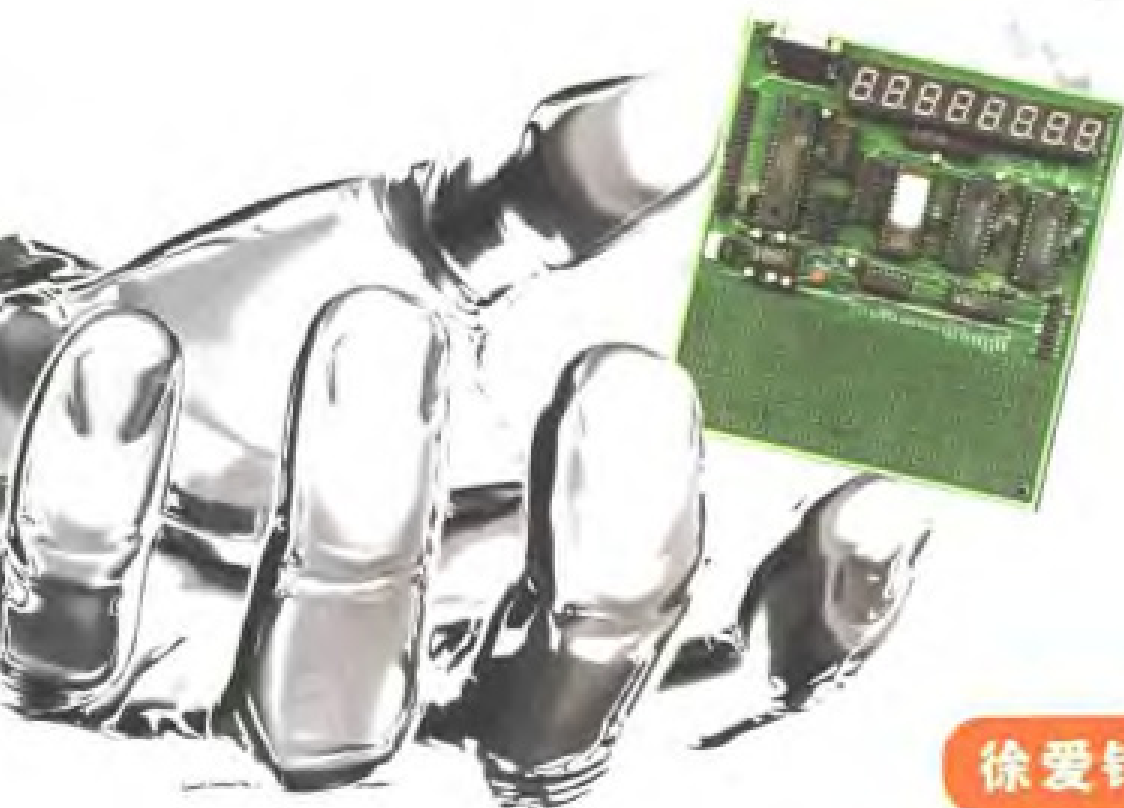


Broadview
WWW.BROADVIEW.COM.CN



徐爱钧 彭秀华 编著

Keil Cx51 V7.0 单片机高级语言编程 与 μ Vision2 应用实践



电子工业出版社

地址: 北京市西城德胜门内大街2号
邮编: 100088
电话: (010) 68254234
网址: <http://www.phot.com.cn>

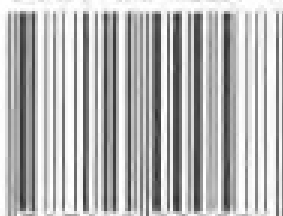


本书详细介绍了 Keil Cx51 V7.0 版本单片机 C 语言编译器和全新 Windows 集成开发环境 μ Vision2 的强大功能和具体使用方法。对最新版本 Cx51 编译器新增加的控制命令作了全面介绍,给出了全部 Cx51 运行库函数及其应用范例。

本书作者曾以 Keil 公司较早期的 C51 编译器 V5.0 版本为核心编写了《单片机高级语言 C51 Windows 环境编程与应用》一书,出版后受到广大读者的欢迎,已经多次重印。

本书的特点是强调先进性和实用性,给出了大量程序实例,并带有一张由 Keil 公司提供的配套 CD-ROM 光盘,其中包括 Keil Cx51 V7.0 版本全功能评估软件包,以及本书各章的范例程序代码。

ISBN 7-120-00057-8



9 787120 000578 >



责任编辑:毕宁

封面设计:张子建

本书贴有激光防伪标志,凡没有防伪标志者,属盗版书。
ISBN 7-120-00057-8 定价:69.00 元(含光盘1张)

Keil Cx51 V7.0 单片机高级 语言编程与 μ Vision2 应用实践

徐爱钧 彭秀华 编著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书详细介绍了 Keil Cx51 V7.0 版本单片机 C 语言编译器和全新 Windows 集成开发环境 μ Vision2 的强大功能和具体使用方法。全面介绍了最新版本 Cx51 编译器新增加的控制命令,给出了全部 Cx51 运行库函数及其应用范例,对 Keil Cx51 软件包中各种应用工具,如 BL51/Lx51 连接定位器、A51/Ax51 宏汇编器、LIB51 库管理程序以及 OH51 符号转换程序等都作了详细介绍,还介绍了单片机实时多任务操作系统 RTX51 及其子集 RTX51 TINY 的具体功能与应用方法。 μ Vision2 已经将调试器功能集成于其中,用户可以在单一环境下完成从源程序编写、编译、链接定位一直到目标文件的仿真调试等全部工作,书中详细介绍了 μ Vision2 各种功能和应用,包括软件模拟调试和硬件目标板实时在线仿真。

本书的特点是强调先进性和实用性,给出了大量程序实例,并带有一张由 Keil 公司提供的配套 CD-ROM 光盘,其中包括 Keil Cx51 V7.0 版本全功能评估软件包,本书各章中列出的全部程序代码,以及 Keil Cx51 源程序仿真硬件目标板的照片和使用说明。本书适合于从事单片机应用系统开发研制的广大工程技术人员阅读,也可以作为高等院校相关专业大学生或研究生的教学参考书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

Keil Cx51 V7.0 单片机高级语言编程与 μ Vision2 应用实践 / 徐爱钧, 彭秀华编著. —北京: 电子工业出版社, 2004.6

ISBN 7-120-00057-8

I. K… II. ①徐… ②彭… III. 单片微型计算机—程序设计 IV. TP368.1

中国版本图书馆 CIP 数据核字 (2004) 第 046879 号

责任编辑: 毕 宁 bn@phei.com.cn

印 刷: 北京智力达印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

经 销: 各地新华书店

开 本: 787×1092 1/16 印张: 45.25 字数: 1077 千字

印 次: 2004 年 6 月第 1 次印刷

印 数: 6000 册 定价: 69.00 元 (含光盘 1 张)

凡购买电子工业出版社的图书,如有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系。联系电话: (010) 68279077。质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

前 言

8051 单片机是目前国内外工业测量控制领域内使用极为广泛的一类 8 位微控制器,它的特点是使用方便灵活,外围硬件支持十分丰富,世界上许多大半导体厂商,如 Atmel、Analog Device、Dallas、Infineon、Philips、SST、TI 等公司都推出了具有各自特点的增强型 8051 系列单片机,使用户有了更大的选择范围。另外,世界上许多软件公司还都致力于 8051 单片机高级语言编译器的开发研究,使用户得以采用高级语言编程,从烦琐的汇编语言中解脱了出来。C 语言是一种特别适合于开发计算机操作系统的高级语言,德国 Keil 公司在开发单片机 C 语言编译器方面取得了相当大的成功,从编译器、调试器、实时操作系统到集成开发环境,全面支持 8051、251、166 等单片机主流产品及其众多的派生系列。

本书作者曾以 Keil 公司较早期的 C51 编译器 V5.0 版本为核心编写了《单片机高级语言 C51 Windows 环境编程与应用》一书,出版后受到广大读者的欢迎,已经多次重印。V5.0 版的 Keil C51 开发环境实际上是从 DOS 向 Windows 的一种升级,它在集成度和易用性方面还不尽如人意,如不支持长文件名、编译选项的设定较为烦琐、与 Windows 环境的兼容性不够完善等。鉴于以上因素 Keil 公司于 1999 年发表了 V6.0 版本的 C51 编译器,同时推出了一种全新的 Windows 集成开发环境 μ Vision2。最近 Keil 公司又将 C51 编译器升级到 V7.0 版本,使之能够完全支持 Philips 公司新推出的超大容量内存单片机 80C51Mx,并命名为 Cx51 编译器,它是目前最高效的、灵活的 8051 开发平台,可以支持所有 8051 的衍生产品,同时支持第三方开发工具。

V7.0 版本的 Cx51 编译器具有如下一些优越性。

1. C51 源程序经过优化编译后生成的代码,其效率接近于汇编语言生成的代码。
2. 支持所有 8051 系列单片机,提供对所有外围硬件部件的操作。
3. 无论在有无工作寄存器区转换的情况下,Cx51 都能产生快速中断代码。
4. 支持 Atmel, Dallas, Infineon, Philips 和 Temic 等公司 8051 衍生产品上的双数据指针及高速算术单元 (arithmetic units)。
5. 能够在整个应用程序中执行全局寄存器优化。
6. 所有应用工具均可产生详细警告信息和错误信息,帮助用户处理难于寻找的问题。
7. 支持再入功能和寄存器区的独立代码,便于中断服务程序和多任务应用程序的执行。
8. 采用分组方式执行代码分组和调试,用户程序代码可以轻松突破 64KB 空间的限制。

Keil Cx51 V7.0 版本编译器的 Windows 环境兼容性以及集成度得到极大的提升,Cx51 已被完全集成到一个功能强大的全新集成开发环境 μ Vision2 中,它将项目 (project) 管理器、Cx51 编译器、Ax51 汇编器、BL51/Lx 连接定位器、RTX51 实时操作系统、Simulator

模拟器以及 Monitor51 目标调试器的功能全部集成在单一而灵活的 μ Vision2 中, 为用户提供极为简便的操作环境。 μ Vision2 具有强大的项目管理功能, 项目中包括源程序文件、开发工具选项以及编程说明等, 一个 μ Vision2 项目能够产生一个或多个目标文件。 μ Vision2 内部集成了一个器件数据库 (device database), 其中储存了各种型号单片机的片上存储器和集成外围功能信息, 通过器件数据库可以自动设置 Cx51 编译器、Ax51 汇编器、BL51/Lx51 连接定位器、Debug 调试器等开发工具的功能选项, 充分满足用户使用特定单片机的要求。 μ Vision2 还可以为片外存储器产生必要的连接选项, 确定应用程序的起始地址及代码大小规模。 μ Vision2 集成开发环境具有如下一些新的功能特点。

1. 简便易用的集成开发环境, 源级浏览器 (Browser) 利用符号数据库使用户可以快速浏览源文件, 用详细的符号信息来优化用户变量存储器。

2. 内嵌源程序文本编辑器, 允许用户在编辑源程序时 (甚至在未经编译和汇编之前) 设置程序调试断点, 便于在调试程序时快速检查和修改程序。

3. 文件查找功能, 能在当前文件或全局文件中搜索指定的文本信息。

4. 集成仿真调试功能, 允许在统一的 μ Vision2 环境下进入 Debug 调试器, 提供纯软件模拟仿真调试 (Simulator) 和硬件目标板在线仿真调试 (Monitor-51), 可以进行应用程序代码综合性能分析和代码覆盖率分析。

5. 单一而灵活的集成开发环境, 可以对所有开发工具的功能选项进行必要的设置。同时提供一个可配置的工具菜单, 允许在 μ Vision2 环境下启动用户功能。

6. 提供对软件版本控制系统 SVCS 的接口、对应用程序代码进行深层语法分析的 PC-LINT 接口以及对便于生成含有 N-S 流程的块集代码 EasyCase 编辑器的接口。

本书是作者在对最新版本 Keil Cx51 专业开发软件包全面了解和熟练应用的基础上编写而成的, 力求先进性和实用性, 不仅完整地介绍了 V7.0 版本 Keil Cx51 开发工具的功能和使用方法, 还给出了大量具体应用实例, 所有程序例子都在 μ Vision2 环境下调试通过。全书共分 12 章, 第 1 章阐述了 8051 单片机的存储器结构, 以及在 μ Vision2 集成开发环境下进行 C51 程序设计和编译调试的基本方法。第 2 章~第 6 章阐述了 8051 单片机 C 语言编程的基础知识, 以及 Keil Cx51 对 ANSI C 的扩展。第 7 章阐述了 μ Vision2 集成开发环境, 详细介绍了工作环境、目标程序的仿真调试以及各种开发工具应用选项的设置方法。第 8 章阐述了在 μ Vision2 环境下进行 C51 应用程序设计的技巧, 并通过大量具体实例详细讨论了许多 C51 实际应用中的常见问题和解决方法。第 9 章阐述了 V7.0 版本 Keil Cx51 编译器, 详细介绍了各种编译控制命令、数据调用协议, 给出了全部库函数及其应用范例。第 10 章阐述了 Ax51 宏汇编器, 详细介绍了各种符号与表达式、汇编伪指令、宏处理以及汇编控制命令。第 11 章阐述了 BL51/Lx51 连接定位器与实用工具, 详细介绍了 BL51/Lx51 的各种连接控制命令、符号转换程序 OH51、库管理程序 LIB51 的使用方法。第 12 章阐述了 RTX51 实时多任务操作系统及其子集 RTX51 TINY 的功能和具体使用方法。

为帮助读者更好地学习掌握 Keil Cx51 进行应用程序设计, 本书带有一张由 Keil 公司提供的配套 CD-ROM 光盘, 其中包含 Keil 公司最新版本的全功能 Cx51 评估软件包和本书各章所有范例程序代码。另外, 我们还设计并制作了一种能与本书介绍的 μ Vision2 环境配套使用的 Keil Cx51 源程序仿真硬件目标板, 可以将用户程序下载到评估板中直接进行硬件目标系统仿真调试, 节省购买专用仿真器的费用。关于硬件评估板的使用方法书中作

了详细介绍，光盘中还附有 Keil Cx51 源程序仿真硬件目标板的照片。

本书在编写出版过程中得到电子工业出版社的热情支持，并得到了福坤、燕满、张玉芳、郭邦云、王珍、邓骊、冯金栋、易金生、商杰、袁晓莉、陈召军、李进、帖伟鹏等同志的协助，在此表示衷心感谢。由于作者水平有限，书中难免会有错误和不妥之处，恳请广大读者批评指正，读者可通过电子邮件：ajxu@tom.com，ajxu41@sohu.com 直接与作者联系。

徐爱钧 2004 年 4 月

目 录

第 1 章 8051 单片机与 Keil Cx51 基础	1
1.1 8051 单片机的存储器组织结构	1
1.2 Keil Cx51 开发工具	6
1.3 Cx51 简单编程与调试	6
第 2 章 Cx51 程序设计基础	16
2.1 标识符与关键字	16
2.2 Cx51 程序设计的基本语法	18
2.2.1 数据类型	18
2.2.2 常量	21
2.2.3 变量及其存储模式	22
2.2.4 用 typedef 重新定义数据类型	26
2.2.5 运算符与表达式	27
2.3 Cx51 程序的基本语句	39
2.3.1 表达式语句	39
2.3.2 复合语句	39
2.3.3 条件语句	41
2.3.4 开关语句	43
2.3.5 循环语句	45
2.3.6 返回语句	50
第 3 章 函数	51
3.1 函数的定义	51
3.2 函数的调用	53
3.2.1 函数的调用形式	53
3.2.2 对被调用函数的说明	54
3.2.3 函数的参数和函数的返回值	56
3.2.4 实际参数的传递方式	58
3.3 函数的递归调用与再入函数	58
3.4 中断服务函数与寄存器组定义	60
3.5 函数变量的存储方式	63

3.5.1	局部变量与全局变量	63
3.5.2	变量的存储种类	65
3.5.3	函数的参数和局部变量的存储器模式	72
第 4 章	数组与指针	73
4.1	数组的定义与引用	73
4.2	字符数组	74
4.3	数组作为函数的参数	77
4.4	指针	80
4.4.1	指针与地址	80
4.4.2	指针变量的定义	81
4.4.3	指针变量的引用	87
4.4.4	指针变量作为函数的参数	88
4.5	数组的指针	89
4.5.1	用指针引用数组元素	89
4.5.2	字符数组指针	91
4.5.3	指针的地址计算	92
4.6	函数型指针	94
4.7	返回指针型数据的函数	96
4.8	指针数组与指针型指针	97
4.8.1	指针数组	97
4.8.2	指针型指针	101
4.8.3	抽象型指针	103
第 5 章	结构、联合与枚举	106
5.1	结构变量的定义与引用	106
5.2	结构变量的初值	109
5.3	结构数组	110
5.4	结构型指针	111
5.4.1	结构型指针的概念	111
5.4.2	用结构型指针引用结构元素	111
5.5	结构与函数	112
5.5.1	将结构作为函数的参数	112
5.5.2	将结构型指针作为函数的参数	115
5.6	联合变量的定义与引用	116
5.7	枚举变量的定义与引用	121
第 6 章	预处理器	124
6.1	宏定义	124

6.1.1	不带参数的宏定义	124
6.1.2	带参数的宏定义	126
6.2	文件包含	129
6.3	条件编译	129
6.4	其他预处理命令	132
第 7 章	μ Vision2 集成开发环境	133
7.1	μ Vision2 的下拉菜单	135
7.1.1	File 菜单	136
7.1.2	Edit 菜单	137
7.1.3	View 菜单	140
7.1.4	Project 菜单	143
7.1.5	Tools 菜单	154
7.1.6	SVCS 菜单	155
7.1.7	Window 菜单	157
7.1.8	Help 菜单	157
7.2	μ Vision2 中的调试器	158
7.2.1	Debug 状态下窗口分配与 View 菜单操作	160
7.2.2	通过 Debug 菜单进行程序代码调试	168
7.2.3	通过“Peripherals”菜单观察仿真结果	174
7.3	μ Vision2 的调试命令	176
7.3.1	显示和更新存储器内容命令	176
7.3.2	程序执行控制命令	181
7.3.3	断点管理命令	184
7.3.4	其他通用命令	187
7.4	μ Vision2 的表达式	195
7.4.1	表达式的组成	195
7.4.2	μ Vision2 表达式与 C 语言表达式之间的差别	201
7.4.3	μ Vision2 表达式应用举例	201
7.5	μ Vision2 的函数	204
7.5.1	内部函数	204
7.5.2	用户函数	209
7.5.3	信号函数	211
7.5.4	μ Vision2 函数与 Cx51 函数的差别	213
第 8 章	μ Vision2 环境下 Cx51 编程技巧与应用实例	214
8.1	编写 Cx51 应用程序的基本原则	214
8.2	Cx51 应用中的一些常见问题与解决方法	216
8.2.1	Cx51 程序设计中容易出错的地方	216

8.2.2	有关 Cx51 程序设计的若干实际应用技巧	218
8.3	8051 单片机的片内定时器应用编程	224
8.3.1	多模块编程	224
8.3.2	片上集成外围功能应用	246
8.3.3	在 μ Vision2 中生成应用库函数	249
8.4	在 μ Vision2 中应用硬件目标板	250
8.5	8051 单片机的片内串行 I/O 应用编程	255
8.5.1	用 8051 串行口扩展的矩阵键盘	255
8.5.2	利用 8051 串行口实现多机通信	257
8.5.3	8051 串行口的中断驱动程序	259
8.5.4	利用 8051 串行口实现人机对话命令的输入输出	263
8.6	8051 单片机串行接口扩展应用编程	267
8.6.1	5 位共阴极 LED 驱动器 MC14489 的应用	267
8.6.2	8 位共阴极 LED 驱动器 MAX7219 的应用	271
8.6.3	单总线温度传感器 DS1820 的应用	280
8.6.4	实时时钟芯片 DS1302 的应用	292
8.7	I ² C 总线驱动程序	302
8.7.1	I ² C 总线简介	302
8.7.2	I ² C 总线通用软件模拟驱动程序	305
8.7.3	I ² C 接口器件 24C04 的读写程序	308
8.8	8051 单片机存储器扩展与测试应用编程	313
8.8.1	测试 8051 应用系统总线与扩展存储器的 Cx51 程序	313
8.8.2	8051 扩展 FLASH 存储器在系统编程的 Cx51 程序	316
8.8.3	P89C51RD2 单片机片内 FLASH 在应用中编程的 Cx51 程序	329
8.9	8051 单片机并行接口扩展应用编程	341
8.9.1	打印输出接口及其驱动程序	342
8.9.2	D/A 及 A/D 转换接口及其驱动程序	345
8.9.3	用可编程芯片 8155 实现并行 I/O 接口扩展	353
8.9.4	实时日历/时钟芯片 DS12887 的 Cx51 驱动程序	369
8.9.5	用可编程芯片 8279 扩展键盘/LED 显示器接口	374
8.9.6	LCD 显示模块 EA-D20040AR 的 Cx51 应用编程	380
8.9.7	内置 T6963C 驱动器的 LCD 显示模块 Cx51 应用编程	388
8.10	80C552 单片机内部 A/D 转换器的 Cx51 驱动程序	402
8.10.1	80C552 单片机简介	402
8.10.2	80C552 内部 ADC 的应用	404
8.10.3	使用 80C552 内部 ADC 时印刷电路板的设计要点	408
8.10.4	使用 80C552 内部 ADC 的 Cx51 驱动程序	410
8.11	87C752 单片机在气流量测量中的应用	415
8.11.1	87C752 单片机简介	415

8.11.2	气流量测量仪表的硬件设计	420
8.11.3	气流量测量仪表的软件设计	423
第 9 章	Cx51 编译器	435
9.1	Cx51 编译器简介	435
9.2	Cx51 编译器控制命令详解	438
9.2.1	源控制命令	438
9.2.2	列表控制命令	441
9.2.3	目标控制命令	446
9.3	Keil Cx51 编译器对 ANSI C 的扩展	470
9.3.1	存储器类型与编译模式	470
9.3.2	关于 bit, sbit, sfr, sfr16 数据类型	471
9.3.3	一般指针与基于存储器的指针及其转换	475
9.3.4	Cx51 编译器对 ANSI C 函数定义的扩展	476
9.4	Cx51 编译器的数据调用协议	483
9.4.1	数据在内存中的存储格式	483
9.4.2	目标代码的段管理	485
9.5	Cx51 编译器的特殊支持	487
9.5.1	Analog Device 公司的 Aduc8xxB2 系列	487
9.5.2	Atmel 89x8252 及衍生产品	488
9.5.3	Dallas 80C320、C420、C520、C530	488
9.5.4	Dallas 80C390、C400、C5240 及衍生产品	488
9.5.5	Infineon 80C517、C509、C537 及衍生产品	489
9.5.6	Philips 8xC750、C751、C752	490
9.5.7	Philips 8xC51Mx	491
9.5.8	Philips 和 Atmel WM 系列的双数据指针	491
9.6	用户配置文件	491
9.6.1	启动代码文件	491
9.6.2	变量初始化文件	494
9.6.3	基本 I/O 函数文件	495
9.6.4	分组配置文件	495
9.7	与汇编语言程序的接口	497
9.8	与 PL/M51 程序的接口	507
9.9	绝对地址访问	507
9.9.1	采用扩展关键字 “_at_” 或指针定义变量的绝对地址	507
9.9.2	采用预定义宏指定变量的绝对地址	508
9.9.3	采用连接定位控制命令指定变量的绝对地址	509
9.10	Cx51 的库函数	511
9.10.1	字符函数 CTYPE.H	511

9.10.2	标准 I/O 函数 <code>STDIO.H</code>	517
9.10.3	字符串函数 <code>STRING.H</code>	525
9.10.4	标准函数 <code>STDLIB.H</code>	534
9.10.5	数学函数 <code>MATH.H</code>	540
9.10.6	绝对地址访问 <code>ABSACC.H</code>	546
9.10.7	内部函数 <code>INTRINS.H</code>	548
9.10.8	变量参数表 <code>STDARG.H</code>	550
9.10.9	全程跳转 <code>SETJMP.H</code>	551
9.10.10	计算结构体成员的偏移量 <code>STDDEF.H</code>	552
第 10 章 Ax51 宏汇编器		554
10.1	符号与表达式.....	556
10.2	汇编伪指令.....	561
10.2.1	段控制指令.....	561
10.2.2	符号定义指令.....	564
10.2.3	存储器初始化指令.....	567
10.2.4	存储器保留指令.....	568
10.2.5	过程声明指令（仅用于 Ax51）.....	570
10.2.6	程序连接指令.....	571
10.2.7	地址控制指令.....	572
10.2.8	其他指令.....	573
10.3	宏处理器.....	574
10.3.1	标准宏处理器.....	574
10.3.2	嵌套宏定义.....	577
10.3.3	宏调用.....	578
10.3.4	C 宏处理器.....	582
10.3.5	MPL 宏处理器.....	584
10.3.6	MPL 函数.....	587
10.3.7	MPL 条件处理函数.....	590
10.3.8	MPL 字符串处理函数.....	593
10.4	汇编控制命令.....	595
10.4.1	汇编控制命令详解.....	596
10.4.2	条件汇编命令.....	605
第 11 章 BL51/Lx51 连接定位器与实用工具		608
11.1	BL51/Lx51 的连接定位方式.....	608
11.1.1	连接定位中的数据段处理.....	608
11.1.2	连接定位器的引用.....	611
11.1.3	应用程序在存储器空间中的定位.....	612

11.1.4	数据覆盖	614
11.1.5	代码分组	616
11.1.6	分组配置	618
11.2	连接定位控制命令详解	625
11.2.1	列表文件控制命令	626
11.2.2	输出文件控制命令	629
11.2.3	段与存储器定位控制命令	633
11.2.4	高级语言控制命令	650
11.3	符号转换工具	658
11.3.1	Intel HEX 文件格式与符号转换工具	658
11.3.2	引用符号转换工具 OH51/OHX51	659
11.3.3	引用分组目标文件转换工具 OC51	660
11.4	库管理器 LIBx51	660
第 12 章	RTX51 实时多任务操作系统	664
12.1	RTX51 一般介绍	664
12.2	RTX51 技术参数	669
12.3	使用 RTX51 TINY 的要求和限定	671
12.4	RTX51 TINY 的任务管理	673
12.5	RTX51 TINY 的配置	674
12.6	RTX51 TINY 的系统函数	676
12.7	RTX51 TINY 应用系统调试	681
附录 A	Keil Cx51 与 ANSI C 的差别	696
附录 B	Keil Cx51 不同版本的差别	698
附录 C	代码优化	702
附录 D	Cx51 编译器的限制	705
附录 E	关于配套光盘及硬件目标板	706
参考文献	707

第 1 章 8051 单片机与 Keil Cx51 基础

1.1 8051 单片机的存储器组织结构

8051 单片机最早由 Intel 公司推出，它在一块超大规模集成电路芯片上同时集成了 CPU、ROM、RAM 以及 TIMER/COUNTER，使用者只需外接少量的接口电路就可组成自己的专用微处理器系统。目前，市场上 8051 单片机的硬件支持芯片及软件应用程序十分丰富，除了 Intel 公司之外，还有 Philips、Siemens、ADM、Fujitsu、OKI、ATMEL 等公司都推出了以 8051 为核心的单片机，新一代的 8051 单片机集成度更高，在片内集成了更多的功能部件，如 A/D、PWM、PCA、WDT 以及高速 I/O 口等，在工业测量控制领域内获得极为广泛的应用，因此，有人指出 8051 单片机已成为事实上的工业标准。目前已有多个厂家生产不同型号的 8051 单片机，它们各有特点，但其基本内核相同，指令系统也完全兼容。

图 1.1 所示为 8051 基本内核的结构框图，包括中央处理器 CPU，用于执行各种指令和运算处理；内部数据存储器 RAM，用于存放可以读写的数据；内部程序存储器 ROM，用于存放程序指令或某些常数表格；四个 8 位的并行 I/O 接口 P0、P1、P2 和 P3（每个口都可以用做输入或者输出）；二个定时器/计数器，用做外部事件计数器或内部定时；中断系统具有 5 个中断源（2 个外部中断、2 个定时器中断、1 个串行口中断，采用 2 个优先级的嵌套中断结构，可实现二级中断服务程序嵌套，每一个中断源都可用软件程序规定为高优先级中断或低优先级中断）；一个串行接口电路（用做异步接收发送器）；内部时钟电路（晶体和微调电容需要外接，振荡频率可以高达 40MHz）。以上各部分通过内部总线相连接。在很多情况下，单片机还要和外部设备或外部存储器相连接，连接方式采用三总线（地址、数据、控制）方式，但在 8051 单片机中，没有单独的地址总线和数据总线，而是与并行 I/O 口中的 P0 口及 P2 口公用的，进行外部扩展时，P0 口分别作为低 8 位地址线和 8 位数据线，P2 口则作为高 8 位地址线用，所以也是 16 条地址线和 8 条数据线。但是一定要建立一个明确的概念，单片机进行外部扩展的地址线和数据线都不是独立的总线，而是与并行 I/O 口公用的，这是 8051 单片机结构上的一个特点。

对于采用高级语言 Keil Cx51 的用户来说，了解和熟悉 8051 单片机的存储器组织结构是十分必要的，这样在具体编程时可以合理安排各种变量，最大限度实现代码优化。从使用者的角度看，8051 单片机有如下三个存储器空间。

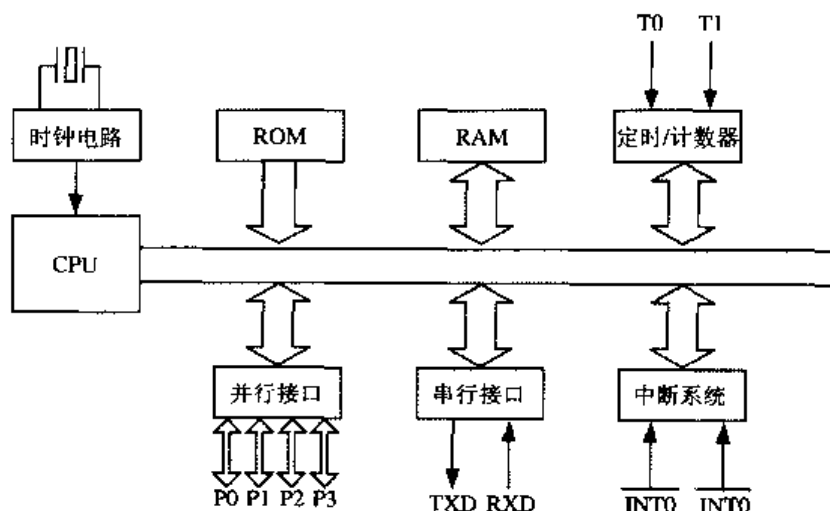


图 1.1 8051 基本内核的结构框图

程序存储器 ROM 对于普通 8051 单片机，程序存储器 ROM 空间大小为 64KB，用于存放程序代码和一些表格常数，称为 CODE 空间。普通 8051 可采用“代码分组”(CODE BANK)设计技术，将 ROM 空间扩展到 $32 \times 64\text{KB}$ ，新型 Philips 80C51Mx 单片机的 ROM 空间最大可扩展到 16MB，称为 ECODE 和 HCONST 空间。8051 单片机专门提供一个引脚“EA”来区分片内 ROM 和片外 ROM， $\overline{\text{EA}}$ 引脚接高电平时，单片机从片内 ROM 中读取指令，当指令地址超过片内 ROM 空间范围后，就自动地转向片外 ROM 读取指令； $\overline{\text{EA}}$ 引脚接低电平时，所有的取指操作均对片外 ROM 进行。程序存储器的某些地址单元是保留给系统使用的：0000H~0002H 单元是所有执行程序的入口地址，复位后 CPU 总是从 0000H 地址开始执行程序；0003H~002BH 单元均匀地分为 5 段，用于 5 个中断服务程序的入口，产生某个中断时，将自动进入其对应入口地址开始执行中断服务程序，一些新型 8051 单片机增加了更多的中断源，它们的中断入口地址也相应增加。

片内数据存储器 RAM 对于普通 8051 单片机，片内数据存储器 RAM 空间最大为 256B，用于存放程序执行过程的各种变量及临时数据。片内 RAM 的低 128 个字节可用直接寻址方式进行访问，也可用间接寻址方式访问，称为 DATA 区，其中，00H~1FH 地址范围平均分为 4 组，每组都有 8 个工作寄存器 R0~R7，称为工作寄存器区 (Register Banks)。20H~2FH 地址范围中，每个存储器单元的每一位都可以用位处理指令直接操作，该段地址范围称为位寻址区 (BDATA 区)，其中每一位称为一个 bit。对于 51 子系列单片机仅有上述低 128 个字节，对于 52 子系列单片机，增加了高 128 个字节的片内 RAM，地址范围为 80H~FFH，该范围只能采用间接寻址方式访问，整个片内 RAM 地址范围 00H~FFH 称为 IDATA 区。与 IDATA 空间高 128 个字节（地址范围 80H~FFH）重叠部分称为特殊功能寄存器区 (SFR SPACE)，有些特殊功能寄存器是可以位寻址的，其可寻址位称为 sbit。Philips 公司推出的新型单片机 80C51Mx，其片内 RAM 最大可扩充到 64KB，称为 EDATA 区。

片外数据存储器 RAM 对于普通 8051 单片机，片外数据存储器 RAM 空间大小为 64KB，称为 XDATA 区。在 XDATA 空间内进行分页寻址操作时，称为 PDATA 区。有些

新型 80C51 单片机的扩充片内 RAM，需要用专门的特殊功能寄存器“映像”（MAP）到 XDATA 地址空间；还有一些新型 80C51 单片机可以将片外 RAM 最大扩展到 16MB，称为 HDATA 区。

图 1.2 所示为普通 8051 单片机的存储器组织结构，其中，各部分空间说明及地址范围见表 1-1。

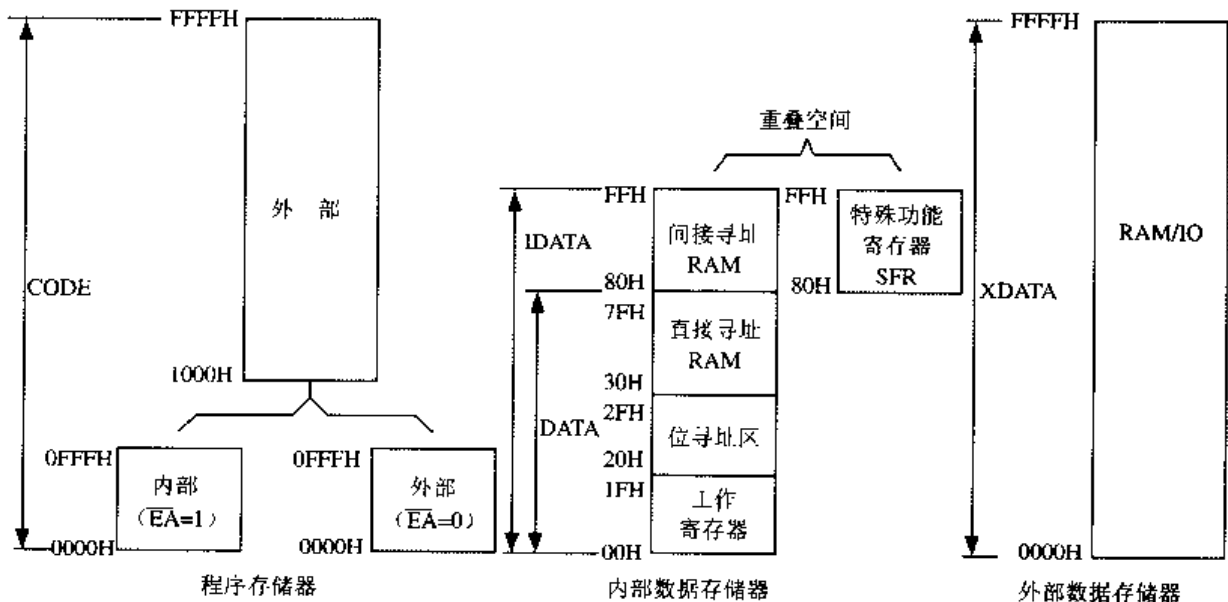


图 1.2 普通 8051 单片机的存储器组织结构

表 1-1 普通 8051 单片机存储器空间分配表

空间名称	地址范围	说明
DATA	D:00H~D:7FH	片内 RAM 直接寻址区
BDATA	D:20H~D:2FH	片内 RAM 位寻址区
IDATA	I:00H~I:FFH	片内 RAM 间接寻址区
XDATA	X:0000H~X:FFFFH	64KB 片外 RAM 数据区
CODE	C:0000H~C:FFFFH	64KB 片内外 ROM 代码区
BANK0~BANK31	B0:0000~B0:FFFFH ⋮ B31:0000~B31:FFFFH	分组代码区,最大可扩展 32×64KB ROM

图 1.3 所示为新型 80C51 单片机的扩展存储器组织结构，其中各部分空间说明及地址范围见表 1-2。

图 1.4 所示为 Philips 80C51Mx 单片机的存储器组织结构，其中，各部分空间说明及地址范围见表 1-3。

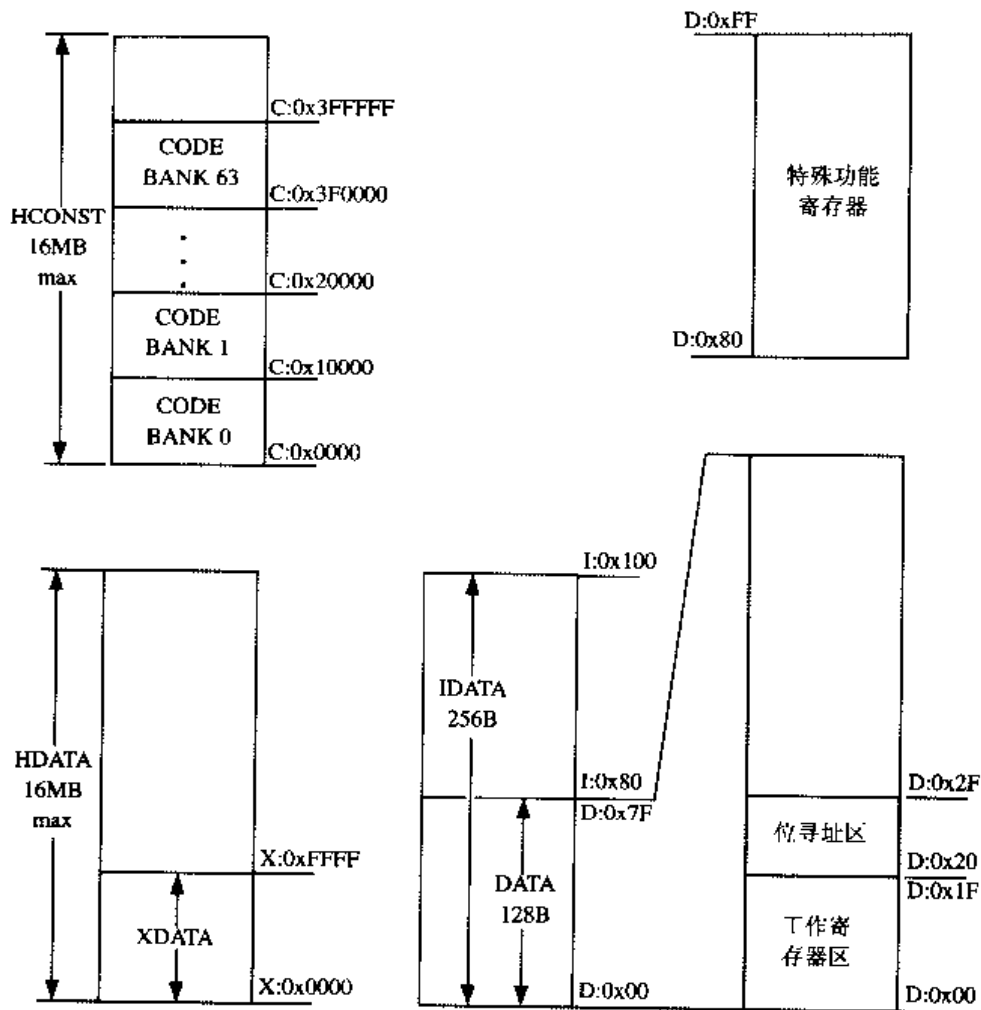


图 1.3 新型 8051 单片机的扩展存储器组织结构

表 1-2 新型 80C51 单片机扩展存储器空间分配表

空间名称	地址范围	说明
DATA	D:00H~D:7FH	片内 RAM 直接寻址区
BDATA	D:20H~D:2FH	片内 RAM 位寻址区
IDATA	I:00H~I:FFH	片内 RAM 间接寻址区
XDATA	X:0000H~X:FFFFH	64KB 常规片外 RAM 数据区
HDATA	X:0000H~X:FFFFFFH	16MB 扩展片外 RAM 数据区
CODE	C:0000H~C:FFFFH	64KB 常规片内外 ROM 代码区
HCONST (ECODE)	C:0000H~C:FFFFFFH	16MB 扩展片外 ROM 常数区 (对 Dallas390 可用做代码区)
BANK0~BANK31	B0:0000~B0:FFFFH ... B31:0000~B31:FFFFH	分组代码区, 最大可扩展 32×64KB ROM

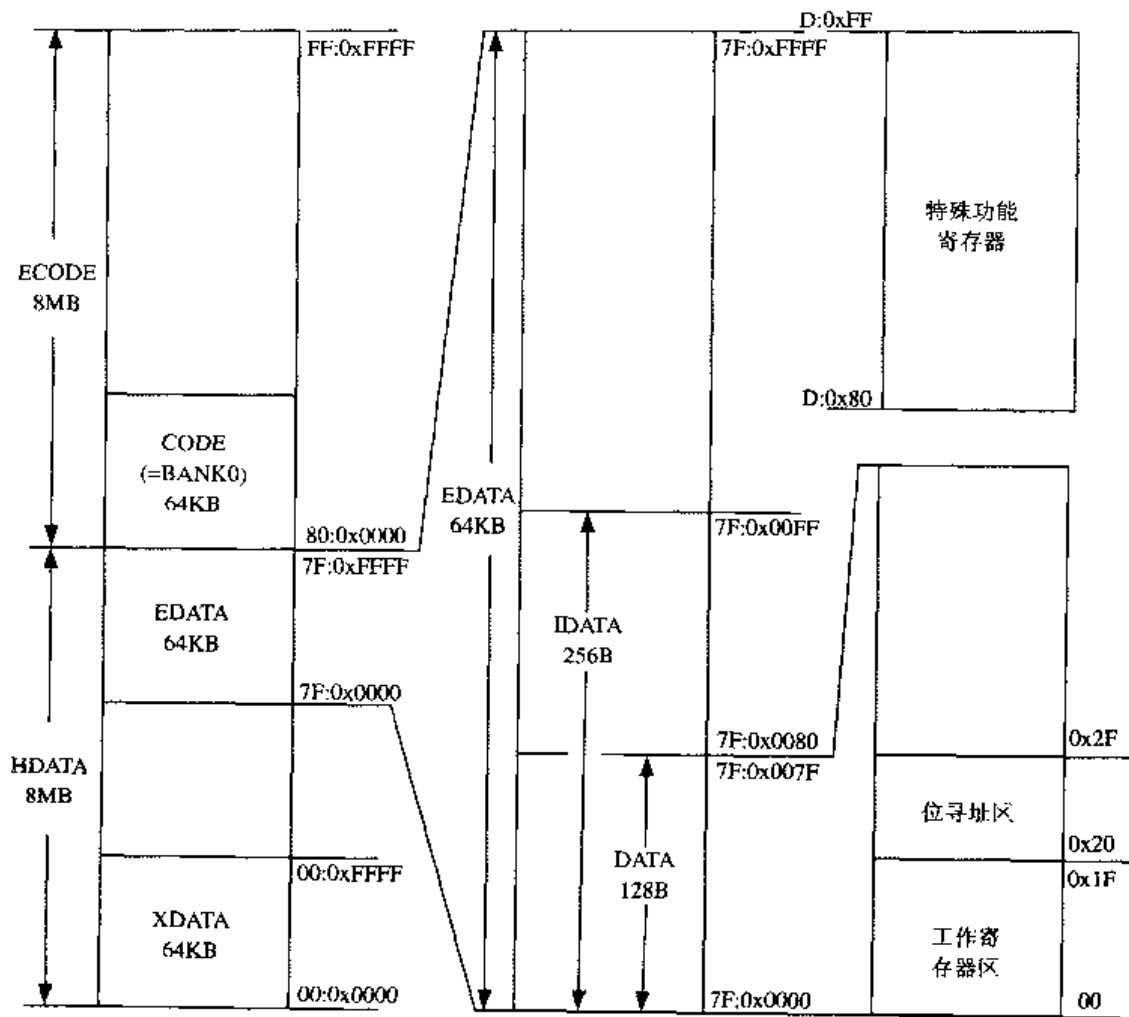


图 1.4 Philips 80C51Mx 单片机的存储器组织结构

表 1-3 Philips 80C51Mx 单片机存储器空间分配表

空间名称	地址范围	说明
DATA	7F:0000H~7F:007FH	片内RAM直接寻址区
BDATA	7F:0020H~7F:002FH	片内RAM位寻址区
IDATA	7F:0000H~7F:00FFH	片内RAM间接寻址区
EDATA	7F:0000H~7F:FFFFH	扩展片内RAM, 可用做堆栈区
XDATA	00:0000H~00:FFFFH	64KB常规片外RAM数据区
HDATA	00:0000H~7F:FFFFH	8MB扩展片外RAM数据区
CODE	80:0000H~80:FFFFH	64KB常规片内外ROM代码区
ECODE	80:0000H~FF:FFFFH	8MB扩展片内外ROM代码区
BANK0~BANK63	80:0000~80:FFFFH ... BF:0000~BF:FFFFH	分组代码区, 最大可扩展64×64KB ROM

1.2 Keil Cx51 开发工具

C 语言是一种通用的计算机程序设计语言，在国际上十分流行，它既可用于编写计算机的系统程序，也可用来编写一般的应用程序。以前计算机的系统软件主要是用汇编语言编写的，对于单片机应用系统来说更是如此。由于汇编语言程序的可读性和可移植性都较差，采用汇编语言编写单片机应用程序的周期长，而且调试和排错也比较困难。为了提高编制单片机应用程序的效率，改善程序的可读性和可移植性，采用高级语言无疑是一种最好的选择。C 语言既具有一般高级语言的特点，又能直接对计算机的硬件进行操作，表达和运算能力也较强，许多以前只能采用汇编语言来解决的问题现在都可以改用 C 语言来解决。德国 Keil Software 公司多年来致力于单片机 C 语言编译器的研究，该公司开发的 Keil Cx51 是一种专为 8051 单片机设计的高效率 C 语言编译器，符合 ANSI 标准，生成的程序代码运行速度极高，所需要的存储器空间极小，完全可以和汇编语言相媲美。

Keil 公司目前已经推出了 V7.0 以上版本的 Cx51 编译器，为 8051 单片机软件开发提供了全新的 C 语言环境，同时保留了汇编代码高效、快速的特点。Cx51 已被完全集成到一个功能强大的全新集成开发环境 μ Vision2 中，其中包括项目（project）管理器、Cx51 编译器、Ax51 宏汇编器、BL51/Lx51 连接定位器、RTX51 实时操作系统、Simulator 软件模拟器以及 Monitor51 硬件目标调试器，所有这些功能均可在 μ Vision2 提供的单一而灵活的开发环境中极为简便地进行操作。 μ Vision2 提供了强大的项目管理功能，可以十分方便地进行结构化多模块程序设计。 μ Vision2 内部集成源级浏览器（Browser）利用符号数据库中详细的符号信息，使用户可以快速浏览源文件，并优化用户的变量数据存储。 μ Vision2 内部集成器件数据库（device database）储存了多种不同型号单片机的片上资源信息，通过它可以自动设置 Cx51 编译器、Ax51 宏汇编器、BL51/Lx51 连接定位器及调试器的默认选项，充分满足用户利用特定单片机片上集成外围功能的要求。 μ Vision2 内部集成源程序编辑器允许用户在编辑源程序文件时（甚至在未经编译和汇编之前）设置程序调试断点，便于在程序调试过程中快速检查和修改程序。 μ Vision2 提供文件查找功能，能对单一文件或全部项目文件进行指定搜索。此外还提供了用户工具菜单接口，允许在 μ Vision2 中直接启动用户功能。 μ Vision2 支持软件模拟仿真（Simulator）和用户目标板调试（Monitor51）两种工作方式，在软件模拟仿真方式下不需要任何 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，在用户目标板调试方式下，利用硬件目标板中的监控程序可以直接调试目标硬件系统，使用户可以节省购买昂贵硬件仿真器的费用。

1.3 Cx51 简单编程与调试

采用 Keil Cx51 开发 8051 单片机应用程序一般需要以下步骤。

① 在 μ Vision2 集成开发环境中创建一个新项目（Project），并为该项目选定合适的单片机 CPU 器件。

② 利用 μ Vision2 的文件编辑器编写 C 语言（或汇编语言）源程序文件，并将文件添

加到项目中去。一个项目可以包含多个文件,除源程序文件外还可以有库文件或文本说明文件。

③ 通过 μ Vision2 的各种选项,配置 Cx51 编译器,As51 宏汇编器,BL51/Lx51 连接定位器以及 Debug 调试器的功能。

④ 利用 μ Vision2 的构造(Build)功能对项目中的源程序文件进行编译连接,生成绝对目标代码和可选的 HEX 文件,如果出现编译连接错误则返回到第②步,修改源程序中的错误后重新构造整个项目。

⑤ 将没有错误的绝对目标代码装入 μ Vision2 调试器进行仿真调试,调试成功后将 HEX 文件写入到单片机应用系统的 EPROM 中。

下面通过一个简单实例进行说明。启动 μ Vision2 后,用鼠标左键单击“Project 菜单/New Project”选项,在弹出的对话框窗口中输入项目文件名 max,并选择合适的保存路径(通常为每个项目建一个单独的文件夹),单击“保存”按钮,这样就创建了一个文件名为 max.uv2 的新项目文件,如图 1.5 所示。



图 1.5 在 μ Vision2 中新建一个项目

项目名保存完毕后将弹出如图 1.6 所示器件数据库对话框窗口,用于为新建项目选择一个 CPU 器件(窗口的 Description 栏对不同公司生产的 8051CPU 器件做了必要说明),根据需要选择 CPU 器件(例如 Atmel 公司的 AT89C51),选定后 μ Vision2 将按所选器件自动设置默认的工具选项,从而简化了项目的配置过程。

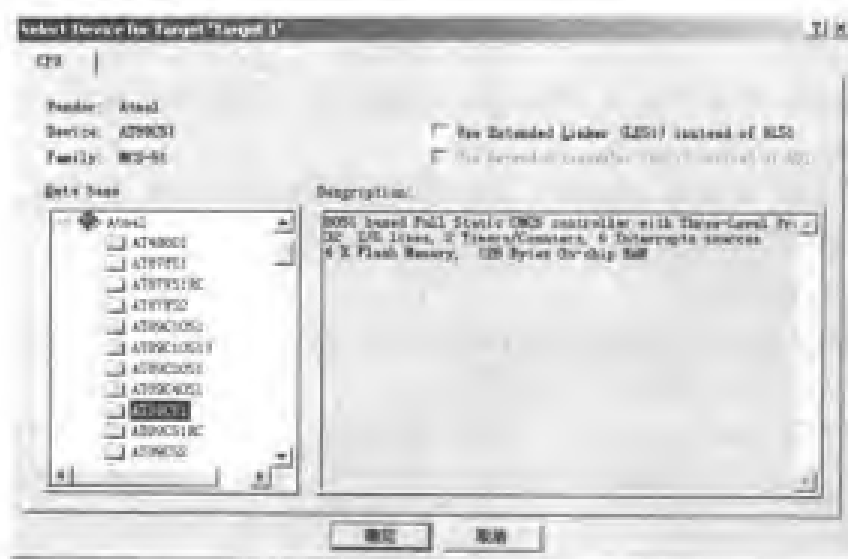


图 1.6 为项目选择 CPU 器件

创建一个新项目后，项目中会自动包含一个默认的目标（Target 1）和文件组（Source Group 1）。用户可以给项目添加其他文件组（Group）以及文件组中的源文件，这对于模块化编程特别有用。项目中的目标名，组名以及文件名都显示在 μ Vision2 的“项目窗口/Files”标签页中。接下来要给项目添加源程序文件，源文件可以是已有的，也可以是新建的。新建源文件时单击“File 菜单/New”选项，从打开的编辑窗口中输入下面例 1.1 的 C51 源程序。

例 1.1: 求两个输入数据中的较大者。

```
#include<stdio.h> /* 预处理命令 */
#include<reg51.h>

char max (char x , char y) { /* 定义 max 函数，x、y 为形式参数 */
    if ( x > y ) return (x); /* 将计算得到的最大值返回到调用处 */
    else return(y);
} /* max 函数结束 */

main() { /* 主函数 */
    char a, A, c; /* 主函数的内部变量类型说明 */
    SCON=0x52; /* 8051 单片机串行口初始化 */
    TMOD=0x20;
    TCON=0x69;
    TH1=0x0F3;
    scanf ("%c %c", &a, &A); /* 输入变量 a 和 b 的值 */
    c= max (a,A); /* 调用 max 函数 */
    printf ( " \n max =%c \n ", c); /* 输出变量 c 的值 */
} /* 主程序结束 */
```

程序输入完成后，用鼠标左键单击“File 菜单/Save As...”选项，如图 1.7 所示，将

其另存为扩展名为.C 的源程序文件,保存路径一般设为与项目文件相同。



图 1.7 保存新建项目中的源程序文件

μVision2 具有十分完善的右键功能,将鼠标指向“项目窗口/Files”标签页中的“Source Group 1”文件组并单击右键,弹出一个快捷菜单,如图 1.8 所示。



图 1.8 项目窗口的右键菜单

用左键单击右键快捷菜单中的“Add Files to Group ‘Source Group 1’”选项,弹出如图 1.9 所示添加源文件选择窗口,选中刚才保存的源程序文件“max.c”并单击“Add”按钮,将其添加到新创建的项目中去。

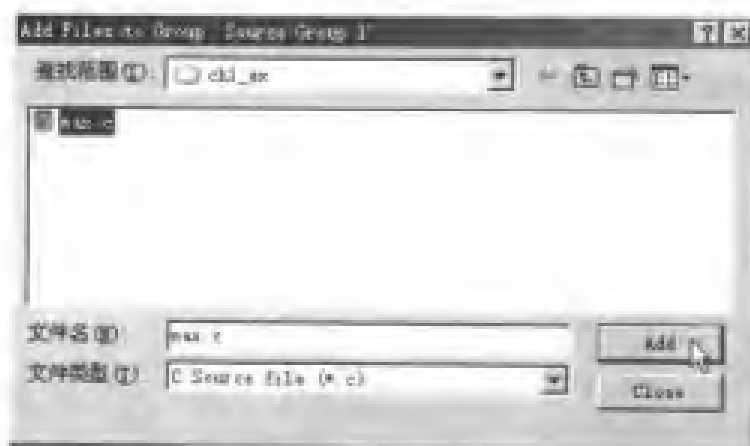


图 1.9 添加源文件选择窗口

接下来根据需要配置 Cx51 编译器、A51 宏汇编器、BL51/Lx51 连接定位器以及 Debug 调试器的各项功能。单击“Project 菜单/Options for Target”选项,弹出如图 1.10 所示窗口,这是一个十分重要的窗口,包括“Device”、“Target”、“Output”、“Listing”、“C51”、“A51”、“BL51 Locate”、“BL51 Misc”和“Debug”等多个选项标签页,其中许多选项可以直接用其默认值,必要时可进行适当调整。图 1.10 所示为其中的“Target”标签页,用于设定目标硬件系统的时钟频率 Xtal 为 24.0MHz、C51 编译器的存储器模式为 Small (C51 程序中局部变量位于片内数据存储器 DATA 空间)、程序存储器 ROM 空间设为 Large (使用 64KB 程序存储器)、不采用实时操作系统、不采用代码分组设计。



图 1.10 配置目标选项

图 1.11 所示为“Output”标签页，用于设定当前项目在编译连接之后生成的执行代码输出文件。输出文件名默认为与项目文件同名（也可以指定其他文件名），存放在当前项目文件所在的目录中，也可以单击“Select Folder For Objects”来指定存放输出文件的目录路径。选中“Create Executable”表示项目编译连接后生成执行代码输出文件。选中方形复选框“Debug Information”将在输出文件中包含进行源程序调试的符号信息。选中方形复选框“Browse Information”将在输出文件中包含源程序浏览信息。选中方形复选框“Create HEX File”表示当前项目编译连接完成之后生成一个用于 EPROM 编程的 HEX 文件。在“After Make”栏中选中方形复选框“Beep When Complete”和“Start Debugging”表示编译连接完成之后计算机将发出一声提示音，并立即进入调试状态。



图 1.11 设定执行代码输出文件选项

图 1.12 所示为“C51”标签页，用于设定当前项目对 Cx51 编译器的控制命令选项。“Preprocessor Symbols”栏用于定义 Cx51 预处理器符号，定义符号后一般要在源程序中增加相应的“ifdef”、“ifndef”、“endif”等预处理器命令；“Code Optimization”栏用于设定 Cx51 编译器的优化级别，需要注意的是优化级别并非越高越好，应根据具体要求适当选择，关于 Cx51 编译器优化级别的详细描述请参阅本书第 9 章。“Warning”栏用于选择编译时给出警告信息的详细程度，号码越大越详细。“Include Paths”栏用于指定用户规定的包含文件路径，可以手工指定路径，也可以通过该栏右边的“...”按钮来浏览选择路径。“Misc Controls”用于增加除了 Cx51 编译器默认选项之外的其他命令选项。所有选定的编译命令选项都会显示在“Compiler control string”栏内。

图 1.13 所示为“BL51 Locate”标签页，用于设定当前项目对 BL51 连接定位器的命令选项。选中方形复选框“Use Memory Layout from Target Dialog”时 BL51 连接定位器将按前面“Target”选项标签页中的设定对执行代码进行存储器地址空间定位，这也是 BL51 的默认选项。不选中该复选框则应在其他栏内填入希望的存储器地址空间范围值。关于 BL51 连接定位器对存储器地址空间处理的详细讨论请参阅本书第 11 章。

拟仿真, 排除一般性错误, 然后再进行目标硬件仿真调试。进行软件模拟仿真时应选中“Debug”标签页中的“Use Simulator”圆形单选框, 进行目标硬件仿真调试时则要选中“Use Keil Monitor-51 Driver”圆形单选框, 为了便于读者进一步学习 Keil Cx51 应用编程, 开发自己的单片机系统, 本书作者设计了一种能直接与 μ Vision2 接口的“Keil Cx51 源程序仿真硬件目标板”, 其中预装了 Keil 公司提供的 Monitor-51 监控程序, 可以实现与 μ Vision2 无缝连接, 使用非常方便, 节省用户购买昂贵硬件仿真器的费用, 关于它的具体使用方法请参阅本书第 8.4 节。



图 1.14 设定 Debug 仿真调试选项

在该窗口所有的标签页中都有一个“Defaults”按钮, 用于设定各种默认命令选项, 初次使用时可以直接采用这些默认值, 待熟悉之后再进一步采用其他选项。有关其他标签页的详细描述请参阅本书第 7 章。

完成上述关于编译、链接定位、仿真调试工具配置的基本选项设定之后, 就可以对当前新建项目进行整体创建 (Build target)。将鼠标指向项目窗口中的文件“max.c”并单击右键, 从弹出的快捷菜单中单击“Build target”选项, 如图 1.15 所示, μ Vision2 将按“Options for Target”窗口内各种选项设置, 自动完成对当前项目中所有源程序模块文件的编译链接, 同时 μ Vision2 的输出窗口将显示编译链接提示信息, 如图 1.16 所示, 如果有编译链接错误, 将鼠标指向窗口内的提示信息双击左键, 光标将自动跳到编辑窗口源程序文件发生错误的地方, 以便于修改, 如果没有编译链接错误则生成绝对目标代码文件。

编译链接完成后将 μ Vision2 转入仿真调试状态, 在此状态下的“项目窗口”自动转到“Regs”标签页, 显示调试过程中单片机内部工作寄存器 R0~R7、累加器 A、堆栈指针 Sp、数据指针 DPTR、程序计数器 PC 以及程序状态字 PSW 等的值, 如图 1.17 所示。在仿真调试状态下单击“Debug 菜单/Go”选项, 启动用户程序全速运行, 再单击“View 菜单/Serial Window #1”选项打开调试状态下 μ Vision2 的串行窗口 1, 用户程序中采用 scanf()和 printf()所进行的输入和输出操作, 都是通过串行窗口 1 实现的, 将鼠标指向该窗口并键入数字 2

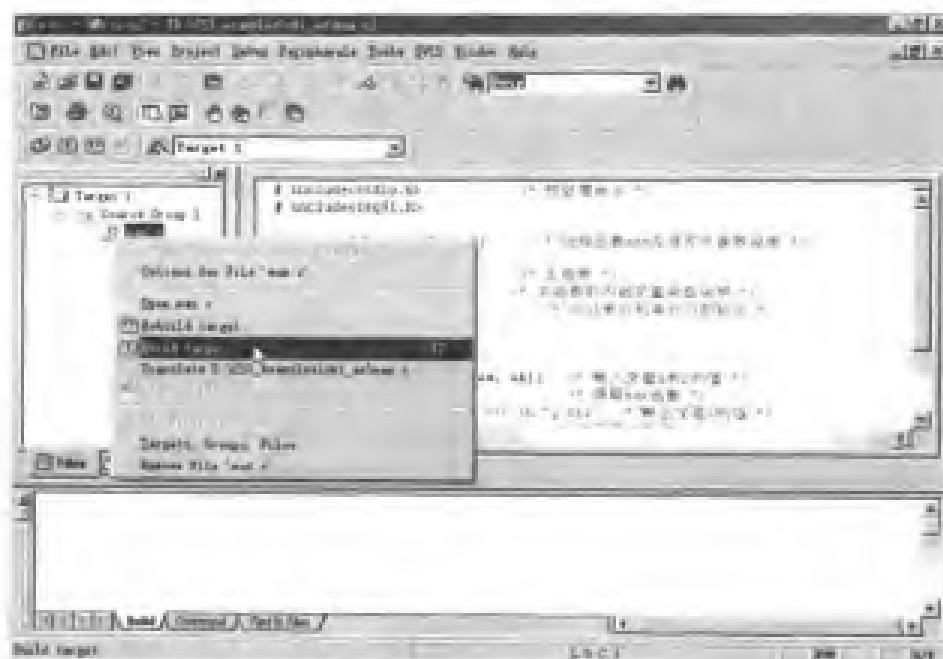


图 1.15 利用右键快捷菜单对当前项目进行编译链接

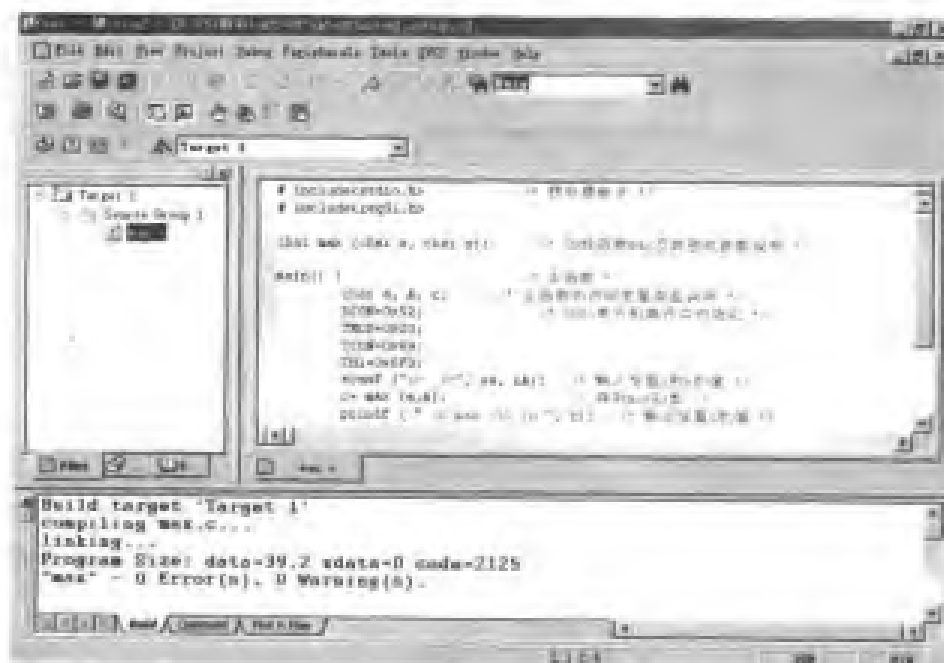


图 1.16 编译链接完成后输出窗口的提示信息

和 9，立即得到输出结果“max=9”，如图 1.18 所示。

μ Vision2 调试器的仿真功能十分完善，除了全速运行之外还可以进行单步、设置断点、运行到光标指定位置等多种操作，调试过程中可随时观察局部变量以及用户设置的观察点状态、存储器状态、片内集成外围功能状态，通过调用信号函数或用户函数可实现其他多种仿真功能。关于 μ Vision2 调试器的详细介绍请参阅本书第 7 章。



图 1.17 μVision2 的仿真调试状态窗口

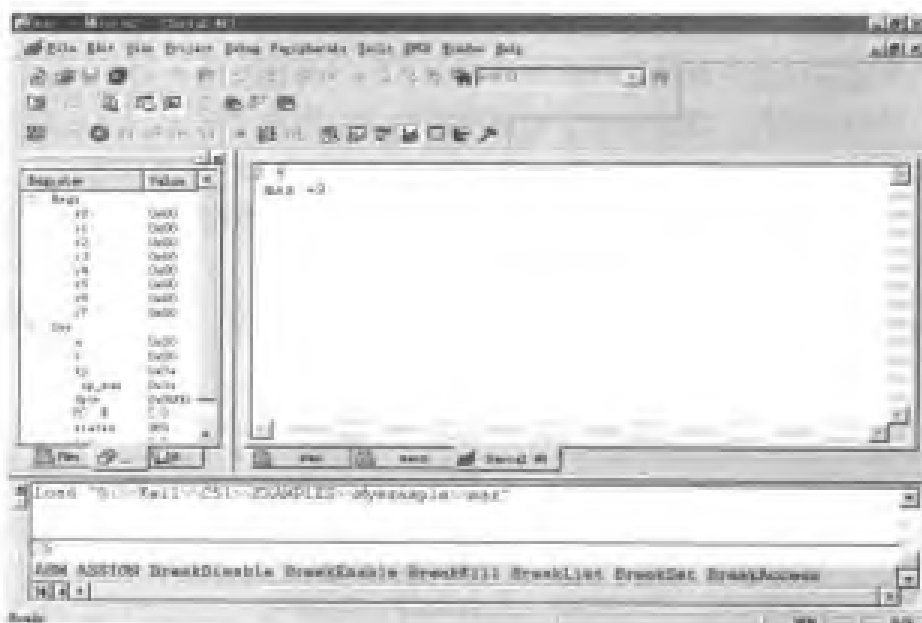


图 1.18 μVision2 调试状态下串行窗口 1 及其数据输入和结果输出

第 2 章 Cx51 程序设计基础

Keil Cx51 是一种专为 8051 单片机设计的高级语言 C 编译器,支持符合 ANSI 标准的 C 语言进行程序设计,同时针对 8051 单片机自身特点作了一些特殊扩展。为了帮助以前惯于使用汇编语言编程的单片机用户尽快掌握 Cx51 编程技术,本章对 C 语言的一些基本知识结合 Cx51 特点进行阐述。

2.1 标识符与关键字

C 语言的标识符是用来标识源程序中某个对象名字的。这些对象可以是函数、变量、常量、数组、数据类型、存储方式、语句等。一个标识符由字符串、数字和下划线等组成,第一个字符必须是字母或下划线。C 语言是对大小写字母敏感的,如“max”与“MAX”是两个完全不同的标识符。程序中对于标识符的命名应当简洁明了,含义清晰,便于阅读理解,如用标识符“max”表示最大值,用“TIMER0”表示定时器 0 等。

关键字是一类具有固定名称和特定含义的特殊标识符,有时又称为保留字。在编写 C 语言源程序时一般不允许将关键字另作他用,换句话说就是对于标识符的命名不要与关键字相同。与其他计算机语言相比,C 语言的关键字是比较少的,ANSI C 标准一共规定了 32 个关键字,表 2-1 按用途列出了 ANSI C 标准的关键字。

表 2-1 ANSI C 标准的关键字

关 键 字	用 途	说 明
auto	存储种类声明	用以声明局部变量,默认值为此
break	程序语句	退出最内层循环体
case	程序语句	switch 语句中的选择项
char	数据类型声明	单字节整型数或字符型数据
const	存储类型声明	在程序执行过程中不可修改的变量值
continue	程序语句	转向下一次循环
default	程序语句	switch 语句中的失败选择项
do	程序语句	构成 do...while 循环结构
double	数据类型声明	双精度浮点数
else	程序语句	构成 if...else 选择结构
enum	数据类型声明	枚举
extern	存储种类声明	在其他程序模块中声明了的全局变量
float	数据类型声明	单精度浮点数
for	程序语句	构成 for 循环结构
goto	程序语句	构成 goto 转移结构

续表

关 键 字	用 途	说 明
if	程序语句	构成 if...else 选择结构
int	数据类型声明	基本整型数
long	数据类型声明	长整型数
register	存储种类声明	使用 CPU 内部寄存器的变量
return	程序语句	函数返回
short	数据类型声明	短整型数
signed	数据类型声明	有符号数，二进制数据的最高位为符号位
sizeof	运算符	计算表达式或数据类型的字节数
static	存储种类声明	静态变量
struct	数据类型声明	结构类型数据
switch	程序语句	构成 switch 选择结构
typedef	数据类型声明	重新进行数据类型定义
union	数据类型声明	联合类型数据
unsigned	数据类型声明	无符号数据
void	数据类型声明	无类型数据
volatile	数据类型声明	声明该变量在程序执行中可被隐含地改变
while	程序语句	构成 while 和 do...while 循环结构

Keil Cx51 编译器除了支持 ANSI C 标准的关键字以外，还根据 8051 单片机自身特点扩展了如表 2-2 所示的关键字。

表 2-2 Keil Cx51 编译器的扩展关键字

关 键 字	用 途	说 明
at	地址定位	为变量进行存储器绝对空间地址定位
alien	函数特性声明	用以声明与 PL/M51 兼容函数
bdata	存储器类型声明	可位寻址的 8051 内部数据存储器
bit	位变量声明	声明一个位变量或位类型的函数
code	存储器类型声明	8051 程序存储器空间
compact	存储器模式	指定使用 8051 外部分页寻址数据存储器空间
data	存储器类型声明	直接寻址的 8051 内部数据存储器
idata	存储器类型声明	间接寻址的 8051 内部数据存储器
interrupt	中断函数声明	定义一个中断服务函数
large	存储器模式	指定使用 8051 外部数据存储器空间
pdata	存储器类型声明	分页寻址的 8051 外部数据存储器
priority	多任务优先声明	规定 RTX51 或 RTX51 Tiny 的任务优先级
reentrant	再入函数声明	定义一个再入函数
sbit	位变量声明	声明一个可位寻址变量
sfr	特殊功能寄存器声明	声明一个 8 位的特殊功能寄存器
sfr16	特殊功能寄存器声明	声明一个 16 位的特殊功能寄存器
small	存储器模式	指定使用 8051 内部数据存储器空间
task	任务声明	定义实时多任务函数
using	寄存器组定义	定义 8051 的工作寄存器组
xdata	存储器类型声明	8051 外部数据存储器

2.2 Cx51 程序设计的基本语法

虽然 C 语言对语法的限制不太严格,用户在编写程序时有较大的自由,但它毕竟还是一种程序设计语言,与其他计算机语言一样,采用 C 语言进行程序设计时,仍需要遵从一定的语法规则。

2.2.1 数据类型

任何程序设计都离不开对于数据的处理,一个程序如果没有数据,它就无法工作。数据在计算机内存中的存放情况由数据结构决定。C 语言的数据结构是以数据类型出现的,数据类型可分为基本数据类型和复杂数据类型,复杂数据类型由基本数据类型构造而成。C 语言中的基本数据类型有 char, int, short, long, float 和 double。对于 Cx51 编译器来说,short 类型与 int 类型相同,double 类型与 float 类型相同。

1. Char: 字符型

有 signed char (带符号数) 和 unsigned char (无符号数) 之分,默认值为 signed char。它们的长度均为一个字节,用于存放一个单字节的数据。对于 signed char 类型数据,其字节中的最高位表示该数据的符号,“0”表示正数,“1”表示负数。负数用补码表示,数值的表示范围是-128~+127;对于 unsigned char 类型数据,其字节中的所有位均用来表示数据的数值,数值的表示范围是 0~255。

2. Int: 整型

有 signed int 和 unsigned int 之分,默认值为 signed int。它们的长度均为两个字节,用于存放一个双字节的数据。signed int 是有符号整型数,字节中的最高位表示数据的符号,“0”表示正数,“1”表示负数。所能表示的数值范围是-32 768~+32 767。unsigned int 是无符号整型数,所能表示的数值范围是 0~65 535。

3. Long: 长整型

有 signed long 和 unsigned long 之分,默认值为 signed long。它们的长度均为四个字节。signed long 是有符号的长整型数据,字节中的最高位表示数据的符号,“0”表示正数,“1”表示负数,数值的表示范围是-2 147 483 648~+2 147 483 647。unsigned long 是无符号长整型数据,数值的表示范围是 0~4 294 967 295。

4. Float: 浮点型

它是符合 IEEE-754 标准的单精度浮点型数据,在十进制数中具有 7 位有效数字。Float 类型数据占用四个字节 (32 位二进制数),在内存中的存放格式如下:

字节地址	+0	+1	+2	+3
浮点数内容	S EEEEEEE	E MMMMMMM	MMMMMMMM	MMMMMMMM

其中, S 为符号位,“0”表示正,“1”表示负。E 为阶码,占用 8 位二进制数,存放

在两个字节中。注意，阶码 E 值是以 2 为底的指数再加上偏移量 127，这样处理的目的是为了避免出现负的阶码值，而指数是可正可负的。阶码 E 的正常取值范围是 1~254，从而实际指数的取值范围为 -126~+127。M 为尾数的小数部分，用 23 位二进制数表示，存放在三个字节中。尾数的整数部分永远为 1，因此不予保存，但它是隐含存在的。小数点位于隐含的整数位“1”的后面。一个浮点数的数值范围是 $(-1)^S \times 2^{E-127} \times (1.M)$ 。

例如，浮点数 -12.5=0xC1480000，在内存中的存放格式为：

字节地址	+0	+1	+2	+3
浮点数内容	11000001	01001000	00000000	00000000

需要指出的是，对于浮点型数据除了有正常数值之外，还可能出现非正常数值。根据 IEEE 标准，当浮点型数据取以下数值（16 进制数）时即为非正常值：

0xFFFFFFFF 非数 (NaN)
 0x7F800000 正溢出 (+INF)
 0xFF800000 负溢出 (-INF)

另外，由于 8051 单片机不包括捕获浮点运算错误的中断向量，因此必须由用户自己根据可能出现的错误条件用软件来进行适当的处理。

除了以上四种基本数据类型之外，还有以下一些数据类型。

5. *: 指针型

指针型数据不同于以上四种基本数据类型，它本身是一个变量，但在这个变量中存放的不是普通的数据而是指向另一个数据的地址。指针变量也要占据一定的内存单元，在 C51 中指针变量的长度一般为 1~3 个字节。指针变量也具有类型，其表示方法是在指针符号“*”的前面冠以数据类型符号，如 char * point1 表示 point1 是一个字符型的指针变量；float * point2 表示 point2 是一个浮点型的指针变量。指针变量的类型表示该指针所指向地址中数据的类型。使用指针型变量可以方便地对 8051 单片机各部分物理地址直接进行操作。

6. bit: 位类型

这是 Keil Cx51 编译器的一种扩充数据类型，利用它可定义一个位变量，但不能定义位指针，也不能定义位数组。

7. sfr: 特殊功能寄存器

这也是 Keil Cx51 编译器的一种扩充数据类型，利用它可以定义 8051 单片机的所有内部 8 位特殊功能寄存器。sfr 型数据占用一个内存单元，其取值范围是 0~255。

8. sfr16: 16 位特殊功能寄存器

它占用两个内存单元，取值范围是 0~65 535，利用它可以定义 8051 单片机内部 16 位特殊功能寄存器。

9. Sbit: 可寻址位

这也是 Keil Cx51 编译器的一种扩充数据类型, 利用它可以定义 8051 单片机内部 RAM 中的可寻址位或特殊功能寄存器中的可寻址位。

例如, 采用如下语句:

```
sfr P0=80H;
sbit FLAG1=P0^1;
```

可以将 8051 单片机 P0 口地址定义为 80H, 将 P0.1 位定义为 FLAG1。

表 2-3 列出了 Keil Cx51 编译器能够识别的数据类型。

表 2-3 Keil Cx51 编译器能够识别的数据类型

数据类型	长度	值域
unsigned char	单字节	0~255
signed char	单字节	-128~+127
unsigned int	双字节	0~+65 536
signed int	双字节	-32 768~+32 767
unsigned long	四字节	0~+4 294 967 295
signed long	四字节	-2 147 483 648~+2 147 483 647
float	四字节	$\pm 1.175494\text{E}-38 \sim \pm 3.402823\text{E}+38$
*	1~3 字节	对象的地址
bit	位	0 或 1
sfr	单字节	0~255
sfr16	双字节	0~65 536
sbit	位	0 或 1

在 C 语言程序的表达式或变量赋值运算中, 有时会出现运算对象的数据不一致的情况, C 语言允许任何标准数据类型之间的隐式转换。隐式转换按以下优先级别自动进行:

bit → char → int → long → float

signed → unsigned

其中箭头方向仅表示数据类型级别的高低, 转换时由低向高进行, 而不是数据转换时的顺序。例如, 将一个 bit (位类型) 变量赋给一个 int (整型变量) 时, 不需要先将 bit 型变量转换成 char 型之后再转换成 int 型, 而是将 bit 型变量直接转换成 int 型并完成赋值运算。一般来说, 如果有几个不同类型的数据同时参加运算, 先将低级别类型的数据转换成高级别类型, 再作运算处理, 并且运算结果为高级别类型数据。C 语言除了能对数据类型作自动的隐式转换之外, 还可以采用强制类型转换符“()”对数据类型作显式转换, 强制类型转换符“()”的应用将在 2.2.5 节中介绍。

Keil Cx51 编译器除了能支持以上这些基本数据之外, 还能支持复杂的构造类型数据, 如结构类型、联合类型等。这些复杂的数据类型将在本书第 5 章详细讨论。

2.2.2 常量

常量又称为标量，它的值在程序执行过程中不能改变。常量的数据类型有整型、浮点型、字符型和字符串型等。

1. 整型常量

整型常量就是整型常数，可表示为以下几种形式。

十进制整数：如 1234，-5678，0 等。

十六进制整数：ANSI C 标准规定十六进制数据以 0x 开头，数字为 0~9，a~f。如 0x123 表示十六进制数，相当于十进制数 291。-0x1a 表示十六进制数，相当于十进制数-26。

长整数：在数字后面加一个字母 L 就构成了长整数，如 2048L、0123L、0xff00L 等。

2. 浮点型常量

浮点型常量有十进制数表示形式和指数表示形式。十进制数表示形式又称定点表示形式，由数字和小数点组成。如 0.3141、.3141、314.1、3141. 及 0.0 都是十进制数表示形式的浮点型常量。在这种表示形式中，如果整数或小数部分为 0 可以省略不写，但必须有小数点。指数表示形式为：

[±]数字[.数字]e[±]数字

其中，[] 中的内容为可选项，根据具体情况可有可无，但其余部分必须有。如 123e4、5e6、-7.0e-8 等都是合法的指数形式浮点型常量；而 e9、5e4.3 和 e 则都是不合法的表示形式。

3. 字符型常量

字符型常量是单引号内的字符，如 'a'，'b' 等。对于不可显示的控制字符，可以在该字符前面加一个反斜杠“\”组成转义字符。利用转义字符可以完成一些特殊功能和输出时的格式控制。常用转义字符如表 2-4 所示。

表 2-4 常用转义字符表

转义字符	含 义	ASCII 码 (16 进制数形式)
\0	空字符 (NULL)	0x00
\n	换行符 (LF)	0x0A
\r	回车符 (CR)	0x0D
\t	水平制表符 (HT)	0x09
\b	退格符 (BS)	0x08
\f	换页符 (FF)	0x0C
\'	单引号	0x27
\"	双引号	0x22
\\	反斜杠	0x5C

4. 字符串型常量

字符串型常量由双引号“”内的字符组成，如“ABCD”、“\$1234”等都是字符串常量。

当双引号内的字符个数为 0 时,称为空串常量。需要注意的是,字符串常量首尾的双引号是界限符,当需要表示双引号字符串时,可用双引号转义字符“\”来表示。另外,C 语言将字符串常量作为一个字符类型数组来处理,在存储字符串常量时要在字符串的尾部加一个转义字符\0 作为该字符串常量的结束符。因此不要将字符常量与字符串常量混淆,如字符常量‘a’与字符串常量“a”是不一样的。

2.2.3 变量及其存储模式

变量是一种在程序执行过程中其值能不断变化的量。使用一个变量之前,必须进行定义,用一个标识符作为变量名并指出它的数据类型和存储模式,以便编译系统为它分配相应的存储单元。在 Cx51 中对变量进行定义的格式如下:

[存储种类] 数据类型 [存储器类型] 变量名表;

其中,“存储种类”和“存储器类型”是可选项。变量的存储种类有四种:自动(auto)、外部(extern)、静态(static)和寄存器(register)。定义一个变量时如果省略存储种类选项,则该变量将为自动(auto)变量。定义一个变量时除了需要说明其数据类型之外,Keil Cx51 编译器还允许说明变量的存储器类型。Keil Cx51 编译器完全支持 8051 系列单片机的硬件结构和存储器组织,对于每个变量可以准确地赋予其存储器类型,使之能够在单片机系统内准确地定位。表 2-5 列出了 Keil Cx51 编译器所能识别的存储器类型。

表 2-5 Keil Cx51 编译器所能识别的存储器类型

存储器类型	说 明
DATA	直接寻址的片内数据存储器 (128B), 访问速度最快
BDATA	可位寻址的片内数据存储器 (16B), 允许位与字节混合访问
IDATA	间接访问的片内数据存储器 (256B), 允许访问全部片内地址
PDATA	分页寻址的片外数据存储器 (256B), 用 MOVX @Ri 指令访问
XDATA	片外数据存储器 (64KB), 用 MOVX @DPTR 指令访问
CODE	程序存储器 (64KB), 用 MOVC @A+DPTR 指令访问

定义变量时如果省略“存储器类型”选项,则按编译时使用的存储器模式 SMALL、COMPACT 或 LARGE 来规定默认存储器类型,确定变量的存储器空间,函数中不能采用寄存器传递的参数变量和过程变量也保存在默认的存储器空间。Keil Cx51 编译器的三种存储器模式(默认的存储器类型)对变量的影响如下。

1. SMALL

变量被定义在 8051 单片机的片内数据存储器中,对这种变量的访问速度最快。另外,所有的对象,包括堆栈,都必须位于片内数据存储器中,而堆栈的长度是很重要的,实际栈长取决于不同函数的嵌套深度。

2. COMPACT

变量被定义在分页寻址的片外数据存储器中,每一页片外数据存储器的长度为 256 字

节。这时对变量的访问是通过寄存器间接寻址 (MOVX @Ri) 进行的, 堆栈位于 8051 单片机片内数据存储器中。采用这种编译模式时, 变量的高 8 位地址由 P2 口确定, 低 8 位地址由 R0 或 R1 的内容决定。采用这种模式的同时, 必须适当改变启动配置文件 STARTUP.A51 中的参数: PDATASTART 和 PDATALEN; 在用 BL51 进行连接时还必须采用连接控制命令 “PDATA” 对 P2 口地址进行定位, 这样才能确保 P2 口为所需要的高 8 位地址。

3. LARGE

变量被定义在片外数据存储器中 (最大可达 64KB), 使用数据指针 DPTR 来间接访问变量 (MOVX @DPTR)。这种访问数据的方法效率是不高的, 尤其是对于 2 个以上字节的变量, 用这种方法相当影响程序的代码长度。

需要特别指出的是, 变量的存储种类与存储器类型是完全无关的。例如:

```
static unsigned char data x;    /* 在片内数据存储器中定义一个静态无符  
                                号字符型变量 x */  
int y;                          /* 定义一个自动整型变量 y, 它的存储器类型由  
                                编译模式确定 */
```

8051 系列单片机具有多种内部寄存器, 其中一些是特殊功能寄存器, 如定时器方式控制寄存器 TMOD、中断允许控制寄存器 IE 等。为了能够直接访问这些特殊功能寄存器, Keil Cx51 编译器扩充了关键字 sfr 和 sfr16, 利用这种扩充关键字可以在 C 语言源程序中直接对 8051 单片机的特殊功能寄存器进行定义。定义方法如下:

sfr 特殊功能寄存器名=地址常数;

例如:

```
sfr P0 = 0x80;    /* 定义 I/O 口 P0, 其地址为 0x80 */
```

这里需要注意的是, 在关键字 sfr 后面必须跟一个标识符作为寄存器名, 名字可任意选取, 但应符合一般习惯。等号后面必须是常数, 不允许有带运算符的表达式, 而且该常数必须在特殊功能寄存器的地址范围之内 (0x80~0xFF)。在新一代的 8051 单片机中, 特殊功能寄存器经常组合成 16 位来使用。采用关键字 sfr16 可以定义这种 16 位的特殊功能寄存器。例如: 对于 8052 单片机的定时器 T2, 可采用如下的方法来定义:

```
sfr16 T2 = 0xCC;    /* 定义 TIMER2, 其地址为 T2L=0xCC, T2H=0xCD */
```

这里 T2 为特殊功能寄存器名, 等号后面是它的低字节地址, 其高字节地址必须在物理上直接位于低字节之后。这种定义方法适用于所有新一代的 8051 单片机中新增加的特殊功能寄存器。

在 8051 单片机应用系统中经常需要访问特殊功能寄存器中的某些位, Keil Cx51 编译器为此提供了一个扩充关键字 sbit, 利用它定义可位寻址对象。定义方法有如下三种。

(1) sbit 位变量名=位地址

这种方法将位的绝对地址赋给位变量, 位地址必须位于 0x80~0xFF 之间。例如:

```
sbit OV = 0xD2;
```

```
sbit CY = 0xD7;
```

(2) sbit 位变量名=特殊功能寄存器名^位位置

当可寻址位位于特殊功能寄存器中时可采用这种方法，“位位置”是一个 0~7 之间的常数。例如：

```
sfr PSW = 0xD0;
sbit OV = PSW^2;
sbit CY = PSW^7;
```

(3) sbit 位变量名=字节地址^位位置

这种方法以一个常数（字节地址）作为基地址，该常数必须在 0x80H~0xFF 之间。“位位置”是一个 0~7 之间的常数。例如：

```
sbit OV = 0xD0^2;
sbit CY = 0xD0^7;
```

当位对象位于 8051 单片机片内存储器中可位寻址区时称之为“可位寻址对象”。Keil Cx51 编译器提供了一个 bdata 存储器类型，允许将具有 bdata 类型的对象放入 8051 单片机片内可位寻址区。例如：

```
int bdata ibase;           /*在位寻址区定义一个整型变量 ibase */
char bdata bary[4];       /*在位寻址区定义一个数组 array[4]  */
```

使用关键字 sbit 可以独立访问可位寻址对象中的某一位。例如：

```
sbit mybit0=ibase^0;
sbit mybit15=ibase^15;
sbit Ary07=bary[0]^7;
sbit Ary37=bary[3]^7;
```

采用这种方法定义可位寻址变量时要求基址对象的存储器类型为 bdata，操作符“^”后面“位位置”的最大值取决于指定的基地址类型，对于 char 类型来说是 0~7；对于 int 类型来说是 0~15；对于 long 类型来说是 0~31。

需要注意的是，sbit 是一个独立的关键字，不要将它与关键字 bit 相混淆。关键字 bit 是 Keil Cx51 编译器的一种扩充数据类型，用来定义一个普通位变量，它的值是二进制数的 0 或 1。一个函数中可以包含 bit 类型的参数，函数的返回值也可为 bit 类型。例如：

```
static bit direction_bit    /* 定义一个静态位变量 direction_bit */
extern bit lock_prt_port    /* 定义一个外部位变量 lock_prt_port */
bit bfunc(bit b0,bit b1);   /* 定义一个返回位型值的函数 bfunc，函
    {...                      数中包含有两个位型参数 b0 和 b1 */
    return(b1)              /* 返回一个位型值 b1 */
}
```

如果在函数中禁止使用中断（#pragma disable）或者函数中包含有明确的寄存器组切换（using n），则该函数不能返回位型值，否则在编译时会产生编译错误。另外，不能定

义位指针，也不能定义位数组。

上面介绍了变量及其定义方法，这在编写 C 语言程序时是十分重要的。从变量的作用范围来看，还有全局变量和局部变量之分。全局变量是指在程序开始处或各个功能函数的外面定义的变量，在程序开始处定义的全局变量对于整个程序都有效，可供程序中所有函数共同使用；而在各功能函数外面定义的全局变量只对从定义处开始往后的各个函数有效，只有从定义处往后的那些功能函数才可以使用该变量，定义处前面的函数则不能使用它。

局部变量是指在函数内部或以花括号{ }围起来的功能块内部所定义的变量，局部变量只在定义它的函数或功能块以内有效，在该函数或功能块以外则不能使用它。局部变量可以与全局变量同名，但在这种情况下局部变量的优先级较高，而同名的全局变量在该功能块内被暂时屏蔽。

从变量的存在时间来看又可分为静态存储变量和动态存储变量。

静态存储变量是指该变量在程序运行期间其存储空间固定不变；动态存储变量是指该变量的存储空间不确定，在程序运行期间根据需要动态地为该变量分配存储空间。一般来说全局变量为静态存储变量，局部变量为动态存储变量。

在进行程序设计的时候经常需要给一些变量赋以初值，C 语言允许在定义变量的同时给变量赋初值。下面是一些变量定义的例子。

```
char data var1;          /* 在 data 区定义字符型变量 var1 */
int  idata var2;         /* 在 idata 区定义整型变量 var2 */
int  a=5;                /* 定义变量 a，同时赋以初值 5，变量 a 位于
                           由编译模式确定的默认存储区 */

char code text[ ]="ENTER PARAMETER:"; /* 在 code 区定义字符串数组 */
unsigned char xdata vector [10][4][4]; /* 在 xdata 区定义无符号字符型三维数组
                                           变量 vector[10][4][4] */
static unsigned long xdata array [100]; /* 在 xdata 区定义静态无符号长整型数组
                                           变量 array[100] */

extern float idata x,y,z; /* 在 idata 区定义外部浮点型变量 x,y,z */
char xdata * px;         /* 在 xdata 区定义一个指向对象类型为
                           char 的指针 px，指针 px 自身在默认存
                           储区（由编译模式确定），长度为 2 字节
                           （0~0xFFFF） */

char xdata * data pdx;   /* 除了指针明确位于内部数据存储区
                           (data) 之外，与上例完全相同，由于指定
                           了存储器类型，所以与编译模式无关 */

extern bit data lock_prt_port; /* 在 data 区定义一个外部位变量 */
char bdata flags;           /* 在 bdata 区定义字符型变量 */
sbit flag0=flags^0;        /* 在 bdata 区定义可位寻址变量 */
sfr P0=0x80;               /* 定义特殊功能寄存器 P0 */
sfr16 T2=0xCC;             /* 定义特殊功能寄存器 T2 */
```

2.2.4 用 typedef 重新定义数据类型

在 C 语言程序中除了可以采用上面所介绍的数据类型之外,用户还可以根据自己的需要对数据类型重新定义。重新定义时需用到关键字 `typedef`,定义方法如下:

`typedef` 已有的数据类型 新的数据类型名;

其中“已有的数据类型”是指上面所介绍的 C 语言中所有的数据类型,包括结构、指针和数组等,“新的数据类型名”可按用户自己的习惯或根据任务需要决定。关键字 `typedef` 的作用只是将 C 语言中已有的数据类型作了置换,因此可用置换后的新数据类型名来进行变量的定义。例如:

```
typedef int word;          /* 定义 word 为新的整型数据类型名 */
word i, j;                 /* 将 i, j 定义为 int 型变量 */
```

在这个例子中,先用关键字 `typedef` 将 `word` 定义为新的整型数据类型,定义的过程实际上是用 `word` 置换了 `int`,因此下面就可以直接用 `word` 对变量 `i, j` 进行定义,而此时 `word` 等效于 `int`,所以 `i, j` 被定义成整型变量。例如:

```
typedef int NUM[100] ;    /* 将 NUM 定义为整型数组类型 */
NUM n;                   /* 将 n 定义为整型数组变量 */
typedef char * POINTER;  /* 将 POINTER 定义为字符指针类型 */
POINTER point;           /* 将 point 定义为字符指针变量 */
```

用 `typedef` 还可以定义结构类型:

```
typedef struct            /* 定义结构体 */
{ int month;
  int day;
  int year;
} DATE;
```

这里 `DATE` 为一个新的数据类型(结构类型)名,可以直接用它来定义变量:

```
DATE birthday;           /* 定义 birthday 为结构类型变量 */
DATE * point;            /* 定义指向这个结构类型数据的指针 */
```

关于结构类型数据在本章后面还要详细讨论。一般而言,用 `typedef` 定义的新数据类型用大写字母表示,以便与 C 语言中原有的数据类型相区别。另外还要注意,用 `typedef` 可以定义各种新的数据类型名,但不能直接用来定义变量。`typedef` 只是对已有的数据类型作了一个名字上的置换,并没有创造出一个新的数据类型,例如前面例子中的 `word`,它只是 `int` 类型的一个新名字而已。采用 `typedef` 来重新定义数据类型有利于程序的移植,同时还可以简化较长的数据类型定义(如结构数据类型等)。在采用多模块程序设计时,如果不同的模块程序源文件中用到同一类型的数据时(尤其是像数组、指针、结构、联合等复杂数据类型),经常用 `typedef` 将这些数据重新定义并放到一个单独的文件中,需要时再用预处理命令 `#include` 将它们包含进来。

2.2.5 运算符与表达式

C 语言对数据有很强的表达能力, 具有十分丰富的运算符。运算符就是完成某种特定运算的符号, 表达式则是由运算符及运算对象所组成的具有特定含义的一个式子。C 语言是一种表达式语言, 在任意一个表达式的后面加一个分号“;”就构成了一个表达式语句。由运算符和表达式可以组成 C 语言程序的各种语句。

运算符按其在表达式中所起的作用, 可分为赋值运算符、算术运算符、增量与减量运算符、关系运算符、逻辑运算符、位运算符、复合赋值运算符、逗号运算符、条件运算符、指针和地址运算符、强制类型转换运算符和 sizeof 运算符等。运算符按其在表达式中与运算对象的关系, 又可分为单目运算符、双目运算符和三目运算符等。单目运算符只需要有一个运算对象, 双目运算符要求有两个运算对象, 三目运算符要求有三个运算对象。掌握各种运算符的意义和使用规则, 对于编写正确的 C 语言程序是十分重要的。

1. 赋值运算符

在 C 语言中, 符号“=”是一个特殊的运算符, 称之为赋值运算符。赋值运算符的作用是将一个数据的值赋给一个变量, 利用赋值运算符将一个变量与一个表达式连接起来的式子称为赋值表达式, 在赋值表达式的后面加一个分号“;”便构成了赋值语句。赋值语句的格式如下:

变量 = 表达式;

该语句的意义是先计算出右边表达式的值, 然后将该值赋给左边的变量。上式中的“表达式”还可以是另一个赋值表达式, 即 C 语言允许进行多重赋值。例如:

```
x=9;          /* 将常数 9 赋给变量 x */  
x=y=8;        /* 将常数 8 同时赋给变量 x 和 y */
```

都是合法的赋值语句。在使用赋值运算符“=”时应注意不要与关系运算符“==”相混淆, 运算符“==”用来进行相等关系运算。

2. 算术运算符

C 语言中的算术运算符有:

- + 加或取正值运算符
- 减或取负值运算符
- * 乘运算符
- / 除运算符
- % 取余运算符

上面这些运算符中加、减、乘、除为双目运算符, 它们要求有两个运算对象。对于加、减和乘法符合一般的算术运算规则。除法运算有所不同, 如果是两个整数相除, 其结果为整数, 舍去小数部分, 例如: 5/3 的结果为 1, 5/10 的结果为 0。如果是两个浮点数相除, 其结果为浮点数, 例如: 5.0/10.0 的结果为 0.5。取余运算要求两个运算对象均为整型数据, 例如: 7%4 的结果为 3。取正值和取负值为单目运算符, 它们的运算对象只有一个, 分别是取运算对象的正值和负值。

用算术运算符将运算对象连接起来的式子即为算术表达式。算术运算的一般形式为：

表达式 1 算术运算符 表达式 2

例如： $x+y/(a-b)$ ， $(a+b)*(x-y)$ 都是合法的算术表达式。C 语言中规定了运算符的优先级和结合性。在求一个表达式的值时，要按运算符的优先级别进行。算术运算符中取负值（-）的优先级最高，其次是乘法（*）、除法（/）和取余（%）运算符，加法（+）和减法（-）运算符的优先级最低。需要时可在算术表达式中采用圆括号来改变运算符的优先级，例如在计算表达式 $x+y/(a-b)$ 的值时，首先计算 $(a-b)$ ，然后再计算 $y/(a-b)$ ，最后计算 $x+y/(a-b)$ 。如果在一个表达式中各个运算符的优先级别相同，则计算时按规定的结合方向进行。例如计算表达式 $x+y-z$ 的值，由于+和-优先级别相同，计算时按“从左至右”的结合方向，先计算 $x+y$ ，再计算 $(x+y)-z$ 。这种“从左至右”的结合方向称为“左结合性”，此外还有“右结合性”。

3. 增量和减量运算符

C 语言中除了基本的加、减、乘、除运算符之外，还提供一种特殊的运算符：

++ 增量运算符

-- 减量运算符

增量和减量是 C 语言中特有的一种运算符，它们的作用分别是对运算对象作加 1 和减 1 运算。例如： $++i$ ， $i++$ ， $--j$ ， $j--$ 等。

看起来 $++i$ 和 $i++$ 的作用都是使变量 i 的值加 1，但是由于运算符++所处的位置不同，使变量 i 加 1 的运算过程也不同。 $++i$ （或 $--i$ ）是先执行 $i+1$ （或 $i-1$ ）操作，再使用 i 的值，而 $i++$ （或 $i--$ ）则是先使用 i 的值，再执行 $i+1$ （或 $i-1$ ）操作。

增量运算符++和减量运算符--只能用于变量，不能用于常数或表达式。

例 2.1：使用增量“++”和减量“--”运算符的例子。

```
#include <stdio.h>
main(){
    int x,y,z;
    x = y = 8;  z = ++x;
    printf("\n %d %d %d",y,z,x);
    x = y = 8;  z = x++;
    printf("\n %d %d %d",y,z,x);
    x = y = 8;  z = --x;
    printf("\n %d %d %d",y,z,x);
    x = y = 8;  z = x--;
    printf("\n %d %d %d",y,z,x);
    printf("\n");
    while(1);
}
```

程序执行结果：

```
8 9 9
8 8 9
8 7 7
8 8 7
```

在这个程序例子中使用了 Keil Cx51 编译器提供的输出库函数 `printf`, 在 C 语言程序中凡是使用了库函数的, 都必须在程序开始处将该库函数的预定义文件包含进来, 才能使程序得到正确的编译和执行。本程序在开始处使用了预处理命令 `#include` 将声明库函数 `printf` 原型的头文件 `stdio.h` 包含到程序中去。另外, 为了使库函数 `printf` 能够在 μ Vision2 仿真调试状态下正确工作, 应在 C 语言源程序中增加对 8051 单片机串行口初始化的语句, 或者将 C 语言源程序与修改后(加入了 8051 单片机串口初始化指令)的启动程序 `STARTUP.A51` 连接在一起。关于输入输出库函数的详细介绍请参见本书第 9 章。

4. 关系运算符

C 语言中有 6 种关系运算符:

> 大于

< 小于

>= 大于等于

<= 小于等于

== 等于

!= 不等于

前 4 种关系运算符具有相同的优先级, 后两种关系运算符也具有相同的优先级; 但前 4 种的优先级高于后 2 种。用关系运算符将两个表达式连接起来即成为关系表达式。关系表达式的一般形式为:

表达式 1 关系运算符 表达式 2

例如: $x > y$ 、 $x + y > z$ 、 $(x = 3) > (y = 4)$ 都是合法的关系表达式。

关系运算符通常用来判别某个条件是否满足, 关系运算的结果只有 0 和 1 两种值。当所指定的条件满足时结果为 1, 条件不满足时结果为 0。

例 2.2: 使用关系运算符的例子。

```
#include <stdio.h>
main() {
    int x,y,z;
    printf("input data x,y ? \n");
    scanf("%d %d",&x,&y);
    printf("\n    x    y    x<y  x<=y x>y  x>=y x!=y x==y");
    printf("\n%5d%5d",x,y);
    z = x < y; printf("%5d",z);
    z = x <= y; printf("%5d",z);
    z = x > y; printf("%5d",z);
    z = x >= y; printf("%5d",z);
    z = x != y; printf("%5d",z);
    z = x == y; printf("%5d",z);
    printf("\n");
    while(1);
}
```

程序执行结果 (1):

input data x, y ?

5 3 回车

x	y	x<y	x<=y	x>y	x>=y	x!=y	x==y
5	3	0	0	1	1	1	0

程序执行结果 (2):

input data x, y ?

-5 -3 回车

x	y	x<y	x<=y	x>y	x>=y	x!=y	x==y
-5	-3	1	1	0	0	1	0

程序执行结果 (3):

input data x, y ?

4 4 回车

x	y	x<y	x<=y	x>y	x>=y	x!=y	x==y
4	4	0	1	0	1	0	1

在本例中使用了 Keil Cx51 编译器提供的输入库函数 scanf, 与 printf 函数一样, scanf 也是通过 8051 单片机的串行口实现数据输入的, 它的使用方法与 printf 函数类似。

5. 逻辑运算符

C 语言中有 3 种逻辑运算符:

|| 逻辑或

&& 逻辑与

! 逻辑非

逻辑运算符用来求某个条件式的逻辑值, 用逻辑运算符将关系表达式或逻辑量连接起来就是逻辑表达式。逻辑运算的一般形式为:

逻辑与: 条件式1 && 条件式2

逻辑或: 条件式1 || 条件式2

逻辑非: ! 条件式

例如: x&& y、allb、!z 都是合法的逻辑表达式。

进行逻辑与运算时, 首先对条件式 1 进行判断, 如果结果为真 (非 0 值), 则继续对条件式 2 进行判断, 当结果也为真时, 表示逻辑运算的结果为真 (值为 1); 反之, 如果条件式 1 的结果为假, 则不再判断条件式 2, 而直接给出逻辑运算的结果为假 (值为 0)。进行逻辑或运算时, 只要两个条件式中有一个为真, 逻辑运算的结果便为真 (值为 1), 只有当条件式 1 和条件式 2 均不成立时, 逻辑运算的结果才为假 (值为 0)。进行逻辑非运算时, 对条件式的逻辑值直接取反。逻辑运算符的优先级为 (由高至低): ! (非) → && (与) → || (或), 即逻辑非的优先级最高。

例 2.3: 使用逻辑运算符的例子。

```
#include <stdio.h>
main() {
    int x,y,z;
```



```

printf("input data x,y ? \n");
scanf("%d %d",&x,&y);
printf("\n    x    y    !x    x||y    x&& y");
printf("\n%5d%5d",x,y);
z = !x;    printf("%8d",z);
z = x || y; printf("%8d",z);
z = x && y; printf("%8d",z);
printf("\n");
while(1);
}

```

程序执行结果 (1):

```

input data x, y ?
12 8 回车
x    y    !x    x||y    x&&y
12 8    0    1    1

```

程序执行结果 (2):

```

input data x, y ?
9 -3 回车
x    y    !x    x||y    x&&y
9 -3    0    1    1

```

程序执行结果 (3):

```

input data x, y ?
0 81 回车
x    y    !x    x||y    x&&y
0 81    1    1    0

```

程序执行结果 (4):

```

input data x, y ?
-23 0 回车
x    y    !x    x||y    x&&y
-23 0    0    1    0

```

程序执行结果 (5):

```

input data x, y ?
0 0 回车
x    y    !x    x||y    x&&y
0 0    1    0    0

```

6. 位运算符

能对运算对象进行按位操作是 C 语言的一大特点,正是由于这一特点使 C 语言具有了汇编语言的一些功能,从而使之能对计算机的硬件直接进行操作。C 语言中共有 6 种位运算符:

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

位运算符的作用是按位对变量进行运算，并不改变参与运算的变量的值。若希望按位改变运算变量的值，则应利用相应的赋值运算。另外位运算符不能用来对浮点型数据进行操作。位运算符的优先级从高到低依次是：按位取反（~）→左移（<<）和右移（>>）→按位与（&）→按位异或（^）→按位或（|）。位运算的一般形式如下：

变量1 位运算符 变量2

表 2-6 列出了按位取反、按位与、按位或和按位异或的逻辑真值。

表 2-6 按位取反、按位与、按位或和按位异或的逻辑真值

x	y	~x	~y	x&y	x y	x^y
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

例 2.4：位逻辑运算。

```

#include <stdio.h>
main() {
    unsigned int x = 0x57db, y = 0xb0f3;
    printf("\n x y x&y x^y x|y ~x");
    printf("\n%6x%6x%6x%6x%6x%6x", x, y, x&y, x^y, x|y, ~x);
    printf("\n");
    while(1);
}

```

程序执行结果：

```

x      y      x&y      x^y      x|y      ~x
57db b0f3  10d3  e728  f7fb  a824

```

位运算符中的移位操作比较复杂，左移（<<）运算符是用来将变量1的二进制位值向左移动由变量2所指定的位数。例如：a=0x8f（即二进制数10001111），进行左移运算a<<2，就是将a的全部二进制位值一起向左移动2位，其左端移出的位值被丢弃，并在其右端补以相应位数的“0”。因此，移位的结果是a=0x3c（即二进制数00111100）。右移（>>）运算符是用来将变量1的二进制位值向右移动由变量2指定的位数。进行右移运算时，如果变量1属于无符号类型数据，则总是在其左端补“0”；如果变量1属于有符号类型数据，则在其左端补入原来数据的符号位（即保持原来的符号不变），其右端的移出位被丢弃。对于a=0x8f，如果a是无符号数，则执行a>>2之后结果为a=0x23（即二进制数00100011）；

如果 a 是有符号数, 则执行 $a >> 2$ 之后结果为 $a=0xe3$ (即二进制数 11100011)。

例 2.5: 移位运算。

```
#include <stdio.h>
main() {
    int a,b;
    unsigned int x,y;
    a = b = 0xaa55; x = y = 0xaa55;
    printf("\n a=%4x b=%4x x=%4x y=%4x",a,b,x,y);
    a = a << 1; b = b >> 1;
    x = x << 1; y = y >> 1;
    printf("\n a=%4x b=%4x x=%4x y=%4x",a,b,x,y);
    printf("\n");
    while(1);
}
```

程序执行结果:

a=aa55	b=aa55	x=aa55	y=aa55
a=54aa	b=d52a	x=54aa	y=552a

7. 复合赋值运算符

在赋值运算符“=”的前面加上其他运算符, 就构成了所谓复合赋值运算符:

+= 加法赋值

-= 减法赋值

*= 乘法赋值

/= 除法赋值

%= 取模赋值

<<= 左移位赋值

>>= 右移位赋值

&= 逻辑与赋值

|= 逻辑或赋值

^= 逻辑异或赋值

~= 逻辑非赋值

复合赋值运算首先对变量进行某种运算, 然后将运算的结果再赋给该变量。复合运算的一般形式为:

变量 复合赋值运算符 表达式

例如: $a+=3$ 等价于 $a=a+3$; $x*=y+8$ 等价于 $x=x*(y+8)$ 。凡是二目运算符, 都可以和赋值运算符一起组合成复合赋值运算符。采用复合赋值运算符, 可以使程序简化, 同时还可以提高程序的编译效率。

例 2.6: 利用复合赋值运算符实现算术运算。

```
#include <stdio.h>
main() {
```

```

int a,b,c,d,x,y,z;
x = 634; y = 19; z = 28;
a = 3 * (b = x/(y-4)) - z/2;
printf("\n%10d%10d", a, b);
a = 100; b = 45; c = -19; d = 94; x = -2; y = 5;
a += 6;
b -= x;
c *= 10;
d /= x+y;
z %= 8;
printf("\n%10d%10d%10d%10d%10d",a,b,c,d,z);
printf("\n");
while(1);
}

```

程序执行结果:

```

112    42
106    47   -190    31    4

```

例 2.7: 利用复合赋值运算符实现位逻辑运算。

```

#include <stdio.h>
main() {
    int x,y;
    x = 2; y = 3;
    x <<= 2; printf("\n%3d",x);
    x >>= 1; printf("\n%3d",x);
    x <<= y; printf("\n%3d",x);
    x = 2;
    x &= y; printf("\n%3d",x);
    x |= y; printf("\n%3d",x);
    x ^= y; printf("\n%3d",x);
    printf("\n");
    while(1);
}

```

程序执行结果:

```

8
4
32
2
3
0

```

8. 逗号运算符

在 C 语言中逗号“,”是一个特殊的运算符,可以用它将两个(或多个)表达式连接起来,称为逗号表达式。逗号表达式的一般形式为:

表达式 1, 表达式 2, ..., 表达式 n

程序运行时对于逗号表达式的处理,是从左至右依次计算出各个表达式的值,而整个逗号表达式的值是最右边表达式(即表达式 n)的值。

例 2.8: 逗号运算符的使用。

```
#include <stdio.h>
main() {
    int a,b,c,w,x,y,z;
    w = ( x=5,y=-11,z=43,3);
    printf("\n%d %d %d %d",w,x,y,z);
    a = 3 * (b = w + x,c = y * ( z - 10)) - 6;
    printf("\n%d %d %d",a,b,c);
    printf("\n");
    while(1);
}
```

程序执行结果:

```
3 5 -11 43
-1095 8 -363
```

在许多情况下,使用逗号表达式的目的只是为了分别得到各个表达式的值,而并不一定要得到和使用整个逗号表达式的值。另外还要注意,并不是在程序的任何地方出现的逗号,都可以认为是逗号运算符。例如函数中的参数也是用逗号来间隔的,上例中库输出函数 `printf("\n%d %d %d",a,b,c)` 中的“a,b,c”是函数的三个参数,而不是一个逗号表达式。

9. 条件运算符

条件运算符“?:”是 C 语言中惟一的一个三目运算符,它要求有三个运算对象,用它可以将三个表达式连接构成一个条件表达式。条件表达式的一般形式如下:

逻辑表达式 ? 表达式 1: 表达式 2

其功能是首先计算逻辑表达式,当值为真(非 0 值)时,将表达式 1 的值作为整个条件表达式的值;当逻辑表达式的值为假(0 值)时,将表达式 2 的值作为整个条件表达式的值。例如:条件表达式 `max=(a>b)?a:b` 的执行结果是将 a 和 b 中较大者赋值给变量 max。另外,条件表达式中逻辑表达式的类型可以与表达式 1 和表达式 2 的类型不一样。

10. 指针和地址运算符

指针是 C 语言中的一个十分重要的概念,在 C 语言的数据类型中专门有一种指针类型。变量的指针就是该变量的地址,还可以定义一个指向某个变量的指针变量。为了表示指针变量和它所指向的变量地址之间的关系, C 语言提供了两个专门的运算符:

* 取内容

& 取地址

取内容和取地址运算的一般形式分别为:

变量 = * 指针变量

指针变量 = & 目标变量

取内容运算的含义是将指针变量所指向的目标变量的值赋给左边的变量;取地址运算的含义是将目标变量的地址赋给左边的变量。需要注意的是,指针变量中只能存放地址(即指针型数据),不要将一个非指针类型的数据赋值给一个指针变量。

例 2.9: 指针及地址运算符的使用。

```
#include <stdio.h>
main() {
    int i;
    int *int_ptr;
    int_ptr = &i;
    *int_ptr = 5;
    printf("\n i = %d",i);
    while(1);
}
```

程序执行结果:

i=5

11. 强制类型转换运算符

C 语言中的圆括号“()”也可作为一种运算符使用,这就是强制类型转换运算符,它的作用是将表达式或变量的类型强制转换成为所指定的类型。在 C 语言程序中进行算术运算时,需要注意数据类型的转换。有两种数据类型转换方式,即隐式转换和显式转换。隐式转换是在对程序进行编译时由编译器自动处理的。隐式转换遵循以下规则。

① 所有 char 型的操作数转换成 int 型。

② 用运算符连接的两个操作数如果具有不同的数据类型,按以下次序进行转换:如果一个操作数是 float 类型,则另一个操作数也转换成 float 类型;如果一个操作数是 long 类型,则另一个操作数也转换成 long 类型;如果一个操作数是 unsigned 类型,则另一个操作数也转换成 unsigned 类型。

③ 在对变量赋值时发生的隐式转换,将赋值号“=”右边的表达式类型转换成赋值号左边变量的类型。例如,把整型数赋值给字符型变量,则整型数的高 8 位将丧失;把浮点数赋值给整型变量,则小数部分将丧失。在 C 语言中只有基本数据类型(即 char、int、long 和 float)可以进行隐式转换。其余的数据类型不能进行隐式转换,例如,我们不能把一个整型数利用隐式转换赋值给一个指针变量,在这种情况下就必须利用强制类型转换运算符来进行显式转换。强制类型转换运算符的一般使用形式为:

(类型)=表达式

显式类型转换在给指针变量赋值时特别有用。例如,预先在 8051 单片机的片外数据存储(xdata)中定义了一个字符型指针变量 px,如果想给这个指针变量赋一初值 0xB000,可以写成:px=(char xdata *)0xB000;这种方法特别适合于用标识符来存取绝对地址。

例 2.10: 强制类型转换运算符的使用。

```
#include <stdio.h>
main() {
    char xdata * px;
    char q;
    int x = 0xf32a;
    long y = 0x901af364;
    float z = 3.14159;
    px=(char xdata *)0xB000;
    * px='A';
    q=*((char xdata *)0xB000);
    printf("\n%bx %x %d %c", (char)x, (int)y, (int)z, q) ;
    while(1);
}
```

程序执行结果:

2a f364 3 A

12. sizeof 运算符

C 语言中提供了一种用于求取数据类型、变量以及表达式的字节数的运算符: sizeof, 该运算符的一般使用形式为:

Sizeof (表达式) 或 sizeof (数据类型)

应该注意的是, sizeof 是一种特殊的运算符, 不要错误地认为它是一个函数。实际上, 字节数的计算在程序编译时就完成了, 而不是在程序执行的过程中才计算出来的。

例 2.11: sizeof 运算符的使用。

```
#include <stdio.h>
main() {
    printf("\n char: %bd byte", sizeof(char));
    printf("\n int: %bd bytes", sizeof(int));
    printf("\n long: %bd bytes", sizeof(long));
    printf("\n float: %bd bytes", sizeof(float));
    while(1);
}
```

程序执行结果:

char: 1 byte
int: 2 bytes
long: 4 bytes
float: 4 bytes

前面对 C 语言中的各种运算符分别作了介绍, 此外还有三个运算符: 数组下标运算符 “[]”、存取结构或联合中变量的运算符 “->” 或 “.”, 它们将在第 5 章予以介绍。表 2-7 给出了这些运算符在使用过程中的优先级和结合性。

表 2-7 运算符的优先级和结合性

优 先 级	类 别	运算符名称	运 算 符	结 合 性
1	强制转换 数组 结构, 联合	强制类型转换 下标 存取结构或联合成员	() [] -> 或 .	右结合
2	逻辑 字位 增量 减量 指针 算术 长度计算	逻辑非 按位取反 增 1 减 1 取地址 取内容 单目减 长度计算	! ~ ++ -- & * - sizeof	左结合
3	算术	乘 除 取模	* / %	右结合
4	算术和 指针运算	加 减	+ -	
5	字位	左移 右移	<< >>	
6	关系	大于等于 大于 小于等于 小于	>= > <= <	
7		恒等于 不等于	== !=	
8	字位	按位与	&	
9		按位异或	^	
10		按位或		
11	逻辑	逻辑与	&&	
12		逻辑或		右结合
13	条件	条件运算	?:	左结合
14	赋值	赋值 复合赋值	= op =	
15	逗号	逗号运算	,	右结合

2.3 Cx51 程序的基本语句

2.3.1 表达式语句

C 语言是一种结构化的程序设计语言，它提供了十分丰富的程序控制语句。表达式语句是最基本的一种语句。在表达式的后边加一个分号“;”就构成了表达式语句。下面的语句都是合法的表达式语句：

```
a=++b*9;
x=8; y=7;
z=(x+y)/a;
++i;
```

表达式语句也可以仅由一个分号“;”组成，这种语句称为空语句。空语句是表达式语句的一个特例。空语句在程序设计中有时是很有用的，当程序在语法上需要有一个语句，但在语义上并不要求有具体的动作时，便可以采用空语句。空语句通常有以下两种用法。

① 在程序中为有关语句提供标号，用以标记程序执行的位置。例如，采用下面的语句可以构成一个循环。

```
repeat;;
...
goto repeat ;
```

② 在用 while 语句构成的循环语句后面加一个分号，形成一个不执行其他操作的空循环体。这种空语句在等待某个事件发生时特别有用。例如，下面这段程序是读取 8051 单片机串行口数据的函数，其中就用了一个空语句 while (!RI); 来等待单片机串行口接收结束。

```
#include <reg51.h>      /* 插入 8051 单片机的预定义文件 */
char _getkey ()         /* 函数定义 */
{                       /* 函数体开始 */
    char c;             /* 定义变量 */
    while (!RI);        /* 空语句，等待8051单片机串行口接收结束 */
    c = SBUF;           /* 读串行口内容 */
    RI = 0;             /* 清除串行口接收标志 */
    return (c);         /* 返回 */
}                       /* 函数体结束 */
```

采用分号“;”作为空语句使用时，要注意与简单语句中有效组成部分的分号相区别。不能滥用空语句，以免引起程序的误操作，甚至造成程序语法上的错误。

2.3.2 复合语句

复合语句是由若干条语句组合而成的一种语句，它是用一个大括号“{ }”将若干条语句组合在一起而形成的一种功能块。复合语句不需要以分号“;”结束，但它内部的各条

单语句仍需以分号“;”结束。复合语句的一般形式为:

```
{
    局部变量定义;
    语句 1;
    语句 2;
    :
    语句 n;
}
```

复合语句在执行时,其中的各条单语句依次顺序执行。整个复合语句在语法上等价于一条单语句,因此在C语言程序中可以将复合语句视为一条单语句。复合语句允许嵌套,即在复合语句内部还可以包含别的复合语句。通常复合语句都出现在函数中,实际上,函数的执行部分(即函数体)就是一个复合语句。复合语句中的单语句一般是可执行语句,此外还可以是变量的定义语句(说明变量的数据类型)。在复合语句内所定义的变量,称为该复合语句中的局部变量,它仅在当前这个复合语句中有效。利用复合语句将多条单语句组合在一起,以及在复合语句中进行局部变量定义是C语言的一个重要特征。

例 2.12: 复合语句及其局部变量的使用。

```
#include <stdio.h>
main() {                                /* 主函数体开始 */
    int a,b,c,d;                        /* 定义变量 a, b, c, d, 它们在整个主函数中有效 */
    a = 1; b = 2; c = 3; d = 4;
    printf("\nX: %d %d %d %d",a,b,c,d);
    {                                    /* 复合语句1 */
        int b,m;                        /* 定义局部变量 b, m, 它们仅在复合语句1 中有效 */
        b = 8; m = 100;
        printf("\nY: %d %d %d %d | %d",a,b,c,d,m);
        {                                /* 复合语句2 */
            int c,n;                    /* 定义局部变量 c, n, 它们仅在复合语句2中有效 */
            c = 9; n = 150;
            printf("\nZ: %d %d %d %d | %d %d",a,b,c,d,m,n);
        }                                /* 复合语句2 结束 */
        printf("\nY: %d %d %d %d | %d",a,b,c,d,m);
    }                                    /* 复合语句1 结束 */
    printf("\nX: %d %d %d %d",a,b,c,d);
    printf("\n");
    while(1);
}
```

程序执行结果:

```
X: 1 2 3 4
Y: 1 8 3 4 | 100
Z: 1 8 9 4 | 100 150
Y: 1 8 3 4 | 100
```

X: 1 2 3 4

在这个程序的主函数体开始处, 定义了变量 a, b, c, d , 它们在整个主函数体中都是有效的, 在主函数体中的复合语句 1 和复合语句 2 中都可以使用它们。另外, 在复合语句 1 中又定义了局部变量 m 和与主函数体中定义的变量同名的局部变量 b , 这种局部变量 m 和 b 仅在定义它的复合语句 1 中有效, 而且局部变量 b 的优先级高于在主函数体中定义的同名变量 b 。因此在复合语句 1 中执行 `printf` 函数输出的 b 值为 8, 而不是 2。同样, 在复合语句 2 中定义了一个与主函数体中同名的局部变量 c , 在复合语句 2 中执行 `printf` 函数输出的 c 值为 9, 而不是 3。一旦出了复合语句, 则其中的局部变量立即失效。如果有同名的局部变量, 则恢复该变量在上一层位置所定义的初值。读者可以通过仔细分析本程序的执行结果来弄清各个变量的作用范围。

2.3.3 条件语句

条件语句又称为分支语句, 它是用关键字 `if` 构成的。C 语言提供了三种形式的条件语句。

(1) `if (条件表达式) 语句`

其含义为: 若条件表达式的结果为真 (非 0 值), 就执行后面的语句; 反之若条件表达式的结果为假 (0 值), 就不执行后面的语句。这里的语句也可以是复合语句。这种条件语句的执行过程如图 2.1 (a) 所示。

(2) `if (条件表达式) 语句 1`

`else 语句 2`

其含义为: 若条件表达式的结果为真 (非 0 值), 就执行语句 1; 反之若条件表达式的结果为假 (0 值), 就执行语句 2。这里的语句 1 和语句 2 均可以是复合语句。这种条件语句的执行过程如图 2.1 (b) 所示。

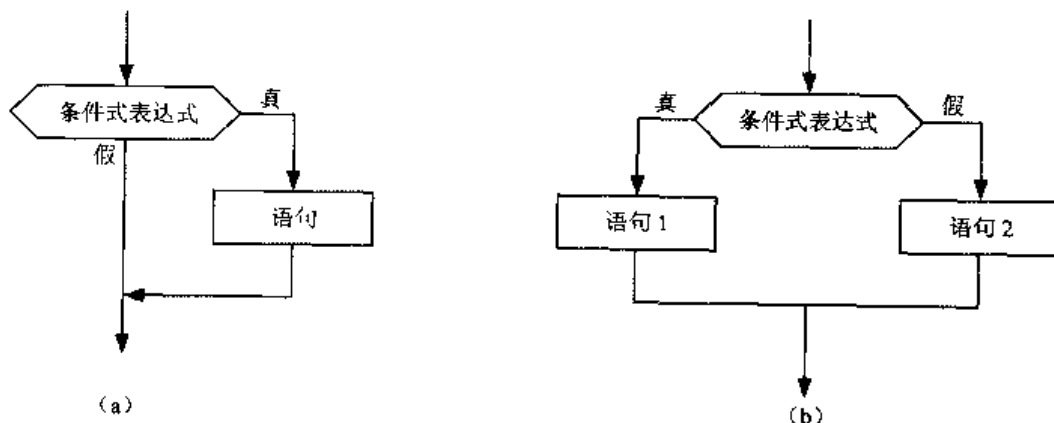


图 2.1 条件语句的执行过程

(3) <code>if (条件表达式 1)</code>	<code>语句 1</code>
<code>else if (条件式表达 2)</code>	<code>语句 2</code>
<code>else if (条件式表达 3)</code>	<code>语句 3</code>
\vdots	\vdots
<code>else if (条件表达式 n)</code>	<code>语句 m</code>

else

语句 n

这种条件语句常用来实现多方向条件分支，其执行过程如图 2.2 所示。

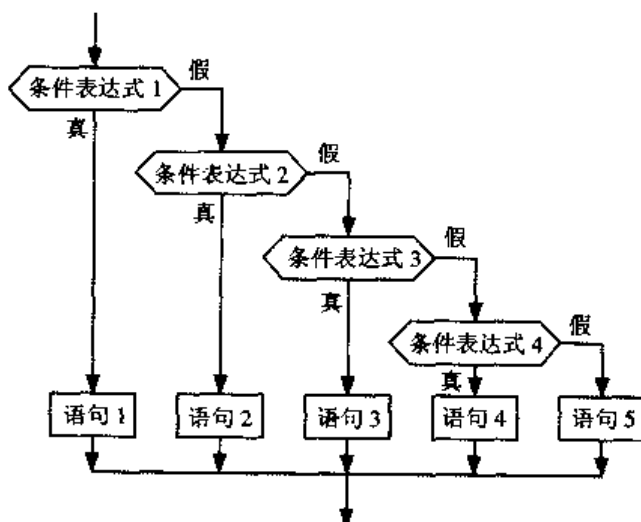


图 2.2 多分支条件语句的执行过程

例 2.13: 条件语句的使用——求一元二次方程的根。

```

#include <stdio.h>
#include <math.h>
main() {
    float a,b,c,x1,x2;
    float r,s;
    a = 2.0; b = 3.0; c = 4.0;
    r = b * b - 4.0 * a * c;
    if( r > 0.0 )
    {
        s = sqrt(r);
        x1 = (-b + s) / (2.0 * a);
        x2 = (-b - s) / (2.0 * a);
        printf("real: x1 =%15.7f,x2 =%15.7f\n",x1,x2);
    }
    else if( r == 0.0 )
        printf("double: x1,x2 =%15.7f\n",-b/(2.0*a));
    else
    {
        x1 = -b / (2.0 * a);
        x2 = sqrt(-r) / (2.0 * a);
        printf("complex: Re=%15.7f,Im=%15.7f\n",x1,x2);
    }
    while(1);
}

```

程序执行结果:

complex: Re= -0.7500000, Im= 1.1989580

在这个程序中使用了库函数 `sqrt(r)` 来求方程的根, `sqrt` 是一个算术库函数。为了使程序能得到正确的编译和执行, 在本程序的开始处使用了预处理命令 `#include` 将库函数 `sqrt` 所在的预处理文件 `math.h` 包含到程序中去。

2.3.4 开关语句

开关语句也是一种用来实现多方向条件分支的语句。虽然采用条件语句也可以实现多方向条件分支, 但是当分支较多时会使条件语句的嵌套层次太多, 程序冗长, 可读性降低。开关语句直接处理多分支选择, 使程序结构清晰, 使用方便。开关语句是用关键字 `switch` 构成的, 它的一般形式如下:

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

开关语句的执行过程是: 将 `switch` 后面表达式的值与 `case` 后面各个常量表达式的值逐个进行比较, 若遇到匹配时, 就执行相应 `case` 后面的语句, 然后执行 `break` 语句, `break` 语句又称间断语句, 它的功能是中止当前语句的执行, 使程序跳出 `switch` 语句。若无匹配的情况, 则只执行语句 `d`。开关语句的执行过程如图 2.3 所示。

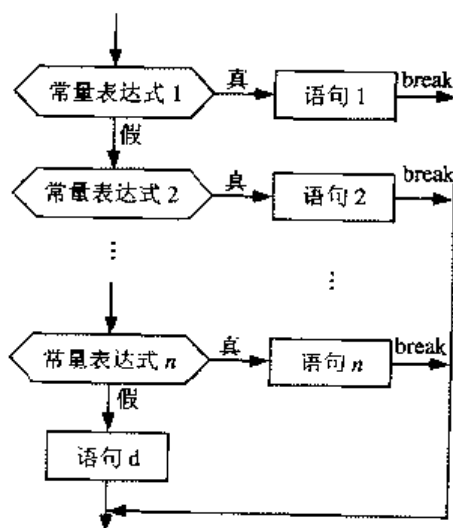


图 2.3 开关语句的执行过程

例 2.14: 开关语句的使用。

本程序按照输入的年份 `year` 和月份 `month`, 计算该月有多少天。程序需要判断该年是否为闰年。闰年的 2 月有 29 天, 平年 2 月只有 28 天。闰年的条件是: 年份数 `year` 能被 4 整除, 但不能被 100 整除; 或者年份数 `year` 能被 400 整除。这个条件可以用一个逻辑关系式来表达:

$$year \% 4 == 0 \ \&\& \ year \% 100 != 0 \ || \ year \% 400 == 0$$

当这个表达式的值为真 (非 0 值) 时, `year` 为闰年, 否则为平年。

```
#include <stdio.h>
main() {
    int year, month, len;
    while(1){
        printf("Enter year & month: \n");
        scanf("%d%d", &year, &month);
        switch(month) {
            case 1: len=31; break;
            case 3: len=31; break;
            case 5: len=31; break;
            case 7: len=31; break;
            case 8: len=31; break;
            case 10: len=31; break;
            case 12: len=31; break;
            case 4: len=30; break;
            case 6: len=30; break;
            case 9: len=30; break;
            case 11: len=30; break;
            case 2: if(year%4==0&&year%100 != 0||year%400==0) len=29;
                    else len=28;
                    break;
            default: printf("Input error \n");
                    len=0;
                    break;
        }
        if(len != 0)
            printf("The lenth of %d, %dis %d \n", year, month, len);
    }
}
```

程序执行结果:

Enter year & month:

1996 2 回车

The lenth of 1996, 2 is 29

2.3.5 循环语句

实际应用中很多地方需要用到循环控制,如对于某种操作需要反复进行多次等,这时可以用循环语句来实现。在C语言程序中用来构成循环控制的语句有:while语句、do-while语句、for语句以及goto语句,分述如下。

① 采用while语句构成循环结构的一般形式如下:

while (条件表达式) 语句;

其意义为,当条件表达式的结果为真(非0值)时,程序就重复执行后面的语句,一直执行到条件表达式的结果变为假(0值)时为止。这种循环结构是先检查条件表达式所给出的条件,再根据检查的结果决定是否执行后面的语句。如果条件表达式的结果一开始就为假,则后面的语句一次也不会被执行。这里的语句可以是复合语句。图2.4所示为while语句的执行过程。

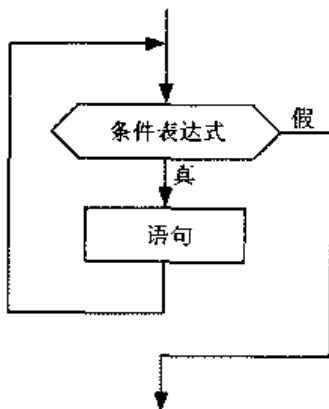


图 2.4 while 语句的执行过程

例 2.15: 使用 while 语句计算自然数 1~100 的累加和。

```
#include<stdio.h>
main() {
    int i,s=0;
    i=1;
    while (i<=100)
    {
        s=s+i;
        i++;
    }
    printf("1+2+ ... +100 = %d\n",s);
    while(1);
}
```

程序执行结果:

1+2+ ... +100 = 5050

② 采用 do-while 语句构成循环结构的一般形式如下:

do 语句 while (条件表达式);

这种循环结构的特点是先执行给定的循环体语句,然后再检查条件表达式的结果。当条件表达式的值为真(非0值)时,则重复执行循环体语句,直到条件表达式的值变为假(0值)时为止。因此,用 do-while 语句构成的循环结构在任何条件下,循环体语句至少会被执行一次。图 2.5 给出了这种循环结构的流程图。

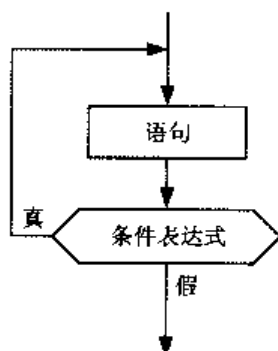


图 2.5 do-while 循环结构的流程图

例 2.16: 用 do-while 语句构成的循环计算自然数 1~100 的累加和。

```

#include<stdio.h>
main() {
    int i,s=0;
    i=1;
    do
    {
        /* 复合语句循环体 */
        s=s+i;
        i++;
    }
    /* 循环体结束 */
    while(i<=100);
    printf("1+2+ ... +100 = %d\n",s);
    while(1);
}
  
```

程序执行结果:

1+2+...+100 = 5050

例 2.16 的程序与例 2.15 的程序十分相似。它们的区别仅仅是执行循环体语句和判断条件表达式的结果的顺序不同。另外,用 do-while 语句构成的循环结构中,while(条件表达式)的后面必须有一个分号,而用 while 语句构成的循环结构中 while(条件表达式)后面是没有分号的。这一点在写程序时一定要注意。

③ 采用 for 语句构成循环结构的一般形式如下:

for ([初值设定表达式]; [循环条件表达式]; [更新表达式]) 语句

for 语句的执行过程是:先计算出初值设定表达式的值作为循环控制变量的初值,再

检查循环条件表达式的结果，当满足条件时就执行循环体语句并计算更新表达式，然后再根据更新表达式的计算结果来判断循环条件是否满足，一直进行到循环条件表达式的结果为假（0 值）时退出循环体。for 语句的执行过程如图 2.6 所示。

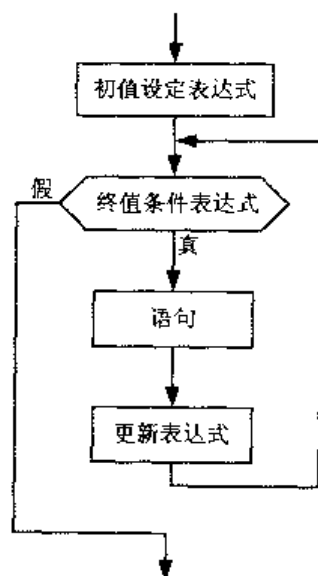


图 2.6 for 语句的执行过程

例 2.17: 用 for 语句构成的循环计算自然数 1~100 的累加和。

```
#include<stdio.h>
main() {
    int i,s=0;
    for (i=1; i<=100; i++)
        s=s+i;          /* 循环体语句 */
    printf("1+2+ ... +100 = %d\n",s);
    while(1);
}
```

程序执行结果:

1+2+ ... +100 = 5050

在 C 语言程序的循环结构中，for 语句的使用最为灵活，它不仅可用于循环次数已经确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况。另外，for 语句中的三个表达式是相互独立的，并不一定要求三个表达式之间有依赖关系。并且 for 语句中的三个表达式都可能缺省，但无论缺省哪一个表达式，其中的两个分号都不能缺省。一般不要缺省循环条件表达式，以免形成死循环。

例 2.18: for 语句中缺省表达式的例子—计算自然数 1~100 的累加和。

```
#include<stdio.h>
main() {
    int i,s=0;
    i=1;          /* 设置循环初值 */
    for ( ; ; ) {
        s=s+i;
        i++;
    }
    printf("1+2+ ... +100 = %d\n",s);
}
```

```

    for ( ; i<=100 ; ) {      /* 缺省初值设定表达式和更新表达式 */
        s=s+i;                /* 循环体语句 */
        i++;                  /* 循环控制变量更新 */
    }
    printf("1+2+ ... +100 = %d\n",s);
    while(1);
}

```

程序执行结果:

1+2+ ... +100 = 5050

④ goto 语句是一个无条件转向语句，它的一般形式为：

goto 语句标号;

其中语句标号是一个带冒号“:”的标识符。将 goto 语句和 if 语句一起使用，可以构成一个循环结构。但更常见的是在 C 语言程序中采用 goto 语句来跳出多重循环，需要注意的是只能用 goto 语句从内层循环跳到外层循环，而不允许从外层循环跳到内层循环。

例 2.19: 使用 goto 语句跳出循环结构。

本程序采用循环结构来求一整数的等差数列，该数列满足条件：头四个数的和值为 26，积值为 880。该数列的公差应为正整数，否则将产生负的项，此外该数列的首项数必须小于 5，且其公差也应小于 5，否则头四项的和值将大于 26。

```

#include<stdio.h>
main() {
    int a,b,c,d,i;
    for(a=1; a<5; ++a) {
        for(d=1; d<5; ++d) {
            b=a+(a*d)+(a*2*d)+(a*3*d);
            c=a*(a*d)*(a*2*d)*(a*3*d);
            if(b==26 && c==880)
                goto pt;
        }
    }
    pt: for(i=0; i<=10; ++i)
        printf("%d,",a+i*d);
    printf("...\n");
    while(1);
}

```

程序执行结果:

2,5,8,11,14,17,20,23,26,29, ...

在这个程序中采用 for 语句构成了两重循环嵌套，即在第一个 for 语句的循环体中又出现了另一个 for 语句的循环体，需要时还可以构成多重循环结构。程序在最内层循环体中采用了一个 goto 语句，它的作用是直接从跳出两层循环，即跳到第一层循环体外边由标号 pt:所指出的地方。前面在介绍开关语句时提到采用 break 语句可以跳出开关语句，break 语

句还可以用于跳出循环语句。对于上面的例子，也可以采用 **break** 语句来终止循环。

例 2.20: 用 break 语句终止循环。

```
#include<stdio.h>
main() {
    int a,b,c,d,i;
    for(a=1; a<5; ++a) {
        for(d=1; d<5; ++d) {
            b=a+(a*d)+(a*2*d)+(a*3*d);
            c=a*(a*d)*(a*2*d)*(a*3*d);
            if(b==26 && c==880)
                break;
        }
        if(b==26 && c==880)
            break;
    }
    for(i=0; i<=10; ++i)
        printf("%d,",a+i*d);
    printf("...\n");
    while(1);
}
```

程序执行结果:

2,5,8,11,14,17,20,23,26,29, ...

从例 2.20 可以看到，对于多重循环的情况，**break** 语句只能跳出它所处的那一层循环，而不像 **goto** 语句可以直接从最内层循环中跳出来。由此可见，要退出多重循环时，采用 **goto** 语句比较方便。需要指出的是，**break** 语句只能用于开关语句和循环语句之中，它是一种具有特殊功能的无条件转移语句。另外还要注意，在进行实际程序设计时，为了保证程序具有良好的结构，应当尽可能少地采用 **goto** 语句，以使程序结构清晰易读。

在循环结构中还可以使用一种中断语句 **continue**，它的功能是结束本次循环，即跳过循环体中下面尚未执行的语句，把程序流程转移到当前循环语句的下一个循环周期，并根据循环控制条件决定是否重复执行该循环体。**continue** 语句的一般形式为：

continue;

continue 语句通常和条件语句一起用在由 **while**、**do-while** 和 **for** 语句构成的循环结构中，它也是一种具有特殊功能的无条件转移语句，但与 **break** 语句不同，**continue** 语句并不跳出循环体，而只是根据循环控制条件确定是否继续执行循环语句。

例 2.21: 利用 continue 语句把 10~20 之间不能被 3 整除的数输出。

```
#include<stdio.h>
main() {
    int n;
    for(n=10; n<=20; n++) {
        if(n%3==0)
```

```

        continue;
    printf("%d ",n);
}
while(1);
}

```

程序执行结果:

10 11 13 14 16 17 19 20

2.3.6 返回语句

返回语句用于终止函数的执行,并控制程序返回到调用该函数时所处的位置。返回语句有两种形式:

return (表达式); 或者 **return;**

如果 **return** 语句后边带有表达式,则要计算表达式的值,并将表达式的值作为该函数的返回值。若使用不带表达式的第 2 种形式,则被调用函数返回主调用函数时,函数值不确定。一个函数的内部可以含有多个 **return** 语句,但程序仅执行其中的一个 **return** 语句而返回主调用函数。一个函数的内部也可以没有 **return** 语句,在这种情况下,当程序执行到最后一个界限符“**}**”处时,就自动返回主调用函数。

例 2.22: return 语句的使用。

```

#include <stdio.h>
main() {
    /* 主函数体 */
    int x,n,p,power(int x,int n);
    printf("calculate X to the power of N\n please input X and N ? ");
    scanf("%d%d",&x,&n);
    p = power(x,n);
    /* 在此处调用函数 */
    printf("%d to the power of %d = %d\n",x,n,p);
    while(1);
}

int power(int x,int n)
/* 被调用函数 */
int x,n;
{
    int i,p = 1;
    for( i=0; i<n; ++i ) p *= x;
    return( p );
    /* 带值返回到调用处 */
}

```

程序执行结果:

```

calculate X to the power of N
please input X and N ? 5 3 回车
5 to the power of 3 = 125

```

第3章 函 数

函数是C语言中的一种基本模块,实际上,一个C语言程序就是由若干个模块化的函数所构成的。前面我们已经看到,C语言程序总是由主函数 `main()` 开始, `main()` 函数是一个控制程序流的特殊函数,它是程序的起点。在进行程序设计的过程中,如果所设计的程序较大,一般应将其分成若干个子程序模块,每个模块完成一种特定的功能。在C语言中,子程序是用函数来实现的。对于一些需要经常使用的子程序可以设计成一个专门的函数库,以供反复调用。此外,Keil Cx51 编译器还提供了丰富的运行库函数,用户可以根据需要随时调用。这种模块化的程序设计方法,可以大大提高编程效率和速度。

3.1 函数的定义

从用户的角度来看,有两种函数:标准库函数和用户自定义函数。标准库函数是 Keil Cx51 编译器提供的,不需要用户进行定义,可以直接调用。用户自定义函数是用户根据自己需要编写的能实现特定功能的函数,它必须先进行定义之后才能调用。函数定义的一般形式为:

函数类型 函数名 (形式参数表)

形式参数说明

```
{  
    局部变量定义  
    函数体语句  
}
```

其中,“函数类型”说明了自定义函数返回值的类型。

“函数名”是用标识符表示的自定义函数名字。

“形式参数表”中列出的是在主调用函数与被调用函数之间传递数据的形式参数,形式参数的类型必须加以说明。ANSI C 标准允许在形式参数表中对形式参数的类型进行说明。如果定义的是无参函数,可以没有形式参数表,但圆括号不能省略。

“局部变量定义”是对在函数内部使用的局部变量进行定义。

“函数体语句”是为完成该函数的特定功能而设置的各种语句。

如果定义函数时只给出一对花括号 `{}` 而不给出其局部变量和函数体语句,则该函数为“空函数”,这种空函数也是合法的。在进行C语言模块化程序设计时,各模块的功能可通过函数来实现。开始时只设计最基本的模块,其他作为扩充功能在以后需要时再加上。编写程序时可在将来准备扩充的地方写上一个空函数,这样可使程序的结构清晰,可读性好,

而且易于扩充。

例 3.1: 定义一个计算整数的正整数次幂的函数。

```
int power(x, n)
{
    int x, n;
    {
        int i, p;
        p=1;
        for(i=1; i<=n; ++i)
            p=p*x;
        return(p);
    }
}
```

这里定义了一个返回值为整型值的函数 `power()`，它有两个形式参数：`x, n`。形式参数的作用是接受从主调用函数传递过来的实际参数的值。上例中形式参数 `x` 和 `n` 被说明为 `int` 类型。花括号以内的部分是自定义函数的函数体。上例中在函数体内定义了两个局部变量 `i` 和 `p`，它们均为整型数据。需要注意的是，形式参数的说明与函数体内的局部变量定义是完全不同的两个部分，前者应写在花括号的外面，而后者是函数体的一个组成部分，必须要写在花括号的里面。为了不发生混淆，ANSI C 标准允许在形式参数表中对形式参数的类型进行说明，如上例可写成：`int power(int x, int n)`。

在函数体中可以根据用户自己的需要，设置各种不同的语句。这些语句应能完成所需要的功能。上例在函数体中用一个 `for` 循环结构完成一个整数的正整数次幂的计算，计算结果赋值给变量 `p`。函数体中最后一条语句 `return(p)` 的作用是将 `p` 的值返回到主调用函数中去。`return` 语句后面圆括号中的值称为函数的返回值，圆括号可以省略，即 `return p` 和 `return(p)` 是等价的。由于 `p` 是函数的返回值，因此在函数体中进行变量定义时，应将变量 `p` 的类型定义得与函数本身的类型相一致。如果二者类型不一致，则函数调用时的返回值可能发生错误。如果函数体中没有 `return` 语句，则该函数由函数体最后面的右闭花括号“`}`”返回。在这种情况下，函数的返回值是不确定的。对于不需要有返回值的函数，可以将该函数定义为 `void` 类型（空类型）。对于上例，如果定义为：`void power(int x, int n)`，则可将函数体中的 `return` 语句去掉，这样，编译器会保证在函数调用结束时不使函数返回任何值。为了使程序减少出错，保证函数的正确调用，凡是不要求有返回值的函数，都应将其定义成 `void` 类型。

例 3.2: 不同函数的定义方法。

```
char fun1(x, y)          /* 定义一个 char 型函数 */
{
    int x;                /* 说明形式参数的类型 */
    char y;
    {
        char z;           /* 定义函数内部的局部变量 */
        z=x+y;            /* 函数体语句 */
        return(z);        /* 返回函数的值 z，注意变量 z 与函数本身
                             的类型均为 char 型 */
    }
}
```

```
int fun2(float a, float b) /* 定义一个 int 型函数，在形式参数表中说明形式参数的类型 */
{
    int x;                /* 定义函数内部的局部变量 */
    x=a-b;                /* 函数体语句 */
    return(x);            /* 返回函数的值 x，注意变量 x 与函数本身的类型均为 int 型 */
}

long fun3()               /* 定义一个 long 型函数，它没有形式参数 */
{
    long x;               /* 定义函数内部的局部变量 */
    int i, j;
    x=i*j;                /* 函数体语句 */
    return(x);            /* 返回函数的值 x，注意变量 x 与函数本身的类型均为 long 型 */
}

void fun4(char a, char b) /* 定义一个无返回值的 void 型函数 */
{
    char x;               /* 局部变量定义 */
    x=a+b;                /* 函数体语句 */
}                          /* 函数不需要返回值，省略 return 语句 */

void fun5( )              /* 定义一个空函数 */
{
}
```

3.2 函数的调用

3.2.1 函数的调用形式

C 语言程序中函数是可以互相调用的。所谓函数调用就是在一个函数体中引用另外一个已经定义了的函数，前者称为主调用函数，后者称为被调用函数。函数调用的一般形式为：

函数名（实际参数表）

其中，“函数名”指出被调用的函数。

“实际参数表”中可以包含多个实际参数，各个参数之间用逗号隔开。实际参数的作用是将它的值传递给被调用函数中的形式参数。需要注意的是，函数调用中的实际参数与函数定义中的形式参数必须在个数、类型及顺序上严格保持一致，以便将实际参数的值正确地传递给形式参数。否则在函数调用时会产生意想不到的结果。如果调用的是无参函数，则可以没有实际参数表，但圆括号不能省略。

在 C 语言中可以采用三种方式完成函数的调用。

(1) 函数语句

在主调函数中将函数调用作为一条语句，例如：

```
fun1();
```

这是无参调用，它不要求被调用函数返回一个确定的值，只要求它完成一定的操作。

(2) 函数表达式

在主调函数中将函数调用作为一个运算对象直接出现在表达式中，这种表达式称为函数表达式。例如：

```
c = power(x,n) + power(y,m);
```

这其实是一个赋值语句，它包括两个函数调用，每个函数调用都有一个返回值，将两个返回值相加的结果，赋值给变量 c。因此这种函数调用方式要求被调函数返回一个确定的值。

(3) 函数参数

在主调函数中将函数调用作为另一个函数调用的实际参数。例如：

```
y=power(power(i, j), k);
```

其中，函数调用 power(i, j) 放在另一个函数调用 power(power(i, j), k) 的实际参数表中，以其返回值作为另一个函数调用的实际参数。这种在调用一个函数的过程中又调用了另外一个函数的方式，称为嵌套函数调用。在输出一个函数的值时经常采用这种方法，例如：

```
printf("%d", power(i,j));
```

其中，函数调用 power(i,j) 是作为 printf() 函数的一个实际参数处理的，它也属于嵌套函数调用方式。

3.2.2 对被调用函数的说明

与使用变量一样，在调用一个函数之前（包括标准库函数），必须对该函数的类型进行说明，即“先说明，后调用”。如果调用的是库函数，一般应在程序的开始处用预处理命令 `#include` 将有关函数说明的头文件包含进来。例如前面例子中经常出现的预处理命令 `#include<stdio.h>`，就是将库输出函数 `printf()` 有关的头文件 `stdio.h` 包含到程序文件中来。头文件“`stdio.h`”中有关于库输入输出函数的一些说明信息，如果不使用这个包含命令，库输入输出函数就无法被正确地调用。

如果调用的是用户自定义函数，而且该函数与调用它的主调函数在同一个文件中，一般应该在主调函数中对被调用函数的类型进行说明。函数说明的一般形式为：

类型标识符 被调用的函数名（形式参数表）；

其中，“类型标识符”说明了函数返回值的类型。

“形式参数表”中说明各个形式参数的类型。

需要注意的是，函数的说明与函数的定义是完全不同的。函数的定义是对函数功能的确立，它是一个完整的函数单位。而函数的说明，只是说明了函数返回值的类型。二者在

书写形式上也不一样，函数说明结束时在圆括号的后面需要有一个分号“;”作为结束标志，而在函数定义时，被定义函数名的圆括号后面没有分号“;”，即函数定义还未结束，后面应接着书写形式参数说明和被定义的函数体部分。

如果被调函数是在主调函数前面定义的，或者已经在程序文件的开始处说明了所有被调函数的类型，在这两种情况下可以不必再在主调函数中对被调函数进行说明。也可以将所有用户自定义函数的说明另存为一个专门的头文件，需要用#include 将其包含到主程序中去。

C 语言程序中不允许在一个函数定义的内部包括另一个函数的定义，即不允许嵌套函数定义。但是允许在调用一个函数的过程中包含另一个函数调用，即嵌套函数调用在 C 语言程序中是允许的。

例 3.3: 函数调用的例子。

```
#include <stdio.h>
int Max(int x, int y);          /* 对被调用函数进行说明 */
void main() {                  /* 主函数 */
    int a, b;                  /* 主函数的局部变量定义 */
    printf("input a and b: \n");
    scanf("%d %d", &a, &b);    /* 调用库输入函数，从键盘获得 a、b 的值 */
    printf("Max=%d", Max(a, b)); /* 调用库输出函数，输出 a、b 中较大者的值 */
    while(1);
}

int Max(int x, int y) {        /* 功能函数定义 */
    int z;                    /* 局部变量定义 */
    if(x>y)                   /* 函数体语句 */
        z=x;
    else
        z=y;
    return(z);
}
```

程序执行结果:

```
input a and b:
123 456 回车
Max=456
```

在这个例子中，主函数 main()先调用库输入函数 scanf(), 从键盘输入两个值分别赋值给局部变量 a 和 b, 然后调用库输出函数 printf()将 a、b 中较大者输出。在调用库输出函数 printf()的过程中又调用了自定义功能函数 Max(), 将键盘输入的 a、b 的值作为实际参数传递给 Max()函数中的形式参数 x、y。在 Max()函数中对实际输入值进行比较以获得较大者的值。这也是一个嵌套函数调用的例子，图 3.1 是例 3.3 程序中函数调用的执行过程。

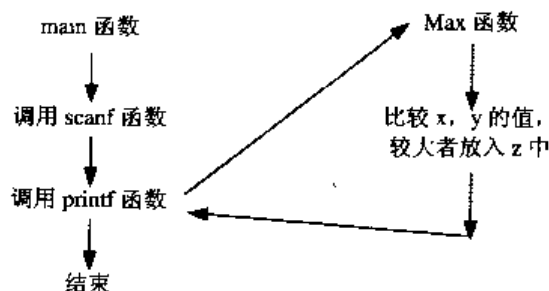


图 3.1 函数的嵌套调用过程

3.2.3 函数的参数和函数的返回值

通常在进行函数调用时，主调用函数与被调用函数之间具有数据传递关系。这种数据传递是通过函数的参数实现的。在定义一个函数时，位于函数名后面圆括号中的变量名称为“形式参数”，而在调用函数时，函数名后面括号中的表达式称为“实际参数”。形式参数在未发生函数调用之前，不占用内存单元，因而也是没有值的。只有在发生函数调用时它才被分配内存单元，同时获得从主调用函数中实际参数传递过来的值。函数调用结束后，它所占用的内存单元也被释放。

实际参数可以是常数，也可以是变量或表达式，但要求它们具有确定的值。进行函数调用时，主调用函数将实际参数的值传递给被调用函数中的形式参数。为了完成正确的参数传递，实际参数的类型必须与形式参数的类型一致，如果两者不一致，则会发生“类型不匹配”错误。

例 3.4：计算一个整数的正整数次幂。

```

#include<stdio.h>
main() {
    int power(int x, int n);
    int a, b, c;
    printf("please input X and n: \n");
    scanf("%d %d",&a, &b);
    c=power(a, b);
    printf("\n%d to the power of %d is: %d", a, b, c);
    while(1);
}

int power(int x, int n) {
    int i, p;
    p=1;
    for(i=1; i<=n; ++i)
        p=p*x;
    return(p);
}
  
```

程序执行结果:

please input X and n:

5, 3 回车

5 to the power of 3 is 125

在这个程序中定义了一个计算整数的正整数次幂的函数 `int power(int x, int n)`; 它有两个整型的形式参数 `x` 和 `n`。在程序开始时, 变量 `x` 和 `n` 是不占用内存单元的, 因此也是没有值的。在主函数 `main()` 中先从键盘输入两个整数值 `a` 和 `b`, 然后通过函数调用语句 `c=power(a, b)`; 将实际参数 `a` 和 `b` 的值传递给被调用函数 `power()` 中的形式参数。调用发生时, 形式参数变量 `x` 和 `n` 被赋以实际参数 `a` 和 `b` 的值, 从而使函数 `power()` 能按实际参数的值进行计算。从这个例子可以看到, 形式参数和实际参数可以不同名, 但它们的类型必须要一致。

一般情况下, 希望通过函数调用使主调用函数获得一个确定的值, 这就是函数的返回值。例如, 上例中的函数调用语句 `c=power(a, b)`; 就是将函数 `power()` 的返回值赋给变量 `c`。函数的返回值是通过 `return` 语句获得的, 如果希望从被调用函数中带回一个值到主调用函数, 被调用函数中必须包含有 `return` 语句。一个函数中可以有一个以上的 `return` 语句, 执行到哪一个 `return` 语句, 哪一个 `return` 语句起作用。`return` 后面可以跟一个表达式, 例如, `return(x>y? x: y)`; 这种写法只用一条 `return` 语句即可同时完成表达式的计算和函数值的返回。`return` 后面还可以跟另外一个已定义了的函数名, 例如: `return keyval(rdkey)`; 采用这种写法可实现函数的嵌套调用, 即在函数返回的同时调用另一个函数。

函数返回值的类型确定了该函数的类型, 因此在定义一个函数时, 函数本身的类型应与 `return` 语句中变量或表达式的类型一致。例如, 上例中 `power()` 函数被定义为 `int` 类型, `return` 语句中的变量 `p` 也被定义为 `int` 类型。如果函数类型与 `return` 语句中表达式的值类型不一致, 则以函数的类型为准。对于数值数据可以自动进行类型转换, 即函数的类型决定返回值的类型。如果不需要被调用函数返回一个确定的值, 则可以不要 `return` 语句, 同时应将被调用函数定义成 `void` 类型。事实上, `main()` 函数就是一个典型的没有返回值的函数, 因此可以将其写成 `void main()` 的形式。由于 `void` 类型的函数没有 `return` 语句, 因此在一个 `void` 类型函数的调用结束时, 将从该函数的最后一个花括号处返回到主调用函数。

例 3.5: 使用 `void` 类型函数的例子。

```
#include<stdio.h>

void main() {                                /* void 类型的主函数 */
    void prn_char(char x); /* 功能函数说明 */
    prn_char('w');         /* 功能函数调用 */
    while(1);
}

void prn_char(char x) /* 将功能函数定义为 void 类型, 无返回值 */
{
    printf("%c has ASCII value %d\n", x, x);
}
/* void 型功能函数从此处返回主调用函数 */
```

程序执行结果:

w has ASCII value 119

3.2.4 实际参数的传递方式

在进行函数调用时,必须用主调函数中的实际参数来替换被调函数中的形式参数,这就是所谓的参数传递。在C语言中,对于不同类型的实际参数,有三种不同的参数传递方式。

(1) 基本类型的实际参数传递

当函数的参数是基本类型的变量时,主调函数将实际参数的值传递给被调函数中的形式参数,这种方式称为值传递。前面讲过,函数中的形式参数在未发生函数调用之前是不占用内存单元的,只有在进行函数调用时才为其分配临时存储单元。而函数的实际参数是要占用确定的存储单元的。值传递方式是将实际参数的值传递到为被调函数中形式参数分配的临时存储单元中,函数调用结束后,临时存储单元被释放,形式参数的值也就不复存在,但实际参数所占用的存储单元保持原来的值不变。这种参数传递方式在执行被调函数时,如果形式参数的值发生变化,可以不必担心主调函数中实际参数的值会受到影响。因此值传递是一种单向传递。

(2) 数组类型的实际参数传递

当函数的参数是数组类型的变量时,主调函数将实际参数数组的起始地址传递到被调函数中形式参数的临时存储单元,这种方式称为地址传递。地址传递方式在执行被调函数时,形式参数通过实际参数传来的地址,直接到主调函数中去存取相应的数组元素,故形式参数的变化会改变实际参数的值。因此地址传递是一种双向传递。

(3) 指针类型的实际参数传递

当函数的参数是指针类型的变量时,主调函数将实际参数的地址传递给被调函数中形式参数的临时存储单元,因此也属于地址传递。在执行被调函数时,也是直接到主调函数中去访问实际参数变量,在这种情况下,形式参数的变化会改变实际参数的值。

前面介绍的一些函数调用中所涉及的都是基本类型的实际参数传递,这种参数传递方式比较容易理解和应用。关于数组类型和指针类型实际参数的传递较为复杂,将在第4章中详细介绍。

3.3 函数的递归调用与再入函数

如果在调用一个函数的过程中又间接或直接地调用该函数本身,称为函数的递归调用。例如,计算阶乘函数 $f(n)=n!$,可以先计算 $f(n-1)=(n-1)!$,而计算 $f(n-1)$ 时又可以先计算 $f(n-2)=(n-2)!$,这就是递归算法。再入函数是一种可以在函数体内不直接或间接调用其自身的一种函数,显然再入函数是可以进行递归调用的。

Keil Cx51 编译器采用一个扩展关键字 `reentrant`,作为定义函数时的选项,需要将一个函数定义为再入函数时,只要在函数名后面加上关键字 `reentrant` 即可:

函数类型 函数名 (形式参数表) [reentrant]

再入函数可被递归调用, 无论何时, 包括中断服务函数在内的任何函数都可调用再入函数。与非再入函数的参数传递和局部变量的存储分配方法不同, Cx51 编译器为再入函数生成一个模拟栈, 通过这个模拟栈来完成参数传递和存放局部变量。模拟栈所在的存储器空间根据再入函数存储器模式的不同, 可以是 DATA、PDATA 或 XDATA 存储器空间。当程序中包含有多种存储器模式的再入函数时, Cx51 编译器为每种模式单独建立一个模拟栈并独立管理各自的栈指针。对于再入函数有如下规定。

① 再入函数不能传送 bit 类型的参数, 也不能定义一个局部位变量, 再入函数不能包括位操作以及 8051 系列单片机的可位寻址区。

② 与 PL/M51 兼容的函数不能具有 reentrant 属性, 也不能调用再入函数。

③ 在编译时存储器模式的基础上为再入函数在内部或外部存储器中建立一个模拟堆栈区, 称为再入栈。在 small 模式下再入栈位于 idata 区, 在 compact 模式下再入栈位于 pdata 区, 在 large 模式下再入栈位于 xdata 区。再入函数的局部变量及参数都被放在再入栈中, 从而使再入函数可以进行递归调用。而非再入函数的局部变量被放在再入栈之外的暂存区内, 如果对非再入函数进行递归调用, 则上次调用时使用的局部变量数据将被覆盖。

④ 在同一个程序中可以定义和使用不同存储器模式的再入函数, 任意模式的再入函数不能调用不同模式的再入函数, 但可任意调用非再入函数。

⑤ 在参数的传递上, 实际参数可以传递给间接调用的再入函数。无再入属性的间接调用函数不能包含调用参数, 但是可以使用定义的全局变量来进行参数传递。

例 3.6: 利用函数的递归调用计算整数的阶乘。

```
#include<stdio.h>
fac(int n) reentrant {
    if (n<1) return(1);
    else return(n*fac(n-1));
}

main() {
    int n;
    printf("please input a number: \n");
    scanf("%d", &n);
    printf("fac(%d)=%d\n", n, fac(n));
    while(1);
}
```

程序执行结果:

```
please input a number
3 回车
fac(3)=6
```

在这个程序中定义了一个再入函数 $fac(n)$, 它是用来计算阶乘 $n!$ 的函数。在 $fac()$ 的函数体中又调用了 $fac()$ 函数本身, 因此这是一种函数的递归调用。再入函数在进行递归调用

时,新的局部变量和参数在再入栈中重新分配存储单元,并以新的变量重新开始执行。每次递归调用返回时,前面压入的局部变量和参数会从再入栈中弹出,并恢复到上次调用自身的地址继续执行。如果是非再入函数进行递归调用,每次调用函数自身时,上次调用时使用的局部变量数据将被覆盖,因而在递归调用结束时不能得到正确的结果。对于例 3.6 的程序,如果将函数 $fac(n)$ 定义成非再入函数,则程序的运行结果为 0,显然这是不正确的。

采用函数的递归调用可使程序的结构紧凑,但是递归调用要求采用再入函数,以便利用再入栈来保存有关的局部变量数据,从而要占据较大的内存空间。另外递归调用时对函数的处理速度也比较慢,因此一般情况下应尽量避免采用函数递归调用,定义函数时应尽量避免使用再入属性。

3.4 中断服务函数与寄存器组定义

Keil Cx51 编译器支持在 C 语言源程序中直接编写 8051 单片机的中断服务函数程序,从而减轻了采用汇编语言编写中断服务程序的烦琐程度。为了在 C 语言源程序中直接编写中断服务函数的需要,Keil Cx51 编译器对函数的定义进行了扩展,增加了一个扩展关键字 `interrupt`,它是函数定义时的一个选项,加上这个选项即可以将一个函数定义成中断服务函数。定义中断服务函数的一般形式为:

函数类型 函数名(形式参数表) [`interrupt n`] [`using n`]

关键字 `interrupt` 后面的 n 是中断号, n 的取值范围为 $0 \sim 31$ 。编译器从 $8n+3$ 处产生中断向量,具体的中断号 n 和中断向量取决于 8051 系列单片机芯片型号,常用中断源和中断向量如表 3-1 所示:

表 3-1 常用中断号与中断向量

中断号 n	中 断 源	中断向量 $8n+3$
0	外部中断 0	0003H
1	定时器 0	000BH
2	外部中断 1	0013H
3	定时器 1	001BH
4	串行口	0023H

8051 系列单片机可以在片内 RAM 中使用 4 个不同的工作寄存器组,每个寄存器组中包含 8 个工作寄存器 ($R0 \sim R7$)。Keil Cx51 编译器扩展了一个关键字 `using`,专门用来选择 8051 单片机中不同的工作寄存器组。`using` 后面的 n 是一个 $0 \sim 3$ 的常整数,分别选中 4 个不同的工作寄存器组。在定义一个函数时 `using` 是一个选项,如果不用该选项,则由编译器自动选择一个寄存器组作绝对寄存器组访问。需要注意的是,关键字 `using` 和 `interrupt` 的后面都不允许跟带运算符的表达式。

关键字 `using` 对函数目标代码的影响如下:在函数的入口处将当前工作寄存器组保护到堆栈中;指定的工作寄存器内容不会改变;函数退出之前将被保护的工作寄存器组从堆

栈中恢复。

使用关键字 `using` 在函数中确定一个工作寄存器组时必须十分小心，要保证任何寄存器组的切换都只在仔细控制的区域内发生，如果不做到这一点将产生不正确的函数结果。另外还要注意，带 `using` 属性的函数原则上不能返回 `bit` 类型的值。并且关键字 `using` 不允许用于外部函数。

关键字 `interrupt` 也不允许用于外部函数，它对中断函数目标代码的影响如下：在进入中断函数时，特殊功能寄存器 `ACC`、`B`、`DPH`、`DPL`、`PSW` 将被保存入栈；如果不使用关键字 `using` 进行工作寄存器组切换，则将中断函数中所用到的全部工作寄存器都入栈保存；函数退出之前所有的寄存器内容出栈恢复；中断函数由 8051 单片机指令 `RETI` 结束。

下面给出一个带有寄存器组切换的中断函数定义的例子，该例中还给出了 C51 编译器所生成的 8051 单片机的指令代码。

例 3.7：带有寄存器组切换的中断函数定义。

```

stmt level    source

1             #pragma  cd
2             #include <reg51.h>
3             extern void alfunc(bit b0);
4             extern bit alarm;
5             int DTIMES;
6             char bdata flag;
7             sbit flag0=flag^0;
8             int dtime1=0x0a;
9
10            void int0 () interrupt 0 using 1 {
11  1          TR1=0;
12  1          flag0=!flag0;
13  1          DTIMES=dtime1;
14  1          dtime1=0;
15  1          TR1=1;
16  1          }
17
18            void timer1 () interrupt 3 using 3 {
19  1          alfunc(alarm=1);
20  1          TH1=0x3c;
21  1          TL1=0xB0;
22  1          dtime1=dtime1+1;
23  1          if (dtime1==0)
24  1          {
25  2          P0=0;
26  2          }
27  1          }

```

28

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION int0 (BEGIN)
; SOURCE LINE # 10
; SOURCE LINE # 11
0000 C28E          CLR      TR1
; SOURCE LINE # 12
0002 B200          R        CPL      flag0
; SOURCE LINE # 13
0004 850000          R        MOV      DTIMES,dtimel
0007 850000          R        MOV      DTIMES+01H,dtimel+01H
; SOURCE LINE # 14
000A 750000          R        MOV      dtimel,#00H
000D 750000          R        MOV      dtimel+01H,#00H
; SOURCE LINE # 15
0010 D28E          SETB     TR1
; SOURCE LINE # 16
0012 32            RETI
; FUNCTION int0 (END)

; FUNCTION timer1 (BEGIN)
0000 C0E0          PUSH     ACC
0002 C0F0          PUSH     B
0004 C083          PUSH     DPH
0006 C082          PUSH     DPL
0008 C0D0          PUSH     PSW
000A 75D018          MOV      PSW,#018H
; SOURCE LINE # 18
; SOURCE LINE # 19
000D D3            SETB     C
000E 9200          E        MOV      alarm,C
0010 9200          E        MOV      ?alfunc?BIT,C
0012 120000          E        LCALL   alfunc
; SOURCE LINE # 20
0015 758D3C          MOV      TH1,#03CH
; SOURCE LINE # 21
0018 758BB0          MOV      TL1,#0B0H
; SOURCE LINE # 22
001B 0500          R        INC      dtimel+01H
001D E500          R        MOV      A,dtimel+01H
001F 7002          JNZ      ?C0004
0021 0500          R        INC      dtimel

```



```

0023      ?C0004:
                                           ; SOURCE LINE # 23
0023 4500      R      ORL      A,dtime1
0025 7002              JNZ      ?C0003
                                           ; SOURCE LINE # 24
                                           ; SOURCE LINE # 25
0027 F580              MOV      P0,A
                                           ; SOURCE LINE # 26
                                           ; SOURCE LINE # 27
0029      ?C0003:
0029 D0D0              POP      PSW
002B D082              POP      DPL
002D D083              POP      DPH
002F D0F0              POP      B
0031 D0E0              POP      ACC
0033 32              RETI
                        ; FUNCTION timer1 (END)

```

编写 8051 单片机中断函数时应遵循以下规则。

- ① 中断函数不能进行参数传递,如果中断函数中包含任何参数声明都将导致编译出错。
- ② 中断函数没有返回值,如果企图定义一个返回值将得到不正确的结果。因此建议在定义中断函数时将其定义为 void 类型,以明确说明没有返回值。
- ③ 在任何情况下都不能直接调用中断函数,否则会产生编译错误。因为中断函数的退出是由 8051 单片机指令 RETI 完成的,RETI 指令影响 8051 单片机的硬件中断系统。如果在没有实际中断请求的情况下直接调用中断函数,RETI 指令的操作结果会产生一个致命的错误。

④ 如果在中断函数中调用了其他函数,则被调用函数所使用的寄存器组必须与中断函数相同。用户必须保证按要求使用相同的寄存器组,否则会产生不正确的结果,这一点必须引起足够的注意。如果定义中断函数时没有使用 using 选项,则由编译器自动选择一个寄存器组作绝对寄存器组访问。另外,由于中断的产生不可预测,中断函数对其他函数的调用可能形成递归调用,需要时可将被中断函数所调用的其他函数定义成再入函数。

⑤ Keil Cx51 编译器从绝对地址 $8n+3$ 处产生一个中断向量,其中 n 为中断号。该向量包含一个到中断函数入口地址的绝对跳传。在对源程序编译时,可用编译控制命令 NOINTVECTOR 抑制中断向量的产生,从而使用户有能力从独立的汇编程序模块中提供中断向量。

3.5 函数变量的存储方式

3.5.1 局部变量与全局变量

按照变量的有效作用范围可划分为局部变量和全局变量。局部变量是在一个函数内部

定义的变量，它只在定义它的那个函数范围以内有效。在此函数之外局部变量即失去意义，因而也就不能使用这些变量了。不同的函数可以使用相同的局部变量名，由于它们的作用范围不同，不会相互干扰。函数的形式参数也属于局部变量。在一个函数内部的复合语句中也可以定义局部变量，该局部变量只在该复合语句中有效。

全局变量是在函数外部定义的变量，又称为外部变量。全局变量可以为多个函数共同使用，其有效作用范围是从它定义的位置开始到整个程序文件结束。如果全局变量定义在一个程序文件的开始处，则在整个程序文件范围内都可以使用它。如果一个全局变量不是在程序文件的开始处定义的，但又希望在它的定义点之前的函数中引用该变量，这时应在引用该变量的函数中用关键字 `extern` 将其说明为“外部变量”。另外，如果在一个程序模块文件中引用另一个程序模块文件中定义的变量时，也必须用 `extern` 进行说明。外部变量说明与外部变量定义是不相同的。外部变量定义只能有一次，定义的位置在所有函数之外。而同一个程序文件中的外部变量说明可以有多次，说明的位置在需要引用该变量的函数之内。外部变量说明的作用只是声明该变量是一个已经在外部定义过了的变量而已。如果在同一个程序文件中，全局变量与局部变量同名，则在局部变量的有效作用范围之内，全局变量不起作用。换句话说，局部变量的优先级比全局变量高。在编写 C 语言程序时，不是特别必要的地方一般不要使用全局变量，而应当尽可能地使用局部变量。这是因为局部变量只在使用它时，才为其分配内存单元，而全局变量在整个程序的执行过程中都要占用内存单元。另外，如果使用全局变量过多，在各个函数执行时都有可能改变全局变量的值，使人们难以清楚地判断出在各个程序执行点处全局变量的值，这样会使降低程序的通用性和可读性。还有一点需要说明，如果程序中的全局变量在定义时赋给了初值，按 ANSI C 标准规定，在程序进入 `main()` 函数之前必须先对该全局变量进行初始化。这是由连接定位器 BL51 对目标程序连接定位时，在最后生成的目标代码中自动加入一段运行库“INIT.OBJ”来实现的。由于增加了这么一段代码，程序的长度会增加，运行速度也会受到影响。因此要限制使用全局变量。

下面通过一个例子来说明局部变量与全局变量的区别。

例 3.8: 局部变量与全局变量的区别。

```
#include<stdio.h>
int a=3, b=5;          /* 定义 a、b 为全局变量，并赋以初值 */
max(int a, int b) {    /* 形参 a、b 为局部变量 */
    int c;             /* 定义 c 为局部变量 */
    c=a>b? a:b;
    return(c);
}

main() {
    int a=8;           /* 定义 a 为局部变量 */
    printf("%d", max(a,b));
    while(1);
}
```

程序执行结果:

8

这个程序中故意使用了相同的变量名 *a* 和 *b*, 请读者仔细区别它们的作用范围。程序的第一行将 *a* 和 *b* 定义成全局变量并且赋了初值, 由于具有初值的全局变量需要先行初始化, 因此读者如果用 *dScope51* 对这个例子程序进行调试, 可以看到程序在进入 *main()* 函数之前, 除了要执行一段启动程序 *STARTUP* 的代码之外, 还需要执行一段全局变量初始化程序 *INIT* 的代码。第二行开始是定义一个求最大值函数 *max()*, 其作用是求得 *a* 和 *b* 中较大者的值。这里的 *a* 和 *b* 是 *max()* 函数的形式参数, 属于局部变量。外部变量 *a* 和 *b* 在函数 *max()* 内部不起作用, 即形式参数 *a* 和 *b* 的值不再是 5, 它们的值是通过主调函数中的实际参数传递过来的。程序的最后四行是 *main()* 函数, 在 *main()* 函数内部定义了一个局部变量 *a* 并赋值为 8, 全局变量 *a* 在这里不起作用, 而全局变量 *b* 在此范围内有效。因此 *printf()* 函数中的 *max(a, b)* 相当于 *max(8, 5)*, 故程序的最后执行结果为 8。

3.5.2 变量的存储种类

按变量的有效作用范围可以将其划分为局部变量和全局变量; 还可以按变量的存储方式为其划分存储种类。在 C 语言中变量有四种存储种类, 即自动变量 (*auto*)、外部变量 (*extern*)、静态变量 (*static*) 和寄存器变量 (*register*)。这四种存储种类与全局变量和局部变量之间的关系如图 3.2 所示。

1. 自动变量 (*auto*)

定义一个变量时, 在变量名前面加上存储种类说明符 “*auto*”, 即将该变量定义为自动变量。自动变量是 C 语言中使用最为广泛的一类变量。按照默认规则, 在函数体内部或复合语句内部定义的变量, 如果省略存储种类说明, 该变量即为自动变量。习惯上通常采用默认形式, 例如:

```
{
    char x;
    int y;
    ...
}
等价于
{
    auto char x;
    auto int y;
    ...
}
```

自动变量的作用范围在定义它的函数体或复合语句内部, 只有在定义它的函数被调用, 或是定义它的复合语句被执行时, 编译器才为其分配内存空间, 开始其生存期。当函数调用结束返回, 或复合语句执行结束时, 自动变量所占用的内存空间就被释放, 变量的

值当然也就不复存在，其生存期结束。当函数被再次调用或复合语句被再次执行，编译器又会为它们内部的自动变量重新分配内存空间，但它不会保留上次运行时的值，而必须被重新赋值。因此自动变量始终是相对于函数或复合语句的局部变量。

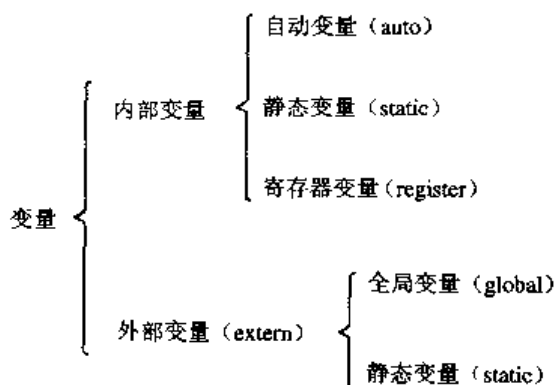


图 3.2 变量的存储种类

2. 外部变量 (extern)

使用存储种类说明符“extern”定义的变量称为外部变量。按照默认规则，凡是在所有函数之前，在函数外部定义的变量都是外部变量，定义时可以不写 extern 说明符。但是，在一个函数体内说明一个已在该函数体外或别的程序模块文件中定义过的外部变量时，则必须使用 extern 说明符。一个外部变量被定义之后，它就被分配了固定的内存空间。外部变量的生存期为程序的整个执行时间，即在程序的执行期间外部变量可被随意使用，当一条复合语句执行完毕或是从某一个函数返回时，外部变量的存储空间并不被释放，其值也仍然保留。因此外部变量属于全局变量。

C 语言允许将大型程序分解为若干个独立的程序模块文件，各个模块可分别进行编译，然后再将它们连接在一起。在这种情况下，如果某个变量需要在所有程序模块文件中使用，只要在一个程序模块文件中将该变量定义成全局变量，而在其他程序模块文件中用 extern 说明该变量是已被定义过的外部变量就可以了。

函数是可以相互调用的，因此函数都具有外部存储种类的属性。定义函数时如果冠以关键字 extern 即将其明确定义为一个外部函数。例如，extern int func2 (char a, b)。如果在定义函数时省略关键字 extern，则隐含为外部函数。如果要调用一个在本程序模块文件以外的其他模块文件所定义的函数，则必须用关键字 extern 说明被调用函数是一个外部函数。对于具有外部函数相互调用的多模块程序，利用μVision51 集成开发环境很容易完成编译连接。这个例子中有两个程序模块文件“ex1.c”和“ex2.c”，可以在μVision51 环境下将它们分别添加到一个项目文件“ex.prj”中，然后执行 Project 菜单中的 Make:Updat Project 选项即可将它们连接在一起，生成 OMF51 绝对目标文件 ex，绝对目标文件可以装入 dScope51 中进行仿真调试。

例 3.9: 多模块程序。

(程序模块 1 文件名为 ex1.c)

```
#include<stdio.h>
int x = 5;
```

```

void main() {
    extern void fun1();           /* 说明函数 fun1 在其他文件中定义 */
    extern int fun2(int y);       /* 说明函数 fun2 在其他文件中定义 */
    fun1(); fun1(); fun1();
    printf("\n%d  %d\n",x,fun2(x));
    while(1);
}
(程序模块 2 文件名为 ex2.c)
#include <stdio.h>
extern int x;                    /* 说明变量 x 在其他文件中定义 */
void fun1() {
    static int a = 5;
    int b = 5;
    printf("%d  %d  %d | ",a,b,x);
    a -= 2;
    b -= 2;
    x -= 2;
    printf("%d  %d  %d\n",a,b,x);
}

int fun2(int y) {
    return( 35 * x * y );
}

```

程序的执行结果为:

```

5  5  5 | 3  3  3
3  5  3 | 1  3  1
1  5  1 | -1 3  -1

-1  35

```

由于 C 语言不允许在一个函数体内嵌套定义另一个函数,为了能够访问不同文件中各个函数的变量,除了可以采用我们在前面介绍过的参数传递方法之外,还可以采用外部变量的方法。上面的例子就说明了这一点。需要指出的是,尽管使用外部变量在不同函数之间传递数据有时比使用函数的参数更为方便,但是当外部变量较多时,会增加程序调试排错时的困难,使程序不便于维护。另外,不通过参数传递而直接在函数中改变全局变量的值,有时还会发生一些意想不到的副作用。因此一般情况下最好还是使用函数的参数来传递数据。

3. 静态变量 (static)

使用存储种类说明符“static”定义的变量称为静态变量。在例 3.9 的模块 2 程序文件中使用了一个静态变量: static int a=5。由于这个变量是在函数 fun1()内部定义的,因此称

为内部静态变量或局部静态变量。局部静态变量不像自动变量那样只有当函数调用它时才存在，退出函数后它就消失，局部静态变量始终都是存在的，但只能在定义它的函数内部进行访问，退出函数之后，变量的值仍然保持，但不能进行访问。还有一种全局静态变量，它是在函数外部被定义的，作用范围从它的定义点开始，一直到程序结束。当一个 C 语言程序由若干个模块文件所组成时，全局静态变量始终存在，但它只能在被定义的模块文件中访问，其数据值可为该文件内的所有函数共享，退出该文件后，虽然变量的值仍然保持着，但不能被其他模块文件访问。

局部静态变量是一种在两次函数调用之间仍能保持其值的局部变量。有些程序需要在多次调用之间仍然保持变量的值，使用自动变量无法实现这一点，使用全局变量有时又会带来意外的副作用，这时就可采用局部静态变量。

例 3.10：局部静态变量的使用——计算并输出 1~5 的阶乘值。

```
#include<stdio.h>

int fac(int n) {
    static int f=1;
    f=f*n;
    return(f);
}

main() {
    int i;
    for (i=1; i<=5; i++)
        printf("%d! = %d\n", i, fac(i));
    while(1);
}
```

程序执行结果：

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

在这个程序中，一共调用了 5 次计算阶乘的函数 $fac(i)$ ，每次调用后输出一个阶乘值 $i!$ ，同时保留这个 $i!$ 值，以便下次再乘 $(i+1)$ 。由此可见，如果要保留函数上一次调用结束时的值，或是在初始化之后变量只被引用而不改变其值，则这时使用局部静态变量较为方便，以免在每次调用时都要重新进行赋值。但是，使用局部静态变量需要占用较多的内存空间，而且降低了程序的可读性，当调用次数较多时往往弄不清局部静态变量的当前值是什么。因此，建议不要多用局部静态变量。

全局静态变量是一种作用范围受限制的外部变量，它的有效作用范围从其定义点开始直至程序文件的末尾，而且只有在定义它的程序模块文件中才能对它进行访问。全局静态变量与我们在前面介绍过的单纯全局变量是有区别的。全局静态变量有一个特点，就是只

有在定义它的程序文件中才可以使用它，其他文件不能改变其内容。C语言允许进行多模块程序设计，一个较大型的程序可被分成若干个模块，分别由几个人来完成。如果各人在独立设计各自的程序模块时，有些变量可能只希望在自己的程序模块文件中使用，而不希望被别的模块文件引用，对于这种变量就可以定义为全局静态变量。需要指出的是，全局静态变量和单纯全局变量都是在编译时就已经分配了固定的内存空间的变量，只是它们的作用范围不同而已。

对于函数也可以定义成具有静态存储种类的属性。定义函数时在函数名前面冠以关键字 `static` 即将其定义为一个静态函数。例如，`static int func1(char x, int y)`。使用静态函数可使该函数只局限于其所在的模块文件。由于函数都是外部型的，因此静态外部函数定义就限制了该函数只能在定义它的模块文件中使用，其他模块文件是不能调用它的。换句话说，在其他模块文件中可以定义与静态函数完全同名的另一个函数，分别编译并连接成为一个可执行程序之后，不会由于程序中存在相同的函数名而发生函数调用时的混乱。这一特点在进行模块化程序设计时是十分有用的。

4. 寄存器变量 (register)

为了提高程序的执行效率，C语言允许将一些使用频率最高的那些变量，定义为能够直接使用硬件寄存器的所谓寄存器变量。定义一个变量时在变量名前面冠以存储种类符号“`register`”即将该变量定义成为了寄存器变量。寄存器变量可以被认为是自动变量的一种，它的有效作用范围也与自动变量相同。由于计算机中的寄存器是有限的，不能将所有变量都定义成寄存器变量。通常在程序中定义的寄存器变量时只是给编译器一个建议，该变量是否能真正成为寄存器变量，要由编译器根据实际情况来确定。另一方面，C51编译器能够识别程序中使用频率最高的变量，在可能的情况下，即使程序中并未将该变量定义为寄存器变量，编译器也会自动将其作为寄存器变量处理。下面来看一个带有汇编码的程序例子。

例 3.11：使用寄存器变量的例子——计算以整数为底的指数的幂。

stmt level source

```
1      #include<stdio.h>
2      int _power(m, e)
3      int m;
4      register int e;
5      {
6  1      register int temp;
7  1      temp=1;
8  1      for (; e; e--)
9  1      temp*=m;
10  1      return(temp);
11  1      }
12
13      main() {
```

```

14 1      int x, y;
15 1      printf("please input X Y\n");
16 1      scanf("%d %d", &x, &y);
17 1      printf("%d to the power of %d = %d",x,y,int_power(x, y));
18 1      }
19

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION _int_power (BEGIN)
; SOURCE LINE # 2
0000 8E00      R      MOV      m,R6
0002 8F00      R      MOV      m+01H,R7
;---- Variable 'e' assigned to Register 'R2/R3' ----
0004 AB05              MOV      R3,AR5
0006 AA04              MOV      R2,AR4
; SOURCE LINE # 3
; SOURCE LINE # 7
;---- Variable 'temp' assigned to Register 'R6/R7' ----
0008 7F01              MOV      R7,#01H
000A 7E00              MOV      R6,#00H
; SOURCE LINE # 8
000C          ?C0001:
000C EB              MOV      A,R3
000D 4A              ORL      A,R2
000E 600E              JZ      ?C0002
; SOURCE LINE # 9
0010 AC00      R      MOV      R4,m
0012 AD00      R      MOV      R5,m+01H
0014 120000     E      LCALL    ?C?IMUL
0017 EB              MOV      A,R3
0018 1B              DEC      R3
0019 70F1              JNZ      ?C0001
001B 1A              DEC      R2
001C          ?C0006:
001C 80EE              SJMP     ?C0001
001E          ?C0002:
; SOURCE LINE # 10
; SOURCE LINE # 11
001E          ?C0004:
001E 22              RET
; FUNCTION _int_power (END)
; FUNCTION main (BEGIN)
; SOURCE LINE # 13

```



```

; SOURCE LINE # 15
0000 7BFF      MOV     R3,#0FFH
0002 7A00      R      MOV     R2,#HIGH ?SC_0
0004 7900      R      MOV     R1,#LOW ?SC_0
0006 120000    E      LCALL  _printf

; SOURCE LINE # 16
0009 750000    E      MOV     ?_scanf?BYTE+03H,#00H
000C 750000    R      MOV     ?_scanf?BYTE+04H,#HIGH x
000F 750000    R      MOV     ?_scanf?BYTE+05H,#LOW x
0012 750000    E      MOV     ?_scanf?BYTE+06H,#00H
0015 750000    R      MOV     ?_scanf?BYTE+07H,#HIGH y
0018 750000    R      MOV     ?_scanf?BYTE+08H,#LOW y
001B 7BFF      MOV     R3,#0FFH
001D 7A00      R      MOV     R2,#HIGH ?SC_19
001F 7900      R      MOV     R1,#LOW ?SC_19
0021 120000    E      LCALL  _scanf

; SOURCE LINE # 17
0024 AD00      R      MOV     R5,y+01H
0026 AC00      R      MOV     R4,y
0028 AF00      R      MOV     R7,x+01H
002A AE00      R      MOV     R6,x
002C 120000    R      LCALL  _int_power
002F 8E00      E      MOV     ?_printf?BYTE+07H,R6
0031 8F00      E      MOV     ?_printf?BYTE+08H,R7
0033 7BFF      MOV     R3,#0FFH
0035 7A00      R      MOV     R2,#HIGH ?SC_26
0037 7900      R      MOV     R1,#LOW ?SC_26
0039 850000    E      MOV     ?_printf?BYTE+03H,x
003C 850000    E      MOV     ?_printf?BYTE+04H,x+01H
003F 850000    E      MOV     ?_printf?BYTE+05H,y
0042 850000    E      MOV     ?_printf?BYTE+06H,y+01H
0045 020000    E      LJMP   _printf

; FUNCTION main (END)

```

程序执行结果:

please input X Y

5 3 回车

5 to the power of 3 = 125

在这个程序中定义了一个计算以整数为底的指数幂的函数 `int_power()`，该函数中有两个形式参数 `int m` 和 `register int e`。它们都是 `int` 类型的变量，但参数 `e` 前面带有存储种类说明符 `register`，被特别说明为寄存器变量。另外，在该函数体中还定义了一个 `int` 类型的寄存器变量 `register int temp`。从编译得到的汇编码可以看到，形式参数 `e` 被分配给了 8051 单片机的工作寄存器 `R2` 和 `R3`，但变量 `temp` 则未分配到工作寄存器，而是作为临时工作单

元被存放到内存之中。由此可见，尽管可以在程序中定义寄存器变量，但实际上被定义的变量是否真能成为寄存器变量最终是由编译器决定的。

3.5.3 函数的参数和局部变量的存储器模式

Keil Cx51 编译器允许采用三种存储器模式：SMALL、COMPACT 和 LARGE。一个函数的存储器模式确定了函数的参数和局部变量在内存中的地址空间。处于 SMALL 模式下函数的参数和局部变量位于 8051 单片机的内部 RAM 中，处于 COMPACT 和 LARGE 模式下函数的参数和局部变量则使用 8051 单片机的外部 RAM。在定义一个函数时可以明确指定该函数的存储器模式，一般形式为：

函数类型 函数名（形式参数表）[存储器模式]

其中，“存储器模式”是 Keil Cx51 编译器扩展的一个选项。不用该选项时即没有明确指定函数的存储器模式，这时该函数按编译时的默认存储器模式处理。

例 3.12：函数的存储器模式。

```
#pragma large                      /* 默认存储器模式为 LARGE */
extern int calc(char i, int b) small; /* 指定 SMALL 模式 */
extern int func(int i, float f) large; /* 指定 LARGE 模式 */
extern void * tcp(char xdata *xp, int ndx) small; /* 指定 SMALL 模式 */

int mtest(int i, int y) small      /* 指定 SMALL 模式 */
{
    return(i*y+y*i+func(-1, 4.75));
}

int large_func(int i, int k) /* 未指定模式，按默认的 LARGE 模式处理 */
{
    return(mtest(i,k)+2);
}
```

这个例子程序的第一行用了一个预编译命令“#pragma”，它的意思是告诉 Keil Cx51 编译器在对程序进行编译时，按该预编译命令后面给出的编译控制指令“LARGE”进行编译，即本例程序编译时的默认存储器模式为 LARGE。程序中一共有五个函数：calc()、func()、*tcp()、mtest()和 large_func()，其中前面四个函数都在定义时明确指定了其存储器模式，只有最后一个函数未指定。在用 Cx51 进行编译时，只有最后一个函数按 LARGE 存储器模式处理，其余四个函数则分别按它们各自指定的存储器模式处理。这个例子说明，Keil Cx51 编译器允许采用所谓存储器的混合模式，即允许在一个程序中某个（或几个）函数使用一种存储器模式，另一个（或几个）函数使用另一种存储器模式。采用存储器混合模式编程，可以充分利用 8051 系列单片机中有限的存储器空间，同时还可加快程序的执行速度。

第 4 章 数组与指针

在第 2 章中介绍了 C 语言的基本数据类型，如整型、字符型、浮点型等，除此之外 C 语言还提供一种构造类型的数据。构造类型数据是由基本类型数据按一定规则组合而成的，又称为导出类型数据。C 语言中的构造类型数据有数组类型、结构类型以及联合类型等。本章介绍数组类型的数据，结构和联合类型数据将在第 5 章介绍。由于数组和指针有着十分密切的联系，因此本章还将介绍在 C 语言中用途极为广泛的指针类型数据。

4.1 数组的定义与引用

数组是一组有序数据的集合，数组中的每一个数据都属于同一种数据类型。数组中的各个元素可以用数组名和下标来惟一地确定。一维数组只有一个下标，多维数组有两个以上的下标。在 C 语言中数组必须先定义，然后才能使用。一维数组的定义形式如下：

数据类型 数组名[常量表达式];

其中，“数据类型”说明了数组中各个元素的类型。

“数组名”是整个数组的标识符，它的定名方法与变量的定名方法一样。

“常量表达式”说明了该数组的长度，即该数组中的元素个数。常量表达式必须用方括号“[]”括起来，而且其中不能含有变量。下面是几个定义一维数组的例子：

```
char x[5];           /* 定义字符型数组 x，它具有 5 个元素 */
int y[10];           /* 定义整型数组 y，它具有 10 个元素 */
float z[15];         /* 定义浮点型数组 z，它具有 15 个元素 */
```

定义多维数组时，只要在数组名后面增加相应于维数的常量表达式即可。对于二维数组的定义形式为：

数据类型 数组名[常量表达式 1][常量表达式 2];

例如，要定义一个 10×10 的整数矩阵 A，可以采用如下的定义方法：

```
int A[10][10];
```

需要指出的是，C 语言中数组的下标是从 0 开始的，因此对于数组 `char x[5]` 来说，其中的 5 个元素是 `x[0]~x[4]`，不存在元素 `x[5]`，这一点在引用数组元素时是应当加以注意的。C 语言规定在引用数值数组时，只能逐个引用数组中的各个元素而不能一次引用整个数组；但如果是字符数组则可以一次引用整个数组。

例 4.1：用数组计算并输出 Fibonacci 数列的前 20 项。

Fibonacci 数列在数学和计算机算法研究中是十分有用的。Fibonacci 数列是这样的一

组数：第一个数字为 0，第二个数字为 1，之后每个数字都是前两个数字之和。

```
#include<stdio.h>
main() {
    int f[20], i;
    f[0]=0;
    f[1]=1;
    for (i=2;i<20; i++)
        f[i]=f[i-2]+f[i-1];
    for(i=0; i<20; i++)
    {
        if(i%5==0)
            printf("\n");
        printf("%10d",f[i]);
    }
    while(1);
}
```

程序执行结果：

0	1	1	2	3
5	8	13	21	34
55	89	144	233	377
610	987	1597	2584	4181

4.2 字符数组

用来存放字符数据的数组称为字符数组，它是 C 语言中常用的一种数组。字符数组中的每个元素都是一个字符，因此可用字符数组来存放不同长度的字符串。字符数组的定义方法与一般数组相同，下面是两个定义字符数组的例子：

```
char menu[20];
char string[50];
```

在 C 语言中字符串是作为字符数组来处理的。一个一维的字符数组可以存放一个字符串，这个字符串的长度应小于或等于字符数组的长度。为了测定字符串的实际长度，C 语言规定以“\0”作为字符串结束标志，对字符串常量也自动加一个“\0”作为结束符。因此字符数组 `char menu[20]` 可存储一个长度 ≤ 19 的不同长度的字符串。在访问字符数组时，遇到“\0”就表示字符串结束，因此在定义字符数组时，应使数组长度大于它允许存放的最大字符串的长度。另外，符号“\0”是一个表示 ASCII 码值为 0 的字符，它不是一个可显示字符，而是一个“空操作符”，在这里仅仅起一个结束标志的作用。

对于字符数组的访问可以通过数组中的元素逐个进行访问，也可以对整个数组进行访问。

例 4.2：对字符数组进行输入和输出。

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

```
main() {  
    char c[10];  
    scanf("%s",c);  
    printf("%s\n",c);  
    while(1);  
}
```

程序中用“%s”格式控制输入输出字符串,这里的输入输出操作是对整个字符数组进行的,因此输入项必须是数组名c,而不能用数组元素名c[i]。在μVision2环境下对例4.2程序编译链接通过后进行仿真调试,启动程序全速运行,将光标移到串行窗口,从键盘输入HELLO并回车,系统会自动在输入的字符串后面加一个结束符‘\0’,然后输出HELLO,如果输入的字符数大于10,则只取前10个字符作为有效字符输出。

上节介绍了数组的定义方法,可以在内存中开辟一个相应于数组元素个数的存储空间,数组中各个元素的赋值是在程序运行过程中进行的。如果希望在定义数组的同时给数组中各个元素赋以初值,可以采用如下方法:

数据类型 [存储器类型] 数组名[常量表达式]={常量表达式};

其中,“数据类型”指出数组元素的数据类型。

“存储器类型”是可选项,指出定义数组所在的存储器空间。

“常量表达式表”中给出各个数组元素的初值。

需要注意的是,在定义数组的同时对数组元素赋初值时,初值的个数必须小于或等于数组中元素的个数(即数组长度),否则在程序编译时作为出错处理。赋初值时可以不指定数组的长度,编译器会根据初值的个数自动计算出该数组的长度。因此数组名后面的“常量表达式”为可选项,省略该选项时数组的长度由实际初值的个数决定。例如:

```
unsigned char a[5]={0x11,0x22,0x33,0x44,0x55}  
unsigned char xdata a[]={0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88,0x99}
```

对于多维数组可以采用同样的方法来赋值,例如,可用下面的方法在定义一个二维数组的同时赋以初值:

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

例4.3: 用冒泡法对一组数据进行排序。

```
#include <reg51.h>  
#include <stdio.h>  
void main() {  
    unsigned char xdata a[10]=  
    {0x3f,0x44,0x32,0x54,0x66,0x56,0x99,0x88,0x77,0x11,0x34};  
    unsigned char i,j,t;  
    printf("the unsorted numbers : \n");  
    for (i=0;i<=9;i++)  
        printf("%bx ",a[i]);  
    printf("\n");
```

```

    for (j=0;j<=8;j++)
        for (i=0;i<=9-j;i++)
            if (a[i]>a[i+1]) {t=a[i];a[i]=a[i+1];a[i+1]=t;}
    printf("the sorted numbers : \n");
    for (i=0;i<=10;i++) printf("%bx ",a[i]);
    while(1);
}

```

程序执行结果:

the unsorted numbers :

0x3f,0x44,0x32,0x54,0x66,0x56,0x99,0x88,0x77,0x11,0x34

the sorted numbers :

0x11,0x32,0x34,0x3f,0x44,0x54,0x56,0x66,0x77,0x88,0x99

给字符数组赋初值对于在程序存储器 ROM 中制作一些常数表格特别有用,例如,可以采用如下方法在 ROM 中制作一张共阴极 LED 的显示字符段码表:

```
char code SEG[11] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
```

利用字符数组可以很方便地实现 LED 段码的查表显示。图 4.1 是利用 8031 单片机串行口实现的动态 LED 扫描显示接口电路,在串行口上扩展一片移位寄存器 74LS164 作为共阴极 7 段 LED 的段码数据口,8031 的 P1.0~P1.3 作为 LED 显示器的位扫描信号,串行口工作于移位寄存器方式(方式 0)。执行下面的程序后可在 LED 上显示出“8031”这几个数字。

例 4.4: 利用字符数组实现 LED 数字显示。

```

#include <reg51.h>
char code seg[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
void dsp(char digit, j) {
    unsigned char tmp;
    SBUF=seg[digit];
    P1=j;
    tmp=0x7f;
    while(tmp--);
}

void main() {
    SCON=0x00;
    while(1) {
        dsp(0x08, 1);
        dsp(0x00, 2);
        dsp(0x03, 4);
        dsp(0x01, 8);
    }
}

```

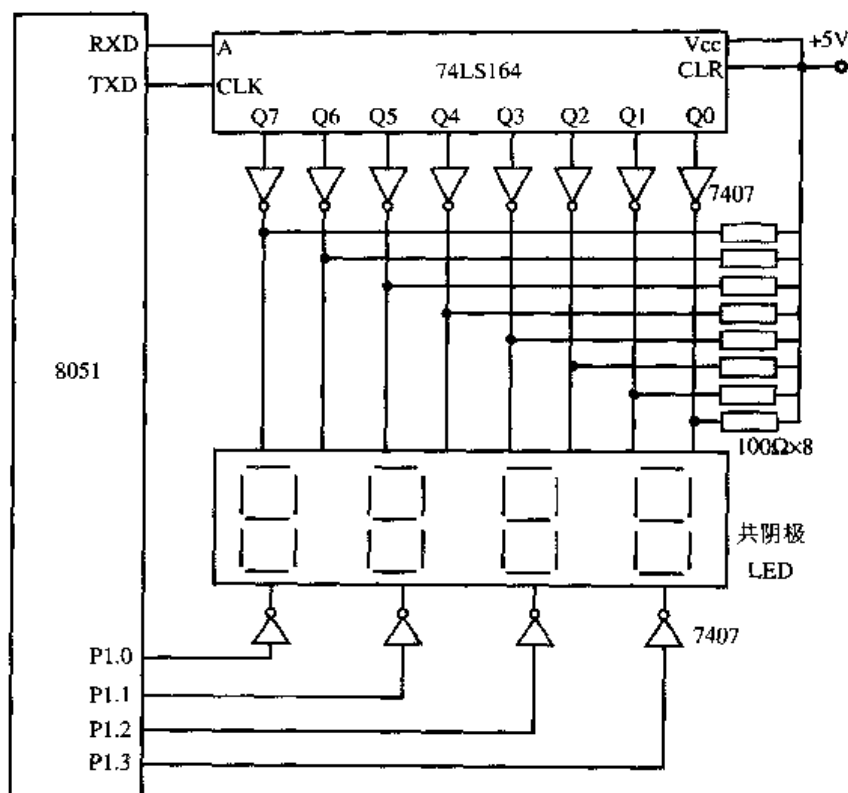


图 4.1 用串行口实现的 LED 动态显示电路

4.3 数组作为函数的参数

除了可以用变量作为函数的参数之外，还可以用数组名作为函数的参数。一个数组的数组名表示该数组的首地址。用一个数组名作为函数的参数时，在进行函数调用的过程中参数传递方式采用的是地址传递。将实际参数数组的首地址传递给被调函数中的形式参数数组，这样一来两个数组就占用同一段内存单元。如图 4.2 所示，若数组 a 的起始地址为 0x1000，则数组 b 的起始地址也是 0x1000，显然数组 a 和 b 占用同一段内存单元。

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
起始地址 1000	1	3	5	7	9	11	13	15	17	19
	b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]

图 4.2 两个数组占用同一段内存单元

用数组名作为函数的参数，应该在主调函数和被调函数中分别进行数组定义，而不能只在一方定义数组。而且在两个函数中定义的数组类型必须一致，如果类型不一致将导致编译出错。实参数组和形参数组的长度可以一致也可以不一致，编译器对形参数组的长度不作检查，只是将实参数组的首地址传递给形参数组。如果希望形参数组能得到实参数组的全部元素，则应使两个数组的长度一致。定义形参数组时可以不指定长度，只在数组名后面跟一个空的方括号[]。这时为了在被调函数中处理数组元素的需要，应另外设置一个参数来传递数组元素的个数。

例 4.5: 用数组作为函数的参数, 计算两个不同长度的数组中所有元素的平均值。

