
8051 系列单片机 C 程序设计完全手册

求是科技 编著

人民邮电出版社

图书在版编目 (CIP) 数据

8051 系列单片机 C 程序设计完全手册 / 求是科技编著. —北京: 人民邮电出版社, 2006.4
ISBN 7-115-14601-2

I . 8... II . 求... III. ①单片微型计算机, 8051—程序设计—技术手册②C 语言—程序设计—技术手册 IV. ①TP368.1-62②TP312-62

中国版本图书馆 CIP 数据核字 (2006) 第 019706 号

内 容 提 要

本书首先从单片机基础、C 语言、开发编译环境、典型资源编程、单片机通信等几个方面介绍了 8051 系列单片机 C 程序设计所应掌握的基础知识, 内容全面、讲解清楚。然后通过工程开发实例和典型模块应用实例两部分内容系统地介绍单片机系统设计的方法使理论与实际相结合。

本书可以作为大学本、专科单片机技术课程的教材, 也可作为 8051 系列单片机 C 程序设计开发的技术人员以及高等学校相关专业师生的参考用书。

8051 系列单片机 C 程序设计完全手册

- ◆ 编 著 求是科技
责任编辑 张立科
- ◆ 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>
北京鸿佳印刷厂印刷
新华书店总店北京发行所经销
- ◆ 开本: 787×1092 1/16
印张: 35.25
字数: 867 千字 2006 年 4 月第 1 版
印数: 1~5 000 册 2006 年 4 月北京第 1 次印刷

ISBN 7-115-14601-2/TP · 5301

定价: 56.00 元 (附光盘)

读者服务热线: (010) 67132692 印装质量热线: (010) 67129223

前 言

单片机是微型计算机的一个重要的分支。自 1976 年 Intel 公司推出第一款 8 位单片机 MCS-48 开始，单片机在短短的几十年间获得了长足的发展，从最开始的 4 位机到更高性能的 8 位机，到速度更快、片内资源更为丰富的增强型 8 位机、16 位机甚至 32 位机，其性能不断增强，功能也日益完善。

随着计算机技术的发展，单片机的应用领域也越来越广泛，在工业控制、数据采集以及仪器/仪表自动化等许多领域都起着十分重要的作用。这对单片机产品设计开发人员即是机遇也是挑战。本书专门为使用 C 程序开发 8051 系列单片机的设计人员量身定做，既适合初学者按部就班学习，也适合单片机设计开发技术人员参考。

本书各章的安排如下。

第 1 章 单片机基础

主要介绍了单片机的发展状况、体系结构，同时对单片机的编程方法及编程环境进行了简单描述。

第 2 章 C 语言编程基础

主要介绍了 C 语言中常用的数据类型和程序控制语句。

第 3 章 C 语言高级编程

主要介绍了函数、数组、指针以及结构体等内容，同时对各部分进行了说明，分别列举了相应的示例。

第 4 章 C51 程序设计

主要介绍了 C51 对标准 C 语言的扩展、C51 函数库和 C 程序基本结构。

第 5 章 Windows 集成开发环境 μVision2

主要介绍了 μVision2 中各个菜单栏的作用，通过创建程序流程和调试流程详细介绍各菜单的使用以及仿真功能的应用。

第 6 章 C51 编译器

主要介绍了预处理的方法、C51 编译器控制指令和 C51 的高级配置文件。

第 7 章 C51 的典型资源编程

主要介绍了 C51 典型资源的编程方法，包括中断系统设计、定时/计数器的使用、I/O 口的使用和扩展存储器的方法等，最后还给出了一个使用多种资源的完整例程。

第 8 章 单片机通信

主要介绍了单片机通信的相关内容，包括串口通信、单片机点对点通信、单片机多机通信、单片机 I²C 总线通信、单片机与计算机的互连等。

第 9 章 C51 单片机的工程开发实例

通过一个典型的应用实例介绍了单片机工程开发的一般方法。

第 10 C51 单片机典型模块实例

主要以实例的形式，介绍了单片机典型模块的开发方法。

在本书的编写过程中，借鉴了许多现行教材的宝贵经验，在此，谨向这些作者表示诚挚的感谢。由于时间仓促，加之编者水平有限，书中有错误或是不足之处在所难免，敬请广大读者批评指正。

欢迎读者访问求是科技网站 <http://www.cs-book.com>，提出您的宝贵意见和建议。对于您遇到的问题，我们也将予以尽快解答。

编者
2006.3

目 录

第 1 章 单片机基础	1
1.1 单片机技术发展状况	1
1.2 51 系列单片机体系结构	2
1.2.1 内部结构	2
1.2.2 存储器组织结构	4
1.2.3 内部功能模块	10
1.2.4 外部引脚	16
1.2.5 系统资源扩展	20
1.3 单片机的编程方法	24
第 2 章 C 语言编程基础	25
2.1 基本概念	25
2.1.1 概述	25
2.1.2 变量与算术表达式	28
2.1.3 for 语句	33
2.1.4 符号常量	35
2.2 数据类型、运算符和表达式	36
2.2.1 C 语言的数据类型	36
2.2.2 常量与变量	37
2.2.3 整型数据	38
2.2.4 实型数据	40
2.2.5 字符型数据	41
2.2.6 运算符	46
2.2.7 表达式	51
2.3 程序控制语句	54
2.3.1 程序的 3 种基本结构	54
2.3.2 条件控制语句	54
2.3.3 程序应用举例	62
2.4 循环控制语句	64
2.4.1 while 语句	64
2.4.2 do...while 语句	66
2.4.3 for 语句	67
2.4.4 break 与 continue 语句	70
2.4.5 程序应用举例	71
2.5 小结	74
第 3 章 C 语言高级编程	75
3.1 函数与程序结构	75
3.1.1 函数的基本知识	75
3.1.2 返回非整数值的函数	79

3.1.3 外部变量	81
3.1.4 作用域规则	82
3.1.5 头文件	83
3.1.6 静态变量	84
3.1.7 寄存器变量	84
3.1.8 分程序结构	85
3.1.9 初始化	85
3.1.10 递归	87
3.2 数组	88
3.2.1 一维数组	88
3.2.2 二维数组	91
3.2.3 多维数组	96
3.2.4 数组的初始化	96
3.3 指针	98
3.3.1 指针与指针变量	98
3.3.2 指针变量的定义与引用	99
3.3.3 指针运算符与指针表达式	100
3.3.4 指针与数组	103
3.3.5 指针的地址分配	119
3.3.6 指针数组	121
3.3.7 指向指针的指针	129
3.4 结构体与共用体	132
3.4.1 结构体类型变量的定义和引用	132
3.4.2 结构体数组的定义和引用	136
3.4.3 结构体指针的定义和引用	143
3.4.4 共用体	147
3.5 小结	151
第 4 章 C51 程序设计	152
4.1 C51 对标准 C 语言的扩展	152
4.1.1 存储区域	152
4.1.2 数据变量分类	153
4.1.3 存储器模式	156
4.1.4 绝对地址的访问	157
4.1.5 指针	158
4.1.6 函数	163
4.2 C51 函数库	166
4.2.1 字符函数 CTYPER.H	167
4.2.2 一般 I/O 函数 STDIO.H	174
4.2.3 字符串函数 STRING.H	183
4.2.4 标准函数 STDLIB.H	192

4.2.5 数学函数 MATH.H.....	194
4.2.6 绝对地址访问 ABSACC.H	196
4.2.7 内部函数 INTRINS.H	197
4.2.8 变量参数表 STDARG.H.....	199
4.2.9 全程跳转 SETJMP.H	200
4.2.10 访问 SFR 和 SFR_bit 地址 REGxxx.H.....	201
4.3 C51 程序编写	202
4.3.1 C 程序基本结构	202
4.3.2 编写高效的 C51 程序及优化程序.....	206
第 5 章 Windows 集成开发环境 μVision2	209
5.1 μVision2 编辑界面及其功能介绍	209
5.1.1 μVision2 界面综述	209
5.1.2 主菜单栏	210
5.1.3 μVision2 功能按钮	212
5.1.4 μVision2 窗口环境	215
5.2 应用 μVision2 开发流程介绍	221
5.2.1 建立新项目	221
5.2.2 常用环境配置	223
5.2.3 代码优化	227
5.2.4 目标代码调试	227
5.3 CPU 仿真	228
5.3.1 μVision2 调试器	228
5.3.2 调试命令	234
5.3.3 存储器空间	237
5.3.4 表达 (Expressions)	237
5.3.5 技巧	248
5.4 深入了解 μVision2	250
5.4.1 μVision2 的项目管理	250
5.4.2 使用技巧	253
5.4.3 μVision2 调试函数	259
第 6 章 C51 编译器	269
6.1 预处理	272
6.1.1 宏定义	272
6.1.2 文件包含	273
6.1.3 条件编译	274
6.1.4 其他预处理命令	276
6.2 C51 编译器控制指令详解	278
6.2.1 源控制指令	278
6.2.2 列表控制指令	280
6.2.3 目标控制指令	286

6.3	C51 的高级配置文件	299
6.3.1	目标程序启动配置文件——STARTUP.A51.....	299
6.3.2	CPU 初始化文件——START751.A51.....	307
6.3.3	静态变量初始化文件——INIT.A51.....	309
6.3.4	专用变量初始化文件——INIT751.A51	319
第 7 章	C51 的典型资源编程	323
7.1	中断系统设计	323
7.2	定时/计数器的使用	327
7.3	I/O 口的使用	332
7.4	扩展存储器	336
7.4.1	外部 ROM	336
7.4.2	外部 RAM	337
7.4.3	外部串行 E ² PROM	343
7.5	一个使用多种资源的完整例程	351
7.5.1	项目需求	351
7.5.2	步进电机背景知识	351
7.5.3	解决方案设计与实现	354
第 8 章	单片机通信	374
8.1	串口通信	374
8.1.1	串行通信基础	374
8.1.2	单片机串口使用	375
8.2	单片机点对点通信	379
8.2.1	通信接口设计	379
8.2.2	单片机点对点通信程序设计	380
8.3	单片机多机通信	390
8.3.1	主机部分通信程序设计	392
8.3.2	从机部分通信程序设计	395
8.4	单片机 I ² C 总线通信	399
8.4.1	I ² C 总线介绍	399
8.4.2	I ² C 总线硬件接口设计	409
8.4.3	I ² C 总线模拟硬件接口软件设计	415
8.4.4	I ² C 总线系统的设计要点	420
8.5	单片机与计算机的互连	420
8.5.1	电路设计	420
8.5.2	电路的 C51 程序代码	421
8.5.3	计算机端的 Visual C++ 程序代码	422
第 9 章	C51 单片机的工程开发实例	442
9.1	单片机系统设计方法	442
9.2	C51 系统设计的相关知识	444
9.2.1	硬件以及电路的知识	444

9.2.2 软件以及编程语言的知识	447
9.3 C51 系统设计需要注意的一些问题	447
9.3.1 单片机资源的分配	447
9.3.2 单片机的寻址	448
9.3.3 C51 函数的返回值	448
9.3.4 单片机的看门狗功能	449
9.3.5 单片机的外设	449
9.3.6 单片机的功耗	449
9.4 有关 C51 的一些问题	450
9.5 键盘和发光数码管显示	452
9.5.1 电路设计的背景及功能	452
9.5.2 电路的设计	453
9.5.3 键盘扫描电路的 C51 程序代码	453
9.5.4 电路的改进——键盘的消抖动程序	457
9.5.5 电路的显示部分——LED 数码管电路	461
9.6 A/D、D/A 转换器使用	465
9.6.1 电路设计的背景及功能	465
9.6.2 电路的设计	466
9.6.3 电路的 C51 程序代码	468
9.7 基于单片机的数字钟	472
9.7.1 电路设计的背景及功能	472
9.7.2 电路的设计	472
9.7.3 电路的 C51 程序代码	473
第 10 章 C51 单片机典型模块实例	481
10.1 典型外部 ROM 和 RAM 器件的使用	481
10.1.1 实例功能	481
10.1.2 器件和原理	481
10.1.3 电路	485
10.1.4 程序设计	487
10.2 液晶显示和驱动实例	488
10.2.1 实例功能	488
10.2.2 器件和原理	489
10.2.3 电路	494
10.2.4 程序设计	496
10.3 用 A/D 芯片进行电压测量	507
10.3.1 实例功能	507
10.3.2 器件和原理	507
10.3.3 电路	514
10.3.4 程序设计	516
10.4 使用 DS1820 进行温度补偿和测量	518

10.4.1 实例功能	518
10.4.2 器件和原理	519
10.4.3 电路	522
10.4.4 程序设计	523
10.5 语音芯片在单片机系统中的使用	528
10.5.1 实例功能	528
10.5.2 器件和原理	528
10.5.3 电路	532
10.5.4 程序设计	534
10.6 时钟芯片在单片机系统中的应用	536
10.6.1 实例功能	536
10.6.2 器件和原理	537
10.6.3 电路	539
10.6.4 程序设计	540
10.7 单片机中滤波算法的实现	543
10.8 信号数据的 FFT 变换	549

第1章 单片机基础

单片机是微型计算机的一个重要的分支。随着计算机技术的发展，单片机的应用领域也越来越广泛，它在工业控制、数据采集以及仪器仪表自动化等许多领域都起着十分重要的作用。本章首先简要介绍了单片机的发展状况，使读者对单片机技术的现状有一个初步的了解。接着以 Intel 公司的 8051 系列单片机为例，介绍单片机的体系结构。最后，对单片机的编程方法及编程环境进行了简单的描述。

1.1 单片机技术发展状况

从 1971 年美国德州仪器 (Texas Instrument) 公司首次推出 TMS-1000 单片机 (4 位机) 至今，单片机技术已成长为计算机技术领域中的一个非常重要的分支。在不断地发展和完善中，单片机技术已经建立了自己的技术特征、发展道路和独特的应用环境。按照单片机的生产技术水平，单片机的发展过程可以分为 4 位机、8 位机、16 位机和 32 位机 4 个阶段。

1. 4 位机阶段

1971 年美国德州仪器公司首次推出 4 位单片机 TMS-1000 后，各个计算机生产公司迅速跟进，相继推出了自己的 4 位单片机。如美国国家半导体公司 (NS, National Semiconductor) 的 COP4XX 系列，日本电气公司 (NEC) 的 μPD75XX 系列，日本东芝公司 (Toshiba) 的 TMP47XXX 系列以及日本松下公司 (Panasonic) 的 MN1400 系列等。4 位单片机的控制功能较弱，多用于家用电器、电子玩具等控制器。

2. 8 位机阶段

从 1976 年 9 月美国 Intel 公司推出 MCS-48 系列单片机开始，单片机技术进入了 8 位单片机时代，这一系列的单片机集成了 8 位 CPU、并行 I/O 口、8 位定时/计数器、寻址范围不大于 4kB，不包括串行口。这期间的 8 位单片机因为功能有限，属于低档 8 位单片机。

随着半导体集成工艺的提高，从 1978 年起许多公司纷纷推出了一些高性能的 8 位单片机。如 Intel 公司的 MCS-51 系列，Motorola 公司的 MC6801 系列，齐洛格公司 (Zilog) 的 Z8 系列，NEC 公司的 μPD78XX 系列，Atmel 的 AT89 等。这类单片机的寻址能力达到 64kB，具备更大的片内 RAM 和 ROM，提供全双工的串口，有的产品还增加了片内 A/D、D/A 转换器。其中 Intel 公司的 MCS-51 系列单片机以其出色的性价比和良好的兼容性，获得了良好的市场声誉。

8 位单片机的控制功能较为出色，且品种繁多，因此得到了最为广泛的应用，其技术不断地得到完善和发展。近年来，在高档 8 位机的基础上，单片机功能进一步得到提高，如 Intel 公司的 8X252、Zilog 公司的 Super8、Motorola 公司的 MC68HC 等，各公司不但进一步扩大了片内 ROM 和 RAM 的容量，还增加了通信功能、DMA 传输功能以及高速 I/O 功能等。这些产品

代表了 8 位单片机的发展方向。

3. 16 位机阶段

16 位单片机是在 1983 年以后发展起来的，这类单片机的 CPU 是 16 位的，运算速度普遍高于 8 位机，部分单片机寻址能力达到 1MB，片内含 A/D、D/A 转换器，支持高级语言，多用于智能仪表等复杂的应用控制领域。典型产品有 Intel 公司的 MCS-96/98 系列、Motorola 公司的 M68HC16 系列、NS 公司的 HPCXXXX 系列等。

4. 32 位机阶段

近年来，随着家用电子系统的发展，32 位单片机的应用前景广泛。32 位的单片机字长为 32 位，是单片机中的顶端产品，具有极高的运算速度，部分产品还集成了 MMU，多用于嵌入式系统。这类产品包括 Motorola 公司的 M68300 系列、日本 Hitachi 公司的 SH 系列等。

从市场的需要情况看，目前 8 位机的市场最大，因此，熟悉 8 位机的新发展十分必要。8 位单片机的发展主要体现在以下几个方面。

- CPU 功能增强

提高 CPU 的功能主要体现在提高 CPU 的运算速度和运算精度上。传统的 MCS-51 系列单片机的最高频率为 12MHz，而新的 51 系列兼容机使用更高的时钟频率，如 Atmel 公司的 AT89 系列最高频率为 24MHz，Philips 公司的 51 系列产品最高频率可以达到 33MHz。

- 增加内部资源

单片机的内部资源越丰富，在单片机硬件系统中需要的外部硬件开销就越小，这样就可以有效地减小产品的体积，提高产品的可靠性。因此，世界的各大计算机厂商都热衷于开发增强型 8 位单片机。这类单片机集成了 A/D、D/A 转换器、看门狗电路、DMA 通道和总线接口等，有些厂商还在单片机中集成了晶振和 LCD 驱动。

- 低电压和低功耗

在实际的工业应用场合，对单片机系统的体积和功耗的要求是比较高的。因此，单片机制造商普遍采用 CMOS 工艺，并提供空闲和掉电两种工作方式。

1.2 51 系列单片机体系结构

在 8 位单片机中，Intel 公司的 MCS-51 系列单片机凭借其稳定的性能、高性价比以及良好的兼容性，在各个领域得到了最为广泛的应用。本节主要介绍 8051 单片机的一些基础知识，包括 8051 单片机的硬件结构、存储器组织结构、外部引脚功能和系统资源的扩展。

1.2.1 内部结构

8051 系列的各种单片机由于其生产厂商和型号的不同，在片内存储器容量、中断系统、外围功能模块、最高时钟频率以及处理器速度等方面有很大的不同，但其硬件系统的基本结构相同，均包括算术逻辑单元 ALU、片内 RAM、片内 ROM、I/O 端口、定时系统、中断系统等基本的功能单元。8051 单片机的内部结构如图 1-1 所示。

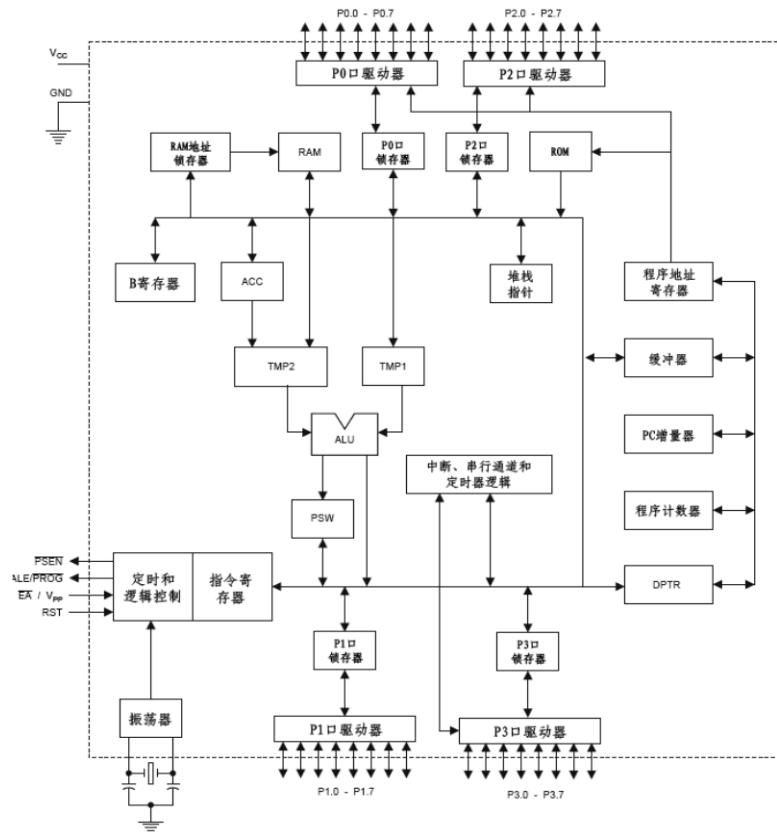


图 1-1 8051 内部结构图

- 算术逻辑单元 ALU

8051 内部包含一个 8 位的算术逻辑单元 ALU，用于处理各种算术运算和逻辑运算。作为 8051 系统的核心，ALU 为用户提供了丰富的指令系统和极快的指令运行速度，在外部时钟频率为 12MHz 的情况下，多数指令的执行时间仅为 1 μ s，乘法指令为 4 μ s。

- 片内 RAM 存储器

8051 单片机提供了 128Byte 片内 RAM 存储器，其地址空间为 00H~7FH。其中 00H~1FH 为 4 组通用工作存储器区，每个区包含 8 个编号为 R0~R7 的 8 位寄存器。此外 1FH~2FH 为位寻址空间，映射位地址的 00H~7FH。

- 片内 ROM 存储器

8051 系列单片机通常提供至少 4kB 的片内 ROM 空间，用于保存用户的程序指令。早期的 8051 单片机使用掩模 ROM 作片内程序存储器，随着存储技术的发展，当前常用的 8051 单片机多使用可电擦写的 E²PROM 作为单片机的程序存储器，如 Atmel 公司的 AT89C51 等。由于 8051 单片机的地址总线宽度为 16 位，因此 ROM 和 RAM 的最大寻址空间均为 64kB。

- I/O 端口

8051 单片机共有 32 根 I/O 线，即 4 个并行 8 位接口，分别记为 P0、P1、P2 和 P3。此外，8051 还提供了一个全双工的串口，使用两根 I/O 线，可编程选定 4 种工作方式。

- 定时系统

8051 系列单片机包含两个 16 位的定时/计数器，支持 4 种工作方式。

- 中断系统

8051 的中断系统包含 5 个中断源，两级中断优先级，其中每个中断源的优先级都是可编程设置的。

- 布尔处理器

8051 还内置了一个 1 位微处理器，这个微处理器有自己的 CPU、位寄存器、I/O 口和指令集，在开关决策、逻辑电路仿真和实时测控方面都有明显的优势，极大地增强了 8051 单片机的处理功能。在 8051 系列单片机中，8 位机和布尔处理器的硬件资源是复合在一起的。

1.2.2 存储器组织结构

8051 单片机的存储器结构与普通的微机系统不同，其程序存储器和数据存储器使用两个独立的地址空间，是单独编址的。从结构上看，8051 的存储空间可以分为 4 个部分，分别为程序存储空间（片内 ROM）、片内数据存储空间（片内 RAM）和片外数据存储空间（片外 RAM）、特殊功能寄存器（SFR）。而 8051 兼容单片机又可以分为 51 系列和 52 系列两个子体系，这两个系列的存储器组织并不是完全一样的，在以后对存储器的各个部分进行介绍时，将会指出这些不同点。

8051 单片机的存储器结构如图 1-2 所示，其中图（a）为 51 子系列结构图，图（b）为 52 子系列结构图。



图 1-2 8051 存储器结构

1. 程序存储空间

单片机中的程序存储空间用于保存在单片机中执行的程序和表格常数等信息。通常由 ROM、EPROM 或 E²PROM 等组成。

在 8051 单片机中，使用专门的程序计数器（PC）来存放将要执行的指令地址。每取出指令的一个字节，PC 中的值就自动加 1，指向下一地址字节，使程序顺序执行。PC 的位长为 16 位，因此程序存储器的最大可寻址空间为 64kB。

程序存储器也可以分为两个部分，分别为片内程序存储区和片外程序存储区。这里片内程序存储区是指单片机片内自带的程序存储空间，是与单片机处理器集成在一起的，51 系列和 52 系列单片机这部分空间的大小是不同的，前者为 4kB，后者为 8kB。当程序的大小超出了片内存储空间的范围时，就需要扩展片外程序存储器。8051 单片机执行指令时是根据 EA 引脚的电平来决定从片内/片外程序存储器读取指令的顺序的。当 EA=1 时，先执行片内存储器的程序，当 PC 的内容超出片内程序存储器的最大范围后（对 51 子系列，该值为 0FFFH。对 52 子系列，该值为 1FFFH），将自动转向片外程序存储器，当 EA=0 时，CPU 直接从片外程序存储器读取指令，这实际上相当于不使用片内的存储器空间。因此，对早期的无片内程序存储器的 8031 系列单片机，EA 引脚应直接接地。

8051 单片机上电复位后，程序计数器为 0000H，此外，0003H~0032H 被保留，用于中断服务程序，其地址分配如表 1-1 所示。

表 1-1 复位、中断入口地址

中断源	入口地址
复位	0000H
外部中断 0	0003H
定时/计数器 0 溢出	000BH
外部中断 1	0013H
定时/计数器 1 溢出	001BH
串行口中断	0023H
定时器 2 溢出或 T2EX 负跳变（52 子系列）	002BH

8051 只为每个中断保留了 8 个单元存放中断的服务程序，对一般的应用而言，这是远远不够的，因此，在实际的应用中，常常在相应的入口地址处使用转移指令，使程序转向被实际分配的中断服务程序段。

2. 数据存储空间

8051 的数据存储空间分为片内和片外两个数据存储空间，这两个存储空间相互独立编址，分别使用不同的指令访问。其片内数据存储器通常被分为两个部分，分别为片内 RAM 块和特殊功能寄存器（SFR）块。对 51 子系列，前者有 128Byte RAM，编址为 00H~7FH，后者编址为 80H~7FH。对 52 子系列，片内 RAM 为 256Byte，编址为 00~FFH，其中后 128Byte 地址空间与 SFR 块是重合的，但由于其访问指令不同，不会引起逻辑上的混乱。

8051 单片机片内 RAM 分为工作寄存器区、位寻址区、数据缓冲区 3 个部分，结构如图 1-3 所示。



图 1-3 片内 RAM 结构 (前 128Byte)

(1) 工作寄存器区

8051 单片机的 00H~1FH 部分为工作寄存器区，共包含 4 组工作寄存器，每组含 8 个寄存器，表示为 R0~R7。其地址与寄存器对应关系如表 1-2 所示。

表 1-2 RAM 工作寄存器区地址与寄存器的对应关系

工作区 0		工作区 1		工作区 2		工作区 3	
地址	寄存器	地址	寄存器	地址	寄存器	地址	寄存器
00H	R0	08H	R0	10H	R0	18H	R0
01H	R1	09H	R1	11H	R1	19H	R1
02H	R2	0AH	R2	12H	R2	1AH	R2
03H	R3	0BH	R3	13H	R3	1BH	R3
04H	R4	0CH	R4	14H	R4	1CH	R4
05H	R5	0DH	R5	15H	R5	1DH	R5
06H	R6	0EH	R6	16H	R6	1EH	R6
07H	R7	0FH	R7	17H	R7	1FH	R7

当前程序使用的工作寄存器是由程序状态字 (PSW) 中的 RS0 位和 RS1 位来决定的，通过修改 PSW 寄存器中的 RS0 和 RS1 位，可以快速地切换工作区，达到快速保护现场的目的，有效提高程序的相应速度。RS0、RS1 位与工作区的对应关系如表 1-3 所示。

表 1-3 RS0、RS1 位与工作区的对应关系

RS1	RS0	当前工作区
0	0	工作区 0
0	1	工作区 1
1	0	工作区 2
1	1	工作区 3

(2) 位寻址区

20H~2FH 单元为 8051 的位寻址区，对这 16 个单元中的每一位，8051 都为其分配了一个位地址，位地址范围为 00H~07H。位寻址空间的每一位都可以作为软件触发器使用 8051 的位指令直接进行处理。位寻址区各单元与位地址空间对应关系如表 1-4 所示。

表 1-4 位寻址区单元与位地址对应关系

位寻址单元地址	对应位地址空间	位寻址单元地址	对应位地址空间
20H	00H~07H	28H	40H~47H
21H	08H~0FH	29H	48H~4FH

续表

位寻址单元地址	对应位地址空间	位寻址单元地址	对应位地址空间
22H	10H~17H	2AH	50H~57H
23H	18H~1FH	2BH	58H~5FH
24H	20H~27H	2CH	60H~67H
25H	28H~2FH	2DH	68H~6FH
26H	30H~37H	2EH	70H~77H
27H	38H~3FH	2FH	78H~7FH

(3) 数据缓冲区

30H~7FH 是 8051 的数据缓冲区，可以作为用户 RAM 使用。由于 8051 的片内 RAM 是统一编址的，使用同一命令访问，因此，前两个区未使用的资源也可以作为数据缓冲区使用，对 52 子系列的单片机，这部分的空间为 30H~FFH。

数据缓冲区的空间多用于堆栈的数据空间。堆栈是按照先进后出的原则进行读写的特殊 RAM 区域。8051 系列单片机的堆栈区域是不固定的，栈顶地址由寄存器 SP 指定。

在 8051 中，堆栈是向上生长的，即栈顶地址总大于栈底地址，堆栈从栈底地址单元开始，向高地址延伸。因此，数据压入堆栈，SP 值自动增 1；数据从堆栈弹出，SP 自动减 1。8051 单片机的堆栈示意如图 1-4 所示。

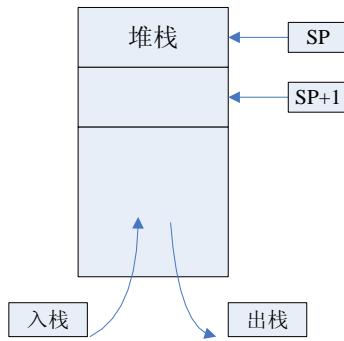


图 1-4 8051 单片机堆栈

单片机复位后，SP 的初值为 07H，通常在使用时将其初始化为 20H，以避开工作寄存器空间。

3. 特殊功能寄存器

8051 的特殊功能寄存器区（SFR）又称为专用寄存器区，是用来对片内的定时控制、中断控制、串行接口等内部功能单元进行管理、控制、监控的一组特殊功能的 RAM 区。8051 单片机中共有 26 个特殊功能寄存器，离散地分布在 80H~FFH 的地址空间范围内。其中 12 个寄存器可以使用位寻址，映射 80H~FFH 的位地址空间。此外，还有 5 个寄存器 TL2、TH2、T2CON、RCAP2L 和 RCAP2H 是 52 子系列所特有的，这些寄存器所在的地址在 51 系列单片机中无定义。8051 特殊功能寄存器的名称及地址映射如表 1-5 所示。对 52 子系列特有的寄存器表中用*标出。

表 1-5

8051 特殊功能寄存器地址映射表

特殊功能寄存器	地址	特殊功能寄存器	地址	特殊功能寄存器	地址
P0	80H	TH0	8CH	TL2*	CCH
SP	81H	TH1	8DH	TH2*	CDH
DPL	82H	P1	90H	IE	A8H
DPH	83H	SCON	98H	P3	B0H
PCON	87H	SBUF	99H	IP	B8H
TCON	88H	P2	A0H	PSW	D0H
TMOD	89H	T2CON*	C8H	ACC	E0H
TL0	8AH	RCAP2L*	CAH	B	F0H
TL1	8BH	RCAP2H*	CBH		

可以位寻址的特殊寄存器，其地址均为 8 的整数倍，这些寄存器的各位对应位地址空间的 80H~FFH，其位地址及位功能标记如表 1-6 所示。

表 1-6

特殊功能寄存器各位位地址及位功能

特殊功能寄存器	地址	映射位地址	位功能标记							
			D7	D6	D5	D4	D3	D2	D1	D0
P0	80H	80H~87H	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
TCON	88H	88H~8FH	TF1	TR1	TF0	TR0	IE1	IT0	IE0	IT0
P1	90H	90H~97H	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
SCON	98H	98H~9FH	SM0	SM1	SM2	REN	TB8	RB8	T1	R1
			D7	D6	D5	D4	D3	D2	D1	D0
P2	A0H	A0H~A7H	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0
IE	A8H	A8H~AFH	EA	—	—	ES	ET!	EX1	ET0	EX0
P3	B0H	B0H~B7H	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0
IP	B8H	B8H~BFH	—	—	—	PS	PT1	PX1	PT0	PX0
T2CON*	C8H	C8H~CFH	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T	CP/RL2
PSW	D0H	D0H~D7H	CY	AC	F0	RS1	RS0	OV	—	P
ACC	E0H	E0H~E7H	—	—	—	—	—	—	—	—
B	F0H	F0H~F7H	—	—	—	—	—	—	—	—

下面依次介绍 8051 特殊寄存器的各项功能。

(1) 程序与数据指针控制

- PC：程序计数器（PC）用于保存下一条要执行的指令的地址，每取一条指令，PC 中的地址就自动增加 1。PC 的位长为 16 位，因此最大寻址空间为 64kB。在硬件结构上，PC 独立于 SFR 之外。
- SP：栈指针（SP）是 8 位的特殊功能寄存器，用于指示 8051 堆栈栈顶的位置，系统复位后，SP 初始化为 07H。
- DPTR：数据指针（DPTR）是一个 16 位的特殊功能寄存器，其高位字节用 DPH 表示，低位字节用 DPL 表示。DPTR 在程序中既可以作为一个 16 位的寄存器 DPTR 来使用，也可以作为两个 8 位寄存器 DPH、DPL 来访问。

(2) 累加器与寄存器

- ACC：累加器（ACC）是最常用的特殊功能寄存器，大部分单操作数指令均以 ACC 作为其操作数，多数双操作数指令的第一个操作数也取自 ACC。此外，各种运算的结果

一般也保存在 ACC 中。

- B: 寄存器 B 主要用于乘、除等操作, 作为运算的第二个操作数, 也用于保存运算的结果。

(3) I/O 口锁存器

- P0: P0 口锁存器, 允许进行位寻址操作。
- P1: P1 口锁存器, 允许进行位寻址操作。
- P2: P2 口锁存器, 允许进行位寻址操作。
- P3: P3 口锁存器, 允许进行位寻址操作。

(4) 定时控制

- T0、T1: 8051 单片机内含有两个 16 位定时/计数器, 它们各自由两个独立的 8 位寄存器组成, 分别为 TH0、TL0、TH1 和 TL1。程序可以对这 4 个 8 位寄存器寻址, 但不能将 T0、T1 作为独立的 16 位寄存器来访问。
- TMOD: TMOD 用于控制定时/计数器的工作方式及 4 种工作模式, 其中低 4 位为定时器 T0 的方式控制字, 高 4 位为定时器 T1 的方式控制字。

(5) 中断控制

- TCON: TCON 寄存器的高 4 位为定时/计数器 T0、T1 的控制寄存器和定时/计数溢出中断标志。低 4 位为控制外部中断 1 和外部中断 0 的中断触发方式选择和中断产生标志。
- IE: IE 寄存器用于开放或屏蔽单片机的各个中断源。
- IP: 8051 提供两级中断优先级, 可以通过设置 IP 寄存器的相应位对各个中断源的中断优先级进行独立控制。

(6) 串行控制

- SCON: SCON 寄存器用于设置串行口的工作方式和查询接收、发送中断产生标志。
- SBUF: 串行数据缓冲器 SBUF 用于存放串口中欲发送或已接收的数据, 它由两个独立的寄存器构成, 一个是发送缓冲器, 一个是接收缓冲器, 它们共用一个地址。当从 SBUF 取数据时, 访问接收缓冲器, 当向 SBUF 写数据时, 访问发送缓冲器。

(7) 程序状态寄存器

- PSW: 程序状态寄存器 PSW 用于显示当前 8051 处理器的运行状况。PSW 的结构如下:

D7	D6	D5	D4	D3	D2	D1	D0
CY	AC	F0	RS1	RS0	OV	—	P

PSW 寄存器各位功能如下:

CY 进位标志位, 表示在字节运算时发生进位或借位。

AC 辅助进位标志位, 当 ACC 低 4 位发生进位时置位该标志位。

F0 通用标志位, 由用户定义。

RS1、RS0 工作寄存器组选择位。

OV 溢出标志位, 表示运算时发生溢出。

P 奇偶校验标志位。

通过 PSW 的各状态位, 可以实现程序跳转、多字节运算等功能。

1.2.3 内部功能模块

1. 8051 定时系统

对 8051 单片机而言，定时系统是其中最为重要的功能模块。在检测、控制等应用中常用来进行系统的定时检测、定时控制等，在单片机通信领域，定时器也起着十分重要的作用。下面对单片机定时系统的工作原理进行说明。

(1) 8051 内部定时器/计数器

8051 系列单片机内部设置了两个 16 位可编程的定时/计数器 T0 和 T1，具有定时器方式和计数器方式两种工作方式，可编程控制 4 种工作模式。8051 定时系统还包括两个定时控制寄存器 TCON 和 TMOD。8051 的定时系统如图 1-5 所示。

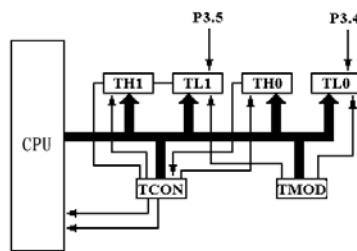


图 1-5 8051 定时系统

8051 的两个 16 位定时/计数器 T0 和 T1 均由两个独立的 8 位寄存器组成，分别为 TH0、TL0、TH1 和 TL1。程序可以对这 4 个 8 位寄存器寻址，但不能将 T0、T1 作为独立的 16 位寄存器来访问。

寄存器 TMOD 用于控制和设定定时器的工作方式和 4 种工作模式，其中低 4 位用于 T0，高 4 位用于 T1。其各位定义如下：

D7	D6	D5	D4	D3	D2	D1	D0
T1				T0			
GATE	C/T	M1	M0	GATE	C/T	M1	M0

- C/T：该位为定时或计数功能选择位，当 C/T=1 时，定时/计数器工作在计数方式，此时定时/计数器使用外部引脚 P3.4、P3.5 输入的脉冲作为计数脉冲。当 C/T=0 时，定时/计数器工作在定时方式，此时定时/计数器使用单片机的机器周期作为计数脉冲。
- M1、M0：定时/计数器工作模式选择位，用于设定定时/计数器的 4 种工作模式。其对应关系如表 1-7 所示。

表 1-7 定时/计数器工作模式选择

M1	M0	工作模式	功能说明
0	0	模式 0	13 位定时/计数器
0	1	模式 1	16 位定时/计数器
1	0	模式 2	常数自动装入的定时/计数器
1	1	模式 3	仅适用于 T0，分为两个 8 位定时/计数器，对 T1 停止计数

- GATE：定时/计数器门控位，用于设定定时/计数器的启动是否受外部中断请求信号的

控制。GATE=1 时，T0 和 T1 的启动分别受芯片引脚 $\overline{INT0}$ (P3.2) 和 $\overline{INT1}$ (P3.3) 的控制。GATE=0 时，定时/计数器的启动与引脚 $\overline{INT0}$ 、 $\overline{INT1}$ 无关。

寄存器 TCON 的高 4 位用于定时/计数器的控制和溢出标识。其高 4 位定义如下：

D7	D6	D5	D4	D3~D0
TF1	TR1	TF0	TR0	

- TF1：T1 定时/计数器的溢出中断标志位，当 T1 计数溢出时，该位置位，并在允许 T1 中断时，向 CPU 发出中断请求，CPU 响应中断后，该位清 0。TF1 也可以由程序查询或清 0。
- TR1：T1 定时/计数器的运行控制位，当 TR1=1 时启动 T1 开始计数，当 TR1=0 时，T1 停止计数。
- TF0：T0 定时/计数器的溢出中断标志位，当 T0 计数溢出时，该位置位，并在允许 T0 中断时，向 CPU 发出中断请求，CPU 相应中断时，该位清 0。TF0 也可以由程序查询或清 0。
- TR0：T0 定时/计数器的运行控制位，当 TR0=1 时启动 T0 开始计数，当 TR0=0 时，T0 停止计数。

(2) 8051 定时/计数器工作模式

8051 单片机的定时/计数器有 4 种工作模式，由 TMOD 寄存器的 M0、M1 两位确定。下面具体介绍这 4 种工作模式。

- 工作模式 0

将 TMOD 的 M1、M0 位分别设为 0，定时/计数器工作在模式 0 下。此时定时/计数器是一个 13 位定时/计数器，由 TLx 的低 5 位和 THx 的高 8 位构成，TLx 的高 3 位未用。TLx 的低 5 位溢出后向 THx 进位，最大计数值为 2^{13} 。以定时/计数器 0 为例，工作模式 0 下定时/计数器的等效逻辑电路如图 1-6 所示。

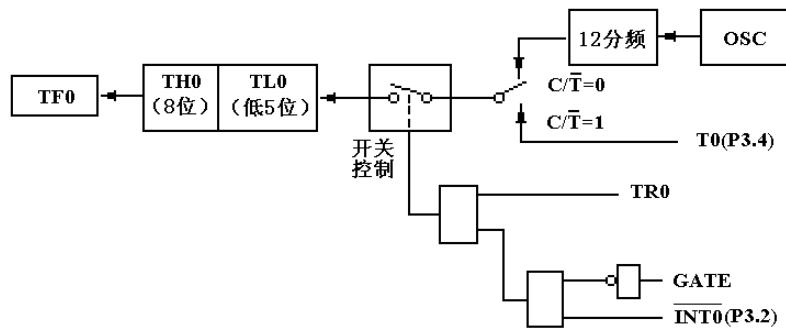


图 1-6 定时/计数器 0 在工作模式 0 下的逻辑电路图

GATE=0 时，定时/计数器的启动由 TR0 来决定，TR0 为 1 时，允许定时/计数器开始计数。GATE=1 时，定时/计数器受外部中断 $\overline{INT0}$ 控制，仅当 $\overline{INT0}$ 为高电平且 TR0=1 时才允许定时/计数器开始计数。

- 工作模式 1

将 TMOD 的 M1、M0 位分别设为 0、1，定时/计数器工作在模式 1 下。此时定时/计数器是

一个 16 位定时/计数器，TLx 组成定时/计数器的低 8 位，THx 组成定时/计数器的高 8 位，TLx 溢出后向 THx 进位，最大计数值为 2^{16} 。以定时/计数器 0 为例，工作模式 1 下定时/计数器的等效逻辑电路如图 1-7 所示。

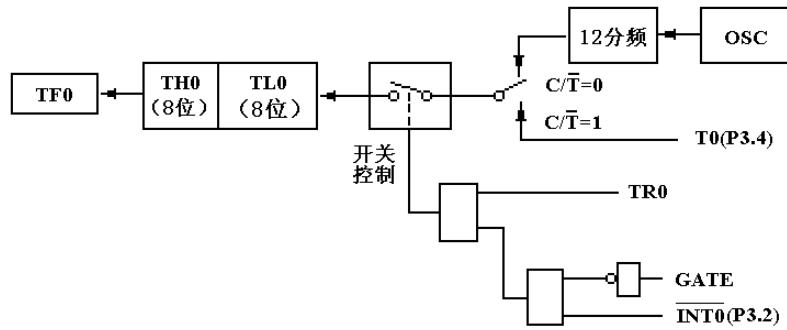


图 1-7 定时/计数器 0 在工作模式 1 下的逻辑电路图

除去计数位数不同，工作模式 1 下的定时/计数器各方面均与工作模式 0 下的相同。

- 工作模式 2

将 TMOD 的 M1、M0 位分别设为 1、0，定时/计数器工作在模式 2 下。此时定时/计数器是一个可自动装入初值的 8 位定时/计数器。其中 THx 用于保存计数初值，TLx 用于计数。当 TLx 计数溢出时，会自动将 THx 中的数据送入 TLx 重新开始计数。以定时/计数器 0 为例，工作模式 2 下定时/计数器的逻辑电路如图 1-8 所示。

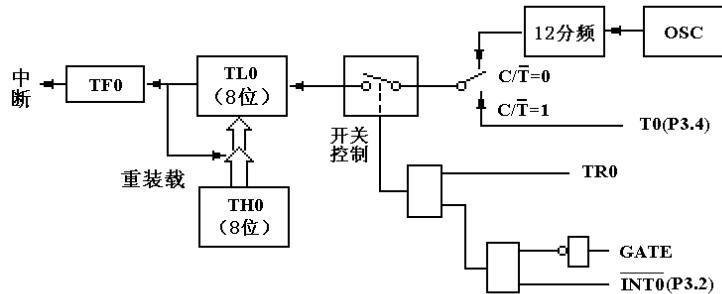


图 1-8 定时/计数器 0 在工作模式 2 下的逻辑电路图

在这种工作模式下，定时器在溢出后会自动装载初值，这样就避免了软件重新装载初值耽误的时序，可以精确地进行时间定位。

- 工作模式 3

将 TMOD 的 M1、M0 位分别设为 1、1，定时/计数器工作在模式 3 下，该模式仅对定时/计数器 0 有效，此时定时/计数器 0 被拆成两个独立的 8 位计数器 TL0 和 TH0。在这种模式下，TL0 既可以作计数器使用，也可以作为定时器使用，使用定时/计数器 0 的包括控制位和引脚信号等的全部资源。其功能和操作与模式 0 或模式 1 完全相同。而 TH0 使用定时/计数器 1 的 TR1 和 TF1 两位，只能用作定时器使用。定时/计数器 0 在工作模式 3 下的逻辑电路如图 1-9

所示。

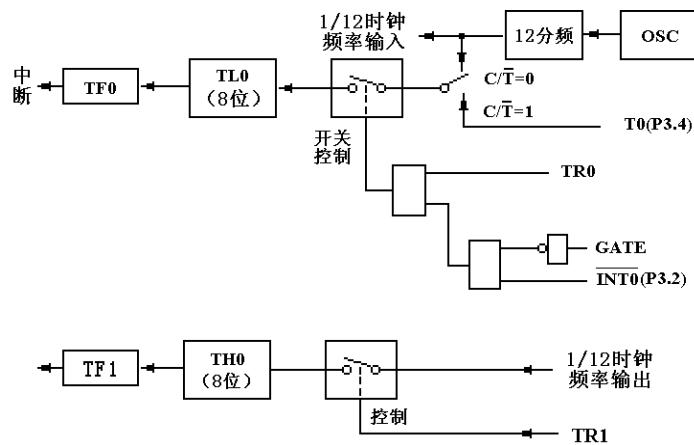


图 1-9 定时/计数器 0 在工作模式 3 下的逻辑电路图

当 T0 在工作模式 3 下时，T1 虽然仍可以工作在模式 0、1、2 下，但无法应用在需要中断的场合，通常用作串行口的波特率发生器。T1 此时的工作模式逻辑电路图如图 1-10 所示。

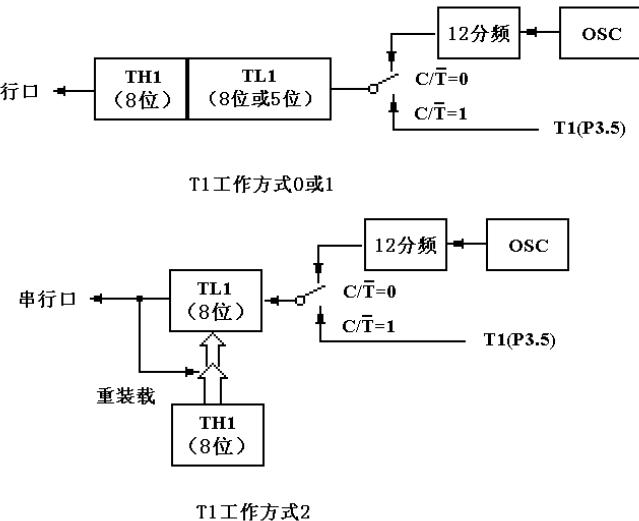


图 1-10 T0 在模式 3 下 T1 的工作模式逻辑电路图

2. 8051 中断系统

8051 单片机提供了相当强大的单片机中断系统，提供了 5 个中断请求源，具有两个中断优先级，可以提供两级中断服务程序嵌套。用户可以屏蔽所有的中断，也可以独立地对每一个中断源进行屏蔽或开启。此外，还可以用软件独立设置每个中断源的优先级。8051 的中断系统如图 1-11 所示。

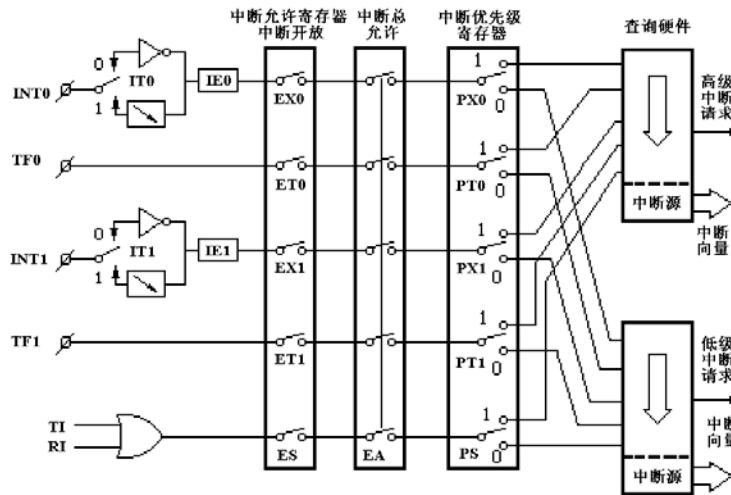


图 1-11 8051 中断系统

(1) 中断源

8051 单片机提供的 5 个中断源中，包括两个外部中断源 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$ ，两个时钟溢出中断 TF0 和 TF1 ，以及一个片内串口中断请求 TI 或 RI 。寄存器 TCON 和 SCON 提供相应位用于锁存这些中断源。

定时溢出中断的产生原理为：定时/计数器 T0 和 T1 计数产生溢出时，由硬件将 TF0 和 TF1 置 1，同时向 CPU 请求中断，CPU 相应中断后，将 TF0 、 TF1 用硬件清 0。 TF0 和 TF1 位于 TCON 寄存器的位置如下：

D7	D6	D5	D4	D3~D0
TF1		TF0		

外部中断源则是指由单片机的 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$ 引脚输入的中断请求信息。 TCON 的低 4 位用于外部请求中断源的控制，其各位如下：

D7~D4	D3	D2	D1	D0
	IE1	IT1	IE0	IT0

- IE1：为外部中断 $\overline{\text{INT1}}$ 请求标志。 $\text{IE1}=1$ 表示外部中断向 CPU 请求中断，当 CPU 响应中断时由硬件清 0。
- IT1：为外部中断 $\overline{\text{INT1}}$ 触发控制位。 $\text{IT1}=0$ ，外部中断为电平触发方式，当 $\overline{\text{INT1}}$ 为低电平时，置位 IE1 。使用电平触发方式时，CPU 响应中断之前， $\overline{\text{INT1}}$ 必须保持低电平有效。 $\text{IT1}=1$ ，外部中断为边沿触发方式，当 CPU 检测到 $\overline{\text{INT1}}$ 有从高电平到低电平的负跳变时，置位 IE1 。使用电平触发方式时， $\overline{\text{INT1}}$ 处的高电平和低电平必须保持在 12 个振荡周期之上才能保证 CPU 检测到从高到低的负跳变。
- IE0：为外部中断 $\overline{\text{INT0}}$ 请求标志。 $\text{IE0}=1$ 表示外部中断向 CPU 请求中断，当 CPU 响应中断时由硬件清 0。
- IT0：为外部中断 $\overline{\text{INT0}}$ 触发控制位。 $\text{IT0}=0$ ，外部中断为电平触发方式，当 $\overline{\text{INT0}}$ 为低电平时，置位 IE0 。使用电平触发方式时，CPU 响应中断之前， $\overline{\text{INT0}}$ 必须保持低电平有效。

平有效。 $IT0=1$, 外部中断为边沿触发方式, 当 CPU 检测到 $\overline{INT0}$ 有从高电平到低电平的负跳变时, 置位 $IE0$ 。使用电平触发方式时, $\overline{INT0}$ 处的高电平和低电平必须保持在 12 个振荡周期之上才能保证 CPU 检测到从高到低的负跳变。

而串口中断是指, 当 8051 单片机内置的串行口接收到或发送完数据时, 向 CPU 发出的中断信息。该中断产生时会根据其产生的原因相应地置 TI 位和 RI 位。CPU 响应中断后并不自动清 0。 TI 位和 RI 位位于寄存器 $SCON$ 中, 其所在位置如下:

D7~D2	D1	D0
	TI	RI

- TI : 串口中断发送标志。当串行口数据发送完毕时置位 TI , 同时向 CPU 发送串口中断请求, CPU 响应中断后不对该位清 0。
- RI : 串口中断接收标志。当串行口数据接收到一个数据时置位 TI , 同时向 CPU 发送串口中断请求, CPU 响应中断后不对该位清 0。

CPU 响应上述各个中断请求后, 会自动根据中断请求的类型自动跳转至相应的中断服务入口处。上述中断源对应的中断服务入口地址如下:

- 外部中断 0: 0003H。
- 定时/计数器 T0 溢出中断: 000BH。
- 外部中断 1: 0013H。
- 定时/计数器 T1 溢出中断: 001BH。
- 串行口中断: 0023H。

(2) 中断控制

使用中断允许寄存器 IE 可以方便地对所有的中断源或对某些特定的中断源进行开启和屏蔽控制。中断允许寄存器 IE 的各位如下:

D7	D6	D5	D4	D3	D2	D1	D0
EA	—	—	ES	ET1	EX1	ET0	EX0

- EA: 为 CPU 的中断开放控制位。EA=1, CPU 开放中断; EA=0, CPU 屏蔽所有的中断。
- ES: 为 CPU 的串行口中断开放控制位。ES=1, CPU 响应串行口中断; ES=0, CPU 禁止串行口中断。
- ET1: 为 CPU 的定时溢出中断 1 开放控制位。ET1=1, CPU 响应定时/计数器 1 溢出产生的中断; ET1=0, CPU 禁止定时/计数器 1 溢出产生的中断。
- EX1: 为 CPU 的外部中断 1 开放控制位。EX1=1, CPU 响应外部中断 1; EX1=0, CPU 禁止外部中断 1。
- ET0: 为 CPU 的定时溢出中断 0 开放控制位。ET0=1, CPU 响应定时/计数器 0 溢出产生的中断; ET0=0, CPU 禁止定时/计数器 0 溢出产生的中断。
- EX0: 为 CPU 的外部中断 0 开放控制位。EX0=1, CPU 响应外部中断 0; EX0=0, CPU 禁止外部中断 0。

(3) 中断优先级

8051 的中断系统提供两级中断优先级, 对每一个中断请求源均可编程为高优先级中断和低优先级中断, 允许实现两级中断嵌套。

8051 的中断响应遵循以下原则:

- 低优先级的中断可以被高优先级的中断所中断, 高优先级的中断则不会被低优先级的

中断打断：

- 当中断得到响应后，将不会被其他的同级中断所中断。

在 8051 中，中断优先级寄存器 IP 专门用于控制各中断源的中断优先级。其各位表示如下：

D7	D6	D5	D4	D3	D2	D1	D0
—	—	—	PS	PT1	PX1	PT0	PX0

- PS：为串行口中断优先级控制位。PS=1，串行口中断设为高优先级中断；PS=0，串行口中断设为低优先级中断。
- PT1：为定时/计数器 T1 中断优先级控制位。PT1=1，定时 / 计数器 T1 中断设为高优先级中断；PT1=0，定时/计数器 T1 中断设为低优先级中断。
- PX1：为外部中断 1 优先级控制位。PX1=1，外部中断 1 设为高优先级中断，PX1=0，外部中断 1 设为低优先级中断。
- PT0：为定时/计数器 T0 中断优先级控制位。PT0=1，定时 / 计数器 T0 中断设为高优先级中断；PT0=0，定时/计数器 T0 中断设为低优先级中断。
- PX0：为外部中断 0 优先级控制位。PX0=1，外部中断 0 设为高优先级中断；PX0=0，外部中断 0 设为低优先级中断。

当同时收到多个同级的中断请求时，CPU 将按照内部的一个优先级顺序响应中断。该优先顺序如表 1-8 所示。

表 1-8 中断源内部优先级

中断源	优先级
外部中断 0	最高
T0 溢出中断	
外部中断 1	
T1 溢出中断	
串行口中断	最低

1.2.4 外部引脚

8051 系列单片机通常采用 3 种封装方式：40 引脚的 DIP 封装、44 引脚的 PLCC 封装和 44 引脚的 QFP 封装。各引脚的分布方式分别如图 1-12、图 1-13 和图 1-14 所示。

下面以 40 引脚的 DIP 封装为例，介绍 8051 单片机各个引脚的功能。

DIP 封装的 40 个引脚包括两个电源引脚、两个外接晶振引脚、4 个控制线引脚和 32 个 I/O 口引脚。

1. 电源部分

- VCC (40 脚)：接+5V 电压。
- GND (20 脚)：接信号地。

DIP

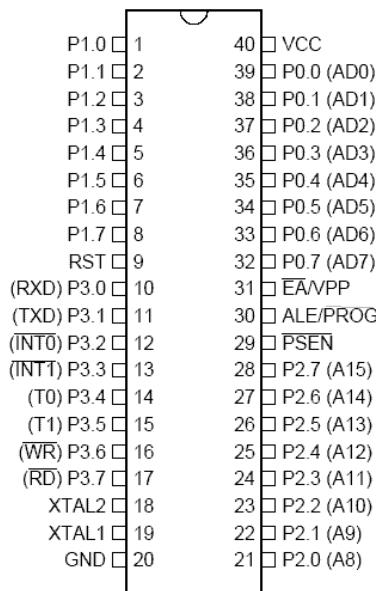


图 1-12 8051 系列单片机 DIP 封装引脚图

PLCC

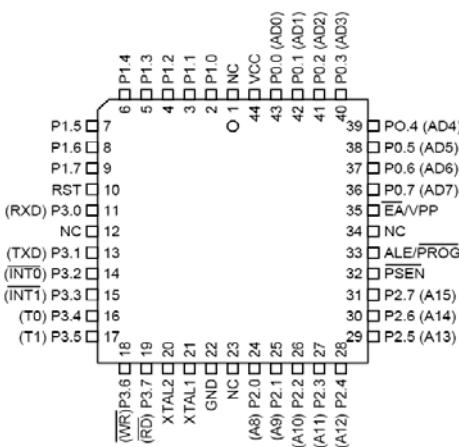


图 1-13 8051 系列单片机 PLCC 封装引脚图

QFP

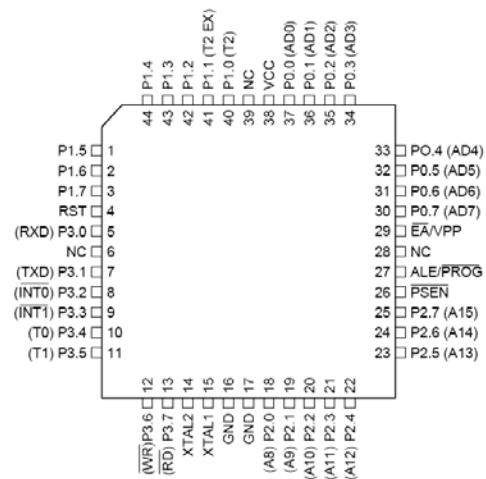


图 1-14 8051 系列单片机 QFP 封装引脚图

2. 外接晶振部分

- XTAL1 (19 脚): 接外部晶振的一个引脚。当采用外部振荡器为单片机提供时钟信号时, 对 HMOS 单片机, 该引脚接地, 对 CMOS 单片机, 该引脚作为驱动端。
- XTAL2 (18 脚): 接外部晶振的另一个引脚, 当采用外部振荡器时, 对 CMOS 单片机, 该引脚接地, 对 HMOS 单片机, 该引脚作为驱动端。图 1-15 所示为 8051 的时钟接入电路。

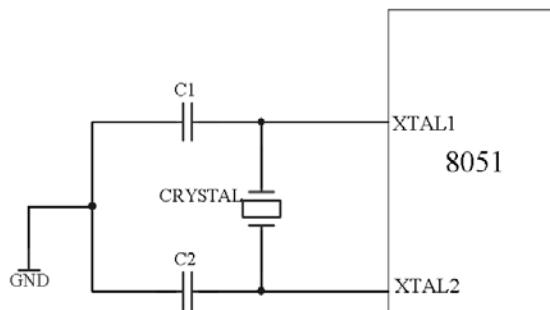


图 1-15 时钟接入电路

3. 控制与复位部分

- RST/VPD (9 脚): 正常工作状态下，该引脚为单片机复位信号输入端。当此引脚上出现两个机器周期的高电平时将使单片机复位。图 1-16 所示为一种最简单的上电复位电路。VCC 掉电期间，该引脚用于接备用电源 (VPD)，以保持内部 RAM 数据不丢失。

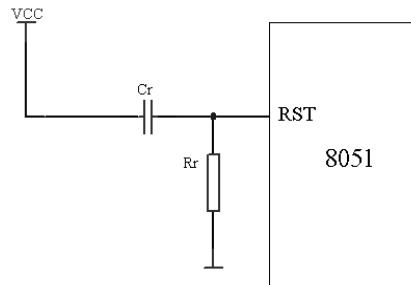


图 1-16 上电复位电路

- ALE/PROG (30 脚): 正常工作状态下，该引脚为允许地址锁存信号输出。当访问外部存储器时，ALE 引脚的输出用于锁存地址的低位字节，由于上电复位后，ALE 端会以振荡器频率的 1/6 周期性地出现正脉冲信号，因此，也可将其用于对外输出的时钟。此外，实际应用中还可以通过检测该引脚的输出脉冲来判断单片机是否处于正常工作状态。对 EPROM 型单片机编程时，该引脚被用作编程脉冲的输入端。
- PSEN (29 脚): 该引脚为外部程序存储器的读选通信号输出。在从外部程序存储器取指令期间，每个机器周期 PSEN 两次有效。但在访问片内程序存储器和外部数据存储器时，不产生 PSEN 信号。
- EA/VPP (31 脚): 正常工作状态下，该引脚为外部程序存储器的选通信号。当 EA 信号为高电平，且程序计数器 (PC) 值超过片内程序存储器地址时，自动转向外部程序存储器。当 EA 信号保持低电平时，直接使用外部程序存储器。对 EPROM 型单片机编程时，该引脚接 +12V 电压。

4. I/O 口部分

- P0 口 (39脚~32脚): 双向 8位三态 I/O 口, 在外接存储器时, 与地址总线低 8位及数据总线复用。P0 可以驱动 8个 LS TTL 负载。
- P1 口 (1脚~8脚): 具有内部上拉电阻的 8位准双向 I/O 口, 该接口输出不包含高阻态, 输入不能锁存。P1 可以驱动 4个 LS TTL 负载。
- P2 口 (21脚~28脚): 具有内部上拉电阻的 8位准双向 I/O 口, 在访问外部存储器时, 作为高 8位地址总线。P2 可以驱动 4个 LS TTL 负载。
- P3 口 (10脚~17脚): 具有内部上拉电阻的 8位准双向 I/O 口, 在 8051 系列中, P3 口的 8个引脚还用于专门的功能——复用双功能口。P3 口可以驱动 4个 LS TTL 负载。P3 口作为第一功能使用时, 就是普通的 I/O 口, 作为第二功能使用时, 各个引脚的定义如表 1-9 所示。

表 1-9 P3 口各引脚第二功能定义

口线	引脚	第二功能	说明
P3.0	10	RXD	串行口接收端
P3.1	11	TXD	串行口发送端
P3.2	12	<u>INT0</u>	外部中断 0
P3.3	13	<u>INT1</u>	外部中断 1
P3.4	14	T0	定时/计数器 0
P3.5	15	T1	定时/计数器 1
P3.6	16	<u>WR</u>	外部数据存储器写选通信号
P3.7	17	<u>RD</u>	外部数据存储器读选通信号

在 8051 单片机的 40 个引脚中, 除去电源、接地、时钟接入和片外程序存储器片选外, 其他引脚构成了 8051 单片机的片外总线结构, 包括地址总线 (AB)、数据总线 (DB) 和控制总线 (CB), 如图 1-17 所示。

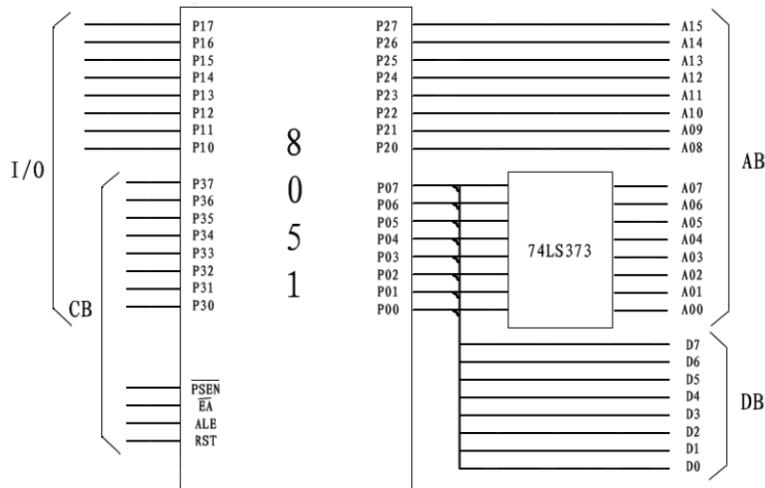


图 1-17 8051 片外总线结构

1.2.5 系统资源扩展

在单片机的实际应用中，单片机自带的资源往往无法满足实际应用的需要，这就需要对单片机系统进行系统扩展。本节主要介绍 8051 单片机的基本系统扩展，包括程序存储器的扩展、数据存储器的扩展以及 I/O 口的扩展。

1. 程序存储器的扩展

8051 的程序存储空间最大寻址空间为 64kB，片内自带 4kB 的 ROM 或 EEPROM，当片内 ROM 不够用时，需扩展程序存储器。

由于 8051 单片机的 P0 口是分时复用的，因此在进行程序存储器扩展时，需要使用地址锁存器将地址信号从地址/数据总线中分离出来。通常使用 8D 锁存器 74LS373 或是带清除端的 8D 锁存器 74LS273 来作地址锁存器，注意在使用中不同锁存器的地址锁存信号 ALE 的接入方法是不同的。

对 74LS373，当三态门使能信号 \overline{OE} 为低电平时，三态门导通，允许 Q0~Q7 输出， \overline{OE} 为高电平时，输出悬空。当 74LS373 用作地址锁存器时，应使 \overline{OE} 为低电平导通输出，此时锁存使能端 C 为高电平时，输出 Q0~Q7 状态与输入端 D1~D7 状态相同；当 C 发生负跳变时，输入端 D0~D7 数据锁入 Q0~Q7。因此在使用 74LS373 时，8051 的 ALE 信号可以直接与 74LS373 的 C 相连。

对 74LS273，只有清除端为高电平时才具有锁存功能，此时锁存控制端 CLK 在上升沿锁存数据，因此在使用 74LS273 时，8051 的 ALE 信号需接反相器后才可以与 74LS273 的 CLK 端相连。

74LS373 和 74LS273 的 ALE 信号接入方法分别如图 1-18 和图 1-19 所示。

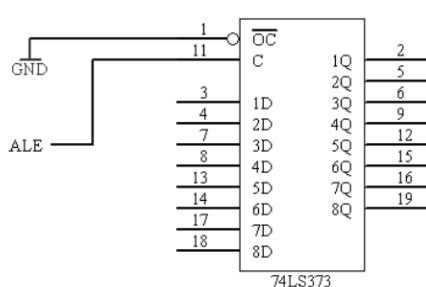


图 1-18 74LS373 作地址锁存器

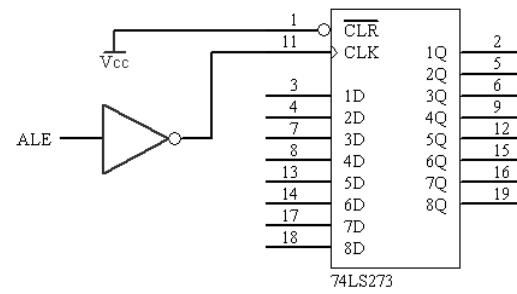


图 1-19 74LS273 作地址锁存器

外接程序存储器一般采用 EPROM 芯片，典型的 EPROM 芯片包括 Intel 公司的 2764A (8kB×8)、27128A (16kB×8)、27256 (32kB×8)、27512 (64kB×8) 等。图 1-20、图 1-21 所示分别为 2764A 和 27256 芯片的管脚图。

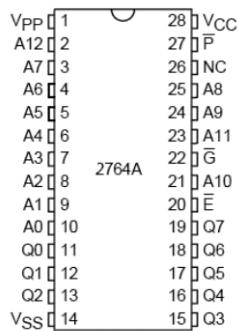


图 1-20 2764A 管脚图

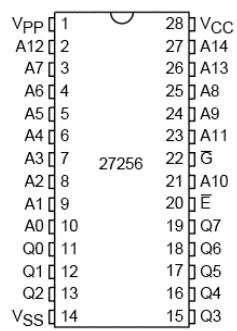


图 1-21 27256 管脚图

8051 外接 27256 芯片的电路图如图 1-22 所示。

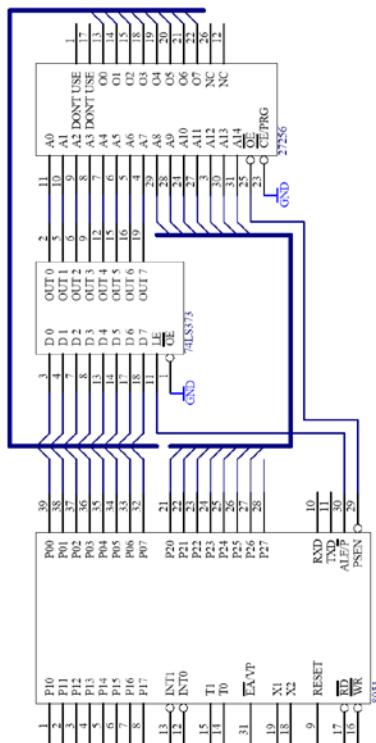


图 1-22 8051 外接 27256 电路图

2. 数据存储器的扩展

8051 系列单片机的内部仅有 128Byte 的 RAM，在实时数据采集和处理等领域，仅靠片内提供的 RAM 容量是远远不够的，这就需要扩展外部数据存储器。常用的数据存储器有静态 RAM 和动态 RAM 两种。在对外部 RAM 进行读写时，数据存储器使用 **WR**、**RD** 控制线，以与外部扩展 ROM 的读写相区别。

在数据存储器的扩展中，静态 RAM 最为常用，典型的静态 RAM 芯片包括 6116 (2kB×8)、6264 (8kB×8)、62256 (32kB×8) 等。6116、6264 及 62256 的芯片管脚分别如图 1-23、图

1-24、图 1-25 所示。

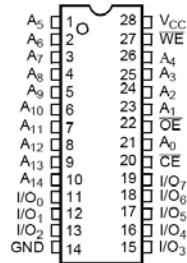
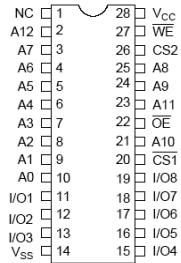
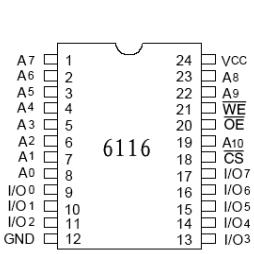


图 1-23 6116 管脚图

图 1-24 6264 管脚图

图 1-25 62256 管脚图

图 1-26 所示为 8051 外接 6264 静态 RAM 的典型电路图。

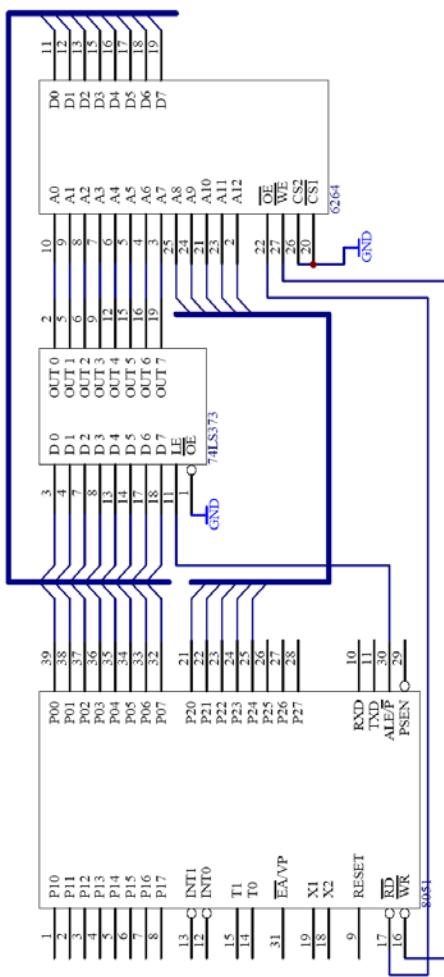


图 1-26 8051 外接 6264 电路图

3. 8051 外部 I/O 口扩展

当单片机系统中的外接资源过多时，8051 本身提供的 I/O 口不能满足应用本身的需求，

这时就要对单片机的 I/O 口进行扩展。由于 8051 的外部 RAM 和 I/O 口是统一编址的，因此，用户可以把外部 RAM 空间的一部分用于扩展外部 I/O 的地址空间。在实际应用中，多使用 Intel 公司的 8255A 和 8155 芯片实现 8051 单片机 I/O 口的扩展，本节将以 8255A 为例介绍单片机 I/O 口的扩展。

8255A 是 Intel 公司生产的可编程输入/输出接口芯片。它具有 3 个 8 位的并行 I/O 口，提供 3 种可编程的工作方式，因此使用灵活方便、通用性强，适用于许多领域的中间接口电路。8255A 的管脚如图 1-27 所示。

8255A 的管脚功能如下：

- D7~D0 三态双向数据线，用来传递数据信息。
- CS 片选信号线，低电平有效。
- RD、WR 读、写信号线，低电平有效，控制数据的读出和写入。
- PA7~PA0、PB7~PB0、PC7~PC0 A 口、B 口和 C 口的输入/输出线。
- RST 复位信号线。
- A1、A0 地址线，用来选择 8255A 内部端口。
- GND 地线。

8255A 的 3 个 I/O 口可以分为两组，A 组包括 A 口和 C 口的上半部分，B 组包括 B 口和 C 口的下半部分。当地址线 A1、A0 为 11 时，可以访问 8255A 的状态控制寄存器。可以以写入控制命令字的方法来决定两组端口的工作方式，也可以根据控制字的要求对 C 口按位置 1 或清 0。8255A 支持 3 种工作方式：基本输入/输出方式、选通工作方式和双向传送方式。关于 8255A 的具体使用方法，读者可以自行参考有关资料，这里就不在赘述了。图 1-28 所示为 8255A 与 8051 连接的典型电路图。

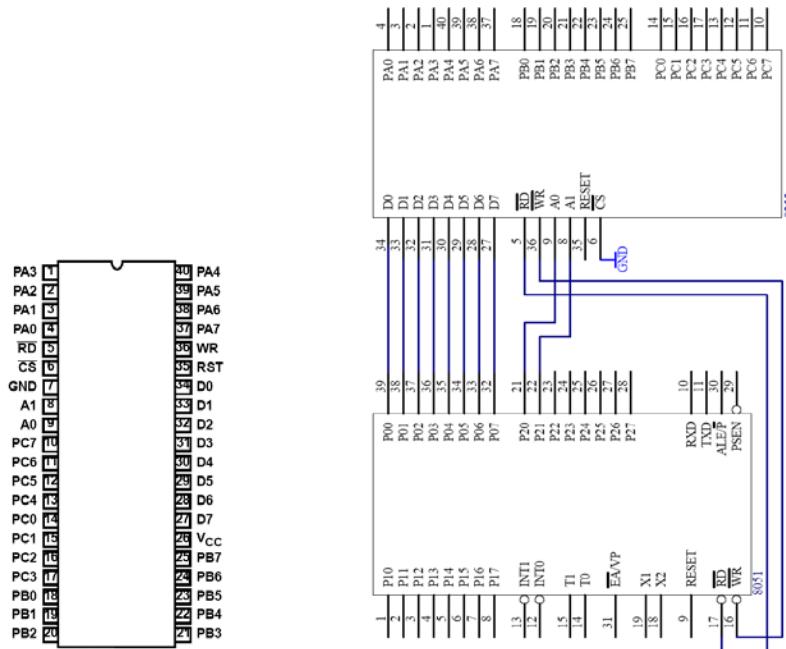


图 1-27 8255A 管脚图

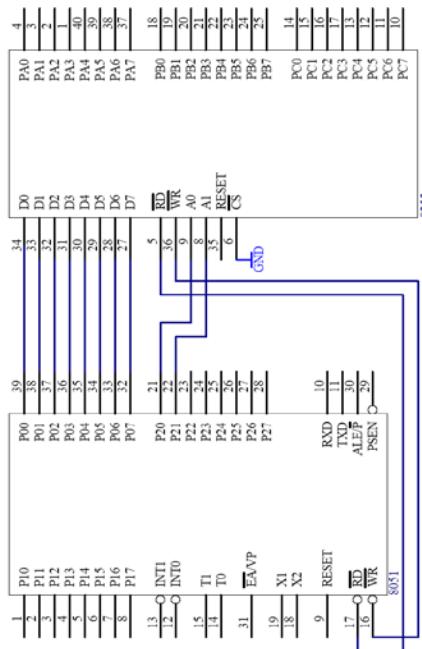


图 1-28 8051 外接 8255A 电路图

1.3 单片机的编程方法

8051 单片机的应用程序设计，既可以使用汇编语言，也可以使用 C51 语言。两种方法各有其优势。汇编语言与硬件紧密相关，可以方便地实现诸如中断管理以及模拟/数字量的输入/输出等功能，且占用系统资源小、执行速度快。但当应用程序达到一定规模后，由于汇编语言的代码可读性较差，将增加编写和阅读代码的难度，不利于应用系统的升级和维护。使用 C51 语言进行程序设计虽然相对于汇编语言代码效率有所下降，但可以方便地实现程序设计模块化，代码结构清晰、可读性强，易于维护、更新和移植，适合较大规模的单片机程序设计。近年来，随着 C51 语言的编译器性能的不断提高，在绝大多数的应用环境下，C51 程序的执行效率已经非常接近汇编程序，因此，使用 C51 进行单片机程序设计已经成为单片机程序设计的主流选择之一。

C51 是在完全支持标准 C 全部指令的基础上添加了许多用来优化 8051 指令结构的 C 的扩展指令而形成的。其程序结构也类似于标准 C 程序的编写，因此，本书后续内容将从 C 语言基础开始，由浅入深地对 C51 程序的设计和编写方法进行详细说明。

另外，对于 C51 程序的编译执行，本书将以当前最普遍使用的 KEIL 51 为例进行说明。KEIL 51 是德国 KEIL 公司开发的单片机 C 语言编译器，其前身是 FRANKLIN C51，现在的最新版本 V6 功能已经相当不错，特别是兼容 ANSI C 后又增加很多与硬件密切相关的编译特性，使得在 8051 系列单片机上开发应用程序更为方便和快捷。在 KEIL 51 下，μ Vision2 是一种集成化的文件管理编译环境，集成了文件编辑处理、编译链接、项目管理、窗口、工具引用和软件仿真调试等多种功能，是相当强大的 C51 开发工具。在 μ Vision2 的仿真功能中，有两种仿真模式：软件模拟方式和目标板调试方式。在软件模拟方式下，不需要任何 8051 单片机硬件即可完成用户程序仿真调试，极大地提高了用户程序开发效率。在目标板调试方式下，用户可以将程序装到自己的 8051 单片机系统板上，利用 8051 的串口与 PC 机进行通信来实现用户程序的实时在线仿真。

在本书中不对 KEIL 51 和 μ Vision2 两个术语做严格的区分，一般来说，都指的是 μ Vision2 集成开发环境。

第 2 章 C 语言编程基础

C51 语言基本上是以 PC 机上的 ANSI C 为标准发展起来的。不同厂家，不同版本的 C51 编译器对 C 语言的支持功能不一样——有很多版本的编译器已经支持 C++，而有些版本的编译器就连像结构这样稍稍复杂一些的数据类型都不支持。本章将介绍 C 语言中常用的所有数据类型和功能，至于到具体的编译器上哪些数据类型可用，哪些不支持，请读者阅读厂家的说明书。

2.1 基本概念

本节首先对 C 语言做简要介绍，以作为了解后续各节的详细内容的框架。目的是通过实际的程序向读者介绍 C 语言的本质要素，而不是马上深入到具体细节、规则及例外情况中。

2.1.1 概述

学习新的程序设计语言的最佳途径是编写程序。对于所有语言，编写的第一个程序可能都是相同的。例如，打印如下单词：

```
hello, world
```

在初学语言时这是一个很大的障碍，要越过这个障碍，首先必须建立程序文本，然后成功地对它进行编译，并装入、运行，最后再观察输出的结果。只要把这些操作细节掌握了，其他内容就比较容易了。

在 C 语言中，用如下程序打印“hello, world”：

```
#include <stdio.h>
main() {
    printf("hello, world\n");
}
```

下面我们对这个程序本身做一些解释说明。每一个 C 程序，不论大小如何，都由函数和变量组成。函数中包含若干用于指定所要做的计算操作的语句，而变量则用于在计算过程中存储有关值。在本例中，函数的名字为 main。一般而言，可以给函数任意命名，但在 C 语言中，main 是一个特殊的函数名，每次运行程序都从名为 main 的函数的起点开始执行。这意味着每一个程序都必须包含一个 main 函数。

main 函数通常要调用其他函数来协助其完成某些工作，调用的函数有些是程序人员自己编写的，有些则由系统函数库提供。上述程序的第一行

```
#include <stdio.h>
```

用于告诉编译程序在本程序中包含标准输入输出库的有关信息。许多 C 源程序的开始都

包含这一行。

在函数之间进行数据通信的一种方法是让调用函数向被调用函数提供一串叫做变元的值。函数名后面的一对圆括号用于把这一串变元（变元表）括起来。在本例子中，所定义的 main 函数不要求任何变元，故用空变元表()表示。函数中的语句用一对花括号 {} 括起来。本例中的 main 函数只包含一个语句：

```
printf("hello, world\n");
```

当要调用一个函数时，先要给出这个函数的名字，再紧跟用一对圆括号括住的变元表。上面这个语句就是用变元"hello, world\n"来调用函数 printf。printf 是一个用于打印输出的库函数，在本例中，它用于打印用引号括住的字符串。用双引号括住的字符序列叫做字符串或字符串常量，如"hello, world\n"就是一个字符串。目前仅使用字符串作为 printf 及其他函数的变元。在 C 语言中，字符序列\n 表示换行符，在打印时它用于指示从下一行的左边换行打印。如果在字符串中遗漏了\n（一个值得做的试验），那么输出打印不换行。在 printf 函数的变元中必须用\n 引入换行符，如果用程序中的换行来代替\n，如：

```
printf("hello,world  
");
```

那么 C 编译器将会产生一个错误信息。

至于如何运行这个程序，在不同的操作系统上有所不同。由于本书主要目的在于教会读者使用 C51，所以本书中所有的 C 语言的例子都在 KEIL51 的开发环境中来实现。在这里，我们请读者注意，由于单片机自己构成一个系统，并不像 PC 机一样有标准的输入输出，所以我们上面的程序需要做一些改动。在单片机中，可用作输入输出口的有普通 I/O 引脚和串口。一般来说，使用串口作为输出口更为通用。

在 KEIL51 的 C 编译器中，也提供了一个名为 printf 的库函数，但是该函数是将字符输出到串口上，读者可以在 Debug 模式下使用菜单“View”下的“Serial Window”子菜单，在弹出的窗口下观察输出的结果。不过实践过程中要使用这个 printf 函数，还可能需要对 51 单片机的串口的若干寄存器作一些初始化的设置，因此在 KEIL51 中的程序与上面的例程有简单不同的不同。

[例 2-1] 打印“hello, world”

程序清单如下：

```
#include <REG52.H> //特殊寄存器的头文件
//专供 8051 扩展系列的单片机使用

#include <stdio.h> //I/O 库函数原型声明
#ifndef MONITOR51 //是否需要使用 Monitor-51 调试
char code reserve [3] _at_ 0x23; //如果是，留下该空间供串口中断使用
#endif //停止程序执行

//-----
C 程序的主函数，在堆栈等初始化完成
以后，程序从此处开始执行
```

```

void main (void) {
    //-----
    //设定串口的数据传输速率为 1200bit/s，晶振频率为 16MHz

    #ifndef MONITOR51
        SCON = 0x50;           // SCON: 模式 1, 8-bit 异步串口通信
        TMOD |= 0x20;          // TMOD: 定时器 1 为模式 2, 8-bit 自动装载方式
        TH1 = 221;             // TH1:1200bit/s 的装载值@ 16MHz
        TR1 = 1;               // TR1:timer1 运行
        TI = 1;                // TI:设置为 1, 以发送第一个字节
    #endif
    //-----
    //注意: 由于没有操作系统来接受 main 函
    //数的返回值, 所以对一个嵌入式系统来说,
    //main 函数永远不会被退出。它必须有一个
    //循环来保证程序不会被终止
    //-----
    while (1) {
        printf ("Hello World\n"); // 打印 "Hello World"
    }
}

```

读者只要在 KEIL51 的 debug 环境下, 通过 Serial Window #1 窗口就可以看到输出的内容, 如图 2-1 所示。对 KEIL51 的开发环境的更加详细的操作方法, 请读者参看第 5 章的内容。

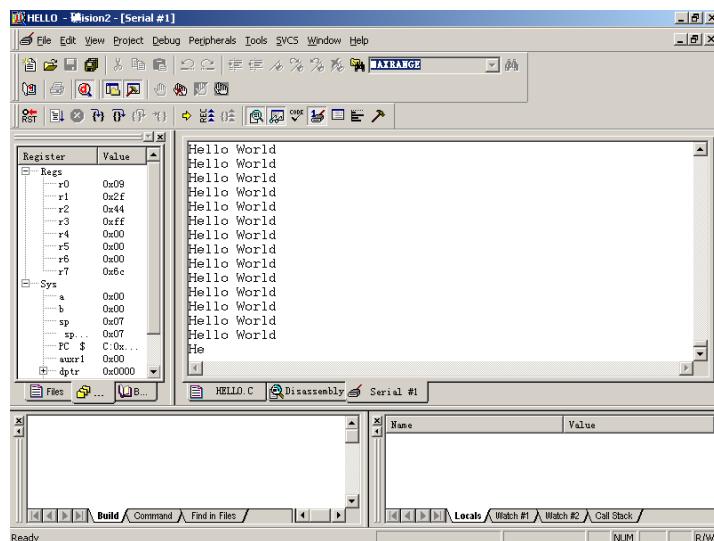


图 2-1 KEIL51 环境中 Serial Window #1 中的输出内容

上面的程序段中 main() 函数之前的内容主要是包含库函数的一些声明, 在 main() 中的

while(1) {……} 循环之前，则是对串口的一些初始化的操作，真正的主程序在 while(1) 中。该程序完整源代码请参看 www.cs-book.com 页面上相应目录上 ex01 目录里 hello.uv2 项目中 hello.c 的内容。关于 KEIL51 的使用方法以及 51 系列单片机的特殊寄存器，请参见本书的其他章节。为了使读者将注意力放在 C 语言本身的语法语义上，我们在本章后面的其他例程序中都不再含有 C51 的特殊寄存器初始化的内容，而只列出我们所关心的正在讨论的部分。

由此可见，KEIL51 中的 printf 函数与 PC 机上的 ANSI C 的 printf 函数用法相同，而且在 KEIL51 仿真环境中查看结果也很方便。

printf 函数永远不会自动换行，我们可以多次调用这个函数来分几次打印一行输出。上面给出的第一个程序也可以写成如下形式：

```
#include <stdio.h>
main() {
    printf("hello, ");
    printf("world");
    printf("\n");
}
```

它所产生的输出与前面一样。

请注意，\n 只表示一个字符。诸如\n 等换码序列表示不能打印或不可见字符提供了一种通用可扩充机制。除此之外，C 语言还提供其他的换码序列，将分别在本章后面的小节中介绍。

2.1.2 变量与算术表达式

[例 2-2] 打印华氏温度与摄氏温度对照表

转换公式为： ${}^{\circ}\text{C} = (5/9)({}^{\circ}\text{F} - 32)$ ，具体数值如表 2-1 所示。

表 2-1 华氏温度与摄氏温度对照表

华氏温度	摄氏温度
0	-17
20	-6
40	4
60	15
80	26
100	37
120	48
140	60
160	71
180	82
200	93
220	104
240	115
260	126
280	137

这个程序本身仍只由一个名为 main 的函数的定义组成，它要比前面用于打印

“hello, world”的程序长，但并不复杂。这个程序中引入了一些新的概念，包括注解、说明、变量、算术表达式、循环以及格式输出。该程序如下：

```
#include <REG52.H>           //特殊寄存器的头文件
//专供8051扩展系列的单片机使用

#include <stdio.h>           //I/O库函数原型声明
#include <float.h>

#ifndef MONITOR51             //是否需要使用Monitor-51调试
char code reserve [3] _at_ 0x23; //如果是，留下该空间供串口中断使用
#endif                         //停止程序执行

//-----
C程序的主函数，在堆栈等初始化完成
以后，程序从此处开始执行

//打印对 fahr = 0, 20, ..., 300
打印华氏温度与摄氏温度对照表
void main (void) {

    int fahr, celsius;
    int lower, upper, step;
//-----
//设定串口的数据传输速率为1200bit/s，晶振频率为16MHz
// -----
#ifndef MONITOR51
    SCON = 0x50;           // SCON: 模式1, 8-bit异步串口通信
    TMOD |= 0x20;          // TMOD: 定时器1为模式2, 8-bit自动装载方式
    TH1   = 221;            // TH1: 1200bit/s的装载值@ 16MHz
    TR1   = 1;               // TR1: timer1运行
    TI    = 1;               // TI: 设置为1, 以发送第一个字节
#endif

//-----
//注意：由于没有操作系统来接受main函
//数的返回值，所以对一个嵌入式系统来说，
//main函数永远不会被退出。它必须有一个
//循环来保证程序不会被终止
```

```

//-----
lower = 0;           // 温度表的下限
upper = 300 ;        // 温度表的上限
step = 20;           // 步长
fahr = lower;
while (fahr <= upper) {
    celsius = 5 * (fahr-32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
}
while(1);
}

```

其中每一行在“//”之后的字符序列在编译时被忽略掉，它们可以在程序中自由地使用，目的是为了使程序更易于理解。这种形式叫做注解，用于解释某一段代码是做什么的。注解可以出现在任何空格、制表符或换行符可以出现的地方。例如“//温度表的下限”，“//温度表的上限”等，在 C 语言中还可以用“//注解”的形式来进行注解。

在 C 语言中，所有变量都必须先声明后使用，声明通常放在函数开始处的可执行语句之前。声明用于声明变量的性质，它由一个类型名与若干所要声明的变量组成，例如：

```

int fahr, celsius;
int lower, upper, step;

```

其中，类型 int 表示所列变量为整数变量，与之相对，float 表示所列变量为浮点变量（浮点数可以有小数部分）。除 int 与 float 之外，C 语言还提供了其他一些基本数据类型，包括以下类型：

- char：单字节字符。
- short：短整数。
- long：长整数。
- double：双精度浮点数。

另外，还有由这些基本数据类型组成的数组、结构与联合类型、指向这些数据类型的指针类型以及返回这些数据类型的函数，这些内容将在后面的章节分别介绍。

上面温度转换程序计算以 4 个赋值语句开始，用于为变量设置初值。各个语句均以分号结束：

```

lower= 0;
upper= 300 ;
step= 20;
fahr= lower;

```

温度转换表中的每一行均以相同的方式计算，故可以用循环语句来重复产生各行输出，每行重复一次。这就是 while 循环语句的用途：

```

while(fahr<= upper) {
    .....
}

```

while 循环语句的执行步骤如下：首先测试圆括号中的条件。如果条件为真（fahr 小于等于 upper），则执行循环体（括在花括号中的三个语句）。然后再重新测试该条件，如果为真，则再次执行该循环体。当该条件测试为假（fahr 大于 upper）时，循环结束，继续执行跟在该循环语句之后的下一个语句。在本程序中，循环语句后再没有其他语句，因此整个程序终止执行。

while 语句的循环体可以是用花括号括住的一个或多个语句（如上面的温度转换程序），也可以是不用花括号括住的单个语句，例如：

```
while (i < j)
    i = 2 * i;
```

在这两种情况下，我们总是把由 while 控制的语句向里缩入一个制表位（在书中以 4 个空格表示），这样就可以很容易地看出循环语句中包含哪些语句。这种缩进方式强化了程序的逻辑结构。尽管 C 编译程序并不关心程序的具体形式，但使程序在适当位置采用缩进空格的风格对于使程序更易于为人们阅读是很重要的。我们建议每行只写一个语句，并在运算符两边各放一个空格字符以使运算组合更清楚。花括号的位置不太重要，尽管每个人都有他所喜爱的风格。我们从一些比较流行的风格中选择了一种。读者可以选择合适的风格并一直使用它。

绝大多数任务都是在循环体中做的。循环体中的赋值语句

```
celsius=5*(fahr-32)/9;
```

用于求与指定华氏温度所对应的摄氏温度值并将值赋给变量 celsius。在该语句中，之所以把表达式写成先乘 5 然后再除以 9 而不直接写成 $5/9$ ，是因为在 C 语言及其他语言中，整数除法要进行截取，即结果中的小数部分被丢弃。由于 5 和 9 都是整数， $5/9$ 相除后所截取得的结果为 0，故这样所求得的所有摄氏温度都变成 0。

这个例子也对 printf 函数的工作功能做了更多的介绍。printf 是一个通用输出格式化函数，在后面的小节中将对此做详细介绍。该函数的第一个变元是要打印的字符串，其中百分号（%）指示用其他变元（第 2、第 3 个…变元）之一对其进行替换，以及打印变元的格式。例如，%d 指定一个整数变元，语句

```
printf("%d\t%d\n", fahr, celsius);
```

用于打印两个整数 fahr 与 celsius 值并在两者之间空一个制表位 (\t)。

printf 函数第 1 个变元中的各个%分别对应于第 2 个、第 3 个…第 n 个变元，它们在数目和类型上都必须匹配，否则将出现错误。顺便指出，printf 函数并不是 C 语言本身的一部分，C 语言本身没有定义输入输出功能。printf 是标准库函数中一个有用的函数，标准库函数一般在 C 程序中都可以使用。ANSI 标准中定义了 printf 函数的行为，从而其性质在使用每一个符合标准的编译程序与库中都是相同的。而在 KEIL 的 C51 中，printf、scanf 这些函数的输出都传送到串口上，用户只要对其稍作修改就可以用在自己的外挂键盘或者显示器上。

上面这个温度转换程序存在一个较为严重的问题是，由于使用的是整数算术运算，故所求得的摄氏温度不很精确，

例如，与 0°F 对应的精确的摄氏温度为 -17.8°C ，而不是 -17°C 。为了得到更精确的答案，应该用浮点算术运算来代替上面的整数算术运算。这就要求对程序做适当修改。下面给出这个程序的第二个版本：

[例 2-3] 使用浮点类型打印华氏温度与摄氏温度对照表

```

#include <REG52.H> //特殊寄存器的头文件
//专供 8051 扩展系列的单片机使用

#include <stdio.h> //I/O 库函数原型声明
#include <float.h>

#ifndef MONITOR51 //是否需要使用 Monitor-51 调试
char code reserve [3] _at_ 0x23; //如果是，留下该空间供串口中断使用
#endif //停止程序执行

//-----
//C 程序的主函数，在堆栈等初始化完成
//以后，程序从此处开始执行
//-----

// 对 fahr = 0, 20, ..., 300 打印华氏温度与摄氏温度对照表;
//浮点数版本
void main (void) {
    float fahr, celsius;
    int lower, upper, step;
//-----
//设定串口的数据传输速率为 1200bit/s，晶振频率为 16MHz
// -----
#ifndef MONITOR51
    SCON = 0x50; // SCON: 模式 1, 8-bit 异步串口通信
    TMOD |= 0x20; // TMOD: 定时器 1 为模式 2, 8-bit 自动装载方式
    TH1 = 221; // TH1:1200bit/s 的装载值@ 16MHz
    TR1 = 1; // TR1:timer1 运行
    TI = 1; // TI:设置为 1, 以发送第一个字节
#endif

//-----
//注意：由于没有操作系统来接受 main 函数的返回值，所以对一个嵌入式系统来说，main 函数永远不会被退出。它必须有一个循环来保证程序不会被终止
//-----
    lower = 0; // 温度表的下限
}

```

```

upper = 300 ; // 温度表的上限
step = 20; // 步长
fahr = lower;
while (fahr <= upper) {
    celsius = (5.0 / 9.0) * (fahr-32.0);
    printf("%3.0f %6.1f\n", fahr, celsius);
    fahr = fahr + step;
}
while (1) {}
}

```

这个版本与前一个版本基本相同，只是把 fahr 与 celsius 说明成 float 浮点类型，转换公式的表达式也更自然。在前一个版本中，之所以不用 5/9 是因为按整数除法它们相除截取的结果为 0。在此版本中 5.0/9.0 是两个浮点数相除，不会有截取的现象。

如果某个算术运算符的运算分量均为整数类型，那么就执行整数运算。然而，如果某个算术运算符有一个浮点运算分量和一个整数运算分量，那么这个整数运算分量在开始运算之前会被转换成浮点类型。例如，对于表达式 fahr-32，32 在运算过程中将被自动转换成浮点数再参与运算。不过，在写浮点常数时最好还是把它写成带小数点的形式，以便使该浮点常数取的是整数值，因为这样可以强调其浮点性质，便于人们阅读。

请注意，赋值语句

```
fahr = lower;
```

与条件测试

```
while ( fahr <= upper )
```

也都是以自然的方式执行，即在运算之前先把 int 转换成 float。

printf 函数还可以识别如下格式说明：表示八进制数的%o、表示十六进制数的%x、表示字符的%c、表示字符串的%s 以及表示百分号%本身的%%等等。

2.1.3 for 语句

对于一个特定任务，可以用多种方法来编写程序。下面的例程是前面讲述的温度转换程序的一个变种：

[例 2-4] 另一个打印温度对照表的程序

```

// 打印华氏与摄氏温度对照表
#include <REG52.H> //特殊寄存器的头文件
//专供 8051 扩展系列的单片机使用

#include <stdio.h> //I/O 库函数原型声明
#include <float.h>

#ifndef MONITOR51 //是否需要使用 Monitor-51 调试

```

```

char code reserve [3] _at_ 0x23;           //如果是, 留下该空间供串口中断使用
#endif                                     //停止程序执行

//-----
//C 程序的主函数, 在堆栈等初始化完成
//以后, 程序从此处开始执行
//-----
void main (void) {

    int fahr;
//-----
//设定串口的数据传输速率为 1200bit/s, 晶振频率为 16MHz
// -----
#ifndef MONITOR51
    SCON = 0x50;                         // SCON: 模式 1, 8-bit 异步串口通信
    TMOD |= 0x20;                        // TMOD: 定时器 1 为模式 2, 8-bit 自动装载方式
    TH1   = 221;                          // TH1:1200bit/s 的装载值@ 16MHz
    TR1   = 1;                            // TR1:timer1 运行
    TI    = 1;                            // TI:设置为 1, 以发送第一个字节
#endif

//-----
//注意: 由于没有操作系统来接受 main 函
//数的返回值, 所以对一个嵌入式系统来说,
//main 函数永远不会被退出。它必须有一个
//循环来保证程序不会被终止
//-----
    for ( fahr = 0; fahr <= 300; fahr = fahr + 20 )
        printf ( "%3d %.1f\n", fahr, (5.0 / 9.0) * (fahr - 32) );
}

```

这个版本与前一个版本执行的结果相同, 但看起来其过程有些不同。一个主要的变化是它删去了大部分变量, 只留下了一个 fahr, 其类型为 int。本来用变量表示的下限、上限与步长都在新引入的 for 语句中作为常量出现, 用于求摄氏温度的表达式现在已变成了 printf 函数的第 3 个变元, 而不再是一个独立的赋值语句。这最后一点变化说明了一个通用规则: 在所有可以使用某个类型的变量的值的地方, 都可以使用该类型的更复杂的表达式。由于 printf 函数的第 3 个变元必须为与%6.1f 匹配的浮点值, 在这里可以使用的是任何浮点表达式。

for 语句是一种循环语句, 是 while 语句的推广。如果将其与前面介绍的 while 语句比
- 34 -

较，就会发现其操作要更清楚一些。在圆括号内共包含3个部分，它们之间用分号隔开。第一部分 `fahr=0` 是初始化部分，仅在进入循环前执行一次。第二部分是用于控制循环的条件测试部分：`fahr<= 300`。这个条件要进行求值。如果所求得的值为真，那么就执行循环体（本例循环体中只包含一个 `printf` 函数调用语句）。然后再执行第三部分 `fahr=fahr+20` 加步长，并再次对条件求值。一旦求得的条件值为假，那么就终止循环的执行，否则则继续执行循环。

像 `while` 语句一样，`for` 循环语句的循环体可以是单个语句，也可以是用花括号括住的一组语句。初始化部分（第一部分）、条件部分（第二部分）与加步长部分（第三部分）均可以是任何表达式。至于在 `while` 与 `for` 这两个循环语句中使用哪一个，这是随意的，主要看使用哪一个更能清楚地描述问题。`for` 语句比较适合描述这样的循环：初值和增量都是单个语句并且是逻辑相关的，因为 `for` 语句把循环控制语句放在一起，比 `while` 语句更紧凑。

2.1.4 符号常量

在结束对温度转换程序的讨论之前，再来看看符号常量。把 300、20 这样的数值埋在程序中并不是一种好的习惯，这些数几乎没有向以后可能要阅读该程序的人提供什么信息，而且使程序的修改变得困难。处理这种幻数的一种方法是赋予它们有意义的名字。`#define` 指令就用于把符号名字（或称为符号常量）定义为一特定的字符串，其表达式如下：

```
#define 名字替换文本
```

此后，所有在程序中出现的在“`#define`”中定义的名字，该名字既没有用引号括起来，也不是其他名字的一部分，都用所对应的替换文本替换。这里的名字与普通变量名有相同的形式：它们都是以字母打头的字母或数字序列。替换文本可以是任何字符序列，而不仅限于数。

[例 2-5] 使用符号常量来写温度对照表程序

```
#include <REG52.H> //特殊寄存器的头文件
//专供 8051 扩展系列的单片机使用

#include <stdio.h> //I/O 库函数原型声明
#include <float.h>

#ifndef MONITOR51 //是否需要使用 Monitor-51 调试
char code reserve [3] _at_ 0x23; //如果是，留下该空间供串口中断使用
#endif //停止程序执行

#define LOWER 0 // 表的下限
#define UPPER 300 // 表的上限
#define STEP 20 // 步长
//-----
//C 程序的主函数，在堆栈等初始化完成
```

```

//以后，程序从此处开始执行
//-----
// 打印华氏-摄氏温度对照表
void main (void) {

    int fahr;
//-----
//设定串口的数据传输速率为 1200bit/s，晶振频率为 16MHz
// -----
#ifndef MONITOR51
    SCON = 0x50;           // SCON: 模式 1, 8-bit 异步串口通信
    TMOD |= 0x20;          // TMOD: 定时器 1 为模式 2, 8-bit 自动装载方式
    TH1   = 221;            // TH1:1200bit/s 的装载值@ 16MHz
    TR1   = 1;              // TR1:timer1 运行
    TI    = 1;              // TI:设置为 1, 以发送第一个字节
#endif
//-----
//注意：由于没有操作系统来接受 main 函
//数的返回值，所以对一个嵌入式系统来说，
//main 函数永远不会被退出。它必须有一个
//循环来保证程序不会被终止
//-----
for ( fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP )
    printf ( "%3d %.1f\n", fahr, (5.0 / 9.0) * (fahr - 32) );

    while (1) {};
}

```

LOWER、UPPER 与 STEP 等几个量是符号常量，而不是变量，故不需要出现在说明中。符号常量名通常用大写字母拼写，这样就可以很容易与用小写字母拼写的变量名相区别。请注意，#define 指令行的末尾没有分号。

2.2 数据类型、运算符和表达式

从本节开始，将会对 C 语言中的各种语义、语法进行分类介绍，以方便读者的使用。

2.2.1 C 语言的数据类型

C 语言有 5 种基本数据类型：字符、整型、单精度实型、双精度实型和空类型。这些数据类型的长度和范围会因处理器的类型和 C 语言编译程序的实现而有所不同，对于 KEIL51 产生的目标文件，表 2-2 给出了几种数据的长度和范围。

表 2-2 在 C51 编译器中几种常用的数据的长度和范围

类型	长度(单位 bit)	范围
char	8	-128~+127
unsigned char	8	0~255
signed char	8	-128~+127
int	16	-32768~+32767
unsigned int	0~65535	16
signed int	-32768~+32767	16
short int	16	-32768~+32767
unsigned short int	16	0~65535
signed short int	16	-32768~+32767
long int	32	-2147483648~+2147483647
unsigned long int	32	0~4294967295
signed long int	32	-2147483648~+2147483647
float	32	-1.175494E-38~+3.402823E+38,

标准 ANSI C 语言还提供了几种聚合类型，包括数组、指针、结构、共用体（联合）、位域和枚举，但是对于 KEIL51 编译器，只提供简单的数据指针类型，对含有函数指针、结构体和共用体等数据结构的程序并不能编译通过。不过，也有许多嵌入式的 C 语言编译器提供这些功能，所以我们也会在以后的章节中讨论这些复杂类型。

为了使用方便，C 编译程序允许使用整型的简写形式：

short int 简写为 short、long int 简写为 long、unsigned short int 简写为 unsigned short、unsigned int 简写为 unsigned、unsigned long int 简写为 unsigned long。

2.2.2 常量与变量

1. 标识符命名

在 C 语言中，标识符是对变量、函数标号和其他各种用户定义对象的命名。标识符的长度可以是一个或多个字符。绝大多数情况下，标识符的第一个字符必须是字母或下划线，随后的字符必须是字母、数字或下划线（某些 C 语言编译器可能不允许下划线作为标识符的起始字符）。表 2-3 是一些正确或错误标识符命名的实例。

表 2-3 正确或错误标识符命名的实例

正确形式	错误形式
count	2 count
test23	hi! there
high _ balance	high .. balance

ANSI 标准规定，标识符可以为任意长度，但外部名必须至少能由前 8 个字符唯一地区分。这里外部名指的是在链接过程中所涉及的标识符，其中包括文件间共享的函数名和全局变量名。

ANSI 标准还规定内部名必须至少能由前 31 个字符唯一地区分。内部名指的是仅出现于定义该标识符的文件中的那些标识符。C 语言中的字母是有大小写区别的，因此 count、

Count、COUNT 是 3 个不同的标识符。标识符不能和 C 语言的关键字相同，也不能和用户已编写的函数或 C 语言库函数同名。但是由于历史的原因，在一些 C51 编译器中，对大小并不区分，所以读者需要注意。

2. 常量

C 语言中的常量是不接受程序修改的固定值，常量可为任意数据类型，数据类型常量举例：

```
char 'a'、'\n'、'9'  
int 21、123、2100、-234  
long int 35000、-34  
short int 10、-12、90  
unsigned int 10000、987、40000  
float 123.23、4.34e-3  
double 123.23、12312333、-0.9876234
```

C 语言还支持另一种预定义数据类型的常量，这就是串。所有串常量括在双撇号之间，例如"This is a test"。切记，不要把字符和串相混淆，单个字符常量是由单撇号括起来的，如'a'。

3. 变量

其值可以改变的量称为变量。一个变量应该有一个名字（标识符），在内存中占据一定的存储单元，在该存储单元中存放变量的值。请注意区分变量名和变量值这两个不同的概念。所有的 C 变量必须在使用之前定义。定义变量的一般形式是：

```
type variable_list;
```

这里的 type 必须是有效的 C 数据类型，variable _ list（变量表）可以由一个或多个由逗号分隔的多个标识符名构成。下面给出一些定义的范例。

```
int i, j, l;  
short int si;  
unsigned int ui;  
double balance, profit, loss;
```

注意 C 语言中变量名与其类型无关。

2.2.3 整型数据

1. 整型常量

整型常量及整常数。它可以是十进制、八进制、十六进制数字表示的整数值。

十进制常数的形式是：

```
digits
```

这里 digits 可以是从 0 到 9 的一个或多个十进制数位。

八进制常数的形式是：

0digits

在此, digits 可以是一个或多个八进制数 (0~7 之间), 起始 0 是必须的引导符。

十六进制常数是下述形式:

0xdigits

0Xhdigits

这里 hdigits 可以是一个或多个十六进制数 (从 0~9 的数字, 并从 “a” ~ “f”的字母)。引导符 0 是必须有的, X 即字母可用大写或小写。

注意, 空白字符不可出现在整数数字之间。表 2-4 列出了整常数的形式。

表 2-4

整常数的形式

十进制	八进制	十六进制
10	012	0xa 或 0XA
132	0204	0X84
32179	076663	0X7db3 或 0X7DB3

整常数在不加特别说明时总是正值。如果需要的是负值, 则负号 “-” 必须放置于常数表达式的前面。每个常数依其值要给出一种类型。当整常数应用于一表达式时, 或出现有负号时, 常数类型自动执行相应的转换, 十进制常数可等价于带符号的整型或长整型, 这取决于所需的常数的大小。

八进制和十六进制常数可对应整型、无符号整型、长整型或无符号长整型, 具体类型也取决于常数的大小。如果常数可用整型表示, 则使用整型。如果常数值大于一个整型所能表示的最大值, 但又小于整型位数所能表示的最大数, 则使用无符号整型。同理, 如果一个常数比无符号整型所表示的值还大, 则它为长整型。如果需要, 当然也可用无符号长整型。

在一个常数后面加一个字母 L 或 l, 则认为是长整型。如 10L、79L、012L、0115L、0XAL、0x4fL 等。

2. 整型变量

前面已提到, C 规定在程序中所有用到的变量都必须在程序中指定其类型, 即“定义”。这是和 BASIC、FORTRAN 不同的, 而与 PASCAL 相似。

[例 2-6] 整型变量的例子

```
#include <REG52.H> //特殊寄存器的头文件
//专供 8051 扩展系列的单片机使用

#include <stdio.h> //I/O 库函数原型声明

#ifndef MONITOR51 //是否需要使用 Monitor-51 调试
char code reserve [3] _at_ 0x23; //如果是, 留下该空间供串口中断使用
#endif //停止程序执行

//-----
//C 程序的主函数, 在堆栈等初始化完成
```

```

//以后，程序从此处开始执行
//-----
void main (void) {

    int a, b, c, d; //指定 a , b , c , d 为整型变量
    unsigned u; //指定 u 为无符号整型变量
//-----
//设定串口的数据传输速率为 1200bit/s，晶振频率为 16MHz
// -----
#ifndef MONITOR51
    SCON = 0x50;           // SCON: 模式 1, 8-bit 异步串口通信
    TMOD |= 0x20;          // TMOD: 定时器 1 为模式 2, 8-bit 自动装载方式
    TH1   = 221;            // TH1:1200bit/s 的装载值@ 16MHz
    TR1   = 1;              // TR1:timer1 运行
    TI    = 1;              // TI:设置为 1, 以发送第一个字节
#endif
//-----
//注意: 由于没有操作系统来接受 main 函
//数的返回值, 所以对一个嵌入式系统来说,
//main 函数永远不会被退出。它必须有一个
//循环来保证程序不会被终止
//-----
a=12; b=-24; u=10;
c=a+u; d=b+u;
printf("a+u=%d, b+u=%d\n", c, d);
while (1) {};
}

```

运行结果为：

```
a+u=22, b+u= - 14
```

可以看到不同类型的整型数据可以进行算术运算。在本例中是 int 型数据与 unsigned int 型数据进行相加减运算。

2.2.4 实型数据

1. 实型常量

实型常量又称浮点常量，是一个十进制表示的符号实数。符号实数的值包括整数部分、尾数部分和指数部分。实型常量的形式如下：

[digits][. digits][E|e[+|-]digits]

在此 digits 是一位或多位十进制数字（从 0~9）。E（也可用 e）是指数符号。小数点

之前是整数部分，小数点之后是尾数部分，它们是可省略的。小数点在没有尾数时可省略。指数部分用 E 或 e 开头，幂指数可以为负，当没有符号时视为正指数的基数为 10，如 1.575E10 表示为： 1.575×10^{10} 。在实型常量中不得出现任何空白符号。在不加说明的情况下，实型常量为正值。如果表示负值，需要在常量前使用负号。

下面是一些实型常量的示例：

```
15.75, 1.575E10, 1575e-2, -0.0025, -2.5e-3, 25E-4
```

所有的实型常量均视为双精度类型。

注意字母 E 或 e 之前必须有数字，且 E 或 e 后面指数必须为整数，如 e3、2.1e3.5、.e3、e 等都是不合法的指数形式。

2. 实型变量

实型变量分为单精度（float 型）和双精度（double 型）。对每一个实型变量都应再使用前加以定义。如：

```
float x, y;           //指定 x, y 为单精度实数
double z;             //指定 z 为双精度实数
```

在一般系统中，一个 float 型数据在内存中占 4 个字节（32 位）一个 double 型数据占 8 个字节（64 位）。单精度实数提供 7 位有效数字，双精度提供 15~16 位有效数字，数值的范围随编译器不同而不同。

值得注意的是，实型常量是 double 型，当把一个实型常量赋给一个 float 型变量时，系统会截取相应的有效位数。例如：

```
float a;
a= 111111.111 ;
```

由于 float 型变量只能接收 7 位有效数字，因此最后两位小数不起作用。如果将 a 改为 double 型，则能全部接收上述 9 位数字并存储在变量 a 中。

2.2.5 字符型数据

1. 字符常量

字符常量是指用一对单引号括起来的一个字符。如 ‘a’，‘9’，‘!’。字符常量中的单引号只起定界作用并不表示字符本身。单引号中的字符不能是单引号（‘）和反斜杠（\），它们特有的表示法在转义字符中介绍。

在 C 语言中，字符是按其所对应的 ASCII 码值来存储的，一个字符占一个字节。如表 2-5 所示。

表 2-5

字符与其所对应的 ASCII 码值

字符	ASCII 码值
!	33
0	48
1	49
9	57
A	65

B	66
a	97
b	98

注意字符'9'和数字9的区别，前者是字符常量，后者是整型常量，它们的含义和在计算机中的存储方式都截然不同。

由于C语言中字符常量是按整数(short型)存储的，所以字符常量可以像整数一样在程序中参与相关的运算。例如：

```
'a' -32;           // 执行结果 97 -32 = 65
'A' +32;           // 执行结果 65+32 = 97
'9' -9;            // 执行结果 57 -9 = 48
```

2. 字符串常量

字符串常量是指用一对双引号括起来的一串字符。双引号只起定界作用，双引号括起的字符串中不能是双引号(“)和反斜杠(\)，它们特有的表示法在转义字符中介绍。例如：

"China", "Cprogram", "YES&NO", "33312-2341", "A"等。

C语言中，字符串常量在内存中存储时，系统自动在字符串的末尾加一个“串结束标志”，即ASCII码值为0的字符NULL，常用\0表示。因此在程序中，长度为n个字符的字符串常量，在内存中占有n+1个字节的存储空间。

例如，字符串China有5个字符，作为字符串常量"China"存储于内存中时，共占6个字节，系统自动在后面加上NULL字符。

要特别注意字符串与字符串常量的区别，除了表示形式不同外，其存储性质也不相同，字符串'A'只占1个字节，而字符串常量"A"占2个字节。

3. 转义字符

转义字符是C语言中表示字符的一种特殊形式。通常使用转义字符表示ASCII码字符集中不可打印的控制字符和特定功能的字符，如用于表示字符常量的单撇号(')，用于表示字符串常量的双撇号(")和反斜杠(\)等。转义字符用反斜杠\后面跟一个字符或一个八进制或十六进制数表示。表2-6给出了C语言中常用的转义字符。

表2-6 C语言中常用的转义字符

转意字符	含义	ASCII码值(十进制)
\a	响铃(BEL)	007
\b	退格(BS)	008
\f	换页(FF)	012
\n	换行(LF)	010
\r	回车(CR)	013
\t	水平制表(HT)	
\v	垂直制表(VT)	011
\\\	反斜杠	092
\?	问号字符	063
\'	单引号字符	039
\"	双引号字符	034
\0	空字符(NULL)	

\ddd	任意字符三位八进制
\xhh	任意字符两位十六进制

字符常量中使用单引号和反斜杠以及字符常量中使用双引号和反斜杠时，都必须使用转义字符表示，即在这些字符前加上反斜杠。

在C程序中使用转义字符\ddd或者\xhh可以方便灵活地表示任意字符。\\ddd为斜杠后面跟三位八进制数，该三位八进制数的值即为对应的八进制ASCII码值。\\x后面跟两位十六进制数，该两位十六进制数为对应字符的十六进制ASCII码值。

使用转义字符时需要注意以下问题：

- (1) 转义字符中只能使用小写字母，每个转义字符只能看作一个字符。
- (2) \\v垂直制表和\\f换页符对屏幕没有任何影响，但会影响打印机执行响应操作。
- (3) 在C程序中，使用不可打印字符时，通常用转义字符表示。

4. 符号常量

C语言允许将程序中的常量定义为一个标识符，称为符号常量。符号常量一般使用大写英文字母表示，以区别于一般用小写字母表示的变量。符号常量在使用前必须先定义，定义的形式是：

```
#define <符号常量名> <常量>
```

例如：

```
#define PI 3.1415926
#define TRUE 1
#define FALSE 0
#define STAR '*'
```

这里定义PI、TRUE、FALSE、STAR为符号常量，其值分别为3.1415926，1，0，'*'。

#define是C语言的预处理命令，它表示经定义的符号常量在程序运行前将由其对应的常量替换。定义符号常量的目的是为了提高程序的可读性，便于程序的调试和修改。因此在定义符号常量名时，应使其尽可能地表达它所代表的常量的含义，例如前面所定义的符号常量名PI(p)，表示圆周率3.1415926。此外，若要对一个程序中多次使用的符号常量的值进行修改，只须对预处理命令中定义的常量值进行修改即可。

5. 字符变量

字符变量用来存放字符常量，注意只能存放一个字符。

字符变量的定义形式如下：

```
char c1, c2;
```

它表示c1和c2为字符变量，各放一个字符。因此可以用下面语句对c1、c2赋值：

```
c1='a'; c2='b';
```

[例2-7] 字符变量的例子。

```
#include <REG52.H> //特殊寄存器的头文件
//专供8051扩展系列的单片机使用
```

```

#include <stdio.h>                                //I/O 库函数原型声明

#ifndef MONITOR51                                //是否需要使用 Monitor-51 调试
char code reserve [3] _at_ 0x23;                //如果是, 留下该空间供串口中断使用
#endif                                         //停止程序执行

//-----
//C 程序的主函数, 在堆栈等初始化完成
//以后, 程序从此处开始执行
//-----
void main (void) {

    char c1, c2;
    //-----

    //设定串口的数据传输速率为 1200bit/s, 晶振频率为 16MHz
    //

#ifndef MONITOR51
    SCON = 0x50;                                // SCON: 模式 1, 8-bit 异步串口通信
    TMOD |= 0x20;                               // TMOD: 定时器 1 为模式 2, 8-bit 自动装载方式
    TH1 = 221;                                  // TH1:1200bit/s 的装载值@ 16MHz
    TR1 = 1;                                    // TR1:timer1 运行
    TI = 1;                                     // TI:设置为 1, 以发送第一个字节
#endif

    //-----
    //注意: 由于没有操作系统来接受 main 函数的返回值, 所以对一个嵌入式系统来说,
    //main 函数永远不会被退出。它必须有一个
    //循环来保证程序不会被终止
    //-----

    c1 = 97 ; c2 = 98 ;
    printf("%c %c", c1, c2);
    while (1) {};
}

```

c1、c2 被指定为字符变量。但在第 3 行中，将整数 97 和 98 分别赋给 c1 和 c2，它的作用相当于以下两个赋值语句：

```
c1 = 'a' ; c2 = 'b' ;
```

因为'a' 和'b' 的 ASCII 码为 97 和 98。第 4 行将输出两个字符。”%c”是输出字符的格式。
程序输出：

```
a b
```

[例 2-8] 利用字符变量进行大小写转换

```

#include <REG52.H> //特殊寄存器的头文件
//专供 8051 扩展系列的单片机使用

#include <stdio.h> //I/O 库函数原型声明

#ifndef MONITOR51 //是否需要使用 Monitor-51 调试
char code reserve [3] _at_ 0x23; //如果是，留下该空间供串口中断使用
#endif //停止程序执行

//-----
//C 程序的主函数，在堆栈等初始化完成
//以后，程序从此处开始执行
//-----
void main (void) {

    char c1, c2;
//-----
//设定串口的数据传输速率为 1200bit/s，晶振频率为 16MHz
// -----
#ifndef MONITOR51
    SCON = 0x50; // SCON: 模式 1, 8-bit 异步串口通信
    TMOD |= 0x20; // TMOD: 定时器 1 为模式 2, 8-bit 自动装载方式
    TH1 = 221; // TH1:1200bit/s 的装载值@ 16MHz
    TR1 = 1; // TR1:timer1 运行
    TI = 1; // TI:设置为 1, 以发送第一个字节
#endif

//-----
//注意：由于没有操作系统来接受 main 函数的返回值，所以对一个嵌入式系统来说，main 函数永远不会被退出。它必须有一个循环来保证程序不会被终止
//-----
    c1='a';c2='b';
    c1=c1-32;c2=c2-32;
    printf("%c %c", c1, c2);
}

```

```
while (1) {};
```

运行结果为：

A B

它的作用是将两个小写字母转换为大写字母。因为' a' 的 ASCII 码为 97，而' A' 为 65，' b' 为 98，' B' 为 66。从 ASCII 代码表中可以看到每一个小写字母比大写字母的 ASCII 码大 32。即' a' = ' A' + 32。

2.2.6 运算符

C 语言的内部运算符很丰富，运算符是告诉编译程序执行特定算术或逻辑操作的符号。C 语言有 3 大运算符：算术、关系与逻辑、位操作。另外，C 语言还有一些特殊的运算符，用于完成一些特殊的任务。

1. 算术运算符

表 2-7 列出了 C 语言中允许的算术运算符。在 C 语言中，运算符“+”、“-”、“*”和“/”的用法与大多数计算机语言的相同，几乎可用于所有 C 语言内定义的数据类型。当“/”被用于整数或字符时，结果取整。例如，在整数除法中， $10/3=3$ 。

一元减法的实际效果等于用 -1 乘单个操作数，即任何数值前放置减号将改变其符号。模运算符“%”在 C 语言中也同它在其他语言中的用法相同。切记，模运算取整数除法的余数，所以“%”不能用于 float 和 double 类型。

表 2-7 算术运算符

运算符	作用
+	加法
-	减法
*	乘法
/	除法
++	自加
--	自减
%	模运算

下面是说明%用法的程序段。

```
int x, y;
x = 10 ;
y = 3;
printf("%d", x/y);           //显示 3
printf("%d", x%y);           //显示 1, 整数除法的余数
x = 1 ;
y = 2 ;
printf("%d, %d", x/y, x%y);  //显示 0, 1
```

最后一行打印一个 0 和一个 1，因为 $1/2$ 整除时为 0，余数为 1，故 $1\%2$ 取余数 1。

2. 自增和自减

C语言中有两个很有用的运算符，通常在其他计算机语言中是找不到它们的——自增和自减运算符，“`++`”和“`--`”。运算符“`++`”是操作数加1，而“`--`”是操作数减1，换句话说：“`x = x + 1`”；同“`++x`”；“`x = x - 1`”；同“`-x`”。

自增和自减运算符可用在操作数之前，也可放在其后，例如：“`x = x + 1`”；可写成“`++x`”；或“`x ++`”；但在表达式中这两种用法是有区别的。自增或自减运算符在操作数之前，C语言在引用操作数之前就先执行加1或减1操作；运算符在操作数之后，C语言就先引用操作数的值，而后再进行加1或减1操作。请看下例：

```
x=10;
y=++x;
```

此时，`y=11`，如果程序改为：

```
x=10 ;
y=x++;
```

则`y=10`。在这两种情况下，`x`都被置为11，但区别在于设置的时刻，这种对自增和自减发生时刻的控制是非常有用的。

在大多数C编译程序中，为自增和自减操作生成的程序代码比等价的赋值语句生成的代码要快得多，所以尽可能采用加1或减1运算符是一种好的选择。

下面是算术运算符的优先级：

最高	<code>++</code> 、 <code>--</code>
	<code>-</code> （一元减）
	<code>*</code> 、 <code>/</code> 、 <code>%</code>
最低	<code>+</code> 、 <code>-</code>

编译程序对同级运算符按从左到右的顺序进行计算。当然，括号可改变计算顺序。C语言处理括号的方法与几乎所有的计算机语言相同：强迫某个运算或某组运算的优先级升高。

3. 关系和逻辑运算符

关系运算符中的“关系”二字指的是一个值与另一个值之间的关系，逻辑运算符中的“逻辑”二字指的是连接关系的方式。因为关系和逻辑运算符常在一起使用，所以将它们放在一起讨论。关系和逻辑运算符概念中的关键是True（真）和False（假）。C语言中，非0为True，0为False。使用关系或逻辑运算符的表达式对False和True分别返回值0或1，如表2-8所示。

表2-8 关系和逻辑运算符

关系和逻辑运算符	含义
<code>></code>	大于
<code>>=</code>	大于等于
<code><</code>	小于
<code><=</code>	小于等于
<code>==</code>	等于
<code>!=</code>	不等于
<code>&&</code>	与
<code> </code>	或

!	非
---	---

表 2-8 给出于关系和逻辑运算符，下面用 1 和 0 给出逻辑真值表。

关系和逻辑运算符的优先级比算术运算符低，即像表达式“ $10 > 1 + 12$ ”的计算可以假定是对表达式“ $10 > (1 + 12)$ ”的计算，当然，该表达式的结果为 False。

在一个表达式中允许运算的组合。例如：

```
10>5&&! (10<9) || 3<=4
```

这一表达式的结果为 True。

下面给出了关系和逻辑运算符的相对优先级：

最高	!
	$>= <=$
	$=!=$
	$\&\&$
最低	$ $

同算术表达式一样，在关系或逻辑表达式中也使用括号来修改原计算顺序。切记，所有关系和逻辑表达式产生的结果不是 0 就是 1，所以 下面的程序段不仅正确而且将会输出数值 1。

```
int x;
x = 100 ;
printf ( " %d ", x > 10 ) ;
```

4. 位操作符

与其他语言不同，C 语言支持全部的位操作符 (Bitwise Operators)。位操作是对字节或字中的位 (bit) 进行测试、置位或移位处理，这里字节或字是针对 C 标准中的 char 和 int 数据类型而言的。位操作不能用于 float、double、long double、void 或其他复杂类型。表 2-9 给出了位操作的操作符。位操作中的 AND、OR 和 NOT (1 的补码) 的真值表与逻辑运算等价，唯一不同的是位操作是逐位进行运算的。

表 2-9 位操作的操作符

关系和逻辑运算符	含义
$\&$	与
$ $	或
$^$	异或
\sim	补
$>>$	右移
$<<$	左移

表 2-10 是异或的真值表。

表 2-10 异或的真值表

P	q	$p \hat{^} q$
0	0	0
1	0	1
1	1	0
0	1	1

位操作通常用于设备驱动程序，例如调制解调器程序、磁盘文件管理程序和打印机驱动

程序。这是因为位操作可屏蔽掉某些位，如奇偶校验位（奇偶校验位用于确保字节中的其他位不会发生错误通常奇偶校验位是字节的最高位）。通常我们可把位操作 AND 作为关闭位的手段，这就是说两个操作数中任一为 0 的位，其结果中对应位置为 0。例如，下面的函数通过调用函数 `read_modem()`，从调制解调器端口读入一个字符，并将奇偶校验位置成 0。

```
char get_char_from_modem()
{
    char ch;
    ch=read_modem();           //从调制解调器端口中得到一个字符
    return ( ch & 127 ) ;
}
```

字节的位 8 是奇偶位，将该字节与一个位 1~7 为 1、位 8 为 0 的字节进行与操作，可将该字节的奇偶校验位置成 0。表达式 `ch&127` 正是将 `ch` 中每一位同 127 数字的对应位进行与操作，结果 `ch` 的位 8 被置成 0。在下面的例子中，假定 `ch` 接收到字符“`A`”并且奇偶位已经被置位。

奇偶位

110000001 内容为 ‘A’ 的 `ch`，其中奇偶校验位为 1

011111111 二进制的 127 执行与操作

& 与操作

010000001 去掉奇偶校验的 ‘A’

位操作 OR 与 AND 操作相反，可用来置位。任一操作数中为 1 的位将结果的对应位置 1。如下所示， $128|3$ 的情况是：

1000000	128 的二进制
0000011	3 的二进制
	或操作
1000011	结果

异或操作通常缩写为 XOR，当且仅当做比较的两位不同时，才将结果的对应位置位。如下所示，异或操作 127^120 的情况是：

0111111	127 的二进制
0111100	120 的二进制
^	异或操作
0000011	结果

一般来说，位的 AND、OR 和 XOR 操作通过对操作数运算，直接对结果变量的每一位分别处理。正是因为这一原因，位操作通常不像关系和逻辑运算符那样用在条件语句中，我们可以用例子说明这一点，假定 `X=7`，那么 `x&&8` 为 True(1)，而 `x&8` 却为 False(0)。

记住，关系和逻辑操作符结果不是 0 就是 1。而相似的位操作通过相应处理，结果可为任意值。换言之，位操作可以有 0 或 1 以外的其他值，而逻辑运算符的计算结果总是 0 或 1。

移位操作符“`>>`”和“`<<`”将变量的各位按要求向右或向左移动。右移语句通常形式是：

变量`>>`右移位数

左移语句的通常形式是：

变量`<<`左移位数

当某位从一端移出时，另一端移入 0（某些计算机是送 1，详细内容请查阅相应 C 编译程序用户手册）。切记：移位不同于循环，从一端移出的位并不送回到另一端去，移去的位永远丢失了，同时在另一端补 0。

移位操作可对外部设备（如 D/A 转换器）的输入和状态信息进行译码，移位操作还可用于整数的快速乘除运算。例如表 2-11 所示的情况。

表 2-11 移位操作进行乘除的例子

字符 x	每个语句执行以后的 xx 的值	化为十进制
x=7	00000111	7
x<<1	00001110	14
x<<3	01110000	112
x<<2	11000000	192
x>>1	01100000	96
x>>2	00011000	24

每左移一位乘 2，注意 $x \ll 2$ 后，原 x 的信息已经丢失了，因为一位已经从一端出，每右移一位相当于被 2 除，注意，乘后再除时，除操作并不带回乘法时已经丢掉的高位。

反码操作符为 \sim 。 \sim 的作用是将特定变量的各位状态取反，即将所有的 1 位置成 0，所有的 0 位置成 1。

位操作符经常用在加密程序中，例如，若想生成一个不可读磁盘文件时，可以在文件上做一些位操作。最简单的方法是用下述方法，通过 1 的反码运算，将每个字节的每一位取反。

原字节	00101100
第一次取反码	11010011
第二次取反码	00101100

注意，对同一行进行连续的两次求反，总是得到原来的数字，所以第一次求反表示了字节的编码，第二次求反进行译码又得到了原来的值。

可以用下面的函数 encode() 对字符进行编码。

```
char encode(char ch)
{
    return (~ch);
}
```

5. ?操作符

C 语言提供了一个可以代替某些“if…else”语句的简便易用的操作符“?”。该操作符是三元的，其一般形式为：

EXP1?EXP2:EXP3

其中，EXP1，EXP2 和 EXP3 是表达式，注意冒号的用法和位置。

操作符“?”的作用是，在计算 EXP1 之后，如果数值为 True，则计算 EXP2，并将结果作为整个表达式的数值；如果 EXP1 的值为 False，则计算 EXP3，并以它的结果作为整个表达式的值，请看下例：

```
x = 10 ;
y = x > 9 ? 100 : 200 ;
```

例中，赋给 y 的数值是 100，如果 x 被赋给比 9 小的值，y 的值将为 200，若用 if…else

语句改写，有下面的等价程序：

```
x = 10 ;
if(x>9) y=100;
else y=200;
```

有关C语言中的其他条件语句将在后面的章节进行讨论。

6. 逗号操作符

作为一个操作符，逗号把几个表达式串在一起。逗号操作符的左侧总是作为void(无值)，这意味着其右边表达式的值变为以逗号分开的整个表达式的值。例如：

```
x = ( y = 3 , y + 1 );
```

上式将3赋给y，然后将4赋给x，因为逗号操作符的优先级比赋值操作符优先级低，所以必须使用括号。

实际上，逗号表示操作顺序。当它在赋值语句右边使用时，所赋的值是逗号分隔开的表中最后那个表达式的值。例如，

```
y = 10;
x = ( y = y - 5 , 25/y );
```

执行后，x的值是5，因为y的起始值是10，减去5之后结果再除以25，得到最终结果。在某种意义上可以认为，逗号操作符和标准英语的and是同义词。

7. 关于优先级的小结

本节总结出了C语言所有操作符的优先级，其中包括将在本书后面讨论的某些操作符。注意，所有操作符（除一元操作符和?之外）都是左结合的。一元操作符(*, &和-)及操作符“?”则为右结合。

最高级	(), [] →
	! , ~, ++, --, -(type), *, &, sizeof
	*, /, %
	+ , -
	<<, >>
	<= , >=
	== , !=
	&, ^,
	&&,
	,
	?,
	=, +=, -=, *=, /=
最低级	,

2.2.7 表达式

表达式由运算符、常量及变量构成。C语言的表达式基本遵循一般代数规则，有几点却

是与 C 语言紧密相关的，以下将分别加以讨论。

1. 表达式中的类型转换

混合于同一表达式中的不同类型常量及变量，应均变换为同一类型的量。C 语言的编译程序将所有操作数变换为与最大类型操作数同类型。变换以一次一操作的方式进行。具体规则如下：

- ① 所有 char 及 short int 型量转为 int 型，所有 float 转换为 double。
- ② 如操作数对中一个为 long double，另一个转换为 long double。
- ③ 如果没有操作数为 long double，但是有一个为 double，另一个转为 double。
- ④ 如果没有操作数为 double，但是有一个为 long，另一个转为 long。
- ⑤ 如果没有操作数为 long，但是有一个为 unsigned，另一个转为 unsigned。

一旦运用以上规则。每一对操作数均变为同类型。

2. 强制转换

可以通过强制类型转换的方式来将一个表达式变为特定类型。其一般形式为：

(type) expression

(type) 是标准 C 语言中的一个数据类型。例如，为确保表达式 $x/2$ 的结果具有类型 float，可写为：

(float) $x/2$

虽然强制类型转换在程序中用得不多，但有时它的使用的确很有价值。例如，假设希望用一整数控制循环，但在执行计算时又要有小数部分。

[例 2-9] 使用构成符的例子

```
#include <REG52.H> //特殊寄存器的头文件
//专供 8051 扩展系列的单片机使用

#include <stdio.h> //I/O 库函数原型声明

#ifndef MONITOR51 //是否需要使用 Monitor-51 调试
char code reserve [3] _at_ 0x23; //如果是，留下该空间供串口中断使用
#endif //停止程序执行

//-----
//C 程序的主函数，在堆栈等初始化完成
//以后，程序从此处开始执行
//-----
void main (void) {
```

```

int i ;
//-----
//设定串口的数据传输速率为 1200bit/s，晶振频率为 16MHz
// -----
#ifndef MONITOR51
    SCON = 0x50;           // SCON: 模式 1, 8-bit 异步串口通信
    TMOD |= 0x20;          // TMOD: 定时器 1 为模式 2, 8-bit 自动装载方式
    TH1   = 221;           // TH1:1200bit/s 的装载值@ 16MHz
    TR1   = 1;              // TR1:timer1 运行
    TI    = 1;              // TI:设置为 1, 以发送第一个字节
#endif

//-----
//注意：由于没有操作系统来接受 main 函
//数的返回值，所以对一个嵌入式系统来说，
//main 函数永远不会被退出。它必须有一个
//循环来保证程序不会被终止
//-----
for (i+1;i<=100;++i)
printf("%d/2 is :%f", i, (float)i/2);
while (1) {};
}

```

若没有强制类型转换(float)，就仅执行一次整数除；有了它就可保证输出显示答案的小数部分。

3. 空格与括号

为了增加可读性，可以随意在表达式中插入 tab 和空格符。例如，下面两个表达式是相同的。

```

x = 10/y * ( 127/x ) ;
x = 10/y * ( 127/x ) ;

```

冗余的括号并不导致错误或减慢表达式的执行速度。我们鼓励使用括号，它可使执行顺序更清楚一些。例如，下面两个表达式中哪个更易读一些呢？

```

x = y/2 - 34 * temp & 127 ;
x = ( y/2 ) - ( ( 34 * temp ) & 127 ) ;

```

4. C语言中的简写形式

C语言提供了某些赋值语句的简写形式。例如语句：

```
x=x+10;
```

在C语言中简写形式是：

```
x+=10;
```

这组操作符对+=通知编译程序将 X+10 的值赋予 X。这一简写形式适于 C 语言的所有二元操作符(需两个操作数的操作符)。在 C 语言中, variable1 operator expression 与 variable1 operator=expression 相同。

请看另一个例子:

```
x= x-100;
```

其等价语句是

```
x=100;
```

简写形式广泛应用于专业 C 语言程序中, 希望读者能熟悉它。

2.3 程序控制语句

2.3.1 程序的 3 种基本结构

通常的计算机程序总是由若干条语句组成, 从执行方式上看, 从第一条语句到最后一条语句完全按顺序执行, 是简单的顺序结构; 若在程序执行过程当中, 根据用户的输入或中间结果去执行若干不同的任务则为选择结构; 如果在程序的某处, 需要根据某项条件重复地执行某项任务若干次或直到满足或不满足某条件为止, 这就构成循环结构。大多数情况下, 程序都不会是简单的顺序结构, 而是顺序、选择、循环 3 种结构的复杂组合。

C 语言中, 有一组相关的控制语句, 用以实现选择结构与循环结构:

选择控制语句: if、switch、case

循环控制语句: for、while、do...while

转移控制语句: break、continue、goto

我们将在本节中详细介绍。

2.3.2 条件控制语句

在程序的 3 种基本结构中, 第 2 种即为选择结构, 其基本特点是: 程序的流程由多路分支组成, 在程序的一次执行过程中, 根据不同的情况, 只有一条支路被选中执行, 而其他分支上的语句被直接跳过。

C 语言中, 提供 if 语句和 switch 语句选择结构, if 语句用于两者选一的情况, 而 switch 用于多分支选一的情形。

1. if 语句

(1) if 语句的两种基本形式

首先, 我们看一个例子, 由此了解选择结构的意义及设计方法。

[例 2-10] 输入 3 个数, 找出并打印其最小数

分析: 设 3 个数为 A、B、C, 由键盘读入, 我们用一个变量 MIN 来标识最小数, A、B、

C与MIN皆定义为int型变量。每次比较两个数，首先比较A和B，将小的一个赋给MIN，再把第三个数C与MIN比较，再将小的一个赋给MIN，则最后MIN即为A、B、C中最小数。

算法如下：

- (1) 输入A、B、C。
- (2) 将A与B中小的一个赋给MIN。
- (3) 将MIN与C中小的一个赋给MIN。
- (4) 输出MIN。

将第(2)步细化为：若A < B，则MIN <= A，否则：MIN <= B。

第(3)步细化为：若C < MIN，则MIN <= C。

if语句的格式为：

if<表达式>语句

当表达式为真时，执行语句，表达式为假时跳过语句。

要注意的是：if或if...else，包括后面要讲到的嵌套if，即if...else if...被看成是一条语句，即使其中的语句是包含多条语句的复合语句，仍然如此。

```
#include <REG52.H> //特殊寄存器的头文件
//专供8051扩展系列的单片机使用

#include <stdio.h> //I/O库函数原型声明
#ifndef MONITOR51 //是否需要使用Monitor-51调试
char code reserve [3] _at_ 0x23; //如果是，留下该空间供串口中断使用
#endif //停止程序执行

//-----
//C程序的主函数，在堆栈等初始化完成
//以后，程序从此处开始执行
//-----
void main (void) {

    int a, b, c, min;
//-----
//设定串口的数据传输速率为1200bit/s，晶振频率为16MHz
// -----
#ifndef MONITOR51
    SCON = 0x50; // SCON: 模式1, 8-bit异步串口通信
    TMOD |= 0x20; // TMOD: 定时器1为模式2, 8-bit自动装载方式
    TH1 = 221; // TH1: 1200bit/s的装载值@ 16MHz
    TR1 = 1; // TR1: timer1运行
    TI = 1; // TI: 设置为1, 以发送第一个字节
#endif
}
```

```

//-----
//注意：由于没有操作系统来接受 main 函
//数的返回值，所以对一个嵌入式系统来说，
//main 函数永远不会被退出。它必须有一个
//循环来保证程序不会被终止
//-----

printf(" input a, b, c :\n");
scanf( " %d %d %d", &a, &b, &c ) ;
if(a<b)
    min = a;
else
    min = b;
if(c<min)
    min = c;
printf("The result is %d\n", min);

while (1) {};
}

```

事实上，除去 C51 中初始化的代码段以外，上面的程序中的主程序的内容已经全部包含在下面的程序段中。

```

main()
{
    int a, b, c, min;
    printf(" input a, b, c:");
    scanf("%d %d %d", &a, &b, &c);
    if (a<b)
        min = a;
    else
        min = b;
    if (c<min)
        min = c;
    printf("The result is %d\n", min);
}

```

为了减少篇幅，也让读者在阅读代码的时候能够更多的把精力集中在我们需要学习的代码部分上，以后的程序中将都只包含有简化以后的代码。如果读者需要看完整的源程序，请参考 www.cs-book.com 页面上相应目录上的代码。

程序的执行情况如下：

```

input a, b, c:3 5 2
The result is :2

```

这里顺便提一下程序书写的缩排问题，所谓缩排，就是下一行与上一行相比，行首向右缩进若干字符，如上例的 `min=a`、`min=b` 等。适当的缩排能使程序的结构、层次清晰、一目了然，增加程序的易读性。应该从一开始就养成一个比较好的书写习惯，包括必要的注释、适当的空行以及缩排。

(2) 复合语句

`if` 语句中，有时需要执行的语句不止一条，这就要用到复合语句。

复合语句，就是用一对花括号括起来的一条或多条语句，形式如下：

```
{
语句 1;
语句 2;
.....
语句 n;
}
```

无论包括多少条语句，复合语句从逻辑上讲，被看成是一条语句。复合语句在分支结构、循环结构中，使用十分广泛。

[例 2-11] 读入两个数 `x`、`y`，将大数存入 `x`，小数存入 `y`。

分析：`x`、`y` 从键盘读入，若 $x >= y$ ，只需顺序打出，否则，应将 `x`、`y` 中的数进行交换，然后输出。两数交换必须使用一个中间变量 `t`，定义三个浮点数 `x`、`y`、`t`。

算法：

- ①读入 `x`、`y`；
- ②大数存入 `x`，小数存入 `y`；
- ③输出 `x`、`y`。

第②步更细化的逻辑如下：

若 $x < y$ ，则交换 `x` 与 `y`；

再求精，`x` 与 `y` 交换；

- a. $t \leq x$
- b. $x \leq y$
- c. $y \leq t$

程序如下：

```
# include <stdio.h>
main ( )
{
    float x, y, t;
    printf("input x, y:");
    scanf ( " %f %f " , &x , &y ) ;           //输入 x、y 的值
    if (x<y)
    {
        //如果 x<y, x 和 y 互换
        t=x;
    }
}
```

```

    x=y;
    y=t;
}
printf ("result:%7.3f\t%7.3f\n", x, y);
}

```

执行结果：

```

input x,y :43.2 56.7
result : 56.700 43.200

```

(3) if...else if 语句

实际应用中常常面对更多的选择，这时，将 if...else 扩展一下，就得到 if...else if 结构，其一般形式为：

```

if <表达式 1>
语句 1
else if<表达式 2>
语句 2
else if <表达式 3>
语句 3
else 语句 4

```

下面这个问题可以帮我们说明 if...else...语句的用法：

[例 2-12] 货物征税问题

价格在 1 万元以上税率为 5%，5000 元以上 1 万元以下的税率为 3%，1000 元以上 5000 以下的税率为 2%，1000 元以下的免税，读入货物价格，计算并输出应缴纳的税额。

分析：设 price 为价格，tax 为税费，读入 price，再计算 tax，假定货物的价格在 1 万元以上，征税应分段累计，即超出 1 万的部分按 5% 征，5000~10000 的部分按照 3% 征，1000~5000 之间的部分按照 2% 征，1000 以下的不征。即各段采用不同税率进行征收。

总的算法如下：

```

若 price>=10000, 则 tax=0.05*(price-10000)+0.03*(10000-5000)+0.02*(5000-1000) ;
否则, 若 price>=5000, 则 tax=0.03*(price-5000)+0.02*(price-1000) ;
否则, 若 price>=1000, 则 tax=0.02*(price-1000) ;

```

程序如下：

```

# include <stdio.h>
main ( )
{
    float price, tax=0;
    printf("input price:");
    scanf ( "%f" , &price ) ; //输入价格
    if ( price >= 10000.0 ) //price 在 10000 以上的
    {

```

```

        tax=0.05*(price-10000)+ 0.03*(10000-5000)+0.02*(5000-1000);
    }
    else if (price>=5000.0) //price 在 5000 以上的
    {
        tax = 0.03 * ( price - 5000 )+0.02*(price-1000);
    }
    else if( price >= 1000.00) //price 在 1000 以上的
    {
        tax = 0.02*(price-1000);
    }
    printf("the tax=%10.3f", tax);
}

```

运行程序结果如下：

```

input price:15000
the tax=480.000

```

(4) if语句嵌套

在一个 if 语句中可以又出现另一个 if 语句，这称为 if 语句的嵌套或多重 if 语句：

```

if <表达式 1 >
if <表达式 1 1 >
.....
else
语句 2 ;

```

[例 2-13] 计算函数值

$$y = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

本例是计算一个简单的数学函数，该函数中两个变量的关系并不是公式的计算，而是通过简单的逻辑判断就可以确定的，所以在这样的情况下我们也应该使用 if…else…语句。

这个程序有多种写法，不过我们为了说明嵌套的 if 语句，所以将该例的源程序写为如下：

```

main ( )
{
    float x, y;
    printf("input x, y:");
    scanf("%f", &x) ;
    if (x>=0) //如果 x>=0
        if (x>0) //如果 x>0

```

```

        y=1;
    else //否则 x=0
        y=0;
    else
        y=-1;
    printf("y=%4.0f\n", y);
}

```

对多重 if, 最容易犯的错误是 if 与 else 配对错误,
例如, 写成如下形式:

```

y = 0 ;
if (x>=0)
if (x>0)
y =1;
else
y =-1 ;

```

从缩排上可以看出, 作者希望 else 是与 if $x \geq 0$ 配对, 但是 C 语言规定 else 总是与离它最近的上一个 if 配对, 结果, 上述算法完全违背了设计者的初衷。

改进的办法是使用复合语句, 将上述程序段改写如下:

```

y = 0 ;
if(x >= 0)
{
    if(x > 0)
        y = 1 ;
}
else
y =-1;

```

2. switch 语句

if 语句只能处理从两者间选择之一, 当要实现几种可能之一时, 就要用 if...else if 甚至多重的嵌套 if 来实现, 当分支较多时, 程序变得复杂冗长, 可读性降低。C 语言提供了 switch 开关语句专门处理多路分支的情形, 使程序变得简洁。

switch 语句的一般格式为:

```

switch <表达式>
    case 常量表达式 1 : 语句序列 1 ;
        break ;
    case 常量表达式 2 : 语句序列 2 ;
        break ;
    .....
    case 常量表达式 n :语句 n ;
        break ;

```

```
default: 语句 n+1 ;
```

其中常量表达式的值必须是整型，字符型或者枚举类型，各语句序列允许有多条语句，不需要按复合语句处理，若语句序列 i 为空，则对应的 break 语句可去掉。

特殊情况下，如果 switch 表达式的多个值都需要执行相同的语句，可以采用下面的格式：

```
switch (i)
{
    case 1:
    case 2:
    case 3: 语句 1 ;
              break ;
    case 4:
    case 5: 语句 2 ;
              break ;
    default: 语句 3 ;
}
```

当整型变量 i 的值为 1、2 或 3 时，执行语句 1，当 i 的值为 4 或 5 时，执行语句 2，否则，执行语句 3。

[例 2-14] 输入月份，打印 1999 年该月有几天

分析：1999 年为非闰年，所以该年的 1, 3, 5, 7, 8, 10, 12 月有 31 天，4, 6, 9, 11 月有 30 天，2 月有 28 天。

不过请读者注意，作为程序的编制者来说，必须要考虑到用户不小心误操作的情况，也就是说，如果用户输入的数值不在 1~12 之间，程序也应该能有正常的运转，这也是 default 命令的作用。在我们的程序中，如果用户出现错误操作的情况，我们将给出天数的值为 -1，并输出：“Invalid month input！”（无效月份）。

程序如下：

```
#include <stdio.h>
main ( )
{
    int month;
    int day;
    printf("please input the month number :");
    scanf("%d", &month) ; // 用户输入月份
    switch (month) // 多路分支语句
    {
        case 1:
        case 3:
        case 5:
        case 7:
```

```

        case 8:
        case 10:
        case 12: day=31;                                //1, 3, 5, 7, 8, 10, 12 月有 31 天
                    break;
        case 4:
        case 6:
        case 9:
        case 11:day=30;                                //4, 6, 9, 11 月有 30 天
                    break;
        case 2: day=28;                                //2 月有 28 天
                    break;
        default : day=-1;
    }
    if(day== -1)                                     //输出
    printf("Invalid month input !\n");
    else
    printf("1999.%d has %d days \n",month,day);
}

```

2.3.3 程序应用举例

[例 2-15] 解一元二次方程

分析：一元二次方程 $ax^2+bx+c=0$, a、b、c 由键盘输入。这是一个简单方程求解的问题，具体算法如下：

对系数 a、b、c 考虑以下情形

(1) 若 $a = 0$:

① $b \neq 0$, 则 $x = -c/b$;

② $b=0$, 则: 若 $c = 0$, 则 x 无定根; $c \neq 0$, 则 x 无解。

(2) 若 $a \neq 0$:

① $b^2-4ac > 0$, 有两个不等的实根;

② $b^2-4ac=0$, 有两个相等的实根;

③ $b^2-4ac < 0$, 有两个共轭复根。

用嵌套的 if 语句完成。程序如下:

```

#include <math.h>
# include <stdio.h>
main()
{
    float a, b, c, s, x1, x2;
    double t;

```

```

printf(" please input a, b, c:");
scanf("%f %f %f ", &a, &b, &c) ; //输入方程系数 a、b、c
if(a==0.0) //a 为 0 的情况
{
    if(b!=0.0) //b 是否为 0
        printf("the root is :%f\n", -c/b);
    else if(c==0.0) //如果 a、b 均为 0, c 是否为 0
        printf("x is inexactive\n");
    else
        printf("no root!\n");
}
else //a 不为 0 的情况
{
    s=b*b-4*a*c; //求特征多项式
    if(s>=0.0)
    {
        if(s>0.0)
        {
            t=sqrt(s);
            x1=-0.5*(b+t)/a;
            x2=-0.5*(b-t)/a;
            printf("There are two different roots:%f and %f\n", x1, x2);
        }
        else
            printf("There are two equal roots:%f\n", -0.5*b/a);
    }
    else
    {
        t=sqrt(-s) ;
        x1=-0.5*b/a; //实部
        x2=abs(0.5*t/a); //虚部的绝对值
        printf("There are two virtual roots:");
        printf("%f+i%f\t\t%f-i%f\n", x1, x2, x1, x2 );
    }
}
}
}

```

运行结果如下：

```

please input a,b,c: 1 2 3
There are two virtual roots:
-1.000000 + i 1.000000 -1.000000 - i 1.000000

please input a,b,c: 2 5 3
There are two different roots : -1.500000 and -1.000000

```

```
please input a, b, c: 0 0 3
No root!
```

2.4 循环控制语句

循环控制结构（又称重复结构）是程序中的另一个基本结构。在实际问题中，常常需要进行大量的重复处理，循环结构可以使我们只写很少的语句，而让计算机反复执行，从而完成大量类同的计算。

C 语言提供了 while 语句、do while 语句和 for 语句实现循环结构。

2.4.1 while 语句

while 语句是当型循环控制语句，一般形式为：

```
while<表达式>语句;
```

语句部分称为循环体，当需要执行多条语句时，应使用复合语句。

while 语句的特点是先判断，后执行，若条件不成立，有可能一次也不执行。

[例 2-16] 求 n!

