Principle and Application of MCU

单片机原理及实用技术

——凌阳16位单片机原理及应用

[redacted]

Specially Designed
for Engineers and Technicians of Electronics

[redacted]

单片机原理及实用技术

——凌阳□□位单片机原理及应用

雷思孝　李伯成　雷向莉　编著

西安电子科技大学出版社

2004

内 容 简 介

本书系统介绍了凌阳 SPCE061A 单片机原理及实用技术，重点说明了该单片机在语音处理方面的特点及应用。本书共分 10 章：单片机系统概述、SPCE061A 单片机简介、SPCE061A 指令系统、SPCE061A 硬件系统、中断系统及应用、语音处理技术及应用、实用 C 语言程序设计、单片机应用系统设计、SPCE061A 实用技术、集成开发环境等。书中附录给出了 SPCE061A 单片机编程的基本函数，以便读者查阅。

本书力求简洁实用，注重设计方法介绍，侧重实际应用，可作为工科院校电子信息、计算机科学与技术、仪器仪表、工业自动化等相关专业本科生教材，也可作为高职相关专业学生的教材，并可作为学生科协及参加各类电子设计大赛的实用培训教材，也是工程技术人员及广大单片机爱好者参考使用。

图书在版编目（CIP）数据

单片机原理及实用技术：凌阳 16 位单片机原理及应用 / 雷思孝等编著.

—西安：西安电子科技大学出版社，2004.1

ISBN 7–5606–1321–7

Ⅰ. 单… Ⅱ. 雷… Ⅲ. 单片微型计算机，凌阳 Ⅳ. TP368.1

中国版本图书馆 CIP **数据核字（**2003**）第** 105077 **号**

策　　划　陈宇光

责任编辑　曹华　陈宇光

出版发行　西安电子科技大学出版社(西安市太白南路 2 **号)**

电　　话　(029)88242885　88201467　　**邮　编**　710071

http: //www.xduph.com　　E-mail: xdupfxb@pub.xaonline.com

经　　销　新华书店

印　　刷　西安兰翔印刷厂

版　　次　2004 **年** 1 **月第** 1 **版**　2004 **年** 1 **月第** 1 **次印刷**

开　　本　787 **毫米×**1092 **毫米**　1/16　**印张**　20.5

字　　数　485 **千字**

印　　数　1～4 000 **册**

定　　价　30.00 **元**

ISBN 7－5606－1321–7/TP・0699

XDUP 1592001–1

如有印装问题可调换

本社图书封面为激光防伪覆膜，谨防盗版。

前　言

近几年来，“后 PC 时代”的提法不时见诸报端。863 计算机首席专家高文教授曾经说过：所谓后 PC 时代，是英文 Pervasive Computing 的中文意译，Pervasive 的原意是普遍的、蔓延的、渗透的，所以 Pervasive Computing 应译为渗透到各个方面的计算。因此，后 PC 时代是指计算机无处不在，将渗透到我们生活的方方面面。后 PC 时代的绝大多数计算机会以非计算机的形式出现。例如，电视机、电冰箱、机器人、数控设备等，其中心控制部件是计算机，但它几乎都是以嵌入式系统的形式存在的，而不是像 PC 机一样摆放在计算机工作台上的。作为嵌入式系统主要部件的单片机已成为一个重要的研究领域。

单片机作为计算机技术的一个重要分支、嵌入式系统的先头兵，广泛应用于工业控制、智能仪器、机电一体化产品、家用电器、智能产品、个人数字处理器等领域。随着微电子技术的迅速发展，电子系统设计已进入了片上系统(SOC，System On Chip)时代，单片机的功能也越来越强，使其真正成为系统单片机。随着社会需求和单片机应用领域的不断扩展，学生在校期间参加的各类电子设计大赛、遇到的各种开发应用课题，都是以单片机技术为基础的。各种应用系统开发研制更是离不开单片机应用技术。单片机应用技术已经成为理工科院校电子类专业学生的必备技术，大多数工科院校已将“单片机原理及应用”作为一门重要的专业技术课。所以，如何使学生在有限的时间内掌握最新的单片机应用技术，就成为单片机教学的一个重要课题。经过比较和教学实践，我们认为凌阳科技公司最新研制的 SPCE061A 型 16 位单片机不但具有单片机的基本功能，而且在语音识别和处理方面有其独到之处，开发环境简单，便于学生学习和实践，应用十分广泛。因此，我们选用该单片机为对象，结合教学实际，系统介绍其原理与应用，重在掌握系统设计方法和开发技能，力求使学生尽快掌握单片机的应用技能和设计方法，配合相关实验，设计出有特色的智能产品，并能在实际中得到应用。

2002 年，为了及时推广凌阳公司最新研制的基于 SOC 的单片机，在凌阳公司和北阳公司提供资料的基础上，我们编写了《SPCE061A 16 位单片机原理及应用》讲义，经过教学实践，学生能够在有限的课时内掌握系统设计方法，设计出有特色的应用产品，反映良好。为了进一步提高学生的学习积极性，也能为工程技术人员提供一本介绍凌阳单片机及其在语音处理应用方面的参考书，作者结合多年来的教学、科研经验，在原讲义的基础上，增强了系统设计方法和应用实例方面的内容，编写了本教材。本书可作为工科院校电子信息、计算机、仪器仪表、工业自动化等相关专业本科学生的教材，也可作为高职相关专业学生的教材，是学生科协及参加各类电子设计大赛的实用培训教材，也是工程技术人员及单片机爱好者的一本实用参考书。相信随着“凌阳大学”计划的不断深入，本书一定能够在推广单片机应用领域展现出其特有的风采。

本书共分 10 章：单片机系统概述、SPCE061A 单片机简介、SPCE061A 指令系统、SPCE061A 硬件系统、中断系统及应用、语音处理技术及应用、实用 C 语言程序设计、单

片机应用系统设计、SPCE061A 实用技术、集成开发环境等。

本书在编写过程中，得到了台湾凌阳科技公司同仁的大力支持；得到了北京北阳电子公司罗亚非、刘宏韬老师及工程技术人员的支持和帮助，张向艳、李华丽、魏广远、刘新颜、袁军等工程师参与了书稿的审校；受到了西安电子科技大学计算机学院副院长武波教授的热情关怀和大力支持；西安电子科技大学出版社总编助理陈宇光、副编审云立实、编辑曹华为本书的及时出版提供了帮助；任家鲁、谈宗玮同学参与了程序的整理和校对，在此一并表示真诚的感谢。同时，本书还借鉴了现有许多教材的宝贵经验，在此也对各位作者表示衷心的感谢。

李伯成老教授不辞辛苦，指导和审阅了全书并提出了许多非常有益的意见，在此表示诚挚的感谢。

由于编者水平有限，对于相关技术掌握不精，加之时间仓促，书中难免存在不妥和错误之处，敬请各位老师、读者不吝指正。

编　者

2003 年 8 月

目　录

第 1 章　单片机系统概述 1
1.1　单片机系统的特点 1
1.2　单片机技术的发展 1
1.2.1　发展过程 1
1.2.2　技术发展 2
1.3　单片机系统应用 3
第 2 章　SPCE061A 单片机简介 5
2.1　SPCE061A 概述 5
2.1.1　主要性能 5
2.1.2　SPCE061A 模块结构 5
2.1.3　芯片管脚排列及功能 6
2.1.4　最小应用系统 7
2.1.5　开发应用方式 8
2.2　μ'nSP™ 内核结构 8
2.2.1　算术逻辑运算单元 9
2.2.2　寄存器组 10
2.2.3　总线结构 10
2.3　SPCE061A 单片机存储器 11
2.3.1　RAM 11
2.3.2　堆栈 11
2.3.3　闪存 FLASH (闪速存储器) 12
思考题 13
第 3 章　SPCE061A 指令系统 14
3.1　概述 14
3.1.1　指令分类 14
3.1.2　寻址方式 14
3.2　数据传送指令 16
3.3　算术运算指令 19
3.3.1　加法运算指令 19
3.3.2　减法运算指令 21
3.3.3　带进位的加减法运算指令 23
3.3.4　乘法运算指令 23
3.3.5　内积运算指令 24
3.3.6　比较指令 25
3.4　逻辑运算指令 25
3.4.1　逻辑与指令 25
3.4.2　逻辑或指令 27
3.4.3　逻辑异或指令 28
3.4.4　测试指令 30
3.4.5　位移操作指令 32
3.5　控制转移类指令 35
3.6　常用伪指令 38
3.6.1　伪指令的语法格式及特点 38
3.6.2　伪指令符号约定 39
3.6.3　标准伪指令 39
3.6.4　宏定义与调用 40
3.6.5　段定义与调用 41
3.6.6　结构定义与调用 42
3.6.7　过程定义与调用 43
思考题 44
第 4 章　SPCE061A 硬件系统 45
4.1　并行输入/输出端口 45
4.1.1　I/O 端口结构 45
4.1.2　端口设置寄存器 46
4.1.3　工作方式设置 47
4.1.4　B 口的特殊功能 50
4.2　系统时钟与复位电路 52
4.2.1　时钟电路 52
4.2.2　锁相环 PLL(Phase Lock Loop) 振荡器 52
4.2.3　系统时钟 52
4.2.4　时间基准信号 53
4.2.5　复位电路 54
4.3　定时器/计数器 55
4.3.1　工作原理 55
4.3.2　工作寄存器设置 56
4.3.3　定时器/计数器应用 60
4.4　A/D(模/数)转换器 60
4.4.1　A/D 转换 60

4.4.2 A/D 转换器设置 62
4.4.3 A/D 转换器的物理性能 63
4.5 DAC 音频输出 66
4.5.1 硬件组成 66
4.5.2 寄存器设置 67
4.6 串行设备输入/输出端口 SIO 68
4.6.1 寄存器设置 69
4.6.2 应用举例 70
4.7 通用异步串行通信口 UART 72
4.7.1 UART 数据帧格式 72
4.7.2 工作寄存器设置 72
4.8 工作方式 76
4.8.1 睡眠方式 76
4.8.2 低电压监测/低电压复位 (LVD/LVR) 77
4.8.3 看门狗监视器 77
4.8.4 保密设定 78
4.8.5 语音编码类型 78
思考题 78
第5章 中断系统及应用 80
5.1 概述 80
5.2 SPCE061A中断系统 81
5.2.1 中断类型 81
5.2.2 中断源 82
5.2.3 中断控制 84
5.2.4 中断向量及优先级 87
5.2.5 中断响应 88
5.3 中断系统应用 90
5.3.1 单中断源应用 90
5.3.2 多中断源应用 103
思考题 110
第6章 语音处理技术及应用 111
6.1 概述 111
6.1.1 音频信号 111
6.1.2 音频信号的抽样和量化 111
6.1.3 音频格式 111
6.1.4 语音压缩编码基础 113
6.1.5 语音合成和识别技术 114
6.2 凌阳音频技术简介 115
6.2.1 压缩算法的编码标准 115
6.2.2 压缩分类 115
6.2.3 常用的音频形式和压缩算法 116
6.3 常用应用程序接口 116
6.3.1 程序接口概述 116
6.3.2 SACM_A2000 117
6.3.3 SACM_S480 123
6.3.4 SACM_S240 126
6.3.5 SACM_MS01 130
6.3.6 SACM_DVR 135
6.4 语音压缩方法 136
6.4.1 DOS环境下压缩 136
6.4.2 Windows环境下压缩 136
6.5 键控放音程序 137
6.5.1 常用的键盘API函数 138
6.5.2 系统资源模块 138
6.5.3 应用实例 139
6.6 语音识别技术 143
6.6.1 设置 143
6.6.2 应用实例 145
第7章 实用C语言程序设计 153
7.1 基本数据类型 153
7.2 流程控制语句 154
7.3 构造数据类型 157
7.4 C 语言与汇编相互调用 166
7.5 I/O 端口的 C 语言程序设计 169
7.6 系统设置的 C 语言程序设计 171
7.6.1 系统设置的 C 函数 171
7.6.2 应用实例 173
7.7 定时器/计数器设置的 C 函数 174
7.7.1 库中提供的 API 函数 174
7.7.2 定时器/计数器应用实例 176
7.8 中断系统的 C 语言程序设计 178
7.8.1 中断控制的相关 C 函数 178
7.8.2 中断系统应用实例 180
7.9 A/D、D/A 转换的 C 语言程序设计 187
7.9.1 相关 C 函数 187
7.9.2 应用实例 191

7.10 SIO 和 UART 的 C 语言程序设计.....193
7.10.1 相关 C 函数.....193
7.10.2 应用实例.....197
第 8 章 单片机应用系统设计.....200
8.1 系统设计内容.....200
8.1.1 系统硬件组成.....200
8.1.2 系统设计内容.....202
8.2 系统开发过程.....203
8.3 系统设计方法.....204
8.3.1 确定系统功能.....204
8.3.2 确定系统结构.....205
8.3.3 系统设计原则.....206
8.3.4 系统硬件设计.....207
8.3.5 系统软件设计.....208
8.3.6 资源分配.....209
8.4 系统调试.....209
8.4.1 常用工具.....210
8.4.2 调试方法.....211
8.5 SPCE061A 系统设计举例.....215
8.5.1 市场调研与功能确定.....215
8.5.2 硬件设计.....215
8.5.3 软件设计.....219
第9章 SPCE061A实用技术.....239
9.1 应用领域.....239
9.1.1 数字信号处理.....239
9.1.2 便携式终端.....240
9.1.3 嵌入式系统.....240
9.1.4 凌阳单片机应用.....241
9.2 应用举例.....241
9.2.1 作息时间控制.....241
9.2.2 温度计设计.....245
9.2.3 红外遥控器.....247
9.2.4 学习机.....250
9.2.5 多功能信号发生器.....252
9.2.6 语音识别.....255
思考题.....259
第 10 章 集成开发环境.....266
10.1 调试菜单.....266
10.2 工具栏.....267
10.3 窗口使用.....269
10.4 项目管理.....278
10.4.1 建立项目.....278
10.4.2 在项目中新建 C 文件(.C).....279
10.4.3 在项目中新建汇编文件(.asm).....280
10.4.4 在项目中新建头文件(.H).....281
10.4.5 在项目中新建文本文件(.txt).....281
10.4.6 在项目中新建二进制文件.....282
10.4.7 在项目中添加/删除文件.....282
10.4.8 在项目中使用资源.....283
10.4.9 项目选项的设置.....283
10.4.10 项目的编译.....286
10.5 代码剖视器.....286
10.5.1 激活Profile的方法.....286
10.5.2 使用Profile的步骤.....287
附录.....290
附录A 端口功能总述.....290
附录 B SPCE061A 指令集.....291
附录 C C_LIB 函数集.....299
附录 D 常用伪指令表.....309
附录 E SPCE061A 编译相关错误信息.....310
参考文献.....319

第 1 章　单片机系统概述

随着嵌入式系统开发技术的快速发展及其在各个领域的广泛应用，人们对电子产品的小型化和智能化要求越来越高，作为高新技术之一的单片机以其体积小、功能强、价格低、使用灵活等特点，显示出其明显的优势和广泛的应用前景。在航空航天、机械加工、智能仪器仪表、家用电器、通信系统、智能玩具等领域，单片机都发挥了很大的作用。可以认为，单片机技术已成为现代电子技术应用领域十分重要的技术之一，是电子技术应用领域工程技术人员必备的知识和技能，它能够使您设计的产品更具智能化和先进性。本章将重点介绍单片机应用系统的特点及发展概况。

1.1　单片机系统的特点

所谓单片机(Single Chip Microcomputer)，是指在一块芯片中集成有中央处理器(CPU)、存储器(RAM 和 ROM)、基本 I/O 接口以及定时器/计数器等部件，并具有独立指令系统的智能器件，即在一块芯片上实现一台微型计算机的基本功能。如果是简单控制对象，只需利用单片机作为控制核心，不需另外增加外部设备就能完成。对于较复杂的系统，只需对单片机进行适当扩展即可，十分方便。归纳起来，单片机及应用系统有以下特点：

(1) 单片机具有独立的指令系统，可以将我们的设计思想充分体现出来。

(2) 系统配置以满足控制对象的要求为出发点，使得系统具有较高的性能价格比。

(3) 应用系统通常将程序驻留在片内(外)ROM 中，抗干扰能力强，可靠性高，使用方便。

(4) 由于系统规模较小，其本身不具有自我开发能力，一般需借助专用的开发工具进行系统开发和调试，而实际应用系统简单实用，成本低，效益好。

(5) 应用系统所用存储器芯片可选用 EPROM、E^2PROM、OTP 芯片或利用掩膜形式生产，便于批量开发和应用。许多单片机(如 80C51 系列)的开发芯片和扩展应用芯片相互配套，降低了系统成本。

(6) 由于系统小巧玲珑，控制功能强、体积小，便于嵌入被控设备之内，大大推动了产品的智能化。如数控机床、机器人、智能仪器仪表、洗衣机、电冰箱、电视机等都是典型的机电一体化设备和产品。

1.2　单片机技术的发展

1.2.1　发展过程

单片机是随着微型计算机、单板机的发展及其在智能测控系统中的应用而发展起来的。

以 8 位单片机为例，其发展过程大致可归纳为四个阶段。

第一阶段：低性能单片机阶段(1976～1980 年)。该阶段是以较简单的 8 位低档单片机为主，将原有的单板机功能集成在一块芯片上，是该芯片具有原来单板机的功能。其主要代表芯片为 Intel 公司的 MCS-48 系列，该芯片内集成了 8 位 CPU、并行 I/O 端口和 8 位定时器/计数器，寻址范围为 4 KB，没有串行通信接口。

第二阶段：高性能单片机阶段(1980～1983 年)。该阶段仍以 8 位机为主，主要增加了串行口、多级中断处理系统和 16 位定时器/计数器，除片内 RAM、ROM 容量加大外，片外寻址可达 64 KB，有的片内还集成有 A/D、D/A 转换器。这一阶段单片机以 Intel 公司的 MCS-51 系列、Motorola 公司的 6801 系列和 Zilog 公司的 Z8 系列为代表。上述机型由于功能强，使用方便，目前仍在广泛应用。

第三阶段：高性能的 16 位单片机(1983～80 年代末)。该单片机性能更加完善，主频速率提高，运算速度加快，具有很强的实时处理能力，更加适用于速度快、精度高、响应及时的应用场合。其主要代表为 Intel 公司的 MCS-96 系列等。

第四阶段：该阶段(90 年代)单片机在集成度、速率、功能、可靠性、应用领域等全方位向更高水平发展。该阶段 CPU 数据线有 8 位、16 位、32 位，采用双 CPU 结构或内部流水线结构，以提高数据处理能力和运算速度；采用内部锁相环技术，时钟频率已高达 50 MHz，指令执行加快；提供了运算能力较强的乘法指令和内积运算指令，具有较强的数据处理能力；设置了新型的串行总线结构，为系统扩展提供了方便；增加了常用的特殊功能部件(如系统看门狗 WDT、通信控制器、调制解调器、脉宽调制输出 PWM 等)。随着微电子技术的发展和半导体工艺的不断改进，芯片正在向着高集成化、低功耗的方向发展。随着应用范围的不断扩大，一些专用单片机也迅速发展壮大。

1.2.2 技术发展

随着微电子技术的迅速发展，目前各个公司研制出了适用于各种应用领域的单片机。高性能单片机芯片市场也异常活跃，不断采用新技术，使单片机的种类、性能不断提高，应用领域不断扩大。如 ATMEL 公司开发的 89C51 芯片，片内含有 4 KB E^2PROM；89C51FA 芯片，片内有 8 KB E^2PROM；89C51FB 芯片，片内含 16 KB E^2PROM；凌阳公司最新推出的 SPCE061A 芯片，片内有 32 KB FLASH 和 2 KB RAM，使得片内可储存量增加，控制能力增强。这些性能的提高和改进归纳起来，有以下四个方面。

1. CPU 的改进

CPU 的改进体现在以下几个方面：

(1) 采用双 CPU 结构，提高芯片的处理能力。如 Rockwell 公司的单片机 R6500/21 和 R65C29 均采用双 CPU 结构，大大提高了系统的处理能力。

(2) 增加数据总线宽度，提高处理速度，从 8 位、16 位到 32 位。

(3) 采用流水线结构，类似于高性能的微处理器，提高了运算速度，能够实现简单的 DSP 功能，适合于做数字信号处理。

(4) 串行总线结构，将外部数据总线改为串行传送方式，提高了系统可靠性。

2. 存储器的发展

存储器的发展体现在以下几个方面：

(1) 增大片内存储器容量，有利于提高系统的可靠性。

(2) 片内采用 E^2PROM 和 FLASH 技术，不需紫外线擦除，读写方便，可对某些需要保留的数据和参数长期保存，提高了单片机的可靠性和实用性，如 89C51、SPCE061A 等单片机。

(3) 编程加密技术。为保护知识产权，开发者希望软件不被复制、破译，而增加了编程加密位或 ROM 加锁方式，达到了程序保密的目的。

3. 片内 I/O 接口

片内 I/O 接口的特点有以下几个方面：

(1) 增加驱动能力，减少了外围驱动芯片的使用，可直接驱动 LED、LCD 显示器等，简化了系统设计，降低了系统成本。

(2) 增加了异步串行通信口，提高了单片机系统的灵活性。

(3) 增加了逻辑操作功能，具有位寻址操作，增强了操作和控制的灵活性。

(4) 带有 A/D、D/A 转换器，直接对模拟量信号输入和输出。

(5) 并行 I/O 端口设置灵活，可以利用指令将端口的任一位设置为输入、输出、上拉、下拉和悬浮等状态。

(6) 带有 PWM 输出，可直接驱动控制小型直流电机调速，大大方便了使用。

4. 制造工艺

随着微电子技术的不断发展，单片机正朝着高集成度、低能耗、低电压、多功能的方向发展。

1.3　单片机系统应用

由于单片机体积小、价格低、可靠性高、适用范围大以及有其本身的指令系统等诸多优势，在各个领域、各个行业都得到了广泛应用。目前不仅有常用的 8 位单片机，而且 16 位机已得到了广泛的应用。

1. 机电一体化

机电一体化是机械设备发展的方向。用单片机代替常规的逻辑顺序控制，简化了结构设计，提高了控制性能。当前的许多产品，如数控车床等都是采用这种方式。最典型的机电产品是机器人，每个关节或动作部位都是由一个单片机系统控制。

2. 集散数据采集系统

在实时控制系统中，要求数据采集具有较好的同步性和实时性，若采用单个计算机顺序采集，存在不能同时采集、实时性不强等缺点，以致会造成计算、处理上的误差而引起分析统计困难。使用单片机作为系统的前端采集单元，由主控计算机发出同时采集命令，当采集完成后，将采集到的数据再逐一传送到主机中进行处理，保证了同步数据采集。如

气象部门、供电系统、自来水管网、过程控制等均可采用集散数据采集系统。

3. 分布式控制系统

通常分布式控制系统采用模块化设计，而单片机正是某些模块的控制中心。如生产线、过程控制、遥测遥控系统等等。

4. 智能仪器仪表

单片机的应用使自动化仪器仪表的智能化程度越来越高，如自动计费电度表、燃气表，许多工业仪表中的智能流量计、气体分析仪、成分分析仪等，各种检测仪器仪表中的多功能信号发生器、智能电压电流测试仪、医疗器械、检测仪器等都使用了单片机。

5. 家用电器

在洗衣机、空调器、汽车控制系统、保安系统、电视机、录像机、VCD 机、音响设备、电子秤、IC 卡、手机、智能玩具等系统及设备中使用了大量各种各样的单片机，使其性能大大提高，实现了智能化和最优化控制。

6. 终端及外部设备控制

在计算机网络终端设备(如银行终端、商业 POS(自动收款机)、GPS 电子地图、复印机等)和计算机外部设备(如打印机、绘图仪、键盘和通信终端等)中都使用了单片机。单片机的使用使这些设备既具有计算、存储、显示和数据处理等功能，又具有和计算机连接的端口，使计算机的应用能力和范围大大提高，更好地发挥了计算机的性能。

第 2 章　SPCE061A 单片机简介

2.1　SPCE061A 概述

SPCE061A 型单片机是凌阳科技公司最新推出的一款 16 位微处理器，具有体积小、集成度高、易扩展、可靠性高、功耗低、结构简单、中断处理能力强等特点，内嵌 32 K 字闪存 FLASH，处理速度高，能够很方便地完成普通单片机的功能，尤其适应于数字语音播报和识别等应用领域，是数字语音识别与语音信号处理的理想产品，得到了广泛的应用。

2.1.1　主要性能

SPCE061A 单片机的主要性能有：

- 内核采用 16 位 μ'nSP™微处理器。
- 工作电压 VDD 为 2.6～3.6 V(CPU)，I/O 端口高电平 VDDH 为 VDD～5.5 V。
- CPU 时钟频率为 0.320～49.152 MHz。
- 内置 2 K 字 SRAM。
- 内置 32 K 字闪存 FLASH。
- 具有可编程音频处理功能。
- 低功耗，系统处于备用状态下(时钟处于停止状态)耗电小于 2 μA@3.6 V。
- 两个 16 位可编程定时器/计数器(可自动预置初值)。
- 两路 10 位数/模转换(DAC)输出通道。
- 32 位通用可编程输入/输出端 A 口和 B 口。
- 14 个中断源(定时器 A、B，时基信号，两个外部时钟源，触键唤醒等)。
- 具有触键唤醒功能。
- 使用凌阳音频编码 SACM_S240 方式(2.4 kb/s)，能容纳 210 s 语音数据。
- 实时时钟频率为 32 768 Hz，由锁相环 PLL 振荡器提供系统时钟信号。
- 7 通道 10 位电压模/数转换器(ADC)和单通道声音模/数转换器。
- 语音模/数转换器输入通道，内置麦克风放大器和自动增益控制(AGC)电路。
- 具有串行设备通信接口 SIO。
- 具有低电压复位(LVR)和低电压监测(LVD)功能。
- 内置看门狗监视器。
- 内置在线仿真(ICE，In-Circuit Emulator)接口。

2.1.2　SPCE061A 模块结构

SPCE061A 单片机内部结构和组成如图 2.1 和图 2.2 所示。其主要功能模块有并行 I/O 端口、模/数转换 ADC、数/模转换 DAC、存储器 RAM & FLASH、定时器/计数器 T/C(脉

宽调制输出 PWM)、WatchDog、异步串行通信口 UART、指令寄存器 IR、设备串行口 SIO、低电压检测 LVD(低电压复位)等。

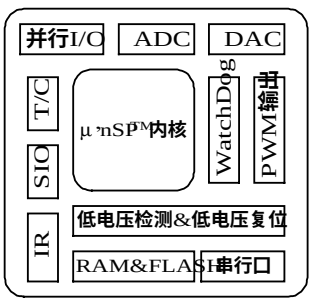


图 2.1 SPCE061A 模块结构图

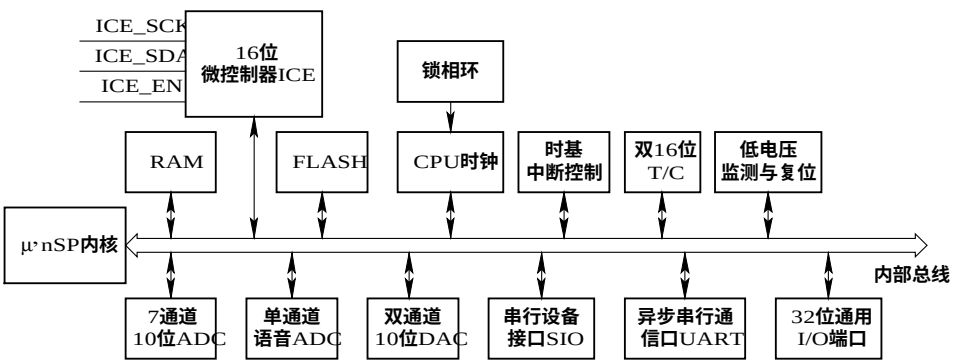


图 2.2 SPCE061A 内部结构图

2.1.3 芯片管脚排列及功能

芯片管脚排列如图 2.3 所示，管脚功能如表 2.1 所示。

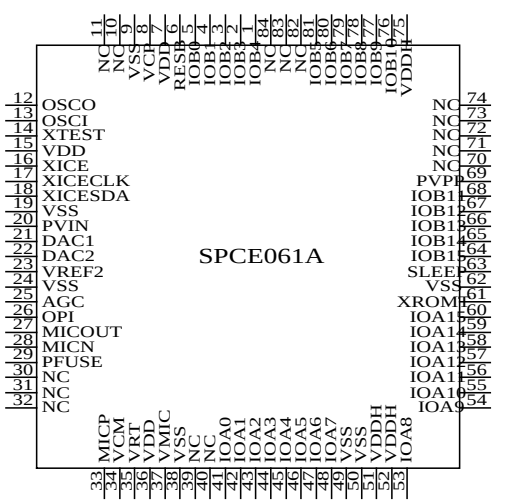


图 2.3 SPCE061A 引脚排列图

表 2.1　SPCE061A 管脚功能表

管脚名称		管脚功能
IOA0～IOA15	(41～48,53～60)	I/OA 口，16 个
IOB0～IOB15	(5～1,81～76,68～64)	I/OB 口，16 个
OSCI	13	振荡器输入，采用石英晶振时，接晶振
OSCO	12	振荡器输出，采用石英晶振时，接晶振
RES_B	6	复位信号输入端，低电平有效
ICE_EN	16	ICE 使能端，接在线调试器 PROBE 的使能脚 ICE_EN
ICE_CLK	17	ICE 时钟脚，接在线调试器 PROBE 的时钟脚 ICE_CLK
ICE_SDA	18	ICE 数据脚，接在线调试器 PROBE 的数据脚 ICE_SDA
PVIN	20	程序保密设定电源输入脚
PFUSE	29	程序保密设定输入脚
DAC1	21	音频输出通道 1
DAC2	22	音频输出通道 2
VREF2	23	2 V 参考电压输出脚
AGC	25	语音输入自动增益控制引脚
OPI	26	Microphone 第二级运放输入脚
MICOUT	27	Microphone 第一级运放输出脚
MICN	28	Microphone 正向输入脚
MICP	33	Microphone 负向输入脚
VRT	35	A/D 转换参考电压输入脚
VCM	34	ADC 参考电压输出脚
VMIC	37	Microphone 电源
SLEEP	63	睡眠状态指示，当 CPU 进入睡眠状态时，输出高电平
VCP	8	锁相环压控振荡器阻容输入端
XROMT PVPP XTEST (61. 69.14)		出厂测试用管脚，使用时悬空即可
VDDH	51. 52. 75	I/O 电平参考电压输入端，当输入参考电压为 5 V/3.3 V 时，I/O 输入、输出高电平为 5 V/3.3 V
VDD	7	锁相环 PLL 电源
VSS	9	锁相环 PLL 地
VSS	19. 24	模拟信号地
VSS	38. 49. 50. 62	数字信号地
VDD	15. 36	数字信号电源

2.1.4　最小应用系统

SPCE061A 单片机的最小应用系统构成非常方便，只需在 OSCO、OSCI 端接石英晶体振荡器及谐振电容，在复位端接复位电路，在锁相环压控振荡器的阻容输入 VCP 端接上相应的电容、电阻后即可工作，其它不用的电源端和接地端可接上 104 小电容或 100 μF 的去耦电容，以提高抗干扰能力，如图 2.4 所示。

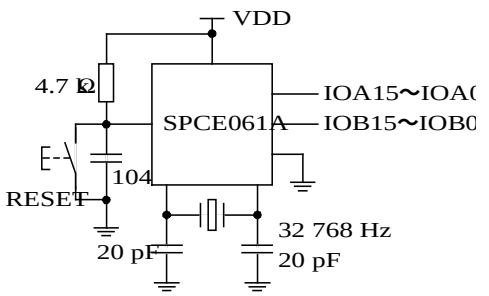


图 2.4 SPCE061A 最小系统原理图

2.1.5 开发应用方式

SPCE061A 单片机开发系统由 PC 机、在线调试器 PROBE 和用户目标板构成。在线调试器 PROBE 实现目标板与 PC 机之间的信息转换和传输，它既是一个编程器(完成程序烧写功能)，又是一个实时在线调试器，用它可以替代传统单片机应用项目开发过程中常用的两种工具——实时仿真器和程序烧写器。PROBE 是利用 SPCE061A 内置的在线仿真电路 ICE(In-Circuit Emulator)和凌阳公司的在线串行编程技术实现仿真和编程的。PROBE 工作于凌阳 IDE 集成开发环境下，其 5 芯仿真头连接到目标电路板上的 SPCE061A 相应管脚，直接在目标电路板的 SPCE061A 单片机上调试和运行用户程序，PROBE 的另一头是标准的 25 针并行接口，与计算机并行端口(打印机接口)相连，实现与上机通信。在 IDE 集成开发环境下，完成在线调试、下载功能(详见第 10 章“集成开发环境”)。图 2.5 是计算机、PROBE 和用户目标板三者之间的连接示意图。

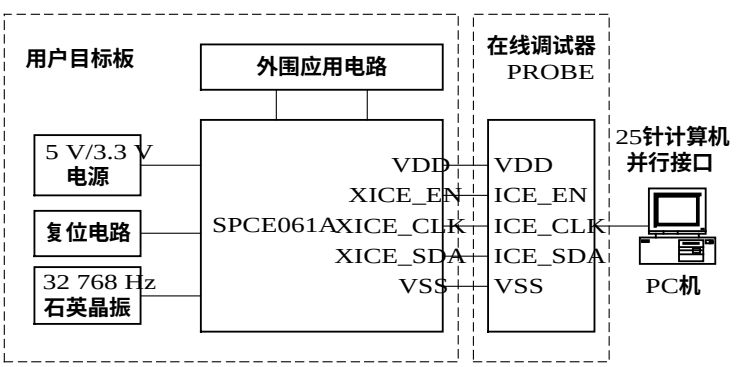


图 2.5 计算机、PROBE 和用户目标板连接示意图

2.2 μ'nSP™内核结构

SPCE061A 单片机采用凌阳 μ'nSP™内核，为了使读者对 SPCE061A 单片机有一个总体了解，首先我们对 μ'nSP™内核结构作一简单介绍。

μ'nSP™内核主要由总线、算术逻辑运算单元、寄存器组、中断系统及堆栈等部分组成，如图 2.6 所示。

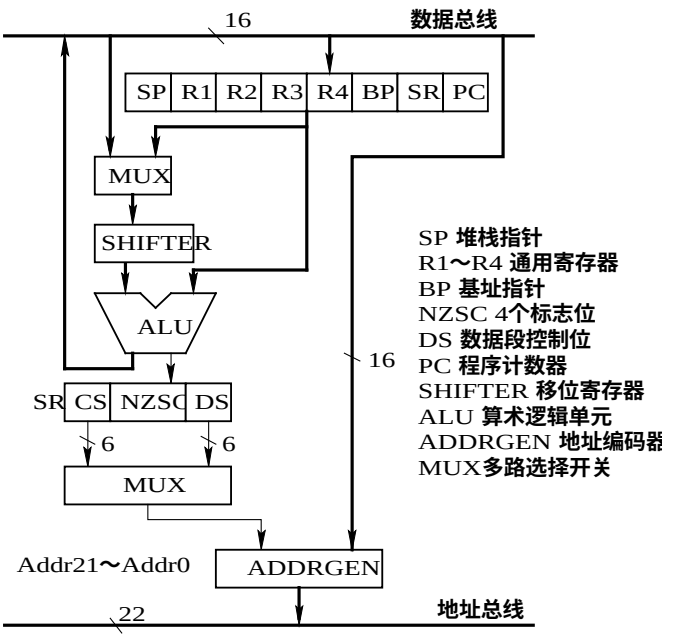


图 2.6　μ'nSP™内核结构示意图

2.2.1　算术逻辑运算单元

μ'nSP™的算术逻辑运算单元 ALU 能够完成 16 位基本的算术逻辑运算和带移位操作的 16 位算术逻辑运算以及用于数字信号处理的 16 位乘法运算、内积运算等。

1. 16 位算术逻辑运算

与大多数 CPU 类似，μ'nSP™提供了基本的 16 位算术逻辑运算及数据传送指令，如加、减、乘、比较、取补、异或、或、与、测试等。

2. 带移位操作的 16 位算术逻辑运算

由图 2.6 可以看出，μ'nSP™内核的 ALU 前面串接有一个移位寄存器 SHIFTER，也就是说，操作数在经过 ALU 运算操作前可先进行移位处理，然后再由 ALU 完成相应的运算操作。移位包括算术右移、逻辑左移、逻辑右移、循环左移及循环右移等，所以μ'nSP™的指令系统里有一组复合的移位算术逻辑操作指令，即一条指令能够完成移位和算术逻辑操作两项功能，指令功能很强。程序设计者可利用这些复合指令编写出更精简的程序，进而增加程序代码密集度(Code Density)，减少程序代码对 ROM 或 FLASH 的需求，降低系统成本，提高执行效率。

3. 16 位×16 位的乘法运算和内积运算

除了普通的 16 位算术逻辑运算指令外，μ'nSP™指令系统还提供了处理速度较高的 16 位×16 位乘法运算指令(Mul)和内积运算指令(Muls)，这两种指令都可以用于两个有符号数或一个有符号数与一个无符号数的运算，Mul 指令只需花费 12 个时钟周期，Muls 指令需花费 10n+8 个时钟周期，其中 n 为乘积求和的项数。

例如：MR=[R2]*[R1],4；表示求 4 项乘积的和，Muls 指令只需花费 48(10×4＋8=48)个时钟周期，为μ'nSP™应用于复杂的数字信号处理提供了极大的方便。

2.2.2 寄存器组

μ'nSP™寄存器组由 8 个 16 位寄存器组成，分为通用寄存器和专用寄存器两大类。通用寄存器包括 R1～R4(作为算术逻辑运算的源寄存器及目标寄存器)；专用寄存器包括 SP、BP、SR、PC 等，下面分别作一介绍。

1. 通用寄存器 R1～R4

通用寄存器 R1～R4 通常用于数据运算、数据传送的源寄存器及目标寄存器。寄存器 R4 和 R3 配对使用，可组成一个 32 位的乘法结果寄存器 MR(其中 R4 存结果的高字组，R3 存结果的低字组)，用于存放乘法运算或内积运算的结果。

2. 堆栈指针寄存器 SP

CPU 在执行压栈/出栈指令(PUSH/POP)、子程序调用/返回指令(CALL/RETF)以及进入中断服务程序 ISR(Interrupt Service Routine)，或从中断服务程序返回(RETI)时，随着堆栈的变化，堆栈指针随之移动，SP 就是用来存放堆栈指针的专用寄存器。堆栈的最大容量范围限制在 2 K 字的(RAM)范围内，即在 0x0000H～0x07FFH 的存储器范围内。

3. 基址指针寄存器 BP

μ'nSP™提供了一种变址寻址方式[BP+IM6]，可直接存取 FLASH 与 RAM 中的各种数据，包括局部变量(Local Variable)、函数参数(Function Parameter)、返回地址(Return Address)等等，这在 C 语言程序设计时特别有用。BP 除了上述用途外，也可作为通用寄存器 R5 使用，用于数据运算或传送寄存器，因此本教材(或程序)中 BP 与 R5 是共享的，均代表基址指针寄存器。

4. 段寄存器 SR

SR 有多种功能，有代码段选择字段(CS)和数据段选择字段(DS)，它们可分别与其它 16 位的寄存器组合在一起形成 22 位地址信息，寻址空间为 4 M 字容量的存储器，如下所示：

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

D15～D10：数据段选择字段(DS)6 位；

D9～D6：算术逻辑运算结果标志位(NZSC) 4 位；

D5～D0：代码段选择字段(CS)6 位。

算术逻辑运算结果标志位为 N、Z、S、C(SR 中的 D9～D6)，CPU 在执行条件跳转指令时，需测试这些标志位，以控制程序的流向。

2.2.3 总线结构

μ'nSP™具有 16 位数据总线和 22 位地址总线，由此决定其基本数据类型是 16 位的 Word 型，而不是 8 位单片机的 Byte 型，因而每次存储器都是按 Word 操作的。22 位地址线最多可寻访 4 M 字的存储容量，地址线中的高 6 位 A21～A16 来自段寄存器 SR 中的 6

位代码段 CS 和 6 位数据段 DS 选择字段，低 16 位 A15～A0 来自内部寄存器。通常，地址线的高 6 位称为存储器地址的段码选择简称段码(Segment)，而低 16 位则称为存储器地址的偏移量 Offset。μ'nSP™通过对段(Segment)的编码来实现存储器页的检索，通过 Segment 与 Offset 的配合即可产生 22 位地址信号。

注意：SPCE061A 只有 32 K 字闪存 FLASH，仅占一页存储空间，所以在 SPCE061A 单片机中，代码段选择字段(CS)和数据段选择字段(DS)均未使用。

2.3　SPCE061A 单片机存储器

SPCE061A 单片机片内存储器地址映射表如图 2.7 所示，单片机的存储器有 2 K 字的 SRAM(包括堆栈区)和 32 K 字闪存 FLASH。



图 2.7　SPCE061A 内存映射表

2.3.1　RAM

SPCE061A 有 2 K 字的 SRAM(包括堆栈区)，其地址为 0x0000～0x07FF，前 64 个字即 0x0000～0x003F 地址范围内可采用 6 位地址直接寻址，寻访速度为 2 个时钟周期，其余 0x0040～0x07FF 地址范围内采用 16 位地址直接寻址，存储器的寻访速度为 3 个时钟周期。

2.3.2　堆栈

堆栈是在内存 RAM 区专门开辟出来的按照“先进后出”的原则进行数据存取的一个工作区域，如图 2.8 所示。它主要用于子程序调用及返回和中断处理断点的保护及返回等。堆栈的最大容量范围限制在 2 K 字 RAM 内，即其地址为 0x07FF～0x0000。

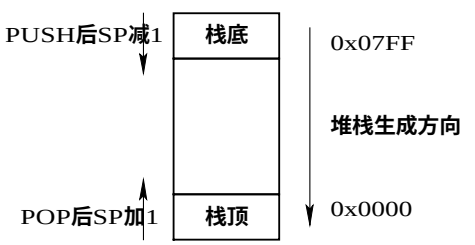


图 2.8　堆栈生成方向

注意：堆栈的生成方向向下(见图 2.8)，SPCE061A 复位后，SP 指针初始值为 0x07FF, 每执行一次 PUSH 指令 SP 指针减 1。

2.3.3 闪存 FLASH(闪速存储器)

32 K 字的内嵌式闪存被划分为 128 个页(每个页存储容量为 256 个字)，地址空间为 0x8000～0xFFFF，其中 0xFFF6～0xFFFF 为中断向量单元。CPU 正常工作时，均可通过程序写入或擦除。32 K 字的内嵌式闪存均可在 ICE 工作方式下编程写入或擦除。

1. **读存储单元操作**

芯片上电后，处于读存储单元状态，其操作与读 SRAM 相同。

2. **擦除操作**

在对闪存编程前，必须对其进行擦除。由于闪存采用模块分区的阵列结构，因此，各个存储模块(页)可以独立擦除。当给出的地址在某一模块地址范围内并且向命令用户接口写入模块擦除命令时，相应的模块就被擦除。要保证擦除操作的正确完成，必须考虑以下几个参数：

(1) **该闪速存储器的内部模块分区结构。**

(2) **每个模块分区的擦除时间。**

3. **编程操作**

闪速存储器芯片的编程操作是自动按字节编程的，既可以顺序写入，也可指定地址写入，编程操作时，注意芯片的编程时间，FLASH 程序空间为 0x8000～0xFFFF。Flash 命令用户接口地址为 0x7555,第 1 页地址范围是 0x8000～0x80FF,最后一页是 0xFF00～0xFFFF。

(1) **擦除一页的流程如下：**

① **给命令用户接口地址 0x7555 送 0xAAAA。**

② **给命令用户接口地址 0x7555 送 0x5511。**

③ **给要擦除页地址送任意数约 20 ms 即可完成擦除操作。**

例如：擦除第 6 页[0x8500～0x85FF]的流程如下：

① **给 0x7555 单元写 0xAAAA。**

② **给 0x7555 单元写 0x5511。**

③ **给 0x85XX 写 XXXX(X 为任意值保持 20 ms)。**

(2) **写入一个字的流程如下：**

① **给命令用户接口地址 0x7555 送 0xAAAA。**

② **给命令用户接口地址 0x7555 送 0x5533。**

③ **给写入字地址送数据约 40 μs 即可完成写入操作。**

例如：向 0x8000 单元写入 0xFFFF 的流程如下：

① **给 0x7555 单元写 0xAAAA。**

② **给 0x7555 单元写 0x5533。**

③ **给 0x8000 单元写 0xFFFF。**

(3) 写多个字的流程如下：

① 为命令用户接口地址 0x7555 送 0xAAAA。

② 为命令用户接口地址 0x7555 送 0x5544。

③ 写入字首地址送数据约 40 μs 即可完成 1 个字写入操作。

④ 为命令用户接口地址 0x7555 送 0x5544。

⑤ 写入字地址送数据等待 40 μs，循环操作即可完成多字的写入。

注意：上面所提到的延时等待是由硬件自动完成的，不需要软件延时。闪存的擦写过程如图 2.9 所示。

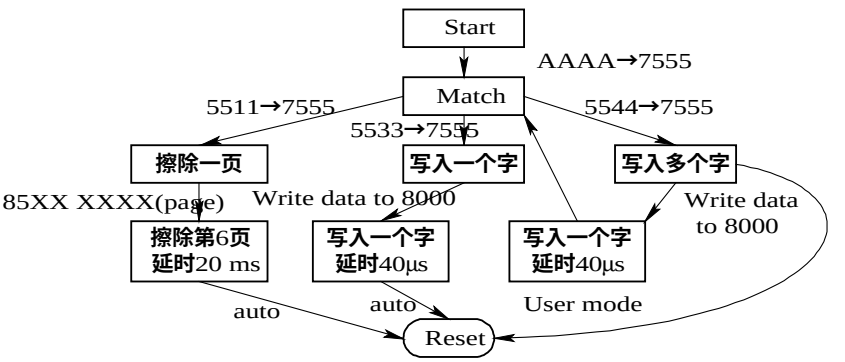


图 2.9　闪存的擦写过程

思　考　题

1. 简述 SPCE061A 单片机的主要性能。
2. 怎样构成 SPCE061A 单片机最小应用系统？其开发方式如何？
3. SPCE061A 单片机的外接时钟源频率是多少？
4. SPCE061A 单片机具有哪些寄存器？请分别说明其作用。
5. 如何判别运算过程中是否有溢出？
6. SPCE061A 单片机的堆栈区域是多少？
7. SPCE061A 单片机内存是如何分配的？请简述各部分作用。
8. 请简述 SPCE061A 单片机 FLASH 内存的编程过程。

第 3 章　SPCE061A 指令系统

3.1 概　　述

指令是 CPU 执行某种操作的命令。微处理器(MPU)或微控制器(MCU)所能识别全部指令的集合称为指令系统，是研制厂家在设计 CPU 时所赋予的功能。只有正确书写和使用指令，才能完成设计任务。因此，学习和掌握指令的功能与应用是程序设计的基础。本章将详细介绍 SPCE061A 单片机的寻址方式和指令系统。由于 SPCE061A 采用 μ'nSP™为内核，因此，μ'nSP™的指令集完全适用于 SPCE061A 单片机。本章将对 μ'nSP™指令系统作分类介绍。

3.1.1 指令分类

以 μ'nSP™为内核的单片机系列指令集按功能可划分为：

(1) 数据传送指令：立即数到寄存器、寄存器到寄存器、寄存器到存储器、存储器到寄存器的数据传送操作。

(2) 算术运算指令：加、减、乘等运算。

(3) 逻辑运算指令：与、或、异或、测试、移位等操作。

(4) 转移指令：条件转移、无条件转移、中断返回、子程序调用等操作。

(5) 控制指令：如开中断、关中断、有限冲击响应 FIR(Finite Impluse Response)滤波器的数据自由移动等操作。

3.1.2 寻址方式

寻址方式主要有以下几种：

(1) 立即寻址：操作数以立即数的形式出现。

例如，R1 = 0x1234 是把十六进制数 0x1234 赋给寄存器 R1。

(2) 直接寻址：通过存储器地址来访问存储器中的数据。

例如，R1 =[0x2222] 将 0x2222 单元的数据传送到寄存器 R1。

(3) 寄存器寻址：操作数在寄存器中。

例如，R1 = R2 是把寄存器 R2 中的数据赋给寄存器 R1。

(4) 寄存器间接寻址：寄存器中存储的是操作数的地址。

例如，R1 = [BP] 是把由 BP 指向的内存单元的数据传送到寄存器 R1。

(5) 变址寻址：操作数的地址由基址和偏移量共同提供。

例如，R1 = [BP+0x34]，BP 为基址，0x34 为偏移量，两者共同确定操作数存放的地址。

为了描述方便，对在指令系统叙述过程中所要用到的符号作一统一约定，如表 3.1 所示。

表 3.1　符 号 约 定

符号	功能说明
R1,R2,R3,R4,R5(BP)	通用寄存器
PC	程序计数器
CS，DS	段寄存器中的代码段选择字段和数据段选择字段
NZSC	寄存器中的四个标志位
SR	段寄存器
IM6	6 位立即数
IM16	16 位立即数
A6	6 位地址码
A16	16 位地址码
Rd	目的(destination)寄存器或存储器指针
Rs	源寄存器或存储器指针
→	数据传送符号
MR	由 R4 和 R3 组成的 32 位结果寄存器，R4 为高 16 位，R3 为低 16 位
&, |, ^	逻辑与、逻辑或、逻辑异或符号
{ }	可选项
[ ]	寄存器间接寻址标志
++, － －	指针单位字增量，字减量
ss，us	两个有符号数之间的操作，无符号数与有符号数之间的操作
Label	程序标号
FIR	Finite Impulse Response(有限冲击响应)，数字信号处理中的一种具有线性相位及任意幅度特性的数字滤波器算法
N	负标志，N=0 表示运算结果最高有效位为 0；N=1 表示最高有效位为 1
Z	零标志，Z=0 表示运算结果不为 0；Z=1 表示运算结果为 0
S	符号标志，S=0 表示结果不为负；S=1 表示结果为负数(2 的补数)
C	进位标志，C=0 表示运算过程中无进位或有借位产生；C=1 表示运算过程中有进位或无借位产生
//	注释符

3.2 数据传送指令

数据传送指令是把源操作数传送到指令所指定的目标地址。数据传送操作属复制性质，不是搬家性质，指令执行后源操作数保持不变，只是用源操作数代替目的操作数。

数据传送指令的通用格式如下：

<目的操作数> = <源操作数>

源操作数可以是立即数、寄存器寻址、寄存器间接寻址、直接寻址、变址寻址等。目的操作数可以是寄存器和直接地址。

下面按寻址方式介绍 SPCE061A 的数据传送指令，各个指令的执行周期数、指令长度等可参见表 3.2。

表 3.2 数据传送指令一览表(ISA1.1 指令集)

语法	指令长度(Word)	影响标志	周期数
Rd = IM16	2		4
Rd =IM6	1		2
Rd = [BP +IM6]			6
Rd = [A6]			5
Rd = [A16]	2		7
Rd = Rs	1		4
Rd = [Rs]			4
Rd = [Rs++]			
Rd = [++Rs]		N,Z	
Rd = [Rs−−]			
[BP + IM6] = Rs			6
[A6] = Rs			5
[A16] = Rs	2		7
[Rd] = Rs	1		6
[++Rd] = Rs			
[Rd−−] = Rs			
[Rd++] = Rs			

注意：数据传送指令影响标志位 N 和 Z。

1. 立即寻址

格式 1：Rd = IM16

说明：16 位立即数送入目标寄存器 Rd。

格式 2：Rd = IM6

说明：6 位立即数扩展为 16 位送入目标寄存器 Rd。例如：

R1=0xF001 //R1 的值变为 0xF001

2. 寄存器寻址

格式 1：Rd = Rs

说明：将源寄存器 Rs **的数据送给目标寄存器** Rd**。例如：**

```
R2=0xF001                    //R2 的值为 0xF001
R1=R2                        //R1 的值变为 0xF001
```

3. 直接寻址

格式 1：Rd = [A6]

说明：把 A6 **指定的存储单元数据读到** Rd **寄存器。**

格式 2：Rd = [A16]

说明：把 A16 **指定的存储单元数据读到** Rd **寄存器。**

格式 3：[A6] = Rs

说明：将源寄存器 Rs **中的数据送给以** A6 **为地址的存储单元。**

格式 4：[A16] = Rs

说明：把 Rs **数据存储到** A16 **指出的存储单元。例如：**

```
R1=0x0011                    //R1 的值为 0x0011
[0x0010]=R1                  //0x0010 单元的值变为 0x0011
```

4. 变址寻址

格式 1：Rd = [BP +IM6]

说明：把基址指针 BP **与** 6 **位的立即数的和作为地址，将该存储单元的数据读到** Rd **寄存器。**

格式 2：[BP + IM6] = Rs

说明：把 Rs **的值存储到基址指针** BP **与** 6 **位的立即数之和为地址的存储单元。例如：**

```
R1=0x0010
[BP+0x0002]=R1
```

5. 寄存器间接寻址

格式 1：[Rd] = Rs

说明：把 Rs **的数据存储到** Rd **的值所指的存储单元(**Rd **中存放的是操作数的地址)。**

格式 2：[++Rd] = Rs

说明：首先把 Rd **的值加** 1**，再将** Rs **的数据存储到** Rd **的值所指的存储单元。**

格式 3：Rd = [Rs++]

说明：读取 Rs **所指的存储单元的值存入** Rd**，而后** Rs **的值加** 1**。**

例 1　将 R3 **的值保存于** 0x25 **单元。**

```
R3 =0x5678                   //把 16 位立即数 0x5678 赋给 R3
```

方法 1：

```
[0x25]=R3                    //将 R3 的值存储于 0x25 存储单元(直接寻址)，0x25 单元的内容
                             //为 0x5678
```

方法 2：

```
R2 =0x25                    //立即数 0x25 送入 R2
[R2]=R3                     //将 R3 的值存储于 0x25 存储单元(寄存器间接寻址)，0x25 单元的
                            //内容为 0x5678
```

方法 3：

```
BP= 0x20                    //立即数 0x20 送入 BP
[BP+5]= R3                  //将 R3 的值存储于 0x25 存储单元(变址寻址)，0x25 单元的内容
                            //为 0x5678
```

例 2　将 0x25，0x26 **和** 0x27 **单元清零。**

方法 1：

```
R1 = 0
R2 = 0x25                   //立即数 0x25 送入 R2 中
[R2++] = R1                 //R1 的值存储于 0x25，存储单元 R2=R2+1
[R2++] = R1                 //R1 的值存储于 0x26，存储单元 R2=R2+1
[R2] = R1                   //R1 的值存储于 0x27
```

方法 2：

```
R1 = 0
R2 = 0x27
[R2−−] =R1                  //R1 的值存储于 0x27，存储单元 R2=R2−1
[R2−−] =R1                  //R1 的值存储于 0x26，存储单元 R2=R2−1
[R2] = R1                   //R1 的值存储于 0x25
```

方法 3：

```
R1 = 0                      //影响标志位 Z=1,N=0
R2 = 0x24                   //立即数 0x24  送入 R2 中
[++R2]= R1                  //R2=R2 +1，R2=0x25，而后 R1 的值存储于 0x25 单元
[++R2]=R1                   //R2=R2 +1，R2=0x26，而后 R1 的值存储于 0x26  单元
[++R2]=R1                   //R2=R2 +1，R2=0x27，而后 R1 的值存储于 0x27 单元
```

例 3　用不同方式读取存储器的值。

```
BP = 0x20                   //将立即数 0x20  赋给 BP
R1 = [BP + 5]               //BP+5=0x25，则读 0x25 单元数据到 R1 中
R1 = [0x2345]               //读 0x2345 单元数据到 R1 中
R1 = [BP]                   //读 0x20 单元的数据到 R1 中
R1 = [BP++]                 //读 0x20  单元的数据到 R1 中，并使 BP=0x20+1
R1 = [BP−−]                 //读取 0x21 单元的数据到 R1 中，并使 BP=0x21−1
R1 = [++BP]                 //修改 BP=0x20＋1，读 0x21 单元的数据到 R1
```

6. 堆栈操作

除以上介绍的指令外，堆栈(stack)**操作属于一种特殊的数据传送指令，在此也一并作以介绍。堆栈操作如图** 3.1 **所示。**

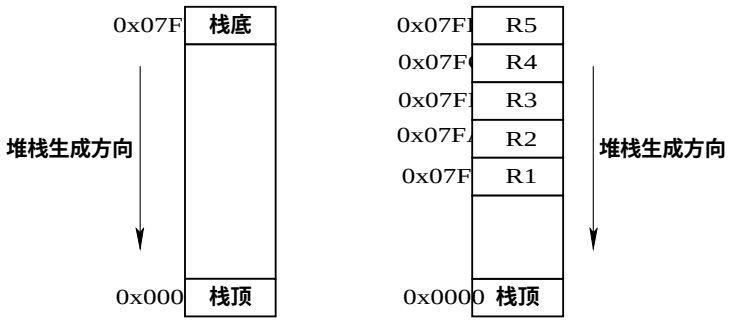


图 3.1　堆栈操作

堆栈指针 SP 总是指向栈顶的第一个空项，压入一个字后 SP 减 1，将多个寄存器同时压栈，总是序号最高的寄存器先入栈，然后依次压入序号较低的寄存器。所以，执行指令

PUSH R1,R4 TO [SP]

与

PUSH R4,R1 TO [SP]

是等效的，即将 R4，R3，R2，R1 压栈，在数据出栈前 SP 加 1，总是先弹出入栈指令中序号最低的寄存器，而后依次弹出序号较高的寄存器，即先进后出，后进先出。

堆栈操作的指令格式：　PUSH　Rx，Ry　TO [SP]

POP　Rx，Ry　FROM [SP]

说明：Rx，Ry 可以是 R1～R4、BP、SP 或 PC 中的任意两个或一个，执行后将 Rx、Ry 的序列寄存器压栈，或将堆栈中的数据弹入 Rx、Ry 序列寄存器中。压栈操作不影响标志位，出栈操作影响 N 和 Z 标志。当 Rx、Ry 中含有 SR 时，所有标志位都会改变。压栈、出栈操作的执行周期为 3n + 4，若出栈操作的目的寄存器中含有 PC 时，执行周期为 3n+6，其中 n 是压栈数据的个数。压栈和出栈的指令长度均为 1 字长。例如：

```
PUSH R1,R5 to [SP]          //将 R5,R4,R3,R2,R1 压栈
PUSH R2,R2 to [SP]          //将 R2 压栈
PUSH R3 to [SP]             //将 R3 压栈
POP R3 from [SP]            //R3 出栈
POP R2,R2 from [SP]         //R2 出栈
POP R1,R5 from [SP]         //R1,R2,R3,R4,R5 出栈
```

3.3　算术运算指令

SPCE061A 单片机的算术运算主要包括加、减、乘及 n 项内积运算。加、减运算按是否带进位分为不带进位和带进位的加减运算。带进位加减运算的格式及寻址方式与不带进位的加减运算类似，因此，本节详细介绍不带进位的加减运算，而对带进位的加减运算只作简要说明。

3.3.1　加法运算指令

加法运算影响标志位 N，Z，S 和 C，寻址方式有以下几种。

1. 立即寻址(不带进位)

格式 1：Rd += IM6 **或** Rd = Rd + IM6

操作：Rd + IM6 →Rd

说明：Rd **的数据与** 6 **位(高位扩展成** 16 **位)立即数相加，结果送** Rd。

格式 2：Rd = Rs + IM16

操作：Rs + IM16 →Rd

说明：Rd **的数据与** 16 **位的立即数相加，结果送** Rd。**例如：**

```
R1=0x0099                    //R1 的值为 0x0099
R1+=0x0001                   //R1 的值变为 0x009A
R1+=0x0012                   //R1 的值变为 0x00AC
```

2. 直接寻址

格式 1：Rd += [A6] **或** Rd = Rd + [A6]

操作：Rd + [A6] →Rd

说明：Rd **的数据与** 6 **位地址指定的存储单元中的数据相加，结果送** Rd。

格式 2：Rd = Rs + [A16]

操作：Rs + [A16] →Rd

说明：Rs **的数据与** 16 **位地址指定的存储单元中的数据相加，结果送** Rd。**例如：**

```
R2=0x0010                    //R2 的值为 0x0010
[0x0088]=R2                  //把 0x0010 送到内存单元 0x0088
R1=0xF099                    //R1 的值为 0xF099
R1+=[0x0088]                 //R1 的值变为 0xF0A9
```

3. 变址寻址

格式：Rd += [Bp + IM6] **或** Rd = Rd + [BP + IM6]

操作：Rd + [Bp + IM6] →Rd

说明：取基址指针 BP **与** 6 **位立即数的和作为存储单元的地址，存储单元中的数据与** Rd **相加，结果送** Rd **寄存器。例如：**

```
R1=0x0010                    //R1 的值变为 0x0010
R2=0x0090                    //R2 的值变为 0x0090
[0x0015]=R2                  //R2 的值送到 0x0015 内存单元
R1+=[BP+0x0015]              //R1 的值变为 0x00A0(设 BP=0x0000)
```

4. 寄存器寻址

格式: Rd += Rs

操作: Rd + Rs →Rd

说明：Rd **与** Rs **的数据相加，结果送** Rd。**例如：**

```
R1=0x0010                    //R1 的值为 0x0010
R2=0x0032                    //R2 的值为 0x0032
R1+=R2                  //R1 的值变为 0x0042
```

5. **寄存器间接寻址**

格式 1：Rd += [Rs]

操作：Rd + [Rs] → Rd

说明：Rd **的数据与** Rs **所指定的存储单元中的数据相加，结果送** Rd。

格式 2：Rd += [Rs++]

操作：Rd + [Rs] →Rd, Rs + 1→ Rs

说明：Rd **的数据与** Rs **所指定的存储单元中的数据相加，结果送** Rd，**并且** Rs=Rs+1。

格式 3：Rd += [Rs– –]

操作：Rd + [Rs] → Rd, Rs – 1→ Rs

说明：Rd **的数据与** Rs **所指定的存储单元中数据相加，结果送** Rd，**且** Rs=Rs–1。

格式 4：Rd += [++Rs]

操作：Rs + 1→ Rs, Rd + [Rs] → Rd

说明：首先修改 Rs=Rs+1，Rd **的数据与** Rs **所指定的存储单元中的数据相加，结果送** Rd。**例如：**

```
R1=0x0010                  //R1 的值为 0x0010
R2=0x0020                  //R2 的值为 0x0020
[0x0010]= R2               //R2 的值送到内存单元 0x0010 中
R2=0x0010                  //R2 的值为 0x0010
R1+=[R2++]                 //R1 的值变为 0x0030，同时 R2 的值变为 0x0011
```

3.3.2　减法运算指令

同不带进位的加法运算一样，不带进位的减法运算同样可分为立即寻址、直接寻址、寄存器寻址、寄存器间接寻址等方式。减法运算影响标志位 N，Z，S **和** C。

1. **立即寻址**

格式 1：Rd – = IM6 **或** Rd = Rd – IM6

操作：Rd – IM6→ Rd

说明：Rd **的数据减去** 6 **位立即数，结果送** Rd。

格式 2：Rd = Rs – IM16

操作：Rs – IM16→ Rd

说明：Rs **的数据减去** 16 **位立即数，结果送** Rd。**例如：**

```
R1=0x0010                  //R1 的值为 0x0010
R2=0x0001                  //R2 的值为 0x0001
R1– =R2                    //R1 的值为 0x000F
```

