

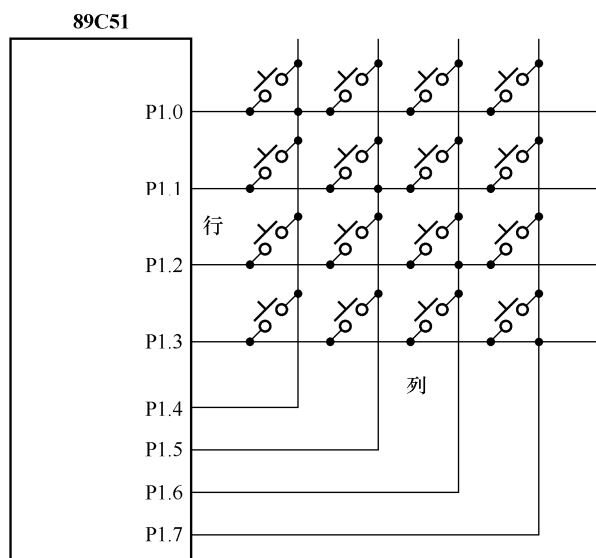


图 7-2 4×4 (16 键) 行列式键盘电路

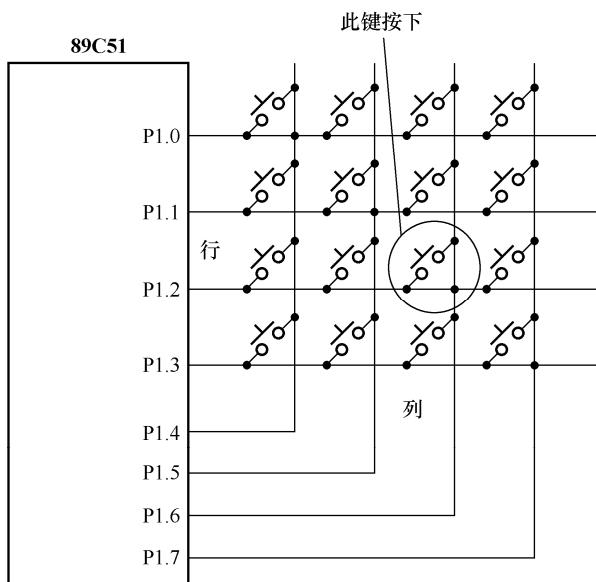


图 7-3 两步扫描判别法

首先判别按键所在的行，由单片机 P1 口向键盘送（输出）列扫描字，然后读入（输入）行线状态来判断。其方法是：向 P1 口输出 0FH，即列线（图中垂直线）输出全 0，行线（图中平行线）输出全 1，然后将 P1 口低 4 位（即行线）的电平状态读入一临时变量 x_temp 中。如果有按键按下，总会有一根行线被拉至低电平，从而使行输入不全为 1。在图 7-3 中，对应 P1.2 为低，即 x_temp=0x0b。

然后判别按键所在的列，由单片机 P1 口向键盘送（输出）行扫描字，然后读入（输入）

列线状态来判断。其方法是：向 P1 口输出 0FH，即行线输出全 0，列线输出全 1，然后将 P1 口高 4 位（即列线）的电平状态读入另一临时变量 y_temp 口中。如果有按键按下，总会有一根列线被拉至低电平，从而使列输入不全为 1。在图 7-3 中，对应 P1.6 为低，即 y_temp=0xb0。

最后将行和列的状态相或得到 0xbb，再把该值取反得到该位置键值为 0x44，对应的二进制数为 01000100B。如表 7-1 所示，该键值对应第 3 行第 3 列的按键。

表 7-1 键值与行判的对应关系

列				行			
4	3	2	1	4	3	2	1
0	1	0	0	0	1	0	0

同理求出上述 16 个位置的键值如表 7-2 所示。这种键盘的键值表示方式分散度大且不等距，还需进一步的程序处理，以依次排列键值。

表 7-2 行列式键盘键值

行 \ 列	1	2	3	4
1	11	21	41	81
2	12	22	42	82
3	14	24	44	84
4	18	28	48	88

7.1.3 键盘扫描的控制程序

单片机对按键的控制通常有以下三种方式：

- (1) 程序控制扫描方式，即利用程序连续地对键盘进行扫描；
- (2) 定时扫描方式，即单片机定时地对键盘进行扫描；
- (3) 中断扫描方式，即键的按下引起中断后，单片机对键盘进行扫描。

下面以图 7-2 所示的 4×4 行列式键盘为例，介绍程序控制扫描方式的工作过程和扫描子程序。程序控制扫描工作过程包括以下几项工作：

- (1) 查询是否有键按下；
- (2) 查询按下键所在的行列位置；
- (3) 对所得到的行号和列号译码得到键值；
- (4) 键的抖动处理。

【例 7-2】按照图 7-3 所示的电路，使用两步扫描判别法，编写相应的键扫描程序。有键按下，扫描函数返回值为键值，若无键按下，返回值为 0xff。

具体的键扫描 C 语言程序如下所示：

```
#include<reg51.h>
#include<absacc.h>
```

```

#define uchar unsigned char
#define uint unsigned int
#define TRUE 1
#define FALSE 0
uchar code KeyValue[16]={0x11, 0x21, 0x41, 0x81, 0x12, 0x22, 0x42 0x82,
0xl4, 0x24, 0x44, 0x84, 0x18, 0x28, 0x48, 0x88};
extern void delay 10ms(void);          /*延时 10ms 函数*/
uchar keyscan(void);                  /*扫描按键函数*/
void main(void)
{
    key_flag=0;                        /*设置按键标志为 0*/
    d0
    {
        if(key_flag)
        {                             /*如果按键有效*/
            switch(key_value)
            {                           /*根据按键分支*/
                case KeyValue[0]:
                .....                  /*处理 KeyValue[0]按键*/
                break;
                case KeyValue[1]:
                .....                  /*处理 KeyValue[1]按键*/
                break;
                .....
                case KeyValue[15]:
                .....                  /*处理 KeyValue[15]按键*/
                break;
                default:
                break;                 /*p 无效按键*/
            }
            key_flag=0;                /*清按键标志*/
        }
    }while(TRUE);
}
/*****扫描按键函数*****/
uchar keyscan(void)                   /*扫描按键函数*/
{
    uchar readkey, rereadkey;
    uchar x_temp, y_temp;

```

```

key_flag=0;                                /*清按键标志*/
x_temp=(P1=0x0f)&0x0f;
if(x_temp==0x0f) return(0xff);            /*无按键，退出*/
y_temp=(P1=0xf0)&0xf0;
readkey=x_temp|y_temp;
delay10ms();                              /*延时 10ms 后测再按键*/
x_temp=(P1=0x0f)&0x0f;
if(x_temp==0x0f) return(0xff);
y_temp=(P1=0xf0)&0xf0;
rereadkey=x_temp|y_temp;
if(readkey==rereadkey)
{
    key_flag=1;                            /*置按键标志*/
    return(readkey);
}
}

```

7.2 LED 数码显示技术

显示器常作为单片机系统中最简单的输出设备，用以显示单片机系统的运行结果与运行状态等。常用的显示器主要有 LED 数码显示器、LCD 液晶显示器和 CRT 显示器。在单片机系统中，通常用 LED 数码显示器显示各种数字或符号。由于它具有显示清晰、亮度高、使用电压低、寿命长的特点，因此使用非常广泛。本节以 LED 为例，介绍其结构、工作原理以及与单片机的接口技术。

7.2.1 LED 显示器的结构与原理

LED 显示器是由发光二极管显示字段的显示器件，也可称为数码管。单片机系统中通常使用 8 段 LED 数码显示器，其外形及引脚如图 7-4 (a) 所示，由图可见 8 段 LED 显示器由 8 个发光二极管组成。其中，7 个长条形的发光二极管排列成“日”字形，另一个圆点形的发光二极管在显示器的右下角作为显示小数点用。通过不同的组合可用来显示各种数字，包括 A~F 在内的部分英文字母和小数点“.”等字样。

LED 显示器有两种不同的形式：一种是 8 个发光二极管的阳极都连在一起的，称为共阳极 LED 显示器；另一种是 8 个发光二极管的阴极都连在一起的，称为共阴极 LED 显示器。共阳极和共阴极 LED 显示器如图 7-4 (b) 所示。

共阴和共阳结构的 LED 显示器各笔画段名和安排位置是相同的，当二极管导通时，相应的笔画段发亮，由发亮的笔画段组合从而显示各种字符。8 个笔画段 dpgfedcba 对应于 1B (8 位) 的 D7, D6, D5, D4, D3, D2, D1, D0，于是用 8 位二进制码就可以表示欲显示字符

的字形代码。例如，对于其阴极 LED 显示器，当公共阴极接地（为零电平），而阳极 dp g f e d c b a 各段为 01110011 时，显示器显示“P”字符，即对于共阴极 LED 显示器，“P”字符的字形码是 0x73。如果是共阳极 LED 显示器，公共阳极接高电平，显示“P”字符的字形代码应为 10001100 (0x8C)。这里必须注意的是，很多产品为方便接线，常不按规则的方法去对应字段与位的关系，这时字形码就必须根据接线自行设计了。

LED 显示器的显示方法有静态显示与动态显示两种，下面分别予以介绍。

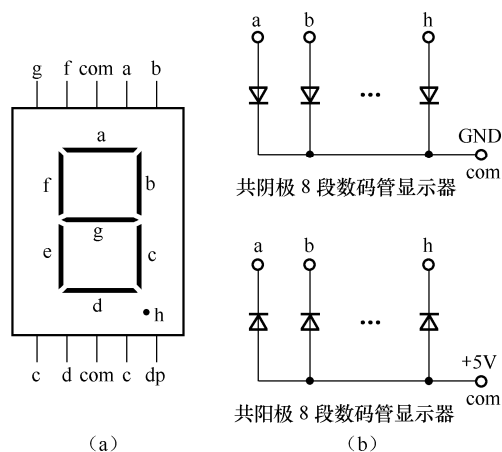


图 7-4 8 段 LED 数码显示器

7.2.2 LED 静态显示接口

数码管工作在静态显示方式时，共阴极（共阳极）的公共端 com 连接在一起接地（电源）。每位的段选线与一个 8 位并行口相连。只要在该位的段选线上保持段选码电平，该位就能保持相应的显示字符。这里的 8 位并行口可以直接采用并行 I/O 接口片（例如 89C51 的 P1 端口、8155 和 8255 的 I/O 端口等），也可以采用串行输入/并行输出的移位寄存器。

考虑到若采用并行 I/O 接口，占用 I/O 资源较多，因而静态显示方式常采用串行接口方式，外接 8 位移位寄存器 74HC164 构成显示电路。图 7-5 所示是通过串行口扩展 8 位 LED 显示器静态驱动电路，在 TXD (P3.1) 运行时钟信号，将显示数据由 RXD (P3.0) 口串行输出，串行口工作在移位寄存器方式（方式 0）。

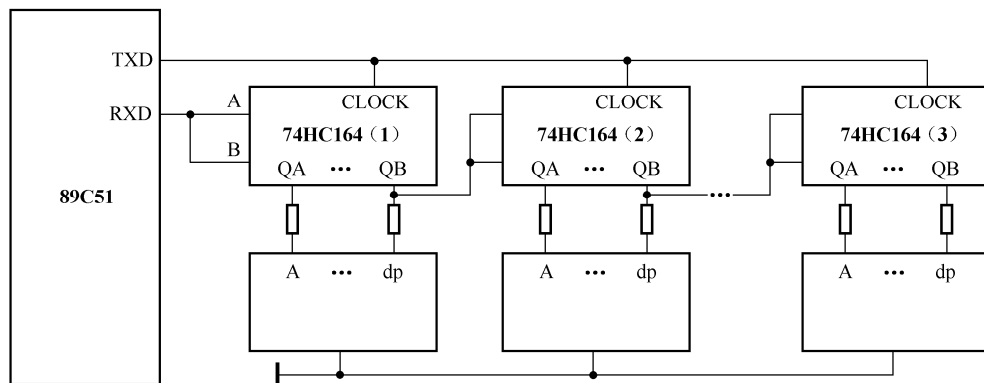


图 7-5 串行扩展 8 位 LED 显示器静态驱动电路

图 7-5 所示的电路中使用的是共阴极数码管，因而各数码管的公共极 com 端接地，要显示某字段，则相应的移位寄存器 74HC164 的输出线必须是高电平。

显然，要显示某字符，首先要把这个字符转换成相应的字形码，然后再通过串行口发送到 74HC164，最后 74HC164 把串行口收到的数变为并行输出加到数码管上。

表 7-3 共阴极字形码表

显 示 字 符	字 形 码	显 示 字 符	字 形 码
0	0x3f	A	0x77
1	0x06	B	0x7c
2	0x5b	C	0x39
3	0x4f	D	0x5e
4	0x66	E	0x79
5	0x6d	F	0x71
6	0x7d	P	0x73
7	0x07	U	0x3e
8	0x7f	8	0xff
9	0x6f	灭	0x00

这里先建立一个字形码表，以十六进制数的次序存放它们的相应字形码，共阴极字形码表如表 7-3 所示。例如要显示字符 6，查表可知 6 的字形码为 0x7d，把 0x7d 送到 8 位移位寄存器 74HC164 即可。显然，要显示字符 0~9、A~F，其高 4 位为全 0，而低 4 位为十六进制数。如果要显示的数高半字节不是 0，则要通过程序加以变换。

【例 7-3】按照图 7-5 所示的显示电路编写显示驱动程序。

```
void display(void) /*显示 0, 1, ..., 7*/
{
    uchar code LEDValue[8]={0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07};
    uchar i;
    TI=0;
    for(i=0; i<8, i++)
    { /*8 位数码管依次显示 0, 1, ..., 7*/
        SBUF=LEDValue[7-i];
        while(TI==0)TI=0;
    }
}
```

这种静态 LED 显示法有着显示亮度大、软件较为简单的优点，但硬件上使用芯片较多，并且每个 LED 显示器需要一个驱动电路。

7.2.3 LED 动态扫描显示接口

LED 动态显示的基本做法在于分时轮流选通数码管的公共端，使得各数码管轮流导通，在选通相应 LED 后，即在显示字段上得到显示字形码。这种方式不但能提高数码管的发光效率，而且由于各个数码管的字段线是并联使用的，从而大大简化了硬件线路。

动态扫描显示接口是单片机系统中应用最为广泛的一种显示方式。其接口电路是把所有

显示器的 8 个笔画段 a~dp 同名端并联在起，而每一个显示器的公共极 com 是各自独立地受 I/O 线控制。CPU 向字段输出输出送出字形码时，所有显示器由于同名端并连接收到相同的字形码，但究竟是哪个显示器亮，则取决于 com 端，而这一端是由 I/O 控制的，所以就可以自行决定何时显示哪一位了。而所谓动态扫描是指采用分时的方法，轮流控制各个显示器的 com 端，使各个显示器轮流点亮。

在轮流点亮扫描过程中，每位显示器的点亮时间是极为短暂的（约 1ms），但由于人的视觉暂留现象及发光二极管的余辉效应，尽管实际上各位显示器并非同时点亮，但只要扫描的速度足够快，给人的印象就是一组稳定的显示数据，不会有闪烁感。

图 7-6 所示是一个典型的动态扫描 8 位 LED 显示接口电路。该电路由 74HC245 提供段 a~dp 的驱动，74LS145 提供位 com1~com8 的驱动。注意，89C51 的 P0.0~P0.7 每个口线上均有 1 个 10kΩ 的上拉电阻（图中未示出）。