分析： $n!=n \times (n-1) \times (n-2) \cdots 2 \times 1$, $0!=1$ 。即 $S_0=1$, $S_n=S_{n-1} \times n$ 。可以从 S_0 开始，依次求出 S_1 、 S_2 、… S_n 。

统一令 S 等于阶乘值，S 的初值为 $0!=1$ ；变量 i 为计数器，i 从 1 变到 n，每一步令 $S=S \times i$ ，则最终 S 中的值就是 $n!$ 。

不过，阶乘的 n 必须要大于 0，否则输出应显示为非法操作。

```
main()
{
    int n, i;
    long int s;
    printf(" please input n (n>=0) :");
    scanf("%d", &n); //输入 n
    if(n>=0)
    {
        s=1;
        if(n>0)
        {
            i=1;
            while (i<=n) //循环, 求阶乘
            {
                s*=i ;
                i=i+1 ;
            }
        }
    }
}
```

```

    }
}

printf("%d! = %ld \n", n, s);
}

else
    printf("Invalid input! \n");
}

```

运行结果如下：

```
please input n(n>=0): 0
```

```
0!= 1
```

```
please input n(n>=0): 6
```

```
6!= 720
```

```
please input n(n>=0): -2
```

```
Invalid input!
```

[例 2-17] 利用格里高利公式求 p

$p/4=1-1/3+1/5-1/7+\dots$

直到最后一项的绝对值小于等于 10^{-6} 为止。

分析：这也是一个典型的需要利用循环的例子。对上面等式的右边来说，每一个参与运算的分式的分母为递增的奇数，即分别为 1, 3, 5, 7, 9…。我们可以定义一个变量，作为分母，在循环中将该分式与以前运算出的结果相加/相减，并将分母加 2，然后在循环的条件中来判断新的分式是否已经小于等于 10^{-6} ，如果是，则停止循环，否则继续。

程序如下：

```
# include <stdio.h>
# include <math.h>

main ( )
{
    double pi, t;
    long int n, s;
    t=1.0;
    n=1;
    s=1;
    pi=0.0;
    while(fabs(t)>=1e-6)
    {
        pi=pi+t;
        n=n+2;                                //分母加 2
    }
}
```

```

    s=-s;                                //运算符号反向
    t=(float)(s)/(float)(n);
}
pi=pi*4.0;
printf("pi=%lf\n",pi);
}

```

运行结果为：

```
pi=3.141591
```

本题中，将多项式的每一项用 t 表示，s 代表符号，在每一次循环中，只要改变 s、n 的值，就可求出每一项 t。

一般情况下，while 型循环最适合于这种情况：知道控制循环的条件为某个逻辑表达式的值，而且该表达式的值会在循环中被改变，如同例 17 的情况一样。

2.4.2 do...while 语句

在 C 语句中，直到型循环的语句是“do...while”，它的一般形式为：

```
do 语句 while <表达式>
```

其中语句通常为复合语句，称为循环体。

do ...while 语句的基本特点是：先执行后判断，因此，循环体至少被执行一次。但需要注意的是，do ...while 与标准的直到型循环有一个极为重要的区别，直到型循环是当条件为真时结束循环，而 do ...while 语句恰恰相反，当条件为真时循环，一旦条件为假，立即结束循环，请注意 do ...while 语句的这一特点。

[例 2-18] 计算 $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$ ，直到最后一项的绝对值小于 $1e-7$ 时为止

分析：这道题与例 2-17 相似，仍可以使用递推方法来做，不过我们这次将使用 do...while...语句。

算法：让多项式的每一项与一个变量 n 对应，n 的值依次为 1, 3, 5, 7, . . .，从多项式的前一项算后一项，只需将前一项乘一个因子： $(-x^2)/((n-1)*n)$

如果我们用 s 表示多项式的值，用 t 表示每一项的值，那么程序如下：

```

#include <math.h>
#include <stdio.h>
main()
{
    double s, t, x;
    int n;
    printf("please input x :");
    scanf("%lf", &x);                      //输入 x 的值
    t = x;
    n=1;
}

```

```

s = x;
do
{
    n = n + 2 ;
    t = t * (-x * x) / ((float)(n) - 1) / (float)(n);      //递归相加
    s = s + t ;
}while(fabs(t)>=1e-7);
printf("sin(%f )=%lf", x, s);
}

```

运行结果如下：

```

please input x:1.5753
sin(1.575300 ) = 0.999990
please input x:- 0.65
sin(-0.650000 ) =-0.605186

```

2.4.3 for语句

for语句是循环控制结构中使用最广泛的一种循环控制语句，特别适合已知循环次数的情况。它的一般形式为：

for(<表达式1>; <表达式2>; <表达式3>)语句

for语句很好地体现了正确表达循环结构应注意的3个问题：

- (1) 控制变量的初始化。
- (2) 循环的条件。
- (3) 循环控制变量的更新。

表达式1：一般为赋值表达式，给控制变量赋初值；

表达式2：关系表达式或逻辑表达式，循环控制条件；

表达式3：一般为赋值表达式，给控制变量增量或减量。

语句：循环体，当有多条语句时，必须使用复合语句。

首先计算表达式1，然后计算表达式2，若表达式2为真，则执行循环体；否则，退出for循环，执行for循环后的语句。如果执行了循环体，则循环体每执行一次，都计算表达式3，然后重新计算表达式2，依此循环，直至表达式2的值为假，退出循环。

[例2-19] 计算自然数1到n的平方和

分析：读者在经过对上面的一些例子的分析以后，应该可以看出这也是一个需要使用循环的语句。为了说明for语句的使用，在本例中我们将使用本语句来求解。

程序的源代码如下：

```

# include <stdio.h>
# include <math.h>
main()
{
    int i, n;

```

```

float s;
printf("please input n :");
scanf("%d", &n); //输入 n
s = 0.0 ;
for(i=1;i<=n;i++)
    s=s+(float)(i)*(float)(i);
printf("1*1 + 2*2 +...+%d*d = %f\n", n, n, s) ; //用 for 语句求平方和
}

```

运行结果如下：

```

please input n : 5
1*1 + 2*2 + ... + 5* 5 =55.000000

```

for 语句的三个表达式都是可以省略的，但分号“;”绝对不能省略。

for 语句一般有以下几种使用格式：

- for(;;)语句；

这是一个死循环，一般用条件表达式加 break 语句在循环体内适当位置，一旦条件满足时，用 break 语句跳出 for 循环。

例如，在编制菜单控制程序时，可以如下：

```

for( ; ;)
{
    printf("please input choice( Q=Exit):"); // 显示菜单语句块
    scanf (" %c", & ch ) ;
    if (ch=='Q' ) or (ch=='q' ) break; // 语句段
}

```

- for(;表达式 2 ;表达式 3)

使用条件是：循环控制变量的初值不是已知常量，而是在前面通过计算得到，例如：

```

i = m - n ;
.....
for (;i < k;i++) 语句;

```

- for (表达式 1 ;表达式 2 ;)语句

一般当循环控制变量非规则变化，而且循环体中有更新控制变量的语句时使用。

例如：

```

for(i=1;i<=100;)
{
.....
i =i*2+1;
.....
}

```

- for(i=1 , j = n ;i < j ;i + +, j --)语句；

在 for 语句中，表达式 1、表达式 3 都可以有一项或多项，如本例中，表达式 1 同时为 i 和 j 赋初值，表达式 3 同时改变 i 和 j 的值。当有不止一项时，各项之间用逗号“,”分隔。

另外，C语言还允许在循环体内改变循环变量的值，这在某些程序的设计中是很有用的。

到此，我们已经介绍了C语言中三种循环控制语句while、do...while和for语句，下面再讨论两个问题：

- 三种语句的选用

同一个问题，往往既可以用while语句解决，也可以用do...while或者for语句来解决，但在实际应用中，应根据具体情况来选用不同的循环语句，选用的一般原则是：

(1) 如果循环次数在执行循环体之前就已确定，一般用for语句；如果循环次数是由循环体的执行情况确定的，一般用while语句或者do...while语句。

(2) 当循环体至少执行一次时，用do...while语句，反之，如果循环体可能一次也不执行，选用while语句。

- 循环的嵌套

一个循环的循环体中有另一个循环叫循环嵌套。这种嵌套过程可以有很多重。一个循环外面仅包围一层循环叫二重循环；一个循环外面包围两层循环叫三重循环；一个循环外面包围多层循环叫多重循环。

三种循环语句for、while、do...while可以互相嵌套自由组合。但要注意的是，各循环必须完整，相互之间绝不允许交叉。如下面这种形式是不允许的：

```
do
{
    .....
    for (;;)
    {
        .....
    }while();
}
```

[例 2-20] 打印 8 行 7 列的星形矩阵

分析：要打印8行的星，显然要用一个循环；而打印7列，又要用到另外一个循环语句。本例将用for语句的嵌套循环来实现，并一直说明嵌套循环的使用。

源代码如下：

```
# include<stdio.h>
main( )
{
    int i, j;
    for(i=0;i<8;i++) //控制行
    {
        for(j=0;j<7;j++) //控制列
            printf("* ");
        printf("\n"); //换行
    }
}
```

打印结果如下：

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

将程序中 `for(j=0; j<7; j++)` 改为 `for(j=0; j<i; j++)`，用行数来控制每行星号的多少，就可以打印三角形。

2.4.4 break 与 continue 语句

有时，我们需要在循环体中提前跳出循环，或者在满足某种条件下，不执行循环中剩下的语句而立即从头开始新一轮循环，这时就要用到 `break` 和 `continue` 语句。

1. break 语句

在前面介绍 `switch` 语句时，我们已经接触到 `break` 语句，在 `case` 子句执行完后，通过 `break` 语句使控制立即跳出 `switch` 结构。在循环语句中，`break` 语句的作用是在循环体中测试到应立即结束循环时，使控制立即跳出循环结构，转而执行循环语句后的语句。

[例 2-21] 打印半径为 1 到 10 的圆的面积，若面积超过 100，则不予打印

分析：本例要用到一个从 1 到 10 的循环来表示半径，而在后面的条件“面积超过 100”满足的时候，需要中断循环并跳出。出于程序结构化和可读性的考虑，C 语言中不提倡使用类似汇编的直接跳转的语句，所以只能使用 `break`。这也是一个 `break` 关键字的一个典型的应用。

程序的源代码如下：

```
# include <stdio.h>
main( )
{
    int r;
    float area;
    for(r=1;r<=10;r++)
    {
        area=3.141593*r*r; //计算面积
        if(area>100.0) break; //面积超过 100 则跳出
        printf("square=%f\n", area);
    }
    printf("now r=%d\n", r);
}.
```

运行程序后执行结果如下：

```
square = 3.141593
square = 12.566373
square = 28.274338
square = 50.265488
square = 78.539825
now r=6
```

当 break 处于嵌套结构中时，它将只跳出最内层结构，而对外层结构无影响。

2. continue 语句

continue 语句只能用于循环结构中，一旦执行了 continue 语句，程序就跳过循环体中位于该语句后的所有语句，提前结束本次循环周期并开始新一轮循环。

[例 2-22] 计算半径为 1 到 15 的圆的面积，仅打印出超过 50 的圆面积

分析：该例与例 21 不同的是，当条件“面积超过 50”的条件满足的时候才执行循环中的打印语句，而不需要中止这一层循环。当然，本例可以直接将条件变更，改为“面积小于 50”的才打印，即将打印语句跟在 if 之后。不过，为了说明 continue 语句的使用，我们将在这个例子中使用该语句。

源代码如下：

```
# include <stdio.h>
main ( )
{
    int r;
    float area;
    for (r=1;r<=5;r++)
    {
        area=3.141593*r*r;
        if(area<50.0) //如果面积小于 50，跳过循环中后面的语句
            continue;
        printf("square =%f",area);
    }
}
```

执行以后输入结果为：

```
square = 50.265488
square = 78.539825
```

同 break 一样，continue 语句也仅仅影响该语句本身所处的循环层，而对外层循环没有影响。

2.4.5 程序应用举例

[例 2-23] 验证哥德巴赫猜想

任一充分大的偶数，可以用两个素数之和表示，例如：

$$4=2+2$$

$$6=3+3$$

.....

$$98=19+79$$

哥德巴赫猜想是世界著名的数学难题，至今未能在理论上得到证明，自从计算机出现后，人们就开始用计算机去尝试解各种各样的数学难题，包括费马大定理、四色问题、哥德巴赫猜想等，虽然计算机无法从理论上严密地证明它们，而只能在很有限的范围内对其进行检验，但也不失其意义。费马大定理已于 1994 年得到证明，而哥德巴赫猜想这枚数学王冠上的宝石，至今无人能及。

分析：我们先不考虑怎样判断一个数是否为素数，而从整体上对这个问题进行考虑，可以这样做：读入一个偶数 n，将它分成 p 和 q 两个整数，使得 $n=p+q$ 。因此可以令 p 从 2 开始，每次加 1，而令 $q=n-p$ ，如果 p、q 均为素数，则正为所求，否则令 $p=p+q$ 再试。

其基本算法如下：

- ①读入大于 3 的偶数 n。
- ② $P=1$
- ③do {
- ④ $p=p+1; q=n-p;$
- ⑤p 是素数吗？
- ⑥q 是素数吗？
- ⑦}while p、q 有一个不是素数。
- ⑧输出 $n=p+q$ 。

为了判明 p、q 是否是素数，我们设置两个标志量 flagp 和 flagq，初始值为 0，若 p 是素数，令 flagp=1，若 q 是素数，令 flagq=1，于是第 7 步变成：

⑦}while (flagp*flagq==0);

再来分析第 5、第 6 步，怎样判断一个数是不是素数呢？

素数就是除了 1 和它自身外，不能被任何数整除的整数，由定义可知：

2、3、5、7、11、13、17、19 等是素数；

1、4、6、8、9、10、12、14 等不是素数。

要判断 i 是否是素数，最简单的办法是用 2、3、4、……i-1 这些数依次去除 i，看能否除尽，若被其中之一除尽，则 i 不是素数，反之，i 是素数。

但其实，没必要用那么多的数去除，实际上，用反证法很容易证明，如果小于等于 i 的平方根的数都除不尽，则 i 必是素数。于是，上述算法中的第 5 步、第 6 步可以细化为：

⑤p 是素数吗？

flagp =1;

for (j=2;j<=[sqrt(p)];j++)

if p 除以 j 的余数= 0

{flagp=0;

```

break; }

⑥q 是素数吗?
flagq =1;
for (j=2;j<=[sqrt(q)];j++)
if q 除以 j 的余数= 0
{ flagq=0;
break;}
程序如下:

```

```

#include <math.h>
# include <stdio.h>
main ( )
{
    int j, n, p, q, flagp, flagq;
    printf("please input n :");
    scanf("%d",&n);                                //输入 n 的值
    if (((n%2)!=0) || (n<=4))
        printf("input data error!\n");
    else
    {
        p = 1 ;
        do {                                         //运算, 直到 p、q 均为素数时为止
            p = p + 1 ;
            q = n - p ;
            flagp=1;                                //假设 p 是素数
            for(j=2; j <= (int)(floor(sqrt((double)(p)))) ;j++)
            {
                if ((p%j)==0)
                {
                    flagp=0;                         //p 不是素数, flag=0
                    break;
                }
            }
            flagq=1;                                //假设 q 为素数
            for (j=2;j<=(int)(floor(sqrt((double)(q)))) ;j++)
            {
                if ((q%j)==0)
                {
                    flagq= 0 ;                     //q 不是素数, flag=0
                    break;
                }
            }
        }
    }
}

```

```
    }
} while (flagp*flagq==0);
printf("%d = %d + %d \n", n, p, q);
}
}
```

程序运行结果如下：

```
please input n :8
8=3+5
```

```
please input n : 98
98=19+79
```

```
please input n :9
input data error!
```

2.5 小结

在本章中重点介绍了 C 语言的基本概念：变量与常量、算术运算、程序控制语句等，通过例子程序的形式，引导读者掌握编辑 C 语言程序的初步知识。

第3章 C语言高级编程

上一章就C语言编程的基本概念进行了详细的介绍和分析，本章将进一步介绍编写较大的程序所需要的重要特性，包括函数、数组、指针、以及结构体等内容，同样以程序代码的举例形式对各部分进行形象的说明，引导读者循序渐进的掌握C语言的高级编程方法。

3.1 函数与程序结构

函数用于把较大的计算任务分解成若干个较小的任务，使程序人员可以在其他函数的基础上构造程序，而不需要从头做起。一个设计得当的函数可以把具体操作细节对程序中不需要知道它们的那些部分隐藏掉，从而使整个程序结构清楚，阅读简单。

C语言在设计函数时兼顾了效率与易于使用两个方面。一个C程序一般都由许多较小的函数组成，而不是只由几个比较大的函数组成。这些函数可以存放在一个文件中，也可以存放在多个文件中。并且各个文件可以单独编译与库中已经编译过的函数装配在一起。但我们不打算详细讨论这一编译装配过程，因为具体编译与装配细节在各个编译系统中不尽相同。

3.1.1 函数的基本知识

C程序与汇编相比，最大的优势就在于程序的结构化和模块化。在程序编制中，对于匹配模式或匹配内容的查找，往往是非常常见的。下面，我们将使用子函数调用的方式来编写一个程序，该程序可以把输入内容中包含特定的“模式”或字符串的各行打印出来（这是UNIX程序grep的一种特殊情况）。

[例3-1] 查找包含固定字符串的行

分析：如果有一个固定的字符文本，例如下一组文本行：

```
Ah Love! could you and I with Fate conspire  
To grasp this sorry Scheme of Things entire,  
Would not we shatter it to bits -- and then  
Re-mould it nearer to the Heart's Desire!
```

如果我们要查找包含字母字符串“ould”的行，那么我们的程序就会产生如下输出：

```
Ah Love! could you and I with Fate conspire  
Would not we shatter it to bits -- and then  
Re-mould it nearer to the Heart's Desire!
```

这个程序段可以清楚地分成3部分：

- while（还有未处理的行）

- if (该行包含指定的模式)
- 打印该行

虽然可以把所有这些代码都放在主程序 main 中，但是为了我们模块化程序设计的需要，一个更好的方法是把每一部分设计成一个独立的函数。并将不同功能的函数放在不同的文件中。分别处理 3 个较小的部分要比处理一个大的整体容易，因为这样可以把不相关的细节隐藏在函数中，从而减少了相互影响的机会。而且这些函数也可以在其他程序中使用。

我们用 printf 函数来实现“打印该行”，这是一个常用的库函数，另外编写一个 getline 函数来实现“还有未处理的行”，还需编写一个判定“该行包含指定的模式”的函数。

getline 函数的算法如下：

- 读取一个字符。
- 判断得到的字符是否是一行结束的字符 (EOF 或' \n')。
- 如果不是，那么将这个字符存到缓存中，并继续读取新的字符。
- 如果是那么向缓存中写入字符串结束标志' \0'，并返回长度。

程序代码如下：

```
int getline(char s[ ], int lim)
{
    int c, i;
    i = 0;
    while ( -- lim > 0&& ( c = getchar() ) != EOF&&c != '\n' )           //读入一个字符并判断是否满足继续读取的条件
        s[i++] = c;
    if (c == '\n' )
        s[i++] = c;
    s[i] = '\0';
    return i;
}
```

读者从上面的程序中可以看出，我们事实上将 1、2、3 步都包含在了 while 循环的条件下，这对程序保持紧凑来说是有利的，但是并不会带来执行速度上的提高。

对于判定“该行包含指定的模式”的函数，我们可以通过编写一个函数 strindex(s, t) 来解决这个问题，该函数返回字符串 t 在字符串 s 中出现的开始位置或位标，但当 s 中不包含 t 时，返回值为 -1。由于 C 语言数组的下标从 0 开始，下标的值为 0 或正数，故用 -1 之类的负数作为失败信号是比较方便的。若以后需要更复杂的模式匹配，只需替换掉 strindex 函数即可，程序的其余部分可保持不动。在做了这样的设计后，填写程序的细节就比较简单了。下面即整个程序，读者可以看看各个部分是怎样组合在一起的。我们现在所要查找的模式是字面值字符串，它并不是一种最通用的机制。

strindex 函数的源代码如下：

```
int strindex(char s[], char t[])
{
    int i, j, k;
    for ( i = 0; s[i] != '\0'; i++ ) {                                //
```

```

搜索
    for ( j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++ ); // 

查找
    if ( k > 0 && t[k] == '\0' ) //匹配
        return i;
    }
    return -1;
}

```

剩下的就只需要编写主函数了。主函数的内容应该是包含所有的逻辑操作，包括决定何时调用各个子函数等。对于本例来说，主函数的逻辑并不复杂，只需要读入字符串、查找、判断、输出就可以了，并没有太多的程序分支跳转。

主程序代码如下：

```

main ( )
{
    char line[MAXLINE];
    int found = 0;

    while ( getline(line, MAXLINE) > 0 ) //读入
        if ( strindex(line, pattern) >= 0 ) { //查找
            printf( "%s", line );
            found++;
        }
    printf("\n%d match found", found); //输出
}
}

```

为了结构清晰，我们将主程序和子程序分别存放于2个文件中，即 main.c 和 function.c 中，另外，由于主函数所调用的函数不在 main.c 中，所以需要使用 extern 关键字来声明函数。该声明放在 def.h 文件中。

所有文件和完整的代码如下：

```

//*****
//          main.c          *
//***** 

#include <stdio.h>
#include "def.h"

#define MAXLINE 100//最大输入行长度
char pattern[] = "ould";//要查找的模式
// 找出所有与模式匹配的行

main ( )
{
    char line[MAXLINE];
    int found = 0;
}

```

```

        while ( getline(line, MAXLINE) > 0 )           //读入
            if ( strindex(line, pattern) >= 0 ) {          //查找
                printf( "%s", line);
                found++;
            }
        printf("\n%d match found", found);             //输出

    }

*****
*          function.c      *
*****
int getline(char s[ ], int lim)
{
    int c, i;
    i = 0;
    while ( -- lim > 0&& ( c = getchar() ) != EOF&&c != '\n') //读入一个字符并判断是否满足继续
        //读取的条件
    s[i++] = c;
    if (c == '\n' )
        s[i++] = c;
        s[i] = '\0';
    return i;
}

// strindex : 返回 t 在 s 中的位置, 若未找到则返回-1
int strindex(char s[], char t[])
{
    int i, j, k;
    for ( i = 0; s[i] != '\0'; i++ ) // 搜索
        for ( j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++ ); // 查找
    if ( k > 0 && t[k] == '\0' ) //匹配
        return i;
    return -1;
}

*****

```

```

*           def.h          *
*****
extern int getline (char line[ ], int max);
extern int strindex(char source[ ], char searchfor[ ]);
```

每一个函数定义均具有如下形式：

返回类型函数名（变元说明表）

{

说明序列与语句序列

}

函数定义的各个部分都可以缺省。最简单的函数结构如下：

```
dummy( ) { }
```

这个函数什么也不做、也不返回任何结果。像这种函数有时很有用，它可以在程序开发期间用作占位符。如果在函数定义中省略了返回类型，则默认为 int。

程序是变量定义和函数定义的集合。函数之间的通信可以通过变元、函数返回值以及外部变量进行。函数可以以任意次序出现在源文件中。源程序可以分成多个文件，只要不把一个函数分在几个文件中就行。

return 语句用于从被调用函数向调用者返回值，return 之后可以跟任何表达式，格式如下：

```
return 表达式;
```

在必要时要把表达式转换成函数的返回类型（结果类型）。表达式两边往往要加一对圆括号，但不是必需的，而是可选的。

调用函数可以随意忽略掉返回值。而且，return 之后也不一定要跟一个表达式。在 return 之后没有表达式的情况下，不向调用者返回值。当被调用函数因执行到最后的右花括号而完成执行时，控制同样返回调用者（不返回值）。如果一个函数在从一个地方返回时有返回值而从另一个地方返回时没有返回值，那么这个函数虽然不一定非法，但可能存在一个问题。在任何情况下，如果一个函数不能返回值，那么它的“值”肯定是没有用的。

上面的模式查找程序从主程序 main 中返回一个状态，即所匹配的字符串的数目。这个值可以在调用该程序的环境中使用。

在不同系统上对驻留在多个源文件中的 C 程序的编译与载入机制有很大的区别。例如，在 UNIX 系统上是用在前文中已提到过的 cc 命令来完成这一任务的。假定有 3 个函数分别存放在名为 main.c、getline.c 与 strindex.c 的 3 个文件中，那么命令 ccmain.cgetline.cstrindex.c 用于编译这 3 个文件，并把目标代码分别存放在文件 main.o、getline.o 与 strindex.o 中，然后再把这 3 个文件一起载入到可执行文件 a.out 中。如果源程序中出现了错误（比如文件 main.c 中出现了错误），那么可以用命令“cc main.c getline.o strindex.o”对 main.c 文件重新编译，并将编译的结果与以前已编译过的目标文件 getline.o 和 strindex.o 一起载入。cc 命令用.c 与.o 这两种扩展名来区分源文件与目标文件。

3.1.2 返回非整数值的函数

到目前为止，我们所讨论的函数均不返回任何值（void）或只返回 int 类型的值。假如

一个函数必须返回其他类型的值，那么该怎么办呢？许多数值函数（如 sqrt、sin 与 cos 等函数）返回的是 double 类型的值，另一些专用函数则返回其他类型的值。

为了说明让函数返回非整数值的方法，编写一个简单的进行加法运算的计算器，并使用函数 atof(s)，它用于把字符串 s 转换成相应的双精度浮点数。

[例 3-2] 简单的加法计算器

atof 函数要处理可选的符号与小数点以及整数部分与小数部分。该版本并不是一个高质量的输入转换函数，它所占用的资源比最优情况要多。标准库中包含了具有类似功能的 atof 函数，它在头文件<stdlib.h>中说明。

首先，由于 atof 函数返回值的类型不是 int，因此在该函数中必须说明它所返回值的类型。返回值类型的名字要放在函数名字之前：

```
#include <ctype.h>
// 把字符串 s 转换成相应的双精度浮点数
double atof( char s[ ] )
{
    double val, power;
    int i, sign;
    for ( i = 0; isspace(s[i]); i++ )                                // 跳过空白 ;
        sign = (s[i] == '-' ) ? -1 : 1;
    if ( s[i] == '+' || s[i] == '-' )                                    // 判断加法或减法
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val +(s[i] -'0' );
    if (s[i] == '.')
        i++;
    for ( power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val +(s[i] -'0' );
        power *= 10.0;
    }
    return sign * val/power;
}
```

其次，调用函数必须知道 atof 函数返回的是非整数值。为了保证这一点，一种方法是在调用函数中显式说明 atof 函数。下面所示的基本计算器程序（仅适用于支票簿计算）中给出了这个说明，程序一次读入一行数（一行只放一个数，数的前面可能有一个正负号），并把它们加在一起，在每一次输入后把这些数的连续和打印出来：

```
#include <stdio.h>
// 基本计算器程序
main()
{
    double sum, atof( char s[ ]);
```

```

char sum1[10], sum2[10];
scanf("%s", sum1);                                //输入两个数字
scanf("%s", sum2);
sum=atof(sum1)+atof(sum2);                      //计算
printf("%s+%s=%f\n", sum1, sum2, sum);          //打印结果
}

```

其中，说明语句

```
double sum, atof ( char [ ] );
```

表明 sum 是一个 double 类型的变量，atof 是一个具有 char[]类型的变元且返回值类型为 double 的函数。

函数 atof 的说明与定义必须一致。如果 atof 函数与调用它的主函数 main 放在同一源文件中，并且具有不一致的类型，那么编译程序将会检测出这个错误。但是，如果 atof 函数是独立编译的（这是一种更可能的情况），那么这种不匹配的错误就不会被检测出来，atof 函数将返回 double 类型的值，而 main 函数则将之处理为 int 类型，从而这样所求得的结果毫无意义。

如果在前面已经说明过的某个名字出现在某个表达式中并且左边跟一个左圆括号，那么就根据上下文认为该名字是函数名字，该函数的返回值类型为 int，但对变元没有给出上面信息。而且，如果一个函数说明中不包含变元，比如

```
double atof ( );
```

那么也认为没有给出 atof 函数的变元信息，所有参数检查都被关闭。对空变元表做这种特殊的解释是为了使新的编译程序能编译比较老的 C 程序。但是，在新程序中也如此做是不明智的。如果一个函数有变元，那么需要说明它们；如果没有变元，那么使用 void。

借助恰当说明的 atof 函数，可以编写出函数 atoi（将字符串转换成整数）：

```
// atoi : 利用 atof 函数把字符串 s 转换成整数
int atoi( char s[ ] )
{
    double atof(char s[ ]);
    return (int) atof ( s );
}
```

请注意其中说明和 return 语句的结构。在 return 语句：

```
return 表达式;
```

表达式的值在返回之前被转换成函数的返回类型。因此，如果对 atof 函数的调用直接出现在 atoi 函数中的 return 语句中，如

```
return atof ( s );
```

那么，由于函数 atoi 的返回值类型为 int，系统要把 atof 函数的 double 类型的结果返回值自动转换成 int 类型。然而，这种操作可能会丢失信息，有些编译程序可能会为此给出警告信息。在此函数中由于采用了强制转换的方法显式地表明了所要做的转换操作，可以屏蔽有关警告信息。

3.1.3 外部变量

C 程序由一组外部对象（外部变量或函数）组成。形容词 `external` 与 `internal` 相对，`internal` 用于描述定义在函数内部的函数变元以及变量。外部变量在函数外面定义，故可以在许多函数中使用。由于 C 语言不允许在一个函数中定义其他函数，因此函数本身是外部的。在缺省情况下，外部变量与函数具有如下性质：所有通过名字对外部变量与函数的引用（即使这种引用来自独立编译的函数）都是引用的同一对象（标准中把这一性质叫做外部连接）。在这个意义上，外部变量类似于 FORTRAN 语言的 COMMON 块或 Pascal 语言中最外层分程序中说明的变量。后面将介绍如何定义只能在某个源文件使用的外部变量与函数。

由于外部变量是可以全局访问的，这就为在函数之间交换数据提供了一种可以代替函数变元与返回值的方法。任何函数都可以用名字来访问外部变量，只要这个名字已在某个地方做了说明。

如果要在函数之间共享大量的变量，那么使用外部变量要比使用一个长长的变元表更方便也更有效。然而，正如在前面所指出的，这样使用务必小心，因为这样可能对程序结构产生不好的影响，而且可能会使程序在各个函数之间产生太多的数据联系。

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

3.1.4 作用域规则

用以构成 C 程序的函数与外部变量没有必要同时编译，一个程序可以放在几个文件中，可以从库中调入已编译过的函数。我们关心的问题主要有：

- 怎样编写说明才能使所说明的变量在编译时被认为是正确的？
- 怎样安排说明才能保证在程序载入时各部分能正确相连？
- 怎样组织说明才能使得只需一份拷贝？
- 怎样初始化外部变量？

为了便于讨论这些问题，我们把计算器程序组织在若干个文件中。从实用角度看，计算器程序比较小，不值得分几个文件存放，但通过这个例子可以很好地说明在较大的程序中所遇到的有关问题。

一个名字的作用域指程序中可以使用该名字的部分。对于在函数开头说明的自动变量，其作用域是说明该变量名字的函数。在不同函数中说明的具有相同名字的各个局部变量毫不相关。

对于函数的参数也如此，函数参数实际上可以看作是局部变量。

外部变量或函数的作用域从其说明处开始一直到其所在的被编译的文件的末尾。例如，如果 `main`、`sp`、`val`、`push` 与 `pop` 是 5 个依次定义在某个文件中的函数与外部变量，即：

```
main( ) { ...}
int sp = 0;
double val[MAXVAL];
```

```
void push( double f ) {...}
double pop( void ) {...}
```

那么，在push与pop这两个函数中不需做任何说明就可以通过名字来访问变量sp与val，但是，这两个变量名字不能用在main函数中，push与pop函数也不能用在main函数中。另一方面，如果一个外部变量在定义之前就要使用到，或者这个外部变量定义在与所要使用它的源文件不相同的源文件中，那么要在相应的变量说明中强制性地使用关键词extern。将对外部变量的说明与定义严格区分开来很重要。变量说明用于通报变量的性质（主要是变量的类型），而变量定义则除此以外还引起存储分配。如果在函数的外部包含如下说明：

```
int sp;
double val[MAXVAL];
```

那么这两个说明定义了外部变量sp与val，并为之分配存储单元，同时也用作供源文件其余部分使用的说明。另一方面，如下两行：

```
extern int sp;
extern double val[MAXVAL];
```

为源文件剩余部分说明了sp是一个int类型的外部变量，val是一个double数组类型的外部变量（该数组的大小在其他地方确定），但这两个说明并没有建立变量或为它们分配存储单元。

在一个源程序的所有源文件中对一个外部变量只能在某个文件中定义一次，而其他文件可以通过extern说明来访问它（在定义外部变量的源文件中也可以包含对该外部变量的extern说明）。

在外部变量的定义中必须指定数组的大小，但在extern说明中则不一定要指定数组的大小。外部变量的初始化只能出现在其定义中。

假定函数push与pop在一个文件中定义，变量val与sp在另一个文件中定义并初始化（虽然一般不可能这样组织程序）。这些定义与说明必须把这些函数和变量捆在一起：

在文件file1中：

```
extern int sp;
extern double val [ ];
void push( double f ) { ...}
double pop( void ) { ...}
```

在文件file2中：

```
int sp = 0;
double val[MAXVAL];
```

由于文件file1中的extern说明不仅放在函数定义的外面而且还放在它们前面，故它们适用于所有函数，这一组说明对文件file1已足够了。如果sp与val的定义跟在对它们的使用之后，那么也要这样来组织文件。

3.1.5 头文件

下面考虑把一个比较大的程序分成若干个源文件。主函数main单独放在文件main.c中，而其他文件也按一定规则分开放置。之所以把它们分开，是因为在实际程序中它们来自于一

个独立编译的库。

还有一个问题需要考虑，即这些文件之间的定义与说明的共享问题。我们将尽可能使所要共享的部分集中在一起，以使得只需一个拷贝，当要对程序进行改进时也能保证程序的正确性。

我们对如下两个方面做了折衷：一方面是对每一个文件只访问它完成任务所需要的信息的要求，另一方面是维护较多的头文件比较困难的现实。对于某些中等规模的程序，最好是只使用一个头文件来存放程序中各个部分需要共享的实体，这是我们在这里所做的结论。对于比较大的程序，需要做更精心的组织，使用更多的头文件。

3.1.6 静态变量

`static` 说明适用于外部变量与函数，用于把这些对象的作用域限定为被编译源文件的剩余部分。通过外部 `static` 对象，可以把名字隐藏在函数组合中，使得这两个外部变量可以被某些函数共享，但不能被函数的调用者访问。

可以在通常的说明之前以关键词 `static` 来指定静态存储。如果把上述两个函数与两个变量放在一个文件中编译，如下：

```
static char buf[BUFSIZE];           // 供 ungetch 函数使用的缓冲区
static int bufp = 0;                // 缓冲区 buf 的下一个自由位置
int getch( void ) { ... }
void ungetch( int c ) { ... }
```

那么其他函数不能访问变量 `buf` 与 `bufp`，做这两个名字不会和同一程序中其他文件中的同名名字相冲突。基于同样的理由，可以通过把变量 `sp` 与 `val` 说明为静态的，使这两个变量只能供进行栈操作的 `push` 与 `pop` 函数使用，而对其他文件隐藏。

外部 `static` 说明最常用于说明变量，当然它也可用于说明函数。通常情况下，函数名字是全局的，在整个程序的各个部分都可见。然而，如果把一个函数说明成静态的，那么该函数名字就不能用在除该函数说明所在的文件之外的其他文件中。

`static` 说明也可用于说明内部变量。内部静态变量就像自动变量一样局部于某一特定函数，只能在该函数中使用，但与自动变量不同的是，不管其所在函数是否被调用，它都是一直存在的，而不像自动变量那样，随着所在函数的调用或退出而存在或消失。换而言之，内部静态变量是一种只能在某一特定函数中使用的但一直占据存储空间的变量。

3.1.7 寄存器变量

`register` 说明用于提醒编译程序所说明的变量在程序中使用频率较高。其思想是，将寄存器变量放在处理器的寄存器中，这样可以使程序更小、执行速度更快。但编译程序可以忽略此选项。

`register` 说明如下所示：

```
register int x;
register char c;
```

寄存器说明只适用于自动变量以及函数的形式参数。对于后一种情况，如下：

```
f( register unsigned m, register long n )
{
    register int i;
    ...
}
```

在实际使用时，由于硬件环境的实际情况，对寄存器变量会有一些限制。在每一个函数中只有很少的变量可以放在寄存器中，也只有某些类型的变量可以放在寄存器中。然而，过量的寄存器说明并没有什么害处，因为对于过量的或不允许的寄存器变量说明，编译程序可以将之忽略掉。另外，不论一个寄存器变量实际上是不是存放在寄存器中，它的地址都是不能访问的。对寄存器变量的数目和类型的具体限制视不同的处理器而有所不同。

3.1.8 分程序结构

C语言不是Pascal等语言意义上的分程序结构的语言，因为它不允许在函数中定义函数。但另一方面，变量可以以分程序结构的形式在函数中定义。变量的说明（包括初始化）可以跟在用于引入复合语句的左花括号的后面，而不是只能出现在函数的开始部分。以这种方式说明的变量可以隐藏在该分程序外面说明的同名变量，并在与该左花括号匹配的右花括号出现之前一直存在。例如，在如下程序段中：

```
if ( n > 0 ) {
    int i;                                // 说明一个新的 i
    for ( i = 0; i < n; i++ )
    ...
}
```

变量*i*的作用域是if语句的“真”分支，这个*i*与在该分程序之外说明的*i*无关。在分程序中说明与初始化的自动变量每当进入这个分程序时就被初始化。静态变量只在第一次进入分程序时初始化一次。

自动变量（包括形式参数）也隐藏同名的外部变量与函数。对于如下说明：

```
int x;
int y;
f ( double x )
{
    double y;
    ...
}
```

在函数*f*内，所出现的*x*引用的是参数，其类型为double，而在函数*f*之外，引用的是类型为int的外部变量。对变量*y*也如此。

就风格而言，最好避免出现变量名字隐藏外部作用域中同名名字的情况，否则可能会出现大量混乱与错误。

3.1.9 初始化

在前几节中已多次提到过初始化的概念，但一直没有认真讨论它。这一节在前面讨论的基础上总结一些初始化原则。

在没有显式初始化的情况下，外部变量与静态变量都被初始化为 0，而自动变量与寄存器变量的初值则没有定义（即，其初值是“垃圾”）。

在定义纯量变量时，可以通过在所定义的变量名字后加一个等号与一个表达式来进行初始化：

```
int x = 1;
char squote = '\'';
long day = 1000L * 60L * 60L * 24L; // 每天的毫秒数
```

对于外部变量与静态变量，初始化符必须是常量表达式，初始化只做一次（从概念上讲是在程序开始执行前进行初始化）。对于自动变量与寄存器变量，则在每当进入函数或分程序时进行初始化。

对于自动变量与寄存器变量，初始化符不一定限定为常量：它可以是任何表达式，其中可以包含前面已定义过的值甚至可以包含函数调用。对前一章 2.3 节介绍的二分查找程序的初始化可以用如下形式：

```
int binsearch( int x, int v[ ], int n )
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

来代替原来的形式：

```
int low, high, mid;
low = 0;
high = n - 1;
```

实际上，自动变量的初始化部分就是赋值语句的缩写。到底使用哪一种形式还是一个尚待尝试的问题，我们一般使用显式的赋值语句，因为说明中的初始化符比较难以为人们发现，并且距使用点比较远。

数组的初始化也是通过说明中的初始化符完成的。数组初始化符是用花括号括住并用逗号分隔的初始化符序列。例如，当要用每一个月的天数来初始化数组 days 时，可用如下变量定义：

```
int days[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

当数组的大小缺省时，编译程序就通过统计花括号中初始化符的个数作为数组的长度，本例中数组的大小为 12 个元素。

如果初始化符序列中初始化符的个数比数组元素数少，那么对于没有得到初始化的数组元素在该数组为外部变量、静态变量与自动变量时被初始化为 0。如果初始化符序列中初始化符的个数比数组元素数多，那么就是错误的。我们既无法一次性地为多个数组元素指定一

个初始化符，也不能在没有指定前面数组元素值的情况下初始化后面的数组元素。

字符数组的初始化比较特殊，可以用一个字符串来代替用花括号括住并用逗号分隔的初始化符序列：

```
char pattern [] = "ould";
```

它是如下虽然长些但却与之等价的定义的缩写：

```
char pattern [] = { 'o', 'u', 'l', 'd', '\0' };
```

在此情况下，数组的大小是 5（4 个字符外加一个字符串结束符'\0'）。

3.1.10 递归

C 函数可以递归调用，即一个函数可以直接或间接调用自己。在这里请读者注意，如果使用 KEIL 的 C51 编译器，对于要递归调用的函数，需要使用 reentrant 关键字。

考虑把一个数作为字符串打印的情况。如前所述，数字是以相反的次序生成的：低位数字先于高位数字生成，但它们必须以相反的次序打印出来。

对这一问题有两种解决方法。一种方法是将生成的各个数依次存储到一数组中，然后再以相反的次序把它们打印出来，正如 3.2 节对 itoa 函数所做的那样。另一种方法是使用递归解法，用于完成这一任务的函数 printd 首先调用自身处理前面的（高位）数字，然后再把后面的数字打印出来。这个版本不能处理最大的负数。

```
#include <stdio.h>
// printd : 以十进制打印数 n
reentrant printd(int n)
{
    if ( n<0 ) {
        putchar( '-' );
        n = -n;
    }
    if ( n/10 )
        printd( n/10 );
    putchar( n %10 + '0' );
}
```

当一个函数递归调用自身时，每一次调用都会得到一个与以前的自动变量集合不同的新的自动变量集合。因此，在调用 printd(123) 时，第一次调用 printd 的变元 n= 123。它把 12 传递给对 printd 的第二次调用，后者又把 1 传递给对 printd 的第三次调用。第三次对 printd 的调用将先打印 1，然后再返回到第二次调用。从第三次调用返回后的第二次调用同样先打印 2，然后再返回到第一次调用。后者打印出 3 后结束执行。

另一个用于说明递归的一个例子是快速排序。快速排序算法是 C. A. R. Hoare 于 1962 年发明的。对于一个给定的数组，从中选择一个元素（叫做分区元素），并把其余元素划分成两个子集合——一个是由所有小于分区元素的元素组成的子集合，另一个是由所有大于等于分区元素的元素组成的子集合。对这样两个子集合递归应用同一过程。当某个子集合中的元素数小于两个时，这个子集合不需要再进行排序，故递归停止。

下面这个版本的快速排序函数可能不是最快的一个，但它是最简单的一个。在每一次划分子集合时都选取各个子数组的中间元素。

```
// qsort : 以递增顺序对 v[left] …v[right] 进行排序
reentrant qsort( int v[], int left, int right )
{
    int i, last;
    void swap( int v[], int i, int j );
    if ( left >= right )                                // 若数组所包含的元素数少于两个，则什么也不做
        return;
    swap( v, left, (left + right)/2 );                  // 把分区元素移到 v[0]
    last = left;
    for ( i = left+1; i <= right; i++ )                // 分区
        if ( v[i] < v[left] )
            swap( v, ++last, i );
    swap( v, left, right );                            // 恢复分区元素
    qsort( v, left, last-1 );
    qsort( v, last+1, right );
}
```

这里把数组元素交换操作作为一个独立的函数 swap，是因为它在 qsort 函数中要使用 3 次。

```
// swap : 交换 v[i]与 v[j]的值
void swap( int v[], int i, int j )
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

请读者要注意的是，递归的执行速度并不快，但递归代码比较紧凑，要比相应的非递归代码易于编写与理解。在描述诸如树等递归定义的数据结构时使用递归尤其方便。

3.2 数组

数组是一个由若干同类型变量组成的集合，引用这些变量时可用同一名字。数组均由连续的存储单元组成，最低地址对应于数组的第一个元素，最高地址对应于最后一个元素，数组可以是一维的，也可以是多维的。

3.2.1 一维数组

一维数组的一般说明形式如下：

```
type-specifier var_name [size];
```

在C语言中，数组必须显示地说明，以便编译程序为它们分配内存空间。在上式中，类型说明符指明数组的类型，也就是数组中每一个元素个数，一维数组的总字节数可按下式计算：

`sizeof(类型)*数组长度=总字节数`

[例3-3] 给整型数组赋值

```
main( )
{
    int x[10];           // 定义包含10个整型数的数组，引用为x[0],x[1]...
x[9]
    int t;
    for (t=0; t<10; ++t) x[t]=t;
}
```

C语言并不检验数组边界，因此，数组的两端都有可能越界而使其他变量的数组甚至程序代码被破坏。在需要的时候，数组的边界检验便是程序员的职责。例如，当使用`gets()`接收字符输入时，必须确认字符数组的长度足以存放最长的字符串。

一维数组在本质上是由同类数据构成的表。

1. 向函数传递一维数组

将一维数组传递给函数时，把数组名作为参数直接调用函数即可，无需任何下标。这样，数组的第一个元素的地址将传递给该函数。C语言并不是将整个数组作为实参来传递，而是用指针来代替它。例如，下面的程序将数组*i*的第一个元素的地址传递给函数`func1()`。

```
main( ){
    int i[10];
    func1(i);           // 函数调用，实参是数组名
    ...
}
```

函数若要接收一维数组的传递，则可以用下面的两种方法之一来说明形式参数；（1）有界数组；（2）无界数组。例如，函数`func1()`要接收数组*i*可如下说明：

```
func1(str)
char str[10];           // 有界数组，数组的下标只能小于或等于传递数组的大小。
{
    .
    .
}
```

也可说明为：

```
func1(str)
char str[ ];
{
.
.
.
}
```

这两种说明方法的效果是等价的，它们都通知编译程序建立一个字符指针。第一种说明使用的是标准的数组说明；后一种说明使用了改进型的数组说明，它只是说明函数将要接收一个具有一定长度的整型数组。细想就会发现，就函数而言，数组究竟有多长并无关紧要，因为 C 语言并不进行数组的边界检验。事实上，就编译程序而言，下面的说明也是可行的。

```
func1 (str);
int str[32];
{
.
.
.
```

因为编译程序只是产生代码使函数 func1() 接收一个指针，并非真正产生一个包含 32 个元素的数组。

2. 字符串使用的一维数组

显然，一维数组的最普通的用法是作为字符串。在 C 语言中，字符串被定义为一个以空字符终结的字符数组。空字符以' \0' 来标识，它通常是不显示的。因此，在说明字符数组时，必须比它要存放的最长字符串多一个字符。例如，假如要定义一个存放长度为 10 的字符串的数组 s，可以写成：

```
char s[11];
```

这样就给字符串末尾的空字符保留了空间。

尽管 C 语言并不把字符串定义为一种数据类型，但却允许使用字符串常量。字符串常量是由双引号括起来的字符表。例如，下面两个短语均为字符串常量：

```
"hello there"
```

```
"this is a test"
```

字符串的末尾不必加空字符，C 编译程序会自动完成这一工作。

C 语言支持多串操作函数，最常用的有：

strcpy(s1, s2)：将 s2 拷贝到 s1

strcat(s1, s2)：将 s2 连接到 s1 的末尾

strlen(s1)：返回 s1 的长度

strcmp(s1, s2)：若 s1 与 s2 相等，返回值为 0；若 s1 < s2，返回值小于 0；若 s1 > s2，返回值大于 0。

更详细的内容请参见 C 编译器一章的库函数介绍部分。

例 3-4 说明了这些函数的用法。

[例3-4] 字符串长度

分析：本例只是为了提醒读者注意，在定义字符串变量的时候需要用字符型数组。另外，本例也使用了提到的几个字符串操作库函数，所以程序主体很简单。

程序代码如下：

```
#include <stdio.h>
#include <string.h>
main ( )
{
    char s1[80], s2[80]; // 定义字符数组
    printf("input string 1:\n");
    gets(s1, 80); // 输入字符串
    printf("input string 2:\n");
    gets(s2, 80);
    printf ("lengths: %d %d \n", strlen(s1), strlen(s2));
    if (!strcmp(s1, s2))
        printf("the strings are equal \n");
    strcat(s1, s2);
    printf("%s\n", s1) ;
}
```

切记，当两个串相等时，函数 `strcmp()` 将返回 `False`，因而当测试串的等价性时，要像前例中的那样，必须用逻辑运算符 `!` 将测试条件取反。

当程序运行并以“hello”和“hello”这两个串作为输入时，其输出为：

```
input s1:hello
input s2:hello
lengths:5 5
The strings are equal
hello hello
```

3.2.2 二维数组

1. 二维数组的一般形式

C语言允许使用多维数组，最简单的多维数组是二维数组。实际上，二维数组是以一维数组为元素构成的数组，要将 `d` 说明成大小为(10, 20)的二维整型数组，可以写成：

```
int d[10][20]
```

注意，C不像其他大多数计算机语言那样使用逗号区分下标，而是用方括号将各维下标括起，并且，数组的二维下标均从0计算。

与此相似，要存取数组 `d` 中下标为(3, 5)的元素可以写成：

```
d[3][5]
```

在例 3-5 中，整数 1 到 12 被装入一个二维数组。

[例 3-5] 给二维数组数组赋值。

分析：对二维数组的赋值与一般变量的赋值基本相同，只需要使用前面学过的循环语句对数组中的每一个元素分别赋值就可以了。

源程序如下：

```
main ( )
{
    int t, i, num[3][4];
    for (t=0; t<3; ++t)                                //赋值循环
        for (i=0; i<4; ++i)
            num[t][i]=(t*4)+i+1;
    while (1) {};
}
```

在此例中， $\text{num}[0][0]$ 的值为 1， $\text{num}[0][2]$ 的值为 3，……， $\text{num}[2][3]$ 的值为 12。可以将该数组想象为如下的组合：

0	1	2	3	
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

二维数组以行~列矩阵的形式存储。第一个下标代表行，第二个下标代表列，这意味着按照在内存中的实际存储顺序访问数组元素时，右边的下标比左边的下标的变化快一些。实际上，第一下标可以认为是行的指针。

记住，一旦数组被声明，所有的数组元素都将分配相应的存储空间。对于二维数组可用下列公式计算所需的内存字节数：

$$\text{行数} \times \text{列数} \times \text{类型字节数} = \text{总字节数}$$

因而，假定为双字节整型，大小为 (10, 5) 的整型数组将需要： $10 \times 5 \times 2 = 100$ 字节的空间，当二维数组用作函数的参数时，实际上传递的是第一个元素 ($\text{Num}[0][0]$) 的指针。不过该函数至少要定义第二维的长度，这是因为 C 编译程序若要使得对数组的检索正确无误，就需要知道每一行的长度。例如，将要接收大小为 (10, 10) 的二维数组的函数，可以说明如下：

```
func1(x)
int x[][10]
{
    .
    .
}
```

第一维的长度也可指明，但不是必要的。

C 编译程序对函数中变量 $X[2][4]$ 处理时，需要知道二维的长度。若行长度没定义，那么它就不可能知道第三行从哪儿开始。

[例 3-6] 二维数组操作

用一个二维数组存放某一教师任教的各班学生的分数。假定教师有三个班，每班最多有三十名学生。注意各函数存取数组的方法。

分析：我们可以将上面的问题分解为几个部分，即：输入成绩功能和显示成绩功能，分别使用了 disp_grades() 函数和 enter_grades() 函数。对于输入成绩的部分，专门使用一个子函数 get_grades 来输入某个学生的成绩。

源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

#define classes 3
#define grades 30

void disp_grades(int g[ ][grades]) //显示学生成绩
{
    int t, i;
    for(t=0; t<classes; ++t) {
        printf("class # %d:\n", t + 1);
        for(i=0; i<grades; ++i)
            printf("grade for student #%d is %d\n", i+1, g[t][i]);
    }
}

int get_grades(int num)
{
    char s[80];
    printf("enter grade for student # %d:\n", num+1);
    scanf("%s", s); //输入成绩
    return atoi(s));
}

void enter_grades(int a[][grades])
{
    int t, i;
    for (t=0; t<classes; t++)
    {
        printf (" class #%d:\n", t+1);
        for (i=0; i<grades; i++)
    }
```

```

        a[t][i]=get_grades(i);
    }

}

void main (void) {
    int a[classes][grades]; //定义二维数组，每行存放一个班学生成绩
    char ch;
    for( ;;)
    {
        do { //菜单显示
            printf("(E)nter grades\n");
            printf("(R)eport grades\n");
            printf("(Q)uit\n");
            ch=toupper(getchar()); //将键盘输入字符转换大写
        } while(ch!='E' && ch!='R' && ch!='Q');
        switch(ch) //判断输入并作相应操作
        {
            case 'E':
                enter_grades(a);
                break;
            case 'R':
                disp_grades(a);
                break;
            case 'Q':printf("Quit This Program");
                while(1);
        }
    }
    while (1) {};
}

```

我们将实际问题简化为共有 2 个班，每班两个学生，即将程序中的常量定义修改如下：

```
#define classes 2
#define grades 2
```

运行程序结果如下：

```
(E)nter grades
(R)eport grades
(Q)uit:e
class #1:
enter grade for student #1: 7 8
enter grade for student #2: 8 9
```

```

class #2
enter grade for student #1: 9 8
enter grade for student #2: 9 0

(E)nter grades
(R)eport grades
(Q)uit:e
class #1
grade for student #1 is 78
grade for student #2 is 89
class #2
grade for student #1 is 98
grade for student #2 is 90

(E)nter grades
(R)eport grades
(Q)uit:q

```

运行程序，我们首先看到一个菜单，选择“e”输入成绩，选择“r”显示成绩，选择“q”退出。atoi()函数用于将实参字符串转换为整型。

2. 字符串数组

程序设计中经常要用到字符串数组。例如，数据库的输入处理程序就要将用户输入的命令与存在字符串数组中的有效命令相比较，检验其有效性。可用二维字符数组的形式建立字符串数组，左下标决定字符串的个数，右下标说明串的最大长度。例如，下面的语句定义了一个字符串数组，它可存放 30 个字符串，串的最大长度为 80 个字符：

```
char str_array[30][80];
```

要访问单独的字符串是很容易的，只需标明左下标就可以了。例如，下面的语句以数组 str_array 中的第 3 个字符串为参数调用函数 gets()。

```
gets(str_array[2]);
```

该语句在功能上等价于：

```
gets(&str_array[2][0]);
```

但第一种形式在专业程序员编制的 C 语言程序中更为常见。

为帮助读者理解字符串数组的用法，我们研究例 3-7。它以一个字符串数组为基础做简单的文本编辑。

[例 3-7] 使用字符串数组

分析：对于本例，可以先定义一个二维的字符数组，以为表示字符串的最大长度，第二维表示字符串的数目。字符传输组完全可以使用有关字符变量或者一般字符串变量使用的库函数。

源程序如下：

```

#include <stdio.h>
#define MAX 100
#define LEN 80
char text[MAX][LEN];
void main (void) {
    register int t ,i ,j ;
    for(t=0;t<MAX; t++) //逐行输入字符串
    {
        gets(text[t],LEN);
        if(!text[t][0]) break; // 空行退出
    }

    for(i=0;i<t;i++) //按行,逐个字符输出字符串
    {
        for(j=0; text [i][j];j++)
            putchar(text [i][j]);
        putchar( '\n');
    }
    while (1) {};
}

```

该程序输入文本行直至遇到一个空行为止，而后每次一个字符地重新显示各行。

3.2.3 多维数组

C 语言允许有大于二维的数组，维数的限制（如果有的话）是由具体编译程序决定的。多维数组的一般说明形式为：

Type-specifier name [a][b][c]...[z];

由于大量占有内存的关系，二维或更多维数组较少使用。如前所述，当数组定义之后，所有的数组元素都将分配到地址空间。例如，大小为 (10, 6, 9, 4) 的四维字符数组需要 $10 \times 6 \times 9 \times 4$ 即 2160Byte。

如果上面的数组是两字节整型的，则需要 4320Byte，若该数组是双精度浮点数的（假定每个双精度浮点数为 8Byte），则需要 34560Byte，存储量随着维数的增加呈指数增长。

关于多维数组，需要注意一点：计算机要花大量时间计算数组下标，这意味着存取多维数组中的元素要比存取一维数组的元素花更多的时间。因此，大量的多维数组一般采用 C 语言动态分配函数及指针的方法，每次对数组的一部分动态地分配存储空间。

多维数组传递给函数时，除第一维外，其他各维都必须说明。例如，将数组 m 定义成：

int m[4][3][6][5];

那么接收 m 的函数应写成：

func1 (d)

int d[][3][6][5];

当然，如果愿意，也可加上第一维的说明。

3.2.4 数组的初始化

1. 数组初始化

C语言允许在说明时对全局数组和静态局部数组初始化，但不能对非静态局部数组初始化。与其他变量相似，数组初始化的一般形式如下：

```
type-specifier array_name[size1]...[sizenn]={value-list};
```

数值表是一个由逗号分隔的常量表。这些常量的类型与类型说明相容，第一个常量存入数组的第一个单元，第二个常量存入第二个单元，等等，注意在括号“}”后要加上分号。

下列中一个10元素整型数组被初始化装入数字1到10：

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

这意味着 $i[0]$ 的值为 1，而 $i[9]$ 的值为 10。

存放字符串的字符数组的初始化可采用如下简化形式：

```
char array_name[size] = "string";
```

例如，以下代码段将 str 初始化为“hello”。

```
char str[6] = "hello";
```

上面代码产生和下面代码相同的结果：

```
char str[6]={‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
```

因为C语言中的字符串都以空(NULL)字符为终结，故要确认定义的数组足够长以存放空字符。这就是为什么 hello 只有 5 个字符，而 str 要有 6 个字符长的原因。使用字符串常量时，编译程序自动地在末尾加上空字符。

多维数组初始化的方法与一维数组相同，例如，下式将 sqrs 初始化为从 1 到 10 及它们各自的平方数。

```
int sqrs[10][2]={
    1 ,1 ,
    2 ,4 ,
    3 ,9 ,
    4 ,16,
    5 ,25,
    6 ,36,
    7 ,49,
    8 ,64,
    9 ,81,
    10,100
};
```

2. 变长数组的初始化

设想用数组初始化的方法建立一个如下错误信息表：

```
char e1[12] = "read error\n";
char e2[13] = "write error\n";
char e3[18] = "cannot open file\n";
```

可以想象，如果用手工方法去计算每一条信息的字符数以确定数组的长度是很麻烦的。利用变长数组初始化的方法可以使 C 编译器自动地计算数组的长度。变长数组初始化就是使 C 编译程序自动建立一个不指明长度的足够大的数组以存放初始化数据。使用这种方法，以上信息表变为：

```
char e1[] = "read error\n";
char e2[] = "write error\n";
char e3[] = "cannot open file\n";
```

给定上面的初始化，下面的语句

```
printf("%s has length %d\n", e2, sizeof(e2));
```

将打印出：

```
write error
has length 13
```

除了减少麻烦外，应用变长数组初始化使程序员可以修改任何信息，而不必担心随时可能发生的计算错误。

变长数组初始化的方法不仅仅限于一维数组。但在对多维数组初始化时，必须指明除了第一维以外其他各维的长度，以使编译程序能够正确地检索数组。其方法与数组形式参数的说明类似。这样就可以建立变长表，而编译程序自动地为它们分配存储空间。例如，下面用变长数组初始化的方法定义数组 sqrs：

```
int sqrs[ ][2]={
    1 ,1 ,
    2 ,4 ,
    3 ,9 ,
    4 ,16 ,
    5 ,25 ,
    6 ,36 ,
    7 ,49 ,
    8 ,64 ,
    9 ,81 ,
    10 ,100
};
```

相对定长数组的初始化而言，这种说明的优点在于可以在不改变数组各维长度的情况下，随时增加或缩短表的长度。

3.3 指针

指针是 C 语言的精华部分，利用指针，能够很好地利用内存资源，使其发挥最大的效率。

有了指针技术，可以描述复杂的数据结构，对字符串的处理可以更灵活，对数组的处理更方便，使程序的书写简洁，高效。但由于指针对初学者来说，比较难于理解和掌握，需要一定的计算机硬件的知识做基础，这就需要多做多练，多上机动手，才能在实践中尽快掌握。

3.3.1 指针与指针变量

过去，我们在编程中定义或说明变量，编译系统就为已定义的变量分配相应的内存单元，也就是说，每个变量在内存会有固定的位置，有具体的地址。由于变量的数据类型不同，它所占的内存单元数也不相同。若在程序中定义：

```
int a=1, b=2;
float x=3.4, y = 4 . 5 ;
double m=3.124;
char ch1='a', ch2='b' ;
```

变量在内存中按照数据类型的不同，占内存的大小也不同，都有具体的内存单元地址，如变量 a 在内存的地址是 2000，占据两个字节后，变量 b 的内存地址就为 2002，变量 m 的内存地址为 2012 等。对内存中变量的访问，过去用 `scanf ("%d%d%f", &a, &b, &x)` 表示将数据输入变量的地址所指示的内存单元。那么，访问变量，首先应找到其在内存的地址，或者说，一个地址唯一指向一个内存变量，我们称这个地址为变量的指针。如果将变量的地址保存在内存的特定区域，用变量来存放这些地址，这样的变量就是指针变量，通过指针对所指向变量的访问，也就是一种对变量的“间接访问”。

设一组指针变量 pa、pb、px、py、pm、pch1、pch2，分别指向上述的变量 a、b、x、y、m、ch1、ch2，指针变量也同样被存放在内存中。

3.3.2 指针变量的定义与引用

1. 指针变量的定义

在 C 程序中，存放地址的指针变量需专门定义；

```
int *ptr1;
float *ptr2;
char *ptr3;
```

表示定义了 3 个指针变量 ptr1、ptr2、ptr3。ptr1 指向一个整型变量，ptr2 指向一个实型变量，ptr3 指向一个字符型变量，换句话说，ptr1、ptr2、ptr3 可以分别存放整型变量的地址、实型变量的地址、字符型变量的地址。

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

```
int *ptr1, m= 3;
float *ptr2, f=4.5;
char *ptr3, ch='a' ;
ptr1 = &m ;
```

```
ptr2 = &f ;
ptr3 = &ch ;
```

上述赋值语句 `ptr1 = &m` 表示将变量 `m` 的地址赋给指针变量 `ptr1`, 此时 `ptr1` 就指向 `m`。三条赋值语句产生的效果是 `ptr1` 指向 `m`; `ptr2` 指向 `f`; `ptr3` 指向 `ch`。

需要说明的是, 指针变量可以指向任何类型的变量, 当定义指针变量时, 指针变量的值是随机的, 不能确定它具体的指向, 必须为其赋值, 才有意义。

2. 指针变量的引用

利用指针变量, 是提供对变量的一种间接访问形式。对指针变量的引用形式为:

`*指针变量`

其含义是指针变量所指向的值。

[例 3-8] 用指针变量进行输入和输出。

```
main( )
{
    int *p, m;
    printf("input a number:\n");
    scanf("%d", &m) ;
    p=&m;                                //指针 p 指向变量 m
    printf("%d", *p);                     //p 是对指针所指的变量的引用形式, 与此
m 意义相同
}
```

运行程序结果如下:

```
3
3
```

上述程序可修改为:

```
main( )
{
    int *p, m;
    p = &m ;
    scanf ( " %d " , p ) ;                // p 是变量 m 的地址, 可以替换&m
    printf("%d", m);
}
```

运行效果完全相同。请读者思考一下如果将程序修改为如下形式:

```
main( )
{
    int *p, m;
    scan f("%d", p) ;
    p = &m ;
```

```

    printf("%d", m);
}

```

上述代码会产生什么样的结果呢？事实上，若定义了变量以及指向该变量的指针为 int a,*p;若 p=&a; 则称 p 指向变量 a，或者说 p 具有了变量 a 的地址。在以后的程序处理中，凡是可能写&a 的地方，就可以替换成指针的表示 p，a 就可以替换成*p。

3.3.3 指针运算符与指针表达式

1. 指针运算符与指针表达式

在 C 中有两个关于指针的运算符：

- &运算符：取地址运算符，&m 即是变量 m 的地址。
- *运算符：指针运算符，*ptr 表示其所指向的变量。

[例 3-9] 整数排序

分析：本例当然可以以前面的普通整形变量来求解，不过在这个例子中，我们使用指针指向这两个变量，并通过对指针指向的值的操作来完成计算。

具体代码如下：

```

main()
{
    int *p1, *p2, a, b, t;
    scanf("%d %d", &a, &b); // 定义指针变量与整型变量
    p1 = &a; // 使指针变量指向整型变量
    p2 = &b;
    if (*p1 < *p2)
    {
        t = *p1; // 交换指针变量指向的整型变量
        *p1 = *p2;
        *p2 = t;
    }
    printf("%d %d \n", a, b);
}

```

在程序中，当执行赋值操作 p1=&a 和 p2=&b 后，指针指向了变量 a 与 b，这时引用指针 *p1 与 *p2，就代表了变量 a 与 b。

运行程序结果如下：

输入：

3, 4

输出：

4, 3

[例 3-10] 使用指针进行整数排序

分析：上面的例子中，指针的值一直是不变的，变化的只是指针指向的内存中的值。而本例要求的是内存中的值不变，而只改变指针。请读者仔细阅读下面的代码，并体会两种方法的差别。

```
main()
{
    int *p1, *p2, a, b, *t;
    scanf ("%d ,%d" ,&a, &b);
    p1 =&a;
    p2 =&b;
    if( *p1< *p2)
    {
        //指针交换指向
        t=p1;
        p1=p2;
        p2=t;
    }
    printf ("%d ,%d \n", *p1, *p2);
}
```

程序的运行结果完全相同，但程序在运行过程中，实际存放在内存中的数据没有移动，而是将指向该变量的指针交换了指向。

当指针交换指向后，p1 和 p2 由原来指向的变量 a 和 b 改变为指向变量 b 和 a，这样一来，*p1 就表示变量 b，而*p2 就表示变量 a。在上述程序中，无论在何时，只要指针与所指向的变量满足 p=&a；我们就可以对变量 a 以指针的形式来表示。此时 p 等效于&a，*p 等效于变量 a。

2. 指针变量作函数的参数

函数的参数可以是我们在前面介绍过的简单数据类型，也可以是指针类型。使用指针类型做函数的参数，实际向函数传递的是变量的地址。由于子程序中获得了所传递变量的地址，在该地址空间的数据当子程序调用结束后被物理地保留下来。

[例 3-11] 子程序方法进行排序

分析：将指针传到子函数中以后，在子函数中更改指针所指向的地址的值，以此来完成需要的操作。

源程序如下：

```
main()
{
    void change(int *pt1, int *pt2);           //函数声明
    int *p1,*p2,a,b;
    scanf("%d ,%d" ,&a, &b);
```

```

p1 =&a;
p2 =&b;
change(p1, p2); //子程序调用
printf("%d , %d \n", *p1, *p2);
}

void change(int *pt1, int *pt2)
{
    int t;
    if (*pt1<*pt2)                                //子程序实现将两数值调整为由大到小
    {
        t=*pt1; *pt1=*pt2; *pt2=t;
    }
    return ;
}

```

由于在调用子程序时，实际参数是指针变量，形式参数也是指针变量，实参与形参相结合，传值调用将指针变量传递给形式参数 pt1 和 pt2。但此时传值传递的是变量地址，使得在子程序中 pt1 和 pt2 具有了 p1 和 p2 的值，指向了与调用程序相同的内存变量，并对其在内存存放的数据进行了交换。

思考下面的程序，是否也能达到相同的效果呢？

```

main( )
{
    void chang();
    int *p1, *p2, a, b, *t;
    scanf ("%d , %d " , &a , &b ) ;
    p1 = &a ;
    p2 = &b ;
    change(p1, p2);
    printf("%d , %d\n", *p1, *p2) ;
}

void change(int *pt1, int *pt2)
{
    int *t;
    if (*pt1<*pt2)
    {
        t=pt1; pt1=pt2; pt2= t ;
    }
    return ;
}

```

程序运行结束，并未达到预期的结果，输出与输入完全相同。原因是对于子程序来说，函数内部进行指针相互交换指向，而在内存存放的数据并未移动，子程序调用结束后，main()

函数中 p1 和 p2 保持原指向，结果与输入相同。

3.3.4 指针与数组

变量存放于内存中时是有地址的，数组在内存存放时也同样具有地址。对数组来说，数组名的值就是数组在内存安放的首地址。指针变量是用于存放变量的地址，可以指向变量，当然也可存放数组的首址或数组元素的地址，这就是说，指针变量可以指向数组或数组元素，对数组而言，数组和数组元素的引用，也同样可以使用指针变量。下面就分别介绍指针与不同类型的数组。

1. 指针与一维数组

假设定义一个一维数组，该数组在内存会有系统分配的一个存储空间，其数组的名字就是数组在内存的首地址。若再定义一个指针变量，并将数组的首址传给指针变量，则该指针就指向了这个一维数组。我们说数组名是数组的首地址，也就是数组的指针。而定义的指针变量就是指向该数组的指针变量。对一维数组的引用，既可以用传统的数组元素的下标法，也可使用指针的表示方法。

例如，如果已经使用如下语句定义了变量：

```
int a[10], *ptr; // 定义数组与指针变量
```

那么，如果我们要做赋值操作，我们可以使用语句：

```
ptr=a;
```

或

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

则 ptr 就得到了数组的首址。其中，a 是数组的首地址，&a[0] 是数组元素 a[0] 的地址，由于 a[0] 的地址就是数组的首地址，所以，两条赋值操作效果完全相同。指针变量 ptr 就是指向数组 a 的指针变量。

若 ptr 指向了一维数组，现在看一下 C 规定指针对数组的表示方法：

(1) ptr+n 与 a+n 表示数组元素 a[n] 的地址，即&a[n]。对整个 a 数组来说，共有 10 个元素，n 的取值为 0~9，则数组元素的地址就可以表示为 ptr+0~ptr+9 或 a+0~a+9，与 &a[0]~&a[9] 保持一致。

(2) *(ptr+n) 和*(a+n) 表示为数组的元素即等效于 a[n]。

(3) 指向数组的指针变量也可用数组的下标形式表示为 ptr[n]，其效果相当于*(ptr+n)。

[例 3-12] 下标法输入输出数组各元素

分析：本例的要求是分别对数组的每一个元素进行赋值。

程序源代码如下：

```
# include <stdio.h>
main()
{
    int n, a[10], *ptr=a;
    for(n=0;n<=9;n++) //输入
```

```

        scanf("%d", &a[n]);
        printf("1-----output! \n");           //输出
        for(n=0;n<=9;n++)
            printf("%4d", a[n]);
        printf("\n");
    }

```

运行程序结果如下：

```

1 2 3 4 5 6 7 8 9 0
1-----output!
1   2   3   4   5   6   7   8   9   0

```

[例 3-13] 用指针法输入输出数组各元素

分析：本例的要求是使用指针表示的地址，在与数组中的元素的序号相加，得到每个元素的地址，并对之进行操作。

源代码如下：

```

# include<stdio.h>
main( )
{
    int n, a[10], *ptr=a;           //定义时对指针变量初始化
    for(n=0;n<=9;n++)
        scanf("%d", ptr+n);
    printf("2-----output!\n");
    for(n=0;n<=9;n++)
        printf("%4d", *(ptr+n));
    printf("\n");
}

```

运行程序结果如下：

```

1 2 3 4 5 6 7 8 9 0
2-----output!
1   2   3   4   5   6   7   8   9   0

```

[例 3-14] 采用数组名表示的地址法输入输出数组各元素

分析：本例的要求其实是将数组名看作一个指针，并对之进行加减操作，以更改数组中的每一个元素。

源代码如下：

```

main( )
{
    int n, a[10], *ptr=a;           //定义时对指针变量初始化
    for(n = 0;n <= 9 ;n++)
        scanf("%d" , a + n ) ;
}

```

```

    printf("3-----output! \n");
    for ( n = 0 ; n <= 9 ; n++)
        printf( " %4d " , *(a+n) ) ;
    printf("\n") ;
}

```

运行程序结果如下：

```

1 2 3 4 5 6 7 8 9 0
3-----output!
1   2   3   4   5   6   7   8   9   0

```

[例 3-15] 用指针表示的下标法输入输出数组各元素

分析：该例的要求其实是将指针看成数组名，并按照数组脚表的方式，来对数组中的每一个元素进行操作。

源代码如下：

```

main( )
{
    int n, a[10], *ptr=a;
    for( n = 0 ; n <= 9 ; n++ )
        scanf ( "%d" , &ptr[n] ) ;
    printf("4-----output! \n");
    for( n = 0 ; n <= 9 ; n++ )
        printf(" %4d " , ptr[n] ) ;
    printf(" \n " ) ;
}

```

运行程序结果如下：

```

1 2 3 4 5 6 7 8 9 0
4-----output!
1   2   3   4   5   6   7   8   9   0

```

[例 3-16] 利用指针法输入输出数组各元素。

分析：本例其实是要求使用指针本身的运算，如加减操作等，来得到数组的各个元素，并对之进行操作。

源代码如下：

```

main( )
{
    int n, a[10], *ptr=a;
    for( n = 0 ; n <= 9 ; n++ )
        scanf( "%d" , ptr++ ) ;
    printf("5-----output! \n");
    ptr = a ;                                //指针变量重新指向数组首址
}

```

```

for(n = 0 ; n <= 9 ; n++)
    printf( " %4d ",*ptr++ );
printf( " \n " );
}

```

运行程序结果如下：

```

1 2 3 4 5 6 7 8 9 0
5-----output!
1 2 3 4 5 6 7 8 9 0

```

在程序中要注意`*ptr++`所表示的含义。`*ptr` 表示指针所指向的变量；`ptr++`表示指针所指向的变量地址加 1 个变量所占字节数，具体地说，若指向整型变量，则指针值加 2，若指向实型，则加 4，依此类推。而 `printf("%4d" , *ptr++)` 中，`* ptr++` 所起作用为先输出指针指向的变量的值，然后指针变量加 1。

指针变量的值在循环结束后，指向数组的尾部的后面。假设元素 `a[9]` 的地址为 1000，整型占 2 字节，则 `ptr` 的值就为 1002。请思考下面的程序段：

```

main()
{
    int n,a[10],*ptr=a;
    for( n = 0 ; n <= 9 ; n++)
        scanf( " %d " , ptr++);
    printf("4-----output! \n");
    for( n = 0 ; n <= 9 ; n++)
        printf( " %4d " , *ptr++ );
    printf( "\n" );
}

```

程序与例 41 相比，只少了赋值语句 `ptr= a;` 程序的运行结果还相同吗？

2. 指针与二维数组

定义一个二维数组的语句的例子如下：

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

该语句表示二维数组有三行四列共 12 个元素，在内存中按行存放，其中 `a` 是二维数组的首地址，`&a[0][0]`既可以看作数组 0 行 0 列的首地址，同样还可以看作是二维数组的首地址，`a[0]`是第 0 行的首地址，当然也是数组的首地址。同理 `a[n]` 就是第 `n` 行的首址；`&a[n][m]` 就是数组元素 `a[n][m]` 的地址。

既然二维数组每行的首地址都可以用 `a[n]` 来表示，我们就可以把二维数组看成是由 `n` 行一维数组构成，将每行的首地址传递给指针变量，行中的其余元素均可以由指针来表示。

我们定义的二维数组其元素类型为整型，每个元素在内存占两个字节，若假定二维数组从 1000 单元开始存放，则以按行存放的原则，数组元素在内存的存放地址为 1000~1022。用地址法来表示数组各元素的地址。对元素 `a[1][2]`，`&a[1][2]` 是其地址，`a[1]+2` 也是其地址。分析 `a[1]+1` 与 `a[1]+2` 的地址关系，它们地址的差并非整数 1，而是一个数组元素的所占位置 2，原因是每个数组元素占两个字节。对 0 行首地址与 1 行首地址 `a` 与 `a+1` 来说，地

址的差同样也并非整数 1，是一行的大小，4 个元素占的字节数为 8。由于数组元素在内存的连续存放。给指向整型变量的指针传递数组的首地址，则该指针指向二维数组。

int *ptr, a[3][4]; 若赋值：ptr=a; 则用 ptr++ 就能访问数组的各元素。

[例 3-17] 用地址法输入输出二维数组各元素

分析：该例的要求其实是使用 a[i] 来表示已经被定义的二维数组 a[3][4] 的某一行第 0 个的元素的地址，并使用它来找到该行中的所有的元素。

例程序如下：

```
#include <stdio.h>
main( )
{
    int a[3][4];
    int i, j;
    for( i = 0 ; i < 3 ; i++ )
        for( j = 0 ; j < 4 ; j++ )
            scanf( "%d" , a[i]+j ) ; // 地址法
    for( i = 0 ; i < 3 ; i++ )
    {
        for( j = 0 ; j < 4 ; j++ )
            printf("%d",*(a[i]+j)); // *(a[i]+j) 是地址法所表示的数组
    }
}
```

元素

运行程序结果如下：

```
1 2 3 4 5 6 7 8 9 10 11 12
1   2   3   4
5   6   7   8
9  10  11  12
```

[例 3-18] 用指针法输入输出二维数组各元素

分析：本例的要求其实是希望读者了解二维数组中，a[i] 等效于一个指针，使用指针的运算也可以得到数组中的每一个元素的位置和值。

源代码如下：

```
#include<stdio.h>
main( )
{
    int a[3][4], *ptr;
    int i, j;
    ptr=a[0] ;
```

```

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

```

运行程序结果如下：

输入： 1 2 3 4 5 6 7 8 9 10 11 12

输出：

```

1   2   3   4
5   6   7   8
9   10  11  12

```

对指针法而言，程序可以把二维数组看作展开的一维数组：

```

main( )
{
    int a[3][4],*ptr;
    int i,j;
    ptr= a [ 0 ] ;
    for( i = 0 ; i < 3 ; i++)
        for( j = 0 ; j < 4 ; j++)
            scanf( "%d" , ptr++) ;
                                //指针的表示方法
ptr=a[0] ;
    for( i = 0 ; i < 12 ;i++)
        printf( "%4d" , *ptr++);
    printf("\n");
}

```

运行程序结果如下：

输入： 1 2 3 4 5 6 7 8 9 10 11 12

输出：

```

1   2   3   4
5   6   7   8
9   10  11  12

```

3. 数组指针作函数的参数

学习了指向一维和二维数组指针变量的定义和正确引用后，我们现在介绍用指针变量作

函数的参数。

[例 3-19] 子程序求解一维数组中的最大值元素

分析：首先假设一维数组中下标为 0 的元素是最大值，用指针变量指向该元素。后续元素与该元素一一比较，若找到更大的元素，就互相替换。子程序的形式参数为一维数组，实际参数是指向一维数组的指针。

源代码如下：

```
main (void) {
    int sub_max(int b[], int i); //函数声明
    int n, a[10], *ptr=a; //定义变量，并使指针指向数组
    int max;
    for( n = 0 ; n <= 9 ; n++ ) //输入数据
        scanf ( "%d" , &a[n] );
    max=sub_max(ptr, 10); //函数调用，其实参是指针
    printf( "max=%d\n" , max );
}

int sub_max(int b[], int i) //函数定义，其形参为数组
{
    int temp, j;
    temp= b[0];
    for( j = 1 ; j <= i-1 ; j++ )
        if(temp<b[j]) temp=b[j];
    return temp;
}
```

程序的 main() 函数部分，定义数组 a 共有 10 个元素，由于将其首地址传给了 ptr，则指针变量 ptr 就指向了数组，调用子程序，再将此地址传递给子程序的形式参数 b，这样一来，b 数组在内存与 a 数组具有相同地址，即在内存完全重合。在子程序中对数组 b 的操作，与操作数组 a 意义相同。

main() 函数完成数据的输入，调用子程序并输出运行结果。sub_max() 函数完成对数组元素找最大的过程。在子程序内数组元素的表示采用下标法。运行程序结果如下：

```
输入: 1 3 5 7 9 2 4 6 8 0
输出: max=9
```

[例 3-20] 指针变量作子程序的形式参数

分析：经过数组一节中的若干例子，相信读者已经知道，一维数组其实和一维指针是等效的。所以在本例中，只需要简单的将形参改变一下就可以了。

源程序如下：

```
#include <stdio.h>
```

```

main()
{
    int sub_max(int *b, int i);
    int n, a[10], *ptr=a;
    int max;
    for( n = 0 ; n <= 9 ; n++)
        scanf ( "%d" , &a[n]) ;
    max = sub_max (ptr, 10) ;
    printf("max=%d\n", max) ;
}

int sub_max(int *b, int i) //形式参数为指针变量
{
    int temp, j;
    temp=b[0] ; //数组元素指针的下标法表示
    for( j = 1 ; j <= i - 1 ; j++)
        if(temp<b[j]) temp=b[j];
    return temp;
}

```

在子程序中，形式参数是指针，调用程序的实际参数 ptr 为指向一维数组 a 的指针，虚实结合，子程序的形式参数 b 得到 ptr 的值，指向了内存的一维数组。数组元素采用下标法表示，即一维数组的头指针为 b，数组元素可以用 b[j] 表示。

运行程序结果如下：

```

输入: 1 3 5 7 9 2 4 6 8 0
输出: max=9

```

[例 3-21] 用指针表示数组元素

分析：本例只是将子函数中的数组写法改成指针形式，程序执行的结果并不会改变。读者应该注意这种在子程序中忽略外部究竟是数组还是指针的编程方式，使得程序的封装性变得更好。

源程序如下：

```

#include <stdio.h>
main()
{
    int sub_max(int *b, int i);
    int n, a[10], *ptr=a;
    int max;
    for( n = 0 ; n <= 9 ; n++)
        scanf ( "%d" , &a[n]) ;
    max = sub_max (ptr, 10) ;
    printf("max=%d\n", max) ;
}

```

```

}

int sub_max(int *b, int i) //形式参数为指针变量
{
    int temp, j;
    temp=*b++;
    for(j=1;j<=i-1;j++)
        if(temp<*b) temp=*b++;
    return temp;
}

```

在程序中，赋值语句 `temp=*b++;` 可以分解为： `temp=*b;` `b++;` 两句，先作 `temp=*b;` 后作 `b++;` 程序的运行结果与上述完全相同。

如果对上面的程序作修改，在子程序中不仅找最大元素，同时还要将元素的下标记录下来，那么源程序如下：

```

#include <stdio.h>

main()
{
    int *max(int *a, int n) //函数声明
    int n, a[10], *s, i;
    for(i=0;i<10;i++) //输入数据
        scanf("%d", a+i);
    s = max(a, 10); //函数调用
    printf("max=%d, index=%d\n", *s, s-a) ;
}

int *max(int *a, int n) //定义返回指针的函数
{
    int *p, *t; // p 用于跟踪数组, t 用于记录最大
    值元素的地址
    for(p=a, t=a;p-a<n;p++)
        if(*p>*t) t=p;
    return t;
}

```

在 `max()` 函数中，用 `p-a<n` 来控制循环结束，`a` 是数组首地址，`p` 用于跟踪数组元素的地址，`p-a` 正好是所跟踪元素与数组首地址的距离，即距离数组首地址的元素个数，所以在 `main()` 中，最大元素的下标就是该元素的地址与数组头的差，即 `s-a`。运行程序结果如下：

输入： 1 3 5 7 9 2 4 6 8 0

输出： max=9, index=4

[例 3-22] 一维数组的冒泡排序

编写 3 个函数用于输入数据、数据排序、数据输出。

分析：在本例中，要求使用一种排序算法，将一组 `n` 个无序的数整理成由小到大的顺序，

将其放入一维数组 $a[0]$ 、 $a[1] \dots a[n-1]$ 。排序算法有若干种，冒泡排序是一种实现起来简单但是运算效率并不算高的一种。它的具体算法如下：

① 相邻的数组元素依次进行两两比较，即 $a[0]$ 与 $a[1]$ 比、 $a[1]$ 与 $a[2]$ 比. . . $a[n-2]$ 与 $a[n-1]$ 比，通过交换保证数组的相邻两个元素前者小，后者大。此次完全的两两比较，能免实现 $a[n-1]$ 成为数组中最大。

② 余下 $n-1$ 个元素，按照上述原则进行完全两两比较，使 $a[n-2]$ 成为余下 $n-1$ 个元素中最大。

③ 进行共计 $n-1$ 趟完全的两两比较，使全部数据整理有序。

下面给出一趟排序的处理过程：

原始数据：3 8 2 5

第一次相邻元素比：3 8 2 5

第二次相邻元素比：3 2 8 5

第三次相邻元素比：3 2 5 8

4 个元素进行 3 次两两比较，得到一个最大元素。若相邻元素表示为 $a[j]$ 和 $a[j+1]$ ，用指针变量 P 指向数组，则相邻元素表示为 $*(\text{P}+j)$ 和 $*(\text{P}+j+1)$ 。

算法的核心部分定义为一个函数，取名为 sort ，程序的具体实现如下：

```
void sort(int *ptr, int n) //冒泡排序, 形参 ptr 是指针变量
{
    int i, j, t;
    for( i = 0 ; i < n - 1 ; i++ )
        for( j = 0 ; j < n - 1 - i ; j++ )
            if ( *(ptr+j) > *(ptr+j+1) ) //相临两个元素进行比较
            {
                t = * (ptr+ j) ;
                *(ptr+ j) = * (ptr +j+1) ;
                *(ptr+ j + 1) = t ;
            }
}
```

另外，程序还有主函数、输入函数和输出函数，由于并不复杂，所以为篇幅起见，这里就不一一介绍。整个程序的源代码如下：

```
#include<stdio.h>
#define N 10
main( )
{
    void input(int arr[], int n); //函数声明
    void sort(int *ptr, int n);
    void output(int arr[], int n);
    int a[N],*p; //定义一维数组和指针变量
    input(a,N); //数据输入函数调用, 实参 a 是数组
```

```

名
    input(a, N) ;                                //数据输入函数调用, 实参 a 是数组
名
    p = a ;                                     //指针变量指向数组的首地址
    sort(p, N) ;                                //排序, 实参 p 是指针变量
    output(p, N) ;                            //输出, 实参 p 是指针变量
}
void input(int arr[], int n)
//无需返回值的输入数据函数定义, 形参 arr 是数组
{
    int i;
    printf("input data:\n");
    for( i = 0 ; i < n ; i++ )                  //采用传统的下标法
        scanf( "%d" , &arr[i] ) ;
}
void sort(int *ptr, int n)                      //冒泡排序, 形参 ptr 是指针变
量
{
    int i, j, t;
    for( i = 0 ; i < n - 1 ; i++ )
        for( j = 0 ; j < n - 1 - i ; j++ )
            if ( *(ptr+j) > *(ptr+j+1) )          //相临两个元素进行比较
            {
                t = * (ptr+ j) ;                   //两个元素进行交换
                *(ptr+ j) = * (ptr +j+1) ;
                *(ptr+ j + 1) = t ;
            }
}
void output(int arr[], int n)                   //数据输出
{
    int *ptr=arr;                                //利用指针指向数组的首地址
    printf("output data:\n");
    for( ; ptr-arr<n ; ptr++ )                  //输出数组的 n 个元素
        printf("%4d",*ptr) ;
    printf("\n");
}

```

运行程序结果如下：

```
输入: 3 5 7 9 3 23 43 2 1 10
```

```
输出: 1 2 3 3 5 7 9 10 23 43
```

由于C程序的函数调用是采用传值调用，即实际参数与形式参数相结合时，实参将值传给形式参数，所以当我们利用函数来处理数组时，如果需要对数组在子程序中修改，只能传递数组的地址，进行传地址的调用，在内存相同的地址区间进行数据的修改。在实际的应用中，如果需要利用子程序对数组进行处理，函数的调用利用指向数组（一维或多维）的指针作参数，无论是实参还是形参共有下面4种情况：

- 数组名数组名
- 数组名指针变量
- 指针变量数组名
- 指针变量指针变量

在函数的调用时，实参与形参的结合要注意所传递的地址具体指向什么对象，是数组的首地址，还是数组元素的地址，这一点很重要。

[例3-23] 二维数组的指针作函数的参数

分析：要计算二维数组的每行的和，我们必须要包含有二维数组的指针，以便将数组中的元素传入。另外，我们需要与行数相对应的一维指针，来将计算出来的每行元素的和返回。

程序的源代码如下：

```
#include <stdio.h>
#define M 3
#define N 4
main()
{
    float a[M][N];
    float score1, score2, score3, **pa=a[0]; //指针变量 p a 指向二维数组
    // score1, score2, score3 分别记录三行的数据相加
    int i, j;
    void fun(float b[][N], float *p1, float *p2, float *p3);
    for( i = 0 ; i < M ; i++)
        for(j=0;j<N; j++) //二维数组的数据输入
            scanf("%f", &a[i][j]);
        fun(pa, &score1, &score2, &score3);
    //函数调用，不仅传递数组首地址，还要传递变量的地址
    printf("%.2f, %.2f, %.2f\n", score1, score2, score3);
}
void fun(float b[][N], float *p1, float *p2, float *p3)
{
    int i, j;
    *p1=*p2=*p3=0;
    for(i=0; i < M ; i++)
        for(j=0; j < N; j++)
            *p1+=a[i][j];
    *p2=score1;
    *p3=score2;
}
```

```

        for ( j = 0 ; j < N ; j++)
    {
        if(i==0) *p1=*(p1+b[i][j]);           //第 0 行的数据相加
        if(i==1) *p2=*(p2+b[i][j]);           //第 1 行的数据相加
        if(i==2) *p3=*(p3+b[i][j]);           //第 2 行的数据相加
    }
}

```

程序中与形式参数 p1、p2 和 p3 相对应的是参数&score1、&score2 和&score3，其实际含义为 p1=&score1 等，即将变量的地址传递给指针变量达到按行相加。运行程序结果如下：

```

输入：1 2 3 4 3 4 5 6 5 6 7 8
输出：10.00, 18.00, 26.00

```

[例 3-24] 二维数组中的最大值及位置。

分析：二维数组在内存中是按行存放，假定二维数组和指针定义如下：

```
int a[3][4], *p = a[0];
```

则指针 p 就是指向二维数组的指针。

从上述存放情况来看，若把二维数组的首地址传递给指针 p，我们只要找到用 p 所表示的一维数组中最大的元素及下标，就可转换为在二维数组中的行列数。

具体程序如下：

```

#include<stdio.h>
main( )
{
    int a[3][4], *ptr, i, j, max, maxi, maxj;
    // max 是数组的最大， maxi 是最大元素所在行， maxj 是最大元素所在列
    int max_arr(int *b, int *p1, int *p2, int *n);           //函数声明
    for( i = 0 ; i < 3 ; i++)
        for( j = 0 ; j < 4 ; j++)
            scanf( "%d" , &a[i][j]) ;
    ptr= a[0] ;                                              //将二维数组的首地址传递给指针
变量
    max_arr(ptr, &max, &maxi, 12) ;
    maxj=maxi%4;                                            //每行有四个元素，求该元素所在列
    maxi=maxi/4 ;                                           //求该元素所在行
    printf("max=%d, maxi=%d, maxj=%d", max, maxi, maxj);
}

int max_arr(int *b, int *p1, int *p2, int *n)
// b 指向二维数组的指针， p1 指向最大值， p2 指向最大值在一维数组中的位置，
// n 是数组的大小
{
    int i;

```

```
*p1=b[0]; *p1=0;
for(i=1;i<n;i++)
    if (b[i]>*p1) {*p1=b[i]; *p2=i;}
}
```

运行程序结果如下：

```
输入:
4 7 8 9
3 7 9 3
1 5 2 6
输出: max=9, maxi=0, maxj=3
```

4. 指针与字符数组

在前面的课程中，我们用过了字符数组，即通过数组名来表示字符串，数组名就是数组的首地址，是字符串的起始地址。下面的例子用于简单字符串的输入和输出。

```
#include <stdio.h>
main( )
{
    char str[20];
    gets(str) ;
    printf("%s\n", str) ;
}
```

```
输入: good morning!
输出: good morning!
```

现在，我们将字符数组的名赋予一个指向字符类型的指针变量，让字符类型指针指向字符串在内存的首地址，对字符串的表示就可以用指针实现。其定义的方法为：char str[20]，* P = str；这样一来，字符串 str 就可以用指针变量 p 来表示了。

```
#include <stdio.h>
main( )
{
    char str[20],*p=str ;                                // p=str 则表示将字符数组的首地址传递给指针变量 p
    gets(str) ;
    printf("%s\n", p) ;
}
```

运行结果如下：

```
输入: good morning!
输出: good morning!
```

需要说明的是，字符数组与字符串是有区别的，字符串是字符数组的一种特殊形式，存储时以“\0”结束，所以，存放字符串的字符数组其长度应比字符串大 1。对于存放字符的字符数组，若未加“\0”结束标志，只能按逐个字符输入输出。

[例 3-25] 字符串输出比较

分析：未加结束符时，程序输出将会不稳定，也就是说可能出现不可预见的输出结果。读者可以自己在 KEIL51 的 IDE 开发环境下仿真一下。

具体源程序如下：

```
#include<stdio.h>
main( )
{
    char str[10], *p=str;
    int i;
    scanf( "%s" ,str);                                //输入的字符串长度超过 10
    for( i=0;i<10;i++)
        printf("%c",*p++);
    printf("\n");
    p=str;
    printf("%s",p);                                    //字符数组无'\0'标志，输出出错
    puts(str);                                       //字符数组无'\0'标志，输出出错
}
```

对上述程序中字符数组以字符串形式输出，若无“\0”标志，则找不到结束标志，输出出错。

[例 3-26] 两个字符串的复制

分析：字符串的复制可以直接通过指针的操作或者库函数的操作来完成。本例中使用的方式前一种方式。需要注意的是，若将串 1 复制到串 2，一定要保证串 2 的长度大于或等于串 1。

具体源代码如下：

```
# include<stdio.h>
main( )
{
    char str1[30], str2[20], *ptr1=str1, *ptr2=str2;
    printf("input str1:");
    gets(str1, 30);                                 //输入 str1
    printf("input str2:");
    gets(str2, 20);                                 //输入 str2
    printf("str1 - - - - - - - - str2\n");
    printf("%s.....%s\n",ptr1,ptr2) ;
    while(*ptr2) *ptr1++=*ptr2++;
    *ptr1='\'0';                                     //字符串复制
    printf("str1 - - - - - - - - str2\n");
    printf("%s . . . . . %s\n",str1,str2) ;
}
```

在程序的说明部分，定义的字符指针指向字符串。语句 `while(*ptr2) *ptr1++=*ptr2++;` 先测试表达式的值，若指针指向的字符是“\0”，该字符的 ASCII 码值为 0，表达式的值为假，循环结束，表达式的值非零，则执行循环`*ptr1++=*ptr2++`。语句`*ptr1++`按照运算优先级别，先算`*ptr1`，再算`ptr1++`。

运行程序结果如下：

```
输入:
input str1: I love China!
input str2: I love Chengdu!
输出:
str1-----str2
I love China! ..... I love Chengdu!
str1 ----- str2
I love Chengdu! ..... I love Chengdu!
```

现在，我们修改程序中语句 `printf("%s.....%s\n", str1, str2);` 为 `printf("%s.....%s\n", ptr1, ptr2);` 会出现什么结果呢？请思考。

[例 3-27] 两个字符串的合并

分析：本例完成的其实是库函数 `strcat` 的功能。只需要找出前一个字符串的串尾，在将第二个字符串复制到其后即可。

具体代码如下：

```
# include<stdio.h>
main( )
{
    char str1[50], str2[20], *ptr1=str1, *ptr2=str2;
    printf("input str1:");
    gets(str1, 20) ;
    printf("input str2:");
    gets(str2, 20) ;
    printf("str1-----str2\n");
    printf("%s.....%s\n", ptr1, ptr2) ;
    while(*ptr1) ptr1++; //移动指针到串尾
    while(*ptr2) *ptr1++=*ptr2++; //串连接
    *ptr1 = '\0'; //写入串的结束标志
    ptr1=str1; ptr2=str2;
    printf("str1----- str2\n");
    printf("%s.....%s \n ", ptr1, ptr2) ;
}
```

运行程序结果如下：

```
输入:
input str1: I love China!
```

```

input str2: I love Chengdu!
输出:
str1-----str2
I love China! ..... I love Chengdu!
str1-----str2
I love China! I love Chengdu! ..... I love Chengdu!.

```

需要注意的是，串复制时，串 1 的长度应大于等于串 2；串连接时，串 1 的长度应大于等于串 1 与串 2 的长度之和。

3.3.5 指针的地址分配

可以定义指针变量指向任何类型的变量。在上述的处理过程中，指针变量指向的变量通过传递变量的地址来实现。指针变量的取值是内存的地址，这个地址应当是安全的，不可以是随意的，否则，写入内存单元的值将会使得已存放的数据或程序丢失。如果程序达到一定的规模，那么就应使用编译系统提供的标准函数来实现地址分配。

ANSI 标准建议设置了两个最常用的动态分配内存的函数 `malloc()` 和 `free()`，并包含在 `stdlib.h` 文件中，KEIL C 亦是如此，但有些 C 编译却使用 `malloc.h` 文件包含。使用时请参考具体的 C 编译版本。

我们这里所指的动态内存分配其含义是指：当定义指针变量时，其变量的取值是随机的，可能指向内存的任一单元。若指针的指向是不安全的内存地址，在该地址空间上的数据交换就会产生意想不到的效果。为此，在程序的执行过程中，要保证指针操作的安全性，就要为指针变量分配一个安全地址。在程序执行时为指针变量所做的地址分配就称之为动态内存分配。

当无需指针变量操作时，可以将其所分配的内存归还系统，此过程我们称之为内存单元的释放。

`malloc()` 用以向编译系统申请分配内存；`free()` 用以在使用完毕释放掉所占内存。

[例 3-28] 两个字符串的交换

分析：本例需要用 `malloc` 动态内存分配指令为 2 个字符串分配内存空间，然后再为一个中间变量分配同样大小的内存，以进行交换的操作。不过请读者注意，在 PC 机上调用 `malloc` 之前不需要初始化内存池，因为操作系统掌管着所有的内存资源，编译器不需要了解这些，但是对于 C51 又不同了，由于没有操作系统的存在，需要被分配为动态内存的区间程序并不清楚，所以首先需要调用 `init_mempool` 来初始化内存池。内存池的起始地址和大小一般与硬件和特殊需求有关。在本例中，我们将内存池初始化为起始地址为 0x1000，大小为 0x500 的一块外部 RAM。

具体代码如下：

```

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
main( )

```

```

{
    char *ptr1,*ptr2,*temp;
    init_mempool (0x1000, 0x500);           //初始化内存池
    ptr1=malloc(30);                      //动态为指针变量分配长度为 30 字节的存储空
间
    ptr2=malloc(30) ;
    temp=malloc(30) ;
    printf("input str1:");
    gets(ptr1, 30);                     //输入字符串
    printf("input str2:");
    gets(ptr2, 30) ;
    printf("str1-----str2\n") ;
    printf("%s.....%s\n",ptr1,ptr2) ;
    strcpy(temp,ptr1);                  //串复制
    strcpy(ptr1,ptr2) ;
    strcpy(ptr2,temp) ;
    printf("str1-----str2\n") ;
    printf("%s.....%s\n",ptr1,ptr2) ;
    free(ptr1) ;
    free(ptr2) ;
}

```

为指针变量分配的存储空间长度取决于存放字符的多少。在上述的程序中，两个串的交换可以通过标准函数 strcpy() 来完成，也可以通过串指针交换指向完成，用“temp=ptr1; ptr1=ptr2; ptr2=temp;”3条赋值语句实现。但是，利用指针交换指向，其物理意义与串通过函数进行的复制完全不同。前者是存放串地址的指针变量数据交换，后者是串在内存物理空间的数据交换。指针变量用完后，将指针变量所占的存储空间释放。

运行程序结果如下：

```

输入:
input str1: China
input str2: Chengdu
输出:
str1-----str2
China.....Chengdu
str1-----str2
Chengdu.....China

```

3.3.6 指针数组

前面介绍了指向不同类型变量的指针的定义和使用，我们可以让指针指向某类变量，并替代该变量在程序中使用；我们也可以让指针指向一维、二维数组或字符数组，来替代这些

数组在程序中使用，给我们在编程时带来许多方便。

下面我们定义一种特殊的数组，这类数组存放的全部是指针，分别用于指向某类的变量，以替代这些变量在程序中的使用，增加灵活性。指针数组定义形式：

类型标识 *数组名[数组长度]

例如：char *str[4]；

由于[]比*优先权高，所以首先是数组形式 str[4]，然后才是与“*”的结合。这样一来指针数组包含 4 个指针 str[0]、str[1]、str[2]、str[3]，各自指向字符类型的变量。例如：int *ptr[5]；

该指针数组包含 5 个指针 ptr[0]、ptr[1]、ptr[2]、ptr[3]、ptr[4]，各自指向整型类型的变量。

[例 3-29] 指针数组的应用

分析：在本例中，我们希望读者对这几种操作方式进行一个对比，并参考前面的数组和指针的对比，理解指针数组的含义。

具体的源代码如下：

```
#include <stdlib.h>
#include <stdio.h>
main( )
{
    char *ptr1[4]={"china","chengdu","sichuang","chongqin"};
    for (i=0;i<4;i++)
        printf("\n%s",ptr1[i]); //依此输出 ptr1 数组 4 个指针指向
的 4 个字符串
    printf("\n");
    for(i=0;i<3;i++)
        ptr2[i]=&a[i]; //将整型一维数组 a 的 3 个元素的地址传
递给指针数组 ptr2
    for(i=0;i<3;i++) //依此输出 ptr2 所指向的 3 个整型
变量的值
        printf("%4d",*ptr2[i]);
    printf("\n");
    for(i=0;i<3;i++)
        ptr2[i]=b[i]; //传递二维数组 b 的每行首地址给指针数
组的 4 个指针
    for(i=0;i<3;i++) //按行输出
        printf("%4d%4d\n",*ptr2[i],*ptr2[i]+1);
}
```

ptr1 指针数组中的 4 个指针分别指向 4 个字符串，程序中依此输出；ptr2 指针数组共有 3 个指针，若将整型一维数组 a 中各元素地址分别传递给指针数组的各指针，则 ptr2[0] 就指向 a[0]；ptr2[1] 就指向 a[1]；ptr2[2] 就指向 a[2]。若将二维数组各行的首地址分别

传递给指针数组的各指针，这样一来，ptr2[0]就指向了b数组的第0行，该行有两个元素，其地址为ptr2[0]与ptr2[0]+1；相应指针数组第i个元素ptr2[i]指向的b数组的第i行两个元素地址分别为ptr2[i]与ptr2[i]+1。

运行程序结果如下：

```
输入:
china
chengdu
sichuang
chongqin
输出:
123
12
24
56
```

在处理二维字符数组时，可以把二维字符数组看成是由多个一维字符数组构成，也就是说看成是多个字符串构成的二维字符数组，或称为字符串数组。

指针数组对于解决这类问题提供了灵活方便的操作方式。

有一点需要说明，若定义一个指针数组后，指针数组各元素的取值（即地址）要注意其安全性。如定义指针数组：

```
char *ptr[3];
```

那么，该数组包含3个指针，但指针的指向是不确定的，指针现在可能指向内存的任一地址。假定现在作语句：scanf("%s", ptr[i])，则输入的字符串在内存的存放其地址由ptr[i]决定。除非给指针数组元素赋值安全的地址。

[例3-30] 字典排序

定义字符指针数组，包含5个数组元素。同时再定义一个二维字符数组其数组大小为5*10，即5行10列，可存放5个字符串。若将各字符串的首地址传递给指针数组各元素，那么指针数组就成为名副其实的字符串数组。下面对各字符串进行按字典排序。

在字符串的处理函数中，strcmp(str1, str2)函数就可以对两个字符串进行比较，函数的返回值>0、=0、<0分别表示串str1大于str2、str1等于str2、str1小于str2。再利用strcpy()函数实现两个串的复制。下面选用冒泡排序法。

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
main( )
{
    char *ptr1[4], str[4][20], temp[20];
    //定义指针数组、二维字符数组、用于交换的一维字符数组
    int i, j;
    for (i=0; i<4; i++)

```

```

        gets(str[i], 20);                                //输入 4 个字符串
        printf("\n");
        for(i=0;i<4;i++)
            ptr1[i] = str[i];                          //将二维字符数组各行的首地址传
递给指
                                                //针数组的各指针
        printf("original string:\n");
        for(i=0;i <4;i++)                           //按行输出原始各字符串
            printf("%s\n",ptr1[i]);
        printf("ordinal string:\n");
        for(i=0;i<3;i++)                            //冒泡排序
            for ( j = 0 ; j <4-i-1; j++)
                if(strcmp(ptr1[j],ptr1[j+1])>0)
                {
                    strcpy(temp,ptr1[j]);
                    strcpy(ptr1[j],ptr1[j+1]);
                    strcpy(ptr1[j+1],temp);
                }
        for( i=0;i<4;i++)                           //输出排序后的字符串
            printf("%s\n" , ptr1[i]);
    }
}

```

运行程序结果如下：

输入：

```

jkjkdks
fhfgkjkgkf
hkfgkgfkklg
jjkdjdk

```

输出：

original string:

```

jkjkdks
fhfgkjkgkf
hkfgkgfkklg
jjkdjdk

```

ordinal string:

```

fhfgkjkgkf
hkfgkgfkklg
jjkdjdk
jkjkdks

```

程序中一定要注意指针的正确使用。一旦将二维字符数组的各行首地址传递给指针数组

的各指针，则相当于给指针分配了安全可操作的地址，地址空间大小由二维字符数组来决定。当然也可由编译系统为指针分配地址用于字符串的存放。

[例 3-31] 字符串的排序

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
main( )
{
    char *ptr1[4], *temp;
    int i, j;
    init_mempool (0x1000, 0x500);           //初始化内存池
    for (i=0;i<4;i++)
    {
        ptr1[i] = malloc(20) ;             //为指针数组各指针分配 20 字节的
                                         //存储空间
        gets(ptr1[i], 20) ;
    }
    printf("\n") ;
    printf("original string:\n");
    for(i=0;i<4;i++)
        printf("%s\n",ptr1[i]) ;
    printf("ordinal string:\n");
    for( i = 0 ; i < 3 ; i++)
        for( j = 0 ; j < 4 - i - 1 ; j++)
            if(strcmp(ptr1[j],ptr1[j+1])>0)
            {
                temp=ptr1[j];           //利用指向字符串的指针，进行指针地址
                                         //的交换
                ptr1[j]=ptr1[j+1] ;
                ptr1[j+1]=temp;
            }
    for( i=0;i<4;i++)                   //字符串输出
        printf("%s\n" , ptr1[i]);
}
```

运行程序，其结果与上述例 56 完全相同。

[例 3-32] 指定字符串的查找

字符串按字典顺序排列，查找算法采用二分法，或称为折半查找。

分析：折半法也是一种查找的算法，效率比冒泡法要高。具体的分析，有兴趣的读者可

以参考数据结构方面的书籍。

折半查找算法如下：

- ① 设按升序（或降序）输入 n 个字符串到一个指针数组。
- ② 设 low 指向指针数组的低端，high 指向指针数组的高端， $mid=(low+high)/2$ 。
- ③ 测试 mid 所指的字符串，是否为要找的字符串。
- ④ 若按字典顺序，mid 所指的字符串大于要查找的串，表示被查字符串在 low 和 mid 之间，否则，表示被查字符串在 mid 和 high 之间。
- ⑤ 修改 low 或 high 的值，重新计算 mid，继续寻找。

```
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
#include <stdio.h>

main()
{
    char *binary(); // 函数声明
    char *ptr1[5], *temp;
    int i, j;
    init_mempool (0x1000, 0x500); // 初始化内存池
    for (i=0;i<5;i++)
    {
        ptr1[i]=malloc(20); //按字典顺序输入字符串
        gets(ptr1[i], 20);
    }
    printf("\n");
    printf("original string:\n");
    for(i= 0 ; i < 5 ; i++)
        printf("%s\n",ptr1[i]);
    printf("input search string:\n");
    temp=malloc(20);
    gets(temp, 20); //输入被查找字符串
    i=5;
    temp=binary(ptr1, temp, i); //调用查找函数
    if (temp) printf("succesful----%s\n", temp);
    else printf("no succesful!\n");
    return;
}

char *binary(char *ptr[], char *str, int n) //定义返回字符指针的函数
{ //折半查找
    int hig, low, mid;
```

```

low=0;
high=n-1;
while(low<=high)
{
    mid=(low+high)/2 ;
    if (strcmp(str,ptr[mid])<0)
        high=mid-1 ;
    else if(strcmp(str,ptr[mid])>0)
        low=mid+1 ;
    else return(str);                                //查找成功，返回被查字符串
}
return NULL;                                       //查找失败，返回空指针
}

```

运行程序结果如下：

```

输入：
chengdu
chongqin
beijing
tianjin
shanghai
输出：
original string:
chengdu chongqin beijing tianjin shanghai
input search string:
beijing
succesful----- beijing

```

[例 3-33] 插入后排序

分析：我们将一个字符串插入到字符数组中，就必须要先将该字符的位置找出来。查找的算法仍然可以采用上例提供的折半算法。找到位置以后，将字符串插入，插入的位置可以是数组头、中间或数组尾。我们可以将插入的函数独立出来，取名 insert，主要作用是将所有处于插入位置之后的字符串后移，并将新的字符串放入正确的位置上。

insert 函数代码如下：

```

void insert(char *ptr[], char *str, int n, int i)
{
    int j;
    for (j=n;j>i;j--)                                //将插入位置之后的字符串后移
        strcpy(ptr[j],ptr[j-1]) ;
    strcpy(ptr[i],str);                               //将被插字符串按字典顺序插入字
}

```

字符串数组

}

整个程序的源代码如下：

```
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
#include <stdio.h>
main( )
{
    int binary(char *ptr[], char *str, int n);           //查找函数声明
    void insert(char *ptr[], char *str, int n, int i);   //插入函数声明
    char *temp,*ptr1[6];
    int i, j;
    init_mempool(0x1000, 0x500);
    for (i=0;i<5;i++)
    {
        ptr1[i]=malloc(20);                            //为指针分配地址后
        gets(ptr1[i], 20);                           //输入字符串
    }
    ptr1[5]=malloc(20);
    printf("\n");
    printf("original string:\n");
    for(i=0;i<5;i++)                                //输出指针数组各字符串
        printf("%s\n",ptr1[i]);
    printf("input search string:\n");
    temp=malloc(20);
    gets(temp, 20);                                 //输入被插字符串
    i=binary(ptr1, temp, 5);                         //寻找插入位置 i
    printf("i=%d\n", i);
    insert(ptr1, temp, 5, i);                        //在插入位置 i 处插入字符串
    printf("output strings:\n");
    for(i=0;i<6;i++)                                //输出指针数组的全部字符串
        printf("%s\n",ptr1[i]);
}
int binary(char *ptr[], char *str, int n)
{//折半查找插入位置
    int hig, low, mid;
    low = 0 ;
    hig= n - 1 ;
    if (strcmp(str,ptr[0])<0) return 0;
```

```

//若插入字符串比字符串数组的第 0 个小，则插入位置为 0
if (strcmp(str, ptr[hig])>0) return n;
//若插入字符串比字符串数组的最后一个大，则应插入字符串数组的尾部
while(low<=hig)
{
    mid=(low+hig)/2;
    if (strcmp(str, ptr[mid])<0)
        hig=mid-1;
    else if(strcmp(str, ptr[mid])>0)
        low=mid+1;
    else return(mid); //插入字符串与字符串数组的某个字符串
相同
}
return low; //插入的位置在字符串数组中间
}

void insert(char *ptr[], char *str, int n, int i)
{
    int j;
    for (j=n; j>i; j--) //将插入位置之后的字符串后移
        strcpy(ptr[j], ptr[j-1]);
    strcpy(ptr[i], str); //将被插字符串按字典顺序插入字符串数组
}

```

在程序中，字符串数组的 6 个指针均分配存放 20 字节的有效地址。语句 `ptr1[5]=malloc(20)` 保证插入字符串后，也具有安全的存储空间，字符串的长度以串中最长的为基准向系统申请存储空间，以保证在串的移动中有足够的存储空间。

3.3.7 指向指针的指针

一个指针变量可以指向整型变量、实型变量、字符类型变量，当然也可以指向指针类型变量。当这种指针变量用于指向指针类型变量时，我们称之为指向指针的指针变量。

例如，如果整型变量 `i` 的地址是`&i`，将其传递给指针变量 `p`，则 `p` 指向 `i`；实型变量 `j` 的地址是`&j`，将其传递给指针变量 `p`，则 `p` 指向 `j`；字符型变量 `ch` 地址是`&ch`，将其传递给指针变量 `p`，则 `p` 指向 `ch`；整型变量 `x` 的地址是`&x`，将其传递给指针变量 `p2`，则 `p2` 指向 `x`，`p2` 是指针变量，同时，将 `p2` 的地址`&p2` 传递给 `p1`，则 `p1` 指向 `p2`。这里的 `p1` 就是我们谈到的指向指针变量的指针变量，即指针的指针。

指向指针的指针变量定义如下：

类型标识符**指针变量名

例如：

```
float **ptr;
```

其含义为定义一个指针变量 ptr，它指向另一个指针变量（该指针变量又指向一个实型变量）。由于指针运算符“*”是自右至左结合，所以上述定义相当于：

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

下面看一下指向指针变量的指针变量怎样正确引用。

[例 3-34] 用指向指针的指针变量访问一维和二维数组

分析：指向指针的指针其实是二维指针，由前面分析的指针和数组的关系可以知道，二维指针在某些情况下相当于二维数组。由于指针的方便性和易用性，在程序中使用多维指针变量来访问多维数组的情况也相当普遍。

源程序如下：

```
#include <stdio.h>
#include <stdlib.h>
main( )
{
    int a[10], b[3][4], *p1, *p2, **p3, i, j;
    //p3 是指向指针的指针变量
    for( i = 0 ; i < 10 ; i++)
        scanf( "%d", &a[i]) ;
    //一维数组的输入
    for (i=0;i<3;i++)
        for( j = 0 ; j < 4 ; j++)
            scanf( "%d" , &b[i][j]) ;
    //二维数组输入
    for (p1=a, p3=&p1, i=0;i<10;i++)
        printf( "%4d" , *(p3+i) ) ;
    //用指向指针的指针变量输出一维数组
    printf( "\n" ) ;
    for (p1=a;p1-a<10;p1++)
        // 用指向指针的指针变量输出一维数组
    {
        p3 = &p1 ;
        printf("%4d",**p3);
    }
    printf( "\n" ) ;
    for(i=0;i<3;i++)
        // 用指向指针的指针变量输出二维数组
    {
        p2=b[i];
        p3=&p2;
        for (j=0;j<4;j++)
            printf("%4d",*(p3+j));
        printf("\n");
    }
    for(i= 0 ; i < 3 ;i++)
        //用指向指针的指针变量输出二维数组
    {
        p2 =b[i];
```

```

        for(p2 = b[i] ; p2-b[i]<4;p2++)
    {
        p3 = &p2;
        printf("%4d", **p3) ;
    }
    printf("\n");
}
}

```

对一维数组 a 来说，若把数组的首地址即数组名赋给指针变量 p1，p1 就指向数组 a，数组的各元素用 p1 表示为*(p1+i)，也可以简化为*p1+i 表示。

如果继续作将 p3=&p1，则将 p1 的地址传递给指针变量 p3，*p3 就是 p1。用 p3 来表示一维数组的各元素，只需要将用 p1 表示的数组元素*(p1+i) 中的 p1 换成*p3 即可，表示为*(p3+i)。

同样，对二维数组 b 来说，b[i] 表示第 i 行首地址，将其传递给指针变量 p2，使其指向该行。该行的元素用 p2 表示为*(p2+i)。若作 p3=&p2，则表示 p3 指向 p2，用 p3 表示的二维数组第 i 行元素为：*(p3+i)。这与程序中的表示完全相同。

运行程序结果如下：

```

输入:
1 2 3 4 5 6 7 8 9 0
1 3 5 7
2 4 6 8
5 7 9 2
输出:
1 2 3 4 5 6 7 8 9 0
1 2 3 4 5 6 7 8 9 0
1 3 5 7
2 4 6 8
5 7 9 2
1 3 5 7
2 4 6 8
5 7 9 2

```

[例 3-35] 利用指向指针的指针变量对二维字符数组的访问

分析：这个例子在本质上与上面的一个例子是一样的，不过由于二维字符数组一般来说都用作一维的字符串数组，而初学者往往对此容易感到困惑，所以我们在这里专门单举一例来说明这个问题。

源代码如下：

```

#include <stdio.h>
#include <stdlib.h>
main( )

```

```

{
    int i;
    static char c[][16]={"clanguage","fox","computer","home page"};
    //二维字符数组
    static char *cp[]={c[0],c[1],c[2],c[3]};           // 指针数组
    static char **cpp;                                // 指向字符指针的指针变量
    cpp=cp;                                         //将指针数组的首地址传递给指向字符指
    针的指针变量
    for (i=0;i<4;i++)//按行输出字符串
        printf("%s\n",*cpp++);
    printf("-----\n");
    for (i=0;i<4;i++)                               //按行输出字符串
    {
        cpp=&cp[i];
        printf("%s\n",*cpp);
    }
}

```

运行程序结果如下：

输出：

```

clanguage
fox
computer
home page
-----
-c
language
fox
computer
home page

```

程序中需要注意的是，执行 `cpp=cp` 是将指针数组的首地址传递给双重指针，所以`* (cpp+i)` 表示第 `i` 行的首地址，而不是 `cpp+i`。在程序设计时一定分清。

3.4 结构体与共用体

前面的章节中介绍了一些简单数据类型（整型、实型、字符型）的定义和应用，并且介绍了数组（一维、多维）的定义和应用，这些数据类型的特点是：当定义某一特定数据类型时，同时也就限定了该类型变量的存储特性和取值范围。对简单数据类型来说，既可以定义单个的变量，也可以定义数组。而数组的全部元素都具有相同的数据类型，或者说是相同数据类型的一个集合。在日常生活中，我们常会遇到一些需要填写的登记表，如住宿表、成绩