2. **直接寻址**

格式 1：Rd – = [A6] **或** Rd = Rd – [A16]

操作：Rd – [A6] → Rd

说明：Rd **的数据减去**[A6]**存储单元中的数据，结果送** Rd。

格式 2：Rd = Rs –[A16]
操作：Rs – [A16] →Rd
说明：Rs **的数据减去**[A16]**存储单元中的数据，结果送** Rd。
例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0002                  //R1 的值为 0x0002,N=0,Z=0,S=0,C=1
[0x0020]=R1                //把 R1 的值送到内存单元 0x0020 中，标志位不变
R2=0x0001                  //R2 的值为 0x0001,N=0,Z=0,S=0,C=1
R2– =[0x0020]              //R2 的值变为 0xFFFF，C 为 0 表明运算过程产生了借位,
                           //N=1,Z=0,S=1
```

3. 变址寻址

格式：Rd – = [BP + IM6] **或** Rd = Rd – [BP + IM6]
操作：Rd – [BP + IM6] →Rd
说明：Rd **的值减去基址加变址指定的存储单元的值，结果送** Rd。**例如：**

```
R1=0x8031                  //R1 的值为 0x8031
R2=0x0020                  //R2 的值为 0x0020
[0x0010]=R2                //把 0x0020 送到地址单元 0x0010
R1– =[BP+0x0010]           //R1 的值变为 0x8011(设 BP=0x0000)
```

4. 寄存器寻址

格式：Rd–=Rs
操作：Rd–Rs→Rd
说明：寄存器 Rd **的数据减去** Rs **的数据结果送** Rd **寄存器。例如：**

```
R1 =0x31E0                 //R1 的初值为 0x31E0
R2 =0x31DF                 //R2 的初值为 0x31DF
R1– =R2                    //R1 的值变为 0x0001
```

5. 寄存器间接寻址

格式 1：Rd–= [Rs]
操作：Rd– [Rs]→Rd
说明：Rd **的数据与** Rs **所指定的存储单元中的数据相减，结果送** Rd。
格式 2：Rd – = [Rs++]
操作：Rd – [Rs] →Rd, Rs+1→ Rs
说明：Rd **的数据与** Rs **所指定的存储单元中的数据相减，结果送** Rd，**且** Rs＝Rs+1。
格式 3：Rd – = [Rs– –]
操作：Rd –[Rs] → Rd, Rs–1→ Rs
说明：Rd **的数据与** Rs **所指定的存储单元中的数据相减，结果送** Rd, **且** Rs＝Rs–1。
格式 4：Rd–= [++Rs]
操作：Rs + 1→Rs, Rd– [Rs] →Rd
说明：Rs **的值加** 1，Rd **的数据与** Rs **所指定单元的数据相减，结果送** Rd。

3.3.3　带进位的加减法运算指令

由于带进位的加减运算与不带进位的加减运算在寻址方式、周期数、指令长度、影响标志位等均相同，格式相似，故这里只给出指令格式供读者参考。

带进位的加法指令格式：

Rd += IM6,Carry
Rd = Rd + IM6,Carry
Rd = Rs + IM16,Carry
Rd += [BP + IM6] ,Carry
Rd = Rd + [BP + IM6] ,Carry
Rd += [A6] ,Carry
Rd = Rd + [A6] ,Carry
Rd = Rs + [A16] ,Carry
Rd += Rs,Carry
Rd += [Rs] ,Carry
Rd += [++Rs] ,Carry
Rd += [Rs– –],Carry
Rd += [Rs++],Carry

带进位的减法指令格式：

Rd – = IM6,Carry
Rd = Rd – IM6,Carry
Rd = Rs – IM16,Carry
Rd – = [BP + IM6] ,Carry
Rd = Rd – [BP + IM6] ,Carry
Rd – = [A6] ,Carry
Rd = Rd – [A6] ,Carry
Rd = Rs – [A16] ,Carry
Rd – = Rs,Carry
Rd – = [Rs] ,Carry
Rd – = [++Rs] ,Carry
Rd – = [Rs– –],Carry
Rd – = [Rs++],Carry

3.3.4　乘法运算指令

Rd **和** Rs **可用** R1～R2 **和** BP；MR **由** R4、R3 **构成，**R4 **为高位字，**R3 **为低位字。不影响标志位。**

格式 1：MR = Rd ∗ Rs **或** MR = Rd ∗ Rs,ss

功能：Rd ∗ Rs →MR

说明：表示两个有符号数相乘，结果送 MR **寄存器。**

格式 2：MR = Rd * Rs,us

功能：Rd * Rs →MR

说明：表示无符号数与有符号数相乘，结果送 MR **寄存器。**

例如，计算一年 365 **天共有多少小时，结果存放** R4(**高位**)、R3(**低位**)。

```
R1 = 365                //R1 的值为 0x016D
R2 = 24                 //R2 的值为 0x0018
MR = R1 * R2,us         //计算乘积，结果 R3 的值为 0x2238，R4 的值为 0x0000
```

3.3.5 内积运算指令

内积运算指令不影响标志位。

格式：MR = [Rd] * [Rs] {,ss} {,n}

功能：Rd **与** Rs **寄存器内容作为地址的存储器内有符号数与有符号数之间或无符号数与有符号字数之间进行** n **项内积运算，结果存入** MR。**符号的缺省选择为** ss，**即有符号数据之间的运算。**n **的取值为** 1～16，**缺省值为** 1。**内积运算操作如图** 3.2 **所示，其中** Rd **与** Rs **可为寄存器** R1～R2 **或** BP。

执行周期：10n+6 **个周期**

说明：当 FIR_MOV ON **时，允许** FIR **运算过程中数据自由移动。为新样本取代旧样本进行数据移动准备：**Xn–4=Xn–3，Xn–3=Xn–2，Xn–2=Xn–1。**如图** 3.2 **所示，当完成一次内积运算后** X1、X2、X3 **自动右移，**X4 **移出，在数字信号处理中十分有用。比如你要计算连续** 4 **次采样值的平均值，可将采样值放到** X1～X4 **中，加权系数放到** C **中，然后完成**[Rd] * [Rs] {,ss} {,4}**运算，再求平均值。当** FIR_MOV ON **时，运算完后** X1、X2、X3 **自动右移，**X4 **移出。这样我们就可以把下一次的采样值** X0 **放到原** X1 **的位置，下一次直接完成**[Rd] * [Rs] {,ss} {,4}**运算，即** X0 –X3 **的运算，不必另外移动** Xn。

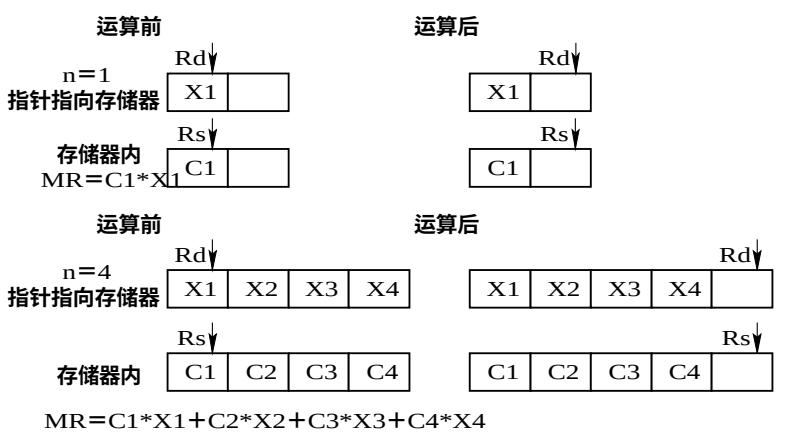


图 3.2 **内积运算操作示意图**

注意：

(1) Rs **和** Rd **中任意一个都不能为** R3 **或** R4；

(2) Rs **和** Rd **不能为同一个寄存器。**

3.3.6　比较指令

比较指令执行两数的减法操作，不存储运算结果，只影响标志位 N、Z、S 和 C。下面按寻址方式分别介绍比较指令。

1. 立即寻址

格式 1：CMP Rd,IM6
说明：将 Rd 的值与 6 位立即数相比较。
格式 2：CMP Rd,IM16
说明：将 Rd 的值与 16 位立即数相比较。

2. 直接寻址

格式 1：CMP Rd, [A6]
说明：将 Rd 的值与 A6 指定地址单元的数据相比较。
格式 2：CMP Rd, [A16]
说明：将 Rd 的值与 A16 指定地址单元的数据相比较。

3. 寄存器寻址

格式：CMP Rd，Rs
说明：将 Rd 与 Rs 的值相减，只影响标志位，不存储结果。

4. 变址寻址

格式：CMP Rd，[BP + IM6]
说明：将 Rd 与[BP + IM6]指定地址单元的数据相比较。

5. 寄存器间接寻址

格式 1：CMP Rd，[Rs]
说明：将 Rd 的值与寄存器 Rs 指定存储单元的数据相比较。
格式 2：CMP Rd，[Rs+ +]
说明：将 Rd 的值与寄存器 Rs 指定存储单元的数据相比较，且 Rs＝Rs+1。
格式 3：CMP Rd，[Rs− −]
说明：将 Rd 的值与寄存器 Rs 指定存储单元的数据相比较，且 Rs＝Rs−1。
格式 4：CMP Rd，[++Rs]
说明：Rs 加 1，再将 Rd 的值与寄存器 Rs 指定存储单元的数据相比较，结果只影响标志位。

3.4　逻辑运算指令

3.4.1　逻辑与指令

逻辑与运算影响标志位 N 和 Z，寻址方式有以下几种。

1. 立即寻址

格式 1：Rd &= IM6 **或** Rd = Rd & IM6
功能：Rd & IM6→Rd
说明：将 Rd **的数据与** 6 **位立即数进行逻辑与操作，结果送** Rd **寄存器。**
格式 2：Rd = Rs & IM16
功能：Rs & IM16→Rd
说明：将 Rs **的数据与** 16 **位立即数进行逻辑与操作，结果送** Rd **寄存器。**
例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0010                    //R1 的初值为 0x0010，Z=0
R1&=0x000F                   //结果为 0，标志位 Z 由 0 变为 1
```

2. 直接寻址

格式 1：Rd &=[A6] **或** Rd=Rd&[A6]
功能：Rd & [A6] →Rd
说明：将 Rd **和** A6 **指定存储单元的数据进行逻辑与操作，结果送** Rd **寄存器。**
格式 2：Rd = Rs & [A16]
功能：Rs & [A16] →Rd
说明 :将 Rs **中的数据和** A16 **指定存储单元中的数据进行逻辑与操作,结果送** Rd **寄存器。**
例如，设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0010                    //R1 赋值为 0x0010，Z=0，N=0
R2=0xFFFF                    //R2 赋值为 0xFFFF，Z=0，N=1
[0x000F]=R2
R1&=[0x000F]                 //执行后 R1=0x0010，标志位 Z=0，N=0
```

3. 寄存器寻址

格式：Rd &= Rs **或** Rd=Rd&Rs
功能：Rd & Rs→Rd
说明：将 Rd **和** Rs **的数据进行逻辑与操作，结果送** Rd **寄存器。**
例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x00FF                    //R1 的初值为 0x00FF，Z=0，N=0
R2=0XFFFF                    //R1 的初值为 0XFFFF，Z=0，N=1
R1&=R2                       //结果为 0x00FF 标志位 N 变为 0，Z=0
```

4. 寄存器间接寻址

格式 1：Rd &= [Rs]
功能：Rd & [Rs] →Rd
说明: **将** Rd **的数据与** Rs **指定的存储单元数据进行逻辑与操作，结果送** Rd **寄存器。**
格式 2：Rd &= [Rs++]
功能：Rd & [Rs]→Rd，Rs+1→Rs
说明：将 Rd **的数据与** Rs **指定的单元的数据进行逻辑与操作，结果送** Rd **寄存器，并**

使 Rs 加 1。

格式 3：Rd &= [Rs−−]

功能：Rd & [Rs]→Rd，Rs–1→Rs

说明：将 Rd 的数据与 Rs 指定的单元的数据进行逻辑与操作，结果送 Rd 寄存器，并使 Rs 减 1。

格式 4：Rd &= [++Rs]

功能：Rs+1→Rs，Rd&[Rs]→Rd

说明：Rs 加 1,将 Rd 的数据与 Rs 指定的单元的数据进行逻辑与操作,结果送 Rd 寄存器。

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x00FF                    //R1 的初值为 0x00FF，Z=0，N=0
R2=0XFFFF                    //R2 的初值为 0XFFFF，Z=0，N=1
[0x0001]=R2
R2=0x0001
R1&=[R2]                     //结果为 0x00FF，标志位 N 变为 0，Z=0
```

3.4.2　逻辑或指令

逻辑或影响标志位 N 和 Z，寻址方式有以下几种。

1. 立即寻址

格式 1：Rd|=IM6 **或** Rd=Rd|IM6

功能：Rd|IM6→Rd

说明：将 Rd 的数据与 6 位立即数进行逻辑或操作，结果送 Rd 寄存器。

格式 2：Rd=Rs|IM16

功能：Rs|IM16→Rd

说明：将 Rs 的数据与 16 位立即数进行逻辑或操作，结果送 Rd 寄存器。

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x00FF                    //R1 的初值为 0x00FF，Z=0，N=0
R1|=0xF000                   //R1 的值变为 0xF0FF, Z=0,N=1
```

2. 直接寻址

格式 1：Rd|=[A6]**或** Rd=Rd|[A6]

功能：Rd|[A6]→Rd

说明：将 Rd 和 A6 指定单元的数据进行逻辑或操作，结果送 Rd 寄存器。

格式 2：Rd =Rs|[A16]

功能：Rs|[A16]→Rd

说明：将 Rs 的数据和 A16 指定单元的数据进行逻辑或操作，结果送 Rd 寄存器。

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x00FF          //R1 的初值为 0x00FF，Z=0，N=0
R1|=[0x0009]    //R1 值变为 0x00FF, Z=0，N=0，假设[0x0009]的值为 0
```

3. **寄存器寻址**

格式：Rd|=Rs

功能：Rd|Rs →Rd

说明：将 Rd **和** Rs **的数据进行逻辑或操作，结果送** Rd **寄存器。**

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0000                    //R1 的初值为 0x0000，Z=1，N=0
R2=0xFFF0
R1|=R2                       //R1 的值变为 0xFFF0，Z=0，N=1
```

4. **寄存器间接寻址**

格式 1：Rd|= [Rs]

功能：Rd|[Rs] →Rd

说明：将 Rd **的数据与** Rs **指定单元的数据进行逻辑或操作，结果送** Rd **寄存器。**

格式 2：Rd |= [Rs++]

功能：Rd | [Rs] →Rd，Rs + 1→Rs

说明：将 Rd **的数据与** Rs **指定单元的数据进行逻辑或操作，结果送** Rd **寄存器，并使** Rs **加** 1。

格式 3：Rd|= [Rs−−]

功能：Rd|[Rs]→Rd，Rs–1→Rs

说明：将 Rd **的数据与** Rs **指定单元的数据进行逻辑或操作，结果送** Rd **寄存器，并使** Rs **减** 1。

格式 4：Rdl= [++Rs]

功能：Rs+1→Rs，Rdl[Rs]→Rd

说明：首先将 Rs **的值加** 1，**然后将** Rd **与** Rs **指定单元的数据进行逻辑或操作，结果送** Rd **寄存器。**

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0000；                  //R1 的初值为 0x0000，Z=1，N=0
R2=0xFFFF；                  //把 0xFFFF 送到地址单元[0x0002]中
[0x0002]=R2；
R2=0x0002；                  //R2 的值为 0x0002，Z=0，N=0
R1|=[R2]；                   //R1 的值变为 0xFFFF，Z=0，N=1
```

3.4.3 逻辑异或指令

逻辑异或影响标志位 N **和** Z，**寻址方式有以下几种。**

1. **立即寻址**

格式 1：Rd^=IM6 **或** Rd= Rd^IM6

功能：Rd^IM6→Rd

说明：将 Rd **的数据与** 6 **位立即数进行逻辑异或操作，结果送** Rd **寄存器。**

格式 2：Rd=Rs^IM16

功能：Rs^IM16→Rd

说明：将 Rs **的数据与** 16 **位立即数进行逻辑异或操作，结果送** Rd **寄存器。**

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0F00                    //R1 的初值为 0x0F00，Z=0，N=0
R1^=0x0FFF                   //R1 的值变为 0x00FF，Z=0，N=0
```

2. 直接寻址

格式 1：Rd^=[A6]**或** Rd＝Rd^[A6]

功能：Rd ^ [A6] →Rd

说明：将 Rd **和** A6 **指定存储单元的数据进行逻辑异或操作，结果送** Rd **寄存器。**

格式 2：Rd=Rs^[A16]

功能：Rs^[A16] →Rd

说明：将 Rs **的数据和** A16 **指定存储单元的数据进行逻辑异或操作，结果送** Rd **寄存器。**

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0F00                    //R1 的初值为 0x0F00，Z=0，N=0
R2=0x0FF0
[0x0010]=R2
R1^=[0x0010]                 //R1 的值变为 0x00F0，Z=0，N=0
```

3. 寄存器寻址

格式：Rd ^= Rs

功能：Rd ^ Rs→Rd

说明：将 Rd **和** Rs **的数据进行逻辑异或操作，结果送** Rd **寄存器。**

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0E01                    //R1 的初值为 0x0E01，Z=0，N=0
R2=0x0FF1                    //R2 的初值为 0x0FF1，Z=0，N=0
R1^=R2                       //R1 的值变为 0x01F0，Z=0，N=0
```

4. 寄存器间接寻址

格式 1：Rd^= [Rs]

功能：Rd^[Rs]→Rd

说明：将 Rd **的数据与** Rs **指定的存储单元的数据进行逻辑异或操作，结果送** Rd **寄存器。**

格式 2：Rd^= [Rs++]

功能：Rd^[Rs] →Rd，Rs + 1→Rs

说明：将 Rd **的数据与** Rs **指定的存储单元的数据进行逻辑异或操作，结果送** Rd **寄存器，并使** Rs **加** 1。

格式 3：Rd^=[Rs−−]

功能：Rd^[Rs]→Rd，Rs–1→Rs

说明：将 Rd **的数据与** Rs **指定的存储单元中的数据进行逻辑异或操作，结果送** Rd **寄存器，并使** Rs **减** 1。

格式 4：Rd^=[++Rs]

功能：Rs+1→Rs , Rd^[Rs]→Rd

说明：首先使 Rs 加 1，然后 Rd 的数据与 Rs 指定的存储单元中的数据进行逻辑异或操作，结果送 Rd 寄存器。

例如，假设开始时的标志位为 N=0，Z=1，S=0，C=1。

```
R1=0x0000                    //R1 的初值为 0x0000，Z=1，N=0
R2=0xFFFF                    //R2 的初值为 0xFFFF，N=1，Z=0
[0x0002]=R2
R2=0x0002
R1^=[R2]                     //R1 的值变为 0xFFFF, Z=0，N=1
```

3.4.4 测试指令

测试指令是将两个数进行逻辑与操作，不存储结果，只影响 N 和 Z 标志位。

1. 立即寻址

格式 1：TEST Rd，IM6

说明：将 Rd 与 IM6 进行逻辑与操作，不存储结果，只影响 N 和 Z 标志位。

格式 2：TEST Rd，IM16

说明：将 Rd 与 IM16 进行逻辑与操作，不存储结果，只影响 N 和 Z 标志位。

例如，假设初始时标志位分别为 N=0，Z=1，S=0，C=1。

```
        R1=0x0E01            //R1 的初值为 0x0E01,N=0,Z=0,S=0,C=1
        TEST R1,0x0000       //测试 R1 和 0x0000 相与的结果,N=0,Z=1,S=0,C=1
        JZ loop1             //测试结果为 0，跳转到 loop1
        R2=0x0020
        ⋮
loop1:  R1=0x0001            //标号
```

2. 直接寻址

格式 1：TEST Rd, [A6]

说明：将 Rd 与 A6 指定存储单元的数据进行逻辑与操作，不存储结果，只影响 N 和 Z 标志位。

格式 2：TEST Rd, [A16]

说明：将 Rd 与 A16 指定存储单元的数据进行逻辑与操作，不存储结果，只影响 N 和 Z 标志。

例如，假设初始时标志位分别为 N=0，Z=1，S=0，C=1。

```
R1=0x0E01            //R1 的初值为 0x0E01,N=0,Z=0,S=0,C=1
R2=0x0011            //将 0x0011 送到内存单元[0x0000]中
[0x0000]=R2
TEST R1,[0x0000]     //R1 和 0x0000 单元的值相与，此时 N=0,Z=0,S=0,C=1
JNZ loop1            //Z 为 0 时跳转到 loop1,(此时测试结果不为 0)
```

```
        NOP
        NOP
        ⋮
loop1: R1=0x0000
```

3. 变址寻址

格式：TEST Rd,[BP+IM6]

说明：将 Rd **与** BP+IM6 **指定存储单元的数据进行逻辑与操作，不存储结果，只影响** N **和** Z **标志位。**

例如，假设初始状态标志位分别为 N=0，Z=1，S=0，C=1。

```
        R1=0x0E01                  //R1 的初值为 0x0E01,N =0,Z=0,S=0,C=1
        R2=0x0011                  //将 0x0011 送到内存单元[0x0000]中
        [0x0000]=R2
        TEST R1,[BP+0x0000]        //已知 BP 的值为 0，R1 和 BP+0x0000 单元的值相与，
                                   //N=0,Z=0,S=0,C=1
        JNZ loop1                  //测试结果不为 0 时跳转到 loop1
        NOP
        NOP
        ⋮
loop1:  R1+=1
```

4. 寄存器寻址

格式：TEST Rd,Rs

说明：将 Rd **与** Rs **的数据进行逻辑与操作，不存储结果，只影响** N **和** Z **标志位。**

例如，假设初始时标志位分别为 N=0，Z=1，S=0，C=1。

```
        R1=0x0000                  //R1 的初值为 0x0000,N=0,Z=1,S=0,C=1
        R2=0x1111                  //R2 的初值为 0x1111,N=0,Z=0,S=0,C=1
        TEST R1,R2                 //测试 R1 和 R2 的相与的结果,N=0,Z=1,S=0,C=1
        JZ loop1                   //测试结果为 0 时跳转到 loop1
        NOP
        ⋮
loop1:  R1-=1                      //标号
```

5. 寄存器间接寻址

格式 1：TEST Rd,[Rs]

说明：将 Rd **与**[Rs] **指定存储单元中的数据进行逻辑与操作，不存储结果，只影响** N **和** Z **标志位。**

格式 2：TEST Rd,[Rs++]

说明：将 Rd **与**[Rs]**指定存储单元中的数据进行逻辑与操作，不存储结果，只影响** N **和** Z **标志位，并且使** Rs **加** 1。

格式 3：TEST Rd,[Rs−−]

说明：将 Rd **与** Rs **指定存储单元中的数据进行逻辑与操作，不存储结果，只影响** N **和** Z **标志位，并且使** Rs **减** 1。

格式 4：TEST Rd,[++Rs]

说明：首先 Rs **加** 1，**然后** Rd **与** Rs **指定存储单元中的数据进行逻辑与操作，不存储结果，只影响** N **和** Z **标志位。**

例如，假设初始时标志位分别为 N=0，Z=1，S=0，C=1。

```
        R1=0x0000              //R1 的初值为 0x0000,N=0,Z=1,S=0,C=1
        R2=0x1111              //[0x0001]的初值为 0x1111,N=0,Z=0,S=0,C=1
        [0x0001]=R2
        R2=0x0001
        TEST R1,[R2]           //测试 R1 和[R2]相与的结果,N=0,Z=1,S=0,C=1
        JZ loop1               //测试结果为 0 时跳转到 loop1
        Nop
        ⋮
loop1: R1=0x0000               //标号
```

3.4.5 移位操作指令

SPCE061A **移位操作指令包括逻辑左移、逻辑右移、循环左移、循环右移、算术右移等。移位的同时还可进行其它运算，如加、减、比较、取负、与、或、异或、测试等。**

指令长度为 1， **指令周期为** 3/8，**影响** N **和** Z **标志位。移位操作每条指令可以移** 1～4 **位，**SB **的** S3～S0 **参与移位操作。**SB **是** μ'nSP™**内核中的移位缓冲器。**

1. **逻辑左移**(LSL)

格式：Rd＝Rs LSL n

说明：对 Rs **进行** n(**可设为** 1～4)**位逻辑左移，将** Rs **高** n **位移入** SB **寄存器，同时** Rs **的低** n(1～4)**位用** 0 **补足，结果送** Rd **寄存器。**

例如，逻辑左移 3 **位，寄存器移位前的状态如下：**

SB

S3	S2	S1	S0

RS

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

寄存器移位后的状态如下:

SB

S0	B15	B14	B13

RD

B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0	0	0	0

```
程序:   R1=0xF00F              //R1 初值为 0xF00F
        R1=R1 LSL 3            //R1 左移 3 位后的值变为 0x8078
```

2. **逻辑右移**(LSR)

格式：Rd=Rs LSR n

说明: **对** Rs **进行** n(**可设为** 1～4)**位逻辑右移，将** Rs **低** n **位移入** SB **寄存器，同时** Rs **的高** n(1～4)**位用** 0 **补足，结果送** Rd **寄存器。**

例如，逻辑右移 3 **位，寄存器移位前的状态如下：**

SB

S3	S2	S1	S0

RS

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

寄存器移位后的状态如下:

SB

0	0	0	S3

RD

S2	S1	S0	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3

程序：　R1=0xF00F　　//R1 **的初值为** 0xF00F

R1=R1 LSR 3　　//R1 **右移** 3 **位后的值变为** 0x1E01

3. **循环左移**(ROL)

格式：Rd＝Rs ROL n

说明：对 Rs **进行** n(**可设为** 1～4)**位循环左移，将** Rs **的高** n **位移入** SB **寄存器，同时将** SB **寄存器的高** n **位移入** Rs **的低** n **位，结果送** Rd **寄存器。**

例如，循环左移 1 **位，寄存器移位前的状态如下：**

SB

S3	S2	S1	S0

RS

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

寄存器移位后的状态如下:

SB

S2	S1	S0	B15

RD

B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0	S3

程序：　R1=0x0010　　//R1 **的初值为** 0x0010

R1=R1 ROL 1　　//R1 **循环左移** 1 **位后的值变为** 0x0020

4. **循环右移**(ROR)

格式：Rd＝ Rs ROR n

说明：对 Rs **进行** n(**可设为** 1～4)**位循环右移，将** Rs **的低** n **位移入** SB **寄存器，同时将** SB **寄存器的低** n **位移入** Rs **的高** n **位，结果送** Rd **寄存器。**

例如，循环右移 1 **位，寄存器移位前的状态如下：**

SB

S3	S2	S1	S0

RS

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

寄存器移位后的状态为:

SB

B0	S3	S2	S1

RD

S0	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1

5. **算术右移** ASR

格式：Rd＝ Rs ASR n

说明：将 Rs **算术右移** n(**可设为** 1～4)**位，将** Rs **的低** n **位移入** SB **寄存器，并对最高有效位进行符号扩展(保持原符号不变)，结果送** Rd **寄存器。该指令适用于有符号数的移位操作。**

例如，算术右移 3 **位。**

R1=0xF111 //R1 **的初值为** 0xF111

R1=R1 ASR 3 //R1 **移位后的值为** 0xFE22

SB

S3	S2	S1	S0

RS

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

寄存器移位后的状态如下:

SB

B2	B1	B0	S3

RD

E2	E1	E0	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3

注意：E2，E1，E0 **是** RS **的最高有效位的符号扩展位。**

6. **多功能移位指令**

SPCE061A **在进行移位的同时，还可进行其它运算，现以算术右移为例说明如下。(先移位后完成相应的运算)**

格式 1：Rd += Rs ASR n {,Carry}

说明：将 Rs **移位后的结果与** Rd **带进位相加，结果送** Rd **寄存器。**

格式 2：Rd – = Rs ASR n {,Carry}

说明：将 Rs **移位后的结果与** Rd **带进位相减，结果送** Rd **寄存器。**

格式 3：CMP Rd, Rs ASR n

说明：将 Rs **移位后的结果与** Rd **相比较。**

格式 4：Rd = – Rs ASR n

说明：取 Rs **移位后结果的负值，送** Rd **寄存器。**

格式 5：Rd&= Rs ASR n

说明：将 Rs **移位后的值与** Rd **逻辑与，结果送** Rd **寄存器。**

格式 6：Rd|= Rs ASR n

说明：将 Rs **移位后的值与** Rd **逻辑或，结果送** Rd **寄存器。**

格式 7：Rd^= Rs ASR n

说明：将 Rs **移位后的值与** Rd **逻辑异或，结果送** Rd **寄存器。**

格式 8：TEST Rd,Rs ASR n

说明：将 Rs **移位后的值与** Rd **逻辑与，不保存结果，只影响标志位** N **和** Z**。例如：**

R1=0xFF00 //R1 **初值为** 0xFF00

R2=0x0001

R2+=R1 ASR 1 //R1 **算术右移** 1 **位后的值变为** 0xFF80,**再与** R2 **相加，最后** R2 **的值为** 0xFF81

3.5　控制转移类指令

SPCE061A 单片机的控制转移类指令主要有条件转移指令、无条件转移指令、中断指令、中断返回、子程序调用等，如表 3.3、表 3.4 所示。

表 3.3　条件转移指令(指令长度 1，周期 3/5)

助记符	操作数类型	条件	标志位状态
JB	无符号数	小于	C=0
JNB		不小于	C=1
JAE		大于等于	C=1
JNAE		小于	C=0
JA		大于	Z=0 and c=1
JNA		不大于	Not (z =0 and c=1)
JBE		小于等于	Not(z=0 and c=1)
JNBE		大于	Z=0 and c=1
JGE	有符号数	大于等于	S=0
JNGE		小于	S=1
JL		小于	S=1
JNL		大于等于	S=0
JLE	有符号数	小于等于	Not (z=0 and s=0)
JNLE		大于	z=0 and s=0
JG		大于	z=0 and s=0
JNG		小于等于	Not (z=0 and s=0)
JVC		无溢出	N=s
JVS		有溢出	N!=s
JCC	无符号数	进位为 0	C=0
JCS		进位为 1	C=1
JSC		符号位为 0	S=0
JSS		符号位为 1	S=1
JNE		不等于	Z=0
JNZ		非 0	Z=0
JZ		为零	Z=1
JMI		为负	N=1
JPL		为正	N=0
JE		相等	Z=1

表 3.4 无条件转移指令

指令语法	指令长度	指令	指令周期	寻址方式	标志位反应			
					N	Z	S	C
BREAK	单字	BREAK	13	[A16]	–	–	–	–
CALL Label	双字	CALL Label	13	[A22]	–	–	–	–
GOTO Label	双字	GOTO Label	12	[A16]	–	–	–	–
Jmp Label	单字	JMP Label	3/5	PC+IM6	–	–	–	–
RETF	单字	RETF	12	[A22]	√	√	√	√
RETI	单字	RETI	12	[A22]	√	√	√	√

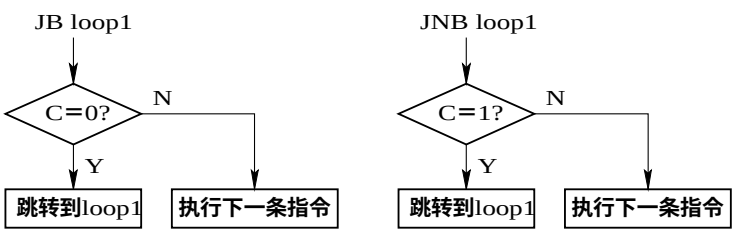
下面介绍各条件转移指令并举例说明。

1. 无符号数的跳转指令

判 C 的转移指令 JB 和 JNB：

```
JB loop1                    //小于，转 loop1
JNB loop1                   //不小于，转 loop1
```

这两条指令都是通过判断标志位 C 决定程序的走向的，它们的执行过程如图 3.3 所示。



(a) JB的执行过程　　(b) JNB的执行过程

图 3.3 指令 JB 和 JNB 的执行过程

例 1 用 JB 指令。

```
        R1=0x0001           //初值
        R2=0x0006
        CMP R1,R2           //比较 R1 和 R2 的大小
        JB loop1            //判 C=0(R1<R2) 吗？是，跳到 loop1；否，执行下一条指令
        R1=0x00FF
        ⋮
loop1:  R1=0x0000
        R2=0x0000
```

例 2 用 JNB 指令。

```
        R1=0x0010           //初值
        R2=0x0006
```

```
        CMP R1,R2          //比较 R1 和 R2 的大小
        JNB loop1          //判 C＝1(R1>R2)吗？是，跳到 loop1
        R1=0x00FF
        ⋮
loop1:  R1=0x0000
        R2=0x0000
```

2. **有符号数的跳转指令**

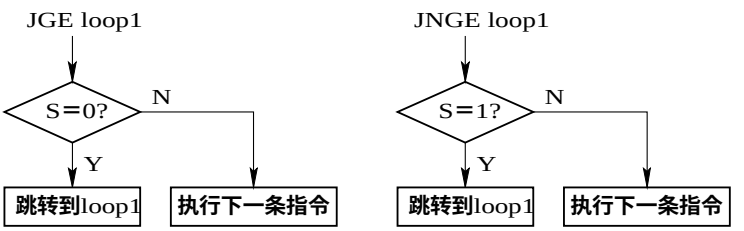
判 S 跳转指令 JGE(**大于等于**)**和** JNGE(**小于**)**：**

JGE loop1

JNGE loop1

JGE **表示大于等于，通过对标志位 S 的判断来决定程序的走向。**S＝0 **，产生跳转；**S＝1**，继续执行下一条指令。**JNGE **表示小于。**S＝1**，产生跳转；**S＝0**，继续执行下一指令。**

这两条指令的执行过程如图 3.4 **所示。**



(a) JEG**的执行过程**　　(b) JNGE**的执行过程**

图 3.4　**指令** JGE **和** JNGE **的执行过程**

例 3　**用** JGE **指令。**

```
        R1=2               //赋初值，分别为有符号数
        R2=−3
        CMP R1,R2          //比较 R1 和 R2 的大小
        JGE loop1          //R1 大于等于 R2 时跳转到 loop1
        R2=0x00FF
        ⋮
loop1:  R1=0x0000
```

例 4　**用** JNGE **指令。**

```
        R1=−2              //赋初值，分别为有符号数
        R2=3
        CMP R1,R2          //比较 R1 和 R2 的大小
        JNGE loop1         //R1 小于 R2(S=1)时跳转到 loop1
        R2=0x00FF
        ⋮
loop1:  R1=0x0000
```

3. **其它跳转指令**

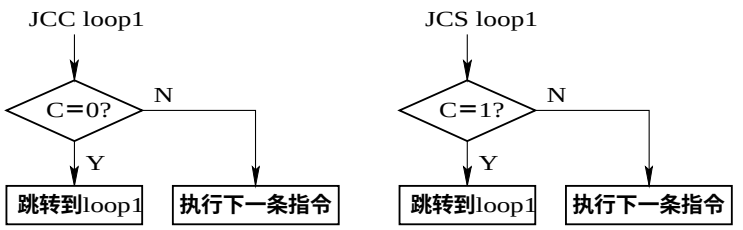
判 C **跳转指令** JCC **和** JCS**：**

JCC loop1

JCS loop1

JCC **是以** C=0 **作为判断的标准，决定程序的跳转；**JCS **是以** C=1 **作为判断的标准，决定程序的跳转。**

这两条指令的执行过程如图 3.5 **所示。**



(a) JCC**的执行过程** (b) JCS**的执行过程**

图 3.5 **指令** JCC **和** JCS **的执行过程**

例 5 **用** JCC **指令。**

```
        R1=0x0020              //赋初值
        R1+=1
        JCC loop1              //如果 C＝0 跳转到 loop1
        R1=0x00FF
        ⋮
loop1:  R1=0x0000
```

例 6 **用** JCS **指令。**

```
        R1=0x0002              //赋初值
        R1-=1
        JCS loop1              //如果 C＝1 跳转到 loop1
        R1=0x00FF
        ⋮
loop1:  R1=0x0000
```

3.6 常用伪指令

伪指令与汇编指令不同，它不会被编译，仅用来控制汇编器的操作。伪指令的作用有点像语言中的标点符号，它能使语言中的句子所表达意思的结构更加清晰，成为语言中不可缺少的一部分。在汇编语言中正确使用伪指令不仅能使程序的可读性增强，而且会提高汇编器的编译效率。本节只介绍常用伪指令，其它指令及其应用请参见附录 D**。**

3.6.1 伪指令的语法格式及特点

伪指令可以写在程序文件中的任意位置，但在其前面必须用一个小圆点引导以区分汇

编指令，伪指令行中方括弧里的参量是任选项，如果某一个参量使用双重方括弧括起来，则说明这个任选项参量本身就必须带着方括弧。例如，[[count]]表示引用该任选参量时必须写出[count]。

SPCE061A 汇编器规定的标准伪指令不必区分字母的大小写，即书写伪指令时既可全用大写也可全用小写甚至可以大小写混用。但所有定义的标号、宏名、结构名、结构变量名、段名及程序名一律区分其字母的大小写。

3.6.2　伪指令符号约定

为了叙述方便，在此对使用的符号作如下统一约定。

bank	存储器的页单元
ROM	程序存储器
RAM	随机数据存储器
label	程序标号
value	常量数值
variable	变量名
number	数据的数目
ASCII	数值或符号的 ASCII 代码
argument#	参量表中的参量序号
filename	文件名
[ ]	任选项

3.6.3　标准伪指令

伪指令依照其用途可分为五类：定义类、存储类、存储定义类、条件类及汇编方式类。它们的具体分类及用途详见表 3.5。

表 3.5　伪指令类别

类　别	用　途	伪　指　令
定义类	程序说明	CODE DATA TEXT
	程序中所用数据的性质范围或结构	IRAM ISRAM ORAM OSRAM RAM SRAM
	宏或结构	MACRO MACEXIT ENDM
	程序	PROC ENDP STRUCT ENDS
	其它	DEFINE VAR PUBLIC EXTERNAL EQU VDEF
存储类	以指定的数据类型存储数据或设定程序地址等	DW、DD、FLOAT、DOUBLE、END
存储定义类	定义若干指定数据类型的数据存储单元	DUP
条件类	对汇编指令进行条件汇编	IF、ELSE、ENDIF、IFMA、IFDEF、 IFNDEF
汇编方式类	包含汇编文件或创建用户定义段	INCLUDE、SECTION

下面介绍两个常用的伪指令 DEFINE 和 PUBLIC。

1. DEFINE

类别：定义类

功能：定义常量符号

语法格式：.DEFINE variable [value][，…]

应用解释: 给常量符号所赋的值既可以是已定义过的常量符号，也可以是表达式，切忌符号超前引用，即如果赋值引用的符号不是在引用前定义的，则会出现非法超前引用的错误。例如：

```
.DEFINE BODY 1
.DEFINE IO_PORT 0x7016
.IFDEF BODY
    R1=0XFFFF
    [IO_PORT]=R1
.ENDIF
```

2. PUBLIC

类别：定义类

功能：声明将被引用在其它文件中的全局标号

语法格式：.PUBLIC label[label][，…]

应用解释：在文件中声明将被引用在外部文件中的全局标号。故在外部文件中用伪指令 EXTERNAL 所声明的标号必须是用 PUBLIC 伪指令声明过的，当要声明多个全局标号时要用逗号(，)将每一标号分开。例如：

```
.PUBLIC  sym1                    //声明要引用在其它文件中的全局标号
.PUBLIC sym1，sym2               //声明多个全局标号需用逗号将每一标号分开，空格
                                 //会被忽略
```

3.6.4 宏定义与调用

1. **宏定义**

所谓宏(Macro)是指在源程序里将一序列的源指令行用一个简单的宏名(Macro Name)所取代。这样做的好处是使程序的可读性增强。

宏在使用之前一定要先经过定义，可分别用伪指令 .MACRO 和 .ENDM 来起始和结束宏定义。定义的宏名将被存入标号域，在汇编器首次编译通过汇编指令时，先将宏定义存储起来，待指令中遇有被调用的宏名时则会用同名宏定义里的序列源指令行取代此宏名。宏定义里可以包括宏参数，这些参数可被代入除注释域之外的任何域内，虚参数不能含有空格。

2. **宏标号**

宏定义里可以用显式标号(由用户定义)，亦可用隐含标号(由汇编器自动定义)。汇编器不会改变用户定义的显式标号。在宏标号后加上后缀符“ # ”则表明该标号为隐含标号，汇编器会自动生成一个后缀数字符号“ _X_XXXX ”(X 表示一位数符，XXXX 表示 4 位扩展

数符)来取代这个隐含标号中的后缀符“#”。隐含标号中的字母字符及其后缀数符总共不能超过 32 个字符。

3. 宏调用

在调用宏时，可以使用任何类型的参数直接型、间接型、字符串型或寄存器型。只有字符串型参数才能含有空格，但必须用引号将此空格括起‘　’或“　”(而字符串参数中的单引号，必须用双引号将其括起)。即只要在宏嵌套中的形参名相同，那么这些参数就可以穿过嵌套的宏使用。宏嵌套使用惟一所受的限制是内存空间的容量。宏的多个参数应以逗号隔开，参数前的空格及 Tab 键都将被忽略。单有一个逗号，而后面未带有任何参数会被汇编器表示为参数丢失错误。

4. 宏参数分隔符

在一个宏体中宏参数的有效分隔符为“，”(逗号)。

5. 宏内字符串连接符

符号“@”(40H)是字符串连接符。注意：字符串连接只能在宏内进行。

6. 助记符搜索顺序

通常汇编器以表 3.6 列出的顺序来搜索各种助记符或符号。

表 3.6　汇编器助记符搜索顺序

序　号	功　能
1	记符表
2	定义表
3	编器伪指令表
4	段名表

3.6.5　段定义与调用

段是应用在汇编器 Xasm16 中的地址标签。Xasm16 除了定义的预定义段以外，还可由用户自己定义段。

1. 预定义段

在 Xasm16 里共定义有 9 个预定义段：CODE、DATA、TEXT、ORAM、OSRAM、RAM、IRAM、SRAM 及 ISRAM 段。这些预定义段都分别被规定了以下内容：

(1) 段内存储数据类型：指令/数据，无初始值的变量/有初始值的变量；

(2) 存储介质类型：ROM/RAM (SRAM)；

(3) 存储范围：零页(或零页中前 64 个字)或当前页/整个 64 页；

(4) 定位排放方式：合并排放/重叠排放。

当用这些预定义段的伪指令来定义自己程序的数据块以后，Xasm16 在汇编时会采取相应的措施进行处理，实际上为链接器的链接处理贴好了地址标签。各预定义段的具体规定可参见相应的预定义段的伪指令内容介绍。

2. 用户定义段

为了使程序在链接时具有更大的灵活性，可以用伪指令.SECTION 来定义段。定义的段名最多不可超过 32 个字符，且最多可定义 4096 个段，但不可嵌套使用。段定义的格式为:

label .SECTION .attribute

其中，属性参数 .attribute 可以是上述 9 个预定义段中的任意一个，表明用户定义段的链接属性与这个预定义段相同，定义一个段之后可以将该段名作为助记符用来进行段的转换。例如：

```
.CODE                          //设置 CODE 预定义段
  NOP
.DATA                          //转换到 DATA 预定义段
.DW       0x20                 //该字节将被存放到 DATA 段
section1： .SECTION  .CODE     //定义一个新的段，其属性与 CODE 预定义段相同
R3 = R3–0x10                   //该指令将被存入 section1 段
.CODE                          //转换到 CODE 段
R1 = 1                         //该指令将被存入 CODE 段
.section1                      //转换到用户定义段 section1
R1 = R1 + 2                    //该指令将被存入 section1 段
.DW       0x30
      //任何用户定义段都可包含代码段或数据段或同时包含二者
.TEXT                          //转换到 TEXT 预定义段
```

3.6.6 结构定义与调用

像在 ANSI–C 那样，汇编器可以把不同类型的数据组织在一个结构体里，为处理复杂的数据结构提供了手段，并为在过程间传递不同类型的参数提供了便利。

1. 结构的定义

结构作为一种数据构造类型在 SPCE061A 汇编语言程序中也要经历定义—说明—使用的过程。在程序中使用结构时，首先要对结构的组成进行定义。定义的一般格式为：

结构名 1：.STRUCT

说明：定义结构开始数据存储类型定义。

结构名 2：.ENDS

说明：定义结构结束。

结构的定义以伪指令.STRUCT 和.ENDS 作为标识符。结构名由用户命名，命名原则与标号等相同。在两个伪指令之间包围的是组成该结构的各成员项数据存储类型的定义。例如：

```
test1：          .STRUCT      //定义了一个结构 test1
ad： .DW         10                 //它包括 3 个成员“ad”，“bs”和“gh”
bs： .DW         “abcd”
```

```
gh：  .DD      0x0FFFC          //且分别被初始化为 10 “ abcd ” 0x0FFFC
.ENDS                           //结构定义结束
```

2. 结构变量的定义——结构的说明

某个结构一经定义，便可指明使用该结构的具体对象，称为结构的说明。其一般格式为：

结构变量名：.结构名[结构成员表]

其中，[结构成员表]用来存放结构变量中成员的值。例如：

```
Stru_var1：        .test1 [20, ‘ ad ’,0x7D]
Stru_var2：        .test1 [10,,0x7D]
```

//第二个成员未被存入新值，因此它的初值可被保留

3. 结构变量的引用

结构是不同数据类型的若干数据变量的集合体。在程序中使用结构时不能把结构作为一个整体来参加数据处理、各种运算和操作，而应是结构中各个成员项数据参加。结构成员项引用的一般格式为：

结构变量名：　. 成员名

例如，R1 + = [stru_var1.ad]，其中，“ stru_var1 ”是一个定义过的结构变量，“ ad ”是它的一个成员。

3.6.7　过程定义与调用

过程是一个子程序块，它有点像 ANSI-C 中的函数，可以把一个复杂且规模较大的程序由整化零变成一个个简单的过程，以使程序结构化。

1. 过程的定义

编写完成某一功能的子程序块，用伪指令.PROC 和.ENDP 作为定义的标识符。定义的一般格式为：

```
过程名：.PROC
            程序指令列表
        RETF
        .ENDP
```

由此格式可以看出，过程的定义主要由过程名和两个伪指令之间的过程体组成。过程名的命名规则同标号。例如：

```
    qw    .PROC                 //定义一个过程 qw
    label1
    R1 + = 0x20
    R2 = R1
    JMP label1
    RETF
.ENDP                           //过程定义结束
```

2. **过程的调用**

在程序中调用一个过程时，程序控制就从调用程序中转移到被调用的过程，且从其起始位置开始执行该过程的指令，在执行完过程体中各条指令并执行到 RETF **指令时，程序控制就返回调用过程时原来断点位置，继续执行下面的指令过程。**

调用的一般格式为：

CALL **过程名**

例如：

```
sub1 .PROC                          //定义一个过程 sub1
     label1
          R1 + = 0x0020；
          R2 = R1；
     JMP label1
RETF
.ENDP                               //过程定义结束
  ⋮
     CALL sub1                      //调用过程 sub1
```

思 考 题

1. SPCE061A **单片机具有哪几种寻址方式？**
2. **试编程序将** 0x0020，0x0021 **单元的数据传送至** 0x1020，0x1021 **单元。**
3. **请编程序将** 0x0100～0x0200 **单元的数据传送至** 0x0400 **起始的单元中，并将原存储单元清零。**
4. **指出下列指令的寻址方式：**

```
R1=0x0005;
R1=06;
R1=[0x0006];
R2=R1;
R2=[R3];
R2=[BP+05];
R1+=R1。
```

5. **请编程计算** 1～10 **的累加值。**

第 4 章　SPCE061A 硬件系统

SPCE061A 单片机内部集成了 ICE 仿真电路、FLASH 程序存储器、SRAM 数据存储器、通用 I/O 端口、定时器/计数器、中断控制、时基系统、A/D 转换器、DAC、异步串行通信口及低电压监测/低电压复位等模块。本章我们将通过对各模块的原理及应用进行讨论，使读者系统地掌握 SPCE061A 单片机的硬件系统的功能和应用。

4.1　并行输入/输出端口

输入/输出端口简称为 I/O 口，是单片机与外设交换信息的主要通道。输入端口从外界接收检测的输入信号、键盘信号等各种开关量信号；输出端口向外部输出处理结果、显示信息、控制命令、驱动信号等。SPCE061A 内部有并行和串行两种方式的 I/O 口。

两个 16 位通用的并行 I/O 端口即 A 口和 B 口，这两个口的每一位都可通过编程单独定义为输入或输出口，A 口的 IOA0～IOA7 用作输入口时具有唤醒功能，即具有输入电平变化中断功能(当输入的电平发生变化，电平由高变低或由低变高时)，唤醒处于睡眠状态的 CPU。对于那些用电池供电追求低能耗的应用场合，可以通过软件设置应用 CPU 的睡眠模式以降低功耗，需要时用按键来唤醒 CPU 使其进入工作状态。例如，手持遥控器、电子字典、个人数字助理 PDA(Personal Digital Assistant)、计算器、移动电话等。

4.1.1　I/O 端口结构

SPCE061A 提供了位控制结构的 I/O 端口，每一位都可以单独定义为输入或输出方式，通常对某一位的设定包括三个基本项：数据向量 Data、属性向量 Attribution 和方向控制向量 Direction，如图 4.1 所示。三个向量的每个对应位组合在一起形成一个控制字，用来定义相应 I/O 口位的输入、输出状态和工作方式，如表 4.1 所示。

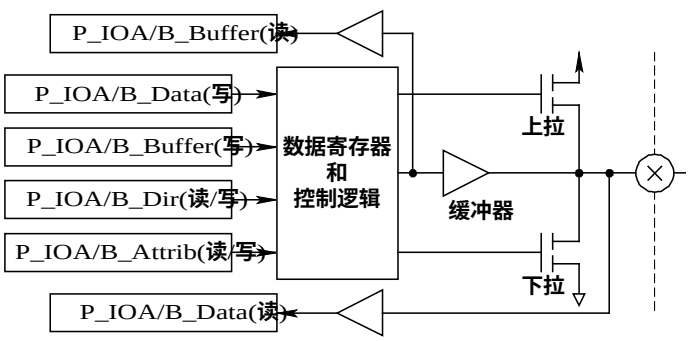


图 4.1　并行 I/O 端口结构

表 4.1 I/O 端口控制器设置

Dir	Attrib	Data	功 能	唤醒功能	功 能 描 述
0	0	0	下拉*	是**	带下拉电阻的输入管脚
0	0	1	上拉	是**	带上拉电阻的输入管脚
0	1	0	悬浮	是**	悬浮式输入管脚
0	1	1	悬浮	否	悬浮式输入管脚
1	0		反相输出	否	Data 数据位反相输出
1	1		同相输出	否	Data 数据位同相输出

注：* 端口位默认为带下拉电阻的输入管脚。

** 只有当 IOA7～IOA0 位的控制字为 000、001 和 010 时，相应位才具有唤醒的功能。

4.1.2 端口设置寄存器

端口设置寄存器有以下几种：

(1) P_IOA_Data(读/写)(7000H) A 口数据单元：用于向 A 口写入或从 A 口读出数据。当 A 口处于输入状态时，读 A 口管脚电平状态，写入数据将写到 A 口的数据寄存器；当 A 口处于输出状态时，写入输出数据到 A 口的数据寄存器。

(2) P_IOA_Buffer (读/写) (7001H) A 口数据向量单元：用于向数据向量寄存器写入或从该寄存器读出数据。当 A 口处于输入状态时，写入是将 A 口的数据向量写入 A 口的数据寄存器，读出是从 A 口数据寄存器内读数据；当 A 口处于输出状态时，写入输出数据到 A 口的数据寄存器。

(3) P_IOA_Dir(读/写) (7002H) A 口方向向量单元：用来设置 A 口是输入还是输出。该方向控制向量寄存器可以写入或从该寄存器内读出方向控制向量，Dir 位决定了端口位的输入/输出方向，即 0 为输入，1 为输出。

(4) P_IOA_Attrib(读/写)(7003H) A 口属性向量单元：用于 A 口属性向量的设置。

(5) P_IOA_Latch(读)(7004H) A 口数据锁存单元：读该单元以锁存 A 口上的输入数据，用于进入睡眠状态前，触键唤醒功能的启动(参见睡眠/唤醒部分)。

(6) P_IOB_Data(读/写)(7005H) B 口数据单元：用于向 B 口写入或从 B 口读出数据。当 B 口处于输入状态时，读的是 B 口管脚电平状态，写入是将数据写入 B 口的数据寄存器；当 B 口处于输出状态时，写入数据到 B 口的数据寄存器。

(7) P_IOB_Buffer(读/写)(7006H) B 口数据向量单元：用于向数据寄存器写入或从该寄存器读出数据。当 B 口处于输入状态时，写入是将数据写入 B 口的数据寄存器，读出则是从 B 口数据寄存器里读数据；当 B 口处于输出状态时，写入数据到 B 口的数据寄存器。

(8) P_IOB_Dir(读/写)(7007H) B 口方向向量单元：用于设置 IOB 口的状态，0 为输入，1 为输出。

(9) P_IOB_Attrib(读/写)(7008H) B 口属性向量单元：用于设置 IOB 口的属性。

例如：假设需要 IOA0 是下拉输入管脚，则相应的 Data、Attribution 和 Direction 对应位的值被置为 000；如果需要 IOA1 是带唤醒功能的悬浮式输入管脚，则 Attribution、Data 和 Direction 的值被置为 010，见表 4.1。

与其它的单片机相比，除了每个 I/O 端口可以单独定义其状态外，每个对应状态下的 I/O 端口电路都是内置的，实际应用中不需要外接电阻(见图 4.1)。例如，设 A 口为带下拉电阻的输入口，在连接硬件时不需在片外接下拉电阻，只需将三个向量的对应位设为 000 即可。

在进行 I/O 口设置时，要特别注意，A 口和 B 口的 Data、Attribution 和 Direction 的设定值均保存在不同的寄存器里(见附录 A)。

4.1.3　工作方式设置

1. 控制组合

方向向量_Dir、属性向量_Attr 和数据向量_Data 分别代表三个控制口，这三个口中每个对应位组合在一起形成一个控制字，定义相应 I/O 口位的工作方式。

? Dir 位决定端口位的输入/输出方向，0 为输入，1 为输出。

? Attr 位决定在端口位的工作方式。在输入状态下是悬浮式输入还是非悬浮式输入，0 为带上拉或下拉电阻式输入，1 为悬浮式输入。在输出状态下，决定其输出是反相输出还是同相输出，0 为反相输出，1 为同相输出。

? Data 位在端口位的输入状态下被写入时，与 Attr 位组合在一起形成输入方式的控制字(Attr Data)00 01 10 11，以决定输入口是带唤醒功能的下拉电阻式、带唤醒功能的上拉电阻式、带唤醒功能的悬浮式、不带唤醒功能的悬浮式输入等。Data 位在端口位的输出状态下写入的是输出数据，不过数据是经过反相器输出还是经过同相缓存器输出，则由 Attr 位来决定。

例如：假设要把 A 口的 b0 位定义成下拉电阻式的输入口，则 A 口_Dir、_Attr 和_Data 向量的三个相应的 b0 位应组合为 000。

如果要将 A 口的 b1 位定义成悬浮式并具有唤醒功能的输入口，则只需将_Dir、_Attr 和_Data 向量中相应的 b1 位组合设置为 010 即可。

例 1　设置 IOA15～IOA12 为带数据缓存器的低电平输出口，IOA11～IOA8 为带数据缓存器的高电平输出口，IOA7～IOA4 为带上拉电阻的输入口，IOA3～IOA0 为带下拉电阻的输入口。

解：设置端口功能只需对相关的寄存器进行相应的设置即可。

	IOA15～IOA12	IOA11～IOA8	IOA7～IOA4	IOA3～IOA0	
Dir:	1 1 1 1	1 1 1 1	0 0 0 0	0 0 0 0	FF00H
Attr:	1 1 1 1	1 1 1 1	0 0 0 0	0 0 0 0	FF00H
Data:	0 0 0 0	1 1 1 1	1 1 1 1	0 0 0 0	0FF0H

编程：

```
R1 = 0xFF00
[P_IOA_Dir]=R1                  //设置 A 口的属性向量
[P_IOA_Attrib]=R1               //设置 A 口的属性向量
R1 = 0x0FF0                     //设置 A 口的数据向量
[P_IOA_Data]=R1
```

由此可见，SPCE061A 端口设置是非常方便灵活的。

2. 睡眠与唤醒

A 口的 IOA0～IOA7 作为唤醒源常用于键盘输入，要激活 IOA0～IOA7 唤醒功能，必须读 P_IOA_Latch 单元，以此来锁存 IOA0～IOA7 管脚上的键状态，随后系统才可通过指令进入低功耗的睡眠状态。当有键按下时，IOA0～IOA7 的输入状态将不同于其在进入睡眠前被锁存的状态，从而唤醒系统。

3. 端口操作

当作为输入口时，所读的数值来自不同地方，读 P_IOA_Data 是 A 口的管脚上的当前状态，读 P_IOA_Buffer 是来自数据寄存器的值。

当 A 口作为输出口时，数据都是写到 A 口的数据寄存器，这种读写方式可以节省许多用于存放端口数据的 RAM 空间。

注意：如果直接读引脚的数据，有时会发生错误。如用某端口驱动三极管的基极，当引脚输出为 1 时，三极管导通，引脚的电平比拉到低电平(0.7 V 左右)，若从引脚直接读取数据，则读为 0，发生了错误。此时，从 Buffer 读寄存器的状态就不会有错误。

例 2　在单步运行程序期间将 A 口的任意管脚接 VDD，通过观察寄存器表中的 R2、R3 的值可观察到 P_IOA_Data、P_IOA_Buffer 二者的不同。

程序：

```
        .include hadware.inc                    //包含头文件
        .code
        .public _main                           //主程序
_main:
            R1=0x0000                           //设置 A 口为带下拉电阻的输入口
            [P_IOA_Data]=r1                     //设置 A 口的数据向量
            [P_IOA_Attrib]=r1                   //设置 A 口的属性向量
            [P_IOA_Dir]=r1                      //设置 A 口的方向向量
          //此时将 A 口的任一端口接高电平//
            R3=[P_IOA_Data]                     //把 P_IOA_Data 中的值送到寄存器 R3
            R2=[P_IOA_Buffer]                   //把 P_IOA_Buffer 中的值送到寄存器 R2
WAIT:       NOP
            JMP WAIT                            //主程序循环
```

例 3　读端口与读 Buffer 的区别。

```
 _main:
            R1=0xFFFF                           //设置 B 口为带反向器的低电平输出
            [P_IOB_Dir]=R1
            [P_IOB_Data]=R1
            R1=0x0000
            [P_IOB_ Attrib]=R1
            R1=0x0000                           //向寄存器写入 0x0000
```

```
        [P_IOB_Data]=R1
    //此时检测 B 口的电平为高//
        R3=[P_IOB_Buffer]          //读回 Buffer 的值为 0x0000
LOOP:NOP
        JMP LOOP                   //主程序循环
```

例 4　设置 A 口低 8 位为带上拉电阻的输入口用来输入按键信号，B 口低 8 位为带数据缓存器的高电平输出口，外接发光二极管。当有键按下时，对应的灯点亮，key1 按下，B0 口 LED 点亮……依此类推。硬件电路如图 4.2 所示。

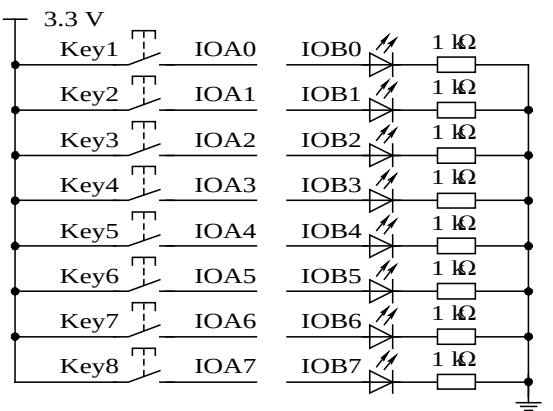


图 4.2　例 4 硬件连接图

程序如下：

```
    R1=0x0000
    [P_IOA_Dir]=R1              //设置 A 口的方向向量
    [P_IOA_Attr]=R1             //设置 A 口的属性向量
    [P_IOA_Data]=R1             //设置 A 口为下拉输入
    R2=0x00FF                   //R2 的值为 0x00FF
    [P_IOB_Dir]=R2              //设置 B 口的方向向量
    [P_IOB_Attr]=R2             //设置 B 口的属性向量
    [P_IOB_Data]=R2             //设置 B 口低 8 位为同相输出
//设已经取到键值在[keycode]单元，判断键值并输出显示的程序//
        R1=[keycode]            //将所取的键值送到 R1 中
        CMP R1,0x0000           //若所取的键值为 0 表明无键按下，返回主程序
        JE _MAIN
        CMP R1 ,0x0001          //比较是否为第一键按下
        JE LOOP1                //是，跳转到 LOOP1
        CMP R1 ,0x0002          //比较是否为第二键按下
        JE LOOP2                //是，跳转到 LOOP2
            ⋮
        CMP R1 ,0x0040          //比较是否为第七键按下
        JE LOOP7                //是，跳转到 LOOP7
```

```
                R2=[P_IOA_Data]          //否则是，第八个键按下 IOB7 口输出高电平
                R2&=0x0080
                [P_IOB_Data]=R2          //IOB7 口输出高电平点亮该口的指示灯
                JMP _MAIN                //返回主程序
                    ⋮
LOOP1：  R2=[P_IOA_Data];                //是第一键按下，IOB0 口输出高电平
                R2&=0x0001               //确保只有 IOB0 口输出高电平
                [P_IOB_Data]=R2
                JMP _MAIN
LOOP2：  R2=[P_IOA_Data]                 //是第二键按下 IOB1 口输出高电平
                R2&=0x0002               //确保只有 IOB1  口输出高电平
                [P_IOB_Data]=R2
                JMP _MAIN
                    ⋮
LOOP7：  R2=[P_IOA_Data]                 //是第七键按下，IOB6 口输出高电平
                R2&=0x0040               //确保只有 IOB6 口输出高电平
                [P_IOB_Data]=R2
                JMP _MAIN
```

4.1.4 B 口的特殊功能

1. **特殊功能**

正如前面提到的那样，B 口除具有常规的输入/输出功能外，还具有特殊功能，如表 4.2 所示。

表 4.2 B 口的特殊功能

口位	特殊功能	功能描述	备注
IOB0	SCK	串行接口 SIO 时钟信号	参见 SIO 功能设置
IOB1	SDA	串行接口 SIO 数据信号	参见 SIO 功能设置
IOB2	EXT1	外部中断源 1 信号输入	IOB2 设为输入状态
	Feedback_Output1	与 IOB4 组成一个 RC 反馈电路，以获得振荡信号，作为外部中断源 EXT1	设置 IOB2 为反相输出方式
IOB3	EXT2	外部中断源 2 信号输入	IOB3 设为输入状态
	Feedback_Output2	与 IOB5 组成一个 RC 反馈电路，以获得一个振荡信号作为外部中断源 EXT2	设置 IOB3 为反相输出方式
IOB4	Feedback_Input1		与 IOB2 配合使用
IOB5	Feedback_Input2		与 IOB3 配合使用
IOB7	Rx	串行数据接收端口	参见 4.7 节
IOB8[1]	APWMO	TimerA 脉宽调制输出	参见 4.3 节
IOB9	BPWMO	TimerB 脉宽调制输出	参见 4.3 节
IOB10[2]	Tx	串行数据发送端口	参见 4.7 节