```
#include<stdio.h>
float average(array, n)
int n;
float array[];
{
    int i;
    float aver, sum=array[0];
    for (i=1; i<n; i++)
        sum=sum+array[i];
    aver=sum/n;
    return(aver);
}

main() {
    float pot_1[5]={99.9,88.8,77.7,66.6,0};
    float pot_2[10]={11.1,22.2,33.3,44.4,55.5,99.9,88.8,77.7,66.6,0};
    printf("the average of A is %6.2f\n", average(pot_1,5));
    printf("the average of B is %6.2f\n", average(pot_2,10));
    while(1);
}
```

程序执行结果:

the average of A is 66.60

the average of B is 49.95

在这个程序中定义了一个求平均值的函数 `average()`, 它有两个形式参数 `array` 和 `n`。`array` 是一个长度不定的 `float` 类型的数组, `n` 是 `int` 类型的变量。在主函数 `main()` 中定义了两个长度确定的 `float` 类型的数组 `pot_1[5]` 和 `pot_2[10]`。通过嵌套函数调用实现求数组元素平均值的运算并输出。可以看到, 两次调用 `average()` 函数时数组的长度是不同的, 在调用时通过一个实际参数 (5 或 10) 将数组的长度传递给形式参数 `n`, 从而使 `average()` 函数能处理数组 `pot_1` 和 `pot_2` 中的所有元素。

用数组名作为函数的参数, 参数的传递过程采用的是地址传递。地址传递方式具有双向传递的性质, 即形式参数的变化将导致实际参数也发生变化, 这种性质在程序设计中有时是很有用的。下面的例子就是利用这一性质来对数组 `a` 中的 10 个整数按从小到大的顺序排序。

例 4.6: 采用选择法对数组元素进行排序。

```
#include<stdio.h>
void sort (array, n)
int array[];
int n;
```



```
{
    int i, j, k, t;
    for (i=0; i<n-1; i++)
    {
        k=i;
        for (j=i+1; j<n; j++)
            if (array[j]<array[k]) k=j;
        t=array[k]; array[k]=array[i]; array[i]=t;
    }
}

main() {
    int a[10], i;
    printf("please enter the array\n");
    for (i=0; i<10; i++)
        scanf("%d", &a[i]);
    sort(a, 10);
    printf("the sorted array is: \n");
    for (i=0; i<10; i++)
        printf("%d ", a[i]);
    printf("\n");
    while(1);
}
```

程序执行结果:

please enter the array

9

7

5

3

1

8

6

4

2

0

the sorted array is:

0 1 2 3 4 5 6 7 8 9

在这个例子中, 主程序在执行函数调用语句 `sort(a, 10)` 的前后, 数组 `a` 中各个元素的值是不同的。执行函数调用语句之前, 数组 `a` 中的元素是无序的, 执行函数调用语句之后, 数组 `a` 中的元素已经按大小排序。这是因为在执行被调用函数时, 形式参数数组已经进行了排序, 而形式参数数组的改变会使实际参数数组随之改变, 从而使主函数中的数组 `a` 也

进行了排序。

对于多维数组作为函数的参数与一维数组的情形类似。可以用多维数组名作为函数的实际参数和形式参数，在被调用函数中对形式参数说明时可以指定每一维的长度，也可以省略数组第一维的长度说明，但是绝不能省略第二维以及其他高维的长度说明。因为从实际参数传送过来的是数组的起始地址，在内存中数组是按行存放的，而并不区分数组的行和列。如果在形式参数中不说明列数，编译器就无法确定该数组有几行几列。

例 4.7：求一个 3×4 矩阵中的最大元素。

```
#include<stdio.h>
max(int array[][4]) {
    int i, j, max;
    max=array[0][0];
    for (i=0; i<3; i++)
        for (j=0; j<4; j++)
            if (array[i][j]>max) max=array[i][j];
    return(max);
}

main() {
    int a[3][4]={{1,3,5,7},{2,4,6,8},{15,17,34,12}};
    printf("max value is %d\n", max(a));
    while(1);
}
```

程序执行结果：

max value is 34

4.4 指 针

指针是 C 语言中的一个重要概念，指针类型数据在 C 语言程序中的使用十分普遍。正确地使用指针类型数据，可以有效地表示复杂的数据结构，直接处理内存地址，而且可以更为有效地使用数组。

4.4.1 指针与地址

众所周知，一个程序的指令、常量和变量等都要存放在机器的内存单元中，而机器的内存是按字节来划分存储单元的。给内存中每个字节都赋予一个编号，这就是存储单元的地址。各个存储单元中所存放的数据，称为该存储单元的内容。计算机在执行任何一个程序时都要涉及许多的寻址操作，所谓寻址，就是按照内存单元的地址来访问该存储单元中的内容，即按地址来读或写该单元中的数据。由于通过地址可以找到所需要的存储单元，因此可以说地址是指向存储单元的。

在 C 语言中为了能够实现直接对内存单元进行操作，引入了指针类型的数据。指针类

型数据是专门用来确定其他类型数据地址的，因此一个变量的地址就称为该变量的指针。例如，有一个整型变量 *i* 存放在内存单元 40H 中，则该内存单元地址 40H 就是变量 *i* 的指针。如果有一个变量专门用来存放另一个变量的地址，则称之为“指针变量”，例如，如果用另一个变量 *ip* 来存放整型变量 *i* 的地址 40H，则 *ip* 即为一个指针变量。

变量的指针和指针变量是两个不同的概念。变量的指针就是该变量的地址，而一个指针变量里面存放的内容是另一个变量在内存中的地址，拥有这个地址的变量则称为该指针变量所指向的变量。每一个变量都有它自己的指针（即地址），而每一个指针变量都是指向另一个变量的。为了表示指针变量和它所指向的变量之间的关系，C 语言中用符号“*”来表示“指向”。例如，整型变量 *i* 的地址 40H 存放在指针变量 *ip* 中，则可用 **ip* 来表示指针变量 *ip* 所指向的变量，即 **ip* 也表示变量 *i*，下面两个赋值语句：

```
i=0x50;  
*ip=0x50;
```

都是给同一个变量赋值 0x50。图 4.3 形象地说明了指针变量 *ip* 和它所指向的变量 *i* 之间的关系。



图 4.3 指针变量和它所指向的变量

从图 4.3 可以看到，对于同一个变量 *i*，可以通过变量名 *i* 来访问它，也可以通过指向它的指针变量 *ip*，用 **ip* 来访问它。前者称为直接访问，后者称为间接访问。符号“*”称为指针运算符，它只能与指针变量一起联用，运算的结果是得到该指针变量所指向的变量的值。在 2.2.5 节中介绍了一个取地址运算符“&”，它可以与一个变量联用，其作用是求取该变量的地址。通过运算符“&”可以将一个变量的地址赋值给一个指针变量。例如：赋值语句 *ip=&i*；它的作用是取得变量 *i* 的地址并赋给指针变量 *ip*。通过这种赋值后即可以说指针变量 *ip* 指向了变量 *i*。不要将符号“&”和“*”弄混淆，&*i* 是取变量 *i* 的地址，**ip* 是取指针变量 *ip* 所指向的变量的值。

4.4.2 指针变量的定义

指针变量的定义与一般变量的定义类似，其一般形式如下：

数据类型 [存储器类型 1] * [存储器类型 2] 标识符；

其中“标识符”是所定义的指针变量名。

“数据类型”说明了该指针变量所指向的变量的类型。

“存储器类型 1”和“存储器类型 2”是可选项，它是 Keil Cx51 编译器的一种扩展，如果带有“存储器类型 1”选项，指针被定义为基于存储器的指针，无此选项时，被定义为一般指针。这两种指针的区别在于它们的存储字节不同。一般指针在内存中占用 3 个字节，第一个字节存放该指针存储器类型的编码（由编译时编译模式的默认值确定），第二

个和第三个字节分别存放该指针的高位和低位地址偏移量。存储器类型的编码值如下：

存储器类型 1	idata/data/bdata	xdata	pdata	code
编码值	0x00	0x01	0xFE	0xFF

“存储器类型 2”选项用于指定指针本身的存储器空间。

一般指针可用于存取任何变量而不必考虑变量在 8051 单片机存储器空间的位置，许多 C51 库函数采用了一般指针，函数可以利用一般指针来存取位于任何存储器空间的数据。下面是一个一般指针定义的例子，同时给出了汇编语言代码，请注意汇编语言代码中指针第一个字节是存储器类型编码值，第二、三字节是地址偏移量。

例 4.8：定义一般指针。

```

stmt level   source
1           char *c_ptr;           /* char ptr */
2           int *i_ptr;            /* int ptr */
3           long *l_ptr;           /* long ptr */
4
5           void main (void)
6           {
7 1         char data dj;           /* data vars */
8 1         int data dk;
9 1         long data dl;
10 1
11 1        char xdata xj;          /* xdata vars */
12 1        int xdata xk;
13 1        long xdata xl;
14 1
15 1        char code cj = 9;       /* code vars */
16 1        int code ck = 357;
17 1        long code cl = 123456789;
18 1
19 1
20 1        c_ptr = &dj;            /* data ptrs */
21 1        i_ptr = &dk;
22 1        l_ptr = &dl;
23 1
24 1        c_ptr = &xj;            /* xdata ptrs */
25 1        i_ptr = &xk;
26 1        l_ptr = &xl;
27 1
28 1        c_ptr = &cj;            /* code ptrs */
29 1        i_ptr = &ck;
30 1        l_ptr = &cl;
31 1        }

```

32

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION main (BEGIN)
; SOURCE LINE # 5
; SOURCE LINE # 6
; SOURCE LINE # 20
0000 750000    R    MOV    c_ptr,#00H
0003 750000    R    MOV    c_ptr+01H,#HIGH dj
0006 750000    R    MOV    c_ptr+02H,#LOW dj
; SOURCE LINE # 21
0009 750000    R    MOV    i_ptr,#00H
000C 750000    R    MOV    i_ptr+01H,#HIGH dk
000F 750000    R    MOV    i_ptr+02H,#LOW dk
; SOURCE LINE # 22
0012 750000    R    MOV    l_ptr,#00H
0015 750000    R    MOV    l_ptr+01H,#HIGH dl
0018 750000    R    MOV    l_ptr+02H,#LOW dl
; SOURCE LINE # 24
001B 750001    R    MOV    c_ptr,#01H
001E 750000    R    MOV    c_ptr+01H,#HIGH xj
0021 750000    R    MOV    c_ptr+02H,#LOW xj
; SOURCE LINE # 25
0024 750001    R    MOV    i_ptr,#01H
0027 750000    R    MOV    i_ptr+01H,#HIGH xk
002A 750000    R    MOV    i_ptr+02H,#LOW xk
; SOURCE LINE # 26
002D 750001    R    MOV    l_ptr,#01H
0030 750000    R    MOV    l_ptr+01H,#HIGH xl
0033 750000    R    MOV    l_ptr+02H,#LOW xl
; SOURCE LINE # 28
0036 7500FF    R    MOV    c_ptr,#0FFH
0039 750000    R    MOV    c_ptr+01H,#HIGH cj
003C 750000    R    MOV    c_ptr+02H,#LOW cj
; SOURCE LINE # 29
003F 7500FF    R    MOV    i_ptr,#0FFH
0042 750000    R    MOV    i_ptr+01H,#HIGH ck
0045 750000    R    MOV    i_ptr+02H,#LOW ck
; SOURCE LINE # 30
0048 7500FF    R    MOV    l_ptr,#0FFH
004B 750000    R    MOV    l_ptr+01H,#HIGH cl
004E 750000    R    MOV    l_ptr+02H,#LOW cl
; SOURCE LINE # 31
0051 22        RET

```

```
; FUNCTION main (END)
```

上例中一般指针 `c_ptr`、`i_ptr`、`l_ptr` 全部位于 8051 单片机的片内数据存储器中，如果在定义一般指针时带有“存储器类型 2”选项，则可指定一般指针本身的存储器空间位置，例如：

```
char * xdata strptr;          /* 位于 xdata 空间的一般指针 */
int * data numptr;            /* 位于 data 空间的一般指针 */
long * idata varptr;          /* 位于 idata 空间的一般指针 */
```

由于一般指针所指对象的存储器空间位置只有在运行期间才能确定，编译器在编译期间无法优化存储方式，必须生成一般代码以保证能对任意空间的对象进行存取，因此一般指针所产生的代码运行速度较慢，如果希望加快运行速度则应采用基于存储器的指针。基于存储器的指针所指对象具有明确的存储器空间，长度可为 1 个字节（存储器类型为 `idata`、`data`、`pdata`）或 2 个字节（存储器类型为 `code`、`xdata`），例如：

```
char data * str;              /* 指向 data 空间 char 型数据的指针 */
int xdata * numtab;           /* 指向 xdata 空间 int 型数据的指针 */
long code * powtab;           /* 指向 code 空间 long 型数据的指针 */
```

与一般指针类似，若定义时带有“存储器类型 2”选项，则可指定基于存储器的指针本身的存储器空间位置，例如：

```
char data * xdata str;
int xdata * data numtab;
long code * idata powtab;
```

基于存储器的指针长度比一般指针短，可以节省存储器空间，运行速度快，但它所指对象具有确定的存储器空间，缺乏灵活性。下面是一个基于存储器的指针定义例子，其汇编代码长度明显小于一般指针。

例 4.9：基于存储器的指针定义。

stmt	level	source
1		<code>char data *c_ptr; /* memory-specific char ptr */</code>
2		<code>int xdata *i_ptr; /* memory-specific int ptr */</code>
3		<code>long code *l_ptr; /* memory-specific long ptr */</code>
4		
5		<code>long code powers_of_ten [] =</code>
6		<code>{</code>
7		<code>1L,</code>
8		<code>10L,</code>
9		<code>100L,</code>
10		<code>1000L,</code>
11		<code>10000L,</code>
12		<code>100000L,</code>
13		<code>1000000L,</code>

```

14      100000000L,
15      1000000000L,
16      };
17
18      void main (void) {
19  1      char data strbuf [10];
20  1      int xdata ringbuf [1000];
21  1
22  1      c_ptr = &strbuf [0];
23  1      i_ptr = &ringbuf [0];
24  1      l_ptr = &powers_of_ten [0];
25  1      }
26
ASSEMBLY LISTING OF GENERATED OBJECT CODE
      ; FUNCTION main (BEGIN)
                                     ; SOURCE LINE # 18
                                     ; SOURCE LINE # 22
0000 750000      R      MOV      c_ptr,#LOW strbuf
                                     ; SOURCE LINE # 23
0003 750000      R      MOV      i_ptr,#HIGH ringbuf
0006 750000      R      MOV      i_ptr+01H,#LOW ringbuf
                                     ; SOURCE LINE # 24
0009 750000      R      MOV      l_ptr,#HIGH powers_of_ten
000C 750000      R      MOV      l_ptr+01H,#LOW powers_of_ten
                                     ; SOURCE LINE # 25
000F 22          RET
      ; FUNCTION main (END)

```

在一些函数调用中进行参数传递时需要采用一般指针，例如，Cx51 的库函数 `printf()`、`sprintf()`、`gets()` 等便是如此。当传递的参数是基于存储器的指针时，若不特别指明，Keil Cx51 编译器会自动将其转换为一般指针，如果被调用函数的参数应该为某种较短长度指针，则会产生程序出错，为避免此类错误，应采用 `#include` 预处理器命令将函数的说明文件包含到 C 语言源程序中去。

一般指针与基于存储器的指针转换规则如下（一般指针用 GP 表示）：

GP→xdata: 使用 GP 的偏移部分（2 字节）。

GP→code: 同上。

GP→idata: 使用 GP 偏移部分的低字节，高字节不用。

GP→data: 同上。

GP→pdata: 同上。

xdata→GP: 一般指针的存储器类型编码被设定为 0x01，使用 xdata * 的双字节偏移量。

code→GP: 一般指针的存储器类型编码被设定为 0xFF，使用 code * 的双字节偏移量。

idata→GP: 一般指针的存储器类型编码被设定为 0x00，指针的一字节偏移量被转换

为 unsigned int 类型。

data→GP: 同上。

pdata→GP: 一般指针的存储器类型编码被设定为 0xFE, 指针的一字节偏移量被转换为 unsigned int 类型。

例 4.10: 指针转换及其所生成的代码。

stmt level source

```

1      int *p1;          /* 一般指针, 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 指针转换为一般指针 */
8  1      p1 = p3;        /* idata 指针转换为一般指针 */
9  1      p1 = p4;        /* code 指针转换为一般指针 */
10  1
11  1      p4 = p1;        /* 一般指针转换为 code 指针 */
12  1      p3 = p1;        /* 一般指针转换为 idata 指针 */
13  1      p2 = p1;        /* 一般指针转换为 xdata 指针 */
14  1
15  1      p2 = p3;        /* idata 指针转换为 xdata 指针 (警告错误) */
16  1      p3 = p4;        /* code 指针转换为 idata 指针 (警告错误) */
17  1      }
18

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

; FUNCTION pconvert (BEGIN)

; SOURCE LINE # 6

; SOURCE LINE # 7

```

0000 750001    R    MOV    p1,#01H
0003 850000    R    MOV    p1+01H,p2
0006 850000    R    MOV    p1+02H,p2+01H

```

; SOURCE LINE # 8

```

0009 750000    R    MOV    p1,#00H
000C 750000    R    MOV    p1+01H,#00H
000F 850000    R    MOV    p1+02H,p3

```

; SOURCE LINE # 9

```

0012 AA00      R    MOV    R2,p4
0014 A900      R    MOV    R1,p4+01H
0016 7BFF      MOV    R3,#0FFH
0018 8B00      R    MOV    p1,R3
001A 8A00      R    MOV    p1+01H,R2

```



```

001C 8900      R    MOV    p1+02H,R1
                                   ; SOURCE LINE # 11
001E AE02            MOV    R6,AR2
0020 AF01            MOV    R7,AR1
0022 8E00      R    MOV    p4,R6
0024 8F00      R    MOV    p4+01H,R7
                                   ; SOURCE LINE # 12
0026 AF01            MOV    R7,AR1
0028 8F00      R    MOV    p3,R7
                                   ; SOURCE LINE # 13
002A AE02            MOV    R6,AR2
002C 8E00      R    MOV    p2,R6
002E 8F00      R    MOV    p2+01H,R7
                                   ; SOURCE LINE # 15
0030 750000      R    MOV    p2,#00H
0033 8F00      R    MOV    p2+01H,R7
                                   ; SOURCE LINE # 16
0035 850000      R    MOV    p3,p4+01H
                                   ; SOURCE LINE # 17
0038 22          RET
                                   ; FUNCTION pconvert (END)

```

4.4.3 指针变量的引用

指针变量是含有一个数据对象地址的特殊变量，指针变量中只能存放地址。与指针变量有关的运算符有两个，它们是取地址运算符&和间接访问运算符*。例如：&a 为取变量 a 的地址，*p 为指针变量 p 所指向的变量。

指针变量经过定义之后可以像其他基本类型变量一样引用。例如：

- 变量定义

```
int i, x, y, *pi, *px, *py;
```

- 指针赋值

```
pi=&i;    /* 将变量 i 的地址赋给指针变量 pi, 使 pi 指向 i */
px=&x;    /* px 指向 x */
py=&y;    /* py 指向 y */

```

- 指针变量引用

```
*pi=0;    /* 等价于 i=0; */
*pi+=1;   /* 等价于 i+=1; */
(*pi)++;  /* 等价于 i++; */

```

指向相同类型数据的指针之间可以相互赋值。例如：

```
px=py;
```

原来指针 px 指向 x, py 指向 y, 经上述赋值之后, px 和 py 都指向 y。

例 4.11: 输入两个整数 x 和 y, 经比较后按大小顺序输出。

```
#include<stdio.h>
main() {
    int x, y;
    int *p, *p1, *p2;
    printf("Input x and y: \n");
    scanf("%d %d", &x, &y);
    p1=&x; p2=&y;
    if(x<y) {
        p=p1; p1=p2; p2=p;
    }
    printf("max=%d, min=%d, \n", *p1, *p2);
    while(1);
}
```

程序执行结果:

```
Input x and y:
4 8 回车
max=8, min=4
```

这个程序中定义了三个指针变量 *p、*p1 和 *p2, 它们都指向整型变量。经过赋值之后, p1 指向 x, p2 指向 y。然后比较变量 x 和 y 的大小, 若 $x < y$, 则将 p1 和 p2 交换, 使 p1 指向 y, p2 指向 x; 若 $x > y$ 则不交换。最后的结果必然使指针 p1 指向较大的数, p2 指向较小的数, 按顺序输出 *p1 和 *p2 的值, 就可得到正确的结果。值得注意的是在程序执行过程中, 变量 x 和 y 的值并未交换, 所交换的只是它们的指针。由于指针 p、p1 和 p2 都是指向 int 类型数据的指针, 故可以相互赋值, 实现指针 p1 和 p2 的交换。

4.4.4 指针变量作为函数的参数

函数的参数不仅可以是整型、实型、字符型等数据, 还可以是指针类型的数据。指针变量作为函数的参数的作用是将一个变量的地址传送到另一个函数中去, 地址传递是双向的, 即主调用函数不仅可以向被调用函数传递参数, 而且还可以从被调用函数返回其结果。下面通过一个例子来进行说明。

例 4.12: 利用指针变量作为函数的参数实现两个元素的相互交换。

```
#include <stdio.h>
swap(int * pi, int * pj) {
    int temp;
    temp = * pi;
    * pi = * pj;
    * pj = temp;
}
```

```
}

main() {
    int a,b;
    int *pa, *pb;
    printf("Please input a and b: \n");
    scanf("%d %d", &a, &b);
    pa = &a;
    pb = &b;
    if(a<b) swap(pa, pb);
    printf("\n max=%d, min=%d \n", a, b);
    while(1);
}
```

程序执行结果:

```
Please input a and b:
1234 5678 回车
max=5678, min=1234
```

程序中自定义函数 swap()的功能是交换两个变量 a 和 b 的值, swap()函数的两个形式参数 p1 和 p2 是指针变量。程序开始执行时,先输入 a 和 b 的值,然后将 a 和 b 的地址分别赋给指针变量 pa 和 pb,使 pa 指向 a, pb 指向 b。接着执行 if 语句,如果 a<b 则调用 swap()函数。注意函数调用时的实参 pa 和 pb 也是指针变量,在调用开始时,实参变量将它的值传递给形参变量,采取的仍然是“值传递”方式,但这时传递的是指针的值(即地址)。参数传递完成后,形参 pi 的值为&a, pj 的值为&b,即指针变量*pi 和*pa 都指向了变量 a, *pj 和*pb 都指向了变量 b。接着执行 swap()函数体,使*pi 和*pj 的值互换,从而实现了 a 和 b 的值的互换。函数返回时,虽然 pi 和 pj 被释放而不再存在,但 main()函数中 a 和 b 的值已经被交换,因此最后能输出正确的结果。

由此可见以指针变量作为函数的参数,被调用函数在执行过程中使指针变量所指向的变量值发生变化,函数调用结束后,这些变量值的变化将被保留下来,从而可以实现“在被调用函数中改变变量的值,在主调用函数中使用这些被改变了的值”。如果希望通过函数调用得到 n 个要改变的变量,可以在主调用函数中设置 n 个变量,再用 n 个指针变量来指向它们,然后将指针变量作为实参将这 n 个变量的地址传递给被调用函数中的形参,通过形参指针变量来改变这 n 个变量的值,被改变了值将被保留下来,最后在主调用函数中就可以使用这些被改变了的值。

4.5 数组的指针

4.5.1 用指针引用数组元素

在 C 语言中指针与数组有着十分密切的关系,任何能够用数组实现的运算都可以通过

指针来完成。例如，定义一个具有十个元素的整型数组可以写成：

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

数组 *a* 中各个元素分别为 *a*[0]、*a*[1]、...、*a*[9]。数组名 *a* 表示元素 *a*[0] 的地址，而 **a* 则表示 *a* 所代表的地址中的内容，即 *a*[0]。

如果定义一个指向整型变量的指针 *pa* 并赋以数组 *a* 中第一个元素 *a*[0] 的地址：

```
int *pa;  
pa=&a[0];
```

则可以通过指针 *pa* 来操作数组 *a* 了。即可用 **pa* 代表 *a*[0]，*(*pa*+*i*) 代表 *a*[*i*] 等，也可以使用 *pa*[0]、*pa*[1]、...、*pa*[9] 的形式。

例 4.13：使用指针与数组的例子——计算质数。

```
#include <stdio.h>  
#define MAX 20  
main() {  
    int i, j, n, p, r, primes[MAX];  
    int *pntw, *pntr;  
    long q;  
    pntw = primes;  
    n = 2;  
    *pntw++ = 2; *pntw++ = 3;  
    i = 5;  
    do {  
        pntr = primes;  
        do {  
            p = *pntr++;  
            q = i / p;  
            r = i - q * p;  
        } while( r && i < q*q );  
        if( r ) {  
            *pntw++ = i;  
            n++;  
        }  
        i += 2;  
    } while( n < MAX );  
    j = 0;  
    pntr = primes;  
    for( i=0; i<MAX; ++i ) {  
        printf("%4d", *pntr++);  
        if( ++j == 10 ) {  
            printf("\n");  
            j = 0;  
        }  
    }  
}
```

```

    }
}
while(1);
}

```

程序执行结果:

```

2   3   5   7   11  13  17  19  23  29
31  37  41  43  47  53  59  61  67  71

```

4.5.2 字符数组指针

用指针来描述一个字符数组是十分方便的。前面已经讲过,字符串是以字符数组的形式给出的,并且每个字符数组都以转义字符“\0”作为字符串的结束标志。因此在判别一个字符数组是否结束时,通常不采用计数的方法,而是以是否读到转义字符“\0”来判别。利用这个特点,可以很方便地用指针来处理字符数组。

例 4.14: 利用指针将一个字符数组中的字符串复制到另一个字符数组中去。

```

#include<stdio.h>
main() {
    char *s1;
    char xdata *s2;
    char code str[]={"How are you ? "};
    s1=str;
    s2=0x1000;
    while ((*s2-*s1)!='\0') {
        s2++; s1++;
    };
    s1=str;
    s2=0x1000;
    printf("%s\n, %s\n", s1, s2);
    while(1);
}

```

程序执行结果:

```

How are you ?
How are you ?

```

任何一个数组及其数组元素都可以用一个指针及其偏移值来表示,但要注意的是,指针是一个变量,因此像上例中的赋值运算 `s1=str`、`s2=0x1000` 等都是合法的;而数组名是一个常量,不能像变量那样进行运算,即数组的地址是不能改变的。例如上面程序中的语句:

```
char code str[]={"How are you ? "};
```

是将字符串“`How are you ?`”置到数组 `str[]`中作为初值,而语句:

```
s1=str;
```

则是将数组 `str[]` 的首地址，即指向数组 `str` 的指针赋值给指针变量 `s1`。如果对数组名进行如下的操作：

```
str=s1;
str++;
```

则都是错误的。

4.5.3 指针的地址计算

指针的地址计算包括以下几个方面的内容。

(1) 赋初值

指针变量的初值可以是 `NULL`（零），也可以是变量、数组、结构以及函数等的地址。例如：

```
int a[10], b[10];
float fbuf[100];
char *cptr1=NULL;
char *cptr2=&ch;
int *iptr1=&a[5];
int *iptr2=b;
float *fptr1=fbuf;
```

(2) 指针与整数的加减

指针可以与一个整数或整数表达式进行加减运算，从而获得该指针当前所指位置前面或后面某个数据的地址。假设 `p` 为一个指针变量，`n` 为一个整数，则 `p±n` 表示离开指针 `p` 当前位置的前面或后面第 `n` 个数据的地址。

(3) 指针与指针相减

指针与指针相减的结果为一整数值，但它并不是地址，而是表示两个指针之间的距离或元素的个数。注意，这两个指针必须指向同一类型的数据。

(4) 指针与指针的比较

指向同一类型数据的两个指针可以进行比较运算，从而获得两指针所指地址的大小关系。

此外，在计算指针地址的同时，还可进行间接取值运算。不过在这种情况下，间接取值的地址应该是地址计算后的结果，并且还必须注意运算符的优先级和结合规则。设 `p1`、`p2` 都是指针，对于

```
a=*p1++;
```

由于运算符 `*` 和 `++` 具有相同的优先级而指针运算具有右结合性，按右结合规则，有 `++`、`*` 的运算次序，而运算符 `++` 在 `p1` 的后面，因此上述赋值运算的过程是首先将指针 `p1` 所指的内容赋值给变量 `a`，然后 `p1` 再指向下一个数据，表明是地址增加而不是内容增加。对于

```
a=*--p1;
```

与上例相同，按右结合规则有 `--`、`*` 的运算次序，而运算符 `--` 在 `p1` 的前面，因此首先将

p1 减去 1, 即指向前面一个数据, 然后再把 p1 此时所指的内容赋值给变量 a。对于

```
a=(*p2)++;
```

由于使用括号()使结合次序变为*、++, 因此首先将 p2 所指的内容赋值给变量 a, 然后再把 p2 所指的内容加 1, 表明是内容增加而不是地址增加。

例 4.15: 指针运算的例子。

```
#include <stdio.h>
main() {
    char data *p1, *p2, *p3, *px;
    char x[]={1,2,3,4};
    char px1, px2, px3;
    px=p1=p2=p3=x;
    px1=*p1++;
    p2=p1+2;
    px2=*--p2;
    px3=(*p3)++;
    printf("The start address of ARRAY x[] is %P\n",px);
    printf("px1 = %bd, p1 = %P\n",px1, p1);
    printf("px2 = %bd, p2 = %P\n",px2, p2);
    printf("px3 = %bd, p3 = %P\n",px3, p3);
    while(1);
}
```

程序执行结果:

```
The start address of ARRAY x[] is D:002C
px1=1, p1=D:002D
px2=3, p2=D:002E
px3=1, p3=D:002C
```

例 4.16: 两指针相减——计算字符串长度的函数。

```
#include <stdio.h>
main() {
    char *s = "abcdef";
    int strlen(char *s);
    printf("\n length of '%s' = %d\n",s,strlen(s));
    while(1);
}

int strlen(char *s) {
    char *p = s;
    while( *p != '\0' ) p++;
    return( p - s );
}
```

程序执行结果:

lenth of 'abcdef' = 6

需要指出的是, 指针的运算是很有限的, 它只能进行如上所述的四种运算操作, 除此之外所有其他的指针运算都是非法的。特别强调一点, 不允许对两个指针进行加、乘、除、移位或屏蔽运算, 也不允许用 float 类型数据与指针作加减运算。

4.6 函数型指针

函数不是变量, 但它在内存中仍然需要占据一定的存储空间, 如果将函数的入口地址赋给一个指针, 该指针就是函数型指针。由于函数型指针指向的是函数的入口地址, 因此在进行函数调用时可用指向函数的指针代替被调用的函数名。在 C 语言中函数与变量不同, 函数名不能作为参数直接传递给另一个函数。但是利用函数型指针, 可以将函数作为参数传递给另一个函数。此外还可以将函数型指针放在一个指针数组中, 则该指针数组的每一个元素都是指向某个函数的指针。

定义一个函数型指针的一般形式为:

数据类型 (* 标识符)()

其中, “标识符”就是所定义的函数型指针变量名。

“数据类型”说明了该指针所指向的函数返回值的类型。例如:

```
int (* func1)();
```

定义了一个函数型指针变量 func1, 它所指向的函数返回值为整型数据。函数型指针变量是专门用来存放函数入口地址的, 在程序中把哪个函数的地址赋给它, 它就指向那个函数。在程序中可以对一个函数型指针多次赋值, 该指针可以先后指向不同的函数。

给函数型指针赋值的一般形式为:

函数型指针变量名 = 函数名

如果有一个函数 max(x,y), 则可用如下的赋值语句将该函数的地址赋给函数型指针 func1, 使 func1 指向函数 max:

```
func1=max;
```

引入了函数型指针之后, 对于函数的调用可以采用两种方法。例如, 程序中要求将函数 max(x,y)的值赋给变量 z, 可采用如下方法:

```
z=max(x,y); 或 z=(* func1)(x, y);
```

用这两种方法实现函数调用的结果是完全一样的。但需要注意的是, 若采用函数型指针来调用函数, 必须先对该函数指针进行赋值, 使之指向所需调用的函数。

函数型指针通常用来将一个函数的地址作为参数传递到另一个函数中去。这种方法对于要调用某个非固定函数的场合特别适用。下面通过一个例子来说明函数型指针的这种应用。

例 4.17: 函数型指针作为函数的参数。

设置一个函数 `process()`，每次调用它时完成不同的功能。输入两个整型数 `a` 和 `b`，第一次调用时找出 `a` 和 `b` 中较大者，第二次调用时找出较小者，第三次调用时求出 `a` 与 `b` 的和。

```
#include <stdio.h>

int max(int x, int y) {
    int z;
    if (x>y) z=x;
    else z=y;
    return(z);
}

int min(int x, int y) {
    int z;
    if (x<y) z=x;
    else z=y;
    return(z);
}

int add(int x, int y) {
    int z;
    z=x+y;
    return(z);
}

int process(int x, int y, int (*f)()) {
    int result;
    result=f();
    printf("%d\n", result);
}

main() {
    int a, b;
    printf("Please input a and b: \n");
    scanf("%d %d", &a, &b);
    printf("max=");
    process(a, b, max);
    printf("min=");
    process(a, b, min);
    printf("sum=");
    process(a, b, add);
    while(1);
}
```

程序执行结果:

 Please input a and b:

```
1234 5678 回车
max=5678
min=1234
sum=6912
```

本例中的三个函数 max()、min()和 add()分别用来实现求较大数、求较小数和求和的功能。在主函数 main()中第一次调用 process()函数时,除了将 a 和 b 作为实参将两个数传递给 process()的形参 x、y 之外,还将函数名 max 作为实参将其入口地址传递给 process()函数中的形参—指向函数的指针变量*f。这样一来, process()函数中的函数调用语句 result=f();就相当于 result=max(x, y);因此,执行 process()函数即可求出 a 与 b 中的较大者。第二次调用 process()函数时改用函数名 min 作为实参,第三次调用 process()函数时改用函数名 add 作为实参,从而实现每次调用 process()函数时完成不同的功能。

4.7 返回指针型数据的函数

上一节中介绍了函数型指针的概念,在使用过程中要注意函数型指针与返回指针型数据的函数的区别。在函数的调用过程结束时,被调用的函数可以带回一个整型数据、字符型数据、实型数据等,也可以带回一个指针型数据,即地址。这种返回指针型数据的函数又称为指针函数,它的一般定义形式为:

数据类型 * 函数名(参数表);

其中,数据类型说明了所定义的指针函数在返回时带回的指针所指向的数据类型。例如:

```
int * x(a, b);
```

定义了一个指针函数*x,调用它以后可以得到一个指向整型数据的指针,即地址。请读者注意,在指针函数*x 的两侧没有括号(),这是与函数型指针完全不同的,也是容易搞混的,实际使用时一定要注意。

例 4.18: 指针函数应用。

内存中存有巡回检测 3 个通道的温度值(每个通道有 4 个点),要求用户在输入通道号以后,能立即输出该通道所有点的温度值。

```
#include<stdio.h>
main() {
    float T[3][4]=
        {{60.1,70.3,80.5,90.7},{30.0,40.1,50.2,60.3}, {90.0,80.2,70.4,60.6}};
    float * search(float (* pointer)[4], int n);
    float * p;
    int i, m;
    printf("Enter the number of chanal: ");
    scanf("%d", &m);
    printf("\n The temperature of chanal %d are: \n", m);
    p=search(T, m);
```

```

    for (i=0; i<4; i++)
        printf("%5.1f ", *(p+i));
    while(1);
}

float * search (float (* pointer)[4], int n)
{
    float *pt;
    pt= * (pointer+n);
    return(pt);
}

```

程序执行结果:

```

Enter the number of chanal: 2    回车
The temperature of chanal are:
90.0  80.2  70.4  60.6

```

程序中将巡回检测得到的某个通道各点的温度值存放在一个二维数组 $T[3][4]$ 中, 通道号作为数组的行, 各点温度值作为数组的列。在输入通道号时要注意, 通道号是从 0 算起的。函数 `search` 被定义为指针型函数, 它的形式参数 `pointer` 是指向包含 4 个元素的一维数组的指针变量。`pointer+1` 指向二维数组 T 的第 0 行, 而 `*(pointer+1)` 则指向第 0 行第 0 列元素。`pt` 是一个指针变量, 它指向实型变量 (而不是指向一维数组)。`main()` 函数调用 `search()` 函数, 将 T 数组的首地址传递给 `pointer` (注意 T 是指向数组行的指针, 而不是指向数组列元素的指针)。 m 是要查找的通道号。调用 `search()` 函数后, 得到一个地址 (指向第 m 个通道第 0 点温度值), 并将这个地址赋给变量 `p`。`*(p+i)` 表示该通道第 i 点的温度值, 从而可将该通道 4 个点的温度值输出来。读者可参考图 4.4 来加深对这个例子的理解。

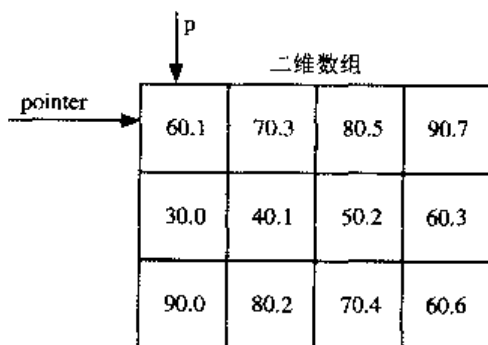


图 4.4 数组与指针的关系

4.8 指针数组与指针型指针

4.8.1 指针数组

由于指针本身也是一个变量, 因此 C 语言允许定义指针的数组, 指针数组适合于用来

指向若干个字符串,使字符串的处理更为方便。指针数组的定义方法与普通数组完全相同,一般格式为:

数据类型 * 数组名[数组长度]

例如:

```
int * x[2];          /* 指向整型数据的 2 个指针 */
char * sptr[5];      /* 指向字符型数据的 5 个指针 */
```

指针数组在使用之前往往需要先赋初值,方法与一般数组赋初值类似。使用指针数组最典型的场合是通过对字符数组赋初值而实现各维长度不一致的多维数组的定义,下面通过两个例子来进一步说明指针数组赋初值的问题。

例 4.19: 使用指针数组的例子。

```
#include<stdio.h>
main() {
    int i;
    char code *season[4]=
    {
        "spring", "summer", "fall", "winter"
    };
    for(i=0; i<4; ++i)
        printf("\n%c---%s", *season[i], season[i]);
    while(1);
}
```

程序执行结果:

```
s---spring
s---summer
f---fall
w---winter
```

这个例子中在 code 区定义了指向 char 型数据的 4 个指针,其初值分别为“spring”、“summer”、“fall”和“winter”。它们可用图 4.5 直观地表示。

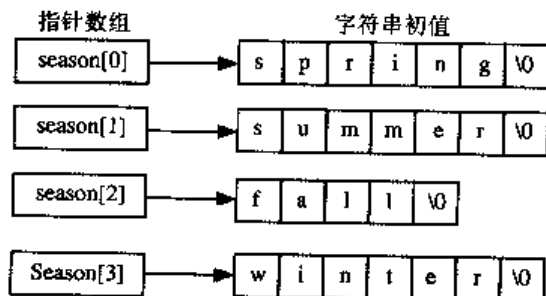


图 4.5 指针数组与所赋初值的关系

从图 4.5 中可以看到各个指针数组的长度可以不同,实际的存储空间分配情况如图 4.6 所示,它们是各行首尾相接的连续区域,这样可以更为有效地利用内存空间。如果不使用

指针数组而采用二维字符数组, 则该二维数组各列的长度必须相等, 并且要等于最大一列的长度, 这样会造成内存空间的浪费。

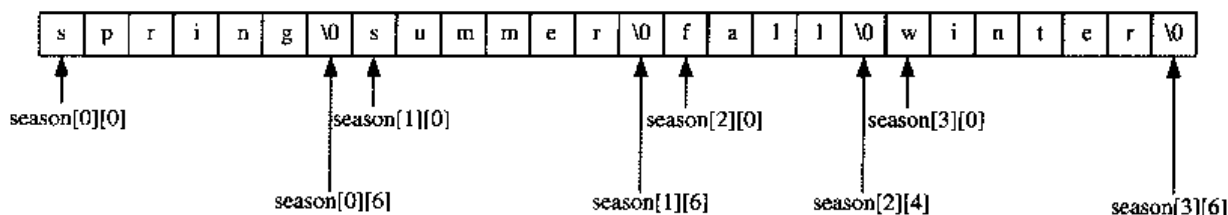


图 4.6 指针数组的初值在内存中的存放格式

例 4.20: 指针数组的应用。

日期转换程序, 输入年份和天数, 输出这一天所在的月份及该月天数。

```
#include<stdio.h>
char code daytab[2][13]=
{
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}
};

char * mname(int n) {
    char code *mn[]=
    {
        "illegal month", "January", "February",
        "March", "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };
    return((n<1||n>12)? mn[0]: mn[n]);
}

monthday(int y, int yd) {
    int i, leap;
    leap=y%4==0&&y%100!=0||y%400==0;
    for(i=1; yd>daytab[leap][i]; i++)
        yd-=daytab[leap][i];
    printf("%s, %d\n", mname(i), yd);
}

main() {
    int year, yearday;
    printf("Input year and yearday: \n");
    scanf("%d, %d", &year, &yearday);
    monthday(year, yearday);
}
```

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

程序执行结果:

Input year and yearday:

1996, 60 回车

February, 29

这个程序由三个函数组成: 主函数 `main()`、求月份名函数 `mname()` 和求月天数函数 `monthday()`。在主函数 `main()` 中输入年份 `year` 和天数 `yearday`, 然后调用求月天数函数将该年的这一天转换为该年的某月某日。

求月天数函数 `monthday()` 中定义了一张每月的天数表。由于二月的天数因闰年和平年而不同, 因此将天数表设计为二维数组, 用变量 `leap` 来判别输入的年份是否为闰年, 闰年 `leap` 为 1, 平年 `leap` 为 0。for 语句的作用是控制当输入天数大于每月的天数时, 用输入的天数减去该月的天数, 并使月数 `i+1`, 再进入下一个循环, 直到天数不大于第 `i` 月的天数时, 退出循环。此时的 `i` 值即为所求的月数, 而天数则为该月的天数。例如, 在程序执行中, 输入 1996, 60, 经过上述转换得到 `i=2`, `yd=29`, 即 1996 年的第 60 天为 2 月 29 日。

求月份名函数 `mname()` 的作用是将由函数 `monthday()` 所求得的月数能用相应的月份名来表示。即要求对于给出的月份数 `n`, 能返回一个指向 `n` 月名字字符串的指针。函数中定义了一个指针数组 `mn` 并给它赋了初值, `mn` 中共有 13 个元素, 它们都是指针, 指向字符类型数据, 其初值分别是字符串 “illegal month”、“January”、…、“December” 的首地址。函数 `monthday()` 调用 `mname(i)` 后, 返回一个指向第 `i` 月的月名字字符串指针 `mn[i]`, 然后用格式 “%s” 输出, 就能得到该指针所指向的月名字字符串。

如果一个指针数组中的元素都是函数指针, 则称之为函数指针数组。利用函数指针数组可以很方便地实现散转处理。在设计一个单片机应用系统时, 键盘管理程序是整个系统监控程序的一个重要组成部分。为了实现各个不同按键的功能, 通常是根据不同的键值进行散转。下面是一个键值散转处理的例子, 根据从键盘输入不同的数字键进行不同的处理。

例 4.21: 键值散转处理。

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

```
void Key_0() {  
    /* "0" 键处理 */  
}
```

```
void Key_1() {  
    /* "1" 键处理 */  
}
```

```
void Key_2() {  
    /* "2" 键处理 */  
}
```

```

}

/* k3, k4, ... 其他键处理 */

/* 函数指针数组定义 */
code void (code * KeyProcTab[])()= {Key_0, Key_1, Key_2/*k2,...,k9 */ };

void main() {
    unsigned char key,num;
    while(1){
        scanf ("%c", &key);          /* 等待按键 */
        num=key-0x30;                  /* 计算键值 */
        (*KeyProcTab[num])();         /* 按键值散转 */
    }
}

```

4.8.2 指针型指针

在掌握了指针数组的基础上,再来介绍一种指针型的指针,这种指针所指向的是另一个指针变量的地址,故有时也称之为多级指针。从前面的图4.5中可以看到,指针数组 `season` 中的每一个元素都是一个指针型数据。既然 `season` 是一个数组,那么它的每一个元素都有相应的地址,而数组名 `season` 则代表了该指针数组的首地址。因此我们可以定义一个指向指针数组中元素的指针变量,这就是指向指针型数据的指针变量,即指针型指针变量。

定义一个指针型指针变量的一般形式为:

数据类型 ** 标识符

其中,“标识符”就是所定义的指针型指针变量名,而“数据类型”则说明一个被指针型指针所指向的指针变量所指向的变量数据类型。

例 4.22: 指针型指针变量的应用。

```

#include<stdio.h>
main() {
    int x, *p, ** q;
    x=10;
    p=&x;
    q=&p;
    printf("%d\n:", x);
    printf("%d\n:", * p);
    printf("%d\n:", ** q);
    while(1);
}

```

程序执行结果:

10

```
10
```

```
10
```

本例程序中定义了一个指向整型数据的指针变量 *p*，又定义了一个指向整型数据的指针型指针 *q*。经过赋值之后，指针 *p* 指向整型变量 *x*，而指针型指针 *q* 则指向指针变量 *p*。换句话说，在 *q* 中存放的是 *p* 的地址，在 *p* 中存放的是 *x* 的地址，在 *x* 中存放的才是整型数据 10。若要将这个整型数据的值输出，可以通过变量名 *x* 直接输出：

```
printf("%d\n:", x);
```

这种方法称为直接取值。也可以先通过指针变量 *p* 找到变量 *x* 的地址，然后再从这个地址中将整型数据的值输出：

```
printf("%d\n:", *p);
```

这种方法称为单重间接取值。还可以通过指针型指针变量 *q* 先找到指针 *p* 的地址，再通过指针 *p* 找到 *x* 的地址，最后从 *x* 的地址中将整型数据的值输出：

```
printf("%d\n:", **q);
```

这种方法称为多重间接取值。从程序的执行结果可以看到，这三种方法得到的结果是完全一样的。由此可知，一个指针型指针是一种间接取值的形式，而且这种间接取值的方式还可以进一步延伸，故可以将这种多重间接取值的形式看成为一个指针链。

指针型指针常用来作为指向指针数组的指针变量。例如：

```
int * x[5];  
int ** y;
```

其中，第一条语句定义了一个指向整型数据的指针数组 *x[5]*，它由 5 个元素组成，各个元素 *x[0]*、*x[1]*、…、*x[4]* 均为指针变量，它们都指向整型数据；第二条语句定义了一个指针型指针 *y*。如果经过如下赋值：

```
y=x;
```

则 *y* 就成了指向指针数组 *x* 的多级指针了。

例 4.23：使用指针型指针的例子。

```
#include<stdio.h>  
main() {  
    char i;  
    char ** j;  
    char *season[4]=  
    {  
        "spring", "summer", "fall", "winter"  
    };  
    for(i=0; i<4; ++i) {  
        j=season+i;  
        printf("\n%c---%s", *season[i], *j);  
    }
```



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

程序执行结果:

```
s---spring  
s---summer  
f---fall  
w---winter
```

这个程序的执行结果与例 4.19 程序是完全一样的。在这个程序中先定义了一个指针数组 `season`，并给它赋以字符串初值。在指针数组 `season` 中有 4 个元素，它们分别指向 4 个字符串的首地址。程序中还定义了一个指针型指针 `j`，并进行了赋值：

```
j=season+1;
```

当 `i` 为 0 时，`*j` 为 `season[0]` 的值，即第一个字符串的首地址，用字符串格式 “%s” 输出 `*j` 就可以得到第一个字符串，以后循环 `i` 为 1、2、3 时，`j` 分别为 `j[1]`、`j[2]` 和 `j[3]`，用 “%s” 输出即可依次得到各个字符串。

4.8.3 抽象型指针

ANSI 新标准增加了一种 “void *” 指针类型，这是一种抽象型指针，即可以定义一个指针变量，但不指定该指针是指向哪一种类型数据的。对于这种抽象型指针在给另一个指针变量赋值时，需要进行强制类型转换，使之适合于被赋值的变量的类型。例如：

```
char *p1;  
void *p2;  
p1=(char *)p2;
```

当然也可以用 `(void *)p1` 将 `p1` 的地址转换成 `void *` 类型之后赋值给 `p2`：

```
p2=(void *)p1;
```

函数也可以定义为 `void *` 类型，例如：

```
void * fun(x, y)
```

表示函数 `fun` 返回的是一个地址，它指向 “空类型”。

抽象型指针可以用来在每个存储区内访问任意绝对地址，或者用来产生绝对调用。下面是一个使用 Keil Cx51 编译器对抽象型指针形成汇编码的例子，读者可以从这个例子中进一步了解如何正确使用这种抽象类型的指针。

例 4.24：抽象指针的使用。

stmt level	source
1	char xdata * px;
2	char idata * pi;
3	char code * pc;

```

4      char c;
5      int i;
6      void main(void){
7  1    pc=(void *) main;
8  1    pi=(char idata *)&i;
9  1    pi=(char idata *)px;
10  1    pc=(char code *)0x7788;
11  1
12  1    i=((int (code *) (void))0xff00)(); /* LCALL 0FF00H */
13  1    c=*((char code *)0x8000);          /* char [code[0x8000]] */
14  1    c+*((char xdata *)0xff00);         /* char [xdata[0xFF00]] */
15  1    c+*((char idata *)240);            /* char [idata[0xF0]] */
16  1    c+*((char pdata *)240);           /* char [pdata[0xF0]] */
17  1
18  1    i=((int code *)0x1200);             /* int from code[0x1200] */
19  1    px=*((char xdata *xdata *)0x4000); /* x-ptr from xdata[0x4000] */
20  1    px=((char xdata *xdata *)0x4000)[0]; /* same as before */
21  1    px=((char xdata *xdata *)0x4000)[1]; /* x-ptr from xdata[0x4002] */
22  1  }
23
24

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

; FUNCTION main (BEGIN)

; SOURCE LINE # 6

; SOURCE LINE # 7

0000 750000 R MOV pc,#HIGH main

0003 750000 R MOV pc+01H,#LOW main

; SOURCE LINE # 8

0006 750000 R MOV pi,#LOW i

; SOURCE LINE # 9

0009 850000 R MOV pi,px+01H

; SOURCE LINE # 10

000C 750077 R MOV pc,#077H

000F 750088 R MOV pc+01H,#088H

; SOURCE LINE # 12

0012 12FF00 LCALL 0FF00H

0015 8E00 R MOV i,R6

0017 8F00 R MOV i+01H,R7

; SOURCE LINE # 13

0019 908000 MOV DPTR,#08000H

001C E4 CLR A

001D 93 MOVC A,@A+DPTR

001E F500 R MOV c,A

; SOURCE LINE # 14

```

0020 90FF00      MOV     DPTR,#0FF00H
0023 E0          MOVX    A,@DPTR
0024 2500        R      ADD     A,C
0026 F500        R      MOV     C,A
                                ; SOURCE LINE # 15
0028 78F0        MOV     R0,#0F0H
002A E6          MOV     A,@R0
002B 2500        R      ADD     A,C
002D F500        R      MOV     C,A
                                ; SOURCE LINE # 16
002F E2          MOVX    A,@R0
0030 2500        R      ADD     A,C
0032 F500        R      MOV     C,A
                                ; SOURCE LINE # 18
0034 901200      MOV     DPTR,#01200H
0037 E4          CLR     A
0038 93          MOVC    A,@A+DPTR
0039 F500        R      MOV     i,A
003B 7401        MOV     A,#01H
003D 93          MOVC    A,@A+DPTR
003F F500        R      MOV     i+01H,A
                                ; SOURCE LINE # 19
0040 904000      MOV     DPTR,#04000H
0043 E0          MOVX    A,@DPTR
0044 FE          MOV     R6,A
0045 A3          INC     DPTR
0046 E0          MOVX    A,@DPTR
0047 8E00        R      MOV     px,R6
0049 F500        R      MOV     px+01H,A
                                ; SOURCE LINE # 20
004B 8E00        R      MOV     px,R6
004D F500        R      MOV     px+01H,A
                                ; SOURCE LINE # 21
004F A3          INC     DPTR
0050 E0          MOVX    A,@DPTR
0051 FE          MOV     R6,A
0052 A3          INC     DPTR
0053 E0          MOVX    A,@DPTR
0054 8E00        R      MOV     px,R6
0056 F500        R      MOV     px+01H,A
                                ; SOURCE LINE # 22
0058 22          RET
; FUNCTION main (END)

```

第 5 章 结构、联合与枚举

在本书第 2 章介绍了 C 语言中的基本数据类型,但是在实际进行 C51 程序设计时仅有这些基本类型的数据是不够的,例如,有时需要将一批各种类型的数据放在一起使用,从而引入了所谓构造类型的数据。本书第 4 章中所介绍的数组就是一种构造类型的数据,一个数组实际上是一批顺序存放的相同类型数据。本章介绍 C 语言中另一类更为常用的构造类型数据:结构、联合和枚举。

5.1 结构变量的定义与引用

结构是一种构造类型的数据,它是将若干个不同类型的数据变量有序地组合在一起而形成的一种数据的集合体。组成该集合体的各个数据变量称为结构成员,整个集合体使用一个单独的标识符作为结构变量名。一般来说结构中的各个变量之间是存在某些关联的,时间数据中的时、分、秒,日期数据中的年、月、日等。由于结构是将一组相关联的数据变量作为一个整体来进行处理,因此在程序中使用结构将有利于对一些复杂而又具有内在联系的数据进行有效的管理。

有三种定义结构变量的方法,分述如下。

1. 先定义结构类型再定义结构变量名

定义结构类型的一般格式为:

```
struct 结构名  
{ 结构元素表 };
```

其中,“结构元素表”为该结构中的各个成员(又称为结构的域),由于结构可以由不同类型的数据组成,因此对结构中的各个成员都要进行类型说明。例如,定义一个日期结构类型 `date`,它由三个结构元素 `year`、`month`、`day` 组成,定义格式如下:

```
struct date {  
    int year;  
    char month, day;  
}
```

定义好一个结构类型之后,就可以用它来定义结构变量。一般格式为:

```
struct 结构名 结构变量名 1, 结构变量名 2, ..., 结构变量名 n;
```

例如,可以用结构 `date` 来定义两个结构变量 `d1` 和 `d2`:

```
struct date d1, d2;
```

结构变量 d1 和 d2 都具有 struct date 类型的结构, 即它们都是由一个整型数据和两个字符型数据所组成的。

2. 在定义结构类型的同时定义结构变量名

这种方法是将方法 1 的两个步骤合在一起, 一般格式为:

```
struct 结构名
```

```
{ 结构元素表 } 结构变量名 1, 结构变量名 2, ..., 结构变量名 n;
```

例如, 对于上述日期结构变量也可按以下格式定义:

```
struct date {  
    int year;  
    char month, day;  
} d1, d2;
```

3. 直接定义结构变量

这种方法可以省略掉结构名, 又称为无名结构, 一般格式为:

```
struct
```

```
{ 结构元素表 } 结构变量名 1, 结构变量名 2, ..., 结构变量名 n;
```

例如, 上述日期结构变量可按以下格式定义:

```
struct {  
    int year;  
    char month, day;  
} d1, d2;
```

第三种方法与第二种方法十分相似, 所不同的只是第三种方法中省略了结构名“date”。这种方法一般只用于定义几个确定的结构变量的场合。在上例中, 如果只需要定义 d1 和 d2 而不打算再定义任何别的结构变量, 则可省略掉结构名“date”。不过为了便于记忆和以备将来进一步定义其他结构变量的需要, 一般还是不要省略结构名为好。此外, 也可以使用关键字 typedef 来命名一个结构类型, 这时结构名就不太重要了。例如:

```
typedef struct {  
    int year;  
    char month, day;  
} complex;
```

这样 complex 实际上就成了一个结构类型, 可以直接用 complex 来定义结构变量, 例如:

```
complex x, y;
```

则变量 x 和 y 就都具有上述的结构类型。

关于结构变量的应用, 有以下 5 点需要注意的地方。

① 结构类型与结构变量是两个不同的概念。定义一个结构类型时只是给出了该结构的组织形式, 并没有给出具体的组织成员。因此结构名不占用任何存储空间, 也不能对一个结构名进行赋值、存取和运算。而结构变量则是一个结构中的具体组织成员, 编译器会

给具体的结构变量名分配确定的存储空间,因此可以对结构变量名进行赋值、存取和运算。

② 将一个变量定义为标准类型与将其定义为结构类型略有不同。前者只需要用类型说明符指出变量的类型即可,如 `int x; char y, z;` 等。后者不仅要求用 `struct` 指出该变量为结构类型,而且还要求指出该变量是哪种特定的结构类型,即要指出它所属的特定结构类型的名字。如上面例子中的 `date` 就是这种特定的结构类型的名字。

③ 一个结构中的结构元素还可以是另外一个结构类型的变量,即可以形成结构的嵌套。例如:

```
struct clock {
    unsigned char hour, min, sec;
}
struct mrec {
    unsigned char port1;
    unsigned char analog[4];
    struct clock time;
} m1
```

其中,结构类型 `mrec` 中的结构元素 `time` 又是另一个结构类型 `clock` 的结构变量,形成了结构的结构,即结构的嵌套。结构的嵌套可以是多层次的,但这种嵌套不能包含其自身,即不能自己定义自己。

④ 一个结构中的结构元素的名字可以与程序中其他地方使用的变量同名,它们二者各自代表不同的对象,在使用中不会互相影响。与普通变量相似,在定义结构变量时,还可以说明它的存储种类,可以有 `extern`、`auto` 和 `static` 三种形式。

⑤ 定义了一个结构变量之后,就可以对它进行引用,完成赋值、存取和运算等。一般情况下,结构变量的引用是通过对其结构元素的引用来实现的。引用结构元素的一般格式为:

结构变量名.结构元素

其中“.”是存取结构元素的成员运算符。例如,`d1.year` 表示结构变量 `d1` 中的元素 `year`,`d2.day` 表示结构变量 `d2` 中的元素 `day` 等。如果一个结构变量中的结构元素又是另外一个结构变量,即出现结构的嵌套时,则需要采用若干个成员运算符,一级一级地找到最低一级的结构元素,而且只能对这个最低级的结构元素进行访问。

对结构变量中的各个元素可以像普通变量一样进行赋值、存取和运算。例如:

```
d1.year=1997;
sum=d1.day+d2.day;
d1.month++;
m1.time.hour=0x22;
```

其中成员运算符“.”的优先级别最高,因此 `d1.month++` 是对 `d1.month` 进行自加运算,而不是先对 `month` 进行自加运算。`m1.time.hour` 是对嵌套结构类型中最低一层的结构元素进行访问。

对于结构变量和结构元素,可以在程序中直接引用它们的地址。例如,若需要输入

d1.year 的值，可以写成：

```
scanf("%d",&d1.year);
```

又如，需要输出结构变量 d2 的首地址，可以写成：

```
printf("%c",&d2);
```

结构变量的地址通常用作函数参数，用来传递结构的地址。

5.2 结构变量的初值

结构变量有三种存储种类，它们是 `extern`、`static` 和 `auto`（即外部、静态和自动）。当结构变量为外部全局变量或静态变量时，可以在定义结构类型时给它赋初值，但不能给自动存储种类的动态局部结构变量赋初值。

例 5.1：给外部结构变量赋初值。

```
#include<stdio.h>
struct mepoint {
    unsigned char name[11];
    unsigned char pressure;
    unsigned char temperature;
} pol={ "firstpoint", 0x99, 0x64};
void main(void) {
    printf(" name: %s\n pressure: %c\n temperature :%c\n", pol.name,
        pol.pressure, pol.temperature);
    while(1);
}
```

程序执行结果：

```
name: firstpoint
pressure: 99
temperature: 64
```

例 5.2：给静态结构变量赋初值。

将例 5.1 中的外部定义部分放到主函数 `main()` 中即使之成为静态结构变量。

```
#include<stdio.h>
void main(void) {
    struct mepoint {
        unsigned char name[11];
        unsigned char pressure;
        unsigned char temperature;
    } pol={ "firstpoint", 0x99, 0x64};

    printf(" name: %s\n pressure: %c\n temperature :%c\n", pol.name,
```

```
        pol.pressure, pol.temperature);  
    while(1);  
}
```

程序的执行结果与例 5.1 完全相同。

对于自动结构变量不能在定义时赋初值，只能在程序执行中用赋值语句给各个结构元素分别赋值。给结构变量赋初值与给数组变量赋初值是一样的，即当初值个数不够时，余下的结构变量元素就以 0 作为其初值。但是，如果初值个数过多则会导致编译时出错。

5.3 结构数组

在实际使用中，结构变量往往不止一个，通常是将多个相同的结构组成一个数组，这就是结构数组。结构数组的定义方法与结构变量完全一致，例如：

```
struct mepoint  
{  
    unsigned char name[11];  
    unsigned char pressure;  
    unsigned char temperature;  
};  
struct mepoint po[3];
```

这就定义了一个包含有 3 个元素的结构数组变量 po[3]，其中每个元素都是具有 mepoint 结构类型的变量。当然也可以采用另外两种方法来定义结构数组变量 po[3]。对于外部或静态结构数组也可以赋初值。

例 5.3：结构数组赋初值。

```
#include<stdio.h>  
#include<reg51.h>  
struct mepoint {  
    unsigned char name[11];  
    unsigned char pressure;  
    unsigned char temperature;  
};  
struct mepoint po[3]=  
{  
    { "1stpoint", 0x99, 0x64 },  
    { "2ndpoint", 0x10, 0x20 },  
    { "3tdpoint", 0x15, 0x18 }  
};  
void main(void) {  
    unsigned char i;  
    printf("No.   point      pressure  temperature \n");
```



```

for (i=0; i<3; i++)
printf("%bd    %s    %bx    %bx\n",
        i+1,po[i].name,po[i].pressure,po[i].temperature);
while(1);
}

```

程序执行结果:

No.	point	pressure	temperature
1	1stpoint	99	64
2	2ndpoint	10	20
3	3tdpoint	15	18

5.4 结构型指针

5.4.1 结构型指针的概念

一个指向结构类型变量的指针称为结构型指针,该指针变量的值是它所指向的结构变量的起始地址。结构型指针也可用来指向结构数组,或指向结构数组中的元素。

定义结构型指针的一般格式为:

struct 结构类型标识符 *结构指针标识符

其中,“结构指针标识符”就是所定义的结构型指针变量的名字,“结构类型标识符”就是该指针所指向的结构变量的具体类型名字。对于上节例 5.3 中所定义的 **mepoint** 结构类型,可以定义如下的结构指针:

```
struct mepoint *mp;
```

这里的 **mp** 即可用来指向 **mepoint** 类型的结构变量或结构数组。图 5.1 是该结构指针的示意图。与一般指针相同,对于结构指针 **mp** 也必须先赋值后才能引用。

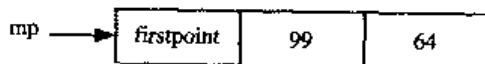


图 5.1 指向结构 **mepoint** 的指针

5.4.2 用结构型指针引用结构元素

通过结构型指针来引用结构元素的一般格式为:

结构指针->结构元素

与前面讲过的引用结构元素的格式相比较,这里只不过是用符号“->”(减号加大于号)取代了符号“.”而已。例如: **mp->pressure** 完全等效于 **(* mp).pressure**。

结构指针也可以进行运算,运算规则与普通指针相同。例如,对于指向结构数组的结构指针 **mp** 的增量运算 **mp++**,将根据该结构的实际大小来增加,增加的结果将使该指针指向下一个结构元素,如图 5.2 所示。

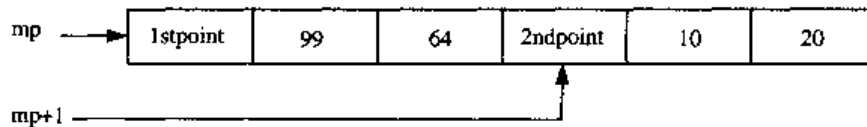


图 5.2 指向结构的指针的增量运算

例 5.4: 用结构指针修改例 5.3 的程序。

```
#include<stdio.h>
#include<reg51.h>
struct mepoint {
    unsigned char name[11];
    unsigned char num, pressure, temperature;
};
struct mepoint po[3]=
{
    { "1stpoint", 0x01, 0x99, 0x64 },
    { "2ndpoint", 0x02, 0x10, 0x20 },
    { "3tdpoint", 0x03, 0x15, 0x18 }
};
void main(void) {
    struct mepoint * mp;
    printf("No.   point      pressure  temperature \n");
    for (mp=po; mp<po+3; mp++)
        printf("%bx    %s      %bx          %bx\n",
            mp->num, mp->name, mp->pressure, mp->temperature);
    while(1);
}
```

程序执行结果:

No.	point	pressure	temperature
1	1stpoint	99	64
2	2ndpoint	10	20
3	3tdpoint	15	18

5.5 结构与函数

5.5.1 将结构作为函数的参数

一般来说, 结构既可作为函数的参数, 也可作为函数的返回值。当结构被用作函数的参数时, 其用法与普通变量作为实参是一样的, 其参数传递属于“值传递”方式。

例 5.5: 将结构作为函数的参数, 该程序接受在键盘上输入的今日的日期, 同时输出明日的日期。

```
#include <stdio.h>

struct ydate {
    unsigned int year;
    unsigned char month, day;
};

leapyear(struct ydate d) {
    unsigned char leapy=0;
    if((d.year%4==0 && d.year%100!=0) || (d.year%400==0))
        leapy=1;
    return(leapy);
}

numdays(struct ydate d) {
    unsigned char day;
    static char daytab[]={
        {31,28,31,30,31,30,31,31,30,31,30,31}};
    if(leapyear(d)&& d.month==2)
        day=29;
    else
        day=daytab[d.month-1];
    return(day);
}

main() {
    struct ydate today, tomorrow;
    printf("Please enter today's date(mm,dd,yyyy):\n");
    scanf("%bd,%bd,%d",&today.month,&today.day,&today.year);
    if(today.day!=numdays(today)) {
        tomorrow.day=today.day+1;
        tomorrow.month=today.month;
        tomorrow.year=today.year;
    }
    else
        if(today.month==12) {
            tomorrow.day=1;
            tomorrow.month=1;
            tomorrow.year=today.year+1;
        }
    else {
        tomorrow.day=1;
        tomorrow.month=today.month+1;
        tomorrow.year=today.year;
    }
}
```

```
    }  
    printf("Tomorrow's date is %bd/%bd/%d. \n", tomorrow.month,  
          tomorrow.day, tomorrow.year);  
    while(1);  
}
```

程序执行结果:

Please enter today's date(mm,dd,yyyy):

2,28,1996

Tomorrow's date is 2,29,1996

程序再执行一次的结果为:

Please enter today's date(mm,dd,yyyy):

12,31,1996

Tomorrow's date is 1,1,1997

这个程序由三个函数组成,它们是主函数 `main()`、确定该月天数的函数 `numdays(struct ydate d)` 和闰年判断函数 `leapyear(struct ydate d)`。程序一开始在所有函数的外部定义了一个全程结构类型 `ydate`,它可以供整个程序中的所有函数来定义具有这种类型的结构变量。

`leapyear(struct ydate d)` 函数以具有 `ydate` 类型的结构变量 `d` 为参数,其作用是判断 `d.year` 是否为闰年,若是,函数返回值为 1,若不是,函数返回值为 0。

`numdays(struct ydate d)` 函数也是以结构变量 `d` 作为参数,其作用是计算 `d.month` 所指出的月份的天数。函数中定义了一个静态一维数组并且赋以 1 至 12 月每月的天数。只有当 `d.year` 为闰年且 `d.month` 为 2 月时,要重新对 `day` 赋以初值 29,其他情况下 `day` 都是从数组中取出当月的天数。函数的返回值为 `day`,即为所求对应于 `d.month` 月的天数。

主函数 `main()` 首先接受键盘输入的今日日期: `today.month`、`today.day` 以及 `today.year`,然后进行分支判断:

如果今日的日期不等于该月的天数,

则明日的日期为本年本月本日的下一日;

否则(即今日的日期等于该月的天数)

如果今日的月数等于 12,

则明日日期为明年 1 月 1 日;

否则(即今日月数不等于 12)

明日日期为本年下月 1 日。

在上述判断过程中,程序在进行函数调用时,将整个结构变量作为参数传递给被调函数。在主函数 `main()` 中采用条件语句 `if(today.day != numdays(today))` 调用了函数 `numdays()`,实参是结构类型 `ydate` 的结构变量 `today`。而在函数 `numdays()` 中又调用了函数 `leapyear()`,实参是具有结构类型的结构变量 `d`。

与一般变量相同,将整个结构变量作为参数传递的过程是值的传递,函数调用时,系统为形式参数的结构变量分配存储空间,并从相应的实际参数中取得各个元素的值。函数对形参中各个结构元素值进行的修改不会对相应的实参结构变量产生任何影响。

5.5.2 将结构型指针作为函数的参数

当一个结构较大时,若将该结构作为函数的参数,由于参数传递采用值传递方式,需要较大的存储空间(堆栈)来将所有的结构元素压栈和出栈,尤其当函数参数是结构数组时,影响更大。此外还会影响程序的执行速度。实际上可以用结构型指针来作为函数的参数,此时参数的传递是按地址传递方式进行的。由于采用的是地址传递方式,只需要传递一个地址值,与前者相比,既可节省存储空间,同时还可加快程序的执行速度。其缺点是在调用函数时对结构指针所作的任何变动都会影响到原来的结构变量。

例 5.6: 用结构指针作为函数的参数。有 4 个测量参数,每个测量参数中包括测量次数、参数名以及测量值,要求找出测量值最大的那个参数名及其测量值。

```
#include <stdio.h>
main() {
    struct mepoint {
        unsigned char num;
        unsigned char name[11];
        float score;
    } mp[4]={
        {1, "1stpoint", 368.78},
        {2, "2ndpoint", 134.90},
        {3, "3tdpoint", 798.90},
        {4, "4thpoint", 436.01}
    };
    struct mepoint *p;
    unsigned char i,temp=0;
    float max;
    for (max=mp[0].score, i=1;i<4;i++)
        if (mp[i].score>max) {
            max=mp[i].score; temp=i;
        }
    p=mp+temp;
    printf("\nThe maxmum score: \n");
    printf("No.:   %bd\nname: %s\nscore: %4.2f\n",
           p->num,p->name,p->score);
    while(1);
}
```

程序执行结果:

```
The maxmum score:
No.:   3
name:  3tdpoint
score: 798.90
```

在这个程序中, 定义 `mp[4]` 为 `struct mepoint` 结构类型的数组, `*p` 为指向该结构类型数据的指针变量。在 `for` 循环语句中先使变量 `max` 的值等于 `mp[0].score` (即第一个参数的测量值), 然后将 `max` 与其余 3 个参数的测量值 (即 `mp[i].score`) 进行比较, 若后者大于 `max`, 则将较大者放入 `max` 中, 并记下此时的 `i` 值 (放入 `temp` 中保存)。接着使指针 `p=mp+temp`, 也就是将指针 `*p` 定位于测量值最高的那个数组元素处, 最后输出指针 `*p` 所指向的那个数组元素中各个成员的值。由此可见, 用结构型指针作为函数的参数传递, 只需传递一个结构变量的地址即可, 而不必将结构变量的全部元素值一个一个地传递, 既节省时间又节省空间。尤其当结构变量中的成员较多, 或者有一些成员是数组时, 用指针作参数传递, 能大大提高程序的运行效率。

5.6 联合变量的定义与引用

联合也是 C 语言中一种构造类型的数据结构。在一个联合中可以包含多个不同类型的数据元素, 例如, 可以将一个 `float` 型变量、一个 `int` 型变量、一个 `char` 型变量放在同一个地址开始的内存单元中, 如图 5.3 所示。以上三个变量在内存中的字节数不同, 但却都从同一个地址开始存放, 即采用了所谓的“覆盖技术”。这种技术可使不同的变量分时使用同一个内存空间, 提高内存的利用效率。

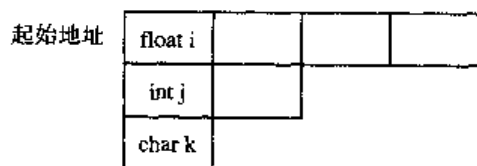


图 5.3 联合中变量的存储方法

联合类型变量的一般定义格式为:

union 联合类型名

{成员表列} 变量表列;

例如:

```
union data {  
    float i;  
    int j;  
    char k;  
}a,b,c;
```

也可以将类型定义与变量定义分开, 即先定义一个 `union data` 类型, 再将 `a`、`b`、`c` 定义为 `union data` 类型的变量。例如:

```
union data {  
    float i;  
    int j;  
    char k;
```

```
};  
union data a,b,c;
```

当然还可以直接定义联合变量，例如：

```
union {  
    float i;  
    int j;  
    char k;  
}a,b,c;
```

由此可见，联合类型与结构类型的定义方法是很相似的，只是将关键字 `struct` 改成了 `union`。但是在内存的分配上它们二者却有着本质的区别。结构变量所占用的内存长度是其中各个元素所占用内存长度的总和；而联合变量所占用的内存长度是其中最长的元素的长度。对于上面的例子，`float` 型数据需要 4 个内存单元，因此系统按 4 个字节长度为联合变量 `a`、`b`、`c` 分配内存空间，在不同时刻只能保存一个变量，对一个变量进行操作。如果将 `a`、`b`、`c` 定义成结构类型变量，则系统将按 7 字节个（`float` 型数据占 4 个字节，`int` 型数据占 2 个字节，`char` 型数据占 1 个字节）给变量 `a`、`b`、`c` 分配内存空间，在不同时刻可同时操作三个变量。

与结构变量类似，对联合变量的引用也是通过对其联合元素的引用来实现的。引用联合元素的一般格式为：

联合变量名.联合元素

或

联合变量名->联合元素

例如，对于前面定义的联合变量 `a`、`b`、`c`，下面的引用方法都是正确的：

```
a.i /* 引用联合变量 a 中 float 型元素 i */  
b.j /* 引用联合变量 b 中 int 型元素 j */  
c.k /* 引用联合变量 c 中 char 型元素 k */
```

在引用联合元素时，要注意联合变量用法的一致性。因为联合类型中定义的各个不同类型的元素都可以分时地赋给变量，而所读取变量的值是最近放入的某一元素的值，因此在表达式中对它进行处理时，必须注意其类型与表达式所要求的类型保持一致，否则将导致程序运行出错。不能只引用联合变量，例如，下面的写法就是错误的：

```
printf("%f",a);
```

因为变量 `a` 可能是 `float`、`int` 和 `char` 三种类型，分别占用不同长度的内存区域，若在引用时仅写联合变量名 `a`，系统将难以确定究竟应该输出哪一个联合元素的值。正确的写法应为：

```
printf("%f",a.i);
```

联合类型的数据可以采用同一个内存段来存放几种不同类型元素的值，但是在每一瞬时只能存放其中一种类型的元素，而不能同时存放几种。换句话说，每一瞬时只有一个元

素在起作用。起作用的是联合中最后一次存放的元素，如果存入了一个新的元素，则上次存入的元素就失去作用。例如，对于下列语句：

```
a.i=100.05;
a.j=100;
a.k=10;
```

在执行以上三条赋值语句之后，只有 a.k 是有效的，而 a.i 和 a.j 都已失去作用。因此在引用联合变量时一定要十分注意当前在联合变量中存放的究竟是哪一个元素。不能直接对联合变量进行赋值，也不能在定义联合变量时对它进行初始化。联合变量不能作为函数的参数，但可以使用指向联合变量的指针。

联合可以出现在结构和数组中，结构和数组也可以出现在联合中。当需要存取结构中的联合或联合中的结构时，其存取方法与存取嵌套的结构相同。

前面在 2.2 节中已经讲过，Keil Cx51 编译器支持 IEEE-754 标准（32 位）的单精度二进制浮点数及其存储格式，这种浮点数在内存中占用 4 个字节（32 位二进制数），在内存中的存放格式如下：

字节地址	+0	+1	+2	+3
浮点数内容	S EEEEEEE	E MMMMMMM	MMMMMMMM	MMMMMMMM

一个浮点数的正常数值范围是： $(-1)^S \times 2^{E-127} \times (1.M)$ ，其中， $E=1 \sim 254$ ； $S=\pm 1$ 。超过最大正常数的浮点数就认为是无穷大，其阶码 E 为全 1，小数部分 M 为全 0，表示为：

$$\pm \infty = (-1)^S \times 2^{128} \times (1.000...000) = \pm 2^{128}$$

对于阶码 E 为全 0，小数部分 M 也为全 0 的浮点数认为是 0，表示为：

$$(-1)^S \times 2^{-127} \times (1.000...000) = \pm 2^{-127}$$

绝对值最小的正常浮点数为阶码 E 为 1，小数部分 M 为全 0 的数，表示为：

$$(-1)^S \times 2^{-126} \times (1.000...000) = \pm 2^{-126}$$

除了正常数之外，介于 $+2^{-126} \sim +2^{-127}$ 以及 $-2^{-126} \sim -2^{-127}$ 之间的数为非正常数，它们可表示为阶码 E 为全 0，小数部分 M 为非 0 的数，但尾数的整数部分不再是正常数所规定的 1，而必须是 0。因此有：

$$\text{最大非正常数} = (-1)^S \times 2^{-126} \times (0.111...111)$$

$$\text{最小非正常数} = (-1)^S \times 2^{-126} \times (0.000...000) = (-1)^S \times 2^{-149}$$

按 IEEE 标准，浮点数的数值如果在正常数值之外，即为溢出错误。IEEE 标准可能出错的条件用下面的二进制数表示：

非正常数：NaN=0FFFFFFFH

正无穷：+INF=7F800000H

负无穷：-INF=FF800000H

由于 8051 系列单片机中没有捕获浮点数溢出错误的中断功能，因此用户在编程时必须根据浮点数的出错条件作出适当的反应。下面是一个采用联合类型处理浮点数运算错误的例子。程序中定义了包含有一个 float 类型和一个 unsigned long 类型数据的联合，用来保存浮点数的值。程序运行时先从键盘输入两个浮点数 a 和 b，然后对这两个浮点数进行乘法运算，如果未产生浮点数溢出，则输出正确的计算结果，如果产生浮点溢出错误，即

根据 IEEE 标准的出错条件输出相应的提示。

例 5.7: 采用联合类型处理浮点数溢出错误。

```
#include <stdio.h>
#define NaN    0xffffffff
#define plusINF 0x7f800000
#define minusINF 0xf80f0000
union f {
    float    f;
    unsigned long    ul;
};
union f x;
main() {
    float a,b;
1: printf("please input a, b:\n");
    scanf("%f %f",&a,&b);
    x.f=a*b;
    if (x.ul==NaN) {
        printf("NaN error! \n"); goto 1;
    }
    else
    if (x.ul==plusINF) {
        printf("+INF error! \n"); goto 1;
    }
    else
    if (x.ul==minusINF) {
        printf("-INF error! \n"); goto 1;
    }
    else
        printf("OK! x.f=%e",x.f);printf("\n");
    while(1);
}
```

程序运行结果:

```
please input a, b:
1.40e-45 1 回车
NaN error!
please input a, b:
3.41e39 1 回车
+INF error!
please input a, b:
-3.41e39 1 回车
-INF error!
```

```
please input a, b:
1.7e38    1 回车
OK! x.f=1.700000e+38
```

从这个例子可以看到,联合类型提供了一种能够在—个单独的存储空间内处理不同类型的数据的方法,因此适当使用联合类型可以节约内存空间。

例 5.8: 采用联合类型读取不同类型的数据。

```
#include <stdio.h>
union f {
    unsigned char date[7];
    struct
    {
        unsigned char year1, year2, month1, month2, day1, day2;
    } xdate;
} udate;
main() {
    unsigned char i;
    udate.date[0]=9, udate.date[1]=6;
    udate.date[2]=1, udate.date[3]=2;
    udate.date[4]=3, udate.date[5]=1;
    printf("The date is: ");
    for(i=0; i<6; i=i+2) printf("%bx%bx-", udate.date[i], udate.date[i+1]);
    printf("\n");
    printf("The year is: ");
    printf("%bx%bx", udate.xdate.year1, udate.xdate.year2); printf("\n");
    printf("The month is: ");
    printf("%bx%bx", udate.xdate.month1, udate.xdate.month2); printf("\n");
    printf("The day is: ");
    printf("%bx%bx", udate.xdate.day1, udate.xdate.day2); printf("\n");
    while(1);
}
```

程序执行结果:

```
The date is:  96-12-31
The year is:  96
The month is: 12
The day is:   31
```

在这个例子中,对于联合变量 `udate`,可以通过它的元素(数组) `date[]` 获得一个完整的日期值,也可以利用它的另一个元素(结构) `xadte` 分别取得年 `year`、月 `month`、日 `day` 的值。

5.7 枚举变量的定义与引用

ANSI C 新标准中增加了枚举类型, 如果一个变量只有几种可能的值, 可以将其定义为枚举类型, 所谓“枚举”就是将变量的值一一列举出来。

定义枚举类型变量的一般格式:

enum 枚举名{枚举值表列} 变量表列;

也可以将枚举的定义和说明分写成两句:

enum 枚举名{枚举值表列};

enum 枚举名 变量表列;

例如:

```
enum day{Sun, Mon, Tue, Wed, Thu, Fri, Sat} d1, d2;
```

或

```
enum day{Sun, Mon, Tue, Wed, Thu, Fri, Sat};  
enum day d1, d2;
```

只有在建立了枚举类型的原型 `enum day`, 将枚举名与枚举值表列联系起来, 并进一步说明该原型的具体变量 `enum day d1,d2` 之后, Cx51 编译器才会给 `d1,d2` 分配存储器空间, 这些变量才可以具有所与定义的相应枚举表列中的值。

在枚举表列中, 每一项符号代表一个整数值。在默认情况下, 第一项取值为 0, 第二项取值为 1, 第三项取值为 2, 依次类推。此外也可以通过初始化指定某些项的符号值。某项符号值初始化以后, 该项后续各项符号值依次递增, 例如:

```
enum direct{up, down, left=10, right};
```

Cx51 编译器将给 `up` 赋值为 0, 将 `down` 赋值为 1, 由于 `left` 被初始化为 10, 故 `right` 的值将为 11。

例 5.9: 将若干个红、绿、黄三种颜色的小球作全排列组合, 输出每种组合的三种颜色。

```
#include<stdio.h>  
#include<reg51.h>  
main() {  
    enum color{red, green, blue} i,j,k,st;  
    int n = 0, lp;  
    for (i=red; i<=blue; i++)  
        for (j=red; j<=blue; j++)  
            for (k=red; k<=blue; k++) {  
                n=n+1;  
                printf("%-4d",n);  
                for (lp=1;lp<=3; lp++) {  
                    switch(lp) {
```

```

        case 1: st=i; break;
        case 2: st=j; break;
        case 3: st=k; break;
        default: break;
    }
    switch(st) {
        case red:  printf("%-10s", "red"); break;
        case green: printf("%-10s", "green"); break;
        case blue:  printf("%-10s", "blue"); break;
        default:    break;
    }
    }
    printf("\n");
}
printf("\n total: %5d\n", n);
while(1);
}

```

程序执行结果:

1	red	red	red
2	red	red	green
3	red	red	blue
4	red	green	red
5	red	green	green
6	red	green	blue
7	red	blue	red
8	red	blue	green
9	red	blue	blue
10	green	red	red
11	green	red	green
12	green	red	blue
13	green	green	red
14	green	green	green
15	green	green	blue
16	green	blue	red
17	green	blue	green
18	green	blue	blue
19	blue	red	red
20	blue	red	green
21	blue	red	blue

22	blue	green	red
23	blue	green	green
24	blue	green	blue
25	blue	blue	red
26	blue	blue	green
27	blue	blue	blue

total: 27

第 6 章 预处理器

C 语言与其他高级程序设计语言的一个主要区别就是对程序的编译预处理功能，编译预处理器是 C 语言编译器的一个组成部分。在 C 语言中，通过一些预处理命令可以在很大程度上为 C 语言本身提供许多功能和符号等方面的扩充，增强了 C 语言的灵活性和方便性。预处理命令可以在编写程序时加在需要的地方，但它只在程序编译时起作用，且通常是按行进行处理的，因此又称为编译控制行。C 语言的预处理命令类似于汇编语言中的伪指令。编译器在对整个程序进行编译之前，先对程序中的编译控制行进行预处理，然后再将预处理的结果与整个 C 语言源程序一起进行编译，以产生目标代码。Keil Cx51 编译器的预处理器支持所有满足 ANSI 标准 X3J11 细则的预处理命令。常用的预处理命令有：宏定义、文件包含和条件编译命令。为了与一般 C 语言语句相区别，预处理命令由符号“#”开头。

6.1 宏定义

宏定义命令为 `#define`，它的作用是用一个字符串来进行替换，而这个字符串既可以是常数，也可以是其他任何字符串，甚至还可以是带参数的宏。宏定义的简单形式是符号常量定义，复杂形式是带参数的宏定义。

6.1.1 不带参数的宏定义

不带参数的宏定义又称符号常量定义，一般格式为：

`#define 标识符 常量表达式`

其中，“标识符”是所定义的宏符号名（也称宏名）。它的作用是在程序中使用所指定的标识符来代替所指定的常量表达式。实际上在前面的章节中我们已经见过这种用法，例如：`#define NaN 0xFFFFFFFF` 就是用 NaN 这个符号来代替常数 `0xFFFFFFFF`。使用了这个宏定义之后，程序中就不必每次都写出常数 `0xFFFFFFFF`，而可以用符号 NaN 来代替。在编译时，编译器会自动将程序中所有的符号名 NaN 都替换成常数 `0xFFFFFFFF`。这种方法使得可以在 C 语言源程序中用一个简单的符号名来替换一个很长的字符串。还可以使用一些有一定意义的标识符，提高程序的可读性。例如，在第 5 章例 5.7 程序中使用了如下的宏定义：

```
#define NaN      0xffffffff    /* 定义非正常数出错条件 */
#define plusINF  0x0000807f    /* 定义正无穷出错条件 */
#define minusINF 0x000080ff    /* 定义负无穷出错条件 */
```

采用这些定义可以使人对程序中这些常数的意义一目了然，有助于构造程序和整理程序文本。通常程序中的所有符号定义都集中放在程序的开始处，便于检查和修改，提高程序的可靠性。另外如果需要修改程序中的某个常量，可以不必修改整个程序，而只要修改一下相应的符号常量定义行即可。

在实际使用宏定义时，按一般习惯，通常将宏符号名用大写字母表示，以区别于其他的变量名。宏定义不是 C 语言的语句，因此在宏定义行的末尾不要加分号，否则在编译时将连同分号一起进行替换而导致出现语法错误。在进行宏定义时，可以引用已经定义过的宏符号名，即可以进行层层代换，但最多不能超过 8 级嵌套。需要注意的是预处理命令对于程序中用双引号括起来的字符串内的字符，即使该字符与宏符号名相同也不作替换。

宏符号名的有效范围是从宏定义命令 `#define` 开始，直到本源文件结束。通常将宏定义命令 `#define` 写在源程序的开头，函数的外面，作为源文件的一部分，从而在整个文件范围内有效。需要时可以用命令 `#undef` 来终止宏定义的作用域。

例 6.1：符号常量定义的应用。

```
#include <stdio.h>
#define PI 3.141592
#define R 3.0
#define L 2*PI*R
#define S PI*R*R
main() {
    printf("L=%f\nS=%f\n",L,S);
    while(1);
}
```

程序执行结果：

```
L=18.849560
S=28.274330
```

程序中 `printf` 函数中的输出项 `L` 经过宏展开后分别被替换成为 `2*3.141592*3.0`，输出项 `S` 被替换成为 `3.141592*3.0*3.0`。对于 `printf` 函数中的输出控制部分的字符 `L` 和 `S`，虽然它们与前面定义的宏符号 `L` 和 `S` 同名，但由于它们位于双引号之内，因此不被替换。即 `printf` 函数经过替换之后成为：

```
printf("L=%f\nS=%f\n",2*3.141592*3.0, 3.141592*3.0*3.0);
```

在 Keil Cx51 编译器中已经预先定义了下列宏符号名（注意首尾都有双下划线）：

`__C51__`：Keil Cx51 编译器的版本号，如 550 为 5.50 版

`__DATE__`：编译开始日期

`__FILE__`：被编译的文件名

`__LINE__`：被编译文件的当前行号

`__MODEL__`：编译时所选择的存储器模式，0 为 SMALL，1 为 COMPACT，2 为 LARGE

`__STDC__`：始终为“1”

`__TIME__`：编译开始时间

在使用符号常量定义时要注意,这些预定义宏符号名不能用`#define`和`#undef`命令进行重复定义。

6.1.2 带参数的宏定义

带参数的宏定义与符号常量定义的不同之处在于,对于源程序中出现的宏符号名不仅进行字符串替换,而且还进行参数替换。带参数宏定义的一般格式为:

`#define` 宏符号名(参数表) 表达式

其中,表达式内包含了在括号中所指定的参数,这些参数称为形式参数,在以后的程序中它们将被实际参数所替换。带参数的宏定义将一个带形式参数的表达式定义为一个带形式参数表的宏符号名,对程序中所有带实际参数表的该宏符号名,用指定的表达式来替换,同时用参数表中的实际参数替换表达式中对应的形式参数。下面通过一些实例来说明带参数的宏定义的法。

带参数的宏定义常用来代表一些简短的表达式,它用来将直接插入的代码代替函数调用,从而提高程序的执行效率。例如:

```
#define MIN(x,y) ((x)<(y))?(x):(y)
```

定义了一个带参数的宏 `MIN(x,y)`,以后在程序中就可以用这个宏而不用函数 `min()`。语句 `m=MIN(u,v)`; 经宏展开后成为: `m=((u)<(v))?(u):(v)`;

带参数的宏定义可以引用已定义过的宏定义,即宏定义的嵌套(最多不超过 8 级)。例如:

```
#define SQ(x) (x*x)
#define CUBE(x) (SQ(x)*x)
#define FIFTH(x) (CUBE(x)*SQ(x))
```

语句 `y=FIFTH(a)`; 经宏展开后成为: `y=((((a*a)*a)*(a*a))`;

带参数的宏定义在进行宏展开时,只是用语句中宏符号名后面括号内的实际参数字符串来替换`#define`命令行中的形式参数。因此对于宏展开后容易引起误解的表达式,在进行宏定义时,应将该表达式用圆括号括起来。例如:

```
#define S(r) PI*r*r
```

对于语句 `area=S(a)`; 经宏展开后成为 `area=PI*a*a`; 这时没有问题。但是对于如下语句: `area=S(a+b)`; 经宏展开后成为 `area=PI*a+b*a+b`; 而程序设计者的原意是希望在展开后得到 `area=PI*(a+b)*(a+b)`; 为此应按如下方式进行宏定义:

```
#define S(r) PI*(r)*(r)
```

进行宏定义时,宏符号名与带参数的圆括号之间不能存在空格,否则,在宏展开时会将空格以后的所有字符作为实际字符串对前面的宏名进行替换。例如:

```
#define SQ (x) (x*x)
```

语句 `y=SQ(5)`; 经宏展开后成为 `y=(x) (x*x)(5)`; 这显然是一个错误的语句,其原因

就在于 SQ 与(x)之间存在空格,宏定义#define 将它们简单地作为符号常量定义,即误认为 SQ 代表(x) (x*x),所以展开后得出上述错误的语句。

宏定义命令#define 要求在一行内写完,若一行之内写不下时需用“\”表示下一行继续。例如:

```
#define PR(a,b) printf("%d\t%d\n", \
    (a)>(b)? (a):(b), (a)<(b)? (b):(a))
```

在宏定义中可利用符号“#”将实际参数转换为一个字符串。例如:

```
#define stringer(x) printf("#\n")
```

语句 stringer(text); 经宏展开后成为 printf("text\n");

在宏定义中可利用符号“##”可将两个变量合并。例如:

```
#define paster(n) printf("token#n"=%d",token##n)
```

语句 paster(9); 经宏展开后成为 printf("token9=%d",token9); 这里同时使用了符号“#”和符号“##”。

利用带参数的宏定义可以省去在程序中重复书写相同的程序段,实现程序的简化。

例 6.2: 带参数宏定义的应用。

```
#include <stdio.h>
#define PI 3.141592
#define CIRCLE(R,L,S) L=2*PI*R; S=PI*(R)*(R)
main() {
    float r,l,s;
    printf("Please input r: \n");
    scanf("%f", &r);
    CIRCLE(r,l,s);
    printf("r=%f\ncirc=%f\narea=%f\n",r,l,s);
    while(1);
}
```

程序执行结果:

```
    Please input r:
    2.5 回车
    r=2.500000
    circ=15.707960
    area=19.634950
```

如果善于利用宏定义,可以实现程序的简化。下面的例子中,先利用宏定义将输出格式定义好,以减少在输出语句中每次都要写出具体的输出格式的麻烦。

例 6.3: 利用宏定义简化输出语句(程序名为 DEMO.C)。

```
#include <stdio.h>
#define PR printf
```

```
#define NL "\n"
#define D " %d"
#define D1 D NL
#define D2 D D NL
#define D3 D D D NL
#define D4 D D D D NL
#define S " %s"
main() {
    unsigned int a,b,c,d;
    unsigned char string[]="MICROCONTROLLER";
    a=1; b=2; c=3; d=4;
    PR(D1, a);
    PR(D2, a, b);
    PR(D3, a, b, c);
    PR(D4, a, b, c, d);
    PR(S, string);PR(NL);
    PR(" %s",_ _FILE_ _);PR(NL);
    PR(" %s",_ _TIME_ _);PR(NL);
    PR(" %s",_ _DATE_ _);PR(NL);
    PR(" %bd",_ _STDC_ _);PR(NL); ;
    PR(" %d",_ _C51_ _);PR(NL); ;
    PR(" %bd",_ _MODEL_ _);PR(NL); ;
    PR(" %bd",_ _LINE_ _);PR(NL); ;
    while(1);
}
```

程序执行结果:

```
1
1 2
1 2 3
1 2 3 4
MICROCONTROLLER
DEMO.C
10:20:53
jun 18 2003
1
700
0
250
```

实际应用中可以参照上例利用宏定义写出各种输入输出格式,把它们单独编成一个文件,形成一个“格式库”,需要时可用文件包含命令#include 将其包含到所编写的程序中去。

代码, 而忽略掉程序段 2, 否则程序段 2 参加编译并产生有效代码而忽略掉程序段 1。其中 `#else` 和程序段 2 可以没有。这里的程序段可以是 C 语言的语句组, 也可以是命令行。这种条件编译对于提高 C 语言源程序的通用性是很有好处的。例如, 对于工作于 6MHz 和 12MHz 时钟频率下的 8051 和 8052 单片机, 可以采用如下的条件编译使编写的程序具有通用性:

```
#define CPU 8051
#ifdef CPU
#define FREQ 6
#else
#define FREQ 12
#endif
```

这样后面的源程序不作任何修改就可以适用于两种时钟频率的单片机系统。当然还可以仿此设计出其他多种条件编译。

条件编译命令格式二:

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

该命令格式与第一种命令格式只在第一行上不同, 它的作用与第一种刚好相反, 即: 如果指定的标识符未被定义, 则程序段 1 参加编译并产生有效代码, 而忽略掉程序段 2, 否则程序段 2 参加编译并产生有效代码而忽略掉程序段 1。

以上两种格式的用法也很相似, 可根据实际情况视需要而定。例如, 对于上面的例子也可以采用如下的条件编译:

```
#define CPU 8052
#ifndef CPU
#define FREQ 12
#else
#define FREQ 6
#endif
```

其效果是完全一样的。

条件编译命令格式三:

```
#if 常量表达式 1
    程序段 1
#elif 常量表达式 2
    程序段 2
...
#elif 常量表达式 n-1
```

程序段 n-1

#else

程序段n

#endif

这种格式条件编译的功能是：如果常量表达式 1 的值为真（非 0），则程序段 1 参加编译，然后将控制传递给匹配的 #endif 命令，结束本次条件编译，继续下面的编译处理。否则，如果常量表达式 1 的值为假（0），则忽略掉程序段 1（不参加编译）而将控制传递给下面的一个 #elif 命令，对常量表达式 2 的值进行判断。如果常量表达式 2 的值为假（0），则将控制再传递给下一个 #elif 命令。如此进行，直到遇到 #else 或 #endif 命令为止。使用这种条件编译格式可以事先给定某一条件，使程序在不同的条件下完成不同的功能。

例 6.4：将一个字符串数组中的各个字母按所设置的条件，或者按大写输出，或者按小写输出。

```
#include <stdio.h>
#define CONDITION 1
main() {
    char str[]="C Programing", c;
    int i=0;
    while((c=str[i])!='\0') {
        i++;
        #if CONDITION
            if(c>='a'&&c<='z')
                c=c-32;
        #else
            if(c>='A'&&c<='Z')
                c=c+32;
        #endif
        printf("%c",c);
    }
}
```

程序执行结果：

C PROGRAMING

由于程序第一行宏定义命令 #define CONDITION 1 的作用，在程序编译过程中执行条件编译命令时，判断 CONDITION 为真，故将对下述程序段：

```
if(c>='a'&&c<='z')
    c=c-32;
```

进行编译并产生可执行代码。由于大写字母比与其相应的小写字母的 ASCII 码值小 32，若遇到小写字母时只要将其 ASCII 码值减去 32，就可以变成大写字母输出。同样地，如果要按小写字母输出，只需要将程序第一行的宏定义改成：

```
#define CONDITION 0
```

则在预处理时，将对下述程序段

```
if (c>='A'&& c<='Z')  
    c=c+32;
```

进行编译，运行程序时就会将大写字母变成小写字母输出。

6.4 其他预处理命令

除了上面介绍的宏定义、文件包含和条件编译预处理命令之外，Cx51 编译器还支持 **#error**、**#pragma** 和 **#line** 预处理命令。**#line** 命令一般很少使用，下面介绍 **#error** 和 **#pragma** 命令的功能和使用方法。

#error 命令通常嵌入在条件编译之中，以便捕捉到一些不可预料的编译条件。正常情况下该条件的值应为假，若条件的值为真，则输出一条由 **#error** 命令后面的字符串所给出的错误信息并停止编译。例如，如果有 **#define MYVAL**，它的值必须为 0 或 1，为了测试 **MYVAL** 的值是否正确，可在程序中安排如下一段条件编译：

```
#if (MYVAL!=0 && MYVAL!=1)  
#error MYVAL must be defined to either 0 or 1  
#endif
```

当 **MYVAL** 的值出错时，将输出出错信息并停止编译。

#pragma 命令通常用在源程序中向编译器传送各种编译控制命令，其使用格式为：

#pragma 编译命令名序列

例如，对例 6.4 的程序进行编译时希望采用 **DEBUG**、**CODE**、**LARGE** 编译命令，则只要在源程序的开始处加入一个命令行：**#pragma DB CD LA** 即可。

#pragma 命令可以出现在 C 语言源程序中的任何一行，从而使编译器能重复执行某些编译控制命令以达到某种特殊的目的。如果 **#pragma** 命令后面的参数不是 Keil Cx51 编译器的合法译控制命令，编译器将忽略其作用。需要指出的是，并非所有的 C51 编译控制命令都可以在 C 语言源程序中采用 **#pragma** 预处理命令多次使用，对于 Keil Cx51 编译器的首要控制命令，只能使用一次，如果多次使用将导致致命的编译错误。

第7章 μ Vision2 集成开发环境

Keil 公司于 1999 年推出了一种全新的集成开发环境 μ Vision2，它是一种 32 位标准的 Windows 应用程序，支持长文件名操作，其界面类似于 MS Visual C++，可以在 Windows 95/98/2000/XP 平台上运行，功能十分强大。 μ Vision2 中包含了源程序文件编辑器，项目管理器 (Project)、源程序调试器 (Debug) 等，并且为 Cx51 编译器、Ax51 汇编器、BL51/Lx51 连接定位器、RTX51 实时操作系统等提供了单一而灵活的开发环境。 μ Vision2 的源级浏览器功能利用符号数据库使用户可以快速浏览源文件，用户可通过详细的符号信息来优化变量存储器；文件查找功能可在指定的若干种文件中进行全局文件搜索；工具菜单功能允许启动指定的用户应用程序， μ Vision2 还提供了对第三方工具软件的接口。

μ Vision2 具有强大的项目管理功能，一个项目由源程序文件、开发工具选项以及编程说明三部分组成，通过目标创建 (Build Target) 选项很容易实现对一个 μ Vision2 项目进行完整的编译和连接，直接产生最终应用目标程序。

μ Vision2 中包含一个器件数据库 (device database)，数据库中有各种单片机片上存储器和外围集成功能资源信息。在项目开发过程中通过数据库选定一种单片机之后， μ Vision2 会自动设置默认的 Ax51 汇编器、Cx51 编译器、BL51/Lx51 连接定位器及 Debug 调试器选项。此外用户可以根据不同需要手工设置各种编译、连接和调试选项。

μ Vision2 中集成的 Debug 调试器具有十分强大的仿真调试功能，支持软件模拟和用户目标板调试两种工作方式。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下载到自己的 8051 单片机系统板上，利用 8051 的串行口与 PC 机进行通信实现用户程序的实时在线仿真，这种方式使用户可以避免购买昂贵的硬件仿真器而达到相同的仿真效果，最大限度地保护了用户的利益。

在 Windows 中安装了 Keil 公司的 PK51 软件包之后，会自动在桌面和开始菜单中生成一个“Keil μ Vision2”图标，双击桌面上的“Keil μ Vision2”图标即可启动运行，也可以单击“开始”按钮，将鼠标指向“程序”，找到“Keil μ Vision2”图标并单击鼠标左键启动运行，启动运行后将显示如图 7.1 所示 μ Vision2 提示信息，几秒钟后提示信息自动消失，出现如图 7.2 所示主窗口。主窗口由标题



图 7.1 μ Vision2 启动提示信息

栏、下拉菜单、快捷工具条按钮、项目窗口、文件编辑窗口、输出窗口以及状态栏等组成。

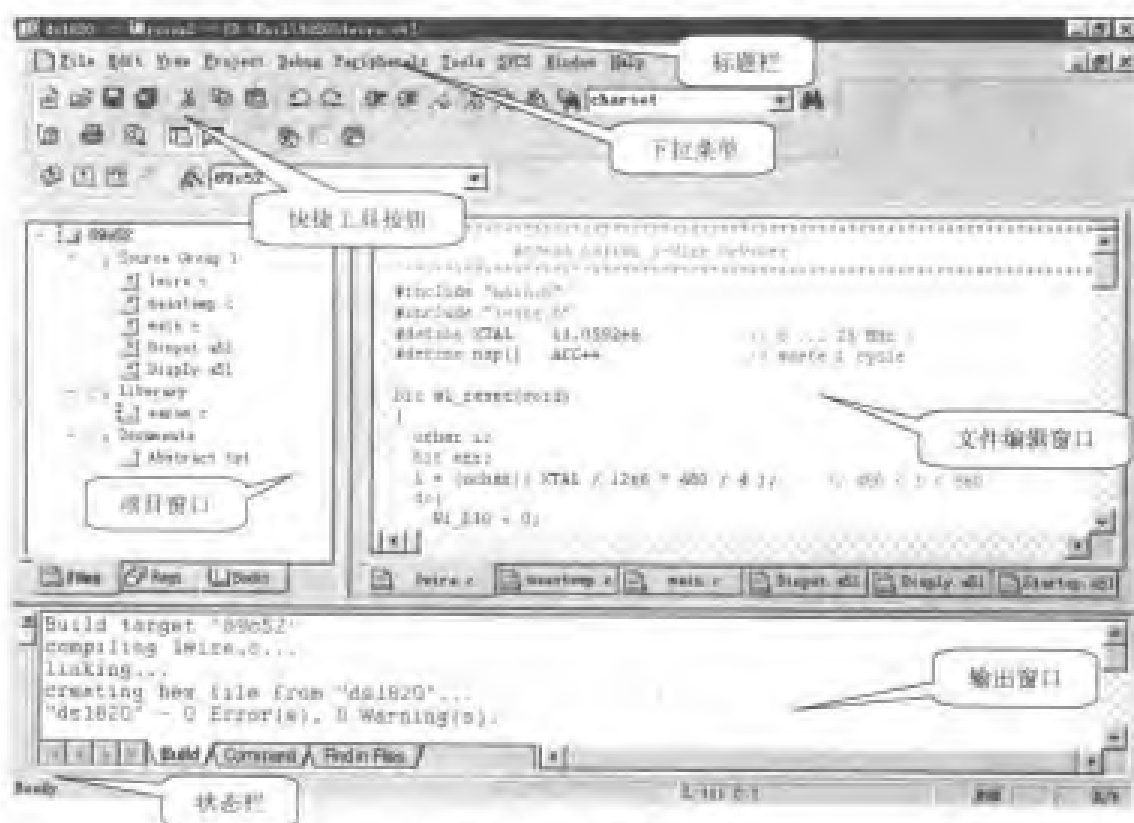


图 7.2 μVision2 的窗口分配

标题栏用于显示应用程序名和当前打开的文件名。下拉菜单提供多种选项，可以根据不同需要选用。快捷工具条按钮分为三组：文件工具按钮（如图 7.3 所示）、调试工具按钮（如图 7.4 所示）、编译选项工具按钮（如图 7.5 所示），它们实际上是下拉菜单中一些重要选项的快捷方式，将鼠标放在某个工具按钮上稍作停留，屏幕上会自动显示该按钮的功能名称，很多时候使用工具按钮要比下拉菜单方便。



图 7.3 μVision2 的文件工具按钮



图 7.4 μVision2 的调试工具按钮





图 7.5 μVision2 的编译选项工具按钮


项目窗口用于显示当前打开项目的有关信息，包括三个标签页：Files（文件）、Regs（寄存器）和 Books（参考书），其中 Regs 标签页只有在调试状态才有效。Books 标签页中


显示 Cx51 软件包附带的各种在线参考书籍和帮助信息,用鼠标左键双击其中某个文件名,可立即将其打开进行浏览。


启动 μ Vision2 时项目窗口自动进入 Files 标签页,用鼠标左键双击其中一个文件名,将立即在文件编辑窗口中打开该文件。Files 标签页内用不同图标来表示文件的属性,下面是这些图标及其对应属性的描述。

 文件图标,表示不被编译、链接到项目中去文件,如文档文件等。

 此图标是在文件图标上加一箭头而成,表示被编译、链接到项目中去文件。

 图标上有一把钥匙,表示只读文件,该文件不能被编辑修改。

 在图标左边显示有三个小点,表示用户已经在该文件或文件组中设置了某些特殊要求的选项。

 文件组图标,当文件组中已经添加了文件时,该图标左边将出现一个“+”号,单击“+”号可以展开该文件组。

利用图标能够快速浏览一个项目中不同目标(Target)的选项设置,窗口中显示的图标总是反映当前所选目标的属性。例如,在一个目标中对一个文件或文件组设置了特殊选项,那么只有当选择了此目标时,那些设置了特殊选项的文件图标上才会出现三个点,换句话说,在一个目标中某个文件图标上有箭头或三个点,在另一个目标中并不一定如此。只有只读属性是属于文件自己的,一个文件具有只读属性,那它在任何目标中都是如此。

文件编辑窗口用于对当前打开的文件进行编辑, μ Vision2 提供了一种多功能的文件操作环境,其中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,可以用鼠标操作,也可以用快捷键操作,编辑器已经包含有许多预定义快捷键,用户也可以根据自己的爱好对快捷键重新定义。在 μ Vision2 文件编辑窗口中可以同时打开多个不同类型的文件分别进行处理,这一点对结构化多模块程序设计特别方便。编辑器还提供一种可选的彩色语句显示功能,对于 C51 程序中的变量、关键字、语句等采用不同颜色显示,提高了程序的可读性。

输出窗口中有三个标签页:Build(创建)标签页用于显示对当前打开的项目或文件进行编译链接的提示信息,当编译链接出错时,用鼠标左键双击该窗口中某个错误提示信息,光标将自动跳转到文件编辑窗口中对应文件发生错误的地方,以利于用户分析错误原因和修改源文件;Command(命令)标签页用于在调试状态下输入各种调试命令以及观察命令的执行结果;Find in Files(文件查找)选项用于显示多文件查找结果。

状态栏用于显示对各种快捷工具按钮的简要说明以及文件编辑窗口中当前光标所在的行号、列号等信息。

7.1 μ Vision2 的下拉菜单

μ Vision2 提供下拉菜单和快捷工具按钮两种操作方式。下拉菜单中有多种选项,根据不同需要选用。下拉菜单中各个选项若能够采用快捷键操作,则在该选项右边列出了对应的快捷键,若能够采用工具条按钮操作,则在该选项左边列出了一个按钮图标,使用熟练后可以不用下拉菜单而直接采用快捷键,或通过 μ Vision2 主窗口中的工具条按钮进行操作。

下面先从 μ Vision2 的下拉菜单着手, 对它的具体功能作详细介绍。

7.1.1 File 菜单

File 菜单如图 7.6 所示, 分为 5 栏。第一栏为文件操作, 用鼠标左键单击其中的“New”选项, 可创建一个新文件, 也可用快捷键“Ctrl+N”来创建新文件。对于已有的文件可用



图 7.6 File 菜单

“Open...”选项(快捷键为“Ctrl+O”)打开。单击“Open...”选项将弹出如图 7.7 所示的打开文件窗口, 包括驱动器、文件夹、文件类型和文件名等内容, 将鼠标指向所需要的文件名然后双击左键即可将其打开。进行模块化程序设计时根据需要可同时创建多个文件, 对每个文件分别编辑, 完成后应单击“Save”选项(快捷键为“Alt+S”)存盘。如果希望将所编辑的文件换名存盘, 可单击“Save As...”选项, 这时将弹出类似于图 7.7 所示的窗口, 只要在所选定的驱动器和文件夹中指定希望保存的文件名就可以了。选项“Save All”可将打开的多个文件同时存盘。存盘完成后文件并不关闭, 只有单击“Close”选项才能将文件关闭。

第二栏为 μ Vision2 的器件库管理(Device Database...), 这是 μ Vision2 的一个独具特色的选项, 通过它可以选择当前项目所采用的单片机器件。现在已有多个厂家生产 400 多种不同类型的 8051 单片机, 其中大部分已经包含在了 μ Vision2 的器件库中, 用户还可以根据需要向器件库添加新型号的单片机器件。用鼠标左键单击“Device Database...”选项, 将弹出如图 7.8 所示的窗口, 用鼠标左键双击窗口左边的厂家名及其所生产 8051 单片机型号, 选定器件的生产厂商名将自动显示在窗口右边的 Vendor 栏, 器件名显示在“Device”栏, 对该器件的简单描述显示在“Description”栏, 该器件在当前项目中的应用选项显示在“Option”栏中。选定器件后若改动了其中任一应用选项可单击窗口右边的 Update 按钮进行更新, 此外窗口右边还有一个 Add 按钮, 用来向数据库添加新的器件。



图 7.7 打开文件窗口

第三栏为文件打印处理, “Print setup...”选项用来设置打印机, “Print Preview”选项

用来进行文件打印预览,用鼠标左键单击“Print”选项(快捷键为“Ctrl+P”),即可将当前编辑的文件打印输出。

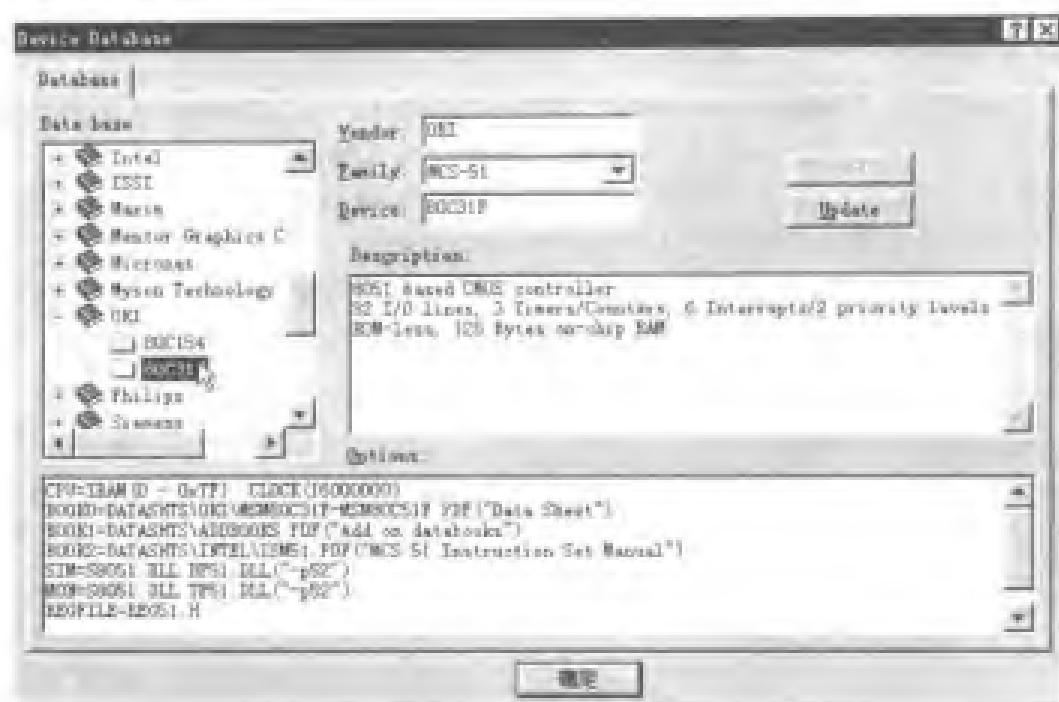


图 7.8 器件库窗口

第四栏中保存有最近打开过的文件名,单击文件名可立即打开该文件。
第五栏的“Exit”选项为退出 μ Vision2 环境,返回到 Windows 状态。

7.1.2 Edit 菜单

Edit 菜单如图 7.9 所示,主要用于对当前已打开的文件进行编辑处理。Edit 菜单分为 6 栏,第一栏为撤销与恢复操作,如果在文件编辑处理过程中发生误操作可用“Undo”选项(快捷键为“Ctrl+Z”)进行撤销,对于误撤销的编辑文本可用“Redo”(快捷键为“Ctrl+Shift+Z”)进行恢复。

第二栏为剪切、复制与粘贴操作,其中“Cut”选项(快捷键为“Ctrl+X”)可将当前编辑窗口中所选定的文本剪切到 Windows 剪贴版中,“Copy”选项(快捷键为“Ctrl+C”)将选定的文本复制到 Windows 剪贴版中,Paste 选项(快捷键为“Ctrl+V”)将剪贴版中的内容粘贴到当前编辑的文件中去。

第三栏为文本缩进操作,“Indent Selected Text”选项用于将所选定的文本右移一个 Tab 键

距离,“Unindent Selected Text”选项用于将所选定的文本左移一个 Tab 键距离,Tab 键距



图 7.9 Edit 菜单

离可以在 View 菜单的“Options”选项中设定,采用文本缩进方式可以编写格式清晰的 C51 程序。

第四栏为文本标记操作,在当前所编辑的文件中,将光标定位到希望进行标记的那一行,用鼠标左键单击“Toggle Bookmark”(快捷键为“Ctrl+F2”)选项,可在该行处标定一个记号,再次单击“Toggle Bookmark”选项将取消该标记;“Goto Next Bookmark”(快捷键为“F2”)和“Goto Previous Bookmark”(快捷键为“Shift+F2”)选项分别用于将光标快速定位到下一个标记点或上一个标记点,“Clear All Bookmarks”选项为清除所有标记点,利用文本标记功能很容易实现对 C51 源程序的编写或修改。

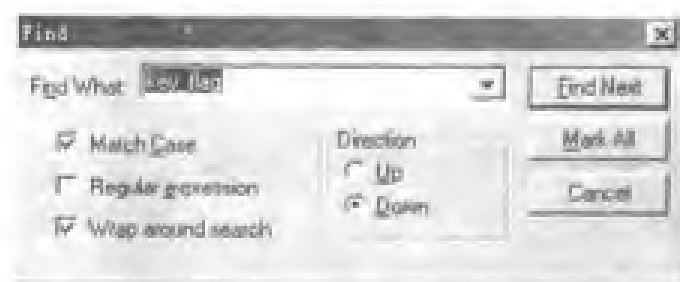


图 7.10 查找窗口

第五栏为查找和替换操作,需要在当前编辑的文件中查找某个字符串(一般为变量名或表达式)时,用鼠标左键单击“Find...”选项(快捷键为“Ctrl+F”),弹出如图 7.10 所示的窗口,在“Find What:”栏中键入希望查找的字符串,再单击窗口右边的 Find Next 按钮,将从当前光标所在行开始查找指定的字符串。如果文件中存在

相符的字符串,光标就会自动跳转到该字符串的位置,单击“Mark All”按钮可以对当前文件中所有相符的字符串进行标记。查找时还可以通过窗口中的“Up”和“Down”选项来指定从当前光标所在位置向上或向下查找。窗口中还有三个复选框:“Match Case”为按指定的字符串大小写形式查找,“Regular expression”为查找规则表达式,“Wrap around search”为循环查找。需要对指定的字符串进行查找替换时,可单击“Replace...”选项(快捷键为“Ctrl+H”),弹出如图 7.11 所示窗口,在“Find What:”栏中键入待查找的字符串,在“Replace With:”栏中键入希望替换的字符串,单击窗口右边的“Replace”按钮即可完成替换;单击“Find Next”则对当前查找到的字符串不作替换而进行下一次查找;单击“Replace All”按钮则对当前文件中所有相符的字符串全部进行替换。该窗口中的几个复选框与查找窗口类似。除了可以在当前编辑的文件中查找之外,还可以同时在其他文件中进行查找,单击“Find in Files...”选项,弹出如图 7.12 所示的窗口,在“Find what:”栏中键入待查找的字符串,在“In Files of”栏中键入希望进行查找的文件类型,在“In Folder”栏中指定查找路径,按钮“Browse...”按钮可浏览所有文件目录,以便于用户指定查找路径。窗口中有三个复选框,“Look in subfolders”框可对指定路径的子目录进行查找;“Match case”框规定按指定字符串的大小写形式进行查找;“Match whole word only”框规定仅查找完全匹配的字符串。对文件的查找结果显示在 μ Vision2 输出窗口的文件查找栏中,如图 7.13 所示,用鼠标左键双击某一条查找结果,编辑窗口会自动调入查找字符串所在的文件,并将光标停留在找到字符串的那一行。熟练应用上述查找与替换功能可极大提高 C51 源程序文件的编辑效率。

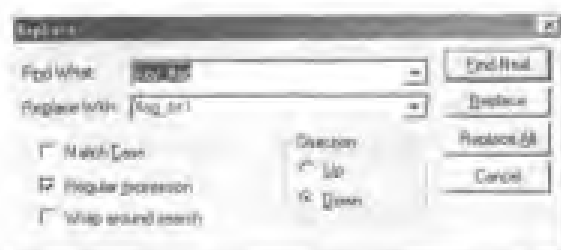


图 7.11 替换窗口

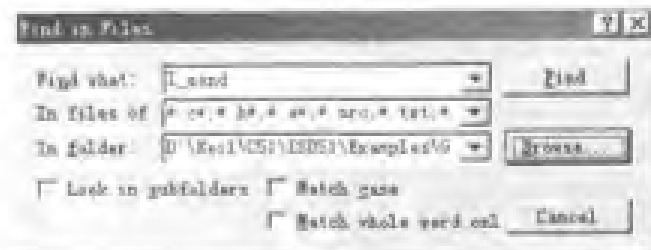


图 7.12 多文件查找窗口

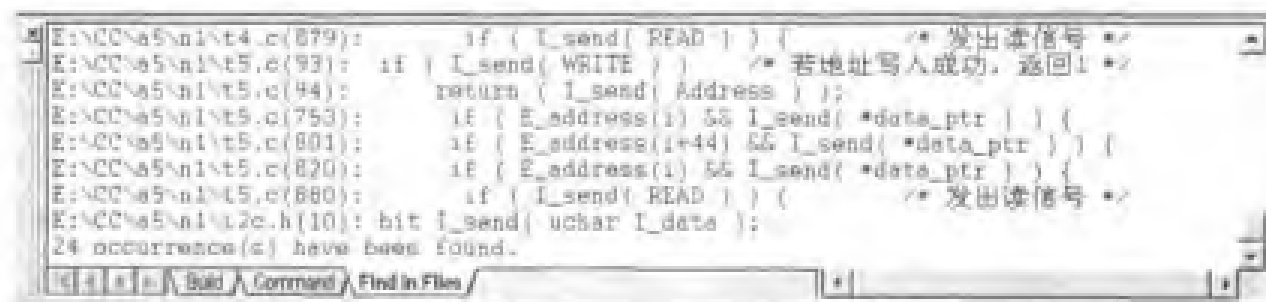


图 7.13 文件查找结果

编辑菜单最后一栏用于选定 Cx51 源程序里一对匹配括号中的内容。先将光标放在某个括号旁边，单击选项“Goto Matching Brace”，与该括号匹配的那部分文件内容将被选定并反转显示，该功能对于检查 Cx51 源程序文件中括号的匹配情况特别有用。

μ Vision2 提供了丰富的鼠标右键功能，在文件编辑窗口内单击鼠标右键可弹出如图 7.14 所示的窗口，单击窗口内的各选项可完成一些特定功能：其中对文本编辑的撤销、恢复、剪切、复制与粘贴、文本标记等操作与 Edit 下拉菜单类似。第三栏的选项“Insert #include <AT89X52.H>”用于插入当前项目所选定单片机器件的包含头文件。第四栏中“Go To Line”选项用于跳转到程序文件的指定行，单击该选项弹出如图 7.15 所示窗口，在其中键入希望跳转到的行号再单击“Go To”按钮，光标将立即跳转到指定的程序行； μ Vision2 允许在编辑状态下进行断点插入操作，先将光标定位在希望插入断点的行上，单击“Insert/Remove Breakpoint”选项，可在选定行插入/删除一个程序调试断点，插入一个断点后，该行最左

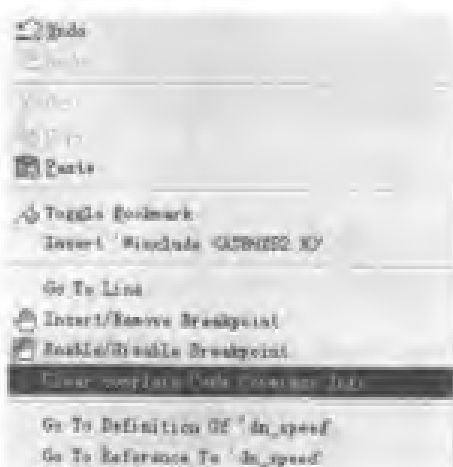


图 7.14 编辑状态下的右键窗口

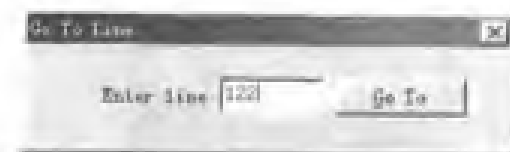


图 7.15 Go To Line 窗口

端将出现一个红方块，并且当进入调试状态后该断点位置不变，这给程序调试带来了极大的方便。单击“Enable/Disable Breakpoint”选项可激活/禁止选定的一个断点，单击“Clear complete Code Coverage Info”选项可清零代码覆盖窗口信息，关于代码覆盖窗口请参见下一节“View”下拉菜单中的“Code Coverage Window”选项。编辑状

态下右键功能窗口的最后两个选项是一种特殊查找功能,先选定程序文件中的一个标识符(变量或函数名),单击“Go To Definition of...”选项,光标将立即跳转到定义该标识符的程序行,单击“Go To Reference of...”选项,光标将立即跳转到引用该标识符的程序行。灵活运用 μ Vision2 提供的右键功能对于 C51 程序的编写和调试具有许多益处,请读者注意掌握这一新特点。

7.1.3 View 菜单

View 菜单如图 7.16 所示。第一栏用于快捷工具条按钮的显示/隐藏切换,有四个选项:

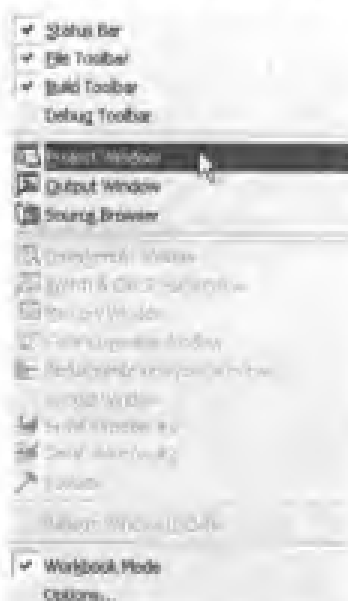


图 7.16 View 菜单

“Status Bar”、“File Toolbar”、“Build Toolbar”和“Debug Toolbar”,选中某个选项时,与之对应的状态栏、文件工具条按钮、编译选项工具条按钮以及调试工具条按钮将显示在 μ Vision2 窗口中,不想显示某个工具条时只要不选中其对应的那个选项即可。

第二栏用于 μ Vision2 中项目窗口、输出窗口和资源浏览器窗口的显示/隐藏切换。选中“Project Window”选项打开项目窗口,编辑状态下打开项目窗口时自动进入“Files”标签页,如图 7.17 所示,显示当前项目的文件管理状态,调试状态(Debug)下则自动进入“Regs”标签页;如图 7.18 所示,显示用户程序执行过程中单片机内部寄存器的当前状态。利用项目窗口的 Files 栏可以调整对当前项目的文件管理,在该窗口中单击右键弹出如图 7.19 所示窗口,其中第一栏两个选项为当前项目选取器件(Select Device for Target ...),指定项目文件属性(Options for File ...).

右键窗口第二栏有 5 个选项分别用于打开文件(Open 文件名)、重新编译并创建目标文件(Rebuild target)、编译并创建目标文件(Build target)、编译程序文件(Translate 路径\程序文件名)以及停止创

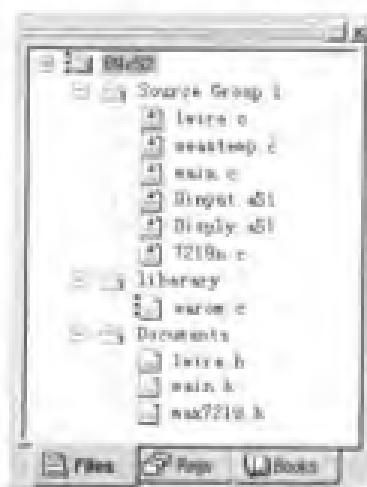


图 7.17 编辑状态下的项目窗口

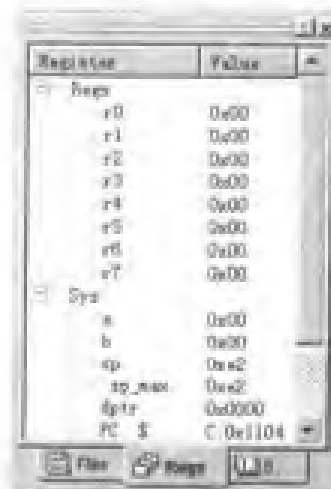


图 7.18 调试状态下的项目窗口

建 (Stop build)；右键窗口第三栏有 3 个选项：“Add Files to Group...”选项用于向当前项目的某个文件组增加文件，“Remove File ‘lwire.c’”用于从项目中删除一个文件（这里 lwire.c 是所选定的程序文件，也可以是其他文件），“Targets, Groups, Files...”选项用于调整当前项目的文件组成。项目窗口“Files”栏右键菜单的其他选项功能与 Project 下拉菜单中的对应选项功能相同，它们的详细描述请见下一节 Project 菜单。

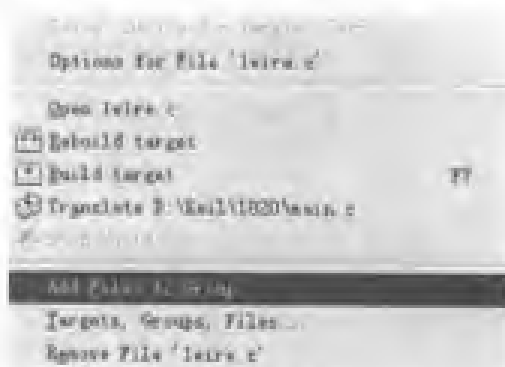


图 7.19 项目窗口 Files 栏的右键菜单

选中“View”菜单第二栏中的“Output Window”选项，打开输出窗口，对当前项目进行目标文件创建时，输出窗口自动进入其中的“Build”标签页，如图 7.20 所示。若目标文件创建成功，窗口内将显示“0 Error(s), 0 Warning(s)”；若创建出错，窗口内将显示相应的错误信息，用鼠标左键双击某个错误信息，光标自动跳转到相应程序文件发生错误的那一行，便于对文件进行编辑修改。



图 7.20 输出窗口的 Build 栏

如果在“Project 菜单/Options for Target”选项中“Output”栏选中了“Browse Information”复选框，则可以在对当前项目成功编译连接之后，使用“View”菜单第二栏中的“Source Browser”选项打开如图 7.21 所示的资源浏览器窗口，窗口上半部分左边的“Symbol”栏可键入希望显示的符号名，可以使用三个通配符：#（数字 0~9）、\$（标识字符）、*（任意符号）；“Symbol”栏下边是 6 个“Filter”按钮：Macros（宏定义）、Data（数据）、Functions（函数）、Sfr(Bits)（特殊功能寄存器（位））、Parameters（参数）以及 Type（类型），某个按钮按下时将显示对应的符号信息。“File”栏可设定符号信息所在的文件；窗口上半部分右边“Memory”复选框可以设定符号信息的存储器类型；窗口下半部分左边显示的是用户程序文件中的各种符号信息，用鼠标左键单击其中某一条符号信息，窗口下半部分右边将显示出在哪个用户文件中对该符号进行了定义，以及在文件的哪一行引用了该符号（其中[D]表示定义、[R]表示引用、[r]表示数据读、[w]表示数据写、[r/w]表示数据读写、[&]表示地址引用）；将鼠标指向右边某一信息并单击右键，可打开对应的文件仔细察看该符号的定义或引用情况。利用资源窗口能够很方便地分析用户文件中各种符号的定义和引用，对于程序的调试或修改带来了极大方便。

“View”菜单第三栏和第四栏中的各个选项用于打开 Debug 状态下的各种窗口，未进入 Debug 状态时它们呈灰色显示，不能使用。

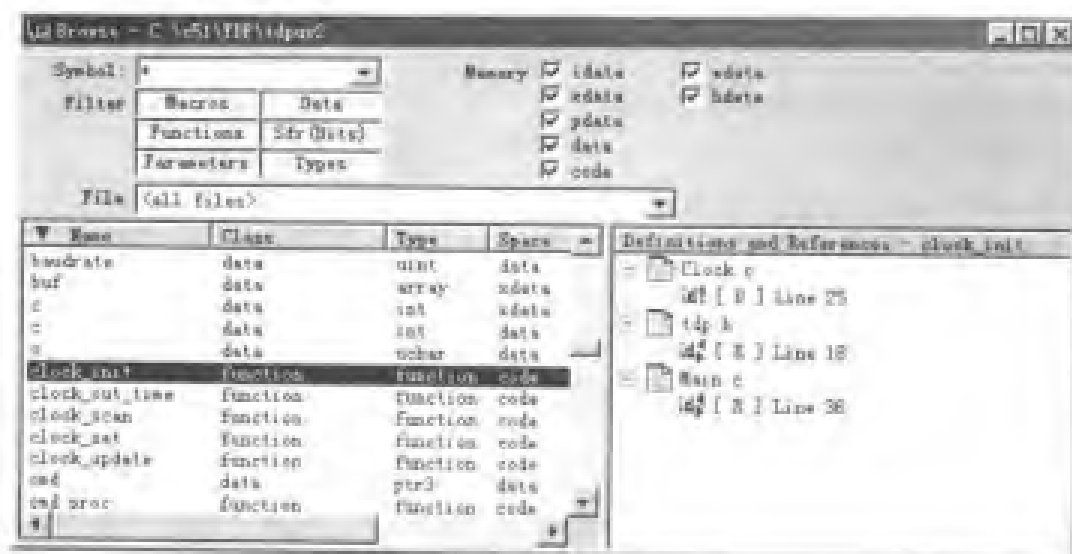


图 7.21 资源浏览器窗口

“View”菜单最后一栏有两个选项，选项“Workbook Mode”用于将主窗口按书签方式显示，如图 7.22 所示。这种显示方式下将鼠标指向不同文件的书签名并单击左键，很容易实现各文件之间的切换。选项“Options...”用于设置µVision2 的内部编辑器、显示字体与颜色，以及进行快捷键定义，单击该选项弹出如图 7.23 所示窗口。窗口分为三个标签页：“Editor”栏用于设置内部编辑器工作方式，其中有 4 个复选框，选中“Create Backup files(*.Bak)”复选框时可在编辑时创建备份文件，选中“Auto Indent”复选框时可在编辑文件时采用自动缩进方式，选中“Automatic reload externally modified files”复选框时可在对外部文件进行过修改后自动装入，该栏中还可以设置 C/C++ 文件、ASM 文件以及其他文件的彩色语句显示方式及 Tab 键的插入空格数；窗口中“Colors & Fonts”栏用于设置µVision2 的显示字体及颜色；窗口中“Shortcut Keys”栏用于设置µVision2 的快捷键。

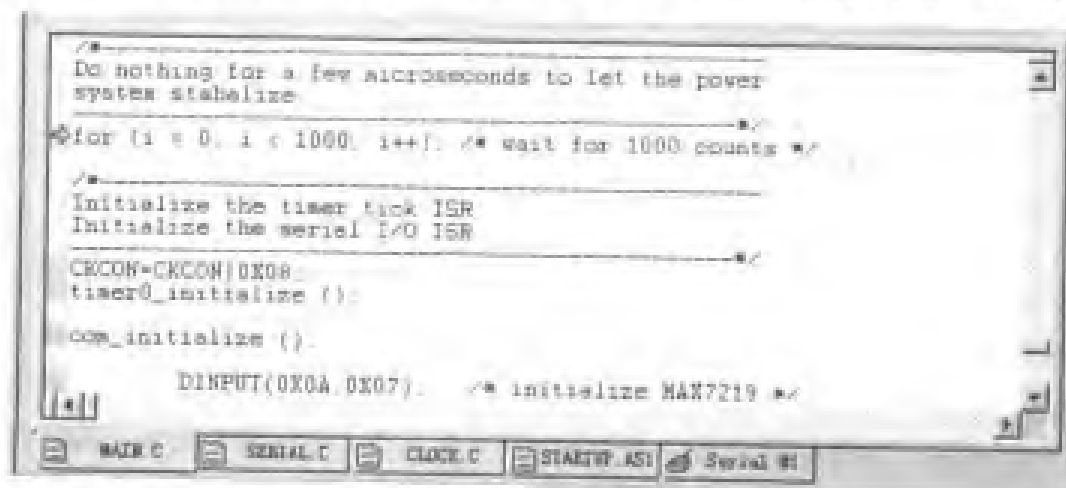


图 7.22 书签显示方式下的主窗口



图 7.23 View 菜单的 Options 窗口

7.1.4 Project 菜单

μ Vision2 集成环境提供了强大的项目 (Project) 管理功能, 通过项目文件可以十分方便地进行应用程序开发, 一个项目中可以包括各种文件, 如源程序文件、包含头文件、说明文件。通过项目管理操作用户能够随时对项目文件进行调整, 如增删程序模块文件、改变目标 CPU、配置不同的 C51 编译链接命令、使用模拟器 (Simulator) 或带监控的用户目标板 (Monitor-51) 调试程序等, 读者通过实际操作逐步学习并熟练掌握项目操作对于开发自己的单片机应用系统帮助极大。“Project”菜单如图 7.24 所示, 分为 4 栏。第一栏中的“New Project...”选项用于创建一个新项目。单击该选项弹出如图 7.25 所示窗口, 先选定希望保存项目文件的目录, 然后在“文件名”栏内键入项目文件名, 单击“保存”按钮, 弹出如图 7.26 所示器件选择窗口,



图 7.24 Project 菜单



图 7.25 创建新项目窗口

从窗口左边 μ Vision2 提供的器件库中选定希望采用的单片机器件厂家及其器件名, 窗口右边将显示对应器件的简单描述, 窗口中还有两个复选框用来确定是否采用扩展链接定位器 Lx51 和扩展宏汇编器 Ax51, 而不采用常规连接定位器 BL51 和常规宏汇编器 A51, 单击确定按钮即可在指定的目录中创建出一个项目文件。

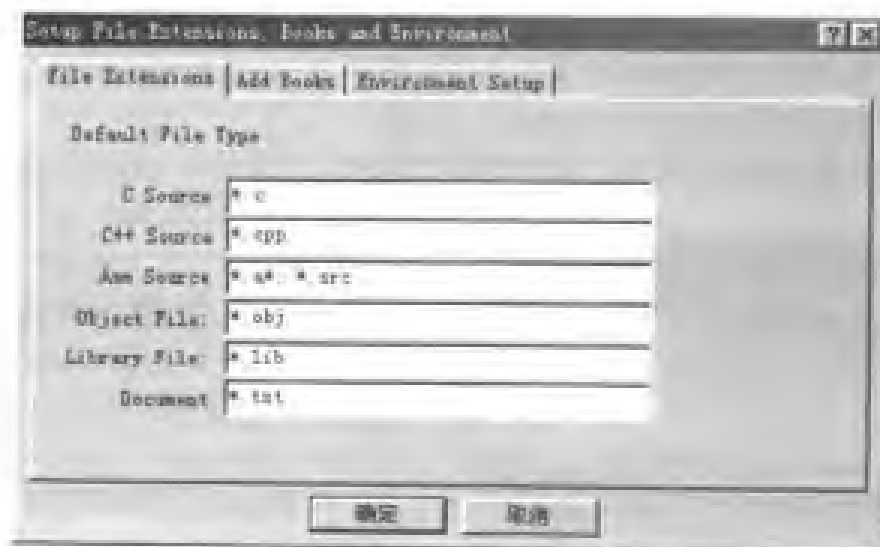


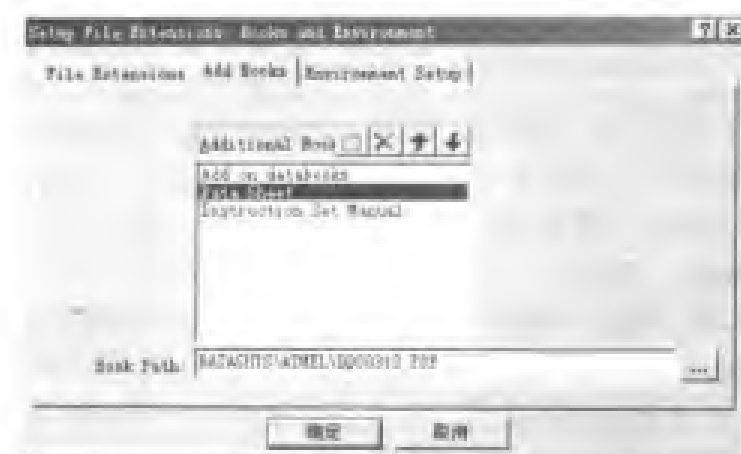
图 7.26 器件选择窗口

“Project”菜单第一栏中的“Import μ Vision1 Project...”选项用于导入 Keil C51 V5.0 版本 μ Vision1 中的项目文件，通过导入功能使得在 μ Vision1 环境下创建的项目可以在 μ Vision2 环境中使用。

“Project”菜单第一栏中的“Open Project”和“Close Project”选项分别用于打开已有的项目文件和关闭当前项目文件。

“Project”菜单第一栏中最后一个选项“File Extensions, Books and Environment”用于设定文件扩展名、增加在线书籍和设定 μ Vision2 工作环境。单出该选项弹出如图 7.27 所示窗口，其中已经具备默认的文件扩展名，需要时用户可以自己重新设定。单击该窗口中的“Add Books”标签页弹出如图 7.28 所示窗口，用于向 μ Vision2 环境中增加工作时的在线书籍。从“View”下拉菜单打开“Project Window”，可以看到三个标签页“Files”、“Regs”和“Books”，分别显示工作文件、工作寄存器状态以及在线书籍。

图 7.27 μ Vision2 环境设置窗口中的文件扩展名标签页

图 7.28 μ Vision2 环境设置窗口中的增加书籍标签页

“Books” 标签页如图 7.29 所示，其中“ μ Vision2”和“Tools User's Guide”两项是随软件安装的在线书籍，“Device Data Books”则是用户自己增加的书籍，需要注意的是增加书籍时必须保证对应书籍文件存在。将鼠标指向“Books”标签页中任一项并双击左键，可以随时打开该书籍进行查阅，这一点对于用户来说提供了很大的方便。

图 7.30 所示为 μ Vision2 环境设置窗口的“Environment Setup”标签页，用于设置的工作环境，可以分别设置 Cx51 编译器执行文件目录路径（BIN Folder）、包含文件目录路径（INC Folder）、库文件目录路径（LIB Folder）以及寄存器文件目录（Regfile），对于每一个目录路径均可单击其中的“...”按钮进行浏览选择，窗口中还有一个复选框“Use Settings from TOOLS.INI”，选中该框将使用软件安装时所创建配置文件“TOOLS.INI”的默认设置，只有设置了正确的文件目录路径之后 Cx51 编译器才能够正常运行。

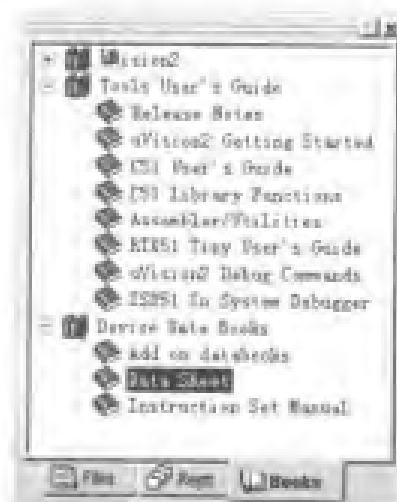
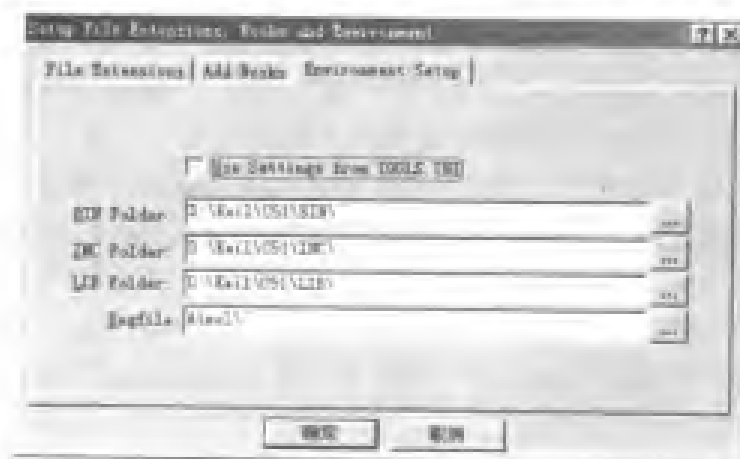


图 7.29 Project 窗口中的 Books 标签页

图 7.30 μ Vision2 的环境设置窗口中的工作环境标签页

“Project”菜单第二栏的各个选项用于对当前已打开的项目进行管理操作， μ Vision2 规定一个项目中可以有不同的目标（Targets），例如，采用模拟器（Simulator）的目标或采用监控程序（Monitor）的目标，每个目标都有自己的文件组（Groups）和源文件（Files），创建新项目时将自动默认“Targets 1”目标和“Source Group 1”文件组，用户可以根据需要进行调整。单击“Project”菜单第二栏中“Targets, Groups, Files”选项弹出如图 7.31 所示窗口，其中的“Targets”标签页用于设置当前项目中的目标，在“Target to Add:”栏中键入希望增加的目标名，单击“Add”按钮即可在当前项目中增加一个目标，如果选中复选框“Copy all Settings from Current Target”，则使增加的目标具有与当前目标相同的设置。“Available Targets”栏中显示所有可用目标，选中其中一个目标并单击“Set as Current Target”按钮，可将其设为项目中的当前目标；单击“Remove Target”按钮可将其从当前项目中删除。



图 7.31 Targets 标签页

图 7.32 为“Groups/Add Files”标签页，用于调整当前项目中的文件组和源文件。在“Group to Add”栏键入组名，单击“Add”按钮可将其加入到当前项目之中；“Available Groups”栏中显示所有可用的组，选中其中一个组，单击“Remove Group”按钮可将其删除；单击“Add Files to Group...”按钮弹出如图 7.33 所示窗口，通过“搜寻”栏找到源文件所在的目录路径，在“文件名”栏键入希望加入的源文件名，单击“Add”按钮即可将其加入到当前项目的指定组中。“Project”菜单第二栏的“Select Device for Target ‘Target 1’”选项用于为当前项目中的目标选择（更改）器件，其操作与创建新项目时的器件选择方法相同。单击“Project”菜单第二栏的“Remove Group ‘Source Group 1’ and it’s Files”选项，将从当前项目中删除一个指定组以及其中的所有文件（操作前要先用鼠标从项目窗口中选定一个组）。

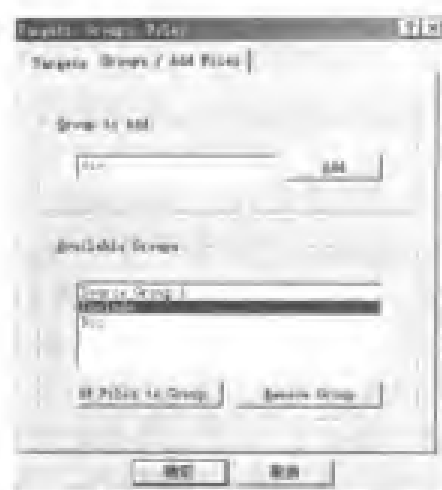


图 7.32 Groups/Add Files 标签页

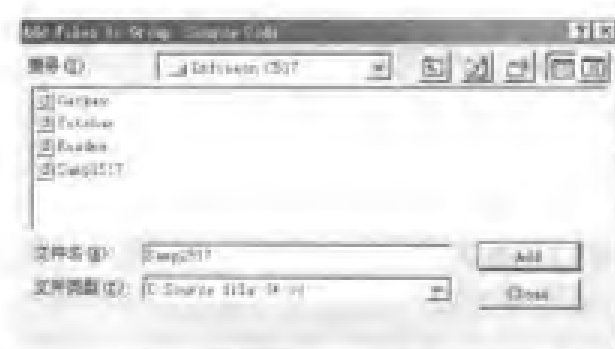


图 7.33 加入源文件窗口

“Project”菜单第二栏的“Options for Target ‘Target 1’”是一个十分重要的选项，很多有关编译、链接、定位、输出文件等控制命令都可利用该选项功能来完成，图 7.34 所示为该选项中的“Target”标签页。在“Xtal(MHz)”栏中键入的数值可以设定模拟仿真时单片机的振荡器频率；通过“Memory Model”栏可以设定编译模式（Small、Compact、Large）；通过“Code Rom Size”栏可以设定 ROM 空间大小（Small、Compact、Large）；通过“Operating”栏可以设定是否采用 RTX 实时操作系统（None、RTX-51 Tiny、RTX-51 Full）；“Off-chip Code memory”栏和“Off-chip Xdata memory”栏分别用来设置单片机片外 EPROM 和片外 RAM 存储器空间的起点及其大小，如果在选择器件时采用了具有片内 ROM 的单片机，并且希望使用片内 ROM，则可以选中窗口内的复选框“Use On-chip ROM”；窗口左下角还有一个复选框“Code Baking”，用于存储器分组设置，分组数最大为 64，“Bank Area”用于设定每个存储器分组的起始与终止点地址；如果选择器件时采用了 Lx51 扩展连接定位器，则可以使用窗口右下角的两个复选框“‘far’ memory type support”（‘far’存储器类型支持）和“Save address extension SFR in interrupt”（中断内保存扩展 SFR 地址），如果选择器件时采用的是常规 BL51 连接定位器，这两个复选框是不能使用的。



图 7.34 Options 选项中的 Target 标签页

“Options”选项中的“Output”标签页如图 7.35 所示，用于设置当前项目经创建之后生成的可执行代码文件输出。单击“Select Folder for Objects”按钮将弹出一个目录路径浏览窗口，便于用户设定存放目标代码文件的目录；在“Name of Executable”栏键入将生成的可执行代码文件名（默认值就是当前项目名）；窗口中间有两个圆形单选框：“Create Executable”（生成可执行代码文件）和“Creat Library”（生成库文件），二者只能选其一。在“Create Executable”下面还有 4 个方形复选框：“Debug Information”（调试信息）、“Browse Information”（浏览信息）、“Merge32K Hexfile”（合并 32K Hex 文件）、“Create HEX File”（生成 HEX 文件），一般为了调试程序方便通常要选中前两个复选框，如果用户的编程器只能支持 HEX 文件，则需要选中最后一个复选框，通过 HEX 栏还可以选择 HEX 文件格式，采用常规 BL51 连接定位器生成标准 HEX-80 格式文件，采用 Lx51 扩展连接定位器生成 HEX-386 格式文件。窗口下面有 4 个复选框用于设定项目创建完成之后的操作：“Beep When Complete”（完成后发提示音）、“Start Debugging”（开始调试）、“Run User Program #1”（运行用户程序#1）、“Run User Program #2”（运行用户程序#2），在运行用户程序栏可以直接键入希望运行的程序路径与程序名，也可以先单击“Browse”按钮，从弹出的浏览窗口中查找到用户程序所在目录路径，再双击该程序名。

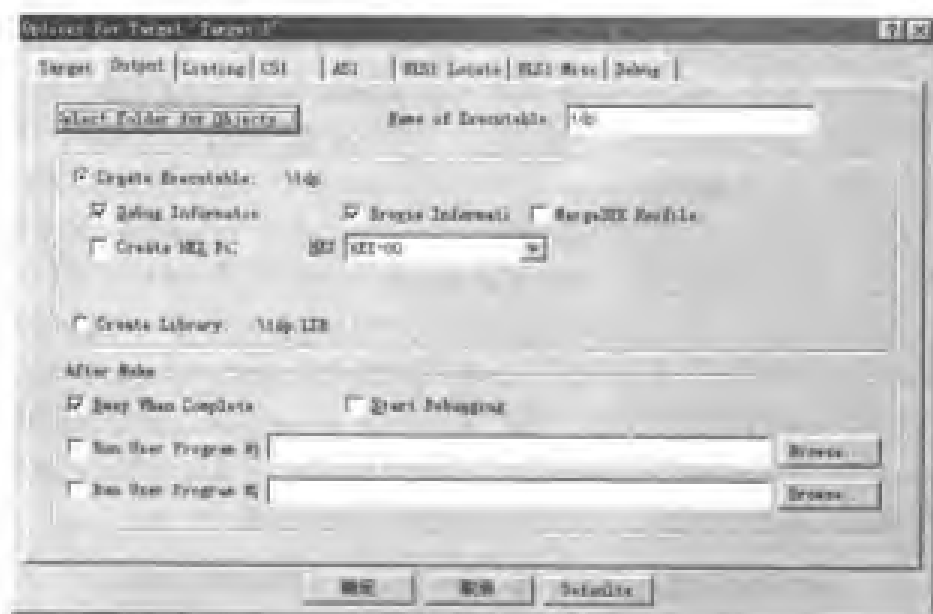


图 7.35 Options 选项中的 Output 标签页

“Options”选项中的“Listing”标签页如图 7.36 所示，用于设置当前项目经创建之后生成的列表文件。单击“Select Folder for Listings”按钮将弹出一个目录路径浏览窗口，便于用户设定存放列表文件的目录，通过“Page Width”和“Page Length”栏可以设定列表文件的页宽和页长；复选框“C Compiler Listing”用于设定 C51 编译器列表控制，它下面有 4 个复选框：“Conditions”（条件编译）、“Symbols”（符号）、“#include File”（包含文件）、“Assembly Code”（汇编代码），可根据需要进行设定；复选框“C Preprocessor Listing”用于设定列表文件中是否生成预处理器列表；复选框“Assembler Listing”用于设定 A51 宏汇编器列表控制，它下面有 3 个复选框：“Conditions”（条件汇编）、“Symbols”（符号）、

“Cross Reference”(交叉参考),通过“Macros”栏可以设定三种宏操作(Final、1st、All);复选框“Linker Listing”用于设定 BL51(或 Lx51)连接定位器列表控制,它下面有 8 个复选框:“Memory Map”(存储器映像)、“Public Symbols”(公共符号)、“Line Numbers”(行号)、“Cross Reference”(交叉参考)、“Local Symbols”(局部符号)、“Comment Records”(注释记录)、“Generated Symbols”(生成符号)、“Library Symbols”(库符号);可根据需要来选定。



图 7.36 Options 选项中的 Listing 标签页

“Options”选项中的“C51”标签页如图 7.37 所示,用于设置当前项目进行创建时的 Cx51 编译器控制命令。“Preprocessor Symbols”栏用于设定 Cx51 编译器预处理命令符号,在“Define”处可直接键入需要处理的符号;“Code Optimization”栏用于设置 Cx51 编译器的代码优化,通过“Level”栏可以设定 0~11 级优化级别,其中 10、11 级别只有在复选框“Linker Code Packing”被选中的情况下才能使用;通过“Emphasis”栏可以设定两种优化方式(size:代码长度优化, speed:代码速度优化),选中复选框“Global Register Coloring”时将为全局寄存器优化规定一个寄存器文件,选中复选框“Linker Code Packing”将对生成代码进行“跳转优化”,即尽可能地将 Ljmp/Lcall 指令用 Ajmp/ACall 指令代替;选中复选框“Don't use absolute register accesses”时将不使用绝对寄存器访问;通过“Warning”栏可以设定 0~2 级警告;通过“Bits to round for float”栏可以设定浮点数比较运算时的舍入位数(0~7);复选框“Interrupt vectors at”用于设定中断向量基地址;若选中复选框“Keep variables in order”则 C51 编译器对变量在内存中的排列按源程序文件中变量的定义进行;复选框“Enable ANSI integers promotion rule”用于是否激活 ANSI 提升规则;在“Include Paths”栏中可以直接键入包含文件的目录路径,也可以按下该栏右边的按钮,通过弹出的搜寻窗口来确定包含文件所在的目录路径;在“Misc Controls”栏中可以键入其他各种 Cx51 的控制命令,在“Compiler control string”栏中显示所有已设定的 Cx51 编译控制命令。

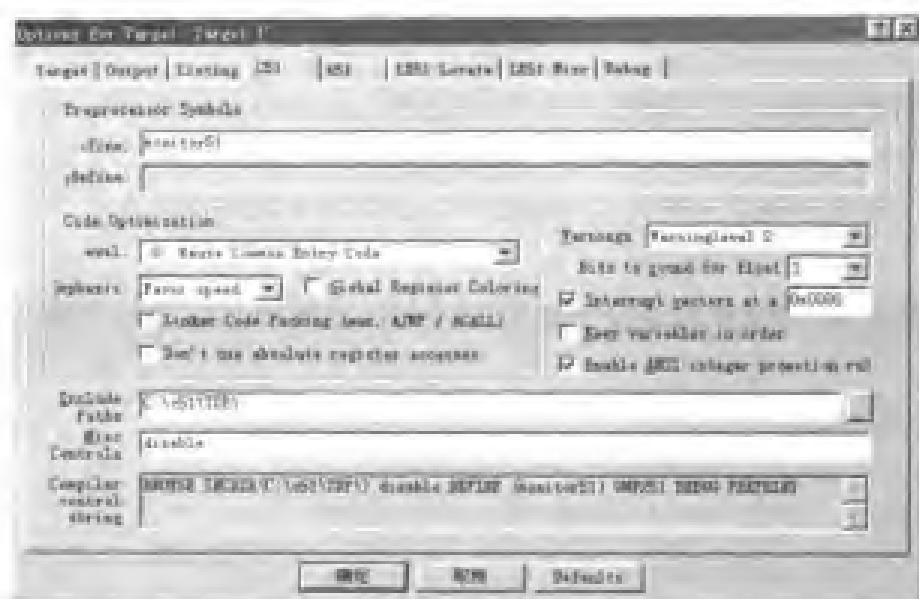


图 7.37 Options 选项中的 C51 标签页

“Options”选项中的“A51”标签页如图 7.38 所示，用于设置当前项目进行创建时的 A51 宏汇编器控制命令。“Conditional assembly control Symbols”栏用于设置 A51 宏汇编器的条件汇编控制符号；“Macro processor”栏用于设置宏处理方式，选中复选框“Standard”时按标准方式处理，选中复选框“MPL”时按 Intel 兼容方式处理；在“Special Function Registers”栏中可以通过复选框“Define 8051 SFR Names”来设定是否采用特殊功能寄存器名定义；在“Include Paths”栏中可以直接键入包含文件的目录路径，也可以按下该栏右边的按钮，通过弹出的搜寻窗口来确定包含文件的目录路径；在“Misc Controls”栏中可以键入其他各种 A51 的控制命令。所有已设定的 A51 控制命令都显示在“Assembler control string”栏中。

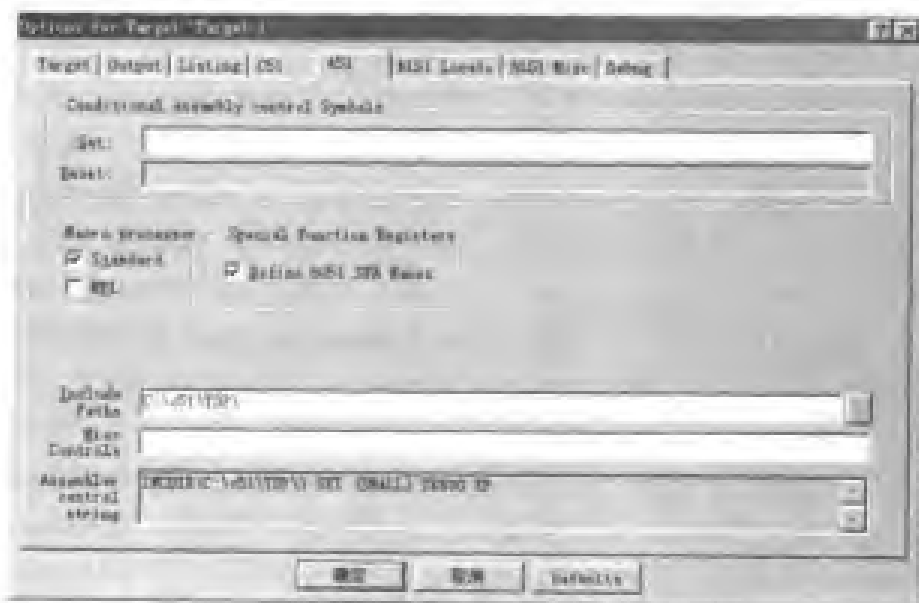


图 7.38 Options 选项中的 A51 标签页

“Options”选项中的“BL51”标签页如图 7.39 所示,用于设置当前项目进行创建时的 BL51 连接定位器控制命令。选中复选框“Use Memory Layout from Target Dialog”时将采用由“Target”标签页设定的存储器组织形式;未选中该复选框时可以通过“Code Range”栏和“Xdata Range”栏键入地址值来设定 ROM 空间范围和片外 RAM 空间范围;此外还可以分别设定 Code、Xdata、Pdata、Precdata、Bit、Data、Idata、Stack 等段的基地址(Base)以及待定位段名(Segments),所有已设定的 BL51 控制命令都显示在“Linker control string”栏中。

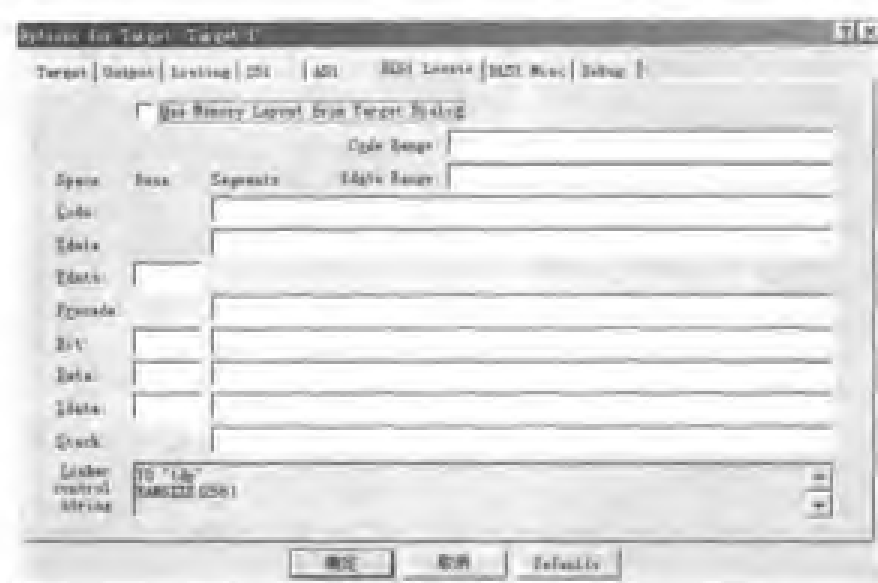


图 7.39 Options 选项中的 BL51 标签页

“Options”选项中的“BL51 Misc”标签页如图 7.40 所示,用于设置当前项目进行创建时 BL51 连接定位器的警告禁止以及覆盖分析等控制命令。在“Disable Warning”栏中可以直接键入希望禁止的警告编号,这样当连接定位操作完成时对应于该编号的警告信息将不再出现;若选中“use linker control file”复选框,将按照“连接定位控制文件”中预先设定的控制命令进行连接定位,单击“Create”按钮将按照当前连接定位设置在指定的目录路径下创建该文件,单击“Browse”按钮将搜寻目录路径来查找已经存在的该文件,单击“Edit”按钮将在 μ Vision2 编辑窗口中打开该文件进行编辑;通过“Overlay”栏可以键入各种 BL51 连接定位器的覆盖控制命令;在“Misc Controls”栏中可以键入其他各种 BL51 的控制命令;所有已设定的 BL51 控制命令都显示在“Linker control string”栏中。

“Options”选项中的“Debug”标签页如图 7.41 所示,用于设置对用户程序的调试方式。选中圆形单选框“Use Simulator”时采用 μ Vision2 模拟器进行调试,选中圆形单选框“Use Keil Monitor-51 Driver”时采用 Keil 公司提供的监控程序进行调试,前者可以在 μ Vision2 环境中仅用软件方式即可完成对用户程序的调试工作,后者则需要硬件目标板支持,两者都可通过复选框“Load Application at Start”来决定是否在启动调试器时装入用户程序,通过复选框“Go till Main”来决定是否装入用户程序后自动运行到 Main 函数处;在“Initialization”栏中可以直接键入调试初始化文件名(Ini File),也可以单击“...”按钮从弹出的窗口中按目录路径搜寻该文件,单击“Edit”按钮将在 μ Vision2 编辑窗口中对

该文件进行编辑, 初始化文件将在启动调试器时自动装入并运行: “Restore Debug Session Settings” 栏用于恢复上次调试对话设置, 有 4 个复选框: “Breakpoints” (断点)、“Toolbox” (工具箱)、“Watchpoints & PA” (观察点与性能分析)、“Memory Display” (存储器显示); “CPU DLL” 栏和 “Driver DLL” 栏中显示 CPU 驱动动态连接库文件名, 通常为 “S8051.DLL”, 用户不要改动这个 DLL 文件及其参数; “Dialog DLL” 栏中显示对话驱动动态连接库文件名, 采用 μ Vision2 模拟器时通常为 DP51.DLL, 采用监控硬件目标板时通常为 TP51.DLL。



图 7.40 Options 选项中的 BL51 Misc 标签页

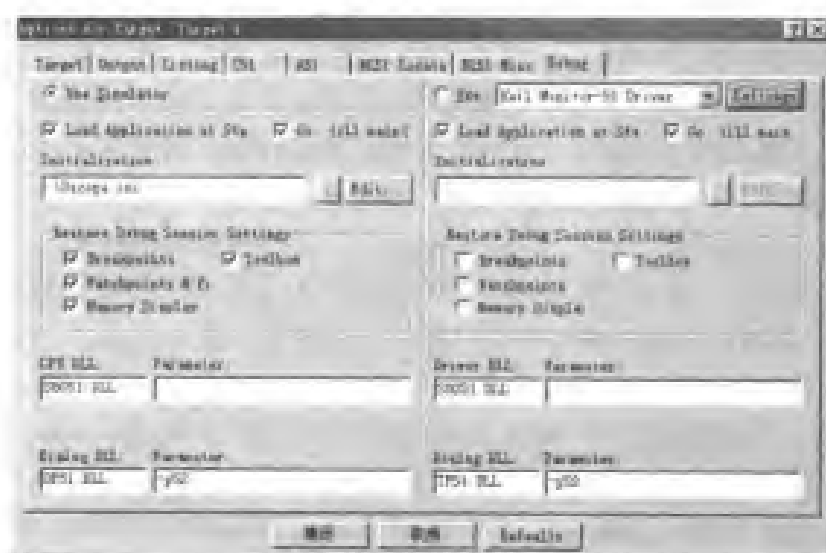


图 7.41 Options 选项中的 Debug 标签页

如果用鼠标选定项目窗口中的一个程序文件名, “Project” 菜单中将出现 “Remove File” 和 “Options for File” 选项。单击 “Remove File” 选项将从当前项目中删除选定的程序文件; 单击 “Options for File” 选项, 弹出如图 7.42 所示窗口, 显示所选定文件的属性, 有两个标签页, 其中 “C51” 标签页与前面介绍的基本一样, “Property” 标签页中的 “Path”、

“Type”、“Size”、“Last change” 栏分别显示该文件所在目录路径、文件类型、文件大小、上次修改时间；“Code Bank” 栏用于设置在分组方式下选定文件的代码组；通过“Stop on Exit” 栏可以选择 μ Vision2 在对当前项目进行创建时的停止和出口条件，在默认方式下 μ Vision2 将对项目中所有文件进行编译处理，无论出现何种错误或警告都不会停止；对于库文件可以通过“Select Modules to Always Include” 栏设定库中的某些模块总是被包含到当前项目之中。图 7.42 中有 5 个方形复选框：“Include in Target Build”（包含到目标代码创建中），利用该复选框可以根据具体要求调整项目中的源程序文件进行编译连接，“Always Build”（总是创建），选中该复选框时无论指定的源程序文件是否经过修改总要进行编译连接操作，这一点对于使用了诸如 `_DATE_` 和 `_TIME_` 等版本信息的源文件来说是很实用的；“Generate Assembler SRC File”（生成汇编 SRC 文件），当 Cx51 源程序文件中采用了 `#pragma asm/endasm` 预处理器命令，则需要选中该复选框，Cx51 编译器将生成对应的汇编语言源文件；“Assemble SRC File”（汇编 SRC 源文件），该复选框通常与前面一个复选框同时应用，对由 Cx51 源程序生成的汇编语言源文件进行汇编而生成目标文件；“Link Publics Only”（仅连接全局符号），该复选框只适用于 Lx51 扩展连接定位器，仅对模块中的全局符号进行连接。

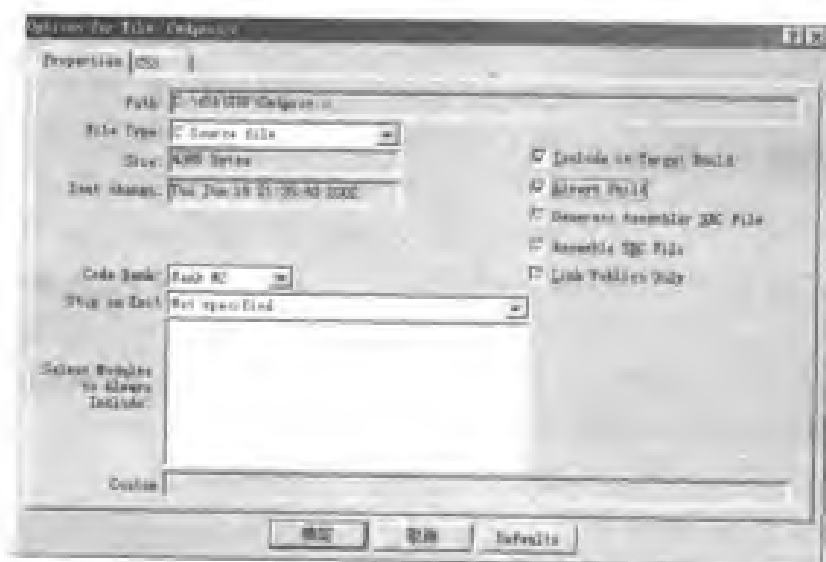


图 7.42 文件属性标签页

如果加入到项目中的文件不是 Cx51 或 A51 源程序文件， μ Vision2 将会弹出如图 7.43 所示窗口，要求用户指定该文件的类型，当用户指定该文件为 Custom 类型时，可以通过图 7.42 的“Custom”栏设定该文件所使用的特殊文件转换器，通常是利用某种特殊文件转换器将该文件转换为 Cx51 或 A51 源文件，然后再将转换得到的文件加入到项目中去，并对整个项目重新进行编译连接生成应用程序代码。



图 7.43 设定用户文件类型窗口

“Project”菜单第二栏最后一个选项“Clear Group and File Options”，用于清除已经设定的各个组和文件的属性。

“Project”菜单第三栏中4个选项都是用于项目目标代码创建的，单击“Build target”选项将对项目中的源文件进行编译连接并生成可执行目标代码，如果在原来项目中添加了新的源文件或者修改过源文件，则仅对新文件或修改过的源文件进行编译连接，以便加快创建速度；单击“Rebuild all target files”选项将对项目中所有源文件（无论是否新文件）进行编译连接并生成目标代码；单击“Translate current file”选项将对项目中选定的源文件进行转换；单击“Stop build”选项将停止当前正在进行目标代码创建。

“Project”菜单最后一栏用于快速打开项目，该栏内保存有最近10个曾经打开的项目名及其路径，单击其中任意一个将立即打开该项目。

μ Vision2 环境提供了丰富的右键功能，上述“Project”菜单中的各种选项功能，都可以从 Project（项目）窗口的右键菜单中找到，实际应用中往往采用右键功能可以简化操作。

μ Vision2 中集成了 Debug 调试器，可以在不退出 μ Vision2 的条件下直接进行用户程序的仿真调试，相应地提供了与调试有关的两个下拉菜单“Debug”和“Peripherals”，关于这两个菜单的操作将在 7.2 节“ μ Vision2 的调试器”中详细介绍。

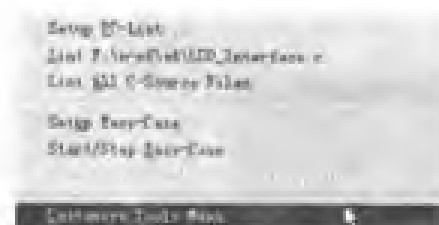


图 7.44 Tools 菜单

7.1.5 Tools 菜单

“Tools”菜单如图 7.44 所示。第一栏用于设置 Gimpel Software 公司的 PC-Lint 软件，PC-Lint 是一种对 C 语言源程序进行快速详细语法查错的专用软件，一些编译器不能识别的错误可由 PC-Lint 查出。第二栏用于设置 Siemens 公司的 Ease-Case 软件，Ease-Case

是一种具有流程图功能的 C 语言文件编辑器，可以很方便地按结构方式编写 C 语言源程序。由于以上两种软件都不包括在 Keil Cx51 软件包中，需要单独购买，因此这里不作详细介绍。

“Tools”菜单的第三栏“Customize Tools Menu...”选项用于扩展 Tools 菜单，添加用户自己的应用工具。单击该选项弹出如图 7.45 所示窗口，其中“Menu Content”栏用于设置显示在 Tools 扩展菜单中的用户工具文本选项，文本中可以包括“键码”和“文件码”，用符号“&”可以定义扩展选项的快捷键。“键码”和“文件码”的定义如下：

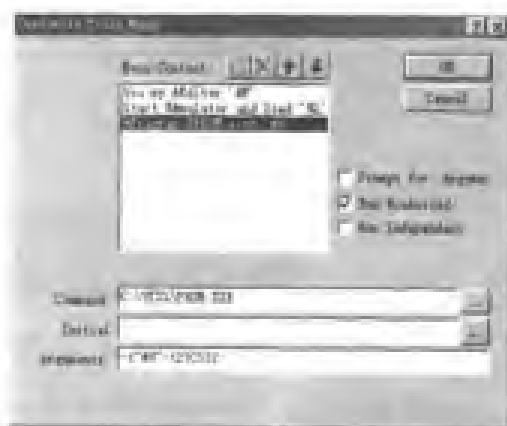


图 7.45 用户工具设置窗口

键码	定义
%	包含扩展名但不包含路径的文件名，（如：PROJECT1.UV2）
#	包含绝对路径的文件名，（如：C:\MYPROJECT\PROJECT1.UV2）
%	包含扩展名但不包含路径的文件名，（如：PROJECT1.UV2）
@	不包含路径和扩展名的文件名（如：PROJECT1）
\$	文件夹名（如：C:\MYPROJECT）

~	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)
文件码	定义
F	指定在 Project 窗口-Files 标签页中选定的文件 (如: MEASURE.C)。如果选定的是目标名 (Target), 则指定该项目文件; 如果选定的是文件组名 (Group), 则指定当前 μ Vision2 编辑器中的文件
P	当前项目名 (如: PROJECT1.UV2)
L	连接定位器的输出文件, 通常是为调试而生成的可执行文件 (如: PROJECT1)
H	Hex 应用文件 (如: PROJECT1.H86)
X	μ Vision2 可执行文件 (如: C:\KEIL\UV2\UV2.EXE)

“Command” 栏用于键入用户工具的可执行文件名及其所在路径。

“Initial” 栏用于键入用户工具的当前工作目录, 如果为空, 则使用当前 μ Vision2 项目所在的目录。

“Arguments” 栏用于键入用户工具执行文件所要求的参数。

窗口中还有三个方形复选框, 若选中 “Prompt for Argument” 复选框, 在使用扩展菜单相应工具选项时, 将弹出一个对话框, 提示输入用户程序的命令行参数; 若选中 “Run Minimized” 复选框, 将使用户工具应用程序运行时的窗口最小化; 若选中 “Run Independent” 复选框, 将使用户工具应用程序独立运行。

采用如图 7.45 的设置后, Tools 菜单扩展为如图 7.46 所示。

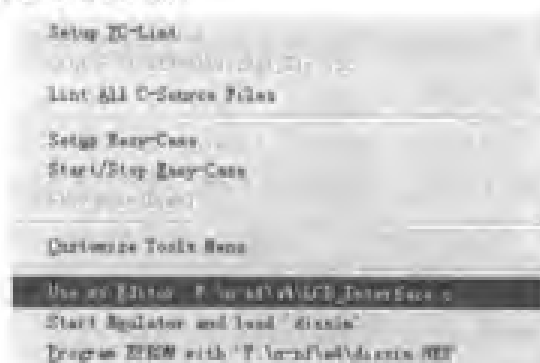


图 7.46 Tools 用户工具扩展菜单

7.1.6 SVCS 菜单

μ Vision2 为软件版本控制系统 (SVCS) 提供了一个可配置的接口, “SVCS” 菜单的配置存储在一个模板文件中。Keil Cx51 软件包中提供了如下几个预先配置的 SVCS 模板文件: Intersolv PVCS、Microsoft SourceSafe 以及 MKS Source Integrity。通过适当配置后的 SVCS 菜单用户可以调用自己的版本控制系统命令行工具。图 7.47 所示为配置之前的 “SVCS” 菜单, 它只有一个选项: “Configure Version Control...”, 单击该选项弹出如图 7.48 所示配置窗口, 其中 “Template” 栏用于键入 “SVCS” 菜单配置文件的名字。



图 7.47 配置之前的 SVCS 菜单

“User Name” 栏用于键入用户名, 以便于登录到 SVCS 系统, 在参数行上它是通过文件码 “%U” 传递的。

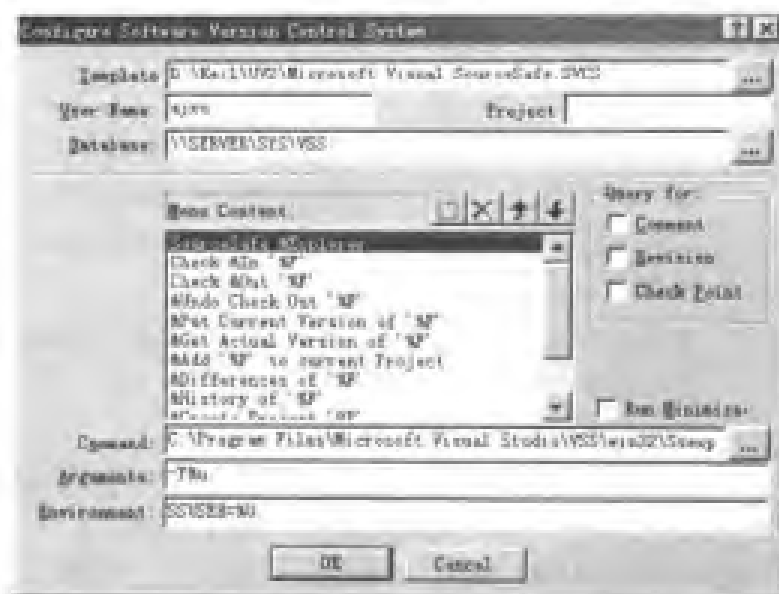


图 7.48 SVCS 的配置窗口

“Database”栏用于键入 SVCS 系统使用的数据库的文件名或路径，在参数行上通过文件码“%V”传递。

“Menu Content”栏用于显示在经过配置之后“SVCS”菜单中的文本行，可以包括键码和文件码，字符“&”用于定义快捷键。

“Query for…”栏有三个复选框：“Comment”、“Revision”和“Check Point”，通过它们允许用户在执行 SVCS 命令前询问一些附加信息。Comment 信息将被复制到一个临时文件中，此文件通过文件码“%Q”作为一个参数传递给 SVCS 命令。Revision 和 Check Point 信息通过文件码“%R”和“%C”作为字符串传递给 SVCS 命令。

“Run Minimized”复选框用于以最小化窗口方式执行命令。

“Command”栏用于键入配置“SVCS”菜单项时将调用的可执行程序文件。

“Arguments”栏用于键入传递给 SVCS 程序文件的命令行参数。

“Environment”栏用于键入在执行 SVCS 程序前需要设置的环境变量。

如果在“Template”栏键入 Keil Cx51 软件包中已有的配置模板文件（例如：D:\Keil\UV2\Microsoft Visual SourceSafe.SVCS），将使配置过程变得十分简单。

图 7.49 所示为经过配置之后的“SVCS”菜单。根据配置的不同，菜单中的选项也会有所不同。包含文件可以作为文档资料添加到项目中，以便 SVCS 可以迅速访问到它们。下面对该“SVCS”菜单中的选项解释如下。

“Source Safe Explorer”：打开交互式的 SVCS 浏览器。

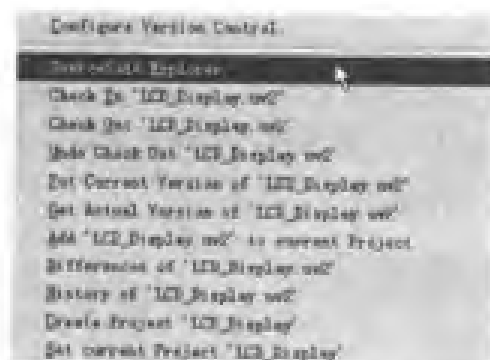


图 7.49 经过配置之后的 SVCS 窗口

- “Check In”: 将文件复制到 SVCS 的数据库中, 并将复制件设置为只读属性。
- “Check Out”: 从 SVCS 中取得最新版本文件, 设置为可修改属性。
- “Undo Check Out”: 取消 Check Out 操作。
- “Put Current Version”: 将文件保存到 SVCS 的数据库中, 对文件仍然可以修改。
- “Get Actual Version”: 从 SVCS 中取得一个当前只读文件。
- “Add file to Project”: 添加文件到 SVCS 项目。
- “Differences 和 History”: 显示关于文件的 SVCS 信息。
- “Create Project”: 创建一个与 μ Vision2 项目文件同名的 SVCS 项目。

需要注意的是, 可以用文本编辑器修改软件包中预先配置好的*.SVCS 文件以调整适当的程序路径和工具参数。若采用 Microsoft SourceSafe 配置 SVCS, 在选定一个新的 μ Vision2 项目后, 要求运行“SET CURRENT PROJECT”命令。若使用工作站的登录名, 要从配置中删除 SSUSER 环境变量。

SVCS 配置完成之后, μ Vision2 项目被保存为两个单独的文件。项目设置保存在*.UV2 文件中, 此文件将被 SVCS 查找, 并且可以用来指导编译生成应用代码; μ Vision2 的当前配置被保存在*.OPT 文件中, 包含对窗口位置和调试信息的设置。

7.1.7 Window 菜单

“Window”菜单如图 7.50 所示。窗口分为两栏, 第一栏中“Cascade”选项用于将已打开的多个编辑窗口按层叠方式显示; “Tile Horizontally”选项将已打开的多个编辑窗口按水平平铺方式显示; “Tile Vertically”选项将打开的多个编辑窗口按垂直平铺方式显示; “Arrange Icons”选项用于图标排列; “Split”选项用于分割当前文件编辑窗口。“Close All”选项将所有打开的文件编辑窗口全部关闭。第二栏保存着当前项目中已打开的文件名, 单击文件名可快速切换到该文件的编辑窗口。

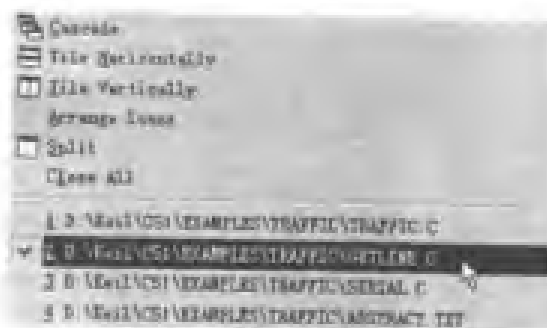


图 7.50 Window 菜单

7.1.8 Help 菜单

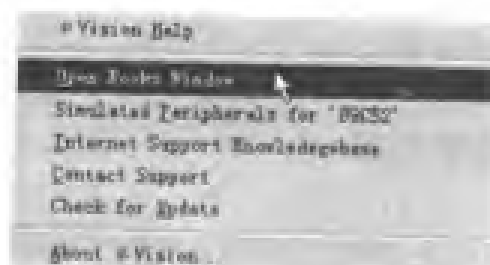


图 7.51 Help 菜单

“Help”菜单如图 7.51 所示。单击第一栏中的“ μ Vision2 Help”选项弹出如图 7.52 所示帮助主题窗口, 键入需要查找的单词或直接单击需要帮助的索引项, 再单击显示按钮立即将对应的帮助信息显示在屏幕上。

第二栏有 5 个选项, 单击“Open Books Window”选项打开项目窗口中的“Books”标签页, 从中可以选择需要察看的用户手册等各种书籍。单击“Simulated Support for ‘CPU’”打开如图 7.53 所示窗口, 从中可以查看能够仿真

的单片机集成外围功能。“Internet Support Knowledgebase”、“Contact Support”和“Check Update”选项,可通过 Internet 连接到 Keil 公司的技术支持网站,从中可以找到许多有用的技术资料。

单击第三栏中的“About μ Vision...”选项将弹出一个有关 μ Vision2 版本信息的窗口。



图 7.52 帮助主题窗口

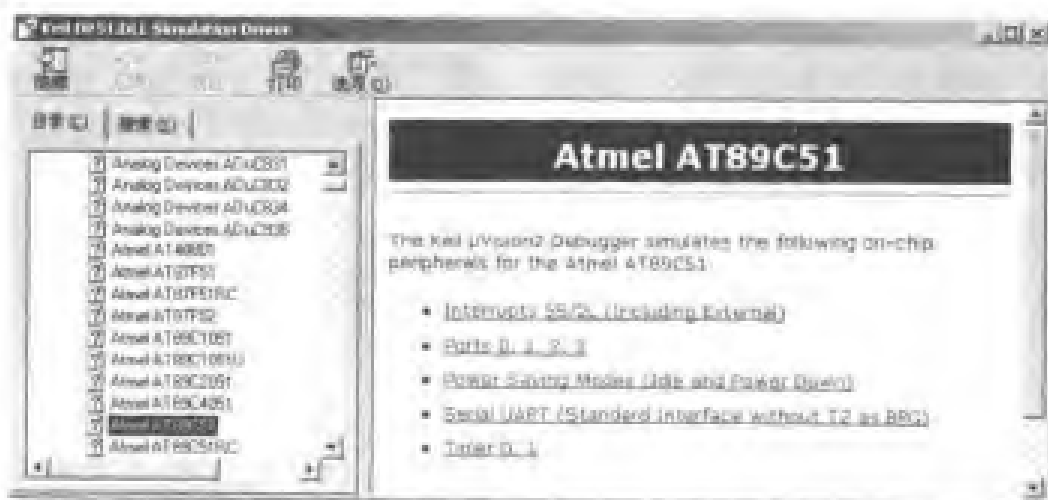


图 7.53 仿真 CPU 集成外围功能信息窗口

7.2 μ Vision2 中的调试器

μ Vision2 中集成了一种新型调试器 (Debug), 它可以进行纯软件模拟仿真和硬件目标板在线仿真, 使用之前应该先进行适当配置。单击“Project 菜单/Options for Target 选项/Debug 标签页”, 弹出如图 7.54 所示窗口。选中圆形单选框“Use Simulator”采用软件模拟方式进行仿真, 可以在没有任何实际 8051 单片机硬件的条件下, 仅用一台普通的 PC 实

现对 8051 应用程序仿真调试。在创建用户项目的时候通过内部器件库选定一种 CPU 器件， μ Vision2 根据所选定的 CPU 器件自动设置能够仿真的单片机片内集成功能，例如，选定 Intel8052 后，可仿真调试 8052 单片机内部定时器 T2，选定 Philips80C552 后，可仿真调试 80C552 单片机内部 A/D 转换器功能等。选中圆形单选框“Use Keil Monitor-51 Driver”采用硬件目标板方式进行仿真，可以与预装有 Keil 公司提供的可配置监控程序 MONITOR51 的用户硬件目标板通过 RS232 与 PC 机连接，实现对硬件目标系统的在线仿真。用户硬件目标板与 PC 机的通信速率可以调整，单击“Setting”按钮弹出如图 7.55 所示窗口，在“Port”栏内键入 PC 机的串口名，在“Baudrate”栏内键入希望采用的通信波特率。窗口右边的“Catch Option”栏有 4 个方形复选框，用于设定调试过程中数据的缓存选项，选中时可以加快数据在 PC 机屏幕上的显示速度，但是若希望直接观察单片机内部定时/计数器、I/O 引脚电平状况以及外部扩展端口实际数据的变化，则不要选中这些复选框。窗口下边的“Stop Program Execution with”栏还有一个复选框“Serial Interrupt”，选中时可以在运行用户程序过程中通过 Debug 调试器的“Stop”按钮或 PC 机键盘的“Esc”键停止用户程序的运行，但这时用户程序中不能使用单片机的串行口，同时不能禁止特殊功能寄存器 IE 中的 EA 位。



图 7.54 Debug 配置窗口

选中方形复选框“Load Application at Startup”，在启动 Debug 时将自动装入用户程序，选中方形复选框“Go till main()”，用户程序将从复位入口一直运行到 main()函数处，通常这两个选项都需要选中以便于调试。在“Initialization”栏内可以键入一个带路径的初始化文件名，该文件的内容为 Debug 调试器的各种调试命令，可以在启动调试时一次执行。单击“Edit...”按钮可以在编辑窗口打开初始化文件进行编辑。



图 7.55 硬件目标板通信端口设置窗口

在“Restore Debug Session Settings”栏中有4个方形复选框:“Breakpoints”、“Watchpoints & PA”、“Memory Display”和“Toolbox”,分别用于在启动 Debug 调试器时自动恢复上次调试过程中所设置的断点、观察点与性能分析器、存储器及工具箱的显示状态,如果希望启动 Debug 仿真调试时能够使用在编辑源程序文件时就设置的断点,应该选中这些复选框。

窗口下边还有几栏:“CPU DLL”、“Driver DLL”、“Dialog DLL”及“Parameter”,它们是根据项目配置时从器件库中所选择的单片机 CPU 器件,由μVision2 自动设置的内部驱动程序及参数,一般不要轻易加以改动。

7.2.1 Debug 状态下窗口分配与 View 菜单操作

Debug 选项配置完成之后,单击“Debug 菜单/Start/Stop Debug Session”选项,即可启动 Debug 开始调试,启动 Debug 后μVision2 窗口分配如图 7.56 所示,项目窗口自动切换到 Regs 标签页,用于显示程序调试过程中单片机内部寄存器状态的变化情况,主调试窗口用于显示用户源程序,窗口左边的小箭头指向当前程序语句,每执行一条语句小箭头会自动向后移动,便于观察程序当前执行点。如果用户创建的项目中包含有多个程序文件,执行过程中将自动切换到不同文件显示。命令窗口用于键入各种调试命令,存储器窗口用于显示程序调试过程中单片机的存储器状态,观察窗口用于显示局部变量和观察点的状态。此外在主调试窗口位置还可以显示反汇编窗口、串行窗口以及性能分析窗口,通过单击 View 菜单中的相应选项(或单击工具条中相应按钮),可以很方便地实现窗口切换。



图 7.56 调试状态下μVision2 窗口分配

Debug 状态下 View 菜单如图 7.57 所示, 菜单第三栏原来以灰色显示的不可用选项现在都成为可用的了, View 菜单的第三栏用于 Debug 状态下各种窗口的显示/隐藏切换。选中“Disassembly Window”选项显示如图 7.58 所示反汇编窗口, 用于显示已经装入到 μ Vision2 的用户程序汇编语言指令、反汇编代码及其地址, 当采用单步或断点方式运行程序时, 反汇编窗口的显示内容会随指令的执行而滚动。在反汇编窗口中可以使用右键功能, 将鼠标指向反汇编窗口并单击右键, 可弹出如图 7.59 所示窗口。该窗口第一栏中的选项用于选择窗口内反汇编内容的显示方式, “Mixed Mode”选项采用高级语言与汇编语言混合方式显示; “Assembly Mode”选项采用汇编语言方式显示; “Inline Assembly...”选项用于程序调试中的“在线汇编”, 方法是先将光标定位在反汇编窗口中的希望行上, 再单击该选项弹出如图 7.60 所示窗口, 其中“Current Instruction”栏显示的是指定行当前的汇编指令, “Current Assembly”栏显示当前指定行的地址, 在“Enter New”栏中可以键入希望的新汇编语言指令 (如 LIMP I2C_STOP), 完成后回车, 新键入的指令将取代指定行上原来的指令, 实现“在线汇编”功能。

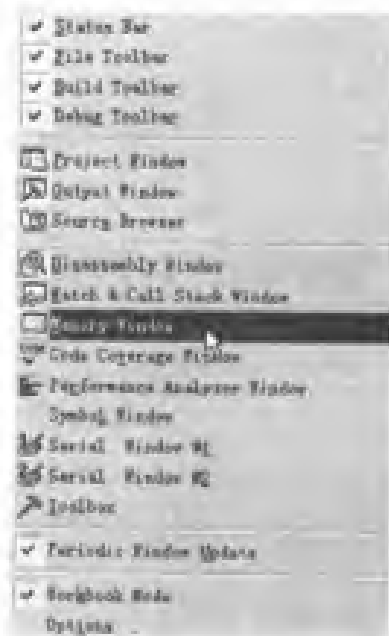


图 7.57 调试状态下 View 菜单

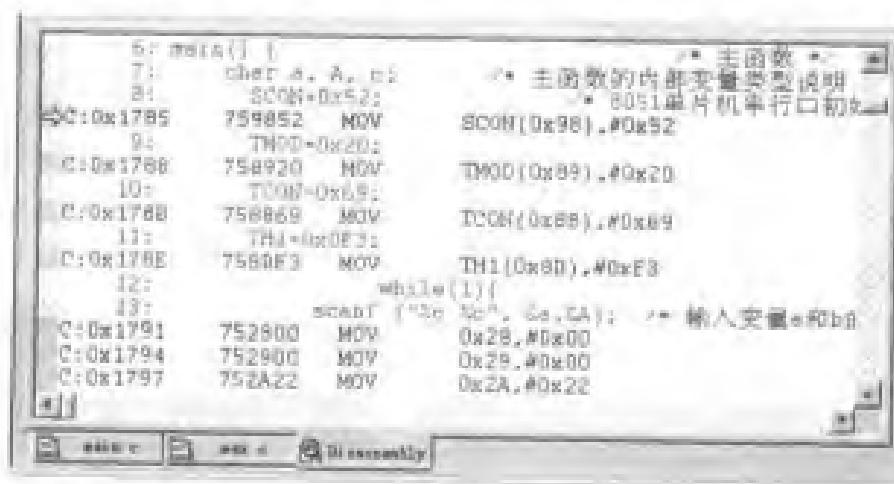


图 7.58 反汇编窗口

右键窗口第二栏的“Address Range”选项用于显示用户程序的地址范围; “Load Hex or Object file...”用于重新装入 Hex 或 Object 文件到 μ Vision2 中进行调试。

右键窗口第三栏的“View Trace Records”选项用于在反汇编窗口显示指令执行的历史记录, 该选项只有在该栏中另一个选项“Enable/Disable Trace Recording”被选中, 并且已经执行过用户程序指令的情况下才能起作用, 这时反汇编窗口上部将以不同颜色显示出已经被执行过指令的历史记录, 将光标向上移动可以察看更多历史记录, 察看历史记录的同时项目窗口“Regs”标签页中的显示内容也随之发生变化。选项“Show next statement”用

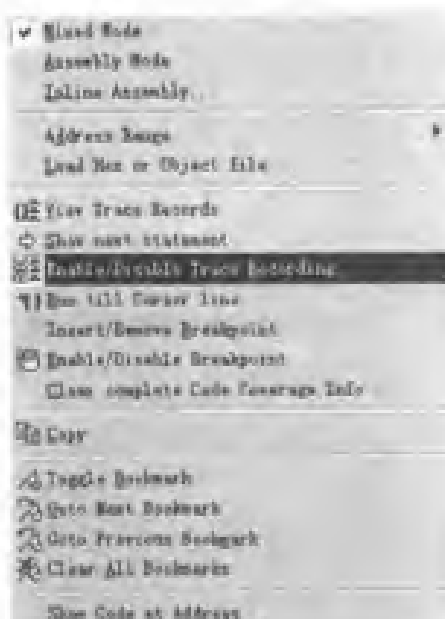


图 7.59 反汇编窗口中的右键菜单

于显示下一条将被执行的指令。选项“Run to Cursor line”用于将程序执行到当前光标所在的那一行，具体方法是先将光标定位在希望的程序行上，然后单击该选项，用户程序会执行到指定行暂停。选项“Insert/Remove Breakpoint”用于插入/删除程序执行时的断点，先将光标定位在希望插入断点的程序行上，单击“Insert/Remove Breakpoint”选项，将在选定行插入/删除一个断点。对于已经存在的断点，单击“Enable/Disable Breakpoint”选项可激活/禁止选定的一个断点。“Clear complete Code Coverage Info”选项用于清零代码覆盖信息（Code Coverage）。

右键窗口第四栏的“Copy”选项用于复制反汇编窗口中的内容，先选定反汇编窗口中需要复制的内容，单击该选项即可将选定的内容复制到 Windows 的剪贴板中，然后再将其粘贴到用户文件中。这一功能对于保存调试过程中采用“在线汇编”临时修改

了的程序文件特别有用，因为“在线汇编”仅在调试程序时对装入到 μ Vision2 中的汇编代码起作用，它对用户源文件并未进行修改，退出调试之后“在线汇编”的作用将不复存在。

右键窗口第五栏用于文本标记操作，其中各个选项与 Edit 菜单中文本标记选项的功能相同。

右键窗口最后一栏的“Show Code at Address...”选项用于显示指定地址处的用户程序代码，单击该选项弹出如图 7.61 所示窗口，在“Enter Address”栏内键入指定的用户源程序的地址值（可以是数字地址或符号地址），单击“Go To”按钮，反汇编窗口将立即显示出指定地址处的用户程序代码。



图 7.60 “在线汇编”窗口



图 7.61 显示指定地址程序代码窗口

View 菜单的第三栏的“Watch & Call Stack Window”选项用于调试状态下观察窗口的显示/隐藏切换，观察窗口有四个标签页，分别是局部变量（Locals），观察 1（Watch1），观察 2（Watch2）以及调用栈（Call Stack），图 7.62 所示为观察窗口的“Locals”标签页，显示用户程序调试过程中当前局部变量的使用情况；图 7.63 所示为观察窗口的“Watch1”标签页，显示用户程序中已经设置了的观察点在调试过程中的当前值；在“Locals”栏或“Watch1”栏中单击鼠标右键可改变局部变量或观察点的值按十六进制（Hex）或十进制（Decimal）方式显示。图 7.64 所示为观察窗口的“Call Stack”标签页，显示程序执行过程中对子程序的调用情况。

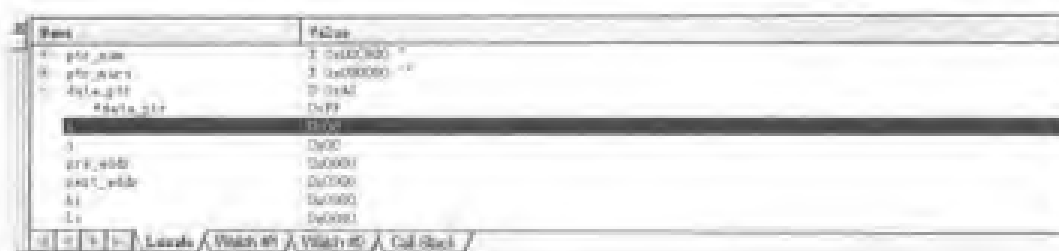


图 7.62 观察窗口的 Locals 标签页



图 7.63 观察窗口的 Watch1 标签页



图 7.64 观察窗口的 Call Stack 标签页

View 菜单的第三栏的“Memory Window”选项用于系统存储器空间的显示/隐藏切换，如图 7.65 所示。在窗口中“Address”处键入存储器地址，将立即显示对应存储器空间的内容。需要注意的是键入地址时要指定存储器类型（c、d、i、x 等）。窗口分为“Memory #1”~“Memory #4”四栏，每栏可指定不同的地址空间。在存储器窗口中单击鼠标右键弹出如图 7.66 所示窗口，用于改变存储器内容按不同方式显示：可以采用十进制（Decimal）、按无符号（Unsigned）或有符号（Signed）的字符类型（char）、整型（int）或长整型（long）、Ascii 码、浮点型（Float）、双精度型（Double）等方式进行显示。右键菜单最后一个选项是对指定的存储器地址内容进行修改，方法是先将鼠标指向希望修改的存储器地址（如 D:0x09A），再单击该选项弹出如图 7.67 所示的数据修改窗口，键入希望修改的新字符或数据值，单击“OK”按钮后新字符或数据值将取代存储器窗口指定地址处原来的字符或数据。

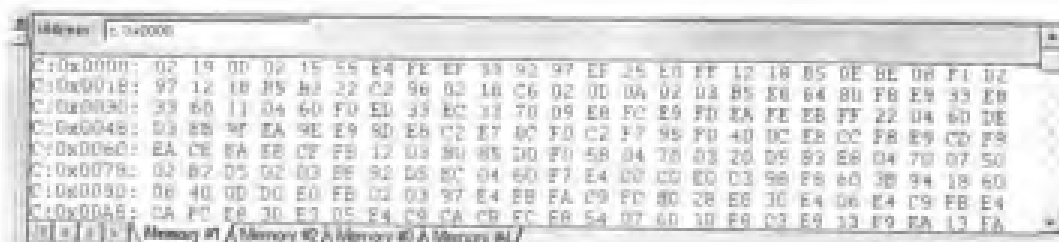


图 7.65 存储器窗口



图 7.66 存储器窗口的右键菜单



图 7.67 数据修改窗口

View 菜单的第三栏的“Code Coverage Window”选项用于代码覆盖窗口的显示/隐藏切换,选中该选项弹出如图 7.68 所示窗口,其中显示的是指定用户程序模块的代码执行情况,在“Current”栏内键入指定的用户程序模块名,该模块中各函数所包含的指令条数及其已经被执行指令的百分数将显示出来。按钮“Update”用于对显示值进行更新,按钮“Reset”用于复位被执行指令的百分数。