表、通讯地址等。

在这些表中，填写的数据是不能用同一种数据类型描述的，在住宿表中我们通常会登记上姓名、性别、身份证号码等项目；在通讯地址表中我们会写下姓名、邮编、邮箱地址、电话号码、E-mail等项目。这些表中集合了各种数据，无法用前面学过的任一种数据类型完全描述，因此C引入一种能集中不同数据类型于一体的数据类型—结构体类型。结构体类型的变量可以拥有不同数据类型的成员，是不同数据类型成员的集合。

3.4.1 结构体类型变量的定义和引用

在上面描述的各种登记表中，让我们仔细观察一下住宿表、成绩表、通讯地址等，并将这些内容用C提供的结构体类型描述如下：

住宿表：

```
struct accommod
{
    char name[20]; //姓名
    char sex; //性别
    char job[40]; //职业
    int age; //年龄
    long number; //身份证号码
};
```

成绩表：

```
struct score
{
    char grade[20]; //班级
    long number; //学号
    char name[20]; //姓名
    float os; //操作系统
    float datastru; //数据结构
    float compnet; //计算机网络
};
```

通讯地址表：

```
struct addr
{
    char name[20];
    char department[30]; //部门
    char address[30]; //住址
    long box; //邮编
    long phone; //电话号码
    char email[30]; //Email
};
```

这一系列对不同登记表的数据结构的描述类型称为结构体类型。由于不同的问题有不同的数据成员，也就是说有不同描述的结构体类型。我们也可以理解为结构体类型根据所针对的问题其成员是不同的，可以有任意多的结构体类型描述。

下面给出 C 对结构体类型的定义形式：

```
struct 结构体名
{
    成员项表列
};
```

有了结构体类型，我们就可以定义结构体类型变量，以对不同变量的各成员进行引用。

1. 结构体类型变量的定义

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

(1) 先定义结构体类型，再定义结构体类型变量：

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

struct stu student1,student2;           //定义结构体类型变量
struct stu student3,student4;
```

用此结构体类型，可以定义更多的该结构体类型变量。

(2) 定义结构体类型同时定义结构体类型变量：

```
struct data
{
    int day;
    int month;
    int year;
} time1,time2;
```

也可以再定义如下变量：

```
struct data time3,time4;
```

用此结构体类型，同样可以定义更多的该结构体类型变量。

(3) 直接定义结构体类型变量：

```
struct {
    char name[20];                         //学生姓名
    char sex;                            //性别
```

```

long num; //学号
float score[3]; //三科考试成绩
} person1, person2; //定义该结构体类型变量

```

该定义方法由于无法记录该结构体类型，所以除直接定义外，不能再定义该结构体类型变量。

2. 结构体类型变量的引用

学习了怎样定义结构体类型和结构体类型变量，怎样正确地引用该结构体类型变量的成员呢？C 规定引用的形式为：

<结构体类型变量名>. <成员名>

若我们定义的结构体类型及变量如下：

```

struct data
{
    int day;
    int month;
    int year;
} time1, time2;

```

则变量 time1 和 time2 各成员的引用形式为：time1.day、time1.month、time1.year 及 time2.day、time2.month、time2.year。其结构体类型变量的各成员与相应的简单类型变量使用方法完全相同。

3. 结构体类型变量的初始化

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

结构体类型变量的定义和初始化为：

```

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

struct stu student={"liping", 'f', 970541, 98.5, 97.4, 95};

```

上述对结构体类型变量的 3 种定义形式均可在定义时初始化。结构体类型变量完成初始化后，即各成员的值分别为：student.name="liping"、student.sex='f'、student.num=970541、student.score[0]=98.5、student.score[1]=97.4、student.score[2]=95。

我们也可以通过 C 提供的输入输出函数完成对结构体类型变量成员的输入输出。由于结构体类型变量成员的数据类型通常是不一样的，所以要将结构体类型变量成员以字符串的形式输入，利用 C 的类型转换函数将其转换为所需类型。类型转换的函数是：

- ① int atoi(char *str); 转换 str 所指向的字符串为整型，其函数的返回值为整型。
- ② double atof(char *str); 转换 str 所指向的字符串为实型，其函数的返回值为双精度的实型。
- ③ long atol(char *str); 转换 str 指向的字符串为长整型，其函数的返回值为长整型。

使用上述函数，要包含头文件“`stdlib.h`”。

对上述的结构体类型变量输入采用的一般形式：

```
char temp[20];
gets(student.name);                                //输入姓名
student.sex=getchar();                            //输入性别
gets(temp);                                     //输入学号
student.num= atol(temp);                         //转换为长整型
for(i=0;i<3;i++)                                //输入三科成绩
{
    gets(temp);
    student.score[i]=atoi(temp);
}
```

对该结构体类型变量的输出也必须采用各成员独立输出，而不能将结构体类型变量以整体的形式输入输出。

C 允许针对具体问题定义各种各样的结构体类型，甚至是嵌套的结构体类型。

```
struct data
{
    int day;
    Liping f 970541 98.5 97.4 95
    int mouth;
    int year;
};

struct stu
{
    char name[20];
    struct data birthday;                           //出生年月，嵌套的结构体类型
    long num;
}person;
```

该结构体类型变量成员的引用形式：`person.name`、`person.birthday.day`、`person.birthday.month`、`person.birthday.year`、`person.num`。

3.4.2 结构体数组的定义和引用

单个的结构体类型变量在解决实际问题时作用不大，一般是以结构体类型数组的形式出现。结构体类型数组的定义形式为：

```

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

struct stu stud[20]; //定义结构体类型数组 stud ,
//该数组有 20 个结构体类型元素

```

其数组元素各成员的引用形式为：

```

stud[0].name、stud[0].sex、stud[0].score[i];
stud[1].name、stud[1].sex、stud[1].score[i];
...
...
stud[19].name、stud[19].sex、stud[19].score[i];

```

[例 3-36] 成绩统计实例

设某组有 4 个人，填写如下的登记表，除姓名、学号外，还包括 3 门课的成绩，编程实现对表格的计算，求解出每个人的三科平均成绩，求出 4 个学生的单科平均，并按平均成绩由高分到低分输出。

题目要求的问题多，采用模块化编程方式，将问题进行分解如下：

- (1) 结构体类型数组的输入。
- (2) 求解各学生的三科平均成绩。
- (3) 按学生的平均成绩排序。

Number	Name	English	Mathema	Physics	Average
1	Liping	78	98	76	
2	Wangling	66	90	86	
3	Jiangbo	89	70	76	
4	Yangming90	100		67	

(4) 求解组内学生单科平均成绩并输出。

(5) 定义 main() 函数，调用各子程序。

第一步，根据具体情况定义结构体类型。

```

struct stu
{
    char name[20]; //姓名
    long number; //学号
    float score[4]; //数组依此存放 English、Mathema、
Physics, 及 Average
};

```

由于该结构体类型会提供给每个子程序使用，是共用的，所以将其定义为外部的结构体

类型，放在程序的最前面。

第二步，定义结构体类型数组的输入模块。

```
void input(struct stu arr[], int n) //输入结构体类型数组 arr 的 n
个元素
{
    struct stu arr[];
    int n;
    {
        int i, j;
        char temp[30];
        for (i=0;i<n;i++)
        {
            printf("\ninput name, number, English, mathema, physic\n");//打印提示信息
            gets(arr[i].name); //输入姓名
            gets(temp); //输入学号
            arr[i].number= atol(temp);
            for(j=0;j<3;j++)
            {
                gets(temp); //输入三科成绩
                arr[i].score[j]=atoi(temp);
            }
        }
    }
}
```

第三步，求解各学生的三科平均成绩。

在结构体类型数组中第 i 个元素 $arr[i]$ 的成员 $score$ 的前三个元素为已知，第四个 $Average$ 需计算得到。

```
void aver(struct stu arr[], int n)
{
    int i, j;
    for(i=0;i<n;i++) //n 个学生
    {
        arr[i].score[3]=0;
        for(j=0;j<3;j++)
            arr[i].score[3]=arr[i].score[3]+arr[i].score[j]; //求和
        arr[i].score[3]=arr[i].score[3]/3; //平均成绩
    }
}
```

第四步，按平均成绩排序，排序算法采用冒泡法。

```
void order(struct stu arr[], int n)
{
    struct stu temp;
```

```

int i, j, x, y;
for(i=0;i<n-1;i++)
    for(j=0;j<n-1-i;j++)
        if (arr[j].score[3]>arr[j+1].score[3])
            {temp=arr[j];                                //结构体类型变量不允许以整体输入或输出
             but allow mutual assignment
            arr[j]=arr[j+1];                           //进行交换
            arr[j+1]=temp;
            }
}

```

第五步，按表格要求输出。

```

void output(struct stu arr[], int n)           //以表格形式输出有 n 个元素的结构体类型数组各成员
{
    int i, j;
    printf("*****TABLE*****\n"); //打印表头
    printf("-----\n");
    //输出一条水平线
    printf("%10s%8s%7s%7s%7s%7s\n", "Name", "Number", "English", "Mathema",
           "physics", "average");
    //输出效果为：|Name|Number|English|Mathema|Physics|Average|
    printf( " ----- \n" );
    for (i=0;i<n;i++)
    {
        printf("%10s%8d", arr[i].name, arr[i].number); //输出姓名、学号
        for(j=0;j<4;j++)
            printf("%7.2f", arr[i].score[j]);           //输出三科成绩及三科的平均
        printf("\n");
        printf("-----\n");
    }
}

```

第六步，求解组内学生单科平均成绩并输出。在输出表格的最后一行，输出单科平均成绩及总平均。

```

void out_row(struct stu arr[], int n)           //对 n 个元素的结构体类型数组求单项平均
{
    float row[4]={0, 0, 0, 0};                   //定义存放单项平均的一维数组
    int i, j;

```

```

for(i=0;i<4;i++)
{
    for(j=0;j<n;j++)
        row[i]=row[i]+arr[j].score[i];           //计算单项总和
        row[i]=row[i]/n;                         //计算单项平均
}
printf("|%19c|",'');                           //按表格形式输出
for (i=0;i<4;i++)
    printf("%7.2f|",row[i]) ;
printf("\n-----\n");
}

```

第七步，定义 main() 函数，列出完整的程序清单。

```

#include<stdlib.h>
#include <stdio.h>
struct stu
{
    char name[20];
    long number;
    float score[4];
} ;
main()
{
    void input(struct stu arr[], int n);           //函数声明
    void aver(struct stu arr[], int n);
    void order(struct stu arr[], int n);
    void output(struct stu arr[], int n);
    void out_row(struct stu arr[], int n);
    struct stu stud[4];                          //定义结构体数组
    float row[3];
    input(stud, 4);                            //依此调用自定义函数
    aver(stud, 4) ;
    order(stud, 4) ;
    output(stud, 4) ;
    out_row(stud, 4) ;
}
// *****
void input(struct stu arr[], int n)             //输入结构体类型数组 arr 的 n
个元素
{

```

```

int i, j;
char temp[30];
for (i=0;i<n;i++)
{
    printf("\ninput name, number, English, mathema, physic\n");//打印提示信息
    gets(arr[i].name);                                //输入姓名
    gets(temp);                                     //输入学号
    arr[i].number= atol(temp);
    for(j=0;j<3;j++)
    {
        gets(temp);                                //输入三科成绩
        arr[i].score[j]=atoi(temp) ;
    } ;
}
// * * * * * * * * * * * * * * * * * * * * *
void aver(struct stu arr[], int n)
{
    int i, j;
    for(i=0;i<n;i++)                               //n 个学生
    {
        arr[i].score[3]=0;
        for(j=0;j<3;j++)
            arr[i].score[3]=arr[i].score[3]+arr[i].score[j];    //求和
        arr[i].score[3]=arr[i].score[3]/3;                  //平均成绩
    }
}
// * * * * * * * * * * * * * * * * * * * * *
void order(struct stu arr[], int n)
{
    struct stu temp;
    int i, j, x, y;
    for(i=0;i<n-1;i++)
        for(j=0;j<n-1-i;j++)
            if (arr[j].score[3]>arr[j+1].score[3])
                {temp=arr[j];                                //结构体类型变量不允许以整体输入或输出，但允许相互赋值
                 arr[j]=arr[j+1];                         //进行交换
                 arr[j+1]=temp;
                }
}

```

```

}

// *****
void output(struct stu arr[], int n)
{
    int i, j;
    printf("*****TABLE*****\n"); // 打印表头
    printf("-----\n");
    // 输出一条水平线
    printf("|%10s|%8s|%7s|%7s|%7s|\n", "Name", "Number", "English", "Mathema",
           "physics", "average");
    // 输出效果为：|Name|Number|English|Mathema|Physics|Average|
    printf( " ----- \n" );
    for (i=0;i<n;i++)
    {
        printf("|%10s|%8ld|", arr[i].name, arr[i].numbe r); // 输出姓名、学号
        for(j=0;j<4;j++)
            printf("%7.2f|", arr[i].score[j]); // 输出三科成绩及三科的平均
        printf("\n");
        printf("----- \n");
    }
}

// *****
void out_row(struct stu arr[], int n)
{
    float row[4]={0, 0, 0, 0}; // 定义存放单项平均的一维数组
    int i, j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<n;j++)
            row[i]=row[i]+arr[j].score[i]; // 计算单项总和
        row[i]=row[i]/n; // 计算单项平均
    }
    printf("|%19c|,", ' ');
    for (i=0;i<4;i++)
        printf("%7.2f|", row[i]);
    printf("\n----- ");
}

```

```
-- -- - \ n " ) ;
}
```

运行程序结果如下：

输入：

```
Input Name, Number, English, Mathema, Physic
```

Liping

1

78

98

76

```
Input Name, Number, English, Mathema, Physic
```

Wangling

2

66

90

86

```
Input Name, Number, English, Mathema, Physic
```

Jiangbo

3

89

70

76

```
Input Name, Number, English, Mathema, Physic
```

Yangming

4

90

100

输出：

```
* * * * * * * * * * * * * * * * TABLE * * * * * * * * * * * * * * * * * *
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
- - - - - - - - - |
```

Number	Name	English	Mathema	Physics	Average
--------	------	---------	---------	---------	---------

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
- - - - - - - - - |
```

Yangming	4	90.00	100.00	67.00	85.67
----------	---	-------	--------	-------	-------

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
- - - - - - - - - |
```

Liping	1	78.00	98.00	76.00	84.00
--------	---	-------	-------	-------	-------

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
- - - - - - - - - |
```

Wangling 2 66.00 90.00 86.00 80.72					
<hr/>					
Jiangbo 3 89.00 70.00 76.00 78.33					
<hr/>					
80.75 89.50 76.25 82.18					
<hr/>					
<hr/>					

程序中要谨慎处理以数组名作函数的参数。由于数组名作为数组的首地址，在形参和实参结合时，传递给子程序的就是数组的首地址。形参数组的大小最好不定义，以表示与调用函数的数组保持一致。在定义的结构体内，成员 score[3] 用于表示计算的平均成绩，也是我们用于排序的依据。我们无法用数组元素进行相互比较，而只能用数组元素的成员 score[3] 进行比较。在需要交换的时候，用数组元素的整体包括姓名、学号、三科成绩及平均成绩进行交换。在程序 order() 函数中，比较采用：arr[j].score[3]>arr[j+1].score[3]，而赋值则采用：arr[j]=arr[j+1]。

3.4.3 结构体指针的定义和引用

指针变量非常灵活方便，可以指向任一类型的变量，若定义指针变量指向结构体类型变量，则可以通过指针来引用结构体类型变量。

1. 指向结构体类型变量的使用

首先让我们定义结构体：

```
struct stu
{
    char name[20];
    long number;
    float score[4];
};
```

再定义指向结构体类型变量的指针变量：

```
struct stu *p1, *p2;
```

定义指针变量 p1、p2，分别指向结构体类型变量。引用形式为：指针变量→成员；

[例 3-37] 使用指向结构体类型的变量

分析：本例主要希望读者熟悉结构体的每个成员的访问方法。除非为结构体编制一个特定的输出函数，一般来说，结构体作为一个整体不能直接使用标准的输入输出函数。对于每个结构体的输入和输出操作，都必须是通过每一个成员的访问来完成。

具体的源代码如下：

```
#include <stdlib.h> //使用 malloc() 需要
```

```

struct data //定义结构体
{
    int day, month, year;
} ;
struct stu//定义结构体
{
    char name[20];
    long num;
    struct data birthday; //嵌套的结构体类型成员
} ;
main() //定义 main( )函数
{
    struct stu *student; //定义结构体类型指针
    init_mempool(0x1000, 0x500);
    student=malloc(sizeof(struct stu)); //为指针变量分配安全的地址
    printf("Input name, number, year, month, day:\n");
    scanf("%s", student->name); //输入学生姓名、学号、出生年月日
    scanf("%ld", &student->num) ;
    scanf("%d%d%d", &student->birthday.year, &student->birthday.month, &student->birthday.day);
    printf("\nOutput name, number, year, month, day\n");
    //打印输出各成员项的值
    printf("%20s%10d%10d/%d\n", student->name, student->num,
           student->birthday.year, student->birthday.month,
           student->birthday.day) ;
}

```

程序中使用结构体类型指针引用结构体变量的成员，需要通过C提供的函数malloc()来为指针分配安全的地址。函数sizeof()返回值是计算给定数据类型所占内存的字节数。指针所指各成员形式为：

```

student->name
student->num
student->birthday.year
student->birthday.month
student->birthday.day

```

运行程序结果如下：

```

输出：Input name, number, year, month, day:
输入：Wangjian 34 1987 5 23
输出：Wangjian 34 1987//5//23

```

2. 指向结构体类型数组的指针的使用

定义一个结构体类型数组，其数组名是数组的首地址，这一点前面的课程介绍得很清楚。定义结构体类型的指针，既可以指向数组的元素，也可以指向数组，在使用时要加以区分。

在例 37 中定义了结构体类型，根据此类型再定义结构体数组及指向结构体类型的指针如下：

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

struct stu//定义结构体
{
    char name[20];
    long num;
    struct data birthday;           //嵌套的结构体类型成员
};

struct stu student[4], *p;          //定义结构体数组及指向结构体类型的指针
```

如果让 `p=student`，此时指针 `p` 就指向了结构体数组 `student`。

`p` 是指向一维结构体数组的指针，对数组元素的引用可采用 3 种方法。

(1) 地址法

`student+i` 和 `p+i` 均表示数组第 `i` 个元素的地址，数组元素各成员的引用形式为：

`(student+i) ->name`、`(student+i) ->num` 和 `(p+i) ->name`、`(p+i) ->num` 等。`student+i` 和 `p+i` 与 `&student[i]` 意义相同。

(2) 指针法

若 `p` 指向数组的某一个元素，则 `p++` 就指向其后续元素。

(3) 指针的数组表示法

若 `p=student`，我们说指针 `p` 指向数组 `student`，`p[i]` 表示数组的第 `i` 个元素，其效果与 `student[i]` 等同。对数组成员的引用描述为：`p[i].name`、`p[i].num` 等。

[例 3-38] 使用指向结构体数组的指针变量

分析：前面已经说过，结构体与 C 语言中的所有其他类型的变量一样，都可以使用指针来访问。不过有一点区别请读者注意，在使用一般变量时用“.”操作符来完成对成员的访问，而如果是指针的话则要使用操作符“->”。

具体程序如下：

```
struct data           //定义结构体类型
{
    int day, month, year;
};

struct stu           //定义结构体类型
```

```

{
    char name[20];
    long num;
    struct data birthday;
} ;
main( )
{
    int i;
    struct stu *p, student[4]={{"liying", 1, 1978, 5, 23}, {"wangping", 2, 1979, 3, 14},
    {"libo", 3, 1980, 5, 6}, {"xuyan", 4, 1980, 4, 21}};

    //定义结构体数组并初始化
    p=student;           //将数组的首地址赋值给指针p, p指向了一维数组student
    printf("\n1----Output name, number, year, month, day\n");
    for(i=0;i<4;i++)      //采用指针法输出数组元素的各成员

        printf("%20s%10d%10d//%d//%d\n", (p+i)->name, (p+i)->num, (p+i)->birthday.year, (p+i)->birthday.month, (p+i)->birthday.day);

    printf("\n2----Output name, number, year, month, day\n");
    for(i=0;i<4;i++, p++) //采用指针法输出数组元素的各成员

        printf("%20s%10d%10d//%d//%d\n", p->name, p->num,
               p->birthday.year, p->birthday.month,
               p->birthday.day);

    printf("\n3----Output name, number, year, month, day\n");
    for(i=0;i<4;i++)      //采用地址法输出数组元素的各成员

        printf("%20s%10d%10d//%d//%d\n", (student+i)->name, (student+i)->num,
               (student+i)->birthday.year, (student+i)->birthday.month,
               (student+i)->birthday.day);

    p=student;
    printf("\n4----Output name, number, year, month, day\n");
    for(i=0;i<4;i++)      //采用指针的数组描述法输出数组元素的各成员

        printf("%20s%10d%10d//%d//%d\n", p[i].name, p[i].num,
               p[i].birthday.year, p[i].birthday.month,
               p[i].birthday.day);
}

```

运行程序结果如下：

输出：

```

1----Output name, number, year, month, day
liying 1 1978//5//2 3
wangping 2 1979//3//1 4
libo 3 1980//5//6

```

```

xuyan 4 1980//4//2 1
2----Output name, number, year, month, day
liying 1 1978//5//2 3
wangping 2 1979//3//1 4
libo 3 1980//5//6
xuyan 4 1980//4//2 1
3----Output name, number, year, month, day
liying 1 1978//5//2 3
wangping 2 1979//3//1 4
libo 3 1980//5//6
xuyan 4 1980//4//2 1
4----Output name, number, year, month, day
liying 1 1978//5//2 3
wangping 2 1979//3//1 4
libo 3 1980//5//6
xuyan 4 1980//4//2 1

```

对二维或多维数组的指针，有兴趣的读者可以自己编写小程序试验一下。

3.4.4 共用体

所谓共用体类型是指将不同的数据项组织成一个整体，它们在内存中占用同一段存储单元。其定义形式为：

```

union 共用体名
{成员表列}；

```

1. 共用体的定义

```

union data
{
    int a ;
    float b ;
    double c;
    char d;
} obj;

```

该形式定义了一个共用体数据类型 union data，定义了共用体数据类型变量 obj。共用体数据类型与结构体在形式上非常相似，但其表示的含义及存储是完全不同的，举例如下。

[例 3-39] 使用共用体实例

分析：我们要了解到 union 中的每一个元素是公用存储空间的，而 struct 中的每一个元素则分别有各自专用的地址空间。为了理解他们，最好的方法就是比较它们占用存储空间的大小。

具体的代码段如下：

```
union data          //共用体
{
    int a;
    float b;
    double c;
    char d;
} mm;

struct stud        //结构体
{
    int a;
    float b;
    double c;
    char d;
};

main( )
{
    struct stud student
}
```

在编译环境中稍做设置：将 Option for Target 对话框中的 Listing 页的 C compiler 下 Symbols 复选框选中（如图 3-1 所示），就可以在生成的.1st 文件中看到变量的占用空间。为了读者阅读方便，在此仅摘录两条与我们所关心的结构和共用体有关的两句话如下：

```
...
.

stud . . . . . . . . . . . . * TAG * ----- STRUCT ----- 11
    a. . . . . . . . . . . MEMBER ----- INT      0000H 2
    b. . . . . . . . . . . MEMBER ----- FLOAT   0002H 4
    c. . . . . . . . . . . MEMBER ----- FLOAT   0006H 4
    d. . . . . . . . . . . MEMBER ----- CHAR    000AH 1

...
.

mm . . . . . . . . . . . . PUBLIC XDATA UNION 0000H 4
.
```

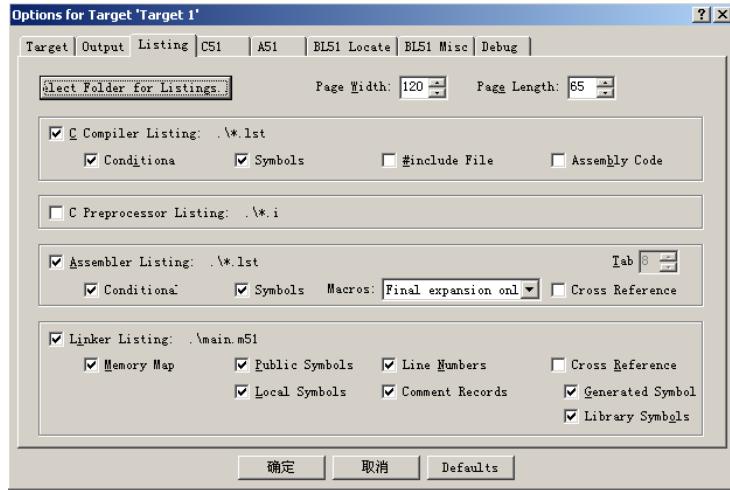


图 3-1 编译环境设置

从输出文件中可以看到，结构体类型所占的内存空间为其各成员所占存储空间之和（11Byte）。而形同结构体的共用体类型实际占用存储空间为其最长的成员所占的存储空间（4Byte）。

对共用体的成员的引用与结构体成员的引用相同。但由于共用体各成员共用同一段内存空间，使用时，根据需要使用其中的某一个成员，方便程序设计人员在同一内存区对不同类型的数据类型的交替使用，增加灵活性，节省内存。

2. 共用体变量的引用

可以引用共用体变量的成员，其用法与结构体完全相同。若定义共用体类型为：

```
union data //共用体
{
    int a;
    float b;
    double c;
    char d;
} mm;
```

其成员引用为：mm.a, mm.b, mm.c, mm.d

[例 3-40] 对共用体变量的使用

分析：由于共用体存储结构上的特殊，所以要注意的是，除非特殊安排，否则不能同时引用四个成员，在某一时刻，只能使用其中之一的成员。

具体代码如下：

```
union data1 //共用体定义
{
    int a;
    float b;
```

```

        double c;
        char d;
    } mm;

main()
{
    mm.a=6;                                //共用体成员赋值与输出
    printf("%d\n", mm.a);
    mm.c=67.2;
    printf(" %5.1f\n ", mm.c);
    mm.d='W';
    mm.b=34.2;
    printf( "%5.1f,%c\n" , mm.b, mm.d);
}

```

运行程序输出为：

```

6
67.2
34.2, =

```

程序最后一行的输出数据是我们无法预料的。其原因是连续执行 `mm.d='W'; mm.b=34.2;` 两个连续的赋值语句，最终使共用体变量的成员 `mm.b` 所占四字节被写入 34.2，而写入的字符被覆盖了，输出的字符变成了符号“=”。事实上，字符的输出是无法得知的，由写入内存的数据决定。

例子虽然很简单，但却说明了共用体变量的正确用法。

[例 3-41] 通过共用体成员显示其在内存的存储情况

分析：我们可以定义一个名为 time 的结构体，再定义共用体 dig：

```

struct time
{
    int year;                                //年
    int month;                               //月
    int day;                                 //日
};

union dig
{
    struct time data;                         //嵌套的结构体类型
    char byte[6];
};

```

假定共用体的成员在内存的存储是从地址 1000 单元开始存放，整个共用体类型需占存储空间 6 个字节，即共用体 dig 的成员 data 与 byte 共用这 6Byte 的存储空间。

由于共用体成员 data 包含三个整型的结构体成员，各占 2 个字节。其中，`data.year` 是

由 2Byte 组成，用 byte 字符数组表示为 byte[0] 和 byte[1]。byte[1] 是高字节，byte[0] 是低字节。下面用程序实现共用体在内存中的存储。

```

struct time
{
    int year;                                //年
    int month;                               //月
    int day;                                 //日
} ;

union dig
{
    struct time data;                      //嵌套的结构体类型
    char byte[6];
} ;

main( )
{
    union dig unit;
    int i;
    printf("enter year:\n");
    scanf("%d",&unit.data1.year);           //输入年
    printf("enter month:\n");
    scanf("%d",&unit.data1.month);          //输入月
    printf("enter day:\n");
    scanf("%d",&unit.data1.day);            //输入日
    printf("year=%dmnh%day=%d\n",unit.data1.year,unit.data1.month,unit.data1.day);// 打
印输出
    for(i=0;i<6;i++)
        printf("%d,",unit.byte[i]);          //按字节以十进制输出
    printf("\n");
}

```

运行程序结果如下：

输入：

enter year:

1976

enter month:

4

enter day:

23

输出：

r=1976 month=4 day=23

184, 7, 4, 0, 23, 0

从程序的输出结果来看，1976 占两个字节，由第 0、1 字节构成，即 $7 \times 256 + 184 = 1976$ 。4 同样占两个字节，由第 2、3 字节构成， $0 \times 256 + 4 = 4$ ，23 由第 4、5 字节构成， $23 = 0 \times 256 + 23$ 。

3.5 小结

本章为读者介绍了 C 语言高级编程的相关知识，结合前一章介绍的 C 语言基本概念、以及各种数据类型以及常用的指令，读者到此已经对 C 语言有了整体的了解，在后面的章节中，本书将着重于 C51 与 ANSI C 的不同之处，以及 C51 与硬件的结合使用等方面进行介绍。

第 4 章 C51 程序设计

4.1 C51 对标准 C 语言的扩展

在前面的章节中已经提到，C51 是以 PC 机上的 ANSI C 为基础发展起来的，但是由于在单片机上编程的特殊需求，C51 有许多 ANSI C 中没有的特殊关键字，称之为 C51 的扩展关键字。下面就是所有可用的 C51 关键字。

at	idata	sfr
alien	interrupt	sfr16
bdata	large	small
bit	pdata	_task_
code	_priority_	using
compact	reentrant	xdata
data	sbit	

4.1.1 存储区域

在第 1 章，我们就已经知道，51 系列单片机的内存区域可以被分为两大类。一类是程序存储区，也就是 ROM，另一类是数据存储区，包括内部数据存储区的和外部数据存储区（外设 RAM）。内部数据存储区中又包含了 51 的特殊寄存器。下面将会分别结合相应的特殊关键字对这些类型的存储区域作相应的说明。

1. 程序存储区

程序存储区域由 code 关键字进行说明，在不使用 code banking 的情况下，该种类型的存储器最多可达 64kByte。程序存储区可以是片内自带的 FLASH，也可以是外挂的 ROM，依系统的硬件设置而异。该存储区是只读的，内容为所有的用户程序以及库函数，程序中的一些常量也存放在这个区域中。请注意，对 51 系列芯片来说，被执行的程序只能放在该区域中，而不能存放在数据区。

2. 内部数据存储区

内部数据存储器在 51 系列 CPU 的内部，可以进行读写操作。根据 51 系列单片机的不同型号，最多有 256Byte 的内部数据存储区可以使用。其中，前 128Byte 既可以被直接寻址，也可以被间接寻址，这其中还包括从 20H 开始的 16 个可以位寻址的字节。剩下的 128Byte 之可以被间接寻址。

由于使用 8bit 地址，内部数据存储区的访问速度会比外部存储器的速度快。在 C51 中可用以下关键字对变量进行内部数据存储区的定位。

- **data**: 直接寻址区，为内部 RAM 的低 128 字节 00H~7FH。
- **idata**: 间接寻址区，包括整个内部 RAM 区 00H~FFH。
- **bdata**: 可位寻址区，20H~2FH。

3. 外部数据存储区

外部数据存储区域也是可以进行读写操作的区域。该区域一般由硬件上外挂 RAM 或者映射成内存的 I/O 来实现。由于需要使用 16 位的数据指针 DPTR 寄存器，所以相比于内部数据存储区来说，对外部 RAM 的访问会慢得多。不过，外部数据存储区大小可以达到 64kB。

在 C51 中，有两个关键字是专门为外部数据存储区准备的，他们是 **xdata** 和 **pdata**。

- **xdata**: 可指定多达 64kB 的外部直接寻址区，地址范围 0000H~0FFFFH。
- **pdata**: 能访问 1 页 (25bBytes) 的外部 RAM，主要用于紧凑模式 (Compact Model)。

4. 特殊寄存器区

8051 提供 128Bytes 的 SFR 寻址区，这区域可位寻址、字节寻址，用以控制定时器、计数器、串口、I/O 及其他部件，可由以下几种关键字说明。

- **sfr**: 字节寻址，比如 **sfr P0=0x80**; 为 P0 口地址为 80H，“=”后 H~FFH 之间的常数。
- **sfr16**: 字寻址，如 **sfr16 T2=0xcc**; 指定 Timer2 口地址 T2L=0xcc T2H=0xCD。
- **sbit**: 位寻址，如 **sbit EA=0xAF**; 指定第 0xAF 位为 EA，即中断允许。

还可以有如下定义方法，比如 **sbit OV=PSW^2**; 或 **sbit OV=0XD0^2**; 或 **bit OV-=0xD2** 的含义是均是定义 OV 为 PSW 的第 2 位。

4.1.2 数据变量分类

在 C51 中，扩展了若干数据类型，例如比特寻址等。下面分别结合例子进行详细说明。

1. bit 型变量

C51 提供了比特型变量，该种类型的变量可以用于变量的定义和声明、函数的参数传递和函数的返回值中。比特类型的定义方式与 C 语言中的其他数据类型一样。例如：

```
static bit done_flag = 0;           //比特类型变量
bit testfunc()                     //比特类型函数
{
    bit flag1,                      //比特参数
    bit flag2)
{
    .
    .
    .
    return(0);                     //返回比特类型
}
```

所有的比特类型的变量都被存放在 51 系列单片机的一片内部数据存储区中，该区域通

常被称为比特寻址区。因为该区域只有 16Byte，所以在某个程序区中，比特类型的变量只可以有 128 个。

比特类型的变量也可以进行存储区域的定位。由于该类型变量只可能存在于内部数据存储区中，所以只有 data 和 idata 型的定义是有效的。

不仅如此，使用比特型的变量还有如下 3 种限制：

①如果在函数中禁止使用中断（即有#pragma disable 的函数），以及使用了显式的 register bank 的函数，那么函数将不允许返回比特类型的值。

②比特类型的值不能被声明为指针。例如：bit *ptr；是非法的。

③比特类型不可以声明数组。例如：bit ware[5]；是非法的。

2. 可按位寻址的数据

比特可寻址的对象也可以被按照字节的方式被访问。在 C51 中，使用 bdata 关键字来定义这种变量。例如：

```
int bdata ibase; //比特寻址区的 int 型变量
char bdata bary [4]; //比特寻址区的数组
```

上面的定义中，ibase 和 bary 都是可比特寻址的变量。也就是说，上述变量中的每个比特都可以直接被访问和修改。使用 sbit 关键字就可以在已经定义的 idata 变量的基础上声明新的变量。例如：

```
sbit mybit0 =ibase^0; //ibase 的第 0 比特
sbit mybit15 =ibase^15; //ibase 的第 15 比特
sbit Ary07 =bary[0]^7; //bary[0]的第 7 比特
sbit Ary37 =bary[3]^7; //bary[3]的第 7 比特
```

上面的例子代表新声明一些变量，而不是把原来已经定义的比特分配出去。C51 中的标识符[^]表示为新的比特型的变量指定具体的位置，后面带的参数必须是一个常数。该常数的范围取决于以前定义的 bdata 变量的类型，例如，如果是 char 或 unsigned char 类型，那么这个常数的范围就是 0~7；如果是 int，unsigned int，short，或者 unsigned short，那么常数的范围就是 0~15；如果 bdata 变量是 long 或者 unsigned long 型的，那么这个范围就是 0~31。

用户可以使用外部定义在其他模块中定义各比特，然后再在程序中来使用。例如：

```
extern bit mybit0; //ibase 的第 0 比特
extern bit mybit15; //ibase 的第 15 比特
extern bit Ary07; //bary[0]的第 7 比特
extern bit Ary37; //bary[3]的第 7 比特
```

使用 sbit 定义新的变量的时候请注意，除非是对 sfr，否则必须要先使用 bdata 关键字定义一个变量以后才能在该变量的基础上使用 sbit。

下面的例子说明了如何使用上面的声明：

```
Ary37 = 0; //bary[3]的第 7 比特清 0
bary[3] = 'a'; //bary[3]的整个字节改动
ibase = -1; //Word (2 个字节) 该动
mybit15 = 1; //ibase 的 15 比特置 1
```

bdata类型的变量在程序的处理过程和data型的变量是一样的，读者只需要注意程序中所有的bdata类型的变量总长不可以超过16Byte。

当然，也可以为结构和联合这两种数据类型来定义sbit型的变量。例如：

```
union lft
{
    float mf;
    long ml;
};

bdata struct bad
{
    char m1;
    union lft u;
} tcp;

sbit tcpf31 = tcp.u.ml ^ 31; //float类型的变量的第31比特
sbit tcpm10 = tcp.ml ^ 0;
sbit tcpm17 = tcp.ml ^ 7;
```

在此请读者注意，sbit型的变量不能定义在float类型的数据上。不过，可以先定义一个float和long组成的结构，然后再通过对long型的数据的某个比特进行定义，以此来访问float类型中的比特。

3. 特殊寄存器

前面已经说过，51系列单片机提供了一块与众不同的寄存器区域，被称为特殊寄存器区，也就是SFR。SFR在程序中被用来控制定时器、计数器、串口、输入输出端口以及各种外设。SFR在51单片机中处于地址0x80到0xFF的位置，并可以被按照比特、字节或者字的方式来访问。如果读者要了解更多的内容，请参考本书的第1章。

在51系列单片机中，SFR的类型和数量都不相同。不过对SFR的定义都已经在头文件中进行了说明。C51对SFR寄存器提供了sfr、sfr16和sbit这几种关键字，下面分别进行说明。

(1) sfr

sfr的定义格式和其他变量一样。唯一的不同之处就是前面不再需要加char或者int关键字。例如：

```
sfr P0 = 0x80; //P0端口，地址为80h
sfr P1 = 0x90; //P1端口，地址为90h
sfr P2 = 0xA0; //P2端口，地址为OA0h
sfr P3 = 0xB0; //P3端口，地址为OB0h
```

P0、P1、P2和P3是特殊寄存器的名字，后面的值则是该寄存器的地址。地址范围必须是(0x80到0xFF)。

(2) sfr16

在比较新一些的51系列芯片中，有时会使用连续的2个特殊寄存器来组成一个16Byte

的数据。例如，8052 使用 0xCC 和 0xCD 来作为定时器/计数器的低位和高位字节。而 sfr15 关键字就是用来对这种类型的存储器进行定义的。

使用 sfr16 关键字时，必须确保低字节在高字节之前，并在定义时使用低字节来进行定义。例如：

```
sfr16 T2 = 0xCC;                                // 定时计数器 2: T2L 0CCh, T2H 0CDh
sfr16 RCAP2 = 0xCA;                                // RCAP2L: 0CAh, RCAP2H: 0CBh
```

在这个例子中，T2 和 RCAP2 都被定义成了 16Byte 的特殊寄存器。sfr16 的定义格式和 sfr 相同，在此不再多说。

(3) sbit

在 51 系列单片机的使用中，常常会碰到只需要使用某一个比特的情况，sbit 关键字就可以在这种情况下发挥巨大的作用。在前面的部分已经简单提过 sbit 的用法，在这里再对这个关键字进行一些说明：

- 可以用已经定义好的 bdata 或者 SFR 变量名的基础上再进行 sbit 的定义，形式为

```
sfr_name ^ int_constant
```

例如：

```
sfr PSW = 0xD0;
sfr IE = 0xA8;
sbit OV = PSW ^ 2;
sbit CY = PSW ^ 7;
sbit EA = IE ^ 7;
```

- 可以使用直接指出特殊寄存器地址，再在这个地址的基础上进行定义，形式为：

```
int_constant ^ int_constant
```

例如：

```
sbit OV = 0xD0 ^ 2;
sbit CY = 0xD0 ^ 7;
sbit EA = 0xA8 ^ 7;
```

- 可以直接使用位地址进行定义，形式为：

```
int_constant
```

例如：

```
sbit OV = 0xD2;
sbit CY = 0xD7;
sbit EA = 0xAF;
```

不过请读者注意，并不是所有的 SFR 都可以位寻址的，因此，不是所有的 SFR 都可以使用 sbit 关键字的。

4.1.3 存储器模式

存储模式和程序模式一样，也可以分为 3 种，分别为 small, compact 和 large 模式。对存储模式的选定是在 C51 编译器选项中选择的。它决定了没有明确指定存储类型的变量，

函数参数等数据的默认存储区域。如果在某些函数中需要使用非默认的存储模式，也可以使用关键字直接说明。

1. small 模式

small 模式中，所有缺省变量参数均装入内部 RAM（与使用显式的 data 关键字来定义是一样的结果）。使用该模式的优点是访问速度快，缺点是空间有限，而且是对堆栈的空间分配比较少，难以把握，碰到需要递归调用的函数的时候需要小心。所以这种模式只适用于小程序。

2. compact 模式

所有缺省变量均位于外部 RAM 区的一页（和显式的使用关键字 pdata 来定义效果是相同的。）最大变量数为 256kB，优点是空间较 Small 为宽裕速度较 Small 慢，较 large 要快，是一种中间状态。使用 compact 模式时，可能会依据地址空间结构而受到一些限制（与 R0 和 R1 有关）。使用本模式时，程序通过@R0 和@R1 指令来进行访问存储器的操作（这两个寄存器是用来提供低位字节的地址的）。如果在 compact 模式下要使用多于 256Byte 的变量，高位字节（也就是具体哪一页）可由 P2 口指定，在 STARTUP.A51 文件中说明，也可用 pdata 指定。

3. large 模式

在 large 模式中，所有缺省变量可放在多达 64kB 的外部 RAM 区（和显式的使用 xdata 关键字来定义是相同的），均使用数据指针 DPTR 来寻址。这种模式的优点是空间大，可存变量多，缺点是速度较慢，尤其对于 2 个以上的多字节变量的访问速度来说更是如此。

4.1.4 绝对地址的访问

在一些情况下，可能希望把一些变量定位在 51 单片机的某个固定的地址空间上。C51 为此专门提供了一个关键字_at_。_at_的使用格式如下：

[memory_space] type variable_name_at_ constant;

格式中各参数的含义如下：

memory_space：变量的存储空间。如果没有这一项的话会使用默认的存储空间。

type：变量类型。

variable_name：变量名。

at：C51 关键字。

constant：常量。该常量的值为变量定位的地址值。这个值必须在设置中的物理地址范围之内，否则 C51 编译器会报错。

下面，举几个例子来说明一下_at_的用法：

```
struct link
{
    struct link idata *next;
    char code *test;
```

```

};

idata struct link list _at_ 0x40; //list 结构在 idata 类型数据区的 0x40
地址处

xdata char text[256] _at_ 0xE000; //数组在 xdata 型存储区的 0xE000
地址处

xdata int i1 _at_ 0x8000; //int 类型变量，存放在 xdata 的 0x8000
地址处

void main( void ) {
    link.next =(void *) 0;
    i1 = 0x1234;
    text [0] = 'a';
}

```

有时候，也许会希望在某段代码中定义一个变量或者类型，而希望在别的代码段中对其进行定位，这在 C51 中也是允许的。例如：

```

struct link
{
    struct link idata *next;
    char code *test;
};

extern idata struct link list; //list 结构在 idata 类型数据区的
0x40 地址处

extern xdata char text[256]; //数组在 xdata 型存储区的 0xE000 地址处
extern xdata int i1; //int 类型变量，存放在 xdata 的 0x8000
地址处

```

关于_at_关键字，有两点请读者在使用时注意：

- 绝对地址的变量是不可以被初始化的。
- 函数或者类型为 bit 的变量是不可以被定为成绝对地址的。

4.1.5 指针

指针是 C 语言的精华，在 C51 中也是程序访问硬件必不可少的工具。与 C 语言一样，C51 支持使用*符号定义一个指针。下面对指针的若干类型及其特性进行讨论。

1. 通用指针

通用指针和一般的 C 语言中的指针定义相同。例如：

```

char *s; //字符串指针
int *numptr; // int 指针
long *state; //long 指针

```

通用指针通常用 3 个字节来存储：第一个字节为存储器的类型，第二个为高字节偏移地址，第三个则是低字节偏移地址。通用指针可以被用来指示 51 单片机存储器中的任何类型的

变量，所以在C51库函数中通常使用这种指针类型。

下面的代码和编译的结果显示了使用通用指针的变量在51单片机中是如何实现的。请读者注意指针的各个字节的作用。

C程序如下：

```
char *c_ptr;                                // char 指针
int *i_ptr;                                  // int 指针
long *l_ptr;                                 // long 指针

void main(void)
{
    char data dj;                           // data 区变量
    int data dk;
    long data dl;

    char xdata xj;                         // xdata 区变量
    int xdata xk;
    long xdata xl;

    char code cj = 9;                      // code 区变量
    int code ck = 357;
    long code cl = 123456789;

    c_ptr = &dj;                           // data 区指针
    i_ptr = &dk;
    l_ptr = &dl;

    c_ptr = &xj;                           // xdata 区指针
    i_ptr = &xk;
    l_ptr = &xl;

    c_ptr = &cj;                           // code 区指针
    i_ptr = &ck;
    l_ptr = &cl;
}
```

在上面的例子中，通用指针c_ptr, i_ptr, 以及l_ptr都被存放在51单片机的内部数据存储区中。当然，如果有需要，也可以使用以前介绍过的关键字对指针的存储位置进行声明，如：

char * xdata strptr;	// 存放在 xdata 的通用指针
int * data numptr;	// 存放在 data 的通用指针

```
long * idata varptr; // 存放在 idata 的通用指针
```

2. 存储器专用指针

这种指针一般来说在定义的时候包含了数据类型和数据空间的说明。例如：

```
char data *str; //data 的字符串指针
int xdata *numtab; //xdata 的 int 指针
long code *powtab; //code 的 long 指针
```

因为数据类型会在编译的时候处理，所以通用指针中的用来存放数据类型的指针在这里并不需要。所以，存储器专用指针只需要一个字节（当数据类型为 idata, data, bdata 或者 pdata 时）或者两个字节（当数据类型为 code 或者 xdata 类型时）。

由于专用指针对比于通用指针来说少了一个字节来指示类型，所以在程序执行的时候会快一些。而且由于专用指针的一些特性是在编译时由编译器来处理的，所以优化选项有时会对编译的结果产生一些影响。

与通用指针相同，读者也可以为专用指针指定存储空间。例如：

```
char data *xdata str; //xdata 的指针，指向 data 的 char 类型
int xdata *data numtab; //data 的指针，指向 xdata 的 int 类型
long code *idata powtab; //idata 的指针，指向 code 的 long 类型
```

3. 指针变换

指针变换也是 C 语言中一个相当重要的特点。而对于 C51 来说，由于区分了存储器专用指针和通用指针，指针变换的含义又更加的丰富。C51 编译器支持这两种不同类型的指针的转换。指针类型转换可以通过显式的程序代码完成，或者由编译器自动完成。

在程序中，如果一个存储器专用指针被传递到函数中作为参数，而该函数中需要的参数实际上为通用指针的时候，C51 编译器会自动对其进行转化。这种情况在使用库函数的时候时常发生，如 printf, sprintf, 以及 gets 等。例如：

```
extern int printf(void *format, ...);
extern int myfunc(void code *p, int xdata *pq);
int xdata *px;
char code *fmt = "value = %d | %04XH\n";
void debug_print(void) {
    printf(fmt, *px, *px); //fmt 被转化
    myfunc(fmt, px); //无转化
}
```

在调用函数 printf 时，参数 code 类型的指针 fmt 就会被强制转换为通用指针。

表 4-1 说明了 C51 编译器在进行指针转换时的方法：

表 4-1 C51 编译器在进行指针转换时的方法

指针转换类型	说明
通用指针转为 code *	通用指针的偏移部分的 2 个字节使用，其他的部分丢弃
通用指针转为 xdata *	通用指针的偏移部分的 2 个字节使用，其他的部分丢弃
通用指针转为 data *	通用指针的偏移部分的低字节被使用，其他字节丢弃

通用指针转为 idata *	通用指针的偏移部分的低字节被使用，其他字节丢弃
通用指针转为 pdata *	通用指针的偏移部分的低字节被使用，其他字节丢弃
code * 转为通用指针	通用指针的指针类型字节被写为 0xFF，表示 code 类型，偏移字节与 xdata *中的内容相同
xdata * 转为通用指针	通用指针的指针类型字节被写为 0x01，表示 xdata 类型，偏移字节与 xdata *中的内容相同

续表

指针转换类型	说明
data * 转为通用指针 idata * 转为通用指针	通用指针的指针类型字节被写为 0x00，表示 idata/data 类型，idata data*的一个字节的偏移地址被转换成无符号 int 型，并被用作偏移地址
pdata * 转为通用指针	通用指针的指针类型字节被写为 0xFE，表示 pdata 类型，pdata*的一个字节的偏移地址被转换成无符号 int 型，并被用作偏移地址

下面的列表文件说明了一些指针类型转换的方法，为了方便起见，不妨先把列表文件拆开阅读：

C 代码部分如下：

```

stmt level source
1 int *p1;                                //通用指针(3字节)
2 int xdata *p2;                            //xdata 指针(2字节)
3 int idata *p3;                           //idata 指针(1字节)
4 int code *p4;                            //code 指针(2字节)
5
6 void pconvert(void) {
7 1 p1 = p2;                                // xdata 指针到通用指针
8 1 p1 = p3;                                // idata 指针到通用指针
9 1 p1 = p4;                                // code 指针到通用指针
10 1
11 1 p4 = p1;                               //通用指针到 code 指针
12 1 p3 = p1;                               //通用指针到 idata 指针
13 1 p2 = p1;                               //通用指针到 xdata 指针
14 1
15 1 p2 = p3;                               //idata 指针到 xdata 指针(警告)
*** WARNING 259 IN LINE 15 OF P.C: pointer: different mspace
16 1 p3 = p4;                               //code 指针到 idata 指针(警告)
*** WARNING 259 IN LINE 16 OF P.C: pointer: different mspace
17 1 }
```

相信读者已经可以看明白这仅仅是一个简单的不同类型指针的直接赋值的例子。在这里强调一下，编写这个例子的目的并不是让读者仿效这种做法，因为这种做法在实际程序中，尤其是较大规模的程序中应用起来是很危险的。之所以提供了上面例程，只是为了说明这些指针的转换规则。

由于这并不是 C 程序代码，所以在标号为 15、16 行的代码之间，以及标号 16、17 行的代码之间，列表文件中将会报告警告信息，这正是因为对不同类型的指针直接进行了赋值的

原因。

4. 抽象指针

抽象指针类型能够访问存储器的任何空间的内容。读者甚至可以通过这种类型的指针来访问定位在某个地址的函数。

在这里，主要通过代码的例子来说明抽象指针类型：

```
char xdata *px; //指向 xdata 的指针
char idata *pi; //指向 idata 的指针
char code *pc; //指向 code 的指针
char c; //data 空间中的 char 型变量
int i; //data 空间中的 int 型变量
```

(1) 将 C 程序中的 main 函数地址转化为代码存储段中的一个 char 变量，然后赋给一个指针，该指针是存放在数据存储区的一个变量。

源程序：

```
pc=(void *)main;
```

(2) 将一个变量 i (类型为 int data*) 的地址转化为一个 idata 中存放的 char 指针。

源程序：

```
pi =(char idata *) &i;
```

(3) 把一个指向 xdata 中的 char 类型的指针转化成一个 idata 中的 char 类型指针。

由于 xdata 类型的指针需要 2Byte，而 idata 类型的指针只需要用 1Byte，所以，如果要把一个指向 xdata 中的 char 类型的指针转化成一个 idata 中的 char 类型指针，xdata 类型的指针的高字节将会被忽略，因此有可能得不到需要的结果：

源程序：

```
pi =(char idata *) px;
```

(4) 如果要把 0x1234 这个值赋给一个 code 代码区域中的 char 类型的指针。

源程序：

```
pc = (char code *) 0x1234;
```

(5) 将某些函数的地址直接赋给一个指针。

在有些时候，可能需要将某些函数的地址直接赋给一个指针。例如，如果一个函数被存放在 0xFF00 处，该函数没有参数，并返回一个 int 型的值。如果需要直接调用这个函数，并且将返回的值赋给一个变量 i，那么程序应为：

源程序：

```
i =((int(code *) (void)) 0xFF00)();
```

(6) 把一个存放在 0x8000 中的数据转换成 code 存储区域中的 char 类型指针，并将值赋给变量 c。

源程序：

```
c = *((char code *) 0x8000);
```

(7) 将 0xFF00 中的内容强制转化为 xdata 区域的 char 类型指针，并将这个指针指向的值与变量 c 相加，结果存放在变量 c 中。

源程序：

```
c += *((char xdata *) 0xFF00);
```

(8) 将 0xF0 的值转化为 idata 中的 char 类型指针的值，并与变量 c 相加。

源程序：

```
c += *((char idata *) 0xF0);
```

(9) 将 0xE8 强制转换为一个 pdata 型的 char 指针，并与变量 c 相加。

源程序：

```
c += *((char pdata *) 0xE8);
```

(10) 将 0x2100 转化为一个 code 存储区的 int 型指针，并将指针指向的内容赋给变量 I

源程序：

```
i = *((int code *) 0x2100);
```

(11) 将 0x4000 转化为一个指针，该指针为 xdata 中指向 xdata 中的 char 类型的指针的指针。最后将结果赋给变量 px。

源程序：

```
px = *((char xdata * xdata *) 0x4000);
```

(12) 把 0x4000 转化为一个 xdata 中的指针，该指针指向 xdata 中的 char 类型的指针，并以数组的方式来进行访问。

这个例子中，程序的功能相当于访问数组的第 0 个元素：

源程序：

```
px = ((char xdata * xdata *) 0x4000) [0];
```

(13) 访问数组中的一个元素

读者可以自己把这个例子和上一个例子比较一下。

源程序：

```
px = ((char xdata * xdata *) 0x4000) [1];
```

4.1.6 函数

在 C51 中，函数的定义与 ANSI C 中是相同的。唯一不同的就是可能在函数的后面需要带若干的 C51 专用的关键字。

C51 的函数定义格式如下：

```
[return_type] funcname([args]) [{small | compact | large}] [reentrant]  
[interrupt n] [using n]
```

各参数含义如下：

return_type：返回值类型。

funcname：函数名

args：函数参数列表

{small | compact | large}：函数模式选择。

reentrant：重入函数。

interrupt n：中断函数。

using n：有 code banking 是究竟使用哪一块空间。

下面，将会对这些关键字分别进行说明。

1. 函数模式选择

在没有显式的选择函数模式的情况下，C51 会使用默认的模式来编译。函数模式选择的几个关键字使用方法如下：

```
#pragma small                                // Default to small model
extern int calc(char i, int b) large reentrant;
extern int func(int i, float f) large;
extern void *tcp(char xdata *xp, int ndx) small;
int mtest(int i, int y)                      //Small 模式
{
    return(i * y + y * i + func(-1, 4.75));
}
int large_func(int i, int k) large            //Large 模式
{
    return(mtest(i, k) + 2);
}
```

各种模式的优缺点与前面章节中所提到的存储模式的优缺点完全相同，在此不再赘述。

2. 寄存器 bank 选择

51 单片机的最低端的 32 个字节被分为了 4 个不同的块，每块 8 个字节。程序可以通过 R0 到 R7 来访问这些字节。R0~R7 具体为哪一块中的内容，则通过程序控制字 (PSW) 来访问。寄存器的 banking 功能在中断处理函数或者实时操作系统中尤其有用，因为 CPU 可以通过切换到一个不同的 bank 来执行程序而不需要对若干寄存器进行保存。

using 关键字用来选择哪一个寄存器 bank 供函数使用。例如：

```
void rb_function(void) using 3
{
    .
    .
    .
}
```

using 关键字只可以带一个 0~3 之间的整数作为参数。该关键字对代码的影响如下：

- 当前选定的寄存器 bank 被存储到堆栈中。
- 指定的寄存器 bank 被设置。
- 函数退出时，从前的内容被恢复。

由于函数推出前必须恢复原 PWS 内容，因此 using 关键字不可用在有返回值的函数中。使用 using 关键字的时候一定要小心，否则有可能得到不正确的结果。

一般来说，using 关键字一般在不同的优先级别的中断函数中很有用，这样可以不用在每次中断的时候都对所有寄存器进行保存。

3. 中断函数

对于中断函数应该有 interrupt 关键字。中断函数不可以有输入或者返回值，例如：

```
unsigned int interruptcnt;
unsigned char second;
void timer0(void) interrupt 1 using 2 {
    if(++interruptcnt == 4000) {           //计数到4000
        second++;                         //秒计数
        interruptcnt = 0;                  //清空中断计数
    }
}
```

interrupt 关键字后所带的数字为 51 单片机的中断入口号。标准 51 单片机的基本中断如表 4-2 所示。

表 4-2 标准 51 单片机的基本中断

中断号	描述	入口地址
0	外部中断 0	0003H
1	定时器/计数器 0	000BH
2	外部中断 1	0013H
3	定时器/计数器 1	001BH
4	串口	0023H

表 4-3 列出了所有的中断号和入口地址。

表 4-3 所有的中断号和入口地址

中断号	入口地址
0	0003H
1	000BH
2	0013H
3	001BH
4	0023H
5	002BH
6	0033H
7	003BH
8	0043H
9	004BH
10	0053H
11	005BH
12	0063H
13	006BH
14	0073H
15	007BH
16	0083H
17	008BH
18	0093H
19	009BH
20	00A3H

21	00ABH
22	00B3H
23	00BBH
24	00C3H
25	00CBH
26	00D3H
27	00DBH
28	00E3H
29	00EBH
30	00F3H
31	00FBH

在使用 interrupt 关键字时，编译器有可能会进行以下操作：

- (1) 在函数被激活的时候，如果有需要，将会把 ACCB、DPH、DPL 以及 PSW 中的内容压入堆栈中进行保存。
- (2) 所有正在使用的寄存器都会被推入堆栈中进行保存，除非使用了 using 关键字。
- (3) 在函数退出时，所有的使用的寄存器，以及特殊寄存器，都会被推出堆栈，以恢复函数执行前的状态。
- (4) 函数结束时会调用 51 的 RETI 指令。

在使用 interrupt 关键字时，还应该注意下面的问题：

- 中断函数一定不能有入口参数或者返回值，否则就会编译出错。
- 中断函数不能直接在程序中调用，因为该函数在编译以后是使用 RETI 命令返回的，而不是普通的返回指令。
- 编译器会为每一个中断函数生成一个中断向量。不过，中断向量的产生可以被 NOINTVECTOR 控制指令禁止掉，所以在使用 interrupt 关键字的时候要注意这些设置。
- C51 编译器允许中断号为 0~31，入口地址已经在表 6-4 中给出。

4. 重入函数

重入函数可以被若干个进程同时执行。当一个重入函数正在执行时，另一个进程可以中断执行过程，并开始执行同样的重入函数。一般来说，C51 中的函数不可以被递归调用，其原因是函数的参数和本地变量都被存放在固定的地址空间中。reentrant 关键字定义的函数则没有这样的限制。例如：

```
int calc(char i, int b) reentrant {
    int x;
    x = table [i];
    return(x * b);
}
```

reentrant 定义的函数不光可以递归调用，而且还可以“同时”被 2 个或者多个进程调用。这种性质常常用在实时处理的系统中。

使用 reentrant 关键字时请注意：

- bit 类型的函数不可以被定义为重入函数。
- 重入函数不能被 alien 关键字定义的函数所调用。

- 重入函数的返回值被存放在51硬件的堆栈中。该堆栈受PUSH和POP指令的影响。
- 不同模式的重入函数使用不同的堆栈。

5. 实时参数

_priority_和_task_参数为实时操作系统RTX51所使用的参数。由于实时操作系统不属于本书讨论的范畴，所以如果读者有需要的话，可以参考其他文献。

4.2 C51函数库

C51编译器的运行库中包含有丰富的库函数，使用库函数可以大大简化用户的程序设计工作，提高编程效率。由于8051系列单片机本身的特点，某些库函数的参数和调用格式与ANSI C标准有所不同，例如函数isdigit返回的类型就是bit，而不是char或者int类型。每个库函数都在相应的头文件中给出了函数原型声明，用户如果需要使用库函数，必须在源程序的开始处采用预处理器指令#include将有关的头文件包含起来。如果省略了头文件，将不能保证函数的正确运行。C51库函数种类的选择考虑到了8051系列单片机的结构特性，用户在自己的应用程序中应该尽可能的使用最小的数据类型，以最大限度地发挥8051系列单片机的性能，同时可以减少应用程序的代码长度。下面将对这些库函数进行分类说明。

4.2.1 字符函数 ctype.h

- 函数原型: extern bit isalpha(unsigned char);
再入属性: reentrant
功能: 检查参数字符是否为英文字符，是则返回1，否则返回0。
例子:

```
#include <ctype.h>
#include <stdio.h>

void tst_isalpha(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(isalpha(i) ? "YES" : "NO");

        printf("isalpha(%c) %s\n", i, p);
    }
}
```

- 函数原型: extern bit isalnum(unsigned char);

再入属性: reentrant

功能: 检查参数字符是否位应为字母或者数字字符, 是则返回 1, 否则返回 0。

例子:

```
#include <ctype.h>
#include <stdio.h>

void tst_isalnum(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(isalnum(i) ? "YES" : "NO");

        printf("isalnum(%c) %s\n", i, p);
    }
}
```

- 函数原型: extern bit iscntrl(unsigned char);

再入属性: reentrant

功能: 检查参数值是否在 0x00~0x1F 之间或者等于 0x7F, 是则返回 1, 否则返回 0。

例子:

```
#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_iscntrl(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(iscntrl(i) ? "YES" : "NO");

        printf("iscntrl(%c) %s\n", i, p);
    }
}
```

- 函数原型: extern bit isdigit(unsigned char);

再入属性: reentrant

功能: 检查参数值是否为数字字符, 是则返回 1, 否则返回 0。

例子:

```
#include <ctype.h>
```

```
#include <stdio.h> //提供打印库函数

void tst_isdigit(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(isdigit(i) ? "YES" : "NO");

        printf("isdigit(%c) %s\n", i, p);
    }
}
```

- 函数原型: extern bit isgraph(unsigned char);

再入属性: reentrant

功能: 检查参数是否为可打印字符, 可打印字符的值域为 0x21~0x7E。为真时返回值为 1, 否则返回值为 0。

例子:

```
#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_isgraph(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(isgraph(i) ? "YES" : "NO");

        printf("isgraph(%c) %s\n", i, p);
    }
}
```

- 函数原型: extern bit isprint(unsigned char);

再入属性: reentrant

功能: 除了还接受空格符 (0x20) 以外, 其余与 isgraph 相同。

例子:

```
#include <ctype.h>
#include <stdio.h> //提供打印库函数
```

```

void tst_isprint(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(isprint(i) ? "YES" : "NO");

        printf("isprint(%c) %s\n", i, p);
    }
}

```

- 函数原型: extern bit isprint(unsigned char);

再入属性: reentrant

功能: 检查字符参数是否为标点、空格或者格式字符。如果是空格或者是 32 个标点和格式字符之一(假定使用 ASCII 字符集中 128 个标准字符)则返回 1, 否则返回 0。Ispunct 对下列字符返回 1: 空格 、!、 "、 #、 \$、 %、 ^、 &、 *、 (、)、 _、 +、 .、 /、 :、 ?、 _<、 >、 [、]、 \、 '、 ~、 {、 }、 |。

例子:

```

#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_ispunct(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(ispunct(i) ? "YES" : "NO");

        printf("ispunct(%c) %s\n", i, p);
    }
}

```

- 函数原型: extern bit islower(unsigned char);

再入属性: reentrant

功能: 检查参数字符的值是否为小写字母, 是则返回 1, 否则返回 0。

例子:

```

#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_islower(void) {

```

```

unsigned char i;
char *p;

for(i = 0; i < 128; i++) {
    p =(islower(i) ? "YES" : "NO");

    printf("islower(%c) %s\n", i, p);
}

}

```

- 函数原型: extern bit isupper(unsigned char);

再入属性: reentrant

功 能: 检查参数字符的值是否为大写字母, 是则返回 1, 否则返回 0。

例 子:

```

#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_isupper(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(isupper(i) ? "YES" : "NO");

        printf("isupper(%c) %s\n", i, p);
    }

}

```

- 函数原型: extern bit isspace(unsigned char);

再入属性: reentrant

功 能: 检查参数是否为下列之一: 空格、制表符、回车、换行、垂直制表符和送纸符号。是则返回 1, 否则返回 0。

例 子:

```

#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_isspace(void) {
    unsigned char i;
    char *p;

```

```

    for(i = 0; i < 128; i++) {
        p =(isspace(i) ? "YES" : "NO");

        printf("isspace(%c) %s\n", i, p);
    }

}

```

- 函数原型: extern bit isxdigit(unsigned char);

再入属性: reentrant

功能: 检查参数字符是否为十六进制数字字符, 是则返回 1, 否则返回 0。

例子:

```

#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_isxdigit(void) {
    unsigned char i;
    char *p;

    for(i = 0; i < 128; i++) {
        p =(isxdigit(i) ? "YES" : "NO");

        printf("isxdigit(%c) %s\n", i, p);
    }

}

```

- 函数原型: extern unsigned char tolower(unsigned char);

再入属性: reentrant

功能: 将大写字符转换成小写形式, 如果字符不在 “A” 到 “Z” 之间, 则不做变换而直接返回这个字符。

例子:

```

#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_tolower(void) {
    unsigned char i;

    for(i = 0x20; i < 0x7F; i++) {
        printf("tolower(%c) = %c\n", i, tolower(i));
    }
}

```

```
}
```

- 函数原型: extern unsigned char toupper(unsigned char);

再入属性: reentrant

功能: 将小写字符转换成大写形式, 如果字符不在“a”到“z”之间, 则不做变换而直接返回这个字符。

例子:

```
#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_toupper(void) {
    unsigned char i;

    for(i = 0x20; i < 0x7F; i++) {
        printf("toupper(%c) = %c\n", i, toupper(i));
    }
}
```

- 函数原型: extern unsigned char toint(unsigned char);

再入属性: reentrant

功能: 将ASCII字符的0~9, A~F(大小写无关)转换为十六进制数字, 返回值0H~9H由ASCII字符的0~9得到, 返回值0AH~0FH由ASCII字符A~F(大小写无关)得到。

例子:

```
#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_toint(void) {
    unsigned long l;
    char k;

    for(l = 0; isdigit(k = getchar());
        l *= 10) {

        l += toint(k);
    }
}
```

- 函数原型: #define _tolower(c) ((c)-'A'+'a')

再入属性: reentrant

功能: 该宏将字符c与常数0x20逐位相或。

例 子：

```
#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_tolower(void) {
    unsigned char i;

    for(i = 0x20; i < 0x7F; i++) {
        printf("tolower(%c) = %c\n", i, tolower(i));
    }
}
```

- 函数原型：#define _toupper(c) ((c)-'a'+'A')

再入属性：reentrant

功 能：该宏将字符 c 与常数 0x20 逐位相与。

例 子：

```
#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_toupper(void) {
    unsigned char i;

    for(i = 0x20; i < 0x7F; i++) {
        printf("toupper(%c) = %c\n", i, toupper(i));
    }
}
```

- 函数原型：#define toascii(c) ((c) & 0x7F)

再入属性：reentrant

功 能：该宏将任何整形数值缩小到有效的 ASCII 字符范围之内，它将变量和 0x7F 相与从而去掉第 7 位以上的所有位。

例 子：

```
#include <ctype.h>
#include <stdio.h> //提供打印库函数

void tst_toascii( char c) {
    char k;

    k = toascii(c);
```

```
    printf("%c is an ASCII character\n", k);
}
```

4.2.2 一般I/O函数STDIO.H

C51库中包含有字符I/O函数，他们通过51系列单片机的串行接口工作，如果希望支持其他I/O接口，只需要改动getkey()和putchar()函数，库中所有其他I/O支持函数都依赖于这两个函数模块，不需要改动。另外需要注意，在使用51系列单片机的串行口之前，应先对其进行初始化。例如，以2400波特率(12MHz时钟频率)初始化串行口程序如下：

```
SCON=0x52;
TMOD=0x20;
TH1=0xF3;
TR1=1;
```

- 函数原型：extern char _getkey(void);

再入属性：reentrant

功能：从8051的串口中读入一个字符，然后等待字符输入。这个函数是改变整个输入端口机制时应作修改的唯一一个函数。

例子：

```
#include <stdio.h>

void tst_getkey(void) {
    char c;

    while((c = _getkey()) != 0x1B) {
        printf("key = %c %bu %bx\n", c, c, c);
    }
}
```

- 函数原型：extern char getchar(void);

再入属性：reentrant

功能：该函数使用_getkey()函数从串口读入字符，并将读入的字符马上传给putchar()函数输出，其他与_getkey()函数相同。

例子：

```
#include <stdio.h>

void tst_getchar(void) {
    char c;
```

```

while((c = getchar()) != 0x1B) {
    printf("character = %c %bu %bx\n", c, c, c);
}

}

```

- 函数原型: `extern char ungetchar(char);`

再入属性: reentrant

功能: 将输入的字符回送输入缓冲区, 因此下次 gets 或者 getchar 可以使用该字符。成功时返回 “char”, 失败时返回 EOF, 不能用 ungetchar 处理多个字符。

例子:

```

#include <stdio.h>

void tst_ungetchar(void) {
    char k;

    while(isdigit(k = getchar())) {
        //当 k 是一个数字的时候停留在此循环中
    }
    ungetchar(k);
}

```

- 函数原型: `extern char putchar(char);`

再入属性: reentrant

功能: 通过 51 单片机的串口输出字符, 与 _getkey() 函数一样。

例子:

```

#include <stdio.h>

void tst_putchar(void) {
    unsigned char i;

    for(i = 0x20; i < 0x7F; i++)
        putchar(i);
}

```

- 函数原型: `extern int printf(const char *, ...);`

再入属性: non-reentrant

功能: printf 以一定的格式通过 8051 串行口输出数值和字符串, 返回值为实际输出的字符数。参数可以是字符串指针、字符或者数值, 第一个参数必须是格式控制字符串指针。允许作为 printf 参数的总字节数受 C51 库限制。由于 51 单片机结构上存储空间有限, 在 SMALL 和 COMPACT 编译模式下最大可传递 15 个字节的参数(即 5 个指针, 或者 1 个指针和

3个长字),在LARGE编译模式下,最多可以传递40个字节的参数。格式控制字符串具有如下形式(方括号内是可选项):

```
%[flags][width][.precision]type
```

格式控制串总是以%开始。

flag称为标志字符,用于控制输出位置、符号、小数点以及八进制和十六进制的前缀等等。其内容和意义如表4-4所示。

表4-4 flag的内容和意义

flag	含义
-	输出左对齐
+	输出如果是有符号数值,则在前面加上+/-号
空格	输出值如果为正则左边补以空格,否则不显示空格

续表

flag	含义
#	如果它与0、x或者X连用,则在非0输出值前面加上0、0x或者0X。当它与值类型字符g、G、f、e、E连用时,是输出值中产生一个十进制的小数点
b, B	当它们与格式类型字符d、o、u、x或者X连用时,使参数类型被接受为[unsigned]char,如%bu、%bx等
l, L	他们与格式类型字符d、o、u、x或者X连用时,使参数类型被接受为[signed]long,如%ld、%lx等
*	下一个参数将不做输出

Width用来定义参数欲显示的字符数,它必须是一个正的十进制数,如果实际显示的字符数小于width,在输出左端补以空格。如果width以0开始,则在左端补0。

precision用来表示输出精度,它是由小数点加上一个非负的十进制数构成的。指定精度时可能会导致输出值被截断,或者在输出浮点数时引起输出值的四舍五入。可以用精度来控制输出字符的数目、整数值的位数或者浮点数的有效位数。也就是说对于不同的输出格式,精度具有不同的意义。

type称为输出格式转换字符,其内容和意义如表4-5所示。

表4-5 type的内容和意义

格式转换字符	类型	含义
D	int	有符号十进制数(16位)
u	int	无符号十进制数
o	int	无符号八进制数
x、X	int	无符号十六进制数
f	float	[-]dddd. dddd形式的浮点数
e、E	float	[-]d. ddddE[-]dd形式的符点数
g、G	float	选择e或者f形式中更紧凑的一种输出格式
c	char	单个字符
s	pointer	结束符为“\0”的字符串
p	pointer	带存储器类型标志和偏移的指针M:aaaa。其中,M:=C(ode), D(ata), I(data), P(data);a:=指针偏移值

例子:

```
#include <stdio.h>
```

```

void tst_printf(void) {
    char a;
    int b;
    long c;
    unsigned char x;
    unsigned int y;
    unsigned long z;
    float f,g;
    char buf [] = "Test String";
    char *p = buf;

    a = 1;
    b = 12365;
    c = 0xFFFFFFFF;
    x = 'A';
    y = 54321;
    z = 0x4A6F6E00;
    f = 10.0;
    g = 22.95;

    printf("char %bd  int %d  long %ld\n", a, b, c);
    printf("Uchar %bu  Uint %u  Ulong %lu\n", x, y, z);
    printf("xchar %bx  xint %x  xlabel %lx\n", x, y, z);
    printf("String %s is at address %p\n", buf, p);
    printf("%f != %g\n", f, g);
    printf("%*f != %*g\n", 8, f, 8, g);
}

```

- 函数原型: `extern int sprintf(char *, const char *, ...);`

再入属性: non-reentrant

功 能: 与 printf 功能相似, 但数据不是输出到串行口, 而是通过一个指针 s 送入可寻址的内存缓冲区, 并以 ASCII 码的形式储存。该函数允许的输出参数总字节数与 printf 完全相同。

例 子:

```

#include <stdio.h>

void tst_sprintf(void) {
    char buf [100];
    int n;

```

```

int a,b;
float pi;

a = 123;
b = 456;
pi = 3.14159;

n = sprintf(buf, "%f\n", 1.1);
n += sprintf(buf+n, "%d\n", a);
n += sprintf(buf+n, "%d %s %g", b, "----", pi);
printf(buf);
}

```

- 函数原型: `extern int vprintf(const char *, char *)`;

再入属性: reentrant

功 能: 与 `printf` 的功能基本相同, 但是该函数使用指针来作为参数列表。

例 子:

```

#include <stdio.h>
#include <stdarg.h>
void error(char *fmt, ...) {
    va_list arg_ptr;

    va_start(arg_ptr, fmt); //格式化字符串
    vprintf(fmt, arg_ptr);
    va_end(arg_ptr);
}

void tst_vprintf(void) {
    int i;
    i = 1000;
    //使用一个参数调用 error
    error("Error: '%d' number too large\n", i);
    //使用一串字符调用 error 函数
    error("Syntax Error\n");
}

```

- 函数原型: `extern int vsprintf(char *, const char *, char *)`;

再入属性: reentrant

功 能: 与 `printf` 的功能基本相同, 但是该函数使用指针来作为参数列表。

例 子:

```

#include <stdio.h>
#include <stdarg.h>

xdata char etxt[30]; //文本指针

void error(char *fmt, ...) {
    va_list arg_ptr;

    va_start(arg_ptr, fmt); //格式化
    vsprintf(etxt, fmt, arg_ptr);
    va_end(arg_ptr);
}

void tst_vprintf(void) {
    int i;
    i = 1000;

    //使用 1 个参数调用函数
    error("Error: '%d' number too large\n", i);

    //使用字符串调用函数
    error("Syntax Error\n");
}

```

- 函数原型: extern char *gets(char *s, int n);

再入属性: non-reentrant

功 能: 该函数通过 getchar() 函数从串口中读入一个长度为 n 的字符串并存入由“s”指向的数组。输入时一旦检测到换行符就结束字符输入。输入成功时返回传入的参数指针，失败时返回 NULL。

例 子:

```

#include <stdio.h>

void tst_gets(void) {
    xdata char buf [100];

    do {
        gets(buf, sizeof(buf));
        printf("Input string \"%s\"\n", buf);
    } while(buf [0] != '\0');
}

```

- 函数原型: `extern int scanf(const char *, ...);`

再入属性: non-reentrant

功能: 该函数在格式控制串的控制下, 利用 `getchar` 函数从串行口中读入数据, 每遇到一个符合格式控制串规定的值, 就将它按顺序存入由参数指针指向的存储单元。注意, 每个参数都必须是指针。`scanf` 返回它所发现并转换的输入项数, 若遇到错误则返回 EOF。

格式控制串具有如下形式(方括号内为可选项):

`%[flags][width]type`

格式控制串总是以%开始。

`flag` 称为标识符, 它的内容和意义如表 4-6 所示。

表 4-6

flag 的内容和意义

Flag	含义
b, B	当它们与格式类型字符 d、o、u、x 或者 X 连用时, 使参数类型被接受为 [unsigned]char, 如%bu、%bx 等
l, L	它们与格式类型字符 d、o、u、x 或者 X 连用时, 使参数类型被接受为 [signed]long, 如%ld、%lx 等
*	输入被忽略

`width` 是一个十进制的正整数, 用来控制输入数据的最大长度或者字符数目。

`type` 称为输入格式转换字符, 其内容和意义如表 4-7 所示。

表 4-7

type 的内容和意义

格式转换字符	类型	含义
d	int	有符号十进制数(16位)
u	int	无符号十进制数
o	int	无符号八进制数
x、X	int	无符号十六进制数
f、e、E、g、G	float	浮点数
c	char	单个字符
s	pointer	一个字符串

例子:

```
#include <stdio.h>

void tst_scanf(void)  {
    char a;
    int b;
    long c;

    unsigned char x;
    unsigned int y;
    unsigned long z;

    float f, g;
```

```

char d, buf [10];

int argsread;

printf("Enter a signed byte, int, and long\n");
argsread = scanf("%bd %d %ld", &a, &b, &c);
printf("%d arguments read\n", argsread);

printf("Enter an unsigned byte, int, and long\n");
argsread = scanf("%bu %u %lu", &x, &y, &z);
printf("%d arguments read\n", argsread);

printf("Enter a character and a string\n");
argsread = scanf("%c %9s", &d, buf);
printf("%d arguments read\n", argsread);

printf("Enter two floating-point numbers\n");
argsread = scanf("%f %f", &f, &g);
printf("%d arguments read\n", argsread);

}

```

- 函数原型: extern int sscanf(char *, const char *, ...);

再入属性: non-reentrant

功 能: 该函数与 scanf 的输入方式相似, 但是字符串的输入不是通过串行口, 而是通过另一个以空结束的指针。sscanf 参数允许的总字节数受 C51 库的限制, 在 SMALL 和 COMPACT 编译模式下, 最大允许传递 15Byte 的参数。在 LARGE 编译模式下, 最大允许传递 40Byte 的参数。

例 子:

```

#include <stdio.h>

void tst_scanf(void)  {
    char a;
    int b;
    long c;

    unsigned char x;
    unsigned int y;
}

```

```

void tst_sscanf(void) {
    char a;
    int b;
    long c;

    unsigned char x;
    unsigned int y;
    unsigned long z;

    float f, g;

    char d, buf [10];

    int argsread;

    printf("Reading a signed byte, int, and long\n");
    argsread = sscanf("1 -234 567890", "%bd %d %ld", &a, &b, &c);
    printf("%d arguments read\n", argsread);

    printf("Reading an unsigned byte, int, and long\n");
    argsread = sscanf("2 44 98765432", "%bu %u %lu", &x, &y, &z);
    printf("%d arguments read\n", argsread);

    printf("Reading a character and a string\n");
    argsread = sscanf("a abcdefg", "%c %9s", &d, buf);
    printf("%d arguments read\n", argsread);

    printf("Reading two floating-point numbers\n");
    argsread = sscanf("12.5 25.0", "%f %f", &f, &g);
    printf("%d arguments read\n", argsread);
}

```

- 函数原型: extern int puts(const char *);

再入属性: reentrant

功 能: 将字符串和换行符写入串行口, 错误是返回 EOF, 否则返回一个非负数。

例 子:

```
#include <stdio.h>
```

```
void tst_puts(void) {
```

```

    puts("Line #1");
    puts("Line #2");
    puts("Line #3");

}

```

4.2.3 字符串函数 STRING.H

- 函数原型: void *memccpy (void *dest, void *src, char c, int len);

再入属性: non-reentrant

功 能: 该函数将从 src 开始的字节复制到 dest 所指向的位置。该函数可以复制 0 个或者多个字节。调用该函数以后, 字符串会被复制, 直到 c 字符被复制, 或者复制了 len 个字节为止。该函数返回值为指向最后被复制到 dest 的字节之后的那个字节的指针。

例 子:

```

#include <string.h>
#include <stdio.h>                                //提供打印库函数

void tst_memccpy (void) {
    static char src1 [100] = "Copy this string
        to dst1";
    static char dst1 [100];

    void *c;

    c = memccpy (dst1, src1, 'g', sizeof (dst1));

    if (c == NULL)
        printf ("'g' was not found in the src
            buffer\n");
    else
        printf ("characters copied up to 'g'\n");

}

```

- 函数原型: void *memchr (void *buf, char c, int len);

再入属性: reentrant

功 能: 该函数将从 buf 开始, 长度为 len 的字符串中查找字符 c。如果查找到字符 c, 那么返回值为指向该字符的指针, 如果未找到, 返回值为 NULL。或者为最后被复制到 dest 的字节之后的那个字节的指针。

例 子：

```
#include <string.h>
#include <stdio.h> //提供打印库函数

void tst_memchr (void) {
    static char src1 [100] =
        "Search this string from the start";

    void *c;

    c = memchr (src1, 'g', sizeof (src1));

    if (c == NULL)
        printf ("'g' was not found in the buffer\n");
    else
        printf ("found 'g' in the buffer\n");

}
```

- 函数原型：char memcmp(void *buf1, void *buf2, int len);

再入属性：reentrant

功 能：该函数将比较从 buf1 和 buf2 开始，长度为 len 的两个字符串的大小。如果 buf1 指向的字符串小于 buf2 指向的字符串，那么返回一个小于 0 的值，如果大于，返回一个大于 0 的值，如果相等，那么返回 0。

例 子：

```
#include <string.h>
#include <stdio.h> //提供打印库函数

void tst_memcmp (void) {
    static char hexchars [] = "0123456789ABCDEF";
    static char hexchars2 [] = "0123456789abcdef";

    char i;

    i = memcmp (hexchars, hexchars2, 16);

    if (i < 0)
        printf ("hexchars < hexchars2\n");
    else if (i > 0)
```

```

        printf ("hexchars > hexchars2\n");

    else
        printf ("hexchars == hexchars2\n");

}

```

- 函数原型: void *memcpy (void *dest, void *src, int len);

再入属性: reentrant

功能: 该函数将把长度为 len, 起始地址为 src 的字符串复制到起始地址为 dest 的地方。返回值为 dest。

例子:

```

#include <string.h>
#include <stdio.h>                                //提供打印库函数

void tst_memcpy (void) {
    static char src1 [100] =
        "Copy this string to dst1";

    static char dst1 [100];

    char *p;

    p = memcpy (dst1, src1, sizeof (dst1));

    printf ("dst = \"%s\"\n", p);

}

```

- 函数原型: void *memmove (void *dest, void *src, int len);

再入属性: reentrant

功能: 该函数将把长度为 len, 起始地址为 src 的字符串复制到起始地址为 dest 的地方, 如果存储区域重叠, 该函数保证 src 指向的内容能被复制到 dest 中。返回值为 dest。

例子:

```

#include <string.h>
#include <stdio.h>                                //提供打印库函数

void tst_memmove (void) {
    static char buf [] = "This is line 1 "
        "This is line 2 "

```

```

    "This is line 3  ";

    printf ("buf before = %s\n", buf);

    memmove (&buf [0], &buf [16], 32);

    printf ("buf after  = %s\n", buf);

}

```

- 函数原型: void *memset (void *buf, char c, int len);

再入属性: reentrant

功 能: 该函数将把长度为 len, 起始地址为 buf 的所有字符的内容写为 c。

例 子:

```

#include <string.h>
#include <stdio.h>                                //提供打印库函数

void tst_memset (void)  {
    char buf [10];

    memset (buf, '\0', sizeof (buf));
    //用空字符填满缓存

}

```

- 函数原型: char *strcat (char *dest, char *src);

再入属性: reentrant

功 能: 该函数将 src 指向的字符串复制到 dest 指向的字符串之后。返回值为 dest。

例 子:

```

#include <string.h>
#include <stdio.h>                                //提供打印库函数

void tst_strcat (void)  {
    char buf [21];
    char s [] = "Test String";

    strcpy (buf, s);
    strcat (buf, "#2");

    printf ("new string is %s\n", buf);
}

```

```
}
```

- 函数原型: char *strchr (const char *string, char c);

再入属性: reentrant

功能: 该函数将从起始地址为 string 的字符串中查找出现的第一个 c 字符。返回值为查找到的符合条件的字符指针, 或者为空 (在没有发现匹配字符时)。

例子:

```
#include <string.h>
#include <stdio.h> //提供打印库函数

void tst_strchr (void) {
    char *s;
    char buf [] = "This is a test";

    s = strchr (buf, 't');

    if (s != NULL)
        printf ("found a 't' at %s\n", s);
}
```

- 函数原型: char strcmp (char *string1, char *string2);

再入属性: reentrant

功能: 该函数将比较字符串 string1 和 string2 的大小, 如果前者大于后者, 那么返回大于 0 的值; 前者小于后者则返回一个小于 0 的值, 否则返回 0。

例子:

```
#include <string.h>
#include <stdio.h> //提供打印库函数

void tst_strcmp (void) {
    char buf1 [] = "Bill Smith";
    char buf2 [] = "Bill Smithy";
    char i;

    i = strcmp (buf1, buf2);

    if (i < 0)
        printf ("buf1 < buf2\n");
    else if (i > 0)
```

```

        printf ("buf1 > buf2\n");

    else
        printf ("buf1 == buf2\n");
}

```

- 函数原型: char *strcpy (char *dest char *src);

再入属性: reentrant

功能: 该函数将把 src 指向的字符串复制到 dest。返回值为 dest。

例子:

```

#include <string.h>
#include <stdio.h>                                //提供打印库函数

void tst strcpy (void) {
    char buf [21];
    char s [] = "Test String";

    strcpy (buf, s);
    strcat (buf, "#2");

    printf ("new string is %s\n", buf);
}

```

- 函数原型: int strcspn (char *src, char *set);

再入属性: non-reentrant

功能: 该函数在字符串 src 中查找是否有字符数组 set 中的任何一个元素。返回值为 src 中第一个与字符数组 set 中的某个元素匹配的字节的序号。如果没有找到匹配的字符, 那么返回 src 的长度。

例子:

```

#include <string.h>
#include <stdio.h>                                //提供打印库函数

void tst_strcspn (void) {
    char buf [] = "13254.7980";
    int i;

    i = strcspn (buf, ".,");

    if (buf [i] != '\0')
        printf ("%c was found in %s\n", (char)
buf [i], buf);
}

```

```
}
```

- 函数原型: int strlen(char *src);

再入属性: reentrant

功 能: 该函数返回字符串 src 的长度。

例 子:

```
#include <string.h>
#include <stdio.h> //提供打印库函数

void tst_strlen (void) {
    char buf [] = "Find the length of this string";
    int len;

    len = strlen (buf); // len = 30

    printf ("string length is %d\n", len);

}
```

- 函数原型: char *strncat (char *dest, char *src, int len);

再入属性: non-reentrant

功 能: 该函数将把最长为 len, 起始地址为 src 的字符串, 复制到 dest 字符串之后。

返回值为 dest。

例 子:

```
#include <string.h>
#include <stdio.h> //提供打印库函数

void tst_strncat (void) {
    char buf [21];

    strcpy (buf, "test #");
    strncat (buf, "three", sizeof (buf) - strlen
             (buf));
}
```

- 函数原型: char *strcmp(char *string1, char *string2, int len);

再入属性: reentrant

功 能: 该函数比较字符串 string1 和 string2 的前 len 个字符的大小。

- 函数原型: char *strncpy (char *dest, char *src, int len);

再入属性: reentrant

功 能: 该函数将 src 字符串中最多 len 字节长度的字符复制到 dest 中。

例 子:

```
#include <string.h>
#include <stdio.h> //提供打印库函数

void tst_strncpy ( char *s) {
    char buf [21];

    strncpy (buf, s, sizeof (buf));
    buf [sizeof (buf)] = '\0';
}
```

- 函数原型: char *strpbrk (char *string, char *set);

再入属性: non-reentrant

功 能: 该函数在字符串 src 中查找是否有字符数组 set 中的任何一个元素。返回值为查找到的字符的指针或者 NULL (未查找到的时候)。

例 子:

```
#include <string.h>
#include <stdio.h> //提供打印库函数

void tst_strpbrk (void) {
    char vowels [] ="AEIOUaeiou";
    char text [] = "Seven years ago...";

    char *p;

    p = strpbrk (text, vowels);

    if (p == NULL)
        printf ("No vowels found in %s\n", text);

    else
        printf ("Found a vowel at %s\n", p);
}
```

- 函数原型: int strpos (const char *string, char c);

再入属性: reentrant

功 能: 该函数在字符串 string 中查找字符 c, 返回查找到的字符的序号。如果没有

查找到，则返回-1。

例 子：

```
#include <string.h>
#include <stdio.h> //提供打印库函数
void tst_strpos (void) {
    char text [] = "Search this string for
                    blanks";

    int i;

    i = strpos (text, ' ');

    if (i == -1)
        printf ("No spaces found in %s\n", text);

    else
        printf ("Found a space at offset %d\n", i);
}
```

- 函数原型：char *strrchr (const char *string, char c);

再入属性：reentrant

功 能：该函数与 strchr 相同，只不过是反向查找。

- 函数原型：char *strrpbrk (char *string, char *set);

再入属性：non-reentrant

功 能：功能与 strpbrk 基本相同，只不过是反向查找。

- 函数原型：int strpos (const char *string, char c);

再入属性：reentrant

功 能：与 strpos 基本相同，只不过是反向查找。

- 函数原型：int strspn (char *src, char *set);

再入属性：non-reentrant

功 能：与 strcspn 基本相同，只不过查找的是 set 中没有的字符。

4.2.4 标准函数 STDLIB.H

- 函数原型：float atof (void *string);

再入属性：non-reentrant

功 能：将字符串转换成浮点数。

例 子：

```
#include <stdlib.h>
#include <stdio.h> //提供打印库函数
```

```

void tst_atof (void) {
    float f;
    char s [] = "1.23";

    f = atof (s);
    printf ("ATOF(%s) = %f\n", s, f);
}

```

- 函数原型: int atoi (void *string);

再入属性: non-reentrant

功能: 将字符串转换成整数。

- 函数原型: long atol (void *string);

再入属性: non-reentrant

功能: 将字符串转换成长整数。

- 函数原型: unsigned long strtod (const char *string, char **ptr);

再入属性: non-reentrant

功能: 将字符串转换成整数。

- 函数原型: long strtol (const char *string, char **ptr, unsigned char base);

再入属性: non-reentrant

功能: 将字符串转换成长整数。

```

#include <stdlib.h>
#include <stdio.h>                                // 提供打印库函数
char s [] = "12abCDe8";
void tst_strtol (void) {
    long l;
    l = strtol (s, NULL, 16);
    printf ("strtol(%s) = %lx\n", s, l);
}

#include <reg51.h>
void main (void) {
    SCON = 0x50;                                     // SCON: 模式 1, 8-bit UART, 允许 rcvr
    TMOD |= 0x20;                                    // TMOD: 定时器 1, 模式 2, 8-bit 自动载入
    TH1   = 221;                                     // TH1: 重装值, 16MHz 晶振下 1200bit/s
    TR1   = 1;                                       // TR1: 定时器 1 计数器
    TI    = 1;                                       // TI: TI 置位, 以保证 UART 能发送第一个字节
    tst_strtol ();
}

```

```
}
```

- 函数原型: `unsigned long strtoul (const char *string, char **ptr);`

再入属性: non-reentrant

功 能: 将字符串转换成无符号长整数。

- 函数原型: `void *calloc (unsigned int num, unsigned int len);`

再入属性: non-reentrant

功 能: 申请有 `num` 个元素的, 每个元素的大小为 `len` 的数组。

例 子:

```
#include <stdlib.h>
#include <stdio.h> //提供打印库函数

void tst_calloc (void) {
    int xdata *p; //指针, 指向有 100int 型变量的数组

    p = calloc (100, sizeof (int));

    if (p == NULL)
        printf ("Error allocating array\n");
    else
        printf ("Array address is %p\n", (void *) p);
}
```

- 函数原型: `void *malloc (unsigned int size);`

再入属性: non-reentrant

功 能: 申请一块大小为 `size` 的内存。

- 函数原型: `void *realloc (void xdata *p, unsigned int size);`

再入属性: non-reentrant

功 能: 重新申请一块大小为 `size` 的内存。

- 函数原型: `void free (void xdata *p);`

再入属性: non-reentrant

功 能: 释放一块被申请的内存。

- 函数原型: `void init_mempool (void xdata *p, unsigned int size);`

再入属性: non-reentrant

功 能: 初始化内存池。

- 函数原型: int rand (void);
再入属性: reentrant
功 能: 取一个 0~32767 之间的随机数。

- 函数原型: void srand (int seed);
再入属性: non-reentrant
功 能: 根据种子 seed 的值取随机数。

4.2.5 数学函数 MATH.H

- 函数原型: int abs(int val);
char cabs(char val);
float fabs(float val);
long labs(long val);

再入属性: reentrant
功 能: 计算并返回 val 的绝对值。这 4 个函数除了变量和返回值类型不同以外，没有太大区别。

- 函数原型: float exp(float x);
float log(float x);
float log10(float x);

再入属性: non-reentrant
功 能: exp 返回以 e 为底 x 的幂。Log 返回自然对数，log10 返回以 10 为底的对数。

- 函数原型: float sqrt(float x);
再入属性: non-reentrant
功 能: 返回 x 的正平方根。

- 函数原型: float cos(float x);
float sin(float x);
float tan(float x);

再入属性: non-reentrant
功 能: 返回相应的三角函数值。所有的变量范围必须在 $-\frac{\pi}{2} \sim +\frac{\pi}{2}$ 之间，否则会返回错误。

- 函数原型: float acos(float x);
float asin(float x);
float atan(float x);
float atan2(float y, float x);

再入属性: non-reentrant

功 能: 返回相应的反三角函数值。atan 返回 x 的反正切值, 值域为 $-\frac{\pi}{2} \sim +\frac{\pi}{2}$; atan2

返回 x/y 的反正切值, 值域为 $-\pi \sim +\pi$ 。

- 函数原型: float cosh(float x);
float sinh(float x);
float tanh(float x);

再入属性: non-reentrant

功 能: 返回 x 的相应的双曲函数值。

- 函数原型: void fpsave(struct FPBUF *p);
void fprestore(struct FPBUF *p);

再入属性: reentrant

功 能: fpsave 保存浮点子程序的状态, fprestore 恢复浮点子程序的原始状态, 当中断程序中需要执行浮点运算的时候, 这两个函数是很有用的。

- 函数原型: float ceil(float x);

再入属性: non-reentrant

功 能: ceil 返回一个不小于 x 的最小整数 (作为浮点数)。

- 函数原型: float floor(float x);

再入属性: non-reentrant

功 能: 返回一个不大于 x 的最大整数 (作为浮点数)。

- 函数原型: float modf(float x, float *ip)

再入属性: non-reentrant

功 能: 将浮点数 x 分为整数和小数两个部分, 两者都含有与 x 相同的符号, 整数部分放入*ip, 小数部分作为返回值。

例 子:

```
#include <math.h>
#include <stdio.h> //提供打印库函数

void tst_modf (void)  {
    float x;
    float int_part, frc_part;

    x = 123.456;
    frc_part = modf (x, &int_part);
```

```

    printf ("%f = %f + %f\n", x, int_part, frc_part);

}

```

- 函数原型: float pow(float x, float y);

再入属性: non-reentrant

功 能: 计算。

4.2.6 绝对地址访问 ABSACC.H

- 函数原型: #define CBYTE ((unsigned char volatile code *)0)
#define DBYTE ((unsigned char volatile idata *)0)
#define PBYTE ((unsigned char volatile pdata *)0)
#define XBYTE ((unsigned char volatile xdata*)0)

再入属性: ----

功 能: 上述宏定义用来对 8051 系列单片机的存储器空间进行绝对地址访问, 可以作为字节寻址。CBYTE 寻址 CODE 区, DBYTE 寻址 DATA 区, PBYTE 寻址分页 XDATA 区, XBYTE 寻址 XDATA 区。

例 子:

如果要访问外部数据存储器区域的 0x1000 处的内容, 可以使用如下指令:

```
val=XBYTE[0x1000];
```

- 函数原型: #define CWORD ((unsigned int volatile code *)0)
#define DWORD ((unsigned int volatile idata *)0)
#define PWORD ((unsigned int volatile pdata *)0)
#define XWORD ((unsigned int volatile xdata*)0)

再入属性: ----

功 能: 这个宏与前面的一些宏类似, 只不过数据类型为 unsigned int 型。

4.2.7 内部函数 INTRINS.H

- 函数原型: unsigned char _crol_(unsigned char c, unsigned char b);
unsigned char _irol_(unsigned int i, unsigned char b);
unsigned char _lrol_(unsigned long l, unsigned char b);

再入属性: reentrant/inrinsc

功 能: 将第一个参数循环左移 n 位。这些函数的唯一不同在于参数和返回值的类型不同。

例 子:

使用_lrol_函数的例子:

```
#include <intrins.h>

void tst_lrol (void) {
    long a;
    long b;

    a = 0xA5A5A5A5;

    b = _lrol_(a, 3); // b=0x2D2D2D2D

}
```

使用_crol_函数的例子：

```
#include <intrins.h>

void tst_cror (void) {
    char a;
    char b;

    a = 0xA5;

    b = _crol_(a, 1); // b=0xD2

}
```

- 函数原型: `unsigned char _cror_(unsigned char val, unsigned char n);`
`unsigned char _iror_(unsigned int val, unsigned char n);`
`unsigned char _lror_(unsigned long val, unsigned char n);`

再入属性: reentrant/intrinsicsc

功能: 将第一个参数循环右移 n 位。但是三个函数的参数和返回值类型不同。

- 函数原型: `void _nop_(void);`

再入属性: reentrant/intrinsicsc

功能: 产生一个 8051 单片机中的空操作指令, 一般用于延时或者等待。

- 函数原型: `bit _testbit_(bit x)`

再入属性: reentrant/intrinsicsc

功能: 产生一个 8051 单片机的位操作指令 JBC。该函数对子节中的一个比特进行测试, 如果该比特为 1 则返回 1, 同时将该比特复位为 0; 否则直接返回 0。

例子:

```
#include <intrins.h>
#include <stdio.h> //提供打印库函数
```

```

void tst_testbit (void) {
    bit test_flag;

    if (_testbit_ (test_flag))
        printf ("Bit was set\n");

    else
        printf ("Bit was clear\n");
}

```

4.2.8 变量参数表 STDARG.H

C51 编译器允许再入的函数的参数个数和类型是可变的，可以使用简略形式（记号为“...”），这时是参数表的长度和参数的数据类型在定义的时候时未知的。头文件 stdarg.h 中定义了处理函数参数表的宏，利用这些宏，程序可以识别和处理变化的参数。

- 函数原型: `typedef char *va_list`

功 能: `va_list` 被定义成指向参数表的指针。

- 函数原型: `type va_arg (argptr, type);`

再入属性: `reentrant`

功 能: `va_arg` 从 `argptr` 指向的参数表中返回类型为 `t` 的当前参数。

例 子:

```

#include <stdarg.h>
#include <stdio.h> //提供打印库函数

int varfunc (char *buf, int id, ...) {
    va_list tag;

    va_start (tag, id);

    if (id == 0) {
        int arg1;
        char *arg2;
        long arg3;

        arg1 = va_arg (tag, int);
        arg2 = va_arg (tag, char *);
        arg3 = va_arg (tag, long);
    }
}

```

```

    else  {
        char *arg1;
        char *arg2;
        long arg3;

        arg1 = va_arg (tag, char *);
        arg2 = va_arg (tag, char *);
        arg3 = va_arg (tag, long);
    }
}

void caller (void)  {
    char tmp_buffer [10];

    varfunc (tmp_buffer, 0, 27, "Test Code", 100L);
    varfunc (tmp_buffer, 1, "Test", "Code", 348L);
}

```

- 函数原型: void va_start (argptr, prevparm);
再入属性: reentrant
功 能: va_start 初始化指向参数的指针。
- 函数原型: void va_end (argptr);
再入属性: reentrant
功 能: va_end 关闭参数表, 结束队可变参数表的访问。

4.2.9 全程跳转 SETJMP.H

该头文件中的函数可以用于正常的系列函数调用和函数结束, 它允许从深层函数调用中直接返回。

- 函数原型: int setjmp (jmp_buf env);
再入属性: reentrant
功 能: 该函数将程序执行的当前环境状态信息存入变量 env 之中, 以便嵌套调用的底层函数使用 longjmp 将执行控制权直接返回到调用 setjmp 语句的下一条语句。当直接调用 setjmp 时返回值为 0, 当从 longjmp 调用时, 返回非 0 值。函数 setjmp 只能在 if 或者 switch 语句中调用一次。
- 函数原型: void longjmp (jmp_buf env, int retval);
再入属性: reentrant
功 能: longjmp 恢复调用 setjmp 时存在 env 中的状态。程序从调用 setjmp 语句的下一条语句执行。参数 val 为调用 setjmp 的返回值。在调用函数 longjmp 后, 由 setjmp 调用的函数中的所有自动变量的值都将被改变。

例 子：

```
#include <setjmp.h>
#include <stdio.h>                                //提供打印库函数

jmp_buf env;                                         // jump 环境(必须为全局变量)
bit error_flag;

void trigger (void)  {
    .
    .
    .
    if (error_flag != 0)  {
        longjmp (env, 1);                            //返回 1 到 setjmp
    }
    .
    .
}

void recover (void)  {
    .
    .
    .
    //恢复代码置于此处
}

void tst_longjmp (void)  {
    .
    .
    .
    if (setjmp (env) != 0)  {                      // setjmp 返回 0
        printf ("LONGJMP called\n");
        recover ();
    }
    else  {
        printf ("SETJMP called\n");
        error_flag = 1;                            //强制一个错误
    }
}
```

```

        trigger ();
    }
}

```

4.2.10 访问 SFR 和 SFR_bit 地址 REGxxx.H

头文件 REGxxx.H 中定义了多种 8051 单片机所有的特殊功能寄存器 (SFR) 名，从而可以简化用户的程序。实际上，用户也可以自己定义相应的头文件。下面是一个采用头文件 reg51.h 的例子：

```

#include <reg51.h>
main()
{
    if (P0==0x10) P1=0x510;           //P0、P1 已经在头文件中被定义
}

```

4.3 C51 程序编写

4.3.1 C 程序基本结构

C 程序共有 3 种基本结构，分别为顺序、选择和循环。一般的程序都是这 3 种基本结构组合的结果。

1. 顺序结构

顺序结构是指从前向后依次执行语句。即先执行语句 A，再执行语句 B，如图 4-1 所示。

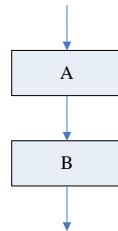


图 4-1 顺序结构

[例 4-1] 顺序结构程序实例

计算表达式 $(x^2 + y^2)^3$ 的值，该程序就是一个纯粹的顺序结构。

```
/* 顺序结构示例 */
```

```
#include <math.h>
#include <stdio.h>

void main()
{
    float x, y;
    float tmp;
    float result;

    x = 3;
    y = 4;
    tmp = x*x+y*y;      // 计算表达式的值
    result = pow(tmp, 3);

    printf("The result is %d\n", result);    // 输出结果
}
```

2. 选择结构

选择结构中，将程序分为两个分支 A 和 B，并提供一个条件 p，当条件 p 成立时，执行语句 A，反之则执行语句 B，如图 4-2 所示。

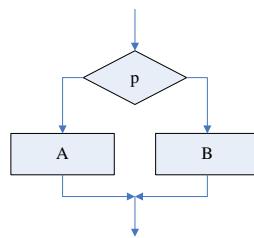


图 4-2 选择结构

C 语言中，有两种语句实现选择分支，分别为 if 和 switch。

[例 4-2] 分枝程序设计实例

```
/* 选择结构示例 */

#include <stdio.h>

void main()
{
    char week;
```

```

printf("Choose a number:"); // 输入数据
scanf("%d", &week);

if((week<0) || (week>7)) // 判断输入是否有效
{
    printf("Error number.\n");
}
else
{
    printf("Your choose is .");
    switch(week) // 根据输入选择要输出的语句
    {
        case 1: printf("Monday.\n");
        break;
        case 2: printf("Tuseday.\n");
        break;
        case 3: printf("Wednesday.\n");
        break;
        case 4: printf("Thursday\n");
        break;
        case 5: printf("Friday.\n");
        break;
        case 6: printf("Saturday.\n");
        break;
        case 7: printf("Sunday.\n");
        break;
    default:
        printf("\n***error***\n");
    }
}
}

```

3. 循环结构

循环结构有两种：当型循环结构、直到型循环结构。前者先进行条件判断，当条件成立时，执行循环，当条件为假时停止循环。后者先执行指令，然后判断条件，如不成立，则停止循环。当型循环结构和直到型循环结构分别如图 4-3 和图 4-4 所示。

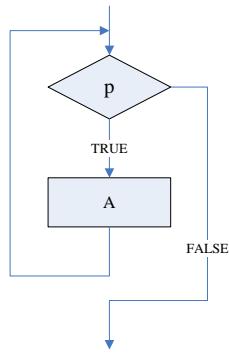


图 4-3 当型循环结构

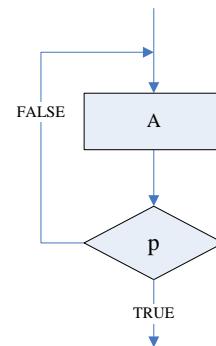


图 4-4 直到型循环结构

[例 4-3] 当型循环结构实例

使用循环计算表达式 $\sum_{n=1}^{10} n! / 10$ 的值，使用结构为当型循环结构。

/ 当型循环结构示例 */*

```

#include <stdio.h>

void main()
{
    long sum = 0;
    long mul = 1;
    char i;
    float result;

    for(i=1; i<=10; i++)      // 循环体
    {
        mul = mul*i;          // 计算阶乘
        sum += mul;            // 求和
    }

    result = sum/10;          // 求均值

    printf("The result is %f", result);
}

```

[例 4-4] 直到型循环结构实例

将输入的字符串中的小写字母转为大写字母，使用直到型循环结构。

/ 直到型循环结构示例 */*

```

#include <stdio.h>

void main()
{
    char ch;
    char buf[256];
    char *p;

    printf("Input a string.\n"); // 读取字符串
    scanf("%s", buf);

    p = buf;
    do
    {
        ch = *p; // 读取字符
        if((ch>'a') && (ch<'z')) // 判断当前字符是否为小写
            ch -= 32;
        p++;
    }while((ch!=0) || ((p-buf)>255)); // 字符串结束或缓冲区溢出则终止

    printf("The result is:\n");
    printf("%s", buf);
}

```

4.3.2 编写高效的 C51 程序及优化程序

帮助读者尽快熟悉 C51 并写出有良好结构和高效率的程序就是本书的目标之一。尽管嵌入式 C 语言是一种强大而方便的开发工具，但开发人员如果要快速编出高效而易于维护的嵌入式系统程序，就必须对 C 语言编程有较为透彻的掌握。不仅如此，优秀的程序员还应该对实际电子硬件系统有深入的理解。因此在学习嵌入式 C 之前，了解汇编语言编程和硬件结构是十分必要的。下面简单介绍一下开发的要点，更具体的内容需要读者在长期的工作中慢慢体会。

1. 良好的、规范化的编程风格

以软件工程的方式进行总体程序设计安排，形成良好的、规范化的编程风格，对高效地编出易于维护的嵌入式系统程序有至关重要的作用，图 4-5 给出了较为合理的工作流程框图。

正规、一致的编程风格十分重要，建议使用 “The C Programming Language” 一书中使用的书写风格，对变量命名采用大部分 Windows 程序员所采纳的 Hungarian 命名法则，再加上模块化的编程方式，详尽的程序说明文档，这样可以促进高效编程的实现，并适应多人分

工协作，也有益于项目开发的规范化以及后期维护。同时，这样写出的程序逻辑清楚，代码简单，注释明白，也便于重新利用。



图 4-5 合理的工作流程框图

2. 灵活选择变量的存储类型，是提高程序运行效率的重要途径

嵌入式系统资源有限，它有各种有限的存储器资源，对需进行位操作的变量，其存储类型为 bdata，直接寻址存储器类型为 data，间接寻址存储器类型为 idata（间接寻址存储器也可访问直接寻址存储器区），外部寻址存储器类型为 xdata，当对不同的存储器类型进行操作时，尽管反映在 C 语言中相同，但是编译后的代码执行效率各不相同，内部存储器中直接寻址空间和间接寻址空间也不相同。

为了提高执行效率，对存储器类型的设定，应该根据以下原则：只要条件满足，尽量先使用内部直接寻址存储器（data），其次设定为间接寻址存储器（idata）。在内部存储器数量不够的情况下，才使用外部存储器。而且在外部存储器中，优先选择 pdata，最后才是 xdata。同时，在内部和外部存储器共同使用的情况下，要合理分配存储器，对经常使用和频繁计算的数据，应该使用内部存储器，其他的则使用外部存储器。要根据它们的数量进行分配，在例程中，内部数据存储器数量充足时，如果将所有变量都声明为直接寻址存储类型，将使程序访问数据存储器所用的时间最少，从而提高程序的运行效率。

3. 合理设置变量类型以及设置运算模式可以减少代码量，提高程序执行效率

由于 51 系列是 8 位机，它只能直接处理 8 位无符号数的运算，而处理其他类型的数据结构则需通过额外的算法来实现，因此，在对变量进行类型设置时，要尽可能选用无符号的字符类型，少用有符号以及多字节类型，因此，程序设计中，都尽量采用无符号数以提高运算速度，以此避免进行多余运算。

在运算时，可以进行定点运算的尽量进行定点运算，避免进行浮点运算。如乘 2 或除 2，就可以使用移位操作来代替。这样不仅可以减少代码量，同时，还能提高程序执行效率。

4. 灵活设置变量，高效利用存储器，提高程序执行效率

嵌入式系统资源有限，而 C 语言编程通常是采用模块化方法，因此在 C 嵌入式编程中，

如何实现高效的数据传输对提高程序执行效率十分关键。

通常，C 程序中，子程序模块中与其他变量无关的变量尽可能使用局部变量，对整个程序都要使用的变量将其设置为全局变量更合适，这样，子程序模块可以直接申明要使用的变量为外部变量即可。

其次要结合 C 语言的特点进行灵活的数据传输。灵活使用 C 语言中所特有的指针、结构、联合等数据类型，可以提高编程效率，也可以方便数据传输。主程序和子程序传递数据量不能过多，变量也要尽量少，否则将影响执行效率。子程序模块和主程序模块都定义了相同的数据类型，在进行数据传输时，主程序模块和子程序模块定义相同的数据类型，在进行数据传输时，只需将指针传送到子程序模块，这样，既可以使不同的程序有很好的独立性和封装性，又能实现不同程序间数据的灵活高效传输。这样做可使不同的开发人员开发独立性很强的通用子程序模块。

5. 将汇编程序嵌入嵌入式 C 程序中，完成实时响应或大运算量的任务

对一些实时性或者运算能力要求很高的程序，如中断程序处理、数据采集程序、实时控制程序以及一些实时的带符号或多位运算，都建议将汇编语言嵌入 C 程序中进行处理。

在 KEILC51 的 C 编译系统中，C 程序能够与汇编程序实现方便灵活的接口，C 程序中调用汇编十分方便灵活，二者之间调用的主要难度在于数据的准确传输。汇编与 C 中实现数据传输可通过两种方式，一是利用工作寄存器进行数据传送，这种方式安全，但根据传送数据类型的不同，只能传送 1~3 个参数。另外一种方案是指定特定的数据区，二者在特定的工作区内进行数据传输，这种方式，其传递数据量可自行控制，但不安全，需要开发人员小心控制。对于特定的编辑器，可以参考其 C 语言的函数编译出来的汇编程序，按照 C 语言函数的格式来写汇编，在 C 中可以直接当作带参数的函数来调用。

6. 利用丰富的库函数，可以大大提高编程效率

在 KEILC51 中，开发厂家提供了许多常用的库函数，主要是数学函数、内存分配等标准函数供编程人员使用。开发人员可灵活的使用这些函数，以提高开发效率。

第5章 Windows集成开发环境μVision2

μVision2是KEIL公司关于8051系列MCU的开发工具，可以用来编译C源码、汇编源程序、连接和重定位目标文件和库文件、创建HEX文件、调试目标程序等。本章首先横向地简单介绍μVision2中各个菜单栏的作用，然后再通过创建程序的流程和调试的流程来详细介绍各菜单的使用以及仿真功能的应用。

5.1 μVision2 编辑界面及其功能介绍

5.1.1 μVision2 界面综述

在Windows下选择开始|程序|Keil μVision2命令，运行Keil μVision2，即出现μVision2的启动界面（如图5-1所示），几秒钟后，主画面中间的版本提示将会消失，由文本编辑窗口（如图5-2所示）所代替。如果上次退出μVision2时没有关闭文件，那么将会恢复显示上次文件的编辑窗口状态，否则要重新打开文件。

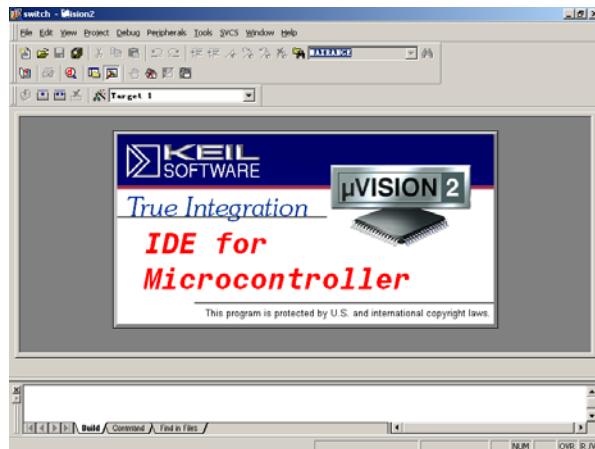


图 5-1 μVision2 的启动界面

μVision2 提供的多功能的文件操作环境，包含一个内藏式编辑器，它是标准的 Windows 文件编辑器，具有十分强大的文件编辑功能，例如文件块的移动、剪切、拷贝、查找、删除等。它支持鼠标操作，也有快捷键。编辑器中不仅有许多预定义热键，而且用户还可以根据自己的喜好对热键进行重定义。μVision2 文件编辑器还提供了一种可选的彩色语句显示功能，对 C51 程序中的变量和关键字等采用不同的颜色显示以提高程序的可读性。

在 μVision2 中，用户可以同时打开多个窗口，对多个不同的文件进行处理，这一特性

有利于使用 C51 进行结构化的多模块程序设计。在模块化编程时，如果同时打开多个不同文件，可以在 μVision2 中对他们分别进行编辑处理。

另外，图 5-2 中 Keil μVision2 的状态栏位于整个界面的下方，实时显示当前命令执行状况、光标所在位置等信息。

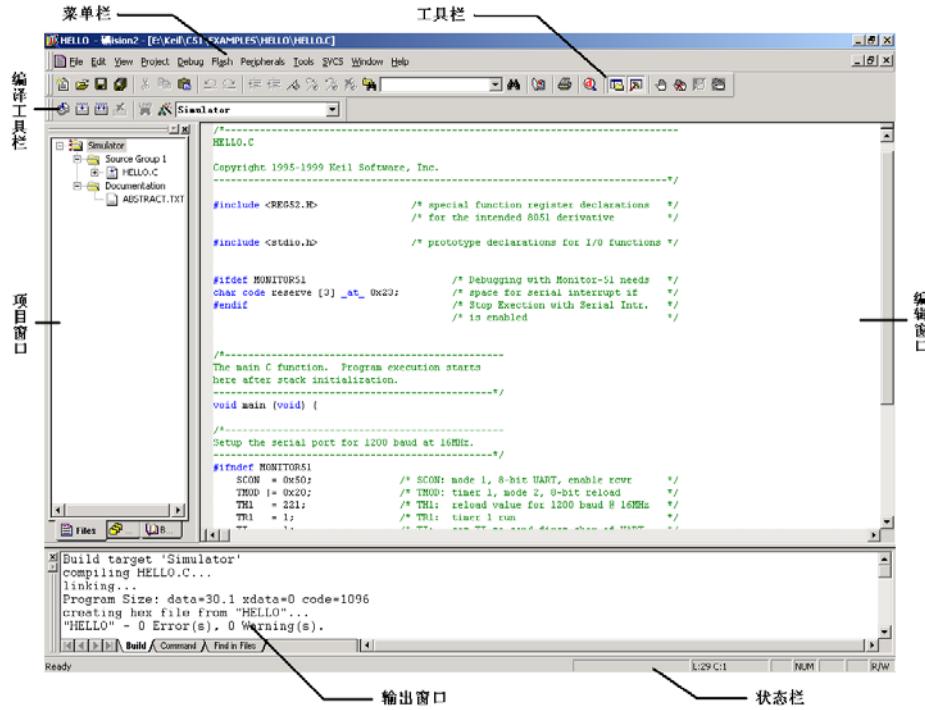


图 5-2 Keil μVision2 文本编辑窗口

5.1.2 主菜单栏

Keil μVision2 的主菜单栏涵盖了几乎所有的 C51 编辑、编译以及调试等功能的实现方式，共有 11 个选项，分别为 File（文件）、Edit（编辑）、View（视图）、Project（项目）、Debug（调试）、Flash（烧写）、Peripherals（外设）、Tools（工具）、SVCS（软件版本控制系统）、Window（窗口）和 Help（帮助）。下面分别对各菜单的列表项所指向的功能进行说明。

File 菜单命令主要用于对文件的一些操作，如新建、打开、关闭、输出等，以及有关打印的操作，如页面设置、打印等。此外，File 菜单中的 Device Database 选项用于修改 μVision2 支持的 8051 芯片型号的设定。Device Database 对话框如图 5-3 所示，用户可以在对话框中添加或修改 μVision2 支持的单片机型号。

Device Database 对话框各个选项功能如下：

- Database 列表框 浏览 μVision2 支持的单片机型号。
- Vendor 文本框 用于设定单片机的类别。
- Family 下拉列表框 用于设定单片机的兼容结构。
- Device 文本框 用于设定单片机的型号。

- Description 列表框 用于设定型号的说明。
- Options 列表框 用于输入支持型号对应的 DLL 文件等信息。
- Add 按钮 单击 Add 按钮添加新的支持型号。
- Updata 按钮 单击 Updata 按钮确认当前修改。

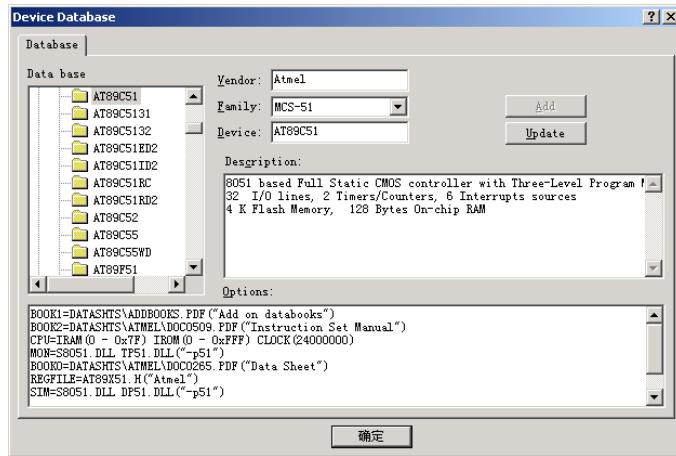


图 5-3 Device Database 对话框

Edit 菜单命令主要包括剪切、复制、粘贴、查找、替换等编辑操作和书签管理命令。单击鼠标将光标移动到欲定位位置，选择 Toggle Bookmark 命令设定书签，以后使用快捷键 F2 和 Shift+F2 可以在不同书签之间跳转。选择 Clear All Bookmarks 清除全部已定义的书签。

View 菜单用于控制 μVision2 的界面显示，使用 View 菜单中的命令可以显示或隐藏 μVision2 的各个窗口和工具栏等。

Project 菜单命令包括项目的建立、打开、关闭、维护、目标环境设定、编译等命令。Project 菜单各个选项功能如下：

- New Project 建立一个新项目。
- Import μVision1 Project 导入 μVision1 中的项目并转换。
- Open Project 打开一个已存在的项目。
- Close Project 关闭当前项目。
- Target Environment 定义工具链、头文件和库文件的路径。
- Target, Group, Files 维护项目的目标、文件组和文件。
- Select Device for Target 为目标选择器件。
- Remove Item 从项目中移除文件或文件组。
- Options 修改目标、组或文件的选项设置。
- Clear Group and File Options 清除对组和文件的选项设定。
- Bulid Target 编译修改过的文件并生成应用程序。
- Rebulid Target 重新编译所有文件并生成应用程序。
- Translate 编译当前文件。
- Stop Build 停止编译。

Debug 菜单命令用于软件仿真环境下的调试，提供断点、单步、跟踪等操作指令。

Flash 菜单主要用于程序下载到 E²PROM 的控制。

Peripherals 菜单是外围模块菜单命令，用于控制芯片的复位和片内功能模块的控制。

Tools 菜单主要用于支持第三方调试系统，包括 Gimpel Software 公司的 PC-Lint 和西门子公司的 Easy-Case。

SVCS 菜单命令用于设置和运行软件版本控制系统 (SVCS, Software Version Control System)。

Window 菜单命令用于设置窗口的排列方式，与 Windows 的窗口管理兼容。

Help 菜单用于提供软件帮助信息和版本说明。

5.1.3 μVision2 功能按钮

1. 工具栏

Keil μVision2 的工具栏如图 5-4 所示。工具栏中包括 μVision2 的一些常用操作。



图 5-4 Keil μVision2 工具栏

工具栏中各个按钮的功能如下：

- 该按钮用于建立新文件，对应菜单命令为 File | New。
- 该按钮用于打开已有文件，对应菜单命令为 File | Open。
- 该按钮用于保存文件，对应菜单命令为 File | Save。
- 该按钮用于保存全部文件，对应菜单命令为 File | Save All。
- 该按钮用于剪切选中文本，对应菜单命令为 Edit | Cut。
- 该按钮用于复制选中文本，对应菜单命令为 Edit | Copy。
- 该按钮用于粘贴选中文本，对应菜单命令为 Edit | Paste。
- 该按钮用于取消上一步编辑操作，对应菜单命令为 Edit | Undo。
- 该按钮用于恢复上一步编辑操作，对应菜单命令为 Edit | Redo。
- 该按钮用于将选中文本右移一个制表符，对应菜单命令为 Edit | Indent Selected Text。
- 该按钮用于将选中文本左移一个制表符，对应菜单命令为 Edit | UnIndent Selected Text。
- 该按钮用于在鼠标光标当前行创建书签，对应菜单命令为 Edit | Toggle Bookmark。
- 该按钮用于将鼠标光标移动到下一个已定义好的书签处，对应菜单命令为 Edit | Goto Next Bookmark。
- 该按钮用于将鼠标光标移动到上一个已定义好的书签处，对应菜单命令为 Edit | Goto Previous Bookmark。
- 该按钮用于清除所有已定义的书签，对应菜单命令为 Edit | Clear All Bookmarks。
- 单击该按钮弹出如图 5-5 所示的 Find in Files 对话框，使用该对话框可以进行多文件查找。 按钮对应的菜单命令为 Edit | Find in Files。

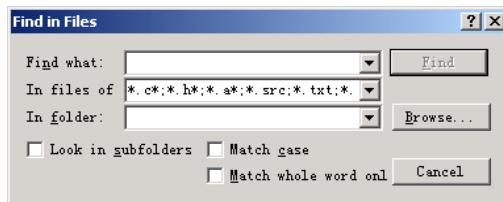


图 5-5 Find in Files 对话框

- 单击该按钮弹出如图 5-6 所示的 Find 对话框，使用该对话框可以在当前文件中查找指定内容。该按钮对应的菜单命令为 Edit | Find。



图 5-6 Find 对话框

- 该按钮用于显示或隐藏源程序浏览器窗口，对应的菜单命令为 View | Source Browser。
- 该按钮用于打印当前文件，对应菜单命令为 File | Print。
- 该按钮用于启动或停止 dScope51 调试模式，对应菜单命令为 Debug | Start/Stop Debug Session。
- 该按钮用于显示或隐藏项目窗口，对应菜单命令为 View | Project Window。
- 该按钮用于显示或隐藏输出窗口，对应菜单命令为 View | Output Window。
- 该按钮用于在鼠标光标所在行建立或删除断点，对应菜单命令为 Debug | Insert /Remove Break Point。
- 该按钮用于清除所有已建立的断点，对应菜单命令为 Debug | Kill All Break Points。
- 该按钮用于允许或禁止当前断点，对应菜单命令为 Debug | Enable/Disable Break Point。
- 该按钮用于禁止所有断点，对应菜单命令为 Debug | Disable All Break Points。

2. 编译工具栏

Keil μVision2 的编译工具栏如图 5-7 所示。该工具栏用于 μVision2 中应用程序的生成和文件的编译等操作。



图 5-7 Keil μVision2 编译工具栏

下面介绍 μVision2 的编译工具栏的各个按钮功能：

- 该按钮用于编译当前文件，对应菜单命令为 Project | Translate。

- 该按钮用于编译修改过的文件并生成应用程序，对应菜单命令为 Project | Build Target。
- 该按钮用于重新编译所有文件并生成应用程序，对应菜单命令为 Project | Rebuild Target。
- 该按钮用于停止正在编译的任务，对应菜单命令为 Project | Stop Build。
- 用于设置选定目标的各个选项。在右侧的下拉列表框中选择要设置的目标名称，单击左侧的 按钮，将打开 Options for Target 对话框，使用该对话框可以对选中目标进行详细的设置。关于该对话框的详细设置，将在讲述建立项目的方法时对其进行介绍。 按钮对应的菜单命令为 Project | Option for Target。

3. 调试工具栏

在调试状态下，Keil μVision2 的调试工具栏位于用户界面的上方，主要用于程序调试方面的控制。Keil μVision2 的调试工具栏如图 5-8 所示。



图 5-8 Keil μVision2 调试工具栏

下面介绍 μVision2 的调试工具栏的各个按钮功能：

- 单击该按钮，可以使程序复位，对应菜单命令为 Peripherals | Reset CPU。
- 单击该按钮，可以使程序运行，对应菜单命令为 Debug | Go。
- 单击该按钮，程序停止运行，对应菜单命令为 Debug | Stop。
- 单击该按钮，执行单步，必要时可进入函数内，对应菜单命令为 Debug | Step。
- 单击该按钮，执行单步，必要时可跳过函数，对应菜单命令为 Debug | Step Over。
- 单击该按钮，执行单步，并跳出当前函数，对应菜单命令为 Debug | Step Out of Current Function。
- 单击该按钮，直接跳至光标所在行，对应菜单命令为 Debug | Run to Cursor Line。
- 单击该按钮，显示当前寄存器的下一步状态。
- 单击该按钮，允许或禁止跟踪记录，用于回顾指令执行，对应菜单命令为 Debug | Enable/Disable Trace Recording。
- 单击该按钮，可以查看跟踪记录，对应菜单命令为 Debug | View Trace Records。
- 单击该按钮，可以显示或隐藏编译后窗口，对应菜单命令为 View | Disassembly Window。
- 单击该按钮，可以显示或隐藏监视窗口，对应菜单命令为 View | Watch & Call Stack Window。
- 单击该按钮，可以显示或隐藏存储器窗口，对应菜单命令为 View | Code Converge Window。
- 单击该按钮，可以显示或隐藏串口输出窗口 1，对应菜单命令为 View | Serial Window #1。

- 单击该按钮，可以显示或隐藏存储器窗口，对应菜单命令为 View | Memory Window。
- 单击该按钮，可以显示或隐藏串口参数分析窗口，对应菜单命令为 View | Performance Analyzer Window。
- 单击该按钮，可以显示或隐藏工具箱，对应菜单命令为 View | Toolbox。

5.1.4 μVision2 窗口环境

本节内容将就 μVision2 开发环境涉及的各功能窗口分别进行介绍。为了便于理解，依照前面图 5-2 中对 μVision2 编辑界面的标注方式，也给出调试状态下 μVision2 的用户界面，如图 5-9 所示。

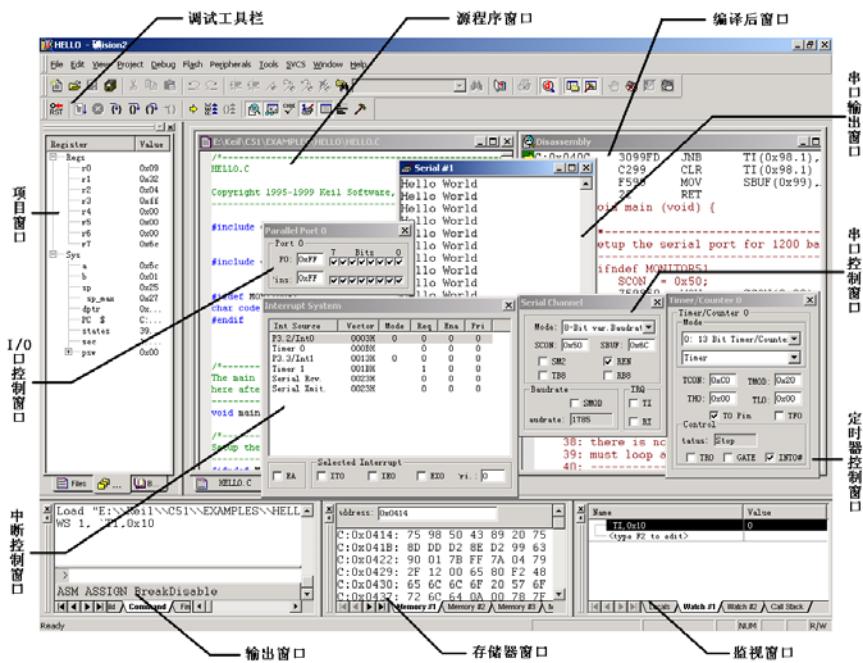


图 5-9 调试状态下 μVision2 用户界面

下面对图 5-2 和 5-9 中的各功能窗口进行依次说明。

1. 编辑窗口

在编辑窗口中，用户可以输入或修改源程序，Keil μVision2 的编辑器支持程序行自动对齐和语法高亮显示。

在编辑窗口中单击鼠标右键，弹出如图 5-10 所示的右键菜单。

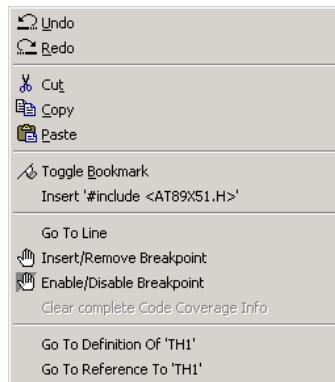


图 5-10 编辑窗口右键菜单

使用该菜单可以实现剪切、粘贴等编辑操作，也可以控制书签和断点的插入和删除。选择 Go To Definition Of 命令，跳转至程序中光标所在关键字的定义部分。选择 Go To Reference To 命令，跳转至程序中光标所在关键字的实现部分。

2. 项目窗口

选择菜单命令 View | Project Window 可以显示或隐藏项目窗口（Project Window）。该窗口主要用于显示当前项目的文件结构和寄存器状态等信息。项目窗口中共有 3 个选项页，分别为 Files、Regs 和 Books。Files 选项页显示当前项目的组织结构，可以在该窗口中直接单击文件名打开文件。进入调试状态后，Regs 选项页显示单片机在当前状态下各个主要寄存器的值，对刚刚改变完状态的寄存器，会以高亮的形式显示。Books 选项页显示帮助文档的树型图，单击标题可以访问相应说明。

图 5-11 和图 5-12 所示为项目窗口中 Files 选项页和 Regs 选项页的内容。

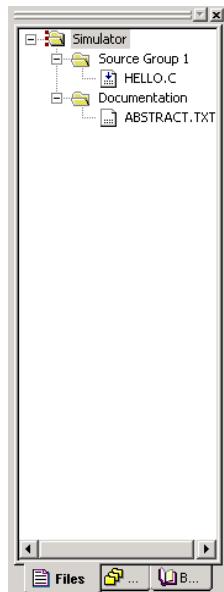


图 5-11 项目窗口中的 Files 选项页

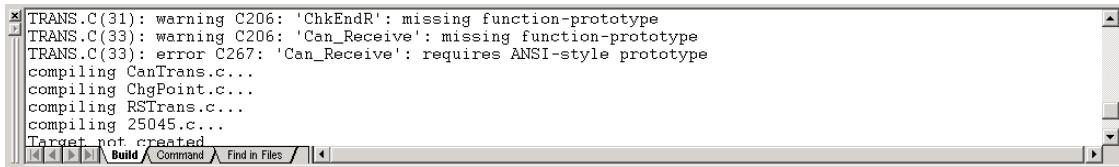
Register	Value
Regs	
r0	0x00
r1	0x00
r2	0x00
r3	0x00
r4	0x00
r5	0x00
r6	0x00
r7	0x00
Sys	
a	0x00
b	0x00
sp	0x21
sp_max	0x21
dptr	0x...
PC	\$ C...
states	389
sec	0....
psw	0x00

图 5-12 项目窗口中的 Regs 选项页

3. 输出窗口

Keil μVision2 的输出窗口 (Output Window) 用于显示 Keil μVision2 本身的调试、命令等信息。该窗口有 3 个选项页，分别为 Build、Command 和 Find in Files。

Build 选项页如图 5-13 所示，用于显示编译时的输出信息。在窗口中双击输出的 Warning 或 Error 信息，可以直接跳转至源程序的警告或错误所在行。



```

TRANS.C(31): warning C206: 'ChkEndR': missing function-prototype
TRANS.C(33): warning C206: 'Can_Receive': missing function-prototype
TRANS.C(33): error C267: 'Can_Receive': requires ANSI-style prototype
compiling CanTrans.c...
compiling ChgPoint.c...
compiling RSTrans.c...
compiling 25045.c...
Target not created

```

图 5-13 Output Window 中的 Build 选项页

Command 选项页如图 5-14 所示，为用户提供命令行模式的调试命令交互环境，用户可以在窗口下方 > 的右侧输入调试命令。



```

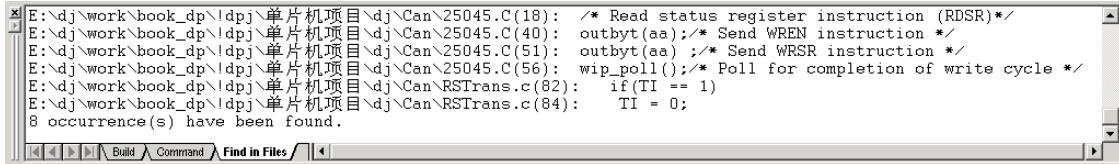
asm
current assembly address = C:0x0000
asm
current assembly address = C:0x0000

>aa =
<C-style expression> variable = <expression>

```

图 5-14 输出窗口中的 Command 选项页

Find in Files 选项页如图 5-15 所示，该选项页用于显示多文件搜索的结果，双击搜索结果可以自动跳转至相应位置。