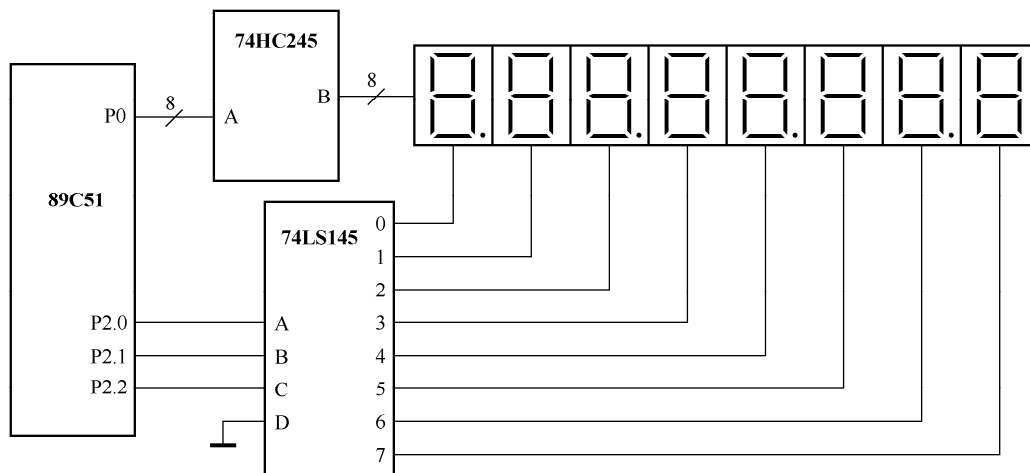


图 7-6 动态扫描 8 位 LED 显示接口电路

图中采用了共阴极的数码管。使用总线驱动器 74HC245 作为段驱动，由于 74LS145 具有 10 个 OC 门输出（图中用了 8 个），足够用来驱动 8 段显示器的公共极 com。

数码管是 8 段共阴极 LED 显示器，所以发光时字形驱动输出 1 有效，位驱动输出 0 有效。但注意位驱动是 74LS145 的译码输出，如果要显示第 5 位（数码管序号为 0~7）数码管，74LS145 的输入端应为 DCBA=0101。

【例 7-4】 按照图 7-6 所示的电路，编写在 8 个数码管依次显示 0, 1, ..., 7 的程序。

```
#include<reg51.h>

extern void delay_1ms(void);    /*延时 1ms 函数*/

void display(void)              /*在数码管 L 显示 0, 1, ..., 7 子程序*/
{
    uchar code LEDValue[8]={0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07};
    uchar i;
    for(i=0; i<8; i++)
```

```

{ /*8 位数码管依次显示 0, 1, ..., 7*/
  P0=LEDValue[i];
  P2=[i];
  delay_1ms();
}
}

```

采用此显示子程序，每调用一次，仅扫描一遍，因此要得到稳定的显示，必须不断地调用显示子程序。

7.3 可编程键盘/显示接口芯片 8279

8279 是可编程的键盘、显示接口芯片，它能自动完成键盘的扫描输入和 LED 扫描显示输出。键盘部分提供的扫描方式，可以和具有 64 个触点的键盘或传感器相连。它能自动清除按键抖动，并实现多键同时按下的保护。显示部分按扫描方式工作，可以连接 8 位或 16 位 LED 显示块。

使用 8279 可以代替单片机完成键盘和显示器的许多接口操作，从而大大地减轻了单片机的负担，因此，8279 在单片机领域中应用较为广泛。使用其键盘接口功能，可以实现单片机与键盘、乒乓开关、钮子开关等相连；使用其显示器接口功能，可以实现单片机与发光二极管显示（LED）、液晶显示器（LCD）、一般指示灯等相连。

7.3.1 8279 引脚和内部结构

1. 8279 的引脚结构

8279 的引脚和引线功能如图 7-7 所示。下面给出 8259 引脚的具体功能描述。

- (1) D0~D7：双向三态数据总线，用于和系统数据总线相连。
- (2) CLK：时钟输入端。
- (3) RESET：复位输入端。
- (4) $\overline{CS}$ ：片选。
- (5) A0：当 A0=1，I/O 口传输内容为命令/状态；当 A0=0，I/O 口传输内容为数据。
- (6) $\overline{RD}$ 、 $\overline{WR}$ ：8279 的读、写信号线。
- (7) IRQ：中断请求输出线，高电平有效。
- (8) SL0~SL3：扫描输出线。
- (9) RL0~RL7：回复线。
- (10) SHIFT：移位控制信号，用于上下档功能键切换。
- (11) CNTL/STB：控制/选通信号线，作为控制功能键。
- (12) OUTA0~OUTA3：A 组显示信号输出线。
- (13) OUTB0~OUTB3：B 组显示信号输出线。
- (14) BD：显示消隐输出线。

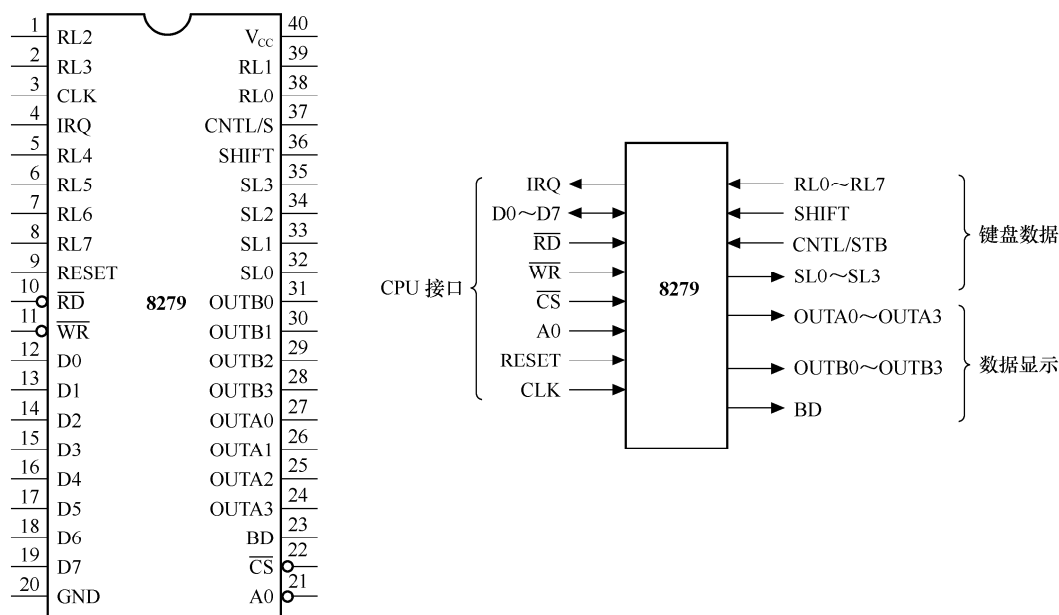


图 7-7 8279 的引脚和引线功能

2. 8279 内部结构

图 7-8 所示是 8279 内部结构框图，可见它主要分为六大部分，下面对它们进行介绍。

- (1) I/O 控制及数据缓冲器。
- (2) 控制与定时寄存器及定时控制。
- (3) 扫描计数器。

扫描计数器有两种工作方式：

① 编码工作方式。这时计数器作二进制计数，四位扫描线 $SL0 \sim SL3$ 输出四位计数状态，这种计数状态只有经外部译码后，方可作为键盘和显示的扫描码。

② 译码工作方式。这时扫描计数器的最低两位经内部译码后，由 $SL0 \sim SL3$ 扫描线输出，其输出可直接用做键盘和显示的扫描码。

- (4) 回复缓冲器和键盘消抖与控制。

从 $RL0 \sim RL7$ 输入的 8 个回复信号线作为键盘的检测输入线，由回复缓冲器缓冲并锁存。当某一键闭合时，该键的地址、附加的移位和控制状态、扫描码以及回复信号拼装成一字节键盘数据送入 8279 内的 FIFO（先进先出）RAM。

输入的键盘数据格式如下所示：

D7	D6	D5	D4	D3	D2	D1	D0
控制	移位	扫描			回复		

其中，控制（D7）和移位（D6）的状态由引脚外接的两个互相独立的附加开关（CNTL 和 SHIFT）决定；扫描（D5，D4，D3）是行（列）扫描编码，回复（D2，D1，D0）是回复线 $RL0 \sim RL7$ 的编码。扫描编码和回复编码反映了被按键的行、列位置。

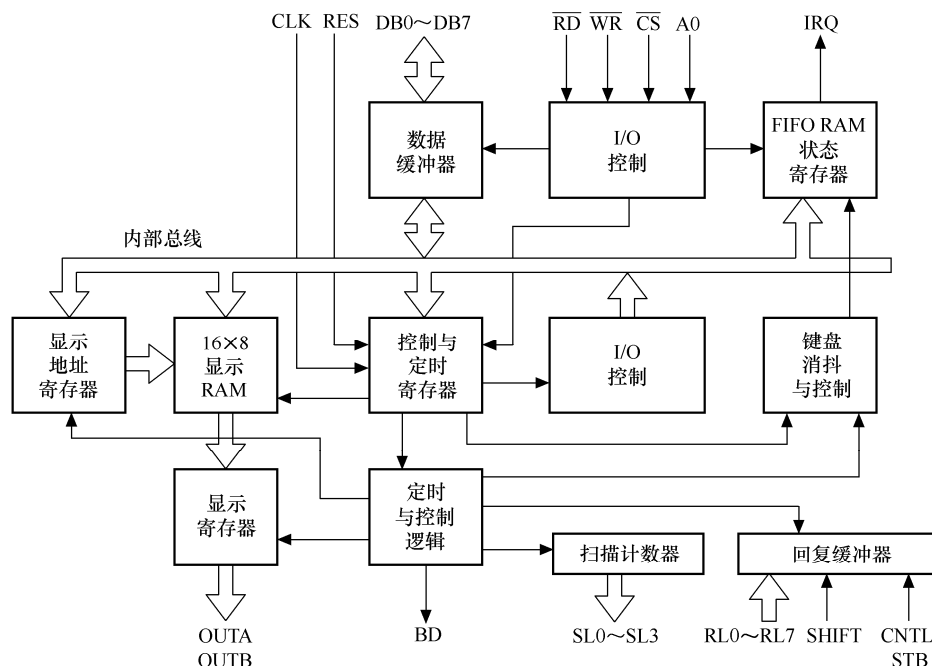


图 7-8 8279 结构框图

在传感器矩阵方式和选通输入方式中，回复线 RL0~RL7 的内容被直接送往相应的传感器 RAM（即 FIFO）。

（5）FIFO/传感器 RAM 及其状态寄存器。

FIFO/传感器 RAM 是一个双功能的 8×8 RAM。

在键盘或选通方式工作时，它是 FIFO 寄存器。状态寄存器寄存 FIFO 的当前工作状态。只要 FIFO 存储器不空，状态逻辑将置中断请求 IRQ=1。

在传感器矩阵方式时，这个 FIFO 存储器作传感器 RAM。当检测出传感器的变化时，中断请求信号 IRQ=1。

（6）显示 RAM 和显示地址寄存器。

显示 RAM 用来存储显示数据，容量为 16×8 位。在显示过程中，存储的显示数据轮流从显示寄存器输出显示。OUTA0~OUTA3 和 OUTB0~OUTB3 作外接显示器件的段选码端口，内部实现动态扫描显示，位选线由扫描线 SL0~SL3 提供。

显示地址寄存器用来寄存 CPU 读/写显示 RAM 的地址。

7.3.2 8279 的命令字和状态字

8279 有 8 个可编程的命令字，用来设定键盘（传感器）和 LED 显示器的工作方式以及实现对各种数据的读、写操作。

1. 键盘/显示方式设置命令字

键盘/显示方式设置命令字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	D	D	K	K	K

(1) D7、D6、D5：命令特征字，它们的 8 种组合编码对应了 8279 的 8 个命令。D7 D6 D5=000 为方式设置命令字。

(2) DD (D4 D3)：设定显示方式。00 为 8 个字符显示，左入口；01 为 16 个字符显示，左入口；10 为 8 个字符显示，右入口；11 为 16 个字符显示，右入口。

(3) KKK (D2 D1 D0)：用来设定七种键盘工作方式，000 为编码扫描键盘，双键锁定；001 为译码扫描键盘，双键锁定；010 为编码扫描键盘，N 键轮回；011 为译码扫描键盘，N 键轮回；100 为编码扫描传感器矩阵；101 为译码扫描传感器矩阵；110 为选通输入，编码显示扫描；111 为选通输入，译码显示扫描。

2. 编程时钟命令字

编程时钟命令字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	P	P	P	P	P

其中 PPPPP (D4 D3 D4 D1 D0) 用来设定对 CLK 端输入时钟的分频次数 N ， $N=2\sim 31$ 。

3. 读 FIFO/传感器 RAM 命令字

读 FIFO/传感器 RAM 命令字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	AI	X	A	A	A

(1) AAA (D2 D1 D0)：要读取的传感器 RAM 地址。

(2) AI (D4)：自动地址增量标志。每读出一字节后，AAA 位地址自动加 1。

4. 读显示 RAM 命令字

读显示 RAM 命令字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	AI	A	A	A	A

(1) AAAA (D3 D2 D1 D0)：用来寻址显示 RAM 的 16 个存储单元。

(2) AI (D4)：自动地址增量标志。

5. 写显示 RAM 命令字

写显示 RAM 命令字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	AI	A	A	A	A

(1) AAAA (D3 D2 D1 D0): 要写入的显示 RAM 地址。

(2) AI (D4): 自动地址增量标志。

6. 显示禁止写入/消隐命令字

显示禁止写入/消隐命令字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
1	0	1	X	IW/A	IW/B	BL/A	BL/B

(1) IW/A、IW/B (D3 D2): A、B 组显示 RAM 写入屏蔽位。其为 1 时显示 RAM 禁止写入。

(2) BL/A、BL/B (D1 D0): 消隐设置位。当 BL=1 时, 显示输出消隐; BL=0 时, 恢复显示。

7. 清除命令字

清除命令字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
1	1	0	CD	CD	CD	CF	CA

(1) CDCDCD (D4 D3 D2): 用于设定清除 RAM 方式。1 0 X 将显示 RAM 全部清“0”; 1 1 0 将显示 RAM 清成 20H; 1 1 1 将显示 RAM 全部清“1”; 0 X X 表示不清除。

(2) CF (D1): 用来置空 FIFO RAM。

(3) CA (D0): 总清特征值, 它兼有 CD 和 CF 的联合效用。

8. 结束中断/错误方式设置命令字

结束中断/错误方式设置命令字格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
1	1	1	E	X	X	X	X

其中 E 为 1 时有效。

除上述命令字以外, 8279 还有一个状态字, 用于反应键盘的 FIFO RAM 的工作状态, 其格式如下所示。

D7	D6	D5	D4	D3	D2	D1	D0
DU	S/E	O	U	F	N	N	N

- (1) DU (D7): 为 1 时表示显示无效。
- (2) S/E (D6): 为 1 时表示传感器的最后一个信号已进入或出现了多键同时按下的错误。
- (3) O、U (D5 D4): 数据超出和不足标志。
- (4) F (D3): FIFO RAM 中 8 个数据已满时, F=1。
- (5) NNN (D2 D1 D0): 指明 FIFO RAM 中的数据个数。

7.3.3 8279 和 C51 单片机的接口

如图 7-9 所示, C51 单片机与 8279、键盘、LED 显示器的接口电路。当有键按下时, 8279 可用中断方式通知 C51。可见, 接口电路实现的功能是: 当有键 0~FH 按下时, 完成键值获取, 并用 LED 输出显示键值。