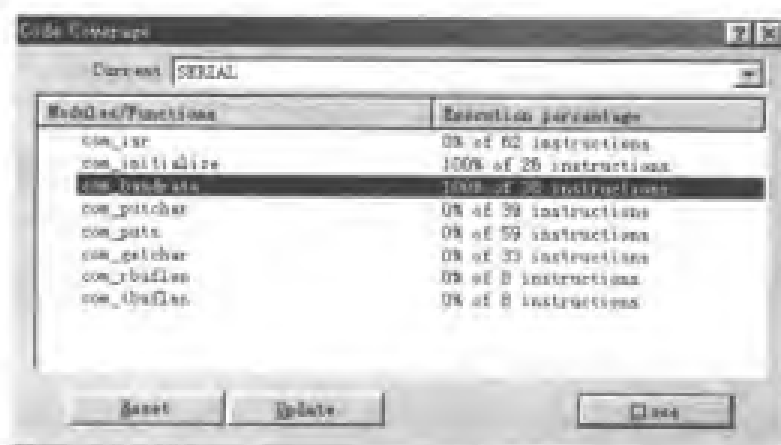


图 7.68 代码覆盖窗口

View 菜单的第三栏的“Performance Analyzer Window”选项用于性能分析窗口的显示/隐藏切换,选中该选项弹出如图 7.69 所示窗口,用于显示指定程序模块被调用的次数及执行时间,分析结果以棒状图形式显示在窗口中,通过窗口内的一个运行性能统计标尺很容易了解某个程序模块的运行性能。窗口下边显示有如下一些信息:

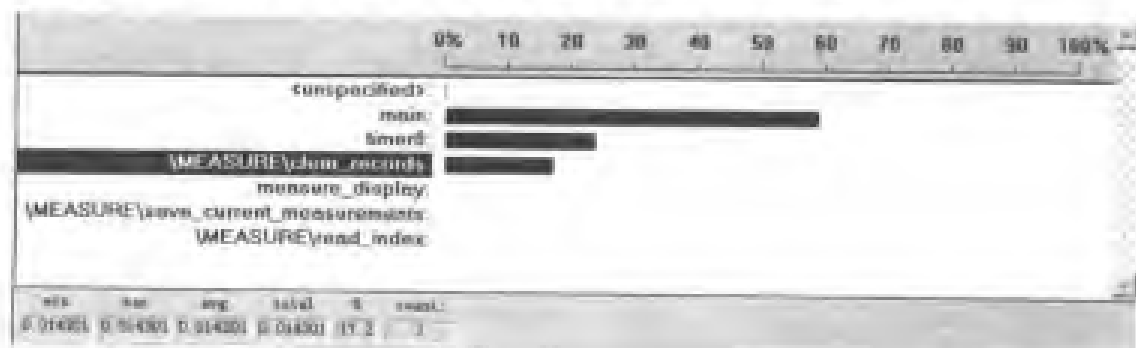


图 7.69 性能分析窗口

min: 指定程序模块所消耗的最小时间。

max: 指定程序模块所消耗的最大时间。

- avg: 指定程序模块所消耗的平均时间。
- total: 指定程序模块所消耗的总时间。
- %: 指定程序模块所消耗的时间占全部运行时间的百分数。
- count: 指定程序模块被调用的次数。

将鼠标指向性能分析窗口单击右键可弹出如图 7.70 所示窗口, 其中选项“Reset PA”用于复位所有程序模块的性能分析值; 选项“Active PA”用于启动/停止性能分析; 选项“Update Times”用于更新程序执行时间; 最后一个选项“Setup PA”用于对性能分析窗口的内容进行设置, 单击“Setup PA”选项弹出如图 7.71 所示窗口, 其中“Current PA Range”栏显示的是当前已经设定的性能分析程序模块, 单击“Kill All”按钮将删除全部已设定的模块, 也可以用鼠标选中其中某一个模块再单击“Kill Selected”按钮删除指定模块; 在“Define Performance Analyzer”栏中键入希望进行分析的程序模块名, 再单击“Define”按钮可将其设定为一个性能分析程序模块; 窗口右边的“Function Symbols”栏中显示了当前所有可用的程序模块, 将鼠标指向其中某个模块名并双击左键, 选定的程序模块名将立即出现在“Define Performance Analyzer”栏中, 再单击“Define”按钮即可将其设定为一个性能分析程序模块。



图 7.70 性能分析右键窗口



图 7.71 性能分析设置窗口

View 菜单的第三栏的“Symbol Window”选项用于符号窗口的显示/隐藏切换, 选中该选项弹出如图 7.72 所示窗口, 显示当前程序中的各种符号。通过窗口左上角“Mode”栏可选择显示公共符号 (Publics)、局部符号 (Locals) 或行号 (Lines); Current 栏用于选择当前程序模块; 在“Mask”栏中键入希望显示的符号名 (可以使用通配符*), 对应的符号信息将立即显示在窗口中。

View 菜单的第三栏的“Serial Window #1”和“Serial Window #2”选项用于串行窗口 1 和串行窗口 2 的显示/隐藏切换, 选中该选项弹出如图 7.73 所示窗口。该窗口在进行用户程序调试时十分有用, 如果用户程序中调用了 C51 的库函数 scanf() 和 printf(), 则必须利用该窗口来完成 scanf() 函数的输入操作, printf() 函数的输出结果也将显示在该窗口中, 利用

串行窗口可在用户程序仿真调试过程实现人机交互对话, 或者对程序运行结果进行实时显示。在串行窗口中单击鼠标右键将弹出一个显示方式选项菜单, 可按需要将窗口内容以 HEX 或 ASCII 格式显示, 也可随时清除显示内容。串行窗口中可保持最近 8KB 串行输入/输出数据, 并可进行翻滚显示。

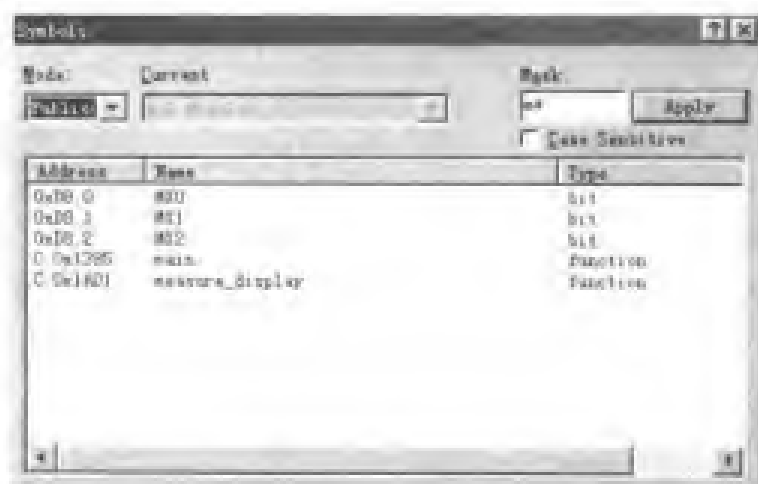


图 7.72 符号窗口

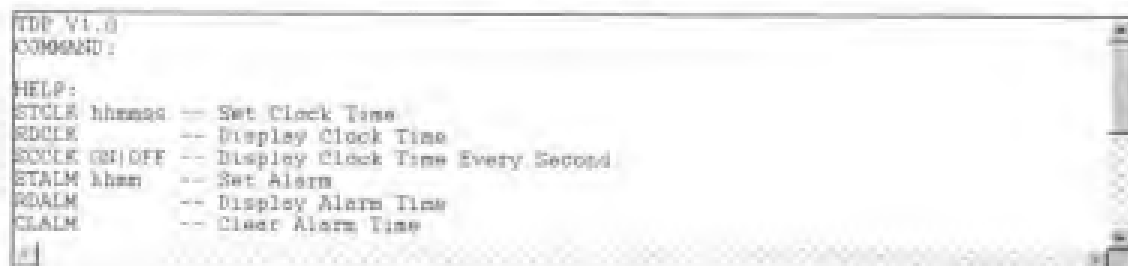


图 7.73 串行窗口



图 7.74 工具箱窗口

View 菜单的第三栏的“Toolbox”选项用于工具箱窗口的显示/隐藏切换, 选中该选项弹出如图 7.74 所示窗口。该窗口可以包含多个命令按钮, 其中“Update Windows”是μVision2 内部预定义按钮, 用户还可以按需要自定义其他命令按钮。每当按钮被按下时它所对应的命令立即被执行, 例如, 单击“Update Windows”按钮将对当前显示内容进行更新。

用户自定义按钮可以在命令窗口中采用“Define Button”命令来实现, 其格式如下:

Define Button “button_label”, “button_command”

其中第一个参数“button_label”是按钮名, 第二个参数“button_command”必须是一个有效的μVision2 命令。

例 7.1: 定义工具箱按钮:

```

>Define Button “go to main” : “g, main”      /* 定义 “go to main” 按钮 */
>Define Button “clr dptr” : “dptr=0”         /* 定义 “clr dptr” 按钮 */
  
```

进行正确定义后,对应的按钮将立即出现在工具盒窗口中,每个用户定义命令按钮前面将自动冠以一个编号,以便于删除该按钮。随着定义按钮数目的增加工具盒窗口将自动增大。在命令窗口采用 Kill Button 命令可以删除用户自定义命令按钮,需要注意的是预定义按钮“Update Windows”前面没有标号,因此它是不能被删除的。

例 7.2: 删除工具盒按钮:

```
>Kill Button 1 /* 删除编号为 1 的自定义按钮 */
>Kill Button 2 /* 删除编号为 2 的自定义按钮 */
```

View 菜单的第四栏的“Periodic Window Update”选项用于对个窗口显示内容进行周期性更新,在进行用户程序调试时若希望从观察窗口中看到某个变量值的改变情况,必须选中该选项。

调试状态下命令窗口如图 7.75 所示,可以通过“View 菜单/Output Window/ Command”标签页打开。窗口中有一个显示提示符“>”的命令输入行,其中可以输入各种命令字,如装入用户程序的目标文件、运行、设置观察点或断点等。在命令输入行上键入命令字并回车,该命令将立即被执行,执行结果也显示在输出窗口中;若命令输入错误,窗口中将显示出错信息。 μ Vision2 内部集成了一个方便实用的命令语法产生器,在输入命令时自动将所有可选的命令字、命令需要的参数等显示在窗口下边的命令提示行上。利用 Tab 键可滚动显示提示行上的内容,键入提示行上某个命令字中的大写字母可直接输入该命令。在命令输入时提示行上的显示内容会自动减少,只剩下一个命令字被保留时,键入空格可立即输入该命令字。一个命令字输入后提示行上还将显示该命令所要求的各种参数,以帮助用户避免出错。



图 7.75 调试状态下的命令窗口

例 7.3: 输入命令:

```
>U 空格
```

即可输入“Unassemble”命令,命令输入后提示行上显示出:

```
<unassemble start address> <cr>
```

提示用户继续输入反汇编起始地址后回车,以完成整个命令的输入。

除了可以执行单个命令之外,还可以将多个命令放在一起组成一个“命令文件”,然后用“INCLUDE”命令一次执行。利用“命令文件”可以很方便地一次性同时装入用户程序目标文件,设定存储器方式,运行程序等。命令文件的扩展名必须为“.ini”,如果希望一次输入并执行下面几条命令:

```
load tdp /* 装入用户目标文件 */
```



```
map 0,0xffff          /* 定义 64K 存储器空间 */
g,main                /* 启动程序运行到标号 main 处 */
```

可将这些命令预先在磁盘上存为一个命令文件“user.ini”，然后在命令行中键入：

```
INCLUDE user.ini <<cr>
```

上述命令将被一次执行。 μ Vision2 允许在输入命令字时带有 C 语言格式的注释。命令行可利用下列控制键来改变命令字的输入：

控制键	功能
Enter	执行所输入的命令
Backspace	删除光标前面的一个字符
Del	删除光标处的一个字符
Esc	中止命令输入并开始新的命令输入行
Home	置光标于输入行首
End	置光标于输入行尾
Tab	滚动显示帮助行上的信息
←	光标左移一格
→	光标右移一格
↑	重复输入命令
↓	重复输入命令

View 菜单的第四栏的“Periodic Window Update”选项用于对各窗口显示内容进行周期性更新，用户在进行程序调试时若希望从观察窗口中看到某个变量值随程序执行而改变的情况，必须选中该选项。

7.2.2 通过 Debug 菜单进行程序代码调试



图 7.76 Debug 菜单

在 μ Vision2 调试器中可以进行两种类型的代码调试：带调试信息的源程序代码调试和十六进制 HEX 代码调试，前者允许调试过程中显示高级语言源程序语句，后者仅能显示基本汇编语言指令。用户在成功完成项目编译连接之后，通过 Debug 菜单进入程序调试状态，在调试状态下仍可通过调试主窗口进行源程序的编辑修改，这也是 μ Vision2 调试器的一大特点，可以根据当前调试结果修改源程序，不过修改后的源程序不能立即进行调试，要先退出当前调试状态，重新编译连接成为新的目标代码再次装入之后才能调试。

Debug 菜单如图 7.76 所示，第一栏“Start/Stop Debug Session”（快捷键 Ctrl+F5）选项用于启动 / 停止调试功能，启动之前应先确定是

采用软件模拟仿真还是采用硬件目标板仿真,采用软件模拟仿真时,应在“Project 菜单/Options for Target 选项/Debug”标签页中选定“Use Simulator”圆形单选框,采用硬件目标板仿真时,目标板上必须装有 Keil 公司提供的监控程序,并在“Project 菜单/Options for Target 选项/Debug”标签页中选定“Use Keil Monitor-51 Driver”圆形单选框。此外不论采用哪种仿真方式,都应当选定“Load Application at Start”和“Go till main”方形复选框,这样在启动调试功能时 μ Vision2 将自动装入用户程序的目标代码并运行到主函数 main 所在的行上。启动调试功能之后 μ Vision2 的项目窗口(Project Windows)自动切换到 Regs 标签页,显示 CPU 内部各寄存器的当前状态,寄存器状态将随着程序语句或指令的执行而变化。

Debug 菜单第二栏各选项用于控制目标代码的执行方式。“Go”(快捷键 F5)选项用于启动用户程序从当前地址处开始全速运行,遇到断点或是执行“Stop Running”选项时停止。“Step”(快捷键 F11)选项用于单步执行,在高级语言显示方式下单击一次该选项执行一条 C 语言语句,在汇编语言显示方式下则执行一条 8051 指令,遇到函数调用(高级语言方式)或子程序调用(汇编语言方式)语句,将跟踪进入函数或子程序中执行。“Step Over”选项用于启动用户程序从当前地址处开始执行一条语句,对于函数或子程序调用语句不跟踪进入被调用函数,而是将整个函数或子程序与调用语句一起一次执行。“Step Out of current Function”(快捷键 Ctrl+F11)选项用于在调用函数或子程序过程中,启动函数或子程序从当前地址处开始执行并返回到调用该函数或子程序的下一条语句,该选项对于非函数或子程序以及采用硬件目标板仿真时无效。“Run to Cursor Line”(快捷键 Ctrl+F10)选项用于启动用户程序从当前地址处开始执行到光标所在行。“Stop Running”选项用于停止运行用户程序。

“Debug”菜单第三栏各选项用于调试用户程序过程中的断点管理。断点功能对于用户程序的仿真调试是十分重要的,它可在某个特定地址或是满足某种特定条件下暂停用户程序运行,以便于观察了解程序的运行状况,查找或排除错误。单击该栏中“Breakpoints...”选项将弹出如图 7.77 所示的断点设置窗口。窗口中“Current Breakpoints”栏显示当前已经设置的断点列表, μ Vision2 自动为每个断点分配一个内部序号,序号前面有一个复选框,选中时使能该断点,否则禁止该断点,禁止断点不同于删除断点,它仅仅暂时禁止断点发挥作用,需要时可以重新激活该断点,让它重新发挥作用。“Expression”栏用于键入断点表达式。“Count”栏用于键入断点通过次数,例如,当“count”的值等于 2 时,表示在第 2 次运行到该断点时停止程序运行或执行规定的命令。“Command”栏用于键入当程序执行到断点时需要执行的命令。

有三种不同类型的断点:执行断点(Execution Break)、条件断点(Conditional Break)和存取断点(Access Break),它们在使用中各有优缺点。 μ Vision2 可以根据断点表达式对不同断点进行自动分类,执行断点前面冠以字母(E),条件断点前面冠以字母(C),存取断点前面冠以字母(A)。“Access”栏用于规定断点的存取类型,选中“Read”复选框表示读,选中“Write”复选框表示写,同时选中这两个复选框表示读写,“Size”栏用于规定存取断点的长度,选中“Bytes”复选框时按断点表达式的值从第一个地址开始计算其字节长度,选中“Objects”复选框时按断点表达式的值计算其总长度。

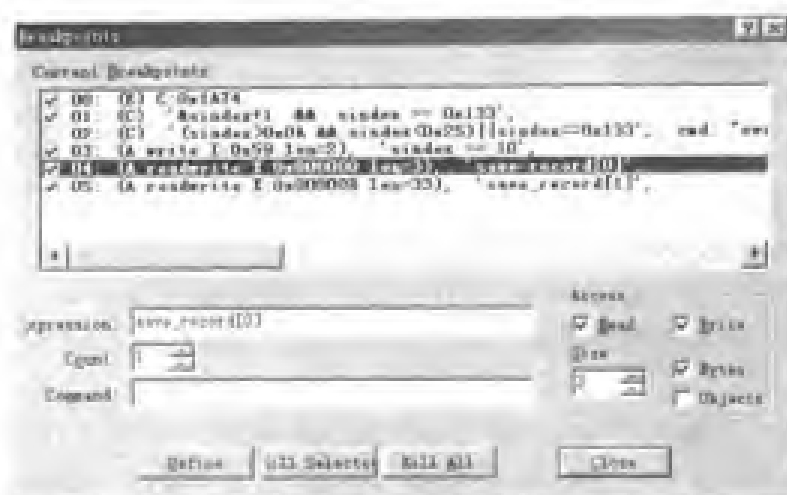


图 7.77 断点设置窗口

若断点表达式为一个特定代码地址（如函数地址或语句行号），则为执行断点，程序每当运行到该地址时被暂停。执行断点必须设置在正确的代码地址处（8051 单片机指令的第一个代码字节地址），对于某些代码地址只允许设置一次断点，不能重复定义。利用执行断点可以检查程序流程是否正确。若断点表达式不是代码地址时即为条件断点。对于条件断点，每一条指令执行结束后重新计算其逻辑表达式，若计算逻辑结果为假（0 值），继续执行程序，若计算逻辑结果为真（非 0 值），则暂停执行程序。利用条件断点可以检查用户程序中的一些特殊错误，但可能会降低程序的执行速度。带有存取类型（Read、Write）标志的断点为存取断点，用于捕捉对于非法存储器地址的存取操作。

断点可以带有命令串和一个计数值。命令串用于规定当程序到达断点处时自动执行的 μVision2 命令序列（包括 μVision2 函数），执行完规定的命令后继续执行用户程序；若希望执行完规定的命令后在断点处暂停执行用户程序，应在命令串中增加一个系统变量表达式：_BREAK_=1。计数值用于规定暂停执行程序时该断点出现的次数，默认计数值为 1。

用鼠标左键双击“Current Breakpoints”窗口中某个断点，与该断点有关的信息将在窗口下边的相关栏中显示出来。选中某个断点后单击“Kill Selected”按钮，可立即删除该断点。单击“Kill All”按钮将删除全部断点。

设置新断点时，可直接在“Expression”栏内键入断点表达式，在“Command”和“Count”栏键入有效的 μVision2 命令串和计数值（如果 Count 栏为空，则断点计数值默认为 1），对于存取断点还需要在“Access”栏选定相应的存取属性（Read、Write），然后单击“Define”按钮，新断点即设置完成。需要注意的是，当“Command”栏中键入了命令串时，该命令串将在用户程序到达断点时被立即执行，而程序并不在断点处暂停，若需要在断点处暂停用户程序，必须在命令串中增加表达式：_BREAK_=1；断点命令串为可选项，并非每个断点都需要命令串，因此“Command”栏可以为空，此时每当用户程序执行到断点时都将会暂停。

下面举几个设置断点的子。

例 7.4: 设置执行断点。

Expression 栏键入: main
Command 栏键入: printf ("main has been reached\n"), _BREAK_=1
Count 栏键入: 1
单击“Define”按钮, 结束。

本例在用户程序中标号“main”处设置一个执行断点, 当程序执行到标号“main”时将自动执行命令串“printf (“main has been reached\n”);”, 同时在 μ Vision2 的命令窗口内输出命令串执行结果, 由于命令串中存在表达式“_BREAK_=1”, 因此用户程序在断点处被暂停。

例 7.5: 设置执行断点。

Expression 栏键入: \MEASURE\110
Command 栏:
Count 栏键入: 1000
单击“Define”按钮, 结束。

本例在程序模块 MEASURE 的行号 110 处设置一个执行断点, 计数值为 1000, 当程序在第 1000 次执行到该断点地址时将被暂停, 断点一旦起作用之后计数值将变为 1。注意, 以上两个例子都不是存取断点, 存取属性复选框 (Read、Write) 均未选中。

例 7.6: 设置条件断点。

Expression 栏键入: (sindex > 0x0A && sindex < 0x25) || sindex == 0x133
Command 栏键入: eval sindex
Count 栏键入: 1
单击“Define”按钮, 结束。

本例当满足条件: $0x0A < sindex < 0x25$ 或 $sindex == 0x133$ 时用户程序将被暂停, 并执行命令串“eval sindex”, 命令执行结果显示在 μ Vision2 的命令窗口中, 执行完命令串后继续运行用户程序。

例 7.7: 设置条件断点。

Expression 栏键入: \$ == timer0 && sindex > 5
Command 栏:
Count 栏键入: 1
单击“Define”按钮, 结束。

本例当满足条件: 用户程序执行到函数 timer0() 并且当变量 sindex > 5 时, 暂停执行用户程序 (符号“\$”表示程序计数器的值, 表达式中应避免对当前程序计数器进行赋值操作, 如 \$=timer0 将出现错误)。注意, 以上两个例子中若选中存取属性复选框 (Read、Write), 将被设置为存取断点。

设置存取断点时其表达式必须表示一个存储器地址和存储器类型, 并且要遵循以下两条规则:

- ① 表达式结果必须具有唯一的存储器类型。
- ② 只允许使用 &、&&、<、<=、>、>=、==、!= 等运算符。

例 7.8: 设置存取断点。

Expression 栏键入: `sindex && sindex == 0x133`

Command:

Access 栏选中: `Write`

Count 栏键入: `1`

单击“Define”按钮, 结束。

本例希望当用户程序向变量 `sindex` 进行写入, 且当变量 `sindex` 的值为 `0x133` 时暂停执行。

除了可以通过上述断点设置窗口来设置断点外, 还可以直接通过“Debug”下拉菜单来设置断点。进入调试状态后单击“Debug”菜单第三栏中“Insert/Remove Breakpoint”选项, 可在编辑窗口当前光标所在文件行上插入 / 删除一个断点; 单击“Debug”菜单第三栏中“Enable/Disable Breakpoint”选项, 可激活 / 禁止当前光标所指向的一个断点; 单击“Debug”菜单第三栏中“Disable All Breakpoints”将禁止所有已经设置的断点; 单击“Debug”菜单第三栏中“Kill All Breakpoints”将删除所有已经设置的断点; 单击“Debug”菜单第三栏中“Show Next Statement”选项将在编辑窗口中显示下一条将要被执行的用户程序语句。

“Debug”菜单第四栏中的“Enable/Disable Trace Recording”选项用于激活 / 禁止程序调试时跟踪指令执行的历史记录, 在该选项被激活并且已经执行过用户程序的情况下, 单击第四栏中的“View Trace Records”选项, 可以从汇编窗口察看源程序指令执行的历史记录, 汇编窗口上半部分以不同颜色显示出几条指令执行的历史记录, 将光标向上移动可以看到更多的历史记录, 察看历史记录时寄存器窗口的值也将同时变化, 该功能对于分析和优化用户程序特别有用。

“Debug”菜单第五栏中的“Memory Map...”选项用于在进行软件模拟仿真时设置存储器空间映像, 单击该选项弹出如图 7.78 所示的存储器映像设置窗口。窗口中的“Current Mapped”栏显示所有已被设置了存储器空间映像, 选中其中某一项并单击“Kill Selected Range”按钮将删除选定的存储器空间映像, 删除后用户程序不能对该部分存储器进行操作, 否则将在μVision2 的命令窗口显示出“access violation”错误。



图 7.78 存储器映像设置窗口

存储器映像设置窗口中的“Map Range”栏用于键入希望设置存储器空间映像的地址范围，格式如下：

字母前缀：起始地址，字母前缀：终止地址

μ Vision2 的存储器空间映像可高达 16MB，采用以下字母前缀来区分不同的存储器空间：

字母前缀	存储器空间
I:	256 字节内部数据存储器 (DATA、IDATA) 空间，地址范围为 0x00~0xFF；
X:	64K 字节外部数据存储器 (XDATA) 空间，地址范围为 0x0000~0xFFFF； 16M 字节外部数据存储器 (HDATA) 空间，地址范围为 0x000000~0xFFFFF；
C:	64K 字节代码存储器 (CODE) 空间，地址范围为 0x0000~0xFFFF； 16M 字节代码存储器 (ECODE) 空间，地址范围为 0x0000~0xFFFFF；

B0~B31: 分组代码空间，地址范围为 B0: 0x0000~0xFFFF~B31: 0x0000~0xFFFF；

μ Vision2 除了可以设置以上标准单片机存储器空间映像之外，还可以设置 4 个用户定义存储器空间映像：

S:	用户定义存储器空间，地址范围为 0x0000~0xFFFF；
T:	用户定义存储器空间，地址范围为 0x0000~0xFFFF；
U:	用户定义存储器空间，地址范围为 0x0000~0xFFFF；
V:	用户定义存储器空间，地址范围为 0x0000~0xFFFF。

存储器空间映像还可带有存取属性 Read、Write、Execute 和 von Neumann，带有属性的存储器空间映像只能进行与之相应操作。内部数据存储器空间具有默认的 Read、Write 属性。von Neumann 用于将外部数据存储器与代码存储器相重叠，此时对于任何外部数据存储器空间 (XDATA) 的存取操作同时发生在代码空间 (CODE)，需要注意的是设置 von Neumann 存储器映像应同时具有 Read、Write 属性，代码存储器 (CODE) 空间不能被设置为 von Neumann 属性。单击“Map Range”按钮立即完成存储器空间映像设置。

虽然 μ Vision2 支持 16M 字节的存储器空间映像，但通常只需要对用户程序所使用的存储器空间范围进行设置，存储器空间映像设置得过大将降低 μ Vision2 的运行速度。启动调试功能 (Debug) 并装入用户目标程序之后， μ Vision2 将自动按当前项目创建时对存储器的要求设置存储器空间映像，一般来说自动设置的存储器空间映像已经可以满足程序调试要求，只有在一些特殊场合才需要重新进行设置。

例 7.9: 设置内部数据存储器空间映像。

Map Range 栏键入: I:0x00, I:0xFF
存取属性: Read、Write

例 7.10: 将外部数据存储器 (XDATA) 空间 0x8000~0xFFFF 与代码存储器 (CODE) 空间 0x8000~0xFFFF 相重叠。

Map Range 栏键入: X:0x8000, X:0xFFFF
属性: Read、Write、von Neumann

上例中，在外部数据存储器地址范围 0x8000~0xFFFF 内进行读写操作时，与在代码存储器地址范围 0x8000~0xFFFF 内进行操作的效果相同。

“Debug”菜单第五栏中的“Performance Analyzer...”选项，用于设定需要进行性能分析的某段程序地址范围（简称 PA 范围），关于 Performance Analyzer 的详细设置方法请参阅前面 7.1.3 节中的有关内容。

“Debug”菜单第五栏中的“Inline Assembly...”选项，用于程序调试过程中的在线汇编，在线汇编的设置方法与在汇编语言窗口用右键菜单方法一样，请参阅前面一节相关内容。

“Debug”菜单第五栏中的“Function Editor(Open Ini File)...”选项，用于编辑或创建 μ Vision2 的初始化文件（Ini File），初始化文件中可以包含各种 μ Vision2 命令及调试函数（Debug Function）。单击该选项弹出如图 7.79 所示窗口，窗口中“Open”按钮用于打开已有的 Ini 文件进行编辑，“New”按钮用于创建一个新的 Ini 文件；“Save”和“Save as...”按钮用于保存当前编辑的 Ini 文件；“Compile”按钮用于编译并加载当前编辑的 Ini 文件，如果发生编译错误，将在“Compile”栏显示相应的错误号，同时光标自动跳转到第一个错误所在的行上，以利于编辑修改，对于 Ini 文件中包含的 μ Vision2 命令，编译时将会自动执行。

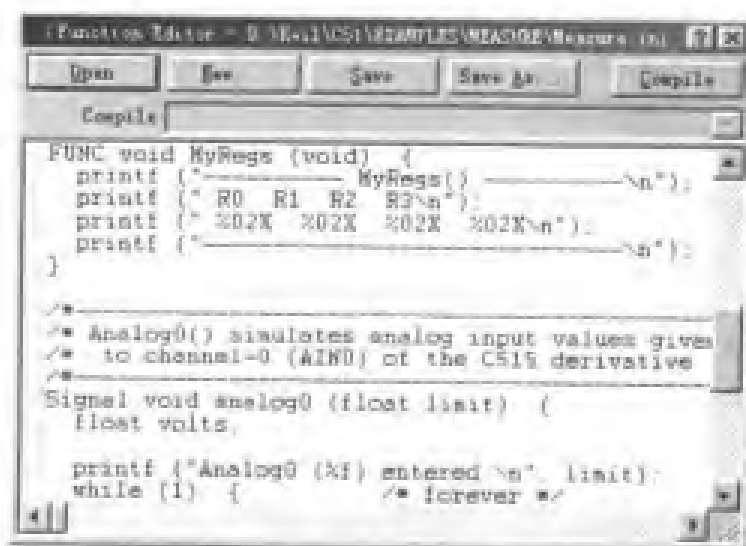


图 7.79 Ini 文件编辑窗口

7.2.3 通过“Peripherals”菜单观察仿真结果

目前 8051 单片机已有 400 多个品种和型号，不同型号单片机具有不同的外围集成功能（Peripherals）， μ Vision2 通过内部集成器件库实现对各种单片机外围集成功能的模拟仿真，在调试状态下可以通过“Peripherals”下拉菜单来观察仿真结果。

“Peripherals”菜单如图 7.80 所示，它的选项内容会根据选用器件库中不同器件而有所变化，图 7.80 所示为选用 8052 单片机器件后的“Peripherals”菜单内容。

单击“Peripherals”菜单第一栏“Reset CPU”选项可以对模拟仿真的 8051 单片机进行复位。

单击“Peripherals”菜单第二栏中的“Interrupt”选项



图 7.80 Peripherals 菜单

将弹出如图 7.81 所示的窗口,用于显示 8051 单片机中断系统状态。选中不同的中断源,窗口中“Selected Interrupt”栏将出现与之相对应的中断允许和中断标志位的复选框,通过对这些状态位的置位和复位操作(选中或不选中)很容易实现对单片机中断系统的仿真。对于具有多个中断源的单片机如 80552 等,除了如上所述几个基本中断源之外,还可以对其他中断源如监视定时器(Watchdog Timer)等进行模拟仿真。

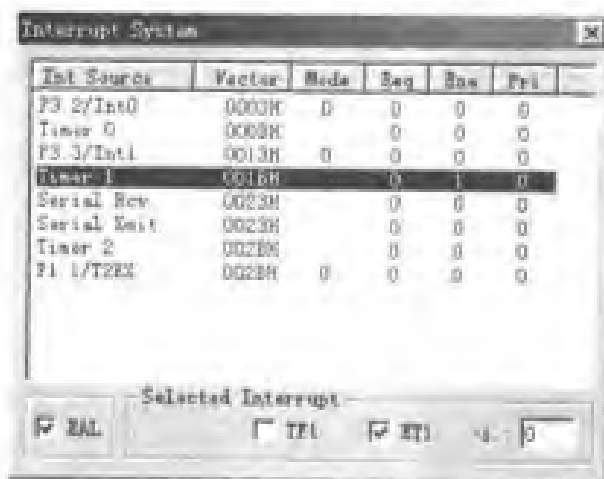


图 7.81 中断系统状态窗口

“Peripherals”菜单第二栏中 I/O-Ports 选项用于仿真 8051 单片机的并行 I/O 接口 Port0~Port3, 选中 Port1 后将弹出如图 7.82 所示窗口, 其中“P1”栏显示 8051 单片机 P1 口锁存器状态, “Pins”栏显示 P1 口各个引脚的状态, 仿真时它们各位的状态可根据需要进行修改。对于具有多个 I/O 口的单片机如 80552 等, 其中 Port0~Port3 都是一样的, 其他 I/O 口略有不同。



图 7.82 Port1 窗口

“Peripherals”菜单第二栏中“Serial”选项用于仿真 8051 单片机的串行口, 单击该选项将弹出如图 7.83 所示窗口。窗口中“Mode”栏用于选择串行口的工作方式, 单击其中的箭头很容易选择 8 位移位寄存器, 8 位/9 位可变波特率 UART, 9 位固定波特率 UART 等不同工作方式。选定工作方式后相应特殊工作寄存器 SCON 和 SBUF 的控制字也显示在窗口中。通过对特殊控制位 SM2、REN、TB8、RB8、TI 和 RI 复选框的置位和复位操作(选中或不选中), 很容易实现对 8051 单片机内部串行口的仿真。“Baudrate”栏用于显示串行口的工作波特率, SMOD 位置位时将使波特率加倍, “IRQ”栏用于显示串行口的发送和接收中断标志。

“Peripherals”菜单中的“Timer”选项用于仿真 8051 单片机内部定时器/计数器, 选中其中 Timer1 后将弹出如图 7.84 所示窗口。窗口中“Mode”栏用于选择工作方式, 可选择定时器或计数器方式, 单击其中的箭头很容易实现选择, 图 7.84 所示为 16 位定时器工作方式。选定工作方式后相应特殊工作寄存器 TCON 和 TMOD 的控制字也显示在窗口中, TH1 和 TL1 用于显示计数初值, T1 Pin 和 TF1 复选框用于显示 T1 引脚和定时器/计数器的

溢出状态。窗口的“Control”栏用于显示和控制定时器和计数器的工作状态（Run 或 Stop），“TR1”、“GATE”和“INT1#”复选框是启动控制位，通过对这些状态位的置位和复位操作（选中或不选中）很容易实现对 8051 单片机内部定时器/计数器仿真。对于具有多个定时器/计数器的单片机如 80552 等，其 Timer0 和 Timer1 与 8051 是一样的，其他如监视定时器（Watchdog Timer）等状态和控制略有不同。

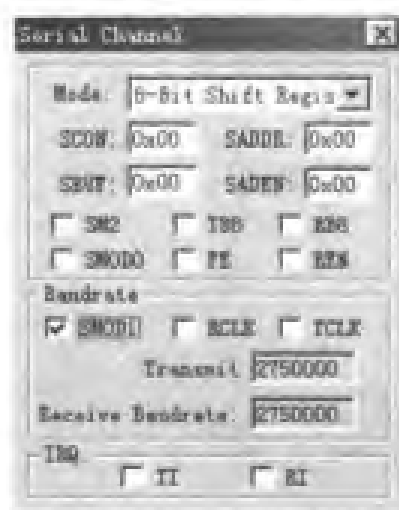


图 7.83 串行口窗口

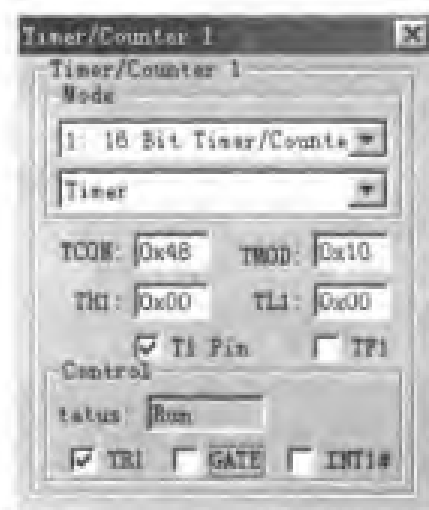


图 7.84 Timer1 窗口

7.3 µVision2 的调试命令

进入µVision2 的仿真状态后，可以在命令窗口中直接执行各种调试命令，也可以将多个命令集中于一个 Ini 文件中，通过调用该文件一次执行多个命令。主要有 4 种类型的命令：显示和更新存储器内容命令、用户程序的执行控制命令、断点管理命令以及其他一些通用命令等。下面分别介绍这些命令的功能与使用方法。

7.3.1 显示和更新存储器内容命令

1. ASM（汇编）命令

命令格式： ASM [起始地址]
ASM 汇编指令

功能说明：带起始地址参数的 ASM 命令用于指定汇编指令地址，其中起始地址为可选项，省略时仅显示当前汇编指令地址。带汇编指令的 ASM 命令用于在当前地址处对输入指令进行汇编，汇编结果显示在反汇编窗口中，同时代码存储器窗口的内容也随之改变，汇编结束后当前地址按输入指令字节数自动增加。汇编指令中的操作数可以是表达式，并可带有行号或标号。还可以通过在线汇编对话框窗口（Inline Assembly...）直接输入汇编指令。

例 7.11:

```

>ASM C:0x0000      /* 将汇编指令地址指定为 C:0x0000 */
>ASM mov a,#12      /* 从 C:0x0000 处开始输入指令 */
>ASM mov r0,#0x20
>ASM movx @r0,A
>ASM inc r0
>ASM movx @r0,A
>ASM jmp C:0x8000
>ASM C:0020H        /* 将汇编指令地址指定为 C:0x0020 */
>ASM CLR A          /* 从 C:0x0020 处开始输入指令 */

```

2. DEFINE (定义) 命令

命令格式: **DEFINE** <类型> <变量名>

功能说明: **DEFINE** 命令用于创建一个带类型的变量, 该变量可以被赋值。用这种方式创建的变量可以用来装载 μ Vision2 函数的返回值, 或者指定 μ Vision2 函数的输入值。

用 **DEFINE** 命令创建的变量并不占用存储器空间, 它们只不过是具有指定类型的符号名, 可以像任何其他公共符号一样使用。命令格式中的<类型>可为:

```

CHAR      /* 带符号字符型 */
DOUBLE    /* 双精度浮点型 */
FLOAT     /* 单精度浮点型 */
INT       /* 带符号整型 */
LONG      /* 带符号长整型 */

```

<变量名>为所定义的符号名, 它应遵循一般变量的命名规则。

例 7.12:

```

>DEFINE CHAR TmpByte /* 定义 TmpByte 为无符号字符型变量 */
>DEFINE FLOAT TmpFloat /* 定义 TmpFloat 为浮点型变量 */
>TmpFloat = 3.14159 /* 给 TmpFloat 赋值 */
>TmpFloat /* 显示 TmpFloat 的值 */
3.14159

```

3. Display (显示) 命令

命令格式: **D** <类型> [起始地址[, 结束地址]]

功能说明: **D** 命令按字节显示某个存储器空间内指定地址范围的内容, 若省略命令中的地址选项, 则从前面一个 **D** 命令的结束地址处开始显示。如果前面并没有使用过 **Display** 命令, 将从 0x0000 地址开始显示。命令执行时若已经打开了存储器窗口, 执行结果将显示在存储器窗口中, 否则将显示在命令窗口中。起始地址中必须包含一个惟一的存储器类型, 以便选择 8051 单片机的有效地址空间, 如果是符号地址则可不必给出存储器类型, μ Vision2 将根据该符号地址自动确定存储器空间。有效的存储器类型如下:

```

X:      XDATA (外部数据存储器)
D:      DATA (内部可直接寻址的数据存储器)

```

I: IDATA (内部可间接寻址的数据存储器)
 B: BIT (位寻址区或特殊功能位)
 C: CODE (程序存储器)
 CO: CONST (常量存储器)
 EB: EBIT (扩展位寻址的数据存储器)
 HC: HCONST (扩展常量存储器)
 ED: EDATA (扩展内部直接寻址数据存储器)

例 7.13:

```
>D main          /* 从 main 地址处开始显示 */
>D menu          /* 从 menu 开始显示      */
>D X:0,0x100      /* 显示 XDATA 存储器中从地址 0000H 开始的 256 个字节 */
>D save_record, save_record + 0x2F /* 显示指定符号地址范围的内容 */
X:4000 62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 bbbbbbbbbbbbbbbbbbb
X:4010 62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 bbbbbbbbbbbbbbbbbbb
X:4020 62 62 62 62 62 62 00 00-00 00 00 00 00 00 00 bbbbbbb.....
```

4. Enter (输入) 命令

命令格式: E <类型> 地址 = 表达式 [, 表达式 [...]]

功能说明: Enter 命令用于接受从指定地址开始输入的存储器内容。命令中的“地址”必须包含一个惟一的存储器类型,以便选择 8051 单片机的有效地址空间,赋值号“=”后面可跟多个表达式,表达式之间由逗号“,”隔开,各个表达式的值按指定类型自动转换并按顺序依次存入以指定地址的存储器空间。命令格式中的<类型>可为以下数据类型:

CHAR /* 有符号或无符号字符型 */
 DOUBLE /* 双精度浮点型 */
 FLOAT /* 浮点型 */
 INT /* 有符号或无符号整型 */
 LONG /* 有符号或无符号长整型 */

例 7.14:

```
>E CHAR x:0 = 1,2,"- μVision2-" /* 从 XDATA 空间地址 0000H 开始输入字符型数据 */
>D x:0 /* 显示输入值 */
X:0000 01 02 2D 64 53 63 6F 70 65 2D 00 00 00 00 00 00 ...-μVision2-.....
>E float x:0x2000 = 3.14,0.33 /* 从 XDATA 空间地址 2000H 开始输入浮点型数据 */
```

5. EVALUATE (评估) 命令

命令格式: EVAL 表达式

功能说明: 该命令计算指定表达式的值并以十进制、八进制、十六进制或 ASCII 码形式输出计算结果。表达式中可以包含若干个用逗号隔开的子表达式。

例 7.15:

```
>EVAL -1
65535T 177777Q FFFFH '.....'
```

```

>EVAL intcycle
0T 0Q 0H '.....'
>intcycle=0x12
>EVAL intcycle
18T 22Q 12H '.....'
>EVAL 'a'+'b'+'c'
294T 446Q 126H '...&'

```

6. MAP (存储器映像) 命令

命令格式: MAP 起始地址,结束地址 [READ WRITE EXEC] [VNM] [CLEAR]

RESET MAP

功能说明: MAP 命令用于进行存储器映像分配。在大多数情况下 μ Vision2 会使用目标程序中的变量信息自动分配存储器映像,MAP 命令定义程序中需要使用但 μ Vision2 不会自动检测的存储器。MAP 命令可以指定一个地址段,并指明存取方式:读(Read)、写(Write)和执行(EXEC)。 μ Vision2 支持单字节存储器映像,也就是说可以将总共 16M 字节存储器空间中的每一个字节都指定具有不同的存储属性。存储属性选项“VNM”用来定义“Von Neumann”结构的外部存储器空间,即外部数据空间与代码空间相重叠,其中既可存放数据也可存放程序代码,“Von Neumann”结构的外部数据存储器空间不得超过 64K 字节。“CLEAR”选项清除指定地址范围内的存储器映像。

尽管 μ Vision2 最大可以支持 16MB 的目标程序存储器映像,但分配大容量的存储器映像可能会减慢 μ Vision2 的运行速度。 μ Vision2 对 8051 单片机存储器空间的默认分段设置如下:

段码(首字节)	段空间	地址范围
0x00	内部数据空间(DATA)段	0x000000~:0x0000FF
0x01	外部数据空间(XDATA)段	0x010000~0x01FFFF
0x80-0x9F	分组代码空间(CODE BANK)段	0x80 代码组 0, 0x81 代码组 1,...
0xFE	分页外部数据空间(PDATA)段	0xFE0000~0xFE00FF
0xFF	代码空间(CODE)段	0xFF0000~0xFFFFF

启动 μ Vision2 调试器后将自动按以下默认方式进行存储器映像分配:

0x000000~0x00FFFF (DATA) READ WRITE

0x010000~0x01FFFF (XDATA) READ WRITE

0xFF0000~0xFFFFF (CODE) EXEC READ

不带任何选项参数的 MAP 命令用来显示当前外部数据存储器的分配情况。

“RESET MAP”命令用来将已分配的存储器映像复位为默认方式。

还可以单击“Debug 菜单/Memory Map”选项打开存储器映像窗口查看和改动存储器映像分配情况。

对于 8051 单片机来说,分配小于 256 字节的外部数据区存储器映像(XDATA)可能会产生问题。8051 通过使用 MOVX @Ri 指令来实现 256 字节的外部数据区分页快速存取,对于大于 256 字节的 XDATA, μ Vision2 使用 P2 口和 P0 口输出 16 位外部地址。对于少于 256 字节的 XDATA, μ Vision2 仅使用 P0 口的低 8 位地址而忽略 P2 口的高 8 位地址,因

此, XDATA 必须从一个段头处开始 (默认为 0x010000~0x0100FF)。

例 7.16:

```
MAP 0x010000,0x01FFFF read write      /* 定义 64K XDATA 存储器空间 */
MAP 0xFF0000,0xFFFFFFF exec read       /* 定义默认代码段空间 */
RESET MAP                               /* 复位存储器空间分配 */
MAP 0x018000,0x1FFFF read write VNM    /* 定义“von Neumann”存储器空间 */
```

7. Unassemble (反汇编) 命令

命令格式: U [地址表达式]

功能说明: U 命令从指定地址开始对程序存储器的内容进行反汇编, 若省略地址表达式选项, 则从上一个 U 命令的终止地址处开始反汇编, 结果显示在反汇编窗口中。显示内容取决于显示方式: 在高级语言方式下将显示源程序语句, 在混合方式下将显示源程序语句及其对应的汇编指令, 在汇编语言方式下只能显示 8051 汇编指令。

例 7.17:

```
>U main          /* 从地址“main”处开始反汇编 */
>U              /* 从上一个 U 命令的终止地址处开始反汇编 */
>U C:0x0         /* 从地址 C:0000H 处开始反汇编 */
```

8. Watchset (观察点设置) 命令

命令格式: WS [观察窗口页号,] 表达式 [,基数]

功能说明: WS 命令用来定义观察点, 并在观察窗口中显示这些观察点的值。观察点表达式中可包含操作数和运算符, 结构、数组和结构指针等类型的数据也可出现在观察点表达式中。“基数”用于指定显示数据格式 (10 进制或 16 进制)。“观察窗口页号”用于指定观察窗口的第几号标签页。每执行一次单步运行 (STEP) 或全速运行 (GO) 命令之后, 观察窗口内的显示值将被更新。

例 7.18:

```
>WS 2,interval,0x0a
>WS save_record[0].analog
>WS save_record[0]
>WS sindex
```

9. WatchKill (观察点删除) 命令

命令格式: WK 观察窗口页号

功能说明: WK 命令删除指定观察窗口内的所有观察点表达式。

例 7.19:

```
>WK 1          /* 删除观察窗口 1 号标签内的所有观察点表达式 */
```

7.3.2 程序执行控制命令

1. COVERAGE 命令

命令格式: COVERAGE

COVERAGE\模块名\

COVERAGE\模块名\函数名

功能说明: 不带参数的 COVERAGE 命令显示已装入到 μ Vision2 调试器中的用户程序全部代码运行覆盖情况, 若在命令中指定了“模块名\函数名”, 则显示相应模块\函数的运行覆盖情况。也可以与 LOG 命令搭配使用, 将运行覆盖情况复制到记录文件。还可以通过单击“View 菜单/ Code Coverage Window”选项打开代码覆盖窗口观察代码运行覆盖的情况。

例 7.20: 下面是对 Cx51 编译器软件包中 MEASURE 范例使用“COVERAGE”命令后的显示结果。

```
\\Measure\MEASURE\{CvtB} - 32% of 52 instructions executed
\\Measure\MEASURE\{CvtB} - 0% of 127 instructions executed
\\Measure\MEASURE\{CvtB} - 100% of 1 instructions executed
\\Measure\MEASURE\{CvtB} - 100% of 6 instructions executed
\\Measure\MEASURE\{CvtB} - 0% of 595 instructions executed
\\Measure\MEASURE\{CvtB} - 0% of 5 instructions executed
\\Measure\MEASURE\{CvtB} - 100% of 1 instructions executed
\\Measure\MEASURE\SAVE_CURRENT_MEASUREMENTS - 0% of 46 instructions
executed
\\Measure\MEASURE\TIMER0 - 72% of 94 instructions executed
\\Measure\MEASURE\READ_INDEX - 0% of 64 instructions executed
\\Measure\MEASURE\CLEAR_RECORDS - 100% of 24 instructions executed
\\Measure\MEASURE\MAIN - 31% of 215 instructions executed
\\Measure\MCOMMAND\{CvtB} - 0% of 74 instructions executed
\\Measure\MCOMMAND\MEASURE_DISPLAY - 100% of 44 instructions executed
\\Measure\MCOMMAND\SET_TIME - 0% of 58 instructions executed
\\Measure\MCOMMAND\SET_INTERVAL - 0% of 101 instructions executed
\\Measure\GETLINE\GETLINE - 78% of 51 instructions executed
\\Measure\TOUPPER\{CvtB} - 100% of 18 instructions executed
\\Measure\TOUPPER\{CvtB} - 51% of 91 instructions executed
\\Measure\TOUPPER\{CvtB} - 0% of 0 instructions executed
```

2. GO (全速运行) 命令

命令格式: G [起始地址][, 结束地址]

功能说明: G 命令从指定的起始地址处启动用户程序全速运行, 若不指定起始地址则从程序计数器所指示的当前地址处开始执行, 一直到结束地址停止, 若省略结束地址, 则程序执行到断点处停止, 也可以单击“Debug 菜单/Stop Running”选项或者将鼠标指向命令窗口然后按下“Esc”键来停止用户程序。结束地址被定义为一个临时断点, 当程序运

行到该断点时,它将被自动删除,如果定义了条件断点则 GO 命令在内部按单步方式执行,这是因为断点条件需要在每执行一次单步时进行一次判断。

例 7.21:

```
>G                      /* 启动程序从当前地址开始全速运行 */
>G,main                 /* 启动程序从当前地址开始运行,直到 main 地址处 */
```

3. Performance Analyzer (性能分析) 命令

命令格式: PA

PA 起始地址 [,终止地址]

PA KILL *

PA KILL 序号 [,序号 [...]]

PA RESET

功能说明: PA 命令用于设置性能分析程序段(简称 PA 范围)。它是位于一段指定地址范围内的用户程序,该段程序在运行过程中被调用的次数以及执行时间都可以显示在性能分析窗口中。不带任何参数的 PA 命令用于显示当前所有已经设置了性能分析的程序段;带有“起始地址”、“终止地址”参数的 PA 命令按指定地址范围设置性能分析程序段,若采用函数名或行号作为设置性能分析程序段“起始地址”时可忽略“终止地址”,此时 μ Vision2 将自动根据函数长度决定其“终止地址”。设置多个性能分析程序段时其地址范围不能相互覆盖,否则将导致错误。 μ Vision2 自动为每个已设置的性能分析程序段加上一个内部序号。带序号的 PA KILL 命令删除指定性能分析程序段;带参数“*”的 PA KILL 命令删除所有已设置的性能分析程序段;PA RESET 命令复位性能分析窗口并清零所有记录。需要注意的是,为了实现正确显示性能分析结果,用户程序应包含完整的调试信息(Cx51 编译时采用 DEBUG、OBJECTEXTEND 命令, A51 汇编时采用 DEBUG、LINES 命令)。

例 7.22:

```
>PA main                /* 定义函数 main()为性能分析程序段 */
>PA timer0              /* 定义函数 timer0()为性能分析程序段 */
>PA clear_records       /* 定义其他性能分析程序段 */
>PA measure_display
>PA save_current_measurements
>PA read_index
>PA set_time
>PA set_interval
>
>PA                    /* 显示所有已定义的性能分析程序段 */
0: main: (FF01EF-FF03B6)
1: timer0: (FF006A-FF0135)
2: clear_records: (FF01C0-FF01EE)
3: measure_display: (FF07E7-FF084A)
4: save_current_measurements: (FF000E-FF0069)
5: read_index: (FF0136-FF01BF)
```

```

6: set_time: (FF084B-FF08CA)
7: set_interval: (FF08CB-FF09A5)
/* 下面的命令在用户程序运行之后执行 */
>PA                      /* 显示所有性能分析程序段及其记录信息 */
0: main: (FF01EF-FF03B6)
   count=1, min=-1, max=0, total=167589
1: timer0: (FF006A-FF0135)
   count=2828, min=33, max=254, total=226651
2: clear_records: (FF01C0-FF01EE)
   count=1, min=27086, max=27086, total=27086
3: measure_display: (FF07E7-FF084A)
   count=10, min=19495, max=19503, total=185027
4: save_current_measurements: (FF000E-FF0069)
   count=491, min=205, max=209, total=100665
5: read_index: (FF0136-FF01BF)
6: set_time: (FF084B-FF08CA)
7: set_interval: (FF08CB-FF09A5)
>
>PA KILL 7                /* 删除几个性能分析程序段 */
>PA KILL 6
>PA KILL 5
>PA                      /* 显示当前性能分析程序段及其记录信息 */
0: main: (FF01EF-FF03B6)
   count=1, min=-1, max=0, total=167589
1: timer0: (FF006A-FF0135)
   count=2828, min=33, max=254, total=226651
2: clear_records: (FF01C0-FF01EE)
   count=1, min=27086, max=27086, total=27086
3: measure_display: (FF07E7-FF084A)
   count=10, min=19495, max=19503, total=185027
4: save_current_measurements: (FF000E-FF0069)
   count=491, min=205, max=209, total=100665
>
>PA RESET                /* 复位性能分析窗口, 清除记录信息 */
>PA                      /* 显示性能分析程序段及其记录信息 */
0: main: (FF01EF-FF03B6)
1: timer0: (FF006A-FF0135)
2: clear_records: (FF01C0-FF01EE)
3: measure_display: (FF07E7-FF084A)

```

4. PSETP (过程单步运行) 命令

命令格式: P[表达式]

功能说明: P 命令的执行方式根据调试窗口中采用的显示方式有所不同。在高级语言

方式下, 执行一条或多条 C51 程序语句, 遇到函数调用语句时不跟踪进入被调用的函数, 而是将其与调用语句一起作为一条程序语句一次执行, 然后进入程序的下一条语句。在汇编语言或混合方式下, 执行一条或多条汇编指令, 对于 LACLL 或 ACALL 指令将连同被调用的子程序一起作为一条 8051 指令一次执行, 然后进入下一条指令。

例 7.23:

```
>P          /* 执行 1 条程序语句, 忽略调用过程 */
>P 100      /* 执行 100 条程序语句, 忽略调用过程 */
```

5. OSTEP (执行并跳出当前函数) 命令

命令格式: O

功能说明: O 命令只有在软件模拟调试方式 (Simulate) 下才有意义, 并且只能够在函数内部执行, 在硬件目标板调试方式下或在函数外部该指令无效。在函数内部从当前程序计数器指向处开始执行, 直到本函数结束并返回到函数调用点下一条语句。

例 7.24:

```
>O          /* 执行并跳出当前函数 */
```

6. TStep (单步运行) 命令

命令格式: T [表达式]

功能说明: T 命令的执行方式根据调试窗口中采用的显示方式有所不同, 在高级语言方式下, T 命令执行一条或按表达式的值执行若干条程序语句, 遇到函数调用语句则跟踪进入被调用的函数继续执行。在汇编语言或混合方式下, T 命令执行一条或按表达式的值执行若干条 8051 指令, 遇到 LCALL 或 ACALL 指令时跟踪进入被调用的子程序继续执行。

例 7.25:

```
>T          /* 执行一条程序语句, 跟踪调用过程 */
>T 100      /* 执行 100 条程序语句, 跟踪调用过程 */
```

7.3.3 断点管理命令

1. BreakSet (断点设置) 命令

命令格式: BS 表达式 [, 计数值 [, “命令串”]]

BS READ 表达式 [, 计数值 [, “命令串”]]

BS WRITE 表达式 [, 计数值 [, “命令串”]]

BS READWRITE 表达式 [, 计数值 [, “命令串”]]

功能说明: μVision2 调试器允许在用户程序中设置三类不同的断点: 执行断点、存取断点和条件断点。当命令中的“表达式”为代码地址时, 该断点即为执行断点。若表达式前面带有存取类型“READ”、“WRITE”、“READWRITE”则为存取断点, 存取断点表达式仅允许使用&, &&, <, <=, >, >=, ==, and !=操作符。若“表达式”不能被简化为代码地址则为条件断点。对于执行断点, 每当程序运行到规定的代码地址时暂停程序运行或执行一条由“命令串”给出的命令, 该地址值必须是一个有效的 8051 指令或程序语句的首地址。

对于存取断点，当程序运行到由“表达式”规定的地址（只能是数据地址）进行指定类型的存取操作时，暂停程序运行或执行一条由“命令串”给出的命令。对于条件断点，其逻辑表达式在每一条汇编指令执行结束后被重新计算，若计算结果为假（0 值），继续执行程序，若计算结果为真（非 0 值），暂停程序运行或执行一条由“命令串”给出的命令。条件断点最为灵活，因为可以通过判断逻辑表达式的值来决定是否需要暂停程序的执行，另一方面设置条件断点会降低程序的执行速度，因为需要反复计算逻辑表达式的值。“计数值”用来规定程序执行暂停时该断点出现的次数，即程序只有在遇到对该断点规定的次数后才暂停运行，默认的计数值为 1。“命令串”为有效的 μ Vision2 命令序列，用户程序运行到断点处时并不暂停，而是在执行完该命令序列后继续运行，若需要在执行该命令序列后暂停程序运行，可通过在命令序列中增加一条系统变量赋值命令：“_BREAK_ = 1”，利用“命令串”还可以调用 μ Vision2 的用户函数或信号函数。定义一个断点后， μ Vision2 自动为该断点生成一个内部序号，以简化其他断点管理命令的操作。

设 Cx51 程序中包含下列数据：

```
struct {
    char hour;
    char min;
    char sec;
}time;
int i0,i1;
```

下面的存取断点命令设置正确：

```
BS WRITE time.sec                /* 正确，可以产生惟一的存储器地址 */
BS WRITE time.sec == i0
BS WRITE time.sec != i0*i
BS WRITE time.sec && (ACC==5 && i1 != i0)
```

下面的存取断点命令设置错误：

```
BS WRITE time.sec + i0           /* 错误，不能产生惟一的存储器地址 */
```

例 7.26:

```
>BS main                        /* 在地址 main 处设置一个执行断点 */
>BS timer0,10, "myregs()"      /* 在地址 timer0 处设置一个执行断点，
                                程序在第十次运行到地址 timer0 时，
                                利用“命令串”调用  $\mu$ Vision2 的用户
                                函数“myregs()”并继续运行程序 */
>BS sindex==8                  /* 设置一个条件断点，程序在每执行完一
                                条汇编指令后，都将检查表达式 sindex
                                ==8 的值，若为真(非 0 值)则暂停运行
                                用户程序，否则继续运行用户程序。*/

>BS save_record[5].time.sec>5, 3 /* 与上例相似，不同之处为当第三次检
                                查出表达式的值为真时才暂停用户程
```

```

                                序的运行。 */
>BS READ interval.min==3      /* 设置一个存取断点，每当程序对变量
                                interval.min 进行读取并识别出其值
                                为 3 时才暂停用户程序的运行。 */
>BS WRITE savefirst==5&&acc==0x12 /* 设置一个存取断点，当程序对符号地
                                址 savefirst 和 acc 进行写入，并识别
                                出表达式的值为真时才暂停用户程序
                                的运行。 */

```

2. BreakList (断点列表) 命令

命令格式: BL

功能说明: BL 命令在命令窗口中列表输出已定义的所有断点，每个输出行中包括断点序号、表示断点类型的标识字符 (E 为执行断点，A 为存取断点，C 为条件断点)、可能的符号地址值、存取断点的存取类型 (RD、WR、RW)、设置断点命令中的表达式、计数值 (CNT)、断点的当前状态 (活动态为 Enabled，禁止态为 Disabled) 以及设置断点命令中的命令串等内容。

例 7.27:

```

>BL
0:(E C:0x231) 'main',CNT=1,enabled
1:(E C:0x74) 'timer0',CNT=10,enabled exec("myregs()")
2:(C) 'sindex==8',CNT=1,enabled
3:(C) 'save_record[5].time.sec>5',CNT=3,enabled
4:(A RD D:0x37) 'read interval.min==3',CNT=1,enabled
5:(A WR D:0x34) 'writefirst==5&&acc==0x12',CNT=1,enabled

```

3. BreakKill (断点删除) 命令

命令格式: BK 序号[,序号[,...]]

BK *

功能说明: 带“序号”的 BK 命令删除与序号相应的断点，带“*”号的 BK 命令删除所有断点。

例 7.28:

```

>BK 0,1          /* 删除断点 0 和断点 1 */
>BK *           /* 删除所有断点 */

```

4. BreakEnable (断点激活) 命令

命令格式: BE 序号[,序号[,...]]

BE *

功能说明: 带“序号”的 BE 命令激活与序号相应的禁止态断点，带“*”号的 BE 命令激活所有禁止态断点，被激活的断点恢复暂停用户程序运行的功能。

例 7.29:

```
>BE 0,1      /* 激活断点 0 和断点 1 */
>BE *        /* 激活所有断点 */
```

5. BreakDisable (断点禁止) 命令

命令格式: BD 序号[序号[,...]]

BD *

功能说明: 带“序号”的 BD 命令禁止与序号相应的活动态断点, 带“*”号的 BD 命令禁止所有活动态断点。处于禁止态的断点不能暂停用户程序的运行, 即该断点暂时失去作用, 但在断点列表时仍有输出。

例 7.30:

```
>BD 0,1      /* 禁止断点 0 和断点 2 */
>BD *        /* 禁止所有断点 */
```

以上这些断点管理命令的功能还可以通过单击“Debug 菜单/Breakpoints...选项”, 从打开的“断点设置窗口”中来实现。

7.3.4 其他通用命令**1. ASSIGN (分配) 命令**

命令格式: ASSIGN

ASSIGN WIN|COM1|COM2 |COM3|COM4 [<inreg] [>outreg]

功能说明: 不带参数的 ASSIGN 命令用于显示串行口当前的分配情况, 带参数的 ASSIGN 命令用来改变串行口当前的分配情况。目前, μ Vision2 支持以下串行口: WIN、COM1、COM2、COM3 和 COM4。其中, “WIN”为 μ Vision2 的串行窗口 (Serial window), 用于完成 C51 程序的输入输出操作, “COMx”为 PC 机的串行口, 可以用来与单片机目标系统通信。选项“<inreg”和“>outreg”分别表示单片机的串口名 (如 80517 单片机有两个串行口 s0in、s0out、s1in 和 s1out, 而 8051 单片机只有 sin 和 sout), 它们由创建用户项目时从器件库中选定的单片机 CPU 所定义。

例 7.31:

```
>ASSIGN      /* 询问串口的分配情况 */
WIN:  <S0IN >S0OUT    /* S0IN 提供串行输入, S0OUT 提供串行输出 */
>ASSIGN WIN <S1IN >S1OUT /* 分配 S1IN、S1OUT 给  $\mu$ Vision2 的串行窗口 */
>ASSIGN COM1 <SIN >SOUT /* 分配 SIN、SOUT 给 PC 机的 COM1 */
```

2. DEFINE BUTTON (定义按钮) 命令

命令格式: DEFINE BUTTON “按钮名”, “命令”

功能说明: DEFINE BUTTON 命令用于创建工具箱窗口中的按钮, 每个按钮可执行一个确定的 μ Vision2 命令。“按钮名”确定了工具箱窗口中按钮的显示名称, “命令”必须是一个有效的 μ Vision2 命令, 每当按下该按钮时该命令立即被执行。 μ Vision2 自动为每一个

新创建的按钮加上一个序号，以便于删除工具箱窗中的某个按钮。

例 7.32:

```
>DEFINE BUTTON "clr dptr", "dptr=0"
>DEFINE BUTTON "show main()", "u main"
>DEFINE BUTTON "show r7", "printf (\"R7=%02XH\n\", R7)"
```

3. DIR (列目录) 命令

命令格式: DIR

```
DIR LINE
DIR PUBLIC
DIR VTREG
DIR DEFSYM
DIR \module
DIR \module LINE
DIR \module\func
DIR \module\func LINE
DIR FUNC
DIR UFUNC
DIR BFUNC
DIR SIGNAL
```

功能说明: 该命令用来输出各种模块、函数、变量的符号名, 若命令中不给出参数, 则输出当前程序中的所有符号名。命令中各参数的意义如下:

PUBLIC	输出所有全局符号名、存储器类型及地址。
VTREG	输出 CPU 动态驱动库文件 (DLL) 中所规定的虚拟寄存器的内容。
DEFSYM	输出由“DEFINE”命令创建的符号名。
\module	输出指定模块或当前模块的所有局部符号及地址。
\module\func	输出指定模块 module 中函数 func 的局部符号及地址。
\module LINE	输出指定模块或当前模块的行号及代码起始地址。
FUNC	输出所有已定义的 μ Vision2 函数名。
UFUNC	输出所有用户定义的 μ Vision2 函数名。
BFUNC	输出予定义的 μ Vision2 内部函数名。
SIGNAL	输出所有 μ Vision2 的信号函数名。

例7.33: 下面是对Cx51编译器软件包中MEASURE范例使用“DIR”命令后的显示结果。

```
>DIR MODULE /* 显示所有模块名称 */
MEASURE
MCOMMAND
GETLINE
?C_FPADD
?C_FPMUL
```

```

...
>DIR \MEASURE      /* 显示模块'MEASURE'中各函数名 */
MODULE: MEASURE
  C:0x000000 . . . . _ICE_DUMMY_ . . uint
  FUNCTION: {CvtB} RANGE: 0xFF03B7-0xFF07E5
  C:0x000000 . . . . _ICE_DUMMY_ . . uint
  FUNCTION: {CvtB} RANGE: 0xFF000B-0xFF000D
  C:0x000000 . . . . _ICE_DUMMY_ . . uint
  FUNCTION: SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069
  FUNCTION: TIMER0 RANGE: 0xFF006A-0xFF0135
  D:0x00000F . . . . i . . uchar
  FUNCTION: _READ_INDEX RANGE: 0xFF0136-0xFF01BF
  D:0x00003F . . . . buffer . . ptr to char
  D:0x000042 . . . . index . . int
  D:0x000007 . . . . args . . uchar
  FUNCTION: CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE
  D:0x000006 . . . . idx . . uint
  FUNCTION: MAIN RANGE: 0xFF01EF-0xFF03B6
  I:0x000067 . . . . cmdbuf . . array[15] of char
  D:0x00003C . . . . i . . uchar
  D:0x00003D . . . . idx . . uint
>DIR \MEASURE LINE      /*显示MESURE 模块的行号信息 */
MODULE: MEASURE
C:0x000E . . . . #87
C:0x000E . . . . #88
C:0x003A . . . . #89
C:0x0049 . . . . #90
...
C:0x03B6 . . . . #291
C:0x03B6 . . . . #292
DIR PUBLIC
>DIR PUBLIC      /* 显示所有全局变量 */
  B:0x000640 . . . . T2I0 . . bit
  B:0x000641 . . . . T2I1 . . bit
  ...
  D:0x000023 . . . . current . . struct mrec
  C:0x0007CD . . . . ERROR . . array[16] of char
  X:0x004000 . . . . save_record . . array[744] of struct mrec
  C:0x00000E . . . . save_current_measurements . . void-function
  C:0x0001EF . . . . main . . void-function
  C:0x00047E . . . . menu . . array[847] of char
  D:0x000030 . . . . setinterval . . struct interval
  ...

```

```

B:0x000601 . . . . IEX2 . . bit
B:0x000600 . . . . IADC . . bit
>DIR VTREG          /*显示引脚寄存器及其值*/
PORT0: uchar, value = 0xFF
PORT1: uchar, value = 0xFF
PORT2: uchar, value = 0xFF
PORT3: uchar, value = 0xFF
PORT4: uchar, value = 0xFF
PORT5: uchar, value = 0xFF
PORT6: uchar, value = 0xFF
PORT7: uchar, value = 0x00
PORT8: uchar, value = 0x00
AIN0: float, value = 0
AIN1: float, value = 0
AIN2: float, value = 0
AIN3: float, value = 0
AIN4: float, value = 0
AIN5: float, value = 0
AIN6: float, value = 0
AIN7: float, value = 0
AIN8: float, value = 0
AIN9: float, value = 0
AIN10: float, value = 0
AIN11: float, value = 0
S0IN: uint, value = 0x0000
S0OUT: uint, value = 0x0000
S1IN: uint, value = 0x0000
S1OUT: uint, value = 0x0000
VAGND: float, value = 0
VAREF: float, value = 5
XTAL: ulong, value = 0xB71B00
PE_SWD: uchar, value = 0x00
STIME: uchar, value = 0x00
>$ = MAIN          /* 设置当前运行点为main() */
>DIR              /* 显示当前模块的变量信息*/
FUNCTION: MAIN RANGE: 0xFF01EF-0xFF03B6
I:0x000067 . . . . cmdbuf . . array[15] of char
D:0x00003C . . . . i . . uchar
D:0x00003D . . . . idx . . uint
>DIR DEFSYM      /* 显示由 'DEFINE <type> <name>' 命令定义的变量 */
word00: int, value = 0x0000
byte00: char, value = 0x00
dword00: long, value = 0x0

```

```

float00: float, value = 0
>DIR FUNC      /* 显示预定义 $\mu$ Vision2 函数*/
predef'd: void MEMSET (ulong, ulong, uchar)
predef'd: void TWATCH (ulong)
predef'd: int RAND (uint)
predef'd: float GETFLOAT (char *)
predef'd: long GETLONG (char *)
predef'd: int GETINT (char *)
predef'd: void EXEC (char *)
predef'd: void PRINTF (char *, ...)

```

4. EXIT (退出) 命令

命令格式: EXIT

功能说明: 该命令退出 μ Vision2 调试状态, 它只能在 μ Vision2 的命令窗口中使用, 不能作为“exec”函数的参数。

5. INCLUDE (包含) 命令

命令格式: INCLUDE [路径]文件名

功能说明: 该命令用来加载一个初始化文件 (即 ini 文件) 并逐行执行其中的命令, 文件中可以嵌套 4 层 INCLUDE 命令。执行 INCLUDE 命令之前应首先停止运行用户程序, 否则将导致错误而使 INCLUDE 命令被取消。

例 7.34:

```
INCLUDE measure.ini      /* 加载并执行初始化文件 “measure.ini” */
```

6. KILL (删除) 命令

命令格式: KILL FUNC 函数名

KILL FUNC *

KILL BUTTON 序号

功能说明: 带有“函数名”参数的 KILL FUNC 命令, 删除指定的 μ Vision2 用户函数或信号函数; 带有符号“*”的 KILL FUNC 命令删除所有 μ Vision2 用户函数或信号函数; 不能删除 μ Vision2 的预定义函数; 带有序号的 KILL BUTTON 命令删除工具箱窗口中指定序号的用户定义按钮。

例 7.35:

```

>KILL FUNC analog0 /* 删除用户函数 analog0 */
>KILL FUNC *       /* 删除所有用户函数 */
>KILL BUTTON 3     /* 删除工具箱窗口中按钮 3 */

```

7. LOAD (加载) 命令

命令格式: LOAD [路径]文件名 NOCODE

功能说明: LOAD 命令可用来加载下列类型的文件

(1) Intel HEX 文件

这种格式的文件没有调试信息，不能进行符号化调试，因此用户程序只能在汇编语言方式下调试。

(2) 绝对目标文件文件

这种格式的文件是由 A51、C51 或 PL/M51 对源程序文件编译之后再由 BL51 处理而产生的，文件中可以包含完整的符号、类型和行号等调试信息。高级语言程序的符号化调试只能采用这种格式的文件，为此，源程序在编译时必须采用 DEBUG 和 OBJECTEXTEND 编译控制指令。

在硬件目标板调试方式下，可以使用“NOCODE”参数，此时 μ Vision2 仅载入符号信息而忽略代码记录。 μ Vision2 自动分析加载文件的内容并确定文件类型，如果不能确定文件类型，就不能载入，同时显示错误信息。

例 7.36:

```
>LOAD C:\DSW\SAMPL51\MEASURE\MEASURE /* 加载指定路径下的用户程序文件 */
>LOAD MYPROG.HEX /* 加载 myprog.hex 文件 */
```

8. LOG (记录) 命令

命令格式: LOG>[路径]文件名
LOG>>[路径]文件名
LOG
LOG OFF

功能说明: 带参数的 LOG 命令用来产生或打开一个记录文件，同时启动后台记录过程，自动将命令窗口的显示输出记录到该文件中。若文件名前面带有“>”符号，表示产生一个新的记录文件并覆盖同名的旧文件，若文件名前面带有“>>”符号，表示将记录追加到该文件中。不带参数的 LOG 命令用于显示当前记录文件的状态。LOG OFF 命令关闭已打开的记录文件并结束后台记录过程。

例 7.37:

```
>LOG>C:\TMP\dslog /* 产生一个记录文件 */
>LOG /* 显示记录文件状态 */
C:\TMP\dslog
>LOG OFF /* 关闭记录文件 */
```

9. MODE (串口设置) 命令

命令格式: MODE COMx, 波特率, 校验位, 数据位, 停止位

功能说明: MODE 命令用于改变 PC 机的串口设置。参数“COMx”中的“x”可为 1、2、3、4，表示 PC 机的串行口。参数“波特率”必须是一个有效的波特率值，如 1200，2400，9600 或 19200。参数“校验位”为 0 时表示无检验位，1 为奇校验，2 为偶校验。参数“数据位”可为 7 或 8。“停止位”可为 1，1.5 或 2。MODE 命令可以与 ASSIGN 命令联合使用，以设置仿真 CPU 的串口输入输出渠道。

例 7.38:

```
>MODE COM2, 19200, 0, 8, 1 /* 设置 COM2 的波特率为 19200 bps, */
/* 无校验位, 8 位数据, 1 位停止位 */
```

10. RESET (复位) 命令

命令格式: RESET

RESET MAP

RESET SRC

功能说明: 不带参数的 RESET 命令用来复位被仿真的单片机恢复到初始状态, 但保留已装入的用户程序; 带“MAP”参数的 RESET 命令对所有已定义的存储器映像进行复位, 清零已加载的用户程序存储器以及所有调试信息; 带“SRC”参数的 RESET 命令复位 μ Vision2 系统变量“SRC”的值。

例 7.39:

```
>RESET /* 复位 CPU */
>RESET MAP /* 复位已分配的存储器映像 */
```

11. SAVE (存盘) 命令

命令格式: SAVE [路径]文件名 地址 1, 地址 2

功能说明: SAVE 命令将一段内存映像以 HEX386 格式存盘, “地址 1, 地址 2”用于指定要保存的内存映像地址空间范围, 所保存的文件可以采用 LOAD 命令装入 μ Vision2 调试器, 但没有调试信息。不能将 8051 单片机片内 RAM I:0x80~I:0xFF 的内容存盘。

12. SCOPE (显示地址范围) 命令

命令格式: SCOPE[模块[函数]]

功能说明: 该命令显示用户程序中指定模块和函数的地址范围。如果加载的程序文件带有调试信息, μ Vision2 将产生一个内部地址分配表, 其中包括模块、函数以及符号的地址信息。若命令中没有规定模块及其函数名, 则显示所有模块的地址分配, 若规定了模块或函数名则只输出该模块或函数的地址分配。

例 7.40:

```
>scope \measure\main /* 显示函数 main() 的地址范围 */
MAIN RANGE: 0xFF01EF-0xFF03B6
>
>scope \measure /* 显示模块 MEASURE 的地址范围 */
MEASURE
{CvtB} RANGE: 0xFF03B7-0xFF07E5
{CvtB} RANGE: 0xFF000B-0xFF000D
SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069
TIMER0 RANGE: 0xFF006A-0xFF0135
_READ_INDEX RANGE: 0xFF0136-0xFF01BF
CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE
MAIN RANGE: 0xFF01EF-0xFF03B6
```

```

>
>scope                                /* 输出所有模块的地址范围 */
MEASURE
  {CvtB} RANGE: 0xFF03B7-0xFF07E5
  {CvtB} RANGE: 0xFF000B-0xFF000D
  SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069
  TIMER0 RANGE: 0xFF006A-0xFF0135
  _READ_INDEX RANGE: 0xFF0136-0xFF01BF
  CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE
  MAIN RANGE: 0xFF01EF-0xFF03B6
MCOMMAND
  {CvtB} RANGE: 0xFF09A6-0xFF0A23
  MEASURE_DISPLAY RANGE: 0xFF07E7-0xFF084A
  _SET_TIME RANGE: 0xFF084B-0xFF08CA
  _SET_INTERVAL RANGE: 0xFF08CB-0xFF09A5
GETLINE
  _GETLINE RANGE: 0xFF0A24-0xFF0A87
?C_FPADD
?C_FPMUL
?C_FPDIV
?C_FPCMP
...

```

在以上个命令的输出显示行中，带有符号“{CvtB}”者表示该段地址范围内的程序模块没有足够的调试信息，其原因可能是该段程序为库文件，或是编译时未采用 DEBUG 编译指令。类似 0xFF0A24-0xFF0A87 等输出项表示模块或函数占用的地址空间。

13. SET (设置) 命令

命令格式: SET SRC[=路径名]

功能说明: SET SRC 命令显示或定义高级语言源文件或列表文件的搜索路径，加载一个目标文件时 μ Vision2 会自动定义其搜索路径，用户可以增加定义搜索路径，最多可定义 10 个路径名。

例 7.41:

```

>LOAD C:\C51\EXAMPLES\MEASURE      /* 装入程序模块 MEASURE */
>SET SRC                             /* 输出程序模块的路径 */
C:\C51\EXAMPLES\
>SET SRC = D:\Myexample             /* 增加定义搜索路径 */
>SET SRC                             /* 输出路径 */
C:\C51\EXAMPLES\
D:\Myexample

```

14. SIGNAL (信号) 命令

命令格式: SIGNAL KILL 函数名

SIGNAL STATE

功能说明：**SIGNAL KILL** 命令从当前活动的信号函数表中禁止指定的信号函数，禁止的信号函数可被重新激活。**SIGNAL STATE** 命令输出当前信号函数表的状态，其输出行中的第一个数字是信号函数的内部序号，位于其后的项表示信号函数的状态，最后是信号函数名和已执行的语句数。

例 7.42:

```
>INCLUDE ANALOG.INC           /* 加载信号函数 */
>analog0(10000)
ANALOG0(10000.000000) ENTERED
>SIGNAL STATE                 /* 输出当前信号函数表的状态 */
0 idle signal=ANALOG0(line 9)
>SIGNAL KILL analog0          /* 禁止信号函数 analog0 */
```

15. SLOG (串行口记录) 命令

命令格式：**SLOG** > [路径] 文件名
SLOG >> [路径] 文件名
SLOG
SLOG OFF

功能说明：带参数的 **SLOG** 命令用于产生或打开一个记录文件，同时启动后台记录过程，自动将串行窗口的输入和输出数据记录到该文件中。若文件名前面带有“>”符号，表示产生一个新的记录文件并覆盖同名的旧文件，若文件名前面带有“>>”符号，表示将记录追加到该文件中。不带参数的 **SLOG** 命令用于显示当前记录文件状态。**SLOG OFF** 命令关闭已打开的记录文件并结束后台记录过程。

例 7.43:

```
SLOG >C:\TMP\dslog             /* 生成一个新串口记录文件 */
SLOG                           /* 检查串口记录文件状态 */
serial log file: C:\TMP\dslog
SLOG OFF                       /* 关闭串口记录文件 */
```

7.4 μ Vision2 的表达式

上一节介绍了 μ Vision2 的各种命令，大多数 μ Vision2 命令的参数都可以是表达式。一个表达式由若干个数字、调试目标和运算符组成。 μ Vision2 除了能识别和处理作为命令参数的表达式之外，还能识别和处理从命令窗口直接输入的表达式（非命令参数），计算和输出非命令参数表达式的值。本节详细介绍有关 μ Vision2 表达式的语法规则。

7.4.1 表达式的组成

一个表达式通常由常数、系统变量、符号、片上集成外围功能符号、程序符号、行号、位地址、存储器类型以及运算符等组成。

1. 常数

μVision2 表达式中的数字常数可以带有前缀或后缀以表示其制式,如 1234H 或 0x1234 (16 进制数)、1234T (10 进制数)、777Q (8 进制数)、10100101Y (二进制数)等。当某个常数中第一个有效字符为 16 进制数的 A~F 或 a~f 时,必须在其前面加一个前导数字字符“0”。对于整数可带有一个后缀字符“L”以便将数据或计算结果的表示范围扩大到 32 位整数值(长整数),如 0x1234L、1234TL 等,如果一个数字已超出了整数的表示范围(0~65535),μVision2 会自动将其转换成长整数。对于表达式中的浮点数与 C51 中浮点数的格式相似,但在小数点前面必须以数字开头。例如 0.12 不能写成.12。对于表达式中的字符和字符串常数的语法规则与 C51 一致,支持转义字符,但不能将几个单独字符串连成一个字符串。

2. 系统变量

μVision2 具有如下预定义系统变量,它们可以直接在表达式中使用,其符号的输入是大小写无关的。

(1) \$

\$是无符号长整型变量(unsigned long),用于显示或调整程序计数器(PC)的当前值,并在 Register 窗口中显示。

例 7.44:

```
$=0x1000          /* 将程序计数器 PC 的值调整为 0x1000 */
```

(2) STATES

STATES 是无符号长整型变量(unsigned long),且具有“只读”属性,表示用户程序已经经历的机器周期数,其值随程序指令的执行而增加,并在 Register 窗口中显示。

(3) RADIX

RADIX 是无符号整型变量(unsigned int),它决定数值输出时的制式,默认值为 0x10 (即 16 进制数输出)。

例 7.45:

```
>RADIX=0x0a       /* 10 进制数输出 */
```

```
>RADIX=0x10       /* 16 进制数输出 */
```

(4) _IIP_

IIP (Interrupt In Progress) 是无符号字符型变量(unsigned char),表示当前中断的嵌套状态。

(5) _BREAK_

_BREAK_是无符号整型变量(unsigned int),将变量_BREAK_置为 1 时,可中止用户程序的运行,该变量通常用在断点设置命令(BS)的“命令串”参数中。例如,在 BS 命令的“命令串”中加入表达式“_BREAK_=1”,则用户程序运行到断点处时μVision2 执行完指定“命令串”后暂停程序运行。

(6) ITRACE

ITRACE 是无符号整型变量 (unsigned int), 表示跟踪缓冲区的当前状态, 将其值置为 0 时表示关闭跟踪缓冲区, 将其值置为非 0 时表示打开跟踪缓冲区。

例 7.46:

```
>ITRACE=1          /* 打开跟踪缓冲区 */
>ITRACE=0          /* 关闭跟踪缓冲区 */
>ITRACE            /* 显示跟踪缓冲区当前状态 */
```

3. 片上集成外围功能符号

根据从器件库中选定的单片机 CPU, μ Vision2 将自动定义两类符号: 特殊功能寄存器 (SFR) 和虚拟 CPU 引脚寄存器 (VTREGs)。

(1) 特殊功能寄存器 (SFR)

μ Vision2 支持所选定单片机 CPU 中的全部特殊功能寄存器 (SFR), 每个 SFR 都有相应的地址, 并可用于表达式之中。

(2) 虚拟 CPU 引脚寄存器 (VTREGs)

目前 8051 系列单片机已有 400 多种不同的型号, 各种型号单片机具有不同的集成外围功能。为了使 μ Vision2 能够正确仿真各种型号的 8051 单片机, 必须有相应的 CPU 动态驱动库文件 (DLL) 支持, DLL 文件通过定义一系列虚拟 CPU 引脚寄存器 (VTREGs) 来实现对单片机外围集成功能的仿真, VTREGs 中包括以下符号。

AINx: 片上模拟输入引脚 (analog input pin), 向其写入的数值可由用户程序读取。

PORTx: 8051 单片机的并行 I/O 端口, 用于输入/输出端口的仿真。

SxIN: 串行口接收缓冲区, 可以向 SxIN 写入一个 8 位或 9 位的数据, 并被用户程序读取。也可以通过读 SxIN 来决定串行口接收缓冲区是否就绪, SxIN 的值为 0xFFFF 时上次输入的数据已经处理完毕, 可以写入新的数据。

SxOUT: 串行口发送缓冲区, μ Vision2 根据用户程序向缓冲区复制一个 8 位或 9 位的数据。

XTAL: 晶体振荡器频率, 可以通过单击“Project 菜单/Options For Target/Target”选项来设定。

CLOCK: CPU 内部时钟频率, 它是振荡器频率 XTAL 的 12 分频值, 用于计算指令的执行时间。

SxTIME: 串行口的波特率发生器定时控制, 当 SxTIME 为 1 时, μ Vision2 根据用户程序中设定的波特率仿真串行口的定时, 当 SxTIME 为 0 时, 将忽略用户程序中对波特率定时的设定。

例 7.47: Infineon80C517 单片机定义了如下虚拟 CPU 引脚寄存器 (VTREGs)。

- PORT0: I/O 端口 0, 8 位输入/输出线
- PORT1: I/O 端口 1, 8 位输入/输出线
- PORT2: I/O 端口 2, 8 位输入/输出线
- PORT3: I/O 端口 3, 8 位输入/输出线

- PORT4: I/O 端口 4, 8 位输入/输出线
- PORT5: I/O 端口 5, 8 位输入/输出线
- PORT6: I/O 端口 6, 8 位输入/输出线
- PORT7: I/O 端口 7, 8 位输入/输出线
- PORT8: I/O 端口 8, 8 位输入/输出线
- AIN0: 模拟量输入线 AIN0 (浮点值)
- AIN1: 模拟量输入线 AIN1 (浮点值)
- AIN2: 模拟量输入线 AIN2 (浮点值)
- AIN3: 模拟量输入线 AIN3 (浮点值)
- AIN4: 模拟量输入线 AIN4 (浮点值)
- AIN5: 模拟量输入线 AIN5 (浮点值)
- AIN6: 模拟量输入线 AIN6 (浮点值)
- AIN7: 模拟量输入线 AIN7 (浮点值)
- AIN8: 模拟量输入线 AIN8 (浮点值)
- AIN9: 模拟量输入线 AIN9 (浮点值)
- AIN10: 模拟量输入线 AIN10 (浮点值)
- AIN11: 模拟量输入线 AIN11 (浮点值)
- VAGND: 模拟参考电压 VAGND (浮点值)
- VAREF: 模拟参考电压 VAREF (浮点值)
- S0IN: 串行口 0 接收缓冲区 (9 位)
- S0OUT: 串行口 0 发送缓冲区 (9 位)
- S1IN: 串行口 1 接收缓冲区 (9 位)
- S1OUT: 串行口 1 发送缓冲区 (9 位)
- XTAL: 振荡器频率
- STIME: 串行口定时允许
- CLOCK: CPU 内部时钟频率

特殊功能寄存器 (SFR) 符号与虚拟 CPU 引脚寄存器 (VTREGs) 符号是不相同的。SFR 符号可以通过 CPU 存储器空间进行存取操作, 而 VTREGs 符号所表示的是出现在单片机引脚上的信号。另外两者的表示形式也不相同, 例如, 对于 I/O 口 2, VTREGs 符号为 PORT2, 而 SFR 符号为 P2, 使用中不要混淆。利用 CPU 引脚寄存器 (VTREGs) 符号很容易实现对并行 I/O 口和串行口的仿真。

4. 程序符号

在 μ Vision2 中允许使用与用户目标程序中相同的变量或符号, 称为程序符号, 从而实现符号化源程序调试, 为此应在创建目标代码时, 通过单击“Project 菜单/Options For Target/Output/”选项, 选中其中的“Debug Information”复选框, 使所创建的目标代码中包含调试信息。程序符号包括限定符号、非限定符号和文字符号。

(1) 限定符号

限定符号包括定义该符号的模块名或函数名, 其格式如下:

\模块名\行号	/* 指定模块中的一个行号 */
\模块名\函数名	/* 指定模块中的一个函数 */
\模块名\符号名	/* 指定模块中的一个全局符号 */
\模块名\函数名\符号名	/* 指定模块的指定函数中的一个全局符号 */

例 7.48:

```

\MEASURE\clear_records\idx /* 指定 MEASURE 模块的 clear_records 函数中局部符
                             号 idx */
\MEASURE\MAIN\cmdbuf      /* 指定 MEASURE 模块的 MAIN 函数中局部符号 cmdbuf */
\MEASURE\sindx            /* 指定 MEASURE 模块中的符号 sindex */
\MEASURE\225              /* 指定 MEASURE 模块中的行号 225 */
\MEASURE\TIMER0           /* 指定 MEASURE 模块中的符号 TIMER0, 该符号可以是
                             函数名, 也可以是全局变量名 */

```

(2) 非限定符号

非限定符号是一个不包括路径的简单变量或函数名。 μ Vision2 在分析一个表达式时, 按以下顺序搜索表达式中的非限定符号:

- 首先搜索 CPU 内部寄存器符号;
- 接着搜索目标程序中当前函数的局部变量;
- 再接着搜索目标程序中当前模块的静态变量 (即模块中函数外部声明的变量);
- 再接着搜索目标程序中的全局或公共符号, 包括特殊功能寄存器 (SFR) 符号;
- 再接着搜索由 μ Vision2 的 “DEFINE” 命令所创建的符号, 该符号不属于目标程序;
- 再接着搜索系统变量, 系统变量不属于目标程序;
- 再接着搜索虚拟 CPU 引脚寄存器 (VTREGs) 符号。

对于由用户函数或信号函数所创建的符号, 先在用户函数或信号函数中搜索, 然后再按以上顺序搜索。

(3) 文字符号

μ Vision2 调试命令及选项、CPU 寄存器名、汇编语言中的指令助记符、各种数据类型名等都属于 “保留字”, 如果在表达式中需要使用与 “保留字” 同名的符号, 应对其进行 “文字化”, 即在该表达式符号前面加一个撇号 “'”。如果某个虚拟 CPU 引脚寄存器 (VTREGs) 符号与表达式中的符号同名, 则在存取该 VTREGs 符号时也要进行 “文字化”, 即在该 VTREGs 符号前面加一个撇号 “'”。

例 7.49: 用户程序中定义了一个与 CPU 寄存器名 “R5” 同名的变量。

```

>R5 = 123          /* 对 CPU 寄存器 R5 赋值 */
>' R5 = 123        /* 对文字符号变量名 R5 赋值 */

```

例 7.50: 用户程序中定义了一个与 VTREGs 符号 “CLOCK” 同名的函数。

```

>CLOCK            /* 存取 CLOCK 函数 */
>' CLOCK          /* 存取 VTREGs 符号 CLOCK */

```


5. 行号

行号是由编译器产生的,它表示源程序中一条语句与该语句的第一条汇编指令代码物理地址之间的联系。行号作为一种调试信息存储在目标文件中,采用高级语言调试方式时行号是不可缺少的。 μ Vision2 允许在表达式中使用行号来表示被仿真单片机的程序存储器地址。其格式如下:

\整数行号

\模块名\整数行号

用第一种格式表示的行号必须位于当前模块中,用第二种格式表示的行号必须是指定模块中的有效行号,否则将产生错误信息。

例 7.51:

```
>\measure\108      /* 模块 measure 中的 108 行 */
>\140               /* 当前模块的 140 行 */
```

6. 位地址

位地址表示 8051 单片机特殊功能寄存器或内部 RAM 可位寻址区中某一位的地址,其格式如下:

表达式.位

其中,“表达式”的取值范围在 0x20~0x2f 之间或 0x80~0xff 之间,“位”必须是一个 0~7 之间的一个整数。

例 7.52:

```
>20H.0              /* 位地址 0 */
>0x2f.7             /* 位地址 127 */
>ACC.0              /* 累加器 ACC 的位 0 */
```

7. 存储器类型

存储器类型用来规定 8051 存储器地址空间,对于一条输出命令中的地址或表达式,必须指出其存储器类型,否则 μ Vision2 无法确定对象的存储器空间。有效的存储器类型如下:

B: 位寻址区 (BIT)

C: 程序存储器 (CODE)

Bx: 分组代码程序存储器 (CODE BANK),x 为有效的代码组号

D: 直接寻址的内部数据存储器 (DATA)

I: 间接寻址的内部数据存储器 (IDATA)

X: 外部数据存储器 (XDATA)

例 7.53:

```
C:0x100             /* 代码存储器地址 0x100 */
I:100               /* 内部数据存储器地址 64H */
X:0FFFFH           /* 外部数据存储器地址 0FFFFH */
B:0x7F              /* 位地址 127 或 2FH.7 */
```

```
B2:0x9000          /* 分组代码程序存储器 CODE BANK2 的地址 9000H */
```

8. 类型转换

采用类型转换可改变一个表达式的数据类型。通常 μ Vision2 可自动完成表达式的类型转换，用户也可在需要时强制进行类型转换，转换规则与 C 语言相同。

例 7.54:

```
>(float) 0x100          /* 将 0x100 强制转换成浮点类型 256 */
>(unsigned int) 3.14     /* 将浮点数 3.14 强制转换成整型数 0x0003 */
```

9. 运算符

在 μ Vision2 的表达式中可使用 ANSI C 以及 C51 扩展的所有运算符。

7.4.2 μ Vision2 表达式与 C 语言表达式之间的差别

μ Vision2 表达式与 C 语言表达式之间具有如下一些不同之处。

- ① μ Vision2 表达式中不区分符号名和命令名的大、小写字母。
- ② μ Vision2 表达式中不允许出现带类型的指针（例如 char *），指针类型只能从被调试程序的调试信息中获得。
- ③ 不支持对结构变量的赋值。
- ④ 不能从命令窗口调用用户程序中的函数。

7.4.3 μ Vision2 表达式应用举例

本节列举了一些在程序调试过程中使用 μ Vision2 各种表达式的例子，以帮助读者进一步熟悉 μ Vision2 调试器的使用方法。本节的例子适用于对范例程序 MEASURE.C 的调试，为此应先启动 μ Vision2 调试器并装入 MEASURE 目标文件。

(1) 常数

```
>0x1234              /* 输出简单常数 */
0x1234
>EVAL 0x1234         /* 按几种不同数制输出常数 */
4660T 11064Q 1234H '...4'
```

(2) CPU 寄存器

```
>ACC                 /* 输出累加器 ACC 的值 */
0x00
>ACC=R7              /* 将 R7 的值赋给累加器 */
>PCON                /* 输出寄存器 PCON 的内容 */
0x80
```

(3) 函数符号

```
>main                /* 询问 MEASURE.C 中 main 的地址 */
C:0x1395
```

```

>&main                /* 同上, 询问 MEASURE.C 中 main 的地址 */
C:0x1395
>D main                /* 从地址 main 处开始显示存储器内容 */
C:0x1395  75 98 5A D2 DF 43 87 80  u.Z..C..
C:0x139D  75 8C 06 75 8A 06 43 89  u..u..C.
C:0x13A5  02 D2 8C D2 A9 D2 AF 12  .....
C:0x13AD  1A D2 7B FF 7A 0C 79 E7  ..{.z.y.

```

(4) 地址应用

```

&\measure\main\cmdbuf[0] + 0x10  /* 地址计算 */
I:0x21
_RBYTE (C:0x0000)                /* 读出代码地址 C:0x0000 中的内容 */
0x02

```

(5) 符号输出

```

>DIR \measure\main          /* 列出 measure 模块中 main 函数的符号 */
I:0x11  cmdbuf... array[15] of uchar
D:0x22  i... uchar
D:0x23  idx... uint

```

(6) 程序计数器

```

>$ = main                  /* 将程序计数器 PC 置为地址 main 的值 */
>$                          /* 显示当前地址 */
0xFF001395
>DIR                      /* 列出当前函数 main 中的局部符号 */
I:0x11  cmdbuf... array[15] of uchar
D:0x22  i... uchar
D:0x23  idx... uint

```

(7) 程序变量

```

>cmdbuf                  /* 询问 cmdbuf 地址 */
I:0x11
>cmdbuf[0]              /* 输出第一个数组值 */
0x00
>i                      /* 输出变量 i 的值 */
0x00
>idx                    /* 输出变量 idx 的值 */
0x0000
>idx=$                  /* 置变量 idx 的值等于程序计数器的内容 */
>idx                    /* 输出变量 idx 的值 */
0x13C1

```

```

(8) 行号
>\211                                /* 输出行号 211 所对应的地址 */
0xFF0013C1
>$ = \211                            /* 将程序计数器 PC 的值置为程序第
                                     211 行 */
>\MCOMMAND\91                       /* 输出模块 MCOMMAND 第 91 行所
                                     对应的地址 */

0xFF00179B

(9) 运算符
>ACC                                /* 输出累加器 ACC 的值 */
0xFF
>--ACC                              /* 将累加器 ACC 减 1 后输出 */
0xFE
>mdisplay                            /* 输出一个 PUBLIC 符号 */
0
>mdisplay=1                          /* 赋值 */
>mdisplay                            /* 输出 */
1
>ACC==0xFE && R7==0x00 && $==main    /* 输出逻辑表达式的值 */
0x01                                /* 0x01=真, 0x00=假 */

(10) 结构数据
>save_record[0]                     /* 输出结构地址 */
X:0x000000
>save_record[0].time.hour=R7        /* 改变结构成员的值 */
>save_record[0].time.hour          /* 输出结构成员的值 */
0x11
>interval.min=interval.sec=interval.msec=A/2 /* 改变结构成员的值 */
>menu                                /* 输出数组地址 */
C:0x0CE7

(11) 限定符号
>--\measure\main\idx                /* 限定符号自减 */
0xFFFF
>++\measure\main\idx                /* 限定符号自加 */
0x0000

(12) 调用  $\mu$ Vision2 函数
>printf("\muVision2 is coming now!\n") /* 调用  $\mu$ Vision2 内部函数
    \muVision2 is coming now!          显示字符串常数 */

```

```

>_WBYTE (0x2000,_RBYTE(0x2001))          /* 调用 $\mu$ Vision2 内部函数
                                           进行字节读、写 */
>printf( "RegBank=%d\n" ,(PSW&0x18)>>3)    /* 输出当前工作寄存器组 */
RegBank=0
>printf( "RegBank=%d\n" ,(PSW&0x18)>>3)
RegBank=3
>interval.min=getint( "enter integer:" )     /* 调用 $\mu$ Vision2 函数输入整数 */
>interval.min                                /* 输出变量的值 */
0x15

```

7.5 μ Vision2 的函数

μ Vision2 支持 C 语言风格的函数, 包括 μ Vision2 预定义内部函数、用户自定义函数以及信号函数。描述 μ Vision2 函数的语言是 C 语言的一个子集。所有的控制流程语句如 for、while、do、if、switch、break、continue、goto 等都与 C 语言类似, 它们的功能也完全相同。函数中允许传递参数, 可以定义局部变量。

通过执行包括命令序列或重复计算表达式的用户函数, 可扩展 μ Vision2 的命令功能。后台执行的信号函数可在被仿真单片机的引脚上 (VTREG) 提供不同波形和频率的模拟信号, 从而可实现用户系统的实时仿真调试。预定义的 μ Vision2 内部函数可提供一些常用的功能, 如格式化输出、存储器初始化等。

在调试状态下单击“Debug 菜单/Function Editor”选项, 打开 μ Vision2 的内置函数编辑器创建用户函数并进行编译, 无编译错误时函数被加入到内部函数表列中, 一个函数文件中可定义多个 μ Vision2 函数。在编辑状态下单击“Project 菜单/Options for Target/Debug/Initializatuin File”选项, 可以规定在启动 μ Vision2 调试器的同时加载函数文件, 也可用 INCLUDE 命令加载函数文件, 或者直接从命令行或另一个函数中进行调用。使用“DIR FUNC”命令可以列出所有内部函数、已加载的用户函数和信号函数, 使用“DIR BFUNC”可列出所有内部函数, 使用“DIR UFUNC”可列出已加载的用户函数, 使用“DIR SFUNC”可列出已加载的信号函数, 不再需要的函数可通过“KILL”命令删除。

7.5.1 内部函数

表 7-1 列出了全部 μ Vision2 预定义内部函数, 它们不用加载即可随时调用, 但不能被重新定义, 也不能被删除。

表 7-1 μ Vision2 预定义内部函数

类 型	函 数 名	参 数 表	功 能
void	exec	("command_string")	执行命令
double	getdbl	("prompt_string")	要求用户输入双精度型数据
int	getint	("prompt_string")	要求用户输入整型数据

续表

类 型	函 数 名	参 数 表	功 能
long	getlong	("prompt_string")	要求用户输入长整型数据
void	memset	(start_addr, len, value)	初始化存储器
void	printf	("string" ,...)	格式化输出
int	rand	(int seed)	产生随机数
void	rwatch	(address)	延时执行信号函数, 直到指定的存储器地址被读取
void	wwatch	(address)	延时执行信号函数, 直到指定的存储器地址被写入
void	swatch	(float second)	按指定秒数延时执行信号函数
void	twatch	(long states)	按指定 CPU 周期数延时执行信号函数
int	_TarskRunning	(long func_address)	仅采用 RTX 系统时有效, 检查指定任务函数是否为当前运行任务
char	_RBYTE	(address)	从指定存储器地址读取一个字符型数据
int	_RWORD	(address)	从指定存储器地址读取一个整型数据
long	_RDWORD	(address)	从指定存储器地址读取一个长整型数据
float	_RFLOAT	(address)	从指定存储器地址读取一个浮点型数据
double	_RDOUBLE	(address)	从指定存储器地址读取一个双精度型数据
void	_WBYTE	(address,char val)	向指定存储器地址写入一个字符型数据
void	_WORD	(address,char val)	向指定存储器地址写入一个整型数据
void	_DWORD	(address,char val)	向指定存储器地址写入一个长整型数据
void	_WFLOAT	(address,char val)	向指定存储器地址写入一个浮点型数据
void	_WDOUBLE	(address,char val)	向指定存储器地址写入一个双精度型数据

1. 执行命令函数

函数原型: void exec("command_string")

功能: 在用户函数和信号函数中执行有效的 μ Vision2 命令, 参数表中可包含多个用分号“;”隔开的有效的 μ Vision2 命令。

例 7.55:

```
>exec( "DIR PUBLIC; EVAL dptr+r7" )
>exec( "BS timer0" )
>exec( "BK *" )
```

2. 数据输入函数

函数原型: double getdbl("prompt_string")

int getint("prompt_string")

long getlong("prompt_string")

功能: 调用上述函数时将弹出如图 7.85 所示数据输入窗口, 从中输入数据 (只能输入数值, 不允许有运算符), 单击“OK”按钮后在命令窗口返回用户输入的双精度型数据、整型数据、长整型数据的数值。



图 7.85 数据输入窗口

例 7.56:

```
>getdbl("Enter double please: ")
```

从数据输入窗口中键入: 0x7fffffff

在命令窗口中输出: 2147483647

```
>getint("Enter int please: ")
```

从数据输入窗口中键入: 32768

在命令窗口中输出: 0x8000

```
>getlong("Enter long please: ")
```

从数据输入窗口中键入: 12345678

在命令窗口中输出: 0xBC614E

3. 初始化存储器函数

函数原型: void memset(start address, ulong length, uchar value)

功能: 该函数用参数“value”规定的值初始化指定地址范围的存储器空间, 其中第一个参数“start address”表示起始地址, 第二个参数“ulong length”表示存储器长度, 第一个参数应带有存储器类型。

例 7.57:

```
>memset(X:0x2000,0x1000,'a') /* 将'a' 写入片外 RAM 0x2000~0x3000 单元 */
```

```
>memset(D:0x20,0x10,1) /* 将片内 RAM 0x20~0x2f 单元置 1 */
```

```
>memset(D:0x30,0x7F,r7) /* 将 r7 的内容写入片内 RAM 30H~7FH 单元 */
```

4. 格式化输出函数

函数原型: void printf("format_string",...)

功能: 与 Cx51 库函数中的 printf() 函数功能相同, 完成格式化输出。第一个参数必须是一个格式化字符串, 其余参数可以是表达式或字符串。

例 7.58:

```
>printf("random number=%04XH\n", rand(0))
```

```
random number=0029H
```

```
>printf("%u\n", (unsigned long)-1)
```

```
4294967295
```

5. 随机函数

函数原型: int rand(int seed)

功能: 产生并返回一个在-32768~32767 之间的随机整数, 如果其参数值不为 0, 则由

该参数值对随机数发生器进行初始化。

例 7.59:

```
>rand(0x1234)
0x3B98
>rand(0)
0x64BD
>rand(0)
0x12B5
```

6. 周期延时函数

函数原型: void twatch(long states)

功能: 该函数只能由信号函数调用并在 μ Vision2 后台执行, 产生指定机器周期(即 CPU 周期)值的延时基准。请参见信号函数一节。

例 7.60: 下面的信号函数每隔 200000 个机器周期改变一次 P3.2 (INT0) 的电平。

```
>signal void int0_signal (void) {
    while (1) {
        PORT3 |= 0x04;      /* 将 INT0(P3.2) 电平拉高 */
        PORT3 &= ~0x04;     /* 将 INT0(P3.2) 电平拉低并产生中断 */
        PORT3 |= 0x04;      /* 再次将 INT0(P3.2) 电平拉高 */
        twatch (200000);    /* 延时 200000 个机器周期 */
    }
}
```

7. 秒延时函数

函数原型: void swatch(float seconds)

功能: 该函数只能由信号函数调用并在 μ Vision2 后台执行, 产生指定秒时间的延时。

例 7.61: 下面的信号函数每隔 0.5 秒改变一次 P3.2 (INT0) 的电平。

```
>signal void int0_signal (void) {
    while (1) {
        PORT3 |= 0x04;      /* pull INT0(P3.2) high */
        PORT3 &= ~0x04;     /* pull INT0(P3.2) low and generate interrupt */
        PORT3 |= 0x04;      /* pull INT0(P3.2) high again */
        swatch (0.5);       /* wait for 1 second */
    }
}
```

8. 存储器地址读延时函数

函数原型: void rwatch(address)

功能: 该函数只能由信号函数调用并在 μ Vision2 后台执行, 产生连续延时直到对指定地址的存储器单元进行读取时为止。

例 7.62: 下面的信号函数每当对 XDATA 地址 0x1234 进行读取时改变一次 P1.0 的电平。

```
>signal void my_signal (void) {
    while (1) {
        PORT1 ^= 0x01; /* toggle P1.0 */
        rwatch (X:0x1234); /* wait until X:0x1234 is read */
    }
}
```

9. 存储器地址写延时函数

函数原型: void wwatch(address)

功能: 该函数只能由信号函数调用并在 μ Vision2 后台执行, 产生连续延时直到对指定地址的存储器单元进行写入时为止。

例 7.63: 下面的信号函数每当对 XDATA 地址 0x4000 进行写入时改变一次 P1.0 的电平。

```
>signal void my_signal (void) {
    while (1) {
        PORT1 ^= 0x01; /* toggle P1.0 */
        wwatch (X:0x4000); /* wait until X:0x4000 is written */
    }
}
```

10. 任务运行延时函数

函数原型: int _TaskRunning_(ulong func_address)

功能: 该函数只能够用于 RTX51 实时操作系统, 检查指定的任务函数是否为当前运行任务。该函数可以与系统变量 “_break_” 一起使用, 当激活指定任务时暂停用户的运行。

例 7.64:

```
>_TaskRunning_ (command)          /* 检查任务 “command” 是否正在运行 */
0001                             /* 任务正在运行则返回 “1” */
>_break_ = _TaskRunning_ (init) /* 当任务 “init” 运行时暂停用户程序 */
```

11. 读取指定地址的内容函数

函数原型: uchar _RBYTE(address)

uint _RWORD(address)

ulong _RDWORD(address)

float _RFLOAT(address)

double _RDOUBLE(address)

功能: 这些函数返回指定存储器地址单元的内容。

例 7.65:

```
>_RBYTE (0x20000) /* 从地址 0x20000 返回字符型数据 */
>_RFLOAT (0xE000) /* 从地址 0xE000 返回 FLOAT 型数据 */
>_RDWORD (0x1000) /* 从地址 0x1000 返回长整型数据 */
```

12. 向指定地址写入数据函数

函数原型: `uchar _WBYTE(address)`
`uint _WORD(address)`
`ulong _DWORD(address)`
`float _WFLOAT(address)`
`double _WDOUBLE(address)`

功能: 这些函数向指定存储器地址单元写入相应数据。

例 7.66:

```
>_WBYTE (0x20000, 0x55)          /* 向地址 0x20000 写入字符型数据 0x33 */
>_RFLOAT (0xE000, 1.5)           /* 向地址 0xE000 写入浮点型数据 1.5 */
>_RDWORD (0x1000, 12345678)      /* 向地址 0x1000 写入长整型数据 12345678 */
```

7.5.2 用户函数

用户函数可用于扩展 μ Vision2 命令或在函数中重复调用某个表达式, μ Vision2 的命令都可以在用户函数和信号函数中使用。创建一个用户函数的格式如下:

```
FUNC 类型 函数名(参数表) {
    函数体 }
```

其中,“FUNC”是定义函数和引用 μ Vision2 内部编译器的关键字,创建一个函数时该关键字是必不可少的。“类型”是函数返回值的类型,若省略类型说明,返回值默认为 int 类型。“函数名”是所定义的用户函数名, μ Vision2 使用该名字来进行用户函数调用。“参数表”中包含用逗号隔开的各个参数名及其类型, μ Vision2 函数最多可带有 8 个参数。花括号“{ }”中的内容是 C 语言风格的函数体,其中可定义局部变量。可以在 μ Vision2 的命令窗口中直接创建用户函数,也可以通过 μ Vision2 的内置函数编辑器创建用户函数。

例 7.67: 用 μ Vision2 命令方式创建一个用户函数。在 μ Vision2 的命令窗口按如下格式输入用户函数文本,输入完毕(即最后一个花括号)后 μ Vision2 将自动编译该函数,编译通过后可随时调用,但退出 μ Vision2 时该函数将被自动删除。

```
>FUNC int uadd(int x,int y) {
    return(x+y);
}
```

调用该函数:

```
>uadd(1,2)
0x0003
```

例 7.68: 利用 μ Vision2 内置函数编辑器创建一个用户函数。在调试状态下单击“Debug 菜单/Function Editor”选项,从打开的 μ Vision2 内置函数编辑器中键入如下用户函数文本,并将其存盘为“MuRegs.fnc”的函数文件:

```
FUNC void display_regs(void) {
    printf("CURRENT REGISTER VALUES: \n");
    printf("R0=%02xH R1=%02xH R2=%02xH R3=%02xH",r0,r1,r2,r3);
}
```

```

printf("R4=%02xH R5=%02xH R6=%02xH R7=%02xH\n",r4,r5,r6,r7);
printf("A=%02xH B=%02xH SP=%02xH P0=%02xH\n",ACC,B,SP,P0);
}
FUNC void display_pc(void) {
    printf("CURRENT PC VALUE: %04xH\n", $);
}

```

键入完成后单击“Compile”按钮对上述用户函数进行编译处理，编译通过后文件中所有函数都被输入到 μ Vision2 的函数表中，可随时调用执行，退出 μ Vision2 时该函数文件不会被删除。

例 7.69：用户函数列表。在 μ Vision2 的命令窗口中键入 DIR UFUNC 命令将列出所有已经加载的用户函数，键入 DIR BFUNC 命令将列出 μ Vision2 的内部函数，键入 DIR FUNC 命令将列出所有函数：

```

>DIR UFUNC                      /* 用户函数列表 */
user: void DISPLAY_PC(void)
user: void DISPLAY_REGS(void)
>
>DIR BFUNC                      /* 列出 $\mu$ Vision2 的内部函数 */
predef'd: void MEMSET ( ulong, ulong, uchar )
predef'd: void TWATCH ( long )
predef'd: int RAND ( uint )
predef'd: uchar TIMEWAIT ( uint )
predef'd: void KEYWAIT ( char * )
predef'd: float GETFLOAT ( char * )
predef'd: long GETLONG ( char * )
predef'd: int GETINT ( char * )
predef'd: void EXEC ( char * )
predef'd: void PRINTF ( char *, ... )

>DIR FUNC                      /* 列出所有函数 */
user: void DISPLAY_PC(void)
user: void DISPLAY_REGS(void)
user: int UADD(INT, INT)
predef'd: void MEMSET ( ulong, ulong, uchar )
predef'd: void TWATCH ( long )
...
predef'd: long GETLONG ( char * )
predef'd: int GETINT ( char * )
predef'd: void EXEC ( char * )
predef'd: void PRINTF ( char *, ... )

```

例 7.70：用户函数应用。

```

>display_regs()                /* 调用 display_regs() 显示寄存器当前值 */

```

```

CURRENT REGISTER VALUES:
R0=00H R1=00H R2=00H R3=00H R4=00H R5=00H R6=00H R7=00H
A=00H B=00H SP=07H P0=00H
>display_pc()                /* 调用 display_pc() 显示程序计数器当前值 */
CURRENT PC VALUE: ff000e00H
>$=main                      /* 将程序计数器当前值置为地址 main */
>display_pc()                /* 调用 display_pc() 显示程序计数器当前值 */
CURRENT PC VALUE: ff000de3H

```

例 7.71: 利用 KILL FUNC 命令删除用户函数。

```

>DIR UFUNC                  /* 列出用户函数 */
user: void DISPLAY_PC(void)
user: void DISPLAY_REGS(void)
user: int UADD(INT, INT)
>KILL FUNC uadd             /* 删除一个用户函数 */
>DIR UFUNC                  /* 用户函数列表 */
user: void DISPLAY_PC(void)
user: void DISPLAY_REGS(void)

```

7.5.3 信号函数

μ Vision2 提供一种信号函数, 利用该函数可在虚拟的 8051 单片机 I/O 引脚上(VTREG)产生不同波形的模拟信号。信号函数可用于模拟测试串行 I/O、模拟 I/O、接口通信以及其他外部事件。信号函数是在 μ Vision2 的后台执行的, 它可以调用预定义内部函数和用户函数, 但不能调用其他信号函数, 用户函数可以调用信号函数。信号函数应至少调用内部函数 twatch() 一次, 否则将不能正常运行。信号函数只有运行和空闲两种状态, 信号函数在调用内部函数 twatch() 期间为空闲态, 空闲时间取决于传递给 twatch() 函数的参数值。当信号函数被其他函数调用时, μ Vision2 将该信号函数加入到函数表中并标记为运行态。一个信号函数只能被调用激活一次。通过命令“SIGNAL STATE”可以查看信号函数的状态, 通过命令“SIGNAL KILL”可以将信号函数从函数表中删除, 被删除的信号函数还可被再次调用。

创建信号函数的语法格式与创建用户函数类似, 只要将关键字 FUNC 改为 SIGNAL:

```

SIGNAL void 函数名(参数表) {
    函数体 }

```

信号函数的返回类型必须为 void, 最多允许有 8 个参数。

例 7.72: 下面是一个由内置函数编辑器创建的名为 ANALOG.FNC 的信号函数文件, 它以 0.5V 的增、减幅度向 Infineon 80C515 单片机内部 A/D 转换器的模拟输入通道 0 施加一个三角波电压, 电压变化的上限值由函数参数“float limit”给定, 两次电压变化的时间间隔为 25000 个机器周期。

```

SIGNAL void analog0 (float limit) {
    float volts;

```

```

printf ("ANALOG0 (%f) ENTERED\n", limit);
while (1) {
    volts = 0;
    while (volts <= limit) {
        ain0 = volts;          /* 模拟输入通道 0 */
        twatch (25000);        /* 调用 twatch(), 延时 25000 个机器周期 */
        volts += 0.5;          /* 增加输入电压 */
    }
    volts = limit-0.5;
    while (volts >= 0.5) {
        ain0 = volts;
        twatch (25000);        /* 调用 twatch(), 延时 25000 个机器周期 */
        volts -= 0.5;          /* 降低输入电压 */
    }
}
}

```

在 μ Vision2 中调用信号函数的方法与调用内部函数和用户函数类似, 信号函数总是在 μ Vision2 的后台执行, 一个正在执行中的信号函数不能被重复调用。

例 7.73: 信号函数应用示例。本例通过调用上例所创建的 ANALOG.FNC 文件中的信号函数 analog0()在单片机 80C515 的一个虚拟 I/O 引脚上 (VTREG) 产生无限循环的三角波电压信号。

```

>DIR SIGNAL                      /* 输出信号函数名 */
signal: void ANALOG0(float limit)
>ANALOG0(5.0)                    /* 调用信号函数, 电压最大幅值为 5V */
analog0(5.000000) ENTERED
>SIGNAL STATE                    /* 输出当前信号函数的状态 */
0 idle Signal = analog0 (line 43)

```

命令“SIGNAL STATE”的输出表明, 信号函数 analog0()正在执行第 43 行的 twatch() 函数, 因此处于“空闲”状态, 这时开始运行用户程序, 在延时 25000 个机器周期后, 信号函数将从第 44 行继续往下执行, 直到又遇到下一个 twatch()函数。

例 7.74: 任何时候都可根据需要从当前信号函数表中删除 analog0(), 删除之后可再次调用。

```

>SIGNAL KILL ANALOG0             /* 删除信号函数 */
>DIR SIGNAL                      /* 输出信号函数名 */
signal: void ANALOG0(float limit)
>SIGNAL STATE                    /* 输出当前信号函数的状态 */
>                                /* 已从当前信号函数表中删除 */

```

7.5.4 μ Vision2 函数与 Cx51 函数的差别

μ Vision2 函数与 Cx51 函数之间存在如下差别。

- 对大、小写字母不加区别，因此可任意采用大写或小写字母来输入符号或关键字。
- μ Vision2 中没有预处理器，因此 μ Vision2 函数中不允许出现诸如 `#define`、`#include` 等预处理命令语句。
- 只能定义局部变量，变量不能在定义时初始化，而必须采用赋值语句。需要使用全局变量时可利用 `DEFINE` 命令定义符号的方法来代替。
- 不支持结构、数组及指针类型的变量、参数和返回值。
- μ Vision2 函数不能进行直接或间接递归调用。
- 只支持直接函数调用，不允许通过指针间接调用函数。
- 只支持新标准的 ANSI 形式的参数说明。

例 7.75: 函数参数的说明方法。

```
func test(int val1, int val2)    /* ANSI 形式参数说明, 正确 */
{
    /* ..... */
}

func test(val1, val2)           /* 旧形式参数说明, 错误 */
int val1, val2;
{
    /* ..... */
}
```

第 8 章 μ Vision2 环境下 Cx51 编程技巧与应用实例

在 Windows 集成开发环境 μ Vision2 中进行 Cx51 应用程序设计非常方便, μ Vision2 对用户程序按“项目”(Project)进行管理, 一个“项目”中可以包含多个“文件组”(Group), 每个“文件组”中又可以包含多个文件, 这一特点使得用户可以在同一个环境之下进行多模块程序设计, 同时完成各个模块文件的编译和链接工作, 如果发生错误可及时修改, 极大地提高了编程效率。首先单击“Project 菜单/New Project...”选项创建一个项目文件, 根据设计要求为项目选取合适的 CPU 器件、添加不同的文件组。然后单击“File 菜单/New”选项创建用户程序文件, μ Vision2 将自动打开程序文件编辑窗口, 编程结束后将程序文件存盘, 并将已经存盘的程序模块文件添加到各个文件组中。接下来利用“Project 菜单/Options for Target”选项设置应用目标(Target)、输出文件(Output)、列表文件(Listing)、编译控制(C51 或 Cx51)、汇编控制(A51 或 Ax51)、连接定位控制(BL51 或 Lx51)以及仿真调试控制(Debug)等。上述每种设置中都包含若干个复选框, 其中有一些已经设为默认值, 大部分场合直接利用这些默认值即可, 必要时可以适当添加其他控制命令。设置完成之后单击“Project 菜单/Build Target”选项, 对项目中的所有程序模块文件进行编译和连接定位, 生成与项目文件同名的绝对目标代码文件。最后启动 μ Vision2 调试器对该目标代码文件进行仿真调试。

8.1 编写 Cx51 应用程序的基本原则

μ Vision2 本身是一个标准的 Windows 应用程序, 在编辑窗口编写应用程序文件时可以充分利用 Windows 的各种功能, 如剪切、粘贴、复制等。虽然 C 语言程序不要求具有固定的格式, 但我们在实际编写程序时还是应该遵守一定的规则。首先要采用清晰的书写方法。在编写一个 Cx51 程序时, 对于 while、for、do-while、if-else、switch-case 等语句, 或这些语句的嵌套组合, 应采用“缩格”的书写形式。对于复合语句或函数, 通常需要使用花括号“{}”, 当语句嵌套较多时, 容易产生花括号不匹配的情况。 μ Vision2 的 Edit 下拉菜单中提供了一个“Goto Matching Brace”选项, 将光标放在某个括号处, 单击该选项, 与之相匹配括号中的内容将反转显示, 特别适用于检查各种括号的匹配情况。

对于一个表达式中各种运算执行的优先顺序不太明确或容易混淆的地方, 应当采用圆括号“()”明确指定它们的优先顺序。对于程序中的函数, 在使用之前应对函数的类型进行说明。对函数类型的说明必须保持与原来定义的函数类型相一致, 不一致时将导致编译

出错。对于具有返回值的函数，使用 `return` 语句时，最好使用括号“`()`”将被返回的内容括起来，这样可使程序执行过程更清晰，便于理解和维护。

一般情况下，对于普通的变量名或函数名采用小写字母表示，对于一些特殊变量名或由预处理命令 `#define` 所定义的常数，则采用大写字母表示。为了帮助理解和记忆，变量或函数名中可带有下列划线，例如，`ext_int0`、`data_max` 等，但是以下划线“`_`”开头的变量或函数名通常保留为 Cx51 编译系统所使用，为了避免混淆，不要将下划线用做变量或函数名的第一个字符。给变量或函数取名时，应按照见名知义的原则，例如“`ext_int0`”表示外部中断函数，“`data_max`”表示最大数据值等。

数组与指针语句具有十分密切的关系。对一个字符数组：`char * name="hello"`；可以采用数组形式 `name[0]` 或指针形式 `* name` 来表示字符串的第一个字母 `h`，两者在意义上是完全相同的。在实际程序设计中使用数组还是使用指针应视具体情况而定，一般来说，指针比较灵活简洁，而数组则比较直观，容易理解。

C 语言是一种高级程序设计语言，与汇编语言不同，C 语言提供了十分完备的规范化流程控制结构。因此在采用 Cx51 设计单片机应用系统程序时，首先要注意尽可能采用结构化的程序设计方法，这样可使整个应用系统程序结构清晰，便于调试和维护。对于一个较大的应用程序，为了能够集中精力考虑各种具体问题，通常将整个程序按功能分成若干个模块，不同模块完成不同的功能。各个模块程序可以分别编写，甚至可以由多个人员分头编写。由于单个模块程序所完成的功能较为简单，程序的设计和调试也相应要容易一些。对于一些常用的功能模块，可以作为一个应用程序库，以便于以后直接调用。在 C 语言中进行模块化程序设计是比较容易实现的，一个 C 语言函数就可以认为是一个模块。所谓程序的模块化，不仅仅是要将整个程序划分成若干个功能模块，更重要的是还应当注意保持各个模块之间变量的相对独立性，即保持模块的独立性。在 C 语言的模块化编程过程中，如果过多地采用外部变量会减弱各个模块的独立性，因此为了保持整个程序具有较好的模块化结构，应尽量避免使用外部全局变量来传递数据信息，而应通过指定的参数来完成数据信息传递。对于不同的功能模块，可以分别指定相应的入口参数和出口参数，这样不会引起整个程序中变量管理的混乱。在 μ Vision2 中很容易实现模块化编程，只要将分别编写的各个程序模块文件分别添加到项目中就可以了。

在程序设计过程中，对于经常使用的一些常数，如果将它们直接写到程序中去，一旦常数的数值发生变化，就必须逐个找出程序中的所有对应的常数，逐一进行修改，这样必然会降低程序的可维护性和可移植性。因此为了便于对整个程序进行修改维护，或纯粹是为了帮助记忆，应当采用预处理命令的方式来定义常数。对于一些常用的常数，如 π 、 e 、`EOF`、`TRUE`、`FALSE` 以及不同型号 8051 单片机中各种特殊功能寄存器和位地址等，应当集中起来放在一个头文件中进行定义，需要时再采用预处理命令 `#include` 将其包含到程序中去。这样做不仅可以提高编程效率，而且还可以避免输入错误。

一般来说，程序的执行效率主要取决于所采用算法的优劣和繁简。但对 C 语言而言，程序的执行效率在一定程度上还与程序的结构和设计方法有关。C 语言具有十分丰富的运算符，合理地运用这些运算符可以设计出高效率的程序。例如，当条件表达式是由多个“`&&`”或“`||`”运算符连接在一起时，对于条件的判定总是从左至右逐个进行的，一旦条件满足时，就不再对后面其他条件进行判断。因此对于条件表达式的安排，应尽可能地将

满足条件可能性较高的表达式放在整个条件式的前面。合理使用中间变量往往也可以提高程序的执行效率。

8.2 Cx51 应用中的一些常见问题与解决方法

8.2.1 Cx51 程序设计中容易出错的地方

在使用 C 语言进行程序设计时,有一些在语法上是正确的语句,却具有语义上的错误。

1. 赋值运算符“=”与等值运算符“==”

初次使用 C 语言编程时,往往容易将这两个符号弄错。例如,若将条件判断式 `if(x==y);` 误写成 `if(x=y);` 由于 C 语言属于非逻辑型语言,对条件式的判断是按照表达式的值为 1 (真)或 0 (假)来进行的。因此对于上面的例子,在用 Cx51 进行编译时,并不能检查出任何错误。但是这两个条件表达式的意义却完全不同,前者表示当变量 `x` 与变量 `y` 相等时,其结果为真,否则其结果为假。而后者表示将 `y` 赋值给 `x`,只有当 `y` 的值为 0 时,其结果才为假,否则其结果总为真,这样一来就使条件判断与编程者的设想大相径庭。

2. 运算符的优先顺序

在 C 语言设计过程中,如果弄错了运算符的优先顺序将导致意想不到的结果。

① 指针运算符“*”比加法运算符“+”的优先级别高。因此,表达式 `x=*pa+1` 表示取指针 `pa` 所指的内容并将上 1 之后再赋值给变量 `x`,而表达式 `y=*(pa+1)` 则表示取指针 `pa` 的下一个地址中的内容并赋值给变量 `y`,显然这两个表达式的值是不一定相等的。

② “!”的优先级别高于“=”。因此表达式 `(c=getchar())!=EOF` 与表达式 `c=getchar() != EOF` 的结果是不一样的。前者表示先将函数 `getchar()` 的结果赋值给变量 `c`,然后再进行 `!=EOF` (不等于文件结束)判断。而后者则表示先将函数 `getchar()` 的结果与 `EOF` 进行 `!=` (不相等)判断,然后再将判断的逻辑结果赋值给变量 `c`。

③ “+”的优先级别高于“*”。因此表达式 `b*=c+1` 等价于表达式 `b=b*(c+1)`,而等价于表达式 `b=b*c+1`。

④ “++”与指针运算符“*”的优先级别相同,但结合顺序为先“++”后“*”。因此表达式 `*ptr++`; 等价于表达式 `*(ptr++)`; 其结果表示先取 `ptr` 所指的内容,然后将 `ptr` 指向下一个地址。而表达式 `*(ptr)++` 是将 `ptr` 所指的内容加 1。

3. 非法表达式

C 语言中具有十分丰富的运算符,写程序时应注意正确使用,否则将导致表达式非法。

① 地址运算符 `&` 只能用来取出变量的地址。因此表达式 `&(x+1)` 是非法的。

② 数组名是一个常量而不是指针变量。

例如,定义一个二维整型数组: `int array[10][20]`; 这时数组名“`array`”实际上就是代表该数组首地址的常量, `array[0]` 则表示该数组第一行元素的首地址,也是一个常量,因此表达式 `*(array+1)` 是正确的,而表达式 `array++` 则是非法的。

③ 增量运算符“++”和减量运算符“--”只对变量有效。类似于 `(i+j)++` 或 `(i+j)--` 这样

的表达式都是非法的。

④ 对于一般的变量，不能采用指针运算符“*”来取值。因此对于变量 `int x`；表达式 `*x` 是非法的。

⑤ 字符 'x' 与字符串 "x" 区别。在 C 语言中，'x' 与 "x" 的意义是完全不相同的。'x' 代表一个字符常量，并且 'x' 的值就是 x 的 ASCII 码，而 "x" 是一个字符串常量，它表示由字符 x 和转义字符 \0 组成的字符串。因此表达式 `char str="x"`；是非法的，不能将一个字符串赋值给一个字符变量。

4. 数组的下标范围

在 C 语言中如果定义了一个具有 N 个元素的数组，则该数组的下标范围是 0~N-1。例如，对于数组：`int array[5]`；其中共有 `array[0]~array[4]` 这样 5 个有效数组元素，而 `array[5]` 已经不是该数组的有效元素了。另外 Keil Cx51 编译器对于数组的边界不作任何检查，即使数组的下标越界也不会给出错误信息。例如，对于以上数组，如果使用 `array[5]=1`；或 `array[6]=0`；都会被认为是合法的，但是如果地址 `array[5]` 或 `array[6]` 已经分配给了其他的变量，则在程序的执行过程中将会产生不可预料的结果。因此在进行实际程序设计时，必须人为地对数组的下标范围进行正确地管理和核对。

5. 变量的初始化

① 指针变量的初始化。定义一个指针变量仅仅是明确地指定了指针本身所需要的内存空间，而该指针的初始值是它所指向的内存地址。例如，可以采用如下方法进行指针变量的初始化：

```
char * var, buf[20];  
...  
var=buf;
```

一个数组被定义之后，就获得了一个确定的内存区域，因此这样的初始化是有效的，它使指针变量 `var` 指向一个确定的地址（即 `buf[0]`）。

② 编译链接时的初始化与程序执行过程中的初始化具有不同的意义。通常对于静态和外部变量在编译链接时进行初始化，例如：

```
sub() {  
    static int count=0x1234;  
    ...  
}
```

其中静态变量 `count` 是在编译链接时通过运行库中的初始化模块 `INIT.OBJ` 对其进行初始化，而不是在每次调用函数 `sub()` 时都将对其进行初始化。但是如果改成：

```
sub() {  
    static int count;  
    count=0x1234;  
    ...  
}
```

这实际上是程序执行过程中的赋值,因此在每次调用函数 sub()时都会对其进行赋值操作。另外,对于自动变量只能在程序执行过程中进行初始化。例如:

```
sub() {  
    auto int count=0;  
    ...  
}
```

其中自动变量 count 在每次调用函数 sub()时都将被初始化为 0。需要指出的是,由于对静态变量和外部变量的初始化是通过 INIT.OBJ 模块实现的,在具有这类变量的程序开头会增加一段初始化程序 INIT.A51 的代码,这样显然会增加程序的运行时间,同时还会增加程序代码所占用的空间。

6. 结构变量的赋值方式。

在使用结构时不允许将一个结构体变量的内容一次赋值给另一个结构变量。例如,定义如下结构:

```
struct form {  
    int bust;  
    char waist;  
} x, y;
```

此时不能将结构变量 x 的内容整体地赋值给结构变量 y,即语句 y=x; 是无效的。若希望对结构变量进行复制,必须以结构成员为单位逐个复制,即必须采用如下语句:

```
y.bust=x.bust;  
y.waist=x.waist;
```

8.2.2 有关 Cx51 程序设计的若干实际应用技巧

① 通常一个 8051 单片机应用系统程序总是要经过仿真调试,才知道该程序是否能满足要求。为了获得足够多的调试信息,在对一个 Cx51 程序进行编译时,应该采用 CODE、DEBUG、SYMBOLS 以及 OBJECTTEXTEND 控制指令,最好在源程序的开始处加上如下预处理命令行:

```
#pragma CODE DEBUG SYMBOLS OBJECTTEXTEND
```

这样得到的目标程序中将包含所有的调试信息,从而可以方便地进行源程序调试。

② 在采用μVision2 对 Cx51 应用程序进行调试时,为了能从串行窗口中立即看到程序的运行结果,经常需要使用 Cx51 运行库中的输出函数 printf()。为了能获得字节形式的输出结果,可以在 printf()函数中使用“%bd、%bo、%bu、%bx”等格式控制符,或者使用强制类型转换符,一般来说使用前者较为方便。下面是一个例子。

```
#pragma code symbols debug oe  
#include <reg52.h>  
#include <stdio.h>
```

```

struct FS {
    unsigned int co,kp,ki,kd,kl;
};

data struct FS filter;

void main( void ) {
    TI = 1;
    /* 下面这行将导致错误的结果 */
    printf("Start = %p, length = %d\n", &filter,sizeof( struct FS ));
    /* 下面这两行才是正确的 */
    printf("Start = %p, length = %bd\n", &filter,sizeof( struct FS ));
    printf("Start = %p, length = %d\n", &filter,(int )sizeof( struct FS ));
}

```

③ 在进行 8051 单片机应用系统程序设计时，用户十分关心如何直接操作系统的各个存储器地址空间。Cx51 程序经过编译之后产生的目标代码具有浮动地址，其绝对地址必须经过 BL51 连接定位后才能确定。为了能够在 Cx51 程序中直接对任意指定的存储器地址进行操作，可以采用指针的办法来实现。例如：

```

void testmodule( void ) {
    char xdata *xdp; /* 定义一个指向 XDATA 存储器空间的指针 */
    char data *dp;   /* 定义一个指向 DATA 存储器空间的指针 */
    xdp = 0x0002;    /* XDATA 指针赋值，指向 XDATA 存储器地址 0002h */
    *xdp = 0xAA;     /* 将数据 0xAA 送往指定的地址 */
    dp = 0x30;       /* DATA 指针赋值，指向 DATA 存储器地址 30H */
    *dp = 0xBB;      /* 将数据 0xBB 送往指定的地址 */
}

```

另外一种直接操作存储器空间地址的方法是利用 Cx51 运行库中提供的一套预定义宏“ABSACC.H”，例如：

```

#include <ABSACC.H>
char c_var;
int i_var;
XBYTE[0x12]=c_var; /* 向 XDATA 存储器地址 0012H 写入数据 c_var */
i_var=XWORD[0x100]; /* 从 XDATA 存储器地址 0200H 中读取数据并赋值给 i_var */

```

上面第二条赋值语句中采用的是 XWORD[0x100]，其意义是将字节地址 0x200 和 0x201 中内容取出来并赋值给 int 型变量 i_var，注意不要将 XWORD 与 XBYTE 混淆。如果将这条语句改成：

```
i_var=XWORD[0x100/2];
```

这样读取的就是 0x100 和 0x101 地址单元中的内容了。用户可以充分利用 Cx51 运行

库中提供的预定义宏“ABSACC.H”来进行任意地址的直接操作。例如，可以采用如下方法定义一个 D/A 转换接口地址，每向该地址写入一个数据即完成一次 D/A 转换：

```
#include <ABSACC.H>
#define DAC0832 XBYTE[0x7fff] /* 定义 DAC0832 端口地址 */
DAC0832=0x80; /* 启动一次 D/A 转换 */
```

还可以在 Cx51 源程序中定义变量时利用 Cx51 编译器提供的扩展关键字“_at_”来对指定存储器空间的绝对地址进行定位，一般格式如下：

· [存储器类型]数据类型标识符 _at_ 常数

其中，“存储器类型”为 idata、data、xdata 等 Cx51 编译器能够识别的所有类型，如果省略该选项，则按编译模式 LARGE、COMPACT 或 SMALL 规定的默认存储器类型确定变量的存储器空间；“数据类型”除了可用 int、long、float 等基本类型外，还可以采用数组、结构等复杂数据类型；标识符为要定义的变量名；常数规定了变量的绝对地址，它必须位于有效存储器空间之内。利用扩展关键字“_at_”定义的变量称为“绝对变量”，对该变量的操作就是对指定存储器空间绝对地址的直接操作，因此不能对“绝对变量”进行初始化，另外函数和位（bit）类型变量不能采用这种方法进行绝对地址定位。

例如：

```
struct link {
    struct link idata *next;
    char code *test;
};
idata struct link list _at_ 0x40; /* 结构变量 list 定位于 idata 空间地址 0x40 */
xdata char text[256] _at_ 0xE000; /* 数组 array 定位于 xdata 空间地址 0xE000 */
xdata int i1 _at_ 0x8000; /* int 变量 i1 定位于 xdata 空间地址 0x8000 */

void main ( void ) { /* 在主函数中直接操作上述已定义的绝对地址 */
    list.next = (void *) 0;
    i1 = 0x1234;
    text [0] = 'a';
}
```

④ 能够进行指针操作是 C 语言的一个重要特征。对函数而言也可以进行指针操作，这就是函数指针。函数指针指向的是函数体中第一条可执行指令的地址，即函数的入口地址。在调用函数时，可以采用函数指针来实现。由于函数指针只是用来存放函数的入口地址，而不是固定指向某个函数，因此可以通过给函数指针赋以不同的值而实现不同函数的调用。在给函数指针赋值时，只要给出函数名而不必给出函数的参数。通过函数指针来调用函数时，只需用已赋值的函数指针代替被调用的函数名，这样通过一个函数指针可调用不同的函数。下面是一个通过函数指针实现不同函数调用的例子。

```
#pragma cd sb db oe
#include <reg51.h>
#include <stdio.h>
```

```

void func1( ) {
    printf("In FUNC1()\n");
}

void func2( ) {
    printf("In FUNC2()\n");
}

void main( void ) {
    void (*f)();      /* 定义一个函数指针 */
    SCON = 0x50;      /* 8051 串行口初始化, 8 位 UART, 允许接收 */
    TMOD |= 0x20;     /* 定时器 T1, 工作方式 2, 8 位重装方式 */
    TH1 = 0xf3;       /* 波特率为 2400 */
    TR1 = 1;          /* 启动定时器 T1 */
    TI = 1;

    While( 1 ) {
        f = func1;
        f();          /* 调用函数 func1() */
        f = func2;
        f();          /* 调用函数 func2() */
    }
}

```

⑤ 在 Cx51 的实际应用中, 如果需要将一些特殊数据或某个接口的特殊功能函数定位在指定的地址处, 则必须采用连接定位器 BL51 来完成。关于 BL51 对程序中的各种代码段和数据段的定位方法在本书第十一章中作了详细介绍, 这里对实际使用中可能遇到的问题作一点补充。

在连接定位的过程中会涉及一种所谓的“应用段”, 其中包含代码和数据两部分内容, 它们可能是由用户程序产生的, 也可能是由应用工具(如编译器、汇编器或连接定位器等)所产生的。

应用段的数据部分包含如下内容。

DATA_GROUP	在 SMALL 模式下所有被连接的目标模块中可覆盖的“自动变量”和“传递参数”的组合。
XDATA_GROUP	在 LARGE 模式下所有被连接的目标模块中可覆盖的“自动变量”和“传递参数”的组合。
PDATA_GROUP	在 COMPACT 模式下所有被连接的目标模块中可覆盖的“自动变量”和“传递参数”的组合。
REG BANK#	工作寄存器组段, “#”为工作寄存组号, 其值为 0~3。
?STACK	再定位堆栈段。该符号定义在启动程序 STARTUP.A51 中。BL51 将在内部 IDATA 区最后定位该段。

应用段的代码部分包含如下内容。

?C_C51STARTUP	启动程序模块 STARTUP.A51 中的起始代码段名。
?C_LIB_CODE	用户程序中使用了的运行库模块代码。
?CO?module	用户模块中的常数代码段。
?C_START	这个标号是启动程序中最后一条转移语句的符号地址，连接定位器将运行库中的初始化模块定位到该地址处，并开始执行 Cx51 程序的 main() 函数。如果用户希望转移到一个自己的汇编程序模块，可用自己的汇编程序模块入口地址取代启动程序中的 ?C_START。
?C_INITSEG	当需要对 DATA 存储器区进行初始化时，运行库将调用这个初始化段。

应用段中还包含有一些别的段内容，而上面这些是最常见的，它们的绝对地址都显示在 BL51 所产生的映像文件 (.M51 或 .Map) 中，可随时调出来察看，以确定它们的定位地址是否合理。

例如，用户模块文件 exm.c 中使用了下面的函数：

```
ex(){  
    char var1;  
    int var2;  
    ...  
}
```

函数 ex() 包含两个变量，其中字符型变量 var1 是自动变量，属于 DATA_GROUP 的内容，它不能被单独取出来重新定位。如果必须将 var1 置于一个特定的地址处，可以选用下面的一种方法。

- 将 var1 放在一个具有绝对段的汇编语言程序模块中，然后再将该汇编程序模块与其他 C51 程序模块连接在一起。
- 将 var1 单独放在一个函数中，使之具有一个独立的数据段，在调用 BL51 时可将这个数据段定位在指定的地址处。对于上例中的 var1 可在模块文件 exm.c 中安排一个仅包含变量 var1 的函数 test1()，函数 ex() 中则不再包含变量 var1：

```
test1(){  
    char var1;  
    ...  
}  
  
ex(){  
    int var2;  
    ...  
}
```

若需要将 var1 定位在 DATA 存储器的 30H 地址处，则可采用如下连接控制命令：

BL51 exm.obj DATA(?DT?test1?exm(30H))

在 μ Vision2 环境下上述命令只要在“Project 菜单 / Options 选项 / BL51 Locate 标签页 / Data 栏 / ”对应的“Segments”框中键入如下段名即可：

?DT?test1?exm(30H)

若需要将 var1 定位在 XDATA 存储器的 0E000H 地址处,则可采用如下连接控制命令:

BL51 exm.obj XDATA(?XD?test1?exm(0E000H))

在 μ Vision2 环境下上述命令只要在“Project 菜单 / Options 选项 / BL51 Locate 标签页 / Xdata 栏 / ”对应的“Segments”框中键入如下段名即可:

?XD?test1?exm(0E000H)

如果需要将 C51 程序模块中的某个函数定位在指定的地址处,可采用类似的方法。例如下面的命令行将函数 test1()定位在 CODE 区域的 1000H 地址处:

BL51 exm.obj CODE(?PR?test1?exm(1000H))

在 μ Vision2 环境下上述命令只要在“Project 菜单 / Options 选项 / BL51 Locate 标签页 / Code 栏 / ”对应的“Segments”框中键入如下段名即可:

?PR?test1?exm(1000H)

而下面的命令行将整个 exm.obj 模块定位在 CODE 的 1000H 地址处:

BL51 exm.obj CODE (1000H)

在 μ Vision2 环境下上述命令只要在“Project 菜单 / Options 选项 / BL51 Locate 标签页 / Code 栏 / ”对应的“Segments”框中键入地址 1000H 即可。

对于常数表格的定位也可以采用类似的方法,例如,对于常数表格:

```
char code table[]={1,2,3,4,5};
```

若希望将其定位到以地址 0x7000 开始的代码空间,只需要在“Project 菜单 / Options 选项 / BL51 Locate 标签页 / Code 栏”对应的“Segments”框中键入如下段名即可:

?CO?TABLE (0x7000)

这样, table[]表就被定位到了 0x7000 开始的程序段。

⑥ 在使用 BL51 进行连接定位时,有时需要采用 OVERLAY 指令对 BL51 的覆盖分析作适当的调整,否则将产生错误的结果。例如,下面的程序模块 exm1.c 中,主程序 main()和中断服务函数 timer0()都调用了库函数 printf():

```
#pragma code symbols debug oe
#include <stdio.h>
void func ( void ) {
    printf ("12345\n");
}
code void (*func_array[])() = {func};
void main ( void ) {
    (*func_array[0])();
}
```

由于 printf()函数是非再入函数,如果不加调整地进行编译链接,将产生如下警告错误:


```
*** WARNING L15: MULTIPLE CALL TO SEGMENT
SEGMENT: ?PR?PRINTF?PRINTF
CALLER1: ?PR?TIMER0?CLK
CALLER2: ?C_C51STARTUP
```

表示在所链接的程序模块中存在对同一个代码段的多重调用错误,这种错误通常发生在主函数和中断服务函数需要同时调用另一个函数时。避免产生这类错误最简单的方法是禁止BL51对被调用的函数进行覆盖分析,对于本例可以采用如下链接控制命令:

BL51 EXM1.OBJ OVERLAY (?PR?PRINTF?PRINTF!*)

在 μ Vision2 环境下,只要在“Project 菜单 / Options 选项 / BL51 Misc 标签页 / ”中的“Overlay”框中键入需要禁止覆盖分析的代码段名即可:

?PR?PRINTF?PRINTF!*

这样就可以将 printf() 函数排除在覆盖分析之外,即不对它的数据段进行覆盖,从而实现正确的定位。

⑦ Cx51 编译器对于程序中未被调用的函数(如调试函数等)不会产生任何错误信息。但是当使用 BL51 对包含这种未调用函数的 Cx51 程序模块进行链接定位时,则会产生如下警告错误:

UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS.

如果程序模块中包含有直接用 Cx51 编写的中断函数,并且在编译时采用了编译控制指令“NOINVECTOR”抑制中断向量的产生,则在使用 BL51 进行连接定位时也会产生这种错误。BL51 将这种未调用函数排除在覆盖过程之外,即未调用函数将具有其自己独立的数据空间,显然这时将耗用更多的存储器空间。因此在调试 Cx51 程序的过程中发现存储器空间不够时,应注意察看 BL51 所产生的映像文件,以确定是否存在未调用函数。对于附加的调试函数,在调试通过后应及时去掉,对于 Cx51 中断函数的入口地址向量则应作适当处理。

⑧ 在 Cx51 程序中 BDATA 类型变量和位变量应定义成全局变量,而不要定义在某一个函数内部,否则将发生编译错误。

8.3 8051 单片机的片内定时器应用编程

本节通过对 8051 单片机片内定时器的应用,介绍了在 μ Vision2 环境下进行 Cx51 应用程序设计的具体过程,包括多模块编程、片上集成外围功能应用、代码分组应用、软件模拟仿真调试以及用户硬件目标板仿真调试等内容。

8.3.1 多模块编程

利用 8051 内部定时器 T0 定时中断实现实时时钟。这是一个典型多模块应用程序,由 9 个模块文件组成:启动代码模块文件 Startup.a51、主程序模块文件 main.c、定时器模块文件 Timer.c、时钟模块文件 Clock.c、命令创建模块文件 Cmdbuild.c、命令处理模块文件 Comproc.c、串行通信模块文件 Serial.c、硬件目标板初始化模块文件 Dinput.a51、硬件目

标板显示处理模块文件 **DISPLY.A51** 以及应用函数说明头文件 **TDP.H**。启动代码模块文件 **Startup.a51** 是 Keil Cx51 软件包中自带的一个汇编语言程序, 位于 Keil\C51\LIB 目录之下, 其中有许多可配置的 EQU 语句, 可按需要进行修改。下面列出了其他模块文件代码。

主程序模块文件 **MAIN.C** 代码:

```
#include <reg51.h>
#include <stdlib.h>
#include "tdp.h"
#ifdef evalboard      /* 采用硬件目标板时需要使用以下外部函数 */
extern void DINPUT(unsigned char aa, unsigned char bb);
extern void DISPLY(unsigned char data *cc);
static unsigned char data dg[]={0,0,10,0,0,10,0,0};
#endif

void main (void) {
    unsigned i;
    for (i = 0; i < 1000; i++);          /* 延时, 等待系统上电稳定 */
    timer0_initialize ();                /* 定时器 0 初始化 */
    com_initialize ();                  /* 串行口初始化 */
#ifdef evalboard                      /* 采用硬件目标板时 */
    DINPUT(0X0A,0X07);                 /* 需要对板上 MAX7219 初始化 */
    DINPUT(0X0B,0X07);
    DINPUT(0X09,0X0FF);
    DINPUT(0X0c,0X01);
    DISPLY(dg);                        /* 板上 LED 显示 00-00-00 */
#endif
    timer0_wait (TIMER0_TICKS_PFR_SEC / 10);
    clock_init ();
    com_puts ("TDP V1.0\r\n");
    while (1) {
        const char *cmd;
        cmdb_init ();
        cmdb_prompt ();
        for (cmd = NULL; cmd == NULL; cmd = cmdb_scan ()) {
            clock_update ();
        }
        cmd_proc (cmd);
    }
}
```

定时器模块文件 **TIMER.C** 代码:

```
#include "tdp.h"
#include <reg51.h>
```

```

#define TIMER0_COUNT    0xDC11  /* 10000h-((11059200Hz/(12*FREQ))-17) */

static xdata unsigned timer0_tick;

/* 定时器 T0 中断服务函数 */
static void timer0_isr (void) interrupt 1 using 1 {
    P1 |= 0x80;
    TR0 = 0;          /* 停止 T0 */
    TL0 = TL0 + (TIMER0_COUNT & 0x00FF);
    TH0 = TH0 + (TIMER0_COUNT >> 8);
    TR0 = 1;          /* 启动 T0 */
    timer0_tick++;
    P1 &= ~0x80;
}

/* 定时器 T0 初始化函数 */
void timer0_initialize (void) {
    INT_DISABLE;      /* 关中断 */
    timer0_tick = 0;
    TR0 = 0;          /* 停止 T0 */
    TMOD &= ~0x0F;     /* 设置 T0 工作方式 */
    TMOD |= 0x01;
    TL0 = (TIMER0_COUNT & 0x00FF);
    TH0 = (TIMER0_COUNT >> 8);
    PT0 = 0;          /* 设置 T0 中断优先级 */
    ET0 = 1;          /* 允许 T0 中断 */
    TR0 = 1;          /* 启动 T0 */
    INT_ENABLE;       /* 开中断 */
}

/* 定时器 T0 节拍函数 */
unsigned timer0_count (void) {
    xdata unsigned t;
    INT_DISABLE;
    t = timer0_tick;
    INT_ENABLE;
    return (t);
}

/* 定时器 T0 节拍计数函数 */
unsigned timer0_elapsed_count (unsigned count) {
    return (timer0_count () - count);
}

```

```

/* 定时器 T0 延时函数 */
void timer0_wait (unsigned count) {
    xdata unsigned start_count;
    start_count = timer0_count ();
    while (timer0_elapsed_count (start_count) <= count)
    {
    }
}

```

时钟程序模块文件 CLOCK.C 代码:

```

#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include "tdp.h"
#ifdef evalboard          /* 采用硬件目标板时需要使用以下外部函数 */
extern void DINPUT(unsigned char aa, unsigned char bb);
extern void DISPLY(unsigned char data *cc);
#endif

/* 全局变量定义 */
static xdata unsigned long dayhsecs;
static xdata unsigned last_tick;
static xdata unsigned char scan_flag;
static xdata unsigned char alm_flag;
static xdata unsigned almmmins;

#define MAX_HSEC_DAY      (100 * 60 * 60 * 24)
#ifdef evalboard          /* 采用硬件目标板时需要使用以下数组 */
static unsigned char data dg[]={0,0,10,0,0,10,0,0};
#endif

/* 时钟初始化函数 */
void clock_init (void) {
    scan_flag = 0;
    alm_flag = 0;
    dayhsecs = 0L;
    last_tick = timer0_count ();
}

/* 时钟更新函数 */
void clock_update (void) {
    static xdata unsigned long last_daysecs;

```

```
    dayhsecs += timer0_elapsed_count (last_tick);
    last_tick = timer0_count ();
    while (dayhsecs >= MAX_HSEC_DAY)
        dayhsecs -= MAX_HSEC_DAY;
    if ((dayhsecs / 100) == last_daysecs)
        return;
    last_daysecs = dayhsecs / 100;
    if (alm_flag != 0)
        if ((dayhsecs / (100 * 60)) == almmmins)
            com_putchar ('\x7');
    if (scan_flag != 0) {
        clock_out_time ();
        cmdb_prompt ();
    }
}
```

/* 时钟设置函数 */

```
void clock_set ( unsigned long sethsec) {
    dayhsecs = sethsec;
    last_tick = timer0_count ();
    clock_update ();
}
```

/* 时钟扫描函数 */

```
void clock_scan (unsigned char flag) {
    scan_flag = flag;
}
```

/* 时间输出函数 */

```
void clock_out_time (void) {
    xdata char buf [21];
    unsigned hsecs, secs, mins, hours;
    unsigned long t;

    t = dayhsecs;
    hsecs = t % 100;
    t /= 100;
    secs = t % 60;
    t /= 60;
    mins = t % 60;
    t /= 60;
    hours = t % 24;
```

```

buf [0] = (hours / 10) + '0';
buf [1] = (hours % 10) + '0';
buf [2] = ':';
buf [3] = (mins / 10) + '0';
buf [4] = (mins % 10) + '0';
buf [5] = ':';
buf [6] = (secs / 10) + '0';
buf [7] = (secs % 10) + '0';
buf [8] = '.';
buf [9] = (hsecs / 10) + '0';
buf [10] = (hsecs % 10) + '0';
buf [11] = '\0';

#ifdef evalboard    /* 采用硬件目标板时需要使用以下数组 */
dg[7]=buf[7];        /* 准备硬件目标板 LED 显示数据 */
dg[6]=buf[6];
dg[4]=buf[4];
dg[3]=buf[3];
dg[1]=buf[1];
dg[0]=buf[0];
DISPLY(dg);          /* 点亮硬件目标板 LED */
#endif

com_puts ("\r\n");
com_puts (buf);
com_puts ("\r\n");
}

/* 设置屏幕时间按“HHMMSS”格式显示 */
char strtotm ( unsigned long *t, char *s) {
    char *s2;
    unsigned char tmp;
    if (strlen (s) != 6)
        return (-1);

    for (s2 = s; *s2 != '\0'; s2++) {
        if (!isdigit (*s2))
            return (-1);
    }

    tmp = ((s[0] - '0') * 10) + (s[1] - '0');
    if (tmp >= 24)
        return (-1);

```

```
    *t = tmp;
    tmp = ((s[0] - '0') * 10) + (s[1] - '0');
    if (tmp >= 24)
        return (-1);
    *t = tmp;
    tmp = ((s[2] - '0') * 10) + (s[3] - '0');
    if (tmp >= 60)
        return (-1);
    *t = (60 * *t) + tmp;
    tmp = ((s[4] - '0') * 10) + (s[5] - '0');
    if (tmp >= 60)
        return (-1);
    *t = (60 * *t) + tmp;
    return (0);
}
```

/* 闹钟设置函数 */

```
void alarm_set (unsigned setmins) {
    almmmins = setmins;
    alm_flag = 1;
}
```

/* 闹钟清零函数 */

```
void alarm_clr (void) {
    alm_flag = 0;
}
```

/* 闹钟时间输出函数 */

```
void alarm_out_time (void) {
    xdata char buf [21];
    unsigned mins;
    unsigned hours;
    unsigned t;

    if (alm_flag == 0) {
        com_puts ("\r\nNone\r\n");
        return;
    }
    t = almmmins;
    mins = t % 60;
    t /= 60;
    hours = t % 24;
```

```
    buf [0] = (hours / 10) + '0';
    buf [1] = (hours % 10) + '0';
    buf [2] = ':';
    buf [3] = (mins / 10) + '0';
    buf [4] = (mins % 10) + '0';
    buf [5] = '\0';

    com_puts ("\r\n");
    com_puts (buf);
    com_puts ("\r\n");
}
```

命令创建模块文件 CMDBUILD.C 代码:

```
#include <stdlib.h>
#include <ctype.h>
#include "tdp.h"

xdata char cmdbuf [1 + MAX_CMD_LEN];
xdata unsigned char cmdndx;

/* 初始化 */
void cmdb_init (void) {
    cmdndx = 0;
    cmdbuf [0] = '\0';
}

/* 命令提示 "COMMAND" */
void cmdb_prompt (void) {
    com_puts ("COMMAND: ");
    com_puts (cmdbuf);
}

/* 输入命令扫描 */
const char *cmdb_scan (void) {
    int c;
    while (1) {
        if ((c = com_getchar ()) == -1)
            break;
        if (c == '\r') {
            com_puts ("\r\n");
            return (cmdbuf);
        }
        if ((c == '\b') && (cmdndx != 0)) {
```



```

        com_puts ("\b \b");
        cmdbuf [--cmdndx] = '\0';
        continue;
    }
    if (!isprint (c)) {
        BEEPCHAR: com_putchar ('\x7');
        continue;
    }
    if (cmdndx >= MAX_CMD_LEN)
        goto BEEPCHAR;
    com_putchar (c);
    cmdbuf [cmdndx++] = (unsigned char) c;
    cmdbuf [cmdndx] = '\0';
}
return (NULL);
}

```

命令处理模块文件 CMDPROC.C 代码:

```

#include <ctype.h>
#include <string.h>
#include "tdp.h"

/* 定义帮助提示 */
static pdata char helptext [] =
    "\r\n"
    "HELP:\r\n"
    "STCLK hhmss -- Set Clock Time\r\n"
    "RDCLK      -- Display Clock Time\r\n"
    "SCCLK ON|OFF -- Display Clock Time Every Second\r\n"
    "STALM hhmm  -- Set Alarm\r\n"
    "RDALM      -- Display Alarm Time\r\n"
    "CLALM      -- Clear Alarm Time\r\n";
enum {
    CID_SET_CLK,
    CID_READ_CLK,
    CID_SCAN_CLK,
    CID_SET_ALM,
    CID_READ_ALM,
    CID_CLR_ALM,
    CID_LAST
};

struct cmd_st {

```

```

    const char *cmdstr;
    unsigned char id;
};

static pdata struct cmd_st cmd_tbl [] = {
    { "STCLK",      CID_SET_CLK },
    { "RDCLK",      CID_READ_CLK },
    { "SCCLK",      CID_SCAN_CLK },
    { "STALM",      CID_SET_ALM },
    { "RDALM",      CID_READ_ALM },
    { "CLALM",      CID_CLR_ALM },
};

#define CMD_TBL_LEN (sizeof (cmd_tbl) / sizeof (cmd_tbl [0]))

static unsigned char cmdid_search (char *cmdstr) {
    struct cmd_st *ctp;
    for (ctp = cmd_tbl; ctp < &cmd_tbl [CMD_TBL_LEN]; ctp++) {
        if (strcmp (ctp->cmdstr, cmdstr) == 0)
            return (ctp->id);
    }
    return (CID_LAST);
}

char *strupr ( char *src ) {
    char *s;
    for (s = src; *s != '\0'; s++)
        *s = toupper (*s);
    return (src);
}

/* 命令处理 */
void cmd_proc (const char *cmd) {
    xdata char cmdstr_buf [1 + MAX_CMD_LEN];
    xdata char argstr_buf [1 + MAX_CMD_LEN];
    char *argsep;
    unsigned char id;

    strncpy (cmdstr_buf, cmd, sizeof (cmdstr_buf) - 1);
    cmdstr_buf [sizeof (cmdstr_buf) - 1] = '\0';
    strupr (cmdstr_buf);
    argsep = strchr (cmdstr_buf, ' ');
    if (argsep == NULL) {

```

```
    argstr_buf[0] = '\0';
}
else {
    strcpy (argstr_buf, argsep + 1);
    *argsep = '\0';
}
id = cmdid_search (cmdstr_buf);
switch (id) {
    unsigned long tm;
    case CID_SET_CLK:
        if (strtotm (&tm, argstr_buf) != 0)
            goto CMDERR;
        clock_set (tm * 100);
        break;
    case CID_READ_CLK:
        clock_out_time ();
        break;
    case CID_SCAN_CLK:
        if (strcmp (argstr_buf, "ON") == 0)
            clock_scan (1);
        else if (strcmp (argstr_buf, "OFF") == 0)
            clock_scan (0);
        else
            goto CMDERR;
        break;
    case CID_SET_ALM:
        strcat (argstr_buf, "00");
        if (strtotm (&tm, argstr_buf) != 0)
            goto CMDERR;
        alarm_set (tm / 60);
        break;
    case CID_READ_ALM:
        alarm_out_time ();
        break;
    case CID_CLR_ALM:
        alarm_clr ();
        break;
    case CID_LAST:
    CMDERR:
        com_puts (helptext);
        break;
}
}
```

串行通信模块文件 SERIAL.C 代码:

```
#include "tdp.h"
#include <reg51.h>
#include <string.h>
#define TBUF_SIZE 256      /* 这两行不要改动 */
#define RBUF_SIZE 256

static xdata unsigned char tbuf [TBUF_SIZE];
static xdata unsigned char rbuf [RBUF_SIZE];
static xdata unsigned char t_in = 0;
static xdata unsigned char t_out = 0;
static xdata unsigned char t_disabled = 0;
static xdata unsigned char r_in = 0;
static xdata unsigned char r_out = 0;

/* 串行口中断服务函数 */
static void com_isr (void) interrupt 4 using 2 {
    if (RI != 0) {          /* 接收处理 */
        RI = 0;
        if ((r_in + 1) != r_out)
            rbuf [r_in++] = SBUF;
    }

    if (TI != 0) {          /* 发送处理 */
        TI = 0;
        if (t_in != t_out)
            SBUF = tbuf [t_out++];
        else
            t_disabled = 1;
    }
}

/* 串行口初始化函数 */
void com_initialize (void) {
    com_baudrate (57600);    /* 设定波特率 */
    INT_DISABLE;
    t_in = 0;                /* 清零串行口缓冲器 */
    t_out = 0;
    t_disabled = 1;
    r_in = 0;
    r_out = 0;
    SM0 = 0; SM1 = 1;        /* 设定串行口工作于方式 1 */
}
```

```

    SM2 = 0;
    REN = 1;                /* 允许接收 */
    TI = 0;                 /* 清零串行口中断标志位 */
    RI = 0;
    ES = 1;                /* 允许串行口中断 */
    PS = 0;                /* 设置串行口中断优先级 */
    INT_ENABLE;
}

/* 波特率设置函数 */
void com_baudrate (unsigned baudrate) {
    INT_DISABLE;
    TI = 0;                /* 清零发送中断标志 */
    t_in = 0;              /* 清零发送缓冲区 */
    t_out = 0;
    t_disabled = 1;        /* 禁止发送 */
    TR1 = 0;               /* 停止 T1 */
    ET1 = 0;               /* 禁止 T1 中断 */
    PCON |= 0x80;          /* 设置 SMOD=1: 波特率加倍 */
    TMOD &= ~0xF0;         /* 设置 T1 工作于方式 2 */
    TMOD |= 0x20;
    TH1 = (unsigned char) (256 - (TCLK / (16L * 12L * baudrate)));
    TR1 = 1;               /* start timer 1 */
    INT_ENABLE;
}

/* 字符发送函数 */
char com_putchar (unsigned char c) {
    if ((TBUF_SIZE - com_tbuflen ()) <= 2) /* 若缓冲区满, 则返回出错标志 */
        return (-1);
    INT_DISABLE;
    tbuf [t_in++] = c;      /* 数据送往发送缓冲区 */
    if (t_disabled) {
        t_disabled = 0;
        TI = 1;
    }
    INT_ENABLE;
    return (0);
}

/* 字符串发送函数 */
char com_puts (char *s) {
    if ((TBUF_SIZE - com_tbuflen ()) <= strlen (s))

```

```

        return (-1);
    INT_DISABLE;
    for (; *s != '\0'; s++)
        tbuf [t_in++] = *s;
    if (t_disabled) {
        t_disabled = 0;
        TI = 1;
    }
    INT_ENABLE;
    return (0);
}

```

/* 字符接收函数 */

```

int com_getchar (void) {
    xdata int c;
    if (com_rbuflen () == 0)
        return (-1);
    INT_DISABLE;
    c = rbuf [r_out++];
    INT_ENABLE;
    return (c);
}

```

```

unsigned char com_rbuflen (void) { /* 计算接收缓冲区长度 */
    return (r_in - r_out);
}

```

```

unsigned char com_tbuflen (void) { /* 计算发送缓冲区长度 */
    return (t_in - t_out);
}

```

硬件目标板初始化模块文件 DINPUT.A51 代码:

```

NAME DINPUT
?PR?_DINPUT?DINPUT SEGMENT CODE
PUBLIC _DINPUT
RSEG ?PR?_DINPUT?DINPUT
_DINPUT:
    MOV A,R7
    MOV R2,#08
LOOP1:  RLC A
    MOV P1.0, C
    CLR P1.2
    SETB P1.2

```

```

        DJNZ R2,LOOP1
        MOV A,R5
        MOV R2,#08
LOOP2:   RLC A
        MOV P1.0, C
        CLR P1.2
        SETB P1.2
        DJNZ R2,LOOP2
        CLR P1.1
        SETB P1.1
        RET

```

END

硬件目标板显示处理模块文件 DISPLY.A51 代码:

```

NAME DISPLY
?PR?_disply?DISPLY SEGMENT CODE
PUBLIC _DISPLY
EXTRN CODE (_DINPUT)
RSEG ?PR?_disply?DISPLY
_DISPLY:
        MOV A,R7      ;R7 IS THE dsp_buf ENTRY VAL FOR DSP
        MOV R0,A
        MOV R1,#01
        MOV R3,#08
LOOP3:   MOV A,@R0
        MOV R5,A
        MOV A,R1
        MOV R7,A
        LCALL _DINPUT
        INC R0
        INC R1
        DJNZ R3,LOOP3
        RET
END

```

用户函数头文件 TDP.H 代码如下:

```

/* 宏定义 */
#define INT_DISABLE EA = 0
#define INT_ENABLE  EA = 1
#define TCLK      11059200    /* Clock speed in Hz */

/* CLOCK.C 文件中的函数定义 */
void clock_init (void);

```

```
void clock_update (void);
void clock_set (unsigned long sethsec);
void clock_scan (unsigned char flag);
void clock_out_time (void);
char strtotm (unsigned long *t,char *s);
void alarm_set (unsigned setmins);
void alarm_clr (void);
void alarm_out_time (void);
```

```
/* CMDBUILD.C 文件中的函数定义 */
```

```
#define MAX_CMD_LEN      40
void cmdb_init (void);
void cmdb_prompt (void);
const char *cmdb_scan (void);
```

```
/* CMDPROC.C 文件中的函数定义 */
```

```
char *strupr (char *src);
void cmd_proc (const char *cmd);
```

```
/* MAIN.C 文件中的函数定义 */
```

```
void main (void);
```

```
/* SERIAL.C 文件中的函数定义 */
```

```
void com_initialize (void);
void com_baudrate (unsigned baudrate);
char com_putchar (unsigned char c);
char com_puts (char *s);
int com_getchar (void);
unsigned char com_rbuflen (void);
unsigned char com_tbuflen (void);
```

```
/* TIMER.C 文件中的函数定义 */
```

```
#define TIMER0_TICKS_PER_SEC    100
void timer0_initialize (void);
unsigned timer0_count (void);
unsigned timer0_elapsed_count (unsigned count);
void timer0_wait (unsigned count);
```


进入 μ Vision2 环境后单击“File 菜单 / New...选项”，在打开的编辑窗口中键入上述源程序文件并存盘。

下面要在保存源程序文件的目录中创建一个项目文件，单击“Project 菜单 / New Project...”选项弹出如图 8.1 所示窗口，在文件名栏键入项目名“tdp.uv2”，单击“保存”按钮后弹出如图 8.2 所示 CPU 器件选择窗口，选定 Atmel 公司的 AT89C52 单片机，这样就创建了项目文件 tdp.uv2。



图 8.1 项目创建窗口



图 8.2 CPU 器件选择窗口

μ Vision2 会自动为新创建的项目文件生成一个默认的目标组“Target1”和文件组“Source Group 1”，为清晰起见我们可以给项目文件添加新的目标或文件组。将鼠标指向 μ Vision2 的项目窗口中的目标组“Target1”并单击右键，弹出如图 8.3 所示窗口，单击右键窗口中的“Targets、Groups、Files...”选项，弹出如图 8.4 所示窗口，在“Group to Add”栏分别键入希望添加的文件组名“System Files”、“IO Files”并单击“Add”按钮，添加一个系统文件组和一个输入输出文件组。将鼠标指向 μ Vision2 的项目窗口中的文件组“Source Group 1”并单击右键，从弹出的右键窗口单击“add Files to Group...”选项，弹出如图 8.5 所示窗口，从窗口中选定刚才保存的源程序文件名并单击“Add”按钮，将它们添加到文件组中去。文件添加完成之后的 μ Vision2 项目窗口如图 8.6 所示。

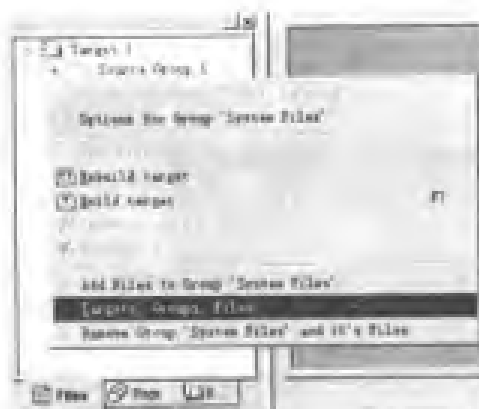


图 8.3 项目窗口中的右键窗口

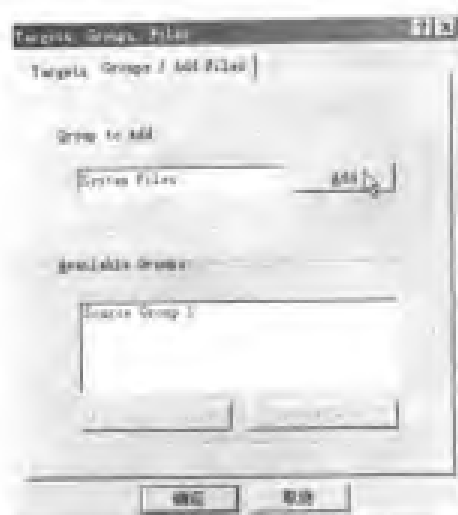


图 8.4 添加文件组窗口



图 8.5 添加文件窗口



图 8.6 添加文件之后的项目窗口

接下来需要对项目的编译连接控制进行设置。单击“Project 菜单 / Options for Target / ”选项，弹出包含多个标签页的“Options”窗口，其中“Target”标签页如图 8.7 所示，选中该标签页中的“Use On chip ROM”复选框，表示使用 AT89C52 片内 ROM，并在“Off-chip Xdata memory”栏的“Start:”和“size:”中分别键入外部数据存储器的起始地址 0x8000 和存储器大小 0x1000。



图 8.7 设置 Options 选项中的 Target 标签页

“Options 选项 / Output” 标签页如图 8.8 所示, 选中其中 “Create Executable...” 圆形单选框以及它下面的三个方形复选框 “Debug Information”、“Browse Information” 和 “Create Hex File”, 这样编译连接完成之后将生成可执行绝对目标代码文件和可用于 EPROM 编程的 Hex 代码文件, 并且在目标代码文件中包含了符号调试信息以及浏览信息, 以利于进行符号化源程序调试。

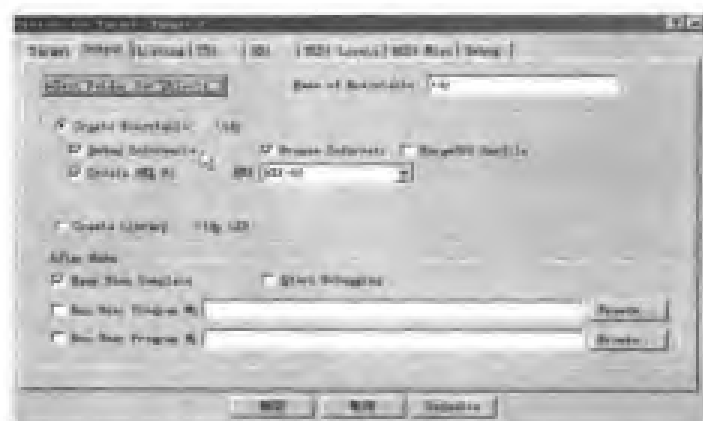


图 8.8 设置 Options 选项中的 Output 标签页

“Options 选项 / Cx51” 标签页如图 8.9 所示, 如果使用硬件目标板, 需要在其中 “Define” 栏键入 “evalboard”, 让源程序文件中的预处理器命令 “#ifdef evalboard” 发挥作用。“Code Optimization” 栏用于对 Cx51 源程序进行优化处理的优化级别与优化重点, 特别需要注意的是优化级别并非越高越好, 要根据实际情况以及自己的编程设想取较为合适的优化级别; 优化重点 (Emphasis) 通常取速度优化 (Favor speed), 以缩短目标代码的执行时间。

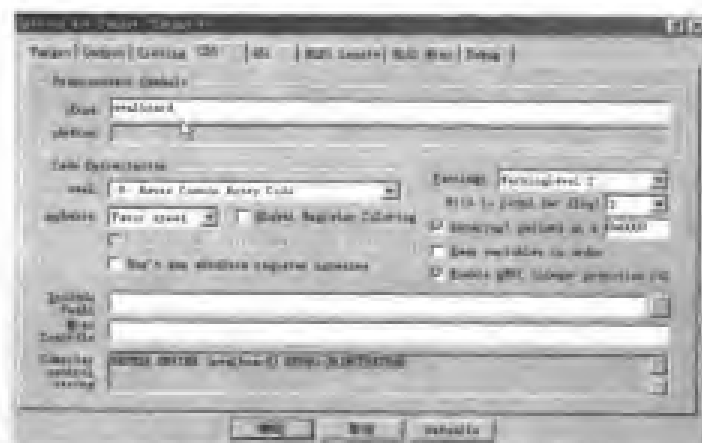


图 8.9 设置 Options 选项中的 C51 标签页

“Options 选项 / BL51” 标签页如图 8.10 所示, 选中该标签页中 “Use Memory Layout from Target Dialog” 复选框, 表示采用前面 “Target” 标签页中的存储器设置, 另外由于 Cx51 源程序中有些变量被安排在外部分页数据存储空间 (PDATA), 因此要在 “Pdata” 栏键入分页地址 0x9000, 并且为了实现对 PDATA 空间的正确定位和显示, 应对启动代码文件 STARTUP.A51 中的参数作如下修改:

PDATALEN	EQU	0FFH	: 定义 PDATA 页空间长度
PPAGEENABLE	EQU	1	: 允许使用 PDATA 存储器
PPAGE	EQU	90H	: 定义 PDATA 存储器的起始页地址

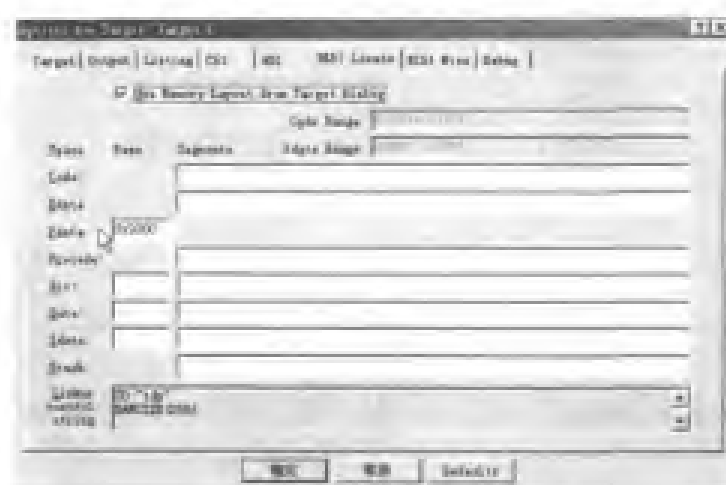


图 8.10 设置 Options 选项中的 BL51 标签页

“Options 选项 / Debug” 标签页如图 8.11 所示, 采用软件模拟仿真时要选中标签页内的 “Use Simulator” 单选框, 选中 “Load Application at Start” 和 “Go till main()” 复选框, 可以在启动软件模拟仿真时自动装入应用程序目标代码并运行到 main() 函数处。在 “Restore Debug Session Settings” 栏中有 4 个方形复选框: “Breakpoints”、“Watchpoints & PA”、“Memory Display” 和 “Toolbox”, 分别用于在启动 Debug 调试器时自动恢复上次调试过程中所设置的断点、观察点与性能分析器、存储器及工具箱的显示状态, 如果在编辑源程序文件时就设置了断点并希望在启动 Debug 仿真调试时能够使用, 则应该选中这些复选框。

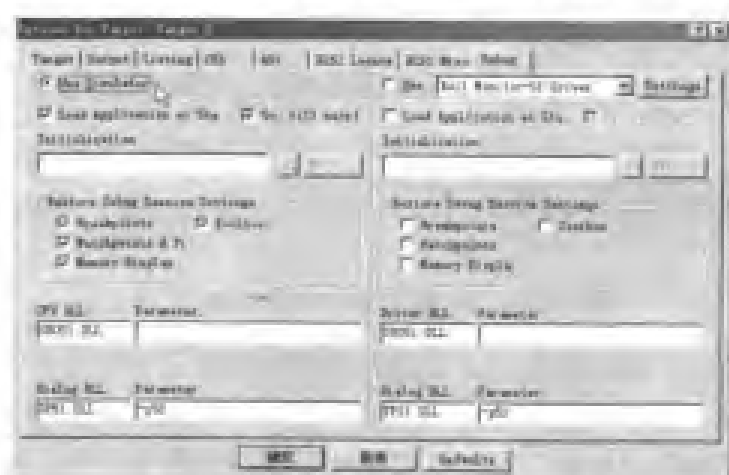


图 8.11 设置 Options 选项中的 Debug 标签页

设置完以上选项后, 单击 “Project 菜单 / Build Target” 选项, 启动对当前项目进行编译链接, 如果没有错误, 命令窗口将出现如图 8.12 所示提示信息, 并生成一个与项目文件同名的 OMF51 格式绝对目标文件, 它可以被装入到 μ Vision2 调试器中进行仿真调试, 同时还将生成一个项目文件同名的 Hex 文件, 它可用于进行 EPROM 编程。

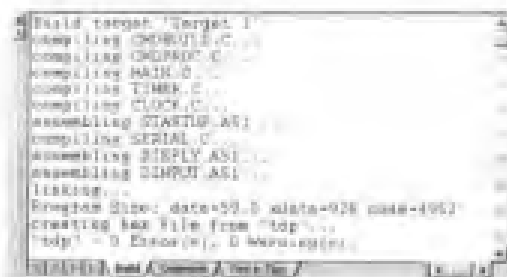


图 8.12 编译连接正确时命令窗口的提示信息

下面可以单击“Debug 菜单 / Start / Stop Debug Session”选项，启动μVision2 调试器，根据前面对“Options 菜单 / Debug 标签页的设置”，启动调试器的同时自动装入目标代码并运行到 main()函数处。调试状态下主窗口以默认的高级语言方式显示被调试的用户源程序，项目窗口自动切换到“Regs”标签页，命令窗口自动切换到“Command”标签页，如图 8.13 所示。



图 8.13 启动调试器后的μVision2 窗口

通过“Debug 菜单”或工具条按钮可以很方便地对源程序进行单步运行、全速运行、设置断点等仿真调试，同时可通过命令窗口输入各种μVision2 调试命令（如调入信号函数等）进行辅助仿真，通过“Regs”标签页可以观察调试过程中 CPU 内部寄存器状态的变化情况。如果希望在调试过程中察看源程序的汇编代码，可以单击“View 菜单 / Disassembly Window”打开反汇编窗口，在该窗口中还可以利用右键菜单进行混合模式（Mixed Mode）与汇编模式（Assembly Mode）切换、在线汇编（Inline Assembly）、查看跟踪记录（View Trace Recorder）、插入 / 删除断点等操作，如图 8.14 所示。

在调试状态下从观察窗口可以观察局部变量、用户设置的观察点以及调用栈的状态。用户程序中的局部变量会自动出现在观察窗口的 Locals 标签页中，如图 8.15 所示。

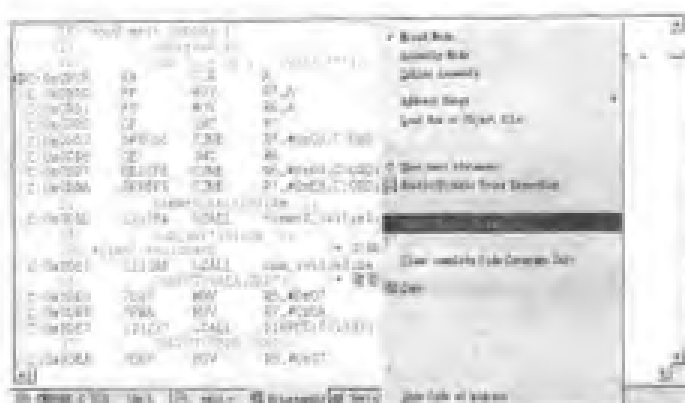


图 8.14 反汇编窗口及其右键菜单

用户可以根据需要在命令窗口中使用“WS”命令来设置观察点。所设置的观察点将出现在观察 1 或观察 2 标签页内。需要注意的是观察点必须位于当前程序模块之中。图 8.16 所示为对 CLOCK_C 模块中 4 个变量设置观察点的情况。

调用栈标签页如图 8.17 所示, 其中显示的是当前函数的嵌套调用情况, 用鼠标左键双击其中某一项, 将使主窗口中光标自动跳到程序对该函数的调用语句行上。

利用µVision2 提供的性能分析器 (Performance Analyzer) 可以对程序各部分运行情况进行分析。单击“Debug 菜单 / Performance Analyzer...”选项弹出如图 8.18 所示窗口。

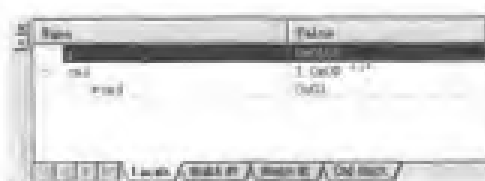


图 8.15 局部变量观察窗口



图 8.16 观察占位口 1

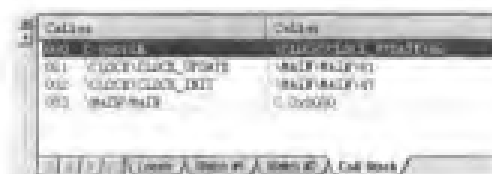


图 8-17 调用栈窗口

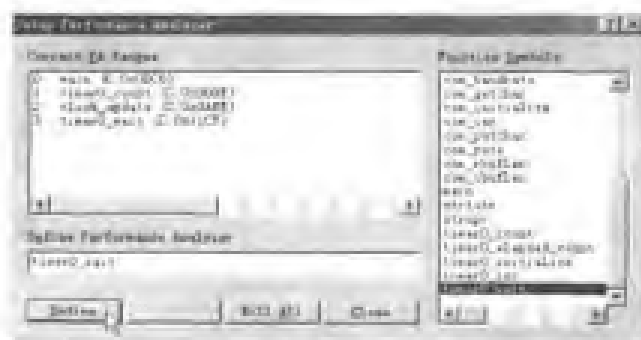


图 8.18 性能分析器设置窗口

在性能分析窗口的 Define Performance Analyzer 栏键入希望进行分析的函数名（也可以从窗口右边 Function Symbols 栏内选取需要分析的函数名并双击鼠标左键），单击“Define”按钮即可设置一个分析范围，本例设置了 4 个分析范围。

别点击这三个复选框可以控制定时器 T0 的运行状态。将其中复选框 TR0 清除,可停止内部定时器 T0,此时将不会产生定时器溢出中断,中断服务程序也不会被触发。

单击“Peripherals 菜单 / Interrupt”选项,弹出如图 8.23 所示窗口,可以检查 8051 单片机各个中断状态,选中 Timer0 并用鼠标左键点击其中的复选框 ET0,可禁止或允许 T0 中断的产生,同时可以看到定时器 T0 的运行状态会随之发生变化。



图 8.22 片内集成功能——Timer0 窗口

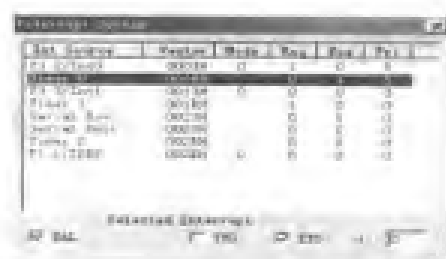


图 8.23 片内集成功能——中断窗口

下面给出一个利用 8051 片内定时器 T0 来测量外部脉冲宽度的例子。将定时器 T0 设置为 16 位定时方式,并将特殊功能寄存器 TMOD 中的 GATE 位置 1,外部脉冲接到 INTO (P3.2) 引脚,这样外部脉冲的上升沿将启动定时器 T0,外部脉冲的下降沿停止定时器 T0,从而可以测量出加到 INTO 引脚上的正脉冲宽度(对于负脉冲之要用一个非门反向后再连到 INTO 引脚)。

主程序 main.c 文件代码如下:

```
#include <reg52.h>
#include <stdio.h>
#define ulong unsigned long

unsigned int T0_ISR_count = 0; /* 定义 T0 中断次数 */

/***** 定时器 T0 中断服务函数 *****/
void T0_ISR (void) interrupt 1 {
    T0_ISR_count++; /* 每次 T0 中断时,中断次数加 1 */
    TF0 = 0;
}

/***** 主函数 *****/
void main (void) {
    SCON = 0x50; /* 串行口初始化 */
    TMOD |= 0x20; /* 利用定时器 T1 作为波特率发生器 */
    TH1 = 0xFA; /* 晶振为 11.0592MHz 时波特率为 9600 */
    TR1 = 1;
    TI = 1;
    PCON |= 0x80;
    printf ("%nPulse Width Example Program\n\n"); /* 输出标题信息 */
}
```



```

ET0 = 1;
EA = 1;                      /* 开中断 */
TMOD = (TMOD & 0xF0) | 0x09; /* 设置 T0 为 16 位定时器方式 */
while (1) {
    T0_ISR_count = 0; TH0 = 0; TL0 = 0;
    TR0 = 1;
    printf ("\nStart a pulse.\n"); /* 输出提示信息 */
    while (!INT0);                 /* 等待脉冲上升沿, 开始测量 */
    while (INT0);                 /* 等待脉冲下降沿, 停止测量 */
    /* 以每个 T0 计数值为 1 $\mu$ s 计算脉冲宽度并输出 */
    printf ("The width pulse is: %ld uSec\n",
            (ulong)((TH0 << 8) | TL0 | ((ulong)T0_ISR_count << 16)));
}
}

```

为了程序模拟调试方便, 编制了一个调试信号函数文件 debug.ini, 其中定义了 5 个按钮, 分别用于在 INT0 引脚上产生 10 μ s~5.011966s 的脉冲信号, debug.ini 文件代码如下:

```

/* 禁止 T0 的外部 GATE 信号 */
PORT3 &= ~0x04;

/* 定义产生 10, 100, 1000, 10000 $\mu$ s 和 5.011966s 脉冲按钮 */
define button "10 uSec Pulse" , "t0_gate (0.000010)"
define button "100 uSec Pulse" , "t0_gate (0.00010)"
define button "1 mSec Pulse" , "t0_gate (0.0010)"
define button "10 mSec Pulse" , "t0_gate (0.010)"
define button "5.011966 Sec Pulse" , "t0_gate (5.0119660)"

/* 定义信号函数, 利用 swatch() 内部函数在 INT0 引脚上产生不同宽度的正脉冲 */
signal void t0_gate (float secs) {
    PORT3 |= 0x04;
    swatch (secs);
    PORT3 &= ~0x04;
}

```

创建一个 μ Vision2 项目并添加上述程序文件, 如图 8.24 设置“Options 选项 / Debug”标签页, 在“Initialization”栏键入初始化文件名 debug.ini, 这样在启动 μ Vision2 调试器时将自动装入初始化文件并运行其中的命令。

进入 μ Vision2 调试器后, 单击“Peripherals 菜单 / IO-Ports / Port3”选项弹出如图 8.25 所示 Port3 窗口, 在全速运行状态下通过置 1 或清零 P3.2 可以模拟加在 INT0 引脚上的脉冲宽度, 也可以单击“View 菜单 / Toolbox”选项弹出如图 8.26 所示工具箱按钮, 单击不同按钮调用 μ Vision2 信号函数来模拟外加脉冲宽度。单击“Peripherals 菜单 / Timer / Timer0”选项, 弹出如前面图 8.22 一样 Timer0 窗口, 从中可以观察 T0 运行状态, 同时从

μ Vision2 串行窗口中可以看到如图 8.27 所示模拟输出结果。

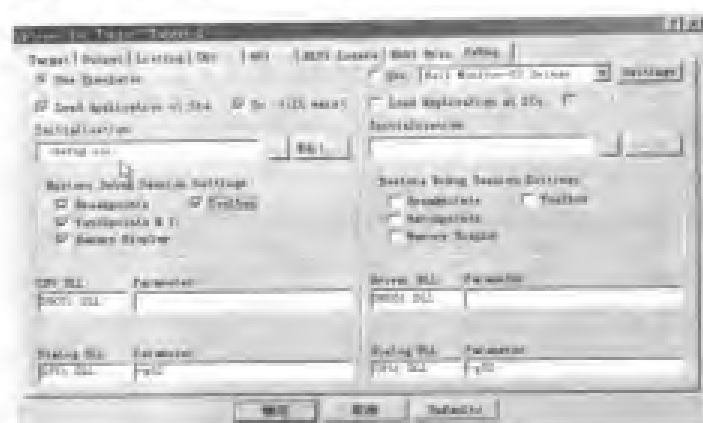


图 8.24 在 Debug 标签页设置初始化文件



图 8.25 继承外围功能 Port 3 窗口



图 8.26 工具箱按钮

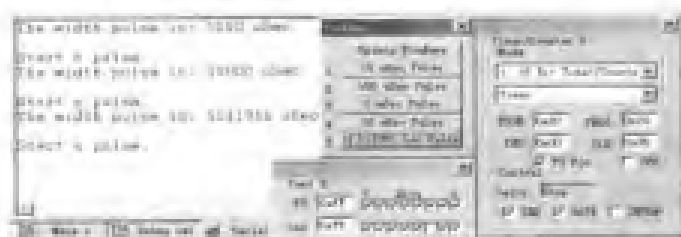


图 8.27 从串行窗口观察输出结果

8.3.3 在 μ Vision2 中生成应用库函数

将开发成功的单片机应用系统提供给客户时,为了保护开发人员的劳动成果,可能仅向客户提供执行代码而不提供源程序,或只提供部分源程序而将关键部分以库函数形式提供。本节介绍在 μ Vision2 集成环境中生成应用库函数的方法。

如果希望将 8.3.1 节实时时钟例子中的输入输出模块 SREIAL.C、DINPUT.A51、DISPLY.A51 生成应用库函数,可以单独创建一个项目“evalboard.uv2”并将这三个模块文件添加到项目中去,然后如图 8.28 所示设置该项目的“Option 选项 / Output 标签页”,选中标签页内的“Create Library”圆形单选框,对该项目进行编译连接即可生成一个与项目名称同名的应用库函数文件“evalboard.LIB”。

接着对原 8.3.1 节的项目文件稍作修改,删除原项目中“IO Files”文件组及其模块文件 SREIAL.C、DINPUT.A51、DISPLY.A51,另建一个文件组“Lib Files”并将应用库文件“evalboard.LIB”添加到该文件组中去,如图 8.29 所示,其他设置选项不变,然后再对修

改后的项目重新进行编译链接,即可生成与原来目标代码相同的可执行代码文件,但现在向客户提供的执行代码中可以不包含源程序模块文件 SREIAL.C、DINPUT.A51、DISPLY.A51,而只要包含库文件 evalboard.LIB 就可以了。

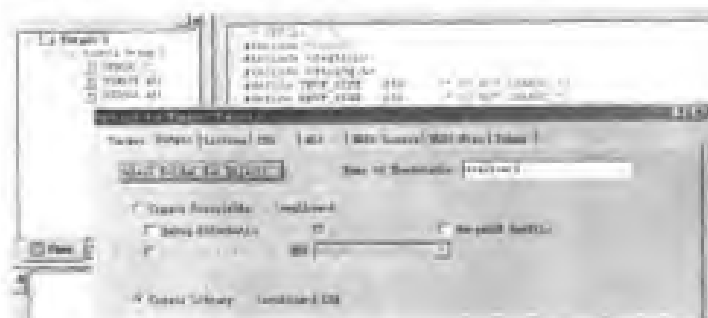


图 8.28 设置生成应用库函数选项



图 8.29 修改之后的项目窗口

8.4 在 μ Vision2 中应用硬件目标板

μ Vision2 可以与安装了 Keil 公司提供的可配置监控程序 MONITOR-51 的用户硬件目标板实现无缝连接,实现硬件目标系统的在线仿真调试。监控程序 MONITOR-51 可依据用户硬件系统灵活配置,使用户避免购买昂贵的硬件仿真器,降低应用系统的开发成本。

正确运行 MONITOR-51 所需要的最小硬件配置如下。

- 8051CPU。
- 从 0x0000 地址开始的 5K 字节的 EPROM,并固化有 MONITOR-51 监控程序。
- 最少 256 字节冯·诺依曼 (Von Neumann) 形式连接的外部 RAM,附加 5K 字节 RAM 用于跟踪缓冲区。
- 6 字节内部 RAM 用于堆栈。
- 一个内部定时器用于波特率发生器。
- 串行口用于通信。

MONITOR-51 的基本工作原理是,通过固化在 EPROM 内的监控程序将用户应用程序装入到以冯·诺依曼形式连接的 RAM 中,并监控用户应用程序的运行。所谓冯·诺依曼形式连接,就是将 8051 的 PSEN 和 RD 引脚相“与”之后作为外部 RAM 的 OE 选通信号。安装了 Keil Cx51 软件包之后,在“KEIL\C51\MON51”目录中提供了 MONITOR-51 的配置文件“INSTALL.A51”,用户可以根据自己目标硬件修改该配置文件,然后运行批处理文件“INSTALL.BAT”即可生成 MONITOR-51 监控程序的 HEX 代码,将 HEX 代码固化到 EPROM 中就可以运行了。

直接采用 MONITOR-51 时要求将监控程序安排在从 0x0000 地址开始的 5K 字节 EPROM 中,而 8051 单片机的复位入口也是 0x0000,因此用户程序通常配置为从 0x8000 开始,这样给用户进行实际应用程序设计造成不便。

为了帮助读者更好地学习 Keil Cx51 应用编程,本书作者设计了一种通用的 Keil Cx51 源程序仿真硬件目标板,将 MONITOR-51 监控程序安排在高端地址 0xe000,单片机的复

位入口地址 0x0000 留给用户, 并可与 μ Vision2 环境无缝连接, 用户可以直接利用该硬件仿真目标板开发用户自己的应用系统, 不会有任何不便之处。

硬件目标板上具有如下资源。

- 8KB~32KB 仿真 RAM/用户 ROM, 配有 32KB RAM 芯片 62256, 仿真调试时用户程序被下载到该芯片中, 调试完成后可将其换为 27256 芯片用于固化调试通过的用户程序。
- 8KB/24KB 用户 RAM, 配有 8KB RAM 芯片 6264, 作为用户的数据存储器。
- MAX7219 共阴极 8 段 LED 驱动器, 可直接驱动 8 个共阴极数码显示器。
- 8031 单片机 (基本配置), 可选配其他类型单片机如 Winbond77E58、DS80C320、P98C51RDx (ISP)、SST89C58 (ISP) 等新型单片机。
- 一个 40 脚插座将单片机全部引脚都引出, 可作为仿真插头接口。
- 一个 20 脚插座可作为 8279 键盘显示板接口。
- 置了若干译码输出端口 (地址线) 以及数据线以便于用户使用, 板上留有充分的用户扩展区, 方便用户进行各种接口扩展。

Keil Cx51 源程序硬件仿真目标板上存储器地址空间分配如表 8-1 所示。

表 8-1 Keil Cx51 源程序硬件仿真目标板上存储器地址空间分配

存储器空间	地址范围
仿真 RAM/用户 ROM 区	0x0000~0x7EFF
用户数据/IO 扩展区	0x8000~0x9EFF, 0xA000~0xBFFF
用户扩展区	0xC000~0xDFFF
监控 RAM 区	0x9F00~0x9FFF
监控 ROM 区	0xE000~0xFFFF

硬件目标板通过板上各个插座跳线来实现不同配置, U2 插座可插入 32KB 仿真 RAM (62256) 芯片, 用于从 μ Vision2 环境下载用户程序进行调试, 也可插入 ROM (27256) 芯片, 用于全速运行调试通过的用户程序。该插座地址范围为 0x0000~0x7FFF, 配有 4 个跳线, 其作用如表 8-2 所示。

表 8-2 U2 插座跳线的作用

跳线名	跳线位置	功能作用	连接方式	说 明	默认位置
JP1	RSE	仿真		仿真调试	RSE
	A15	用户 ROM		运行用户程序	
JP2	PSEN	用户 ROM	哈弗	数据与程序分开	P/S
	P/S	仿真	冯·诺依曼	数据与程序共享	
JP3	27256	用户 ROM		运行用户程序	62256
	62256	仿真		仿真调试	
JP4	27256	用户 ROM		运行用户程序	62256
	62256	仿真		仿真调试	

U4 插座可插入 RAM (6264) 或 (62256) 芯片, 用于存贮用户数据, 该插座地址范围为 0x8000~0xDFFF, 配有 3 个跳线, 其作用如表 8-3 所示。

表 8-3 U4 插座跳线的作用

跳线名	跳线位置	连接方式	说 明	默认位置
JP5	RD	哈弗	数据与程序分开	P/S
	P/S	冯·诺依曼	数据与程序共享	
JP6	64		插入 6264 芯片	64
	56		插入 62256 芯片	
JP7	64		插入 6264 芯片	64
	56		插入 62256 芯片	

GAL16V8 插座配有 1 个跳线, 其作用如表 8-4 所示。

表 8-4 GAL16V8 插座跳线的作用

跳线名	跳线位置	说 明	默认位置
JP9	56K	不用板上扩展地址	48K
	48K	使用板上扩展地址	

U10 插座上插入的是单片机 8031 芯片, 用户可根据自己的需要插入其他类型的兼容 8051 芯片。该插座配有 1 个跳线, 其作用如表 8-5 所示。

表 8-5 U10 插座跳线的作用

跳线名	跳线位置	说 明	默认位置
JP10	VCC	运行片内程序或进行 ISP 操作	GND
	GND	运行片外程序	

U8 插座上插入的是固化 74LS04 芯片, 配有 1 个跳线, 其作用如表 8-6 所示。

表 8-6 U8 插座跳线的作用

跳线名	跳线位置	说 明	默认位置
JP11	EMU	运行监控程序	EMU
	USER	运行用户程序	

板上左边靠近 MAX232 芯片处有一个 P2 跳线, 是为需要使用单片机 DS80C320 的双串行口预留的, 目的是将 DS80C320 的 P1.3 和 P1.4 引脚作为第二串行口 UART1, UART1 插座已经预留在板上, 但需要用户自己焊接。P2 跳线的作用如表 8-7 所示。

表 8-7 P2 跳线的作用

跳线名	连接方式说明	默认位置
P1.3	此两脚短路时可 使用 DS80320 的第二个串行口 UART1	断开
空脚		

续表

跳线名	连接方式说明	默认位置
空脚	此两脚短路时可使用	断开
PL4	DS80C320 的第二个串行口 UART1	

以上跳线的默认位置是为便于与 μ Vision2 仿真环境接口,方便下载调试用户而设置的,用户可根据自己需要改变跳线位置。一般在进行仿真调试时应按默认方式配置,若用户改变跳线位置后出现无法与 μ Vision2 环境联机的现象,请将跳线恢复到默认位置。

在 μ Vision2 环境下时用 Keil Cx51 源程序硬件仿真目标板时,由于仿真环境要求存储器采用“冯·诺依曼(数据程序共享)”方式连接,因此要按如图 8.30 所示设置“Options 选项/Target”标签页中片外 ROM 存储器(Off-chip Code memory)和片外 RAM 存储器(Off-chip Xdata memory)地址范围。

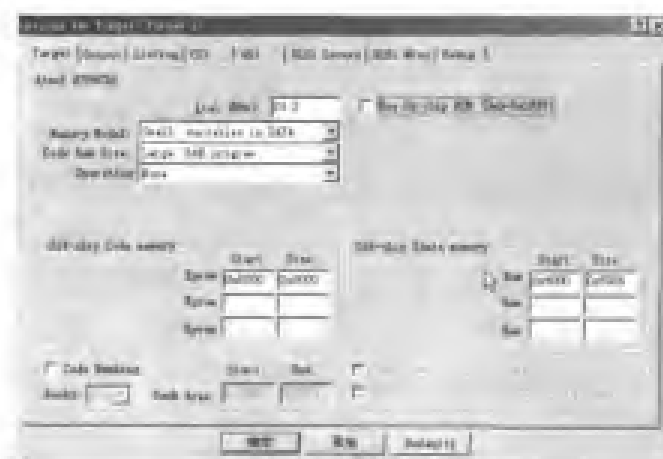


图 8.30 设置硬件目标板地址范围

为了能够在编译完成后直接将文件下载到硬件目标板上进行仿真调试,需要按如图 8.31 所示设置“Options 选项 / Debug”标签页,选中标签页中的“Use Keil MONITOR-51 Driver”圆形单选框。同时选中“Load Application at Start”和“Go till main()”方形复选框,可以在启动仿真时自动装入应用程序目标代码并运行到 main()函数处。