```

E:\dj\work\book_dp\l dpj\单片机项目\ dj\Can\25045.C(18): /* Read status register instruction (RDSR) */
E:\dj\work\book_dp\l dpj\单片机项目\ dj\Can\25045.C(40): outbyt(aa);/* Send WREN instruction */
E:\dj\work\book_dp\l dpj\单片机项目\ dj\Can\25045.C(51): outbyt(aa);/* Send WRSR instruction */
E:\dj\work\book_dp\l dpj\单片机项目\ dj\Can\25045.C(56): wip_poll();/* Poll for completion of write cycle */
E:\dj\work\book_dp\l dpj\单片机项目\ dj\Can\RSTrans.c(82): if(TI == 1)
E:\dj\work\book_dp\l dpj\单片机项目\ dj\Can\RSTrans.c(84): TI = 0;
8 occurrence(s) have been found.

```

图 5-15 输出窗口中的 Find in Files 选项页

4. 编译后窗口

进入调试模式后，选择菜单命令 View | Disassembly Window，可以显示或隐藏编译后窗口 (Disassembly Window)。编译后窗口中将同时显示 C51 的源程序和相应的汇编程序，如图 5-16 所示。从图中可以看出，左侧有行号的语句为 C51 源程序，左侧为 C: 0x041D 等字样的语句为相应的汇编程序，其中 0x041D 等字样为语句对应的内存地址。

```

23: void main (void) {
24:
25: /*-----
26: Setup the serial port for 1200 baud at 16MHz.
27: -----*/
28: #ifndef MONITOR51
29:     SCON = 0x50;           /* SCON: mode 1, 8-bit UART, enable rcv.
C:0x0414    759850  MOV      SCON(0x98),#0x50
30:     TMOD |= 0x20;        /* TMOD: timer 1, mode 2, 8-bit reload
C:0x0417    438920  ORL      TMOD(0x89),#0x20
31:     TH1 = 221;          /* TH1: reload value for 1200 baud @ 16MHz
C:0x041A    758DDD  MOV      TH1(0x8D),#0xDD
32:     TR1 = 1;            /* TR1: timer 1 run
C:0x041D    D28E    SETB    TR1(0x88.6)
33:     TI = 1;             /* TI: set TI to send first char of UART
34: #endif
35:
36: /*-----
37: Note that an embedded program never exits (because
38: there is no operating system to return to). It
39: must loop and execute forever.
40: -----*/
C:0x041F    D299    SETB    TI(0x98.1)
41:     while (1) {
42:         P1 ^= 0x01;        /* Toggle P1.0 each time we print */
C:0x0421    639001  XRL      P1(0x90),#0x01

```

图 5-16 编译后窗口

使用该窗口可以方便地查看编译后程序的情况，方便程序调试。但是，需要注意在选择了编译后窗口作为激活窗口后，如果单步执行的话，所有的程序将按照 CPU 指令，也就是汇编来单步执行，而不是 C 语言的单步执行。用户可以选择一行程序并设置断点。此时如果要修改 CPU 指令的话，可以使用“Debug”菜单中的“Inline Assembly...”选项。这样就可以更改错误或者对正在调试的目标程序作临时的改动。

另外，在单击 Debug 菜单下的 View Trace Records 选项后，过去执行命令的历史纪录也可以显示在此窗口。而且，要想有历史记录，需单击 Debug 菜单下的 Enable/Disable Trace Recording 选项，来改变 Trace Recording 选项为有效。

5. 存储器窗口

进入调试模式后，选择菜单命令 View | Memory Window，可以显示或隐藏如图 5-17 所示的存储器窗口（Memory Window）。存储器窗口用于显示当前程序内部数据存储器、外部数据存储器以及程序存储器的内容。

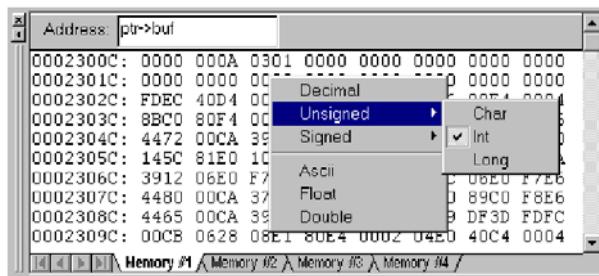


图 5-17 存储器窗口

在窗口中单击鼠标右键，可以在弹出的右键菜单中修改内存内容的显示格式。在上方的 Address 地址框中输入地址，可以显示相应的内存单元的内容。有效的输入方式包括：

- 输入“C: 内存地址”，显示 code 存储区相应地址的内容。
- 输入“D: 内存地址”，显示 data 存储区相应地址的内容。
- 输入“I: 内存地址”，显示 idata 存储区相应地址的内容。
- 输入“X: 内存地址”，显示 xdata 存储区相应地址的内容。

在存储器窗口的地址区域，可以键入所期望显示的内存真实地址的任何表述形式。如果想要改变存储器的内容，双击并在弹出的编辑框中更改为想要的值即可。要想让存储器窗口中的内容实时改变，应将 View 菜单下的 Periodic Window Update 选项选为有效。

6. 监视窗口

进入调试模式后，选择菜单命令 View | Watch & Call Stack Window，可以显示或隐藏如图 5-18 所示的监视窗口（Watch & Call Stack Window）。使用该窗口可以观察和修改程序运行中特定变量或寄存器的状态以及函数调用时的堆栈信息。当程序执行中止时，观察窗口中的内容会自动更新。如果要让目标程序在执行时窗口内容也能更新，则需要将 View 菜单下的“Periodic Window Update”选为有效。

Name	Value
buf	I: 0x40
i	0x00
p	0x0000
temp	0x00

图 5-18 监视窗口

该窗口共有 4 个选项页：

- Locals 该选项页用于显示当前运行状态下的变量信息。
- Watch #1 监视窗口 1，可以单击键盘 F2 添加要监视的变量名称，Keil μVision2 会在程序运行中全程监视该变量的值，如果该变量为局部变量，则运行变量有效范围外的程序时，该变量的值以????形式显示。
- Watch #2 监视窗口 2。
- Call Stack 该选项页用于显示函数调用时的堆栈信息。

除使用快捷键以外，添加监视变量还有3种操作方式：

- 选中文本<enter here>，不移动鼠标，稍后再次单击，就可以进入编辑模式输入变量的名字。按此方式也可以更改变量的内容。
- 在编辑窗口中打开上下文菜单，单击鼠标右键并选择 Add to Watch Window。μVision2 自动选择变量的名字。
- 在“Output Window”窗口的 Command 页可以使用 WatchSet 命令来添加变量名。

要删掉一个正在观察的变量，只要单击该行并使用 Delete 键即可。

7. 串口输出窗口

进入调试模式后，选择菜单命令 View | Serial Window #1，可以显示或隐藏如图 5-19 所示的串口输出窗口（Serial Window #1）。这样的串口窗口共有两个，分别为 Serial Window #1，和 Serial Window #2，用于显示程序到串口的输出信息。



图 5-19 串口输出窗口

需要注意，被仿真的 CPU 串行口数据输出将被显示在这些窗口之中。而输入到串行窗口中的字符将会被输入到仿真的 CPU 中。并且为了在仿真 CPU 的串口时不需要外部的硬件，在“Output Window”窗口的“Command”页中使用 ASSIGN 命令，串行输出也能够被分配到 PC 的 COM 口。

8. 中断控制窗口

进入调试模式后，选择菜单命令 Peripherals | Interrupt，可以显示或隐藏如图 5-20 所示的中断控制窗口。中断控制窗口用于显示和设置 8051 单片机的中断系统，根据单片机型号的不同，中断控制窗口也会有所区别。

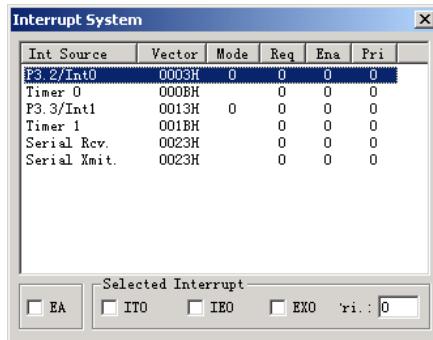


图 5-20 中断控制窗口

9. I/O 口控制窗口

进入调试模式后，选择菜单命令 Peripherals | I/O-Ports，在子菜单中可以选择显示或隐藏指定的 I/O 口控制窗口，如图 5-21 所示。使用该窗口可以查看和控制单片机的 I/O 端口。

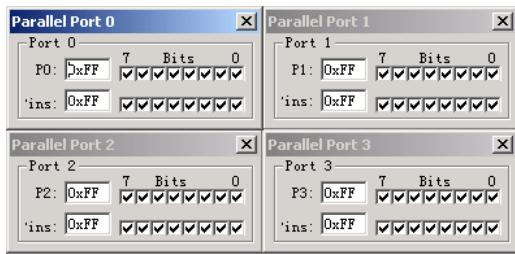


图 5-21 I/O 口控制窗口

10. 串口控制窗口

进入调试模式后，选择菜单命令 Peripherals | Serial，可以显示或隐藏如图 5-22 所示的串口控制窗口。

使用该窗口可以设置串口的工作方式，观察和修改串口相关寄存器的各个位，以及发送、接收缓冲器的内容。在串口控制窗口中，还可以得到串口发送和接收的波特率。



图 5-22 串口控制窗口

11. 定时器控制窗口

进入调试模式后，选择菜单命令 Peripherals | Timer，在子菜单中可以选择显示或隐藏指定的定时器控制窗口，如图 5-23 所示。

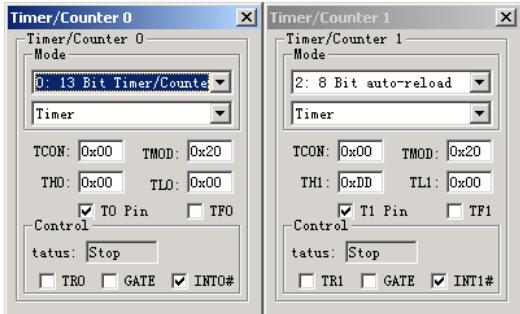


图 5-23 定时器控制窗口

使用该窗口可以设置对应定时/计数器的工作方式，观察和修改定时/计数器相关寄存器

的各个位，以及定时/计数器的当前状态。

5.2 应用 μVision2 开发流程介绍

5.2.1 建立新项目

Keil μVision2 中的项目是一个特殊结构的文件，它包含应用开发系统相关所有文件的相互关系。在 Keil μVision2 中，主要使用项目来进行应用系统的开发。下面介绍在 Keil μVision2 中创建一个新项目的详细步骤：

(1) 选择菜单命令 Project | New Project，弹出如图 5-24 所示的 Create New Project 对话框。

(2) 在对话框中选择新项目要保存的路径和文件名，单击【保存】按钮即可。Keil μVision2 的项目文件的扩展名为.uv2。

(3) 单击【保存】按钮后，弹出如图 5-25 所示的 Select Device for Target 对话框。用户需要在左侧的芯片型号列表中选择调试使用的 8051 单片机型号，使用对话框右侧的 Description 文本框可以查看选中单片机型号的说明。

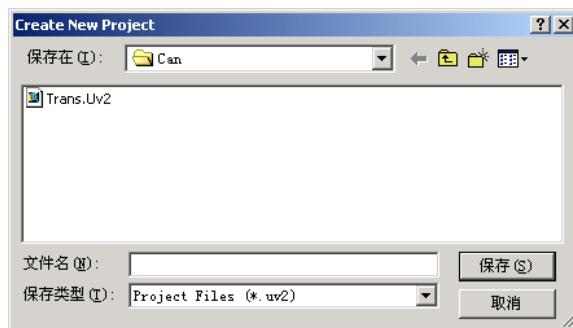


图 5-24 Create New Project 对话框

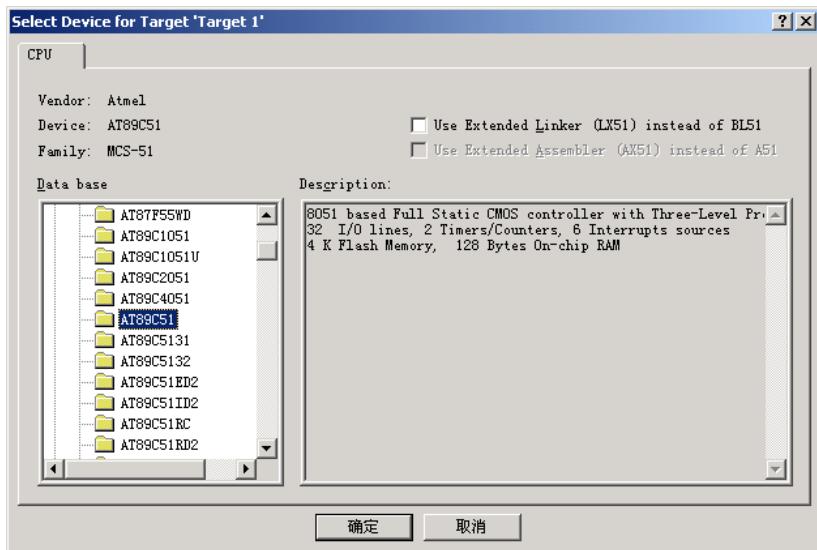


图 5-25 Select Device for Target 对话框

(4) 单击 Select Device for Target 对话框中的【确定】按钮，程序会询问是否将标准 51 初始化程序加入到项目中，如图 5-26 所示。选择【是】，程序会自动拷贝标准 51 初始化程序到项目所在目录并将其加入项目文件。

新建的项目界面如图 5-27 所示。界面的左侧为项目窗口，使用项目窗口可以直观地查看项目中文件的隶属关系。由于目前还没有为项目加入任何文件，因此只能看到创建项目时加入的 51 标准初始化程序 STARTUP.A51。



图 5-26 询问是否加入 51 初始化程序

(5) 下面需要向项目中添加文件。选中项目窗口中的文件组后单击鼠标右键，在弹出的菜单中选择 Add Files to Group 项添加所需文件，如图 5-28 所示。

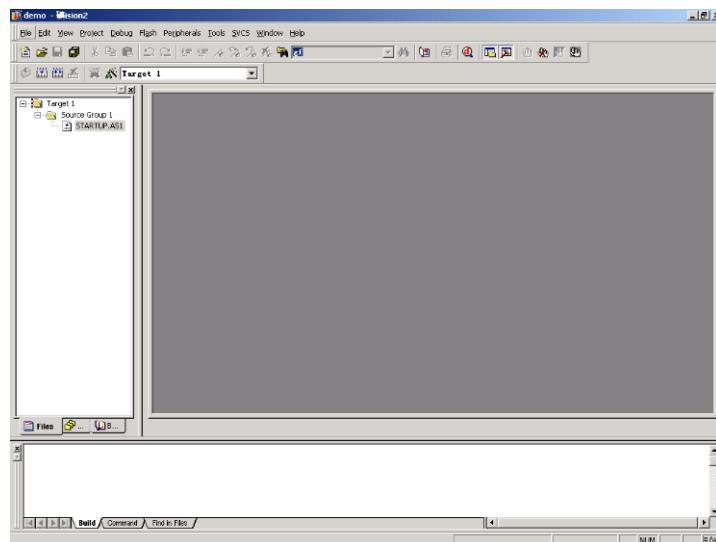


图 5-27 新建项目

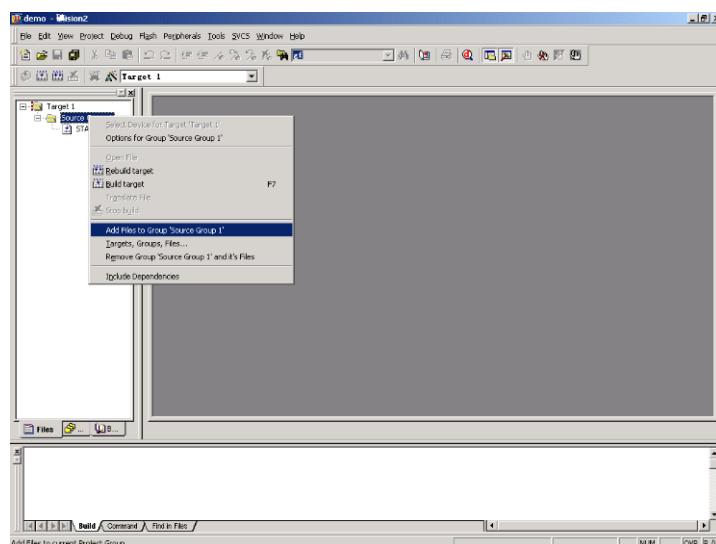


图 5-28 添加文件

添加所有必要的文件后，就可以方便地利用项目菜单对这些文件进行管理，双击选中的文件名可以在编辑窗口中打开该文件。

5.2.2 常用环境配置

项目创建完成后，就可以对项目文件进行编译创建目标文件了。在进行编译和连接前，需要根据样机的硬件环境先在 Keil μVision2 中进行目标配置。

选择菜单命令 Project | Options for Target，弹出 Options for Target 对话框，在该对话框中能够设置所有的工具选项。要设定样机环境，各选项页对应的功能如表 5-1 所示。

表 5-1

Target 对话框中各个选项的含义

对话框选项	含义
Target	为应用程序所适用的硬件环境做设定（前面的目标程序进行工具设定中已经说得很详细，在此不再赘述）
Output	定义输出文件，并且允许用户在编译后就运行用户程序
Listing	指定所有的列表文件，对之进行阐述设定
C51	用来对如代码优化或者变量分配等选项进行配置的编译工具
A51 AX51	对汇编工具的一些功能，如宏编译、预处理等选项进行配置
BL51 Locate L51 Locate LX51 Locate	定义存储器的类别和段。一般来说，用户需要先允许Use Memory Layout from Target Dialog选项
L51 Misc LX51 Misc	其他的一些连接设置，例如警告或者存储器保留等设置
Debug	关于μVision2调试器的一些设置
Properties	文件信息或者对文件、文件组的指定设置项

下面对其中最为常用选项页进行介绍。

1. Device 选项页

Device 选项页如图 5-29 所示。使用该选项页可以修改目标使用的单片机型号，其布局和设定方法与创建项目时选择设备是一样的。

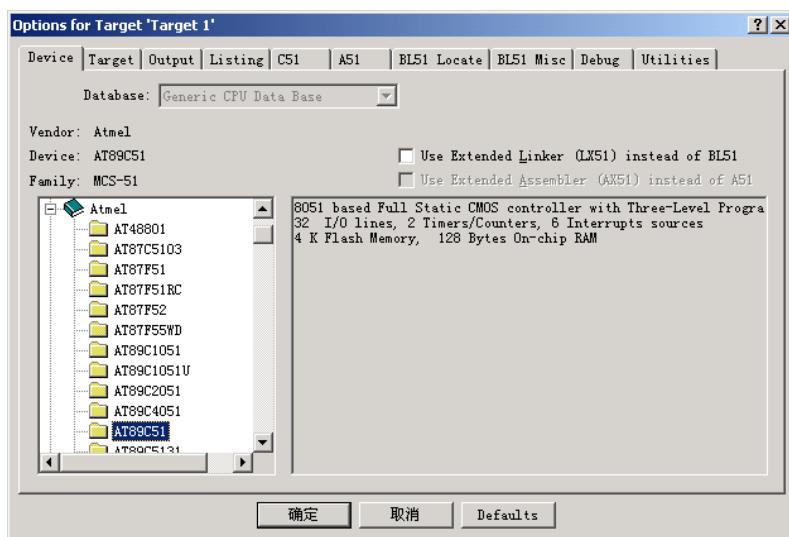


图 5-29 Device 选项页

2. Target 选项页

Target 选项页主要是用于设置目标样机的存储器环境，如图 5-30 所示。

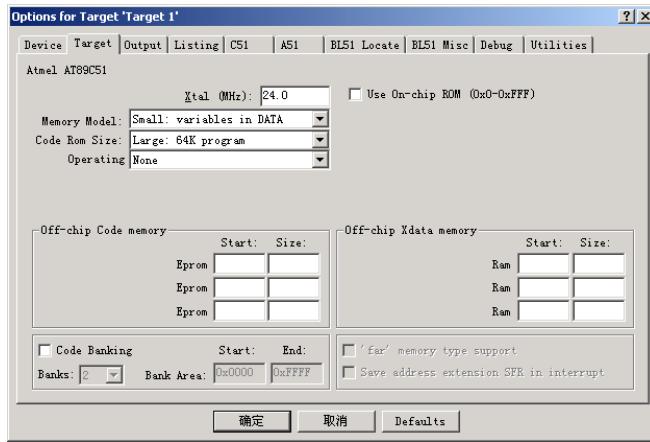


图 5-30 Target 选项页

下面详细介绍该选项页各个选项的功能：

- Xtal 该选项用于设定单片机使用的振荡频率，对图中的 AT89C51，默认振荡频率为 24MHz。
- Use On-chip ROM 勾选该选项，使用片内 ROM。
- Memory Model 该选项用于选择 C51 使用的存储模式。
- Code Rom Size 该选项用于选择程序存储器的大小。
- Operating 该选项用于选择是否需要 RX51 操作系统环境。
- Off-chip Code memory 根据单片机系统硬件的连接情况设置片外 ROM 的起始地址和大小。最多可以设置 3 段不连续的外部程序存储器空间。
- Off-chip Xdata memory 根据单片机系统硬件的连接情况设置片外 RAM 的起始地址和大小。最多可以设置 3 段不连续的外部数据存储器空间。
- Code Banking 勾选该选项，Keil μVision2 使用代码存储体。代码存储体是一种使用代码映射，将最大 2MB 的空间映射到 8051 单片机支持的 64kB 寻址区域的方法。如果选择了该方式，可以使用下方的 Bank Area 文本框设定映射空间的起始和结束地址。

3. Output 选项页

Output 选项页用于设定 Keil μVision2 的输出选项，如图 5-31 所示。

该选项页中一些较为重要的选项功能如下：

- Select Folder for Objects 单击该按钮，可以设定文件输出的路径，默认情况下在当前路径输出编译和连接后文件。
- Name of Executable 该选项用于设定输出文件名。
- Create HEX File 勾选该选项，生成编程器使用 HEX 格式的文件。
- Run User Program 该选项用于设定生成文件后调用的外部程序名称。例如可以在文件生成后调用 Hex2Bin 程序将 HEX 格式文件转为 BIN 文件。Keil μVision2 最多支持调用两个外部程序。

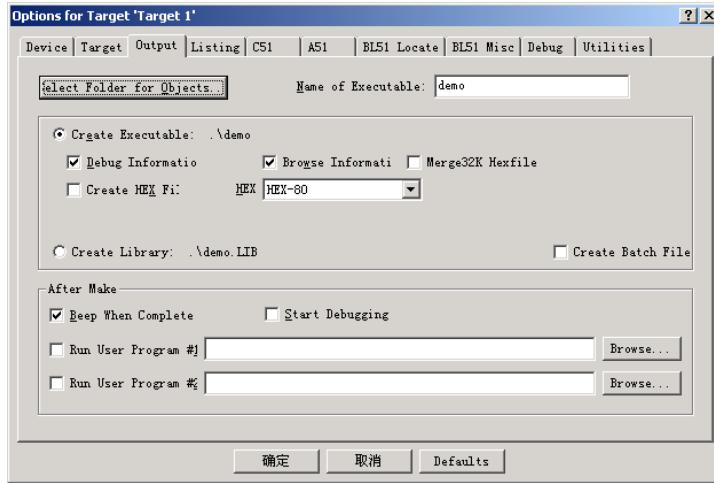


图 5-31 Output 选项页

4. C51 选项页

C51 选项页如图 5-32 所示。

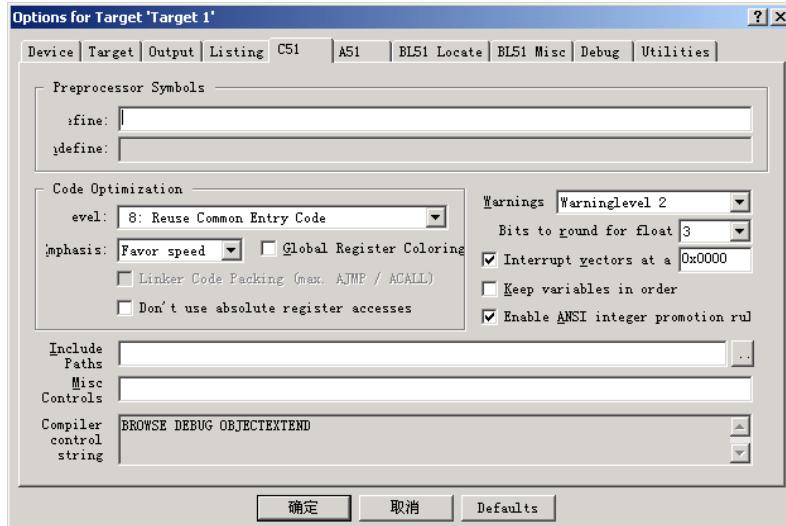


图 5-32 C51 选项页

C51 选项页主要用于 C51 编译器的设定，使用这些选项可以改进代码的质量。该选项页的各个选项功能如下：

- Define 针对预处理符号输入的 C51 DEFINE 指示字。
- Undefine 仅在 Group 和 File Option 对话框中有效，用于消除高层目标或组中设定的 Define 号。
- Code Optimization Level 该选项用于设定 C51 的优化程度，一般情况下应保持默认值不变。
- Code Optimization Emphasis 该选项用于设定代码优化侧重的方向。选择 Favor

size，优化时使用库调用，可以减小代码的大小。选择 Favor speed，代码优化将注重代码执行速度。

- Global Register Coloring 勾选该选项则允许全局寄存器优化。
- Don't use absolute register accesses 勾选该选项，将禁止对 R0~R7 的绝对寄存器寻址，这将增加代码的长度，但可以使代码与选择的寄存器无关。
- Warnings 该选项用于设定 C51 的警告等级，建议选择最高的警告等级。
- Bits to round for float compare 该选项用于设定浮点数比较前保留的位数。
- Interrupt vectors at address 该选项用于设定中断矢量表的基地址。
- Keep variables in order: 该选项用于使编译器按 C 源程序声明变量的先后次序按顺序分配内存。
- Enable ANSI integer promotion rules 勾选该选项，将允许 ANSI 整数扩展，这样会增加代码长度。
- Misc Controls 该选项允许输入特殊的 C51 指示符。
- Compiler control string 该选项用于显示 C51 编译器限制字符串。

5.2.3 代码优化

许多配置参数会影响编译出来的代码的质量。对于大多数应用程序来说，默认的设置选项就能优化代码。不过需要注意的是要改进程序代码长度和执行速度的参数。下面分别介绍各种优化代码工具。

- 内存模式和内存类型

影响代码长度和执行速度最重要的因素是内存模式。内存模式影响了变量的访问方式，参看前面的内存模式部分。存储器模式在“Options for Target”对话框中的“Target”页中更改。

- 全局寄存器优化

KEIL8051工具支持寄存器优化的应用。只要将“Global Register Optimization”复选框选中就可以了。在寄存器优化应用中，C51编译器知道外部设备使用了哪些寄存器。没有被外部函数更改的存储器可以被用来保存寄存器变量。C编译器产生的代码需要更少的数据和程序空间，并且执行速度会更快。为了改进寄存器分配，μVision2编译器会自动重编译C的源程序文件。

还有一些其他的C51编译器功能可以改进代码质量。这些功能在Options-C51对话框中设置为允许或禁止。可以使用不同的编译选项来编译C程序，也可以在列表文件中检验不同配置下编译出来的代码质量。详细说明可参考上一节中对C51选项的介绍。

- 数据类型

8051CPU是一个8bit的微处理器，因此所有变量操作必须按照8bit的数据类型（如char和unsigned char）处理才是最有效率的操作方式。

5.2.4 目标代码调试

设定完项目选项后，选择菜单命令Project | Build Target，就可以直接在编辑环境下

对程序进行编译了，如果在编译连接过程中发现错误，将自动弹出错误窗口并显示出相应的错误信息。用鼠标左键双击错误信息，将自动跳转到产生错误的文件位置，以便可以容易地修改程序文件中的错误。如果编译中没有发生错误，使用菜单命令Debug | Start/Stop Debug Session就可以进入编译状态了。此时用户界面进入调试状态，各子窗口的功能情况可参考本章第1节的内容。依靠“Options for Target”对话框的“Debug”页中的配置内容，μVision2会载入应用程序，并运行初始化代码。μVision2保存了编辑窗口中的内容，并恢复屏幕显示到上次调试的部分。如果程序执行中止，μVision2打开一个显示源程序或者CPU指令的编辑窗口。下一个被执行的指令将会被黄色的箭头标注。

用户可以在编辑窗口中双击为光标所在行添加或删除断点。当需要单步执行查看变量状态时，为了跳过开始的大量初始化代码，可以在程序开始处设定断点，然后选择菜单命令Debug | Run，程序将在断点处停止，此时可以使用单步工具逐步执行代码。

使用界面上的监视窗口可以设定程序中要观察的变量，随时监视其变化，也可以使用存储器窗口观察各个存储区指定地址的内容。

使用 Peripherals 菜单，还可以调用 8051 单片机的各个片内集成模块的控制窗口。使用这些窗口可以实现对单片机硬件资源的完全控制。

在调试过程中，大部分的编译功能仍然可用。例如，可以使用查找命令或者更改程序中的错误。应用程序的源文件也在同一个窗口中显示。不过，μVision2调试程序模式与编辑模式有以下方面不同：

- (1) 可以使用Debug Menu和Debug Commands。详细内容将在后文详述。
- (2) 不能更改项目结构或者工具参数，禁止所有的编译命令。

有关C51目标代码调试仿真的实现，我们将在下一节中详细说明。

5.3 CPU 仿真

在编译和连接完成以后，可以使用μVision2调试器来测试应用程序。μVision2调试器提供了2种操作模式，这些操作模式可以在“Options for Target”对话框中的“Debug”栏中选择，该栏中有两个选项，如下：

Use Simulator: 能够将μVision2调试器配置为纯软件产品，它能够仿真8051系列产品的绝大多数功能而不需要任何硬件目标板。用户可以在硬件板准备好以前就开始调测应用程序。μVision2仿真功能可以仿真各种外设，包括串口、I/O、定时器等。究竟有何外设是在用户为目标板选定一款型号的CPU以后就确定的。

Use Advanced GDI drivers: 比如 KEIL Monitor 51 接口。在高级 GDI 接口程序中，用户可以直接把这个环境与仿真程序或者 KEIL 监控程序相连。本书不讨论这方面的内容。

5.3.1 μVision2 调试器

μVision2仿真器可以仿真高达16MB的存储器，这16MB存储器可以被映射为读、写或者代码执行区间。μVision2仿真器会跟踪存储器访问状态，并报告非法内存访问。

除了存储器映射以外，仿真器还提供了对8051系列各种不同外设的支持。CPU的片上外

设在芯片数据库中选择芯片时就已经进行了配置。在Debug菜单中可以选中和显示片上外设。也可以通过控制对话框中的内容来改变各种外设的值。

1. 设置调试选项

“Options for Target-Debug”对话框中配置μVision2的调试器，如图5-33所示。

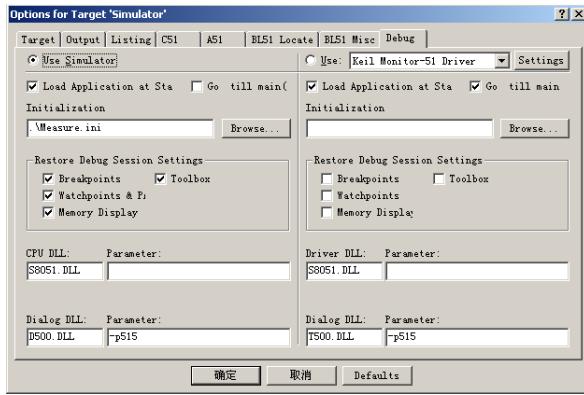


图5-33 配置μVision2的调试器

表5-2详细描述了Debug对话框中各项的含义。

表 5-2 Debug 对话框中各项的含义

对话框选项	含义
Use simulator	将μVision2的仿真器选为Debug模式
Use KEIL Monitor-51 Driver	将高级GDI驱动选为与调试硬件相连接的模式。KEIL Monitor-51的驱动允许用户使用KEIL Monitor直接连接到目标板上。选定该项之后将使用μVision2的仿真器和OCDS驱动
Settings	为选定的高级GDI驱动打开配置对话框
其他的对话框选项分别供高级GDI部分和模拟器使用	
Locat Application at Startup	将该选项选定以后，在启动μVision2调试器时会自动载入目标应用程序
Go till main()	在启动μVision2调试器时，自动执行到main开始处
Initialization File	启动μVision2调试器时，执行命令中输入的若干程序和文件
Breakpoints	恢复上次使用μVision2调试器时的断点设置
Toolbox	恢复上次使用μVision2调试器时的工具箱设置
Watchpoints &PA	恢复上次使用μVision2调试器时的观察断点和性能分析的设置
Memory Display	恢复上次使用μVision2调试器时的存储器显示设置
CPU DLL Driver DLL parameter	为μVision2调试器配置内部的DLL（动态连接库）。这些设置是在设备数据 库中自动设置的。一般不要修改DLL或者DLL的参数

2. 断点

μVision2能够按照若干种方式定义断点。在编辑源程序的过程中，甚至在程序尚未被编译时，用户就可以设置执行断点（Execution Breaks）。不过μVision2的设置断点的功能更加强大。断点主要可以按以下几种方式来设置：

(1) 使用File Toolbar按钮。只要在文本编辑框中选定了所在行或者反汇编窗口选定

了指令位置即可。

(2) 在本地菜单中选择断点。当在文本编辑窗口或者反汇编窗口本地菜单上单击右键的时候，本地菜单即可打开。

(3) 单击Debug菜单下的Breakpoints选项会出现一个对话框，在这个对话框中可以查看定义或者更改断点的设置。对话框可以定义断点有效的情况。下面的例子中将做说明。

(4) 在Output Window窗口的Comand页可以使用BreakSet、BreakKill、BreakList、BreakEnable和BreakDisable命令选项。

在Breakpoint对话框（如图5-34所示）中可以查看和修改断点。只要在Current Breakpoints列表中进行选择，就可以迅速地允许或者禁止断点。双击该列表中的某一项则可对断点属性进行设置。

在Breakpoint对话框中Expression后的文本框中定义一个断点。利用前面所介绍的那些关键字，就可以选定下面的某一种断点类型：

(1) 当表述是一个代码地址时，Execution Break(E)就被定义。当程序运行时，该断点有效。该程序地址必须是CPU指令存放的首地址。

(2) 当可访问的存储器被选择为Access Break(A)时，断点将在处于特定范围的存储器被访问时有效。在定义这块存储器空间时可以使用字节或者目标的长度为单位。Access Break中的表达式必须缩减为内存地址和内存类型。在Access Break中止程序的执行或者其他的操作命令之前，可以使用操作符(&、&&、<、<=、>、>=、==、!=)来做变量的比较。

(3) 当表达式不能被简述为地址时，可以使用Conditional Break(C)。该类型的断点将在特定的条件满足时才会有效。条件表达式在每条CPU指令执行以后都会被重新计算一次，因此，在调试的过程中程序执行速度会变慢。

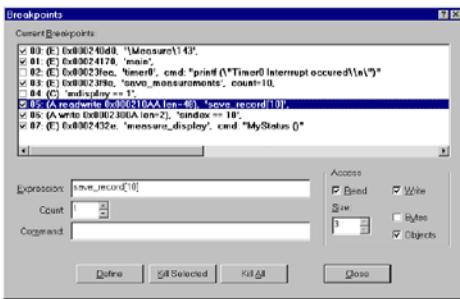


图5-34 Breakpoint窗口

设置完某个断点后，可以在对话框中的Command命令后的文本框中添加相应的指令以便对该断点进行补充说明。μVision2会执行补充的调试命令，并继续执行目标程序。这里的命令可能是一条μVision2调试或者信号函数。在μVision2的函数中，要挂起程序，请更改系统变量_break_。更多的细节参见本章系统变量部分。

从图5-34中可以看到，对话框中还有一个Count值的参数。该值记录了在断点有效之前有多少条符合断点处的条件语句。

下面举几个断点的例子，以详细说明Expression、Count、Command等参数在断点设置中的作用。

- Expression: \Measure\143

Execution Break(E) 会在代码到达MEASURE模块的第143行时挂起程序。

- Expression: \main

Execution Break(E) 会在代码调用主函数时挂起程序。

- Expression: timer() Command: printf("Timer() Interrupt occurred\n")

Execution Break(E) 会在代码到达timer() 函数时将文本Timer() Interrupt occurred\n 打印到Output Window-Command 页。

- Expression: save_measurements Count: 10

Execution Break(E) 会在代码第10次到达函数save_measurements时将目标程序挂起。

- Expression: mcommand==1

Conditional Break(C) 会在表示式mcommand==1为真时将目标程序挂起。

- Expression: save_record[10] Access: Read Write Size: 3 Objects

Access Break(A) 会在程序对save_record[10] 以及其后的2个对象进行读写操作时将目标程序挂起。因为save_record是一个长为16字节的结构，所以这个断点定义的访问区域为48个字节。

- Expression: sindex==10 Access: Write

Access Break(A)，会在程序向变量sindex中写入10的时候将目标程序挂起。

- Expression: measure_display Command: MyStatus()

Execution Break(E)，会在目标程序到达函数measure_display 的时候执行程序MyStatus。当MyStatus函数运行结束以后，目标程序继续执行。

3. 工具箱 (Toolbox)

工具箱有很多用户配置的按钮。单击一个工具箱的按钮就可以执行一条相应的命令。工具箱的按钮可以在任何情况下被执行，即使是在目标程序正在运行的过程中也不例外。

工具箱按钮在Output Window窗口的Command页中，使用DEFINE BUTTON命令来定义。一般来说表达式如下所示：

```
>DEFINE BUTTON "button_label", "command"
```

这里提醒读者注意一下，上面命令行中的“>”号并不是输入的内容，而是作者为了表示该行为输入的命令行而添加的。在本书的其他地方，除非特别说明，否则都按这种格式来表示输入命令。上面命令行的两个参数含义如下。

- button_label: 在工具箱中要显示的按钮的名字。
- command: 在按下该按钮以后需要执行的μVision2 命令。

下面是为工具箱创建按钮的例子，创建按钮结束后工具箱如图5-35所示。

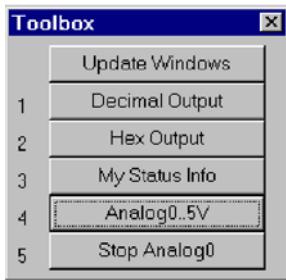


图5-35 创建按钮结束后的工具箱

```
>DEFINE BUTTON "Decimal Output", "radix=0x0A"
>DEFINE BUTTON "Hex Output", "radix=0x10"
>DEFINE BUTTON "My Status Info", "MyStatus ()"          // 调用debug函数
>DEFINE BUTTON "Analog0..5V", "analog0 ()"            //调用 signal 函数
>DEFINE BUTTON "Show R15", "printf (\\"R15=%04XH\\n\\")"
也可以使用KILL BUTTON命令来删掉工具箱按钮中的内容。例如：
>Kill Button 5                                     //删除Show R15按钮
上面用//.....注释掉的内容也是为了读者阅读方便而添加的，并非实际的输入内容。
```

注意：Update Windows按钮在工具箱中自动被创建的，所以不能被删除。Update Windows 按钮的作用是在程序执行的过程中更新若干调试窗口中的内容。

4. 性能分析

μVision2性能分析器（如图5-36所示）显示了指定的函数或者代码区域的执行时间记录。

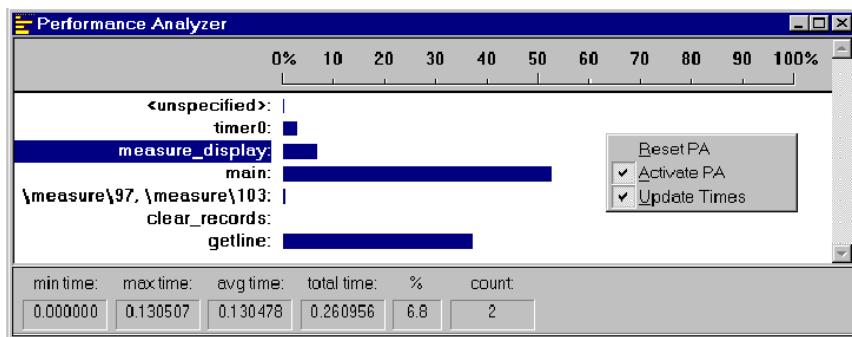


图5-36 μVision2性能分析器

<unspecified>地址范围是自动产生的。它显示了执行没有指定的函数或者地址范围所需要花费的时间。

分析结果用柱状图表示。对于选中的函数或者地址范围，可以显示像激活次数、最短时间、最长时间、平均时间这样的信息。所有的这些统计信息被描述在表5-3中。

如果要设置性能分析器，使用菜单命令Debug-Performance Analyzer。可以在命令窗口

中键入PA命令来设置范围或者要打印的结果。

表 5-3

统计信息

对话框选项	含义
Min time	执行选定的函数或者代码区间的最短时间
Max time	执行选定的函数或者代码区间的最大时间
Avg time	执行选定的函数或者代码区间的平均时间
Total time	执行选定的函数或者代码区间的总时间
%	执行选定的函数或者代码区间的时间占用所有执行时间的百分比
Count	执行选定的函数或者代码区间的总次数

5. 代码覆盖

μ Vision2调试器提供了代码屏蔽的功能。该功能会使某段代码不被执行。在调试窗口中，被执行过的代码行左边被标记为绿色。该功能可以用来测试嵌入式应用程序中还没有被测试过的代码。

Code Coverage对话框提供了统计信息，如图5-37所示。在Output Window窗口的Command页中使用COVERAGE命令可以输出这些信息。

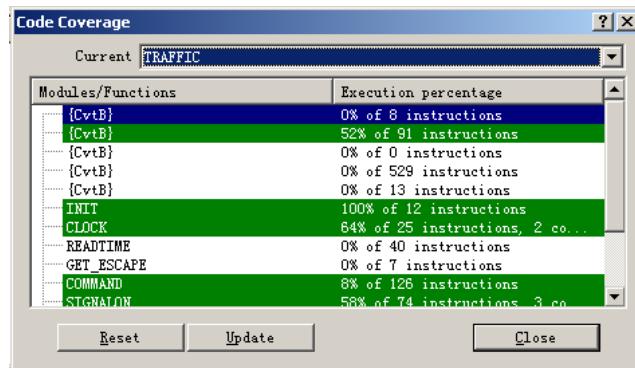


图5-37 信息和统计信息

6. 内存映射

内存映射框能够查看目标程序中用来做数据或程序存储的某一个特定部分。用户也可以使用MAP命令来配置目标程序的内存映射。

如图5-38所示的对话框是通过单击Debug菜单下的Memory Map...选项来打开的。

当装载一个目标应用程序时， μ Vision2自动映射应用程序中的所有地址范围。一般来说没有必要来额外映射地址。用户只需要自己映射没有清除的变量定义的部分，例如内存映射的I/O空间。

当目标程序运行时， μ Vision2使用内存映射来确认程序没有使用无效的地址范围。对每个内存范围，只需要说明访问的方式：读、写、程序执行或者上述三者的某种形式的综合。

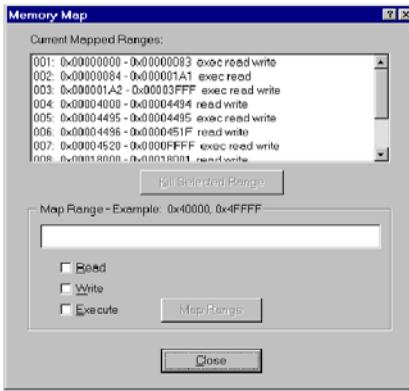


图5-38 内存映射

7. View 变量框

变量框显示了当前载入的应用程序中的公共变量、本地变量、行号信息和CPU的SFR（特殊寄存器），如图5-39所示。

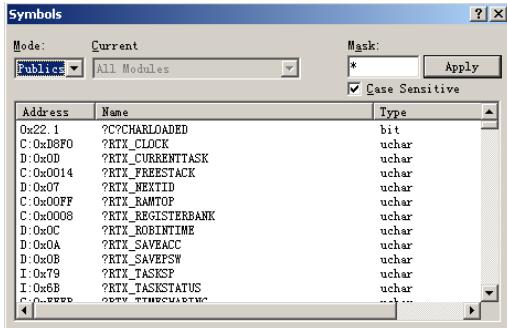


图5-39 变量框

用户可以选择要显示的变量类型，也可以过滤掉显示的变量。表5-4显示了使用该功能的一些选项。

表 5-4

使用该功能的一些选项。

对话框选项	含义
Mode	在该选项中，可以选择显示的变量为PUBLIC、LOCALS或者LINE模式。PUBLIC变量显示模式显示所有的变量和函数，LOCAL显示出现在某个模块或者函数中的变量，LINE则将显示的变量定义到源文件的某一行上
Current module	选择需要显示的在某个模块中的变量
Mask	指定一个与某个变量名匹配的一个屏蔽方式。屏蔽必须由若干字母或者数字加上屏蔽符号组成。屏蔽符号有以下几种： # 使与数字(0-9)匹配 \$ 匹配任何字符 * 匹配零个或者更多的字符
apply	使用屏蔽规则并显示更新后的变量。

表5-5提供了一些对变量名进行屏蔽的例子。

表 5-5

对变量名进行屏蔽的例子

屏蔽符号	屏蔽含义
*	找出所有的变量。为默认的选项
*##	在名称的任何位置有一个数字的变量名
_a\$##	有一处下划线，并且下划线后的字母为a，在后面是任一字符，该字符后为一个数字的变量名。例如，_ab1、_a10value等
_ABC	有一处下划线，下划线后跟有0个或者多个字符，在后面是字符ABC的变量名

5.3.2 调试命令

可以在Output Window窗口中的Command页中通过输入命令来操作μVision2调试器。在下表中列出了所有可用的μVision2调试命令。使用下划线标注出来字母，就可以进行与之对应的操作。例如，键入WS即可进行WATCHSET命令的操作。

在命令入口，语义产生器显示了可能的命令、操作或者参数。当键入命令时，μVision2将会显示一些命令来与键入的字母匹配。

如图5-40所示，如果键入B，语义产生器将减少列出的命令内容。

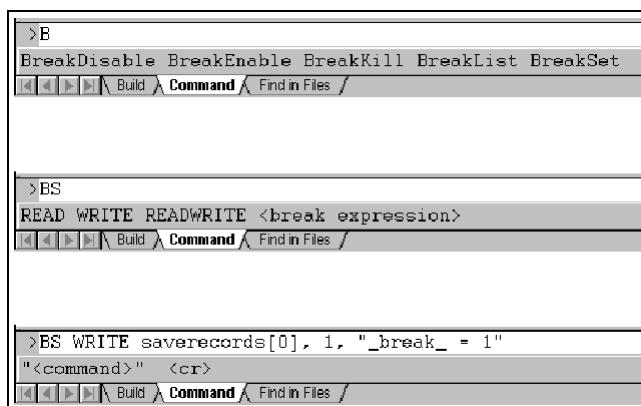


图5-40 调试命令

当使用的命令清楚后，可用的命令选项被列出。语义产生器提供的引导作用可以帮助读者减少错误。

1. 内存命令

表5-6中所示的存储器命令可以用来显示或者更改存储器中的内容。

表 5-6

存储器命令

命令	含义
ASM	编译某行的程序
DEFINE	定义可能在μVision2调试函数中可能要用到的类型或符号
DISPLAY	显示存储器内容
ENTER	将某个值写入规定的存储器空间中
EVALUATE	执行某个表达式，并将结果输出
MAP	对指定的存储器区域设定访问参数
UNASSEMBLE	反编译程序存储器中的内容

WATCHSET	添加一个被观察的变量到观察窗口中
----------	------------------

2. 程序执行命令

程序执行命令用来控制程序的运行状态，例如全速运行程序或者单步执行。表5-7列出了各种程序执行的命令。

表 5-7 各种程序执行的命令

命令	含义
Esc	停止执行程序
GO	开始执行程序
PSTEP	执行到跳过一条指令为止。但是不跳过函数
QSTEP	执行到调处正在执行的函数为止
TSTEP	执行到跳过某条指令或者一个函数为止

3. 断点命令

μVision2提供断点的功能。使用这些断点，就可以使目标程序在某些条件下挂起。断点可以通过读操作、写操作或者运行操作来设定。表5-8列出了各种断点命令。

表 5-8 各种断点命令

命令	含义
BREAKDISABLE	停止执行程序
BREAKENABLE	开始执行程序
BREAKKILL	执行到跳过一条指令为止。但是不跳过函数
BREAKLIST	执行到调处正在执行的函数为止
BREAKSET	执行到跳过某条指令或者一个函数为止

4. 一般命令

表5-9列出的称作一般命令，它们不属于上面的任何一种特殊命令。这些命令是专门为满足调试的某些功能而设。

表 5-9 一般命令

命令	含义
ASSIGN	停止执行程序
COVERAGE	开始执行程序
DEFINE BUTTON	执行到跳过一条指令为止。但是不跳过函数
DIR	执行到调处正在执行的函数为止
EXIT	执行到跳过某条指令或者一个函数为止
INCLUDE	停止执行程序
KILL	开始执行程序
LOAD	执行到跳过一条指令为止。但是不跳过函数
LOG	执行到调处正在执行的函数为止
MODE	执行到跳过某条指令或者一个函数为止
Performance Analyze	停止执行程序
RESET	开始执行程序
SAVE	执行到跳过一条指令为止。但是不跳过函数
SCOPE	执行到调处正在执行的函数为止
SET	执行到跳过某条指令或者一个函数为止
SETMODULE	停止执行程序

SIGNAL	开始执行程序
SLOG	执行到跳过一条指令为止。但是不跳过函数

读者可以通过命令窗口来显示或者改变变量、寄存器和内存中的内容或者位置。例如，表5-10中是一些命令的例子。

表 5-10 一些窗口命令

命令	含义
MDH	显示MDH寄存器
R7=12	将R7的值改写为12
time.hour	显示time这个数据结构中的hour成员
time.hour++	将time这个数据结构中的hour成员的值加1
index=0	将index的值赋为0

5.3.3 存储器空间

8051微控制器提供为变量和程序提供不同的存储器空间。存储器空间被反映在表述的前缀上。各种可用的前缀在表5-11中已经列出。

表 5-11 各种可用的前缀

前缀	存储器空间	说明
B:	BIT	可位寻址区域
C:	CODE	代码区间
Bx:	CODE BANK	代码地址复用区间 (Code bank), x表示bank的序号
D:	DATA	内部直接寻址RAM
I:	IDATA	内部间接寻址的RAM
X:	XDATA	外部数据存储区

注意：前缀并非必要，因为表述的名称通常由一个统一的存储器空间。

下面的例子向读者更详细的说明了这些前缀的含义，后面的//……部分为笔者所添注释：

```
C:0x100          // 代码存储器，地址为0x100
I:100           // 8051的内部RAM，地址为0x64
X:0FFFFH        // 外部数据存储器，地址为0xFFFF
B:0x7F          // 位寻址区，位地址为127，或者2FH.7
B2:0x9000       // 代码地址的第2复用区Code bank 2。地址为0x9000
```

5.3.4 表达式 (Expressions)

许多调试命令接收数字表述作为参数。数字表述是一个数字或者一个复杂的含有数字、调试对象或操作的表述。表5-12中说明了组成表述的构成。

表 5-12 组成表述的构成

组成	含义
位地址	单片机中可以位寻址的数据寄存器
常量	程序中的固定的数值或者字符串
行号	指可执行程序的代码地址。当编译一个程序时，编译器会将行号信息记录在目

	标模块中
操作符	操作符包括+, -, *, /。操作符可以和子表述共同使用
程序变量(符号)	程序变量是目标程序中的变量。它们也常常被称为符号(Symble)或者符号名称(Symbolic names)
系统变量	系统变量会影响μVision2的操作，并可能随着μVision2的操作的进行而改变
类型说明	类型说明用来说明一个数据的类型

1. 常量

μVision2接受十进制常量、十六进制常量、八进制常量、二进制常量、浮点数常量、字符串常量和字符串常量。

- 十进制常量、十六进制常量、八进制常量、二进制常量

默认条件下，数字变量是十进制的。表5-13中是各种进制表述的前缀或者后缀。

表 5-13 各种进制表述的前缀或者后缀

类型	前缀	后缀	例子
二进制	无	Y, y	111111Y.....
十进制	无	T或者不要	1234T, 1234.....
十六进制	0x或者0X	H, h	0x123, 1234H.....
八进制	无	Q, q, o, O	345o, 345Q.....

下面是一些说明：

- (1) 为了方便阅读，数字可用\$符号来断开。例如，1111\$1111y就是11111111y。
- (2) 十六进制变量在第一个数字为A~F时，必须在前面加一个0。
- (3) 缺省来说，数字常量是16bit的。它们可以使用L后缀让它们变成长型的，就是32bit的值。例如：0x1234L、1234L、1255HL。
- (4) 当数字常量以16bit无法表示时，编译器自动将其按照32bit的整数处理。

- 浮点型常量

浮点型常量可以按照以下3种方式中的任意一种来定义：

number. number

number e[+|-]number

number. number[e[+|-]number]

例如，4.12、0.1e3、12.12e-5等。与一般的C语言相比，浮点数必须在小数点前面有一个数字，例如，.12是非法的，而0.12才是合法的。

- 字符常量

字符常量的表述与C语言中的方法完全相同，例如，下面的几种都是合法的字符常量；

'a', '1', '\n', '\v', '\x0FE', '\015'。

表5-14中为一些特殊的字符常量。

表 5-14 特殊的字符常量

组成	含义
\\	\
\”	”
\'	,
\a	响铃

\b	退格
\f	表格反馈
\n	新起一行
\r	回车
\t	制表键
\0nn	八进制常数
\Xnnn	十六进制常数

- 字符串常量

字符串常量的表达方式也与C语言中的方法完全相同，例如：

在某些情况下也许会使用到嵌套的字符串，但是，双引号中不能直接再加入双引号。下面式一些合法的字符串常量：

```
"string\x007\n"
"value of %s = %04XH\n"
```

对比与普通C语言，后续的字符串不能连为一个字符串。例如，“string1+”“string2”不能被连为一个字符串。

- 系统变量

系统变量可以被一些函数访问并且可以在程序变量或者其他表述所在的任何地方使用。表5-15中列出了可用的系统变量、数据类型和它们的用途。

表 5-15 可用的系统变量、数据类型和它们的用途

变量	类型	功能
\$	unsigned long	代表单片机的程序计数器。可以使用\$来显示或者更改程序计数器的内容。例如：\$=C:0x4000命令会将程序计数器的内容设为程序存储器的0x4000地址上
break	unsigned int	该变量允许用户停止执行目标程序。当把_break_设置为一个非零的值时，μVision2会将程序挂起。用户可以在用户函数和信号函数中使用这个变量。这两种函数会在下一节调试函数中详细介绍
ip	unsigned char	显示现在的中断嵌套次数。调试函数可能会需要改变量的值以知道某个中断是否正在进行
states	unsigned long	单片机指令状态计数器的当前值。该计数器从0开始，每执行一条指令就自加1 注意：该计数器为只读的变量
itrace	unsigned int	显示trace(跟踪调试)记录是否在目标程序运行时执行。当itrace为0时，没有trace记录，否则就有调试记录
radix	unsigned int	确定数字输出的格式，可以为10或者16。16是默认值

- 片上外设变量

μVision2在项目选定CPU型号以后就自动定义了外设。定义有两种变量：特殊功能寄存器（SFRs）和CPU管脚寄存器（VTREGs）。

- 特殊功能寄存器（SFRs）

μVision2提供所选择CPU中所有的特殊功能寄存器。SFRs拥有固定的地址，并且可以用简记来表示。

- CPU 管脚寄存器（VTREGs）

CPU管脚寄存器能使用CPU仿真的管脚来确定输入和输出。VTREGs不是公共变量，也不存在于CPU的某个内存地址中。它们可以被用某种表述记法来表示，但是它们的值和使用却是和CPU有关的。VTREGs提供了一种CPU缺省硬件的仿真方式。可以使用DIR VTREG命令来显示它们。

表5-16中列出了所有的VTREG标志。VTREG标志其实是由选定的CPU型号所决定的。

表 5-16

VTREG 标志

标志名	含义
AINx	片上的模拟输入。目标程序可以将写入AINx VTREG中的值读出
CLOCK	CPU内部时钟的频率。该频率决定了CPU每秒钟执行多少个指令。系统变量states和VTREG中的CLOCK一起，被用来计算程序执行的时间（以秒为单位）。CLOCK是只读的，并且是根据Options-Target对话框中的XTAL frequency计算而得
PORTx	单片机上的某个I/O端口上的一组输入输出管脚。例如，PORT2代表P2口上的所有16根或者8根管脚。这些寄存器使用户可以模拟I/O口的输入
SxIN	串行接口x的输入缓存。该寄存器中可以写入8比特或者9比特的数值。这些值被目标程序所读取。用户可以读SxIN来确定输入缓存是否已经准备好一个新的字符。如果该寄存器中的值为0xFFFF，那么就表示过去的值已被读取，此时新值可以被写入
SxOUT	串行接口x的输出。μVision2将应用程序执行以后的串口输出值写入该寄存器中

续表

标志名	含义
SxTIME	定义串行接口x的数据传输率。当SxTIME的值为1时，μVision2模拟时使用应用程序中写入的数据传输率的值。当SxTIME的值为0时（缺省值），应用程序中写入的数据传输率的值会被忽略，此时串行传输是瞬时的
XTAL	单片机的晶振频率。在Options-Target对话框中被定义

例如，C517 CPU的所有VTREG表示如表5-17所示。

表 5-17

C517 CPU 的所有 VTREG

标志名	含义
AIN0	模拟输入腿AIN0（浮点值）
AIN1	模拟输入腿AIN1（浮点值）
AIN2	模拟输入腿AIN2（浮点值）
AIN3	模拟输入腿AIN3（浮点值）
AIN4	模拟输入腿AIN4（浮点值）
AIN5	模拟输入腿AIN5（浮点值）
AIN6	模拟输入腿AIN6（浮点值）
AIN7	模拟输入腿AIN7（浮点值）
AIN8	模拟输入腿AIN8（浮点值）
AIN9	模拟输入腿AIN9（浮点值）
AIN10	模拟输入腿AIN10（浮点值）
AIN11	模拟输入腿AIN11（浮点值）
CLOCK	CPU内部执行指令的时钟频率。
PORT0	I/O端口PORT0（8比特值）
PORT1	I/O端口PORT1（8比特值）
PORT2	I/O端口PORT2（8比特值）
PORT3	I/O端口PORT3（8比特值）
PORT4	I/O端口PORT4（8比特值）
PORT5	I/O端口PORT5（8比特值）
PORT6	I/O端口PORT6（8比特值）
PORT7	I/O端口PORT7（8比特值）
PORT8	I/O端口PORT8（8比特值）
SOIN	串口0的串口输入（9比特值）
SOOUT	串口0的串口输出（9比特值）

S1IN	串口1的串口输入 (9比特值)
S1OUT	串口1的串口输出 (9比特值)
STIME	串口时钟允许
VAGND	模拟参考电压VAGND (浮点值)
VAREF	模拟参考电压VAREF (浮点值)
XTAL	晶振频率

下面的例子显示了如何使用VTREGs来帮助程序的仿真调试。在绝大多数情况下，应使用VTREGs在信号函数中来仿真目标硬件中的某些部分。

(1) I/O端口

μ Vision2为每个I/O端口定义了一个VTREG，例如，PORT2。不要混淆各个端口的VTREGs和SFRs，例如P2。SFRs可以被当作CPU的内存空间来访问，但是VTREGs仅仅只是引脚上的信号。

使用 μ Vision2，仿真外部硬件的输入更为方便。如果有一个脉冲进入一个端口引脚上，可以使用信号函数来仿真这个信号。例如，下面的信号函数输入一个方波，方波的频率为1000Hz。

```
signal void one_thou_hz (void) {
    while (1) {                                //永久循环
        PORT2 |= 1;                            // 将 P1.2 置位
        twatch ((CLOCK / 2) / 2000);           // 延迟 0.0005 s
        PORT2 &= ~1;                           // 将 P1.2 置低
        twatch ((CLOCK / 2) / 2000);           // 延迟 0.0005 s
    }                                         // 重复
}
```

下面的命令使这个信号函数开始运行：

```
one_thou_hz ()
```

读者可以从后文有关使用调试函数的小节中了解更多的内容。

如果某个外部电路的输出与单片机的硬件输出有一定的逻辑关系，要仿真这种电路的逻辑也并不困难。只需要做两步，第一步，写一个 μ Vision2的用户或者信号函数来进行需要的操作，第二步，创建一个断点来激活这个函数。

假定使用一个输出腿(P2.0)来允许或者禁止一个LED。下面的信号函数使用PORT2 VTREG来检查CPU的输出并且显示信息在Command Window中。

```
signal void check_p20 (void) {
    if (PORT2 & 1) {                         //检测 P2.0 管脚
        printf ("LED is ON\n"); }             // 如果是 1，那么 LED 是亮的
    else {                                     //如果是 0，那么 LED 是灭的
        printf ("LED is OFF\n"); }
}
```

现在，如果要为端口1的写端口操作加一个断点，其命令如下：

```
BS WRITE PORT2, 1, "check_p20 ()"
```

现在，不管在什么情况下目标程序对PORT2进行写操作时，check_P20函数都将打印出LED的目前状态。

(2) 串口

片上的串口是被以下几个寄存器来控制的，即SOTIME、SOIN、SOOUT。SOIN和SOOUT代表了CPU的串行输入和输出流。SOTIME用来确定串口的输出是瞬时显示的(STIME=0)还是和特定的数据传输率相关的(STIME=1)。当SOTIME是1时，被显示在Serial window中的串行数据是规定数据传输率下的输出。当SOTIME是0时，Serial window中的串口数据会被显示得更快一些。

仿真串口输入和串口输出一样简单。假定有一个外部的设备周期性的(假定频率为每秒一次)输入特定的数据。那么可以编写一个信号函数来将数据输入CPU的串行口。

```
signal void serial_input (void) {
    while (1) {                                //不断循环
        twatch (CLOCK);                      //延迟1秒
        SOIN = 'A';                           //发送第一个字符
        twatch (CLOCK / 900);                 //延迟1个字符的时间
                                                //900 对于9600的数据传输率来说已经足够
        SOIN = 'B';                           //发送下一个字符
        twatch (CLOCK / 900);
        SOIN = 'C';                           //发送最后一个字符
    }                                         //重复
}
```

当这个信号函数运行时，它每隔一秒，就输入'A'，'B'和'C'到串口，并周期性重复。

串口输出和I/O仿真的方式一样，也要先写一个用户或者信号函数，然后再加入中断，并调用函数。

2. 程序变量(符号)

μVision2能够访问目标程序中的变量或者符号，只要简单地键入它们的名字就可以了。变量的名称或者符号的名称，代表了数值或者地址。符号访问功能使调试程序简单了许多，因为它允许在调试器中使用和程序中相同的名字。

当载入一个目标程序模块时，符号的信息也被载入了调试器。这些符号包括了本地变量(在函数中定义的)、函数名和行号。必须选中Options for Target-Output-Debug Information为有效。如果没有调试器信息，μVision2不能够在源程序级或者符号级的调试。

- 模块名

模块名是组成目标程序的全部或者部分的一个目标模块的名字。源程序级的调试信息和符号的信息都被存在各个模块中。

模块名是根据源文件的名字演变而来的。如果目标程序由一个名为MCOMMAND.C的源文件组成，那么C编译器将生成一个名为MCOMMAND.OBJ的目标文件，该模块的名字就是MCOMMAND。

- 符号命名规则

- (1) 符号不区分大小写：SYMBOL和Symbol是等同的。
- (2) 符号中的第一个字母必须为：'A'~'Z'、'a'~'z'、'_'，或者'?'。
- (3) 符号中随后的字母必须为：'A'~'Z'、'a'~'z'、'0'~'9'、'_'，或者'?'。

注意：当在μVision2中使用运算符“?:”的时候，如果其中的一个变量的前面以?打头，那么必须在运算符和变量之间插入空格。例如：R5=R6? ?symbol:R7。

- 全称符号

全称符号可以通过键入全名来访问，包括模块名和定义变量被定义的函数名。全称符号由以下几个部分组成：

- (1) 模块名：表示符号在哪个模块被定义。
- (2) 行号：显示模块中的指定行产生的代码地址。
- (3) 函数名：显示符号在模块中的那个函数中被定义的。
- (4) 符号名：表示该符号的名称。

这些组成部分可以像表5-18中那样组合。

表 5-18

全称符号

符号组成	含义
\模块名\行号	名为“模块名”的模块中的第“行号”行
\模块名\函数名	名为“模块名”的模块中的名为“函数名”的函数
\模块名\符号名	名为“模块名”的模块中的名为“符号名”的全局符号（变量）
\模块名\函数名\符号名	在名为“模块名”的模块，名为“函数名”的函数中的名为“符号名”的本地符号（变量）

表 5-19

一些例子

符号组成	含义
\MEASURE\clear_records\idx	在MESURE模块中的clear_records函数中的本地符号idx
\MEASURE\MAIN\cmdbuf	在MESURE模块中的MAIN函数中的本地符号cmdbuf
\MEASURE\sindx	在MESURE模块中的符号sindx
\MEASURE\225	在MESURE模块中的225行
\MCOMMAND\82	在MCOMMAND模块中的82行
\MEASURE\TIMER0	在MESURE模块中的TIMER0符号。这个符号可以表示一个函数或者全局变量

- 非全称符号

非全称符号就是那些可以直接键入变量或者函数的符号的名称。这些符号没有全名，当输入这些符号时，程序会在一系列的表中搜索，直到找到一个匹配的名称。搜索的顺序如下所示：

- (1) CPU的寄存器符号：R0-R15、RL0-RH7、DPP0-DPP3。
- (2) 当前函数的本地变量：当前函数由程序寄存器中的值来决定。
- (3) 当前模块中的静态变量：和当前函数相同，当前模块也由程序寄存器中的值来决定。这一条指的是函数外但是在模块内定义的变量。
- (4) 全局或者公共符号：μVision2中定义的SFR符号也被当作公共符号搜索。
- (5) 使用μVision2 DEFINE命令来定义的符号：这些符号用来做调试，不是目标程序的一部分。
- (6) 系统变量：系统变量是用来调试和更改调试器参数的。它们不是目标程序的一部分。
- (7) CPU驱动符号 (VTREGs)：这是由CPU的驱动来定义的。

注意：当创建了用户或者信号函数时，搜索的顺序会有所改变：μVision2先搜索用户或者信号函

数中的变量，然后才按照上面所列出的顺序搜索。

- 直译符号

使用单引号（'）可以得到直译符号。这种符号必须用来访问：

(1) 程序中的变量或者符号。这些符号与预定义的保留字相同。这里所说到的保留字包括μVision2调试命令和选项、数据类型名、CPU寄存器名和编译助记符。

(2) 与程序中变量名同名的CPU的VTREGs。

如果直译符号名已经被给定，则μVision2改变非全称符号搜索的顺序。对直译符号，先搜索的是VTREGs而不是CPU寄存器符号。

下面是几个使用直译变量的例子。

若程序中定义了一个名为R5的变量，并且在调试过程中要去访问它，那么实际上访问的将是CPU的R5寄存器。如果要访问R5变量，必须在变量名称前加单引号。

访问R5寄存器：

```
>R5 = 121
```

访问名为R5的变量：

```
'R5 = 212
```

如果程序中有一个函数名为clock，而现在要访问VTREG中的clock。如下所示：

访问名为clock的函数：

```
>clock
```

如果输出为：0x00000DB2，表示clock函数为0x00000DB2处。

访问VTREG中的clock：

```
'clock
```

如果输出为：20000000，表示VTREG中的clock为20000000。

3. 行号

使用行号允许进行源程序级的调试，并且是针对直接由编译器产生的机器码调试。有了行号功能以后，源程序模块编译会将编译产生的物理地址和源程序对应起来。因为每一个行号对应一个程序地址，所以，μVision2允许在此使用表述。寻找某个行号的语义见表5-20。

表 5-20

某个行号的语义

行号符号	代码地址…
\LineNumber	对行号LineNumber在当前模块中时
\ModuleName\LineNumber	对行号LineNumber在模块ModuleName中时

下面是两个利用行号的例子：

```
\measure\108          // 模块"MEASURE"中的 108 行
\143                 // 当前模块的 143 行
```

4. 位地址

位地址表示内存中的可比特寻址部分。包括特殊寄存器中的那些可寻址比特。位地址的语法为expression, bit_position。

例如：

```
ACC.2 // A 寄存器的第二个比特
0x20.5 // 8051 位存储空间的值
```

5. 几个表述的例子

下面给出几个表述的例子，如果行的前面有“>”号，表示该行为输入到Output Window 的Command页的命令，而下面的一行则表示Output页的输出。其中所有//……中的内容为笔者所添加的注释。

- Constant (常量)

(1) 如果要显示简单的常量，则可以输入如下命令：

```
>0x1234
```

输出为：

```
0x1234
```

(2) 如果要把一个常量用若干进制格式表示出来，则应该输入如下命令：

```
>EVAL 0x1234
```

输出为：

```
4660T 11064Q 1234H '...4'
```

- Register (存储器)

(1) 要查询存储器R1的内容，那么可以输入如下命令：

```
>R1
```

(2) 将R1和R7都赋值成R7-1，那么可以使用下面的命令：

```
>R1 = --R7
```

- Function Symbol (函数变量)

(1) 如果要得到main()函数的起始地址，那么可以输入如下命令：

```
>main
```

若输出为：

```
0x00233DA
```

那么main的起始地址就为0x233DA。

(2) 如果要得到main()函数的起始地址，而不想采用上面的方式，那么可以输入如下命令：

```
>&main
```

若输出为：

```
0x00233DA
```

那么，main的起始地址就为0x233DA。

- 如果想显示起始地址为 main 中的存储器的内容，那么可以输入如下指令：

```
>d main
```

其输出可能如下：

```
0x0233DA: 76 E2 00 04 76 E3 00 04 - 66 E3 FF F7 E6 B6 80 00 v....v....f.....
0x0233EA: E6 B7 00 00 E6 5A 40 00 - E6 D8 11 80 E6 2A 3C F6 .....Z@.....*<
0x0233FA: E6 28 3C F6 E6 CE 44 00 - BF 88 E6 A8 40 00 BB D8 .(<...D.....@..
0x02340A: E6 F8 7A 40 CA 00 CE 39 - E6 F8 18 44 CA 00 CE 39 ..z@...9...D...
```

- Address Utilization Examples (利用地址的例子)

(1) 如果想计算地址, 那么可以使用如下命令行:

```
>&\measure\main\cmdbuf[0] + 10
```

输出可能为:

```
0x23026
```

(2) 想读入代码区间地址为0x233DA处的字节, 那么命令行可以如下:

```
>_RBYTE (0x233DA)
```

输出可能为:

```
0x76 // Reply
```

- Symbol Output Examples (符号输出的例子)

如果需要输出MEASURE模块内main()中的所有符号 (symbol), 那么可以输入如下命令行:

```
>dir \measure\main
```

输出可能为:

```
R14 idx . . . uint
```

```
R13 i . . . uint
```

```
0x0002301C cmdbuf . . . array[15] of char
```

表示总共有3个变量: idx为uint型的; i为uint型的; cmdbuf为长度为15的字符型数组。

- Program Counter Examples (程序指针的例子)

(1) 如果要将程序寄存器的值更改为main(), 那么可以使用如下命令:

```
>$ = main
```

(2) 如果要指向当前的存储器符号 (symbol), 那么接下来只需要使用如下命令:

```
>dir
```

输出为:

```
R14 idx . . . uint
```

```
R13 i . . . uint
```

```
0x0002301C cmdbuf . . . array[15] of char
```

可以看出, 与(5)中的效果相同。

- Program Variable Examples (程序变量的例子)

(1) 如果想查询cmdbuf的指针, 那么可以输入:

```
>cmdbuf
```

可能得到的输出为:

```
0x0002301C
```

(2) 如果想输出cmdbuf中的第一个元素的指针, 那么:

```
>cmdbuf[0]
```

可能的输出为:

```
0x00
```

(3) 如果想输出I的内容, I为一个字节的变量, 那么:

```
>I
```

可能的输出为:

```
0x00
```

(4) 如果想输出idx的内容, idx为大小为两个字节的变量, 那么:

```
>idx
```

可能的输出为:

```
0x0000
```

(5) 如果想将DPP2中的内容复制到index中, 那么可以按如下方式:

```
>idx = DPP2
```

(6) 如果想输出idx的内容, 那么可以:

```
>idx
```

```
0x0008
```

- Line Number Examples (行号的例子)

(1) 如果想得到行号#104处的地址, 那么:

```
>\163
```

可能的输出为:

```
0x000230DA
```

(2) 到模块“MCOMMAND”的91行的地址, 那么:

```
>\MCOMMAND\91
```

可能的输出为:

```
0x000231F6
```

- Operator Examples (操作符的例子)

(1) 果想让R5做自减操作, 那么:

```
>--R5
```

如果R5原先的值为0xFF, 那么输出为:

```
0xFE
```

(2) 如果需要输出一个PUBLIC类型的变量mdisplay, 那么:

```
>mdisplay
```

如果mdisplay的值为0, 那么输出值为:

```
0
```

(3) 如果需要更改该PUBLIC类型的变量, 那么:

```
>mdisplay =1
```

(4) 如果再检查一下结果, 那么可以得到mdisplay现在的值:

```
>mdisplay //检查结果
```

输出为:

```
1
```

- Structure Examples (结构的例子)

假设save_record为一个结构类型的数组。

(1) 想查看一个地址, 那么可以输入:

```
>save_record[0]
```

输出为:

```
0x002100A
```

(2) 如果要该结构的一个成员, 那么请输入:

```
>save_record[0].time.hour = DPP3
```

其中，DPP3为一个变量。

(3) 如果要查询该成员，请输入：

```
>save_record[0].time.hour
```

如果原先DPP3为3，那么结果如下：

```
0x03
```

- Debug Function Invocation Examples (调试函数调用的例子)

(1) 果要调用printf()输出字符串，那么格式如下：

```
>printf ("μVision2 is coming!\n")
```

输出为：μVision2 is coming!

(2) 如果要读/写存储器中的一个字节，那么可以输入如下命令：

```
>_WBYTE(0x20000, _RBYTE(0x20001))
```

(3) 如果要使用输入函数，那么可以输入如下命令：

```
>interval.min = getInt ("enter integer: ");
```

Fully Qualified Symbol Examples (对指定的符号操作的例子)

(4) 如果要自动对指定的变量进行减一操作，那么可以输入如下命令：

```
>--\measure\main\idx
```

若idx变量为2个字节，并且以前为0，现在则为：

```
0xFFFF
```

5.3.5 技巧

下面的内容讨论μVision2的读者在调试中可能用到的一些高级功能。这些功能可能对一般读者来说也许使用得并不多，但是，了解这些功能在将会在某些情况下使调试变得更方便。

1. 模拟 I/O 端口输出

μVision2提供一个对话框来直观的显示各个I/O端口的状态。I/O管脚的值被存在某个VTREG寄存器中，读者可以直接使用信号函数或者断点的功能来访问，也可以直接在对话框中修改它们。I/O的对话框通过Peripherals-I/O-ports打开。下面是个例子：

在用户程序中有如下语句：

```
// 用户 C 程序
p1value = P1;           // 读入端口 1 的输入
P3 = p1value;          // 将读入的值写到端口 3 上
```

则应该对断点做如下定义：

```
>bs write PORT3, 1, "printf (\\"Port3 value=%X\\n\\", PORT3)"
>bs read PORT1, 1, "PORT1 = getInt (\\"Input Port1 value\\")"
```

当执行C程序时，μVision2会打开一个对话框要求输入“端口1”的值，如图5-41所示。在“端口3”输出新的值的时候，Output Window-Command页上会显示一条信息。

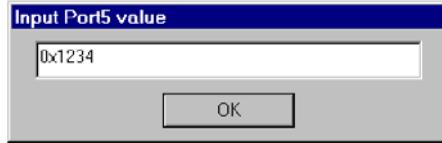


图5-41 Output Window-Command页上显示的信息

2. 模拟中断和时钟输入

μ Vision2可以仿真I/O输入的情况。如果有一条I/O管脚被配置成计数器的输入（如图5-42所示），计数器的值在该条管腿的有效时增加1。下面的例子说明了如何对计数器3的管腿进行模拟输入。

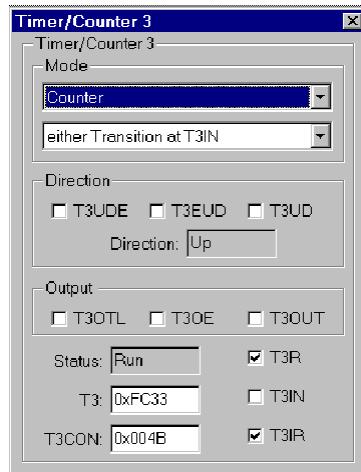


图5-42 模拟中断和时钟输入

```
//用户 C 程序
T3CON = 0x004B; //设置 T3 定时/计数器的工作模式
读者可以通过对VTREG PORT3的设置来使P3.6有效。例如，可以使用信号函数（信号函数的详细内容将会在下一节调试函数中详细说明）：
```

```
signal void ToggleT3Input (void) {
    while (1) {
        PORT3 = PORT3 ^ 0x40;           //改变 P3.6
        twatch (CLOCK / 100000);       //使用 100kHz
    }
}
```

3. 模拟外部 I/O 设备

外部I/O设备一般来说是内存映射的，所以在调试的时候，这种I/O可以通过 μ Vision2提供的Memory窗口来访问。因为C的用户程序并没有对这样的存储器地址做定义，所以需要自己

使用下面的命令对内存进行映射：

```
>MAP X:0x1000, X:0xFFFF READ WRITE // 映射内存为I/O空间
```

读者可以使用断点功能与调试函数结合使用来模仿这些I/O设备的操作逻辑。例如：

```
>BS WRITE 0x100000, 1, "IO_access ()"
```

如果读者对上面的语句不太明白，那也没关系，可以先看后面的调试函数部分，然后回头再看上面的例子就会清楚了。

4. 检查对存储器的非法访问

在某些情况下，需要跟踪对存储器的非法访问。μVision2对中断可以对访问条件进行设置，所以可以通过把中断和系统变量_break_结合起来使用，对寄存器的非法访问过程进行捕捉。在图5-43所显示的例子中，数组save_record如果在函数clear_records以外被访问，那么程序就会被挂起。

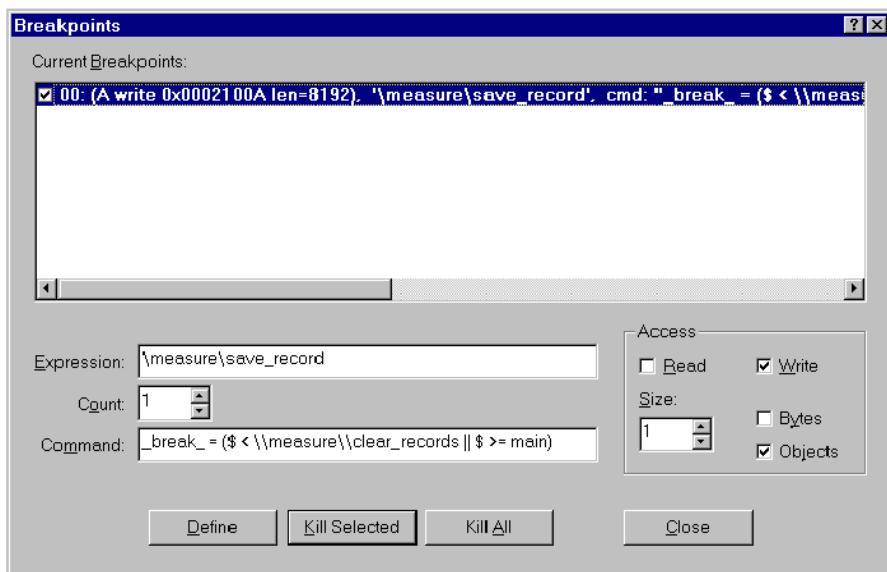


图5-43 检查对存储器的非法访问

5.4 深入了解μVision2

本部分描述了μVision2的build模式，并说明如何使用用户接口来创建一个例程序。此外，本部分还讨论了创建和维护项目的若干选项，包括输出文件选项、C51编译器的代码优化选项、以及μVision2项目管理器的其他特性。

5.4.1 μVision2 的项目管理

1. 项目目标和文件组

使用不同的项目目标 (Project Targets) 可以在μVision2的一个项目中创建若干程序。有时可能需要一个目标 (Target) 作为测试版本，而另一个才是要发布的版本。项目中允许创建多个程序的功能极大地方便了在这种情况下程序的开发。在μVision2中，同一个项目中的不同目标允许有不同的工具设置 (Tool Settings)。

文件组 (File Groups) 在项目内将相关的文件列为一组。这样就能够将文件按功能模块分组，或者辨明在软件工作组中终究是谁编写了哪一部分程序。在这些功能的帮助下，使得维护有几百个文件的复杂项目的工作变得容易许多。

通过Project菜单中的“Targets, Groups, Files”对话框能够创建项目目标或者文件组（上文已经介绍如何使用过这个对话框来添加系统配置文件）。

Project Windows显示了所有的组和相关的文件。文件将会按照窗口中所显示的顺序编译和连接。用户可以使用鼠标拖拉来改变文件的位置。也可以给目标名、组名、文件重命名。单击鼠标右键会出现一个菜单，在这个菜单中可以实现设置工具选项、增加文件到组中、删除条目、打开文件的功能。

在编译工具栏中，可以迅速改变所希望编译的目标。

2. 在 Project Window 中查看文件和组的属性

在Project Window窗口中的File页下使用不同的图标来表示文件和文件组的属性。不同图标的含义如下：

：表示被编译和连接过的文件。

：表示不被连接的文件。在文档文件中常常使用。当然，也可以将源文件程序排除在连接之外，只要将Properties对话框中的Include in Target Build选项禁止掉就可以了。

：表示只读文件。这种应用一般出现在软件版本控制系统中。更详细的内容参阅使用SVCS菜单。

 ：表示由特殊选项的文件或文件组。

3. 源程序浏览器 (Source Browser)

Source Browser显示了程序中的symbols。如果在编译目标程序的时候允许Options for Target对话框中的Output页下的Browser Information选项，那么编译器将会在目标文件中含有浏览的信息。使用View菜单下的Source Browser选项来打开浏览窗口，如图5-44所示。

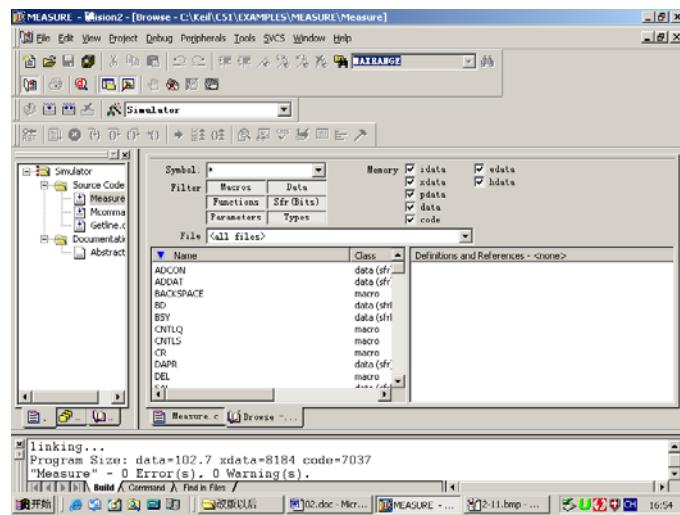


图5-44 使用View-Source Browser选项打开浏览窗口

浏览窗口有变量名称、类、内存空间、使用次数等参数的列表。单击列表条目可以对信息归类。可以按照表5-21中的选项对浏览信息进行过滤。

表 5-21

浏览器选项的含义

浏览器选项	含义
Symbol	定义一个屏蔽格式，用来显示与其相匹配的符号。 此屏蔽格式可以包含实际的字符和以下通配字符： # 匹配数字0~9 \$ 匹配任意字符 * 匹配任意字符串（包括NULL）
Filter On	选择需要显示的符号的类型
File Outline	选择一个文件，显示此文件中的符号资源
Memory Space	定义需要显示的符号的存储类型

表5-22举出了一些对变量过滤的例子。

表 5-22

变量过滤的例子

浏览器选项	含义
*	可代表任何符号。在符号浏览器中是默认的代码
***	可代表有一个数字的符号，数字在符号的任何位置
_a\$##	代表的符号为下划线开头，其后可接字母a、任一字母、任一数字，最后可不加字符也可加任何字符
_*ABC	下划线加任意字符或不加，然后接ABC

单击鼠标右键可以显示浏览窗口的本地菜单。对于函数，还可以查看它们的调用，如图5-45所示和调用者（如图5-46所示）的图示。定义和引用窗口可以提供更多的信息，如表5-23所示。

表 5-23

定义和引用窗口可给的额外信息

浏览器选项	含义
-------	----

[D]	定义处
[R]	引用处
[r]	读操作处
[w]	写操作处
[r/w]	读/写操作处
[&]	地址引用处

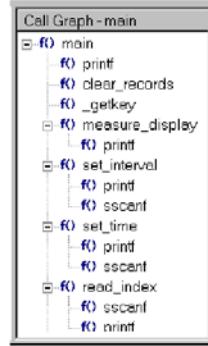


图 5-45 查看函数的调用

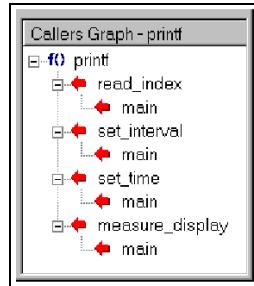


图 5-46 查看函数的调用者

可以在编辑窗口中查看浏览窗口中的信息，方法如下：选定希望查看的条目，然后单击鼠标右键打开本地菜单，或者使用如表5-24所示的快捷键。

表 5-24

快捷键及其功能

快捷键	功能
F12	跳转到定义处；将光标置到符号定义处
Shift+F12	跳转到引用处；将光标置到符号引用处
Ctrl+num+	跳转到下一处的引用或定义处
Ctrl+num-	跳转到前一处的引用或定义处

5.4.2 使用技巧

μVision2 能够提供诸多的功能，掌握这些功能的使用技巧将非常有助于软件的设计开发。

1. 地址复用技术——Code banking

地址复用技术Code banking，可以对现有CPU的程序存储器的寻址空间进行扩展，也就是可以让CPU拥有更多的程序存储器。

标准的8051设备拥有64kB代码地址的程序空间。为了让程序的长度扩展到超过64KB程序存储器空间的限制，KEIL8051工具支持Code banking。该功能让用户能够有32块64KB的程序存储空间，也就是2MB的空间。

例如，硬件设计也许包含有一块32kB ROM的地址空间映射在0000H~7FFFH，另外有4块32kB ROM的地址空间映射在8000H~0FFFFH。Code bank ROM一般通过I/O端口来选定。图5-47显示了存储器结构。

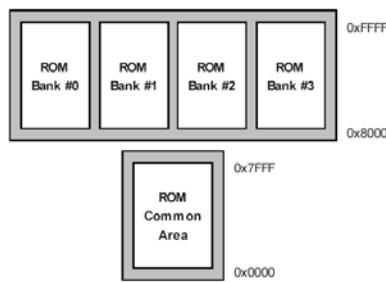


图 5-47 存储器结构

如果需要设定Code banking，那么需要在Options for Target对话框中的Target页中来进行配置。在这里可以输入硬件所支持的Code banks的数目以及bank的地址区域。

如果硬件中使用了banking，那么需要添加C51\LIB\L51_BANK.A51来完成对目标工程的配置。将L51_BANK.A51文件拷贝到项目的目录下，并对该文件进行修改，以使之符合特定的硬件需求。

对项目中的每个文件或者文件组来说，程序代码中的Code bank可在Options for Group对话框中的Properties页中做特殊说明设置。

Options for Group对话框在选定到项目窗口中的文件或者文件组时单击鼠标右键即可出现。这个对话框允许选择使用普通的代码空间还是使用Code bank。

普通的代码空间可以进入所有的Code bank。普通代码空间一般来说包括入口地址和常量等必须常常被访问的内容。例如：中断入口地址、中断和重启向量、字符串常量、bank switch routines等。这样，连接器只将一些模块的程序段定位在bank area。如果能够确保程序只在特定的Code bank部分访问特定地址段的内容，那么就可以使用BANKx连接器来对这些段直接定位，这些设置可以在Options for Target对话框中的L51 Misc页中完成。

上面的步骤能够完成对Code banking应用的配置。μVision2调试器全面支持Code banking，并且支持Code banking的调试。如果在Options for Target对话框中的Output页中选中了Create HEX File项，μVision2就会为每个BANK生成物理上64KB字节的代码段，起始地址均为0。所以在读取程序到PROM上时就有可能需要用编程器对程序进行重定位。

2. 开发工具参数的键序列值表示

一个键序列可以用来从μVision2开发环境向外部用户程序传递参数。键序列可以应用在

工具菜单、SVCS菜单以及用户程序（在Options for Target对话框中的Output页中定义）的参数传递中，一个键序列是键码和文件码的组合。表5-25列出有效的键码，表5-26列出了有效的文件码。

表 5-25

有效的键码

键码	有效的键码
%	定义由文件码所确定的文件的路径
#	包含扩展名的文件名，但是不包含路径（如：PROJECT1.UV2）
%	包含绝对路径的文件名（如：C:\MYPYJECT\PROJECT1.UV2）
@	包含扩展名的文件名，但是不包含路径（如：PROJECT1.UV2）
\$	不包含路径和扩展名的文件名（如：PROJECT1）
\$	文件夹名（如：C:\MYPYJECT）
~	当前光标所在位置的行号（仅仅在文件码为F时有效）
~	当前光标所在位置的列号（仅仅在文件码为F时有效）

注意：键码~和~只在文件码为F时有效。要在用户程序的命令行使用\$、#、%、@、~ 或 ~^、用\$\$、##、%%、@@、~~ 或 ~~^ 格式。如@@在用户程序的命令行上提供一个单独的@字符。

表 5-26

有效的文件码

文件码	有效的文件码
F	定义插入在用户程序命令行上的文件名或参数 在Project Window - Files 页中选定的文件（如：MEASURE.C）。如果选择的是目标名，则返回项目文件；如果选择的是文件组名，则返回当前活动编辑器中的文件
P	当前项目名（如：PROJECT1.UV2）
L	连接器输出文件，通常为调试而生成的可执行文件（如：PROJECT1）

续表

文件码	有效的文件码
H	HEX 应用文件（如：PROJECT1.H86）
X	μ Vision2 可执行文件（如：C:\KEIL\UV2\UV2.EXE）

表5-27是应用在SVCS系统中的文件码，更详细的信息参照“使用SVCS菜单”一节。

表 5-27

应用在 SVCS 系统中的文件码

文件码	应用在 SVCS 系统中的文件码
Q	为SVCS系统保持注释的文件名
R	为SVCS系统保持修正码的字符串
C	为SVCS系统保持校验码的字符串
U	用户名（在SVCS-Configure Version Control - User Name中定义）
V	文件名（在SVCS-Configure Version Control - Database中定义）

注意：Q, R, C, U和V只能和键码%组合使用。

3. 使用 Tools 菜单

通过使用Tools菜单，可以运行外部程序。单击Tools菜单下的Customize Tools menu选项后所就可以产生一个对话框，改对话框能添加定制的程序到Tools菜单中，这个对话框为外部用户应用配置参数。图5-48说明了一个Tools设置的例子。表5-28说明了对话框中选项的含义。

表 5-28

对话框中的选项的含义

浏览器选项	含义
-------	----

Menu Content	工具菜单上显示的文本。其中的每行都可以包括键码和文件码。例如可以使用与号(&)来定义一个快捷键
Prompt for arguments	如果选中，那么在单击此菜单时，一个对话框将弹出，提示输入用户程序的命令行参数
Run minimized	如果选中，将使此应用程序运行时的窗口最小化
Command	输入单击此菜单时运行的程序的路径
Initial Folder	应用程序的当前工作目录。如果为空，μVision2用当前项目所在的目录
Arguments	传递给应用程序的命令行参数

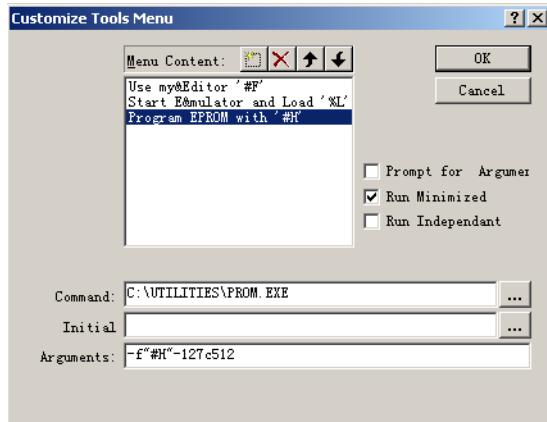


图 5-48 Tool 设置

上面的入口扩展Tools菜单如图5-49所示。



图 5-49 入口扩展 Tools 菜单

命令行在应用程序基础上的输出被存在了一个临时文件中。当应用程序执行结束时，临时文件的内容被显示在“Output Window”窗口的“Build”页面中。

4. 使用 SVCS 菜单

μVision2为软件版本控制系统(SVCS)提供了一个可以配置的接口。通过SVCS菜单(如图5-50所示)，可以调用版本控制系统的命令行工具。SVCS菜单的配置参数被存放在一个模版文件中。这些参数可在单击“SVCS”菜单的“Customize SVCS Menu...”选项后所出现的对话框中配置。表5-29详细解释了这个对话框中选项的含义。

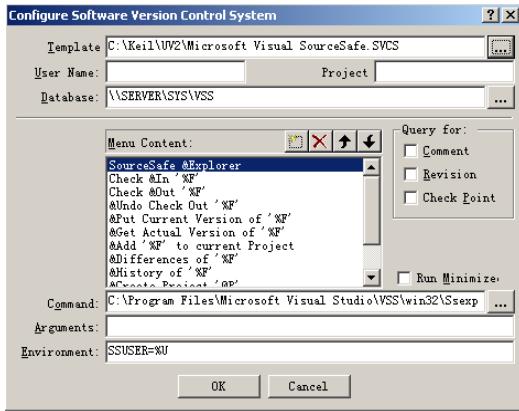


图 5-50 SVCS 菜单

表 5-29

对话框中的选项的含义

对话框选项	含义
Template File	SVCS菜单的配置模板文件。在一个开发组中，推荐所有的软件开发人员选用同样的模板文件。这样，模版文件就可以被拷贝到文件服务器上
User Name	SVCS系统中需要使用的日志用户名。在参数行中，用户名是用%U文件代码来传递的
Database	SVCS系统中使用的数据库的文件名或者路径名。在argument行中，数据库字符串是用%V文件代码来传递的
Menu Content	SVCS菜单中显示的文字。该行可以有关键代码或者文件代码。快捷键被&符号定义。选定的菜单行允许对下面列出的条目进行设定
Query for ... Comment Revision Checkpoint	允许用户使用SVCS查询更多的信息。Comment被拷贝到一个模版文件中，该模版文件可以在argument命令行用SVCS命令使用%Q文件码来传递的。Revision和Checkpoint则需要用%R和%C
Run Minimized	选定该选项后，应用程序执行时是在最小化窗口中
Command	在单击SVCS菜单条目时被激活的程序文件
Arguments	通过SVCS程序文件的命令行参数
Environment	在执行SVCS程序之前设定的环境参量

SVCS应用程序的命令行输出被保存在一个临时文件中。当SVCS命令完成以后，这个临时文件将会在Output Window窗口的Build页被列出。

下面是一个SVCS的例子菜单。在Project Window窗口的Files页中被选定的文件就是SVCS的参数。目标名选择*.UV2项目文件。这个被查看的本地备份就成的一个只读文件，图标上多了一个钥匙的图案。

将μVision2项目保存为2个独立的文件。项目的设置在*.UV2中。该文件需要和SVCS一同查看，而且其中的信息对于重编译一个应用程序来说已经足够了。本地的μVision2配置在*.OPT文件中，它包含了窗口的位置和调试的设置。

表5-30列出了典型的SVCS菜单项。根据设置需要，加入额外的或者不同的选项。包含文件被当作文档文件加入项目中，以加快SVCS访问这些文件的速度。

表 5-30

典型的 SVCS 菜单项

SVCS菜单项	含义
Explorer	打开交互式的SVCS浏览器
Check In	把文件保存到SVCS的数据库中，并把本地文件设置为只读属性

Check Out	从SVCS中取得最新版本的文件，并把本地文件设置为可修改属性
Undo Check Out	取消Check Out操作
Put Current Version	把文件保存到SVCS的数据库中，但对本地文件仍然可以修改
Get Actual Version	从SVCS中取得一个只读文件的最新版本
Add file to Project	添加文件到SVCS项目
Differences, History	显示关于每个特定文件的SVCS信息
Create Project	创建一个和本地μVision2项目文件名相同的SVCS项目

5. 导入μVision2版本1的项目文件

如果要导入版本1的项目文件，请按照以下步骤操作：

- 创建一个新的μVision2项目文件，并从设备数据库中选定一款CPU。
- 单击Project菜单下的Import μVision21 Project选项，选定已经存在于项目文件夹中的旧的μVision21项目文件。
- 把旧的μVision21连接设置导入到连接对话框中。推荐使用μVision2的Options for Target对话框中的Target页来定义目标硬件内存结构。这时应该选中在Options for Target对话框中的L51 Locate页下的Use Memory Layout from Target Dialog复选框，并且去掉对话框中User Classes和User Sections的设置。
- 仔细检查是否所有的设置都被正确地复制到μVision2的项目文件中。
- 现在可以在新的μVision2项目中按照前文所述的方法创建文件组了。

6. 使用μVision2设备数据库中没有的CPU

μVision2设备数据库包含有所有标准的8051产品。但是，也有很多产品没有出现在现在的数据库中。如果需要使用列表中没有的CPU，可以使用如下两种方法：

在Generic中选择一个设备，8051设备允许配置所有的工具参数，这样就可以支持所有的CPU产品了。在Options for Target对话框中Target页下的External Memory一栏中指明片上存储器。

可以键入一个新的CPU资料到μVision2的设备数据库中。单击File菜单下的Device Database选项，即可在对话框里可以选择想使用和修改参数的芯片。在Options对话框中定义了基本的工具设置。具体内容参见前面窗口介绍的章节。

7. 创建一个库文件

在对话框Options for Target的Output页中选择Create Library。μVision2将会调用库管理器而不是连接/定位器。因为库中的代码不会被编译或者定位，L51 Locate 和 L51 Misc 中的设置会被忽略。而且，Target页中的CPU 和内存设置会被忽略。如果想要让代码适用于不同型号的8051芯片的话，在CPU列表下选择Generic。

8. 定位程序段到绝对地址中

有些时候，程序员需要将某些部分的程序定位到程序存储器的某个特定的地址上。在下面的例子中，名为alarm_control的结构需要被定位在0xC000。假设这个结构是在名为ALMCTRL.C的源程序中定义的，并且该文件中也只包含有该结构的定义。

...

```

struct alarm_st {
    unsigned int alarm_number;
    unsigned char enable_flag;
    unsigned int time_delay;
    unsigned char status;
};

struct alarm_st xdata alarm_control;
...

```

C51编译器为ALAMCTRL.C生成一个目标文件，并且包含了一个在XDADA区域专门分配给变量的段。 μ Vision2能够对任何段的基地址进行定位，只要在Options for Target对话框中的L51 Locate页中对存储空间地址做相应的设置就可以了，如图5-51所示。

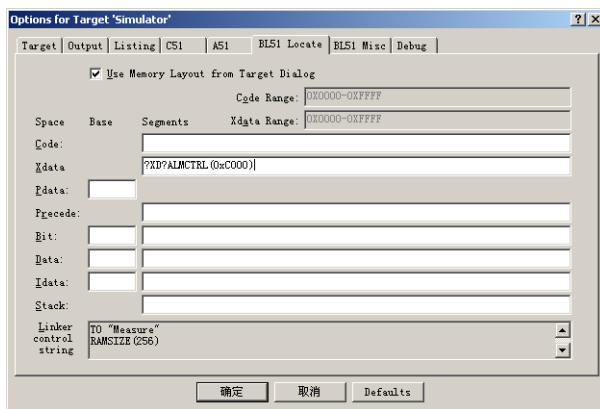


图 5-51 定位基地址

注意：C51也可以用宏的方式来访问绝对地址。

5.4.3 μ Vision2 调试函数

本部分讨论 μ Vision2的一个功能强大的特点：调试函数。这些函数能用来扩展 μ Vision2的调试能力。使用调试函数，可以产生外部中断、记录存储器内容到日志文件、周期性地更新模拟输入的值、输入串口数据到CPU等。

注意：不要混淆了 μ Vision2的调试函数和目标程序中的函数。前者是为了完成应用程序的调试而设，并且是在Function Editor或者 μ Vision2的命令行中输入的。

μ Vision2调试函数使用了C编程语言的一个子集。基本的功能和限定如下：

- (1) 跳转和条件语句if、else、while、do、switch、case、break、continue和goto都可以在调试函数中使用。所有这些操作方法都和ANSI C中的使用方法相同。
- (2) 本地scalar变量在调试函数中的定义和ANSI C中的定义相同。但是在调试函数中是不可以使用数组的。

1. 创建函数

μVision2有一个调试函数编辑器（如图5-52所示），该编辑器可以通过单击Debug菜单下的Function Editor选项来打开。打开函数编辑器以后，该编辑器就会要求输入一个文件名。对调试函数的指定也可以单击Options for Target对话框中的Debug页下的Initialization File按钮后设定。调试函数编辑器和μVision2自带的编辑器的工作方式一样，并可以编译函数。

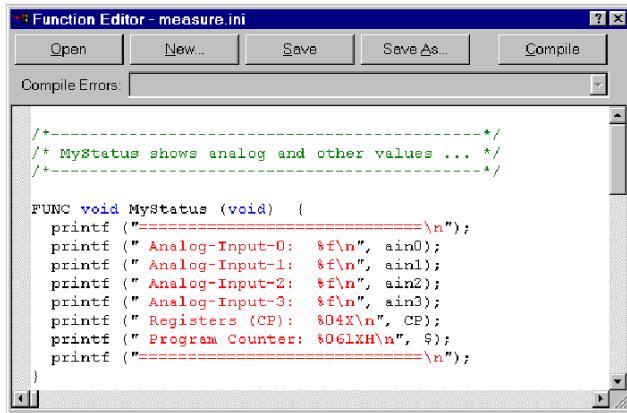


图 5-52 调试函数编辑器

函数编辑器对话框中的各个按钮功能如表5-31所示。

表 5-31 函数编辑器对话框中的各个按钮功能

按钮	功能
Open	打开一个已经存在的μVisioan2调试函数或者命令
New	新建一个文件
Save	将编辑窗口中的内容保存
Save as	将编辑窗口中的内容另存为一个文件
Compile	使用μVisioan2的命令解释器来编译现有编辑窗口中的内容。该功能编译窗口中的所有调试函数
Compile Errors	显示错误列表。选择一个错误，光标就会定位到编辑窗口的错误位置上

一旦使用μVision2调试函数创建了一个文件，就可以使用INCLUDE命令来读取或者执行该文本的内容。例如，如果在命令窗口键入了下面的命令，μVision2读取并解释MYFUNCS. INI文件中的内容。

```
>INCLUDE MYFUNCS. INI
```

MYFUNCS. INI可以包括调试命令和函数定义。在Options for Target对话框中的Debug页下的Initialization File指定的文本框中输入这个文件名。在每次μVision2调试器开始工作的时候，MYFUNCS. INI文件都将会被执行。不再需要被用到的函数可以使用KILL命令来删除。

2. 激活函数

如果在命令窗口中键入函数的名字以及其他必要的参数，就可以激活或者运行一个调试函数。例如，运行printf自带的函数来打印“Hello World”，在命令窗口中键入下面的内容：

```
>printf ("Hello World\n")
```

μ Vision2调试器将会在Output Window的Command 页打印出“Hello World”的文本以对上述命令做出响应。

3. 函数分类

μ Vision2支持以下3种函数类：预定义的函数，用户函数，信号函数。三种函数的特点如下：

预定义函数：提供有用的工具，如等待一段时间或者打印一条信息。预定义的函数不能被删除或者重定义。

用户函数：扩展了 μ Vision2的功能。能够使用命令级的表述。预定义的函数exec可以被用来执行调试用户函数或者信号函数中的命令。

信号函数：仿真复杂的信号发生器的行为，并能为目标程序产生各种输入信号。例如，信号可以被输入到CPU的各个管脚上。当程序在进行仿真执行时，信号函数在后台运行。 μ Vision2最多支持64个信号函数的同时运行。

函数被定义后，会被记录到内部的用户或者信号函数的内部的表中。如果要列出所有的预定义的、用户的和信号函数，可以使用DIR命令来完成。例如：DIR BFUNC显示所有的自带的函数。DIR UFUNC显示所有的用户函数。DIR SIGNAL显示了所有信号函数的名字。DIR FUNC显示所有用户、信号和自带的函数。

(1) 预定义的函数

μ Vision2包括一些预先定义的函数，用来实现一些调试功能。这些函数不能被删除或者重定义。预定义的函数一般用来更好的实现用户或者信号函数的功能。表5-32中列出了所有的预定义的 μ Vision2的调试函数。

表 5-32 预定义的 μ Vision2 的调试函数

返回类型	函数名	参数	说明
void	exec	(“command_string”)	执行调试命令
double	Getdbl	(“prompt_string”)	要求用户输入一个double型的数值
int	getint	(“prompt_string”)	要求用户输入一个int型的数值
long	getlong	(“prompt_string”)	要求用户输入一个long型的数值
void	memset	(start_addr, len, value)	将某段内存填充为固定值

续表

返回类型	函数名	参数	说明
Void	printf	(“string”, …)	与ANSI C的同名函数的功能和用法相同
int	rand	(int seed)	返回一个随机整数，随机整数的范围在-32768~32767
void	rwatch	(address)	延迟信号函数的执行直到某块指定的内存被应用程序进行读操作
void	wwatch	(address)	延迟信号函数的执行直到某块指定的内存被应用程序进行写操作
void	Swatch	(ulong states)	延迟信号函数的执行直到延时规定

			的时间（以秒为单位）以后
void	twatch	(float seconds)	延迟信号函数的执行直到CPU经过了规定的指令状态数以后
int	_TaskRunning_	(ulong func_address)	检查是否指定的函数在执行某个任务。这个命令只有在DLL或者RTX运行时才有效
uchar	_RBYTE	(address)	从指定的存储器中读取一个存储空间大小为char的值
uint	_RWORD	(address)	从指定的存储器中读取一个存储空间大小为int的值
ulong	_RDWORD	(address)	从指定的存储器中读取一个存储空间大小为long的值
float	_RFLOAT	(address)	从指定的存储器中读取一个存储空间大小为float的值
double	_RDOUBLE	(address)	从指定的存储器中读取一个存储空间大小为double的值
void	_WBYTE	(address, uchar val)	将指定值写入存储器某处的大小为char的地址空间
void	_WWORD	(address, uint val)	将指定值写入存储器某处的大小为int的地址空间
void	_WDWORD	(address, ulong val)	将指定值写入存储器某处的大小为long的地址空间
void	_WFLOAT	(address, float val)	将指定值写入存储器某处的大小为float的地址空间
void	_WDOUBLE	(address, double val)	将指定值写入存储器某处的大小为double的地址空间

下面是对这些预定义的函数的详细说明：

```
void exec ("command_string")
```

exec函数能够在用户或者信号函数中激活μVision2调试命令。Command_string可以是多个被分号分开的命令。

Command_string必须是有效的调试命令。例如：

```
>exec ("DIR PUBLIC; EVAL R7")
>exec ("BS timer0")
>exec ("BK *)
```

```
Double getdbl ("prompt_string")
int getInt ("prompt_string")
long getlong ("prompt_string")
```

这些函数提示用户键入一个数字，并且返回键入的数值。如果没有输入值，返回值为0。例如：

```
>age = getInt ("Enter Your Age")
Void memset (startaddress, ulong length, uchar value)
```

这个函数将起点为startaddress，长度为length的内存块写入固定值。该数值为value。例如：

```
>MEMSET (0x20000, 0x1000, 'a') //将从0x20000开始，到x20FFF为止的所有内存填为“a”
```

```
Void printf ("format_string", ...)
```

该函数和ANSI C中的同名函数的工作方式相同。第一个参数为某种格式的字符串，后面的若干参数可以是表达式或者字符串。例如：

```
>printf ("random number = %04X\n", rand(0))
random number = 1014H
>printf ("random number = %04X\n", rand(0))
random number = 64D6H
>printf ("%s for %d\n", "uVision2", 8051)
μVision2 for 8051
>printf ("%lu\n", (ulong) -1)
4294967295
```

上面的文字中，“>”后的表示输入的命令行，没有“>”的表示为命令执行后的输出。

```
Int rand (int seed)
```

该函数产生一个范围在-32768~+32767的随机数。随机数产生器在每次使用时会根据输入的非零种子值被重新初始化。这个函数可以用来产生一个随机的时钟周期，或者来产生随机的数据以适应特定的逻辑或者输入。

```
>rand (0x1234) //使用 0x1234 产生随机数
0x3B98
>rand (0) //种子为 0 的随机数
0x64BD
```

```
Void twatch (long states)
```

该函数可以在特定的CPU状态下延迟程序的连续的执行。 μ Vision2在执行目标程序的时候会不断的更新状态计数器。

请读者阅读下面的信号函数代码，注意twatch的用法。

```
signal void int0_signal (void) {
    while (1) {
        PORT3 |= 0x04;                      //将 INT0(P3.2) 置为高
        PORT3 &= ~0x04;                     //将 INT0(P3.2) 置为低，并产生中断
        PORT3 |= 0x04;                      //再次将 INT0(P3.2) 置为高
        twatch (CLOCK);                   //等待 1 秒
    }
}
```

该信号函数会让INT0输入（P3.2）每秒钟有效一次。

注意：twatch函数只能在信号函数内被调用。在信号函数外调用是不允许的并且会返回错误信息。

```
Void swatch (float seconds)
```

swatch函数可以用在信号函数中用来延迟若干秒连续的程序执行。

例如：下面的信号函数会让INT0输入腿（P3.2）每半秒钟有效一次。

```
signal void int0_signal (void) {
```

```

while (1) {
    PORT3 |= 0x04; //将INT0(P3.2)置为高
    PORT3 &= ~0x04; //将INT0(P3.2)置为低，并产生中断
    PORT3 |= 0x04; //再次将INT0(P3.2)置为高
    swatch (0.5); //等待0.5秒
}
}

```

注意: twatch函数只能在信号函数内被调用。在信号函数外调用是不允许的并且会返回错误信息。

Void rwatch (address)

不断等待，直到规定的存储器地址被读取。

例如：下面的信号函数将会让port1.0信号线开关每当XDATA地址0x1234的内容被读取时。

```

signal void my_signal (void) {
    while (1) {
        PORT1 ^= 0x01; //改变P1.0
        rwatch (X:0x1234); //等待直到X:0x1234中的内容被读取
    }
}

```

注意: rwatch函数只能在信号函数内被调用。在信号函数外调用是不允许的并且会返回错误信息。

Void wwatch (address)

不断等待，直到规定的存储器地址被写入。

例如：在XDATA地址0x4000的内容被写入的时候，下面的信号函数将会让port1.0信号线开关。

```

signal void my_signal (void) {
    while (1) {
        PORT1 ^= 0x01; //改变P1.0
        wwatch (X:0x4000); //等待直到X:0x4000中的内容被写入
    }
}

```

注意: rwatch函数只能在信号函数内被调用。在信号函数外调用是不允许的并且会返回错误信息。

Int _TaskRunning_ (ulong func_address)

该函数检查当前运行的任务是否是某个特定的任务函数。`_TaskRunning_`只有在Options for Target-Target下选择了Operating System才可以使用。`μVision2`载入额外的DLL操作系统内核所知道的。

`_TaskRunning_`调试函数的结果可以在某个特定的任务处于激活状态时被指定到`_break_`系统变量中来中止程序的执行。

例如：

```

>_TaskRunning_ (command)           //检测任务' command' 是否在运行
0001                               //如果任务在运行，则返回 1
>_break_= _TaskRunning_ (init)     //中止程序执行，如果' init' 程序在运行的话

uchar _RBYTE (address)
uint _RWORD (address)
ulong _RDWORD (address)
float _RFLOAT (address)
double _RDOUBLE (address)

这些函数返回规定的存储器地址中的内容。例如：
>_RBYTE (0x20000)                //返回 0x20000 处的字节
>_RFLOAT (0xE000)                 //返回 0xE000 处的浮点数值
>_RDWORD (0x1000)                 //返回 0x1000 处的 long 型值

_WBYTE (address, uchar value)
_WWORD (address, uint value)
_WDWORD (address, ulong value)
_WFLOAT (address, float value)
_WDOUBLE (address, double value)

这些函数向规定的存储器地址中写入特定的内容。例如：
>_WBYTE (0x20000, 0x55)          //将 byte 类型的常量 0x55 写入 0x20000 处
>_RFLOAT (0xE000, 1.5)            //将浮点值 1.5 写入 0xE000 处
>_RDWORD (0x1000, 12345678)       //将 long 型值 12345678 写入 0x1000 处

```

(2) 用户函数

用户函数是创建以后和μVision2调试器配合使用的。在函数编辑器中可以直接进入用户函数或者使用INCLUDE命令来包括含有一个或者多个用户函数的文件。

注意：μVision2提供了一些系统变量可以在用户函数中使用它们。

使用FUNC关键字来定义的用户函数可以被如下方式定义：

```

FUNC return_type fname (parameter_list) {
    statements
}

```

上面定义中的几个参数的含义如下：

return_type: 是函数返回值的类型，它们可以是bit、char、float、int、long、uchar、uint、ulong、void等类型。如果函数不需要返回值的话，可以使用void类型。但是如果没有给用户函数指定返回类型的话，缺省类型是int。

Fname: 是函数的名字。

Parameter_list: 函数的参数的名称列表。每个参数必须有一个名字和并被指定类型。如果该函数中没有参数，在parameter_list中使用void。多个参数之间必须用逗号分开。

Statements: 函数中执行的指令。

例如，下面的用户函数显示了一些CPU寄存器中的内容。

```
FUNC void MyRegs (void) {
    printf ("----- MyRegs() -----\\n");
    printf (" R4 R8 R9 R10 R11 R12\\n");
    printf (" %04X %04X %04X %04X %04X %04X\\n",
            R4, R8, R9, R10, R11, R12);
    printf ("-----\\n");
}
```

为了激活这个函数，在命令窗口中键入一下字符。

>MyRegs()

当函数被激活以后，MyRegs函数显示了存储器中的内容，输出内容如下所示：

```
----- MyRegs() -----
R4 R8 R9 R10 R11 R12
B02C 8000 0001 0000 0000 0000
-----
```

在工具箱中可以定义一个按钮来激活这个程序，如下：

```
>DEFINE BUTTON "My Registers", "MyRegs ()"
```

不过，用户函数在使用时有一些限制，如下：

- μVision2 检查返回的值是否和用户函数定义的类型相同。返回类型为 void 的函数不可以有返回值，而不是 void 类型的函数则必须有返回值。
- 用户函数不能激活信号函数或者 twatch 函数等。
- 本地目标的值没有被定义直到被分配一个确定值为止。
- 可以使用 KILL FUNC 命令删除用户函数。

(3) 信号函数

信号函数可以完成一些重复操作，比方说信号的输入或者脉冲。当μVision2在执行目标程序时，信号函数会在后台执行。信号函数可以仿真、测试串口、模拟输入口、端口的通信以及其他的一些外部事件。

信号函数必须在某些位置调用twatch函数来延迟并让μVision2运行目标程序。对没有调用过twatch的信号函数，μVision2报告错误。

注意：μVision2提供了一系列的系统变量，在信号函数中可以使用这些变量。

信号函数以关键字SIGNAL开始，并按如下方式定义：

```
SIGNAL void fname (parameter_list) {
    statements
}
```

return_type: 是函数返回值的类型。它们可以是bit、char、float、int、long、uchar、uint、ulong、void等类型。如果函数不需要返回值的话，可以使用void类型作为函数类型。如果没有指定信号函数的返回值类型，那么缺省类型是int。

Fname: 是函数的名字。

Parameter_list: 函数的参数的名称列表。每个参数必须有一个名字和并被指定类型。如果该函数中没有参数，在parameter_list中使用void。多个参数之间必须用逗号分开。

Statements: 函数中执行的指令。

例如下面的例子，该例显示了一个信号函数在每1 000 000个CPU状态后将字符'A'输入串口输入缓存。

```
SIGNAL void StuffS0in (void) {
    while (1) {
        S0IN = 'A';
        twatch (1000000);
    }
}
```

为了激活这个函数，在命令窗口中键入以下内容：

```
>StuffS0in()
```

当被激活以后，StuffS0in信号函数会将ASCII字符'A'输入到串口输入缓存中，延迟1,000,000CPU状态后再重复此操作。

在信号函数使用时也有一些限定，如下：

- 信号函数的返回值类型必须为 void。
- 信号函数最多有 8 个函数参数。
- 信号函数可以激活调用其他的预定义函数或者用户函数。
- 信号函数不能调用其他的信号函数。
- 信号函数可以被一个用户函数所激活。
- 信号函数必须至少调用 twatch 函数一次。没有调用 twatch 的信号函数没有给目标程序以执行事件。因为不能用 Ctrl+C 来中止一个信号函数的执行，所以 μVision2 可能会进入一个无限的循环。

μVision2维护了一个有效信号函数队列。信号函数有两种状态：挂起和运行。所谓挂起的信号函数，一般是处于调用twatch的状态中，它会等待CPU运行若干周期直到达到twatch中规定的值。而正在执行的信号函数则指的是函数内的命令正在被执行。

当激活一个信号函数以后，μVision2将这个函数添加到队列中，并将它标注为运行状态。信号函数只能运行一次。如果要执行的函数已经在队列中了，μVision2会给出警告。要查看信号函数的状态，可以使用SIGNAL STATE命令；要删除信号函数的时候使用命令SIGNAL KILL。

当一个信号函数激活了twatch函数时，它就被挂起，处于等待的状态，直到等待的CPU时钟周期数到达预设值。在这以后，信号函数才会从twatch后面继续执行。

如果信号函数在return命令执行后退出了，那么它将自动从有效的信号函数队列中被移除。

下面是一个模拟信号的例子。该例是一个输入到有A/D转换器的某款8051芯片模拟输入0管脚的程序。输入的模拟信号不断变化。函数将输入的电压增加或者减少0.5伏特，电压变化范围为0伏到最高限定电压。最高限定电压为该函数的唯一参数。该信号函数无限循环，并在每个电压跳变以后延迟200 000个CPU周期。

```
signal void analog0 (float limit) {
```

```

float volts;
printf ("Analog0 (%f) entered.\n", limit);
while (1) {                                //永久循环
    volts = 0;
    while (volts <= limit) {
        ain0 = volts;                      //模拟输入-0
        twatch (200000);                  //200000 个周期用来等待
        volts += 0.1;                     //增加电压值
    }
    volts = limit;
    while (volts >= 0.0) {
        ain0 = volts;
        twatch (200000);                  //200000 个周期用来等待
        volts -= 0.1;                     //减少电压值
    }
}
}

```

信号函数analog0可以被按照如下方式激活:

```
>ANALOG0 (5.0)                                //开始运行'ANALOG()'
```

输出为:

```
ANALOG0 (5.000000) ENTERED
```

如果要用SIGNAL STATE命令显示analog0的当前状态,那么请输入:

```
>SIGNAL STATE
```

输出为:

```
1 idle Signal = ANALOG0 (line 8)
```

μVision2显示了内部函数的编号、信号函数的状态(挂起或者运行)、函数名和当前正在执行的行号。

在上面的例子中,信号函数的是挂起状态,所以可以知道analog0执行的是twatch函数(analog0的第8行),并且正在等待CPU执行完规定的周期数。当到达200,000周期以后,analog0继续执行,直到遇到下一条twatch语句(第8行或者第14行)。

下面的命令就会将analog0信号函数从有效信号函数队列中移除。

```
>SIGNAL KILL ANALOG0
```

4. 调试函数和C的不同

ANSI C和μVision2的用户函数和信号函数有一些不同,主要如下:

- μVision2不区分大小写。对象或者控制变量可以用大写或者小写来进行。
- μVision2没有预处理器,不支持例如#define、#include、#ifdef等预处理功能。
- μVision2不支持全局定义。所有的变量必须在函数中定义。可以使用DEFINE命令定义一个变量,然后像全局变量那样使用它。
- μVision2中,变量在定义的时候不能被赋初值。要初始化变量,必须显式地来给变

量赋值。

- μVision2 中仅仅支持标量变量，而不支持结构、数组和指针等。参数类型和返回类型均是如此。
- μVision2 中不支持函数的递归调用。在函数执行过程中，如果 μVision2 发现了递归调用，那么将中止函数的继续执行。
- μVision2 中的函数只能被直接使用函数名来调用。不支持通过指针的方式来调用函数。
- μVision2 支持 ANSI 的函数定义形式。不支持旧的 K&R 的函数定义方式。例如：

```
func test (int pa1, int pa2) {                                // ANSI 格式, 正确
    // ...
}
```

而下面的K&R函数方式则是非法的：

```
func test (pa1, pa2)                                //K&R 格式
int pa1, pa2;                                         //KEIL C51 不支持这种定义方式
{
    // ...
}
```

第 6 章 C51 编译器

C51 编译器的作用是将 C 语言源程序翻译成为 51 系列单片机的可执行代码，并且为程序调试提供必要的符号信息。KEILC 编译器根据 51 系列单片机的特征提供二十多条编译控制指令，这些指令既可以在 DOS 命令行中引用，也可以在源程序中作为预处理器命令行#pragma 的参数来引用。

C51 可以在 μ Vision2 IDE 开发环境下使用，，也可以在 DOS 命令行或者命令提示符下引用 C51 编译器，使用的基本格式如下：

C51 源程序文件名 控制指令表列

其中：

- “C51”是编译器调用命令。
- “源程序文件名”必须连同扩展名一块儿输入。
- “控制指令表列”是若干条用空格分开的编译控制指令。

例如：对于一个已经编写完成的 C 语言源程序 sample.c，DOS 命令行下可以按以下格式进行编译：

```
c51 sample.c debug code optimize(3) nopreprint
```

该例的控制表列中给出了 4 条控制指令，前三条是指令的肯定形式，最后一条是指令的否定形式。指令的否定形式只要在指令前面加 no 即可。这些指令的具体含义将在下一节详细介绍。某些控制指令还可以带有参数，该参数放在指令之后的圆括号中。大部分控制指令都有缩写形式，如上例可按缩写形式写成：

```
c51 sample.c DB CD OT(3) NOPP
```

对于大多数编译过程，用户只需要给出很少的几条控制指令就够了，因为编译程序设置了一组缺省控制项，如果用户不给出控制参数，C51 编译器将按照缺省控制项进行编译。

C51 编译器的控制指令也可以在 C 语言源程序中采用预处理命令#pragma 来引用。对于上面的例子，可以在 C 语言源程序文件的开始处加入以下内容：

```
#pragma DB CD OT(3) NOPP
```

在 DOS 状态下只要键入以下命令即可：

```
C51 sample.c
```

这样，在对源程序进行编译时，将自动引用#pragma 后面的各条控制参数。采用预处理命令#pragma 的好处是可以在 C 语言源程序中按照实际需要设置编译参数。

C51 编译器的控制命令可以分为两组：首要控制命令和一般控制命令。首要控制命令只能引用一次，一般控制命令则可以多次引用。这一点在源程序中采用#pragma 来引用时必须要注意。C51 编译器的首要控制命令通常放在源程序的开始处，并且只能出现一次，如果重复使用，将导致指明错误而停止编译。对于一般控制命令，则可以出现在源程序的任意位置，并且可以多次出现。

表 6-1 列出了 KEIL51 编译器的编译控制命令、缩写形式和默认值。其中，“P/G”列表

示该命令是首要控制还是一般控制命令：P 表示首要控制，G 表示一般控制。表中某些命令后面的参数“bname（基本名）”都与被编译的源程序有关，其默认值为源程序文件名。

编译命令又可以分为源控制、列表控制和目标控制 3 大类。源控制命令用于宏定义以及决定是否支持对 C51 的特殊扩展。列表控制命令用于规定编译后所产生的列表文件的格式以及是否生成某些特殊内容，列表文件的扩展名为“.LST”。目标控制命令最多，作用最大，使用最频繁，用于控制编译之后生成目标文件的形式和内容，例如控制对目标文件的优化级别，使用不同的编译模式来规定变量的存储器空间，是否在目标文件中加入符号调试信息等等。目标文件的扩展名为“.OBJ”。在本章的 6.2 节“C51 编译器控制指令详解”中将对 C51 编译器所有的编译控制命令功能进行详细描述。

表 6-1 KEIL51 编译器的编译控制命令、缩写形式和默认值

分类	P/G	指令	缩写	默认值	命令功能
源	G	DEFINE	DF	--	定义在命令行中由预处理器使用的名字
	P	[NO] EXTEND	--	EXTEND	支持 C51 编译器的特殊扩展
	G	ASM	--	--	在线汇编起始标记
	G	ENDASM	--	--	在线汇编终止标记
		INCDIR	--	--	指定包含文件的查找路径
列表	P	[NO] CODE	[NO] CD	NOCODE	列表文件后附加汇编程序的代码
	G	[NO] COND	[NO] CO	COND	列出被条件编译忽略的程序行
	G	EJECT	EJ	--	列表文件换页
	G	[NO] LISTINCLUDE	[NO] LC	NOLISTINCLUDE	在列表文件中列出包含文件内容
	P	PAGELENGTH	PL	PAGELENGTH(60)	设置列表文件中每页的行数
	P	PAGEWIDTH	PW	PAGEWIDTH(132)	设置列表文件中每行字符数
	P	[NO] PREPRINT	[NO] PP	NOPREPRINT	产生预处理器列表文件
	P	[NO] PRINT	[NO] PR	PRINT(bname. LST)	产生列表文件
	P	[NO] SYMBOLS	[NO] SB	NOSYMBOLS	产生程序模块中使用过的符号表
	P	WARINGLEVEL	WL	WARINGLEVEL(2)	选择警告检测级别
目标	G	[NO] ARGES	[NO] AR	ARGES	使用绝对寄存器寻址方式
	P	BROWSE	BR		指令 BROWSE 可以是编译器产生浏览信息
	P	COMPACT	CP	--	紧凑编译模式
	P	[NO] DEBUG	[NO] DB	NODEBUG	将符号调试信息加入目标文件
	G	DISABLE	--	--	在函数执行期间禁止中断
	G	FLOATFUZZY	FF	FLOATFUZZY(3)	规定浮点数进行比较时的舍入位数
	P	INTERVAL	--	INTERVAL(8)	规定中断向量之间的间隔
	P	[NO] INTPROMOTE	[NO] IP	INTPROMOTE	允许 ANSI 标准整形数据提升
	P	[NO] INTVECTOR	[NO] IV	INTVECTOR	产生中断向量地址
	P	LARGE	LA	--	大编译模式
	P	MAXARGS	MA	MAXARGS(40)	规定参数变量表列的大小
	G	[NO] MODA2	--	[NO] MODA2	激活双 DPTR 寄存器对 Atmel82x8252 及变量的支持

续表

分类	P/G	指令	缩写	默认值	命令功能
目标	G	[NO]MODAB2	--	[NO]MODAB2	激活双 DPTR 寄存器对 AduC B2 系列的支持
	G	[NO]MODDA	--	[NO]MODDA	激活代码对 Dallas80c390, 80c400 和 5240 种的算术加速器的支持
	G	[NO]MODP2	--	[NO]MODP2	激活双 DPTR 寄存器对 PHILIPS 附加硬件的支持
	G	[NO]MOD517	--	NOMOD517	允许使用 80517 附加硬件功能
	G	[NO]MODDP2	--	NOMODDP2	允许使用 DS 和 AMD 附加硬件功能
源文件	P	NOAMAKE	NOAM	AMAKE	禁止使用项目文件的 AOTOMAKE 功能
	P	NOAMAKE	NOAM	AMAKE	禁止使用项目文件的 AOTOMAKE 功能
	P	[NO]OBJECT	[NO]OJ	OBJECT(bname.OBJ)	命名目标文件
	P	OBJECTEXTEND	OE	--	目标文件中包含附加变量类型信息
	P	OMF2	O2	--	输出 OMF2 形式的输出文件
	G	ONEREGBANK	OB	--	认定只有 BANK0 能用于中断模式
	G	OPTIMIZE	OT	OPTIMIZE(6, SPEED)	设置优化级别
	P	ORDER	OR	NOORDER	目标文件中变量按照源文件的定义定位
	P	REGFILE	RF		为全局优化规定一个寄存器定义文件
	G	REGISTERBANK	RB	REGISTERBANK(0)	设置当前使用的工作寄存器组
	G	[NO]REGPARMS	--	REGPARMS	[不] 允许使用寄存器进行参数传递
	P	ROM	--	ROM(LARGE)	决定程序空间范围
	P	RET_PSTK\ RET_XSTK	--		使用深堆栈存储返回地址
	G	RESTORE	--	--	恢复由 SAVE 命令保存的设置
	G	SAVE	--	--	保存当前的设置
	P	SMALL	SM	SMALL	小编译模式
	P	SRC	--	--	产生汇编语言源文件
	P	USERCLASS	UCL		为灵活变量的定位区重新命名储存类
	P	VARBANKING	VB		允许使用 FAR 储存类型的变量
	P	XCROM	XC		为 const XDATA 变量指定 ROM 地址

6.1 预处理

单片机 8051 系统下的 Ax51 宏编译器支持 C 源程序，因此在这里介绍 C 的编译预处理功能。在 C 编译系统对程序进行编译之前，先要对这些程序进行预处理。然后再将预处理的结果和源程序一起再进行正常的编译处理得到目标代码。预处理主要有以下 3 种：宏定义、文件包含、和条件编译。通常的预处理命令都用“#”开头。例如：

```
#pragma
#include <stdio.h>
#define DEBUG 1
```

6.1.1 宏定义

宏定义，即#define 指令，具有如下形式：

```
#define 名字 替换文本
```

它是一种最简单的宏替换。出现在各处的“名字”都将被“替换文本”替换。#define 指令中的名字与变量名具有相同的形式，替换文本可以是任意字符串。正常情况下，替换文本是#define 指令所在行的剩余部分，但也可以把一个比较长的宏定义分成若干行，这时只需在待延续的行后加上一个反斜杠\即可。#define 指令所定义的名字的作用域从其定义点开始，到被编译的源文件的结束。在宏定义中也可以使用前面已经被定义过的宏定义。替换只对单词进行，对括在引号中的字符串不起作用。例如，如果 YES 是一个被定义的名字，那么在 printf (“YES”) 或 YESMAN 中不能进行替换。

1. 不带参数的宏定义

不带参数的宏定义指用一个指定的符号来表示一个字符串，其形式为：

```
#define SYMBOL STRING
```

其中：“SYMBOL”就是指定的符号，而“STRING”就是赋给符号的字符串。其使用如下：

```
#define H 128
```

它的作用就是指定符号 H 来代替 128，在预处理时，把程序在该命令以后的所有 H 都用 128 来代替。所以使用不带参数的宏定义我们可以用宏名代替一个字符串，可以减少程序中的重复书写。例如：

```
#define H 128
main()
{
    int H, l;
    l=H*H;
    printf("l=%d", l);
}
```

注意：

①宏定义只是用宏名代替一个字符串，不做语法检查。所以当输入字符串有误时，预处理

时不会产生错误信息。

②宏定义由于不是C的语句所以不用在行末加分号“;”。

③#define命令在程序之外，所以宏名的有效范围为定义命令之后到源文件结束。

但是我们可以使用#undef命令终止宏定义的作用域，例如：

```
#define H 128
main()
{
#define H
FUNCTION()...
```

在上面的例子中H的范围从#define到#undef H结束。

2. 带参数的宏定义

和前面的不带参数的宏定义不同的是带参数的宏定义不是进行简单的字符串替换，还有参数替换。其定义的形式一般如下：#define 宏名(参数表) 字符串

如下：

```
#define stringer(x) DB #X, 0x0B, 0x0C
stringer(text)
```

预处理的输出结果为DB “text”， 0x0B, 0x0C

注意：对于带参数的宏定义时，在宏名与带参数的圆括号之间不应该有空格，否则空格以后的字符都作为替代字符串的一部分。

6.1.2 文件包含

文件包含指令，即#include指令，使我们比较容易处理一组#define指令以及说明等。在源程序文件中，任何形如：#include"文件名"，或#include<文件名>的行都被替换成由文件名所指定的文件的内容。如果文件名用引号括起来，那么就在源程序所在位置查找该文件；如果在这个位置没有找到该文件，或者如果文件名用尖括号“<>”括起来，那么就按定义的规则来查找该文件。被包含的文件本身也可包含#include指令。

在源文件的开始处一般都要有一些#include指令，或包含#define语句与extern说明，或访问诸如<stdio.h>等头文件中库函数的函数原型说明。（严格地说，这些没有必要做成文件。访问头文件的细节依赖于编程者是如何实现它们的）

对于比较大的程序，#include指令是把各个说明放在一起的优选方法。这样可以使得所有源文件都被提供以相同的定义与变量说明，从而可避免发生一些特定的错误。自然，如果一个被包含的文件的内容做了修改，那么所有依赖于这个被包含文件的源文件都必须重新编译。

所谓文件包含处理就是指一个源文件可以将另外一个源文件的全部内容包含进来。使用指令

```
#include "FILENAME"
```

其中FILENAME是想要包含进去的文件名。在编译预处理时，对#include命令进行文件

包含处理。实际上就是将文件 (FILENAME) 中全部内容复制插入到#include “FILENAME” 命令处。文件包含命令可以减少不必要的重复劳动。如下文件 “file1.c” 的内容如下：

```
#include "file2.c"
main()
{...}
```

注意对文件包含命令并不是把两个文件连接起来，而是编译时作为一个源程序编译，得到一个目标文件 (.obj)。由于这种常用在文件头部的被包含的文件常被称为“头文件”，经常以“.h”为后缀名，当然后缀为“.C”也是正确的。

注意：

①如果文件 1 包含文件 2，文件 2 中又要用到文件 3，我们可以在文件 1 中用两个 include 命令。而且文件 3 的包含应该文件文件 2 的包含之前，如下：

```
#include "file3.c"
#include "file2.c"
```

当文件更多时，只要以此类推即可，但要注意包含命令的顺序问题。

②一个 include 命令只能指定一个被包含文件，如果要包含多个文件要用多个 include 命令。

③在 include 命令中，文件名除了可以用双引号之外还可以用尖括号。如下：在 file1.c 中用如下的文件包含命令。