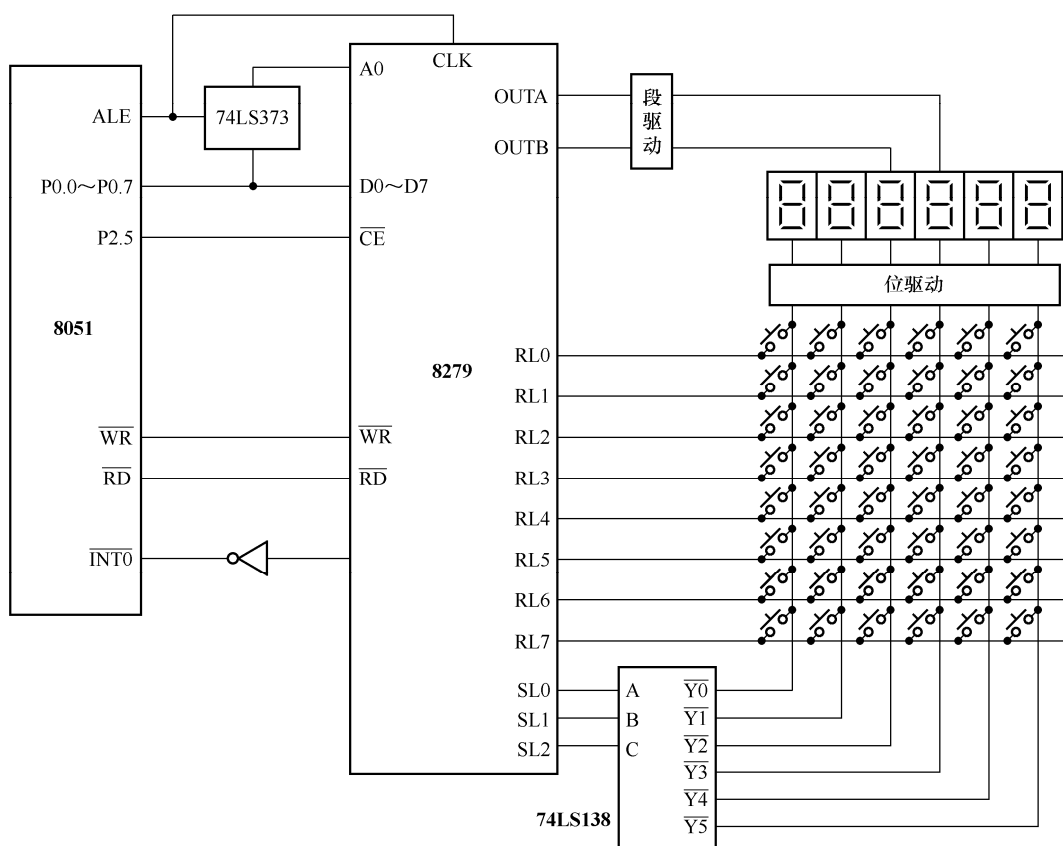


图 7-9 C51 单片机与 8279、键盘、LED 显示器的接口电路

根据接口电路中 8279 片选信号和 A0 的接法, 8279 的端口地址为: 0DFFE_H (数据口) 和 0DFFF_H (命令/状态口)。

【例 7-5】8279 显示和键盘控制。

主程序调用显示和键盘输出函数; 采用 diss 数组作缓冲区; table 为 0~FH 所对应的段码表。程序代码如下所示:

```

#include<reg51.h>
#include<absacc.h>
#define COM XBYTE[0xdfff]    /*命令/状态口*/
#define DAT XBYTE[0xdffe]    /*数据口*/
#define uchar unsigned char
uchar code table[]={0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d,
0x07, 0x7f, 0x6f, 0x77, 0x7c, 0x39, 0x5e, 0x79, 0x71};
uchar idata diss[8]={0, 1, 2, 3, 4, 5, 6, 7};
sbit clflag=ACC^7;
uchar keyin();
uchar deky();
void disp(uchar idata *d);
void main(void)
{
    uchar i;
    COM=0xd1;                /*总清除命令*/
    do{ACC=COM; }
    while(clflag==1);        /*等待清除结束*/
    COM=0x00;
    COM=0x2a;                /*键盘、显示方式，时钟分频*/
    while(1)
    {
        for(i=0; i<8; i++)
        {
            disp(diss);      /*显示缓冲区内容*/
            diss[i]=keyin();  /*键盘输入到显示缓冲*/
        }
    }
}
void disp(d)                /*显示函数*/
uchar idata *d;
{
    uchar i;
    COM=0x90;
    for(i=0; i<8; i++)
    {
        COM=i+0x80;
        DAT=table[*d];
        d++;
    }
}

```

```

    }
}
uchar keyin(void)          /*取键值的函数*/
{
    uchar i;
    while(deky()= =0);     /*无键按下时等待*/
    COM=0x40;              /*读 FIFO RAM 命令*/
    i=DAT;
    i=i&0x3f;              /*取键盘数据低 6 位*/
    return(i);             /*返回键值*/
}
uchar deky(void)           /*判 FIFO 有键按下的函数*/
{
    uchar k;
    k=COM;
    return(k&0x0f);        /*非 0 则有键按下*/
}

```

7.4 字符型 LCD 显示模块

液晶显示器以其低功耗、体积小、重量轻、超薄型等诸多其他显示器件所无法比拟的优点，广泛应用在袖珍式仪表和低功耗应用系统中。

7.4.1 字符型 LCD 的结构和引脚

字符型 LCD 是一种用 5×7 点阵图形来显示字符的液晶显示器，它主要包括 DDRAM、CGROM、CGRAM 等。

(1) DDRAM: 显示数据 RAM，用来寄存待显示的代码。

(2) CGROM: 字符发生器 ROM，它内部已经存储了 160 个不同的点阵字符图形，字符图形用于字符的显示。

(3) CGRAM: 字符发生器 RAM，它是 8 个允许用户自定义的字符图形 RAM。

(4) DDRAM 的地址: LCD 控制器的指令系统规定，在送待显示字符代码的指令前，先要送 DDRAM 的地址，实际上是待显示的字符显示位置。若 LCD 为双行字符显示，每行 40 个显示位置，第一行地址为 00H~27H；第二行地址为 40H~67H。双行显示的 DDRAM 地址与显示位置的对应关系如下所示：

显示位置		1	2	3	4	5	6	7	...	39	40
DDRAM 地址	第一行	00H	01H	02H	03H	04H	05H	06H	...	26H	27H
	第二行	40H	41H	42H	43H	44H	45H	46H	...	66H	67H

(5) 指令寄存器：用来接收 CPU 送来的指令码；也寄存 DDRAM 和 CGRAM 的地址。

(6) 数据寄存器：用来寄存 CPU 发来的字符代码数据。

(7) 状态标志位：LCD 控制器有一个忙信号标志位 BF，BF=1 时表示 LCD 正在进行内部操作，此时不接收外部命令。

(8) AC：地址计数器，AC 的内容是 DDRAM 或 CGRAM 的单元地址。当对 DDRAM 或 CGRAM 进行读写操作后，AC 自动加 1 或减 1。

(9) 光标/闪烁控制：可产生光标或使光标在显示位置处闪烁，显示位置为 AC 中的 DDRAM 地址。

通常，字符型 LCD 显示有 14 条引脚线，这 14 条线的定义是标准的下面对其功能进行说明。

(1) V_{SS} (1)：地。

(2) V_{DD} (2)：电源电压。

(3) V₀ (3)：对比调整电压。

(4) RS (4)：寄存器选择。RS=0 时，读状态寄存器或写命令寄存器；RS=1 时，读写数据。

(5) R/ $\overline{W}$ (5)：读写信号线。R/ $\overline{W}$ =1 时，读操作；R/ $\overline{W}$ =0 时，写操作。

(6) E (6)：显示板控制使能端。

(7) D0~D7 (7~14)：8 位双向三态 I/O 线。

7.4.2 显示板控制器的指令系统

字符型 LCD 显示板控制器有 11 条指令，它的读写操作、屏幕和光标的操作都是通过指令编程来实现的。LCD 控制器的 11 条指令如表 7-4 所示。

表 7-4 指令表

指 令	RS	R/ $\overline{W}$	D7	D6	D5	D4	D3	D2	D1	D0
清显示	0	0	0	0	0	0	0	0	0	0
光标返回	0	0	0	0	0	0	0	0	1	*
置输入模式	0	0	0	0	0	0	0	1	I/D	S
显示开/关控制	0	0	0	0	0	0	1	D	C	B
光标或字符移位	0	0	0	0	0	1	S/C	R/L	*	*
置功能	0	0	0	0	1	DL	N	F	*	*
置字符发生存储器地址	0	0	0	1	字符发生存储器地址（AGG）					
置数据存储器地址	0	0	1	显示数据存储器地址（ADD）						
读忙标志或地址	0	1	BF	计数器地址（AC）						
写数到 CGRAM 或 DDRAM	1	0	要写的数							
从 CGRAM 或 DDRAM 读数	1	1	读出的数据							

下面对指令表中 D0~D7 位所使用字符进行说明。

- (1) I/D=1/0: 增量/减量。
- (2) S=1: 全显示屏移动。
- (3) S/C=1/0: 显示屏移动/光标移动。
- (4) R/L=1/0: 右移/左移。
- (5) DL=1/0: 8 位/4 位。
- (6) N=1/0: 2 行/1 行。
- (7) F=1/0: 5×10 点阵/5×7 点阵。
- (8) BF=1/0: 内部操作正在进行/允许指令操作。
- (9) *: 无关项。

接下来逐条解释各指令的功能。

- (1) 指令 1: 清显示, 光标复位到地址 00H 位置。
- (2) 指令 2: 光标返回, 光标返回到地址 00H。
- (3) 指令 3: 读/写方式下的光标和显示模式设置命令。

① I/D: 表示地址计数器的变化方向, 即光标移动的方向。I/D=1 表示 AC 自动加 1, 光标右移一字符位; I/D=0 表示 AC 自动减 1, 光标左移一字符位。

② S: 显示屏上画面向左或向右全部平移一个字符位。S=0 表示无效; S=1 表示有效。S=1 和 I/D=1 表示显示画面左移; S=1 和 I/D=0 表示显示画面右移。

- (4) 指令 4: 显示开/关控制, 控制显示、光标和光标闪烁的开关。

- ① D: 当 D=0 时显示关闭, DDRAM 中数据保持不变。
- ② C: 当 C=1 时显示光标。
- ③ B: 当 B=1 时光标闪烁。

(5) 指令 5: 光标或显示移位, DDRAM 中内容不改变。S/C=1 时, 移动显示; S/C=0 时, 移动光标。R/L=1 时, 为右移; R/L=0 时, 为左移。

- (6) 指令 6: 功能设置命令。

- ① DL=1 时, 内部总线为 4 位宽度 DL=0 时, 内部总线为 8 位宽度。
- ② N=0 时, 单行显示; N=1 时, 双行显示。
- ③ F=0 时, 为显示字形 5×7 点阵; F=1 时, 为 5×10 点阵。

- (7) 指令 7: CGRAM 地址设置。
- (8) 指令 8: DDRAM 地址设置。
- (9) 指令 9: 读状态标志和 AC 中地址。
- (10) 指令 10: 写数据。
- (11) 指令 11: 读数据。

7.4.3 LCD 显示板与单片机的接口和编程

图 7-10 所示是 C51 单片机与字符型 LCD 显示板的接口。其中, 74HC245 作为总线驱动器; A0 连接到寄存器选择信号 RS; A0A1A2 经 3-8 译码器 74LS138 产生设备选择信号 E; P2.7 作为 74LS138 的选通信号。通过分析可知, 图中译码电路产生的显示板命令/状态寄存器地址为 7FF0H, 数据寄存器地址为 7FF1H。

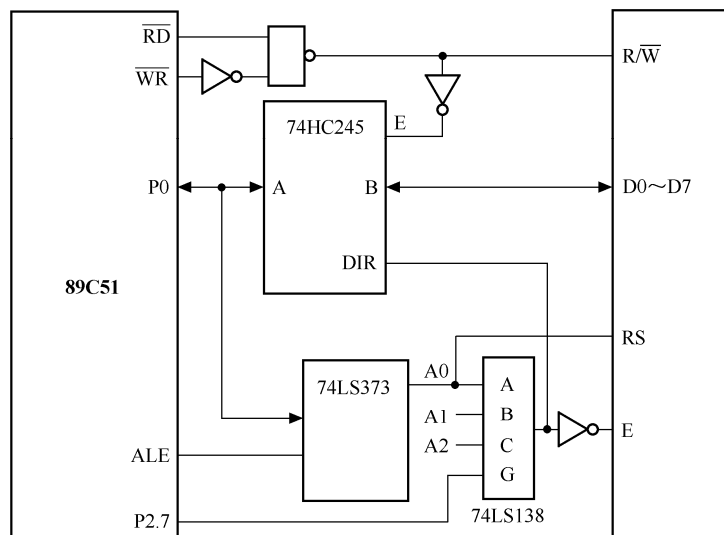


图 7-10 C51 单片机与字符型 LCD 显示板的接口

【例 7-6】 用字符型的 LCD 显示模块来显示字符。

下面是在双行显示板的第一行第三个位置显示 4 个字符“GOOD”的程序，程序包括初始化函数 init_lcd、显示字符函数 d_char 和显示字符串函数 display。

```
#include<reg51.h>
#include<absacc.h>
#define uchar unsigned char
#define LCDCOM XBYTE[0x7ff0] /*命令/状态寄存器地址*/
#define LCDDATA XBYTE[0x7ff1] /*数据寄存器地址*/
sbit bflag=ACC^7;
uchar idata s_GOOD[5]={0x47, 0x4f, 0x4f, 0x44, 0x00}; /*GOOD*/
void wait(void)
{
    do{ACC=LCDCOM; /*查询 BF 标志*/
    }while(bflag==1);
}
void init_lcd(void)
{
    LCDCOM=0x38; /*置功能：2 行，5×7 字符*/
    wait();
    LCDCOM=0x06; /*置输入模式：地址增量，显示屏不移动*/
    wait();
    LCDCOM=0x0f; /*显示打开，显示光标，光标闪烁*/
    wait();
    LCDCOM=0x01; /*清显示*/
}
```

```

    wait();
}
void d_char(uchar a)                /*显示字符函数*/
{
    LCDDATA=a;                      /*输出到数据寄存器*/
    wait();
}
void display(x, s)                  /*显示字符串函数*/
uchar x;                           /*显示位置*/
uchar idata *s;                     /*显示的字符串*/
{
    uchar i=0;
    LCDCOM=x;                       /*置显示位置地址*/
    while(s[i]!=0x00)
    {
        d_char(s[i]);
        i++;
    }
}
void main(void)                     /*在第 2 位置显示字符串“GOOD”*/
{
    init_lcd();
    display(2, s_GOOD);
}

```

7.5 点阵型 LCD 显示模块

本节以内藏 HD61830 控制器的液晶模块 MGLS-240128 为例，来说明点阵型 LCD 显示模块的应用。

7.5.1 HD61830 的特点和引脚

1. HD61830 的特点

(1) HD61830 是点阵式液晶图像显示控制器，可与 M6800 系列相适配的 MPU 直接接口。

(2) 具有专用指令集，可完成文本显示或图形显示的功能设置，以及实现画面卷动、光标、闪烁、位操作等功能。

- (3) HD61830 可管理 64KB 显示 RAM，其中图形方式为 64KB；字符方式为 4KB。
- (4) 内部字符发生器 CGROM 共有 192 种字符，其中 5×7 字体 160 种；5×11 字体有 32 种。HD61830 还可以外接字符发生器，使字符量达到 256 种。
- (5) HD61830 可以静态方式显示，能以最大为 1/128 占空比的动态方式显示。

2. HD61830 的受控引脚

- (1) D7~D0：三态数据总线。
 - (2) $\overline{CS}$ 输入片选信号，低电平有效。
 - (3) E：输入使能信号，高电平有效。
 - (4) R/ $\overline{W}$ ：输入读、写选择信号。R/ $\overline{W}$ =1 表示 MPU 读取 HD61830 的信息；R/ $\overline{W}$ =0 表示 MPU 向 HD61830 写入数据。
 - (5) RS：输入寄存器选择信号。RS=1 表示指令寄存器及忙标志位；RS=0 表示数据寄存器。
 - (6) RES：输入复位信号，低电平有效。
- HD61830 的工作时序如图 7-11 所示。

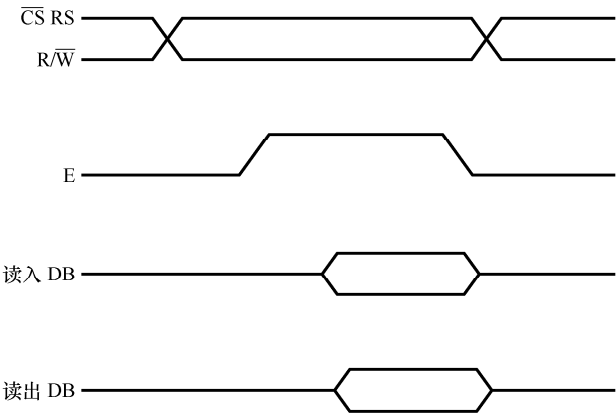


图 7-11 HD61830 与 MPU 的时序图

从时序上分析，MPU 与 HD61830 联络的关键信号是使能信号 E，使能信号 E 在读、写操作过程中的作用如表 7-5 所示，读写信号 R/ $\overline{W}$ 可认为是数据总线上数据流方向的控制信号。