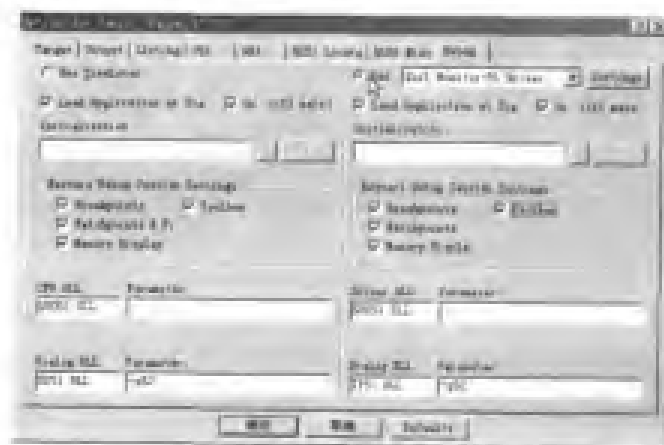


图 8.31 设置硬件目标板时的 Debug 选项

为了实现硬件目标板与 PC 机通信, 还需要设置 PC 机 COM 端口和通信波特率, 单击“Options 选项 / Debug”标签页内的“Settings”按钮, 弹出如图 8.32 窗口, 在“Port”栏选取合适的 COM 端口, 在“Baudrate”栏选取合适的波特率, 硬件目标板可以进行自动波特率调整, 适当降低波特率可提高联机可靠性。选中窗口右边的“Cache Options”方形复选框可以加快调试过程中对变量的显示速度, 但为了能从 μ Vision2 调试状态下直接观察硬件目标板上 CPU 引脚的状态, 则不要选中这些复选框。窗口下边还有一个方形复选框“Stop Program Execution with Serial Interrupt”, 选中时可以在通过“Debug 菜单 / Stop Running”选项停止用户程序的运行, 但此时将占用 8051 串行口中断。



图 8.32 设置通信端口及波特率

以上设置完成后, 接通硬件目标板的电源, 连接好目标板与 PC 机之间的通信线, 单击“Debug 菜单 / Start / Stop Debug Session”选项, 进入 μ Vision2 调试器, 联机成功后将弹出如图 8.33 所示窗口, 在 μ Vision2 的命令窗口中会看到硬件目标板中监控程序的版本号。此时单击“Debug 菜单 / Go”选项, 启动用户程序全速运行, 不仅可以通过 μ Vision2 的串行窗口看到与前面图 8.20 相同的输出结果, 同时还可以从硬件目标板的 LED 显示器上看到真实的时钟运行情况。



图 8.33 硬件目标板仿真调试状态窗口

本书中所有程序例子都可以在 Keil Cx51 源程序仿真硬件目标板上运行, 前面 8.3.1 节介绍的实时时钟程序就是一个很好的实例。读者如果对硬件目标板感兴趣可以直接通过电子信箱: ajxu@tom.com、ajxu41@sohu.com 与本书作者联系。

8.5 8051 单片机的片内串行口应用编程

8.5.1 用 8051 串行口扩展的矩阵键盘

Cx51 编译器的运行库中提供了一套输入输出库函数: getchar()、putchar()、printf()和 scanf(), 这些是所谓的“标准 I/O 函数”, 它们利用 8051 单片机的串行口完成输入输出功能。实际上用户还可以对 8051 单片机串行口重新进行编程以完成自己需要的某种特殊功能。8051 单片机的串行口工作于方式 0 时, 可作为移位寄存器用于扩展 I/O 接口。图 8.34 所示为利用 8051 串行口扩展的一种矩阵键盘接口电路。74LS164 是串入 / 并出移位寄存器, 它将来自 8051 串行口线 P3.0 (RXD) 的串行数据转换成 8 位并行数据, P3.4 和 P3.5 定义为输入口线, 可实现一个 2×8 矩阵键盘接口。

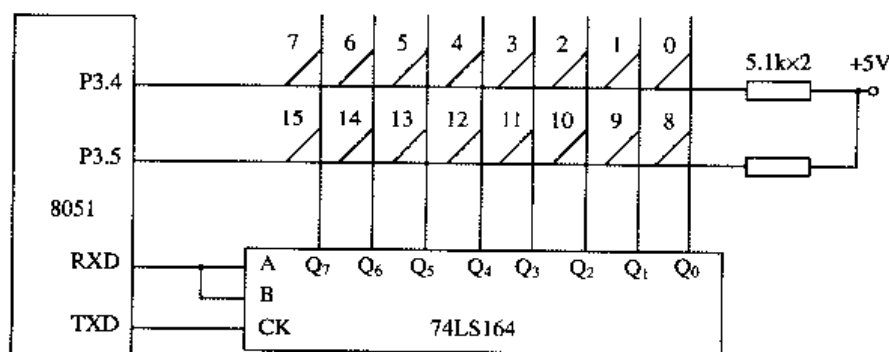


图 8.34 用 8051 串行口扩展的矩阵键盘接口

下面给出针对图 8.34 键盘接口的 Cx51 驱动程序。程序由主函数 main()、读键盘函数 get_char() 和延时函数 delay() 组成。主函数将 8051 串行口初始化为工作方式 0, 采用查询式输入输出, 然后调用读键盘函数读入按键的编码值并存入以 keybuf 为首地址的 16 个内部 RAM 单元中。读键盘函数 get_char() 判断是否有键压下, 有按键时进行键值分析并将按键的键值返回给主调用函数。延时函数 delay() 的功能是提供一段延时时间以防止按键抖动对键值分析的影响。

程序代码如下 (文件名为: sercom.c):

```
#include <reg51.h>
#include <intrins.h>
sbit P34 = 0xB4;
sbit P35 = 0xB5;
unsigned char get_char(void);          /* 函数说明 */
void delay(void);
```



```

/***** 主函数 *****/
main() {
    unsigned char keybuf[16], count;    /* 键盘缓冲区和读键计数变量 */
    SCON=0;                            /* 将串行口设置成工作方式 0 */
    ES=0;                              /* 禁止串口中断 */
    EA=0;
    count=0;
    while(count<16) keybuf[count++]=get_char(); /* 读入 16 个按键的键值 */
}

/***** 读键盘函数 *****/
unsigned char get_char(void) {
    /* 定义表示列号、键序号和待发送数据的变量 column、key_code 和 mask */
    unsigned char key_code, column=0, mask=0x00;
    /* 下列语句从串行口向 74LS164 移位输出 8 个 0 */
    TI=0;
    SBUF=mask;
    while(TI==0);                      /* 等待发送完毕 */
    /* 下列语句通过检测 P3.4 和 P3.5 是否为 0 来判断是否有键压下, 检测到有键压下时延时 10ms
       以消除按键抖动, 然后继续检测 P3.4 和 P3.5 是否为 0, 若不为 0 则表明检测到干扰信号并继续
       等待按键, 则表示有一个键被可靠地按下并退出循环。
    */
    while(1) {
        while((P34&P35)!=0);
        delay();
        if((P34&P35)!=0) continue;
        else break;
    }
    /* 下列语句分析被按下的键所在的列号 */
    mask=0xfe;
    while(1) {
        TI=0;
        SBUF=mask;
        while(TI==0);
        if((P34&P35)!=0) {
            mask=_crol_(mask,1);        /* mask 的值循环左移一位 */
            column++;
            if(column>=8) column=0;
            continue;
        }
        else break;
    }
    /* 下列语句分析被按下的键所在的行号并计算键序号 */
}

```

```

    if(P34==0) key_code=column;
    else      key_code=8+column;
    return(key_code);
}

/***** 10ms 延时函数 *****/
void delay(void) {
    unsigned int i=10;          /* 延时 10ms */
    while(i--);
}

```

8.5.2 利用 8051 串行口实现多机通信

下面给出一个利用 8051 串行口进行多机通信的 Cx51 程序。一个主机与多个从机进行单工通信，主机发送，从机接收。主机先向从机发送一帧地址信息，然后再发送 10 位数据信息。从机接收主机发来的地址，并与本机的地址相比较，若不相同则仍保持 SM2=1 不变，自动抛弃数据帧。若地址相同，则使 SM2=0，准备接收主机发来的数据信息，直至接收完 10 位数据。通信双方均采用 11.0592MHz 的晶振，用定时器/计数器 1 产生 9600 的波特率，采用中断方式传送数据。从机的地址为 0~255 的编码。实际通信中还应考虑通信协议，为简单起见，下面的程序中未予考虑。主机发送程序文件名为 T.C，从机接收程序文件名为 R.C。

发送程序代码如下（文件名为 T.C）：

```

#include <reg51.h>
#define COUNT 10                      /* 定义发送缓冲区大小 */
#define NODE_ADDR 64                  /* 定义目的节点地址 */
unsigned char buffer[COUNT];          /* 定义发送缓冲区 */
int pointer;                          /* 定义当前位置指针 */

/***** 主函数 *****/
main() {
    while(pointer<COUNT) {           /* 发送缓冲区初始化 */
        buffer[pointer]='A'+pointer;
        pointer++;
    }
    SCON=0xc0;                        /* 初始化串行口和波特率发生器 */
    TMOD=0x20;
    TH1=0xfd;
    TR1=1;
    ET1=0;
    ES=1;
    EA=1;
    pointer=-1;
}

```

```

    TB8=1;
    SBUF=NODE_ADDR;                /* 发送地址帧 */
    while(pointer<COUNT);         /* 等待全部数据帧发送完毕 */
    /* ..... */
}

/***** 串行口发送中断服务函数 *****/
void send(void) interrupt 4 using 3 {
    TI=0;                          /* 清发送中断标志 */
    pointer++;                      /* 修改发送缓冲区当前位置指针 */

    if(pointer>=COUNT) return;    /* 如果全部数据发送完毕则返回 */
    else {                          /* 否则发送一帧数据 */
        TB8=0;                     /* 设置数据帧标志 */
        SBUF=buffer[pointer];      /* 启动发送 */
    }
}

```

接收程序代码如下 (文件名为 R.C):

```

#include <reg51.h>
#define COUNT 10                  /* 定义接收缓冲区大小 */
#define NODE_ADDR 64              /* 定义本节点地址 */

unsigned char buffer[COUNT];      /* 定义接收缓冲区和当前位置指针 */
int pointer;

/***** 主函数 *****/
main() {
    SCON=0xf0;                    /* 初始化串行口和波特率发生器 */
    TMOD=0x20;
    TH1=0xfd;
    TR1=1;
    ET1=0;
    ES=1;
    EA=1;

    pointer=0;                    /* 等待接收地址帧和全部数据帧 */
    while(pointer<COUNT);
    /* ..... */
}

/***** 串行口接收中断服务函数 *****/
void receive(void) interrupt 4 using 3 {

```

```

    RI=0;                                /* 清接收中断标志 */
    if(RB8==1) {                          /* 如果为本节点地址帧 */
        if(SBUF==NODE_ADDR) SM2=0;      /* 则置 SM2=0 以便接收数据帧 */
        return;
    }
    buffer[pointer++]=SBUF;               /* 将收到的数据帧送接收缓冲区 */
    if(pointer>=COUNT)                  /* 若已收完全部数据, 则此次通信结束 */
        SM2=1;                          /* 置 SM2=1 准备下一次通信 */
}

```

8.5.3 8051 串行口的中断驱动程序

在 Keil Cx51 库函数中提供了两个通过 8051 串行口实现输入输出的底层函数 `_getkey()` 和 `putchar()`, 并且给出了这两个函数的源代码, 用户可以根据自己的需要适当修改这两个函数来完成不同的输入输出功能。下面给出一个实例, 包含 3 个文件: 主函数文件 `main.c`、串行口中断驱动函数文件 `sio.c` 和应用函数原型说明头文件 `sio.h`。在程序文件 `main.c` 中对 `_getkey()` 和 `putchar()` 函数进行了一些修改, 让它们调用串行口中断驱动程序 `sio.c` 中的应用函数来完成串行口的输入和输出, 实际上用户可以根据自己的需要进行修改。

`main.c` 程序文件代码如下:

```

#include <stdio.h>
#include "sio.h"
/*****
* 这是修改后的_getkey()函数, 它将取代运行库中原来的_getkey()函数。
* 功能: 等待从串行口输入字符, 需要调用串行口中断驱动程序 sio.c
*****/
char _getkey (void) {
    int k;
    do {
        k = com_getchar ();
    }
    while (k == -1);
    return ((unsigned char) k);
}

/*****
* 这是修改后的 putchar()函数, 它将取代运行库中原来的 putchar()函数。
* 功能: 从串行口输出一个字符, 需要调用串行口中断驱动程序 sio.c
*****/
char putchar (char c) {
    volatile unsigned int i;
    while (com_putchar (c) != 0) {
        for (i=0; i<1000; i++) {

```

```

        /*** 什么也不做 ***/
    }
}
return (c);
}

code char message [] =
    "This is a test to see if the interrupt driven
      serial I/O routines really work.\n\n";

/***** 主函数 *****/
void main (void) {
    com_initialize ();          /* 串行口初始化 */
    com_baudrate (38400);       /* 波特率设为 38400 */
    printf ("Interrupt-driver Serial I/O Example\r\n");
    printf (message);
    while (1) {
        unsigned char c;
        printf ("Please Press a key.\r\n");
        c = getchar ();
        printf ("\r\n");
        printf ("You pressed '%c'.\r\n\r\n", c);
    }
}

```

sio.c 程序文件如下:

```

#include <reg51.h>
#include <string.h>
#include "sio.h"

#define TBUF_SIZE    256          /* 这两行不要修改 */
#define RBUF_SIZE    256

static xdata unsigned char tbuf [TBUF_SIZE];
static xdata unsigned char rbuf [RBUF_SIZE];
static xdata unsigned char t_in = 0;
static xdata unsigned char t_out = 0;
static xdata unsigned char t_disabled = 0;
static xdata unsigned char r_in = 0;
static xdata unsigned char r_out = 0;

/***** 中断处理函数 *****/
* 功能: 完成数据的接收和发送

```

```

*****/
static void com_isr (void) interrupt 4 using 2 { /*
    if (RI != 0) { /* 接收 */
        RI = 0;
        if ((r_in + 1) != r_out)
            rbuf [r_in++] = SBUF;
    }
    if (TI != 0) { /* 发送 */
        TI = 0;
        if (t_in != t_out)
            SBUF = tbuf [t_out++];
        else
            t_disabled = 1;
    }
}

/***** 串行口初始化函数 *****/
* 功能: 初始化单片机的串行口
*****/
void com_initialize (void) {
    com_baudrate (1200); /* 利用定时器 T1 作为波特率发生器 */
    EA = 0; /* 关中断 */
    t_in = 0; /* 清除接收和发送缓冲区 */
    t_out = 0;
    t_disabled = 1;
    r_in = 0;
    r_out = 0;
    SM0 = 0; SM1 = 1; /* 串行口方式 1 */
    REN = 1; /* 允许接收 */
    TI = 0; /* 清除发送中断标志 */
    RI = 0; /* 清除接收中断标志 */
    ES = 1; /* 允许串行口中断 */
    PS = 0; /* 设置串行口中断为低优先级 */
    EA = 1; /* 开中断 */
}

/***** 波特率发生器函数 *****/
* 功能: 利用定时器 T1 产生由参数 baudrate 指定的波特率
*****/
void com_baudrate (unsigned baudrate) {
    EA = 0; /* 关中断 */
    TI = 0; /* 清除发送中断标志 */
    t_in = 0; /* 清除发送缓冲区 */

```

```

    t_out = 0;
    t_disabled = 1;          /* 禁止发送 */

    TR1 = 0;                 /* 停止定时器 T1 */
    ET1 = 0;                 /* 禁止 T1 中断 */
    PCON |= 0x80;            /* 波特率加倍 */
    TMOD &= 0xF0;            /* 设置 T1 工作方式 2 */
    TMOD |= 0x20;
    TH1 = (unsigned char) (256 - (XTAL / (16L * 12L * baudrate)));
    TR1 = 1;                 /* 启动 T1 */
    EA = 1;                  /* 开中断 */
}

/***** 串行口字符输出函数 *****/
* 功能: 从串行口输出一个字符 c
*****/
char com_putchar (unsigned char c) {
    if ((TBUF_SIZE - com_tbuflen ()) <= 2)
        return (-1);        /* 如果发送缓冲区满则返回-1 */

    EA = 0;                  /* 关中断 */
    tbuf [t_in++] = c;       /* 将数据加入到发送缓冲区并开中断 */
    if (t_disabled) {
        t_disabled = 0;
        TI = 1;
    }
    EA = 1;
    return (0);              /* 串行口输出正确, 返回 0 */
}

/***** 串行口字符输入函数 *****/
* 功能: 从串行口输入一个字符
*****/
int com_getchar (void) {
    int c;
    if (com_rbuflen () == 0)
        return (-1);        /* 如果接收缓冲区长度为 0 (未收到字符), 返回-1 */
    EA = 0;
    c = rbuf [r_out++];
    EA = 1;
    return (c);              /* 串行口输入正确, 返回输入的字符 */
}

```

```

/***** 计算接收缓冲区长度 *****/
* 功能: 计算接收缓冲区长度并返回给主调函数
*****/
unsigned char com_rbuflen (void) {
    return (r_in - r_out);
}

/***** 计算发送缓冲区长度 *****/
* 功能: 计算发送缓冲区长度并返回给主调函数
*****/
unsigned char com_tbuflen (void) {
    return (t_in - t_out);
}

```

sio.h 头文件代码如下:

```

void com_initialize (void);          /* 串行口初始化函数 */
void com_baudrate (unsigned baudrate); /* 波特率发生器函数 */
char com_putchar (unsigned char c);  /* 串行口字符输出函数 */
int com_getchar (void);              /* 串行口字符输入函数 */
unsigned char com_rbuflen (void);    /* 计算接收缓冲区长度函数 */
unsigned char com_tbuflen (void);    /* 计算发送缓冲区长度函数 */

```

8.5.4 利用 8051 串行口实现人机对话命令的输入输出

在单片机应用系统中,人机对话是需要考虑的一个重要功能。在简单的单机应用场合往往采用键盘—显示器来完成人机对话,随着技术的发展现在很多地方都要求多机应用,并且还要求能够进行远距离通信,利用 8051 单片机的串行口的通信功能,可以很方便地实现上—下位机之间的人机对话。下面给出一个具体实例,包含 3 个模块文件:主模块文件 main.c、命令处理模块文件 command.c 和新类型定义及应用函数原型说明文件 types.h。

主模块文件 main.c 代码如下:

```

#include<reg51.h>
#include<stdio.h>
#include<types.h>

uchar code *s = "Please input the following commands:\n"
    "Inport p      (read port)\n"
    "Outport p nnn  (write port)\n"
    "  where p=0..3, nnn=0..255\n"
    "Help or ?      (display this text)\n";

/***** 输出帮助信息函数 *****/
* 功能: 从串行口输出帮助提示信息
*****/

```



```

uchar help(){
    printf( "\n%s\n", s );
    return 0;
}

/***** 单片机 I/O 口输入命令函数 *****/
* 功能: 输入单片机 P0~P3 口的状态
* 命令格式: Inport p, 其中参数 p = 0~3 为端口号
*****/
uchar inport( uchar idata *buf ) {
    uint p;
    uchar v;
    if( sscanf( buf, "%i", &p ) != 1 )
        return 1;          /* 命令中缺少端口参数则返回错误号 1 */
    switch( p ){
        case 0: v = P0; break;    /* 输入 P0 状态 */
        case 1: v = P1; break;    /* 输入 P1 状态 */
        case 2: v = P2; break;    /* 输入 P2 状态 */
        case 3: v = P3; break;    /* 输入 P3 状态 */
        default: return 2;        /* 端口参数超出范围则返回错误号 2 */
    }
    printf( "Port P%d = %02X\n", p, (uint)v );
    return 0;
}

/***** 单片机 I/O 口输出命令函数 *****/
* 功能: 从单片机 P0~P3 口输出给定值
* 命令格式: Outport p nnn, 其中参数 p = 0~3 为端口号, nnn 为输出给定值 0~255
*****/
uchar outport( uchar idata *buf ) {
    uint p, v;
    if( sscanf( buf, "%i%i", &p, &v ) != 2 )
        return 1;          /* 命令中缺少端口参数则返回错误号 1 */
    switch( p ){
        case 0: P0 = v; break;    /* 从 P0 口输出给定值 */
        case 1: P1 = v; break;    /* 从 P1 口输出给定值 */
        case 2: P2 = v; break;    /* 从 P2 口输出给定值 */
        case 3: P3 = v | RXD_ | TXD_; break; /* 从 P3 口输出给定值 */
        default: return 2;        /* 端口参数超出范围则返回错误号 2 */
    }
    return 0;
}

```

```

/***** 主函数 *****/
* 功能: 初始化单片机串行口, 等待从串行口输入命令并调用命令处理函数
*****/
void main( void ) {
    uchar idata buf[21];
    uchar elevel;

    PCON |= SMOD_;          /* 初始化单片机的串行口 UART */
    SCON = TI_ + REN_ + SM1_;
    TMOD = T1_M1_;
    TH1 = (uchar) -(0.5 + XTAL / 12.0 / 16 / BAUD);
    TL1 = -1;
    TR1 = 1;

    for(;;){
        help();              /* 输出帮助提示信息 */
        putchar('>');
        gets( buf, 20 );     /* 等待从串行口输入命令 */
        switch( elevel = command(buf)){ /* 根据输入值调用命令处理函数 */
            case 0: break;
            default: printf( "Error %d\n\n", (uint)elevel );
        }
    }
}

```

命令处理模块文件 command.c 代码如下:

```

#include<stdlib.h>
#include<types.h>

typedef struct {
    uchar code *name;
    uchar (code *func)(uchar idata *);
}comm_struct;

comm_struct code comm_tab[] = {          /* 命令表 */
    "inport", inport,
    "outport", outport,
    "help", help,
    "?", help,
    "", NULL                               /* 命令表结束符 */
};

```

```

/***** 命令处理函数 *****/
* 功能: 根据串行口输入命令, 查命令表, 调用不同的命令函数
*****/
uchar command( uchar idata *buf ) {
    uchar i, j;                                /* 最多可处理 256 个命令 */
    for( i = 0;; )
        for( j = 0;; ){
            if( comm_tab[i].name[j] != 0 ){
                if( ((comm_tab[i].name[j] ^ buf[j]) & 0x5F) == 0 ){
                    j++;
                    continue;                /* 下一个输入字符 */
                }
                i++;
                break;                        /* 下一条命令 */
            }
            if( j == 0 ) return 255;        /* 查表未找到命令返回错误号 255 */
            return comm_tab[i].func( buf+j ); /* 执行命令 */
        }
}

```

新类型定义及应用函数原型说明文件 types.h 代码如下:

```

typedef unsigned char uchar;
typedef unsigned int uint ;

uchar help();
uchar inport( uchar idata *buf );
uchar outport( uchar idata *buf );
uchar command( uchar idata *buf );

#define XTAL      20e6
#define BAUD      9600
#define SMOD_     0x80
#define RXD_      0x01
#define TXD_      0x02
#define TI_       0x02
#define REN_      0x10
#define SM1_      0x40
#define T1_M1_    0x20

```

8.6 8051 单片机串行接口扩展应用编程

8.6.1 5 位共阴极 LED 驱动器 MC14489 的应用

MC14489 是 MOTOROLA 公司生产的串行接口 5 位 7 段 LED 译码/驱动芯片, 它不需要外接驱动三级管就可直接驱动 5 位共阴极 LED 数码显示器或 25 个 LED 指示灯, 也可实现共阴极 LED 数码显示器和 LED 指示灯的组合驱动。使用一个外接电阻 R_x 即可控制每一段的输出电流, 而不需要在每一段上都加段限流电阻。MC14489 的主要特性如下。

- 具有三线串行接口 (SPI), 可直接与具有 SPI 接口的 CPU 相连, 也可采用软件模拟 SPI 接口, 最大串行时钟频率可达 4MHz。
- 有一个 8 位的内部系统设置寄存器, 它的内容决定芯片的工作模式。
- 有一个 24 位的显示寄存器, 该寄存器用于存放包括小数点段在内的显示代码。内部系统设置寄存器和显示寄存器的内容都是通过串行接口输入的, MC14489 采用了所谓“位抓取”技术, 在向内部系统设置寄存器和显示寄存器送数时不需要引导码和地址码, 而是根据一次串行输入数据的字节数由内部自动确定送往哪一个寄存器。若输入的是单字节数据, 将被送往系统设置寄存器, 若输入的是多字节数据, 则将被送往显示寄存器。
- 除可通过一个公共的外接电阻 R_x 来确定每一段的最大峰值电流外, 还可由输入的显示代码决定全部显示段为满亮度显示或半亮度显示。
- 通过系统设置寄存器的第 0 位可设置全部显示段为全亮或全暗, 上电复位后芯片自动处于全暗的低功耗方式。
- 芯片内部具有 BCD 七段译码电路, 可显示 0~9 和 A~F 十六进制数, 还可显示 15 种其他字符。
- 多片 MC14489 可级连使用以增加显示位数。
- 工作电压范围: 4.5~6V。

MC14489 的引脚排列如图 8.35 所示。各引脚的功能如下:

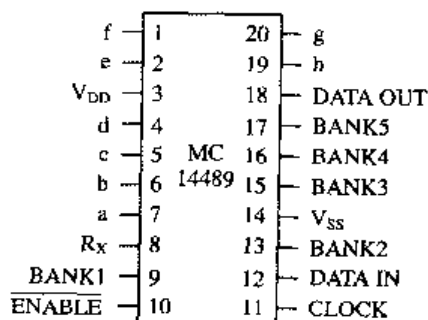


图 8.35 MC14489 的引脚排列

- DATA IN:** 串行数据输入。在 ENABLE 为低电平期间, 串行数据由 CLOCK 的上跳沿移入内部移位寄存器, 移入时最高位 (MSB) 在前。
- CLOCK:** 时钟输入脚。它是串行数据输入时所需的移位脉冲。最高时钟频率

- 为 4MHz, 最低可为 0Hz。
- ENABLE:** 串行输入使能。当其为低电平时, 使能 MC14489 的串行接口, 可以输入串行数据。当其为高时, 禁止 MC14489 的串行接口。在 ENABLE 的上升沿将数据寄存器中的内容打入系统设置寄存器或显示寄存器中 (取决于输入的字节数)。
- Rx:** 外接限流电阻。Rx 的取值范围为 700Ω 到无穷大。当 Rx 为 700Ω 时, 段的峰值电流约为 35mA, 当 Rx 为无穷大时, 显示器全灭。
- DATA OUT:** 串行数据输出。在移位时钟 CLOCK 的下降沿移出数据, 多片 MC14489 级联使用时, 可通过该引脚与下一个芯片的 DATA IN 脚相连来实现。不作级联使用时该引脚可悬空。
- a-h:** 各段输出驱动脚。在驱动 LED 数码显示器时, 可将它们分别接至 LED 的各阳极端。在驱动 LED 指示灯时, a~d 可驱动 20 个指示灯 (因为共有 5 个显示位), 输出 h 可驱动 5 个 LED 指示灯, e、f 和 g 空着。
- BANK1~BANK5:** 位扫描输出。它们是低阻抗的接地开关, 每个引脚可吸收 320mA 的电流。这些引脚可与 LED 数码显示器的共阴极直接相连。当芯片的扫描频率为 1kHz 时可得到较理想的显示效果。实际扫描频率一般取 700~1900Hz。
- V_{SS}:** 电源地线。
- V_{DD}:** 电源正端。

MC14489 内部系统设置寄存器中各位的作用如图 8.36 所示, 显示寄存器中各位的作用如图 8.37 所示。表 8-8 所示为三种不同译码方式的段译码表。

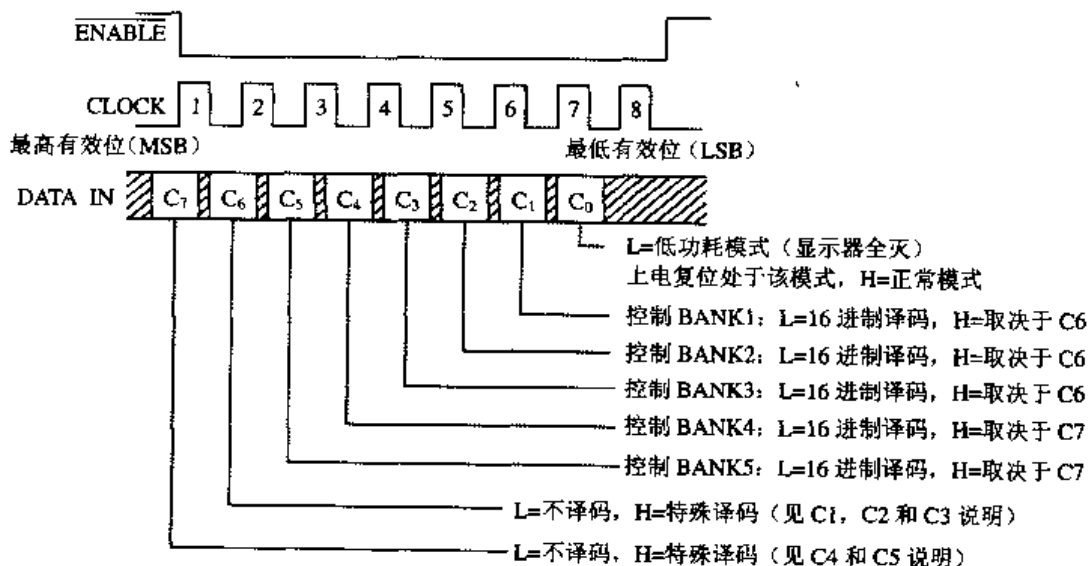


图 8.36 MC14489 内部设置寄存器中各位的作用

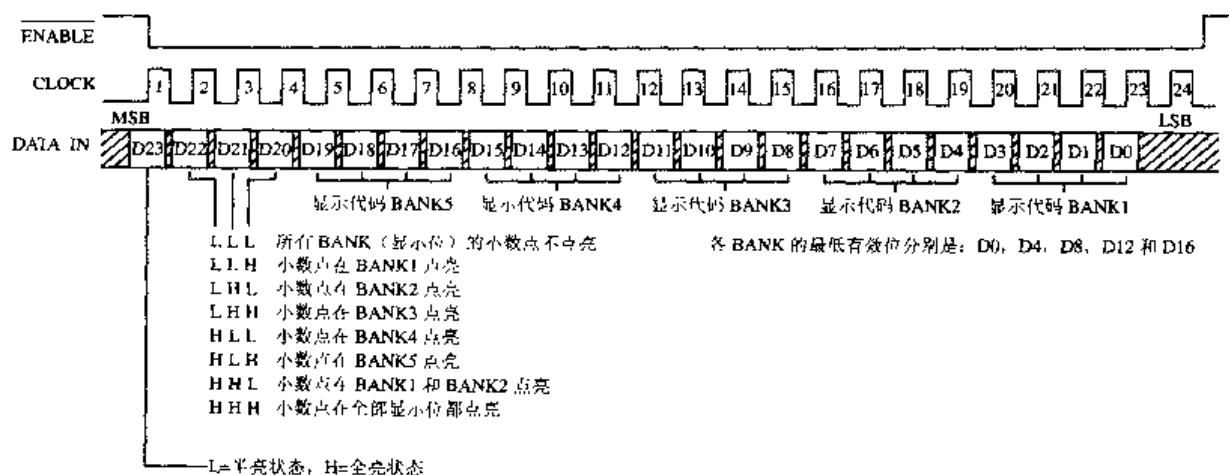


图 8.37 MC14489 内部显示寄存器中各位的作用

表 8-8 三种不同译码方式的段译码表

BANK		值		七段显示字符		指示灯			
						不译码			
16 进制数	2 进制数	16 进制	特殊	d	c	b	a		
	MSB	LSB	译码	译码					
0H	L	L	L	L	0				
1H	L	L	L	H	1	C			on
2H	L	L	H	L	2	H			on
3H	L	L	H	H	3	h			on
4H	L	H	L	L	4	J	on		
5H	L	H	L	H	5	L	on		on
6H	L	H	H	L	6	n	on	on	
7H	L	H	H	H	7	o	on	on	on
8H	H	L	L	L	8	P	on		
9H	H	L	L	H	9	r	on		on
AH	H	L	H	L	A	U	on	on	
BH	H	L	H	H	b	u	on	on	on
CH	H	H	L	L	c	y	on	on	
DH	H	H	L	H	d	-	on	on	on
EH	H	H	H	L	E	=	on	on	on
FH	H	H	H	H	F	□	on	on	on

图 8.38 所示为 8031 与 MC14489 的一种接口。8031 单片机内部没有 SPI 串行接口, 因此采用软件模拟的方法实现与三线串行接口芯片的连接, P1.0、P1.1 和 P1.2 分别连到 MC14489 的 DATA IN、CLOCK 和 ENABLE 引脚, 然后按图 8.36 和图 8.37 的时序用软件方法实现串行数据的输出和输入。

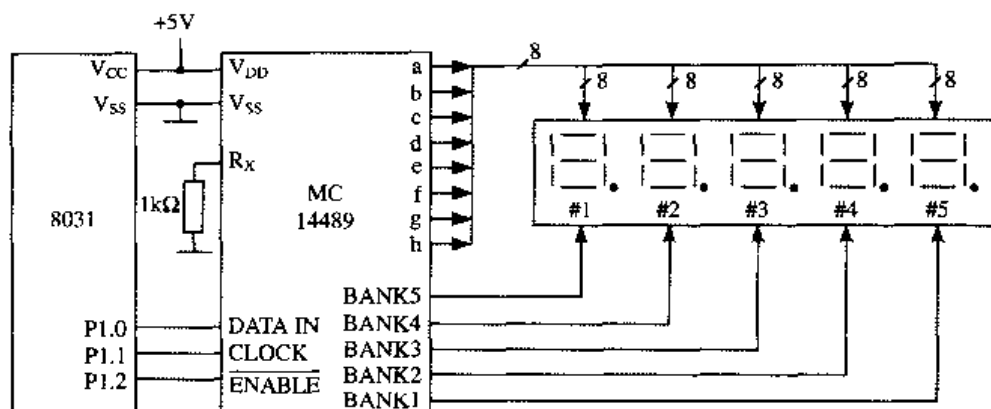


图 8.38 8031 单片机与 MC14489 的一种接口电路

下面给出对应于图 8.38 接口电路的 Cx51 驱动程序。

```
#pragma db oe sb
#define uchar unsigned char

sbit DATA    = 0x90 ;          /* 定义 P1.0 为 DATA IN */
sbit CLK      = 0x91 ;          /* 定义 P1.1 为 CLOCK */
sbit ENA      = 0x92 ;          /* 定义 P1.2 为 ENABLE */

void DSPCMD(uchar CMD) {        /* 单字节命令函数，写入 MC14489 内部设置寄存器 */
    uchar i;
    ENA=0;                      /* 使能 MC14489 */
    for (i=8;i>=1;i--) {        /* 写入单字节命令 */
        DATA=CMD&0x80;
        CMD=CMD<<1;
        CLK=0;
        CLK=1;
    }
    ENA=1;                      /* 禁止 MC14489 */
}

void DSPDATA(uchar DSCMD, uchar DSDATA1, uchar DSDATA2) {
    uchar DSP,i,j;              /* 多字节命令函数，写入 MC14489 显示寄存器 */
    i=0;
    ENA=0;                      /* 使能 MC14489 */
    While(i<24) {               /* 写入三字节显示数据 */
        if(i<8) {DSP=DSCMD;}
        else if(i<16) {DSP=DSDATA1;}
        else {DSP=DSDATA2;}
        for(j=8;j>=1;j--) {
            DATA=DSP&0x80;

```

```

        DSP=DSP<<1;
        CLK=0;
        CLK=1;
    }
    i=i+8;
}
ENA=1;          /* 禁止 MC14489 */
}

void main() {
    DSPCMD(0x01);          /* 单字节命令, 点亮所有 LED */
    DSPDATA(0x85,0x43,0x21); /* 在 5 位 LED 上分别显示 54321, 满亮度显示 */
    DSPDATA(0x01,0x23,0x45); /* 在 5 位 LED 上分别显示 12345, 半亮度显示 */
}

```

8.6.2 8 位共阴极 LED 驱动器 MAX7219 的应用

MAX7219 是 MAXIM 公司生产的一种串行接口方式 7 段共阴极 LED 显示驱动器, 其片内包含有一个 BCD 码到 B 码的译码器、多路复用扫描电路、字段和字位驱动器以及存储每个数字的 8×8 RAM, 每位数字都可以被寻址和更新, 允许对每一位数字选择 B 码译码或不译码。采用三线串行方式与 MCU 接口, 电路十分简单, 只需要一个 $10k\Omega$ 左右的外接电阻来设置所有 LED 的段电流。

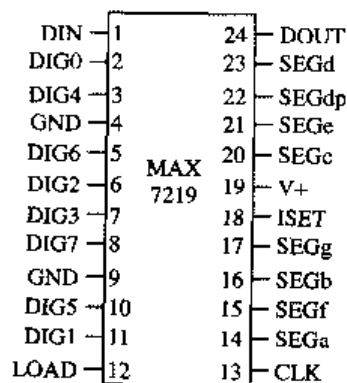


图 8.39 MAX7219 的引脚排列

MAX7219 的引脚排列如图 8.39 所示。各引脚功能如下:

- DIN:** 串行数据输入。在 CLK 时钟的上升沿, 串行数据被移入内部移位寄存器, 移入时最高位 (MSB) 在前。
- DIG0~DIG7:** 8 根字位驱动引脚, 它从 LED 显示器吸入电流。
- GND:** 地, 两根 GND 引脚必须相连。
- CLK:** 时钟输入。它是串行数据输入时所需的移位脉冲。最高时钟频率为 10MHz, 在 CLK 的上升沿串行数据被移入内部移位寄存器, 在 CLK 的下降沿数据从 DOUT 移出。

- SEGa~g,dp: 7 段和小数点驱动输出, 它提供 LED 显示器源电流。
 ISET: 通过一个 10k Ω 电阻 R_{SET} 接到 $V+$ 以设置峰值段电流。
 $V+$: +5V 电源电压。
 DOUT: 串行数据输出。输入到 DIN 的数据经过 16.5 个时钟周期后, 在 DOUT 端有效。

MAX7219 采用串行数据传输方式, 由 16 位数据包发送到 DIN 引脚的串行数据在每个 CLK 的上升沿被移入到内部 16 位移位寄存器中, 然后在 LOAD 的上升沿将数据锁存到数字或控制寄存器中。LOAD 信号必须在第 16 个时钟上升沿同时或之后, 但在下一个时钟上升沿之前变高, 否则将会丢失数据。DIN 端的数据通过移位寄存器传送, 并在 16.5 个时钟周期后出现在 DOUT 端。DOUT 端的数据在 CLK 的下降沿输出。串行数据以 16 位为一帧, 其中 D15~D12 可以任意, D11~D8 为内部寄存器地址, D7~D0 为寄存器数据, 格式如表 8-9 所示。

表 8-9 MAX7219 的串行数据格式

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
×	×	×	×	地 址				MSB	数 据						LSB

MAX7219 的数据传输时序如图 8.40 所示。

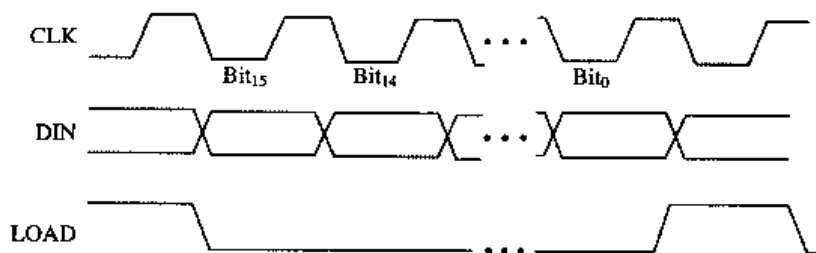


图 8.40 MAX7219 的数据传输时序

MAX7219 具有 14 个可寻址的内部数字和控制寄存器, 8 个数字寄存器由一个片内 8 \times 8 双端口 SRAM 实现, 它们可以直接寻址, 因此可以对单个数字进行更新, 并且只要 $V+$ 超过 2V, 数据就可以保留下去。控制寄存器有 5 个, 分别为: 译码方式、显示亮度、扫描界限 (扫描数位的个数)、停机和显示测试。另外还有一个控操作寄存器 (NO-OP), 在不改变显示或影响任一控制寄存器的条件下器件级联时, 它允许数据从 DIN 传送到 DOUT。表 8-10 所示为 MAX7219 的内部寄存器及其地址。

表 8-10 MAX7219 的内部寄存器及其地址

寄存器	地 址						十六进制代码
	D15~D12	D11	D10	D9	D8		
NO-OP	×	0	0	0	0		×0H
数字 0	×	0	0	0	1		×1H
数字 1	×	0	0	1	0		×2H
数字 2	×	0	0	1	1		×3H
数字 3	×	0	1	0	0		×4H

续表

寄存器	地 址						
	D15-D12	D11	D10	D9	D8	十六进制代码	
数字 4	×	0	1	0	1	×5H	
数字 5	×	0	1	1	0	×6H	
数字 6	×	0	1	1	1	×7H	
数字 7	×	1	0	0	0	×8H	
译码方式	×	1	0	0	1	×9H	
亮度	×	1	0	1	0	×AH	
扫描界限	×	1	0	1	1	×BH	
停机	×	1	1	0	0	×CH	
显示测试	×	1	1	1	1	×FH	

下面以表格形式对 MAX7219 内部寄存器中不同数据所表示的含义说明如下。表 8-11 所示为译码方式寄存器中数据的含义,从表中可见,寄存器中的每一位与一个数字位相对应,逻辑高电平选择 B 码译码,而逻辑低电平则选择旁路译码器。

表 8-11 译码方式寄存器 (地址 = ×9H)

含 义	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
7-0 位均不译码	0	0	0	0	0	0	0	0	00H
0 位译成 B 码, 7-1 均不译码	0	0	0	0	0	0	0	1	01H
3-0 位译成 B 码, 7-4 均不译码	0	0	0	0	1	1	1	1	0FH
7-0 位均译成 B 码	1	1	1	1	1	1	1	1	FFH

MAX7219 可用 V+和 ISET 之间所接外部电阻 R_{SET} 来控制显示亮度。来自段驱动器的峰值电流通常为进入 ISET 电流的 100 倍。 R_{SET} 既可为固定电阻,也可为可变电阻,以提供来自面板的亮度调节,其最小值为 9.52k Ω 。段电流的数字控制由内部脉宽调制 DAC 控制,该 DAC 通过亮度寄存器向低 4 位加载,该 DAC 将平均峰值电流按 16 级比例设计,从 R_{SET} 设置峰值电流的 31/32 的最大值到 1/32 的最小值,如表 8-12 所示,最大亮度出现在占空比为 31/32 时。

表 8-12 亮度寄存器 (地址 = ×AH)

占空比 (亮度)	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
1/32 (最小亮度)	×	×	×	×	0	0	0	0	×0H
3/32	×	×	×	×	0	0	0	1	×1H
5/32	×	×	×	×	0	0	1	0	×2H
...
29/32	×	×	×	×	1	1	1	0	×EH
31/32 (最大亮度)	×	×	×	×	1	1	1	1	×FH

扫描界限寄存器用于设置所显示的数字位,可以从 1 到 8。通常以扫描速率为 1300Hz、8 位数字、多路方式显示。因为所扫描数字的多少会影响显示亮度,所以要注意调整。如果扫描界限寄存器被设置为 3 个数字或更少,各数字驱动器将消耗过量的功率。因此 R_{SET}

电阻的值必须按所显示数字的位数多少适当调整,以限制各个数字驱动器的功耗。表 8-13 所示为扫描界限寄存器中数据的含义。

表 8-13 扫描界限寄存器 (地址 = xBH)

显示数字位	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
只显示第 0 位数字	x	x	x	x	x	0	0	0	x0H
显示第 0 位~第 1 位数字	x	x	x	x	x	0	0	1	x1H
显示第 0 位~第 2 位数字	x	x	x	x	x	0	1	0	x2H
...
显示第 0 位~第 6 位数字	x	x	x	x	x	0	1	1	x6H
显示第 0 位~第 7 位数字	x	x	x	x	x	1	1	1	x7H

当 MAX7219 处于停机方式时,扫描振荡器停止工作,所有的断电流源被拉到地,而所有的位驱动器被拉到 V+,此时 LED 将不显示。在数字和控制寄存器中的数据保持不变。停机方式可用于节省功耗或使 LED 处于闪烁。MAX7219 退出停机方式的时间不到 250 微秒,在停机方式下显示驱动器还可以进行编程,停机方式可以被显示测试功能取消。表 8-14 所示为停机寄存器中数据的含义。

表 8-14 停机寄存器 (地址 = xCH)

工作方式	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
停机	x	x	x	x	x	x	x	0	x0H
正常	x	x	x	x	x	x	x	1	x1H

显示测试寄存器有两种工作方式:正常和显示测试。在显示测试方式下 8 位数字被扫描,占空比为 31/32,通过不考虑(但不改变)所有控制寄存器和数据寄存器(包括停机寄存器)内的控制字来接通所有的 LED 显示器。表 8-15 所示为显示测试寄存器中数据的含义。

表 8-15 显示测试寄存器 (地址 = xFH)

工作方式	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
正常	x	x	x	x	x	x	x	0	x0H
显示测试	x	x	x	x	x	x	x	1	x1H

数字 0~7 寄存器受译码方式寄存器的控制:译码或不译码。数字寄存器可将 BCD 码译成 B 码 (0~9、-、E、H、L、P),如表 8-16 所示。如果不译码则数字寄存器中数据的 D6~D0 位分别对应 7 段 LED 显示器的 A~G 段, D7 位对应 LED 的小数点 DP,某一位数据为 1 则点亮与该位对应的 LED 段,数据为 0 则熄灭该段。

表 8-16 数字 0~7 寄存器 (地址 = x1H~x8H)

7 段字形	寄存器数据						点亮段						
	D7	D6~D4	D3	D2	D1	D0	DP	A	B	C	D	E	F G
0		x	0	0	0	0		1	1	1	1	1	0
1		x	0	0	0	1		0	1	1	0	0	0
2		x	0	0	1	0		1	1	0	1	1	0

续表

7 段字形	寄存器数据						点亮段							
	D7	D6-D4	D3	D2	D1	D0	DP	A	B	C	D	E	F	G
3		x	0	0	1	1		1	1	1	1	0	0	1
4		x	0	1	0	0		0	1	1	0	0	1	1
5		x	0	1	0	1		1	0	1	1	0	1	1
6		x	0	1	1	0		1	0	1	1	1	1	1
7		x	0	1	1	1		1	1	1	0	0	0	0
8		x	1	0	0	0		1	1	1	1	1	1	1
9		x	1	0	0	1		1	1	1	1	0	1	1
-		x	1	0	1	0		0	0	0	0	0	0	1
E		x	1	0	1	1		1	0	0	1	1	1	1
H		x	1	1	0	0		0	1	1	0	1	1	1
L		x	1	1	0	1		0	0	0	1	1	1	0
P		x	1	1	1	0		1	1	0	0	1	1	1
暗		x	1	1	1	1		0	0	0	0	0	0	0

注：小数点 DP 由 D7 位控制，D7=1 点亮小数点。

MAX7219 可以级联使用，这时需要用到空操作寄存器 (NO-OP)，空操作寄存器的地址为 $\times 0H$ 。将所有级联器件的 LOAD 端连在一起，将 DOUT 端连接到相邻 MAX7219 的 DIN 端。例如，将 4 个 MAX7219 级联使用，那么要对第 4 片 MAX7219 写入时，发送所需要的 16 位字，其后跟 3 个空操作代码 ($\times 0x$)，当 LOAD 变高时数据被锁存在所有器件中。前 3 个芯片接受空操作指令，而第 4 个芯片将接受预期的数据。

图 8.41 所示 8031 单片机与 MAX7219 的一种接口，8031 的 P1.0 连到 MAX7219 的 DIN 端，P1.1 连到 LOAD 端，P1.2 连到 CLK 端，采用软件模拟方式产生 MAX7219 所需的工作时序。下面给出相应的 Cx51 驱动程序。

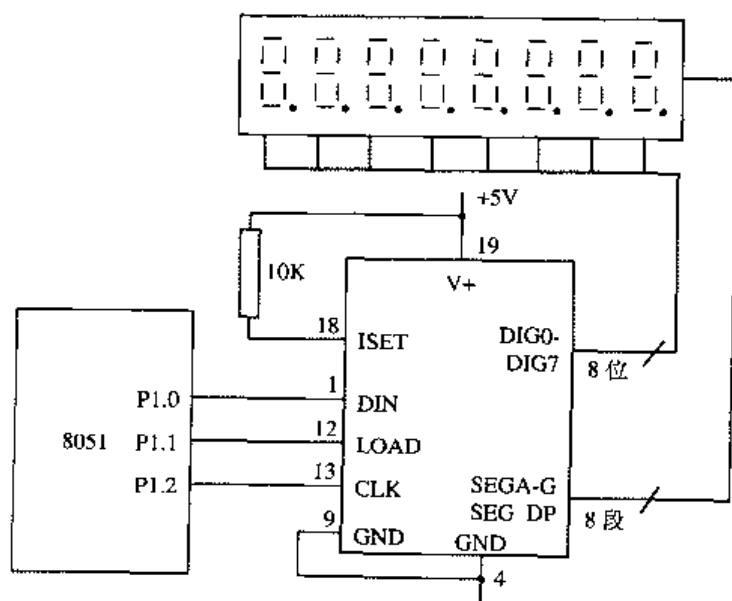


图 8.41 MAX7219 与 8031 单片机接口

```

#include <reg51.h>
/***** 定义 MAX7219 内部寄存器 *****/
#define REG_DECODE          0x09          /* 译码方式寄存器 */
#define REG_INTENSITY       0x0a          /* 亮度寄存器 */
#define REG_SCAN_LIMIT     0x0b          /* 扫描界限寄存器 */
#define REG_SHUTDOWN        0x0c          /* 停机寄存器 */
#define REG_DISPLAY_TEST    0x0f          /* 显示测试寄存器 */

#define INTENSITY_MIN       0x00          /* 最小显示亮度 */
#define INTENSITY_MAX       0x0f          /* 最大显示亮度 */

/***** 定义 MAX7219 端口信号 *****/
sbit DATA=P1^0;
sbit LOAD=P1^1;
sbit CLK=P1^2;

/***** 定义显示常数表格 *****/
static const struct {
    char  ascii;
    char  segs;
} Font[] = {
    {' ', 0x00}, {'0', 0x7e}, {'1', 0x30}, {'2', 0x6d}, {'3', 0x79}, {'4', 0x33},
    {'5', 0x5b}, {'6', 0x5f}, {'7', 0x70}, {'8', 0x7f}, {'9', 0x7b}, {'A', 0x77},
    {'B', 0x1f}, {'C', 0x4e}, {'D', 0x3d}, {'E', 0x4f}, {'F', 0x47}, {'H', 0x37},
    {'L', 0x0e}, {'O', 0x7e}, {'\0', 0x00}
};

/***** 函数原型说明 *****/
static void Write (unsigned char reg_number, unsigned char dataout);
static void SendByte (unsigned char dataout);
static unsigned char LookupCode (char character);
void Init (void);
void ShutdownStart (void);
void ShutdownStop (void);
void DisplayTestStart (void);
void DisplayTestStop (void);
void SetBrightness (char brightness);
void Clear (void);
void DisplayChar (char digit, char character);

/*****
* 函数原型: void Init();
* 功    能: 对 MAX7219 进行初始化, 调用任何其他函数之前必需先调用此函数。
*****/

```

```

*****/
void Init (void) {
    DATA=1;LOAD=1;CLK=1;
    Write(REG_SCAN_LIMIT, 7);          /* 扫描 8 位数字 */
    Write(REG_DECODE, 0x00);           /* 设置不译码方式 */
    ShutdownStop();                    /* 结束停机方式, 正常工作 */
    DisplayTestStop();                 /* 结束显示测试, 正常工作 */
    Clear();                           /* 清零全部显示器 */
    SetBrightness(INTENSITY_MAX);      /* 设置最大亮度 */
}

/*****
* 函数原型: void ShutdownStart();
* 功    能: 进入停机方式。
*****/
void ShutdownStart (void) {
    Write(REG_SHUTDOWN, 0);
}

/*****
* 函数原型: void ShutdownStop();
* 功    能: 结束停机方式。
*****/
void ShutdownStop (void) {
    Write(REG_SHUTDOWN, 1);
}

/*****
* 函数原型: void DisplayTestStart();
* 功    能: 启动显示测试。
*****/
void DisplayTestStart (void) {
    Write(REG_DISPLAY_TEST, 1);
}

/*****
* 函数原型: void DisplayTestStop();
* 功    能: 结束显示测试。
*****/
void DisplayTestStop (void) {
    Write(REG_DISPLAY_TEST, 0);
}

```

```

/*****
* 函数原型: void SetBrightness(char brightness);
* 功    能: 设置 LED 显示亮度。
* 参    数: 亮度值(brightness = 0-15)。
*****/
void SetBrightness (char brightness) {
    brightness &= 0x0f;                /* 屏蔽参数中的多余位 */
    Write(REG_INTENSITY, brightness);  /* 设置亮度 */
}

/*****
* 函数原型: void Clear();
* 功    能: 关闭全部显示器。
*****/
void Clear (void) {
    char i;
    for (i=1; i <= 8; i++)
        Write(i, 0x00);                /* 关闭全部显示器 */
}

/*****
* 函数原型: void DisplayChar(char digit, char character);
* 功    能: 在指定数位上显示字符。
* 参    数: digit = 数位(0-7)
           character = 显示字符(0-9, A-Z)。
*****/
void DisplayChar (char digit, char character) {
    Write(digit, LookupCode(character));
}

/*****
* 函数原型: static unsigned char LookupCode(char character);
* 功    能: 查段码表将显示字符转换为显示段码。
* 参    数: character = 显示字符。
* 返 回 值: 显示段码。
*****/
static unsigned char LookupCode (char character) {
    char i;
    for (i = 0; Font[i].ascii; i++)        /* 查表搜寻显示字符的 ASCII 码 */
        if (character == Font[i].ascii)
            return Font[i].segs;           /* 找到则返回对应的显示段码 */
    return 0;                               /* 未找到则返回 0 */
}

```

```

/*****
* 函数原型: static void Write (unsigned char reg_number, unsigned char
dataout);
* 功    能: 向指定的 MAX7219 内部寄存器写入数据。
* 参    数: reg_number = MAX7219 内部寄存器地址
            dataout = 待写入的数据。
*****/
static void Write (unsigned char reg_number, unsigned char dataout) {
    LOAD=1;                                /* 使 LOAD 变高, 启动串行数据发送 */
    SendByte(reg_number);                  /* 发送 MAX7219 内部寄存器地址 */
    SendByte(dataout);                    /* 发送待写入的数据 */
    LOAD=0;                                /* 使 LOAD 变低, 锁存数据 */
    LOAD=1;                                /* 使 LOAD 变高, 串行数据结束 */
}

/*****
* 函数原型: static void SendByte (unsigned char dataout);
* 功    能: 向 MAX7219 发送一个字节串行数据。
* 参    数: dataout = 待发送的数据。
*****/
static void SendByte (unsigned char dataout) {
    char i;
    for (i=8; i>0; i--) {
        unsigned char mask = 1 << (i - 1);
        CLK=0;                            /* 使 CLK 变低 */
        if (dataout & mask)                /* 发送一位数据 "1" 或 "0" */
            DATA=1;
        else
            DATA=0;
        CLK=1;                            /* 使 CLK 变高 */
    }
}

/*****
* 函数原型: main();
* 功    能: 初始化 MAX7219, 设置亮度, 在指定数位上显示字符。
*****/
void main(void) {
    Init ();                              /* 初始化 MAX7219 */
    ShutdownStart ();                     /* 停机 */
    ShutdownStop ();                      /* 结束停机 */
    DisplayTestStart ();                  /* 显示测试 */
}

```



```

DisplayTestStop ();          /* 结束显示测试 */
SetBrightness (0x02);        /* 设置显示亮度 */
DisplayChar (0x01, '0');      /* 第 1 位 LED 显示 0 */
DisplayChar (0x02, '1');      /* 第 2 位 LED 显示 1 */
DisplayChar (0x03, '2');      /* 第 3 位 LED 显示 2 */
DisplayChar (0x04, '3');      /* 第 4 位 LED 显示 3 */
Clear ();                     /* 关闭全部 LED 显示器 */
DisplayChar (0x05, 'A');      /* 第 5 位 LED 显示 A */
DisplayChar (0x06, 'B');      /* 第 6 位 LED 显示 B */
DisplayChar (0x07, 'C');      /* 第 7 位 LED 显示 C */
DisplayChar (0x08, 'D');      /* 第 8 位 LED 显示 D */
DisplayChar (0x08, '\0');      /* 第 8 位 LED 显示“暗码” */
while(1);
}

```

8.6.3 单总线温度传感器 DS1820 的应用

DS1820 是 DALLAS 公司生产的一种“单总线”温度传感器，它采用独特的单线接口方式，仅需一个端口引脚来发送或接收信息，在 MCU 和 DS1820 之间仅需一条数据线和一条地线进行接口；用于读写和温度转换的电源可以从数据线本身获得，无需外部电源。每个 DS1820 都有一个惟一的 ROM 序列号，所以可以将多只 DS1820 同时连在一根单总线上，进行简单的多点分布应用，包括温度控制、工业系统、消费品、温度计或任何热感测系统。DS1820 采用 TO-92 或 8 脚 SOIC 封装，引脚排列如图 8.42 所示。各引脚功能如下。

GND: 地。
DQ: 单线应用的数据输入 / 输出引脚。
VDD: 可选的外部供电电源引脚。
NC: 空脚。

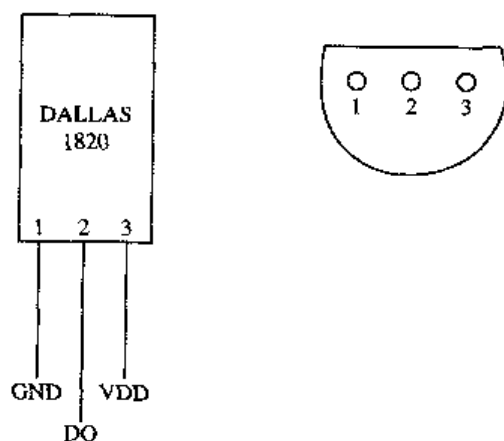


图 8.42 DS1820 的引脚排列

DS1820 内部有三个主要数字部件：64 位激光 ROM，温度传感器，非易失性温度报警触发器 TH 和 TL。DS1820 可以采用寄生电源方式工作，从单总线上汲取能量，在信号线

处于高电平期间把能量储存在内部电容里,在信号线处于低电平期间消耗电容上的电能工作,直到高电平到来再给寄生电源(电容)充电,DS1820 也可用外部 3~5.5V 电源供电,这两种供电方式的电路如图 8.43 所示。采用寄生电源方式时,VDD 引脚必须接地,另外为了得到足够的工作电流,应给 I/O 线提供一个强上拉,一般可以使用一个场效应管将 I/O 线直接拉到电源上。采用外部供电方式时可以不用强上拉,但只要外部电源处于工作状态,GND 引脚不得悬空。温度高于 100℃时,不推荐使用寄生电源,应采用外部电源供电。

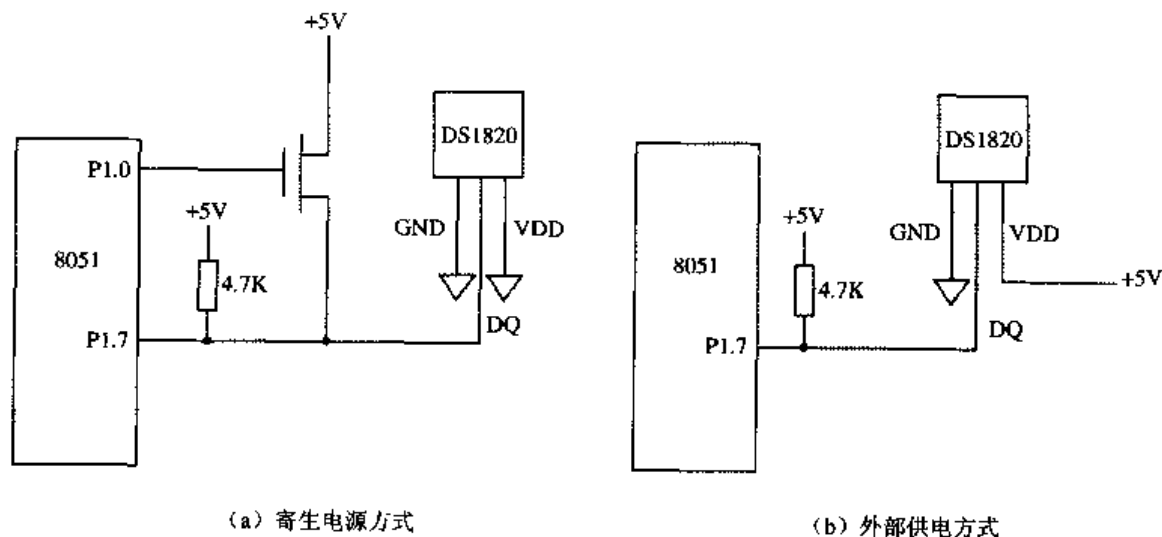


图 8.43 DS1820 的供电方式

DS1820 依靠一个单线端口通讯,必须先建立 ROM 操作协议,才能进行存储器和控制操作。因此,主机(MCU)必须首先提供下面 5 个 ROM 操作命令之一。

- ① 读出 ROM, 代码为 33H, 用于读出 DS1820 的序列号, 即 64 位激光 ROM 代码。
- ② 匹配 ROM, 代码为 55H, 用于辨识(或选中)某一特定的 DS1820 进行操作。
- ③ 搜索 ROM, 代码为 F0H, 用于确定总线上的节点数以及所有节点的序列号。
- ④ 跳过 ROM, 代码为 CCH, 命令发出后系统将对所有 DS1820 进行操作, 通常用于启动所有 DS1820 转换之前, 或系统中仅有一个 DS1820 时。

⑤ 报警搜索, 代码为 ECH, 主要用于鉴别和定位系统中超出程序设定的报警温度界限的节点。

这些命令对每个器件的激光 ROM 部分进行操作, 在单线总线上挂有多个器件时, 可以区分出单个器件, 同时可以向总线上的主机指明有多少器件或是什么型号的器件。

DS1820 内部存储器映像示于图 8.44。存储器由一个高速暂存器(scratchpad memory)和一个存储高低温报警触发值 TH 和 TL 的非易失性电可擦除 E²RAM 组成。当在单总线上通讯时, 暂存器帮助确保数据的完整性。数据先被写入暂存器, 并可被读回。数据经过校验后, 用一个拷贝暂存器命令会把数据传到非易性 E²RAM 中。这一过程确保更改存储器时数据的完整性。

DS1820 提供了如下存储器操作命令。

- ① 温度转换, 代码为 44H, 用于启动 DS1820 进行温度测量, 温度转换命令被执行后 DS1820 保持等待状态。如果主机在这条命令之后跟着发出读时间隙, 而 DS1820 又忙于做

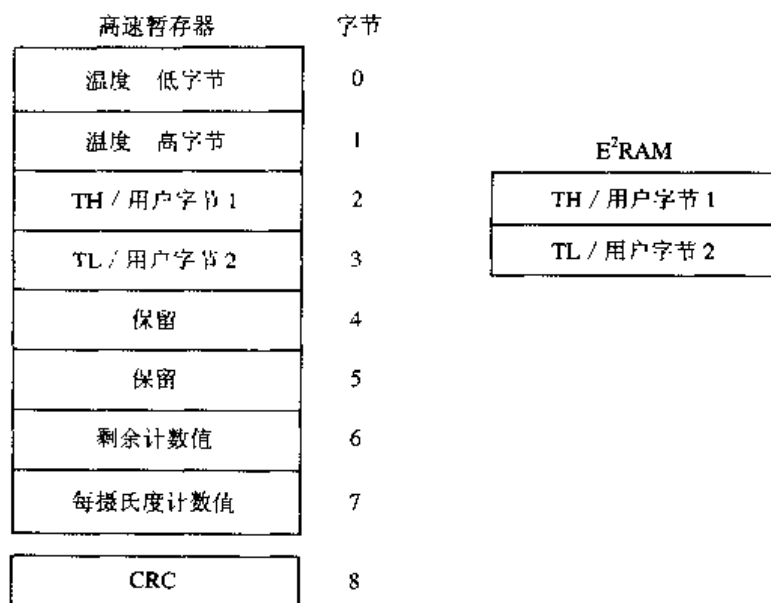


图 8.44 DS1820 的存储器映像

温度转换的话，DS1820 将在总线上输出“0”，若温度转换完成，则输出“1”。如果使用寄生电源，主机必须在发出这条命令后立即启动强上拉，并保持 750ms，在这段时间内单总线上不允许进行任何其他操作。

② 读暂存器，代码为 BEH，用于读取暂存器中的内容，从字节 0 开始最多可以读取 9 个字节，如果不想读完所有字节，主机可以在任何时间发出复位命令来中止读取。

③ 写暂存器，代码为 4EH，用于将数据写入到 DS1820 暂存器的地址 2 和地址 3 (TH 和 TL 字节)。可以在任何时刻发出复位命令来中止写入。

④ 复制暂存器，代码为 48H，用于将暂存器的内容复制到 DS1820 的非易性 E²RAM 中。即把温度报警触发字节存入非易失性存储器里。如果主机在这条命令之后跟着发出读时间隙，而 DS1820 又正在忙于把暂存器拷贝到 E²RAM 存储器，DS1820 就会输出一个“0”，如果拷贝结束的话，DS1820 则输出“1”。如果使用寄生电源，主机必须在这条命令发出后立即启动强上拉并最少保持 10ms，在这段时间内单总线上不允许进行任何其他操作。

⑤ 重读 E²RAM，代码为 B8H，用于将存储在非易性 E²RAM 中的内容重新读入到暂存器（温度触发器）中。这种复制操作在 DS1820 上电时自动执行，这样器件一上电暂存器里马上就存在有效的数据了。若在这条命令发出之后发出读时间隙，器件会输出温度转换忙的标志：“0”=忙，“1”=完成。

⑥ 读电源，代码为 B4H，用于将 DS1820 的供电方式信号发送到主机。若在这条命令发出之后发出读时间隙，DS1820 将返回它的供电模式：“0”= 寄生电源，“1”= 外部电源。

一条温度转换命令启动 DS1820 完成一次温度测量，测量结果以 16 位带符号位扩展的二进制补码形式存放在的高速暂存器中，占用暂存器的字节 0 (LSB) 和字节 1 (MSB)。用一条读暂存器内容的存储器操作命令可以把暂存器中的数据读出。温度报警触发器 TH 和 TL 各由一个 EEPROM 字节构成，可以用一条写存储器操作命令对 TH 和 TL 进行写入，对这些寄存器的读出需要通过暂存器。所有数据都是以最低有效位 (LSB) 在前的方式进

行读写。

DS1820 的温度分辨率为 9 位, 对应于 0.5°C , 其温度数据格式如表 8-17 所示。

表 8-17 DS1820 的温度数据格式

S	S	S	S	S	S	S	S	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}
---	---	---	---	---	---	---	---	-------	-------	-------	-------	-------	-------	-------	----------

表 8-18 所示为 DS1820 的温度—数据关系。

表 8-18 DS1820 的温度—数据关系

温度/ $^{\circ}\text{C}$	数据 (二进制)	数据 (十六进制)
+125 $^{\circ}\text{C}$	0000 0000 1111 1010	00FAH
+25 $^{\circ}\text{C}$	0000 0000 0011 0010	0032H
+0.5 $^{\circ}\text{C}$	0000 0000 0000 0001	0001H
0 $^{\circ}\text{C}$	0000 0000 0000 0000	0000H
-0.5 $^{\circ}\text{C}$	1111 1111 1111 1111	FFFFH
-25 $^{\circ}\text{C}$	1111 1111 1100 1110	FFCEH
-55 $^{\circ}\text{C}$	1111 1111 1001 0010	FF92H

用下述方法可以获得更高的分辨率: 先从暂存器中读取温度值 (暂存器的字节 0 和字节 1), 并从读取的值中截去 0.5°C 位 (最低有效位), 这个值叫做 TEMP_READ, 然后读取暂存器的字节 6 (COUNT_REMAIN), 最后读取暂存器的字节 7 (COUNT_PER_C)。利用下面的公式就可以计算出实际温度值:

$$\text{温度} = \text{TEMP_READ} - 0.25 = (\text{COUNT_PER_C} - \text{COUNT_REMAIN}) / \text{COUNT_PER_C}$$

DS1820 完成一次温度转换后, 将温度值和存储在 TH 和 TL 中的值进行比较, 如果测得的温度高于 TH 或低于 TL, 将置位器件内部的报警标志, 每进行一次测温对该标志进行一次更新。当报警标志置位时, DS1820 会对报警搜索命令有反应。这样就允许多个 DS1820 并联在一起同时测温, 如果某个地方的温度超过了限定值, 报警的器件就会被立即识别出来并读取其温度值, 而不用读未报警的器件。

通过单总线端口访问 DS1820 的过程如下:

- 初始化
- ROM 操作命令
- 存储器操作命令
- 数据处理

DS1820 需要严格的时序协议以确保数据的完整性。协议包括几种单线信号类型: 复位脉冲、存在脉冲、写 0、写 1、读 0 和读 1。所有这些信号, 除存在脉冲外, 都是由主机发出的。与 DS1820 之间的任何通讯都需要以初始化开始, 初始化包括一个由主机发出的复位脉冲和一个紧跟其后由从机发出的存在脉冲, 存在脉冲通知主机 DS1820 在总线上且已准备好进行发送和接收数据 (适当的 ROM 命令和存储器操作命令)。

DS1820 的数据读写是通过时间隙处理位和命令字来完成信息交换的。说明如下。

① 写时间隙：当主机把数据线从逻辑高电平拉到逻辑低电平的时候，写时间隙开始。有两种写时间隙：写 1 时间隙和写 0 时间隙。所有写时间隙必须最少持续 $60\mu\text{s}$ ，包括两个写周期间至少 $1\mu\text{s}$ 的恢复时间。DQ 引脚上的电平变低后，DS1820 在一个 $15\mu\text{s}$ 到 $60\mu\text{s}$ 的时间窗口内对 DQ 引脚采样。如果 DQ 引脚是高电平，就是写 1，如果 DQ 引脚是低电平，就是写 0。主机要生成一个写 1 时间隙，必须把数据线拉到低电平然后释放，在写时间隙开始后的 $15\mu\text{s}$ 内允许数据线拉到高电平。主机要生成一个写 0 时间隙，必须把数据线拉到低电平并保持 $60\mu\text{s}$ 。

② 读时间隙：从 DS1820 读取数据时，主机生成读时间隙。当主机把数据线从逻辑高电平拉到逻辑低电平时，读时间隙开始。数据线必须保持至少 $1\mu\text{s}$ ；从 DS1820 输出的数据在读时间隙的下降沿出现后 $15\mu\text{s}$ 内有效。因此，主机在读时间隙开始后必须停止把 DQ 引脚驱动为低电平 $15\mu\text{s}$ ，以读取 I/O 脚状态。在读时间隙的结尾，DQ 引脚将被外部上拉电阻拉到高电平。所有读时间隙最少必须为 $60\mu\text{s}$ ，包括两个读周期时间和至少 $1\mu\text{s}$ 的恢复时间。

下面给出 DS1820 的 ROM 搜索和温度转换 Cx51 驱动程序，单片机的 P1.7 引脚用作单总线接口。

```

/***** 本程序是基于 DS5000/DS2251T 单片机开发的 *****/
#include <absacc.h>
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <reg5000.h>
/***** 参数配置 *****/
#define XtalFreq (11059490) /* 主晶振频率 */
#define CntrFreq (XtalFreq/12) /* 主计数器频率 */
#define BaudRate (38400) /* 波特率 */
#define CntrTime (8) /* 计数器的周期数 */
#define Ft (32768.0) /* 目标晶振频率 */

#define FALSE 0
#define TRUE 1

sbit DQ = 0x97; /* 定义 DQ 引脚为 P1.7 */

/***** 全局变量 *****/
unsigned char ROM[8]; /* ROM 位 */
unsigned char lastDiscrep = 0; /* 上次差值 */
unsigned char doneFlag = 0; /* 完成标志 */
unsigned char FoundROM[5][8]; /* ROM 代码表 */
unsigned char numROMs;
unsigned char dowcrc;

```

```

unsigned char code dscrc_table[] = {
0, 94,188,226, 97, 63,221,131,194,156,126, 32,163,253, 31, 65,
157,195, 33,127,252,162, 64, 30, 95, 1,227,189, 62, 96,130,220,
35,125,159,193, 66, 28,254,160,225,191, 93, 3,128,222, 60, 98,
190,224, 2, 92,223,129, 99, 61,124, 34,192,158, 29, 67,161,255,
70, 24,250,164, 39,121,155,197,132,218, 56,102,229,187, 89, 7,
219,133,103, 57,186,228, 6, 88, 25, 71,165,251,120, 38,196,154,
101, 59,217,135, 4, 90,184,230,167,249, 27, 69,198,152,122, 36,
248,166, 68, 26,153,199, 37,123, 58,100,134,216, 91, 5,231,185,
140,210, 48,110,237,179, 81, 15, 78, 16,242,172, 47,113,147,205,
17, 79,173,243,112, 46,204,146,211,141,111, 49,178,236, 14, 80,
175,241, 19, 77,206,144,114, 44,109, 51,209,143, 12, 82,176,238,
50,108,142,208, 83, 13,239,177,240,174, 76, 18,145,207, 45,115,
202,148,118, 40,171,245, 23, 73, 8, 86,180,234,105, 55,213,139,
87, 9,235,181, 54,104,138,212,149,203, 41,119,244,170, 72, 22,
233,183, 85, 11,136,214, 52,106, 43,117,151,201, 74, 20,246,168,
116, 42,200,150, 21, 75,169,247,182,232, 10, 84,215,137,107, 53};

/***** 延时函数 *****/
* 功能: 在 11.059MHz 的晶振条件下调用本函数需要 24 $\mu$ s, 然后每次计数需 16 $\mu$ s.
*****/
void delay(int useconds) {
    int s;
    for (s=0; s<useconds;s++);
}

/***** 复位函数 *****/
* 功能: 完成单总线的复位操作。
* 复位时间为 480 $\mu$ s, 因此延时时间为  $(480-24)/16 = 28.5$ , 取 29 $\mu$ s。
* 经过 70 $\mu$ s 之后检测存在脉冲, 因此延时时间为  $(70-24)/16 = 2.875$ , 取 3 $\mu$ s。
*****/
unsigned char ow_reset(void) {
    unsigned char presence;
    DQ = 0;          /* 将 DQ 线拉低 */
    delay(29);        /* 保持 480 $\mu$ s */
    DQ = 1;          /* DQ 返回高电平 */
    delay(3);         /* 等待存在脉冲 */
    presence = DQ;     /* 获得存在信号 */
    delay(25);        /* 等待时间隙结束 */
    return(presence); /* 返回存在信号, 0 = 器件存在, 1 = 无器件 */
}

/***** 位写入函数 *****/

```

```

* 功能: 向单总线写入 1 位值: bitval
*****/
void write_bit(char bitval) {
    DQ = 0;                /* 将 DQ 拉低开始写时间隙 */
    if(bitval==1) DQ =1;    /* 如果写 1, DQ 返回高电平 */
    delay(5);              /* 在时间隙内保持电平值不变 */
    DQ = 1; /* Delay 函数每次循环延时 16 $\mu$ s, 因此 delay(5) = 104 $\mu$ s */
}

/***** 字节写入函数 *****/
* 功能: 向单总线写入一个字节值: val。
*****/
void write_byte(char val) {
    unsigned char i;
    unsigned char temp;
    for (i=0; i<8; i++) { /* 写入字节, 每次写入一位 */
        temp = val>>i;
        temp &= 0x01;
        write_bit(temp);
    }
    delay(5);
}

/***** 位读取函数 *****/
* 功能: 从单总线上读取一位信号, 所需延时时间为 15 $\mu$ s, 因此无法调用前面定义
*       的 delay() 函数, 而采用一个 for() 循环来实现延时。
*****/
unsigned char read_bit(void) {
    unsigned char i;
    DQ = 0;                /* 将 DQ 拉低开始读时间隙 */
    DQ = 1;                /* 再将其拉高 */
    for (i=0; i<3; i++); /* 延时 15 $\mu$ s */
    return(DQ);            /* 返回 DQ 线上的电平值 */
}

/***** 字节读取函数 *****/
* 功能: 从单总线读取一个字节的值。
*****/
unsigned char read_byte(void) {
    unsigned char i;
    unsigned char value = 0;
    for (i=0; i<8; i++) { /* 读取字节, 每次读取一个字节 */
        if(read_bit()) value|=0x01<<i; /* 然后将其左移 */
    }
}

```

```

        delay(6);
    }
    return(value);
}

```

/* ***** 读取 ROM 代码函数 ***** */

* 功能: 用于读取单总线上单个 DS1820 器件的 ROM 代码, 对于多个器件需要采用搜索 ROM 函数。

***** */

```

void Read_ROMCode(void) {
    int n;
    char dat[9];
    printf("\nReading ROM Code\n");
    ow_reset();
    write_byte(0x33);
    for (n=0;n<8;n++){dat[n]=read_byte();}
    printf("\nROMCode=%X%X%X%X\n",
           dat[7],dat[6],dat[5],dat[4],dat[3],dat[2],dat[1],dat[0]);
}

```

/* ***** 单总线 CRC 函数 ***** */

* 功能: 完成一次循环冗余校验, 进行搜索 ROM 操作时应包含循环冗余校验。

***** */

```

unsigned char ow_crc( unsigned char x) {
    dowcrc = dscrc_table[dowcrc^x];
    return dowcrc;
}

```

/* ***** NEXT 函数 ***** */

* 功能: 搜索单总线上的下一个器件, 如果单总线上没有其他器件则返回“假”。

***** */

```

unsigned char Next(void) {
    unsigned char m = 1;    /* ROM 位索引 */
    unsigned char n = 0;    /* ROM 字节索引 */
    unsigned char k = 1;
    unsigned char x = 0;
    unsigned char discrepMarker = 0;
    unsigned char g;        /* 输出位 */
    unsigned char nxt;      /* 返回值 */
    int flag;
    nxt = FALSE;
    dowcrc = 0;
    flag = ow_reset();      /* 复位单总线 */
    if(flag||doneFlag) {    /* 如果没有其他器件则返回“假” */

```



```

        lastDiscrep = 0;
        return FALSE;
    }
    write_byte(0xF0);          /* 发送搜索 ROM 命令 */
    do {
        x = 0;
        if(read_bit()==1) x = 2;
        delay(6);
        if(read_bit()==1) x |= 1;
        if(x == 3) break;
        else {
            if(x>0) g = x>>1;
            else {
                if(m<lastDiscrep) g = ((ROM[n]&k)>0);
                else g = (m==lastDiscrep);
                if (g==0) discrepMarker = m;
            }
            if(g==1) ROM[n] |= k;
            else ROM[n] &= ~k;
            write_bit(g);
            m++;
            k = k<<1;
            if(k==0) {
                ow_crc(ROM[n]);
                n++; k++;
            }
        }
    } while(n<8);              /* 循环, 直到全部 ROM 字节 0~7 都完成 */
    if(m<65535||!dowcrc) lastDiscrep=0;
    else {                      /* 搜索成功, 置位 lastDiscrep, lastOne 和 nxt */
        lastDiscrep = discrepMarker;
        doneFlag = (lastDiscrep==0);
        nxt = TRUE;            /* 表示搜索还未结束, 单总线上还有其他器件 */
    }
    return nxt;
}

/***** FIRST 函数 *****/
* 功能: 复位当前 ROM 搜索状态并调用 NEXT 函数搜索单总线的第一个器件。
*****/
unsigned char First(void) {
    lastDiscrep = 0;
    doneFlag = FALSE;

```

```

    return Next();
}

/***** 读暂存器 *****/
* 功能: 读取暂存器中 9 个字节的数据。
*****/
void Read_ScratchPad(void) {
    int j;
    char pad[10];
    printf("\nReading ScratchPad Data\n");
    write_byte(0xBE);
    for (j=0;j<9;j++){pad[j]=read_byte();}
    printf("\n ScratchPAD DATA =%X%X%X%X%X%X\n",
        pad[8],pad[7],pad[6],pad[5],pad[4],pad[3],pad[2],pad[1],pad[0]);
}

/***** FIND DEVICES *****/
* 功能: 首先复位单总线以确定是否存在任何器件, 如果存在则将其唤醒。然后调用 FIRST 函数跟踪冲突位, 并返回到 NEXT 函数。NEXT 函数完成鉴别单总线上每个器件惟一 ROM 代码的大部分工作。
*****/
void FindDevices(void) {
    unsigned char m;
    if(!low_reset()) {                /* 如果单总线上存在器件则开始处理 */
        if(First()) {                 /* 至少发现一个器件才开始 */
            numROMs=0;
            do {
                numROMs++;
                for(m=0;m<8;m++) {
                    FoundROM[numROMs][m]=ROM[m]; /* 对发现的器件鉴别 ROM 代码 */
                }
                printf("\nROM CODE =%02X%02X%02X%02X\n",
                    FoundROM[5][7],FoundROM[5][6],FoundROM[5][5],FoundROM[5][4],
                    FoundROM[5][3],FoundROM[5][2],FoundROM[5][1],FoundROM[5][0]);
            } while (Next()&&(numROMs<10)); /* 一直持续到没有发现其他器件 */
        }
    }
}

/***** Match ROM 函数 *****/
* 功能: 选中某一特定的 DS1820 进行操作。
*****/

```

```

unsigned char Send_MatchRom(void) {
    unsigned char i;
    if(ow_reset()) return FALSE;
    write_byte(0x55);          /* 发送匹配 ROM 命令 */
    for(i=0;i<8;i++) {
        write_byte(FoundROM[numROMs][i]); /* 发送 ROM 代码 */
    }
    return TRUE;
}

/***** 读取温度函数 *****/
* 功能: 如果单总线节点上只有一个器件则可以直接调用本函数。如果节点上有多个器
*       件, 为了避免数据冲突, 应使用 Match ROM 函数来选中特定器件。
* 注: 本函数是根据 DS1820 的温度数据格式编写的, 若用于 DS18B20, 必须根据
*       DS18B20 的温度数据格式作适当修改。
*****/
void Read_Temperature(void) {
    char get[10];
    char temp_lsb,temp_msb;
    int k;
    char temp_f,temp_c;
    ow_reset();
    write_byte(0xCC);          /* 跳过 ROM */
    write_byte(0x44);          /* 启动温度转换 */
    delay(5);
    ow_reset();
    write_byte(0xCC);          /* 跳过 ROM */
    write_byte(0xBE);          /* 读暂存器 */
    for (k=0;k<9;k++){get[k]=read_byte();}
    printf("\n ScratchPAD DATA = %X%X%X%X%X\n",
        get[8],get[7],get[6],get[5],get[4],get[3],get[2],get[1],get[0]);
    temp_msb = get[1];
    temp_lsb = get[0];
    if (temp_msb <= 0x80){
        temp_lsb = (temp_lsb/2); /* 移位得到完整的温度值 */
    }
    temp_msb = temp_msb & 0x80; /* 屏蔽除符号位之外的所有位 */
    if (temp_msb >= 0x80) {
        temp_lsb = (~temp_lsb)+1; /* 对于负温度值取补 */
    }
    if (temp_msb >= 0x80) {
        temp_lsb = (temp_lsb/2); /* 移位得到完整的温度值 */
    }
    if (temp_msb >= 0x80) {

```

```

        temp_lsb = ((-1)*temp_lsb);    /* 加入符号位 */
    }
    printf( "\nTempC= %d degrees C\n",
            (int)temp_lsb );           /* 输出摄氏温度值 */
    temp_c = temp_lsb;                /* 华氏温度转换 */
    temp_f = (((int)temp_c)* 9)/5 + 32;
    printf( "\nTempF= %d degrees F\n",
            (int)temp_f );           /* 输出华氏温度值 */
}

/***** main 函数 *****/
main() {
    unsigned char Select_Type;
    TA = 0xAA;                        /* 设置 DS5000T 存储器为同步访问(timed access) */
    TA = 0x55;
    PCON = 0x00;                      /* 禁止看门狗 */
    SCON = 0x50;                      /* 设置串行口: 方式 1, 8 位 UART, 允许接收 */
    TMOD = 0x21;                      /* 设置定时器 T1: 方式 2, 8 位自动重装 */
                                    /* 设置定时器 T0: 方式 1, 16 位定时器 */
    PCON |= 0x80;                     /* SMOD = 1 波特率加倍 */
    TH0=TL0 = 0;
    TH1=TL0 = (unsigned int)(256 - ((XtalFreq/BaudRate)/192));
    TR0 = 1;                          /* 启动 T0 */
    TR1 = 1;                          /* 启动 T1 */
    TI = 1;                          /* 启动发送 UART 中的第一个字符 */

    printf (" Source for DS1820 Temperature Reading and\n");
    printf (" Search ROM code.\n");
    printf ("C51 Program for DS5000 or 8051 Compatible Microcontroller");
    printf("\n*****\n");
    printf (" Select Menu Option\n");
    printf (" 1. One-Wire Reset\n");
    printf (" 2. Read ROM Code of Single Device On Net\n");
    printf (" 3. Perform Search ROM\n");
    printf (" 4. Read Scratch PAD\n");
    printf (" 5. Read Temperature\n");
    printf (" 6. Find All Devices\n");
    do {
        EA = 0;                      /* 关中断 */
        TA = 0xAA;                    /* 同步访问(timed access) */
        TA = 0x55;
        MCON = MCON |= 0x04;         /* 允许 CE2 */
        TA = 0xAA;                    /* 同步访问(timed access) */
    } while (1);
}

```

```

TA = 0x55;
MCON = 0xC8;           /* 禁止 CE2 */
EA = 1;                /* 开中断 */
Select_Type = _getkey(); /* 从键盘输入选择数字 */
switch(Select_Type){
    case '1': printf ("\n 1. Sent 1-Wire Reset\n");
        ow_reset();
        break;
    case '2': printf ("\n 2. Read ROM Code of Single Device On Net\n");
        ow_reset();
        Read_ROMCode();
        break;
    case '3': printf ("\n 3. Performing Search ROM\n");
        ow_reset();
        First();
        printf ("\nROM CODE =%02X%02X%02X%02X\n",
            FoundROM[5][7],FoundROM[5][6],FoundROM[5][5],FoundROM[5][4],
            FoundROM[5][3],FoundROM[5][2],FoundROM[5][1],FoundROM[5][0]);
        break;
    case '4': printf ("\n 4. Read Scratch PAD\n");
        ow_reset();
        write_byte(0xCC);           /* 跳过 ROM */
        Read_ScratchPad();
        break;
    case '5': printf ("\n 5. Read Temperature\n");
        Read_Temperature();         /* 读取温度值 */
        break;
    case '6': printf ("\n 6. Find All Devices\n");
        ow_reset();
        FindDevices();
        break;
    default: printf ("\n Typo: Select Another Menu Option\n");
        break;
};
} while(1);
}

```

8.6.4 实时时钟芯片 DS1302 的应用

DS1302 是美国 DALLAS 公司推出的一种串行接口实时时钟芯片。芯片内部具有可编程日历时钟和 31 个字节的静态 RAM，日历时钟可自动进行闰年补偿，计时准确，接口简单，使用方便，工作电压范围宽（2.5V~5.5V），功耗低，芯片自身还具有对备份电池进行涓流充电功能，可有效延长备份电池的使用寿命。DS1302 采用 8 脚 DIP 封装，其引脚排

列如图 8.45 所示, 各引脚功能如下。

- V_{cc1}, V_{cc2} : 电源输入。
 GND: 地。
 X_1, X_2 : 外接 32.768KHz 石英晶振输入。
 RST: 复位端。
 I/O: 数据输入 / 输出端。
 SCLK: 串行时钟输入。

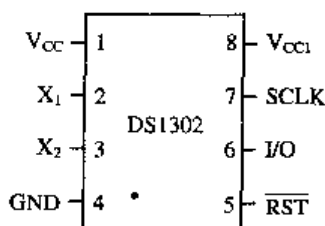


图 8.45 DS1302 的引脚排列

图 8.46 所示为 8051 单片机与 DS1302 的一种接口电路, 单片机与 DS1302 之间采用 3 线串行通信方式, RST (P1.5) 为通信允许信号, RST=1 允许通信, RST=0 禁止通信, IO (P1.4) 为双向串行数据传送信号, SCLK (P1.3) 为串行数据的位同步脉冲信号。8051 作为主机通过控制 RST、SCLK 和 IO 信号实现两芯片间的数据传送。DS1302 芯片的 X_1 和 X_2 端外接 32.768KHz 的石英晶振, V_{cc1} 和 V_{cc2} 是电源引脚, 单电源供电时接 V_{cc1} 脚, 双电源供电时主电源接 V_{cc2} , 备份电池接 V_{cc1} , 如果采用可充电镍镉电池, 可启用内部涓流充电器在主电压正常时向电池充电, 以延长电池使用时间。备份电池也可用 1 微法以上的超容量电容代替, 需要注意备份电池电压应略低于主电源工作电压。

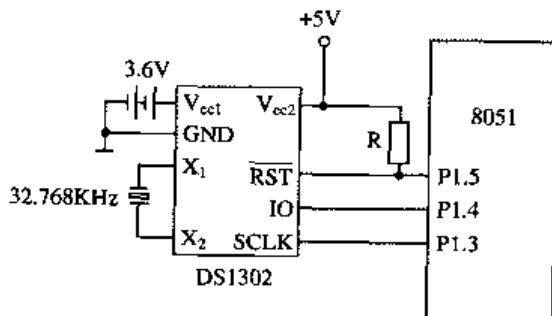


图 8.46 8051 单片机与 DS1302 的接口

数据传送是以 8051 单片机为主控芯片进行的, 每次传送时由 8051 向 DS1302 写入一个命令字节开始。命令字节的格式如下:

D7	D6	D5	D4	D3	D2	D1	D0
1	RAM/CK	A4	A3	A2	A1	A0	RD/W

命令字节的最高位必须为 1, RAM/CK 位为 DS1302 片内 RAM/时钟选择位, RAM/CK=1 选择 RAM 操作, RAM/CK=0 选择时钟操作。RD/W 位为读写控制位: RD/W=1 为读操作, 表示 DS1302 接受完命令字节后, 按指定的选择对象及寄存器 (或 RAM) 地址,

读取数据并通过 IO 线传送给单片机 8051, RD/W=0 为写操作, 表示 DS1302 接受完命令字节后, 紧接着再接收来自于单片机 8051 的数据字节并写入到 DS1302 相应的寄存器或 RAM 单元中。A4~A0 为片内日历时钟寄存器或 RAM 的地址选择位。

DS1302 的数据格式如图 8.47 所示。选择时钟操作时, A4~A0 的值 00H~06H 对应日历时钟寄存器地址, 其中以 BCD 码形式分别存放秒、分、时、日、月、星期和年等信息。秒寄存器的最高位为时钟暂停控制位, 该位为 0 时时钟振荡器暂停, DS1302 进入低功耗状态, 该位为 1 时启动时钟。地址值 07H 为写保护控制寄存器, 只有当该寄存器的最高位 WP=0 时才能够对日历时钟或 RAM 的内容进行写操作, WP=1 时禁止写操作。地址值 08H 为涓流充电控制寄存器, 控制内部涓流充电过程及充电电路的连接方式, 该字节各位数据

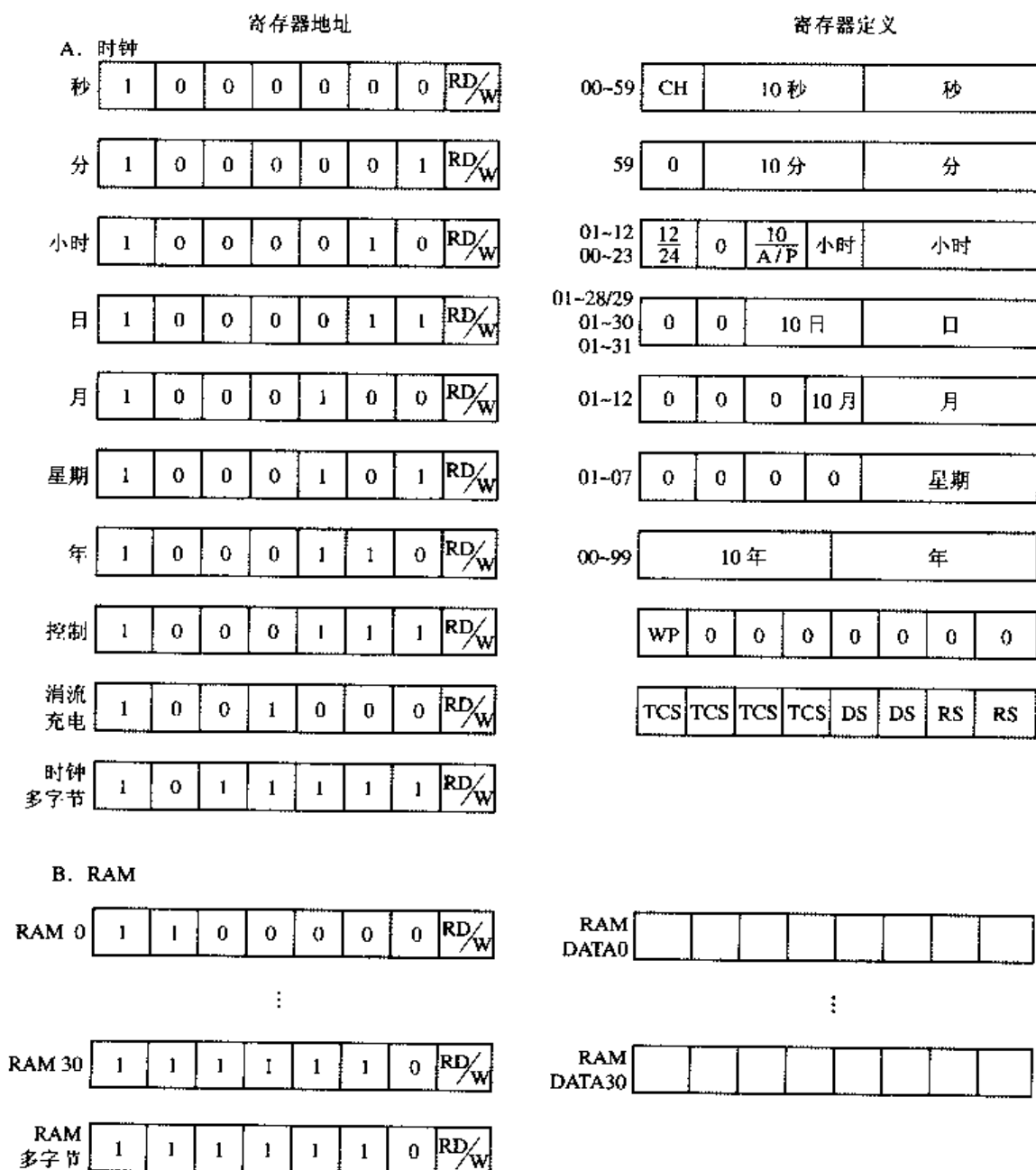


图 8.47 DS1302 的数据格式

与涓流充电控制寄存器的关系如图 8.48 所示。地址值 1FH 为多字节突发方式 (burst) 控制寄存器, 通过对该寄存器寻址, 可以将对日历时钟或 RAM 的读写操作规定为多字节方式。以多字节方式写时钟寄存器时, 必须按数据传送的次序写入最先 8 个寄存器; 而以多字节方式写 RAM 时, 为了传送数据不必写入所有 31 个 RAM 字节。

选择片内 RAM 操作时 A4~A0 用于表示片内 RAM 单元地址, 地址范围为 00~1EH。

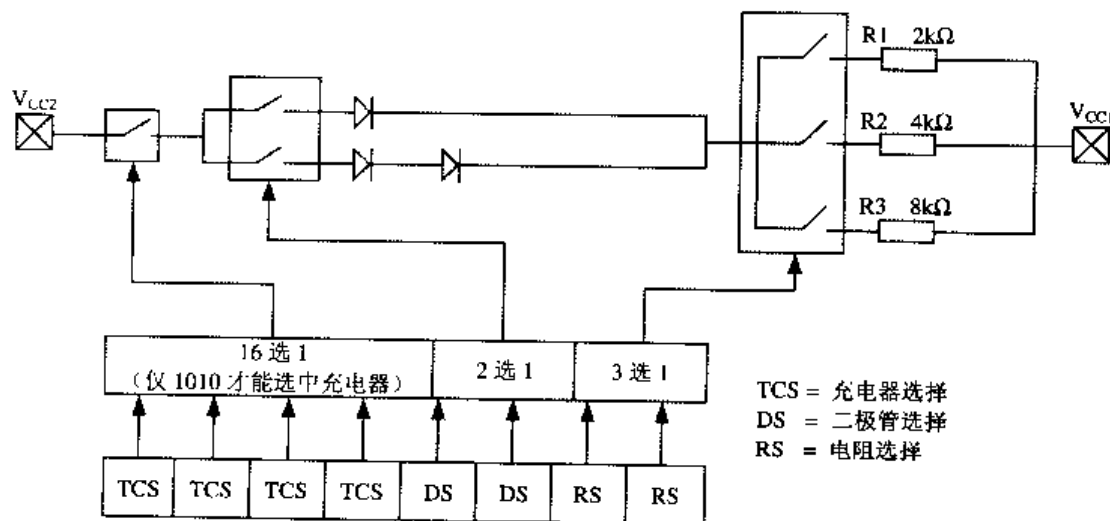


图 8.48 DS1302 的涓流充电控制

DS1302 与 8051 之间通过 IO 线传送同步串行数据, SCLK 为串行通信时的位同步时钟, 一个 SCLK 脉冲传送一位数据。每次数据传送时都以字节为单位, 低位在前, 高位在后, 传送一个字节需要 8 个 SCLK 脉冲。数据传送可以单字节方式或多字节突发方式进行。数据单字节方式传送时序如图 8.49 所示, 在 RST=1 期间, 8051 单片机先向 DS1302 发送一个命令字节, 紧接着发送一个字节的数, DS1302 在接收到命令字节后自动将数据写入指定的片内地址或从该地址读取数据。

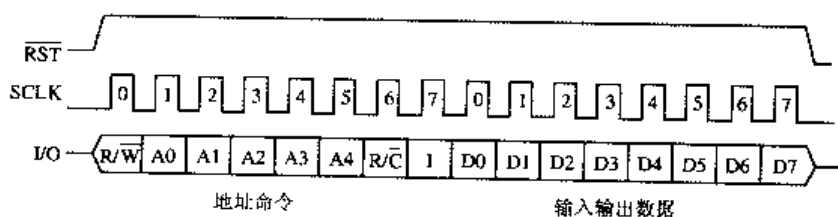


图 8.49 数据单字节方式传送时序

数据多字节突发方式传送时序如图 8.50 所示, RST=1 期间, 若 8051 单片机向 DS1302 发送的命令字节中 A0~A4 全为 1, 则 DS1302 在接收到这个字节命令后可以一次进行 8 个字节日历时钟数据或是 31 个片内 RAM 单元数据的读写操作。

从以上时序可知, 单字节方式传送一次数据需要 16 个 SCLK 脉冲, 多字节方式传送一次数据在对日历时钟进行读写时需要 72 个 SCLK 脉冲, 而在对片内 RAM 单元读写时则最多需要 256 个 SCLK 脉冲。单字节操作方式可保证数据传送时的安全性和可靠性, 多字节操作方式则可提高数据传送速度, 两种方式可视需要灵活选用。

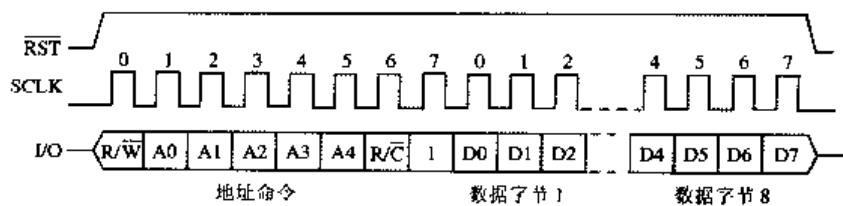


图 8.50 数据多字节突发方式传送时序

下面给出 DS1302 的 Cx51 驱动程序。

```
#include <stdio.h>
#include <reg5000.h>
#define uchar unsigned char
sbit SCLK = P1^3;
sbit IO = P1^4;
sbit RSTB = P1^5;

/***** 函数原型说明 *****/
void clkwritebyte();          /* 时钟字节写入函数 */
void ramwritebyte();          /* RAM 字节写入函数 */
uchar rbyte_3w();             /* 字节读取函数 */
void reset_3w();              /* 复位函数 */
void wbyte_3w(uchar);         /* 字节写入函数 */
void initialize_DS1302();     /* DS1302 初始化函数 */
void disp_clk_regs(uchar);    /* 显示时钟寄存器内容函数 */
void burstramrd();            /* 多字节突发方式读取 RAM 函数 */
void burstramwr();            /* 多字节突发方式写入 RAM 函数 */

/***** 复位函数 *****/
void reset_3w() {
    SCLK = 0;
    RSTB = 0;
    RSTB = 1;
}

/***** 字节写入函数 *****/
void wbyte_3w(uchar W_Byte) {
    uchar i;
    for(i = 0; i < 8; ++i) {
        IO = 0;
        if(W_Byte & 0x01) IO = 1;
        SCLK = 0;
        SCLK = 1;
        W_Byte >>= 1;
    }
}
```

```

/***** 字节读取函数 *****/
uchar rbyte_3w() {
    uchar i;
    uchar R_Byte;
    uchar TmpByte;

    R_Byte = 0x00;
    IO = 1;
    for(i=0; i<8; ++i) {
        SCLK = 1;
        SCLK = 0;
        TmpByte = (uchar)IO;
        TmpByte <<= 7;
        R_Byte >>= 1;
        R_Byte |= TmpByte;
    }
    return R_Byte;
}

/***** 时钟字节写入函数 *****/
void clkwritebyte() {
    uchar ClkAdd;
    uchar ClkData;

    printf("\nWrite Clock ADDRESS:");
    scanf("%bu", &ClkAdd);          /* 输入时钟地址 */
    printf("\nWrite Clock DATA:");
    scanf("%bx", &ClkData);         /* 输入时钟数据 */
    ClkAdd = ((ClkAdd * 2) | 0x80);   /* 时钟数据写入命令 */
    reset_3w();
    wbyte_3w(ClkAdd);
    wbyte_3w(ClkData);
    reset_3w();
}

/***** RAM 字节写入函数 *****/
void ramwritebyte() {
    uchar ramadd;
    uchar ramdata;

    printf("\nWrite Ram ADDRESS (HEX):");
    scanf("%bx", &ramadd);          /* 输入 RAM 地址 */

```

```

    printf("\nWrite Ram DATA (HEX):");
    scanf("%bx", &ramdata);          /* 输入 RAM 数据 */
    ramadd = ((ramadd * 2) | 0xC0);    /* RAM 数据写入命令 */
    reset_3w();
    wbyte_3w(ramadd);
    wbyte_3w(ramdata);
    reset_3w();
}

/***** 显示时钟寄存器内容函数 *****/
void disp_clk_regs(uchar loop) {
    uchar lsec = 99, sec, min, hrs, dte, mon, day, yr;

    do {                                /* 读取并显示时钟寄存器的内容 */
        reset_3w();
        wbyte_3w(0xBF);                /* 以多字节突发方式读取时钟数据 */
        sec = rbyte_3w();               /* 秒 */
        min = rbyte_3w();               /* 分 */
        hrs = rbyte_3w();               /* 小时 */
        dte = rbyte_3w();               /* 日期 */
        mon = rbyte_3w();               /* 月份 */
        day = rbyte_3w();               /* 星期 */
        yr = rbyte_3w();                /* 年 */
        reset_3w();
        if(sec != lsec || !loop) {      /* 每秒显示一次 */
            printf("\nYr  Day  Mon  Dte  Hrs  Min  Sec");
            printf("\n%2.bX  %2.bX  %2.bX  %2.bX", yr, day, mon, dte);
            printf("  %2.bX  %2.bX  %2.bX", hrs, min, sec);
            lsec = sec;
        }
        if(!loop) break;
    } while(!RI);
    if(loop) _getkey();
}

/***** 多字节突发方式读取 RAM 函数 *****/
void burstramrd() {
    uchar rmary[31];
    uchar i;

    reset_3w();
    wbyte_3w(0xFF);                    /* 以多字节突发方式读取 RAM */
    for (i=0; i<31; ++i) {

```

```

        rmary[i] = rbyte_3w();
    }
    reset_3w();

    printf("\nDS1302 Ram");
    printf("\n%2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX",
           rmary[0], rmary[1], rmary[2], rmary[3],
           rmary[4], rmary[5], rmary[6], rmary[7]);
    printf("\n%2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX",
           rmary[8], rmary[9], rmary[10], rmary[11],
           rmary[12], rmary[13], rmary[14], rmary[15]);
    printf("\n%2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX",
           rmary[16], rmary[17], rmary[18], rmary[19],
           rmary[20], rmary[21], rmary[22], rmary[23]);
    printf("\n%2.bX %2.bX %2.bX %2.bX %2.bX %2.bX %2.bX",
           rmary[24], rmary[25], rmary[26], rmary[27],
           rmary[28], rmary[29], rmary[30]);
}

/***** 多字节突发方式写入 RAM 函数 *****/
void burstramwr() {
    uchar ramdata;
    uchar i;

    printf("\nWrite Ram DATA (HEX):");
    scanf("%bx", &ramdata);          /* 输入 RAM 数据 */

    reset_3w();
    wbyte_3w(0xfe);                  /* 以多字节突发方式写入 RAM */
    for (i=0; i<31; ++i) {
        wbyte_3w(ramdata);
    }
    reset_3w();
}

/***** DS1302 初始化函数 *****/
void initialize_DS1302() {
    uchar yr, mn, date, dy, hr, min, sec;

    printf("\nEnter the year (0-99): ");
    scanf("%bx", &yr);
    printf("\nEnter the month (1-12): ");
    scanf("%bx", &mn);

```

```

printf("\nEnter the date (1-31): ");
scanf("%bx", &date);
printf("\nEnter the day (1-7): ");
scanf("%bx", &dy);
printf("\nEnter the hour (1-24): ");
scanf("%bx", &hr);
hr = hr & 0x3f;          /* 设置时钟为 24 小时方式 */
printf("\nEnter the minute (0-59): ");
scanf("%bx", &min);
printf("\nEnter the second (0-59): ");
scanf("%bx", &sec);

reset_3w();
wbyte_3w(0x8e);          /* 写保护控制寄存器 */
wbyte_3w(0);             /* 允许写入 */
reset_3w();
wbyte_3w(0x90);          /* 涓流充电控制寄存器 */
wbyte_3w(0xab);          /* 允许充电, 双二极管, 8K 电阻 */
reset_3w();
wbyte_3w(0xbe);          /* 以多字节突发方式写入时钟数据(8 个字节) */
wbyte_3w(sec);
wbyte_3w(min);
wbyte_3w(hr);
wbyte_3w(date);
wbyte_3w(mn);
wbyte_3w(dy);
wbyte_3w(yr);
wbyte_3w(0);             /* 以多字节突发方式写入时钟数据时 */
                        /* 必须对写保护控制寄存器写入 0 值 */

reset_3w();
}

/***** 主函数 *****/
main (void) {
    uchar M, M1;
    SCON = 0x50;          /* 设置串行口: 方式 1, 8 位 UART, 允许接收 */
    TMOD |= 0x20;         /* 设置定时器 T1, 方式 2, 8 位自动重装 */
    TH1 = 221;            /* 设置串行口波特率: 1200 @ 16MHz */
    TR1 = 1;
    TI = 1;

    while (1) {
        printf("*****C51 PROGRAM FOR DS1302*****\n\n");
    }
}

```

```
printf("I. initialize DS1302\n");
printf("CW. Write Byte  CR. Read Time\n");
printf("RW. Write RAM  RR. Read RAM\n");
printf("\nEnter Menu Selection:");

M = _getkey();
switch(M) {
    case 'C':
    case 'c':
        printf("\nEnter Clock Routine to run:C");

        M1 = _getkey();
        switch(M1) {
            case 'R':
            case 'r':  disp_clk_regs(1);  break;
            case 'W':
            case 'w':  clkwritebyte(); break;
        }
        break;

    case 'I':
    case 'i':  initialize_DS1302();break;

    case 'R':
    case 'r':
        printf("\nEnter Ram Routine to run:R");

        M1 = _getkey();
        switch(M1) {
            case 'B':
            case 'b':  ramwritebyte(); break;
            case 'R':
            case 'r':  burstramrd();  break;
            case 'W':
            case 'w':  burstramwr();  break;
        }
        break;
    }
}
```

8.7 I²C 总线驱动程序

8.7.1 I²C 总线简介

I²C 总线是一种简单、双向二线制同步串行总线，它只需要两根线（串行时钟线和串行数据线）即可在连接于总线上的器件之间传送信息。这种总线的主要特性如下。

- 总线只有两根线：串行时钟线和串行数据线。
- 每个连到总线上的器件都可由软件以惟一的地址寻址，并建立简单的主/从关系，
- 主器件既可作为发送器，也可作为接收器。
- 它是一个真正的多主总线，带有竞争检测和仲裁电路，可使多个主机任意同时发送数据而不破坏总线上的数据信息。
- 同步时钟允许器件通过总线以不同的波特率进行通信。
- 同步时钟可以作为停止和重新启动串行口发送的握手方式。
- 连接到同一总线上的集成电路器件数只受 400PF 的最大总线电容的限制。

I²C 总线极大地方便了系统设计者，无须设计总线接口，因为总线接口已经集成在片内了，从而使设计时间大为缩短，并且从系统中移去或增加集成电路芯片对总线上的其他集成电路芯片没有影响。I²C 总线的简单结构便于产品改型或升级，改型或升级时只须从总线上取消或增加相应的集成电路芯片即可。目前具有 I²C 总线的 8051 系列单片机有 8XC550、8XC552、8XC652、8XC654、8XC751、8XC752 等，以及包括 LED 驱动器、LCD 驱动器、A/D、D/A 转换器、RAM、EPROM 及 I/O 接口等在内的上百种 I²C 接口电路芯片供应市场。对于原来没有 I²C 总线的单片机如 8031 等，可以使用 I²C 总线接口扩展器件 PCD8584 扩展出 I²C 总线接口，或者采用软件程序模拟 I²C 总线的时序来完成接口功能。

I²C 总线接口的电气结构如图 8.51 所示，组成 I²C 总线的串行数据线 SDA 和串行时钟线 SCL 必须经过上拉电阻 R_p 接到正电源上，连接到总线上的器件的输出级必须为“开漏”或“开集”的形式，以便完成“线与”的功能。

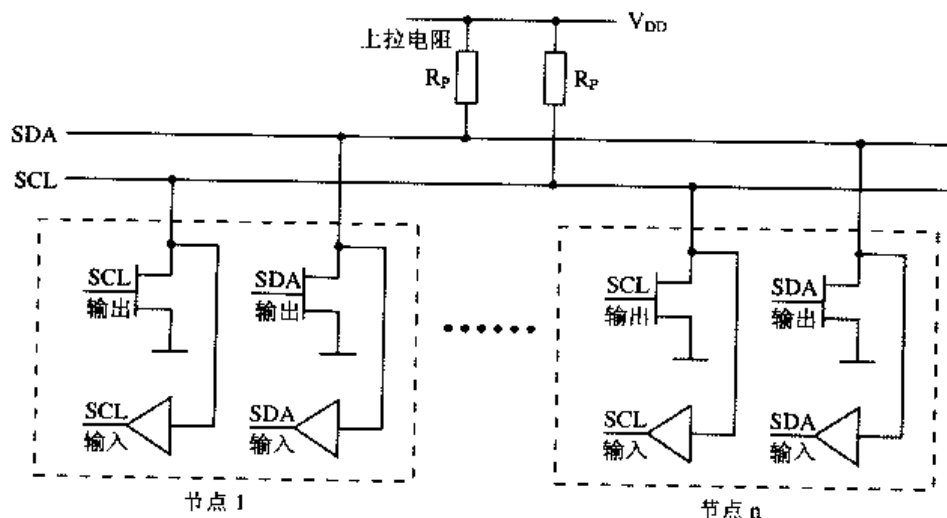


图 8.51 I²C 总线接口的电气结构

I^2C 总线上可以实现多主双向同步数据传送, 所有主器件都可发出同步时钟, 但由于 SCL 接口的“线与”结构, 一旦一个主器件时钟跳变为低电平, 将使 SCL 线保持为低电平直至时钟达到高电平, 因此 SCL 线上时钟低电平时间由各器件中时钟最长的低电平时间决定, 而时钟高电平时间则由高电平时间最短的器件决定。为了使多主数据传送能够正确实现, I^2C 总线中带有竞争检测和仲裁电路。总线竞争的仲裁及处理由内部硬件电路来完成。当两个主器件发送数据相同时不会出现总线竞争; 当两个主器件发送数据不同时才出现总线竞争。其竞争过程如图 8.52 所示。当某一时刻主器件 1 发送高电平而主器件 2 发送低电平, 此时由于 SDA 的“线与”作用, 主器件 1 发送的高电平在 SDA 线上反映的是主器件 2 的低电平状态, 这个低电平状态通过硬件系统反馈到数据寄存器中, 与原有状态比较不同而退出竞争。

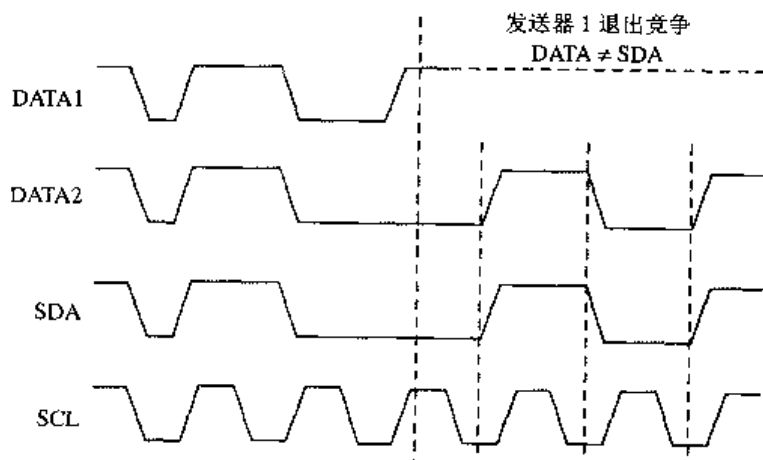


图 8.52 总线竞争的仲裁过程

I^2C 总线可以构成多主数据传送系统, 但只有带 CPU 的器件可以成为主器件。主器件发送时钟、启动位、数据工作方式, 从器件则接收时钟及数据工作方式。接收或发送则根据数据的传送方向决定。 I^2C 总线上数据传送时的启动、结束和有效状态都由 SDA、SCL 的电平状态决定, 在 I^2C 总线规约中启动和停止条件规定如下:

启动条件: 在 SCL 为高电平时, SDA 出现一个下降沿则启动 I^2C 总线。

停止条件: 在 SCL 为高电平时, SDA 出现一个上升沿则停止使用 I^2C 总线。

除了启动和停止状态, 在其余状态下, SCL 的高电平都对应于 SDA 的稳定数据状态。

每一个被传送的数据位由 SDA 线上的高、低电平表示, 对于每一个被传送的数据位都在 SCL 线上产生一个时钟脉冲。在时钟脉冲为高电平期间, SDA 线上的数据必须稳定, 否则被认为是控制信号。SDA 只能在时钟脉冲 SCL 为低电平期间改变。启动条件后总线为“忙”, 在结束信号过后的一定时间总线被认为是“空闲”的。在启动和停止条件之间可转送的数据不受限制, 但每个字节必须为 8 位。首先传送最高位, 采用串行传送方式, 但在每个字节之后必须跟一个响应位。主器件收发每个字节后产生一个时钟应答脉冲, 在这期间, 发送器必须保证 SDA 为高, 由接收器将 SDA 拉低, 称为应答信号 (ACK)。主器件为接收器时, 在接收了最后一个字节之后不发应答信号, 也称为非应答信号 (NOT ACK)。当从器件不能再接收另外的字节时也会出现这种情况。 I^2C 总线的数据传送格式如图 8.53 所示。

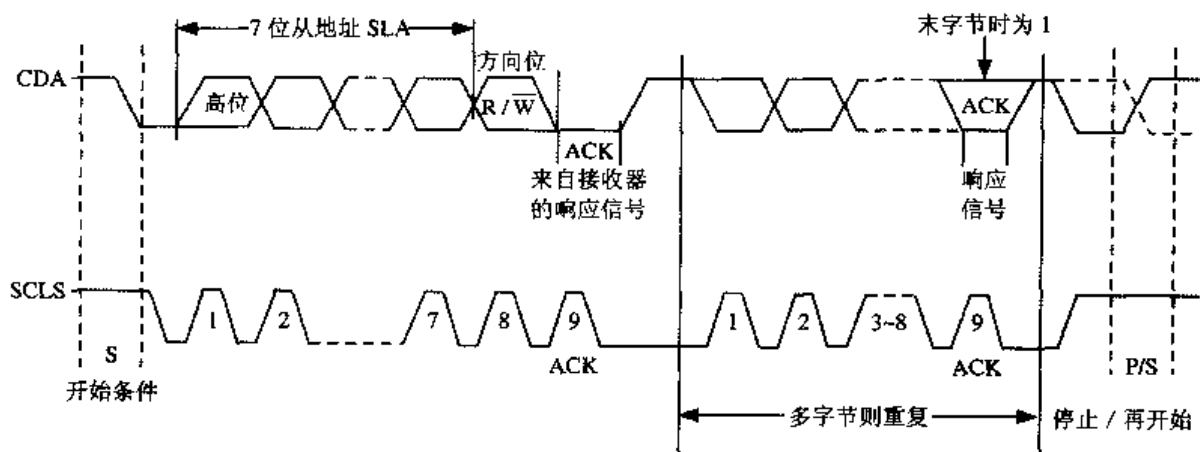


图 8.53 I²C 总线上的数据传送格式

I²C 总线中每个器件都有自己惟一确定的地址，启动条件后主机发送的第一个字节就是被读写的从器件的地址，其中第 8 位为方向位，“0”（W）表示主器件发送，“1”（R）表示主器件接收。总线上每个器件在启动条件后都把自己的地址与前 7 位相比较，如相同则器件被选中，产生应答，并根据读写位决定在数据传送中是接收还是发送。无论是主发、主收还是从发、从收都是由主器件控制。在主发送方式下，由主器件先发出启动信号（S），接着发从器件的 7 位地址（SLA）和表明主器件发送的方向位 0（W），即这个字节为 SLA+W。被寻址的从器件在收到这个字节后，返回一个应答信号（A），在确定主从握手应答正常后，主器件向从器件发送字节数据，从器件每收到一个字节数据后都要返回一个应答信号，直到全部数据都发送完为止。在主接收方式下，主器件先发出启动信号（S），接着发从器件的 7 位地址（SLA）和表明主器件接收的方向位 1（R），即这个字节为 SLA+R。在发送完这个字节后，P1.6（SCL）继续输出时钟，通过 P1.7（SDA）接收从器件发来的串行数据。主器件每接收到一个字节后都要发送一个应答信号（A）。当全部数据都发送或接收完毕后，主器件应发出停止信号（P）。图 8.54 所示为主器件发送和接收数据的过程。

关于 I²C 总线的详细操作请参阅有关参考资料。

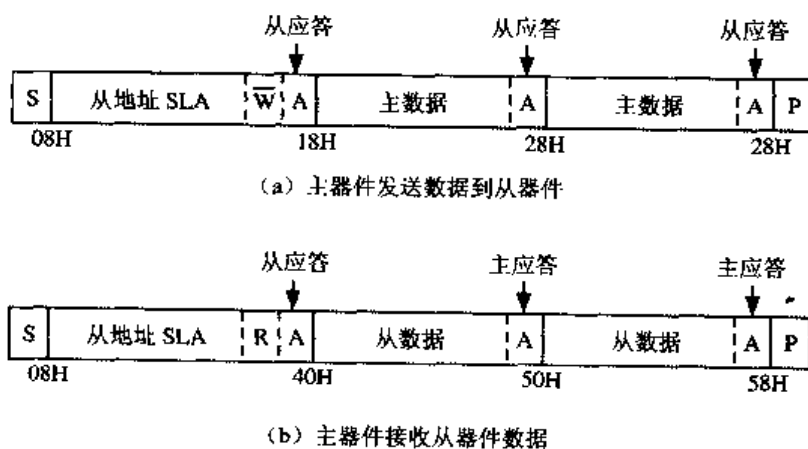


图 8.54 主器件发送和接收数据的过程

8.7.2 I²C 总线通用软件模拟驱动程序

对于没有内部硬件 I²C 总线接口的 8051 系列单片机, 可以采用软件模拟的方法实现 I²C 总线接口功能, 下面给出一个采用 Cx51 编写的通用 I²C 总线模拟驱动程序, 它可用于没有内部 I²C 硬件的 8051 单片机与 I²C 总线器件的接口。在程序开始处定义了 8051 单片机的 P1.6 和 P1.7 作为 I²C 总线的 SCL 和 SDA 信号, 这是为了与具有 I²C 总线的 8051 单片机引脚兼容, 实际上用户可以定义其他 I/O 口引脚作为 SCL 和 SDA 信号, 程序中包括如下 I²C 功能函数:

I_init(), 初始化。

delay(), 延时。

I_clock(), SCL 时钟信号。

I_start(), 起始信号。

I_stop(), 结束信号。

I_send(), 数据发送。

I_Ack(), 应答信号。

实现 I²C 总线基本操作的 C51 驱动程序文件 I2C.C 代码如下:

```

/*****
*
*               I2C 总线基本操作 Cx51 驱动函数
*
*****/
/* 全局符号定义 */
#define HIGH 1
#define LOW 0
#define FALSE 0
#define TRUE ~FALSE
#define uchar unsigned char

sbit SCL      = 0x96 ;
sbit SDA      = 0x97 ;

/*****
* 函数原型: void delay(void);
* 功    能: 本函数实际上只有一条返回指令, 在具体应用中可视具体要求增加延
*          时指令。
*****/
void delay( void ) {
    ;
}

/*****

```

* 函数原型: void I_start(void);

* 功 能: 提供 I²C 总线工作时序中的起始位。

*****/

```
void I_start( void ) {
```

```
    SCL = HIGH ;
```

```
    delay() ;
```

```
    SDA = LOW ;
```

```
    delay() ;
```

```
    SCL = LOW ;
```

```
    delay() ;
```

```
}
```

* 函数原型: void I_stop(void);

* 功 能: 提供 I²C 总线工作时序中的停止位。

*****/

```
void I_stop( void ) {
```

```
    SDA = LOW ;
```

```
    delay() ;
```

```
    SCL = HIGH ;
```

```
    delay() ;
```

```
    SDA = HIGH ;
```

```
    delay() ;
```

```
    SCL = LOW ;
```

```
    delay() ;
```

```
}
```

* 函数原型: void I_init(void);

* 功 能: I²C 总线初始化。在 main() 函数中应首先调用本函数, 然后再调用

* 其他函数。

*****/

```
void I_init( void ) {
```

```
    SCL = LOW ;
```

```
    I_stop() ;
```

```
}
```

* 函数原型: bit I_clock(void);

* 功 能: 提供 I²C 总线的时钟信号, 并返回在时钟电平为高期间 SDA 信号线上状

* 态。本函数可用于数据发送, 也可用于数据接收。

*****/

```
bit I_clock( void ) {
```

```

    bit sample ;
    SCL = HIGH ;
    delay() ;
    sample = SDA ;
    SCL = LOW ;
    delay() ;
    return ( sample ) ;
}

/*****
* 函数原型: bit I_send(uchar I_data);
* 功    能: 向 I2C 总线发送 8 位数据, 并请求一个应答信号 ACK。如果收到 ACK 应答
*           则返回 1(TRUE), 否则返回 0(FALSE)。
*****/
bit I_send(uchar I_data ) {
    uchar I ;
    /* 发送 8 位数据 */
    for ( I=0 ; I<8 ; I++ ) {
        SDA = (bit)( I_data & 0x80 ) ;
        I_data = I_data << 1 ;
        I_clock() ;
    }
    /* 请求应答信号 ACK */
    SDA = HIGH ;
    return ( ~I_clock() ) ;
}

/*****
* 函数原型: uchar I_receive(void);
* 功    能: 从 I2C 总线上接收 8 位数据信号, 并将接收到 8 位数据作为一个字节
*           返回, 不回送应答信号 ACK。主函数在调用本函数之前应保证 SDA 信
*           号线处于浮置状态, 即使 8051 的 P1.7 脚置 1。
*****/
uchar I_receive( void ) {
    uchar I_data = 0 ;
    uchar I ;
    for ( I=0 ; I<8 ; I++ ) {
        I_data *= 2 ;
        if ( I_clock() )
            I_data++ ;
    }
    return ( I_data ) ;
}

```

```

/*****
* 函数原型: void I_Ack(void);
* 功    能: 向 I2C 总线发送一个应答信号 ACK, 一般用于连续数据读取时。
*****/
void I_Ack( void ) {
    SDA = LOW ;
    I_clock() ;
    SDA = HIGH ;
}

```

8.7.3 I²C 接口器件 24C04 的读写程序

上面给出了 I²C 总线基本操作函数, 下面给出一个应用以上基本操作函数实现对 I²C 总线接口器件 24C04 进行读写的 C51 应用实例。24C04 是一种 I²C 接口 EEPROM 器件, 它具有 512×8 位的存储容量, 工作于从器件方式, 每个字节可擦/写 100 万次, 数据保存时间大于 40 年。写入时具有自动擦除功能, 具有页写入功能, 可一次写入 16 个字节。24C04 芯片采用 8 脚 DIP 封装, 具有 V_{cc}、V_{ss} 电源引脚, SCL、SDA 通讯引脚, A0、A1、A2 地址引脚和 WP 写保护引脚。WP 脚接 V_{cc} 时, 禁止写入高位地址 (100H~1FFH), WP 脚接 V_{ss} 时, 允许写入任何地址。A1 和 A2 决定芯片的从机地址, 可接 V_{cc} 或 V_{ss}, A0 不用, 应接 V_{cc} 或 V_{ss}。图 8.55 所示为 24C04 于 8051 单片机的一种接口。

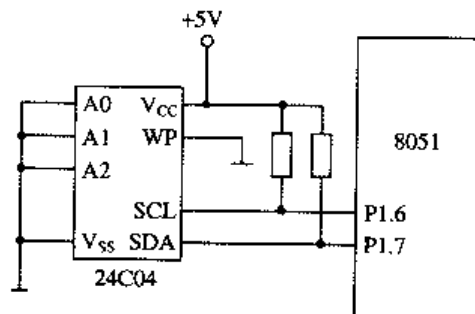


图 8.55 单片机 8051 与 24C04 的接口

8051 单片机与 24C04 之间进行数据传递时, 首先传送器件的从地址 SLA, 格式如下:

START	1	0	1	0	A2	A1	BA	R/W	ACK
-------	---	---	---	---	----	----	----	-----	-----

START 为起始信号, 1010 为 24C04 器件地址, A2 和 A1 由芯片的 A2、A1 引脚上的电平决定, 这样可最多接入 4 片 24C04 芯片, BA 为块地址 (每块 256 字节), R/W 决定是写入 (0) 还是读出 (1), ACK 为 24C04 给出的应答信号。在对 24C04 进行写入时, 应先发出从机地址字节 SLAW (R/W 为 0), 再发出字节地址 WORDADR 和写入的数据 data (可为 1~16 个字节), 写入结束后应发出停止信号。

通常对 EEPROM 器件写入时总需要一定的写入时间 (5~10ms), 因此在写入程序中无法连续写入多个数据字节。为了解决连续写入多个数据字节的问题, EEPROM 器件中常设

有一定容量的页写入数据寄存器。用户一次写入 EEPROM 的数据字节不大于页写入字节数时,可按通常 RAM 的写入速度,将数据装入 EEPROM 的数据寄存器中,随后启动自动写入定时控制逻辑,经过 5~10ms 的时间,自动将数据寄存器中的数据同步写入 EEPROM 的指定单元。这样一来,只要一次写入的字节数不多于页写入容量,总线对 EEPROM 的操作可视为对静态 RAM 的操作,但要求下次数据写入操作在 5~10ms 之后进行。24C04 的页写入字节数为 16。对 24C04 进行页写入是指向其片内指定首地址 (WORDADR) 连续写入不多于 n 个字节数据的操作。 n 为页写入字节数, m 为写入字节数, $m \leq n$ 。页写入数据操作格式如下:

S	SLAW	A	WORDADR	A	data1	A	data2	A	...	datam	A	P
---	------	---	---------	---	-------	---	-------	---	-----	-------	---	---

这种数据写入操作实际上就是 $m+1$ 个字节的 I^2C 总线进行主发送的数据操作。

对 24C04 写入数据时也可以按字节方式进行,即每次向其片内指定单元写入一个字节的数,这种写入方式的可靠性高。字节写入数据操作格式如下:

S	SLAW	A	WORDADR	A	data	A	P
---	------	---	---------	---	------	---	---

24C04 的读操作与通常的 SRAM 相同,但每读一个字节地址将自动加 1。24C04 有三种读操作方式,即现行地址读、指定地址读和序列读。现行地址读是指不给定片内地址的读操作,读出的是现行地址中的数据。现行地址是片内地址寄存器当前的内容,每完成一个字节的读操作,地址自动加 1,故现行地址是上次操作完成后的下一个地址。现行地址读操作时,应先发出从机地址字节 SLAR (R/W 为 1),接收到应答信号 (ACK) 后即开始接收来自 24C04 的数据字节,每接收到一个字节的数都必须发出一个应答信号 (ACK)。现行地址读的数据操作格式如下:

S	SLAR	A	data	A	P
---	------	---	------	---	---

指定地址读是指按指定的片内地址读出一个字节数据的操作。由于要写入片内指定地址,故应先发出从机地址字节 SLAW (R/W 为 0),再进行一个片内字节地址的写入操作,然后发出重复起始信号和从机地址 SLAR (R/W 为 1),开始接收来自 24C04 的数据字节。数据操作格式如下:

S	SLAW	A	WORDADR	A	S	SLAR	A	data	A	P
---	------	---	---------	---	---	------	---	------	---	---

序列读操作是指连续读入 m 个字节数据的操作。序列读入字节的首地址可以是现行地址或指定地址,其数据操作可以在上述两种操作的 SLAR 发送之后进行。数据操作格式如下:

S	SLAR	A	data1	A	data2	...	datam	A	P
---	------	---	-------	---	-------	-----	-------	---	---

实现对 24C04 进行读写的 C51 驱动程序中包含如下功能函数:

- E_address() 写入器件从地址和片内字节地址。
E_read_block() 从 24C04 中读出指定个字节 (BLOCK_SIZE=32) 的数据并送入外部数据存储器单元,采用的是序列读操作方式。

E_write_block() 将外部数据存储器中的数据内容写入从 24C04 首地址开始的指定个字节 (BLOCK_SIZE=32), 采用的是字节写入操作方式。如果希望采用页写入操作方式, 可对该函数作适当的修改。

Wait_5ms() 为保证写入正确而设置的 5ms 延时。

另外, 需要将前面介绍的 I²C 基本功能函数文件作为一个项目文件, 同时要采用一个头文件 “I2C.H” 将前面介绍的 I²C 总线基本操作函数包含到主程序文件中来。

主程序文件 main.c 代码如下:

```

/*****
*
*          实现对 24C04 进行读写的 Cx51 驱动程序
*****/
#include <reg51.h>
#include <stdio.h>
#include <i2c.h>

#define WRITE 0xA0          /* 定义 24C04 的器件地址 SLA 和方向位 W */
#define READ 0xA1          /* 定义 24C04 的器件地址 SLA 和方向位 R */
#define BLOCK_SIZE 32      /* 定义指定字节个数 */
#define uchar unsigned char

xdata uchar EAROMImage[BLOCK_SIZE];    /* 在外部 RAM 中定义存储映像单元 */

/*****
* 函数原型: bit E_address(uchar Address);
* 功    能: 向 24C04 写入器件地址和一个指定的字节地址。
*****/
bit E_address(uchar Address ) {
    I_start();
    if ( I_send( WRITE ) )
        return ( I_send( Address ) );
    else
        return ( FALSE );
}

/*****
* 函数原型: bit E_read_block(void);
* 功    能: 从 24C04 中读取 BLOCK_SIZE 个字节的数据并转存于外部 RAM 存储映像
*          单元, 采用序列读操作方式从片内 0 地址开始连续读取数据。如果
*          24C04 不接受指定的地址则返回 0 (FALSE)。
*****/
bit E_read_block( void ) {
    uchar I ;

```

```

/* 从地址 0 开始读取数据 */
if ( E_address( 0 ) ) {
    /* 发送重复启动信号 */
    I_start() ;
    if ( I_send( READ ) ) {
        for ( I=0 ; I<=BLOCK_SIZE ; I++ ) {
            EAROMImage[I] = ( I_receive() ) ;
            if ( I != BLOCK_SIZE )
                I_Ack() ;
            else {
                I_clock() ;
                I_stop() ;
            }
        }
        return ( TRUE ) ;
    }
    else {
        I_stop() ;
        return ( FALSE ) ;
    }
}
else
    I_stop() ;
return ( FALSE ) ;
}

/*****
* 函数原型: void wait_5ms(void);
* 功    能: 提供 5ms 延时(时钟频率为 12MHz)。
*****/
void wait_5ms( void ) {
    int I ;
    for ( I=0 ; I<1000 ; I++ ) {
        ;
    }
}

/*****
* 函数原型: bit E_write_block(void);
* 功    能: 将外部 RAM 存储映像单元中的数据写入到 24C04 的头 BLOCK_SIZE 个字节。
*           采用字节写操作方式, 每次写入时都需要指定片内地址。如果 24C04
*           不接受指定的地址或某个传送的字节未收到应答信号 ACK, 则返回 0
*           否则返回 1。
*****/

```



```

*****/
bit E_write_block( void ) {
    uchar I ;
    for ( I=0 ; I<=BLOCK_SIZE ; I++ ) {
        if ( E_address(i) && I_send( EAROMImage[I] ) ) {
            I_stop() ;
            wait_5ms();
        }
        else
            return ( FALSE ) ;
    }
    return ( TRUE ) ;
}

/***** 主函数 *****/
void main() {
    SCON = 0x5a;
    TMOD = 0x20;
    TCON = 0x69;
    TH1 = 0xfd;
    I_init(); /* I2C 总线初始化 */
    if (E_write_block()) /* 写入 24C04 */
        printf("write I2C good.\r\n"); /* 输出写入成功提示信息 */
    else
        printf("write I2C bad.\r\n"); /* 输出写入失败提示信息 */
    if (E_read_block()) /* 读出 24C04 */
        printf("read I2C good.\r\n"); /* 输出读出成功提示信息 */
    else
        printf("read I2C bad.\r\n"); /* 输出读出失败提示信息 */
    while(1);
}

```

头文件 I2C.H 代码如下:

```

#define uchar unsigned char
#define uint unsigned int

void delay( void );
void I_stop( void );
void I_init( void );
void I_start( void );
bit I_clock( void );
void I_Ack( void );

```

```
bit I_send( uchar I_data );
uchar I_receive( void );
```

8.8 8051 单片机存储器扩展与测试应用编程

8.8.1 测试 8051 应用系统总线与扩展存储器的 Cx51 程序

在很多单片机应用系统中都会涉及存储器或外围设备的扩展，外扩存储器或设备的使用可靠性很大程度上取决于系统总线是否稳定，本节给出一个对 8051 单片机应用系统数据总线、地址总线以及外扩存储器本身能否正常工作的一个 Cx51 测试程序。数据总线的测试采取对某一固定地址存储器区域写入“步进的”1 来完成，地址总线的测试采取对某一段存储器地址范围的地址相关位写入“移动的”1，并检查是否发生混淆，测试中可以发现单个位地址的错误，譬如“粘滞高、粘滞低”（stuck-high、stuck-low）以及引脚间短路现象。为了获得最好的测试结果，测试基地址的低位部分（LSB）应有足够多的 0，以保证单个位地址能够正确变化，例如，要测试 64KB 范围的地址总线，测试基地址应选择在 64KB 地址范围的边界上，并且应尽可能使被测试的地址范围为 2 的整数次幂。

测试模块 memtest.c 文件代码如下：

```
#include <reg51.h>
#include <stdio.h>

typedef unsigned char datum;

/***** 数据总线测试函数 *****/
* 函数原型: datum memTestDataBus(volatile datum xdata * address)。
* 功 能: 对某一固定地址存储器区域写入“步进的”1，存储器地址由
* 主调函数给定。测试成功返回 0，测试失败返回写入值。
*****/
datum memTestDataBus(volatile datum xdata * address) {
    datum pattern;
    for (pattern = 1; pattern != 0; pattern <<= 1) {
        /* Write the test pattern */
        *address = pattern;          /* 写入测试值 */
        /* Read it back (immediately is okay for this test). */
        if (*address != pattern) {   /* 如果读取值与写入值不相等 */
            return (pattern);        /* 测试失败返回写入值 */
        }
    }
    return (0);                     /* 测试成功返回 0 */
}
```

```

/***** 地址总线测试函数 *****/
* 函数原型: datum xdata *memTestAddressBus(volatile datum xdata
*                                     baseAddress, unsigned long nBytes);
* 功    能: 对某一段存储器地址范围的地址相关位写入“步进的”1, 并检查是否
*           发生混淆, 测试中将发现单个位地址的错误, 譬如“粘滞高、粘滞低”
*           (stuck-high、stuck-low)以及引脚间短路现象。测试基地址以及测试
*           范围由主调函数给定。测试成功返回 NULL, 测试失败返回发生混淆
*           错误的首地址。
* 注:      为了获得最好的测试结果, 测试基地址低位部分(LSB)应有足够多的
*           0 以保证单个位地址能够正确变化, 例如要测试 64KB 范围的地址总线, 测
*           试基地址应选择在 64KB 地址范围的边界上, 并且应尽可能使被测试的地址
*           范围为 2 的整数次幂。通过检查存储器内容可以找出其他错误信息。
*****/
datum xdata *memTestAddressBus(volatile datum xdata * baseAddress,
                                unsigned long nBytes) {
    unsigned long addressMask = (nBytes - 1);
    unsigned long offset;
    unsigned long testOffset;

    datum pattern      = (datum) 0xAAAAAAAA;
    datum antipattern = (datum) 0x55555555;

    /* 对每个 2 的整数次幂的偏移地址写入默认测试数据 */
    for (offset=sizeof(datum); (offset & addressMask)!=0; offset<<= 1) {
        baseAddress[offset] = pattern;
    }

    /* 检查地址位是否粘滞高 */
    testOffset = 0;
    baseAddress[testOffset] = antipattern;

    for (offset=sizeof(datum); (offset & addressMask)!=0; offset <<= 1) {
        if (baseAddress[offset] != pattern) {
            return ((datum xdata *) &baseAddress[offset]);
        }
    }

    baseAddress[testOffset] = pattern;

    /* 检查地址位是否粘滞低或发生短路 */
    for (testOffset=sizeof(datum); (testOffset & addressMask)!=0;
        testOffset <<= 1) {
        baseAddress[testOffset] = antipattern;
    }
}

```

```

    for (offset=sizeof(datum); (offset & addressMask)!=0; offset <= 1) {
        if ((baseAddress[offset] != pattern) && (offset != testOffset)) {
            return ((datum xdata *) &baseAddress[testOffset]);
        }
    }
    baseAddress[testOffset] = pattern;
}
return (NULL);
}

```

***** 器件测试函数 *****

* 函数原型: datum xdata *memTestDevice(volatile datum xdata * baseAddress,
* unsigned long nBytes);

* 功 能: 通过对存储器器件的整个地址单元进行加 1 和减 1 来测试其完整性, 每一
* 个存储位都要进行写 0 和写 1 测试。测试基地址以及测试范围由主调函数
* 给定。测试成功返回 NULL, 并且整个存储器区域将用 0 填充。
* 测试失败则返回发生错误的第一个存储器地址。通过检查存储器内容可
* 以找出其他错误信息。

*****/

```

datum xdata *memTestDevice(volatile datum xdata * baseAddress,
                           unsigned long nBytes) {

```

```
    unsigned long offset;
```

```
    unsigned long nWords = nBytes / sizeof(datum);
```

```
    datum pattern;
```

```
    datum antipattern;
```

```
    /* 用已知数据填充存储器单元 */
```

```

    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++) {
        baseAddress[offset] = pattern;
    }

```

```
    /* 检查每个存储器单元并将其取反 */
```

```

    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++) {
        if (baseAddress[offset] != pattern) {
            return ((datum xdata *) &baseAddress[offset]);
        }
    }

```

```
    antipattern = ~pattern;
```

```
    baseAddress[offset] = antipattern;
```

```
    }
```

```
    /* 检查每个存储器单元并将其清零 */
```

```

    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++) {

```

```

    antipattern = ~pattern;
    if (baseAddress[offset] != antipattern) {
        return ((datum xdata *) &baseAddress[offset]);
    }
    baseAddress[offset] = 0;
}
return (NULL);
}

/***** 存储器测试函数 *****/
* 函数原型: memTest();
* 功    能: 测试外部 RAM 存储器数据总线、地址总线以及 RAM 器件。
*          测试成功返回 0, 测试失败返回-1。
*****/
int memTest(void) {
    #define BASE_ADDRESS (volatile datum xdata *) 0x00009000
    #define NUM_BYTES    8 * 1024
    if ((memTestDataBus(BASE_ADDRESS) != 0) ||
        (memTestAddressBus(BASE_ADDRESS, NUM_BYTES) != NULL) ||
        (memTestDevice(BASE_ADDRESS, NUM_BYTES) != NULL))
    {
        return (-1);
    }
    else
    {
        return (0);
    }
}

void main(void) {
    SCON = 0x5a;
    TMOD = 0x20;
    TCON = 0x69;
    TH1 = 0xfd;
    if (!memTest()) printf("test OK\n");
    else printf("test fail\n");
    while(1);
}

```

8.8.2 8051 扩展 FLASH 存储器在系统编程的 Cx51 程序

随着存储器技术发展, FLASH 闪速存储器在单片机应用系统中应用日益广泛。采用 FLASH 存储器后可使系统具有在系统编程 (In-System Programing) 或在应用中编程

(In-Application Programming) 功能, 可以很容易实现按需要随时修改应用程序代码。本节介绍一个采用 Atmel 公司的 FLASH 存储器 AT49HF010 与 AT89C51 单片机接口实现 ISP 功能的例子, 硬件配置如图 8.56 所示。

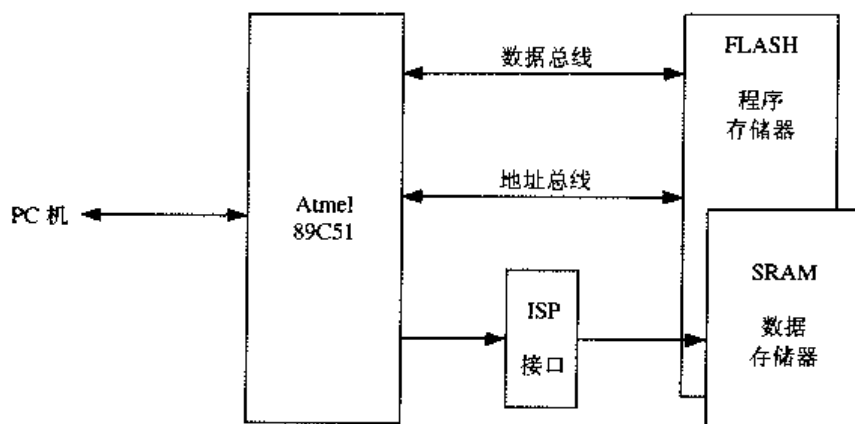


图 8.56 实现 ISP 功能的硬件配置

单片机系统采用 SRAM 作为外部数据存储器, 采用 FLASH 作为外部程序存储器, 系统正常工作时单片机执行外部 FLASH 存储器中的程序代码, 需要进行 ISP 时单片机执行片内 ROM 中的 ISP 程序代码 (即本例程序代码), 同时利用 ISP 控制电路将外部 FLASH 映射到数据存储器空间, 通过单片机的串行口接收来自 PC 机的新代码或数据 (HEX 文件)。

8051 单片机的存储器配置及 ISP 控制逻辑如图 8.57 所示。从 ISP 端输入高电平时 89C51 的 EA 引脚为低电平, CPU 执行外部 FLASH 中的程序代码, 系统正常工作; 从 ISP 端输

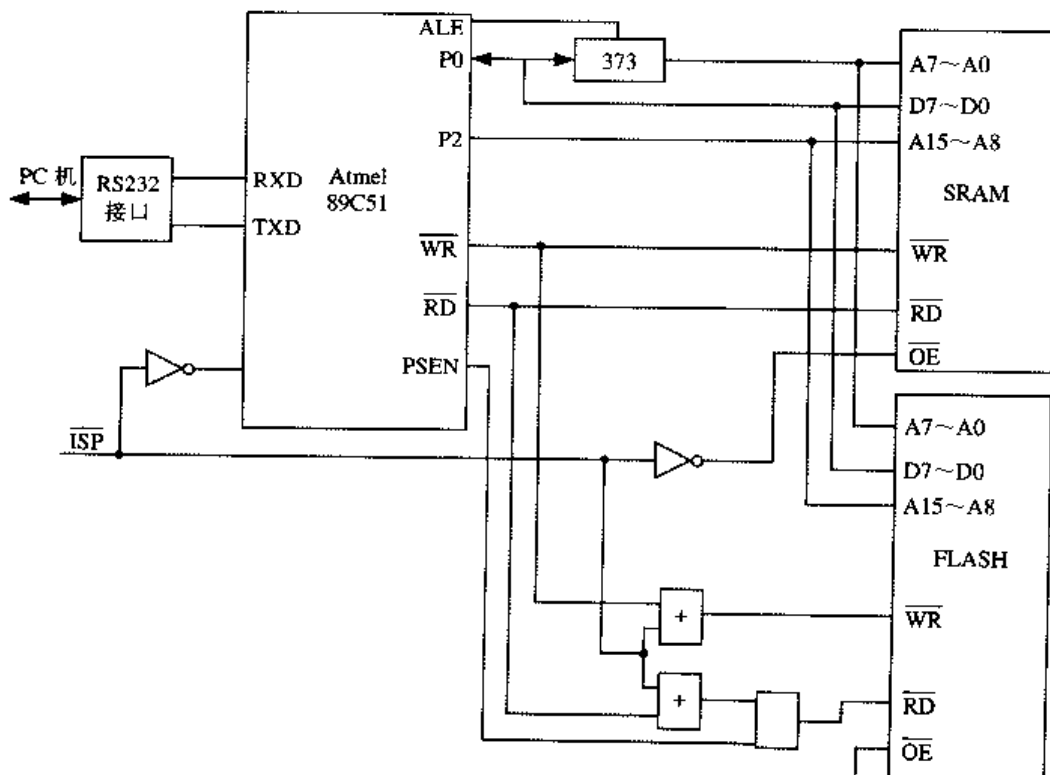


图 8.57 89C51 外部 FLASH 实现 ISP 的存储器配置

入低电平时 89C51 的 EA 引脚为高电平, CPU 执行片内 ROM 中的 ISP 程序代码, 同时外部 SRAM 的 OE 端为高电平, 禁止对其进行操作, 而外部 FLASH 被映射到数据存储器空间 (RD=PSEN & RD), 从而可以通过 MOVX 指令接收来自 PC 机的新代码或数据。这里需要注意的是如果采用带有片内 XRAM 的 8051 单片机 (如 AT89C51RD2 等), 由于这种片内 XRAM 实际上是外部数据空间的部分映射, 因此在进行 ISP 操作时应禁止其片内 SRAM 以便能够对整个外部 FLASH 进行更新, 否则可能有一部分更新代码会被写入到片内 XRAM 中。

本例 ISP 主要完成 FLASH 存储器检测 (AT49HF010)、FLASH 存储器芯片擦除、从 PC 机下载更新 HEX 文件到 FLASH 存储器。整个程序分为 4 个模块文件: 主程序模块 main.c、输入输出接口模块 io.c、HEX 文件处理模块 hex.c, 以及 FLASH 编程模块 flash.c, 另外还包含 3 个头文件: compiler.h、config.h 和 isp.h。

主程序模块 main.c 完成系统初始化并管理从芯片擦除到芯片编程的全部 ISP 工作过程, 文件代码如下:

```
#include <stdio.h>
#include "config.h"
#include "isp.h"
/***** _DEAD_() 函数 *****/
* 功    能: 输出错误提示信息并进入一个死循环, 当发生致命错误时调用本函数。
*****/
void _DEAD_(void) {
    printf("\nError: cannot continue!\n");
    while (1);
}

/***** 系统初始化函数 *****/
* 功    能: 进行系统初始化, 禁止片上 XRAM, 设置串行口工作方式。
*****/
void system_init(void) {
    #ifdef HAVE_XRAM                /* 禁止片上 XRAM */
        AUXR = NO_XDATA;
    #endif

    uart_init();                    /* 设置串行口工作方式 */
    EA = 1;                          /* 开中断 */
}

/***** 主函数 *****/
* 功    能: 提供程序入口, 管理从芯片擦除到芯片编程的全部 ISP 工作过程。
*****/
void main(void) {
    Uchar status;
```

```

system_init();           /* 系统初始化 */
printf("\n\n** Welcome to the ISP program! **\n\n"); /* 输出提示信息 */

/* 检查 FLASH 存储器生产厂商及芯片 ID (Atmel=1F, 49HF010=17) */
printf("Check if FLASH device is a Atmel AT49HF010... ");
if (flash_id()==0x1F17) {
    printf("OK.\n");           /* FLASH 存储器是 AT49HF010, 继续 */
}
else {
    printf("KO!\n");           /* FLASH 存储器不是 AT49HF010 */
    _DEAD_();                 /* 发生致命错误, 进入死循环 */
}

/* 擦除 FLASH 存储器 */
printf("\nErasing the on-board FLASH memory... ");
flash_erase();
printf("OK.\n\n");

/* 擦除完成, 准备从 PC 机下载更新代码, 输出提示信息 */
printf("Ready for FLASH programming.\n");
printf("Send .hex file with the following terminal configuration:\n");
printf(" - ASCII character transmission,\n");
printf(" - 8 bits, 1 stop, parity none,\n");
printf(" - XON-XOFF flow control.\n");
printf("\nWaiting for download...\n");

/* 整理从 PC 机下载的 HEX 文件更新代码, 并将其编程到 FLASH 存储器中 */
uart_rx_enable();
status = hex_parser();
uart_rx_disable();
if (status == HEX_DEC_CSERR) {
    printf("\nChecksum error: external FLASH memory is not programmed!\n");
    _DEAD_();
}
else
    printf("\nExternal FLASH memory is now programmed.\n");
while(1);
}

```

输入输出接口模块 io.c 完成串行口初始化、中断处理以及环形接收缓冲区管理, 文件代码如下:

```

#include "compiler.h"
#include "config.h"

```



```

#include "isp.h"

/***** 宏定义 *****/
#define XON      17      /* 定义 XON / XOFF 控制字符 */
#define XOFF     19
#define RX_BUF_SIZE 16    /* 定义接收缓冲区大小, 必须为 2^y */

/* 定义当接收缓冲区仅剩 50% 时 XOFF 标志 */
#define XOFF_THRESH (RX_BUF_SIZE - (RX_BUF_SIZE / 2))
/* 定义当 25% 接收缓冲区满时 XON 标志 */
#define XON_THRESH  (RX_BUF_SIZE / 4)

/* 定义等待串行口发送完毕标志 */
#define WAIT_EO_TX  {while (TI==0); TI=0;}

/***** 变量定义 *****/
static Uchar rx_buffer[RX_BUF_SIZE]; /* 接收环形缓冲区 */
static Uchar rx_index_wr;             /* 接收环形缓冲区索引 */
static Uchar rx_index_rd;
static Bool  tx_off;                  /* XOFF 状态标志 */
Uchar nb_rx_data;                     /* 接收缓冲区内的数据个数 */

void rx_buffer_wr(Uchar rx_data);

/***** 串行口中断函数 *****/
* 功 能: 仅处理串行口接收中断。
*****/
Interrupt(void uart(void), 4) {
    if (RI == 1) {
        rx_buffer_wr(SBUF); /* 将接收到的数据写入接收缓冲区 */
        RI = 0;
    }
}

/***** 数据发送函数 *****/
* 功 能: 从串行口发送一个字符。
*****/
void uart_tx(Uchar tx_data) {
    SBUF = tx_data;
    WAIT_EO_TX;
}

/***** 串行口初始化函数 *****/

```

* 功 能: 设置串行口工作方式。

```
void uart_init(void) {
    SCON = 0x50;
    TMOD = TMOD | 0x20 ;      /* 设置定时器 T1 为方式 2, 作为波特率发生器 */
    TH1 = 0xFD;               /* 采用 11.059200MHZ 晶振时, 波特率为 9600 */
    TL1 = 0xFD;
    PCON = PCON & 0x80;
    TCON |= 0x40;
    TT=1;
}
```

***** 允许串行口接收函数 *****

* 功 能: 复位环形缓冲区, 初始化 XON/XOFF 协议, 允许串行口中断。

```
void uart_rx_enable(void) {
    rx_index_wr = rx_index_rd = nb_rx_data = 0;
    WAIT_EO_TX;          /* 等待前面发送完成 printf */
    uart_tx(XON);
    ES = 1;
}
```

***** 禁止串行口接收函数 *****

* 功 能: 禁止串行口接收中断。

```
void uart_rx_disable(void) {
    ES = 0;
    TI = 1;              /* 允许串行口发送中断, 以使用 printf 函数进行查询 */
}
```

***** 接收缓冲区检测函数 *****

* 功 能: 检测是否有新接收数据存入到接收缓冲区, 当缓冲区为空时返回 1, 否

* 则返回 0。

```
Bool rx_buffer_empty(void) {
    if (nb_rx_data == 0)
        return TRUE;
    else
        return FALSE;
}
```

***** 缓冲区数据写入函数 *****

* 功 能: 将新接收的数据 Uchar rx_data 写入缓冲区并更新缓冲区索引值。

```

*****/
void rx_buffer_wr(Uchar rx_data) {
    nb_rx_data++;
    rx_buffer[rx_index_wr] = rx_data;
    rx_index_wr=(rx_index_wr+1)%RX_BUF_SIZE; /* 计算环形缓冲区索引值 */
    /* 当接收缓冲区中存入的字符多于 XOFF_THRES 标志时 PC 机停止发送数据 */
    if ((tx_off==FALSE) && (nb_rx_data > XOFF_THRESH)) {
        uart_tx(XOFF);
        tx_off = TRUE;
    }
}

/***** 读取接收缓冲区数据函数 *****/
* 功 能: 读取接收缓冲区中的下一个数据并返回。
*****/
Uchar rx_buffer_rd(void) {
    static Uchar data_cnt=0;
    Uchar rx_data;
    EA=0; /* 关中断, 以避免与 rx_buffer_wr() 函数发生冲突 */
    nb_rx_data--;
    EA=1; /* 开中断 */
    rx_data = rx_buffer[rx_index_rd];
    rx_index_rd=(rx_index_rd+1)%RX_BUF_SIZE; /* 计算环形缓冲区索引值 */
    data_cnt++;
    if (data_cnt == 0) uart_tx('.');
    /* 当接收缓冲区中存入的字符少于 XOFF_THRES 标志时 PC 机继续发送数据 */
    EA = 0;
    if ((tx_off == TRUE) && (nb_rx_data < XON_THRESH)) {
        uart_tx(XON);
        tx_off = FALSE;
    }
    EA = 1;
    return(rx_data);
}

```

HEX 文件处理模块 hex.c 处理从 PC 机接收到的 ACSII 码文件, 从中分离出各种 HEX 文件信息以便于 FLASH 编程, 文件代码如下:

```

#include "config.h"
#include "isp.h"
#include <ctype.h>

/***** 宏定义 *****/
/* 定义 HEX 文件解码符号 */

```

```

#define REC_MARK    0x01
#define REC_LEN_1    0x02
#define REC_LEN_2    0x03
#define OFFSET_1     0x04
#define OFFSET_2     0x05
#define OFFSET_3     0x06
#define OFFSET_4     0x07
#define REC_TYP_1    0x08
#define REC_TYP_2    0x09
#define DATA_1      0x0A
#define DATA_2      0x0B
#define CHEKSUM_1    0x0C
#define CHEKSUM_2    0x0D

```

```
Uchar hex_decoder(Uchar hex_data);
```

```

/***** HEX 文件分析函数 *****/

```

```

* 功    能: 监视 HEX 文件接收缓冲区, 一旦收到数据就调用 HEX 文件解码函数。

```

```

*          函数的返回值为 HEX 文件分析状态 (OK 或 CRC 错)。

```

```

*****/

```

```

Uchar hex_parser(void) {
    Uchar hex_data, status;
    status = HEX_DEC_OK;
    while (status == HEX_DEC_OK) { /* HEX 文件解码正确则继续处理 */
        if(nb_rx_data != 0) {
            hex_data = rx_buffer_rd();
            status = hex_decoder(hex_data);
        }
    }
    return status;
}

```

```

/***** HEX 文件解码函数 *****/

```

```

* 功    能: 对从 PC 机接收到的 ASCII 码文件按字节进行 HEX 文件记录解码, 分离出用于

```

```

*          FLASH 编程的数据以及校验和。函数参数 Uchar hex_data 为待解码的

```

```

*          HEX 文件数据。函数返回值为 HEX 文件分析状态 (OK 或 CRC 错)。

```

```

*****/

```

```

Uchar hex_decoder(Uchar hex_data) {
    static state = REC_MARK;
    static Uchar length, type, nb_byte, data_value, sum, sum_1, sum_2;
    static Uint16 offset;
    Uchar status;
    status = HEX_DEC_OK;

```

```
switch(state) {
    case REC_MARK: {                                /* 开始一个新记录 */
        if (hex_data == ':') {                      /* 检查首字符是否正确 */
            state = REC_LEN_1;                      /* 正确, 获取第一个长度字符 */
            nb_byte=0;
        }
        else
            state = REC_MARK;                        /* 错误, 跳过伪字符 */
        break;
    }
case REC_LEN_1: {                                  /* 获取长度字节中的第一个数字 */
    length = toint(hex_data) * 16;
    state = REC_LEN_2;
    break;
}
case REC_LEN_2: {                                  /* 获取长度字节中的第二个数字 */
    length = length + toint(hex_data);
    state = OFFSET_1;
    break;
}
case OFFSET_1: {                                    /* 获取地址字节中的第一个数字 */
    offset = toint(hex_data) * 4096;
    state = OFFSET_2;
    break;
}
case OFFSET_2: {                                    /* 获取地址字节中的第二个数字 */
    offset = offset + toint(hex_data) * 256;
    sum_1= offset / 256;
state = OFFSET_3;
    break;
}
case OFFSET_3: {                                    /* 获取地址字节中的第三个数字 */
    offset = offset + toint(hex_data) * 16;
state = OFFSET_4;
    break;
}
case OFFSET_4: {                                    /* 获取地址字节中的第四个数字 */
offset = offset + toint(hex_data);
sum_2 = offset - sum_1 * 256;
    state = REC_TYP_1;
    break;
}
case REC_TYP_1: {                                  /* 获取记录字节中的第一个数字 */
```

```
    type = toint(hex_data) * 16;
state = REC_TYP_2;
    break;
}
case REC_TYP_2: {          /* 获取记录字节中的第二个数字 */
    type = type + toint(hex_data);
    sum = length + sum_1 + sum_2 + type;
    if (length==0x00)
        state=CHEKSUM_1;    /* 如果没有数据, 转到计算校验和 */
    else
        state = DATA_1;    /* 否则下一步为获取数据记录 */
    break;
}
case DATA_1: {           /* 获取一个数据中的第一位数字 */
    data_value = toint(hex_data) * 16 ;
    state = DATA_2;
    break;
}

case DATA_2: {           /* 获取一个数据中的第二位数字 */
data_value = data_value + toint(hex_data);
sum = sum + data_value;
    flash_prog(offset + nb_byte, data_value);
    nb_byte++;
    if (nb_byte == length)
        state = CHEKSUM_1;    /* 若已到数据块尾, 转到校验和 */
    else
        state = DATA_1;    /* 否则获取块中的下一个数据 */
    break;
}
case CHEKSUM_1: {         /* 获取校验和中的第一个数字 */
    sum_1 = toint(hex_data) * 16;
sum = (~sum) + 1;
    state = CHEKSUM_2;
    break;
}
case CHEKSUM_2: {         /* 获取校验和中的第二个数字 */
    sum_1 = sum_1 + toint(hex_data);
    if (sum_1 != sum)
        status = HEX_DEC_CSERR;
    state = REC_MARK;    /* 准备好进行下一个记录 */
    if(type==0x01)
        status = HEX_DEC_END;    /* 传送结束 */
}
```

```

        break;
    }
}
return status;
}

```

FLASH 编程模块 flash.c 完成对 FLASH 存储器的编程，文件代码如下：

```

#include "config.h"
#include "isp.h"

/***** FLASH 写入函数 *****/
* 函数原型: void flash_wr(Uchar xdata *addr, Uchar val);
* 功    能: 向映射到外部数据存储器空间的 FLASH 地址单元 addr 写入一个字节
*           的数据 val。
*****/
void flash_wr(Uchar xdata *addr, Uchar val) {
    *addr = val;
}

/***** FLASH 读取函数 *****/
* 函数原型: Uchar flash_rd(Uchar xdata *addr);
* 功    能: 从映射到外部数据存储器空间的 FLASH 地址单元 addr 读取一个字节
*           的数据并返回主调函数。
*****/
Uchar flash_rd(Uchar xdata *addr) {
    return *addr;
}

/***** FLASH 命令函数 *****/
* 函数原型: void flash_cmd(Uchar cmd);
* 功    能: 执行 FLASH 命令序列 cmd。
*****/
void flash_cmd(Uchar cmd) {
    flash_wr(0x5555, 0xAA);
    flash_wr(0x2AAA, 0x55);
    flash_wr(0x5555, cmd);
}

/***** FLASH 擦除函数 *****/
* 函数原型: void flash_erase(void);
* 功    能: 擦除整个 FLASH 存储器。
*****/
void flash_erase(void) {

```

```

    Uchar pol_n, pol_n_1;
    flash_cmd(0x80);
    flash_cmd(0x10);
    pol_n_1 = flash_rd(0x5555);
    pol_n   = flash_rd(0x5555);
    while ((pol_n ^ pol_n_1) == 0x40) {
        pol_n_1 = pol_n;
        pol_n   = flash_rd(0x5555);
    }
}

/***** FLASH 编程函数 *****/
* 函数原型: void flash_prog(Uint16 addr, Uchar value);
* 功    能: 对指定的 FLASH 存储器地址 addr 编程一个字节的 data value。
*****/
void flash_prog(Uint16 addr, Uchar value) {
    flash_cmd(0xA0);
    flash_wr(addr, value);
    while (flash_rd(addr) != value);
}

/***** FLASH_ID 函数 *****/
* 函数原型: Uint16 flash_id();
* 功    能: 读取双字节 FLASH 存储器生产厂商和器件 ID, 高字节为厂商 ID,
*           低字节为器件 ID。
*****/
Uint16 flash_id() {
    Uint16 flash_id=0;
    flash_cmd(0x90);
    flash_id = flash_rd(0x0000) << 8 | flash_rd(0x0001);
    flash_cmd(0xF0);
    return flash_id;
}

```

头文件 compiler.h 代码如下:

```

#ifndef _COMPILER_H_
#define _COMPILER_H_
/* 宏定义 */
#define FALSE 0
#define TRUE 1
/* 数据类型定义 */
typedef unsigned char  Uchar;
typedef unsigned short Uint16;

```



```

typedef signed int      Int16;
typedef float           Float16;
typedef unsigned char   Bool;
/* KEIL C51 语法重新定义 */
#define Reentrant(x)    x reentrant
#define Sfr(x,y)        sfr x = y
#define Sbit(x,y,z)     sbit x = y ^ z
#define Interrupt(x,y)  x interrupt y
#define At(x)           _at_ x
#endif

```

头文件 config.h 代码如下:

```

#ifndef _CONFIG_H_
#define _CONFIG_H_
/* 文件包含 */
#include <reg51.h>
#include "compiler.h"
/* 用户配置 */
#define HAVE_XRAM          /* 定义默认单片机为带有片上 XRAM 的 AT89C51RD2 */
    #ifdef HAVE_XRAM
        #define NO_XDATA 0x02
        Sfr (AUXR, 0x8E);
    #endif
#endif /* _CONFIG_H_ */

```

头文件 isp.h 代码如下:

```

#ifndef _ISP_H_
#define _ISP_H_
/* 串行口操作函数说明 */
extern Uchar nb_rx_data;
void uart_init(void);
void uart_rx_enable(void);
void uart_rx_disable(void);
Bool rx_buffer_empty(void);
Uchar rx_buffer_rd(void);
/* HEX 文件处理 */
#define HEX_DEC_OK      0x01
#define HEX_DEC_END     0x02
#define HEX_DEC_CSERR   0x03
Uchar hex_parser (void);
/* FLASH API 函数说明 */
void flash_erase(void);
void flash_prog(Uint16, Uchar value);

```

```
Uint16 flash_id(void);  
#endif
```

8.8.3 P89C51RD2 单片机片内 FLASH 在应用中编程的 Cx51 程序

P89C51RD2 是 Philips 公司推出的带有 64KB 片内非易失性 FLASH 程序存储器的新型 80C51 单片机, 引脚和指令系统与普通 8051 完全兼容, 其最大特点是可通过串行口对器件进行在系统编程 (ISP) 和在应用中编程 (IAP), 编程次数可达 10000 次以上, 数据至少可保存 10 年。片内 ROM 中固化了默认的串行加载程序 (Boot Loader), 在 ISP 过程中通过 UART 将程序代码从 PC 机装入到片内 FLASH 存储器, 在 IAP 过程中用户程序可以通过调用片内 ROM 中的标准程序对 FLASH 存储器进行擦除和重新编程。该器件还可以选择 6 时钟或 12 时钟模式, 选择 6 时钟模式时每个机器周期为 6 个时钟周期, 时钟频率可高达 20MHz, 采用 12 时钟周期时频率可达 33MHz, 工作速度将比普通 8051 单片机提高一倍。可通过特殊功能寄存器的控制位在运行中改变 6 时钟/12 时钟模式。P89C51RD2 并行编程的硬件接口与 87C51 兼容。

P89C51RD2 的片内 FLASH 是一种带块擦除功能的非易失性存储器, 以 4K 字节为一个块单位。芯片内部带有 1K 字节的 Boot ROM 固件, 其中包含一个汇编语言子程序库和一个默认的用于在系统编程 (ISP) 的串行装载程序, 用户也可通过调用固件内的子程序来实现在应用中编程 (IAP)。Boot ROM 固件位于 64KB 程序存储器的最高端, 与片内 FLASH 地址 0FC00H~0FFFFH 相重叠, 两者之间的切换是通过特殊功能寄存器 AUXR1 中的 ENBOOT 位控制的:

ENBOOT=1 地址在 0FC00H 0FFFFH 范围寻址到 Boot ROM,

ENBOOT=0 地址在 0FC00H 0FFFFH 范围寻址到 FLASH。

P89C51RD2 提供一个专用的状态字节 (Status Byte) 和一个引导向量 (Boot Vector), Boot 向量和状态字节不在 SFR 中而在 FLASH 空间。当对 FLASH 进行擦除时状态字节和 Boot 向量也同时被擦除, 擦除后必须重新编程, 状态字节和 Boot 向量的编程是各自独立的。P89C51RD2 在复位信号的下降沿检查状态字节中的内容, 如果为 0 则转倒 0000H 地址开始执行程序, 这是用户应用程序的正常起始地址; 如果状态字节不为 0 则将 Boot 向量的值作为程序计数器的高字节, 低字节固定为 00H, Boot 向量的默认值为 0FCH, 对应 Boot ROM 地址 0FC00H。允许用户修改 Boot 向量, 从而为用户提供了应用的灵活性。此外还可以在正常时执行用户程序时手动强制进入 Boot ROM 固件, 方法是通过拉低 PSEN、ALE 悬浮为高且在复位下降沿使 EA 为高电平来实现, 这和设置状态字节非零的效果是一样的。P89C51RD2 有 3 个可编程保密位, 提供对片内代码和数据不同级别的保护, 防止对片内内容进行非法读写, 可通过 ISP/IAP 或并行方式对保密位编程。P89C51RD2 进行 ISP 编程和擦除电压为 +5V, ISP 字节编程时间为 $8\mu\text{s}$, 块擦除 (4KB) 时间小于 3 秒, 整片擦除 (64KB) 小于 11 秒。

有以下三种方法来实现对 FLASH 的编程或擦除。

- ① 通过 Boot ROM 固件由主机进行串行在系统中编程 (ISP)。
- ② 在应用程序中调用 Boot ROM 固件中的子程序进行串行在应用中编程 (IAP)。

③ 使用商用编程器进行并行编程。

P89C51RD2 在应用中编程 (IAP) 是通过带参数调用 Boot ROM 固件中的 IAP 子程序来实现的, 调用时根据 P89C51RD2 数据手册的描述, 先向单片机的工作寄存器 R0、R1、累加器 A 以及数据指针 DPTR 中写入必要的参数, 然后用一条汇编语言长调用语句 “LCALL 0FFFH” 即可完成由不同参数指定的 IAP 功能。

下面给出一个实现 P89C51RD2 在应用中编程 (IAP) 的 C51 应用程序实例, 包含两个模块文件: 主程序模块文件 main.c 和 IAP 应用文件 rx2iaplib.a51, 后者实际上是一个向 Boot ROM 固件中的 IAP 子程序传递不同参数来完成不同 IAP 功能的汇编语言程序, 如果将 rx2iaplib.a51 作为一个库文件加入到 μ Vision2 的项目中, 就可以根据需要随时对 P89C51RD2 进行在应用中编程。

主程序模块文件 main.c 代码如下:

```
#include <stdio.h>
/* 定义 P89C51RD2 特殊功能寄存器 */
sfr RCAP2L = 0xCA;
sfr RCAP2H = 0xCB;
sfr SCON = 0x98;
sbit RCLK = 0xCD;
sbit TCLK = 0xCC;
sbit TR2 = 0xCA;
/* 定义 P89C51RD2 保密位 */
#define SECURITY_BIT_1 0x02
#define SECURITY_BIT_2 0x04
#define SECURITY_BIT_3 0x08
#define SECURITY_BIT_1_SET(v) (v & SECURITY_BIT_1)
#define SECURITY_BIT_2_SET(v) (v & SECURITY_BIT_2)
#define SECURITY_BIT_3_SET(v) (v & SECURITY_BIT_3)
/* 定义 P89C51RD2 的 FLASH 块 */
#define BLOCK_0 0x00
#define BLOCK_1 0x20
#define BLOCK_2 0x40
#define BLOCK_3 0x80
#define BLOCK_4 0xc0
#define BLOCK_0x0000_0x1FFF BLOCK_0
#define BLOCK_0x2000_0x3FFF BLOCK_1
#define BLOCK_0x4000_0x7FFF BLOCK_2
#define BLOCK_0x8000_0xBFFF BLOCK_3
#define BLOCK_0xc000_0xFFFF BLOCK_4
/* 外部汇编语言 IAP 函数说明 */
extern unsigned char iap_read_manufacturer_id(void);
extern void iap_init(unsigned char frequency);
extern unsigned char iap_read_device_id(unsigned char id_number);
```

```

extern unsigned char iap_read_security_bits(void);
extern void iap_program_security_bits(unsigned char bits);
extern unsigned char iap_program_data_byte(unsigned char val,
                                           unsigned int addr);
extern unsigned char iap_read_data_byte(unsigned int addr);
extern void iap_erase_block(unsigned char block);
extern void iap_erase_chip(void);
extern unsigned char iap_read_boot_vector(void);
extern unsigned char iap_read_status_byte(void);
extern void iap_erase_boot_vector_status_byte(void);
extern void iap_program_status_byte(unsigned char status_byte);
extern void iap_program_boot_vector(unsigned char boot_vector);
/* 定义指令助记符 MOV 和 RET 的机器代码 */
#define MOV 0x75
#define RET 0x22

unsigned char data foo;
void (*funcat4000)(void) = (void (code *) (void))0x4000;

/***** 串行口初始化函数 *****/
* 功    能: 初始化 P89C51RD2 串行口, 定时器 T2 作为波特率发生器, 采用 6 时钟
*          模式, 采用 20MHz 晶振时波特率为 9600。
*****/
void uart_init(void) {
    RCLK = 1;
    TCLK = 1;
    RCAP2H = 255;
    RCAP2L = 126;
    SCON = 0x52;
    TR2 = 1;
}

/***** 主函数 *****/
* 功    能: 通过调用汇编语言程序 IAP 库文件 rx2iaplib.a51 中的不同函数
*          来完成各种 IAP 功能。
*****/
void main(void) {
    unsigned char man_id, id1, id2, boot, status;

    uart_init();           /* 初始化串行口 */
    iap_init(20);          /* IAP 库初始化, 用整数指定所用晶振值 */
    foo = 0;
    man_id = iap_read_manufacturer_id(); /* 读取器件生产厂家 ID */

```

```

    id1 = iap_read_device_id(1);          /* 读取器件 ID */
    id2 = iap_read_device_id(2);
    printf("Manufacturer ID = %2.2bxH\n",
           Device ID 1 = %2.2bxH\n",
           Device ID 2 = %2.2bxH\n", man_id, id1, id2); /* 输出 ID */

    boot = iap_read_boot_vector();        /* 读取并输出器件引导向量 */
    printf("Boot Vector = %2.2bxH\n", boot);
    status = iap_read_status_byte();      /* 读取并输出器件状态字节 */
    printf("Status Byte = %2.2bxH\n", status);

    printf("Erasing 4000H -> 7FFFH...\n");
    iap_erase_block(BLOCK_0x4000_0x7FFF); /* 擦除 FLASH 存储器 BLOCK_2 块 */

    printf("Programming memory...\n");
    /* 编程 FLASH 的一个字节, 地址为 4000H, 内容为 MOV 指令代码 */
    if(iap_program_data_byte(MOV, 0x4000))
        printf("Error programming 4000H\n");
    /* 编程 FLASH 的一个字节, 地址为 4001H, 内容为 foo 的值 */
    if(iap_program_data_byte((unsigned char)&foo, 0x4001))
        printf("Error programming 4001H\n");
    /* 编程 FLASH 的一个字节, 地址为 4002H, 内容为 FFH */
    if (iap_program_data_byte(0xff, 0x4002))
        printf("Error programming 4002H\n");
    /* 编程 FLASH 的一个字节, 地址为 4003H, 内容为 RET 指令代码 */
    if (iap_program_data_byte(RET, 0x4003))
        printf("Error programming 4003H\n");
    /* 编程保密位 1 和保密位 2 */
    printf("Setting security bits...\n");
    iap_program_security_bits(SEcurity_BIT_1 | SEcurity_BIT_2);
    printf("Calling new code...\n");
    funcat4000();          /* 调用位于 FLASH 地址 4000H 处的代码 */
    if (foo == 0xff) printf("foo = FFH\n");
    printf("Finished.\n");
    while(1);
}

```

IAP 应用文件 rx2iaplib.a51 代码如下:

```

$NOMOD51
NAME    RX2IAPLIB
; 段定义
?DT?RX2IAPLIB                                SEGMENT DATA
?PR?iap_read_manufacturer_id?RX2IAPLIB        SEGMENT CODE

```

```

?PR?_iap_init?RX2IAPLIB          SEGMENT CODE
?PR?_iap_read_device_id?RX2IAPLIB SEGMENT CODE
?PR?iap_read_security_bits?RX2IAPLIB SEGMENT CODE
?PR?_iap_program_security_bits?RX2IAPLIB SEGMENT CODE
?PR?_iap_program_data_byte?RX2IAPLIB SEGMENT CODE
?PR?_iap_read_data_byte?RX2IAPLIB SEGMENT CODE
?PR?_iap_erase_block?RX2IAPLIB    SEGMENT CODE
?PR?iap_erase_chip?RX2IAPLIB      SEGMENT CODE
?PR?iap_read_boot_vector?RX2IAPLIB SEGMENT CODE
?PR?iap_read_status_byte?RX2IAPLIB SEGMENT CODE
?PR?iap_erase_boot_vector_status_byte?RX2IAPLIB SEGMENT CODE
?PR?_iap_program_status_byte?RX2IAPLIB SEGMENT CODE
?PR?_iap_program_boot_vector?RX2IAPLIB SEGMENT CODE

```

；函数名及全局变量

```

PUBLIC iap_freq
PUBLIC iap_read_manufacturer_id
PUBLIC _iap_init
PUBLIC _iap_read_device_id
PUBLIC iap_read_security_bits
PUBLIC _iap_program_security_bits
PUBLIC _iap_program_data_byte
PUBLIC _iap_read_data_byte
PUBLIC _iap_erase_block
PUBLIC iap_erase_chip
PUBLIC iap_read_boot_vector
PUBLIC iap_read_status_byte
PUBLIC iap_erase_boot_vector_status_byte
PUBLIC _iap_program_status_byte
PUBLIC _iap_program_boot_vector

```

；特殊功能寄存器定义

```

DPH      DATA    083H
DPL      DATA    082H
AUXR1    DATA    0A2H
EA        BIT      0AFH
CMOD     DATA    0D9H
ACC       DATA    0E0H
IE        DATA    0A8H

```

```

RSEG ?DT?RX2IAPLIB
?DT?RX2IAPLIB?BASE:

```

```

iap_freq: DS 1 ; 芯片使用的晶振频率(近似整数值)

```

```

; ***** 芯片厂商 ID 读取函数 *****
; 函数原型: unsigned char iap_read_manufacturer_id(void);
; 功 能: 返回值为厂商 ID (15H = Philips)。
; *****

        RSEG ?PR?iap_read_manufacturer_id?RX2IAPLIB
iap_read_manufacturer_id:
        PUSH    IE                ; IE 保存堆栈
        CLR     EA                ; 关中断
        MOV     A,CMOD            ; 保存 CMOD
        MOV     R2,A
        JNB     ACC.6,?IAPTAG2
        ANL     CMOD,#0BFH        ; 禁止看门狗
?IAPTAG2:
        ORL     AUXR1,#020H        ; 允许 Boot Rom
        MOV     R0,iap_freq        ; 晶振频率值->R0
        MOV     R1,#00H
        MOV     DPTR,#0000H
        CALL    0FFF0H            ; 调用 IAP 子程序
        MOV     R7,A              ; 将 A 中的 ID 值->R7
        ANL     AUXR1,#0DFH        ; 禁止 Boot Rom
        MOV     CMOD,R2            ; 恢复 CMOD (恢复看门狗状态)
        POP     IE                ; 恢复中断状态
        RET

; ***** IAP 初始化函数 *****
; 函数原型: void iap_init(unsigned char frequency);
; 功 能: 设置芯片晶振频率。
; 参 数: 芯片所使用晶振的近似整数值。
; *****

        RSEG ?PR?_iap_init?RX2IAPLIB
_iap_init:
        MOV     iap_freq,R7
        RET
; end of iap_init

; ***** 芯片 ID 读取函数 *****
; 函数原型: unsigned char iap_read_device_id(unsigned char id_number);
; 功 能: 读取器件 ID。
; 参 数: id_number 可为 1 或 2。
; *****

        RSEG ?PR?_iap_read_device_id?RX2IAPLIB
_iap_read_device_id:

```

```

    PUSH    IE                      ; 关中断
    CLR     EA
    MOV     A,CMOD
    MOV     R2,A                    ; 保存 CMOD
    JNB     ACC.6,?IAPTAG4          ; 禁止看门狗
    ANL     CMOD,#0BFH
?IAPTAG4:
    ORL     AUXR1,#020H             ; 允许 bootrom
    MOV     R0,iap_freq             ; 晶振频率->R0
    MOV     R1,#00H
    MOV     DPH,#00H
    MOV     DPL,R7                  ; 参数 id_number -> DPL
    CALL    0FFF0H                  ; 调用 IAP 子程序
    MOV     R7,A                    ; 将 A 中的 ID 值->R7
    ANL     AUXR1,#0DFH             ; 禁止 bootrom
    MOV     CMOD,R2                 ; 恢复 CMOD(恢复看门狗状态)
    POP     IE                      ; 恢复中断状态
    RET

```

; ***** 保密位读取函数 *****

; 函数原型: unsigned char iap_read_security_bits(void);

; 功 能: 读取保密位状态, 返回字节中的第 1, 2, 3 位分别对应保密位 1, 2, 3。

; *****

RSEG ?PR?iap_read_security_bits?RX2IAPLIB

iap_read_security_bits:

```

    PUSH    IE                      ; 关中断
    CLR     EA
    MOV     A,CMOD
    MOV     R2,A                    ; 保存 CMOD
    JNB     ACC.6,?IAPTAG6          ; 禁止看门狗
    ANL     CMOD,#0BFH
?IAPTAG6:
    ORL     AUXR1,#020H             ; 允许 bootrom
    MOV     R0,iap_freq             ; 晶振频率->R0
    MOV     R1,#07H
    MOV     DPTR,#0000H
    CALL    0FFF0H                  ; 调用 IAP 子程序
    MOV     R7,A                    ; 将 A 中的 ID 值->R7
    ANL     AUXR1,#0DFH             ; 禁止 bootrom
    MOV     CMOD,R2                 ; 恢复 CMOD(恢复看门狗状态)
    POP     IE                      ; 恢复中断状态
    RET

```



```

; ***** 保密位编程函数 *****
; 函数原型: void iap_program_security_bits(unsigned char bits);
; 功 能: 编程保密位, 参数字节中的第 1, 2, 3 位分别对应保密位 1, 2, 3。
; *****

        RSEG ?PR?_iap_program_security_bits?RX2IAPLIB
_iap_program_security_bits:
; prologue
        PUSH    IE                ; 关中断
        CLR     EA
        MOV     A,CMOD
        MOV     R2,A              ; 保存 CMOD
        JNB     ACC.6,?IAPTAG10   ; 禁止看门狗
        ANL     CMOD,#0BFH
?IAPTAG10:
        ORL     AUXR1,#020H        ; 允许 bootrom
        MOV     R6,#00H            ; 位计数器
        MOV     A,R7              ; 参数 -> A
        MOV     DPH,#00H
?IAPTAG7:
        JNB     ACC.1,?IAPTAG8     ; 检查 ACC.1 是否为 1
        MOV     R0,iap_freq        ; 晶振频率->R0
        MOV     R1,#05H
        MOV     DPL,R6
        CALL    0FFF0H             ; 调用 IAP 子程序
?IAPTAG8:
        CLR     C                  ; A 带进位右移 1 位
        RRC     A
        INC     R6                  ; 位计数器加 1
        CJNE    R6,#003H,?IAPTAG9 ; 检查 3 位全部完成否?
?IAPTAG9:
        JC      ?IAPTAG7           ; 未完, 返回编程下一位
; epilogue
        ANL     AUXR1,#0DFH        ; 禁止 bootrom
        MOV     CMOD,R2            ; 恢复 CMOD(恢复看门狗状态)
        POP     IE                 ; 恢复中断状态
        RET

; ***** 数据字节编程函数 *****
; 函数原型: unsigned char iap_program_data_byte(unsigned char val,
;
;                                     unsigned int addr);
; 功 能: 编程 FLASH 中的 1 个字节。
; 参 数: val = 待编程的字节数据。
;
;         addr = 16 位 FLASH 地址。

```

; 返回值: 编程成功返回 0, 编程失败返回非 0 值。

; *****

```

        RSEG ?PR?_iap_program_data_byte?RX2IAPLIB
_iap_program_data_byte:
        PUSH    IE                      ; 关中断
        CLR     EA
        MOV     A,CMOD
        MOV     R2,A                    ; 保存 CMOD
        JNB     ACC.6,?IAPTAG11         ; 禁止看门狗
        ANL     CMOD,#0BFH
?IAPTAG11:
        ORL     AUXR1,#020H             ; 允许 bootrom
        MOV     R0,iap_freq             ; 晶振频率 -> R0
        MOV     R1,#02H
        MOV     DPH,R4                  ; 编程地址 -> DPTR
        MOV     DPL,R5
        MOV     A,R7                    ; 编程数据 -> A
        CALL    0FFF0H                  ; 调用 IAP 子程序
        MOV     R7,A                    ; A 中返回值 -> R7
        ANL     AUXR1,#0DFH             ; 禁止 bootrom
        MOV     CMOD,R2                 ; 恢复 CMOD(恢复看门狗状态)
        POP     IE                      ; 恢复中断状态
        RET

```

; ***** 数据字节读取函数 *****

; 函数原型: unsigned char iap_read_data_byte(unsigned int addr);

; 功 能: 从 FLASH 中读取一个字节的数并返回给主调函数。

; 参 数: addr = 16 位 FLASH 地址。

; *****

```

        RSEG ?PR?_iap_read_data_byte?RX2IAPLIB
_iap_read_data_byte:
        PUSH    IE                      ; 关中断
        CLR     EA
        MOV     A,CMOD
        MOV     R2,A                    ; 保存 CMOD
        JNB     ACC.6,?IAPTAG12         ; 禁止看门狗
        ANL     CMOD,#0BFH
?IAPTAG12:
        ORL     AUXR1,#020H             ; 允许 bootrom
        MOV     R0,iap_freq             ; 晶振频率 -> R0
        MOV     R1,#03H
        MOV     DPH,R6                  ; FLASH 地址 -> DPTR
        MOV     DPL,R7

```

```

        CALL    0FFF0H          ; 调用 IAP 子程序
        MOV     R7,A            ; A 中的返回值 -> R7
        ANL     AUXR1,#0DFH     ; 禁止 bootrom
        MOV     CMOD,R2         ; 恢复 CMOD(恢复看门狗状态)
        POP     IE              ; 恢复中断状态
        RET

; ***** 块擦除函数 *****
; 函数原型: void iap_erase_block(unsigned char block);
; 功    能: 擦除 FLASH 中的一个块。
; 参    数: 待擦除的 FLASH 块地址。
; *****
        RSEG    ?PR?_iap_erase_block?RX2IAPLIB
_iap_erase_block:
        PUSH    IE              ; 关中断
        CLR     EA
        MOV     A,CMOD
        MOV     R2,A            ; 保存 CMOD
        JNB     ACC.6,?IAPTAG13 ; 禁止看门狗
        ANL     CMOD,#0BFH
?IAPTAG13:
        ORL     AUXR1,#020H     ; 允许 bootrom
        MOV     R0,iap_freq     ; 晶振频率 -> R0
        MOV     R1,#01H
        MOV     DPH,R7          ; 块地址参数 -> DPH
        MOV     DPL,#00H
        CALL    0FFF0H          ; 调用 IAP 子程序
        ANL     AUXR1,#0DFH     ; 禁止 bootrom
        MOV     CMOD,R2         ; 恢复 CMOD(恢复看门狗状态)
        POP     IE              ; 恢复中断状态

; ***** 芯片擦除函数 *****
; 函数原型: void iap_erase_chip(void);
; 功    能: 擦除整个 FLASH。
; *****
        RSEG    ?PR?iap_erase_chip?RX2IAPLIB
iap_erase_chip:
        CLR     EA              ; 关中断
        ANL     CMOD,#0BFH     ; 禁止看门狗
        ORL     AUXR1,#020H     ; 允许 bootrom
        MOV     R0,iap_freq     ; 晶振频率 -> R0
        MOV     R1,#08H
        CALL    0FFF0H          ; 调用 IAP 子程序

```

```

RET

; ***** 引导向量读取函数 *****
; 函数原型: unsigned char iap_read_boot_vector(void);
; 功    能: 返回读取的引导向量字节。
; *****

RSEG ?PR?iap_read_boot_vector?RX2IAPLIB
iap_read_boot_vector:
    PUSH    IE                ; 关中断
    CLR     EA
    MOV     A,CMOD
    MOV     R2,A              ; 保存 CMOD
    JNB     ACC.6,?IAPTAG14   ; 禁止看门狗
    ANL     CMOD,#0BFH
?IAPTAG14:
    ORL     AUXR1,#020H       ; 允许 bootrom
    MOV     R0,iap_freq       ; 晶振频率 -> R0
    MOV     R1,#07H
    MOV     DPTR,#0002H
    CALL    0FFF0H            ; 调用 IAP 子程序
    MOV     R7,A              ; 返回值 -> R7
    ANL     AUXR1,#0DFH       ; 禁止 bootrom
    MOV     CMOD,R2           ; 恢复 CMOD(恢复看门狗状态)
    POP     IE                ; 恢复中断状态
    RET

; ***** 状态字节读取函数 *****
; 函数原型: unsigned char iap_read_status_byte(void);
; 功    能: 返回读取的状态字节。
; *****

RSEG ?PR?iap_read_status_byte?RX2IAPLIB
iap_read_status_byte:
    PUSH    IE                ; 关中断
    CLR     EA
    MOV     A,CMOD
    MOV     R2,A              ; 保存 CMOD
    JNB     ACC.6,?IAPTAG15   ; 禁止看门狗
    ANL     CMOD,#0BFH
?IAPTAG15:
    ORL     AUXR1,#020H       ; 允许 bootrom
    MOV     R0,iap_freq       ; 晶振频率 -> R0
    MOV     R1,#07H
    MOV     DPTR,#0001H

```

```

CALL    0FFF0H          ; 调用 IAP 子程序
MOV     R7,A            ; 返回值 -> R7
ANL     AUXR1,#0DFH     ; 禁止 bootrom
MOV     CMOD,R2         ; 恢复 CMOD(恢复看门狗状态)
POP     IE              ; 恢复中断状态
RET

; ***** 引导向量与状态字节擦除函数 *****
; 函数原型: void iap_erase_boot_vector_status_byte(void);
; 功    能: 擦除引导向量与状态字节。
; *****

RSEG ?PR?iap_erase_boot_vector_status_byte?RX2IAPLIB
iap_erase_boot_vector_status_byte:
    PUSH    IE          ; 关中断
    CLR     EA
    MOV     A,CMOD
    MOV     R2,A         ; 保存 CMOD
    JNB     ACC.6,?IAPTAG16 ; 禁止看门狗
    ANL     CMOD,#0BFH
?IAPTAG16:
    ORL     AUXR1,#020H  ; 允许 bootrom
    MOV     R0,iap_freq  ; 晶振频率 -> R0
    MOV     R1,#04H
    MOV     DPH,#00H
    CALL    0FFF0H       ; 调用 IAP 子程序
    ANL     AUXR1,#0DFH  ; 禁止 bootrom
    MOV     CMOD,R2      ; 恢复 CMOD(恢复看门狗状态)
    POP     IE           ; 恢复中断状态
    RET

; ***** 状态字节编程函数 *****
; 函数原型: void iap_program_status_byte(unsigned char status_byte);
; 功    能: 编程状态字节。
; 参    数: status_byte = 待编程的状态字节。
; *****

RSEG ?PR?_iap_program_status_byte?RX2IAPLIB
_iap_program_status_byte:
    PUSH    IE          ; 关中断
    CLR     EA
    MOV     A,CMOD
    MOV     R2,A         ; 保存 CMOD
    JNB     ACC.6,?IAPTAG17 ; 禁止看门狗
    ANL     CMOD,#0BFH

```

```

?IAPTAG17:
    ORL    AUXR1,#020H          ; 允许 bootrom
    MOV    R0,iap_freq          ; 晶振频率 -> R0
    MOV    R1,#06H
    MOV    A,R7                  ; 待编程的状态字节参数 -> A
    MOV    DPTR,#0000H
    CALL   0FFF0H               ; 调用 IAP 子程序
    ANL    AUXR1,#0DFH          ; 禁止 bootrom
    MOV    CMOD,R2               ; 恢复 CMOD(恢复看门狗状态)
    POP    IE                    ; 恢复中断状态
    RET

; ***** 引导向量编程函数 *****
; 函数原型: void iap_program_boot_vector(unsigned char boot_vector);
; 功    能: 重新编程引导向量。
; 参    数: boot_vecto = 待编程的引导向量字节。
; *****

RSEG ?PR?_iap_program_boot_vector?RX2IAPLIB
_iap_program_boot_vector:
    PUSH   IE                    ; 关中断
    CLR    EA
    MOV    A,CMOD
    MOV    R2,A                  ; 保存 CMOD
    JNB    ACC.6,?IAPTAG18       ; 禁止看门狗
    ANL    CMOD,#0BFH
?IAPTAG18:
    ORL    AUXR1,#020H          ; 允许 bootrom
    MOV    R0,iap_freq          ; 晶振频率 -> R0
    MOV    R1,#06H
    MOV    A,R7                  ; 待编程的引导向量字节参数 -> A
    MOV    DPTR,#0001H
    CALL   0FFF0H               ; 调用 IAP 子程序
    ANL    AUXR1,#0DFH          ; 禁止 bootrom
    MOV    CMOD,R2               ; 恢复 CMOD(恢复看门狗状态)
    POP    IE                    ; 恢复中断状态
    RET

    END

```

8.9 8051 单片机并行接口扩展应用编程

在 8051 单片机实际应用系统中经常需要进行并行 I/O 接口的扩展。本节介绍几种常用

的并行 I/O 接口扩展电路及其 C51 驱动程序。

8.9.1 打印输出接口及其驱动程序

通用并行接口打印机采用规范化的“Centronics”标准与计算机进行通信,下面以 PP40 彩色描绘器为例介绍其与单片机 8031 的接口方法,并给出该接口的 C51 驱动程序。表 8-19 所示为 PP40 的接口信号,所有 I/O 信号都与 TTL 电平兼容。

表 8-19 PP40 的接口信号

针 位	信 号	针 位	信 号	针 位	信 号	针 位	信 号
1	STROBE	10	ACK	19	GND*	28	GND*
2	DATA1	11	BUSY	20	GND*	29	GND*
3	DATA2	12	GND	21	GND*	30	GND*
4	DATA3	13	NC	22	GND*	31	NC
5	DATA4	14	GND	23	GND*	32	NC
6	DATA5	15	GND	24	GND*	33	GND
7	DATA6	16	GND	25	GND*	34	NC
8	DATA7	17	GND	26	GND*	35	NC
9	DATA8	18	NC	27	GND*	36	NC

* 用以和信号线绞线以提高抗干扰能力, NC 为空脚。

各信号意义如下。

DATA1~DATA8: 数据信号。

STROBE: 选通输入信号。在它的上升沿将 DATA1~DATA8 上的信息打入 PP40, 并启动 PP40 机械装置开始描绘。

BUSY: 状态输出信号。PP40 正在处理主机的命令或数据时 BUSY 输出高电平, 空闲时输出低电平。

ACK: 响应输出信号。当 PP40 接收并处理完主机的命令或数据时, ACK 输出一个负脉冲。

图 8.58 所示为 PP40 的工作时序 (Centronics 标准)。

PP40 具有文本模式和图案模式两种操作方式, 初始加电后为文本模式。当 PP40 处于文本模式时, 主机将回车符 (0DH) 和控制 2 编码 (12H) 写入 PP40, 则由文本模式变为图案模式; 再将回车符 (0DH) 和控制 1 编码 (11H) 写入 PP40, 则又回到文本模式。PP40 在文本模式下可打印出所有的 ASCII 字符, 在图案模式下可描绘出用户设计的各种彩色图案。表 8-20 为 PP40 在文本模式下的可打印 ASCII 字符。表中除了字符编码之外, 还列出了一些控制编码, 它们的定义如下。

回位 (08H): 使笔回到前面一个字符位置, 若笔已处于最左边位置, 则该命令失效。

进纸 (0AH): 将打印纸推进一行。

退纸 (0BH): 将打印纸退后一行。

回车 (0DH): 使笔回到最左边, 并进纸一行。

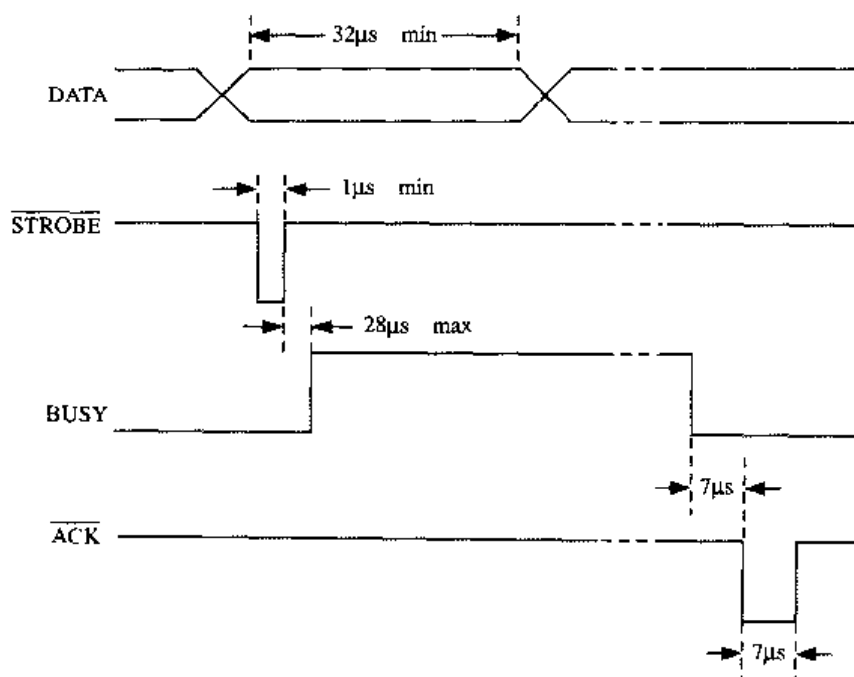


图 8.58 PP40 的工作时序 (Centronics 标准)

控制 1 (11H): 与回车符配合, 将 PP40 置为文本模式。

控制 2 (12H): 与回车符配合, 将 PP40 置为图案模式。

转色 (1DH): 使笔架转动一个位置, 更换一种颜色的描绘笔。

表 8-20 PP40 的可打印 ASCII 字符

	0	1	2	3	4	5	6	7
0				0	@	P	,	p
1		DC1	0	1	A	Q	a	q
2		DC2	"	2	B	R	b	r
3			#	3	C	S	c	s
4			\$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7			'	7	G	W	g	w
8	BS		l	8	H	X	h	x
9			j	9	I	Y	i	y
A	LF		*	:	J	Z	j	z
B	LU		+	;	K	[k	{
C			,	<	L	\	l	
D	CR	NC	-	=	M]	m	}
E			.	>	N	^	n	~
F			/	?	O	_	o	×

图 8.59 所示为 8031 与 PP40 的一种接口电路, 采用查询方式工作, 8031 的 P1 口输出打印数据, P3.5 作为 PP40 的选通信号, P3.3 用来查询 PP40 的工作状态。

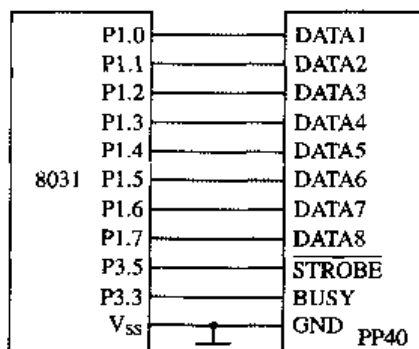


图 8.59 PP40 与 8031 的查询方式接口

下面给出基于图 8.59 接口电路的 Cx51 驱动程序。

```
#pragma db oe sb
#include<reg51.h>
#define uchar unsigned char

sbit STB=P3^5;
sbit BUSY=P3^3;

uchar code line[]={
    {0x57,0x45,0x4c,0x43,0x4f,0x4d,0x45}; /* 预定义字符信息"WELCOME" */

/***** 字符打印函数 *****/
void prnchar(uchar x) {
    P1=x; /* 输出一个 ASCII 字符 */
    STB=0; /* 产生 STROBE 低电平 */
    STB=1; /* 产生 STROBE 上升沿 */
    While(BUSY); /* 查询等待 PP40 打印结束 */
}

/***** 行打印函数 *****/
void prnline(void) {
    uchar i;
    for (i=0;i<=6;i++) { /* 打印输出一行预定义信息 */
        P1=line[i];
        STB=0;
        STB=1;
        While(BUSY);
    }
}
```

```

}

/***** 主函数 *****/
void main(void) {
    prnline();          /* 打印输出"WELCOME" */
    prnchar(0x0D);      /* 换行 */
    prnchar(0x31);      /* 打印输出"1997" */
    prnchar(0x39);
    prnchar(0x39);
    prnchar(0x37);
}