注：[1] **脉宽调制** PWM (Pulse Width Modulation)。

[2] **口位默认为带下拉电阻的输入管脚。**

2. 寄存器设置

P_FeedBack(写)(7009H)B 口工作方式控制单元，用于控制 B 口的 IOB2(IOB3)和 IOB4(IOB5)是用作普通 I/O 口，还是作为特殊功能口。其特殊功能包括以下两个部分：

(1) 单个 IOB2 或 IOB3 口可设置为外部中断输入口。

(2) 设置 P_FeedBack 单元，再将 IOB2(IOB3)和 IOB4(IOB5)之间接一个电阻和电容，电路连接如图 4.3 所示。形成反馈电路以产生振荡信号，此信号可作为外部中断源 EXT1 或 EXT2 输入，当然此时所得到的中断频率为 RC 振荡器的频率。由于该频率较高，所以通常情况下都是通过(1)获得外部中断信号。此特殊功能仅运用于当外部电路需要用到一定频率的振荡信号时，可以在 IOB2(IOB3)端获得。

注意：为了使反馈回路正常工作，必须将 IOB2 (IOB3) 设置成反相输出口，将 IOB4 (IOB5)设置成悬浮式输入口。

当 P_FeedBack(写)(7009H)单元的 FBKEN2(FBKEN3)位被置为“1”时，IOB2、IOB4 或 IOB3、IOB5 之间的反馈结构如表 4.3 所示。

表 4.3　P_FeedBack(7009H)设置

b3	b2
FBKEN3	FBKEN2
1：IOB3 和 IOB5 之间接反馈元件 0：IOB3、IOB5 作为普通的 I/O 口(默认)	1：IOB2 和 IOB4 之间接反馈元件 0：IOB2、IOB4 作为普通的 I/O 口(默认)

IOB8 的控制向量 TAON 的设置如下：

TAON 为 0：IOB8 是普通 I/O 端口。

TAON 为 1：IOB8 是 APWMO 端口。

注意：

(1) TAON TimerA 脉宽调制输出 APWMO 的允许信号，详见 4.3 节。

(2) APWMO TimerA 脉宽调制的输出信号，详见 4.3 节。

例 5　以下程序说明如何编程使得通过在 IOB2 和 IOB4 之间连接一个 RC 电路形成反馈回路，以获得振荡源频率。

```
//将 IOB4 设置成悬浮式输入口，IOB2 设置成反相输出口
R1=0x0004                          //R1=0000 0000 0000 0100B
[P_IOB_Dir]=R1
R1=0x0010
[P_IOB_Attrib]=R1
[P_IOB_Data]=R1
R1=0x0004                          //设定 IOB2、IOB4 为特殊功能
[P_FeedBack]=R1
```

4.2 系统时钟与复位电路

4.2.1 时钟电路

SPCE061A 单片机的时钟电路采用外接时钟源方式，推荐使用晶体振荡器，频率采用 32 768 Hz。电路如图 4.3 所示。

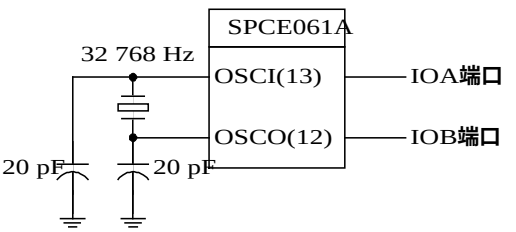


图 4.3 SPCE061A 时钟电路

SPCE061A 单片机利用 32 768 Hz 实时时钟源产生了多种实时时钟源，可作为精确的计时基准信号。

4.2.2 锁相环 PLL(Phase Lock Loop)振荡器

PLL 电路的作用是将系统提供的实时时钟基频(32 768 Hz)进行倍频调整至 49.152 MHz、40.960 MHz、32.768 MHz、24.576 MHz 或 20.480 MHz。系统默认的 PLL 自激振荡频率为 24.576 MHz，如图 4.4 所示。

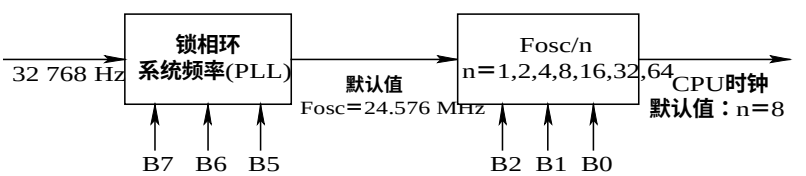


图 4.4 系统时钟框图

系统时钟选频 P_SystemClock(写)(7013H)的第 7、6、5 位和第 2、1、0 位

4.2.3 系统时钟

1. 工作方式

32 768 Hz RTC(Real Time Clock)振荡器有两种工作方式：强振模式和自动弱振模式。强振模式时，RTC 振荡器始终运行在高耗能的状态下；自动弱振模式时，系统在上电复位后的前 7.5 s 内处于强振模式，然后自动切换到弱振模式以降低功耗。CPU 被唤醒后默认的时钟频率为 Fosc/8，可根据需要调整该值。CPU 被唤醒后，经过 32 个时钟周期的缓冲时间后进行其它的操作，这样可以避免在系统被唤醒后造成 ROM 读取错误。

2. **时钟设置**

32 768 Hz **的实时时钟信号经过** PLL **倍频电路产生的系统时钟频率**(Fosc)，Fosc **再经过分频得到** CPU **时钟**(CPUCLK)。**默认** Fosc **和** CPUCLK **分别为** 24.576 MHz **和** Fosc/8，CPU **时钟频率为** 3.072 MHz。**通过对** P_SystemClock(**写**)(7013H)**单元编程，可完成对系统时钟和** CPU **时钟频率的编程。**

P_SystemClock **单元中** b7～b5 **为频率选择，如表** 4.4 **所示**; b4 **为睡眠状态选择；**b3 **为方式选择；**b2～b0 **为** CPU **时钟选择，如表** 4.5 **所示。**

b4　1：**在备用状态下** 32 768 Hz **时钟仍处于工作状态(默认)。**

　　0：**在备用状态下** 32 768 Hz **时钟被关闭。**

b3　1：32 768 Hz **时钟处于强振模式。**

　　0：32 768 Hz **时钟处于自动弱振模式(默认)。**

表 4.4　**系统时钟**(PLL) **频率选择**

b7	b6	b5	**频率**/MHz
0	0	0	24.5760
0	0	1	20.480
0	1	0	32.7680
0	1	1	40.960
1	—	—	49.1520

表 4.5　CPU **时钟选择**

b2	b1	b0	CPU **时钟频率**
0	0	0	Fosc
0	0	1	Fosc/2
0	1	0	Fosc/4
0	1	1	Fosc/8
1	0	0	Fosc/16
1	0	1	Fosc/32
1	1	0	Fosc/64
1	1	1	**停止(睡眠状态)**

注意:

(1) **只有当** b2～b0 **同时被置为** 1 **时(即睡眠状态)，**b4 **设置才有效。**

(2) **上电复位或系统从备用状态(即睡眠状态)被唤醒后，默认的** CPU **时钟频率为** Fosc/8。

4.2.4　时间基准信号

时间基准信号简称时基信号，来自于 32 768 Hz **实时时钟，通过频率选择组合而成。时基信号发生器的选频逻辑** TMB1 **为** TimerA **的时钟源** B **提供各种频率选择信号，并为中**

断系统提供中断源(IRQ6)信号。此外，TMB2 只为 IRQ6 提供时钟信号(128 Hz、256 Hz、512 Hz、1024 Hz)。时基信号发生器还可以通过分频产生 2 Hz、4 Hz、1024 Hz、2048 Hz 以及 4096 Hz 的时基信号，为中断系统提供各种实时中断源(IRQ4 和 IRQ5)信号。时基信号发生器的结构如图 4.5 所示。

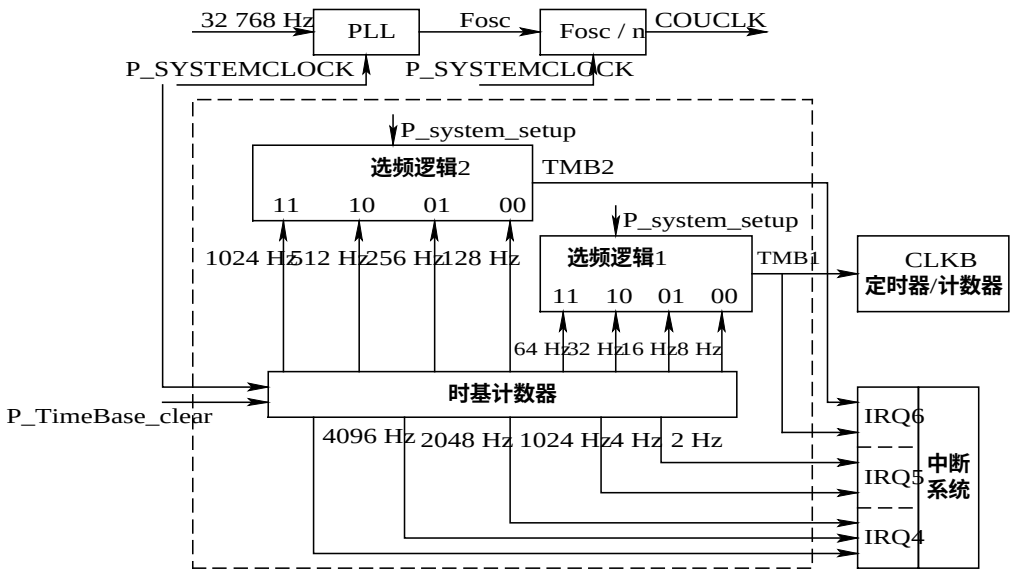


图 4.5 时基信号发生器的结构

1. 时基信号设置单元 P_Timebase_Setup(写)(700EH)

时基信号发生器通过对 P_Timebase_Setup(写)(700EH)单元编程来实现选频操作，b3、b2: TMB2 选频逻辑；b1、b0: TMB1 选频逻辑，见表 4.6。

表 4.6 选 频 逻 辑

b3	b2	TMB2	b1	b0	TMB1
0	0	128 Hz*	0	0	8 Hz**
0	1	256 Hz	0	1	16 Hz
1	0	512 Hz	1	0	32 Hz
1	1	1024 Hz	1	1	64 Hz
* 默认的 TMB2 输出频率为 128 Hz			** 默认的 TMB1 输出频率为 8 Hz		

2. 时基信号清零单元(校准) P_Timebase_Clear(写)(700FH)

P_Timebase_Clear (写)(700FH)单元是控制端口，设置该单元可以完成时基计数器复位和时间校准。向该单元写入任意数值，时基计数器将被清 0，从而对时基信号发生器进行精确的时间校准。

4.2.5 复位电路

通过某种方式使单片机内存及各寄存器的值变为初始值的操作称为复位，SPCE061A

复位电路如图 4.6 所示。在 SPCE061A 单片机 RES_B(第 6 脚)端加上一个低电平就可令其复位。该电路具有手动和上电复位两种功能。

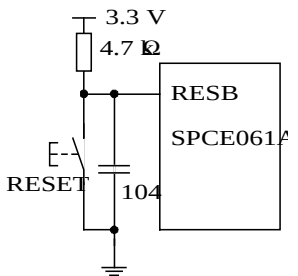


图 4.6　复位电路

4.3　定时器/计数器

4.3.1　工作原理

SPCE061A 提供了两个具有自动重置初值的 16 位定时器/计数器 TimerA 和 TimerB。TimerA 为通用计数器，TimerB 为多功能计数器。TimerA 的时钟源由时钟源 A 和时钟源 B 进行与操作形成，TimerB 的时钟源仅为时钟源 A，它们的内部结构如图 4.7 和图 4.8 所示。

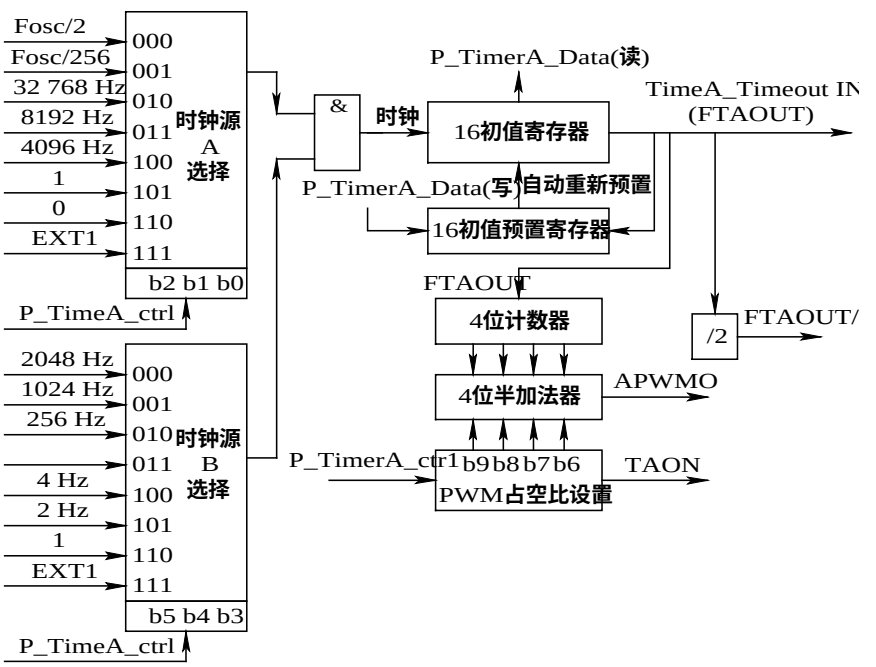


图 4.7　TimerA 结构图

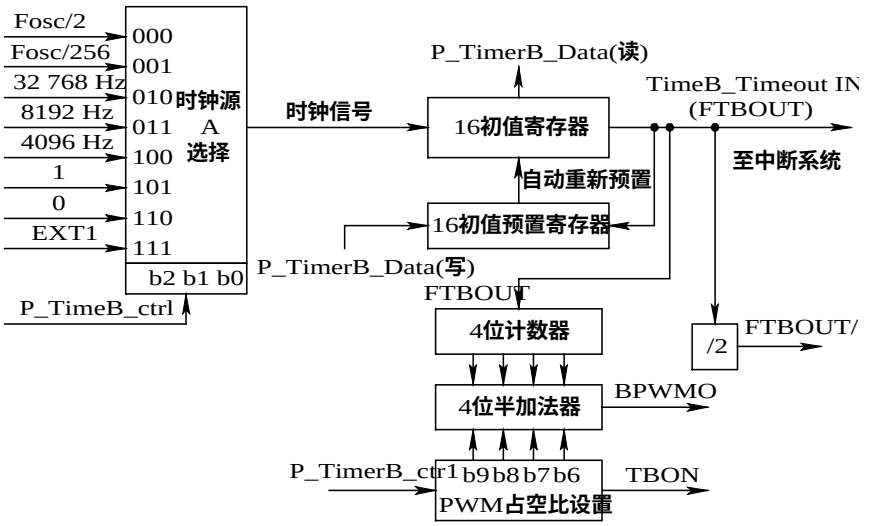


图 4.8 TimerB 结构图

定时器发生溢出会产生一个溢出信号(TAOUT/TBOUT)，作为定时器中断请求信号传输到 CPU 中断系统；作为 4 位计数器时钟源信号，输出一个具有 4 位占空比可调的脉宽调制输出信号 APWMO 或 BPWMO(分别从 IOB8 和 IOB9 输出)控制电机或其它设备的速度。此外，定时器溢出信号还可以用于触发 ADC 输入的自动转换控制和 DAC 输出的数据锁存控制信号。

选择一个合适的时钟源，向定时器的 P_TimerA_Data(读/写)(700AH)单元或 P_TimerB_Data(读/写)(700CH)单元写入一个计数初值 N，定时器/计数器将在此时钟频率下，开始以递增方式计数。N，N+1，N+2，…，0xFFFE，0xFFFF，当计数达到 0xFFFF 后，定时器/计数器溢出，产生中断请求信号，向 CPU 请求中断，CPU 响应后，送入中断控制器处理，执行中断服务程序，完成相应的中断任务。同时，将初值 N 重新载入定时器/计数器初值寄存器，并重新开始计数。

时钟源 A 是高频时钟源，来自带锁相环的晶体振荡器输出 Fosc；时钟源 B 是低频时钟源，来自 32 768 Hz 实时时钟系统，时钟源 B 可以作为精确的计时器。例如，2 Hz 定时器可以作为实时时钟的时钟源。

在 TimerA 内，时钟源 A 和时钟源 B 的组合为 TimerA 提供了多种组合的计数频率。例如，若以 CLKA 作为门控信号，1 为允许时钟源 B 信号通过，0 为禁止时钟源 B 信号通过，以此来控制 TimerA 的计数或停止。如果时钟源 A 为 1，TimerA 时钟频率将取决于时钟源 B；如果时钟源 A 为 0，将停止 TimerA 的计数。EXT1 和 EXT2 为外部时钟源。

TimerB 只有一个时钟源 A。

4.3.2 工作寄存器设置

1．TimerA 计数单元 P_TimerA_Data

TimerA 计数单元 P_TimerA_Data(读/写)(700AH)用于向 16 位预置寄存器写入计数初值或读取数据。

2．TimerA 控制单元 P_TimerA_Ctrl

通过设置 TimerA 控制单元 P_TimerA_Ctrl(写)(700BH)的 b0～b5 来选择 TimerA 的时钟源(时钟源 A、B)，设置该单元的 b6～b9，TimerA 将输出不同频率的脉宽调制信号，即控制脉宽占空比输出 APWMO，如表 4.9 所示，每位的具体意义如表 4.10～表 4.12 所示。

表 4.9　P_TimerA_Ctrl 单元设置

b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
Output_pulse_ctrl				时钟源 B 选择位			时钟源 A 选择位		

表 4.10　b5、b4、b3 位的意义

b5	b4	b3	时钟源 B 的频率
0	0	0	2048 Hz
0	0	1	1024 Hz
0	1	0	256 Hz
0	1	1	TMB1
1	0	0	4 Hz
1	0	1	2 Hz
1	1	0	1
1	1	1	EXT2

注：时钟源 B 的默认值为 1。

表 4.11　b2、b1、b0 的意义

b2	b1	b0	时钟源 A 的频率
0	0	0	Fosc/2
0	0	1	Fosc/256
0	1	0	32 768 Hz
0	1	1	8192 Hz
1	0	0	4096 Hz
1	0	1	1
1	1	0	0
1	1	1	EXT1

注：时钟源 A 的默认值为 0。

表 4.12 b9、b8、b7、b6 位的意义

b9	b8	b7	b6	脉宽占空比(APWMO)	TAON[1]
0	0	0	0	关断	0
0	0	0	1	1/16	1
0	0	1	0	2/16	1
0	0	1	1	3/16	1
0	1	0	0	4/16	1
0	1	0	1	5/16	1
0	1	1	0	6/16	1
0	1	1	1	7/16	1
1	0	0	0	8/16	1
1	0	0	1	9/16	1
1	0	1	0	10/16	1
1	0	1	1	11/16	1
1	1	0	0	12/16	1
1	1	0	1	13/16	1
1	1	1	0	14/16	1
1	1	1	1	TAOUT[2]触发信号	1

注：[1] TAON 是 TimerA(APWMO)的脉宽调制信号输出允许位，默认值为 0，当 TimerA 的 b6～b9 不全为零时，TAON=1。

[2] TAOUT 是 TimerA 的溢出信号，当 TimerA 的计数从 N 达到 0xFFFF 后，产生的溢出信号可以作为 Timer 的中断标志信号送至中断控制系统，同时 N 值将被重新载入预置寄存器，使 Timer 重新开始计数。TAOUT 触发信号(TAOUT/2)的占空比为 50%，频率为 FTAOUT/2，其它输入信号的频率为 FTAOUT/16，请参考图 4.7 所示的 TimerA 结构图。

不同占空比的 PWM 输出波形如图 4.9 所示。

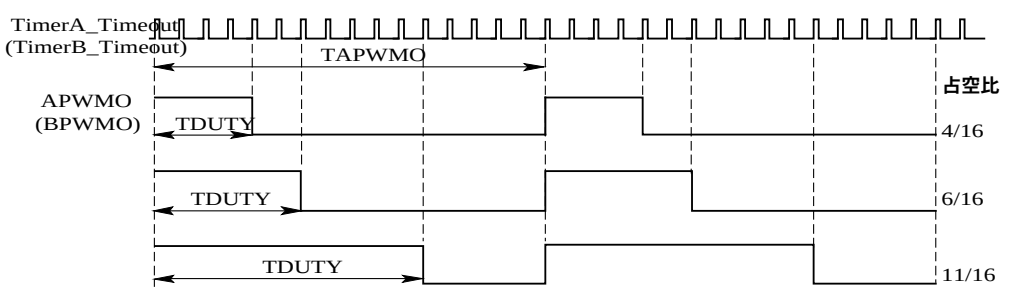


图 4.9 不同占空比的 PWM 输出波形

3. TimerB 数据单元 P_TimerB_Data

TimerB 数据单元 P_TimerB_Data(读/写)(700CH)用于向 16 位预置寄存器写入计数初值或从其中读取数据。

4. TimerB 控制单元 P_TimerB_Ctrl

通过设置 TimerB 控制单元 P_TimerB_Ctrl(写)(700DH)的 b0～b2 来选择 TimerB 的时钟源，设置 b6～b9 选择 TimerB 输出不同脉宽调制信号即控制脉宽占空比输出 PWMO，如表 4.13 所示，每位的具体意义如表 4.14 和表 4.15 所示。

表 4.13　P_TimerB_Ctrl 单元设置

b9	b8	b7	b6	b2	b1	b0
Output_pulse_ctrl				时钟源 A 选择位		

表 4.14　b9、b8、b7、b6 位的意义

b9	b8	b7	b6	脉宽占空比(BPWMO)	TBON[1]
0	0	0	0	关断	0
0	0	0	1	1/16	1
0	0	1	0	2/16	1
0	0	1	1	3/16	1
0	1	0	0	4/16	1
0	1	0	1	5/16	1
0	1	1	0	6/16	1
0	1	1	1	7/16	1
1	0	0	0	8/16	1
1	0	0	1	9/16	1
1	0	1	0	10/16	1
1	0	1	1	11/16	1
1	1	0	0	12/16	1
1	1	0	1	13/16	1
1	1	1	0	14/16	1
1	1	1	1	TBOUT[2]触发信号	1

注：[1] TBON 是 TimerB(BPWMO)的脉宽调制信号输出允许位，默认值为 0。

[2] TBOUT 是 TimerB 的溢出信号，当 TimerB 的计数从 N 达到 0xFFFF 后，产生的溢出信号作为 Timer 中断标志信号送至中断控制系统，同时 N 值将被重新装入预置寄存器，使 TimerB 重新开始计数。TBOUT 触发信号(TBOUT/2)的占空比为 50%，频率为 FTBOUT/2，其它输入信号的频率为 FTBOUT/16。

表 4.15　b2、b1、b0 位的意义

b2	b1	b0	时钟源 A 的频率
0	0	0	Fosc/2
0	0	1	Fosc/256
0	1	0	32 768 Hz
0	1	1	8192 Hz
1	0	0	4096 Hz
1	0	1	1
1	1	0	0
1	1	1	EXT1

注：时钟源 A 的默认值为 0。

4.3.3 定时器/计数器应用

系统设计时常常要用到定时器/计数器，编程之前首先要根据项目的要求，确定控制寄存器、初值寄存器的值，然后再编程实现，一般要经过以下三步。

1. 选择时钟源

由 P_TimerA_Ctrl、P_TimerB_Ctrl 控制寄存器的 b5～b3 和 b2～b0 设定时钟源 A、B，以确定 TimerA 的时钟频率。即根据项目要求选择合适的频率，确定 P_TimerA_Ctrl、P_TimerB_Ctrl 控制寄存器 b5～b3 和 b2～b0 的值。

2. 确定脉冲调制信号占空比

由 P_TimerA_Ctrl、P_TimerB_Ctrl 控制寄存器的 b9～b6 设定脉冲调制信号占空比。即根据项目要求选择输出脉宽调制信号占空比，确定 P_TimerA_Ctrl、P_TimerB_Ctrl 控制寄存器 b9～b6 的值。

3. 设定初值

确定定时器/计数器初值寄存器的预制初值 N。

设所需定时时间为 t，输入时钟信号周期为 T，频率为 f，初值为 N。则

$$t=(FFFFH-N)\times T$$

可得：

$$N=FFFFH-t/T=FFFFH-t\times f$$

例如：利用 TimeA 设置 2 s 定时，选择时钟频率为 1024 Hz，PWM 占空比为 4/16。

解：确定控制寄存器 P_TimerA_Ctrl 和初值寄存器 P_TimerA_Data 的值。

(1) 选择时钟源。设定时钟源 A 为 1，b2 b1 b0=101；设定时钟源 B 为 1024 Hz，b5 b4 b3=001。

(2) 确定脉冲调制信号占空比。PWM 占空比为 4/16，设定 b9 b8 b7 b6=0100，所以，控制寄存器 P_TimerA_Ctrl 的值为 000000 0100 001 101B＝010DH。

(3) 设定初值。所需定时时间为 t=2 s，输入时钟信号频率为 f=1024 Hz，初值为 N，则

$$N=FFFFH-t / T=FFFFH - t\times f=FFFFH - 2\times 1024=FFFFH - 0800H=F7FFH$$

所以，初值寄存器 P_TimerA_Data 的值为 F7FFH。

(4) 编程：

```
R1=F7FFH
[P_TimerA_Data]=R1
R1=010DH
[P_TimerA_Ctrl]=R1
```

可见 SPCE061A 定时器/计数器的设置比较简单方便。

4.4 A/D(模/数)转换器

4.4.1 A/D 转换

SPCE061A 有 8 个通道的 10 位 A/D 转换器，其中 7 个为普通 A/D 转换通道，模拟信

号直接通过 IOA0～IOA6 输入，另一个是语音信号输入通道，语音信号通过内置具有自动增益控制放大器的麦克风通道(MIC_IN)输入。

实际上可以把 A/D 转换器(ADC，Analog to Digital Converter)看作是一个实现 A/D 信号转换的编码器。在 ADC 内由 A/D 转换器 DAC0 和逐次逼近寄存器 SAR 组成逐次逼近式 A/D 转换器，如图 4.10 所示。

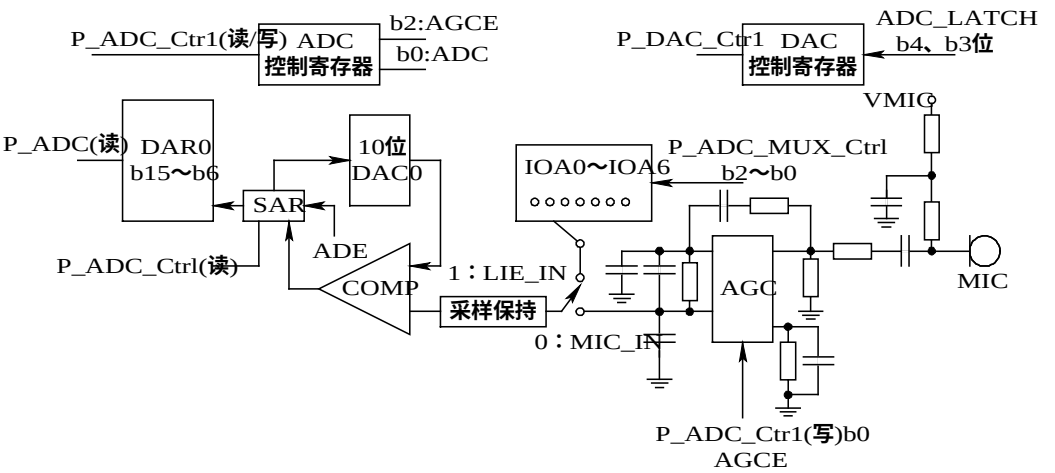


图 4.10　ADC 输入接口原理图

向 P_ADC_Ctrl(写)(7015H)单元 b0(ADE)写入 1，用以激活 ADC，系统默认的设置为屏蔽 ADC(ADE=0)。ADC 采用自动方式工作，硬件限定 ADC 的最高响应频率为(Fosc/32)Hz，如果速率超过此值，会发生转换错误。

下表列出了 ADC 在各种系统时钟频率下的响应速率。

FOSC/MHz	20.480	24.576	32.768	40.960	49.152
ADC 响应频率/kHz	640	768	1024	1280	1536

即 ADC 的最高响应频率为 1536 kHz。

ADC 采用逐次逼近式，在 ADC 自动方式启用后，会产生一个启动信号 RDY=0，此时 DAC0 的电压模拟量输出值与外部的电压模拟量输入值进行比较，以尽快找出与外部电压模拟量相适应的数字量输出值。逐次逼近式控制首先将 SAR 中数据的最高有效位试设为 1，而其它位则全设为 0，即 10 0000 0000B。这时 DAC0 输出电压 VDAC0(1/2 满量程)与输入电压 Vin 进行比较，如果 Vin>VDAC0，则保持原先设置为 1 的位(最高有效位)，否则该位会被清 0。接着逐次逼近式控制又将下一位试设为 1，其余低位依旧设为 0，即 110000 0000B，VDAC0 与 Vin 进行比较，若 Vin>VDAC0，则仍保持原先设置位，否则该位清 0。这一过程一直延续到 10 位中的所有位都被测试之后，A/D 转换的结果才保存在 SAR 内。当 10 位 A/D 转换完成时，RDY 置 1，此时，通过读取 P_ADC(7014H)或 P_ADC_MUX_Data(702BH)单元，可以获得 10 位 A/D 转换的数据。而从该单元读取数据后，又会使 RDY 自动清 0 重新启动 A/D 转换，若未读取 P_ADC(7014H)或 P_ADC_MUX_Data(702BH)单元中的数据，RDY 仍保持为 1，不会启动下一次的 A/D 转换。外部模拟信号由 LIN_IN[1～7]即

IOA[0～6]直接输入缓冲器 P_ADC_MUX_Data(702BH)。从 MIC_IN 输入的模拟信号则要经过缓冲器和放大器(具有 AGC 功能)将信号幅值控制在一定范围内，然后经采样—保持模块送至比较器，进行 A/D 转换后送入 P_ADC (7014H)。

4.4.2 A/D 转换器设置

1. 转换结果寄存器 P_ADC

转换结果寄存器 P_ADC(读/写)(7014H)储存 MIC 输入模拟量转换的数据。逐次逼近式的 ADC 由一个 10 位 DAC(DAC0)、一个 10 位缓存器 DAR0、一个逐次逼近寄存器 SAR 和一个比较器 COMP 组成。

P_ADC(读)：读出本单元，实际为 A/D 转换输出的 10 位数字量，使用了高 10 位(b15～b6，而且如果 P_DAC_Ctrl(702AH)单元 b3、b4 位被设为 00，那么，在转换过程中读出本单元(7014H)，亦会使 A/D 转换重新开始。

2. ADC 控制寄存器 P_ADC_Ctrl

ADC 控制寄存器 P_ADC_Ctrl(读/写)(7015H)各位的功能如表 4.16 所示。

表 4.16 ADC 控制寄存器各位的功能

b15	b6	b2	b0	控制功能描述
RDY(读)	DAC_I(写)	AGCE(写)	ADE(写)	
0	—	—	—	A/D 转换未完成
1	—	—	—	A/D 转换完成输出 10 位数字量
	0			DAC 电流=3 mA @VDD=3 V
	1			DAC 电流=2 mA @VDD=3 V
—	—	0	—	取消自动增益控制功能
—	—	1	—	设置自动增益控制功能
—	—	—	0	禁止 A/D 转换工作
—	—	—	1	允许 A/D 转换工作

注意：

(1) DAC 的电流和 VDD 的值为 DAC_I 的缺省选择。

(2) b15 只用于 MIC_IN 通道输入。

(3) 当模拟信号通过麦克风的 MIC_IN 通道输入时，选择 AGCE 为 1 ，运算放大器的增益可在其线性区域内自动调整。AGCE 缺省选择为 0 即取消自动增益控制功能。

(4) 写入时需注意 b5 = 1, b4=1, b3 =1 且 b1=0。

3. 通道选择寄存器 P_ADC_MUX_Ctrl

ADC 多通道控制是通过对 P_ADC_MUX_Ctrl (读/写) (702BH)单元编程实现的，具体功能如表 4.17 所示。

表 4.17　通道选择寄存器各位的功能

b15	b14	b2	b1	b0	控制功能描述
Ready_MUX(读)[1]	FAIL(读)[2]	Channel_sel(读/写)			
0		—	—	—	模/数转换未完成
1	0	—	—	—	模/数转换已完成
—		0	0	0	模拟电压信号通过 MIC_IN 输入
—		0	0	1	模拟电压通过 LINE_IN1 输入
—		0	1	0	模拟电压通过 LINE_IN2 输入
—		0	1	1	模拟电压通过 LINE_IN3 输入
—		1	0	0	模拟电压通过 LINE_IN4 输入
—		1	0	1	模拟电压通过 LINE_IN5 输入
—		1	1	0	模拟电压通过 LINE_IN6 输入
—		1	1	1	模拟电压通过 LINE_IN7 输入

注：[1] Ready_MUX 只用于 Line_in。

[2] 一般情况下该位总为 0。以下情况除外，由于 MIC_IN 的优先级高于 AD LINE_IN，所以，在 LINE_IN AD 转换过程中又有 MIC_IN 时，若 AD 切换到 MIC 输入，原 LINE_IN 的数据会出现问题，此时 fail 被置 1 表示转换失败。MIC AD 完成之后，该位被清为 0。

ADC 的多路 LINE_IN 输入端与 IOA[6～0]共用：

IOA6	IOA5	IOA4	IOA3	IOA2	IOA1	IOA0
LIN_IN7	LIN_IN6	LIN_IN5	LIN_IN4	LIN_IN3	LIN_IN2	LIN_IN1

4. P_ADC_MUX_Data

P_ADC_MUX_Data(读)(702BH)单元用于读出 LINE_IN[7～1]10 位 ADC 转换的数据。

b15	b14	b13	B12	b11	b10	b9	b8	b7	b6
D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

4.4.3　A/D 转换器的物理性能

1．ADC 直流电气特性

ADC 直流电气特性如表 4.18 所示。

表 4.18　ADC 直流电气特性表

直流电气参数	符号	最小值	典型值	最大值	单位
分辨率	RESO		10		bit
有效位数	ENOB	8			bit
信噪比	SNR	50			dB
积分非线性	INL			±4	LSB
差分非线性	DNL			±2	LSB
转换率	FCONV			96 K	Hz
电源电流@Vdd=3 V	IADC		3.4		mA
功耗@Vdd=3 V	PADC		10.2		mW

注意：

(1) LSB **表示为最小有效单位，在** VRT=3 V **的情况下** 1LSB **为** 3000/1024＝2.93 mV。

(2) **此表采用最大采样率**(Samplerate_max)，**即** Samplerate_max=ADC **响应率**/16=1536 kHz/16=96 kHz。

2. ADC **的交流电气特性**

ADC **交流电气特性如表** 4.19 **所示。**

表 4.19　ADC **交流电气特性表**

交流电气参数	符号	最小值	典型值	最大值	单位
建立时间	Tint	0.33			s
采样时间	Tsh	2			μs
通道选择时间	Tsel	1			μs
VRT 开关时间	Tsw	1			μs
转换时间	Tconv	16			μs

3. ADC **的管脚功能**

ADC **管脚功能如表** 4.20 **所示。**

表 4.20　ADC **管脚功能表**

ADC 管脚	I/O	功能描述
Vref2	O	2 V 输出参考电压
MICP	I	MIC_IN 的正相输入
MICN	I	MIC_IN 的负相输入
AGC	I/O	自动增益控制端
MICOUT	O	麦克风的第一运放输出
VIN[7～1]	I	模拟信号源输入[7～1]源自通用 I/O 口
OPI	I	麦克风的第二运放输入
VCM	O	ADC 参考电压
VRTPAD	I	外部 A/D 最高参考电压
VMIC	I	麦克风的电源

例 6　ADC **转换实验。**

利用模拟量输入口 LINE_IN **输入电压，通过读取** P_ADC_MUX_Data **单元可以获得** 10 **位** A/D **转换的数据，而从该单元读取数据后又会使** RDY **自动清** 0，**重新启动** A/D **转换。若未读取** P_ADC_MUX_Data **单元中的数据** RDY **仍保持为** 1，**则不会启动下一次的** A/D **转换。**

程序如下：

```
.define P_IOB_DATA 7005H
.define P_IOB_DIR 7007H
```

```
        .define P_IOB_ATTRI 7008H
        .define P_INT_Ctrl 7010H
        .define P_INT_CLEAR 7011H
        .define P_ADC_Ctrl 7015H
        .define P_ADC_MUX_Ctrl 702BH
        .define P_ADC_MUX_DATA 702CH
        .define P_DAC_Ctrl 702AH
    .CODE
        .PUBLIC _main                   //主程序
    _main:
        r1=0xFFFF
        [P_IOB_ATTRI]=r1                //IOB 口设置为带数据缓存器的低电平输出口
        [P_IOB_DIR]=r1
        r1=0x0000
        [P_IOB_DATA]=r1
        R1=0x0001                       //选择通道 LINE_IN1
        [P_ADC_MUX_Ctrl]=R1
        R1 =0x0001                      //设置 P_ADC_Ctrl 单元允许 A/D 转换
        [P_ADC_Ctrl]=R1
        NOP                             //等待
        NOP
        NOP
        r2=0x0000                       //r2 的初值为 0x0000
    _AD:
        r1=[P_ADC_MUX_Ctrl]
        TEST r1,0x8000                  //判断是否转换完毕
        JZ_AD                           //是否继续等待
        r1=[P_ADC_MUX_DATA]             //是则读出转换结果同时重新启动 A/D 转换
        [P_IOB_DATA]=r1                 //IOB 口输出转换结果
    JMP_AD                              //继续检测下一次转换
```

例 7　采用自动方式即定时器 A 溢出执行 ADC 转换，通过 A/D 将 MIC_IN 输入的语音信号转换为数字信号，再通过 D/A 的两个通道 AUD1 和 AUD2 播放。

程序如下：

```
        .define P_ADC_Ctrl 7015H
        .define P_ADC_MUX_Ctrl 702BH
        .define P_ADC_MUX_DATA 702CH
        .define P_DAC_Ctrl 702AH
    .CODE
        .PUBLIC _main                   //主程序
```

```
_main:
    r1=0xFFFF                    //r1 的值为 0xFFFF
    [P_IOB_ATTRI]=r1             //IOB 口设置为带数据缓存器的低电平输出口
    [P_IOB_DIR]=r1
    r1=0x0000
    [P_IOB_DATA]=r1
    R1=0x0001;                   //选择通道 LINE_IN1
    [P_ADC_MUX_Ctrl]=R1
    R1 = 0x0001;                 //设置 P_ADC_Ctrl 单元允许 A/D 转换
    [P_ADC_Ctrl] = R1
    NOP                          //等待
    NOP
    NOP
    r2=0x0000                    //r2 的初值为 0x0000
_AD:
    r1=[P_ADC_MUX_Ctrl]          //判断是否转换完毕
    TEST r1,0x8000
JZ_AD                            //是否继续转换
    r1=[P_ADC_MUX_DATA]          //是则读出转换结果同时重新启动 A/D 转换
    [P_IOB_DATA]=r1              //IOB 口输出转换结果
    JMP_AD                       //跳转到_AD 处

.PUBLIC _IRQ1                    //中断服务程序
_IRQ1:
    PUSH R1 TO [SP]              //压栈
    R1 = [P_ADC]                 //读取 A/D 转换的数值
    [P_DAC1] = R1                //DAC 输出
    [P_DAC2] = R1
    R1 = 0x1000
    [P_INT_Clear] = R1           //清中断标识位
    POP R1 FROM [SP]             //出栈
    RETI                         //中断返回
```

4.5 DAC 音频输出

4.5.1 硬件组成

SPCE061A 提供了双通道音频输出方式。数字量分别写入 P_DAC1(写)(7017)和 P_DAC2(写)(7016)单元。DAC1、DAC2 转换输出的模拟量为电流信号，分别通过 DAC1

和 DAC2 管脚输出，如图 4.11 所示。

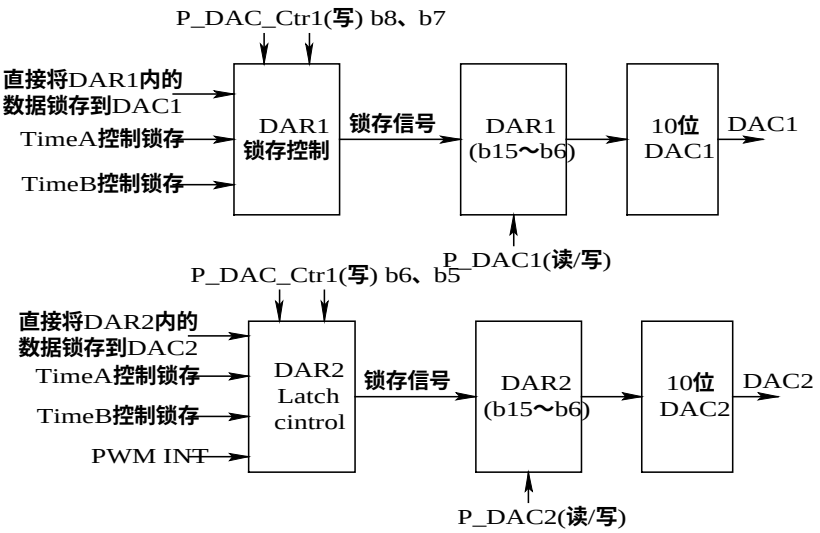


图 4.11　音频输出原理图

4.5.2　寄存器设置

1. DAC 音频输出方式控制单元 P_DAC_Ctrl

b5～b8 选择 DAC 数据锁存方式，b3 和 b4 控制 A/D 转换方式，b1 总为 0，用于双 DAC 音频输出，b9～b15 为保留位。表 4.21 详细地列出了 P_DAC_Ctrl(写)(702AH)单元的 b3～b8 的控制功能。

表 4.21　P_DAC_Ctrl 单元的 b3～b8 位的功能

b8	b7	b6	b5	b4	b3
DAC1_Latch(写)		DAC2_Latch(写)		AD_Latch(写)	
00：直接将 DAR1 内数据锁存到 DAC1 内 (缺省设置)		00：直接将 DAR2 内的数据锁存到 DAC2 内 (缺省设置)		00 通过读 P_ADC(读)(7014H)触发 ADC 自动转换 (缺省设置)	
01：通过 TimerA 溢出将 DAR1 内的数据锁存到 DAC1 内		01：通过 TimerA 溢出将 DAR2 内的数据锁存到 DAC2 内		01 通过 TimerA 溢出触发 A/D 转换	
10：通过 TimerB 溢出将 DAR1 内的数据锁存到 DAC1 内		10：通过 TimerB 溢出将 DAR2 内的数据锁存到 DAC2 内		10 通过 TimerB 溢出触发 A/D 转换	
11：通过 TimerA 或 TimerB 的溢出将 DAR1 内的数据锁存到 DAC1 内		11：通过 TimerA 或 TimerB 的溢出将 DAR2 内的数据锁存到 DAC2 内		11 通过 TimerA 或 TimerB 的溢出触发 A/D 转换	

2. P_DAC2

在 DAC 方式下，P_DAC2(读/写)(7016H)单元是一个带 10 位缓冲寄存器 DAR2 的 10 位 D/A 转换单元(DAC2)。使用该单元的 b15～b6 这 10 位，b5～b0 未用。

P_DAC2(写)：通过此单元直接写入 10 位数据到缓存器 DAR2，来锁存 DAC2 的输入数字量(无符号数)。

P_DAC2(读)：从 DAR2 内读出 10 位数据。

3. P_DAC1

P_DAC1(读/写)(7017H)单元为一个带 10 位缓存器(DAR1)的 10 位 D/A 转换单元(DAC1)，用于向 DAR1 写入或从其中读出 10 位数据。使用该单元的 b15～b6 这 10 位，b5～b0 未用。

例 8　DAC 应用实验。采取直接方式通过编程输出一个锯齿波信号，将实验板的 DAC 输出端接示波器观察锯齿波形，同时也可以听到 AUD1 和 AUD2 两端的扬声器有持续间断的声音。

程序如下：

```
        .define P_DAC_Ctrl 702AH
        .define P_DAC1 7017H
        .define P_DAC2 7016H
        .code
        .PUBLIC _main;
    _main:
            INT Off;                    //关中断
            r1=0x0000;                  //设置 DAC 输出数据锁存方式，A/D 转换方式
            [P_DAC_Ctrl]=r1;
            r3=0x0040；                 //D/A 转换为 10 位即 b15～b6，递增值为 1
            r1=0x0000;
    MainLoop1:
            [P_DAC1]=r1;                //输出为 0
            [P_DAC2]=r1;
            CALL Delay                  //调用延时子程序
            r1+ =r3                     //幅度提高 1 个台阶
            JMP MainLoop1               //返回 MainLoop1
Delay:      r2=0000H；                  //延时子程序
Delay1:     r2+=1
            JNZ Delay1                  //记到溢出时 R2 变为 0 延时结束
        RETF                            //子程序返回
```

4.6　串行设备输入/输出端口 SIO

串行输入/输出端口 SIO 提供了一个 1 位的串行接口，用于与其它设备进行数据通信。SPCE061A 通过 IOB0 和 IOB1 这 2 个端口实现与设备进行串行数据交换功能，IOB0 作为时钟端口(SCK)，IOB1 作为数据端口(SDA)，用于串行数据的接收或发送。通过以下寄存器来设定其工作方式。

4.6.1　寄存器设置

1. P_SIO_Ctrl

通过设置 P_SIO_Ctrl(读/写)(701EH)单元的第 7 位，若 b7=1，则将 IOB0、IOB1 分别设置为 SCK 管脚和 SDA 管脚；若 b7=0，则 IOB0、IOB1 为普通 I/O 端口。该单元的 b6 设置串行输入/输出端口的收发传输，该单元的 b3、b4 设置数据传输速率，而通过设置 b0、b1 可以指定串行设备的寻址位数，如表 4.22 所示。

表 4.22　P_SIO_Ctrl 控制单元功能

b7	b6	b5	b4	b3	b1	b0	设置功能说明
IO	R/W	R/W_EN	Speed_Sel		Addr_Sel		
X	X	X	X	X	0	0	串行设备地址(缺省)设置为 16 位(A15～A0)
X	X	X	X	X	0	1	无地址设置
X	X	X	X	X	1	0	串行设备地址设置为 8 位(A0～A7)
X	X	X	X	X	1	1	串行设备地址设置为 24 位(A23～A0)
X	X	X	0	0	X	X	数据传输速率设为 CPUClk/16(缺省设置)
X	X	X	0	1	X	X	数据传输速率设为 CPUClk/4
X	X	X	1	0	X	X	数据传输速率设为 CPUClk/8
X	X	X	1	1	X	X	数据传输速率设为 CPUClk/32
1	X	X	X	X	X	X	设置 IOB0 为 SCK，IOB1 为 SDA
0	X	X	X	X	X	X	用作普通的 I/O 口(默认)
X	1	X	X	X	X	X	设置数据帧的写传输
X	0	X	X	X	X	X	设置数据帧的读传输(默认)
X	X	1	X	X	X	X	关断读/写帧的传输
X	X	0	X	X	X	X	接通读/写帧的传输(默认)

2. P_SIO_Data

P_SIO_Data(读/写)(701AH)单元为接收/发送串行数据的缓冲单元，向该单元写入或读出数据，可按串行方式发送或接收数据字节。通过写入 P_SIO_Start(701FH)单元启动 P_SIO_Data(701AH)单元与串行设备的数据交换，传输是从串行设备的起始地址(由 P_SIO_Addr_Low、P_SIO_Addr_Mid 和 P_SIO_Addr_High 三个单元指定)开始，然后是数据。

写操作：在写入 P_SIO_Start(写)单元任意一个数值之后，第一次向 P_SIO_Data(写)(701AH)单元写入数值，随后，SIO 将首先传送串行设备的起始地址，接着传送写入 P_SIO_Data 单元中的 8 位数据，使用低 8 位(b7～b0)，高 8 位(b15～b8)未用。

读操作：在向 P_SIO_Start(写)(701FH)单元写入任一数值后，再读 P_SIO_Data(读)(701AH)单元数据，SIO 将首先传送串行设备的起始地址。

b7	b6	b5	b4	b3	b2	b1	b0
D7	D6	D5	D4	D3	D2	D1	D0

3. P_SIO_Addr_Low

P_SIO_Addr_Low(读/写)(701BH)单元的串行设备起始地址的低字节(默认值为 00H)功能：

b7	b6	b5	b4	b3	b2	b1	b0
A7	A6	A5	A4	A3	A2	A1	A0

4. P_SIO_Addr_Mid

P_SIO_Addr_Mid(读/写)(701CH)单元的串行设备起始地址的中字节(默认值为 00H)功能：

b7	b6	b5	b4	b3	b2	b1	b0
A15	A14	A13	A12	A11	A10	A9	A8

5. P_SIO_Addr_High

IP_SIO_Addr_High(读/写)(701DH)单元的串行设备起始地址的高字节(默认值为 00H)功能：

b7	b6	b5	b4	b3	b2	b1	b0
A23	A22	A21	A20	A19	A18	A17	A16

6. P_SIO_Start

向 P_SIO_Start(读/写)(701FH)单元写入任意数值，可以启动数据传输过程。接着当对 P_SIO_Data (701AH)单元读写操作时，会使 SIO 根据 P_SIO_Addr_Low 、P_SIO_Addr_Mid 和 P_SIO_Addr_High 的内容传输读/写操作的起始地址，再读写 P_SIO_Data 单元时，SIO 将不再传输此起始地址。

如果需要重新指定一个起始地址，可以先向 P_SIO_Stop (7020H)单元写入任意数值，停止 SIO 操作，然后向 P_SIO_Addr_Low、P_SIO_Addr_Mid 和 P_SIO_Addr_High 写入新的地址，再向 P_SIO_Start(701FH)单元写入任意数值，重新启动 SIO 操作即可。

读出 P_SIO_Start(701FH)单元，可查询 SIO 的数据传输状态，该单元的 b7 位 Busy 为占用标志位，Busy=1 表示正在传输数据，传输操作完成后，该位将被清为 0，可以开始传输新的数据字节。

7. P_SIO_Stop

向 P_SIO_Stop(写)(7020H)单元写入任一数值，作为停止数据传输指令。通常在激活数据传输的启动指令之前，应先使用停止数据传输的终止指令，但上电复位后的第一个启动命令之前不需要终止命令。

4.6.2 应用举例

通过串行输入/输出端口将 SPCE061A 与 SPRS512C 串行静态 RAM(SSRAM，Serial Static RAM)连接起来，在数据传输过程中 SPRS512 会自动在起始地址的基础上递增地址计数器。硬件连接如图 4.12 所示，时序如图 4.13 所示。

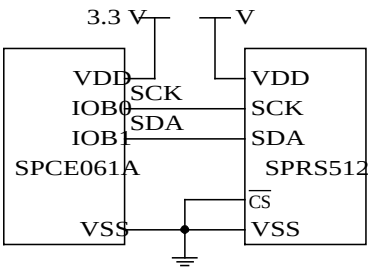


图 4.12　SPCE061A 与 SPRS512 连接图

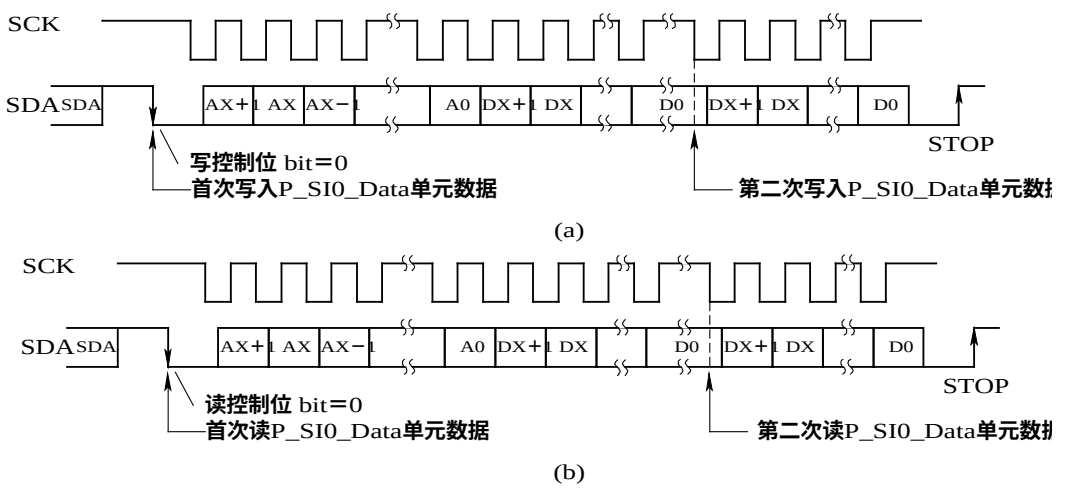


图 4.13　SIO 发送/接收时序

例 9　从串行输入/输出端口向 SSRAM 写入一个长度为 1 个字节的数据，SPRS512 有 24 位地址线，工作频率为 CPUCLK/4，可进行串行数据帧的写操作。

程序如下：

```
                                //设置串行输入/输出控制端口(输出状态)
        R1=0x00CB               // R1=0000 0000 110 01011B
        [P_SIO_Ctrl]=R1         //写入串行输入输出启动端口
        R1=0x0000
        [P_SIO_Start]=R1        //写入起始地址到地址单元
        R1=0x0000
        [P_SIO_Addr_High]=R1    // Addr23～Addr16 =0000 0000B
        [P_SIO_Addr_Mid]=R1     //Addr15～Addr8 =0000 0000B
        [P_SIO_Addr_Low]=R1     //Addr7～Addr0 =0000 0000B
                                //向缓冲单元写入传送数据
        R1=0x0011
        [P_SIO_Data]=R1         //d7～d0=0001 0001B
L_busy：R1=0x0080               //查询占用标志 Busy，等待传输结束
```

```
Test R1,[P_SIO_Start]
JNZ L_Busy
[P_SIO_Stop]=R1                    //向 P_SIO_STOP(写)(7020H)单元写入任意一个数值，
                                   //结束数据帧的写操作
```

4.7 通用异步串行通信口 UART

UART 模块提供了一个全双工标准通信口，用于完成 SPCE061A 与外设之间的串行通信。根据 RS−232 的标准，SPCE061A 单片机也是按照字节传输数据的。利用 IOB 口的特殊功能和 UART IRQ 中断，可以同时完成 UART 接口数据的接收和发送。此外，UART 还可以带缓冲接收数据，即可以在读取缓存器数据之前接收新的数据。但是，如果新的数据被接收到缓存器之前一直未从中读取，先前的数据会发生数据丢失。P_UART_Data (读/写)(7023 H)单元用于接收和发送数据的缓存，向该单元写入数据，将发送的数据送入缓存器；读该单元取数据，可以从缓存器读出接收到的单字节数据。UART 模块的接收管脚 Rx 和发送管脚 Tx 分别与 IOB7 和 IOB10 共用。通过 UART 接收管脚 Rx(IOB7)来接收数据时须保证 P_IR_Ctrl(7018H)单元的 b2 位 RXEN 被置为 0，即允许接收。UART 的示意图如图 4.14 所示。

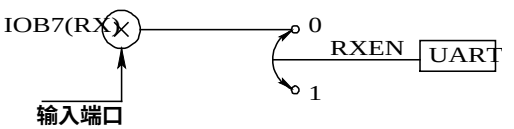


图 4.14 UART 示意图

4.7.1 UART 数据帧格式

SPCE061A 单片机串行通信帧结构为：1 bit 启动位，8 bit 数据位，1 bit 奇偶校验位，1 bit 停止位，如图 4.15 所示。

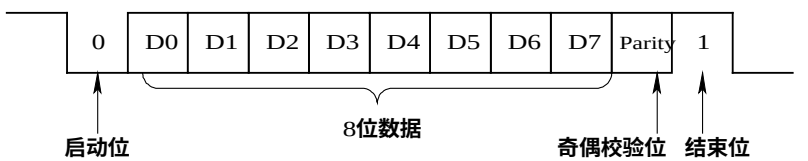


图 4.15 UART 帧格式

4.7.2 工作寄存器设置

使用 UART 模块进行通信时，必须事先分别将管脚 Rx(IOB7)设置为输入状态，Tx(IOB10)设置为输出状态；然后，设置 P_UART_BaudScalarLow(7024H)、P_UART_BaudScalar High(7025H)单元，指定所需波特率；设置 P_UART_Command1(7021H)和

P_UART_ Command2(7022H)单元以激活 UART 通信功能。以上设置完成后，UART 将处于激活状态。

1. 红外通信控制单元 P_IR_Ctrl(读/写)(7018H)

红外通信控制单元 P_IR_Ctrl(读/写)仅使用 b2 位 RXEN(写)为 1 禁止接收；为 0 允许接收。

2．串行口控制寄存器 1 P_UART_Command1(写)(7021H)

串行口控制寄存器仅使用 P_UART_Command1 的 b2～b7 这 5 位，其它位未用。

b2(P_Check)　1：激活奇偶校验功能；0：屏蔽奇偶校检功能。

b3(Parity)　1：激活偶校检功能；0：激活奇校检功能。

b5(I_Reset)　1：控制器状态恢复为默认值；0：控制器状态为设定值。

b6(TxIntEn)　1：允许由 TxRDY 信号触发的 UART IRQ 中断；

0：禁止由 TxRDY 信号触发的 UART IRQ 中断。

b7(RxIntEn)　1：允许由 RxRDY 信号触发的 UART IRQ 中断；

0：禁止由 RxRDY 信号触发的 UART IRQ 中断。

P_UART_Command1(写)(7021H)单元的缺省值为 0000H。

该单元为 UART 控制寄存器，设置该单元的第 2、3 位，可以控制数据奇偶校验功能。设置 b6、b7 位，可以激活 UART IRQ 中断，并决定中断是由 TxRDY 或 RxRDY 信号触发，以及由二者共同触发。b6、b7 位控制着 UART IRQ 中断，二者的区别在于，如果 b6 位 TxIntEn=1 中断由 TxRDY 信号触发，即数据发送完毕将产生 UART IRQ 中断；如果 b7 位 RxIntEn=1 中断由 RxRDY 信号触发，即数据接收完毕将产生 UART IRQ 中断。

如果该单元的 b5 位 I_Reset=1，所有 UART 控制寄存器状态将恢复为系统默认值。

3. 串行口控制寄存器 2 P_UART_Command2

1) 写入

P_UATR_Command2(读/写)(7022H)单元写入时，为 UART 数据发送/接收控制端口 b6、b7 位分别控制着数据发送和接收管脚的允许/禁止。

设置 P_UART_Command2 单元的 b6、b7 位可以激活 UART Tx 和 Rx 管脚功能，当 SPCE061A 接收或发送一个字节数据时，P_UART_Command2 (7022H)单元的 b6、b7 位被置 1，同时触发 UART IRQ，无论 UART IRQ 中断是否被激活，UART 接收/发送功能都可以由 P_UART_Command2 (7022H)单元的 b6、b7 位控制。

注意：UART IRQ 中断向量存储在 FFFFH 单元，相对于其它 IRQ 中断来说，该中断的优先级别最低。

b6(TxPinEn)　1：允许发送；0：禁止发送。

b7(RxPinEn)　1：允许接收；0：禁止接收。

P_UART_Command2 单元的缺省值为 0000H。

此时，IOB7 和 IOB10 必须分别被设置为输入和输出管脚作为 Rx 和 Tx。当允许发送时，IOB10 的 Tx 输出管脚将自动被置为高电平。

2) 读出

P_UART_Command 2(读/写)单元读出为 UART 状态信息，b7 位是 RxRDY 标志位，当接收到数据时，该标志位被置为 1，读 P_UART_Data 单元将清除该标志位。b6 位是 TxRDY 标志位，置 1 为允许发送。该标志位被置为 1，表示发送器的数据缓存器空，已准备好，可以发送(写入 P_UART_Data 单元的)数据。

b7(RxRDY) 1：接收到数据；0：未接收到数据。

b6(TxRDY) 1：数据发送已准备好；0：数据发送未准备好。

b5(FE) 1：帧错误，不能接收完整的一帧数据；0：无帧错。

b4(OE) 1：溢出错误；0：无溢出错。

b3(PE) 1：奇偶校验错误；0：无奇偶校验错。

向 P_UART_Data 单元写入数据，可以清除 TxRDY 标志位。读出 P_UART_Command2 (7022 H)单元的 b3～b5 位是传输错误标志位，如果在传输过程中发生错误，相应位将被置为 1。

在任意时刻读出 P_UART_Command2(7022H)单元，将清除 UART IRQ 中断标志。读 P_UART_Data(7023H)单元数据将清除错误标志位。

串行口通信出错的原因及解决方法如表 4.23 所示。

表 4.23 串行口通信出错的原因及解决方法

错误类型	原因	解决方法
FE (帧错误)	发送管脚 Tx 和接收管脚 Rx 的数据帧的格式或波特率不一致	(1) 使用一致的数据格式 (2) 设置一致的波特率
OE (溢出错误)	接收端 Rx 接收数据的速度低于发送端 Tx 发送数据的速度从而导致 Rx 端数据溢出	(1) 提高接收数据的速率 (2) 降低数据传输速率
PE (奇偶校验错误)	传输条件差可能有噪声干扰	改善传输条件

4. P_UART_Data(读/写)(7023H)

P_UATR_Data 单元读/写 8 位数据 d7～d0。

5. UART_BaudScalarLow 和 P_UART_BaudScalarHigh

P_UART_BaudScalarHigh(读/写)(7025H)和 P_UART_BaudScalarLow(读/写)(7024H)单元的组合控制着数据传输速率(波特率)，通过设置波特率控制器初值来设定通信速率。UART 波特率的计算公式如下：

设波特率为 B，波特率控制器初值为 Scale，则有

$$B=(Fosc/2)/Scale$$

所以

$$Scale=Fosc/(2\times B)$$

其中：Fosc 为系统频率。

7024 H 单元和 7025H 单元分别存放 N 的高字节和低字节。

常见的波特率值如表 4.24 所示。

表 4.24　常用波特率值(Fosc = 24.576 MHz)

波特率/(b/s)	高字节 (7025H)	低字节 (7024H)	Scale (十进制)	实际波特率 /(b/s)
1500 (最小值)	20H	00H	8192	1500
2400	14H	00H	5120	2400
4800	0AH	00H	2560	4800
9600	05H	00H	1280	9600
19 200	02H	80H	640	19 200
38 400	01H	40H	320	38 400
48 000 (默认值)	01H	00H	256	48 000
51 200	00H	F0H	240	51 200
57 600	00H	D5H	213	57 690
102 400	00H	78H	120	102 400
115 200 (最大值)	00H	6BH	107	114 841