表 7-5 HD61830 的信号功能			
RS	R/ $\overline{W}$	E	功 能
0	0	下降沿	写数据或指令参数
0	1		读数据
1	0	下降沿	写指令代码
1	1		读忙标志位

7.5.2 HD61830 指令集

HD61830 的指令结构是一致的，一条指令由一个字节的指令代码与一个字节的指令参数

组成。

1. 方式控制指令，代码为 00H

向指令寄存器写入 00H 后紧接着向数据存储器写入参数，即可定义显示方式。方式控制参数格式如下所示：

0	0	D5	D4	D3	D2	D1	D0
---	---	----	----	----	----	----	----

- (1) D0：字符发生器选择。0 时为 CGROM，1 时为 EXCGROM。
- (2) D1：显示方式选择。0 时为文本方式，1 时为图形方式。
- (3) D2、D3 组合实现功能如下：

D3	D2	功 能
0	0	光标禁止
0	1	启用光标
1	0	光标禁止，字符闪烁
1	1	光标闪烁

- (4) D4：工作方式选择。0 时为从方式，1 时为主方式。
- (5) D5：显示状态选择。0 时为禁止显示，1 时为启用显示。

2. 字体设置指令，代码为 01H

该指令设置文本方式下字符的点阵大小，指令参数格式如下所示：

		VP-1				HP-1	
--	--	------	--	--	--	------	--

- (1) VP：字符点阵行数，取值范围 1~16。
- (2) HP：字符点阵列数，图形方式表示一字节显示数据的有效位数，HP 的取值范围为 6、7、8。

3. 显示域设置，指令代码为 02H

该指令参数格式如下所示：

0				HN-1			
---	--	--	--	------	--	--	--

HN 为一行显示所占的字节数，其取值范围为 2~128 内的偶数值，由 HN 和 HP 可得显示屏有效显示列数 $N=HN \times HP$ 。

4. 帧设置，指令代码为 03H

该指令参数格式如下所示：

其中, NX 为显示时的帧扫描行数, 其倒数即为占空比。

文本方式下光标为一行点阵显示，该指令用来指明该行点阵在字符体中的第几行，指令参数格式如下所示：

其中, CP 表示光标在字符体中的行位置, CP 取值范围在 1~VP 之间。

7. SADH 设置, 指令代码为 09H。

SAD	SAD+1	...	SAD+HN-1
SAD+HN	SAD+HN+1	...	SAD+2HN-1
...	...	...	...
SAD+MHN	SAD+MHN+1	...	SAD+(M+1) HN-1

9. CACH 设置，指令代码为 0BH

10. 数据写指令，代码为 0CH

11. 数据读，指令代码为 0DH

12. 位清零, 指令代码为 0EH。

13. 位置 1，指令代码为 0FH。

以上两条指令的功能是将光标地址指针所指向的显示缓冲区单元中的字节某位清零或置 1，指令执行一次光标地址指针自动增 1。指令参数格式如下所示：

0	0	0	0	0		NB-1
---	---	---	---	---	--	------

其中，NB 为清零或置 1，取值 1~8，对应该字节的 LSB~MSB。

7.5.3 与内藏 HD61830 的液晶模块的接口和编程

内藏 HD61830 的液晶模块 MGLS-240128 有 18 个外出引线可供接口使用，其引脚顺序如下所示：

1	2	3	4	5	6	7~14	15	16	17	18
GND	V _{CC}	V ₀	RS	R/ $\overline{W}$	E	DB0~DB7	$\overline{CS}$	RST	LED+	LED-

其中，GND，V_{CC} 为地和+5V 电源；V₀ 为负向液晶驱动电源，对 MGLS-240128 来说 V₀ 的取值为-15V 左右；LED+和 LED-为接背景光时的电源。

图 7-12 是采用 8255 控制 MGLS-240128 模块的接口电路，8255 的地址为 8000H~8003H。

【例 7-7】 点阵型 LCD 显示模块显示英文字符串。

下面是显示字符串“WELCOME!”的程序，程序包括显示字符串函数 disstr、写指令函数 wcode 和写数据函数 wdata。

```
#include<reg51.h>
#include<absacc.h>
#define uchar unsigned char
#define uint unsigned int
#define PA XBYTE[0x8000]
#define PB XBYTE[0x8001]
#define PC XBYTE[0x8002]
#define COM XBYTE[0x8003]
#define DELAY 3
uchar idata welc[11]={0x20, 0x57, 0x45, 0x4c, 0x43,
0x4f, 0x4d, 0x45, 0x21, 0x20, 0x00}; /*"WELCOME! "*/
uchar idata sadl, sadh;
uchar idata addl, addh;
void wcode(uchar c);
void wdata(uchar d);
void disstr(uchar idata *str);
void main(void)
```

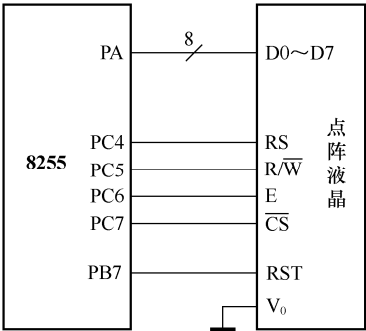


图 7-12 8255 与 MGLS-240128 的接口

```

{
    COM=0x81;
    PB=0x00;                                /*MGLS-240128 模块复位*/
    PB=0xf0;
    disstr(welc);                            /*显示字符串*/
    while(1);
}
void wcode(uchar c)                        /*写指令代码*/
{
    uchar i=DELAY;
    while(i)--;
    PC=0x9f; PA=c; PC=0xdf; PC=0x5f; PC=0x1f; PC=0x9f;
}
void wdata(uchar d)                       /*写指令参数*/
{
    uchar i=DELAY;
    while(i)--;
    PC=0x8f; PA=d; PC=0xcf; PC=0x4f; PC=0x0f; PC=0x8f;
}
void comd(x, y)                            /*写一条指令*/
uchar x, y;
{
    wcode(x);
    wdata(y);
}

void disstr(uchar idata *str)
{
    uchar i, j;
    comd(0x00, 0x3c);                      /*方式设置, 主方式显示, 光标闪烁, 文本方式,
                                           选用 CGROM*/

    comd(0x01, 0x77);                      /*字体设置, VP=8, HP=8, 8×8 字体*/
    comd(0x02, 0x1d);                      /*显示域设置, HN=30, 即一行显示 30 个字符*/
    comd(0x03, 0x7f);                      /*帧设置, NX=128 即占空比为 1/128*/
    comd(0x04, 0x07);                      /*光标设置, CP=8, 光标位于字符的最下端*/
    sadl=0x00;
    sadh=0x00;
    comd(0x08, sadl);
    comd(0x09, sadh);                      /*设置缓冲区起始地址*/
}

```



```
comd(0x0a, 0x00);
comd(0x0b, 0x00);
wcode(0x0c);
for(j=0; j<10; j++) wdata(0x20);    /*清屏*/
addl=0x00;
addh=0x00;
comd(0x0a, addl);
comd(0x0b, addh);                    /*设置光标地址指针*/
i=0;
wcode(0x0c);
while(str[i]!=0x00)
{
    wdata(str[i]);                    /*输出字符*/
    i++;
}
}
```

习题和思考题

- 7-1 七段 LED 显示器主要有哪几种显示方式？动态显示方式的原理是什么？有什么特点？
- 7-2 利用图 7-9 所示的接口电路编程。键盘的键值可自行确定，当有键按下时键值送 LED 显示，显示为右入方式，用中断方式进行按键处理。
- 7-3 与并行扩展相比，串行扩展有什么优缺点？
- 7-4 I²C 总线只有两根连线（数据线和时钟线），如何识别扩展器件的地址？
- 7-5 利用图 7-12 所示的接口电路，实现 LCD 显示功能。要求 LCD 单行显示，由左起第五位开始显示字符“HELLO!”。试编写初始化程序和显示程序。

第 8 章 Keil C51 上机指南

重点与难点

Keil C51 软件的配置和使用。

基本要求

1. Keil C51 软件安装
2. 项目工程及文件的建立和添加
3. 工程属性的配置，包括各属性标签的设置
4. 代码的编译及排错等
5. 代码的调试，断点的设置，观察窗口的使用等

本章内容

1. Keil C51 软件的安装
2. Keil C51 工程的建立
3. Keil C51 工程属性的配置
4. 代码的编译
5. 代码的调试

单片机开发中除必要的硬件外，同样离不开软件。单片机的开发软件也在不断发展，Keil 软件是目前最流行开发 MCS-51 系列单片机的软件，这从近年来各仿真机厂商纷纷宣布全面支持 Keil 即可看出。Keil 提供了包括 C 编译器、宏汇编、连接器、库管理和一个功能强大的仿真调试器等在内的完整开发方案，通过一个集成开发环境（uVision）将这些部分组合在一起。如果使用 C51 系列单片机，掌握这一软件的使用是十分必要的，即使不使用 C 语言而仅用汇编语言编程，其方便易用的集成环境、强大的软件仿真调试工具也会令用户事半功倍。

Keil C51 的 IDE（集成开发环境）主要有两个版本，一个是 UV1（Keil C51 V5.20 以下版本），一个是现在大家常用的 UV2 版本。本章以 Keil C51 7.0 版本及 UV2 版本为例介绍 Keil C51 编译软件的使用。

8.1 Keil C51 软件的安装

8.1.1 系统需求

下面是按照 Keil C51 UV2 版本的最低软件和硬件需求：

- (1) Pentium, Pentium-II 或以上版本的兼容处理器;
- (2) Windows95, Windows98, Windows NT 4.0, WinMe, Windows2003, WindowsXP 或以上版本的操作系统;
- (3) 最低 16MB RAM 内存;
- (4) 最低 20MB 的磁盘空间。

8.1.2 软件安装

运行安装软件包中的 setup.exe 文件, 会出现安装初始化窗口, 如图 8-1 所示。



图 8-1 安装初始化界面

接下来是版本选择界面, 如图 8-2 所示, 有两种选择, Eval Version (试用版) 和 Full Version (完整版), 默认的是 Eval 版, 如果您不是正版用户请选择 Eval 版, 否则选择 Full 版。本章以 Full 版为例加以说明。选择相应版本后, 将打开安装目录选择界面, 如图 8-3 所示。



图 8-2 版本选择界面

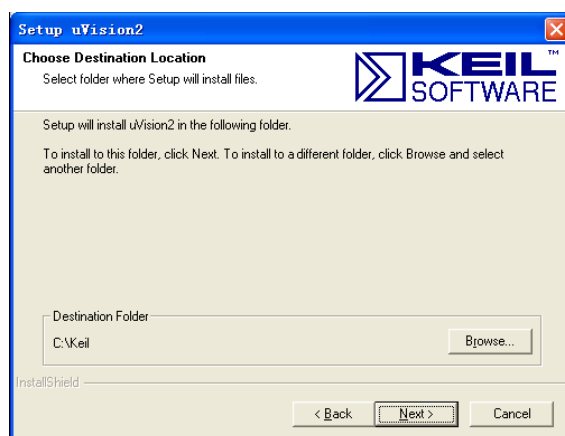


图 8-3 安装目录选择界面

默认安装目录是 C:\Keil，选择安装目录确认后单击“Next”按钮，打开注册信息界面如图 8-4 所示。

输入正版 SN 码及 First Name，Last Name。Company Name 可以任意填写，E-mail 可选填。确认后单击“Next”进行下一步。此时安装程序要求插入 Add-on 盘，如图 8-5 所示。

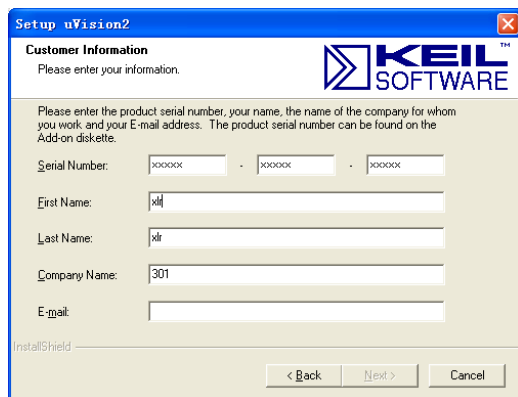


图 8-4 注册信息界面

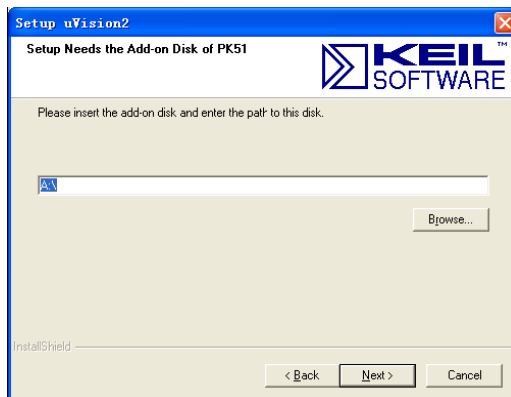


图 8-5 插入 Add-on 盘

插入软盘或者将路径指到包含这些内容的硬盘目录，确认后单击“Next”，安装程序开始正式安装，安装进度界面如图 8-6 所示。

安装完成后，安装程序将会询问用户是否进行网络注册，如图 8-7 所示。确认是否通过网络注册，建议取消，确认后单击“Next”，完成安装。

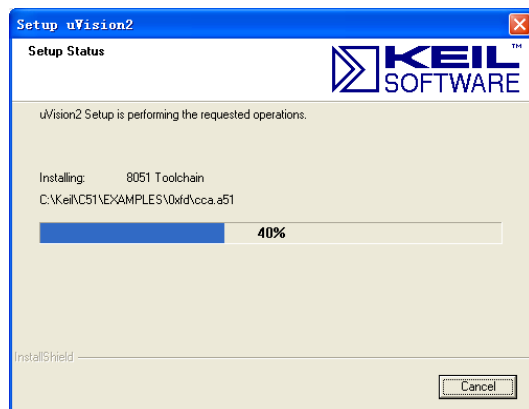


图 8-6 安装进度界面

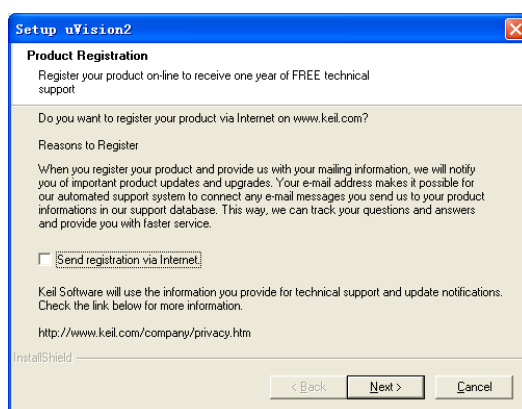


图 8-7 网络注册选择

8.2 Keil C51 工程的建立及设置

8.2.1 Keil 工程的建立

首先启动 Keil 软件的集成开发环境，可以从桌面上直接双击 uVision 的图标以启动该软件，还可以从“开始”菜单的“程序组”里找到 uVision 启动项。

UVision 启动后, 程序窗口的左边有一个工程管理窗口, 该窗口有 3 个标签, 分别是 Files, Regs 和 Books, 这三个标签页分别显示当前项目的文件结构、CPU 的寄存器及部份特殊功能寄存器的值 (调试时才出现) 和所选 CPU 的附加说明文件。如果是第一次启动 Keil, 那么这三个标签页全是空的, 如图 8-8 所示。