```

8.9.2 D/A 及 A/D 转换接口及其驱动程序

在单片机应用系统中只要涉及到测量与控制,就必然需要 D/A 及 A/D 转换接口。DAC0832 是一种使用十分广泛的 D/A 转换芯片,图 8.60 所示是 DAC0832 的内部逻辑结构及引脚分配图,各引脚的功能如下:

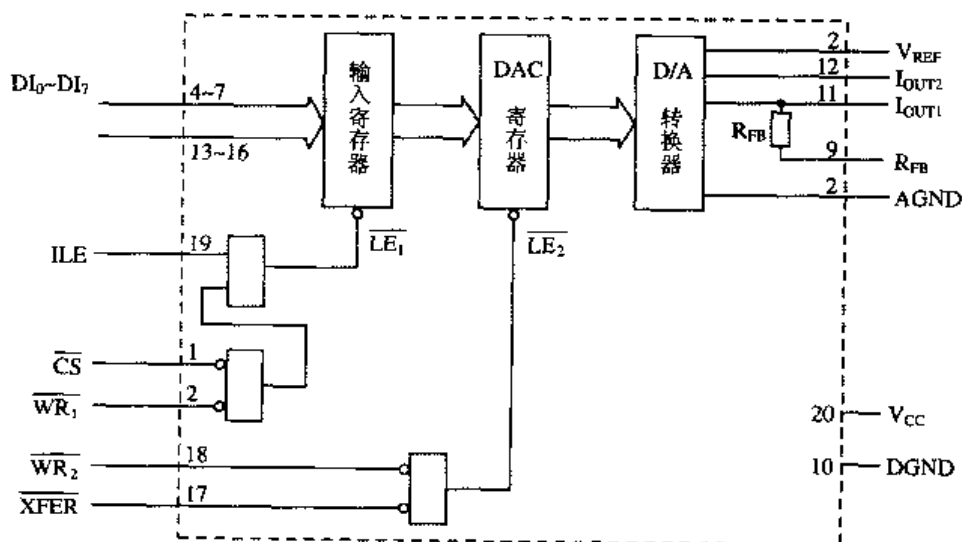


图 8.60 DAC0832 的逻辑结构及引脚分配

- | | |
|---------------------------|----------------------------|
| $DI_0 \sim DI_7$: | 8 位数据输入端。 |
| ILE: | 输入寄存器的数据允许锁存信号。 |
| CS: | 输入寄存器选择信号。 |
| WR_1 : | 输入寄存器的数据写信号。 |
| XFER: | 数据向 DAC 寄存器传送信号, 传送后即启动转换。 |
| WR_2 : | DAC 寄存器写信号, 并启动转换。 |
| I_{OUT1} 、 I_{OUT2} : | 电流输出端。 |
| V_{REF} : | 参考电压输入端。 |
| R_{FB} : | 反馈信号输入端。 |
| V_{CC} : | 电源电压端。 |

AGND: 模拟地。

DGND: 数字地。

DAC0832 的输出是电流型的, 如果需要输出电压信号, 可通过运算放大器进行变换实现。图 8.61 是 DAC0832 与 8031 单片机的一种接口电路, 通过运算放大器 5G24 可得到 0~5V 的输出电压。该电路采用线选方式规定 DAC0832 的端口地址为 7FFFH。下面给出对应于该接口电路的 Cx51 驱动程序。

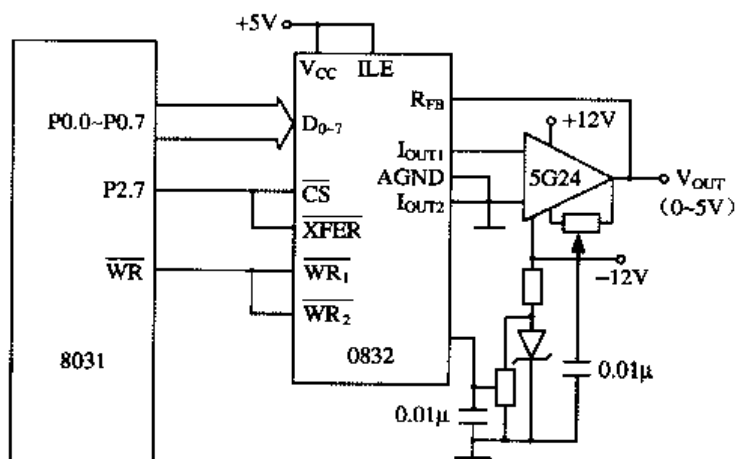


图 8.61 DAC0832 与单片机 8031 的接口

```
#pragma db oe sb
#include<reg51.h>
#include<absacc.h>
```

```
#define DAC0832 XBYTE[0x7fff] /* 定义 DAC0832 端口地址 */
#define uchar unsigned char
```

```
/* ***** 延时函数 ***** */
void delay(uchar t) {
    while(t--);
}
```

```
/* ***** 锯齿波发生函数 ***** */
void saw(void) {
    uchar i;
    for (i=0;i<255;i++) {
        DAC0832=i;
    }
}
```

```
/* ***** 方波发生函数 ***** */
void square(void) {
    DAC0832=0x00;
```

```

delay(0x10);
DAC0832=0xff;
delay(0x10);
}

/***** 主函数 *****/
void main(void) {
    uchar i,j;
    i=j=0xff;
    while(i--) {
        saw();          /* 产生一段锯齿波 */
    }
    while(j--) {
        square();       /* 产生一段方波 */
    }
}

```

ADC0809 是一种 8 路模拟量输入，8 位数字量输出的逐次比较式 ADC 芯片，图 8.62 为 ADC0809 的原理结构框图，芯片的主要部分是一个 8 位的逐次比较式 A/D 转换器。为了能够实现 8 路模拟信号的分时采集，在芯片内部设置了多路模拟开关及通道地址锁存和译码电路，因此能对多路模拟信号进行分时采集和转换。转换后的数据送入三态输出数据锁存器。ADC0809 的最大不可调误差为 $\pm 1\text{LSB}$ ，典型时钟频率为 640kHz，时钟信号应由外部提供。每一个通道的转换时间约为 $100\mu\text{s}$ 。

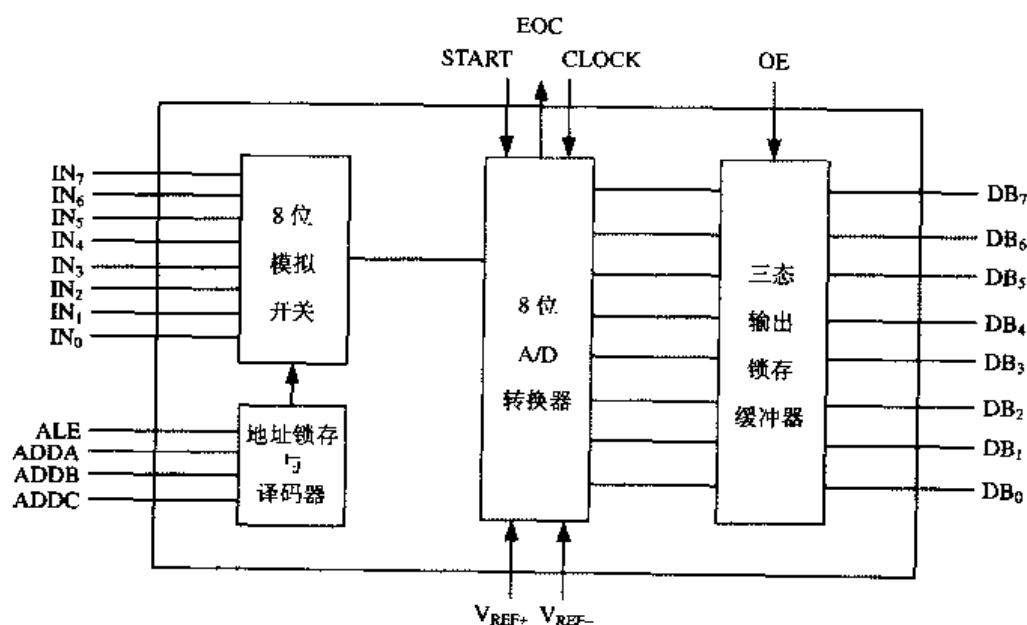


图 8.62 ADC0809 的原理结构框图

图 8.63 为 ADC0809 的引脚排列图，各引脚的功能如下。

IN₀~IN₇: 8 路模拟量输入端。

D ₀ ~D ₇ :	数字量输出端。
START:	启动脉冲输入端, 脉冲上升沿复位 0809, 下降沿启动 A/D 转换。
ALE:	地址锁存信号, 高电平有效时把三个地址信号送入地址锁存器, 并经地址译码得到地址输出, 用以选择相应的模拟输入通道。
EOC:	转换结束信号, 转换开始时变低, 转换结束时变高, 变高时将转换结果打入三态输出锁存器。如果将 EOC 和 START 相连, 加上一个启动脉冲则连续进行转换。
OE:	输出允许信号输入端。
CLOCK:	时钟输入信号, 最高允许值为 640KHz。
VREF (+):	正基准电压输入端。
VREF (-):	负基准电压输入端。通常将 VREF (+) 接+5V, VREF (-) 接地。
V _{cc} :	电源电压, 可从+5V~+15V。

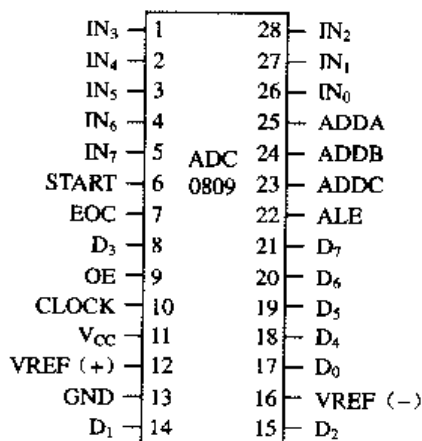


图 8.63 ADC0809 的引脚排列图

图 8.64 为 ADC0809 的工作时序, 由于 ADC0809 芯片没有专门的片选端, 因此在设计与单片机接口时必须参考其工作时序, 图 8.65 所示为 ADC0809 与单片机 8031 的一种接口电路。采用线选法规定其端口地址, 用单片机的 P2.7 引脚作为片选信号, 片选信号和 WR 信号一起经或非门产生 0809 的启动信号 START 和地址锁存信号 ALE; 片选信号和 RD 信号一起经或非门产生 0809 输出允许信号 OE, 因此端口地址为 0x7fff。0809 的 EOC 信号经反相后接到 8031 的 INT1 引脚用于产生转换完成的中断请求信号。0809 芯片的 3 位模拟量输入通道地址码输入端 A、B、C 分别接到 8031 的 P0.0、P0.1 和 P0.2, 故只要向端口地址 0x7fff 分别写入数据 0x00~0x07, 即可启动模拟量输入通道 0~7 进行 A/D 转换。

下面给出采用中断工作方式操作以上接口电路的 C51 程序, 该程序对 8 路模拟输入信号依次进行 A/D 转换, 并把转换结果存入外部 RAM 的 0x10000~0x1007 单元。本例采用指针变量实现对存储器地址的直接操作, 程序中定义了两个指针变量 *ADC 和 *ADCdata, 分别指向 0809 端口地址 0x7fff 和外部 RAM 单元地址 0x1000~0x1007。在 main() 函数中通过赋值语句 “*ADC=i;” 启动 0809 进行 A/D 转换, 转换结束时产生 INT1 中断。在中断服务函数 int1() 中通过赋值语句 “tmp=*ADC;” 和 “*ADCdata=tmp;” 读取 A/D 转换的结果值并存储到外部 RAM 单元中去。

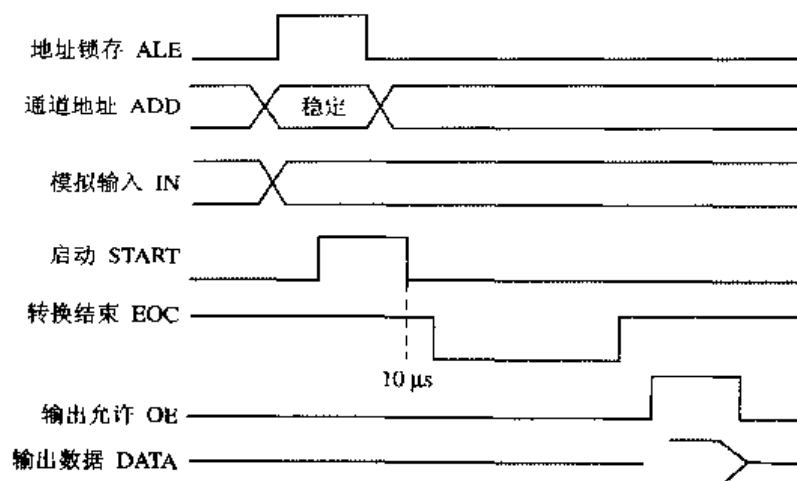


图 8.64 ADC0809 的工作时序

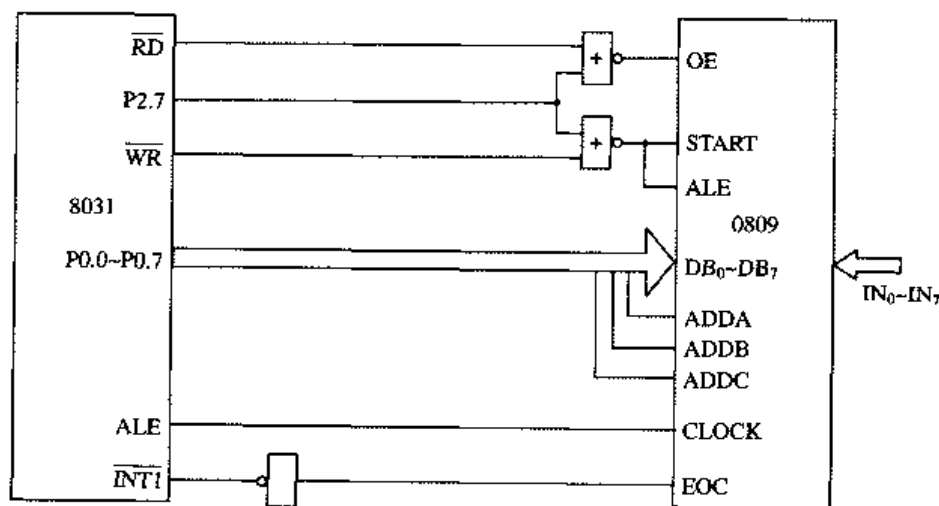


图 8.65 ADC0809 与 8031 单片机的接口电路

采用中断工作方式的 Cx51 程序代码如下:

```
#include <reg51.h>
unsigned char xdata * ADC;          /* 定义 ADC0809 端口指针 */
unsigned char xdata * ADCdata;      /* 定义 ADC0809 数据缓冲器指针 */
unsigned char i;

/***** 主函数 *****/
void main() {
    ADC=0x7fff; ADCdata=0x1000; /* 定义端口地址和数据缓冲器地址 */
    i=8;                        /* ADC0809 有 8 个模拟输入通道 */
    EA=1; EX1=1; IT1=1;        /* 开中断 */
    *ADC=i;                     /* 启动 ADC0809 */
    while(i);                   /* 等待 8 个通道 A/D 转换完毕 */
}
```

```

/***** 外部 INT1 中断服务函数 *****/
void int1() interrupt 2 {
    unsigned char tmp;
    tmp=*ADC;                /* 读取 A/D 转换结果 */
    *ADCdata=tmp;            /* 结果值存储到数据缓冲器 */
    ADCdata++;               /* 数据缓冲器指针地址加 1 */
    i--;
    *ADC=i;                  /* 启动下一个模拟输入通道 A/D 转换 */
}

```

在实际应用中除了可以采用指针变量来实现对内存地址的直接操作之外, Cx51 编译器还提供了一组关于访问绝对地址的预定义宏“ABSACC.H”, 利用它可以十分方便地实现对任意内存空间直接操作, 关于宏“ABSACC.H”的详细描述请参看本书第 9.10.6 节, 采用预定义宏“ABSACC.H”可将上面的程序改写如下:

```

#include <reg51.h>
#include <absacc.h>          /* 包含绝对地址操作预定义头文件 */
#define ADC 0x7fff           /* 定义 ADC0809 端口地址 */
#define ADCdata 0x1000       /* 定义数据缓冲器地址 */
unsigned char i;

/***** 主函数 *****/
void main() {
    i=8;                      /* ADC0809 有 8 个模拟输入通道 */
    EA=1; EX1=1; IT1=1;      /* 开中断 */
    XBYTE[ADC]=i;             /* 启动 ADC0809 */
    while (i);                /* 等待 8 个通道 A/D 转换完毕 */
}

/***** 外部 INT1 中断服务函数 *****/
void int1() interrupt 2 {
    unsigned char tmp;
    tmp=XBYTE[ADC];           /* 读取 A/D 转换结果 */
    i--;
    XBYTE[ADCdata+i]=tmp;     /* 结果值存储到数据缓冲器 */
    XBYTE[ADC]=i;            /* 启动下一个模拟输入通道 A/D 转换 */
}

```

AD574A 也是一种较为常见的逐次逼近式 12 位 ADC 芯片, 它的转换时间为 $25\mu\text{s}$, 转换误差为 $\pm 1\text{LSB}$, 可采用 $+5\text{V}$ 、 $\pm 12\text{V}$ 或 $\pm 15\text{V}$ 电源供电。片内有输出三态缓冲器, 可与 8 位或 16 位微处理器直接相连, 输出数据可以 12 位一起读出, 也可以分成两次读出。输入模拟信号可以是单极性 $0\sim+10\text{V}$, 也可以是双极性 $\pm 5\text{V}$ 或 $\pm 10\text{V}$ 。AD574A 的结构及引脚示于图 8.66。图中 A0 和 12/8 用于控制转换数据是 12 位或 8 位以及数据输出格式。它们的功能见表 8-21。由表 8-21 可见, 在 $\text{CE}=1$ 且 $\text{CS}=0$ (大于 300ns 的脉冲宽度) 时,

才启动转换或读出数据，因此启动转换或读数可用 CE 或 CS 信号来触发。在启动信号有效前，R/C 必须为低电平，否则将产生读数据的操作。启动转换后 STS 引脚变为高电平表示转换正在进行，转换结束后，STS 变成低电平。图 8.67 所示为 AD574A 的工作时序。

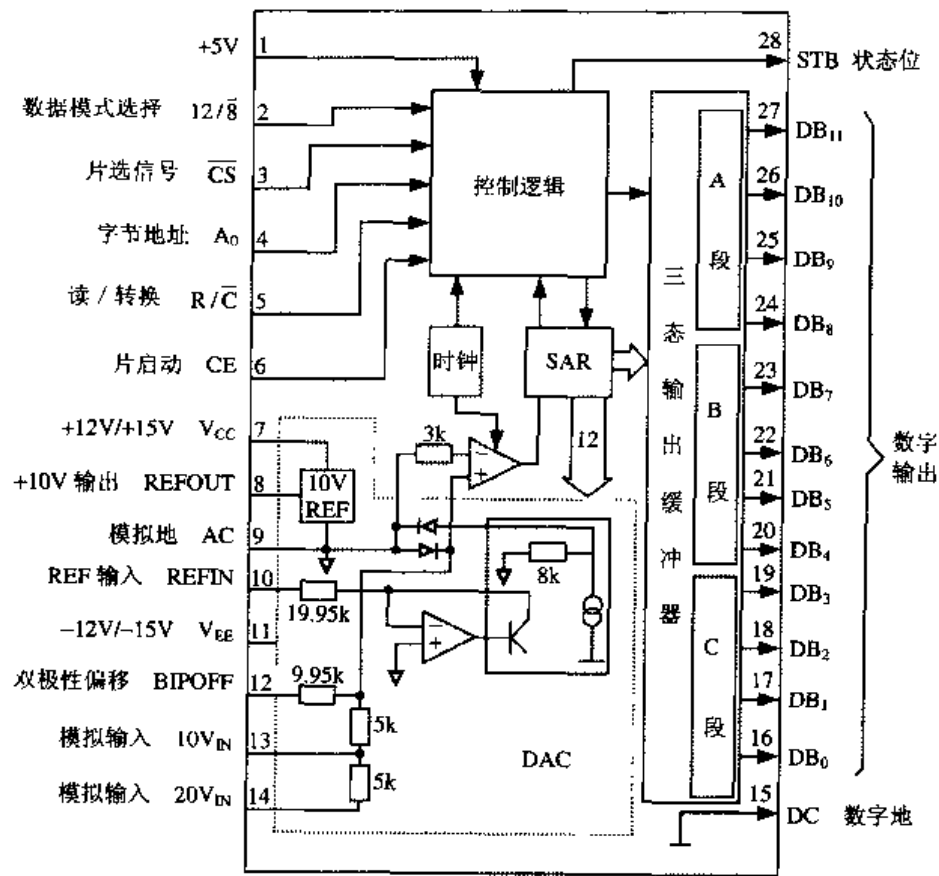


图 8.66 AD574A 的结构图及管脚分配

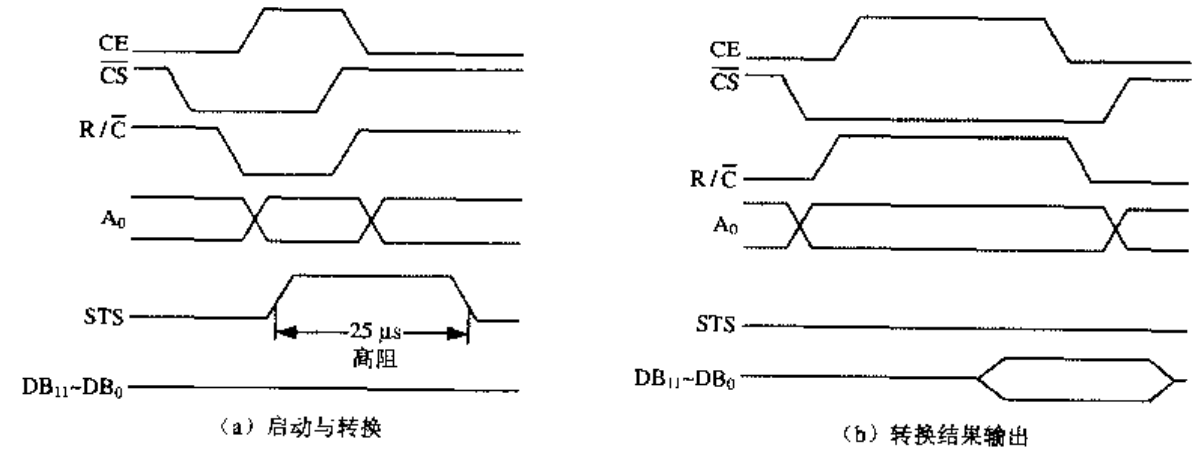


图 8.67 AD574A 的工作时序

表 8-21 AD574 的转换方式和数据输出格式

CE	R/C	CS	12/8	A0	功 能
1	0	0	×	0	12 位 A/D 转换
1	0	0	×	1	8 位 A/D 转换

续表

CE	R/C	CS	12/8	A0	功 能
1	0	1	+5V	×	输出数据格式为并行 12 位
1	0	1	接地	0	从 20~27 脚输出高 8 位数据
1	0	1	接地	1	从 16~19 脚输出低 4 位数据, 从 20~23 脚输出 0

AD574 与单片机 8031 的接口示于图 8.68。AD574A 片内有时钟电路, 不需外加时钟信号。该接口电路采用双极性输入方式, 可对 $\pm 5V$ 或 $\pm 10V$ 的输入电压进行转换。由于 AD574A 输出 12 位数据, 故单片机应分两次读取转换结果数据, 先读高 8 位数据, 再读低 4 位数据。由 $A_0=0$ 或 $A_0=1$ 分别控制高、低位数据的读取。AD574 进行 12 位 A/D 转换的速度很快, 单片机可以采用查寻方式来读取 AD574A 的转换结果数据。采用查询方式时, 应将转换结束状态线 STS 与单片机的某一 I/O 口线相连, 图 9.59 中将 STS 与 8031 的 P1.0 相连。当单片机 8031 执行对外部数据存储器的写指令时, 使 $CE=1$, $CS=0$, $R/C=0$, $A_0=0$, 从而启动转换。然后 8031 通过 P1.0 口线不断查询 STS 的状态, 当 $STS=0$ 时, 表示 A/D 转换结束, 8031 执行两次读取外部数据存储器的操作, 读取 12 位的转换结果。当 $CE=1$, $CS=0$, $R/C=1$, $A_0=0$ 时, 读取转换结果高 8 位; 当 $CE=1$, $CS=0$, $R/C=1$, $A_0=1$ 时, 读取转换结果低 4 位。在图 8.68 中, AD574A 的 CS 端与 8031 的锁存地址 A_7 相连, A_0 端与 8031 的锁存地址 A_1 相连, R/C 与 8031 的锁存地址 A_0 相连, 因此在启动 AD574A 的端口地址中只要 A_7 、 A_1 、 A_0 为 0, 其余位可视具体要求而定。

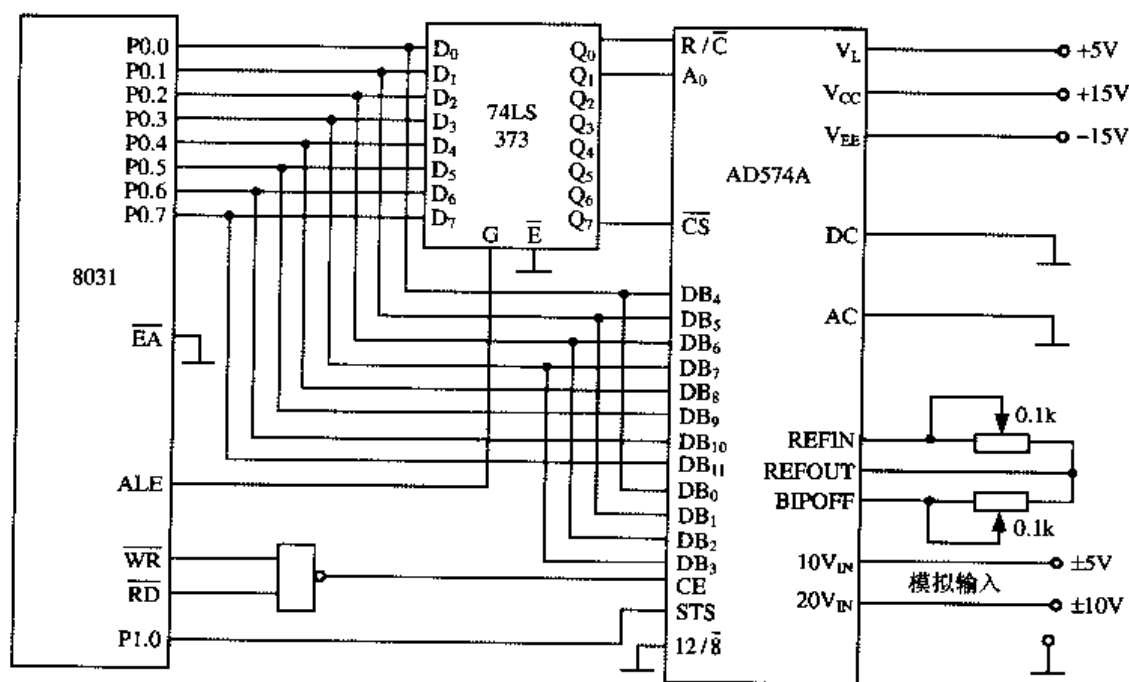


图 8.68 AD574A 与 8031 的接口电路

下面给出对应于图 8.68 接口电路的 Cx51 驱动程序。

```
#include<reg51.h>
#include<absacc.h>
```

```

#define ADCOM XBYTE[0xff7c]    /* 使 A0=0, R/C=0, CS=0 */
#define ADLO  XBYTE[0xff7f]    /* 使 R/C=1, A0=1, CS=0 */
#define ADHI  XBYTE[0xff7d]    /* 使 R/C=1, A0=0, CS=0 */

#define uint  unsigned int

sbit r=P3^7;                    /* RD */
sbit w=P3^6;                    /* WR */
sbit adbusy=P1^0;              /* STS */

/***** 启动 AD574 转换函数 *****/
uint AD574(void) {
    r=0;                        /* 产生 CE=1 */
    w=0;
    ADCOM=0;                    /* 启动 A/D 转换 */
    while(adbusy==1);          /* 等待转换结束 */
    return ((uint)(ADHI<<4)+(ADLO & 0x0f)); /* 返回 12 位 A/D 转换结果 */
}

/***** 主函数 *****/
void main(void) {
    uint idata result;
    result=AD574();             /* 启动 AD574 进行一次转换, 获得 12 位转换结果 */
}

```

8.9.3 用可编程芯片 8155 实现并行 I/O 接口扩展

在对 8051 单片机进行并行 I/O 接口扩展时 Intel 8155 是使用最多的一种芯片, 该芯片内集成有 256 字节的静态 RAM, 二个可编程的 8 位并行 I/O 口 PA、PB, 一个可编程的 6 位并行 I/O 口 PC, 一个 14 位的定时器/计数器。

图 8.69 所示为 8155 的引脚排列图, 各引脚功能如下。

- AD₀~AD₇: 地址数据线, 单片机与 8155 之间的地址、数据、命令及状态信息都通过它们传送。
- ALE: 地址锁存信号输入线。ALE 的下降沿将单片机 8031P0 口输出的地址信号以及 CE、IO/M 状态都锁存到 8155 的内部锁存器中。
- CE、IO/M: 分别为片选信号和 RAM/IO 选择线。当 CE=0、IO/M=0 时单片机对 8155 的 RAM 进行读写; 当 CE=0、IO/M=1 时, 单片机对 8155 的 I/O 口进行读写。
- RD、WR: 分别为读选通信号输入线和写选通信号输入线。
- TIMERIIN: 8155 内部定时器 / 计数器的输入线。
- TIMERIOUT: 8155 内部定时器 / 计数器的输出线。

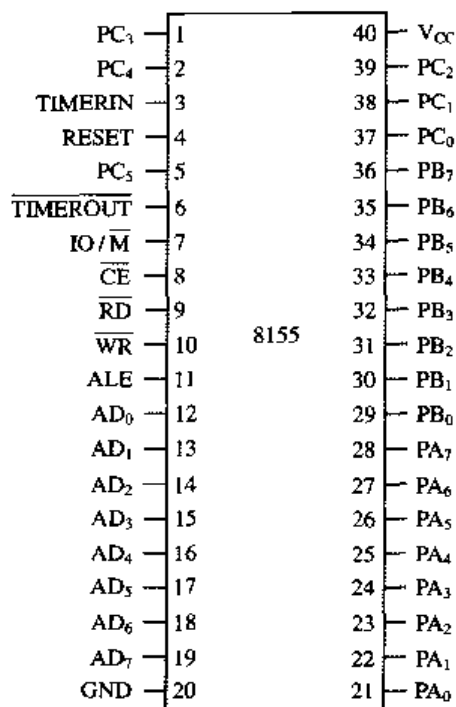


图 8.69 8155 的引脚排列

8155 在与单片机接口时是按片外数据存储单元统一编址的，为 16 位地址，其高 8 位由片选线 CE 提供，低 8 位为片内地址。内部 I/O 口及定时器的低 8 位编址如表 8-22 所示。

表 8-22 8155 的 I/O 口编址

A7	A6	A5	A4	A3	A2	A1	A0	I/O 口
×	×	×	×	×	0	0	0	命令状态寄存器
×	×	×	×	×	0	0	1	PA 口
×	×	×	×	×	0	1	0	PB 口
×	×	×	×	×	0	1	1	PC 口
×	×	×	×	×	1	0	0	定时器低 8 位寄存器
×	×	×	×	×	1	0	1	定时器高 6 位和方式 (2 位) 寄存器

8155 内部的命令寄存器和状态寄存器共用同一个端口地址，命令寄存器只能写入不能读出，状态寄存器只能读出不能写入。8155 I/O 口的工作方式由单片机写入命令寄存器的控制字确定。命令字格式如图 8.70 所示。

命令字低 4 位定义 A 口、B 口和 C 口的工作方式，D₄、D₅ 位确定 A 口、B 口以选通输入输出方式工作时是否允许申请中断，D₆、D₇ 位为 8155 片内定时器的运行控制位。I/O 口的工作方式如下。

- 8155 编程为 ALT1、ALT2 时，A、B、C 口均工作于基本输入输出方式。
- 8155 编程为 ALT3 时，A 口定义为选通输入输出方式，B 口定义为基本输入输出方式。
- 8155 编程为 ALT4 时，A 口和 B 口均定义为选通输入输出工作方式。

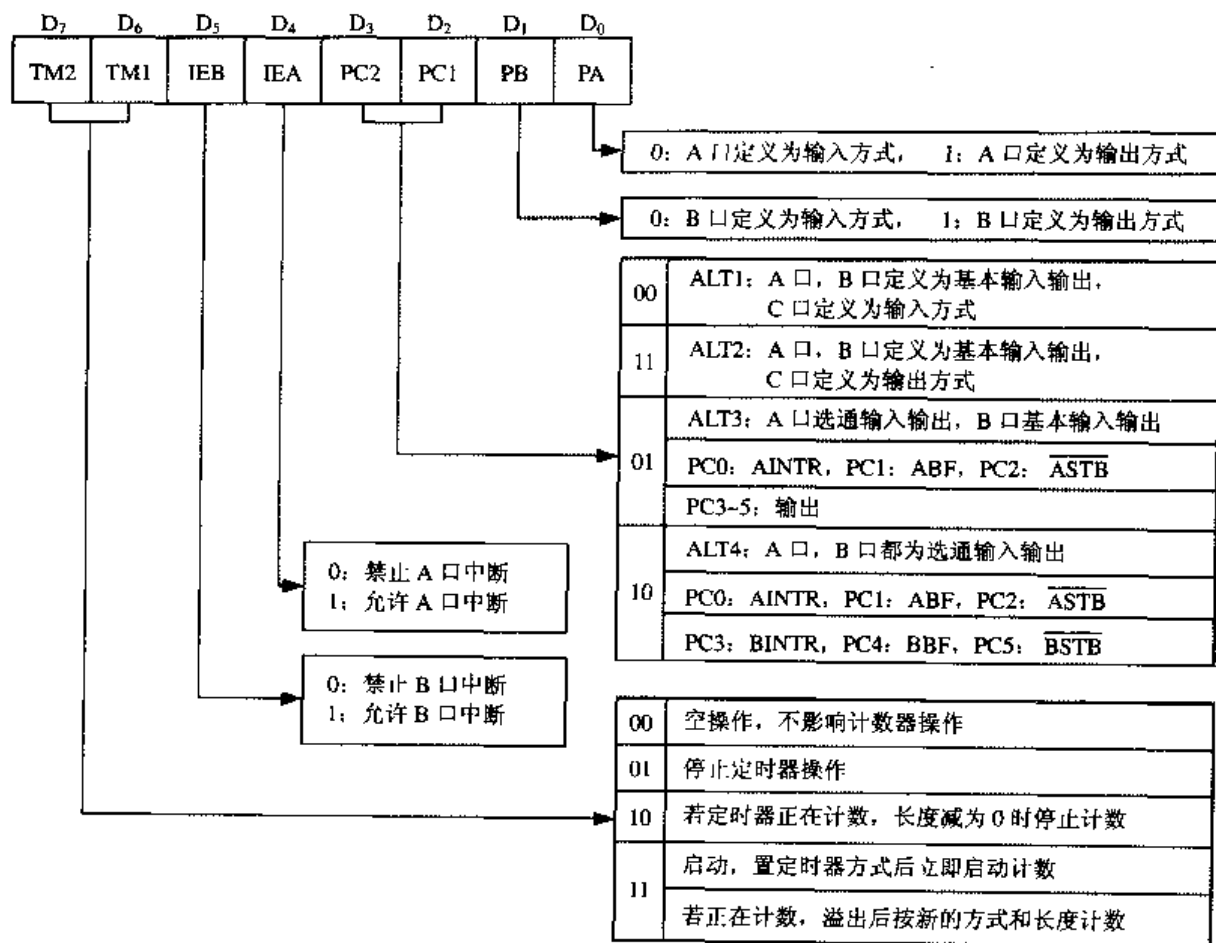


图 8.70 8155 的命令字格式

8155 的状态寄存器用来锁存输入输出和定时器的当前状态，供 CPU 查询。状态寄存器格式如图 8.71 所示。

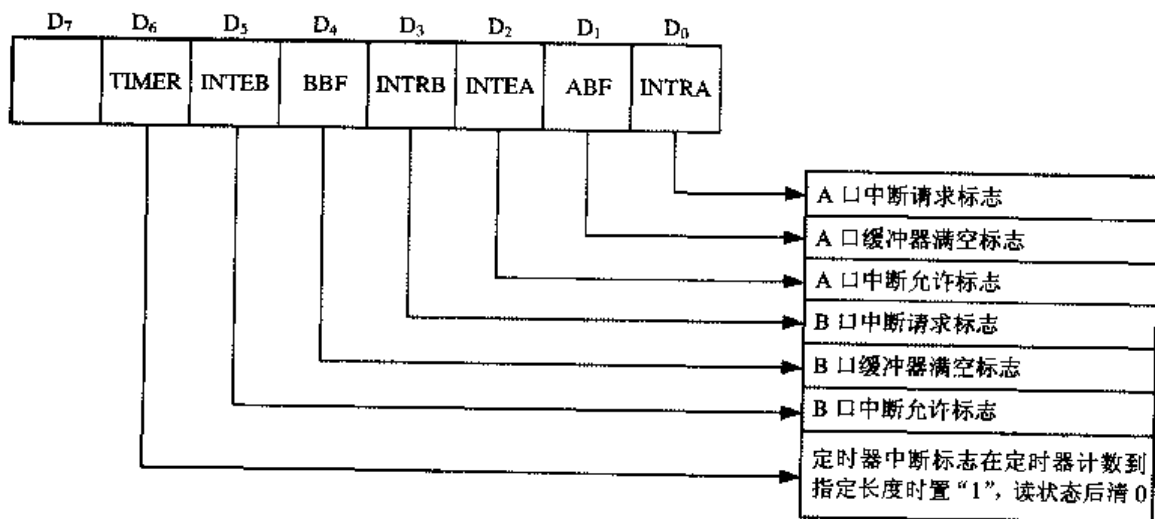


图 8.71 8155 状态寄存器的格式

8155 片内定时器实际是一个 14 位的减法计数器，由两个字节组成，其格式如图 8.72 所示。定时器有四种工作方式，由 M2、M1 两位确定，每一种工作方式的输出波形如图

8.73 所示。对定时器进行编程时,先要将计数常数和定时器工作方式送入定时器口地址(定时器低 8 位及定时器高 6 位、定时器方式 M)。计数常数在 2~3FFFH 之间选择。定时器的启动和停止由命令寄存器的最高两位控制。

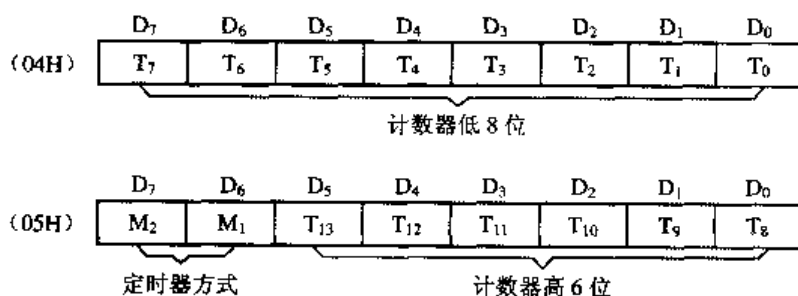


图 8.72 8155 定时器的格式

M ₂	M ₁	方式	定时器输出波形
0	0	单方波	
0	1	连续方波	
1	0	单脉冲	
1	1	连续脉冲	

图 8.73 8155 的定时方式和输出波形

任何时候都可以设置定时器的长度和工作方式,然后将启动命令写入命令寄存器中,即使计数器在计数期间,在写入启动命令后仍可改变其工作方式。如果写入定时器的计数常数值为奇数,则输出的方波不对称。8155 复位后并不预置定时器工作方式和计数常数值。若作为外部事件计数,由计数器状态求取外部输入事件脉冲的方法如下:

停止计数器计数,分别读取计数器的两个字节,取低 14 位计数值,若为偶数,右移一位即为外部输入事件的脉冲数;若为奇数,则右移一位后再加上计数初值的二分之一的整数部分。

下面介绍一个以 8155 芯片进行 I/O 扩展实现矩阵键盘和 LED 显示器接口的例子,并给出基于该接口实现日历时钟的 C51 程序例子。图 8.74 所示为该键盘显示器接口电路。

采用 8155 扩展一个 8 位 LED 显示器和 2×8 矩阵键盘的并行 I/O 接口。8155 的命令口地址为 7F00H,输入输出接口 PA~PC 的口地址分别为 7F01H~7F03H。另外还使用了 8155 内部 256 个字节的 RAM,其地址为 7E00H~7EFFH,该地址在程序中被定义成 PDATA 存储器,因此应对 Cx51 编译器的启动程序 STARTUP.A51 作如下修改:

```

PDATASTART    EQU  07E00H    ; 定义 PDATA 存储器绝对起始地址
PDATALEN      EQU  0FFH      ; 定义 PDATA 存储器空间大小(字节数)
PPAGEENABLE    EQU  1        ; 定义允许使用 PDATA 存储器的使能标志
PPAGE          EQU  07EH      ; 定义 PDATA 存储器的页地址

```

同时,还需要在 μVision2 环境的“Project 菜单 / Options for Target 选项 / BL51 Locate

标签页 / Pdata” 对应的 “Base” 框中键入 PDATA 基地址 0x7e00。

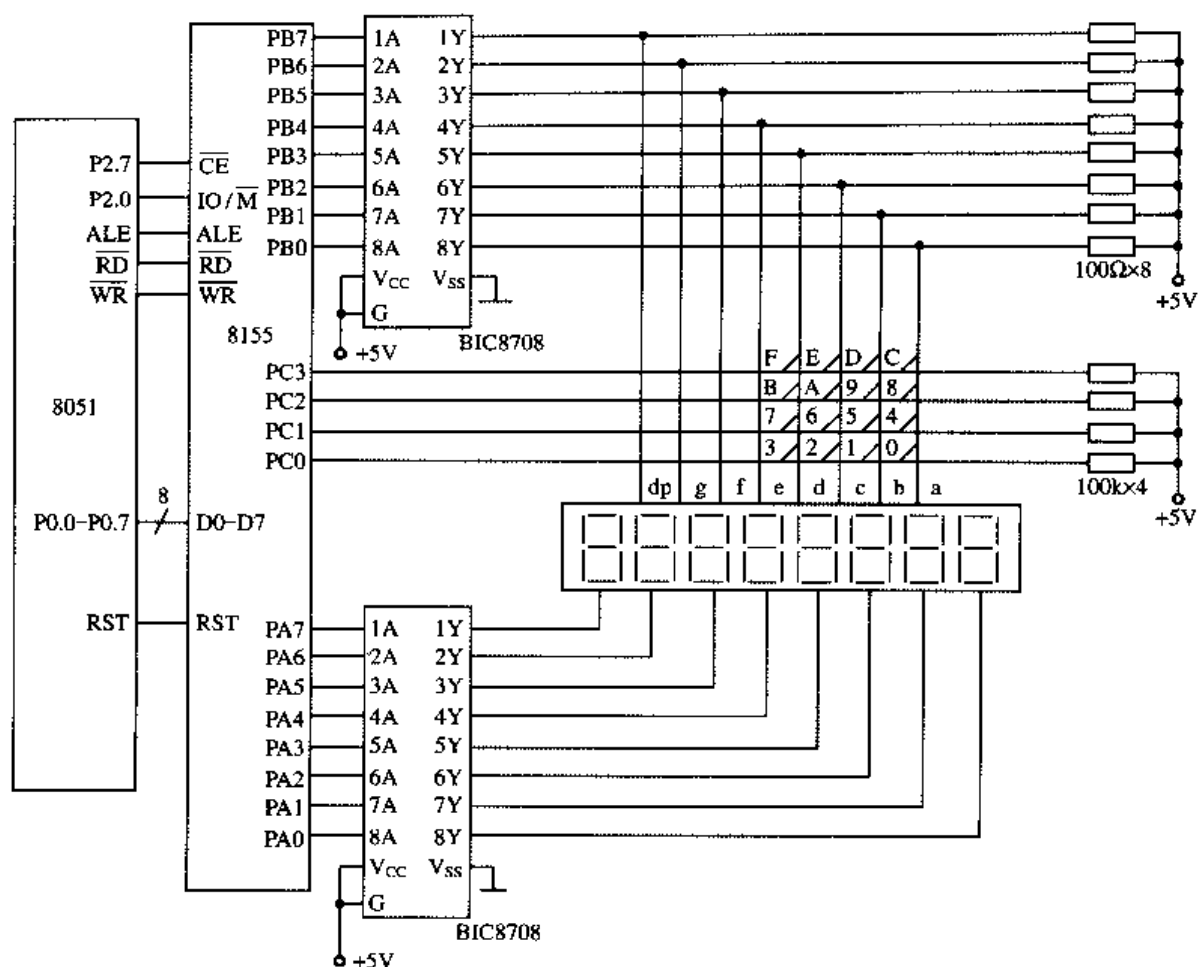


图 8.74 用并行接口扩展芯片 8155 实现的矩阵键盘和 LED 接口电路

本例由两个模块文件组成：Cx51 程序文件 MOD1.C 和汇编程序文件 MOD2.A51。文件 MOD1.C 中包括全局变量定义，主函数 main()，系统初始化函数 init_sys()，功能调度函数 monitor()，定时器 T0 中断服务函数 timer0()，矩阵键盘驱动函数 kbhit() 和 get_ch()，LED 显示驱动函数 set_led_buf()、put_off_leds() 和 put_on_leds() 以及初始化时间设置与校时函数 set_date_time()，当前日期输出函数 ask_date()，字符串输入函数 get_str() 和错误信息输出函数 error_message()。文件 MOD2.A51 是一个用汇编语言编写的延时程序，它用于键盘消颤和动态显示延时，采用汇编语言可实现较为精确的软件延时。

程序提供初始化时间设置、自动计算时间、日历、动态显示时间、查询日期和校时功能。系统采用 6MHz 的晶振，用定时器 T0 提供计时基准，每隔 50ms 产生一次定时中断，由软件完成时间和日期的计算，并控制动态扫描显示。采用 6MHz 晶振，定时器 T0 设置为工作方式 0，可计算出产生 50ms 定时所对应的初始时间常数为 9E58H，但由于软件重装时间常数会影响定时精度，若采用该计算值作为初始时间常数，24 小时将产生大约 2 分钟的累积误差，这个误差显然太大。通过实验确定使用初始值 9E75H 可使累积误差减少到 24 小时不大于 3 秒钟。

程序运行时，LED 数码管将首先显示 “———” 提示输入基准时间，按任

意键后可按“××××××××××××××××”格式输入基准日期和时间,例如,输入“20030806110825”表示2003年8月6日11点8分25秒。输入过程中LED自右向左滚动显示。如果输入的是合法的日期时间,系统即开始正常工作,并按“××-××-××”格式动态显示当前时间,否则重新显示提示符并等待输入合法的时间日期值。

程序在正常运行过程中,可按“0”键来修改日期时间(校时功能),修改值的输入方法与基准值输入方法相同。若输入正确则以新输入的日期时间作为基准重新开始计时,否则不作任何修改并显示“ERROR”信息10秒后自动恢复以前的动态时间显示。按“1”键显示当前日期,显示格式为“××-××-××”,10秒后自动恢复时间动态显示。在显示日期或错误信息“ERROR”时,按任意键(除“0”和“1”键之外)可提前恢复时间动态显示。

模块文件MOD1.C代码如下:

```
#include <reg51.h>
#include <absacc.h>
#include <string.h>
#include <intrins.h>
#include <ctype.h>
#include <stdlib.h>
#define Uchar unsigned char
#define Uint unsigned int
#define Ulong unsigned long

/* 定义8155的I/O端口地址 */
#define P8155CW 0x7f00          /* 8155 命令口地址 */
#define P8155IA 0x7f01         /* 8155 的 PA 口地址 */
#define P8155IB 0x7f02         /* 8155 的 PB 口地址 */
#define P8155IC 0x7f03         /* 8155 的 PC 口地址 */

/* 定义定时器 T0 的时间常数值和方式控制字 */
#define V_TH0 0x9e              /* 时间常数高 8 位 */
#define V_TL0 0x75              /* 时间常数低 8 位 */
#define V_TMOD 0x01             /* 定时器 T0 方式控制字 */

/* 定义 LED 显示字符段码 */
static struct {
    Uchar ascii;
    Uchar stroke;
} code led_strokes[27] =\
{{'0',0x3f},{'1',0x06},{'2',0x5b},{'3',0x4f},{'4',0x66},\
{'5',0x6d},{'6',0x7d},{'7',0x07},{'8',0x7f},{'9',0x6f},\
{'A',0x77},{'B',0x7c},{'C',0x39},{'D',0x5e},{'E',0x79},\
{'F',0x71},{'H',0x76},{'O',0x5c},{'P',0x73},{'U',0x3e},\
```

```
{'R',0x50}, {'Y',0x6e}, {'.',0x80}, {'-',0x40}, {'=',0x48}, \
{0x00,0x00}, {0xff,0xff}};

/* 定义非闰年每月的天数 */
Uchar code days_month[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};

struct TIME {                                /* 定义时间结构 */
    Uchar sec;
    Uchar min;
    Uchar hour;
};

struct DATE {                                /* 定义日期结构 */
    Uchar year;
    Uchar month;
    Uchar day;
};

struct TIME time;
struct DATE date;

/* 定义一组全局变量和函数原型 */
Uchar bdata flag;
sbit time_init=flag^0;
sbit auto_flush=flag^1;
sbit message_flag=flag^2;
Uint message_time;
Uchar led_buf[8];
extern void delay(Uint);
void led_buf_auto_flush(void) reentrant;
bit leap_year(void) reentrant;
void init_sys(void);
void monitor(void);
void set_led_buf(Uchar,Uchar,Uchar);
Uchar get_strokes(Uchar);
void put_on_leds(void);
void put_off_leds(void);
bit kb_hit(void);
Uchar get_ch(void);
Uchar * get_str(Uchar *, Uint);
void ask_date(void);
bit set_date_time(void);
void error_message(void);
```



```

/***** 主函数 *****/
*   函数原型: main();
*   功    能: 调用 init_sys() 函数对系统进行初始化, 调用 monitor() 函数对
*             用户输入的键盘命令进行解释。
/*****/
main() {
    init_sys();
    monitor();
}

/***** 系统初始化函数 *****/
*   函数原型: void init_sys(void);
*   功    能: 对系统进行初始化并接受用户的初始化日期时间设置。
/*****/
void init_sys(void) {
    /* 8155 初始化 */
    XBYTE[P8155CW]=0x03; XBYTE[P8155IA]=0xff; XBYTE[P8155IB]=0xff;
    /* 定时器 T0 初始化 */
    TMOD=V_TMOD; TH0=V_TH0; TL0=V_TL0;
    TR0=1; ET0=1; EA=1;
    /* 标志变量初始化 */
    time_init=0; auto_flush=0; message_flag=0; message_time=0;
    /* 显示 8 个 "-" 直到用户按下 "0" 键并成功设置初始化日期时间为止 */
    set_led_buf('-', 0, 8);
    while(1) {
        put_on_leds();
        if(!get_ch()) continue;
        if(!set_date_time())
            set_led_buf('-', 0, 8);
        else
            break;
    }
}

/***** 运行监控函数 *****/
*   函数原型: void monitor(void);
*   功    能: 用当前的时、分、秒值自动刷新 LED 显示, 并对用户按键进行解释。
*             "0" 键表示校正日期和时间, 如果校时成功则动态显示时间, 否则
*             显示出错信息 "ERROR"。"1" 键表示查询并显示当前日期。在显示出
*             错信息 "ERROR" 期间, 按任意键可清除显示, 否则 10 秒后自动恢复
*             动态时间显示。
/*****/

```

```

void monitor(void) {
    Uchar command;
    while(1) {
        put_on_leds();                /* 显示缓冲区当前内容 */
        if(message_time==0) {         /* 如果日期或出错信息显示时间为 0 秒 */
            message_flag=0;           /* 清除错误信息显示标志 */
            auto_flush=1;             /* 设置用当前时间自动刷新缓冲区标志 */
        }
        if((command=get_ch())) {       /* 如果有键按下 */
            message_flag=0;           /* 清除错误信息显示标志 */
            message_time=0;           /* 设置错误信息显示时间为 0 秒 */
        }
        switch(command) {             /* 对键盘命令进行解释 */
            case'0':if(!set_date_time()) /* 如果校时不成功 */
                error_message();      /* 显示错误信息"ERROR" */
                break;
            case'1':ask_date();        /* 查询当前日期 */
                break;
            default : break;
        }
    }
}

```

***** 定时器 T0 中断服务函数 *****

* 函数原型: void timer0(void);
 * 功 能: 每次执行时先重装时间常数, 然后检查日期和时间是否已经初始化, 如果未初始化则立即返回, 否则每中断 20 次(即一秒钟)根据信息显示标志的设置来决定是否将日期或错误信息显示时间减 1, 然后自动修改日期和时间结构。如果缓冲区自动刷新标志为 1, 则用当前时间刷新显示缓冲区。

*****/

```

void timer0(void) interrupt 1 using 2 {
    static Uchar click=0;            /* 中断次数计数器变量 */
    /* 重装定时器 T0 时间常数 */
    TH0=V_TH0; TL0=V_TL0;
    if(!time_init) return;          /* 如果日期时间未初始化, 返回 */
    ++click;
    if(click>=20) {                  /* 间隔 1 秒钟(20*50ms=1s) */
        click=0;
        /* 根据消息显示标志决定是否将消息显示时间减少 1 秒 */
        if(message_flag && message_time) --message_time;
        /* 计算并修改时间和日期 */
        if(++time.sec>=60) {

```

```

    time.sec=0;
    if(++time.min>=60) {
        time.min=0;
        if(++time.hour>=24) {
            time.hour=0;
            if(++date.day>days_month[date.month]) {
                if(date.month==2 && leap_year());
                else date.day=1;
            }
            if(++date.month>12) {
                date.month=1;
                ++date.year;
            }
        }
    }
}

/* 如果已设置缓冲区自动刷新标志, 则用当前时间刷新显示缓冲区 */
if(auto_flush) led_buf_auto_flush();
}

/***** 显示缓冲区更新函数 *****/
*   函数原型: void led_buf_auto_flush(void) ;
*   功    能: 用当前时间的时、分、秒值填充显示缓冲区。
*****/
void led_buf_auto_flush(void) reentrant {
    led_buf[0]=time.sec%10+0x30;
    led_buf[1]=time.sec/10+0x30;
    led_buf[2]='-';
    led_buf[3]=time.min%10+0x30;
    led_buf[4]=time.min/10+0x30;
    led_buf[5]='-';
    led_buf[6]=time.hour%10+0x30;
    led_buf[7]=time.hour/10+0x30;
}

/***** 闰年计算函数 *****/
*   函数原型: bit leap_year(void);
*   功    能: 判断某年是否为闰年, 若是闰年则返回 1, 否则返回 0。
*****/
bit leap_year(void) reentrant {
    if(date.year%4==0 && date.year%100 != 0) return((bit)1);
}

```

```

    if(date.year%400==0) return((bit)1);
    return((bit)0);
}

/***** 显示缓冲区填充函数 *****/
*   函数原型: void set_led_buf(Uchar c, Uchar pos, Uchar cnt);
*   功    能: 从显示缓冲区的 pos 位置开始, 用字符 c 填充 cnt 个字节。
*****/
void set_led_buf(Uchar c,Uchar pos, Uchar cnt) {
    Uchar ledbuf_pos;
    auto_flush=0;                      /* 在填充期间关闭 LED 显示器 */
    for(ledbuf_pos=pos; cnt>0; cnt--) {
        led_buf[ledbuf_pos]=((islower(c)) ? (toupper(c)) : (c));
        ledbuf_pos++;
        if(ledbuf_pos>=8) ledbuf_pos=0;
    }
}

/***** 段码查找函数 *****/
*   函数原型: Uchar get_strokes(Uchar c);
*   功    能: 从 LED 显示段码表中查找并返回字符 c 的字型段码。
*****/
Uchar get_strokes(Uchar c) {
    Uchar i=0;
    while(led_strokes[i].ascii != c) i++;
    return(led_strokes[i].stroke);
}

/***** 显示输出函数 *****/
*   函数原型: void put_on_leds(void);
*   功    能: 将当前显示缓冲区的字符输出到 LED 显示器上。
*****/
void put_on_leds(void) {
    Uchar dmask=0xfe;                  /* 显示位控制码 */
    Uchar pos;
    for(pos=0; pos<8; pos++) {
        if(led_buf[pos]==0x00) put_off_leds();    /* 关显示 */
        else {
            XBYTE[P8155IB]=get_strokes(led_buf[pos]); /* 输出字型码 */
            XBYTE[P8155IA]=dmask;                    /* 输出位控制码 */
        }
        delay(1);                                  /* 延时 1ms */
        dmask=_crol_(dmask,1);                     /* 修改位控制码 */
    }
}

```

```

    }
}

/***** 关闭显示函数 *****/
*   函数原型: void put_off_leds(void);
*   功    能: 关闭 LED 显示器。
*****/
void put_off_leds(void) {
    XBYTE[P8155IA]=0xff;
    XBYTE[P8155IB]=0xff;
}

/***** 判键按下函数 *****/
*   函数原型: bit kb_hit(void);
*   功    能: 判断是否有键被可靠地按下, 有则返回 1, 否则返回 0。
*****/
bit kb_hit(void) {
    put_off_leds();           /* 关显示 */
    XBYTE[P8155IB]=0x00;      /* 往矩阵键盘列线送低电平 */
    if((XBYTE[P8155IC] & 0x3f)==0x3f) /* 输入并检测行线状态 */
        return((bit)0);      /* 行线为高电平时返回 0 */
    delay(8);                 /* 延时 8ms 消颤 */
    if((XBYTE[P8155IC] & 0x3f)==0x3f) /* 再次输入并检测行线状态 */
        return((bit)0);      /* 行线为高电平时返回 0 */
    return((bit)1);           /* 有键可靠按下时返回 1 */
}

/***** 等待按键函数 *****/
*   函数原型: Uchar get_ch(void);
*   功    能: 等待用户按键, 从矩阵键盘上输入一个 ASCII 字符, 若输入成功
*           则返回所输入的字符, 否则返回 0。
*****/
Uchar get_ch(void) {
    Uchar row=0, col=0;
    Uchar mask=0xfe;
    Uchar pic;
    if(!kb_hit()) return(0);   /* 无键按下, 输入不成功, 返回 0 */
    /* 分析按键所在的列号 */
    XBYTE[P8155IB]=mask;
    while((XBYTE[P8155IC] & 0x3f)==0x3f && mask>0xef) {
        ++col;
        mask=_crol_(mask,1);
        XBYTE[P8155IB]=mask;
    }
}

```

```

    }
    /* 分析按键所在的行号 */
    pic=XBYTE[P8155IC] & 0x3f;
    mask=0x01;
    while(pic & mask) {
        ++row;
        mask=_crol_(mask,1);
    }
    /* 等待按键释放 */
    while((XBYTE[P8155IC] & 0x3f)!=0x3f);
    XBYTE[P8155IB]=0xff;
    /* 计算按键序号并将其转换成 ASCII 码值返回 */
    pic=row*4+col;
    if(pic<10) pic+='0';
    else      pic+='A'-10;
    return(pic);
}

/***** 取键盘输入字符串函数 *****/
*   函数原型: * get_str(Uchar * str, Uchar len);
*   功    能: 从矩阵键盘输入长度为 len 的字符串, 并将其存储在由指针 str 所
*               指向的存储器空间, 并返回该指针。在输入的同时滚动显示所输
*               入的字符。
*****/
Uchar * get_str(Uchar * str, Uint len) {
    Uchar pdata i;
    Uchar pdata ch;
    Uchar pdata keyboard_buf[14];          /* 定义键盘缓冲区 */
    Uchar pdata ledbuf_pos=0, keybuf_pos=0; /* 消除显示缓冲区 */
    set_led_buf(0x00,0,8);
    while(keybuf_pos<len) {
        put_on_leds();
        ch=get_ch();                        /* 输入一个字符 */
        /* 如果成功, 将显示缓冲区原有字符向左滚动一位 */
        /* 并将新字符放在显示缓冲区末尾, 并送键盘缓冲区 */
        if(ch) {
            for(i=ledbuf_pos; i>0; i--)
                led_buf[i]=led_buf[i-1];
            led_buf[0]=ch;
            if(++ledbuf_pos>7) ledbuf_pos=7;
            keyboard_buf[keybuf_pos++]=ch;
        }
    }
}

```

```

    for(i=0; i<255; i++) put_on_leds();          /* 短时显示最后输入的字符 */
    memcpy(str, keyboard_buf, len);                /* 拷贝键盘缓冲区中的输入串 */
    return(str);
}

/***** 设置日期时间函数 *****/
*   函数原型: bit set_date_time(void);
*   功    能: 输入一个完整的表示日期和时间的字符串, 分析其合法性, 并将其
*               转换成年、月、日、时、分、秒值后存储到相应的结构变量中, 然
*               后启动计时和动态时间显示。如果成功则返回 1, 否则返回 0。
*****/
bit set_date_time(void) {
    Uchar pdata str[14]; Uchar pdata ltime[5]; Uint pdata lyear;
    Uchar pdata lmonth, lday, lhour, lmin, lsec;
    get_str(str, 14);
    for(lsec=0; lsec<14 && isdigit(str[lsec]); lsec++);
    if(lsec!=14) return((bit)0);                  /* 日期时间字符串长度错, 返回 0 */
    /* 下列语句从日期和时间字符串中分离出年月日和时分秒子字符串,
       判断它们的合法性, 如果合法则转换成对应的整数值, 否则返回 0 */
    memcpy(ltime, str, 4);
    ltime[4]=0;
    if(strcmp(ltime, "1995")<0) return((bit)0);
    lyear=atoi(ltime);
    memcpy(ltime, str+4, 2);
    ltime[2]=0;
    if(strcmp(ltime, "01")<0 || strcmp(ltime, "12")>0) return((bit)0);
    lmonth=atoi(ltime);
    memcpy(ltime, str+6, 2);
    ltime[2]=0;
    lday=atoi(ltime);
    if(lmonth==2 && (lyear%4==0 && lyear%100 != 0 || lyear%400==0))
        if(lday<1 || lday>29) return((bit)0);
    if(lday<1 || lday>days_month[lmonth]) return((bit)0);
    memcpy(ltime, str+8, 2);
    ltime[2]=0;
    if(strcmp(ltime, "23")>0) return((bit)0);
    lhour=atoi(ltime);
    memcpy(ltime, str+10, 2);
    ltime[2]=0;
    if(strcmp(ltime, "59")>0) return((bit)0);
    lmin=atoi(ltime);
    memcpy(ltime, str+12, 2);
    ltime[2]=0;

```

```

if(strcmp(ltime,"59")>0)    return((bit)0);
lsec=atoi(ltime);
/* 将年月日时分秒值转储到相应的结构变量中去, */
/* 启动计时并设置日期时间初始化和动态显示刷新标志 */
TR0=0;
date.year=lyear;
date.month=lmonth;
date.day=lday;
time.hour=lhour;
time.min=lmin;
time.sec=lsec;
TMOD=V_TMOD;
TL0=V_TL0;
TH0=V_TH0;
TR0=1;
time_init=1;
auto_flush=1;
return((bit)1);
}

```

/* ***** 日期数据转换函数 ***** */

* 函数原型: void ask_date(void);

* 功 能: 将日期结构变量的年月日整数值转换成 ASCII 码字符后拷贝到显示缓冲区并清除显示缓冲区自动刷新标志。

*****/

```

void ask_date(void) {
    Uchar s[8];
    /* 将日期结构转换成日期字符串 */
    s[0]=date.day%10+0x30;
    s[1]=date.day/10+0x30;
    s[2]='-';
    s[3]=date.month%10+0x30;
    s[4]=date.month/10+0x30;
    s[5]='-';
    s[6]=date.year%10+0x30;
    s[7]=(date.year/10)%10+0x30;
    auto_flush=0;                /* 清除显示缓冲区自动刷新标志 */
    memcpy(led_buf,s,8);        /* 将日期字符串拷贝到显示缓冲区 */
    message_time=10;            /* 将日期字符串显示时间置为 10 秒 */
    message_flag=1;              /* 设置日期显示标志 */
}

```

/* ***** 出错信息函数 ***** */


```

*   函数原型: void error_message(void);
*   功    能: 将字符串"ERROR"填充到显示缓冲区。
*****/
void error_message(void) {
    auto_flush=0;                /* 清除显示缓冲区自动刷新标志 */
    set_led_buf('R',0,1);
    set_led_buf('O',1,1);
    set_led_buf('R',2,1);
    set_led_buf('R',3,1);
    set_led_buf('E',4,1);
    set_led_buf(0x00,5,3);
    message_time=10;             /* 将出错信息显示时间置为 10 秒 */
    message_flag=1;              /* 设置错误信息显示标志 */
}

```

MOD2.A51 程序代码如下:

```

; /***** 延时函数 *****/
; *   函数原型: void delay(unsigned int count);
; *   功    能: 用软件方法延时 count 个毫秒。
; *****/
NAME DELAY
PUBLIC _DELAY
SEG_DELAY SEGMENT CODE
RSEG SEG_DELAY
USING 0
_DELAY: PUSH ACC
        PUSH AR5
        MOV A,R7
        ORL A,R6
        JZ  RETMAIN
ONE_MS: NOP
        NOP
        NOP
        MOV R5,#246
HERE:   DJNZ R5,HERE
        DEC R7
        MOV A, R7
        JNZ ONE_MS
        ORL A, R6
        JZ  RETMAIN
        DEC R6
        DEC R7
        SJMP ONE_MS

```

```

RETMAIN:POP  AR5
        POP  ACC
        RET
END

```

8.9.4 实时日历/时钟芯片 DS12887 的 Cx51 驱动程序

美国 DALLAS 公司推出的新型时钟日历芯片 DS12C887 能够自动产生年、月、日、时、分、秒等时间信息，芯片内部带有锂电池，外部掉电时，其内部时间信息能够保持 10 年之久；有 12 小时制和 24 小时制两种工作模式；时间的表示方法也有两种，一种用二进制数表示，一种是用 BCD 码表示。此外用户还可对 DS12887 进行编程以实现多种方波输出，并可对其内部的三路中断通过软件进行屏蔽。

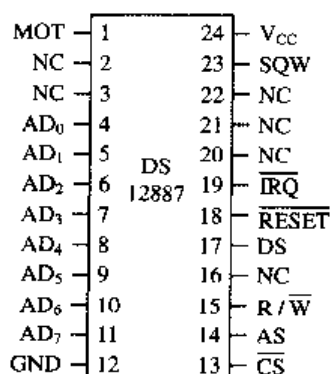


图 8.75 DS12887 的引脚排列

DS12887 的引脚排列如图 8.75 所示，各引脚的功能如下。

- V_{CC}:** 直流电源+5V 输入，当 V_{CC} 输入为+5V 时，用户可以访问 DS12C887 内 RAM 中的数据，并可对其进行读、写操作；当 V_{CC} 的输入小于+4.25V 时，禁止用户对内部 RAM 进行读、写操作，此时用户不能正确获取芯片内的时间信息；当 V_{CC} 的输入小于+3V 时，DS12C887 会自动将电源切换到内部自带的锂电池上，以保证内部的电路能够正常工作。
- GND:** 地。
- MOT:** 总线模式选择，当 MOT 接 V_{CC} 时选用 MOTOROLA 总线模式，当 MOT 接 GND 时选用 Intel 总线模式。
- SQW:** 方波输出，当供电电压 V_{CC} 大于 4.25V 时，SQW 脚可进行方波输出，此时用户可以通过对控制寄存器编程来得到 13 种方波信号的输出。
- AD₀~AD₇:** 双向地址/数据总线。
- AS:** 地址有效输入。
- DS/RD:** 数据选通/读允许，当 MOT 接 V_{CC} 时作为数据选通；当 MOT 接 GND 时，作为读允许输入（即 RD）。
- R/W:** 读 / 写允许输入，若 MOT 接 V_{CC}，该引脚为高电平时为读操作，为低电平时为写操作；若 MOT 接 GND，该引脚作为写允许输入（即 WR）。

- CS: 片选输入, 低电平有效。
- IRQ: 中断请求输出, 低电平有效。
- RESET: 复位端, 可以直接接 V_{CC} , 这样可以保证 DS12C887 在掉电时, 其内部控制寄存器不受影响。
- NC: 空引脚。

DS12887 带有 128 字节片内 RAM, 其中 10 字节用来存储时间信息, 称为时标寄存器, 地址为 00H~09H, CPU 可以通过读取时标寄存器获得时间与日历值, 也可以编程设置其初值, 时标寄存器的值可以用二进制或 BCD 码表示。4 字节用来存储控制信息, 称为控制寄存器, 地址为 0AH~0DH, 用户可通过对控制寄存器编程实现从 SQW 引脚输出多种不同频率的方波, 并可对其内部的三路中断通过软件进行屏蔽。114 字节作为通用 RAM 供用户使用, 地址为 0EH~7FH。DS12887 片内 RAM 与寄存器地址分配如图 8.76 所示。

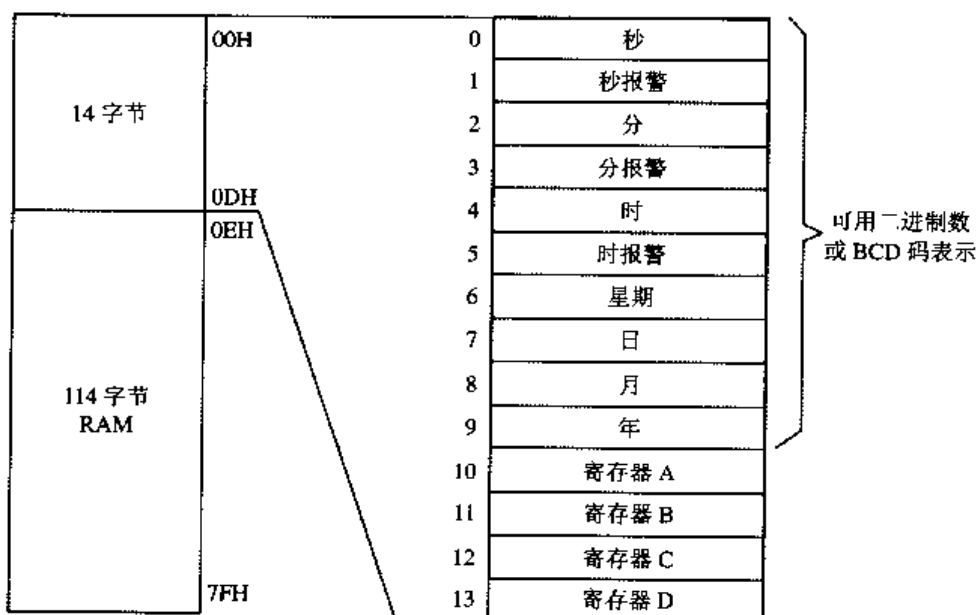


图 8.76 DS12887 片内 RAM 与寄存器地址分配

寄存器 A 主要用于选择时钟频率、中断周期和 SQW 输出频率, 格式如下:

D7	D6	D5	D4	D3	D2	D1	D0
UIP	DV2	DV1	DV0	RS3	RS2	RS1	RS0

UIP: 更新周期标志。UIP=1 时, 芯片正处于或即将开始更新周期, 在此期间不允许读写时标寄存器。

DV2~DV0: 芯片内部振荡器 RTC 控制位。当芯片解除复位状态, 并将 010 写入 DV0、DV1、DV2 后, 另一个更新周期将在 500ms 后开始。因此, 在程序初始化时可用这三位精确地使芯片在设定的时间开始工作。

RS3~RS0: 周期中断可编程方波输出速率选择位。这 3 位不同的组合可以产生不同的方波输出, 程序可以通过设置寄存器 B 的 SQWF 和 PIE 位控制是否允许周期中断方波输出。DS12887 固定使用 32.768kHz 的内部晶体, RS3~RS0 与中断周期和 SQW 输出频率的对应关系如表 8-23 所示。

表 8-23 时钟频率选择

寄存器 A 选择位				32.768kHz 时基	
RS3	RS2	RS1	RS0	中断周期	SQW 输出频率
0	0	0	0	—	—
				3.09625 ms	256 Hz
				7.8125 ms	128 Hz
				122.070 ms	8.192 kHz
				244.141 ms	4.095 kHz
				488.281 ms	2.048 kHz
				976.562 ms	1.024 kHz
				1.953125 ms	512 Hz
				3.90625 ms	256 Hz
				7.8125 ms	128 Hz
				15.625 ms	64 Hz
				31.25 ms	32 Hz
				62.85 ms	16 Hz
				125 ms	8 Hz
				250 ms	4 Hz
				500 ms	2 Hz

寄存器 B 主要用于设置芯片的工作状态，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
SET	PIE	AIE	UIE	AQWE	DM	24/12	DSE

SET：当该位为“0”时，芯片处于正常工作状态，每秒产生一个更新周期来更新时标寄存器，当该位为“1”时，芯片停止工作，程序在此期间可初始化芯片的各个时标寄存器。

PIE、AIE、UIE：分别为周期中断、报警中断、更新周期结束中断允许位。各位分别为“1”时允许发相应的中断。

SQWE：方波输出允许位。SQWE=1，按寄存器 A 输出速率选择位所确定的频率输出方波；SQWE=0，SQW 脚保持低电平。

DM：时标寄存器用 BCD 码表示或用二进制表示的格式选择位。DM=0，为 BCD 码；DM=1，为二进制码。

24/12：24 小时或 12 小时模式设置位。24/12=1，选择 24 小时工作模式；24/12=0，选择 12 小时工作模式。

DSE：夏令时服务位。DSE=“1”时，夏时制设置有效，夏时制结束可自动刷新恢复时间；DSE=0 无效。

寄存器 C 为中断标志位寄存器，该寄存器的特点是程序访问读寄存器后，该寄存器的

内容将自动清零,从而使 IRQF 标志位变为高电平,否则,芯片将无法向 CPU 申请下一次中断。格式如下:

D7	D6	D5	D4	D3	D2	D1	D0
IRQF	PF	AF	UF	0	0	0	0

IRQF: 中断申请标志位。该位逻辑表达式为: $IRQF=PF*PIE+AF*AIE+UF*UIE$ 。当 IRQF=1 时, IRQ 引脚将变低电平引发中断申请。

PF、AF、UF: 这三位分别为周期中断、报警中断、更新周期结束中断标志位。只要满足各中断的条件,相应的中断标志位将置“1”。

寄存器 C 的 D3~D0 位为 0。

寄存器 D 为状态标志寄存器,只有一个标志位 VRT (D7),其余各位均为 0。格式如下:

D7	D6	D5	D4	D3	D2	D1	D0
VRT	0	0	0	0	0	0	0

VRT: 芯片内部 RAM 与寄存器内容有效标志位。该位为“1”时,表示芯片内部 RAM 和寄存器内容有效。读该寄存器后,该位将自动置“1”。

DS12887 处于正常工作状态时,每秒钟将产生一个更新周期,芯片处于更新周期的标志是寄存器 A 中的 UIP 位为“1”。在更新周期内,芯片内部时标寄存器数据处于更新阶段,故在该周期内,微处理器不能读芯片时标寄存器的内容,否则将得到不确定数据。更新周期的基本功能主要是刷新各个时标寄存器中的内容,同时秒时标寄存器内容加 1,并检查其他时标寄存器内容是否有溢出。如果有溢出则相应进位日、月、年。另外一个功能是检查三个时、分、秒报警时标寄存器的内容是否与对应时标寄存器的内容相符,如果相符则寄存器 C 中的 AF 位置“1”。如果报警时标寄存器的内容为 C0H 到 FFH 之间的数据,则为不关心状态。

为了采样时标寄存器中的数据,DS12887 提供了两种避开更新周期内访问时标寄存器的方法:一种方法是利用更新周期结束发出的中断。它可以编程允许在每次更新周期结束后发中断申请,提醒 CPU 将有 998ms 左右的时间去获取有效的数据,在中断之后的 998ms 时间内,程序可先将时标数据读入芯片内部不掉电静态 RAM 中。因为芯片内部的静态 RAM 和状态寄存器是可随时读写的,在离开中断服务子程序前应清除寄存器 C 中的 IRQF 位。另一种方法是利用寄存器 A 中的 UIP 位来指示芯片是否处于更新周期。在 UIP 位从低变高 244μs 后,芯片将开始更新周期,所以检测到 UIP 位为低电平时,利用 224μs 的间隔时间去读取时标信息。如检测到 UIP 位为“1”,则可暂缓读数据,等到 UIP 变成低电平后再去读数据。

DS12887 采取连续工作方式,一般无须每次都进行初始化,即使是系统复位时也如此。如果必须要进行初始化,首先应禁止芯片内部的更新周期操作,即先将状态寄存器 B 中的 SET 位置“1”,然后初始化 00H~09H 时标参数寄存器和状态寄存器 A,此后再通过读状态寄存器 C,清除寄存器 C 中的周期中断标志 PF,报警中断标志 AF,更新周期结束中断标志 UF。通过读寄存器 D 中的 VRT 位,读状态寄存器 D 后 VRT 位将自动置“1”,最

后将状态寄存器 B 中的 SET 位置“0”，芯片开始计时工作。

DS12887 共有 3 个闹钟单元，分别为时、分、秒闹钟单元。在其中写入闹钟时间值并且在时钟中断允许的情况下每天到点就会产生中断申请信号。但这种方式每天只提供一次中断信号。另一种方式是在闹钟单元写入“不关心码(C0H~FFH)”，在时闹钟单元写入“不关心码”，可每小时产生一次中断；在时、分闹钟单元写入“不关心码”，可每分钟产生一次中断；而时、分、秒闹钟单元全部写入 FFH，则每秒钟产生一次中断。这种方式也只能在整点、整分或整秒产生一次中断。如果控制系统要求的定时间隔不是整数时，则应该通过软件来调整实现。

图 8.77 是 DS12887 与 8031 单片机的一种接口电路，片选信号由译码电路产生，设片内寄存器及 RAM 地址为 FF00H~FFFFH，下面给出对应该接口电路的 Cx51 驱动程序。

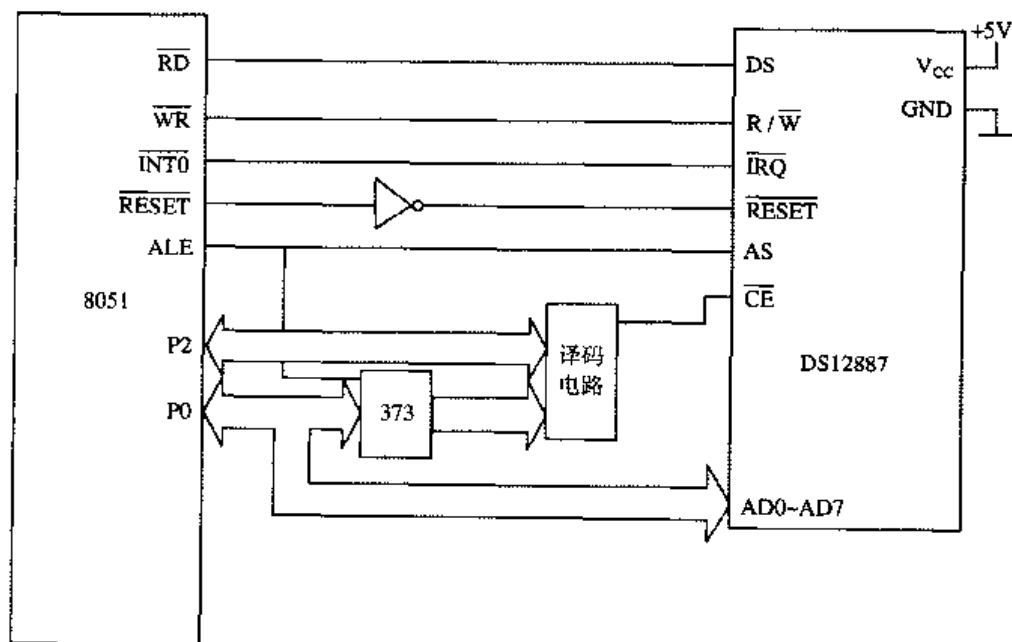


图 8.77 DS12887 与 8031 单片机的接口电路

```
#include <absacc.h>
#include <reg51.h>
#define uchar unsigned char
#define MCA XBYTE[0xff0a]          /* 寄存器 A */
#define MCB XBYTE[0xff0b]          /* 寄存器 B */
#define MCC XBYTE[0xff0c]          /* 寄存器 C */
#define MCD XBYTE[0xff0d]          /* 寄存器 D */
#define HOUR XBYTE[0xff05]         /* 时报警 */

sbit uip_bit=ACC^7;
uchar idata dt[6]={99,4,5,11,11,25}; /* 99/4/5 11: 11:25 */
uchar at[6]={9,8,7,4,2,0};          /* 年、月、日、时、分、秒 */
uchar xdata *addr=0xff00;           /* 片选地址 */
```

```

/***** 初始化函数 *****/
void initstat(void) {
    MCA=0x2f;          /* 时钟频率为 32.768kHz, 中断周期 500ms */
    ACC=MCC;           /* 读寄存器 C, 清零中断请求标志 */
    ACC=MCD;           /* 读寄存器 D, 使 VRT=1, 数据有效 */
    MCB=0x72;          /* 允许中断, 时标为 BCD 码, 24 小时方式 */
    HOUR=0xff;         /* 每小时闹钟中断 */
    EA=1;              /* 开中断 */
    EX0=1;
}

/***** 写时标函数 *****/
void wMC(void) {
    uchar i;
    MCB=0x80;          /* 使 SET=1 */
    for(i=0;i<=9;i++)  /* 置时标初值 */
        *(addr+at[i])=dt[i];
}

/***** 读时标函数 *****/
void rMC(void) {
    uchar i;
    do {ACC=MCA;} while(ui_p_bit); /* 判 UIP 位 */
    for(i=0;i<6;i++)    /* 读时标寄存器的值 */
        dt[i]=*(addr+at[i]);
}

/***** 主函数 *****/
void main(void) {
    wMC();
    initstat();
    rMC();
    while(1);
}

```

8.9.5 用可编程芯片 8279 扩展键盘/LED 显示器接口

Intel 公司生产的 8279 是一种较为常用可编程键盘 / 显示器接口芯片, 它能够自动完成键盘的扫描输入和 LED 显示输出, 可以连接 8 位或 16 位 LED 显示器。8279 的工作分为二个部分: 键盘部分和显示部分。键盘部分能够提供 64 按键阵列 (可扩展为 128) 的扫描接口, 也可以接传感器阵列。键的按下可以是双键锁定或 N 键互锁。键盘输入经过内部反弹跳电路自动消除前后沿按键抖动影响之后, 被选通送入一个 8 字符的 FIFO (先进先出) 存储器。如果送入的字符多于 8 个, 则溢出状态置位。按键输入后将中断输出线升

到高电平向 CPU 发中断申请。显示部分对 LED 提供显示接口。8279 有一个内部的 16×8 显示 RAM, 组成一对 16×4 存储器。显示 RAM 可由 CPU 写入或读出。显示方式有从右进入的计算器方式和从左进入的电传打字方式。显示 RAM 每次读写之后, 其地址自动加 1。

8279 芯片的引脚排列如图 8.78 所示, 各引脚的功能如下:

- DB₀~DB₇: 双向三态数据总线。
- RD、WR: 读、写信号输入。
- CS: 片选信号输入。
- RESET: 复位信号输入。
- CLK: 同步时钟信号输入。
- C/D: 命令 / 数据选择输入, C/D=1 为命令状态, C/D=0 为数据状态。
- IRQ: 中断请求输出, 高电平有效。
- SL₀~SL₃: 扫描信号输出。
- RL₀~RL₇: 回馈信号输入。
- OUTA₀~OUTA₃: A 组显示数据输出线。
- OUTB₀~OUTB₃: B 组显示数据输出线。
- BD: 显示消隐信号输出, 在更换数据时, 其输出信号可使显示器熄灭。

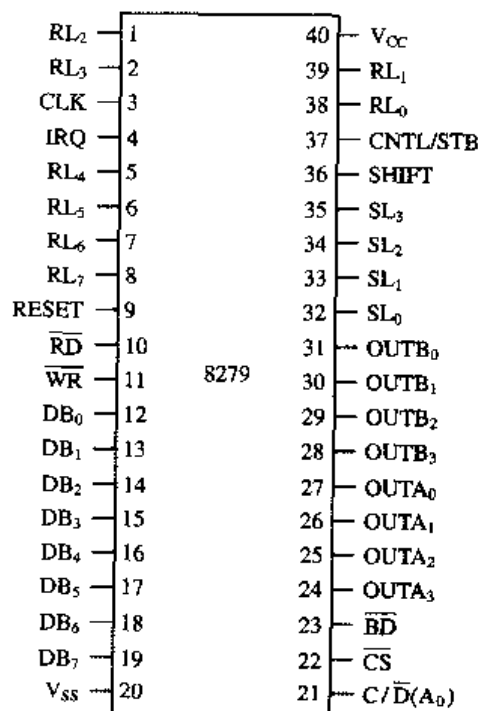


图 8.78 8279 芯片的引脚排列

8279 芯片内部主要由 6 部分组成。

① I/O 控制及数据缓冲器。

② 定时与控制寄存器。

③ 扫描计数器。有编码和译码 2 种工作方式, 在编码方式下, 计数器作 2 进制计数, 4 位扫描线 SL₀~SL₃ 输出 4 位计数状态, 这种计数方式只有经过外部译码之后才能作为键

盘和显示器的扫描码。在译码工作方式下，扫描计数器的最低 2 位经内部译码后，由 $SL_0 \sim SL_3$ 扫描线输出，可以直接用做键盘和显示器的扫描码。

④ 返回缓冲器。从 $RL_0 \sim RL_7$ 输入的 8 个返回信号作为键盘的输入检测线，被锁存到返回缓冲器中。当有键按下时，该键的键值按以下格式送入 8279 的 FIFO 中：

D7	D6	D5	D4	D3	D2	D1	D0
CTRL	SHIFT	行扫描编码			列回馈编码		

⑤ FIFO / 传感器 RAM 及其状态寄存器。这是一个双功能的 8×8 RAM，在键盘或选通方式下用做 FIFO，在传感器矩阵方式下用做传感器 RAM。

⑥ 显示 RAM 和显示地址寄存器。显示 RAM 用来存储显示数据，容量为 8×8 位。显示过程中存储的显示数据轮流从现实寄存器输出。 $OUTA_0 \sim OUTA_3$ 和 $OUTB_0 \sim OUTB_3$ 用做外接显示器件的段码端口，而显示器件的位选则由扫描线 $SL_0 \sim SL_3$ 提供。

8279 的工作方式是由各种控制命令决定的，8279 共有 8 条命令，分述如下。

① 键盘、显示器工作模式设置命令，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	D1	D0	K2	K1	K0

其中最高 3 位 D7、D6、D5 为 000，是本命令的特征码。

D1、D0 位用于决定显示方式，定义如下：

- 00 8 字符显示，左端输入；
- 01 16 字符显示，左端输入；
- 10 8 字符显示，右端输入；
- 11 16 字符显示，右端输入；

K2、K1、K0 位用于设置键盘的工作方式，定义如下：

- 000 编码扫描键盘，两键互锁；
- 001 译码扫描键盘，两键互锁；
- 010 编码扫描键盘，多键有效；
- 011 译码扫描键盘，多键有效；
- 100 编码扫描传感器列阵检测；
- 101 译码扫描传感器列阵检测；
- 110 选通输入，编码扫描显示器；
- 111 选通输入，译码扫描显示器；

② 扫描频率设置命令，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	P4	P3	P2	P1	P0

其中最高 3 位 D7、D6、D5 为 001，是本命令的特征码。

P4~P0 位用于设定从 8279 的 CLK 端输入时钟的分频系数 N， $N=2 \sim 31$ 。

③ 读 FIFO 堆栈的命令，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	AI	×	A2	A1	A0

其中最高3位 D7、D6、D5 为 010，是本命令的特征码。

D4 位为地址自动加 1 标志，当 AI=1 时，每执行一次输入指令，地址 A2、A1、A0 自动加 1。接收到本命令后，CPU 执行输入指令，从 FIFO 中读取数据。地址由 A2、A1、A0 决定，例如 A2、A1、A0=000H，则输入指令执行的结果是将 FIFO 堆栈顶的数据读入 CPU 的累加器。利用 AI 标志位可省去每次读取数据前都要设置读取地址的操作。

④ 读显示 RAM 命令，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	AI	A3	A2	A1	A0

其中最高3位 D7、D6、D5 为 011，是本命令的特征码。

A3、A2、A1、A0 用于区别 16 个 RAM 地址，AI 是地址自动加“1”标志。

⑤ 写显示 RAM 命令，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	AI	A3	A2	A1	A0

其中最高3位 D7、D6、D5 为 100，是本命令的特征码。

命令中的地址码 A3、A2、A1、A0 决定 8279 芯片接收来自 CPU 的数据存放在显示 RAM 的哪个单元。AI 是地址自动增“1”标志。

⑥ 显示屏蔽消隐命令，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
1	0	1	×	IWA	IWB	BLA	BLB

其中最高3位 D7、D6、D5 为 101，是本命令的特征码。

IWA 和 IWB 分别用以屏蔽 A 组和 B 组显示 RAM。在双 4 位显示器使用时，即 OUTA0~OUTA3 和 OUTB0~OUTB3 独立地作为两个半字节输出时，可改写显示 RAM 中的低半字节而不影响高半字节的状态（若 IWA=1），或者可改写高半字节而不影响低半字节（若 IWB=1）。BLA 和 BLB 是消隐特征位，要消隐两组显示输出，必须使 BLA 和 BLB 同时为“1”，要恢复显示时则使它们同时为“0”。

⑦ 清除命令，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
1	1	0	Cd	Cd	Cd	Cf	Ca

其中最高3位 D7、D6、D5 为 110，是本命令的特征码。

Cd、Cd、Cd 用来设定清除显示 RAM 的方式，定义如下：

10× 显示 RAM 所有单元均置“0”；

110 显示 RAM 所有单元均置“20H”；

111 显示 RAM 所有单元均置“1”;

0×× 不清除。

Cf 位用于置空 FIFO, Cf=1, FIFO 被置成空状态 (无数据), 并复位中断输出 IRQ。

Ca 是总清的特征位, Ca=1, 清除 FIFO 状态和显示 RAM。

⑧ 中断结束 / 设置出错方式命令, 格式如下:

D7	D6	D5	D4	D3	D2	D1	D0
1	1	1	E	×	×	×	×

其中最高 3 位 D7、D6、D5 为 111, 是本命令的特征码。

E=1 使本命令有效。

在传感器工作方式中, 该命令使 IRQ 输出线变为低电平 (即中断结束), 允许再次对 RAM 写入 (在检测到传感器变化后, IRQ 可能已经变成高电平, 这时禁止在复位前再次将信息写入 RAM)。在 N 键巡回工作方式中, 若 E=1, 在消颤期内如果有多键同时按下, 则产生中断, 并且阻止对 RAM 的写入。

除了上述 8 条命令之外, 8279 还有一个状态字。状态字用来指出 FIFO 中的字符个数、出错信息以及能否对显示 RAM 进入写入操作。状态字格式如下:

D7	D6	D5	D4	D3	D2	D1	D0
DU	S/E	O	U	F	N2	N1	N0

在清除命令执行期间 DU 为“1”, 此时对显示 RAM 写操作无效。

S / E 用于传感器扫描方式, 几个传感器同时闭合时置“1”。

当 FIFO 已满, 又输入一个字符时发生溢出, 置“1” O

在 FIFO 中没有输入字符时, CPU 读 FIFO, 则置“1” U。

F=1 时, 表示 FIFO 已满 (存有 8 个键入数据)。

N2、N1、N0 表示 FIFO 中数据的个数。

图 8.79 所示为 8279 与 8031 单片机的一种接口电路, 8279 的命令状态口和数据口地址分别为外部 RAM 的 0x8001 和 0x8000, 通过 Cx51 编译器的预定义宏“ABSACC.H”实现对端口地址寻址。显示缓冲区也位于外部 RAM 中。8279 初始化为 8×8 字符显示, 右端输入, 编码扫描键盘, N 键轮回方式, 程序对于键的处理采用查表散转方法, 定义一个位于 ROM 中的函数指针数组“code*KeyProcTab[]”作为键值处理散转表, 通过函数指针调用 ReadKey()函数获得键值并转到相应的键值处理服务程序。作为范例程序仅对数字键进行了简单处理, 将键盘输入的数字从右往左循环显示, 其他功能键可以根据需要很容易编写相应的功能处理函数。下面给出对应该电路的 Cx51 驱动程序。

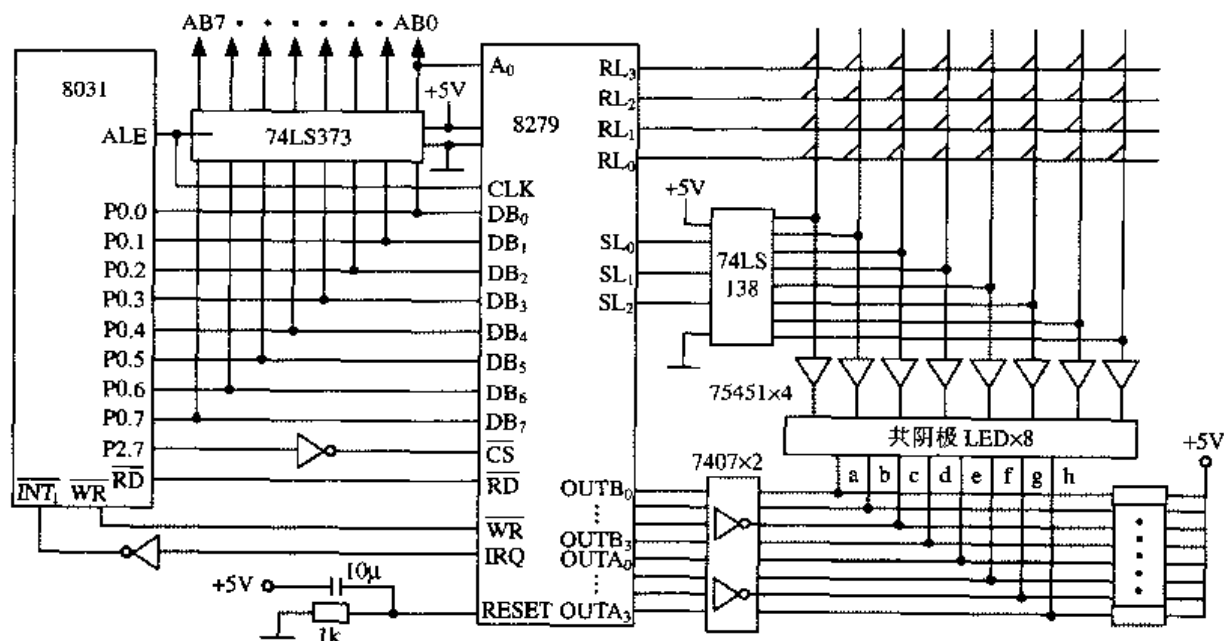


图 8.79 8279 与 8031 单片机的一种接口电路

```
#include <absacc.h>
char xdata DisBuf[8];          /* 显示缓冲区 */
unsigned char code keyval[]={0xc1,0xc8,0xc9,0xd0}; /* 键值表 */

/***** 8279 初始化函数 *****/
void KbDisInit() {
    XBYTE[0xc801]=0x00;        /* 设置 8279 工作方式 */
    XBYTE[0xc801]=0x22;        /* 设置 8279 分频系数 */
    XBYTE[0xc801]=0xdF;        /* 清零 8279 */
    while (XBYTE[0xc801] & 0x80);
    XBYTE[0xc801]=0x80;
    XBYTE[0xc800]=0xc8;
}

/***** 读键值函数 *****/
unsigned char ReadKey() {
    unsigned char i,j;
    if (XBYTE[0xc801] & 0x07) { /* 8279 Key put */
        XBYTE[0xc801]=0x40;    /* read 8279 FIFO */
        i=XBYTE[0xc800];        /* i=keyval */
        j=0;
        while (i!=keyval[j]){j++;} /* look up keyval tabel */
        return(j+1);
    }
    // return (XBYTE[0xc800]+1);
}
```

```

    return (0);          /* no Key put */
}

/***** 循环显示函数 *****/
void KeyShift() {
    unsigned char i;
    for (i=0; i<8; i++)
        DisBuf[i]=DisBuf[i+1];
}

/***** 无按键处理函数 *****/
void NoKey() {
    ;
}

/***** 0 键处理函数 *****/
void k0() {
    KeyShift();
    DisBuf[7]=0;
}

/***** 1 键处理函数 *****/
void k1() {
    KeyShift();
    DisBuf[6]=1;
}

/* k2, ... 其他按键处理函数可在此处插入 */

code void (code * KeyProcTab[])()={NoKey, k0,k1 /* k2,... */};

/***** 主函数 *****/
void main() {
    KbDisInit();          /* 初始化 */
    while(1)
        (* KeyProcTab[ReadKey()])(); /* 根据不同按键的值查表散转 */
}

```

8.9.6 LCD 显示模块 EA-D20040AR 的 Cx51 应用编程

液晶显示器 LCD 体积小, 重量轻, 功耗极低, 应用十分广泛。在单片机系统中采用 LCD 时可以通过专门的 LCD 驱动器芯片进行接口, 也可以直接采用 LCD 显示模块, 一般来说采用显示模块比采用显示驱动器芯片方便。日本 EPSON 公司生产的 EA-D 系列点阵

字符型 LCD 模块是应用较多的一种显示模块, 它由 TN 型液晶显示器、CMOS 驱动器和 CMOS 控制器组成, 模块内集成有字符发生器和数据存储器, 采用单 +5V 电源供电, 内部固化有 96 个 ASCII 字符和 92 个特殊字符的字库。是一种 5×7 点阵字符型 LCD 模块, 内部结构如图 8.80 所示, 它由点阵式液晶显示面板、SED1278 专用集成电路和 4 个列驱动器组成, SED1278 完成显示模块的时序控制, 同时也可以驱动 16 行 40 列的点阵库, 另外内部还集成了一个 80×8 位的显示数据存储器 DDRAM、一个 7200 位的只读字符发生器 CGROM 和一个 512 位的随机存取字符发生器 CGRAM。

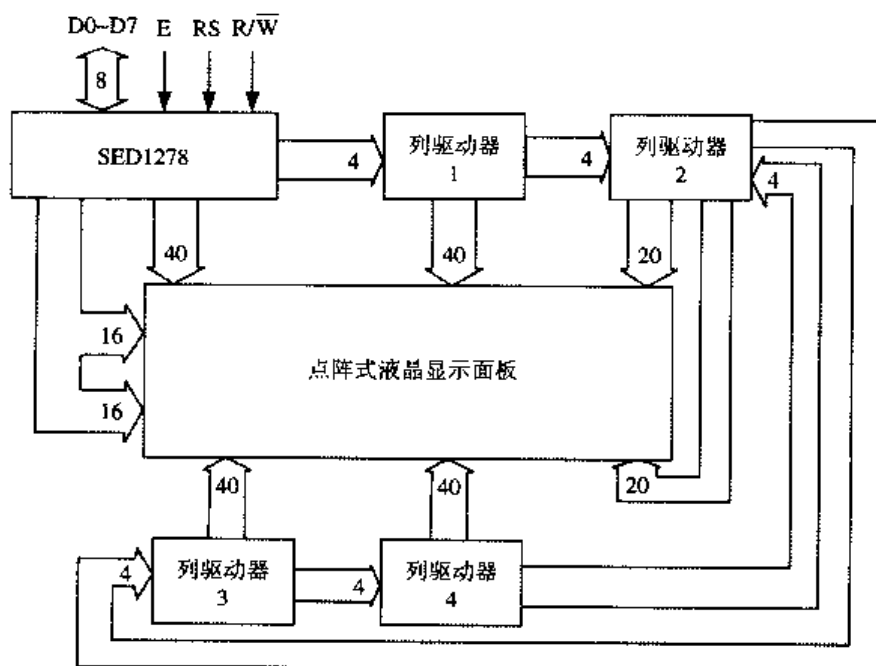


图 8.80 EA-D20040AR 点阵字符型 LCD 模块的内部结构

EA-D20040AR 有 14 条引脚:

VSS: 地线输入端。

VDD: +5V 电源输入端。

VO: 液晶显示面板亮度调节, 通过 10~20k Ω 的电阻接到 +5V 和地之间起调节亮度的作用。

RS: 寄存器选择信号输入线, 低电平选通指令寄存器, 高电平选通数据寄存器。

R/W: 读 / 写信号输入线, 低电平为写入, 高电平为读出。

E: 片选信号输入线, 高电平有效。

D0~D7: 数据总线。

显示数据存储器 (DDRAM) 与显示屏幕的物理位置是一一对应的, 当给显示数据存储器的某一单元写入一个字符的编码时, 该字符就在对应的位置上显示出来。表 8-24 所示为 EA-D20040AR 的 DDRAM 与显示屏物理位置对应关系。

表 8-24 EA-D20040AR 的 DDRAM 与显示屏物理位置对应关系

显示列位置		1	2	3	4	5	6	7	8	9	10	11	12	...	20
DDRAM 地址 (16 进制)	第 1 行	00	01	02	03	04	05	06	07	08	09	0A	0B	...	13
	第 2 行	40	41	42	43	44	45	46	47	48	49	4A	4B	...	53
	第 3 行	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	...	27
	第 4 行	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	...	67

字符编码就是要写入 DDRAM 的数据, 字符编码位于字符存储器中。字符存储器可分为随机存取存储器(CGRAM)和只读存储器(CGROM)。表 8-25 所示为 CGRAM 和 CGROM 地址与显示字符的对应关系。表中第一列为 CGRAM, 其余为 CGROM, CGROM 内已经固化了 192 个显示字符, 由用户随意使用, 如果想显示其中任一个字符, 只要把该字符对应的编码送入 DDRAM 即可。如果想显示 192 个字符以外的字符, 则可以利用 CGRAM 来进行定义。

表 8-25 CGROM 和 CGRAM 中字符编码与显示字符的关系

字符编码低位	字符编码高位												
	0000	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
xxxx0000	CGRAM (1)		0	@	P	\	p		-	々	ミ	α	p
xxxx0001	(2)	!	1	A	Q	a	q	•	ヌ	チ	ム	a	q
xxxx0010	(3)	“	2	B	R	b	r	└	イ	ツ	ヌ	β	0
xxxx0011	(4)	#	3	C	S	c	s	┘	ウ	チ	モ	e	∞
xxxx0100	(5)	\$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)	%	5	E	U	e	u	。	オ	ナ	ユ	σ	O
xxxx0110	(7)	&	6	F	V	f	v	ラ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)	,	7	G	W	g	w	ア	キ	ヌ	ラ	g	π
xxxx1000	(1)	(8	H	X	h	x	イ	ク	ネ	リ	f	X
xxxx1001	(2))	9	I	Y	i	y	ウ	ケ	ノ	ル	-1	Y
xxxx1010	(3)	*	:	J	Z	j	z	エ	コ	ハ	レ	j	千
xxxx1011	(4)	+	;	K	[k	[オ	サ	ヒ	ロ	x	万
xxxx1100	(5)	,	<	L	Y	l	l	セ	シ	フ	ワ	Φ	◎
xxxx1101	(6)	-	=	M]	m]	コ	ス	ヘ	ン	ε	÷
xxxx1110	(7)	.	>	N	^	n	→	ヨ	セ	ホ	ハ	n	
xxxx1111	(8)	/	?	O	-	o	←	ツ	ソ	マ	ロ	○	

EA-D20040AR 显示模块的显示功能是由命令来实现的, 共有 11 条命令, 其编码格式如表 8-26 所示。

表 8-26 EA-D20040AR 显示模块的命令编码格式

命 令	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
清显示	0	0	0	0	0	0	0	0	0	1
光标返回	0	0	0	0	0	0	0	0	1	×
设置输入方式	0	0	0	0	0	0	0	1	I/D	S
显示开/关控制	0	0	0	0	0	0	1	D	C	B
光标或字符移位	0	0	0	0	0	1	S/C	R/L	×	×
功能设置	0	0	0	0	1	DL	N	F	×	×
设置字符发生器地址	0	0	0	1	A5	A4	A3	A2	A1	A0
设置 DDRAM 地址	0	0	1	A6	A5	A4	A3	A2	A1	A0
读忙标志和地址	0	1	BF	A6	A5	A4	A3	A2	A1	A0
写数据到 CGRAM 或 CGROM	1	0	D	D	D	D	D	D	D	D
从 CGRAM 或 CGROM 读数	1	1	D	D	D	D	D	D	D	D

各条命令的功能解释如下。

① 清显示：该命令把空格编码 20H 写入显示数据存贮器的所有单元。

② 光标返回：该命令把地址计数器中 DDRAM 地址清 0，如果显示屏上显示了字符，则光标移到起始位置。如果显示两行，则光标移到第一行第一个字符的位置，显示数据存贮器的内容不变。

③ 设置输入方式：当一个字符编码被写入 DDRAM 或从 DDRAM 中读出时，若 I/D=1，则 DDRAM 地址加 1，若 I/D=0，则 DDRAM 地址减 1。地址加 1 时，光标右移；地址减 1 时，光标左移。对 CGRAM 的读写操作和 DDRAM 一样，只是 CGRAM 与光标无关。当 S=1 时，整个显示屏向左 (I/D=1) 或向右 (I/D=0) 移动。在从 DDRAM 中读数、向 CGRAM 写数或从 CGRAM 中读数、S=0 这三种情况下，显示屏不移动。

④ 显示开 / 关控制：当 D=0 时，显示器关闭，显示数据存贮器中的数据不变；当 D=1 时，显示器立即显示 DDRAM 中的数据。当 C=0 时，不显示光标；当 C=1 时，显示光标。当 B=1 时，显示闪烁光标。

⑤ 光标或字符移位：该命令使显示和光标向左或向右移位。对两行显示而言，光标从第一行的第 40 个字符位置移到第二行的首位。从第二行的第 40 个位置不能移位到清屏的起始位置，而是回到第二行的第一个位置。命令中 S/C 和 R/L 位的作用如下：

S/C	R/L	作用
0	0	光标左移，地址计数器减一
0	1	光标右移，地址计数器加一
1	0	显示屏左移，光标跟随显示屏移动
1	1	显示屏右移，光标跟随显示屏移动

⑥ 功能设置：当 DL=1 时，内部总线为 4 位宽度 (DB7~DB4)，当 DL=0 时，内部总线为 8 位宽度 (DB7~DB0)。当 N=0 时，单行显示，当 N=1 时，双行显示。当 F=0 时，显示字符为 5×7 点阵，当 F=1 时，显示字符为 5×10 点阵。

⑦ 设置字符发生器地址：该命令的功能是设置 CGRAM 的地址。

⑧ 设置 DDRAM 地址：该命令的功能是设置 DDRAM 的地址。

⑨ 读忙标志和地址：该命令的功能是读出忙标志 BF 的值。当读出的 BF=1 时，则说明系统内部正在进行操作，不能接收下一条命令。在读出 BF 值的同时，CGRAM 和 DDRAM 所使用的地址计数器的值也被同时读出。

⑩ 写数据到 CGRAM 或 DDRAM：该命令的功能是把二进制数 DDDDDDDD 写入 CGRAM 或 DDRAM 中，若先送入 CGRAM 的地址则向 CGRAM 写入；若先送入 DDRAM 地址则向 DDRAM 写入。

⑪ 从 CGRAM 或 DDRAM 读数：该命令的功能是将数据从用写数据命令建立的 CGRAM 或 DDRAM 地址指出的 RAM 中读出。在本命令之前的命令应是设置字符发生器地址命令、光标移位命令、或是上次 CGRAM/DDRAM 数据读出命令，若是其他命令，读出的数据可能会出错。

在执行读数据或写数据命令之后，地址计数器会自动加 1 或减 1。一般是先执行一条地址建立命令或光标移位命令，再执行读数据命令，一旦一条读数据命令被执行后，就可连续执行数据读取命令，而不需再执行其他命令了。

图 8.81 所示为液晶显示模块 EA-D20040 与单片机 8031 的一种接口电路，下面给出对应接口的 Cx51 驱动程序。

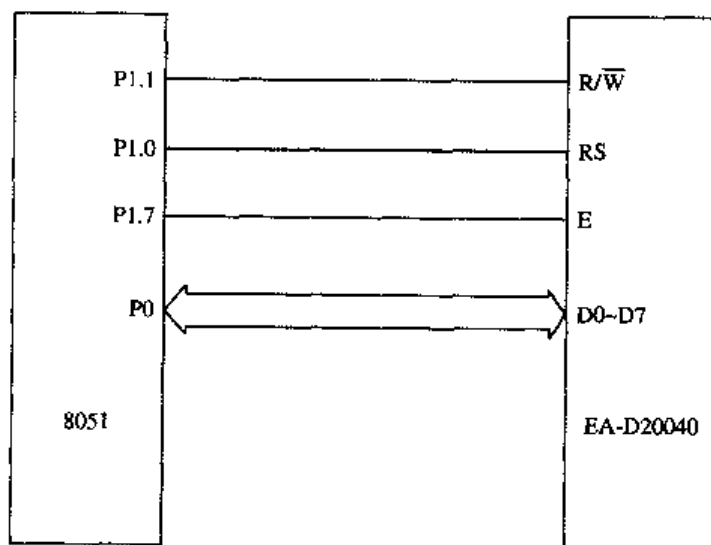


图 8.81 液晶显示模块 EA-D20040 与单片机 8031 的一种接口

```
#include <reg51.h>
#include <intrins.h>
#define Uchar unsigned char
#define Uint unsigned int

sbit RS = P1^0;      /* 定义 LCD 模块控制引脚，可根据需要进行修改 */
sbit RW = P1^1;
sbit E = P1^7;
```

```

#define DataPort P0      /* 定义 LCD 模块数据端口 */
#define Busy0x80

code char exampl[]="This is an example for HD44780 LCM.      \n";

/***** 短延时函数 *****/
void Delay5Ms(void) {
    Uint i = 5552;
    while(i--);
}

/***** 长延时函数 *****/
void Delay400Ms(void) {
    Uchar i = 5;
    Uint j;
    while(i--) {
        j=7269;
        while(j--);
    };
}

/***** 读忙状态函数 *****/
* 功能: 在正常读写操作之前检测 LCD 模块的忙状态,
*      DB7 = 0: LCD 控制器空闲; DB7 = 1: LCD 控制器忙。
*****/
void WaitForEnable( void ) {
    DataPort = 0xff;
    RS =0; RW = 1; _nop_(); E = 1; _nop_(); _nop_();
    while( DataPort & Busy );
    E = 0;
}

/***** 命令写入函数 *****/
* 功能: 向 LCD 模块写入命令字符 CMD,
*      若 AttribC = 1 检测忙信号, 若 AttribC = 0 不检测忙信号。
*****/
void LcdWriteCommand(Uchar CMD,Uchar AttribC) {
    if (AttribC) WaitForEnable(); /* 检测忙信号? */
    RS = 0; RW = 0; _nop_();
    DataPort = CMD; _nop_(); /* 将命令字符送往数据口 */
    E = 1;_nop_();_nop_();E = 0;
}

```

```

/***** 光标定位函数 *****/
* 功能: 将显示光标定位到 posx, posy 坐标处。
*****/
void LocateXY( char posx,char posy) {
    Uchar temp;
    temp = posx & 0xf;
    posy &= 0x1;
    if ( posy )temp |= 0x40;
    temp |= 0x80;
    LcdWriteCommand(temp,0);
}

/***** 显示字符写入函数 *****/
* 功能: 在当前光标位置显示一个字符。
*****/
void LcdWriteData( char dataW ) {
    WaitForEnable();          /* 检测忙信号 */
    RS = 1; RW = 0; _nop_();
    DataPort = dataW; _nop_(); /* 将显示字符送往数据口 */
    E = 1; _nop_(); _nop_(); E = 0;
}

/***** 单个字符显示函数 *****/
* 功能: 按指定的 x, y 位置显示一个字符 wdata。
*****/
void DispOneChar(Uchar x,Uchar y,Uchar Wdata) {
    LocateXY( x, y );          /* 定位显示地址 */
    LcdWriteData( Wdata );      /* 写字符 */
}

/***** 显示字符串函数 *****/
* 功能: 从坐标 x, y 处开始显示 ptr 指定的字符串。
*****/
void ePutstr(Uchar x,Uchar y, Uchar code *ptr) {
    Uchar i,l=0;
    while (ptr[l] >31){l++;};
    for (i=0;i<l;i++) {
        DispOneChar(x++,y,ptr[i]);
        if ( x == 16 ){
            x = 0; y ^= 1;
        }
    }
}

```

```

/***** 显示字符串函数 *****/
* 功能: 显示一行连续字符串。
*****/
void Display( Uchar dd ) {
    Uchar i;
    for (i=0;i<16;i++) {
        DispOneChar(i,1,dd++);
        dd &= 0x7f;
        if (dd<32) dd=32;
    }
}

/***** 初始化函数 *****/
* 功能: 向 LCD 模块写入不同命令, 完成必要的初始化过程。
*****/
void LcdReset( void ) {
    LcdWriteCommand( 0x38, 0); /* 设置显示模式(不检测忙信号) */
    Delay5Ms(); /* 短延时 */
    LcdWriteCommand( 0x38, 0); /* 共进行三次 */
    Delay5Ms();
    LcdWriteCommand( 0x38, 0);
    Delay5Ms();
    LcdWriteCommand( 0x38, 1); /* 设置显示模式(以后均检测忙信号) */
    LcdWriteCommand( 0x08, 1); /* 显示关闭 */
    LcdWriteCommand( 0x01, 1); /* 显示清屏 */
    LcdWriteCommand( 0x06, 1); /* 设置显示光标移动 */
    LcdWriteCommand( 0x0c, 1); /* 设置显示开及光标位置 */
}

/***** 主函数 *****/
void main(void) {
    Uchar temp;
    Delay400Ms(); /* 长延时 */
    LcdReset(); /* LCD 模块初始化 */
    temp = 32;
    ePutstr(0,0,exempl); /* 显示一个预定字符串 */
    Delay400Ms(); /* 长延时 */
    Delay400Ms();
    Delay400Ms();
    Delay400Ms();
    Delay400Ms();
    Delay400Ms();
}

```

```

while(1){
    temp &= 0x7f;          /* 只显示 ASCII 字符 */
    if (temp<32)temp=32;   /* 屏蔽控制字符, 不予显示 */
    Display( temp++ );
    Delay400Ms();
}
}

```

8.9.7 内置 T6963C 驱动器的 LCD 显示模块 Cx51 应用编程

在中规模点阵图形 LCD 模块中, 内置 T6963C 驱动器的 LCD 是目前较为常用且品种较多的一类点阵图形液晶显示模块, T6963C 是日本东芝公司的产品, 它最大的特点是具有硬件初始值设置功能, 其初始化工作在加电时就已经基本完成, 软件操作的主要精力可以全部用于显示画面的设计上。T6963C 内置有 128 种 5×8 点阵的 ASCII 字符发生器 CGROM, 并允许在显示存储器内开辟一个用户自定义的 8×8 点阵字模库 CGRAM。T6963C 可以管理 64KB 的显示存储器, 它可以把显示存储器分成文本显示区、图形显示区以及自定义字符库区等。香港精电公司的推出的 MGLS240128T 点阵图形 LCD 模块所采用的就是内置 T6963C 驱动器, 它有 21 条引脚:

FG: 显示屏框夹外壳地接地。

Vss: 电源地。

VDD: +5V 电源。

Vo: 对比度调节负电压输入。

VEE: DC-DC 负电源输出 (液晶屏工作电压, 作对比度调节)。

WR: 数据写入, 低电平有效。

RD: 数据读出, 低电平有效。

CE: 片选端, 低电平有效。

C/D: 通道选择端, C/D=1 为命令通道, C/D=0 为数据通道。

Reset: 复位信号, Reset=1 为正常工作 (T6963C 有内部上拉电阻), Reset=0 为初始化 T6963C, 文本和图形的地址、文本和图形区域设定被保持。

D0~D7: 数据总线。

FS: 字体选择: FS = 1, 选择 6×8 点阵字体, FS = 0, 选择 8×8 点阵的字体。

LED+: 背光电源正端。

LED-: 背光电源负端。

T6963C 提供两种命令形式: 带参数命令和无参数命令。带参数命令中的参数需要在命令编码之前输入, 格式如下:

参数 1 参数 2 命令编码

无参数命令只要给出命令编码即可。表 8-27 列出了 T6963C 的全部命令编码。

表 8-27 T6963C 的命令编码

命 令	参数 1	参数 2	编 码	功 能
寄存器设置	水平位置 (低 7 位有效)	垂直位置 (低 5 位有效)	0010 0001	设置光标位置
	偏置地址 (低 5 位有效)	00H	0010 0010	设置 CGRAM 偏置地址
	地址低 8 位	地址高 8 位	0010 0100	设置显示地址
显示区域设置	地址低 8 位	地址高 8 位	0100 0000	设置文本起始地
	列	00H	0100 0001	设置文本区宽度
	地址低 8 位	地址高 8 位	0100 0010	设置图形起始地址
	列	00H	0100 0011	设置图形区宽度
模式设定	—	—	1000 x000	文本与图形以“或”关系合成显示
	—	—	1000 x001	文本与图形以“异或”关系合成显示
	—	—	1000 x010	文本与图形以“与”关系合成显示
	—	—	1000 x011	文本显示特征以双字节表示
	—	—	1000 0xxx	内部 CGROM 模式
	—	—	1000 1xxx	外部 CGRAM 模式
显示模式	—	—	1001 0000	显示关闭
	—	—	1001 xx10	打开光标, 黑色关闭
	—	—	1001 xx11	打开光标, 黑色显示
	—	—	1001 01xx	打开文本方式, 关闭图形方式
	—	—	1001 10xx	关闭文本方式, 打开图形方式
	—	—	1001 11xx	图形文本混合方式
光标形式	—	—	1010 0000	1 条线
	—	—	1010 0001	2 条线
	—	—	1010 0010	3 条线
	—	—	1010 0011	4 条线
	—	—	1010 0100	5 条线
	—	—	1010 0101	6 条线
	—	—	1010 0110	7 条线
	—	—	1010 0111	8 条线
数据自动读写	—	—	1011 0000	数据自动写入设定
	—	—	1011 0001	数据自动读出设定
	—	—	1011 0000	自动复位
数据读写	—	—	1100 0000	数据写入, 地址自动增量
	—	—	1100 0001	数据读出, 地址自动增量
	—	—	1100 0010	数据写入, 地址自动减量
	—	—	1100 0011	数据读出, 地址自动减量
	—	—	1100 0100	数据写入, 地址保持不变
	—	—	1100 0101	数据读出, 地址保持不变

续表

命 令	参数 1	参数 2	编 码	功 能
屏幕读取	—	—	1110 0000	读取屏幕显示数据
屏幕拷贝	—	—	1110 1000	复制屏幕显示数据
位操作	—	—	1111 0xxx	位清零
	—	—	1111 1xxx	位置 1
	—	—	1111 x000	位 0
	—	—	1111 x001	位 1
	—	—	1111 x010	位 2
	—	—	1111 x011	位 3
	—	—	1111 x100	位 4
	—	—	1111 x101	位 5
	—	—	1111 x110	位 6
	—	—	1111 x111	位 7

注：表中参数栏的“—”表示无参数。

T6963C 提供一个状态字，格式如下：

S7	S6	S5	S4	S3	S2	S1	S0
----	----	----	----	----	----	----	----

其中各位的含义如下：

- S0: 命令读写状态, S0=1 为准备好, S0=0 为忙。
- S1: 数据读写状态, S1=1 为准备好, S1=0 为忙。
- S2: 数据自动读状态, S2=1 为准备好, S2=0 为忙。
- S3: 数据自动写状态, S3=1 为准备好, S3=0 为忙。
- S4: 未用
- S5: 控制器运行检测可能性, S5=1 为可能, S5=0 为不能。
- S6: 屏幕读取 / 屏幕拷贝出错状态, S6=1 为出错, S6=0 为正确。
- S7: 闪烁状态检测, S7=1 为显示, S6=0 为关闭。

这些标识位各有各的应用场合, 并非同时有效。CPU 在写命令一次读写数据时, S0 和 S1 要同时有效, 即“准备好”状态。当 CPU 采用自动读写功能时, S2 或 S3 将取代 S0 和 S1 作为忙标志。S6 是考察 T6963C 屏幕读取和屏幕拷贝命令执行情况标志位。S5 和 S7 表示控制器内部的运行状态, T6963C 应用时不会使用它们。对 T6963C 进行每一次软件操作之前都要判读忙标志, 只有在不忙 (即“准备好”) 状态下 CPU 对 T6963C 的操作才有效。

T6963C 的读写时序如图 8.82 所示。

图 8.83 所示为香港精电公司的 MGLS240128T 点阵图形 LCD 模块与单片机 8031 的一种接口电路, 下面给出针对该电路的 Cx51 驱动程序。

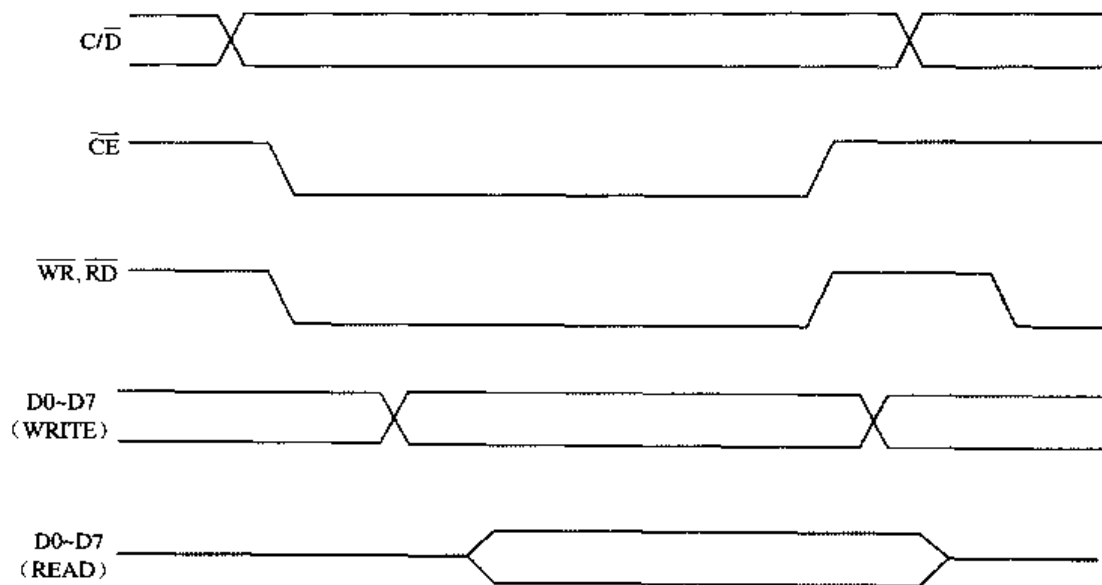


图 8.82 T6963C 的读写时序

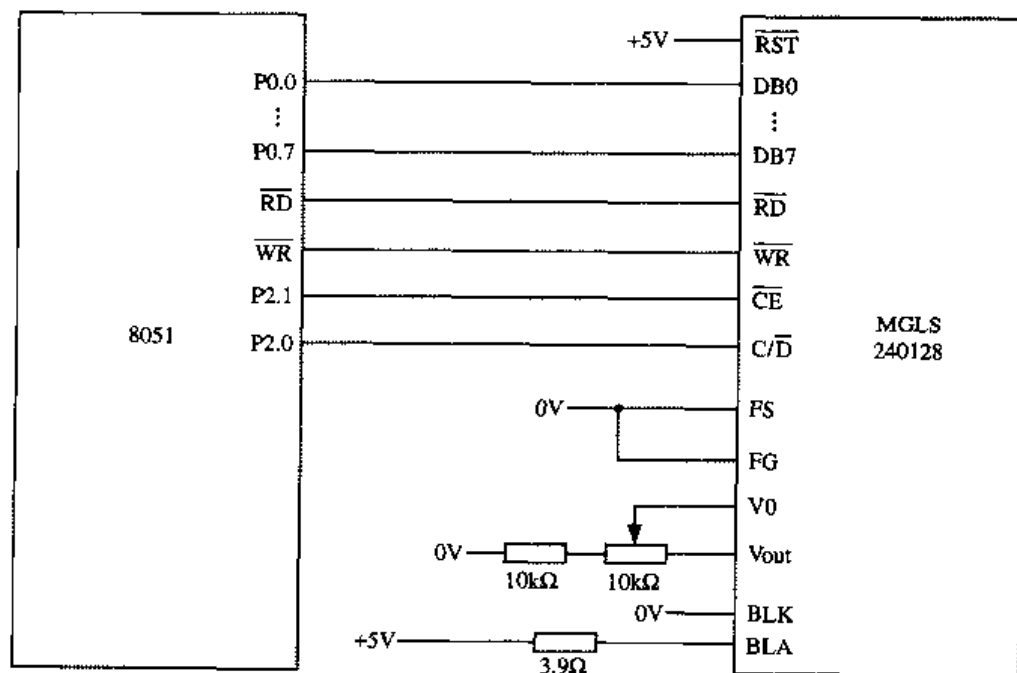


图 8.83 MGLS240128T 点阵图形 LCD 模块与单片机 8031 的接口电路

```
#include <absacc.h>
#include <reg52.h>
#include <stdarg.h>
#include <stdio.h>

#define ulong unsigned long
#define uint unsigned int
#define uchar unsigned char
#define STX 0x02
```



```

#define ETX 0x03
#define EOT 0x04
#define ENQ 0x05
#define BS 0x08
#define CR 0x0D
#define LF 0x0A
#define DLE 0x10
#define ETB 0x17
#define SPACE 0x20
#define COMMA 0x2C
#define TRUE 1
#define FALSE 0
#define HIGH 1
#define LOW 0

/* T6963C 端口定义 */
#define LCMDW XBYTE[0x8000] /* 数据口 */
#define LCMCW XBYTE[0x8100] /* 命令口 */

/* T6963C 命令定义 */
#define LC_CUR_POS 0x21 /* 光标位置设置 */
#define LC_CGR_POS 0x22 /* CGRAM 偏置地址设置 */
#define LC_ADD_POS 0x24 /* 地址指针位置 */
#define LC_TXT_STP 0x40 /* 文本区首址 */
#define LC_TXT_WID 0x41 /* 文本区宽度 */
#define LC_GRH_STP 0x42 /* 图形区首址 */
#define LC_GRH_WID 0x43 /* 图形区宽度 */
#define LC_MOD_OR 0x80 /* 显示方式: 逻辑“或” */
#define LC_MOD_XOR 0x81 /* 显示方式: 逻辑“异或” */
#define LC_MOD_AND 0x82 /* 显示方式: 逻辑“与” */
#define LC_MOD_TCH 0x83 /* 显示方式: 文本特征 */
#define LC_DIS_SW 0x90 /* 显示开关: */
/* D0=1/0: 光标闪烁启用/禁用; */
/* D1=1/0: 光标显示启用/禁用; */
/* D2=1/0: 文本显示启用/禁用; */
/* D3=1/0: 图形显示启用/禁用; */
#define LC_CUR_SHP 0xA0 /* 光标形状选择: 0xA0-0xA7 表示光标占的行数 */
#define LC_AUT_WR 0xB0 /* 自动写设置 */
#define LC_AUT_RD 0xB1 /* 自动读设置 */
#define LC_AUT_OVR 0xB2 /* 自动读/写结束 */
#define LC_INC_WR 0xC0 /* 数据一次写, 地址加 1 */
#define LC_INC_RD 0xC1 /* 数据一次读, 地址加 1 */
#define LC_DEC_WR 0xC2 /* 数据一次写, 地址减 1 */

```

```

#define LC_DEC_RD 0xC3      /* 数据一次读, 地址减 1 */
#define LC_NOC_WR 0xC4      /* 数据一次写, 地址不变 */
#define LC_NOC_RD 0xC5      /* 数据一次读, 地址不变 */
#define LC_SCN_RD 0xE0      /* 屏读 */
#define LC_SCN_CP 0xE8      /* 屏拷贝 */
#define LC_BIT_OP 0xF0      /* 位操作 */

code uchar const uPowArr[] = {0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};

/* ASCII 字模宽度及高度定义 */
#define ASC_CHR_WIDTH 8
#define ASC_CHR_HEIGHT 12

/* ASCII 字模, 显示为 8×16 */
char code ASC_MSK[96*12] = { /* 此字体下对应的点阵为: 宽×高= 8×12 */
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,0xff,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, // ' '
0x00,0x0C,0x1E,0x1E,0x1E,0x0C,0x0C,0x00,0x0C,0x0C,0x00,0x00, // '!'
0x00,0x66,0x66,0x66,0x24,0x00,0x00,0x00,0x00,0x00,0x00,0x00, // '*'
0x00,0x36,0x36,0x7F,0x36,0x36,0x36,0x7F,0x36,0x36,0x00,0x00, // '#'
0x0C,0x0C,0x3E,0x03,0x03,0x1E,0x30,0x30,0x1F,0x0C,0x0C,0x00, // '$'
0x00,0x00,0x00,0x23,0x33,0x18,0x0C,0x06,0x33,0x31,0x00,0x00, // '%'
0x00,0x0E,0x1B,0x1B,0x0E,0x5F,0x7B,0x33,0x3B,0x6E,0x00,0x00, // '&'
0x00,0x0C,0x0C,0x0C,0x06,0x00,0x00,0x00,0x00,0x00,0x00, // '''
0x00,0x30,0x18,0x0C,0x06,0x06,0x06,0x0C,0x18,0x30,0x00,0x00, // '{'
0x00,0x06,0x0C,0x18,0x30,0x30,0x30,0x18,0x0C,0x06,0x00,0x00, // '}'
0x00,0x00,0x00,0x66,0x3C,0xFF,0x3C,0x66,0x00,0x00,0x00,0x00, // '*'
0x00,0x00,0x00,0x18,0x18,0x7E,0x18,0x18,0x00,0x00,0x00,0x00, // '+'
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x06,0x00, // ','
0x00,0x00,0x00,0x00,0x00,0x7F,0x00,0x00,0x00,0x00,0x00,0x00, // '-'
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00, // '.'
0x00,0x00,0x40,0x60,0x30,0x18,0x0C,0x06,0x03,0x01,0x00,0x00, // '/'
0x00,0x3E,0x63,0x73,0x7B,0x6B,0x6F,0x67,0x63,0x3E,0x00,0x00, // '0'
0x00,0x08,0x0C,0x0F,0x0C,0x0C,0x0C,0x0C,0x0C,0x3F,0x00,0x00, // '1'
0x00,0x1E,0x33,0x33,0x30,0x18,0x0C,0x06,0x33,0x3F,0x00,0x00, // '2'
0x00,0x1E,0x33,0x30,0x30,0x1C,0x30,0x30,0x33,0x1E,0x00,0x00, // '3'
0x00,0x30,0x38,0x3C,0x36,0x33,0x7F,0x30,0x30,0x78,0x00,0x00, // '4'
0x00,0x3F,0x03,0x03,0x03,0x1F,0x30,0x30,0x33,0x1E,0x00,0x00, // '5'
0x00,0x1C,0x06,0x03,0x03,0x1F,0x33,0x33,0x33,0x1E,0x00,0x00, // '6'
0x00,0x7F,0x63,0x63,0x60,0x30,0x18,0x0C,0x0C,0x0C,0x00,0x00, // '7'
0x00,0x1E,0x33,0x33,0x37,0x1E,0x3B,0x33,0x33,0x1E,0x00,0x00, // '8'
0x00,0x1E,0x33,0x33,0x33,0x3E,0x18,0x18,0x0C,0x0E,0x00,0x00, // '9'
0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00, // ':'

```

```

0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x1C,0x1C,0x18,0x0C,0x00, // ';'
0x00,0x30,0x18,0x0C,0x06,0x03,0x06,0x0C,0x18,0x30,0x00,0x00, // '<'
0x00,0x00,0x00,0x00,0x7E,0x00,0x7E,0x00,0x00,0x00,0x00, // '='
0x00,0x06,0x0C,0x18,0x30,0x60,0x30,0x18,0x0C,0x06,0x00,0x00, // '>'
0x00,0x1E,0x33,0x30,0x18,0x0C,0x0C,0x00,0x0C,0x0C,0x00,0x00, // '?'
0x00,0x3E,0x63,0x63,0x7B,0x7B,0x7B,0x03,0x03,0x3E,0x00,0x00, // '@'
0x00,0x0C,0x1E,0x33,0x33,0x33,0x3F,0x33,0x33,0x33,0x00,0x00, // 'A'
0x00,0x3F,0x66,0x66,0x66,0x3E,0x66,0x66,0x66,0x3F,0x00,0x00, // 'B'
0x00,0x3C,0x66,0x63,0x03,0x03,0x03,0x63,0x66,0x3C,0x00,0x00, // 'C'
0x00,0x1F,0x36,0x66,0x66,0x66,0x66,0x66,0x36,0x1F,0x00,0x00, // 'D'
0x00,0x7F,0x46,0x06,0x26,0x3E,0x26,0x06,0x46,0x7F,0x00,0x00, // 'E'
0x00,0x7F,0x66,0x46,0x26,0x3E,0x26,0x06,0x06,0x0F,0x00,0x00, // 'F'
0x00,0x3C,0x66,0x63,0x03,0x03,0x73,0x63,0x66,0x7C,0x00,0x00, // 'G'
0x00,0x33,0x33,0x33,0x33,0x3F,0x33,0x33,0x33,0x33,0x00,0x00, // 'H'
0x00,0x1E,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x1E,0x00,0x00, // 'I'
0x00,0x78,0x30,0x30,0x30,0x30,0x33,0x33,0x33,0x1E,0x00,0x00, // 'J'
0x00,0x67,0x66,0x36,0x36,0x1E,0x36,0x36,0x66,0x67,0x00,0x00, // 'K'
0x00,0x0F,0x06,0x06,0x06,0x06,0x46,0x66,0x66,0x7F,0x00,0x00, // 'L'
0x00,0x63,0x77,0x7F,0x7F,0x6B,0x63,0x63,0x63,0x63,0x00,0x00, // 'M'
0x00,0x63,0x63,0x67,0x6F,0x7F,0x7B,0x73,0x63,0x63,0x00,0x00, // 'N'
0x00,0x1C,0x36,0x63,0x63,0x63,0x63,0x63,0x36,0x1C,0x00,0x00, // 'O'
0x00,0x3F,0x66,0x66,0x66,0x3E,0x06,0x06,0x06,0x0F,0x00,0x00, // 'P'
0x00,0x1C,0x36,0x63,0x63,0x63,0x73,0x7B,0x3E,0x30,0x78,0x00, // 'Q'
0x00,0x3F,0x66,0x66,0x66,0x3E,0x36,0x66,0x66,0x67,0x00,0x00, // 'R'
0x00,0x1E,0x33,0x33,0x03,0x0E,0x18,0x33,0x33,0x1E,0x00,0x00, // 'S'
0x00,0x3F,0x2D,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x1E,0x00,0x00, // 'T'
0x00,0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x1E,0x00,0x00, // 'U'
0x00,0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x1E,0x0C,0x00,0x00, // 'V'
0x00,0x63,0x63,0x63,0x63,0x6B,0x6B,0x36,0x36,0x36,0x00,0x00, // 'W'
0x00,0x33,0x33,0x33,0x1E,0x0C,0x1E,0x33,0x33,0x33,0x00,0x00, // 'X'
0x00,0x33,0x33,0x33,0x33,0x1E,0x0C,0x0C,0x0C,0x1E,0x00,0x00, // 'Y'
0x00,0x7F,0x73,0x19,0x18,0x0C,0x06,0x46,0x63,0x7F,0x00,0x00, // 'Z'
0x00,0x3C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x3C,0x00,0x00, // '['
0x00,0x00,0x01,0x03,0x06,0x0C,0x18,0x30,0x60,0x40,0x00,0x00, // '\'
0x00,0x3C,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x3C,0x00,0x00, // ']'
0x08,0x1C,0x36,0x63,0x00,0x00,0x00,0x00,0x00,0x00,0x00, // '^'
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0x00, // '_'
0x0C,0x0C,0x18,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, // ``
0x00,0x00,0x00,0x00,0x1E,0x30,0x3E,0x33,0x33,0x6E,0x00,0x00, // 'a'
0x00,0x07,0x06,0x06,0x3E,0x66,0x66,0x66,0x66,0x3B,0x00,0x00, // 'b'
0x00,0x00,0x00,0x00,0x1E,0x33,0x03,0x03,0x33,0x1E,0x00,0x00, // 'c'
0x00,0x38,0x30,0x30,0x3E,0x33,0x33,0x33,0x33,0x6E,0x00,0x00, // 'd'
0x00,0x00,0x00,0x00,0x1E,0x33,0x3F,0x03,0x33,0x1E,0x00,0x00, // 'e'

```

```

0x00,0x1C,0x36,0x06,0x06,0x1F,0x06,0x06,0x06,0x0F,0x00,0x00, // 'f'
0x00,0x00,0x00,0x00,0x6E,0x33,0x33,0x33,0x3E,0x30,0x33,0x1E, // 'g'
0x00,0x07,0x06,0x06,0x36,0x6E,0x66,0x66,0x66,0x67,0x00,0x00, // 'h'
0x00,0x18,0x18,0x00,0x1E,0x18,0x18,0x18,0x18,0x7E,0x00,0x00, // 'i'
0x00,0x30,0x30,0x00,0x3C,0x30,0x30,0x30,0x30,0x33,0x33,0x1E, // 'j'
0x00,0x07,0x06,0x06,0x66,0x36,0x1E,0x36,0x66,0x67,0x00,0x00, // 'k'
0x00,0x1E,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x7E,0x00,0x00, // 'l'
0x00,0x00,0x00,0x00,0x3F,0x6B,0x6B,0x6B,0x6B,0x63,0x00,0x00, // 'm'
0x00,0x00,0x00,0x00,0x1F,0x33,0x33,0x33,0x33,0x33,0x00,0x00, // 'n'
0x00,0x00,0x00,0x00,0x1E,0x33,0x33,0x33,0x33,0x1E,0x00,0x00, // 'o'
0x00,0x00,0x00,0x00,0x3B,0x66,0x66,0x66,0x66,0x3E,0x06,0x0F, // 'p'
0x00,0x00,0x00,0x00,0x6E,0x33,0x33,0x33,0x33,0x3E,0x30,0x78, // 'q'
0x00,0x00,0x00,0x00,0x37,0x76,0x6E,0x06,0x06,0x0F,0x00,0x00, // 'r'
0x00,0x00,0x00,0x00,0x1E,0x33,0x06,0x18,0x33,0x1E,0x00,0x00, // 's'
0x00,0x00,0x04,0x06,0x3F,0x06,0x06,0x06,0x36,0x1C,0x00,0x00, // 't'
0x00,0x00,0x00,0x00,0x33,0x33,0x33,0x33,0x33,0x6E,0x00,0x00, // 'u'
0x00,0x00,0x00,0x00,0x33,0x33,0x33,0x33,0x1E,0x0C,0x00,0x00, // 'v'
0x00,0x00,0x00,0x00,0x63,0x63,0x6B,0x6B,0x36,0x36,0x00,0x00, // 'w'
0x00,0x00,0x00,0x00,0x63,0x36,0x1C,0x1C,0x36,0x63,0x00,0x00, // 'x'
0x00,0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x3C,0x30,0x18,0x0F, // 'y'
0x00,0x00,0x00,0x00,0x3F,0x31,0x18,0x06,0x23,0x3F,0x00,0x00, // 'z'
0x00,0x38,0x0C,0x0C,0x06,0x03,0x06,0x0C,0x0C,0x38,0x00,0x00, // '{'
0x00,0x18,0x18,0x18,0x18,0x00,0x18,0x18,0x18,0x18,0x00,0x00, // '|'
0x00,0x07,0x0C,0x0C,0x18,0x30,0x18,0x0C,0x0C,0x07,0x00,0x00, // '}'
0x00,0xCE,0x5B,0x73,0x00,0x00,0x00,0x00,0x00,0x00,0x00, // '~'
};

```

```

typedef struct typFNT_GB16 {          /* 汉字字模显示数据结构 */
    char Index[2];
    char Msk[32];
};

```

```

struct typFNT_GB16 xdata GB_16[] = { /* 显示为 16×16 */
    "中", 0x01,0x00,0x01,0x00,0x21,0x08,0x3F,0xFC,
           0x21,0x08,0x21,0x08,0x21,0x08,0x21,0x08,
           0x21,0x08,0x3F,0xF8,0x21,0x08,0x01,0x00,
           0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,
    "文", 0x02,0x00,0x01,0x00,0x01,0x00,0xFF,0xFE,
           0x08,0x20,0x08,0x20,0x08,0x20,0x04,0x40,
           0x04,0x40,0x02,0x80,0x01,0x00,0x02,0x80,
           0x04,0x60,0x18,0x1E,0xE0,0x08,0x00,0x00,
    "测", 0x40,0x02,0x27,0xC2,0x24,0x42,0x84,0x52,
           0x45,0x52,0x55,0x52,0x15,0x52,0x25,0x52,

```

```

    0x25,0x52,0x25,0x52,0xC5,0x52,0x41,0x02,
    0x42,0x82,0x42,0x42,0x44,0x4A,0x48,0x04,
    "试", 0x00,0x20,0x40,0x28,0x20,0x24,0x30,0x24,
    0x27,0xFE,0x00,0x20,0xE0,0x20,0x27,0xE0,
    0x21,0x20,0x21,0x10,0x21,0x10,0x21,0x0A,
    0x29,0xCA,0x36,0x06,0x20,0x02,0x00,0x00,
};

```

```

uchar gCurRow,gCurCol;      /* 当前行、列存储, 行高 16 点, 列宽 8 点 */

```

```

/***** 获取当前行函数 *****/

```

```

* 功能: 将当前行数据返回给主调函数。

```

```

/*****

```

```

uchar fnGetRow(void) {
    return gCurRow;
}

```

```

/***** 获取当前列函数 *****/

```

```

* 功能: 将当前列数据返回给主调函数。

```

```

/*****

```

```

uchar fnGetCol(void) {
    return gCurCol;
}

```

```

/***** 状态位 ST1,ST0 判断函数 *****/

```

```

* 功能: 读写指令和读写数据。

```

```

/*****

```

```

uchar fnST01(void) {
    uchar i;
    for(i=10;i>0;i--) {
        if((LCMCW & 0x03) == 0x03)
            break;
    }
    return i;          /* 若返回零, 说明错误 */
}

```

```

/***** 状态位 ST2 判断函数 *****/

```

```

* 功能: 数据自动读取。

```

```

/*****

```

```

uchar fnST2(void) {
    uchar i;
    for(i=10;i>0;i--) {

```

```

        if((LCMCW & 0x04) == 0x04)
            break;
    }
    return i;                /* 若返回零, 说明错误 */
}

/***** * 状态位 ST3 判断函数 *****/
* 功能: 数据自动写。
*****/
uchar fnST3(void) {
    uchar i;
    for(i=10;i>0;i--) {
        if((LCMCW & 0x08) == 0x08)
            break;
    }
    return i;                /* 若返回零, 说明错误 */
}

/***** * 状态位 ST6 判断函数 *****/
* 功能: 屏读 / 屏拷贝。
*****/
uchar fnST6(void) {
    uchar i;
    for(i=10;i>0;i--) {
        if((LCMCW & 0x40) == 0x40)
            break;
    }
    return i;                /* 若返回零, 说明错误 */
}

/***** * 写双参数命令函数 *****/
uchar fnPR1(uchar uCmd,uchar uPar1,uchar uPar2) {
    if(fnST01() == 0)
        return 1;
    LCMDW = uPar1;
    if(fnST01() == 0)
        return 2;
    LCMDW = uPar2;
    if(fnST01() == 0)
        return 3;
    LCMCW = uCmd;
    return 0;                /* 返回 0 成功 */
}

```

```
/****** 写单参数命令函数 ******/
```

```
uchar fnPR11(uchar uCmd,uchar uPar1) {  
    if(fnST01() == 0)  
        return 1;  
    LCMDW = uPar1;  
    if(fnST01() == 0)  
        return 2;  
    LCMCW = uCmd;  
    return 0;          /* 返回 0 成功 */  
}
```

```
/****** 写无参数命令函数 ******/
```

```
uchar fnPR12(uchar uCmd) {  
    if(fnST01() == 0)  
        return 1;  
    LCMCW = uCmd;  
    return 0;          /* 返回 0 成功 */  
}
```

```
/****** 写数据函数 ******/
```

```
uchar fnPR13(uchar uData) {  
    if(fnST3() == 0)  
        return 1;  
    LCMDW = uData;  
    return 0;          /* 返回 0 成功 */  
}
```

```
/****** 读数据函数 ******/
```

```
uchar fnPR2(void) {  
    if(fnST01() == 0)  
        return 1;  
    return LCMDW;  
}
```

```
/****** 设置当前地址函数 ******/
```

```
void fnSetPos(uchar urow, uchar ucol) {  
    uint iPos;  
    iPos = urow * 30 + ucol;  
    fnPR1(LC_ADD_POS,iPos & 0xFF,iPos / 256);  
    gCurRow = urow;  
    gCurCol = ucol;  
}
```

```

/***** 设置当前显示行、列函数 *****/
void cursor(uchar uRow, uchar uCol) {
    fnSetPos(uRow * 16, uCol);
}

/***** 清屏函数 *****/
void cls(void) {
    uint i;
    fnPR1(LC_ADD_POS, 0x00, 0x00); /* 置地址指针 */
    fnPR12(LC_AUT_WR);             /* 自动写 */
    for(i=0; i<240*30; i++) {
        fnST3();
        fnPR13(0x00);              /* 写数据 */
    }
    fnPR12(LC_AUT_OVR);            /* 自动写结束 */
    fnPR1(LC_ADD_POS, 0x00, 0x00); /* 重置地址指针 */
    gCurRow = 0;                  /* 置地址指针存储变量 */
    gCurCol = 0;
}

/***** 液晶模块初始化函数 *****/
char fnLCMInit(void) {
    if(fnPR1(LC_TXT_STP, 0x00, 0x00) != 0) /* 文本显示区首地址 */
        return -1;
    fnPR1(LC_TXT_WID, 0x1E, 0x00);         /* 文本显示区宽度 */
    fnPR1(LC_GRH_STP, 0x00, 0x00);         /* 图形显示区首地址 */
    fnPR1(LC_GRH_WID, 0x1E, 0x00);         /* 图形显示区宽度 */
    fnPR12(LC_CUR_SHP | 0x01);             /* 光标形状 */
    fnPR12(LC_MOD_OR);                     /* 显示方式设置 */
    fnPR12(LC_DIS_SW | 0x08);              /* 显示开关设置 */
    return 0;
}

/***** ASCII 码(8×16)及汉字(16×16)显示函数 *****/
uchar dprintf(char *fmt, ...) {
    va_list arg_ptr;
    char c1, c2, cData;
    char tmpBuf[64];                      /* LCD 显示数据缓冲区 */
    uchar i=0, j, uLen, uRow, uCol;
    uint k;
    va_start(arg_ptr, fmt);
    uLen = (uchar)vsprintf(tmpBuf, fmt, arg_ptr);

```



```

va_end(arg_ptr);
while(i<uLen) {
    c1 = tmpBuf[i];
    c2 = tmpBuf[i+1];
    uRow = fnGetRow();
    uCol = fnGetCol();
    if(c1 >= 0) {
        /* ASCII 码 */
        if(c1 < 0x20) {
            switch(c1) {
                case CR:
                case LF: i++;
                    /* 回车或换行 */
                    if(uRow < 112)
                        fnSetPos(uRow+16,0);
                    else
                        fnSetPos(0,0);
                    continue;
                case BS:
                    /* 退格 */
                    if(uCol > 0) uCol--;
                    fnSetPos(uRow,uCol);
                    cData = 0x00;
                    break;
                default:
                    /* 其他 */
                    c1 = 0x1f;
            }
        }
        for(j=0;j<16;j++) {
            fnPR12(LC_AUT_WR);
            /* 写数据 */
            if(c1 >= 0x1f) {
                if(j < (16-ASC_CHR_HEIGHT)) fnPR13(0x00);
                else
                    fnPR13(ASC_MSK[(c1-0x1f)*ASC_CHR_HEIGHT+
                        j-(16-ASC_CHR_HEIGHT)]);
            }
            else
                fnPR13(cData);
            fnPR12(LC_AUT_OVR);
            fnSetPos(uRow+j+1,uCol);
        }
        if(c1 != BS) uCol++;
        /* 非退格 */
    }
    else {
        /* 中文 */
        for(j=0;j<sizeof(GB_16)/sizeof(GB_16[0]);j++) {

```

```

        if(c1 == GB_16[j].Index[0] && c2 == GB_16[j].Index[1])
            break;
    }
    for(k=0;k<sizeof(GB_16[0].Msk)/2;k++) {
        fnSetPos(uRow+k,uCol);
        fnPR12(LC_AUT_WR);                /* 写数据 */
        if(j < sizeof(GB_16)/sizeof(GB_16[0])) {
            fnPR13(GB_16[j].Msk[k*2]);
            fnPR13(GB_16[j].Msk[k*2+1]);
        }
        else {                            /* 未找到该字 */
            if(k < sizeof(GB_16[0].Msk)/4) {
                fnPR13(0x00);
                fnPR13(0x00);
            }
            else {
                fnPR13(0xff);
                fnPR13(0xff);
            }
        }
        fnPR12(LC_AUT_OVR);
    }
    uCol += 2;
    i++;
}

if(uCol >= 30) {                        /* 光标后移 */
    uRow += 16;
    if(uRow < 0x80) uCol -= 30;
    else {
        uRow = 0;
        uCol = 0;
    }
}

fnSetPos(uRow,uCol);
i++;
}

return uLen;
}

```

```

/***** 主函数 *****/
void main(void) {
    fnLCMInit();
    cls();
}

```

```
cursor(0,0);  
dprintf("%s","This is a test: 中文测试");  
}
```

8.10 80C552 单片机内部 A/D 转换器的 Cx51 驱动程序

8.10.1 80C552 单片机简介

80C552 是荷兰 PHILIPS 公司生产的增强型 80C51 单片机, 它除了具有常规 80C51 单片机的全部功能之外, 又增加了许多功能, 在指令系统上与 8051 单片机完全兼容, 在硬件上具有如下资源特性:

- 80C51CPU, 8K 字节的内部 ROM (83C552) 或 EPROM (87C552), 可在外部扩展 64K 字节 ROM;
- 256 字节的内部 RAM, 可在外部扩展 64K 字节 RAM 或 I/O;
- 2 个标准 16 位定时器, 1 个附加 16 位定时器, 配有 4 个捕捉寄存器和 3 个比较寄存器;
- 1 个 8 路 10 位片内 A/D 转换器, 两路 8 位分辨率的脉冲宽度调制器输出 PWM;
- 5 个 8 位的 I/O 口, 1 个与 A/D 合用的输入口;
- 1 个全双工异步串行口 UART, I²C 串行总线接口, 监视定时器 WDT;
- 15 个中断向量。

图 8.84 为 80C552 的结构框图, 图 8.85 为 68 脚 PLCC 封装形式的 80C552 引脚功能分配图。

各引脚的功能如下:

V _{DD} :	+5V 电源。
EA:	存储器访问选择输入端, 为“0”时访问外部 ROM, 为“1”时访问内部 ROM。
PSEN:	外部 ROM 的读选通信号。
ALE:	地址锁存允许信号, 当访问外部 ROM 时, ALE 的有效输出用来锁存低 8 位地址信号。
STADC:	片内 A/D 转换器的启动输入, 该脚不得悬空。
PWM0:	脉宽调制器 PWM 通道 0 输出。
PWM1:	脉宽调制器 PWM 通道 1 输出。
EW:	监视定时器 WDT 的时钟使能端, 为“0”时允许 WDT 时钟和禁止低功耗方式, 为“1”时禁止 WDT 时钟和允许低功耗方式, 该脚不得悬空。
P0.0~P0.7:	8 位准双向 I/O 口。
P1.0~P1.7:	8 位准双向 I/O 口。
P2.0~P2.7:	8 位准双向 I/O 口。
P3.0~P3.7:	8 位准双向 I/O 口。
P4.0~P4.7:	8 位准双向 I/O 口。

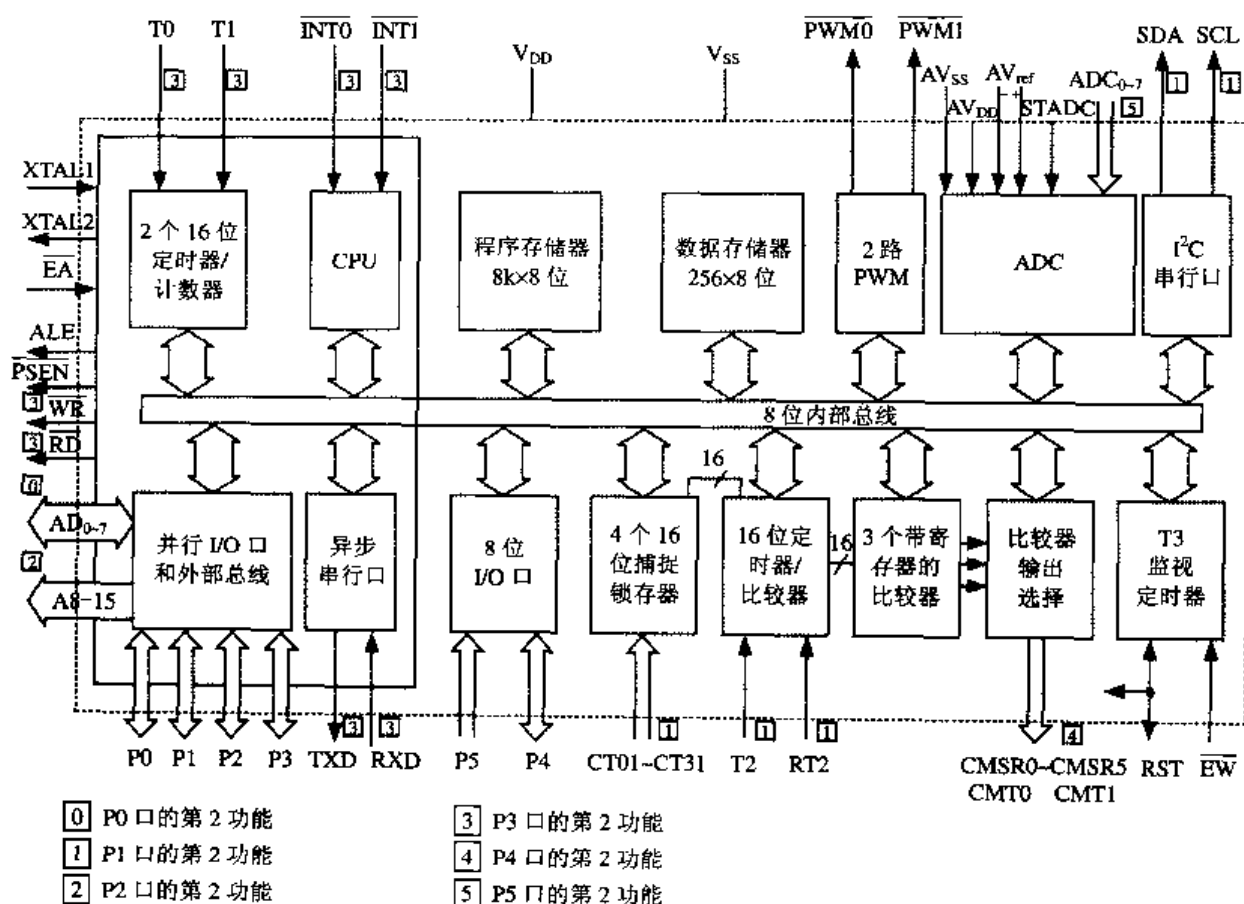


图 8.84 80C552 的结构框图

P5.0~P5.7: 8 位输入口。

RST: 复位输入端，当监视定时器 WDT 计数溢出时，输出复位信号。

XTAL1~2: 振荡器输入端。

V_{SS}: 数字地。

AV_{SS}: 模拟地。

AV_{ref+}: A/D 转换器基准电压输入端 (+)。

AV_{ref-}: A/D 转换器基准电压输入端 (-)。

AV_{DD}: 模拟电源。

8XC552 单片机有 6 个 8 位的输入输出端口，其中 P0、P2、P3 口的结构和功能与 8051 基本相同。P1 口除保留了 8051 的输入输出功能之外，还增加了第二功能。P1.0~P1.3 的第二功能是捕捉寄存器的跳变信号输入端；P1.4 的第二功能是定时器 T2 的外部输入端；P1.5 的第二功能是 T2 的复位输入端；P1.6 的第二功能是 I²C 总线的时钟线 SCL；P1.7 的第二功能是 I²C 总线的数据线 SDA。需要指出的是，P1.6 和 P1.7 无内部上拉电阻，因此将它们用作输入输出线时，要外接上拉电阻。

P4 口是 8 位双向口，输出能带 4 个 LS TTL 电路。它的第二功能是在预定的时刻（定时器 T2 与 3 个不同的比较器匹配时）P4 口的某些位能自动置位、复位和触发翻转。

P5 口是 8 位开关量信号输入口。它的第二功能是 8 路 10 位 A/D 转换器的模拟信号输入端。当 P5 口同时用作开关量和模拟量信号输入时，应考虑到通道之间的串扰。

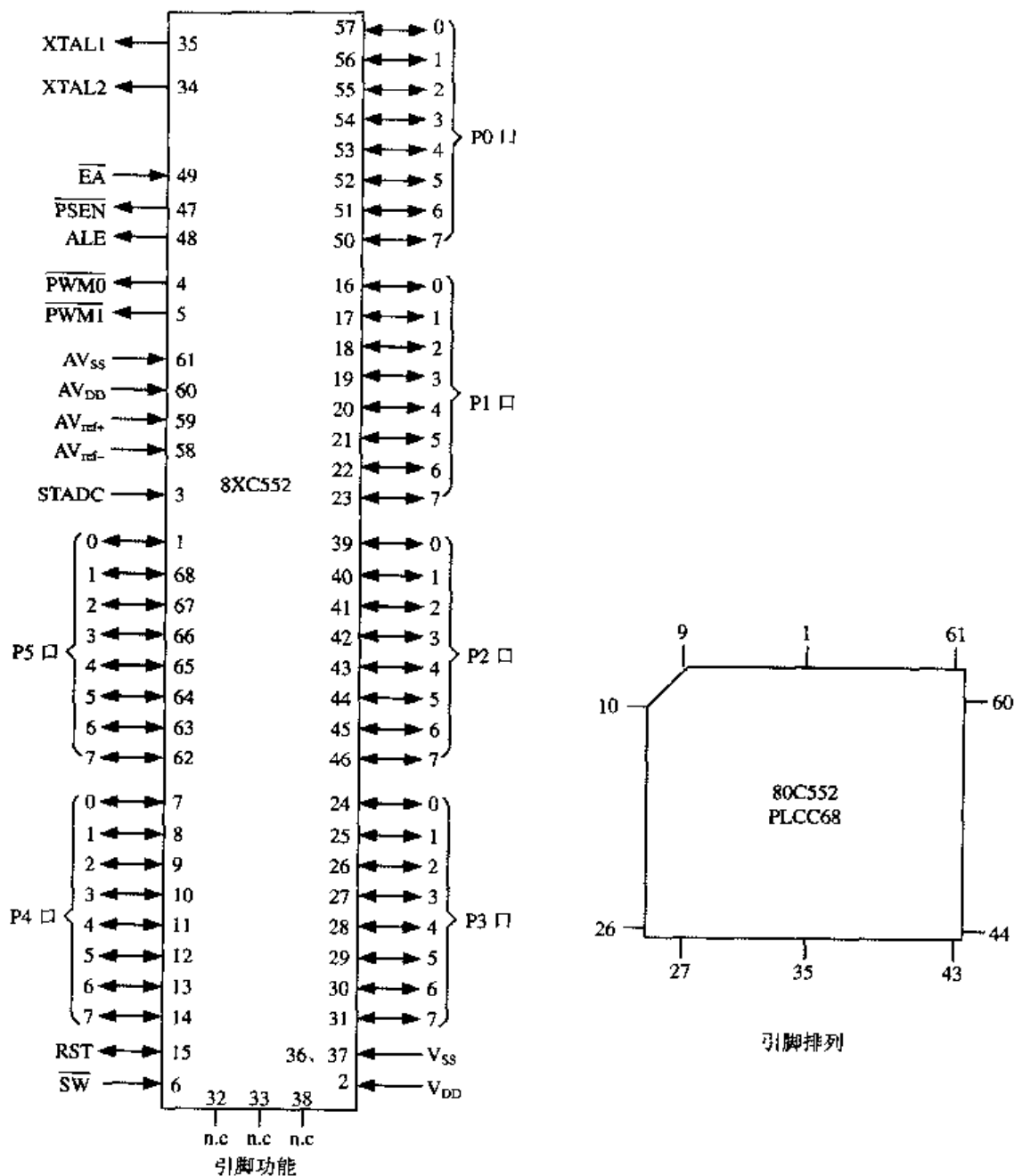


图 8.85 80C552 的引脚功能分配 (68 脚 PLCC 封装)

8.10.2 80C552 内部 ADC 的应用

80C551 单片机内部集成了一个 8 通道输入、10 位分辨率的逐次比较式 A/D 转换器 (ADC)。该 ADC 具有单独的模拟电源和参考电压, 从而可最大限度地减少数字电路的干扰。ADC 完成一次转换需要 50 个机器周期, 当使用 30MHz 的振荡频率时转换时间为 $20\mu s$, 模拟输入信号电压范围为 0~5V。下面介绍 80C552 内部 ADC 的工作过程, 讨论一些可能影响转换结果精度的设计问题, 并给出一个 C51 应用程序范例。

图 8.86 所示为 80C552 内部 ADC 的逻辑框图。对 A/D 转换器的操作是通过访问特殊功能寄存器 ADCON 来实现的, 并且 ADCON 寄存器只能通过字节寻址的方法访问。

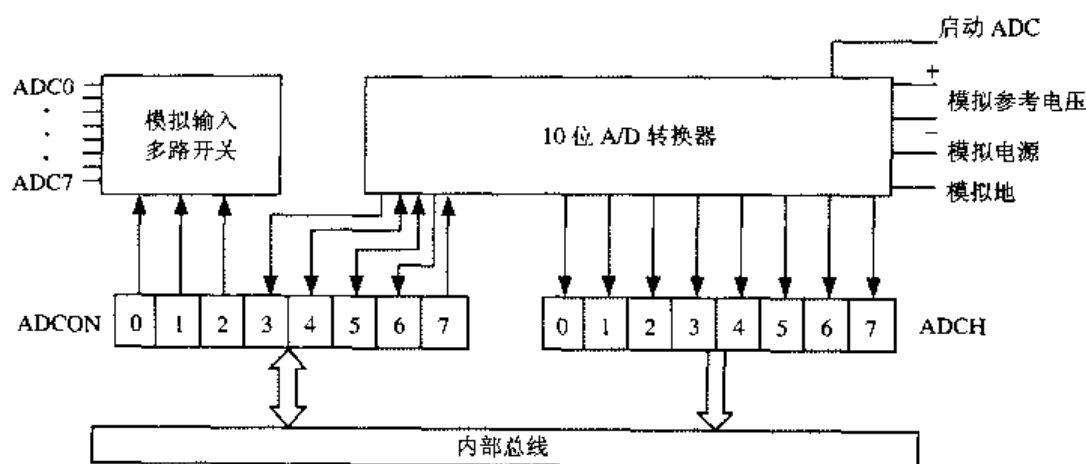


图 8.86 80C552 内部 A/D 转换器的逻辑框图

ADCON 寄存器的地址为 0C5H，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
ADC.1	ADC.0	ADEX	ADCI	ADCS	AADR2	AADR1	AADR0

其中各位的意义如下：

ADC.1: A/D 转换结果位 1。

ADC.0: A/D 转换结果位 0。

ADEX: 允许从外部引脚 STADC 上输入信号启动 A/D 转换。ADEX=0，禁止外部输入 STADC 启动，只能由软件启动 A/D 转换；ADEX=1，允许外部输入 STADC 启动，可由软件或外部启动 A/D 转换，如由外部信号的上升沿启动 A/D 转换，则外部信号应在 STADC 端上至少保持一个机器周期的高电平。

ADCI: ADC 中断标志。A/D 转换结束时由硬件置位，如果允许，则将引起中断。该标志由中断服务程序清“0”，但它不能用软件来置位。

ADCS: ADC 启动和状态标志。该位置“1”时启动 A/D 转换。它可由软件或外部 STADC 置位。A/D 转换正在进行时控制逻辑保证该位为“1”，A/D 转换结束后，该位清“0”，同时置位中断标志。ADCS 不能用软件来复位。ADCS 与 ADCI 有 4 种状态组合，其中三种为有效功能状态：

ADCS	ADCI	功能
0	0	ADC 空闲，可以启动 A/D 转换。
0	1	ADC 忙，禁止再启动 A/D 转换。
1	0	ADC 转换完成，禁止重新启动 A/D 转换。
1	1	无效。

AADR2~AADR0 是模拟通路选择。AADR2~AADR0 共有 8 种状态组合，即 000~111，分别选择 8 路模拟输入信号中的 1 路（P5.0~P5.7）输入到 A/D 转换器，只有在 ADCS 和 ADCI 都为低电平时才能变化。

当使用 80C552 内部 ADC 时, 模拟信号从 80C552 的 P5 口输入, 经过内部多路开关进入 ADC。启动 ADC 转换可以采用软件方式, 也可以采用硬件方式。当采用软件启动时 (置位 ADCON 寄存器中的 ADCS 位), 内部启动信号立即启动 ADC 进行一次 A/D 转换。采用硬件启动时, 由 STDAC 引脚上的一个上升沿实现 A/D 转换的硬件触发, 这时加在 STADC 引脚上的低电平和高电平持续时间应至少为 1 个机器周期。在进行 A/D 转换的过程中, 不理睬外部或软件的启动信号, 转换结束后, 置位 ADCI, 并将转换结果的高 8 位存放在特殊功能寄存器 ADCH 中, 低 2 位存放在特殊功能寄存器 ADCON 的最高两位 ADC.1 和 ADC.0 中。

80C552 中 A/D 转换器的中断入口地址为 0053H。

图 8.87 所示为 A/D 转换器的实现原理。A/D 转换器具有自己独立的电源引脚 AV_{DD} 和 AV_{SS} 以及 2 个连到 T 型电阻网络的引脚 AV_{ref+} 和 AV_{ref-} 。 AV_{ref+} 和 AV_{ref-} 可以在 $AV_{DD}+0.2V$ 和 $AV_{SS}-0.2V$ 之间。A/D 转换结果由下式计算:

$$\text{转换结果} = 1024 \times (V_{in} - AV_{ref-}) / (AV_{ref+} - AV_{ref-})$$

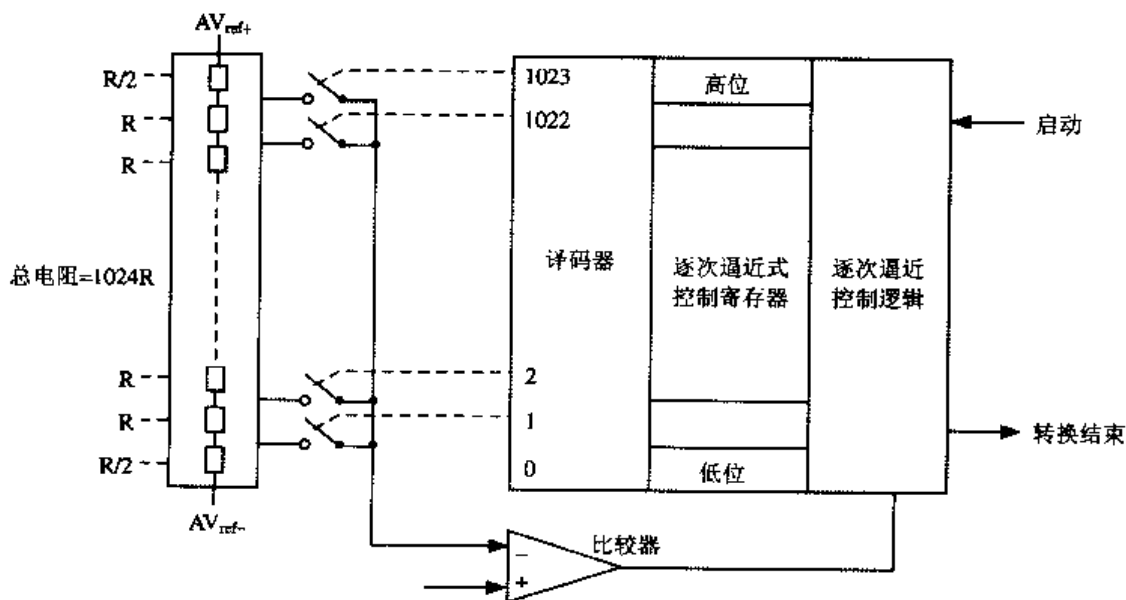


图 8.87 80C552 内部 ADC 的实现原理

虽然 80C552 内部 ADC 具有 10 位的分辨率, 在实际应用中必须仔细设计才能达到这个分辨率。模拟输入信号的值必须在 V_{ref+} 和 V_{ref-} 之间, 其动态范围是 $V_{ref} = (V_{ref+} - V_{ref-})$ 。这个动态范围有一个最小极限, 该极限值取决于比较器的增益。1LSB ($1LSB = V_{ref}/1024$) 的比较器输入差分电压应该可以在逐次比较寄存器的输入端产生“1”或“0”电平, 如果不是这样, ADC 将达不到 10 位的分辨率。80C552 中比较器至少需要 0.3mV 的差分输入电压来产生一个有效的逻辑输出电平, 这意味着要使 ADC 达到 10 位的分辨率, V_{ref} 至少应为 $1024 \times 0.3mV$ 。决定这个动态范围的 V_{ref+} 和 V_{ref-} 的绝对值不能超过 AV_{DD} 和 AV_{SS} 。

对模拟输入信号的变化速率也必须加以限制。其原因有两个, 一是保证对模拟信号采样的精度, 二是为了防止模拟信号变化过快导致转换结果与输入信号根本没有联系, 从而使 A/D 转换结果失去意义。

为了能从 ADC 中稳定地读取数据, 模拟输入信号在采样过程中应当是稳定的。例如, 如果要求达到 $1/2\text{LSB}$ 的稳定性, 则模拟输入信号在 6 个机器周期中的变化就不允许超过 $1/2\text{LSB}$ 。在这种情况下模拟输入信号的最大变化速率是: $dV/dt=0.5\text{LSB}/6T$, 其中, T 是机器周期。如果模拟输入信号的变化速率超过允许的最大变化速率, A/D 转换的精度将迅速下降, 转换结果将不再与输入信号有确定的关系。实验表明这时最可能产生的转换结果是 3FFH。为了避免上述误差的产生, 使用 80C552 内部 ADC 时, 必须使模拟输入信号的变化速率小于 10V/ms 。

可以采用以下措施来降低模拟输入信号的变化速率。

- 从低输入阻抗的信号源上采集信号以减少串扰。
- 将模拟输入信号线与数字输入信号线分开, 在印刷版上采用一个接地环将模拟输入信号线与数字信号线隔离。
- 在 80C552 的 P5 口引脚上不要混接模拟信号和数字信号。
- 在模拟输入通道中增加 RC 滤波器, 滤波器的时间常数应大于 $50\mu\text{s}$ 。

对于 5Vpp 的输入信号, 最大变化速率 10V/ms 的限制对应于最大输入频率为 637Hz 。根据奈奎斯特判据 (即采样频率 \geq 信号中最大频率分量的 2 倍), 当 80C552 工作在 1.2MHz 的时钟频率时 (每 $500\mu\text{s}$ 转换一次) 最大输入信号频率为 1kHz 。因此 80C552 内部 ADC 的输入信号最高允许频率是由变化速率决定的, 而增加 80C552 的晶体振荡频率并不一定能够使输入信号频率成比例地上升。

模拟输入信号源的输出阻抗应足够小以便使 A/D 转换结果不会产生重大误差。输出阻抗对于转换精度的影响有两点: 信号源内阻会产生电压降; 向采样电容充电时有一定的时间常数。图 8.88 是 80C552 内部 ADC 的输入电路。Cs 代表杂散电容, Cc 代表采样电容, Rm 代表多路转换开关的串联电阻。ADC 在工作时具有一定的输入电流, 即泄漏电流。泄漏电流应小于 $1\mu\text{A}$ 。由于泄漏电流的作用, 在输入信号源的输出阻抗 R_s 上将产生一个电压降。被转换的电压是采样电容上的电压, 该电压是输入电压 V_{IN} 减去 R_s 上的电压降。因此 R_s 上的电压降将在输入信号电压 V_{IN} 的转换结果中产生误差。

当要求达到 $1/2\text{LSB}$ 的转换精度时, 最大信号源内阻 R_s 应满足:

$$R_s \leq 0.5\text{LSB}/I_1 = (0.5 \times (V_{\text{ref}}/1024)) / I_1$$

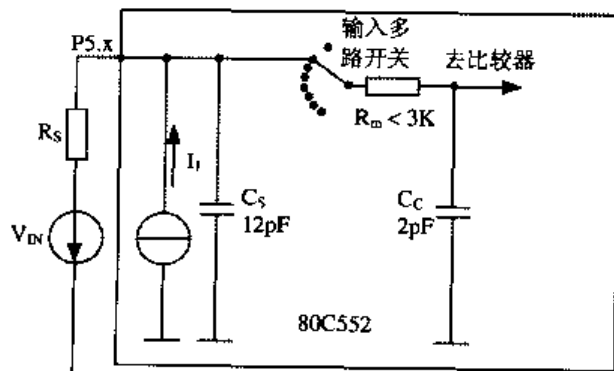


图 8.88 80C552 内部 ADC 的输入电路

例如, $V_{\text{ref}}=5.12\text{V}$, $I_1=1\mu\text{A}$, 则 R_s 应小于 $2.5\text{K}\Omega$ 。如果信号源内阻不能达到要求,

则应使用一个具有足够低输出内阻的缓冲器。还可以在缓冲器中加入滤波器使输入信号的变化速率限制在 10V/ms 之内。

图 8.88 也可用于说明模拟输入电路的等效电容的数值范围。该等效电容由杂散电容和实际采样电容所组成。这些电容必须在 80C552 的 4 个机器周期内充电完毕, 因此为保证给电容快速充电就必须限制信号源的最大输出阻抗。

8.10.3 使用 80C552 内部 ADC 时印刷电路板的设计要点

80C552 单片机具有独立的模拟电源和数字电源引脚。在片上模拟电源引脚 AV_{DD} 和 AV_{SS} 信号引线之间的并联负载导纳将使模拟交流电流 I_a 从 AV_{DD} 中流入, 从 AV_{SS} 中流出。同样, 数字交流电流也会从数字电源引脚 V_{DD} 和 V_{SS} 之间流过, 如图 8.89 所示。因为共同导纳在片外更加难以实现, 所以对于 I_a 和 I_d 必须分别建立一条低阻抗的信号通道, 这个问题可通过解耦电容来解决。对于模拟电路部分在 AV_{SS} 与 AV_{DD} 之间加上去耦电容来实现模拟部分去耦合; 对于数字电路部分在 V_{DD} 与 V_{SS} 之间加上去耦电容来实现数字部分的去耦合。

为保证低阻抗的地线回路, 去耦电容 (例如 100nF 的陶瓷电容) 必须尽可能地靠近 AV_{SS} 和 AV_{DD} 以最大限度地减少电源回路的面积。在电源线上串联电感可以改善去耦效果 (例如采用 1~5 μ H 的电感)。采用这种去耦设计, 可以将模拟电源和数字电源引线一起接到一个单独的 +5V 稳压电源上。

当在一个印刷版上同时有模拟和数字电路时, 通常应尽可能地使模拟电源和模拟地与数字电源和数字地隔离。这样可以减少通过公共地线阻抗产生的串扰, 公共地线阻抗可以使来自数字电路的噪音串入敏感的模拟电路。

但是在某些地方模拟地与数字地必须连接在一起, 当连接点与单片机相距很远时将增加模拟地与数字地之间的阻抗。这将在模拟地与数字地引线之间产生一个相当大的差分电压。单片机内部的模拟和数字电路将在不同的地电平上工作从而产生不正常现象。大接地阻抗的另一个影响是模拟与数字电路之间可能通过内部寄生电容和衬底接触而产生串扰。

为了防止地电平误差, 80C552 内部的模拟地与数字地通过一个 2 Ω 的电阻连接在一起 (如图 8.89 所示), 这样就使两个地保持在同一个直流电平上。但是这个内部电阻并不意味着可以采用一个公共地线引脚来导通模拟与数字地电流。

当电源引线解耦良好时, 在片上电源之间的并联导纳将保证流过 AV_{DD} 和 V_{DD} 引脚的电源电流的交流部分分别通过 AV_{SS} 与 V_{SS} 而离开, 即使 AV_{SS} 与 V_{SS} 通过一个 2 Ω 的电阻连接在一起。设计印刷电路板时最好在单片机外边将 AV_{SS} 与 V_{SS} 连接在一起。建议采用一个与单片机芯片一样大的地线面积。如果除了 80C552 之外, 还需要采用其他的模拟集成电路器件, 则应该采用两个分离的地, 这时 80C552 的 AV_{SS} 和 V_{SS} 接到模拟地上, 而数字地可以线驱动器作为返回通路。这将产生一个比模拟地干扰略大的地。对于数字电路来说这个“不太干净的地”不是大问题, 因为数字电路具有确定的噪声容许门限。另一个方案是采用星型地线, 即在 80C552 的 ADC 区域将 AV_{SS} 与 V_{SS} 连接在一起。要注意防止在除了电源以外的其他地方产生地线回路。

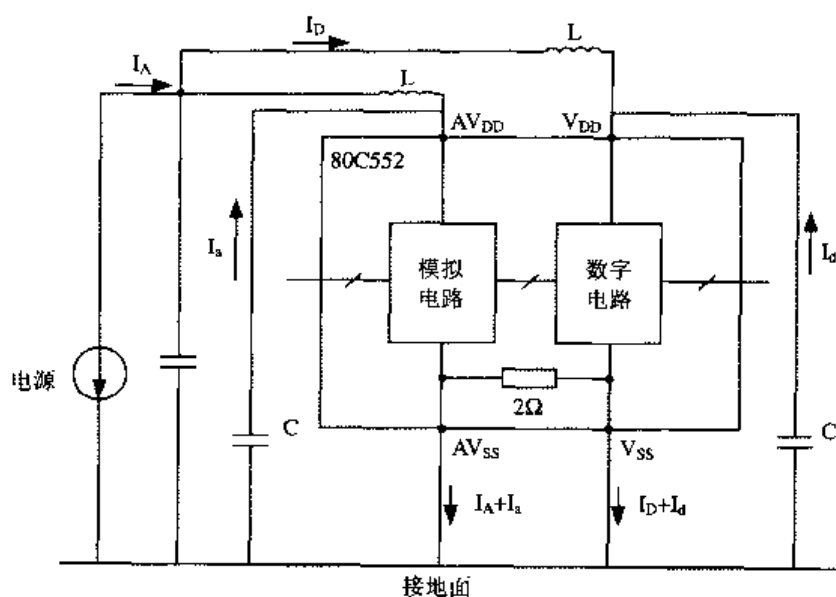


图 8.89 80C552 中模拟电路与数字电路的解耦和接地

如果在使用 80C552 时还需要采用外部数字电路，地电平的不同可能会牺牲噪声电容门限。可以通过在 80C552 的地线引脚与有关的数字电路之间连接两个反向并联的二级管来减轻这种影响，如图 8.90 所示。与单片机有关的外部电路应尽可能地靠近单片机芯片以减小高频干扰。高频干扰信号可能来自外部，也可能是由单片机自身所产生的。当干扰信号来自外部时，可能导致单片机工作不正常，如果来自单片机内部，则可能产生不希望的辐射。

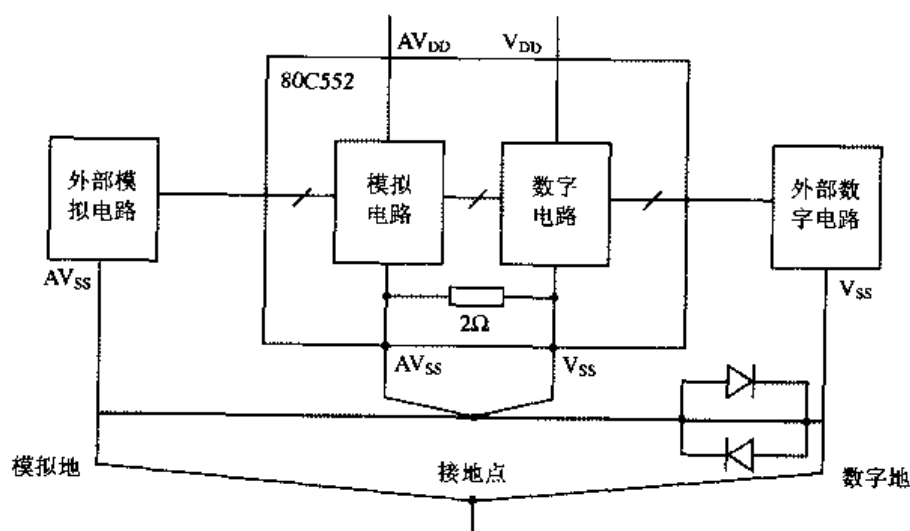


图 8.90 80C552 外部电路的接地

在不能实现一个公共地时，80C552 和直接连接到它上面的电路必须有一个局部地。局部地应当在某个地方与主地线有一个连接点。如果由于布线的原因不能使外部电路靠近单片机，则可以在单片机与外部电路之间采用 RC 滤波器，因为单片机对于瞬时高压 (30~40V) 脉冲十分敏感。虽然这些脉冲不会损坏单片机，但可能导致单片机工作不正常。

随着单片机速度的增加,片上的 I/O 驱动器的工作带宽也会随之增加,因而对瞬时脉冲也就更加敏感。

8.10.4 使用 80C552 内部 ADC 的 Cx51 驱动程序

下面给出了两个使用 80C552 内部 ADC 的 C51 驱动程序。程序 ADC_pol.c 采用查询工作方式,程序 ADC_int.c 采用中断工作方式。

当 80C552 的 STADC 引脚上出现一个上升沿跳变时程序即开始扫描 ADC 的所有输入端。这可以通过在 STADC 与地之间接一个电阻,同时在 STADC 与 V_{DD} 之间接一个开关电路来实现。如果 STADC 连到 P4.0 上,则内部 ADC 将每隔 1.14ms 对所有 8 个通道进行一次 A/D 转换。用内部定时器 T2 控制定时。当 P5 口上的所有模拟输入信号都被转换后,转换结果被送往 UART 串口,并可从 PC 机的终端上看到,因此提供了一个输出程序 output.c,应用时将它与程序文件“ADC_plc.c”或“ADC_int.c”连接在一起。80C552 的工作频率为 11.0592MHz。PC 机终端通信要求为:8 位数据位,无校验位,1 位停止位,波特率为 19200。

在使用 80C552 内部 ADC 时,还应注意以下几点。

- ADCS 和 ADCI 必须在编程 AADR0~AADR2 之前予以清零。
- 必须先对 AADR0~AADR2 编程以确定 ADC 通道,然后再置位 ADCS (软件启动)或 ADEX (硬件启动)。
- 使用查询工作方式时,应测试 ADCI 位以决定 A/D 转换是否结束,而不要测试 ADCS 位。因为当使用软件启动 ADC 时,在向 ADCS 写入“1”与从 ADCON 寄存器中的 ADCS 位读出“1”状态之间存在 2 个机器周期的延时。当使用硬件启动 ADC 时,ADCS 何时为“1”是不知道的,它取决于外部启动脉冲。

ADC_pol.c 程序代码如下:

```

/*****
* 功能描述: 在 STADC 引脚的上升沿之后扫描所有 ADC 通道。由定时器 T2
*           控制在 P4.7 引脚上产生重复周期为 1.14ms 的上升沿信号。
*           A/D 转换结果被送往 UART。
*****/
#include <reg552.h>
#define ADEX 0x20
#define ADCI 0x10
#define ADCS 0x08

void write_UART(unsigned int *, unsigned int);

void main(void) {
    unsigned int conversion, result_ADC[8];
    unsigned char ADC_Channel;
    S0CON=0x40;          /* 串行口初始化 */

```

```

TH1=TL1=0xfd;          /* 19200 Baud @11.0592MHz */
TMOD=0x20;
TR1=1;
TM2CON=0xfd;          /* 定时器 T2 初始化: osc/96 */
RTE=0x80;              /* 在 STADC 的上升沿开始 A/D 转换,转换速率=1.14ms */
conversion=0;
while(1) {
    for (ADC_Channel=0; ADC_Channel<8; ADC_Channel++) {
        ADCON=0;          /* 在选通 ADC 通道之前 */
        ADCON=ADC_Channel; /* 保证 ADCI 和 ADCS 被清 0 */
        if (ADC_Channel==0)
            ADCON=ADEX;    /* ADC0: 外部启动 */
        else
            ADCON=ADCON|ADCS; /* ADC1...ADC7: 软件启动 */
        while((ADCON&ADCI)==0); /* 检测 ADCI 状态并等待 A/D 转换结束 */
        result_ADC[ADC_Channel]=5*((256*ADCH+(ADCON&0xc0))>>6);
            /* 计算相应于 5.12V 参考电压的 10 位 A/D 转换结果 */
    }
    write_UART(&result_ADC, conversion++); /* 结果送往 UART */
    if (conversion==10000)
        conversion=0;
}
}

```

ADC_int.c 程序代码如下:

```

/*****
* 功能描述: 在 STADC 引脚的上升沿之后扫描所有 ADC 通道。由定时器 T2
*           控制在 P4.7 引脚上产生重复周期为 1.14ms 的上升沿信号。
*           A/D 转换结果被送往 UART。
*****/
#include <reg552.h>
#define ADEX 0x20
#define ADCI 0x10
#define ADCS 0x08
#define ADCIn 0x0ef
#define FALSE 0
#define TRUE 1

void write_UART(unsigned int *, unsigned int);
bit conversion_finished;

void main(void) {

```

```

unsigned int conversion, result_ADC[8];
unsigned char ADC_Channel;
S0CON=0x40;          /* 串行口初始化 */
TH1=TL1=0xfd;        /* 19200 Baud @11.0592MHz */
TMOD=0x20;
TR1=1;
TM2CON=0xfd;         /* 定时器 T2 初始化: osc/96 */
RTE=0x80;            /* 在 STADC 的上升沿开始 A/D 转换, 转换速率=1.14ms */
EAD=1;               /* ADC 开中断 */
EA=1;
conversion_finished=FALSE;
ADC_Channel=conversion=0;
ADCON=0;              /* 第一次转换由外部启动 */
ADCON=ADEX;
while(1) {
    if (conversion_finished==FALSE) {
        /* 在 A/D 转换期间可执行用户的其他程序代码 */
    }
    else {
        result_ADC[ADC_Channel]=5*((256*ADCH+(ADCON&0xc0))>>6);
        /* 存储 A/D 转换结果 */
        if (ADC_Channel!=7) {
            /* 准备对下一个通道进行转换 */
            ADCON=++ADC_Channel;
            ADCON=ADCON|ADCS;
        }
        else {
            /* ADC0...ADC7 全部转换完毕, 结果送往 UART */
            write_UART(&result_ADC, conversion++);
            if (conversion==10000) conversion=0;
            ADC_Channel=0;
            ADCON=0;          /* 准备下一次扫描 */
            ADCON=ADEX;
        }
        conversion_finished=FALSE;
    }
}

void ADC(void) interrupt 10 using 1 {
    ADCON=ADCON&ADCI;       /* 清除 ADCI 标志 */
    conversion_finished=TRUE;
}

```

)

output.c 程序代码如下:

```

/*****
* 功能描述: 将 A/D 转换结果送往 80C552 的 UART
*****/
#include <reg552.h>
char code string_0[] = "Conversion #";
char code string_1[] = ": (Ref is 5.12V)";
char code string_2[] = "ADC_Channel #";
char code string_3[] = "mV";
char code string_4[] = ": ";
char code new_line[] = "\r\n";

/*****
* 函数原型      : void send_byte(char src_byte);
* 功    能      : 将一个字节的数据送往 UART 并等待传送结束。
*****/
void send_byte(char src_byte) {
    S0BUF=src_byte;      /* 装入传送字节 */
    while(TI==0);        /* 等待传送结束 */
    TI=0;                 /* 清除传送标志 */
}

/*****
* 函数原型      : void decode(char src_nibble);
* 功    能      : 将数据转换成 ASCII 码并传送。
*****/
void decode(char src_nibble) {
    if (src_nibble<0x0a)
        send_byte(src_nibble+0x30);
    else
        send_byte(src_nibble+0x41-0x0a);
}

/*****
* 函数原型      : void send_bin_byte(char src_byte)
* 功    能      : 分离出数据中的一个字节, 将其转换成 ASCII 码并传送。
*****/
void send_bin_byte(char src_byte) {
    decode((src_byte>>4) & 0x0f);
    decode(src_byte & 0x0f);
}

```

```
}

```

```

/*****

```

```

* 函数原型      : void send_dec_int(unsigned int src_wrd)

```

```

* 功    能      : 将二进制整数转换成十进制整数并传送。

```

```

*****/

```

```

void send_dec_int(unsigned int src_wrd) {
    unsigned char a,b,c,d,e;
    a=src_wrd/1000;          /* a=千位 */
    b=((src_wrd%1000)/100);  /* b=百位 */
    c=((src_wrd%100)/10);   /* c=十位 */
    d=src_wrd%10;           /* d=个位 */
    e=16*c+d;
    if (a==0) {
        send_byte(0x20);
        if (b==0) {
            send_byte(0x20);
            if (c==0) {
                send_byte(0x20);
                decode(d);
            }
            else
                send_bin_byte(e);
        }
        else {
            decode(b);
            send_bin_byte(e);
        }
    }
    else {
        send_bin_byte((16*a)+b);
        send_bin_byte(e);
    }
}

```

```

/*****

```

```

* 函数原型      : void send_string(char code *str_ptr)

```

```

* 功    能      : 取出 ROM 中的字符串并传送。

```

```

*****/

```

```

void send_string(char code * str_ptr) {
    while (*str_ptr !=0)
        send_byte(*(str_ptr++));    /* 发送字节 */
}

```

```

}

/*****
* 函数原型      : void write_UART(unsigned int *ADC_result,
*                  unsigned int conversion_cnt);
* 功    能      : 将结果转换成正确的格式并送往 UART。
*****/
void write_UART(unsigned int *result_ptr, unsigned int conversion_cnt) {
    unsigned char cnt;

    send_string(new_line);
    send_string(new_line);
    send_string(string_0);          /* 发送信息号 */
    send_dec_int(conversion_cnt);
    send_string(string_1);
    for (cnt=0; cnt<8; cnt++) {
        send_string(new_line);
        send_string(string_2);      /* 发送通道号 */
        decode(cnt);
        send_string(string_4);
        send_dec_int(*(result_ptr++)); /* 发送结果 */
        send_string(string_3);
    }
}

```

8.11 87C752 单片机在气流量测量中的应用

8.11.1 87C752 单片机简介

87C752 是 PHILIPS 公司生产的一种小体积、低价格的 80C51 系列单片机。与其他 8051 单片机相比,其最大差别是它的内部总线不对外开放,即没有用于扩展外部程序存储器和数据存储器的并行扩展总线端口。指令系统与 8051 兼容(但没有 MOVX、LCALL 和 LJMP 指令)。图 8.91 所示为 87C752 的结构框图。87C752 片内包含有 2K 字节的 EPROM 程序存储器,64 字节的 RAM,2 个 8 位 I/O 口 P1 和 P3,1 个 5 位 I/O 口 P0,共有 21 根 I/O 口线,1 个常数可自动重装的 16 位定时器/计数器,1 个固定速率的定时器,7 个中断源,1 个中断优先级,5 路 8 位 A/D 转换器,1 路脉宽调制输出以及位方式的 I²C 总线接口。

图 8.92 所示为 87C752 的引脚排列图。

各个引脚的功能如下。

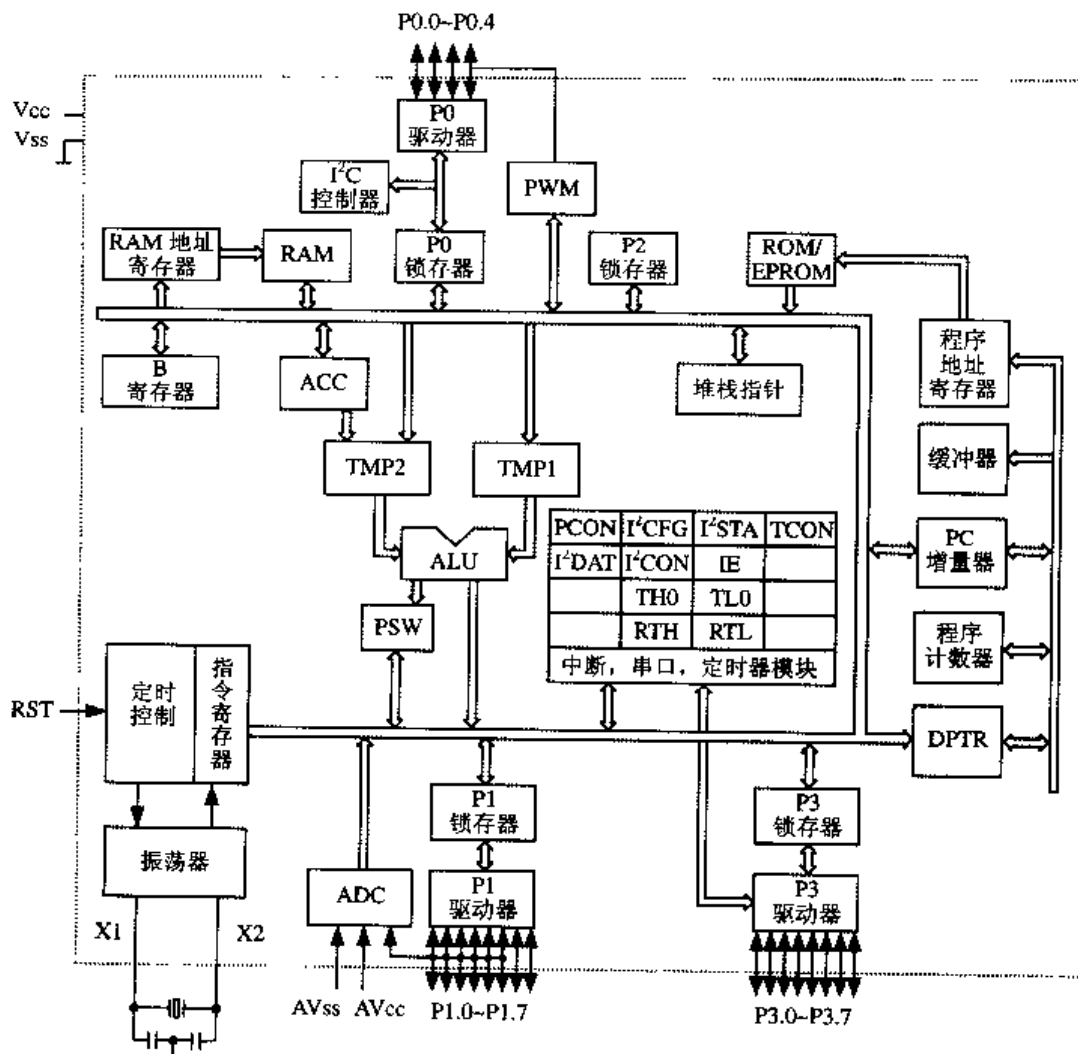


图 8.91 87C752 结构框图

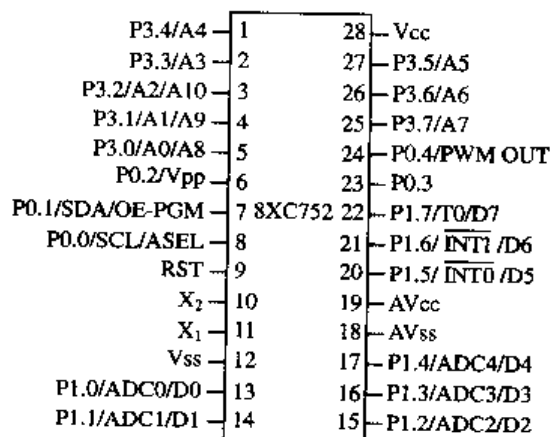


图 8.92 87C752 的引脚排列图

V_{CC}: +5V 电源。

V_{SS}: 地。

AV _{CC} :	模拟参考电源。
AV _{SS} :	模拟地。
X ₁ ~X ₂ :	晶振输入端。
RST:	复位输入端。
P0.0:	TTL 输入 / 输出 (开漏), I ² C 时钟 (SCL)。
P0.1:	TTL 输入 / 输出 (开漏), I ² C 数据 (SDA)。
P0.2:	TTL 输入 / 输出 (开漏)。
P0.3:	TTL 输入 / 输出 (内部上拉)。
P0.4:	TTL 输入 / 输出 (内部上拉), PWM 输出。
P1.0~P1.4	TTL 输入 / 输出 (内部上拉), ADC0~ADC4。
P1.5:	TTL 输入 / 输出 (内部上拉), 外部中断 0INT0 输入。
P1.6:	TTL 输入 / 输出 (内部上拉), 外部中断 1INT1 输入。
P1.7:	TTL 输入 / 输出 (内部上拉), 定时器 T0 输入。

注意: P1.0~P1.4 的功能只能一起改变, 即它们同时用作 TTL I/O 口线或同时用作 A/D 转换器的输入。当用作 A/D 转换器的输入时, 同时可用作 TTL 输入。

P3.0~P3.7: TTL 输入/输出 (内部上拉)。

87C752 可以对 3 个存储器空间进行操作, 第一个是程序存储器空间, 该空间用来存放指令代码和常数 (如表格常数等), 程序存储器空间的长度为 2K 字节。第二个是内部数据存储器空间, 该空间逻辑上为 128 个字节, 但在物理上只有 64 个字节 (地址为 00H~3FH)。第三个是特殊功能寄存器空间, 地址为 80H~FFH, 但实际上只能使用其中的某些地址单元。由此可见, 87C752 的存储器空间与 8051 是一致的, 不同之处在于它们的存储容量不同, 而且 87C752 不能直接访问外部程序存储器和外部数据存储器, 因此 8051 指令系统中的 MOVX、LCALL 和 LJMP 指令对于 87C752 是无效的。

87C752 内部集成了一个脉冲宽度调制器 PWM, 它由一个 8 位预分频器, 一个 8 位计数器和一个 8 位比较器组成。87C752 中 P0.4 口线的第二功能为脉宽调制输出 PWM, 能输出频率和占空比可编程的脉冲信号。脉冲频率取决于 8 位预分频器, 由它产生计数信号, 分频系数保存在 PWMP 寄存器中。

8 位计数器的计数范围从 0~254, 在每个机器周期都将 8 位计数器的计数值与比较寄存器的内容相比较, 当计数值与比较寄存器的内容相等时, P0.4 (PWM) 输出高电平, 在计数器计数到 0 时 P0.4 (PWM) 输出低电平。脉冲的占空比由比较寄存器的内容所决定, 其范围为 0~1, 可编程的分辨率为 1/255。装入比较寄存器的值为 0 时, P0.4 (PWM) 输出连续高电平, 装入比较寄存器的值为 0FFH 时, P0.4 (PWM) 输出连续低电平。通过置位特殊功能寄存器 PWENA (PWM 输出使能寄存器) 中的 PWE 位 (PWENA.0) 来允许 PWM 输出, 允许 PWM 输出时, P0.4 口线的输出由强拉高电路所驱动。禁止 PWM 输出时, P0.4 表现为一般的 I/O 引脚特性。在禁止 PWM 输出时计数器仍保持其计数状态。复位时 PWM 被禁止, 在掉电和空闲方式下 PWM 输出高电平而计数器停止计数。

PWM 输出的重复频率由下式给定:

$$f_{\text{PWM}} = f_{\text{OSC}} / [2 \times (1 + \text{PWMP}) \times 255]$$

对于 12MHz 的晶振, fPWM 在 92Hz 至 23.5kHz 的范围之内。PWM 输出的占空比(低/高)为:

比较寄存器 PWM 的值/(255-比较寄存器 PWM 的值)

当比较寄存器 PWM 的值为 255 时将导致 PWM 输出连续低电平。

若允许 PWM 中断, 则当 PWM 计数器溢出时将产生一个中断。为了使 P0.4 (PWM) 口线作为一般的 I/O 口线使用, 可以将特殊功能寄存器 PWENA 中的 PWE 位 (PWENA.0) 置为“0”, 这时可通过允许 PWM 中断使 PWM 计数器仍可作为一个内部定时器使用。