例 10　用 UART 来接收 PC 机 RS−232 串行口发送的 2048 个字，设 Fosc = 24.576 MHz，串行口速率为 48 000 b/s。

解：由于 UART 每次只接收 8 位数据，所以在接收了两个 8 位数据之后，才将其存入 16 位的 SRAM 内。

初始化编程：

```
//将 IOB10、IOB7 分别设为输出、输入状态
        R1=0x0480                   //R1=0000 0100 1000 0000B
        [P_IOB_Attrib]=R1
        R1=0x0400                   //IOB10 同相输出 TX；IOB7 输入 RX
        [P_IOB_Dir]=R1
    //设置波特率：选择波特率为 48 000，N=24 576 000/48 000/2＝256D=0100H
        R1=0x0000
        [P_UART_BaudScalarLow]=R1
        R1=0x01
        [P_UART_BaudScalarHigh]=R1
     //设置 UART_Command1 和 UART_Command2 单元
        R1=0xc0                     //R1=1100 0000B
        [P_UART_Command1]=R1        //允许接收、发送中断
        [P_UART_Command2]=R1        //允许发送、允许接收
    //准备接收数据
L_begin_loop：R4=0                  //SRAM 的接收数据的起始地址
L_loop：    R2=0
        R2=R2 LSL 4                 //清除移位缓存器 S3～S0=0
```

```
                CALL F_UART_RECV      //调用接收子程序
                R1=R1 LSL 4           //R1 左移 8 位
                R1=R1 LSL 4
                R3=R1
                R3=R3 and 0xFF00      //R3=高 8 位
                R2=0
                R2=R2 LSL 4           //清除移位缓存器
CALL F_UART_RECV：
                R1=R1 and 0X00FF      //R1=低 8 位
                R1=R1 or R3           //R1=从 UART 接收来的最终数据字
                [R4++]=R1             //将接收的数据字存入 SRAM 的数据队列
                CMP R4 0x0800         //SRAM 中的数据队列接收到 2048 个字
                JNE L_loop
                JMP L_begin_loop      //从新开始接收
```

UART 接收子程序：

```
F_UART_RECV：   PUSH R2 R3 TO [SP]
L_RxRDY：       R2= 0x0080            //查寻 RxRDY 是否为 1
                TEST R2 [P_UART_Command2]
                JZ L_RxRDY
                R1=[P_UART_Data]
                POP R2 R3 FROM[SP]
                RETF                  //返回到主程
```

4.8 工 作 方 式

4.8.1 睡眠方式

1. 睡眠

SPCE061A 上电复位开始工作后，当接收到睡眠信号时，关闭系统时钟(PLL 振荡器)进入睡眠状态。通过对 P_SystemClock(写)(7013H)单元写入 CPUCLK STOP 控制字(CPU 睡眠信号 b2 b1 b0=111B)，使系统由运行状态转入睡眠状态，程序计数器(PC)会停在程序的下一条指令，当有任一唤醒事件发生后，开始由此继续执行程序。

2．唤醒

系统接收到唤醒信号后，接通系统时钟(PLL 振荡器)，程序从进入睡眠时指令计数的断点处继续执行。同时 CPU 会响应唤醒事件的处理，并初始化 IRQ3_KEY 为触键唤醒源(IOA7～0)，其它中断源 FIQ、IRQ1～IRQ6 及 UART IRQ 都可以作为唤醒源(关于触键唤醒源请参考 4.1.3 节)。

4.8.2　低电压监测/低电压复位(LVD/LVR)

SPCE061A 可通过屏蔽选项设置或取消低电压监测功能，为了通过对系统电源电压进行监控，使系统运行在一个正常可靠的工作环境下，一旦出现电源异常的情况，立即采取相应的措施，使系统及时恢复正常。

1. 低电压监测(LVD)

低电压监测功能可以提供系统电源电压的使用情况，如果系统电压 VCC 低于用户设定的电压监测低限值 VLVD，P_LVD_Ctrl 单元的 b15 位(LVD 监测标志位)将被置为 1；反之，当 VCC> VLVD 时，该位置为 0。

SPCE061A 具有 4 级电压监测低限：2.4 V、2.8 V、3.2 V 和 3.6 V。可通过对 P_LVD_Ctrl 单元编程来控制。假定 VLVD=3.2 V, 当系统电压 VCC 低于 3.2 V 时，P_LVD_Ctrl 单元的 b15 位置 1。CPU 通过可编程电压监测低限来完成低电压监测，系统默认的电压监测低限为 2.4 V。

2. P_LVD_Ctrl(读/写)(7019H)

P_LVD_Ctrl 寄存器的低电压监测设置如表 4.25 所示。

表 4.25　低电压监测设置

选择位			电压监测低限(VLVD)
b15	b1	b0	
Result_of_LVD	LVD_level_define(写)		
0： VDD > VLVD 1： VDD < VLVD	0	0	2.4 V(默认)
	0	1	2.8 V
	1	0	3.2 V
	1	1	3.6 V

3. 低电压复位(LVR)

当电源电压低于 2.2 V 时，系统会变得不稳定且易出故障。导致电源电压过低的原因很多，如电压的反跳、负载过重、电池能量不足等。如果系统设置了低电压复位(LVR)功能，当电源电压低于该值时，会在 4 个时钟周期之后产生一个复位信号使系统复位，从而使系统恢复到初始工作状态。

4.8.3　看门狗监视器

设置看门狗监视器的目的是为了监测系统是否正常运行，看门狗监视器实际上是一个计数器，它每隔一段时间必须被人为清除(SPCE061A 提供的清除周期为 0.75 s)，否则，将产生计数溢出，使 CPU 认为系统运行异常，导致系统复位且重新开始执行程序。

看门狗监视器的保护作用表现为当发生看门狗计数器计数溢出时，系统复位。看门狗监视器功能可以通过指令设置或屏蔽。

看门狗清除寄存器 P_Watchdog_Clear(写)(7012 H)在 SPCE061A 系统正常运行时，只要

每隔小于 0.75 s 向 P_Watchdog_Clear(写)(7012H)单元写入 XXXX XXXX XXXX XX01B 即可清除看门狗计数器(俗称喂狗)，这样便不会发生看门狗计数器计数溢出，在正常运行时不会导致系统被复位。如果由于电源干扰等原因，使系统程序无法正常运行时，不能按时执行写操作，无法清除计数器的值，计数器就会溢出产生一个复位信号使系统复位，防止了系统出现误动作现象。在备用状态下若 32 768 Hz 振荡器工作，看门狗计数器功能也可设置。

4.8.4 保密设定

如果希望将内部的闪存进行保密设定，可将 29 引脚 PFUSE 接 5 V 和 20 引脚 PVIN 接 GND，并维持 5 s 以上，即可将内部保险丝熔化，此后就无法再完成 download(下载)、debug(调试)等功能，使写入单片机的程序无法读出。

4.8.5 语音编码类型

在此只给出 SPCE061A 单片机支持的语音编码方式，具体功能请参见第 6 章有关语音信号处理技术的内容。SPCE061A 支持的语音编码方式如表 4.26 所示。

表 4.26 SPCE061A 支持的语音编码方式

编码方式	编码名称	功能
脉冲编码调制方式	PCM	
对数脉冲编码调制方式	LOG PCM	
凌阳音频编码方式	SACM_A2000	以 24 K、20 K、16 Kb/s 的速率编码/解码
	SACM_S720	以 7.2 Kb/s 的速率编码/解码
	SACM-S480	以 4.8 Kb/s 的速率编码/解码
	SACM_S240	以 2.4 Kb/s 的速率编码/解码
	SACM_MS01	用于音调合成器的 FM 合成器和波表合成器

思 考 题

1. SPCE061A 单片机有哪些并行 I/O 端口？各具有什么功能？
2. 设置并行 I/O 端口功能需用哪些寄存器？请简述其功能。
3. 哪些 I/O 端口具有唤醒功能？
4. 并行 I/O 端口 B 有哪些特殊功能？如何设置？
5. 请画出 SPCE061A 单片机的外接时钟电路，一般选取实时时钟频率是多少？
6. 系统时钟信号是如何产生的？怎样设置？SPCE061A 默认的系统时钟频率是多少？
7. 怎样产生时基信号？如何设置时基频率？
8. 请画出 SPCE061A 单片机的实用复位电路。
9. SPCE061A 单片机具有哪些定时器/计数器，请简述其作用。
10. 如何设置定时器/计数器功能？请编程利用定时器 A，在 IOB0 位输出一频率为 50 Hz 的方波信号。

11. SPCE061A 单片机有哪些 A/D 通道？请简述各自的功能。

12. 设置 A/D 功能要用到哪些寄存器？

13. 请编程利用 D/A 转换器输出频率为 100 Hz 的锯齿波。

14. 怎样设置 UART 功能？请编程利用 SPCE061A 单片机接收 PC 机 RS−232 串行口发送的 16 个 8 位数据，存于 0x0100 起始的存储单元。其中，Fosc=24.576 MHz, PC 机 RS−232 发送波特率为 9600 b/s。

15. SPCE061A 单片机是怎样实现看门狗监视器功能的？

16. 利用 SPCE061A 的 SIO 功能实现与 SSRAM SPRS512 数据传输。请编程将 SPCE061A 内存 0020H～0030H 单元的字按字节形式写入 000000H 起始的 SPRS512 存储器内。

第 5 章　中断系统及应用

5.1　概　　述

中断系统是为 CPU 对外部事件做出快速反应和处理而设置的，中断技术的引入极大地推进了计算机的发展，使计算机在实时控制方面得到了广泛的应用，因此中断功能的强弱已成为衡量一种计算机性能的重要标志。

本节只对中断的基本概念进行简要介绍。

1. 中断

中断是指计算机在执行某一程序的过程中，由于计算机系统内、外的某种原因而必须终止原程序的执行，转去完成相应的处理程序，待处理结束之后再返回继续执行被终止原程序的过程。

中断技术能实现 CPU 与外部设备的并行工作，提高 CPU 的利用率及数据的输入/输出效率，能及时发现计算机运行过程中的突发事件并自动处理，如计算机对外部事件、硬件故障、运算错误、定时器溢出等做出响应。触键唤醒中断能使我们通过键盘发出申请，随时对运行中的 CPU 提出请求，转而完成一些必须马上处理的事件。

2. 中断源

中断源是指在计算机系统中向 CPU 发出中断请求的事件。中断源可以是人为设定的，也可以是突发的随机事件。如定时器中断就是由定时器溢出产生的中断。

3. 中断优先级

由于在实际应用系统中往往有多个中断源，且中断申请是随机的，有时还可能会有多个中断源同时提出中断申请，但 CPU 一次只能响应一个中断源发出的中断请求，这时 CPU 响应哪个中断请求，就需要用软件或硬件按中断源工作性质的轻重缓急给它们安排一个优先顺序，即所谓的优先级排队。中断优先级越高则响应优先权就越高。当 CPU 正在执行中断服务程序时，又有中断优先级更高的中断申请产生，CPU 就会暂停原来的中断处理程序而转去处理优先级更高的中断请求，处理完毕后再返回原低级中断服务程序，这一过程称为中断嵌套，具有这种功能的中断系统称为多级中断系统。没有中断嵌套功能的则称为单级中断系统。图 5.1 所示为具有二级中断服务程序嵌套的中断过程。

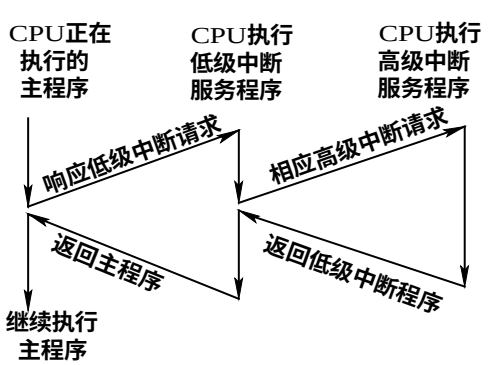


图 5.1　中断嵌套示意图

4. 中断响应过程

中断响应过程有以下几步：

(1) 检测中断。在每一条指令结束后，系统自动检测中断请求信号，如果有中断请求且相应的中断允许位为真(允许中断)时，则响应中断。

(2) 保护现场。CPU 一旦响应中断，中断系统会自动保存当前内容 PC 和 SR 寄存器内容(通过入栈完成)，执行中断服务程序，中断服务程序还可以通过压入堆栈来保护原程序中用到的数据和寄存器的值。保护现场前一般要关中断，以防止现场保护过程中有高优先级中断发生而破坏现场保护。保护现场一般是用堆栈指令将原程序中用到的寄存器压入堆栈，现场保护之后要开中断，以便响应更高优先级的中断申请。

(3) 中断服务。通过执行中断服务程序完成相应的功能。

(4) 清除中断标志位。CPU 响应中断后，要请除相应的中断请求标志位，以免 CPU 再次响应该中断。

(5) 恢复现场。中断服务完成后，返回之前要用弹出堆栈指令使保护在堆栈中的数据和寄存器的值弹出，以实现恢复原有数据的目的。

注意：在恢复现场前要关中断，以防止恢复现场过程中再有中断响应破坏恢复现场。现场恢复后应及时开中断。

(6) 中断返回。此时 CPU 将 PC 指针和 SR 内容弹出堆栈恢复断点，从而使 CPU 继续执行刚才被中断的程序。

在单片机中，中断技术主要用于实时控制和及时处理外部突发事件。要求计算机能及时响应被控对象提出的分析计算和控制等请求，使被控对象保持在最佳工作状态，以达到预定的控制效果。由于这些控制参量的请求都是随机发生的，而且要求单片机必须做出快速响应并及时处理，对此只有利用中断技术才能实现。

5.2　SPCE061A 中断系统

SPCE061A 单片机中断系统是凌阳 16 位单片机系列中中断功能较强的一种，它提供了 14 个中断源，具有两个中断优先级，可实现两级中断嵌套，可以使用关中断指令来屏蔽所有的中断请求，也可以用开中断指令使 CPU 接受中断申请。每一个中断源可以用软件独立控制为开放或屏蔽状态，但中断优先级不能用软件设置。

5.2.1　中断类型

SPCE061A 单片机提供了三种类型的中断：软件中断、异常中断和事件中断。

1. 软件中断

软件中断是由软件指令 break 产生的中断，软件中断的向量地址为 FFF5H。

2. 异常中断

异常中断是非常重要的事件，该中断一旦发生，CPU 必须立即进行处理。目前，

SPCE061A 定义的异常中断只有复位一种。通常系统复位可以由以下三种情况引起：上电、看门狗计数器溢出和系统电源电压低于限值。不论什么情况引起复位，都会使复位引脚的电位变低，进而使程序指针 PC 指向由复位向量 FFF7H 所指的系统复位程序入口地址。

3. 事件中断

事件中断简称中断，以下提到的中断均为事件中断，一般产生于片内某部件或由外设中断输入引脚引入的某个事件，这种中断的开放/屏蔽由对应位和相应的 IRQ 或 FIQ 总使能控制，由软件设置。

事件中断可采用两种方式：快速中断请求(FIQ)和中断请求(IRQ)，这两种中断都由相应的总使能来控制。

5.2.2 中断源

SPCE061A 有 14 个中断源，分别为两个定时器溢出中断、两个外部中断、一个串行通信口中断、一个触键唤醒中断、7 个时基信号中断、PWM 音频输出中断等，如表 5.1 所示。

表 5.1 中断源列表

中断源	中断优先级	中断向量	保留字
Fosc/1024 溢出信号 PWM INT 音频输出	FIQ/IRQ0	FFF6H/FFF8H	_FIQ/_IRQ0
TimerA 溢出信号	FIQ /IRQ1	FFF6H/FFF9H	_FIQ/_IRQ1
TimerB 溢出信号	FIQ /IRQ2	FFF6H/FFFAH	_FIQ/_IRQ2
外部信号输入 EXT2	IRQ3	FFFBH	_IRQ3
外部信号输入 EXT1			
触键唤醒信号			
4096 Hz 时基信号	IRQ4	FFFCH	_IRQ4
2048 Hz 时基信号			
1024 Hz 时基信号			
4 Hz 时基信号	IRQ5	FFFDH	_IRQ5
2 Hz 时基信号			
频选信号 TMB1	IRQ6	FFFEH	_IRQ6
频选信号 TMB2			
UART 串行通信中断	IRQ7	FFFFH	_IRQ7

SPCE061A 有九个中断向量，即快速中断请求 FIQ、中断请求(IRQ0～IRQ6)及 UART IRQ。这九个中断向量对应 14 个中断源，其中有三个中断源设置为 FIQ (或 IRQ0～IRQ2)，另有 10 个中断源设置为 IRQ3～IRQ6，还有一个专门用于通用异步串行通信口 UART 的

中断源设置为 UART IRQ。

从表 5.1 中可以看出有多个中断入口地址对应不止一个中断源，因此在中断服务程序中需通过中断请求标志位来判断是哪个中断源请求的中断。

1. 定时器中断

定时器中断由单片机内部定时器 TimerA 和 TimerB 产生。在定时脉冲作用下从预置数单元初值开始加 1 计数，当计数为 0xFFFFH 时，再加 1 便产生溢出，向 CPU 提出中断请求，以表明定时器的定时时间到，定时时间可通过程序设定，通常用于需要进行定时控制的场合。

2. 外部中断

SPCE061A 单片机有两个外部中断 EXT1 和 EXT2。两个外部输入引脚分别与 B 口的 IOB2 和 IOB3 复用，即 EXT1(IOB2)和 EXT2(IOB3)。引入两个外部中断源的中断请求信号，外部中断以负跳沿触发方式输入中断请求信号，如图 5.2 所示。

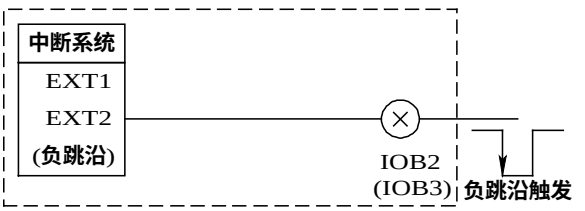


图 5.2　外部中断信号

EXT1、EXT2 可以由外部单脉冲产生中断请求信号，也可以通过接入相应的阻容元件实现定时中断。

在 IOB2 和 IOB4 之间以及 IOB3 和 IOB5 之间分别接入两个反馈电路，外接 RC 振荡器做外部定时中断使用，如图 5.3 所示，此时，外部中断的反馈电路使用四个管脚(B 口的 IOB2、IOB3、IOB4 和 IOB5 引脚)，其中 IOB4 和 IOB5 主要用来组成 RC 反馈电路，通过 IOB2 和 IOB4 之间(或者 IOB3 和 IOB5 之间)增加一个 RC 振荡电路，便可在 EXT1(或 EXT2)端得到振荡信号作为外中断触发信号。为使反馈电路正常工作，必须将 IOB2(或 IOB3)设置为反相输出方式，IOB4(或 IOB5)设置为输入方式。

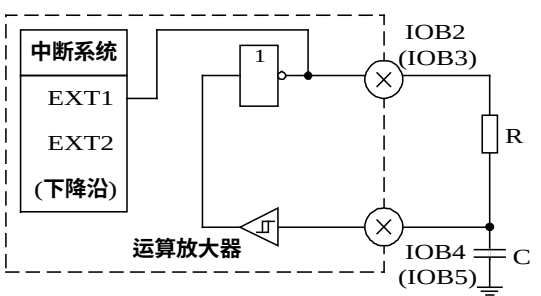


图 5.3　IOB2、IOB4 或 IOB3、IOB5 之间的反馈结构

3. 串行通信口中断

串行通信口中断由 SPCE061A 内部串行通信口中断源产生。串行通信口中断分为串行口发送中断和接收中断两种，共用一个中断向量。因此，在进入串行通信口中断服务程序后，需要判断是接收中断还是发送中断。

当串行通信口发送/接收完一组串行数据时，自动使串行通信口控制寄存器 P_UART_Command2 中的 TxReady 和 RxReady 中断标志置位，并向 CPU 发出中断请求，CPU 响应串行通信口中断后，转入中断服务程序执行。因此，只要在串行通信口中断服务程序中，安排一段对 P_UART_Command2 寄存器的 TxReady 和 RxReady 中断标志位的状态判断程序，便可区分发生了接收中断还是发送中断请求。当然串行传输中既可以使用中断方式收发数据，也可通过查询 TxReady 和 RxReady 中断标志位的状态收发数据。串行通信口接收、发送端为 B 口的 IOB7(RxD)和 IOB10(TxD)两个复用脚。

4. 触键唤醒中断

当系统执行睡眠命令时，CPU 便关闭 PLL 倍频电路，停止 CPU 时钟，使系统进入睡眠状态。系统处于睡眠状态时，通过 IOA 口低八位接的键盘给出唤醒信号，使系统接通 PLL 倍频电路，启动 CPU 时钟，将系统从睡眠状态转到工作状态，CPU 从睡眠处继续执行程序。与此同时产生一个 IRQ3_KEY 中断请求，进入唤醒中断。一般来讲，中断系统提供的中断源 FIQ(TMA)、IRQ1～IRQ7 均可作为系统的唤醒源，作为定时唤醒系统。

5. 时基信号中断

SPCE061A 单片机提供了多组时基信号能作为中断源。时基信号来自实时时钟 32 768 Hz，输出有通过选频逻辑的 TMB1、TMB2 信号和直接从时基计数器溢出的各种实时时基信号，在开启时基信号中断的情况下，有时基信号时，便发出时基信号中断申请，CPU 查询到有中断请求后，置位 P_INT_Ctrl 中相应的中断请求标志位，在中断服务程序中通过测试 P_INT_Ctrl 来确定是哪个频率的时基信号产生的中断请求。

5.2.3 中断控制

SPCE061A 单片机有多个中断源，为了使每个中断源都能独立地开放和屏蔽，便于灵活使用，为每个中断信号设置了一个中断屏蔽触发器，只有该触发器无效，该中断开放，它所对应的中断请求信号才能进入 CPU，否则即使其对应的中断请求标志位置 1，CPU 也不会响应中断，即该中断被屏蔽。CPU 内还设置了一个中断允许触发器来控制 CPU 能否响应中断，触发器的状态可以通过中断控制器寄存器设置。

1. 中断控制寄存器

SPCE061A 单片机对中断源的开放和屏蔽以及每个中断源是否被允许中断，都受中断允许寄存器 P_INT_Ctrl 和一些中断控制指令的控制。下面分别作一介绍。

1) 中断控制寄存器 P_INT_Ctrl(读/写)(7010H)

P_INT_Ctrl 控制单元具有读/写属性，其读写时表达的意义不同。中断控制寄存器功能如表 5.2 所示。

表 5.2　中断控制寄存器功能表

中断控制位	中断向量	中断源	说明
B15	FFF6H	FIQ_Fosc/1024	Fosc/1024：系统时钟 1024 分频
B14		IRQ0_Fosc/1024	
B13		FIQ_ TMA	定时器 A 溢出信号
B12	FFF8H	IRQ1_TMA	
B11	FFF9H	FIQ_TMB	定时器 B 溢出信号
B10	FFFAH	IRQ2_TMB	
B9	FFFBH	IRQ3_EXT2	外部信号 2 中断
B8		IRQ3_EXT1	外部信号 2 中断
B7		IRQ3_KEY	键盘唤醒中断
B6	FFFCH	IRQ4_4 kHz	4096 Hz 时基信号中断
B5		IRQ4_2 kHz	2048 Hz 时基信号中断
B4		IRQ4_1 kHz	1024 Hz 时基信号中断
B3	FFFDH	IRQ5_4 Hz	4 Hz 时基信号中断
B2		IRQ5_2 Hz	2 Hz 时基信号中断
B1	FFFEH	IRQ6_TMB1	时基发生器 1 输出选频信号中断
B0		IRQ6_TMB2	时基发生器 2 输出选频信号中断
	FFF7H	RESET	复位信号
	FFFFH	UART IRQ	串行通信口中断
	FFF5H	BREAK	软中断

写入时，1：允许该位所代表的中断开放。此时当有该中断申请时，CPU 会响应。

0：禁止该位所代表的中断开放，即使有中断申请，CPU 也不会响应。

读出时，作为中断标志，其每一位代表一个中断源。

1：表示有中断请求。

0：表示无中断请求。

2) 清中断标志寄存器 P_INT_Clear (写)(7011H)

清中断标志寄存器用于清除中断控制标志位。当 CPU 响应中断后中断标志仍为 1，硬件并不自动清除该标志，当进入中断服务程序后，必须将其控制标志清零，否则，CPU 还会响应该中断。

P_INT_Clear 寄存器的每一位对应一个中断源(如表 5.2 所示，具体位置与中断控制寄存器相同)，如果想清除某个中断标志，只要将该寄存器中对应位置 1，即可清除该中断标志。该寄存器只有写的属性，读该寄存器是无任何意义的。

2. 中断控制端口配置

中断控制端口配置如表 5.3 所示。

表 5.3 中断控制端口配置表

P_INT_Ctrl (写)	P_INT_Ctrl (读)	P_INT_Clear (写)	功能
1	—	—	允许中断
0	—	—	屏蔽中断，但不清除 P_INT_Ctrl (读) 单元相应的中断标志位
—	1	—	有中断事件发生
—	0	—	没有中断事件发生
—	—	1	清除中断标志
—	—	0	不改变中断源的状态

3. 中断控制指令

中断控制指令的格式及说明如表 5.4 所示。中断控制指令有以下几种：

表 5.4 中断控制指令一览表

指令格式	说明
FIQ ON FIQ OFF	允许 FIQ 中断 禁止 FIQ 中断
IRQ ON IRQ OFF	允许 IRQ 中断 禁止 IRQ 中断
INT FIQ	允许 FIQ 中断，禁止 IRQ 中断
INT IRQ	允许 IRQ 中断，禁止 FIQ 中断
INT FIQ,IRQ	允许 FIQ 和 IRQ 中断
INT OFF	禁止 FIQ 和 IRQ 中断

注：指令长度：1 字；运行周期：3

(1) FIQ ON 指令。

功能：开放 FIQ 中断，是 FIQ 中断的总中断允许命令，其控制指令不能代替 P_INT_Ctrl。也就是说，即使在程序中写了该指令，但没有将 P_INT_Ctrl 寄存器的 FIQ 位置 1，CPU 也无法响应该中断。FIQ ON 与 FIQ OFF 配对使用。

例 1 FIQ ON 的使用。

```
_main:
        FIQ OFF
        r1 = 0x8000H
        [P_INT_Ctrl] = r1              //在 P_INT_Ctrl 开放 FIQ 中断
        FIQ ON
loop: NOP
        GOTO loop
```

(2) FIQ OFF 指令。

功能：屏蔽 FIQ 中断，该指令可以屏蔽 P_INT_Ctrl 控制寄存器打开的 FIQ 中断。

例 2 快速中断的屏蔽与开放。

```
_main:
        FIQ ON
```

```
        r1 = 0x8000H
        [P_INT_Ctrl] = r1          //开放 FIQ 中断
        FIQ OFF                    //关闭 FIQ 中断
        FIQ ON                     //开放 FIQ 中断
loop:   NOP
        GOTO loop
```

(3) IRQ ON　IRQ 总中断允许指令。

功能：开放 IRQ **中断。该控制指令不能代替** P_INT_Ctrl**，与** FIQ ON **相似，必须通过** P_INT_Ctrl **允许中断，与** IRQ OFF **配对使用。**

(4) IRQ OFF IRQ **总中断屏蔽指令。**

功能：屏蔽 IRQ **中断，与** FIQ　OFF **相似，可以屏蔽** P_INT_Ctrl **开放的中断，并通过** IRQ ON **开放。**

(5) INT **指令。**

功能：设置允许/禁止 FIQ **和** IRQ **中断。该控制指令与前面的指令相同，只有先通过** P_INT_Ctrl **寄存器打开中断，才能完全开放中断。**

INT **控制指令可以细分为：**

① INT FIQ　　**功能：允许** FIQ **中断，关闭** IRQ **中断。**

② INT IRQ　　**功能：允许** IRQ **中断，关闭** FIQ **中断。**

③ INT FIQ IRQ　**功能：允许** FIQ **中断，允许** IRQ **中断。**

④ INT OFF　　**功能：关闭** FIQ **中断，关闭** IRQ **中断。**

例 3　**中断命令的使用。**

```
_main:
        r1=0x8004H                 //开中断 IRQ5_2 Hz 和 FIQ_PWM
        [P_INT_Ctrl]=r1
        INT FIQ                    //允许 FIQ_PWM 中断，屏蔽 IRQ5 中断
        INT IRQ                    //允许 IRQ5 中断，屏蔽 FIQ_PWM
        INT FIQ，IRQ               //允许 IRQ 和 FIQ 中断
        INT OFF                    //屏蔽 IRQ 和 FIQ 中断
loop:   NOP
        GOTO loop
```

5.2.4　中断向量及优先级

SPCE061A **单片机每一中断都对应一个向量单元，称为中断向量。在中断向量单元中存放中断服务程序的入口地址，而不是转移指令(与** MCS−8051 **不同)，如表** 5.5 **所示。在允许中断的情况下，当有中断请求时，将使程序转移至由中断向量单元内容所决定的中断服务程序起始地址运行。**

SPCE061A **单片机快速中断的优先级高于普通中断的优先级。在** IRQ **中断中，**IRQ0 **的中断优先于** IRQ1**，**IRQ1 **的中断优先于** IRQ2**，**IRQ **的序号越大，中断优先级越低。串行通信口**(UART)**的中断优先级最低。**

表 5.5 SPCE061A 中断的优先级

中断类型	中断向量	中断查询顺序
RESET	FFF7 H 复位向量	0
FIQ	FFF6 H	1
IRQ0	FFF8 H	2
IRQ1	FFF9 H	3
IRQ2	FFFA H	4
IRQ3	FFFB H	5
IRQ4	FFFC H	6
IRQ5	FFFD H	7
IRQ6	FFFE H	8
UART、IRQ	FFFF H	9

在 IRQ 中断中，只是中断查询有先后，不能进行中断嵌套。同一中断向量内的中断源中断优先级相同。

中断优先关系如下：

FIQ>IRQ0>IRQ1>IRQ2>IRQ3>IRQ4>IRQ5>IRQ6>UART、IRQ 同级

5.2.5 中断响应

1. 中断响应过程

从中断请求发生到被响应，再到转向执行中断服务程序，完成中断所要求的操作，是一个复杂的过程。整个过程都是在 CPU 的控制下有序进行的。下面按顺序叙述中断响应过程。

1) 中断查询

把所有的中断请求都汇集到 P_INT_Ctrl 和 P_UART_Command2(该寄存器用于检测串行通信口中断标志位)寄存器中。外部中断时运用采样将中断请求信号锁存在 P_INT_Ctrl 寄存器的相应标志位，而音频输出中断、触键唤醒、定时中断、时基中断、串行通信口中断等中断请求，由于都发生在芯片的内部，可以直接去置位 P_INT_Ctrl 和 P_UART_Command2 中各自的中断请求标志位。所谓查询就是由 CPU 测试 P_INT_Ctrl 和 P_UART_Command2 中各标志位的状态，确定是否有中断请求发生以及是哪一个中断请求。

中断查询发生在每条指令周期结束，按中断优先级顺序对中断请求进行查询，即先查询 FIQ，再查询 IRQ。同级中断按 IRQ0、IRQ1、IRQ2、IRQ3、IRQ4、IRQ5、IRQ6、UART 的顺序查询。如果查询到有标志位为 1，则表明有中断请求发生。由于中断请求是随机发生的，无法预先知道，因此，程序执行过程中，在每条指令结束都要进行中断查询。

2) 中断响应

当允许中断且查询到中断请求时，才能响应中断。此时，CPU 首先将程序计数器 PC 的内容压入堆栈，再将 SR 压入堆栈以保护断点，然后将存放在中断向量单元的中断服务程序入口地址装入 PC，执行中断服务程序。

3) 中断响应条件

中断响应条件如下：

(1) 中断开放；

(2) 对应中断源允许中断；

(3) 无高优先级的中断请求；

(4) 一条指令执行结束。

2. 中断请求的撤销

中断响应后 P_INT_Ctrl 和 P_UART_Command2 中的中断请求标志应及时清除，否则就意味着中断请求仍然存在，会造成中断的重复响应。因此，必须及时撤销中断请求。SPCE061A 中断撤销只需向中断标志位写 1 即可。

除 UART 中断外，所有的中断均需软件清除标志位，即将 P_INT_Ctrl 中相应的中断标志位清除，就撤销了中断请求。而 UART 中断则是硬件自动清除，不需要软件操作。

如当接收到数据后 P_UART_Command2 中的接收标志位自动置 1，进入 UART 中断。在 UART 中断服务程序中读出数据，P_UART_Command2 相应的中断标志位自动清除。

3. 中断服务流程

SPCE061A 单片机的中断服务流程图如图 5.4 所示。

1) 中断入口

中断入口即中断的入口地址，每个中断源都有自己的入口地址，如表 5.6 所示。

表 5.6　中断入口地址表

中断类型	中断向量
RESET	FFF7H 复位向量
FIQ	FFF6H
IRQ0	FFF8H
IRQ1	FFF9H
IRQ2	FFFAH
IRQ3	FFFBH
IRQ4	FFFCH
IRQ5	FFFDH
IRQ6	FFFEH
UART IRQ	FFFFH
BREAK	FFF5H

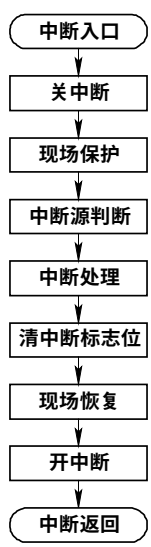


图 5.4　中断服务流程图

当 CPU 响应中断后，通过中断向量(存放中断服务程序第一条指令的地址)进入中断服务程序。

2) 关中断和开中断

在一个中断执行过程中，有可能有新的中断请求发生，对于重要的中断必须执行完成，不允许被其它的中断所嵌套，对此可以采用关闭中断的方法来解决。如在 IRQ 中断中不允许 FIQ 中断嵌套，可以在 IRQ 中断响应后关闭中断，当中断服务程序执行结束后，再开放

中断，响应 FIQ。也可在进入中断服务程序后，先关闭中断系统，彻底屏蔽其它中断请求，等中断处理完成后，再打开中断系统。还有一种情况是中断处理可以被打断，但现场的保护和恢复不允许打扰以免现场被破坏，为此应在保护现场和恢复现场的前后，进行关中断和开中断，这样除现场保护和现场恢复的片刻外，仍然保持着系统中断嵌套功能。

SPCE061A 单片机开中断和关中断通过中断控制指令来完成。如果只要对单个中断源进行控制，通过 P_INT_Ctrl 控制寄存器相应位置 1 和清 0 来打开或关闭某个中断源。

3) 现场保护和现场恢复

所谓现场是指中断时刻单片机存储单元的数据状态。为了使中断服务的执行不破坏这些数据或状态，以免在中断返回后影响主程序的运行，通常要把它们送入堆栈中保存起来，这就是现场保护。

现场保护一定要位于中断服务程序的前面，中断服务结束后在返回主程序前把保存的现场内容从堆栈中弹出，以恢复那些存储单元的原有内容，这就是现场恢复。现场恢复一定要位于中断处理程序之后返回主程序前。

4) 中断源判断

由于 SPCE061A 中断源多于中断向量，多个中断源共用一个中断向量，所以当 CPU 响应中断进入中断服务程序后，要通过读 P_INT_Ctrl 判断产生中断请求的中断源。

5) 中断处理

中断处理是中断服务程序的核心内容，是中断的具体目的，即中断后要完成的操作。

6) 清中断标志位

CPU 是根据中断标志位来判断并进行响应中断的，除串行通信口中断外所有的中断标志位不是硬件清除，而是由软件清除的，所以在中断服务程序中必须将中断标志位清除。

中断标志位的清除只要在中断服务程序中即可，位置不是固定的。一般在中断服务程序返回前清除中断标志。

7) 中断返回

中断服务程序最后一条指令必须是中断返回指令 RETI，当 CPU 执行这条指令时，从堆栈中弹出断点 PC 及 SR，即可恢复断点，重新执行被中断的程序。

5.3 中断系统应用

5.3.1 单中断源应用

1. 定时中断

定时器中断包括定时器 A 中断和定时器 B 中断，而且不仅 FIQ 中断方式中有定时器中断源，在 IRQ1(TimerA)、IRQ2(TimerB)中也有定时器中断源，可根据具体程序需要选择。如果需要定时器中断优先级高，可以打开 FIQ 方式的定时器中断；如果不要求较高的中断优先级，可以将定时器中断放在 IRQ 中断方式中。

例 4 利用定时器 A，在 A 口的 IOA0 脚输出周期为 20 ms 的方波(定时 10 ms)。

分析：

(1) 选择使用哪种中断方式，是 FIQ 中断方式还是 IRQ 中断方式。当然在这个例子中

采用哪一种中断都可以，这里我们采用 IRQ 中断，开中断时需打开 IRQ1_TMA 中断即定时器 A 中断，将 P_INT_Ctrl 的 IRQ1_TMA 置位即可。

(2) 产生 T=20 ms 的方波，只要每 10 ms 使 IOA0 脚输出电平翻转一次，因此只需定时 10 ms 即可。

(3) 选择时钟源，以方便准确为原则，我们选用 8 kHz 时钟源 A，时钟源 B 为 1。

(4) 确定定时器 A 的预置初数(P_imerA_Data)。

计算方法：

定时器溢出率＝1/10 ms=100 Hz

定时器初值 N=0xFFFF−时钟源频率/定时器溢出频率

=0xFFFF−8192/100 =0xFFAD

定时中断使用的寄存器如表 5.7 所示。

表 5.7　定时中断使用的寄存器

配置单元	属性	地址	配置单元功能说明
P_TimerA_Data	读/写	700A H	TimerA 预置初值存储单元
P_TimerA_Ctrl	读/写	700B H	TimerA 控制单元，时钟源 ClkA、ClkB 输入选择和脉宽调制占空比输出 APWMO 的控制
P_TimerB_Data	读/写	700C H	TimerB 预置初值存储单元
P_TimerB_Ctrl	读/写	700D H	TimerB 控制单元，时钟源 ClkA 输入选择和脉宽调制占空比输出 BPWMO 的控制
P_INT_Ctrl	读/写	7010 H	写：控制各中断源允许或禁止中断 读：判断产生中断请求的中断源
P_INT_Clear	写	7011 H	清除中断请求

主程序：完成 IOA0 端口、定时器、时基信号及中断初始化，主程序流程图如图 5.5 所示，中断服务程序流程图如图 5.6 所示。

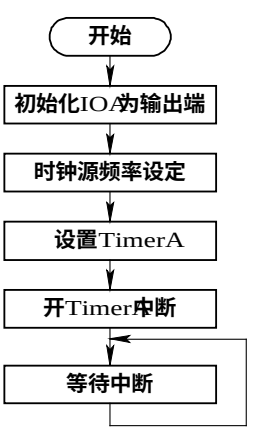


图 5.5　定时器中断主程序流程图

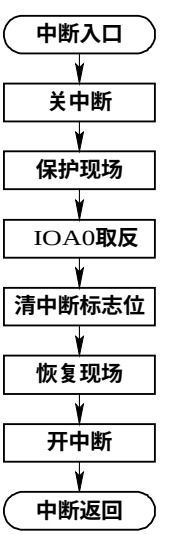


图 5.6　定时器中断服务程序流程图

程序名称： TimerA _10ms _Pulse .asm。

```
.include hardware.inc
.define TIMERA_CLKA_8 K 0x0003          //时钟源 A 选择 8192Hz
.define   TIMERA_CLKB_1 0x0030H         //时钟源 B 选择 1
.define TIMER_DATA_FOR_8 kHz 0xFFAD
                                        //计算: 0xFFFFH-时钟源频率/定时器溢出率
                                              =0xFFFF-8192/100
                                              =OxFFAD
    .RAM
    .CODE
    .PUBLIC_main
_main:
    r1 = 0x0001                         //定义 IOA0 为输出
    [P_IOA_Dir] = r1
    [P_IOA_Attrib] = r1
    [P_IOA_Data] = r1
    r1 = TIMER_DATA_FOR_8 kHz           //定时器 A 置初值 0xFFAD
    [P_TimerA_Data]=r1                  //定义时钟源 A 为 8192 Hz，时钟源 B 为 1
    r1 = TIMERA_CLKA_8K + TIMERA_CLKB_1
    [P_TimerA_Ctrl]=r1
    r1 = Ox1000                         //开放定时 A 的中断
    [P_INT_Ctrl]=r1
    INT IRQ                             //开 IRQ 中断
loop:  NOP                              //等待中断
    GOTO loop
```

定时器 A 中断服务程序：每 10 ms 中断一次，将 IOA0 端口状态取反。

```
.TEXT
.PUBLIC _IRQ1
_IRQ1:  INT OFF                         //关中断
    PUSH r1,r5 TO [sp]                  //保护现场
    r1 = [P_IOA_DATA]
    r1 ^= 0xFFFFH                       //方波标识位取反，输出波形
    [P_IOA_Data]=r1
    r1=0x1000                           //清除中断标志位
    [P_INT_Clear]=r1
    POP r1,r5 FROM[sp]                  //恢复现场
    INT IRQ FIQ                         //开中断
RETI                                    //中断返回
```

2. **时基中断**

SPCE061A **单片机时基信号频率资源丰富，有** 2 Hz、4 Hz、8 Hz、16 Hz、32 Hz、64 Hz、128 Hz、256 Hz、512 Hz、1024 H、2048 Hz、4096 Hz **等多种频率供实时处理时钟选择。提供了七个时基中断源，占用了三个中断向量，可根据实际需要选择适当的时基信号源。时基中断使用的控制寄存器如表** 5.8 **所示。**

表 5.8　**时基中断使用的控制寄存器**

配置单元	属性	地址	配置单元功能说明
P_TimeBase_Setup	写	700EH	时基信号发生器输出的 TMB1、TMB2 频率设定
P_TimeBase_Clear	写	700FH	时基信号发生器复位并进行时间的精确校准
P_SystemClock	写	7013H	系统时钟的选择控制单元
P_INT_Ctrl	读/写	7010H	写：控制各中断源允许或禁止 读：判断产生中断请求的中断源
P_INT_Clear	写	7011H	用来清除中断请求标志位

例 5　**在** A **口低** 8 **位接八个发光二极管，利用时基信号设计一定时器，定时** 0.5 s **闪烁一次。**

分析：首先考虑定时 0.5 s **采用哪个时基信号比较方便，我们可以很明显地看出** 2 Hz **时基信号中断是最方便的。只要设定** 2 Hz **的时基信号中断为** 0.5 s **的定时时间即可。**

硬件：IOA0～IOA7 **接八个发光二极管。**

主程序：端口初始化，2 Hz **中断开放。**

程序名称：TimeBase_2Hz.asm。

时基中断主程序流程图如图 5.7 **所示，中断服务程序流程图如图** 5.8 **所示。**

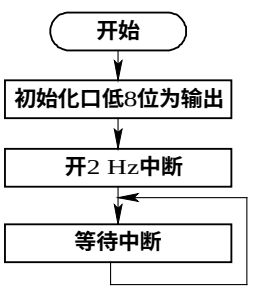


图 5.7　**时基中断主程序流程图**

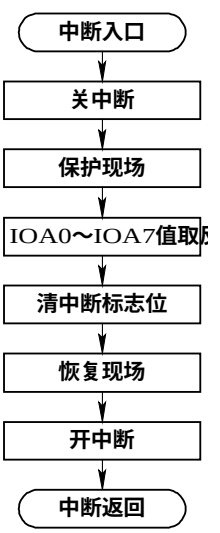


图 5.8　**时基中断中断服务程序流程图**

主程序：

```
        .include hardware.inc
        .CODE
        .PUBLIC _main
_main:
        r1=0x00FF                       //初始化 A 口低 8 位为输出
        [P_IOA_Attrib]=r1
        [P_IOA_Dir]=r1
        r1=0x0000
        [P_IOA_Data]=r1
        r1 =0x0004                      //开放 IRQ5_2 Hz 中断
        [P_INT_Ctrl]=r1
        INT IRQ                         //开 IRQ 中断
loop: NOP
        GOTO loop
```

中断服务程序：每中断一次，使 IOA0～IOA7 **状态取反，所接** LED **状态变化。**

```
.TEXT
.PUBLIC _IRQ5
_IRQ5:      PUSH r1,r5 TO [sp]          //保护现场
            r1 = 0x0004
            TEST r1,[P_INT_Ctrl]        //判 2 Hz 中断源
            JNZ IRQ5_2Hz                //是，2 Hz 中断，则转至对应程序段
IRQ5_4Hz:   r1=0x0008                   //否则是 4 Hz 中断
            GOTO Exit_INT
IRQ5_2Hz:   r1 = [P_IOA_DATA]           //2 Hz 中断应用程序
            r1 ^= 0xFFFF                //IOA 口值取反
            [P_IOA_Data] = r1
            r1 = 0x0004
Exit_INT:   [P_INT_Clear]=r1            //清中断标志
            POP r1,r5 FROM [sp]         //恢复现场
            RETI                        //中断返回
```

3. **触键唤醒中断**

触键唤醒中断是系统进入睡眠状态后，通过 A **口低** 8 **位的按键来唤醒系统时钟，恢复睡眠时的** PC **指针的，同时进入触键唤醒中断。触键唤醒中断使用的控制寄存器如表** 5.9 **所示。**

例 6　**使** CPU **进入睡眠状态，通过触键唤醒系统。**IOA0～IOA7 **接八个发光二极管并利用唤醒中断点亮** LED**。**

分析：首先考虑如何使系统进入睡眠状态，前面已经介绍了，通过设置时钟系统控制寄存器 P_SystemClock 的 b4 位使系统进入睡眠状态。其次设置唤醒按键，因为只有 A 口的低 8 位具有唤醒功能，所以按键必须在 A 口低 8 位中选择。本例选用 A 口低 8 位接键盘。

表 5.9　触键唤醒中断使用的控制寄存器

配置单元	属性	地址	配置单元功能说明
P_IOA_Latch	读	7004H	从其读数可激活 A 口的唤醒功能并锁存 IOA0～IOA7 上的键状态
P_SystemClock	写	7013H	系统时钟的选择控制单元
P_INT_Ctrl	读/写	7010H	写: 控制各中断源允许或禁止中断 读: 判断产生中断请求的中断源
P_INT_Clear	写	7011H	用来清除中断源的中断请求

硬件：A 口低 8 位接键盘；高 8 位接八个 LED。
主程序：IOA 口、触键唤醒/中断初始化，系统进入睡眠状态。
触键唤醒中断主程序流程图如图 5.9 所示，中断服务程序流程图如图 5.10 所示。

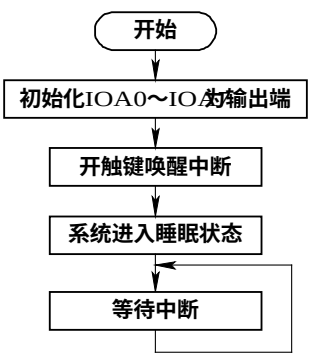


图 5.9　触键唤醒主程序流程图

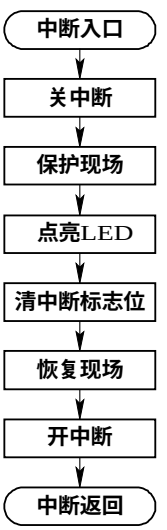


图 5.10　触键唤醒中断服务程序流程图

主程序：

```
    .include hardware.inc
    .PUBLIC _main
    .define P_IOA_latch 0x7004
    .CODE
_main:
```

```
        r1=0xFF00                    //初始化 A 口低 8 位为带下拉输入，高 8 位为输出
        [P_IOA_Dir]=r1
        [P_IOA_Attrib]=r1
        r1 =0x0000
        [P_IOA_Data]=r1
        r1 =0x0080                   //开放 IRQ3_KEY 触键唤醒中断
        [P_INT_Ctrl] = r1
        r1=[P_IOA_latch]             //激活 A 口触键唤醒
        INT IRQ                      //开 IRQ 中断
        r1=0x0007
        [P_SystemClock]=r1           //系统进入睡眠状态
Loop:   NOP                          //当有键唤醒时继续执行
        GOTO Loop
```

中断服务程序：判断中断源类型，是触键唤醒中断点亮八个发光二极管，否则返回。

```
.TEXT
.PUBLIC _IRQ3
_IRQ3:
            INT OFF
            PUSH r1,r4 TO [sp]          //保护现场
            r1 = 0x0080
            test r1,[P_INT_Ctrl]        //是否为触键唤醒中断
            jz L_notKeyArouse           //否，外部中断
L_KeyArouse: r1 = [P_IOA_Data]          //是，触键唤醒中断
            r1 |= 0xFF00                //点亮八个发光二极管
            [P_IOA_Data] = r1
            r1 =0x0080
            GOTO Exit_INT
L_notKeyArouse:                         //外部中断
            r1 =  0x0100                //判断是 EXT1 中断吗
            TEST r1,[P_INT_Ctrl]
            JNZ L_EXT1
            r1 = 0x0200                 //外部中断 2
L_EXT1:     NOP
Exit_INT:   [P_INT_Clear]=r1            //清中断标志
            POP r1,r4 FROM [sp]         //恢复现场
            INT IRQ
            RETI
        .END
```

4. **外部中断**

SPCE061A **单片机有两个外部中断源** EXT1 **和** EXT2，**这两个外部中断源可以使用反馈电路定时触发，也可以通过** IOB2 **或** IOB3 **引脚的下降沿触发。外部中断使用的控制寄存器如表** 5.10 **所示。**

表 5.10　**反馈电路外部中断使用的控制寄存器**

配置单元	属性	地址	配置单元功能说明
P_INT_Ctrl	读/写	7010H	写: 可控制各中断源允许或禁止中断 读: 可判断产生中断请求的中断源
P_INT_Clear	写	7011H	清除中断源的中断请求
P_FeedBack	写	7009H	B 口的应用方式控制向量

例 7　**利用外部中断点亮接在** A **口低** 8 **位的八个** LED。

分析：选择使用外部中断 EXT1 **还是外部中断** EXT2。**两个外部中断都可以，只是初始化时略有不同。选择** EXT1 **初始化** IOB2 **为带上拉电阻的输入端口，选择** EXT2 **初始化** IOB3 **为带上拉电阻的输入端口。此例我们选择** EXT1 **中断源。**

硬件：A **口低** 8 **位接八个** LED。

主程序：端口初始化，开放 EXT1 **中断。**

程序名称：Ext1.asm。

外部中断的主程序流程图如图 5.11 **所示，中断服务程序流程图如图** 5.12 **所示。**

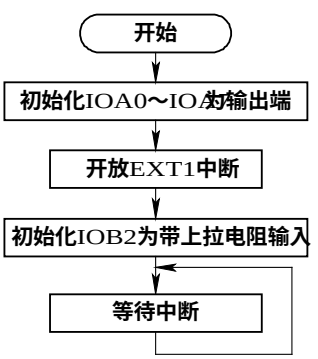


图 5.11　**外部中断主程序流程图**

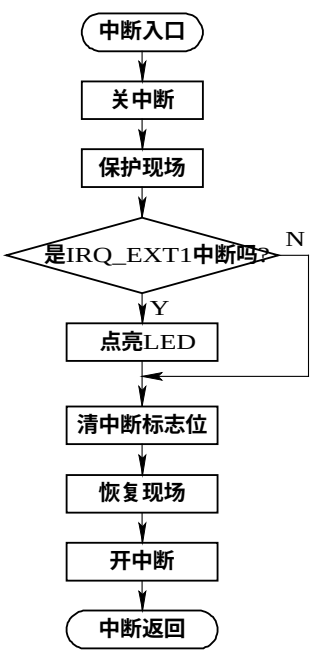


图 5.12　**外部中断中断服务程序流程图**

主程序：

```
.include hardware.inc
.CODE
.PUBLIC _main
_main:
        r1=0x00FF                       //设置 A 口低 8 位为输出
        [P_IOA_Attrib]=r1
        [P_IOA_Dir]=r1
        r1 = 0x0000
        [P_IOA_Data]=r1
        r1=0x0000                       //设 IOB2 带上拉电阻输入
        [P_IOB_Dir]=r1
        [P_IOB_Attrib]=r1
        r1=0x0004
        [P_IOB_Data]=r1
        r1=0x0100                       //开中断 IRQ3_EXT1
        [P_INT_Ctrl]=r1
        INT IRQ
loop: NOP
        JMP loop
```

中断服务程序：判断中断源、点亮 LED。

```
.TEXT
.PUBLIC _IRQ3
_IRQ3:
        INT OFF
        PUSH r1,r5 TO [sp]              //现场保护
        r1=0x0100
        TEST r1,[P_INT_Ctrl]            //判断 IRQ3_EXT1 中断
        JNZ irq3_ext1                   //是，则转至 IRQ3_EXT1 程序段
        r1=0x0200
        TEST r1,[P_INT_Ctrl]            //判断 IRQ3_EXT2
        JNZ irq3_ext2
irq3_key:
        r1 = 0x0080                     //键唤醒中断
        GOTO Exit_INT
irq3_ext2:                              //外部中断 2
        GOTO Exit_INT
irq3_ext1:                              //外部中断 1
```

```
        r2=0x00FF
        [P_IOA_Data]=r2                    //点亮 LED
Exit_INT:
        [P_INT_Clear]=r1
        POP r1,r5 FROM sp]                 //恢复现场
        INT IRQ,FIQ
        RETI
```

5. 串行通信口中断

串行通信口中断用于串行通信过程中数据的接收和发送。UART 可以实现全双工通信，P_UART_Data (读/写) ($7023H)单元用于接收和发送数据的缓存。串行通信口中断使用的控制寄存器如表 5.11 所示。

表 5.11　异步串行通信口中断使用的控制寄存器

配置单元	属性	地址	配置单元功能说明
P_UART_Command1	写	7021 H	UART 功能的控制单元
P_UART_Command2	读/写	7022 H	读: UART 工作的状态口 写: 控制 UART RX、TX 端口数据传输的允许和禁止
P_UART_Data	读/写	7023 H	接收或发送的数据
P_UART_BaudScalar Low	读/写	7024 H	传输波特率控制字的低位字
P_UART_BaudScalar High	读/写	7025 H	传输波特率控制字的高位字
P_INT_Ctrl	读/写	7010 H	写：各中断源允许或禁止中断 读：判断中断请求的中断源
P_INT_Clear	写	7011 H	用来清除中断请求

写入：将发送的数据送入缓存器，按字节传送。

读出：从缓存器读出接收的单字节数据。UART 模块的接收管脚 Rx 和发送管脚 Tx 分别可与 IOB7 和 IOB10 共用。

注意：由于 UART 串行传输方式为 8 位数据传输，所以传输一个字需要发送两次，接收也是如此。

例 8　准备一组数据自发自收。系统时钟频率 Fosc=24.576 MHz，串行通信口波特率为 115.2 kb/s。

分析：允许串行通信口中断并设置 P_UART_Command1，允许 UART 接收、发送中断。串行通信口波特率控制字为 0000006BH。

异步串行通信主程序：Rx、Tx 端口初始化、串行通信口工作方式、波特率设定、接收、发送指针设定。自发自收五个数据：0x1122, 0x3344，0x5566，0x7788 和 0xAAFF。

异步串行通信主程序流程图如图 5.13 所示，中断服务程序流程图如图 5.14 所示。

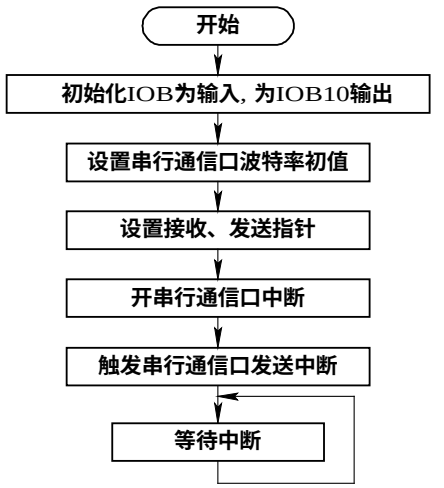


图 5.13　异步串行通信主程序流程图

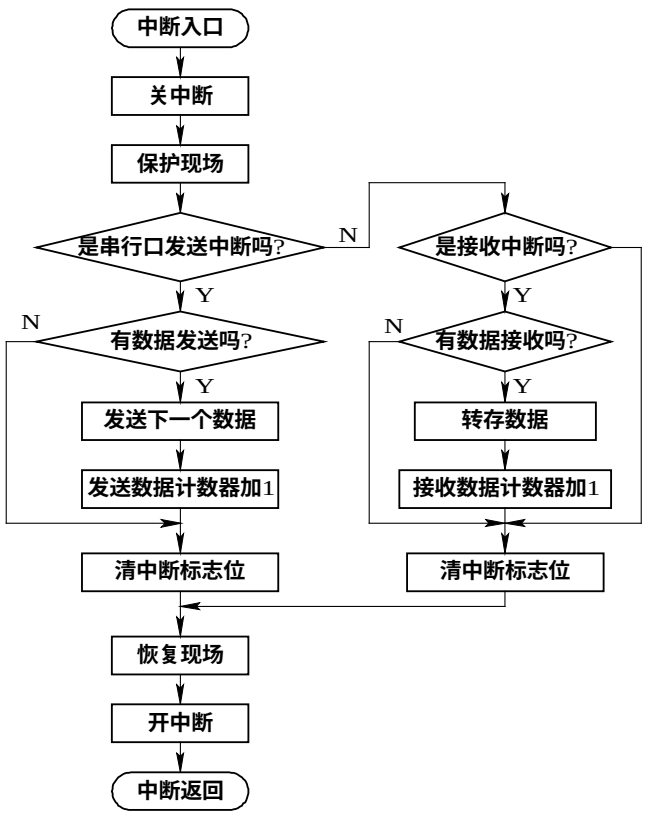


图 5.14　异步串行通信中断服务程序流程图

主程序：

```
.include hardware.inc
.define UART_DATA_SIZE 0x0005H
.DATA
SendData: .DW 0x1122, 0x3344，0x5566，0x7788, 0xAAFF  //发送的数
```

```
.ISRAM
.PUBLIC RecData                          //接收数据
RecData: .DW 5 DUPdup(0)
 .var recNum                             //接收计数
 .var sendNum                            //发送计数
 .var sendFlag                           //发送标志　1:发送高 8 位, 0:发送低 8 位
 .var recFlag                            //接收标志　1:接收高 8 位, 0:接收低 8 位
 .CODE
 .PUBLIC_main
    _main:                               //主程序
            r2 = RecData
            R1 = 0x0480                  //设置 IOB7 为悬浮输入, IOB10 为同相输出
            [P_IOB_Attrib] = R1
            R1 =0x0400
            [P_IOB_Dir] = R1
            R1 = 0x0000
            [P_IOB_Data] = R1
            R1 = 0x006B                  //设置波特率 114.84 kHz(~=115.2 kHz)
            [P_UART_BaudScalar Low] = R1
            R1 = 0x0000
            [P_UART_BaudScalar High] = R1
            R1 = 0x00C0                  //R1=0000 0000 1100 0000B
            [P_UART_Command1] = R1       //开接收、发送中断
            [P_UART_Command2] = R1       //允许接收、发送
            R1 =0x0000
            [sendNum] = r1               //发送指针清 0
            [recNum] = R1                //接收指针清 0
            R1 = 0x0001
            [sendFlag] = R1              //初始化发送标志位
            [recFlag] = r1               //初始化接收标志位
            INT IRQ                      //开中断
  loop:     NOP
            GOTO loop
```

串行通信口中断服务程序：接收、发送一组数据。

```
            .TEXT
            .PUBLIC _IRQ7
            UART_RECC_IRQ:.PROC
_IRQ7:
                  INT OFF
```

```
                PUSH r1,r5 TO [sp]
                r1 = 0x0080                    //判断接收中断标志 RxRDY
                TEST R1,[P_UART_Command2]      //判是否为接收中断
                JNZ UART_RECV_IRQ
UART_SEND_IRQ:                                 //发送数据
                r2 = [sendFlag]
                r2 ^= 0x0001
                [sendFlag] = r2
                r1 = SendData
                r4 = [sendNum]
                r3 = UART_DATA_SIZE
                CMP r4,r3                      /比较发送是否结束
                JE exit_INT                    //结束发送
                r1 = r1+r4                     //继续发送
                r1 = [r1]                      //取发送数据
                r2 = [sendFlag]
                JZ Send_Data
                r1 = r1 lsr 4                  /发送高 8 位
                r1 = r1 lsr 4
                r4 += 1                        //发送计数加 1
                [sendNum] = r4
Send_Data:
                [P_UART_Data] = r1             //发送数据
                GOTO exit_INT
```

接收中断服务程序：

```
UART_RECV_IRQ:
                r2 = [recFlag]
                r2 ^= 0x0001                   //接收标识取反
                [recFlag] = r2
                r4 = [recNum]
                r3 = UART_DATA_SIZE
                CMP r3,r4
                JE exit_INT                    //判接收结束
                r1 = [P_UART_Data]             /接收没结束，继续接收
                r2 = [recFlag]
                JNZ Shift_Data
                r3 = r4 + RecData              //保存高 8 位
                [r3] = r1
            GOTO exit_INT
```

```
Shift_Data:                                    //接收高 8 位数
        r1 = r1 lsl 4
        r1 = r1 lsl 4
        r2 = [recNum]
        r3 = r2 + RecData                      //保存高 8 位
        r4 = [r3]
        r4 |= r1
        [r3] = r4
        r2 += 1                                //接收计数加 1
        [recNum] = r2

    exit_INT:
        POP r1,r5 FROM [sp]
        INT IRQ,FIQ
    RETI
.ENDP
```

5.3.2　多中断源应用

在单片机应用系统中，通常使用多个中断源。SPCE061A **单片机系统通常以两种方式使用多中断源，一种是使用同一中断向量的多中断源，另一种是使用不同中断向量的多个中断源。下面分别举例说明。**

1. **同中断向量多中断源的应用**

例 9　IRQ6 **中断有两个中断源** IRQ6_TMB1 **和** IRQ6_TMB2**，此处利用两个中断源控制四个发光二极管分别为** 1 s **和** 0.5 s **周期闪烁。**

分析：IRQ6_TMB1 **时钟源有多种选择，如** 8 Hz、16 Hz、32 Hz、64 Hz **等，可以选择其中任一频率做** 0.5 s **定时，此处我们选择** 64 Hz **时钟源。同样，**IRQ6_TMB2 **有** 128 Hz、256 Hz、512 Hz、1024 Hz **时钟源可供选择，此处选择** 128 Hz **时钟源。**

硬件：IOA0～IOA7 **接八个** LED**。**

主程序：利用 IRQ6_TMB1 **使** IOA0～IOA3 **接的** LED 1 s **闪烁** 1 **次。**IRQ6_TMB2 **使** IOA4～IOA7 **接的** LED 0.5 s **闪烁** 1 **次。**