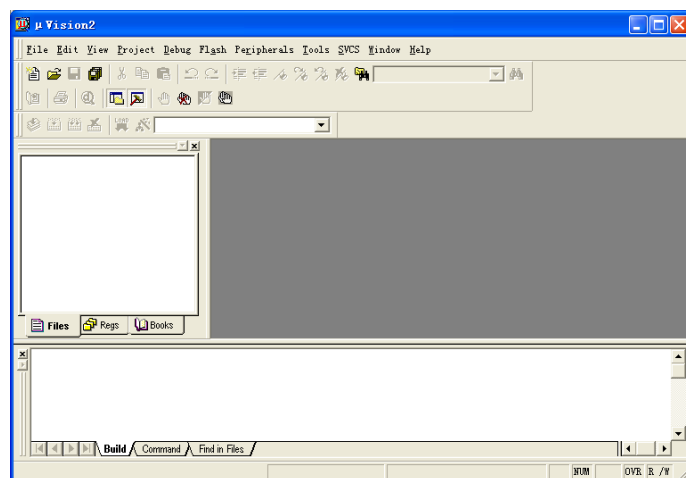


图 8-8 程序界面

keil c51 uVision2.0 的版本, 需要一个文件来存储工程的相关信息, 不管是汇编语言的, 还是 C 语言的, 只有一个文件, 还是有多个文件的程序都要有一个工程文件, 都需要一个工程文件才能进行编译和仿真。

点击菜单 project, 选择 new project, 如 8-9 所示。

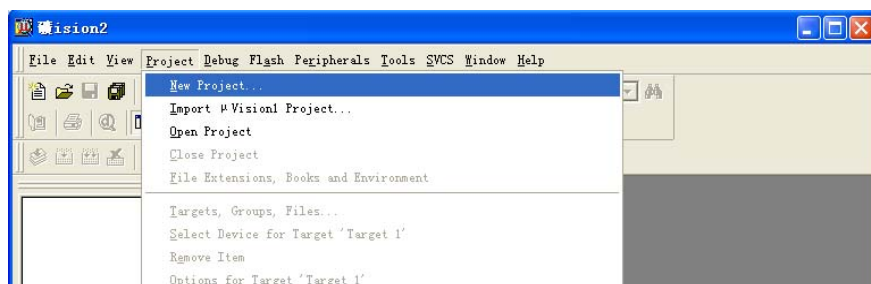


图 8-9 创建工程

然后选择要保存的路径, 如图 8-10 所示, 输入工程文件的名字, 如 pro, 然后单击“保存”。

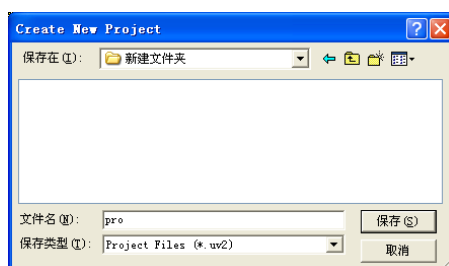


图 8-10 保存工程

这时会弹出一个要求用户选择单片机型号的对话框，如图 8-11 所示。用户可以选择所使用的单片机，如果该单片机型号在列表中找到，也可以找一款与目前所使用单片机相兼容的型号来代替。这里还是以目前使用较多的 Atmel 的 89C51 来说明。

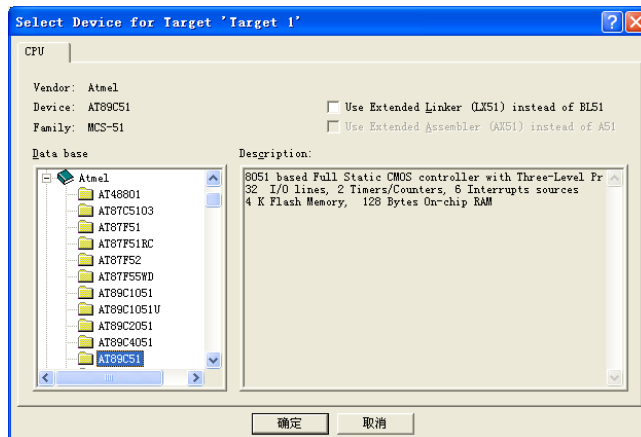


图 8-11 选择单片机型号

下面要为该工程添加所需的程序文件。

新建一个源程序文件，如图 8-12 所示建立一个 c 文件；如果已经有源程序文件，可以忽略这一步，直接添加源程序文件。

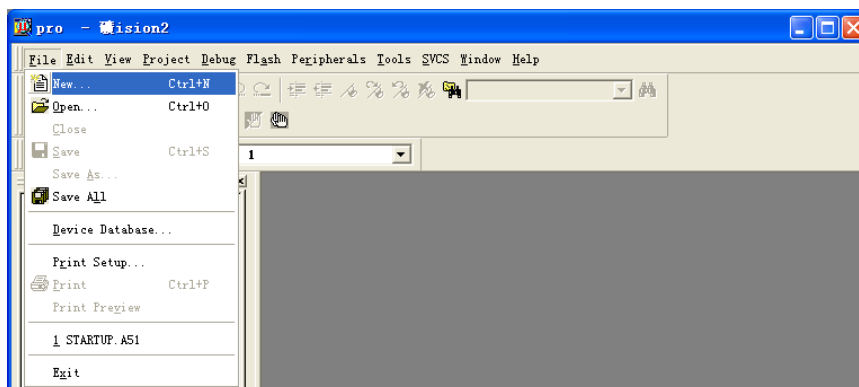


图 8-12 新建源程序文件

选择菜单 File->New，会产生一个名字为“Text×”的文件。需要把它存成 C 格式，这里命名为 pro.c，如图 8-13 所示。注意，C 程序文件保存为*.c，汇编文件保存为*.a51 或*.asm，如果是说明文件也可以保存为*.txt。

在此文件中输入程序的代码即可。

下面要将刚才的程序文件添加到工程中。如图 8-14 所示，鼠标右键单击工程“Source Group”，在弹出菜单中，选择“Add Files to Group ‘Source Group’”。

此时弹出选择文件对话框，如图 8-15 所示，先选择合适的“文件类型”，再选定源程序文件，点击“Add”，就把程序文件加入到工程中去。可以一次选择多个文件。

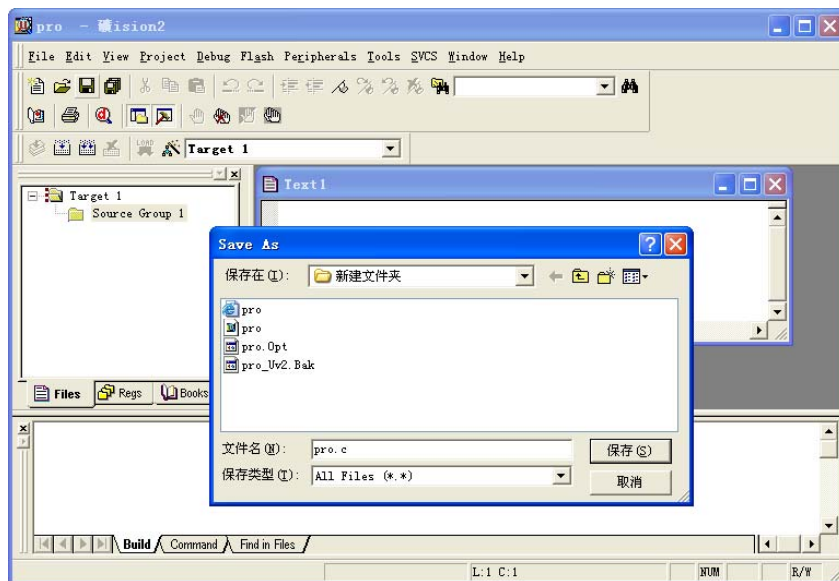


图 8-13 保存源程序文件

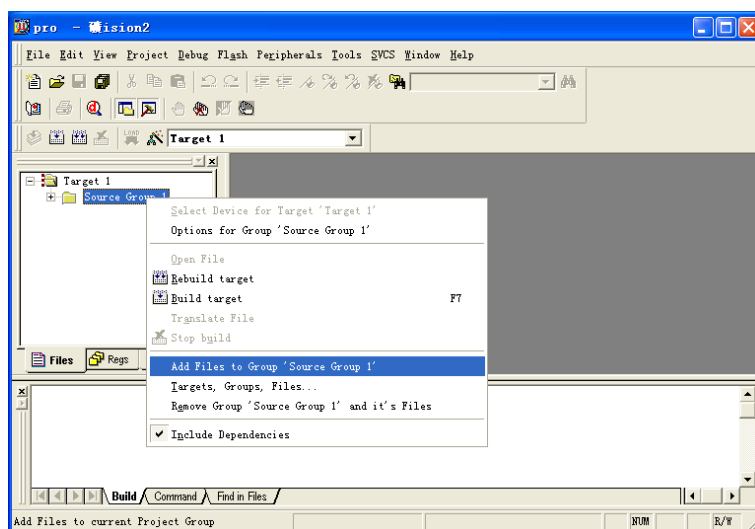


图 8-14 添加源程序文件



图 8-15 添加源程序文件对话框

8.2.2 设置工程属性

如图 8-16 所示，右键单击工程名“Target 1”。在弹出菜单中，第一项是更改该项目的单片机型号，和前面选择单片机型号的操作相同。第二项是更改工程设置，单击该选项将弹出如图 8-17 所示的属性对话框。

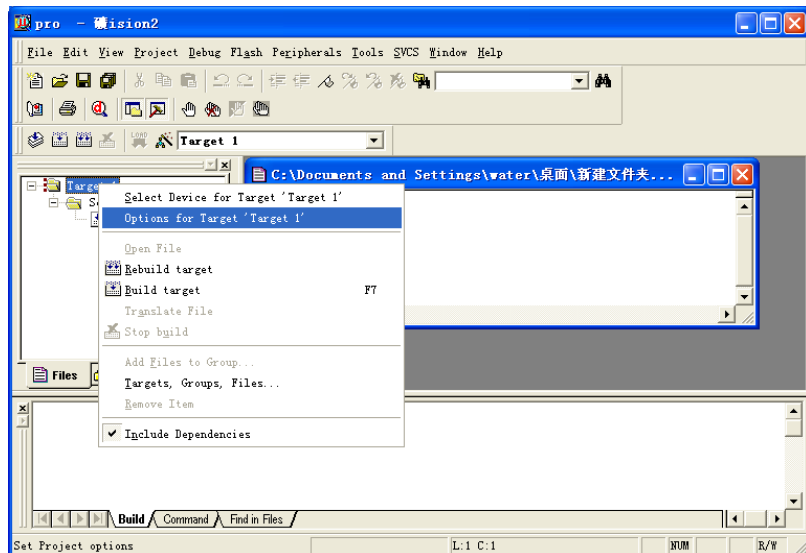


图 8-16 设置工程属性

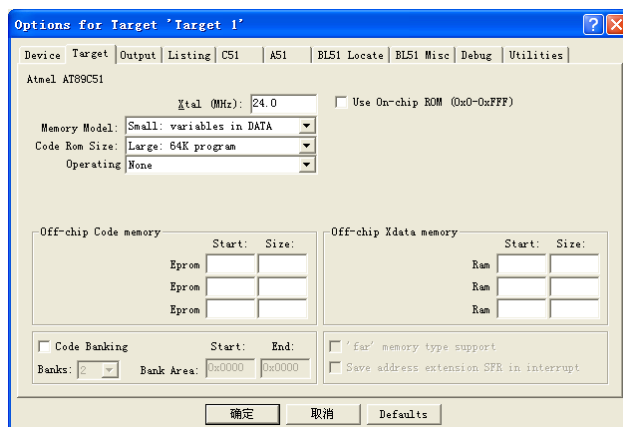


图 8-17 Target 属性标签

在属性对话框中，常用的标签页是 Device，Target，Output，Listing，C51 和 Debug。

1. Target 标签属性

Xtal (Mhz): 设置单片机的工作频率。该数值与最终产生的目标代码无关，仅用于软件调试时显示程序执行时间，正确设置该数值可使显示时间与实际所用时间一致，一般将其设置成与硬件所用晶振频率相同，如果没必要了解程序执行的时间也可以不设，默认是

24.0MHz。

Use On-chip ROM(0x0-0x1fff): 这个选项是使用片上的 Flash Rom, 默认不选, 请选择该项。

Off-chip Code memory: 表示在片外接的 Rom 的开始地址和大小, 默认无。

Off-Chip Xdata Memory: 外部数据存储器的起始地址和大小, 默认无。

Code Banking: 使用 Code Banking 技术.keil 可以支持程序代码超过 64KB 的情况, 最大可以有 2MB 的程序代码, 默认不选。

Memory Model: 有 3 个选项, 如图 8-18 所示。

(1) Small: 变量存储在内部 Ram 里;

(2) Compact: 变量存储在外部 Ram 里, 使用 8 位间接寻址;

(3) Large: 变量存储在外部 Ram 里, 使用 16 位间接寻址。

3 种存储方式都支持内部 256byte 和外部 64KB 的 Ram, 区别是变量的优先 (或默认) 存储位置。默认选 Small。

Code Rom Size:有 3 个选项, 如图 8-19 所示。

(1) Small: program 2K or less, 表示每个子程序最大为 2KB。

(2) Compact: 2K functions, 64K program, 表示每个子函数的程序大小不超过 2KB, 整个工程可以有 64KB 的代码;

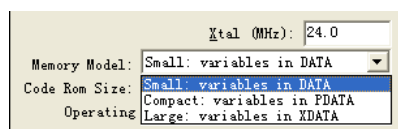


图 8-18 Memory Model 选项

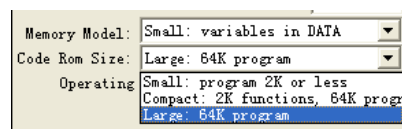


图 8-19 在 Code Rom Size 选项

(3) Large: 64K program, 表示程序或子函数都可以大到 64KB, 使用 code bank 还可以更大。

默认选 Large。在确认每个子程序不会超过 2KB 时可以选择 Compact。

Operating: 有 3 个选项, 如图 8-20 所示。

(1) None: 表示不使用操作系统

(2) RTX-51 Tiny: 表示使用 Tiny 操作系统

(3) RTX-51 Full: 表示使用 Full 操作系统

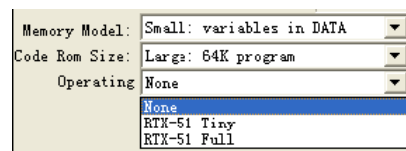


图 8-20 Operating 选项

Keil c51 提供了 Tiny 系统。Tiny 多任务操作系统使用定时器 0 来做任务切换, 效率很低, 无实用价值。Full 需要用户使用外部 Ram, 且需要单独购买运行库, 不能使用。默认选 None。

2. Output 标签属性

Output 标签属性如图 8-21 所示。

Select Folder for Objects: 选择编译之后的目标文件存储在哪个目录里, 默认位置为工程文件的目录里。

Name of Executable: 生成的目标文件的名称, 缺省是工程的名字。

Create Executable: 生成 OMF 以及 HEX 文件。OMF 文件名同工程文件名但没有带

扩展名。

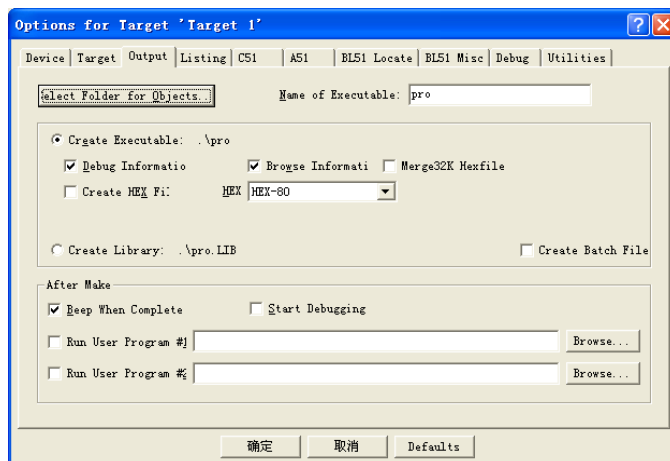


图 8-21 Output 属性标签

Create Hex File: 该选项默认未选，如果要写片做硬件实验就必须选中该项，否则编译之前不生成 Hex 文件。

Create Library: 生成 lib 库文件，默认不选。

After Make 有几个设置：

- (1) **Beep when complete:** 编译完成之后发出咚的声音；
- (2) **Start Debugging:** 马上启动调试（软件仿真或硬件仿真），默认不选中；
- (3) **Run User Program #1, Run User Program #2:** 这个选项可以设置编译完之后运行别的应用程序，比如有些用户自己编写的烧写芯片的程序，或调用外部的仿真程序，用户可以根据自己的需要设置。