```
#include "file2.h"
#include<file2.h>
```

两者的区别是：用尖括号时，系统不检查源文件 file1.c 所在的文件目录而直接按系统指定的标准方式检索问检目录。而对于双引号的形式，系统先在引用被包含文件的源文件 file1.c 所在得文件目录中寻找要包含的文件，找不到之后，再按系统指定的标准方式检索目录。

④由于被包含文件与其所在文件在预编译之后成为一个源文件。所以被包含文件中的全局静态变量，在现在的源文件中同样有效。

6.1.3 条件编译

在预处理语句中还有一种条件语句，用于在预处理中进行条件控制。这提供了一种在编译过程中可以根据所求条件的值有选择地包含不同代码的手段。#if 语句中包含一个常量整数表达式（其中不得包含 sizeof、类型强制转换运算符或枚举常量），若该表达式的求值结果不等 0 时，则执行其后的各行，直到遇到#endif、#elif 或#else 语句为止（预处理语句#endif 类似于 if 语句的 else if 结构）。在#if 语句中可以使用一个特殊的表达式 defined(名字)：当名字已经定义时，其值 1；否则，其值为 0。

例如，为了保证 hdr.h 文件的内容只被包含一次，可以像下面这样用条件语句把该文件的内容包围起来：

```
#if !defined(HDR)
#define HDR
```

```
//hdr.h文件的内容
```

```
#endif
```

被#if 与#endif 包含的第一行定义了名字 HDR，其后的各行将会发现该名字已有定义并跳到#endif。

还可以用类似的样式来避免多次重复包含同一文件。如果连续使用这种，那么每一个头文件中都可以包含它所依赖的其他头文件，而不需要它的用户去处理这种依赖关系。

下面的预处理语句序列用于测试名字 SYSTEM 以确定要包含进哪一个版本的头文件：

```
#if SYSTEM== SYSV
#define HDR "sysv.h"
#elif SYSTEM == BSD
#define HDR "bsd.h"
#elif SYSTEM == MSDOS
#define HDR "msdos.h"
#else
#define HDR "default.h"
#endif
#include HDR
```

当需要测试一个名字是否已经定义时，可以使用两个特殊的预处理语句：#ifdef 与 #ifndef。可以使用#ifndef 将上面第一个关于#if 的例子改写如下：

```
#ifdef HDR
#define HDR
//hdr.h文件的内容
#endif
```

条件编译命令的总共有 3 种形式：

(1) #ifdef 标识符

```
程序段 1
#else
程序段 2
#endif
```

其实现的功能是：如果标志符已经被定义过了，则编译程序段 1。否则编译程序段 2。其中#else 部分也可以没有。如下：

```
#ifdef 标识符
程序段 1
#endif
```

这种条件编译通常用于调试程序中。因为在调试程序时，有时希望输出一些所需信息，而在调试完之后就不需要在输出这些信息。可在源程序中加入以下的条件编译：

```
#ifdef DEBUG
printf( "a=%f, b=%f, c=%f\n" , a, b, c)
#endif
```

(2) #ifndef 标识符

```
程序段 1
#else
程序段 2
#endif
```

其实现的功能和上面的差不多，只不过是当标识符未被定义时则编译程序段 1，否则编译程序段 2。

(3) #if 条件表达式

```
程序段 1
#else
程序段 2
#endif
```

其实现的功能差不多，只是当指定的条件表达式为真时，就编译程序段 1，否则编译程序段 2。这样就可以自己设置条件编译的条件。

6.1.4 其他预处理命令

在前面介绍预处理的 3 种形式时，用到了一部分预处理命令。但是还有一些比较简单但也很常用的预处理器命令，如字符串化的操作符，连接符等。本节着重介绍这两种形式。

1. 字符串化的操作

在宏定义中，我们可以使用#符来将宏中的某一个参数转换成一个字符串的常数。这种操作形式必须用在一个需要某个参数或者参数列表的宏定义中。在字符串化的操作是一个宏的参数名时，传递到该宏中的参数实际上在编译时已经被用引号括起来了，并会在程序中被直译成一个字符串。例如：

```
#define stringer(x) printf(#x"\n")
stringer(text)
```

该例经编译后与以下语句相同：

```
printf("text\n")
```

这说明参数被直接当作一个字符串了。事实上，在编译的时候，首先生成的是如下格式的命令：

```
printf("text""\n")
```

因为被空格分割的字符串在编译时自动会被连起来，所以这两个字符串会被合为 "text\n"。

如果字符串化中有特殊的字符，例如"\\"等，那么额外的"\\"会自动在编译时添加。

2. 连接符

连接符在 C51 中用##来表示。该字符允许两个宏定义中的字符被连接成一串字符。

如果使用了参数的宏定义中又使用了连接符，那么可能会有特殊的效用。例如：

```
#define paster(n) printf ("token"#n"%=d", token##n)
paster(9);
```

上面的语句等效于：

```
printf("token9=%d", token9);
```

3. 预定义好的常量

这些常量主要供编译器使用，所有的常量都已经列在了表 6-2 中。

表 6-2

预定义的常量

常量名	说明
C51	C51 编译器的版本号。例如，如果为 610，那么就是版本 6.10
CX51	CX51 编译器的版本号。例如，如果为 610，那么就是版本 6.10
DATE	在编译中以 ANSI 格式开始的数据。例如 (month dd yyyy)
DATE2	在编译中以短格式开始的数据。例如 (mm/dd/yy)
FILE	被编译的文件名
LINE	当前被编译的文件的行数
MODEL	选择的存储模式 (0 为 SMALL 模式, 1 为 COMPACT 模式, 2 为 LARGE 模式)
TIME	编译开始时的时间
STDC	在为 1 时表示完全与 ANSI C 兼容

为了方便大家的使用，下面我们将所有的所有预处理命令列在下面，并加上简单的说明。

(1) define: 预处理的宏定义指令，其用法前面有详细的介绍。

(2) elif: 条件编译中，当前面的 if, ifdef, ifndef 或 elif 后的程序段不满足条件时，则选择后面的条件继续做条件编译。其形式如下：

```
#if 条件表达式 1
    程序段 1
#elifif 条件表达式 2
    程序段 2
#elifif 条件表达式 3
    程序段 3
...
#else 程序段 n
#endif
```

其功能是：条件表达式 1 如果为真，则编译程序段 1；若为假再看看条件表达式 2 是否为真，为真则编译程序段 2；若为假则再看看条件表达式 3 是否为真若为真编译程序段 3；若为假再看看条件表达式 4 以此类推下去。

(3) else: 条件编译命令。当前面的条件表达式为假时，编译 else 后面的程序段。其用法在前文已经做过介绍。

(4) endif: 结束条件编译的指令。在前文有其具体用法的例子。

(5) error: 该命令通常嵌入在条件编译中，以捕捉到一些预料之外的编译条件。正常情况下该条件的值为假，若为真，则输出一条由用户指定的#error 后面的字符串所给出的错误信息并停止编译。如#define value, 其值必为“5”我们就可以测试 value 是否正确，可用如下的程序测试：

```
#if(value!=5)
#error value is wrong, please correct
```

```
#endif
```

- (6) **ifdef**: 条件编译命令。前文有其具体用法的例子。
 - (7) **ifndef**: 和 **ifdef** 一样条件相反而已。前文有其具体用法的例子。
 - (8) **if**: 也是用在条件编译中，引出条件编译中所设置的条件。
 - (9) **include**: 该命令是文件包含命令，其比较详细的用法请见前文的文件包含预处理部分。
 - (10) **line**: 该命令连同可选择的文件名一起指定行号。这些设置在错误信息中用于标明发生错误的地方。
 - (11) **pragma**: 该命令可以让使用者指定汇编控制并将其转换成编译器 Ax51 控制行，即向编译器传送编译控制命令。其形式如下：
- ```
pragma 编译命令
```
- (12) **undef** 删除宏。

## 6.2 C51 编译器控制指令详解

按 C51 编译器控制指令的功能可以将其分为三类：源控制指令、列表控制指令和目标控制指令。下面将其分别介绍。

### 6.2.1 源控制指令

这类指令用于进行宏定义并声明被编译的文件名。

- 指令名: **DEFINE**

缩写: DF

变量: 一个或几个用逗号隔开的变量名，变量的定义方法与 C 语言的命令规则一致

属性: 首要控制

默认值: 无

功能: **DEFINE** 指令定义了在命令行使用的名字，可为“if”、“ifdef”和“ifndef”等预处理命令作条件编译。所定义的变量名字对大小写敏感。

例子:

```
C51 SAMPLE.C DEFINE(check, NoExtRam)
C51 MYPROG.C DF(X1="1+5", iofunc="getkey()")
```

- 指令名: **DISABLE**

缩写: 无

变量: 无

属性: 一般控制

默认值: 无

功能: **DISABLE** 指令在一个函数的执行期间禁止所有的中断。**DISABLE** 指令必须在函数的前面以**#pragma** 参数的形式出现，并且只能用在一个函数中，因此它是由编译器内部设置

的。每个要禁止中断的函数必须在函数的前面放一个独立的#pragma行。

**注意:** DISABLE指令只能以#pragma参数的形式调用,不能用于DOS命令行。使用DISABLE指令禁止中断的函数不能返回一个比特(bit)值。

- 指令名: EXTEND/NOEXTEND

缩写: 无

变量: 无

属性: 首要控制

默认值: EXTEND

功能: EXTEND指令使编译器支持ANSI-C对C51的特殊扩展。NOEXTEND指令则使编译器只处理ANSI-C,这时C51的扩展关键字如bit、reentrant、interrupt和using等对编译器都是未知的。

例子:

```
C51 SAMPLE.C NOEXTEND
#pragma NOEXTEND
```

- 指令名: ASM/ENASM

缩写: 无

变量: 无

属性: 一般控制

默认值: 无

功能: ASM指令将每一区的源文档的开头部分并入由SRC指令产生的.SRC文件中,但是该源文档并不输出到目标文件中。ENASM指令表明源文档区的结束。

例子:

```
#pragma asm/pragma enasm
```

源文件为如下的C程序:

| stmt | level | source               |
|------|-------|----------------------|
| 1    |       | extern void test     |
| 2    |       |                      |
| 3    |       | main() {             |
| 4    | 1     | test();              |
| 5    | 1     |                      |
| 6    | 1     | # pragma asm         |
| 7    | 1     | jmp \$; endless loop |
| 8    | 1     | # pragma enasm       |
| 9    | 1     | }                    |

- 指令名: INCDIR

缩写: 无

变量：圆括号中包含文件的详细路径

属性：一般控制

默认值：无

功能：INCDIR 指令为 CX51 编译器的找寻包含文件指定路径，最大为 50 个。若有多个路径必须用分号隔开。当有命令行#include “filename.h” 时，CX51 编译器先搜寻当前目录。当找不到包含文件时，便按照 INCDIR 指定的目录来查找包含文件。当此时]仍然找不到包含文件，则由 C51 INC 外界变量指定的路径来找寻。

例子：

```
C51 SAMPLE.C INCDIR (C:\KEIL\C51\MYPRO;C:\CHIP_DIR)
```

### 6.2.2 列表控制指令

这类指令控制列表文件的格式及产生的某些特殊内容。

- 指令名：CODE

缩写：CD, NOCD

变量：无

属性：首要控制

默认值：NOCODE

功能：CODE 指令在列表文件的后面附加一个汇编代码文件，将 C 语言源程序中的每个函数都用汇编代码表示出来。使用 NOCODE 指令将不产生附加汇编代码文件。

例子：

```
C51 SAMPLE.C CD
```

```
#pragma code
```

下面是一个采用 CODE 控制指令产生的汇编代码列表文件的例子，汇编代码所对应的每条 C 语言源程序的行号都显示出来。字符“R”和“E”分别表示“可重定位”和“外部”。

下面是某程序编译产生的.LST 文件：

| stmt                                      | level | source                        |
|-------------------------------------------|-------|-------------------------------|
| 1                                         |       | extern unsigned char a, b;    |
| 2                                         |       | unsigned char c;              |
| 3                                         |       |                               |
| 4                                         |       | main()                        |
| 5                                         |       | {                             |
| 6                                         | 1     | c = 14 + 15 *((b / c) + 252); |
| 7                                         | 1     | }                             |
| .                                         |       |                               |
| .                                         |       |                               |
| .                                         |       |                               |
| ASSEMBLY LISTING OF GENERATED OBJECT CODE |       |                               |
| ; FUNCTION main(BEGIN)                    |       |                               |
| ; SOURCE LINE # 5                         |       |                               |

```
; SOURCE LINE # 6
0000 E500 E MOV A, b
0002 8500F0 R MOV B, c
```

```
0005 84 DIV AB
0006 75F00F MOV B, #0FH
0009 A4 MUL AB
000A 24D2 ADD A, #0D2H
000C F500 R MOV c, A
; SOURCE LINE # 7
000E 22 RET
; FUNCTION main(END)
```

其实前面的不少.LST文件都是使用了CODE命令编译才生成的。

- 指令名: COND

缩写: CO, NOCO

变量: 无

属性: 首要控制

默认值: COND

功能: 指令COND和NOCOND决定被编译的源文件中条件编译部分是否出现在列表文件中。被编译的源文件中如果使用了条件编译预处理命令,在编译时将根据不同的条件使得一些程序行被忽略而不进行编译,COND指令使这些被忽略的程序行出现在列表文件中,但他们没有行号和嵌套级,从而易于识别。使用NOCOND指令将禁止在列表文件中输出由于条件编译而忽略的程序行。

例子:

```
C51 SAMPLE.C COND
#pragma noco
```

下面是对一个使用了条件编译预处理命令的源程序采用COND指令进行编译所产生的列表文件,文件中列出了由于条件编译而忽略掉的源程序行,请注意它们没有行号。

```
...
...
...
stmt level source
1 extern unsigned char a, b;
2 unsigned char c;
3
4 main()
5 {
6 1 #if defined(VAX)
7 c = 13;
8 #elif defined(_ _TIME_ _)
```

```

9 1 b = 14;
10 1 a = 15;
11 1 #endif
12 1 }
...
...
...
...
...
...
stmt level source
1 extern unsigned char a, b;
2 unsigned char c;
3
4 main()
5 {
6 1 #if defined(VAX)
9 1 b = 14;
10 1 a = 15;
11 1 #endif
12 1 }
...
...
...

```

从上面的文件中可以看出，与源程序相比少了

c = 13;

及

#elif defined(\_\_TIME\_\_)

两条语句。读者可以自己在多试几个例子，以体会该命令的用法。

- 指令名: EJECT

缩写: EJ

变量: 无

属性: 一般控制

默认值: 无

功能: 使列表文件换页

例子:

#pragma eject

**注意:** EJECT 指令只能在源程序文件中以#pragma 语句的参数形式出现，不能够用于 DOS 命令行。

- 指令名: LISTINCLUDE

缩写: LC, NOLC

变量: 无

属性: 首要控制

默认值: NOLISTINCLUDE

功能: LISTINCLUDE 指令将在编译时所产生的列表文件中列出头文件的内容, 使用 NOLISTINCLUDE 指令时将不列出头文件内容。

例子:

```
C51 SAMPLE.C LISTINCLUDE
```

```
#pragma listinclude
```

- 指令名: PAGELENGTH

缩写: PL

变量: 括号内一个不超过 65,535 的十进制数

属性: 首要控制

默认值: PAGELENGTH(69)

功能: PAGELENGTH 指令规定了列表文件中每页的行数。默认值为每页 69 行, 包括抬头和空行

例子:

```
C51 SAMPLE.C PAGELENGTH(70)
```

```
#pragma pl(70)
```

- 指令名: PAGEWIDTH

缩写: PW

变量: 括号内一个 78~132 的十进制数

属性: 首要控制

默认值: PAGEWIDTH(132)

功能: PAGEWIDTH 指令规定了列表文件中每行的字符数, 若超过此数该行将变为两行或者多行。默认值为每行 132 个字符。

例子:

```
C51 SAMPLE.C PAGEWIDTH(79)
```

```
#pragma pw(79)
```

- 指令名: PREPRINT

缩写: PP, NOPP

变量: 括号内可选的文件名

属性: 首要控制

默认值: NOPREPRINT

功能: PREPRINT 指令产生一个预处理器列表文件, 宏调用被展开并且将注释删掉。如果 PREPRINT 指令不带变量, 则用源文件名加上扩展名 “.i” 作为列表文件名。如果希望采用其他列表文件名, 则必须用括号内的变量指出希望的文件名。使用 NOPREPRINT 指令将不产生于处理器列表文件。

例子:

C51 SAMPLE.C PREPRINT

C51 SAMPLE.C PP(PREPRO.LSI)

- 指令名: PRINT

缩写: PR, NOPR

变量: 括号内可选的文件名

属性: 首要控制

默认值: PRINT (bname.LST)

功能: PRINT 指令使用制定的路径、源文件名及扩展文件名“.LST”为每个被编译的源程序产生一个列表文件。如果不希望以源文件名作为列表文件名，则可在指令后面的括号内用一个变量名来指定所希望的列表文件名。使用 NOPRINT 指令将不产生列表文件。

例子:

```
C51 SAMPLE.C PRINT (CON:)

#pragma pr(\usr\list\sample.lstkankna ni)

C51 SAMPLE.C NOPRINT

#pragma nopr
```

- 指令名: SYMBOLS

缩写: SB, NOSB

变量: 无

属性: 首要控制

默认值: NOSYMBOLS

功能: SYMBOLS 指令在编译时产生一个被程序模块使用过的符号表。对于每一个符号对象，都给出其属性、存储器类型、偏移量及对象的大小。NOSYMBOLS 指令将使编译器不产生该信息。

例子:

```
C51 SAMPLE.C SYMBOLS

#pragma SYMBOLS
```

下面是部分符号的列表:

| NAME      | CLASS   | MSPACE | TYPE   | OFFSET | SIZE  |
|-----------|---------|--------|--------|--------|-------|
| EA        | ABSBIT  | -----  | BIT    | 00AFH  | 1     |
| update    | PUBLIC  | CODE   | PROC   | -----  | ----- |
| dtime     | PARAM   | DATA   | PTR    | 0000H  | 3     |
| setime    | PUBLIC  | CODE   | PROC   | -----  | ----- |
| mode      | PARAM   | DATA   | PTR    | 0000H  | 3     |
| dtime     | PARAM   | DATA   | PTR    | 0003H  | 3     |
| setuptime | AUTO    | DATA   | STRUCT | 0006H  | 3     |
| time      | * TAG * | -----  | STRUCT | -----  | 3     |
| hour      | MEMBER  | DATA   | U_CHAR | 0000H  | 1     |
| min       | MEMBER  | DATA   | U_CHAR | 0001H  | 1     |

|      |        |      |        |       |     |
|------|--------|------|--------|-------|-----|
| sec  | MEMBER | DATA | U_CHAR | 0002H | 1   |
| SBUF | SFR    | DATA | U_CHAR | 0099H | 1   |
| ring | PUBLIC | DATA | BIT    | 0001H | 1   |
| SCON | SFR    | DATA | U_CHAR | 0098H | 1   |
| TMOD | SFR    | DATA | U_CHAR | 0089H | 1   |
| TCON | SFR    | DATA | U_CHAR | 0088H | 1   |
| mnu  | PUBLIC | CODE | ARRAY  | 00FDH | 119 |

- 指令名: WARINGLEVEL

缩写: WL

变量: 圆括号中的从 0~2 之间的数字

属性: 首要控制

默认值: WARINGLEVEL(2)

功能: 指令 WARINGLEVEL 允许使用者抑制编译警告信息。其中抑制的程度是通过圆括号中的参数来体现的。

- 参数为 0: 抑制所有所能抑制的警告信息。
- 参数为 1: 只列出有可能产生错误代码的警告信息。
- 参数为 2: 列出所有的警告信息。

例子:

```
C51 SAMPLE.C WL(1)
#pragma WARINGLEVEL(1)
```

- 指令名: BROWSE

缩写: BR

变量: 无

属性: 首要控制

默认值: 不产生浏览信息

功能: 指令 BROWSE 可以使编译器产生浏览信息, 浏览信息包括标志符、存储空间、及其类型、定义和相关列表。

例子:

```
C51 SAMPLE.C BROWSE
#pragma browse
```

- 指令名: SRC

缩写: 无

变量: 括号内可选的文件名

属性: 首要控制

默认值: 不产生 SRC 文件

功能: SRC 指令使 C51 编译器在对 C 语言的源文件进行编译时不产生目标文件, 而产生一个汇编语言源文件, 所产生的汇编语言源文件可用汇编程序 A51 进行汇编而产生目标代码。如果不用括号内的变量指出汇编语言源文件名, 则用 C 语言源文件名加上扩展名 “.SRC” 作为汇编语言源文件名。

例子：

```
C51 SAMPLE.C SRC
C51 SAMPLE.C SRC(SML.A51)
```

**注意：**如果使用 SRC 指令产生一个汇编语言文件，编译器会自动抑制目标文件的产生，也就是说，不能同时产生汇编语言文件和目标文件。

### 6.2.3 目标控制指令

这类指令影响生成的目标文件的形式和内容，例如调试信息或者控制目标文件的优化级别等选项的添加都是由这类指令来完成的。

- 指令名：AREGS/NOAREGS

缩写：无

变量：无

属性：一般控制

默认值：AREGS

功能：AREGS 指令时编译器使用绝对寄存器寻址方式，从而增加效率。例如，PUSH 和 POP 指令只能使用绝对寻址。用 REGISTERBANK 指令可以建立所使用的寄存器组。NOAREGS 指令关闭绝对寻址方式。未使用 NOAREGS 指令编译的函数不依赖于寄存器组，即可以使用 8051 单片机所有的寄存器组。

下面是一个使用 AREGS/NOAREGS 指令的例子。

| stmt                         | level  | source               |
|------------------------------|--------|----------------------|
| 1                            |        | extern char func();  |
| 2                            |        | char k;              |
| 3                            |        |                      |
| 4                            |        | #pragma NOAREGS      |
| 5                            |        | noaregfunc() {       |
| 6                            | 1      | k = func() + func(); |
| 7                            | 1      | }                    |
| 8                            |        |                      |
| 9                            |        | #pragma AREGS        |
| 10                           |        | aregfunc() {         |
| 11                           | 1      | k = func() + func(); |
| 12                           | 1      | }                    |
| ; FUNCTION noaregfunc(BEGIN) |        |                      |
| ; SOURCE LINE # 6            |        |                      |
| 0000                         | 120000 | E LCALL func         |
| 0003                         | EF     | MOV A, R7            |
| 0004                         | C0E0   | PUSH ACC             |
| 0006                         | 120000 | E LCALL func         |

```

0009 D0E0 POP ACC
000B 2F ADD A, R7
000C F500 R MOV k, A
; SOURCE LINE # 7
000E 22 RET
; FUNCTION noaregfunc(END)
; FUNCTION aregfunc(BEGIN)
; SOURCE LINE # 11
0000 120000 E LCALL func
0003 C007 PUSH AR7
0005 120000 E LCALL func
0008 D0E0 POP ACC
000A 2F ADD A, R7
000B F500 R MOV k, A
; SOURCE LINE # 12
000D 22 RET
; FUNCTION aregfunc(END)

```

注意到这两个程序对 R7 的处理方式。对 noaregfunc 函数：

MOV A, R7

PUSH ACC

对 aregfunc 则是：

PUSH AR7

**注意：**AREGS/NOAREGS 指令可以作为#pragma 命令的参数在 C 语言源程序中出现多次，但是只能在函数的外部使用。

- 指令名：SMALL/COMPACT/LARGE

缩写：SM/CP/LA

变量：无

属性：首要控制

默认值：SMALL

功能：这 3 条指令控制编译器对源程序进行编译时所采用的存储器模式。不同的存储器模式对不同的变量影响如下：

**SMALL 模式：**所有函数和过程的变量及局部数据段被定义在 8051 单片机的内部数据存储器中，因此对数据对象的访问最快，缺点是地址空间有限。

**COMPACT 模式：**所有的函数和过程的变量及局部数据段被定义在 8051 单片机外部数据存储器中，该存储器以 256Byte 为 1 页。这种模式使用访问外部数据存储器的简洁形式@R0/@R1，高 8 位地址由 8051 的 P2 口输出。

**LARGE 模式：**所有变量和局部数据段都定义在 8051 单片机的外部数据存储器中，可访问 64kB 字节的全部地址空间。由于在访问数据对象是需要采用数据指针 DPTR，因而是效率不高的数据访问形式。

例子：

```
C51 SAMPLE.C COMPACT
#pragma compact
```

注意：调用子程序时所用到的堆栈始终放在 8051 单片机内部数据存储器中。

- 指令名： DEBUG

缩写： DB, NODB

变量： 无

属性： 首要控制

默认值： NODEBUG

功能： DEBUG 指令使编译器将符号调试信息加入到目标文件中，该信息对于程序的符号化调试是必要的，它包括全局和局部变量及其地址、函数名以及行号等。目标模块中的调试信息可以被模拟仿真调试程序 DS51 或者 Inter 兼容的仿真器用来对 C 语言源程序进行源级符号化调试。使用 NODEBUG 指令将不产生符号调试信息，因此也就不能对源程序进行符号化调试。

例子：

```
C51 SAMPLE.C DEBUG
#pragma db
```

- 指令名： DISABLE

缩写： 无

变量： 无

属性： 一般控制

默认值： 无

功能： 指令 DISABLE 指示编译器产生无中断的代码即将程序中的中断视为无效。该指令必须在定义函数之前直接用指令 #pragma DISABLE。注意一个 DISABLE 指令只能用于一个函数，若需拥有新的函数只需重新使用该命令即可。

例子：

```
#pragma DISABLE
uchar dfunc(uchar p1, uchar p2) {
 return(p1*p2+p2*p1)
}
```

- 指令名： FLOATFUZZY

缩写： FF

变量： 圆括号中的 0~7 的数字

属性： 一般控制

默认值： FLOATFUZZY(3)

功能： 指令 FLOATFUZZY 用于确定 FLOAT 型数据在执行之前的有效数字。

例子：

```
C51 MYFILE.C FLOATFUZZY(3)
#pragma FF(2)
```

- 指令名: INTERVAL

缩写: 无

变量: 圆括号中的为中断向量列表所设置的可选择的间距

属性: 首要控制

默认值: INTERVAL(8)

功能: 指令 INTERVAL 为中断向量设置大小。大小的规格一般符合 SIECO-51, 在该规格中将中断向量的间距定义为 3bit。所以在计算编译器的定位的绝对地址时可以使用公式 (interval\*n+offset+3); 其中, interval 是指令 INTERVAL 的参数值, 默认值为 8, n 是中断的号码, offset 是指令 INTVECT 的参数值, 默认值为 0。

例子:

```
C51 SAMPLE.C INTERVAL(2)
#pragma interval(2)
```

- 指令名: [NO]INTPROMOTE

缩写: [NO]IP

变量: 无

属性: 首要控制

默认值: INTPROMOTE

功能: 指令名 INTPROMOTE 将激活 ANSI 取整进位法则。在 IF 条件句中的参数在比较之前都已经由 ANSI 取整进位法则处理了。而命令 NOINTPROMOTE 将撤除取整进位法则。

例子:

```
C51 SAMPLE.C INTPROMOTE
#pragma interpromote
```

- 指令名: [NO]INTVECTOR

缩写: [NO]IV

变量: 圆括号中的可选择的中断向量表的偏移量

属性: 首要控制

默认值: INTVECTOR(0)

功能: 指令 INTVECTOR 为需要中断的函数生成中断向量。如果中断向量列表的开始地址不为零, 在圆括号中设置其地址偏移量。使用这个指令编译器会根据由指令 ROM 指定的程序存储器空间为指令 AJMP 和 LJMP 产生一个中断向量, 以便寻找中断入口。指令 NOINTVECTOR 将阻止产生中断向量列表, 以便使用者使用其他程序工具生成中断向量。

例子:

```
C51 SAMPLE.C INTVECTOR(0X8000H)
```

- 指令名: LARGE

缩写: LA

变量: 无

属性: 首要控制

默认值: SMALL

功能: 指令 LARGE 选择 LARGE 存储模式。在 LARGE 模式下, 所有的变量、程序的局部数据段和程序都被存储在 8051 系统的外部数据存储空间, 最大的字节可达 64bit。但是将常用

的数据放在内部存储空间能极大地改善系统的性能。

例子：

C51 SAMPLE.C LARGE

- 指令名：MAXARGS

缩写：MA

变量：圆括号中编译器为长度可变参量表准备的比特数

属性：首要控制

默认值：MAXARGS(15) SMALL 模式下

MAXARGS(40) LARGE 模式下

功能：使用该指令，使用者可以定义长度可变参数表中参数传递过程中缓冲器的大小。

指令 MAXARGS 中的参数表示的是可传递的最大的参数比特数。

下面的例子是一个简单的使用可变参量表的例子，这个例子必须使用 KEIL C51 自定义的 va\_list 类型，该类型的原型定义在头文件<stdarg.h>中。作者用这个例子来简单的说明 MAXARGS 命令的使用方法。

```
#pragma maxargs(4)
#include <stdarg.h>
void func(char type, ...) {
 va_list ptr;
 char c;
 int i;

 va_start(ptr, type);
 switch *type {
 case 0: //char 型
 c=va_arg(ptr, char);break;
 case 1: //int 型
 i= va_arg(ptr, int);break;
 }
}
void testfunc(void) {
 func(0, 'c'); //char 类型
 func(1, 0x1234); // int 类型
}
```

上面的例子中，第一句话就使用了命令 maxargs，这句话使最大参数被限定在 4 个以内。

- 指令名：[NO]MODA2

缩写：无

变量：无

属性：一般控制

默认值：NOMODA2

功能：指令名 MODA2 可以使编译器为 Atmel82x8252 及相当的附加的硬件生成代码，并

可使用额外数据指针以提高库函数 memcpy、memmove、memcmp、strcmp 和 strcpy 的效率。指令 NOMODA2 不允许编译器生成可使用额外数据指针的代码。

例子：

```
C51 SAMPLE.C MODA2
#pragma moda2
• 指令名：[NO]MODAB2
缩写：无
变量：无
属性：一般控制
默认值：NOMODAB2
```

功能：指令 MODAB2 可以使编译器为 AduC B2 系列及相当的附加的硬件生成代码，并可使用额外数据指针以提高库函数 memcpy、memmove、memcmp、strcmp 和 strcpy 的效率。指令 NOMODAB2 不允许编译器生成可使用额外数据指针的代码。

例子：

```
C51 SAMPLE.C MODAB2
#pragma modab2
• 指令名：[NO]MODDA
缩写：无
变量：无
属性：一般控制
默认值：NOMODDA
```

功能：指令 MODDA 可以使编译器生成支持 Dallas80c390, 80c400 和 5240 系列的算术加速器的代码。这样做有利于提高整数和长操作。

例子：

```
C51 SAMPLE390.C MODDA
#pragma modda
• 指令名：[NO]MODP2
缩写：无
变量：无
属性：一般控制
默认值：NOMODP2
```

功能：指令 MOD P2 可以使编译器为包含 PHILIPS 系列在内的相当的附加硬件生成代码，并可使用额外数据指针以提高库函数 memcpy、memmove、memcmp、strcmp 和 strcpy 的效率。指令 NOMODP2 不允许编译器生成可使用额外数据指针的代码。

例子：

```
C51 SAMPLE.C MODP2
#pragma modp2
• 指令名：[NO]MOD517
缩写：无
变量：无
```

属性：一般控制

默认值：NOMOD517

功能：指令 MOD517 可以使编译器为 Infineon C517 系列的附加硬件（即算术处理器和额外数据指针）生成可支持它们的代码，可使用额外数据指针以提高库函数 memcpy、memmove、memcmp、strcmp 和 strcpy 的效率。指令 NOMOD517 不允许编译器生成可使用额外数据指针的代码。

例子：

```
C51 SAMPLE517.C MOD517
```

- 指令名：[NO]MODDP2

缩写：无

变量：无

属性：一般控制

默认值：NOMODDP2

功能：指令 MODDP2 可以使编译器为 Dallas 80C320、520、C530、C550 系列及相当的附加的硬件生成代码，并可使用额外数据指针以提高库函数 memcpy、memmove、memcmp、strcmp 和 strcpy 的效率。指令 NOMODP2 不允许编译器生成可使用额外数据指针的代码。

例子：

```
C51 SAMPLE320.C MODDP2
```

```
#pragma moddp2
```

- 指令名：NOAMAKE

缩写：NOAM

变量：无

属性：首要控制

默认值：AUTOMAKE 信息被生成

功能：指令 NOAMAKE 不显示由编译器产生的 ATUOMAKE 目标文件记录（包括有关寄存器的优化信息）。

例子：

```
C51 SAMPLE320.C NOAMAKE
```

```
#pragma nomake
```

- 指令名：OBJECT, NOOBJECT

缩写：OJ, NOOJ

变量：括号内文件名

属性：首要控制

默认值：OBJECT (bname. OBJ)

功能：OBJECT 指令使编译器按照该指令后面括号内的变量名作为生成的目标文件名，默认的目标文件名是路径名、源文件名及扩展名 “. OBJ”。使用 NOOBJECT 指令将不产生目标文件。

例子：

```
C51 SAMPLE.C OBJECT(sample1.obj)
```

```
#pragma oj(sample_1.obj)
```

```
C51 SAMPLE.C NOOBJECT
```

```
#pragma nooj
```

- 指令名: OBJECTTEXTEND

缩写: OE

变量: 无

属性: 首要控制

默认值: No object-extensions

功能: OBJECTTEXTEND 指令使编译器所产生的目标文件中包含附加的变量类型定义信息,该信息可在目标文件中具有相同的名字时将其区分出来以供仿真器在调试中使用。当 C 语言源程序中使用了数组或者结构类型变量时,如果希望在 DS51 模拟调试器的观察窗口中观察完整的书组或者结构变量,则必须使用这条指令。

例子:

```
C51 SAMPLE.C OBJECTTEXTEND DEBUG
```

```
#pragma oe db
```

**注意:** 该指令所产生的目标文件包含 OMF-51 队在定位目标格式指定的一个超集。这种扩展的目标文件格式要求连接定位器 L51 具有 2.3 以上的版本, 仿真器必须具备增强的目标装入器。

- 指令名: OMF2

缩写: O2

变量: 无

属性: 首要控制

默认值: C51 的默认的形式是 OMF51, CX51 的默认形式是 OMF2

功能: 该指令能够激活 OMF2 输出文件形式, OMF2 提供了符号更加详细的有关于类型的信息并修补了 OMF51 形式的一些不足。如果想使用编译器 CX51 的下列功能时, 要使用 OMF2 命令: 可变的 BANKING: 指令 VARBANKING 允许使用远存储类型(FAR); XDATA ROM; RAM STRINGS; 邻近模式。而且 OMF2 形式要求连接定位器必须是 LX51 及其扩展版, 不能用在 BL51 上。

例子:

```
C51 SAMPLE.C OMF2
```

```
#pragma o2
```

- 指令名: ONEREGBANK

缩写: OB

变量: 无

属性: 一般控制

默认值: 无

功能: CX51 是选择 bank0 来作为无 USING 特性中断(即低优先级中断)的入口, 这步往往是在中断程序开始时通过指令 MOV PSW, #0 来完成的。这样就可以保证没有 USING 特性高优先级的中断打断使用其他 BANK 的低优先级中断。当应用程序只使用一个寄存 BANK 时, 就可以使用命令 ONEREGBANK, 该指令将消除 MOV PSW, #0 的作用。

例子:

```
C51 SAMPLE.C ONEREGBANK
```

```
#pragma ob
```

- 指令名: OPTIMIZE

缩写: OT

变量: 括号内一个 0~5 的十进制数, 另外可选 SIZE 或 SPEED 一决定优化重点是放在代码的长度上还是放在程序执行的速度上。

属性: 首要控制

默认值: OPTIMIZE (5, SPEED)

功能: OPTIMIZE 指令设置优化级, 共有 6 个优化级别, 高的优化级中包含前一个低优化级。

- 0 级优化 OPTIMIZE (0)

常数折叠: 编译时只要有可能, 编译器就执行包含常数的计算, 其中包括运行地址的计算。

简单访问优化: 对 8051 系统的内部数据和位地址进行访问优化。

跳转优化: 编译器总是将跳转延至最终目标上, 因此跳转到跳转的指令将被消除。

- 1 级优化 OPTIMIZE (1)

死码消除: 无用的代码段被消除。

跳转否决: 根据一个测试回溯, 条件跳转被仔细检查, 以决定是否能够简化或者消除。

- 2 级优化 OPTIMIZE (2)

数据覆盖: 适于静态覆盖的数据和位段被鉴别并标记出来。连接定位器 L51 通过对全局数据流的分析, 选择可静态覆盖的段。

- 3 级优化 OPTIMIZE (3)

“窥孔”优化: 将多余的 MOV 指令去掉, 包括不必要的从存储器装入对象及装入常数的操作。另外, 如果能节省存储器空间或程序执行时间, 复杂操作将由简单操作所取代。

- 4 级优化 OPTIMIZE (4)

寄存器变量: 使自动变量和参数传递变量尽可能位于寄存器中, 只要有可能, 将不为量保留数据存储器空间。

扩展访问优化: 来自 IDATA、XDATA 和 CODE 区域的变量直接包含在操作之中, 因此大多数时候没有必要将其装入中间寄存器。

局部公共子式消除: 如果表达式中有一个重复执行的计算, 第一次计算的结果, 只要有可能, 将会被用于后续的计算, 因此可从代码中消除繁杂的计算。

CASE/SWITC 语句优化: 将 CASE/SWITCH 语句作为跳转表或者跳转串优化。

- 5 级优化 OPTIMZE (5)

全局公共子式的消除: 只要有可能, 函数内部的相同的子表达式只计算一次。中间结一个寄存器以代替新的计算。

简单循环优化: 一常量占据一段内存的循环在运行的时候被优化。

- 6 级优化 OPTIMZE (6)

循环优化: 如果变更以后的程序更加有效率的话, 程序中的循环顺序会被改变。

- 7 级优化 OPTIMZE (7)

扩展访问优化: 用存储器或者变量来代替用 DPTR 来访问外部存储器。不管从指令执行的速度来说还是从代码的大小来说, 指针和数组的访问在这种情况下都会变得更有效率。

- 8 级优化 OPTIMZE (8)

共同部分合并：当对某个函数进行多次调用的时候，有可能某些固定的初始化代码会复使用，这样就可以减小程序的占用空间。

- 9 级优化 OPTIMZE (9)

程序子分支代替共同代码块：自动发现代码的共同部分，并以程序执行的子路径来代 Cx51 程序甚至会重新编排代码的执行顺序来进行这种优化。

OPTIMIZE (9) 包括从 0~8 级的所有优化。

注：全局优化从 4 级优化开始。同时，一个完整的函数在优化的时候，如果分配给生成优化代码所必需的数据结构的内存不够时，全局优化将只执行一部分或者根本不执行，并将显示如下错误信息：

```
***can't optimize function '<functionname>', no memory available
```

这时所产生的代码不是最优，但是是正确的。解决此错误的办法是删掉复杂的驱动文件或者减少函数的数量。

例子：

```
C51 SAMPLE.C OPTIMIZE(9)
C51 SAMPLE.C OPTIMIZE(0)
#pragma ot(6, SIZE)
#pragma ot(size)
```

- 指令名：ORDER

缩写：OR

变量：无

属性：首要控制

默认值：变量未被有序排列

功能：指令 ORDER 使编译器 CX51 按变量在源 C 程序中的定义顺序在存储空间排列。该指令可以消除 C 编译器使用的一些无用运算，但是这样也会导致编译器 CX51 的编译速度变慢。

例子：

```
C51 SAMPLE.C ORDER
#pragma order
```

- 指令名：REGFILE

缩写：RF

变量：圆括号中的文件名

属性：首要控制

默认值：无

功能：指令 REGFILE 使编译器使用定义寄存器文件以便于全局寄存器的优化使用。定义寄存器文件中指定了外部函数的寄存器使用情况。根据这些使用情况编译器可以优化使用普通寄存器的使用。

例子：

```
C51 SAMPLE.C REGFILE
```

```
#pragma regfile
```

- 指令名: REGISTERBANK

缩写: RB

变量: 括号内一个 0~3 的十进制数

属性: 一般控制

默认值: REGISTERBANK (0)

功能: 如果在 C 语言源程序中没有采用关键字 “using” 指定工作寄存器组, 则编译时由 registerbank 指令选择当前调用时使用的寄存器组。当能够计算寄存器的绝对数量时, 生成的代码为可以使用寄存器访问的绝对调用形式 (代码中用 “Arn” 表示)。

例子:

```
C51 SAMPLE.C REGISTERBANK(1)
#pragma rb(3)
```

下面是一个多次使用 REGISTERBANK 指令的实例, 在函数内使用时将会被忽略。

```
#pragma rb(2) //在函数 lyze 中使用
char lyze() {
//...
}
```

```
char lyze() {
 #pragma rb(2) //被忽略
//...
}
```

注意: 与使用 “using n” 相比, registerbank 指令不能转换寄存器组。另外有返回值的函数必须始终使用同一寄存器组, 否则函数可能会在执行过程或者返回值中出现错误。REGISTERBANK 指令可以作为#pragma 命令的参数在 C 语言源程序中出现多次, 但是如果在某一个函数或者过程中使用时将会被忽略。

- 指令名: REGPARMS/NOREGPARMS

缩写: 无

变量: 无

属性: 一般控制

默认值: REGPARMS

功能: 使用 REGPARMS 指令时, 编译器在寄存器中至多传递 3 个参数。这种类型的参数传递可与汇编语言相比。使用 NOREGPARMS 指令时, 参数在固定的存储区域传递, 这种传递方法也可以为不能装入寄存器的参数所使用。该指令产生的参数传递代码与早期的 C51 版本兼容。REGPARM 和 NOREGPARMS 指令可作为预处理命令#pragma 的参数在 C 语言源程序中使用多次, 从而允许程序的某一节使用寄存器来传递参数, 而其他函数仍然使用旧的参数传递方式, 汇编语言函数及原有的库文件则不必更改。

例子：

```
C51 SAMPLE.C NOREGPARMS
#pragma REGPARMS
```

下面是一个采用预处理命令#pragma 引用 REGPARMS 和 NOREGPARMS 指令的例子。这个例子比较简单，相信读者通过代码段的阅读就可以理解该命令的使用方法。

```
#pragma NOREGPARMS //使用旧的参数传递方式
extern int old_func(int, char);

#pragma REGPARMS //使用新的参数传递方式
extern int new_func(int, char);

main() {
 char a;
 int x1, x2;
 x1 = old_func(x2, a);
 x1 = new_func(x2, a);
}
```

- 指令名：ROM

缩写：无

变量：SMALL, COMPACT, LARGE

属性：首要控制

默认值：ROM (LARGE)

功能：ROM 指令用来决定程序空间的大小，它影响跳转指令的编码

- ROM (SMALL)：调用和转移以 ACALL 和 AJMP 指令编码，最大程序空间为 2kB 字节，整个用户程序的长度必须分布在这个 2kByte 字节的空间之内。
- ROM (COMPACT)：调用以 LCALL 指令编码，函数内的转移以 AJMP 指令编码。因此函数的长度不得超过 2kByte，而整个程序的长度可以达到 64kByte。这种应用必须视不同的需要确定，看它是否比 LARGE 模式更好。
- ROM (LARGE)：调用和转移以 LCALL 和 LJMP 指令编码，这样可以不加限制的使用整个地址空间，即用户程序可达 64kByte。

例子：

```
C51 SAMPLE.C ROM(SMALL)
#pragma ROM(SMALL)
```

- 指令名：RET\_PSTK\RET\_XSTK

缩写：RP\RXP

变量：无

属性：首要控制

默认值：无

功能：指令 RET\_PSTK 和指令 RET\_XSTK 将返回地址储存在 PDATA 或 XDATA 深堆栈中。通常的情况下返回的地址存储在 8051 的硬件堆栈中，这两个指令使编译器产生代码，产生的代码将返回的地址从硬件堆栈中推出并将其存储在指定的深堆栈中。其中，指令 RET\_PSTK 使用的是 COMOACT 模型的深堆栈，指令 RET\_XSTK 使用的是 LAGRE 模型的深堆栈。

例子：

```
C51 SAMPLE.C RET_PSTK
```

注意：指令 RET\_PSTK 和指令 RET\_XSTK 可以用来将返回地址从片上堆栈或硬件堆栈中推出。但是在那些包含最深堆栈的模型中要有选择性的使用这些命令。在使用这些命令时必须在启动代码中初始化深堆栈的指针。

- 指令名：SAVE/RESTORE

缩写：无

变量：无

属性：一般控制

默认值：无

功能：SAVE 指令保存当前的 ARGES、REGPARMS、OPTIMIZE 因子和优化选项 SIZE（代码长度）以及 SPEED（执行速度）的设置。RESTORE 指令从 SAVE 栈中取出最近一个 SAVE 指令所保存的值。SAVE 指令最大的嵌套深度为 8。

例子：

```
#pragma save
#pragma noregparms
extern void test1(char c, int i);
extern char test2(long l, float f);
#pragma restore
```

注意：SAVE/RESTORE 指令只能在源程序文件中以#pragma 的参数形式出现，不能用于 DOS 命令行。

- 指令名：INVECTOR

缩写：IV, NOIV

变量：无

属性：首要控制

默认值：INVECTOR

功能：INVECTOR 指令使编译器对 8051 单片机的中断产生一个 3Byte 的跳转指令(LJMP)，转移到中断向量目标值。这些中断向量放在绝对地址“8n+3”处，n 为中断号。NOINVECTOR 指令禁止产生这些向量地址，从而用户可以利用其他方法（比方说在汇编程序中）编写中断程序并确定中断向量的入口地址。

例子：

```
C51 SAMPLE.C INTVECTOR(0x8000)
#pragma iv(0x8000)
```

```
C51 SAMPLE.C NOINTVECTOR
#pragma noiv
```

## 6.3 C51的高级配置文件

本节讨论一些在使用 KEILC51 时的高级的技巧。本节所讨论的内容对于编出一个完整的 C51 程序来说并不是必不可少的，但是，读者或许需要对编译的某些内容作些自定义的修改，比方说与 PL/M-51 的程序接口等。本节就是专为这些希望自定义标准的程序员所准备的。

本节主要讨论的内容包括启动文件设置、数据和代码段命名格式、与汇编和 PL/M-51 程序自定义接口、修改各种数据类型的存储格式等等。

C51 编译器为用户准备了一系列的源文件，这些源文件可以被按照特殊的硬件需求修改以后使用。这些文件分别为：启动初始化代码 (STARTUP.A51)、变量初始化代码 (INIT.A51)、底层 I/O 流操作代码 (GETKEY.C 和 PUTCHAR.C)。为动态内存分配而准备的源代码程序则包括 CALLOC.C、FREE.C、INIT\_MEM.C、MALLOC.C 以及 REALLOC.C。这些源文件已经是被编译过的，并且包含在 C51 的库函数中，并可以被直接调用。

要想包含某些特定的启动和初始化程序，读者必须把对它们使用 linker 命令行。下面的命令就是要将原来的程序替换为自定义的 STARTUP.A51 和 PUTCHAR.C。

```
BL51 MYMODUL1.OBJ, MYMODUL2.OBJ, STARTUP.OBJ, PUTCHAR.OBJ
```

读者已经清楚了可能在使用中需要修改哪些文件，那么现在将分别来对这些文件进行说明。

### 6.3.1 目标程序启动配置文件——STARTUP.A51

STARTUP.A51 文件中包含了 C51 目标程序的启动代码。如果要在目标程序中使用定制的启动代码，那么就把 LIB 目录下的这个文件拷贝到目标目录中，并添加到目标项目里。

这个文件产生的代码会在系统复位以后立刻开始按顺序执行以下一条或多条任务：

- (1) 清空内部数据存储区。
- (2) 清空外部数据存储区。
- (3) 清空分页的外部数据存储区。
- (4) 初始化 SMALL/COMPACT/LARGE 模式的重入函数的堆栈和指针。
- (5) 初始化 8051 硬件的堆栈指针。
- (6) 将程序控制权交给 C 程序的 main 主函数。

STARTUP.A51 文件提供一些编译的常量，读者也可以根据需要做一些修改。这些常量的名称及含义如表 6-3 所示。

表 6-3

常量的名称及含义

| 常量名 | 说明 |
|-----|----|
|-----|----|

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IDATALEN                | 声明 51 单片机系统开始运行时有多少内部 RAM 字节需要用 0 进行初始化，默认值为 80H。对于由 256 字节内部 RAM 的 51/52 系列单片机，可以使用 100H。当用户程序在开始时需要用 0 进行初始化内部 RAM 时，才有必要对 IDATALEN 进行赋值。如果希望内部 RAM 具有掉电保护的功能，则应该将 IDATALEN 的值置为 0。在这种情况下至少得保持位于段?C_LIB_DATA 和段?C_LIB_DBIT 中的变量都置为 0，否则有些库函数将不能正常运行。?C_LIB_DATA 段的长度因不用的应用而不同，其当前长度可以在 L51 产生的 MAP 文件中找到                                                                                                                                                                                                    |
| XDATASTART              | 需要以 0 初始化的 XDATA 存储器的首地址                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| XDATALEN                | 需要以 0 初始化的 XDATA 存储器的字节数                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| PDATASTART              | 需要以 0 初始化的 PDATA 存储器的首地址                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| PDATALEN                | 需要以 0 初始化的 PDATA 存储器的字节数                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| IBPSTACK<br>IBPSTACKTOP | 定义在 SMALL 编译模式下创建的再入函数的模拟栈区。IBPSTACK=1 时创建模拟堆栈并对堆栈指针（变量?C_IBP）进行初始化，IBPSTACK=0 时不创建模拟堆栈。IBPSTACKTOP 指出栈顶首地址。对于具有 256Byte 内部 RAM 的 52 系列单片机来说，当使用地址为 OFFH 的内部存储区作为堆栈时，可以不做初始化。注意：C51 编译器对于该栈是否能满足特定要求不作检查，用户必须自己进行测试                                                                                                                                                                                                                                                                                           |
| XBPSTACK<br>XBPSTACKTOP | 定义在 LARGE 编译模式下创建的再入函数的模拟栈区。XBPSTACK=1 时创建模拟堆栈并对堆栈指针（变量?C_XBP）进行初始化，XBPSTACK=0 时不创建模拟堆栈。XBPSTACKTOP 指出栈顶首地址。如果使用首地址为 OFFFH 的 XDATA 存储器区域作为该栈时，可以不做初始化。注意：C51 编译器对于该栈是否能满足特定要求不作检查，用户必须自己进行测试                                                                                                                                                                                                                                                                                                                  |
| PBPSTACK<br>PBPSTACKTOP | 定义在 COMPACT 编译模式下创建的再入函数的模拟栈区。PBPSTACK=1 时创建模拟堆栈并对堆栈指针（变量?C_PBP）进行初始化，PBPSTACK=0 时不创建模拟堆栈。PBPSTACKTOP 指出栈顶首地址。如果使用首地址为 OFFH 的 PDATA 存储器区域作为该栈时，可以不做初始化。注意：C51 编译器对于该栈是否能满足特定要求不作检查，用户必须自己进行测试                                                                                                                                                                                                                                                                                                                 |
| PPAGEENABLE<br>PPAGE    | 在 COMPACT 编译模式下使用页寻址方式操作 XDATA 存储器区域时需要用到这两条指令。对于 LARGE 编译模式，使用这些指令可以提高程序的运行速度或者减少程序代码长度。PPAGEENABLE 允许对 8051 系列单片机端口 2 的初始化，以实现对 XDATA 存储器空间的页寻址操作。PPAGEENABLE 和 PPAGE 必须与连接定位器 L51 的控制指令 PDATA 一起使用。L51 的 PDATA 指令用来指定 XDATA 存储器中 PDATA 区的首地址。例如在 STARTUP.A51 中将 PPAGEENABLE 设为 1，PPAGE 设为 10H，这是 PDATA 存储器区域的首地址将为 1000H（即 10H 页，用 L51 进行连接定位时必须使用 PDATA 指令，并且 PDATA 指令的参数值必须在 1000H 与 10FFH 之间，例如：L51 <输入模块>PDATA(1050H) 注意，C51 编译器和 L51 连接定位器都不对 PPAGE 和 PDATA 指令的正确与否进行检测，用户必须保证 PPAGE 和 PDATA 包含一个合适的值 |

下面将 STARTUP.A51 源文件分块解读如下：

首先定义了各类存储空间的大小和地址范围，如下：

```
IDATALEN EQU 80H ; the length of IDATA memory in bytes.
;
XDATASTART EQU 0H ; the absolute start-address of XDATA memory
XDATALEN EQU 0H ; the length of XDATA memory in bytes.
;
PDATASTART EQU 0H ; the absolute start-address of PDATA memory
PDATALEN EQU 0H ; the length of PDATA memory in bytes.
```

上面的代码表示 idata 类型的存储空间为 80H 个字节，xdata 和 pdata 类型的存储器起始地址均为 0，大小也为 0。读者可以根据自己的系统的配置对这些参数直接进行修改。

接下来，STARTUP.A51 会定义一些供重入函数（使用 reentrant 关键字）参数调用时使用的堆栈空间，如下：

```
; Stack Space for reentrant functions in the SMALL model.
IBPSTACK EQU 0 ; set to 1 if small reentrant is used.
IBPSTACKTOP EQU OFFH+1 ; set top of stack to highest location+1.
;
; Stack Space for reentrant functions in the LARGE model.
XBPSTACK EQU 0 ; set to 1 if large reentrant is used.
XBPSTACKTOP EQU OFFFFH+1; set top of stack to highest location+1.
;
; Stack Space for reentrant functions in the COMPACT model.
PBPSTACK EQU 0 ; set to 1 if compact reentrant is used.
PBPSTACKTOP EQU OFFFFFH+1; set top of stack to highest location+1.
```

上面的代码中，IBPSTACK 如果为 1 表示使用 SMALL 模式下重入堆栈；XBPSTACK 如果为 1 表示使用 LARGE 模式下的重入堆栈；PBPSTACK 如果为 1 则表示使用 COMPACT 模式下的重入堆栈。而后面的“?BPSTACKTOP”参数则为设置重入堆栈的大小所用。

然后，STARTUP.A51 将进行一些名字的定义和“?C\_C51STARTUP”段和“?STACK”段的类型的定义。

```
NAME ?C_STARTUP
?C_C51STARTUP SEGMENT CODE
?STACK SEGMENT IDATA

RSEG ?STACK
DS 1

EXTRN CODE (?C_START)
PUBLIC ?C_STARTUP
```

接着，是用 AT 命令将程序定位在 0 起始的地址上，一般来说，单片机在启动的时候就会从地址为 0 的地方读取指令，所以这就是单片机程序启动的位置。在这里使用了 LJMP STARTUP1 将程序跳转到了初始化代码段的位置。当然，如果是在调试模式下，则在这里可能就不是将程序定位在 0 起始处，而要根据调试器的要求来进行初始代码的定位。

```
CSEG AT 0
?C_STARTUP: LJMP STARTUP1
```

接下来就是标准的初始化代码，进行清空内部、外部以及分页数据存储区的作用。代码如下：

```
RSEG ?C_C51STARTUP
STARTUP1:
IF IDATALEN <> 0
 MOV R0, #IDATALEN - 1
```

```

 CLR A
IDATALOOP: MOV @R0, A
 DJNZ R0, IDATALOOP
ENDIF

IF XDATALEN <> 0
 MOV DPTR, #XDATASTART
 MOV R7, #LOW(XDATALEN)
 IF (LOW(XDATALEN)) <> 0
 MOV R6, #(HIGH(XDATALEN)) +1
 ELSE
 MOV R6, #HIGH(XDATALEN)
 ENDIF
 CLR A
XDATALOOP: MOVX @DPTR, A
 INC DPTR
 DJNZ R7, XDATALOOP
 DJNZ R6, XDATALOOP
ENDIF

IF PPAGEENABLE <> 0
 MOV P2, #PPAGE
ENDIF

IF PDATALEN <> 0
 MOV R0, #PDATASTART
 MOV R7, #LOW(PDATALEN)
 CLR A
PDATALOOP: MOVX @R0, A
 INC R0
 DJNZ R7, PDATALOOP
ENDIF

```

上面的工作完成以后就开始初始化 SMALL/COMPACT/LARGE 模式的重入函数的堆栈和指针。

```

IF IBPSTACK <> 0
EXTRN DATA (?C_IBP)
 MOV ?C_IBP, #LOW IBPSTACKTOP
ENDIF

IF XBPSTACK <> 0

```

```

EXTRN DATA (?C_XBP)

 MOV ?C_XBP, #HIGH XBPSTACKTOP
 MOV ?C_XBP+1, #LOW XBPSTACKTOP
ENDIF

IF PBPSTACK <> 0
EXTRN DATA (?C_PBP)
 MOV ?C_PBP, #LOW PBPSTACKTOP
ENDIF

```

然后初始化8051硬件的堆栈指针:

```
MOV SP, #?STACK-1
```

如果有banking的设置，则还有以下语句来调用初始化banking的初始化代码:

```

EXTRN CODE (?B_SWITCH0)
 CALL ?B_SWITCH0 ; init bank mechanism to code bank 0

```

将控制权交给C程序的main主函数:

```
LJMP ?C_START
END
```

为读者阅读方便，下面附上完整的STARTUP.A51的源文件:

```

;-----;
; This file is part of the C51 Compiler package
; Copyright(c) 1988-1999 KEILElektronik GmbH and KEILSoftware, Inc.
;-----;

; STARTUP.A51: This code is executed after processor reset.

;

; To translate this file use A51 with the following invocation:

;

; A51 STARTUP.A51

;

; To link the modified STARTUP.OBJ file to your application use the following
; BL51 invocation:

;

; BL51 <your object file list>, STARTUP.OBJ <controls>

;

;-----;

;

; User-defined Power-On Initialization of Memory

;

; With the following EQU statements the initialization of memory
; at processor reset can be defined:

```

```

;
; ; the absolute start-address of IDATA memory is always 0
IDATALEN EQU 80H ; the length of IDATA memory in bytes.
;

XDATASTART EQU 0H ; the absolute start-address of XDATA memory
XDATALEN EQU 0H ; the length of XDATA memory in bytes.
;

PDATASTART EQU 0H ; the absolute start-address of PDATA memory
PDATALEN EQU 0H ; the length of PDATA memory in bytes.
;

; Notes: The IDATA space overlaps physically the DATA and BIT areas of the
; 8051 CPU. At minimum the memory space occupied from the C51
; run-time routines must be set to zero.
;

;
;

; Reentrant Stack Initialization
;

; The following EQU statements define the stack pointer for reentrant
; functions and initialized it:
;

; Stack Space for reentrant functions in the SMALL model.
IBPSTACK EQU 0 ; set to 1 if small reentrant is used.
IBPSTACKTOP EQU OFFH+1 ; set top of stack to highest location+1.
;

; Stack Space for reentrant functions in the LARGE model.
XBESTACK EQU 0 ; set to 1 if large reentrant is used.
XBESTACKTOP EQU OFFFFH+1; set top of stack to highest location+1.
;

; Stack Space for reentrant functions in the COMPACT model.
PBPSTACK EQU 0 ; set to 1 if compact reentrant is used.
PBPSTACKTOP EQU OFFFFH+1; set top of stack to highest location+1.
;

;
;

; Page Definition for Using the Compact Model with 64 KByte xdata RAM
;

; The following EQU statements define the xdata page used for pdata
; variables. The EQU PPAGE must conform with the PPAGE control used
; in the linker invocation.
;

```

```

PPAGEENABLE EQU 0 ; set to 1 if pdata object are used.
PPAGE EQU 0 ; define PPAGE number.

;

```

NAME ?C\_STARTUP

```

?C_C51STARTUP SEGMENT CODE
?STACK SEGMENT IDATA

RSEG ?STACK
DS 1

EXTRN CODE(?C_START)
PUBLIC ?C_STARTUP

CSEG AT 0
?C_STARTUP: LJMP STARTUP1

RSEG ?C_C51STARTUP

STARTUP1:

IF IDATALEN <> 0
 MOV R0, #IDATALEN - 1
 CLR A
 IDATALOOP: MOV @R0, A
 DJNZ R0, IDATALOOP
ENDIF

IF XDATALEN <> 0
 MOV DPTR, #XDATASTART
 MOV R7, #LOW(XDATALEN)
 IF (LOW(XDATALEN)) <> 0
 MOV R6, #(HIGH(XDATALEN)) +1
 ELSE
 MOV R6, #HIGH(XDATALEN)
 ENDIF
 CLR A

```

```

XDATALOOP: MOVX @DPTR, A
 INC DPTR
 DJNZ R7, XDATALOOP
 DJNZ R6, XDATALOOP
ENDIF

IF PPAGEENABLE <> 0
 MOV P2, #PPAGE
ENDIF

IF PDATALEN <> 0
 MOV R0, #PDATASTART
 MOV R7, #LOW(PDATALEN)
 CLR A
PDATALOOP: MOVX @R0, A
 INC R0
 DJNZ R7, PDATALOOP
ENDIF

IF IBPSTACK <> 0
EXTRN DATA(?C_IBP)

 MOV ?C_IBP, #LOW IBPSTACKTOP
ENDIF

IF XBPSTACK <> 0
EXTRN DATA(?C_XBP)

 MOV ?C_XBP, #HIGH XBPSTACKTOP
 MOV ?C_XBP+1, #LOW XBPSTACKTOP
ENDIF

IF PBPSTACK <> 0
EXTRN DATA(?C_PBP)

 MOV ?C_PBP, #LOW PBPSTACKTOP
ENDIF

 MOV SP, #?STACK-1
EXTRN CODE(?B_SWITCH0)
 CALL ?B_SWITCH0 ; init bank mechanism to code bank 0

```

```
LJMP ?C_START
```

```
END
```

### 6.3.2 CPU 初始化文件——START751.A51

START751.A51 文件是专门为 Signetics 8xC751 系列 CPU 的初始化文件。该文件的使用方法与 STARTUP.A51 基本相同，不过由于 8xC751 不可以访问 2kByte 以上的程序空间，也不可以访问外部的 RAM，所以程序中没有 XDATA 和 PDATA 部分。

该程序的源代码如下：

```
;-----
; This file is part of the C51 Compiler package
; Copyright(c) 1988-1999 KEILElektronik GmbH and KEILSoftware, Inc.
;
; START751.A51: This code is executed after processor reset.
;
; To translate this file use A51 with the following invocation:
;
; A51 START751.A51
;
; To link the modified START751.OBJ file to your application use the following
; BL51 invocation:
;
; BL51 <your object file list>, START751.OBJ <controls>
;
;
;-----
;
; User-defined Power-On Initialization of Memory
;
; With the following EQU statements the initialization of memory
; at processor reset can be defined:
;
; ; the absolute start-address of IDATA memory is always 0
IDATALEN EQU 40H ; the length of IDATA memory in bytes.
;
;
; Notes: The IDATA space overlaps physically the DATA and BIT areas of the
; 8051 CPU. At minimum the memory space occupied from the C51
; run-time routines must be set to zero.
;
```

```

;
; Reentrant Stack Initialization
;

; The following EQU statements define the stack pointer for reentrant
; functions and initialized it:
;

; Stack Space for reentrant functions in the SMALL model.
IBPSTACK EQU 0 ; set to 1 if small reentrant is used.
IBPSTACKTOP EQU OFFH+1 ; set top of stack to highest location+1.
;

;-----

NAME ?C_STARTUP

?C_C51STARTUP SEGMENT CODE
?STACK SEGMENT IDATA

 RSEG ?STACK
 DS 1

 EXTRN CODE(?C_START)
 PUBLIC ?C_STARTUP

 CSEG AT 0
?C_STARTUP: AJMP STARTUP1

 RSEG ?C_C51STARTUP

STARTUP1:

 IF IDATALEN <> 0
 MOV R0, #IDATALEN - 1
 CLR A
 IDATALOOP: MOV @R0, A
 DJNZ R0, IDATALOOP
 ENDIF

 IF IBPSTACK <> 0

```

```

EXTRN DATA (?C_IBP)

 MOV ?C_IBP, #LOW IBPSTACKTOP
ENDIF

 MOV SP, #?STACK-1
AJMP ?C_START

END

```

与 STARTUP.A51 比较可以看出，虽然上面的程序看起来比 STARTUP.A51 要短的多，但是实际上两个程序的结构是基本一样的，缺少的代码行仅仅是关于 XDATA 和 PDATA 的初始化语句。

### 6.3.3 静态变量初始化文件——INIT.A51

INIT.A51 文件是关于初始化程序中明确说明的静态变量而用的。如果系统中使用了看门狗，那么可以使用 watchdog 宏来进行设置。不过只有在初始化程序执行时间长于看门狗设置的最大时间的时候才会使用到这个宏。如果读者使用的是 80515，那么这个宏可以按照如下方式来使用：

```

WATCHDOG MACRO
SETB WDT
SETB SWDT
ENDM

```

提醒读者注意的是，如果使用了 INIT.A51 文件，那么就应该将这个文件放在项目文件中的最后一个，否则有可能出现异常。

就像分析 STARTUP.A51 程序时所做的一样，下面将 INIT.A51 程序分块解读。

首先是看门狗（WATCHDOG）的宏定义，该宏只有一个结构，内容是空出来的，读者在使用的时候根据自己的硬件结构以及需要来填写其中的内容。该宏定义代码如下：

```

WATCHDOG MACRO
; Include any Watchdog refresh code here
ENDM
;
;
```

接下来是芯片类型的选择和 far 类型的变量初始化设定：

```

$SET (XBANK = 0)
$SET (DS390 = 0)

```

然后是 ACC、DPL、DPH 的地址宏定义：

```

ACC DATA OEOH

```

```
DPL DATA 82H
DPH DATA 83H
```

这些都完成以后就是最后一个需要定义的内容，为程序名和代码段定义：

```
NAME ?C_INIT

?C_C51STARTUP SEGMENT CODE
?C_INITSEG SEGMENT CODE ; Segment with Initializing Data
```

程序应该从“?C\_START”开始运行，该部分首先将需要初始化的变量空间的地址放在 DPTR 中。在 C51 程序中，所有需要在 INIT.A51 中初始化的变量均被放在“?C\_INITSEC”段中，这些变量虽然有可能在不同的文件中定义，但是在连接的时候会将它们合为一个段。在这个段的末尾，将出现一个值为 0 的字节。将 DPTR 赋值的代码如下：

```
MOV DPTR, #?C_INITSEG
```

然后需要设置看门狗的参数，并查看需要 INIT.A51 初始化的变量的格式。在 C51 中，所有存放在?C\_INITSEC 的变量都应该有如下格式：用一个字节（不妨称该字节为命令字节）的高两个比特来表示类型（TYPE），第 5 个比特用来表示是否是 BIG 类型，剩下的  $0^{\sim}4$ （总共 5 个比特）用来表示数据的长度（LENGTH）。

因为类型用两个比特来表示，所以总共可以表示 4 种类型，分别介绍如下：

- 类型值为 0 时，表示是 idata 类型的变量，命令字节后为 1 个字节的地址，再后面为需要初始化的值的内容，这些内容的长度由 5 个比特的（LENGTH）的值来决定。
- 类型值为 1 时，表示是 xdata 类型的变量，命令字节后为 2 个字节的地址，再后面为需要初始化的值的内容，这些内容的长度由 5 个比特的（LENGTH）的值来决定。
- 类型值为 2 时，表示是 pdata 类型的变量，命令字节后为 1 个字节的地址，再后面为需要初始化的值的内容，这些内容的长度由 5 个比特的（LENGTH）的值来决定。
- 类型值为 3 时，如果 BIG（命令字节的第 5 比特）为 0，表示下面的每一个字节为一个比特变量初始化的控制命令，其中，比特变量初始化命令的格式为：最高比特表示该比特变量的值，剩下的 7 个比特用来表示比特地址；如果 BIG（命令字节的第 5 比特）为 1，表示下面的一个字节仍然为命令字节，用来表示长度，接下来是 3 个字节的地址（far 类型变量），随后的字节则是要初始化的值，字节数由 LENGTH 来决定。

可以想到，喂狗、查看?C\_INITSEG 中的命令字节根据命令字节初始化相应的变量显然应该是一个循环，退出这个循环的条件应该为新的命令字节为 0。

实现这一部分功能的代码段如下：

```
Loop:
 WATCHDOG
 CLR A
 MOV R6, #1
 MOVC A, @A+DPTR
 JZ INITEND
```

```

 INC DPTR
 MOV R7, A
 ANL A, #3FH
 JNB ACC. 5, NOBIG
 ANL A, #01FH
 MOV R6, A
 CLR A
 MOVC A, @A+DPTR
 INC DPTR
 JZ NOBIG
 INC R6
NOBIG: XCH A, R7

```

上面的代码段中，JZ INITEND 就完成了循环条件判断的逻辑，而 ANL A, #3FH 完成了判断 BIG 及 LENGTH 的逻辑，如果 BIG 为 1，那么 ANL A, #01FH 完成了得到长度的逻辑。

上面的代码段执行完毕以后，程序判断是否是 far 类型的变量，

```

; ---- Init for far Variables
$IF (XBANK = 1)
EXTRN CODE (?C?CSTPTR)

 ANL A, #0EOH
 CJNE A, #0EOH, NOHDATA
;

HPTRINIT: CLR A
 MOVC A, @A+DPTR
 MOV R3, A
 INC DPTR
 CLR A
 MOVC A, @A+DPTR
 MOV R2, A
 INC DPTR
 CLR A
 MOVC A, @A+DPTR
 MOV R1, A
 INC DPTR
HLOOP: CLR A
 MOVC A, @A+DPTR
 PUSH DPH
 PUSH DPL
 CALL ?C?CSTPTR
 POP DPL

```

```

 POP DPH
 INC DPTR
 INC R1
 MOV A, R1
 JNZ HINC
 INC R2
HINC: DJNZ R7, HLOOP
 DJNZ R6, HLOOP
 SJMP Loop
NOHDATA:
$ENDIF

```

接下来判断其他类型：使用语句 ANL A, #0COH，并进行判定，如果是类型 1, 3，那么就跳转到 IorPData，这部分专门处理 idata 和 pdata 类型的初始化，否则看是否是 bit 类型，如果再不是，那么就是 xdata 的处理程序。这部分的逻辑控制语句如下：

```

ANL A, #0COH ; Typ is in Bit 6 and Bit 7
ADD A, ACC
JZ IorPData
JC Bits

```

对于各种类型的数据的处理，由于语句繁长，但逻辑简单，在这里就不说了，不过为方便有兴趣的读者，下面附上完整的 INIT.A51 程序的源代码：

```

$NOMOD51
;
;-----;
; This file is part of the C51 Compiler package
; Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
;-----;
; INIT.A51: This code is executed, if the application program contains
; initialized variables at file level.
;
;
; If you are using uVision2, just add the file as last file to your project.
; *** IMPORTANT NOTE ***: this file needs to be the last file of the linker
; input list. If you are using uVision2 this file should be therefore the
; last file in your project tree.
;
;
;
; To translate this file use Ax51 with the following invocation:
;
; Ax51 INIT.A51
;
;
; To link the modified INIT.OBJ file to your application use the following
; linker invocation:

```

```
;
; Lx51 <your object file list>, INIT.OBJ <controls>
;
;
;
;
; User-defined Watch-Dog Refresh.
;
;
; If the C application contains many initialized variables uses a watchdog
; it might be possible that the user has to include a watchdog refresh into
; the initialization process. The watchdog refresh routine can be included
; in the following MACRO and can alter all CPU registers except
; DPTR.
;
;
WATCHDOG MACRO
 ; Include any Watchdog refresh code here
 ENDM
;
;
;
;
; Far Memory Support
;
;
; If the C application contains variables in the far memory space that are
; initialized, you need to set the following define to 1.
;
;
; --- Set XBANK = 1 when far variables should be initialized
$SET (XBANK = 0)
;
;
;
;
; Dallas 390/400/5240 CPU Contigious Mode
;
;
; If you are using the Dallas Contigious Mode you need to set the following
; define to 1.
;
;
; --- Set DS390 = 1 when CPU runs in Dallas Contigious Mode
$SET (DS390 = 0)
;
;
;
;
; Standard SFR Symbols
```

```

ACC DATA 0EOH
DPL DATA 82H
DPH DATA 83H

NAME ?C_INIT

?C_C51STARTUP SEGMENT CODE
?C_INITSEG SEGMENT CODE ; Segment with Initializing Data

INIT_IorP MACRO
IorPData: ; If CY=1 PData Values
 CLR A
 MOVC A, @A+DPTR
 INC DPTR
 MOV R0, A ; Start Address
IorPLoop: CLR A
 MOVC A, @A+DPTR
 INC DPTR
 JC PData
 MOV @R0, A
 SJMP Common
PData: MOVX @R0, A
Common: INC R0
 DJNZ R7, IorPLoop
 JMP Loop
 ENDM

EXTRN CODE (MAIN)
PUBLIC ?C_START

RSEG ?C_C51STARTUP
INITEND: LJMP MAIN

$IF (XBANK = 0)
 INIT_IorP
$ENDIF

Bits: CLR A

```

|           |          |                                   |
|-----------|----------|-----------------------------------|
|           | MOVC     | A, @A+DPTR                        |
|           | INC      | DPTR                              |
|           | MOV      | R0, A                             |
|           | ANL      | A, #007H                          |
|           | ADD      | A, #Table-LoadTab                 |
|           | XCH      | A, R0                             |
|           | CLR      | C                                 |
|           | RLC      | A ; Bit Condition to Carry        |
|           | SWAP     | A                                 |
|           | ANL      | A, #00FH                          |
|           | ORL      | A, #20H ; Bit Address             |
|           | XCH      | A, R0 ; convert to Byte Addressen |
|           | MOVC     | A, @A+PC                          |
| LoadTab:  | JC       | Setzen                            |
|           | CPL      | A                                 |
|           | ANL      | A, @R0                            |
|           | SJMP     | BitReady                          |
| Setzen:   | ORL      | A, @R0                            |
| BitReady: | MOV      | @R0, A                            |
|           | DJNZ     | R7, Bits                          |
|           | SJMP     | Loop                              |
| Table:    | DB       | 00000001B                         |
|           | DB       | 00000010B                         |
|           | DB       | 00000100B                         |
|           | DB       | 00001000B                         |
|           | DB       | 00010000B                         |
|           | DB       | 00100000B                         |
|           | DB       | 01000000B                         |
|           | DB       | 10000000B                         |
| ?C_START: |          |                                   |
|           | MOV      | DPTR, #?C_INITSEG                 |
| Loop:     |          |                                   |
|           | WATCHDOG |                                   |
|           | CLR      | A                                 |
|           | MOV      | R6, #1                            |
|           | MOVC     | A, @A+DPTR                        |
|           | JZ       | INITEND                           |

```

 INC DPTR
 MOV R7, A
 ANL A, #3FH
 JNB ACC. 5, NOBIG
 ANL A, #01FH
 MOV R6, A
 CLR A
 MOVC A, @A+DPTR
 INC DPTR
 JZ NOBIG
 INC R6
NOBIG: XCH A, R7

; ----- Init for far Variables
$IF (XBANK = 1)
EXTRN CODE (?C?CSTPTR)
 ANL A, #0EOH
 CJNE A, #0EOH, NOHDATA
;
HPTRINIT: CLR A
 MOVC A, @A+DPTR
 MOV R3, A
 INC DPTR
 CLR A
 MOVC A, @A+DPTR
 MOV R2, A
 INC DPTR
 CLR A
 MOVC A, @A+DPTR
 MOV R1, A
 INC DPTR
HLOOP: CLR A
 MOVC A, @A+DPTR
 PUSH DPH
 PUSH DPL
 CALL ?C?CSTPTR
 POP DPL
 POP DPH
 INC DPTR
 INC R1

```

```

 MOV A, R1
 JNZ HINC
 INC R2
HINC: DJNZ R7, HLOOP
 DJNZ R6, HLOOP
 SJMP Loop

NOHDATA:
$ENDIF

 ANL A, #0COH ; Typ is in Bit 6 and Bit 7
 ADD A, ACC
 JZ IorPData
 JC Bits

XdataMem: CLR A
 MOVC A, @A+DPTR
 INC DPTR
 MOV R2, A ; High
 CLR A
 MOVC A, @A+DPTR
 INC DPTR
 MOV R0, A ; LOW

XLoop: CLR A
 MOVC A, @A+DPTR
 INC DPTR
 XCH A, R0
 XCH A, DPL
 XCH A, R0
 XCH A, R2
 XCH A, DPH
 XCH A, R2

$IF (DS390)
DPX DATA 93H
EXTRN CODE (?C?XDATASEG)
 MOV DPX, #BYTE0 (?C?XDATASEG)
$ENDIF

 MOVX @DPTR, A

$IF (DS390)
EXTRN CODE (?C?CODESEG)
 MOV DPX, #BYTE0 (?C?CODESEG)
$ENDIF

```

```

 INC DPTR
 XCH A, R0
 XCH A, DPL
 XCH A, R0
 XCH A, R2
 XCH A, DPH
 XCH A, R2
 DJNZ R7, XLoop
 DJNZ R6, XLoop
 SJMP Loop

$IF (XBANK = 1)
 INIT_IorP
$ENDIF

 RSEG ?C_INITSEG
 DB 0

;

;-----+
; STRUCTURE OF THE INITIALIZATION INFORMATION
;-----+
;
; This section describes the initialization data generated by C51 for
; explicit variable initializations (in segment ?C_INITSEC).
;
; Explicit variable initializations at C source level are stored by C51 in
; the segment ?C_INITSEC. All partial segments are combined at linker level
; to one segment. The segment end value DB 0 is taken from this library module
; INIT.A51.
;
; Structure of the ?C_INITSEC information:
;
; <Info> (see below) [BYTE] -----+ repeated
; <additional info> [BYTES depend on Info] -----+ repeated
; 0x00 [BYTE] <end of list mark>
;
; <Info> has the following format:
;
; Bit 7 6 5 4 3 2 1 0
; <Info> T T B L L L L L T=Type B=BIGBIT L=LENGTH
;
; If BIGBIT is set, another LENGTH BYTE FOLLOWS. The LENGTH

```

```

; info of the first byte is then the HIGH part.

;

; Typ is one of the following:
; 0 := IDATA init values; the following bytes follow:
; - 1 byte address
; - init data bytes according LENGTH specification
;

; 1 := XDATA init values; the following bytes follow:
; - 2 byte address (high byte first)
; - init data bytes according LENGTH specification
;

; 2 := PDATA init values; the following bytes follow:
; - 1 byte address
; - init data bytes according LENGTH specification
;

; 3, BIGBIT=0 := BIT init values; the following bytes follow:
; - 1 byte for each bit according LENGTH specification
; this byte has the following format:
;

; Bit 7 6 5 4 3 2 1 0
; I B B B B B B I := state of the bit
; B := bit address
;

; 3, BIGBIT=1 := HDATA init values; the following bytes follow:
; - another LENGTH byte (since BIGBIT is always 1)
; - 3 byte address (MSB first)
; - data bytes according LENGTH specification
;

;
;
```

END

### 6.3.4 专用变量初始化文件——INIT751.A51

INIT751.A51 的功能与 INIT.A51 的功能基本相同，不过是专门为 Signetics 8xC751 使用的。由于 8xC751 不支持访问 2kByte 以上的程序空间，也不可以访问外部的 RAM，所以程序中没有 XDATA 和 PDATA 部分。下面是 INIT751.A51 文件的源代码：

```

;-----
; This file is part of the C51 Compiler package
```

```
; Copyright(c) 1988-1999 KEILElektronik GmbH and KEILSoftware, Inc.
;
; INIT751.A51: This code is executed, if the application program contains
; initialized variables at file level.
;
; To translate this file use A51 with the following invocation:
;
; A51 INIT751.A51
;
; To link the modified INIT.OBJ file to your application use the following
; BL51 invocation:
;
; BL51 <your object file list>, INIT751.OBJ <controls>
;
;
; User-defined Watch-Dog Refresh.
;
; If the C application contains many initialized variables uses a watchdog
; it might be possible that the user has to include a watchdog refresh into
; the initialization process. The watchdog refresh routine can be included
; in the following MACRO and can alter all CPU registers except
; DPTR.
;
WATCHDOG MACRO
 ; Include any Watchdog refresh code here
 ENDM
;
;
NAME ?C_INIT

?
?C_C51STARTUP SEGMENT CODE
?C_INITSEG SEGMENT CODE ; Segment with Initializing Data

 EXTRN CODE(MAIN)
 PUBLIC ?C_START

 RSEG ?C_C51STARTUP
```

```

INITEND: AJMP MAIN

IorPData: ; If CY=1 PData Values
 CLR A
 MOVC A, @A+DPTR
 INC DPTR
 MOV R0, A ; Start Address

IorPLoop: CLR A
 MOVC A, @A+DPTR
 INC DPTR
 MOV @R0, A

Common: INC R0
 DJNZ R7, IorPLoop
 SJMP Loop

Bits: CLR A
 MOVC A, @A+DPTR
 INC DPTR
 MOV R0, A
 ANL A, #007H
 ADD A, #Table_LoadTab
 XCH A, R0
 CLR C
 RLC A ; Bit Condition to Carry
 SWAP A
 ANL A, #00FH
 ORL A, #20H ; Bit Address
 XCH A, R0 ; convert to Byte Addressen
 MOVC A, @A+PC

LoadTab: JC Setzen
 CPL A
 ANL A, @R0
 SJMP BitReady

Setzen: ORL A, @R0
BitReady: MOV @R0, A
 DJNZ R7, Bits
 SJMP Loop

Table: DB 00000001B
 DB 00000010B

```

```

 DB 00000100B
 DB 00001000B
 DB 00010000B
 DB 00100000B
 DB 01000000B
 DB 10000000B

?C_START:
 MOV DPTR, #?C_INITSEG
LOOP: CLR A
 MOV R6, #1
 MOVC A, @A+DPTR
 JZ INITEND
 INC DPTR
 MOV R7, A
 ANL A, #3FH
 JNB ACC. 5, NOBIG
 ANL A, #01FH
 MOV R6, A
 CLR A
 MOVC A, @A+DPTR
 INC DPTR
 JZ NOBIG
 INC R6
NOBIG: XCH A, R7
 ANL A, #0COH ; Typ is in Bit 6 and Bit 7
 ADD A, ACC
 JZ IorPDATA
 JC Bits
 SJMP $

RSEG ?C_INITSEG
DB 0
END

```

在上面的程序中，逻辑和 INIT.A51 基本相同，不过由于 8xC751 访问区间的限制，程序与 INIT.A51 相比简单了很多。

# 第 7 章 C51 的典型资源编程

基于 C51 语言基础知识后，本章开始介绍使用 C51 进行单片机应用程序设计的方法。本章就以 Keil 公司的 C 语言编译器为基础介绍 C51 单片机设计语言的相关知识，在本章的最后一节，还给出了部分 C51 程序示例。通过本章的学习，读者应该初步掌握使用 C51 语言进行 8051 单片机应用程序设计的方法，为后面具体实例的学习打下坚实的基础。

## 7.1 中断系统设计

8051 的中断系统包括 5 个中断请求源，并提供两个中断优先级，允许用户对中断源进行独立控制和中断优先级设置。8051 支持的 5 个中断源分别为外部中断 0、定时器 0 溢出中断、外部中断 1、定时器 1 溢出中断和串口中断。本节中将主要介绍外部中断 0 和中断嵌套的程序设计方法，关于定时器溢出中断的实现，将在 7.2 节结合单片机定时器的使用一并介绍，而串口中断的处理将在第 8 章中进行介绍。

C51 中，中断服务程序是以中断函数的方式实现的。C51 的中断函数格式如下：

```
Void 函数名() interrupt 中断号 using 工作组
{
 中断服务程序内容
}
```

### 1. 外部中断响应

#### [例 7-1] 使用外部中断 0

使用外部中断 0 提供读信号的方法，有关定时器读取的内容将被放到外部中断 0 的中断服务程序中。这里，设置  $\overline{\text{INT0}}$  为下降沿时，读取定时器内容。程序源码如下：

```
/* 外部中断触发读取运行中定时器的值 */
#ifndef __DEMO_4_12_C__
#define __DEMO_4_12_C__

#include <AT89X51.H>

#define _MHZ_ 12 // 设置单片机使用的晶振频率

void main()
```

```

{
 /* 定时、中断初始化 */
 TMOD = 0x10; // T1 使用定时模式，工作模式 1，无门控位
 TH1 = 0x3C; // 为 T1 填入初值，定时时间 50ms
 TL1 = 0xB0;
 TR1 = 1; // 启动 T1
 IT0 = 1; // 设置 INTRO 中断方式为边沿触发方式，负跳变时产生中断
 ET1 = 1; // 允许定时器 1 中断
 EX1 = 1; // 允许外部中断 0 中断
 EA = 1; // CPU 开放中断

 while(1); // 循环等待
}

/* T1 溢出中断处理函数 */
void timer1_int() interrupt 3 using 2 // T1 溢出中断，使用工作组 2
{
 TH1 = 0x3C; // 重新填入初值
 TL1 = 0xB0;
}

/* 外部中断 0 处理函数 */
void intr0_int() interrupt 0 using 2 // INTRO 中断，使用工作组 2
{
 unsigned char tmp1, tmp2;

 do // 读取 T1 中的内容
 {
 tmp1 = TH1;
 tmp2 = TL1;
 }while(tmp1 != TH1); // 数据无效则反复循环

 P0 = tmp2; // 发送有效数据
 P2 = tmp1;
}

#endif

```

## 2. 中断嵌套

当 CPU 处理一个中断源的请求，发生了更加重要的任务需要处理时，CPU 暂停当前中断

任务的处理转而去处理当前更加重要的任务，这就是 CPU 的中断嵌套。中断任务处理的先后顺序由其优先级决定，被中止的任务的优先级较低，被响应任务的优先级较高。

### [例 7-2] 两级中断嵌套

对定时器 1 溢出中断和外部中断 0 设置了不同的中断级别，其中定时器 1 溢出中断处于高级别，其服务程序负责程序的计时和 LED 显示，外部中断 0 服务程序用于简单的控制响应。当产生外部中断 0 时，读取 P1 口的内容，延时 1 S 后，将其取反后再重新发送给 P1 口，延时过程随时可以被计时过程打断，读写 P1 口语句收到保护，将不会受定时中断影响。

这个例子中涉及到了定时器中断的使用，读者在阅读这部分内容时，可以暂时将有关定时器的使用方法部分略过，重点学习初始化部分中断优先级的设置方法，其他部分可在学习了 7.2 节的内容后再重新阅读。

程序源代码如下，其中 LED\_show() 函数的程序这里没有给出，其功能为将计时得到的秒值送入该单片机系统的 LED 模块部分显示：

```
/* 两级中断嵌套 */

#ifndef __DEMO_4_13_C__
#define __DEMO_4_13_C__
#include <AT89X51.H>

#define _MHZ_ 12 // 设置单片机使用的晶振频率

void delay10ms(unsigned int count); // 声明延时函数

/* 将得到的秒数转化为时、分、秒的时间格式送显，由于其源码过长，本例未给出 */
void LED_show(unsigned long second);

unsigned char g_count; // 全局变量，timer_int 函数使用
unsigned char g_second; // 全局变量，timer_int 函数使用

void main()
{
 g_count = 0; // 设置全局变量初值
 g_second = 0;

 /* 定时、中断初始化 */
 TMOD = 0x10; // T1 使用定时模式，工作模式 1，无门控位
 TH1 = 0xD8; // 为 T1 填入初值，定时时间 10ms
 TL1 = 0xF0;
 TR1 = 1; // 启动 T1
 IT0 = 1; // 设置 INT0 中断方式为边沿触发方式，负跳变时产生中断
```

```

PT1 = 1; // 设置定时器 1 中断为高优先级
ET1 = 1; // 允许定时器 1 中断
EX1 = 1; // 允许外部中断 0 中断
EA = 1; // CPU 开放中断

while(1); // 循环等待
}

/* T1 溢出中断处理函数 */
void timer1_int() interrupt 3 using 3 // T1 溢出中断，使用工作组 3
{
 // 不同级别的中断函数使用不同的工作组

 TH1 = 0xD8; // 重新填入初值
 TL1 = 0xF0;

 g_count++;
 if(g_count == 100) // 计时满 1s
 {
 g_count = 0; // 清 0
 g_second++; // 总秒数增 1，溢出时会自动归 0
 LED_show(g_second); // LED 送显
 }
}

/* 外部中断 0 处理函数 */
void intr0_int() interrupt 0 using 2 // INTRO 中断，使用工作组 2
{
 unsigned char tmp;

 EA = 0; // 关中断
 P1 = 0xff; // 在读取 P1 口数据前，应先对其寄存器置 1
 tmp = P1; // 读 P1 口
 EA = 1; // 开中断

 delay10ms(100);

 EA = 0; // 关中断
 tmp = ~tmp; // 变量取反
 P1 = tmp; // 送 P1 口
 EA = 1; // 开中断
}

```

```

/* 延时 10ms，精度较低，参数 count 为延时时间 */
void delay10ms(unsigned int count)
{
 unsigned int i, k;
 unsigned char j;
 unsigned int tmp;
 tmp = (int)((100*_MHZ_)/12);

 for(i=0; i<count; i++)
 for(j=0; j<100; j++)
 for(k=0; k<tmp; k++);
}

#endif

```

## 7.2 定时/计数器的使用

8051 系列单片机内部设置了两个 16 位可编程的定时/计数器 T0 和 T1，具有计数器和定时器两种功能，提供 4 种工作模式。本节将介绍程序中使用 8051 定时系统的方法，其中包括定时器的使用以及定时器中断的设置方法等内容。

在程序设计中使用 8051 的定时系统应按照以下步骤进行。

- 计算定时常数

由于 8051 定时/计数器是随机器周期或外部计数递增，并在定时/计数器溢出时产生中断的，因此给定时器赋适当的初值（定时常数），可以控制定时器的时间。设需要的初值为 X，定时器位数为 n，计算定时常数的公式如下：

$$X = 2^n - \frac{\text{定时时间}}{\text{机器周期}}$$

例如，如果系统中使用的单片机的晶振频率为 12MHz，可以计算得到：

$$\text{机器周期} = \frac{12}{\text{晶振频率}} = \frac{12}{12 \times 10^6} = 1\mu\text{s}$$

使用定时器模式 1，16 位定时器，定时时间设为 10ms，计算定时常数如下：

$$X = 2^n \frac{\text{定时时间}}{\text{机器周期}} = 12^{16} \frac{1 \times 10^2}{1 \times 10^{-6}} = 55536$$

将其转换为十六进制，X=D8F0H，因此初值应设为 TH0=D8H，TL0=F0H。

- 初始化

主要是对定时器相关的 TCON、TMOD 以及和中断相关的 IP、IE 进行初始化，选定定时器的工作方式，并将初值写入定时器中。

- 程序设计

如果使用中断方式，就要进行中断服务程序设计和主程序设计。中断服务程序除完成相应工作外，如果定时器不是工作在模式 2 下，还要负责重新装载定时初值。

按照上述控制 8051 定时器的方法，下面举几个实例来帮助读者熟悉。

### 1. 产生频率为 200Hz 的方波

#### [例 7-3] 使用定时器

使用 8051 单片机的定时控制在 P1.0 脚上输出频率为 200Hz 的方波。要产生 200Hz 的方波，定时时间应为 5ms，如果晶振频率为 12MHz，使用工作模式 0，可以求得定时时间 X 如下：

$$X = 2^n - \frac{\text{定时时间}}{\text{机器周期}} = 2^{13} - \frac{5 \times 10^{-3}}{1 \times 10^{-6}} = 3192$$

将其转化为十六进制，X=0C78H，对 13 位定时器，8051 使用 TH0 的高 5 位和整个 TL0，因此，应将初值设为 TH0=60H，TL0=78H。

使用定时器 0，中断方式实现，程序代码如下：

```
/* 使用定时器在 P1.0 口产生 200Hz 方波 */

#ifndef __DEMO_4_14_C__
#define __DEMO_4_14_C__
#include <AT89X51.H>

void main()
{
 P1 = 0; // 清 P0 口

 /* 定时、中断初始化 */
 TMOD = 0x0; // T0 使用定时模式，工作模式 0，无门控位
 TH0 = 0x60; // 为 T0 填入初值，定时时间 5ms
 TL0 = 0x78;
 TR0 = 1; // 启动 T0
 ET0 = 1; // 允许定时器 0 中断
 EA = 1; // CPU 开放中断

 while(1); // 循环等待
}

/* T0 溢出中断处理函数 */
void timer0_int () interrupt 1 using 2 // T0 溢出中断，使用工作组 2
{
```

```

TH0 = 0x60; // 重新填入初值
TL0 = 0x78;
P1 ^= 0x01; // P1.0 取反，产生方波
}

#endif

```

## 2. 运行中读取定时器初值

读取运行中的定时器内容时要注意可能出现的错读问题。由于不可能在同一时刻同时读取 THX 和 TLX 的内容，如果先读入 TLX，此时恰好 TLX 溢出向 THX 进位，然后再读入 THX，则读取到的数值就会产生很大偏差。

一种可能的解决方法步骤如下：

- 读入 THX；
- 读入 TLX；
- 再读 THX；
- 比较两次读取的 THX 值，如果相同则证明结果有效，返回读取的定时器值。如果不同，则重复上述过程，直至得到有效的结果。

### [例 7-4] 读取定时器的值

将 T1 设为工作模式 1，定时时间定为 50ms，程序检测 P1.0 脚的电平，如果为高电平，则读取定时器 T1 的值，将高 8 位送入 P0 口，低 8 位送入 P2 口，然后延时 10ms。单片机使用晶振频率为 12MHz。

代码如下：

```

/* 运行中读取定时器的值 */

#ifndef __DEMO_4_15_C__
#define __DEMO_4_15_C__

#include <AT89X51.H>

#define _MHZ_ 12 // 设置单片机使用的晶振频率

void delay10ms(unsigned int count); // 声明延时函数

void main()
{
 bit btmp;
 unsigned char tmp1, tmp2;

 /* 定时、中断初始化 */

```

```

 TMOD = 0x10; // T1 使用定时模式, 工作模式 1, 无门控位
 TH1 = 0x3C; // 为 T1 填入初值, 定时时间 50ms
 TL1 = 0xB0;
 TR1 = 1; // 启动 T1
 ET1 = 1; // 允许定时器 1 中断
 EA = 1; // CPU 开放中断

 while(1) // 循环检测
 {
 P1_0 = 1; // 读取 P1.0 前, 应将其寄存器置 1
 btmp = P1_0;
 if(btmp) // 检测 P1.0, 如果为高电平, 则读取定时器值
 {
 do // 读取 T1 中的内容
 {
 tmp1 = TH1;
 tmp2 = TL1;
 }while(tmp1 != TH1); // 数据无效则反复循环

 P0 = tmp2; // 发送有效数据
 P2 = tmp1;
 }
 delay10ms(1); // 延时 10ms
 }

}

/* 延时 10ms, 精度较低, 参数 count 为延时时间 */
void delay10ms(unsigned int count)
{
 unsigned int i, k;
 unsigned char j;
 unsigned int tmp;
 tmp = (int)((100*_MHZ_)/12);

 for(i=0; i<count; i++)
 for(j=0; j<100; j++)
 for(k=0; k<tmp; k++);
}

```

```
/* T1 溢出中断处理函数 */
void timer1_int () interrupt 3 using 2 // T1 溢出中断, 使用工作组 2
{
 TH1 = 0x3C; // 重新填入初值
 TL1 = 0xB0;
}

#endif
```

### 3. 模式 3 的应用

8051 的 4 种定时/计数器模式中, 模式 3 是相对较为复杂的, 该模式仅对定时/计数器 0 有效, 此时定时/计数器 0 被拆成两个独立的 8 位计数器 TL0 和 TH0。TL0 使用 T0 的全部硬件资源, TH0 占用定时/计数器 1 的 TR1 和 TF1 两位。这种模式下, T1 仍然可以工作, 但无法使用中断。

#### [例 7-5] T0 使用工作模式

使定时/计数器 T0 工作在模式 3 下, TL0 和 TH0 作为两个独立的 8 位定时器, 分别在 P1.0 和 P1.1 口产生频率为 5kHz 和 10kHz 的方波。单片机使用的晶振频率为 12MHz。

代码如下:

```
/* T0 使用工作模式 3 */

#ifndef __DEMO_4_16_C__
#define __DEMO_4_16_C__

#include <AT89X51.H>

void main()
{
 /* 定时、中断初始化 */
 TMOD = 0x03; // T0 使用定时模式, 工作模式 3, 无门控位
 TL0 = 0x38; // 为 TL0 填入初值, 定时时长 0.2ms
 TH0 = 0x9C; // 为 TH0 填入初值, 定时时长 0.1ms
 TR0 = 1; // 启动 TL0
 TR1 = 1; // 启动 TH0
 ET0 = 1; // 允许定时器 0 中断, 此时该中断被 TL0 占用
 ET1 = 1; // 允许定时器 1 中断, 此时该中断被 TH0 占用
 EA = 1; // CPU 开放中断

 while(1); // 循环等待
}
```

```

/* T0 溢出中断处理函数 */
void timer0_int() interrupt 1 using 2 // T0 溢出中断, 使用工作组 2
{
 // 此时该中断被 TL0 使用
 TL0 = 0x38; // 重新填入初值

 P1_0 = !P1_0; // 在 P1.0 口产生方波
}

/* T1 溢出中断处理函数 */
void timer1_int() interrupt 3 using 2 // T1 溢出中断, 使用工作组 2
{
 // 此时该中断被 TH0 使用
 TH0 = 0x9C; // 重新填入初值

 P1_1 = !P1_1; // 在 P1.1 口产生方波
}

#endif

```

### 7.3 I/O 口的使用

输入/输出 (I/O) 接口是单片机和外部设备之间信息交换和控制的桥梁。它主要有以下几种作用：

#### 1. 实现和不同外部设备的速度匹配

不同的外设工作速度差别很大,而且一般来讲外设的响应速度远远小于 CPU 的运算速度。所以接口电路就必须适应 CPU 和外设的速度上的这个矛盾。一般来讲, CPU 和外设间的数据传送方式有同步、异步、中断、DMA 等。但本书的侧重点在 C51 程序的编写上,故对这些内容不深入讨论。有需要的读者可以参考其他的书籍。

下面以同步传输为例,说明一个慢的外设和单片机之间是如何配合工作的。

#### [例 7-6] 数据采集

一个数据采集系统,收集数据的速度比较慢,当它每采集完数据时就会向外送出 8bit 的编码,同时使 READY 信号高,并等待 CPU 将数据取走。CPU 读出数据以后需要向 CLR 信号线发送一个脉冲,数据采集器就知道数据已被采走,并自动将信号变为低电平,同时开始下一轮的数据采集。其中,数据采集器的 READY 信号与单片机的 P0.0 相连,CLR 信号与单片机的 P0.1 相连。

分析:这其实是单片机的一个典型简单应用,即逻辑操作的控制、转换问题。程序主体由一个大循环组成,不断检查 READY 信号是否有效,如果有效,就读取数据,并给出脉冲通

知外设。

C程序代码如下：

```
#include "reg51.h"

sbit CLR=P0^0;
sbit READY=P0^1;

void delay();

main()
{
 unsigned char indata;
 CLR=0;
 while(1)
 {
 while(READY)
 {
 indata=P1;
 CLR=1;
 delay();
 CLR=0;
 }
 }
}

void delay()
{
 int i=0;
 while(i<100) i++;
}
```

该程序即为 <http://www.cs-book.com> 页面上相应的目录 ex08 中的程序。

上面的收集数据的方案并不是实际应用中所采用的一般形式，这种方案存在相当的缺陷，但是通过它可以简单地说明速度匹配的问题。

当然，采用采用中断和异步方式也可以和外设之间实现数据传输，这两种方式将留在后面的中断和串口通信部分再加以说明。而如果要使用 DMA（直接内存读取，Direct Memory Access）方式，则要另外添加专用的 DMA 芯片，本书不再涉及，有兴趣的读者可以参考相关的资料。

## 2. 改变数据传送的方式

一般来说，I/O 数据有并行和串行两种方式。并行数据的优点在于带宽大，传送的数据

量也大，但是连线复杂，一般只存在于印刷电路板上。串行数据的优点在于连线简单，但是由于受带宽的限制，传送的数据量小。尽管如此，串行数据传送仍然在远程设备的数据传输，如多机通信和远程控制中占有相当重要的地位。由于数据在印刷电路板上是并行传输的，而通信是又变成串行，故需要 I/O 接口来改变传输方式及电平特性。

### 3. 改变信号的性质和电平

CPU 和外设之间的信号有几类：有数据型的，如程序代码和计算结果等；有状态型的，用来反映外部设备的工作状态（如外部设备的已启动、未启动、忙、空闲等等）；有控制型的，用来控制外部设备的工作状态（如外部设备的开、关，电机的转动、停止、速度的控制等），因此，I/O 接口必须既能把外设送来的状态信息归整划一后送给 CPU，又能自动根据要求给外部设备发送合适的控制电平和控制命令。

下面以一个闪灯程序为例说明使用 P0 口作为 I/O，直接进行开关控制的例子。

#### [例 7-7] 彩灯系统实例

某系统如下：系统中总共有 8 个彩灯，需要他们轮流闪动。先是只有 1 号灯亮，其他灯灭，过一会儿以后只有二号灯亮，再过一会儿只有三号灯亮……，在 8 号灯亮过以后，然后再只让 1 号灯亮，如此循环。

分析：这个程序逻辑上其实很简单，就是按顺序控制 I/O 口 P0 的每一个管脚的电平。在 C51 中，I/O 访问与变量访问在方式上是一样的。

C51 程序代码如下：

```
#include "reg51.h"
```

```
#define ON 1
#define OFF 0
```

```
sbit Lamp1=P0^0;
sbit Lamp2=P0^1;
sbit Lamp3=P0^0;
sbit Lamp4=P0^1;
sbit Lamp5=P0^0;
sbit Lamp6=P0^1;
sbit Lamp7=P0^0;
sbit Lamp8=P0^1;
```

```
void initial();
void delay();
```

```
main()
{
 initial();
```

```
delay();
while(1)
{
 Lamp1=ON;
 delay();
 Lamp1=OFF;
 delay();

 Lamp2=ON;
 delay();
 Lamp2=OFF;
 delay();

 Lamp3=ON;
 delay();
 Lamp3=OFF;
 delay();

 Lamp4=ON;
 delay();
 Lamp4=OFF;
 delay();
 Lamp5=ON;
 delay();
 Lamp5=OFF;
 delay();

 Lamp6=ON;
 delay();
 Lamp6=OFF;
 delay();

 Lamp7=ON;
 delay();
 Lamp7=OFF;
 delay();

 Lamp8=ON;
 delay();
 Lamp8=OFF;
```

```

 delay();

}

void initial()
{
 Lamp1=OFF;
 Lamp2=OFF;
 Lamp3=OFF;
 Lamp4=OFF;
 Lamp5=OFF;
 Lamp6=OFF;
 Lamp7=OFF;
 Lamp8=OFF;
}

void delay()
{
 int i=0;
 while(i<10000) i++;
}

```

即为 <http://www.cs-book.com> 页面上本章相应的 ex09 目录下的程序。

## 7.4 扩展存储器

### 7.4.1 外部 ROM

一个对外部 ROM 访问的程序见下面的两个代码清单。对 ROM 的操作不外乎读出和写入两个环节，下面的代码清单是向外部 ROM 写入两个字节数据。

```

void senddate(unsigned char* a)
{
 XBYTE[DTOAHIGH]=*a;
 delay(DELAY_VALUE);
 XBYTE[DTOALOW]=*(a+1);
 delay(DELAY_VALUE);
} //向外部 ROM 写入两个字节数据，数据内容存在于变量 a 中

```

下面的代码清单功能是从外部 ROM 中读入两个字节数据。XBYTE 的相关用法见本书前面

相关章节的内容。

```
unsigned char *getdate(void);
{
 unsigned char* a;
 PinADCS=TRUE;
 PinADStatus=TRUE;
 while(PinADStatus==TRUE);
 *a=XBYTE[ATODHIGH];
 *(a+1)=XBYTE[ATODLOW];
 return a;
} //从外部 ROM 读入两个字节，存入到指针变量 a 中
```

## 7.4.2 外部 RAM

### [例 7-8] 串行发送

片外 RAM 1000H~104FH 中存放的是 ASCII 码，设计一个发送程序，将其中的所有内容串行发送。工作在模式 1。串口速率 9.6kbit/s，单片机主时钟频率 24MHz。

分析：根据我们上面的介绍，使用 51 单片机的串口发送一个字符并不困难，只要设置好数据传输率等参数，然后将要发送的值写入到规定的缓存中去就可以了。不过，由于本例是要传送多个数据，所以在自己编制的发送程序中，还存在怎样得知何时发送下一个数据的问题。这个问题可以用两种方式解决：一种是中断的方式，即在缓冲中的字符发送结束以后，串口中断激活相应的中断处理程序，然后在中断处理程序中写入下一个要发送的字符；另外的一种方式是查询的方式，即在主程序中不断查询相关标志，如果表示串口发送缓冲空的标志位有效了，那么我们就再填入新的字符。另外，对于 ASCII 码字符，KEIL C51 还提供了 printf 库函数可以使用。下面我们就分别给出这 3 种操作方式的程序。

- 中断方式

分析：不管对何种方式来说，最先要做的是串口的初始化工作。我们将串口的初始化工作全部写在一个函数 initial() 之中，该函数的主要作用是初始化中断优先级、允许串口中断、定义串口工作模式，以及数据传输率设置，同时启动与数据传输率有关的定时器（一般是定时器 2，这也是本例使用的定时器）。initial 子函数的程序代码如下所示：

```
void initial(void)
{
 IP=0x10; // 定义串口为高优先级中断
 IE=0x90; // 允许串口、中断 0、1、定时器 0
 TCON=0x05;
 TMOD=0x20; // 定时器 1 为自动装入(auto-load)方式
```

```

 PCON=0; //SMOD(PCON.7)=1 时, baudrate doubled. //smode=1 (11M 晶振
时
 //为 0)

 SCON=0xD0; //串行口工作方式 : 9 位 UART, 数据传输率可变
 TH1=0xf3;
 TL1=0xf3; //
 PCON=0x80|PCON; //数据传输率设置 : 9600 baud(E8—24MHz)
 TR1=1; //启动定时器 1
}

```

另外，我们需要在中断中对串口发送缓冲进行填写，并将发送位设置为有效。由于串口的接收和发送共用一个中断号，所以在中断的程序中，我们必须先判断是否是发送中断有效，即 TI 是否为 1，只有该比特为 1 的时候，我们才能继续发送。

发送的逻辑如下：如果发送字节数不到 0x4F，那么将新的内容填写到发送缓存 SBUF 中，并将 TI 复位，如果已经超过 0x4F，那么就不再向 SBUF 中填入新的内容。

根据上面的逻辑，发送中断的函数应该如下：

```

void Rcv_INT(void) interrupt 4
{
 if(TI)
 {
 if(i<0x4F)
 {
 i++;
 ACC=p[i];
 SBUF=ACC;
 }
 TI=0;
 }
}

```

上面的代码中，中断处理程序的函数名是任意的，其入口地址由 interrupt 关键字后面的中断号来确定。必须注意的是，中断处理函数不能有入口参数或返回值，否则有可能出错。

剩下的就是主函数的编写了。主函数的逻辑比较简单，只需要调用初始化函数对串口进行初始化。另外，主函数还必须控制要发送的第一个字节，这样才能在发送结束后保证发送中断函数的调用，从而导致发送逻辑的自动更新。

主函数的代码如下所示：

```

main()
{
 initial();
 p=TxDATA;
 ACC=p[0];
}

```

```
 SBUF=ACC;
```

```
 while(1);
}
```

为了读者的阅读方便，我们将完整的代码附在下面：

```
#include <stdio.h>
#include <reg51.h>

#define TxDATA (unsigned xdata char*)1000H
void initial(void);
unsigned xdata char *p;
int i=0;
```

```
main()
```

```
{
 initial();
 p=TxDATA;
 ACC=p[0];
 SBUF=ACC;
 while(1);
}
```

```
void Rcv_INT(void) interrupt 4
```

```
{
 if(TI)
 {
 if(i<0x4F)
 {
 i++;
 ACC=p[i];
 SBUF=ACC;
 }
 TI=0;
 }
}
```

```
void initial(void)
```

```
{
 IP=0x10; //定义串口为高优先级中断
```

```

IE=0x90; //允许串口、中断 0、1、定时器 0
TCON=0x05;
TMOD=0x20; //定时器 1 为自动装入(auto-load)方式

PICON=0; //SMOD(PICON.7)=1 时, baudrate doubled. //smode=1 (11M 晶振
时为 0)

SCON=0xD0; //串行口工作方式 : 9 位 UART, 数据传输率可变
TH1=0xf3;
TL1=0xf3; //
PICON=0x80|PICON; //数据传输率设置 : 9600 baud(E8--24MHz)
TR1=1; //启动定时器 1
}

```

该程序在 <http://www.cs-book.com> 页面上相应的 ex11a 目录下。

- 查询方式

分析：读者阅读过中断的发送方式以后，相信对程序的大体流程已经有所了解。在查询的方式中，我们的初始化逻辑基本一样，只不过不打开任何中断（由于我们没有任何中断处理程序，所以在这个时候如果打开中断是危险的，这种做法可能会导致系统的不稳定状态）。初始化代码如下：

```

void initial(void)
{
 IE=0x00; //中断禁止
 TCON=0x05;
 TMOD=0x20; //定时器 1 为自动装入(auto-load)方式

 PICON=0; //SMOD(PICON.7)=1 时, baudrate doubled. //smode=1 (11M 晶振时为
0)

 SCON=0xD0; //串行口工作方式 : 9 位 UART, 数据传输率可变
 TH1=0xf3;
 TL1=0xf3; //
 PICON=0x80|PICON; //数据传输率设置 : 9600 baud(E8--24MHz)
 TR1=1; //启动定时器 1
}

```

由于在这个程序中并不需要中断，所以也就没有了中断处理程序。不过因此会导致主程序相对复杂一些。在这个主程序中，我们的程序始终运行在一个循环中，并不断查询 TI 的状态，如果 TI 被置位并且发送的字节数没有超过 0x4F，那么就将新的内容写入缓存，否则就什么也不做。

主程序如下：

```
main()
{
 initial();
 p=TxDATA;
 ACC=p[0];
 SBUF=ACC;

 while(1)
 {
 if(TI==1&&i<0x4F)
 {
 i++;
 ACC=p[i];
 SBUF=ACC;
 }
 }
}
```

以下是完整的代码：

```
#include <stdio.h>
#include <reg51.h>

#define TxDATA (unsigned xdata char*)1000H
void initial(void);
unsigned xdata char *p;

int i=0;
main()
{
 initial();
 p=TxDATA;
 ACC=p[0];
 SBUF=ACC;

 while(1)
 {
 if(TI==1&&i<0x4F)
 {
 i++;
 }
 }
}
```

```

 ACC=p[i];
 SBUF=ACC;
 }

}

void initial(void)
{
 IE=0x00; //中断禁止
 TCON=0x05;
 TMOD=0x20; //定时器 1 为自动装入(auto-load)方式
 PCON=0; //SMOD(PCON.7)=1 时, baudrate doubled. //smode=1 (11M 晶振时为
0)
 SCON=0xD0; //串行口工作方式 : 9 位 UART, 数据传输率可变
 TH1=0xf3;
 TL1=0xf3; //
 PCON=0x80|PCON; //数据传输率设置 : 9600 baud(E8--24MHz)
 TR1=1; //启动定时器 1
}

```

该程序在 <http://www.cs-book.com> 页面上本书中的 ex11b 目录下。

需要提醒读者注意的是, 由于本例比较简单, 所以查询和中断方式在程序运行的时候看不出明显的区别。不过如果主函数的逻辑比较复杂的话, 要及时的发送每一个字节, 并且不希望系统将过多的时间资源浪费在查询的工作上的话, 那么最好使用中断的方式, 这样会在一定程度上提高程序的运行性能。

- 使用 KEIL 的库函数

**分析:** 在前面的章节中就已经说过, KEIL 自带了输入输出的库函数, 而 printf 函数是输出到串口的, 所以为了方便, 我们可以在串口初始化结束以后, 直接调用 printf 函数进行一个字符的发送, 然后重复调用 printf, 直到发送完规定的字节数。

程序的源代码如下:

```

#include <stdio.h>
#include <reg51.h>

#define TxDATA (unsigned xdata char*)1000H
void initial(void);
unsigned xdata char *p;

int i=0;
main()

```

```

{
 initial();
 p=TxDATA;
 ACC=p[0];
 SBUF=ACC;

 while(1)
 {
 if(i<0x4F)printf("%c", p[i++]);
 }
}

void initial(void)
{
 IE=0x00; //中断禁止
 TCON=0x05;
 TMOD=0x20; //定时器 1 为自动装入(auto-load)方式

 PCON=0; //SMOD(PCON.7)=1 时, baudrate doubled. //smode=1 (11M 晶振时为 0)

 SCON=0xD0; //串行口工作方式 : 9 位 UART, 数据传输率可变
 TH1=0xf3;
 TL1=0xf3; //
 PCON=0x80|PCON; //数据传输率设置 : 9600 baud(E8--24MHZ)
 TR1=1; //启动定时器 1
}

```

该程序在 <http://www.cs-book.com> 页面上相应的 ex11c 目录下。

从上面的程序我们可以看出, KEIL 自带的输入输出函数可以让我们忽略很多发送的细节, 并且一般来说, KEIL 的库函数都是经过长时间的测试与多人的使用的, 所以其稳定性和可靠性都十分好。建议读者在可能的情况下应尽量考虑使用库函数, 这样也可以大大降低自己的工作量。

### 7.4.3 外部串行 E<sup>2</sup>PROM

#### [例 7-9] 读写 AT93C56

AT93C56 是 Atmel 公司生产的低功耗、低电压、电可擦除、可编程只读存储器, 采用 CMOS 工艺技术, 容量为 4kB, 可重复写 100 万次, 数据可保存 40 年以上, 接口使用 National Semiconductor 公司的 Microwire 三线同步串行总线, 提供 8 脚 PDIP/SOIC 封装和 14 脚 SOI 封装, 其 DIP 封装的管脚图如图 7-1 所示。

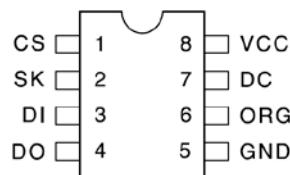


图 7-1 AT93C56 管脚图

AT93C56 各管脚功能如下：

- VCC、GND 电源、接地端，电压范围 2.7~5.5V。
- CS 片选信号，高电平有效。
- SK 串行时钟信号，SK 上升沿进行芯片读写。
- DI 串行数据输入端。
- DO 串行数据输出端，地址擦/写周期或芯片擦/写周期时，该端提供忙/闲信息。
- ORG 存储器构造配置端。该端接高电平时，输出为 16 位；接低电平时，输出为 8 位。
- DC 空脚，无连接。

AT93C56 共有 7 条控制命令，其格式如下：

| 起始位 | 操作码 | 地址段    | 数据段    |
|-----|-----|--------|--------|
| 1 位 | 2 位 | A7~A 0 | D15~D0 |

各个控制命令如表 7-1 所示。

表 7-1 AT93C56 控制命令

| 操作命令  | 起始位 | 操作码 | 地址段      |          | 数据段   |        | 功能       |
|-------|-----|-----|----------|----------|-------|--------|----------|
|       |     |     | 8 位      | 16 位     | 8 位   | 16 位   |          |
| READ  | 1   | 10  | A7~A 0   | A6~A 0   | 空     |        | 从指定单元读数据 |
| EWEN  | 1   | 00  | 11XXXXXX | 11XXXXXX | 空     |        | 允许写指令    |
| ERASE | 1   | 11  | A7~A 0   | A6~A 0   | 空     |        | 擦除指定单元   |
| WRITE | 1   | 01  | A7~A 0   | A6~A 0   | 空     |        | 写入指定单元   |
| ERAL  | 1   | 00  | 10XXXXXX | 10XXXXXX | 空     |        | 擦除所有存储单元 |
| WRAL  | 1   | 00  | 01XXXXXX | 01XXXXXX | D7~D0 | D15~D0 | 写入所有存储单元 |
| EWDS  | 1   | 00  | 00XXXXXX | 00XXXXXX | D7~D0 | D15~D0 | 禁止写指令    |

8051 单片机与 AT93C56 的硬件连接如图 7-2 所示。其 CS、SK、DI、DO 分别连接单片机的 P1.0、P1.1、P1.2 和 P1.3 口。

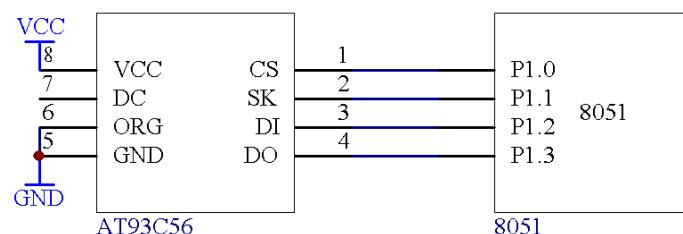


图 7-2 AT93C56 的硬件连接

以下代码为在 ORG 接地（输出为 8 位）情况下访问 AT93C56 的所有库函数。在阅读代码

时，要注意AT93C56读写时序的处理。

```
/* demo_4_11.c */
/**
 * 程序说明：AT93C56 操作函数库，ORG 端接地
 * 使用语言：C51
 * 编译工具：Keil uVision2.38a
 * 作者：djws
 * 版本：v1.1
 * 时间：2004.7
 *****/
#ifndef __DEMO_4_11_C__
#define __DEMO_4_11_C__

#include <REG51.H>

#define uchar unsigned char
#define uint unsigned int

sbit CS = P1^0; // 使用 sbit 命令将 P1_0 表示为 CS
sbit SK = P1^1; // 使用 sbit 命令将 P1_1 表示为 SK
sbit DI = P1^2; // 使用 sbit 命令将 P1_2 表示为 DI
sbit DO = P1^3; // 使用 sbit 命令将 P1_3 表示为 DO

void at93c56_ewen(void); /* 写使能 */
void at93c56_ewds(void); /* 写禁止 */
uchar at93c56_read(uchar addr); /* 读指定地址字节 */
void at93c56_write(uchar addr, uchar mybyte); /* 将指定字节写入指定存储单元 */
void at93c56_erase(uchar addr); /* 擦除指定单元 */
void at93c56_wral(uchar mybyte); /* 将指定字节写入所有存储单元 */
void at93c56_eral(void); /* 擦除所有存储单元 */

/* 写使能函数，使 EEPROM 处于可写状态 */
void at93c56_ewen(void)
{
 uchar i, tmp;

 CS = 0; SK = 0; CS = 1; // 时序同步
 DI = 1; SK = 1; SK = 0; // 送起始位 1
}
```

```

 tmp = 0x30; // 0B0011XXXX
 for(i=0; i<8; i++) // 送命令字
 {
 DI = tmp&0x80;
 SK = 1; SK = 0;
 tmp <= 1;
 }

 CS = 0;
}

/* 写禁止函数，禁止写入 EEPROM */
void at93c56_ewds(void)
{
 uchar i, tmp;

 CS = 0; SK = 0; CS = 1; // 时序同步
 DI = 1; SK = 1; SK = 0; // 送起始位 1

 tmp = 0x00; // 0B0000XXXX
 for(i=0; i<8; i++) // 送命令字
 {
 DI = tmp&0x80;
 SK = 1; SK = 0;
 tmp <= 1;
 }

 CS = 0;
}

/* 读指定地址字节，参数 addr 为内存地址 */
uchar at93c56_read(uchar addr)
{
 uchar i, result;

 CS = 0; SK = 0; CS = 1; // 时序同步
 DI = 1; SK = 1; SK = 0; // 送起始位 1
 DI = 1; SK = 1; SK = 0; // 送操作码 10
 DI = 0; SK = 1; SK = 0;
}

```

```
for(i=0; i<8; i++) // 送地址
{
 DI = addr&0x80;
 SK = 1; SK = 0;
 addr <= 1;
}

CS = 0; DO = 1; CS = 1; // 置接收端为 1
SK = 0;
while(!DO) // 检查 DO 是否为 0, DO 为 0 表示芯片开始传送数据
{
 SK = 1; SK = 0;
}

SK = 1; SK = 0; // 空过第 1 位标志位
result = 0;
for(i=0; i<8; i++) // 接收数据
{
 result <= 1;
 result = result|DO;
 SK = 1; SK = 0;
}

CS = 0;

return(result);
}

/* 将指定字节写入指定存储单元, 参数 addr 为内存地址, 参数 mybyte 为写入数据 */
void at93c56_write(uchar addr, uchar mybyte)
{
 uchar i;

 at93c56_ewen(); // 写使能

 CS = 0; SK = 0; CS = 1; // 时序同步
 DI = 1; SK = 1; SK = 0; // 送起始位 1
 DI = 0; SK = 1; SK = 0; // 送操作码 01
 DI = 1; SK = 1; SK = 0;
```

```

 for(i=0; i<8; i++) // 送地址
 {
 DI = addr&0x80;
 SK = 1; SK = 0;
 addr <= 1;
 }

 for(i=0; i<8; i++) // 送数据
 {
 DI = mybyte&0x80;
 SK = 1; SK = 0;
 mybyte <= 1;
 }

 CS = 0; DO = 1; CS = 1; // 置接收端为 1
 SK = 0;
 while(DO) // DO 为 0 表示芯片忙
 {
 SK = 1; SK = 0;
 }

 SK = 0; CS = 0;

 at93c56_ewen(); // 写禁止
}

/* 擦除指定单元，参数 addr 为内存地址 */
void at93c56_erase(uchar addr)
{
 uchar i;

 at93c56_ewen(); // 写使能

 CS = 0; SK = 0; CS = 1; // 时序同步
 DI = 1; SK = 1; SK = 0; // 送起始位 1
 DI = 1; SK = 1; SK = 0; // 送操作码 11
 DI = 1; SK = 1; SK = 0;

 for(i=0; i<8; i++) // 送地址
 {

```

```
DI = addr&0x80;
SK = 1; SK = 0;
addr <= 1;
}

CS = 0; DO = 1; CS = 1; // 置接收端为1
SK = 0;
while(DO) // DO为0表示芯片忙
{
 SK = 1; SK = 0;
}

SK = 0; CS = 0;

at93c56_ewen(); // 写禁止
}

/* 将指定字节写入所有存储单元，参数 mybyte 为写入数据 */
void at93c56_wral(uchar mybyte)
{
uchar i, tmp;

at93c56_ewen(); // 写使能

CS = 0; SK = 0; CS = 1; // 时序同步
DI = 1; SK = 1; SK = 0; // 送起始位1

tmp = 0x10; // 0B0001XXXX
for(i=0; i<8; i++) // 送命令字
{
 DI = tmp&0x80;
 SK = 1; SK = 0;
 tmp <= 1;
}

for(i=0; i<8; i++) // 送数据
{
 DI = mybyte&0x80;
 SK = 1; SK = 0;
 mybyte <= 1;
```

```

 }

 CS = 0; DO = 1; CS = 1; // 置接收端为 1
 SK = 0;
 while(DO) // DO 为 0 表示芯片忙
 {
 SK = 1; SK = 0;
 }

 SK = 0; CS = 0;

 at93c56_ewen(); // 写禁止
}

/* 擦除所有存储单元 */
void at93c56_eral(void)
{
 uchar i, tmp;

 at93c56_ewen(); // 写使能

 CS = 0; SK = 0; CS = 1; // 时序同步
 DI = 1; SK = 1; SK = 0; // 送起始位 1

 tmp = 0; // 0B0000XXXX
 for(i=0; i<8; i++) // 送命令字
 {
 DI = tmp&0x80;
 SK = 1; SK = 0;
 tmp <= 1;
 }

 CS = 0; DO = 1; CS = 1; // 置接收端为 1
 SK = 0;
 while(DO) // DO 为 0 表示芯片忙
 {
 SK = 1; SK = 0;
 }

 SK = 0; CS = 0;
}

```

```

at93c56_ewen() ; // 写禁止
}

#endif

```

## 7.5 一个使用多种资源的完整例程

为了详细说明各个资源的协同使用，并为读者提供一个完整的协同使用多资源的例子，下面我们来看一个简单的单片机在实际工程中的例子。

### 7.5.1 项目需求

工程要求如下：在一个平面坐标系下，一个物体在 2 个步进电机（分别控制 X 方向和 Y 方向）的控制下前进。X 方向上的电机控制该物体在 X 方向上的运动速度，Y 方向上的电机控制该物体在 Y 方向上的运动速度。在另一个系统中，需要该物体在平面坐标系中的速率作为参数，来进行其他操作。由于本书只是为了向读者介绍 C 语言的写法，所以在此这个系统简化另外一个步进电机转动速度与参考物体的速率成正比。由于步进电机转动的时候步进电机驱动器发出的脉冲信号是离散的，所以必须采用微积分的离散近似计算。本系统要求以 100ms 为单位，也就是说，取 T 为 100ms，每隔 100ms 表示速度值变更一次，并近似认为这 100ms 以内物体的运动是匀速的。

### 7.5.2 步进电机背景知识

为了读者理解起来方便，在这里首先向大家简单介绍一下有关的背景知识。

#### 1. 步进电机

步进电机是一种控制用的特种电机，它的旋转是以固定的角度（称为“步距角”）一步一步运行的，其特点是没有积累误差（精度为 100%），所以广泛应用于各种开环控制。步进电机的运行要有一电子装置进行驱动，这种装置就是步进电机驱动器，它是把控制系统发出的脉冲信号转化为步进电机的角位移，或者说，控制系统每发一个脉冲信号，驱动器就使步进电机旋转一步距角，所以步进电机的转速与脉冲信号的频率成正比。

因此，控制步进脉冲信号的频率，可以对电机精确调速；控制步进脉冲的个数，可以达到对电机精确定位的目的。

在实际的工作中，为了满足不同要求的系统，步进电机驱动器一般都支持所谓的细分功能。要了解“细分”，先要弄清“步距角”这个概念：它表示控制系统每发一个步进脉冲信号，电机所转动的角度。电机出厂时给出了一个步距角的值，如 86BYG250A 型电机给出的值为  $0.9^\circ / 1.8^\circ$ （表示半步工作时为  $0.9^\circ$ 、整步工作时为  $1.8^\circ$ ），这个步距角可以称之为电机固有步距角，它不一定是电机实际工作时的真正步距角，真正的步距角和驱动器有关，参见

表 7-2 (还以 86BYG250A 型电机为例):

表 7-2

86BYG250A 的细分

| 电机固有步距角    | 所用驱动器类型及工作状态     | 电机运行时的真正步距角 |
|------------|------------------|-------------|
| 0.9° /1.8° | 驱动器工作在半步状态       | 0.9°        |
| 0.9° /1.8° | 细分驱动器工作在 5 细分状态  | 0.36°       |
| 0.9° /1.8° | 细分驱动器工作在 10 细分状态 | 0.18°       |
| 0.9° /1.8° | 细分驱动器工作在 20 细分状态 | 0.09°       |
| 0.9° /1.8° | 细分驱动器工作在 40 细分状态 | 0.045°      |

从表 7-2 可以看出：步进电机通过细分驱动器的驱动，其步距角变小了，如驱动器工作在 10 细分状态时，其步距角只为‘电机固有步距角’的十分之一，也就是说：‘当驱动器工作在不细分的整步状态时，控制系统每发一个步进脉冲，电机转动 1.8°；而用细分驱动器工作在 10 细分状态时，电机只转动了 0.18°’，这就是细分的基本概念。

细分功能完全是由驱动器靠精确控制电机的相电流所产生的，与电机无关。

在实际的工作中，是否需要进行细分完全是由系统要求来决定的。驱动器细分后的优点为：

- 完全消除了电机的低频振荡。低频振荡是步进电机（尤其是反应式电机）的固有特性，消除它的途径就是细分模式，如果步进电机有时要在共振区工作（如走圆弧），选择细分驱动器是唯一的选择。
- 提高了电机的输出转矩。尤其是对三相反应式电机，其力矩比不细分时提高约 30~40%。
- 提高了电机的分辨率。由于减小了步距角、提高了步距的均匀度，“提高电机的分辨率”是不言而喻的。
- 以上这些优点，尤其是在性能上的优点，并不是一个量的变化，而是质的飞跃。根据各个公司多年来的纪录发现，原来使用不细分驱动器的用户通过比较后，大都改选为细分驱动器。所以一般来说，最好选用细分驱动器。不过由于在本例中我们仅仅是要从原理上来让读者明白使用 51 单片机搭建嵌入式系统的方法，所以在系统的实现中并没有使用细分。

## 2. 步进电机驱动器的使用方法

如图 7-3 所示，一般来说，驱动器的输入信号共有 3 路，它们是：步进脉冲信号 CP、方向电平信号 DIR、脱机信号 FREE。它们在驱动器内部分别通过限流电阻接入光耦的负输入端，且电路形式完全相同，在这三路输入信号共同控制下，驱动器将输出合适的电流来控制步进电机完成制定的操作。另外，驱动器一般有一个接入端 OPTO，该端口为三路信号的公共正端（三路光耦的正输入端），三路输入信号在驱动器内部接成共阳方式，所以 OPTO 端须接外部系统的 VCC，并在需要的情况下加限流电阻 R，保证给驱动器内部光耦提供合适的驱动电流。

### • 步进脉冲信号 CP

步进脉冲信号 CP 用于控制步进电机的位置和速度，也就是说：驱动器每接受一个 CP 脉冲就驱动步进电机旋转一个步距角（细分时为一个细分步距角），CP 脉冲的频率改变则会使步进电机的转速改变，控制 CP 脉冲的个数，则可以使步进电机精确定位。这样就可以很方便地达到步进电机调速和定位的目的。例如，对 SH-3F090M 型驱动器来说，CP 信号为低电平有效，要求 CP 信号的驱动电流为 8~15mA，对 CP 的脉冲宽度也有一定的要求，一般不小于 5 μs。

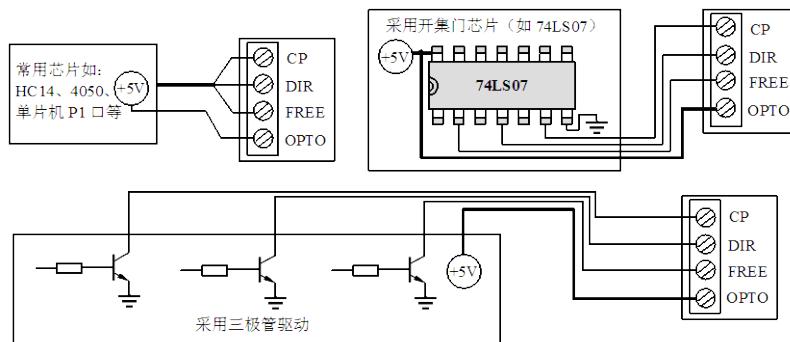


图 7-3 步进电机驱动器的接口端说明

- 方向电平信号 DIR

方向电平信号 DIR 用于控制步进电机的旋转方向。此端为高电平时，电机一个转向；此端为低电平时，电机为另一个转向。电机换向必须在电机停止后再进行，并且换向信号一定要在前一个方向的最后一个 CP 脉冲结束后以及下一个方向的第一个 CP 脉冲前发出。

- 脱机电平信号 FREE

当驱动器上电后，步进电机处于锁定状态（未施加 CP 脉冲时）或运行状态（施加 CP 脉冲时），但如果用户想手动调整电机而又不想关闭驱动器电源，这时就可以用到此信号。当此信号起作用时（低电平有效），电机处于自由无力矩状态；当此信号为高电平或悬空不接时，取消脱机状态。此信号用户可选用，如果不需要此功能，此端不接即可。

三个输入信号配合下让步进电机操作所对应的时序图如图 7-4 所示。

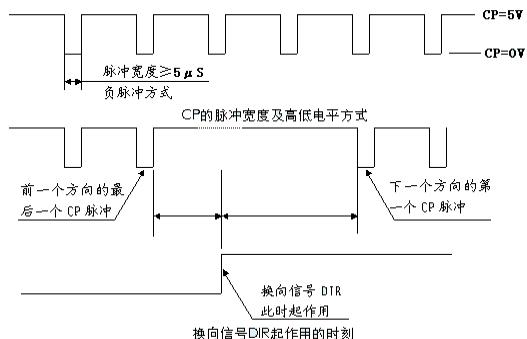


图 7-4 输入信号的时序图

### 3. 升降速曲线

步进电机启动时，必须有升速、降速过程，升降速的设计至关重要。如果设计不合适，将引起步进电机的堵转、失步、升降速过程慢等问题。升速过程由突跳频率加升速曲线组成（减速过程反之），理想的升降速曲线为指数曲线（见图 7-5），根据用户的负载情况选择不同的突跳频率和不同的指数曲线，以找到一条最理想的曲线，一般需要多次“试机”才行。突跳频率不宜过大。指数曲线在实际软件编程中比较麻烦，一般事先算好后存贮在 ROM 内，工作过程直接选取。

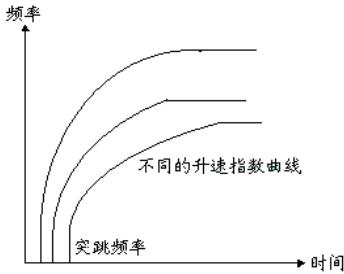


图 7-5 升降速曲线示意图

在本例中，由于主要说明其原理，所以没有做出升降速曲线，不过请读者在实际工作中对此进行注意。

### 7.5.3 解决方案设计与实现

具体实现时物理模型如图 7-6 所示，原理如下：物体的速度可以分解为 X 和 Y 两个方向，由速度分解的定理， $\vec{V} = \vec{V}_x + \vec{V}_y$ ，如图 7-7 所示，速率的值  $V = \sqrt{V_x^2 + V_y^2}$ 。所以，要知道物体的速率，就必须要知道 X、Y 两个方向的步进电机的转动速度，然后根据 X、Y 方向的速度做平方和运算后再开方，就求出了物体的运动速率的值。

不过，在实际设计的时候，需要注意一些问题：

- 物体在电机的控制下有 4 个运动方向，分别是 X 正方向，Y 正方向，X 负方向和 Y 负方向。其中，对于输入电机的控制信号来讲，不管是 X 方向还是 Y 方向，一般是由一根脉冲输出的信号来控制转动速度，而另有一根电平的信号线来控制转动的方向是顺时针还是逆时针。对应到我们的项目上来说，就是有两根信号线分别控制  $|\vec{V}_x|$  和  $|\vec{V}_y|$ ，而另有两根信号线来帮助确定究竟是沿 X 正方向还是负方向。

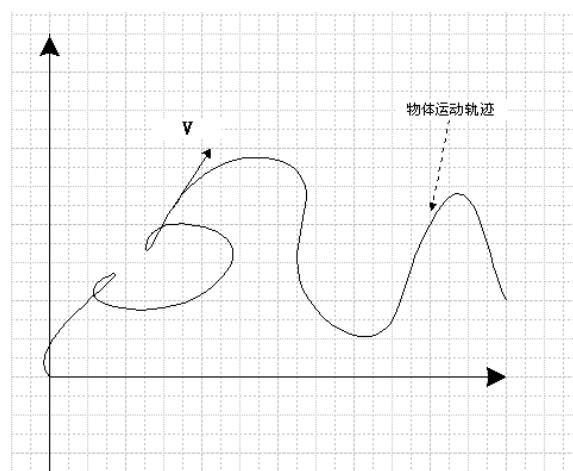


图 7-6 系统的物理模型

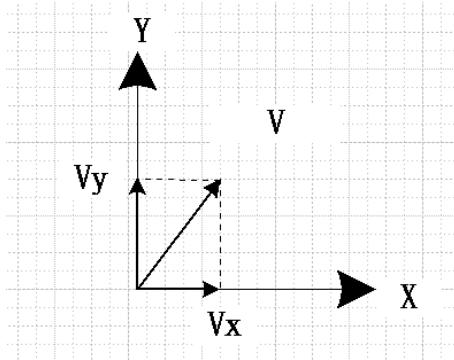


图 7-7 速度分解示意图

- 在设计方案确定之前，应该先估计计算时间的问题。也就是说，一块 89C51 能不能满足响应速度的问题。在本项目中，由于运算量稍大的只有乘法和开方运算，而且每隔 100ms 只需要进行一次，所以大体上可以估计出该时间是满足需要的。

由上面的分析，我们基本上已经可以确定硬件上的连接了。由于硬件原理图的讨论并不属于本节的范畴，所以我们简单地给出描述如下。

硬件实现方案说明：

硬件上将  $X$  方向的脉冲驱动信号输入到 INT1 的中断管脚上，方向电平输入到 P20 上， $Y$  方向的脉冲驱动信号输入到 INT0 的管脚上，方向电平输入到 P21 上，这样就可以通过对脉冲的计数来测出  $X$  方向及  $Y$  方向的速度。

另外有正转或者反转的输入控制，分别接入到 P01 和 P02 上，输出的几条管脚分别为：

P10: OPT0。

P11: FREE。

P12: DIR。

P13: CP。

另外有几条 I/O 口输出到 LED 上，用来显示目前的工作状态。它们分别为：

P14: 工作状态正常指示。

P15: 转动指示。

P17: 致命错误指示。

P16: 调试控制指示。

P22: 溢出指示。

P23: 通信指示。

现在，我们对任务及其解决方案已经基本明确，那么在具体开始编程之前，我们只需要对函数及功能进行划分就可以了。

读过以前的例子，相信读者最先想到的子函数应该是初始化函数 initial 了。在 intial 子函数中，需要做的有硬件（包括中断、串口、定时器等）的初始化，还有程序中需要使用的变量的初始化。在这个项目中，我们需要用的设备有：

- (1) 串口，用于程序的测试，以及调试阶段的观测界面。
- (2) 外部中断 0、1，用于  $X$ 、 $Y$  方向的脉冲计数。
- (3) 定时器，用于转动定时以及串口的数据传输率发生器。

这里的全局变量主要有 4 个，其中 Xpulse、Ypulse 用于记录上一个 100ms 中的 *X*、*Y* 方向的脉冲值，以控制现在的电机转动，另有 XpulseTemp 和 YpulseTemp 被外部中断程序使用，用于对这个 100ms 中的脉冲值进行累加计数。

初始化程序代码如下：