多中断源主程序流程图如图 5.15 **所示，中断服务程序流程图如图** 5.16 **所示。**

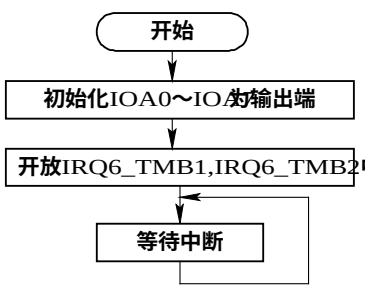


图 5.15　**同中断向量的多中断源主程序流程图**

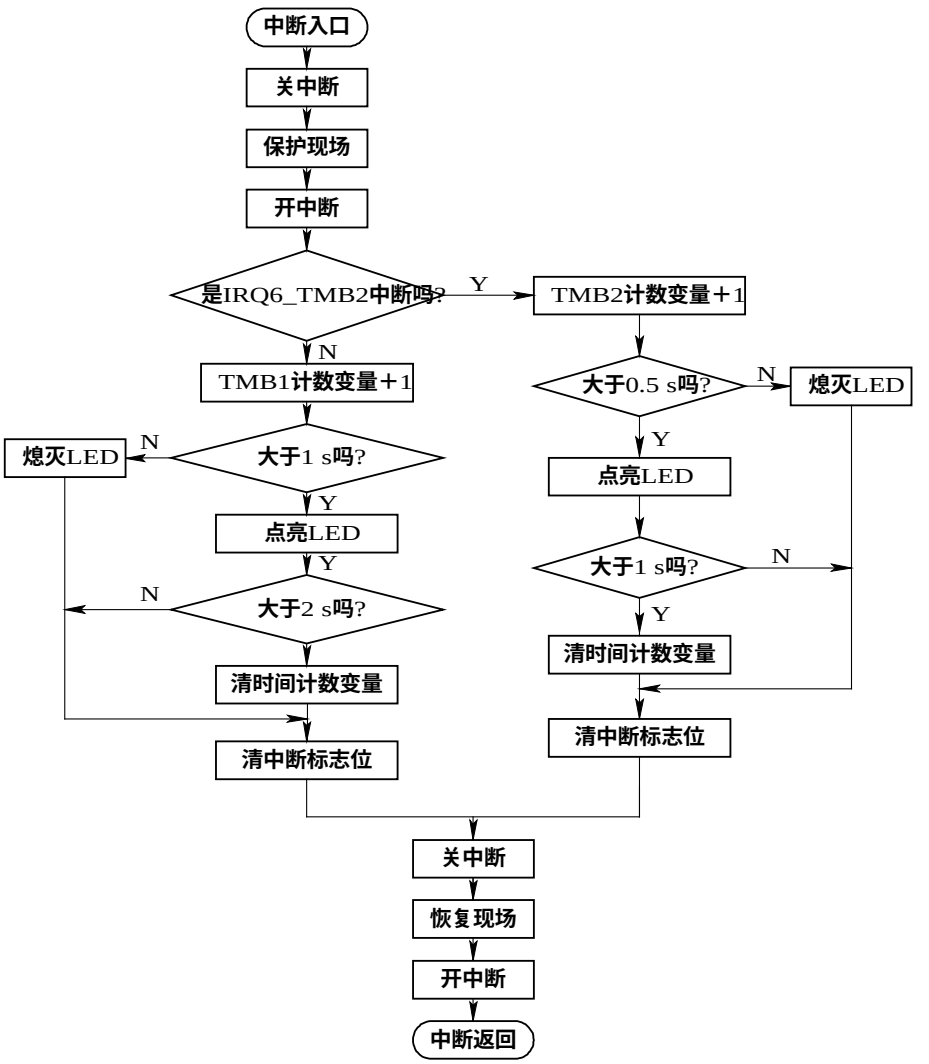


图 5.16　同中断向量的多中断源中断服务程序流程图

主程序：

```
.define tmb1clk_8Hz 0x0000
.define tmb1clk_16Hz 0x0001
.define tmb1clk_32Hz 0x0002
.define tmb1clk_64Hz 0x0003
.define tmb2clk_128Hz 0x0000
.define tmb2clk_256Hz 0x0004
.define tmb2clk_512Hz 0x0008
.define tmb2clk_1024Hz 0x000C
.define tmb1_clk tmb1clk_64Hz
.define tmb2_clk tmb2clk_128Hz
```

```
.define P_IOA_DATA 0x7000
.define P_IOA_DIR 0x7002
.define P_IOA_ATTRI 0x7003
.define P_IOB_DATA 0x7005
.define P_IOB_DIR 0x7007
.define P_IOB_ATTRI 0x7008
.define P_INT_CTRL 0x7010
.define P_INT_CLEAR 0x7011
.define P_Timebase_setup 0x700E
.define P_Timebase_clear 0x700F
.RAM
.VAR TIME1
.VAR TIME2
.CODE
.PUBLIC _main
    _main:
        INT OFF
        r1=0xFFFF                       //IOA 口为输出口
        [P_IOA_ATTRI]=r1
        [P_IOA_DIR]=r1
        r1=0x0000
        [P_IOA_DATA]=r1

        r1=0xFFFF；                     //IOB 口为输出口
        [P_IOB_ATTRI]=r1
        [P_IOB_DIR]=r1
        r1=0x0000
        [P_IOB_DATA]=r1
        r1=tmb1_clk                     //TMB1 选 64 Hz，TMB2 选 128 Hz
        r1|=tmb2_clk                    //0000H 或 0003H，此时 R1 的值为 0003H
        [P_Timebase_setup]=r1
        r1=0x0003                       //允许 IRQ6_TMB1 和 IRQ6_TMB2 中断
        [P_INT_CTRL]=r1
        R1=0x0000
        [TIME1]=r1
        [TIME2]=r1
    INT IRQ
loop:   NOP                             //等待中断
        NOP
        JMP loop
```

IRQ6 中断服务程序：

```
.TEXT
.PUBLIC _IRQ6
    _IRQ6:
        PUSH r1,r5 TO [sp]              //保护现场
        r1=0x0001
        TEST r1,[P_INT_CTRL]            //判断 IRQ6_TMB2 的中断
        JNZ l_irq6_tmb2                 //转至 IRQ6_TMB2 中断服务程序
    l_irq6_tmb1:                        //否，则进入 IRQ6_TMB1 中断服务程序
        r2 = [TIME1]
        r2 += 0x0001
        [TIME1] = R2
        CMP r2,64                       //比较 1 s
        JBE LED1_OFF                    //小于等于则 LED 灭
        r1=[P_IOA_DATA]
        r1 | =0x000F                    //大于则 LED 亮
        [P_IOA_DATA]=r1
        CMP r2,128                      //比较
        JBE LED1_RET                    //小于等于则 LED 继续亮
        r2=0x0000                       //否则 TIME1 单元清零，返回中断
        [TIME1]=R2
        JMP LED1_RET
LED1_OFF:
    r1=[P_IOA_DATA]
        r1&=0x00F0
        [P_IOA_DATA]=r1
LED1_RET:                               //清除 IRQ6_TMB1 中断标志位
        r1=0x0002
        [P_INT_CLEAR]=r1
        POP r1,r5 FROM [sp]
     RETI

l_irq6_tmb2:
        r2=[TIME2]
        r2+=0x0001
        [TIME2]=R2
        CMP r2,64                       //比较 0.5 s
        JBE LED2_OFF                    //小于等于则 LED 灭
        r1=[P_IOA_DATA]
        r1 | =0x00F0                        //大于则 LED 亮
```

```
        [P_IOA_DATA]=r1
        CMP r2,128                          //比较 1 s
        JBE LED2_RET                        //小于等于则 LED 继续亮
        r2=0x0000                           //否则 TIME2 单元清零
        [TIME2]=R2
        JMP LED2_RET

LED2_OFF:
    r1=[P_IOA_DATA]
        r1&=0x000F
        [P_IOA_DATA]=r1

LED2_RET:
        r1=0x0001                           //清除 IRQ6_TMB1 中断标志位
        [P_INT_CLEAR]=r1
       POP r1,r5 FROM [sp]
    RETI
```

2. **不同中断入口的多中断源应用**

例 10　采用不同中断入口的中断源控制八个发光二极管分别为 1 s 和 0.5 s 闪烁。

分析：使用 IRQ2 中的定时器 B 定时 0.5 s，利用 IRQ4 中的 IRQ_1 kHz 中断定时 1 s。

硬件：IOA0～IOA7 接八个 LED，IOB0～IOB7 接八个 LED。

主程序：利用定时 B 定时 0.5 s 使 B 口的 LED 闪烁，利用时基信号 1024 Hz 产生中断 1 s，使 A 口 LED 闪烁。

不同中断入口的多中断源的主程序流程图如图 5.17 所示，中断服务程序流程图如图 5.18 和图 5.19 所示。

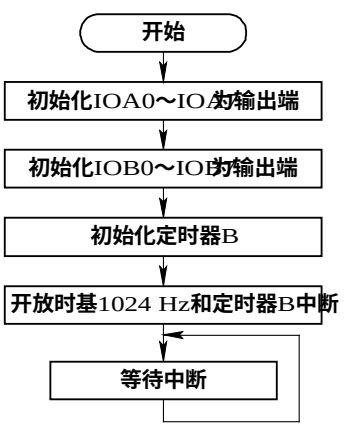


图 5.17　主程序流程图

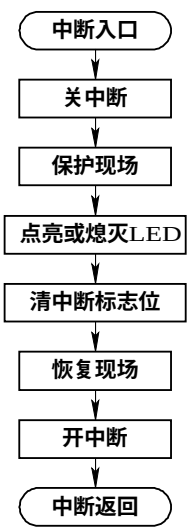


图 5.18　定时器 B 中断服务程序流程图

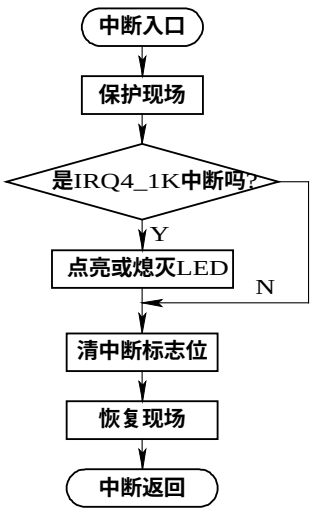


图 5.19　**时基信号** 1024 Hz **中断服务程序流程图**

主程序：

```
.include hardware.inc
.define RUN_TIMERB 0x0400                          //定义启动
.define TIMER_DATA_FOR_4 kHz (0xFFFF−2048)          //定时 0.5 s
.define RUN_TIMEBASE_1024 0x0010                   //时钟 1024 Hz 中断
.define TIMER_CLKA_4096 0x0004                     //时钟 A 选择 4096 Hz
.RAM                                               //IOA_LED 为定时器 B 数据
.VAR clock_cnt                                     //时钟计数指针
.CODE
.PUBLIC _main
_main:
    INT OFF
    r1=0xFFFF                                      //A 口为输出口
    [P_IOA_Attrib]=r1
    [P_IOA_Dir]=r1
    r1=0x00FF
    [P_IOA_Data]=r1
    r1=0xffff                                      //B 口为输出口
    [P_IOB_Attrib]=r1
    [P_IOB_Dir]=r1
    r1=0x00FF
    [P_IOB_Data]=r1
    r1 = TIMER_DATA_FOR_4 kHz                      //设置定时器 B
    [P_TimerB_Data]=r1
```

```
    r1 = TIMER_CLKA_4096                    //定义时钟源 A 为 4096 Hz
    [P_TimerB_Ctrl]=r1
    r1 = RUN_TIMEBASE_1024 + RUN_TIMERB     //开放定时器 B、1024 Hz 中断
    [P_INT_Ctrl]=r1
    INT IRQ                                 //开 IRQ 中断
loop: NOP                                   //等待中断
    GOTO loop
```

定时器 B 中断服务程序：使 B 口的八个 LED 0.5 s 闪烁一次。

```
.TEXT
.PUBLIC _IRQ2
_IRQ2:
        PUSH r1,r5 TO [sp]                  //保护现场
        r1=[P_IOB_Data]                     //LED  赋值
        r1 ^= 0xFFFF
        [P_IOB_Data]=r1
        r1=0x0400                           //清中断标志位
        [P_INT_Clear]=r1;
        POP r1,r5 FROM [sp];
        RETI
```

1024 Hz 时基信号中断服务程序：使 IOA0～IOA7 接的八个 LED 1 s 闪烁一次。

```
.TEXT
.PUBLIC _IRQ4
_IRQ4:  PUSH r1,r5 TO [sp]                  //压栈保护
        r1=0x0010
        TEST r1,[P_INT_Ctrl]                //测试 1 kHz 的中断
        JNZ l_irq4_1k                       //是，则转至对应程序
        r1=0x0020
        TEST r1,[P_INT_Ctrl]                //测试 2 kHz 的中断
        JNZ l_irq4_2k                       //是，则转至对应程序
l_irq4_4k:                                  //否，则进入 4 kHz 程序
        r1 = 0x0040
        GOTO exit_int
l_irq4_2k:
        GOTO exit_int
l_irq4_1k:
        r1 = [clock_cnt]
        CMP r1,1024
        JE LED_pro
        r1 += 1
```

```
                [clock_cnt] = r1
                r1 = 0x0010
                GOTO exit_int
LED_pro:                                        //P_IOA_DATA LED 取反
                r1 =[P_IOA_DATA]
                r1 ^= 0xFFFF
                [P_IOA_DATA]=r1
                r1 = 0x0000
                [clock_cnt] = r1                //清计数单元
                r1 = 0x0010
exit_int:
                [P_INT_Clear] = r1
                POP r1,r5 FROM [sp]
                RETI
```

思 考 题

1. SPCE061A 单片机有哪些中断源？
2. SPCE061A 单片机有哪几种类型的中断？请指出对应的中断向量。
3. 请简述 P_INT_Ctrl 单元的作用。
4. SPCE061A 单片机哪些中断源在系统响应后要用软件清除中断标志位？哪些中断源在系统响应后不需用软件清除中断标志位？
5. 请使用定时器 A 采用中断方式，在 IOA8 输出一周期为 10 ms 的方波。
6. 请使用定时器 B 采用中断方式，产生 1 s 时钟信号，并在 IOA0 脚输出电压信号。
7. 请编程，使系统在 30 s 无人操作时，进入睡眠状态，当有触键唤醒时，进入工作状态，并在 IOA1 脚输出周期为 2 s 的方波。
8. 请将 0x0100～0x0200 单元的数据通过串行通信口发送至 PC 机。Fosc=24.576 MHz，采用 48 000 b/s 波特率传送。
9. 通过串行通信口接收 PC 机传送的 100 个数据存放在 0x1000 为起始单元的存储器中。Fosc=24.576 MHz，波特率=115.2 kb/s。

第 6 章　语音处理技术及应用

6.1 概　　述

6.1.1　音频信号

我们所说的音频是指频率在 20 Hz～20 kHz 的声音信号，分为波形声音信号、语音信号和音乐信号三种。其中波形声音就是自然界中的声音，是声音数字化的基础。语音也可以表示为波形声音，但波形声音表示不出语言语音学的内涵，语音是对讲话声音的一种抽象，是语言的载体，是人类社会特有的一种信息交流系统，是社会交际工具的符号。音乐与语音相比更规范一些，是符号化了的声音。但音乐不能对所有的声音进行符号化，乐谱是符号化声音的符号组，表示比单个符号更复杂的声音信息。

6.1.2　音频信号的抽样和量化

要将音频模拟信号进行数字化处理，就必须将模拟信号转换为数字信号。模拟信号数字化有多种方法，目前采用最多的是信号波形的 A/D 变换法(波形编码)。它直接将时域信号波形变换为数字序列，接收恢复的信号质量高。此外，还有参量编码(后面介绍)等。常用的波形编码方法有脉冲编码调制(PCM 调制)和增量调制(DM)。数字音频信号的质量取决于采样频率和量化位数这两个重要参数。此外，声道的数目、相应的音频设备质量也会影响音频质量。

6.1.3　音频格式

音频文件通常分为两类：声音文件和 MIDI 文件。声音文件指的是通过声音录入设备录制的原始声音信号，直接记录了真实声音的二进制数据，通常文件较大。MIDI 文件是一种音乐演奏指令序列，相当于乐谱。可以利用声音输出设备或与计算机相连的电子乐器进行演奏，由于不包含声音数据，其文件较小。

1. 声音文件格式

1) WAVE 文件(*.wav)

WAVE 文件使用三个参数来表示声音，分别是采样位数、采样频率和声道数。在计算机中采样位数一般为 8 位和 16 位两种，而采样频率有 11 025 Hz(11 kHz)、22 050 Hz(22 kHz)和 44 100 Hz(44 kHz)三种。一般 WAVE 文件的波特率可达到 88～704 kb/s。

WAVE 格式是 Microsoft 公司开发的一种声音文件格式，它符合 RIFF(Resource Interchange File Format)文件规范。它用于保存 Windows 平台的音频信息资源，Windows

平台及其应用程序均支持 WAVE 格式。

WAVE 格式支持多种压缩算法，支持多声道、多种音频位数和采样频率，是 PC 机上最流行的声音文件格式。但其文件较大，多用于存储简短的声音片段。

2) AIFF 文件(AIF/AIFF)

AIFF 是音频交换文件格式(Audio Interchange File Format)的英文缩写，是苹果计算机公司开发的声音文件格式。Macintosh 平台及其应用程序、Netscape Navigator 浏览器中的 LiveAudio、SGI 及其它专业音频软件包均支持 AIFF 文件格式。AIFF 支持 ACE2、ACE8、MAC3 和 MAC6 压缩，支持 16 位 44.1 kHz 立体声。

3) Audio 文件(.au)

Audio 文件是 Sun Micosystems 公司推出的一种经过压缩的数字声音格式，是 Internet 中常用的声音文件格式。

Netscape Navigatir 浏览器中的 LiveAudio 支持 Audio 格式的声音文件。

4) MPEG 文件(.mp1/.mp2/.mp3)

MPEG 是运动图像专家组(Moving Picture Experts Group)的英文缩写，MPEG 音频层(MPEG Audio Layer)代表 MPEG 标准中的音频部分。

MPEG 音频文件的压缩是一种有损压缩，根据压缩质量和编码复杂程度的不同可分为三层(MPEG Audio Layer1/2/3)，分别对应 MP1、MP2 和 MP3 这三种声音文件。

MPEG 音频编码具有很高的压缩率，MP1 和 MP2 的压缩率分别为 4:1 和 6:1～8:1，MP3 的压缩率则高达 10:1～12:1，就是说若一分钟 CD 音质的音乐未经压缩需要 10 MB 存储空间，而经过 MP3 压缩编码后只需 1 MB 左右的存储空间，其音质基本保持不失真。因此，目前使用最多的是 MP3 文件格式。

5) RealAudio 文件(.ra/.rm/.ram)

RealAudio 文件是 RealNerworks 公司开发的一种新型流式音频 Streaming Audio 文件格式。它包含在 RealMedia 中，主要用于在低速的广域网上实时传输音频信息。

网络传输速率不同，客户端所获得的声音质量也不尽相同，对于 28.8 kb/s 的速率可以达到广播级的声音质量，如果拥有 ISDN 或更快的线路连接，则可获得 CD 音质的声音。

2. MIDI 文件(.mid/.rmi)

MIDI 是乐器数字接口(Musical Instrument Digital Interface) 的英文缩写，是数字音乐、电子合成乐器的国际标准。它定义了计算机音乐程序合成器及其它电子设备交换音乐信号的方式，还规定了不同厂家的电子乐器与计算机连接的电缆和硬件及设备间的数据传输协议。用于为不同乐器创建数字声音信号，可以模拟大提琴、小提琴、钢琴等常见乐器。

在 MIDI 文件中，只包含产生某种声音的指令，这些指令包括使用什么 MIDI 设备的音色、声音的强弱、声音持续时间等，计算机将这些指令发送给声卡，声卡按照指令将声音合成，MIDI 在重放时可以有不同的效果。

相对于保存真实采样数据的声音文件，MIDI 文件显得更加紧凑，其文件大小比声音文件小得多。

6.1.4　语音压缩编码基础

1. 语音压缩编码中的数据量

语言压缩编码中的数据量表达式为

数据量=［(采样频率×量化位数)/8］×声道数目　(字节数)

压缩编码的目的是通过对数据的压缩，达到高效率存储和数据转换的目的。即在保证一定音质的条件下，以最小数据量来表达和传送声音信息。

2. 压缩编码的必要性

实际应用中音频数据量很大，直接进行传输或存储是不现实的，要通过对信号趋势的预测和冗余信息处理进行数据压缩，这样可以用较少的资源传输或存储更多的语音信息。

例如，没有压缩过的 CD 品质一分钟数据需要占用 11 MB 的内存空间，如果将原始数据进行压缩处理，在确保声音品质不失真的前提下，将数据压缩一半，5.5 MB 就可以完全达到原来的效果。在实际操作中可以根据需要选择适当的算法。

3. 常见的几种音频压缩编码

1) 波形编码

波形编码是将时间域信号直接变换为数字代码，使重建语音波形保持原来语音信号的形状。

波形编码的基本原理是在时间轴上对模拟语音信号按一定的速率抽样，然后将幅度样本量化、编码。译码是其逆过程，将收到的数字序列经过译码和滤波恢复成模拟信号。如脉冲编码调制 PCM (Pulse Code Modulation)、差值脉冲编码调制 DPCM、增量调制 DM、自适应差分编码 ADPCM、自适应增量调制 ADM、自适应差值脉冲编码调制 ADPCM、自适应增量调制 ADM、自适应传输编码 ATC(Adaptive Transfer Coding)和子带编码 SBC 等都属于波形编码技术。(由于篇幅有限，对各种调制的原理在此不作讨论，请读者参阅相关著作。)

其特点为话音质量高，数码率高，适用于高保真音乐及语音。

2) 参量编码

参量编码(又称为声源编码)是将信源信号在频率域或其它正交变换域提取特征参数，并将其变换成数字代码进行存储或传输的过程。它是通过对语音信号特征参数的提取和编码重建语音信号，具有尽可能高的准确性，但重建信号的波形同原语音信号的波形会有相当大的差别。线性预测编码 LPC 及其它各种改进型编码都属于参量编码。译码是编码的逆过程，将收到的数字序列经变换恢复特征参量，根据特征参量重建语音信号。参数编码比特率可压缩到 2～4.8 kb/s 甚至更低，但语音质量只能达到中等，自然度较低，常用于设备播报的提示音等。

其特点为压缩比大，计算量大，音质一般，廉价。

3) 混合编码

混合编码是将参数编码技术和波形编码技术结合起来使用。计算机的发展为语音编码技术的研究提供了强有力的工具，超大规模集成电路的出现则为语音编码的实现提供了技

术支持。20 世纪 80 年代以来语音编码技术有了实质性的进展，产生了新一代的编码算法即混合编码。它将波形编码和参数编码结合起来，克服了原有波形编码和参数编码的弱点，结合各自的优点，保持了波形编码的高质量和参数编码的速率低的特点。如多脉冲激励线性预测编码 MPLPC、规划脉冲激励线性预测编码 KPELPC、码本激励线性预测编码 CELP 等都是属于混合编码技术，其数据率和音质均介于参数编码与波形编码之间。

4. 音频压缩技术发展趋势

音频压缩技术的发展趋势如下：

(1) 降低数据率，提高压缩比，用于廉价低保真场合。如电话、机器提示音等。

(2) 追求高保真度复杂的压缩技术，用于对音质要求较高的场合。如 CD、音乐等。

6.1.5 语音合成和识别技术

1. 语音合成技术分类

按照实现的功能来分，语音合成可分为两类：

(1) 有限词汇的计算机语音输出技术。

(2) 基于语音合成技术的文字语音转换技术(TTS，Text-to-Speech)。

按照人类语言功能的不同层次分，语音合成可分为三个层次：

(1) 从文本到语音的合成(Text-to-Speech)，如图 6.1 所示；

(2) 从概念到语音的合成(Concept-to-Speech)；

(3) 从意向到语音的合成(Intention-to-Speech)。

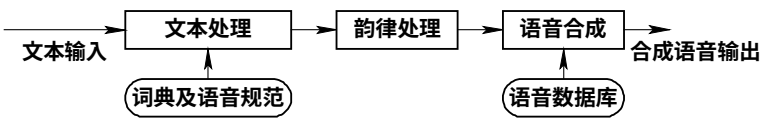


图 6.1 文本到语音转换过程

2. 语音识别

语音识别技术有三个研究领域：口音独立、连续语音和可辨认词汇数量。语音识别原理框图如图 6.2 所示。

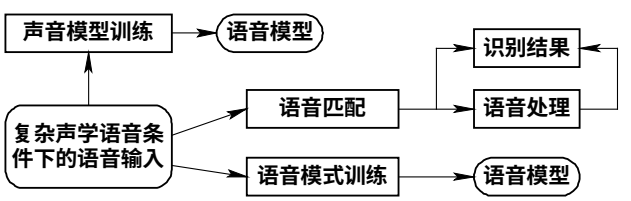


图 6.2 语音识别原理框图

1) 口音独立

口音独立的特点有：

(1) 早期只能辨认特定的使用者即特定语者 SD (Speaker Dependent)模式，可针对特定语者辨认词汇(可由使用者自行定义，如人名声控拨号)做简单快速的训练，纪录使用者的

声音特性来加以辨认。随着技术的成熟进入了语音适应阶段 SA(Speak Aadaptation)，使用者只要对于语音识别系统经过适当的口音训练即可达到一定的识别率。

(2) 非特定语者模式 SI (Speaker Independent) 使用者无需训练即可使用，任何人皆可随时使用此方式。

2) 连续语音

连续语音分为下面两种：

(1) 单字辨认：为了确保每个字音可以正确地切割出来必须逐字分开念，很不自然。

(2) 整句识别：只要按照正常说话的速度(中间不需停顿)说出来，是最直接最自然的一种方式，当然，难度也最大。虽然现阶段连续语音的识别率及正确率效果还不错，但仍需提高。由于中文有很多同音字，所以目前几乎所有的中文语音识别系统都是以词为依据来判断的。

3) 可辨认词汇数量

内建的词汇数据库的大小直接影响其识别能力，因此就语音识别的词汇数量来说可分为小词汇量(10～100)、中词汇量(100～1000)和无限词汇量(即听写机)三种。

6.2　凌阳音频技术简介

6.2.1　压缩算法的编码标准

表 6.1 列出了不同音频质量的编码技术标准响应频率。凌阳音频压缩算法处理的语音信号频率范围为 200 Hz～3.4 kHz。

表 6.1　编码技术标准频响

信号类型	频率范围/Hz	采样频率/kHz	量化精度/位
电话话音	200～3400	8	8
宽带音频 (Am 音质)	50～7000	16	16
调频广播 (FM 音质)	20～15 k	37.8	16
高质量音频 (CD 音质)	20～20 k	44.1	16

6.2.2　压缩分类

压缩分为无损压缩和有损压缩。无损压缩一般指磁盘文件，压缩比低(2:1～4:1)；有损压缩指语音、视频文件，压缩比高(100：1)。

凌阳音频压缩算法根据不同的压缩比分为以下几种(以压缩 8 KB 16 位的 WAVE 文件为例):

SACM_A2000 压缩比为 8:1, 8:1.25, 8:1.5；

SACM_S480 压缩比为 80:3, 80:4.5；

SACM_S240 压缩比为 80:1.5。

音质排序为：A2000>S480>S240。

6.2.3 常用的音频形式和压缩算法

1. 波形编码

波形编码采用子带编码技术(SBC，Sub-Band Coding)压缩算法 SACM_A2000。特点为话音质量高，编码率高，适用于高保真语音和音乐。

2. 参量编码

参量编码采用压缩算法 SACM_S240。特点为压缩比高，计算量大，音质一般，价格低廉。

3. 混合编码

混合编码采用压缩算法 SACM_S480。特点为综合了参数编码和波形编码之优点。除此之外，还具有 FM 音乐合成式即 SACM_MS01 等。

凌阳 SPCE061A 是 16 位单片机，具有 DSP 功能，有很强的信息处理能力，最高时钟可达到 49 MHz，具备运算速度高的优势。这无疑为语音的录制、播放、合成及识别提供了条件。

凌阳压缩算法中 SACM_A2000、SACM_S480、SACM_S240 主要是用来放音，可用于语音提示；DVR 则用来录、放音。由于语音合成(MS01)算法较繁琐，而且需要具备音乐理论配器法和声学知识，这里不作进一步介绍，读者可参考有关手册。

语音识别功能框图如图 6.3 所示。

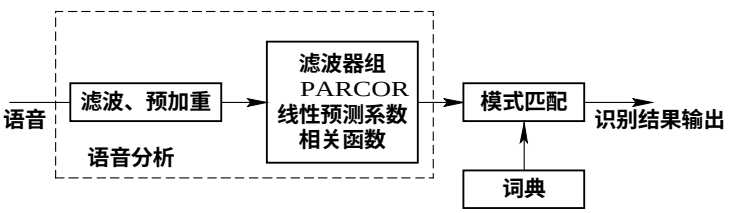


图 6.3 语音识别功能框图

6.3 常用应用程序接口

6.3.1 程序接口概述

语音和音乐与我们的生活有着非常密切的关系，而单片机对语音的控制如录放音合成及识别也广泛应用于现实生活中。语音处理可以分为 A/D 编码处理、存储解码处理以及 D/A 转换等，如图 6.4 所示。

由于麦克风输入所生成的 WAVE 文件占用的存储空间较大，对于单片机来说想要存储大量的信息显然是不现实的，凌阳公司提出了解决的方法，即建立 SACM-LIB 库及相应的应用程序接口 API(Application Programming Interface)函数。将 A/D 编码、解码、存储及 D/A 转换成相应的模块，每个模块都有其 API，我们只需了解每个模块所要实现的功能及其参

数的内容，然后调用该 API 函数即可实现其功能。如在程序中插入语音提示或连续播放一段语音或音乐，也可以根据自己的需要，选择合适的算法，如表 6.2 所示。

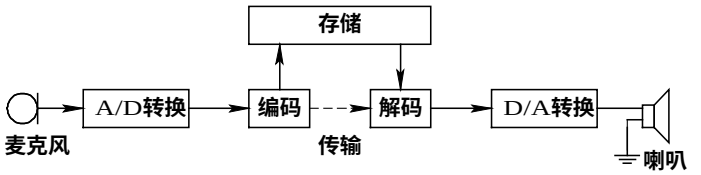


图 6.4　语音处理模块组成

表 6.2　SACM-LIB 库中模块及其算法

模块名称 (Model-Index)	语音压缩编码率	采样速率 /kHz
SACM_A2000	16 kb/s, 20 kb/s, 24 kb/s	16
SACM_S480/S720	4.8 kb/s, 7.2 kb/s	16
SACM_S240	2.4 kb/s	24
SACM_MS01	音乐合成(16 kb/s, 20 kb/s, 24 kb/s)	16
SACM_DVR (A2000)	16 kb/s 数据率, 8 KB 的采样率, 用于 ADC 通道录音功能	16

我们知道，对于语音处理大致可以分为 A/D、编码、存储、解码及 D/A 等。凌阳提供了 SACM-LIB，该库将 A/D、编码、解码、存储及 D/A 作为相应的模块，对于每个模块都有其应用程序接口 API，你只需了解每个模块所要实现的功能及其参数的内容，然后调用该 API 函数即可实现该功能。SACM-LIB 目前主要有 sacmv25.lib 和 sacmv32.lib 两种，请读者注意这两个库在使用时有所不同。

6.3.2　SACM_A2000

SACM_A2000 压缩算法具有压缩比较小(8:1)、语音质量好、传输码率高的特点，适用于高保真音乐和语音。

1. 相关 API 函数

SACM_A2000 相关 API 函数及注释如下：

```
void SACM_A2000_Initial(int Init_Index)                          //初始化
void SACM_A2000_ServiceLoop(void)                                //获取语音数据填入译码队列
void SACM_A2000_Play(int Speech_Index, int Channel, int Ramp_Set)    //播放
void SACM_A2000_Stop(void)                                       //停止播放
void SACM_A2000_Pause (void)                                     //暂停播放
void SACM_A2000_Resume(void)                                     //暂停后恢复
void SACM_A2000_Volume(Volume_Index)                             //音量控制
unsigned int SACM_A2000_Status(void)                             //获取模块状态
void SACM_A2000_InitDecode(int Channel)                          //译码初始化
void SACM_A2000_Decode(void)                                     //译码
```

void SACM_A2000_FillQueue(unsigned int encoded-data) //填充队列
unsigned int SACM_A2000_TestQueue(void) //测试队列
Call F_FIQ_Service_ SACM_A2000 //中断服务函数

2. 功能介绍

下面对上述的 API 函数作具体介绍。

(1) C：void SACM_A2000_Initial(int Init_Index)
ASM： R1=[Init_Index]
Call F_ SACM_A2000_Initial

功能：SACM_A2000 语音播放之前的初始化。
参数：Init_Index=0 表示手动方式。
Init_Index=1 表示自动方式。
返回值： 0 代表语音模块初始化失败。
1 代表初始化成功。
注意：该函数用于对定时器中断和 DAC 等初始化。

(2) C：void SACM_A2000_ServiceLoop(void)
ASM： Call F_ SACM_A2000 _ServiceLoop

功能：从资源中获取 SACM_A2000 语音数据，并将其填入译码队列中。
参数：无。
返回值：无。

(3) C：int SACM_A2000_Play(int Speech_Index,
int Channel, int Ramp_Set)
ASM： R1=[Speech _Index]
R2=[Channel]
R3=[Ramp_Set]
Call SACM_A2000_Play

功能：播放资源中 SACM_A2000 语音或乐曲。
参数：Speech _Index 表示语音索引号。
Channel 1：通过 DAC1 通道播放。
2：通过 DAC2 通道播放。
3：通过 DAC1 和 DAC2 双通道播放。
Ramp_Set 0：禁止音量增/减调节。
1：仅允许音量增调节。
2：仅允许音量减调节。
3：允许音量增/减调节。
返回值：无。
注意：

① SACM_A2000 的数据率有 16 kb/s、20 kb/s 和 24 kb/s 三种，可在同一模块的几种算法中选择一种。

② Speech_Index 是定义在 resource.inc 文件中资源表 T_SACM_A2000_SpeechTable 的偏移地址。

③ 中断服务子程序 F_FIQ_Service_ SACM_A2000 必须安置在 TMA_FIQ 中断向量上。

④ 函数允许 TimerA 以所选的数据采样率计数，并产生溢出中断信号。

例 1　以 SCAM_A2000 自动方式播报一段语音，并自动结束。

解：程序流程图如图 6.5 所示。

主程序：

```
#include    "A2000.h"
#define Speech_1                0
#define DAC1                    1
#define DAC2                    2
#define Ramp_UpDn_Off           0
#define Ramp_Up_On              1
#define Ramp_Dn_On              2
#define Ramp_UpDn_On            3

Main ()
{
        SACM_A2000_Initial(1);
        SACM_A2000_Play(Speech_1, DAC1+DAC2, Ramp_UpDn_On);   //放音
        while(SACM_A2000_Status()&0x01)
    {
SACM_A2000_ServiceLoop();
    }
}
```

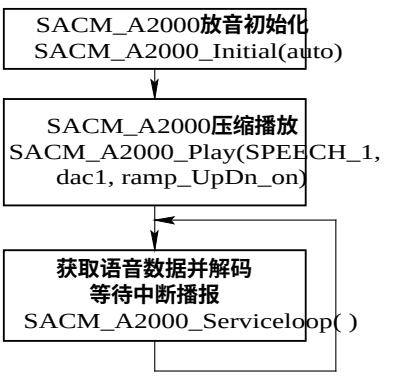


图 6.5　A2000 自动方式主程序流程图

注意：本例要链接语音库 sacmv25.lib 以及和该库对应的 hardware.asm。

中断服务程序:

```
_FIQ:
    PUSH R1,R4 TO [SP]                          //将寄存器压栈保护
    CALL F_FIQ_Service_SACM_A2000
    R1 = 0x2000                                 //清中断标志位
    [P_INT_Clear] = R1
    POP R1,R4 TO [SP]                           //寄存器出栈
    RETI
```

注意：当播放语音文件数据中出现 FF FF FFH 时停止播放。

(4) C：void SACM_A2000_Stop(void)

ASM：Call F_ SACM_A2000_Stop

功能：停止播放 SACM_A2000 语音或乐曲。

参数：无。

返回值：无。

(5) C：void SACM_A2000_Pause (void)

ASM：Call F_ SACM_A2000_Pause

功能：暂停播放 SACM_A2000 **语音或乐曲。**

参数：无。

返回值：无。

(6) C：void SACM_A2000_Resume(void)

ASM：Call F_ SACM_A2000_Resume

功能：播放暂停播放的 SACM_A2000 **语音或乐曲。**

参数：无。

返回值：无。

(7) C：void SACM_A2000_Volume(Volume_Index)

ASM： R1=[Volume_Index]

Call F_ SACM_A2000_Volume

功能：在播放 SACM_A2000 **语音或乐曲时改变主音量**

参数：Volume_Index **为音量数，音量从最小到最大可在** 0～15 **之间选择。**

返回值：无。

(8) C：unsigned int SACM_A2000_Status(void)

ASM：Call F_ SACM_A2000_ Status

[**返回值**]=R1

功能：获取 SACM_A2000 **语音播放的状态。**

参数：无。

返回值：当 R1 **的** bit0=0 **时，表示语音播放结束；**bit0=1 **表示语音在播放中。**

(9) ASM: Call F_FIQ_Service_ SACM_A2000

功能：作 SACM_A2000 **语音背景程序的中断服务子程序，通过前台子程序自动方式的** ACM_A2000_ServiceLoop **及手动方式的** SACM_A2000_Decode **对语音数据进行解码，然后将其送入** DAC **通道播放。**

参数：无。

返回值：无。

注意： SACM_A2000 **语音中断服务程序只有汇编指令形式且应将此程序放置在** TMA_FIQ **中断源上。**

(10) C: void SACM_A2000_InitDecode(int Channel)

ASM: Call F_ SACM_A2000_Decode

功能：开始对 SACM_A2000 **语音数据以非自动方式(编程控制)进行译码。**

参数：Channel=1、2、3 **分别表示使用** DAC1、DAC2 **通道以及** DAC1 **和** DAC2 **双通道。**

返回值：无。

注意：只能通过非自动方式对语音数据解压缩。

(11) C：void SACM_A2000_Decode(void)

ASM： Call F_ SACM_A2000_Decode

功能：从语音队列里获取语音数据并进行译码，然后通过中断服务子程序将其送入 DAC **通道播放。**

参数：无。

返回值：无。

注意：只能通过非自动方式对语音数据进行译码。

(12) C：void SACM_A2000_FillQueue(unsigned int encoded−data)

ASM： R1=[**语音编码数据**]

Call F_ SACM_A2000_FillQueue

功能：将从用户存储区里获取 SACM_A2000 **语音编码数据填入语音队列中等候译码处理。**

参数：encoded−data **为语音编码数据。**

返回值：无。

注意：只能通过非自动方式对语音数据进行译码。

(13) C：unsigned int SACM_A2000_TestQueue(void)

ASM： Call F_ SACM_A2000_TestQueue

[**返回值**]=R1

功能：获取语音队列的状态。

参数：无。

返回值：R1=0、1、2 **分别表示语音队列不空不满、语音队列满及语音队列空。**

注意：只能通过非自动方式测试语音队列状态。

例 2　**编程控制以非自动方式利用** SACM_A2000 **函数播报语音。**

解：主程序和中断服务程序流程图如图 6.6 **和图** 6.7 **所示。**

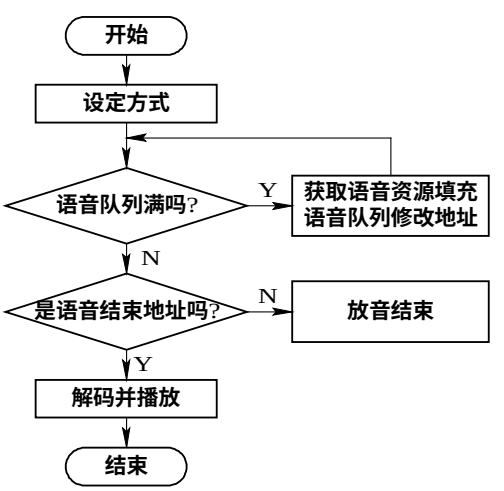


图 6.6　A2000 **非自动方式主程序流程图**

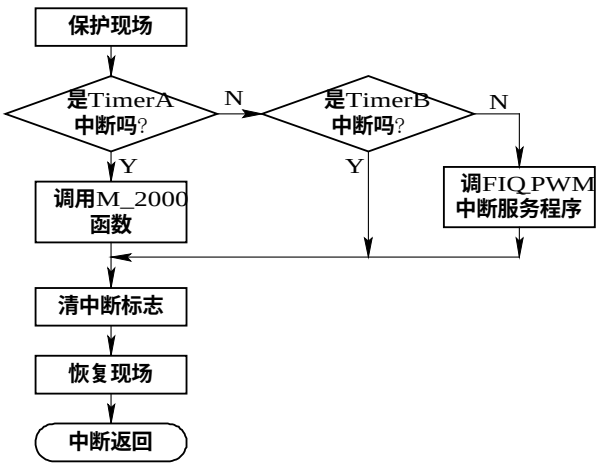


图 6.7　中断服务子程序流程图

主程序：

```
#include    "A2000.h"
#define     Manual          0
#define     Auto            1
#define     Full            1
#define     Empty           2
#define     DAC1            1
#define     DAC2            2

        Main()
{
        Addr=RES_A32_SA;                        //长整型资源地址
        SACM_A2000_Initial(Manual);             //选择非自动方式
        SACM_A2000_InitDecode(DAC1+DAC2);       //使用双通道
        While(SACM_A2000_TestQueue()!=Full)     //若队列不满填入数据

        {
        Ret=GetResource(Addr);                  //从 ROM 中取语音数据
        SACM_A2000_FillQueue(Ret);              //将语音数据填入队列
        Addr++;                                 //指向下一个数据地址
        }
    While(1)
  {
        If(SACM_A2000_TestQueue()!=Full)        //继续填数据到队列中
     {
```

```
            Ret=GetResource(Addr);
            SACM_A2000_FillQueue(Ret);
            Addr++;
        }
            if(Addr< RES_A32_EA)
            SACM_A2000_Decode();                    //对语音数据进行译码
            else SACM_A2000_Stop();                 //地址结束停止播放
    }
}
```

注意:

① 文件的结束是由用户位址变量控制的。

② 非自动方式播放语音时，其音量的增/减是通过外部子程序 SP_Ramp_Up **和** SP_Ramp_Dn **控制的。**

③ 本例要链接语音库 sacmv25.lib **以及和该库对应的** hardware.asm**。**

6.3.3　SACM_S480

SACM_S480 **压缩算法压缩比大**(80:3), **存储容量、音质介于** A2000 **和** S240 **之间，适用于一般场合的语音播报，如文曲星词典等。**

1. **相关** API **函数**

相关 API **函数如下所示**

```
int SACM_S480_Initial(int Init_Index)                    //初始化
void SACM_ S480_ServiceLoop(void)                        //获取语音数据填入译码队列
void SACM_ S480_Play(int Speech_Index, int Channel, int Ramp_Set)
                                                         //播放
void SACM_ S480_Stop(void)                               //停止播放
void SACM_S480_Pause (void)                              //暂停播放
void SACM_S480_Resume(void)                              //暂停后恢复
void SACM_S480_Volume(Volume_Index)                      //音量的控制
unsigned int SACM_S480_Status(void)                      //获取模块的状态
Call F_FIQ_Service_ SACM_S480                            //中断服务函数
```

2. **功能介绍**

下面对 ASCM_S480 **的功能作具体介绍。**

(1) C：int SACM_S480_Initial(int Init_Index)

ASM： R1=[Init_Index]

Call F_ SACM_ S480_Initial

功能：SACM_S480 **语音播放之前的初始化。**

参数：Init_Index=0 **表示手动方式；**

Init_Index=1 **表示自动方式。**

返回值：0　**代表语音模块初始化失败。**

1　**代表初始化成功。**

注意：该函数用于对定时器中断和 DAC **等的初始化。**

(2) C：void SACM_S480_ServiceLoop(void)

ASM：Call F_ SACM_S480_ServiceLoop

功能：从资源中获取 SACM_S480 **语音数据并将其填入解码队列中。**

参数：无。

返回值：无。

注意：播放语音文件中数据，当出现 FF FF FFH **数据时便停止播放。**

(3) C：int SACM_S480_Play(int Speech_Index,

int Channel, Int Ramp_Set)

ASM：　R1=[Speech _Index]

R2=[Channel]

R3=[Ramp_Set]

Call SACM_S480_Play

功能：播放语音资源库中 SACM_S480 **语音。**

参数：Speech _Index **表示语音索引号。**

Channel　1：**通过** DAC1 **通道播放。**

2：**通过** DAC2 **通道播放。**

3：**通过** DAC1 **和** DAC2 **双通道播放。**

Ramp_Set　0：**禁止音量增/减调节。**

1：**仅允许音量增调节。**

2：**仅允许音量减调节。**

3：**允许音量增/减调节。**

返回值：无。

注意：

① SACM_S480 **的数据率有** 4.8 kb/s **和** 7.2 kb/s **两种，可在同一模块的几种算法中自动选择一种。**

② Speech_Index **是定义在** resource.inc **文件中资源库**(T_SACM_S480_SpeechTable)**的偏移地址。**

③ **中断服务程序中** F_FIQ_Service_SACM_S480 **必须放在** TMA_FIQ **中断向量上。**

④ **函数允许** TimerA **以所选的的数据采样率(计数溢出)中断。**

例 3　**以** SACM_S480 **自动方式播放一段语音并自动结束。**

解：主程序流程图如图 6.8 **所示，中断程序流程同上例。**

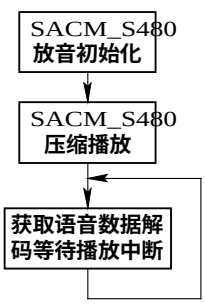


图 6.8　S480 **自动方式主程序流程图**

主程序：

```
#include        "s480.h"
#define         Speech_1            0
#define         DAC1                1
#define         DAC2                2
#define         Ramp_UpDn_Off       0
#define         Ramp_Up_On          1
#define         Ramp_Dn_On          2
#define         Ramp_UpDn_On        3

  Main ()
{
        SACM_S480_Initial(1);
        SACM_S480_Play(Speech_1, DAC1+DAC2, Ramp_UpDn_On);
                                                    //放音
        while(SACM_S480_Status()&0x01)
    {
        SACM_S480_ServiceLoop();
    }
}
```

中断服务程序:

```
_FIQ:
    PUSH R1,R4 TO [SP]                          //将寄存器压栈保护
    CALL F_FIQ_Service_SACM_S480
    R1 = 0x2000
    [P_INT_Clear] = R1                          //清除中断请求标志
    POP R1,R4 FROM [SP]                         //寄存器出栈
    RETI
```

注意：

① 自动放音时，当语音资源文件中的数据为 FF FF FFH **时停止播放。**

② 本例要链接语音库 sacmv25.lib **以及和该库对应的** hardware.asm**。**

(4) C：void SACM_S480_Stop(void)

ASM：Call F_ SACM_S480_Stop

功能：停止播放 SACM_S480 **语音。**

参数：无。

返回值：无。

(5) C：void SACM_S480_Pause (void)

ASM：Call F_ SACM_S480_Pause

功能：暂停播放 SACM_S480 **语音。**

参数：无。

返回值：无。

(6) C：void SACM_S480_Resume(void)

ASM：Call F_ SACM_S480_Resume

功能：恢复暂停播放的 SACM_S480 **语音。**

参数：无。

返回值：无。

(7) C：void SACM_S480_Volume(Volume_Index)

ASM：R1=[Volume_Index]

Call F_Model-Index_Volume

功能：在播放 SACM_S480 **语音时改变主音量。**

参数：Volume_Index **为音量数音量从最小到最大，可在** 0～15 **之间选择。**

返回值：无。

(8) C：unsigned int SACM_S480_Status(void)

ASM：Call F_ SACM_S480_ Status

[**返回值**]=R1

功能：获取 SACM_S480 **语音播放的状态。**

参数：无。

返回值：当 R1 **的值** bit0=0 **时表示语音播放结束，**bit0=1 **时表示语音在播放中。**

(9) ASM：Call F_FIQ_Service_ SACM_S480

功能：用作 SACM_S480 **语音背景程序的中断服务子程序，通过前台子程序自动方式的** SACM_S480_ServiceLoop **及手动方式的** SACM_S480_Decode **对语音数据进行解码送入** DAC **通道播放。**

参数：无。

返回值：无。

注意：SACM_S480 **语音背景子程序只有汇编指令形式且应将此子程序安置在** TMA_FIQ **中断源上。**

6.3.4 SACM_S240

SACM_S240 **压缩算法的压缩比大**(80:1.5)**、价格低、音质一般，适用于对保真度要求不高的场合**，**如玩具类产品的批量生产，编码率仅为** 2.4 kb/s**。**

1. **相关** API **函数**

SACM_S240 **相关** API **函数如下：**

```
int SACM_S240_Initial(int Init_Index)                    //初始化
void SACM_ S240_ServiceLoop(void)                        //获取语音数据填入译码队列
void SACM_ S240_Play(int Speech_Index, int Channel, int Ramp_Set)
                                                         //播放
void SACM_ S240_Stop(void)                               //停止播放
```

```
void SACM_S240_Pause (void)                    //暂停播放
void SACM_S240_Resume(void)                    //暂停后恢复
void SACM_S240_Volume(Volume_Index)            //音量控制
unsigned int SACM_S240_Status(void)            //获取模块状态
Call F_FIQ_Service_ SACM_S240                  //中断服务函数
```

2. **功能介绍**

下面对 SACM_S240 **的相关** API **函数作具体介绍。**

(1) C：void SACM_ S240_Initial(int Init_Index)

ASM：R1=[Init_Index]

Call F_ SACM_ S240_Initial

功能：SACM_ S240 **语音播放之前的初始化。**

参数：Init_Index=0 **表示手动方式；**

Init_Index=1 **表示自动方式。**

返回值：0 **代表语音模块初始化失败，**1 **代表初始化成功。**

注意：函数用于 S240 **语音译码的初始化以及相关设备的初始化。**

(2) C：void SACM_ S240_ServiceLoop(void)

ASM：Call F_ SACM_S240 _ServiceLoop

功能：从资源中获取 SACM_S240 **语音数据并将其填入解码队列中。**

参数：无。

返回值：无。

(3) C：int SACM_ S240_Play(int Speech_Index, int Channel, int Ramp_Set)

ASM：R1=[Speech _Index]

R2=[Channel]

R3=[Ramp_Set]

Call SACM_S240_Play

功能：播放资源中 SACM_ S240 **语音。**

参数：Speech _Index **表示语音索引号。**

Channel 1：**通过** DAC1 **通道播放。**

2：**通过** DAC2 **通道播放。**

3：**通过** DAC1 **和** DAC2 **双通道播放。**

Ramp_Set 0：**禁止音量增/减调节。**

1：**仅允许音量增调节。**

2：**仅允许音量减调节。**

3：**允许音量增/减调节。**

返回值：无。

注意：

① SACM_S240 **的数据率为** 2.4 kb/s。

② Speech_Index **是定义在** resource.inc **文件中资源表**(T_SACM_S240_SpeechTable)**的**

偏移地址。

③ 中断服务子程序 F_FIQ_Service_ SACM_S240 必须安置在 TMA_FIQ 中断向量上。

④ 函数允许 TimerA 以所选的的数据采样率计数溢出中断。

例 4 以 SACM_S240 自动方式播报一段语音并自动结束。

解：主程序流程图如图 6.9 所示，中断服务子程序流程同上例。

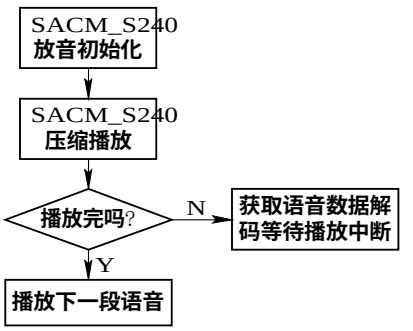


图 6.9 S240 在自动方式下的主程序流程图

主程序：

```
#include        "sp_lib.h"
#include        "sacmv32.h"
#define         Speech_1            0
#define         DAC1                    1
#define         DAC2                    2
#define         Ramp_UpDn_Off           0
#define         Ramp_Up_On          1
#define         Ramp_Dn_On          2
#define         Ramp_UpDn_On            3
 main()
{
        int SpeechIndex = 0;                            //选择第一首语音
        while(1)
    {
        SACM_S240_Initial(Auto);
        SACM_S240_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);//播放第一首
        while(SACM_S240_Status()&0x01)                  //判断第一首是否播完
        SACM_S240_ServiceLoop();
        SpeechIndex++;
     SACM_S240_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On); //播放第二首
        while(SACM_S240_Status()&0x01)                  //判断第二首是否播完
        SACM_S240_ServiceLoop();
        SpeechIndex++;
```

```
        SACM_S240_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);   //播放第三首
            while(SACM_S240_Status()&0x01)
            SACM_S240_ServiceLoop();
            SpeechIndex=0;                                      //重新播放
        }
}
```

中断服务程序:

```
_FIQ:
            PUSH R1,R4 TO [SP]                                  //现场保护
            CALL F_FIQ_Service_SACM_S240
            R1 = 0x2000
            [P_INT_Clear] = R1                                  //清除中断请求标志
            POP R1,R4 FROM[SP]                                  //现场恢复
            RETI
```

注意：自动放音时当语音资源文件中的数据为 FF FF FFH **时便停止播放。**

(4) C：void SACM_ S240_Stop(void)
ASM：Call F_ SACM_ S240_Stop
功能：停止播放 SACM_ S240 **语音。**
参数：无。
返回值：无。

(5) C：void SACM_ S240_Pause (void)
ASM：Call F_ SACM_ S240_Pause
功能：暂停播放 SACM_ S240 **语音。**
参数：无。
返回值：无。

(6) C：void SACM_ S240_Resume(void)
ASM：Call F_ SACM_ S240_Resume
功能：恢复暂停播放的 SACM_ S240 **语音的播放。**
参数：无。
返回值：无。

(7) C：void SACM_ S240_Volume(Volume_Index)
ASM： R1=[Volume_Index]
Call F SACM_ S240_Volume
功能：在播放 SACM_ S240 **语音时改变主音量。**
参数：Volume_Index **为音量数，音量从最小到最大可在** 0～15 **之间选择。**
返回值：无。

(8) C：unsigned int SACM_ S240_Status(void)
ASM：Call F_ SACM_ S240_ Status
[**返回值**]=R1

功能：获取 SACM_ S240 **语音播放的状态。**

参数：无。

返回值：当 R1 **中** bit0=0 **时表示语音播放结束，**bit0=1 **时表示语音在播放中。**

(9) ASM：Call F_FIQ_Service_ SACM_ S240

功能：用作 SACM_S240 **语音背景程序的中断服务子程序，通过前台子程序自动方式的** SACM_S240_ServiceLoop **及手动方式的** Model−Index_Decode **对语音数据进行译码送入** DAC **通道播放。**

参数：无。

返回值：无。

注意：SACM_S240 **语音背景子程序只有汇编指令形式，且应将此子程序放置在** TMA_FIQ **中断源上。**

6.3.5 SACM_MS01

SACM_MS01 **算法较繁琐，但只要具备音乐理论、配器法和声学知识，了解** SPCE **编曲格式者均可尝试。遵照** SPCE **编曲格式用** DTM MIDI(**音源＋**MIDI **键盘＋作曲软件)的方法演奏自动生成***.mid **文件，然后用凌阳** MIDI2POP.exe **转成***.pop **文件，但需要专业设备，具备键盘声乐演奏技能，了解** SPCE **编曲格式。对于初学者或非专业用途，一般了解放音或录放音及相关函数即可。**

1. **相关** API **函数**

SACM_MS01 **的相关** API **函数如下：**

```
void SACM_MS01_Initial(int Init_Index)          //初始化
void SACM_ MS01_ServiceLoop(void)               //获取语音数据填入译码队列
int SACM_ MS01_Play(int Speech_Index, int Channel,
                    int Ramp_Set)               //播放
void SACM_ MS01_Stop(void)                      //停止播放
void SACM_ MS01_Pause (void)                    //暂停播放
void SACM_ MS01_Resume(void)                    //暂停后恢复
void SACM_ MS01_Volume(int Volume_Index)        //音量控制
unsigned int SACM_ MS01_Status(void)            //获取模块状态
void SACM_ MS01_ChannelOn(int Channel)          //接通通道
void SACM_ MS01_ChannelOff(int Channel)         //关闭通道
void SACM_ MS01_SetInstrument(Channel,Instrument,Mode)
                                                //设置乐曲配器类型中断服务函数：
Call F_FIQ_Service_ SACM_MS01
Call F_IRQ2_Service_ SACM_MS01
Call F_IRQ4_Service_ SACM_MS01
```

2. **功能介绍**

下面对 SACM_MS01 **的相关** API **函数作具体介绍。**

(1) C：void SACM_MS01_Initial(int Init_Index)

ASM：R1=[Init_Index]

Call F_ SACM_MS01_Initial

功能：SACM_MS01 **语音播放之前的初始化，设置中断源、定时器和播放方式(手动、自动)。**

参数：Init_Index=0 **表示手动方式。**

Init_Index=1 **表示自动方式。**

Init_Index　0：**代表** PWM **音频输出方式。**

1：DAC **音频输出方式下** 24 kHz **的播放率。**

2：DAC **音频输出方式下** 20 kHz **的播放率。**

3：DAC **音频输出方式下** 16 kHz **的播放率。**

返回值：0 **代表语音模块初始化失败，**

1 **代表** SACM_MS01 **初始化成功。**

注意：

① **该函数初始化** MS01 **译码器以及系统时钟** System clock、TimerA、TimerB、DAC，**并且以** 16/20/24 kHz **采样率触发** FIQ_TMA **中断。**

② **初始化后接通所有播放通道**(0～5)。

③ **对于** SACM_MS01 **模块，**FIQ **中断服务子程序用于从前台程序**(SACM_MS01_ServiceLoop)**的执行过程中获取乐曲译码数据。若事件不是由音符而是鼓点节奏引起的，则其自适应音频脉冲编码方式** ADPCM **数据将被传入** IRQ2 **进行译码，然后将二者混合在一起送** DAC **通道播放。**

(2) C：void SACM_ MS01_ServiceLoop(void)

ASM：Call F_ SACM_MS01 _ServiceLoop

功能：从资源中获取 SACM_ MS01 **语音数据并将其填入译码队列自动译码。**

参数：无。

返回值：无。

(3) C：int SACM_ MS01_Play(int Speech_Index, int Channel, int Ramp_Set)

ASM： R1=[Speech _Index]

R2=[Channel]

R3=[Ramp_Set]

Call SACM_ MS01_Play

功能：开始播放一种 SACM_ MS01 **音调。**

参数：Speech _Index **表示语音索引号。**

Channel　1：**通过** DAC1 **通道播放。**

2：**通过** DAC2 **通道播放。**

3：**通过** DAC1 **和** DAC2 **双通道播放。**

Ramp_Set　0：**禁止音量增/减调节。**

1：**仅允许音量增调节。**

2：**仅允许音量减调节。**

3：**允许音量增/减调节。**

返回值：无。

注意：

① SACM_MS01 **的数据率有** 16 kb/s、20 kb/s **和** 24 kb/s **三种，可在同一模块的几种算法中自动选择一种。**

② 0Speech_Index **是定义在** resource.inc **文件中资源表** T_SACM_MS01_SpeechTable **的偏移地址。**

③ **中断服务子程序中** F_FIQ_Service_ SACM_MS01 **必须放在** TMA_FIQ **中断向量上。**

例 5 采用 SACM_MS01 **自动方式播报一段音乐。**

解：程序较简单，流程图请参考上例。

主程序：

```
#define     PWM             0
#define     DAC_24K         1
#define     DAC_20K         2
#define     DAC_16K         3
#define     DAC1            1
#define     DAC2            2
#define     Ramp_UpDn_Off   0
#define     Ramp_Up_On      1
#define     Ramp_Dn_On      2
#define     Ramp_UpDn_On    3
Main ()
{
        SACM_MS01_Initial(1);
        SACM_MS01_Play(Speech_1, DAC1+DAC2, Ramp_UpDn_On);          //放音
        while(SACM_MS01_Status()&0x01)
    {
        SACM_MS01_ServiceLoop();
    }
}
```

中断服务程序：

```
_FIQ:
        PUSH registers
        CALL F_FIQ_Service_SACM_MS01            //4 通道的配器服务程序
        clear interrupt flag
        POP registers
    RETI
_IRQ2:
        PUSH registers
```

```
        CALL F_IRQ2_Service_SACM_MS01          //2 通道鼓点节奏服务程序
        clear interrupt flag
        POP registers
    RETI
_IRQ4:
        PUSH registers
        CALL F_IRQ4_Service_SACM_MS01          //控制节拍服务程序
        clear interrupt flag
        POP registers
    RETI
```

(4) C：void SACM_ MS01_Stop(void)

ASM：Call F_ SACM_ MS01_Stop

功能：停止播放 SACM_ MS01 **乐曲。**

参数：无。

返回值：无。

(5) C：void SACM_ MS01_Pause (void)

ASM：Call F_ SACM_ MS01_Pause

功能：暂停播放 SACM_MS01 **乐曲。**

参数：无。

返回值：无。

(6) C：void SACM_ MS01_Resume(void)

ASM：Call F_ SACM_ MS01_Resume

功能：恢复暂停播放的 SACM_ MS01 **语音或乐曲。**

参数：无。

返回值：无。

(7) C：void SACM_ MS01_Volume(int Volume_Index)

ASM：R1=[Volume_Index]

Call F SACM_ MS01_Volume

功能：在播放 SACM_ MS01 **语音时改变主音量。**

参数：Volume_Index **为音量数，音量从最小到最大可在** 0～15 **之间选择。**

返回值：无。

(8) C：unsigned int SACM_ MS01_Status(void)

ASM：Call F_ SACM_ MS01_ Status

[Return_Value]=R1

功能：获取 SACM_ MS01 **合成音乐播放的状态。**

参数：无。

返回值：R1 **中** bit0=0 **表示语音播放结束，**bit0=1 **表示语音在播放中。其它状态位** bit8～bit13 **的功能如图** 6.10 **所示。**

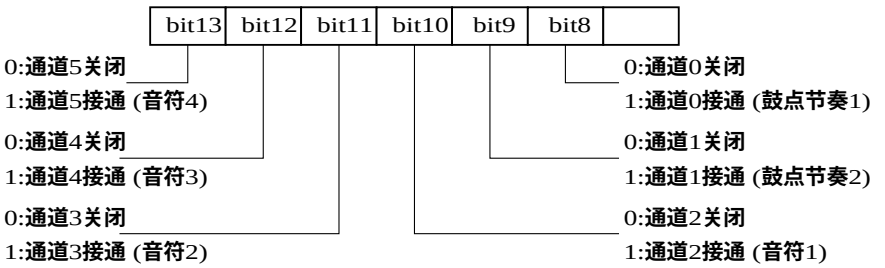


图 6.10 SACM_MS01 **部分状态返回值**

(9) C：void SACM_MS01_ChannelOn(int Channel)

ASM：R1=[Channel]

Call F_SACM_MS01_ChannelOn

功能：将 SACM_MS01 **乐曲播放通道之一接通。**

参数：Channel **为** 0～5 **之间整数，其中** 0 **和** 1 **代表鼓点节奏通道，**2～5 **则代表音符通道。**

返回值：无。

(10) C：void SACM_MS01_ChannelOff(int Channel)

ASM：R1=[Channel]