3. Listing 标签属性

Listing 属性标签如图 8-22 所示。

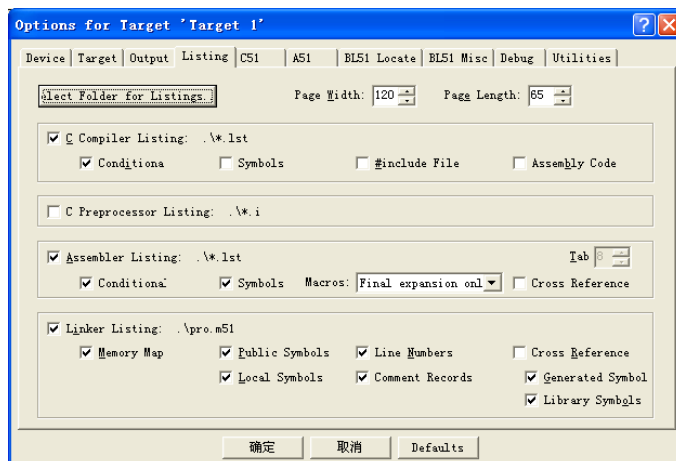


图 8-22 Listing 属性标签

Select Folder for Listings: 选择生成的列表文件存放的目录，默认为工程文件所在的目录。
Assembly Code: 生成汇编的代码，根据需要决定是否选择。

4. C51 属性标签

C51 属性标签，如图 8-23 所示。

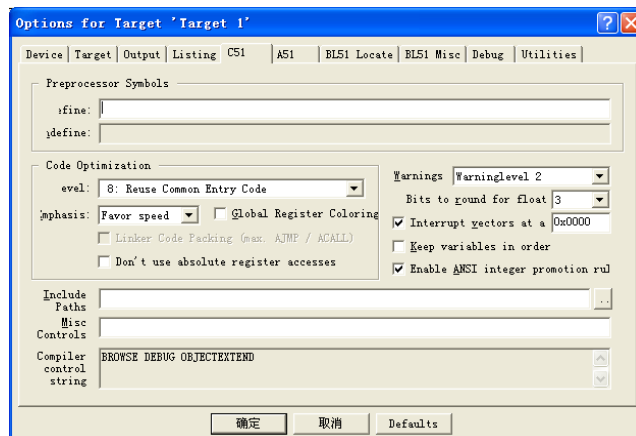


图 8-23 C51 属性标签

C51 标签页用于对 Keil C51 编译器的编译过程进行控制，其中比较常用的是 Code Optimization 组。

Level: 优化等级，默认第 8 级，一般不必修改，如果在编译中出现问题可以降低优化级别试一试。

Emphasis: 选择编译优先方式，第一项是代码量优化，第二项是速度优先，第三项是节省默认的，是速度优先，可根据需要更改。

5. Debug 属性标签

Debug 属性标签如图 8-24 所示。

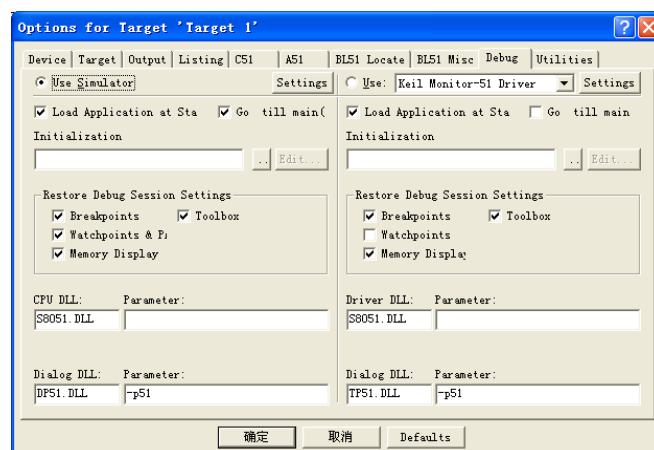


图 8-24 Debug 属性标签

Debug 页用来设置调试器:

Use Simulator: 选择 Keil 内置的模拟调试, 使用 Keil 的模拟调试器进行软件调试工作。该设置为工程默认设置。

Use: 设置硬件仿真, 默认是 Keil Monitor-51 Drive。

load Application at start: 在启动时直接装载程序, 默认选择。

Go till main: 装载后直接运行到 main 函数, 建议选择。

下面介绍如何设置用户的仿真器:

点击 **Setting** 按钮, 选择所用的端口和波特率, 其他设置一般不需要更改, 如图 8-25 所示, 点击 OK。

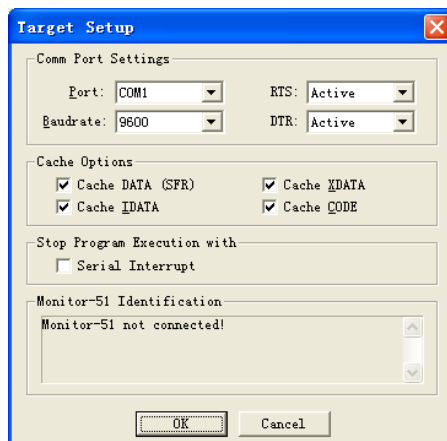


图 8-25 硬件通信设置

8.3 代码编译

在设置好工程后, 即可进行编译、连接。选择菜单 **Project->Build target**, 对当前工程进行连接, 如果当前文件已修改, 软件会先对该文件进行编译, 然后再连接以产生目标代码; 如果选择 **Rebuild All target files**, 将会对当前工程中的所有文件重新进行编译然后再连接, 确保最终生产的目标代码是最新的, 而 **Translate ...**项则仅对该文件进行编译, 不进行连接。

以上操作也可以通过工具栏按钮直接进行。图 8-26 所示的是有关编译、设置的工具栏按钮, 从左到右分别是编译、编译连接、全部重建、停止编译和对工程进行设置。

编译过程中的信息将出现在输出窗口中的 **Build** 页中, 如果源程序中有语法错误, 会有错误报告出现, 双击该行, 可以定位到出错的位置, 对源程序反复修改之后, 最终会得到如图 8-27 所示的结果, 提示获得了名为 **pro.hex** 的文件, 该文件即可被编程器读入并写到芯片中, 同时还产生了一些其他相关的文件, 可被用于 Keil 的仿真与调试, 这时可以进入下一步调试工作。



图 8-26 编译、连接、项目设置工具条

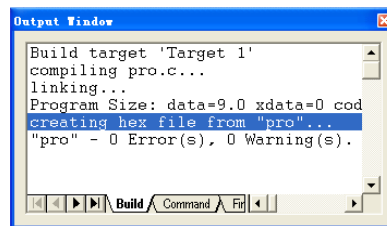


图 8-27 正确编译、连接之后的结果

8.4 程序调试

8.3 节中学习了如何建立工程、汇编、连接工程, 并获得目标代码, 这些只能确定源程

序没有语法错误，至于源程序中是否存在着其他错误，必须通过调试才能发现并解决，事实上，除了极简单的程序以外，绝大部份的程序都要通过反复调试才能得到正确的结果。因此，调试是软件开发中的一个重要环节，这一节将介绍常用的调试命令，利用在线汇编、各种设置断点进行程序调试的方法，并通过实例介绍这些方法的使用。

8.4.1 常用调试命令

在对工程成功地进行汇编、连接以后，按 **Ctrl+F5** 或者使用菜单 **Debug->Start/Stop Debug Session** 即可进入调试状态，模拟执行程序。

进入调试状态后，**Debug** 菜单项中原来不能用的命令现在已可以使用了，工具栏会多出一个用于运行和调试的工具条，如图 8-28 所示，**Debug** 菜单上的大部份命令可以在此找到对应的快捷按钮，从左到右依次是复位、运行、暂停、单步、过程单步、执行完当前子程序、运行到当前行、下一状态、打开跟踪、观察跟踪、反汇编窗口、观察窗口、代码作用范围分析、1# 串行窗口、内存窗口、性能分析、工具按钮等命令。



图 8-28 调试工具条

全速执行是指一行程序执行完以后紧接着执行下一行程序，中间不停止，并可以看到该段程序执行的总体效果，即最终结果正确还是错误，但如果程序有错，则难以确认错误出现在哪些程序行。

单步执行是每次执行一行程序，执行完该行程序以后即停止，等待命令执行下一行程序，此时可以观察该行程序执行完以后得到的结果，是否与编写该行程序所想要得到的结果相同，借此可以找到程序中问题所在。程序调试中，这两种运行方式都要用到。

使用菜单 **STEP** 或相应的命令按钮或使用快捷键 **F11** 可以单步执行程序，使用菜单 **STEP OVER** 或功能键 **F10** 可以以过程单步形式执行命令。所谓过程单步，是指将汇编语言中的子程序或高级语言中的函数作为一个语句来全速执行。

8.4.2 在线汇编

在调试过程中如果发现错误，可以直接对程序进行修改，但是不能编译，必须退出调试环境才能编译，然后再进入调试，这样使调试过程变得麻烦。为此 Keil 软件提供了在线汇编的能力。

把光标放在需要修改的程序行上，选择菜单 **Debug->Inline Assembly...**，出现如图 8-29

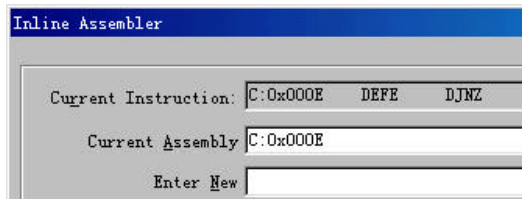


图 8-29 在线汇编窗口

的对话框，在 Enter New 后面的编辑框内直接输入需更改的程序语句，输入完后键入回车将自动指向下一条语句，可以继续修改，如果不再需要修改，可以单击右上角的关闭按钮关闭窗口。

8.4.3 断点设置

程序调试时，一些程序行必须满足一定的条件才能被执行到，这时就要使用到程序调试中的另一种非常重要的方法——断点设置。

断点设置的方法有多种，常用的是在某一程序行设置断点，设置好断点后可以从全速运行程序，一旦执行到该程序行即停止，可在此观察有关变量值，以确定问题所在。在程序行设置/移除断点的方法是将光标定位于需要设置断点的程序行，使用菜单 Debug->Insert/Remove BreakPoint 设置或移除断点（也可以在该行双击实现）；Debug->Enable/Disable Breakpoint 是开启或暂停光标所在行的断点功能；Debug->Disable All Breakpoint 暂停所有断点；Debug->Kill All BreakPoint 清除所有的断点设置。这些功能也可以用工具条上的快捷按钮进行设置。

除了在程序行设置断点外，Keil 还有多种设置断点的方法，按 Debug->Breakpoints... 弹出一个对话框，通过该对话框可对断点进行详细的设置，如图 4-30 所示。

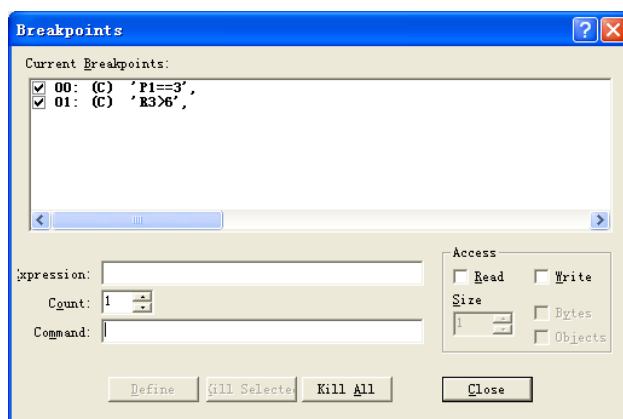


图 8-30 断点设置对话框

图 4-27 中 Expression 后的编辑框内用于输入表达式，该表达式用于确定程序停止运行的条件，这里表达式的定义功能非常强大，涉及 Keil 内置的一套调试语法。Count 指当前 Expression 中表达式满足的次数时设为断点，如假如条件“P1==3”的 Count=2，则在第二次“P1==3”时设为断点。Command 为满足该条件时在信息窗口输出的信息，与 Count 无关，每次满足该条件都会输出信息。

8.5 Keil 程序调试窗口

Keil 软件在调试程序时提供了多个窗口，如图 8-31 所示，主要包括输出窗口（Output Windows）、观察窗口（Watch&Call Stack Windows）、存储器窗口（Memory Window）、反汇编窗口（Disassembly Window）和串行窗口（Serial Window）等。进入调试模式后，可以通过菜单 View 下

的相应命令打开或关闭这些窗口。下面介绍常用的存储器窗口、观察窗口和工程窗口寄存器页。

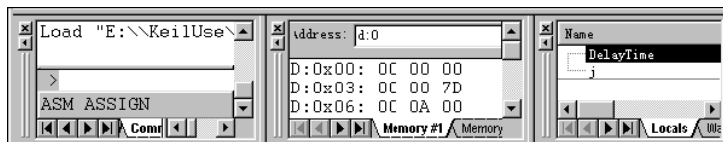


图 8-31 调试窗口（命令窗口、存储器窗口、观察窗口）

8.5.1 存储器窗口

存储器窗口中可以显示系统中各种内存中的值，如图 8-32 所示，通过在 Address 后的编辑框内输入“字母：数字”即可显示相应内存值，其中字母可以是 C，D，I，X，分别代表代码存储空间、直接寻址的片内存储空间、间接寻址的片内存储空间、扩展的外部 RAM 空间，数字代表想要查看的地址。例如输入“D：0”即可观察到地址 0 开始的片内 RAM 单元值，键入“C：0”即可显示从 0 开始的 ROM 单元中的值，即查看程序的二进制代码。

该窗口的显示值可以以各种形式显示，十进制、十六进制、字符型等，改变显示方式的方法是点鼠标右键，在弹出的快捷菜单中选择，该菜单用分隔条分成三部份，其中第一部分与第二部分的三个选项为同一级别。

选中第一部分的任一选项，内容将以整数形式显示，其中 Decimal 项是一个开关，如果选中该项，则窗口中的值将以十进制的形式显示，否则按默认的十六进制方式显示。Unsigned 和 Signed 后分别有三个选项，即 Char，Int，Long，分别代表以单字节方式显示、将相邻 2 字节组成整型数方式显示、将相邻 4 字节组成长整型方式显示，而 Unsigned 和 Signed 则分别代表无符号形式和有符号形式，究竟从哪一个单元开始的相邻单元则与用户的设置有关，以整型为例，如果你输入的是“I:0”，那么 00H 和 01H 单元的内容将会组成一个整型数，而如果输入的是“I: 1”，01H 和 02H 单元的内容全组成一个整型数，以此类推。有关数据格式与 C 语言规定相同，请参考 C 语言相关书籍，默认以无符号单字节方式显示。

第二部分有三项，ASCII 项是字符型式显示，Float 项是将相邻 4byte 组成浮点数形式显示，Double 项是将相邻 8byte 组成双精度形式显示。

第三部分的 Modify Memory at X:xx 用于更改鼠标处的内存单元值，选中该项即出现如图 8-33 所示的对话框，可以在对话框内输入要修改的内容。

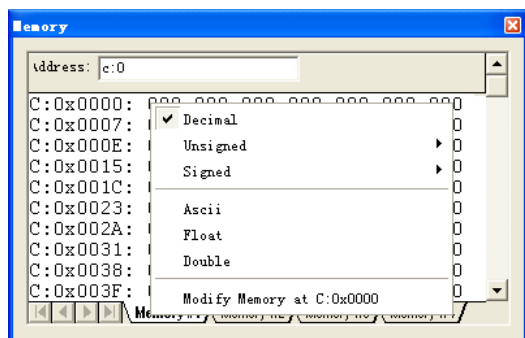


图 8-32 存储器数值各种方式显示选择

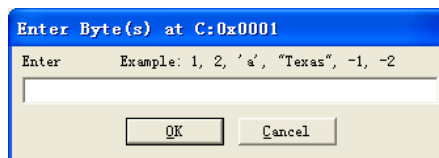


图 8-33 存储器的值的修改

8.5.2 观察窗口

观察窗口是很重要的一个窗口，工程窗口中仅可以观察到工作寄存器和有限的寄存器，如 A, B, DPTR 等，如果需要观察其他寄存器的值或者在高级语言编程时需要直接观察变量，就要借助于观察窗口了。一般情况下，仅在单步执行时才对变量值的变化感兴趣，全速运行时，变量的值是不变的，只有在程序停下来之后，才会将这些值最新的变化反映出来。但是，在一些特殊场合下也可能需要在全速运行时观察变量的变化，此时可以选择 View->Periodic Window Update (周期更新窗口)，确认该项处于被选中状态，即可在全速运行时动态地观察有关值的变化。但是，选中该项将会使程序模拟执行的速度变慢。