```
void initial(void)
{
 IP=0x10; //定义串口为高优先级中断
 IE=0x97; //允许串口、中断 0、1、定时器 0
 TCON=0x05;
 TMOD=0x21; //定时器 1 为自动装入(auto-load)方式
 TH0=0xB1; //10ms 产生中断 24MHz
 TL0=0xE0;
 TR0=1;

 PCON=0; //SMOD(PCON.7)=1 时, baudrate doubled. //smode=1 (11M 晶振时为 0)
 SCON=0xD0; //串行口工作方式 : 9 位 UART, 数据传输率可变
 TH1=0xf3;
 TL1=0xf3;
 PCON=0x80|PCON; //SMOD=1;数据传输率设置 : 9600 baud(E8—24MHz)
 TR1=1; //启动定时器 1

 //变量初始化
 Xpulse=0;
 Ypulse=0;
 XpulseTemp=0;
 YpulseTemp=0;
}
```

下面我们介绍一下对脉冲进行计数的程序。*X*、*Y* 方向的脉冲不定时的发出，而且脉冲波形很窄，如果使用查询的方式，一是加重程序的负担，另外也可能导致漏记脉冲，所以我们只能使用中断的方式。

前面已经介绍了，*X*、*Y* 方向的电机的转动方向有一根信号线控制，而速度（角度）由脉冲数来控制，所以我们要了解 *X*、*Y* 方向的电机转动速度（也就是物体的运动速度），只需要在中断中查询表示方向的信号线来确定电机是朝哪个方向转动的（在程序中反映为应该将这个脉冲记为正脉冲还是负脉冲）。

中断程序如下所示：

```
void Xpulse_INT(void) interrupt 2
{
 EX0=0;
 if(PinXClockWise) XpulseTemp++;
 else XpulseTemp--;
}
```

```

EX0=1;
}

void YPulse_INT(void) interrupt 0
{
 EX1=0;
 if(PinYClockWise) YPulseTemp++;
 else YPulseTemp--;

 EX1=1;
}

```

其中，上面程序中的 PinXClockWise 和 PinYClockWise 已经被宏定义为 P01 和 P02，即 X 方向和 Y 方向的正转/反转输入控制。

由于程序每 100ms 要控制电机转动一次，所以定时器中断程序自然是必不可少。但是，除非由特殊需求，否则在中断程序中进行过多的运算和处理并不是一个良好的编程习惯，它会导致其他任务的响应缓慢以及某些低优先级中断可能丢失的问题。在我们的程序中，我们将会在 100ms 的时候用一个 bit 类型的变量来记录 100ms 中断事件，该事件将在主程序中被发现，并触发相应的执行逻辑。另外，由于定时器无法产生 100ms 的中断，所以我们在初始化程序中对定时中断的设定为 10ms。在中断程序中，我们就需要对中断进行计数，到 100ms 的时候就将 XPulseTemp 和 YPulseTemp 中所纪录的值写入 XPulse 和 Ypulse 中，然后将 XPulseTemp 和 YPulseTemp 更新为 0，以保证可以正确纪录新的 100ms 中的脉冲值。

定时器中断服务程序流程图如图 7-8 所示。

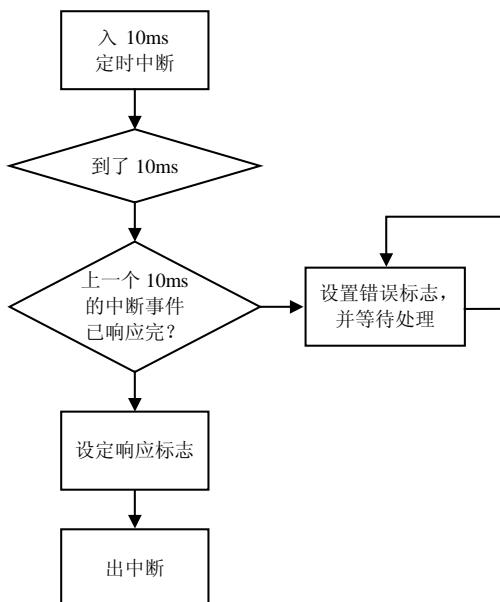


图 7-8 定时器中断服务程序

定时器中断源代码如下：

```

void TIMER(void) interrupt 1
{
 TH0=0xB1; //10ms 产生中断 24Mhz
 TL0=0xE0;
 counter10ms++;

 if(counter10ms>=10)
 {
 counter10ms=0;
 EvTimer=TRUE;
 XPulse=XPulseTemp;
 YPulse=YPulseTemp;
 XPulseTemp=0;
 YPulseTemp=0;
 }
}

```

从任务上来说，比较独立又在多处要使用的自然是转动的子程序了，这里将该程序取名 FuncRotate。该函数应该包含有转动方向（不妨取名为 BClockWise，以表示示顺时针或者逆时针方向）和转动步长（不妨取名 step，表示电机转动的步数）的信号参数，同时，为了方便使用，通过这个函数我们还应该能够得到当前转动角度的信息。当然，我们可以直接返回转动的角度，不过，由于电机转速的要求，如果在 100ms 以内要转动过大的角度，是会引起超时的。所以我们应该在转动的函数中规定一个安全的角度，如果要转动的角度超过了这个极限值，我们应该通知调用者调用失败。根据一般通用的函数调用准则，这种有可能涉及到调用失败的函数返回的应该是 BOOL 型变量 TRUE 或 FALSE。所以，我们只能把存放当前角度值的存储器指针作为参数，在函数中对指针指向值（PRegAngle）进行修改，以保证当前角度值的正确性。

另外，一般来说，在实际的工程中，程序在做什么工作都希望外部的指示灯能够显示出来，所以在本例中，如果正在转动，那么就让某个 LED 发光，以表示转动操作正在进行。51 单片机与该 LED 相连的 I/O 管脚被宏定义为 PinRotIndicator。

由于本例只是示例程序，所以在转动函数中没有设置这个极大值，读者可以在自己的实际工作中根据需要添加这方面的内容。函数 FuncRotate 的代码如下：

```

int FuncRotate(bit BClockWise, int step, int *PRegAngle)
{
 PinRotIndicator=TRUE;
 if (BClockWise) //如果是顺时针转动
 {
 PinDrvAntiClock=INVALID; //方向电平设置
 *PRegAngle+=step;
 if (*PRegAngle>=MAXSTEPS) *PRegAngle-=MAXSTEPS;
 }
}

```

```

 }
 else //逆时针转动
 {
 PinDrvAntiClock=VALID; //方向电平设置
 *PRegAngle-=step;
 if (*PRegAngle<0) *PRegAngle+=MAXSTEPS;
 }
 while(step) //如果需要转动的角度不为 0, 驱动 CP 发送脉冲
{
 PinDriver=INVALID;
 delay(delaycount);
 PinDriver=VALID;
 delay(delaycount);
 --step;
}
PinRotIndicator=FALSE;
return(TRUE); //返回成功
}

```

为了在测试的时候能够有效地对步进电机的转动速率与转动力矩的关系进行试验，即为了保证在我们需要的转动速率下电机能够正常的带动机械传动部分，也为了在与机械协同测试时能有一个观测点，我们将编写串口通信的程序，该程序可以接受微机串口传送过来的命令更改工作模式、转动速率，微机端的主程序可以根据工作模式的值来判断需要发送给微机的内容，微机通过程序将这些内容按照规定的格式显示出来，测试者可以根据显示来判断当前单片机的工作状态。

根据上面的需要，我们为单片机定义了 3 种工作模式：模式 0、模式 1 和模式 2。模式 0 为正常工作模式，也就是最终实际工作的模式。该模式下单片机只控制电机的转动等，而不通过串口输入任何内容；模式 1 为调试模式 1，该模式下，单片机发送出当前步进电机的角度（以步数的形式给出）；模式 2 为调试模式 2，该模式下，单片机不但发送出当前步进电机的角度（以步数的形式给出），而且还给出 X、Y 方向的物体运动速度（也是以 X、Y 方向的步进电机的步数的形式给出）。

为了协议的简洁，并保证协议能在规定的范围内调整工作模式和电机的转动速度，我们规定 PC 机端发出的命令只有一个字节，并规定字节的格式如下：

字节的第 0~5 为延时设置，用来控制脉冲发生的频率，也就是控制电机转动的速度。字节的剩下的高 3 个字节被用来作模式的控制，也就是说总共可以设置 8 种模式，实际只用了 3 个，在需要的时候，还可以对模式的类型进行扩展。

接收 PC 机端的控制命令的过程是这样的：首先通过收中断程序来将收到的字节保存，并用一个比特来记录这次收到命令的事件，然后在主程序中响应该事件，并对相应的参数进

行改动。在这里请读者注意，之所以把收到数据以后的处理放在主程序中，也是由于改动的方便性以及程序的模块化要求决定的。

因此，收命令的程序应有两部分，一是收数据中断服务程序，另外一部分是收到命令以后的参数改动部分。收数据中断服务程序流程如图 7-9 所示。

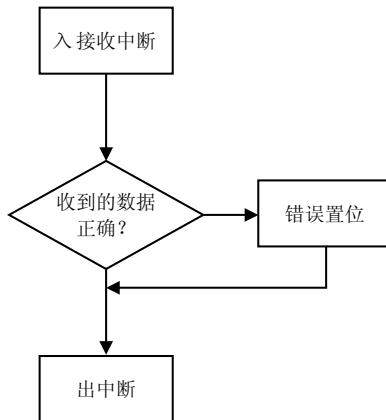


图 7-9 收数据中断服务程序

收中断服务程序代码如下：

```

void Rcv_INT(void) interrupt 4
{
 LampComm=TRUE;
 if(RI) //是否是收中断
 {
 ACC=SBUF;
 if(P==RB8)
 {
 rcvdata=ACC; //保存收到的内容
 EvRcv=TRUE;
 }
 RI=0;
 }
 LampComm=FALSE;
}

```

参数改动部分的程序代码如下：

```

if(EvRcv)
{
}

```

```

 delaycount=recvdata&0x1F;
 workmode=(recvdata&0xE0)/32;
 EvRcv=FALSE;
}

```

意思是：如果有收中断事件，那么就将 delaycount（转动物脉冲的延时）改动为接收到的数据的低 5 位的值，将 workmode（工作模式）改动为高 3 位的值。这部分程序为主程序代码的一部分。

另外，由于发送状态需要一个通用的发送程序，并且在测试中我们可以不太考虑效率上的需求，所以我们编制了一个名为 send 的子函数。该函数以等待的方式来发送规定地址指向的规定数目的字节。在我们工作在某种模式下时，我们只需要按规律将要发送的字节排在内存中，在调用此函数就可以了。

发送子函数源代码如下：

```

void send(char *temp, int j)
{
 int i=0;
 LampComm=TRUE; //发送指示灯亮
 EA=0;
 for(i=0;i<=j-1;i++) //准备发送数据
 {
 ACC=*(temp+i);
 TB8=P;
 SBUF=ACC;
 while(TI==0); //等待
 TI=0;
 }
 EA=1;
 LampComm=FALSE; //发送指示灯灭
}

```

在主程序中，我们需要调用该函数来发送调试的内容，该部分的代码如下：

```

if(workmode==1)
{
 send((char*)&RegAngle, 2);
}
else if(workmode==2)
{
 send((char*)&RegAngle, 2);
 send((char*)&XPulse, 4);
 send((char*)&YPulse, 4);
}

```

send 函数后面的第二个参数是与第一个参数的类型相匹配的，例如，我们在程序中定义

RegAngle 的类型为 int 型，该类型在 C51 中为 2 个字节，所以第二个参数就为 2，而其他的几个变量的类型为 long int，在 C51 中会用 4 个字节来表示，所以第二个参数就为 4。

编写完所有相关的子函数，剩下的就是主函数的编写了。该函数必须包含我们项目的所有要求，例如正转反转控制，正常测速控制等等。为了是读者清晰的了解整个过程，我们将主函数的流程图画出，如图 7-10 所示。

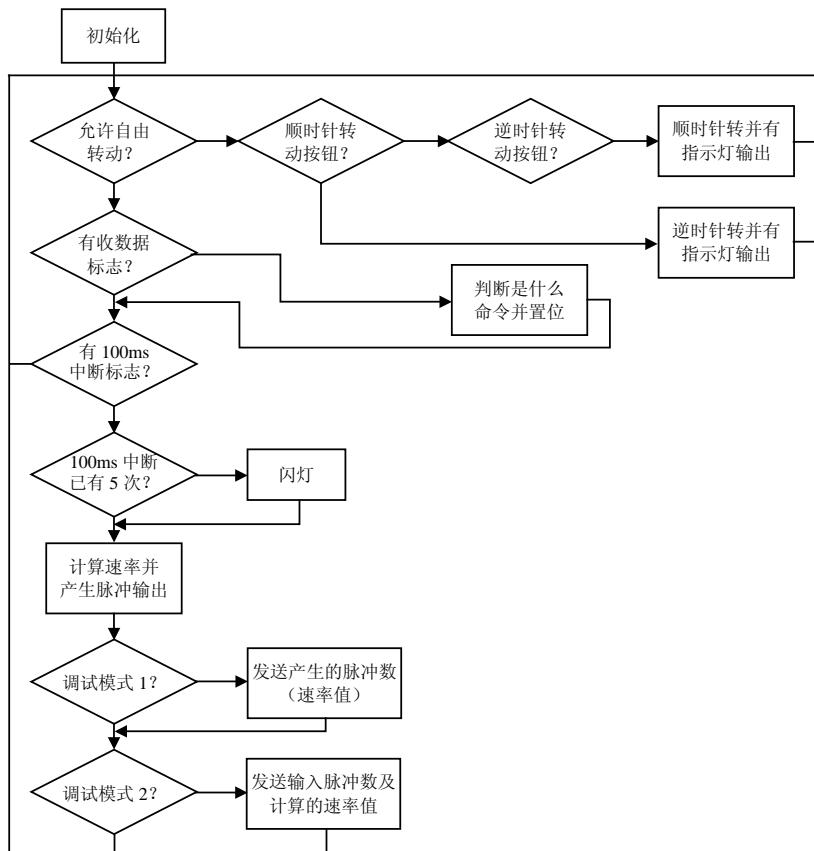


图 7-10 主程序流程图

根据流程图，我们可以写出主程序的代码如下：

```

void main(void)
{
 int RegAngle=0, RegNewAngle=0, temp=0;
 long int Xtmp=0, Ytmp=0;

 initial(); // 初始化子程序
 PinDriver=VALID;
 PinLamp1=FALSE;
 PinLamp2=FALSE;
 PinLamp3=FALSE;
 PinLamp4=FALSE;
 PinLamp5=FALSE;
}

```

```
PinLamp6=FALSE;

PinDrvOPT0=FALSE;
PinDrvFree=FALSE;
delay(1);

while(1)
{
 while(PinInFree==VALID)
 {
 TR0=0;
 PinDrvFree=TRUE;
 while(PinClockWiseRot==VALID)
 {
 PinDrvFree=FALSE;
 FuncRotate(ConstClockWise, 1, &RegAngle);
 delay(500);
 }
 while(PinAntiClockRot==VALID)
 {
 PinDrvFree=FALSE;
 FuncRotate(ConstAntiClock, 1, &RegAngle);
 delay(500);
 }
 TR0=1;
 EvTimer=FALSE;
 }
 PinDrvFree=FALSE;
 if(EvRcv)
 {
 delaycount=rcvdata&0x1F;
 workmode=(rcvdata&0xE0)/32;
 EvRcv=FALSE;
 }

 if (EvTimer)
 {
 ++debugtime;
 if(debugtime==5)
 {
 PinLampBlink=TRUE;
 }
 else if(debugtime==10)
 {
 }
```

```
PinLampBlink=FALSE;
debugtime=0;
}

RegNewAngle=XPulse*XPulse+YPulse*YPulse;
RegNewAngle=(int)sqrt((float)RegNewAngle);

temp=RegNewAngle-RegAngle;
if(temp>HALFMAXSTEPS)
{
 temp=temp-MAXSTEPS;
}
else if(temp<-HALFMAXSTEPS)
{
 temp=temp+MAXSTEPS;
}

if(temp>0)
{
 FuncRotate(ConstClockWise, temp, &RegAngle);
}
if(temp<0)
{
 FuncRotate(ConstAntiClock, -temp, &RegAngle);}
}

if(workmode==1)
{
 send((char*)&RegAngle, 2);
}
else if(workmode==2)
{
 send((char*)&RegAngle, 2);
 send((char*)&XPulse, 4);
 send((char*)&YPulse, 4);
}

EvTimer=FALSE;
}
```

下面是完整的源程序代码：

```
#include <stdio.h>
#include <reg51.h>
#include <math.h>

#define DEBUG
#define VALID 0
#define INVALID 1
#define TRUE 1
#define FALSE 0
#define ConstClockWise 1
#define ConstAntiClock 0

#define DELAY_VALVE 3

#define MAXSTEPS 400 //最大转动角度为360度，步长0.9度，故最大步长=400
#define HALFMAXSTEPS 200 //180度
#define ONEQUARTERSTEPS 100
#define THREEQUARTERSTEPS 300
#define OCTANT 50
#define MAXINPULSE 20
#define PinClockWiseRot P0_1 //输入，顺时针转动信号
#define PinAntiClockRot P0_2 //输入，逆时针转动信号
#define PinInFree P0_3 //输入，自由转动信号
#define PinDrvOPTO P1_0 //输出：opto.
#define PinDrvFree P1_1 //输出，FREE
#define PinDrvAntiClock P1_2 //输出，DIR
#define PinDriver P1_3 //输出，CP

#define PinXClockWise P2_0 //输入，X方向脉冲
#define PinYClockWise P2_1 //输入，Y方向脉冲

#define PinLamp1 P1_4
#define PinLamp2 P1_5
#define PinLamp3 P1_6
#define PinLamp4 P1_7

#define PinLamp5 P2_2
#define PinLamp6 P2_3
```

```
#define PinLampBlink PinLamp1
#define PinRotIndicator PinLamp2
#define LampFatalErrorPinLamp4
#define PinLampDebug PinLamp3
#define LampOVErr PinLamp5
#define LampComm PinLamp6

sbit P0_0=P0^0;
sbit P0_1=P0^1;
sbit P0_2=P0^2;
sbit P0_3=P0^3;
sbit P0_4=P0^4;
sbit P0_5=P0^5;
sbit P0_6=P0^6;
sbit P0_7=P0^7;

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;

sbit P2_0=P2^0;
sbit P2_1=P2^1;
sbit P2_2=P2^2;
sbit P2_3=P2^3;
sbit P2_4=P2^4;
sbit P2_5=P2^5;
sbit P2_6=P2^6;
sbit P2_7=P2^7;

sbit P3_2=P3^2;
sbit P3_3=P3^3;

bit EvTimer=FALSE;
```

```
bit EvRcv=FALSE;

char rcvdata=0;
char workmode=2;
int delaycount=20;

long int XPulse=0,YPulse=0;
long int XPulseTemp=0,YPulseTemp=0;

int debugYinput=0;
int debugXinput=0;
int counter10ms=0;
int debugtime=0;

void initial(void); //初始化子程序
void delay(int i);
int FuncRotate(bit BClockWise, int step, int *PRegAngle);
int FuncGoAngleZ(int *PRegAngle);
void send(char *temp, int j);

void main(void)
{
 int RegAngle=0,RegNewAngle=0,temp=0;
 long int Xtmp=0,Ytmp=0;

 initial(); //初始化子程序
 PinDriver=VALID;
 PinLamp1=FALSE;
 PinLamp2=FALSE;
 PinLamp3=FALSE;
 PinLamp4=FALSE;
 PinLamp5=FALSE;
 PinLamp6=FALSE;

 PinDrvOPT0=FALSE;
 PinDrvFree=FALSE;
 delay(1);
```

```

while(1)
{
 while(PinInFree==VALID)
 {
 TR0=0;
 PinDrvFree=TRUE;
 while(PinClockWiseRot==VALID) //按钮控制顺时针转动
 {
 PinDrvFree=FALSE;
 FuncRotate(ConstClockWise, 1, &RegAngle);
 delay(500);
 }
 while(PinAntiClockRot==VALID) //按钮控制逆时针转动
 {
 PinDrvFree=FALSE;
 FuncRotate(ConstAntiClock, 1, &RegAngle);
 delay(500);
 }
 TR0=1;
 EvTimer=FALSE;
 }

 PinDrvFree=FALSE;
 if(EvRcv) //串口收到命令
 {
 delaycount=rcvdata&0x1F;
 workmode=(rcvdata&0xE0)/32;
 EvRcv=FALSE;
 }

 if (EvTimer) //定时事件触发
 {
 ++debugtime;
 if(debugtime==5) //闪灯
 {
 PinLampBlink=TRUE;
 }
 else if(debugtime==10)
 {
 PinLampBlink=FALSE;
 }
 }
}

```

```
 debugtime=0;
 }

RegNewAngle=XPulse*XPulse+YPulse*YPulse;//计算
RegNewAngle=(int)sqrt((float)RegNewAngle);

temp=RegNewAngle-RegAngle; //转动
if(temp>HALFMAXSTEPS)
{
 temp=temp-MAXSTEPS;
}
else if(temp<-HALFMAXSTEPS)
{
 temp=temp+MAXSTEPS;
}

if(temp>0)
{
 FuncRotate(ConstClockWise,temp,&RegAngle);
}
if(temp<0)
{
 FuncRotate(ConstAntiClock,-temp,&RegAngle);
}

if(workmode==1) //工作模式控制
{
 send((char*)&RegAngle,2);
}
else if(workmode==2)
{
 send((char*)&RegAngle,2);
 send((char*)&XPulse,4);
 send((char*)&YPulse,4);
}

EvTimer=FALSE; //事件结束
}
```

```

void initial(void)
{
 IP=0x10; //定义串口为高优先级中断
 IE=0x97; //允许串口、中断 0、1、定时器 0
 TCON=0x05;
 TMOD=0x21; //定时器 1 为自动装入(auto-load)方式
 TH0=0xB1; //10ms 产生中断 24MHz
 TL0=0xE0;
 TR0=1;

 PCON=0; //SMOD(PCON.7)=1 时，数据传输速率加倍
 //smode=1 (11M 晶振时为 0)
 SCON=0xD0; //串行口工作方式：9 位 UART，数据传输率可
 //变
 TH1=0xf3;
 TL1=0xf3;
 PCON=0x80|PCON; //SMOD=1；数据传输率设置：9600
 //baud(E8--24MHz)
 TR1=1; //启动定时器 1

 XPulse=0;
 YPulse=0;
 XpulseTemp=0;
 YpulseTemp=0;

}

void send(char *temp, int j)
{
 int i=0;
 LampComm=TRUE;
 EA=0;
 for(i=0;i<=j-1;i++)
 {
 ACC=*(temp+i);
 TB8=P;
}

```

```
 SBUF=ACC;
 while(TI==0);
 TI=0;
 }
 EA=1;
 LampComm=FALSE;
}

int FuncRotate(bit BClockWise, int step, int *PRegAngle) //转动子程序
{

 PinRotIndicator=TRUE;
 if (BClockWise) //转动方向
 {
 PinDrvAntiClock=INVALID;
 *PRegAngle+=step;
 if (*PRegAngle>=MAXSTEPS) *PRegAngle-=MAXSTEPS;
 }
 else
 {
 PinDrvAntiClock=VALID;
 *PRegAngle-=step;
 if (*PRegAngle<0) *PRegAngle+=MAXSTEPS;
 }

 while(step) //转动步数
 {
 PinDriver=INVALID;
 delay(delaycount);
 PinDriver=VALID;
 delay(delaycount);
 --step;
 }
 PinRotIndicator=FALSE;
 return(TRUE);
}

void delay(int i) //延时子程序
```

```

{
 int j=0;
 i=i*DELAY_VALVE;

 while (j<i) j++;

}

void Xpulse_INT(void) interrupt 2 //X 方向脉冲计数
{
 EX0=0;

 if(PinXClockWise) XpulseTemp++;
 else XpulseTemp--;

 EX0=1;
}

void Ypulse_INT(void) interrupt 0 //Y 方向脉冲计数
{
 EX1=0;

 if(PinYClockWise) YpulseTemp++;
 else YpulseTemp--;

 EX1=1;
}

void TIMER(void) interrupt 1 //定时中断处理程序
{
 TH0=0xB1; //10ms 产生中断 24MHz
 TL0=0xE0;

 counter10ms++;

 if(counter10ms>=10)
 {
 counter10ms=0;
 EvTimer=TRUE; //设置中断事件
 Xpulse=XpulseTemp;
 Ypulse=YpulseTemp;
 XpulseTemp=0;
 }
}

```

```
YPulseTemp=0;
}

}

void Rcv_INT(void) interrupt 4 //接收中断
{
 LampComm=TRUE;
 if(RI)
 {
 ACC=SBUF;
 if(P==RB8)
 {
 recvdata=ACC;
 EvRcv=TRUE;
 }

 RI=0;

 }
 LampComm=FALSE;
}
```

# 第 8 章 单片机通信

在微机测控技术领域，对较大规模的测控系统经常需要双机或者多机进行通信。微机与微机之间的距离可能是近程的（几米甚至同一块电路板上），也可能是远程的（几百米以上），信息交换的方式可以使用并行通信，也可以使用串行通信。一般情况下，多使用占用资源较少的串行通信。对 8051 单片机而言，其在结构、性能和经济成本上均为点对点的串行通信和多机串行通信提供了很好的条件。本章将以实例的形式详细介绍使用 8051 单片机的双机和多机通信系统的设计方法。

## 8.1 串口通信

串行通信是一种能够把二进制数据按比特传送的通信方式，适用于远程通信和远程控制。AT89C51 单片机有一个串行接口。本节讨论它的串口通信问题。

### 8.1.1 串行通信基础

串行通信一般被分为同步和异步两种方式。

#### (1) 同步串行通信

所谓同步方式，是一种连续串行传送数据的通信方式，一次通信只能传送一帧信息，同时需要有至少一条的信号线来为对方提供和数据传送同步的精确时钟，以确保数据无误传输。同步帧一般有固定的帧格式，其中包含了帧类型、帧长度或结束标志、校验位等若干内容。同步帧一般使用在需要高速串行传输数据的场合，一般在异步串行传输速率不能达到的情况下才考虑使用同步帧。目前同步串行通信的传输速率可高达百兆比特每秒 (Mbit/s)。由于使用 51 系列单片机时甚少涉及同步串行通信的内容，故本书也不再对此深入讨论。有兴趣的读者可以参考其他硬件、计算机网络或者数字通信的书籍。

#### (2) 异步串行通信

异步串行通信则是另外一种串行通信的形式，也是使用 51 系列单片机比较常用的一种串行通信方式。它具有实现比较简单的优点，但缺点是传送速率很低，一般使用时均为 9.6kbit/s，但也可以达到更高的速率。它通常以字符或者字节为单位，组成字符帧来传送。它不需要信号线来传送数据的时钟，通信双方均使用自己的时钟来控制数据的发送和接收。需要注意，通信双方自己所具有的时钟是相互独立和不同步的。

那么，在收发端都使用自己的时钟的情况下是如何协调数据的收发的呢？原来这就是字符帧格式的作用。在平时的时候，发送线为高电平（即为 1），每当接收端检测到传输线上发送过来的电平为低电平逻辑 0，即字符帧中的起始位时，就知道一帧信息已经开始。在收到规定个数的数据以后，就停止接收。此时信号线上的电平也应该为逻辑高电平 1，即结束位。

图8-1说明了字符帧的一般格式。一个字符帧一般由起始位、数据位、奇偶校验位、停止位等部分组成。

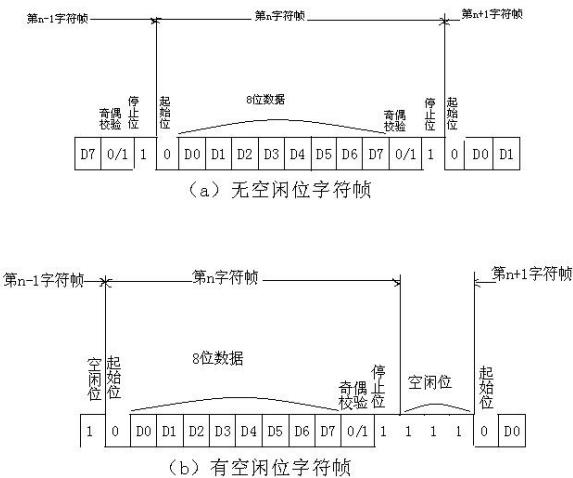


图8-1 异步通信的字符帧格式

- 起始位

起始位一般位于字符帧的开头，一般为逻辑低电平0，持续时间根据不同的系统或设置的不同而不同，一般为1位。用于向接收设备表示发送端开始发送一帧信息。

- 数据位

数据位在起始位之后，用户可以根据情况选取5位、6位、7位或8位，低位在前高位在后。

- 奇偶校验位

奇偶校验位位于数据位之后，用于表征串行通信采用奇校验还是偶校验，由用户根据需要决定。也可以不使用这一位。

- 停止位

停止位位于一帧的最末，为逻辑高电平1，通常可取1位、1.5位或2位。用于向接收端表示一帧字符信息已发送完毕，也为发送下一帧字符做准备。

- 空闲位

在两个相邻帧之间可以有若干空闲位，以保证如果接收端在数据处理能力不足的情况下仍能可靠传输。也可以没有空闲位直接发送数据。

在异步串行通信中，字符帧格式和数据传输率都是很重要的指标，所以需要由用户根据实际情况而定。

### 8.1.2 单片机串口使用

虽然使用51单片机可以实现同步帧的收发，但是，由于51单片机自己带有串行通信口来支持异步数据的传递，所以一般来说使用得更多的是直接利用该串口来进行异步数据传输。51单片机的串行口由串行口控制寄存器SCON、发送电路、接收电路等组成。本书以C51程序的编写为主要内容，故在此只说明使用单片机串口进行数据发送、接收的操作过程和使用方

法。

对程序编制来说，需要了解的有串口收发缓存、串口控制寄存器、串口工作模式及工作速率的选定这几个方面，下面分别讨论。

### 1. 缓存 SBUF

在数据收发过程中，串口收发的数据都存放在一个称为 SBUF 的 8 位寄存器中。如果要发送数据，则将数据先移到累加器 ACC 中，然后再从 ACC 中移到 SBUF。如果是接收数据，则一般在接收中断程序中把 SBUF 的数据取出，即为串口上实际传送的数据。不过在这里要注意，在单片机的硬件上，收发虽然共用一个选口地址 SBUF (99H)，但实际上对 SBUF 读和写时并不是访问的一个寄存器。这样才有可能使单片机能够同时进行收发的工作。

### 2. 串口控制寄存器

单片机对串口的工作控制还通过串口控制寄存器 SCON 和电源控制寄存器 PCON 来实现。SCON 和 PCON 都是特殊功能寄存器，选口地址分别为 98H 和 87H。SCON 的各个比特位的定义如图 8-2 所示，PCON 的各个比特位的定义如图 8-3 所示。

- SCON 各位定义

SM0 和 SM1：为串行口工作模式设置位。如果设置为模式 0，表示为同步移位寄存器工作方式，所用的数据传输率为晶振频率的 1/12；模式 1 表示为 10 位异步收发模式，数据传输率由定时器来控制；模式 2 表示为 11 位异步收发模式，数据传输率为晶振频率的 1/32 或者 1/64；模式 3 为 11 位异步收发模式，数据传输率由定时器来控制。更详细的内容将在下文的串口工作模式部分来说明。

SM2：多机通信控制位。主要在模式 2 和模式 3 下使用。在模式 0 时，SM2 不用，应设置为 0 状态，在模式 1 下，SM2 也应该设置为 0，此时 RI 只有在接收电路接收到停止位时才被激活，并自动发出串口中断请求。在模式 2 或者模式 3 下，如果该比特为 0，串口以单机发送或者接收方式工作，TI 和 RI 以正常方式被激活，但不引起中断请求；若该比特为 1 并且 SCON 中的 RB8 也被置位时，RI 不仅被激活而且可以向 CPU 请求中断。

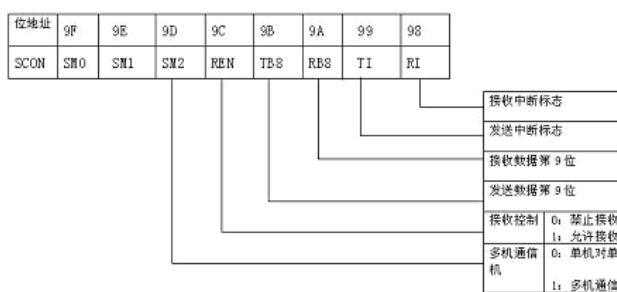


图 8-2 SCON 中各位定义

REN：允许接收控制位。为 0 时禁止串口接收，为 1 时允许串口接收。

TB8：为数据发送第 9 位，用于模式 2 和模式 3。由软件更改。

RB8：为数据接收第 9 位。用于模式 2 和模式 3。在模式 1 中，如果 SM2 为 0，则 RB8 用于存放接收到的停止位。在模式 0 下不使用该比特。

TI：发送中断标志位。用于指示一帧数据是否发送完毕。在模式0下，当发送电路发送完第8位数据时，TI由硬件置位；在其他模式下，TI在发送电路开始发送停止位时置位。也就是说，TI必须由软件复位，硬件置位。故CPU查询该比特可知一帧信息是否已发送完毕。

RI：接收中断标志位，用于指示一帧数据是否接受完毕。在模式1下，RI在接收电路接收到第8位数据时由硬件置位；在其他模式下，TI在发送电路开始发送停止位的中间位置时置位。RI也由软件复位，也可以供CPU查询，以决定CPU是否需要从SBUF中提取接收到的数据。

- PCON中各位的定义，如图8-3所示。

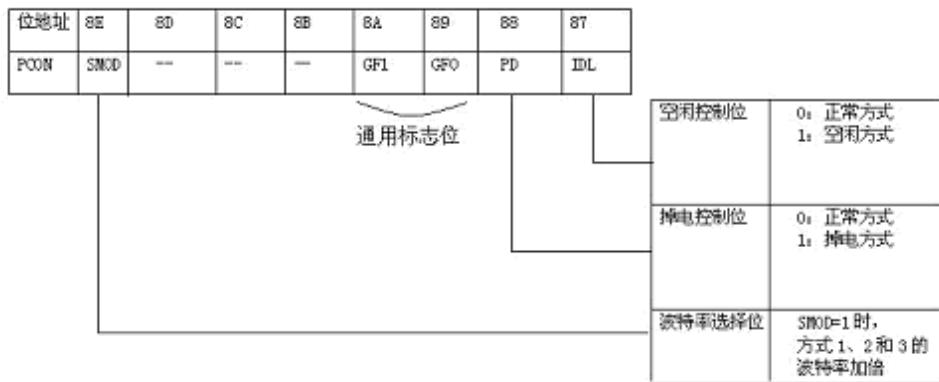


图8-3 PCON中各位定义

SMOD：数据传输率选择位。在模式1、模式2、模式3中，串行通信数据传输率在SMOD为1时通信数据传输率提高1倍。

PCON的其他各位与串口通信无关，在此就不讲了。

### 3. 串口的工作模式

上文已经说明，单片机的串口有4种工作模式：模式0，表示为同步移位寄存器工作方式；模式1表示为10位异步收发模式；模式2表示为11位异步收发模式；模式3为11位异步收发模式。下面分别说明。

#### • 模式0

在模式0下，串口的SBUF是作为同步的移位寄存器使用的。在串行口发送时，SBUF相当于一个并行进入、串行输出的移位寄存器，由单片机的内部总线并行接收8位数据，并从TxD信号线上串行输出。在接收操作时，它又相当于一个串行输入、并行输出的移位寄存器。该模式下，SM2、RB8、TB8不起作用。

发送操作在TI=0时进行，CPU将数据移入SBUF之后，RxD线上即可发出8位数据，TxD上发送同步脉冲。8位数据发送完后，TI由硬件置位，并在中断允许的情况下会向CPU请求中断。CPU响应中断后先用软件使TI清零，然后再给SBUF送下一个需要发送的字符，以重复上面的过程。

接收过程是在RI=0和REN=1的条件下启动的。此时，串行数据由RxD线输入，TxD线输出同步脉冲。接收电路接收到8位数据后，RI自动置位并发出串行口中断请求。CPU查询到

RI 为 1 或者响应中断以后便将 SBUF 中的数据送到累加器。RI 也需要由软件复位。

应该指出，串行口工作模式 0 并不是一个同步串口通信的方式。它的主要用途是外面的同步移位寄存器相接，以达到扩展一个并行口的目的。

- 模式 1

在模式 1 下，串行口设定为 10 位异步通信方式，字符帧中除了 8 位数据位外，还有一位起始位和一位停止位。发送位和停止位是硬件自动产生的。

发送操作过程如下：在 TI=0 时，发送电路就自动将 SBUF 的数据加上起始位和停止位后依次发出。发送结束以后将 TI 置位，并同时自动让 TxD 保持为逻辑高电平 1。TI 最终由软件复位。

接收操作在 RI=0 和 REN=1 条件下运行，这一点与在模式 0 下工作的情况是相同的。平常接收电路对高电平的 RxD 采样，采样脉冲频率是接收时钟的 16 倍。当接收电路连续 8 次采样到 RxD 线为低电平时，相应的检测器便可以确认 RxD 线上有了起始位。此后，接收电路就改为对第 7、8、9 三个脉冲采样到的值进行位检测，并以三中取二的原则来确定所采样的数据的值。

在接收到停止位时，接收电路必须同时满足下面两个条件：RI=0 和 SM2=0 或者接收到的停止位为 1，才能把接收到的 8 位字符存放到 SBUF 中；把停止位送入 RB8 中，并使 RI=1 和发出串口中断请求。若上述条件不满足，则这次收到的数据就会被舍去，也就是丢失了 1 组接收数据。

注意在本模式下，发送时钟、接收时钟和通信数据传输率都是由定时器溢出率脉冲经过 32 分频得到的，并且由 SMOD 来控制是否倍频。因此，模式 1 的数据传输率是可变的。这一点同样适用于模式 3。

- 模式 2

模式 2 是 11 位异步收发模式。它的数据传输率由单片机的主频 32 或 64 分频提供，具体是多少分频依赖于 SMOD 的设置。

模式 2 的发送过程与模式 1 类似，唯一不同的就是它的数据位有 9 位，包括 SBUF 中的 8 位数据位和 TB8 中的 1 位，可以是奇偶校验或者其他控制内容。

模式 2 的接收过程也与模式 1 类似，但是模式 1 中 RB8 存放的是停止位，模式 2 中存放的是第 9 位数据位。所以本模式下必须满足接收有效字符的条件是：RI=0 和 SM2=0 或者接收到的第 9 个数据位为 1。

其实，上面所说的第一个条件是要求 SBUF 空，即用户应预先读走 SBUF 中的信息，好让接收电路确认它已空。第二个条件是提供了利用 SM2 和第 9 个数据位共同对接收加以控制。若第 9 个数据位是奇偶校验，则可让 SM2=0 以确保串口能可靠接收；如果是接收控制，则可让 SM2=1，然后依靠第 9 位数据的状态来确定接收是否有效。

- 模式 3

模式 3 基本上与模式 2 相同，唯一的区别就是通信的数据传输率有所不同：模式 2 的数据传输率由单片机的主频经过 32 或者 64 分频以后提供；模式 3 的数据传输率则由定时器的溢出率经过 32 分频以后提供，故数据传输率是可变的。

#### 4. 串口通信的数据传输率的计算

在模式 0 下，串口的数据传输率是固定的，它的值为单片机的晶振频率的 1/12。

在模式 2 下，SMOD=0 时，通信的数据传输率为主机频率的 1/64；SMOD=1 时，数据传输率为主机频率的 1/32。

在模式 1 和模式 3 下，通信数据传输率是由定时器的溢出频率来决定的，所以数据传输率可以变化的范围较大。相应的公式为：

$$\text{数据传输率} = \frac{2^{SMOD}}{32} \cdot \text{定时器 T1 溢出率}$$

定时器 T1 的溢出率的计算公式为：

$$\text{定时器 T1 溢出率} = \frac{f_{osc}}{12} \left( \frac{1}{2^k - \text{初值}} \right)$$

因此，由上面的两个式子，就可以得到模式 1 或者模式 3 的数据传输率计算公式：

$$\text{数据传输率} = \frac{2^{SMOD}}{32} \cdot \frac{f_{osc}}{12} \cdot \left( \frac{1}{2^k - \text{初值}} \right)$$

其实，定时器 T1 通常采用工作模式 2，因为定时器 T1 在模式 2 工作，即自动载入模式工作时，不但使操作简单，也可以避免因为重装初值（时间常数）而带来的定时误差。

表 8-1 列出了常用的一些数据传输率和晶振频率下的定时器 T1 的初装值，以供大家使用方便。

表 8-1 常用的一些数据传输率和晶振频率下对应的定时器 T1 的初装值

| 数据传输率                 | fosc | SMOD | 定时器 T1 |      |       |
|-----------------------|------|------|--------|------|-------|
|                       |      |      | C/T    | 所选方式 | 相应初值  |
| 0.5Mbit/s (串行口方式 0)   | 6MHz | ×    | ×      | ×    | ×     |
| 187.5kbit/s (串行口方式 2) | 6MHz | 1    | ×      | ×    | ×     |
| 19.2kbit/s (方式 1 或 3) | 6MHz | 1    | 0      | 2    | FEH   |
| 9.6kbit/s (方式 1 或 3)  | 6MHz | 1    | 0      | 2    | FDH   |
| 4.8kbit/s (方式 1 或 3)  | 6MHz | 0    | 0      | 2    | FDH   |
| 2.4kbit/s (方式 1 或 3)  | 6MHz | 0    | 0      | 2    | FAH   |
| 1.2kbit/s (方式 1 或 3)  | 6MHz | 0    | 0      | 2    | F4H   |
| 0.6Kbit/s (方式 1 或 3)  | 6MHz | 0    | 0      | 2    | E8H   |
| 110bit/s (方式 1 或 3)   | 6MHz | 0    | 0      | 2    | 72H   |
| 55bit/s (方式 1 或 3)    | 6MHz | 0    | 0      | 1    | FEEBH |

## 8.2 单片机点对点通信

使用 8051 单片机自带的串口通信模块，可以方便地在两台单片机之间进行点对点通信，本节将对这部分内容进行介绍。

### 8.2.1 通信接口设计

如果两个 8051 应用系统之间的距离很短，可以通过将两个 8051 的串行接口直接相连的

方法实现双机通信，连接时注意要将一方的 TxD 与另一方的 RxD 引脚连接，如图 8-4 所示。

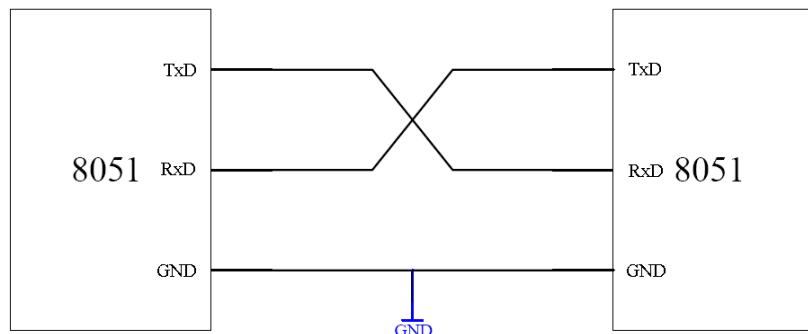


图 8-4 单片机点对点异步通信接口

如果通信距离较远，可以利用 RS-232 接口延长通信距离。前面章节已经介绍过，使用 RS-232 接口进行异步通信，必须要将单片机的 TTL 电平转换为 RS-232 电平，这就需要在通信双方的单片机接口部分增加 RS-232 电气转换接口。在实际应用中，常用 Maxim 公司的 MAX232 集成芯片构成这样的接口电路，图 8-5 所示为使用 RS-232 接口电路进行双机通信的电路图。

## 8.2.2 单片机点对点通信程序设计

无论是直接将单片机串口相连还是使用 RS-232 接口延长通信距离，单片机的点对点通信的程序设计都是一样的。本节将通过一个具体的实例介绍其设计方法。在实际应用中，很多时候单片机之间的通信环境都是比较好的，协议往往没有那么复杂，本节就介绍一个常用的简单数据传输的通信过程。这里规定协议内容如下：

- 通信双方均使用 9 600bit/s 的速率传送数据，使用主从式通信，主机发送数据，从机接收数据，双方在发送数据和接收数据时使用查询方式。
- 双机开始通信时，主机发送呼叫信号 06H 启动握手过程，询问从机是否可以接收数据。
- 从机接收到握手信号后，如果同意接收数据则回送应答信号 00H，表示可以接收，否则发送应答信号 15H 表示暂时无法接收数据。
- 主机在发送呼叫信号后等待，直到接收到从机的应答信号 00H 时，才确认完成握手过程，开始将数据缓冲区的内容发送给从机，如果接收到其他信息，主机将继续向从机呼叫。
- 从机在接收完数据后，将根据最后的校验字节判断数据接收是否正确，若接收正确，则向主机发送 0FH 信号，表示接收成功，若接收错误，则发送 F0H 信号，表示错误，并请求重发。
- 主机接收到 0FH，则通信结束，接收到其他任何信号都将导致主机重新发送这组数据。

通信协议中，主机发送的数据的格式如下：

| 字节数 n | 数据 1 | 数据 2 | ..... | 数据 n | 字节奇偶校验 |
|-------|------|------|-------|------|--------|
|-------|------|------|-------|------|--------|

- 字节数 n 由主机向从机发送的数据个数；

- 数据 1~数据  $n$  主机向从机发送的  $n$  个数据;
- 字节奇偶校验 字节数  $n$ 、数据 1、数据 2、……、数据  $n$  共  $n+1$  个字节相异或的结果，用于数据校验。

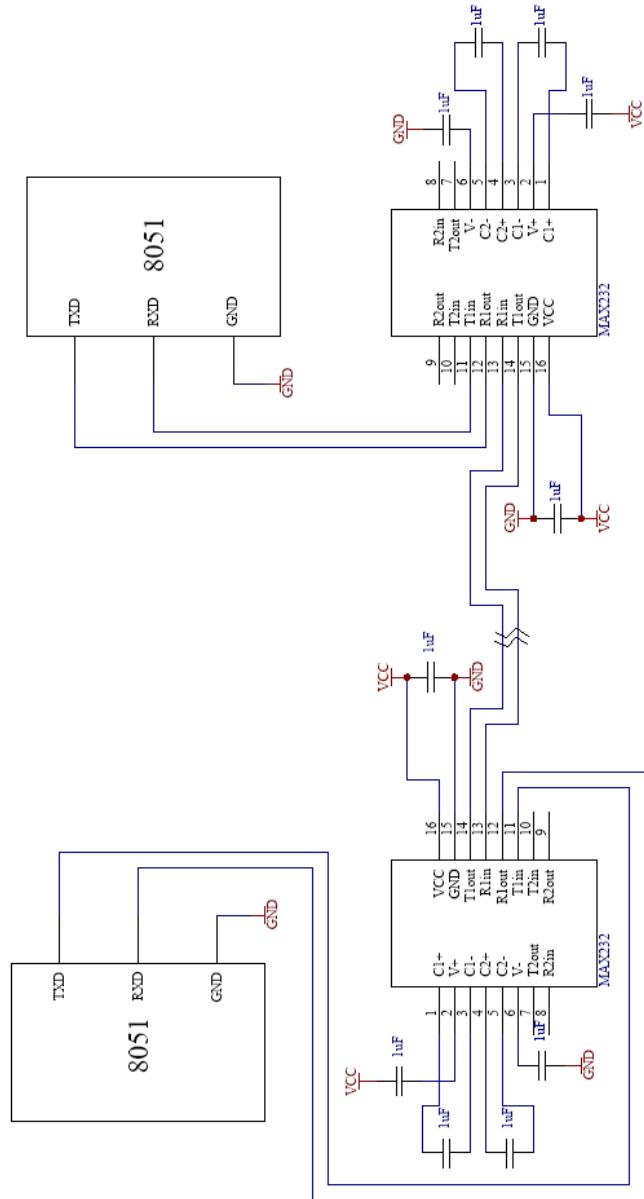


图 8-5 使用 RS-232 接口进行单片机点对点通信

下面分别介绍主机和从机的通信实现过程。

### 1. 主机通信部分

主机通信程序大体上可以分为 4 个部分，分别为预定义及全局变量部分、程序初始化部分、数据通信流程和发送数据部分。

### (1) 预定义及全局变量部分

这一部分中，主要声明程序中用到的预定义和子函数。预定义部分对程序中使用的握手信号进行了规定和定义，程序中用到的握手信号共有 5 个，其定义和内容如表 8-2 所示。

表 8-2

握手信号定义

| 信号   | 宏定义         | 说明                   |
|------|-------------|----------------------|
| 0x06 | <u>RDY</u>  | 主机开始通信时发送的呼叫信号       |
| 0x15 | <u>BUSY</u> | 从机“忙”应答，表示从机暂时无法接收数据 |
| 0x00 | <u>OK</u>   | 从机准备好，表示从机可以接收数据     |
| 0x0f | <u>SUCC</u> | 数据传送成功               |
| 0xf0 | <u>ERR</u>  | 数据传送错误               |

此外，这部分程序还对整个程序中用到的 3 个子函数 `init_serial()`、`send_data()` 和 `delay10ms()` 作出了声明，以便后面程序调用。

### (2) 程序初始化部分

程序初始化部分中，主要是对数据缓冲区以及串口部分的初始化。程序中定义的数据缓冲区部分最大为 64 个字节，通过读取 P0 口的内容完成对数据缓冲区部分的初始化。程序中每隔 100ms 就读取一次 P0 口的数据，并将其送入缓冲区，如果读取到的数据为 0x00，则表明数据读取完毕，程序将缓冲区的最后一个字节置“\0”表示数据的结束。这部分程序的代码内容如下，其中程序的延时部分使用 `delay10ms()` 函数实现：

```
char buf[MAX_LEN];
unsigned char i = 0;
unsigned char tmp = BUSY;

/* 为缓冲区赋初值 */
P0 = 0xff;
while(P1 != 0) // 每隔 100ms 从 P0 口读取，若读取到 0 则表明数据采集结束
{
 *(buf+i) = P0;
 delay10ms(10); // 延时 100ms
 P0 = 0xff;
 i++;
}
*(buf+i) = 0; // 缓冲区最后一个字节为 0 表示数据结束
/* 串口初始化 */
init_serial(); // 初始化串口
EA = 0; // 关闭所有中断
```

上面的程序中，通过调用 `init_serial()` 函数实现对串口的初始化。`init_serial()` 函数中，定义串口的工作方式为工作方式 1，波特率设为 9600bit/s，单片机晶振为 11.0592MHz，该函数的实现代码如下：

```
/* 初始化串口 */
```

```

void init_serial()
{
 TMOD = 0x20; // 定时器 T1 使用工作方式 2
 TH1 = 250; // 设置初值
 TL1 = 250;
 TR1 = 1; // 开始计时
 PCON = 0x80; // SMOD = 1
 SCON = 0x50; // 工作方式 1, 波特率 9600bit/s, 允许接收
}

```

### (3) 数据通信流程

主机部分的数据通信的基本流程如下：

- 主机发送`_RDY_`信号询问是否可以发送数据，随后主机进入等待接收状态，等待从机的应答信息。
- 如果接收到的应答信号为`_BUSY_`，表明从机现在处于“忙”状态，暂时无法接收数据，主机将反复发送`_RDY_`信号查询，直至从机空闲。如果接收到的信号为`_OK_`，表示从机可以接收数据，主机将调用`send_data()`函数完成数据发送过程。
- 发送完数据后，主机等待从机的校验信号，如果接收到`_SUCC_`信号，表示发送成功，通信结束，否则主机将重新发送数据，直至发送成功。

该部分程序对应的流程图如图 8-6 所示。

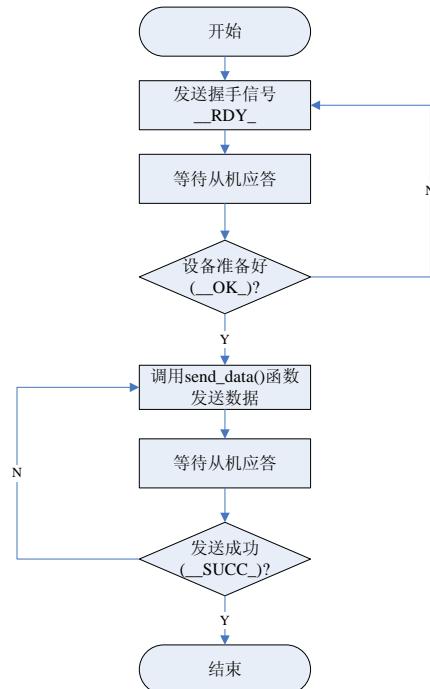


图 8-6 主机部分数据通信流程

具体的实现代码如下：

```

/* 发送握手信号 06H */
TI = 0;
SBUF = __RDY_;
while(!TI);
TI = 0;
/* 接收应答信息，如果接收的信号为 00H，表示从机允许接收 */
while(tmp != __OK__)
{
 RI = 0;
 while(!RI);
 tmp = SBUF;
 RI = 0;
}
/* 发送数据并接收校验信息，如果接收的信号为 0FH，表示从机接收成功，否则将重新发送该组
数据 */
tmp = __ERR_;
while(tmp != __SUCC__)
{
 send_data(buf); // 发送数据
 RI = 0;
 while(!RI);
 tmp = SBUF;
 RI = 0;
}
while(1); // 程序结束，进入死循环

```

#### (4) 发送数据部分

具体的数据发送过程是调用子函数 `send_data()` 实现的，这个函数的实现步骤如下：

- (1) 程序首先检查缓冲区中数据部分的长度，并将其保存在变量 `len` 中。
- (2) 发送数据长度 `len` 作为数据部分的第一个字节，并将该值赋给变量 `ecc`，开始校验过程。
- (3) 陆续发送缓冲区中的数据，在发送数据的同时，计算整个数据部分的校验字节（相异或）。
- (4) 发送完数据后，最后发送校验字节，数据部分发送完毕。

上述的具体流程如图 8-7 所示。

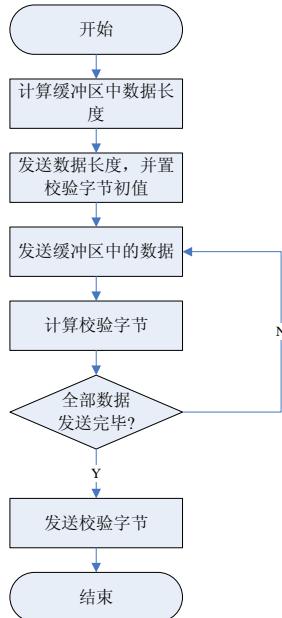


图 8-7 send\_data() 函数流程

send\_data() 函数的代码如下：

```

/* 发送数据 */
void send_data(unsigned char *buf)
{
 unsigned char len; // 保存数据长度
 unsigned char ecc; // 保存校验字节

 len = strlen(buf); // 计算要发送数据的长度
 ecc = len; // 开始进行校验字节计算
 /* 发送数据长度 */
 TI = 0;
 SBUF = len; // 发送长度
 while(!TI);

 TI = 0;
 /* 发送数据 */
 for(i=0; i<len; i++)
 {
 ecc = ecc ^ (*buf); // 计算校验字节
 SBUF = *buf; // 发送数据
 buf++;
 while(!TI);
 TI = 0;
 }
}

```

```

/* 发送校验字节 */
SBUF = ecc; // 发送校验字节
while(!TI);
TI = 0;
}

```

## 2. 从机通信部分

与主机通信程序相似，从机通信程序也可以分为预定义及全局变量部分、程序初始化部分、数据通信流程和接收数据部分共 4 个部分，下面分别对其进行介绍。

### (1) 预定义及全局变量部分

预定义部分的宏定义与主机通信程序是相同的，子函数声明包括 3 个函数，分别为串口初始化函数 init\_serial()、接收数据函数 recv\_data() 和蜂鸣函数 Beep\_ok()。其中 Beep\_ok() 函数在数据成功接收后被调用，它控制系统的蜂鸣器蜂鸣表示数据接收成功，因为与本章中讲述的内容关系不大，该函数的实现代码并没有在实例中给出。

### (2) 程序初始化部分

该部分的代码主要是对串口的初始化，注意这里为了保证通信双方可以正常通信，需要使从机的串口设置与主机的串口设置相同，因此，程序中的 init\_serial() 函数的代码与主机通信程序中的完全相同。

### (3) 数据通信流程

从机部分的数据通信过程受主机控制，其基本流程如下：

- (1) 初始化完成后，从机将进入等待状态，监视串口数据。
- (2) 如果串口接收到的数据为\_RDY\_信号，则从机结束等待状态，进入下一步流程，否则丢弃当前接收字节，继续等待。

(3) 程序在接收到\_RDY\_信号后，将通过读取 P0 口的值判断本机当前状态，如果 P0 口的值为 0xab，则表明设备忙，程序返回\_BUSY\_信号，并重新进入程序开始部分的等待状态。如果 P0 口为其他值，则发送\_OK\_信号，表示从机可以接收数据。

(4) 程序调用 recv\_data() 函数接收主机发送的数据部分并作出应答，接收到的数据保存至 buf 指向的缓冲区中。

(5) 如果 recv\_data() 函数返回 0xff，表示数据校验失败，程序等待主机重新发送数据。否则使用 Beep\_ok() 函数蜂鸣表示数据接收成功，随后，程序重新进入开始的等待状态，开始下一个数据通信流程。

该部分程序对应的流程图如图 8-8 所示。

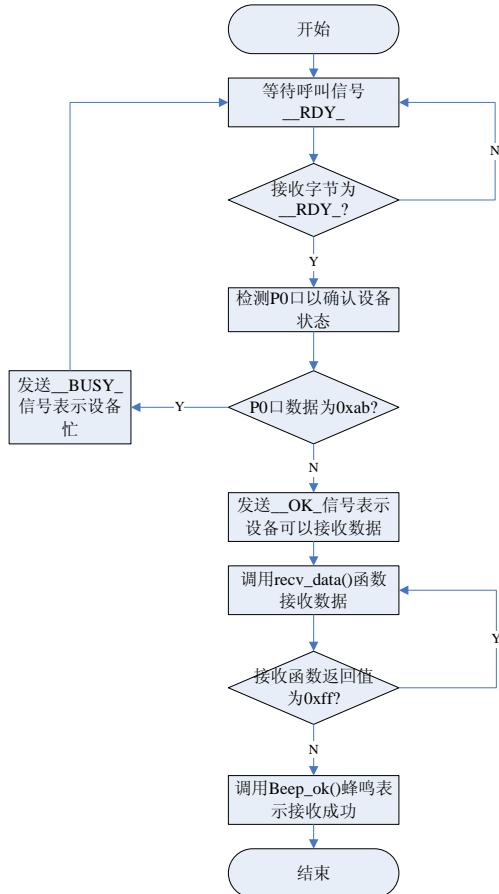


图 8-8 从机部分数据通信流程

具体的实现代码如下：

```

/* 进入设备应答阶段 */
while(1)
{
 /* 如果接收到的数据不是握手信号_RDY_，则继续等待 */
 while(tmp != _RDY_)
 {
 RI = 0;
 while(!RI);
 tmp = SBUF;
 RI = 0;
 }

 /* 程序通过检测 P0 口数据判断当前设备状态，若 P0=0xab，表示当前设备忙 */
 P0 = 0xff;
 tmp = P0;
 if(tmp == 0xab) // 如果 P0 口数据为 0x, 则当前设备忙，发送_BUSY_信号

```

```

{
 TI = 0;
 SBUF = _BUSY_;
 while(!TI);
 TI = 0;
 continue;
}

TI = 0; // 否则发送_OK_信号表示可以接收数据
SBUF = _OK_;
while(!TI);
TI = 0;
/* 数据接收 */
tmp = 0xff;
while(tmp == 0xff)
{
 tmp = recv_data(buf); // 校验失败返回 0xff, 接收成功则返回 0
}
Beep_ok(); // 蜂鸣表示数据接收成功
}

```

#### (4) 接收数据部分

具体的数据接收过程是使用函数 `recv_data()` 实现的。这里检验数据是否正确的方法是：程序对接收到的每一个字节计算校验和，如果得到的校验字节为 0，表示校验正确，函数返回 0，否则表示校验错误，函数将返回 0xff。这个函数的实现步骤如下：

- (1) 串口接收的第一个字节为数据长度，程序将该字节内容保存在 `len` 变量中，并为保存校验字节的变量设置初值。
- (2) 随后程序启动循环，顺序接收 `len` 个字节数据，并将其保存至缓冲区中。在接收数据的同时，还需要进行校验字节的计算。
- (3) 数据接收完毕后，程序将缓冲区的最后一个字节置为“\0”表示数据结束。
- (4) 随后程序接收最后一个字节的校验字节，并与当前校验字节进行异或运算。如果结果为 0，表示校验成功，从机向主机发送\_SUCC\_信号，函数返回 0，否则表示校验失败，从机向主机发送\_ERR\_信号，随后将缓冲区清空并返回 0xff。

上述的具体流程如图 8-9 所示。

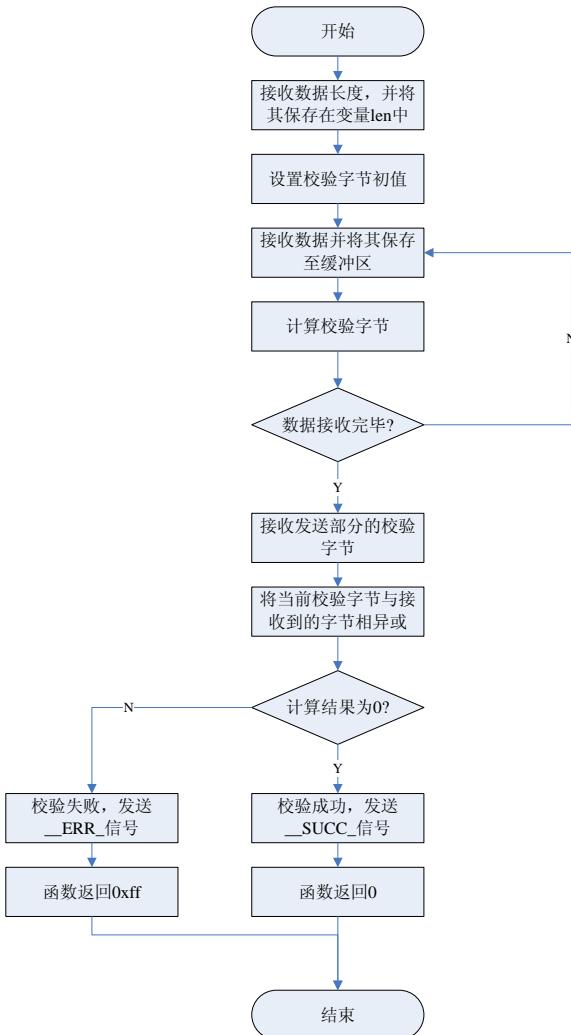


图 8-9 recv\_data() 函数流程

recv\_data() 函数的代码如下：

```

/* 接收数据，注意该函数使用 buf 指向的缓冲区保存数据，在数据末尾使用'\0'表示数据结束
 * 返回值为 0，数据校验成功，返回值为 0xff，数据校验失败
 */
unsigned char recv_data(unsigned char *buf)
{
 unsigned char len; // 该字节用于保存数据长度
 unsigned char ecc; // 该字节用于保存校验字节
 unsigned char i, tmp;
 /* 接收数据长度 */
 RI = 0;
 while(!RI);
 len = SBUF;

```

```

RI = 0;
/* 使用 len 的值为校验字节 ecc 赋初值 */
ecc = len;
/* 接收数据 */
for(i=0; i<len; i++)
{
 while(!RI);
 *buf = SBUF; // 接收数据
 ecc = ecc^(*buf); // 进行字节校验
 RI = 0;
 buf++;
}
*buf = 0; // 表示数据结束
/* 接收校验字节 */
while(!RI);
tmp = SBUF;
RI = 0;
/* 进行数据校验 */
ecc = tmp^ecc;
if(ecc != 0) // 校验失败
{
 *(buf-len) = 0; // 清空数据缓冲区
 TI = 0; // 发送校验失败信号
 SBUF = __ERR_;
 while(!TI);
 TI = 0;
 return 0xff; // 返回 0xff 表示校验错误
}
TI = 0; // 校验成功
SBUF = __SUCC_;
while(!TI);
TI = 0;
return 0; // 校验成功, 返回 0
}

```

### 8.3 单片机多机通信

在实际应用系统中，经常会遇到多个微处理器协调工作的情况，这就构成了一个分布式的多机系统。8051 系列单片机的接口在设计上就考虑到了这种多机通信的情况，用户可以直

接利用其硬件特点方便地构建多机网络。一个多机网络的结构可以如图 8-10 所示。

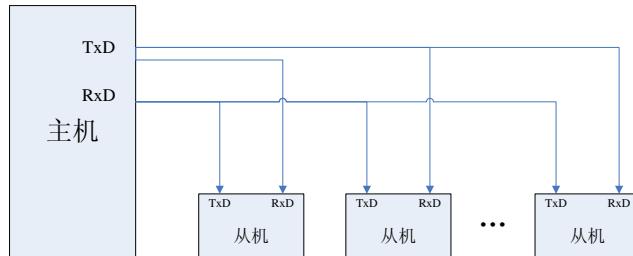


图 8-10 多机全双工通信连接图

图 8-10 中，系统使用一台主机和多台从机的连接方式，主机的 RxD 端与所有从机的 TxD 端相连，主机的 TxD 端与所有从机的 RxD 端相连，主机发送的信号可被各从机接收，而各从机发送的信息则只能由主机接收。如果各个从机之间需要进行相互通信，则只能通过主机转发来进行。

多机通信中，主机与各个从机进行通信，必须能对各个从机进行识别，这一识别功能可以由通信双方的串口部分硬件完成，也可以使用软件实现。8051 单片机的串口专门为这种多机通信提供了识别功能，该功能是利用串口控制寄存器 SCON 的 SM2 位实现的。当串口以方式 2 或方式 3 工作时，发送和接收的每一帧信息都是 11 位，其中第 9 位数据位是可编程的，通过对 SCON 寄存器的 TB8 位置 1 或置 0，以区别发送的是地址帧还是数据帧（规定地址帧的第 9 位为 1，数据帧的第 9 位为 0）。若从机的控制位 SM2 被设为 1，则当接收的是地址帧时，数据装入 SBUF，并置 RI=1，向 CPU 发出中断请求，若接收的是数据帧，则不产生中断，信息被抛弃。若 SM2 被设为 0，则无论是地址帧还是数据帧都将产生 RI=1 中断标志，数据装入 SBUF。利用这一功能，可以按照如下步骤进行数据通信：

- (1) 将所有 SM2 位置 1，使其处于只接收地址帧的状态。
- (2) 主机发送一帧地址信息，其中前 8 位数据位表示通信的从机地址，第 9 位为 1，表示当前帧为地址帧。
- (3) 从机接收到地址帧后，将本机地址与帧中地址进行比较。如果地址相同，将其 SM2 位置 0，准备接收数据。如果地址不同，则丢弃当前数据，SM2 位不变。
- (4) 主机发送数据帧，相应的从机接收，其他从机则不受影响。
- (5) 当主机需要与其他从机通信时，可以再次发出地址帧寻呼从机，重复这一过程。

下面给出一个具体的多机通信实例，这个实例中应用了 SCON 寄存器中的 SM2 位判断地址帧和数据帧，与前面介绍的通信过程相比，为了提高通信的可靠性，本例中从机要对主机发送的地址帧和数据作出应答，以保证通信链路的建立。主机在发送数据时，仍将按照前面点对点通信的示例中的数据格式进行传输，如下：

| 字节数 n | 数据 1 | 数据 2 | ..... | 数据 n | 字节奇偶校验 |
|-------|------|------|-------|------|--------|
|-------|------|------|-------|------|--------|

在程序中，第 9 位发送数据位为 SCON 中的 TB8 位，第 9 位接收数据位为 SCON 的 RB8 位，因此，发送数据前，可以通过对 TB8 位置 1 或 0 来确定要发送的是地址帧还是数据帧。而接收数据时，对地址帧的判断则是通过读取 RB8 位来获得的，RB8=1，当前帧为地址帧，RB8=0，当前帧为数据帧。

下面分别介绍主机部分和从机部分的程序设计方法。

### 8.3.1 主机部分通信程序设计

多机系统中的主机通信程序也是分为 4 个部分，分别为预定义及全局变量部分、程序初始化部分、数据通信流程和发送数据部分。

#### 1. 预定义及全局变量部分

该部分程序中也对通信中用到的握手信号进行了定义。本例的通信过程中，由于地址帧的存在，握手过程较点对点通信简单一些，只有从机返回给主机通知接收数据是否校验成功的两个信号。

#### 2. 程序初始化部分

程序初始化部分中，除了要对数据发送缓冲区赋初值外，还需要获得通信的从机地址，该地址是在从 P0 口读取完数据结束标志后的下一个字节中获得的。程序中定义了 addr 变量保存该信息。初始化程序中，还要完成串口的初始化，由于程序中使用查询方式收发串口数据，因此需要关闭所有中断信号，初始化程序的代码如下：

```
char buf[__MAX_LEN__];
unsigned char i = 0;
unsigned char tmp;
unsigned char addr; // 该字节用于保存要通信的从机地址
/* 为缓冲区赋初值 */
P0 = 0xff;
while(P1 != 0) // 每隔 100ms 从 P0 口读取，若读取到 0 则表明数据采集结束
{
 *(buf+i) = P0;
 delay10ms(10); // 延时 100ms
 P0 = 0xff;
 i++;
}
*(buf+i) = 0; // 缓冲区最后一个字节为 0 表示数据结束
/* 读要访问的分机地址 */
P0 = 0xff;
addr = P0;
/* 串口初始化 */
init_serial(); // 初始化串口
EA = 0; // 关闭所有中断
```

与前面点对点通信实例的程序不同，该例中串口被初始化为工作方式 3，以便使用数据通信中的第 9 位数据位，串口初始化程序是调用 init\_serial() 完成的，其代码如下：

```
/* 初始化串口 */
void init_serial()
{
```

```

TMOD = 0x20; //定时器 T1 使用工作方式 2
TH1 = 250; // 设置初值
TL1 = 250;
TR1 = 1; // 开始计时
PCON = 0x80; // SMOD = 1
SCON = 0xd0; //工作方式 3, 9 位数据位, 波特率 9600bit/s, 允许接收
}

```

### 3. 数据通信流程

主机部分的数据通信的基本流程如下：

- (1) 主机首先向所有从机发送地址帧对要通信的主机进行呼叫，发送地址帧时需将 TB8 位置 1。
- (2) 发送地址帧后，主机接收应答，如果应答信号中的地址与前面发送的地址并不相同，主机将重新发送地址帧呼叫，否则调用 send\_data() 函数发送数据。
- (3) 发送完数据后，主机等待从机的校验信号，如果接收到 \_SUCC\_ 信号，表示发送成功，通信结束，否则主机将重新发送数据，直至发送成功。

该部分程序对应的流程图如图 8-11 所示。

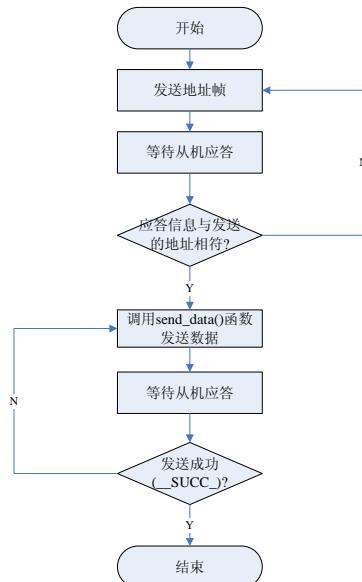


图 8-11 主机部分数据通信流程

具体的实现代码如下：

```

/* 发送地址帧并接收应答信息，如果接收的信号与发送的地址信息不同，则重新发送地址帧 */
tmp = addr-1;
while(tmp != addr)
{
 /* 发送从机地址 */
 TI = 0;

```

```

 TB8 = 1; // 发送地址帧
 SBUF = addr;
 while(!TI);
 TI = 0;
 /* 接收从机应答 */
 RI = 0;
 while(!RI);
 tmp = SBUF;
 RI = 0;
 }
/* 发送数据并接收校验信息，如果接收的信号为 0FH，表示从机接收成功，否则将重新发送该组
数据 */
tmp = __ERR__;
while(tmp != __SUCC__)
{
 send_data(buf); // 发送数据
 RI = 0;
 while(!RI);
 tmp = SBUF;
 RI = 0;
}
while(1); // 程序结束，进入死循环
}

```

#### 4. 发送数据部分

该程序中，具体的数据发送过程仍是被包含在 `send_data()` 函数中，该函数在进行数据发送时，为了与地址帧区别，需要将数据的第 9 位 `TB8` 位置 0，其他的流程与点对点通信中主机程序的相应部分是相同的，具体的代码如下：

```

/* 发送数据 */
void send_data(unsigned char *buf)
{
 unsigned char len; // 保存数据长度
 unsigned char ecc; // 保存校验字节
 len = strlen(buf); // 计算要发送数据的长度
 ecc = len; // 开始进行校验字节计算
 /* 发送数据长度 */
 TI = 0;
 TB8 = 0; // 发送数据帧
 SBUF = len; // 发送长度
 while(!TI);
}

```

```

TI = 0;
/* 发送数据 */
for(i=0; i<len; i++)
{
 ecc = ecc ^ (*buf); // 计算校验字节
 TB8 = 0; // 发送数据帧
 SBUF = *buf; // 发送数据
 buf++;
 while(!TI);
 TI = 0;
}
/* 发送校验字节 */
TB8 = 0; // 发送数据帧
SBUF = ecc; // 发送校验字节
while(!TI);
TI = 0;
}

```

### 8.3.2 从机部分通信程序设计

本例的从机通信程序也被分为预定义及全局变量部分、程序初始化部分、数据通信流程和接收数据部分共4个部分。

#### 1. 预定义及全局变量部分

预定义部分给出了握手信号的定义和程序中调用的子函数的声明，这部分的内容与主机程序相应部分基本相同。

#### 2. 程序初始化部分

程序初始化部分包括对本机地址的设置和串口部分的初始化。本机地址是通过读取P1口的数据获得的，其内容保存在addr变量中。串口初始化程序在init\_serial()中，单片机串口的设置与主机是完全相同的。这里仅给出程序的初始化代码，init\_serial()函数的实现与主机通信程序相同。

```

char buf[_MAX_LEN_];
unsigned char i = 0;
unsigned char tmp = 0xff;
unsigned char addr; // 保存本机地址
/* 从P1口读取本机地址 */
P1 = 0xff;
addr = P1;

```

```
/* 串口初始化 */
init_serial(); // 初始化串口
EA = 0; // 关闭所有中断
```

### 3. 数据通信流程

从机部分的数据通信过程受主机控制，该部分程序对应的流程图如图 8-12 所示。其基本流程如下：

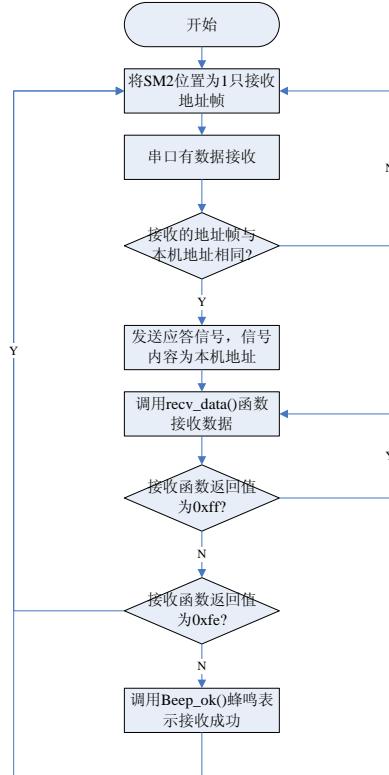


图 8-12 从机部分数据通信流程

(1) 初始化完成后，从机设置 SM2 位为 1，串口只接收第 9 位数据位为 1 的地址帧，数据帧将被直接抛弃。

(2) 如果串口有数据接收（收到地址帧），则从机会将该帧中的地址信息与本机地址比较，如果相同，则发送应答信号，应答信号内容应为本机地址，否则丢弃当前数据，从机继续处于等待呼叫状态。

(3) 如果接收的地址帧与本机地址相符，从机发送应答信号后，还要将 SM2 位置 0，准备接收数据信息。

(4) 程序调用 `recv_data()` 函数接收主机发送的数据部分并作出应答，接收到的数据保存至 `buf` 指向的缓冲区中。

(5) 如果 `recv_data()` 函数返回 `0xff`，表示数据校验失败，程序等待主机重新发送数据。如果函数返回值为 `0xfe`，表示从机在数据接收过程中发现主机发送地址帧，程序将放弃当前

接收过程，将 SM2 位重新置 1，开始下一通信过程。如果函数返回 0，表示数据被成功接收，程序将调用 Beep\_ok() 函数蜂鸣，随后，程序将 SM2 位置 1，重新开始下一个数据通信流程。

具体的实现代码如下：

```
/* 进入设备应答阶段 */
while(1)
{
 SM2 = 1; // 只接收地址帧
 /* 如果接收到的地址帧不是本机地址，则继续等待 */
 tmp = addr-1;
 while(tmp != addr)
 {
 RI = 0;
 while(!RI);
 tmp = SBUF;
 RI = 0;
 }
 /* 发送应答信号，并做好接收数据的准备 */
 TI = 0;
 TB8 = 0; // 主机不检测该位
 SBUF = addr;
 while(!TI);
 TI = 0;
 SM2 = 0; // 允许接收数据信息
 /* 数据接收 */
 tmp = 0xff;
 while(tmp == 0xff) // 如果数据校验失败则重新接收数据
 {
 tmp = recv_data(buf); // 校验失败返回 0xff，检测到地址帧则返回 0xfe，接收
 成功则返回 0
 }
 if(tmp == 0xfe) // 在数据接收过程中，如果发现地址帧，则重新开始整个接收过程
 continue;
 Beep_ok(); // 蜂鸣表示数据接收成功
}
```

#### 4. 接收数据部分

数据的接收是使用函数 `recv_data()` 实现的。该函数在进行数据接收时，还需要检测每个帧的第 9 位数据位。

- 串口接收的第一个字节为数据长度，程序将该字节内容保存在 `len` 变量中，并为保存校验字节的变量设置初值。这里如果 RB8 位为 1，则函数直接返回 `0xfe`。

- 随后程序启动循环，顺序接收 len 个字节数据，并将其保存至缓冲区中。在接收数据的同时，还需要进行校验字节的计算。每个接收到的数据均要检验其第 9 位数据位，如果为 1 则表明接收到地址帧，函数将返回 0xfe。
- 数据接收完毕后，程序将缓冲区的最后一个字节置为“\0”表示数据结束。
- 随后程序接收最后一个字节的校验字节，并与当前校验字节进行异或运算。如果结果为 0，表示校验成功，从机向主机发送\_SUCC\_信号，函数返回 0，否则表示校验失败，从机向主机发送\_ERR\_信号，随后将缓冲区清空并返回 0xff。

函数代码如下：

```
/* 接收数据，注意该函数使用 buf 指向的缓冲区保存数据，在数据末尾使用'\0'表示数据结束
 * 返回值为 0，数据校验成功，返回值为 0xfe，接受过程中接收到地址帧，返回值为 0xff，数据校验
失败
*/
unsigned char recv_data(unsigned char *buf)
{
 unsigned char len; // 该字节用于保存数据长度
 unsigned char ecc; // 该字节用于保存校验字节
 unsigned char i, tmp;
 /* 接收数据长度 */
 RI = 0;
 while(!RI);
 if(RB8 == 1) // 若当前接收为地址帧则返回 0xfe
 return 0xfe;
 len = SBUF;
 RI = 0;
 /* 使用 len 的值为校验字节 ecc 赋初值 */
 ecc = len;
 /* 接收数据 */
 for(i=0; i<len; i++)
 {
 while(!RI);
 if(RB8 == 1) // 若当前接收为地址帧则返回 0xfe
 return 0xfe;
 *buf = SBUF; // 接收数据
 ecc = ecc ^ (*buf); // 进行字节校验
 RI = 0;
 buf++;
 }
 *buf = 0; // 表示数据结束
 /* 接收校验字节 */
 while(!RI);
```

```

if(RB8 == 1) // 若当前接收为地址帧则返回 0xfe
 return 0xfe;
tmp = SBUF;
RI = 0;
/* 进行数据校验 */
ecc = tmp ^ ecc;
if(ecc != 0) // 校验失败
{
 *(buf+len) = 0; // 清空数据缓冲区
 TI = 0; // 发送校验失败信号
 TB8 = 0;
 SBUF = __ERR__;
 while(!TI);
 TI = 0;
 return 0xff; // 返回 0xff 表示校验错误
}
TI = 0; // 校验成功
TB8 = 0;
SBUF = __SUCC__;
while(!TI);
TI = 0;
return 0; // 校验成功, 返回 0
}

```

## 8.4 单片机 I<sup>2</sup>C 总线通信

对于较为复杂的单片机应用系统，元件与芯片之间短距离通信的物理线路往往比较多，这样不仅增加了硬件系统设计的难度，而且也不利于系统的稳定性，成为了系统设计中一个瓶颈。针对这一问题，许多公司都提出不同的解决方案，这里 Philips 公司提出的 I<sup>2</sup>C 总线协议有效地解决了这一问题，并被广大的用户所接受。I<sup>2</sup>C 总线协议使用一种简单的双向两线总线来连接系统中的各个 IC 器件，IC 器件利用这两条线路进行串行通信。为了保证通信的正常进行，每一个连接 I<sup>2</sup>C 总线的器件都需要在器件上附加一个符合 I<sup>2</sup>C 总线协议的片上接口。本章中，将首先对 I<sup>2</sup>C 总线协议作一个简单的介绍，随后通过两个具体的实例详细讲述两种不同的 I<sup>2</sup>C 总线系统的设计方法。

### 8.4.1 I<sup>2</sup>C 总线介绍

I<sup>2</sup>C 总线主要应用于板级的 IC 通信需要，即主要被用作硬件系统中电路板上各个 IC 芯片的相互通信的线路，I<sup>2</sup>C 总线在系统设计中十分常见。这里给出一个路桥测重系统中应用

I<sup>2</sup>C 总线进行 IC 互连的设计实例，它的基本结构可以分为以下几个部分：

- 主器件部分

在 I<sup>2</sup>C 总线系统中，必须由其中的一个设备对总线时序进行控制，并提供时钟信号，该设备被称为为主器件。主器件负责总线上的各个设备信息的传输控制，检测并协调数据的发送和接收。I<sup>2</sup>C 总线系统中，主器件对整个数据传输具有绝对的控制权，其他设备只是对主器件发送的命令等控制信息作出响应。通常一个 I<sup>2</sup>C 总线系统中可以有多个主器件，当它们同时发送控制请求时，就需要进行总线仲裁，关于 I<sup>2</sup>C 总线仲裁的知识，读者可以参考 11.2 中的介绍。主器件通常由 MCU 担任，本例中系统中的主器件为 Philips 公司的 P89C662 增强型 51 兼容单片机，该芯片集成了 32kB 的 Flash 程序存储空间和 1kB 的内部数据存储区，可以很好的满足系统设计的需要，而无需任何外扩存储器，使用 6 时钟的机器周期，速度是普通 8051 单片机的一倍，并内置了 I<sup>2</sup>C 硬件接口，是一款优秀的增强型 8051 单片机，非常适合应用于本例的系统中。

- 从器件部分

当 I<sup>2</sup>C 总线中的主器件工作时，其它总线设备均作为从器件存在。总线上的每一个从器件均有一个自身的地址，主器件通过设备地址访问相应的设备，对应的从器件则作出响应，与主器件进行通信。从器件之间无法通信，任何数据传输都必须通过主器件进行。本例中包含了大量的从器件设备，分别为 LED 显示模块、kB 式智能两线键盘、串行 E<sup>2</sup>PROM 存储器和微型针式打印机机芯。

- I<sup>2</sup>C 总线接口

硬件系统中，要使各个设备之间进行 I<sup>2</sup>C 总线连接，各个设备必须使用 I<sup>2</sup>C 总线接口。I<sup>2</sup>C 总线是两线总线，其两条信号分别为 SCL 时钟信号和 SDA 数据信号。作为系统中的从器件，每个设备都必须具有能正确处理 I<sup>2</sup>C 总线时序的硬件 I<sup>2</sup>C 总线接口。而对主器件而言，由于其本身为 MCU，具有控制的功能，因此既可以使用集成了硬件 I<sup>2</sup>C 总线接口的单片机，直接应用其硬件接口，也可以以软件的形式使用两条 I/O 口线模拟 I<sup>2</sup>C 总线的主器件逻辑，所以，对系统中的主器件而言，其可以使用硬件 I<sup>2</sup>C 总线接口，也可以使用软件模拟 I<sup>2</sup>C 总线接口。本系统中，由于选用的 MCU 自带 I<sup>2</sup>C 总线接口，因此直接采用硬件的方式对总线进行控制。

该系统使用压力传感器检测路桥等的承重信息，并实时在 LED 上显示，可以自由对超重标准进行设定，其内置存储器可以保存一整天的数据信息，并能以规定的报表进行整合和打印。由于系统中需要连接大量的设备，而单片机的 I/O 端口资源有限，因此尽量选用带有 I<sup>2</sup>C 总线接口的设备，这样不仅有效的减小了单片机 I/O 端口的使用，还有利于系统的模块化设计，在进行故障判断和功能检测中都有着不可比拟的优势。在本系统中，I<sup>2</sup>C 总线接口通信部分的设计是十分重要的，本章将对这一部分进行重点的介绍，此外，为了增加本章知识的适用性，这里同时给出了硬件 I<sup>2</sup>C 总线接口和软件模拟 I<sup>2</sup>C 总线接口两套通用函数的设计方法，并提供了实现源码。

在对具体的 I<sup>2</sup>C 总线接口设计进行学习之前，必须了解 I<sup>2</sup>C 总线的时序、硬件接口逻辑等知识，下面首先介绍这一部分的内容。

目前比较流行的串行扩展总线中，I<sup>2</sup>C 总线以其严格的规范和众多支持 I<sup>2</sup>C 接口的外围器件而获得广泛应用。

I<sup>2</sup>C 总线主要有以下几个特征：

- 只要求两条总线线路——一条串行数据线（SDA）；一条串行时钟线（SCL）；
- 总线模式包括主发送模式、主接收模式、从发送模式、从接收模式；
- 每个连接到总线上的器件都有惟一的一个地址，通过这个地址，主机可以对从机进行寻址；
- 存在冲突检测和仲裁机制以保证数据传输的完整性和稳定性；
- 标准模式下数据传输速率可以达到 100kbit/s，快速模式下可以达到 400kbit/s，高速模式下可以达到 3.4Mbit/s；
- 由于采用漏极开路工艺，所以总线上要接上拉电阻；
- 连接到相同总线上的 IC 数量只受到最大电容 400pF 的限制；
- 片上滤波器可以滤去总线上的毛刺，保证数据传输的稳定性和完整性。

### 1. I<sup>2</sup>C 总线的数据传输接口特性

连入 I<sup>2</sup>C 总线的器件可以分为主器件（发送器）和从器件（接收器）两种。其中主器件是指向 I<sup>2</sup>C 总线上发送信息的器件，而从器件是指从 I<sup>2</sup>C 总线上接收信息的器件。主器件负责启动总线上的数据传输并产生总线上的时钟信号，在这种情况下总线上的任何被寻址的器件都将作为从器件存在。I<sup>2</sup>C 总线的控制完全是由挂接在总线上的主器件决定的，主器件通过发送的地址和数据控制从器件的行为，从器件响应主器件的请求做出相应的回应。

I<sup>2</sup>C 总线上的两条线路分别为 SDA 和 SCL，均为双向 I/O 口，因为采用漏机开路工艺，所以必须通过上拉电阻接到正电源上，使之具有线“与”功能。

I<sup>2</sup>C 总线是一个多主机总线。这就是说可以连接多于一个能控制总线的器件。例如图 8-13 中显示的是一个带有两个微控制器的 I<sup>2</sup>C 总线系统。在这一系统中，两个微处理器作为主器件，控制总线上作为从器件存在的开关矩阵、LCD 驱动芯片、实时时钟（如 PCF8563）和 SRAM（或 E<sup>2</sup>PROM）存储器等设备。

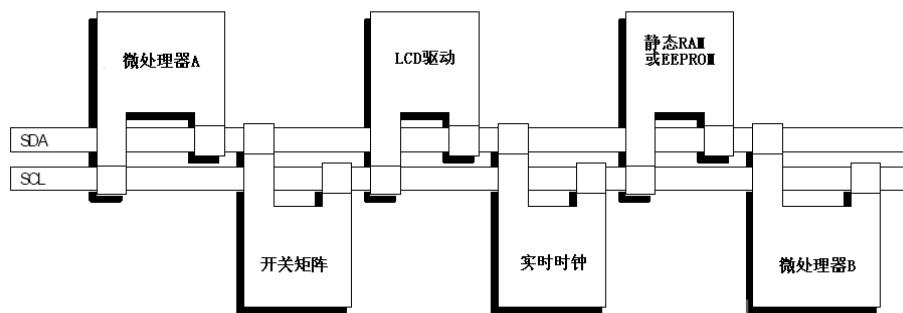


图 8-13 包含两个处理器的 I<sup>2</sup>C 总线系统

### 2. I<sup>2</sup>C 总线的通信时序

使用 I<sup>2</sup>C 总线进行数据通信，必须先要熟悉 I<sup>2</sup>C 总线的通信时序。本节介绍 I<sup>2</sup>C 总线的通信时序部分，分别为起始条件和停止条件、位传输、数据响应、同步仲裁以及 7 位地址格式 5 个部分进行介绍。

### (1) 起始条件和停止条件

在 I<sup>2</sup>C 总线技术规范中，起始条件 (S) 和停止条件 (P) 一般是由主器件产生的。起始条件表明一个 I<sup>2</sup>C 总线传送的开始，停止条件则表明 I<sup>2</sup>C 总线通信结束。SCL 为高电平时，SDA 由高电平到低电平的跳变被定义为起始条件，而 SDA 由低电平到高电平的跳变为停止条件。I<sup>2</sup>C 总线在起始条件以后被认为处于忙状态，在停止条件以后，如果没有起始条件产生，这段时间总线可以被认为是处于空闲状态。

I<sup>2</sup>C 总线起始条件和停止条件的时序如图 8-14 所示。

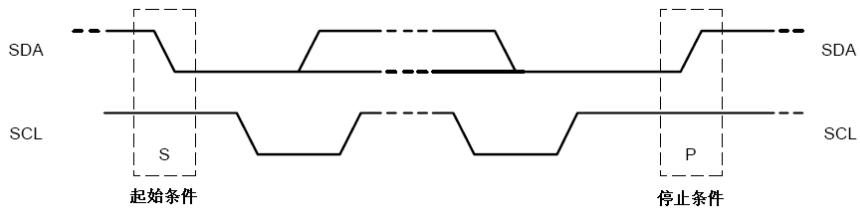


图 8-14 起始条件和停止条件

在一个通信过程中，应该有一个起始条件和一个停止条件，如果在二者之间有起始条件产生，该条件被称为重复起始条件 (Sr)。如果主器件产生重复起始条件而不产生停止条件，总线会一直处于忙状态。此时的起始条件和重复起始条件在功能上是一样的。

如果连接到 I<sup>2</sup>C 总线上的器件集成了必要的接口硬件，那么检测起始条件和停止条件的过程将由硬件自动完成。但是，如果微处理器没有集成 I<sup>2</sup>C 总线的硬件接口（比如后面要讲到的采用软件模拟 I<sup>2</sup>C 总线协议），那么程序设计中每个时钟周期就至少要对 SDA 采样两次，以此判别是否发生电平跳变。

### (2) I<sup>2</sup>C 总线的位传输

I<sup>2</sup>C 总线协议的技术规范中规定每次发送到 I<sup>2</sup>C 总线 SDA 上的数据必须是一个字节，但每次传输可以发送的字节数量是不受限制的。传输的数据字节按照由高位到低位的顺序发送，每发送一个字节后必须跟一个响应位。如果从器件在接收下一字节之前需要时间对当前数据进行处理，那么在从器件完成当前数据的接收后，将保持 SCL 为低电平，通知主器件进入等待状态，直到从器件准备好接收下一字节数据时，释放时钟线 SCL，主器件才可以继续发送数据。

I<sup>2</sup>C 总线的数据传输时序如图 8-15 所示。

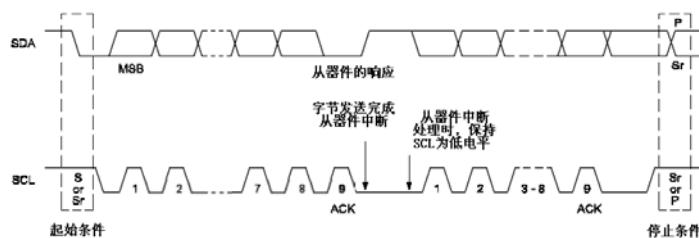


图 8-15 I<sup>2</sup>C 总线的数据传输

### (3) I<sup>2</sup>C 总线上的数据响应

为了保证 I<sup>2</sup>C 总线上数据传输的稳定性和完整性, I<sup>2</sup>C 总线技术规范中规定每一个字节数据传输完成后, 都需要接收方做出响应, 确认数据是否发送成功, 以保证数据的完整性。相关的响应时钟脉冲由主器件产生。

在响应脉冲期间, 从器件将 SDA 拉低, 并使得 SDA 在这个时钟脉冲的高电平期间保持稳定的低电平。从器件响应信号结束后, SDA 返回高电平, 进入下一个传送周期。

如果从器件不能及时做出响应(比如它正在执行一些实时函数不能接收或发送), 则响应时钟脉冲周期器件, 从器件始终保持 SDA 为高电平。

I<sup>2</sup>C 总线的数据响应时序如图 8-16 所示。

### (4) I<sup>2</sup>C 总线的同步和仲裁

所有的主器件均使用 I<sup>2</sup>C 总线的 SCL 发送自己的时钟来传输字节数据。数据只在 SCL 为高电平的时候有效, 所以当有多个主器件向 I<sup>2</sup>C 总线上上传输数据时, 就需要一个确定的时钟来进行仲裁, 这就产生了一个时钟同步的问题。

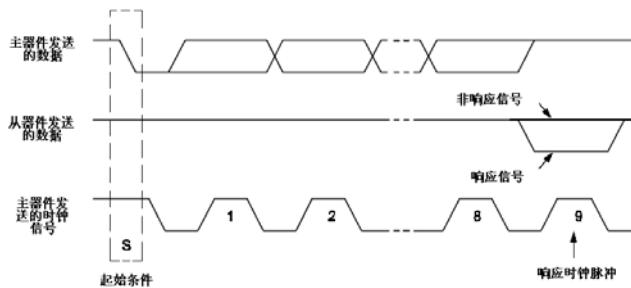


图 8-16 I<sup>2</sup>C 总线的响应时序

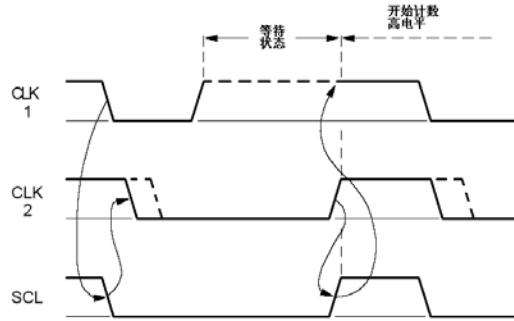
时钟同步是通过线与连接在 I<sup>2</sup>C 总线上的 SCL 来执行的。当主器件 1 的时钟由高电平跳变为低电平时, 主器件 2 的时钟仍保持为高电平, 同时 I<sup>2</sup>C 总线上的 SCL 线由高电平变为低电平, 而当主器件 1 的时钟跳变为高电平时, 主器件 2 的时钟仍保持低电平, 这时, I<sup>2</sup>C 总线的 SCL 继续保持低电平。这就是说, SCL 线被有最长低电平周期的器件保持低电平, 低电平周期短的器件会进入高电平的等待状态。

当所有挂在 I<sup>2</sup>C 总线上的器件的时钟都由低电平切换为高电平后, 时钟线 SCL 被释放并变成高电平。之后, 所有器件的时钟和 SCL 线的状态没有差别。

这样, 实际上产生的同步 SCL 时钟的低电平周期是由低电平时钟周期最长的器件决定的, 而高电平周期是由高电平时钟周期最短的器件决定的。

I<sup>2</sup>C 总线的时钟同步如图 8-17 所示。

I<sup>2</sup>C 总线的竞争仲裁是以时钟同步为基础的。总线竞争是指 I<sup>2</sup>C 总线上的两个或者多个主器件同时想要占用总线。例如, 在分布式单片机系统中, 很有可能出现在某一时刻, 两个或者多个单片机同时向总线发送数据来控制同一个外设。这时 I<sup>2</sup>C 总线的仲裁原则是: 在总线竞争的情况下, 系统首先在 SCL 上产生同步时钟, 然后当 SCL 为高电平时, 在 SDA 线上对发生竞争的主器件进行仲裁, 只有发送的数据与 SDA 上的电平相同的主器件才可以保留其输出级, 而发送的数据与 SDA 上的电平不同的主器件将自动关闭其输出级。

图 8-17 I<sup>2</sup>C 总线的时钟同步

I<sup>2</sup>C 总线的竞争仲裁是可以持续多位的，即首先比较地址位，如果每个主机都尝试寻址相同的器件，仲裁会继续比较数据位，或比较相应位，直到只有一个主器件获得总线控制权为止。

图 8-18 所示为两个主机的仲裁过程。

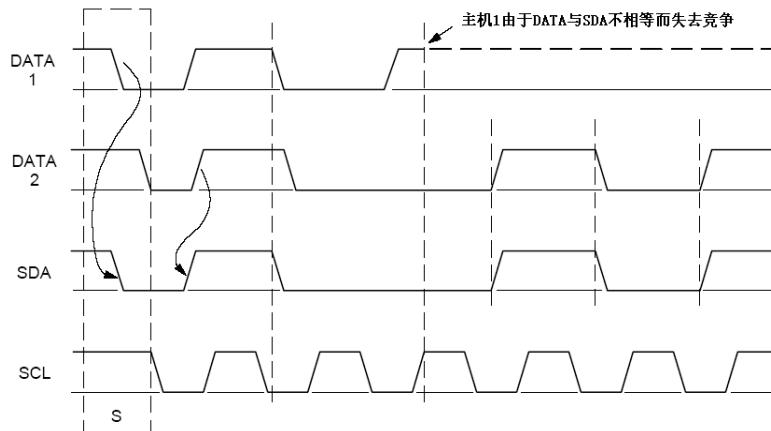


图 8-18 两个主机的仲裁过程

### (5) I<sup>2</sup>C 总线的 7 位地址格式

在进行数据传输之前，I<sup>2</sup>C 总线会首先发送一个字节进行寻址。这个字节的内容是紧跟在起始条件之后发送的，表示要进行通信的从器件的地址。该字节的定义如下：

| D7            | D6 | D5 | D4 | D3 | D2 | D1 | D0 |     |
|---------------|----|----|----|----|----|----|----|-----|
| SLAVE ADDRESS |    |    |    |    |    |    |    | R/W |

这里地址信息实际上是 7 个 bit，占用了地址字节的高 7 位，可以对 127 个器件进行寻址。字节的第 0 位用于表示数据的传输方向。当该位为高电平时，表示由从器件向主器件发送数据，即主器件对从器件进行读操作；当该位为低电平时，表示由主器件向从器件发送数据，即主器件对从器件进行写操作。

字节的第 7 位至第 1 位为从器件的地址信息。对于 I<sup>2</sup>C 总线上所挂的器件来说，每个器件的地址都是唯一的。一般来说，一个从器件的地址是由一部分固定地址和一部分可变地址

组成的，这是为了在一个系统中可以区分多个相同类型的器件。例如，PCF8573 有两位可编程的器件地址，而其他 5 位为固定地址，这样在一个 I<sup>2</sup>C 总线的系统中，最多只可能有 4 个 PCF8573，这是因为固定地址是不可变的。

一个完整的传输过程的时序如图 8-19 所示。

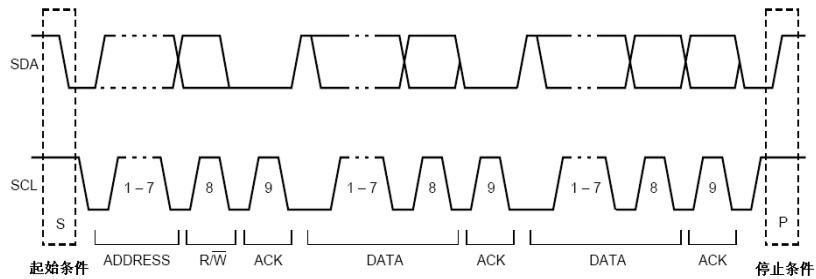


图 8-19 完整传输过程时序图

### 3. I<sup>2</sup>C 总线的技术规范

1992 年，Philips 公司发布的 I<sup>2</sup>C 总线规范的版本为 1.0，1998 年版本升级到 2.0，2000 年版本升级到 2.1。

#### (1) 版本 1.0

1992 年，I<sup>2</sup>C 总线规范的 V1.0 版本包括以下的修正：

- 删除了用软件编写从机地址的内容，因为实现这个功能相当复杂而且使用率很低；
- 删除了“低速模式”，实际上这个模式是整个 I<sup>2</sup>C 总线规范的子集，不需要再详细说明；
- 增加了快速模式，它将位速率增加 4 倍，达到 400kbit/s。快速模式器件都向下兼容，即它们可以在 0~100kbit/s 的 I<sup>2</sup>C 总线系统中使用；
- 增加了 10 位寻址，允许 1 024 个额外的从机地址；
- 快速模式器件的斜率控制和输入滤波改善了 EMC 性能。

#### (2) 版本 2.0

I<sup>2</sup>C 总线实际上已经成为一个国际标准，在超过 100 种不同的 IC 上实现而且得到超过 50 家公司的许可。但是现在的很多应用要求总线速度更高、电源电压更低。2.0 版本的 I<sup>2</sup>C 总线规范满足了这些要求，有了以下的修正：

- 增加了高速模式 (Hs 模式)，它将位速率增加到 3.4Mbit/s，Hs 模式的器件可以和 I<sup>2</sup>C 总线系统中快速和标准模式器件混合使用，位速率从 0~3.4Mbit/s；
- 电源电压是 2V 或更低的器件的低输出电平和滞后被调整到符合噪声容限的要求，而且保持和电源电压更高的器件兼容；
- 快速模式输出级的 0.6V/6mA 要求被删除；
- 新器件的固定输入电平被总线电压相关的电平代替；
- 增加了双向电平转换器的应用信息。

#### (3) 版本 2.1

I<sup>2</sup>C 总线规范的 V2.1 版有以下微小的修改：

- 在 Hs 模式下的重复起始条件后可以延长时钟信号 SCLH；
- Hs 模式中的一些时序参数变得更随意。

#### 4. I<sup>2</sup>C 总线的分类

标准模式的 I<sup>2</sup>C 总线规范在 20 世纪 80 年代初就已经出现了，当时它规定的最高传输速率可以达到 100kbit/s，地址编码为 7bit。随着科学技术的发展，I<sup>2</sup>C 总线的最初版本已经不能满足人们的需要了，经过不断地修改和完善，现在 I<sup>2</sup>C 总线规范的升级版本可以满足更高的数据传输速率与更多的地址空间的需求。例如，快速模式的位速率可以高达 400kbit/s，高速模式（Hs 模式）的位速率可以达到 3.4Mbit/s，以及 10 位寻址模式等。

##### (1) 快速模式（快速模式 F/S）

最初 I<sup>2</sup>C 总线规范中的最高传输速率只有 100kbit/s。这就限制了标准的 I<sup>2</sup>C 总线不能用于一些要求传输速率比较高的场合。为此，国际组织在 1992 年对规范进行了修订，并公布了 I<sup>2</sup>C 总线的版本 1.0，使它支持快速模式（F/S 模式），传输速率最高可以达到 400kbit/s，并保持了与标准模式的向下兼容性。

图 8-20 所示为标准模式与快速模式的 I<sup>2</sup>C 总线器件的连接图。

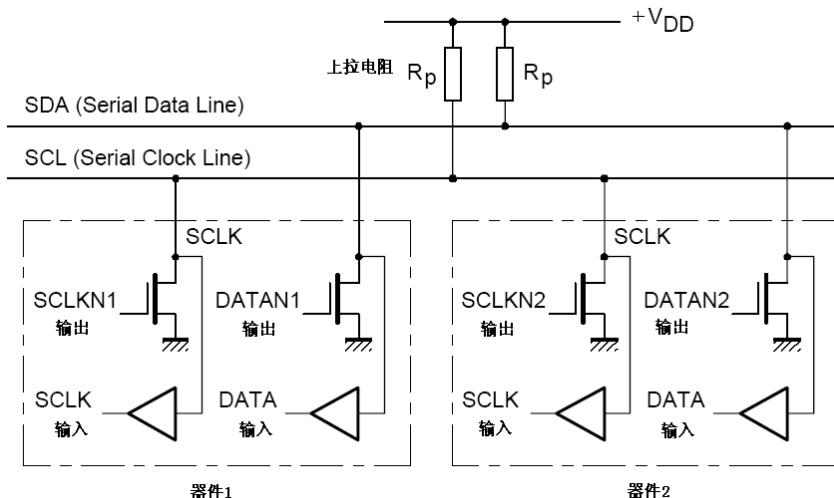


图 8-20 标准模式与快速模式的 I<sup>2</sup>C 总线器件连接图

快速模式 I<sup>2</sup>C 总线规范与标准模式相比有以下特点：

- 最大位速率增加到 400kbit/s；
- 调整了串行数据（SDA）和串行时钟（SCL）信号的时序；
- 快速模式器件的输入有抑制毛刺的功能，SDA 和 SCL 输入有 Schmitt 触发器；
- 快速模式器件的输出缓冲器对 SDA 和 SCL 信号的下降沿有斜率控制功能；
- 如果快速模式器件的电源电压被关断，SDA 和 SCL 的 I/O 引脚必须悬空，不能阻塞总线；
- 连接到总线的外部上拉器件必须调整以适应快速模式 I<sup>2</sup>C 总线更短的最大允许上升时间，对于负载最大是 200pF 的总线，每条总线的上拉器件可以是一个电阻，对于负载

在  $200\text{pF} \sim 400\text{pF}$  之间的总线，上拉器件可以是一个电流源最大值  $3\text{mA}$  或者是一个开关电阻电路。

## (2) 高速模式 (Hs 模式)

1998 年国际组织对 I<sup>2</sup>C 总线规范又进行了升级，其中增加的高速模式对 I<sup>2</sup>C 总线的传输速率是一个重大的突破，高速模式的器件最高数据传输位速率可达  $3.4\text{Mbit/s}$ ，并保持完全向下兼容快速模式或标准模式器件。高速模式传输除了不执行仲裁和时钟同步外，与快速模式或标准模式系统有相同的串行总线协议和数据格式。为了获得高达  $3.4\text{Mbit/s}$  的位速率，高速模式相对于标准的 I<sup>2</sup>C 总线规范作了如下改进：

- Hs 模式主机器件有一个 SDAH 信号的开漏输出缓冲器和一个在 SCLH 输出的开漏极下拉和电流源上拉电路。这个电流源电路缩短了 SCLH 信号的上升时间。任何时候在 Hs 模式，只有一个主机的电流源有效。
- 在多主机系统的 Hs 模式中，不执行仲裁和时钟同步，以加速位处理能力。仲裁过程一般在前面用 F/S 模式传输主机码后结束。
- Hs 模式主机器件以高电平和低电平 1:2 的比率产生一个串行时钟信号。解除了建立和保持时间的时序要求。
- 可以选择使用 Hs 模式器件内建的电桥。在 Hs 模式传输中，Hs 模式器件的高速数据 (SDAH) 和高速串行时钟 (SCLH) 线通过这个电桥与 F/S 模式器件的 SDA 和 SCL 线分隔开来。减轻了 SDAH 和 SCLH 线的电容负载，使上升和下降时间更快。
- Hs 模式从机器件与 F/S 从机器件的惟一差别是它们工作的速度。Hs 模式从机在 SCLH 和 SDAH 输出有开漏输出的缓冲器。SCLH 引脚可选的下拉晶体管可以用于拉长 SCLH 信号的低电平，但只允许在 Hs 模式传输的响应位后进行。
- Hs 模式器件的输出可以抑制毛刺，而且 SDAH 和 SCLH 输出有一个 Schmitt 触发器。
- Hs 模式器件的输出缓冲器对 SDAH 和 SCLH 信号的下降沿有斜率控制功能。

图 8-21 所示为纯高速 I<sup>2</sup>C 总线器件的连接图，图中标号指出的说明部分如下：

- (1) 这里不使用 SDA 和 SCL，它们可作其他用途。
- (2) 到输入滤波器。
- (3) 只有激活的主机能使用它的电流源上拉电路。
- (4) 虚线的晶体管是可选的开漏输出，可以延长串行时钟信号 SCLH。

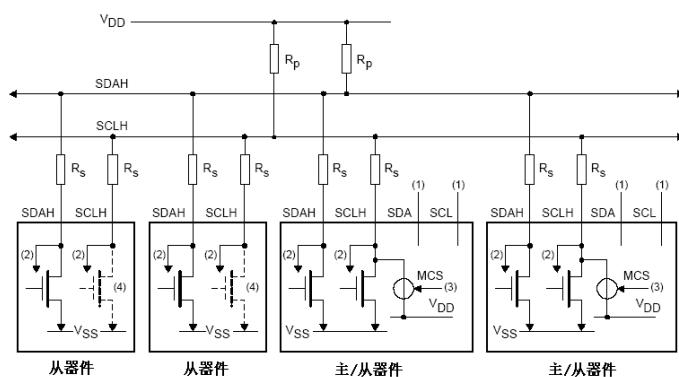


图 8-21 纯高速 I<sup>2</sup>C 总线器件的连接图

图 8-22 所示为高速 I<sup>2</sup>C 总线器件和快速 I<sup>2</sup>C 总线器件的连接图，图中标号指出的说明部分如下：

- ① 不使用电桥，SDA 和 SCL 可作其他用途。
- ② 到输入滤波器。
- ③ 电流源上拉电路保持禁能。
- ④ 虚线的晶体管是可选的开漏输出，可以延长串行时钟信号 SCLH。

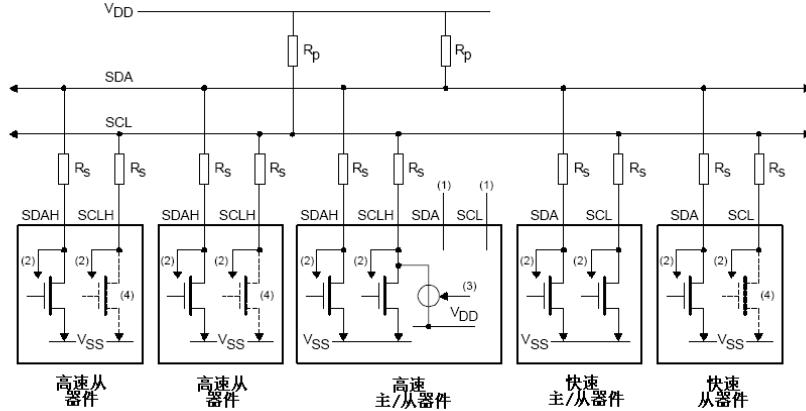


图 8-22 高速 I<sup>2</sup>C 总线器件和快速 I<sup>2</sup>C 总线器件连接图

### (3) 快速模式或标准模式 I<sup>2</sup>C 总线的双向电平转换器

由于现代集成电路工艺的发展，出现了很多低功耗低电压的器件，为了将这些低功耗低电压的器件与已有的 5V 电路连接，就需要一个电平转换器。对于双向的 I<sup>2</sup>C 总线，电平转换器也要求必须是双向的。最简单的解决方法是连接一个分立的 MOS-FET 管到每条总线线路。电平转换电路如图 8-23 所示。

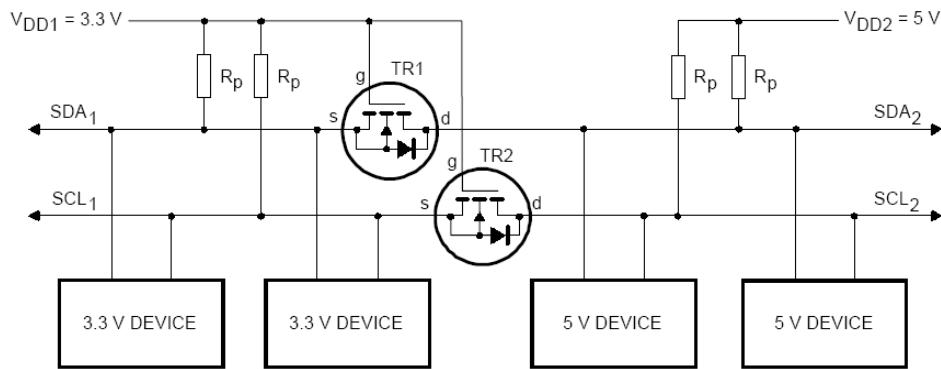


图 8-23 电平转换电路

### (4) 10 位 I<sup>2</sup>C 总线寻址

伴随着 I<sup>2</sup>C 总线器件的不断增多，最初的 7bit 的寻址空间已经难以满足人们的需要。为了解决这一问题，人们对 I<sup>2</sup>C 总线的规范进行了修改，增加了 10bit 寻址方式。10bit 的寻址方式可寻址高达 1 024 个地址编码，而且它也遵从最初的 I<sup>2</sup>C 总线规范的地址格式，因此，

7bit 与 10bit 的地址编码可以用在同一 I<sup>2</sup>C 总线上，并可以任意应用于标准模式、快速模式以及高速模式中。

前面已经讲述过，7bit 地址是由一个字节构成的，高 7 位为从器件地址，最低位为指明数据传输方向的 R/W。与 7bit 地址不同的是，10bit 地址是由两个字节构成，第一个字节高 7 位为 1-1-1-1-0-X-X，第一字节的最低位为 R/W，XX 与第二字节一起组成 10bit 的从地址，这样的从地址结构，就保证了 10bit 寻址方式与 7bit 寻址方式的兼容性。

#### 8.4.2 I<sup>2</sup>C 总线硬件接口设计

本节介绍使用带有 I<sup>2</sup>C 总线硬件接口的单片机进行 I<sup>2</sup>C 总线系统设计的方法。这里使用的单片机是 Philips 公司推出的 P89C66X 系列单片机，该系列芯片的 I<sup>2</sup>C 总线接口与 Philips 推出的 P8XC552、P8XC654 以及 P8XC652 系列单片机是全兼容的，其片内的 I<sup>2</sup>C 总线逻辑提供了符合 I<sup>2</sup>C 总线规范的串行接口，具有性能稳定、速度快、使用方便等优点。

##### 1. P89C66X 系列单片机 I<sup>2</sup>C 总线接口引脚设计

P89C66X 系列单片机有 44 脚的 PLCC 封装和 LQFP 封装两种，其中 P1.6/SCL 和 P1.7/SDA 分别为 I<sup>2</sup>C 总线的时钟线和数据线，由于芯片内部采用漏极开路工艺，所以当用户把这两个引脚作为 I<sup>2</sup>C 总线的接口使用的时候，需要外接上拉电阻，如图 8-24 所示。

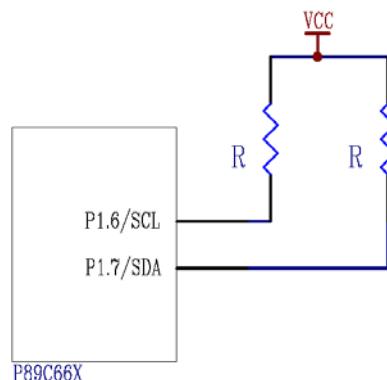


图 8-24 P89C66X I<sup>2</sup>C 总线接口引脚

##### 2. P89C66X 系列单片机 I<sup>2</sup>C 总线相关寄存器

P89C66X 系列单片机内部与 I<sup>2</sup>C 总线相关的寄存器共有 4 个，通过对这 4 个寄存器的编程实现 I<sup>2</sup>C 总线功能。这 4 个寄存器分别为 S1ADR、S1DAT、S1CON 和 S1STA，如表 8-3 所示。

表 8-3 P89C66X 系列单片机的 I<sup>2</sup>C 总线寄存器

| 名称    | 寄存器   | 说明                                |
|-------|-------|-----------------------------------|
| 地址寄存器 | S1ADR | 该寄存器用于保存单片机本身的从地址                 |
| 数据寄存器 | S1DAT | 该寄存器用于保存要发送或者接收到的数据字节             |
| 控制寄存器 | S1CON | 该寄存器用于 I <sup>2</sup> C 总线的设置     |
| 状态寄存器 | S1STA | 该寄存器用于显示当前 I <sup>2</sup> C 总线的状态 |

下面对其分别进行介绍。

### (1) 地址寄存器 S1ADR

该寄存器用于保存单片机本身的从地址，CPU 可以对该寄存器进行读写操作。其中高 7 位为地址本身，最低位为通用地址识别标志 GC。该寄存器结构如下：

| D7    | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|-------|----|----|----|----|----|----|----|
| 器件从地址 |    |    |    |    |    |    | GC |

当 CPU 作为主器件存在的时候，该寄存器不起作用，当 CPU 作为从器件存在的时候，接收到的地址字节的高 7 位将与 S1ADR 中的值相比较，如果相同则接收后面的数据信息。此外，如果通用地址识别标志 GC 为 0，单片机不识别通用调用地址（如广播地址），GC 为 1 时，单片机识别通用调用地址。

### (2) 数据寄存器 S1DAT

S1DAT 寄存器用于保存要发送或者接收到的数据字节。当该寄存器没有进行移位读取数据操作时，CPU 可对其进行读写。其逻辑电路如图 8-25 所示。

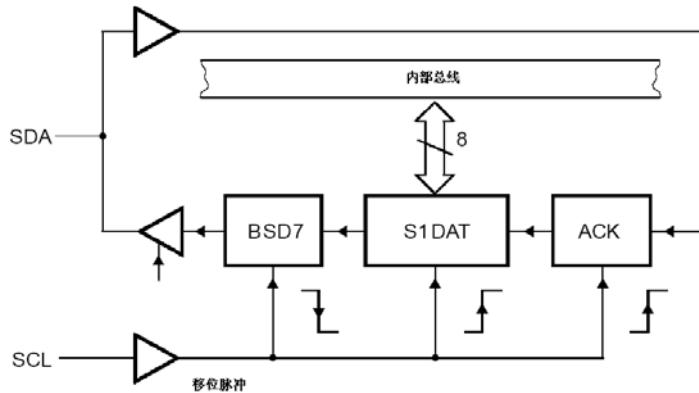


图 8-25 S1DAT 内部逻辑电路

### (3) 控制寄存器 S1CON

该寄存器用于对 I<sup>2</sup>C 总线进行设置，其各功能位定义如下：

| D7  | D6   | D5  | D4  | D3 | D2 | D1  | D0  |
|-----|------|-----|-----|----|----|-----|-----|
| CR2 | ENS1 | STA | ST0 | SI | AA | CR1 | CR0 |

- ENS1 是 I<sup>2</sup>C 总线使能位。当 ENS1 为 0 时，SDA 和 SCL 输出为高阻状态，当 ENS1 为 1 时，I<sup>2</sup>C 总线使能。
- STA 是 I<sup>2</sup>C 总线起始条件标志位。当 STA 为 0 时，单片机不产生起始条件，当 STA 为 1 时，单片机首先利用硬件检测 I<sup>2</sup>C 总线状态。如果 I<sup>2</sup>C 总线为空闲状态，则立刻向 I<sup>2</sup>C 总线发送起始条件，如果 I<sup>2</sup>C 总线为忙状态，则等待到总线为空闲状态后，再发送起始条件。
- ST0 是 I<sup>2</sup>C 总线停止条件标志位。当 ST0 为 0 时，单片机不产生停止条件，当 ST0 为 1 时，如果单片机处于主模式，则立刻向 I<sup>2</sup>C 总线上发送停止条件，如果单片机处于从模式，则可以从错误条件中恢复出来。ST0 位可由硬件清零。如果 STA 和 ST0 同时被置位，则单片机先向 I<sup>2</sup>C 总线发送一个停止条件，再发送一个停止条件。
- SI 是串行中断标志位。当 SI 标志位置位，并且 EA 和 ES1 都置位时，产生一个串行中

断请求。单片机就可以对接收到的数据进行处理，处理完成后，SI 必须由软件清零。当 SI 位为 0 时，将不再产生串行中断请求。

- AA 是声明应答标志位。当 AA 位为 1 时，如果器件接收到自身的从地址（或广播地址），或者接收到一个完整的数据字节之后，将会在 SCL 的响应时钟脉冲期间保持 SDA 为低电平，即发送应答信号。当 AA 位为 0 时，器件在接收到一个完整的数据字节之后，在 SCL 的响应时钟脉冲期间保持 SDA 为高电平，即发送非应答信号。
- CR0、CR1 和 CR2 是串行时钟速率选择位。这 3 位决定了器件在主模式下的串行时钟速率，其值与单片机工作频率有关，如表 8-4 所示。

表 8-4 可变串行时钟速率

| CR2 | CR1 | CR0 | 单片机工作频率                    |                            |                            | 分频数             |
|-----|-----|-----|----------------------------|----------------------------|----------------------------|-----------------|
|     |     |     | 6MHz                       | 12MHz                      | 16MHz                      |                 |
| 0   | 0   | 0   | 23                         | 47                         | 62.5                       | 256             |
| 0   | 0   | 1   | 27                         | 54                         | 71                         | 224             |
| 0   | 1   | 0   | 31                         | 63                         | 83.3                       | 192             |
| 0   | 1   | 1   | 37                         | 75                         | 100                        | 160             |
| 1   | 0   | 0   | 6.25                       | 12.5                       | 17                         | 960             |
| 1   | 0   | 1   | 50                         | 100                        | 133                        | 120             |
| 1   | 1   | 0   | 100                        | 200                        | 267                        | 60              |
| 1   | 1   | 1   | 0.24 $>$ 62.5<br>0 $<$ 255 | 0.49 $<$ 62.5<br>0 $<$ 254 | 0.65 $<$ 55.6<br>0 $<$ 253 | 定时器在模式 2 时的重装数值 |

#### (4) 状态寄存器 S1STA

状态寄存器 S1STA 用于显示当前 I<sup>2</sup>C 总线的状态，该寄存器是只读的。寄存器的低 3 位始终为 0，高 5 位为总线状态编码。总线共有 26 种可能的状态，每种状态都有一个固定的状态编码。总线每进入一个状态都会产生串行中断请求，并将 SI 置位，SI 置位一个机器周期后，当前 S1STA 中的代码有效。在 SI 由软件复位的一个机器周期之后，此代码仍然存在。

### 3. 基于 P89C66X 系列单片机的 I<sup>2</sup>C 接口的软件设计

针对 P89C66X 系列单片机的 I<sup>2</sup>C 总线进行软件设计，用户首先要了解 I<sup>2</sup>C 总线的 26 个状态，其次要根据应用中要求的模式来设定各个寄存器。例如单片机处于主模式，则 S1ADR 就不会被用到。本节将根据 P89C66X I<sup>2</sup>C 总线的不同模式分别给出 C51 的函数代码，并介绍其设计方法。

#### (1) 主模式 C51 函数

下面给出硬件 I<sup>2</sup>C 以主方式工作的 C51 函数库，它包括了申请占用总线、发送字节数据、接收字节数据等函数，在使用这些函数之前，需要在程序开始包含 reg66x.h 文件。

##### • 申请占用总线

在实际应用中，要进行 I<sup>2</sup>C 的数据传输，首先要对 I<sup>2</sup>C 总线进行初始化并申请占用总线。这一任务是由 getbus() 函数完成的，其具体程序如下：

```
void getbus()
{
 S1CON = 0x43; // 设置时钟为 75k(12M), ENS1 置位
```

```

STA = 1; // 申请总线主机, 启动总线
while(SI == 0); // 等待起始位的发送
}

```

- 发送一个数据字节

在主器件申请占用总线后, 如果 I<sup>2</sup>C 总线为空闲状态, 则主器件立刻占用 I<sup>2</sup>C 总线。这时, 由于总线的状态发生变化, S1STA 的状态编码也发生了变化, 导致产生了串行中断, 即 S1CON 中的 SI 位被置位。因此在发送数据后, 可以通过检测 SI 位是否为 0 检测数据是否发送完毕。

发送字节数据由函数 sendbyte() 完成。在该函数中, 首先将要发送的数据装入 S1DAT, 然后软件清除 SI 位以便进行检测。程序代码如下:

```

void sendbyte(unsigned char c)
{
 S1DAT = c; // 将要发送的数据装入 S1DAT
 S1CON = 0x43; // 清除 SI 位
 while(SI == 0); // 等待数据发送
}

```

- 向无子地址器件发送单字节数据

该函数用于单片机在主模式下向从器件发送数据。首先, 单片机要申请总线, 在得到总线的控制权后, 发送从器件的 7bit 地址编码, 这时, 判断 S1STA 的值是否为 0x18 (0x18 代表的状态为主器件已经发送 SLA+W, 并且收到应答), 如果是 0x18, 则继续发送数据字节, 否则表示发送发生错误, 函数跳出。然后程序判断 S1STA 的值是否为 0x28 (0x28 代表的状态是主器件已经发送 S1DAT 中的数据, 并受到应答), 如果是 0x28, 则表示发送数据成功, 这时, SI 复位, 并返回。

该功能由 isendbyte() 函数完成, 其返回值是 bit 型, 如果执行结果正确, 返回 1, 否则返回 0。具体程序如下:

```

bit isendbyte(unsigned char sla, unsigned char c)
{
 getbus(); // 启动总线
 sendbyte(sla); // 发送从器件地址, 若无应答则返回
 if(S1STA != 0x18)
 {
 S1CON = 0x53;
 return 0;
 }
 sendbyte(c); // 发送数据
 if(S1STA != 0x28)
 {
 S1CON = 0x53;
 return 0;
 }
}

```

```

S1CON = 0x43; //结束总线
return 1;
}

```

- 向无子地址器件读字节数据

该函数为 `irecvbyte()`，用于单片机在主模式下向从器件发送数据。单片机在申请到总线后发送从器件地址，随后检测总线状态，如果 S1STA 寄存器的值为 0x40（0x40 代表的状态为主器件已经发送 SLA+R，并且收到应答），表明地址发送成功，程序准备接收数据字节，否则表示发送发生错误，函数跳出。程序在接收字节后判断 S1STA 的值是否为 0x58（0x58 代表的状态是主器件收到数据字节，并返回应答），如果是 0x58，则表示发送数据成功，这时，SI 复位，并返回。

函数返回值是 bit 型，如果执行结果正确，返回 1，否则返回 0。具体程序如下：

```

bit irecvbyte(unsigned char sla, unsigned char *c)
{
 gerbus(); //启动总线
 sendbyte(sla+1); //发送器件地址
 if(S1STA != 0x40)
 {
 S1CON = 0x53;
 return 0;
 }
 S1CON = 0x43; //接收一字节数据即发送非应答位
 while(SI == 0); //等待接收数据
 if(S1STA != 0x58)
 {
 S1CON=0x53;
 return 0;
 }
 *c = S1DAT;
 S1CON = 0x53;
 return 1;
}

```

- 向有子地址器件发送和接收多字节数据

在 I<sup>2</sup>C 总线器件中，每个器件除了拥有其自身的从地址外，部分器件还可以具有子地址。在对子地址指向的空间通信时，主器件需要依次发送从地址和子地址才可以正常寻址。`isendstr()` 函数和 `irecvstr()` 函数分别用于向有子地址的器件发送和接收多字节数据，这两个函数的 4 个参数是相同的，分别为从器件地址 `sla`、子地址 `suba`、发送或接收数据缓冲区地址 `s` 和字节数 `no`。其实现过程与前面发送和接收字节的程序基本相同，只是要注意在发送器件从地址后，还要发送一个子地址信息才可以继续发送或接收具体数据，数据部分使用循环连续发送或接收多个字节数据。这两个函数的流程图如图 8-26 所示。

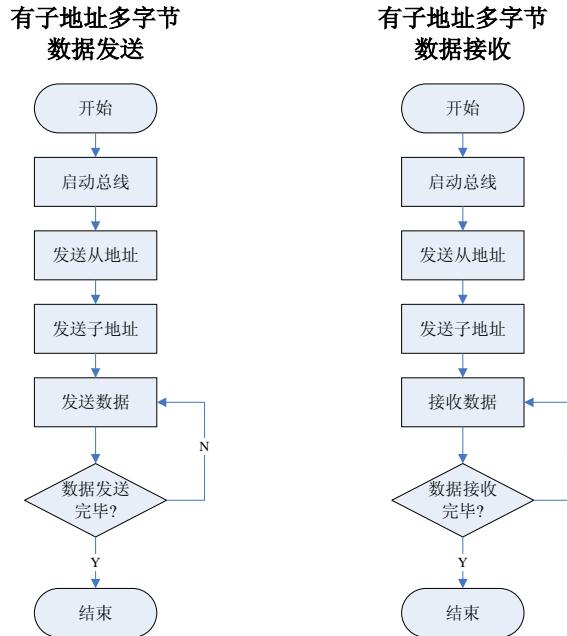


图 8-26 有子地址器件发送和接收多字节数据函数流程图

## (2) 从模式 C51 函数

当单片机工作在从模式方式下时，由于其自身不触发总线，因此不用考虑总线仲裁与竞争，只需要响应主控器的请求即可。从器件不允许控制总线，也无需解决总线的意外或挂起，若发现总线出错，只要退出即可。从模式下的 C51 程序共有 3 个函数，分别为设置总线函数 setbus()、发送字节函数 sendbyte() 和接收字节函数 recvbyte()。

- 设置总线函数

该函数为 sendbus()，用于设置 I<sup>2</sup>C 控制寄存器，包括总线时钟、速率及从地址。注意将 I<sup>2</sup>C 接口设置为不接收广播地址。函数实现代码如下：

```
void setbus(unsigned char addr)
{
 S1ADR=addr&0xfe; //设置从地址,屏蔽高 7 位即广播地址响应位复位
 S1CON=0XC5; //启动硬件 I2C
}
```

- 发送字节函数

函数 sendbyte() 用于向 I<sup>2</sup>C 总线发送字节数据。该函数中，程序先检查状态寄存器以确定上一字节是否发送完成（0xc0 表示发送完成），如果发送完成则发送当前数据，否则函数返回 0 跳出。这部分程序代码如下：

```
bit sendbyte(unsigned char c)
{
 if(S1STA == 0xc0)
 {
 S1CON = 0XC5;
```

```

 return 0;
 }

 S1DAT = c; //发送数据
 S1CON = 0XC5; //释放总线
 while(SI == 0); //等待字节数据发送完成
 return 1;
}

```

- 接收字节函数

函数 recvbyte() 用于读取 I<sup>2</sup>C 总线传来的字节数据并发送应答位。进入该函数后，程序首先清除标志位并释放总线等待接收。接收到数据后，程序检查状态寄存器 S1STA，若其值为 0xa0，表示 I<sup>2</sup>C 总线接口成功接收到数据并作出应答，程序读取数据寄存器内容并将其保存，否则表示数据读取错误，函数返回 0 后跳出。该函数的实现代码如下：

```

bit recvbyte(unsigned char *c)
{
 S1CON = 0XC5; //清除标志位
 while(SI == 0); //放开总线等待接收
 if(S1STA == 0xa0)
 {
 S1CON = 0XC5; //先放开总线再返回 0
 return 0;
 }
 *c = S1DAT; //取数据
 return 1;
}

```

#### 8.4.3 I<sup>2</sup>C 总线模拟硬件接口软件设计

使用硬件接口进行 I<sup>2</sup>C 总线设计，要使用具有 I<sup>2</sup>C 总线硬件接口的单片机。对通常的单片机，要支持 I<sup>2</sup>C 总线，可以选择外接 I<sup>2</sup>C 总线芯片，但这要增加系统的整体成本，另一种解决的方案是使用软件 I/O 口模拟，使用程序控制实现 I<sup>2</sup>C 总线的时序。下面给出一个使用单片机的 P1.0 和 P1.1 口模拟 I<sup>2</sup>C 总线接口的函数库，读者只要参照前面介绍的 I<sup>2</sup>C 总线的时序就可以比较轻松地理解下面的代码。

```

/* 单主器件 I2C 总线模拟子程序 */

#include <At89x51.h>
#include <intrins.h>

#define uchar unsigned char
#define DELAY5US _nop_();_nop_();_nop_();_nop_(); //需要根据晶振频率调整

```

```

sbit VSDA=P1^0; // 将 p1.0 口模拟数据口
sbit VSCL=P1^1; // 将 p1.1 口模拟时钟口

uchar idata SLA; //从器件地址
uchar idata SLAW; //从器件写地址
uchar idata SLAR; //从器件读地址
uchar idata NUMBYT; //数据传送字节
uchar idata MTD[10]; //数据发送缓冲区
uchar idata MRD[10]; //数据接收缓冲区
bit bdata NACK; //器件坏或错误标志位

/* 启动 I2C 总线子程序 */
void STA(void)
{
 VSDA = 1; // 启动 I2C 总线
 VSCL = 1;
 DELAY5US //延时 4.7us, 根据晶振频率调整空操作个数, 这里以 fosc=12MHz, 下同
 VSDA = 0;
 DELAY5US
 VSCL = 0;
}

/* 停止 I2C 总线数据传送子程序 */
void STOP(void)
{
 VSDA = 0; //停止 I2C 总线数据传送
 VSCL = 1;
 DELAY5US
 VSDA = 1;
 DELAY5US
 VSCL = 0;
}

/* 发送应答位子程序 */
void MACK(void)
{
 VSDA = 0; //发送应答位
 VSCL = 1;
 DELAY5US
 VSDA = 1;
}

```

```

VSCL = 0;
}

/* 发送非应答子程序 */
void MNACK(void)
{
 VSDA = 1; //发送非应答位
 VSCL = 1;
 DELAY5US
 VSDA = 0;
 VSCL = 0;
}

/* 应答位检查子程序 */
void CACK(void)
{
 VSDA = 1; //应答位检查（将 p1.0 设置成输入，必须先向端口写 1）
 VSCL = 1;
 F0 = 0;
 if(VSDA == 1) //若 VSDA=1 表明非应答，置位非应答标志 F0
 F0 = 1;
 VSCL = 0;
}

/* 发送一个字节数据子程序，程序入口 p 为发送缓冲区地址 */
void WRBYT(uchar iodata *p)
{
 uchar iodata n=8; //向 VSDA 上发送一位数据字节，共 8 位
 uchar iodata temp;
 temp=*p;
 while(n--)
 {
 if((temp&0x80) == 0x80) //若要发送的数据最高位为 1 则发送位 1
 {
 VSDA = 1; // 传送位 1
 VSCL = 1;
 DELAY5US
 VSDA = 0;
 VSCL = 0;
 }
 }
}

```

```

 else
 {
 VSDA = 0; // 否则传送位 0
 VSCL = 1;
 DELAY5US
 VSCL = 0;
 }
 temp = temp<<1; // 数据左移一位, 或_crol_(*p, 1)
 }
}

/* 接收一字节子程序, 入口参数 p 为接收缓冲区地址 */
void RDBYT(uchar iodata *p)
{
 uchar iodata n=8; // 从 VSDA 线上读取一上数据字节, 共 8 位
 uchar iodata temp=0;
 while(n--)
 {
 VSDA = 1;
 VSCL = 1;
 temp = temp<<1; // 左移一位, 或_crol_(temp, 1)
 if(VSDA == 1)
 temp = temp|0x01; // 若接收到的位为 1, 则数据的最后一位置 1
 else
 temp = temp&0xfe; // 否则数据的最后一位置 0
 VSCL=0;
 }
 *p=temp;
}

/* 发送 n 位数据子程序 */
void WRNBYT(uchar *sla, uchar n)
{
 uchar iodata *p;
 STA(); // 启动 I2C
 WRBYT(sla); // 发送一上位数据
 CACK(); // 检查应答位
 if(F0 == 1)
 {
 NACK = 1;
 }
}

```

```

 return; //若非应答表明器件错误或已坏，置错误标志位 NACK
 }
 p = MTD;
 while(n--)
 {
 WRBYT(p);
 CACK(); //检查应答位
 if (F0 == 1)
 {
 NACK=1;
 return; //若非应答表明器件错误或已坏，置错误标志位 NACK
 }
 p++;
 }
 STOP(); //全部发完则停止
}

/* 接收 n 位数据子程序 */
void RDNBYT(uchar iodata *sla, uchar n)
{
 uchar iodata *p;
 STA();
 WRBYT(sla);
 CACK();
 if(F0 == 1)
 {
 NACK = 1;
 return;
 }
 p = MRD;
 while(n--)
 {
 RDBYT (p);
 MACK(); //收到一个字节后发送一个应答位
 p++;
 }
 MNACK(); //收到最后一个字节后发送一个非应答位
 STOP();
}

```

### 8.4.4 I<sup>2</sup>C 总线系统的设计要点

在硬件系统中应用 I<sup>2</sup>C 总线进行 IC 的连接控制，可以有效的减少单片机 I/O 口资源的占用，这在单片机应用中是十分重要的，这也是 I<sup>2</sup>C 总线得以广泛应用的主要原因。此外，I<sup>2</sup>C 总线在设计上的思想也是十分突出的，其模块化的设计思想符合硬件系统设计的潮流。应用 I<sup>2</sup>C 总线进行设计，可以将整个硬件系统分为不同的功能模块，每个功能模块分别设计，不会收到系统中其它模块的影响，设计完成后，使用统一的 I<sup>2</sup>C 总线接口连入系统，这样的系统各个部分不会互相干扰，每一个模块只与自身电路相关，大大简化了设计和调试的过程，也提高了通用设备的复用率。

例如上面提到的路桥称重系统中，智能 kB 键盘本身就是一个独立的功能模块，其内部包括键盘阵列以及一个 8051 单片机，可以检测按键信息并将事先设定的对应键码直接通过 I<sup>2</sup>C 总线返回给主器件。这样在设计中，主器件根本无需考虑任何与键盘相关的操作，只需根据中断读取相应的键值即可，而 kB 键盘的单片机只需要负责键盘的按键检测、消抖、译码等操作，无需考虑其它任何部分的硬件功能的实现，二者在设计上互不干扰。此外，作为独立设计的 kB 键盘模块还具有通用性，可以直接用于其他系统中提供键盘的功能，减少了硬件的重复设计。

读者在进行 I<sup>2</sup>C 总线的设计时，需要首先将整体系统的功能进行划分，对各个不同的功能部分进行独立设计，最后使用 I<sup>2</sup>C 总线接口将整个系统连接。在设计的过程中，每完成一个模块的设计，就可以进行调试，直至完全达到设计的要求，再开始下一个模块的设计，这样所有模块设计完成后，可以直接连入系统进行最后的整合工作，这样整个调试过程中出现任何问题都可以直接找到对应的部分，降低了系统调试的难度。

## 8.5 单片机与计算机的互连

### 8.5.1 电路设计

和 PC 相连是嵌入式系统的一个趋势。这里将 A/D, D/A 控制的结果传输到计算机上进行监测。计算机端的人机交互界面要友好得多，在计算机端，就可以根据获取的数据进行一定的控制，或者由软件自动实现，或者由人工来控制。

计算机的串口采用的是 RS232 电平协议，是 12V 的电压，而我们单片机系统则采用的是 TTL 电平，是 0~+5V 的电压。因此需要将 TTL 电平转换成 RS232 电平。如图 9-23 所示，我们采用的芯片是 MAX232。完整电路图如图 8-27 所示。

MAX232 芯片的主要性能参数如下：

- 工作电压：单电源+5V；
- 双通道接收和发送；
- 与所有 EIA/TIA-232E 以及 V.28 协议兼容；
- 三态门接收和发送；

更详尽的资料请参阅 MAX232 的性能数据表单。

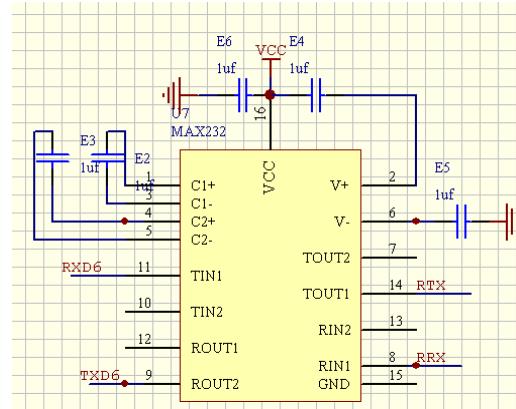


图 8-27 TTL 电平与 RS232 电平的互转换器 MAX232

### 8.5.2 电路的 C51 程序代码

单片机端的代码参见下列代码清单，我们采用和多机通信一样的设置，则初始化过程，初始化函数，以及发送函数，接收函数，都应该与 9.10 节中的一样，因此，这里也只列出主函数。

```
#define MACHINECOMM_DATAHEAD 0x5a
//以上语句定义数据包的头
#define MACHINECOMM_DATAEND 0xa5
//以上语句定义数据包的尾
void main(void)
{
 unsigned char datacomm[6], datarecv[6];
 unsigned char count=0, datahand;
 datahand=MACHINECOMM_RET;
 datacomm[0]=MACHINECOMM_DATAHEAD;
 datacomm[5]=MACHINECOMM_DATAEND;
 datacomm[1]='w';
 datacomm[2]='o';
 datacomm[3]='r';
 datacomm[4]='k';
 initial();
 SelectComm6=1;
 //选择与计算机进行通信。
 while(1)
 {
 if(EvRcv)
 //如果接收到主机端发送过来的握手信号
```

```

{
 if(recvdata==MACHINECOMM_OK)
 {
 LampComm=TRUE;
 delay(DELAY_VALUE);
 LampComm=FALSE;
 }

 //指示灯闪烁
 EvRcv=FALSE;
 break;

 //退出等待
}

send(&datahand, 1);

//返回一个握手信号
break;

}

for(;;) {
 while(1)
 {if(EvRcv) {

 datarecv[count]=recvdata;
 count++;
 EvRcv=FALSE;
 }

 if(count==5)break;
 //如果已经收到六个数据，则认为是收到一个数据包
}

//接收一个数据包
if(datarecv[0]==MACHINECOMM_DATAHEAD&&datarecv[5]==MACHINECOMM_DATAEND)
 send(datacomm, 6);

//如果数据包正确，则返回一个数据包;
}
}
}

```

### 8.5.3 计算机端的 Visual C++ 程序代码

计算机端需要编制一个类，我们用 Visual C++ 进行编制。应用程序可以调用下面的类来进行自己的控制操作。代码分别如下，下面为串口通信类，首先是类函数的定义：

```

// Port.h: interface for the CPort class.