Call F_SACM_MS01_ChannelOff

功能：将 SACM_MS01 **乐曲播放通道之一关断。**

参数：Channel **为** 0～5 **之间整数，其中** 0 **和** 1 **代表鼓点节奏通道，**2～5 **则代表音符通道。**

返回值：无。

例 6 **利用** SACM_MS01 **以** 6 **通道方式播报一段音乐。流程图请参考前面例题。**

程序如下：

```
int main()
  {
        SACM_MS01_Initial(DAC_24K);
                              //选择 DAC 24 kHz 播放方式，且 6 个通道都接通
        SACM_MS01_ChannelOff(0);          //通道 0 被关断(鼓点节奏 1)
        SACM_MS01_ChannelOff(5);          //通道 5 被关断(音符 4，频率 Tone 方式)
        SACM_MS01_Play(SongIndex,DAC1+DAC2);     //开始播放乐曲
        while(SACM_MS01_Status()&0x01)            //若状态返回值的 bit0=1
    {
        SACM_MS01_ServiceLoop();          //SACM_MS01 乐曲播放的译码程序
    }
}
```

(11) C：void SACM_MS01_SetInstrument(int Channel, int Instrument, int Mode)

ASM：R1=[Channel]

R2=[Instrument]

R3=[Mode]

Call F_SACM_MS01_SetInstrument

功能：在 SACM_MS01 **的一个播放通道上改变乐曲配器类型。**

参数：Channel=0～5，**其中** 0 **和** 1 **代表鼓点节奏通道；**2～5 **代表音符通道。**

通道 0 **和** 1 Instrument=0～Max_Drum#**代表不同的鼓点节奏；通道** 2～5 Instrument=0～34 **代表不同的乐曲配器类型。**

Mod=0，1　**分别代表配器类型可以或不可通过乐曲事件改变。**

返回值：无。

(12) ASM：Call F_FIQ_Service_ SACM_ MS01

ASM：Call F_IRQ2_Service_ SACM_ MS01

ASM：Call F_IRQ4_Service_ SACM_ MS01

功能：SACM_MS01 **模块** FIQ **中断服务子程序用于从前台程序。**

SACM_MS01_ ServiceLoop **的执行过程中获取乐曲译码数据，若未来事件不是音符而是由鼓点节奏引起，则其自适应音频脉冲编码方式** ADPCM **数据将被传入** IRQ2 **进行译码，然后将二者混合在一起送出** DAC **通道播放。**

参数：无。

返回值：无。

注意：

① SACM_MS01 **语音背景子程序只有汇编指令形式。**

② **中断服务子程序必须在** TMA_FIQ **中断源上。**

③ **应将两个额外的中断服务子程序分别放置在** IRQ2_TMB **和** IRQ4_1K **中断源上。**

6.3.6　SACM_DVR

SACM_DVR **具有录音和放音功能，采用** A2000 **算法，录音时采用** 16 kHz **数据率及** 8 kHz **采样率获取语音资源，经过** A2000 **压缩后存储在扩展的** SRAM628128A **中，录音完成后自动放音。由于篇幅所限这里只介绍相关** API **函数。**

```
int SACM_DVR_Initial(int Init_Index)          //初始化
void SACM_DVR_ServiceLoop(void)               //获取数据填如译码队列
void SACM_DVR_Encode(void)                    //录音
SACM_DVR_StopEncoder()                        //停止编码
SACM_DVR_InitEncoder(RceMonitorOn)            //初始化解码器
void SACM_DVR_Stop(void)                      //停止录音
void SACM_DVR_Play(void)                      //开始播放
unsigned int SACM_DVR _Status(void)           //获取 SACM_DVR 模块的状态
void SACM_DVR _InitDecode(void)               //开始译码
void SACM_DVR _Decode(void)                   //获取语音数据并译码中断播放
```

SACM_DVR_StopDecoder() //停止解码
unsigned int SACM_DVR _ TestQueue(void) //获取语音队列状态
int SACM_DVR _Fetchqueue(void) //获取录音编码数据
void SACM_DVR_FillQueue(unsigned int encoded-data)
//填充数据到语音队列等待播放
int GetResource(long Address) (Manual)
//从资源文件里获取一个字型语音数据

中断服务函数：

Call F_FIQ_Service_ SACM_DVR (playing)
Call F_IRQ1_Service_ SACM_DVR (recode)

6.4 语音压缩方法

常用的 SACM_A2000 **和** SACM_S480 **两种放音算法要涉及到语音资源的添加问题，即将我们所需要的** WAVE **文件按照需要的压缩比进行压缩，变成资源表形式在程序中调用。这里介绍** DOS **和** Windows **环境下的语音压缩方法，建议使用** Windows **环境压缩。**

6.4.1 DOS 环境下压缩

1. A2000

A2000 **的特点如下：**

(1) **采用** 8 KB 16 **位单声道(用** PC **机的录音功能)录制一个** .wav **文件。**

(2) **用** A2000 **压缩生成** 16 KB(**或** 20 KB, 24 KB)**压缩率的文件。**

(3) **在** MS-DOS **环境下：**

e:\>sacm2000.exe 16 *.wav *.out *.16k

或 (e:\>sacm2000.exe 20 *.wav *.out *.20k

e:\>sacm2000.exe 24 *.wav *.out *.24k)

2. S480

S480 **的特点如下：**

(1) **采用** 8 KB 16 **位单声道录制一个**.wav **文件。**

(2) **用** s480 **压缩生成** 4.8 KB **或** 7.2 KB **压缩率的文件。**

(3) **在** MS-DOS **环境下：**

e:\>sacm.exe *.wav *.48k *.out–s48

或 e:\>sacm.exe *.wav *.72k *.out–s72

6.4.2 Windows 环境下压缩

图 6.11 **是用于压缩的** Windows **工具，可以压缩一个或多个**.wav **文件，具体步骤根据提示操作即可。**

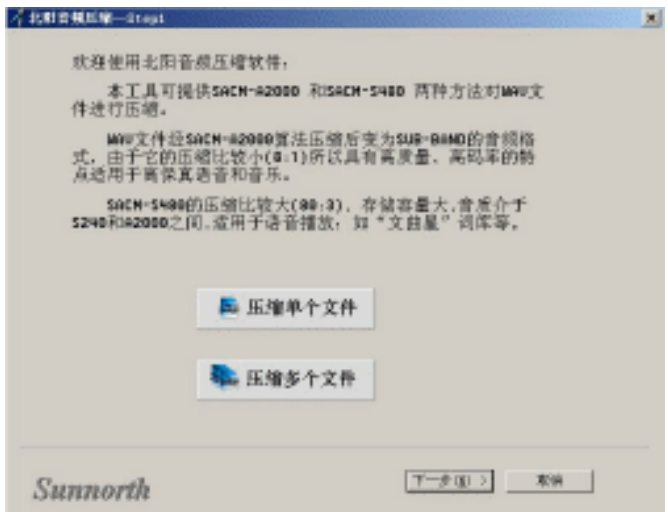


图 6.11　Windows 环境下压缩

6.5　键控放音程序

程序模块中的文件大致分为两类，一类是程序，另一类则是程序接口。程序接口是针对高一级的程序模块而言的，通过接口高一级程序模块可以调用本级程序模块中的子程序或函数，或者使用本级模块中定义的全局变量，这无疑会大大增加软件的可维护性，程序既可以用 C 语言编写也可用汇编语言编写。随着对语音编程越来越熟悉，可以考虑加入一些模块化程序，如键盘，这样可以用按键控制语音播放、停止、暂停、恢复以及音量的大小等。

SPCE061A 提供了一个资源使用模块(即接口文件 Key.inc)。其中，定义了供系统级调用与键扫描相关的一些子程序，如键扫描初始化子程序、键扫描子程序、键扫描防抖动处理子程序以及获得键码子程序等。模块中的 Key.asm 文件是上述接口定义的各子程序，以及定义程序中所需要的全局或局部变量，具体结构如图 6.12 所示。

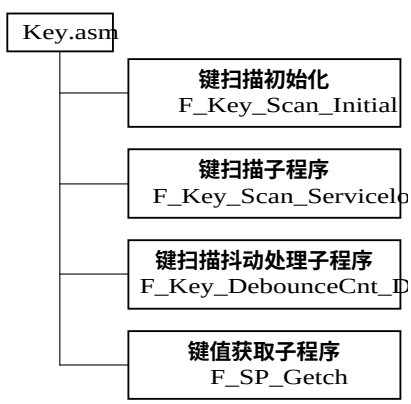


图 6.12　Key.asm 程序结构

6.5.1 常用的键盘 API 函数

1. API 函数

常用的键盘 API **函数如下：**

```
Call F_ Key_Scan_Initial                    //键初始化
Call F_Key_Scan_ServiceLoop                 //键盘扫描
Call F_Key_DebounceCnt_Down                 //键盘防抖动处理
Call F_SP_GetCh()                           //获取键值
```

2. **功能介绍**

下面对常用的键盘 API **函数作具体介绍。**

(1) C：void Key_Scan_Initial void
ASM：Call F_ Key_Scan_Initial
功能：键盘扫描初始化。
参数：无。
返回值：无。

(2) C：void Key_Scan_ServiceLoop void
ASM：Call F_Key_Scan_ServiceLoop
功能：键盘扫描服务循环。
参数：无。
返回值：无。

(3) C：void Key_DebounceCnt_Down void
ASM：Call F_Key_DebounceCnt_Down
功能：键盘扫描过程中消抖动处理。
参数：无。
返回值：无。

(4) C：unsigned int SP_GetCh void
ASM：Call F_SP_GetCh()
[**返回值**]=R1
功能：从扫描缓冲区内获得键值，并将缓冲区清零。
参数：无。
返回值：无。

6.5.2 系统资源模块

SPCE061A **单片机提供了一个系统资源模块，它是建立在上述诸多模块之上的一个模块，其中许多子程序都调用了在它之下模块中的子程序，为了让上一级的程序调用本模块** System.asm **文件中的子程序，设有一个接口文件** System.inc**，一般只需要三条调用语句。当然也可以添加一些模块，具体实现可参考如图** 6.13 **所示的模块框图。**

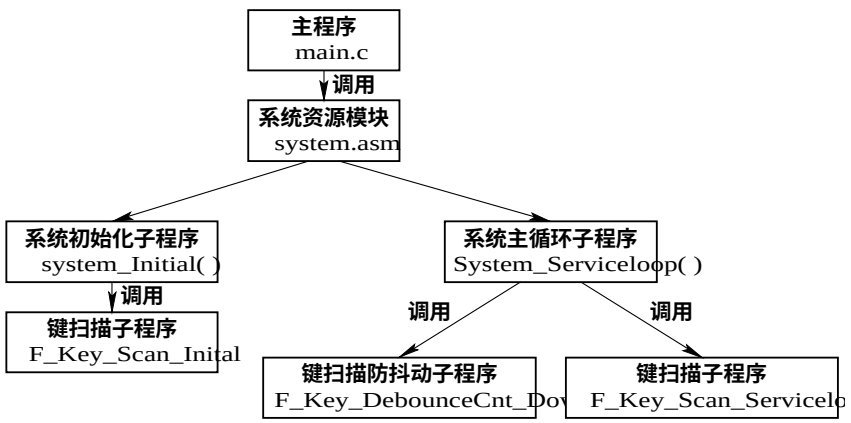


图 6.13　System.asm **模块框图**

6.5.3　应用实例

例 7　**通过一个** A2000 **自动按键控制放音的例子来熟悉一下整个程序的执行过程。**

```
******** 主程序(main.c) **********
main()
{
        int Key = 0;                                    //初始化键值
        int SpeechIndex = 0;                            //初始化语音目录索引号
        int VolumeIndex = 7;                            //初始化音量
            Ret = System_Initial();
            Ret = SACM_A2000_Initial(Auto);
            SACM_A2000_Play(SpeechIndex,DAC1+DAC2,Ramp_UpDn_On); //播放
while(1)
    {
        Key = SP_GetCh();
        switch(Key)
      {
        case 0x00:
            break;
        case 0x01:
            SACM_A2000_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On); //播放
            break;
        case 0x02:
            SACM_A2000_Stop();                          //停止放音
            break;
        case 0x04:
```

```
                SACM_A2000_Pause();                          //暂停放音
                break;
            case  0x08:
                SACM_A2000_Resume();                         //暂停后的恢复
                break;
            case  0x10:
                VolumeIndex++;
                if(VolumeIndex > MaxVolume)
                VolumeIndex = MaxVolume;
                SACM_A2000_Volume(VolumeIndex);              //音量增加
                break;
            case  0x20:
                if(VolumeIndex == 0)
                VolumeIndex = 0;
                else
                VolumeIndex--;
                SACM_A2000_Volume(VolumeIndex);              //音量减
                break;
            case  0x40:
                SpeechIndex++;                               //播放下一首
                if(SpeechIndex == MaxSpeechNum)
                SpeechIndex = 0;
                SACM_A2000_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
                break;
            case  0x80:
                if(SpeechIndex == 0)                         //播放前一首
                SpeechIndex = MaxSpeechNum;
                SpeechIndex--;
                SACM_A2000_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
                break;
            default:
                break;
            }
                System_ServiceLoop();                        //调用系统初始化
                SACM_A2000_ServiceLoop();
                                    //获取 A2000 数据并填入译码队列等待播放
        }
    }
```

```
********** 系统子程序 System.asm **********
    .PUBLIC _System_Initial
    .PUBLIC F_System_Initial
    _System_Initial: .PROC
    F_System_Initial:
        CALL F_Key_Scan_Initial              //键盘扫描
                                             //可以添加一些语音类型子程序或初始化内容
        RETF
    .ENDP
    .PUBLIC _System_ServiceLoop
    .PUBLIC F_System_ServiceLoop
    _System_ServiceLoop: .PROC
    F_System_ServiceLoop:
        CALL F_Key_DebounceCnt_Down          //调用键扫描防抖动处理子程序
        CALL F_Key_Scan_ServiceLoop          //调用键扫描子程序
                                             //可添加其它服务程序，完成相应功能
        RETF
    .ENDP

//键盘子程序 Key.asm
    .RAM
    .VAR R_DebounceReg
    .define C_DebounceCnt 0x0002
    . VAR R_DebounceCnt
    . VAR R_KeyBuf
    . VAR R_KeyStrobe
    .CODE

//键扫描初始化
    F_Key_Scan_Initial:
        r1 = 0x0000
        [R_DebounceReg] = r1                 //初始化
        [R_KeyBuf] = r1
        [R_KeyStrobe] = r1
        r1 = C_DebounceCnt
        [R_DebounceCnt] = r1                 //设定记数初值
        RETF
```

```
*** 键扫描 ************************************
    F_Key_Scan_ServiceLoop:
        r1 = [P_IOA_Data]                  //由 IOA 口获取键值
        r1 = r1 AND 0xFF                   //保留键值
        r2 = [R_DebounceReg]               //将上次获取的键值送给 r2
        [R_DebounceReg] = r1               //将当前键值送给 R_DebounceReg
        CMP r2,[R_DebounceReg]             //比较两次采样的键值是否相同
        JE L_KS_StableTwoSample            //是,则转
        r1 = C_DebounceCnt                 //否则设定记数的时间
        [R_DebounceCnt] = r1
        RETF
    L_KS_StableTwoSample:
        r1 = [R_DebounceCnt]               //判断记数值是否为 0
        JZ L_KS_StableOverDebounce         //是，则转
        RETF
    L_KS_StableOverDebounce:
        r2 = [R_DebounceReg]               //将当前采样的键值送给 r2
        r1 = [R_KeyBuf]                    //暂存上次的键值到 r1
        [R_KeyBuf] = r2                    //当前键值送给暂存寄存器
        r1 = r1 XOR 0x00FF                 //r1 低 8 位取反
        r1 = r1 AND [R_KeyBuf]             //r1 低 8 位取反后和当前的键值相与
        r1 = r1 AND 0x00FF                 //保留与的结果即最后确定的键值
        r1 = r1 OR [R_KeyStrobe]
        [R_KeyStrobe] = r1                 //送键值到键存储单元
        RETF

//键扫描防抖动处理
    F_Key_DebounceCnt_Down:
        r1 = [R_DebounceCnt]               //读计数单元的值
        JZ L_DebounceCntZero               //如果计数单元为零则停止记数
        r1 -= 0x0001
        [R_DebounceCnt] = r1
        L_DebounceCntZero:
        RETF

//从扫描缓冲单元获取键值并清零
    _SP_GetCh:
```

```
F_SP_GetCh:
    r1 = [R_KeyStrobe]                    //取键值
    r2 = 0x0000                           //清键存储单元
    [R_KeyStrobe] = r2
    RETF
```

6.6　语音识别技术

前面我们介绍过语音识别的一些相关内容，这里我们讨论 SPCE061A 特定语者识别(SD，Speaker Dependent)的一个例子，供有兴趣者参考。SD 即语音样板，由单个人训练，也只能识别训练人的语音命令，而他人的命令识别率较低或几乎不能识别。语音识别框图如图 6.14 所示。

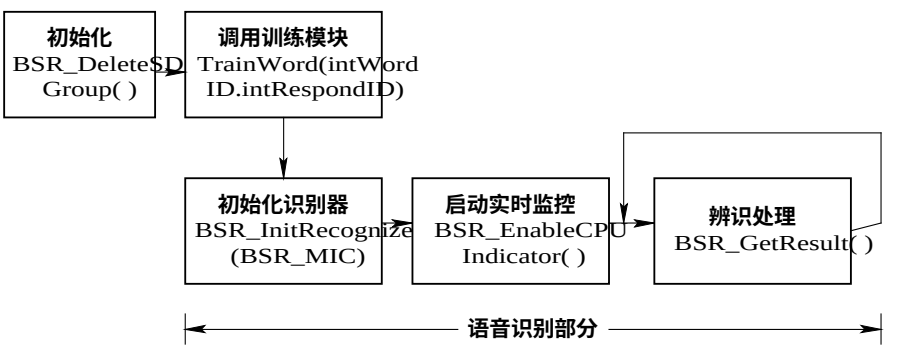


图 6.14　语音识别整体框图

语音识别也将其一些功能做成模块，并通过 API 调用来实现这些功能。

6.6.1　设置

1. 初始化

初始化的设置如下：

(1) API 格式：

C： int BSR_DeleteSDGroup(0)

ASM： F_BSR_DeleteSDGroup(0)

(2) 功能：SRAM 初始化。

(3) 参数：识别的一个标识符，0 代表选择 SRAM，并初始化。

(4) 返回值：当 SRAM 擦除成功返回 0，否则返回 1。

2. 训练部分

训练部分的设置如下：

(1) API 格式：

C：int BSR_Train (int CommandID, int TraindMode)

ASM： F_BSR_Train

(2) 功能：训练函数。

(3) 参数：CommandID 命令序号范围从 0x100 到 0x105，并且对于每组训练语句都是惟一的。

TraindMode 训练次数要求使用者在应用之前，训练一遍或两遍用以下函数区别：

BSR_TRAIN_ONCE: 要求训练一次。

BSR_TRAIN_TWICE：要求训练两次。

(4) 返回值：训练成功返回 0，没有声音返回 1，训练需要更多的语音数据来训练返回 2，环境太吵时返回 3，当数据库满返回 4，当两次输入命令不同返回 5，当序号超出范围返回 6。

注意：

① 在调用训练程序之前要确保识别器正确的初始化。

② 训练次数是 2 时，两次会有一定差异，所以一定要保证两次训练结果接近。

③ 为了增强可靠性，最好训练两次。否则识别的命令就会倾向于噪音。

④ 调用函数后等待 2 s 开始训练，每条命令只有 1.3 s，也就是说当训练命令超出 1.3 s 时，只有前 1.3 s 命令有效。

3. 识别部分

识别部分的设置如下：

(1) C：void BSR_InitRecognizer(int AudioSource)

ASM：F_BSR_InitRecognizer

功能：识别器初始化。

参数：定义语音输入来源通过 MIC 语音输入还是 LINE_IN 电压模拟量输入。

返回值：无。

(2) C：int BSR_GetResult()

ASM：F_ BSR_GetResult

［返回值］=R1

功能：识别中获取数据。

参数：无。

返回值：当无命令识别出来时返回 0；识别器停止未初始化或识别未激活返回−1；当识别不合格时返回−2；当识别出来时返回命令的序号。

注意：该函数用于启动识别 BSR_GetResult()。

(3) C：void BSR_StopRecognizer(void)

ASM：F_ BSR_StopRecognizer

功能：说明停止识别。

参数：无。

返回值：无。

注意：该函数是用于停止识别，当调用此函数时 FIQ_TMA **中断将关闭。**

(4) ASM：_BSR_InitRecognizer

功能：在中断中调用并通过中断将语音信号送 DAC **通道播放。**

参数：无。

返回值：无。

注意：

(1) **该函数在中断** FIQ_TMA **中调用。**

(2) **当主程序调用** BSR_InitRecognizer **时，识别器便打开** 8 kHz **采样率的** FIQ_TMA **中断，并开始将采样的语音数据填入识别器的数据队列中。**

(3) **应用程序需要设置以下程序段在** FIQ_TMA **中。**

程序如下：

```
.PUBLIC _FIQ
.EXTERNAL _BSR_FIQ_Routine                    //定义全局变量
.TEXT
_FIQ:
        PUSH R1,R4 TO [SP]                    //寄存器入栈保护
        R1 = [P_INT_Ctrl]
        Call _BSR_FIQ_Routine                 //调用子程序
        R1 = 0x2000                           //清中断标志位
        [P_INT_Clear] = R1
        POP R1,R4 FROM [SP]                   //寄存器组出栈
        RETI
END
```

其中，实时监控是用来观察识别是否正常工作。如果识别正常，则会产生一周期为 32 ms **的稳定方波信号，如图** 6.15 **所示，否则如果** CPU **超载，则会产生不稳定的信号，如图** 6.16 **所示。此时，需要删除命令或是优化程序，否则会丢失语音数据，产生识别错误的信息。**

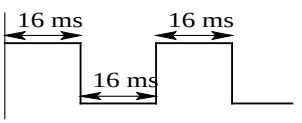


图 6.15　**识别正常波形**

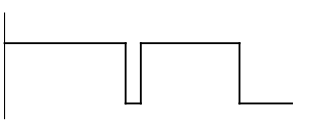


图 6.16　**识别不正常波形**

6.6.2　应用实例

以下是特定人识别的一个范例，在程序中通过三条语句的训练，演示特定人连续音识别，其中第一条语句为触发名称，另外两条为命令，训练完毕开始识别。当识别出触发名称后，开始发布命令则会听到自己设置的应答。

主程序流程图如图 6.17 **所示，中断流程图如图** 6.18 **所示。**

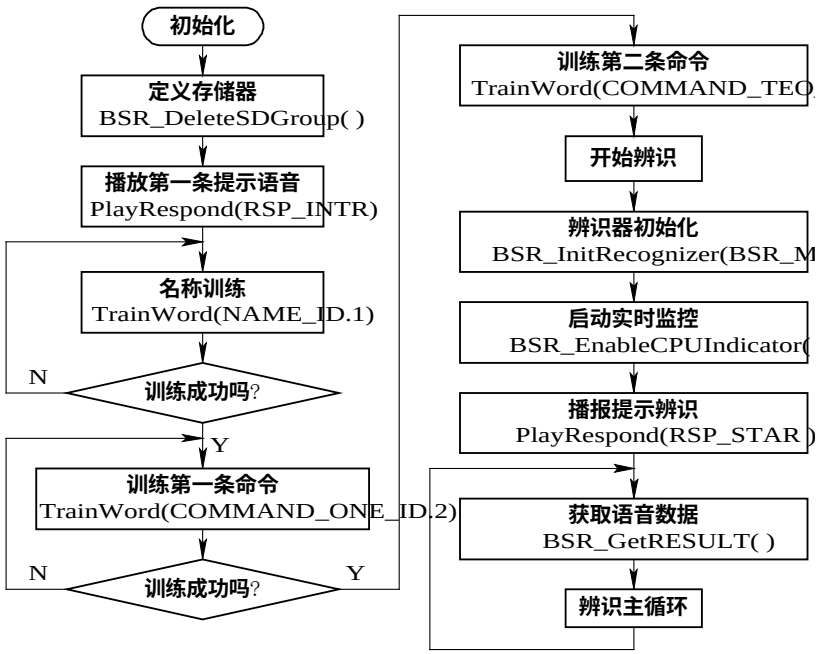


图 6.17　主程序流程图

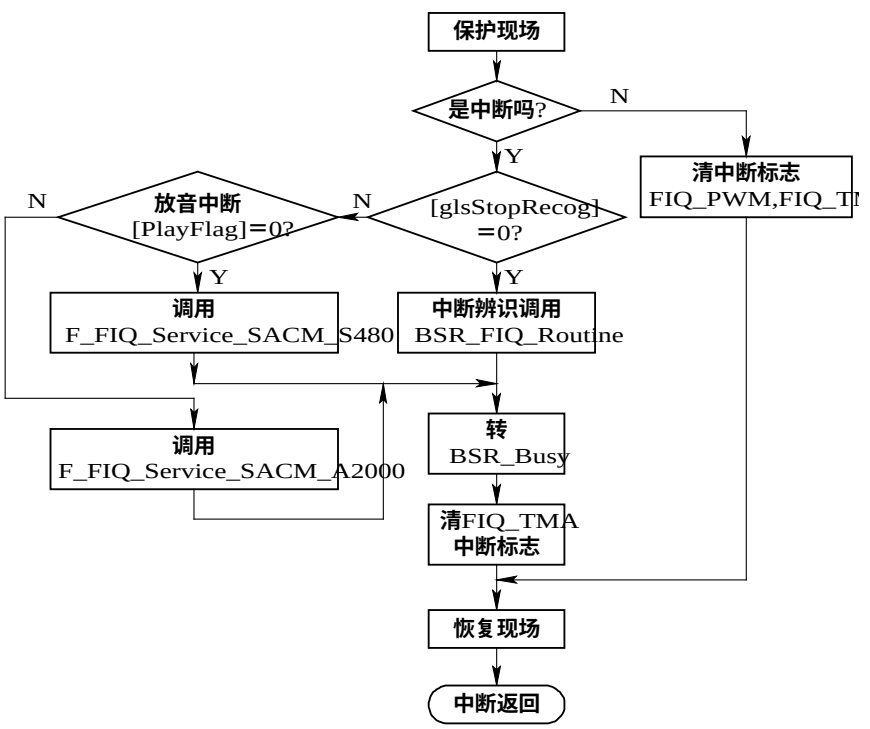


图 6.18　中断服务程序流程图

********** 训练 **********

提示音　　输入语音

"请输入触发名称"　　"警卫"

"请输入第一条命令"　　"开枪"

"请输入第二条命令"　　"你在干什么"

"请再说一遍" (以上提示音每说完一遍出现此命令)

"没有听到任何声音" (当没有检测到声音时出现此命令)

"两次输入名称不相同" (当两次输入的名称不同时出现此命令)

"两次输入命令不相同" (当两次输入的命令有差异时出现此命令)

"准备就绪请开始识别" (以上三条语句全部训练成功时进入识别)

**********识别**********

发布命令　　应答

"警卫"　　"在"、"长官"

"开枪"　　"枪声"

"你在干什么"　　"我在巡逻"、"我在休息"、"我在等人"

注意：在每次提示音结束后 2 或 3 s 再输入命令或当上次应答结束 2 或 3 s 后再发布命令。

```
#include "bsrsd.h"
#define NAME_ID          0x100
#define COMMAND_ONE_ID   0x101
#define COMMAND_TWO_ID   0x102
#define RSP_INTR         0
#define RSP_NAME         1
#define RSP_FIRE         2
#define RSP_GUARD        3
#define RSP_AGAIN        4
#define RSP_NOVOICE      5
#define RSP_NAMEDIFF     6
#define RSP_CMDDIFF      7
#define RSP_STAR         8
#define RSP_MASTER       9
#define RSP_HERE         10
#define RSP_GUNSHOT      0
#define RSP_PATROL       11
#define RSP_READY        12
#define RSP_COPY         13
#define RSP_NOISY        14

//…… 全程变量 ……
```

```
                int gActivated = 0;
    //该变量用于检测是否有触发命令，当有识别出语句为触发名称时，则该位置 1
int gTriggerRespond[] = {RSP_MASTER, RSP_HERE, RSP_MASTER}; //第一条命令应答
            int gComm2Respond[] = {RSP_PATROL, RSP_READY, RSP_COPY};
                                                        //第二条命令应答
            extern void ClearWatchDog();
            int PlayFlag = 0;
            void PlayRespond2(int Result)               //枪声放音子程序
    {
            BSR_StopRecognizer();
            SACM_A2000_Initial(1);
            SACM_A2000_Play(Result, 3, 3);
            while((SACM_A2000_Status()&0x0001) != 0)
        {
            SACM_A2000_ServiceLoop();
            ClearWatchDog();
        }
            SACM_A2000_Stop();
            BSR_InitRecognizer(BSR_MIC);
            BSR_EnableCPUIndicator();
        }
            void PlayRespond(int Result);               //放音子程序
        {
            BSR_StopRecognizer();
            SACM_S480_Initial(1);
            SACM_S480_Play(Result, 3, 3);
            while((SACM_S480_Status()&0x0001) != 0)
        {
            SACM_S480_ServiceLoop();
            ClearWatchDog();
        }
            SACM_S480_Stop();
            BSR_InitRecognizer(BSR_MIC);
            BSR_EnableCPUIndicator();                   //启动实时监控
        }
            int TrainWord(int WordID, int RespondID)    //命令训练
    {
        int res;
            PlayRespond(RespondID);
            while(1)
        {
```

```
            res = BSR_Train(WordID,BSR_TRAIN_TWICE);
            if(res == 0) break;
            switch(res)
        {
        case −1:                                    //没有检测出声音
            PlayRespond(RSP_NOVOICE);
            return −1;
        case −2:                                    //需要重新训练一遍
            PlayRespond(RSP_AGAIN);
            break;
        case −3:                                    //环境太吵
            PlayRespond(RSP_NOISY);
            return −1;
        case −4:                                    //存储器满
            return −1;
        case −5:                                    //检测出声音不同
            if(WordID == NAME_ID)
            PlayRespond(RSP_NAMEDIFF);              //两次输入名称不同
        else
            PlayRespond(RSP_CMDDIFF);               //两次输入命令不同
            return −1;
        case −6:                                    //序号错误
        return −1;
        }
}
        return 0;
}
int main()
{
            int res, timeCnt=0, random_no=0;
            BSR_DeleteSDGroup(0);                   //初始化存储器为 RAM
            PlayRespond(RSP_INTR);                  //播放开始训练的提示音
                                    //.....训练名称.....
            while(TrainWord(NAME_ID,1) != 0) ;
                                    //.....训练第一条命令.....
            while(TrainWord(COMMAND_ONE_ID,2) != 0) ;
                                    //.....训练第二条命令.....
            while(TrainWord(COMMAND_TWO_ID,3) != 0) ;
                                    //.....开始识别命令.....
            BSR_InitRecognizer(BSR_MIC);            //识别器初始化
            BSR_EnableCPUIndicator();
```

```
    PlayRespond(RSP_STAR);                          //播放开始识别的提示音
while(1)
{
    random_no ++;
    if(random_no >= 3) random_no = 0;
    res = BSR_GetResult();
    if(res > 0)                                     //识别出命令
    {
        if(gActivated)
        {
            timeCnt = 0;
            switch(res)
            {
                case NAME_ID:
                PlayRespond(gTriggerRespond[random_no]);
                break;
            case COMMAND_ONE_ID:
                PlayFlag = 1;
                PlayRespond2(RSP_GUNSHOT);
                PlayFlag = 0;
                gActivated = 0;
                break;
            case COMMAND_TWO_ID:
                PlayRespond(gComm2Respond[random_no]);
                gActivated = 0;
            }
        }
            else
        {
                if(res == NAME_ID)
            {
                PlayRespond(gTriggerRespond[random_no]);
                gActivated = 1;
                timeCnt = 0;
            }
        }
    }
        else if (gActivated)
    {
        if (++timeCnt > 450)                        //超出时限
        {
```

```
                    PlayRespond(RSP_NOVOICE);    //在设定时间内没有检测出声音
                    gActivated = 0;
                    timeCnt = 0;
                }
            }
        }
}
```

中断服务程序：

```
.PUBLIC _FIQ
.EXTERNAL _BSR_FIQ_Routine
.EXTERNAL __gIsStopRecog                    //变量值为 0  识别器忙，为 1 识别器停止
.PUBLIC _BREAK,_IRQ0, _IRQ1, _IRQ2, _IRQ3, _IRQ4, _IRQ5, _IRQ6, _IRQ7
.EXTERNAL _PlayFlag
.INCLUDE s480.inc
.INCLUDE A2000.inc
.INCLUDE resource.inc
.INCLUDE hardware.inc
.TEXT
_FIQ:
        PUSH R1,R4 TO [SP]
        R1= [P_INT_Ctrl]
        R1 &= 0x2000
        JZ ?notTimerA                       //当不为 TIQ_TMA 时则转
        R1=[__gIsStopRecog]
        JNZ ?BSR_NotBusy                    //若录音停止则转至放音处理
        CALL _BSR_FIQ_Routine               //调用识别子程序
        JMP ?BSR_Busy                       //返回中断
    ?BSR_NotBusy:                           //放音处理
        R2 = [_PlayFlag]
        JNZ ?Play2000                       //播放标志为 1 则是播放 SACM 2000
        CALL F_FIQ_Service_SACM_S480        //播放标志为 0 播放 SACM 480
        JMP ?BSR_Busy                       //返回中断
    ?Play2000:                              //SACM 2000 播放子程序
        CALL F_FIQ_Service_SACM_A2000
    ?BSR_Busy:                                  //中断返回
        R1 = 0x2000
        [P_INT_Clear] = R1
        POP R1,R4 FROM[SP]
        RETI
    ?notTimerA:
        R1 = 0x8800
```

```
[P_INT_Clear] = R1
POP R1,R4 FROM [SP]
RETI
.END
```

表 6.4 为 API 函数一览表。

表 6.4 API 函数使用寄存器一览表

函数	使用寄存器
SACM_A2000_Initial(int Init_Index) SACM_S480_Initial(int Init_Index) SACM_S240_Initial(int Init_Index) SACM_MS01_Initial(int Init_Index) SACM_DVR_Initial(int Init_Index)	[P_SystemClock] [P_TimerA_Ctrl] [P_TimerA_Data] [P_DAC_Ctrl] [P_INT_Clear] [P_TimerB_Ctrl] [P_TimerB_Data]
SACM_ A2000_Play() SACM_ A2000_InitDecoder()[Manual Mode] SACM_ S480_Play() SACM_ S240_Play() SACM_ MS01_Play() SACM_ DVR_Play() SACM_ DVR_InitDecoder()[Manual Mode]	[P_INT_Clear] [P_TimerA_Data]
SACM_ DVR_Record() SACM_ DVR_InitDecoder()	[P_ADC_Ctrl] [P_TimerA_Data] [P_INT_Ctrl]
SACM_ A2000_Stop() SACM_ A2000_Stop Decoder() SACM_ S480_Stop() SACM_ S240_Stop() SACM_ MS01_Stop() SACM_ DVR_Stop() SACM_ Stop Decoder() SACM_ StopEncoder()	[P_INT_Ctrl] [P_INT_Clear] [P_ADC_Ctrl]
SACM_ A2000_ServiceLoop() SACM_ A2000_ Decoder()[Manual Mode] SACM_ S480_ServiceLoop() SACM_ S240_ServiceLoop() SACM_ MS01_ServiceLoop() SACM_ DVR_ServiceLoop() SACM_ DVR _ Encoder()[Manual Mode]	[P_INT_Ctrl] [P_INT_Clear]
F_FIQ_Service_ SACM_A2000 F_FIQ_Service_ SACM_S480 F_FIQ_Service_ SACM_S240 F_FIQ_Service_ SACM_MS01 F_FIQ_Service_ SACM_DVR	[P_DAC1] [P_DAC2]
F_IRQ1_Service_ SACM_DVR	[P_ADC]

第 7 章　实用 C 语言程序设计

本章主要针对 SPCE061A 单片机编程，介绍常用的 C 语言函数并举例说明其在程序设计中的应用。

7.1　基本数据类型

1. 数据与数据类型

在表 7.1 中列出了 μ'nSP™ GCC 认可的基本数据类型及其值域。请读者注意，表中所列的数据类型及其值域与一般机器使用的 GCC 数据类型之间有一些差别，比如 char 为 16 位等。此外，μ'nSP™ GCC 的 float 与 double 均存储为 32 位浮点数，而 μ'nSP™汇编器的 float 与 double 分别是 32 位与 64 位浮点数。

表 7.1　μ'nSP™ GCC 的基本数据类型

数据类型	数据长度(位数)	值域
char	16	−32 768～32 767
short	16	−32 768～32 767
int	16	−32 768～32 767
long int	32	−2 147 483 648～2 147 483 647
unsigned char	16	0～65 535
unsigned short	16	0～65 535
unsigned int	16	0～65 535
unsigned long int	32	0～4 294 967 295
float	32	以 IEEE 格式表示的 32 位浮点数
double	32	以 IEEE 格式表示的 32 位浮点数

2. 常量、变量、运算符与表达式

在程序运行过程中，其值不能改变的量称为常量。编程时可以用一个字符常量来代表数字，比如：

```
#define    C_Fosc_49M        0x0080
```

这里定义了一个字符常量 C_Fosc_49M，它的值为 0x0080。编程时，我们就可以用 C_Fosc_49M 来代替 0x0080。这样的好处是显而易见的，大家一眼就可以看出 Fosc 选择了 49 MHz。

在程序运行过程中，其值可以改变的量称为变量。下面就有符号和无符号(signed/unsigned)变量作一说明。编程时，如果使用有符号和无符号两种数据类型，就得使用这两种格式的库函数，这将占用较大的存储空间。因此在编程时，如果只强调程序的运算速度而又不进行负数运算时，最好采用无符号格式。

μ'nSP™ GCC 基本的算术运算符和 ANSI-C 是一样的，如表 7.2 所示。

表 7.2 μ'nSP™ GCC 基本的算术运算符

算术逻辑操作符	功能
+、−、*、/、%	加、减、乘、除、求余运算
&&、‖	逻辑与、或
&、|、^、<<、>>	按位与、或、异或、左移、右移
>，>=，<，<=，==，!=	大于、不小于、小于、不大于、等于、不等于
=	赋值运算符
?:	条件运算符
,	逗号运算符
*、&	指针运算符
.	分量运算符
sizeof	求字节数运算符
[ ]	下标运算符

7.2 流程控制语句

归纳起来，C 语言有三种基本结构：顺序结构、选择结构和循环结构。下面我们来具体介绍这三种结构。

1. 顺序结构

顺序结构是一种最基本、最简单的编程结构。在这种结构中，程序由低地址向高地址顺序执行指令程序，如图 7.1 所示。

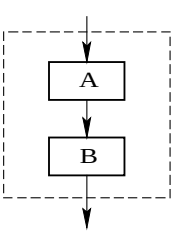


图 7.1 顺序结构示意图

2. 选择结构

在选择结构中，程序首先对一个条件语句进行测试。当条件为真时，执行一个方向的

程序，当条件为假时，执行另一个方向的程序，如图 7.2 所示。T 代表条件，当 T 条件成立时，执行 A 操作，否则执行 B 操作。常见的选择语句有 if，else if，switch/case 语句。

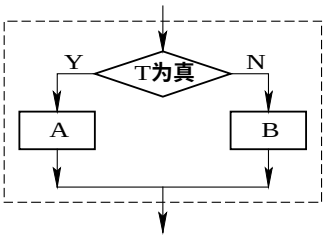


图 7.2　选择结构流程图

1) if 语句

if 语句表达式的格式有下面三种：

(1) if(表达式)语句

语义：如果表达式的值为真，则执行其后的语句，否则不执行该语句。

(2) if(表达式)

　　语句 1；

　else

　　语句 2；

语义：如果表达式的值为真，则执行语句 1，否则执行语句 2。

(3) if-else-if 形式

前两种形式的 if 语句一般都用于两个分支的情况，当有多个分支选择时，可采用 if-else-if 语句，其格式为：

```
if(表达式 1)
    语句 1；
else if(表达式 2)
    语句 2；
 ⋮
else if(表达式 n)
    语句 n；
else
    语句 n+1；
```

语义：依次判断表达式的值，当出现某个值为真时，则执行其对应的语句，然后跳到整个 if 语句之外继续执行程序。如果所有的表达式均为假，则执行语句 n＋1，然后继续执行程序。

由若干条 if，else if 语句嵌套可构成串行多分支结构。在 if 语句的嵌套中，请注意 if 和 else 的对应关系。else 总是与它上面最近的一个 if 语句相对应，如果 if 和 else 的数目不同，可以用花括号将不对称的 if 括起来，以确定它们之间的相应关系。

2) switch 语句

格式：

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

语义：每个 case **后的常量表达式只能是常量组成的表达式，当** switch **后的表达式的值与某一个常量表达式的值一致时，程序就转向此** case **后的语句执行，遇到** break **就退出** switch **语句。如果没有一个常量表达式的值与** switch **后的值一致，就执行** default **后的语句。**

各个 case **和** default **出现的次序不影响执行结果，一般情况下，尽量将出现机率大的** case **放在前面。**

3. **循环结构**

使用循环结构可以使分支流程重复进行，循环执行某段程序。

循环结构又分为“当”(while)**型循环结构和“直到”**(do while)**型循环结构两种，如图** 7.3 **所示。在“当”**(while)**型循环结构中，当判断条件** T **为真时，反复执行操作** A**，当条件为假时才停止循环。在“直到”**(do while)**型循环结构中，先执行操作** A**，再判断条件** T**，若** T **为真，则再执行** A**，如此反复，直到** T **为假为止。**

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

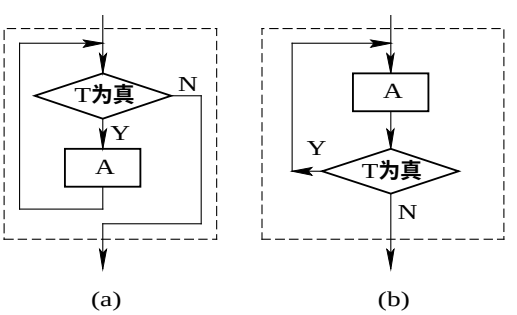


图 7.3 **循环结构示意图**

(a) “while”**型；**(b) “do while”**型**

1) while **语句**

while **语句的格式为**

```
while (表达式) 语句；
```

其中，表达式是循环条件，语句为循环体。

语义：计算表达式的值，当值为真(非 0**)时，执行循环体语句。**

使用 while **语句应注意以下几点：**

(1) while **语句中的表达式一般是关系表达或逻辑表达式，只要表达式的值为真(非** 0**)即可继续循环。**

(2) 循环体如包括有一个以上的语句，则必须用{}括起来，组成复合语句。

(3) 应注意循环条件的选择，以避免出现死循环。

2) do while 语句

格式：

```
do
    语句；
while(表达式)；
```

其中，语句是循环体，表达式是循环条件。

语义：先执行循环体语句一次，再判别表达式的值，若为真(非 0)则继续循环，否则终止循环。

do while 语句的特点是在判断条件是否成立之前，先执行一次循环体语句，这是与 while 语句的一个根本区别。

3) for 语句

for 语句的作用是用来表示循环次数已知的情况，其格式如下：

for(表达式一；表达式二；表达式三) 循环体语句

执行过程如下：

(1) 先求解表达式一。

(2) 求解表达式二，若其值为 0 则结束循环；若其值为非 0 则执行(3)。

(3) 执行循环体语句，可以是一条语句或一个复合语句。

(4) 求解表达式三。

(5) 转到(2)去执行。

说明：

表达式一和表达式三可以省略，看三个例子：

```
for (;i<100;i++)
for(i=1;i<100;)
for (;i<100;)
```

这三例子中，分号不能省略，另外在省略了表达式一和表达式三后，要注意使表达式二能够取到 0 值，以避免形成死循环。

7.3　构造数据类型

前面介绍的字符型(char)、整型(int)和浮点型(float)等数据都属于基本数据类型。C 语言还提供了一些构造数据类型，常见的有数组、指针、联合、枚举等。

1. 数组

数组相当于是由若干类型相同变量组成的一个有前后顺序的集合。利用数组可以通过一个统一的数组名称和一个序号，按照第一个、第二个…… 的方式来访问数组中的数据。

数组有一维数组、多维数组和字符数组三种类型，本书只介绍一维数组。

1) 一维数组的定义

一维数组的格式如下：

类型说明符　数组名[常量表达式];

例如：

int b1[5];

表示定义了一个含有五个整型元素且名称为 b1 的一维数组。

float x[10];

定义了一个含有 10 个单精度型元素且名称为 x 的一维数组。

2) 数组的引用

在已经定义了一个数组以后，怎么来使用数组中的元素呢？C 语言规定只能一个一个地引用数组元素而不能一次引用数组中的全部元素。

在 C 语言中，下标的取值范围是从[0,元素个数减 1]结束。

例如，int b1[3]; 定义了一个含有三个整型元素的数组。我们在引用这三个元素时，下标只能取 0、1、2 三个值。

3) 数组的初始化

在定义了数组后，对计算机来说只相当于定义了一个变量，这个变量的值是不确定的。所以，在使用数组时，最好对它进行初始化操作。

对数组的初始化操作可以采取以下方式：

static 类型说明符　数组名[N]＝｛值 1，值 2，…，值 N｝;

说明：

(1) 对数组的初始化操作只能在定义数组时进行。

(2) 关键字 static 表示定义了一个静态变量。

我们来看一个例子：

Fib(n)=1　　n=0

Fib(n)=1　　n=1

Fib(n−2)+Fib(n−1)　　n>1

```
#define NUM 20
main()
{
  int i;
     static int fib[NUM]={1,1};
           for (i=2;i<NUM;i++)
           fib[i]=fib[i−2]+fib[i−1];
}
```

程序定义了一个有 20 个元素的整型数组，并给前两个元素赋初值：fib[0]=1，f[1]=1。那其它元素的值是什么呢？答案为“0”。用关键字 static 定义的任何变量，如果没有赋初值的话，系统将自动把其初值设为 0。

2. 指针

指针是C语言中广泛使用的一种数据类型，运用指针编程是C语言最主要的风格之一。利用指针变量可以表示各种数据结构，能很方便地使用数组和字符串，并能像汇编语言一样处理内存地址，从而编出精练而高效的程序。

请读者编程时注意，μ'nSP™ GCC 编译器所认可的指针是 16 位的。

1) 指针变量的类型

指针变量的格式如下：

类型说明符 *变量名；

其中，*表示这是一个指针变量，变量名即为定义的指针变量名，类型说明符表示本指针变量所指向变量的数据类型。

例如，int *p1;表示 p1 是一个指针变量，它的值是某个整型变量的地址，或者 p1 指向一个整型变量。至于 p1 究竟指向哪一个整型变量，应由向 p1 赋值的地址来决定。

2) 指针变量的赋值

指针变量同普通变量一样，使用之前不仅要定义说明，而且必须赋予具体的值。指针变量的赋值只能赋予地址，不能赋予任何其它数据，否则将引起错误。

指针变量的赋值运算有以下几种形式：

(1) 指针变量初始化赋值，如：

```
int a;
int p=&a;                /*把整型变量 a 的地址赋予 p*/
```

(2) 把一个变量的地址赋予指向相同数据类型的指针变量，如：

```
int a,*pa;
pa=&a;                   /*把整型变量 a 的地址赋予整型指针变量 pa*/
```

(3) 把一个指针变量的值赋予指向相同类型变量的另一个指针变量，如：

```
int a,*pa,*pb;
pa=&a;
pb=pa;                   /*把 a 的地址赋予指针变量 pb*/
```

由于 pa，pb 均为指向整型变量的指针变量，因此可以相互赋值。

(4) 把数组的首地址赋予指向数组的指针变量，如：

```
int a[5],*pa;
pa=a;
```

说明：数组名表示数组的首地址，故可赋予指向数组的指针变量 pa。

(5) 把字符串的首地址赋予指向字符类型的指针变量，如：

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

这里应说明的是，并不是把整个字符串装入指针变量，而是把存放该字符串字符数组的首地址装入指针变量。

(6) 把函数的入口地址赋予指向函数的指针变量，如：

```
int (*pf)();pf=f;                /*f 为函数名*/
```

3. 结构体和共用体

结构体和共用体是C语言中广泛使用的两种构造数据类型。它们之间的区别在于结构体中各成员有各自的内存空间，一个结构体变量的总长度是各成员长度之和。而共用体中各成员共享一段内存空间，一个共用体变量的长度等于各成员中最长的长度。应该说明的是，这里所谓的共享不是指把多个成员同时装入一个共用体变量内，而是指该共用体变量可被赋予任一成员值，但每次只能赋一种值，赋予新值则替代旧值。

除了上述区别，结构体和共用体在很多地方是相似的。所以本小节只介绍结构体。

结构是一种“构造”而成的数据类型，在说明和使用之前必须先定义它，也就是构造它，如同在说明和调用函数之前要先定义函数一样。

1) 定义结构的类型

定义结构的格式如下：

```
struct 结构名
{
    成员表列
};
```

成员表由若干个成员组成，每个成员都是该结构的一个组成部分，对每个成员也必须作类型说明，如：

```
struct date
{
    unsigned int month;
    unsigned int day；
    unsigned int year;
};
```

在这个结构定义中，结构名为 date，该结构包含由三个成员：month，day 和 year。这三个结构成员的数据类型都是整型变量。应注意，在括号后的分号是不可少的。结构定义之后，即可进行变量说明。凡说明为结构 date 的变量都由上述三个成员组成。由此可见，结构是一种复杂的数据类型，是数目固定、类型不同的若干有序变量的集合。

2) 定义结构类型变量

定义一个结构变量有三种方法，限于篇幅，这里仅介绍一种，以上面定义的 date 为例加以说明。

先定义结构，再说明结构变量，如：

```
struct date
{
    unsigned int month;
    unsigned int day；
    unsigned int year;
};
    date date1，date2；
```

本例说明了两个变量 date 1 和 date 2 为 date 结构类型。

如果结构变量是全局变量或为静态变量，则可对它作初始化赋值。对局部或自动结构变量不能作初始化赋值。

数组的元素也可以是结构类型的，因此可以构成结构型数组。结构数组的每一个元素都是具有相同结构类型的下标结构变量。结构数组的定义方法和结构变量相似，只需说明它为数组类型即可。

结构指针变量说明，当用来指向一个结构变量时可使用一个指针变量，此称之为结构指针变量。结构指针变量中的值是所指向的结构变量的首地址，通过结构指针可访问该结构变量，与数组指针和函数指针的情况是相同的。

结构指针变量说明的格式如下：

struct 结构名 *结构指针变量名

例如，在前面定义了 date 这个结构，如要说明一个指向 date 的指针变量 pdate，可写成：

```
struct date *pdate;
```

3) 结构类型变量的引用

对结构进行引用时，只能对结构体的成员进行赋值、存取和运算。其访问的一般格式如下：

(*结构指针变量).成员名 或为：结构指针变量->成员名

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

4. 函数

在 C 语言中，程序是由一个主函数和其它若干函数构成的，在主函数中可以调用其它函数，其它函数之间也可以相互调用。

1) 函数的功能

函数的功能有：

(1) 一个 C 语言程序(称作源文件)是由一个函数或多个函数组成的。

(2) 对于一个较大的任务，一般将它分解成若干源文件，分别编写和调试，这样可以提高开发效率。一个源文件可以被其它的 C 程序使用。

(3) C 程序必须从 main()函数开始执行。

(4) 所有函数的定义都是互相独立的，即不存在嵌套定义。但是在调用时，可以相互调用或嵌套调用。

(5) C 语言中有两类函数：系统提供的函数(库函数)和用户自定义的函数。

(6) 函数又分为无参函数(只完成固定的功能)和带参函数(这些参数规定了函数要执行什么样的操作)，这两种函数都可以返回或不返回一个函数值。

2) 函数的定义

函数的格式：

类型说明符　函数名(形式参数表列)
形式参数类型说明
{变量定义部分
函数体语句}

说明：

(1) 类型说明符：规定这个函数的返回值类型，我们可以通过对函数返回值的判断，来了解函数的执行情况。

(2) 函数名：函数的名称，通过这个名称才能对某个函数进行调用。

(3) 形式参数说明表列：规定函数有什么样的参数(这部分和下面的形式参数类型说明部分可以省略，省略之后的函数被称作无参函数)，如 x,y,z 等用逗号分隔的变量。这些形式参数的作用是控制函数进行什么样的操作，它们的值由调用这个函数的程序给出。

(4) 形式参数的类型说明：部分规定了形式参数的类型，形参与形参的类型可以放在一起。如可写成这样：tan(int x,int y,int z)。

(5) 变量定义部分：规定在函数内部要用到的变量和它们的类型。

(6) 函数体语句：规定函数中要执行的语句。函数体语句和变量定义部分用一对大括号括起来。

当我们按如下方式定义一个函数时，称此函数为空函数。空函数的格式如下：

类型说明符　函数名() {}

例如“ echoline() {} ”就定义了一个空函数。

3) 形式参数与实际参数

在 C 语言程序中，大部分函数都是带参函数，所以在调用函数与定义函数时存在着数据传递。

在定义函数时函数名后括号内的变量就称为形式参数(形参)，而在调用函数时函数名后括号内的变量被称为实际参数(实参)。

说明：

(1) 注意形参是随函数的调用而产生的，随函数调用的结束而消亡的。

(2) 实参的作用是给对应的形参赋值，所以实参必须要能得到一个值，它可以是常量、变量或表达式。一般情况下实参与形参的类型应一致，这样可避免错误的产生。

(3) 在函数定义中，必须规定形参的类型。

4) 函数的返回值

如果想让函数向调用它的函数带回一个值，那么只能使用 return 语句。return 语句的使用格式如下：

return(表达式);

它将表达式的值作为函数的返回值，结束本次函数调用并回到进行调用函数的语句。

说明：

(1) return 只能返回一个值，而不能返回多个值。

(2) return(表达式)语句中的表达式值的类型应与定义函数时函数的类型一致。如果不一致，以定义函数时规定的函数类型为准进行类型转换。

(3) 如果函数中没有 return 语句，那么返回值是一个不确定的值。

(4) 为了明确规定函数没有返回值，可以用 void 关键字来定义函数，表示“无类型”。

(5) return 表达式也符合使用标准。

5) 函数调用格式

函数的调用格式如下:

函数名（实参表列）

说明：

(1) 调用函数时要给出被调用的函数名和一对括号。如果这个函数是个带参函数，那么还要将实参变量用逗号隔开放到括号中。

(2) 可以将函数调用当成一个表达式。我们可以在任何能够放置表达式的地方来放置函数表达式。

(3) 函数实参的求值顺序在各 C 语言系统中是不一样的。在有的系统中实参的求值顺序是从左到右的，如：

```
i=3;printf("%d,%d",i,i++);
```

其输出为 3,4。

μ'nSP™ IDE 和 Turbo C 一样，实参的求值顺序是从右到左的，如：

```
i=3;printf("%d,%d",i,i++);
```

其输出为 4,3。

6) 函数调用规则

函数调用规则如下：

(1) 被调用的函数必须是已经存在的函数。(库函数或用户已定义过的函数)

(2) 如果使用库函数，还要在使用库函数的源文件开头用＃include 命令，声明库函数所在的头文件。

(3) 如果使用用户自定义的函数，还要在主调函数中说明用户函数的返回值类型。注意，这时我们关心的只是函数的返回值类型，所以对函数的参数就没必要写出来了。

C 语言规定在以下几种情况下，可以不在主调函数中对被调用函数的返回值类型进行说明：

(1) 如果被调用函数的返回值为整型或字符型，可不进行说明，系统将自动将此被调函数的返回值看成整型。

(2) 如果被调用函数定义在主调函数之前，也可不对其进行说明。

(3) 如果在所有函数定义之前，对程序中的函数类型进行了说明，则在各个主调函数中不必再对被调用函数进行说明。

例如：

```
float max();
double min();
void equal();
main()
{
  ⋮
}
```

```
float max(float x,float y)
{
  ⋮
}
double min(double x,double y)
{
  ⋮
}
void echoline()
{
  ⋮
}
```

在这个例子中，我们在程序的开头对程序中的所有函数进行了说明，这样就不用在主调函数中对其它函数进行说明了。建议读者使用这种方法，因为这种方法将所有的函数在程序开头进行了说明，不仅省去了在各函数中进行的函数说明，而且一看就知道这个程序中有哪些函数。

C语言中不允许给嵌套的函数定义，因此各函数之间是平行的，不存在上一级函数和下一级函数的问题。但允许在一个函数的定义中出现对另一个函数的调用，这样就出现了函数的嵌套调用，即在被调函数中又调用其它函数。这与其它语言的子程序嵌套的情形是类似的。

一个函数在它的函数体内调用它自身称为递归调用，这种函数称为递归函数。C语言允许函数的递归调用，在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新一层。例如：

```
int f (int x)
{
    int y;
    z=f(y);
    return z;
}
```

这个函数是一个递归函数，但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无休止地进行，必须在函数内有终止递归调用的手段。常用的办法是加条件判断，满足某种条件后就不再作递归调用，然后逐层返回。

7) C 函数和汇编函数的相互调用

很多系统特别是实时时钟系统都是用 C 和汇编语言联合编程的。μ'nSP™ 单片机的开发仿真环境 IDE 同时提供了 C 语言和汇编语言的开发环境，C 函数和汇编函数可以方便地进行相互调用，如图 7.4 所示。

由于 C 编译器产生的所有标号都以下划线“_”为前缀，因此 C 程序在调用汇编函数时要求汇编函数名也以下划线“_”为前缀。在进行参数传递时，参数以相反的顺序(从右到左)被压入栈中。在汇编函数调用 C 函数时，函数调用者应切记在函数返回时，将调用函数压入栈中的参数弹出。

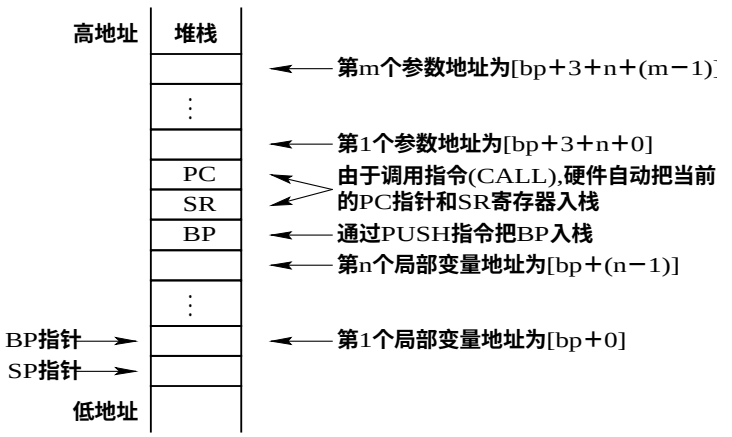


图 7.4　参数传递的堆栈调用

16 位的返回值存放在寄存器 R1 中；32 位的返回值存入寄存器对 R1、R2 中，其中低字在 R1 中，高字在 R2 中。若要返回结构，则需在 R1 中存放一个指向结构的指针。

编译器会产生 prolog/epilog 过程动作来暂存或恢复 PC、SR 及 BP 寄存器。汇编器则通过“CALL”指令可将 PC 和 SR 自动压入栈中，而通过“RETF”或“RETI”指令将其自动弹出堆栈。编译器所认可的指针是 16 位的。函数的指针实际上并非指向函数的入口地址，而是一个段地址向量_function_entry，在该向量里由两个连续 Word 的数据单元存放的值才是函数的入口地址。

下面以具体实例来说明 C 函数和汇编函数的相互调用，其程序对照如表 7.3 所示。

表 7.3　C 语言与汇编语言程序对照表

C 语言程序	汇编语言程序

extern int ADD();
int main(void)
{
　　int i,j,SUM;
　　i=2;
　　j=3;
　　SUM=ADD(i,j);
　　return 0;
}
	.CODE
	.PUBLIC _ADD
	_ADD:.PROC
	　　PUSH BP TO [SP];
	　　BP = SP + 1
	　　R1 = [BP+3]　//取得第二个参数
	　　R2 = [BP+4]　//取得第一个参数
	　　R1 += R2
	　　POP BP FROM [SP]
	　　RETF
	.ENDP

在 C 中要调用一个汇编编写的函数，需要首先在 C 语言中声明此函数的函数原型。尽管不作声明也能通过编译并执行程序，但是会带来很多的潜在的错误。

我们看一个 C 调用汇编的例子，读者可以从这个例子中了解到 C 调用函数时是如何进行参数传递的，函数的返回值是怎样实现的。

主函数 main()为 C 函数，调用汇编函数 ADD()。主函数往 ADD()中传递两个参数，ADD()把这两个参数相加，所得的和返回给主函数。

在汇编函数中要调用 C 语言的子函数，那么应该根据 C 的函数原型所要求的参数类型，分别把参数压入堆栈后，再调用 C 函数。调用结束后还需进行弹栈，以恢复调用 C 函数前的堆栈指针。此过程很容易产生错误，请读者仔细处理。

下面给出汇编调用 C 函数的例子，汇编与 C 语言程序对照如表 7.4 所示。

表 7.4 汇编语言与 C 语言程序对照表

汇编语言程序	C 语言程序

.EXTERNAL _ADD
.RAM
.VAR SUM
.CODE
.PUBLIC _main
_main:
R1 = 3
PUSH R1 TO [SP] //第 2 个参数入栈
R1 = 2
PUSH R1 TO [SP] //第 1 个参数入栈
CALL _ADD
[SUM]=R1
POP R1 FROM [SP] //弹出参数恢复 SP 指针
.END
	int ADD(int i,int j)
	{
	i=i+j;
	return i;
	}

主函数 main()为汇编函数，调用 C 函数 ADD()。主函数往 ADD()中传递两个参数，ADD()把这两个参数相加，所得的和返回给主函数。

7.4 C 语言与汇编相互调用

为了使程序具有更高的效率和更多的功能，需要在 C 语言程序中嵌入用汇编语言编写的子程序。一方面可以提高子程序的执行速度和效率，另一方面可以实现某些用 C 语言无法实现的操作。通常，有两种方法可将汇编语言程序与 C 语言程序结合在一起。一种是把独立的汇编语言程序用 C 函数连接起来，通过 API (Application Program Interface) 的方式调用，这在前面已经介绍了；另一种就是我们下面要介绍的在线汇编方法，即直接插入式汇编指令嵌入到 C 函数中。

采用 GCC 规定的在线汇编指令格式进行汇编指令的输入，就可实现将 μ'nSP™ 汇编指令嵌入到 C 函数中。GCC 在线汇编指令格式规定如下：

ASM (“汇编指令模板”：输出参数：输入参数：clobbers 参数)；

若无 clobber 参数，则在线汇编指令格式可简化为：

ASM (“汇编指令模板”：输出参数：输入参数)；

下面介绍在线汇编指令格式中的各部分的功能。

1. 汇编指令模板

模板是在线汇编指令中的主要成分，GCC 据此可在当前位置产生汇编指令输出。例如：

ASM ("%0 += %1" : "+r" (foo) : "r" (bar))；

此处，“ "%0 += %1" ”就是模板。其中，操作数“ %0 ”、“ %1 ”作为一种形式参数，分别会由第一个冒号后面实际的输出、输入参数取代。带百分号的数字表示的是第一个冒号后参数的序号。