87C752 中 PWM 的中断入口地址为 0033H。

87C752 内部集成有一个 5 路输入的 8 位 A/D 转换器, 完成一次 A/D 转换需要 40 个机器周期(采用 12MHz 晶振时为 40μs)。A/D 转换器由特殊功能寄存器 ADCON 控制, ADCON 只能通过字节寻址, 地址为 0A0H, 格式如下:

D7	D6	D5	D4	D3	D2	D1	D0
—	—	ENADC	ADCI	ADCS	AADR2	AADR1	AADR0

其中各位的意义如下。

D7 和 D6 两位没有定义。

ENADC: A/D 转换允许禁止位, ENADC=1 允许, ENADC=0 禁止。禁止时 P1.0~P1.4 作为一般 I/O 口线使用。复位后 ENADC=0。

ADCI: A/D 中断标志, 1 次 A/D 转换完成时置位 ADCI, 若中断允许寄存器中的 IE.6=1, 则当 ADCI=1 时产生中断请求。读取 A/D 转换结果时清零 ADCI 位, ADCI 位只能读出, 不能写入。

ADCS: ADC 启动和状态标志位, 置位该标志将启动 A/D 转换, 一旦置位后, 将一直保持为 1 直至 A/D 转换完成为止, A/D 转换完成时置位 ADCI, 同时清零 ADCS, ADCS 不能用软件清零。

ADCS 与 ADCI 有 4 种状态组合, 其功能如下:

ADCS	ADCI	功能
0	0	ADC 空闲, 可以启动 A/D 转换。
0	1	ADC 忙, 禁止再启动 A/D 转换。
1	0	ADC 转换完成, 禁止重新启动 A/D 转换。
1	1	无效。

AADR2~AADR0 是模拟通路选择位, 其关系如下:

AADR2	AADR1	AADR0	模拟通道输入
0	0	0	P1.0 (ADC0)
0	0	1	P1.1 (ADC1)
0	1	0	P1.2 (ADC2)
0	1	1	P1.3 (ADC3)
1	0	0	P1.4 (ADC4)

仅当 ADCI 和 ADCS 这两位都为 0 时, 地址选择位才能被改变。

8 位 A/D 转换完成与否由 ADCI 标志位给出, ADCI=1 表示一次 A/D 转换完成, 转换结果存储在特殊功能寄存器 ADAT 中。一个正在进行的 A/D 转换过程不受一个新的 A/D 启动的影响。当 ADCI=1 时, 已完成的 A/D 转换结果不受影响。当 ADCI=1 或 ADCS=1 时将阻止启动 A/D 转换。在进入空闲或掉电模式时, 正在进行中的 A/D 转换将异常结束。进入空闲方式时 A/D 转换结果不受影响 (ADCI=1)。图 8.93 所示为 87C752 中 A/D 转换器的输入等效电路。通过清零 ADCON 中的 ENADC 位禁止 A/D 转换时, 模拟量输入引脚 ADC0~ADC4 可作为数字量的输入输出线。允许 A/D 转换时, 由 AADR2~AADR0 位选中的模拟量输入通道不可以作为数字量的输入线, 未被中的通道可作为数字量输入, 但不能作为输出。

87C752 中 A/D 转换器的中断入口地址为 002BH。

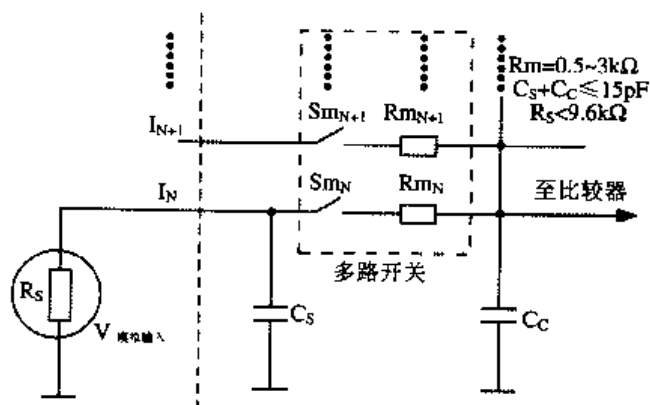


图 8.93 87C752 中 A/D 输入等效电路

87C752 中定时器/计数器 T0 的操作与 8051 中 T0、T1 的方式 2 相同, 但计数长度为 16 位, 即它是一个计数初值可重新装入的 16 位定时器/计数器。其功能由特殊功能寄存器 TCON 控制。TCON 可以位寻址, 字节地址为 88H, 其格式如下:

D7	D6	D5	D4	D3	D2	D1	D0
GATE	C/T	TF	TR	IE0	IT0	IE1	IT1

其中各位的意义如下。

GATE: =1, 表示仅当 INT0 引脚为高电平且 TR=1 时才允许 T0 计数。

=0, 表示只要 TR=1 就允许 T0 计数。

C/T: =1, 以 T0 引脚上的输入脉冲作为计数信号。

=0, 以内部时钟脉冲作为计数信号。

TF: T0 计数溢出标志, 溢出时置“1”, 当 CPU 响应中断而转向 T0 溢出中断处理入口时清零, 也可以用软件清零。

TR: 该位为 1 时允许 T0 计数, 为 0 时禁止 T0 计数。

IE0: 外部中断 INT0 的中断请求标志, 当检测到 INT0 引脚上的电平发生负跳变时, 该位被置“1”。

IT0: =1, 表示 INT0 由负跳变触发。

=0, 表示 INT0 由低电平触发。

IE1 和 IT1 与 IE0 和 IT0 的功能相似, 只是它们控制的是外部中断 INT1。

由此可见这些标志位的功能与 8051 中相应标志位的功能是一致的, 只是将定时器的工作方式和状态控制标志结合到一个寄存器之中。

87C752 中还有一个通讯监视定时器 T1, 它专门用于 I²C 总线通讯。当不使用 I²C 总线时 T1 可以作为固定溢出率的定时器用, 这时 T1 的溢出周期大约为 1024 个机器周期。一个机器周期为 12 个时钟周期, 因此如果采用 12MHz 的晶振, 则 T1 大约每隔 1ms 发生一次溢出。将 I²C 寄存器 I2CFG 中的 T1RUN 位 (I2CFG.4) 置“1”, 其余各位置“0”即为不使用 I²C 总线而将定时器 T1 用做固定溢出率的定时器。

87C752 中定时器 T1 的溢出中断入口地址为 001BH。另外, 87C752 中 I²C 总线采用所谓“位方式”进行数据传送, 它与 80C552 中“字节方式”数据传送的 I²C 总线有所不同。由于在下面的应用中不使用 87C752 的 I²C 总线, 因此这里不予详细介绍, 需要时读者可以查阅有关的资料。87C752 中只有一个中断允许寄存器 IE, 没有中断优先级寄存器, 即它只有一个中断优先级。IE 的字节地址为 A8H, 其格式如下:

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

其中各位的意义如下。

EA: 总中断允许位, EA=1 开中断, EA=0 关中断。

EAD: A/D 转换完成中断允许位, EAD=1 允许, EAD=0 禁止。

ET1: 定时器 T1 溢出中断允许位, ET1=1 允许, ET1=0 禁止。

ES: I²C 总线中断允许位, ES=1 允许, ES=0 禁止。

EPWM: PWM 计数器溢出中断允许位, EPWM=1 允许, EPWM=0 禁止。

EX1: 外中断 INT1 中断允许位, EX1=1 允许, EX1=0 禁止。

ET0: 定时器 T0 溢出中断允许位, ET0=1 允许, ET0=0 禁止。

EX0: 外中断 INT0 中断允许位, EX0=1 允许, EX0=0 禁止。

87C752 中的中断源优先级和中断入口地址如下:

中断源	功 能	优先级	入口地址
INT0	外部中断 0	1 (最高)	0003H
TF	定时器 0 溢出	2	000BH
INT1	外部中断 1	3	0013H
PWM	PWM 计数器溢出	4	0033H
T1	T1 计数器溢出	5	001BH
SIO	I ² C 串口中断	6	0023H
ADC	A/D 转换完成	7 (最低)	002BH

8.11.2 气流量测量仪表的硬件设计

气流量测量在工业和科学上都有重要意义。气流量是指单位时间内传送的气体体积, 单位为“立方英尺/分”(cubic volume per minute), 简称为 CPM。测量仪表不断地将所测

得的气体流量值与预编程设定值进行比较,一旦超过设定值,即发出报警信号,以便进行紧急处理。气流量的设定值可根据实际需要来决定。为了准确计算出气流量,需要同时测量三个气体参数:速度、压力和温度。图 8.94 所示为气流量测量仪表的前面板示意图。

一般情况下仪表通过三个 7 段 LED 连续显示气体的流量值,同时单个 CPM LED 也被点亮以表示测量值的单位。

连续按下 TEMP 键时,7 段 LED 显示温度值(单位为 $^{\circ}\text{C}$),同时点亮单个 TEMP LED,松开 TEMP 键后,7 段 LED 转换到显示压力值。

连续按下 PSI 键时,7 段 LED 显示大气压值(单位为“磅/平方英寸”),同时点亮单个 PSI LED,松开 PSI 键后,7 段 LED 转换到显示气体流量值。

按下 SETPOINT 键时,7 段 LED 显示预编程的气流量设定值(单位为 CFM),同时点亮单个 SETPOINT LED,松开 SETPOINT 键后,7 段 LED 转换到显示气体流量值。

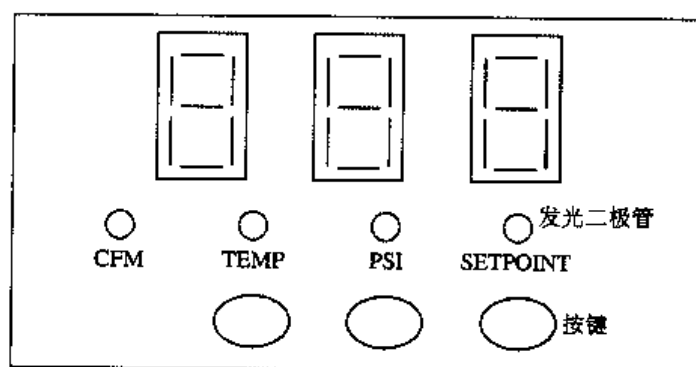


图 8.94 气流量测量仪表的前面板示意图

图 8.95 所示为气流量测量仪表的硬件电路图。三个标识为 X0.1、X1 和 X10 的共阴极 7 段 LED 分别显示十分之一位、个位和十位数值。每个 LED 都有 8 位数据输入线(7 位段码,1 位小数点)和 1 位位选输入线。87C752 的 P3 口通过电流驱动器 U2 和限流电阻 RP1 驱动 7 段 LED 的数据和 4 个单独 LED 的输入线。由于 3 个 7 段 LED 和 4 个单独 LED 共用相同的数据输入线,因此采用 4 个晶体管 Q1~Q4 作为各个 LED 的位选端, Q1~Q4 分别连到 P0 口的 4 根口线上,这样可以减少所需的 I/O 口线,同时可加快显示刷新速度。

仪表采用三个按键(SW1~SW3)来选择显示温度、压力或设定值,无键按下时显示气流量测量值,共有 4 种按键状态。利用 P1 口的 P1.3 和 P1.4 作为输入口线,对按键状态编码如下:

按键状态	P1.3	P1.4
无键按下	1	1
SW1 (TEMP 键) 按下	0	1
SW2 (PSI 键) 按下	1	0
SW3 (SETPOINT 键) 按下	0	0

设定值控制通过改变分压电阻的阻值以改变 87C752 的一路 A/D 输入电压的方法实现,分压电阻的阻值大小根据实际需要确定。当气流量测量值超过设定值时,通过 87C752 的 P1.6 口线的输出使继电器 K1 导通,实现超限报警。

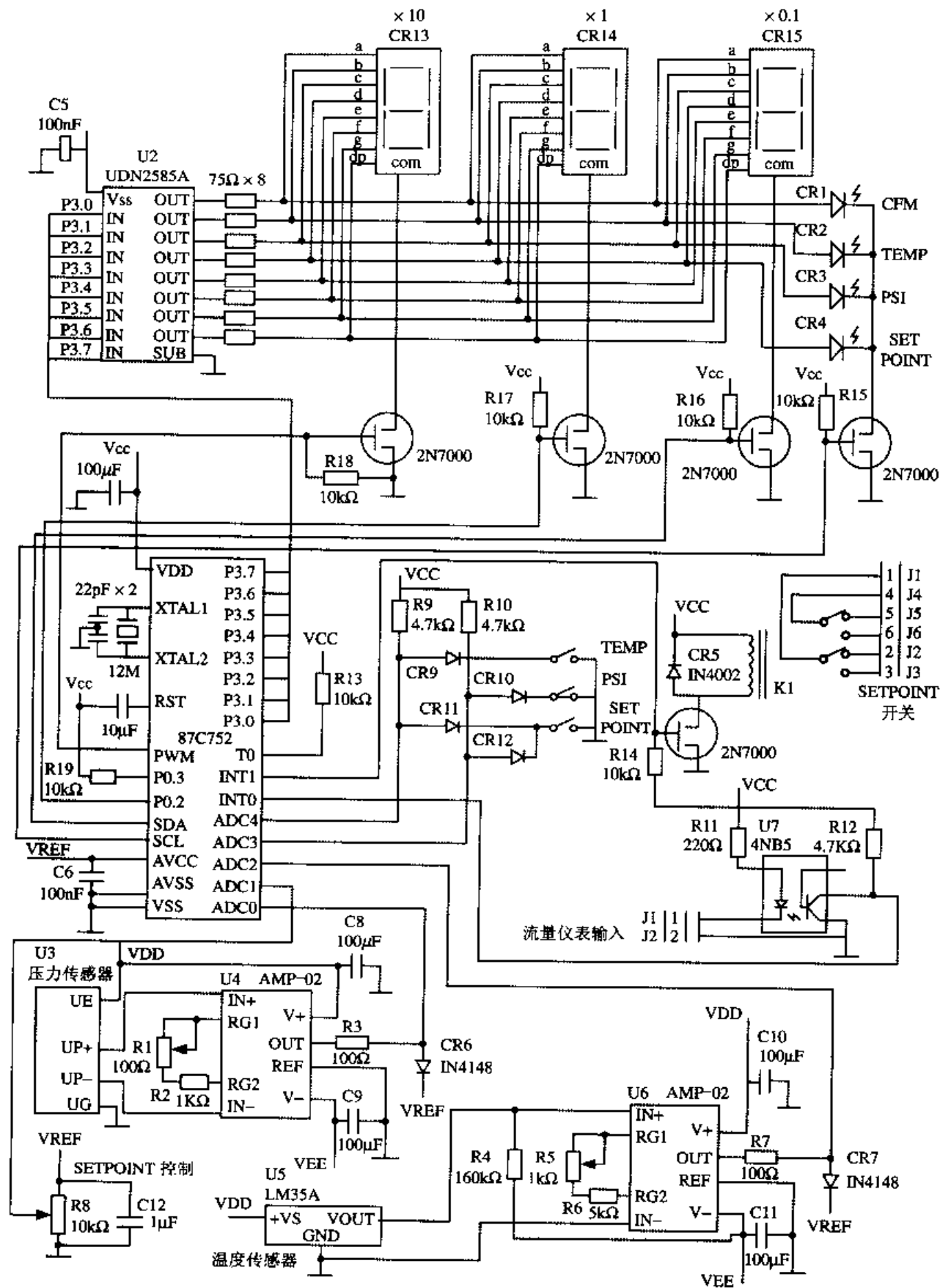


图 8.95 气流量测量仪表的硬件电路

将汽轮机的转速表通过光电耦合器 U7 连到 87C752 的 P1.5 输入口线上实现气体速度的测量。转速表发出的是一系列占空比为 10% 的负脉冲。

在一定的流体速度和环境温度之下, 环境压力的变化将导致气体流量的变化。通过压力传感器对于不同压力下输出不同的电压值来实现环境压力的测量。运算放大器 U4 用来将压力传感器的输出规范到 87C752 中 A/D 转换器的输入范围之内 (模拟电压 AV_{cc} 与模拟地 AV_{ss} 之间), 压力传感器的输出信号经 U4 规范之后连到 87C752 的 A/D 转换输入端 P1.0 上。校正压力传感器的输出时, 按下 PSI 键并调节增益电阻 R1, 直到 7 段 LED 显示出当地大气压力值。

环境温度的测量通过温度传感器输出正比于不同温度的电压值来实现, 运算放大器 U5 用来将温度传感器的输出规范到 87C752 中 A/D 转换器的输入范围之内 (模拟电压 AV_{cc} 与模拟地 AV_{ss} 之间), 经 U5 规范之后的温度传感器输出信号连到 87C752 的 A/D 转换输入端 P1.1 上。校正温度传感器的输出时, 按下 TEMP 键并调节增益电阻 R5, 直到 7 段 LED 的显示值与 U5 的输出值相等。

下面是 87C752 的各个 I/O 口线的安排:

P0.0 (SCL):	TTL 输出, 单个 LED 显示使能端。
P0.1 (SDA):	TTL 输出, 十分之一位 7 段 LED 显示使能端。
P0.2:	TTL 输出, 个位 7 段 LED 显示使能端。
P0.3:	上拉。
P0.4 (PWM):	TTL 输出, 十位 7 段 LED 显示使能端。
P1.0 (ADC0):	A/D 输入, 接压力传感器。
P1.1 (ADC1):	A/D 输入, 接设定值控制。
P1.2 (ADC2):	A/D 输入, 接温度传感器。
P1.3 (ADC3):	TTL 输入, 按键输入。
P1.4 (ADC4):	TTL 输入, 按键输入。
P1.5 (INT0):	INT0 中断输入, 接气轮机转速计。
P1.6 (INT1):	TTL 输出, 接控制继电器。
P1.7 (T0):	T0 输入端, 接上拉电阻。
P3.0:	TTL 输出, 7 段 LED 的 a 段, 单个 CFM LED。
P3.1:	TTL 输出, 7 段 LED 的 b 段, 单个 TEMP LED。
P3.2:	TTL 输出, 7 段 LED 的 c 段, 单个 PSI LED。
P3.3:	TTL 输出, 7 段 LED 的 d 段, 单个 SETPOINT LED。
P3.4:	TTL 输出, 7 段 LED 的 e 段。
P3.5:	TTL 输出, 7 段 LED 的 f 段。
P3.6:	TTL 输出, 7 段 LED 的 g 段。
P3.7:	TTL 输出, 7 段 LED 的 dp 段。

8.11.3 气流量测量仪表的软件设计

气流量测量仪表的软件程序采用 Cx51 编写。整个程序由 4 个中断函数 (前台程序)

和一个主函数（后台程序）所组成。87C752 单片机复位后进入主程序并循环执行之，当产生中断时进入相应的中断函数程序执行。主程序与中断函数程序之间通过参数传递完成相互之间的通信联络。4 个中断函数的功能如下。

(1) multiplex()函数

测量仪表不使用 87C752 中 I²C 总线，因此将内部定时器 T1 用作具有固定溢出率的监视定时器。单片机的晶振为 12MHz 时，T1 将以大约 1000Hz 的频率产生溢出中断。multiplex() 是 T1 溢出中断函数，利用 T1 所产生的固定速率中断，实现仪表的显示刷新。在该函数中建立了一个被点亮 LED 的显示标志，每次进入 T1 溢出中断时，根据该显示标志熄灭前一个被点亮的 LED，将显示数据写入当前 LED 并将其点亮，同时将显示标志指向下一个 LED。由于 T1 溢出中断的频率是固定的，从而可实现各个 LED 显示数据的轮流刷新。图 8.96 为 multiplex()函数的流程。

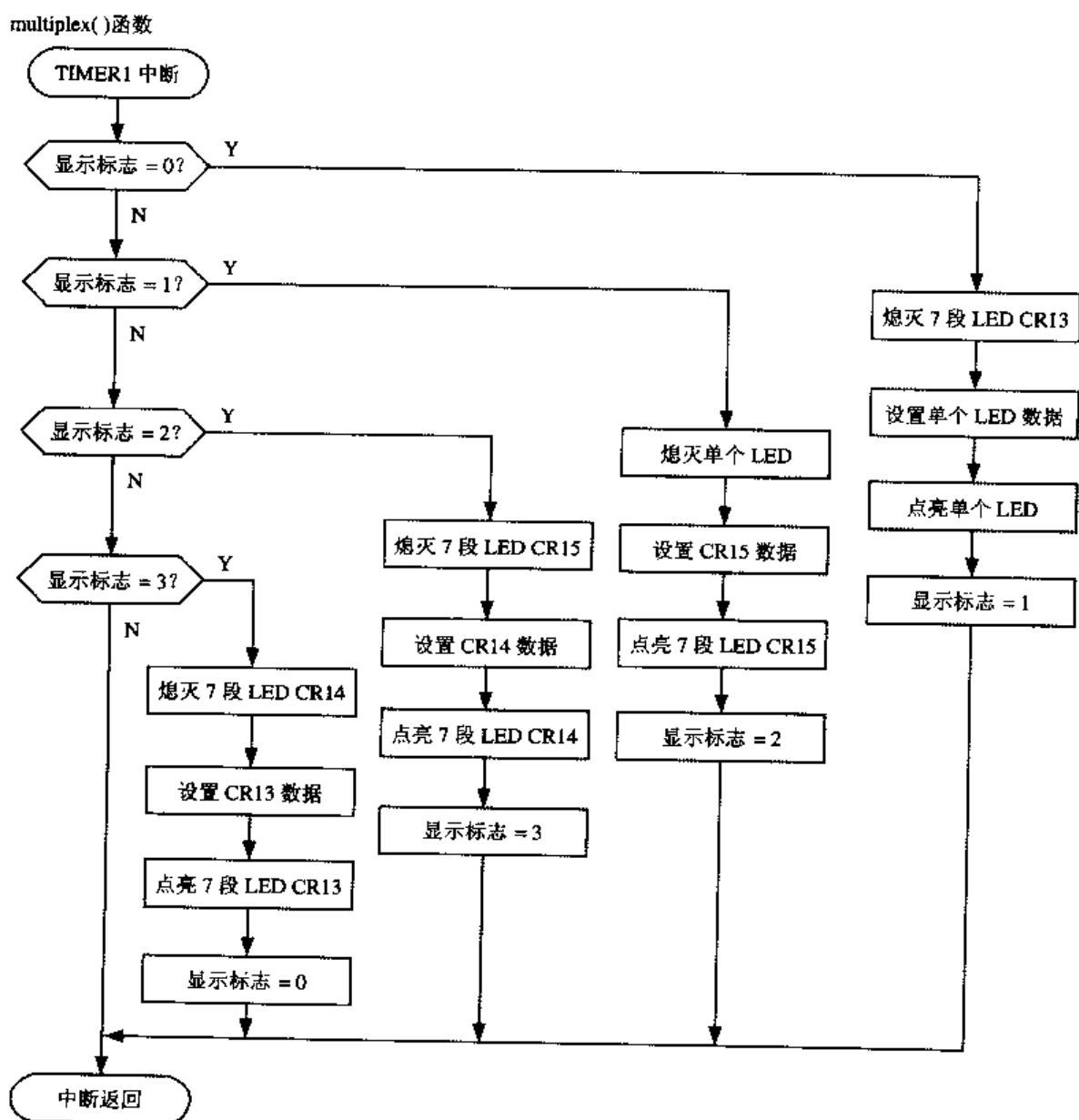


图 8.96 multiplex()函数的流程

(2) read_switch()函数

87C752 内部的 PWM 被用来产生 92Hz 的周期中断。read_switch()是 PWM 中断函数。在该函数中对 PWM 的中断次数进行计数, PWM 中断满 32 次时置位“更新变量”。每当主函数检测到这个“更新变量”被置位时,即对显示数据进行更新,同时将“更新变量”复位为 0。这样大约每隔 33ms 的时间进行一次显示数据更新,可以消除显示闪烁和按键抖动。图 8.97 为 read_switch()函数的流程。

(3) calc_cfm()函数

这是 87C752 外部中断 INT0 的中断函数,它与内部定时器 T0 溢出中断函数相配合,完成气轮机气体速度的测量。每次进入 INT0 中断函数时,读取定时器 T0 的值来确定两次 INT0 之间的间隔时间。将读取的 T0 值作为 24 位微秒计数器的低 16 位,同时将高 8 位计数值清零以便在 T0 溢出中断函数中进行更新。图 8.98 为 cal_cfm()函数的流程。

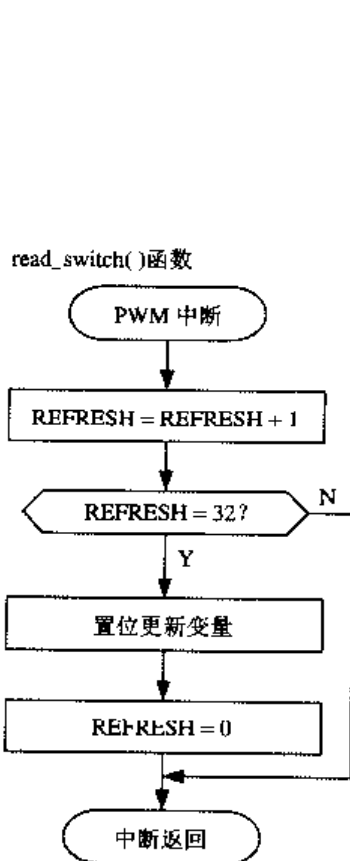


图 8.97 read_switch()函数的流程

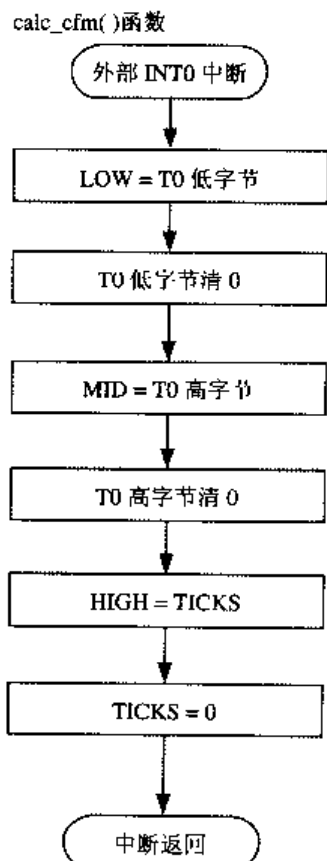


图 8.98 cal_cfm()函数的流程

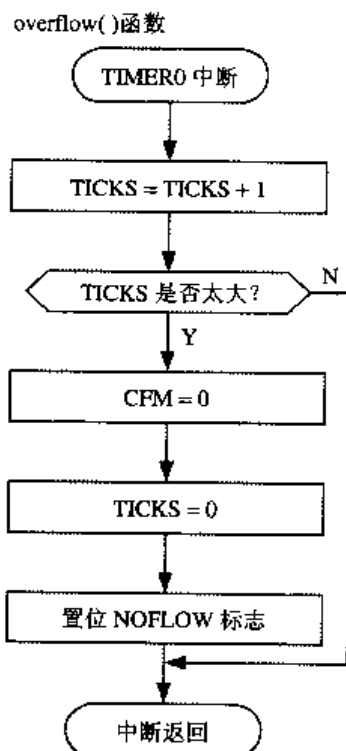


图 8.99 overflow()函数的流程

(4) overflow()函数

这是定时器 T0 溢出中断函数。每当 T0 产生溢出时,将 24 位微秒计数器的高 8 位加 1。如果这个高 8 位数值太大(转速表停转)则置位“NO_FLOW”变量标志,主函数一旦检测到该标志即在 7 段 LED 上显示出错信息“EEE”。图 8.99 为 overflow()函数的流程。

图 8.100 是主函数 main()的流程。上电复位后即进入 main()函数循环执行,在 main()函数中完成三个主要任务。第一是根据测量得到的温度和压力值进行气流量的补偿计算,

第二是将计算得到的气流量值与设定值进行比较, 并根据比较结果值决定输出继电器的开或关。第三是检查“更新标志”的状态(其状态在 read_switch()中断函数中改变), 并根据其状态决定是否进行显示数据的更新。

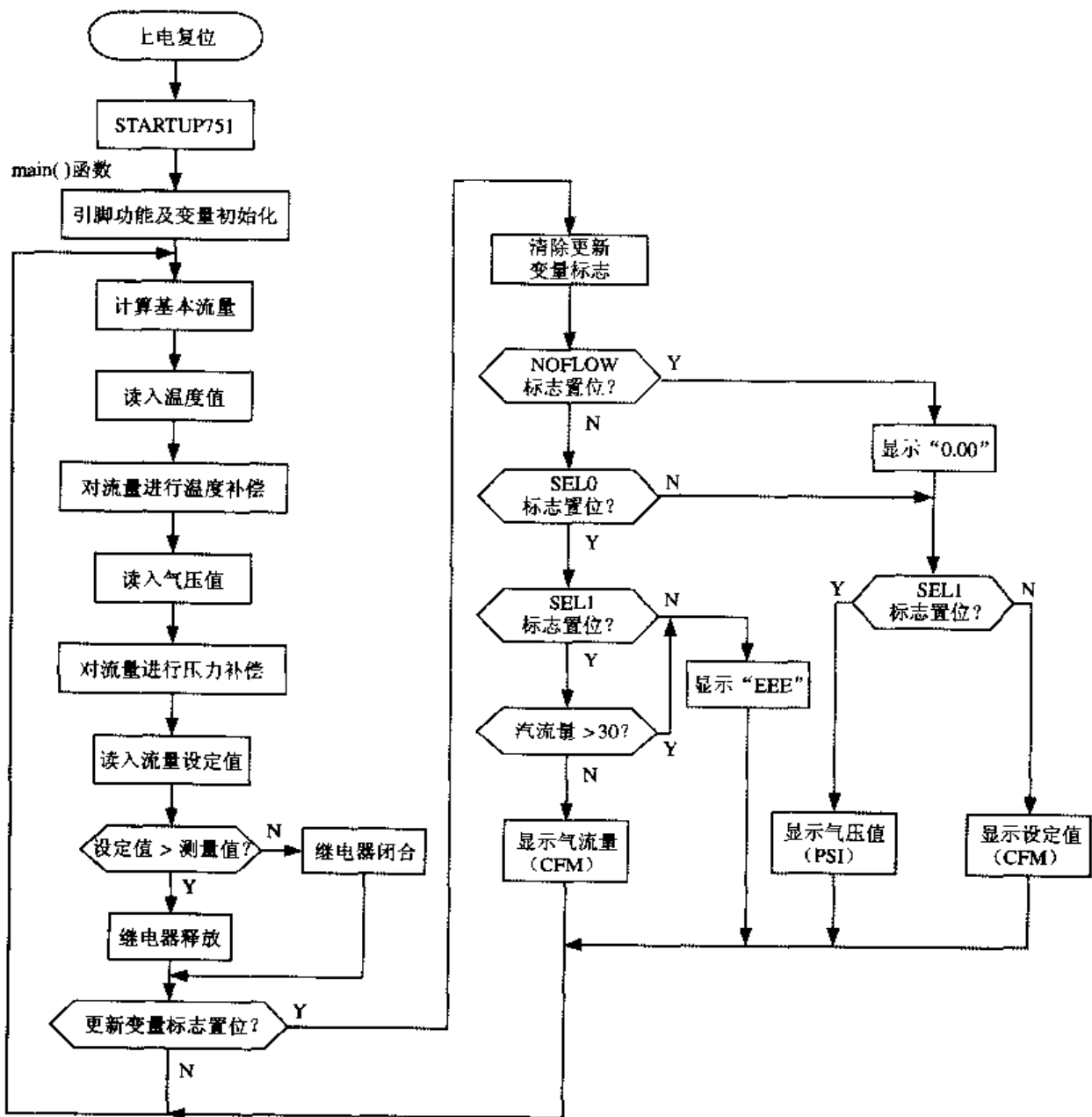


图 8.100 主函数 main()的流程

程序代码如下:

```
#include <reg752.h>
#include <stdio.h>
```

```
/* ***** 定义符号常数 ***** */
#define ZERO_K 2730 /* 0 摄氏度, 1/10 kelvin */
#define ONE_TENTH_CFM 4444444L /* 1/10 CFM 微秒 */
```

```

#define CORRECTION      0x11    /* 时钟频率为 12MHz 时计算 CFM 校正值 */
#define STD_TEMP        2980    /* 25 摄氏度, 1/10 kelvin */
#define STD_ATM         147     /* 一个大气压 1/10 PSI */
#define LOWEST_CFM      0x40    /* 转速表的最大周期值 0x400000 */
#define START_ADC0      0x28    /* 各个通道 A/D 转换启动命令 */
#define START_ADC1      0x29
#define START_ADC2      0x2a
#define START_ADC3      0x2b
#define START_ADC4      0x2c
#define ADCI            0x10    /* A/D 转换状态标志 */
#define ADCS            0x08
#define FREERUN_I       0x10
#define SEG_A           0x01    /* 七段 LED 中的 'a' 段 */
#define CFM             0x01    /* CFM LED */
#define SEG_B           0x02    /* 七段 LED 中的 'b' 段 */
#define DEGREES         0x02    /* DEGREES LED */
#define SEG_C           0x04    /* 七段 LED 中的 'c' 段 */
#define PSI            0x04    /* PSI LED */
#define SEG_D           0x08    /* 七段 LED 中的 'd' 段 */
#define SETPOINT        0x08    /* SETPOINT LED */
#define SEG_E           0x10    /* 七段 LED 中的 'e' 段 */
#define SEG_F           0x20    /* 七段 LED 中的 'f' 段 */
#define SEG_G           0x40    /* 七段 LED 中的 'g' 段 */
#define SEG_DP          0x80    /* 七段 LED 中的 'dp' 段 */

typedef unsigned char byte;    /* 8 位无符号字节型数据 */
typedef unsigned int word;    /* 16 位无符号字型数据 */
typedef unsigned long I_word; /* 32 位无符号长字型数据 */

#define TRUE            1
#define FALSE           0

```

/****** 定义七段 LED 的显示段码表 ******/

code byte segments [] =

```

{
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F      , /* 0 */
        SEG_B | SEG_C                                  , /* 1 */
    SEG_A | SEG_B |          SEG_D | SEG_E              , /* 2 */
    SEG_A | SEG_B | SEG_C | SEG_D |          SEG_G      , /* 3 */
        SEG_B | SEG_C |          SEG_F | SEG_G          , /* 4 */
    SEG_A |          SEG_C | SEG_D | SEG_F | SEG_G      , /* 5 */
    SEG_A |          SEG_C | SEG_D | SEG_E | SEG_F | SEG_G , /* 6 */
    SEG_A | SEG_B | SEG_C                                , /* 7 */

```

```

SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G , /* 8 */
SEG_A | SEG_B | SEG_C | SEG_D |          SEG_F | SEG_G , /* 9 */
SEG_A |          SEG_D | SEG_E | SEG_F | SEG_G , /* E */
};

/***** 定义 87C752 中用于 I/O 线的特殊功能位 *****/
sfr  PWMP      = 0x8F;      /* 定义 87C752 的 PWM 预分频寄存器 */

sbit  RELAY      = 0x96;      /* 为高时闭合 SETPOIN 继电器 */
sbit  STROBE_0    = 0x80;      /* 为高时允许点亮各个状态 LED */
sbit  STROBE_1    = 0x81;      /* 为高时允许点亮 cr15 (十分之一位) */
sbit  STROBE_2    = 0x82;      /* 为高时允许点亮 cr14 (个位) */
sbit  NO_FLOW     = 0x83;      /* 未检测到气流量时的标志位 */
sbit  STROBE_3    = 0x84;      /* 为高时允许点亮 cr13 (十位) */
sbit  SEL_0       = 0x93;      /* 为低时用于按键输入, 选择显示模式 */
sbit  SEL_1       = 0x94;
sbit  INTR        = 0x95;
sbit  UPDATE      = 0x97;      /* 显示更新标志 */

/***** 定义内存变量 *****/
data word  cfm;      /* 十分之一气流量 CFM */
data word  setpoint;  /* 十分之一气流量 CFM 的继电器设定值 */
data word  degree_c;  /* 十分之一摄氏度 */
data I_word corr;     /* 计算值 */
data word  psi;       /* 十分之一 PSI */
data byte  display0;  /* 显示更新值 */
data byte  display1;  /* display0=status LEDs, display1=cr15, */
data byte  display2;  /* display2=cr14, display3=cr13, */
data byte  display3;
data byte  disp_pntr; /* 下一个要点亮的 LED 指针 */
data byte  refresh;   /* 显示更新计数 */
data byte  high;      /* 24 位计数器的第 16 - 23 位 */
data byte  middle;    /* 24 位计数器的第 8 - 15 位 */
data byte  low;       /* 24 位计数器的第 0 - 7 位 */
data byte  ticks;     /* 定时器溢出加一计数 */

/*****
* 函数原型: void multiplex() interrupt 3;
* 功 能: 利用自由运行的定时器 T1 对显示器进行动态更新, 更新速度
*       约为 1000Hz。
*****/
void multiplex() interrupt 3 {
    switch(disp_pntr) {

```

```

    case 0x00:
        STROBE_3 = FALSE;    /* 关闭 cr13 */
        P3 = 0xff;           /* 关闭 7 段 LED */
        P3 = display0;       /* 装入段码 */
        STROBE_0 = TRUE;     /* 点亮状态 LEDs */
        disp_pntr = 1;       /* 指向下一个显示器 */
        break;
    case 0x01:
        STROBE_0 = FALSE;    /* 关闭状态 LEDs */
        P3 = 0xff;           /* 关闭 7 段 LED */
        P3 = display1;       /* 装入十分之一位 LED 段码 */
        STROBE_1 = TRUE;     /* 点亮 cr15 */
        disp_pntr = 2;       /* 指向下一个显示器 */
        break;
    case 0x02:
        STROBE_1 = FALSE;    /* 关闭 cr15 */
        P3 = 0xff;           /* 关闭 7 段 LED */
        P3 = display2;       /* 装入个位 LED 段码 */
        STROBE_2 = TRUE;     /* 点亮 cr14 */
        disp_pntr = 3;       /* 指向下一个显示器 */
        break;
    case 0x03:
        STROBE_2 = FALSE;    /* 关闭 cr14 */
        P3 = 0xff;           /* 关闭 7 段 LED */
        P3 = display3;       /* 装入十位 LED 段码 */
        STROBE_3 = TRUE;     /* 点亮 cr13 */
        disp_pntr = 0;       /* 指向下一个显示器 */
    }
}

/*****
* 函数原型: void read_switch() interrupt 6;
* 功    能: 利用自由运行的 PWM 预分频器产生 92Hz 周期的中断。每中断
*           32 次将 UPDATE 标志置位, main() 函数将根据该标志位的状
*           态来采样按键并进行显示更新。
*****/
void read_switch() interrupt 6 {
    if (refresh++ == 32) {
        UPDATE = TRUE;
        refresh = 0;
    }
}

```

```

/*****
* 函数原型: void overflow() interrupt 1;
* 功    能: 每当定时器 T1 溢出(从 0xffff 到 0x0000)时将变量'ticks'的
*           值加 1, 该值表示 24 位气流量周期微秒计数器的高 8 位(16~
*           23 位)。如果'ticks'的值太大则将 NO_FLOW 标志置位, 从而
*           使 main()函数在 LED 上显示 00.0。
*****/
void overflow() interrupt 1 {
    if (++ticks > LOWEST_CFM) {
        cfm = 0;
        ticks = 0;
        NO_FLOW = TRUE;
    }
}

/*****
* 函数原型: void cal_cfm() interrupt 0;
* 功    能: 由转速计脉冲产生的外部中断(INT0)将定时器 T0 的当前计
*           数值送到 24 位微秒计数器的'low'(低 8 位)和'middle'(中
*           8 位), 同时复位定时器 T0。将前面的'ticks'值拷贝到
*           'high'中同时使'ticks'复位为 0。清除 NO_FLOW 标志, 从而
*           使 main()函数能在 LED 上显示计算出的 cfm 值。
*****/
void cal_cfm() interrupt 0 {
    low = TL;
    TL = 0;
    middle = TH;
    TH = 0;
    high = ticks;
    ticks = 0;
    NO_FLOW = FALSE;
}

/*****
* 函数原型: void main();
* 功    能: 在完成各个 I/O 引脚和变量的初始化之后进入一个连续循
*           环并完成以下任务:
*           -根据转速计脉冲、输入的温度值和压力值计算气流量;
*           -将气流量与设定输入值进行比较, 并控制继电器的开、闭;
*           -在 UPDATE 标志置位时采样按键状态并进行显示更新。
*****/
void main() {
    RELAY    = 0;    /* 初始化输出引脚 */

```

```
INTR      = 1;
UPDATE    = 1;
STROBE_0  = 0;
STROBE_1  = 0;
STROBE_2  = 0;
STROBE_3  = 0;
NO_FLOW   = 0;
I2CFG     = FREERUN_I;      /* 不使用 I2C, 定时器 T1 自由运行 */
RTL       = 0;              /* 定时器 T0 装入初值, 定时时间为 0x10000 微秒 */
RTH       = 0;
PWMP      = 255;            /* PWM 中断周期约为 93 Hz */
TR        = 1;              /* 启动定时器 T0 */
IT0       = 1;              /* 外部中断 INT0 为边沿触发 */
ticks     = 0;              /* 变量初始化 */
cfm       = 0;
low       = 0;
middle    = 0;
high      = 0;
degree_c  = 250;
psi       = 147;
corr      = 0;
refresh   = 0;
disp_pntr = 0;
IE        = 0xab;           /* 开中断 */

/* 进入连续循环 */
while(1) {

    /* 计算气流量 CFM */
    corr = high * 0x10000L;
    corr += (middle * 0x100L);
    corr = low;
    corr -= CORRECTION;
    corr = ONE_TENTH_CFM / corr;

    /* 读取温度测量值 */
    ADCON = START_ADC1;
    while (ADCON & ADCS);
    degree_c = ADAT;
    degree_c *= 2;

    /* 对 CFM 进行温度补偿 */
    corr *= STD_TEMP;
```



```

corr /= (ZERO_K + degree_c);

/* 读取压力测量值 */
ADCON = START_ADC0;
while (ADCON & ADCS);
psi = ADAT;

/* 对 CFM 进行压力补偿 */
corr *= psi;
corr /= STD_ATM;
cfm = corr;

/* 读取气流量设定值 */
ADCON = START_ADC2;
while (ADCON & ADCS);
setpoint = ADAT;

/* 测试 CFM 速率是否大于或等于设定值, 若是则触发控制继电器 */
if (setpoint > cfm)
    RELAY = 0;
else
    RELAY = 1;

/* 测试 UPDATE 标志是否置位, 若是则将其复位 */
if (UPDATE) {
    UPDATE = 0;

/* 测试 NO_FLOW 标志是否置位, 若是则显示 '00.0' */
    if (NO_FLOW) {
        display0 = ~CFM;
        display1 = ~segments[0];
        display2 = ~(segments[0] | SEG_DP);
        display3 = ~segments[0];
    }

/* 若 NO_FLOW 标志未置位, 采样按键并显示合适的数字 */
    else if (SEL_0) {
        if (SEL_1) {

/* 无键按下时则显示气流量值。若气流量值大于或等于 30cfm 则显示过
   载信息 'EEE', 否则以 '××.×' 格式显示气流量值 */
            if (cfm <= 300) {
                display0 = ~CFM;

```

```
        display1 = ~segments[cfm % 10];
        cfm /= 10;
        display2 = ~(segments[cfm % 10]);
        cfm /= 10;
        display3 = ~segments[cfm % 10];
    }
    else {
        display0 = ~CFM;
        display1 = ~segments[10];
        display2 = ~segments[10];
        display3 = ~segments[10];
    }
}

/* 若温度键(SW1)按下则显示气体温度测量值 */
else {
    display0 = ~DEGREES;
    display1 = ~segments[degree_c % 10];
    degree_c /= 10;
    display2 = ~(segments[degree_c % 10] | SEG_DP);
    degree_c /= 10;
    display3 = ~segments[degree_c % 10];
}
}

else {

/* 若 PSI 键(SW2)按下则显示气体压力测量值 */
    if (SEL_1) {
        display0 = ~PSI;
        display1 = ~segments[psi % 10];
        psi /= 10;
        display2 = ~(segments[psi % 10] | SEG_DP);
        psi /= 10;
        display3 = ~segments[psi % 10];
    }

/* 若设定值键(SW3)按下则显示设定值 */
    else {
        display0 = ~SETPOINT;
        display1 = ~segments[setpoint % 10];
        setpoint /= 10;
        display2 = ~(segments[setpoint % 10] | SEG_DP);
        setpoint /= 10;
    }
}
```

```
        display3 = `segments[setpoint % 10];  
    }  
}  
}  
}  
}
```

第 9 章 Cx51 编译器

9.1 Cx51 编译器简介

80C51 系列单片机是应用最为广泛的一类单片机，目前已有多个半导体厂家提供 400 多种不同类型的 80C51 单片机芯片。众所周知，传统的 8051 单片机片外部扩展地址空间为 64KB ROM+64KB RAM，而 PHILIPS 公司新推出的 80C51Mx、DALLAS 公司新推出的 DS80C390 等单片机，其外部扩展空间可达 16MB 的代码和数据空间。为了适应新一代 80C51 单片机不断发展的需要，Keil 公司在原来 C51 编译器的基础上新推出了 Cx51 编译器，它完全符合 ANSI 标准，支持高达 16MB 的代码和数据空间，Keil 公司还推出了配套的 Ax51 宏汇编器和 Lx51 连接定位器，能生成“OMF2”格式的输出文件，特别适用于 PHILIPS 8051Mx 结构。表 9-1 所示为 Keil 公司提供的开发工具及其所支持的单片机。

表 9-1 Keil 公司提供的开发工具及其所支持的单片机

开发工具	所支持的单片机
C51 编译器	支持传统型 8051 单片机，包括支持 32x64KB 代码组
A51 宏汇编器	
BL51 连接定位器	
C51 编译器（有 OMF2 输出）	支持传统型 8051 和扩展型 8051 芯片如 DALLAS390 等，包括支持代码组和最多 16MB 代码和 XDATA 存储区
Ax51 宏汇编	
Lx51 连接定位器	
CX51 编译器	支持 PHILIPS 8051MX 单片机，包括支持最多 16MB 代码和 XDATA 存储区
Ax51 宏汇编	
Lx51 连接定位器	

Cx51 编译器的作用是将 C 语言源程序翻译成为 8051 系列单片机的可执行代码，需要时还可在执行代码中加入程序调试符号信息。Keil Cx51 编译器根据 80C51 系列单片机的特性提供三十多条编译控制命令。在 μ Vision2 集成开发环境中，可以通过“Project 菜单/Options for Target/C51 标签页”来设置各种控制命令，通过“Project 菜单/Build Target 选项”可以很方便地按所设置的命令对源程序进行编译和连接定位。

也可以按以下格式从命令行上引用：

C51 源程序文件名 控制命令表列
或 Cx51 源程序文件名 控制命令表列

其中,“C51”或“Cx51”是编译器调用命令,“源程序文件名”必须连同扩展名一块输入,“控制命令表列”是若干条用空格分开的编译控制命令。例如,对于一个已编写完成的 C51 源程序 SAMPLE.C,可按以下命令行进行编译:

```
C51 SAMPLE.C DEBUG CODE OPTIMIZE(3) NOPREPRINT
```

该例中采用了 4 条控制命令,这些命令的具体含义将在下一节详细介绍,某些控制命令还可以带有参数,该参数放入紧跟在命令后面的圆括号中。大部分控制命令都有缩写形式,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只需要给出很少几条控制命令就够了,因为编译程序设置了一组默认控制项,如果用户不给出控制命令,编译器将按默认控制项进行编译。

Cx51 编译器的控制命令也可以在 Cx51 源程序中通过预处理命令 `#pragma` 来引用。对于上面的例子,可以在 Cx51 源程序文件的开始处,加入以下内容:

```
#pragma DB CD OT(3) NOPP
```

这样在对源程序进行编译时将自动引用 `#pragma` 后面的各条控制命令,采用预处理命令 `#pragma` 的好处是可以在 Cx51 源程序中按需要设置编译控制。

Cx51 编译器的控制命令可分为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,一般控制命令可多次引用。这一点在源程序中采用 `#pragma` 引用时必须加以注意,首要控制命令通常放在源程序开始处,并且只能出现一次,如果重复使用将导致致命的编译错误而停止编译,一般控制命令则可以多次出现在源程序中任意一行上。

编译命令可分为源控制、列表控制和目标控制三个大类,源控制命令用于宏定义以及确定需要进行编译的文件名。列表控制命令用于规定编译后所产生列表文件的格式以及是否生成某些特殊内容,所生成的列表文件扩展名为“.LST”。目标控制命令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、是否在所生成的目标文件中加入符号调试信息等。目标控制命令最多,作用最大,使用最频繁。Cx51 编译器所生成的目标文件扩展名为“.OBJ”,还可以生成扩展名为“.I”和“.SRC”的输出文件,“.I”输出文件中包含由预处理器命令展开的源文件,对所有的宏都进行展开,同时删除了所有注释。“.SRC”输出文件为从 C51 源程序产生的汇编语言代码文件,可以再用 A51 进行汇编。默认状态下输出文件与源程序文件同名,但扩展名不同。

Cx51 编译器在对源程序进行编译过程中将自动查错,并在编译完成之后给出 0~3 级错误提示,0 级表示没有错误和警告,1 级表示仅有警告,2 级表示同时存在错误和可能的警告,3 级表示存在致命错误。用户可根据错误级别决定是否需要修改源程序文件,在 μ Vision2 环境中将鼠标指向某一个错误提示信息并双击左键,光标会自动跳到编辑窗口中发生错误的源程序行,对于判断错误原因及修改源程序十分方便。

表 9-2 列出了 Keil Cx51 编译器(V7.0 版本)的全部控制命令、缩写形式及默认值。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)。表中某些命令的参数“bname”与被编译的源程序文件有关,其默认值为源程序文件名。

表 9-2 Cx51 的编译控制命令

分类	P/G	控制命令	缩写	默认值	命令功能
源	G	ASM/ ENDASM	—	---	在线汇编起始/终止标记
	G	DEFINE	DF	—	定义在命令行由预处理器使用的名字
	P	[NO]EXTEND	—	EXTEND	[不]支持 Cx51 编译器的特殊扩展
	P	INCDIR	—	—	指定头文件的附加路径名
列表	P	[NO]CODE	[NO]CD	NOCODE	[不]列表文件后附加汇编程序代码
	G	[NO]COND	[NO]CO	COND	[不]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	[不]在列表文件中列出包含文件内容
	P	PAGELNGTH	PL	PAGELNGTH(60)	设置列表文件每页的行数
	P	PAGEWIDTH	PW	PAGEWIDTH(132)	设置列表文件每行的字符数
	P	[NO]PREPRINT	[NO]PP	NOPREPRINT	[不]产生预处理列表文件, 展开所有宏
	P	[NO]PRINT	[NO]PR	PRINT(bname.LST)	[不]产生列表文件
	P	[NO]SYMBOLS	[NO]SB	NOSYMBOLS	[不]产生程序模块使用过的符号表
	P	WARINGLEVEL	WL	WARINGLEVEL(2)	选择警告检测级别
目标	G	[NO]ARGES	[NO]AR	ARGES	[不]使用绝对寄存器寻址方式
	P	[NO]BROWSE	BR	NOBROWSE	[不]允许生成浏览信息
	P	COMPACT	CP	—	紧凑编译模式
	P	[NO]DEBUG	[NO]DB	NODEBUG	[不]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	[不]允许 ANSI 标准整型数据提升
	P	[NO]INTVECTOR	[NO]IV	INTVECTOR	[不]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]MOD517	—	NOMOD517	[不]使用 80517 附加硬件功能
	G	[NO]MODA2	—	NOMODA2	[不]允许 AT82x8252 的双 DPTR
	G	[NO]MODAB2	—	NOMODAB2	[不]允许 ADuCB2 系列双 DPTR
	G	[NO]MODDA	—	NOMODDA	[不]允许 DS80C390 的算术加速器
	G	[NO]MODDP2	—	NOMODDP2	[不]允许使用 DS80C320 的双 DPTR
	G	[NO]MODP2	—	NOMODP2	[不]允许 PHILIPS 和 ATMELWLM 的双 DPTR
	P	NOAMAKE	NOAM	AMAKE	禁止使用项目文件的 AOTOMAKE 功能
	P	[NO]OBJECT	[NO]OJ	OBJECT(bname.OBJ)	[不]命名目标文件
	P	OBJECTADVANCE	OA	---	包含连接定位器的优化信息

- 用鼠标右键点击项目窗口中的文件弹出右键菜单;
- 单击右键菜单中的“Options for File”选项;
- 选中“Properties”标签页;
- 选中“Generate Assembler SRC file”和“Assemble SRC file”复选框。

功能: ASM/ENDASM 命令用于在 Cx51 源程序中标记在线汇编块的起始点和终止点。带有 ASM/ENDASM 块标记的 Cx51 源程序只有使用 SRC 控制命令进行编译时才起作用, 标记块部分将不会生成目标代码。

注: ASM/ENDASM 命令只能以 #pragma 参数的形式使用于 Cx51 源程序中。

例子: #pragma asm/#pragma endasm

例 9.1: 使用 ASM/ENDASM 命令的例子, 源程序如下。

```
extern void test ();
main () {
    test ();
    #pragma asm
        JMP $ ; endless loop
    #pragma endasm
}
```

生成的“.SRC”文件如下:

```
; .\asm.SRC generated from: asm.c
; COMPILER INVOKED BY:
;      f:\Keil\C51\BIN\C51.EXE asm.c SRC(.\asm.SRC)
NAME ASM
?PR?main?ASM          SEGMENT CODE
    EXTRN  CODE (test)
    EXTRN  CODE (?C_STARTUP)
    PUBLIC  main
; extern void test ();
; main () {
    RSEG ?PR?main?ASM
main:
    USING  0
                                ; SOURCE LINE # 2
;    test ();
                                ; SOURCE LINE # 3
                                LCALL  test
; #pragma asm
;    JMP $ ; endless loop
                                JMP $ ; endless loop
; #pragma endasm
; }
```


; SOURCE LINE # 7

RET

; END OF main

END

命令名: DEFINE

缩写: DF

参数: 一个或多个用逗号分隔的符合 C 语言规定的符号名, 每个名称可有一个参数

属性: 首要控制

默认值: 无

 μ Vision2: 在“Optionst 选项/C51 标签页/Define 栏”输入符号名

功能: DEFINE 命令定义了命令行使用的符号名, 可为“if”、“ifdef”及“ifndef”等预处理命令作条件编译。所定义的符号名是对大小写敏感的。

例子: C51 SAMPLE.C DEFINE(check, NoExtRam)
C51 MYPROG.C DF (x1= “1=5”, iofunc= “getkey()”)

命令名: INCDIR

缩写: 无

参数: 无

属性: 首要控制

默认值: 无

 μ Vision2: 在“Options 选项/C51 标签页/Include Paths 栏”输入路径

功能: INCDIR 命令指定 Cx51 编译器头文件所在路径, 编译器最多可接受 50 个路径声明。指定多个路径时各路径名之间需用分号隔开。如果采用了预处理器命令#include “filename.h”, Cx51 编译器首先在当前目录搜索头文件, 然后在源程序文件所在目录搜索头文件, 当搜索不到指定的头文件, 或采用了预处理器命令#include <filename.h>, 则在“INCDIR”指定的路径中搜索头文件, 若仍搜索不到就使用“C51INC”环境变量指定的路径。

例子: C51 SAMPLE.C INCDIR (C:\KEIL\C51\MYINC;C\CHIP_DIR)

命令名: NOEXTEND

缩写: 无

参数: 无

属性: 首要控制

默认值: EXTEND

 μ Vision2: 在“Options 选项/C51 标签页/Misc Controls 栏”输入 NOEXTEND

功能: NOEXTEND 命令使 Cx51 编译器不支持对 ANSI C 的特殊扩展。这时 Cx51 编译器的扩展关键字如 bit、reentrant、interrupt 和 using 等对编译

器是未知的，此时使用扩展关键字将导致编译出错。

例子: C51 SAMPLE.C NOEXTEND
#pragma noextend

9.2.2 列表控制命令

命令名: CODE
缩写: CD
参数: 无
属性: 首要控制
默认值: NOCODE
μVision2: 在“Options 选项/Listing 标签页/”选中“C Compiler Listing”复选框和“Assembly Code”复选框
功能: CODE 命令在列表文件的后面附加一个汇编代码文件，对 C51 源程序中的每个函数都用汇编代码表示出来。编译时默认值为 NOCODE，即不产生附加汇编代码文件。
例子: C51 SAMPLE.C CODE
#pragma code

例 9.2: 采用 CODE 控制命令产生汇编代码列表文件的例子，汇编代码所对应的每条 C51 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”。

```
stmt level    source
  1          extern unsigned char a, b;
  2          unsigned char c;
  3
  4          main(){
  5  1          c = 14 + 15 * ((b / c) + 252);
  6  1      }
```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```
      ; FUNCTION main (BEGIN)
                                ; SOURCE LINE # 4
                                ; SOURCE LINE # 5
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 # 6
000E 22          RET
```

```
; FUNCTION main (END)
```

命令名: COND/NOCOND

缩写: CO/NOCO

参数: 无

属性: 一般控制

默认值: COND

μVision2: 在“Options 选项/Listing 标签页/”选中“C Compiler Listing”复选框和“Conditonal”复选框。

功能: 命令 COND 和 NOCOND 决定被编译的源文件的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, 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 命令使这些被忽略的程序行出现在列表文件中, 但它们没有行号和嵌套级, 从而易于识别。使用 NOCOND 命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 9.3: 这是对一个使用了条件编译预处理命令的源程序采用 COND 命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      1          #if defined (VAX)
6                  c = 13;
7          #elif defined (__TIME__)
8      1          b = 14;
9      1          a = 15;
10     1          #endif
11     1          }
```

例 9.4: 这是对例 9.3 的 Cx51 源程序采用 NOCOND 命令进行编译所产生的列表文件, 文件中没有列出由于条件编译而忽略掉的源程序行。与例 9.3 比较可发现它们的不同之处。

```
stmt level  source
1          extern unsigned char a, b;
2          unsigned char c;
3
4          main() {
5      1          #if defined (VAX)
8      1          b = 14;
9      1          a = 15;
10     1          #endif
11     1          }
```

命令名: EJECT
缩写: EJ
参数: 无
属性: 一般控制
默认值: 无
µVision2: 本命令不能用于命令行, 只能在源文件中以 #pragma 语句参数的形式出现。
功能: EJECT 命令使列表文件换页。
例子: #pragma eject

命令名: LISTINCLUDE
缩写: LC
参数: 无
属性: 一般控制
默认值: NOLISTINCLUDE
µVision2: 在“Options 选项/Listing 标签页/”选中“C Compiler Listing”复选框和“#include Files”复选框。
功能: LISTINCLUDE 命令将在编译时所产生的列表文件中列出包含文件的内容, 使用命令默认值 NOLISTINCLUDE 时将不列出包含文件内容。
例子: C51 SAMPLE.C LISTINCLUDE
#pragma listinclude

命令名: PAGELENGTH
缩写: PL
参数: 括号内一个最大值为 65535 的十进制数
属性: 首要控制
默认值: PAGELENHTH(60)
µVision2: 在“Options 选项/Listing 标签页/Page Length 框”中键入参数。
功能: PAGELENGTH 命令规定了列表文件中每页的行数。默认值为每页 60 行, 包括抬头和空行。
参见: PAGEWIDTH 命令
例子: C51 SAMPLE.C PAGELENGTH(70)
#pragma pl(70)

命令名: PAGEWIDTH
缩写: PW
参数: 括号内一个 78~132 的十进制数
属性: 首要控制
默认值: PAGEWIDTH(132)

μ Vision2: 在“Options 选项/Listing 标签页/Page Width 框”中键入参数。
功能: PAGEWIDTH 命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多行。默认值为每行 132 个字符。

参见: PAGELENGTH 命令

例子: C51 SAMPLE.C PAGEWIDTH(79)
#pragma pw(79)

命令名: PREPRINT

缩写: PP

参数: 括号内可选的文件名

属性: 首要控制

默认值: NOPREPRINT

μ Vision2: 在“Options 选项/Listing 标签页/”选中“C preprocessor Listing”复选框。

功能: PREPRINT 命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 命令不带参数,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,则必须用括号内的参数指出希望的文件名。使用命令默认值将不产生预处理器列表文件。

注: PREPRINT 命令只能在命令行引用,不能在 C51 源程序文件中用 #pragma 命令引用。

例子: C51 SAMPLE.C PREPRINT
C51 SAMPLE.C PREPRINT(PREPRO.LST)

命令名: PRINT/NOPRINT

缩写: PR/NOPR

参数: 括号内可选的文件名

属性: 首要控制

默认值: PRINT(bname.LST)

μ Vision2: 在“Options 选项/Listing 标签页/”中单击“Select Folder for List Files”按钮。

功能: PRINT 命令使用指定的路径、源文件名及扩展名“.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,则可在命令后面的括号内用一个参数名来指定所希望的列表文件名。使用 NOPRINT 命令将不产生列表文件。

例子: C51 SAMPLE.C PRINT(PRN.LST)
#pragma pr(\user\list\samples.lst)

命令名: SYMBOLS

缩写: SB

参数: 无

- 属性: 首要控制
- 默认值: 不产生符号列表文件
- μVision2: 在“Options 选项/Listing 标签页/”选中“C Compiler Listing”复选框和“Symbols”复选框。
- 功能: SYMBOLS 命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。使用命令默认值将使编译器不产生符号列表文件。
- 例子: C51 SAMPLE.C SYMBOLS
#pragma symbols

下面是部分符号列表:

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

- 命令名: WARNINGLEVEL
- 缩写: WL
- 参数: 括号内数字 0~2
- 属性: 首要控制
- 默认值: WARNINGLEVEL (2)
- μVision2 控制: 在“Options 选项/C51 标签页/Warnings 框”内选取警告级别。
- 功能: WARNINGLEVEL 命令允许用户按表 9-3 设置编译时产生警告性错误的级别。

表 9-3 警告错误级别

WARNINGLEVE	说 明
0	禁止大部分编译警告错误
1	仅列出那些可能导致错误编码的警告错误
2	列出所有警告错误, 包括未使用的变量、表达式、标号等

例子: C51 SAMPLE.C WL(1)
 #pragma WARNINGLEVE(0)

9.2.3 目标控制命令

命令名: AREGS/NOAREGS

缩写: 无

参数: 无

属性: 一般控制

默认值: AREGS

μ Vision2: 在“Options 选项/C51 标签页/Code Optimization 栏”选中“Don't use absolute register accesses”复选框。

功能: AREGS 命令使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 命令只能使用直接或绝对寻址。使用 AREGS 命令可以直接向堆栈压入或弹出寄存器。可以采用 REGISTERBANK 命令来规定所使用的寄存器组。NOAREGS 命令关闭绝对寻址方式, 使用 NOAREGS 命令编译的函数不依赖于寄存器组, 即它们可使用 8051 单片机所有的寄存器组, NOAREGS 命令可用于使用了不同寄存器组的函数调用中。

注: AREGS/NOAREGS 命令可作为#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。

例子: C51 SAMPLE.C NOAREGS
 #pragma aregs

例 9.5: 使用 AREGS/NOAREGS 命令的例子。

```

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      }

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION noaregfunc (BEGIN)
; SOURCE LINE # 5
; 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 # 10
; 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)

```

程序中定义了两个函数 `aregfunc()` 和 `noaregfunc()`，它们的内容相同，但分别使用了 `#pragma aregs` 和 `#pragma noaregs` 进行编译，所以产生目标代码有所不同。两个函数都需要将寄存器 R7 压栈，由 `#pragma aregs` 编译产生如下代码：

PUSH AR7

而由 `#pragma noaregs` 编译产生如下代码：

MOV A, R7

PUSH ACC

命令名： **BROWSE**

缩写： **BR**

参数： 无

- 属性: 首要控制
默认值: 不产生浏览信息
 μ Vision2: 在“Options 选项/Output 标签页/”选中“Create Executable”和“Browse Information”复选框。
功能: 用 BROWSE 命令使编译器产生浏览信息, 浏览信息包括标识符及其预处理器符号, 存储空间类型定义以及参考列表。浏览信息可以在 μ Vision2 环境中单击“View 菜单/Source Browser”选项打开浏览。
例子: C51 SAMPLE.C BROWSE
#pragma browse
- 命令名: COMPACT
缩写: CP
参数: 无
属性: 首要控制
默认值: SMALL
 μ Vision2: 在“Options 选项/Target 标签页/Memory Model 框”内选取编译模式。
功能: COMPACT 命令控制编译器对源程序进行编译时采用紧凑存储器模式, 在该模式下所有函数和过程的变量及局部数据段被定义在 8051 单片机的外部数据存储器中, 存储器以 256 字节为 1 页。这种模式使用访问外部数据存储器的简洁形式, 低 8 位地址由 @R0/@R1 决定, 高 8 位地址由 8051 的 P2 口输出。
注: 对于指定了存储器类型的变量, 该编译模式控制命令不起作用。将常用变量放在内部数据存储器中将大大提高程序运行效率。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。
参见: SMALL、LARGE、ROM 命令
例子: C51 SAMPLE.C COMPACT
#pragma cp
- 命令名: DEBUG
缩写: DB
参数: 无
属性: 首要控制
默认值: 不产生调试信息
 μ Vision2: 在“Options 选项/Output 标签页”选中“Create Executable”和“Debug Information”复选框。
功能: DEBUG 命令使编译器将符号调试信息加入到目标文件中, 该信息对于源程序进行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 μ Vision2 调试器或 Intel 兼容的仿真器用来对 Cx51 程序进行源级符号化调试。使用命令默认值

将不产生符号调试信息，因而也就不能对源程序进行符号化调试。

参见：OBJECTEXTEND 命令

例子：C51 SAMPLE.C DEBUG

```
#pragma db
```

命令名：DISABLE

缩写：无

参数：无

属性：一般控制

默认值：无

µVision2：DISABLE 命令只能以 #pragma 参数的形式使用，不能用于命令行。

功能：DISABLE 命令在一个函数的执行期间禁止所有的中断。DISABLE 命令必须在函数的前面以 #pragma 参数的形式出现，并且每个要禁止中断的函数前面必须放一个独立的 #pragma 行。

注：使用 DISABLE 命令禁止了中断的函数不能返回一个位 (bit) 值。

例 9.6：使用 DISABLE 命令的例子。

```
stmt level    source
```

```
1
2      typedef unsigned char uchar;
3
4      #pragma disable /* Disable Interrupts */
5      uchar dfunc (uchar p1, uchar p2) {
6  1          return (p1 * p2 + p2 * p1);
7  1      }
```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```
      ; FUNCTION _dfunc (BEGIN)
0000 D3          SETB      C
0001 10AF01      JBC       EA, ?C0002
0004 C3          CLR       C
0005            ?C0002:
0005 C0D0      PUSH      PSW
                                ; SOURCE LINE # 5
;---- Variable 'p2' assigned to Register 'R5' ----
;---- Variable 'p1' assigned to Register 'R7' ----
                                ; SOURCE LINE # 6
0007 ED          MOV      A, R5
0008 8FF0      MOV      B, R7
000A A4          MUL      AB
000B FE          MOV      R6, A
000C EF          MOV      A, R7
```

```

000D 8DF0          MOV      B,R5
000F A4           MUL      AB
0010 2E           ADD      A,R6
0011 FF           MOV      R7,A

                                ; SOURCE LINE # 7

0012          ?C0001:
0012 D0D0          POP      PSW
0014 92AF          MOV      EA,C
0016 22           RET

; FUNCTION _dfunc (END)

```

命令名: **FLOATFUZZY**

缩写: **FF**

参数: 括号内的数字 0~7

属性: 首要控制

默认值: **FLOATFUZZY(3)**

μVision2: 在“Options 选项/C51 标签页”“Bits to round for float”框内选取一个数字。

功能: **FLOATFUZZY** 命令用于决定执行浮点数比较时的舍入位数, 默认值 3 规定了在进行比较之前对浮点数的最低 3 位进行舍入。

例子: **C51 MYFILE.C FLOATFUZZY(2)**
#pragma FF(0)

命令名: **INTERVAL**

缩写: 无

参数: 括号内可选的数字

属性: 首要控制

默认值: **INTERVAL (8)**

μVision2: 在“Options 选项/C51 标签页/Misc Controls 框”内键入命令及参数。

功能: **INTERVAL** 命令规定中断向量之间的间隔, 该命令主要用于 SIECO-51 单片机, 这种单片机的中断向量间隔为 3 个字节。使用该控制命令后编译器按以下公式计算中断向量的绝对地址:

$$(\text{interval} \times n) + \text{offset} + 3,$$

其中“interval”为 **INTERVAL** 命令指定的参数值 (默认值为 8), “n”为中断号, “offset”为 **INTVECTOR** 命令指定的参数值 (默认值为 0)。

参见: **INTVECTOR/NOINTVECTOR** 命令

例子: **C51 SAMPLE.C INTERVAL (3)**
#pragma interval(3)

命令名: **INTPROMOTE/NOINTPROMOTE**

缩写: **IP/NOIP**

- 参数: 无
- 属性: 首要控制
- 默认值: INTPROMOTE
- µVision2: 在“Options 选项/C51 标签页”选中“Enable ANSI integer promotion rules”复选框。
- 功能: INTPROMOTE 允许 ANSI 标准整型数提升规则。位于“IF”语句中的表达式在进行比较之前将从较小类型向整型提升,从而使 Cx51 编译器与其他 ANSI 标准编译器具有更大的兼容性,对用 Microsoft C 或 Borland C 编写的程序只需要进行少量修改便可移植到 Cx51 中。NOINTPROMOTE 命令禁止使用自动整型数提升规则。由于 8051 是一种 8 位单片机,使用 INTPROMOTE 允许整型数提升规则将导致所生成的执行代码效率降低。
- 例子: C51 SAMPLE.C INTPROMOTE
#pragma nointpromote

例 9.7: 使用 INTPROMOTE 控制命令的例子。

```
stmt level  source
1
2      char c;
3      unsigned char c1,c2;
4      int i;
5
6      main () {
7      1          if (c == 0xff) c = 0; /* never true! */
8      1          if (c == -1) c = 1; /* works */
9      1          i = c + 5;
10     1          if (c1 < c2 +4) c1 = 0;
11     1      }
```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```
      ; FUNCTION main (BEGIN)
                                     ; SOURCE LINE # 6
                                     ; SOURCE LINE # 7

0000 AF00      R    MOV     R7,c
0002 EF        MOV     A,R7
0003 33        RLC     A
0004 95E0      SUBB    A,ACC
0006 FE        MOV     R6,A
0007 EF        MOV     A,R7
0008 F4        CPL     A
0009 4E        ORL     A,R6
```

```

000A 7002          JNZ      ?C0001
000C F500          R        MOV      c,A
000E              ?C0001:
                                ; SOURCE LINE # 8
000E E500          R        MOV      A,c
0010 B4FF03        CJNE     A,#0FFH,?C0002
0013 750001        R        MOV      c,#01H
0016              ?C0002:
                                ; SOURCE LINE # 9
0016 AF00          R        MOV      R7,c
0018 EF            MOV      A,R7
0019 33            RLC      A
001A 95E0          SUBB     A,ACC
001C FE            MOV      R6,A
001D EF            MOV      A,R7
001E 2405          ADD      A,#05H
0020 F500          R        MOV      i+01H,A
0022 E4            CLR      A
0023 3E            ADDC     A,R6
0024 F500          R        MOV      i,A
                                ; SOURCE LINE # 10
0026 E500          R        MOV      A,c2
0028 2404          ADD      A,#04H
002A FF            MOV      R7,A
002B E4            CLR      A
002C 33            RLC      A
002D FE            MOV      R6,A
002E C3            CLR      C
002F E500          R        MOV      A,c1
0031 9F            SUBB     A,R7
0032 EE            MOV      A,R6
0033 6480          XRL      A,#080H
0035 F8            MOV      R0,A
0036 7480          MOV      A,#080H
0038 98            SUBB     A,R0
0039 5003          JNC      ?C0004
003B E4            CLR      A
003C F500          R        MOV      c1,A
                                ; SOURCE LINE # 11
003E              ?C0004:
003E 22            RET
                                ; FUNCTION main (END)
TE. 0 WARNING(S), 0 ERROR(S)

```

对上例采用 NOINTPROMOTE 控制命令编译所生成的目标代码如下:

```

; FUNCTION main (BEGIN)
; SOURCE LINE # 6
; SOURCE LINE # 7
0000 AF00      R      MOV      R7,c
0002 EF        MOV      A,R7
0003 33        RLC      A
0004 95E0      SUBB     A,ACC
0006 FE        MOV      R6,A
0007 EF        MOV      A,R7
0008 F4        CPL      A
0009 4E        ORL      A,R6
000A 7002      JNZ      ?C0001
000C F500      R      MOV      c,A
000E          ?C0001:
; SOURCE LINE # 8
000E E500      R      MOV      A,c
0010 B4FF03    CJNE     A,#0FFH,?C0002
0013 750001    R      MOV      c,#01H
0016          ?C0002:
; SOURCE LINE # 9
0016 E500      R      MOV      A,c
0018 2405      ADD      A,#05H
001A FF        MOV      R7,A
001B 33        RLC      A
001C 95E0      SUBB     A,ACC
001E F500      R      MOV      i,A
0020 8F00      R      MOV      i+01H,R7
; SOURCE LINE # 10
0022 E500      R      MOV      A,c2
0024 2404      ADD      A,#04H
0026 FF        MOV      R7,A
0027 E500      R      MOV      A,c1
0029 C3        CLR      C
002A 9F        SUBB     A,R7
002B 5003      JNC      ?C0004
002D E4        CLR      A
002E F500      R      MOV      c1,A
; SOURCE LINE # 11
0030          ?C0004:
0030 22        RET
; FUNCTION main (END)

```

从以上例子可以看到采用 INT PROMOTE 控制命令所生成的代码长度为 63 字节, 而采用 NOINT PROMOTE 控制命令生成的代码长度只有 49 字节, 因此应根据具体需要来决定是否使用该命令。

命令名: INTVECTOR/NOINTVECTOR
缩写: IV, NOIV
参数: 括号内可选的偏移量
属性: 首要控制
默认值: INTVECTOR (0)
μVision2: 在“Options 选项/C51 标签页/Misc Controls 框”内键入命令及参数。
功能: INTVECTOR 命令使编译器对 8051 单片机的中断产生一个跳转命令 (LJMP 或 AJMP, 取决于 ROM 控制命令) 转移到中断向量目标地址。如果中断向量表的首地址不是从 0x0000 开始, 则需采用偏移量参数。NOINTVECTOR 命令禁止产生向量地址, 从而使用户可利用其他方法 (例如汇编语言程序) 编写中断程序并确定中断向量的入口地址。Cx51 编译器通常采用 3 字节跳转指令 “LJMP” 产生中断向量地址, 中断向量入口地址按下式计算:
$$(\text{interval} \times n) + \text{offset} + 3,$$
其中 “interval” 是 INTERVAL 命令指定的参数值 (默认值为 8), “n” 为中断号, “offset” 为 INTVECTOR 命令指定变量参数值 (默认值为 0)。
参见: INTERVAL 命令
例子: C51 SAMPLE.C INTVECTOR (0x8000)
#pragma noiv

命令名: LARGE
缩写: LA
参数: 无
属性: 首要控制
默认值: SMALL
μVision2: 在“Options 选项/Target 标签页/Memory Model 框”内选取编译模式。
功能: LARGE 命令控制编译器对源程序进行编译时采用大存储器模式, 这种存储器模式对变量的影响如下:

- 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 64K 字节的全部地址空间。
- 由于在访问数据对象时需要采用数据指针 DPTR, 因而是效率不高的数据访问形式。

注: 对于指定了存储器类型的变量, 该编译模式控制命令不起作用, 将常用变量放在内部数据存储器中将大大提高程序运行效率。调用子程序时所用的堆栈始终放在 8051 单片机内部数据存储器中。

参见: SMALL、LARGE、ROM 控制命令

例子: C51 SAMPLE.C LARGE

#pragma la

命令名: MAXARGS

缩写: 无

参数: 编译器为可变长度参数列表保留的字节数

属性: 首要控制

默认值: MAXARGS(15), SMALL 和 COMPACT 编译模式
MAXARGS(40), LARGE 编译模式

µVision2: 在“Options 选项/C51 标签页/Misc Controls 框”内键入命令及参数。

功能: MAXARGS 命令用于在可变长度参数表列中进行参数传递时指定缓冲区的大小。用 MAXAREGS 可定义最大参数个数。必须在 Cx51 函数前面应用 MAXAREGS 命令, 本命令不会与再入函数的最大参数传递个数发生冲突。

例子: C51 SAMPLE.C MAXARGS(20)

例 9.8: 使用 MAXARGS 命令的例子。

```
#pragma maxargs (4)           /* 为参数传递保留 4 个字节 */
#include <stdarg.h>
void func (char typ, ...) {
    va_list ptr;
    char c;
    int i;

    va_start (ptr, typ);
    switch (typ) {
        case 0:                /* 传递 CHAR 型数据 */
            c = va_arg (ptr, char); break;
        case 1:                /* 传递 INT 型数据 */
            i = va_arg (ptr, int); break;
    }
}

void main (void) {
    func (0, 'c');             /* 传递 1 个 CHAR 型数据 */
    func (1, 0x1234);          /* 传递 1 个 INT 型数据 */
}
```

命令名: MOD517/NOMOD517

缩写: 无

参数: 括号内可选的参数名

- 属性: 一般控制
- 默认值: NOMOD517
- μ Vision2: 在“Options 选项/Target 标签页”选中“Use On-Chip Arithmetic Unit”或“Use multiple DPTR registers”复选框。
- 功能: MOD517 命令控制 Cx51 编译器为 80C517 中附加的硬件功能(算术处理器、附加数据指针)产生相应的代码,可提高 int、long、float 数据以及使用附加数据指针函数的操作性能。以下库函数使用附加数据指针: memcpy()、memmove()、memcmp()、strcpy() 和 strcmp()。使用算术处理器的库函数带有“517”后缀。NOMOD517 命令是 Cx51 编译器的默认设置,它控制 Cx51 编译器只使用 8051 的常规硬件资源。当 MOD517 命令带有参数时,80C517 的每个附加部件可被控制如表 9-4 所示:

表 9-4 80C517 附加部件控制

参数	控 制 功 能
无	没有指定参数时(这时也不应有括号),C51 编译器为 80C517 的算术处理器和附加的数据指针 DPTR 产生代码。该设置影响 int、long 和 float 类型数据的运算以及库函数 memcpy()、memmove()、memcmp()、strcpy() 和 strcmp() 的执行效率,所有其他标准库函数都不受影响
NOAU	C51 编译器只使用附加的数据指针而不使用算术处理器。该设置特别适用于编写可被中断服务函数所调用的函数
NODP8	C51 编译器只使用算术处理器而不使用附加的数据指针。当中断函数中未使用关键字“using”时可选 NODP8 参数,这种设置可节省堆栈空间。同时指定参数 NOAU 和 NODP8 时与 NOMOD517 命令功能一样,即 Cx51 编译器只使用 8051 的常规硬件资源

参见: MODA2、MODAD2、MODDA、MODDP2、MODP2 命令

例子: C51 SAMPLE517.C MOD517

```
#pragma MOD51(NOAU)
#pragma MOD517(NODP8)
#pragma MOD517(NOAU, NODP8)
C51 SAMPLE517.C NOMOD517
#pragma NOMOD517
```

命令名: MODA2/NOMODA2

缩写: 无

参数: 无

属性: 一般控制

默认值: NOMODA2

μ Vision2: 在“Options 选项/Target 标签页”选中“Use multiple DPTR registers”复选框。

功能: MODA2 命令控制 Cx51 编译器对 ATMEL80x8252 及兼容单片机的附加

- 硬件（特别是附加 CPU 数据指针）产生相应代码，使用附加数据指针可以提高库函数 `memcpy()`、`memmove()`、`memcmp()`、`strcpy()` 和 `strcmp()` 的操作性能。**NOMODA2** 命令是 Cx51 编译器的默认设置，它控制 Cx51 编译器只使用 8051 的常规硬件资源，不对附加 CPU 数据指针产生代码。
- 参见：MOD517、MODAB2、MODDP2、MODP2 命令
- 例子：C51 SAMPLE.C MODA2

```
#pragma moda2
```

C51 SAMPLE.C NOMODA2

```
#pragma nomoda2
```
- 命令名：MODAB2/NOMODAB2
- 缩写：无
- 参数：无
- 属性：一般控制
- 默认值：NOMODAB2
- μVision2：在“Options 选项/Target 标签页”选中“Use multiple DPTR registers”复选框。
- 功能：MODAB2 命令控制 Cx51 编译器对 AduC8xxB2 及兼容单片机的附加硬件（特别是附加 CPU 数据指针）产生相应代码，使用附加数据指针可以提高库函数 `memcpy()`、`memmove()`、`memcmp()`、`strcpy()` 和 `strcmp()` 的操作性能。**NOMODAB2** 命令是 Cx51 编译器的默认设置，它控制 Cx51 编译器只使用 8051 的常规硬件资源，不对附加 CPU 数据指针产生代码。
- 参见：MOD517、MODA2、MODDP2、MODP2
- 例子：C51 SAMPLE.C MODAB2

```
#pragma modab2
```

C51 SAMPLE.C NOMODAB2

```
#pragma nomodab2
```
- 命令名：MODDA2/NOMODDA2
- 缩写：无
- 参数：无
- 属性：一般控制
- 默认值：NOMODDA2
- μVision2：在“Options 选项/Target 标签页”选中“Use ON-Chip Arithmetic Accelerator”复选框。
- 功能：MODDA2 命令控制 Cx51 编译器对 DS80C390、DS80C400、DS5240 单片机的附加算术加速器硬件产生相应代码，提高对 `int`、`long` 类型数据的操作性能。**OMODDA2** 命令是 Cx51 编译器的默认设置，它控制 Cx51 编译器只使用 8051 的常规硬件资源，不对附加算术加速器硬件产生代

码。采用下面的方法可保证只有一个执行线程使用算术加速器:

- 对于只在主程序中执行或只被一个中断服务函数所调用的函数用 MODDA 命令编译;
- 对其余的函数用 NOMODDA 命令编译。

参见: Mod517

例子: C51 SAMPL390.C MODDA2
#pragma modda2
C51 SAMPL390.C NOMODDA2
#pragma nomodda2

命令名: MODDP2/NOMODDP2

缩写: 无

参数: 无

属性: 一般控制

默认值: NOMODDP2

μVision2: 在“Options 选项/Target 标签页”选中“Use multiple DPTR registers”复选框。

功能: MODDP2 命令控制 C51 编译器为 Dallas80C320、C520、C530、AMD80C521 等单片机中附加的硬件功能(双数据指针)产生相应的代码,使用这些附加硬件功能可提高库函数 memcpy()、memmove()、memcmp()、strcpy()、strcmp()等的执行效率。NOMODDP2 命令是 Cx51 编译器的默认设置,它禁止 Cx51 编译器使用双数据指针。

参见: MOD517、MODA2、MODDP2 命令

例子: C51 SAMPL320.C MODDP2
#pragma moddp2
C51 SAMPL320.C NOMODDP2
#pragma nomoddp2

命令名: MODP2/NOMODP2

缩写: 无

参数: 无

属性: 一般控制

默认值: NOMODP2

μVision2: 在“Options 选项/Target 标签页”选中“Use multiple DPTR registers”复选框。

功能: MODDP2 命令控制 Cx51 编译器为 Philips 或 Atmel 公司生产的 8051 衍生产品中附加硬件功能(双数据指针)产生相应的代码,使用这些附加硬件功能可提高库函数 memcpy()、memmove()、memcmp()、strcpy()、strcmp()等的执行效率。NOMODP2 命令是 Cx51 编译器的默认设置,它

- 禁止 Cx51 编译器使用双数据指针。
- 参见: MOD517、MODA2、MODAB2、MODDP2 命令
- 例子: C51 SAMPLE.C MODP2

```
#pragma modp2
C51 SAMPLE.C NOMODP2
#pragma nomodp2
```
- 命令名: NOAMAKE
- 缩写: NOAM
- 参数: 无
- 属性: 首要控制
- 默认值: 产生 AutoMAKE 信息
- µVision2: 本命令不能用于 µVision2。
- 功能: NOMAKE 命令禁止 Cx51 编译器为项目文件产生 AutoMAKE 信息, 同时禁止寄存器优化信息, 使用该控制命令可使 Cx51 编译产生与以前的版本兼容的代码。
- 例子: C51 SAMPLE.C NOMAKE

```
#pragma NOAM
```
- 命令名: OBJECT/NOOBJECT
- 缩写: OJ/NOOJ
- 参数: 括号内文件名
- 属性: 首要控制
- 默认值: OBJECT(bname.OBJ)
- µVision2: 在“Options 选项/Output 标签页”中单击“Select Folder for Objects”按钮。
- 功能: OBJECT 命令控制编译器使用括号内的参数名作为生成的目标文件名, 默认目标文件名是带路径的源文件名加扩展名“.OBJ”。使用 NOOBJECT 命令将不产生目标文件。
- 例子: C51 SAMPLE.C NOOBJECT

```
#pragma oj(samples_1.obj)
```
- 命令名: OBJECTADVANCED
- 缩写: OA
- 参数: 无
- 属性: 首要控制
- 默认值: 无
- µVision2: 在“Options 选项/C51 标签页/Code Optimization 栏”选中“Linker Code Packing”复选框。
- 功能: OBJECTADVANCED 命令控制编译器在 OBJ 文件中包含对程序连接定

位时的优化信息, 本命令与 OPTIMIZE 命令一起使用以缩短程序代码和执行速度。使用 OBJECTADVANCED 命令时 Lx51 连接器/定位器将执行如表 9-5 所示优化。

表 9-5 使用 OBJECTADVANCED 命令时 Lx51 连接器/定位器将执行的优化

优化级别	连接定位器执行的优化
0~7	连接器重新安排代码段以尽可能使用 AJMP 和 ACALL 指令代替 LJMPL 和 LCALL 指令
8	对单个函数进行多次调用时可重复使用设置代码, 重复使用公共入口代码以减小程序代码长度。本优化在全部的应用中执行
9	反复使用的指令被转换成子程序, 以减小程序代码长度, 但会略微增加代码执行时间。本优化在全部应用中执行
10	当检测到公共子程序块时, 重排代码以得到更多的循环序列
11	重复使用相同的出口以进一步减小公共子程序快的代码长度, 本优化可产生最紧凑的程序代码

参见: OPTIMIZE, OMF2 命令

例子: C51 SAMPLE.C OBJECTADVANCED
#pragma oj(samples_1.obj)

命令名: OBJECTEXTEND

缩写: OE

参数: 无

属性: 首要控制

默认值: 无

μ Vision2: 在“Options 选项/Output 标签页”选中“Create Executable”和“Debug Information”复选框。

功能: OBJECTEXTEND 命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具有相同名字时将其区分出来以供仿真器在调试时使用。

注: 该命令所产生的目标文件包含 OMF-51 对再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仿真器必须具备增强的目标装入功能, 如果有问题, 请不要使用 OBJECTEXTEND 命令。

参见: DEBUG、OMF2 命令

例子: C51 SAMPLE.C OBJECTEXTEND DEBUG
#pragma oe db

命令名: ONEREBANK

缩写: OB

参数: 无

属性:	一般控制
默认值:	无
μVision2:	在“Options 选项/C51 标签页/Misc Controls 框”内键入 ONEREBANK。
功能:	对于无“using”属性的中断服务函数, Cx51 编译器在中断服务函数的开始处产生“MOV PSW #0 指令”代码来使用工作寄存器 0 区, 这样可确保没使用“using”属性的高优先级中断源可以中断使用了其他工作寄存器区的低优先级中断函数。如果应用程序中对于中断函数仅使用一个寄存器区, 用 ONEREBANK 命令可不产生“MOV PSW #0”指令代码。
例子:	C51 SAMPLE.C ONEREBANK #pragma OB
命令名:	OMF2
缩写:	O2
参数:	无
属性:	首要控制
默认值:	Cx51 产生 OMF2 格式目标文件, 而 C51 则产生 Intel OMF51 格式目标文件
μVision2:	在“Project 菜单/Select Device for Target 选项栏”内选中“Use Extended Linker (Lx51) instead of BL51”复选框。
功能:	<p>本命令允许产生 OMF2 格式的目标文件, 对各个模块提供详细的符号类型检查, 同时避免 Intel OMF51 格式文件的限制。若希望利用如下 Cx51 编译器特性则应使用 OMF2 格式目标文件:</p> <ul style="list-style-type: none"> • 变量分组: 采用了“VARBANKING”命令来允许使用“far”存储器类型。 • XDATA ROM: 采用“const xdata”存储类型规定在 code 中定位 xdata 变量。 • RAM 字符串: 采用“STRING”命令规定在 xdata 或 far 空间定位字符串常数。 • 邻接模式: 采用 ROM (D512K) 和 ROM (D16M) 命令允许使用 DS80C390 的邻接模式。
注:	产生 OMF2 格式目标文件需要 Lx51 连接定位器, 而不能采用 BL51 连接定位器。
参见:	OBJECTTEXTEND 命令
例子:	C51 SAMPLE.C OMF2 #pragma O2
命令名:	OPTIMIZE
缩写:	OT
参数:	括号内一个 0~9 的十进制数, 另外可选 SIZE 或 SPEED 以决定优化重

点是放在代码长度上还是放在程序执行速度上。

属性：一般控制

默认值：OPTIMIZE(8, SPEED)

μ Vision2: 在“Options 选项/C51 标签页/Code Optimization 栏/Leve 框”内选取优化级别。

功能：OPTIMIZE 命令设置优化级别和优化重点，共有 9 个优化级别，高优化级别中包含了前面所有的低优化级别。各级优化说明如下。

0 级优化 OPTIMIZE(0)

- 常数折叠：只要有可能，编译器就执行将表达式简化为常数数字的计算，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，因此跳转到跳转的命令将被删除。

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删除。

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 BL51 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：将冗余的 MOV 命令去掉，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：使自动变量和函数参数尽可能位于工作寄存器中，只要有可能，将不为这些变量保留数据存储器空间。
- 扩展访问优化：来自 IDATA、XDATA、PDATA 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，第一次计算的结果被保存，只要有可能，将被用作后续的计算，因此可从代码中消除繁杂的计算。
- CASE/SWITCH 语句优化：将 CASE/SWITCH 语句作为跳转表或跳转串优化。

5 级优化 OPTIMIZE(5)

- 全局公共子式消除：只要有可能，函数内部相同的子表达式只计算一次。中间结果存入一个寄存器以代替新的计算。
- 简单循环优化：以常量占据一段内存的循环在运行时被优化。

6 级优化 OPTIMIZE(6)

- 回路循环: 如果程序代码能更快更有效地执行, 程序回路将进行循环。

7 级优化 OPTIMIZE(7)

- 扩展入口优化: 在合适时对寄存器变量使用 DPTR 数据指针。指针和数组访问被优化以减小程序代码和提高执行速度。

8 级优化 OPTIMIZE(8)

- 公共尾部合并: 对同一个函数有多处调用时, 一些设置代码可被重复使用, 从而减小程序代码长度。

9 级优化 OPTIMIZE(9)

- 公共子程序块: 检测重复使用的指令序列, 并将它们转换为子程序。

Cx51 甚至会重新安排代码已获得更多的重复使用指令序列。

9 级优化包括了从 0 到 8 级的所有优化。

例子: C51 SAMPLE.C OPTIMIZE(9)
C51 SAMPLE.C OPTIMIZE(0)
#pragma ot(6, size)

命令名: ORDER

缩写: OR

参数: 无

属性: 首要控制

默认值: NOORDER

µVision2: 在“Options 选项/C51 标签页”内选中“Keep Variables in order”复选框。

功能: ORDER 命令控制 xC51 编译器将所有变量按其在 Cx51 源程序中的定义顺序安排在存储器中, 该命令禁止 Cx51 编译器的混列算法, 编译速度略有下降。

例子: C51 SAMPLE.C ORDER
#pragma OR

命令名: REGFILE

缩写: RF

参数: 括号内文件名

属性: 首要控制

默认值: 无

µVision2: 在“Options 选项/C51 标签页/Code Optimization 栏”内选中“Global Register Coloring”复选框。

功能: REGFILE 命令控制 Cx51 编译器读入寄存器定义文件来进行全局寄存器优化, 寄存器定义文件规定了外部函数对寄存器的使用, 利用该文件 Cx51 编译器能够了解外部函数是如何利用寄存器的, 从而可在整个程序范围内中进行全局寄存器优化。

例子: C51 SAMPLE.C REGFILE(sample.reg)
#pragma REGFILE(sample.reg)

命令名: REGISTERBANK

缩写: RB

参数: 括号内一个 0~3 的十进制数

属性: 一般控制

默认值: REGISTERBANK(0)

μ Vision2: 在“Options 选项/C51 标签页/Misc Controls 框”内键入命令和参数。

功能: 如果在 Cx51 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 命令选择当前使用的寄存器组。当能够计算寄存器的绝对数量时, 生成的代码为使用寄存器访问的绝对调用形式(代码中用“ARn”表示)。

注: 与关键字“using”不同, REGISTERBANK 命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, 否则函数返回值可能被放在错误的寄存器组中。REGISTERBANK 命令可以作为#pragma 命令的参数在 Cx51 源程序中出现多次, 但是如果在某一函数或过程中使用, 将会被忽略。

例子: C51 SAMPLE.C REGISTERBANK(1)
#pragma rb(3)

命令名: REGPARMS/NOREGPARMS

缩写: 无

参数: 无

属性: 一般控制

默认值: REGPARMS

μ Vision2: 在“Options 选项/C51 标签页/Misc Controls 框”内键入命令。

功能: 使用 REGPARMS 命令时, 编译器在寄存器内至多传递三个参数。这种类型的参数传递类似于汇编语言编程, 不能采用寄存器传递的参数使用固定的存储器区域传递。NOREGPARMS 命令迫使所有函数都通过固定的存储器区域进行参数传递, 该命令产生的参数传递代码与早期的 C51 版本兼容。REGPARM 和 NOREGPARMS 命令可作为预处理命令#pragma 的参数在 Cx51 源程序中使用多次, 从而允许程序的某一部分使用寄存器来传递参数, 而其他函数仍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

例 9.9: 采用预处理命令 `#pragma` 引用 `REGPARMS` 和 `NOREGPARMS` 命令的例子。

```
#pragma noregparms    /* 采用旧方式进行参数传递 */
extern int old_func(int, char);

#pragma regparms      /* 采用新方式进行参数传递 */
extern int new_func(int, char);
main() {
    char a;
    int x1, x2;
    x1=old_func(x2, a);
    x1=new_func(x2, a);
}
```

命令名: RET_PSTK, RET_XSTK

缩写: 无

参数: 无

属性: 首要控制

默认值: 无

µVision2: 在“Options 选项/C51 标签页/Misc Controls 框”内键入命令。

功能: RET_PSTK, RET_XSTK 命令允许使用 `pdata` 或 `xdata` 再入栈保存返回地址。通常返回地址保存在 8051 的硬堆栈（片内 RAM），本命令让编译器产生从硬堆栈弹出返回地址并存入规定再入栈的代码。RET_PSTK 使用 COMPACT 编译模式下的再入栈，RET_XSTK 使用 LARGE 编译模式下的再入栈。

注: 可以使用 RET_PSTK, RET_XSTK 命令从 8051 硬堆栈中取出返回地址。该命令可以选择在具有最深嵌套堆栈的编译模式中使用。使用该命令之前必须对配置文件“STARTUP.A51”中的再入栈指针进行初始化。

例子: C51 Sample.c RET_XSTK

例 9.10: 这是一个使用 RET_XSTK 命令的例子。

```
1      #pragma RET_XSTK
2      extern void func2 (void);
3
4      void func (void) {
5  1      func2 ();
6  1      }

ASSEMBLY LISTING OF GENERATED OBJECT CODE
; FUNCTION func (BEGIN)
0000 120000    E      LCALL ?C?CALL_XBP
; SOURCE LINE # 5
0003 120000    E      LCALL func2
; SOURCE LINE # 6
```

```
0006 020000      E      LJMP ?C?RET_XBP
; FUNCTION func (END)
```

命令名: ROM

缩写: 无

参数: (SMALL), (COMPACT), (LARGE), (D512K), (D16M)

属性: 首要控制

默认值: ROM(LARGE)

μ Vision2: 在“Options 选项/Target 标签页/Code Rom Size 框”内选取 ROM 大小。

功能: ROM 命令用来决定程序代码空间的大小, 它影响跳转和调用指令的编码。

(1) ROM(SMALL):

调用和转移以 ACALL 和 AJMP 指令编码, 最大程序空间可达 2K 字节, 整个用户程序的长度必须分布在这 2K 字节的空间之内。

(2) ROM(COMPACT):

调用以 LCALL 指令编码, 函数内的转移以 AJMP 命令编码。因此函数的长度不得超过 2K 字节, 而整个程序的长度可达 64K 字节, 这种应用必须视不同的需要而定, 看其是否比默认设置 ROM (LARGE) 更好。

(3) ROM(LARGE):

调用和转移以 LCALL 和 LJMP 指令编码, 这样可不加限制地使用整个地址空间, 即用户程序可达 64K 字节。

(4) ROM(D512K):

适用于 DS80C390 及其派生产品, 产生 19 位的 ACALL 和 AJMP 指令编码, 程序最大可达 512KB。

(5) ROM(16M):

适用于 DS80C390 及其派生产品, 产生 24 位的 ACALL 和 AJMP 指令编码, 程序最大可达 16MB。

注: 采用 ROM (D512K) 和 ROM (16M) 命令时需要同时采用 OMF2 命令

参见: SMALL、COMPACT、LARGE 命令

例子: C51 SAMPLE.C ROM(SMALL)

```
#pragma rom(compact)
```

命令名: SAVE/RESTORE

缩写: 无

参数: 无

属性: 一般控制

默认值: 无

μ Vision2: 本命令不能用于 μ Vision2。

功能: SAVE 命令保存当前 ARGES、REGPARMS 的设置、优化级别和优化重

点。RESTORE 命令从 SAVE 栈中取出最近一个 SAVE 命令所保存的值。SAVE 命令的最大嵌套深度为 8。

注: SAVE/RETORE 命令只能在源程序文件中以 #pragma 的参数形式出现,不能用于命令行和 μ Vision2。

例子:

```
#pragma save
#pragma noregparms
extern void test1(char c, int i);
extern char test2(long l, float f);
#pragma retore
```

该例中先用 SAVE 命令保存当前的设置,接着用 NOREGPARMS 命令禁止外部函数 test1 和 test2 在寄存器内进行参数传递,最后再用 RESORE 命令将前面 SAVE 命令所保存的设置恢复。

命令名: SMALL

缩写: SM

参数: 无

属性: 首要控制

默认值: SMALL

μ Vision2: 在“Options 选项/Target 标签页/Memory Model 框”内选取编译模式。

功能: SMALL 命令控制编译器对源程序进行编译时采用小存储器模式。该存储器模式对不同变量的影响如下:

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。

注: 调用了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。

参见: COMPACT、LARGE、ROM 命令

例子: C51 SAMPLE.C SMALL

```
#pragma small
```

命令名: SRC

缩写: 无

参数: 括号内可选的文件名

属性: 首要控制

默认值: 无

μ Vision2: 将鼠标指向项目窗口内的一个 C51 源程序名,双击右键弹出菜单,选中“右键菜单/Properties 标签页/”中的“Generate Assembler SRC File”复选框。

功能: SRC 命令控制 Cx51 编译器在对 Cx51 源文件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。如果不用括号内的参数指出汇编语言源文件名,则用 Cx51 源文件名加上扩展名“.SRC”作为汇编语言源文件名。

注：如果用 SRC 命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，即不能同时产生汇编语言文件和目标文件。

参见：ASM、ENDASM 命令

例子：C51 SAMPLE.C SRC
#pragma src(samples.a51)

命令名：STRING

缩写：ST

参数：(CODE), (XDATA), (FAR)

属性：首要控制

默认值：STRING(CODE)

μVision2：在“Options 选项/C51 标签页/Misc Controls 框”内键入命令。

功能：STRING 命令用来指定固定字符串的存储器类型，CODE 参数为默认值，此时固定字符串被存放在代码区，XDATA 参数使固定字符串被定位在 const xdata 空间，far 参数使固定字符串被定位在 const far 空间。使用不同参数时要小心，因为程序中可能采用基于存储器的指针来访问字符串。通过将字符串定位在 xdata 或 far 空间可以避免使用分组代码，这一点特别适用于 Philips 公司的 80C51Mx 单片机。

参见：OMF2, XCROM 命令

例子：C51 SAMPLE.C STRING(XDATA)
#pragma STRING(FAR)

命令名：USERCLASS

缩写：UCL

参数：(mspace = user_classname)

其中 mspace 可为：CODE、CONST、XCONST、XDATA、HDATA、HCONST 等存储器类型，user_classname 为用户指定的存储器类名

属性：首要控制

默认值：段接受默认类名

μVision2：在“Options 选项/C51 标签页/Misc Controls 框”内键入命令和参数。

功能：USERCLASS 命令为编译器产生的段指定一个用户类名，默认状态下 Cx51 编译器采用基本类名进行段管理。扩展连接定位器 Lx51 在进行段定位时会使用用户类名将一段程序定位在指定的存储器范围内。USERCLASS 命令对整个模块中的基本类名重新命名，但不能对覆盖段重新命名。存储器类名只有在使用 OMF2 格式目标文件和扩展连接定位器 Lx51 时才有效。

例子：#pragma userclass (xdata = flash)
#pragma userclass (hconst = patch)
int xdata x1(10) /* XDATA FLASH */

```
const char far tst[] = "Hello" /* HCONST_PATCH */
```

- 命令名: **VARBANKING**
 缩写: **VB**
 参数: 无, 不修改中断代码
 (1), 在中断代码内保存地址扩展 SFR
 属性: 首要控制
 默认值: 使用标准 Cx51 库函数
 μVision2: 在“Options 选项/Target 标签页/”内选中“far memory type support”和“save address extension SFR in interrupt”复选框。
 功能: **VARBANKING** 命令允许在常规 8051 单片机上使用 far 存储器类型, 这时将使用支持 far 存储器的库函数。对 far 变量函数的访问方式通过 **XBANKING.A51** 文件进行配置。**VARBANKING** 命令(1)命令将在中断函数内保存和返回地址扩展 SFR, 采用配置文件 **XBANKING.A51** 中的 ?C?XPAGE1SFR 来指定该 SFR 的地址。在中断函数的开始部分插入汇编指令: **MOV ?C?XPAGE1SFR, C?XPAGE1RST**。
 注: 只有采用了 **VARBANKING** 命令(1)命令进行编译的 Cx51 函数内的中断函数才会保存和返回地址扩展 SFR。如果应用程序中包含有汇编语言模块的中断函数或库函数, 必须对这些函数仔细检查。
 例子: **C51 SAMPLE.C VARBANKING**
 C51 SAMPLE.C VARBANKING (1)

- 命令名: **XCROM**
 缩写: **XC**
 参数: 无
 属性: 首要控制
 默认值: 在启动代码中对所有 xdata 变量进行初始化。
 μVision2: 在“Options 选项/C51 标签页/Misc Controls 框”内键入命令。
 功能: **XCROM** 命令让 Cx51 编译器将常数变量(constant variables)保存在 xdata 空间而不是 code 空间, 这些变量必须采用“const xdata”声明。这样可以为应用程序释放更多的代码空间。有些新型 8051 器件提供一种存储器管理单元, 可以将 ROM 空间映像到 xdata 空间, 若使用传统 8051, 则对于 xdata 空间应采用 ROM 器件而不是 RAM 器件。
 参见: **OMF2, STRING** 命令
 例子: **#pragma XCROM // Enable const xdata ROM**
 /* 下面的字符数组将存放在位于 xdata 地址空间的 ROM 之中 */
 const char xdata text[] = "Hello world\n";
 void main(void) {
 printf(text);
 }

9.3 Keil Cx51 编译器对 ANSI C 的扩展

9.3.1 存储器类型与编译模式

8051 单片机的存储器空间可分为：片内外统一编址的程序存储器 ROM、片内数据存储器 RAM 和片外数据存储器 RAM。Cx51 编译器对于 ROM 存储器提供存储器类型标识符 `code`，用户的应用程序代码以及各种表格常数被定位在 `code` 空间。数据存储器 RAM 用于存放各种变量，通常应尽可能将变量放在片内 RAM 中以加快操作速度，Cx51 编译器对片内 RAM 提供三种存储器类型标识符：`data`、`idata`、`bdata`。`data` 地址范围为：`0x00~0x7f`，位于 `data` 空间的变量以直接寻址方式操作，速度最快；`idata` 地址范围为：`0x00~0xff`，位于 `idata` 空间的变量以寄存器间接寻址方式操作，速度略慢于 `data` 空间；`bdata` 地址范围为：`0x20~0x2f`，位于 `bdata` 空间的变量除了可以进行直接寻址或间接寻址操作之外，还可以进行位寻址操作。片外数据 RAM 简称 XRAM，Cx51 提供两个存储器类型标识符：`xdata` 和 `pdata`，`xdata` 空间地址范围为 `0x0000~0xffff`，位于 `xdata` 空间的变量以 `MOVX @DPTR` 方式寻址，可以操作整个 64KB 地址范围内的变量，但这种方式速度最慢，`pdata` 空间又称为片外分页 XRAM 空间，它将地址 `0x0000~0xffff` 均匀地分成 256 页，每页的地址都为 `0x00~0xff`，位于 `pdata` 空间的变量以 `MOVX @R0`、`MOVX @R1` 方式寻址。实际上 XRAM 空间并非全部用于存放变量，用户扩展的 I/O 接口也位于 XRAM 地址范围之内。有些新型 80C51 单片机还提供片内 XRAM，其操作方式与传统 XRAM 相同，但一般要先对相应的特殊功能寄存器 SFR 进行配置之后才能使用。

一些新型 8051 单片机能够进行大容量存储器扩展，如 Philips 公司的 80C51Mx 系列可扩展高达 8MB 的 `code` 和 `xdata` 存储器空间，Dallas 公司的 80C390 系列以及 Analog 公司的 Aduc8xx 系列采用 24 位的数据指针 DPTR 以邻接方式可扩展高达 16MB 的 `code` 和 `xdata` 存储器空间。Cx51 编译器针对这种大容量扩展存储器定义了 `far` 和 `const far` 两种存储器类型，分别用以操作这种扩展的片外 RAM 和片外 ROM 空间。对于传统的 8051 单片机，如果它具有可以映像到 `xdata` 的附加存储器空间，或者提供了一种地址扩展特殊功能寄存器（address extension SFR），则可以根据具体硬件电路通过修改配置文件 XBANKING.A51 来使用 `far` 和 `const far` 类型的变量。需要注意的是在使用 `far` 和 `const far` 存储器类型时必须采用 Lx51 扩展连接定位器，同时还必须采用 OMF2 格式的目标文件。

表 9-6 所示为 Keil Cx51 编译器能够识别的存储器类型，定义变量时，可以采用上述存储器类型明确指出变量的存储器空间。

表 9-6 Keil Cx51 编译器能够识别的存储器类型

存储器类型	说 明
<code>code</code>	程序存储器（64K 字节），用 <code>MOVC @A+DPTR</code> 指令访问
<code>data</code>	直接寻址的片内数据存储器（128 字节），访问速度最快
<code>idata</code>	间接访问的片内数据存储器（256 字节），允许访问全部片内地址
<code>bdata</code>	可位寻址的片内数据存储器（16 字节），允许位与字节混合访问

续表

存储器类型	说 明
xdata	片外数据存储器 (64K 字节), 用 MOVX @DPTR 指令访问
pdata	分页寻址的片外数据存储器 (256 字节), 用 MOVX @R0, MOVX @R1 指令访问
far	高达 16MB 的扩展 RAM 和 ROM, 专用芯片扩展访问 (Philips 80C51Mx, DS80C390) 或用户自定义子程序进行访问

例 9.11: 带存储器类型的变量定义。

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];
float idata x,y,z;
unsigned int pdata dimension;
unsigned char xdata vector[10][4][4];
char bdata flags;
```

如果定义变量时没有明确指出具体的存储器类型, 则按 Cx51 编译器采用的编译模式来确定变量的默认存储器空间。Keil Cx51 编译器提供三条编译模式控制命令: SMALL、COMPACT、LARGE, 它们对变量存储器空间的影响如下。

SMALL 所有变量都被定义在 8051 单片机的片内 RAM 中, 对这种变量的访问速度最快。另外, 堆栈也必须位于片内 RAM 中, 而堆栈的长度是很重要的, 实际栈长取决于不同函数的嵌套深度。采用 SMALL 编译模式与定义变量时指定 data 存储器类型具有相同效果。

COMPACT 所有变量被定义在分页寻址的片外 XRAM 中, 每一页片外 XRAM 的长度为 256 字节。这时对变量的访问是通过寄存器间接寻址 (MOVB @R0, MOVB @R1) 进行的, 变量的低 8 位地址由 R0 或 R1 确定, 变量的高 8 位地址由 P2 口确定。采用这种模式时, 必须适当改变配置文件 STARTUP.A51 中的参数: PDATASTART 和 PDATALEN; 同时还必须对 uVision2 的“Options 选项/BL51 Locator 标签页/Pdata 框”中键入合适的地址参数, 以确保 P2 口能输出所需要的高 8 位地址。采用 COMPACT 编译模式与定义变量时指定 pdata 存储器类型具有相同效果。

LARGE 所有变量被定义在片外 XRAM 中 (最大可达 64KB), 使用数据指针 DPTR 来间接访问变量 (MOVB @DPTR), 这种编译模式对数据访问的效率最低, 而且将增加程序的代码长度。采用 LARGE 编译模式与定义变量时指定 xdata 存储器类型具有相同效果。

9.3.2 关于 bit, sbit, sfr, sfr16 数据类型

Keil Cx51 编译器支持标准 C 语言的数据类型, 另外还根据 8051 单片机的特点扩展了 bit, sbit, sfr, sfr16 数据类型。

在 C51 程序中可以定义 bit 类型的变量、函数、函数参数及返回值。例如:

```
static bit done_flag = 0;          /* bit 类型变量 */
bit testfunc (                     /* bit 类型函数 */
```



```

bit flag1,                      /* bit 类型函数参数 */
bit flag2)
{
    .
    .
    .
return (0);                      /* bit 类型返回值 */
}

```

所有 bit 类型的变量都被定位在 8051 片内 RAM 的可位寻址区。由于 8051 单片机的可位寻址区只有 16 个字节长，所以在某个范围内最多只能声明 128 个 bit 类型变量。声明 bit 类型变量时可以带有存储器类型 data、idata 或 bdata。对于 bit 类型变量有如下限制：如果在函数中采用预处理命令“#pragma disable”禁止了中断，或者在函数声明时采用了关键字“using n”明确进行了寄存器组切换，则该函数不能返回 bit 类型的值，否则 Cx51 在进行编译时会产生编译错误；另外不能定义 bit 类型指针，也不能定义 bit 类型数组。

关键字 sbit 用于定义可独立寻址访问的位变量——简称可位寻址变量。Cx51 编译器提供一个存储器类型 bdata，带有 bdata 存储器类型的变量被定位在 8051 单片机片内 RAM 的可位寻址区，带有 bdata 存储器类型的变量可以进行字节寻址也可以进行位寻址，因此对 bdata 变量可用 sbit 指定其中任一为可位寻址变量。需要注意的是采用 bdata 及 sbit 所定义变量都必须是全局变量，并且采用 sbit 定义可位寻址变量时要求基址对象的存储器类型为 bdata。例如，可先定义变量的数据类型和存储器类型：

```

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

```

然后使用 sbit 定义可位寻址变量：

```

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

```

操作符“^”后面的数值范围取决于基址变量的数据类型，对于 char 型而言是 0~7，对于 int 型而言是 0~15，对于 long 型是 0~31。bdata 变量 ibase 和 bdata 数组 bary[4] 可以进行字或字节寻址，sbit 变量可以直接操作可寻址位，例如：

```

ibase = -1;                      /* 字寻址，对 ibase 赋值为 -1 */
bary[3] = 'a';                   /* 字节寻址，对 bary[3] 赋值为 'a' */
Ary37 = 0;                       /* 清 0 bary[3] 的第 7 位 */
mybit15 = 1;                     /* 置 1 ibase 的第 15 位 */

```

对于 bdata 变量可以像 data 变量一样处理，所不同的是 bdata 变量必须位于 8051 单片机的片内 RAM 的可位寻址区，其长度不能超过 16 个字节。

sbit 还可以用于定义结构与联合，利用这一特点可以实现对 float 型数据指定 bit 变量，例如：

```

union lft {
    float mf;
    long ml;
};

bdata struct bad {
    char mc;
    union lft u;
} tcp;

sbit tcpf31 = tcp.u.ml ^ 31;          /* float 数据的第 31 位 */
sbit tcpml0 = tcp.mc ^ 0;
sbit tcpml7 = tcp.mc ^ 7;

```

采用 `sbit` 类型时需要指定一个变量作为基地址，再通过指定该基地址变量的 `bit` 位置来获得实际的物理 `bit` 地址。并不是所有类型变量的物理 `bit` 地址都与其逻辑 `bit` 地址相一致，物理上的 `bit 0` 对应第一个字节的 `bit 0`，物理上的 `bit 8` 对应第二个字节的 `bit 0`。对于 `int` 类型的数据，由于它是按高字节在前的方式存储的，`int` 类型数据的 `bit 0` 应位于第二个字节的 `bit 0`，因此采用 `sbit` 指定 `int` 类型数据 `bit 0` 时应使用物理上的 `bit 8`。

8051 单片机片内 RAM 中与 `idata` 空间相重叠的高 128 个字节（地址范围 80~FFH）称为特殊功能寄存器（SFR）区，单片机内部集成功能的操作都是通过特殊功能寄存器来实现的。有些特殊功能寄存器是可以位寻址的，表 9-7 所示为 8051 单片机特殊功能寄存器一览表，表中带*号的为可位寻址的特殊功能寄存器。

表 9-7 8051 单片机特殊功能寄存器一览表

符 号	地 址	说 明
*ACC	E0H	累加器
*B	F0H	乘法寄存器
*PSW	D0H	程序状态字寄存器
SP	81H	堆栈指针寄存器
DPL	82H	数据存储器指针 低 8 位
DPH	83H	数据存储器指针 高 8 位
*IE	A8H	中断允许寄存器
*IP	B8H	中断优先寄存器
*P0	80H	并行 I/O 口 P0
*P1	90H	并行 I/O 口 P1
*P2	A0H	并行 I/O 口 P2
*P3	B0H	并行 I/O 口 P3
PCON	87H	电源控制与波特率选择寄存器
*SCON	98H	串行口控制寄存器
SBUF	99H	串行口缓冲寄存器

续表

符 号	地 址	说 明
*TCON	88H	定时器控制寄存器
TMOD	89H	定时器方式选择寄存器
TL0	8AH	定时器 0 初值寄存器 低 8 位
TH0	8BH	定时器 0 初值寄存器 高 8 位
TL1	8CH	定时器 1 初值寄存器 低 8 位
TH1	8DH	定时器 1 初值寄存器 高 8 位

为了能够直接访问 8051 系列单片机内部特殊功能寄存器, Cx51 编译器扩充了关键字 sfr 和 sfr16, 利用这种扩充关键字可以在 Cx51 源程序中直接定义 8051 单片机的特殊功能寄存器。定义方法如下:

sfr 特殊功能寄存器名 = 地址常数;

例如:

```
sfr P0 = 0x80;          /* 定义 P0 寄存器, 地址为 0x80 */
sfr SCON = 0x90;        /* 定义串行口控制寄存器, 地址为 0x90 */
```

这里需要注意的是, 在关键字 sfr 后面必须跟一个标识符作为特殊功能寄存器名, 名字可任意选取, 但应符合一般习惯。等号后面必须是常数, 不允许有带运算符的表达式。对于传统 8051 单片机地址常数的范围是 0x80~0xff, 对于 Philips80C51Mx 单片机地址常数的范围是 0x180~0x1ff。

在一些新型 8051 单片机中, 特殊功能寄存器经常组合成 16 位来使用。采用关键字 sfr16 可以定义这种 16 位的特殊功能寄存器。例如: 对于 8052 单片机的定时器 T2, 可采用如下的方法来定义:

```
sfr16 T2 = 0xCC;        /* 定义 TIMER2, 其地址为 T2L=0xCC, T2H=0xCD */
```

这里 T2 为特殊功能寄存器名, 等号后面是它的低字节地址, 其高字节地址必须在物理上直接位于低字节之后。这种定义方法适用于所有新一代的 8051 单片机中新增加的特殊功能寄存器。

在 8051 单片机应用系统中经常需要访问特殊功能寄存器中的一些特定位, 可以利用 Cx51 编译器提供的扩充关键字 sbit 来定义特殊功能寄存器中的可位寻址对象。定义方法有如下三种。

① sbit 位变量名 = 位地址;

这种方法将位的绝对地址赋给位变量, 位地址必须位于 0x80~0xFF 之间。例如:

```
sbit OV = 0xD2;
sbit CY = 0xD7;
```

② sbit 位变量名 = 特殊功能寄存器名^位位置;

当可寻址位位于特殊功能寄存器中时可采用这种方法, “位位置”是一个 0~7 之间的常数。例如:

```
sfr PSW = 0xD0;
sbit OV = PSW^2;
sbit CY = PSW^7;
```

③ sbit 位变量名 = 字节地址^位位置;

这种方法以一个常数(字节地址)作为基地址,该常数必须在 0x80H~0xFF 之间。“位位置”是一个 0~7 之间的常数。例如:

```
sbit OV = 0xD0^2;
sbit CY = 0xD0^7;
```

需要注意的是,用 sbit 定义的特殊功能寄存器中的可寻址位是一个独立的定义类(class),不能与其他位定义和位域互换。

9.3.3 一般指针与基于存储器的指针及其转换

Keil Cx51 编译器支持两种指针类型:一般指针和基于存储器的指针,一般指针需要占用 3 个字节,基于存储器的指针只需要 1~2 个字节,一般指针具有较好的兼容性但运行速度较慢,基于存储器的指针是 Cx51 编译器专门针对 8051 单片机存储器特点进行的扩展,它只适用于 8051 单片机,但具有较高的运行速度。

定义一般指针的方法与 ANSI C 相同,例如:

```
char * sptr;          /* char 型指针 */
int * numptr          /* int 型指针 */
```

一般指针在内存中占用 3 个字节,第一个字节存放该指针的存储器类型编码(由编译模式确定),第二和第三个字节分别存放该指针的高位和低位地址偏移量。存储器类型编码值如表 9-8 所示。

表 9-8 一般指针的存储器类型编码

存储器类型 I	idata/data/bdata	xdata	pdata	code
编码值	0x00	0x01	0xFE	0xFF

一般指针可用于存取任何变量而不必考虑变量在 8051 单片机存储器空间的位置,许多 Cx51 库函数采用了一般指针。函数可以利用一般指针来存取位于任何存储器空间的数据。

定义一般指针时可以在“*”号后面带一个“存储器类型”选项,用以指定一般指针本身的存储器空间位置,例如:

```
char * xdata strptr;          /* 位于 xdata 空间的一般指针 */
int * data numptr;           /* 位于 data 空间的一般指针 */
long * idata varptr;         /* 位于 idata 空间的一般指针 */
```

由于一般指针所指对象的存储器空间位置只有在运行期间才能确定,编译器在编译期间无法优化存储方式,必须生成一般代码以保证能对任意空间的对象进行存取,因此一般

指针所产生的代码运行速度较慢, 如果希望加快运行速度则应采用基于存储器的指针。

基于存储器的指针所指对象具有明确的存储器空间, 长度可为 1 个字节 (存储器类型为 `idata`、`data`、`pdata`) 或 2 个字节 (存储器类型为 `code`、`xdata`)。定义指针时如果在 “*” 号前面增加一个 “存储器类型” 选项, 该指针就被定义为基于存储器的指针。例如:

```
char data * str;           /* 指向 data 空间 char 型数据的指针 */
int xdata * num;          /* 指向 xdata 空间 int 型数据的指针 */
long code * pow;          /* 指向 code 空间 long 型数据的指针 */
```

与一般指针类似, 定义基于存储器的指针时还可以指定指针本身的存储器空间位置, 即在 “*” 号后面带一个 “存储器类型” 选项, 例如:

```
char data * xdata str; /*指向 data 空间 char 型数据的指针, 指针本身在 xdata 空间*/
int xdata * data num; /*指向 xdata 空间 char 型数据的指针, 指针本身在 data 空间*/
long code * idata pow; /*指向 code 空间 long 型数据的指针, 指针本身在 idata 空间*/
```

基于存储器的指针长度比一般指针短, 可以节省存储器空间, 运行速度快, 但它所指对象具有确定的存储器空间, 缺乏兼容性。

一般指针与基于存储器的指针可以相互转换。在某些函数调用中进行参数传递时需要采用一般指针, 例如, Cx51 的库函数 `printf()`、`sprintf()`、`gets()` 等便是如此, 当传递的参数是基于存储器的指针时, 若不特别指明, Cx51 编译器会自动将其转换为一般指针。需要注意的是, 如果采用基于存储器的指针作为自定义函数的参数, 而程序中又没有给出该函数原型, 则基于存储器的指针就自动转换为一般指针。假如在调用该函数时的确需要采用基于存储器的指针 (其长度较短) 作为传递参数, 那么指针的自动转换就可能导致错误, 为避免这类错误, 应该在程序的开始处用预处理命令 “`#include`” 将函数原型说明文件包含进来, 或者直接给出函数原型声明。

例 9.12: 函数中指针类型的转换。

```
extern int printf(void *format,...);           /* 函数原型声明 */
extern int myfunc(void code *p,int xdata *pq);

int xdata *px;
char code *fmt = *value = %d | %4XH\n";
void debuf_print(void) {
    printf(fmt,*px,*px); /* printf()函数中的指针 fmt 被自动转换 */
    myfunc(fmt,px);      /* 有原型声明的自定义函数, 指针 fmt 不进行转换 */
}
```

9.3.4 Cx51 编译器对 ANSI C 函数定义的扩展

(1) Cx51 编译器支持的函数定义一般形式

Cx51 编译器提供了几种对于 ANSI C 函数定义的扩展, 可用于选择函数的编译模式、规定函数所使用的工作寄存器组、定义中断服务函数、指定再入方式等。在 Cx51 程序中进行函数定义的一般格式如下:

```

函数类型 函数名 (形式参数表) [编译模式] [reentrant] [interrupt n] [using n]
{ 局部变量定义
  函数体语句
}

```

其中,“函数类型”说明了自定义函数返回值的类型。

“函数名”是用标识符表示的自定义函数名字。

“形式参数表”中列出了在主调用函数与被调用函数之间传递数据的形式参数,形式参数的类型必须要加以说明。如果定义无参函数,可以没有形式参数表,但圆括号不能省略。

“局部变量定义”是对在函数内部使用的局部变量进行定义。

“函数体语句”是为完成该函数的特定功能而设置的各种语句。

“编译模式”选项是 Cx51 对 ANSI C 的扩展,可以是 SMALL、COMPACT 或 LARGE,用于指定函数中局部变量和参数的存储器空间。

“reentrant”选项是 Cx51 对 ANSI C 的扩展,用于定义再入函数。

“interrupt n”选项是 Cx51 对 ANSI C 的扩展,用于定义中断服务函数,其中“n”为中断号,可为 0~31,根据中断号可以决定中断服务程序的入口地址。

“using n”选项是 Cx51 对 ANSI C 的扩展,其中“n”可以是 0~3,用于确定中断服务函数所使用的工作寄存器组。

(2) 堆栈及函数的参数传递

函数在运行过程中需要使用堆栈,8051 单片机的堆栈必须位于片内 RAM 空间,其最大范围只有 256 个字节。(对于一些新的扩展型 8051 单片机, Cx51 编译器可以使用其扩展堆栈区,扩展堆栈区最大可达几千个字节。)为了节省堆栈空间, Cx51 编译器采用一个固定的存储器区域来进行函数参数的传递,发生函数调用时,主调函数先将实际参数复制到该固定的存储器区域,然后再将程序流程控制交给被调函数,被调函数则从该固定的存储器区域取得所需要的参数进行操作。这样就只需要将函数的返回地址保存到堆栈区中。由于中断服务函数可能要进行工作寄存器组切换,因此需要采用较多的堆栈空间。

Cx51 编译器可以采用控制命令“REGPARMS”和“NOREGPARMS”来决定是否通过工作寄存器传递函数参数,在默认状态下, Cx51 编译器可以通过工作寄存器传递最多 3 个函数参数,这种方式可以提高程序执行效率。如果没有寄存器可用,则通过固定的存储器区域来传递函数的参数。

(3) 函数的编译模式

不同类型 8051 单片机片内 RAM 空间大小不同,有些衍生产品只有 64 个字节的片内 RAM,因此在定义函数时要根据具体情况来决定应采用的编译模式,函数参数和局部变量都存放在由编译模式决定的默认存储器空间。可以根据需要对不同函数采用不同的编译模式,在 SMALL 编译模式下函数参数和局部变量被存放在 8051 的片内 RAM 空间,这种方式对数据的处理效率最高。但片内 RAM 空间有限,对于较大的程序若采用 SMALL 编译模式可能不能满足要求,这时就需要采用其他编译模式。

例 9.13: 不同函数采用不同的编译模式。

```

#pragma small                                /* 默认编译模式为 SMALL */
extern int calc (char i, int b) large reentrant; /* 采用 LARGE 编译模式 */
extern int func (int i, float f) large;         /* 采用 LARGE 编译模式 */
extern void *tcp (char xdata *xp, int ndx) small; /* 采用 SMALL 编译模式 */

int mtest (int i, int y) {                   /* 采用默认编译模式 */
    return (i * y + y * i + func(-1, 4.75));
}

int large_func (int i, int k) { large         /* 采用 Large 编译模式 */
    return (mtest (i, k) + 2);
}

```

(4) 寄存器组切换

8051 单片机片内 RAM 中最低 32 个字节平均分为 4 个组, 每组 8 个字节都命名为 R0~R7, 统称为工作寄存器组, 这一特点对于编写中断服务函数或使用实时操作系统都十分有用。利用扩展关键字“using”可以在定义函数时规定所使用的工作寄存器组, 只要在“using”后面跟一个数字 0~3, 即可规定所使用的工作寄存器组。

例 9.14: 利用扩展关键字“using”规定函数所使用的工作寄存器组。

```

stmt level source
1
2          extern bit alarm;
3          int alarm_count;
4          extern void alfunc (bit b0);
5
6          void falarm (void) using 3 {
7      1      alarm_count++;
8      1      alfunc (alarm = 1);
9      1      }

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION falarm (BEGIN)
0000 C0D0          PUSH PSW
0002 75D018      MOV PSW,#018H
; SOURCE LINE # 6
; SOURCE LINE # 7
0005 0500      R      INC alarm_count+01H
0007 E500      R      MOV A,alarm_count+01H
0009 7002          JNZ ?C0002
000B 0500      R      INC alarm_count
000D ?C0002;
; SOURCE LINE # 8

```

```

000D D3          SETB C
000E 9200   E    MOV alarm,C
0010 9200   E    MOV ?alfunc?BIT,C
0012 120000 E    LCALL alfunc
                ; SOURCE LINE # 9
0015 D0D0          POP PSW
0017 22          RET
                ; FUNCTION faalarm (END)

```

从上面例子的汇编代码中可以看到，在函数入口处先将 PSW 保存到堆栈中，然后根据“using”后面的数字重新设置 PSW 以规定函数所使用的工作寄存器组，在函数返回之前要从对堆栈中恢复前面所保存的 PSW。工作寄存器组切换对于中断服务函数十分有用，一般来说，可以对所有一般函数规定同一个工作寄存器组，对高优先级的中断服务函数规定第二个工作寄存器组，对低优先级的中断服务函数规定第三个工作寄存器组。

需要注意的是，关键字 using 不能用在以寄存器返回一个值的函数中，并且要保证任何寄存器组的切换都只在仔细控制的区域内发生，如果不做到这一点将产生不正确的函数结果。另外带“using”属性的函数原则上不能返回 bit 类型的值。

8051 单片机复位时 PSW 的值为 0x00，因此在默认状态下所有非中断函数都将使用工作寄存器 0 区。Cx51 编译器可以通过控制命令“REGISTERBAN”为源程序中的所有函数指定一个默认的工作寄存器组，为此用户需要修改启动代码选择不同的寄存器组，然后采用控制命令“REGISTERBAN”来指定新的工作寄存器组。

在默认状态下 Cx51 编译器生成的代码将使用绝对寻址方式来访问工作寄存器 R0~R7，从而提高操作性能。绝对寄存器寻址方式可以通过编译控制命令“AREGS”或“NOARGES”来激活或禁止。采用了绝对寄存器的函数不能被另一个使用了不同工作寄存器组的函数所调用，否则会导致不可预知的结果。为了使函数对当前工作寄存器组不敏感，该函数必须采用控制命令“NOARGES”进行编译，这一点对于需要同时从主程序和使用不同寄存器组的中断服务程序中调用的函数时十分有用。

特别需要注意的是，Cx51 编译器对函数之间使用的工作寄存器组是否匹配不作检查，因此使用了交替寄存器组的函数只能调用没有设定默认寄存器组的函数。

(5) 中断函数

利用扩展关键字“interrupt”可以直接在 Cx51 程序中定义中断服务函数，在“interrupt”跟一个 0~31 的数字，用于规定中断源和中断入口。关键字“interrupt”对中断函数目标代码的影响如下：

- 在进入中断函数时，特殊功能寄存器 ACC、B、DPH、DPL、PSW 将被保存入栈；
- 如果不使用关键字 using 进行工作寄存器组切换，则将中断函数中所用到的全部工作寄存器都入栈保存；
- 函数退出之前所有的寄存器内容出栈恢复；
- 中断函数由 8051 单片机指令 RETI 结束；
- Cx51 编译器根据中断号自动生成中断函数入口向量地址。

下面给出一个带有寄存器组切换的中断函数定义的例子，该例中还给出了 Cx51 编译

器所生成的 8051 单片机指令代码，由于该例中不需要使用工作寄存器，因此由“using”所产生的寄存器组切换代码被优化掉了。

例 9.15: 利用扩展关键字“interrupt”定义中断函数。

```
stmt level source
1          extern bit alarm;
2          int alarm_count;
3
4
5          void falarm (void) interrupt 1 using 3 {
6      1          alarm_count *= 2;
7      1          alarm = 1;
8      1      }

ASSEMBLY LISTING OF GENERATED OBJECT CODE
          ; FUNCTION falarm (BEGIN)
0000 C0E0          PUSH ACC
0002 C0D0          PUSH PSW
                                ; SOURCE LINE # 5
                                ; SOURCE LINE # 6
0004 E500      R      MOV A,alarm_count+01H
0006 25E0          ADD A,ACC
0008 F500      R      MOV alarm_count+01H,A
000A E500      R      MOV A,alarm_count
000C 33          RLC A
000D F500      R      MOV alarm_count,A
                                ; SOURCE LINE # 7
000F D200      E      SETB alarm
                                ; SOURCE LINE # 8
0011 D0D0          POP PSW
0013 D0E0          POP ACC
0015 32          RETI
          ; FUNCTION falarm (END)
```

在 Cx51 源程序中定义中断函数时应遵循以下规则。

- 中断函数不能进行参数传递，如果中断函数中包含任何参数声明都将导致编译出错。
- 中断函数没有返回值，如果企图定义一个返回值将得到不正确的结果。因此建议在定义中断函数时将其定义为 void 类型，以明确说明没有返回值。
- 在任何情况下都不能直接调用中断函数，否则会产生编译错误，因为中断函数的退出是由 8051 单片机指令 RETI 完成的，RETI 指令影响 8051 单片机的硬件中断系统。如果在没有实际中断请求的情况下直接调用中断函数，RETI 指令的操作结果会产生一个致命的错误。另外也不要通过函数指针来间接调用中断函数。
- 如果在中断函数中调用了其他函数，则被调用函数所使用的寄存器组必须与中断函

数相同。用户必须保证按要求使用相同的寄存器组，否则会产生不正确的结果，这一点必须引起足够的注意。如果没有明确采用编译控制命令“NOAREGS”，Cx51 编译器将根据关键字“using”，或者由“REGISTERBANK”编译控制命令自动选择一个寄存器组并进行绝对寄存器访问。另外，由于中断的产生不可预测，中断函数对其他函数的调用可能形成递归调用，需要时可将被中断函数所调用的其他函数定义成再入函数。

- Cx51 编译器根据关键字“interrupt”后面的数字 n ，从绝对地址 $8n+3$ 处产生一个中断向量，其中 n 为中断号。该向量包含一个到中断函数入口地址的绝对跳转。在对源程序编译时，可用编译控制命令“NOINTVECTOR”抑制中断向量的产生，从而使用户可以从独立的汇编程序模块中提供中断向量。

(6) 再入函数

利用 Cx51 编译器的扩展关键字“reentrant”可以定义一个再入函数，再入函数可以进行递归调用，或者同时被两个以上其他函数同时调用。通常在实时系统应用中，或中断函数与非中断函数需要共享一个函数时，应将该函数定义为再入函数。

例 9.16: 利用扩展关键字“reentrant”定义再入函数。

```
int calc(char i, int b) reentrant {
    int x;
    x = table[i];
    return (x * b);
}
```

再入函数可被递归调用，无论何时，包括中断服务函数在内的任何函数都可调用再入函数。与非再入函数的参数传递和局部变量的存储分配方法不同，Cx51 编译器为再入函数生成一个模拟栈，通过这个模拟栈来完成参数传递和存放局部变量。根据再入函数所采用的编译模式，模拟栈可以位于片内或片外存储器空间，SMALL 模式下再入栈位于 data 空间，COMPACT 模式下再入栈位于 pdata 空间，LARGE 模式下再入栈位于 xdata 空间。当程序中包含有多种存储器模式的再入函数时，Cx51 编译器为每种模式单独建立一个模拟栈并独立管理各自的栈指针。再入函数的局部变量及参数都被放在再入栈中，从而使再入函数可以进行递归调用。而非再入函数的局部变量被放在再入栈之外的暂存区内，如果对非再入函数进行递归调用，则上次调用时使用的局部变量数据将被覆盖。

对于再入函数有如下规定：

- 再入函数不能传送 bit 类型的参数，也不能定义局部位变量，再入函数不能操作可位寻址变量。
- 与 PL/M51 兼容的 alien 函数不能具有 reentrant 属性，也不能调用再入函数。
- 再入函数可以同时具有其他属性，如“interrupt”、“using”等，还可以明确声明其存储器模式（SMALL、COMPACT、LARGE）。
- 在同一个程序中可以定义和使用不同存储器模式的再入函数，每个再入函数都必须具有合适的函数原型，原型中还应包含该函数的存储器模式。
- 再入函数的返回地址保存在 8051 单片机的硬件堆栈内，任意其他的 PUSH 和 POP

指令都会影响 8051 硬件堆栈。

- 不同存储器模式下的再入函数具有其自己的模拟再入栈以及再入栈指针，例如，若在一个模块内定义了 SMALL 和 LARGE 模式的再入函数，则 Cx51 编译器会同时生成对应的两种再入栈及其再入栈指针。

再入函数的模拟再入栈结构的使用效率是不高的，因此使用再入函数时必须小心谨慎。模拟再入栈以全局变量“?C_IBP”、“?C_PBP”和“?C_XBP”作为再入栈指针，这些变量定义在 data 空间，并且可在文件 STARTUP.A51 中进行初始化。表 9-9 所示为模拟再入栈空间与存储器模式之间的关系。

表 9-9 模拟再入栈空间与存储器模式之间的关系

存储器模式	模拟再入栈指针	模拟再入栈空间
SMALL	?C_IBP (1 字节)	间接访问的片内 RAM (idata) 区，最大为 256 字节
COMPACT	?C_PBP (1 字节)	分页寻址的外部 XRAM (pdata) 区，最大为 256 字节
LARGE	?C_XBP (2 字节)	外部 XRAM (xdata) 区，最大为 64K 字节

8051 单片机的常规栈总是位于内部数据 RAM 中而且是“向上生长”型的，而模拟再入栈是“向下生长”型的，如果编译时采用 SMALL 模式，常规栈和再入函数的模拟栈将被放在内部 RAM 中，从而可使有限的内部数据存储器得到充分利用。模拟再入栈及其再入栈指针可以通过配置文件“STARTUP.A51”进行调整，使用再入函数时应根据需要对该配置文件进行适当修改。

(7) “alien”函数（与 PL/M51 接口）

Cx51 编译器允许在 C51 源程序中采用关键字“alien”来与 PL/M51 函数进行直接和简单的接口。C51 源程序中的函数采用关键字“alien”之后，就可以被 PL/M51 程序所调用。PL/M51 函数的参数和返回值可以是以下数据类型：bit、char、unsigned char、int、unsigned int，其他数据类型包括 long、float、各种指针等都可以在 C51 源程序中通过关键字“alien”进行定义，不过再使用这些数据类型时要小心，因为 PL/M51 不直接支持 32 位二进制整数及浮点数。另外，在 PL/M51 中也可以定义“外部公共变量”(external public variable)，这种变量可以在 C51 程序中使用。

例 9.17：利用关键字“alien”实现与 PL/M51 函数接口。

```
extern alien char plm_func(int,char);
char c_func(void) {
    int i;
    char c;
    for( i=0;i<100;i++) {
        c = plm_func(i,c); /* call PL/M func */
    }
    return( c );
}
```

对于 PL/M51 函数必须采用关键字“alien”，只有如此，PL/M51 函数的参数传递及函

数返回值才能在 Cx51 编译器中被考虑。

(8) 实时任务函数

Cx51 编译器通过关键字 “_task_” 和 “_priority_” 提供对实时多任务操作系统 RTX51 FULL 和 RTX51 Tiny 的支持。关键字 “_task_” 用于定义实时任务函数，关键字 “_priority_” 用于规定任务的优先级。例如：

```
void func(void) _task_ num _priority_ pri
```

其中，“num” 为任务函数的 ID 号，采用 RTX51 FULL 时为 0~255，采用 RTX51 Tiny 时为 0~15。

“pri” 为任务函数的优先级。

实时任务函数的返回类型和参数都必须定义为 “void”。

9.4 Cx51 编译器的数据调用协议

9.4.1 数据在内存中的存储格式

本书第 2 章已经介绍过 Cx51 编译器所支持的数据类型。这里将与 8051 系列单片机有关的数据类型在内存中的存储格式叙述如下。

“bit” 类型数据只有一位长度，不允许定义位指针和位数组。“bit” 对象始终位于 8051 单片机内部可位寻址数据存储器空间（20H~2FH），只要有可能 BL51 连接定位器将对位对象进行覆盖操作。

“char” 类型数据的长度为一个字节（8 位），可存放于 8051 单片机内部或外部数据存储器中。

“int” 和 “short” 类型数据的长度为 2 个字节（16 位），可存放于 8051 单片机内部或外部数据存储器中。数据在内存中按高字节地址在前、低字节地址在后的顺序存放，例如，一个值为 0x1234 的 “int” 类型数据，在内存中的存储格式如下：

地址	+0	+1
内容	0x12	0x34

“long” 类型数据的长度为 4 个字节（32 位），可存放于 8051 单片机内部或外部数据存储器中。数据在内存中按高字节地址在前、低字节地址在后的顺序存放，例如，一个值为 0x12345678 的 “long” 类型数据，在内存中的存储格式如下：

地址	+0	+1	+2	+3
内容	0x12	0x34	0x56	0x78

“float” 类型数据的长度为 4 个字节（32 位），可存放于 8051 单片机内部或外部数据存储器中。一个 “float” 类型数据的数值范围是 $(-1)^S \times 2^{E-127} \times (1.M)$ 。在内存中按 IEEE-754 标准单精度 32 位浮点数的格式存储：

地址	+0	+1	+2	+3
内容	SEEEEEEE	EMMMMMMM	MMMMMMMM	MMMMMMMM

其中, S 为符号位, “0” 表示正, “1” 表示负。E 为用原码表示的阶码, 占用 8 位二进制数, 存放在两个字节中, E 的取值范围是 1~254。注意, 实际上以 2 为底的指数要用 E 的值减去偏移量 127, 从而实际幂指数的取值范围为 -126~+127。M 为尾数的小数部分, 用 23 位二进制数表示, 存放在三个字节中。尾数的整数部分永远为 1, 因此不予保存, 但它是隐含存在的。小数点位于隐含的整数位 “1” 的后面。

例如, 一个值为 -12.5 的 “float” 类型数据, 在内存中的存储格式如下:

地址	+0	+1	+2	+3
二进制内容	11000001	01001000	00000000	00000000
十六进制内容	0xC1	0x48	0x00	0x00

按上述规则很容易将用十六进制表示的数据 “0xC1480000” 转换为浮点数 -12.5。

一个浮点数的正常数值范围是: $(-1)^S \times 2^{E-127} \times (1.M)$, 其中, $E=0 \sim 255$, $S=\pm 1$ 。超过最大正常数值的浮点数就认为是无穷大, 其阶码 E 为全 1 (即 255), 小数部分 M 为全 0, 表示为: $\pm \infty = (-1)^S \times 2^{128} \times (1.000...000) = \pm 2^{128}$ 。

对于阶码 E 为全 0, 小数部分 M 也为全 0 的浮点数认为是 0, 表示为:

$$(-1)^S \times 2^{-127} \times (1.000...000) = \pm 2^{-127}.$$

绝对值最小的正常浮点数为阶码 E 为 1, 小数部分 M 为全 0 的数, 表示为:

$$(-1)^S \times 2^{-126} \times (1.000...000) = \pm 2^{-126}.$$

除了正常数之外, 介于 $+2^{-126} \sim +2^{-127}$ 以及 $-2^{-126} \sim -2^{-127}$ 之间的数为非正常数。按 IEEE-754 标准, 浮点数的数值如果在正常数值之外, 即为溢出错误, 用下面的二进制数表示:

非正常数: NaN=0FFFFFFFH

正无穷: +INF=7F800000H

负无穷: -INF=FF800000H

Cx51 编译器支持 “基于存储器” 的指针和 “一般” 指针。基于存储器类型 data、idata 和 pdata 的指针具有 1 个字节的长度, 基于存储器类型 xdata 和 code 的指针具有 2 个字节的长度, 一般指针具有 3 个字节的长度。在一般指针的 3 个字节中, 第一个字节表示存储器类型, 第二、第三个字节表示指针的地址偏移量, 一般指针在内存中的存储格式为:

地址	+0	+1	+2
内容	存储器类型	高字节地址偏移量	低字节地址偏移量

第一个字节中存储器类型的编码如下:

存储器类型	idata/data/bdata	xdata	pdata	code
编码值 (8051)	0x00	0x01	0xFE	0xFF
编码值 (8051Mx)	0x7F	0x00	0x00	0x80

采用一般指针时必须使用规定的存储器类型编码值, 如果使用其他类型的值将导致不可预测的后果。

例如, 将 xdata 类型的地址 0x1234 作为一般指针表示如下:

地址	+0	+1	+2
内容	0x01	0x12	0x34

9.4.2 目标代码的段管理

段是程序代码或数据对象的存储器单位，程序代码被放入代码段，数据对象被放入数据段。段又分为绝对段和再定位段，绝对段只能在汇编语言程序中指定，它包括代码和数据的绝对地址说明。绝对段在用连接定位器 BL51 进行连接时，已经分配的地址将不发生任何改变。再定位段是由 Cx51 编译器在对 C51 源程序编译时所产生的，再定位段中代码或数据的存储器地址是浮动的，实际地址要由连接定位器 BL51 在对程序模块进行连接时决定。再定位段可以保证在进行多模块程序连接时不会发生地址重叠现象。因此绝对段只是用于某些特殊场合，如访问某个固定的存储器 I/O 地址，或是提供某个中断向量的入口地址，而用 Cx51 编译器对 C51 源程序进行编译所产生的段都是再定位段。每一个再定位段都具有段名和存储器类型，绝对段则没有段名。

下面介绍 Cx51 编译器对于再定位段的管理方法。

Cx51 编译器在对 C51 源程序进行编译时，为了适应不同的要求和便于段的管理，将程序中的每个数据对象都转换成大写形式保存，并放入到相应的段中。Cx51 编译器按表 9-10 中的规则将源程序中的函数名转换成目标文件中的函数名，BL51 在连接定位时将使用目标文件中的函数名。

表 9-10 Cx51 编译器的函数名转换规则

函数声明	转换的符号名	说 明
void func(void)...	FUNC	无参数传递或不带寄存器参数的函数名不作改变地转入目标文件中，函数名只简单地转换成大写形式
void func(char)...	_FUNC	带寄存器参数的函数名前面加上“_”前缀，表示这类函数包含有寄存器内的参数传递
void func(void) reentrant...	__FUNC	再入函数在函数名前面加上“__”前缀，表示该函数包含栈内的参数传递

完成函数名转换之后，Cx51 编译器按以下规则将不同的数据对象组合到不同的数据段中。

(1) 全局变量

对于全局变量 Cx51 编译器按表 9-11 中的规则为每个模块生成各自的段名，具有相同存储器类型的全局变量被组合到同一个数据段中。明确定义了存储器类型的全局变量，都有一个单独的数据段。段名由两个问号中间加一个存储器类型符号及紧接着的模块名(modulname)组成，模块名是不带路径和扩展名的源文件名，常数和字符串被放入一个独立的段中。各段名表示在对应类型存储器空间的起始地址。

表 9-11 Cx51 编译器对全局变量的段名生成规则

段 名	存储器类型	说 明
?CO?modulname	code	可执行程序存储器中的常数段
?XD?modulname	xdata	xdata 型数据段 (RAM 空间)

续表

段 名	存储器类型	说 明
?DT?modulname	data	data 型数据段
?ID?modulname	idata	idata 型数据段
?BI?modulname	bit	bit 型数据段
?BA?modulname	bdata	bdata 型数据段
?PD?modulname	pdata	pdata 型数据段
?XC?modulname	const xdata	xdata 型数据段 (const ROM 空间), 需要用“OMF2”编译控制命令
?FC?modulname	const far	far 型常数段 (const ROM 空间), 需要用“OMF2”编译控制命令
?FD?modulname	far	far 型数据段 (RAM 空间), 需要用“OMF2”编译控制命令

(2) 函数和局部变量

Cx51 编译器为各个模块中的每个函数生成一个以?PR? function_name? modulname 为名的代码(CODE)段。例如, 如果程序模块 SAMPLE.C 中包含有一个名为 ERROR_CHECK 的函数, 则代码(CODE)段的名称为?PR? ERROR_CHECK? SAMPLE。

如果函数中包含有非寄存器传递的参数和无明确存储器类型声明的局部变量, Cx51 编译器除了生成该函数的代码段之外, 还将生成一个字节类型的局部数据段(简称局部数据段)和一个位类型的局部数据段(简称局部位段)。局部位段用于存放在函数内部定义的可再定位的位类型变量和参数, 局部数据段则用于存放除位类型以外的所有其他无明确存储器类型声明的局部变量和参数。对于已明确声明了存储器类型的函数局部变量, Cx51 编译器根据变量的存储器类型将其组合到与本模块对应的全局数据段中, 但这些变量仍属于定义它们的函数中的局部变量。函数的局部段(包含函数代码段、局部数据段和局部位段)的命名规则与函数的存储器模式有关, 如表 9-12 所示。

表 9-12 Cx51 编译器的局部段命名规则

存储器模式	局部段类型	段描述	段 名
SMALL	code	函数代码	?PR? function_name? modul_name
	data	局部数据	?DT? function_name? modul_name
	bit	局部位段	?BI? function_name? modul_name
COMPAC	code	函数代码	?PR? function_name? modul_name
	pdata	局部数据	?PD? function_name? modul_name
	bit	局部位段	?BI? function_name? modul_name
LARGE	code	函数代码	?PR? function_name? modul_name
	xdata	局部数据	?XD? function_name? modul_name
	bit	局部位段	?BI? function_name? modul_name

Cx51 编译器为局部数据段和局部位段建立一个可覆盖标志“OVERLAYABLE”, 以便让连接定位器 BL51 在对目标程序进行连接定位时作覆盖分析之用。

Cx51 编译器允许通过寄存器传递最多三个参数, 其他参数则需要采用固定的存储器

区进行传递。对于采用固定存储器区进行参数传递的数据,按以下规则生成一个局部段:

局部数据段 ?function_name? BYTE

局部位段 ?function_name? BIT

段名都表示该段的起始地址,例如,若函数 func1 需要通过固定的存储器区传递参数,则 bit 型参数将从地址“?FUNC1?BIT”开始传递,其他参数则从地址“?FUNC1?BYTE”开始传递。局部段名是全局共享的,因此它们的起始地址可被其他模块访问,从而为汇编语言程序调用 Cx51 函数提供了可能。

以上介绍的是 Cx51 编译器在对用户的 Cx51 程序进行编译时,为实现多模块程序浮动连接而采用的段名管理方法。这些段名都包括在由连接定位器 BL51 所产生的 MAP 文件中,用户可以查看,以分析自己编写的 Cx51 源程序是否合理。

9.5 Cx51 编译器的特殊支持

目前有多家半导体厂商生产具有 8051 内核的增强型 80C51 单片机,这类新型单片机增加了一些新的特殊功能,Cx51 编译器提供对如下增强型单片机的支持。

- Analog Device 公司的 Aduc8xxB2 系列(具有双数据指针和扩展堆栈空间)。
- Atmel 公司的 89x8252(具有双数据指针)。
- Dallas 公司的 80C320、C420、C530、C550 及衍生产品(具有双数据指针)。
- Dallas 公司的 80C390、5240 及衍生产品(邻接地址扩展方式、扩展堆栈空间及算术加速器)。
- Infineon 公司的 80C517、C517A、C509 及衍生产品(8 个数据指针、高速 32 位及 16 位二进制算术运算操作)。
- Philips 公司的 8xC750、C751、C752(最大 2KB 代码空间、64B 片内 RAM、无外部 XRAM 扩展、无 LCALL 和 LJMPL 指令)。
- Philips 公司的 80C51Mx 系列(允许外部扩展大容量存储器、扩展指令集)。
- Philips 和 Atmel 公司的其他产品(具有双数据指针)。

Cx51 编译器通过采用特殊函数库或附加控制命令,来生成能够支持上述新型单片机内部增强功能的目标代码。

9.5.1 Analog Device 公司的 Aduc8xxB2 系列

Analog Device 公司新推出的 Aduc8xxB2 系列,是一种基于 8051 内核的“微转换器”,它具有双数据指针,可用于快速存储器访问,利用这种双数据指针可提高 Cx51 运行库中 memcpy、memmove、memcmp、strcpy 和 strcmp 等库函数的执行速度。对于 Aduc8xxB2 系列微转换器,Cx51 编译器对于 Aduc8xxB2 微转换器系列提供了编译控制命令“MODAB2”,在对用户的 C51 源程序进行编译时采用双数据指针生成执行代码。Cx51 编译器在中断函数中至少使用一个数据指针。如果对一个中断函数进行编译时采用了“MODAB2”控制命令,则即使在中断函数中只使用了一个数据指针,也还是会将两个数据指针都保存到堆栈中。为了节省堆栈空间,编译中断函数时应该使用“NOMODAB2”

控制命令, 这时 Cx51 编译器将不用第二个数据指针。另外, 对于 Aduc8xxB2 系列微转换器提供的扩展堆栈空间, 可以通过配置文件“START_AD.A51”进行配置。

9.5.2 Atmel 89x8252 及衍生产品

Atmel 公司新推出的 AT89x8252 单片机及衍生产品具有双数据指针, 可用于快速存储器访问, 利用这种双数据指针可提高 Cx51 运行库中 memcpy、memmove、memcmp、strcpy 和 strcmp 等库函数的执行速度。对于 AT89x8252 单片机及衍生产品, Cx51 编译器对于 AT89x8252 单片机提供了编译控制命令“MODA2”, 在对用户的 C51 源程序进行编译时采用双数据指针生成执行代码。Cx51 编译器在中断函数中至少使用一个数据指针。如果对一个中断函数进行编译时采用了“MODA2”控制命令, 则即使在中断函数中只使用了一个数据指针, 也还是会将两个数据指针都保存到堆栈中。为了节省堆栈空间, 编译中断函数时应该使用“NOMODA2”控制命令, 这时 Cx51 编译器将不用第二个数据指针。

9.5.3 Dallas 80C320、C420、C520、C530

Dallas 公司新推出的 80C320、C420、C520、C530 单片机具有双数据指针, 可用于快速存储器访问, 利用这种双数据指针可提高 Cx51 运行库中 memcpy、memmove、memcmp、strcpy 和 strcmp 等库函数的执行速度。对于 Dallas 公司的 80C320、C420、C520、C530 单片机, Cx51 编译器对于 DS80C320、C420、C520、C530 单片机提供了编译控制命令“MODDP2”, 在对用户的 Cx51 源程序进行编译时采用双数据指针生成执行代码。Cx51 编译器在中断函数中至少使用一个数据指针。如果对一个中断函数进行编译时采用了“MODDP2”控制命令, 则即使在中断函数中只使用了一个数据指针, 也还是会将两个数据指针都保存到堆栈中。为了节省堆栈空间, 编译中断函数时应该使用“NOMODDP2”控制命令, 这时 Cx51 编译器将不用第二个数据指针。

DS80C420 单片机的双数据指针具有自动翻转、自动递减和自动递增的特性, Cx51 编译器提供一个增强的库函数“C51DS2A.LIB”, 其中包含了 memcpy、memmove、memcmp、strcpy 和 strcmp 等使用这些特性的库函数, 如果要使用 DS80C420 单片机的双数据指针特性时, 应将该函数库添加到用户的项目中去。

9.5.4 Dallas 80C390、C400、C5240 及衍生产品

Cx51 编译器可以完全支持 Dallas 公司新推出的 80C390、C400、C5240 单片机及衍生产品的附加 CPU 工作模式。利用这种单片机的“邻接模式”(contiguous mode), 可使程序存储器突破传统 8051 单片机 64KB 的限制, 以便于编写更大规模的程序。Cx51 编译器对于 80C390、C400、C5240 单片机提供了编译控制命令“ROM(D512K)”和“ROM(D16M)”, 在对用户的 Cx51 源程序进行编译时采用“邻接模式”生成执行代码。采用 24 位 DPTR 寻址方式(邻接模式)访问变量和常数时应使用“far”存储器类型。需要注意的是, 使用“邻接模式”时应采用扩展连接定位器 Lx51 和扩展宏汇编器 Ax51。DS80C390、C400、C5240 单片机除了扩展寻址空间之外, 还提供了对双数据指针的自动翻转、自动递减等操作特性,

Cx51 编译器提供一个增强的库函数“C51DS2T.LIB”，其中包含了 memcpy、memmove、memcmp、strcpy 和 strcmp 等使用这些特性的库函数。对于采用传统 8051 单片机非邻接模式的应用系统，如果要使用 DS80C390、C400、C5240 单片机的双数据指针特性时，应将该函数库添加到用户的项目中去。使用邻接模式时 Cx51 编译器的库函数中已经包含了对双数据指针的自动翻转、自动递减等操作的库程序。

Cx51 编译器支持 DS80C390、C400 和 C5240 单片机使用 32 位或 16 位算术操作，利用单片机内部算术加速器提高算术操作性能，这一特性可以极大提高用户程序的执行速度。采用如下方法可以保证仅在一个程序进程中使用算术加速器：

- “MODDA”进行编译；
- 对于所有其他函数采用控制命令“NOMODDA”进行编译。

另外，对于 DS80C390、C400 和 C5240 单片机提供的扩展堆栈空间，可以通过配置文件“START390.A51”进行配置。

9.5.5 Infineon 80C517、C509、C537 及衍生产品

Infineon 公司生产的 80C517、C517A 和 C509 单片机中包含有一个附加的高速 32 位和 16 位算术处理器以及多达 8 个用于存储器访问的数据指针 (DPTR)，使用算术处理器可明显提高整型数和浮点数的运算速度，使用多数据指针可以减少中断函数的栈装入，并可同时处理位于 code 和 xdata 区域的各种数据。Cx51 编译器可充分利用 80C517、C517A 和 C509 单片机中 32 位和 16 位算术操作特性，从而使用户的 Cx51 程序运行速度得到很大程度的提高。Cx51 编译器对于 Infineon80C517、C517A 和 C509 单片机提供了编译控制指令“MOD517”，在对用户的 Cx51 源程序进行编译时采用其增强功能生成执行代码。

Infineon 公司生产的 80C517、C517A 和 C509 单片机提供 8 个数据指针，利用这种多数据指针操作可提高 Cx51 运行库中 memcpy、memmove、memcmp、strcpy 和 strcmp 等库函数的执行速度，同时可以减少中断函数的栈装入。Cx51 编译器在同一时刻仅用到 8 个数据指针中的两个。为保持中断程序中堆栈的装入较低，在切换寄存器组时 Cx51 编译器将切换到两个未使用的数据指针，特殊功能寄存器 DPSEL 的内容被保存入栈，并选中一对新的数据指针，因此不再需要将数据指针送入堆栈保存。如果中断程序明确表示不切换寄存器组（即未使用关键字“using”），则数据指针必须被保存入栈（需要使用 4 个字节的堆栈空间）。为了使堆栈长度尽量小，对于中断程序及其在中断程序中调用的函数应采用控制命令“MOD517 (NODP8)”进行编译，这样对于中断函数生成的代码仅使用一个数据指针，并且仅需要 2 个字节的堆栈空间。

Cx51 编译器支持 Infineon80C517、C517A 和 C509 单片机使用 32 位或 16 位算术操作，利用单片机内部算术加速器提高算术操作性能，这一特性可以极大地提高用户程序的执行速度。采用如下方法可以保证仅在一个程序进程中使用算术加速器：

- 对于只在主程序中使用的函数，或者只在单个中断程序中使用的函数采用控制命令“MOD517”进行编译；
- 对于所有其他函数采用控制命令“MOD517 (NOAU)”进行编译。

Cx51 编译器提供支持 infineon80C517、C517A 和 C509 单片机的特殊性能的扩充库函

数, 这些库函数的原型都放在“80C517.H”中, 在头文件“REG517.H”中定义了 80C517 的所有特殊功能寄存器 (SFR) 名。下面列出了 80C517 特殊扩充库中的函数原型, 它们的功能和调用格式与标准库函数完全相同, 但对浮点数的操作使用了 80C517 的算术处理器。

```
int printf517(const char *, ...);
int sprintf517(char * s, const char *, ...);
int scanf517(char * s, const char *, ...);
int sscanf517(char * s1);
float atof517(char * s1);
float exp517(float x);
float log517(float x);
float log10517(float x);
float sqrt517(float x);
float cos517(float x);
float sin517(float x);
float tan517(float x);
float acos517(float x);
float asin517(float x);
float atan517(float x);
unsigned long strtod517(const char *string, char **ptr);
```

9.5.6 Philips 8xC750、C751、C752

Philips 公司生产的 8xC750/751/752 单片机具有 2K 字节的片内 ROM, 不能执行 LJMP 和 LCALL 指令, 因此必须使用不包含这些指令的特殊函数库“80C751.LIB”, Cx51 编译器同时还提供控制命令“ROM(SMALL)”来禁止产生 LJMP 和 LCALL 指令代码。为 8xC751 编写 Cx51 程序时必须注意以下几点。

① 由于 8xC750/751/752 单片机只有 2K 字节的片内 ROM 空间而且没有串行接口, 因此在 Cx51 程序中不能使用 ANSI 标准中诸如“printf”, “putchar”之类的函数。

② Cx51 程序中不能使用浮点数运算, 只允许使用(unsigned) char、(unsigned)int、(unsigned)long 和 bit 类型的数据。

③ Cx51 编译器必须用 ROM(SMALL)控制指令进行调用, 以抑制 LJMP 和 LCALL 指令代码的产生。

④ 需要一个特殊的启动程序: START751.A51, 它的功能与常规启动程序 STARTUP.A51 相似。对于具有初值的变量需要使用一个专门的初始化文件: INIT751.A51, 可根据需要适当修改 START751.A51 和 INIT751.A51 文件, 重新汇编后再用连接定位器 BL51 与用户程序连接到一起。

⑤ 在连接定位器 BL51 的输入模块中必须包含 8xC751.LIB。例如: BL51 myprog.obj, 8xC751.lib。

9.5.7 Philips 8xC51Mx

Cx51 编译器支持 Philips 公司生产的 8xC51Mx 单片机提供的扩展指令集和扩展寻址方式, 其存储器空间最高可达 16MB。利用其内部通用指针寄存器及相关指令可以支持一般指针操作, 用户可以采用“far”存储器类型将变量安排在扩展存储器的任意位置。另外, 对于 8xC51Mx 单片机项目文件进行编译连接时除了要采用 Cx51 编译器之外, 还必须采用扩展连接定位器 Lx51 和扩展宏汇编器 Ax51。

9.5.8 Philips 和 Atmel WM 系列的双数据指针

Philips 和 Atmel 公司推出了多种具有双数据指针的 8051 兼容衍生产品, 利用这种多数据指针可提高 Cx51 运行库中 memcpy、memmove、memcmp、strcpy 和 strcmp 等库函数的执行速度。Cx51 编译器提供了控制命令“MODP2”在对用户的 Cx51 源程序进行编译时采用双数据指针生成执行代码。Cx51 编译器在中断函数中至少使用一个数据指针。如果对一个中断函数进行编译时采用了“MODP2”控制命令, 则即使在中断函数中只使用了一个数据指针, 也还是会将两个数据指针都保存到堆栈中。为了节省堆栈空间, 编译中断函数时应该使用“NOMODP2”控制命令, 这时 Cx51 编译器将不用第二个数据指针。

9.6 用户配置文件

Cx51 编译器允许用户根据需要适当修改配置文件以满足不同的硬件环境, 用户配置文件包括启动代码文件 (STARTUP.A51、SATRT751.A51、STARTLPC.A51、START390.A51、START_MX.A51、START_AD.A51)、变量初始化文件 (INIT.A51、INIT751.A51、INIT_MX.A51、INIT_TNY.A51)、基本 I/O 函数文件 (PUTCHAR.C、GETKEY.C)、分组配置文件 (L51_BANK.A51、MX51BANK.A51、XBANKING.A51), 这些文件都以源程序的形式放在 Keil\C51\LIB 文件夹中。

Cx51 编译器的运行库中已经包含了上述配置文件的代码, 如果不作修改, 则在对用户创建的项目进行编译连接时会自动将上述代码添加到用户程序中去。如果需要根据不同硬件系统修改配置文件, 可以通过 μ Vision2 的项目窗口中将需要修改的配置文件添加到自己的文件组中, 在 μ Vision2 的编辑窗口进行修改, 然后进行总体创建就可以将修改后的配置文件代码连接到自己的程序代码中。

9.6.1 启动代码文件

启动代码的功能是在 C51 程序进入 main() 函数之前, 完成对 8051 单片机片内外 RAM 清零、开设常规堆栈以及再入函数的模拟堆栈、设置堆栈指针等工作。Keil Cx51 编译器针对不同类型的 8051 单片机提供了多个启动代码配置文件, 分别介绍如下。

1. STARTUP.A51 文件

启动代码文件中最常用的是 STARTUP.A51, 它的目标代码已经驻留在 C51 编译器的运行库中, 由连接定位器 BL51 自动加入到 Cx51 源程序代码的前面, 它将对采用数据 0 对片内数据存储器进行初始化, 并给堆栈指针赋以适当的初值。在 STARTUP.A51 文件的开头包含有一些 EQU 语句, 下面介绍这些 EQU 语句的功能, 用户可根据自己的实际需要修改这些语句的值。

(1) IDATALEN

声明 8051 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。对于具有 256 字节内部 RAM 的 8052 单片机, 可使用 100H。只有当用户程序需要在开始运行时用 0 初始化内部 RAM, 才有必要对 IDATALEN 进行赋值。如果希望内部 RAM 具有掉电保护功能, 则应将 IDATALEN 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 段的长度因不同的应用而不同, 其当前长度可在由 BL51 产生的 MAP 文件中找到。

(2) XDATASTART XDATALEN

声明需要以 0 初始化的 XDATA 存储器的首地址和长度, XDATASTART 指出 XDATA 存储器的首地址, XDATALEN 指出需要进行 0 初始化的字节数。

(3) PDATASTART PDATALEN

声明需要以 0 初始化的 PDATA 存储器的首地址和长度, PDATASTART 指出 PDATA 存储器的首地址, PDATALEN 指出需要进行 0 初始化的字节数。

(4) IBPSTACK IBPSTACKTOP

定义在 SMALL 编译模式下创建的再入函数的模拟栈区。IBPSTACK=1 时创建模拟栈并对栈指针 (变量 ?C_IBP) 进行初始化, IBPSTACK=0 (默认值) 时不创建模拟栈。IBPSTACKTOP 指出 SMALL 编译模式下再入函数的模拟栈区栈顶首地址, 默认值为 idata 区的 0xFF 地址。Cx51 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。

(5) XBPSTACK XBPSTACKTOP

定义在 LARGE 编译模式下创建的再入函数的模拟栈区。XBPSTACK=1 时创建模拟栈并对栈指针 (变量 ?C_XBP) 进行初始化, XBPSTACK=0 (默认值) 时不创建模拟栈。XBPSTACKTOP 指出栈顶首地址, 默认值为 XDATA 存储器区域的地址 0xFFFF。Cx51 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。

(6) PBPSTACK PBPSTACKTOP

定义在 COMPACT 编译模式下创建的再入函数的模拟栈区。PBPSTACK=1 时创建模拟栈并对栈指针 (变量 ?C_PBP) 进行初始化, PBPSTACKTOP 指出栈顶首地址, 默认值为 PDATA 存储器区域的地址 0xFFFF。Cx51 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。

(7) PPAGEENABLE PPAGE

在 COMPACT 编译模式下使用“页寻址”方式操作 PDATA 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减少程序代码长度。PPAGEENABLE 允许对 8051 系列单片机 P2 口进行初始化,以实现 PDATA 存储器空间的“页寻址”操作。PPAGEENABLE 和 PPAGE 必须与连接定位器 BL51 的控制指令“PDATA”一起使用,BL51 的“PDATA”指令用来指定 XDATA 存储器中 PDATA 区的首地址。例如,在 STARTUP.A51 文件中将 PPAGEENABLE 置为 1,PPAGE 置为 10H,这时 PDATA 存储器区域的首地址将为 1000H(即 10H 页),这时应在 μ Vision2 环境下应在“Project 菜单/Options for Target 选项/BL51 Locate 标签页”的“PDATA”栏中键入适当的值,并且 PDATA 的值应在 0x1000 与 0x10FF 之间。Cx51 编译器和 BL51 连接定位器都不对 PPAGE 和 PDATA 指令的正确与否进行检查,用户必须保证 PPAGE 和 PDATA 包含一个合适的值。

2. START751.A51 文件

START751.A51 是专为使用 Philips 8xC749/750/751/752 单片机而提供的启动代码文件,由于这一类单片机不能进行外部存储器扩展,因此它的启动代码文件中只需要清零片内存储器单元,它的功能与使用方法与一般启动代码文件 STARTUP.A51 类似。

3. STARTLPC.A51 文件

STARTLPC.A51 是专为使用 Philips LPC 系列单片机而提供的启动代码文件,LPC 系列单片机不能进行外部存储器扩展,因此它的启动代码文件中只需要清零片内存储器单元,此外,还可以通过启动代码文件来设置 LPC 单片机内部的配置寄存器,如选择晶体振荡器频率、调整 CPU 时钟、设置警戒电压值、端口复位状态、禁止复位引脚、看门狗定时器、设置片内 EPROM 保密状态等,STARTLPC.A51 文件的使用方法与一般启动代码文件 STARTUP.A51 类似。

4. START390.A51 文件

START390.A51 是专为使用 Dallas80C390、C400、C5240 系列单片机而提供的启动代码文件,它的功能与使用方法与一般启动代码文件 STARTUP.A51 类似,但增加了对 Dallas80C390、C400、C5240 系列单片机的特殊支持,如配置片内数据存储器 SRAM 的地址范围、分配片内 CAN 数据存储器、设置扩展堆栈寻址方式等。

5. START_AD.A51 文件

START_AD.A51 是专为使用 Analog Device 公司的 Aduc8xxB2 系列微转换器而提供的配置文件,它的功能与使用方法与一般启动代码文件 STARTUP.A51 类似,但增加了对 Aduc8xxB2 系列微转换器的特殊支持,如设置片内/片外扩展数据存储器、扩展堆栈模式、堆栈空间的大小等。

6. START_MX.A51 文件

START_MX.A51 是专为使用 Philips 公司的 80C51Mx 系列单片机而提供的配置文件,

它的功能与使用方法与一般启动代码文件 STARTUP.A51 类似,但增加了对 80C51Mx 系列单片机的特殊支持,如设置扩展寻址方式、扩展堆栈模式及堆栈空间的大小、扩展中断结构、访问片内/片外 XDATA 存储器、设置低电源电压、ALE 信号的产生方式等。

9.6.2 变量初始化文件

如果 Cx51 源程序中声明变量的时候进行了赋值,则在编译连接时需要调用变量初始化文件来对这类变量进行初始化处理。如果 Cx51 源程序中使用了“看门狗”定时器,对变量初始化处理所需时间可能超过“看门狗”的定时时间,这时应使用一个宏处理来对“看门狗”定时器进行刷新。主要的变量初始化文件为“INIT.A51”,此外还提供了“INIT_TNY.A51”文件用于不使用外部 XDATA 存储器的系统,“INIT751.A51”用于 Philips80C750/C751/C752 单片机系统,“INIT_MX.A51”用 Philips80C51Mx 单片机系统。

变量初始化文件的目标代码已经驻留在 Cx51 编译器的运行库内,当用户的 Cx51 源程序中包含具有初值的外部变量和静态变量时,连接定位器 BL51 将会自动将该目标代码加入到 Cx51 源程序的前面,它将对已明确初始化了的外部变量和静态变量进行赋值。需要时可对初始化文件进行修改并添加到用户自己的项目文件中去,重新对项目进行整体创建。

1. INIT.A51 文件

INIT.A51 文件中包含有一个“看门狗”定时器的宏定义 WATCHDOG,如果用户系统需要使用“看门狗”,并且用户程序中变量初始化的时间比“看门狗”刷新时间要长时,宏定义 WATCHDOG 中必须包含有“看门狗”刷新的代码。例如,用户系统采用了 80552 单片机中的“看门狗”定时器,则可按如下方法修改 INIT.A51 文件,然后将修改后的文件添加到自己的项目中去。在文件开始处加入以下语句:

```
T3          EQU      0FFH          ;看门狗定时器 T3 的地址
PCON        EQU      087H          ;特殊功能寄存器 PCON 的地址
WATCH_INTV  EQU      156          ;监视间隔为 2×100ms
```

将 WATCHDOG 宏定义修改成:

```
WATCHDOG    MACRO
                ORL   PC0, #10H      ;置位 PCON.4
                MOV   T3, #WATCH_INTV ;装入时间间隔
            ENDM
```

2. INIT751.A51 文件

INI751.A51 主要应用于 Philips 公司生产的 8x750/751/752 系列单片机中,如果采用 Philips 公司生产的 8x750/751/752 系列单片机,并且 Cx51 源程序中包含有明确赋值的初始化变量,则需要采用 INIT751.A51 文件进行初始化处理,它的使用方法与 INIT.A51 类似。

3. INIT_TNY.A51 文件

INI_TNY.A51 主要应用于不包括外部 XDATA 存储器的单片机应用系统中,如果采用 Philips 公司生产的 LPC 系列单片机,并且 Cx51 源程序中包含有明确赋值的初始化变量,

则需要采用 INIT_TNY.A51 文件进行初始化处理, 使用方法与 INIT.A51 文件类似。

4. INIT_MX.A51 文件

INIT_MX.A51 主要应用于 Philips 公司生产的 80C51Mx 系列单片机, 如果采用 Philips 公司生产的 80C51Mx 系列单片机, 并且 Cx51 源程序中包含有明确赋值的初始化变量, 则需要采用 INIT_MX.A51 文件进行初始化处理, 它的使用方法与 INIT.A51 文件类似。

9.6.3 基本 I/O 函数文件

基本 I/O 函数文件有两个: PUTCHAR.C 和 GETKEY.C。

(1) PUTCHAR.C 文件

PUTCHAR.C 是 Cx51 编译器运行库中 “printf”、“puts” 等函数的字符输出核心函数, 通过该函数可以将字符从串行口输出。采用了 XON/XOFF 协议进行流控制, 将 “换行” 字符 “LF” (\n) 被转换为 “回车, 换行” 字符 “CR, LF” (\r\n), 这在很多情况下是需要的。用户可以按自己的实际需要来改写该文件 (例如作 LCD 或 LED 显示), 再与自己的程序模块连接起来。

(2) GETKEY.C 文件

这是 Cx51 编译器运行库中 “getchar”、“scanf” 等函数的字符输入核心程序, 字符通过串行口输入, 不作数据转换。用户可根据需要修改该程序 (例如矩阵键盘输入), 重新编译后再与自己的程序模块连接起来。

9.6.4 分组配置文件

Cx51 编译器支持用户程序进行代码分组设计, 对于传统 8051 单片机可将 ROM 存储器扩展到最大 4MB, 对于新一代扩展型单片机如 Philips80C51MX、DS80C390 等, 可将它们的存储器范围扩展到最大 16MB。为此提供了三个分组配置文件: L51_BANK.A51、MX51BANK.A51、XBANKING.A51。

1. L51_BANK.A51 文件

对于传统 8051 单片机分组代码设计, 应采用 L51_BANK.A51 文件进行分组配置, 同时在对由 Cx51 编译器生成的浮动代码目标文件进行连接定位时, 应采用 BL51 连接定位器。

2. MX51BANK.A51 文件

MX51BANK.A51 是专门用于 Philips80C51Mx 单片机的分组代码配置文件, 使用 Philips80C51Mx 单片机进行分组代码设计时, 按需要配置该文件, 并且在对由 Cx51 编译器生成的浮动代码目标文件进行连接定位时, 应该使用扩展连接定位器 Lx51, 而不是 BL51。

3. XBANKING.A51 文件

XBANKING.A51 提供对 far 和 far const 存储器类型变量的支持, 扩展连接定位器 Lx51 通过 far 和 far const 存储器类型来访问扩展的 HDATA 和 HCONST 存储器地址空间。Cx51 编译器通过 3 个字节的一般指针访问这类存储器区域, 用 far 存储器类型定义的变量存放

在 HDATA 存储类中, 用 far const 存储器类型定义的变量存放在 HCONST 存储类中。Lx51 连接定位器允许在 16MB 的物理 code 空间和 16MB 的物理 xdata 空间中定位 HCONST 类 HDATA 类。如果希望对于传统 8051 单片机使用 far 存储器, 则必须采用“VARBANKING”编译控制命令对 C51 源程序进行编译。

对于新一代的 80C51 单片机, 可以采用“far”和“far const”类型在大容量扩展存储器 codc/xdata 空间中定义变量。如果所使用的新型 80C51 单片机提供 24 位 DPTR 数据指针, 用户可以修改 XBANKING.A51 文件中如下符号来适应自己的需要。

?C?XPAGE1SFR 包含 DPTR 第 16~23 位值的 DPTR 页寄存器地址。
 ?C?XPAGE1RST 将?C?XPAGE1SFR 定位到地址范围 X:0 的复位值。采用编译控制命令“VARBANKING(1)”时 Cx51 编译器在中断函数开始处将 DPTR 页寄存器的内容保存入栈, 然后将 DPTR 页寄存器的值设置为?C?XPAGE1RST 的值。

采用“far”存储器类型允许对特殊存储器空间(如代码分组 ROM 中的字符串或 EEPROM 存储器)进行寻址, 如同对标准 8051 存储器空间寻址一样。用户可以通过修改 XBANKING.A51 文件中如下一些子程序来满足自己的要求:

?C?CLDXPTR, ?C?CSTXPTR 在扩展存储器中装入/存储一个字节(char)
 ?C?ILDXPTR, ?C?ISTXPTR 在扩展存储器中装入/存储一个字(int)
 ?C?PLDXPTR, ?C?PSTXPTR 在扩展存储器中装入/存储一个 3 字节的指针
 ?C?LLDXPTR, ?C?LSTXPTR 在扩展存储器中装入/存储一个双字(long)

上述每个子程序都利用工作寄存器 R1/R2/R3 得到一个 3 字节指针表示的存储器地址参数, R3 的内容为存储器类型, 对于传统 8051 单片机, Cx51 编译器使用如表 9-13 所示存储器类型值。

表 9-13 R3 的值所代表的存储器类型

R3 的值	存储器类型	存 储 类	地 址 范 围
0x00	data/idata	DATA/IDATA	1:0x00~1:0xff
0x01	Xdata	XDATA	X:0x0000~0xffff
0x020~x7f	Far	HDATA	X:0x010000~X:0x7e0000
0x800~xfd	Fra const	HCONST	C:0x800000~C:0xfd0000
0xfe	pdata	XDATA	XDATA 空间 256 字节为 1 页
0xff	code	CODE/CONST	C:0x0000~C:0xffff

R3 的值 0x00、0x01、0xfe、0xff 已经在运行库中进行了处理, 只有 0x02~0xfe 被传送到上述子程序中。扩展宏汇编器 Ax51 提供操作符 MBYTE, 用以计算需要传送到 XPTR 存取函数的 R3 内容。下面是一个使用 Ax51 宏汇编器调用 XPTR 存取函数的例子:

```
MOV R1, #LOW (variable)      ; 给出变量的 LSB 地址字节
MOV R1, #HIGH (variable)     ; 给出变量的 MSB 地址字节
MOV R1, #MBYTE (variable)    ; 给出变量的存储类型
CALL ?C?CLDXPTR              ; 将字节变量装入累加器 A
```

如果用户希望采用新型 8051 单片机进行大容量存储器扩展并使用 far 或 far const 类型的变量, 则应根据具体要求适当修改 XBANKING.A51 文件并添加到自己的项目中去, 然后对进行整个项目进行编译连接生成最终目标代码。

9.7 与汇编语言程序的接口

Cx51 编译器能对 Cx51 源程序进行高效率的编译, 生成高效简洁形式的代码, 在绝大多数场合采用 C 语言编程即可完成预期的任务。尽管如此, 有时仍需要采用一定的汇编语言编程, 例如, 对于某些特殊 I/O 接口地址的处理、中断向量地址的安排、提高程序代码的执行速度等。为此 Cx51 编译器提供了与汇编语言程序的接口规则, 按此规则可以很方便地实现 C 语言程序与汇编语言程序的相互调用。实际上 C 语言程序与汇编语言程序的相互调用也可视为函数的调用, 只不过此时函数是采用不同语言编写的而已。

C 语言程序函数和汇编语言函数在相互调用时, 可利用 8051 单片机的工作寄存器最多传递 3 个参数, 如表 9-14 所示。

表 9-14 参数传递的工作寄存器选择

传递的参数	char、1 字节指针	int、2 字节指针	long、float	一般指针
第一个参数	R7	R6 (高字节), R7 (低字节)	R4~R7	R3 (存储类型), R2 (高字节), R1 (低字节)
第二个参数	R5	R4 (高字节), R5 (低字节)	R4~R7	R3 (存储类型), R2 (高字节), R1 (低字节)
第三个参数	R3	R2 (高字节), R3 (低字节)	无	R3 (存储类型), R2 (高字节), R1 (低字节)

如果在调用时参数无寄存器可用, 或是采用了编译控制命令“NOREGPARDS”, 则通过固定的存储器区域来传递参数, 该存储器区域称为参数传递段, 其地址空间取决于编译时所选择的存储器模式。下面是几个说明参数传递规则的例子。

func1(int a) “a”是第一个 int 型参数, 在 R6, R7 中传递。

func2(int b,int c,int *d) “b”在 R6, R7 中传递, “c”在 R4, R5 中传递, “*d”在 R1, R2, R3 中传递。

func3(long e, long f) “e”在 R4, R5, R6, R7 中传递, “f”不能通过寄存器传递, 而只能在参数传递段中传递。

func4(float g, char h) “g”在 R4, R5, R6, R7 中传递, “h”不能通过寄存器传递, 而只能在参数传递段中传递。

当 C 语言程序与汇编语言程序需要相互调用, 并且参数的传递发生在参数传递段时, 如果传递的参数是 char、int、long 和 float 类型的数据, 则参数传递段的首地址由“?functionname? BYTE”的公共符号 (PUBLIC) 确定, 如果传递的参数是 bit 类型的数据, 参数传递段的首地址由“?functionname? BIT”的公共符号 (PUBLIC) 确定。所有被传递的参数按顺序存放在以首地址开始递增的存储器区域内。参数传递段的存储器空间取决于

所采用的编译模式，在 SMALL 模式下参数传递段位于片内 RAM 空间，在 COMPACT 和 LARGE 模式下参数传递段位于外部 RAM 空间。

函数的返回值被放入 8051 单片机的寄存器内，如表 9-15 所示。

表 9-15 函数返回值所占用工作寄存器

返回值类型	寄存器	说明
Bit	进位位 CY	返回值在进位标志 CY 中
(unsigned)char	R7	返回值在寄存器 R7 中
(unsigned)int	R6, R7	返回值高位在 R6 中，低位在 R7 中
(unsigned)long	R4~R7	返回值高位在 R4 中，低位在 R7 中
float	R4~R7	32 位 IEEE 格式，指数和符号位在 R7 中
一般指针	R3, R2, R1	R3 放存储器类型，高位在 R2，低位在 R1

在汇编语言子程序中，当前选择的工作寄存器组以及特殊功能寄存器 ACC、B、DPTR 和 PSW 的值都可能改变，当从汇编语言程序调用 C 语言函数时必须无条件地假定这些寄存器的内容已被破坏。

如果在连接定位时采用了覆盖过程，则每个汇编语言子程序都将包含一个单独的程序段。这一点是必要的，因为在 BL51 连接定位器的覆盖分析中，函数之间的相互参考是通过子程序各自的段基准进行计算的。如果注意下面两点，汇编语言子程序的数据区也可以包含在覆盖分析中：

- 所有的段名都必须以 Cx51 编译器所规定的方法来建立；
- 每个具有局部变量的汇编语言函数都必须指定自己的局部数据段，这个局部数据段可以用来为其他函数访问作参数传递用，并且参数的传递要按顺序进行。

在汇编语言函数程序中，对于 char、int、long 和 float 类型的数据，局部数据段应以 PUBLIC 符号?_functionname? BYTE 作为首地址，并在数据段中先按被传递参数的顺序定义若干字节，然后再定义其他局部变量数据字节。例如，在 SMALL 编译模式下该数据段应如下建立：

```
RSEG ?DT?functionname? modulname    ; 定义局部数据段名
?_functionname? BYTE:                ; 定义数据段首地址
    charVAL    DS    1                ; 按参数的传递顺序定义字节
    intVAL     DS    2
    longVAL    DS    4
    ...        ; 定义其他局部变量字节
```

对于 bit 类型的数据，局部数据段应以 PUBLIC 符号：?_functionname? BIT 作为首地址，并按被传递参数的顺序先定义若干个位，然后再定义其他局部变量位。例如：

```
RSEG ?BI?functionname? modulname    ; 定义局部数据段名
?_functionname? BIT:                ; 定义数据段首地址
    bitVAL1    DBIT    1                ; 按参数的传递顺序定义位
    bitVAL2    DBIT    1
```

... ; 定义其他局部变量位

这样定义的局部数据段即可为其他函数的访问作参数传递之用,所有参数都将按顺序逐个传递。

通过下面例子可以更清楚地了解参数的传递过程。

例 9.18: SMALL 编译模式下 C 语言函数调用汇编语言函数。

C51 程序文件为 C_CALL.C,文件中定义了两个函数:主调用函数 void C_call()和被调用的外部函数 extern int afunc(int v_a,char v_b,bit v_c,long v_d,bit v_e),该函数是在另一个模块文件 AFUNC.A51 中采用汇编语言编写的。函数中有 5 个参数,函数调用时最多可利用 8051 单片机的工作寄存器传递 3 个参数,因此只有参数 v_a 在寄存器 R6, R7 中传递(高位在 R6,低位在 R7)、参数 v_b 在 R5 中传递。而参数 v_c、v_d 和 v_e 将在参数传递段中传递。程序编译时采用了 SMALL 模式,参数传递段将位于 8051 单片机的内部数据存储器 DATA 区。另外,函数 afunc()是 int 类型的函数,所以函数 afunc()的返回值在工作寄存器 R6 (高位)、R7 (低位)中。

```

stmt level  source
1          #pragma code small
2          extern int afunc(int v_a,char v_b,bit v_c,long v_d,bit v_e);
3          void C_call(){
4      1      int v_a;char v_b;bit v_c;long v_d;bit v_e;
5      1      int A_ret;
6      1      A_ret=afunc(v_a, v_b, v_c, v_d, v_e);
7      1      }

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION C_call (BEGIN)

; SOURCE LINE # 3
; SOURCE LINE # 6
0000 A200      R      MOV      C,v_c
0002 9200      E      MOV      ?_afunc?BIT,C
0004 850000    E      MOV      ?_afunc?BYTE+06H,v_d+03H
0007 850000    E      MOV      ?_afunc?BYTE+05H,v_d+02H
000A 850000    E      MOV      ?_afunc?BYTE+04H,v_d+01H
000D 850000    E      MOV      ?_afunc?BYTE+03H,v_d
0010 A200      R      MOV      C,v_e
0012 9200      E      MOV      ?_afunc?BIT+01H,C
0014 AD00      R      MOV      R5,v_b
0016 AF00      R      MOV      R7,v_a+01H
0018 AE00      R      MOV      R6,v_a
001A 120000    E      LCALL    _afunc
001D 8E00      R      MOV      A_ret,R6
001F 8F00      R      MOV      A_ret+01H,R7

; SOURCE LINE # 7

```

```
0021 22                RET
                ; FUNCTION C_call (END)
```

在被调用的汇编语言程序函数 `afunc()` 中, 将从 C 语言程序函数 `C_call()` 传递过来的参数: `int v_a`、`char v_b`、`bit v_c`、`long v_d` 和 `bit v_e`, 分别放入局部变量 `a`、`b`、`c`、`d` 和 `e` 中。因此该汇编语言程序函数是包含有局部变量的。又由于 C 语言程序模块采用了 **SMALL** 编译模式, 参数传递将在内部数据存储器 **DATA** 区域进行。在汇编语言程序函数中必须按 **Cx51** 编译器关于 **SMALL** 模式下段名的规定建立相应的局部数据段, 即对于需要利用工作寄存器进行参数传递的函数, 函数名前面要加一个下划线, 还需要给出正确的参数传递段地址。

汇编语言程序模块文件 **AFUNC.A51** 列表如下:

```
; AFUNC.A51 generated from: AFUNC.C
NAME                AFUNC
?PR?_afunc?AFUNC    SEGMENT CODE                ; 定义程序代码段
?DT?_afunc?AFUNC    SEGMENT DATA OVERLAYABLE    ; 定义可覆盖局部数据段
?BI?_afunc?AFUNC    SEGMENT BIT OVERLAYABLE      ; 定义可覆盖局部位段
                    PUBLIC ?_afunc?BIT          ; 公共符号定义
                    PUBLIC ?_afunc?BYTE
                    PUBLIC _afunc

                    RSEG ?DT?_afunc?AFUNC        ; 可覆盖局部数据段
?_afunc?BYTE:                ; 起始地址
    v_a?040:    DS      2                ; 定义传递参数字节
    v_b?041:    DS      1
    v_d?043:    DS      4
    ORG 7
    a?045:    DS      2                ; 定义其他局部变量
    b?046:    DS      1
    d?048:    DS      4
    retval?050: DS      2                ; 返回值

                    RSEG ?BI?_afunc?AFUNC        ; 可覆盖局部位段
?_afunc?BIT:                ; 起始地址
    v_c?042:    DBIT    1                ; 定义传递数据位
    v_e?044:    DBIT    1
    ORG 2
    c?047:    DBIT    1                ; 定义其他局部变量位
    e?049:    DBIT    1

; #pragma src(AFUNC.A51) small
; int afunc(int v_a,char v_b,bit v_c,long v_d,bit v_e) {

                    RSEG ?PR?_afunc?AFUNC        ; 程序代码段
_afunc:                ; 起始地址
```

```

        USING      0

                                ; SOURCE LINE # 2
;---- Variable 'v_b?041' assigned to Register 'R5' ----
;---- Variable 'v_a?040' assigned to Register 'R6/R7' ----
;   int a;char b;bit c;long d;bit e;
;   int retval;
;   a=v_a;b=v_b;c=v_c;d=v_d;e=v_e;

                                ; SOURCE LINE # 5
        MOV        a?045,R6            ;a=v_a
        MOV        a?045+01H,R7
        MOV        b?046,R5            ;b=v_b
        MOV        C,v_c?04            ;c=v_c2
        MOV        c?047,C            ;d=v_d
        MOV        d?048+03H,v_d?043+03H
        MOV        d?048+02H,v_d?043+02H
        MOV        d?048+01H,v_d?043+01H
        MOV        d?048,v_d?043
        MOV        C,v_e?044            ;e=v_e
        MOV        e?049,C
;   return(retval);

                                ; SOURCE LINE # 6
        MOV        R6,retval?050        ;函数返回值高位
        MOV        R7,retval?050+01H    ;函数返回值低位
;   )

                                ; SOURCE LINE # 7
?C0001:
        RET
; END OF _afunc
        END

```

Cx51 编译器提供了一个十分有用的编译控制指令“SRC”，在编写汇编语言程序函数时，可以先按需要用 C 语言编写相应的函数，对该函数单独采用编译控制指令 SRC 进行编译，编译完成后将产生一个汇编语言源程序。然后再对这样产生的汇编语言子程序作一些必要调整和修改，即可很方便地写出汇编语言函数子程序，而编写过程中各种段的安排全部由 Cx51 编译器自动完成，从而大大提高汇编语言程序的编写效率。这种方法对于编写汇编语言中断服务函数或是某些具有特殊要求的汇编语言子程序特别有用。上面例子中的汇编语言程序函数 AFUNC()就是采用这种方法编写的。下面给出相应的 C 语言程序模块文件 AFUNC.C:

```

#pragma src(AFUNC.A51) small
int afunc(int v_a,char v_b,bit v_c,long v_d,bit v_e) {
    int a;char b;bit c;long d;bit e;
    int retval;

```

```

a=v_a;b=v_b;c=v_c;d=v_d;e=v_e;
return(retval);
}

```

需要注意的是,在对模块文件 AFUNC.C 编译时,必须采用与模块文件 C_CALL.C 相同的编译模式相同,当调用有参函数时这一点是十分重要的。如果两个文件采用不同的编译模式,将导致它们采用不同的存储器区域作为参数传递段空间,从而不能正确地进行参数传递。用户在编写自己的实际应用程序时,还可以根据需要进行调整,如规定中断服务程序的入口向量地址或某个特殊的 I/O 设备地址、定义 LED 显示段码等。

如果不希望在汇编语言程序中通过工作寄存器来传递参数,可以采用 Cx51 编译器提供的控制命令“NOREGPARMs”,下面给出了对同一个 Cx51 源程序文件 AFUNC.C 采用“NOREGPARMs”命令编译后输出的汇编语言模块文件 AFUNC.A51,与前面的汇编语言模块文件比较可以看到,现在这个文件中对段名转换时函数名前面没有了下划线,同时参数传递也不再通过工作寄存器而是全部通过固定的存储器区域进行的。

NAME	AFUNC		
?PR?afunc?AFUNC	SEGMENT CODE		;定义程序代码段
?DT?afunc?AFUNC	SEGMENT DATA OVERLAYABLE		;定义可覆盖局部数据段
?BI?afunc?AFUNC	SEGMENT BIT OVERLAYABLE		;定义可覆盖局部位段
	PUBLIC ?afunc?BIT		;公共符号定义
	PUBLIC ?afunc?BYTE		
	PUBLIC afunc		
	RSEG ?DT?afunc?AFUNC		;可覆盖局部数据段
?afunc?BYTE:			;起始地址
v_a?040:	DS 2		;定义传递参数字节
v_b?041:	DS 1		
v_d?043:	DS 4		
ORG 7			
a?045:	DS 2		;定义其他局部变量
b?046:	DS 1		
d?048:	DS 4		
retval?050:	DS 2		;返回值
	RSEG ?BI?afunc?AFUNC		;可覆盖局部位段
?afunc?BIT:;起始地址			
v_c?042:	DBIT 1		;定义传递数据位
v_e?044:	DBIT 1		
ORG 2			
c?047:	DBIT 1		;定义其他局部变量位
e?049:	DBIT 1		
; #pragma src(AFUNC.A51) small noregparms			
; int afunc(int v_a,char v_b,bit v_c,long v_d,bit v_e) {			

```

                                RSEG ?PR?afunc?AFUNC           ;程序代码段
afunc:                                ;起始地址

                                USING 0

                                ; SOURCE LINE # 2
;   int a;char b;bit c;long d;bit e;
;   int retval;
;   a=v_a;b=v_b;c=v_c;d=v_d;e=v_e;

                                ; SOURCE LINE # 5
                                MOV     a?045, v_a?040
                                MOV     a?045+01H, v_a?040+01H
                                MOV     b?046, v_b?041
                                MOV     C, v_c?042
                                MOV     c?047,C
                                MOV     d?048+03H,v_d?043+03H
                                MOV     d?048+02H,v_d?043+02H
                                MOV     d?048+01H,v_d?043+01H
                                MOV     d?048,v_d?043
                                MOV     C,v_e?044
                                MOV     e?049,C
;   return(retval);

                                ; SOURCE LINE # 6
                                MOV     R6,retval?050           ;函数返回值高位
                                MOV     R7,retval?050+01H       ;函数返回值低位
; }

                                ; SOURCE LINE # 7
?C0001:

                                RET
; END OF afunc

                                END

```

下面给出一个从汇编语言程序调用 C 语言函数的例子，被调用的外部 C 语言函数为 long 类型，因此函数的返回值在 R4（高位）~R7（低位）中。

例 9.19：汇编语言程序调用 C 语言函数。

产生汇编语言程序的 C 语言源程序模块文件 A_CALL.C 列表如下：

```

#pragma src(A_call.a51) small
extern long cfunc(int v_a,char v_b,bit v_c,long v_d,bit v_e);
void A_call(){
    int v_a;char v_b;bit v_c;long v_d;bit v_e;
    long A_ret;
    A_ret=cfunc(v_a, v_b, v_c, v_d, v_e);
}

```


所产生的汇编语言程序模块文件 A_call.a51 列表如下:

```

NAME          A_CALL

?PR?A_call?A_CALL    SEGMENT CODE          ;定义程序代码段
?DT?A_call?A_CALL    SEGMENT DATA OVERLAYABLE ;定义可覆盖局部数据段
?BI?A_call?A_CALL    SEGMENT BIT OVERLAYABLE  ;定义可覆盖局部位段
                    EXTERN CODE (_cfunc)      ;外部函数代码段名
                    EXTERN DATA (?_cfunc?BYTE) ;外部函数数据段名
                    EXTERN BIT (?_cfunc?BIT)   ;外部函数位数据段名
                    PUBLIC A_call             ;公共符号定义

                    RSEG ?DT?A_call?A_CALL    ;可覆盖局部数据段
?A_call?BYTE:        ;起始地址
    v_a?040:         DS 2                    ;定义传递参数字节
    v_b?041:         DS 1
    v_d?043:         DS 4
    A_ret?045:       DS 4                    ;定义其他局部变量

                    RSEG ?BI?A_call?A_CALL    ;可覆盖局部位段
?A_call?BIT:         ;起始地址
    v_c?042:         DBIT 1                  ;定义传递数据位
    v_e?044:         DBIT 1

; #pragma src(A_call.a51) small
; extern long cfunc(int v_a,char v_b,bit v_c,long v_d,bit v_e);
; void A_call(){

                    RSEG ?PR?A_call?A_CALL    ;程序代码段
A_call:              ;程序起始地址
                    USING 0                    ;工作寄存器组定义

                                ; SOURCE LINE # 3
;   int v_a;char v_b;bit v_c;long v_d;bit v_e;
;   long A_ret;
;   A_ret=cfunc(v_a, v_b, v_c, v_d, v_e);

                                ; SOURCE LINE # 6
    MOV     C,v_c?042            ;传递 bit 参数 v_c
    MOV     ?_cfunc?BIT,C
    MOV     ?_cfunc?BYTE+06H,v_d?043+03H ;传递 long 型参数 v_d
    MOV     ?_cfunc?BYTE+05H,v_d?043+02H
    MOV     ?_cfunc?BYTE+04H,v_d?043+01H
    MOV     ?_cfunc?BYTE+03H,v_d?043
    MOV     C,v_e?044            ;传递 bit 参数 v_e
    MOV     ?_cfunc?BIT+01H,C

```

```

MOV      R5,v_b?041                ;传递 char 型参数 v_b
MOV      R7,v_a?040+01H            ;传递 int 型参数 v_a
MOV      R6,v_a?040
LCALL    _cfunc                    ;调用 C 语言函数 cfunc()
MOV      A_ret?045+03H,R7          ;保存函数调用的返回值
MOV      A_ret?045+02H,R6
MOV      A_ret?045+01H,R5
MOV      A_ret?045,R4

;  )

; SOURCE LINE # 7

RET

; END OF A_call

END

```

例 9.20: Cx51 程序与汇编语言相互调用。

该例在 Cx51 程序文件 C2ASM2C.C 中定义了两个函数: A_FUNC(int *)和 c_func(int *iptr), A_FUNC(int *)是另一个文件模块中的汇编语言函数; c_func(int *iptr)是 Cx51 函数。在主函数 main()中先对一个 int 型变量“min”赋初值 0x2211, 然后调用汇编语言函数 A_FUNC()并将该变量的地址作为传递的参数。汇编语言函数 A_FUNC()将一个新的数据值 0x1122 放到该地址中, 然后又调用 C 语言函数 c_func(), 并将该变量的地址作为传递的参数。Cx51 函数 c_func()又将一个新的数据值 0x9988 放到该地址中, 并返回到汇编语言函数 A_FUNC()中, 最后再从汇编语言函数返回到主函数 main(), 这时变量 min 的内容已经变成 0x9988, 并被重新赋值为 0x6688。在这个例子中主调函数和被调函数的参数都是指针类型, 因此传递的都是地址, 并且在数调用时利用工作寄存器进行参数传递。在主函数中调用 A_FUNC()函数时, 将参数“&min”按“一般指针”的存储格式进行传递, 即先传递变量 min 的存储器类型(IDATA 为 0x00)到 R3 中, 再传递 min 的地址到 R2、R1 中。汇编语言函数 A_FUNC()中, 先对传递过来的地址单元装入 0x1122 数据值, 然后再调用 Cx51 函数 C_FUNC(), 并且参数仍采用地址传递方式。

本例中还给出了采用编译控制命令“SRC”生成汇编语言函数的 Cx51 源程序 A_FUNC.C, 用户可以先将该文件添加到新建的项目中并进行单独编译, 生成汇编语言函数文件 a_func.a51, 再将该汇编语言函数文件 a_func.a51 和 Cx51 程序文件 C2ASM2C.C 一起添加到项目中, 并从项目中删除 Cx51 源程序文件 A_FUNC.C, 然后对项目进行整体创建 (Build Target) 生成目标代码, 最后进入 Debug 状态对目标代码进行调试, 可以清楚地看到参数传递过程以及变量值的改变情况。

下面是产生汇编语言函数 A_FUNC.A51 的 Cx51 源程序文件 A_FUNC.C:

```

#pragma SRC (a_func.a51)
extern void C_FUNC(int *Ptr); /* 定义外部 C51 函数 */
void A_FUNC(int *Ptr) {
    *Ptr = 0x1122;
    C_FUNC(Ptr);
}

```

下面是由以上文件生成的汇编语言函数文件 A_FUNC.A51:

```

NAME          A_FUNC
               ?PR?_A_FUNC?A_FUNC  SEGMENT CODE
               EXTERN  CODE  (_C_FUNC)
               FXTERN  CODE  (?C?ISTPTR)
               PUBLIC  _A_FUNC
; #pragma      SRC(a_func.a51)
; extern void C_FUNC(int *Ptr); /* 定义外部 C51 函数 */
; void A_FUNC(int *Ptr) {

               RSEG  ?PR?_A_FUNC?A_FUNC
_A_FUNC:
               USING  0
                                   ; SOURCE LINE # 4
;---- Variable 'Ptr?040' assigned to Register 'R1/R2/R3' ----
;   *Ptr = 0x1122;
                                   ; SOURCE LINE # 5
               MOV     A,#011H
               MOV     B,#022H
               LCALL   ?C?ISTPTR
;   C_FUNC(Ptr);
                                   ; SOURCE LINE # 6
               LCALL   _C_FUNC
; }
                                   ; SOURCE LINE # 7
               RET
; END OF _A_FUNC
END

```

下面是 Cx51 源程序文件 C2ASM2C.C:

```

extern void A_FUNC(int *); /* 定义外部汇编语言函数 */

void c_func(int *iptr) { /* 定义 C51 函数 */
    *iptr = 0x9988;
}

void main (void) { /* 主函数 */
    idata int min = 0x2211; /* 定义变量 min 并赋初值 */
    while (1) {
        A_FUNC(&min); /* 调用汇编语言函数 */
        min = 0x6655; /* 重新对变量 min 初值 */
    }
}

```

9.8 与 PL/M51 程序的接口

Cx51 编译器允许在 Cx51 源程序中采用关键字“alien”来与 PL/M51 程序进行直接和简单的接口。采用关键字“alien”之后，Cx51 编译器在对 Cx51 源程序进行编译时将按 PL/M51 规定的参数传递方式工作，也就是说可用关键字“alien”将函数或过程定义为 PL/M51 的结构。关键字“alien”可用于外部函数或公共函数，并且可在任意存储器模式下使用，这样就可以很方便地将已有的 PL/M51 程序加入到 Cx51 源程序中。需要注意的是“alien”不能用于再入函数。另外，所使用的函数应始终包含有标准的参数个数，因此 C 语言中关于函数参数的个数未知时的缩写符号（即 func(...)）将不被接受，并会在编译时产生错误信息。

下面是一个与 PL/M51 程序接口的例子：

```
extern alien char plm_function(unsigned char, unsigned int);
alien char c_function(unsigned char x, unsigned char y) {
    return(x*y);
}
```

PL/M 兼容函数必须定义关键字“alien”，只有如此，PL/M 函数的参数传递及参数返回才能在 Cx51 编译器中被考虑。

9.9 绝对地址访问

在进行 8051 单片机应用系统程序设计时，用户十分关心如何直接操作系统的各个存储器地址空间。Cx51 程序经过编译之后产生的目标代码具有浮动地址，其绝对地址必须经过 BL51 连接定位后才能确定。为了能够在 Cx51 程序中直接对任意指定的存储器地址进行操作，可以采用扩展关键字“_at_”、指针、预定义宏以及连接定位控制命令，分别介绍如下。

9.9.1 采用扩展关键字“_at_”或指针定义变量的绝对地址

在 Cx51 源程序中定义变量时，可以利用 Cx51 编译器提供的扩展关键字“_at_”来对指定变量的存储器空间绝对地址，一般格式如下：

[存储器类型] 数据类型 标识符 _at_ 地址常数

其中，“存储器类型”为 idata、data、xdata 等 Cx51 编译器能够识别的所有类型，如果省略该选项，则按编译模式 LARGE、COMPACT 或 SMALL 规定的默认存储器类型确定变量的存储器空间；“数据类型”除了可用 int、long、float 等基本类型外，还可以采用数组、结构等复杂数据类型；标识符为要定义的变量名；地址常数规定了变量的绝对地址，它必须位于有效存储器空间之内。

例 9.21: 采用关键字“at”进行变量的绝对地址定位。

```
struct link {
    struct link idata *next;
    char code *test;
};
idata struct link list _at_ 0x40; /* 结构变量 list 定位于 idata 空间地址 0x40 */
xdata char text[256] _at_ 0xE000; /* 数组 array 定位于 xdata 空间地址 0xE000 */
xdata int i1 _at_ 0x8000;          /* int 变量 i1 定位于 xdata 空间地址 0x8000 */
```

利用扩展关键字“_at_”定义的变量称为“绝对变量”，对该变量的操作就是对指定存储器空间绝对地址的直接操作，因此不能对“绝对变量”进行初始化，对于函数和位（bit）类型变量不能采用这种方法进行绝对地址定位。采用关键字“_at_”所定义的绝对变量必须是全局变量，在函数内部不能采用“_at_”关键字指定局部变量的绝对地址。另外，在 XDATA 空间定义全局变量的绝对地址时，还可以在变量前面加一个关键字“volatile”，这样对该变量的访问就不会被 Cx51 编译器优化掉。

利用基于存储器的指针也可以指定变量的存储器绝对地址，其方法是先定义一个基于存储器的指针变量，然后对该变量赋以存储器绝对地址值，下面通过一个例子来说明。

例 9.22: 利用基于存储器的指针进行变量的绝对地址定位。

```
char xdata temp _at_ 0x4000; /* 定义全局变量 temp, 地址为 XDATA 空间 0x4000 */
void main( void ) {
    char xdata *xdp;          /* 定义一个指向 XDATA 存储器空间的指针 */
    char data *dp;            /* 定义一个指向 DATA 存储器空间的指针 */
    xdp = 0x2000;              /* XDATA 指针赋值, 指向 XDATA 存储器地址 0002h */
    temp = *xdp;               /* 读取 XDATA 空间地址 0x2000 的内容送往 0x4000 单元 */
    *xdp = 0xAA;               /* 将数据 0xAA 送往 XDATA 空间 0x2000 地址单元 */
    dp = 0x30;                 /* DATA 指针赋值, 指向 DATA 存储器地址 30H */
    *dp = 0xBB;                /* 将数据 0xBB 送往指定的 DATA 空间地址 */
}
```

9.9.2 采用预定义宏指定变量的绝对地址

Cx51 编译器的运行库中提供了如下的一套预定义宏：

CBYTE	CWORD	FARRAY
DBYTE	DWORD	FCARRAY
PBYTE	PWORD	FCVAR
XBYTE	XWORD	FVAR

这些宏定义包含在头文件“ABSACC.H”中，在 Cx51 源程序中可以利用这些宏来指定变量的绝对地址，例如：

```
#include <ABSACC.H>
char c_var;
int i_var;
```

```
XBYTE[0x12]=c_var; /* 向 XDATA 存储器地址 0012H 写入数据 c_var */
i_var=XWORD[0x100]; /* 从 XDATA 存储器地址 0200H 中读取数据并赋值给 i_var */
```

上面第二条赋值语句中采用的是 XWORD[0x100]，它是对地址“2*0x100”进行操作，该语句的意义是将字节地址 0x200 和 0x201 的内容取出来并赋值给 int 型变量 i_var，注意不要将 XWORD 与 XBYTE 混淆。如果将这条语句改成：

```
i_var=XWORD[0x100/2];
```

这样读取的就是 0x100 和 0x101 地址单元中的内容了。用户可以充分利用 C51 运行库中提供的预定义宏来进行绝对地址的直接操作。例如，可以采用如下方法定义一个 D/A 转换接口地址，每向该地址写入一个数据即可完成一次 D/A 转换：

```
#include <ABSACC.H>
#define DAC0832 XBYTE[0x7fff] /* 定义 DAC0832 端口地址 */
DAC0832=0x80; /* 启动一次 D/A 转换 */
```

9.9.3 采用连接定位控制命令指定变量的绝对地址

BL51 和 Lx51 连接定位器提供了多个连接定位控制命令，在对用户程序进行连接的时候，可以通过连接定位控制命令指定变量的绝对地址。下面通过一个例子来加以说明。

例 9.23：通过连接定位控制命令指定下面程序模块文件 test.c 中变量的绝对地址。

文件 test.c 代码如下：

```
char xdata temp _at_ 0x4000;
struct alarm_st {
    unsigned int alarm_number;
    unsigned char enable_flag;
    unsigned int time_delay;
    unsigned char status;
};
xdata struct alarm_st alarm_control;

void func(char val) {
    char cc;
    cc = val++;
}

void main( void ) {
    int xdata x=0x66;
    int xdata y=0x88;
    char aa;
    char bb=0x55;
    alarm_control.status = 0x22;
    alarm_control.alarm_number=0x33;
```

```
func(0x01);
}
```

如果希望将明确指定了 XDATA 存储器类型的全局变量定位在 6000H 开始的地址空间, 将 main 函数内 XDATA 存储器类型的局部变量定位在 8000H 开始的地址空间, 可以在 μVision2 环境下“Project 菜单/Options for Target 选项/BL51 Locate 标签页/Xdata 栏”内键入如下段名和起始地址, 关于段名规则可参考 9.3.2 节, 注意, 地址应按从小到大的顺序给定:

```
?XD?test(6000h),?XD?main?test(8000h)
```

如果该程序模块采用 SMALL 模式编译, 并希望将 main 函数和 func 函数中的局部变量分别定位在 DATA 存储器以 40H 和 50H 开始的地址空间, 可以在 μVision2 环境下“Project 菜单/Options for Target 选项/BL51 Locate 标签页/Data 栏”内键入如下段名和起始地址,

```
?DT?main?test(40h),?DT?_func?test(50h)
```

设置完成后再对项目进行整体创建 (Build Target), 即可在所生成的目标代码中按指定的存储器空间定位变量的绝对地址。对于上例, 项目创建完成后生成的 test.M51 文件中关于变量的定位信息如下, 可以看到变量确实是按指定地址空间进行定位:

LINK MAP OF MODULE: test (TEST)

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME

*****		D A T A	M E M O R Y	*****
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
	0008H	0038H		*** GAP ***
DATA	0040H	0002H	UNIT	?DT?MAIN?TEST
	0042H	000EH		*** GAP ***
DATA	0050H	0001H	UNIT	?DT?_FUNC?TEST
IDATA	0051H	0001H	UNIT	?STACK
*****		X D A T A	M E M O R Y	*****
	0000H	4000H		*** GAP ***
XDATA	4000H	0001H	ABSOLUTE	
	4001H	1FFFH		*** GAP ***
XDATA	6000H	0006H	UNIT	?XD?TEST
	6006H	1FFAH		*** GAP ***
XDATA	8000H	0004H	UNIT	?XD?MAIN?TEST
*****		C O D E	M E M O R Y	*****
CODE	0000H	0003H	ABSOLUTE	
CODE	0003H	0027H	UNIT	?PR?MAIN?TEST
CODE	002AH	000CH	UNIT	?C_C51STARTUP
CODE	0036H	0003H	UNIT	?PR?_FUNC?TEST

9.10 Cx51 的库函数

Cx51 编译器的运行库中包含有丰富的库函数, 使用库函数可大大简化用户的程序设计工作, 提高编程效率。由于 8051 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` 的返回值类型为 `bit` 而不是 `int`。Keil Cx51 软件包中提供了如下库函数文件:

C51S.LIB	不包含浮点运算的小型库函数
C51FPS.LIB	包含浮点运算的小型库函数
C51C.LIB	不包含浮点运算的紧凑型库函数
C51FPC.LIB	包含浮点运算的紧凑型库函数
C51L.LIB	不包含浮点运算的大型库函数
C51FPL.LIB	包含浮点运算的大型库函数
80751.LIB	应用于 Philips8Xc751 系列单片机的库函数

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命令 `#include` 将有关的头文件包含进来。如果省略了头文件, 将不能保证函数的正确运行。Cx51 库函数中类型的选择考虑到了 8051 系列单片机的结构特性, 用户在自己的应用程序中应尽可能地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, 同时可减少应用程序的代码长度。

下面将 Cx51 库函数分类列出并作必要的解释。

9.10.1 字符函数 CTYPE.H

在 Cx51 函数库中, 下列字符函数的原型声明包含在头文件 `CTYPE.H` 中。

函数原型: `bit isalpha(char c);`

再入属性: `reentrant`

功能: 检查参数字符是否为英文字母, 是则返回 1, 否则返回 0。

参见: `isalnum`, `isctrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`

例子:

```
#include <ctype.h>
#include <stdio.h>          /* for printf */
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: `bit isalnum(char c);`

再入属性: `reentrant`

功 能: 检查参数字符是否为英文字母或数字字符, 是则返回 1, 否则返回 0。

参 见: `isalpha`, `isctrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`

例 子:

```
#include <ctype.h>
#include <stdio.h>          /* for printf */
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: `bit isctrl(char c);`

再入属性: `reentrant`

功 能: 检查参数值是否为控制字符 (值在 `0x00~0x1f` 之间或等于 `0x7f`), 是则返回 1, 否则返回 0。

参 见: `isalnum`, `isalpha`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`

例 子:

```
#include <ctype.h>
#include <stdio.h>          /* for printf */
void tst_isctrl (void) {
    unsigned char i;
    char *p;
    for (i = 0; i < 128; i++) {
        p = (isctrl (i) ? "YES" : "NO");
        printf ("isctrl (%c) %s\n", i, p);
    }
}
```

函数原型: `bit isdigit(char c);`

再入属性: `reentrant`

功 能: 检查参数的值是否为十进制数字 `0~9`, 是则返回 1, 否则返回 0。

参 见: `isalnum`, `isalpha`, `isctrl`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`

例 子:

```
#include <ctype.h>
#include <stdio.h>          /* for printf */
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: **bit isgraph(char c);**

再入属性: **reentrant**

功 能: 检查参数是否为可打印字符 (不包括空格), 可打印字符的值域为 0x21~0x7e。为真则返回 1, 否则返回 0。

参 见: **isalnum, isalpha, iscntrl, isdigit, islower, isprint, ispunct, isspace, isupper, isxdigit**

例 子: `#include <ctype.h>`
`#include <stdio.h> /* for printf */`
`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: **bit isprint(char c);**

再入属性: **reentrant**

功 能: 除了与 **isgraph** 相同之外, 还接受空格符(0x20)。

参 见: **isalnum, isalpha, iscntrl, isdigit, isgraph, islower, ispunct, isspace, isupper, isxdigit**

例 子: `#include <ctype.h>`
`#include <stdio.h> /* for printf */`
`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: `bit ispunct(char c);`

再入属性: `reentrant`

功 能: 检查字符参数是否为标点、空格或格式字符。如果是空格或是 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), 是则返回 1, 否则返回 0。ispunct 对下列字符返回 1:

!"#\$%&'()*+,-./:;<=>?@[\\]^_`{|}~

参 见: `isalnum`, `isalpha`, `isctrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `isspace`, `isupper`, `isxdigit`

例 子:

```
#include <ctype.h>
#include <stdio.h>                /* for printf */
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: `bit islower(char c);`

再入属性: `reentrant`

功 能: 检查参数字符的值是否为小写英文字母, 是则返回 1, 否则返回 0。

参 见: `isalnum`, `isalpha`, `isctrl`, `isdigit`, `isgraph`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`

例 子:

```
#include <ctype.h>
#include <stdio.h>                /* for printf */
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: `bit isupper(char c);`

再入属性: `reentrant`

功 能: 检查参数字符的值是否为大写英文字母, 是则返回 1, 否则返回 0。

参 见: `isalnum`, `isalpha`, `isctrl`, `isdigit`, `isgraph`, `isprint`, `ispunct`, `isspace`, `islower`, `isxdigit`

例 子:

```
#include <ctype.h>
```

```

#include <stdio.h>                /* for printf */
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: bit isspace(char c);

再入属性: reentrant

功 能: 检查参数字符是否为下列之一: 空格、制表符、回车、换行、垂直制表符和送纸 (值为 0x09~0x0d, 或为 0x20)。为真则返回 1, 否则返回 0。

参 见: isalnum, isalpha, iscntrl, isdigit, isgraph, isprint, ispunct, islower, isupper, isxdigit

例 子: #include <ctype.h>
#include <stdio.h> /* for printf */
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: bit isxdigit(char c);

再入属性: reentrant

功 能: 检查参数字符是否为 16 进制数字字符, 为真则返回 1, 否则返回 0。

参 见: isalnum, isalpha, iscntrl, isdigit, isgraph, isprint, ispunct, isspace, isupper, islower

例 子: #include <ctype.h>
#include <stdio.h> /* for printf */
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: `char toint(char c);`

再入属性: `reentrant`

功 能: 将 ASCII 字符的 0~9、a~f (大小写无关) 转换为 16 进制数字, 对于 ASCII 字符的 0~9, 返回值为 0H~9H, 对于 ASCII 字符的 a~f (大小写无关), 返回值为 0AH~0FH。

参 见: `toascii`

例 子:

```
#include <ctype.h>
#include <stdio.h>                /* for printf */
void tst_toint (void) {
    unsigned long l;
    char k;
    for (l = 0; isdigit (k = getchar ()); l *=10) {
        l += toint (k);
    }
}
```

函数原型: `char tolower(char c);`

再入属性: `reentrant`

功 能: 将大写字符转换成小写形式, 如果字符参数不在 A~Z 之间, 则该函数不起作用。

参 见: `_tolower`, `toupper`, `_toupper`

例 子:

```
#include <ctype.h>
#include <stdio.h>                /* for printf */
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: `char toupper(char c);`

再入属性: `reentrant`

功 能: 将小写字符转换为大写形式, 如果字符参数不在 a~z 之间则该函数不起作用。

参 见: `tolower`, `_tolower`, `_toupper`

例 子:

```
#include <ctype.h>
#include <stdio.h>                /* for printf */
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: `char toascii(char c);`

再入属性: `reentrant`

功 能: 该宏将任何字符型参数值缩小到有效的 ASCII 范围之内, 即将参数值和 0x7f 相与从而去掉第 7 位以上的所有数位。

参 见: `toint`

例 子:

```
#include <ctype.h>
#include <stdio.h>                /* for printf */
void tst_toascii ( char c) {
    char k;
    k = toascii (c);
    printf ("%c is an ASCII character\n", k);
}
```

函数原型: `char _tolower(char c);`

再入属性: `reentrant`

功 能: 该宏将字符参数 `c` 与常数 0x20 逐位相或, 从而将大写字符转换为小写字符。

参 见: `tolower`, `toupper`, `_toupper`

例 子:

```
#include <ctype.h>
#include <stdio.h>                /* for printf */
void tst__tolower ( char k) {
    if (isupper (k)) k = _tolower (k);
}
```

函数原型: `char _toupper(char c);`

再入属性: `reentrant`

功 能: 该宏将字符参数 `c` 与常数 0xdf 逐位相与, 从而将小写字符转换为大写字符。

参 见: `tolower`, `_tolower`, `toupper`

例 子:

```
#include <ctype.h>
#include <stdio.h>                /* for printf */
void tst__toupper ( char k) {
    if (islower (k)) k = _toupper (k);
}
```

9.10.2 标准 I/O 函数 STDIO.H

Cx51 库中标准 I/O 函数的原型声明包含在头文件 `STDIO.H` 中, 标准 I/O 函数通过 8051 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，这两个函数的源文件包含在 C51\LIB\ 目录中。另外需要注意，在使用 8051 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）初始化串行口的语句如下：

```
SCON=0x52;      /* SCON 置初值 */
TMOD=0x20;      /* TMOD 置初值 */
TH1=0xf3;       /* T1 置初值 */
TR1=1;          /* 启动 T1 */
```

函数原型: `char _getkey(void);`

再入属性: `non-reentrant`

功 能: 等待从 8051 的串口读入一个字符并返回读入的字符，这个函数是改变整个输入端口机制时应作修改的惟一一个函数。

参 见: `getchar`, `putchar`, `ungetchar`

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: `char getchar(void);`

再入属性: `reentrant`

功 能: `getchar` 使用 `_getkey` 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。

参 见: `_getkey`, `putchar`, `ungetchar`

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: `char * gets(char *s, int n);`

再入属性: `non-reentrant`

功 能: 该函数通过 `getchar` 从串口读入一个长度为 `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`。

参 见: `printf`, `puts`, `scanf`

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: `char ungetchar(char c);`

再入属性: `non-reentrant`

功 能: 将输入字符回送输入缓冲区, 因此下次 `gets` 或 `getchar` 可用该字符。成功时返回 `char` 型值 `c`, 失败时返回 `EOF`, 不能用 `ungetchar` 处理多个字符。

参 见: `_getkey`, `putchar`, `ungetchar`

例 子: `#include <stdio.h>`

```

void tst_ungetchar (void) {
    char k;
    while (isdigit (k = getchar ())) {
        /* stay in the loop as long as k is a digit */
    }
    ungetchar (k);
}

```

函数原型: `char putchar(char c);`

再入属性: `non-reentrant`

功 能: 通过 8051 串行口输出字符, 与函数 `_getkey` 一样, 这是改变整个输出机制所需修改的惟一个函数。

参 见: `getchar`, `_getkey`, `ungetchar`

例 子: `#include <stdio.h>`

```

void tst_putchar (void) {
    unsigned char i;
    for (i = 0x20; i < 0x7F; i++)
        putchar (i);
}

```

函数原型: `int printf(const char * fmstr [,argument]...);`

再入属性: `non-reentrant`

功 能: `printf` 以一定的格式通过 8051 的串行口输出数值和字符串, 返回值为实际输出的字符数。第一个参数 `fmstr` 是格式控制字符串。参数 `argument` 可以是字符串指针、字符或数值, 允许作为 `printf` 参数的总字节数受 Cx51 库限制, 由于 8051 系列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 1 个指针和 3

个长字), 在 LARGE 编译模式下, 最多可传递 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][(b|B|l|L)]type`

其中: flag 称为标志字符, 用于控制输出位置、符号、小数点以及 8 进制和 16 进制数的前缀等, 其内容和意义如表 9-16 所示。

表 9-16 flag 选项及其意义

flag 选项	意 义
-	输出左齐
+	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
#	如果它与 0、x 或 X 联用, 则在非 0 输出值前面加上 0、0x 或 0X。当它与值类型字符 g、G、f、e、E 联用时, 使输出值中产生一个十进制的小数点
*	忽略指定格式

width 用来定义欲显示的字符数, 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, 则在左端补以 0。precision 用来表示输出精度, 它是由小圆点“.”加上一个非负的十进制整数构成的。指定精度时可能会导致输出值被截断, 或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。

可选字符 b 或 B 和 l 或 L 通常与格式转换字符一起使用, 具体意义如表 9-17 所示。

表 9-17 可选字符 b、B、l、L 的意义

b,B	当它们与格式类型字符 d、o、u、x 或 X 联用时, 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 X 联用, 使参数类型被接受为[unsigned]long, 如%ld、%lx 等

type 称为输出格式转换字符, 其内容和意义如表 9-18 所示。

表 9-18 type 选项及其意义

格式转换字符	类 型	输 出 格 式
d	int	有符号十进制数(16 位)
u	unsigned int	无符号十进制数
o	unsigned int	无符号八进制数
x,X	unsigned int	无符号十六进制数
f	float	[-]dddd.dddd 形式的浮点数
e,E	float	[-]d.ddddE[sign]dd 形式的浮点数
g,G	float	选择 e 或 f 形式中更紧凑的一种输出格式
c	char	单个字符
s	一般指针	结束符为“\0”的字符串
p	一般指针	带存储器类型标志和偏移的指针 M:aaaa。其中, M:=C(ode), D(ata), l(data), P(data) a:=指针偏移值

参 见: gets, puts, scanf, sprintf, sscanf, vprintf, vsprintf

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 = 0x7FFFFFFF;
    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 xlong %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: int sprintf(char * s, const char * fmstr [,argument] ...);

再入属性: non-reentrant

功 能: sprintf 与 printf 的功能相似, 但数据不是输出到串行口, 而是通过一个指针 s, 送入内存缓冲区, 并以 ASCII 码的形式储存。参数 fmstr 与函数 printf 一致。

参 见: gets, puts, scanf, printf, sscanf, vprintf, vsprintf

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: `int puts(const char * s);`

再入属性: `reentrant`

功 能: 利用 `putchar` 函数将字符串和换行符写入串行口, 错误时返回 `EOF`, 否则返回 `0`。

参 见: `gets`, `scanf`, `printf`

例 子: `#include <stdio.h>`

```

void tst_puts (void) {
    puts ("Line #1");
    puts ("Line #2");
    puts ("Line #3");
}

```

函数原型: `int scanf(const char * fmstr [,argument] ...);`

再入属性: `non-reentrant`

功 能: `scanf` 在格式控制串的控制下, 利用 `getchar` 函数从串行口读入数据, 每遇到一个符合格式控制串 `fmstr` 规定的值, 就将它按顺序存入由参数指针 `argument` 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):

`%[*][width][bhl]type`

其中: `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字符数目。不超过规定宽度的字符从输入流中读出, 并被转换到相应的变量, 但是如果先遇到一个空格符或无法辨识的字符, 读入的字符数可能会小于宽度值。

可选字符 `b`、`h`、`l` 可以直接位于输入格式转换字符之前, 其意义如表 9-19 所示。

表 9-19 可选字符 `b`、`h`、`l` 的意义

<code>B,h</code>	它们用作格式类型 <code>d,o,u</code> 和 <code>x</code> 的前缀, 用这个前缀可将参数定义为字符指针, 指示输入整型数, 如 <code>%bu</code> , <code>%bx</code>
<code>l</code>	它被用作格式类型 <code>d,o,u</code> 和 <code>x</code> 的前缀, 用这个前缀可将参数定义成长指针, 指示输入长整数, 如 <code>%lu</code> , <code>%lx</code>

`type` 称为输入格式转换字符, 其内容和意义如表 9-20 所示。

表 9-20 type 选项及其意义

格式转换字符	类 型	输 入 格 式
d	int *	有符号的十进制数
i	int *	有符号的十进制、十六进制、八进制数
u	unsigned int *	无符号十进制数
o	unsigned int *	无符号八进制数
x	unsigned int *	无符号十六进制数
f,e,g	float *	浮点数
c	char *	一个字符
s	char *	一个字符串

参 见: gets, printf, puts, sprintf, sscanf, vprintf, vsprintf

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: int sscanf(char * s, const char * fmstr [,argument] ...);

再入属性: non-reentrant

功 能: sscanf 与 scanf 的输入方式相似, 但字符串的输入不是通过串行口, 而是通过指针 s 指向的数据缓冲区。输入数据根据格式控制字符串 fmstr 被存放到

由 `argument` 指定的地址。格式控制串 `fmstr` 与 `scanf` 函数一致。`sscanf` 参数允许的总字节数受 Cx51 库的限制, 在 `SMALL` 和 `COMPACT` 编译模式下, 最大允许传递 15 个字节的参数 (即 5 个指针, 或 2 个指针、2 个长整型和 1 个字符型), 在 `LARGE` 编译模式下, 最大允许传递 40 个字节的参数。

参 见: `gets`, `printf`, `puts`, `scanf`, `sprintf`, `vprintf`, `vsprintf`

例 子: `#include <stdio.h>`

```
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: `void vprintf(const char * s, char * fmstr, char * argptr);`

再入属性: `non-reentrant`

功 能: `vprintf()` 将格式化字符串和数据值输出到由指针 `s` 指向的内存缓冲区内。该函数似于 `sprintf()`, 但它接受一个指向变量表的指针而不是变量表。返回值为实际写入到输出字符串中的字符数。格式控制字符串 `fmstr` 与 `printf` 函数一致。

参 见: `gets`, `puts`, `printf`, `scanf`, `sprintf`, `sscanf`, `vsprintf`

例 子: `#include <stdio.h>`

`#include <stdarg.h>`

```
void error (char *fmt, ...) {
    va_list arg_ptr;
    va_start (arg_ptr, fmt);          /* format string */
    ...
}
```

```

    vprintf (fmt, arg_ptr);
    va_end (arg_ptr);
}

void tst_vprintf (void) {
    int i;
    i = 1000;

    /* call error with one parameter */
    error ("Error: '%d' number too large\n", i);
    /* call error with just a format string */
    error ("Syntax Error\n");
}

```

函数原型: void vsprintf(char * s, const char * fmtstr, char * argptr);

再入属性: non-reentrant

功 能: vprintf()将格式化字符串和数字值输出到由 s 指定的内存缓冲区内, 该函数类似于 sprintf()函数, 但它接受一个指向变量表的指针而不是变量表。返回值为实际写入到输出字符串中的字符数。格式控制字符串 fmtstr 与 printf 函数一致。

参 见: gets, puts, printf, scanf, sprintf, sscanf, vprintf

例 子: #include <stdio.h>
#include <stdarg.h>
xdata char etxt[30]; /* text buffer */
void error (char *fmt, ...) {
 va_list arg_ptr;
 va_start (arg_ptr, fmt); /* format string */
 vsprintf (etxt, fmt, arg_ptr);
 va_end (arg_ptr);
}

void tst_vprintf (void) {
 int i;
 i = 1000;
 /* call error with one parameter */
 error ("Error: '%d' number too large\n", i);
 /* call error with just a format string */
 error ("Syntax Error\n");
}

9.10.3 字符串函数 STRING.H

字符串函数的原型声明包含在头文件 STRING.H 中, 字符串函数通常接收指针串作为

输入值。一个字符串应包括 2 个或多个字符，字符串的结尾以空字符表示。在函数 `memcmp`、`memcpy`、`memchr`、`memccpy`、`memset` 和 `memmove` 中，字符串的长度由调用者明确规定，这些函数可工作在任何模式。

函数原型: `void * memchr(void * s1, char val, int len);`

再入属性: `reentrant`

功 能: `memchr` 顺序搜索字符串 `s1` 的前 `len` 个字符以找出字符 `val`，成功时返回 `s1` 中指向 `val` 的指针，失败时返回 `NULL`。

参 见: `memccpy`, `memcmp`, `memcpy`, `memmove`, `memset`

例 子: `#include <string.h>`

```
#include <stdio.h>                /* for printf */
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 * s1, void * s2, int len);`

再入属性: `reentrant`

功 能: `memcmp` 逐个字符比较串 `s1` 和 `s2` 的前 `len` 个字符，成功（相等）时返回 0，如果串 `s1` 大于或小于 `s2`，则相应地返回一个正数或一个负数。

参 见: `memccpy`, `memchr`, `memcpy`, `memmove`, `memset`

例 子: `#include <string.h>`

```
#include <stdio.h>                /* for printf */
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: memcpy 从 src 所指向的内存中拷贝 len 个字符到 dest 中, 返回指向 dest 中最后一个字符的指针。如果 src 与 dest 发生交迭, 则结果是不可预测的。

参 见: memccpy, memcmp, memchr, memmove, memset

例 子: #include <string.h>
#include <stdio.h> /* for printf */
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 * memccpy(void * dest, void * src, char val, int len);

再入属性: non-reentrant

功 能: memccpy 拷贝 src 中 len 个元素到 dest 中。如果实际拷贝了 len 个字符则返回 NULL。拷贝过程在拷贝完字符 val 后停止, 此时返回指向 dest 中下一个元素的指针。

参 见: memchr, memcmp, memcpy, memmove, memset

例 子: #include <string.h>
#include <stdio.h> /* for printf */
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 * memmove(void * dest, void * src, int len);

再入属性: reentrant

功 能: memmove 的工作方式与 memcpy 相同, 但拷贝的区域可以交迭。

参 见: memccpy, memcmp, memcpy, memchr, memset

例 子: #include <string.h>
#include <stdio.h> /* for printf */
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 *s, char val, int len);`

再入属性: `reentrant`

功 能: `memset` 用 `val` 来填充指针 `s` 中 `len` 个单元。

参 见: `memcpy`, `memcmp`, `memcpy`, `memmove`, `memchr`

例 子: `#include <string.h>`
`#include <stdio.h> /* for printf */`
`void tst_memset (void) {`
 `char buf [10];`
 `memset (buf, '\0', sizeof (buf));`
 `/* fill buffer with null characters */`
`}`

函数原型: `void *strcat(char *s1, char *s2);`

再入属性: `non-reentrant`

功 能: `strcat` 将串 `s2` 拷贝到 `s1` 的尾部。`strcat` 假定 `s1` 所定义的地址区域足以接受两个串。返回指向 `s1` 串中第一个字符的指针。

参 见: `strcpy`, `strlen`, `strncat`, `strncpy`

例 子: `#include <string.h>`
`#include <stdio.h> /* for printf */`
`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 *strncat(char *s1, char *s2, int n);`

再入属性: `non-reentrant`

功 能: `strncat` 拷贝串 `s2` 中 `n` 个字符到 `s1` 的尾部, 如果 `s2` 比 `n` 短, 则只拷贝 `s2` (包括串结束符)。

参 见: `strcpy`, `strlen`, `strncpy`

例 子: `#include <string.h>`
`#include <stdio.h> /* for printf */`

```

void tst_strncat (void) {
    char buf [21];
    strcpy (buf, "test #");
    strncat (buf, "three", sizeof (buf) - strlen(buf));
}

```

函数原型: `char strcmp(char * s1, char * s2);`

再入属性: `reentrant`

功 能: `strcmp` 比较串 `s1` 和 `s2`, 如果相等则返回 0, 如果 `s<s2`, 则返回一个负数, 如果 `s1>s2`, 则返回一个正数。

参 见: `memcmp`, `strncmp`

例 子: `#include <string.h>`

```

#include <stdio.h>                                /* for printf */
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 strncmp(char * s1, char * s2, int n);`

再入属性: `non-reentrant`

功 能: `strncmp` 比较串 `s1` 和 `s2` 中的前 `n` 个字符。返回值与 `strcmp` 相同。

参 见: `memcmp`, `strcmp`

例 子: `#include <string.h>`

```

#include <stdio.h>                                /* for printf */
void tst_strncmp (void) {
    char str1 [] = "Wrodanahan T.J.";
    char str2 [] = "Wrodanaugh J.W.";
    char i;
    i = strncmp (str1, str2, 15);
    if (i < 0) printf ("str1 < str2\n");
    else if (i > 0) printf ("str1 > str2\n");
    else printf ("str1 == str2\n");
}

```

函数原型: `char * strcpy(char *s1, char *s2);`

再入属性: `non-reentrant`

功 能: `strcpy` 将串 `s2`, 包括结束符, 拷贝到 `s1` 中, 返回指向 `s1` 中第一个字符的指针。

参 见: `strcat`, `strlen`, `strncat`, `strncmp`

例 子:

```
#include <string.h>
#include <stdio.h>          /* for printf */
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: `char * strncpy(char *s1, char *s2, int n);`

再入属性: `non-reentrant`

功 能: `strncpy` 与 `strcpy` 相似, 但它只拷贝 `n` 个字符。如果 `s2` 的长度小于 `n`, 则 `s1` 串以 0 补齐到长度 `n`。

参 见: `strcat`, `strcpy`, `strlen`, `strncat`

例 子:

```
#include <string.h>
#include <stdio.h>          /* for printf */
void tst_strncpy (char *s) {
    char buf [21];
    strncpy (buf, s, sizeof (buf));
    buf [sizeof (buf)] = '\0';
}
```

函数原型: `int strlen(char *s1);`

再入属性: `reentrant`

功 能: `strlen` 返回串 `s1` 中的字符个数, 不包括结尾的空字符。

参 见: `strcat`, `strcpy`, `strncpy`, `strncat`

例 子:

```
#include <string.h>
#include <stdio.h>          /* for printf */
void tst_strlen (void) {
    char buf [] = "Find the length of this string";
    int len;
    len = strlen (buf);
    printf ("string length is %d\n", len);
}
```

函数原型: `char * strstr(const char * s1, char * s2);`

再入属性: `reentrant`

功 能: `strstr` 搜索字符串 `s2` 第一次出现在 `s1` 中的位置, 并返回一个指向第一次出现位置开始处的指针。如果字符串 `s1` 中不包括的字符串 `s2`, 则返回一个空指针。

参 见: `strchr`, `strpos`

例 子:

```
#include <string.h>
#include <stdio.h>          /* for printf */
char s1[] = "My House is small";
char s2[] = "My Car is green";
void tst_strstr(void) {
    char * s;
    s = strstr (s1, "House");
    printf("substr (s1, \"House\") returns %s\n", s);
}
```

函数原型: `char * strchr(char * s1, char c);`

`int strpos(char * s1, char c);`

再入属性: `reentrant`

功 能: `strchr` 搜索 `s1` 串中第一个出现的字符 `c`, 如果成功则返回指向该字符的指针, 否则返回 `NULL`。被搜索的字符可以是串结束符, 此时返回值是指向串结束符的指针。

`strpos` 的功能与 `strchr` 类似, 但返回的是字符 `c` 在串 `s1` 中第一次出现的位置值, 没有找到则返回-1, `s1` 串首字符的位置值是 0。

参 见: `strcspn`, `strpbrk`, `strchr`, `strpbrk`, `strpos`, `strspn`

例 子:

```
#include <string.h>
#include <stdio.h>          /* for printf */
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);
}

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(char * s1, char c);`

`int strnpos(char * s1, char c);`

再入属性: `reentrant`

功 能: `strrchr` 搜索 `s1` 串中最后一个出现的字符 `c`, 如果成功则返回指向该字符的指针, 否则返回 `NULL`。被搜索的字符可以是串结束符。

`strnpos` 的功能与 `strrchr` 相似, 但返回值是字符 `c` 在 `s1` 串中最后一次出现的位置值, 没有找到则返回 `-1`。

参 见: `strchr`, `strcspn`, `strpbrk`, `strpos`, `strpbrk`, `strspn`

例 子: `#include <string.h>`

```

#include <stdio.h> /* for printf */
void tst_strrchr (void) {
    char *s;
    char buf [] = "This is a test";
    s = strrchr (buf, 't');
    if (s != NULL)
        printf ("found the last 't' at %s\n", s);
}

void tst_strnpos ( char *s) {
    int i;
    i = strnpos (s, ' ');
    if (i == -1)
        printf ("No spaces found in %s\n", s);
    else
        printf ("Last space in %s is at offset %d\n", s, i);
}

```

函数原型: `int strspn(char * s1, char * set);`

`int strcspn(char * s1, char * set);`

`char * strpbrk(char * s1, char * set);`

`char * strpbrk(char * s1, char * set);`

再入属性: `non-reentrant`

功 能: `strspn` 搜索 `s1` 串中第一个不包括在 `set` 串中的字符, 返回值是 `s1` 中包括在 `set` 里的字符个数。如果 `s1` 中所有字符都包括在 `set` 里面, 则返回 `s1` 的长度 (不包括结束符)。如果 `set` 是空串则返回 `0`。

`strcspn` 与 `strspn` 相似。但它搜索的是 `s1` 串中第一个包含在 `set` 里的字符。

`strpbrk` 与 `strspn` 相似, 但返回指向搜索到的字符的指针, 而不是个数, 如

果未找到，则返回 NULL。

strpbrk 与 strpbkr 相似，但它返回 s1 中指向找到的 set 字符集中最后一个字符的指针。

参 见: strchr, strchr, strpos, strpos

例 子: #include <string.h>

```
#include <stdio.h>                /* for printf */
void tst_strspn ( char *digit_str) {
    char octd [] = "01234567";
    int i;
    i = strspn (digit_str, octd);
    if (digit_str [i] != '\0')
        printf ("%c is not an octal digit\n", digit_str [i]);
}

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);
}

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);
}

void tst_strrpbrk (void) {
    char vowels [] = "AEIOUaeiou";
    char text [] = "American National Standards Institute";
    char * p;
    p = strpbrk (text, vowels);
    if (p == NULL)
        printf ("No vowels found in %s\n", text);
    else
        printf ("Last vowel is at %s\n", p);
}
```

9.10.4 标准函数 STDLIB.H

标准函数的原型声明包含在头文件 **STDLIB.H** 中, 利用标准函数可以完成数据类型转换以及存储器分配操作。

函数原型: `float atof(char * s1);`

再入属性: `non-reentrant`

功 能: `atof` 将字符串 `s1` 转换成浮点数值并返回它, 输入串中必须包含与浮点值规定相符的数。该函数在遇到第一个不能构成数字的字符时, 停止对输入字符串的读操作。

`atof` 函数要求字符串具有如下格式:

`[{+|-}]数字[.数字][{e|E}][{+|-}]数字`

这里数字可以是一个或多个十进制数。

参 见: `atoi, atof`

例 子:

```
#include <stdlib.h>
#include <stdio.h>          /* for printf */
void tst_atof (void) {
    float f;
    char s [] = "1.23";
    f = atof (s);
    printf ("ATOF(%s) = %f\n", s, f);
}
```

函数原型: `long atol(char * s1);`

再入属性: `non-reentrant`

功 能: `atol` 将字符串 `s1` 转换成一个长整型数值并返回它, 输入串中必须包含与长整型数格式相符的字符串。该函数在遇到第一个不能构成数字的字符时, 停止对输入字符串的读操作。

`atol` 函数要求字符串具有如下格式:

`[whitespace][{+|-}]数字`, 其中 `whitespace` 由空格、`/`、`tab` 符组成。这里数字可以是一个或多个十进制数。

参 见: `atoi, atof`

例 子:

```
#include <stdlib.h>
#include <stdio.h>          /* for printf */
void tst_atol (void) {
    long l;
    char s [] = "8003488051";
    l = atol (s);
    printf ("ATOL(%s) = %ld\n", s, l);
}
```

函数原型: `int atoi(char * s1);`

再入属性: `non-reentrant`

功 能: `atoi` 将串 `s1` 转换成整型数并返回它。输入串中必须包含与整型数格式相符的字符串。该函数在遇到第一个不能构成数字的字符时, 停止对输入字符串的读操作。

`atoi` 函数要求字符串具有如下格式:

`[whitespace][+|-]`数字, 其中 `whitespace` 由空格、`/`、`tab` 符组成。这里数字可以是一个或多个十进制数。

参 见: `atol`, `atof`

例 子:

```
#include <stdlib.h>
#include <stdio.h>          /* for printf */
void tst_atoi (void) {
    int i;
    char s [] = "12345";
    i = atoi (s);
    printf ("atoi(%s) = %d\n", s, i);
}
```

函数原型: `void * calloc(unsigned int n, unsigned int size);`

再入属性: `non-reentrant`

功 能: `calloc` 函数为 `n` 个元素的数组分配内存空间, 数组中每个元素的大小为 `size`, 所分配的内存区域用 0 进行初始化。返回值为已分配的内存单元起始地址, 如不成功则返回 0。

参 见: `free`, `init_mempool`, `malloc`, `realloc`

例 子:

```
#include <stdlib.h>
#include <stdio.h>          /* for printf */
void tst_calloc (void) {
    int xdata *p;           /* ptr to array of 100 ints */
    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 free(void xdata * p);`

再入属性: `non-reentrant`

功 能: `free` 释放指针 `p` 所指向的存储器区域, 如果 `p` 为 `NULL`, 则该函数无效, `p` 必须是以前用 `calloc`、`malloc` 或 `realloc` 函数分配的存储器区域。调用 `free` 函数后, 被释放的存储器区域就可以参加以后的分配。

参 见: `calloc`, `init_mempool`, `malloc`, `realloc`

```

例 子: #include <stdlib.h>
        #include <stdio.h>                /* for printf */
        void tst_free (void) {
            void * mbuf;
            printf ("Allocating memory\n");
            mbuf = malloc (1000);
            if (mbuf == NULL) {
                printf ("Unable to allocate memory\n");
            }
            else {
                free (mbuf);
                printf ("Memory free\n");
            }
        }

```

函数原型: `void init_mempool(void xdata * p, unsigned int size);`

再入属性: `non-reentrant`

功 能: `init_mempools` 对可被函数 `calloc`、`free`、`malloc` 和 `realloc` 管理的存储器区域进行初始化, 指针 `p` 表示存储区的首地址, `size` 表示存储区的大小。

参 见: `calloc`, `free`, `malloc`, `realloc`

```

例 子: #include <stdlib.h>
        void tst_init_mempool (void) {
            xdata void * p;
            int i;
            init_mempool (&XBYTE [0x2000], 0x1000);
            /* initialize memory pool at xdata 0x2000 for 4096 bytes */
            p = malloc (100);
            for (i = 0; i < 100; i++) ((char *) p)[i] = i;
            free (p);
        }

```

函数原型: `void * malloc(unsigned int size);`

再入属性: `non-reentrant`

功 能: `malloc` 函数在内存中分配一个 `size` 字节大小的存储器空间, 返回值为一个 `size` 大小对象所分配的内存指针。如果返回 `NULL`, 则无足够的内存空间可用。

参 见: `calloc`, `free`, `init_mempool`, `realloc`

```

例 子: #include <stdlib.h>
        #include <stdio.h>                /* for printf */
        void tst_malloc (void) {
            unsigned char xdata *p;

```

```

p = malloc (1000);                /* allocate 1000 bytes */
if (p == NULL)
    printf ("Not enough memory space\n");
else
    printf ("Memory allocated\n");
}

```

函数原型: `void * realloc(void xdata * p, unsigned int size);`

再入属性: `non-reentrant`

功 能: `realloc` 函数用于调整先前分配的存储器区域大小。参数 `p` 指示该存储区域的起始地址, 参数 `size` 表示新分配存储器区域的大小。原存储器区域的内容被复制到新存储器区域中。如果新区域较大, 多出的区域将不作初始化。`realloc` 返回指向新存储区的指针, 如果返回 `NULL`, 则无足够大的内存可用, 这时将保持原存储区不变。

参 见: `calloc`, `free`, `init_mempool`, `malloc`

例 子: `#include <stdlib.h>`
`#include <stdio.h>` /* for printf */
`void tst_realloc (void) {`
 `void xdata *p;`
 `void xdata *new_p;`
 `p = malloc (100);`
 `if (p != NULL) {`
 `new_p = realloc (p, 200);`
 `if (new_p != NULL) p = new_p;`
 `else printf ("Reallocation failed\n");`
 `}`
`}`

函数原型: `int rand();`

再入属性: `reentrant`

功 能: `rand` 返回一个 0 到 32767 之间的伪随机数, 对 `rand` 的相继调用将产生相同序列的随机数。

参 见: `srand`

例 子: `#include <stdlib.h>`
`#include <stdio.h>` /* for printf */
`void tst_rand (void) {`
 `int i;`
 `int r;`
 `for (i = 0; i < 10; i++) {`
 `printf ("I = %d, RAND = %d\n", i, rand ());`
 `}`
`}`

函数原型: `void srand(int n);`

再入属性: `non-reentrant`

功 能: `srand` 用来将随机数发生器初始化成一个已知（或期望）值。

参 见: `rand`

例 子: `#include <stdlib.h>`

```
#include <stdio.h>          /* for printf */
void tst_srand (void) {
    int i;
    int r;
    srand (56);
    for (i = 0; i < 10; i++) {
        printf ("I = %d, RAND = %d\n", i, rand ());
    }
}
```

函数原型: `unsigned long strtod(const char * s, char **ptr);`

再入属性: `non-reentrant`

功 能: `strtod` 函数将字符串 `s` 转换为一个浮点型数据并返回它，字符串前面的空格、`/`、`tab` 符被忽略。`atof` 函数要求字符串具有如下格式：

`[[+|-]]数字[.数字][[e|E]][[+|-]]数字`

这里的数字可以是一个或多个十进制数。

`ptr` 被设为指向字符串 `s` 中已经转换部分后面的第一个字符。如果不能进行转换，则 `ptr` 被设为字符串的值，同时返回 0。

参 见: `atof`, `atoi`, `atol`, `strtol`, `stroul`

例 子: `#include <stdlib.h>`

```
#include <stdio.h>          /* for printf */
void tst_strtod (void) {
    float f;
    char s [] = "1.23";
    f = strtod (s, NULL);
    printf ("strtod(%s) = %f\n", s, f);
}
```

函数原型: `long strtol(const char * s, char **ptr, unsigned char base);`

再入属性: `non-reentrant`

功 能: `strtol` 函数将字符串 `s` 转换为一个 `long` 型数据并返回它，字符串前面的空格、`/`、`tab` 符被忽略。`atol` 函数要求字符串具有如下格式：

`[whitespace][[+|-]]数字`

这里的数字可以是一个或多个十进制数。

如果参数 `base` 为 0，则数据为十进制常数、八进制常数、十六进制常数格

式。数值的基数从格式推出，如果 base 的值在 2 到 36 之间，则数据必须是一个字母或数字的非零序列，表示指定基数的一个整数。字母 a 到 z 或 A 到 Z 分别表示数据 10 到 36，只有小于 base 的字母表示的值才有效。如果 base 为 16，则数据可能以 0x 或 0X 开头，但是 0x 或 0X 将被忽略。

ptr 被设为指向字符串 s 中已经转换部分后面的第一个字符。如果不能进行转换，则 ptr 被设为字符串的值，同时返回 0。

参 见: atof, atoi, atol, strtod, strtoul

例 子: #include <stdlib.h>
#include <stdio.h> /* for printf */
char s [] = "-123456789";
void tst_strtol (void) {
 long l;
 l = strtol (s, NULL, 10);
 printf ("strtol(%s) = %ld\n", s, l);
}

函数原型: long strtoul(const char * s, char **ptr, unsigned char base);

再入属性: non-reentrant

功 能: strtoul 函数将字符串 s 转换为一个 unsigned long 型数据并返回它，溢出时则返回 ULONG_MAX。字符串前面的空格、/、tab 符被忽略。atoul 函数要求字符串具有如下格式：

[whitespace][+|-]]数字

这里的数字可以是一个或多个十进制数。

如果参数 base 为 0，则数据为十进制常数、八进制常数、十六进制常数格式。数值的基数从格式推出，如果 base 的值在 2 到 36 之间，则数据必须是一个字母或数字的非零序列，表示指定基数的一个整数。字母 a 到 z 或 A 到 Z 分别表示数据 10 到 36，只有小于 base 的字母表示的值才有效。如果 base 为 16，则数据可能以 0x 或 0X 开头，但是 0x 或 0X 将被忽略。

ptr 被设为指向字符串 s 中已经转换部分后面的第一个字符。如果不能进行转换，则 ptr 被设为字符串的值，同时返回 0。

参 见: atof, atoi, atol, strtod, strtol

例 子: #include <stdlib.h>
#include <stdio.h> /* for printf */
char s [] = "12345AB";
void tst_strtoul (void) {
 unsigned long ul;
 ul = strtoul (s, NULL, 16);
 printf ("strtoul(%s) = %lx\n", s, ul);
}

9.10.5 数学函数 MATH.H

数学函数的原型声明包含在头文件 MATH.H 中。

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: `abs` 计算并返回 `val` 的绝对值, 如果 `val` 为正, 则不作改变就返回, 如果为负, 则返回相反数。其余三个函数除了变量和返回值类型不同之外, 其他功能完全相同。

例 子:

```
#include <math.h>
#include <stdio.h>                                /* for printf */
void tst_abs (void) {
    int x;
    int y;
    x = -42;
    y = abs (x);
    printf ("ABS(%d) = %d\n", x, y);
}

void tst_cabs (void) {
    char x;
    char y;
    x = -23;
    y = cabs (x);
    printf ("CABS(%bd) = %bd\n", x, y);
}

void tst_fabs (void) {
    float x;
    float y;
    x = 10.2;
    y = fabs (x);
    printf ("FABS(%f) = %f\n", x, y);
    x = -3.6;
    y = fabs (x);
    printf ("FABS(%f) = %f\n", x, y);
}

void tst_labs (void) {
    long x;
```

```

    long y;
    x = -12345L;
    y = labs (x);
    printf ("LABS(%ld) = %ld\n", x, y);
}

```

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 计算并返回浮点数 x 的指数函数, log 计算并返回浮点数 x 的自然对数 (自然对数以 e 为底, e=2.718282), log10 计算并返回浮点数 x 以 10 为底 x 的对数。

例 子: #include <math.h>

#include <stdio.h> /* for printf */

```

void tst_exp (void) {
    float x;
    float y;
    x = 4.605170186;
    y = exp (x); /* y = 100 */
    printf ("EXP(%f) = %f\n", x, y);
}

```

```

void tst_log (void) {
    float x;
    float y;
    x = 2.71838;
    x *=x;
    y = log (x); /*y =2 */
    printf ("LOG(%f) = %f\n", x, y);
}

```

```

void tst_log10 (void) {
    float x;
    float y;
    x = 1000;
    y = log10 (x); /*y =3 */
    printf ("LOG10(%f) = %f\n", x, y);
}

```

函数原型: float sqrt(float x);

再入属性: non-reentrant

功 能: sqrt 计算并返回 x 的正平方根。

例 子: #include <math.h>
#include <stdio.h> /* for printf */
void tst_sqrt (void) {
float x;
float y;
x = 25.0;
y = sqrt (x); /* y = 5 */
printf ("SQRT(%f) = %f\n", x, y);
}

函数原型: float cos(float x);

float sin(float x);

float tan(float x);

再入属性: non-reentrant

功 能: cos 计算并返回 x 的余弦值, sin 计算并返回 x 的正弦值, tan 计算并返回 x 的正切值, 所有函数的变量范围都是 $-\pi/2 \sim +\pi/2$, 变量的值必须在 ± 65535 之间, 否则产生一个 NaN 错误。

例 子: #include <math.h>
#include <stdio.h> /* for printf */
void tst_cos (void) {
float x;
float y;
for (x = 0; x < (2 * 3.1415); x += 0.1) {
y = cos (x);
printf ("COS(%f) = %f\n", x, y);
}
}

void tst_sin (void) {
float x;
float y;
for(x =0;x <(2 *3.1415); x += 0.1) {
y = sin (x);
printf ("SIN(%f) = %f\n", x, y);
}
}

void tst_tan (void) {
float x, y, pi;
pi = 3.14159;
for (x = -(pi/4); x < (pi/4); x += 0.1) {

```
    y = tan (x);  
    printf ("TAN(%f) = %f\n", x, y);  
  }  
}
```

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: acos 计算并返回 x 的反余弦值, asin 计算并返回 x 的反正弦值, atan 计算并返回 x 的反正切值, 它们的值域为 $-\pi/2 \sim +\pi/2$ 。atan2 计算并返回 y/x 的反正切值, 其值域为 $-\pi \sim +\pi$ 。

例 子: #include <math.h>

#include <stdio.h>

/* for printf */

void tst_acos (void) {

float x;

float y;

for (x = -1.0; x <= 1.0; x += 0.1) {

y = acos (x);

printf ("ACOS(%f) = %f\n", x, y);

}

}

void tst_asin (void) {

float x;

float y;

for (x = -1.0; x <= 1.0; x += 0.1) {

y = asin (x);

printf ("ASIN(%f) = %f\n", x, y);

}

}

void tst_atan (void) {

float x;

float y;

for (x = -10.0; x <= 10.0; x += 0.1) {

y = atan (x);

printf ("ATAN(%f) = %f\n", x, y);

}

}


```

void tst_atan2 () {
    float x;
    float y;
    float z;
    x = -1.0;
    for (y = -10.0; y < 10.0; y += 0.1) {
        z = atan2 (y,x);
        printf ("ATAN2(%f/%f) = %f\n", y, x, z);
    }
    /* z approaches -pi as y goes from -10 to 0 */
    /* z approaches +pi as y goes from +10 to 0 */
}

```

函数原型: float cosh(float x);

float sinh(float x);

float tanh(float x);

再入属性: non-reentrant

功 能: cosh 计算并返回 x 的双曲余弦值, sinh 计算并返回 x 的双曲正弦值, tanh 计算并返回 x 的双曲正切值。

例 子: #include <math.h>

#include <stdio.h> /* for printf */

```

void tst_cosh (void) {
    float x;
    float y;
    for (x = 0; x < (2 * 3.1415); x += 0.1) {
        y = cosh (x);
        printf ("COSH(%f) = %f\n", x, y);
    }
}

```

```

void tst_sinh (void) {
    float x;
    float y;
    for(x=0;x <(2 *3.1415); x += 0.1) {
        y = sinh (x);
        printf ("SINH(%f) = %f\n", x, y);
    }
}

```

```

void tst_tanh (void) {
    float x;
    float y;
    float pi;

```

```
pi = 3.14159;
for (x = -(pi/4); x < (pi/4); x += 0.1) {
    y = tanh (x);
    printf ("TANH(%f) = %f\n", x, y);
}
}
```

函数原型: `float ceil(float x);`

再入属性: `non-reentrant`

功 能: `ceil` 计算并返回一个不小于 `x` 的最小整数 (作为浮点数)。

参 见: `floor`

例 子:

```
#include <math.h>
#include <stdio.h> /* for printf */
void tst_ceil (void) {
    float x;
    float y;
    x = 45.998;
    y = ceil (x);
    printf ("CEIL(%f) = %f\n", x, y);
    /* output is "CEIL(45.998) = 46" */
}
```

函数原型: `float floor(float x);`

再入属性: `non-reentrant`

功 能: `floor` 计算并返回一个不大于 `x` 的最大整数(作为浮点数)。

参 见: `ceil`

例 子:

```
#include <math.h>
#include <stdio.h> /* for printf */
void tst_floor (void) {
    float x;
    float y;
    x = 45.998;
    y = floor (x);
    printf ("FLOOR(%f) = %f\n", x, y); /* printf 45 */
}
```

函数原型: `float modf(float x, float *ip);`

再入属性: `non-reentrant`

功 能: `modf` 将浮点数 `x` 分成整数和小数两部分, 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>                /* for printf */
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: pow 计算并返回 x^y 的值, 如果 x 不等于 0 而 $y=0$, 则返回 1。当 $x=0$ 且 $y \leq 0$ 或当 $x < 0$ 且 y 不是整数时则返回 NaN。

参 见: sqrt

例 子: #include <math.h>
#include <stdio.h> /* for printf */
void tst_pow (void) {
 float base;
 float power;
 float y;
 base = 2.0;
 power = 8.0;
 y = pow (base, power); /* y = 256 */
 printf ("%f ^ %f = %f\n", base, power, y);
}

9.10.6 绝对地址访问 ABSACC.H

进行绝对地址访问的宏定义包含在头文件 ABSACC.H 中。

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: reentrant

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, 可以作字节寻址。CBYTE 寻址 CODE 区, DBYTE 寻址 DATA 区, PBYTE 寻址分页 XDATA 区(采用 MOVX @R0 指令), XBYTE 寻址 XDATA 区(采用 MOVX @DPTR)指令。例如, 下列语句在外部存储器区域访问地址 0x0002:

```

xval=XBYTE[0x0002];
XBYTE[0x0002]=0x20;

```

通过使用#define 预处理命令,可采用其他符号定义绝对地址,例如:

#define XIO XBYTE[0x1000] 即将符号 XIO 定义成外部数据存储器地址 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: reentrant

功 能: 这个宏与前面一个宏相似,只是它们指定的数据类型为 unsigned int。通过灵活运用不同的数据类型,8051 的所有存储器空间都可以进行访问。例如,下列语句在外部存储器区域访问地址 0x0004:

```
xval=XWORD[0x0002];
XWORD[0x0002]=0x20;
```

注意,使用宏 CWORD、DWORD、PWORD、XWORD 时,实际存储器地址应为 2 *sizeof(unsigned int)。

函数原型: #define FVAR(object, addr)((object volatile far *) ((addr)+0x10000L))
#define FCVAR(object, addr)((object const far *) ((addr)+0x810000L))

再入属性: reentrant

功 能: 宏 FVAR 和 FCVAR 用于访问 far 和 const far 存储器区域。FVAR 用于访问 far 空间(存储类为 HDATA),FCVAR 用于访问 const far 空间(存储类为 HCONST)。

例 子: #define IOVL FVAR(long,0x14FFE) /* long at HDATA address 0x14FFE */
var = IOVAL; /* read */
IOVAL = 0x10; /* write */
var = FCVAR(int,0x24002) /* read int from HCONST address 0x24002 */

注意:绝对地址目标不能穿过 64KB 的段边界,例如不能在地址 0xFFFFE 访问一个 long 变量。

函数原型: #define FARRAY(object, base)((object volatile far *) ((base)+0x10000L))
#define FCARRAY(object, base)((object const far *) ((base)+0x810000L))

功 能: 宏 FARRAY 和 FCARRAY 用于访问位于 far 和 const far 存储区的数组类型目标, FARRAY 用于访问 far 空间(存储类为 HDATA), FCARRAY 用于访问 const far 空间(存储类为 HCONST)。

例 子: int i;
long l;
l = FARRAY(long,0x8000)[i];

```

FARRAY(long,0x8000)[10] = 0x12345678;
#define DualPortRam FARRAY(int,0x24000)
DualPortRam[i] = 0x1234;
l = FCARRAY(long,0x18000)[5];

```

注意：绝对地址目标不能穿过 64KB 的段边界，例如不能访问一个起始地址为 0xFFFF8，有 10 个元素的 long 数组。

9.10.7 内部函数 INTRINS.H

内部函数的原型声明包含在头文件 INTRINS.H 中。

函数原型： `unsigned char _crol_(unsigned char val, unsigned char n);`
`unsigned int _irol_(unsigned int val, unsigned char n);`
`unsigned long _lrol_(unsigned long val, unsigned char n);`

再入属性： `reentrant/intrinsc`

功 能： `_crol_`、`_irol_` 和 `_lrol_` 将变量 `val` 循环左移 `n` 位，它们与 8051 单片机的“RL A”指令相关。这三个函数的不同之处在于参数和返回值的类型不同。

参 见： `_cror`、`_iror`、`_lror`

例 子： `#include <intrins.h>`

```

void tst_crol (void) {
    char a;
    char b;
    a = 0xA5;
    b = _crol_(a,3);      /* b now is 0x2D */
}

void tst_irol (void) {
    int a;
    int b;
    a = 0xA5A5;
    b = _irol_(a,3);      /* b now is 0x2D2D */
}

void tst_lrol (void) {
    long a;
    long b;
    a = 0xA5A5A5A5;
    b = _lrol_(a,3);      /* b now is 0x2D2D2D2D */
}

```

函数原型： `unsigned char _cror_(unsigned char val, unsigned char n);`
`unsigned int _iror_(unsigned int val, unsigned char n);`

```
unsigned long _lror_(unsigned long val, unsigned char n);
```

再入属性: reentrant/intrinsc

功 能: `_cror_`, `_iror_` 和 `_lror_` 将变量 `val` 循环右移 `n` 位, 它们与 8051 单片机的“RR A”指令相关。这三个函数的不同之处在于参数和返回值类型不同。

参 见: `_crol_`, `_lrol_`, `_lror_`

例 子: `#include <intrins.h>`

```
void tst_cror (void) {
    char a;
    char b;
    a = 0xA5;
    b = _cror_(a,1);      /* b now is 0xD2 */
}

void tst_iror (void) {
    int a;
    int b;
    a = 0xA5A5;
    b = _iror_(a,1);      /* b now is 0xD2D2 */
}

void tst_lror (void) {
    long a;
    long b;
    a = 0xA5A5A5A5;
    b = _lror_(a,1);      /* b now is 0xD2D2D2D2 */
}
```

函数原型: `void _nop_(void);`

再入属性: reentrant/intrinsc

功 能: `_nop_` 产生一个 8051 单片机的 NOP 指令, 该函数可用于 Cx51 程序中的时间延时。Cx51 编译器对程序中调用 `_nop_` 函数的地方, 直接产生一条 NOP 指令。

例 子: `#include <intrins.h>`

```
#include <stdio.h>          /* for printf */
void tst_nop (void) {
    P1 = 0xFF;
    _nop_ ();                /* delay for hardware */
    _nop_ ();
    _nop_ ();
    P1 = 0x00;
}
```

函数原型: `bit _testbit_(bit x);`

再入属性: `reentrant/intrinsc`

功 能: `_testbit_` 产生一条 8051 单片机的 JBC 指令, 该函数对字节中的一位进行测试。如果该位置位则函数返回 1, 同时将该位复位为 0, 否则返回 0。
`_testbit_` 函数只能用于可直接寻址的位, 不允许在表达式中使用。

例 子:

```
#include <intrins.h>
#include <stdio.h>          /* for printf */
void tst_testbit (void){
    bit test_flag;
    if (_testbit_ (test_flag))
        printf ("Bit was set\n");
    else
        printf ("Bit was clear\n");
}
```

9.10.8 变量参数表 STDARG.H

Cx51 编译器允许函数的参数个数和类型是可变的, 可使用简略形式 (记号为 "..."), 这时参数表的长度和参数的数据类型在定义时是未知的。头文件 `stdarg.h` 中定义了处理函数参数表的宏, 利用这些宏, 使程序可识别和处理变化的参数。在定义具有可变参数的函数时, 必须声明一个 `va_list` 型的指针, 用 `va_start` 将该指针初始化为指向该参数表, 用 `va_arg` 访问表中不同类型的参数, 对参数的访问结束后, 用 `va_end` 关闭参数表。

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(ap,type)`

再入属性: `reentrant`

功 能: 宏 `va_arg` 从 `ap` 指向的可变长度参数表中检索 `type` 类型的值。对于每一个参数可以只调用一次宏, 并且调用宏时必须按照参数表中参数的次序进行。对 `va_arg` 的第一次调用将返回在 `va_start` 宏中指定的 `v` 参数后的第一个参数。继续调用 `va_arg` 将返回剩下的后续参数。

参 见: `va_start`, `va_end`

例 子:

```
#include <stdarg.h>
#include <stdio.h>          /* for printf */
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);
}
}

```

函数原型: `void va_start (ap,v)`

再入属性: `reentrant`

功 能: 当用在一个具有可变长度参数表的函数中时, 宏 `va_start` 初始化 `ap` 参数, 为以后的 `va_arg` 和 `va_end` 宏所使用参数 `v` 必须是直接位于省略号所指定的可选参数前函数参数名。在使用宏 `va_arg` 进行存取前, 必须调用函数来初始化可变参数表。

参 见: `va_arg`, `va_end`

函数原型: `void va_end(ap)`

再入属性: `reentrant`

功 能: 该宏用于终止在 `va_start` 宏中已被初始化的可变长度参数表的指针 `ap`。关闭参数表, 结束对可变参数表的访问。

参 见: `va_arg`, `va_start`

9.10.9 全程跳转 SETJMP.H

头文件 `SETJMP.H` 中的函数可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: `#define _JBLEN 7`

`typedef char jmp_buf[_jblen]`

功 能: 规定由 `setjmp` 和 `longjmp` 所使用的用于保存和恢复程序环境的缓冲区

函数原型: `int setjmp(jmp_buf env);`

再入属性: `reentrant`

功 能: `setjmp` 将程序执行的当前环境状态信息存入变量 `env` 之中, 以便嵌套调用的低层函数使用 `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。

参 见: longjmp

函数原型: void longjmp(jmp_buf env, int retval);

再入属性: reentrant

功 能: longjmp 恢复先前调用 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。longjmp 和 setjmp 函数提供了一种执行非局部 goto 的方法,它们一般用于将执行控制传递给先前被调用的程序中的错误处理或恢复代码。只有用 volatile 属性声明的局部变量和函数参数才能被恢复。

参 见: setjmp

例 子: #include <setjmp.h>
#include <stdio.h> /* for printf */
jmp_buf env; /* jump environment (must be global) */
bit error_flag;
void trigger (void) {
 . . .
 /* put processing code here */
 . . .
 if (error_flag != 0) {
 longjmp (env, 1); /* return 1 to setjmp */
 } . . .
}
void recover (void) {
 /* put recovery code here */
}
void tst_longjmp (void) {
 . . .
 if (setjmp (env) != 0) { /* setjmp returns a 0 */
 printf ("LONGJMP called\n");
 recover ();
 }
 else {
 printf ("SETJMP called\n");
 error_flag = 1; /* force an error */
 trigger ();
 }
}

9.10.10 计算结构体成员的偏移量 STDDEF.H

头文件 STDDEF.H 包含了一个计算结构体成员偏移量的宏。

函数原型: int offsetof(structure, member);

功 能: `offsetof` 宏计算结构体成员 `member` 从结构体 `structure` 开始位置的偏移量, 并返回字节形式的偏移量值。

例 子: `#include <stddef.h>`

```
struct index_st {
    unsigned char type;
    unsigned long num;
    unsigned int len;
};
typedef struct index_st index_t;
void main (void) {
    int x, y;
    x = offsetof (struct index_st, len); /* x = 5 */
    y = offsetof (index_t, num); /* x = 1 */
}
```

第 10 章 Ax51 宏汇编器

Keil 公司出售的高级语言开发软件包中包括两套宏汇编器：用于传统 8051 单片机的 A51 宏汇编器和用于新一代扩展型 80C51 的 Ax51 宏汇编器，它是一种具有通用特性和用法的重定位宏汇编器，它与 Intel 公司的 MASM51 宏汇编器具有很好的兼容性，支持模块化编程，可以方便地与高级语言接口。实际上 Ax51 是 A51 的超集，一般来说对于 A51 所产生代码需要用连接定位器 BL51 进行连接定位，对于 Ax51 所产生的代码则需要扩展连接定位器 Lx51 进行连接定位。为方便起见下面的讨论中都使用了 Ax51，但是对于只适用于 Ax51 而不适用于 A51 的汇编控制命令会特别指出来。Ax51 宏汇编器支持汇编伪指令、宏处理指令、以及汇编控制命令，在 Windows 集成开发环境 μ Vision2 中，通过“Project 菜单/Options for Target 选项/Ax51 标签页”及“Listing 标签页”很容易设置各种汇编控制及输出列表命令。也可通过命令行引用 Ax51 宏汇编器，引用格式如下：

Ax51 文件名 [汇编控制命令]

其中，“Ax51”是宏汇编器调用命令。“文件名”是以“.A51”、“.ASM”或“.SRC”为扩展名的汇编语言源文件，文件名可连同路径一起给出。“汇编控制命令”是可选项，用于控制 Ax51 宏汇编器产生列表文件、目标文件以及其他一些目的。如果省略该选项，则按默认的汇编控制命令进行汇编，并在源文件名所在的路径上产生列表文件和目标文件。

例如，对当前目录下的源文件 SAMPLE.A51 进行汇编时，可按以下命令行调用 Ax51 宏汇编器：

Ax51 SAMPLE.A51 XREF PW(78)

其中，“XREF”是产生交叉参考列表的控制命令。“PW(78)”是控制列表文件宽度的控制命令。这时将在当前目录下产生列表文件 SAMPLE.LST 和目标文件 SAMPLE.OBJ。下面是列表文件 SAMPLE.LST 的内容：

LOC	OBJ	LINE	SOURCE
	000D	1	CR EQU 13
	000A	2	LF EQU 10
		3	
		4	EXTRN CODE(SEROUT)
		5	PUBLIC START
		6	SAMP SEGMENT CODE
		7	
----		8	RSEG SAMP

```

0000 759852          9      START:  MOV     SCON, #52H
0003 7588D2         10              MOV     TCON, #0D2H
0006 758D0D         11              MOV     TH1, #13
0009 D28E           12              SETB    TR1
000B 900000    F     13      LOAD:   MOV     DPTR, #MSG
000E E4            14      LOOP:   CLR     A
000F 93            15              MOVC    A, @A+DPTR
0010 120000    F     16              CALL   SEROUT
0013 B40A02         17              CJNE    A, #LF, MORE
0016 80F3           18              SJMP    LOAD
0018 E582           19      MORE:  MOV     A, DPL
001A 04            20              INC     A
001B F582           21              MOV     DPL, A
001D 50EF           22              JNC     LOOP
001F 0583           23              INC     DPH
0021 80EB           24              SJMP    LOOP
0023 4D455353       25      MSG:   DB      "MESSAGE", 13, 10
0027 4147450D
002B 0A
                                26
                                27
                                28      END

```

SYMBOL TABLE LISTING

```

-----
N A M E                T Y P E  V A L U E  A T T R I B U T E S
CR . . . . .          N NUMB  000DH    A
DPH. . . . .          D ADDR  0083H    A
DPL. . . . .          D ADDR  0082H    A
LF . . . . .          N NUMB  000AH    A
LOAD . . . . .        C ADDR  000BH    R  SEG=SAMP
LOOP . . . . .        C ADDR  000EH    R  SEG=SAMP
MORE . . . . .        C ADDR  0018H    R  SEG=SAMP
MSG. . . . .          C ADDR  0023H    R  SEG=SAMP
SAMP . . . . .        C SEG   002CH    REL=UNIT
SCON . . . . .        D ADDR  0098H    A
SEROUT . . . . .      C ADDR  -----  EXT
START. . . . .        C ADDR  0000H    R  SEG=SAMP
TCON . . . . .        D ADDR  0088H    A
TH1. . . . .          D ADDR  008DH    A
TR1. . . . .          B ADDR  0088H.6  A

```

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE. 0 WARNING(S), 0 ERROR(S)

列表文件中各部分的含义如下:

LOC: 用 16 进制数表示的指令代码首字节所占用的绝对或相对地址。

OBJ: 用 16 进制数表示的目标指令代码, 如果目标代码是可再定位的 (利用 BL51 或 Lx51 定位), 则在代码后面标以 “F” 标记, 并将再定位代码以数字 “0” 填充。

LINE: 用 10 进制数表示的源程序行号。

SOURCE: 源程序汇编指令行, 包括程序语句标号。

如果在调用 Ax51 宏汇编器时使用了 XREF 命令, 将在列表文件尾部附加一个交叉参考符号表, 表中各部分的含义如下:

NAME 部分: 表示符号的名字。

TYPE 部分: 表示符号为下列类型之一:

N NUMB: 无类型符号

D ADDR: DATA 地址段符号

C ADDR: CODE 地址段符号

X ADDR: XDATA 地址段符号

B ADDR: BIT 类型符号

.....: 未知类型

SEG: 段符号

REG: 寄存器符号

VALUE 部分: 表示符号的数值或地址, 以 16 进制数显示。数值后面的字符 “A” 表明是绝对值, 字符 “R” 表明是可再定位值。

ATTRIBUTES/REFERENCE 部分: 包含所用符号的标志和行号, 外部符号表示为 “EXT”, 公共符号表示为 “PUB”, 对于可再定位符号, 用 “SEG=Segment_Name” 指出定义该符号的段, 段符号的再定位类型表示为: “REL=Relocation_Type”。再定位类型有: UNIT、PAGE、INBLOCK 和 BITADDRESSABLE。行号后面的符号 “#” 指出源程序中定义该符号的行。

10.1 符号与表达式

8051 单片机汇编语言程序由若干条 8051 指令行组成, 8051 指令行的一般形式为:

[标号:]助记符 [操作数 1] [, 操作数 2] [, 操作数 3] [: 注释]

其中, “标号” 是可选项, 它可用来表示程序的转移地址, 同时可方便程序的调试。“助记符” 是 8051 单片机的指令助记符。“操作数 1~3” 是可选项, 它依赖于不同的 8051 指令助记符。有些指令不需要操作数, 有些指令则需要 1~3 个操作数。操作数可以是数字、符号或地址。数字可以使用 10 进制、16 进制、8 进制或 2 进制数。10 进制数以字符 “D” 为后缀, 16 进制数以字符 “H” 为后缀, 8 进制数以字符 “O” 为后缀, 2 进制数以字符 “B” 为后缀, 省略后缀时则默认为 10 进制数。立即数的前面须冠以符号 “#”。

Ax51 宏汇编器允许使用符号来表示数值、地址和寄存器名等,以增加程序的可读性。符号名最长为 31 个字符,第一个字符为英文字母 A~Z 或 a~z、符号“_”或“?”,后续字符可为上述字符或数字 0~9。标号也是一种符号。一些符号已经预定义为 Ax51 的保留字,用户不能对它们重新定义。这些符号及其意义如表 10-1 所示。

表 10-1 Ax51 宏汇编器的保留字

保留字	意 义
A	累加器
R0~R7	当前工作寄存器(共有 4 个寄存器组)
DPTR	16 位数据指针,用于访问内部或外部地址空间的数据
PC	16 位程序计数器,其值为下一条将被执行的指令的地址
C	进位标志
AB	用于乘除操作的寄存器对
AR0~AR7	表示当前工作寄存器的绝对地址,其值取决于指令所选择的工作寄存器组
PRO, PRI (仅用于 80C51Mx)	80C51Mx 的通用指针寄存器,用于访问整个 16M 存储器空间 PRO 由 R1, R2, R3 组成, PRI 由 R5, R6, R7 组成
EPTR (仅用于 80C51Mx)	80C51Mx 提供的一个附加数据指针,用于访问整个 16M 存储器空间

符号“\$”是一个特殊的汇编符号,它表示当前段的当前地址计数器。每执行一条指令,地址计数器的值也随之增加。如果当前段发生变化,地址计数器也将自动改变到新段。例如,下面的一条指令表示跳转到标号 HALT 处:

HALT: SJMP \$

Ax51 宏汇编器中的操作符有:算术运算符、逻辑运算符、关系运算符、存储类操作符以及其他操作符,如表 10-2 所示。

表 10-2 Ax51 宏汇编器的操作符

	操作符	例 子	意 义
算 术 运 算 符	+, -	+5, -4	数或表达式的符号
	+, -	2+10-5	加减运算
	*	1000H*2	乘法运算
	/	17/4	除法运算
	MOD	18MOD4	取模运算
	()	8+(12-5)	改变运算顺序
逻 辑 运 算 符	NOT	NOT 5	取反
	SHL, SHR	2 SHL 3, 8 SHR 4	左、右移位
	AND	12H AND 0F0H	逻辑与运算
	OR	12H OR 177	逻辑或运算
	XOR	12H XOR 14	逻辑异或运算

续表

	操作符	例 子	意 义
关 系 运 算 符	>=	55>=17	大于等于
	<=	23<=44	小于等于
	<>	32<>54	不等于
	=	12H=18	等于
	<	21H<32H	小于
	>	55>17	大于
存 储 类 操 作 符	BIT	BIT 表达式	指定表达式为 BIT 类
	CODE	CODE 表达式	指定表达式为 CODE 类
	CONST	CONST 表达式	指定表达式为 CONST 类
	DATA	DATA 表达式	指定表达式为 DATA 类
	ECODE	ECODE 表达式	指定表达式为 ECODE 类
	EDATA	EDATA 表达式	指定表达式为 EDATA 类
	IDATA	IDATA 表达式	指定表达式为 IDATA 类
	HCONST	HCONST 表达式	指定表达式为 HCONST 类
	HDATA	HDATA 表达式	指定表达式为 HDATA 类
其 他	XDATA	XDATA 表达式	指定表达式为 XDATA 类
	HIGH	HIGH 1234	选择操作数的高字节
	LOW	LOW 1234	选择操作数的低字节

Ax51 宏汇编器的操作符具有如表 10-3 所示的优先级，一个表达式中存在多个不同优先级的操作符时将按它们的优先级顺序进行运算，如果一个表达式中各个操作符都具有相同的优先级，则按从左到右的顺序进行运算。

表 10-3 Ax51 宏汇编器中运算符的优先级

优先级	运算符	意 义
1	()	括号
2	NOT、HIGH、LOW	取反、取高、低地址
3	存储类操作符	指定表达式的存储类
4	+, -	正、负号
5	*, /, MOD	乘、除、取模运算
6	+, -	加、减运算
7	SHL、SHR	左、右移位
8	AND、OR、XOR	逻辑与、逻辑或、逻辑异或
9	>=、<=、=、<、>	大于等于、小于等于、等于、小于、大于

表达式由操作符与运算符组成，表达式根据其所使用的操作数可分为如表 10-4 所示的段类型。

表 10-4 表达式的段类型

表达式类	说 明
N NUMB	无类型数
C ADDR	CODE 地址符号
D ADDR	DATA 地址符号
X ADDR	XDATA 地址符号
B ADDR	BIT 地址符号
CO ADDR	CONST 地址符号
CE ADDR	ECODE 地址符号
ED ADDR	EDATA 地址符号
HD ADDR	HDATA 地址符号
HC ADDR	HCONST 地址符号

大多数表达式都被归类于“NUMBER”，即没有段类型。有些操作允许使用有类型的表达式，这些表达式需要遵守如下规则：

- 单目运算的结果具有与其操作符相同的段类型；
- 双目运算（除+、-之外）结果均归类于“NUMBER”，即没有段类型；
- 对于+、-运算，只有当其中一个操作符具有段类型时，运算结果才具有与其相同的段类型。如果两个操作符均无段类型，则结果为“NUMBER”类型。

以上规则说明一个类型值（如地址符号）与一个数字相加减，其结果将具有相同的类型。无类型（即“NUMBER”类型）的表达式可以在任何地方使用，具有其他类型的表达式则只能在该类型为有效的地方使用。

例子：

```

data_address - 10                : 结果为 data_address 值
10 + edata_address              : 结果为 edata_address value 值
(data_address - data_address)    : 结果为无类型的数
code_address + (data_address - data_address) : 结果为 code_address 值

```

可再定位表达式包含可再定位符号或外部参考名，对于这类表达式由于不知道其可再定位段的最终地址，因此宏汇编器只能进行部分计算，其最终计算结果需要由连接定位器来完成。一个可再定位表达式通常只包含一个可再定位符号，有时也可包含其他操作符与操作数。可再定位符号可以通过加、减一个常数值来进行修改。

例子：

```

可再定位符号 + 绝对表达式
可再定位符号 - 绝对表达式
绝对表达式 + 可再定位符号

```

可再定位表达式可分为简单可再定位表达式和扩展可再定位表达式。

简单可再定位表达式中允许包含在可再定位段中定义的符号，但不允许有段和外部符号。简单可再定位表达式可以用于 8051 的汇编指令以及伪指令 ORG、EQU、SET、DB、

DW 等的操作数。

例子：

可再定位符号 1 + 绝对符号 1 * 10

可再定位符号 2 - 绝对符号 1

可再定位符号 1 + (可再定位符号 2 - 可再定位符号 3)

下面的简单可再定位表达式无效：

(可再定位符号 1 + 绝对符号 1) * 10 ; 可再定位值不允许进行乘法运算

可再定位符号 1 + 可再定位符号 2 ; 可再定位符号不允许相加

LOW (可再定位符号 1) + 绝对符号 1 ; LOW 只能用于最终可再定位表达式

扩展可再定位表达式具有与简单可再定位表达式相同的规则，但这类表达式还允许有段和外部符号，并且只能用在可以产生目标代码的运算中，例如，用做 8051 的汇编指令以及伪指令 DB、DW 的操作数。

例子：

可再定位符号 1 + 绝对符号 1 * 10

外部符号 1 - 绝对符号 1

LOW (可再定位符号 1 + 绝对符号 1)

例 10.1：表达式示例。

EXTRN CODE (CLAB) ; CODE 空间入口

EXTRN DATA (DVAR) ; DATA 空间变量

MSK EQU 0F0H ; 符号定义

VALUE EQU MSK - 1

LVAL EQU 12345678H

?PR?FOO SEGMENT CODE
RSEG ?PR?FOO

```
ENTRY:  MOV A, #40H          ; 装入立即数到 A
        MOV R5, #VALUE      ; 装入符号常数到 R5
        MOV R3, #(0x20 AND MASK) ; 计算的例子
        MOV R7, #LOW (VALUE + 20H)
        MOV R6, #1 OR (MSK SHL 4)
        MOV R0, DVAR+20      ; 装入地址 DVAR+20 的内容到 R0
        MOV R1, #LOW (CLAB+10) ; 装入地址 CLAB+10 的低字节到 R1
        MOV R5, 80H          ; 装入 DATA 地址 80H (= SFR P0) 到 R5
        MOV R5, EDATA 80H    ; 装入 EDATA 地址 80H 到 R5
        SETB 30H.2           ; 置 1 位地址 30H.2 (长地址)
        SETB 20H.2           ; 置 1 位地址 20H.2 (短地址)
```

END

10.2 汇编伪指令

汇编伪指令是一种控制汇编进程的特殊控制符号，其功能是改变汇编器的状态并将一些必要的信息（如段定义等）加入到目标文件中。Ax51 允许采用汇编伪指令进行符号定义、保留和初始化存储器空间、控制程序连接、控制汇编状态和段选择。Ax51 中的符号定义汇编伪指令可分为如下几类。

- 段控制指令：
 - 一般段：SEGMENT, RSEG
 - 绝对段：CSEG, DSEG, BSEG, ISEG, XSEG
- 符号定义指令：
 - 一般符号：EQU, SET
 - 地址符号：BIT, CODE, DATA, IDATE, XDATA
 - SFR 符号：sfr, sfr16, sbit
 - 文本替换：LIT
- 存储器初始化指令：
 - DB, DW, DD
- 存储器保留指令：
 - DBIT, DS, DSB, DSW, DSD
- 过程声明指令：
 - PROC/ENDP, LABEL
- 程序连接指令：
 - PUBLIC, EXTRN/EXTERN, NAME
- 地址控制指令：
 - ORG, USING
- 其他指令：
 - END, __ERROR__

10.2.1 段控制指令

所谓段是宏汇编器根据汇编语言源程序生成的一段存储器代码或数据。8051 单片机包含有几个不同的存储器空间，通过段控制指令可以将程序代码、常数和变量存放在指定的存储器空间之内。段控制指令分为可再定位段控制和绝对段控制指令。分述如下。

1. SEGMENT 指令

SEGMENT 指令用来声明一个可再定位段，指令格式如下：

段名 SEGMENT 存储器类型 [段再定位类型] [段分配类型]

其中，“段名”用于声明所使用的段，段名由后面的“RSEG”指令所引用，在表达式中也可以使用段名来表示复合段的基址或起始地址。

“存储器类型”用于指定所声明的段的存储器地址空间。基本存储器类型如表 10-5 所示。

表 10-5 基本存储器类型及其意义

基本存储器类型	说 明
BIT	可位寻址地址空间: 20H~2FH
CODE	代码空间
CONST	通过 MOVC 指令访问的常数存储器空间
DATA	片内直接寻址的 RAM 空间: 0~7FH 及 SFR 地址
EDATA	扩展片内 RAM 空间
ECONST	用于存放常数的扩展片内 RAM 空间
IDATA	片内间接寻址的 RAM 空间: 0~0FFH
ECODE	Philips80C51Mx 的全部程序存储器空间
HCONST	Philips80C51Mx 的全部常数存储器空间
HDATA	Philips80C51Mx 的全部数据存储器空间
XDATA	通过 MOVX 指令访问片外数据存储器空间

除以上基本存储器类型外, 对于 Ax51 (A51 不能用) 还可以使用一种用户定义类型, 方法是一对单引号括起来的类型名, 如 ‘XDATA_FLASH’, 这种方法可以定义多个段, 然后采用连接定位器将它们定位在一个指定的物理地址空间。

“段再定位类型”是可选项, 用于定义将由连接定位器决定的段位置。有效的再定位类型如表 10-6 所示。

表 10-6 段再定位类型及其意义

段再定位类型	说 明
AT 地址	在指定的地址处定义一个绝对段
BITADDRESSABLE	指定一个将被定位于 BIT 区域(20H~2FH)的段。它只能用于最大长度限制为 16 个字节的数据段
INBLOCK	指定一个必须包含在一个 2048 字节块中的段, 它只能用于 CODE 段
INPAGE	指定一个必须包含在一个 256 字节块中的段, 它只能用于 CODE 和 XDATA 段
OFFS 偏移量	指定一个绝对段, 该段将位于存储器类型加偏移量所形成的起始地址, 这种再定位类型的优点是起始地址可以通过 Lx51 连接定位器的控制命令 “CLASSES” 进行调整
OVERLAYABLE	指定一个可覆盖的段。该段可与其他段交迭, 其段名必须符合 Cx51 编译器规定的段名规则
INSEG	指定一个必须包含在一个 64K 字节中的段

“段分配类型”也是可选项, 用于定义将由连接定位器使用的分配选项。有效的段分配类型如表 10-7 所示。

表 10-7 段分配类型及其意义

段分配类型	说 明
BIT	指定一个段的位队列，为 BIT 存储器类型中所有段的默认值
BYTE	指定一个段的字节队列，为除了 BIT 段之外所有其他段的默认值
WORD	为段指定一个字队列
DWORD	为段指定一个双字队列
PAGE	指定一个起始地址必须位于 256 字节边界的段
BLOCK	指定一个起始地址必须位于一个 2048 字节块边界的段
SEG	指定一个起始地址必须位于一个 64K 字节边界的段

例子：

```
IDS      SEGMENT IDATA      ; 声明一个名为 IDS 的段，其存储器类型为
                                ; IDATA。
MYSEG    SEGMENT CODE AT 0FF2000H ; 声明一个 CODE 段，段名为 MYSEG，起始地址
                                ; 为 0FF2000H。
HDSEG    SEGMENT HDATA INSEG DWORD ; 声明一个 HDATA 段，段名为 HDSEG，并以双
                                ; 字队列形式位于一个 64K 字节的段中。
XDSEG    SEGMENT XDATA PAGE    ; 声明一个 XDATA 段，段名为 XDSEG，其起始
                                ; 地址位于 256 字节的边界。
HCSEG    SEGMENT HCONST SEG    ; 声明一个 HCONST 段，段名为 HCSEG，其起
                                ; 始地址位于 64K 字节的边界。
```

2. RSEG 指令

RSEG 指令用于选择一个已经在前面用 SEGMENT 指令定义过了的可再定位作为当前段，指令格式如下：

RSEG 段名

其中，段名必须是前面已经用 SEGMENT 指令定义过了的可再定位段。

例子：

```
MYPROG   SEGMENT           CODE      ; 声明一个 CODE 段，段名为 MYPROG。
          RSEG              MYSEG     ; 选择 MYSEG 段作为当前段，
          MOV               A, #00H
          MOV               P0, A
```

3. BSEG、CSEG、DSEG、ISEG 和 XSEG 指令

分别用于选择绝对位寻址数据段、绝对代码段、绝对片内直接寻址数据段、绝对片内间接寻址数据段和绝对片外数据段。指令格式如下：

BSEG [AT 绝对地址表达式]

CSEG [AT 绝对地址表达式]

DSEG [AT 绝对地址表达式]

ISEG [AT 绝对地址表达式]

XSEG [AT 绝对地址表达式]

其中, 括号内是可选项, 用来指定当前绝对段的起始地址, 绝对地址表达式中不允许出现前向参考量。如果指令中使用选项 AT 指定了一个地址, Ax51 汇编器将结束上次的段并产生一个起始于指定地址的新段。如果指令中未用 AT 选项指定地址, 则当存在当前段时, 汇编器将忽略该指令; 当不存在当前段时, 汇编器将产生一个绝对地址为 00H 的新段。

例子:

```

                BSEG      AT 30H      ; 选择绝对位段, 起始地址为 30H
DEC_FLAG:      DBIT 1
INC_FLAG:      DBIT 1
                CSEG AT 100h          ; 选择绝对代码段, 起始地址为 100h
PARITY_TAB:    DB 00h
                DB 01h
                DB 01h
                DB 00h
                .
                .
                .
                DB 01h
                DB 00h
                DSEG AT 40h           ; 选择绝对片内直接数据段, 起始地址为 40h
TMP_A:         DS 2
TMP_B:         DS 4
                ISEG AT 40h          ; 选择绝对片内间接数据段, 起始地址为 40h
TMP_IA:        DS 2
TMP_IB:        DS 4
                XSEG AT 1000h        ; 选择绝对片外数据段, 起始地址为 1000h
OEMNAME:       DS 25
PRDNAME:       DS 25
VERSION:       DS 25

```

10.2.2 符号定义指令

符号定义指令可生成用来表示寄存器、数值或地址的各种符号。

1. EQU、SET 指令

EQU 和 SET 指令用于将一个数值或寄存器名赋给一个指定的符号名, 指令格式如下:

```

符号名 EQU  表达式
符号名 EQU  寄存器名
符号名 SET  表达式
符号名 SET  寄存器名

```

其中, “表达式”必须是一个简单再定位表达式, 并且不能包含前向参考量。“寄存器

名”可为 A、R0~R7。

经过 EQU 和 SET 指令赋值的符号可在程序的任何地方使用，已经由 EQU 指令定义了的符号不能再用 SET 指令定义，同样已经由 SET 指令定义了的符号不能再用 EQU 指令定义。但是用 SET 指定可以重复定义同一个符号。

例子：

```

LIMIT EQU 1200
VALUE EQU LIMIT-200+ 'A'
SERIL EQU SBUF
ACCU EQU A
COUNT EQU R5
VALUE SET 100
VALUE SET VALUE-2
COUNTER SET R1
TEMP SET COUNTER
TEMP SET VALUE+COUNTER

```

2. BIT、CODE、DATA、IDATA、XDATA 指令

这些指令将一个确定的地址值赋给指定的符号名，已经定义过的符号名不允许重复定义。指令格式如下：

符号名 BIT 位地址
 符号名 CODE 代码地址
 符号名 DATA 片内直接数据地址
 符号名 IDATA 片内间接数据地址
 符号名 XDATA 片外数据地址

其中，“位地址”范围为 20H~2FH，以及可位寻址特殊功能寄存器中可寻址位。“代码地址”范围为 0000H~0FFFFH。“片内直接数据地址”范围为 00H~7FH 或特殊功能寄存器的直接地址。“片内间接数据地址”范围为 00H~0FFH。“片外数据地址”范围为 0000H~0FFFFH。

例子：

DATA_SEG	SEGMENT BITADDRESSABLE		；声明一个可位寻址段
	RSEG	DATA_SEG	；作为当前再定位段
CTRL:	DS	1	；定义一个字节变量
ALARM	BIT	CTRL.0	；定义可再定位字节中的一位
SHUT	BIT	ALARM+1	；下一位
ENABLE_FLAG	BIT	60H	；定义绝对位
DONE_FLAG	BIT	24H.2	；定义绝对位 a
P1_BIT2	EQU	90H.2	；定义特殊功能寄存器中的一位
RESTART	CODE	00H	
INTVEC_0	CODE	RESTART + 3	
INTVEC_1	CODE	RESTART + 0BH	

INTVEC_2	CODE	RESTART + 1BH	
SERBUF	DATA	SBUF	; 重新定义 SBUF 符号
RESULT	DATA	40H	
RESULT2	DATA	RESULT + 2	
PORT1	DATA	90H	; 定义 P0 口
BUFFER	IDATA	60H	
BUF_LEN	EQU	20H	
BUF_END	IDATA	BUFFER + BUF_LEN - 1	
XSEG1	SEGMENT	XDATA	; 声明一个外部数据段
	RSEG	XSEG1	; 作为当前可再定位段
DTIM:	DS 6		; 保留 6 个字节
TIME	XDATA	DTIM + 0	
DATE	XDATA	DTIM + 3	

3. sfr、sfr16、sbit 指令

sfr、sfr16 和 sbit 指令与 Cx51 编译器完全兼容，用于定义特殊功能寄存器及其可寻址位。esfr 指令用于定义 Philips80C51Mx 单片机的扩展特殊功能寄存器，esfr 指令只能用于 Ax51 宏汇编器（A51 不能用）。指令格式如下：

```
sfr      特殊功能寄存器名 = 地址
sfr16    特殊功能寄存器名 = 地址
sbit     符号名 = 位地址
esfr     特殊功能寄存器名 = 地址
```

其中，“地址”值的范围为：80H~0FFH，“位地址”可采用“字节地址^位位置”或“特殊功能寄存器名^位位置”的形式。注意，sfr16 指令仅用于与 Cx51 编译器兼容，Ax51 宏汇编器将忽略由该指令定义的符号。

例子：

```
sfr P0 = 0x80
sfr P1 = 0x90
sbit P0_0 = P0^0
sbit P1_1 = 0x90^1
esfr MXCON = 0xFF;      /* Philips 80C51MX 扩展 SFR */
sfr16 T2 = 0xCC;        /* Ax51 将忽略该符号 */
```

4. LIT 指令（仅用于 Ax51）

LIT 指令用于文字替代，指令格式如下：

```
符号    LIT    '文字串'
符号    LIT    '文字串'
```

其中“文字串”是将要指定符号所替代的文字或数字表达式，它不能包含前向参考。

例子：

汇编之前的源程序如下：

```
$INCLUDE (REG51.INC)
```

```

REG1    LIT      'R1'
NUM      LIT      'A1'
DBYTE    LIT      "DATA BYTE"
FLAG     LIT      'ACC.3'
?PR?MOD  SEGMENT CODE
        RSEG      ?PR?MOD
        MOV REG1,#5
        SETB FLAG
        JB FLAG,LAB_{NUM}
        PUSH DBYTE 0
LAB_{NUM}:
        END

```

汇编之后的列表文件如下:

```

          1      $INCLUDE (REG51.INC)
+1 80 +1  $RESTORE
          81
          82      REG1    LIT      'R1'
          83      NUM      LIT      'A1'
          84      DBYTE    LIT      "DATA BYTE"
          85      FLAG     LIT      'ACC.3'
          86
-----  87      ?PR?MOD SEGMENT      CODE
-----  88                      RSEG          ?PR?MOD
          89
000000 7E1005  90                      MOV R1, #5
000003 D2E3    91                      SETB ACC.3
000005 20E300 F 92                      JB ACC.3, LAB_A1
000008 C000    93                      PUSH DATA BYTE 0
00000A        94      LAB_A1:
          95
          96                      END

```

10.2.3 存储器初始化指令

Ax51 中的存储器空间初始化指令可用来在存储器空间内初始化字、字节或位单元,存储器空间始于当前地址的绝对段和当前偏移的再定位段。初始化存储器空间指令有 3 个,分述如下。

1. DB 指令

DB 指令以给定表达式值的字节形式初始化代码空间。指令格式如下:

[标号:] DB 表达式, 表达式,

表达式表中可包含符号、字符串、或表达式等项,各个项之间用逗号隔开,字符串应

用引号括起来。括号内的标号是可选项，如果使用了标号，则标号的值将是表达式表中第一个字节的地址。DB 指令必须位于 CODE 段之内，否则将会发生错误。

例子：

```
REQUEST:    DB  'ENTER ANY KEY TO CONTINUE', 0
TABLE:      DB  0,1,8,"A","O",LOW(TABLE),';'
ZERO:       DB  0,""
CASE_TAB:   DB  LOW(REQUEST),LOW(TABLE),LOW(ZERO)
```

2. DW 指令

DW 指令以给定表达式的值的双字节形式（16 位值）初始化代码空间。指令格式如下：

[标号:] DW 表达式, 表达式,

表达式表中可包含符号、字符串或表达式等项，各个项之间用逗号隔开。字符串应用引号括起来，并且一个字符串最多只能包含两个字符。括号内的标号是可选项，如果使用了标号，则标号的值将是表达式表中第一个字节的地址。如果 DW 指令不在 CODE 段内将会发生错误。

例子：

```
TABLE:      DW  TABLE, TABLE+10, ZERO
ZERO:       DW  0, "","", 'AB'
CASE_TAB:   DW  CASE0, CASE1, CASE2, CASE3, CASE4
            DW  $
```

3. DD 指令（仅用于 Ax51）

DD 指令以 32 位双字的值初始化代码存储器。指令格式如下：

[标号:] DD 表达式, 表达式,

DD 指令只能用于代码或常数段中，如果用于其他段将会产生错误。

```
TABLE:      DD  TABLE, TABLE + 10, ZERO
            DD  $
ZERO:       DD  0
LONG_VAL:   DD  12345678H, 0FFFFFFFFH, 1
```

10.2.4 存储器保留指令

Ax51 中的保留存储器空间指令可用来在存储器空间内保留字、字节或位单元，保留空间始于当前地址的绝对段和当前偏移的再定位段。保留存储器空间指令共有 5 个，分述如下。

1. DBIT 指令

DBIT 指令在内部数据区的 BIT 段内以位为单位保留存储器空间。指令格式如下：

[标号:] DBIT 数值表达式

数值表达式中不能包含前向参考量、可再定位或外部符号。DBIT 指令使 BIT 段的地

址计数器增加表达式结果之值，地址以位为单位增加。括号内的标号是可选项，使用了标号时，标号值将是保留区的第一个位地址。

例子：

```
RSEG    BIT_SEG1          ; 选择 BIT 段
ORG     22H               ; 定义起始地址
FLAG_ARE: DBIT  8         ; 在 BIT 段保留 8 个位，FLAG_ARE 为位地址"22H"
FLAG1:   DBIT  2         ; 在 BIT 段保留 2 个位，FLAG1 为位地址"2AH"
```

2. DS 指令

DS 指令以字节为单位在内部或外部数据区保留存储器空间。指令格式如下：

[标号:] DS 数值表达式

数值表达式中不能包含前向参考量、可再定位或外部符号。DS 指令使当前数据段的地址计数器增加表达式结果之值，地址计数器与表达式结果之和不能超过当前地址空间的限制。括号内的标号是可选项，使用了标号时，标号值将是保留区的第一个字节地址。

例子：

```
GAP:     DS  (($+16) AND 0FFF0H)-$
          DS  20
TIME:    DS  8
```

3. DSB 指令（仅用于 Ax51）

DSB 指令用于保留指定字节数的存储器空间。指令格式如下：

[标号:] DSB 数值表达式

数值表达式中不能包含前向参考量、可再定位或外部符号。DSB 指令使当前数据段的地址计数器增加表达式结果之值，地址计数器与表达式结果之和不能超过当前地址空间的限制。括号内的标号是可选项，使用了标号时，标号值将是保留区的第一个字节地址。

```
DAY:     DSB 1
MONTH:   DSB 1
HOUR:    DSB 1
```

4. DSW 指令（仅用于 Ax51）

DSW 指令用于保留指定字数的存储器空间。指令格式如下：

[标号:] DSW 数值表达式

数值表达式中不能包含前向参考量、可再定位或外部符号。DSW 指令使当前数据段的地址计数器增加表达式结果之值，地址计数器与表达式结果之和不能超过当前地址空间的限制。括号内的标号是可选项，使用了标号时，标号值将是保留区的第一个字节地址。

```
YEAR:     DSW 1
DAYinYEAR: DSW 1
```

5. DSD 指令（仅用于 Ax51）

DSD 指令用于保留指定双字数的存储器空间。指令格式如下：

[标号:] DSD 数值表达式

数值表达式中不能包含前向参考量、可再定位或外部符号。DSD 指令使当前数据段的地址计数器增加表达式结果之值，地址计数器与表达式结果之和不能超过当前地址空间的限制。括号内的标号是可选项，使用了标号时，标号值将是保留区的第一个字节地址。

```
SEC_CNT:      DSD 1
```

```
LONG_ARR:     DSD 1
```

10.2.5 过程声明指令（仅用于 Ax51）

Ax51 宏汇编器提供过程操作来完成子程序功能。过程可以被在线执行，也可以跳转到过程或调用过程，推荐采用调用过程。

1. PROC/ENDP 指令（仅用于 Ax51）

PROC 和 ENDP 指令用将为一段汇编指令定义为一段过程。对于 Philips80C51Mx 单片机来说，过程可以为 NEAR 或 FAR 类型，对于 NEAR 类型过程可以采用 LCALL 或 ACALL 进行调用，对于 FAR 类型过程则需要采用 ECALL 指令进行调用。与 C 语言的函数不同，汇编语言程序中的过程没有局部标号。PROC/ENDP 指令格式如下：

```
过程名      PROC      [类型]
```

```
      ; 汇编语言指令
```

```
      :
```

```
      :
```

```
      :
```

```
过程名      ENDP
```

其中，“类型”用于规定所定义过程的类型，如表 10-8 所示。

表 10-8 过程的类型

类 型	说 明
无	默认为 NEAR
NEAR	定义一个 NEAR 类型过程，采用 LCALL 或 ACALL 指令调用
FAR	定义一个 FAR 类型过程，采用 ECALL 指令调用

对于需要从不同的 64K 字节段中调用的过程必须定义为 FAR 类型，一个过程通常以汇编指令 RET 结束，它会被自动地转换成合适的机器指令，对于 NEAR 过程，它被转换为机器指令 RET，对于 FAR 过程，它被转换为 ERET 指令。

例子：

```
P100      PROC      NEAR
          RET              ; near return
ENDP

P200      PROC      FAR
```

```

                RET                ; far return (ERET)

ENDP

P300          PROC          NEAR
                CALL P100      ; ICALL
                CALL P200      ; ECALL
                RET            ; near return
                ENDP

                END

```

2. LABEL 指令（仅用于 Ax51）

LABEL 用于定义一个程序标号，所谓标号其实就是程序段中的一个符号地址。标号后面可以接一个冒号“:”，也可以不接。标号继承了当前活动代码段的属性，因此它不能在程序代码段之外使用。指令格式如下：

标号名[:] **LABEL** [类型]

其中，“类型”为可选项，其意义如表 10-8 所示。一个 **NEAR** 类型的标号可以从当前 64K 字节程序段中进行调用，如果要从不同的 64K 字节程序段中调用的话，则必须定义一个 **FAR** 类型的标号。

例子：

```

                RSEG ECODE_SEG1 ; 选择 ECODE 段
ENTRY:         LABEL FAR      ; 定义一个 FAR 类型标号
                RSEG ECODE_SEG2 ; 选择另一个 ECODE 段
                EJMP ENTRTY    ; 在不同的 64KB 段之间跳转

```

10.2.6 程序连接指令

Ax51 中控制程序连接的伪指令可用来标明当前程序模块、当前模块中需要使用的外部符号名以及可被其他模块使用的公共符号名，从而允许连接定位器将几个不同的程序模块连接成一个绝对模块。控制程序连接的汇编伪指令共有 3 个，分述如下。

1. PUBLIC 指令

PUBLIC 指令用于声明可被其他模块使用的公共符号名。指令格式如下：

PUBLIC 符号[,符号[,...]]

PUBLIC 指令后面可指定多个符号名，各个名字之间用逗号隔开。每个符号名必须已经在当前程序模块内定义过。允许指定前向参考量，但寄存器和段符号不能被指定为 **PUBLIC**。

例子：

```

PUBLIC PUT_CRLF, PUT_STRING, PUT_EOS
PUBLIC ASCBIN, BINASC
PUBLIC GETTOKEN, GETNUMBER

```

2. EXTRN/EXTERN 指令

EXTRN/EXTERN 指令是与 PUBLIC 指令相配套的,如果要使用其他模块中用 PUBLIC 指令声明了的公共符号,则必须采用 EXTRN/EXTERN 指令将这些公共符号列出来。指令格式如下:

EXTRN 存储器类型: 类型 (符号, 符号, ……)

EXTERN 存储器类型: 类型 (符号, 符号, ……)

其中,“存储器类型”可为 BIT、CODE、CONST、DATA、EBIT、ECONST、EDATA、ECODE、HDATA、HCONST、IDATA、XDATA 或 NUMBER 等。“类型”可为 BYTE、WORD、DWORD、NEAR 或 FAR。

EXTRN/EXTERN 指令可以出现在源程序中的任何地方,如果有多个符号,可以用括号扩起来并将每个符号用逗号隔开。

例子:

```
EXTRN    CODE(PUT_CRLF),DATA (BUFFER)
EXTRN    CODE(BINASC,ASCBIN)
EXTRN    NUMBER(TABLE_SIZE)
EXTERN   CODE: FAR    (main)
EXTRN    EDATA: BYTE  (VALUE, CONST)
EXTRN    HCONST: DWORD (LIMIT)
```

3. NAME 指令

NAME 指令用来标明当前程序模块。指令格式如下:

NAME 目标模块名

其中,“目标模块名”最多可包含 40 个字符,第一个字符不能是数字。需要注意的是 NAME 所标明的是模块名而不是文件名,每一个模块只允许有一个模块名,如果源程序中没有给出模块名,则以不带扩展名的文件名作为模块名。

例子:

```
NAME DATA_COMPRESSOR
NAME PARSEMODULE
```

10.2.7 地址控制指令

地址控制指令可用来设定程序的起始地址以及选择 8051 单片机的工作寄存器组。

1. ORG 指令

ORG 指令用来改变当前活动段的地址计数器,设定一个新的程序起始地址。指令格式如下:

ORG 表达式

其中,“表达式”必须是绝对或简单再定位表达式。表达式中不允许有前向参考量,并且只能使用当前段中的绝对地址或符号。

每当遇到一个 ORG 指令,宏汇编器都会计算其中表达式的值,并将其作为当前段的

地址计数器的值。如果当前段是绝对段，表达式的结果值作为地址计数器的绝对地址值。如果当前段是再定位段，则表达式的结果值作为地址计数器的地址偏移量。

用 **ORG** 指令可以改变地址计数器的值，但不会产生一个新的段，因此程序段中可能会存在地址间隙。在绝对段中使用 **ORG** 指令时，表达式的结果值不能小于绝对段的基地址。

例子：

```
ORG 100H
ORG RESTART
ORG EXIT1
ORG ($+16) AND 0FFF0H
```

2. USING 指令

USING 指令用于选择 8051 的工作寄存器组。指令格式如下：

USING 表达式

其中，“表达式”的值必须为 0~3，默认值为 0。**USING** 指令本身并不切换寄存器组，而是在目标代码中标记出所选择的寄存器组以供连接定位器使用。Ax51 中预定义符号 **ARn**($n=0\sim7$) 的值是使用 **USING** 指令选择的工作寄存器组中 **Rn**($n=0\sim7$) 的地址，必须先使用 **USING** 指令，再使用 **ARn** 符号。

例子：

```
USING 3
PUSH AR2          ; 将第 3 组工作寄存器中的寄存器 R2 压入堆栈
USING 1
PUSH AR2          ; 将第 1 组工作寄存器中的寄存器 R2 压入堆栈
```

10.2.8 其他指令

1. END 指令

END 指令用来控制结束汇编。在每个汇编语言程序的最后一行必须有一条 **END** 指令，并且 **END** 指令只能出现一次，如果程序中使用了多个 **END** 指令，则 Ax51 只对第一个 **END** 指令前面的程序行进行汇编，**END** 指令后面的程序行将被忽略。

2. __ERROR__ 指令

__ERROR__ 指令用来产生错误报告信息。指令格式如下：

__ERROR__ 错误信息

其中，“错误信息”将显示在汇编列表文件中。

例子：

```
IF CVAR1LEN > 10
    __ERROR__ "CVAR1 LEN EXCEEDS 10 BYTES"
ENDIF
```

```
$IF TESTVERS AND RELEASE
```

```
    __ERROR__  "TESTVERS GENERATED IN RELEASE MODE"
```

```
$ENDIF
```

10.3 宏处理器

Ax51 宏汇编器提供三种不同的宏处理器：标准宏处理器、C 宏处理器以及 MPL 宏处理器。标准宏处理器与其他宏汇编器相同，采用类似于汇编语言的指令进行宏处理。C 宏处理器提供一种与 Cx51 编译器兼容的宏处理，例如，可以定义能够同时为 Ax51 和 Cx51 使用的公共头文件等。MPL 是“Macro Processing Language”的缩写，即宏处理语言，它提供与 Intel ASM-51 宏汇编器兼容的宏处理，采用 MPL 宏时不能同时使用 C 宏处理器。

宏处理分为宏定义和宏调用。宏定义是将一组经常使用的汇编指令定义成一个宏名，宏调用则是在程序中调用一个已定义过的宏名并将它扩展成一组汇编指令。宏与子程序是两个不同的概念。宏调用时由于要将宏名扩展成一组指令，因此会使整个程序代码加长，但可以加快程序的执行速度。子程序调用不会增加程序代码，但由于需要使用 CALL 指令而会降低程序执行速度。一般而言，对于程序中需要经常调用的一些过程，或者要求系统的存储器空间保持最小时，最好采用子程序。当需要最大程度地发挥处理器的速度，同时系统的存储器空间足够大时，则可采用宏。

10.3.1 标准宏处理器

标准宏处理指令有：MACRO、ENDM、LOCAL、REPT、IRP、IRPC 和 EXITM，分别介绍如下。

1. MACRO/ENDM 指令

在汇编语言程序中使用宏之前应先进行宏定义，宏定义必须以 MACRO 指令开始，以 ENDM 指令结束，位于 MACRO 与 ENDM 之间的 8051 指令行称为宏体。指令格式如下：

宏名 MACRO [形式参数表]

宏名可以自由选择，但必须遵循 Ax51 宏汇编器关于符号的规定，并且不能是 Ax51 的保留字。形式参数表中最多可包含 16 个形式参数，各个形式参数之间用逗号隔开，形式参数将在宏调用时被实际参数所取代，不能采用 Ax51 的保留字作为形式参数，每个形式参数名必须是惟一的。宏定义可以嵌套，宏处理器自动计数每个 MACRO 指令，并将每个 ENDM 指令与正确的 MACRO 指令对应。

例子：

```
CLEAR MACRO G1, G2      ; 宏定义
      MOV    R0, #G1
      MOV    A, #G2
      MOV    @R0, A
      ENDM
CLEAR 20H, 0            ; 宏调用，用 0 定义字节 20H
```

CLEAR 5, 'A' ; 宏调用, 用字符 'A' 的 ASCII 码 41H 定义字节 05H

2. LOCAL 指令

采用 LOCAL 指令允许在宏定义中使用最多 16 个局部标号或任意符号组合, 这些符号只在所定义的宏内是已知的。指令格式如下:

LOCAL 符号名[,...]

LOCAL 指令只能在宏体之内使用, 而且必须紧跟在 MACRO 指令之后, 因为宏体内不允许有空行。在 LOCAL 指令前面不允许有标号。一个宏内可用 LOCAL 指令最多定义 16 个局部符号, 各个符号之间用逗号隔开, 局部符号不能是 Ax51 的保留字或形式参数名, 并且不能彼此重复。

例子:

```
MEMCLR  MACRO  G1,G2,G3      ; 宏定义
          LOCAL  LOOP          ; 宏体内局部符号定义
          MOV    R0,#G1
          MOV    A,#G2
          MOV    R7,#G3
LOOP:    MOV    @R0,A
          INC    R0
          DJNZ   R7,LOOP
          ENDM
MEMCLR  20H,0,10      ; 宏调用, 从 20H 开始用 0 定义 10 个字节
MEMCLR  5, 'A',10     ; 宏调用, 从 05H 开始用 41H 定义 10 个字节
```

如果上例中没有使用 LOCAL 指令, 则在第二次 (以及以后) 调用该宏时将会出现重复定义符号 LOOP 的错误。Ax51 宏汇编器为 LOCAL 指令定义的符号产生一个内部符号, 内部符号以 '??0000' 开始, 每次调用该宏的时候该值加 1。

3. REPT 指令

REPT 指令用于在宏定义中指定需要重复执行的 8051 指令, 位于 REPT 和 ENDM 之间的一组 8051 指令被重复执行。指令格式如下:

[标号:] REPT 表达式

8051 指令组被重复执行的次数取决于表达式的结果值。表达式中可使用符号, 但不能使用前向参考量。表达式的结果值是绝对和无类型的。使用 REPT 之后, 每当遇到 ENDM 指令时, 便立即重复执行所指定 8051 指令组, 同时将表达式的结果值减 1, 直至减为 0 为止。LOCAL 指令在 REPT 块内也是有效的。

例子:

```
DELAY  MACRO              ; 宏定义
          REPT  3          ; 插入 5 个 NOP 指令
          NOP
          ENDM
        ENDM
```


本例所产生的代码如下：

```
NOP
NOP
NOP
NOP
NOP
```

4. IRP 指令

IRP 指令与 REPT 相似，但 IRP 指令的重复因子取决于参数表中的参数。指令格式如下：

[标号:] IRP 形式参数, <参数表>

IRP 指令每次执行时将用参数表中一个实际参数取代指定的形式参数。

例子：

```
CLRREGS      MACRO
                IRP  RNUM, <R0,R1,R2,R3,R4,R5,R6,R7>
                    MOV  RNUM, #00H
                ENDM
            ENDM
```

本例所产生的代码如下：

```
MOV R0, #00
MOV R1, #00
MOV R2, #00
MOV R3, #00
MOV R4, #00
MOV R5, #00
MOV R6, #00
MOV R7, #00
```

5. IRPC 指令

IRPC 指令于 REPT 相似，不同的是 IRPC 指令的重复因子取决于实际参数中的字符个数。指令格式如下：

[标号:] IRPC 形式参数, 实际参数字符串

IRPC 将重复宏体中的一系列 8051 指令，每次重复时用实际参数字符串中的一个字符代替形式参数。如果字符序列用尖括号<>括起，则将全部字符序列作为一个实参。

例子：

```
DEBUGOUT      MACRO                                ; 宏定义
                IRPC CHR, <TEST>
                JNB  TI, $                            ; 等待发送字符
                CLR  TI
                MOV  A, #'T'
```

```

MOV SBUF,A          ; 发送字符
ENDM
ENDM

```

本例所产生的代码如下：

```

JNB TI,$           ; 等待发送字符
CLR TI
MOV A,#'T'
MOV SBUF,A         ; 发送字符'T'

```

6. EXITM 指令

EXITM 指令是一种结束宏扩展或 REPT、IRP、IRPC 循环的指令，Ax51 宏汇编器遇到 EXITM 指令时，将忽略所有的宏体，直到 ENDM 指令。EXITM 指令可以加入到宏体中，但不能取代 ENDM 指令。位于嵌套宏中的 EXITM 指令使宏扩展退到上一级宏，位于 REPT、IRP、IRPC 宏体中的 EXITM 指令结束当前和后继的循环。

例子：

```

MAC2 MACRO X,Y
...
IF X=0
    EXITM
ENDIF
...
ENDM

```

本例中使用了条件汇编语句 IF，关于条件汇编语句将在汇编控制指令一节中详细介绍。

A51 宏汇编器允许使用如下宏操作符：

- & 用来连接文本和形式参数
- <> 使括号内的逗号和空格等定界符有意义，为了将这种特殊的文本传递给嵌套的宏，对每一个嵌套级都需要一对尖括号。
- ； 该符号表明以后直到行结束的正文都不被处理。
- ！ 如果用在另一个字符之前（通常是一个定界符），则使该字符变得有意义，从而可使符号“<”、“>”、“,”、“!”或其他定界符能作为实际文本传递。
- % 用做宏参数的前缀，该参数将作为表达式对待，使用%操作符后，将计算出表达式的数值并传递给所定义的宏，而不再传递表达式文本。
- NUL 有时所选的参数是无值的，需要将它们识别出来。该操作符检查参数是否为 NULL，当参数为 NULL 时将产生一个非 0 值，当参数为 Non-NULL 时将产生一个 0 值。操作符 NUL 通常与条件汇编语句 IF 一起使用。

10.3.2 嵌套宏定义

一个宏定义如果包含在另一个宏定义中，称为嵌套宏定义。A51 的宏处理器允许嵌套

9 级宏定义。在调用嵌套的宏定义时，必须按嵌套级别从高级宏定义开始调用，即在一个较高级别宏中扩展一个低级宏之前，高级宏必须完全扩展。如果企图先调用一个较低级别的宏定义将会发生错误。

例子：

```
MA1  MACRO  X,Y           ; 宏定义,
      MOV  R0,X
      MA2  MACRO  Z       ; 嵌套的低级宏定义
            MOV  R1,Z
            ENDM
      ENDM
MA1  10H,20H             ; 调用高级宏定义
MA2  30H                 ; 调用低级宏定义
```

10.3.3 宏调用

宏一经定义便可在程序中多次调用，调用格式如下：

[标号:] 宏名 [实际参数表]

宏调用时，用实际参数替换宏定义中的形式参数，替换时按实际参数的顺序进行，即第一个实参替换第一个形参，第二个实参替换第二个形参等。如果调用时给出了过多的实参，则多余的实参将被忽略。如果给出的实参个数比形参个数少，则对每个多出的形参用空字符替换。

一个宏可以间接或直接调用其自身，但所有打开的文件和宏的个数不能超过 9。如果嵌套级数大于 9，将导致致命错误而使汇编停止，在递归宏调用时将 EXITM 指令放在适当的位置可避免产生这种错误。

例子：

```
PARAM1  SET    5
RECALL  MACRO  X           ; 宏定义
      MOV  A,X
      IF  PARAM1<>0
            PARAM1  SET  PARAM1-1
            RECALL  10H     ; 递归调用
      ENDIF
      ...
      ENDM
      MOV  A,#0
      RECALL  20H           ; 宏调用
      END
```

本例中宏 RECALL 调用自身 5 次。宏调用中的实际参数如果是一个字符串，该字符串将作为实际文本传递。如果实际参数是以“%”开头的文本，则将从“%”开始至下一个定界符的文本解释为一个表达式，并将它们的十进制数结果作为实际参数传递。

下面是几个关于宏应用的例子。

例 10.2: 嵌套 IRPC 指令的使用。

```

MOVE      MACROE  X,Y,Z
          IRPC    PARAM,Z
              MOV  R0,#X&&PARAM
              MOV  R1,#Y&&PARAM
              MOV  A,@R0
              MOV  @R1,A
          ENDM
        ENDM

          MOVE    SRC,DST,123      ; 宏调用

        END

```

本例中使用了 IRPC 嵌套宏定义，在 IRPC 定义的宏体中使用了两个“&”符号，在调用 MOVE 宏时，IRPC 宏体中的“X&”和“Y&”即被扩展，在为 X&和 Y&插入实参时，后面的一个“&”符号将会消失。本例的宏调用“MOVE SRC, DST, 123”产生如下代码：

```

IRPC      PARAM,123
  MOV      R0,#SRC&PARAM
  MOV      R1,#DST&PARAM
  MOV      A,@R0
  MOV      @R1,A
ENDM

  MOV      R0,#SRC1
  MOV      R1,#DST1
  MOV      A,@R0
  MOV      @R1,A
  MOV      R0,#SRC2
  MOV      R1,#DST2
  MOV      A,@R0
  MOV      @R1,A
  MOV      R0,#SRC3
  MOV      R1,#DST3
  MOV      A,@R0
  MOV      @R1,A

```

例 10.3: 使用嵌套宏产生 DB 指令。

```

GENLAB    MACRO    f1,f2              ; 宏定义
$SAVE     GEN
          F1&F2:    DB  f2,f2,f2,f2
$RESTORE
          ENDM

```

```

BLOCK    MACRO    numb,prefix ; 宏定义
$SAVE    NOGEN
COUNT   SET      0
          REPT     numb
COUNT   SET      COUNT+1
          GENLAB   prefix,%COUNT
          ENDM
$RESTORE
          ENDM
BLOCK    5,LABEL          ; 宏调用
END

```

本例中使用了 A51 的汇编控制命令“\$”，可以改善程序的可读性。关于“\$”的功能将在汇编控制命令一节详细讨论。本例中使用宏调用“BLOCK 5, LABEL”产生如下 DB 指令：

```

PERFIX1: DB  1,1,1,1
PERFIX2: DB  2,2,2,2
PERFIX3: DB  3,3,3,3
PERFIX4: DB  4,4,4,4
PERFIX5: DB  5,5,5,5

```

例 10.4：一个将自己转换成子程序的宏。

```

FIRST    SET  0FFFFH
MODIFY    MACRO          ; 宏定义
          LOCAL  DONE
IF      NOT  FIRST
$SAVE   GEN
          CALL  UPP
$RESTORE
          EXITM
ELSE
FIRST   SET   0000H
          CALL  UPP
          JMP   DONE
UPP:    MOV   A, #77H
          MUL  AB
          RET
ENDIF
DONE:
ENDM
          MODIFY          ; 第一次宏调用
          MODIFY          ; 第二次宏调用
END

```

下面是本例的汇编列表文件。

LOC	OBJ	LINE	SOURCE	
FFFF		1	FIRST SET 0FFFFH	
		2	MODIFY MACRO	; 宏定义
		3	LOCAL DONE	
		4	IF NOT FIRST	
		5	\$save GEN	
		6	CALL UPP	
		7	\$restore	
		8	EXITM	
		9	ELSE	
		10	FIRST SET 0000H	
		11	CALL UPP	
		12	JMP DONE	
		13	UPP: MOV A, #77H	
		14	MUL AB	
		15	RET	
		16	ENDIF	
		17	DONE:	
		18	ENDM	
		19	MODIFY	; 第一次宏调用
		20+1	IF NOT FIRST	
		+1	\$save GEN	
		+1	CALL UPP	
		+1	\$restore	
		+1	EXITM	
		+1	ELSE	
0000		26+1	FIRST SET 0000H	
0000 1104		27+1	CALL UPP	
0002 8004		28+1	JMP ??0000	
0004 7477		29+1	UPP: MOV A, #77H	
0006 A4		30+1	MUL AB	
0007 22		31+1	RET	
		32+1	ENDIF	
0008		33+1	??0000:	
		34	MODIFY	; 第二次宏调用
		35+1	IF NOT FIRST	
		36+1+1	\$save GEN	
0008 1104		37+1+1	CALL UPP	
		38+1+1	\$restore	
		39	EXITM	
		40	END	

列表文件中显示出了所产生的行号，行号后面的“+1”表示汇编语言源程序宏定义中的嵌套级别，该值可为 1~9。如果使用了汇编控制命令“SAVE”和“RESTORE”，在列表文件中将出现“行号+n+n”（如本例中的行号+1+1），它表示“SAVE”和“RESTORE”的嵌套级别。

10.3.4 C 宏处理器

Ax51 宏汇编器具有一个与 Cx51 编译器相一致的 C 宏处理器，允许用户在汇编语言程序和 Cx51 程序中使用公共头文件。Ax51 宏汇编器还可以接受 Cx51 编译器的特殊功能寄存器指令，因此可以同时 Cx51 源程序和 A51 汇编语言程序中使用特殊功能寄存器定义文件。注意，如果允许了汇编控制命令“MPL”，则不能使用 C 宏处理器。Ax51 提供的 C 宏处理指令如表 10-9 所示，其使用方法与 Cx51 编译器相似，所有 C 宏处理指令前面需要加一个“#”号，例如：

```
#include <reg51f.h>
#if TEST
    #define DEBUOG 1
#endif
```

表 10-9 Ax51 宏汇编器提供的 C 宏处理指令

C 宏处理指令	说 明
define	预处理器宏定义指令
elif	条件编译指令
else	条件编译指令
endif	条件编译结束指令
error	输出错误信息指令
ifdef	条件编译指令
ifndef	条件编译指令
if	条件编译指令
include	包含指令
line	行号指令
pragma	编译控制传送指令
undef	删除宏定义指令

Ax51 宏汇编器提供如表 10-10 所示的预定义 C 宏符号（注意首尾都有双下划线）。

表 10-10 Ax51 宏汇编器提供的预定义 C 宏符号

C 宏符号	说 明
<u>__A51__</u>	A51 宏汇编器版本号
<u>__Ax51__</u>	Ax51 宏汇编器版本号
<u>__DATE__</u>	编译开始日期

续表

C 宏符号	说 明
<code>_FILE_</code>	被编译的文件名
<code>_KEIL_</code>	定义为 1, 表示用户使用的是 KEIL 公司的软件
<code>_LINE_</code>	被编译文件的当前行号
<code>_TIME_</code>	编译开始时间
<code>_STDC_</code>	定义为 1, 表示与 ANSI C 标准一致

例 10.5: C 宏处理器的应用。

使用了 C 宏处理指令的汇编语言源程序文件如下:

```
#if !defined ( _ _A51_ _ ) || _ _A51_ _ < 600
    #error "This source file requires A51 V6.00 or higher"
#endif

#pragma NOLIST
#include <reg52.h>
#pragma LIST

#define TEST1  10
#define MYTEXT "Hello World"

#if      TEST1 == 10
        DB  MYTEXT

#endif

        DB  "GENERATED: ", _ _DATE_ _
        MOV R0, #TEST1 * 10

END
```

汇编列表文件文件如下:

LOC	OBJ	LINE	SOURCE
		1	
		4	
		117	\$LIST
		118	
		119	
		120	
		121	
		122	


```

0000 48656C6C      123      DB      "Hello World"
0004 6F20576F
0008 726C64
                                124
                                125
000B 47454E45      126      DB      "GENERATED: ", "May 10 2003"
000F 52415445
0013 443A204D
0017 61792031
001B 30203230
001F 3033
0021 7864          127      MOV R0, #10 * 10
                                128      END

```

采用 C 宏处理器时应注意下两点。

① 位于注释文本最后的“\”将使下面的一行有效指令语句也被注释掉。

② 如果在条件汇编块中使用了“\$INCLUDE”，即使该条件块不参加汇编，包含文件也必须存在。

10.3.5 MPL 宏处理器

MPL 是 Macro Processing Language 的缩写，即宏处理语言的意思，它是一种与 Intel ASM-51 兼容的宏处理器，实际上它是一种字符串替换工具。在对源程序进行汇编之前，先要进行 MPL 处理。MPL 处理器对源程序文件进行扫描以找出宏调用，并将其替换为预定义或用户定义的特殊文本。

MPL 宏处理器函数 DEFINE 用于进行宏定义，格式如下：

%[*] DEFINE (宏名 [参数表列]) (宏体)

其中，“宏名”的长度最大为 31 个字符，首字符应为 26 个大小写英文字母、下划线或问号。

“参数表列”是可选项，宏定义中的形式参数将被宏调用中的实际参数所取代，若有多个参数，各参数之间要用逗号隔开，每个参数必须是惟一的。

“宏体”通常在宏调用中替换的文本，宏体中还可以包含另一个宏调用。

宏调用时只要在符号“%”后面跟一个已定义的宏名即可。

文字化符号“*”是可选项，如果在宏定义时采用了符号“*”，MPL 处理器将以文字化方式进行宏扩展，参数、ESCAPE 函数、COMMENT 函数、BRACKET 函数被展开，但不作进一步的处理，对任何其他宏的调用也不进行扩展。通过下面的例子可以看出是否采用符号“*”进行宏定义的区别：

```
%SET (TOM,1)
```

```
% * DEFINE (AB) (%EVAL (%TOM))
```

```
%DEFINE (CD) (%EVAL (%TOM))
```

定义宏 AB 和宏 CD 时，TOM = 1。对于 AB 的宏体不进行求值，而对于 CD 的宏体则要进

行完全求值。TOM 值的改变不影响宏 CD，但会影响宏 AB：

```
%SET (TOM,2)    → null string
%AB              → 2
%CD              → 1
% * CD           → 1
% * AB           → %EVAL(%TOM)
```

例 10.6: MPL 宏定义与宏调用。

```
*DEFINE (BMOVE (src,dst,cnt)) LOCAL lab (      ; 宏定义
    MOV R0,%%src
    MOV R1,%%dst
    MOV R2,%%cnt
%lab:  MOV A,@R0
        MOV @R1,A
        INC R0
        INC R1
        DJNZ R2,%lab
)
ALEN EQU 10                                ; 定义数组的大小
DSEC SEGMENT IDATA                        ; 定义 IDATA 段
PSEC SEGMENT CODE                         ; 定义 CODE 段

        RSEG DSEC                        ; 激活 IDATA 段
arr1: DS ALEN                             ; 定义数组
arr2: DS ALEN

        RSEG PSEC                        ; 激活 CODE 段
%BMOVE (arr1,arr2,ALEN)                  ; 宏调用，移动存储器模块
END
```

以上例子的列表文件如下：

LOC	OBJ	LINE	SOURCE
		1	
		2	
00000A		3	ALEN EQU 10 ; 定义数组的大小
-----		4	DSEC SEGMENT IDATA ; 定义 IDATA 段
-----		5	PSEC SEGMENT CODE ; 定义 IDATA 段
		6	
-----		7	RSEG DSEC ; 激活 IDATA 段
000000		8	arr1: DS ALEN ; 定义数组
00000A		9	arr2: DS ALEN
		10	
-----		11	RSEG PSEC ; 激活 CODE 段

```

12
13 ; %BMOVE (arr1,arr2,ALEN); 宏调用, 移动存储器模块
14 ;
15 ; MOV R0,#%src
16 ; MOV R1,#%dst
17 ;MOV R2,#%cnt
18 ; %lab: MOV A,@R0
19 ; MOV @R1,A
20 ; INC R0
21 ; INC R1
22 ; DJNZ R2,%lab
23
24 ; MOV R0,#%src
25 ; arr1
000000 7E0000 F 26 MOV R0,#arr1
27 ; MOV R1,#%dst
28 ; arr2
000003 7E1000 F 29 MOV R1,#arr2
30 ; MOV R2,#%cnt
31 ; ALEN
000006 7E200A 32 MOV R2,#ALEN
33 ; %lab: MOV A,@R0
34 ; LAB0
000009 A5E6 35 LAB0: MOV A,@R0
00000B A5F7 36 MOV @R1,A
00000D A508 37 INC R0
00000F A509 38 INC R1
39 ; DJNZ R2,%lab
40 ; LAB0
000011 A5DA00 F 41 DJNZ R2,LAB0
42
43 END

```

从列表文件第 13 行可以看到, 宏调用%BMOVE 中带有 3 个参数, 第 14 到第 20 行为宏扩展。本例中使用了一个局部标号 lab, 它是在宏定义中采用“LOCAL”来定义的局部符号。MPL 宏处理器自动为局部符号加上一个数字后缀, 每次进行宏调用时该数字自动加 1, 从而保证宏调用时标号的惟一性。第一次宏调用时该数字后缀为 0。在宏定义中采用局部符号的一般格式如下:

%[*] DEFINE (宏名 [参数表列]) [LOCAL 局部符号表列] (宏体)

其中, 局部符号表列是一个可选项, 如果有多个局部符号, 则每个符号之间要用逗号或空格隔开。采用局部符号对宏调用的语法没有任何影响。

10.3.6 MPL 函数

MPL 宏处理器具有若干个预定义宏处理函数，MPL 预定义函数与用户自定义宏的主要区别是，用户自定义宏可以重复定义，而 MPL 预定义函数则不能重新定义。上一节中使用的 DEFINE 实际上就是一个 MPL 预定义函数，本节给出其他 MPL 预定义函数。

1. 注释函数

注释函数用于对宏定义进行适当注释，一般格式如下：

% '文本'

% '文本' 行结束符

注释函数中的文本在进行宏调用时总是作为空字符处理。注释函数可以识别两种结束字符：省略号和行结束符。

例子：

% 'this is macro comment.' ; 汇编语言注释行

% 'comment' ; 注释函数，注意没有右边的单引号。

通过下面 MPL 宏处理前后的两条语句比较，可以更清楚地理解注释函数。

MPL 宏处理前：

MOV R5, A %'the following line will be kept separate'

MOV A, %'this comment eats the newline character

R7

MPL 宏处理后：

MOV R5, A

MOV A, R7

2. ESCAPE 函数

ESCAPE 函数中断对宏文本的扫描，用于防止 MPL 宏处理器对不需要的文本进行展开。一般格式如下：

%n 长度为 n 个字符的文本

其中百分号%后面的数字 n 表明文本中不参与计算的字符数（最大为 9）。ESCAPE 函数对于插入百分号、逗号或括号特别有用。

例子：

10%1% OF 10 = 1 ; 扩展为：10% OF 10 = 1

ASM%0251 ; 扩展为：ASM251

3. 括号函数

括号函数与 ESCAPE 函数类似，也可以用于防止 MPL 宏处理器对不需要的文本进行展开。一般格式如下：

%(文本)

括号函数禁止对括号的文本进行 MPL 宏处理，但是对于 ESCAPE 函数、注释函数以

及参数替换仍然可以识别。该函数对括号内文本没有长度限制，因而它比 ESCAPE 函数更容易使用。

例子：

```
ASM(%251)           ; 扩展为:  ASM251
%(1,2,3,4,5)        ; 扩展为:  1, 2, 3, 4, 5
```

定义 ‘DW’ 的宏：

```
% * DEFINE (DW (LIST, LABEL)) (
%LABEL:   DW          %LIST
)
```

对以上 ‘DW’ 宏进行调用：

```
%DW (%(120, 121, 122, 123, -1), TABLE)
```

‘DW’ 宏调用的返回值为：

```
TABLE:   DW   120, 121, 122, 123, -1
```

上面 ‘DW’ 的宏定义中使用了两个形式参数：字表达式 LIST 和标号名 LABEL，在宏调用中分别用实际参数%(120, 121, 122, 23, -1)和 TABLE 进行替换。如果不采用括号函数将无法实现正确的参数传递，因为 MPL 宏处理器会认为逗号是实际参数之间的分隔符。

4. METACHAR 函数

该函数允许用户改变 MPL 宏处理器的哑元字符。一般格式如下：

```
%METACHAR (文本)
```

该函数将括号中文本的第一个字符作为新的哑元字符。字符@、(、)、*、空白符、tab、以及标识符不能用作哑元字符。

例子：

```
%METACHAR (!)       ; 将哑元字符改为 “!”
!(1,2,3,4)           ; 采用新哑元字符的括号函数
```

出现在某些 MPL 预定义函数中的文本可能被认为是数字表达式，如 EVAL 函数中的文本或 “IF”、“WHILE”、“REPEAT”、“SUBSTR” 等控制流程中的参数。表达式按以下方式处理。

- 数字表达式中的文本按一般宏函数的参数计算方法展开。
- 结果字符串被同时计算为数字和字符表示形式。返回值为字符表示形式。

允许采用以下运算符：

- ① 带括号的表达式
- ② HIGH, LOW
- ③ *, /, MOD, SHL, SHR
- ④ EQ, LT, GT, GE, NE
- ⑤ NOT
- ⑥ AND, OR, XOR

算术运算采用 16 位整数进行, 关系运算符的结果为 0 (假) 或 1 (真)。

MPL 宏处理器中可以采用十进制数、十六进制数、八进制数、二进制数, 如果不特别声明其数制, 则认为是十进制数, 十六进制数可以采用与 Cx51 相同形式, 如 0x1234、0xFE02 等。MPL 宏处理器允许在表达式中使用 ASCII 字符串, 表达式中允许包含有位于两个单引号之间的 1 个或 2 个字符, 如: 'A' 将被计算为 0041H, 'AB' 将被计算为 4142H, 而 '' 和 'abc' 则为非法形式。

5. SET 函数

SET 函数允许定义 “macro-time” 符号, 一般格式如下:

%SET (标识符, 表达式)

该函数将数字表达式的值赋给标识符。SET 函数只对 MPL 处理器的符号表产生影响, 由 SET 函数定义的符号可以被另一个 SET 函数或 DEFINE 函数重新定义。

例子:

源文本:

%SET (CNT, 3)

%SET (OFS, 16)

MOV R1, #%%CNT+%%OFS

%SET (OFS, %%OFS + 10)

OFS = %%OFS

输出文本:

MOV R1, #3+16

OFS = 26

SET 符号还可以用在定义其自身值的表达式中:

源文本:

%SET (CNT, 10) %' define variable CNT'

%SET (OFS, 20) %' define variable OFS'

改变 CNT 和 OFS 的值:

%SET (CNT, %%CNT+%%OFS) %' CNT = 30'

%SET (OFS, %%OFS * 2) %' OFS = 40'

MOV R2, #%%CNT + %%OFS %' 70'

MOV R5, #%%CNT %' 30'

输出文本:

MOV R2, #30 + 40

MOV R5, #30

6. EVAL 函数

EVAL 函数以参数方式接受表达式并返回十进制字符。一般格式如下:

%EVAL (表达式)

其中表达式参数必须是已经定义的合法宏标识符。

例子:

源文本:

```
%SET (CNT, 10) %' define variable CNT'
%SET (OFS, 20) %' define variable OFS'
MOV R15, #%EVAL (%CNT+1)
MOV WR14, #%EVAL (14+15*200)
MOV R13, #%EVAL (-(%CNT + %OFS - 1))
MOV R2, #%EVAL (%OFS LE %CNT)
MOV R7, #%EVAL (%OFS GE %CNT)
```

输出文本:

```
MOV R15, #11
MOV WR14, #3014
MOV R13, # -29
MOV R2, #0
MOV R7, #1
```

7. 逻辑表达式与字符串比较

下面的 MPL 处理器函数比较两个字符串参数并返回比较结果的逻辑值, 如果计算结果为真, 则返回 1, 如果计算结果为假, 则返回 0。

%EQS(参数 1, 参数 2)	若两个参数相同则返回 1
%NES(参数 1, 参数 2)	若两个参数不相同则返回 1
%LTS(参数 1, 参数 2)	若第一个参数小于第二个参数则返回 1
%LES(参数 1, 参数 2)	若第一个参数小于或等于第二个参数则返回 1
%GTS(参数 1, 参数 2)	若第一个参数大于第二个参数则返回 1
%GES(参数 1, 参数 2)	若第一个参数大于或等于第二个参数则返回 1

例子:

```
%LTS(A51,a51)      : 返回 1, 因为 a 的 ASCII 码大于 A 的 ASCII 码
%GTS(10,16)         : 返回 0, 因为 10 小于 16
```

10.3.7 MPL 条件处理函数

1. IF 函数

IF 函数计算一个逻辑表达式的值, 并根据该表达式来决定是展开还是跳过其文本参数。一般格式如下:

```
%IF (表达式) THAN (文本 1) [ELSE (文本 2)] FI
```

该函数首先计算表达式的值, 若计算结果为真, 则展开文本 1, 若计算结果为假并包含了可选项 ELSE, 则展开文本 2, 若计算结果为假并且没有包含可选项 ELSE, 则返回一个空字符。该函数必须以 FI 结尾。IF 函数可以嵌套调用, 此时要注意与 ELSE 与 FI 配套的 IF。

例子:

源文本:

```
%*DEFINE (ADDSUB (op,p1,p2)) (
    %IF (%EQS (%op,ADD)) THEN (
        ADD %p1,%p2
    )ELSE (%IF (%EQS (%op,SUB)) THEN (
        SUB %p1,%p2
    )FI
)FI
)

%ADDSUB (ADD,R15,R3)      %' generate ADD R15,R3'
%ADDSUB (SUB,R15,R9)      %' generate SUB R15,R9'
%ADDSUB (MUL,R15,R4)
```

输出文本:

```
ADD R15,R3
SUB R15,R9
```

2. WHILE 函数

用 WHILE 函数可以实现宏循环,一般格式如下:

%WHILE (表达式) (文本)

WHILE 函数先计算表达式的逻辑值,结果为真则展开后面的文本,然后再次计算逻辑表达式的值,如果仍然为真,则再次展开文本,如此循环一直到逻辑表达式的值为假为止。如果第一次计算逻辑表达式的值为假则不展开后面的文本。为了能够退出循环,文本中应包含对表达式逻辑值的修改部分。

例子:

源文本:

```
%SET (count, 5)          %' 设置 count 初值为 5 '
%WHILE (%count GT 0)
(      ADD R15,R15 %SET (count, %count - 1)
)
```

输出文本:

```
ADD R15,R15
ADD R15,R15
ADD R15,R15
ADD R15,R15
ADD R15,R15
```


3. REPEAT 函数

REPEAT 函数将文本展开指定的次数，一般格式如下：

%REPEAT (表达式) (文本)

例子：

源文本：

%REPEAT (5)

(-enter any key to shut down-

)

%REPEAT (5) (+%REPEAT (9) (-))+

输出文本：

-enter any key to shut down-

-enter any key to shut down-

-enter any key to shut down-

-enter any key to shut down-

-enter any key to shut down-

4. EXIT 函数

EXIT 函数终止最近一次对 REPEAT、WHILE 或用户定义宏函数的调用。一般格式如下：

%EXIT

该函数通常用于终止死循环，在同一个宏中允许出现多个退出点。

例子：

源文本：

%SET (count, 0)

%WHILE (1)

(%IF (%count GT 5) THEN (%EXIT)

FI DW %count, -%count

%SET (count, %count + 1))

输出文本：

DW 0, -0

DW 1, -1

DW 2, -2

DW 3, -3

DW 4, -4

DW 5, -5

10.3.8 MPL 字符串处理函数

1. LEN 函数

该函数以十六进制值返回字符串的长度，字符串最多包含 256 个字符，一般格式如下：

%LEN (文本)

例子：

源文本：

```
%LEN (A251) %' len = 4'
```

```
%LEN (A251,A251) %' comma counts also'
```

```
%LEN ()
```

```
%LEN (ABCDEFGHIJKLMNOPQRSTUVWXYZ)
```

```
%DEFINE (TEXT) (QUEEN)
```

```
%DEFINE (LENGTH) (%LEN (%TEXT))
```

```
LENGTH OF '%TEXT' = %LENGTH.
```

输出文本

4

9

0

26

LENGTH OF 'QUEEN' = 5.

2. SUBSTR 函数

该函数返回给定文本参数中的子字符串，一般格式如下：

%SUBSTR (文本, 表达式 1, 表达式 2)

其中，文本中可以包含宏调用，表达式 1 指定子字符串的首字符，表达式 2 指定子字符串中包含字符的个数。如果表达式 1 的值为 0 或大于给定文本的长度，则函数返回一个空字符串。如果表达式 2 的值为 0，则函数返回一个空字符串。如果表达式 2 的值大于剩余字符串的长度，则返回字符串中从头到尾的所有字符。

例子：

源文本：

```
%DEFINE (STRING) (abcdefgh)
```

```
%SUBSTR (%string, 1, 2)
```

```
%SUBSTR (%(1,2,3,4,5), 3, 20)
```

输出文本：

ab

2,3,4,5

3. MATCH 函数

MATCH 函数搜索文本中指定的分隔符，并将其两边的子字符串分别赋给指定的标识符。一般格式如下：

%MATCH (标识符 1 分隔符 标识符 2) (文本)

其中，标识符 1 和标识符 2 必须是有效的宏标识符，分隔符是跟在标识符 1 后面的第一个字符，通常为空格或逗号，也可以是除了宏标识符之外的任何字符。被搜索的文本中还可以包含宏调用。如果 MATCH 函数从文本中搜索到了指定的分隔符，则将其左边的所有字符赋给标识符 1，同时将其右边的所有字符赋给标识符 2，如果 MATCH 函数没有从文本中搜索到指定的分隔符，则将全部文本字符串赋给标识符 1，将空字符串赋给标识符 2。

例子：

源文本：

```
%DEFINE (text) (-1,-2,-3,-4,-5)
```

```
%MATCH (next,list) (%text)
```

```
%WHILE (%LEN (%next) NE 0)
```

```
(      MOV R8,#%next
```

```
        MOV @WR2,R8 %MATCH (next,list)(%list)
```

```
        INC WR2,#1
```

```
)
```

输出文本：

```
MOV R8,#-1
```

```
MOV @WR2,R8
```

```
INC WR2,#1
```

```
MOV R8,#-2
```

```
MOV @WR2,R8
```

```
INC WR2,#1
```

```
MOV R8,#-3
```

```
MOV @WR2,R8
```

```
INC WR2,#1
```

```
MOV R8,#-4
```

```
MOV @WR2,R8
```

```
INC WR2,#1
```

```
MOV R8,#-5
```

```
MOV @WR2,R8
```

```
INC WR2,#1
```

4. 控制台 I/O 函数

MPL 宏处理器提供两个控制台 I/O 函数：IN 和 OUT。IN 函数输出提示符“>”，返回

值为从控制台输入的字符行。OUT 函数将一个字符串输出到控制台。对于 OUT 函数的调用将被空字符串所取代。IN 和 OUT 函数的一般格式如下：

```
%IN
%OUT(文本)
例子：
源文本：
%OUT (enter baud rate)
%set (BAUD_RATE,%in)
BAUD_RATE = %BAUD_RATE
```

```
输出文本：
<假设从控制台输入 19200>
BAUD_RATE = 19200
```

10.4 汇编控制命令

Ax51 宏汇编器能够识别两类汇编控制命令：首要控制命令和一般控制命令。首要控制指令只允许在程序中出现一次，并且它们不能被其他控制命令所改变。首要控制命令可以在命令行上调用 Ax51 汇编器时引用，也可以在汇编语言源程序的开始处引用。一般控制命令可以在程序中多次出现，并且可以放在源程序中任何地方。

在 Windows 集成开发环境 μ Vision2 中，通过“Project 菜单/Options For Target 选项/A51(或 Ax51)标签页”很容易对这些命令进行设置，设置完成后对项目进行整体创建(Build target)即可自动调用 Ax51 宏汇编器的控制命令完成对相关文件的汇编。也可以通过命令行来引用 Ax51 宏汇编器的控制命令，格式如下：

Ax51 汇编语言源程序文件 [汇编控制命令，……]

例如，在命令行引用汇编控制命令对源程序 SAMPLE.A51 汇编程序的格式为：

Ax51 SAMPLE.Ax51 XREF DEBUG NOMOD51 ERRORPRINT

如果需要使用多个汇编控制命令，最好将它们放在源程序的开始处。在源程序中引用汇编控制命令时，引用行必须以符号“\$”开头。上面的汇编控制命令可按如下方式放在源程序的开头：

```
$ XREF
$ DEBUG
$ NOMOD51
$ ERRORPRINT
```

也可以将它们放在一行：

```
$ XREF DEBUG NOMOD51 DATE(31,12,96) ERRORPRINT
```

然后采用如下格式在命令行上调用 Ax51 宏汇编器：

```
Ax51 SAMPLE.Ax51
```

如果同时在命令行和源程序中引用了首要汇编控制命令，则以命令行引用的命令为准。如果命令引用不当，将会导致汇编出错。

表 10-11 列出了 Ax51 宏汇编器的全部控制命令。

表 10-11 Ax51 宏汇编器控制命令

首要/一般	控制命令	缩写	默认值	命令功能
首要	CASE	—	—	仅用于 Ax51，允许对符号名大小写敏感
首要	[NO]COND	[NO]CO	—	[不]列出被条件汇编忽略的程序行
首要	DATE(date)	DA	—	在列表文件开头放置日期字符串
首要	DEBUG	DB	—	将符号调试信息加入目标文件
一般	EJECT	EJ	—	列表文件换页
首要	ERRORPRINT(file)	EP	—	将错误信息输出到一个指定的文件中
一般	[NO]GEN	[NO]GE	—	[不]列出每一个宏扩展
首要	INC DIE(path)	ID	—	定义包含文件的搜索路径
首要	INCLUDE(file)	IC	—	指定包含文件
首要	[NO]LIST	[NO]LI	—	[不]在列表文件中列出汇编源文件
一般	MOD51	M51	—	仅用于 Ax51，选择传统 8051 指令集
一般	MOD_MX51	MX	—	仅用于 Ax51，选择 80C51Mx 指令集
一般	MOD_CONT	MC	—	仅用于 Ax51，选择 80C390 指令集
一般	MPL	—	—	允许 Intel ASM51 兼容宏处理器语言
首要	NOLINES	NOLI	—	不产生行号信息
一般	NOMACRO	NOMR	—	禁止标准宏处理器
首要	NOMOD51	—	—	不使用 8051 预定义符号
首要	[NO]OBJECT	[NO]OJ	—	[不]产生目标文件
首要	[NO]REGISTERBANK	[NO]RB	—	[不]使用代码分组
首要	[NO]SYMBOLS	[NO]SB	—	[不]产生符号列表
首要	[NO]SYMLIST	[NO]SL	—	[不]符号表中不列出符号定义
一般	PAGELENGTH(n)	PL	—	设置列表文件每页的行数
一般	PAGEWIDTH(n)	PW	—	设置列表文件每行的字符数
一般	[NO]PRINT	[NO]PR	—	[不]产生列表文件
首要	REGUSE	RU	—	为 C 优化器定义汇编函数中要修改的寄存器
首要	RESTORE	RS	—	恢复最近保存的设置
首要	SAVE	SA	—	保存 GEN, LIST, SYMLIST 的设置
一般	TITLE(string)	TT	—	在每一分页开头放置一个指定的字符串
一般	XREF	XR	—	产生交叉参考表

注：表中“缩写”和“默认值”栏中的“—”表示无缩写和无默认值。

10.4.1 汇编控制命令详解

命令名： CASE（仅用于 Ax51）

- 缩 写: CA
- 参 数: 无
- 属 性: 首要控制
- 默认值: 对字符大小写不敏感, 所有字符都转换为大写。
- µVision2: 在“Options 选项/Ax51 标签页”选中“Case sensitive symbols”复选框。
- 功 能: CSAE 命令使 Ax51 宏汇编器对大小写字符敏感, 不使用该命令时所有字符都被转换为大写。如果模块文件与由 Cx51 编译器生成的汇编模块文件复合在一起时, 该命令无效。从 C51 文件导入的标识符应该与它在 C51 文件中的大小写形式一样。
- 例 子: \$CASE
Ax51 SAMPLE.A51 CASE
-
- 命令名: COND/NOCOND
- 缩 写: 无
- 参 数: 无
- 属 性: 首要控制
- 默认值: COND
- µVision2: 在“Options 选项/Listing 标签页”选中“Assembler Listing”和“Conditional”复选框。
- 功 能: 该命令表明是否将条件汇编块“IF-ELSEIF-ENDIF”结构中未汇编的部分在列表文件中列出来。
- 例 子: \$COND
\$NOCOND
Ax51 SAMPLE.A51 COND
Ax51 SAMPLE.A51 NOCOND
-
- 命令名: DATE
- 缩 写: DA
- 参 数: 括号内一个字符串 (最多 8 个字符)
- 属 性: 首要控制
- 默认值: 由操作系统提供的系统日期。
- µVision2: 在“Options 选项/Ax51 标签页/Misc Controls 栏”中键入命令。
- 功 能: 不带参数的 DATA 命令使 Ax51 宏汇编器将系统提供的当前日期包含到列文件每一页的开头, 如果带有字符参数, Ax51 宏汇编器将取出参数字符串并将它放在列表文件每页的开头, 如果字符串超过 8 个字符, 则只有前 8 个字符有效。
- 例 子: \$DATE(10/05/03)
Ax51 SAMPLE.A51 DATE(10/05/03)

命令名: DEBUG
缩 写: DB
参 数: 无
属 性: 首要控制
默认值: 不产生调试信息
 μ Vision2: 在“Options 选项/Output 标签页/”选中“Creat Ececutable”和“Debug Information”复选框。
功 能: 该控制命令表明是否将供仿真器使用的调试信息和行号信息加入到目标文件中。
例 子: \$ DEBUG
Ax51 SAMPLE.A51 DB

命令名: EJECT
缩 写: EJ
参 数: 无
属 性: 一般控制
默认值: 当达到 PAGELENGTH 命令所规定的值时换页。
 μ Vision2: 该命令不能用于 μ Vision2。
功 能: 该命令将换页命令插入到列表文件中, 并在下页的开始处产生一个题头。NOPRINT 和 NOLIST 命令将抑制 EJECT。
例 子: \$ EJECT

命令名: ERRORPRINT
缩 写: EP
参 数: 括号内一个可选的文件名
属 性: 首要控制
默认值: 不输出错误信息。
 μ Vision2: 本命令不能在 μ Vision2 中指定。
功 能: 该控制命令将源程序中所有的错误行及其相关的错误信息输出到一个指定的文件中。如果没有指定文件, 则将输出到控制台。
例 子: Ax51 SAMPLE.A51 ERRORPRINT(SAMPLE.ERR)
\$ EP

命令名: GEN/NOGEN
缩 写: 无
参 数: 无
属 性: 一般控制
默认值: NOGEN
 μ Vision2: 在“Options 选项/Lixting 标签页”选中“Assembler Listing”复选框, 并

- 在“**Macros**”栏内选中“**All expasinos**”。
- 功 能:** GEN 命令表明在列表文件中每个宏扩展都将被列出。NOGEN 命令表明只列出源文件的内容，而不列出宏扩展。
- 例 子:** \$GEN
Ax51 SAMPLE.A51 NOGEN
- 命令名:** INCDIR
缩 写: ID
参 数: 括号内可选的路径名
属 性: 一般控制
默认值: 无
μVision2: 在“Options 选项/Ax51 标签页/Include Path 栏”中键入指定的路径名。
功 能: INCDIR 命令用于指定由 INCLUDE 命令所包含文件的路径，可以指定多个路径，每个路径用分号“;”隔开。Ax51 宏汇编器首先搜索当前项目所在的路径，然后搜索由 INDIC 命令指定的路径。
- 例 子:** Ax51 SAMPLE.A51 INCDIR(C:Ax51\MYINC; D:\CHIP_DIR)
- 命令名:** INCLUDE
缩 写: IC
参 数: 括号内可选的文件名
属 性: 一般控制
默认值: 无
μVision2: 该命令不能用于μVision2。
功 能: INCLUDE 命令将括号内指定的文件包含到汇编语言源程序文件中，被包含文件的最多可以嵌套 9 级。通常使用该命令来包含 8051 特殊功能寄存器文件，也可以用于包含外部子程序、变量、宏以及汇编语言代码。
- 例 子:** \$INCLUDE (REG51P.INC)
- 命令名:** LIST/NOLIST
缩 写: LI/NOLI
参 数: 无
属 性: 一般控制
默认值: LIST
μVision2: 在“Options 选项/Ax51 标签页/MISC controls 栏”中键入命令。
功 能: 该命令表明后续的源文件是否在列表文件中显示出来。即使使用了 NOLIST 命令，出错行仍将显示出来。NOPRINT 命令将抑制 LIST 和 NOLIST。
- 例 子:** \$ NOLIST
Ax51 SAMPLE.A51 LI

命令名: MOD51, MOD_CONT, MOD_MX51 (仅用于 Ax51)
缩 写: M51, MC, MX
参 数: 无
属 性: 首要控制
默认值: MOD51: 产生传统 8051 代码。
 μ Vision2: 通过器件库选定 CPU 之后自动选用合适的命令。
功 能: MODxxx 命令选择应用代码将采用的指令集。默认值 MOD51 将选择传统 8051 指令集, MOD_MX51 将选择 Philips80C51Mx 的扩展指令集, 这时至少要有 一个程序模块按该模式汇编; MOD_CONT 选择 DS80C390 的 24 位邻接地址模式, 这时所有程序模块都将按该模式汇编, 并且不能使用传统 8051 指令代码。
例 子: Ax51 SAMPLE.A51 MOD_MX51
Ax51 SAMPLE.A51 MOD_CONT
\$ MX
\$ MC

命令名: MPL
缩 写: 无
参 数: 无
属 性: 首要控制
默认值: 禁止宏处理语言。
 μ Vision2: 在“Options 选项/Ax51 标签页/Macro processor 栏”选中“MPL”复选框。
功 能: MPL 命令允许与 Intel ASM51 兼容的宏处理语言。
例 子: Ax51 SAMPLE.A51 MPL
\$ MPL

命令名: NOLINES
缩 写: NILN
参 数: 无
属 性: 首要控制
默认值: 当采用了 DEBUG 命令时将产生源程序的行号信息。
 μ Vision2: 在“Options 选项/Ax51 标签页/Misc controls 栏”中键入命令。
功 能: NOLINES 命令禁止产生源程序的行号调试信息, 它主要用于与早期的 A51 宏汇编器兼容。
例 子: Ax51 SAMPLE.A51 NOLINES
\$ NOLINES

命令名: NOMACRO

- 缩 写: 无
参 数: 无
属 性: 首要控制
默认值: 标准宏被完全扩展。
µVision2: 在“Options 选项/A51 标签页/Special Function Registers 栏”选中“Define 8051 SFR Names”复选框。
- 功 能: NOMACRO 禁止 Ax51 宏汇编器对标准宏的展开, 即关闭宏处理器。
例 子: \$ MACRO
Ax51 SAMPLE.A51 NOMACRO
- 命令名: NOMOD51
缩 写: NOMO
参 数: 无
属 性: 首要控制
默认值: 使用预定义的 8051 特殊功能寄存器。
µVision2: 在“Options 选项/Ax51 标签页/Macro processor 栏”选中“Standard”复选框。
- 功 能: NOMOD51 命令使 8051 所有的符号对于汇编器都是未知的, 从而允许用户自己定义其他型号单片机内部符号。使用用户自定义符号文件时必须采用 NOMOD51 命令, 否则会发生多重定义错误。
例 子: \$ NOMOD51
Ax51 SAMPLE.A51 NOMO
- 命令名: SYMBOLS/NOSYMBOLS
缩 写: SB/NOSB
参 数: 无
属 性: 首要控制
默认值: SYMBOLS
µVision2: 在“Options 选项/Listing 标签页/”选中“Assembler Listing”和“Symbols”复选框。
- 功 能: 该命令表示是否在列表文件中产生符号表, NOSYMBOLS 命令将抑制 Ax51 宏汇编器在列表文件中产生符号表。
例 子: \$ SYMBOLS
\$ NOSB
Ax51 SAMPLE.A51 NOSB
- 命令名: OBJECT/NOOBJECT
缩 写: OJ/NOOJ
参 数: OBJECT (文件名), NOOBJECT 无参数

属 性: 首要控制
默认值: OBJECT (文件名.OBJ)
 μ Vision2: 在“Options 选项/Output 标签页/”通过“Select Folder for Object”按钮选择存放目标文件的目录。
功 能: 该命令表明是否产生目标代码文件。如果没有指定文件名, 则以源文件的基本名加上“.OBJ”作为输出的目标代码文件名, 目标代码文件的存放目录可以通过在“Options 选项/Output 标签页/”中的“Select Folder for Objec”按钮来选择。
例 子: \$ NOOBJECT
\$ OJ(C:\SAMP\SAMPLE.OBJ)
Ax51 SAMPLE.A51 OBJECT(OBJDIR\SAMPLE.OBJ)

命令名: PAGELENTH
缩 写: PL
参 数: 括号内一个 10~65535 的数
属 性: 首要控制
默认值: PAGELENTH(132)
 μ Vision2: 在“Options 选项/Listing 标签页/Page Lenth 栏”中选择页长值。
功 能: 该命令指定列表文件中每页最大的行数, 指定的行数必须大于等于 10。
例 子: \$ PAGELENTH(132)
\$ PL(78)
Ax51 SAMPLE.A51 PL(132)

命令名: PAGEWIDTH
缩 写: PW
参 数: 括号内一个 78~132 的数
属 性: 首要控制
默认值: PAGEWIDTH(120)
 μ Vision2: 在“Options 选项/Listing 标签页/Page Width 栏”中选择页宽值。
功 能: 该命令指定列表文件中每行最大字符数。
例 子: \$ PW(78)
Ax51 SAMPLE.A51 PW(78)

命令名: PRINT/NOPRINT
缩 写: PR/NOPR
参 数: PRINT (文件名), NOPRINT 无参数
属 性: 首要控制
默认值: PRINT (源文件名.LST)
 μ Vision2: 在“Options 选项/Listing 标签页/”通过“Select Folder for Listing”按钮

- 选择存放列表文件的目录。
- 功 能: 该命令表明是否产生列表文件。如果没有指定列表文件名, 则以源程序文件名加上扩展名“.LST”作为列表文件名。存放列表文件的目录可以在“Options 选项/Listing 标签页/”通过“Select Folder for Listing”按钮来选择。
- 例 子: `$ PRINT(TEMPLST)`
`$ NOPR`
`Ax51 SAMPLE.A51 PR(TESTPRG1.LST)`
- 命令名: REGISTERBANK/NOREGISTERBANK
 缩 写: RB/NORB
 参 数: RB(n[,...]) n=0~3
 属 性: 一般控制
 默认值: RB(0)
 μVision2: 在“Options 选项/Ax51 标签页/Misc controls 栏”中键入命令。
 功 能: RB 命令为源程序中的一个模块指定一组工作寄存器, 该信息保存在所生成的目标文件中以便于 Lx51 连接定位器使用。NORB 命令表示不为寄存器组保留空间, 从而可以生成一般库文件。
- 例 子: `Ax51 SAMPLE.A51 REGISTERBANK(0,1,2)`
`$ RB(0,3)`
- 命令名: REGUSE
 缩 写: RU
 参 数: 括号内一个 PUBLIC 符号及工作寄存器列表。
 属 性: 一般控制
 默认值: RB(0)
 μVision2: 该命令不能在μVision2 中指定。
 功 能: REGUSE 命令指定在一个函数执行期间所修改的工作寄存器, 该命令可以与 Cx51 编译器一起使用, 为在汇编语言程序中的函数进行局部寄存器优化。
- 例 子: `$ REGUSE MYFUNC (ACC, B, R0-R7)`
`$ REGUSE PROCA(DPL,DPH)`
- 命令名: SAVE/RESTORE
 缩 写: SA/RS
 参 数: 无
 属 性: 一般控制
 默认值: 无
 μVision2: 该命令不能在μVision2 中指定。

功 能: SAVE 命令在内部汇编栈中保存当前 LIST 和 GEN 的设置。该栈可保留至多 9 个 SAVE 级。RESTORE 命令恢复最近保留的 GEN 和 LIST 的设置。

例 子: \$ SAVE NOLIST
\$ INCLUDE(FILE.Ax51)
\$ RESTORE

命令名: SYMLIST/NOSYMLIST

缩 写: SL/NOSL

参 数: 无

属 性: 一般控制

默认值: SYMLIST

μ Vision2: 在“Option 选项/Ax51 标签页/Misc controls 栏”中键入命令。

功 能: SYMLIST 命令符号表中列出符号定义。NOSYMLIST 命令禁止 Ax51 宏汇编器在符号表中列出符号定义。

例 子: \$ SYMLIST
\$ NOSYMLIST

命令名: TITLE

缩 写: TT

参 数: 括号内一个字符串（最多 60 个字符）

属 性: 一般控制

默认值: 不带扩展名的源程序文件名

μ Vision2: 在“Option 选项/Ax51 标签页/Misc controls 栏”中键入命令。

功 能: 该命令允许在列表文件每页的开头包含程序标题。省略该命令时将使用由“NAME”命令指定的标题，如果不指定文件名，则使用不带扩展名的源程序文件名。

例 子: \$ TITLE(Oven Controller Version 3.12)

命令名: XREF

缩 写: XR

参 数: 无

属 性: 首要控制

默认值: 不生成交叉参考列表。

μ Vision2: 在“Option 选项/Listing 标签页”选中“Assembler Listing”和“Cross Reference”复选框。

功 能: 该命令表明是否在列表文件中产生一个交叉参考表。在交叉参考表中将列出参照符号行的行号。每行紧跟在符号“#”后面的数字指出使用了该符号的程序行。NOPRINT 命令将抑制 XREF。

例 子: \$ XREF
 Ax51 SAMPLE.A51 XREF

10.4.2 条件汇编命令

本节给出 Ax51 的条件汇编命令。条件汇编命令属于一般控制命令,其中 SET 和 RESET 命令既可在源程序中使用也可在命令行上使用,其他命令则只能用于源程序中。条件汇编命令的作用是使源程序中的一部分指令行根据需要决定是否进行汇编。条件汇编命令可以作为 \$ 控制行的汇编控制,也可以作为不带 \$ 符的正常汇编控制。它们的差别在于前者只能访问由 SET 和 RESET 命令定义的符号,而后者可访问源程序中除 SET 和 RESET 符号之外的所有符号,因此可用于宏处理中。IF 块可嵌套使用,最大嵌套深度为 10 级。如果 IF、ELSEIF 或 ELSE 块不汇编,则其他嵌套的条件块也不汇编。

命令名: SET
缩 写: 无
参 数: SET(变量[,变量[,...]])
 SET(变量=数值[,...])
 SET(变量,变量=数值,数值[,...])
默认值: 无
µVision2: 在“Option 选项/Ax51 标签页/Set 栏”中键入命令及其参数。
功 能: SET 和 RESET 命令将数值赋给一个指定的变量。如果变量没有明确赋值,则隐含为 0FFFFH。这些变量随后可在包含 IF 和 ELSEIF 命令的表达式里用做条件汇编。变量必须遵循一般规则并且只能为汇编控制所用。这些变量被单独管理从而不会与其他 CODE、BIT、DATA 和 XDATA 符号相互干扰。数值是一个单独的数字或表达式并且可以使用其他表达式中的运算符。表达式总是无类型的。

例 子: \$ SET(TMP,TMP1=55,TMP2,TMP3)
 该例中变量 TMP、TMP2、TMP3 都被赋以隐含值 0FFFFH(“TRUE”),TMP1 被赋值为 55。

命令名: RESET
缩 写: 无
参 数: RESET(变量[,变量[,...]])
默认值: 无
µVision2: 在“Option 选项/Ax51 标签页/Reset 栏”中键入命令及其参数。
功 能: RESET 命令将数值 0000H 赋给一个指定的变量。这些变量随后可在包含 IF 和 ELSEIF 命令的表达式里用做条件汇编。变量必须遵循一般规则并且只能为汇编控制所用。这些变量被单独管理从而不会与其他 CODE、BIT、DATA 和 XDATA 符号相互干扰。数值是一个单独的数字或表达式

并且可以使用其他表达式中的运算符。表达式总是无类型的。

例 子:

```
$ RESET(TMP,VER,TMP3)
```

该例中所有变量都被赋值为 0000H(“FALSE”)。

命令名: IF

缩 写: 无

参 数: 数值表达式

默认值: 无

μVision2: 该命令不能用于μVision2。

功 能: 如果数值表达式的计算结果值为真(1), 则执行 IF 块的语句。一个 IF 必须与一个 ENDIF 匹配, 否则将产生汇编错误。

例 子: \$ IF(DEBUG_VAR=3)

```
Version_3: MOV DPTR, #TABLE_3
```

```
<other text lines>
```

```
$ ENDIF
```

```
COUNT SET COUNT+1
```

```
IF COUNT=5
```

```
EXITM
```

```
ENDIF
```

命令名: ELSEIF

缩 写: 无

参 数: 数值表达式

默认值: 无

μVision2: 该命令不能用于μVision2。

功 能: 计算 IF 命令中另一个分支的数值表达式, 如果其结果为真, 则汇编该分支块。

例 子: \$ IF(SWITCH=1)

```
<other text lines>
```

```
$ ELSEIF(SWITCH=2)
```

```
<other text lines>
```

```
$ ELSEIF(SWITCH=3)
```

```
<other text lines>
```

```
$ ENDIF
```

命令名: ELSE 命令

缩 写: 无

参 数: 无

默认值:	无
μVision2:	该命令不能用于μVision2。
功 能:	汇编不满足 IF 或 ELSEIF 条件的分支块。一个 IF 语句块可包含多个 ELSEIF 块, 但只能包含一个 ELSE 块。
例 子:	<pre>\$ IF(DEBUG) <other text lines> \$ ELSEIF(TEST) <other text lines> \$ ELSE <other text lines> \$ ENDIF</pre>
命令名:	ENDIF
缩 写:	无
参 数:	无
默认值:	无
μVision2:	该命令不能用于μVision2。
功 能:	ENDIF 命令结束 IF 语句块, 如果 ENDIF 与 IF 命令的个数不等将产生汇编错误。
例 子:	<pre>\$ IF(TEST) <other text lines> \$ ENDIF</pre>

第 11 章 BL51/Lx51 连接定位器与实用工具

在本书前面各章介绍了采用 C 语言编写 8051 单片机应用程序的方法，用户编写的 C 语言源程序经过 Cx51 编译器编译后，生成浮动地址的目标代码文件。这种浮动地址的目标代码是不能直接装入 8051 单片机系统中运行的，必须经过连接定位器的连接和定位，生成具有绝对地址的目标代码，才能在 8051 单片机中运行。Keil 公司提供的 PK51 软件包中包含两种连接定位器：BL51 和 Lx51，BL51 适用于普通传统 8051 单片机运行代码的连接定位，Lx51 适用于扩展型新一代 80C51 单片机运行代码的连接定位。BL51 或 Lx51 能够进行多程序模块连接，它具有静态覆盖功能，可使 8051 单片机中有限的存储器资源得到充分利用，在连接定位的过程中能自动加入运行库中必要的库函数。BL51 或 Lx51 支持分组连接定位，允许生成代码长度大于 64K 字节的 8051 目标程序，在具有适当硬件扩展逻辑的 8051 系统中进行代码组之间的切换，达到正常运行的目的。BL51 可以管理最多 32 个代码组，每个代码组最大为 64K 字节，Lx51 可以管理多达 16MB 的代码和外部数据存储单元空间。BL51 生成 Intel OMF51 格式的绝对目标文件，Lx51 生成 Keil OMFx51 格式的绝对目标文件，它们可以直接在 μ Vision2 环境下进行软件模拟仿真调试，也可以装入硬件仿真器进行实时在线调试，调试通过后的代码可以通过符号转换程序 OH51 转换成 Intel HEX 文件以便于 EPROM 编程。本章将详细介绍连接定位器 BL51/Lx51、符号转换程序 OH51 和 OC51、库管理程序 LIB51 等的功能与使用方法。

11.1 BL51/Lx51 的连接定位方式

BL51/Lx51 是一种功能强大的多模块程序连接定位器。它能将多个由 Cx51 编译器或 Ax51 宏汇编器生成的浮动目标代码模块连接成为一个单一而完整的绝对目标代码模块，并自动从 Cx51 运行库中挑选出必需的库函数文件嵌入到绝对目标模块中。

11.1.1 连接定位中的数据段处理

BL51/Lx51 能将各个模块中具有相同名字的再定位部分段组合到一个单一的再定位段中，同时产生一个段分配表 (MAP)。段分配表中包含了每个段的类型、基地址、长度和段名。段命名规则与 Cx51 编译器相同，各段都有一个前缀作为区分标志，如表 11-1 所示。

表 11-1 段名前缀与数据类型和存储器空间的关系

段名前缀	数据类型	存储器空间
?PR?	CODE	程序执行代码空间
?CO?	CODE	程序代码空间中的常数
?ED?	EDATA	用于 near 变量的 EDATA 空间
?FD?	HDATA	用于 far 变量的 HDATA 空间
?XD?	XDATA	外部数据空间
?DT?	DATA	内部数据空间
?ID?	IDATA	内部间接寻址空间
?BI?	BIT	内部数据存储器的位数据空间
?BA?	BDATA	内部数据存储器的可位寻址空间
?PD?	PDATA	外部数据存储器的分页寻址空间

在进行段组合时要求所有具有相同名字的部分段都有相同的存储器类型（CODE、DATA、IDATA、EDATA、HDATA、XDATA 或 BIT），如果类型不同则会发生错误。组合段的长度不能超过存储器的实际长度。每个组合的部分段采用相同的定位方式（PAGE、INPAGE、BLOCK、BITADDRESSABLE 或 UNIT）。绝对段不能相互组合，而是直接复制到输出文件中。段经过组合之后，它们定位在 8051 系列单片机的存储器中，每个存储器区域被单独处理。

组合段在存储器区域内按如下顺序进行定位。

（1）内部 RAM 区

- 绝对 BIT、DATA、IDATA 段和寄存器组。
- 命令行上包含在“PRECEDE”参数中的段。
- 命令行上包含在“BIT”参数中的段。
- 用 BITADDRESSABLE（位寻址）方式定位的 DATA 段。
- 其他包含在“DATA”参数中的段。
- 命令行上包含在“IDATA”、“EDATA”、“HCONST”参数中的段。
- 除 ?STACK 段之外的其他再定位 IDATA 段。
- 命令行上包含在“STACK”参数中的段。
- 当 ?STACK 段还未被处理时，它具有 IDATA 类型，在任何情况下，不管其长度多少，它总是最后定位于内部数据 RAM 区中，因此栈空间的可用长度为内部 RAM 的最大剩余空间。

（2）外部 RAM 区

- 绝对 XDATA 段。
- 命令行上包含在“XDATA”参数中的段。
- 其他再定位 XDATA 段。

（3）程序代码 ROM 区

- 绝对 CODE 段。

- 命令行上包含在“CODE”参数中的段以及具有再定位类型“INBLOCK”的段。
- 其他 CODE 段和 CONST 段。
- 具有存储器类型“ECODE”、“HCONST”和“HDATA”的段。

BL51/Lx51 采用数据覆盖技术来操作片内数据存储器,使 8051 单片机中有限的内部 RAM 得到充分的利用。对于由 Cx51 编译器或 Ax51 汇编器生成的目标模块中没有相互调用的参数和局部变量,将在存储器中覆盖,从而可产生非常紧凑的数据区。为实现正确的数据覆盖, BL51/Lx51 对所有不同函数之间的调用进行分析,以决定哪个数据段(或位段)可被覆盖。可以使用连接控制命令“OVERLAY”和“NOOVERLAY”来允许和禁止数据段覆盖功能,“OVERLAY”是 BL51/Lx51 的默认连接方式。

外部符号指向其他模块中的地址。一个已声明的外部符号由具有相同名字的公共符号决定,外部参考地址由公共参考地址决定,它们与符号类型(DATA、IDATA、XDATA、CODE、BIT 或 NUMBER)有关。如果数据类型不符,或未发现外部符号参考地址的公共符号,则会产生错误。公共符号的绝对地址在段定位之后决定。

在段分配和外部及公共参考地址处理完成之后, BL51/Lx51 将计算程序模块中所有的再定位地址和外部地址以确定目标代码的绝对地址,如果生成的目标文件中有调试用的符号信息,它们将反映最新的计算值。

BL51 将生成 Intel OMF51 格式的绝对目标文件, Lx51 将生成 Keil OMf51 格式的绝对目标文件,绝对目标文件中可包含调试用的附加符号信息,带有调试信息的目标文件可在 μ Vision2 中或兼容硬件仿真器中进行源程序的符号化调试。绝对目标文件还可以采用符号转换程序 OH51 转换成 Intel HEX 文件以供 EPROM 编程。

BL51/Lx51 完成连接定位之后将产生一个很有用的列表文件,其扩展名为.M51 或.MAP。列表文件中包括:

- 所有被处理模块的文件名和模块名;
- 存储器分配表,该表中包含段地址、段名、段类型以及段定位方式;
- OVERLAY MAP 表,该表用来显示生成的程序结构及数据位置与位段功能,其中包含所有的代码段,与其共存的数据段和位段被标记为 OVERLAYABLE;
- 符号信息表,该表中包含输入文件中的符号信息;
- 交叉参考报告表,该表包含按字母顺序排列的所有 PUBLIC 和 EXTERN 符号,并列出符号类型的模块名,先列出定义了 PUBLIC 符号的模块名,然后列出定义了 EXTERN 符号的模块名,如果没有 PUBLIC 符号则列出“** UNRESOLVED **”信息,交叉参考报告表的产生与否受控制命令“IXREF”的控制;
- 如果连接时发生错误,还将产生一个错误列表,错误列表出现在列表文件的末尾,其中列出了所有发生的错误及错误原因。

经常察看列表文件可以帮助用户了解和分析所编写的程序中变量的分配是否合理、代码是否优化、堆栈是否溢出等。尤其是采用 C 语言编程时关于堆栈的问题,堆栈栈顶的确定是由 Cx51 编译器和 BL51/Lx51 连接器根据程序中变量的分配自动完成的,如果栈顶的位置安排得不尽合理,程序在实际运行中就可能发生堆栈溢出而导致系统崩溃,因此必要时应对程序中的变量作适当调整,以确保在程序运行过程中不会发生堆栈溢出。

11.1.2 连接定位器的引用

BL51/Lx51 连接定位器有许多控制命令和参数,在 Windows 集成开发环境 μ Vision2 中,通过“Project 菜单/Options for Target 选项/BL51 Locate (或 Lx51 Locate) 标签页/BL51 Misc (或 Lx51 Misc) 标签页”可以很方便地对这些命令进行设置。连接定位控制命令设置完成后,通过 μ Vision2 中“Project 菜单/Build target 选项”对当前项目进行整体创建,即可根据所设置的控制命令对浮动目标代码模块进行连接定位。也可以通过命令行引用 BL51/Lx51 连接定位器,在命令行引用时基本格式如下:

BL51 输入文件表 [TO 输出文件] [控制命令参数表]

Lx51 输入文件表 [TO 输出文件] [控制命令参数表]

其中,“BL51/Lx51”是连接定位器调用命令。

“输入文件表”是一个用逗号隔开的文件名表。表中的每个文件都是可再定位的程序模块,它们最后将被组合到一个绝对目标程序模块中。需要注意的是,输入文件必须带有扩展名,纯目标文件使用扩展名“.OBJ”,库文件使用扩展名“.LIB”。如果输入文件不给出扩展名将在连接时发生错误。对于库文件允许在文件名后面的括号内指定欲包含的模块,每个模块名之间用逗号隔开。

“输出文件”是将产生的绝对目标程序模块的文件名。如果省略该选项,则以输入文件表中第一个文件名的基本名作为输出文件的基本名。输出文件名同时也作为列表文件的基本名。

“控制命令参数表”中包含所有需要的命令和参数,每个命令之间用逗号隔开。

下面是几个在命令行调用连接定位器的例子。

例 11.1: BL51 C:\MYDIR\PROG.OBJ TO C:\MYDIR\PRG.ABS。

本例将对输入文件“C:\MYDIR\PROG.OBJ”进行连接,所产生的绝对目标程序文件为:“C:\MYDIR\PROG.ABS”,列表文件为 C:\MYDIR\PRG.M51。

例 11.2: Lx51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ TO SAMPLE.ABS。

本例将三个输入文件“SAMPLE1.OBJ”、“SAMPLE2.OBJ”和“SAMPLE3.OBJ”连接成一个单一的绝对目标程序文件“SAMPLE.ABS”。

例 11.3: BL51 PROG1.OBJ, PROG2.OBJ, UTILITY.LIB。

本例将两个输入文件“PROG1.OBJ”、“PROG2.OBJ”与库文件“UTILITY.LIB”连接在一起,库中所需要的模块被自动连接,而不需要的模块将不会出现在生成的绝对目标文件中。输出绝对目标文件为:PROG1,列表文件为 PROG1.M51。

例 11.4: Lx51 PROG1.OBJ, PROG2.OBJ, UTILITY.LIB(FPMUL, FPDIV)。

与上例类似,不同的是指出了库文件“UTILITY.LIB”中的模块名“FPMUL”和“FPDIV”,因此除了自动连接库文件“UTILITY.LIB”中所需要的模块之外,还会将这两个模块连接到绝对目标文件中去,不论它们是否需要都将被连接。

也可以采用命令文件方式引用 BL51/Lx51 连接定位器,格式如下:

BL51 @命令文件

Lx51 @命令文件

其中，“命令文件”包括了输入文件表、输出文件以及控制命令参数。命令文件可以采用标准 ASCII 文本编辑器来生成。

例如，先用文本编辑器生成一个命令文件 PROJ1.LIN，其内容如下：

```
PROG1.OBJ, /* Program Module 1 */
```

```
PROG2.OBJ, // program module 2
```

```
UTILITY.LIB (FPMUL, FPDIV) ; include always FPMUL and FPDIV
```

然后按如下格式引用 BL51/Lx51 连接定位器：

```
Lx51 @PROJECT.LIN
```

连接定位结果与例 11.4 完全相同。

BL51/Lx51 在对输入程序模块进行连接时，对于 C 语言程序模块还将在生成的目标文件开头加上一段“启动程序”（STARTUP.A51）代码。如果 C 语言程序中包含有明确初始化了的全局变量，则还要加上“初始化程序”（INIT.A51）代码。这些是由 BL51/Lx51 自动完成的，“启动程序”和“初始化程序”的目标模块已经包含在运行库中，并且不能指定连接。如果需要修改“启动程序”和“初始化程序”，可将修改后的目标文件作为 BL51 的一个输入文件，而不必修改运行库。

例 11.5：BL51 MYPROG.OBJ,STARTUP.OBJ,INIT.OBJ。

本例将输入文件 MYPROG.OBJ 与修改后的启动程序和初始化程序连接在一起，输出文件为 MYPROG，列表文件为 MYPROG.M51。

11.1.3 应用程序在存储器空间中的定位

一般来说 8051 单片机应用系统可以采用三种类型的 CPU 芯片：传统型 8051、扩展型 8051 和 Philips 8051Mx，这三类 CPU 芯片的内核完全相同并且指令系统兼容，但它们的存储器空间范围有所不同，因此应用程序代码的存储器空间定位也有所不同。此外 BL51/Lx51 连接定位器提供对代码分组的支持，可以使传统型 8051 单片机的程序存储器空间突破 64KB 的限制。

1. 传统型 8051

传统型 8051 具有 3 个存储器空间：片上 RAM 空间（DATA、BIT、IDATA）、片外 RAM 空间（XDATA）和代码空间（CODE）。

例 11.6：对于没有代码分组的传统型 8051 存储器空间定位，可以使用连接定位器 BL51 的“RAMSIZE”、“XDATA”和“CODE”控制命令，或使用 Lx51 的“CLASSES”控制命令直接指定存储器的定位空间。假设单片机应用系统具有如下存储器空间范围：

片上 RAM 空间 D:0x00~D:0x7F

片外 RAM 空间 X:0x0000~0x7FFF, X:0xF800~x:0xFFFF

代码与常数空间 C:0x0000~0x7FFF

为了将应用程序目标代码“PROG.OBJ”正确定位到上述存储器空间，应按如下命令行引用 BL51 连接定位器：

BL51 PROG.OBJ XDATA(0-0x7FFF, 0xF800-0xFFFF) CODE(0-0x7FFF) RAMSIZE(128)
也可以按如下命令行引用 Lx51 连接定位器:

```
Lx51 PROG.OBJ CLASSES(IDATA(D:0-D:0x7F),
                      XDATA(X:0-X:0x7FFF, X:0xF800-X:0xFFFF),
                      CODE(C:0-C:0x7FFF))
```

例 11.7: 对于具有代码分组的传统型 8051 存储器空间定位, 除了使用 BL51 的“RAMSIZE”、“XDATA”和“CODE”控制命令, 或 Lx51 的“CLASSES”控制命令之外, 还需要使用“BANKx”控制命令来指定分组代码存储器的定位空间。假设单片机应用系统具有如下存储器空间范围:

片上 RAM 空间	I:0x00~I:0x7F
片外 RAM 空间	X:0x0000~0xEFFF
代码与常数空间	C:0x0000~0x7FFF (公共区域)
	B0:0x8000~B3:0xffff (4 个代码组)

为了将分组应用程序目标代码“A.OBJ”、“B.OBJ”、“C.OBJ”、“D.OBJ”正确定位到上述存储器空间, 用户先要根据应用系统的硬件结构指定公共区域的大小, 并按如下命令行引用 BL51 连接定位器:

```
BL51 BANK0{A.OBJ}, BANK1{B.OBJ}, BANK2{C.OBJ}, BANK3{D.OBJ}
      XDATA(0-0xEFFF) BANKAREA(0x8000-0xFFFF) RAMSIZE(256)
```

也可以按如下命令行引用 Lx51 连接定位器:

```
Lx51 BANK0{A.OBJ}, BANK1{B.OBJ}, BANK2{C.OBJ}, BANK3{D.OBJ}
      CLASSES(IDATA(I:0-I:0xFF), XDATA(X:0-X:0xEFFF),
      CODE(C:0-C:0xFFFF)) BANKAREA(0x8000-0xFFFF)
```

2. 扩展型 8051

新一代扩展型 8051 的程序和数据存储器空间最大可以扩展到 16MB, 对于这种 64KB 之外的扩展存储器空间可以通过存储器类型 HDATA 和 HCONST 进行寻址, 并且只有扩展连接定位器 Lx51 才能支持这种额外的扩展存储器空间。

例 11.8: 在扩展型 8051 单片机的扩展存储器空间定位变量和常数。假设单片机应用系统具有如下存储器空间范围:

片上 RAM 空间	I:0x00~I:0xFF(256 字节)
片外 RAM 空间	X:0x0000~X:0x1FFFF (128KB)
代码与常数空间	C:0x0000~0xFFFFF(1MB)

为了将目标文件 MYPROG.OBJ 中的变量和常数正确定位到上述存储器空间, 应按如下命令行引用 Lx51 连接定位器:

```
Lx51 MYPROG.OBJ CLASSES(HDATA(X:0-X:0x1FFFF),
                      HCONST(C:0-C:0xFFFFF))
```

存储器类型 HDATA 和 HCONST 只能用于变量和常数的定位, 程序代码采用与传统型 8051 单片机相同的代码分组方式定位到扩展存储器空间。

例 11.9: 按如下命令行引用 Lx51 可以在扩展型 8051 单片机的扩展存储器空间同时定位变量、常数以及分组代码。

```
Lx51 BANK0{A.OBJ}, BANK1{B.OBJ}, BANK2{C.OBJ}, BANK3{D.OBJ}
      CLASSES(IDATA(I:0-I:0xFF), XDATA(X:0-X:0xFFFF),
      HDATA(X:0-X:0xFFFF), HCONST(C:0-C:0xFFFFF)
      CODE(C:0-C:0xFFFFF)) BANKAREA(0x8000-0xFFFF)
```

3. Philips 80C51Mx

Philips 公司新近推出的 80C51Mx 单片机, 除了具有标准 8051 的 DATA/IDATA、XDATA、CODE 存储器空间之外, 还可以将片外数据存储器 and 程序存储器分别扩展到最大 8MB, 从而获得最大 16MB 的线性地址空间。

例 11.10: 采用 Lx51 在 Philips 80C51Mx 的存储器空间内定位变量和常数。假设 80C51Mx 单片机应用系统具有如下存储器空间范围:

片上 RAM 空间	7F:0000H~7F:03FFH
RAM 空间	00:0000H~01:FFFFH
代码与常数空间	80:0000H~83:FFFFH

为了将目标文件 MYPROG.OBJ 中正确定位到上述存储器空间, 应按如下命令行引用 Lx51 连接定位器:

```
Lx51 MYPROG.OBJ CLASSES(HDATA(0-0xFFFF),
      EDATA(0x7F0000-0x7F03FF),
      ECODE(0x800000-0x83FFFF),
      HCONST(0x800000-0x83FFFF))
```

对于 Ax51 宏汇编器可以采用 ECODE 存储器类型来指定全部 8MB 代码地址空间作为程序存储器空间。对于 Cx51 编译器时则需要采用与传统型 8051 单片机相同的代码分组方式将程序代码定位到 ECODE 存储器空间。

例 11.11: 采用如下命令行可以在 Philips 80C51Mx 的存储器空间内以代码分组方式定位目标程序。

```
Lx51 BANK0{A.OBJ}, BANK1{B.OBJ}, BANK2{C.OBJ}, BANK3{D.OBJ}
      CLASSES(HDATA(0-0xFFFF), EDATA(0x7F0000-0x7F03FF),
      ECODE(0x800000-0x83FFFF), HCONST(0x800000-0x83FFFF))
```

11.1.4 数据覆盖

众所周知, 8051 单片机的硬件堆栈空间十分有限, 因此 Cx51 程序的局部变量和函数参数被存放在固定的存储器空间而不是堆栈空间。BL51/Lx51 连接定位器将对用户应用程序的结构进行分析, 生成一个“调用树”(call tree), 并且对包含局部变量和函数参数的数据段进行必要的覆盖。在某些不允许进行数据覆盖的情形下, 则需要采用 Lx51 的“NOOVERLAY”控制命令来禁止数据覆盖。

默认状态下, 如果函数之间没有相互调用, BL51/Lx51 将会对各个函数的局部变量和

参数进行覆盖分析，并生成一个函数调用树。函数的局部数据段和位段根据其段名决定，如果满足以下条件，一个函数的局部数据段和位段将与其他函数的局部数据段和位段产生覆盖：

- 函数之间没有调用参考；
- 函数只能被主程序或一个中断服务程序所调用；
- 段定义必须遵守如表 11-2 所示的段名规则。

BL51/Lx51 连接定位器通常只对具有相同类型的局部段进行覆盖分析，被覆盖的段必须具有 OVERLAYABLE 属性，Cx51 和 PL/M51 编译器可以自动为局部数据段和局部位段生成该属性。对于汇编语言程序中需要进行覆盖的段，必须在程序中明确定义成 OVERLAYABLE 再定位类型，具体定义方法请参见第 10 章 Ax51 宏汇编器中关于 SEGMENT 伪指令一节，并且段定义也必须遵守如表 11-2 所示的段名规则。

表 11-2 局部段命名规则

段类型	Cx51 的段名	PL/M51 的段名
CODE	?PR? function_name? module_name	?module_name? PR
BIT	?BI? function_name? module_name	?module_name? BI
DATA	?DT? function_name? module_name	?module_name? DT
IDATA	?ID? function_name? module_name	—
XDATA	?XD? function_name? module_name	—
PDATA	?PD? function_name? module_name	—
EDATA	?ED? function_name? module_name	—
EBIT	?EB? function_name? module_name	—
HDATA	?HD? function_name? module_name	—

例 11.12：汇编语言程序的段定义。

```
?PR?func1?module1 SEGMENT CODE                ;segment for func1 code
?DT?func1?module1 SEGMENT DATA OVERLAYABLE ;data segment belongs to func1

                                RSEG ?DT?func1?module1
func1_var:                      DS 10                ;space for local variables in func1

                                RSEG ?PR?func1?module1
func1:                          MOV func1_var,A
                                RET
```

在绝大多数情况之下 BL51/Lx51 可以进行正确的段覆盖分析，但在某些特殊情形下需要采用 OVERLAY 命令对段的覆盖分析进行调整，以确保在连接定位时能产生正确的覆盖结果。OVERLAY 命令允许改变在覆盖分析过程中使用的调用参考。一般来说在下列情形需要采用 OVERLAY 命令对段的覆盖分析进行调整：

- 将函数指针作为参数进行传递或返回；
- 使用了已初始化了的函数指针数组；

- 用户程序中包含有实时操作系统。

当已经知道应用程序结构时采用 OVERLAY 命令是很容易的, 如果对某些数据段是否需要覆盖存在怀疑, 则可以采用以下方法禁止对这些段进行覆盖:

- 采用 BL51/Lx51 连接定位器的控制命令 NOOVERLAY 禁止对整个应用程序进行数据覆盖;
- 采用 BL51/Lx51 连接定位器的控制命令 OVERLAY(sfname !*), 禁止对函数 sfname 的局部数据段和位段进行覆盖;
- 对于不希望进行数据段覆盖的 Cx51 程序部分, 可采用 Cx51 编译器的控制命令 OPTIMIZE(1)进行编译, 使生成的这一部分目标代码没有可覆盖(OVERLAYABLE)类型, 因此也就不会被 BL51/Lx51 连接定位器所覆盖。

11.1.5 代码分组

BL51/Lx51 连接定位器支持分组连接定位, 允许生成代码长度大于 64K 字节的 8051 目标程序, BL51 可以管理一个公共区域和最多 32 个代码组区域, 每个代码组最大为 64K 字节, Lx51 可以管理多达 16MB 的代码和外部数据存储器空间。

1. 公共区域

公共区域可在任何时候由所有的代码组使用。由于硬件设计的不同, 公共区域中的代码可以与每个代码组中的代码重复, 也可以不重复。用户应根据不同的要求来设计系统的公共区域和代码组区域, 没有一般的准则。

下面的一些代码块必须放在公共区域中。

- 复位和中断向量。由于 8051 的复位和中断可能发生在任何时刻, 所以复位和中断跳转命令必须安排在公共区域, 因此, BL51/Lx51 在公共区域中定位这些绝对代码段。
- 代码常数。按缺省的默认值, BL51/Lx51 将包含常数的所有代码段定位在公共区域中。用户也可采用 BL51/Lx51 的控制命令在代码组中定位这些常数段, 但这时必须保证在存取常数时能够选择正确的代码组。
- 中断函数。中断函数必须总是放在公共区域, 但中断函数可以调用任何代码组中的函数。如果企图在代码组中定位中断函数时 BL51/Lx51 将产生警告错误。
- 组切换跳转表。用于切换代码组的程序代码必须放在公共区域, 因为在所有代码组都需要使用这些切换代码。按缺省默认值, BL51/Lx51 将在公共区域定位这些代码段, 用户不能将它们重新定位在代码组中。
- 库函数。虽然 8051 的工作寄存器 R0~R7 可用于参数的传递, 且不受代码组切换的影响, 但库函数的调用有时需要采用一些别的参数传递方法, 而这些方法有时会受到组间切换的影响, 所以 BL51/Lx51 总是将库函数段放置在公共区域, 用户不能重新在代码组中定位库函数段。

一般来说要预计公共区域的大小是比较困难的, 其大小取决于应用程序及硬件配置。如果用做公共区域的 ROM 空间不足以装下全部公共代码, BL51/Lx51 将把剩余部分复制到每一个代码组中去。如果硬件配置中没有提供 ROM 空间作为公共区域, BL51/Lx51 将把全部公共代码都复制到每一个代码组中去。

2. 代码组

传统 8051 提供了 16 根地址线, 只能够访问 64KB 的代码空间。8051 单片机本身并不能支持代码组切换, 切换是通过附加的地址线来完成的, 这些附加的地址线可以是 8051 的硬件 I/O 口线, 也可以是用 XDATA 存储器映像实现的扩展 I/O 口。代码组切换在软件上受配置文件 L51_BANK.A51 的支持, 用户必须根据自己的硬件结构适当修改该配置文件。

3. 采用代码组切换的最佳程序结构

BL51/Lx51 会自动产生一个跳转表, 表中包含有放置在代码组中的函数信息, 以及从公共区域或其他代码组中进行函数调用的信息。在同一个代码组中调用函数时不使用跳转表, 所以精心地将函数安排在同一个代码组将会极大地提高处理速度。

代码组切换需要大约 50 个机器周期和 2 个字节的堆栈空间, 因此应当仔细安排程序结构以尽量少用代码组之间的切换。对于经常使用的函数或经常在代码组之间调用的函数应安排在公共区域。

在 Windows 集成开发环境 μ Vision2 中实现代码分组连接是很方便的, 首先将需要进行代码分组的每个 Cx51 源程序文件添加到一个项目之中, 通过“Project 菜单/Options for Target 选项/Target 标签页”, 将“Code Rom Size”栏设置为“Large”模式, 同时选中左下角的“Code Banking”复选框, 并根据硬件配置决定需要采用的代码组数在“Banks”栏内键入相应的数字, 在“Bank Area”栏键入代码组的起始 (Start) 和结束 (End) 地址。接下来对项目中的每个分组源程序文件分别编译并通过相应列表 (LST) 文件察看其代码长度 (CODE SIZE), 如果代码长度小于所设置代码组容量, 则将鼠标指向各个分组源程序文件并单击右键, 选中右键菜单中的“Options for File”选项, 在“Properties”标签页的“Code Bank”栏中选取合适的分组号, 未指定分组号的文件将被自动放入公共 (COMMON) 区域, 也可以对相应文件指定公共区域, 这样就完成了分组连接的设置, 最后对整个项目进行创建 (Build Target) 即可一次完成对当前项目中所有文件的编译和分组连接。

如果在命令行上调用 BL51/Lx51 进行分组连接, 一般格式如下:

BL51/Lx51 输入文件表 [TO 输出文件] 控制命令表

其中, “BL51/Lx51”是分组连接定位器的调用命令。

“输入文件表”中列出了将要被连接定位的各个模块和每个模块将要放入的代码组。

可选项“输出文件”是将产生的 OMF51 绝对目标文件, 如果省略该选项, 则以项目文件名作为输出文件名, 输出文件名同时也作为列表文件名。

“控制命令表”是 BL51/Lx51 的分组控制命令。关于分组控制命令的详细介绍请参见本章关于连接控制命令详解一节。

输入文件表具有如下形式:

BANKx{filename[(modulename)][,filename...]}[,...]

COMMON{filename[(modulename)][,filename...]}[,...]