#ifndef AFX_PORT_H__EA70F052_D6AE_11D3_841C_0000B4B5BC6E__INCLUDED_
#define AFX_PORT_H__EA70F052_D6AE_11D3_841C_0000B4B5BC6E__INCLUDED_

```

```

#ifndef _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CPort
{
public:
 BOOL SetMark(BOOL bMark=TRUE);
 //返回端口状态, 是否已经初始化
 BOOL Init();
 //清缓存
 void ClearBuffer();

 //从端口读一个字符, 如果没有则等到有为止
 BOOL WaitChar(unsigned char * pCharData);

 //从端口读 nNumToRead 个字符, 存在以 pDataAddr 为首地址的缓存中
 //m_nWait 表示输入缓存中如果没有足够的数据, 等待时间的长度
 BOOL ReadData(LPVOID pDataAddr, DWORD nNumToRead, UINT m_nWait=500);

 //从端口读一个字符, cReadData 接收字符, cParity 接收第九位, nWait-如果缓冲中没有字符, 等待时间的长度
 BOOL ReadChar(unsigned char *cReadData, UINT nWait=100);

 //往端口写字符, cWriteData 为要写字符的指针
 BOOL WriteChar(unsigned char cWriteData);

 //往端口写字符串, nNumToWrite 为要写字符串的长度, pDataAddr 为字符串首地址
 BOOL WriteData(LPVOID pDataAddr, DWORD nNumToWrite);

 //初始化端口
 BOOL Init1(UINT portnr, DWORD dwCommEvents, UINT writebuffersize);
 BOOL InitPort(UINT portnr, DCB &dcb, DWORD dwCommEvents, UINT writerbuffersize);
 BOOL InitPort(UINT portnr, UINT baud, char parity, UINT databits, UINT stopbits, DWORD dwCommEvents, UINT writebuffersize);

 CPort();
 virtual ~CPort();
}

```

```

protected:
 //端口是否已经初始化了
 BOOL m_bInited;

 //同步变量
 OVERLAPPED m_ov;

 // DCB
 DCB m_dcb;

 //端口句柄
 HANDLE m_hComm;

 COMMTIMEOUTS m_CommTimeouts;

 //端口号
 UINT m_nPortNr;

 //写缓存
 char* m_szWriteBuffer;

 DWORD m_dwCommEvents;
 DWORD m_nWriteBufferSize;
};

#endif // !defined(AFX_PORT_H__EA70F052_D6AE_11D3_841C_0000B4B5BC6E__INCLUDED_)

// Port.cpp: implementation of the CPort class.
//
//

```

然后是对上面的串口通信类的具体功能的实现，下面是代码清单：

```

#include "stdafx.h"
#include "Port.h"

#ifndef _DEBUG
#define THIS_FILE
static char THIS_FILE[]=_FILE_;
#define new DEBUG_NEW
#endif

#ifndef _DEBUG

```

```
#define DebugMsgStr(a) AfxMessageBox(a)
#define DebugMsgStr(a) 0
#endif
//
// Construction/Destruction
//

CPort::CPort()
{
 //初始化数据
 m_hComm = NULL;
 m_szWriteBuffer = NULL;
 m_bInited=FALSE;

 m_ov.Offset = 0;
 m_ov.OffsetHigh = 0;
 m_ov.hEvent = NULL;
}

CPort::~CPort()
{
 CloseHandle(m_hComm);
}

//初始化端口
BOOL CPort::Init1(UINT portnr, DWORD dwCommEvents, UINT writebuffersize)
{
 //端口号只能是1到4
 if (portnr<1||portnr>4)
 {
 DebugMsgStr("端口号超出范围!");
 return FALSE;
 }

 //设置 m_ov 的事件
 if (m_ov.hEvent != NULL)
 ResetEvent(m_ov.hEvent);
 m_ov.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
```

```

//BOOL bResult;//得到函数调用结果
char *szPort = new char[5];//端口号对应的字符串

//生成要求的字符串
sprintf(szPort, "COM%d", portnr);

//对类变量进行赋值
if (m_szWriteBuffer != NULL)
 delete [] m_szWriteBuffer;
m_szWriteBuffer = new char[writebuffersize];
m_nPortNr = portnr;
m_nWriteBufferSize = writebuffersize;
m_dwCommEvents = dwCommEvents;

//设置超时间隔
m_CommTimeouts.ReadIntervalTimeout = 50;
m_CommTimeouts.ReadTotalTimeoutMultiplier = 50;
m_CommTimeouts.ReadTotalTimeoutConstant = 50;
m_CommTimeouts.WriteTotalTimeoutMultiplier = 50;
m_CommTimeouts.WriteTotalTimeoutConstant = 50;

//如果端口已打开了，就把它关闭
if (m_hComm != NULL)
{
 CloseHandle(m_hComm);
 m_hComm = NULL;
}

//创建端口
m_hComm = CreateFile(szPort, // 端口号字符串
 GENERIC_READ | GENERIC_WRITE, // 读写类型
 0, // 绝对访问
 NULL, // 没有安全属性
 OPEN_EXISTING, // 打开方式
 FILE_FLAG_OVERLAPPED, // 异步
 0); // 必须为 0

//如果创建失败则返回
if (m_hComm == INVALID_HANDLE_VALUE)
{

```

```

 delete [] szPort;
 // delete [] szBaud;
 return FALSE;
 }

 //设置端口的其他属性
 //超时
 if (!SetCommTimeouts(m_hComm, &m_CommTimeouts))
 {
 DebugMsgStr("设置端口超时错误！");
 return FALSE;
 }
 if (!SetCommMask(m_hComm, dwCommEvents))
 {
 DebugMsgStr("设置端口触发事件错误！");
 return FALSE;
 }
 return TRUE;
}

BOOL CPort::InitPort(UINT portnr, DCB &dcb, DWORD dwCommEvents, UINT writebuffersize)
{
 if(!Init1(portnr, dwCommEvents, writebuffersize))
 return FALSE;
 if (!SetCommState(m_hComm, &dcb))
 {
 DebugMsgStr("设置端口 DCB 错误！");
 return FALSE;
 }

 //清空缓存
 PurgeComm(m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT | PURGE_TXABORT);

 m_bInit=TRUE;
 return TRUE;
}

BOOL CPort::InitPort(UINT portnr, // 端口号, 数字 1-4
 UINT baud, // 数据传输率, 整数, 如 9600, 等。
 char parity, // 奇偶校验位, 字符
 // N-无校验 0-奇校验 E-偶校验
 // M-保持为 1 S-保持为 0 , 这两项一般用于多机通信模

```

式

```

 UINT databits, // 数据位长度, 整数
 UINT stopbits, // 停止位长度, 整数, 1 或 2
 DWORD dwCommEvents, // 触发事件
 UINT writebuffersize)//写缓存的长度

 {

 if (!Init1(portnr, dwCommEvents, writebuffersize))
 return FALSE;

 char *szBaud = new char[50];//数据传输率对应的字符串
 sprintf(szBaud, "baud=%d parity=%c data=%d stop=%d", baud, parity, databits, stopbits);

 DCB m_dcb;
 if (!GetCommState(m_hComm, &m_dcb))
 {
 DebugMsgStr("试图得到 DCB 错误！");
 return FALSE;
 }

 m_dcb.fRtsControl = RTS_CONTROL_ENABLE; // set RTS bit high!

 if (!BuildCommDCB(szBaud, &m_dcb))
 {
 DebugMsgStr("生成 DCB 错误！");
 return FALSE;
 }
 if (!SetCommState(m_hComm, &m_dcb))
 {
 DebugMsgStr("设置端口 DCB 错误！");
 return FALSE;
 }

 //清空缓存
 PurgeComm(m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT | PURGE_TXABORT);

 m_bInitd=TRUE;
 return TRUE;
 }

 //往端口写字符, cWriteData 为要写字符的指针

```

```
BOOL CPort::WriteChar(unsigned char cWriteData)
{
 if (!m_bInit)
 {
 DebugMsgStr("端口尚未初始化!");
 return FALSE;
 }

 //初始化变量
 DWORD BytesSent = 0;
 m_ov.Offset = 0;
 m_ov.OffsetHigh = 0;

 BOOL bResult;
 //写字符
 bResult=WriteFile(m_hComm, &cWriteData, 1, &BytesSent, &m_ov);

 if (bResult)
 {
 //成功则返回
 return TRUE;
 }

 //得到错误代码
 DWORD dwError = GetLastError();

 //如果不是尚未完成则返回错误信号
 if (dwError!=ERROR_IO_PENDING)
 {
 DebugMsgStr("写字符错误！");
 return FALSE;
 }

 //尚未完成则等待完成
 if (!GetOverlappedResult(m_hComm, &m_ov, &BytesSent, TRUE))
 {
 //失败返回错误信号
 DebugMsgStr("写字符错误！");
 return FALSE;
 }
}
```

```
//成功
return TRUE;
}

//往端口写字符串, nNumToWrite 为要写字符串的长度, pDataAddr 为字符串首地址
BOOL CPort::WriteData(LPVOID pDataAddr, DWORD nNumToWrite)
{
 if (!m_bInited)
 {
 DebugMsgStr("端口尚未初始化!");
 return FALSE;
 }

 //初始化变量
 DWORD BytesSent = 0;
 m_ov.Offset = 0;
 m_ov.OffsetHigh = 0;

 BOOL bResult;
 //写字符
 bResult=WriteFile(m_hComm, pDataAddr, nNumToWrite, &BytesSent, &m_ov);

 if (bResult)
 {
 //成功则返回
 return TRUE;
 }

 //得到错误代码
 DWORD dwError = GetLastError();

 //如果不是尚未完成则返回错误信号
 if (dwError!=ERROR_IO_PENDING)
 {
 DebugMsgStr("写字符串错误!");
 return FALSE;
 }

 //尚未完成则等待完成
```

```
GetOverlappedResult(m_hComm, &m_ov, &BytesSent, TRUE);
if (BytesSent!=nNumToWrite)
{
 //失败返回错误信号
 DebugMsgStr("没有足够的字符数!");
 return FALSE;
}

//成功
return TRUE;
}

//从端口读一个字符, cReadData 接收字符, nWait-如果缓冲中没有字符, 等待时间的长度
BOOL CPort::ReadChar(unsigned char *cReadData, UINT nWait)
{
 BOOL bResult;
 DWORD dwError, dwBytesRead;
 COMSTAT comstat;

 if (!m_bInitied)
 {
 DebugMsgStr("端口尚未初始化!");
 return FALSE;
 }

 //得到当前输入缓存状态
 bResult = ClearCommError(m_hComm, &dwError, &comstat);

 //如果没有数据且不等待则返回错误
 if (comstat.cbInQue==0&&nWait==0)
 {
 DebugMsgStr("CPort::ReadChar-没数据, 不等了!");
 return FALSE;
 }

 //读字符
 bResult = ReadFile(m_hComm, cReadData, 1, &dwBytesRead, &m_ov);

 if (!bResult)
```

```

 //如果不是尚未完成则返回错误
 if (dwError=GetLastError () !=ERROR_IO_PENDING)
 {
 DebugMsgStr ("CPort::ReadChar-有问题!");
 return FALSE;
 }

 if (WaitForSingleObject (m_ov.hEvent, nWait) !=WAIT_OBJECT_0)
 {
 DebugMsgStr ("CPort::ReadChar-没等到(足够)数据!");
 return FALSE;
 }

 }

 //返回正确
 return TRUE;
}

//从端口读 nNumToRead 个字符, 存在以 pDataAddr 为首地址的缓存中
//m_nWait 表示输入缓存中如果没有足够的数据, 等待时间的长度
BOOL CPort::ReadData (LPVOID pDataAddr, DWORD nNumToRead, UINT m_nWait)
{
 BOOL bResult;
 DWORD dwError, dwBytesRead;
 COMSTAT comstat;

 if (!m_bInit)
 {
 DebugMsgStr ("端口尚未初始化!");
 return FALSE;
 }

 //得到当前输入缓存状态
 bResult = ClearCommError (m_hComm, &dwError, &comstat);

 //如果没有数据且不等待则返回错误
 if (comstat.cbInQue==0&&m_nWait==0)
 {
 DebugMsgStr ("CPort::ReadString-没数据, 不等了!");
 }
}

```

```
 return FALSE;
}

//读字符
bResult = ReadFile(m_hComm, pDataAddr, nNumToRead, &dwBytesRead, &m_ov);

if (!bResult)
{
 //如果不是尚未完成则返回错误
 if (dwError=GetLastError() !=ERROR_IO_PENDING)
 {
 DebugMsgStr("CPort::ReadString-有问题!");
 return FALSE;
 }

 if (WaitForSingleObject(m_ov.hEvent, m_nWait) !=WAIT_OBJECT_0)
 {
 DebugMsgStr("CPort::ReadString-没等到(足够)数据!");
 return FALSE;
 }
}

//返回正确
return TRUE;
}

//从端口读一个字符,如果没有则等到有为止
BOOL CPort::WaitChar(unsigned char *pCharData)
{
 DWORD pEvtMask, dwError;
 COMSTAT comstat;
 BOOL bResult;
 unsigned char cGetData;

 //端口未初始化错误
 if (!m_bInited)
 {
 DebugMsgStr("端口尚未初始化!");
 }
}
```

```
 return FALSE;
 }

 //判断输入缓存中是否已经有了字符
 bResult=ClearCommError(m_hComm, &dwError, &comstat);
 //有则读出并返回正确
 if (comstat.cbInQue!=0)
 {
 bResult=ReadChar (&cGetData);

 if (bResult)
 {
 *pCharData=cGetData;
 return TRUE;
 }
 }

 UINT nNum=0;
 //等到读出一个字符为止
 while (nNum==0)
 {
 bResult=WaitCommEvent(m_hComm, &pEvtMask, &m_ov);

 if (bResult)
 {
 //先判断是否是出现了时间差引起的错误
 bResult = ClearCommError(m_hComm, &dwError, &comstat);
 //是错误则跳过这次循环, 重新等待
 if (comstat.cbInQue==0)
 continue;
 }
 else
 dwError = GetLastError();

 //等待字符到来
 WaitForSingleObject(m_ov.hEvent, INFINITE);

 //读字符
 bResult=ReadChar (&cGetData);
```

```

 if (bResult)
 {
 //读出了则返回正确结果
 *pCharData=cGetData;
 nNum++;
 }
 else
 {
 //出了错误,继续下一次等待
 DebugMsgStr("WaitChar:read error!");
 }
}

return TRUE;
}

//清缓存
void CPort::ClearBuffer()
{
 PurgeComm(m_hComm, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT | PURGE_TXABORT);

}

//返回端口状态
BOOL CPort::Inited()
{
 return m_bInited;
}

BOOL CPort::SetMark(BOOL bMark)
{
 DCB m_dc;
 if (!GetCommState(m_hComm, &m_dc))
 {
 DebugMsgStr("试图得到 DCB 错误！");
 return FALSE;
 }

 if (bMark)
 m_dc.Parity = MARKPARITY;
 else
}

```

```

m_dcb.Parity = SPACEPARITY;

if (!SetCommState(m_hComm, &m_dcb))
{
 DebugMsgStr("设置端口 DCB 错误!");
 return FALSE;
}
return TRUE;
}

```

以上计算机上的程序原代码，是附加光盘上第九章目录下的 ex06 文件夹中的 port.h, port.cpp 等文件。

以下代码清单为数据传输协议的类，首先仍然是类的定义：

```

#include "port.h"
//data.h
class CData
{
public:
 CPort m_port;
 //定义端口变量
 BOOL senddata(char *datastr);
 //送数据函数
 BOOL recvdata();
 //测试接收数据的函数，0x11 为正确，0x88 为错误
 BOOL InitPort(UINT portnr,UINT baud,char parity,UINT databits,UINT stopbits);
 //简化的初始化函数
 BOOL outtemperature(char *datastr);
 //此函数输出设定数据结构的数据
 CData();
 //构造函数
 virtual ~CData();
 //析构函数
};

```

下面是类中各个功能函数的具体实现：

```

#include "StdAfx.h"
#include "Data.h"
#define MACHINECOMM_DATAHEAD 0x5a
//以上语句定义数据头
#define MACHINECOMM_DATAEND 0xa5
//以上语句定义数据尾
#define MACHINECOMM_OK 0x11

```

```
//以上语句定义数据传送成功标志
#define MACHINECOMM_FAIL 0x88
//以上语句定义数据传送失败标志

CData::CData()
{
 //初始化数据
}

CData::~CData()
{
}

BOOL CData::InitPort(UINT portnr,UINT baud,char parity,UINT databits,UINT stopbits)
{
 return m_port.InitPort(portnr,baud,parity,databits,stopbits,EV_RXCHAR,256);
}

BOOL CData::senddata(char *datastr)
{
 datastr[0]=MACHINECOMM_DATAHEAD;
 datastr[5]=MACHINECOMM_DATAEND;
 if(!m_port.WriteData((LPVOID)datastr,6))
 return FALSE;
 //数据传送失败
 return TRUE;
 //数据传送成功
}

BOOL CData::recvdata()
{
 unsigned char retchar;
 CString ss;
 m_port.ClearBuffer();
 if(!m_port.ReadChar(&retchar))
 {
 AfxMessageBox("get error");
 return FALSE;
 }
 //数据读取失败;
```

```

if (retchar==MACHINECOMM_OK)
 return TRUE;
//数据传送成功;
ss.Format("Received Data:%d", retchar);
AfxMessageBox(ss);
return FALSE;//}

//所收到的返回数据不是正确的返回值, 数据传送失败, 显示所接收到的数据;
}

BOOL CData::outtemperature(char *datastr)
{
 for(short i_counter=1;i_counter<=10;i_counter++)
 {
 if(senddata(datastr))
 {if(recvdata())return TRUE;
 }
 }
 AfxMessageBox("Transfer Error!");
 return FALSE;
}

```

下面的代码清单则是实际的应用程序类，是一个基于对话框的应用程序，命名为 Trantest。其界面如图 8-28 所示。



图 8-28 温度控制系统的计算机端操作界面

该应用程序能够完成相应的数据控制操作：

```

// trantest.h : main header file for the TRANTEST application

#ifndef AFX_TRANTEST_H__0B49C9E6_2C62_11D5_BA8A_5254AB32728A__INCLUDED_
#define AFX_TRANTEST_H__0B49C9E6_2C62_11D5_BA8A_5254AB32728A__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

```

```
#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h" // main symbols

///////////////////////////////
// CTrantestApp:
// See trantest.cpp for the implementation of this class
//

class CTrantestApp : public CWinApp
{
public:
 CTrantestApp();

 // Overrides
 // ClassWizard generated virtual function overrides
 //{{AFX_VIRTUAL(CTrantestApp)
public:
 virtual BOOL InitInstance();
//}}AFX_VIRTUAL

 // Implementation

 //{{AFX_MSG(CTrantestApp)
 // NOTE - the ClassWizard will add and remove member functions here.
 //DO NOT EDIT what you see in these blocks of generated code !
 //}}AFX_MSG
 DECLARE_MESSAGE_MAP()
};

/////////////////////////////
//{{AFX_INSERT_LOCATION}
// Microsoft Visual C++ will insert additional declarations immediately before the previous
line.
```

```
#endif // !defined(AFX_TRANTEST_H__0B49C9E6_2C62_11D5_BA8A_5254AB32728A__INCLUDED_)

// trantest.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "trantest.h"
#include "trantestDlg.h"

#ifndef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

///////////
// CTrantestApp

BEGIN_MESSAGE_MAP(CTrantestApp, CWinApp)
//{{AFX_MSG_MAP(CTrantestApp)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////
// CTrantestApp construction

CTrantestApp::CTrantestApp()
{
 // Place all significant initialization in InitInstance
}

/////////
// The one and only CTrantestApp object

CTrantestApp theApp;

/////////
// CTrantestApp initialization
```

```
BOOL CTrantestApp::InitInstance()
{
 AfxEnableControlContainer();

 // Standard initialization
 // If you are not using these features and wish to reduce the size
 // of your final executable, you should remove from the following
 // the specific initialization routines you do not need.

#ifndef _AFXDLL
 Enable3dControls(); // Call this when using MFC in a shared DLL
#else
 Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif

 CTrantestDlg dlg;
 m_pMainWnd = &dlg;
 int nResponse = dlg.DoModal();
 if (nResponse == IDOK)
 {
 // dismissed with OK
 }
 else if (nResponse == IDCANCEL)
 {
 // dismissed with Cancel
 }

 // Since the dialog has been closed, return FALSE so that we exit the
 // application, rather than start the application's message pump.
 return FALSE;
}
```

# 第 9 章 C51 单片机的工程开发实例

## 9.1 单片机系统设计方法

一个完整的单片机系统的设计是相当复杂的。硬件设计方面，设计者不仅要对微机系统本身进行设计，还要根据具体的应用添加外围设备的接口电路和驱动电路。软件设计方面，则需要根据具体硬件结构来实现单片机系统的功能。在实际的应用中，由于应用环境不同，开发者还应当考虑到温度、功率、产品体积、可靠性、抗干扰性、实时性等众多问题，并提供硬件的或软件的解决方案，以保证最终产品的可靠性，其复杂程度远比通常所说的微机系统要高。

单片机应用系统的设计应按照以下几个步骤来进行。

### 1. 总体方案设计

在这一阶段，设计者需要考察实际应用环境的需要，确定系统的整体设计方案。

首先是可行性分析，确定能否使用单片机系统达到需要的设计目标，达到目标需要的经济成本是否超出可接受的范围。

其次是对系统中的核心——单片机的选型，这涉及到应用系统本身对数据处理能力的要求，以及是否有其他方面的特殊需要（低功耗、工作温度、接口电路），如果产品需要成批生产，还要考虑市场供应和系统成本等方面的问题。

最后是对系统各项功能的划分，确认软件和硬件的分工问题。经过这一阶段的设计，设计者应该已经有比较成型的系统设计框架，对软硬件系统的分工有较明确的方案。此时，可以开始进行系统的硬件设计工作了。

### 2. 系统硬件设计

系统硬件设计阶段，设计者需要对各个模块的硬件部分进行具体设计。这部分包括单片机系统的设计，外围功能模块的选择，I/O 口的分配，单片机与外围模块，单片机与单片机之间通信线路的选择，模拟输入/输出通道电路设计等方面。

当具体的硬件系统功能框图完成后，可以绘制电路的原理图，同时设计者还要对电路设计进行进一步的验证。

完成电路原理图的绘制后，还需要使用 Protel 等工具软件绘制硬件系统的 PCB 版图，这时主要是实现器件在电路板上的分布、具体的封装、信号线和电源线的走线分布等。其中需要考虑最终产品本身的尺寸要求、工作环境、干扰问题等众多方面。

然后的工作是将绘制完成的 PCB 版图交给电路板制造厂商，需要等待近 1 周的时间才能拿到完工的电路板。这段时间可以集中于单片机系统软件部分的设计，完成部分与硬件关系较小的模块的设计。

### 3. 系统软件设计

一个完整的单片机系统只有硬件还不能工作，必须有软件来控制整个系统的运行。单片机系统的软件设计主要使用汇编语言或C51语言。前者与硬件关系密切，可以方便地实现诸如中断管理以及模拟/数字量的输入/输出等功能，占用系统资源小、执行速度快，但对复杂的大型应用，其代码可读性差，不利于升级和维护。后者使用高级语言，代码效率和长度都不如汇编语言，但其结构清晰、可读性好、开发周期短、有极强的可移植性，在多数应用方面执行效率与汇编语言的差距也不大，近年来得到了极为广泛的应用。单片机的软件部分，主要的任务包括系统的初始化、各模块参数的设置、中断请求管理、定时器管理、外围模块读写、功能算法实现、可靠性和抗干扰设计等方面。

软件的设计可以分为两个阶段。首先，在等待电路板制作期间，设计者可以按照最初的设计思路完成部分的软件设计工作。随后当硬件部分的制作完成后，设计者还需要根据硬件将事先完成的软件部分的各个模块进行组合和调整。

完成系统的软件设计，首先要在计算机上进行软仿真，验证软件部分的逻辑正确性，在拿到硬件后，就可以进行实际的测试了。

### 4. 系统调试

电路板制作完成后，设计者需要按照PCB板的绘制图焊接各个元件，同时检测硬件方面的设计错误。发现问题后，如果能够补救，可以使用飞线等手段修改硬件设计，如果出现无法解决的错误，就只好推倒整个硬件设计，重新进行PCB版图的绘制等工作了。

在对硬件系统进行必要的测试后，可以使用仿真器或干脆将完成的软件部分程序烧写到硬件系统中的ROM中进行系统功能的测试。对可能出现的问题，需要从软件和硬件两个方面考虑，这一阶段需要大量的测试程序对系统的各个部分进行分别的测试，才能找到问题所在。

当软件和硬件能够很好地配合，完成预定的功能后，并不意味着单片机设计的工作已经完成，设计者还要对系统进行全面的测试，保证系统在绝大多数情况下都可以正常的工作。当这一切都完成后，设计者还应该将产品本身放到实际的工作环境中进行测试，这时往往会展露出很多原先没有考虑到的问题。

### 5. 系统完善与升级

产品设计达到预期要求后，设计者还需要最后对整个产品进行进一步的优化和组合，并在可允许的情况下为系统预留升级的接口。当所有步骤完成后，设计者可以宣布产品设计的结束，进入产品的工业生产阶段。

完整的单片机系统设计流程如图9-1所示。

```
graph TD; A[系统可行性分析]
```

图 9-1 单片机系统设计流程

## 9.2 C51 系统设计的相关知识

### 9.2.1 硬件以及电路的知识

在单片机的发展阶段，主要用汇编进行编程。因此对开发者硬件水平的要求很高，甚至在安装计算机中一些带单片机的板卡时，还曾经有过要自己动手跳线来修改配置的时候。

如今，计算机的 PnP（即插即用）模式已经成为主流设置，手动设置 IRQ 配置的时代已经一去不复返，而随着单片机系统的广泛应用，其功能不断加强，集成度日益增高，也逐渐把高级语言引入的单片机系统开发中来。

然而，单片机系统毕竟是一个与硬件密切相关的工程，完全靠软件的工作，什么具体的操作也实现不了。我们现在就所需要的一些硬件的知识向读者提供一些建议。在下面介绍的一些例子中，读者应具备如下的硬件方面的常识：

#### 1. 电路元器件方面

常用的电路器件列举如下：

- ① 常用的电阻、电容器件；
- ② 晶体振荡器；
- ③ 三极管、二极管等简单晶体器件；
- ④ 运算放大器；
- ⑤ 一些数字逻辑门，如与门、或门等。

## 2. 模拟电路方面

读者应当能够了解或者能够读懂下列一些功能的电路：

- ① 简单的振荡电路；
- ② 简单的放大电路；
- ③ 对器件如电阻、电容，以及放大器的一些简单的参数运算；
- ④ 对稳压电源有一定的了解。

在这方面的参考书籍，推荐北京大学出版社出版的《电子线路原理》一书。

## 3. 数字逻辑电路方面

读者应该能够了解数字信号的传输，以及数字信号的逻辑运算等。

- ① 信号的二进制，十六进制表示；
- ② 一些触发器，如D触发器，J-K触发器等；
- ③ 简单的门电路。
- ④ 对单片机的内部工作原理要求读者有尽量多的知识。

在这方面的参考书籍，我们推荐北京大学出版社出版的《数字逻辑电路》一书。

图9-2是51系列单片机的外观图。

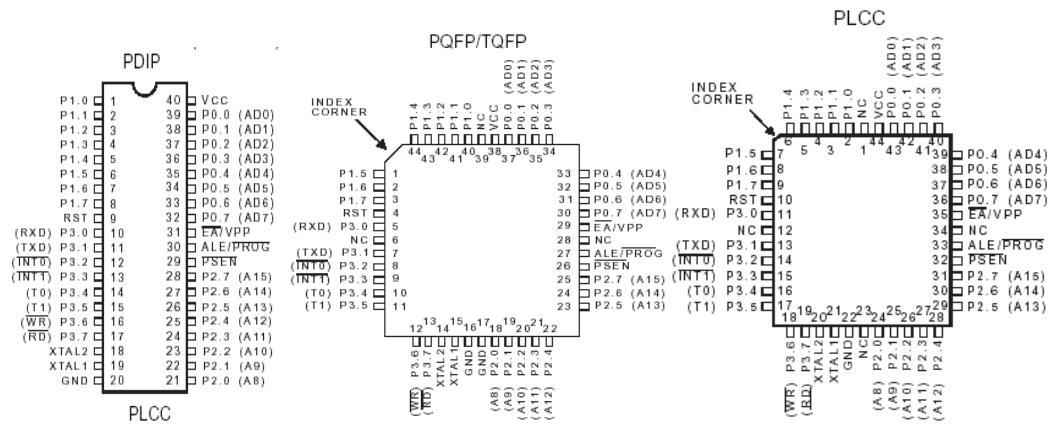


图9-2 51系列单片机几种外观

## 图9-3是51系列单片机的内部结构图。

当然，如果读者要进行单片机系统设计，那么必要的电路的设计知识也是必需具有的，由于单片机系统是数字系统，所以一般来说对电路噪声的容忍度比较高。但是事情不是绝对的，如果要涉及到一些模拟量的精确控制，如控温、控制电压等，对系统的噪声压抑要求就比较高，这时就要在系统设计，尤其是电路的设计上要多加注意。这在下面具体的例子中会看到。图9-4所示的是电路设计常用工具Protel的界面。

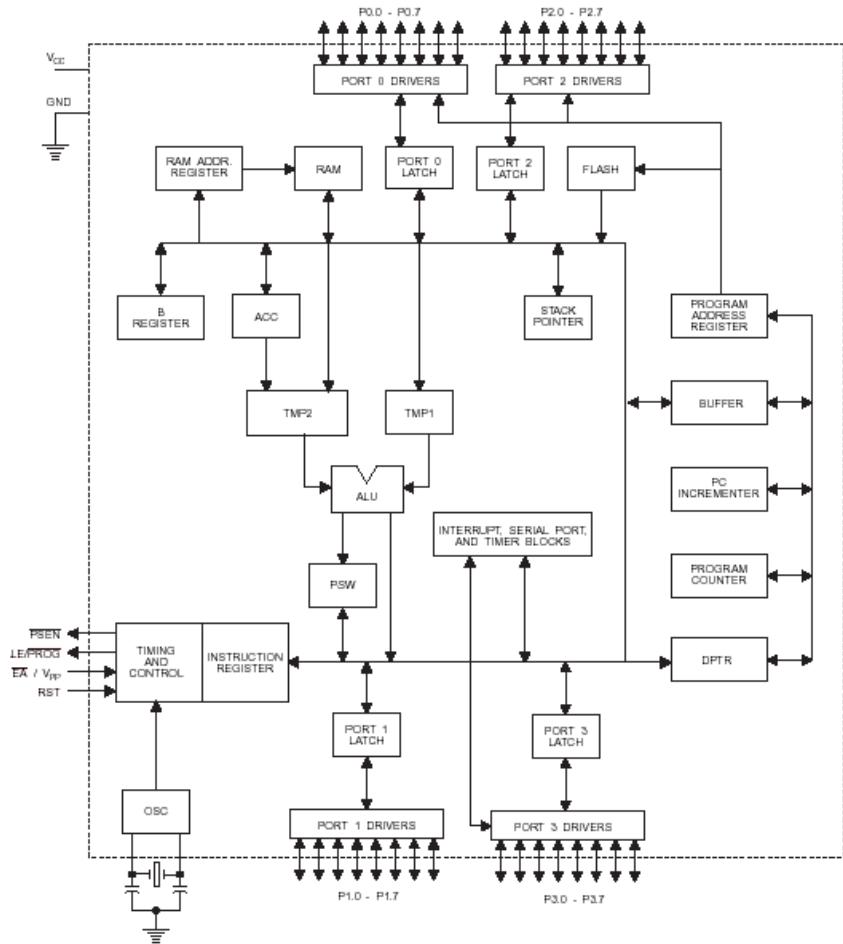


图 9-3 51 系列单片机的内部结构图

## Protel 99 SE

Complete Board-Level Design System for Windows NT/95/98

### Help Topic Areas

- Using the Design Explorer
- Managing design teams
- Customizing the Protel environment
- Design Explorer Text Editor
- Working with design objects in Protel
- Working in schematic documents
- Working in PCB documents
- Synchronizing schematic and PCB documents
- Working with programmable logic
- Working with simulations
- Working with spreadsheets and charts
- Creating and using macros in Protel

To open a help navigation pane containing a full topic listing,  
click the **Contents** button at the top of the help window.

图 9-4 电路设计常用工具 Protel 界面

### 9.2.2 软件以及编程语言的知识

读者应该能够对C语言有一个基本的了解。同时，由于系统的开发的复杂性，在进行工作的时候，有时候还会采用别的编程语言。在下面的例程中，会用到以下的编程语言：

- ① ABEL 描述语言；
- ② Visual C++高级编程语言；
- ③ ASM 汇编语言；

这些语言不是本书讲述的重点，因此在例子中我们只做简单的介绍，在注释的帮助下，读者应该能够比较轻松的理解这些代码的含义。

在进行系统开发的过程中，我们还会用到的其他的一些编程语言有：

- ④ Visual Basic 高级编程语言；
  - ⑤ VHDL 语言，主要用于单片机和大规模可编程逻辑电路联合工作的系统。
- 等等一些语言，在进行系统开发时都可能用到。

图9-5所示是Visual C++的开发界面，MedWin开发环境的界面和它非常相似。

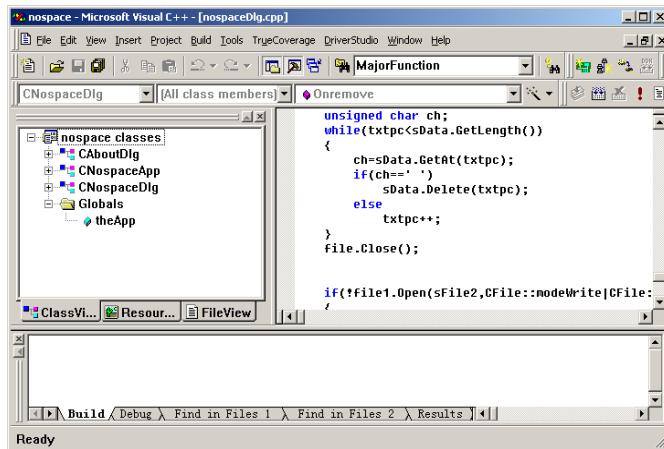


图9-5 Visual C++的开发界面

### 9.3 C51系统设计需要注意的一些问题

#### 9.3.1 单片机资源的分配

单片机系统通常是嵌入式系统。而C语言又是一种高级语言，在编程的过程中，不可避免地要产生一些冗余的信息，在这一点上，C语言跟汇编相比，有一定的差距，同时，这也是C语言在单片机系统开发中迟迟才得到运用的重要原因之一。

对C语言必须要用到的变量来说，C定义了多种变量类型，如字节型、整型、以及浮点型等。在单片机系统中，要尽量使用长度较短的变量类型，如果变量能够定义成short型的，就不要定义成int型。在单片机这种资源相对紧张的系统中，定义使用较短的变量类型应值得注意。而且这也不仅仅是资源的问题，单片机在对长变量进行运算时，通常要花比短变量运算多几倍的时间。

毫无疑问, `unsigned char` 应该是被用得最多的变量类型, 而用单片机进行 `float` 型的浮点运算则是愚不可及的。浮点运算有专门的工具, 如 DSP 芯片等。

下面谈一下堆栈。因为 8051 内部堆栈空间的限制, C51 没有像大系统那样使用调用堆栈。一般在 C 语言中调用过程时, 会把过程的参数和过程中使用的局部变量压入堆栈。为了提高效率, C51 没有提供这种堆栈而是提供一种称为压缩栈的方法, 每个调用过程可以被给定一个空间用于存放局部变量, 过程中的每个变量都存放在这个空间的固定位置, 当递归调用这个过程时会导致变量被覆盖。

而在某些实时应用中, 非再入函数是不可取的。因为函数调用时可能会被中断程序中断, 而在中断程序中可能再次调用这个函数, 所以 C51 允许将函数定义成再入函数。再入函数可被递归调用和多重调用, 但是不用担心变量被覆盖。因为每次函数调用时的局部变量都会被单独保存。

因为这些堆栈是模拟的再入函数一般都比较大, 运行起来也比较慢, 一般来说尽量少用。模拟栈不允许传递 `bit` 类型的变量, 也不能定义局部位标量。

总之, 单片机的存储资源和地址淘汰是相当有限的, 用户在开发的过程中, 要尽量充分的利用每个字节, 每个 `bit`。

### 9.3.2 单片机的寻址

我们再回顾一下单片机的寻址。

- DATA RAM 的低 128 Byte 可在一个周期内直接寻址。
- BDATA RAM 区的 16 Byte 的可位寻址区。
- IDATA RAM 区的高 128 Byte 必须采用间接寻址。
- PDATA 外部存储区的 256 Byte 通过 P0 口的地址对其寻址, 使用指令 `MOVX@Rn`, 需要两个指令周期。
- XDATA 外部存储区使用 DPTP 寻址。
- CODE 程序存储区使用 DPTP 寻址。

对 DATA 区的寻址是最快的, 所以应该把使用频率高的变量放在 DATA 区, 由于空间有限, 必须注意使用 DATA 区, 除了包含程序变量外, 还包含了堆栈和寄存器组 DATA 区的声明。

### 9.3.3 C51 函数的返回值

表 9-1 显示了函数返回值传递规则, 这在要对变量的一部分进行访问时会比较有用。

表 9-1 函数返回值传递规则

| 返回类型                                     | 使用寄存器      | 说明             |
|------------------------------------------|------------|----------------|
| Bit                                      | Carry_Flag | 单个位经由位累加器返回    |
| ( <code>unsigned</code> )char,<br>1 字节指针 | R7         | 单字节类型经由 R7 返回  |
| ( <code>unsigned</code> )int,<br>2 字节指针  | R6, R7     | 高位在 R6, 低位在 R7 |

续表

| 返回类型            | 使用寄存器 | 说明                        |
|-----------------|-------|---------------------------|
| (unsigned) long | R4~R7 | 高位在 R4, 低位在 R7            |
| float           | R4~R7 | 32 位 IEEE 格式              |
| 一般指针            | R1~R3 | 存储器类型在 R3, 高位在 R2, 低位在 R1 |

### 9.3.4 单片机的看门狗功能

大多数 51 系列单片机都有看门狗，当看门狗没有被定时清零时将引起复位，这可防止程序在遭遇到干扰时，可能会跑到不正确的代码段。设计者必须清楚看门狗的溢出时间，这样以决定在合适的时候清零看门狗。

51 系列有专门的看门狗定时器对系统频率进行分频计数，定时器溢出时将引起复位。

### 9.3.5 单片机的外设

51 系列单片机有多种外设。用户为自己特定的目的选择一款合适的单片机是很重要的。

考虑到电路板空间和成本应使外围部件尽可能的少。51 系列最多可以有 512 Byte 的 RAM 和 32kB 的 EPROM。但如果只要使用系统内置的 RAM 和 EPROM 就可以了，就不再需要外接 EPROM 和 RAM，这样就省下了 I/O 口可用来和其他器件相连。在下面的例程中我们除了在第一个例子中专门介绍和外部 EPROM 以及 RAM 的互连外，别的例子都不接外部的 EPROM 和 RAM。

当不需要扩展 I/O 口，并且程序代码较短时，使用 28 脚或者 20 脚的 51 单片机可节省不少空间，这方面的例子有芯片如 AT89C2051 等。

但很多应用需要更多的 RAM 和 EPROM 空间，这时就要用外围器件 SRAM EPROM 等。同时许多外围器件能被 51 系列的内部功能和相应的软件代替，用户在这方面可以多关注业界的发展。

### 9.3.6 单片机的功耗

在设计单片机系统时，功耗也是经常要考虑的。

系统的功耗问题一般不会太突出，但有时也需要多注意，尤其是在使用电池作为能量来源的系统更加要注意。如果处理器有很多工作要做而不能进入低功耗和空闲模式，那么应选择 3.6V 的工作电压以降低功耗。

如果有足够的空闲时间的话可以考虑关闭晶振降低功耗，或者进入低功耗模式，51 系列的单片机有很多有这种功能，典型的如 AT89C51。

晶振的选择是系统功耗要考虑的一个方面，在互相需要通信的系统中，设计者必须仔细选择晶振频率确保标准的通信数据传输率 1 200、4 800、9 600、19 200 等。用户不妨先列出可供选择的晶振所能产生的数据传输率然后根据需要的数据传输率和系统要求选择晶振。

晶振当晶振频率超过 20MHz 时，必须确保总线上的其他器件能够在这种频率下工作。一般 EPROM SRAM 高速 CMOS 版的锁存器都支持 51 的工作频率，当工作频率增加时功耗也会增加这点在使用电池作为电源的系统中应充分考虑。

## 9.4 有关 C51 的一些问题

1. 如何将一个 INT 型数据转换成 2 个 CHAR 型数据？

可以用如下语句：

```
char1=int1/256;
char2=int1%256;
```

或者：

```
char1=int1>>8;
char2=int1&0x00ff;
```

其中 int1 为要转换的 INT 型数据，char1, char2 为两个 CHAR 型数据，经 keil 优化后，上述两种方法效率是一样的。

2. `typedef` 和 `#define` 有何不同？

`typedef` 命名一个新的数据类型，但实际上这个新的数据类型是已经存在的，只不过是定义了一个新的名字。

`#define` 只是一个标号的定义。上述二者没有区别，例如：

```
typedef unsigned char UCHAR ;
#define unsigned char UCHAR ;
```

但是`#define` 还可以这样用：

```
#define MAX 100;
#define FUN(x) 100-(x);
#define LABEL;
```

在上面的情况下是不能用 `typedef` 定义的。这个属于 C 语言规则的范畴。

3. 如何设定 KELC51 的仿真工作频率（时钟）？

用右键单击左边的 target 1，然后在 xtal 一栏输入所需要的时钟频率。

4. `SWITCH( )` 语句中表达式能否是位变量？

可以用位变量，如下代码清单。不过 bit 型数据只有两种状态，用 if 语句完全可以实现功能且逻辑清楚。

```
void main()
{
 bit flag;
 flag=0;
 switch(flag)
 {
 case '0':{printf("0\n");break;}
 case '1':{printf("1\n");break;}
 default:break;
 }
}
```

```

 }
}

```

代码段的含义是,如果标志位 flag 中所存的函数是'1',则输入'1',否则输出'0'。

#### 5. const 常数声明是否占内存?

const 只是用来定义常量,所占用空间与定义的类型有关,如:

```
const code cstStr[] = {"abc"};
```

该定义占用代码空间,而如:

```
const char data cstStr[] = {"abc"};
```

则占用内存空间;

另外, #define 的定义不占用空间,只是做为C51编译的标志。

#### 6. char \*addr=0xc000 和 char xdata \*addr=0xc000 有何区别?

前者是通用定义,指针变量 addr 可指向任何内存空间的值,而后者则指定该指针变量只能指向 xdata 中的值。

后一种定义中该指针变量(addr)将比前者少占用一个存储字节。另外还有

```
uchar xdata *addr=0xc000;
```

指针指向外部 RAM。

如果是:

```
data uchar xdata *addr=0xc000;
```

则指针指向外部 RAM,但是指针本身存在于内部 RAM(data)中。

以此类推,可以明白:

```
idata uchar xdata *addr=0xc000;
pdata uchar xdata *addr=0xc000;
data uchar idata *addr=0xc000;
```

等语句的含义。

#### 7. while(p1\_0)的执行时间是多少?

```

while(P1_0)
{P1_0=0;}
while(!P1_0)
{P1_0=1;}

```

以上代码,在KEIL51中,仿真运行试一下,上述代码大约需要14个周期。

#### 8. KEIL51中,用什么函数可以得到奇偶校验位?

例如32位数据,将四个字节相互异或后检查 P 即可,如果担心 P 被改变,可用内嵌汇编的方式,获得奇偶校验位的代码清单如下。

```

#include <stdio.h>
#include <reg51.h>
unsigned char parity(unsigned char x) {

```

```

x^=x;
if(P) return(1);
else return(0);
}

unsigned char parity2(unsigned int x){
#pragma asm
 mov a, r7
 xrl ar6, a
#pragma endasm
 if(P) return(1);
 else return(0);
}//以上代码即为获得奇偶校验位

```

## 9.5 键盘和发光数码管显示

### 9.5.1 电路设计的背景及功能

下面我们介绍单片机的键盘电路和发光数码管电路。

我们都知道计算机有键盘鼠标等输入设备，同时也有显示器等相关显示设备，有了这些设备，用户就可以很轻松地给计算机系统输入指令，同时用户也可以很清楚地了解到计算机正处在什么样的状态。这些设备，就是计算机的人机交互的手段。

同样的，一个嵌入式系统也需要人机交互的手段，而键盘和发光数码管显示就是一个比较简单的人机交互的手段之一。键盘用来输入指令，发光数码管用来显示单片机的状态。

图 9-6 所示的是十六进制键盘，从第一行从左到右为数字 1, 2, 3, 4，第二行为 5, 6, 7, 8，第三行为 9, 0, A, B，第四行为 C, D, E, F。黄线和绿线都是金属线，每当按下一个键时，对应的黄线和绿线就会连通，这样单片机就能检测出信号。

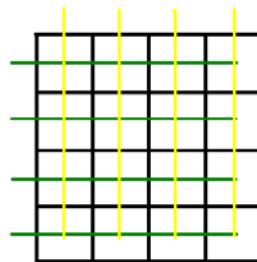


图 9-6 键盘示意图

读者可以看到，这确实是一个比较简单的电路，但它确实能够让用户输入一些指令。

### 9.5.2 电路的设计

电路图见图9-7，图9-6中的黄线（即，竖线）从左到右分别与P1.0、P1.1、P1.2、P1.3相连。绿线（即，横线）从上到下分别与P1.4、P1.5、P1.6、P1.7相连。我们在电路中给键盘留的接口元件是U2，接口封装是SIP8，这是一个比较通用的接口方式。其中标号是D0、D1、D2、D3的连接是连接的键盘中的纵向的黄线，也是键盘的驱动线；标号为K0、K1、K2、K3的连接则是连接的键盘中横向的绿线，也就是键盘的信号线。读者也能够看出来，这里的驱动线和信号线是相对的，可以互相掉换。

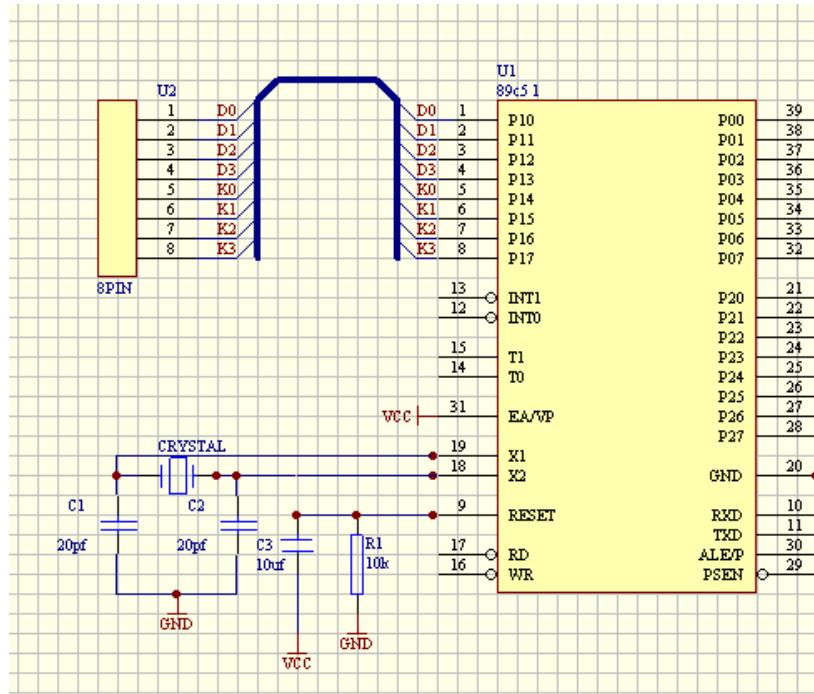


图9-7 键盘与单片机相连

### 9.5.3 键盘扫描电路的C51程序代码

当按键的时候，就可以由单片机进行键盘扫描，获得按键信号。以下代码清单是键盘扫描程序。首先是初始化代码见下面的清单：

```
#include <stdio.h>
#include <reg51.h>

#define TRUE 1
#define FALSE 0 //以上语句定义两个BOOL常量的值
#define DELAY_VALUE 3 //以上语句定义延时的值
#define PinDrvKey1 P1_0
#define PinDrvKey2 P1_1
#define PinDrvKey3 P1_2
#define PinDrvKey4 P1_3 //以上语句定义键盘扫描的驱动线
#define PinScanKey1 P1_4
#define PinScanKey2 P1_5
```

```

#define PinScanKey3 P1_6
#define PinScanKey4 P1_7 //以上语句定义键盘扫描的信号获取线
#define PinLamp1 P0_1 //以上语句定义指示灯驱动管脚
sbit P0_0=P0^0;
sbit P0_1=P0^1;
sbit P0_2=P0^2;
sbit P0_3=P0^3;
sbit P0_4=P0^4;
sbit P0_5=P0^5;
sbit P0_6=P0^6;
sbit P0_7=P0^7;

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;

sbit P2_0=P2^0;
sbit P2_1=P2^1;
sbit P2_2=P2^2;
sbit P2_3=P2^3;
sbit P2_4=P2^4;
sbit P2_5=P2^5;
sbit P2_6=P2^6;
sbit P2_7=P2^7;

void initial(void); //initial 函数为初始化子程序
void delay(short i); //delay 函数为延时子函数
unsigned char keyscan(void); //keyscan 为键盘扫描子程序

```

在主程序中，我们不做其他的特别的工作，仅仅是进行键盘扫描，将键盘扫描到的信号存储到字符变量 keyword 中去，见下面的清单：

```

void main(void)
{
 unsigned char keyword;
 initial(); //键盘驱动信号都设为低

```

```

 keyword=keyscan();
}

```

初始化程序也很简单：

```

void initial(void)
{
 P1_0=FALSE;
 P1_1=FALSE;
 P1_2=FALSE;
 P1_3=FALSE; //以上语句首先将键盘驱动信号都设为低，避免扫描线干扰
 P0_0=TRUE; //发光二极管设为不亮
}

```

下面就是键盘扫描的子程序，我们首先把键盘驱动线的第一根线 D0 置高，然后分别再检测信号线 K0~K3 是否有高电平的信号，如果有信号，那么就证明这根信号线与 D0 相交处的按键被按下了，单片机就读入这个键值。如果所有 4 根信号线都没有信号，那么，我们把 D0 置低，把 D1 置高，再一次检测信号线有没有信号。

由于键盘扫描的速度很快，而人按键总会持续一定的时间，因此，只要单片机处在等待输入的状态（如本例中我们的主程序，就一直处在等待输入状态），这个键盘扫描程序基本上不会错过任何一个按键信号。

```

unsigned char keyscan()
{
 PinDrvKey1=TRUE;
 if(PinScanKey1==TRUE)
 return 1;
 if(PinScanKey2==TRUE)
 return 5;
 if(PinScanKey1==TRUE)
 return 9;
 if(PinScanKey1==TRUE)
 return 12;
 PinDrvKey1=FALSE; //以上语句扫描第一列
 delay(DELAY_VALUE);
 PinDrvKey2=TRUE;
 if(PinScanKey1==TRUE)
 return 2;
 if(PinScanKey2==TRUE)
 return 6;
 if(PinScanKey1==TRUE)
 return 0;
 if(PinScanKey1==TRUE)
 return 13;
}

```

```

PinDrvKey2=FALSE; //以上语句扫描第二列
delay(DELAY_VALUE);
PinDrvKey3=TRUE;
if(PinScanKey1==TRUE)
 return 3;
if(PinScanKey2==TRUE)
 return 7;
if(PinScanKey1==TRUE)
 return 10;
if(PinScanKey1==TRUE)
 return 14;
PinDrvKey3=FALSE; //以上语句扫描第三列
delay(DELAY_VALUE);
PinDrvKey4=TRUE;
if(PinScanKey1==TRUE)
 return 4;
if(PinScanKey2==TRUE)
 return 8;
if(PinScanKey1==TRUE)
 return 11;
if(PinScanKey1==TRUE)
 return 15;
PinDrvKey3=FALSE; //以上语句扫描第四列
delay(DELAY_VALUE);
return 16;
}

void delay(short i)
{
 int j=0;
 int k=0;
 k=i*DELAY_VALUE;
 while (j<k) j++; //延时子程序确定了键盘扫描程序的程序获取时间。
}

```

#### 9.5.4 电路的改进 —— 键盘的消抖动程序

这样就足够了吗？实践告诉我们，这样做出来的键盘效果非常的差，要不就是扫描不进数据，要不就是重复输入很多次数据。如图 9-8 所示，一般人按键会有抖动，这是抖动信号

的原因，因此，键盘扫描会出现一些错误的信号。



图 9-8 键盘抖动信号

因此，我们需要有一个消除抖动的程序。让单片机不响应一些相关的抖动信号，而只响应一次确实存丰的按键信号。消抖动程序是这样实现的，当我们检测到一个脉冲信号时，并不立即认为是一次按键，而是延时一段时间以后再进行检测，如果三次检测都有信号，那么就认为有一次按键动作发生了。同时，在今后一段时间内的按键脉冲我们不再响应，认为这是干扰信号。

延时的选择非常重要，太快了，起不到消除抖到的效果，太慢了又让键盘太不灵活，错过较多的按键信号。程序代码见下面的清单：

```
unsigned char keyscan()
{
 PinDrvKey1=TRUE;
 if(PinScanKey1==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
 if(PinScanKey1==TRUE)
 { delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if(PinScanKey1==TRUE)
 return 1; //确认确实有按键信号，返回键值
 if(PinScanKey2==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
 if(PinScanKey2==TRUE)
 { delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if(PinScanKey2==TRUE)
 return 5; //确认确实有按键信号，返回键值
 if(PinScanKey3==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
```

```

 PinSpeaker=TRUE;
 if (PinScanKey3==TRUE)
 { delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if (PinScanKey3==TRUE)
 return 9; //确认确实有按键信号，返回键值
 if (PinScanKey4==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
 if (PinScanKey2==TRUE)
 { delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if (PinScanKey2==TRUE)
 return 12; //确认确实有按键信号，返回键值
 PinDrvKey1=FALSE; //以上语句扫描第一列
 PinDrvKey2=TRUE;
 if (PinScanKey1==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
 if (PinScanKey1==TRUE)
 { delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if (PinScanKey1==TRUE)
 return 2; //确认确实有按键信号，返回键值
 if (PinScanKey2==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
 if (PinScanKey2==TRUE)
 { delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if (PinScanKey2==TRUE)
 return 6; //确认确实有按键信号，返回键值
 if (PinScanKey3==TRUE)
 PinLamp=FALSE;

```

```
PinSpeaker=FALSE;
delay(DELAY_VALUE); //如果判断有信号，则延时一次
PinLamp=TRUE;
PinSpeaker=TRUE;
if(PinScanKey3==TRUE)
{ delay(DELAY_VALUE); //假如判断还有信号，则再延时
if(PinScanKey3==TRUE)
 return 0; //确认确实有按键信号，返回键值
if(PinScanKey4==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if(PinScanKey4==TRUE)
{ delay(DELAY_VALUE); //假如判断还有信号，则再延时
if(PinScanKey4==TRUE)
 return 13;} //确认确实有按键信号，返回键值
PinDrvKey2=FALSE; //以上语句扫描第二列
PinDrvKey3=TRUE;
if(PinScanKey1==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if(PinScanKey1==TRUE)
{ delay(DELAY_VALUE); //假如判断还有信号，则再延时
if(PinScanKey1==TRUE)
 return 3; //确认确实有按键信号，返回键值
if(PinScanKey2==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if(PinScanKey2==TRUE)
{ delay(DELAY_VALUE); //假如判断还有信号，则再延时
if(PinScanKey2==TRUE)
 return 7;} //确认确实有按键信号，返回键值
```

```

if (PinScanKey3==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if (PinScanKey3==TRUE)
{
 delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if (PinScanKey3==TRUE)
 return 10; //确认确实有按键信号，返回键值
if (PinScanKey4==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if (PinScanKey4==TRUE)
{
 delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if (PinScanKey4==TRUE)
 return 14; //确认确实有按键信号，返回键值
PinDrvKey3=FALSE; //以上语句扫描第三列
PinDrvKey4=TRUE;
if (PinScanKey4==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if (PinScanKey1==TRUE)
{
 delay(DELAY_VALUE); //假如判断还有信号，则再延时
 if (PinScanKey1==TRUE)
 return 4; //确认确实有按键信号，返回键值
if (PinScanKey2==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if (PinScanKey2==TRUE)

```

```

{ delay(DELAY_VALUE); //假如判断还有信号，则再延时
if(PinScanKey2==TRUE)
 return 8; //确认确实有按键信号，返回键值
if(PinScanKey3==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if(PinScanKey3==TRUE)
{ delay(DELAY_VALUE); //假如判断还有信号，则再延时
if(PinScanKey3==TRUE)
 return 11; //确认确实有按键信号，返回键值
if(PinScanKey4==TRUE)
 PinLamp=FALSE;
 PinSpeaker=FALSE;
 delay(DELAY_VALUE); //如果判断有信号，则延时一次
 PinLamp=TRUE;
 PinSpeaker=TRUE;
if(PinScanKey4==TRUE)
{ delay(DELAY_VALUE); //假如判断还有信号，则再延时
if(PinScanKey4==TRUE)
 return 15; //假如判断还有信号，则再延时
PinDrvKey3=FALSE; //以上语句扫描第四列
return 16;
}

```

### 9.5.5 电路的显示部分——LED 数码管电路

本节的第二部分是 LED 数码管的部分。电路如图 9-9 所示。

其中 74249 芯片是从 BCD 码到 SEG7 段码的转换器，而 74F138 则是一个地址分配器。而 LC5011 则是数码管芯片，它在选通的时候能够接受 74F138 传送过来的 SEG 七段码的数据并进行显示，而在非选通的时候，它能够保持原有的显示数据，就是说，它还是一个锁存器。

这个功能非常有用，我们就可以用一个 I/O 口来循环控制 6 个 LED 的显示，逻辑是这样的，选通一个 LED，让它显示数据，然后再不选通它，这时 LED 不会熄灭，而是继续保持原有的数据。我们可以同时去设置其他 LED 显示的数据。

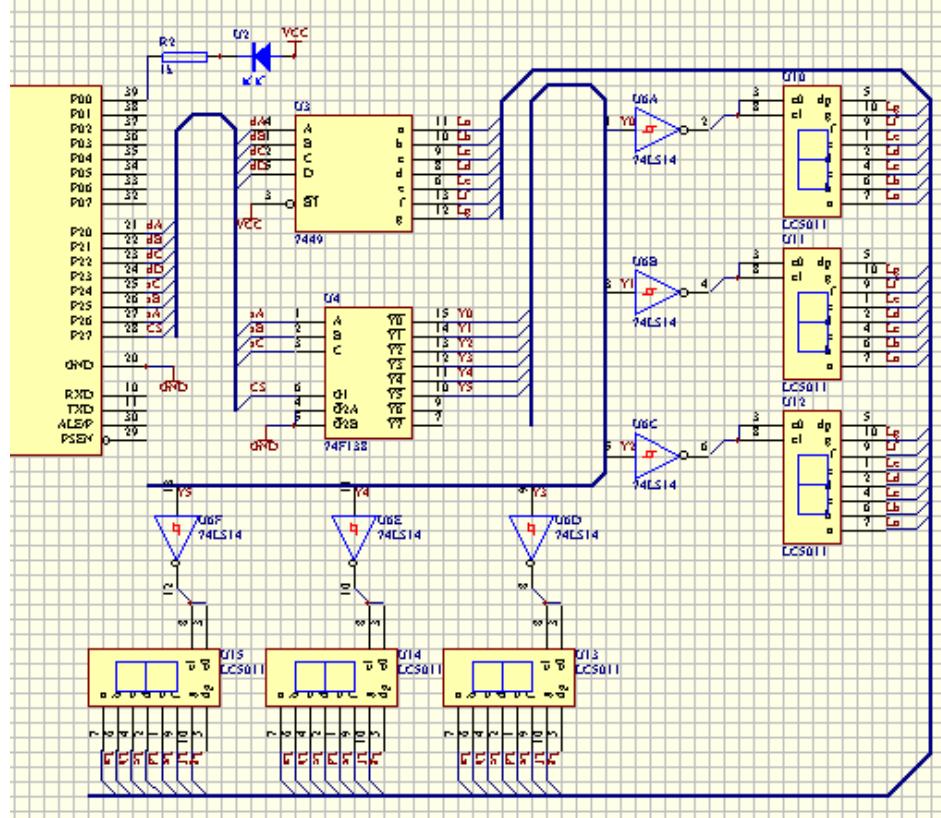


图 9-9 LED 数码管接到单片机

LED 的选通的逻辑和 74F138 的输出逻辑是反相的，所以我们需要一个反门 74LS14 连接在二者之间。

以下代码清单是让数码管分别显示 1 到 6 的数字的程序。首先是初始化代码。

```
#include <stdio.h>
#include <reg51.h>

#define TRUE 1
#define FALSE 0
#define DELAY_VALUE 3

//以上语句定义延时值
#define PinSegA P2_0
#define PinSegB P2_1
#define PinSegC P2_2
#define PinSegD P2_3

//以上语句定义七段码的数据线
#define PinSelectA P2_4
#define PinSelectB P2_5
#define PinSelectC P2_6

//以上语句定义选择器的数据位
```

```
#define PinSelectCS P2_4
//以上语句定义选择器的片选位
#define LedSelect0 0x80
#define LedSelect1 0x90
#define LedSelect2 0xa0
#define LedSelect3 0xb0
#define LedSelect4 0xc0
#define LedSelect5 0xd0
//以上语句定义 LED 驱动器选中信号数据
#define PinLamp P0_0
//以上语句定义指示灯驱动管脚
#define PinSpeaker P0_1
//以上语句定义蜂鸣器驱动管脚
sbit P0_0=P0^0;
sbit P0_1=P0^1;
sbit P0_2=P0^2;
sbit P0_3=P0^3;
sbit P0_4=P0^4;
sbit P0_5=P0^5;
sbit P0_6=P0^6;
sbit P0_7=P0^7;

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;
sbit P2_0=P2^0;
sbit P2_1=P2^1;
sbit P2_2=P2^2;
sbit P2_3=P2^3;
sbit P2_4=P2^4;
sbit P2_5=P2^5;
sbit P2_6=P2^6;
sbit P2_7=P2^7;
void initial(void);
//初始化子程序
```

```

void delay(short i);
//延时子程序
void display(unsigned char i,unsigned char num);
//显示子程序，其中变量 i, num 含义为在第 i 个数码管显示数字 num.

```

然后是主程序实现的功能也比较简单，就是循环给 LED 送出 1~6 的数进行显示。程序中调用的 Display() 子函数是显示函数。

```

void main(void)
{
unsigned char i;
initial();
//键盘驱动信号，发光二极管，以及蜂鸣器都设在无效状态
while(1)
{
for(i=0;i<=5;i++)
{
display(i, 1);
delay(DELAY_VALUE);
display(i, 8);
//所有数码管显示都为 8，只有数字 1 在各个数码管之间循环闪动
}
}
}

```

同样的，我们的程序中也少不了初始化子函数和延时子函数。

```

void initial(void)
{
P1_0=FALSE;
P1_1=FALSE;
P1_2=FALSE;
P1_3=FALSE; //键盘驱动信号都设为低，避免干扰。
P0_0=TRUE; //发光二极管设为不亮
P0_1=TRUE; //蜂鸣器不响
}

void delay(short i)
{
int j=0;
int k=0;
k=i*DELAY_VALUE;
while (j<k) j++;
}

```

```
 } //延时子程序根据设定值延时
```

下面就是我们本节的重点显示子函数。变量 i, num 含义为在第 i 个数码管显示数字 num。程序的机制是首先判断要显示的数字是不是在 0~9 之间，因为我们定义数码管只显示十进制数。判断如果通过，那么根据变量 i 的值选择数码管进行选通，然后将数值 num 送出显示。

```
void display(unsigned char i,unsigned char num)
{
 if(num >= 0 && num <=9)
 {switch(i)
 { case 0:P2= LedSelect0 | num;
 //LedSelect0 的高四位为选择第 0 个数码管的逻辑，低四位则为数码管显示的数字。
 break;
 case 1:P2= LedSelect1 | num;
 //LedSelect1 的高四位为选择第 1 个数码管的逻辑，低四位则为数码管显示的数字。
 break;
 case 2:P2= LedSelect2 | num;
 //LedSelect2 的高四位为选择第 2 个数码管的逻辑，低四位则为数码管显示的数字。
 break;
 case 3:P2= LedSelect3 | num;
 //LedSelect3 的高四位为选择第 3 个数码管的逻辑，低四位则为数码管显示的数字。
 break;
 case 4:P2= LedSelect4 | num;
 //LedSelect4 的高四位为选择第 4 个数码管的逻辑，低四位则为数码管显示的数字。
 break;
 case 5:P2= LedSelect5 | num;
 //LedSelect5 的高四位为选择第 5 个数码管的逻辑，低四位则为数码管显示的数字。
 break;
 default:break;
 }
}
}
```

## 9.6 A/D、D/A 转换器使用

### 9.6.1 电路设计的背景及功能

A/D、D/A 转换器是单片机电路经常要用到的设备。在嵌入式系统中，很多时候要处理模拟量，对模拟量进行控制。这就要用到 A/D、D/A 转换器，将模拟量转换成数字量，由单片机进行处理以后，再将数字量转换成模拟量，对自己的外围设备进行操控。

本节将介绍 12 位 A/D 转换器 AD574A 以及 12 位 D/A 转换器 AD667 的使用，并用它来控制

温度。这样的电路板共有两块，分别作为 9.10 节的多机串行通信电路的从机端，分别控制两个汽室的温度。用 D/A 转换器控制加热器的电流，然后用 A/D 的温度采样做为反馈。

### 9.6.2 电路的设计

下面的几幅图为控制电路的电路图，包含 A/D 转换器 AD574A 电路连接图（如图 9-10 所示），D/A 转换器 AD667 的电路连接图（如图 9-11 所示），带调制的 D/A 转换器的输出图（如图 9-12 所示）以及带反馈的温度控制的电路图全貌图（如图 9-13 所示）。

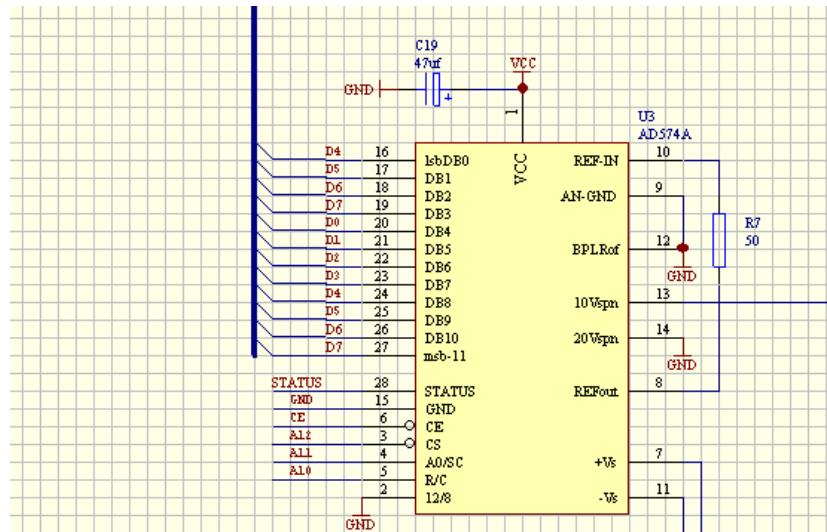


图 9-10 A/D 转换器 AD574A 电路连接图

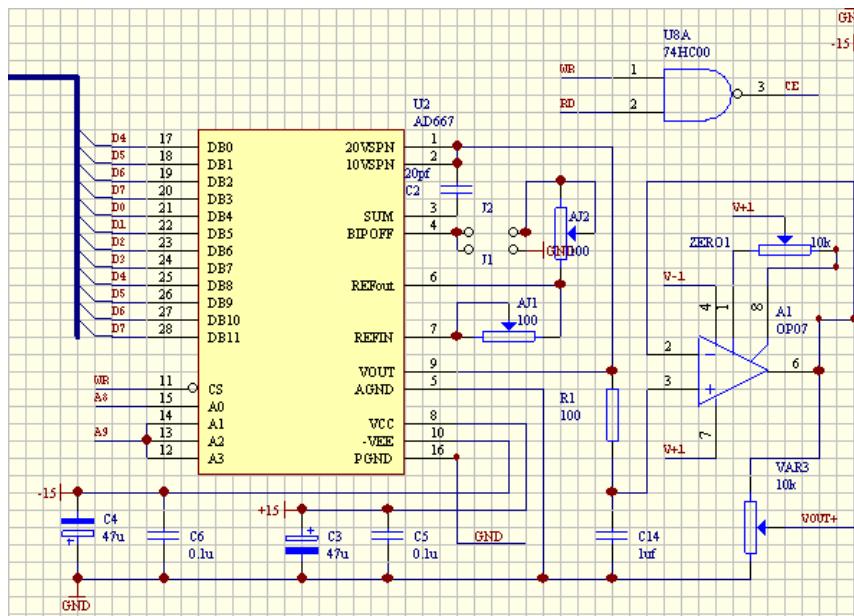


图 9-11 D/A 转换器 AD667 的电路连接图

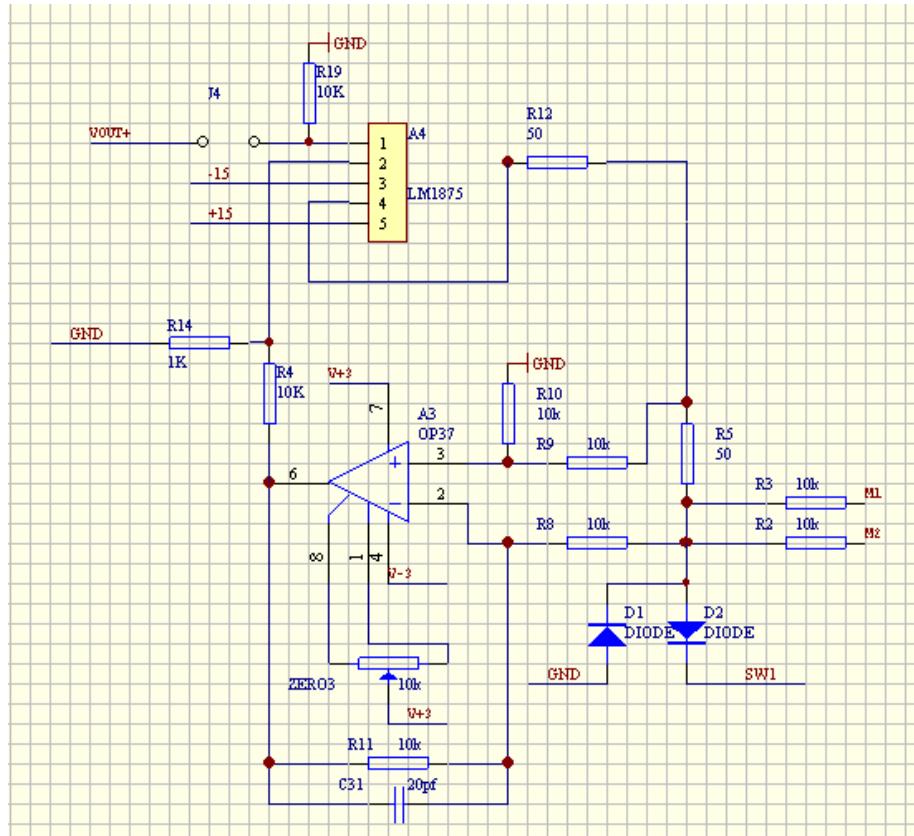


图 9-12 带调制的 D/A 转换器的输出

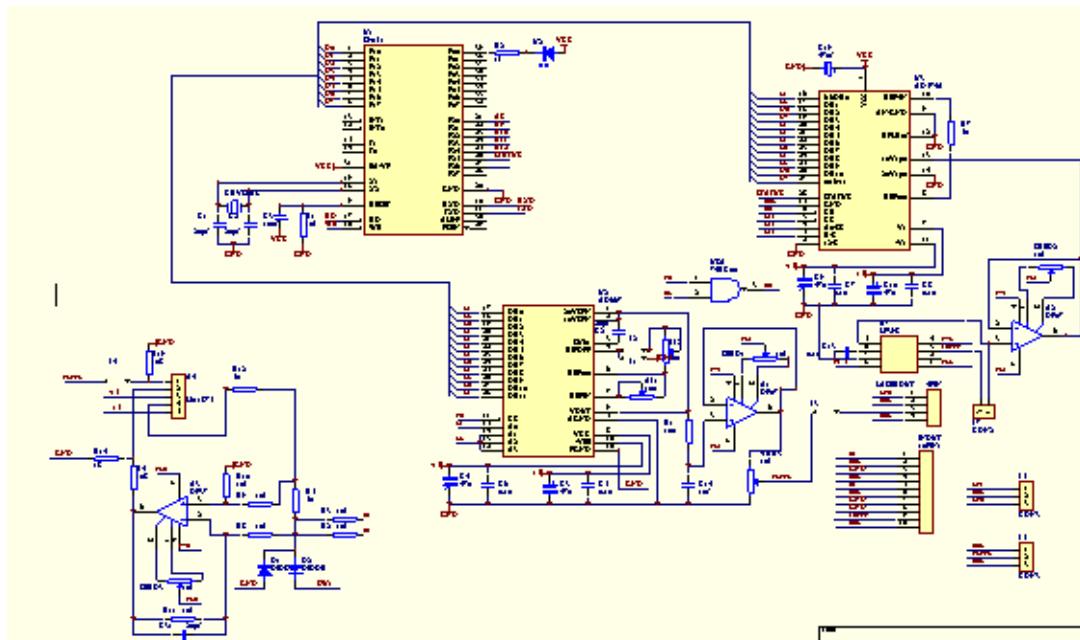


图 9-13 带反馈的温度控制的电路图全貌

我们所选用的 D/A 转换芯片是 AD667，其性能如下：

- 12 位 D/A 转换精度;
  - 双输入缓存;
  - 片内输出放大;
  - 高稳定度齐纳埋藏参考电压;
  - 低温漂, 线性温漂小于 1/2 LSB;
  - 转换时间为 3μs;
  - 供电电压为正负 12V 或正负 15V;
  - 功率为 300mW (包括参考电压源在内);
- A/D 转换芯片选用 AD574A, 其性能如下:
- 12 位 A/D 转换器;
  - 8 位或者 16 位微处理器总线接口;
  - 线性温漂标准 AD574AJ, K, L 为 0°C ~ +70°C, AD574AS, T, U 为 -55°C ~ +125°C;
  - 转换时间为 35μs。

### 9.6.3 电路的 C51 程序代码

控制代码清单如下。首先仍然是初始化代码:

```
#include <stdio.h>
#include <reg51.h>

#define TRUE 1
#define FALSE 0
#define DELAY_VALUE 3
#define ATODHIGH 0x0700
//00000111, 定义 A/D 转换器高位字节的地址, 这是根据电路图选定的
#define ATODLOW 0x0F00
//00001111, 定义 A/D 转换器低位字节的地址, 这是根据电路图选定的
#define DTOAHIGH 0x1100
//00010001, 定义 D/A 转换器高位字节的地址, 这是根据电路图选定的
#define DTOALOW 0x1200
//00010010, 定义 D/A 转换器高位字节的地址, 这是根据电路图选定的
#define ADDARATE 1.1304
//以上语句定义 A/D、D/A 转换器比例精度
#define STEPLENGTH 3
//以上语句定义控制步长, 该值要根据实际情况选择最佳值
#define PinLamp P1_0
//以上语句定义指示灯驱动管脚
```

```
#define PinSpeaker P1_1
//以上语句定义蜂鸣器驱动管脚
#define PinADStatus P1_2
//以上语句定义 A/D 转换器的状态显示
#define PinADCS P2_4
//以上语句定义 A/D 转换器的选中信号

sbit P0_0=P0^0;
sbit P0_1=P0^1;
sbit P0_2=P0^2;
sbit P0_3=P0^3;
sbit P0_4=P0^4;
sbit P0_5=P0^5;
sbit P0_6=P0^6;
sbit P0_7=P0^7;

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;

sbit P2_0=P2^0;
sbit P2_1=P2^1;
sbit P2_2=P2^2;
sbit P2_3=P2^3;
sbit P2_4=P2^4;
sbit P2_5=P2^5;
sbit P2_6=P2^6;
sbit P2_7=P2^7;

void initial(void);
//初始化子程序
void delay(unsigned char i);
//延时子程序
unsigned char *getad(void);
//获取 A/D 转换器的数据的函数
```

```

void sendda(unsigned char* a);
//设置 D/A 转换器的输出值的函数
unsigned char *serialdate(void);
//获得串口传输的信号的函数

```

初始化代码清单里包含有初始化子函数，延时子函数，从 A/D 转换器中获取信号的子函数，向 D/A 转换器输出信号的子函数。以及由 8 位数据向 16 位数据转换的 serialdate() 函数。

主函数的功能是获得我们要设定的温度值，然后从 A/D 转换器中获得温度的实际值，进行比较，然后再控制 D/A 转换器驱动相应的加热或者制冷装置进行温度控制。代码如下：

```

void main(void)
{
 initial(); //调用初始化函数，使得键盘驱动信号都设为低
 unsigned char* ref, a;
 unsigned short refint, aint;
 while(1)
 {
 ref=serialdate(); //获取串口信号
 refint=(*ref)*16+(*(ref+1)); //将两个字节的数据转换成整数型
 a=getad();
 aint=(*a)*16+(*(a+1));
 if(aint>refint) //将参考值与实际值比较
 {
 aint-=STEPLength;
 *a=(unsigned char)aint/16;
 *(a+1)=(unsigned char)aint mod 16;//如果实际值比参考值高，则降低 D/A 输出值
 }
 else if(aint<refint)
 {
 aint+=STEPLength;
 *a=(unsigned char)aint/16;
 *(a+1)=(unsigned char)aint mod 16;// 如果实际值比参考值低，则升高 D/A 输出值
 }
 }
}

```

下面是初始化子函数以及延进子函数的代码清单：

```

void initial(void)
{

```

```

P1_0=TRUE; //发光二极管设为不亮
P1_1=TRUE; //蜂鸣器不叫
}

void delay(unsigned char i) //延迟子程序
{
 short j=0;
 short k=0;
 k=i*DELAY_VALUE;
 while (j<k) j++;
}

```

下面的子函数是对 D/A 转换器的操作：

```

void sendda(unsigned char* a)
{
 XBYTE[DTOAHIGH]=*a;
 delay(DELAY_VALUE);
 XBYTE[DTOALOW]=*(a+1);
 delay(DELAY_VALUE);
}

```

下面的子函数是对 A/D 转换器的操作，可以看到，对 A/D 转换器，D/A 转换器的操作和对外部 ROM 的操作基本类似，这是因为转换器在设计的时候考虑到了通用性，将其逻辑控制的功能与 ROM 的操作兼容了。

在下面的代码清单中，信号 PinADStatus 是 A/D 转换器是否转换完成的标志，只有当 A/D 转换完成了，单片机才能够读入正确的数。

```

unsigned char *getad(void);
{
 unsigned char* a;
 PinADCS=TRUE; //选中 A/D 转换器
 PinADStatus=TRUE; //开始 A/D 转换
 while(PinADStatus==TRUE); //等待 A/D 转换完成

 *a=XBYTE[ATODHIGH];

 *(a+1)=XBYTE[ATODLOW]; //获得转换输出

 return a;
}

```

下面的子函数是将 16 位数据以 8 位数据线进行整合的功能子函数。在本例中，从串口获取温度的设定值，返回给主函数作为控制的标准。

```

unsigned char *serialdate(void)
{

```

```

unsigned char* a;
*a=getserial(); //从串口获取数据。
*(a+1)=getseiral();
return a;
}

```

## 9.7 基于单片机的数字钟

### 9.7.1 电路设计的背景及功能

下面我们来设计一个 24 小时时钟。该项设计也是应用相当广泛的。在一些定时系统中，在一些天气、环境、水文监测中都会用到这个工程或者这个工程的一部分。

以下是我们设计的各个按键的作用：

- 0~9 的数字键：设置时钟的初值；
- A：开始计时；
- B：停止计时；
- C：开始/停止设置时钟的初值，仅在停止计时时起作用；
- D：时钟显示值清零，仅在停止计时时起作用；
- E：开始/停止设置时钟蜂鸣的时间，仅在停止计时时起作用；

### 9.7.2 电路的设计

我们采用如图 9-14 所示的电路，来完成一个 24 小时的时钟功能。为了让它有更多的功能，我们给其添加一个蜂鸣器，使其能够当作一个闹钟。在实际的应用中，这个蜂鸣器可以换成任何一个需要定时启动或者唤醒的装置。

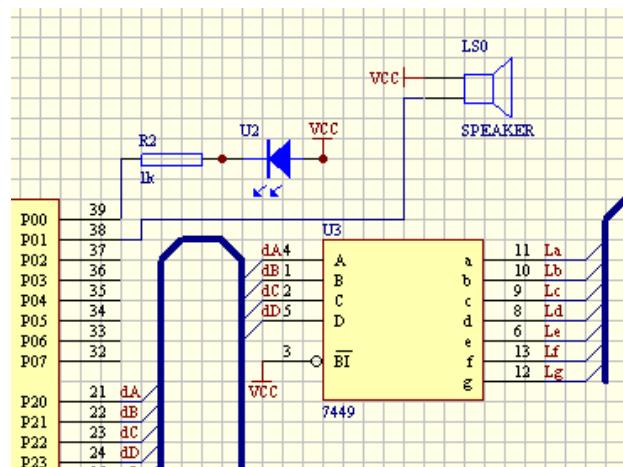


图 9-14 闹钟中的蜂鸣器

### 9.7.3 电路的C51程序代码

时钟的代码清单如下，首先是初始化代码：

```
#pragma SRC
#include <stdio.h>
#include <reg51.h>

#define TRUE 1
#define FALSE 0
#define DELAY_VALUE 3
#define DELAY_VALUE_EXTRA 10100
#define PinDrvKey1 P1_0
#define PinDrvKey2 P1_1
#define PinDrvKey3 P1_2
#define PinDrvKey4 P1_3
//以上语句定义键盘扫描的驱动线
#define PinScanKey1 P1_4
#define PinScanKey2 P1_5
#define PinScanKey3 P1_6
#define PinScanKey4 P1_7
//以上语句定义键盘扫描的信号获取线
#define PinSegA P2_0
#define PinSegB P2_1
#define PinSegC P2_2
#define PinSegD P2_3
//以上语句定义七段码的数据线
#define PinSelectA P2_4
#define PinSelectB P2_5
#define PinSelectC P2_6
//以上语句定义选择器的数据位
#define PinSelectCS P2_4
//以上语句定义选择器的片选位
#define LedSelect0 0x80
#define LedSelect1 0x90
#define LedSelect2 0xa0
#define LedSelect3 0xb0
#define LedSelect4 0xc0
#define LedSelect5 0xd0
#define PinLamp P0_0
//以上语句定义指示灯驱动管脚
```

```
#define PinSpeaker P0_1
//以上语句定义蜂鸣器驱动管脚

sbit P0_0=P0^0;
sbit P0_1=P0^1;
sbit P0_2=P0^2;
sbit P0_3=P0^3;
sbit P0_4=P0^4;
sbit P0_5=P0^5;
sbit P0_6=P0^6;
sbit P0_7=P0^7;

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;

sbit P2_0=P2^0;
sbit P2_1=P2^1;
sbit P2_2=P2^2;
sbit P2_3=P2^3;
sbit P2_4=P2^4;
sbit P2_5=P2^5;
sbit P2_6=P2^6;
sbit P2_7=P2^7;

void initial(void);
//初始化子程序
void delay(short i);
//延时子程序
void display(unsigned char i,unsigned char num);
//显示子程序，其中变量 i, num 含义为在第 i 个数码管显示数字 num.
unsigned char keyscan(void);
//键盘扫描子程序
void updateee(void);
//LED 数码管显示的更新子程序。
```

```

unsigned char h1=0;
unsigned char h0=0;
unsigned char m1=0;
unsigned char m0=0;
unsigned char s1=0;
unsigned char s0=0;
//上述语句所声明的变量用于存储时钟的当前值

unsigned char seth1=0;
unsigned char seth0=0;
unsigned char setm1=0;
unsigned char setm0=0;
unsigned char sets1=0;
unsigned char sets0=0;
//上述语句所声明的变量设定闹钟的闹铃时间,设定为闹钟时间,闹铃时间定为1分钟

```

在初始化代码中, 定义并初始化了 6 个变量 h1、h0、m1、m0、s1、s0, 这 6 个变量分别的含义是十小时位, 小时位, 十分钟位, 分钟位, 十秒位, 秒位。同时还有其他 6 个变量, 为 seth1、seth0、setm1、setm0、sets1、sets0, 这 6 个变量设置的是闹钟闹铃响的时间。

在主函数中, 我们设计了单片机不同的工作状态, 分别是 0、1、2、3 四个状态, 其含义分别如下:

- 0 状态: 为计时状态;
- 1 状态: 为停止状态;
- 2 状态: 为设置初始值状态;
- 3 状态: 为设置闹铃时间的状态;

处在不同的状态下, 数码管有不同的显示状态, 单片机对键盘输入的响应也根据需要而各不相同。而在各个状态之间, 也有不同的转换逻辑。

代码如下:

```

void main(void)
{
 unsigned char status;
 unsigned char keyword;
 initial();
 //键盘驱动信号, 发光二极管, 以及蜂鸣器都设在无效状态

 for(;;)
 {
 switch(status)
 {
 case 0:
 //状态0为计时状态

```

```

keyword=keyscan();
if(keyword==16)
{
if(h1==seth1&&h0==seth0&&m1==setm1&&m0==setm0)
{
PinSpeaker=FALSE;
delay(DELAY_VALUE);
PinSpeaker=TRUE;
//如果到达闹钟定时时间，蜂鸣器响
}
else
delay(DELAY_VALUE_EXTRA);
updatee();
}
else
if(keyword==11)status=1;
//0 状态为计时状态，1 状态为停止状态
else
{
PinSpeaker=FALSE;
delay(DELAY_VALUE);
PinSpeaker=TRUE;
//输入有错，蜂鸣器响
}
break;
case 1:
//状态 1 为停止状态
keyword=keyscan();
if(keyword==10)status=0;
else if(keyword==12)status=2;
//状态 2 为设置初值状态;
else if(keyword==13){h0=0;
h1=0;
s0=0;
s1=0;
m0=0;
m1=0;} //清零
else if(keyword==14)status=3;
//状态 3 为设置闹钟时间状态;

```

```

else
{
 PinSpeaker=FALSE;
 delay(DELAY_VALUE);
 PinSpeaker=TRUE;
 //输入有错，蜂鸣器响
}
break;

case 2:
//状态2设置初始值状态
keyword=keyscan();
if(keyword==10&&s1<=5&&m1<=5&&(h1*10+h0)<24)
 status=0;
else if(keyword==11&&s1<=5&&m1<=5&&(h1*10+h0)<24)
 status=1;
else if(keyword<10&&keyword>=0)
//如果输入的是数字键，则进入设置初始值程序
{
 h1=h0;
 h0=m1;
 m1=m0;
 m0=s1;
 s1=s0;
 s0=keyword;
 updatee();
}

//输入的数字键从低位起往左移
}
else
{
 PinSpeaker=FALSE;
 delay(DELAY_VALUE);
 PinSpeaker=TRUE;
 //输入有错，蜂鸣器响
}
case 3:
//状态3为设置闹铃状态。
keyword=keyscan();
if(keyword==10&&sets1<=5&&setm1<=5&&(seth1*10+seth0)<24)
 status=0;
else if(keyword==11&&sets1<=5&&setm1<=5&&(seth1*10+seth0)<24)

```

```

 status=1;
 else if(keyword<10&&keyword>=0)
 //如果输入的是数字键，则进入设置闹铃状态。
 {
 seth1=seth0;
 seth0=setm1;
 setm1=setm0;
 setm0=setl1;
 setl1=setl0;
 setl0=keyword;
 updatee();
 //输入的数字键从低位起往左移
 }
 else
 {
 PinSpeaker=FALSE;
 delay(DELAY_VALUE);
 PinSpeaker=TRUE;
 //输入有错，蜂鸣器响
 }
 }
}
}

```

下面是初始化子函数和延时函数。注意延时子函数仍要用 ASM 汇编语言编写。

```

void initial(void)
{
 P1_0=FALSE;
 P1_1=FALSE;
 P1_2=FALSE;
 P1_3=FALSE; //键盘驱动信号都设为低，避免干扰。
 P0_0=TRUE; //发光二极管设为不亮
 P0_1=TRUE; //蜂鸣器不响
}

//延迟子程序，为精确计时，需要用汇编语言来做。R7 刚好存的是 i 的值
void delay(short i)
{
 #pragma asm

```

```

push R7
push A
MOV R7, #0FFH
DELAY: NOP
NOP
DJNZ R7, DELAY
pop A
pop R7
#pragma endasm
}

```

下面的清单则是 LED 数码管更新功能子函数。实现的功能和秒表的大致相似，但要注意的是，24 小时时钟和秒表的进位方式不尽相同，尤其是小时位，如果计数达到 24 小时，那么整个时钟都要清零。

```

void updatee()
{
 if(s0==9) //如果 10ms 位需要进位，则进入语句
 {
 s0=0;
 if(s1==5) //如果 100ms 位需要进位，则进入语句
 {
 s1=0;
 if(m0==9) //如果 1sec 位需要进位，则进入语句
 {
 m0=0;
 if(m1==5) //如果 10sec 位需要进位，则进入语句
 {
 m1=0;
 if(h0==9&&h1<=1)
 {h0=0;
 h1+=1;}
 else if(h0==3&&h1==2)
 {h0=0;
 h1=0;}
 else h0+=1; //以上语句处理小时位的进位
 }
 else m1+=1;
 }
 else m0+=1;
 }
 else s1+=1;
 }
}

```

```
 }
else s0+=1;
display(0, s00);
delay(DELAY_VALUE);
display(1, s01);
delay(DELAY_VALUE);
display(2, s0);
delay(DELAY_VALUE);
display(3, s1);
delay(DELAY_VALUE);
display(4, m0);
delay(DELAY_VALUE);
display(5, m1);
delay(DELAY_VALUE); //以上语句则在数码管上显示各个数字
}
```

# 第 10 章 C51 单片机典型模块实例

## 10.1 典型外部 ROM 和 RAM 器件的使用

### 10.1.1 实例功能

在很多的应用场合，51 单片机自身的存储器和 I/O 口资源不能满足系统设计的需要，这时就要进行系统扩展。在本例中，将结合片外 ROM 和片外 RAM 的典型芯片的应用，说明如何扩展单片机的数据存储器和程序存储器。

本例中 3 个功能模块描述如下：

- 单片机系统：扩展单片机的存储器，实现片外存储器的访问。
- 外围电路：分为 3 个内容。首先是用地址锁存器完成单片机系统总线的扩展，其次是扩展片外 ROM 器件 2764，第三是扩展片外 RAM 器件 6264。
- C51 程序：用 C51 完成对片外存储器的读写。

本例的目的在于希望读者在读完本例后，能完成相关的电路设计，并掌握如下的知识点内容：

- 了解单片机的三总线结构。
- 了解片外 ROM 的概念和典型器件。
- 掌握片外 ROM 和单片机的电路连接和编程。
- 了解片外 RAM 的概念和典型器件。
- 掌握片外 RAM 和单片机的电路连接和编程。

### 10.1.2 器件和原理

本实例中将首先介绍单片机的三总线概念和形成，随后介绍单片机系统总线的扩展。在介绍单片机系统扩展时，引入片外典型存储器件，最后给出典型的片外 ROM 和 RAM 的电路连接和使用方法。

#### 1. 单片机的三总线

##### (1) 什么是单片机的三总线？

单片机的三总线指的是数据线、地址线和控制线。单片机 CPU 所要处理的就是这 3 种不同的总线信号。

数据线：数据总线用来传送指令和数据信息。P0 口兼做数据总线 DB0~DB7。

地址线：用来指定数据存储单元的地址分配信号线。在 8051 系列中，提供了引脚 ALE，在 ALE 为有效高电平时，在 P0 口上输出的是地址信息，A7~A0。另外，P2 口可以用于输出地址高 8 位的 A15~A8，所以对外 16 位地址总线由 P2 口和 P0 口锁存器构成。

控制线：8051 系列中引脚的输出控制线，如读写信号线、PSEN、ALE 以及输入控制信号线，如 EA、RST、T0、T1 等构成了外部的控制总线。

## (2) 如何实现外部总线的扩展？

由于单片机的输入/输出引脚有限，一般的，我们采用地址锁存器进行单片机系统总线的扩展。常用的单片机地址锁存器芯片有 74LS373、8282、74LS273 等。图 10-1 所示为 74LS373 的引脚以及它们用作地址锁存器的连接方法。

74LS373 是带三态输出的 8 位锁存器。当三态门 OE 为有效低电平，使能端 G 为有效高电平时，输出跟随输入变化；当 G 端由高变低时，输出端 8 位信息被锁存，直到 G 端再次有效为止。

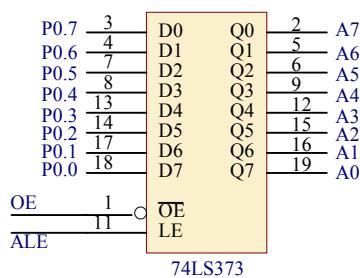


图 10-1 74LS373 的引脚

## 2. 操作片外 ROM

### (1) 什么是片外 ROM？

ROM 是程序存储器的简称，是用来存放用户程序的存储器，可分为 EPROM、OTP、ROM 和 Flash 等类，以下简单介绍各个种类。

EPROM 型存储器编程后其内容可用紫外线擦除，用户可反复使用，故特别适用于开发过程，但 EPROM 型单片机价格很高。

具有 ROM 型（掩膜型）存储器的单片机价格最低，它适用于大批量生产。由于 ROM 型单片机的代码只能由生产厂商在制造芯片时写入，故用户要更改程序代码就十分不便，在产品未成熟时选用 ROM 型单片机风险较高。

OTP 型单片机介于 EPROM 和 ROM 型单片机之间，它允许用户自己对其编程，但只能写入一次。OTP 型单片机生产量完全可由用户自己掌握，不存在 ROM 型有最小起订量和掩膜费问题，故特别受中小客户的欢迎。

Flash 型单片机允许用户使用编程工具或在线快速修改程序代码，且可反复使用，故一推出就受到广大用户的欢迎。Flash 型单片机即可用于开发过程，也可用于批量生产，随着制造工艺的改进，Flash 型单片机价格不断下降，它已是现代单片机的发展趋势。

片外的 ROM 直接挂在外部系统总线上，至于 ROM 的选通操作，需要由控制信号和片选信号确定。外部程序存储器的读信号为 PSEN。

单片机片外 ROM 芯片的种类和型号非常多。例如 27256、27128、2764、2732、27512 等。这里，以 2764 作为典型芯片加以介绍。2764 的各个功能引脚如图 10-2 所示。

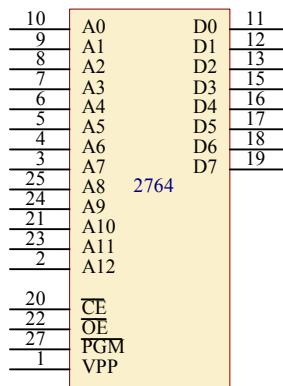


图 10-2 2764 的各个功能引脚

在这一系列的片外 ROM 中，芯片型号的高两位表示的是 EPROM，低两位数字表示存储容量的 kbit 值。例如 2764 表示的就是 64kB 个存储位的 EPROM。

在 2764 中主要有 7 种功能引脚：

- $V_{cc}$ : 电源电压，+5V。
- GND: 地。
- A0 ~ A12: 地址线。
- D0 ~ D7: 数据线。
- OE: 片输出允许，连接单片机的读信号线。
- CE: 片选信号引脚，由地址线译码器或单线选通。
- Vpp: 编程写入电压。

### (2) 单片机和片外 ROM 如何连接？

我们以单片机和 2764 为例，介绍单片机和片外 ROM 的电路连接方法，其连接如图 10-3 所示。当然，这样的连接方法也是一种典型的单片机电路，读者在设计时是可以借鉴的。

图中的 P2 口和 2764 的高 8 位地址线以及片选信号 CE 连接；P0 口经过地址锁存器输出的地址线和 2764 的低 8 位地址线相连，同时 P0 口又与 EPROM 的数据线相连。单片机的 ALE 连接地址锁存器的控制端；PSEN 连接 2764 的输出允许 OE 端。

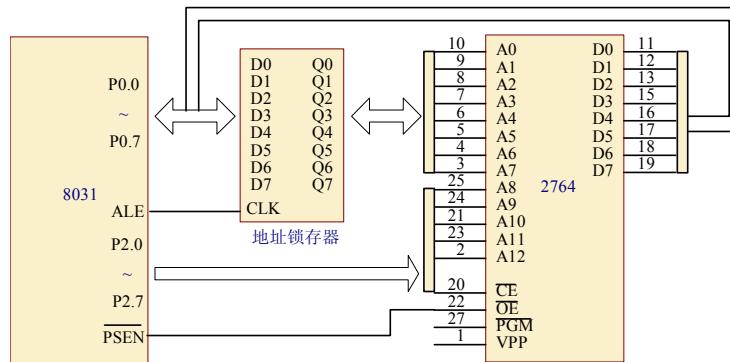


图 10-3 单片机和片外 ROM 的电路连接方法

### 3. 操作片外 RAM

#### (1) 什么是片外 RAM ?

随机存储器 (RAM) 是用来存放程序运行时的工作变量和数据的存储器。由于 RAM 的制作工艺复杂, 价格比 ROM 高得多, 所以单片机的内部 RAM 非常宝贵, 通常仅有几十到几百个字节。RAM 的内容是易失性的, 掉电后会丢失。最近出现了 EEPROM 或 Flash 型的数据存储器, 方便用户存放不经常改变的数据及其他重要信息。单片机通常还有特殊寄存器和通用寄存器, 它们是单片机中存取速度最快的存储器, 但通常存储空间很小。

单片机的读信号和外部 RAM 的输出允许信号引脚相连, 写信号和外部 RAM 的写信号相连。外部 RAM 的片选信号与外部的 I/O 端口的片选信号统一由译码产生。

可供使用的单片机片外 ROM 芯片的种类和型号非常多。例如 6116 (2k)、6264 (8k)、62256 (32k) 等。这里, 以 6264 作为典型芯片加以介绍。6264 的各个功能引脚如图 10-4 所示。

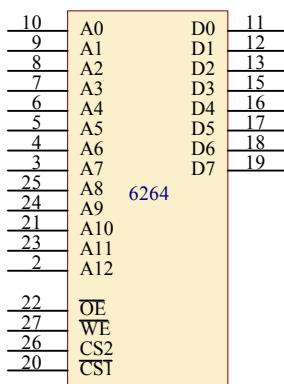


图 10-4 6264 的各个功能引脚

6264 是 8k×8 的 SRAM 芯片, 在 6264 中主要有 6 种功能引脚:

- WE: 写允许引脚, 低电平有效。
- A<sub>0</sub> ~ A<sub>12</sub>: 地址线。
- D<sub>0</sub> ~ D<sub>7</sub>: 数据线
- OE: 片输出允许, 低电平有效。
- CS<sub>1</sub>: 片选信号引脚, 低电平有效。
- CS<sub>2</sub>: 片选信号引脚, 高电平有效。

#### (2) 单片机和片外 RAM 如何连接?

我们以单片机和 6264 为例, 介绍单片机和片外 RAM 的电路连接方法, 如图 10-5 所示。和 ROM 的连接一样, 和 RAM 的连接方法也是一种典型的单片机电路, 读者在设计时是可以借鉴的。

图中的 P2 口和 6264 的高 8 位地址线以及片选信号 CE 连接, P0 口经过地址锁存器输出的地址线和 6264 的低 8 位地址线相连, 同时 P0 口又与 EPROM 的数据线相连。

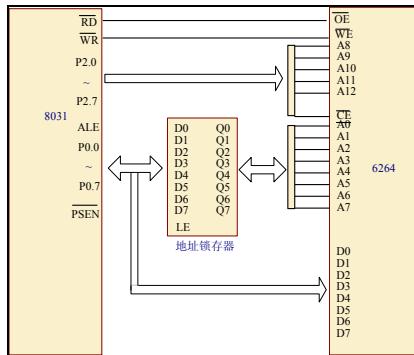


图 10-5 单片机和片外 RAM 的电路连接

### 10.1.3 电路

本例所要重点强调的就是单片机和外围典型器件的连接方法。为了更加清楚地说明 ROM 和 RAM 与单片机的连接方式，并且使得电路在实用中更加灵活，分别给出了 2764 和 6264 与单片机的连接电路，如图 10-6 和图 10-7 所示。

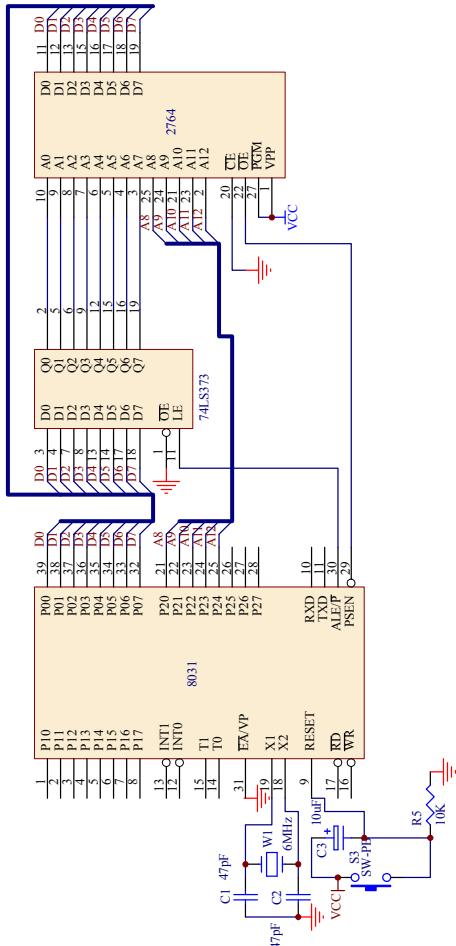


图 10-6 ROM 与单片机的连接

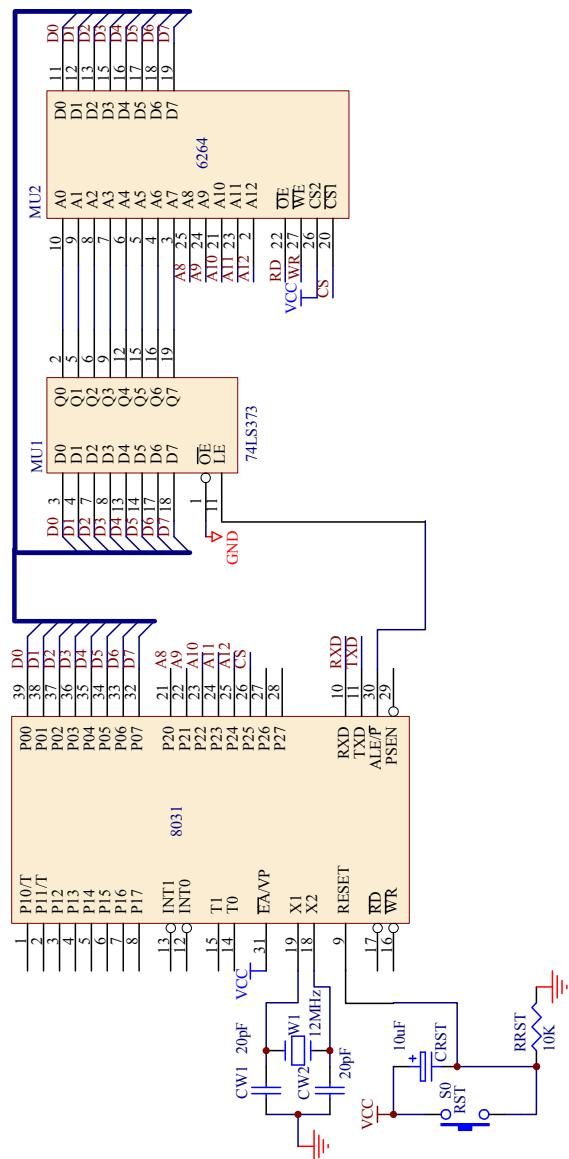


图 10-7 RAM 与单片机的连接

### 1. ROM 连接的电路原理和器件选择

在这里列出和本例相关的、关键部分的器件名称及其在电路中的主要功能。

- 8031: 单片机, 内部不含 ROM。
- OSC: 晶振, 在本例中选择 6MHz 的立式晶振。
- 74LS373: 地址锁存器, 扩展单片机的系统总线, 连接单片机和 2764。
- 2764: 8kB 的片外 EPROM。

### 2. ROM 连接的地址分配和连接

只列出和本例相关的、关键部分的单片机与各个模块管脚的连接和相关的地址分配。

- 2764 的 A0 ~ A7: 与地址锁存器的 Q0 ~ Q7 相连, 对应单片机的低 8 位地址。
- 2764 的 A8 ~ A12: 和单片机的 P2.0 ~ P2.4 相连, 是高 5 位的地址线。
- D0 ~ D7: 单片机的 8 位数据总线, 和 74LS373、2764 的 8 位数据总线相连。
- PSEN: 单片机的片外 ROM 的读信号, 和 2764 的 OE 使能端相连。
- ALE: 单片机的锁存信号线, 和 74LS373 的地址锁存允许引脚相连。

### 3. RAM 连接的电路原理和器件选择

在这里列出和本例相关的、关键部分的器件名称及其在电路中的主要功能。

- 8031: 单片机, 内部不含 ROM。
- OSC: 晶振, 在本例中选择 6MHz 的立式晶振。
- 74LS373: 地址锁存器, 扩展单片机的系统总线, 连接单片机和 2764。
- 6264: 8kB 的片外 SRAM。

### 4. RAM 连接的地址分配和连接

只列出和本例相关的、关键部分的单片机与各个模块管脚的连接和相关的地址分配。

- 6264 的 A0 ~ A7: 与地址锁存器的 Q0 ~ Q7 相连, 对应单片机的低 8 位地址。
- 6264 的 A8 ~ A12: 和单片机的 P2.0 ~ P2.4 相连, 是高 5 位的地址线。
- D0 ~ D7: 单片机的 8 位数据总线, 和 74LS373、2764 的 8 位数据总线相连。
- P2.5: 单片机对 6264 的片选信号线引脚, 对应 6264 的 CS1 引脚。
- ALE: 单片机的锁存信号线, 和 74LS373 的地址锁存允许引脚相连。

#### 10.1.4 程序设计

在本例中, 对于 C51 的编程而言并无太多的变化。前文中所写的程序完全可以在本例中在片外芯片中执行。不过, 由于变量、常量的存储类型不同, 一种是片内的, 一种是片外的, 所以在对变量的定义上要稍微作些修改。

一般的 C51 编译器是完全支持 8051 单片机的硬件结构的, 可以完全访问硬件系统所有部分。变量的存储类型和 51 单片机实际存储空间的对应关系如表 10-1 所示。

表 10-1 变量的存储类型和 51 单片机实际存储空间的对应关系

| 存储类型  | 与存储空间的对应                  |
|-------|---------------------------|
| data  | 直接寻址的片内数据存储区              |
| bdata | 可位寻址的片内数据存储区              |
| idata | 间接寻址片内存储区, 可访问片内全部 RAM 地址 |
| pdata | 分页寻址片外数据存储区               |
| xdata | 片外数据存储区                   |
| code  | 代码存储区                     |

当使用 data、bdata 定义变量时, 它们定位在单片机的片内数据存储区。这个存储区的大小根据 51 单片机的型号不同, 长度分别有 64、128、256 和 512 字节之分。

当使用 xdata 存储类型定义变量时, 它们定位在单片机的片外数据存储区, 该空间位于

本例中所附加的 6264 中，最大的寻址范围为 64kB。

由上述介绍，当我们需要定义片外 RAM 中的变量时，需要定义它们在片外的地址。例如定义片外 RAM 中的 6 个存储单元的变量 a~f，其地址为 0x0100H~0x0600H，可以采用下面的格式：

```
#define a XBYTE[0x0100]
#define b XBYTE[0x0200]
#define c XBYTE[0x0300]
#define d XBYTE[0x0400]
#define e XBYTE[0x0500]
#define f XBYTE[0x0600]
```

## 10.2 液晶显示和驱动实例

### 10.2.1 实例功能

液晶模块在便携式仪表中有着广泛的用途，如万用表、转速表等。液晶显示模块和键盘输入模块作为便携式仪表的通用器件，在单片机系统的开发过程中也可以作为常用的程序和电路模块进行整体设计。本例中使用点阵图形式液晶模块，型号为 MGLS-12032A，是内置 SED1520 控制驱动器的图形液晶显示模块。实物如图 10-8 所示。



图 10-8 点阵图形式液晶模块

在单片机系统中使用液晶显示模块作为输出器件有以下优点。

- 显示质量高

由于液晶显示器每一个点在收到信号后就一直保持那种色彩和亮度，恒定发光，而不像阴极射线管显示器（CRT）那样需要不断刷新亮点。因此，液晶显示器画质高而且不会闪烁。

- 数字式接口

液晶显示器都是数字式的，和单片机系统的接口更加简单，操纵也更加方便。

- 体积小、重量轻

液晶显示器通过显示屏上的电极控制液晶分子状态来达到显示目的，在重量上比相同显示面积的传统显示器件要轻得多。

- 功率消耗小

相比而言，液晶显示器的功耗主要消耗在其内部的电极和驱动 IC 上，因而耗电量比其他显示器件也要小得多。

本例的功能模块分为以下 3 个方面。

- 单片机系统：使用单片机控制液晶模块，并发送需要显示的内容，控制显示格式。
- 外围电路：提供液晶模块的电源以及与单片机的接口电路。
- C51 程序：通过 C51 程序实现单片机数据的液晶显示。

希望读者在读完本例后，能完成相关的电路设计，并掌握如下的知识点内容：

- 液晶模块的选择和使用。
- 单片机和液晶模块之间接口电路。
- 对液晶模块的 C51 程序设计。
- 液晶模块的电源设计。

## 10.2.2 器件和原理

### 1. 如何选择液晶模块？

液晶显示模块是一种将液晶显示器件、连接件、集成电路、PCB 线路板、背光源、结构件装配在一起的组件。英文名称叫“LCD Module”，简称“LCM”，中文一般称为“液晶显示模块”。

根据显示方式和内容的不同，液晶模块可以分为数显液晶模块、液晶点阵字符模块和点阵图形液晶模块 3 种。

数显液晶模块是一种由段型液晶显示器件与专用的集成电路组装成一体的功能部件，只能显示数字和一些标识符号。

液晶点阵字符模块是由点阵字符液晶显示器件和专用的行、列驱动器，控制器及必要的连接件、结构件装配而成的，可以显示数字和西文字符，但是不能显示图形。

点阵图形液晶模块的点阵像素连续排列，行和列在排布中均没有空隔。因此不仅可以显示字符，而且也可以显示连续、完整的图形。

本例中采用的就是点阵图形液晶模块。在选择点阵图形液晶模块时，有下述 3 种类型可供选择：

- 行、列驱动型；
- 行、列驱动-控制型；
- 行、列控制型。

在选择不同类型的模块时，需要结合实际应用的需求，表 10-2 所示为不同类型的模块驱动方式和功能上的比较。

表 10-2 液晶驱动模块比较

| 变量          | 驱动方式        | 功能                                              |
|-------------|-------------|-------------------------------------------------|
| 行、列驱动型模块    | 外接专用控制器的模块  | 模块只装配有通用的行、列驱动器，这种驱动器实际上只有对像素的一般驱动输出端           |
| 行、列驱动-控制型模块 | 依靠计算机直接控制驱动 | 块所用的列驱动器具有 I/O 总线数据接口，可以将模块直接挂在计算机的总线上，省去了专用控制器 |

续表

| 变量       | 驱动方式  | 功能                                                                          |
|----------|-------|-----------------------------------------------------------------------------|
| 行、列控制型模块 | 内藏控制器 | 控制器是液晶驱动器与计算机的接口，它以最简单的方式受控于计算机，接收并反馈计算机的各种信息，具有自己一套专用的指令，并具有自己的字符发生器 CGROM |

## 2. 液晶模块可以直接使用吗？

液晶模块的使用已经使得我们在设计单片机系统时的工作量大大减轻。但是，模块的使用也需要结合具体的设计需求。一般情况下，使用液晶模块时，还需要设计模块的电源。液晶电源电路设计的主要作用是为液晶显示模块提供工作电压。同时，液晶显示模块的电源设计也是整个系统电源设计中的重要组成部分。

一般的，液晶器件的驱动需要两种不同的电源电压，一种是 5V，另一种是-10V。液晶电源电路就是需要将输入的电压转换成这两种电压信号输出，为液晶显示模块提供工作电压。同时，液晶电源的设计也需要综合考虑整个单片机系统的供电方案。

在单片机系统中使用液晶模块，不可避免地会遇到两种甚至两种以上的电源需求，这就是电源部分要解决的关键问题。在设计具体的电源模块时要注意以下几个方面：

- 为降低系统功耗，减小仪表体积，应尽可能地选用 CMOS 器件。
- 根据容许的空间和需求的容量合理地选择电池，从互换性角度考虑应尽量选用普通电池作为电源。
- 选用的合适的电源稳压变换器件，在满足电源需求的前提下，使电源模块的外围电路简单，减小占用的空间。

## 3. 如何设计液晶模块的电源？

一般的，单片机系统如果采用电池供电，其输入电压为+3V，而 LCD 显示输出除了需要提供+5V 工作电压外，还需要提供-10V 的对比度调节电压。所以电源部分的设计要求为+3V 输入，+5V 和-10V 双电压输出。本例中介绍一种采用电压转换芯片 MAX1677 进行电源供电的方案。

MAX1677 是双电压输出升压 DC-DC 变换器，适用于需两种可调电压输出的便携式仪表。MAX1677 芯片管脚图如图 10-9 所示。

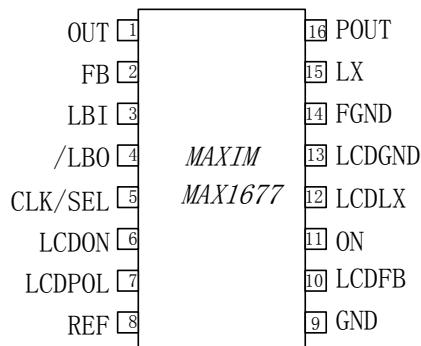


图 10-9 MAX1677 芯片管脚

MAX1677 输入电压范围 (0.7V~5.5V) 较大, 可以依据不同系统提供的电池空间和所需的不同电池电压与容量, 灵活地选择电池的种类, 1~3 节普通干电池、碱性电池、镍镉充电电池或一节锂电池均可以使系统正常工作。其主要性能如下。

允许的输入电压范围: 0.7V~5.5V。

主要输出: 2.5V~5.5V 可调电压输出, 预设值 3.3V 输出, 最大输出电流可达 350mA。

第二输出: 可为 LCD 对比度调节提供-28V~+28V 范围内的电压。

电源效率: 可达 95%。

封装形式: 16 脚 QSOP 封装, 体积很小, 不需要外部场效应管。

其他性能: 还包括 20 $\mu$ A 静态工作电流、1 $\mu$ A 关断维持电流和电池欠电压监测。

#### 4. 有典型的、能直接使用的电源电路吗?

当然有, MAX1677 就是一种专为 LCD 提供电源的芯片, MAX1677 的实际使用电路如图 10-10 所示。

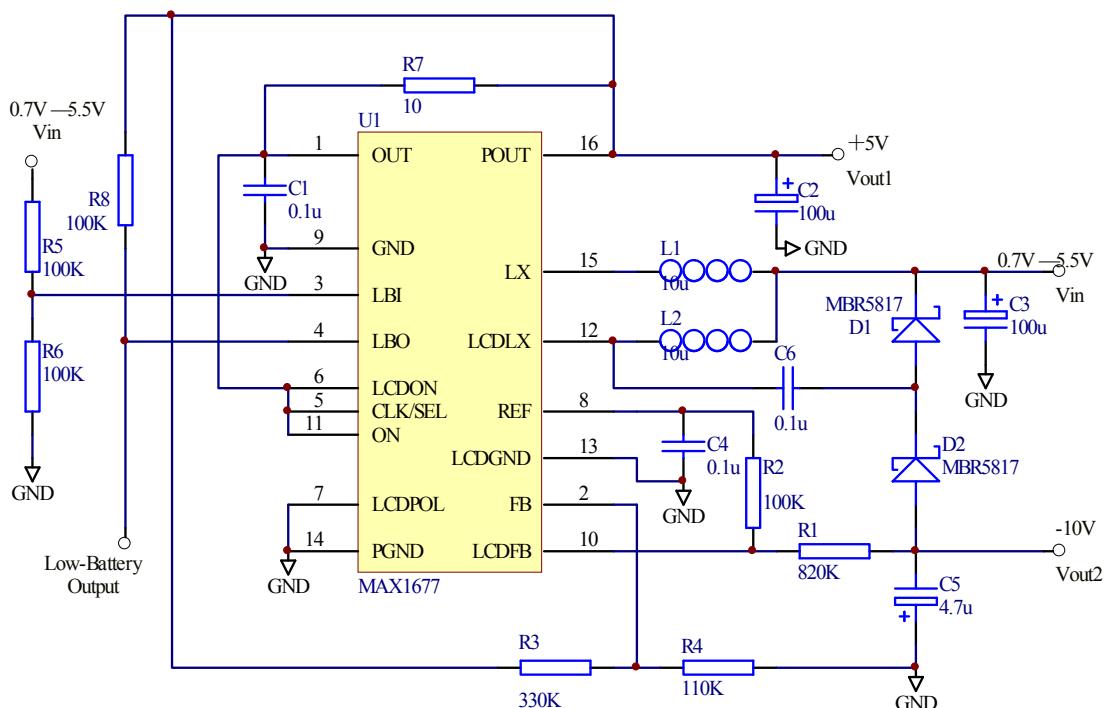


图 10-10 MAX1677 电源处理电路原理图

##### (1) 电路原理和器件选择

- MAX1677: 电压转换芯片, 输入为 3V, 输出两路电压, 分别是 +5V 和 -10V, 作为系统电源和液晶显示模块的背光电源。
- L1、L2: 磁芯电感, 选用 CoilCraft (线艺) 的 DO1608C-103 表贴磁芯电感, 电感值为 10 $\mu$ H。
- D1、D2: 肖特基二极管, 但也可选用其他型号, 只要反相耐压大于 16V 即可。
- R1、R2: 电阻, R1 和 R2 的比值决定了 LCD 对比度输出的电压值  $V_{LCD}$  (图中的  $V_{OUT2}$ ),

关系式为  $R1 = R2 \times |V_{LCD}| \div 1.25$  (V)，其中  $R1$  的取值范围为  $500k\Omega \sim 2M\Omega$ 。

- $R3$ 、 $R4$ : 电阻， $R3$  和  $R4$  的比值决定了主输出电压值  $V_{OUT}$ (对应图中的  $V_{OUT1}$ )，关系式为  $R3 = R4 \times [(V_{OUT}/1.25V) - 1]$ ，其中  $R4$  的取值范围为  $10k\Omega \sim 200k\Omega$ 。
- $R5$ 、 $R6$ : 电阻， $R5$  和  $R6$  的比值决定了系统欠电压监测的门槛电压值  $V_{TRIP}$ ，关系式为  $R5 = R6 \times [(V_{TRIP}/0.614V) - 1]$ ，其中  $R6 \leq 130k\Omega$ 。当电池电压正常时，电池电压过低，输出管脚 LBO (Low-Battery Output) 输出保持高电平；一旦电池电压低于门槛电压  $V_{TRIP}$  时，LBO 管脚输出变为低电平。如果不使用欠电压监测的话，只需将第 3 管脚 (LBI) 接地。

## (2) 地址分配和连接

- $Vin$ : 电源电路的输入端，连接两节 1.5V 的电池，形成便携式仪表的电源。
- $Vout1$ : 连接 MAX1677 的 16 管脚，输出 +5V 的电压，作为系统的电源电压。
- $Vout2$ : 连接 MAX1677 的 10 管脚，输出 -10V 的电压，作为液晶显示模块的背光电源电压。
- Low-battery Output: 连接 MAX1677 的 4 管脚，输出电源电压不足的报警信号，也就是 MAX1677 中的 LBO 的信号。

## 5. 液晶模块是如何显示的呢？

在本例中使用的是点阵图形液晶模块，型号为 MGLS-12032A，该液晶模块的接口端共 16 个管脚，各管脚的具体说明如表 10-3 所示。

表 10-3

MGLS-12032A 液晶模块接口的定义

| 序号   | 管脚符号     | 管脚名称        | 说明                          |
|------|----------|-------------|-----------------------------|
| 1    | GND      | 逻辑电源地       | -                           |
| 2    | $V_{cc}$ | 逻辑电源+5V     | -                           |
| 3    | $V_0$    | 工作负电压       | 提供对比度调节负电压                  |
| 4    | A0       | 数据/指令通道选择   | A0=0 选择数据通道；<br>A0=1 选择指令通道 |
| 5    | R/W      | 读/写选择信号     | R/W=0 写操作； R/W=1 读操作        |
| 6    | E1       | 控制器 1 的读写使能 | E1=0 禁用； E1=1 允许使用          |
| 7    | E2       | 控制器 2 的读写使能 | E2=0 禁用； E2=1 允许使用          |
| 8    | NC       | 空           | -                           |
| 9~16 | DB0~DB7  | 三态数据总线      | -                           |

MGLS-12032A 内置 SED1520 控制驱动器的图形液晶显示模块，点阵数  $120 \times 32$ ，点大小  $0.6 \times 0.425mm$ ，模块尺寸  $75.0 \times 54.0mm$ ，视频尺寸  $60 \times 26.5mm$ ，实物如图 10-8 所示。我们将以此为例介绍液晶模块的显示方法。MGLS-12032A 液晶模块是由两片 SED1520 来驱动的，两个 SED1520 都只用了其中的 60 个列驱动口，分别驱动液晶显示器的左、右半屏。其内部逻辑电路如图 10-11 所示。下面将分 3 步介绍液晶模块的驱动方法。

- 第一步：需要了解 SED1520 液晶显示控制驱动器的特性。

SED1520 液晶显示控制驱动器集行、列驱动器和控制器于一体，被广泛应用于小规模液晶显示模块，其内部的 RAM 结构如图 10-12 所示。

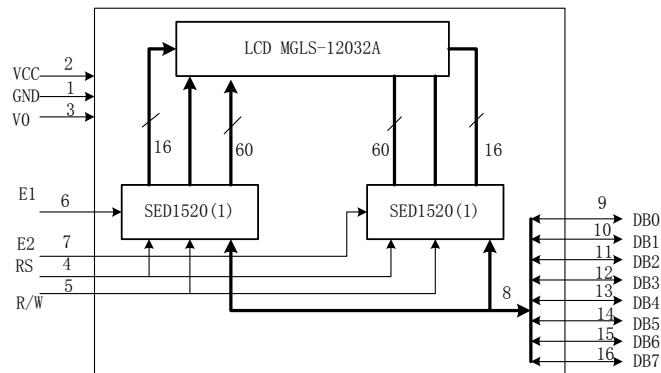


图 10-11 内部逻辑电路

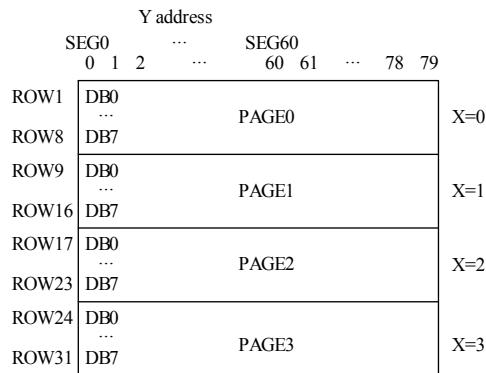


图 10-12 SED1520 显示 RAM 结构图

- 第二步：液晶模块的单片机的接口方式。

SED1520 控制器可以直接与 MCS-51 系列单片机相连，不必使用其他的接口芯片，因此选择存储器映像方式的接口，将液晶模块当作存储器的一部分对待，直接使用存储器读写进行 I/O 操作。

这种存储器映像方式的接口电路示意图如图 10-13 所示，将液晶模块的数据总线与单片机的数据总线（P0 口）直接相连，液晶模块的片选与控制引脚与单片机的高 8 位地址线（P2 口）相连，这样对液晶模块的各种指令操作，实际上就是与相应的控制地址交换数据。

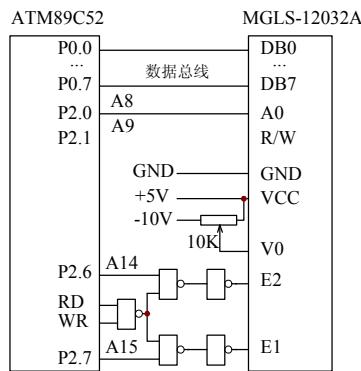


图 10-13 液晶模块与单片机的接口电路图

按照图 10-13 的连接方式，则液晶模块的各基本指令操作对应的控制地址如表 10-4 所示。

表 10-4 液晶模块控制地址的定义

| 操作    | E1 地址 | E2 地址 |
|-------|-------|-------|
| 写指令代码 | 8000H | 4000H |
| 读状态字  | 8200H | 4200H |
| 写显示数据 | 8100H | 4100H |
| 读显示数据 | 8300H | 4300H |

- 第三步：了解液晶显示一个字符的原理和方法。

要在液晶模块上显示一个字符，需要 3 个最基本的控制操作：分别向两个 SED1502 控制器写指令代码、写显示数据和读显示数据。完成这 3 项操作的前提条件是相应 SED1520 处于准备好的状态，当 SED1520 处于忙的状态时，除了读状态字指令外，其他指令均不起作用，因此在访问 SED1520 前，都要先读取控制器当前状态，判断是否准备好。

由于单片机内部 ROM 容量的限制，使用西文字符库进行显示，每个字符大小为  $6 \times 8$  点阵，以二维数组的格式存放在 ROM 中。二维数组的一行表示一个字符，行号即为字符的代码，计算公式：字符代码 = ASCII 码 - 30H；二维数组的每个元素对应各字符的每列中 8 点状态得列数据。向液晶模块输出 1 个字符的过程就是，由液晶屏显示区的指定字符行的指定列开始，连续输出该字符对应的字符库中的 6 个列数据。

MGLS-12032A 液晶模块中液晶屏显示区为  $120 \times 32$  点阵，如图 10-14 所示，每 8 个像素行组成 1 页（字符行），整个显示区共分为 4 页；显示区的左半区受 E1 控制器的 60 个列驱动器控制，右半区受 E2 控制器的 60 个列驱动器控制。

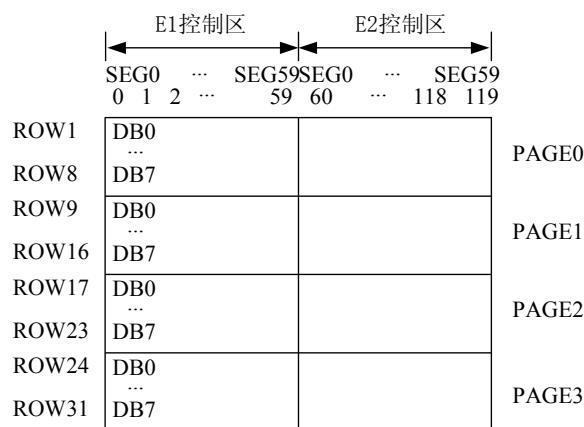


图 10-14 液晶屏显示区示意图

当字符输出的指定位置出现在 E1 和 E2 控制区的交界线附近时，如不进行适当处理的话，字符是无法被完整显示的，因此必须加入切换控制区以及换页的自动调整处理，得到完整显示的字符输出。带自动调整的单个字符输出的程序将在程序代码中详细介绍。

### 10.2.3 电路

单片机和 LCD 的接口电路如图 10-15 所示。

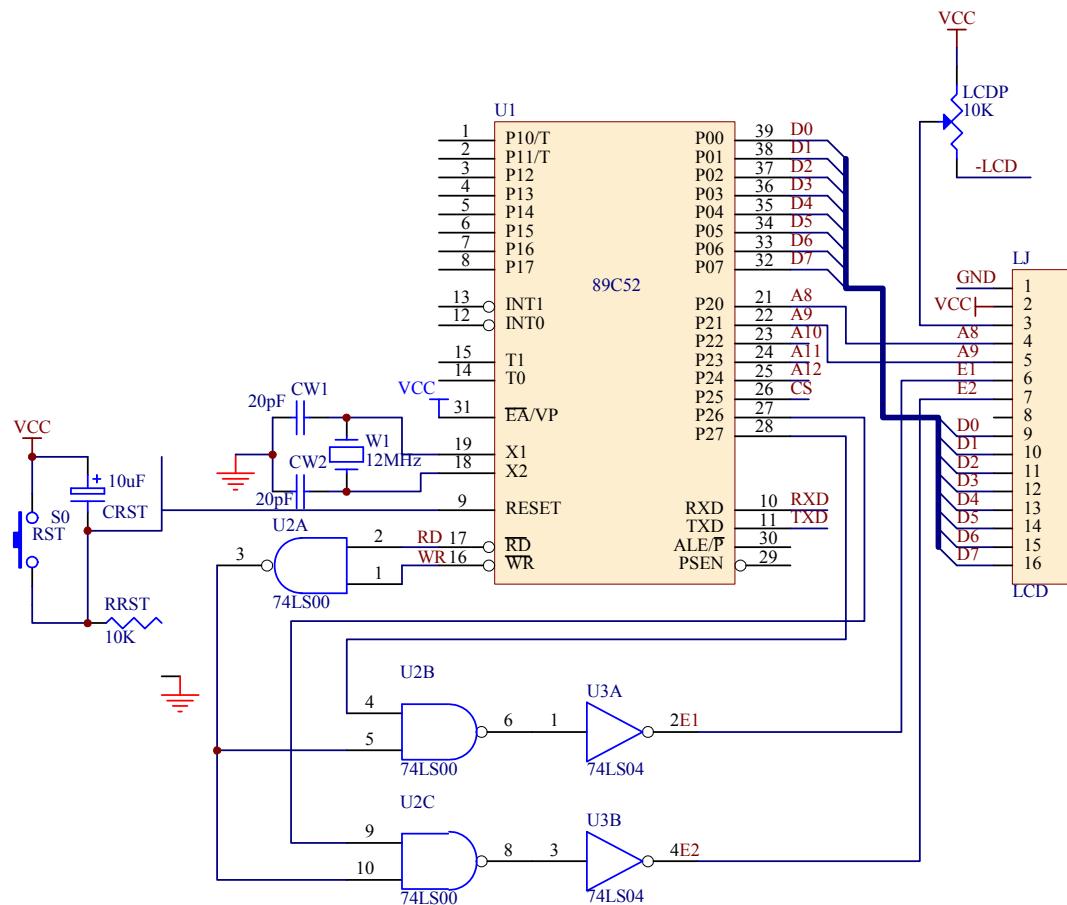


图 10-15 单片机和 LCD 的接口电路

### 1. 电路原理和器件选择

在这里仅列出单片机和 LCD 接口部分电路的器件名称和相关的主要功能。

- 89C52: LCD 的控制器, 控制字符的发送和点阵显示的时序。
- LCD: 液晶显示模块。在单片机的控制下, 按照要求的格式显示接收到的数据。
- 74LS00、74LS04: 由于 LCD 并没有独立的片选信号, 所以使用单片机的读写信号进行选通, 而 74LS00、74LS04 则是转换读写信号的电平, 同时作为片选信号。

### 2. 地址分配和连接

此处只列出和本例相关的、关键部分的单片机管脚连接和相关的地址分配。主要是单片机和 LCD 之间的信号连接和地址分配。

- E1、E2: 连接经过 74LS00、74LS04 转换后的单片机读写信号, 作为单片机对 LCD 的片选信号。
- A8: 单片机对 LCD 的数据/指令通道的选择。
- A9: 单片机对 LCD 的读写选择信号。
- D0~D7: 单片机和 LCD 的数据总线。

## 10.2.4 程序设计

本例的程序主要是完成单片机控制液晶显示模块显示输入的功能。在这里需要说明的是：对于一般的驱动模块，当用户购买的时候，会提供给用户现成的驱动程序。用户只需要根据系统的要求编写主程序，并在需要进行液晶显示的时候调用该驱动就可以了。

在提供的驱动程序中，一般是用汇编语言编写的驱动程序，而在本书中介绍的是 C51 编程，相关的 C51 和汇编的混合编程，将在专门的章节中介绍。

### 1. 程序功能

- 实现按键内容的液晶显示：液晶显示当前被使用者按下的按键内容，以及该键被按下的次数。
- 实现按键功能的液晶显示：液晶显示当前被按下的按键所对应的功能。例如，当按下启动按键时，在液晶上显示“READY”字样等。

为了实现上述的功能，需要有一系列的子程序和主程序，其中参与编译的文件有 cdwe.a51、inclr.a51、keyline.asm、disp.c 和 RESULT.c。各文件功能说明如下。

- cdwe.a51

分别向液晶显示屏的两个 SED1502 控制器写指令代码、写显示数据和读显示数据。

- inclr.a51

液晶显示屏的初始化与清屏。

- disp.c

指定字符的输出位置，并完成字符和字符串的输出。

- keyline.asm

行列键盘的驱动程序。

### 2. 主要器件和变量的说明

完成上述功能的程序中需要使用到的函数及功能如表 10-5 所示。

表 10-5

函数及功能

| 函数                            | 功能说明                              | 所在文件      | 文件类型    |
|-------------------------------|-----------------------------------|-----------|---------|
| void CWE1(void)               | 向液晶模块 E1 写控制指令                    | CDWE.A51  | 汇编源程序   |
| void DWE1(void)               | 向液晶模块 E1 写显示数据                    | -         | -       |
| void CWE2(void)               | 向液晶模块 E2 写控制指令                    | -         | -       |
| void DWE2(void)               | 向液晶模块 E2 写显示数据                    | -         | -       |
| void INITIAL(void)            | 液晶模块初始化                           | INCLR.A51 | 汇编源程序   |
| void CLEAR(void)              | 液晶模块清屏                            | -         | -       |
| void locate<br>(page, column) | 液晶显示字符定位<br>由 page 页 column 列开始显示 | DISP.C    | C51 源程序 |
| void onechar(c)               | 液晶显示输出一个字符                        | -         | -       |

续表

| 函数              | 功能说明          | 所在文件        | 文件类型    |
|-----------------|---------------|-------------|---------|
| void show(void) | 液晶显示字符串输出     | -           | -       |
| void KEY(void)  | 行列式键盘按键状态判断   | KEYLINE.ASM | 汇编源程序   |
| void main(void) | 主程序，实现完整的系统功能 | RESULT.C    | C51 源程序 |

### 3. 程序代码

- 主程序的代码，包含变量的定义和函数的调用：