8.5.3 工程窗口寄存器页

图 8-34 是工程窗口寄存器页的内容，寄存器页包括了当前的工作寄存器组 and 系统寄存器组，系统寄存器组有一些是实际存在的寄存器，如 A, B, DPTR, SP, PSW 等；有一些是实际中并不存在或虽然存在却不能对其操作的，如 PC, Status 等。每当程序中执行到对某寄存器的操作时，该寄存器会以反色（蓝底白字）显示，用鼠标单击然后按下 F2 键，即可修改该值。

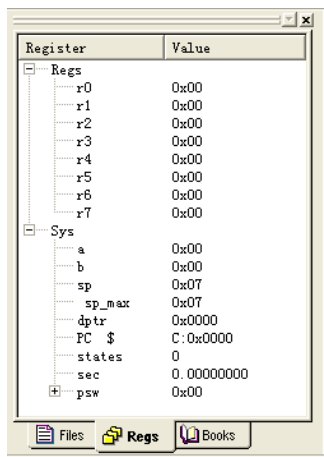


图 8-34 存储器的值的修改

8.5.4 外围接口

为了能够比较直观地了解单片机中定时器、中断、并行端口、串行端口等常用外设的使用情况，Keil 提供了一些外围接口对话框，通过 Peripherals 菜单选择，该菜单的下拉菜单内容与用户建立项目时所选的 CPU 有关，如果是选择的 89C51 这一类“标准”的 51 单片机，那么将会有 Interrupt (中断)、I/O Ports (并行 I/O 口)、Serial (串行口)、Timer (定时/计数器) 这四个外围设备子菜单。这些菜单只有在调试模式才能使用，非调试模式时不可用。打开这些对话框，列出了外围设备的当前使用情况、各标志位的情况等，可以在这些对话框中直观地观察和更改各外围设备的运行情况。图 8-35 所示的是单片机中断窗口，图 8-36 所示的是单片机

P0 口窗口，图 8-37 所示的是单片机串口窗口，图 8-38 所示的是单片机定时器 0 窗口。

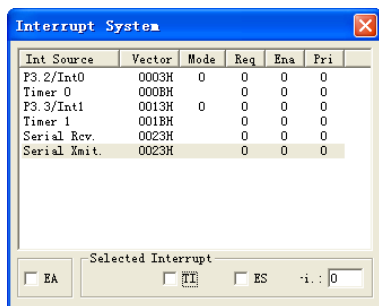


图 8-35 单片机中断窗口

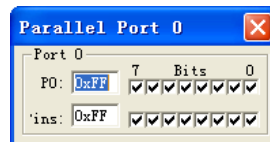


图 8-36 单片机 P0 口窗口

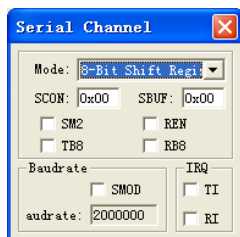


图 8-37 单片机串口窗口

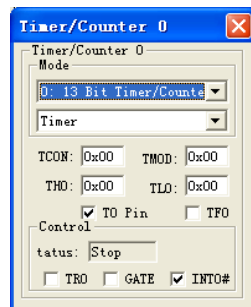


图 8-38 单片机定时器 0 窗口

在这些窗口中可以观察单片机外围设备的状态，还可以对它们的工作模式进行修改。

8.6 Keil 程序调试

Keil 程序调试有 3 种模式，第 1 种是纯片内程序模拟，没有外围芯片参与；第 2 种是用仿真试验板对外围芯片和设备进行仿真模拟，有专门的外围芯片和设备模拟文件；第 3 种是硬件连接调试。

8.6.1 片内程序模拟

如果是片内程序模拟，在属性页 Debug 标签中选择 Use Simulator，如图 8-39 所示，其他设置与前面介绍的一样。

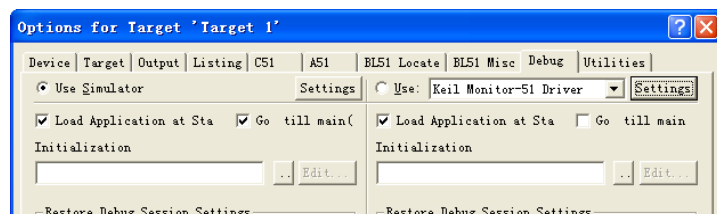


图 8-39 片内程序模拟设置

8.6.2 仿真试验板仿真

仿真试验板是用软件模拟单片机外围设备，实际上就是一个 dll 文件。安装时只要根据需要将该文件拷贝到 keil 软件的 c51\bin 文件夹中即可。

对于仿真试验板的设置，在属性页 Debug 标签中选择 Use Simulator，并且将仿真板文件填到 Dialog Dll Parameter 后面的编辑框里，输入-d 文件名，假如文件名为 simboard.dll，则输入-dsimboard，如图 8-40 所示。

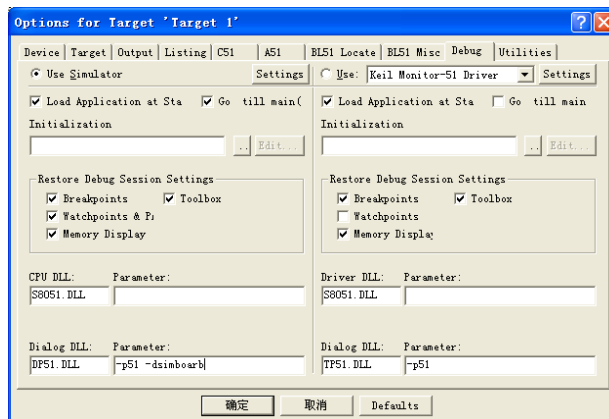


图 8-40 仿真试验板设置

8.6.3 硬件连接调试

硬件连接调试，连接好硬件设备，在属性页 Debug 标签中选择右边的 Use，下拉菜单中选中“Keil Monitor-51 Driver”，如图 8-41 所示。

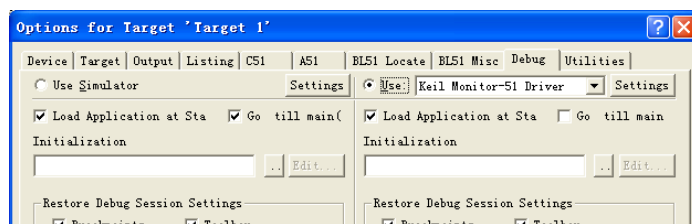


图 8-41 硬件连接调试设置

再单击后面的“Settings”按钮，设置 PC 与硬件板的通信参数，如图 8-42 所示。

Port: 选择 PC 与单片机试验板通信的端口：

Baudrate: 选择串行通信的波特率。

设置好以上参数，就可以在 PC 上直接对硬件试验板进行仿真调试。

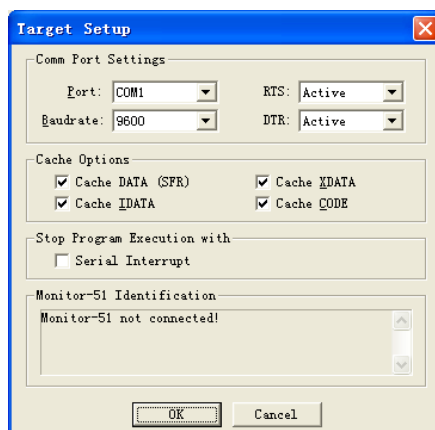


图 8-42 硬件试验板通信设置

习题和思考题

8-1 新建一个工程，编写一个程序文件，实现 P1.0 口输出 2Hz 的方波，用调试窗口实时观察。

8-2 对习题 8-1 中编写的程序进行调试，在调试过程中在程序的不同地方采用不同的方式设置几个断点，练习对断点的设置。

8-3 将习题 8-1 中编写的程序编译生成 Hex 文件，烧写到单片机里。

第9章 C51 单片机系统综合实例

重点与难点

本章重点是掌握 C51 系列单片机的一些基本扩展及其编程。

基本要求

1. 掌握 C51 系列单片机外扩存储器的变量定义及访问
2. 掌握 C51 系列单片机系统中断的编程方法
3. 掌握 C51 系列单片机定时/计数器的编程方法
4. 掌握 C51 系列单片机串口通信的编程方法
5. 掌握 C51 系列单片机外扩并口的设计及编程
6. 掌握 C51 系列单片机外接点阵型 LCD 中文显示的 C 语言编程

本章内容

1. 外部存储器综合扩展
2. 系统中断程序设计
3. 定时/计数器程序设计
4. 串口程序设计
5. 外扩并行 I/O 口的 C 语言编程
6. 点阵型 LCD 中文显示的 C 语言编程

9.1 外部存储器综合扩展

图 9-1 给出了一个单片机系统存储器的综合扩展电路，它扩展了 2 片 2764 和 2 片 6264。图中，P2.0~P2.4 直接与芯片连接；P2.5~P2.7 通过 74HC139 译码器连接各个芯片的片选；74HC373 作为系统扩展的地址锁存器。由此可得程序存储器 IC0 的地址范围：0000H~1FFFH；程序存储器 IC1 的地址范围：2000H~3FFFH；数据存储器 IC2 的地址范围：同 IC0，0000H~1FFFH，用访问指令与 IC0 区分；数据存储器 IC3 的地址范围：4000H~5FFFH。

对于图 9-1 所示的综合扩展电路，程序设计中非常重要的一步是定义存储器变量，下面对存储器变量的定义进行说明。

(1) 普通变量的定义

IC0/ IC1: `char code code1, code2;` 在扩展程序存储器中定义 char 类型变量。

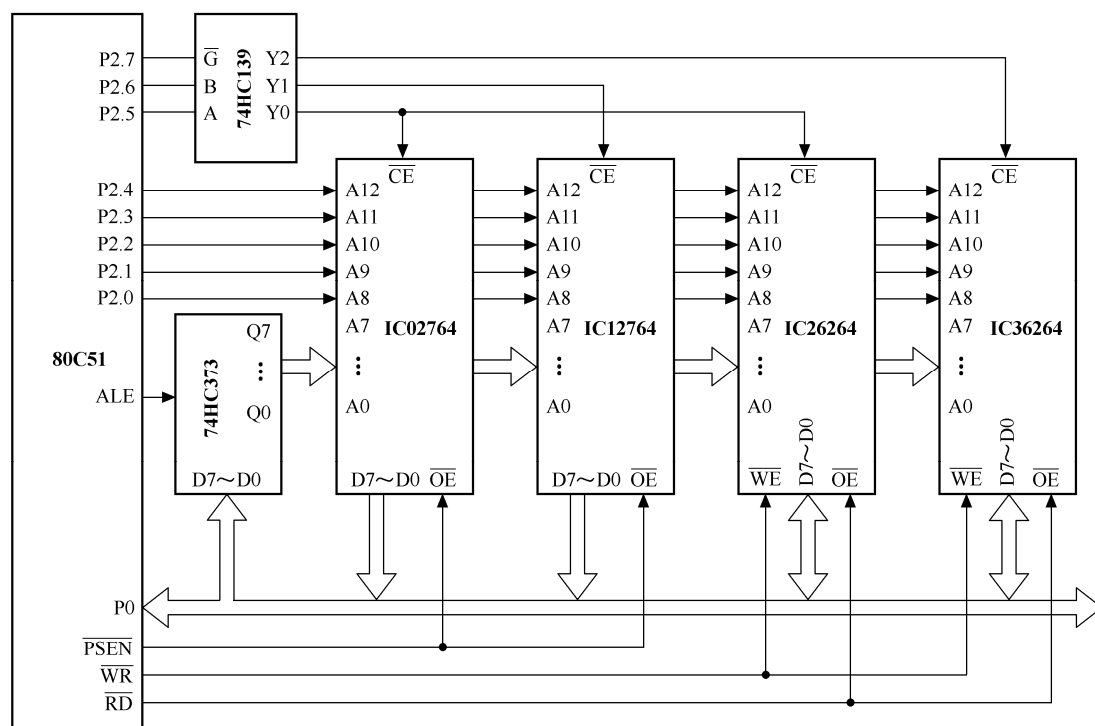


图 9-1 外部存储器综合扩展电路图

IC2/ IC3: unsigned char xdata data1, data2; 在扩展数据存储器中定义 unsigned char 类型变量。

(2) 绝对地址变量的定义

IC0: code3=CBYTE[0x0002]; 地址为 0x0002 的存储单元;

IC1: code4=CBYTE[0x2002]; 地址为 0x2002 的存储单元;

IC2: data3=XWORD[0x0002]; 地址为 0x0002 的存储单元;

IC3: data4=XWORD[0x4002]; 地址为 0x4002 的存储单元。

9.2 系统中断程序设计

【例 9-1】图 9-2 给出了一个记录并显示中断次数的电路图，图中采用 89C51 来进行电路控制。下面采用 C 语言进行程序设计，中断次数的判断放在主程序中。具体程序代码如下所示：

```
#include<reg51.h>
char i;
code char tab[16]={0x3f, 0x06, 0x5b, 0x4f, 0x66,
                   0x6d, 0x7d, 0x07, 0x7f, 0x6f, 0x77, 0x7c, 0x39, 0x5e, 0x79, 0x71};
int() interrupt 2
{
```

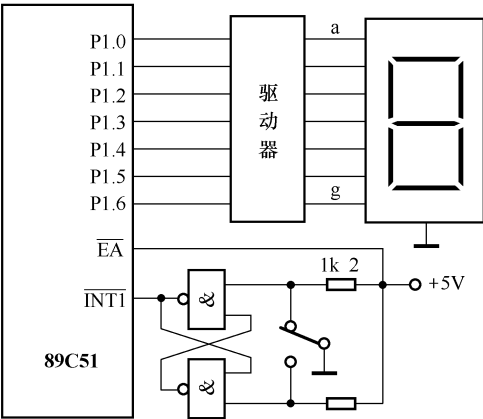


图 9-2 系统中断程序设计

```
i++; /*计中断次数*/
P1=tab[i]; /*查表，次数送显示*/
}
main()
{
    EA=1;
    EX1=1;
    IT1=1;
    ap5: P1=0x3f; /*显示"0"*/
    for(i=0;i<16;); /*当 i 小于 16 等待中断*/
    goto ap5; /*当 i=16 重复下一轮 16 次中断*/
}
```

9.3 定时/计数器程序设计

【例 9-2】 在 P1.7 端接一个发光二极管 LED，要求利用定时控制使 LED 亮一秒灭一秒周而复始，设 fosc=6MHz。

分析：T0 定时 100ms 的初值=100×10³/2=50000，即初值为-50000；T1 计数 5 个脉冲工作于方式 2，计数初值为-5。T0 和 T1 均采用中断方式，程序如下所示：

```
#include <reg51.h>
sbit P1_0=P1^0;
sbit P1_7=P1^7;
timer0() interrupt 1 using 1 /* T0 中断服务程序 */
{
    P1_0=! P1_0; /* 100ms 到 P1.0 反相*/
    TH0=-50000/256; /* 重载计数初值 */
}
```

```

    TL0=-50000%256;
}
timer1() interrupt 3 using 2    /* T1 中断服务程序 */
{
    P1_7=! P1_7;                /* 1s 到, 灯改变状态 */
}
main ()
{
    P1_7=0;                     /* 置灯初始灭 */
    P1_0=1;                     /* 保证第一次反相便开始计数 */
    TMOD=0x61;                  /* T0 方式 1 定时, T1 方式 2 计数 */
    TH0=-50000/256;             /* 预置计数初值 */
    TL0=-50000%256;
    TH1=-5;
    TL1=-5;
    IP=0x08;                    /* 置优先级寄存器 */
    EA=1; ET0=1; ET1=1;        /* 开中断 */
    TR0=1; TR1=1;              /* 启动定时/计数器 */
    for(;;){}                  /* 等待中断 */
}