其中, “BANKx”表示代码组, 其范围是 BANK0~BANK31。

“filename”是目标文件或库文件名, 对于库文件可以其在后面的括号内指定模块名。

“modulname”是库文件中的目标模块名。

“COMMON”表示可以由所有代码组寻址的公共区域，未明确指定代码组的所有程序模块都将被定位到公共区域。

下面是两个在命令行使用 BL51 的例子。

例 11.13: BL51 BANK0{PROG0.OBJ,PROG1.OBJ,PROG2.OBJ},
BANK1{PROG3.OBJ,PROG4.OBJ},
COMMON{PROG5.OBJ,PROG6.OBJ}

本例中，模块 PROG0.OBJ, PROG1.OBJ, PROG2.OBJ 将定位在代码组 BANK0 中。模块 PORG3.OBJ, PROG4.OBJ 定为在代码组 BANK1 中。模块 PROG5.OBJ,PROG6.OBJ 和其他具有常数的代码段定位在公共区域 COMMON 中。所生成的输出文件名默认为当前项目名。

例 11.14: Lx51 BANK0{UTIL.LIB(MOD1)},
BANK1{MOD2.OBJ},
COMMOM{MOD3.OBJ,UTIL.LIB},
TO MYPROG.ABS BANKAREA(8000H,0FFFFH)

本例中，库文件 UTIL.LIB 中的模块 MOD1 被放置在 BANK0 中。模块 MOD2 被放置在 BANK1 中。模块 MOD3 和库文件 UTIL.LIB 中其他模块将被放置在公共区域。所生成的输出为 MYPROG.ABS，每个代码组的起始地址和结束地址分别为 8000H 和 0FFFFH。

在调用 BL51/Lx51 连接定位器时还可以通过分组控制命令“BANKx”和“SEGMENTS”来调整目标代码段的定位地址，也可以将常数、数组或大型表格放置在代码组中，但用户必须保证 Cx51 程序能够通过恰当的代码组访问这些数据，这一点可以通过分组配置文件“L51_BANK.A51”中的“switchbank”函数来实现。

11.1.6 分组配置

在采用 BL51/Lx51 对目标程序进行分组连接定位时，需要根据硬件配置决定所需要的分组数以及分组切换方法，为此提供了相应的配置文件。对于传统 8051 单片机可通过“L51_BANK.A51”文件进行配置，对于 Philips 80C51Mx 单片机可以通过“MXBANK.A51”文件进行配置，这两个配置文件的详细清单位于\C51\LIB 目录之下，用户可以根据自己需要对它们作适当的修改。

在“L51_BANK.A51”文件的开始处有一些用于分组配置的 EQU 语句，它们的功能如下。

- | | |
|-----------|--|
| ?B_NBANKS | 表示所支持的分组数，采用连接定位器 BL51 时分组数应在 2~32 之间。采用扩展连接定位器 Lx51 时分组数最大可达 64。另外，分 2 组需要 1 根 8051 端口地址线，分 3~4 组需要 2 根 8051 端口地址线，分 5~8 组需要 3 根 8051 端口地址线，分 9~16 组需要 4 根 8051 端口地址线，分 17~32 组需要 5 根 8051 端口地址线。 |
| ?B_MODE | 表示是采用 8051 的 I/O 端口还是采用 XDATA 扩展端口进行分 |

	组地址扩展。“0”表示采用 8051 的 I/O 端口进行分组地址扩展，“1”表示采用 XDATA 存储器映像扩展端口进行分组地址扩展，“2”表示采用 80C51Mx 的扩展地址线，“4”表示采用用户提供的分组切换代码。
?B_RTX	表示是否采用实时操作系统 RTX-51，RTX-51 仅支持 ?B_MODE=0 和 ?B_MODE=1 方式。
?B_VARBANKING	允许在 XDATA 空间和 CODE 空间进行变量分组。进行变量分组时必须采用 Lx51，而不能采用 BL51 连接定位器。
?B_RST_BANK	指定 CPU 复位后所使用的默认组。设置值由扩展连接定位器 Lx51 所使用，以减少“组间调用表”的入口，当设置值为 0xff 时将禁止 Lx51 的这一优化功能。另外设置值不能由 BL51 所使用。
?B_PORT	如果 ?B_MODE=0，则必须用 ?B_PORT 来定义一个 8051 单片机 I/O 端口进行地址扩展，默认值为 P1。
?B_XDATAPORT	如果 ?B_MODE=1，则必须用 ?B_XDATAPORT 来确定用做分组地址扩展端口的外部数据存储器单元地址，其值在 0H~0FFFFH 之间，默认值为 0FFFFH。在这种方式下，必须利用启动代码文件(STARTUP.A51)在 Cx51 程序开始时将符号 ?B_CURRENTBANK 和 ?B_XDATAPORT 所指定的存储器单元清 0。
?B_FIRSTBIT	表示采用所定义端口的哪一位作为第一根扩展地址线，当 ?B_MODE=0 时，该参数的默认值为 3，表示 P1.3 口线是用于分组地址扩展的第一根地址线。若扩展时要使用多条口线，可根据组数要求依次使用 P1.3、P1.4、P1.5、P1.6 和 P1.7，其余未使用的 P1 口线可用于其他目的。但是当使用 XDATA 存储器映像作为扩展端口地址时(?B_MODE=1)，XDATA 存储器映像单元中未使用的其他位不能用于其他目的。
SWITCHx	表示当 ?B_MODE=4 时，由用户提供分组切换代码所需要的宏。对于每一个存储器分组都必须定义一个单独的分组切换代码宏。宏的个数必须与 ?B_NBANKS 定义个数一致。例如，分 4 组时需要定义 4 个宏：SWITCH0~SWITCH4。每个宏所产生的代码，其字节数必须完全相同，如果字节数不同，可以增加 NOP 指令迫使其相同。同时还必须保证 CPU 启动时已经选定了一个确定状态。实时操作系统 RTX-51 不支持这种代码切换方式。

在 L51_BANK.A51 文件中除了以上与硬件有关的符号之外，还包括以下公共符号。

?B_CURRENTBANK	它是 8051 内部 DATA 或 SFR 存储器空间的地址符号，该地址单元中存储了当前选择的存储器分组号，其值可被读出修改，但在多数情况下，修改该值不能影响代码组的切换。真正影响组切换的是 ?B_NBANKS 和 ?B_FIRSTBIT 的设置，因此在进行
----------------	---

存储器分组时必须将不需要的位屏蔽掉。

SWITCHBANK

这是一个与 Cx51 兼容的函数,它允许用户在 Cx51 源程序中选择存储器分组。该函数只能从公共区域进行调用,函数原型为:

```
extern void switchbank(unsigned char bank_number);
```

例如,函数调用语句“switchbank(0)”将选择第 0 组存储器。

下面是举几个根据不同分组扩展硬件逻辑来配置 L51_BANK.A51 的例子。

例 11.15: 将 256KB 的 EPROM 存储器分组成为 64KB×4 的存储器配置。

采用 1 个 256K 字节的 EPROM 芯片,可以分成 4 个存储器组,每组 64K 字节。图 11.1 所示为各个存储器分组所对应的地址分配,图 11.2 所示为硬件分组切换逻辑电路图,存储器分组之间的切换通过 8051 的 P1.5 和 P3.3 来实现。

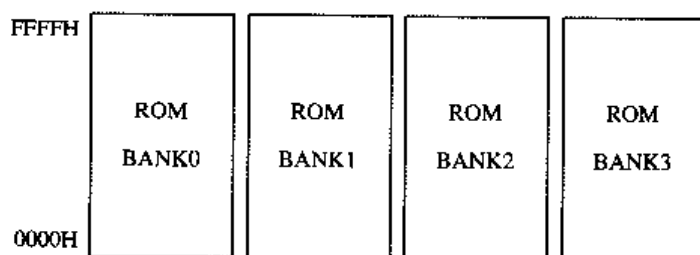


图 11.1 存储器分组的逻辑地址分配

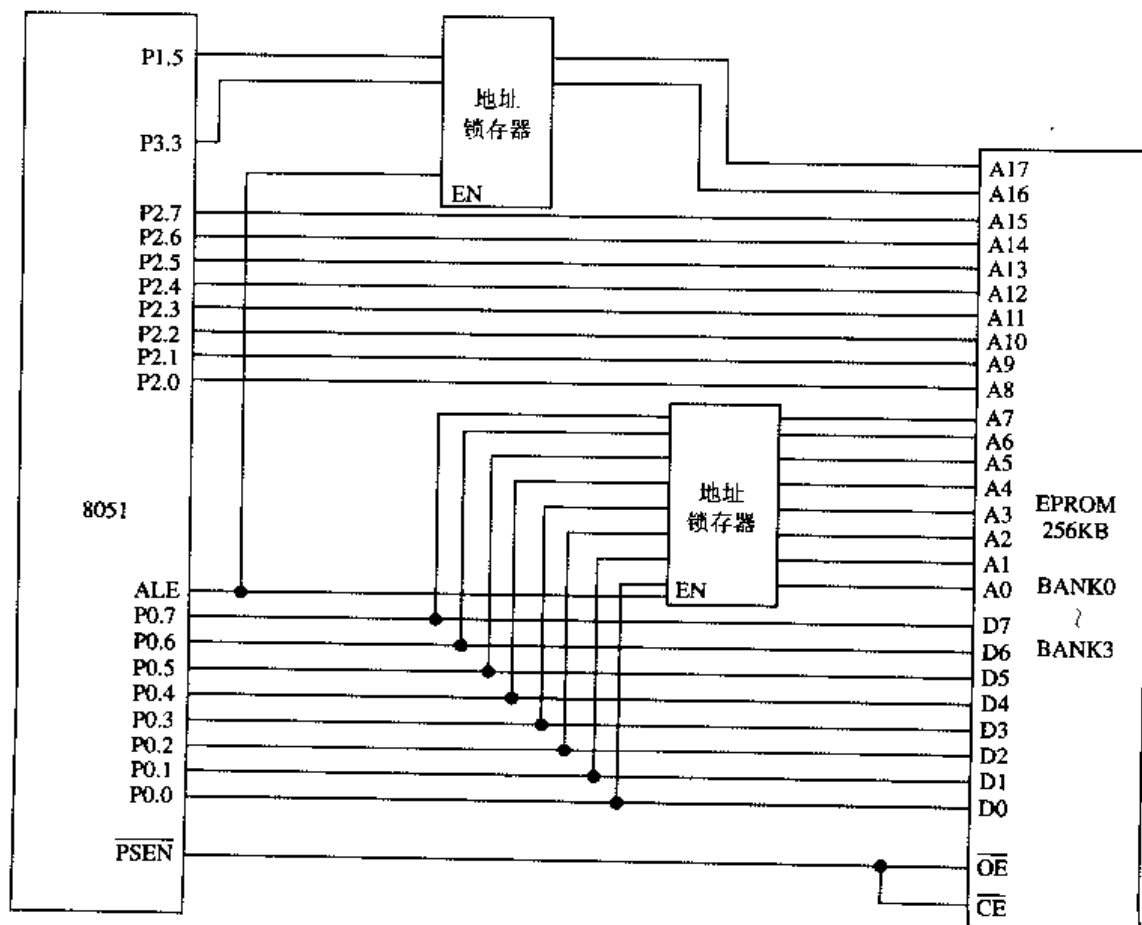


图 11.2 硬件分组切换逻辑电路

对于图 11.2 所示的存储器分组切换逻辑, 可采用如下方式配置 L51_BANK.A51:

```
?B_NBANKS      EQU      4      ;要求分成 4 组。
?B_MODE        EQU      4      ;采用用户提供的分组切换代码。
```

在 L51_BANK.A51 文件中从 “IF ?B_MODE=4” 处开始, 定义了用于代码组之间切换的宏 (MACRO), 这部分代码如下:

```
P1              DATA    90H      ; I/O 口地址
P3              DATA    0B0H      ;
;
SWITCH0         MACRO          ; 切换到存储器分组 0
                CLR      P1.5      ; 清零 P1.5
                CLR      P3.3      ; 清零 P3.3
                ENDM
;
SWITCH1         MACRO          ; 切换到存储器分组 1
                SETB     P1.5      ; 置 1 P1.5
                CLR      P3.3      ; 清零 P3.3
                ENDM
;
SWITCH2         MACRO          ; 切换到存储器分组 2
                CLR      P1.5      ; 清零 P1.5
                SETB     P3.3      ; 置 1 P3.3
                ENDM
;
SWITCH3         MACRO          ; 切换到存储器分组 3
                SETB     P1.5      ; 置 1 P1.5
                SETB     P3.3      ; 置 1 P1.3
                ENDM
;
```

用户应保证 CPU 复位之后从一个确定的状态开始执行指令, 为此需要在启动代码文件 STARTUP.A51 中添加如下代码:

```
                MOV      SP,#?STACK-1
; added for bank switching
P1              DATA    90H      ; 定义 P1 口
P3              DATA    0B0H      ; 定义 P3 口
                EXTRN    DATA (?B_CURRENTBANK)
                MOV      ?B_CURRENTBANK,#0 ; 选择代码组 0
                CLR      P3.3      ; 清零 P3.3
                CLR      P1.5      ; 清零 P1.5
                JMP      ?C_START
```

BL51 会自动地将公共区域的值复制到每个代码组中, 所以所有 EPROM 组的内容在公共区域的地址范围是相同的。由于已将代码区的地址空间缺省地定义为 0H~0FFFFH,

所以不必使用 BL51 的 BANKAREA 参数。

例 11.16: 带有片上程序存储器 ROM 的代码分组配置。

有些 8051 芯片提供了专用特殊功能寄存器用于配置片上 ROM 存储器空间, 用户可以利用这一特点进行代码分组设计。假如, 原来设计中采用无片上 ROM 的 DALLAS 80C320, 外部扩展了 64KB 程序存储器, 现在采用代码分组技术, 用 DALLAS 80C520 取代原来得 80C320 芯片, 可使用户的程序存储器空间增加 16KB。DALLAS 80C520 提供了一个 ROMSIZE 特殊功能寄存器, 利用该寄存器可以实现片上 ROM 与片外 ROM 之间的切换。存储器分组所对应的地址分配如图 11.3 所示。

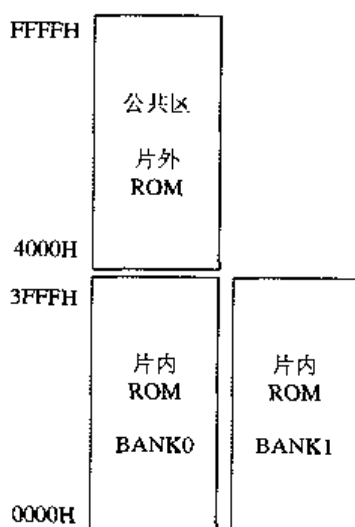


图 11.3 带有片上程序存储器 ROM 的代码分组地址分配

对于图 11.3 所示的 ROM 存储器分组, 可采用如下方式配置 L51_BANK.A51:

```
?B_NBANKS      EQU      2           ; 要求分成 2 组。
?B_MODE         EQU      4           ; 采用用户提供的分组切换代码。
```

对于 L51_BANK.A51 文件中从“IF ?B_MODE=4”处开始定义的实现代码组之间切换的宏 (MACRO) 配置如下:

```
ROMSIZE          DATA  0C2H          ; 特殊功能寄存器地址
;
SWITCH0           MACRO                ; 切换到存储器分组 0
MOV  ROMSIZE,#05H ; 允许片上 16KB ROM
ENDM              ;
;
SWITCH1           MACRO                ; 切换到存储器分组 1
MOV  ROMSIZE,#00H ; 禁止片上 16KB ROM
ENDM              ;
```

为了保证 CPU 复位之后从一个确定的状态开始执行指令, 需要在启动代码文件 STARTUP.A51 中添加如下代码:

```

MOV      SP, #?STACK-1
; 添加分组切换代码
ROMSIZE  DATA    0C2H          ; 定义特殊功能寄存器
;
EXTRN    DATA (?B_CURRENTBANK)
MOV      ?B_CURRENTBANK, #0      ; 选择代码组 0
MOV      ROMSIZE, #05H           ; 从片上 16KB ROM 开始执行指令
JMP      ?C_START
```

此外，引用 BL51 连接定位器时需要使用 BANKAREA 控制命令，如下所示：
BL51 ...BANKAREA(0,0X3FFF)

例 11.17：使用 8051 的外部数据存储器映像单元进行 64KB*8 的存储器分组。

本例将 512KB 的 EPROM 分为 8 组，每组为 64KB，图 11.4 所示为分组存储器的逻辑地址分配，图 11.5 所示为存储器分组硬件切换逻辑，本例采用 XDATA 存储器映像单元地址来寻址 512KB 的扩展 EPROM 存储器。根据该硬件逻辑电路可使用如下方法来配置 L51_BANK.A51：

```

?B_NBANKS EQU      8          ; 要求分成 8 组。
?B_MODE    EQU      1          ; 使用 XDATA 存储器映像单元作为扩展端口。
?B_XDATAPO EQU      0          ; 对于本例可使用任意 XDATA 地址。
?B_FIRSTBIT EQU      0          ; 使用 bit0 作为第一条地址线。
```

本例不需要在启动代码文件 STARTUP.A51 中添加代码。BL51 连接定位器会自动将公共区域中的数据和程序代码复制到每一个代码组中，从而使所有 EPROM 存储器分组中的内容在公共区域地址范围内都相同。由于默认设置已经将分组地址定义为 0000H~0FFFFH，因此不需要使用 BANKAREA 控制命令。

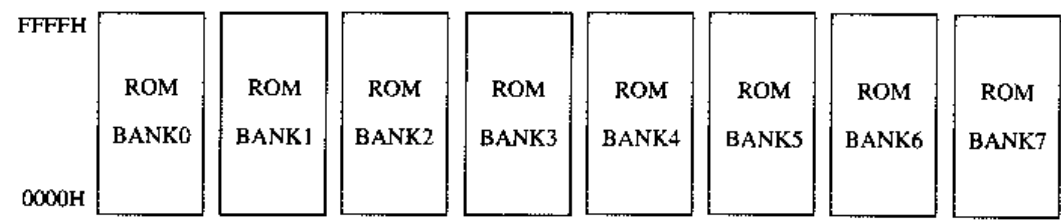


图 11.4 存储器分组的逻辑地址分配

例11.18：带有公共区域的存储器分组。

本例采用单个 256KB 的 EPROM 提供一个 32KB 的公共区域和 7 个 32KB 的代码分组。图 11.6 所示为存储器分组逻辑地址分配，图 11.7 所示为对应的硬件分组切换逻辑电路。根据地址译码逻辑，代码组 0 与公共区域相同，因此应用程序不能使用代码组 0，即不允许在 BANK0 内安排任何应用程序模块。本设计还提供了 256KB 的 XDATA 存储器用于变量分组，其分组存储器映像与代码分组相同。

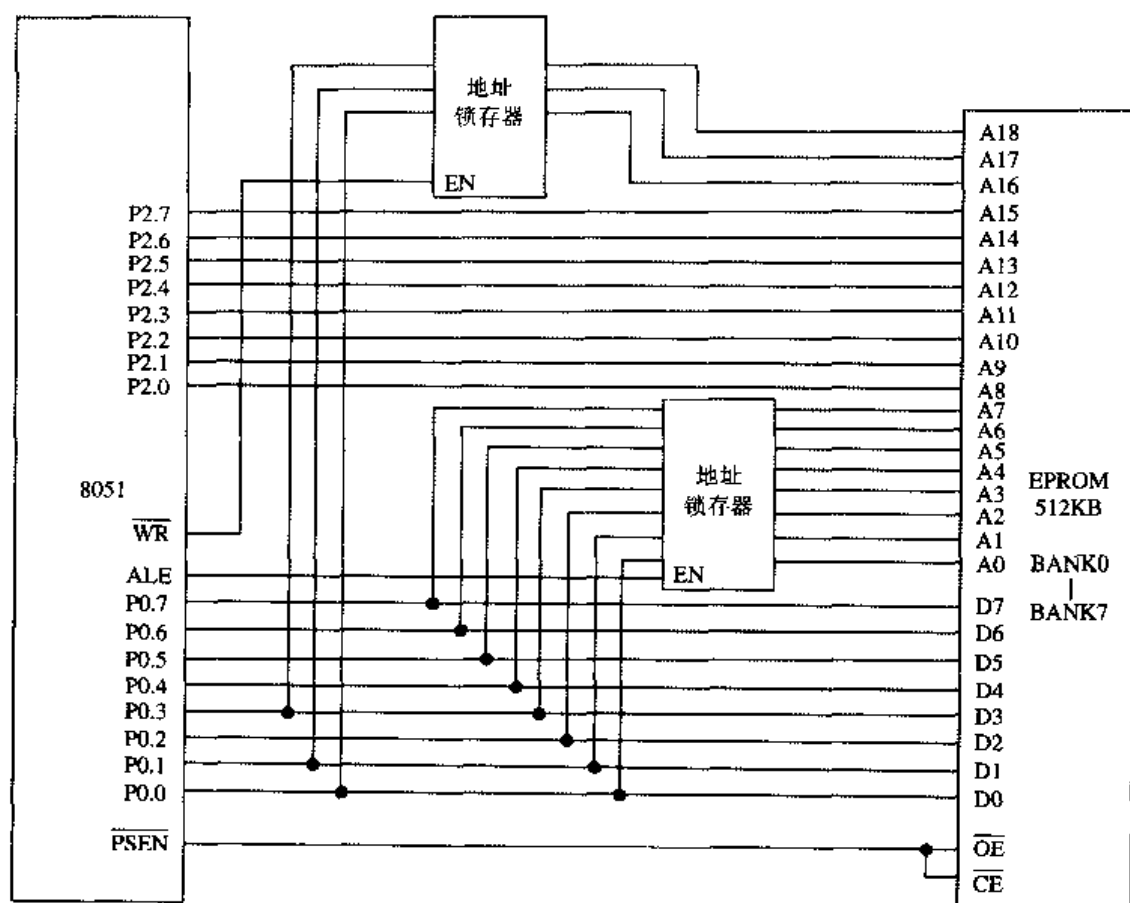


图 11.5 硬件分组切换逻辑电路

根据本例硬件逻辑，可采用如下方式配置 L51_BANK.A51:

```
?B_NBANKS      EQU      8           ; 要求分成 8 组。
?B_MODE        EQU      0           ; 通过片上 I/O 口进行分组切换。
?B_VAR_BANKING EQU      1           ; 也可以使用变量分组。
?B_PORT        EQU      90H         ; 采用 P1 口。
?B_FIRSTBIT    EQU      2           ; 采用 P1.2 作为第 1 根地址线。
```

对于本例来说引用 BL51 连接定位器时需要采用 BANKAREA 控制命令:

BL51 BANK1{A.OBJ},...,BANK7{G.OBJ}...BANKAREA(0X8000,0XFFFF)

如果要进行变量分组，则需要采用 Lx51 扩展连接定位器，并且还要使用 HDATA 和 HCONST 存储器类型来定义附加存储器。对于本例来说可以按如下方式设置存储器类型:

```
Lx51 BANK1 {A.OBJ},...,BANK7{G.OBJ} ... BANKAREA (0x8000,0xFFFF)
CLASSES (XDATA (X:0-X:0x7FFF),
        HDATA (X:0x18000-X:0x1FFFF,X:0x28000-X:0x2FFFF,
              X:0x38000-X:0x3FFFF,X:0x48000-X:0x4FFFF,
              X:0x58000-X:0x5FFFF,X:0x68000-X:0x6FFFF,
              X:0x78000-X:0x7FFFF),
        HCONST (C:0x18000-C:0x1FFFF,C:0x28000-C:0x2FFFF,
```

C:0x38000-C:0x3FFFF,C:0x48000-C:0x4FFFF,
C:0x58000-C:0x5FFFF,C:0x68000-C:0x6FFFF,
C:0x78000-C:0x7FFFF))

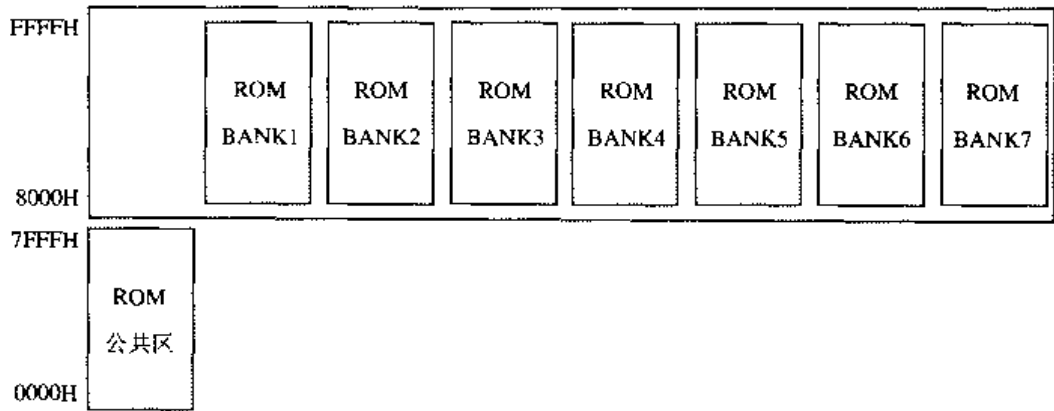


图 11.6 存储器分组的逻辑地址分配

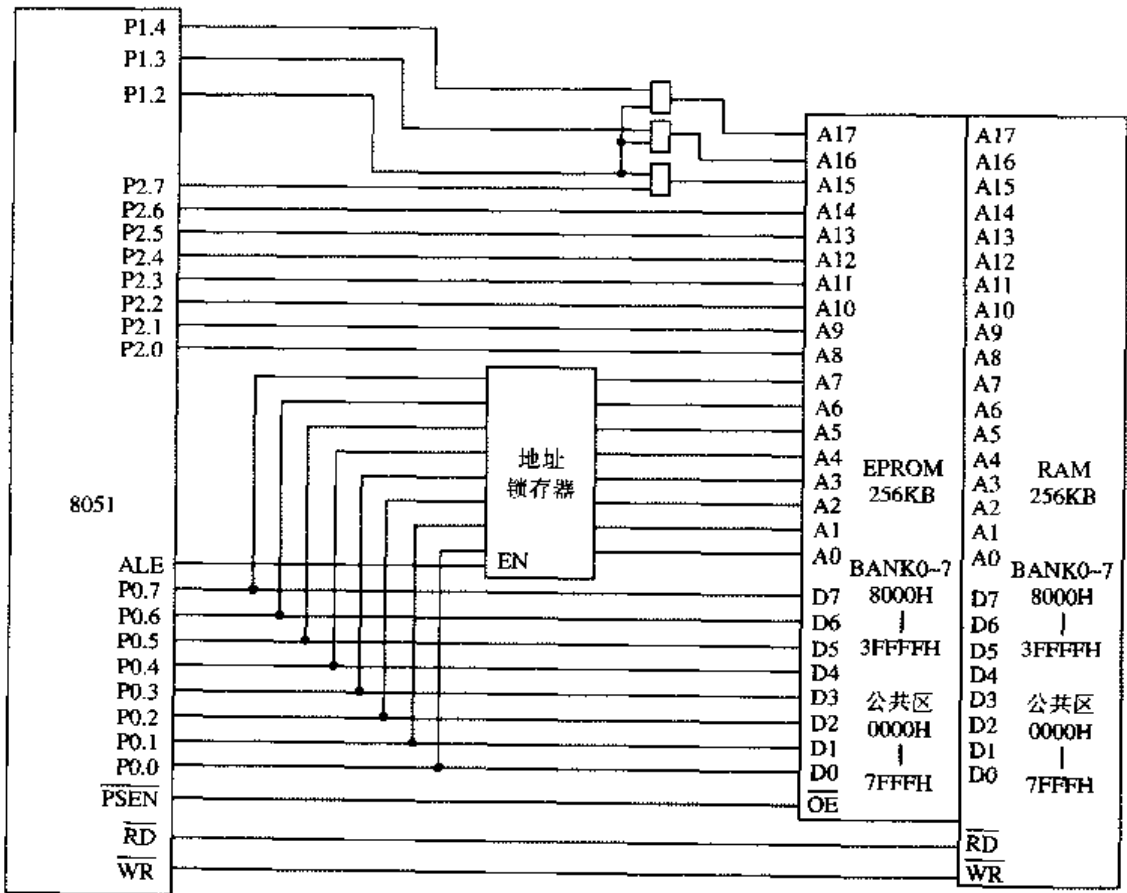


图 11.7 硬件分组切换逻辑电路

11.2 连接定位控制命令详解

BL51/Lx51 连接定位器具有多条控制命令，可分成四组：列表文件控制命令、输出文

件控制命令、段与存储器定位控制命令和高级语言控制命令。采用命令行方式引用连接定位器，可以清楚地指定需要进行连接定位的程序模块，但操作比较麻烦的，利用 Windows 集成开发环境 μ Vision2 则要方便得多，只要通过“Project 下拉菜单/Options for Target 选项”中的“BL51 Locate (或 Lx51 Locate) 标签页/”和“BL51 Misc (或 Lx51 Misc) 标签页/”键入恰当的连接定位控制命令，然后对项目进行整体创建 (Build Target)，即可一次完成对项目中的所有模块程序的连接定位。下面分类对 BL51/Lx51 的各个控制命令进行功能解释。

11.2.1 列表文件控制命令

BL51/Lx51 连接定位器将生成包含有连接定位处理信息的列表文件，又称为映像 (MAP) 文件。表 11-3 给出了 BL51/Lx51 的列表文件控制命令，用于指定列表文件名、文件格式、信息内容等，下面介绍各条命令的使用方法。

表 11-3 BL51/Lx51 的列表文件控制命令

BL51	Lx51	缩 写	功 能 说 明
DISABLEWARNING	DISABLEWARNING	DW	禁止产生指定的警告信息
IXREF	IXREF	IX	在列表文件中产生附加交叉报告表
—	[NO]COMMENTS	[NO]CO	[不]包含注释信息
[NO]LINES	[NO]LINES	[NO]LI	[不]在列表文件中产生行号
[NO]MAP	[NO]MAP	[NO]MA	[不]输出存储器映像
[NO]PRINT	[NO]PRINT	[NO]PR	[不]定义列表文件名
[NO]PUBLICS	[NO]PUBLICS	[NO]PU	[不]在列表文件中列出公共符号
[NO]SYMBOLS	[NO]SYMBOLS	[NO]SY	[不]在列表文件中列出局部符号
PAGELength	PAGELength	PL	设置列表文件每页最大行数
PAGEWIDTh	PAGEWIDTh	PW	设置列表文件每行最大宽度
PRINT	PRINT	PR	定义列表文件名
—	PRINTCONTROLS	PC	不包含指定的调试信息
—	PURGE	PU	不包含所有的调试信息
—	WARNINGLEVEL	WL	控制所产生警告信息的类型和程度

命 令 名: DISABLEWARNING

缩 写: DW

参 数: DISABLEWARNING (number, [...])

默 认 值: 输出显示所有警告信息。

μ Vision2: 在“Options 选项/BL51 Misc(Lx51 Misc)标签页/Warnings 栏/Disable Warning 框”中键入合适的数字。

功 能: DISABLEWARNING 命令用来有选择地禁止产生连接定位警告错误。被禁止的警告错误信息由命令后面括号内的数字指定。

例 子: Lx51 SAMPLE1.OBJ DISABLEWARNING(1, 5)

命 令 名: IXREF

缩 写: IX

参 数: IXREF (NOGENERATED, NOLIBRARIES)

默 认 值: 不产生交叉参考报告

μVision2: 位于“Options 选项/Listing 标签页”内的“Linker Listing”复选框、“Cross Reference”复选框以及“Generated Symbol”和“Library Symbol”复选框, 选中为肯定形式, 不选中为否定形式。

功 能: IXREF 命令用来控制在列表文件中产生一个交叉参考报告, 该命令的默认值为不产生交叉参考报告。参数 `nogenerated` 将抑制以“?”开头的符号, 这些符号通常是由 Cx51 (或 PL/M51) 编译器产生的, 用于调用内部函数或传递参数。参数 `nolibraries` 将抑制在库文件中定义的符号。

例 子: BL51 SAMPLE1.OBJ IXREF

Lx51 SAMPLE1.OBJ IXREF(nogenerated)

命 令 名: [NO]COMMENTS, 仅用于 Lx51

缩 写: [NO]CO

参 数: 无

默 认 值: 包含注释信息

μVision2: 位于“Options 选项/Listing 标签页”内的“Linker Listing”复选框、“Comment Records”复选框, 选中为肯定形式, 不选中为否定形式。

功 能: 该命令的否定形式 `NOCOMMENTS` 用于移去包含在输入文件表中列表文件和目标输出文件内的注释信息。注释信息是为了识别产生目标文件的编译器或汇编器而加入到目标模块文件中的。如果希望仅从列表文件中移去注释信息, 则必须使用“`PRINTCONTRLS`”命令。

另 见: OBJECTCONTROLS, PRINTCONTRLS 命令

例 子: Lx51 SAMPLE1.OBJ NOCOMMENTS

命 令 名: [NO]LINES, [NO]MAP, [NO]PUBLICS, [NO]SYMBOLS

缩 写: [NO]LI, [NO]MA, [NO]PU, [NO]SY

参 数: 无

默 认 值: LINES, MAP, PUBLICS, SYMBOLS

μVision2: 位于“Options 选项/Listing 标签页”内的“Linker Listing”复选框、“Line Numbers”复选框、“Memory Map”、“Public Symbols”、“Local Symbols”复选框选中为肯定形式, 不选中为否定形式。

功 能: 使用这些命令的否定形式可在列表文件中省略行号, 存储器映像, 公共符号或局部符号, 默认值为这些命令的肯定形式。

另 见: NODEBUGLINES, PRINTCONTROLS, OBJECTCONTROLS 命令

例 子: BL51 SAMPLE.OBJ NOMAP NOSYMBOLS

命 令 名: PAGELENGTH, PAGEWIDTH

缩 写: PL, PW

参 数: PAGELENGTH (value)

PAGEWIDTH (value)

默 认 值: PAGELENGTH(60)

PAGEWIDTH(132)

μ Vision2: 在“Options 选项/Listing 标签页”内的“Page Width”和“Page Length”框内键入合适的数字。

功 能: PAGELENGTH 命令用来设置列表文件每页最大行数, 指定的最小行数为 10, PAGEWIDTH 命令用来指定列表文件每页最大宽度, 允许指定范围为 72~132 列。如果不指定行数, 则采用默认值。

例 子: BL51 SAMPLE.OBJ PAGELENGTH(55) PAGEWIDTH(78)

命 令 名: [NO]PRINT

缩 写: [NO]PR

参 数: PRINT (filename)

默 认 值: PRINT (输出文件名.M51) 或 PRINT (输出文件名.MAP)

μ Vision2: 在“Options 选项/Listing 标签页”内, 单击“Select Folder for Listings”按钮来选定存放列表文件的目录。使用“NOPRINT”命令时应不选中“Linker Listing”复选框。

功 能: PRINT 命令用来定义列表文件名。如果不输入参数, BL51 将以输出文件的基本名加上扩展名“.M51”作为列表文件, Lx51 将以输出文件的基本名加上扩展名“.MAP”作为列表文件。NOPRINT 命令将抑制连接定位器产生列表文件。

例 子: Lx51 SAMPLE.OBJ TO MYPROG.ABS PRINT(SAMPLE.MAP)

命 令 名: PRINTCONTROLS, 仅用于 Lx51

缩 写: PC

参 数: PRINTCONTROLS (subcontrol[, ...])

默 认 值: 在列表文件输出所有调试信息。

μ Vision2: 在“Options 选项/Listing 标签页”内, 通过“Linker Listing”复选框以及它下面的各个子复选框来使用本命令。

功 能: PRINTCONTROLS 命令允许从列表文件中移去特定的调试信息。可以使用如下子控制命令参数:

NOCOMMENTS 注释记录

NOLINES 行号信息

NOPUBLICS 公共符号信息

	NOSYMBOLS	局部符号信息
	PURGE	全部调试信息
另 见:	NOCOMMENTS , NOLINES , NOPUBLICS , NOSYMBOLS , OBJECTCONTROLS, PURGE 命令	
例 子:	Lx51 SAMPLE.OBJ PRINTCONTROLS(NOLINES, NOSYMBOLS)	
命 令 名:	PURGE, 仅用于 Lx51	
缩 写:	PU	
参 数:	无	
默 认 值:	处理所有调试信息	
μVision2:	在“Options 选项/Lx51 Misc 标签页”内键入命令。	
功 能:	PURGE 命令允许从列表文件和目标文件中移去全部调试信息, 它与使用“NODCOMMENTS, NOLINES, NOPUBLICS, NOSYMBOLS”命令的作用相同。调试信息仅在对应用程序调试的过程中才需要, 它对执行代码没有影响。如果希望列表文件中不包含行号, 则必须使用 PRINTCONTROLS 命令。	
另 见:	NOCOMMENTS , NOLINES , NOPUBLICS , NOSYMBOLS , OBJECTCONTROLS, PRINTCONTROLS 命令	
例 子:	Lx51 SAMPLE.OBJ PURGE	
命 令 名:	WARNINGLEVEL, 仅用于 Lx51	
缩 写:	WL	
参 数:	数字 0~2	
默 认 值:	WARNINGLEVEL(2)	
μVision2:	在“Options 选项/Lx51 Misc 标签页/Warning 栏”内选取警告级别。	
功 能:	WARNINGLEVEL 命令允许有选择地抑制警告信息。警告级别及其功能如下:	
	警告级别	功 能
	0	禁止几乎所有连接警告信息。
	1	仅列出那些可能产生错误代码的警告信息, 包括数据类型不匹配等。
	2 (默认值)	列出所有警告信息。
例 子:	Lx51 SAMPLE.OBJ WL(1)	

11.2.2 输出文件控制命令

BL51/Lx51 可以生成绝对目标文件或分组目标文件。绝对目标文件中不包含再定位信息或外部参考, 它可以被装入仿真器进行调试, 也可以通过 OH51 转换器转换成 Intel HEX 文件进行 EPROM 编程。分组目标文件必须通过 OC51 转换器转换成单个绝对目标文件(每

组一个文件), 对这样的单个绝对目标文件通过 OH51 转换器转换成 Intel HEX 文件进行 EPROM 编程。BL51/Lx51 所生成的目标文件中可以包含调试信息, 以便于进行源程序符号化调试。也可以通过控制命令抑制目标文件中的调试信息。表 11-4 给出了 BL51/Lx51 的输出文件控制命令, 下面介绍各条命令的使用方法。

表 11-4 BL51/Lx51 的输出文件控制命令

BL51	Lx51	缩 写	功能说明
—	ASSIGN	AS	在命令行定义公共符号
IBANKING	—	IB	为 Infineon 公司 TV TEXT 器件 SDA555x 和 SDA30C16x 生成组切换代码
NAME	NAME	NA	指定目标文件模块名
NOAJMP	NOAJMP	NOAJ	不生成带有 AJMP 指令的组切换代码
[NO]INDIRECTCALL	[NO]INDIRECTCALL	[NO]IC	[不]对间接调用函数生成组切换代码
[NO]JMPTAB	—	[NO]JT	[不]生成组切换代码
—	[NO]TYPE	[NO]TY	[不]指定目标文件模块名
[NO]DEBUGLINES	OBJECTCONTROLS	[NO]DL/OC	[不]在目标文件中包含行号信息
[NO]DEBUGPUBLICS	—	[NO]DP	[不]在目标文件中包含公共符号信息
[NO]DEBUGSYMBOLS	—	[NO]DS	[不]在目标文件中包含局部符号信息

命令名: ASSIGN, 仅用于 Lx51

缩 写: AS

参 数: ASSIGN (symname (value) [, ...])

默认值: 无。

μVision2: 在“Options 选项/Lx51 Misc 标签页/Assigb 栏”内键入符号名。

功 能: ASSIGN 命令用于在 Lx51 连接定位器中定义带有数值的公共符号。该符号作为没有指定存储器类型的数字处理, 并与具有相同名称的未确定外部符号相匹配。

例 子: Lx51 SAMPLE1.OBJ ASSIGN(FUNC (0x21000), BITVAR(20H.2))
本例定义了公共符号 FUNC、BITVAR, 并指定 FUNC 的值为 0x2000, 20H.2 作为 BITVAR 的位地址。

命令名: IBANKING, 仅用于 BL51

缩 写: IB

参 数: IBANKING[(bank_sfr_address)]

默认值: 默认的 bank_sfr_address 为 0x94。它也是所支持的 Infineon 公司器件的 SFR 地址。

μVision2: 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”内键入命令。

功 能: IBANKING 命令可使 BL51 连接定位器使用 Infineon 公司器件 SDA30C16x/26x 和 SDA555x TV TEXT 器件的片上代码组硬件。BL51

连接定位器自动将没有?CO?前缀或后缀的代码段放置在该分组区域,将带有?CO 前缀或后缀的代码段放置在公共区域。

采用 IBANKING 命令时不需要 L51_BANK.A51 配置文件, BL51 将生成如下操作指令码:

MOV bank_sfr, #BANK_NUMBER

LJMP target

- 注: 使用该命令需要 keil 公司提供的特殊 Cx51 运行库函数。
- 另 见: NOAJMP, NOINDIRECTCALL, NOJMPTAB, BANKAREA 命令
- 例 子: BL51 BANK0{MODULA.OBJ}, BANK1{MODULB.OBJ} IBANKING
BL51 BANK0{MODULA.OBJ}, BANK1{MODULB.OBJ} IB(80H)

命令名: NAME

缩 写: NA

参 数: NAME (modulename)

默认值: 输入文件表列中第一个模块名。

μVision2: 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”内键入命令。

功 能: NAME 命令用来为输出文件定义模块名, 不输入该命令则以输入文件中的第一个模块名作为默认的输出文件模块名。

注: 由 NAME 命令指定的模块名并不是绝对目标文件名, 该模块名被存储在目标模块文件中, 并且只能由读取该文件内容的程序来访问。

例 子: BL51 SAMPLE.OBJ NAME(BIGPROG)

命令名: NOAJMP

缩 写: NOAJ

参 数: 无

默认值: 连接定位器为代码分组应用生成组间跳转表, 该表用于从不同的代码组或公共区域跳转到某个代码组。根据该表的大小, 连接定位器将使用 AJMP 或 LJMP 指令。

μVision2: 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”内键入命令。

功 能: NOAJMP 命令可以禁止在组间跳转表中使用 AJMP 指令, 这对于不支持 AJMP 指令的 8051 派生器件特别有用。

另 见: IBANKING, NOINDIRECTCALL, NOJMPTAB, BANKAREA

例 子: BL51 SAMPLE.OBJ NOJMP

命令名: [NO]DEBUGLINES, [NO]DEBUGPUBLICS, [NO]DEBUGSYMBOLS,
仅用于 BL51, 对于 Lx51 需要采用 OBJECTCONTROLS 命令

缩 写: [NO]DL, [NO]DP, [NO]DS

参 数: 无

默认值: 在输出文件中包含完整的调试信息。

μVision2: 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”内键入命令。
功 能: 使用这些命令的否定形式可有选择地将输出文件中的某些调试信息去掉, 默认值为命令的肯定形式。
注: 为了能在输出文件中产生调试信息, 输入的目标文件必须是采用编译控制命令 **DEBUG** 编译的。如果去掉调试信息将不能对源程序进行符号化调试。
另 见: **NOLINES**, **NOPUBLICS**, **NOSYMBOLS**, **OBJECTCONTROLS**, **PRINTCONTROLS**
例 子: **BL51 SAMPLE.OBJ NODEBUGSYMBOLS NODEBUGLINES**

命 令 名: **[NO]INDIRECTCALL**
缩 写: **[NO]IC**
参 数: 无
默 认 值: 在代码分组应用中, 连接定位器对每个通过函数指针进行间接调用的函数生成一个组间调用表, 从而可以保证所有代码组都可以通过函数指针来进行函数调用。

μVision2: 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”内键入命令。
功 能: 该命令的否定形式可禁止对间接函数调用生成组间调用表, 这对于应用程序中使用了函数指针并且用户保证从来不发生穿过代码组的间接函数调用是十分有用的。
另 见: **IBANKING**, **NOAJMP**, **NOJMPTAB**, **BANKAREA**
例 子: **BL51 SAMPLE.OBJ NOINDIRECTCALL**

命 令 名: **[NO]JMPTAB**, 仅用于 **BL51**
缩 写: **[NO]JT**
参 数: 无
默 认 值: 在代码分组应用中, 连接定位器自动生成一个组间跳转表, 对于每一个从不同代码组或公共区域中调用的函数, 连接定位器将在组间跳转表中插入一段分组切换代码, 使函数调用重新指向该表。

μVision2: 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”内键入命令。
功 能: 使用 **NOJMPTAB** 命令时 **BL51** 对于函数调用不再插入组间跳转表, 这可以用于采用用户定义代码实现代码组切换。该命令将修改 **BL51** 的如下特性:

- 连接定位器不再需要配置文件 **L51_BANK.OBJ**;
- 连接定位器不修改任何跳转调用指令。

如果发生对另一个代码组的跳转/调用时, 连接定位器不产生任何警告信息, 由于 **BL51** 不再自动选择代码组, 用户必须在发生调用之前保证选择合适的代码组。

另 见: **IBANKING**, **NOAJMP**, **NOJINDIRECTCALL**, **BANKAREA**

例 子: BL51 SAMPLE.OBJ NOJMPTAB

命 令 名: [NO]TYPE, 仅用于 Lx51

缩 写: [NO]TY

参 数: 无

默 认 值: 在输出文件中包含完整的类型信息。

µVision2: 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”内键入命令。

功 能: 该命令的否定形式从输出文件中移去调试符号中的符号类型信息, 符号类型信息仅在对程序进行调试时才需要, 它对执行代码没有影响。

另 见: OBJECTCONTROLS, PURGE

例 子: Lx51 SAMPLE.OBJ NOTYPE

命 令 名: OBJECTCONTROLS

缩 写: OC

参 数: OBJECTCONTROLS (subcontrol[, ...])

默 认 值: 在输出文件中包含完整的调试信息。

µVision2: 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”内键入命令。

功 能: OBJECTCONTROLS 命令允许从输出的目标文件中移去指定的调试信息。可以使用如下子控制命令参数:

NOCOMMENTS 注释记录

NOLINES 行号信息

NOPUBLICS 公共符号信息

NOSYMBOLS 局部符号信息

PURGE 全部调试信息

另 见: NOCOMMENTS, NOLINES, NOPUBLICS, NOSYMBOLS, PURGE 命令

例 子: Lx51 SAMPLE.OBJ OBJECTCONTROLS(NOCOMMENTS)

11.2.3 段与存储器定位控制命令

BL51/Lx51 连接定位器允许指定不同存储器区域或存储器类的大小、不同存储器区域内各个段的顺序, 以及不同段的绝对存储器地址。连接定位器根据存储器类及预定义的优先顺序进行段定位。标准配置算法一般不需要人工从命令行输入任何段名就可以产生最好的结果。表 11-5 所示为 BL51/Lx51 的段与存储器定位控制命令, 利用这些命令可以按照应用目标系统的物理存储器结构更精确地在不同的存储器类中进行段定位。下面介绍各条命令的使用方法。

表 11-5 BL51/Lx51 的段与存储器定位控制命令

BL51	Lx51	缩 写	功能说明
BANKAREA	BANKAREA	BA	指定存储器分组的地址范围
BANKx(x=0~31)	—	Bx	在代码组中进行段定位
BIT	—	BI	对 BIT 段对进行定位
—	CLASSES	CL	在存储器类中指定段的物理地址范围
CODE	—	CO	对 CODE 段对进行定位
DATA	—	DA	对 DATA 段对进行定位
IDATA	—	ID	对 IDATA 段对进行定位
NOSORTSIZE	NOSORTSIZE	NOSO	安排存储器之前禁止对段进行大小排序
PDATA	—	PD	指定 PDATA 的起始地址
PRECEDE	—	PC	在 DATA 空间对优先于其他部分的段进行定位
RAMSIZE	—	RS	指定 DATA 和 IDATA 存储器空间大小
—	RESERVE	RE	保留一段存储器范围不让连接器使用
—	SEGMENTS	SE	对指定段定义物理存储器地址
—	SEGSIZE	SS	修改指定段的大小
STACK	—	ST	对 STACK 段对进行定位
XDATA	—	XD	对 XDATA 段进行定位

命令名: BANKAREA

缩 写: BA

参 数: BANKAREA (start_address, end_address)

默认值: 无。

μVision2: 在“Options 选项/Target 标签页/”选中“Code Banking”复选框,并在“Banks”框内选取分组数,在“Bank Area/Start/End”框内分别键入起始、终止地址值。

功 能: BANKAREA 命令用来为每个代码组分配所指定的存储器地址。除非使用 BANKn 命令作精细定位,分配给各个代码组的所有代码段都将定位在这个地址范围之内。

另 见: BANKx 命令

例 子: Lx51 COMMON{C_ROOT.OBJ}, BANK0{C_BANK0.OBJ},
BANK0{C_BANK0.OBJ}, BANK1{C_BANK1.OBJ},
BANK2{C_BANK2.OBJ} TO MYPROG.ABS &
BANKAREA(8000H,0FFFFH)

命令名: BANKx, 仅用于 BL51, 对于 Lx51 应使用 SEGMENTS 命令

缩 写: B0,B1,...,B30,B31

参 数: BANKx ([start_address][segname [(address)] [, ...]])

- 默认值: 无。
- μVision2: 在“Options 选项/BL51 Misc 标签页/Misc controls 栏”内键入命令。
- 功 能: **BANKx** 命令用于对各个段指定代码组, 段被定位在指定的代码组以 **start_address** 开始的地址处, 若没给出 **start_address** 则定位在 **0x0000** 地址处。如果在代码组中安排常数段, 则必须保证使用合适的代码组以保证能够访问该常数段, 这可以通过配置文件 **L51_BANK.A51** 中的组切换函数 (**switchbank**) 来实现。
- 另 见: **BANKAREA, CODE**
- 例 子: 下面例子将模块 **A.OBJ** 中的函数 **FUNC1** 定位在 **BANK1** 中从 **0x8000** 开始的地址处, 而模块 **B.OBJ** 中函数 **FUNC2** 将被定位在从 **0x8200** 开始的地址处。
- ```
BL51 COMMON{A.OBJ}, BANK0{B.OBJ}
 BANK1(0x8000, ?PR?FUNC1?A, ?PR?FUNC2?B(0x8200))
```
- 命令名: **BIT, DATA, IDATA**, 仅用于 **BL51**, 对于 **Lx51** 应使用 **CLASSES** 和 **SEGMENTS** 命令
- 缩 写: **BI, DA, ID**
- 参 数: **BIT** ([start\_address] [segname [(address)] [, ...]])  
**DATA** ([start\_address] [segname [(address)] [, ...]])  
**IDATA** ([start\_address] [segname [(address)] [, ...]])
- 默认值: 无。
- μVision2: 在“Options 选项/BL51 Locate 标签页/”的 **Bit, DATA, IDATA** 对应的“Base 栏”内键入起始地址, “Segments 栏”内键入“段名[地址]”, 若有多个段则应用逗号隔开。
- 功 能: 这几条命令允许在 **8051** 片内 **RAM** 空间进行段定位, 指定段起始地址, 调整段的定位顺序。用 **BIT** 命令指定的地址必须是位地址, 位地址范围为 **00H~7FH**, 相应字节地址范围为 **20H~2FH**。用 **DATA** 命令指定片内直接寻址空间, 地址范围为 **00H~7FH**, 用 **IDATA** 命令指定片内间接寻址空间, 地址范围为 **00H~FFH**。另外这些定位参数也影响段的定位算法。通常 **BL51** 默认的定位算法能够产生满意的结果, 但有时用户需要更仔细地对地址空间进行定义, 尤其是对于 **8051** 的内部 **RAM** 来说这些细微调整有时是很有必要的。
- 例 子: **BL51 SAMPLE.OBJ BIT(20H.2)**  
**BL51 SAMPLE.OBJ, A.OBJ,B.OBJ DATA(?DT?A(28H),?DT?B(30H))**  
**BL51 SAMPLE.OBJ, A.OBJ,B.OBJ IDATA(?ID?A(30H),?ID?B(40H))**
- 命令名: **CLASSES**, 仅用于 **Lx51**
- 缩 写: **CL**
- 参 数: **CLASSES**(classname(range[,...])[,...])

默认值: 无

μVision2: 在“Options 选项/Lx51 Locate 标签页/User Classes 栏”键入存储器类名。

功 能: CLASSES 命令用于为一个存储器类中的段指定物理地址范围, 该命令提供一种有效的定义物理存储器结构的方法。如果没有采用该命令指定存储器类的物理地址界限, 连接定位器则将使用该存储器类的物理地址界限, 所采用的地址范围显示在列表文件 (.MAP 文件) 的 ACTIVE MEMORY CLASSES 部分, 建议用户经常察看 MAP 文件的这一部分, 因为其中列出了 Lx51 位用户硬件目标板假设的存储器。

采用 CLASSES 命令可以修改带有 OFFS 类型的再定位段的绝对地址, 在地址范围区用带“\$”前缀的第一个地址指定偏移量, 例如: CLASSES (CODE (\$0xFF8000, 0xFF8000~0xFFFFF), 此时所有用 OFFS 再定位类型定义的段都被重新定向到地址 0xFF8000。典型地, 程序中断和复位向量可以采用这种方法定义。

另 见: SEGMENTS

例 子: 下面的例子为传统 8051 单片机定义了存储器类:

Lx51 MYFILE.OBJ

```
CLASSES (IDATA (I:0-I:0xFF), XDATA (X:0-X:0xEFFF),
 CODE (C:0-C:0x7FFF, C:0xC000-C:0xFFFF))
```

下面的例子定义了用户定义存储器类 XDATA\_FLASH:

Lx51 MYFILE.OBJ

```
CLASSES (XDATA_FLASH (X:0x8000-X:0xEFFF))
```

命令名: CODE, 仅用于 BL51, 对于 Lx51 应使用 CLASSES 和 SEGMENTS 命令  
缩 写: CO

参 数: CODE ([address\_range] [segname [(address)] [, ...]])

默认值: 无

μVision2: 在“Options 选项/BL51 Locate 标签页/”的“Code Range 栏”键入代码地址范围, 在 CODE 对应的“Segments 栏”内键入“段名[地址]”, 若有多个段则应用逗号隔开。

功 能: CODE 命令用于指定 CODE 存储器空间的段地址范围、段顺序以及段的绝对存储器位置。

另 见: BIT, DATA, IDATA, XDATA

例 子: 下面的例子将可再定位的 CODE 段定位在地址空间 0x0000~0x3FFF 和 0x8000~0xFFFF:

```
BL51 MYPROG.OBJ CODE(0-0x3FFF, 0x8000-0xFFFF)
```

下面的例子将目标模块 A.OBJ 中的段?PR?FUNC1?A 和段?PR?FUNC2?A 定位在 CODE 存储器空间的开始处:

```
BL51 A.OBJ CODE(?PR?FUNC1?A, ?PR?FUNC2?A)
```

下面的例子将目标模块 A.OBJ 中的段?PR?FUNC1?A 定位在以 0x800 为

起始地址处，然后在该段的后面定位段?PR?FUNC2?A:  
BL51 A.OBJ CODE(?PR?FUNC1?A (0x800), ?PR?FUNC2?A)

命令名: NOSORTSIZE  
缩 写: NOSO  
参 数: 无  
默认值: 在连接定位器分配存储器空间之前，各个段按照其大小排序，从而可以减少为满足分配要求的存储器间隙。  
μVision2: 在“Options 选项/BL51 Misc 标签页/Misc controls 栏”内键入命令。  
功 能: NOSORTSIZE 命令用于禁止排序算法。此时连接定位器按各个段在输入文件中出现的顺序来分配存储器。  
例 子: BL51 MYPROG.OBJ NOSORTSIZE

命令名: PDATA，仅用于 BL51，对于 Lx51 应使用 CLASSES 和 SEGMENTS 命令  
缩 写: 无  
参 数: PDATA (address)  
默认值: 无。  
μVision2: 在“Options 选项/BL51 Locate 标签页”内 Pdata 对应的“Base 框”键入地址。  
功 能: PDATA 定位命令对应于采用 COMPACT 编译模式的目标程序，PDATA 命令为具有 PDATA 类型的所有段指定首地址（段属性为 XDATA，INPAGE）。此时 8051 单片机通过端口 P2 口采用间接寻址方式（MOVX @R0）寻址。采用这种寻址方式时，用户必须将端口 P2 的内容设置成“页地址”（PAGE）。例如，在 PDATA 存储器区域始于 5080H 的地址空间为某个对象所用，则必须将页地址值 50H 写入到端口 P2。这一初始化过程可通过启动程序 STARTUP.A51 来完成，在启动程序源文件 STARTUP.A51 中包含了一个专用于页寻址的 EQU 语句，需要时可对该 EQU 语句进行适当修改，具体修改方法请参见 9.6.1 “起动代码文件”一节。这种寻址方式的优点在于：PDATA 类型对象的地址只占用一个字节，而 XDATA 类型对象的地址需占用两个字节；对于数组、指针或结构的地址计算可用 8 位数进行，这一点在 8051 中是十分有效的；在 PDATA 地址空间存取一个字节要比在 XDATA 地址空间快 25%~30%。需要注意的是，对于某些不能采用 P2 端口寻址 XDATA 地址空间的 8051 芯片，则不能采用这种寻址方式。  
例 子: BL51 SAMPLE.OBJ PDATA(0x9000)

命令名: PRECEDE，仅用于 BL51，对于 Lx51 应使用 CLASSES 和 SEGMENTS 命令。

**缩写:** PC

**参数:** PRECEDE (segname [(address)][, ...])

**默认值:** 无。

**μVision2:** 在“Options 选项/BL51 Locate 标签页”内 Precede 对应的“Segments 框”键入“段名[地址]”，如果有多个段，各段之间要用逗号隔开。

**功能:** PRECEDE 命令用于在 8051 单片机内部 RAM 中指定需要位于其他段之前的段，在位寻址区和工作寄存器区中进行段定位。对于该命令所指定的段，连接定位器将在完成寄存器组以及所有绝对 BIT, DATA, IDATA 段的定位之后，在定位其他段之前进行定位。例如，BL51 在执行常规定位算法时，先在内部 RAM 的位寻址区定位位段，然后再定位其他数据段。如果程序中的位变量没有占满整个位寻址区，则位寻址区将有一段空余地址 (GAP)，这对于十分有限的 8051 内部数据存储器是一种浪费。使用 PRECEDE 命令则可以避免这一点。

**例子:** BL51 MYPROG.OBJ,A.OBJ,B.OBJ PRECEDE(?DT?A,?DT?B)

用该命令指定的段按指定顺序从 DATA 存储器空间的最低地址开始定位，也可以指定段的存储器地址。下面的例子将目标模块 A.OBJ 中的段?DT?A 定位在 DATA 存储器从地址 09H 开始的位置，将目标模块 B.OBJ 中的段?DT?B 定位在 DATA 存储器从地址 13H 开始的位置：

BL51 A.OBJ,B.OBJ PRECEDE(?DT?A(09h),?DT?B(13h))

**命令名:** RAMSIZE，仅用于 BL51，对于 Lx51 应使用 CLASSES 和 SEGMENTS 命令

**缩写:** RS

**参数:** RAMSIZE (value)

**默认值:** RAMSIZE (128)

**μVision2:** 由项目中选定的器件库信息产生。

**功能:** 该命令以字节为单位定义 8051 单片机内部数据 RAM 的大小，允许范围为 64~256，默认值为 128。通常单片机内部数据 RAM 指的是 DATA 和 IDATA 空间，但有些新型 80C51 单片机具有映射到 XDATA 空间的片内 RAM，在这种情况下可以采用 RAMSIZE(256)来允许整个 DATA 和 IDATA 空间，并且采用 XDATA 命令来定义附加的片内 RAM。

**例子:** BL51 SAMPLE.OBJ RAMSIZE(256)

**命令名:** RESERVE，仅用于 Lx51

**缩写:** RE

**参数:** RESERVE (range[,...])

**默认值:** 不保留存储区。

**μVision2:** 在“Options 选项/Lx51 Misc 标签页/Misc controls 栏”键入命令。

**功能:** 该命令用于阻止连接定位器在指定的物理存储器地址范围内进行段定

位, 此时连接定位器将不会使用指定范围内的物理存储器地址。如果绝对段使用了保留的存储区将产生一个警告信息。

另 见: CLASSES, SEGMENTS

例 子: Lx51 MYPROG.OBJ RESERVE(0x200-0x3FFF, 0xFF8000H-0xFFBFFFH)

命令名: SEGMENTS, 仅用于 Lx51

缩 写: SE

参 数: SEGMENTS (segments[address][,...])

默认值: 无。

µVision2: 在“Options 选项/Lx51 Locate 标签页/User Segments 栏”键入命令。

功 能: 该命令允许控制以下内容:

- 段的绝对存储器位置, 绝对地址可以是起始地址或结束地址;
- 存储器内段的顺序。指定的段按顺序分配, 可以作为第一个段或最后一个段定位。通常第一个段定位在可能的最低地址范围(由 CLASSES 命令指定), 接下来的段按递增的地址定位, 如果使用了“LAST”作为地址, 则该段将被作为最后一个段来定位。

另 见: CLASSES, RESERVE, SEGSIZE

例 子: 下面的例子将目标模块 A.OBJ 和 B.OBJ 中的段?DT?A 和段?DT?B 定位在 DATA 空间的 28H 和 30H:

Lx51 A.OBJ,B.OBJ SEGMENTS (?DT?A(D:0x28),?DT?B(D:0x30))

下面的例子将目标模块 B.OBJ 中的段?DT?B 作为 DATA 存储器类中的最后一个段处理:

Lx51 A.OBJ,B.OBJ,C.OBJ SEGMENTS(?DT?B(LAST))

命令名: SEGSIZE, 仅用于 Lx51

缩 写: 无

参 数: SEGMENTS (segments(size))[,...])

默认值: 无。

µVision2: 在“Options 选项/Lx51 Locate 标签页/Misc Controls 栏”键入命令。

功 能: 该命令允许指定段所使用存储器空间的大小, BIT 段的长度(size)需要采用“.”运算符按位指定。可以采用如下方法指定段长度:

- “+”表示在当前段长度上增加给定值;
- “-”表示在当前段长度上减少给定值;
- 无+、-号表示以指定值作为当前段新的长度。

另 见: SEGMENTS

例 子: Lx51 MYPROG.OBJ SEGSIZE(?STACK (+20H))

命令名: STACK, 仅用于 BL51, 对于 Lx51 应使用 CLASSES 和 SEGMENTS 命令

缩 写: ST



- 参 数:** STACK (segments[(address)][,...])
- 默 认 值:** 无。
- $\mu$ Vision2:** 在“Options 选项/BL51 Locate 标签页/”中 Stack 所对应的“Segments”框内键入“段名[地址]”，如果有多个段，各段之间要用逗号隔开。
- 功 能:** STACK 命令允许在 8051 的内部 RAM (IDATA) 的上部空间 (uppermost) 进行段定位，采用该命令指定的段将位于内部 RAM 中所有其他段的后面，通常用做堆栈。Cx51 编译器会产生一个 ?STACK 的堆栈段，该段将被自动定位到 IDATA 空间的顶部。启动代码将对堆栈指针进行初始化。
- 一般不需要特别指定 ?STACK 段的位置，对于具有几个堆栈的汇编程序才需要采用 STACK 命令。需要注意的是，重新定位 ?STACK 段必须非常小心，因为可能会破坏 DATA 或 IDATA 空间的变量而导致程序无法正常运行。
- 另 见:** DATA, IDATA, PRECEDE
- 例 子:** BL51 SAMPLE.OBJ, A.OBJ,B.OBJ STACK(?DT?A,?DT?B)  
BL51 SAMPLE.OBJ, A.OBJ,B.OBJ STACK(?DT?A(69H),?DT?B(73H))
- 命 令 名:** XDATA
- 缩 写:** XD
- 参 数:** XDATA([address\_rang][segname[(address)][,...]])
- 默 认 值:** 无。
- $\mu$ Vision2:** 在“Options 选项/BL51 Locate 标签页/”的“Xdata Range 栏”键入外部 RAM 地址范围，在 Xdata 对应的“Segments 栏”内键入“段名[地址]”，若有多个段则应用逗号隔开。
- 功 能:** XDATA 命令允许在 8051 的外部 RAM (XDATA) 空间进行段定位。
- 例 子:** BL51 SAMPLE.OBJ XDATA(1000H))  
BL51 SAMPLE.OBJ, A.OBJ,B.OBJ XDATA(?XD?A,?XD?B)  
BL51 SAMPLE.OBJ, A.OBJ,B.OBJ XDATA(?XD?A(100H),?XD?B(200H))

从命令行引用 BL51/Lx51 段与存储器定位控制命令的一般格式如下：

定位命令(起始地址 [...]| 段名[(基地址)][,...])

其中，“起始地址”是该定位命令所对应段空间的首地址，另外该定位命令所对应的段必须是目标文件中已经存在的段。

“段名”是目标文件中已存在的数据段或代码段的段名。

“基地址”是该数据段或代码段的起始地址。

采用定位参数来定位的段按顺序分配地址空间，第一个段总是定位在可能的最低地址（默认值为 0x0000），后续的段则定位在相继的地址上。用户可用定位参数定义一个或所有段的基地址（=起始地址），如果定义的段地址与其他已分配的段发生交迭，则会产生相应的提示信息。

采用 BIT 命令定义的基地址总是位地址，允许的位地址为 00H~7FH（或 20H.x~

2FH.x)。

STACK 段不管其大小总是定位在其他内部数据段的后面, 而由 Cx51 和 PL/M51 编译器生成的?STACK 段则总是定位在其他明确声明了的 STACK 段的后面。这种方式可保证 8051 能够使用内部 RAM 的最大自由空间。通常对于 C51 程序没有必要专门指定 STACK 段, BL51 的 STACK 命令一般多用于具有多个 STACK 段的汇编语言程序的连接定位。使用 STACK 命令进行段定位时必须特别小心, 以避免发生堆栈溢出而导致用户程序的崩溃。

下面是几个使用连接定位命令的例子。例子均用命令行形式给出以利于读者理解, 实际使用时, 只要在 Windows 集成开发环境  $\mu$ Vision2 中, 通过“Project 菜单/Options 选项/BL51 Locate (或 Lx51 Locate) 标签页/BL51 Misc (或 Lx51 Misc) 标签页”, 可以很方便地对各种定位控制命令进行设置。

**例 11.19:** BL51 CSAMPLE1.OBJ,CSAMPLE2.OBJ,STARTUP.OBJ PDATA(8000H)

本例中, PDATA 对象的地址从 8000H 开始, 在使用 BL51 定位控制命令 PDATA 的同时, 必须将 STARTUP.A51 文件中变量“PPAGEENABLE”和“PPAGE”的值分别修改成“1”和“80H”, 对修改后的文件重新进行汇编, 并将汇编得到的目标文件作为 BL51 的一个输入文件。

**例 11.20:** BL51 CMODELE.OBJ XDATA(8000H) CODE(4000H)

本例中外部数据区 (XDATA 段) 的地址从 8000H 开始, 程序代码区 (CODE 段) 的地址从 4000H 开始。

**例 11.21:** BL51 CMOD1.OBJ,CMOD2.OBJ,CMOD3.OBJ BIT(40H) DATA(30H)

          IDATA(30H) XDATA(1000H,?XD?CMOD3(0E000H),?XD?FUNCTION?  
          CMOD3)

本例中, 从位地址 40H(=28H.0)开始的所有再定位 BIT 段和所有从地址 30H 开始的 DATA 与 IDATA 段都定位在 8051 的内部 RAM 中。所有未包括在 XDATA 参数中的再定位 XDATA 段都定位在从地址 1000H 开始的外部 RAM 空间内。全局对象?XD?CMOD3 段中的函数局部变量?XD?FUCTION?CMOD3 段定位在从地址 0E000H 开始的外部 RAM 空间。

**例 11.22:** BL51 CMOD1.OBJ,CMOD2.OBJ PRECEDE(VAR1) XDATA(ARRAY1(100H),  
          ARRAY2)

本例中, 数据段 VAR1 定位在 8051 内部 RAM 的工作寄存器区或位寻址区内, XDATA 段的 ARRAY1 定位在从地址 100H 开始的外部 RAM 区, XDATA 段的 ARRAY2 定位于 ARRAY1 之后。

**例 11.23:** BL51 MYPROG.OBJ,SAMPLE.OBJ, UTIL.LIB CODE((CODESEG1(4000H),  
          CODESEG2(8000H)) STACK(STACK\_AREA)

本例中, 程序代码段 CODESEG1 定位在从地址 4000H 开始的 ROM 区, 程序代码段 CODESEG2 定位在从地址 8000H 开始的 ROM 区, STACK\_AREA 段定位在 8051 单片机内部 RAM 的顶部。

下面再举几个分组连接定位的例子。

**例 11.24:** 本例中有 4 个 C 语言程序模块: C\_ROOT.C、C\_BANK0.C、C\_BANK1.C

和 C\_BANK2.C，它们的程序代码分别放在不同的存储器组中。整个程序从运行库的“STARTUP”模块执行，然后跳转到主程序“main()”中，在每个存储器代码组中都有函数调用，这些函数重复地调用其他代码组中的函数。调用 BL51 的控制命令为：

```
BL51 CO{C_ROOT.OBJ},B0{C_BANK0.OBJ},B1{C_BANK1.OBJ},B2{C_BANK2.OBJ}
 BA(8000H,0FFFFH) TO BANK_EX1
```

模块 C\_ROOT.C 定位在公共区域，模块 C\_BANK0.C~C\_BANK2.C 分别定位在代码组 BANK0~BANK2 中，命令 BA(8000H,0FFFFH)的作用是将代码组的地址空间定义为 8000H~0FFFFH。

模块 C\_ROOT.C 中包含有组间跳转表的存储器分配信息。BL51 产生的输出文件为 BANK\_EX1，它是具有“分组的”目标文件。利用分组目标文件转换程序 OC51 可将这种“分组的”目标文件转换成标准的绝对目标文件，标准绝对目标文件可以装入仿真器中进行调试，也可以利用转换程序 OH51 转换为 Intel HEX 文件，以便于 EPROM 编程。关于 OC51 和 OH51 转换程序的具体操作将在 11.3 节介绍。

C\_ROOT.C 源程序代码如下：

```
#include <stdio.h>
#include <reg51.h>
extern void func0(void);
extern void func1(void);

void main(void) {
 SCON = 0x52; /* 串行口初始化为：2400 BAUD @12MHz */
 TMOD = 0x20;
 TCON = 0x69;
 TH1 = 0xf3;
 printf("MAIN PROGRAM CALLS A FUNCTION IN BANK 0 \n");
 func0();
 printf("MAIN PROGRAM CALLS A FUNCTION IN BANK 1 \n");
 func1();
 while(1);
}
```

C\_BANK0.C 源程序代码如下：

```
#include <stdio.h>
extern void func2(void);

void func0(void) {
 printf("FUNCTION IN BANK 0 CALLS A FUNCTION IN BANK 2 \n");
 func2();
}
```

C\_BANK1.C 源程序代码如下：

```
#include <stdio.h>
extern void func2(void);

void func1(void) {
 printf("FUNCTION IN BANK 1 CALLS A FUNCTION IN BANK 2 \n");
 func2();
}
```

C\_BANK2.C 源程序代码如下:

```
#include <stdio.h>

void func2(void) {
 printf("THIS IS A FUNCTION IN BANK 2! \n");
}
```

BL51 所产生的部分列表文件如下:

MEMORY MODEL: SMALL

INPUT MODULES INCLUDED:

```
C_root.obj (C_ROOT)
C_bank1.obj (C_BANK1)
C_bank2.obj (C_BANK2)
C_bank0.obj (C_BANK0)
L51_bank.obj (?BANK?SWITCHING)
F:\KEIL\C51\LIB\C51S.LIB (?C_STARTUP)
F:\KEIL\C51\LIB\C51S.LIB (PRINTF)
F:\KEIL\C51\LIB\C51S.LIB (?C_INIT)
F:\KEIL\C51\LIB\C51S.LIB (?C?CLDPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?CLDOPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?CSTPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?PLDIIDATA)
F:\KEIL\C51\LIB\C51S.LIB (?C?CCASE)
F:\KEIL\C51\LIB\C51S.LIB (PUTCHAR)
```

LINK MAP OF MODULE: Bank\_ex1 (C\_ROOT)

| TYPE      | BASE  | LENGTH  | RELOCATION  | SEGMENT NAME |
|-----------|-------|---------|-------------|--------------|
| -----     |       |         |             |              |
| * * * * * |       | D A T A | M E M O R Y | * * * * *    |
| REG       | 0000H | 0008H   | ABSOLUTE    | "REG BANK 0" |
| DATA      | 0008H | 0014H   | UNIT        | _DATA_GROUP_ |
| DATA      | 001CH | 0001H   | UNIT        | ?BANK?DATA   |
|           | 001DH | 0003H   |             | *** GAP ***  |

```

BIT 0020H.0 0001H.1 UNIT _BIT_GROUP_
 0021H.1 0000H.7 *** GAP ***
IDATA 0022H 0001H UNIT ?STACK

* * * * * C O D E M E M O R Y * * * * *
CODE 0000H 0003H ABSOLUTE
CODE 0003H 035CH UNIT ?PR?PRINTF?PRINTF
CODE 035FH 008EH UNIT ?C?LIB_CODE
CODE 03EDH 008CH UNIT ?C_C51STARTUP
CODE 0479H 005EH INBLOCK ?BANK?SELECT
CODE 04D7H 0054H UNIT ?CO?C_ROOT
CODE 052BH 0030H UNIT ?CO?C_BANK1
CODE 055BH 0030H UNIT ?CO?C_BANK0
CODE 058BH 0027H UNIT ?PR?PUTCHAR?PUTCHAR
CODE 05B2H 0026H UNIT ?PR?MAIN?C_ROOT
CODE 05D8H 0020H UNIT ?CO?C_BANK2
CODE 05F8H 0008H UNIT ?C_INITSEG
CODE 0600H 003BH PAGE ?BANK?SWITCH

* * * * * C O D E B A N K 0 * * * * *
 0000H 8000H *** GAP ***
* BANK0 8000H 000CH UNIT ?PR?FUNC0?C_BANK0

* * * * * C O D E B A N K 1 * * * * *
 0000H 8000H *** GAP ***
BANK1 8000H 000CH UNIT ?PR?FUNC1?C_BANK1

* * * * * C O D E B A N K 2 * * * * *
 0000H 8000H *** GAP ***
BANK2 8000H 0009H UNIT ?PR?FUNC2?C_BANK2

```

OVERLAY MAP OF MODULE: Bank\_ex1 (C\_ROOT)

| SEGMENT                | BIT_GROUP |        | DATA_GROUP |        |
|------------------------|-----------|--------|------------|--------|
| ---> CALLED SEGMENT    | START     | LENGTH | START      | LENGTH |
| ?C_C51STARTUP          | -----     | -----  | -----      | -----  |
| ---> ?PR?MAIN?C_ROOT   |           |        |            |        |
| ---> ?C_INITSEG        |           |        |            |        |
| ?PR?MAIN?C_ROOT        | -----     | -----  | -----      | -----  |
| ---> ?CO?C_ROOT        |           |        |            |        |
| ---> ?PR?PRINTF?PRINTF |           |        |            |        |

```

+--> ?PR?FUNC0?C_BANK0
+--> ?PR?FUNC1?C_BANK1

?PR?PRINTF?PRINTF 0020H.0 0001H.1 0008H 0014H
+--> ?PR?PUTCHAR?PUTCHAR

?PR?FUNC0?C_BANK0 -----
+--> ?CO?C_BANK0
+--> ?PR?PRINTF?PRINTF
+--> ?PR?FUNC2?C_BANK2

?PR?FUNC2?C_BANK2 -----
+--> ?CO?C_BANK2
+--> ?PR?PRINTF?PRINTF

?PR?FUNC1?C_BANK1 -----
+--> ?CO?C_BANK1
+--> ?PR?PRINTF?PRINTF
+--> ?PR?FUNC2?C_BANK2

```

INTRABANK CALL TABLE OF MODULE: Bank\_ex1 (C\_ROOT)

| ADDRESS | FUNCTION NAME |
|---------|---------------|
| 04C7H   | FUNC0         |
| 04CCH   | FUNC1         |
| 04D1H   | FUNC2         |

**例 11.25:** 本例说明了文本常数的“分组”。利用 BL51 可以将具有文本常数的代码段定位在代码组之内。Cx51 编译器规定, 将具有常数的代码段命名为?CO?MODULE, 这些段可以用 BL51 的 BANKx 命令来控制定位。但用户必须保证在进行常数的存取时, 能选择正确的代码组, 所以在本例的程序模块 C\_PROG.C 中使用了 L51\_BANK.A51 中的“组切换”功能, 即调用了 L51\_BANK.A51 模块中的“switchbank”函数。整个程序由模块: C\_PROG.C、C\_MESS0.C 和 C\_MESS1.C 组成。程序模块的连接命令如下:

```

BL51 C_PROG.OBJ,C_MESS0.OBJ,C_MESS1.OBJ B0(?CO?C_MESS0(8000H),
 B1(?CO?C_MESS1(8000H)) BA(8000H,0FFFFH)

```

C\_PROG.C 源程序代码如下:

```

#include <stdio.h>
#include <reg51.h>
extern code char *message0[];
extern code char *message1[];
extern switchbank (unsigned char);

```

```
void main(void) {
 SCON = 0x52; /* 串行口初始化为: 2400 BAUD @12MHz */
 TMOD = 0x20;
 TCON = 0x69;
 TH1 = 0xf3;
 switchbank(0); /* 切换到代码组 0 */
 printf(message0[0]);
 switchbank(1); /* 切换到代码组 1 */
 printf(message1[0]);
 while(1);
}
```

C\_MESS0.C 源程序代码如下:

```
code char *message0[] = {
 "This is a message from code-bank 0\n",
 "This is another text"
};
```

C\_MESS1.C 源程序代码如下:

```
code char *message1[] = {
 "This is a message from code-bank 1\n",
 "This is another text"
};
```

BL51 所产生的部分列表文件如下:

MEMORY MODEL: SMALL

INPUT MODULES INCLUDED:

```
L51_BANK.obj (?BANK?SWITCHING)
C_PROG.obj (C_PROG)
C_MESS0.obj (C_MESS0)
C_MESS1.obj (C_MESS1)
F:\KEIL\C51\LIB\C51S.LIB (?C_STARTUP)
F:\KEIL\C51\LIB\C51S.LIB (PRINTF)
F:\KEIL\C51\LIB\C51S.LIB (?C?CLDPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?CLDOPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?CSTPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?PLDIIDATA)
F:\KEIL\C51\LIB\C51S.LIB (?C?CCASE)
F:\KEIL\C51\LIB\C51S.LIB (PUTCHAR)
```

LINK MAP OF MODULE: bank\_ex2 (?BANK?SWITCHING)

| TYPE                            | BASE    | LENGTH  | RELOCATION | SEGMENT NAME        |
|---------------------------------|---------|---------|------------|---------------------|
| -----                           |         |         |            |                     |
| * * * * * DATA MEMORY * * * * * |         |         |            |                     |
| REG                             | 0000H   | 0008H   | ABSOLUTE   | "REG BANK 0"        |
| DATA                            | 0008H   | 0014H   | UNIT       | _DATA_GROUP_        |
|                                 | 001CH   | 0004H   |            | *** GAP ***         |
| BIT                             | 0020H.0 | 0001H.1 | UNIT       | _BIT_GROUP_         |
|                                 | 0021H.1 | 0000H.7 |            | *** GAP ***         |
| IDATA                           | 0022H   | 0001H   | UNIT       | ?STACK              |
| * * * * * CODE MEMORY * * * * * |         |         |            |                     |
| CODE                            | 0000H   | 0003H   | ABSOLUTE   |                     |
| CODE                            | 0003H   | 035CH   | UNIT       | ?PR?PRINTF?PRINTF   |
| CODE                            | 035FH   | 008EH   | UNIT       | ?C?LIB_CODE         |
| CODE                            | 03EDH   | 003BH   | UNIT       | ?PR?MAIN?C_PROG     |
| CODE                            | 0428H   | 0034H   | INBLOCK    | ?BANK?SELECT        |
| CODE                            | 045CH   | 0027H   | UNIT       | ?PR?PUTCHAR?PUTCHAR |
| CODE                            | 0483H   | 000CH   | UNIT       | ?C_C51STARTUP       |
|                                 | 048FH   | 0071H   |            | *** GAP ***         |
| CODE                            | 0500H   | 000CH   | PAGE       | ?BANK?SWITCH        |
| * * * * * CODE BANK 0 * * * * * |         |         |            |                     |
|                                 | 0000H   | 8000H   |            | *** GAP ***         |
| BANK0                           | 8000H   | 0054H   | UNIT       | ?CO?C_MESS0         |
| * * * * * CODE BANK 1 * * * * * |         |         |            |                     |
|                                 | 0000H   | 8000H   |            | *** GAP ***         |
| BANK1                           | 8000H   | 0054H   | UNIT       | ?CO?C_MESS1         |

OVERLAY MAP OF MODULE: bank\_ex2 (?BANK?SWITCHING)

| SEGMENT              | BIT_GROUP |        | DATA_GROUP |        |
|----------------------|-----------|--------|------------|--------|
| +++> CALLED SEGMENT  | START     | LENGTH | START      | LENGTH |
| -----                |           |        |            |        |
| ?C_C51STARTUP        | -----     | -----  | -----      | -----  |
| +++> ?PR?MAIN?C_PROG |           |        |            |        |
| ?PR?MAIN?C_PROG      | -----     | -----  | -----      | -----  |
| +++> ?CO?C_MESS0     |           |        |            |        |



```

+--> ?PR?PRINTF?PRINTF
+--> ?CO?C_MESS1

?PR?PRINTF?PRINTF 0020H.0 0001H.1 0008H 0014H
+--> ?PR?PUTCHAR?PUTCHAR

```

**例 11.26:** 代码组中的特殊函数。Cx51 程序中的特殊函数可以重新定位在代码组中, 甚至代码段也可以利用 BL51 来控制定位在其他代码组中。本例将程序中的中断函数定位在公共区域, 中断函数的段名(?PR?TIMER0?C\_MODUL)在 BL51 的控制参数“COMMON”中指定, 函数“tinit”必须重新定位在代码组 BANK1 中, 这个函数的段名由 BANK1 的参数表来指定。连接定位命令如下:

```

BL51 BANK0{C_MODUL.OBJ} BANKAREA(8000H,0FFFFH) COMMON(?PR?TIMER0?C_MODUL)
 BANK1(?PR?TINIT?C_MODUL(8000H))

```

C\_MODUL.C 源文件代码如下:

```

#include <stdio.h>
#include <reg51.h>
unsigned long msec; /* 毫秒计数器变量 */
unsigned char intcycle; /* 中断次数计数器变量 */

/*****
/* T0 中断服务函数 */
/* 采用12MHz晶振时每250微秒执行一次 */
*****/
timer0() interrupt 1 using 1 {
 if (++intcycle == 4) { /* 1 msec = 4* 250 usec */
 intcycle = 0;
 msec++;
 }
}

/*****
/* T0 初始化函数 */
*****/
tinit () {
 TH0 = -250;
 TL0 = -250;
 TMOD = TMOD | 0x02;
 TR0 = 1;
 ET0 = 1;
 EA = 1;
}

```

```

void main(void) {
 SCON = 0x52; /* 串行口初始化为: 2400 BAUD @12MHz */
 TMOD = 0x20;
 TCON = 0x69;
 TH1 = 0xf3;
 tinit (); /* 调用 T0 初始化函数 */
 while(1) {
 printf ("MSEC=%lu\r", msec);
 }
}

```

BL51 所产生的部分列表文件如下:

MEMORY MODEL: SMALL

INPUT MODULES INCLUDED:

```

.\STARTUP.obj (?C_STARTUP)
L51_BANK.obj (?BANK?SWITCHING)
C_MODUL.obj (C_MODUL)
F:\KEIL\C51\LIB\C51S.LIB (PRINTF)
F:\KEIL\C51\LIB\C51S.LIB (?C?CLDPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?CLDOPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?CSTPTR)
F:\KEIL\C51\LIB\C51S.LIB (?C?PLDIIDATA)
F:\KEIL\C51\LIB\C51S.LIB (?C?CCASE)
F:\KEIL\C51\LIB\C51S.LIB (PUTCHAR)

```

LINK MAP OF MODULE: bank\_ex3 (?C\_STARTUP)

| TYPE                            | BASE    | LENGTH  | RELOCATION | SEGMENT NAME |
|---------------------------------|---------|---------|------------|--------------|
| -----                           |         |         |            |              |
| * * * * * DATA MEMORY * * * * * |         |         |            |              |
| REG                             | 0000H   | 0008H   | ABSOLUTE   | "REG BANK 0" |
| REG                             | 0008H   | 0008H   | ABSOLUTE   | "REG BANK 1" |
| DATA                            | 0010H   | 0005H   | UNIT       | ?DT?C_MODUL  |
|                                 | 0015H   | 000BH   |            | *** GAP ***  |
| BIT                             | 0020H.0 | 0001H.1 | UNIT       | _BIT_GROUP_  |
|                                 | 0021H.1 | 0000H.7 |            | *** GAP ***  |
| DATA                            | 0022H   | 0014H   | UNIT       | _DATA_GROUP_ |
| IDATA                           | 0036H   | 0001H   | UNIT       | ?STACK       |
| * * * * * CODE MEMORY * * * * * |         |         |            |              |
| CODE                            | 0000H   | 0003H   | ABSOLUTE   |              |

```

 0003H 0008H *** GAP ***
CODE 000BH 0003H ABSOLUTE
CODE 000EH 002FH UNIT ?PR?TIMER0?C__MODUL
CODE 003DH 035CH UNIT ?PR?PRINTF?PRINTF
CODE 0399H 008EH UNIT ?C?LIB_CODE
CODE 0427H 0039H INBLOCK ?BANK?SELECT
CODE 0460H 0027H UNIT ?PR?MAIN?C__MODUL
CODE 0487H 0027H UNIT ?PR?PUTCHAR?PUTCHAR
CODE 04AEH 000CH UNIT ?C_C51STARTUP
CODE 04BAH 000AH UNIT ?CO?C__MODUL
 04C4H 003CH *** GAP ***
CODE 0500H 000CH PAGE ?BANK?SWITCH

* * * * * C O D E B A N K 1 * * * * *
 0000H 8000H *** GAP ***
BANK1 8000H 0010H UNIT ?PR?TINIT?C__MODUL

```

OVERLAY MAP OF MODULE: bank\_ex3 (?C\_STARTUP)

| SEGMENT                  | BIT_GROUP |         | DATA_GROUP |        |
|--------------------------|-----------|---------|------------|--------|
| ---> CALLED SEGMENT      | START     | LENGTH  | START      | LENGTH |
| -----                    | -----     | -----   | -----      | -----  |
| ?C_C51STARTUP            | -----     | -----   | -----      | -----  |
| ---> ?PR?MAIN?C__MODUL   |           |         |            |        |
| ?PR?MAIN?C__MODUL        | -----     | -----   | -----      | -----  |
| ---> ?PR?TINIT?C__MODUL  |           |         |            |        |
| ---> ?CO?C__MODUL        |           |         |            |        |
| ---> ?PR?PRINTF?PRINTF   |           |         |            |        |
| ?PR?PRINTF?PRINTF        | 0020H.0   | 0001H.1 | 0022H      | 0014H  |
| ---> ?PR?PUTCHAR?PUTCHAR |           |         |            |        |

INTRABANK CALL TABLE OF MODULE: bank\_ex3 (?C\_STARTUP)

| ADDRESS | FUNCTION NAME |
|---------|---------------|
| -----   | -----         |
| 045AH   | TINIT         |

## 11.2.4 高级语言控制命令

BL51/Lx51 的高级语言控制命令如表 11-6 所示。用于设定是否自动包含运行库、是否允许进行局部变量的数据覆盖, 以及提供对实时操作系统 RTX51 和 RTX51\_TINY 有关的

控制。下面介绍各条命令的使用方法。

表 11-6 BL51/Lx51 的高级语言控制命令

| BL51             | Lx51             | 缩 写  | 功能说明                   |
|------------------|------------------|------|------------------------|
| NODEFAULTLIBRARY | NODEFAULTLIBRARY | NLIB | 从运行库中删除模块              |
| [NO]OVERLAY      | [NO]OVERLAY      | NOO  | [4]允许进行数据覆盖            |
| RECURSIONS       | RECURSIONS       | RC   | 允许分析复杂递归应用程序的调用树       |
| REGFILE          | REGFILE          | RF   | 指定包含寄存器使用信息的生成文件名      |
| RTX51            | RTX51            | —    | 包含对 RTX51-full 实时内核的支持 |
| RTX51TINY        | RTX51TINY        | —    | 包含对 RTX51-tiny 实时内核的支持 |
| SPEEDOVL         | —                | SP   | 忽略从常数段到程序代码的分析引用       |

命 令 名: NODEFAULTLIBRARY

缩 写: NLIB

参 数: 无

默 认 值: 自动连接运行库。

μVision2: 在“Options 选项/BL51 Misc 标签页/Misc Controls 栏”键入命令。

功 能: 该命令禁止对运行库的自动连接。如果在调用连接定位器时不给出这条命令, 连接定位器自动将必要的运行库加入到指定的目标文件中, 并根据 Cx51 目标模块的编译模式及 C 语言源程序中是否使用了浮点运算, 从运行库中自动选择正确的库文件。

例 子: BL51 MYPROG.OBJ NODEFAULTLIBRARY

命 令 名: NOOVERLAY

缩 写: NOOL

参 数: 无

默 认 值: 允许进行数据覆盖。

μVision2: 在“Options 选项/BL51 Misc 标签页/Misc Controls 栏”键入命令。

功 能: NOOVERLAY 命令禁止进行数据覆盖, 采用该命令后连接定位器将不会覆盖局部变量和函数参数的数据空间。

例 子: Lx51 MYPROG.OBJ NOOVERLAY

命 令 名: OVERLAY

缩 写: OL

参 数: OVERLAY (sfname { ! | ~ } sfname [...])  
OVERLAY (sfname { ! | ~ } (sfname, sfname [...])) [...]  
OVERLAY (sfname ! \*)  
OVERLAY (\* ! sfname)

默 认 值: 允许进行数据覆盖。

- μVision2: 在“Options 选项/BL51 Misc 标签页/Overlay 栏”键入命令及参数。
- 功 能: OVERLAY 命令允许修改被连接定位器在覆盖分析中识别出的调用树。当应用程序中采用了函数指针或包含有实时操作系统的调度跳转时,要求对程序调用树进行调整。OVERLAY 命令只在至少有一个模块是用高级语言(Cx51 或 PL/M51)编写的情况下才被执行,该命令能够对高级语言程序模块中函数的局部段和位段进行正确的覆盖分析,如果不希望对某些程序的数据段和位段产生交叠,则可以采用 NOOVERLAY 命令来禁止覆盖。连接定位器通常只对具有相同类型的局部段进行覆盖分析,被覆盖的段必须具有 OVERLAYABLE 属性,Cx51 和 PL/M51 编译器可以自动为局部数据段和局部位段生成该属性。对于汇编语言程序中需要进行覆盖的段,必须在程序中明确定义成 OVERLAYABLE 再定位类型,具体定义方法请参见 10.2.1 节的 SEGMENT 伪指令,并且段名也必须遵守 Cx51 编译器的段名规则。
- 使用 OVERLAY 命令的一般格式如表 11-7 所示。

表 11-7 OVERLAY 命令的一般格式

| 命 令 格 式                                                            | 说 明                                                    |
|--------------------------------------------------------------------|--------------------------------------------------------|
| OVERLAY (* ! sfname)                                               | 将 sfname 排除在覆盖分析之外,并在不产生覆盖的存储器区域内定位数据段和位段,不影响其他函数的数据覆盖 |
| OVERLAY (sfname ! *)                                               | 同上                                                     |
| OVERLAY (sfname ! sfname1)<br>OVERLAY (sfname ! (sfname1, fname2)) | 给段或函数增加调用参考(禁止覆盖)                                      |
| OVERLAY (sfname ~ sfname1)<br>OVERLAY (sfname ~ (sfname1, fname2)) | 忽略段或函数之间的调用参考(允许覆盖)                                    |

注:表中 sfname 是一个段名或 C51 函数名,!表示在段或函数的调用表中增加一个调用,~表示从段或函数的调用表中删除一个调用,\*为表示所有段或函数的通配符。

例 子: 下面的例子使函数 FUNC1 的局部数据段和局部位段不与其他任何段产生覆盖。

```
BL51 SAMPLE.OBJ OVERLAY(FUNC1!*)
```

下面的例子规定 MAIN 函数可与 FCT1、FCT2 和 FCT3 函数产生覆盖,而 EXEC 函数不与 FCT1、FCT2 和 FCT3 函数产生覆盖。另外,函数 TESTFUNC1 被彻底排除在段的覆盖分析之外。

```
BL51 CMOD1.OBJ,CMOD2.OBJ OVERLAY(MAIN~(FCT1,FCT2,FCT3)
EXEC!(FCT1,FCT2,FCT3),TESTFUNC!*)
```

下面的例子在覆盖分析中忽略 FUNC1 和 FUNC2 对段?PR?MAINMOD 的引用:

```
BL51 MAINMOD.OBJ,TEXTOUT.OBJ OVERLAY (FUNC1~?PR?
MAINMOD,FUNC2~?PR?MAINMOD)
```

下面再举两个调整数据段覆盖分析的具体例子。

**例 11.27:** 函数指针作为函数参数的 C51 程序, 利用 OVERLAY 命令调整段的覆盖分析。C51 程序文件 OVL1.C 代码如下:

```
bit indirectfunc1(void) {
 unsigned char n1,n2;
 return(n1<n2);
}
bit indirectfunc2(void) {
 unsigned char a1,a2;
 return((a1=0x41)<(a2-0x41));
}
exec(bit (*fct)(void)) {
 unsigned int i,j,k;
 for(i=0; i<10; i++)
 if(*fct) i=10;
}

unsigned char SWITCH;

main() {
 if(SWITCH) exec(indirectfunc1);
 else exec(indirectfunc2);
}
```

对 OVL1.C 文件直接进行编译链接, 不采用 OVERLAY 命令调整数据段覆盖所生成的列表文件 OVL1.M51 中的覆盖分析映像 (OVERLAY MAP) 如下:

```
OVERLAY MAP OF MODULE: OVL1 (OVL1)
SEGMENT DATA_GROUP
+--> CALLED SEGMENT START LENGTH

?C_C51STARTUP -----
+--> ?PR?MAIN?OVL1

?PR?MAIN?OVL1 -----
+--> ?PR?INDIRECTFUNC1?OVL1
+--> ?PR?_EXEC?OVL1
+--> ?PR?INDIRECTFUNC2?OVL1

?PR?INDIRECTFUNC1?OVL1 0008H 0002H

?PR?_EXEC?OVL1 0008H 0004H
```

```
?PR?INDIRECTFUNC2?OVL1 0008H 0002H
```

本例中,函数 indirectfunc1 和 indirectfunc2 是通过 main()函数传递的指针间接调用的,然而列表文件中 OVERLAY MAP 表明,BL51 所识别的却是 main()函数对它们的调用,因此 BL51 错误地将函数 indirectfunc1、indirectfunc2 和 exec 的局部段进行的覆盖处理。纠正的办法是在采用 BL51 进行连接定位时使用如下带参数的 OVERLAY 命令:

```
BL51 OVL1.OBJ OVERLAY(MAIN~(INDIRECTFUNC1,INDIRECTFUNC2),
 EXEC! (INDIRECTFUNC1,INDIRECTFUNC2))
```

具体方法是在μVision2 环境下通过“Project 下拉菜单/Optionst 选项/BL51 Misc 标签页/Overlay 栏”键入“MAIN~(INDIRECTFUNC1,INDIRECTFUNC2),EXEC! (INDIRECTFUNC1,INDIRECTFUNC2)”,这样来对数据段的覆盖分析做出正确的安排,由此所产生的正确 OVERLAY MAP 如下:

```
OVERLAY MAP OF MODULE: OVL1 (OVL1)
SEGMENT DATA_GROUP
+--> CALLED SEGMENT START LENGTH

?C_C51STARTUP -----
+--> ?PR?MAIN?OVL1
?PR?MAIN?OVL1 -----
+--> ?PR?_EXEC?OVL1
?PR?_EXEC?OVL1 0008H 0004H
+--> ?PR?INDIRECTFUNC1?OVL1
+--> ?PR?INDIRECTFUNC2?OVL1
?PR?INDIRECTFUNC1?OVL1 000CH 0002H
?PR?INDIRECTFUNC2?OVL1 000CH 0002H
```

**例 11.28:** 数组中包含有函数指针的 Cx51 程序,利用 OVERLAY 命令调整段的覆盖分析。Cx51 程序文件 OVL2.C 代码如下:

```
#include <reg51.h>
#include <stdio.h>
void func1() {
 unsigned char i;
 for(i=0;i<10;i++) printf("This is function1\n");
}
void func2() {
 unsigned char i;
 for(i=0;i<10;i++) printf("This is function2\n");
}
```

```
code void (* functab[]) ()={func1,func2};
main() {
 (*functab[P1&0x01]) ();
}
```

对 OVL2.C 文件直接进行编译链接, 不采用 OVERLAY 命令调整数据段覆盖所生成的列表文件 OVL2.M51 中的覆盖分析映像 (OVERLAY MAP) 如下:

```
OVERLAY MAP OF MODULE: OVL2 (OVL2)
SEGMENT BIT_GROUP DATA_GROUP
+--> CALLED SEGMENT START LENGTH START LENGTH

?C_C51STARTUP ----- ----- ----- -----
+--> ?PR?MAIN?OVL2

?PR?MAIN?OVL2 ----- ----- ----- -----
+--> ?CO?OVL2

?CO?OVL2 ----- ----- ----- -----
+--> ?PR?FUNC1?OVL2
+--> ?PR?FUNC2?OVL2

?PR?FUNC1?OVL2 ----- ----- 0008H 0001H
+--> ?PR?PRINTF?PRINTF

?PR?PRINTF?PRINTF 0020H.0 0001H.1 0009H 0014H
+--> ?PR?PUTCHAR?PUTCHAR

?PR?FUNC2?OVL2 ----- ----- 0008H 0001H
+--> ?PR?PRINTF?PRINTF

.....

*** WARNING L13: RECURSIVE CALL TO SEGMENT
 SEGMENT: ?CO?OVL2
 CALLER: ?PR?FUNC1?OVL2
*** WARNING L13: RECURSIVE CALL TO SEGMENT
 SEGMENT: ?CO?OVL2
 CALLER: ?PR?FUNC2?OVL2
```

可以看到列表文件中包含有两个警告错误信息。这是因为本例中函数 func1、func2 被 main() 函数间接调用, 函数指针数组已被初始化成两个函数指针常数。Cx51 编译器规定对于字符串和初始化常数将被组合到一个单独的常数段中, 对于本例来说, 常数段 ?CO?OVL2 中存在对函数 func1 和 func2 的参考引用, 而 func 和 func2 又引用了常数段 ?CO?OVL2 中



的字符串, 因此, 无论 main() 函数如何调用, 当不加限制地使用覆盖分析功能时, BL51 都将认为 func1 和 func2 是被递归调用, 从而产生上述连接警告信息。

事实上由于函数 func1 和 func2 被 main() 函数所调用, 因此应将常数段 ?CO?OVL2 中对 func1 和 func2 的参考引用予以删除, 而在 main() 函数中则应增加对 func1 和 func2 的调用。为此可以使用如下连接控制命令:

BL51 OVL2.OBJ OVERLAY(?CO?OVL2~(func1,func2),main!(func1,func2))

具体方法是在  $\mu$ Vision2 环境下通过 “Project 下拉菜单/Optionst 选项/BL51 Misc 标签页/Overlay 栏” 键入 “?CO?OVL2~(func1,func2),main!(func1,func2)”。这样调整之后 BL51 才可以进行正确的覆盖分析, 连接定位时就不会产生上述警告错误信息了。另外, 也可以利用 BL51 的控制命令 SPEEDOVL 来忽略所有从常数段对程序代码的参考引用, 这样就只需要在 main() 函数中增加对 func1 和 func2 的调用:

BL51 OVL2.OBJ OVERLAY(main!(func1,func2)) SPEEDOVL

具体方法是在  $\mu$ Vision2 环境下通过 “Project 下拉菜单/Optionst 选项/BL51 Misc 标签页/Overlay 栏” 键入 “main!(func1,func2)”, 同时在 “Optionst 选项/BL51 Misc 标签页/Misc control 栏” 键入 “SPEEDOVL”。

覆盖分析调整之后 BL51 所生成的正确 OVERLAY MAP 如下:

```
OVERLAY MAP OF MODULE: OVL2 (OVL2)
SEGMENT BIT_GROUP DATA_GROUP
+--> CALLED SEGMENT START LENGTH START LENGTH

?C_C51STARTUP -----
+--> ?PR?MAIN?OVL2

?PR?MAIN?OVL2 -----
+--> ?CO?OVL2
+--> ?PR?FUNC1?OVL2
+--> ?PR?FUNC2?OVL2

?PR?FUNC1?OVL2 ----- 0008H 0001H
+--> ?CO?OVL2
+--> ?PR?PRINTF?PRINTF

?PR?PRINTF?PRINTF 0020H.0 0001H.1 0009H 0014H
+--> ?PR?PUTCHAR?PUTCHAR

?PR?FUNC2?OVL2 ----- 0008H 0001H
+--> ?CO?OVL2
+--> ?PR?PRINTF?PRINTF

命令名: RECURSIONS
缩 写: RC
```

- 参 数: RECURSIONS (number of recursions)  
默 认 值: RECURSIONS(10)  
 $\mu$ Vision2: 在“Options 选项/BL51 Misc 标签页/Misc Controls 栏”键入命令。  
功 能: RECURSIONS 命令用于指定连接定位器产生致命错误 FATAL ERROR 232 之前, 应用程序发生递归调用的次数。每次连接定位器在对应用程序进行覆盖分析中遇到递归调用, 这个递归调用将自动从递归调用树删除, 然后重新启动覆盖分析。可以在每个复杂的递归调用应用程序中增加可以接受的递归次数, 但这会显著增加连接定位器的执行时间。如果应用程序中包含很多指向函数表的指针, 则在采用 OVERLAY 命令调整调用树之前可能会发生致命错误 FATAL ERROR 232, 此时可以采用 RECURSIONS 命令来分析应用程序的覆盖映像 (OVERLAY MAP)。
- 另 见: OVERLAY, SPEEDOVL  
例 子: Lx51 MYPROG.OBJ, A.OBJ, B.OBJ RECURSIONS(100)
- 命 令 名: REGFILE  
缩 写: RF  
参 数: REGFILE(filename)  
默 认 值: 不产生寄存器使用文件。  
 $\mu$ Vision2: 在“Options 选项/Cx51 Compiler 标签页/Code Optimization 栏”选中“Global Register Coloring”复选框。  
功 能: REGFILE 命令用于指定连接定位器生成的寄存器使用文件名。寄存器使用文件中的信息被提供给 Cx51 编译器进行全局寄存器优化, 即在调用外部函数时 Cx51 编译器将对寄存器的使用进行优化。  
例 子: Lx51 MYPROG.OBJ,A.OBJ,B.OBJ REGFILE(MYPROG.REG)
- 命 令 名: RTX51, RTX51TINY  
缩 写: 无  
参 数: 无  
默 认 值: 无。  
 $\mu$ Vision2: 在“Options 选项/Target 标签页/Operating 栏”选取实时操作系统。  
功 能: 该命令通知连接定位器应用程序必须采用 RTX51 实时操作系统进行连接, 包括解决从应用程序调用实时操作系统函数库的问题。  
例 子: BL51 RTX\_EX1.OBJ RTX51
- 命 令 名: SPEEDOVL, 仅用于 BL51, Lx51 通常忽略在覆盖分析过程中常数段对程序代码的引用。  
缩 写: SP  
参 数: 无  
默 认 值: 在覆盖分析过程中考虑常数段对程序代码的引用。

- μVision2: 在“Options 选项/BL51 Misc 标签页/Misc Controls 栏”键入命令。
- 功 能: RECURSIONS 命令使连接定位器忽略覆盖分析过程中常数段对程序代码的引用, 这对使用数组和指向函数的指针一类的应用程序十分有用。但是, 采用 SPEEDOVL 命令将会使连接定位器与已有的用 OVERLAY 命令调整调用树的应用程序不兼容。
- 另 见: OVERLAY
- 例 子: BL51 MYPRG.OBJ SPEEDOVL

## 11.3 符号转换工具

### 11.3.1 Intel HEX 文件格式与符号转换工具

许多 EPROM 编程器都要求输入文件具有 Intel HEX 格式, 一个 Intel HEX 文件的一行称为一个记录, 每个记录都是由 16 进制字符组成的, 两个字符表示一个字节的值。Intel HEX 文件通常由若干个记录组成, 每个记录都具有如下的形式:

: 11 aaaa tt dd...dd cc

其中,

: 是记录起始标志。Intel HEX 文件的每一行都是以: 开头的。

11 记录长度。用来表示该记录的数据字节数。

aaaa 装入地址。它是该记录中第一个数据字节的 16 位地址值, 用来表示该记录在 EPROM 存储器中的起始绝对地址。

tt 记录类型。00 表示数据记录, 01 表示 EOF (文件结束)。

dd...dd 记录的实际字节数据值。每一个记录都有 11 个字节的数据值。

cc 校验和。将它的值与记录中所有字节 (包括记录长度字节) 内容相加, 其结果应为 0, 如果为其他值则表明该记录有错。

一个 Intel HEX 文件中所有数据记录都为 00 记录类型, 当记录类型为 01 时表示文件结束, 具有 01 记录类型的那一行的内容总是: 00000001FF。下面是一个完整的 Intel HEX 文件内容:

```
: 070003008E0F8F108D1122FA
: 10000A007F117E227D30D200750E44750D55750C18
: 0A001A0066750B77C2011200242264
: 100024008E088F098D0A7F227E117D03C202751509
: 1000340077751466751355751244D20312000322A2
: 030000000020044B7
: 0C004400787FE4F6D8FD75812002000AE8
: 00000001FF
```

由连接定位器生成的绝对目标文件需要经过转换成为 HEX 文件才能用于 EPROM 编程, Keil Cx51 软件包中提供了如表 11-8 所示的转换工具。

表 11-8 符号转换工具

| 从……输出       | 符号转换工具      | 说 明                      |
|-------------|-------------|--------------------------|
| BL51        | OH51        | 适用于没有代码分组的传统 8051 应用程序   |
| BL51 分组应用程序 | OH51 与 OC51 | 适用于代码分组的传统 8051 应用程序     |
| Lx51        | OHX51       | 适用于传统 8051 和扩展 8051 应用程序 |

在 Windows 集成开发环境  $\mu$ Vision2 中, 在“Project 菜单/Options 选项/Output 标签页”选中“Creat Executable”圆形单选框以及它下面的“Creat HEX File”方形复选框,  $\mu$ Vision2 项目管理器将自动选取正确的符号转换工具, 需要注意的是采用 BL51 时, 应将 HEX 文件格式选定为 HEX-80, 采用 Lx51 时, 应将 HEX 文件格式选定为 HEX-386。

### 11.3.2 引用符号转换工具 OH51/OHX51

通过命令行引用 OH51/OHX51 符号转换工具的基本格式如下:

OH51 abs\_file [HEXFILE (file)]

OHX51 abs\_file [HEXFILE (file)][H386][RANGE (start-end)][OFFSET (offset)]

其中:

abs\_file 连接定位器生成的绝对目标文件。

file 转换生成的 Intel HEX 文件名, 默认为绝对目标文件名。

H386 指定转换文件具有 HEX-386 格式, 如果指定的地址范围超过 64KB, 将自动采用这个格式。

start-end 指定绝对目标文件转换为 Intel HEX 文件的地址范围, 默认地址范围由所选用的器件决定, 如表 11-9 所示。

Offset 指定附加到保存在绝对目标文件地址上的偏移量。

表 11-9 转换为 Intel HEX 文件的地址范围

| OHX51 转换工具              | 地址范围              |
|-------------------------|-------------------|
| OHX51 (传统和扩展 8051)      | C:0x0000~C:0xFFFF |
| OHX51 (Philips 80C51Mx) | 0x800000~0x80FFFF |

下面的例子为传统 8051 器件分组应用生成一个 HEX 文件, 只转换代码组 0:

OHX51 PROG RANGE (B:0~B:0xFFFF)

下面的例子为 Philips 80C51Mx 单片机生成 Intel HEX386 格式文件, OFFSET 控制用于创建一个可被直接编程到 EPROM 的输出文件:

OHX51 MYAPP RANGE (0x800000~0x81FFFF) OFFSET (-0x800000)

下面的例子将保存在 XDATA 空间的常数转换到一个 HEX 文件中:

OHX51 MYPROG RANGE (X:0~X:FFFF)

对于由扩展连接定位器 Lx51 生成的代码分组目标文件, 可以采用转换工具 OHX51 的默认设置生成包含公共区和所有代码组的 Intel HEX386 格式文件。如果应用程序中没有代码组 0, 则 OHX51 将跳过这个存储区。图 11.8 所示为前面例 11.15 中采用 4 个 64KB 代

码组的 HEX 文件地址分配, 图 11.9 所示为前面例 11.18 例带有公共区域存储器代码分组的 HEX 文件地址分配。

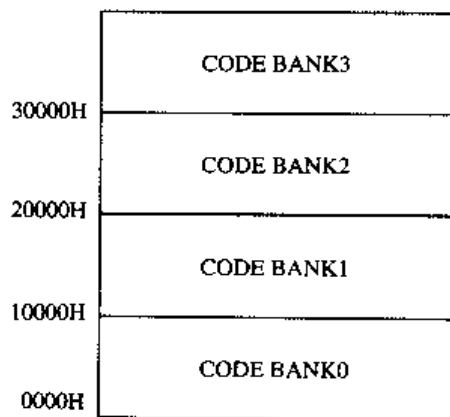


图 11.8 4 个 64KB 代码组 HEX 文件地址安排

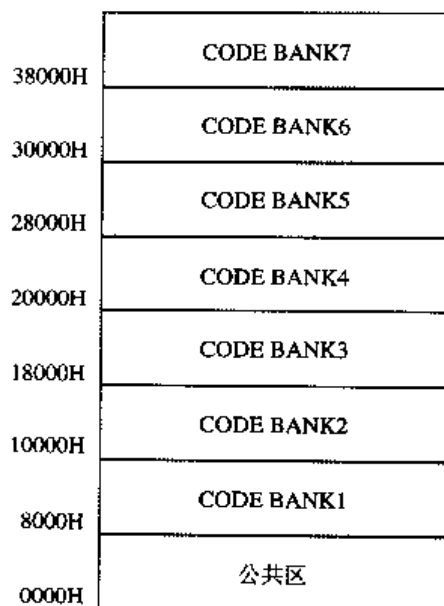


图 11.9 带有公共区域存储器代码分组 HEX 文件地址安排

### 11.3.3 引用分组目标文件转换工具 OC51

分组目标文件转换程序 OC51 用于将由 BL51 连接定位器生成的代码分组目标文件转换成“标准”的绝对目标文件, 还可以将分组目标文件中的符号调试信息也转换成绝对目标文件的形式, OC51 所产生的输出文件可以直接装入仿真器中进行调试, 对于转换后的“标准”绝对目标文件可以利用 OH51 将其转换为 Intel HEX 文件, 这些文件可以被独立地编程到相应的 EPROM 物理地址空间。

通过命令行引用 OC51 的格式如下:

OC51 分组的目标文件

其中: “分组的目标文件”是 BL51 所输出的“分组”目标文件。

例如: OC51 MYPROG 就是从“分组”的目标文件 MYPROG 中产生“标准”的绝对目标文件。OC51 为每个代码组产生一个独立的绝对目标文件, 这些文件中包含了相应代码组和公共区域的全部信息。各个输出文件分别为: 代码组 BANK0 为 MYPROG.B00, 代码组 BANK1 为 MYPROG.B01..., 代码组 BANKn 为 MYPROG.Bn(n=0,1,...,31)。

## 11.4 库管理器 LIBx51

库文件实际上是一些目标模块的格式化组合, 用户可以利用库管理器来生成自己的库, 也可以在已有的库文件中加入或删除某个程序模块。Keil Cx51 软件包中提供如表 11-10 所示的库管理工具。

表 11-10 库管理器工具

| 库管理器   | 处理来自……的文件                                                           | 说 明                                                                                             |
|--------|---------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| LIB51  | A51 宏汇编器<br>C51 编译器<br>Intel ASM51 汇编器<br>Intel PL/M51 编译器          | 适用于传统 8051，支持 32×64KB 代码分组                                                                      |
| LIBX51 | A51、A51 宏汇编器<br>C51、CX51 编译器<br>Intel ASM51 汇编器<br>Intel PL/M51 编译器 | 适用于传统 8051 以及扩展 8051(Philips 80C51Mx, Dallas 80C390 等)，支持代码分组和数据分组，支持高达 16MB 的 CODE 和 XDATA 存储器 |

在 Windows 集成开发环境  $\mu$ Vision2 中，在“Project 菜单/Options 选项/Output 标签页”选中圆形单选框“Creat Library”， $\mu$ Vision2 将自动调用正确的库管理器来生成库文件，同时不再调用连接定位器来生成绝对目标文件。

也可以通过命令行来调用库管理器，调用格式如下：

LIB51 [控制命令]

LIBx51 [控制命令]

其中，方括号内的选项是如表 11-11 所示的 LIB51 库管理控制命令。

表 11-11 LIBx51 库管理器控制命令

| 控制命令    | 命 令 功 能                                                                                                                            |
|---------|------------------------------------------------------------------------------------------------------------------------------------|
| ADD     | 向库中加入一个程序模块。例如，向库文件 MYLIB.LIB 中添加目标模块 GOODCODE.OBJ: LIB51 ADD GOODCODE.OBJ TO MYLIB.LIB                                            |
| CREATE  | 生成一个新的空库。例如，创建一个名为 MYLIB.LIB 的库文件:<br>LIB51 CREATE MYLIB.LIB                                                                       |
| DELETE  | 从一个库中删除程序模块。例如，从库文件 MYLIB.LIB 中删除 GOODCODE 模块:<br>LIBX51 DELETE MYLIB.LIB (GOODCODE)                                               |
| EXTRACT | 从库文件中提取一个目标模块。例如，将 GOODCODE 模块复制到目标文件 GOOD.OBJ:<br>LIB51 EXTRACT MYLIB.LIB (GOODCODE) TO GOOD.OBJ                                  |
| EXIT    | 退出库管理器                                                                                                                             |
| HELP    | 显示库管理器的帮助信息                                                                                                                        |
| LIST    | 显示一个库中的模块名和公共符号名。例如，生成一个列表文件 MYLIB.LST，其中包含保存在 MYLIB.LIB 库文件中的模块名以及公共符号:<br>LIB51 LIST MYLIB.LIB TO MYLIB.LST PUBLICS              |
| REPLACE | 替换库文件中已有的目标模块。例如，替换库文件 MYLIB.LIB 中的目标模块 GOODCODE.OBJ，如果该模块不存在，将在库文件中添加 GOODCODE.OBJ:<br>LIB51 REPLACE GOODCODE.OBJ IN MYLIB.LIB    |
| TRASFER | 生成一个全新的库并添加目标模块。例如，删除已有的库文件 MYLIB.LIB，然后重新创建这个库并添加目标模块 FILE1.OBJ 和 FILE2.OBJ:<br>LIBX51 TRANSFER FILE1.OBJ, FILE2.OBJ TO MYLIB.LIB |

也可以在命令状态下键入 LIBX51 (或 LIB51) 后回车, 屏幕上将显示出 “\*” 提示符, 然后再键入表 11-11 中的各个控制命令。

下面介绍库管理器 LIB51 各个控制命令在命令行状态下的使用方法。

#### (1) 帮助命令

显示库管理器的帮助信息可以采用 HELP, 使用格式为:

**HELP**

例子: **LIB51 HELP**

**\* HELP**

#### (2) 退出命令

退出库管理器可以采用 EXIT 命令, 格式如下:

**\* EXIT**

#### (3) 创建库文件

创建库文件可以使用 CREATE 命令或 TRANSFER 命令。CREATE 命令用于创建一个空的库文件, 格式如下:

**CREATE libfile**

其中, “libfile” 是要创建的库文件名, 而且需要包含扩展名, 通常扩展名为 .lib。

例子:

**LIBX51 CREATE FASTMATH.LIB**

**\* CREATE FASTMATH.LIB**

TRANSFER 命令用于创建一个新的库文件并向其中添加目标模块, 格式如下:

**TRANSFER filename [(modulename, ...)] [, ...] TO libfile**

其中, “filename” 为目标文件或库文件名, 可以指定几个文件, 中间用逗号隔开。

“modulename” 为库文件中的模块名, 如果省略模块名, 则整个目标文件将直接加入到库文件中去。模块名必须写在圆括号中, 可以使用多个模块名, 中间用逗号隔开。

“libfile” 为要创建的库文件名。如果指定的库文件已经存在, 则将被删除并被新创建的库文件所取代。

例子:

**LIB51 TRANSFER FILE1.OBJ, FILE2.OBJ TO MYLIB.LIB**

**LIBX51 @mycmd.lin**

mycmd.lin 文件的内容如下:

**TRANSFER FILE1.OBJ, FILE2.OBJ, FILE3.OBJ TO MYLIB.LIB**

#### (4) 添加或替换目标模块

向已有的库文件中添加或替换目标模块可以采用 ADD 或 REPLACE 命令。ADD 命令用于向已有的库文件中添加一个或多个目标模块, 格式如下:

**ADD filename [(modulename, ...)] [, ...] TO libfile**

其中, “filename” 为目标文件或库文件名, 可以指定几个文件, 中间用逗号隔开。

“modulename” 为库文件中的模块名, 如果省略模块名, 则整个目标文件将直接加入到库文件中去。模块名必须写在圆括号中, 可以使用多个模块名, 它们之间用逗号隔开。

“libfile”为要创建的库文件名，指定的目标模块将被添加到这个库文件中。

REPLACE 命令用于替换库文件中已有的目标模块，如果模块不存在，则在库文件中添加这个模块，格式如下：

**REPLACE filename IN libfile**

其中，“filename”为要更新的目标文件名。

“libfile”为已有的库文件名，该库文件中指定的目标模块将被替换。

例子：

**LIBX51 REPLACE MOD1.OBJ IN MYLIB.LIB**

**\* REPLACE FPMOD.OBJ TO FLOAT.LIB**

#### (5) 删除目标模块

从库文件中删除目标模块可以采用 DELETE 命令，格式如下：

**DELETE libfile (modulename [, modulename ...])**

其中，“libfile”为已有的库文件名，将从该库文件中删除指定的目标模块。

“modulename”为将被删除的目标模块名，可以使用多个模块名，它们之间用逗号隔开。

例子：

**LIB51 DELETE NEW.LIB (MODUL1)**

**\* DELETE NEW.LIB (FPMULT, FPDIV)**

#### (6) 提取目标模块

在库文件中为指定的模块创建一个标准模块可以采用 EXTRACT 命令，格式如下：

**EXTRACT libfile (modulename) TO filename**

其中，“libfile”为已有的库文件名。

“modulename”为库文件中的模块名，模块名只能有一个，并且必须放在圆括号中。

“filename”从库文件中创建的目标模块名。

例子：

**LIBX51 EXTRACT FLOAT.LIB(FPMUL) TO FLOATMUL.OBJ**

#### (7) 列出库文件内容

列出一个库文件中的目标模块名或所指定的公共符号名可以采用 LIST 命令，格式如下：

**LIST libfile [TO listfile] [PUBLICS]**

其中，“libfile”为已有的库文件名。

“listfile”为保存输出列表信息的文件名，如果不指定该文件名，列表信息将被显示在屏幕上。

“PUBLICS”为指定包含在列表文件中的公共符号，如果没有该选项，则只显示模块名。

例子：

**LIBX51 LIST NEW.LIB**

**\* LIST NEW.LIB TO NEW.LST PUBLICS**



## 第 12 章 RTX51 实时多任务操作系统

RTX51 是用于 8051 系列单片机的一种实时多任务操作系统 (RTOS)，利用它可以简化具有实时性要求的多任务复杂软件的设计。RTX51 有两个版本：

**RTX51 FULL** 允许 4 个优先权任务的循环和切换，并且可以并行地利用中断功能。RTX51 FULL 支持与邮箱系统(mailbox system)的信号和消息传递，可以进行存储器分配或释放，可以强迫某个任务进入等待中断、超时(timeout)、另一个任务或中断的信号或消息。

**RTX51 TINY** 它是 RTX51 FULL 的一个子集，可以在没有外部扩展数据存储器的 8051 系统中运行。RTX51 TINY 允许循环任务切换，支持信号传递，可以并行地利用中断功能，具有以下等待操作：中断、超时、另一个任务或中断的信号，但它不支持占先式任务切换，不能进行信息处理，也不支持存储器分配或释放。

### 12.1 RTX51 一般介绍

在许多单片机应用系统中要求能够同时进行多任务处理，RTX51 实时多任务操作系统 (RTOS) 为单片机的时间分配提供了一种灵活的方式，并且可以在所有 8051 系列单片机上实现。使用 RTX51 实时多任务操作系统十分容易，用户可以按一般标准方式编写自己的应用程序，但是要将包含有关实时多任务操作系统的头文件和库文件添加到自己的项目之中，另外在编译连接时需要使用一些额外的控制命令。

#### 1. 简单的单任务程序

一个标准的 C 语言程序总是从 main() 函数开始执行，而在 8051 单片机应用系统中，这个 main() 函数通常是无穷循环，它可以看成是一个连续执行单任务程序。如下例所示。

例 12.1:

```
int counter;
void main (void) {
 counter = 0;
 while (1) {
 counter++;
 }
}
```

/\* 无穷循环 \*/  
/\* 计数器加 1 \*/

## 2. 循环任务切换

一个复杂的 C 语言程序可能需要处理多个任务，下面是一个不采用 RTOS 而在无穷循环中交替调用各个功能函数来完成多任务处理的例子。

### 例 12.2:

```
int counter;
void main (void) {
 counter = 0;
 while (1) { /* 无穷循环 */
 check_serial_io ();
 process_serial_cmds (); /* 处理串行口输入 */
 check_kbd_io ();
 process_kbd_cmds (); /* 处理键盘输入 */
 adjust_ctrlr_parms (); /* 调整控制器 */
 counter++; /* 计数器加 1 */
 }
}
```

## 3. RTX51 的循环任务调度

RTX51 允许同时“准并行”地执行多个任务：各个任务并非持续运行，而是在预先设定的时间片（time slice）内执行。CPU 执行时间被划分为若干时间片，RTX51 为每个任务分配一个时间片，在一个时间片内允许执行某个任务，然后 RTX51 切换到另一个就绪的任务并允许它在其规定的时间片内执行。由于各个时间片非常短，通常只有几个毫秒，因此各个任务看起来似乎就是被同时执行了。

RTX51 利用 8051 单片机内部定时器的中断功能实现定时，用周期性定时中断驱动 RTX51 的时钟。使用 RTX51 时用户程序中不需要包含 main() 函数，它会自动地从任务 0 开始运行。如果用户程序中包含有 main() 函数，则需要利用 os\_start\_system 函数（RTX51 FULL）或 os\_create\_task 函数（RTX51 TINY）来启动 RTX51 实时操作系统。

例 12.3 使用了这种循环任务调度技术。程序中只有两个简单计数器循环任务，RTX51 从函数名为 job0 的任务开始执行，在该函数中添加另一个名为 job1 的任务，函数 os\_create(1) 表示任务 1 已经就绪准备执行。当 job0 的时间片执行完后 RTX51 将任务切换到 job1，当 job1 的时间片执行完后 RTX51 再将任务切换到 job0，如此反复循环。

### 例 12.3:

```
#include <rtx51tny.h>
int counter0;
int counter1;
void job0 (void) _task_ 0 {
 os_create (1); /* 将任务 1 标记为就绪 */
 while (1) { /* 无穷循环 */
 counter0++; /* 更新计数器 */
 }
}
```

```

}
void job1 (void) _task_ 1 {
 while (1) { /* 无穷循环 */
 counter1++; /* 更新计数器 */
 }
}

```

#### 4. RTX51 事件

RTX51 提供了一个 `os_wait` 函数，它可以为 CPU 的时间分配提供更为有效的方式。可以用 `os_wait` 函数通知 RTX51，让它执行下一个任务而不是等待一个任务的时间片到期。`os_wait` 函数会挂起当前正在执行的函数而等待某个特殊事件，在等待期间可以执行其他任务。

在 RTX51 中使用“超时” (time-out)

例 12.4 使用了“超时” (time-out) 进行时间延时，本例与例 12.3 相似，在 `job0` 中允许 `job1`，但在 `counter0` 增量后，`job0` 调用 `os_wait` 函数完成 3 个时钟节拍的延时，在此期间 RTX51 切换到下一个任务即 `job1`。在 `job1` 中 `counter1` 增量后，`job1` 调用 `os_wait` 函数完成 5 个时钟节拍的延时。此时 RTX51 没有其他需要执行的任务，因此进入空循环等待 5 个时钟节拍，然后再次进入任务 `job0`。最后的结果是：每隔 3 个时钟节拍 `counter0` 增量一次，每隔 5 个时钟节拍 `counter1` 增量一次。

##### 例 12.4:

```

#include <rtx51tny.h>
int counter0;
int counter1;
void job0 (void) _task_ 0 {
 os_create (1); /* 将任务 1 标记为就绪 */
 while (1) { /* 无穷循环 */
 counter0++; /* 更新计数器 */
 os_wait (K_TMO, 3); /* 暂停 3 个时钟节拍 */
 }
}

void job1 (void) _task_ 1 {
 while (1) { /* 无穷循环 */
 counter1++; /* 更新计数器 */
 os_wait (K_TMO, 5); /* 暂停 5 个时钟节拍 */
 }
}

```

#### 5. 在 RTX51 中使用信号 (signal)

`os_wait` 函数的另一个事件是信号 (signal) 或二进制信号量 (binary semaphore)。信号可用于任务的相互排序，例如，一个任务利用 `os_wait` 进行等待直到另一个任务产生一

个信号。其操作过程为如果一个任务要等待一个信号，并且信号标志为 0，则任务将被挂起，直到收到信号。如果信号标志在任务进行信号查询时已经为 1，则清除信号标志并继续执行任务。例 12.5 说明了这种应用，job1 等待 job0 产生一个信号，当 job1 收到一个信号时将 counter1 加 1，然后再次等待另一个信号。job0 的 counter0 不断加 1 直到溢出，此时 job0 发送一个信号到 job1，RTX51 将 job1 标记为就绪，RTX51 直到下一个时钟节拍之后才会启动 job1。

### 例 12.5:

```
#include <rtx51tiny.h>
int counter0;
int counter1;
void job0 (void) __task__ 0 {
 os_create (1); /* 将任务 1 标记为就绪 */
 while (1) { /* 无穷循环 */
 if (++counter0 == 0) /* 更新计数器 */
 os_send_signal (1); /* 给任务 1 发信号 */
 }
}

void job1 (void) __task__ 1 {
 while (1) { /* 无穷循环 */
 os_wait (K_SIG, 0, 0); /* 等待信号 */
 counter1++; /* 更新计数器 */
 }
}
```

## 6. 优先级和占先

在例 12.5 中当 job0 发送一个信号到 job1 之后，job1 并不马上开始执行，而是在 job0 的“超时”(time-out)出现后才开始执行，这在某些应用场合是不希望的。RTX51 允许指定任务的优先级 (priority)，高优先级的任务可以中断低优先级的任务，或比低优先级的任务抢先执行，这就是所谓占先式多任务系统 (注意，RTX51 TINY 不支持优先级任务和占先)。默认情况下所有任务的优先级均为 0，这是最低优先级，优先级可以为 0~3。例如，可将例 12.5 中的 job1 重新定义如下：

```
void job1 (void) __task__ 1 __priority__ 1 {
 while (1) { /* 无穷循环 */
 os_wait (K_SIG, 0, 0); /* 等待信号 */
 counter1++; /* 更新计数器 */
 }
}
```

此时 job1 的优先级高于 job0，因此当 job0 的一个信号到达时 job1 将马上开始执行。

## 7. 中断

RTX51 可以与中断函数并行工作, 中断函数可以与 RTX51 进行通信, 也可以向 RTX51 的任务发送信号或消息, RTX51 FULL 允许为任务分配中断。

## 8. 消息传递

RTX51 FULL 支持任务与函数之间的消息交换: “发送”、“接收”以及“等待”消息。一条消息是一个 16 位的值, 它可以是一个数据, 也可以是指向某一块存储器区域的地址指针。RTX51 FULL 使用存储器池系统支持可变长度的消息。

## 9. CAN 总线通信

采用 RTX51/CAN 可以很方便地实现 CAN 总线网络 (Controller Area Networks), RTX51/CAN 是一种集成到 RTX51 FULL 中的 CAN 总线任务, 一个 RTX51 CAN 任务通过 CAN 总线网络进行消息传递, 其他 CAN 站点可以使用 RTX51 也可以不使用 RTX51 来进行配置。

## 10. BITBUS 总线通信

RTX51 FULL 覆盖了主、从位总线 BITBUS 任务, 并且支持采用 Intel 8044 进行信息传递。

## 11. 事件

在 wait 函数中 RTX51 支持以下事件。

- ① 超时 (Timeout): 挂起正在运行的任务, 等待指定的时钟节拍数。
- ② 间隔 (Intervals): 与超时相似, 但不是采用软件定时器的复位来产生周期性间隔, 仅用于 RTX51 TINY。
- ③ 信号 (Signals): 协调内部任务操作。
- ④ 消息 (Messages): 在任务间互传消息, 仅用于 RTX51 FULL。
- ⑤ 中断 (Interrupts): 处理硬件中断, 仅用于 RTX51 FULL。
- ⑥ 信号量 (Semaphores): 共享有限资源, 仅用于 RTX51 FULL。

RTX51 已经全部集成在 Cx51 中, 生成可执行的 RTX51 应用程序是十分容易的。上面列举的例子都是可执行的 RTX51 应用程序。用户没有必要编写 8051 汇编语言子程序或函数, 而只需要将所编写的 RTX51 应用程序文件采用 Cx51 编译器和 BL51 连接定位器进行编译连接即可。在 Windows 集成开发环境  $\mu$ Vision2 中, 将用户编写的任务函数程序文件添加到项目之中, 在“Project 菜单/Options 选项/Target 标签页”的“Operating”栏选取“RTX-51 Full”或“RTX-51 Tiny”, 然后对项目进行整体创建即可很方便地完成 RTX51 应用程序的编译和连接定位。也可以通过命令行进行引用, 例如, 对于用户程序 EXAMPLE.C 采用 RTX51 FULL 编译连接的格式如下:

```
C51 EXAMPLE.C
```

```
BL51 EXAMPLE.OBJ RTX51
```

采用 RTX51 TINY 编译连接的格式如下:

```
C51 EXAMPLE.C
```

BL51 EXAMPLE.OBJ RTX51TINY

## 12.2 RTX51 技术参数

表 12-1 列出了 RTX51 的技术参数。

表 12-1 RTX51 的技术参数

| 描 述      | RTX51 FULL                                                                                 | RTX51 TINY                                                 |
|----------|--------------------------------------------------------------------------------------------|------------------------------------------------------------|
| 任务数      | 最多 256 个任务，可同时<br>激活 19 个任务                                                                | 最多可定义 16 个任务，所有任务<br>可同时激活                                 |
| 需要的存储器空间 | 40~46 字节 DATA 空间<br>20~200 字节的 IDATA 空间(用户栈)<br>最小 650 字节 XDATA 空间<br>6000~8000 字节 CODE 空间 | 7 字节 DATA 空间<br>3 倍于任务数字节的 IDATA 空间<br>—<br>900 字节 CODE 空间 |
| 硬件要求     | Timer0 或 Timer1                                                                            | Timer0                                                     |
| 系统时钟     | 1000~40000 个时钟周期                                                                           | 1000~65535 个时钟周期                                           |
| 中断请求时间   | < 50 个时钟周期                                                                                 | < 20 个时钟周期                                                 |
| 任务切换时间   | 快速任务: 70~100 个时钟周期<br>标准任务: 180~700 时钟周期, 取决于堆栈<br>负载                                      | 100~700 个时钟周期,<br>取决于堆栈负载                                  |
| 邮箱系统     | 8 个邮箱, 每个有 8 个整型入口                                                                         | —                                                          |
| 存储器池系统   | 最多 16 个存储器池                                                                                | —                                                          |
| 信号通知     | 8×1bit                                                                                     | —                                                          |

表 12-2 列出了 RTX51 支持的函数, RTX51 TINY 仅支持其中带“\*”的函数。

表 12-2 RTX51 的函数, 其中带“\*”者为 RTX51 TINY 支持的函数

| 函 数                 | 描 述              | 运行时间 (时钟周期)                                 |
|---------------------|------------------|---------------------------------------------|
| os_create(*)        | 将任务移送到执行排队       | 302                                         |
| os_delete(*)        | 从执行队列中删除一个任务     | 172                                         |
| os_send_signal(*)   | 向任务发送信号 (从任务中调用) | 408 (带任务切换)<br>316 (带快速任务切换)<br>71 (不带任务切换) |
| os_clear_signal(*)  | 删除一个已经发送的信号      | 57                                          |
| isr_send_signal(*)  | 向任务发送信号 (从中断中调用) | 46                                          |
| os_wait(*)          | 等待事件             | 68 (等待信号)<br>160 (等待消息)                     |
| os_attach_interrupt | 为中断源分配任务         | 119                                         |

续表

| 函 数                               | 描 述                 | 运行时间 (时钟周期)                                 |
|-----------------------------------|---------------------|---------------------------------------------|
| os_detach_interrupt               | 删除中断源任务分配           | 96                                          |
| os_disable_isr                    | 禁止 8051 硬件中断        | 81                                          |
| os_enable_isr                     | 允许 8051 硬件中断        | 80                                          |
| os_send_message/<br>os_send_token | 发送消息/设置信号量 (从任务中调用) | 443 (带任务切换)<br>343 (带快速任务切换)<br>94 (不带任务切换) |
| isr_send_message                  | 发送消息 (从中断中调用)       | 53                                          |
| isr_rcv_message                   | 接收消息 (从中断中调用)       | 71 (带消息)                                    |
| os_create_pool                    | 定义一个存储器池            | 644 (大小为 20×10 字节)                          |
| os_get_block                      | 从存储器池获得一个存储器块       | 148                                         |
| os_free_block                     | 释放一个存储器块到存储器池       | 160                                         |
| os_set_slice                      | 定义 RTX51 系统时钟值      | 67                                          |

除了表 12-2 所列出的函数之外, RTX51 还提供如表 12-3 所示附加调试函数。

表 12-3 RTX51 的附加调试函数

| 函 数                 | 说 明             |
|---------------------|-----------------|
| oi_reset_int_mask   | 禁止 RTX51 的外部中断源 |
| oi_set_int_mask     | 允许 RTX51 的外部中断源 |
| os_check_tasks      | 返回系统中所有任务的信息    |
| os_check_task       | 返回指定任务的信息       |
| os_check_mailboxes  | 返回系统中所有邮箱的状态信息  |
| os_check_mailbox    | 返回指定邮箱的状态信息     |
| os_check_semaphores | 返回系统中所有信号量的状态信息 |
| os_check_semaphore  | 返回指定信号量的状态信息    |
| os_check_pool       | 返回存储池中的块信息      |

表 12-4 列出了 RTX51 支持的 CAN 总线函数, 能支持的 CAN 总线控制器有: Philips82C200、80C592、Intel 82526 等, CAN 函数只有在 RTX51 FULL 中提供。

表 12-4 RTX51 支持的 CAN 总线函数

| CAN 函数             | 说 明            |
|--------------------|----------------|
| can_task_create    | 创建 CAN 总线通信任务  |
| can_hw_init        | CAN 控制器硬件初始化   |
| can_def_obj        | 定义通信目标         |
| can_start/can_stop | 启动/停止 CAN 总线通信 |
| can_send           | 通过 CAN 总线发送目标  |

续表

| CAN 函数         | 说 明                       |
|----------------|---------------------------|
| can_write      | 向目标中写入新数据但不发送             |
| can_read       | 直接读取目标数据                  |
| can_receive    | 接收所有非绑定目标                 |
| can_bind_obj   | 向任务中绑定一个目标, 当接收到目标时任务立即开始 |
| can_unbind_obj | 解除目标与任务之间绑定               |
| can_wait       | 等待接收一个绑定目标                |
| can_request    | 向指定目标发送一个远程帧              |
| can_get_status | 获得 CAN 总线控制器目前的实际状态       |

## 12.3 使用 RTX51 TINY 的要求和限定

### 1. 对开发工具的要求

在使用实时多任务操作系统 RTX51 TINY 时需要以下软件支持:

- Cx51 编译器
- BL51 连接定位器
- A51 宏汇编器

库文件 RTX51TINY.LIB 必须存储在目录 C51\LIB 下, 编译连接时必须指定 Cx51 运行库的路径。头文件 RTX51TINY.H 必须存储在目录 C51\INC 下, 必须指定 Cx51 包含文件的路径。

### 2. 对目标系统的要求

RTX51 TINY 可以在没有外部扩展数据存储器的单片 8051 系统中运行, 但应用程序可以访问外部存储器。RTX51 TINY 可以使用 Cx51 编译器支持的所有存储器模式, 存储器模式的选择仅影响应用程序目标文件的定位。RTX51 TINY 的系统变量以及应用程序的堆栈区总是位于 8051 的内部数据存储器中 (DATA/IDATA), 典型地, RTX51 TINY 应用程序应采用 SMALL 编译模式。

RTX51 TINY 仅支持循环任务切换, 不支持占先式任务切换和任务优先权。如果希望在应用程序中采用占先式任务切换, 则必须使用 RTX51 FULL。

RTX51 TINY 不支持代码分组程序, 如果希望在代码分组程序中采用实时多任务操作系统, 则必须使用 RTX51 FULL。

### 3. 中断处理

RTX51 TINY 可以并行处理中断函数。对于 8051 中断允许寄存器的操作必须遵从以下原则: 与其他 8051 应用程序一样, 必须使 EA=1 以便能够触发中断, RTX51 TINY 本身不包括任何中断管理。由于 RTX51 TINY 已经使用了 8051 内部定时器 0, 因此不允许禁止定时器 0 中断。



#### 4. 再入函数

非再入的 Cx51 函数不能同时被几个任务或中断服务程序所调用。非再入函数将它们的参数和局部数据存放在静态存储器段中，当同时出现多个函数调用时这些数据将发生覆盖。因此非再入函数只有在用户程序能保证不发生重复调用的情况下，才可以被几个任务所调用，这意味着必须禁止循环任务切换，并且这些非再入函数还不允许调用任何 RTX51 TINY 系统函数。

仅使用寄存器传递参数和自动变量的 C 语言函数是可再入的，这些函数可以被不同的 RTX51 TINY 任务所调用而没有任何限制。

Cx51 编译器还提供了其他再入函数，再入函数将它们的参数和局部数据存放在“再入栈”中，从而这些数据可以在重复调用中得到保护。但是 RTX51 TINY 不包括任何对“再入栈”的管理，因此在应用程序中当使用再入函数时，必须保证这些函数不调用任何 RTX51 TINY 系统函数，并且这些再入函数不能被 RTX51 TINY 的循环任务切换所中断。

RTX51 FULL 中包括了对再入函数“再入栈”的管理。

#### 5. Cx51 库函数的使用

Cx51 运行库中的所有可再入函数（详见第 9 章）都可以不加限制地在所有任务中使用。对于那些非再入函数，用户必须保证他们不能同时被几个任务所调用。

#### 6. 多数据指针及算术处理器的使用

Cx51 编译器 5.0 以上版本允许使用多数据指针以及算术处理器，由于 RTX51 TINY 不能管理与这些硬件有关的操作，如果能够保证在执行与这些硬件有关的操作时没有循环任务切换发生，用户程序可以使用多数据指针以及算术处理器，否则建议用户程序不要与 RTX51 TINY 同时使用这些功能。

#### 7. 寄存器组选择

RTX51 TINY 将所有的任务分配给寄存器组 0，因此所有的任务函数都必须按 Cx51 的默认设置 REGISTERBANK(0)来进行编译，中断函数可以使用余下的寄存器组。RTX51 TINY 需要使用寄存器组中 6 个固定的字节，该寄存器组可以通过配置文件 CONF\_TNY.A51 中的变量 INT\_REGBANK 来进行设置。

#### 8. 任务定义

实时多任务应用中包含若干个完成特殊功能的任务，RTX51 TINY 允许最多定义 16 个任务。RTX51 TINY 的任务定义格式如下：

```
void func(void) _task_num
```

其中：“func”为任务函数名，“\_task\_”是一个关键字，“num”是任务号，其值为 0~15，每个任务必须有一个惟一的任务号。注意，RTX51 TINY 的任务没有返回值和参数，因此定义时应将其明确定义为 void 类型。例如：

```
void job0 (void) _task_ 0 {
 while (1) {
 counter0++;
 /* 增量计数器 */
 }
```

```
}
}
```

## 12.4 RTX51 TINY 的任务管理

### 1. 任务状态

每个 RTX51 TINY 的任务都必须处在下列状态之一：

- **RUNNING** 任务正在运行中，每次只能有一个任务处于运行状态。
- **READY** 任务正在等待运行，当前运行的任务完成后，RTX51 TINY 启动下一个处于 **READY** 状态的任务。
- **WAITING** 任务正在等待一个事件，如果发生的事件是任务所等待的，则任务进入 **READY** 状态。
- **DELETED** 没有启动的任务都处于 **DELETED** 状态。
- **TIMEOUT** 由于循环“超时”而中断的任务处于 **TIMEOUT** 状态，该状态相当于 **READY**。

### 2. 事件

RTX51 TINY 的 `os_wait` 函数支持以下事件：

- **SIGNAL** 用于任务通信的位。**SIGNAL** 可以由 RTX51 TINY 的系统函数置位或清除。一个任务在运行之前可以等待 **SIGNAL** 置位。如果一个任务调用了 `os_wait` 函数等待 **SIGNAL** 而 **SIGNAL** 未置位，该任务将被挂起直到 **SIGNAL** 置位后才返回到 **READY** 状态并可被再次执行。
- **TIMEOUT** 由 `os_wait` 函数启动的时间延时。其持续时间由定时器节拍确定。以 **TIMEOUT** 值调用 `os_wait` 函数的任务将被挂起直到延时结束，然后返回到 **READY** 状态并可被再次执行。
- **INTERVAL** 由 `os_wait` 函数启动的间隔延时，间隔延时时间由定时器节拍确定。与 **TIMEOUT** 不同的是 RTX51 的定时器不复位，因此 **INTERVAL** 事件将使用一个持续运行的定时器。如果任务以同步间隔执行，就要用到 **INTERVAL** 事件，例如时钟。

### 3. 任务切换

RTX51 TINY 执行循环任务切换，允许“准并行”地执行多个无穷循环或任务。各个任务并非持续运行，而是在一段预先设定的时间片内运行。CPU 执行时间被划分为若干时间片，RTX51 为每个任务分配一个时间片，一个任务只允许在分配的时间片内执行，然后 RTX51 切换到另一个任务并允许它在其规定的时间片内执行。时间片的持续时间可以通过配置文件 `CONF_TNY.A51` 中的变量 `TIMESHARING` 来进行设置。

可以采用系统函数 `os_wait` 来通知 RTX51 TINY 允许另一个任务开始运行，而不必等待一个任务的时间片结束。函数 `os_wait` 将当前执行的任务挂起以等待一个特定事件的发生，在这段等待时间内可以运行任意数目的其他任务。

RTX51 TINY 给 CPU 分配任务的系统单元称为调度器, 它根据以下规则起作用。

- ① 如果出现以下情况, 当前运行的任务将被中断。
  - 任务调用了 `os_wait` 函数并且特定事件还没有发生。
  - 任务执行比循环切换所规定的时间长。
- ② 如果出现以下情况, 将启动另一个任务开始运行。
  - 没有其他任务正在运行。
  - 任务是处于“READY”或“TIMEOUT”状态下等待运行。

## 12.5 RTX51 TINY 的配置

用户可以对 RTX51 TINY 的配置文件 `CONF_TNY.A51` 中如下参数进行修改:

- 用于系统时钟中断的寄存器组
- 系统定时器的间隔
- 循环切换的 TIMEOUT 值
- 内部数据存储器的大小
- 启动 RTX51 TINY 后自由堆栈的大小

配置文件 `CONF_TNY.A51` 列表如下:

```
$DEBUGPUBLICS
;-----
; RTX-51 tiny 硬件定时器
; =====
;
; 利用以下 EQU 语句定义 RTX-51 tiny 硬件定时器的初始化
; (RTX-51 tiny 使用 8051 的定时器 0 作为控制 RTX-51 的软件定时器)
;
; 定义定时器中断所使用的寄存器组
INT_REGBANK EQU 1 ; 默认值为寄存器组 1
;
; 用 8051 机器周期定义硬件定时器溢出
INT_CLOCK EQU 10000 ; 默认值为 10000 个机器周期
;
; 用硬件定时器节拍定义循环切换时间到
TIMESHARING EQU 5 ; 默认值为 5 个节拍
;
;
; 注意: 用 0 作为 TIMESHARING 的值可以显示循环任务切换
;-----
; RTX-51 tiny 堆栈空间
; =====
;
```

```

; 以下 EQU 语句定义了用于堆栈区及其最小自由空间的片内 RAM 大小, 用一个宏
; 定义了堆栈空间耗尽时的执行代码
;
; 定义 CPU 堆栈的最高 RAM 地址
RAMTOP EQU 0FFH ; 默认值为地址 (256-1)
;
FREE_STACK EQU 20 ; 默认值为堆栈中 20 个字节的自由空间
;
STACK_ERROR MACRO
 CLR EA ; 禁止中断
 SJMP $; 如果堆栈耗尽则无穷循环
 ENDM
;
; -----
 NAME ?RTX51_TINY_CONFIG

PUBLIC ?RTX_REGISTERBANK, ?RTX_TIMESHARING, ?RTX_RAMTOP, ?RTX_CLOCK
PUBLIC ?RTX_ROBINTIME, ?RTX_SAVEACC, ?RTX_SAVEPSW
PUBLIC ?RTX_FREESTACK, ?RTX_STACKERROR, ?RTX_CURRENTTASK

?RTX_TIMESHARING EQU -TIMESHARING
?RTX_RAMTOP EQU RAMTOP
?RTX_FREESTACK EQU FREE_STACK
?RTX_CLOCK EQU -INT_CLOCK
?RTX_REGISTERBANK EQU INT_REGBANK * 8
 DSEG AT ?RTX_REGISTERBANK
 DS 2 ;temporary space
?RTX_SAVEACC: DS 1
?RTX_SAVEPSW: DS 1
?RTX_ROBINTIME: DS 1
?RTX_CURRENTTASK: DS 1

?RTX?CODE SEGMENT CODE
 RSEG ?RTX?CODE
?RTX_STACKERROR: STACK_ERROR

 END

```

配置文件中包括一些 EQU 伪指令, 用于系统参数的调整, 下面对这些伪指令的功能说明如下:

**INT\_REGBANK** 表示那个寄存器组被 RTX51 TINY 用作系统中断, 默认值为寄存器组 1。

**INT\_CLOCK** 定义系统时钟间隔。RTX51 TINY 的系统时钟在这个间隔内产生中

断。所定义的数值规定了每个中断的 CPU 时钟周期数。

- TIMESHARING** 定义循环任务切换的 TIMEOUT。所定义的数值表示 RTX51 TINY 在切换到另一个任务之前必须等待的定时器节拍中断数，如果 TIMESHARING 被定义为 0，则禁止循环任务切换。
- RAMTOP** 表示 RTX51 TINY 使用的 8051 单片机内部 RAM 的最高地址。对于 8051 来说 RAMTOP 应定义为 7FH，对于 8052 来说 RAMTOP 应定义为 FFH。
- FREE\_STACK** 以字节为单位规定自由栈区的大小。当进行任务切换时，RTX51 TINY 在栈区内检测所规定的字节数是否可用。如果栈区太小，RTX51 TINY 将启用宏 STACK\_ERROR。FREE\_STACK 值的范围为 0~0FFH，默认值为 20。
- STACK\_ERROR** 当 RTX51 TINY 检测到栈区出错时将启用该宏。用户可以修改该宏以适应用户应用程序的需要。

在创建 RTX51 应用程序时要遵守以下原则。

- 如果可能，应禁止循环多任务切换。使用多任务循环切换的任务要求有 13 个字节的堆栈空间来保存任务的前后关系（如工作寄存器等）。如果任务切换是由 os\_wait 函数触发，则不需要这个保存区。使用 os\_wait 函数后，等待执行的任务不需要等待整个循环超时的时间，从而使系统的响应时间得到改善。
- 不要将定时器节拍中断的速度设置得太快。可以将节拍速度设置为一个小数值，增加每秒的定时器节拍数。每个定时器中断的系统开销大约为 100~200 个 CPU 周期，因此定时器节拍速度应该设置得足够高，以减少中断延时。

## 12.6 RTX51 TINY 的系统函数

RTX51 TINY 为任务管理、任务通信以及其他服务提供系统函数，RTX51 TINY 的系统函数可以直接被 Cx51 调用。本节对所有 RTX51 TINY 的系统函数作详细说明，其中由“os\_”开头的系统函数可以被任务专用，由“isr\_”开头的系统函数可以被 Cx51 的中断函数专用。RTX51 TINY 系统函数的说明以及所有常数说明都包含在头文件“RTX51TINY.H”中，该头文件位于\Keil\C51\INC 目录下，必须在用户应用程序的开始处用#include 包含进来。

下面列出 RTX51 TINY 系统函数并作必要的解释。

函数原型：char isr\_send\_signal(unsigned char task\_id);

参 数：task\_id 为要发送信号的任务。

返回 值：0：信号成功发送，-1：指定的任务不存在。

功 能：向由 task\_id 确定的任务发送信号，如果该任务正在等待信号，则使任务再次就绪，否则信号被保存在所访问的任务信号标志内。isr\_send\_signal() 函数只能被中断函数所调用。

参 见：os\_clear\_signal, os\_send\_signal, os\_wait

例 子: 

```
#include <rtx51tny.h>
void tst_isr_send_signal (void) interrupt 2
{
:
 isr_send_signal (8); /* 向任务 8 发信号 */
:
}
```

函数原型: `char os_clear_signal(unsigned char task_id);`

参 数: `task_id` 为要清除信号标志的任务。

返 回 值: 0: 信号清除成功, -1: 指定的任务不存在。

功 能: 清除由 `task_id` 确定的任务的信号标志。

参 见: `isr_clear_signal`, `os_send_signal`

例 子: 

```
#include <rtx51tny.h>
#include <stdio.h>
void tst_os_clear_signal (void) _task_ 8
{
:
 os_clear_signal (5); /* 清除任务 5 中的信号标志 */
:
}
```

函数原型: `char os_create_task(unsigned char task_id);`

参 数: `task_id` 为要启动的任务, `task_id` 的值为 0~15。

返 回 值: 0: 信号成功启动, -1: 指定的任务不存在或不能启动指定的任务。

功 能: `os_create_task()` 启动一个由 `task_id` 指定的任务, 任务被标记为 **READY** 状态, 并根据任务切换规则执行任务。

参 见: `os_delete_task`

例 子: 

```
#include <rtx51tny.h>
#include <stdio.h>
void new_task (void) _task_ 2
{
:
}

void tst_os_create_task (void) _task_ 0
{
:
 if (os_create_task (2))
 {
 printf ("Couldn't start task 2\n");
 }
:
}
```

}

函数原型: `char os_delete_task(unsigned char task_id);`

参 数: `task_id` 为要删除的任务, `task_id` 的值为 0~15。只有原来用 “`os_create_task`” 说明的任务才可以删除 (在复位后, 除任务 0 之外), 运行中的任务也可以删除其自身。

返 回 值: 0: 任务被成功停止并删除, -1: 指定的任务不存在或没有启动。

功 能: `os_delete_task()` 函数停止一个由 `task_id` 指定的任务, 并将其从任务列表中删除。

参 见: `os_create_task`

例 子:

```
#include <rtx51tny.h>
#include <stdio.h>
void tst_os_delete_task (void) _task_ 0
{
:
if (os_delete_task (2))
{
printf ("Couldn't stop task 2\n");
}
:
}
```

函数原型: `char os_running_task_id(void);`

参 数: 无

返 回 值: 返回正在执行中的任务号 (0~15)。

功 能: `os_running_task_id()` 函数返回当前正在执行中的任务号 `task_id`。

参 见: `os_delete_task`, `os_delete_task`

例 子:

```
#include <rtx51tny.h>
#include <stdio.h>
void tst_os_running_task (void) _task_ 3
{
unsigned char tid;
tid = os_running_task_id ();
/* tid = 3 */
}
```

函数原型: `char os_send_signal(unsigned char task_id);`

参 数: `task_id` 为要发送信号的任务号 (0~15)。

返 回 值: 0: 信号成功发送, -1: 指定的任务不存在。

功 能: 向由 `task_id` 确定的任务发送信号, 如果该任务正在等待信号, 则使任务再次就绪, 否则信号被保存在所访问的任务信号标志内。`os_send_`

signal()函数只能被任务函数所调用。

参 见: isr\_send\_signal, os\_clear\_signal, os\_wait

例 子: #include <rtx51tiny.h>

#include <stdio.h>

void signal\_func (void) \_task\_ 2

{

;

os\_send\_signal (8); /\* 向任务 8 发信号 \*/

;

}

void tst\_os\_send\_signal (void) \_task\_ 8

{

;

os\_send\_signal (2); /\* 向任务 2 发信号 \*/

;

}

函数原型: char os\_wait (unsigned char event\_sel,  
                                unsigned char ticks,  
                                unsigned int dummy);

参 数: event\_sel 为要等待的事件, event\_sel 通过加入以下事件而形成参数:

K\_IVL          等待间隔

K\_SIG          等待信号

K\_TMO 等待超时

上述事件可单独使用, 也可以“相或”使用。例如:

event\_sel= K\_IVL

event\_sel= K\_TMO | K\_SIG

ticks 决定要等待完成超时 (K\_TMO) 或间隔 (K\_IVL) 所需的定时器节拍数, 如果 K\_IVL 或 K\_TMO 没有确定则该参数无意义。ticks 的允许值为 0~255 个定时器节拍数。

参数 dummy 用于对 RTX51 的兼容性, 它不用于 RTX51 TINY。

返 回 值: SIG\_EVENT: 接收到一个信号。

TMO\_EVENT: 超时或间隔已完成。

NOT\_OK: event\_sel 的值非法。

功 能: os\_wait 函数可以暂停当前任务而等待一个或几个事件, 诸如超时、间隔、另一个任务或中断的信号。在任务等待事件的时间内, 其他任务可以执行, 如果有一个指定的任务发生, 则调用的任务被置为就绪状态。

参 见: os\_wait1, os\_wait1

例 子: #include <rtx51tiny.h>

#include <stdio.h>



```

void tst_os_wait (void) _task_ 9
{
 while (1)
 {
 char event;
 event = os_wait (K_SIG + K_TMO, 50, 0);
 switch (event)
 {
 default:
 /* 这应永不发生 */
 break;
 case TMO_EVENT:
 /* 超时 */
 /* 50 个脉冲时间到 */
 break;
 case SIG_EVENT:
 /* 信号收到 */
 /* 信号收到处理 */
 break;
 }
 }
}

```

函数原型: char os\_wait1 (unsigned char event\_sel);

参 数: event\_sel 为要等待的事件, 它只能加入 K\_SIG 事件而形成参数。

返 回 值: SIG\_EVENT: 接收到信号。

NOT\_OK: event\_sel 的值非法。

功 能: os\_wait1 函数是 os\_wait 函数的一个子集, 它可以暂停当前任务而等待一个事件 (仅 K\_SIG 事件)。

参 见: os\_wait, os\_wait2

函数原型: char os\_wait2 (unsigned char event\_sel,  
unsigned char ticks);

参 数: event\_sel 为要等待的事件, event\_sel 通过加入以下事件而形成参数:

K\_IVL 等待间隔

K\_SIG 等待信号

K\_TMO 等待时间到

上述事件可单独使用, 也可以“相或”使用。例如:

event\_sel= K\_IVL

event\_sel= K\_TMO | K\_SIG

ticks 决定要等待完成时间到 (K\_TMO) 或间隔 (K\_IVL) 所需的计时数。

返 回 值: SIG\_EVENT: 信号被接收。

TMO\_EVENT: 时间到或间隔已完成。

- NOT\_OK: event\_sel 的值非法。
- 功 能: os\_wait2 函数可以暂停当前任务而等待一个或几个事件，诸如时间到、间隔、另一个任务或中断的信号。
- 参 见: os\_wait, os\_wait1
- 注: 对于程序代码要求苛刻的应用场合，建议在可能的情况下使用 os\_wait1 或 os\_wait2 函数，这样可以省略 1~2 个函数参数从而减少代码。

12.7 RTX51 TINY 应用系统调试

1. 堆栈管理

RTX51 TINY 为每个任务保留一个单独的栈区。由于 RTX51 TINY 仅使用内部数据存储器，所以全部堆栈管理都在 8051 的 IDATA 空间进行。为了给当前正在运行的任务分配尽可能大的栈区，那些未运行任务所使用的堆栈被移动。图 12.1 说明了每个任务的堆栈分配情况。

从图 12.1 中可以看到 RTX51 TINY 总是为当前正在运行的任务分配全部空闲存储器空间作为栈区，栈区从存储器的 ?STACK 段开始，该段中保留了内部数据存储器开头未分配的字节。

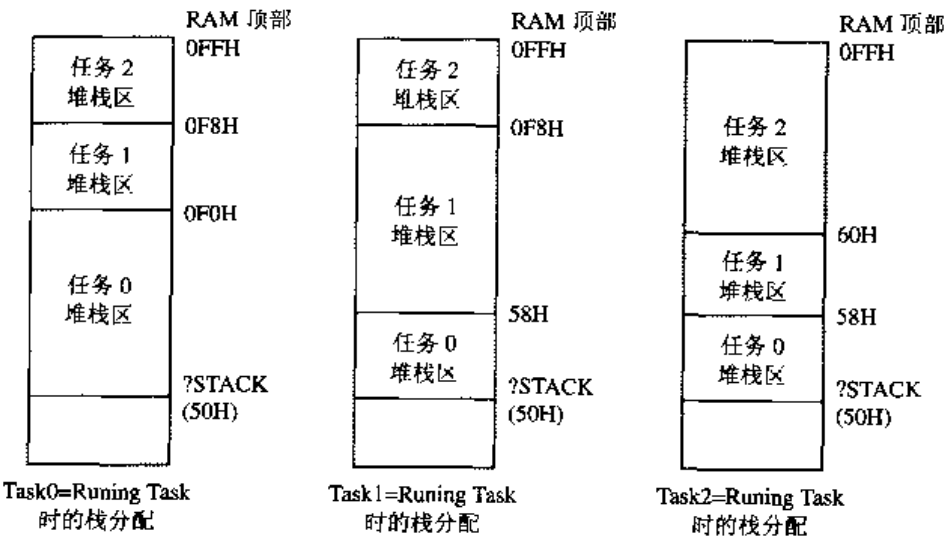


图 12.1 任务的堆栈分配

2. 应用举例

例 12.6: RTX51 TINY 应用，交通信号灯控制器。

这是一个应用 RTX51 TINY 实现的交通信号灯控制器，在用户规定的时间间隔之内“TRAFFIC”灯点亮，表示汽车等交通工具可以通行，一旦超过规定的时间间隔黄灯将闪烁。如果行人按下请求按钮，“TRAFFIC”灯立即变为“WALK”状态，表示行人穿过马路。这是一个典型的实时多任务系统，采用普通前后台编程方式不容易实现，而采用 RTX51 TINY 实时操作系统则可以获得很好的控制效果。整个应用由三部分程序组成：

TRAFFIC.C、SERIAL.C 和 GETLINE.C, 其中 TRAFFIC.C 是主要程序, 它完成如下任务:

- Task 0 串行口初始化, 启动所有其他任务。由于只需要进行一次初始化, 因此 Task 0 还将删除其自身。
- Task 1 交通信号灯命令处理器。该任务控制并处理从串行口接收的命令。
- Task 2 控制时钟。
- Task 3 在规定的通行时间间隔之外, 使黄灯闪烁。
- Task 4 在规定的通行时间间隔以内控制“TRAFFIC”灯打开, 允许交通工具通行。
- Task 5 读取行人按钮信号并将该信号发送到 Task 4。
- Task 6 检查是否有“Esc”键按下, 若有则终止前面的显示命令。

TRAFFIC.C 程序代码如下:

```

/*****
/* TRAFFIC.C: 使用 Cx51 编译器的交通信号灯控制器 */
*****/
code char menu[] =
 "\n"
 "+***** TRAFFIC LIGHT CONTROLLER using C51 and RTX-51 tiny ****+\n"
 "| This program is a simple Traffic Light Controller. Between |\n"
 "| start time and end time the system controls a traffic light |\n"
 "| with pedestrian self-service. Outside of this time range |\n"
 "| the yellow caution lamp is blinking. |\n"
 "+ command +- syntax -----+ function -----+ \n"
 "| Display | D | display times |\n"
 "| Time | T hh:mm:ss | set clock time |\n"
 "| Start | S hh:mm:ss | set start time |\n"
 "| End | E hh:mm:ss | set end time |\n"
 "+-----+-----+-----+ \n";

#include <reg52.h> /* 8052 单片机特殊功能寄存器定义 */
#include <rtx51tiny.h> /* RTX-51 tiny 功能函数定义 */
#include <stdio.h> /* 标准 I/O 头文件 */
#include <ctype.h> /* 字符函数头文件 */
#include <string.h> /* 字符串及存储器函数头文件 */

extern getline (char idata *, char); /* 外部函数: 行输入 */
extern serial_init (); /* 外部函数: 串行口初始化 */

#define INIT 0 /* 任务 0: 初始化 */
#define COMMAND 1 /* 任务 1: 命令 */
#define CLOCK 2 /* 任务 2: 时钟 */
#define BLINKING 3 /* 任务 3: 闪烁 */
#define LIGHTS 4 /* 任务 4: 信号 */

```

```

#define KEYREAD 5 /* 任务 5: 读键 */
#define GET_ESC 6 /* 任务 6: Esc 键退出 */

struct time { /* 定义时间记录结构 */
 unsigned char hour; /* 小时 */
 unsigned char min; /* 分 */
 unsigned char sec; /* 秒 */
};

struct time ctime = { 12, 0, 0 }; /* 存储时钟的时间值 */
struct time start = { 7, 30, 0 }; /* 存储开始时间值 */
struct time end = { 18, 30, 0 }; /* 存储结束时间值 */

sbit red = P1^2; /* I/O 脚: 红灯输出 */
sbit yellow = P1^1; /* I/O 脚: 黄灯输出 */
sbit green = P1^0; /* I/O 脚: 绿灯输出 */
sbit stop = P1^3; /* I/O 脚: stop 灯输出 */
sbit walk = P1^4; /* I/O 脚: walk 灯输出 */
sbit key = P1^5; /* I/O 脚: 自助按键输入 */

idata char inline[16]; /* 存储输入命令行 */

/*****
/* 任务 0: 初始化 */
*****/
init () _task_ INIT { /* 程序从这里开始运行 */
 serial_init (); /* 初始化串行口 */
 os_create_task (CLOCK); /* 启动时钟任务 */
 os_create_task (COMMAND); /* 启动命令任务 */
 os_create_task (LIGHTS); /* 启动信号任务 */
 os_create_task (KEYREAD); /* 启动读键任务 */
 os_delete_task (INIT); /* 停止初始化任务 */
}

bit display_time = 0; /* 标志: 信号命令状态显示时间 */

/*****
/* 任务 2: 时钟 */
*****/
clock () _task_ CLOCK {
 while (1) { /* 时钟为无限循环 */
 if (++ctime.sec == 60) { /* 计算秒值 */
 ctime.sec = 0;

```

```

 if (++ctime.min == 60) { /* 计算分值 */
 ctime.min = 0;
 if (++ctime.hour == 24) { /* 计算小时值 */
 ctime.hour = 0;
 }
 }
 }
 if (display_time) { /* 如果命令状态为延时 */
 os_send_signal (COMMAND); /* 向任务 1 发送信号 */
 }
 os_wait (K_IVL, 100, 0); /* 等待时间间隔: 1 秒 */
}

struct time rtime; /* 暂存输入时间 */

/*****
/* 读取时间: 将行输入转换为时间值并存储
*****/
bit readtime (char idata *buffer) {
 unsigned char args;

 rtime.sec = 0; /* 预置秒值为 0 */
 args = sscanf (buffer, "%bd:%bd:%bd", /* 扫描行输入 */
 &rtime.hour, /* 得到小时、分、秒值 */
 &rtime.min,
 &rtime.sec);

 if (rtime.hour > 23 || rtime.min > 59 || /* 检查输入值是否有效 */
 rtime.sec > 59 || args < 2 || args == EOF) {
 printf ("\n*** ERROR: INVALID TIME FORMAT\n");
 return (0);
 }
 return (1);
}

#define ESC 0x1B /* 定义 Esc 键代码 */

bit escape; /* Esc 键输入标志 */

/*****
/* 任务 6: 检查是否 Esc 键输入
*****/

```

```

get_escape () _task_ GET_ESC {
 while (1) { /* 无限循环 */
 if (_getkey () == ESC) escape = 1; /* 如果 Esc 键输入, 设置标志 */
 if (escape) { /* 如果有 Esc 键标志, 发送信号 */
 os_send_signal (COMMAND); /* 到任务 1 */
 }
 }
}

/*****
/* 任务 1: 命令处理器
*****/
command () _task_ COMMAND {
 unsigned char i;

 printf (menu); / * 显示命令菜单 */
 while (1) {
 printf ("\nCommand: "); /* 显示提示符 */
 getline (&inline, sizeof (inline)); /* 获得命令行输入 */

 for (i = 0; inline[i] != 0; i++) { /* 转换为大写 */
 inline[i] = toupper(inline[i]);
 }

 for (i = 0; inline[i] == ' '; i++); /* 跳过空格 */

 switch (inline[i]) { /* 命令功能处理 */
 case 'D': /* 显示时间命令 */
 printf ("Start Time: %02bd:%02bd:%02bd\n",
 "End Time: %02bd:%02bd:%02bd\n",
 start.hour, start.min, start.sec,
 end.hour, end.min, end.sec);
 printf ("type ESC to abort\r");

 os_create_task (GET_ESC); /* 在显示循环中检查 Esc 键 */
 escape = 0; /* 清除 Esc 标志 */
 display_time = 1; /* 置位显示时间标志 */
 os_clear_signal (COMMAND); /* 清零悬挂标志 */

 while (!escape) { /* 当无 Esc 键输入时 */
 printf ("Clock Time: %02bd:%02bd:%02bd\r", /* 显示时间 */
 ctime.hour, ctime.min, ctime.sec);
 os_wait (K_SIG, 0, 0); /* 等待时间变化或 Esc 键输入 */
 }
 }
 }
 }
}

```

```

 }

 os_delete_task (GET_ESC); /* 不再检查 Esc 键 */
 display_time = 0; /* 清零显示时间标志 */
 printf ("\n\n");
 break;

case 'T': /* 设置时间命令 */
 if (readtime (&inline[i+1])) { /* 读取输入时间 */
 ctime.hour = rtime.hour; /* 并存储于 ctime 中 */
 ctime.min = rtime.min;
 ctime.sec = rtime.sec;
 }
 break;

case 'E': /* 设置结束时间命令 */
 if (readtime (&inline[i+1])) { /* 读取输入时间 */
 end.hour = rtime.hour; /* 并存储于 end 中 */
 end.min = rtime.min;
 end.sec = rtime.sec;
 }
 break;

case 'S': /* 设置开始时间命令 */
 if (readtime (&inline[i+1])) { /* 读取输入时间 */
 start.hour = rtime.hour; /* 并存储 start 中 */
 start.min = rtime.min;
 start.sec = rtime.sec;
 }
 break;

default: /* 错误处理 */
 printf (menu); /* 显示命令菜单 */
 break;
}
}
}

/*****
/* 信号函数: 检查时钟时间是否在 start 与 end 之间
*****/
bit signalon () {
 if (memcmp (&start, &end, sizeof (struct time)) < 0) {

```

```

 if (memcmp (&start, &ctime, sizeof (struct time)) < 0 &&
 memcmp (&ctime, &end, sizeof (struct time)) < 0) return (1);
}

else {
 if (memcmp (&end, &ctime, sizeof (start)) > 0 &&
 memcmp (&ctime, &start, sizeof (start)) > 0) return (1);
}
return (0); /* 信号结束, 开始闪烁 */
}

/*****
/* 任务 3 闪烁: 如果当前时间位于 start 和 end 之外则启动本任务 */
*****/
blinking () _task_ BLINKING { /* 闪烁黄灯 */
 red = 0; /* 熄灭所有灯 */
 yellow = 0;
 green = 0;
 stop = 0;
 walk = 0;

 while (1) {
 yellow = 1; /* 黄灯亮 */
 os_wait (K_TMO, 30, 0); /* 等待超时: 30 个时钟节拍 */
 yellow = 0; /* 黄灯灭 */
 os_wait (K_TMO, 30, 0); /* 等待超时: 30 个时钟节拍 */
 if (signalon ()) { /* 如果闪烁时间到 */
 os_create_task (LIGHTS); /* 启动任务 4: 点亮灯 */
 os_delete_task (BLINKING); /* 并停止任务 3: 停止闪烁 */
 }
 }
}

/*****
/* 任务 4 信号: 如果当前时间位于 start 和 end 之间则启动本任务 */
*****/
lights () _task_ LIGHTS {
 red = 1; /* 点亮红灯和 stop 灯 */
 yellow = 0;
 green = 0;
 stop = 1;
 walk = 0;
 while (1) {

```



```

 os_wait (K_TMO, 30, 0); /* 等待超时: 30 个时钟节拍 */
 if (!signalon ()) { /* 如果信号时间到 */
 os_create_task (BLINKING); /* 启动任务 3: 开始闪烁 */
 os_delete_task (LIGHTS); /* 停止任务 4: 停止亮灯 */
 }
 yellow = 1;
 os_wait (K_TMO, 30, 0); /* 等待超时: 30 个时钟节拍 */
 red = 0; /* 为汽车点亮绿灯 */
 yellow = 0;
 green = 1;
 os_clear_signal (LIGHTS);
 os_wait (K_TMO, 30, 0); /* 等待超时: 30 个时钟节拍 */
 os_wait (K_TMO + K_SIG, 250, 0); /* 等待超时和信号 */
 yellow = 1;
 green = 0;
 os_wait (K_TMO, 30, 0); /* 等待超时: 30 个时钟节拍 */
 red = 1; /* 为汽车点亮红灯 */
 yellow = 0;
 os_wait (K_TMO, 30, 0); /* 等待超时: 30 个时钟节拍 */
 stop = 0; /* 为行人点亮绿灯 */
 walk = 1;
 os_wait (K_TMO, 100, 0); /* 等待超时: 100 个时钟节拍 */
 stop = 1; /* 为行人点亮红灯 */
 walk = 0;
}
}

/*****
/* 任务 5 按键输入: 处理行人按键输入
*****/
keyread () _task_ KEYREAD {
 while (1) {
 if (key) { /* 若有键按下 */
 os_send_signal (LIGHTS); /* 向任务 4 发送信号 */
 }
 os_wait (K_TMO, 2, 0); /* 等待超时: 2 个时钟节拍 */
 }
}

```

SERIAL.C 是一个串行中断驱动程序, 程序中包含了两个基本函数: putchar 和 getkey, 输入输出 I/O 函数 printf 和 getline 需要调用这两个基本函数。用户也可以根据自己的需要另外编写相应的输入输出驱动程序。

SERIAL.C 程序代码如下:

```

/*****
/* SERIAL.C: 应用 RTX-51 tiny 的串行口中断控制器 */
/*****
#include <reg52.h> /* 8052 单片机特殊功能寄存器定义 */
#include <rtx51tiny.h> /* RTX-51 tiny 功能函数定义 */

#define OLEN 8 /* 定义串行口通信发送缓冲区的大小 */
unsigned char ostart; /* 串行口发送缓冲区起点 */
unsigned char oend; /* 串行口发送缓冲区终点 */
idata char outbuf[OLEN]; /* 串行口发送缓冲区 */
unsigned char otask = 0xff; /* 输出任务的任务号 */

#define ILEN 8 /* 定义串行口通信接收缓冲区大小 */
unsigned char istart; /* 串行口接收缓冲区起点 */
unsigned char iend; /* 串行口接收缓冲区终点 */
idata char inbuf[ILEN]; /* 串行口接收缓冲区 */
unsigned char itask = 0xff; /* 输入任务的任务号 */

#define CTRL_Q 0x11 /* 定义 CTRL_Q 代码 */
#define CTRL_S 0x13 /* 定义 CTRL_S 代码 */

bit sendfull; /* 发送缓冲区满标志 */
bit sendactive; /* 发送激活标志 */
bit sendstop; /* XOFF 字符标志 */

/*****
/* putbuf: 向 SBUF 或发送缓冲区写入字符函数 */
/*****
putbuf (char c) {
 if (!sendfull) { /* 仅当缓冲区未滿时发送 */
 if (!sendactive && !sendstop) { /* 如果发送器未处于活动态: */
 sendactive = 1; /* 直接发送第一个字符 */
 SBUF = c; /* 到 SBUF, 启动发送 */
 }
 else { /* 否则: */
 outbuf[oend++ & (OLEN-1)] = c; /* 向发送缓冲区传送字符 */
 if (((oend ^ ostart) & (OLEN-1)) == 0) sendfull = 1;
 }
 /* 置位缓冲区满标志 */
 }
}

/*****
/* putchar: 中断控制器的 putchar 函数 */
/****

```

```

/*****
char putchar (char c) {
 if (c == '\n') { /* 扩展一行新字符: */
 while (sendfull) { /* 等待发送缓冲区空 */
 otask = os_running_task_id (); /* 置位输出任务号 */
 os_wait (K_SIG, 0, 0); /* RTX-51 调用: 等待信号 */
 otask = 0xff; /* 清零输出任务号 */
 }
 putbuf (0x0D); /* 在换行符 LF 之前发送回车符 CR */
 }
 while (sendfull) { /* 等待发送缓冲区空 */
 otask = os_running_task_id (); /* 置位输出任务号 */
 os_wait (K_SIG, 0, 0); /* RTX-51 调用: 等待信号 */
 otask = 0xff; /* 清零输出任务号 */
 }
 putbuf (c); /* 发送字符 */
 return (c); /* 返回字符 */
}

```

```

/*****
/* _getkey: 中断控制器函数_getkey */
/*****
char _getkey (void) {
 while (iend == istart) {
 itask = os_running_task_id (); /* 置位输入任务号 */
 os_wait (K_SIG, 0, 0); /* RTX-51 调用: 等待信号 */
 itask = 0xff; /* 清零输入任务号 */
 }
 return (inbuf[istart++ & (ILEN-1)]);
}

```

```

/*****
/* serial: 串行接收 / 发送中断 */
/*****
serial () interrupt 4 using 2 { /* 中断使用工作寄存器 2 区 */
 unsigned char c;
 bit start_trans = 0;

 if (RI) { /* 如果是接收中断 */
 c = SBUF; /* 读字符 */
 RI = 0; /* 清零中断请求标志 */
 switch (c) { /* 字符处理 */
 case CTRL_S:

```

```

 sendstop = 1; /* 如果是 CTRL_S 则停止发送 */
 break;

 case CTRL_Q:
 start_trans = sendstop; /* 如果是 CTRL_Q 则开始发送 */
 sendstop = 0;
 break;

 default:
 /* 将其他字符读入输入缓冲区 */
 if (istart + ILEN != iend) {
 inbuf[iend++ & (ILEN-1)] = c;
 }

 /* 如果是任务等待: 信号就绪 */
 if (itask != 0xFF) isr_send_signal (itask);
 break;
 }
}

if (TI || start_trans) { /* 如果是发送中断 */
 TI = 0; /* 清零中断请求标志 */
 if (ostart != oend) { /* 如果输入缓冲区接收到字符 */
 if (!sendstop) { /* 且不是 CTRL_S */
 SBUF = outbuf[ostart++ & (OLEN-1)]; /* 发送字符 */
 sendfull = 0; /* 清零 sendfull 标志 */
 /* 如果是任务等待: 信号就绪 */
 if (otask != 0xFF) isr_send_signal (otask);
 }
 }
 else sendactive = 0; /* 如果全部发送完则清零 sendactive */
}

}

/*****
/* serial_init: 串行口初始化
*****/
serial_init () {
 SCON = 0x50; /* 串行口方式 1, 8 位 UART, 允许接收 */
 TMOD |= 0x20; /* 定时器方式 2, 8 位自动重装 */
 TH1 = 0xf3; /* 2400 波特率 */
 TR1 = 1; /* 启动定时器 1 */
 ES = 1; /* 允许串行口中断 */
}

```

GETLINE.C 是一个命令行编辑器，用于处理从串行口接收的字符。

GETLINE.C 程序代码如下：

```

/*****
/* GETLINE.C: 字符输入行编辑器 */
*****/
#include <stdio.h>

#define CNTLQ 0x11
#define CNTLS 0x13
#define DEL 0x7F
#define BACKSPACE 0x08
#define CR 0x0D
#define LF 0x0A

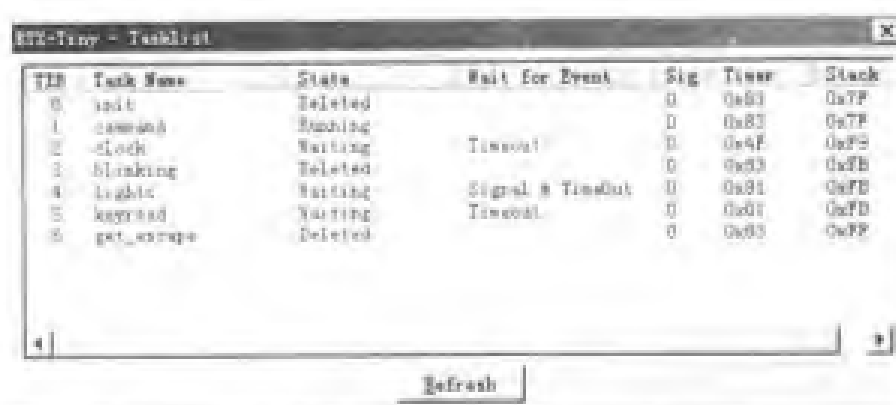
/*****
/* 行编辑器 */
*****/
void getline (char idata *line, unsigned char n) {
 unsigned char cnt = 0;
 char c;

 do {
 if ((c = _getkey ()) == CR) c = LF; /* 读入字符 */
 if (c == BACKSPACE || c == DEL) { /* 处理 BACKSPACE 键 */
 if (cnt != 0) {
 cnt--;
 line--;
 putchar (0x08); /* 回显 backspace */
 putchar (' ');
 putchar (0x08);
 }
 }
 else if (c != CNTLQ && c != CNTLS) { /* 忽略 CNTL_Q 和 CNTL_S */
 putchar (*line = c); /* 回显并保存字符 */
 line++;
 cnt++;
 }
 } while (cnt < n - 1 && c != LF);
 line = 0; /* 字符串结尾标志 */
}

```

### 3. 系统调试

RTX51 TINY 应用程序可以采用与标准 8051 应用程序相同的方法进行调试。在 Windows 集成开发环境  $\mu$ Vision2 中, 将上述三个程序文件添加到一个项目中, “Project 菜单/Options 选项/Target 标签页/Operating 栏” 选取 “RTX-51 Tiny”, 进行整体创建 (Build Target) 完成编译连接后, 即可进入调试状态进行仿真调试。选定 “RTX-51 Tiny” 操作系统后,  $\mu$ Vision2 将自动允许一个附加的调试功能, 进入调试状态之后会在 “Peripherals” 下拉菜单中增加一个 “Rtx-Tiny Tasklist” 选项, 单击该选项弹出如图 12.2 所示当前任务状态窗口。



| TID | Task Name  | State   | Wait for Event   | Sig | Timer | Stack |
|-----|------------|---------|------------------|-----|-------|-------|
| 0   | init       | Deleted |                  | 0   | 0x03  | 0x7F  |
| 1   | camera     | Running |                  | 0   | 0x83  | 0x7F  |
| 2   | clock      | Waiting | Timeout          | 0   | 0x4F  | 0x7F  |
| 3   | blinking   | Deleted |                  | 0   | 0x03  | 0x7F  |
| 4   | light      | Waiting | Signal & Timeout | 0   | 0x01  | 0x7F  |
| 5   | keyboard   | Waiting | Timeout          | 0   | 0x01  | 0x7F  |
| 6   | get_escape | Deleted |                  | 0   | 0x03  | 0x7F  |

Refresh

图 12.2 任务状态列表

窗口中各个输出参数的意义如下:

**TID** 采用关键字 “\_task\_” 进行任务定义的任务数

**Task Name** 任务函数名

**State** 表示任务状态, 参见 12.4 节。

**Wait for Event** 任务正在等待的事件。当用 K\_TMO 或 K\_IVL 参数调用 os\_wait 函数时将显示超时 (timeout) 事件, 表示任务正处于等待状态直到定时器减到 0 为止。当用 K\_SIG 参数调用 os\_wait 函数时将显示信号 (Signal) 事件, 表示任务正处于等待状态直到信号标志置 1 为止。

**Sig** 分配给该任务的信号标志状态, 1 为置位, 0 为复位。

**Timer** 表示产生超时 (timeout) 的定时节拍数。定时器是一直运行的, 只有当用 K\_TMO 参数调用 os\_wait 函数时才被置为超时 (timeout) 值。

**Stack** 表示局部任务在 IDATA 空间内的堆栈起始地址。

另外, 为了调试方便还定义了一个模拟行人按键的信号函数 Traffic.inc, 代码如下:

```
PORT1 <= 0x20; /* set P1.5 to zero: Key Input */
/* define a debug function for the pedestrian push button */
signal void push_key (void) {
 PORT1 |= 0x20; /* set P3.0 */
 switch ("Clock"0.05); /* wait 50 msec */
 PORT1 <= 0x20; /* reset P3.0 */
}
```

```

}
/* define a toolbar button to call push_key */
#define button "Push for Walk", "push_key"
RADIX=10 // decimal output in watch window

```

在“Options 选项/Debug 标签页/Initialization 栏”键入信号函数名，启动模拟调试时将自动加载该信号函数，单击“View 菜单/Toolbox 选项”弹出如图 12.3 所示窗口，可以看到其中增加了一个“Push for Walk”用于模拟行人过街的按钮。在命令窗口执行“Go”命令后，从观察窗口可以看到表示红绿灯的“red”、“green”、“yellow”，以及表示行人过街的“stop”、“walk”状态会随着不同任务的执行而不断变化，如图 12.4 所示。



图 12.3 模拟行人过街按钮

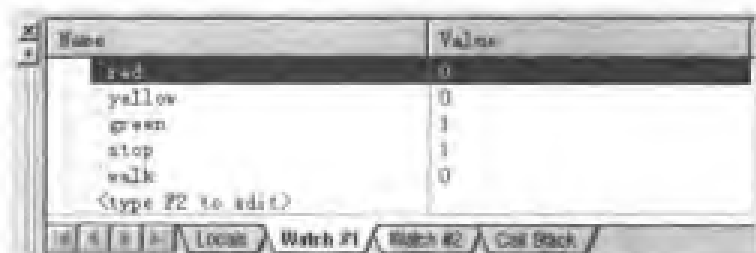


图 12.4 交通信号灯状态

串行窗口支持人机对话功能，本应用程序支持如下命令：

- D 显示时钟、启动和结束时间。
- T 采用 24 小时格式设置当前时间。
- S 采用 24 小时格式设置起始时间。交通灯控制器通常在启动与结束时间之间运行，在这段时间之外，黄灯闪烁。
- E 采用 24 小时格式设置结束时间。

图 12.5 所示为从串行窗口输入 D 命令后的显示结果。



图 12.5 串行窗口的显示状态

以上介绍的是全速运行程序的结果，实际上可以对应用程序设置断点、单步运行等，

可以更清楚地了解整个 RTX51 TINY 应用程序的运行状态。

### 3. RTX51 TINY 应用优化

在应用 RTX51 TINY 时应注意以下两点。

① 尽可能不使用循环任务切换。使用循环任务切换时要求有 13 个字节的堆栈区来保存任务内容（工作寄存器等），如果由 `os_wait` 函数来进行任务触发，则不需要保存任务内容。由于正处于等待运行的任务并不需要等待全部循环切换时间结束，因此 `os_wait` 函数可以产生一种改进的系统反应时间。

② 不要将时钟节拍中断速率设置得太快，设定为一个较低的数值可以增加每秒的时钟节拍个数。每次时钟节拍中断大约需要 100~200 个 CPU 周期，因此应将时钟节拍率设定得足够高以便使中断反应时间达到最小化。



# 附录 A Keil Cx51 与 ANSI C 的差别

由于 Cx51 编译器是专为 8051 单片机设计的，根据单片机自身的特点进行了若干扩展，因此 Keil Cx51 与 ANSI C 在编译器和库函数方面存在少许差别。

## 1. 编译器方面的差别

- 宽字符

Cx51 编译器不支持 16 位宽度的字符，而 ANSI 标准提供对宽字符国际字符集的支持。

- 函数递归调用

Cx51 的默认值不支持函数递归调用，需要进行递归调用的函数必须声明为再入函数。再入函数的局部数据和参数被放入再入栈中，从而允许进行递归调用。未使用再入属性的函数其局部数据被放入静态存储器段，对非再入函数进行递归调用时，上次调用所使用的局部数据被覆盖。

## 2. 库函数方面的差别

ANSI C 标准提供大量的库函数，Cx51 编译器支持其中大部分函数，但是一些不适用于嵌入式系统应用的库函数则没有包含到 Cx51 编译器中。

下列 ANSI 标准库函数包含在 Cx51 运行库中：

**abs acos asin atan atan2 atof atoi atol calloc ceil cos cosh exp  
fabs floor fmod free getchar gets isalnum isalpha iscntrl isdigit  
isgraph islower isprint ispunct isspace isupper isxdigit labs log log10  
longjmp malloc memchr memcmp memcpy memmove memset modf  
pow printf putchar puts rand realloc scanf setjmp sin sinh sprintf  
sqrt srand sscanf strcat strchr strcmp strcpy strcspn strlen strncat  
strncmp strncpy strpbrk strrchr strspn strtod strtol strtoul tan tanh  
tolower toupper va\_arg va\_end va\_start vprintf vsprintf**

下列 ANSI 标准库函数不包含在 Cx51 运行库中：

**abort asctime atexit bsearch clearerr clock ctime difftime div exit  
fclose feof ferror fflush fgetc fgetpos fgets fopen fprintf fputc fputs  
fread freopen frexp fscanf fseek fsetpos ftell fwrite gete getenv  
gmtime ldexp ldiv localeconv localtime mblen mbstowcs mbtowc  
mktime perror putc qsort raise remove rename rewind setbuf  
setlocale setvbuf signal strcoll strerror strftime strstr strtok strxfrm  
system time tmpfile tmpnam ungetc vprintf wctomb**

下列是 Cx51 扩充的非 ANSI 标准库函数:

**acos517 asin517 atan517 atof517 strtod517 cabs \_chkfloat\_ cos517  
\_crol\_ \_cror\_ exp517 \_getkey init\_mempool \_lrol\_ \_lror\_ log10517  
log517 \_lrol\_ \_lror\_ memcpy \_nop\_ printf517 scanf517 sin517  
sprintf517 sqrt517 sscanf517 strpos strpbrk strrpos tan517 \_testbit\_  
toascii toint \_tolower \_toupper ungetchar**

## 附录 B Keil Cx51 不同版本的差别

Keil Cx51 V7.0 以上版本与早期版本相比较在功能和编译效率方面有许多改进，因而不同版本之间存在一些差别。

### 1. 与 V6.0 版本的差别

- 去除外部符号和段的限制

每个模块中外部和段的数目不再局限于 256 个。这个限制是由旧的 INTEL 目标文件格式造成的。

- 变量名可以有 256 个字符

现在一个变量名可以有 256 个字符，以前版本只允许有 32 个字符。

- 支持 PHILIPS 80C51MX 和 DALLAS 的邻接模式

Cx51 编译器提供对 PHILIPS 80C51MX 体系结构和 DALLAS 390 及其派生产品邻接模式的支持。

- 支持 OMF2 命令和 far 存储类型

可以采用 OMF2 命令选择一个新的 OMF 文件格式，该格式提供对模块间符号类型的详细检查，并支持多达 16MB 的扩展 CODE 空间和 XDATA 空间。当采用 STRING、VARBANKING 和 XCROM 命令时要求使用这种格式。

- STRING 命令

Cx51 编译器允许用户将常数字符串定位在 const xdata 或 const far 空间，从而为程序保留更多的代码空间。

- USERCLASS 命令

对编译器生成的段分配用户定义类名。Lx51 连接定位器可以引用用户定义类名，用指定类名进行段定位。

- VARBANKING 命令和 far 存储类型支持

使用两种新的存储类 far 和 const far 以及用户可配置的访问程序，可以对多达 16MB 的扩展 CODE 空间和 XDATA 空间进行访问。采用 VARBANKING 命令后，允许编译器对 far 存储类型的支持。

- XCROM 命令

XCROM 命令可以将常数定位到 xdata ROM 中，从而为程序代码节省 CODE ROM 空间。

- 支持 ANALOG DEVICE 公司 B2 系列微转换器

ANALOG DEVICE 公司生产的 AD $\mu$ C B2 系列微转换器包含双 DPTR 和一个扩展的堆栈。

注：只有 PK51 专业开发工具软件包支持 OMF2 输出文件格式、PHILIPS 80C51MX 体系结构、DALLAS 产品的邻接模式以及 VARBANKING 命令，CA51 和 DK51 软件包不支持这些功能。

## 2. 与 V5.0 版本的差别

- 化级别 7, 8 和 9

Cx51 编译器提供三个新的优化级别，这些新的优化主要集中于代码长度上。

- 支持双 DPTR 的命令

Cx51 编译器提供了 MODA2 和 MODP2 命令，对 ATMEL、ATMEL WM 以及 PHILIPS 双 DPTR 的支持。

- 所有存储模式中的 data、pdata 和 xdata 自动变量覆盖

无论采用什么存储模式，Cx51 编译器现在都可覆盖所有的 data、pdata 和 xdata 自动变量。而以前版本的 C51 编译器只有默认存储模式下的自动变量才是可覆盖的。例如 V5.0 版本的 C51 编译器在 SMALL 存储模式下不支持对 pdata 或 xdata 变量的覆盖。

- 枚举类型可自动调整为 8 位或 16 位

只要在枚举类型允许的范围内，Cx51 编译器将采用字符型变量来表示枚举类型。

- 增加了 modf、strtod、strtol 和 strtoul 库函数

Cx51 编译器增加了 ANSI C 标准库函数 modf、strtod、strtol 和 strtoul。

- BROWSE、INCDIR、ONEREBANK、RET\_XSTK 和 RET\_PSTK 控制命令

Cx51 编译器提供以上新控制命令来产生浏览信息、指定包含路径、优化中断代码以及为返回地址使用再入堆栈。

## 3. 与 V4.0 版本的差别

- 浮点数的存储字节顺序

V5.0 以上版本的 C51 编译器对于四字节浮点类型数据(32 位二进制数)在内存中的存放格式不同于以前版本，具体格式如下：

| 字节地址  | +0        | +1        | +2       | +3       |
|-------|-----------|-----------|----------|----------|
| 浮点数内容 | S EEEEEEE | E MMMMMMM | MMMMMMMM | MMMMMMMM |

其中，S 为符号位，“0”表示正，“1”表示负。E 为阶码，占用 8 位二进制数，存放在两个字节中。阶码 E 的正常取值范围是 1~254。M 为尾数的小数部分，用 23 位二进制数表示，存放在三个字节中。尾数的整数部分永远为 1，因此不予保存，但它是隐含存在的。小数点位于隐含的整数位“1”的后面。例如浮点数  $-12.5 \approx 0xC1480000$ ，在内存中的存放格式为：

| 字节地址  | +0       | +1       | +2       | +3       |
|-------|----------|----------|----------|----------|
| 浮点数内容 | 11000001 | 01001000 | 00000000 | 00000000 |

- \_chkfloat\_ 库函数

提供了 \_chkfloat\_ 库函数，它可用于对浮点数的非正常值 (NaN、 $\pm INF$ )、0 值以及正常数值进行快速测试。

- **FLOATFUZZY 控制命令**

提供了 **FLOATFUZZY** 控制命令, 该命令用于控制在进行浮点数比较运算时忽略的位数。

- **浮点数算术运算为全再入处理**

对于浮点数算术运算操作(加、减、乘、除及比较)为全再入处理, 因此在中断函数中进行浮点数运算时不再需要库函数 **fpsave** 和 **fprestore** 来保存浮点数。

- **长整型和浮点类型数据操作不再使用模拟算术堆栈**

对于长整型和浮点类型数据的算术运算操作的效率更高, 生成完全基于寄存器的代码, 从而不再使用模拟算术堆栈, 这同时也减小代码的存储器空间。

- **存储器类型**

对存储器类型进行了改进以便使运行库能够更好地发挥作用, 同时能够反映 **MCS251** 的存储器映像结构。

- **一般指针的存储器类型编码**

**V5.0** 和 **V4.0** 版本的 **C51** 编译器对于一般指针的存储器类型编码不同, 如下表所示:

| 存储器类型     | idata | data | bdata | xdata | pdata | Code |
|-----------|-------|------|-------|-------|-------|------|
| Cx51 V5.0 | 0x00  | 0x00 | 0x00  | 0x01  | 0xFE  | 0xFF |
| Cx51 V4.0 | 0x01  | 0x04 | 0x04  | 0x02  | 0x03  | 0x05 |

- **WARNINGLEVEL 控制命令**

提供了 **WARNINGLEVEL** 控制命令, 该命令使用户可以规定编译器警告错误的检测强度。**Cx51** 编译器对于未使用的局部变量、标号以及表达式都将进行检查。

#### 4. 与 **V3.4** 版本的差别

- **\_at\_ 关键字**

**Cx51** 编译器支持采用关键字 **\_at\_** 对变量进行定位。**\_at\_** 关键字允许用户在声明一个变量时指定其地址。

- **NOAMAKE 控制命令**

**Cx51** 编译器支持 **NOAMAKE** 控制命令。该命令使 **Cx51** 所生成的目标文件中不包含项目(**project**)信息和寄存器优化记录。这只有在老版本 **Cx51** 编译工具中使用目标文件时才是必要的。

- **OH51 HEX 文件转换程序**

以前版本中的 **Object-HEX** 符号转换程序 **OHS51** 已被新转换程序 **OH51** 所取代。

- **优化级别 6**

**Cx51** 编译器支持关于回路循环的优化级别 6, 其结果代码效率更高, 执行速度更快。

- **ORDER 控制命令**

使用 **ORDER** 控制命令, **Cx51** 编译器将按照变量在源程序文件中的顺序进行存储器定位。

- **REGFILE 控制命令**

**Cx51** 编译器支持 **REGFILE** 控制命令, 该命令使用户可以指定由连接定位器生成的寄

寄存器定义文件的文件名。

- `vprintf` 和 `vsprintf` 库函数  
加入了 `vprintf` 和 `vsprintf` 库函数。

## 5. 与 V3.2 版本的差别

- ANSI 标准自动整型数提升

最新版本的 ANSI C 标准要求, 如果采用 `char` 或 `unsigned char` 类型数据运算时发生溢出, 则在运算过程中采用 `int` 类型数据。这种新要求是基于 16 位 CPU 中 `int` 和 `char` 类型数据的操作很相似这个前提的。C51 编译器的默认值支持这一特性, 同时提供两条新控制命令 `INTPROMOTE` 和 `NOINTPROMOTE` 分别用于允许和禁止 ANSI C 标准整型数据提升。由于 8051 系列单片机的 CPU 为 8 位, 因此它对于 8 位或 16 位数据的操作在代码长度和执行速度上具有很大的不同, 因此用户可以采用 `NOINTPROMOTE` 命令禁止 ANSI C 标准整型数据提升。如果用户希望最大程度地保持与其他 C 编译器或操作平台的兼容性, 则可以采用 Cx51 的默认值 `INTPROMOTE` 允许 ANSI C 标准整型数据提升。

- 在线行汇编

用户可以在 Cx51 源程序中采用新的控制命令 `ASM` 和 `ENDASM` 进行在线行汇编。使用了在线行汇编的 Cx51 源程序必须采用 `SRC` 控制命令进行编译。

- 新的控制命令

增加或增强了如下新的编译控制命令: `ASM`, `ENDASM`, `INTERVAL`, `INTPROMOTE`, `INTVECTOR`, `MAXARGS`, `NOINTPROMOTE`

- 指定中断向量的偏移量和间隔

用户可以为中断向量表指定偏移量和间隔, 这一特性提供了对 SIECO-51 单片机的支持, 并且允许用户当中断向量表不是位于 0000H 地址时指定不同的中断向量地址。

- 向间接调用函数传递参数

如果所有参数都可以用工作寄存器进行传递, 则可以向间接调用函数传递参数, 此时该函数不需要被声明为具有可再入函数的属性。

- 提供存储器定位函数的源代码

Cx51 编译器软件包中提供了存储器定位函数的源代码, 用户可以很容易修改这些函数源代码以适应自己嵌入式应用系统硬件结构的需要。

- 支持所有可变参数表列的函数

Cx51 编译器支持所有可变参数表列的函数, 具有可变参数表列的函数不再必须被声明为具有可再入函数的属性。新控制命令 `MAXARGS` 可以决定参数传递区域的大小。

## 附录 C 代码优化

本节给出了几种提高 Cx51 编译器生成 8051 代码效率（代码更小，速度更快）的方法。

### 1. 存储器模式

对于代码大小和执行速度影响最大的是存储器模式。使用 SMALL 模式对源程序进行编译可以生成长度最小和速度最快的代码。在 SMALL 模式下，所有变量（除非在声明时指定了存储器类型）都将定位在 8051 单片机的片内 RAM 中，而访问片内 RAM 的速度最快（一般为 1~2 个机器周期），同时所生成的代码要比采用 COMPACT 或 LARGE 模式时小得多。例如，对下面的一段循环程序：

```
for (i = 0; i < 100; i++) {
 do_nothing ();
}
```

采用 SMALL 模式进行编译将产生如下代码：

| stmt | level | source                      |
|------|-------|-----------------------------|
| 1    |       | #pragma small               |
| 2    |       | void do_nothing (void);     |
| 3    |       | void main (void) {          |
| 4    | 1     | unsigned char i;            |
| 5    | 1     | for (i = 0; i < 100; i++) { |
| 6    | 2     | do_nothing ();              |
| 7    | 2     | }                           |
| 8    | 1     | }                           |

#### ASSEMBLY LISTING OF GENERATED OBJECT CODE

```
 ; FUNCTION main (BEGIN)
 ; SOURCE LINE # 3
 ; SOURCE LINE # 5
0000 E4 CLR A
0001 F500 R MOV i,A
0003 ?C0001:
 ; SOURCE LINE # 6
0003 120000 E LCALL do_nothing
 ; SOURCE LINE # 7
```

```

0006 0500 R INC i
0008 E500 R MOV A,i
000A C3 CLR C
000B 9464 SUBB A,#064H
000D 40F4 JC ?C0001

 ; SOURCE LINE # 8
000F ?C0004:
000F 22 RET

 ; FUNCTION main (END)

```

可以看到在 **SMALL** 模式下, 变量 *i* 被放在片内 **RAM** 中, 存取变量 *i* 的指令“**MOV A, i**”和“**INC i**”其代码长度均为双字节, 此外这两条指令的执行时间均为一个机器周期。整个 **main** 函数采用 **SMALL** 模式编译所生成的代码长度只有 16 个字节。

若采用 **LARGE** 模式进行编译将产生如下代码:

```

 ; ; FUNCTION main (BEGIN)

 ; SOURCE LINE # 3
 ; SOURCE LINE # 5
0000 E4 CLR A
0001 900000 R MOV DPTR,#i
0004 F0 MOVX @DPTR,A
0005 ?C0001:

 ; SOURCE LINE # 6
0005 120000 E LCALL do_nothing

 ; SOURCE LINE # 7
0008 900000 R MOV DPTR,#i
000B E0 MOVX A,@DPTR
000C 04 INC A
000D F0 MOVX @DPTR,A
000E E0 MOVX A,@DPTR
000F C3 CLR C
0010 9464 SUBB A,#064H
0012 40F1 JC ?C0001

 ; SOURCE LINE # 8
0014 ?C0004:
0014 22 RET

 ; FUNCTION main (END)

```

在 **LARGE** 模式下, 变量 *i* 被放在外部 **XRAM** 中, 存取变量 *i* 时, 编译器首先要装入数据指针: **MOV DPTR, #i**, 然后再执行从外部 **XRAM** 取数操作: **MOVX @DPTR, A**, 执行这两条指令需要 4 个机器周期。整个 **main** 函数采用 **LARGE** 模式编译所生成的代码长度为 21 个字节。



## 2. 变量定位

需要经常访问的数据对象应存放在片内 RAM 中, 8051 单片机访问片内 RAM 的速度要比访问外部 XRAM 快得多。片内 RAM 由工作寄存器组、位寻址区、堆栈区以及其他用户采用 data 类型定义的变量共享。由于片内 RAM 容量的限制 (128~256 字节), 用户程序中的所有变量不可能全部放在片内 RAM 中, 此时必须将某些变量放入其他存储器空间。有以下两种解决问题的方法。

一种方法是改变存储器模式让编译器自动完成变量定位, 这种方法最简单, 但改变存储器模式可能导致生成的代码长度加大, 执行效率降低。

另一种方法是由用户根据需要来决定哪些变量可以存放在外部 XRAM 中, 并将这些变量定义为具有 xdata 存储器类型。通常字符串缓冲区和大型数组可以被定义为 xdata 存储器类型而不会对代码长度和执行效率造成太大的影响。

## 3. 尽可能使用最小的数据类型

8051 系列单片机为 8 位 CPU, 它对于单字节数据 (如 char、unsigned char 类型) 的操作要比多字节数据 (如 int、long 类型) 方便得多。因此建议用户尽可能使用最小的数据类型。如果不是运算时的特殊要求, 就不要进行 int 类型的转换。这一点可用乘法运算为例来加以说明: 两个 char 类型数据相乘恰好与 8051 指令 “MUL AB” 相符, 而如果采用 long 类型数据相乘则需要调用 Cx51 编译器的库函数。

## 4. 尽可能使用 unsigned 数据类型

8051 单片机不能直接支持带符号数的运算, 对于带符号数的操作 Cx51 编译器必须产生更多与之相关的代码, 因此尽可能采用 unsigned 类型数据将使生成的代码小得多。

## 5. 尽可能使用局部变量

只要有可能, 对于循环和临时运算应量采用局部变量。Cx51 编译器在进行优化处理时总是企图用工作寄存器来存放局部变量, 而对工作寄存器的存取操作是最快的, 通常采用 unsigned char 或 unsigned int 类型变量能获得最好的效果。

## 2. 变量定位

需要经常访问的数据对象应存放在片内 RAM 中, 8051 单片机访问片内 RAM 的速度要比访问外部 XRAM 快得多。片内 RAM 由工作寄存器组、位寻址区、堆栈区以及其他用户采用 data 类型定义的变量共享。由于片内 RAM 容量的限制 (128~256 字节), 用户程序中的所有变量不可能全部放在片内 RAM 中, 此时必须将某些变量放入其他存储器空间。有以下两种解决问题的方法。

一种方法是改变存储器模式让编译器自动完成变量定位, 这种方法最简单, 但改变存储器模式可能导致生成的代码长度加大, 执行效率降低。

另一种方法是由用户根据需要来决定哪些变量可以存放在外部 XRAM 中, 并将这些变量定义为具有 xdata 存储器类型。通常字符串缓冲区和大型数组可以被定义为 xdata 存储器类型而不会对代码长度和执行效率造成太大的影响。

## 3. 尽可能使用最小的数据类型

8051 系列单片机为 8 位 CPU, 它对于单字节数据 (如 char、unsigned char 类型) 的操作要比多字节数据 (如 int、long 类型) 方便得多。因此建议用户尽可能使用最小的数据类型。如果不是运算时的特殊要求, 就不要进行 int 类型的转换。这一点可用乘法运算为例来加以说明: 两个 char 类型数据相乘恰好与 8051 指令 “MUL AB” 相符, 而如果采用 long 类型数据相乘则需要调用 Cx51 编译器的库函数。

## 4. 尽可能使用 unsigned 数据类型

8051 单片机不能直接支持带符号数的运算, 对于带符号数的操作 Cx51 编译器必须产生更多与之相关的代码, 因此尽可能采用 unsigned 类型数据将使生成的代码小得多。

## 5. 尽可能使用局部变量

只要有可能, 对于循环和临时运算应量采用局部变量。Cx51 编译器在进行优化处理时总是企图用工作寄存器来存放局部变量, 而对工作寄存器的存取操作是最快的, 通常采用 unsigned char 或 unsigned int 类型变量能获得最好的效果。