2. 操作数

在线汇编指令格式中，第一个冒号后的参数为输出操作数，第二个冒号后的参数为输入操作数，第三个冒号后跟着的则是 clobber 操作数。在各类操作数中，引号中的字符代表的是其存储类型约束符；括弧里面的字符串表示的是实际操作数。如果输出参数有若干个，可用逗号“ , ”将每个参数隔开。同样，该法则适用于输入参数或 clobber 参数。

3. 操作数约束符

约束符的作用在于指示 GCC，使用在汇编指令模板中的操作数的存储类型。表 7.5 列出了一些约束符和它们分别代表的操作数不同的存储类型，也列出了用在操作数约束符之前的两个约束符前缀。

表 7.5　操作数存储类型约束符及约束符前缀

约束符	操作数存储类型	约束符前缀及含义解释	
r	寄存器中的数值	=	+
m	存储器内的数值	为操作数赋值	操作数在被赋值前先要参加运算
i	立即数		
p	全局变量操作数		

4. 利用嵌入式汇编操作端口寄存器

在 C 的嵌入式汇编中，当使用端口寄存器名称时，需要在 C 文件中加入汇编的包含文件，如：

```
ASM(".include hardware.inc");
```

那么，我们就可以使用端口寄存器的名称，而不必去使用端口的实际地址了。

如果你不想包含那么大一个文件的话，也可以单独定义你用到的一些寄存器，如：

```
ASM(".define    P_IOA_Dir    0x7002");
```

1) 写端口寄存器

例如，要设定 PortA 端口为输出端口，需要对 P_IOA_Dir 赋值 0xFFFF，请用 C 语言中嵌入汇编方式实现。

在 C 中有一个 int 型的变量 i，传送到 P_IOA_Dir 中，则嵌入汇编的实现方式如下：

```
ASM(".define    P_IOA_Dir    0x7002");
int main(void)
{
    int i;
    ASM("[P_IOA_Dir] = %0": :"r"(i));
                                    //只有输入参数，通过寄存器传递变量 i 的内容
}
```

如果需要对端口寄存器直接赋一个立即数(比如对 P_IOA_Dir **赋值** 0x1234)，**那么内嵌式汇编为：**

```
ASM("[P_IOA_Dir] = %0" : : "r"(0x1234));
                                    //只有输入参数，通过寄存器传递立即数 0x1234
```

2) **读端口寄存器**

对端口寄存器进行读操作的方法与写类似，下面仍以 P_IOA_Dir **为例。**

如果要实现把端口寄存器 P_IOA_Dir **的值读出并保存在** C **的一个** int **变量** j **中，那么可以通过下面的方法来实现：**

```
int j;
ASM("%0 =[P_IOA_Dir]":"=r"(j));     //只有输出参数，而无输入参数
```

3) **利用** GCC **编程举例**

下面是一段 GCC **的程序，实现对** A **口的初始化，设定** A **口为同相输出高电平。**

```
ASM("[P_IOA_Attrib] = %0\n\t"
    "[P_IOA_Data] = %0\n\t"
    "[P_IOA_Dir] = %0\n\t"
        ⋮
    "r"(0xFFFF)
    );
```

上面的程序通过 GCC **编译后的程序为**

```
R1=(−1)
[P_IOA_Attrib] = R1
[P_IOA_Data] = R1
[P_IOA_Dir] = R1
```

下面是一段 GCC **的程序，实现对** B **口的初始化，设定** B **口为具有上拉电阻的输入。**

```
ASM("[P_IOB_Attrib] = %0\n\t"
    "[P_IOB_Data] = %1\n\t"
    "[P_IOB_Dir] = %0\n\t"
    ⋮
    "r"(0),
    "r"(0xFFFF)
    );
```

上面的程序通过 GCC **编译后的汇编程序为**

```
R2=0xFFFF
R1=0x0000
[P_IOB_Attrib] = R2
[P_IOB_Data] = R1
[P_IOB_Dir] = R2
```

通过上述两段程序，使得 SPCE061A **的** B **口为输入，**A **口为输出，如果我们要实现把** B **口读入的数据从** A **口输出，**GCC **编程就需要在** C **中先建立个** int **型的中间变量，通过这**

个中间变量，写出两个 GCC 的程序来实现。

```
    ⋮
int temp;
    ⋮
ASM("%0 = [P_IOB_Data]" : "=r"(temp));
ASM("[P_IOA_Buffer] = %0"    : : "r"(temp));
```

通过 GCC 编译后的程序如下(这里将看不到 temp 的影子，GCC 会进行优化处理)：

```
R1 = [P_IOB_Data]
[P_IOA_Buffer] = R1
```

通过上述方法的介绍，我们就可以在 C 语言中直接对 SPCE061A 的硬件进行操作。如果采用 C 调用汇编显得有些不便，而使用嵌入式汇编就会使得程序高效简洁。

7.5　I/O 端口的 C 语言程序设计

SPCE061A 中提供的相关 API 函数如表 7.6 所示。

表 7.6　SPCE061.lib 中提供的相关 API 函数

函数	功能说明	用法	参数	返回
Void Set_IOA_Dir(unsigned int) Void Set_IOB_Dir(unsigned int)	设置 I/O 的 Direction 信息	Set_IOA_Dir(Direction_A) Set_IOB_Dir(Direction_B)	1 代表输出 0 代表输入	无
unsigned int Get_IOA_Dir(void) unsigned int Get_IOB_Dir(void)	获取 I/O 的 Direction 信息	Direction_A＝Get_IOA_Dir() Direction_B＝Get_IOB_Dir()	1 代表输出 0 代表输入	
void Set_IOA_Attrib(unsigned int) void Set_IOB_Attrib(unsigned int)	设置 I/O 的 Attribution 信息	Set_IOA_Attrib Attribution_A) ; Set_IOA_Attrib (Attribution_B)；		
unsigned int Get_IOA_Attrib(void) unsigned int Get_IOB_Attrib(void)	获取 I/O 的 Attribution 信息	Attribution_A ＝Set_IOA_Attrib () Attribution_B ＝Set_IOA_Attrib ()		
Void Set_IOA_Data(unsigned int) void Set_IOB_Data(unsigned int)	设置 I/O 的 Data 信息	Set_IOA_Data(Data_A) Set_IOB_Data(Data_B)	1 代表高电平 0 代表低电平	无
unsigned int Get_IOA_Data(void) unsigned int Get_IOB_Data(void)	获取 I/O 的 Data 信息	Data_A ＝Set_IOA_Data() Data_B ＝Set_IOB_Data()	1 代表高电平 0 代表低电平	无
void Set_IOA_Buffer(unsigned int) void Set_IOB_Buffer(unsigned int)	设置 I/O 的 Buffer 信息	Set_IOA_Buffer(Buffer_A) Set_IOB_Buffer(Buffer_B)	1 代表高电平 0 代表低电平	
unsignedint Get_IOA_Buffer(void) unsigned int Get_IOB_Buffer(void)	获取 I/O 的 Buffer 信息	Buffer_A＝Set_IOA_Buffer() Buffer_B＝Set_IOB_Buffer()	1 代表高电平 0 代表低电平	
void Get_IOA_Latch(void)	读 P_IOA_Latch 单元，以此来锁存 IOA0～IOA7 管脚上的键状态	Get_IOA_Latch()	无	无
void SP_Init_IOA(unsigned int, unsigned int, unsigned int)； void SP_Init_IOB(unsigned int, unsigned int, unsigned int)；	同时设置 I/O 的 Direction、Attribution 和 Data 信息	SP_Init_IOA(Direction_A, Data_A, Attribution_A); SP_Init_IOB(Direction_B, Data_B, Attribution_B);		无

sp_lib.asm 中定义了两个很有用的 I/O 端口 API，在 C 中可以调用。它们是 SP_Init_IOA() 和 SP_Init_IOB()。下面举例说明 I/O 端口中的 C 语言程序设计。

例 1　硬件连接图如图 7.5 所示。A 口低 8 位接 1×8 键盘，B 口低 8 位接八个 LED，要求按下某一按键时对应的 LED 熄灭。

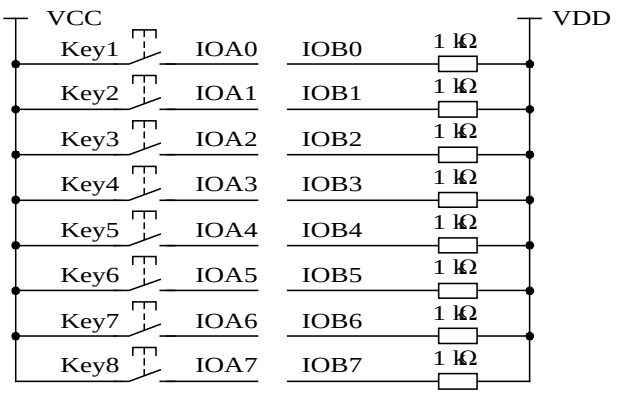


图 7.5　硬件连接图

分析：显然，应该把 A 口的低 8 位设置为输入，B 口的低 8 位设置为输出。从图 7.5 中可以看出，按键按下的时候在 A 口输入高电平。为了使 A 口在按键触发后回到不被触发的状态，我们把 A 口设置为带下拉电阻的输入。八个 LED 是共阳极的(注：本书有不少例子的程序用到八个 LED，如果没有特别声明，指的都是共阳极)，低电平点亮，高电平熄灭。所以，我们在把 B 口设置为同相输出，要熄灭某个 LED，只需在对应的 I/O 口位输出高电平即可。

主程序：

```
#include "SPCE061.H"
int main()
{
    unsigned int I_Key;                    //定义一个变量，用以保存输入的键值
//设置 A 口为带下拉电阻的输入
    Set_IOA_Dir(0x0000);
    Set_IOA_Attrib(0x0000);
    Set_IOA_Data(0x0000);
//设置 B 口为同相低电平输出
    Set_IOB_Dir(0xFFFF);
    Set_IOB_Attrib(0xFFFF);
    Set_IOB_Data(0x0000);                  //I/O 初始化完成

    while(1)
    {
    I_Key = Get_IOA_Data();                //获取 A 口输入的键值
```

```
        switch(I_Key)                          //散转
        {
            case 0x0000: break;                //无键按下
            case 0x0001:                       //按下的键号为 1
            case 0x0002:                       //按下的键号为 2
            case 0x0004:                       //按下的键号为 3
            case 0x0008:                       //按下的键号为 4
            case 0x0010:                       //按下的键号为 5
            case 0x0020:                       //按下的键号为 6
            case 0x0040:                       //按下的键号为 7
            case 0x0080:                       //按下的键号为 8
            Set_IOB_Buffer(I_Key);             //把 A 口输入的键值送到 B 口
                break;
            default:
                break;
            }
        }
    }
```

在程序运行之前，还有几个步骤要完成(以后各节例题中也是如此)：

(1) **在** Tools\Options\Directies **下选择** Include files**，把** SPCE061.H **加入**;

(2) **在** Project\Setting\Link\Library modules **下加入** SPCE061.lib**。**

以上程序 I/O **初始化部分的程序也可以只写以下两行：**

```
SP_Init_IOA(0, 0, 0);
SP_Init_IOB(0XFFFF, 0, 0xFFFF);
```

对 I/O **进行编程的方法很多，读者可以根据自己的习惯选用其中一种。**

7.6　系统设置的 C 语言程序设计

7.6.1　系统设置的 C 函数

SPCE061.lib **中提供的相应** API **函数如下：**

(1) **函数原型：**void Watchdog_Clear(void);

功能说明：清看门狗，Watchdog Reset Timer =0.75 second**。**

用法：Watchdog_Clear()**；**

参数：无。

(2) **函数原型：**void Set_SystemClock(unsigned int);

功能说明: **设置** System **信息。**

用法：Set_SystemClock(System_Information);

参数：System Information=CPU Clock+32 kHz Mode+32 kHz Sleep Status+OSC Frequency

CPU Clock **的取值：**

C_Fosc	→ Fosc
C_Fosc_Div_2	→ Fosc/2
C_Fosc_Div_4	→ Fosc/4
C_Fosc_Div_8	→ Fosc/8 (**默认值**)
C_Fosc_Div_16	→ Fosc/16
C_Fosc_Div_32	→ Fosc/32
C_Fosc_Div_64	→ Fosc/64
C_Sleep	→ Stop (**睡眠**)

32 kHz Mode **的取值：**

C_StrongMode	→ **强振模式**
C_AutoMode	→ **弱振模式** (**默认值**)

32 kHz Sleep Status **的取值：**

C_32K_Work	→ **睡眠时**, 32 K **时钟仍然工作** (**默认值**)
C_32K_Off	→ **睡眠时**, 32 K **时钟停止工作**

OSC Frequency **的取值：**

C_Fosc_24M	→ 24.576 MHz (**默认值**)
C_Fosc_20M	→ 20.480 MHz
C_Fosc_32M	→ 32.768 MHz
C_Fosc_40M	→ 40.960 MHz
C_Fosc_49M	→ 49.152 MHz

(3) **函数原型**: void Set_LVD_Ctrl(unsigned int);

功能说明: **设置电压监测低限。**

用法: Set_LVD_Ctrl(Voltage_Level)**；**

参数: Voltage Level

Voltage Level **的取值：**

C_LVD24V	→ LVD = 2.4 V (**默认值**)
C_LVD28V	→ LVD = 2.8 V
C_LVD32V	→ LVD = 3.2 V

(4) **函数原型：**unsigned int Get_LVD_Ctrl(void)**；**

功能说明：获取电压是否到达监测低限信息。

用法：LVD_Result = Get_LVD_Ctrl();

返回值：返回值的最高位 (b15)**为** 0 **代表** VDD>VLVD**，为** 1 **代表** VDD<VLVD**。**

(5) **函数原型：**void Set_FeedBack(unsigned int);

功能说明：设置 FeedBack **信息。**

用法：Set_FeedBack(FeedBack_Data)

参数：FeedBack_Data **包含了以下三项信息：**

C_FBKEN3	→IOB3 **和** IOB5 **间形成反馈**
C_FBKEN2	→IOB2 **和** IOB4 **间形成反馈**

C_IRTxEN　　→IOB8 和 IOB10 IRTx/Tx 使能

C_FBKEN3、C_FBKEN2、C_IRTxEN 是任意选项，使用者可以根据需要来选择其中一项或多项。如：

```
FeedBack Data= C_FBKEN3 + C_FBKEN2 + C_IRTxEN
FeedBack Data= C_FBKEN3 + C_IRTxEN
FeedBack Data= C_IRTxEN
FeedBack Data= 0
```

7.6.2　应用实例

下面举几个例子说明系统设置中的 C 语言程序设计。

例 2　观察 WatchDog 的作用。

解：我们已经明确，清 WatchDog 的周期要小于 0.75 s，不然，系统会被复位重启。硬件连接图如图 7.5 所示。我们来看这个程序，当把 Delay 设为 count1 时，连接在 B 口低 8 位的 LED 亮一下就灭了；当把 Delay 设为 count0 时，连接在 B 口低 8 位的 LED 会不停地闪。为什么会这样呢？按照程序流程，程序把 LED 熄灭之后一直在 while(1)中循环清 WatchDog，不会再点亮 LED。现在，把 Delay 设为 count0 时，清 WatchDog 的时间间隔大于 0.75 s，LED 会不停地闪，证明系统重新启动了。这就是 WatchDog 作用的结果。

程序如下：

```
#include "SPCE061.h"
#define count0 0xFFFFFF;
#define count1 0xFFF;
int main()
{
    unsigned long int       Delay;
                                                    //设置 B 口为同相低电平输出
    *P_IOB_Dir =0xFFFF;
    *P_IOB_Attrib =0xFFFF;
    *P_IOB_Data =0;                                 //点亮发光管
    Delay =5000;
while(Delay --);
    *P_IOB_Data =0xFFFF;                            //熄灭发光管
    while(1)                                        //循环
    {
    *P_WatchDog_Clear=1;
    Delay =count0;
    while(Delay --);
    }
}
```

例 3　进行系统时钟设置。读者可以选择不同的 Fosc_CLK 和 CPU_CLK，观察发光二

极管亮灭快慢。

解：硬件连接图如图 7.5 所示。读者只需改变与编译处理的两行，就可以改变 Fosc_CLK 和 CPU_CLK。(其中，Fosc_CLK 和 CPU_CLK 的取值范围在 4.2.3 小节中介绍过)

```
#define Fosc_CLK_RATE        C_Fosc_49M;        //选择系统时钟
#define CPU_CLK_RATE         C_Fosc_Div_2;      //选择 CPU 时钟
```

程序如下：

```
#include "SPCE061.H"
#define Fosc_CLK_RATE     C_Fosc_49M;
#define CPU_CLK_RATE      C_Fosc_Div_2;
void  Delay();
main()
{
    unsigned int Fosc_CLK,CPU_CLK;
    *P_IOB_Dir = 0x00FF;                          //B 口低 8 位为输出
    *P_IOB_Attrib = 0x00FF;
    *P_IOB_Data = 0;
    Fosc_CLK = Fosc_CLK_RATE;
    CPU_CLK = CPU_CLK_RATE;
    *P_SystemClock = Fosc_CLK|CPU_CLK;
    while(1)
        {
        *P_IOB_Data = 0x00FF;                     //熄灭发光管
        Delay();
        *P_IOB_Data = 0;                          //点亮发光管
        Delay();
    }
}
void Delay(void)
{
    unsigned int j;
    for(j=0x7FFF;j>0;j--);
*P_Watchdog_Clear=1;
}
```

7.7 定时器/计数器设置的 C 函数

7.7.1 库中提供的 API 函数

库中提供的 API 函数如下：

(1) **函数原型**: void Set_TimerA_Data(unsigned int);
　　void Set_TimerB_Data(unsigned int);

功能说明：预置计数器初值。

用法：Set_TimerA_Data(TimerA_Data);
　　Set_TimerB_Data(TimerB_Data);

参数：

TimerA_Data = 0xFFFF–(Source A & Source B Frequency)/Desired Frequency

TimerB_Data = 0xFFFF–(Source C Frequency)/Desired Frequency

(2) **函数原型：**unsigned int Get_TimerA_Data(void);
　　unsigned int Get_TimerB_Data(void);

功能说明：读计数器的值。

用法：TimerA_Data = Get_TimerA_Data();
　　TimerB_Data = Get_TimerB_Data();

(3) **函数原型**: void Set_TimerA_Ctrl(unsigned int);
　　void Set_TimerB_Ctrl(unsigned int);

功能说明：选择时钟源频率和占空比。

用法：Set_TimerA_Ctrl(TimerA_Ctrl_Data);
　　Set_TimerB_Ctrl(TimerB_Ctrl_Data);

参数：TimerA_Ctrl_Data = Source A + Source B Frequency + Duty Cycle
　　TimerB_Ctrl_Data = Source C Frequency + Duty Cycle

缺省值(Default Value) →
　　TimerA_Ctrl_Data = C_TimerADefault
　　TimerB_Ctrl_Data = C_TimerBDefault

时钟源 A 频率取值范围(Source A Frequency)：

C_SourceA_Fosc2	→Fosc/2
C_SourceA_Fosc256	→Fosc/256
C_SourceA_32768 Hz	→32 768 Hz
C_SourceA_8192 Hz	→8192 Hz
C_SourceA_4096 Hz	→4096 Hz
C_SourceA_1	→1
C_SourceA_0	→0
C_SourceA_Ext1	→EXT1(IOB2)

时钟源 B 频率取值范围(Source B Frequency):

C_SourceB_2048Hz	→2048 Hz
C_SourceB_1024Hz	→1024 Hz
C_SourceB_256Hz	→256 Hz
C_SourceB_TMB1	→TMB1
C_SourceB_4Hz	→4 Hz
C_SourceB_2Hz	→2 Hz

C_SourceB_1 →1
C_SourceB_Ext2 →EXT2(IOB3)

时钟源 C **频率取值范围**(Source C Frequency)：

C_SourceC_Fosc2 →Fosc/2
C_SourceC_Fosc256 →Fosc/256
C_SourceC_32768Hz →32 768 Hz
C_SourceC_8192Hz →8192 Hz
C_SourceC_4096Hz →4096 Hz
C_SourceC_1 →1
C_SourceC_0 →0
C_SourceC_Ext1 →EXT1(IOB2)

(4) **函数原型**: void Set_TimeBase(unsigned int);

功能说明：时基频率选择。

用法：Set_TimeBase (TimerBase_Data);

参数：TimerBase_Data = TMB1 + TMB2 Frequency

缺省值(Default Value)→

TimerBase_Data = C_TimeBaseDefault

时钟源 TMB1 **频率取值范围**(TMB1 Frequency)：

C_TMB1_8Hz →8 Hz
C_TMB1_16Hz →16 Hz
C_TMB1_32Hz →32 Hz
C_TMB1_64Hz →64 Hz

时钟源 TMB2 **频率取值范围**(TMB2 Frequency):

C_TMB2_128Hz →128 Hz
C_TMB2_256Hz →256 Hz
C_TMB2_512Hz →512 Hz
C_TMB2_1024Hz →1024 Hz

(5) **函数原型**: void Clear_TimeBase(void);

功能说明：时基计数器复位。

用法：Clear_TimeBase();

7.7.2 定时器/计数器应用实例

下面是定时器/计数器中时基频率选择的两个例子。

例 1 选择 8 Hz **时基中断，进入中断后** IOB **口交替输出高低电平。通过示波器可观察到** IOB **口输出为方波。本例程由主程序** main.c **和中断服务程序** ISR.c **组成。(与中断有关的部分请参见第** 5 **章)**

以下为 main.c **的程序：**

```
#include "SPCE061.H"
main()
```

```
{
    ASM("INT OFF");
     *P_IOB_Dir=0xFFFF;                          //B 口设为输出
     *P_IOB_Attrib=0xFFFF;
     *P_IOB_Data=0xFFFF;
     *P_TimeBase_Setup=0;                        //时基为 8 Hz
     *P_INT_Ctrl=C_IRQ6_TMB2;                    //设置中断
     ASM("INT IRQ");
while(1)
     *P_WatchDog_Clear = C_WDTCLR;
}
```

以下为 ISR.c **的程序：**

```
#include    "SPCE061.H"
unsigned int g_uiOutput=0xFFFF;
void IRQ6(void) __attribute__ ((ISR));
void IRQ6(void)
{
  unsigned int i;
  g_uiOutput=~g_uiOutput;
  if(!(*P_INT_Ctrl&0x0001))
  {  //IRQ_TMB2
     *P_INT_Clear=0x0001;                        //清中断标志
  }
  else
  {  //IRQ_TMB1
     *P_IOB_Data=g_uiOutput;                     //从 B 口输出
     i=100;
     while(i--);                                 //延时
     *P_INT_Clear=0x0002;                        //清中断标志
  }
}
```

例 5　通过 TimerA **计数，从** IOB **口输出频率为** 1 Hz **的方波。**

解：给 TimerA **预置一个初值，它会在此基础上进行加一计数，加到** 0xFFFF **时会产生溢出，同时产生中断请求信号，送入中断控制器进行处理。与上例不同的是，上例的输出是在中断服务子程序中进行的，而本例没有中断服务程序，只是用到了** P_INT_Ctrl **寄存器的一个标志位而已。**

TimerA **预置初值计算公式：**

TimerA_Data =0xFFFF－(Source A & Source B Frequency)/Desired Frequency

定时 0.5 s**，我们利用时钟源** B **选择** 1024 Hz**，时钟源** A **为** 1**。这样我们就可以得到预**

置初值为 TimerA_Data＝0xFFFF－1024/2＝0xFDFF。

程序如下：

```
#include     "SPCE061.H"
main()
{
    unsigned int uiOutput;
    uiOutput=0x0000;
    *P_IOB_Dir=0xFFFF;
    *P_IOB_Attrib=0xFFFF;
    *P_IOB_Data=0x0000;
    *P_INT_Ctrl=C_IRQ1_TMA;
    ASM("INT OFF");
    *P_TimerA_Ctrl=C_SourceA_1+C_SourceB_1024 Hz;      //定时器 A 时钟为 1024 Hz
    *P_TimerA_Data=0xFDFF;                             //定时 0.5 s
    while(1)
    {
        *P_Watchdog_Clear = C_WDTCLR;
        if(*P_INT_Ctrl&C_IRQ1_TMA)
        {
            *P_INT_Clear=C_IRQ1_TMA;                   //清中断标志
            uiOutput ^= 0xFFFF;                        //B 口值取反输出
            *P_IOB_Data=uiOutput;
        }
    }
}
```

7.8 中断系统的 C 语言程序设计

7.8.1 中断控制的相关 C 函数

SPCE061.lib **中提供了如下与中断控制有关的** API **函数：**

(1) **函数原型：**void Set_INT_Ctrl(unsigned int);
void Set_INT_Mask(unsigned int);

功能说明：Set INT Enable。

用法：Set_INT_Ctrl(INT_Information);
Set_INT_Mask(INT_Information);

参数：INT_Information

(2) **函数原型：**unsigned int Get_INT_Mask(void);

功能说明：Get INT Enabling Information。

用法：INT_Information = Get_INT_Mask();
返回值：INT_Information
(3) **函数原型：**unsigned int　Get_INT_Ctrl(void);
功能说明：Get INT Flag。
用法：INT_Flag = Get_INT_Ctrl();
返回值：INT_Flag
(4) **函数原型：**void　INT_Clear(unsigned int);
功能说明：Clear INT Flag。
用法：INT_Clear(INT_Flag);
参数：INT_Flag
中断标志信息：

C_IRQ6_TMB2	→Timer B IRQ6	(b0)
C_IRQ6_TMB1	→Timer A IRQ6	(b1)
C_IRQ5_2 Hz	→IRQ5 2 Hz	(b2)
C_IRQ5_4 Hz	→IRQ5 4 Hz	(b3)
C_IRQ4_1 kHz	→1024 Hz IRQ4	(b4)
C_IRQ4_2 kHz	→2048 Hz IRQ4	(b5)
C_IRQ4_4 kHz	→4096 Hz IRQ4	(b6)
C_IRQ3_KEY	→Key Change IRQ3	(b7)
C_IRQ3_EXT1	→Ext1 IRQ3	(b8)
C_IRQ3_EXT2	→Ext2 IRQ3	(b9)
C_IRQ2_TMB	→Timer B IRQ2	(b10)
C_FIQ_TMB	→Timer B FIQ	(b11)
C_IRQ1_TMA	→Timer A IRQ1	(b12)
C_FIQ_TMA	→Timer A FIQ	(b13)
C_IRQ0_PWM	→PWM IRQ0	(b14)
C_FIQ_PWM	→PWM FIQ	(b15)

(5) **函数原型：**

```
void   FIQ_ON(void      );
void   FIQ_OFF(void);
void   IRQ_ON(void);
void   IRQ_OFF(void);
void   INT_FIQ(void);
void   INT_IRQ(void);
void   INT_FIQ_IRQ(void);
void   INT_OFF(void);
```

功能说明：INT On/Off Function。
用法：

```
FIQ_ON();                    →允许快速中断
```

FIQ_OFF(); →禁止快速中断
IRQ_ON(); →允许普通中断
IRQ_OFF(); →禁止普通中断
INT_FIQ(); →允许快速中断，禁止普通中断
INT_IRQ(); →禁止快速中断，允许普通中断
INT_FIQ_IRQ(); →允许快速中断，允许普通中断
INT_OFF(); →禁止快速中断，禁止普通中断

7.8.2 中断系统应用实例

1. 单中断源的应用

单中断源分为定时器中断、时基中断、触键中断和外部中断，下面分别对这几个中断进行举例说明。

例 6 定时器中断。

要求：利用定时器 A 定时 10 ms，在 A 口的 IOA0 脚输出周期 20 ms 的方波。

分析：采用定时器 A 定时，首先解决使用哪种中断方式，是选用 FIQ 中断，还是选择 IRQ 中断。当然在本例中，采用哪一种中断都可以，这里采用 IRQ 中断，即开中断时需打开 IRQ1 的定时器 A 中断，将 P_INT_Ctrl 的 IRQ1_TMA 置位。其次考虑定时 10 ms 的问题：确定定时 10 ms 选择采用哪个时钟源(P_TimerA_Ctrl)，这里采用时钟源 A 为 8 kHz，时钟源 B 为 1。当然选用其它频率也可以；接下来确定定时器 A 的初值(P_TimerA_Data)。

P_TimerA_Data =0xFFFF – (Source A & Source B Frequency)/Desired Frequency
=0xFFFF－8K/0.1K=0xFFAF。

主程序：

```
#include    "SPCE061.H"
main()
{
   *P_IOA_Dir=0xFFFF;
   *P_IOA_Attrib=0xFFFF;
   *P_IOA_Data=0xFFFF;
   *P_TimerA_Ctrl=C_SourceA_8192Hz+C_SourceB_1;        //定时器 A 时钟为 8192 Hz
   *P_TimerA_Data=0xFFAF;                              //定时 10 ms
   *P_INT_Ctrl=C_IRQ1_TMA;
   ASM("INT IRQ");
   while(1)
   *P_WatchDog_Clear = C_WDTCLR;
}
```

中断服务程序：

```
   #include "SPCE061.H"
```

```
unsigned int uiOutput=0xFFFF;
void IRQ1(void) __attribute__ ((ISR));
void IRQ1(void)
{
   uiOutput ^= 0xFFFF;                          //A 口取反输出
   *P_IOA_Data=uiOutput;
   *P_INT_Clear=C_IRQ1_TMA;                     //清中断标志
}
```

例 7　时基中断。

要求：定时 0.5 s**，使** B **口的** 8 **个** LED **闪烁。**

分析：定时 0.5 s **采用哪个时基信号比较方便呢？**SPCE061A **单片机时基信号丰富，有** 2 Hz、4 Hz、8 Hz、16 Hz、32 Hz、64 Hz、128 Hz、256 Hz、512 Hz、1024 H、2048 Hz、4096 Hz **等多种频率。显然，**2 Hz **时基信号中断是最方便的，只要触发** 2 Hz **的时基信号中断，就可以达到** 0.5 s **的定时目的。**

主程序：

```
#include    "SPCE061.H"
main()
{
      ASM("INT OFF");                           //B 口低 8 位为输出
      *P_IOB_Dir=0x00FF;
      *P_IOB_Attrib=0x00FF;
      *P_IOB_Data=0x0000;

      *P_INT_Ctrl=C_IRQ5_2Hz;                   //设定中断
      ASM("INT IRQ");
      while(1)
            *P_WatchDog_Clear = C_WDTCLR;
}
```

中断服务程序：

```
#include "SPCE061.H"
unsigned int g_uiOutput=0x0000;
void IRQ5(void) __attribute__ ((ISR));
void IRQ5(void)
{
   if(*P_INT_Ctrl&0x0004)
         {                                      //是 IRQ5_2 Hz 中断
         *P_IOB_Data=g_uiOutput;
         g_uiOutput^= 0xFFFF;
```

```
            *P_INT_Clear=0x0004;
            }
        else
        {                                                    //是 IRQ5_4 Hz 中断
            *P_INT_Clear=0x0008;
        }
}
```

例 8　触键唤醒中断。

触键唤醒中断源主要是在系统进入睡眠状态后，通过 A 口低 8 位的按键来唤醒系统时钟，同时进入触键唤醒中断，恢复睡眠时的 PC 指针。

要求：使系统处于睡眠状态，可通过触键唤醒，唤醒后逐个点亮八个指示灯，然后熄灭八个指示灯，继续进入睡眠状态。

分析：考虑如何使系统进入睡眠状态，可以通过设置时钟系统控制寄存器 (P_SystemClock)的 B4 位使系统进入睡眠状态。因为只有 A 口的低 8 位具有唤醒功能，所以按键必须接 A 口低 8 位，A 口高 8 位接八个 LED。

主程序：

```
#include    "SPCE061.H"
main()
{
    unsigned int uiTem;
    ASM("INT OFF");
    *P_IOA_Dir = 0xFF00;                    //设置 A 口低 8 位输入；高 8 位输出
    *P_IOA_Attrib = 0xFF00;
    *P_IOA_Data = 0x0000;
    *P_INT_Ctrl = C_IRQ3_KEY;               //开 IRQ3_Key 中断
    ASM("INT IRQ");
    uiTem = *P_IOA_Latch;                   //锁存 A 口低 8 位值
    while(1)
    {
     *P_IOA_Data = 0xFF00;                  //熄灭发光管
     *P_SystemClock = C_Sleep;              //进入睡眠，有键按下时唤醒
    }
}
```

中断服务程序：

```
#include    "SPCE061.H"
unsigned int uiOutput,uiDelay;
void IRQ3(void) __attribute__ ((ISR));
void IRQ3(void)
{
```

```
    if(*P_INT_Ctrl&0x0100)
    {                                                    //清外中断 1 标志
        *P_INT_Clear=0x0100;
    }
    else if(*P_INT_Ctrl&0x0200)
    {                                                    //清外中断 2 标志
        *P_INT_Clear=0x0200;
    }
    else
    {                                                    //清触键唤醒中断标志
        for(uiOutput=0x7FFF;uiOutput>0x007f;uiOutput>>=1)
        {
            *P_IOA_Data=uiOutput;
            uiDelay=0x7FFF;
            while(uiDelay--);
            *P_WatchDog_Clear = 0x0001;
        }
        *P_INT_Clear=0x0080;
    }
}
```

例 9　外部中断。

SPCE061A **有两个外部中断，为负跳沿(或者称为下降沿)触发。可以使用形成反馈电路定时触发外部中断，也可以不使用反馈电路，通过** IOB2 **或** IOB3 **引入外部中断触发信号，进入外部中断。**

要求：利用外部中断控制连接在 A **口低** 8 **位的八个** LED**。**

分析：首先考虑使用外部中断 1 **还是外部中断** 2**，此处两个外部中断都可以，只是初始化时略有不同，选择外部中断** 1**，初始化** IOB2 **为带上拉电阻的输入端口；选择外部中断** 2**，初始化** IOB3 **为带上拉电阻的输入端口。此例选择外部中断** 1**。**A **口低** 8 **位接** LED**，**IOB2 **接键盘。程序运行后，按键一次，**LED **点亮，再按一次，**LED **熄灭，如此反复。**

主程序：

```
#include "SPCE061.H"
main()
{
    *P_IOA_Dir=0xFF;                                     //设置 A 口低 8 位为输出
    *P_IOA_Attrib=0xFF;
    *P_IOA_Data = 0x00;

    *P_IOB_Dir=0x00;                                     //IOB2 为输入
    *P_IOB_Attrib=0x00;
```

```
        *P_IOB_Data=0x04;
        *P_INT_Ctrl=C_IRQ3_EXT1;                    //开外中断 1
        ASM("INT IRQ");
        while(1)
        {
            *P_WatchDog_Clear = C_WDTCLR;
        }
}
```

中断服务程序：

```
#include    "SPCE061.H"
unsigned int g_uiOutput=0;
void IRQ3(void) __attribute__ ((ISR));
void IRQ3(void)
{
        unsigned int uiDelay;
        if(*P_INT_Ctrl&C_IRQ3_EXT1)
        {                                           //外中断 1
            *P_IOA_Data=(g_uiOutput^=0xffff);
            uiDelay=0x7FFF;
            while(uiDelay--);
            *P_INT_Clear=C_IRQ3_EXT1;
        }
        else if(*P_INT_Ctrl&C_IRQ3_EXT2)
        {                                           //外中断 2
            *P_INT_Clear=C_IRQ3_EXT2;
        }
        else
        {                                           //按键中断
            *P_INT_Clear=C_IRQ3_KEY;
        }
}
```

2. 多中断源应用

在单片机系统开发过程中，使用多个中断源的机会很多。SPCE061A **单片机多中断源的使用有两种方式，一种是同一个中断入口多个中断源的使用；另一种是不同中断入口的多个中断源的使用。下面针对这两种方式分别举例说明。**

例 10　**同一个中断向量的多个中断源使用。**

要求：IRQ6 **中断有两个中断源** IRQ6_TMB1 **和** IRQ6_TMB2**，利用两个中断源分别控制四个发光二极管，分别为** 1 s **和** 0.5 s **闪烁。**

分析：IRQ6_TMB1 **有** 8 Hz、16 Hz、32 Hz **和** 64 Hz **多种选择，选择其中任一频率均可作** 0.5 s **定时的时钟源。此处我们选择** 64 Hz **；同样，**IRQ6_TMB2 **有** 128 Hz、256 Hz、512 Hz **和** 1024 Hz **多种选择，每种频率都可以达到定时** 1 s**，此处选择** 128 Hz**。**

主程序：

```
#include    "SPCE061.H"
main()
{
    *P_IOA_Dir=0xFFFF;                          //设置 IOA0～IOA3 为输出
    *P_IOA_Attrib=0xFFFF;
    *P_IOA_Data=0x0000;

    *P_IOB_Dir=0xFFFF;                          //设置 IOB0～IOB3 为输出
    *P_IOB_Attrib=0xFFFF;
    *P_IOB_Data=0x0000;

    *P_TimeBase_Setup=C_TMB1_64Hz|C_TMB2_128Hz;          //选择时基
    *P_INT_Ctrl=C_IRQ6_TMB1|C_IRQ6_TMB2;                 //开中断
    ASM("int irq");
    while(1)
        *P_WatchDog_Clear=0x0001;
}
```

中断服务程序：

```
#include    "SPCE061.H"
unsigned int g_uiTime1,g_uiTime2;
void IRQ6(void) __attribute__ ((ISR));
void IRQ6(void)
{
    if(*P_INT_Ctrl&0x0001)
    {    //TMB2
        g_uiTime2+=1;
        if(g_uiTime2<=64)                       //0.5 s
            *P_IOB_Data=0x0000;
        else if(g_uiTime2>128)                  //1 s
            g_uiTime2=0;
        else
            *P_IOB_Data=0x000F;
        *P_INT_Clear=C_IRQ6_TMB2;
    }
    else
```

```
    {     //TMB1
          g_uiTime1+=1;
          if(g_uiTime1<=64)
                *P_IOA_Data=0x0000;
          else if(g_uiTime1>128)
                g_uiTime1=0;
          else
                *P_IOA_Data=0x000F;
          *P_INT_Clear=C_IRQ6_TMB1;
    }
}
```

例 11　不同中断入口的中断源使用。

要求：用不同中断入口的中断源，分别控制四个发光二极管，分别以 1 s 和 0.5 s 闪烁。

分析：可以采用定时器中断也可以采用时基中断。此例中，0.5 s 定时使用的是 IRQ2 中的定时器 B；1 s 定时使用的是 IRQ4 中的 IRQ_1 kHz 中断。

主程序：

```
#include    "SPCE061.H"
#define     TIMER_DATA_FOR_4 kHz    (0xFFFF - 2048)      //0.5 s
main()
{
    *P_IOA_Dir=0xFFFF;                                    //设置 IOA0～IOA4 为输出
    *P_IOA_Attrib=0xFFFF;
    *P_IOA_Data=0x00FF;
    *P_IOB_Dir=0xFFFF;
    *P_IOB_Attrib=0xFFFF;
    *P_IOB_Data=0x00FF;
    *P_TimerB_Data=TIMER_DATA_FOR_4 kHz;                  //定时器 B 初始化
    *P_TimerB_Ctrl=C_SourceA_4096 Hz;
    *P_INT_Ctrl=C_IRQ4_1 kHz|C_IRQ2_TMB;                  //开中断
    ASM("int irq");
    while(1)
          *P_WatchDog_Clear=0x0001;
}
```

中断服务程序：

```
#include    "SPCE061.H"
unsigned int g_uiIOA_LED=0xFF, g_uiIOB_LED=0xFF, g_uiClockCnt=0;
void IRQ2(void) __attribute__ ((ISR));
void IRQ2(void)
{
```

```
        *P_IOB_Data=g_uiIOB_LED;
        g_uiIOB_LED ^= 0xFFFF;
        *P_INT_Clear=0x0400;
}
void IRQ4(void) __attribute__ ((ISR));
void IRQ4(void)
{
        if(*P_INT_Ctrl&C_IRQ4_1 kHz)
        {
                if(g_uiClockCnt<1024)
                   g_uiClockCnt++;
                else
                {
                        *P_IOA_Data=g_uiIOA_LED;
                        g_uiIOA_LED ^= 0xFFFF;
                        g_uiClockCnt = 0x0000;
                        *P_INT_Clear=C_IRQ4_1 kHz;
                }
        }
        if(*P_INT_Ctrl&C_IRQ4_2 kHz)
        {
                *P_INT_Clear=C_IRQ4_2 kHz;
        }
        if(*P_INT_Ctrl&C_IRQ4_4 kHz)
        {
                *P_INT_Clear=C_IRQ4_4 kHz;
        }
}
```

7.9　A/D、D/A 转换的 C 语言程序设计

7.9.1　相关 C 函数

相关的 C 函数如下：

(1) **函数原型：**void　Set_ADC_Data(unsigned int);

功能说明：Set ADC Data，for Microphone input data。

用法：Set_ADC_Data(ADC_Data);

参数：ADC_Data = b15～b6

(2) **函数原型：**unsigned int Get_ADC_Data(void);

功能说明：Get ADC Data，for Microphone input data。

用法：ADC Data = Get_ADC_Data();

返回值：ADC_Data = b15～b6

(3) **函数原型：**unsigned int Get_ADC_Ctrl(void);
unsigned int Get_ADC_MUX_Ctrl(void);

功能说明：Get ADC Conversion Ready。

用法：ADC Ready1 = Get_ADC_Ctrl ();
ADC Ready2 = Get_ADC_MUX_Ctrl();

返回值：ADC Ready1 =b15
1: Ready
0: Converting
ADC Ready2 = b15,b14
b15 →
1: Ready
0: Converting
b14 →
1: Conversion completed successfully
0: Conversion failed

(4) **函数原型：**void Set_ADC_Ctrl(unsigned int);

功能说明：Set ADC Control Register。

用法：Set_ADC_Ctrl(ADC_Ctrl_Data);

参数：ADC_Ctrl_Data →

C_ADCE	→**允许** A/D **转换**
C_ADCN	→**禁止** A/D **转换**
C_MIC_ENB	→**允许** MIC **通路转换**
C_MIC_DIS	→**禁止** MIC **通路转换**
C_AGCE	→**开放** AGC **功能**
C_AGCN	→**禁止** AGC **功能**
C_DAC_OUT2mA	→DAC **输出电流为** 2 mA
C_DAC_OUT3mA	→DAC **输出电流为** 3 mA
C_VEXTREF_ENB	→**采用外接参考电压**
C_VEXTREF_DIS	→**采用内接参考电压**
C_V2VREFB_DIS	→**不用** 2 V **电压**
C_V2VREFB_ENB	→**采用** 2 V **电压**

Note: ADC_Ctrl_Data is a write only data, so any change of data must reload each component.

ADC Enable	Set_ADC_Data(C_ADCE);
Microphone Enable	Set_ADC_Data(C_ADCE+ C_MIC_ENB);
Microphone Enable With AGC	Set_ADC_Data(C_ADCE+ C_MIC_ENB+ C_AGCE);

(5) **函数原型：**void Set_ADC_MUX_Ctrl(unsigned int);

功能说明：Set ADC Line In Channel。

用法：Set_ADC_MUX_Ctrl(Channel Number);

参数：

Channel Number →

C_ADC_CH_MICin	→MIC **通道输入**
C_ADC_CH1	→**线路** 1 **通道**
C_ADC_CH2	→**线路** 2 **通道**
C_ADC_CH3	→**线路** 3 **通道**
C_ADC_CH4	→**线路** 4 **通道**
C_ADC_CH5	→**线路** 5 **通道**
C_ADC_CH6	→**线路** 6 **通道**
C_ADC_CH7	→**线路** 7 **通道**

(6) **函数原型：**unsigned int Get_ADC_LineIn_Data(void);

功能说明：Get ADC Line In Data。

用法：ADC Line In Data = Get_ADC_LineIn_Data();

返回值：ADC Line In Data = b15～b6

(7) **函数原型：**void _Mic_Input_Ctrl(unsigned int,unsigned int);

功能说明：Set Microphone Input Control。

用法：Mic_Input_Ctrl(Trigger Type, AGC On/Off);

参数：Trigger Type = DAC trigger type + Mic AD trigger type

如果使用麦克风输入，DAC **输出，请选择** DAC **触发类型。**

Example:

Trigger Type = C_DA1_LatchA+C_DA2_LatchB+C_AD_LatchA

Trigger Type = C_DA2_LatchAB+C_AD_LatchA

DAC Trigger Type → DAC1 Latch + DAC2 Latch

DAC1_Latch：

C_DA1_Direct	→ **直接应用** DAR1 **数据(默认值)**
C_DA1_LatchA	→ **利用定时器** A **锁存** DAR1 **数据**
C_DA1_LatchB	→ **利用定时器** B **锁存** DAR1 **数据**
C_DA1_LatchAB	→ **利用定时器** A **或** B **锁存** DAR1 **数据**

DAC2_Latch：

C_DA2_Direct	→ **直接应用** DAR2 **数据(默认值)**
C_DA2_LatchA	→ **利用定时器** A **锁存** DAR2 **数据**
C_DA2_LatchB	→ **利用定时器** B **锁存** DAR2 **数据**
C_DA2_LatchAB	→ **利用定时器** A **或** B **锁存** DAR2 **数据**

AD_ Latch：

C_AD_Direct	→ **直接读** P_ADC(R)($70114 H) (**默认值**)
C_AD_LatchA	→ **利用定时器** A **启动转换**

C_AD_LatchB → **利用定时器** B **启动转换**
C_AD_LatchAB → **利用定时器** A **或** B **启动转换**
AGC On/Off：
C_AGCE → **启动** AGC **功能**
C_AGCN → **禁止** AGC **功能**

(8) **函数原型：**void _Line_Input_Ctrl(unsigned int,unsigned int);
功能说明：Set Line Input Control。
用法：Line_Input_Ctrl(Line In Channel , Vref On/Off);
参数：Line in channel：
C_ADC_CH1 →**输入通道**(IOA0)
C_ADC_CH2 →**输入通道**(IOA1)
C_ADC_CH3 →**输入通道**(IOA2)
C_ADC_CH4 →**输入通道**(IOA3)
C_ADC_CH5 →**输入通道**(IOA4)
C_ADC_CH6 →**输入通道**(IOA5)
C_ADC_CH7 →**输入能道**(IOA6)
Vref On/Off：
C_VEXTREF_ENB →**采用外部参考电压**
C_VEXTREF_DIS →**不用外部参考电压**(**默认值**)
C_V2VREFB_ENB →**允许** 2 V **参考电压输出**
C_V2VREFB_DIS →**不允许** 2 V **参考电压输出**

(9) **函数原型：**void Set_DAC1_Data(unsigned int);
void Set_DAC2_Data(unsigned int);
功能说明：Set DAC Data。
用法：Set_DAC1_Data(DAC1 Data);
Set_DAC2_Data(DAC2 Data);
参数：DAC1 Data = b15～b6
DAC2 Data = b15～b6

(10) **函数原型：**unsigned int Get_DAC1_Data(void)；
unsigned int Get_DAC2_Data(void);
功能说明：Get DAC Data。
用法：DAC1 Data = Get_DAC1_Data();
DAC2 Data = Get_DAC2_Data();
返回值：DAC1 Data = b15～b6
DAC2 Data = b15～b6

(11) **函数原型：**void Set_DAC_Ctrl(unsigned int);
功能说明：Set DAC Control Register。
用法：Set_DAC_Ctrl(DAC Ctrl Data);
参数：DAC Ctrl Data = DAC1_Latch + DAC2_Latch + AD_Latch

DAC1_Latch1：

C_DA1_Direct → 直接应用 DAR1 数据(默认值)

C_DA1_LatchA → 利用定时器 A 锁存 DAR1 数据

C_DA1_LatchB → 利用定时器 B 锁存 DAR1 数据

C_DA1_LatchAB → 利用定时器 A 或 B 锁存 DAR1 数据

DAC1_Latch2：

C_DA2_Direct → 直接应用 DAR2 数据(默认值)

C_DA2_LatchA → 利用定时器 A 锁存 DAR2 数据

C_DA2_LatchB → 利用定时器 B 锁存 DAR2 数据

C_DA2_LatchAB → 利用定时器 A 或 B 锁存 DAR2 数据

AD_ Latch：

C_AD_Direct → 直接读 P_ADC(R)的值(默认值)

C_AD_LatchA → 利用定时器 A 启动转换

C_AD_LatchB → 利用定时器 B 启动转换

C_AD_LatchAB → 利用定时器 A 或 B 启动转换

7.9.2　应用实例

下面举例说明 A/D、D/A 转换中的 C 语言程序设计。

例 12　通过模拟量输入口 LINE_IN 输入电压值，通过读取 P_ADC_MUX_Data 单元可以获得 10 位 A/D 转换的数据。而从该单元读取数据后，又会使 RDY 自动清“0”来重新启动进行 A/D 转换。若未读取 P_ADC_MUX_Data 单元中的数据，RDY 仍保持为“1”，则不会启动下一次的 A/D 转换。

解：本例中，电压信号从 IOA0 输入，参考电压为 3.3 V。每调用一次子函数 AD()，将得到一个 A/D 转换的值。调用了 16 次 AD()，取平均值作为 A/D 转换的结果。fVoltage 保存的就是换算后的电压值。

程序：

```
#include "SPCE061V004.H"
unsigned int AD(void);
main()
{
    unsigned long int uiData;
    unsigned int    i;
    float       fVoltage;
    fVoltage=0.0;
    *P_ADC_MUX_Ctrl=C_ADC_CH1;              // IOA0 输入
    *P_ADC_Ctrl= C_ADCE;                    //ADC 允许
    uiData=*P_ADC_LINEIN_Data;              //ADC 启动
    uiData=0;
    for(i=0;i<16;i++)
```

```
    {
        *P_WatchDog_Clear=C_WDTCLR;
        uiData += AD();
    }
    uiData >>= 4;                                    //除 16
    fVoltage=(float)uiData/0xffc0*3.3;
}

unsigned int AD(void)
{
    unsigned int uiData;
    while(!(*P_ADC_MUX_Ctrl&0x8000));                //等待转换完成
    uiData=*P_ADC_LINEIN_Data;
    return(uiData&0xffc0);
}
```

例 13　通过编程实现一个锯齿波，将 DAC 输出端接示波器可以观察到锯齿波形，同时也可以听到 AUD1 和 AUD2 两端的扬声器有持续间断的声音。

程序如下：

```
#include "SPCE061V004.H"
main()
{
    unsigned int uiData,i;
    ASM("int off");                                  //关中断
    *P_DAC_Ctrl=C_DA1_Direct;                        //锁定 DAC1
    uiData=0x0;
    while(1)
    {
        *P_WatchDog_Clear=C_WDTCLR;
        *P_DAC1=uiData;                              //输出转换结果
        i=32;
        while(i--);
        uiData+=0x0040;
    }
}
```

例 14　采用自动方式(定时器 A 溢出执行 ADC 转换)，通过 A/D 将 MIC_IN 输入的语音信号转换为数字信号，再通过 D/A 的两个通道 AUD1 和 AUD2 播放。(本例是即录即放，没有进行滤波处理，效果不是很好。要达到比较好的效果，可以使用凌阳提供的 DVR 函数。详见第 6 章)

主程序：

```
#include     "SPCE061V004.H"
#define      TIMER_DATA_FOR_8 kHz (0xFFFF−1500)      //时钟 Fosc/2, 采样率为 8 kHz
main()
{
    ASM("int off");
    *P_TimerA_Ctrl = 0x0030;                         //时钟频率为 CLKA 的 Fosc/2
    *P_TimerA_Data = TIMER_DATA_FOR_8 kHz;           //数据采样率为 8 kHz
    *P_ADC_Ctrl = 0x0015;                            //设置 AGC、自动方式
                                                     //MIC_IN 通道输入
    *P_DAC_Ctrl = 0x00A8;                            //通过 TimerA 溢出触发
                                                     //ADC 自动方式转换
    *P_INT_Ctrl = 0x1000;                            //设置 IRQ1_TM 中断
    ASM("int irq");
    while(1)
    *P_WatchDog_Clear=C_WDTCLR;
}
```

中断服务程序：

```
#include     "SPCE061V004.H"
void IRQ1(void) __attribute__ ((ISR));
void IRQ1(void)
{
        unsigned int uiData;
        uiData=*P_ADC;                               //读取 A/D 转换的数值
        *P_DAC1 = uiData;                            //DAC 输出
        *P_DAC2 = uiData;
        *P_INT_Clear = C_IRQ1_TMA;                   //清中断标识位
}
```

7.10　SIO 和 UART 的 C 语言程序设计

7.10.1　相关 C 函数

相关 C 函数的描述如下：

(1) **函数原型：**void Set_SIO_Start(void);

功能说明：启动 SIO，向 SIO 接口发送启动信号。

用法：Set_SIO_Start(void);

参数：无。

(2) **函数原型：**unsigned int Get_SIO_Start(void);

功能说明：测试 SIO 状态。

用法：SIO Busy Flag = Get_SIO_Start();
返回值：SIO **状态标志**
忙：0x0080
准备好：0x0000
(3) **函数原型：**void Set_SIO_Stop(void);
功能说明：停止 SIO**，向** SIO **接口发送停止信号。**
用法：Set_SIO_Stop();
参数：无。
(4) **函数原型：**void Set_SIO_Data(unsigned int);
功能说明：发送 SIO **数据。**
用法：Set_SIO_Data();
参数：数据为 b7～b0**。**
(5) **函数原型：**int Get_SIO_Data(void);
功能说明：获取 SIO **数据。**
用法：SIO Data = Get_SIO_Data();
返回值：数据为 b7～b0**。**
(6) **函数原型：**void Set_SIO_Ctrl(unsigned int);
功能说明：设置 SIO **控制寄存器。**
用法：Set_SIO_Ctrl(SIO Ctrl Data);
参数：SIO Ctrl Data = Address Mode + Clock Select + RW_EN + R/W + SIO Config
SIO Ctrl Data contents
Address Mode：
C_SIO_Addr8 → 8 **位地址**(A7～A0)
C_SIO_Addr16 → 16 **位地址**(A15～A0)(**默认**)
C_SIO_Addr24 → 24 **位地址**(A23～A0)
Clock Select：
C_SIO_Clk_Div_4 → CPU CLK/4
C_SIO_Clk_Div_8 → CPU CLK/8
C_SIO_Clk_Div_16 → CPU CLK/16(**默认**)
C_SIO_Clk_Div_32 → CPU CLK/32
RW_EN：
C_SIO_RW_Dis →**读/写控制为禁止**
disable
C_SIO_RW_ENB →**读/写控制为允许**(**默认**)
default
R/W：
C_SIO_RD → SIO **读**
C_SIO_WR → SIO **写**
SIO Config：

C_SIO_ENB　→ 允许 SIO 重新设置
C_SIO_Dis　→ 禁止 SIO 重新设置(默认)

(7) 函数原型：unsigned int Get_SIO_Ctrl(void);
功能说明：测试 SIO 控制寄存器的值。
用法：SIO Ctrl Data = Get_SIO_Ctrl();
返回值：SIO Ctrl Data = Address Mode + Clock Select + RW_EN + R/W + SIO Config

(8) 函数原型: void Set_SIO_Address(unsigned long int);
void Set_SIO_Addr_Low(unsigned int);
void Set_SIO_Addr_Mid(unsigned int);
void Set_SIO_Addr_High(unsigned int);
功能说明：设置 SIO 地址。
用法：Set_SIO_Address(SIO Address);
Set_SIO_Addr_Low(SIO Low Address);
Set_SIO_Addr_Mid(SIO Middle Address);
Set_SIO_Addr_High(SIO High Address);
参数：SIO 地址，SIO 低字节地址，SIO 中字节地址，SIO 高字节地址。
注: SIO Address = SIO High + Middle + Low Address

(9) 函数原型：unsigned long int Get_SIO_Address(void);
unsigned int Get_SIO_Addr_Low(void);
unsigned int Get_SIO_Addr_Mid(void);
unsigned int Get_SIO_Addr_High(void);
功能说明：Get SIO Address。
用法：SIO Address = Get_SIO_Address();
SIO Low Address = Get_SIO_Addr_Low();
SIO Middle = Get_SIO_Addr_Mid();
SIO High Address = Get_SIO_Addr_High();
返回值：SIO 地址，SIO 低字节地址，SIO 中字节地址，SIO 高字节地址
注：SIO Address = SIO High + Middle + Low Address

(10) 函数原型：void Set_UART_Command1(unsigned int);
功能说明：设置串行口控制寄存器 1。
用法：Set_UART_Command1(UART Command1 Register);
参数：串行口控制寄存器 1 的值：
C_UART_Parity_ENB　→ 使用串行口奇偶校验位
C_UART_Parity_Odd　→ 使用串行口奇校验位
C_UART_Parity_Even　→ 使用串行口偶校验位
C_UART_Reset　→ 串行口内部复位
C_UART_Tx_IRQ_ENB　→ 允许串行口发送中断
C_UART_Rx_IRQ_ENB　→ 允许串行口接收中断

(11) 函数原型：void Set_UART_Command2(unsigned int);

功能说明：设置串行口控制寄存器 2。

用法：Set_UART_Command2(UART Command2 Register);

参数：串行口控制寄存器 2 **的值：**

C_UART_Tx_Pin_ENB → **利用串行口发送管脚**

C_UART_Rx_Pin_ENB → **利用串行口接收管脚**

(12) **函数原型：**unsigned int Get_UART_Command2(void);

功能说明：获取串行口控制寄存器 2 **的值。**

用法：UART Command2 Register = Get_UART_Command2();

返回值：串行口控制寄存器 2 **的值：**

C_UART_Parity_Error → **串行口奇偶校验错误**

C_UART_OverRun_Error → **串行口运行溢出错误**

C_UART_Frame_Error → **串行口帧错误**

C_UART_Tx_RDY → **串行口发送准备好**

C_UART_Rx_RDY → **串行口接收准备好**

(13) **函数原型：**void Set_UART_Data(unsigned int);

功能说明：发送串行数据。

用法：Set_UART_Data(UART Data);

参数：串行数据 D7～D0

(14) **函数原型：**unsigned int Get_UART_Data(void);

功能说明：获取串行口数据。

用法：UART Data = Get_UART_Data();

返回值：串行数据 D7～D0

(15) **函数原型：**void Set_UART_BaudRate(unsigned int);

void Set_UART_BaudScalarLow(unsigned int);

void Set_UART_BaudScalarHigh(unsigned int);

功能说明：设置串行口波特率。

用法：Set_UART_BaudRate(Baud Rate);

Set_UART_BaudScalarLow(Low Baud Rate);

Set_UART_BaudScalarHigh(High Baud Rate);

参数：波特率，低字节值，高字节值。

Note：Baud Rate = (Fosc/2)/Scale = High + Low Baud Rate

用户能够自定义波特率

running under Fosc=24.576 MHz：

C_BaudRate_2400 → **串行口波特率为** 2400 b/s

C_BaudRate_4800 → **串行口波特率为** 4800 b/s

C_BaudRate_9600 → **串行口波特率为** 9600 b/s

C_BaudRate_19 200 → **串行口波特率为** 19 200 b/s

C_BaudRate_38 400 → **串行口波特率为** 38 400 b/s

C_BaudRate_48 000 → **串行口波特率为** 48 000 b/s

(16) **函数原型：**unsigned int Get_UART_BaudRate(void);
unsigned int Get_UART_BaudScalarLow(void);
unsigned int Get_UART_BaudScalarHigh(void);

功能说明：获取串行口波特率值。

用法：Baud Rate = Get_UART_BaudRate();
Low Baud Rate = Get_UART_BaudScalarLow();
High Baud Rate = Get_UART_BaudScalarHigh();

返回值：波特率，低字节，高字节

注：Baud Rate = High + Low Baud Rate

7.10.2　应用实例

下面举例说明 SIO 和 UART 中的 C 语言程序设计。

例 15　通过 SIO 接口将 SPCE061A 与 SPR4096 连接起来，如图 7.6 所示，图中只画出了 SPCE061A 与 SPR4096 连接的线路。

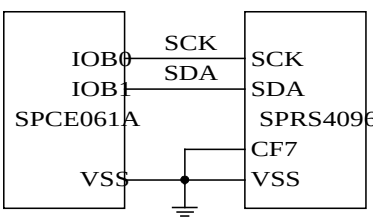


图 7.6　SPCE061A 与 SPR4096 连接图

解：以下程序实现的功能是先对 SPR4096 进行全面擦除，然后往 0x5678 单元写入 0x55AA 的值。

注意，首先应将 SIO 设置为输出口。

```
#define          C_SIOCLOCK      0x0010;                    //定义 CPU 时钟
#include     "SPCE061V004.H"
main()
{
        long int i;
        unsigned long int ulAddr;
        unsigned int uiData;
  //初始化 SIO 口
        *P_IOB_Dir = 0xFFFF;
        *P_IOB_Attrib = 0xFFFF;
       *P_IOB_Data = 0x0003;
       *P_SIO_Ctrl = 0x00C3;
  //擦除 SPR4096
        *P_SIO_Ctrl=0x00C0+C_SIOCLOCK;                    //clk=CPUclk/8,16 位地址
```

```
        *P_SIO_Addr_Low=0x0000;                    //SFLASH 地址
        *P_SIO_Addr_Mid=0x00C0;
        *P_SIO_Addr_High=0x00C0;
        *P_SIO_Start=1;                            //启动写模式
        *P_SIO_Data=0;                             //A7～A0 = 0
  //等待 SIO 空闲
       i=*P_SIO_Start;
        i&=0x0080;
        while(i)
        {
            i=*P_SIO_Start;
             i&=0x0080;
        }
        *P_SIO_Stop=1;
        i=0x7FF;                                          //延时
        while(i--);                          //往 0x5678 单元写入一个数据 0x55AA
        ulAddr=0x5678;
         *P_SIO_Addr_Low=ulAddr;                   //SFLASH 地址
         ulAddr>>=8;
         *P_SIO_Addr_Mid=ulAddr;
         ulAddr>>=8;
         ulAddr&=0x0007;
         *P_SIO_Addr_High=ulAddr;
         *P_SIO_Ctrl=0x00C3+C_SIOCLOCK;            //clk=CPUclk/8,24 位地址
         *P_SIO_Start=1;                           //启动写模式
        i=0x2FF;                                   //延时
        while(i--);
        uiData=0x55AA;
         *P_SIO_Data=uiData;                       //往 SIO 送数据 0x55AA
       i=*P_SIO_Start;                             //等待 SIO 空闲
        i&=0x0080;
        while(i)
        {
            i=*P_SIO_Start;
             i&=0x0080;
        }
        *P_SIO_Stop=1;
  }
```

例 16　用 UART 来接收 PC 机的 RS-232 串行接口发送的数据。如果接收到值为 79(大

写的字符“O”)的 ACSII 码，就点亮与 IOA0～IOA7 相连的 LED；如果接收到值为 67(大写的字符“C”)的 ACSII 码，就熄灭 LED。

程序如下：

```
#include     "SPCE061V004.H"
main()
{
        unsigned int uiData,Ret;
        *P_IOA_Dir=0xFFFF;                          //IOA0～IOA7 初始化为输出
        *P_IOA_Attrib=0xFFFF;
        *P_IOA_Data=0xFFFF;
        *P_UART_Command1=0x20;                      //UART 内部复位
        *P_UART_Command1=0x00;
        *P_UART_BaudScalarHigh=0x00;                //波特率设置为 115 200 b/s
        *P_UART_BaudScalarLow=0x6B;
        *P_UART_Command1=0x000C;                    //允许接收
        *P_UART_Command2=0x00C0;
        Ret=*P_UART_Data;                           //清接收缓冲区
        while(1)
        {
            Ret=*P_UART_Command2;
            Ret=Ret&0x0080;
            while(Ret==0)                           //等待接收完毕
            {
                Ret=*P_UART_Command2;
                Ret=Ret&0x0080;
                *P_WatchDog_Clear=C_WDTCLR;
            }
            uiData=*P_UART_Data;                    //读出接收的数据
            if(uiData==79)
                *P_IOA_Data=0x0000;                 //点亮 LED
            else if(uiData==67)
                *P_IOA_Data=0xFFFF;                 //熄灭 LED
        }
}
```