```
// 定义头文件、函数和变量
#include <stdio.h>
#include <absacc.h>
#include <reg51.h>
/* define variable type */
#define uchar unsigned char
void INITIAL(void); /*液晶显示初始化*/
void CLEAR(void); /*液晶显示清屏*/
void locate(uchar page, column); /*指定首字符显示位置*/
void KEY(void); /*按键识别*/
void show(void); /*从当前位置开始显示显示缓冲区 BUFFER 中的内容*/
char data BUFFER[15]; /*显示缓冲区*/
uchar data NDIG; /*待显示的字符数*/
uchar data KEYSTATE; /*键值*/
/*存放每个键被按下的次数*/
int xdata k1=0,k2=0,k3=0,k4=0,k5=0,k6=0,k7=0,k8=0;
int xdata k9=0,k0=0,kx=0,kj=0,km1=0,km2=0,km3=0,km4=0;
int xdata sumup=0; /*测试用的变量*/

// main()主函数， 初始化变量和液晶的初始显示内容。
void main(void)
{
 uchar i;
 INITIAL(); /*显示初始化*/
 CLEAR(); /*清屏*/
 /*从 page 3, column 4 开始显示“sumup=rdy”*/
 /*固定长度的字符串显示*/
 locate(3,4);
 NDIG=sprintf(BUFFER,"sumup=rdy");
 show(); /*显示子函数*/
 sumup=12345;
```

```

/*从 page 2, column 4 开始显示“sumup=12345”，为含数值的长度不确定的字符串显示*/
locate(2,4);
NDIG=sprintf(BUFFER,"sumup=%d",sumup);
show(); //显示子函数
/*按键查询方式显示初始内容*/
for (;;)
{
 KEY(); /*按键识别*/
 while (KEYSTATE) /*有键被按下时，根据键值处理*/
 {
 switch (KEYSTATE)
 {
 case 0xb7: k0+=1; break; /*0*/
 case 0x7e: k1+=1; break; /*1*/
 case 0xbe: k2+=1; break; /*2*/
 case 0xde: k3+=1; break; /*3*/
 case 0x7d: k4+=1; break; /*4*/
 case 0xbd: k5+=1; break; /*5*/
 case 0xdd: k6+=1; break; /*6*/
 case 0x7b: k7+=1; break; /*7*/
 case 0xbb: k8+=1; break; /*8*/
 case 0xdb: k9+=1; break; /*9*/
 case 0xee: km1+=1; break; /*m1*/
 case 0xed: km2+=1; break; /*m2*/
 case 0xeb: km3+=1; break; /*m3*/
 case 0xe7: km4+=1; break; /*m4*/
 case 0x77: kx+=1; break; /****/
 case 0xd7: kj+=1; break; /*##*/
 }
 KEYSTATE=0;
 }
 locate(0,2); /*从 page 0, column 2 开始显示“k1:<被按下的次数>”*/
 NDIG=sprintf(BUFFER,"k1:%d ",k1);
 show();
 locate(0,32); /*从 page 0, column 32 开始显示“k2:<被按下的次数>”*/
 NDIG=sprintf(BUFFER,"k2:%d ",k2);
 show();
 locate(0,62); /*从 page 0, column 62 开始显示“k3:<被按下的次数>”*/
 NDIG=sprintf(BUFFER,"k3:%d ",k3);
 show();
}

```

```

locate(0,92); /*从 page 0, column 92 开始显示 “k4:<被按下的次数>”*/
NDIG=sprintf(BUFFER,"k4:%d ",k4);
show();
locate(1,2); /*从 page 1, column 2 开始显示 “k5:<被按下的次数>”*/
NDIG=sprintf(BUFFER,"k5:%d ",k5);
show();
locate(1,32); /*从 page 1, column 32 开始显示 “k6:<被按下的次数>”*/
NDIG=sprintf(BUFFER,"k6:%d ",k6);
show();
locate(1,62); /*从 page 1, column 62 开始显示 “k7:<被按下的次数>”*/
NDIG=sprintf(BUFFER,"k7:%d ",k7);
show();
locate(1,92); /*从 page 1, column 92 开始显示 “k8:<被按下的次数>”*/
NDIG=sprintf(BUFFER,"k8:%d ",k8);
show();
}
}

```

- 液晶驱动 cdwe.a51 的内容

为方便理解，在每个子函数前列出各个子函数的流程图（图 10-16 所示为 CWE1() 子函数程序的流程图，图 10-17 所示为 DWE1() 子函数程序的流程图）。

```

;定义全局变量 COM 和 DAT，全局函数 CWE1(), DWE1(), CWE2(), DWE2()
PUBLIC COM, DAT, CWE1, DWE1, CWE2, DWE2
;COM 存放控制指令，DAT 存放显示数据
RAM SEGMENT DATA
PGM SEGMENT CODE
RSEG RAM
COM: DS 1
DAT: DS 1
;指定控制单元地址
XSEG AT 08000H
CWADD1: DS 1 ;E1 写指令代码地址
XSEG AT 08200H
CRADD1: DS 1 ;E1 读状态字地址
XSEG AT 08100H
DWADD1: DS 1 ;E1 写显示数据地址
XSEG AT 4000H
CWADD2: DS 1 ;E2 写指令代码地址
XSEG AT 4200H
CRADD2: DS 1 ;E2 读状态字地址
XSEG AT 4100H

```

```
DWADD2: DS 1 ;E2 写显示数据地址
RSEG PGM
```

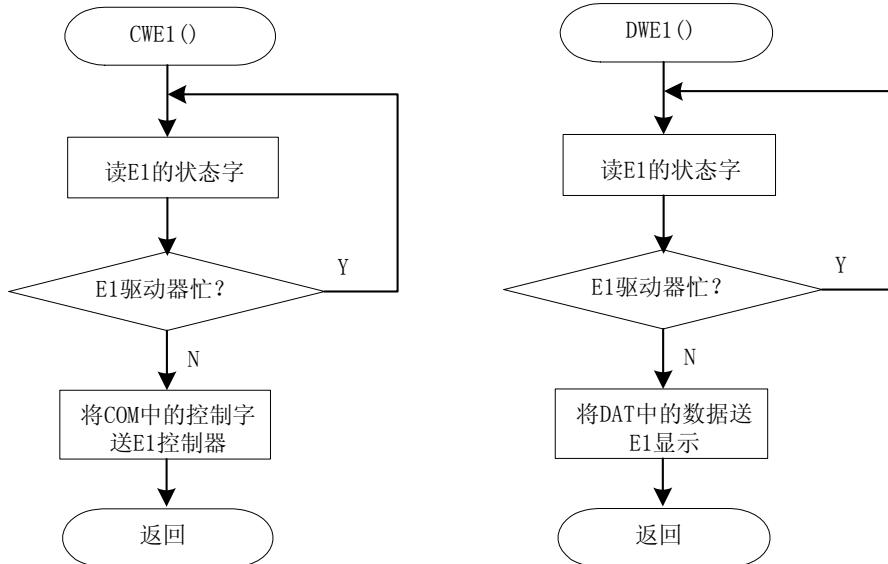


图 10-16 CWE1()子函数程序流程图

图 10-17 DWE1()子函数程序流程图

;CWE1()子函数, 传递变量 COM 到控制器 E1, 参数为#CWADD1, 实现控制指令的传递

```
CWE1: PUSH DPL
 PUSH DPH
 MOV DPTR, #CRADD1
CWE101: MOVX A, @DPTR
 JB ACC.7, CWE101
 MOV DPTR, #CWADD1
 MOV A, COM
 MOVX @DPTR, A
 POP DPH
 POP DPL
 RET
```

;DWE1()子函数, 传递变量 DAT 到控制器 E1, 参数为#CRADD1, 实现控制数据的传递

```
DWE1: PUSH DPL
 PUSH DPH
 MOV DPTR, #CRADD1
DWE101:MOVX A, @DPTR
 JB ACC.7, DWE101
 MOV DPTR, #DWADD1
 MOV A, DAT
 MOVX @DPTR, A
```

```

POP DPH
POP DPL
RET

```

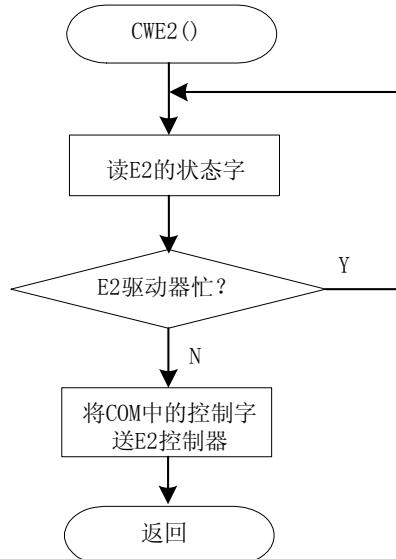


图 10-18 CWE2()子函数程序流程图

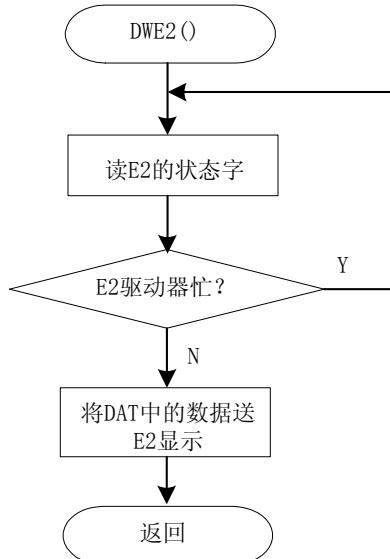


图 10-19 DWE2()子函数程序流程图

;CWE2()子函数, 传递变量 COM 到控制器 E2, 参数为#CWADD2, 实现控制指令的传递

```

CWE2: PUSH DPL
 PUSH DPH
 MOV DPTR, #CRADD2

```

```

CWE201: MOVX A, @DPTR
 JB ACC.7, CWE201
 MOV DPTR, #CWADD2
 MOV A, COM
 MOVX @DPTR, A
 POP DPH
 POP DPL
 RET

```

;DWE2()子函数, 传递变量 DAT 到控制器 E2, 参数为#CRADD2, 实现控制数据的传递

```

DWE2: PUSH DPL
 PUSH DPH
 MOV DPTR, #CRADD2

```

```

DWE201:MOVX A, @DPTR
 B ACC.7, DWE201
 MOV DPTR, #DWADD2

```

```

 MOV A, DAT
 MOVX @DPTR, A
 POP DPH
 POP DPL
 RET
 END

```

- 液晶驱动 inclr.a51 的内容

```

;说明使用到的外部函数和外部变量
EXTRN CODE(CWE1, DWE1, CWE2, DWE2)
EXTRN DATA(COM, DAT)

```

;定义全局函数 INITIAL(), CLEAR(), 对应的流程图分别为图 10-20、图 10-21

```

PUBLIC INITIAL, CLEAR
INITP SEGMENT CODE
INIROM SEGMENT CODE
RSEG INIROM

```

;定义内部变量：初始化变量数组 TABLE

```
TABLE: DB 0E2H, 0A4H, 0A9H, 0A0H, 0C0H, 0AFH
```

```
RSEG INITP
```

;INITIAL()初始化控制器 E1, E2

```

INITIAL: PUSH DPH
 PUSH DPL
 MOV R4, #0
INIT01: MOV DPTR, #TABLE
 MOV A, R4
 MOVC A, @A+DPTR
 MOV COM, A
 LCALL CWE1
 LCALL CWE2
 INC R4
 CJNE R4, #6H, INIT01
 POP DPL
 POP DPH
 RET

```

;清屏，即所有数据单元清 0

```

CLEAR: MOV R4, #00H
CLEAR1: MOV A, R4
 ORL A, #0B8H

```

```

MOV COM, A
LCALL CWE1
LCALL CWE2
MOV COM, #00H
LCALL CWE1
LCALL CWE2
MOV R3, #3CH

```

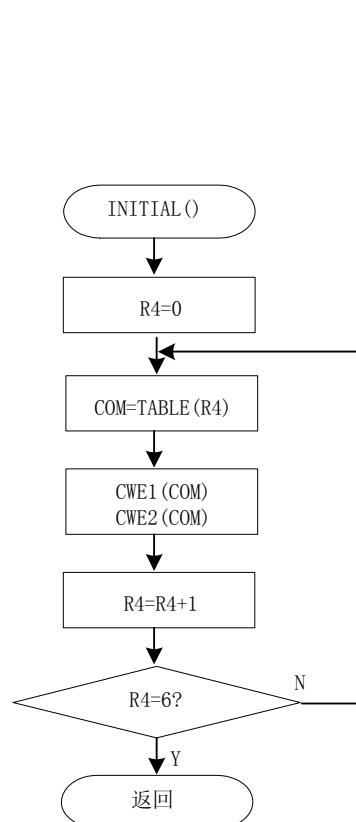


图 10-20 INITIAL()子函数程序流程图

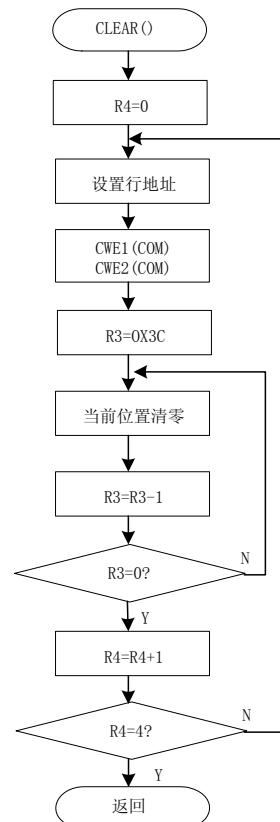


图 10-21 CLEAR()子函数程序流程图

```

CLEAR2: MOV DAT, #00H
 LCALL DWE1
 LCALL DWE2
 DJNZ R3, CLEAR2
 INC R4
 CJNE R4, #04H, CLEAR1
 RET
 END

```

- 液晶字符显示程序 disp.c 的内容

```

#define uchar unsigned char
/*定义全局变量 LINE,CLMN*/

```

```

uchar LINE, CLMN; /*LINE 当前显示页, CLMN 当前显示列*/
bit AREA; /*控制区域, AREA =0, E1 控制区; AREA =1, E2 控制区*/

/*说明使用到的外部变量*/
extern uchar COM, DAT, NDIG;
extern char xdata BUFFER[];

/*定义西文字库*/
uchar code CTAB[0x61][0x08]=
{
 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* " "=00H */
 //..... // 此处需要输入字库内容, 在液晶模块中一般会自带
};

/*说明使用到的外部函数*/
void CWE1(void); // 实现控制器 E1 控制指令的传递
void CWE2(void); // 实现控制器 E2 控制指令的传递
void DWE1(void); // 实现控制器 E1 控制数据的传递
void DWE2(void); // 实现控制器 E2 控制数据的传递

/* locate(uchar page,column)函数, 指定字符串首字符的显示位置*/
void locate(uchar page,column)
{
 page=page+column/120;
 LINE=page%4;
 COM=LINE|0xb8;
 CWE1();
 CWE2();
 column=column%120;
 CLMN=column;
 if (column>=60)
 {
 AREA=1;
 column=column-60;
 }
 else AREA=0;
 COM=column;
 if (AREA) CWE2();
 else CWE1();
}

```

```

/* onechar (char c)子函数，写一个字符，流程图见图 10-22*/
void onechar (char c)
{
 uchar i;
 for (i=0;i<6;i++)
 {
 DAT=CTAB[c-0x20][i];
 if (AREA) DWE2();
 else DWE1();
 locate(LINE, CLMN+1);
 }
}

/* show (void)子函数，显示一个字符串，流程图见图 10-23*/

```

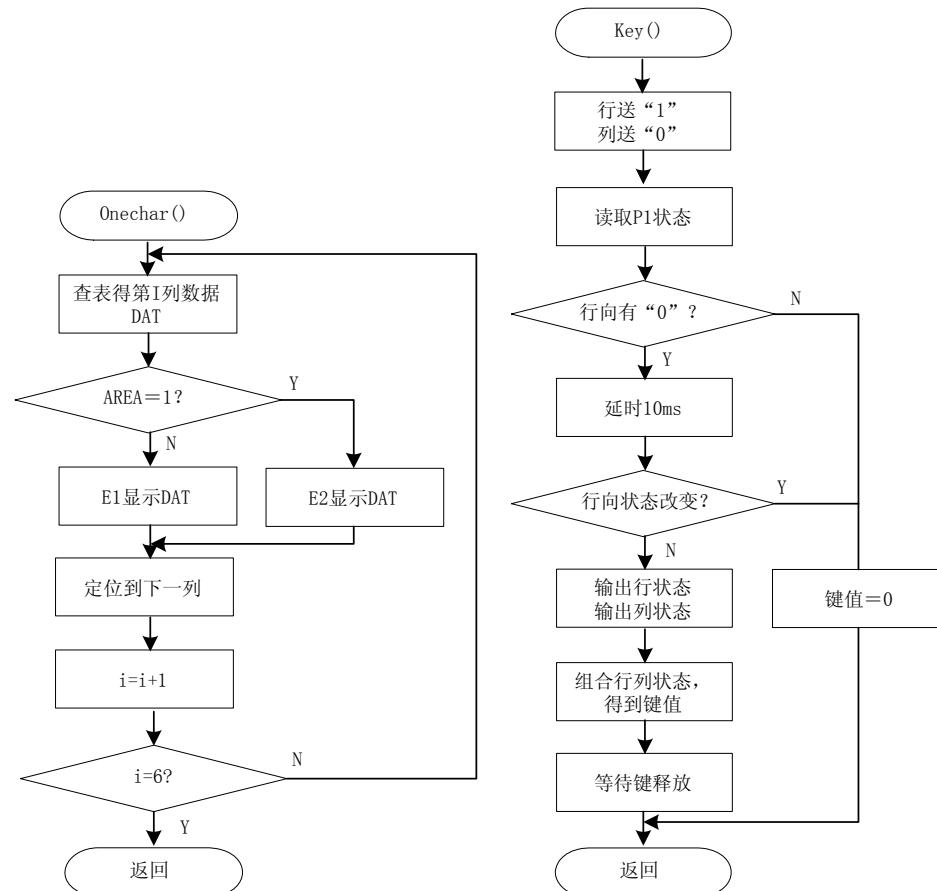


图 10-22 写一个字符子函数的程序流程图

图 10-23 按键输出函数程序流程

```

void show (void)
{

```

```

uchar i;
for (i=0;i<NDIG;i++)
{
 onechar(BUFFER[i]);
}

//行列键盘驱动函数 keyline.asm:
;说明使用到的外部变量
EXTRN DATA(KEYSTATE)
;定义全局函数 KEY(), 按键识别函数
PUBLIC KEY
KEYRAM SEGMENT DATA
KEYP SEGMENT CODE
RSEG KEYRAM
LAST: DS 1

RSEG KEYP
KEY: MOV P1, #0FH ; 输出 0000 列和 1111 行
 MOV A, P1 ; 读 P1 的状态
 ANL A, #0FH ; 从 P1 口读取行状态
 MOV LAST, A ; 保存原始行状态
 CJNE A, #0FH, PRESS ; 没有键按下
 SJMP NOKEY
PRESS: MOV R7, #100 ; 有键按下, 延时 10ms
DELAY: MOV R6, #31
 DJNZ R6, $
 DJNZ R7, DELAY
 MOV A, P1 ; 重新读取 P1 状态
 ANL A, #0FH ; 从 P1 口读取行状态 1
 CJNE A, LAST, NOKEY ; 判断哪个键按下
 ORL A, #0F0H ; .输出行状态, 并检查列状态
 MOV P1, A
 MOV A, P1
 ANL A, #0F0H
 ORL A, LAST ; 合并行、列状态
 MOV KEYSTATE, A ; 读取行列状态, 并保存到 20H 地址
 ACALL RELEASE ; 等待键松开
 SJMP KEYQUIT
NOKEY: MOV KEYSTATE, #00H ; 没有键按下 d
KEYQUIT:RET

```

```

RELEASE: MOV P1, #0FH ; 等待键松开
 MOV A,
 ANL A, #0FH
 CJNE A, #0FH, RELEASE
 RET
END

```

## 10.3 用 A/D 芯片进行电压测量

### 10.3.1 实例功能

使用 A/D（模数转换）芯片进行数据采集是单片机的一个主要应用。微型计算机的广泛应用，促进了测量仪表和测量系统的自动化、智能化。在 A/D、D/A 接口系统设计中，系统设计者的主任务是根据用户对 D/A、A/D 转换通道的技术要求，合理地选择通道的结构以及按一定的技术、经济准则，恰当地选择所需的各种集成电路，在硬件设计的同时还必须考虑通道驱动程序的设计，较好的驱动程序可以使同样规模的硬件设备发挥更高的效率。

利用 A/D 方法进行数据采集系统设计时，需要考虑 3 个方面的内容，一方面是如何针对系统的需求选择合适的 A/D 器件，二是如何根据所选择的 A/D 器件设计外围电路与单片机的接口电路，三是编写控制 A/D 器件进行数据采集的单片机程序。

本例的功能模块分为以下 3 个方面：

- 单片机系统：和外围的 A/D 器件之间进行通信，并控制数据采集过程。
- 外围电路：实现外围的 A/D 器件和单片机之间的接口电路。
- C51 程序：编写单片机控制 A/D 转换的接口程序，实现单片机的数据采集功能。

希望读者在读完本例后，能完成相关的电路设计，并掌握如下的知识点内容：

- A/D 器件的原理和选择。
- 单片机和 A/D 芯片的电路接口。
- 单片机对 A/D 转换的处理过程。
- 掌握数据采集处理过程的 C51 程序设计。

### 10.3.2 器件和原理

#### 1. 如何选择 A/D 器件？

A/D 器件和芯片是实现单片机数据采集的常用外围器件。A/D 转换器的品种繁多、性能各异，在设计数据采集系统时，首先碰到的就是如何选择合适的 A/D 转换器以满足系统设计要求的问题。选择 A/D 转换器件需要考虑器件本身的品质和应用的场合要求，基本上，可以根据以下几个方面的指标选择一个 A/D 器件。

##### (1) A/D 转换器位数

A/D 转换器位数的确定，应该从数据采集系统的静态精度和动态平滑性这两方面进行考

虑。从静态精度方面来说，要考虑输入信号的原始误差传递到输出所产生的误差，它是模拟信号数字化时产生误差的主要部分。量化误差与 A/D 转换器位数有关。一般把 8 位以下的 A/D 转换器归为低分辨率 A/D 转换器，9~12 位的称为中分辨率转换器，13 位以上的称为高分辨率转换器。10 位 A/D 芯片以下误差较大，11 位以上对减小误差并无太大贡献，但对 A/D 转换器的要求却提得过高。因此，取 10 位或 11 位是合适的。由于模拟信号先经过测量装置，再经 A/D 转换器转换后才进行处理，因此，总的误差是由测量误差和量化误差共同构成的。A/D 转换器的精度应与测量装置的精度相匹配。也就是说，一方面要求量化误差在总误差中所占的比重要小，使它不显著地扩大测量误差；另一方面必须根据目前测量装置的精度水平，对 A/D 转换器的位数提出恰当的要求。

目前，大多数测量装置的精度值不小于 0.1%~0.5%，故 A/D 转换器的精度取 0.05%~0.1% 即可，相应的二进制码为 10~11 位，加上符号位，即为 11~12 位。当有特殊的应用时，A/D 转换器要求更多的位数，这时往往可采用双精度的转换方案。

### (2) A/D 转换器的转换速率

A/D 转换器从启动转换到转换结束，输出稳定的数字量，需要一定的转换时间。转换时间的倒数就是每秒钟能完成的转换次数，称为转换速率。

确定 A/D 转换器的转换速率时，应该考虑系统的采样速率。例如，如果用转换时间为 100 $\mu$ s 的 A/D 转换器，则其转换速率为 10kHz。根据采样定理和实际需要，一个周期的波形需采 10 个样点，那么这样的 A/D 转换器最高也只能处理频率为 1kHz 的模拟信号。把转换时间减小，信号频率可提高。对一般的单片机而言，要在采样时间内完成 A/D 转换以外的工作，如读数据、再启动、存数据、循环记数等已经比较困难了。

### (3) 采样/保持器

采集直流和变化非常缓慢的模拟信号时可不用采样保持器。对于其他模拟信号一般都要加采样保持器。如果信号频率不高，A/D 转换器的转换时间短，即采用高速 A/D 时，也可不用采样/保持器。

### (4) A/D 转换器量程

A/D 转换有时需要的是双极性的，有时是单极性的。输入信号最小值有从零开始，也有从非零开始的。有的转换器提供了不同量程的引脚，只有正确使用，才能保证转换精度。在使用中，影响 A/D 转换器量程的因素如下：

- 量程变换和双极性偏置。有些 A/D 转换器如 AD574A 等，提供了两个模拟输入引脚，分别为 10V 和 20V，不同量程的输入电压可从不同引脚输入。有些 A/D 转换器还提供了双极性偏置控制，当此引脚接地时，信号为单极性输入方式；当此引脚接基准电压时，信号为双极性输入方式。
- 双基准电压。有些 A/D 转换器如 AD0809 提供了两个基准电压引脚。通常情况下，可将 REF 接地。当输入的模拟量不是从零开始，最大值也不是满量程时，就可利用这两个基准电压引脚连成对称基准电压接法。
- A/D 转换器内部比较器输入端的正确使用。有些 A/D 转换器如 ADl210，模拟输入端有两个，分别接在内部比较器的同相和反相输入端。分别使用不同的模拟输入端，输出的数字量将得到正逻辑和互补逻辑。

### (5) 满刻度误差

满度输出时对应的输入信号与理想输入信号值之差。

### (6) 线性度

实际转换器的转移函数与理想直线的最大偏移，不包括以上 3 种误差。

借鉴以上的指标，在选择 A/D 器件时还要综合考虑设计的诸项因素，如系统技术指标、成本、功耗、安装等。

## 2. 在单片机系统中如何使用 A/D 芯片？

对于采样系统而言，选择何种 A/D 芯片还需要考虑的就是 A/D 在单片机中的使用情况。单片机中的 A/D 芯片一般有 3 种：片内 A/D 芯片、片外串行总线的 A/D 芯片和片外并行总线的 A/D 芯片。以下就分别介绍这 3 种方式。

### (1) 片内 A/D 芯片

目前，大量的单片机中自带 ADC（模数转换），在这里只是简要地介绍一种带 ADC 的单片机，用以说明片内 A/D 芯片的功能及其在单片机中的使用。

美国 ATMEL 公司推出的 90 系列单片机是增强型的 8 位单片机，通称为 AVR 单片机，设计上采用低功耗 CMOS 技术，而且在软件上有效支持 C 高级语言及汇编语言。本文以 AT90S8535 为代表着重介绍其片内 A/D 转换器的特点及其在实际中的应用。

AT90S8535 带有一个 8 通道的 10 位 ADC，ADC 与一个模拟多路转换器相连，还包含一个采样保持器。ADC 有两个单独的模拟供电引脚 AVCC 和 AGND，使用时，AGND 和 GND 必须相连，AVCC 与 VCC 通过 RC 网络相连，电压差不应超过  $\pm 0.3V$ 。外部参考基准电压输入通过 AREF 引脚加入。该器件 A 口的每一引脚（PA0~PA7）均可作为 ADC 的模拟输入端，ADC 通过内部预分频器 ADCPS 保证将系统时钟频率转化为 50~200kHz 之间的 ADC 可接受的时钟频率，ADC 一般至少需要 13 个时钟周期完成一次转换，因此转换时间范围为 65~260 $\mu s$ 。ADC 为用户提供了内部中断方式的处理，可以满足实时性的要求。每次转换完成时，ADC 转换器完成中断就可以被激活。ADC 被使能后，可以选择单一转换和自由运行两种模式之一。在单一转换模式下，每次转换由用户触发；在自由运行模式下，ADC 连续取样，并更新 ADC 的数据寄存器。建议用户使用单一转换模式。若在转换过程中，选择不同的数据通道，在完成通道变化前 ADC 将结束目前的转换。ADC 产生的 10 位结果保存在数据寄存器 ADCL 和 ADCH 中，其内部特殊数据保护逻辑要求读取数据时，先读 ADCL，后读 ADCH。

### (2) 片外串行总线的 A/D 芯片

串行 A/D 芯片随着科技的发展其传输速率也可以做得很高，加上它体积小、占用单片机的端口线少，因此串行模数转换器的应用越来越广泛。在此介绍一种 16 位串行模数转换器 MAX195，它不仅速度快、精度高、功耗低，而且与单片机连接方便，编程简单。它在高精度的以单片机为核心的测控系统中是很好地选择。

MAX195 是美国 Maxim 公司推出的 16 位逐次逼近式 A/D 转换器。其主要性能包括：

- 16 位转换精度。
- 9.4 $\mu s$  转换时间。
- 内置采样保持电路。
- 三态串行数据输出。

逐次逼近寄存器 (SAR) 用以将输入模拟信号转变为 16 位二进制数码串行输出，输出时高位在前。MAX195 的数据接口包括三态输入信号 BP/UP/SHEN (悬浮为双极性输入，+5V 为单极性输入，接地为关闭模式)、转换时钟输入端 CLK、串行时钟信号 SCLK、转换结束信号/EOC、选片输入信号/CS、转换启动信号/CONV 和复位信号/RESET。CONV 变低后开始模数转换。转换结束时，经过一个转换时钟 CLK 后转换结束信号 EOC 变低。MAX195 在上电时自动校准，校准需要 14 000 个时钟周期；当 RESET 由低升高时，MAX195 启动一次校准。

MAX195 有两种转换传输方式：异步转换传输方式和同步转换传输方式。

- 异步转换传输方式

异步转换传输方式是在一次转换结束后以 SCLK 时钟频率输出。在此应用中串行数据输出与 SCLK 同步，而 CLK 仅作为转换时钟。通过查询转换结束信号 EOC 的状态或者通过在 EOC 下降沿产生的中断信号来确定一次转换的结束，然后在下一次转换开始之前将数据逐位输出。

- 同步转换传输方式

同步转换传输方式就是在 16 位模数转换进行的过程中将转换好的上一位数据位输出。它不需要 SCLK，CLK 既作为转换时钟又作为串行数据输出时钟。显然，此种转换方式可以实现最大的转换传输速度，因为前一次转换结束后，下一次转换可以紧跟着立即开始，实现不间断地连续转换。转换过程必须与转换时钟 CLK 同步，两次转换之间至少应有 4 个转换时钟 CLK 的等待时间。

### (3) 片外并行总线的 A/D 芯片

MAX197 芯片是美国 MAXIM 公司近年的新产品，是多量程 ( $\pm 10V$ 、 $\pm 5V$ 、 $0 \sim 10V$ ， $0 \sim 5V$ )、8 通道、12 位 A/D 转换器。它有标准的微机接口。三态数据 I/O 口用做 8 位数据总线，数据总线的时序与绝大多数通用的微处理器兼容。全部逻辑输入和输出与 TTL/CMOS 电平兼容。

MAX197 采用逐次逼近工作方式，内部的输入跟踪/保持电路把模拟信号转换为 12 位数字量输出，其并行输出口很容易与单片机接口。在本例中，选用的就是片外并行总线的 A/D 芯片。以下就详细介绍基于 MAX197 的数据采集系统的设计方案。MAX197 芯片的特点使其适于作为信号采集芯片，其原因如下。

- 电源：单一 5V 电源，与 MG100 的供电方式和信号输出方式相同。
- 分辨率：12 位分辨率，误差  $\pm 1/2(LSB)$ ，对于 MG100 的输出而言，精度足够高。
- 通道：8 路模拟输入通道；可以同时测量 MG100 的两路输出。
- 量程：输入量程可用软件选定不同范围： $\pm 10V$ 、 $\pm 5V$ 、 $0V \sim 10V$ 、 $0V \sim 5V$ 。
- 转换时间：为  $6\mu s$ ，采样速率为 100kSPS。对于测控系统而言，转换时间足以满足单片机控制算法的要求。
- 时钟：可通过软件选择内部或外部时钟，便于单片机控制采样基准。
- 电压基准：可用软件选择使用内部 4.096V 电压基准或外部电压基准。
- 功耗：可通过 SHDN 引脚和软件选择低功耗工作方式，对于独立电源的测控系统而言尤为重要。

MAX197 芯片的管脚图如图 10-24 所示，相应的功能如下。

- CLK: 时钟输入，在外部时钟模式下，输入与 TTL/MOS 相匹配的时钟脉冲。在内部时钟模式下，从这个引脚接一个电容 CCLK 至地，设置内部时钟频率；当 CCLK=100pF 时，CLK 典型值为 1.56MHz。
- CS: 片选脚，低电平有效。
- WR: 当 CS 为低电平时，在内部采集模式，WR 的上升沿将锁住数据，并发出一个采集脉冲。当 CS 为低电平时，在外部采集模式下，WR 的第一个上升沿启动一次采集，WR 的第二个上升沿结束采集并开始一次转换。

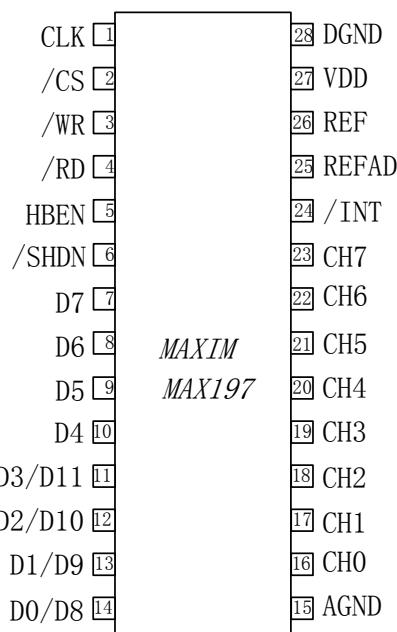


图 10-24 MAX197 芯片封装图

- RD: 如果 CS 为低电平，RD 的下降沿将实现数据总线上的一次读操作。
- HBEN: 输入脚，控制数据总线复用，以得到 12 位转换结果。当 HBEN 为高电平时，数据总线上输出高 4 位数据；当 HBEN 为低电平时，数据总线上为低 8 位数据。
- SHDN: 掉电输入脚，当 SHDN 为低电平时，器件进入掉电工作状态。
- D7 ~ D4: 三态数据 I/O 口。
- D3/D11 ~ D0/D8 三态数据 I/O 口。当 HBEN=0 时，输出为 D3 ~ DO 的数据，当 HBEN=1 时，输出为 D11 ~ D8 数据。
- AGND: 模拟地。CH0 ~ CH7 为八路模拟输入通道。
- INT: 中断输出脚，当转换完毕，输出数据准备就绪，INT 变为低电平。
- REFADJ: 为带隙电压基准输出/外部调节引脚。可联接一个  $0.01\mu F$  电容旁路至地。当在 REF 脚上采用外部基准电压时，此管脚连到 VDD 上。
- REF: 缓冲器基准电压输出/ADC 基准电压输入。在内部基准电压模式下，基准缓冲器提供  $4.096V$  的标准输出电压，可在 RE FADJ 脚微调。在外部基准电压模式下，通过把 REFADJ 接至 VDD 使内部缓冲器无效。
- VDD:  $+5V$  电源，通过  $0.1\mu F$  电容旁路至地。
- DGND: 数字地。

根据 MAX197 芯片的特点和使用要求，利用 MAX197 和 89C52 单片机组成的信号采集电路的基本方案如图 10-25 所示。

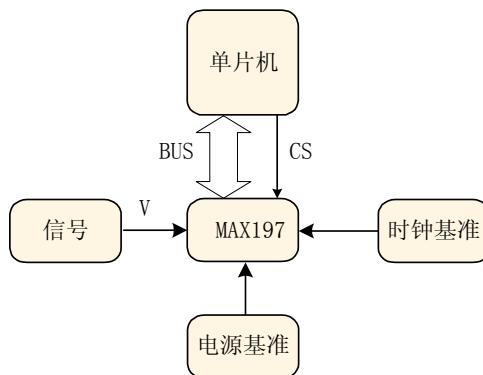


图 10-25 信号采集电路的基本方案

### 3. A/D 芯片在使用时需要注意什么问题？

使用片外的 A/D 芯片是单片机数据采集系统常用的一种方法。在使用片外的 A/D 转换芯片时，需要注意的一个重要问题就是 A/D 的电源设计，这将作为一个独立的章节在后文中进行介绍，这里，只是结合本实例，给出一个基本概念。

通常单片机的 A/D 数据采集对于电源的要求远高于其他数字电路对电源电压的要求。这是因为在 A/D 数据采集中，电源除了需要提供单片机的电源电压之外，还需完成对 A/D 芯片的供电和提供电压采集的基准。所以对于 A/D 而言，需要在已有的电源基础上做 A/D 电源设计。在本例中，可以使用 MAX1675、MAX8875、MAX6192 三种电源变换芯片。将系统的电源电压从不稳定的 5V 变换到稳定的 5V 和电压基准 2.5V。有关这 3 种芯片及其电压转换电路的详细内容将在后文中介绍。

### 4. 有 MAX197 和单片机系统接口的典型电路吗？

在本例中，主要介绍 MAX197A/D 芯片在单片机 89C52 中的使用方法，主要包括单片机对 MAX197 数据采集通道的选择、数据的读取和数据采集方式的选择等。相关的电路将在本节的电路一节中做详细的介绍。

在单片机中使用 MAX197 数据采集芯片，以及数据采集过程中主要涉及以下几个方面关键内容：

- 时钟和功耗模式的选择。
- 采集方式的选择。
- 量程和极性的选择。
- 采样通道的选择。
- 数据的读取。

这些通道和采样方式等内容的选择，主要是依靠单片机向 MAX197 发送控制字来实现的。MAX197 的控制字格式如表 10-6 所示。

表 10-6 MAX197 的控制字

| D7  | D6  | D5     | D4  | D3  | D2 | D1 | D0 |
|-----|-----|--------|-----|-----|----|----|----|
| PD1 | PD0 | ACQMOD | RNG | BIP | A2 | A1 | A0 |

表中的各个控制位如下。

- PD1、PD0: 选择时钟和低功耗模式, 其设置如表 10-7 所示。

表 10-7 PD<sub>1</sub>、PD<sub>0</sub> 位设置

| PD1 | PD0 | 说明               |
|-----|-----|------------------|
| 0   | 0   | 正常工作, 外部时钟模式     |
| 0   | 1   | 正常工作, 内部时钟模式     |
| 1   | 0   | 后备低功耗模式; 不影响时钟模式 |
| 1   | 1   | 低功耗模式, 不影响时钟模式   |

MAX197 可以以内部或外部时钟模式工作。控制字节的 D6、D7 位选择内部或外部时钟模式。一旦选择了所要求的时钟模式, 改变这些位编程选择低功耗模式时, 不会影响时钟模式。刚上电时, 选择外部时钟模式。内部时钟模式设置控制字节的 D7 位为 0, D6 位为 1 可以选择这种模式。在 CLK 脚和地之间接一个 100pF 的电容, 可产生 156MHz 频率。外部时钟模式设置控制字节的 D7 位=0、D6 位=0 选择外部时钟模式。一般情况, 要求 100kHz~2MHz 的外部时钟具有 45%~55% 的占空比。当工作时钟频率低于 100kHz 时, 在保持电容上将产生一个电压降, 导致性能降低。

- ACQMOD: 0 为内部控制采集, 1 为外部控制采集。

通过写控制字节的 ACQMOD 位为 0, 选择内部采集方式。此方式产生一个脉冲初始化采集间隔, 这个时间是内部定时的。当 6 个时钟周期采集间隔结束时, 转换开始。

通过写控制字节的 ACQMOD 位为 0, 选择内部采集方式。外部采集方式可以更精确地控制采样间隔和转换。在这种方式下, 用户通过 2 个写脉冲控制采集和启动转换。在第一个写脉冲中, 要使 ACQMOD 位=1, 它将启动一次采集开始。在第二个写脉冲中, 要使 ACQ MOD 位=0, 在 WR 的上升沿开始转换并结束采集。在发第一和第二个写脉冲时, 多路输入通道的地址位值必须一样。在第二个写脉冲中, 低功耗模式位 (PD0、PD1) 可以设一个新值。

- RNG、BIP: RNG 位是选择输入端的满量程电压范围, BIP 位选择单极性式和双极性转换模式。这两位设置如表 10-8 所示。

表 10-8 RNG、BIP 位选择

| BIP | RNG | 输入范围 (V) |
|-----|-----|----------|
| 0   | 0   | 0~5      |
| 0   | 1   | 0~10     |
| 1   | 0   | ±5       |
| 1   | 1   | ±10      |

- A2、A1、A0: 用于选择多路输入/输出的地址, 如表 10-9 所示。

表 10-9

多路输入/输出的地址

| A2 | A1 | A0 | CH0 | CH1 | CH2 | CH3 | CH4 | CH5 | CH6 | CH7 |
|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0  | 0  | 0  | ✓   | -   | -   | -   | -   | -   | -   | -   |
| 0  | 0  | 1  | -   | ✓   | -   | -   | -   | -   | -   | -   |
| 0  | 1  | 0  | -   | -   | ✓   | -   | -   | -   | -   | -   |
| 0  | 1  | 1  | -   | -   | -   | ✓   | -   | -   | -   | -   |
| 1  | 0  | 0  | -   | -   | -   | -   | ✓   | -   | -   | -   |
| 1  | 0  | 1  | -   | -   | -   | -   | -   | ✓   | -   | -   |
| 1  | 1  | 0  | -   | -   | -   | -   | -   | -   | ✓   | -   |
| 1  | 1  | 1  | -   | -   | -   | -   | -   | -   | -   | ✓   |

- 数据的读取

在单极性方式下，输出数据格式为二进制数；在双极性方式下，其格式为补码形式的二进制数。在读输出数据时，CS 和 RD 必须为低电平。器件输出的数据一共是 12 位，当 HBEN 为低电平时，读低 8 位；当 HBEN 为高电平时，读取较高的 4 个 MSB 位，输出数据的 D4~D7 位。

数据的读取格式如表 10-10 所示。

表 10-10

数据的读取格式

| 数据位 | HBEN=0   | HBEN=1    |
|-----|----------|-----------|
| D0  | B0 (LSB) | B8        |
| D1  | B1       | B9        |
| D2  | B2       | B10       |
| D3  | B3       | B11 (MSB) |
| D4  | B4       | B11       |
| D5  | B5       | B11       |
| D6  | B6       | B11       |
| D7  | B7       | B11       |

### 10.3.3 电路

本节将给出完整的实例电路原理图。读者可以根据该电路设计印刷电路板，其基本的电路如图 10-26 所示。

#### 1. 电路原理和器件选择

在这里列出和本例相关的、关键部分的器件名称及其在电路中的主要功能。

- 89C52：主要通过对 MAX197 的设置，完成对数据采集过程以及数据采集方式的控制，读取 MAX197 的寄存器完成数据的传输。
- MAX197：A/D 转换芯片，在单片机的控制下实现不同方式的数据采集，并将采集到的数据传输到单片机中。
- 74LS138：3-8 译码器，通过单片机的地址产生片选信号，用于选择 MAX197 芯片。

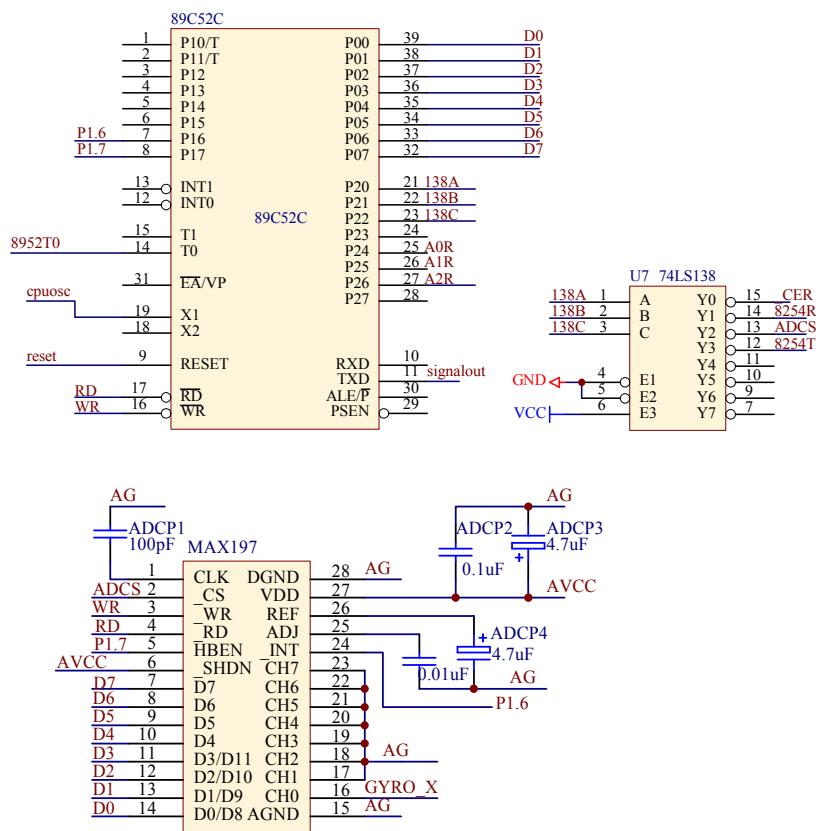


图 10-26 A/D 信号采集系统电路

## 2. 地址分配和连接

只列出和本例相关的、关键部分的单片机与各个功能管脚的连接和相关的地址分配。

- ADCS: MAX197 的片选信号。由 74LS138 译码器产生，单片机通过片选信号选择 MAX197 器件。
- AG: MAX197 的地信号。
- AVCC: 陀螺、MAX6192、MAX197 的电源电压，由一系列的电压处理芯片对系统电源的处理所得。
- GYRO\_X: 经过信号调理的陀螺输出。
- P1.7: MAX197 输出数据的高低位控制，单片机通过设置 P1.7 的电平高低分别读取 12 位数据的高低位。
- P1.6: MAX197 的中断控制位。单片机通过设置 P1.6 的高低电平触发一次数据采集和数据读取。
- 138A、138B、138C: 74LS138 的片选信号输入，分别连接单片机的 P2.0、P2.1 和 P2.2 管脚，单片机通过设置这 3 个管脚的高低电平，在 138 译码器的输出产生不同的片选信号，选择 MAX197 等其他的外部器件。

### 10.3.4 程序设计

#### 1. 程序功能

基于 MAX197 的数据采集程序主要包括 3 个方面的关键内容，一是 MAX197 寄存器的地址定义，二是控制字的写入，三是数据的读取。

#### 2. 主要器件和变量的说明

本例中所使用的主要功能器件就是 MAX197 模数转换芯片。程序中的变量及功能如表 10-11 所示。

表 10-11

变量及功能

| 变量       | 说明                |
|----------|-------------------|
| Adch0    | MAX197 的通道 0 的地址  |
| P1_6     | MAX197 的中断控制位     |
| P1_7     | MAX197 输出数据的高低位控制 |
| Ch0data1 | 采样数据的低 8 位        |
| Ch0datah | 采样数据的高位           |

#### 3. 程序代码

利用单片机控制 MAX197 模数转换代码如下，流程如图 10-27 所示。

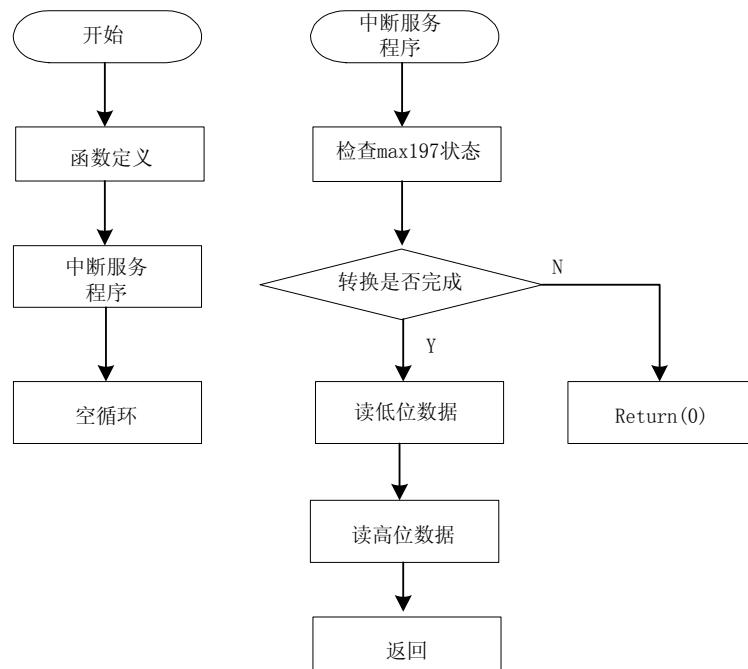


图 10-27 控制 MAX197 模数转换流程图

```
#include<reg51.h>
#include <absacc.h>
```

```

#define uchar unsigned char
#define uint unsigned int

/* MAX197 片外地址的定义，通过如图 5-40 所示的硬件连线，可以看出，当单片机的 P2.1 口的电平
为高时，138 译码器的片选信号选中 ADCS 输出。*/
#define adch0 XBYTE[0x0200]

//设置 P1.7, MAX197 输出数据的高低位控制; P1.6, MAX197 的中断控制位
uchar ch0datal,ch0datah;
sbit P1_6=P1^6;
sbit P1_7=P1^7;

main()
{
 EA=1; /*初始化：打开 INT0、INT1 和 T2 */
 IT0=1;
 IT1=1;
 EX0=1;
 EX1=1;
 ET2=1;
 T2CON=0x08; /*设置 T2 工作方式:允许接收 P1.1 引脚的下降沿中断，计数器停
止*/
 TMOD=0x99; /*设置 T0、T1 工作方式：门控方式 1 定时*/
 TR0=1; TR1=1;
 for (;;) {
}

// 在本例中使用定时中断的方法定时进行数据采集
void intsvr1(void) interrupt 1
{
 // 向 MAX197 的控制字寄存器中写入控制字 0x40;
 // MAX197 处于正常工作的内部时钟模式，0~5V 的测量范围，测量通道为 0 号
 adch0=0x40;

 // 使用查询的方法读 MAX197 的中断信号，检测 MAX197 是否完成了信号的采集
 while(P1_6!=0)
 {
 // 当数据采集完成时，先设置 HBEN=0，即先读低位
 P1_7=0;
 }

 ch0datal=adch0;

 // 当数据采集完成时，设置 HBEN=1，再读高位
}

```

```

P1_7=1;
ch0datah=adch0;
P1_7=0;
}

```

## 10.4 使用 DS1820 进行温度补偿和测量

### 10.4.1 实例功能

单片机系统除了可以对电信号进行测量外，还可以对温度信号进行测量。随着现代化信息技术的飞速发展和传统工业改造的逐步实现，能独立工作的温度检测系统已广泛应用于诸多的领域。传统的温度检测大多以热敏电阻为传感器，但热敏电阻可靠性差、测量温度准确率低，且必须经过专门的接口电路转换成数字信号后才能由微处理器进行处理。

DS1820 是美国 DALLAS 公司生产的单线数字温度传感器，它具有微型化、低功耗、高性能、抗干扰能力强、易配微处理器等优点，特别适合于构成多点温度测控系统，可直接将温度转化成串行数字信号进行处理，而且每片 DS1820 都有惟一的产品号并可存入其 ROM 中，以便在构成大型温度测控系统时在单线上挂接任意多个 DS1820 芯片。从 DS1820 读出或写入 DS1820 信息仅需要一根端口线，其读写及温度变换功率来源于数据总线，该总线本身也可以向所挂接的 DS1820 供电，而无需额外电源。DS1820 能提供 9 位温度读数，它无需任何外围硬件即可方便地构成温度检测系统。图 10-28 显示的是 DS1820 在 LTM-8201/16 和 LTM-8202/32 系列多路巡检报警仪中的应用。



图 10-28 LTM-8201/16 和 LTM-8202/32

本例就以 DS1820 为温度传感器，介绍如何利用 DS1820 和单片机组成一个温度补偿和测量系统。本例主要介绍 DS1820 的基本功能和测量原理、DS1820 和单片机的接口电路以及单片机编程控制温度测量的程序。

本例的功能模块分为以下 3 个方面。

- 单片机系统：采用 89C2051 单片机和 DS1820 温度传感器通信，控制温度的采集过程和进行数据通信。
- 外围电路：提供 DS1820 的使用外围电路、温度指示电路以及 DS1820 和单片机的通信接口电路。
- C51 程序：编写 C51 程序，完成单片机对温度数据的采集过程以及与 DS1820 数据传输过程的控制。

希望读者在读完本例后，能完成相关的电路设计，并掌握如下的知识点内容：

- 掌握 DS1820 的使用和特性。
- 单片机和 DS1820 的接口电路的设计。
- C51 程序设计 DS1820 的温度采集过程。

### 10.4.2 器件和原理

#### 1. 什么是 DS1820 温度传感器？

美国 DALLAS 公司最新推出的 DS1820 数字式温度传感器，与传统的热敏电阻有所不同的是，它可直接将被测温度转化成串行数字信号供微机处理，并且根据具体要求，通过简单的编程实现 9 位的温度读数。并且多个 DS1820 可以并接到多个地址线上与单片机实现通信。由于每一个 DS1820 出厂时都刻有惟一的一个序列号并存入其 ROM 中，因此 CPU 可用简单的通信协议就可以识别，从而节省了大量的引线和逻辑电路。DS1820 采用 3 脚 TO-92 封装或 8 脚 SO 封装，管脚排列如图 10-29 所示。对图中 DS1820 的引脚功能说明如下。

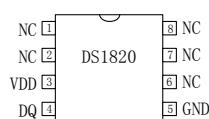


图 10-29 DS1820 的封装和引脚

NC：空引脚，不连接外部信号。

V<sub>DD</sub>：接电源引脚，电源供电 3.0~5.5V。

GND：接地。

DQ：数据的输入和输出引脚。

值得一提的是 DQ 引脚的 I/O 为数据输入 / 输出端（即单线总线），该引脚为漏极开路输出，常态下呈高电平。而单线总线，即 1-wire 技术目前是 DS1820 的一个特点，也是目前的技术热点之一。

#### 2. 什么是 DS1820 中的 1-wire 技术？

接下来就讨论关于 DS1820 中的 1-wire 技术问题。我们知道，目前常用的微机与外设之间进行数据传输的串行总线主要有 I<sup>2</sup>C 总线、SPI 总线等。其中 I<sup>2</sup>C 总线以同步串行 2 线方式进行通信（一条时钟线、一条数据线），SPI 总线则以同步串行 3 线方式进行通信（一条时钟线、一条数据输入线、一条数据输出线）。这些总线至少需要两条或两条以上的信号线。而达拉斯半导体公司推出了一项特有的 1-wire Bus 技术。该技术与上述总线不同，它采用单根信号线，既可传输时钟，又能传输数据，而且数据传输是双向的，因而这种单总线技术具有线路简单，硬件开销少，成本低廉，便于总线扩展和维护等优点。

单总线适用于单主机系统，能够控制一个或多个从机设备。主机可以是微控制器，从机可以是单总线器件，它们之间的数据交换只通过一条信号线。当只有一个从机设备时，系统可按单节点系统操作；当有多个从设备时，系统则按多节点系统操作。

单总线即只有一根数据线，系统中的数据交换、控制都由这根线完成。主机或从机通过一个漏极开路或三态端口连至该数据线，以允许设备在不发送数据时能够释放总线，而让其他设备使用总线，其内部等效电路如图 10-30 所示。单总线通常要求外接一个约为 4.7kΩ 的

上拉电阻，这样，当总线闲置时，其状态为高电平。主机和从机之间的通信可通过 3 个步骤完成，分别为初始化 1-wire 器件、识别 1-wire 器件和交换数据。由于它们是主从结构，只有主机呼叫从机时，从机才能应答，因此主机访问 1-wire 器件都必须严格遵循单总线命令序列，即初始化、ROM、命令功能命令。如果出现序列混乱，1-wire 器件将不响应主机。

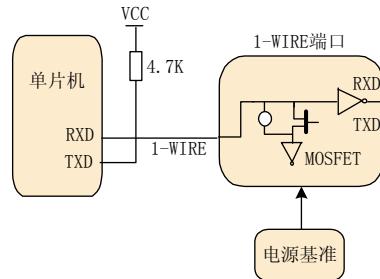


图 10-30 内部等效电路

所有的单总线器件都要遵循严格的通信协议，以保证数据的完整性。1-wire 协议定义了复位脉冲、应答脉冲、写 0、读 0 和读 1 时序等几种信号类型。所有的单总线命令序列都是由这些基本的信号类型组成的。在这些信号中，除了应答脉冲外，其他均由主机发出同步信号，并且发送的所有命令和数据都是字节的低位在前。

基本的通信过程如下：

- 主机通过拉低单总线至少  $480\mu s$  产生 Tx 复位脉冲。
- 然后由主机释放总线，并进入 Rx 接收模式。主机释放总线时，会产生一由低电平跳变为高电平的上升沿。
- 单总线器件检测到该上升沿后，延时  $15 \sim 60\mu s$ 。
- 单总线器件通过拉低总线  $60 \sim 240\mu s$  来产生应答脉冲。
- 主机接收到从机的以应答脉冲后，说明有单总线器件在线，然后主机就可以开始对从机进行 ROM 命令和功能命令操作。

所有的读、写时序至少需要  $60\mu s$ ，且每两个独立的时序之间至少需要  $1\mu s$  的恢复时间。在写时序中，主机将在拉低总线  $15\mu s$  之内释放总线，并向单总线器件写 1；若主机拉低总线后能保持至少  $60\mu s$  的低电平，则向单总线器件写 0。单总线器件仅在主机发出读时序时才向主机传输数据，所以，当主机向单总线器件发出读数据命令后，必须马上产生读时序，以便单总线器件能传输数据。

单总线技术以其线路简单、硬件开销少、成本低廉、软件设计简单优势而有着无可比拟的应用前景。基于单总线技术能较好地解决传统识别器普遍存在的携带不便、易损坏、易受腐蚀、易受电磁干扰等不足，可应用于高度安全的门禁、身份识别等领域。其通信可靠简单，很容易实现。因此单总线技术有着广阔的应用前景，是值得关注的一个发展领域。

### 3. DS1820 的测温原理是什么？

DS1820 的内部框图如图 10-31 所示，它主要包括寄生电源、温度传感器、64 位激光 ROM 单线接口、存放中间数据的高速暂存器、用于存储用户设定的温度上下限值、触发器存储与控制逻辑、8 位循环冗余校验码发生器等 7 部分。

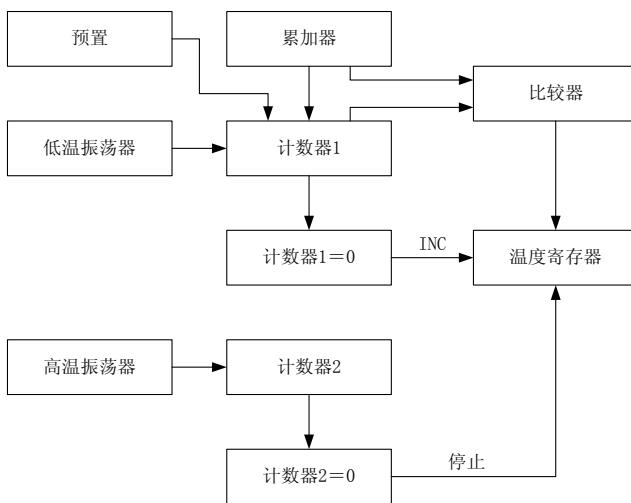


图 10-31 DS1820 的内部框图

低温度系数振荡器是一个振荡频率随温度变化很小的振荡器，为计数器 1 提供一频率稳定的计数脉冲。

高温度系数振荡器是一个振荡频率对温度很敏感的振荡器，为计数器 2 提供一个频率随温度变化的计数脉冲。

初始时，温度寄存器被预置成-55°C，每当计数器 1 从预置数开始减计数到 0 时，温度寄存器中寄存的温度值就增加 1°C，这个过程重复进行直到计数器 2 计数到 0 时便停止。

初始时，计数器 1 预置的是与-55°C 相对应的一个预置值。以后计数器 1 每一个循环的预置数都由斜率累加器提供。为了补偿振荡器温度特性的非线性，斜率累加器提供的预置数也随温度相应变化。计数器 1 的预置数也就是在给定温度处使温度寄存器寄存值增加 1°C 计数器所需的计数个数。

图中比较器的作用是以四舍五入的量化方式确定温度寄存器的最低有效位。在计数器 2 停止计数后，比较器将计数器 1 中的计数剩余值转换为温度值后与 0.25°C 进行比较，若低于 0.25°C，温度寄存器的最低位就置 0；若高于 0.25°C，就置 1，若高于 0.75°C 时，温度寄存器的最低位就进位然后置 0。这样，经过比较后所得的温度寄存器的值就是最终读取的温度值了，其最末位代表 0.5°C，四舍五入最大量化误差为±1/2LSB，即 0.25°C。

温度寄存器中的温度值以 9 位数据格式表示，最高位为符号位，其余 8 位以二进制补码形式表示温度值。测温结束时，这 9 位数据转存到暂存存储器的前两个字节中，符号位占用第 1 字节，8 位温度数据占用第 2 字节。

DS1820 测量温度时使用特有的温度测量技术。DS1820 内部的低温度系数振荡器能产生稳定的频率信号；同样的，高温度系数振荡器则将被测温度转换成频率信号。当计数门打开时，DS1820 进行计数，计数门开通时间由高温度系数振荡器决定。芯片内部还有斜率累加器，可对频率的非线性加以补偿。测量结果存入温度寄存器中。一般情况下的温度值应为 9 位（包含一位符号），但因符号位扩展成高 8 位，故以 16 位补码形式读出，温度和数字量的关系如表 10-12 所示。

表 10-12

温度和数字量的关系

| 变量   | 输出的二进制码          | 对应的十六进制码 |
|------|------------------|----------|
| +125 | 000000011111010  | 00FAH    |
| +25  | 000000000110010  | 0032H    |
| +1/2 | 0000000000000001 | 0001H    |
| 0    | 0000000000000000 | 0000H    |
| -1/2 | 1111111111111111 | FFFFH    |
| -25  | 1111111111001110 | FFCEH    |
| -55  | 1111111110010010 | FF92H    |

### 10.4.3 电路

本实例以 DS1820 作为温度传感器，采用前文使用过的 89C2051 作为微控制器，形成一个温度补偿和检测系统。该系统的输出为一个 LCD 显示器，由于 LCD 的电路在前文中已经做过介绍，这里就不再详述。该系统的硬件电路如图 10-32 所示。

#### 1. 电路原理和器件选择

在这里列出和本例相关的、关键部分的器件名称及其在电路中的主要功能。

- 89C2051：单片机，进行数据采集和转换的工作。
- WU1：12MHz 的晶振。
- SW - PB：复位开关。
- DS1820：温度传感器。
- LCD：液晶显示模块的接口。
- D3：发光二极管，指示系统的工作状态。

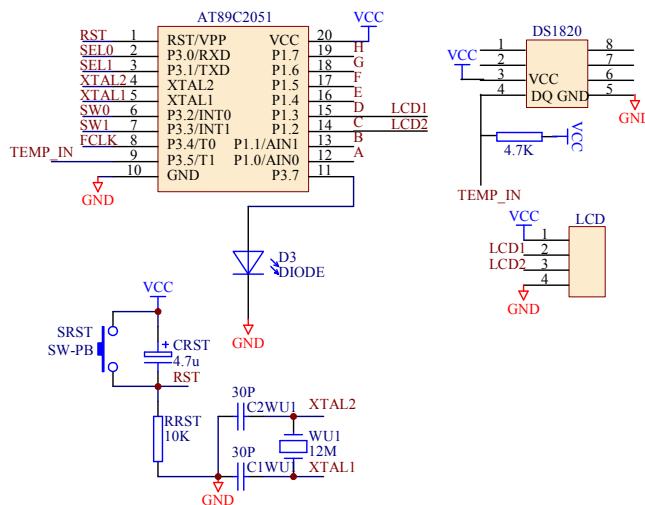


图 10-32 系统的硬件电路

## 2. 地址分配和连接

只列出和本例相关的、关键部分的单片机与各个模块管脚的连接和相关的地址分配。

- P1.0: 模拟量输入。
- P1.1: DA 输入比较基准电压。
- P3.7: LED 的显示输出。
- XTAL1: 晶振的引脚 1。
- XTAL2: 晶振的引脚 2。
- TEMP\_IN: 温度传感器 DS1820 的输入。
- LCD1: LCD 液晶显示模块的输入引脚 1。
- LCD2: LCD 液晶显示模块的输入引脚 2。

### 10.4.4 程序设计

#### 1. 程序功能

该程序的主要功能是使用单片机和温度传感器 DS1820 完成对温度的检测和补偿。通过 89C2051 完成对 DS1820 芯片的控制和数据传输。程序的功能是查询当前的 DS1820 温度采集和转换是否完成，并且完成对转换后数据读取。

#### 2. 主要器件和变量的说明

本例采用的器件主要有 3 个，其中 DS1820 是温度传感器，检测外部的温度信号；单片机负责控制信号采集过程；LCD 模块显示当前的温度数据。程序中的变量及功能如表 10-13 所示。

表 10-13

变量和功能

| 变量         | 说明         |
|------------|------------|
| P3_5       | 数据通信端口     |
| TEMP       | 温度值的变量     |
| flag1      | 结果为负和正的标志位 |
| tmchange() | 开始温度转换     |
| delay()    | 延时函数       |
| tmp()      | 读取温度函数     |
| count      | 延时技术变量     |
| dsreset()  | 复位和初始化函数   |
| tmpread()  | 读取数据的一位    |
| tmpread2() | 读取数据的一个字节  |
| tmpwrite() | 写数据的一个字节   |
| rom()      | 读取器件的序列号   |
| dat        | 读取数据的临时变量  |

### 3. 程序代码

系统的主程序用 C51 编写，主要完成对 DS1820 的调用中断管理、测量温度值的计算及温度值的显示等功能。在这里，我们省略了 LCD 显示部分的代码，因为在前文中已经介绍过，并且代码太长。DS1820 自动的分辨率可以通过编程进行选择。显示程序实现是对各温度值的显示，并且允许中断的产生以修正温度值，实现及时的温度测量。另外，在系统安装之前及工作之前应将主机逐个与 DS1820 挂接，以读出其序列号。系统对 DS1820 操作的总体流程图如图 10-33 所示。单个 DS1820 温度读取程序如下。

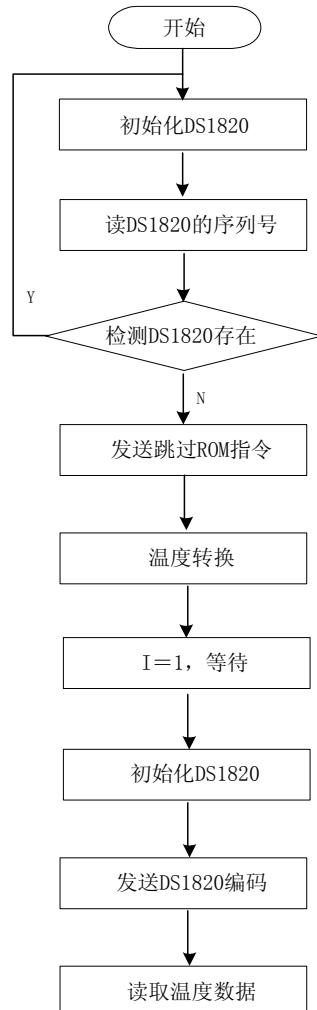


图 10-33 DS1820 操作的总体流程图

```

#include <reg51.h>
#include <absacc.h>
#include <stdio.h>
#include <math.h>
#define uchar unsigned char
#define uint unsigned int
#define P3_5 P3^5;

```

```

uchar TEMP; // 温度值的变量
uchar flag1; // 结果为负和正的标志位

void delay (unsigned int count)
{
 unsigned int i;
 while (count)
 {
 i =200;
 while (i>0) i--;
 count--;
 }
}

void dsreset (void) // 发送复位和初始化
{
 unsigned int i;
 P3_5 = 0;
 i = 103;
 while (i>0) i--; // 延时
 P3_5 = 1;
 i = 4;
 while (i>0) i--;
}

bit tmpread (void) // 读取数据的一位
{
 unsigned int i;
 bit dat;
 P3_5 = 0; i++;
 P3_5 = 1; i++; i++; // 延时
 dat = P3_5;
 i = 8; while (i>0) i--; // 延时
 return (dat);
}

unsigned char tmpread2 (void) // 读一个字节
{
 unsigned char i,j,dat;
 dat = 0;
 for (i=1;i<=8;j++)

```

```

 {
 j = tmpread ();
 dat = (j << 7) | (dat >> 1);
 }
 return (dat);
}

void tmpwrite (unsigned char dat) // 写一个字节
{
 unsigned int i;
 unsigned char j;
 bit testb;
 for (j=1;j<=8;j++)
 {
 testb = dat & 0x01;
 dat = dat >> 1;
 if (testb)
 {
 P3_5 = 0; // 写 0
 i++; i++;
 P3_5 = 1;
 i = 8; while (i>0) i--;
 }
 else
 {
 P3_5 = 0; // 写 0
 i = 8; while (i>0) i--;
 P3_5 = 1;
 i++; i++;
 }
 }
}

void tmpchange(void) // DS1820 开始转换
{
 dsreset (); // 复位
 //tmpre (); // 等待存在脉冲
 delay (1); // 延时
 tmpwrite (0xcc); // 跳过序列号命令
 tmpwrite (0x44); // 发转换命令 44H
}

```

```

}

void tmp (void) // 读取温度
{
 unsigned char a,b;
 dsreset (); // 复位
 delay (1); // 延时
 tmpwrite (0xcc); // 跳过序列号命令
 tmpwrite (0xbe); // 发送读取命令
 a = tmpread2 (); // 读取低位温度
 b = tmpread2 (); // 读取高位温度
 flag1=b; // 若 b 为 1 则为负温
 if(flag1)
 {
 TEMP=~a+1; // 如果为负温则去除其补码
 }
 else
 {
 TEMP=a;
 }
}

rom0 // 读取器件序列号子程序
{
 uchar i;
 uchar sn1;
 uchar sn2;
 dsreset (); // 复位
 delay (1); // 延时
 tmpwrite(0x33); // 发送读序列号子程序
 sn1= tmpwrite (); // 读取第一个序列号, 应为 16H
 sn2= tmpwrite (); // 读取第二个序列号, 应为 10H
}

main()
{
 do{
 tmpchange(); // 开始温度转换
 delay(200); // 读取延时
 tmp(); // 读取温度
 }while(1);
}

```

## 10.5 语音芯片在单片机系统中的使用

### 10.5.1 实例功能

在智能仪器仪表或自动控制设备中，增加语音功能能极大地提高人机界面的友好性，方便用户操作。在许多场合，设计者需要将语音系统和单片机结合在一起，解决上述问题。目前语音服务行业越来越广，如电脑语音钟、语音型数字万用表、手机话费查询系统、排队机以及公共汽车报站器等。

语音芯片可以很方便的在单片机系统中使用，并且和单片机的接口非常容易设计，其体积和重量也能符合单片机系统的要求。如图 10-34 所示为一款 IDS1810 的语音芯片。

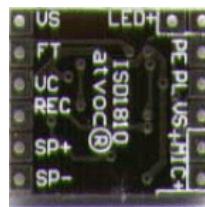


图 10-34 IDS1810 的语音芯片

所谓语音芯片就是在人工或者是控制器的控制下可以录音和放音的芯片，本例中将以 ISD 公司生产的 ISD 系列芯片为例，说明语音芯片的使用方法，并且实现单片机定时播放语音的功能。

在单片机系统中使用 ISD 系列的语音芯片时，需要考虑 3 个方面的内容，一个方面是如何使用 ISD 语音芯片，二是如何根据选择的 ISD 芯片设计外围电路和单片机的接口电路，三是如何编写定时控制语音芯片的单片机程序。

本例的功能模块分为以下 3 个方面。

- 单片机系统：输出控制信号，控制语音芯片定时播放特定的语音。
  - 外围电路：实现外围的 ISD 系列语音芯片，本例中使用的是 ISD2560 芯片和单片机之间的接口电路。
  - C51 程序：编写定时 1 秒钟的程序，并在定时中断到来时播放语音芯片中的内容。
- 希望读者在读完本例后，能完成相关的电路设计，并掌握如下的知识点内容：
- 语音芯片工作的原理和器件的选择。
  - 单片机和语音芯片的电路接口。
  - 掌握语音芯片 ISD2569 语音播放过程的 C51 程序设计。

### 10.5.2 器件和原理

#### 1. 什么是语音芯片？

语音芯片就是在人工或者是控制器的控制下可以录音和放音的芯片。比较典型的器件有美国 ISD 公司生产的 ISD 系列语音芯片。

ISD 系列语音芯片采用模拟数据在半导体存储器直接存储的专利技术，即将模拟语音数

据直接写入单个存储单元，不需经过 A/D 或 D/A 转换，因此能够较好地真实再现语音的自然效果，避免了一般固体语音电路因为量化和压缩所造成的量化噪声和失真现象。另外芯片功能强大：即录即放、语音可掉电保存、10 万次的擦写寿命、手动操作和 CPU 控制兼容、可多片级联、无须开发系统等，确实给欲实现语音功能的单片机应用设计人员提供了单片的解决方案。现在市场上已有公司将以 AT89C2051 单片机与 ISD 语音芯片组成的语音组合板，用串行口通信，芯片里固化有一些常用的语音词汇，用户不需了解语音功能的工作原理，只需通过串口按一定的协议发送代码即可送出语音。

## 2. 如何选择合适的语音芯片？

目前，在市场上比较常见的和使用较为普遍的语音芯片如表 10-14 所示。

表 10-14

普遍的语音芯片

| 型号      | 特征                                                                                                        |
|---------|-----------------------------------------------------------------------------------------------------------|
| TE6310  | 语音长度：10 sec<br>采样频率 (kHz): 6.4<br>放音触发：放音触发<br>工作电压 (V): 4.5~5.5<br>工作电流 (mA): 30<br>静态电流 ( $\mu$ A): 2   |
| TE6332  | 语音长度：32 sec<br>采样频率 (kHz): 4~6.4<br>MIC 前置：YES<br>工作电压 (V): 2.7~3.3<br>工作电流 (mA): 45                      |
| ISD1420 | 语音长度：20 sec<br>采样频率 (kHz): 6.4<br>放音触发：边缘/电平<br>工作电压 (V): 4.5~5.5<br>工作电流 (mA): 30<br>静态电流 ( $\mu$ A): 10 |
| ISD2560 | 语音长度：60 sec<br>采样频率 (kHz): 8<br>放音触发：电平<br>工作电压 (V): 4.5~5.5<br>工作电流 (mA): 30<br>静态电流 ( $\mu$ A): 10      |

以典型的 ISD 语音芯片为例，介绍现在比较流行的语音芯片，以及选择语音芯片的标准。目前，市场上的语音芯片和语音板很多，从性能价格比上看，美国 ISD 公司的 ISD 系列录放芯片可谓是一支独秀。ISD 器件的特点如下：

- 使用直接电平存储技术，省去了 A/D、D/A 转换。

- 内部集成了大容量的 EEPROM，不再需要扩展存储器。
- 控制简单，控制管脚与 TTL 电平兼容。
- 具有集成度高、音质好、使用方便等优点。

### 3. ISD2560 的基本功能是什么？

本例将选择美国 ISD 公司的 2590 语音芯片。该芯片的引脚如图 10-35 所示，其基本特点和引脚的功能说明如下。

- (1) ISD2500 系列具有抗断电、音质好、使用方便、无需专用的语音开发系统的特点。
- (2) 片内 EEPROM 容量 480kB，所以录放时间长，录放时间为 90s。
- (3) 有 10 个地址输入端，寻址能力可达 1024 位。
- (4) 语音最多能分 600 段，设有 OVF 溢出端，便于多个器件级联。
- 地址线：A0~A9。共有 1024 种组合状态。最前面的 600 个状态作内部存储器的寻址用，最后 256 个状态作为操作模式。
- 电源：VCCA、VCCD。芯片内部的模拟和数字电路使用不同的电源总线。模拟和数字电源端最好分别走线。
- 地线：VSSA、VSSD。芯片内部的模拟和数字也可使用不同的地线。
- 节电控制：PD。本端拉高使芯片停止工作，进入不耗电的节电状态，芯片发生溢出，即/OVF 端输出低电平后，要将本端短暂变高复位芯片，才能使之再次工作。
- 片选：CE。本端变低后，而且 PD 为低，允许进行录放操作。芯片在本端的下降沿锁存地直线和 P-R 端的状态。
- 录放模式：P-R。本端状态在/CE 的下降沿锁存。高电平选择放音，低电平选择录音。

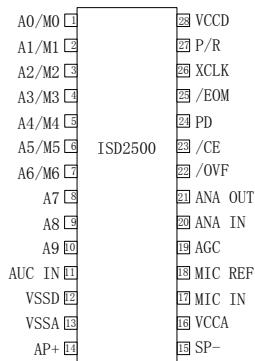


图 10-35 芯片的引脚

- 信息结尾标志：EOM。EOM 标志在录音时由芯片自动插入到该信息的结尾。放音遇到 EOM 时，本端输出低电平脉冲。芯片内部会检测电源电压以维护信息的完整性，当电压低于 3.5V 时，本端变低，芯片只能放音。
- 溢出标志：OVF。芯片处于存储空间末尾时本端输出低电平脉冲表示溢出，之后本端状态跟随 CE 端的状态，直到 PD 端变高。
- 麦克输入：MIC。本端连至片内前置放大器。片内自动增溢控制电路（AGC）将置增益控制在-15 至 24dB。
- 麦克参考：MIC REF。本端是前置放大器的反向输入。当以差分形式连接话筒时，可

减小噪声，提高共模抑制比。

- 自动增益控制：AGC。AGC 动态调整前置增益以补偿话筒输入电平的宽幅变化，使得录制变化很大的音量（从耳语到喧嚣声）时失真都能保持最小。
- 模拟输出：ANA OUT。前置放大器的输出，前置电压增益取决于 AGC 端电平。
- 模拟输入：ANA IN。本端为芯片录音信号输出。对话筒输入来说 ANA OUT 端应通过外接电容连至本端。
- 喇叭输出：SP+、SP-。通过对输出端级驱动  $16\Omega$  以上的喇叭。单端使用时必须在输出端和喇叭间接耦合电容，而双端输出既不用电容又不能将功率提高至 4 倍。录音和节电模式下，它们保持为低电平。
- 辅助输入：AUX IN。当 CE 和 P-R 为高，放音不进行，或处入放音溢出状态时，本端的输入信号过内部功放驱动喇叭输出端。当多个 2500 芯片级联时，后级的喇叭输出通过本端连接到本级的输出放大器。
- 外部时钟：XCLK。本端内部有下拉元件，不用时应接地。芯片内部的采样时钟在出厂前已调节器校，误差地  $\pm 1\%$  内。
- 地址/模式输入：AX/MX。地址端有个作用，取决于最高两位（MSB，即 2532/2548 的 A7 和 A8，或 2560/2590/25120 的 A8 和 A9）的状态。当最高两位中有一个为 0 时，所有输入均解释为地址位，作为当前录入操作的起始地址。地址端只作输入，不输出操作过程中的内部地址信息。

ISD2560 是 ISD 系列单片语音录放集成电路的一种，是一种永久记忆型录放语音电路，录音时间为 60s，能重复录放达 10 万次。它采用直接电平存储技术，省去了 A/D、D/A 转换器。ISD2560 集成度较高，内部包括前置放大器、内部时钟、定时器、采样时钟、滤波器、自动增益控制、逻辑控制、模拟收发器、解码器和 480kB 的 EEPROM 等。内部 EEPROM 存储单元，均匀分为 600 行，具有 600 个地址单元，每个地址单元指向其中一行，每一个地址单元的地址分辨率为 100ms。ISD2560 控制电平与 TTL 电平兼容，接口简单，使用方便。

ISD2560 内置了若干操作模式，可用最少的外围器件实现最多的功能。操作模式也由地址端控制；当最高两位都为 1 时，其他地址端置高就选择某个模式。因此操作模式和直接寻址相互排斥。操作模式可由微控制器也可由硬件实现。这里简单地介绍如何使用 ISD2560 语音芯片。基本电路原理如图 10-36 所示。

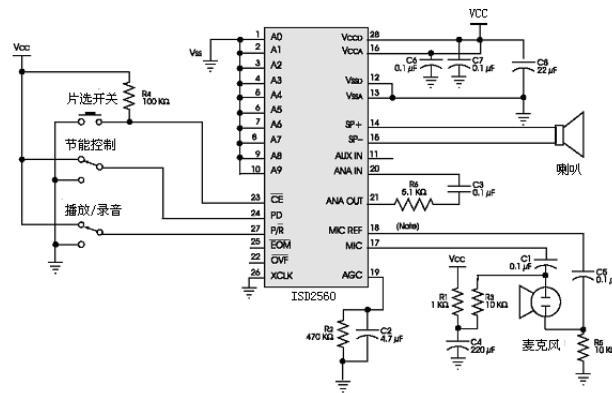


图 10-36 基本电路原理图

录音时按下录音键接地，使 PD 端、P/R 端为低电平，此时启动录音；结束时松开按键，单片机又让 P/R 端回到高电平，即完成一段语音的录制。同样的方法可录取第二段、第三段等。值得注意的是，录音时间不能超过预先设定的每段语音的时间。放音的操作更为简单，按下录音键接高电平，使 PD 端、P/R 端为低电平启动放音功能；结束时，松开按键，即完成一段语音的播放。

在控制上，除去手动外，ISD 器件也可以通过地址寻址来精确定位，但它的地址不是字节地址单元，而是信息段的基本组成单位。以 ISD2560 为例，它内部的 480kB 的 EEPROM 均匀地规划为 600 行，每个地址单元指向其中一行，有 600 个地址单元。ISD2560 的录放时间是 605，因此地址分辨率是 100ms。ISD 器件可进行多段地址操作，每一段称为一个信息段，它可以占用一行和多行存储空间。一个地址单元最多只能作为一个独立的段。因此，ISD2560 最多可以分为 600 个信息段。这就为在单片机系统中使用 ISD2560 语音芯片提供了基本条件。

#### 4. 采用单片机控制语音芯片的好处是什么？

从以下的两个方面考虑实际使用中的要求。

(1) 单片机系统的需要。在一些应用场合，如手机话费查询系统、排队机以及公共汽车报站器等，这些应用中需要实现自动播音，而 ISD2560 实现自动播音的方法，最为简单的就是和单片机系统相连。

(2) 简化人工操作。通常情况下，只能使用 ISD 器件提供的无需知道地址的操作模式，即手动模式，这适合于开发一些简单的语音功能，而无法满足复杂操作或实时系统中应用的要求。为实现以上应用，最好使用对地址直接操作的办法。但在实用中，一些电路开发设计只是在基于语音信号已经写入芯片，并且段地址已经知道的基础上才能进行。然而，不可避免地要遇到必须将语音写入的时候。如果手动处理，采用按录音按键录音，按停止按键停止，假如录音段数特别多，就要频繁地按上述按键，实在让人疲惫不堪。此外，手动按下录音及停止按键的时间也很难掌握，这就容易产生段间空白，造成芯片空间浪费，对语音段特别多，而语句又特别短的提示，如一些单字、单词更是浪费严重。不仅这样，由于短句中空白时间过长，合成放音时出现语音不连贯。

正是由于上述的原因，需要将单片机系统和语音芯片联系起来，形成一个智能化的语音播放系统。在本例中，单片机需要完成以下两个功能：

- 通过 ISD2560 芯片，录制一段语音信息。
- 利用单片机定时 10s，循环播放一段录制的语音。

### 10.5.3 电路

本例的器件主要是单片机和 ISD2560 语音芯片，具体的接口电路如图 10-37 所示。

#### 1. 电路原理和器件选择

在这里列出和本例相关的、关键部分的器件名称及其在电路中的主要功能。

- 89C52：主要通过对 ISD2560 的设置，完成对语音播放过程的控制。
- ISD2560：语音芯片，在单片机的控制下实现语音的定时播放，并且可以通过按键实

现录音功能。

- SPEAKER: ISD2560 语音芯片外接的扬声器。
- MIC: ISD2560 语音芯片外接的麦克风。

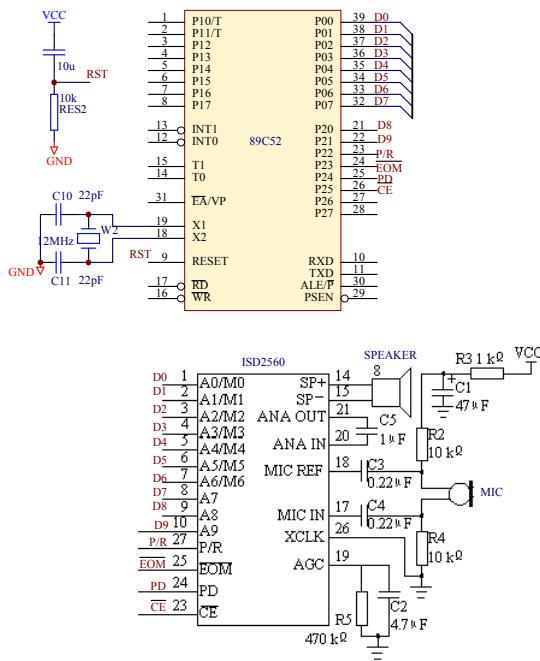


图 10-37 单片机和 ISD2560 语音芯片

## 2. 地址分配和连接

只列出和本例相关的、关键部分的单片机与各个功能管脚的连接和相关的地址分配。

- D0~D9: 单片机和 ISD2560 语音芯片的地址连接，通过对 D8、D9 的设置，单片机可以控制语音芯片的工作方式。
- PD: 节电控制，和单片机的 P2.4 口相连，单片机可以控制芯片的开关。
- CE: 片选，和单片机的 P2.5 口相连，单片机可以选中芯片。
- P/R: 录放模式，和单片机的 P2.2 口相连，单片机可以控制芯片处于录音或者是播放的工作状态。
- EOM: 信息结尾标志，和单片机的 P2.3 口相连，EOM 标志在录音时由芯片自动插入到该信息的结尾。

## 3. 功能简介

录音时，按下录音键，单片机通过 D 端口线设置语音段的起始地址，再使 PD 端、P/R 端为低电平启动录音；结束时，松开按键，单片机又让 P/R 端回到高电平，即完成一段语音的录制。同样的方法可录取第二段、第三段等。值得注意的是，录音时间不能超过预先设定的每段语音的时间。

放音时，根据需播放的语音内容，找到相应的语音段起始地址，并通过口线送出。再将 P/R 端设为高电平，PD 端设为低电平，并让 CE 端产生一负脉冲启动放音，这时单片机只需

等待 ISD2560 的信息结束信号。信号为一负脉冲，在负脉冲的上升沿，该段语音才播放结束，所以单片机必须要检测到的上升沿才能播放第二段，否则播放的语音就不连续。

### 10.5.4 程序设计

#### 1. 程序功能

单片机控制语音芯片定时播放的程序主要包括两个方面的关键内容，一是单片机对 ISD2560 芯片的控制字的写入，二是定时中断的产生。

#### 2. 主要器件和变量的说明

单片机控制语音芯片定时播放的程序中的变量及功能如表 10-15 所示。

表 10-15

变量及功能

| 变量        | 说明        |
|-----------|-----------|
| BUFFER[1] | 定时器计数变量   |
| PD        | 芯片电源开关    |
| EOM       | 结束信号      |
| PR        | 播放=1，录音=0 |
| CE        | 片选信号      |
| play()    | 播放语音子程序   |
| delays()  | 延时子程序     |

#### 3. 程序代码

利用单片机控制语音芯片定时播放的程序流程图如图 10-38 所示，代码如下。

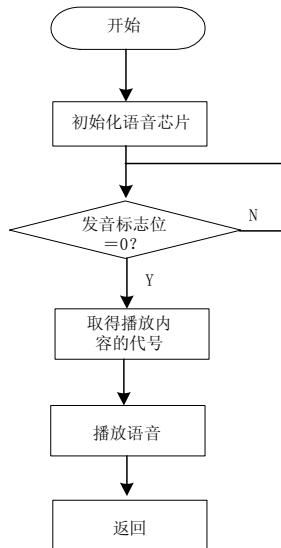


图 10-38 单片机控制语音芯片定时播放流程图

```
#include<reg51.h>
```

```

#include <absacc.h>
#include <stdio.h>
#define uchar unsigned char
#define uint unsigned int
uchar data BUFFER[1]={0}; /*定时器计数变量*/

sbit PR=P2^2; // 定义语音芯片的控制端口，下同
sbit EOM=P2^2;
sbit PD=P2^4;
sbit CE=P2^5;

void delays(void) // 延时程序
{
 uchar i;
 for(i=300;i>0;i--);
}

void play(void)
{
 PD=1; // 打开芯片电源开关
 CE=0; // 选中该芯片
 PR=1; // 开始播放
 while(!EOM); // 等待播放内容结束信号
 delays(); // 延时
 PD=0;
 CE=0;
 PR=0;
}

main()
{
 EA=1; IT1=1; ET0=1;
 TMOD=0x01; /* T0 方式 1 计时 1 秒 */
 TH0=-5000/256;
 TL0=-5000%256;
 TR0=1; /* 开中断，启动定时器 */
 for (++);
}

/* 定时计数器 0 的中断服务子程序 */

```

```

void timer0(void) interrupt 1 using 1
{
 TH0=-5000/256; // 定时器 T0 的高 4 位赋值
 TL0=-5000%256;
 BUFFER[0]=BUFFER[0]+1; // 定时器 T0 的低 4 位赋值
 if(BUFFER[0]==1000) // 百分秒进位
 {
 play(); // 调用播放程序
 }
}

```

## 10.6 时钟芯片在单片机系统中的应用

### 10.6.1 实例功能

在许多的单片机系统中，通常进行一些与时间有关的控制，这就需要使用实时时钟。例如，在测量控制系统中，特别是长时间无人职守的测控系统中，经常需要记录某些具有特殊意义的数据及其出现的时间。在系统中采用实时时钟芯片则能很好地解决这个问题。

以前，通常采用并行的实时时钟芯片计时、EEPROM 作为存储器，但对一些微小型智能控制设备而言，并行实时时钟芯片封装形式大，再加上 EEPROM，占用扩展口线多，使电路结构很难进一步简化。Dallas 公司生产的串行实时时钟芯片 DS1302 具有实时时钟和静态 RAM，采用串行通信，可方便地与单片机接口。除了在工业控制中使用外，还可以应用到一般的时钟计数上，图 10-39 显示的就是一款桌面电子万年历。



图 10-39 桌面电子万年历

本例中利用单片机 89C2051 和时钟芯片 DS1302 进行数据通信，读取和写入实时数据。本例中主要涉及 3 个方面的内容，一个方面是如何针对系统的需求选择合适的时钟芯片，二是如何设计外围电路和单片机的接口电路，三是如何编写控制时钟芯片的单片机程序。

本例的功能模块分为以下 3 个方面。

- 单片机系统：和外围的时钟芯片之间进行通信，并控制数据传输过程。
- 外围电路：实现外围的时钟芯片和单片机之间的接口电路。
- C51 程序：编写单片机控制时钟芯片的接口程序，实现单片机和时钟芯片之间的数据通信功能。

希望读者在读完本例后，能完成相关的电路设计，并掌握如下的知识点内容：

- 时钟芯片 DS1302 的原理和选择。
- 单片机和时钟芯片的电路接口。
- 掌握 C51 设计时钟芯片数据读取和写入的程序。

### 10.6.2 器件和原理

#### 1. DS1302 的工作原理和使用方法如何？

DS1302 是美国 Dallas 公司推出的一种高性能、低功耗的实时时钟芯片，附加 31 字节静态 RAM，采用 SPI 三线接口与 CPU 进行同步通信，并可采用突发方式一次传送多个字节的时钟信号或 RAM 数据。实时时钟可提供秒、分、时、日、星期、月和年，一个月小于 31 日时可自动调整，包括闰年，有效至 2100 年。可采用 12h 或 24h 方式计时，采用双电源（主电源和备用电源）供电，可设置备用电源充电方式，同时提供了对后背电源进行涓细电流充电的能力。芯片为 8 引脚小型 DIP 封装，图 10-40 所示为引脚排列图。

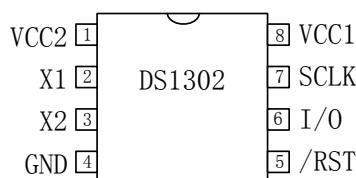


图 10-40 DS1302 引脚排列图

各引脚说明如下。

- X1、X2：连接 32.768kHz 晶振，为芯片提供计时脉冲。
- GND：电源地。
- RST：复位引脚，用于对芯片操作。
- I/O：数据输入、输出引脚。
- SCLK：串行时钟输入。
- V<sub>CC1</sub>、V<sub>CC2</sub>：主电源与后备电源引脚。

要了解 DS1302 的工作原理，首先需要了解其控制字节。DS1302 的控制字节如表 10-16 所示。控制字节的最高有效位（位 7）必须是逻辑 1，如果它为 0，则不能把数据写入到 DS1302 中；位 6 如果为 0，则表示存取日历时钟数据，为 1 表示存取 RAM 数据；位 5~位 1 指示操作单元的地址；最低有效位（位 0）如为 0 表示要进行写操作，为 1 表示进行读操作，控制字节总是从最低位开始输出。

表 10-16 DS1302 的控制字

| 7 | 6          | 5  | 4  | 3  | 2  | 1  |           |
|---|------------|----|----|----|----|----|-----------|
| 1 | RA<br>M/CK | A4 | A3 | A2 | A1 | A0 | RA<br>M/K |

在 DS1302 芯片中，通过把 RST 输入驱动置高电平来启动所有的数据传送。RST 输入有两种功能：首先，RST 接通控制逻辑，允许地址 / 命令序列送入移位寄存器；其次，RST 提供了终止单字节或多字节数据的传送手段。当 RST 为高电平时，所有的数据传送被初始化，

允许对 DS1302 进行操作。如果在传送过程中置 RST 为低电平，则会终止此次数据传送，并且 I/O 引脚变为高阻态。

在控制指令字输入后的下一个 SCLK 时钟的上升沿时数据被写入 DS1302，数据输入从低位即位 0 开始。同样，在紧跟 8 位的控制指令字后的下一个 SCLK 脉冲的下降沿读出 DS1302 的数据，读出数据时从低位 0 位至高位 7，数据读写时序如图 10-41 所示。

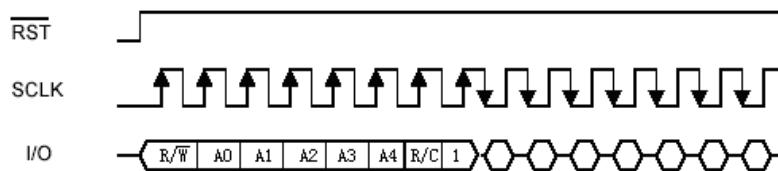


图 10-41 数据读写时序

DS1302 共有 12 个寄存器，其中有 7 个寄存器与日历、时钟相关，存放的数据位为 BCD 码形式。此外，DS1302 还有年份寄存器、控制寄存器、充电寄存器、时钟突发寄存器及与 RAM 相关的寄存器等。时钟突发寄存器可一次性顺序读写除充电寄存器外的所有寄存器内容。DS1302 与 RAM 相关的寄存器分为两类，一类是单个 RAM 单元，共 31 个，每个单元组态为一个 8 位的字节，其命令控制字为 COH~FDH，其中奇数为读操作，偶数为写操作；另一类为突发方式下的 RAM 寄存器，此方式下可一次性读写所有 RAM 的 31 个字节，命令控制字为 FEH（写）、FFH（读）。

## 2. 有哪些可供选择的时钟芯片？

目前市场上可供选择的芯片很多，例如 X1203、HT1380、SD2000 系列等，表 10-17 所示为 Maxim 公司的三线时钟芯片。

表 10-17 MAXIM 公司的三线时钟芯片

| 器件     | 时钟格式   | 工作电压 | 说明                         |
|--------|--------|------|----------------------------|
| DS1302 | BCD    | 2~5V | 涓流充电，电源切换                  |
| DS1305 | BCD    | 2~5V | 时刻/星期闹钟                    |
| DS1306 | BCD    | 2~5V | 时刻/星期闹钟                    |
| DS1602 | BINARY | 5V   | VCC 上电次数计数器                |
| DS1603 | BINARY | 5V   | 包含电池和晶振的模块                 |
| DS1615 | BCD    | 5V   | 时刻/星期闹钟，数字温度计<br>高低温告警     |
| DS1616 | BCD    | 5V   | 时刻/星期闹钟，数字温度计<br>高低温告警     |
| DS1670 | BCD    | 3.3V | 时刻/星期闹钟，数字温度计<br>高低温告警，看门狗 |
| DS1673 | BCD    | 3V   | 时刻/星期闹钟，微处理器复位，看门狗         |
| DS1677 | BCD    | 5V   | 时刻/星期闹钟，微处理器复位，看门狗         |

### 3. 在单片机系统中如何使用 DS1302?

DS1302 与单片机的连接仅需要 3 条线，即 SCLK、I/O、RST。V<sub>CC2</sub>在单电源与电池供电的系统中提供低电源并提供低功率的电池备份。V<sub>CC2</sub>在双电源系统中提供主电源，在这种运用方式下 V<sub>CC1</sub> 连接到备份电源，以便在没有主电源的情况下能保存时间信息以及数据。DS1302 由两者中的较大者供电。当 V<sub>CC2</sub> 大于 V<sub>CC1</sub>+0.2V 时，V<sub>CC2</sub> 给 DS1302 供电。当 V<sub>CC2</sub> 小于 V<sub>CC1</sub> 时，DS1302 由 V<sub>CC1</sub> 供电。

DS1302 在单片机系统中的典型应用电路如图 10-42 所示。图 10-42 为 89C2051 与 DS1302 的接口电路。RST 接在 P1.2 上，此引脚为高位时，选中该芯片，可对其进行操作。串行数据 I/O 与串行时钟线 SCLK 分别接在 P1.0 和 P1.1 上，所有的单片机地址、命令及数据均通过这两条线传输。

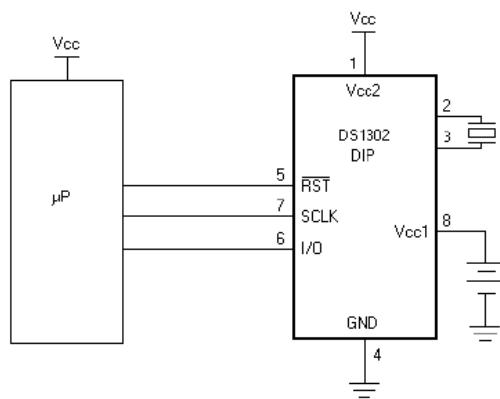


图 10-42 89C2051 与 DS1302 的接口电路

在本系统中，89C2051 为主器件，DS1302 为从器件，主器件在总线上产生时钟脉冲、寻址信号、数据信号等，而从器件则相应接收数据、送出数据。

对 DS1302 的每一次读写需 16 个时钟脉冲，前 8 个脉冲输入操作地址和读写命令，命令格式可以参考 DS1302 的数据手册。其中，Bit7 必须为 1；Bit0 为 0 时向芯片写入数据，为 1 时从芯片读出数据；Bit6~Bit1 选定芯片中的地址。后 8 个脉冲写入或读出数据。以下就给出具体的单片机 89C2051 和 DS1302 的电路和 C51 程序。

### 10.6.3 电路

本例中的主要器件为单片机 89C2051 和 DS1302，其接口电路如图 10-43 所示。

#### 1. 电路原理和器件选择

在这里列出和本例相关的、关键部分的器件名称及其在电路中的主要功能。

- 89C2051：主要通过对 DS1302 的设置，完成时钟的设置以及数据传输。
- DS1302：时钟芯片，在单片机的控制下实现时钟数据记录和输出。
- OSC2：32.768kHz 的晶振，为时钟芯片提供计时脉冲。

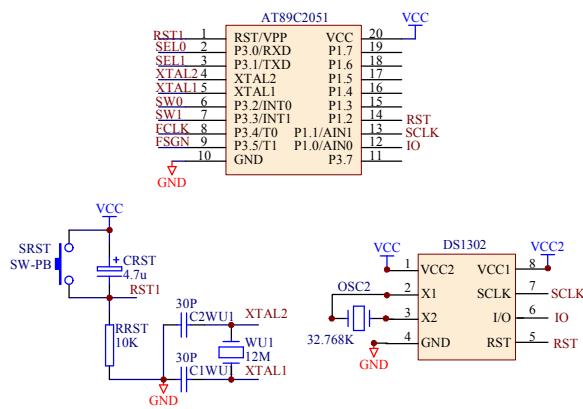


图 10-43 单片机 89C2051 和 DS1302 的接口电路

## 2. 地址分配和连接

只列出和本例相关的、关键部分的单片机与各个功能管脚的连接和相关的地址分配。

- RST: 复位引脚, 和单片机的 P1.2 口相连, 用于对芯片操作。
- IO: DS1302 的数据输入、输出引脚, 和单片机的 P1.0 引脚相连。
- SCLK: 串行时钟输入。和单片机的 P1.1 引脚相连。
- VCC2: 后备电源引脚。
- VCC: 电源引脚, 同时也是单片机系统的电源。
- RST1: 单片机的复位电路引脚。

## 10.6.4 程序设计

### 1. 程序功能

单片机控制 DS1302 时钟芯片的程序主要包括两个方面的关键内容, 一是单片机对 DS1302 寄存器的地址定义和控制字的写入, 二是数据的读取。

### 2. 主要器件和变量的说明

本例中所使用的主要功能器件就是 DS1302 时钟芯片。程序中的变量及功能如表 10-18 所示。

表 10-18

变量及功能

| 变量         | 功能             |
|------------|----------------|
| P1.0       | I/O 引脚         |
| P1.1       | SCLK 引脚        |
| P1.2       | RST 引脚         |
| addr       | 传输的地址          |
| d          | 传输的数据          |
| readclk()  | 读 DS1302 数据程序  |
| writeclk() | 写入 DS1302 数据程序 |

### 3. 程序代码

单片机控制的 DS1302 时钟数据通信程序流程如图 10-44 所示，代码如下。

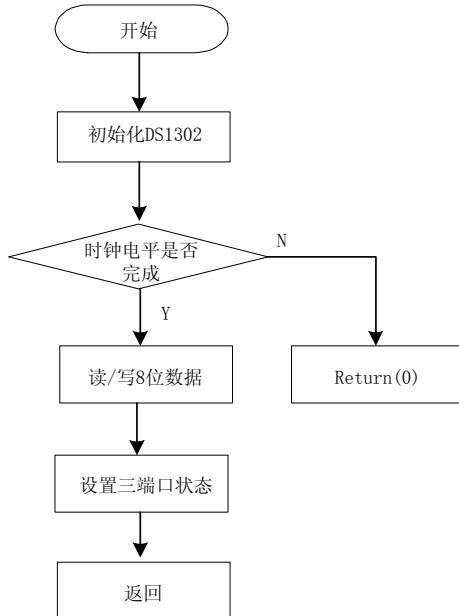


图 10-44 单片机控制的 DS1302 的时钟数据通信流程图

```

#include <reg51.h>
#include <stdio.h>
#define uchar unsigned char
#define uint unsigned int

sbit P1_0=P1^0; // 引脚连接关系
sbit P1_1=P1^1;
sbit P1_2=P1^2;

uchar readclk(uchar addr)
{
 uchar i,k;
 P1_1=0;
 addr=1;
 P1_2=1;
 k=1;
 for (i=0;i<8;i++) // 按时序读 RAM 中的数据
 {
 if (addr&k)P1_1=1;
 else P1_1=0;
 }
}

```

```
P1_1=1;
k<<=1;
P1_1=0;
}

k=0;
P1_1=1;
for (i=0;i<8;i++)
{
 k>>=1;
 P1_1=0;
 if(P1_1) k|=0x80;
 P1_1=1;
}

P1_2=0;
return k;
}

void writeclk(uchar addr, uchar d)
{
 uchar i,k;
 P1_1=0; // clk
 P1_2=1; // 片选
 addr&=0xfe;
 k=1;
 for (i=0;i<8;i++)
 {
 if (addr&k) P1_1=1; // 按时序写 RAM 中的数据
 else P1_1=0;
 P1_1=1;
 k<<=1;
 P1_1=0;
 }
 k=1;
 for (i=0;i<8;i++)
 {
 if(d&k) P1_1=1;
 else P1_1=0;
 P1_1=1;
 }
}
```

```

k<<=1;
P1_1=0;
}
P1_2=0
}

void main(void)
{
 unsigned char addr,d;
 addr=0;
 d=readclk(0xc0|addr); //读内部 RAM 0
 addr=1; //读内部 RAM 1
 d=readclk(0xc0|addr);
 address=0;
 d=5; //写内部 RAM 0, 写入 5
 writeclk(0xc1|addr,d);
 addr=1;
 d=123; //读内部 RAM 1, 写入 123
 d=readclk(0xc1|addr,d);
 address=0;
 d=3; //写内部时钟日历数据 0: 秒, 写入 03 秒
 writeclk(0x80|addr,d);
}

```

## 10.7 单片机中滤波算法的实现

在本例中，介绍一些简单的滤波算法，因为尽管单片机并不擅长实现算法和进行复杂的运算，但是在一些特定的应用场合，如实时的动态测量等应用上还是需要实现一定的滤波算法。由于本例主要是介绍并给出实例算法，没有电路设计要求，所以本例的章节安排和其他实例有所区别。

### 1. 单片机中实现滤波算法有何好处？

在单片机进行数据采集时，会遇到数据的随机误差。随机误差是由随机干扰引起的，其特点是在相同条件下测量同一量时，其大小和符号作无规则变化而无法预测，但多次测量结果符合统计规律。为克服随机干扰引起的误差，硬件上可采用滤波技术；软件上可采用软件算法实现数字滤波，其算法往往是系统测控算法的一个重要组成部分，实时性很强，常采用汇编语言来编写。

采用数字滤波算法克服随机干扰引起的误差具有以下优点。

(1) 数字滤波无需硬件，只用一个计算过程，可靠性高，不存在阻抗匹配问题。尤其是

数字滤波可以对频率很高或很低的信号进行滤波，这是模拟滤波器做不到的。

(2) 数字滤波是用软件算法实现的，多输入通道可共用一个滤波程序，降低系统开支。

(3) 只要适当改变软件滤波器的滤波程序或运算参数，就能方便地改变其滤波特性，这对于低频干扰、随机信号的滤波会有较大的效果。

## 2. 在单片机系统中有哪些常用的滤波算法？

常用的数字滤波器算法有程序判断法、中值判断法、算术平均值滤波法、加权滤波法、滑动滤波法、低通滤波法和复合滤波法等。以下就分别介绍其原理和滤波程序。

### (1) 程序判断法

程序判断法又称限幅滤波法，其方法是把两次相邻的采样值相减，求出其增量，以绝对值表示，然后与两次采样允许的最大差值 $\Delta Y$ 进行比较。 $\Delta Y$ 的大小由被测对象的具体情况而定，若小于或等于 $\Delta Y$ ，则取本次采样值；若大于 $\Delta Y$ ，则取上次采样值作为本次采样数据的样本，其基本原理可用如下公式表示：

$$|y_n - y_{n-1}| \leq \Delta y ; \text{ 则 } y_n \text{ 有效;}$$

$$|y_n - y_{n-1}| > \Delta y ; \text{ 则 } y_{n-1} \text{ 有效。}$$

式中的 $y_n$ 为第 $n$ 次采样数据， $y_{n-1}$ 为第 $(n-1)$ 次的采样数据， $\Delta y$ 为相邻两次采样数据允许的最大偏差。此算法的样例子程序如下：

```
#define A
char data; // 上一次的数据
char filter_1()
{
 char datanew; // 新数据变量
 datanew = get_data(); // 获得新数据
 if((datanew - data > A) || (data - datanew > A)) // 滤波算法
 return data;
 return datanew;
}
```

程序判断法主要用于处理变化比较缓慢的数据，如温度、物体的位置等。使用时，关键是要选取合适的门限值 $\Delta y$ 。通常可根据经验数据获得，必要时也可由实验得到。

### (2) 中值滤波法

中值滤波法即对某一参数连续采样 $N$ 次， $N$ 一般为奇数，然后把 $N$ 次采样值按从小到大排队，再取中间值作为本次采样值。实际上，该滤波方法也是一种排序方法，具体的样例程序如下：

```
#define N 11
char filter_2()
{
 char value_buf[N];
```

```

char count,i,j,temp;
for (count=0;count<N;count++)
{
 value_buf[count] = get_data(); // 获取数据
 delay();
}
for (j=0;j<N-1;j++)
{
 for (i=0;i<N-j;i++)
 {
 if (value_buf[i]>value_buf[i+1])
 {
 temp = value_buf[i];
 value_buf[i] = value_buf[i+1];
 value_buf[i+1] = temp;
 }
 }
}
return value_buf[(N-1)/2];
}

```

中值滤波法比较适用于去掉由偶然因素引起的波动和采样器不稳定而引起的脉动干扰。若被测量变化比较缓慢，采用中值滤波法效果比较好；但对快速变化的数据，则不宜采用中值滤波。

### (3) 算术平均滤波法

算术平均滤波法就是连续取  $N$  次采样值进行算术平均，其数学表达式如下：

$$y = \sum y_i$$

$$\bar{y} = \frac{1}{N} \sum_{i=1}^N y_i$$

具体的算法子程序如下：

```

char filter_3()
{
 int sum = 0;
 for (count=0;count<N;count++)
 {
 sum += get_ad();
 delay();
 }
}

```

```

 return (char)(sum/N);
}

```

算术平均滤波法适用于对具有随机干扰的信号进行滤波。这种信号的特点是有一个平均值，信号在某一数值附近上下波动。信号的平滑程度完全取决于  $N$ 。当  $N$  较大时，平滑度高，但灵敏度低；当  $N$  较小时，平滑度低，但灵敏度高。为方便求平均值， $N$  一般取 4、8、16 之类的 2 的整数幂，以便于用移位来代替除法。

#### (4) 加权平均滤波法

算术平均滤波法存在前面所说的平滑性和灵敏度之间的矛盾。采样次数太少，平滑效果差；次数太多，灵敏度下降，对参数的变化趋势不敏感。为协调两者关系，可采用加权平均滤波。对连续  $N$  次采样值分别乘上不同的加权系数之后再求累加和，加权系数一般先小后大，以突出后面若干采样的效果，加强系统对参数变化趋势的辨识。各个加权系数均为小于 1 的小数，且满足总和等于 1 的约束条件。这样，加权运算之后的累加和即为有效采样值。

为方便计算，可取各加权系数均为整数，且总和为 256，加权运算后的累加和除以 256，即舍去低字节后便是有效采样值。具体的样例子程序如下：

```

char code jq[N] = {1,2,3,4,5,6,7,8,9,10,11,12}; //coe 数组为加权系数表，存在程序存储区。
char code sum_jq = 1+2+3+4+5+6+7+8+9+10+11+12;

char filter_4()
{
 char count;
 char value_buf[N];
 int sum=0;
 for (count=0,count<N;count++)
 {
 value_buf[count] = get_data();
 delay();
 }
 for (count=0,count<N;count++)
 sum += value_buf[count]*jq[count];
 return (char)(sum/sum_jq);
}

```

#### (5) 滑动平均滤波法

以上介绍的各种平均滤波算法有一个共同点，即每取得一个有效采样值必须连续进行若干次采样，当采样速度较慢，如双积分型 A/D 转换，或目标参数变化较快时，系统的实时性不能得到保证。滑动平均滤波算法只采样一次，将这一次采样值和过去的若干次采样值一起求平均，得到的有效采样值即可投入使用。如果取  $N$  个采样值求平均，RAM 中必须开辟  $N$  个数据的暂存区。每新采集一个数据便存入暂存区，同时去掉一个最老的数据，保持这  $N$  个数据始终是最近的数据。这种数据存放方式可以用环行队列结构方便地实现。具体的滤波程序如下：

```

char value_buf[N];
char i=0;

char filter_5()
{
 char count;
 int sum=0;
 value_buf[i++] = get_ad();
 if (i == N) i = 0;
 for (count=0; count<N, count++)
 sum = value_buf[count];
 return (char)(sum/N);
}

```

### (6) 低通滤波法

将普通硬件 RC 低通滤波器的微分方程用差分方程来表示，便可以用软件算法来模拟硬件滤波的功能。经推导，低通滤波算法为： $Y_n = a \cdot X_n + (1-a) \cdot Y_{n-1}$ ，式中的参数如下：

$X_n$ ：当前的数据。

$Y_{n-1}$ ：上次的滤波输出值。

$a$ ：滤波系数。

$Y_n$ ：本次滤波的输出值。

由上式可以看出，本次滤波的输出值主要取决于上次滤波的输出值，这和加权平均滤波是有本质区别的，本次采样值对滤波输出的贡献是比较小的，但多少有些修正作用。这种算法便模拟了具有较大惯性的低通滤波功能。当目标参数为变化很慢的物理量时，这是很有效的。另一方面，它不能滤除高于 1/2 采样频率的干扰信号，本例中采样频率为 2Hz，故对 1Hz 以上的干扰信号应采用其他方式滤除。

低通滤波算法程序与加权平均滤波相似，但加权系数只有两个： $a$  和  $1-a$ 。因为只有两项， $a$  和  $1-a$  均以立即数的形式编入程序中，不另设表格。虽然采样值为单元字节，为保证运算精度，滤波输出值用双字节表示，其中一字节整数，一字节小数，否则有可能因为每次舍去尾数而使输出不会变化。低通滤波的程序如下：

```

char filter_6()
{
 char new_value;
 new_value = get_data();
 return (100-a)*value + a*new_value;
}

```

### (7) 中位值平均滤波

中位值平均滤波方法又称为防脉冲干扰平均值滤波算法。它是把两种不同滤波功能的数字滤波器组合起来，组成复合数字滤波器，或称多级数字滤波器。这种算法的特点是，用中值滤波算法滤掉采样值中的脉冲干扰，然后把剩余的各采样值进行递推平均滤波。其基本算法如下：

$$y_1 \leq y_2 \leq \dots \leq y_n, \text{ 其中, } 3 \leq N \leq 4;$$

$$\bar{y} = (y_2 + y_3 + \dots + y_{n-1}) / (N - 2)$$

中位值平均滤波算法的程序如下：

```
char filter_7()
{
 char count,i,j;
 char value_buf[N];
 int sum=0;
 for (count=0;count<N;count++)
 {
 value_buf[count] = get_data();
 delay();
 }
 for (j=0;j<N-1;j++)
 {
 for (i=0;i<N-j;i++)
 {
 if (value_buf[i]>value_buf[i+1])
 {
 temp = value_buf[i];
 value_buf[i] = value_buf[i+1];
 value_buf[i+1] = temp;
 }
 }
 }
 for(count=1;count<N-1;count++)
 sum += value[count];
 return (char)(sum/(N-2));
}
```

由于这种滤波方法兼容了递推平均滤波算法和中值滤波算法的优点，所以无论对缓慢变化的信号，还是对快速变化的信号，都能取得较好的滤波效果。

## 10.8 信号数据的 FFT 变换

在介绍完滤波算法后，在本例中将介绍数据处理中需要使用到的傅立叶变换（FFT）和相关的数据插值计算在单片机中的使用方法。与本章 10.7 节的案例一样，由于本例主要是介绍实例算法，没有特别的电路要求，所以本例安排采取 10.7 节案例的方案。

### 1. 傅立叶变换算法的基本原理是什么？

FFT/IFFT 在数字信号处理中是一种非常重要的算法，在相当多的应用领域，如调制器、数字电视、手持无线接收装置等，FFT 处理器已经是其中的关键部件。在数字图像匹配应用中，可以先应用 FFT/IFFT 处理器把图像变换到频域，这样就可以将计算两幅图像相似度所需的二维卷积运算转换成直积运算，从而大大减少处理的运算量。

FFT 处理器按照蝶形运算单元的组织分布不同可分成 4 类：顺序处理、流水线处理、并行处理和阵列处理。阵列处理和并行处理对硬件设备的需求较大，不适于嵌入式应用实现。流水线处理方案能够适应多批串行数据 FFT 处理的需要，在通信领域有很多应用，但是结构不灵活，较难满足系统中不同点长度 FFT 运算的需要，只能根据应用的需求专门设计。顺序处理最容易在单芯片系统上实现，但对于低基的顺序处理 FFY 处理器来说处理时间太长，较难满足实时系统对性能的追求。如果采用高基的 FFT 处理器方案，虽然速度上可能能够满足需要，但结构又过于复杂。

FFT 是离散 FOURIER 变换的快速算法，两者没有本质的区别。对于任何一个周期信号：

$$v(t) = \sum_{n=1}^{\infty} A_n \cos(2\pi nft + \varphi_n), \text{ 在 } [0, T] \text{ 内等分 } N \text{ 等分采样数据，其离散的 FOURIER}$$

$$\text{变换为: } F(j\omega_k) = \sum_{n=0}^{N-1} v(t_n) e^{-j\omega_k t_n}.$$

在实际应用中，可以采用倒序位算法实现 FFT 变换。按时间抽取的 FFT 算法通常将原始数据倒序存储，最后按正常顺序输出结果 X(0)、X(1)、...、X(k)、...。数据在数组 dataR[128] 中，程序段如下（假设 128 点 FFT）：

```
void CHANGE()
/* i 为原始存放位置，最后得 invert_pos 为倒序存放位置 */
int b0=b1=b2=b3=b4=b5=b6=0;
/*以下语句提取各位的 0、1 值*/
b0=i&0x01;
b1=(i/2)&0x01;
b2=(i/4)&0x01;
b3=(i/8)&0x01;
b4=(i/16)&0x01;
b5=(i/32)&0x01;
b6=(i/64)&0x01;
```

```
all=x0*64+x1*32+x2*16+x3*8+x4*4+x5*2+x6;
```

在实际的单片机编程中，可以通过实数蝶形运算算法推导 FFT 的结果。首先需要了解如图 10-45 所示的蝶形图。

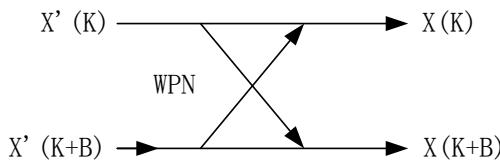


图 10-45 蝶形运算算法

蝶形公式：

$$X(K)=X'(K)+X'(K+B)WPN,$$

$$X(K+B)=X'(K)-X'(K+B)WPN$$

其中， $WPN=\cos(2\pi P/N) - j\sin(2\pi P/N)$ 。

设：

$$X(K+B)=XR(K+B)+jXI(K+B),$$

$$X(K)=XR(K)+jXI(K),$$

有：

$$XR(K)+jXI(K)=XR'(K)+jXI'(K)+[XR'(K+B)+jXI'(K+B)]*\cos(2\pi P/N) - j\sin(2\pi P/N);$$

继续分解得到下列两式：

$$XR(K)=XR'(K)+XR'(K+B)\cos(2\pi P/N) + XI'(K+B)\sin(2\pi P/N)$$

$$XI(K)=XI'(K)-XR'(K+B)\sin(2\pi P/N) + XI'(K+B)\cos(2\pi P/N)$$

需要注意的是： $XR(K)$ 、 $XR'(K)$ 的存储位置相同，所以经过变换后，该位置上的值已经改变，而下面求  $X(K+B)$ 要用到  $X'(K)$ ，因此在编程时要注意保存  $XR'(K)$  和  $XI'(K)$  到 TR 和 TI 两个临时变量中。蝶形运算程序段如下所示，其中 sin 和 cos 函数值直接查表得到。

```
void FFT()
{
 TR=dataR[k];
 TI=dataI[k];
 temp=dataR[k+b];
 dataR[k]=dataR[k]+dataR[k+b]*cos_tab[p]+dataI[k+b]*sin_tab[p];
 dataI[k]=dataI[k]-dataR[k+b]*sin_tab[p]+dataI[k+b]*cos_tab[p];
 dataR[k+b]=TR-dataR[k+b]*cos_tab[p]-dataI[k+b]*sin_tab[p];
 dataI[k+b]=TI+temp*sin_tab[p]-dataI[k+b]*cos_tab[p];
}
```

## 2. 单片机的傅立叶变换算法程序如何实现？

鉴于目前在许多嵌入式系统中要用到 FFT 运算，如以 DSP 为核心的交流采样系统、频谱分析、相关分析等。本例引用了实数的 FFT 算法并给出具体的 C51 语言函数，读者可以借鉴此程序，应用自己的系统中。

```
//128 点 FFT 函数:
/* 采样来的数据放在 dataR[] 数组中，运算前 dataI[] 数组初始化为 0 */
#include <reg52.h>
```

```

#define uint unsigned int
#define uchar unsigned char
uint x0,x1,x2,x3,x4,x5,x6;
int L,i,j,k,b,p;
uint TR,TIs,temp;
uchar w[3];
void FFT(int dataR[],int dataI[])
{
 // 以下是数据掉头功能
 for(i=0;i<128;i++)
 {
 CHANGE();
 }
 for(i=0;i<128;i++)
 {
 dataR[i]=dataI[i]; dataI[i]=0;
 }
 // FFT 算法
 for(L=1;L<=7;L++) // 第一层循环
 {
 b=1; i=L-1;
 while(i>0)
 {
 b=b*2; i--;
 }
 for(j=0;j<=b-1;j++) // 第二层循环
 {
 p=1; i=7-L;
 while(i>0)
 {p=p*2; i--; }
 p=p*j;
 for(k=j;k<128;k=k+2*b) // 第三层循环
 {
 FFT(dataR,dataI);
 }
 }
 }
 for(i=0;i<32;i++) // 32 次以下的谐波分析
 {
 w[i]=sqrt(dataR[i]*dataR[i]+dataI[i]*dataI[i]);
 }
}

```

```
w[i]=w[i]/64;
}
w[0]=w[0]/2;
}
```

### 3. 浮点数的运算问题如何解决？

常用的 FFT 程序中会出现浮点数的运算。而单片机中并没有可用的浮点计算例程。在上面的程序中定义了如下变量。

```
#define uint unsigned int
uint TR,TI,temp;
```

实际上，在常用的 C 语言的算法中，应当定义如下：

```
float TR,TI,temp;
```

由于单片机中没有现成的浮点运算，这里做了简化，主要是介绍 FFT 的方法。如果需要进行浮点运算，那么可以参考其他相关书籍。这里简单地介绍一下单片机中浮点数的运算原理。

MCS-51 单片机在实时测控系统、智能仪表等各种系统中已得到广泛应用，在各种实际应用系统中，浮点数的变制就是要解决的主要问题。

为了编程简单，在进行变换之前，对变换数取绝对值，将符号保护起来，然后按正数进行变换，变换结束后再恢复数的符号。在进行浮点数的数制转换时，涉及到十进制浮点数、十进制定点数、二进制浮点数及二进制定点数的表示和转换。在浮数的计算过程中，为了获得足够的精确度，可增加尾数的数目。为了增加动态范围，可以增加阶码的数目。该程序在设计时考虑到精度要求，尾数字节数可以根据要求增或减。

- 十进制变为二进制

由于浮点数的尾数是纯小数。所以首先将十进制小数变换为二进制小数，我们很容易将十进制小数变为二进制小数。再对变换后的二进制小数进行规格化，最后根据十进制浮点数的阶码用“10”或“0.1”去乘以规格化的二进制浮点数，乘积结果就是十进制浮点数的变换位。

- 二进制变为十进制

根据二进制浮点数表示的数是大于 1 还是小于 1，用 0.1 或 10 去乘这个数，同时对十进制数的阶码计数加 1，再判断乘积结果是大于 1 还是小于 1，重复上述的过程直到二进制浮点数在 0.1 与 1 之间为止。最后将尾数变换为十进制小数，即得到了十进制的尾数。

- 十进制浮点格式

设十进制浮点数放在内部 RAM 38H 为首址的 4 个字节中。尾数为纯小数用 5 位 BCD 码原码表示，尾数低字节的低 4 位无效，第一字节的 D7 为数的符号，D6 为阶码的符号。数符、阶符均用“1”表示负，“0”表示正，阶码以二进制原码表示。

- 二进制浮点数格式

第一个字节 D7 为数的符号，D6 为阶码的符号，“1”表示负，“0”表示正。D5~D0 为阶码，阶码以补码表示。尾数为两个字节，以原码表示。

#### 4. 如果数据不足该怎么办?

在进行 FFT 变换时,常常由于干扰和波动而造成数据的丢失和错误,从而造成 FFT 变换的关键数据缺乏,这时就需要采用插值方法获得所需要的数据。

常用的插值方法很多,也有很多的参考资料,这里引用两个现成的插值算法(参见《单片机程序设计基础》),分别是线性插值算法和抛物线插值算法。算法的详细解释可参看相关的资料。

##### (1) 线性插值法

```
//线性插值算法
uint x0=1,y0=2; //第一个已知点的坐标
uint x1=1,y1=3; //第二个已知点的坐标
uint line (float x) //“点斜式”线性插值算法
{
 return (y0+(y1-y0)*(x-x0)/(x1-x0));
}

main()
{
 float x,y;
 x=1;
 y=line(x);
 x=2;
 y=line(x);
 while (1);
}
```

##### (2) 抛物线插值法

```
//抛物线插值算法
uint x0=1,y0=2; //第一个已知点的坐标
uint x1=1,y1=3; //第二个已知点的坐标
uint x2=1,y2=3; //第三个已知点的坐标

uint line1 (float x) //过第一点和第二点的“点斜式”线性插值算法
{
 return (y0+(y1-y0)*(x-x0)/(x1-x0));
}

uint line2 (float x) //过第一点和第三点的“点斜式”线性插值算法
{
 return (y0+(y2-y0)*(x-x0)/(x2-x0));
}

uint qins (float x) //以线性插值算法为基础的抛物线插值算法
```

```
{
 return (line1(x)+(line2(x)-line1(x))*(x-x1)/(x2-x1));
}

main()
{
 float x,y;
 x=1;
 y=qins(x);
 x=2;
 y=qins(x);
 while (1);
}
```