```

9.4 串口程序设计

【例 9-3】 在内部数据存储器 20H~3FH 单元中共有 32 个数据, 要求采用方式 1 串行发送出去, 传送速率为 1200 波特, 设 $f_{\text{osc}}=12\text{MHz}$ 。

分析: T1 工作于方式 2 作波特率发生器, 取 SMOD=0, T1 的时间常数计算如下:

$$\text{波特率} = (2\text{SMOD}/32) \times f_{\text{osc}} / (12 \times (256 - x))$$

代入数值可得:

$$1200 = (1/32) \times (12 \times 10^6) / (12 \times (256 - x))$$

求得: $x=230=\text{E6H}$ 。

(1) 发送程序

```

#include<reg51.h>
main()
{
    unsigned char i;
    char *p;
    TMOD=0x20;
    TH1=0xe6;

```

```

    TL1=0xe6;
    TR1=1;
    SCON=0x40;
    p=0x20;
    for (i=0;i<=32;i++){
        SBUF=*p;
        p++;
        while (!TI);
        TI=0;
    }
}

```

(2) 接收程序

```

#include <reg51.h>
main()
{
    unsigned char i;
    char *p;
    TMOD=0x20;
    TH1=0xe6;
    TL1=0xe6;
    TR1=1;
    SCON=0x50;
    p=0x20;
    for (i=0;i<=32;i++){
        while (!RI);
        RI=0;
        *p=SBUF;
        p++;
    }
}

```

9.5 外扩并行 I/O 口的 C 语言编程

【例 9-4】 用 8155 作 6 位共阴极 LED 显示器接口，PB 口经驱动器 7407 接 LED 的段选，PA0~PA5 经反相驱动器 7406 接位选，待显示字符依次存于 dis-buf 数组，从右向左顺序显示。8155 命令字为 03H，table 为段码表，动态显示 6 个字符。8155 和 C51 的接口电路如图 9-3 所示。

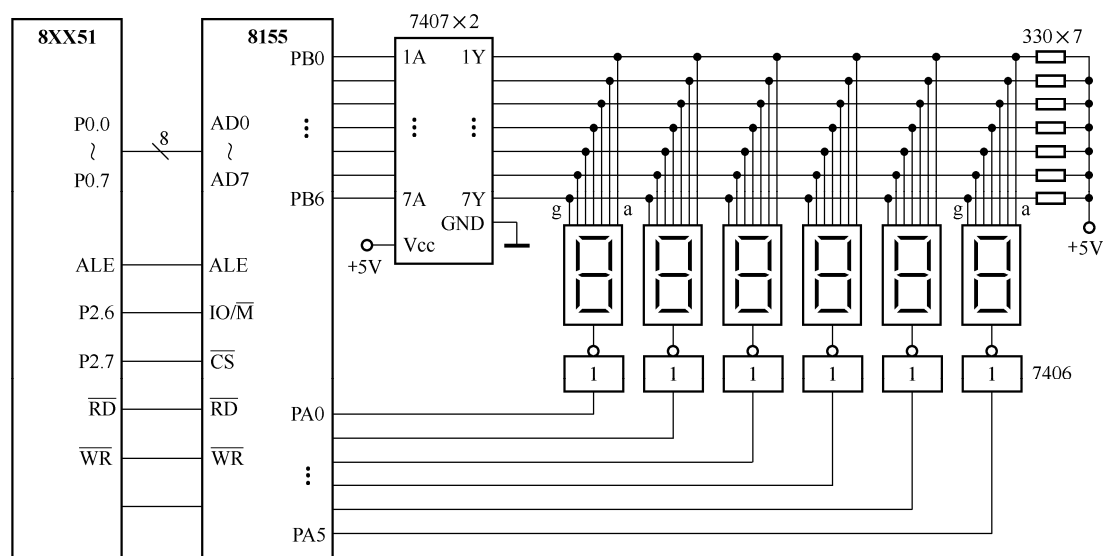


图 9-3 外扩并行 I/O 口的 C 语言编程

其中 8155 各口的地址：A 口对应 7FF1H，B 口对应 7FF2H，C 口对应 7FF3H，命令/状态口对应 7FF0H。

C 语言程序如下所示：

```
#include<absacc.h>
#include<reg51.h>
#define uchar unsigned char
#define COM8155 XBYTE[0x7ff0]
#define PA8155 XBYTE[0x7ff1]
#define PB8155 XBYTE[0x7ff2]
#define PC8155 XBYTE[0x7ff3]
uchar idata dis[6]={2, 4, 6, 8, 10, 12}; /* 存放显示字符 2、4、6、8、A、C */
uchar code table[18]={0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f,
0x6f, 0x77, 0x7c, 0x39, 0x5e, 0x79, 0x71, 0x40, 0x00};
void display(uchar idata *p)
{
    uchar sel, i, j;
    COM8155=0x03;sel=0x01; /*送命令字，选最右边的 LED*/
    for(i=0;i<6;i++)
    {
        PB8155=table[*p];PA8155=sel; /*送段码和位码*/
        for(j=400;j>0;j--); /*延时*/
        p--; /*地址指针下移位*/
        sel=sel<<1; /*左移一位*/
    }
}
```

```
}
main()
{
    display(dis+5);
}
```

9.6 点阵型 LCD 中文显示的 C 语言编程

【例 9-5】 工业生产中会经常用到 LCD 的中文显示，根据图 7-12 所示的电路编写点阵型 LCD 显示模块显示中文字符串“北京欢迎您”。

显示汉字（16×16 点阵）必须使用图形方式，HD61830 在图形方式时显示缓冲区单元与显示屏的对应关系如图 9-4 所示。

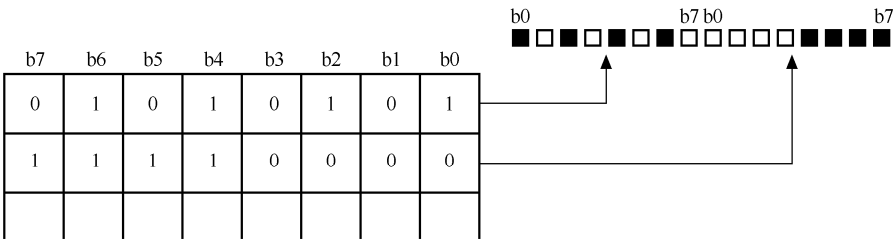


图 9-4 显示缓冲单元与显示屏的对应关系

上面的显示格式与日常习惯正好相反，如想在显示屏上显示 10010110，则须向 RAM 中写入 01101001，为适应习惯，编写函数 inva 实现一个将字节的各位倒转的功能。

16×16 汉字共有 32 个字节，这 32 个字节存放方式如下所示。

1	17
2	18
...	...
16	32

具体的语言程序如下所示：

```
#include<reg51.h>
#include<absacc.h>
#define uchar unsigned char
#define uint unsigned int
#define COM XBYTE[0x8003]
#define PA XBYTE[0x8000]
#define PB XBYTE[0x8001]
#define PC XBYTE[0x8002]
#define ZK CBYTE[0x2000]
```

```

#define DELAY 3
uchar code *hzp;
void dishz(uint chn);
void comd(uchar x, uchar y);
void wcode(uchar c);
void wdata(uchar d);
uchar inva(uchar b);
void main(void)
{
    int i;
    uint sad;
    for(i=0; i<1000; i++) {}
    COM=0x81;
    PB=0x00; PB=0xf0;    /*MGLS-240128 模块复位*/
    comd(0x00, 0x32);    /*方式设置, 方式显示, 光标禁止, 图形方式, 选用 CGROM*/
    comd(0x01, 0x07);    /*字体设置 1×8 字体*/
    comd(0x02, 0x1d);    /*显示域设置, 一行 30 个字符*/
    comd(0x03, 0x7f);    /*帧设置, 占空比为 1/128*/
    comd(0x08, 0x00);
    comd(0x09, 0x00);    /*设置缓冲区起始地址*/
    comd(0x0a, 0x00);
    comd(0x0b, 0x00);    /*设置光标地址指针*/
    wcode(0x0e);
    for(i=0; d<3840; i++) wdata(0x00);    /*清屏*/
    sad=0;
    hzp=&ZK;            /*显示 ZK 地址的汉字串*/
    while(sad<10)
    {
        dishz(sad);
        sad+=2;
    }
    while(1) ;
}

void wcode(uchar c)    /*写指令代码*/
{
    uchar i=DELAY;
    while(i) i--;
    PC=0x9f; PA=c; PC=0xdf; PC=0x5f; PC=0x1f; PC=0x9f;
}

```

```
void wdata(uchar d)          /*写指令参数*/
{
    uchar i=DELAY;
    while(i) i--;
    PC=0x8f; PA=d; PC=0xcf; PC=0x4f; PC=0x0f; PC=0x8f;
}
void comd(x, y)              /*写一条指令*/
uchar x, y;
{
    wcode(x);
    wdata(y);
}
uchar inva(uchar b)          /*字节各位倒转*/
{
    uchar v1=0;
    uchar v2=0;
    char i;
    uchar j1=0x80;
    uchar j2=0x01;
    for(i=7; i>=1; i=i-2)
    {
        v1=((b<<i)&j1)|v1;
        v2=((b>>i)&j2)|v2;
        j1=j1>>1;
        j2=j2<<1;
    }
    return(v1|v2);
}
void dishz(chn)              /*显示一汉字*/
uint chn;
{
    uchar i, j, k;
    uchar addl, addh;
    for(i=0; i<2; i++)
    {
        addl=chn%256;
        addh=chn/256;
        for(j=0; j<16; j++)
        {
```

```

wcode(0x0a);
wdata(addl);
wcode(0x0b);
wdata(addh);
k=inva(*hzp);
wcode(0x0c);
wdata(k);
hzp++;
addl+=30;
if(addl<30) addh++;
}
chn++;
}
}

```

2000H 地址开始放置的汉字由 Intel 十六进制文件 ZK2000.HEX 得到。十六进制文件 ZK2000.HEX 格式如下:

```

: 1020000004040404047C04040404041CE4440004
: 1020100080808898A0C0808080808082827E00DE
: 10202000020IOIFF001F1010101F01090911250214
: 10203000000004FE10F8101010F000403018080006
: 102040000000FC0445462828102824448101020CA5
: 10205000808080FC044840404040AOA010080E046E
: 10206000004126140404F414151514101028470037
: 1020700000847E44444444C4445448404046FC0008
: 102080000909131234599112141110025150900F92
: 102090000000FC044840504C44408000849212F020

```

在实际的设计中, 有专门的从汉字库中提取汉字字模数据的软件可以使用。

附录 C51 单片机指令系统表

1. 算术运算类指令

指 令	字 节 数	指 令 周 期	指 令	字 节 数	指 令 周 期
ADD A,@Ri	1	1	IN CA	1	1
ADD A,Rn	1	1	INC @Ri	1	1
ADD A,direct	2	1	INC Rn	1	1
ADD A,#data	2	1	INC DPTR	1	1
ADD C A,@Ri	1	1	INC direct	2	1
ADDC A,Rn	1	1	DEC A	1	1
ADDC A,direct	2	1	DEC @Ri	1	1
ADDC A,#data	2	1	DEC Rn	1	1
SUBB A,@Ri	1	1	DEC direct	2	1
SUBB A,Rn	1	1	MUL AB	1	4
SUBB A,direct	2	1	DIV AB	1	4
SUBB A,#data	2	1	DA A	1	1

2. 逻辑运算类指令

指 令	字 节 数	指 令 周 期	指 令	字 节 数	指 令 周 期
ANL A,@Ri	1	1	CLR A	1	1
XRL A,@Ri	1	1	ORL A,Rn	1	1
ANL A,Rn	1	1	CPL A	1	1
XRL A,Rn	1	1	ORL A,direct	2	1
ANLA,direct	2	1	RL A	1	1
XRL A,direct	2	1	ORL A,#data	2	1
ANL A,#data	2	1	RLC A	1	1
XRL A,#data	2	1	ORL direct,A	2	1
ANL direct,A	2	1	RR A	1	1
XRL direct,A	2	1	ORL direct,#data	3	2
ANL direct,#data	3	2	RRC A	1	1
XRL direct,#data	3	2	SWAP A	1	1
ORL A,@Ri	1	1			

3. 数据传送类指令

指 令	字 节 数	指 令 周 期	指 令	字 节 数	指 令 周 期
MOV A, @Ri	1	1	MOV Rn, A	1	1
MOV DPTR, #data16	3	2	PUSH direct	2	2
MOV A, Rn	1	1	MOV Rn, direct	2	2

续表

指 令	字 节 数	指 令 周 期	指 令	字 节 数	指 令 周 期
MOVC A, @A+DPTR	1	2	POP direct	2	2
MOV A, direct	2	1	MOV Rn, #data	2	1
MOVC A, @A+PC	1	2	XCH A, @Ri	1	1
MOV A, #data	2	1	MOV direct, A	2	1
MOVX A, @Ri	1	2	XCH A, Rn	1	1
MOV @Ri, A	1	1	MOV direct, @Ri	2	2
MOVX A, @DPTR	1	2	XCH A, direct	2	1
MOV @Ri, direct	2	2	MOV direct, Rn	2	2
MOVX @Ri, A	1	2	XCHD A, @Ri	1	1
MOV @Ri, #data	2	1	MOV direct, direct	3	2
MOVX @DPTR, A	1	2	MOV direct, #data	3	2

4. 位操作指令

指 令	字 节 数	指 令 周 期	指 令	字 节 数	指 令 周 期
CLR C	1	1	CLR bit	2	1
ANL C, bit	2	2	ORL C, bit	2	2
SETB C	1	1	SETB bit	2	1
ANL C, bit	2	2	MOV C, bit	2	1
CPL C	1	1	CPL bit	2	1
ORL C, bit	2	2	MOV bit, C	2	2

5. 控制转移类指令

指 令	字 节 数	指 令 周 期	指 令	字 节 数	指 令 周 期
NOP	1	1	JNB bit, rel	3	2
JZ rel	2	2	LJMP addr16	3	2
RET	1	2	JBC bit, rel	3	2
JNZ rel	2	2	SJMP rel	2	2
RETI	1	2	CJNE A, direct, rel	3	2
JC rel	2	2	JMP @A+DPTR	1	2
ACALL addr11	2	2	CJNE A, #data, rel	3	2
JNC rel	2	2	DJNZ Rn, rel	2	2
AJMP addr11	2	2	CJNE @Ri, #data, rel	3	2
JB bit, rel	3	2	DJNZ direct, rel3	2	2
LCALL addr16	3	2	CJNE Rn, #data, rel3	2	2

参 考 文 献

1. 何立民. 单片机高级教程. 北京: 北京航空航天大学出版社, 2000
2. 闫玉德等. MCS-51 单片机原理与应用. 北京: 机械工业出版社, 2003
3. 孙涵芳, 徐爱卿. MCS-51/96 系列单片机原理及应用. 北京: 北京航空航天大学出版社, 1996
4. 刘文涛. 单片机语言 C51 程序设计. 北京: 人民邮电出版社, 2005
5. 胡汉才. 单片机原理及其接口技术. 北京: 清华大学出版社, 2004
6. 马忠梅等. 单片机的 C 语言应用程序设计. 北京: 北京航空航天大学出版社, 2005
7. 钟睿等. MCS-51 单片机原理及应用开发技术. 北京: 中国铁道出版社, 2006
8. 沙占友等. 单片机外围电路设计. 北京: 电子工业出版社, 2006
9. 李玉梅. 基于 MCS-51 系列单片机原理的应用设计. 北京: 国防工业出版社, 2006