第 8 章　单片机应用系统设计

前面我们介绍了单片机的基本原理、系统组成和接口技术。对于单片机应用系统设计来说，这些内容只能使我们掌握单片机所提供的软、硬件资源，以及怎样合理使用这些资源，是应用系统设计的基础。除此之外，一个实际应用系统还需配置多种接口。系统设计会涉及到更复杂的内容和问题(如多种应用电路：模拟电路、信号传感器及检测电路、伺服驱动电路、隔离电路等)。因此，单片机应用系统的设计应遵循一些基本原则。从实用角度出发，了解系统设计方法，对于单片机系统的工程设计与开发应用有着十分重要的意义。

8.1　系统设计内容

单片机应用系统是由硬件系统和软件系统两部分组成的。硬件系统是指单片机、外部扩展的存储器、外围设备及其接口电路等；软件系统包括监控程序和应用程序。

8.1.1　系统硬件组成

单片机的典型应用系统应包括单片机(CPU)模块、用于检测信号的传感器输入模块、控制输出模块以及基本的人机对话模块等。对于单机系统不能满足要求时，应采用多机系统，即包含多个单片机系统及机与机之间的通信模块。一个单片机应用系统框图如图 8.1 所示。

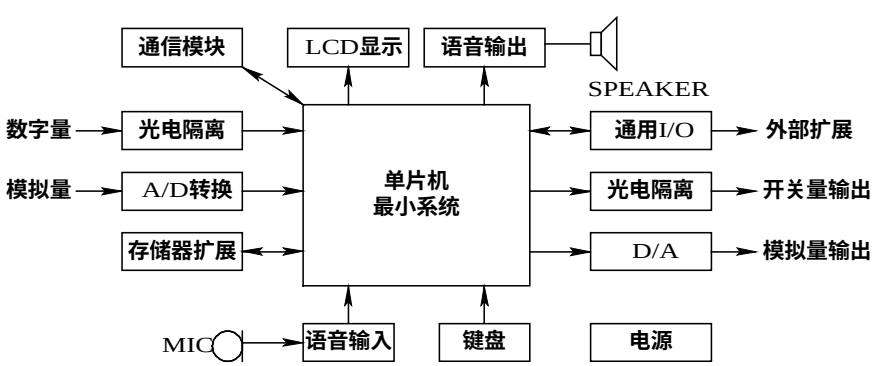


图 8.1　单片机应用系统框图

1. 输入模块

输入模块是单片机与测控对象连接的部分，是系统数据采集的输入通道。通常，来自测控对象的现场信息是多种多样的，按物理量的特征可分为模拟量和数字量两种。

数字量输入：对于数字量的采集，输入比较简单，只需进行光电隔离，可直接作为计数输入。

模拟量输入：模拟量输入通道比较复杂，一般包括传感器、隔离放大器、滤波、采样保持器、多路电子开关、A/D 转换器及其接口电路等。

传感器：用来采集现场的各种信号，并将其变换成电信号，以满足单片机的输入要求。现场信号有各种各样，有电信号(如电压、电流、电磁量等)，也有非电量信号(如温度、湿度、压力、流量、位移量等)，对于不同物理量应选择相应的传感器。这一部分也是计算机专业学生的一个薄弱环节，要成为系统设计师、总工程师，就必须从全局出发，对系统涉及到的各个领域都十分清楚。在此提出来，请读者注意拓宽自己的知识面。

隔离放大与滤波：一般情况下传感器的输出信号比较微弱，不能满足系统的要求，要经过放大处理后才能作为输入信号。另外，信息来自各种工业生产现场，存在大量的工业噪音干扰。为了提高系统的可靠性，必须采取隔离措施，滤掉干扰，这是整个系统抗干扰设计的重点部位。

多路电子开关：用多路电子开关实现一个 A／D 转换器对多路模拟信号进行转换，是受单片机控制的多路模拟电子开关，某一时刻需要对某路模拟信号进行转换，由单片机向多路电子开关发出路地址信息，把该路模拟信号与 A/D 转换器接通，其它模拟信号与 A/D 转换器隔离，从而实现用一个 A/D 转换器对多路信号的转换。

A/D 转换器：前向通道中模拟系统与数字系统连接的核心部件，将模拟信号转换为数字信号。

综上所述，输入模块具有以下特点：

(1) 与现场采集对象相连，是现场干扰进入系统的主要通道，是整个系统抗干扰设计的重点部位。

(2) 由于所采集的对象不同，有模拟量、数字量，而这些信号都是由现场的传感器产生的，信号不能满足要求，故必须有信号处理电路，如测量放大器、I/F 变换、A/D 转换、放大、整形电路等。

(3) 输入模块是一个模拟电路和数字电路混合系统，电路功耗低，信号功率小。

2．输出模块

输出模块是应用系统的输出通道，完成输出信号的锁定和驱动。输出信号通常有两种：开关量信号和模拟信号。开关量信号采用隔离器件进行隔离、电平转换和功率放大；模拟信号需要进行 D/A 转换、放大、功率驱动等。

输出模块的特点如下：

(1) 输出模块是应用系统的输出通道，通常需要信号锁定和功率驱动。

(2) 驱动现场，控制系统的大功率负荷易从该通道进入单片机系统，干扰系统正常工作，因此，输出信号的隔离对系统的可靠性影响很大。系统设计时必须十分重视。

(3) 根据控制对象的不同要求，电路有多种多样，有模拟电路、数字电路、开关电路等，输出信号形式有电流信号输出、电压信号输出、开关量输出及数字量输出等，视具体系统的实际情况而定。

3．人机对话模块

人机对话模块是单片机应用系统中人机之间信息交流的主要通道，是用户为了对应用系统进行干预(如启动、参数设置等)以及了解应用系统运行状态所设置的对话通道。主要有键盘、显示器、打印机等接口。

人机对话模块有以下特点：

(1) 由于常用的单片机系统大多数是小规模嵌入式系统，因此，系统中人机对话设备的配置都较小，如微型打印机、功能键、LED／LCD 显示器等。若需高水平的人机对话配置，如通用打印机、CRT、硬盘、标准键盘等，则往往将单片机系统通过外部总线与 PC 机相连享用 PC 机的外围设备。

(2) 单片机应用系统中，人机对话通道及接口大多采用内总线形式，与系统扩展密切相关。

(3) 人机对话模块接口一般都采用数字电路，结构简单，可靠性较高。

4．通信模块

单片机系统中的通信模块是解决计算机系统间相互通信的接口。在较大规模的多机测控系统中，还需要设计相互通信接口。

通信模块设计中须考虑如下问题：

(1) 单片机具有异步串行通信口，很容易实现串行通信。

(2) 单片机本身的串行口为系统通信提供了硬件支持，并没有提供标准的通信规约。因此，利用单片机串行口构成系统时，要有相应的通信规约并配置相关的通信软件。

(3) 通信接口采用数字通信，抗干扰能力强。

8.1.2 系统设计内容

单片机应用系统设计包含硬件设计与软件设计两部分。

1. 硬件设计

(1) 最小系统设计是给单片机配以必要的外围器件构成单片机最小系统。如片内有程序存储器的机型 89C51、87C51、凌阳 SPCE061A 等，只需在片外设置电源、复位电路和时钟源便构成了最小系统，可实现一些基本功能。

(2) 系统扩展设计是在单片机最小系统基础上，配置能满足应用系统要求的外围器件。

(3) 接口电路设计是根据应用系统的性质与任务，设计相应的输入隔离放大器和输出驱动电路。

(4) 通信模块设计即通信模块通常采用单片机本身提供的串行通信接口，实现与其它设备、系统之间的数据通信。

(5) 系统抗干扰设计要贯穿到设计的全过程。从具体方案、器件选择到电路设计，从硬件系统设计到软件系统设计，都要把抗干扰设计作为一项重要任务。现行的大多数单片机都具有 WatchDog 功能，防止系统死机。这往往还不够，必须有外围电路组成的系统看门狗对整个系统进行监视。

2. 应用软件设计

根据系统功能要求，设计能够满足系统功能的程序，目前大多采用单片机编程语言或 C 语言设计程序。

8.2　系统开发过程

通常，开发一个单片机应用系统需要经历六个过程。

1. 需求分析与方案调研

在确定开发课题后，首先要进行系统需求分析与调研。目的是通过市场调研明确系统的设计目标及目前相关产品的性能、优缺点、发展方向及技术指标等。方案调研包括查找资料、分析研究，并解决以下问题：

(1) 了解国内外同类系统的现状和发展方向、新器件及供应状况；对接收委托研制项目，应充分了解对方的技术要求、环境状况及技术水平，以便明确课题的技术难度。

(2) 了解软、硬件技术支持，能够移植的尽量移植，避免低水平重复劳动。

(3) 分析软、硬件技术难度，明确技术主攻方向。

(4) 综合考虑软、硬件分工。单片机应用系统设计中，软、硬件工作是密切相关的，在系统设计时要综合考虑。

(5) 根据任务要求，选择合适的传感器和执行部件。

经过需求分析与方案调研，整理出需求分析与方案设计报告，将其作为系统可行性分析的主要依据。

2. 可行性分析

可行性分析的目的是对系统开发研制的必要性及可行性做出明确的结论，根据这一结论决定系统的开发研制工作是否继续进行下去。

通常可行性分析要从以下几个方面进行论证：

(1) 市场或用户的需求情况分析。

(2) 经济效益和社会效益分析。

(3) 技术支持与开发环境。

(4) 现在的竞争力与市场前景。

(5) 新技术的发展方向。

3. 方案设计

系统方案设计是依据市场调研结果、用户需求情况、应用系统环境状况及关键技术支持等，来设计系统功能和系统结构的，选择实现方法。

系统功能设计包括系统总体目标功能的确定及系统软、硬件模块功能的划分与协调。

系统结构设计是根据系统软、硬件功能的划分及其协调关系，确定系统硬件结构和软件结构。系统硬件结构设计包括单片机扩展方案和外围设备的配置及其接口电路，最后要以逻辑框图形式描述。系统软件结构设计是确定系统软件功能模块的划分及各功能模块程序实现的技术和方法，最后以结构框图或流程图形式确定下来。

本阶段的工作是为整个应用系统实现建立一个框架，即建立系统的逻辑模型，是系统设计的基础和前提。因此，这项工作必须从整体出发，从系统观念出发，放眼全局，每一步都要十分仔细、周密考虑，尽可能将工作内容具体化，各个模块功能划分要详细，只有这样，才不致于在系统设计时出现重大错误。

4. 样机研制

系统详细设计与制作就是将前面的系统方案付诸实施，将硬件框图转化成具体电路，并制作成电路板，将软件框图或流程图用程序实现，并生产样机。这是系统设计的主要工作阶段，要合理调配，做到分工协作，多头并进，力争软、硬件各个环节能同时完成。

5. 系统调试

系统调试是检测所设计系统的正确性与可靠性。单片机应用系统设计是一个复杂过程，在设计、制作中，难免存在一些局部性问题或错误，系统调试可发现存在的问题和错误，并及时进行修正。调试与修改的过程可能要反复多次，直到系统调试成功运行，达到设计要求。系统软、硬件调试通过后，把联机调试完毕的系统软件固化在 EPROM 中，脱机(脱离开发系统)运行、测试性能。

6. 批量生产

在真实环境或模拟环境下运行，经反复测试运行正常，开发过程即告结束。这时的系统作为样机系统，加上外壳、面板，再配上完整的文档资料，就生成了正式系统(或产品)而被投入批量生产，服务社会。

总之，在整个开发过程中，系统调研、方案论证要仔细周密，应多花点时间，而在方案实施阶段，则应抓紧时间，分工协作，同步进行。

8.3 系统设计方法

在单片机系统开发过程中，我们只介绍了每一个开发阶段的主要任务，但每个阶段还有许多更详细的设计内容。本节主要介绍有关详细设计的内容以及常用的方法。

8.3.1 确定系统功能

由前面的分析可知，系统由若干个功能模块组成，而每一模块又可分为若干个子模块。比如，数据采集可分为模拟信号采集与数字信号采集，两者在硬件支持与软件控制上是有明显差异的。数据处理可分为预处理、功能性处理、抗干扰处理等子功能，而功能性处理还可以继续划分为各种信号处理等。输出信号按控制对象不同可采用多种方式，如继电器控制、D/A 转换控制、数码管显示等。

在确定了系统的各种功能之后，就应确定各个功能的实现途径，即哪些功能由硬件完成，哪些功能由软件完成。这就是系统软、硬件功能划分。

系统性能主要由精度、速度、功耗、体积、重量、价格和可靠性等技术指标来衡量。系统研制前，要根据需求调研结果给出上述各个指标值。一旦这些指标被确定下来，整个系统将在这些指标限定下进行设计。系统的速度、体积、重量、价格、可靠性等指标会左

右系统软、硬件功能的划分。系统功能用硬件实现时，可提高工作速率，但体积、重量、功耗、硬件等成本都相应地增大，而且还增加了硬件所带来的不可靠因素。用软件实现时，可使系统体积、重量、功耗、硬件等成本降低，并可提高硬件系统的可靠性，但是可能会降低系统的工作速率，延长开发周期等。因此，在进行系统功能的软、硬件划分时，一定要依据系统性能指标综合考虑。

8.3.2　确定系统结构

单片机应用系统结构一般是以单片机为核心的。系统中单片机选型、存储空间分配(存储器分配、通道划分)、I/O 方式及系统软、硬件功能划分都会对系统结构产生直接影响。

1．单片机选型

不同系列、不同型号的单片机内部结构和外部总线特征均不同。应用系统中的单片机是系统的核心，直接决定系统的总体结构。因此，在确定系统基本结构时，首先要选择单片机，确定采用哪个系列或型号的单片机。选择单片机应考虑以下几个主要因素：

(1) 单片机性价比。根据应用系统的要求和各种单片机的性能，选择最容易实现产品技术指标的机型，而且能达到较高的性能价格比。性能选择得过低，将给构成系统带来麻烦，甚至不能满足要求；性能选得过高，就可能大材小用，造成浪费，有时还会带来问题，使系统复杂化。

(2) 开发周期。选择单片机时，要考虑具有新技术的新机型。更重要的是应考虑选用技术成熟，有较多软件支持，可得到相应单片机的开发工具，比较熟悉的机型。这样可借鉴许多现成的技术，移植成熟软件，以节省人力、物力及缩短开发周期，降低开发成本，使所开发系统更具竞争力。

总之，对单片机芯片的选择绝不是传统意义上的器件选择，它关系到系统的整体方案、技术指标、功耗、可靠性、外设接口、通信方式、产品价格等，是系统设计过程中十分重要的一环。设计者必须反复推敲、慎重选择。

2. 存储空间分配

存储空间分配既影响单片机系统硬件结构，也影响软件的设计及系统调试。不同系列的单片机具有不同的存储空间。单片机存储器有两种基本结构：一种是在通用微型计算机中广泛采用的将程序和数据合用一个存储器空间的结构，称为普林斯顿(Princeton)结构；另一种是将程序存储器和数据存储器截然分开、分别寻址的结构，称为哈佛(Harvard)结构。MCS−51 单片机的程序存储器与数据存储器空间相互独立，工作寄存器、特殊功能寄存器、与内部数据存储器共享内部存储空间，外部 I/O 端口与数据存储器共享外部空间。8098 单片机的片内 RAM 程序存储区、数据存储区、I/O 端口全部使用同一个存储空间。SPCE061A 单片机是 SRAM 与 I/O 端口、FLASH ROM 共用一个存储空间，分段使用。

总的来说，大多数单片机都存在不同类型的器件共享同一个存储空间的问题。因此，在系统设计时就要合理分配有效的地址空间，并确定相应的译码电路，使 CPU 能准确地访问指定部件。

3. I/O 方式

采用不同的 I/O 方式，对单片机应用系统的软、硬件要求是不同的。在单片机系统中，常用的 I/O 方式主要有无条件传送方式(程序同步方式)、查询方式和中断方式。这三种方式对硬件要求和软件结构各不相同，而且特点鲜明，应根据系统的实际需要和综合性能评价来确定选用哪一种方式。一般来说，无条件传送方式适用于简单的外设接口，如 LED 显示器、继电器等；中断处理效率较高，但硬件较复杂；查询方式硬件简单，但速率较慢，占用 CPU 时间长。随着单片机中断资源越来越丰富，所需硬件越来越简单，通常应尽量选用中断方式和无条件传送方式。只有在一些低速、简单的控制系统中才采用查询方式。

4. 系统软、硬件功能划分

单片机系统的软件和硬件在逻辑上是等效的。同一系统完全可以由不同的软、硬件组合来完成，一些硬件电路实现的功能可以由软件来实现，同样，软件功能也可以由硬件电路来实现。在系统设计时，软、硬件功能的划分要根据具体任务来确定。一般来讲，多用硬件可以提高速度、减少存储量和软件研制的工作量，但会增加硬件成本，降低硬件的利用率和系统的灵活性。相反，若多用软件实现系统功能，可以节省硬件开支、提高系统的灵活性、便于系统升级，但也会增加软件开发工作量、延长开发周期、增加系统存储量等。因此，在总体设计时，必须权衡利弊，仔细划分，确定系统中软、硬件的功能。

8.3.3 系统设计原则

1. 硬件系统设计原则

一个单片机应用系统的硬件电路设计包括两部分：一是单片机系统扩展，即单片机内部的功能单元(如程序存储器、数据存储器、I/O 端口、定时器/计数器及中断系统等)的容量不能满足系统的要求时，必须在片外进行扩展，选择适当的芯片，设计相应的扩展电路；二是系统配置，即按照系统功能要求配置外围设备(如键盘、显示器、打印机、A/D 转换器及 D/A 转换器等)，要设计合适的滤波器、接口驱动电路。

系统扩展应遵循下列原则：

(1) 为了硬件系统设计标准化、模块化，尽可能选择符合单片机系统常规用法的通用电路。

(2) 系统的扩展与外设的配置应充分满足系统当前的功能要求，并留有适当余量，便于以后进行功能的扩充。

(3) 硬件结构应结合应用软件方案一并考虑。硬件结构与软件方案会产生相互影响，考虑的原则是，软件能实现的功能尽可能由软件实现，即尽可能地用软件代替硬件，以简化硬件结构，降低成本，提高可靠性。但必须注意，由软件实现的硬件功能，其响应时间要比直接用硬件长。因此，应综合考虑系统响应速度、实时要求等相关的技术指标。

(4) 整个系统中相关的器件要尽可能做到性能匹配。例如，选用晶振频率较高时，应选择响应速度较快的芯片；选择 CMOS 芯片单片机构成低功耗系统时，系统中的所有芯片都应该选择低功耗产品。如果系统中相关的器件性能差异较大，系统综合性能将降低，甚至不能正常工作。

(5) 可靠性及抗干扰设计是硬件设计中相当重要的一环，它包括芯片、器件选择、滤波器、印刷电路板布线、I/O 端口隔离等。如果设计中只注重功能实现，而忽视可靠性及抗干扰设计，到头来可能是事倍功半，甚至会造成系统崩溃，前功尽弃。

(6) 单片机外接电路较多时，必须考虑其驱动能力。驱动能力不足时，系统工作不可靠。解决的办法是增加驱动能力，利用总线驱动器或者减少芯片功耗以降低总线负载。

(7) 为了提高可靠性，系统设计时尽可能地选用简单 I/O 端口芯片，避免采用功能较复杂的、本身需要编程的器件。

(8) 原则上不提倡将单片机系统扩展得很大，以简洁实用为目的。

2. 系统软件设计

系统软件是根据系统功能要求设计的，应可靠地实现系统的各种功能。应用系统种类繁多，软件功能各不相同，但是一个优秀的应用系统软件应具有以下特点：

(1) 软件结构清晰、简洁，流程合理。

(2) 各功能程序模块化、系统化，既便于调试、连接，又便于移植、修改和维护。

(3) 程序存储区、数据存储区规划合理，既能节约存储空间，又能给程序设计与操作带来方便。

(4) 运行状态实现标志化管理。各个功能程序运行状态、运行结果以及运行条件都设置标志位以便查询，程序的转移、运行、控制通过状态标志控制。

(5) 经过调试修改后的程序应进行规范化，除去修改“痕迹”。规范化的程序便于交流和借鉴，也为今后的软件模块化、标准化打下基础。

(6) 实现全面软件抗干扰设计。软件抗干扰是提高系统可靠性的有力措施。

(7) 为了提高运行的可靠性，在应用软件中设置自诊断程序，在系统运行前先运行自诊断程序，检查系统各特征参数是否正常。

(8) 程序有必要的注释，便于阅读和交流。

8.3.4　系统硬件设计

单片机系统硬件设计是围绕单片机功能扩展和外围设备配置而展开的。

1. 程序存储器扩展

若单片机片内无程序存储器或存储容量不足时，需外部扩展程序存储器。外部扩展的存储器通常选用 EPROM 或 EEPROM。EPROM 集成度高、价格便宜，而 EEPROM 编程方便、容易。当需扩展的存储器容量较小时，使用 EEPROM 较方便；当容量较大时，采用 EPROM 更经济。

2. 数据存储器扩展

数据存储器利用 RAM 构成。大多数单片机都提供了小容量的片内数据存储区，当片内数据存储区不够用时就要扩展外部数据存储器。

存储器的设计原则是容量满足要求，适当留有余量。在存储容量满足要求的前提下，尽可能减少存储芯片的数量。建议使用大容量的存储芯片以减少存储器芯片数目。

3. I/O 端口

由于外设多种多样，使得单片机与外设之间的端口电路也各不相同。因此，I/O 端口常常是单片机应用系统设计过程中较复杂也是较困难的部分之一。

I/O 端口大致可归类为并行端口、串行端口、数字量端口和模拟量端口等。目前，有些单片机已将上述各端口集成在单片机内部，使 I/O 端口的设计大大简化。系统设计时，可以直接选择含有所需端口的单片机。如 SPCE061A 单片机本身提供了 32 路并行 I/O 端口，并且这些端口均可由指令设置为上拉、下拉、悬浮输入或输出状态，而不需附加外围电路，使用十分方便。

4. 译码电路

当需要外部扩展电路时，就需要设计译码电路。译码电路要尽可能简单，一般情况下，单片机应用系统选用现有的集成译码器。考虑到修改方便与保密性强，译码电路除了可以使用常规的译码器外，还可以利用只读存储器、GAL、CPLD 设计等形式。

5. 总线驱动器

如果单片机外部扩展的器件较多，负载过重，就要考虑增加总线驱动器。如 MCS-51 单片机的 P0 口负载能力为八个 TTL 芯片，P2 口负载能力为四个 TTL 芯片，如果 P0、P2 实际连接的芯片数目超出上述定额，就必须在 P0、P2 口增加总线驱动器来提高它们的驱动能力。P0 口应使用双向数据总线驱动器(如 74LS245)，P2 口可使用单向总线驱动器(如 74LS244)。当然，我们不提倡将一个单片机系统扩展得很大，确实由于功能很多，一个单片机无法完成时，可采用模块结构，由若干个单片机组成多机系统共同完成。

6. 抗干扰设计

由于单片机系统大多是嵌入到其它应用系统中使用，工作环境比较复杂，电磁干扰严重，因此，系统的可靠性设计就显得十分重要。针对可能出现的各种干扰，应设计抗干扰电路，增强抗干扰措施。在系统中，一个不可缺少的抗干扰电路就是抗电源干扰电路。最简单的方法是在系统弱电部分(以单片机为核心)的电源入口对地跨接一个大电容(470 μF 左右)与一个小电容，在系统内部芯片的电源端对地跨接 1 个小电容。

另外，可以采用隔离放大器、光电隔离器件抗共地干扰，采用差分放大器抗共模干扰，采用平滑滤波器抗白噪声干扰，采用屏蔽手段抗辐射干扰等等。

要注意的是，在系统硬件设计时，要尽可能充分利用单片机的片内资源，电路要标准化、模块化。硬件设计完成后，应编写出硬件电路原理图及硬件设计说明书，作为调试依据和备案资料。

8.3.5 系统软件设计

整个单片机应用系统是一个整体。在进行系统总体设计时，软件设计和硬件设计应统一考虑，相互结合，取长补短。当系统的硬件电路定型后，软件的任务也随之明确了。

一个应用系统中的软件一般是由监控程序和应用程序两部分构成的。监控程序是控制系统按预定操作方式运行的程序，它负责组织调度各应用程序模块，完成系统自检、初始化、键盘命令处理、端口命令处理、事件触发处理和显示等功能。此外，监控程序还要监

视系统的正常运行状况；应用程序是用来完成诸如测量、计算、显示、打印、输出控制等各种实质性功能的软件。单片机应用系统中的软件一般是用汇编语言(或 C 语言)编写的，编写程序时常常与 I/O 端口设计和存储器扩展交织在一起。因此，软件设计是系统开发过程中十分重要的任务，它直接关系到系统的性能。

系统软件设计时，应根据系统软件功能要求，将系统软件划分成若干个相对独立的子模块，并根据它们之间的联系和时间上的关系，设计出合理的总体结构。通常在编写程序前，先根据系统输入和输出变量建立起正确的数学模型，然后画出程序流程图。要求流程图结构清晰、简洁、合理、功能模块化，画流程图时还要对系统资源作具体的分配和说明。编写程序时一般采用自顶向下的程序设计技术，先设计监控程序，再设计各应用程序模块。各功能程序应模块化，子程序化，这样不仅便于调试、连接，还便于修改和移植，有利于程序优化。

通常在系统方案确定之后，软件设计和硬件设计就要同时进行，软件可通过开发系统提供的模拟仿真软件调试。

8.3.6　资源分配

合理的资源分配涉及到能否充分发挥单片机性能，有效、正确地运行程序的重要工作内容。

单片机系统所拥有的硬件资源分为片内和片外两部分。片内资源是指单片机本身包括的中央处理器、程序存储器、数据存储器、定时器/计数器、中断系统、I／O 端口以及串行通信口等。这部分硬件资源的种类和数量不同类型的单片机之间差别很大，当设计人员选定某种型号的单片机进行系统设计时，应充分利用片内的各种硬件资源。但是在应用系统中，片内的这些硬件资源不够使用时，才需要在片外加以扩展。

由于定时器/计数器、中断源等资源的分配比较容易，鉴于篇幅有限，不再赘述。下面介绍 ROM/EPROM 资源和 RAM 资源的分配。

1．ROM/EPROM 资源的分配

ROM/EPROM 用于存放程序和数据表格。按照单片机的复位及中断入口的规定，某些地址单元作为中断向量、复位入口地址。当程序存储器中存放的程序及子程序数量较多时，应尽可能为它们设置入口地址表。一般的常数、表格集中设置在表格区，二次开发扩展区尽可能留在高段地址区域。

2．RAM 资源分配

RAM 分为片内 RAM 和片外 RAM。片外 RAM 的容量比较大，通常用来存放批量较大的数据；片内 RAM 容量较小，应充分使用，通常用来存放标志、计数指针和常用变量等。

8.4　系 统 调 试

系统调试是开发过程的重要环节。当完成了系统的软、硬件设计和硬件组装后，便可进系统调试阶段。系统调试的目的是要查出用户系统中硬件设计与软件设计中存在的错误

及可能出现的不协调问题，以便修改设计，使系统能正常工作。

应该在方案设计阶段就要考虑系统调试问题，如采取什么调试方法、使用何种测试仪器等，以便在方案设计时将必要的调试方法综合到软、硬件设计中，提早做好调试准备工作。

系统调试包括软件调试、硬件调试及系统调试。根据调试环境不同，系统调试可分为模拟调试与现场调试。各种调试所起的作用不同，所处的时段也不一样，但目标是一致的，都是为了查出系统中的潜在错误，提高可靠性，完善系统功能。

8.4.1 常用工具

在单片机应用系统调试中，最常用的调试工具有以下六种。

1．单片机开发系统

单片机开发系统(又称仿真器)的主要作用是：

(1) 系统硬件电路的诊断与检查。

(2) 程序的输入与修改。

(3) 硬件电路、程序的运行与调试。

(4) 程序固化到 EPROM 中(要用到编程器)。

由于单片机本身不具有调试及输入程序的能力，因此，单片机开发系统就成为开发应用系统不可缺少的工具。

通常，一种开发系统只适宜开发同一系列的单片机，例如，Intel 公司推出的 ICE−5100/252 单片机在线仿真器，用于开发研制 MCS−51 系列单片机应用系统；ATD−96/98B 单片机开发系统用于开发研制 8096/98 单片机应用系统。Motorola 公司推出的 MC68HCllEVM 仿真器用于开发研制 MC68HCll 系列单片机应用系统。凌阳公司提供的 PROBEL 在这方面有所发展，可以很容易地进行系统仿真和调试，并能完成程序下载，实现器件编程。也有少数的开发系统适用于多种类型的单片机，但一般局限于同一公司的产品。因此，单片机开发系统的选择原则是开发系统的单片机与用户系统的单片机必须为同一系列产品。

开发系统可独立工作，也可与通用计算机联机使用。它提供了必要的开发软件及丰富的子程序库，其监控程序支持程序输入、修改、测试、状态查询等功能；占用单片机硬件资源少并具有资源出借功能；具有多种跟踪、运行、调试功能，在与主机联机时可用汇编语言或 C 语言编程。

将单片机开发系统的仿真插头插入用户系统的单片机插座，通过操作开发系统实现对用户系统各部件的操作，就能达到调试、运行用户系统的目的。

2．万用表

万用表(三用表)主要用于测量硬件电路的通断、两点间电阻值、测试点处稳定电流或电压及其它静态工作状态等。系统研发、调试、维护人员应配备一块内阻较大的数字三用表。

例如，当在某个集成芯片的输入端输入稳态电平时，可用万用表来测试其输出，通过测试值与预期值的比较，就可大致判定该芯片的工作是否正常。如 74LS04 六非门器件，

当在非门输入端加高电平时，其输出端应为低电平，可用三用表测试其电平，判断其工作状态。

3. 逻辑笔

逻辑笔是数字系统调试过程中十分有用的测试工具，它以制作容易、携带方便和经济实用深受广大工程技术人员的欢迎。

逻辑笔可以测试数字电路中被测试点的电平状态(高或低)及是否有脉冲信号。假如要检测单片机扩展总线上连接的某译码器是否有译码信号输出，可编写一循环程序使译码器对一特定译码状态不断进行译码。运行该循环程序后，用逻辑笔测试译码器输出端，若逻辑笔上发光二极管交替闪亮，则说明译码器有译码信号输出；若只有红色发光二极管亮(高电平输出)或绿色发光二极管亮(低电平输出)亮，则说明译码器无译码信号输出。这样就可以初步确定由扩展总线到译码器之间是否存在故障。系统运行状态时，测量地址总线、数据总线的状态都很有用。

4．逻辑脉冲发生器与模拟信号发生器

逻辑脉冲发生器能够产生不同宽度、幅度及频率的脉冲信号，它可以作为数字电路的输入源。模拟信号发生器可产生具有不同频率的方波、正弦波、三角波、锯齿波等模拟信号，作为模拟电路的输入源。这些信号源在模拟调试中十分有用。

5．示波器

示波器可以测量电平、模拟信号波形及频率，还可以同时观察多个信号的波形及它们之间的相位关系(双踪或多踪示波器)。它既可以对静态信号进行测试，也可以对动态信号进行测试，而且测试准确性好。它是任何电子系统调试维修的一种必备工具。

6．逻辑分析仪

逻辑分析仪能够以单通道或多通道实时获取触发事件的逻辑信号，可保存显示触发事件前后所获取的信号，供操作者随时观察，作为软、硬件分析的依据，能够快速有效地查出软、硬件中的错误。逻辑分析仪主要用于动态调试中信号的捕获。

在单片机应用系统调试过程中，数字三用表、示波器、逻辑笔及开发系统是最基本的调试工具。

8.4.2　调试方法

1．硬件调试

硬件调试是利用开发系统、基本测试仪器(万用表、示波器等)，通过执行开发系统有关命令或运行适当的测试程序(也可以是与硬件有关的用户程序段)，检查用户系统硬件工作是否正常。

硬件调试可分静态调试与动态调试两步进行。

1) 静态调试

静态调试是在用户系统未加电工作时的一种硬件检查。具体步骤如下：

(1) 目测。单片机应用系统中大部分电路安装在印制电路板上，因此对每一块加工好

的印制电路板要进行仔细的检查。检查它的印制线是否有断线、有无毛刺、是否与其它线或焊盘粘连、焊盘有无脱落、金属化过孔是否连通等。如印制板无质量问题，则将集成芯片的插座、电路元件焊接在印制板上(注意：装配工艺一般先低后高。先焊体积小、高度低的元件，后焊体积大、较高的元件；先装低值元器件，后装价格较高的元件)，并检查其焊点是否有毛刺，是否与其它印制线或焊盘连接，焊点是否光亮饱满，有无虚焊等。对系统中所用的器件与设备，要仔细核对型号，检查它们对外连线(包括集成芯片引脚)是否完整无损。通过目测查出一些明显的器件、设备故障并及时排除。

(2) 万用表检测。目测检查后，可用万用表测试。先用万用表复核目测过程中认为可疑的连接或接点，检查它们的通断状态是否与设计相符。再检查各种电源线与地线之间是否有短路现象，仔细查出并排除(注意：短路现象一定要在器件安装及加电前排除)。如果电源与地之间短路，系统中所有器件或电源设备都可能被毁坏，后果十分严重。所以，对电源与地的处理，在整个系统调试及今后的运行过程中都要相当小心。

如有现成的集成电路性能测试仪器，此时应将要使用的芯片进行测试筛选，其它的器件、设备在购买或使用前也应做必要的测试，以便将性能可靠的器件、设备用于系统。

(3) 加电检查。在静态检测完成之后，当给印制板加电前，首先检查所有插座或器件的电源额定值是否与电源电压相符，然后再加电检查各点电压是否正常(注意：单片机插座上的电压不应该大于 VDD，否则联机时将损坏仿真器)，接地端电压值是否接近于零，接固定电平的引脚端电平是否正确。而后在断电状态下，将芯片逐个插入印制板上的相应插座。每插入一块做一遍上述的检查，特别要检查电源到地是否短路，这样就可以确定电源错误或与地短路发生在哪块芯片上。

在对各芯片、器件加电过程中，还要随时注意观察芯片或器件是否出现打火、过热、变色、冒烟、异味等现象，如果出现，立即断电，仔细检查电源加载等情况，找出产生异常的原因并加以解决。

此外，也可以在加电期间，通过给逻辑功能简单的芯片加载固定输入电平，用万用表测其输出电平的方法来判定该芯片的好坏。如 74LS08 为四二输入与门，当两输入端接高电平时，其输出端应为高电平；任一输入端为低电平时，输出应为低电平，否则说明该器件有问题。

(4) 联机检查。由于只有用单片机开发系统才能完成对用户系统的调试，而动态测试也需要在联机仿真的情况下进行。因此，在静态检查印制板、连接、器件等部分无物理性故障后，即可将用户系统与单片机开发系统用仿真电缆连接起来，联机检查上述连接是否正确、可靠。

2) 动态调试

动态调试是在用户系统工作的情况下，发现和排除系统硬件中存在的器件内部故障、器件间连接逻辑错误的一种检查方法。由于单片机应用系统的硬件动态调试是在开发系统的支持下完成的，因此又称为联机仿真或联机调试。

动态调试一般采用由分到合、由近及远的方式进行。

由分到合指的是，首先按逻辑功能将用户系统硬件电路分为若干模块，如程序存储器、A/D 转换电路、继电器控制电路等，再分别调试。当调试某一模块电路时，与该电路无关的器件可以全部从用户系统中去掉，这样，可将故障范围限定在某个局部范围内。当各个

电路调试无误后，将各模块电路逐一加入系统中，再对电路功能及各电路间可能存在的相互联系进行调试。此时若出现故障，最大可能是在各电路协调关系上出了问题，如交互信息的联络是否正确，时序是否达到要求等。直到所有电路加入系统后各部分电路正确工作为止，由分到合的调试即告完成。在经历了这样一个调试过程后，大部分硬件故障基本上可以排除。

在有些情形下，由于功能要求较高或设备较复杂，使某些逻辑功能模块电路较复杂，为故障的准确定位带来一定的难度。这时，对每块电路可以以信号处理的流向为线索，将信号流经的各器件按照距离单片机的逻辑距离进行由近及远的分层，然后分层调试。调试时，仍采用去掉无关器件的方法，逐层依次调试下去，就可将故障定位在具体器件上。例如，调试外部数据存储器时，可按层先调试总线电路，然后调试译码电路，最后加上存储芯片，利用开发系统对其进行读写操作，就能有效地调试数据存储器。显然，每部分出现的问题只局限在一个小范围内，有利于故障的发现和排除。

动态调试借用开发系统资源(单片机、存储器等)来调试用户系统中单片机的外围电路。利用开发系统友好的人机界面，可以有效地对用户系统的各部分电路进行访问、控制，使系统在运行中暴露问题，从而发现故障。典型有效的访问、控制各部分电路的方法是对电路进行循环读或写操作(时钟等特殊电路除外，这些电路通常在系统加电后会正常运行，只要器件没问题，一般情况下为免调试电路)，使得电路中主要测试点的状态能够用常规检测仪器(如示波器、万用表等)测试，依次检测被调试电路是否按预期的状态工作。

2．软件调试

软件调试是通过对用户程序的汇编、连接、执行来发现程序中存在的语法错误与逻辑错误并加以排除的过程。软件调试一般采用先独立后联机、先分块后组合、先单步后连续的调试方法。

1) 先独立后联机

从宏观来说，单片机应用系统中的软件与硬件是密切相关、相辅相成的。软件是硬件的灵魂，没有软件，系统将无法工作；同时，软件的运行又依赖于硬件，没有硬件支持，软件的功能便荡然无存。因此，将两者完全孤立开来是不可能的。然而，并非所有用户程序都依赖于硬件，当软件对被测试参数进行加工处理或做某项事务处理时，往往与硬件无关，这样，就可以通过分析用户程序，把与硬件无关的功能相对独立的程序段抽取出来，形成与硬件无关和依赖于硬件的两大类用户程序块。这一划分工作在软件设计时就应充分考虑。

在具有交叉汇编软件的主机或与主机联机的仿真机上，此时与硬件无关的程序块调试就可以与硬件调试同步地进行，或借助开发机的仿真功能进行调试，以提高软件调试的速度，缩短系统开发周期。

当程序仿真调试与用户硬件系统调试完成后，可将仿真机、计算机和用户系统连接起来进行系统联调。在系统联调中，先对依赖于硬件的程序块进行调试，调试成功后，再将两大程序块组合在一起调试。

2) 先分块后组合

当用户系统规模较大、任务较多时，即使先行将用户程序分为与硬件无关和依赖于硬

件两大部分，但这两部分程序仍较为庞大的话，若从头至尾调试，既费时间又不容易进行错误定位。所以常规的调试方法是分别对两类程序模块进一步划分，然后分别调试，以提高软件调试的有效性。在调试时所划分的程序模块应基本保持与软件设计时的程序功能模块或任务一致。除非某些程序功能块较大时才将其再细分为若干个子模块。但要注意的是，子模块的划分与一般模块的划分应一致。

每个程序模块调试完成后，将相互有关联的程序模块逐块组合起来加以调试，以解决在程序模块连接中可能出现的逻辑错误。对所有程序模块的整体组合是在系统联调中进行的。由于各个程序模块通过调试已排除了内部错误，所以软件总体调试的错误就大大减少了，能够在较短的时间内完成软件调试。

3) 先单步后连续

调试好程序模块的关键是实现对错误的正确定位。准确发现程序(或硬件电路)中错误的有效方法是采用单步加断点运行方式调试程序，单步运行可以了解被调试程序中每条指令的执行情况，分析指令的运行结果可以知道该指令执行的正确性，并进一步确定是由于硬件错误、数据错误还是程序设计错误等引起了该指令的执行错误，从而发现并排除故障。

但是，所有程序模块都以单步方式查找错误的话，实在是一件既费时又费力的工作，而且对于一个好的软件设计人员来说，设计错误率是较低的。所以，为了提高调试效率，通常采用先使用断点运行方式将故障定位在程序的一个小范围内，然后针对故障程序段再使用单步运行方式来确定错误位置所在，这样就可以做到调试的快捷和准确。一般情况下，单步调试完成后，还要做连续运行调试，以防止某些错误在单步执行的情况下被掩盖。有些实时性操作(如中断等)利用单步运行方式无法调试，必须采用连续运行方法进行调试。为了准确地对错误进行定位，可使用连续加断点运行方式调试这类程序，即利用断点定位的改变，一步步缩小故障范围，直至最终确定出错误位置并加以排除。

3．系统联调

系统联调是指将应用系统的软件在其硬件系统上实际运行，进行软、硬件联合调试，发现硬件故障或软、硬件设计错误，这是对用户系统检验的重要的一环。系统联调主要解决以下几个问题：

(1) 软、硬件能否按设计要求配合工作? 如果不能，那么问题出在哪里? 如何解决?

(2) 系统运行中是否有潜在设计时难以预料的错误? 如硬件延时过长造成工作时序不符合要求，布线不合理造成串扰等。

(3) 系统的动态性能指标(包括精度、速度参数)是否满足设计要求?

系统联调时，首先采用单步、断点、连续运行方式调试与硬件相关的各程序段，既可检验这些用户程序段的正确性，又能在各功能独立的情况下，检验软、硬件的配合情况。然后，将软、硬件按系统工作要求进行综合运行，采用全速断点、连续运行方式进行总调试，使系统总体运行时软、硬件能协调工作以提高系统的动态性能。在具体操作中，应用系统在开发系统环境下，先借用仿真器上的单片机、存储器等资源进行工作。若发现问题，按上述软、硬件调试方法准确定位，分析错误原因，找出解决办法。用户系统调试完后，将用户程序固化到系统的程序存储器中，再借用仿真器上的单片机，使系统运行。若无问题，则用户系统插上单片机即可正确工作。

注意：不要忘记用户系统时钟、复位电路的调试。

4. 现场调试

一般情况下，通过系统联调后，用户系统就可以按照设计目标正常工作了。但在某些情况下，由于系统运行的环境较为复杂(如电磁环境干扰严重、工作现场有特种气体、工频干扰等)，在实际进入现场工作之前，环境对系统的影响无法预料，只能通过现场调试发现问题，找出相应的解决方法。或者虽然已经在系统设计时采取了抗干扰措施，是否行之有效，还必须通过用户系统在实际现场的运行来加以验证。另外，有些用户系统的调试是在用模拟设备代替实际监测、控制对象的情况下进行的，这就更有必要进行现场调试，以检验用户系统在实际工作环境中工作的正确性。必要时，在进入现场调试之前应完成例行试验。

总之，现场调试对用户系统来说是最后必须的一个过程，只有经过现场调试的应用系统才能保证其可靠地工作。现场调试仍可利用开发系统来完成，其调试方法与前述类似。

在系统设计过程中，没有一个固定不变的方法，要根据系统的实际要求，有所侧重。当然，有些过程可以简化，但请读者注意，方案论证和系统设计是必须花精力仔细研究的，不可略过。

8.5　SPCE061A 系统设计举例

前面我们介绍了单片机应用系统设计方法及调试过程，本节将结合一个实例，介绍单片机应用系统的设计方法。

8.5.1　市场调研与功能确定

利用 SPCE061A 单片机设计一个数码录音笔。

目前，数码录音笔的种类很多。它们具有如下功能：闹钟、定时录音、重复播放、播放速度可控、资料上传和资料下载等功能。此外，有些数码录音笔还具有一些特殊功能，如视像、播放 MP3、录音监听等功能。

本节通过一个实例介绍单片机应用系统的开发过程和方法，设计录音笔的功能只有录音、播放、循环播放、重复播放、选择上一段播放、选择下一段播放、上传数据和下载数据等功能。

8.5.2　硬件设计

本系统硬件涉及到电源管理、音频录入和键盘管理，有外扩存储器、通信的接口和音频输出等几部分，如图 8.2 所示。

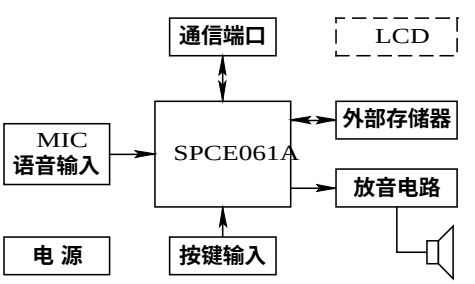


图 8.2　系统结构框图

以下我们对各部分电路作个简要介绍。

1. 电源

图 8.3 是电源部分的电路，5 V 直流电压经过 SPY0029 后给整个系统供电，当然也

可选用其它芯片。SPY0029 是凌阳公司设计的电压调整 IC，采用 CMOS 工艺。SPY0029 具有静态电流低、驱动能力强、线性调整好等特点。

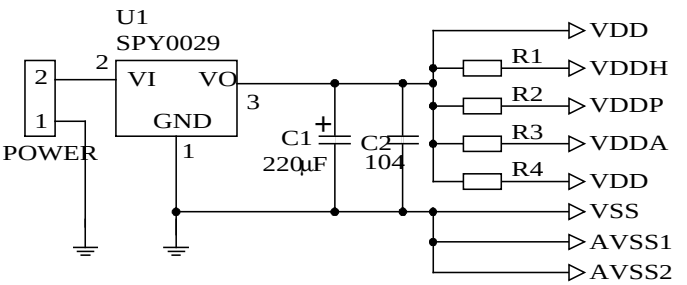


图 8.3 电源电路

图中的 VDDH 为 SPCE061A 的 I/O 参考电平，接 SPCE061A 的 51 脚，这种接法使得 I/O 输出高电平为 3.3 V；VDDP 为 PLL 锁相环电源，接 SPCE061A 的 7 脚；VDD 和 VDDA 为数字电源，分别接 SPCE061A 的 15 脚和 36 脚；AVSS1 是模拟地，接 SPCE061A 的 24 脚；VSS 是数字地，接 SPCE061A 的 38 脚；AVSS2 接音频输出电路的 AVSS2。

2. 音频录入

音频录入部分主要由 Microphone、AGC 电路和 ADC 电路构成，图 8.4 是音频录入电路。因为 SPCE061A 内置了 AGC 电路和 ADC 电路，所以实现音频录入的外围电路非常简单。外围电路与 SPCE061A 连接，AGC 接音频录入 AGC 引脚(25 脚)，OPI 接 Microphone 的第二运放输入脚(26 脚)，MICOUT 接 Microphone 的第一运放输出脚(27 脚)，MICN 接 Microphone 的负向输入脚(28 脚)，MICP 接 Microphone 正向输入脚(33 脚)，VCM 接 ADC 参考电压输出脚(34 脚)，VMIC 接 Microphone 电源(37 脚)。

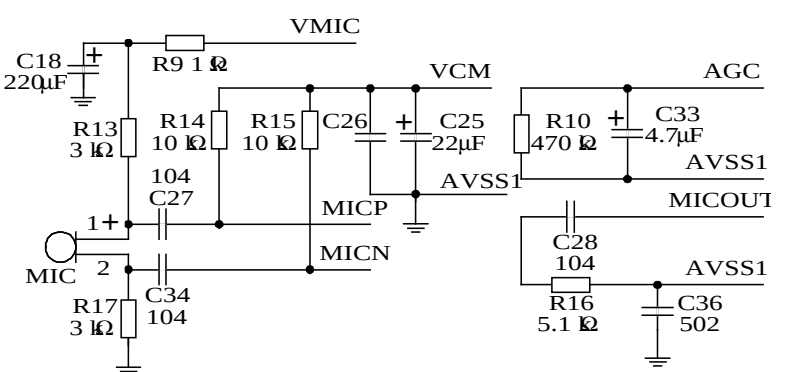


图 8.4 音频录入电路

语音信号经 Microphone 转换成电信号，由隔直电容隔掉直流成分，然后输入 SPCE061A 内部前置放大器。SPCE061A 内部自动增益控制电路 AGC 能随时跟踪前置放大器输出的音频信号电平，当输入信号增大时，AGC 电路自动减小放大器的增益；当输入信号减小时，AGC 电路自动增大放大器的增益，以便使进入 A/D 的信号保持在最佳电平，又可使失真减

至最小。

3. 键盘

图 8.5 是 4×4 键盘的电路。这里用 A 口的低 8 位，扩展了 16 个按键，按键的键值分布和对应功能如表 8.1 所示。这里只用了部分按键，其它按键可用于功能扩展。

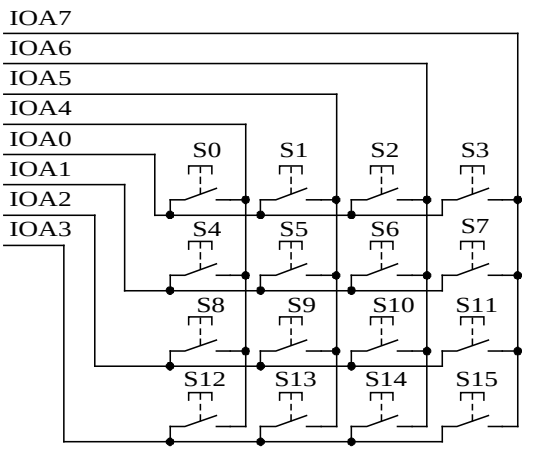


图 8.5　4×4 键盘接口电路

表 8.1　键值分布及其意义

键值	功能	键值	功能	键值	功能	键值	功能
0	播放	4	循环播放	8		12	
1	播放前段/上传	5	重复播放	9		13	
2	播放后段/下载	6		10		14	
3	停止/格式化	7	录音	11		15	开/关 UART

4. 外扩存储器

因为 SPCE061A 的 FLASH 只有 32 K 字，要存放大量的语音资源，就要外扩存储器。在此采用 SIO 扩展串行存储器的方法。本例采用凌阳公司的 SPR4096 芯片。限于篇幅，这里仅选中与我们设计程序有关的部分进行介绍。

SPR4096 是一个高性能的 4 Mb(512 K×8 bit)FLASH，分为 256 个扇区(Sector)，每个扇区为 2 KB。SPR4096 还内置了一个 4 K×8 bit 的 SRAM。在进行 FLASH 的编程/擦除的时候，可以并行执行 SRAM 的读/写。

SPR4096 内置了一个总线存储器接口和一个串行接口，它允许单片机通过 8 bit 并行模式或者 1 bit 串行模式访问 FLASH/SRAM 存储区。本例使用串行模式，串行接口的工作频率为 5 MHz。SPR4096 有两个电源输入端 VDDI 和 VDDQ，VDDI 是给内部 FLASH 和控制逻辑供电的；VDDQ 是专门为 I/O 供电的。最大读电流为 2 mA，最大编程/擦除电流为 6 mA。

图 8.6 是 SPR4096 的模块结构图。它包含了几个部分：总线存储器接口、SIO 串行接口、SRAM 编程与擦除控制器和一个 4 Mb 的 FLASH。SIO 是串行接口，SCLK 是时钟信

号线，SDA 是数据信号线。如果接收到 FLASH 的读指令或者 SRAM 的读/写指令，串行接口可以直接从内存读写数据。但是，如果接收到 FLASH 的编程或擦除指令，串行接口会把这些指令传给编程和擦除控制器，编程和擦除控制器完成相应操作。

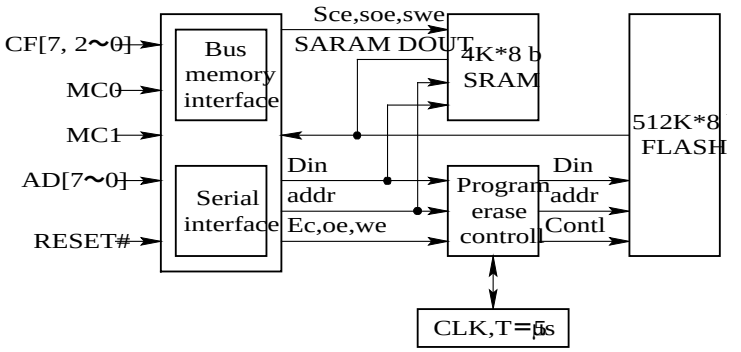


图 8.6 SPR4096 模块结构图

通过设置 CF2～CF0 选择接口通信模式。当 CF2～CF0 均接高电平时，为串行模式。CF7 为低电平时选中 FLASH，高电平时选中 SRAM。

图 8.7 是 SPR4096 和 SPCE061A 的连接图。SPR4096 的 SCLK 接 SPCE061A 的 IOB0，SDA 接 SPCE061A 的 IOB1。CF0～CF2 接高电平，选择串行接口模式；CF7 接低电平，选中 FLASH。

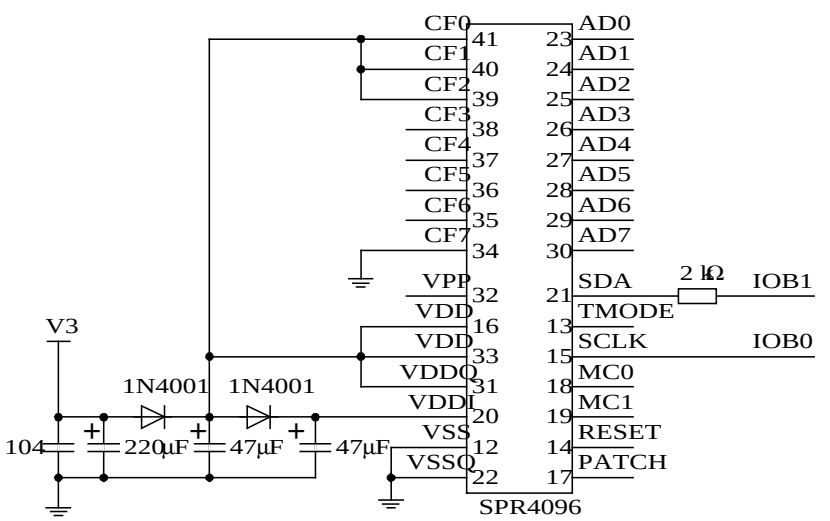


图 8.7 外扩存储器电路

5. 通信端口

使用 SPCE061A 的异步串行端口 UART 实现与 PC 通信。SPCE061A 的 I/O 电平和 PC 不一致，采用一片 MAX232 进行电平转换。电路图如图 8.8 所示，MAX232 的 RXD1 和 TXD1 分别接 SPCE061A 的 IOB10(TX)和 IOB9(RX)。

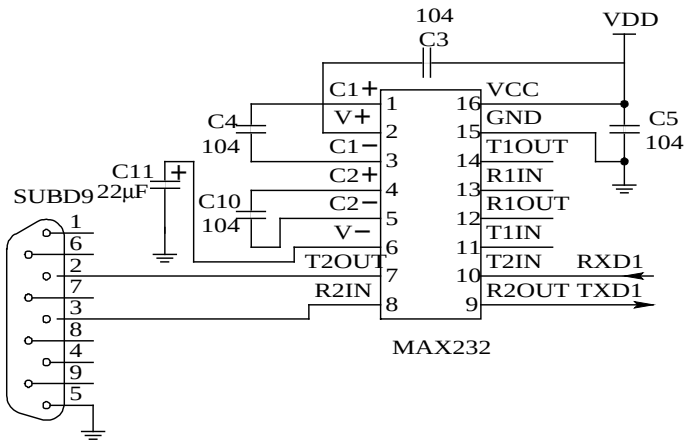


图 8.8　PC 通信端口电路

6. **音频输出部分**

SPCE061A **内置两路** 10 **位** DAC**，只需要外接功放电路即可完成语音的播放。图** 8.9 **是音频输出电路图，可以接耳机，也可以接喇叭。**

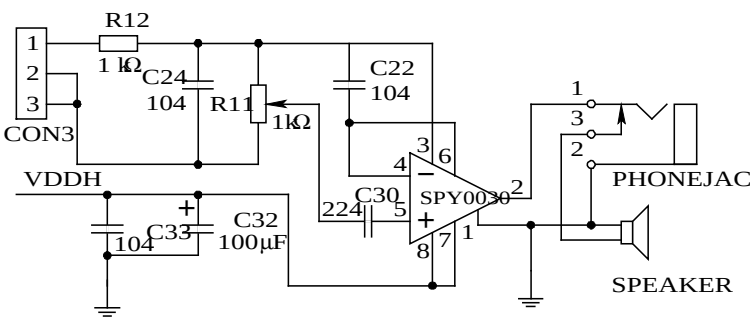


图 8.9　音频输出电路

图中采用 SPY0030 **做功率放大。其工作电压仅需** 2.4 V(**两节电池即可工作**)**；输出功率约** 700 mW**。**

8.5.3　软件设计

同硬件设计一样，软件设计也是分模块进行的。主要包括以下几部分：系统初始化程序、SPR4096 **的读写程序、内部** FLASH **的读写程序、键盘扫描程序、录放音程序、**UART **通信程序等。各部分程序由主程序**(main.c)**调用，组成一个整体。**

1. **主程序**

图 8.10 **是主程序流程图。**

系统初始化之后，就是一个循环。在这个循环里的运作可以分为三部分：键盘扫描、基状态(Status_Base)**和子状态**(Status_Sub)**。**

键盘扫描是扫描键盘提取键值并根据键值调用相应的子程序。

基状态包括 UART **开启和** UART **关闭两种情况。子状态包括录音、播放、循环播放、重复播放、下载、上传和停止等七种情况。**

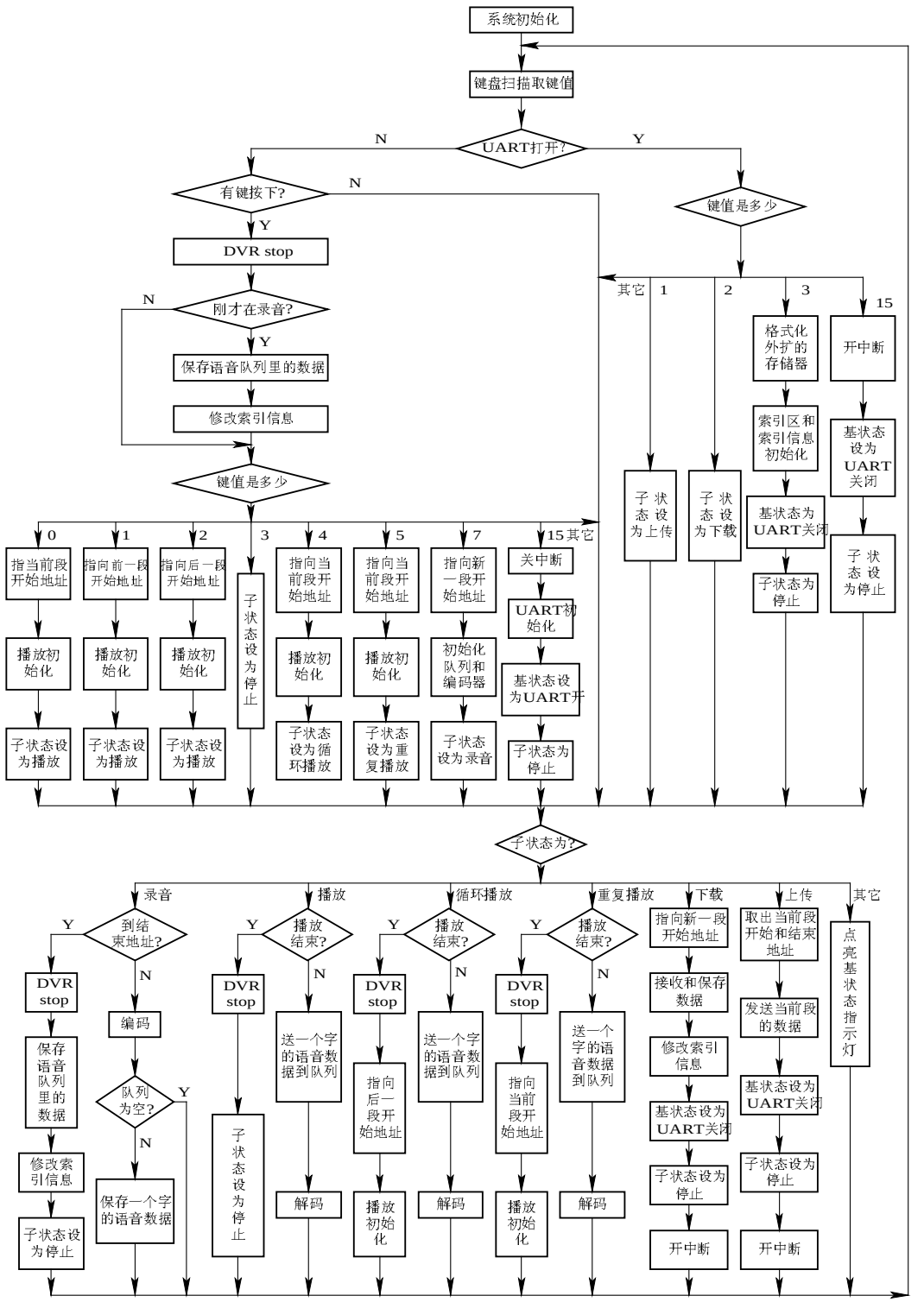

系统初始化
键盘扫描取键值
UART打开?
N
Y
有键按下?
N
Y
DVR stop
刚才在录音?
N
Y
保存语音队列里的数据
修改索引信息
键值是多少
0
1
2
3
4
5
7
15其它
指当前段开始地址
播放初始化
子状态设为播放
指向前一段开始地址
播放初始化
子状态设为播放
指向后一段开始地址
播放初始化
子状态设为播放
子状态设为停止
指向当前段开始地址
播放初始化
子状态设为循环播放
指向当前段开始地址
播放初始化
子状态设为重复播放
指向新一段开始地址
初始化队列和编码器
子状态设为录音
关中断
UART初始化
基状态设为UART开
子状态为停止
键值是多少
其它
1
2
3
15
子状态设为上传
子状态设为下载
格式化外扩的存储器
索引区和索引信息初始化
基状态为UART关闭
子状态为停止
开中断
基状态设为UART关闭
子状态设为停止
子状态为?
录音
播放
循环播放
重复播放
下载
上传
其它
到结束地址?
Y
N
DVR stop
保存语音队列里的数据
修改索引信息
子状态设为停止
编码
队列为空?
Y
N
保存一个字的语音数据
播放结束?
Y
N
DVR stop
子状态设为停止
送一个字的语音数据到队列
解码
播放结束?
Y
N
DVR stop
指向后一段开始地址
播放初始化
送一个字的语音数据到队列
解码
播放结束?
Y
N
DVR stop
指向当前段开始地址
播放初始化
送一个字的语音数据到队列
解码
播放结束?
Y
N
送一个字的语音数据到队列
解码
指向新一段开始地址
接收和保存数据
修改索引信息
基状态设为UART关闭
子状态设为停止
开中断
取出当前段开始和结束地址
发送当前段的数据
基状态设为UART关闭
子状态设为停止
开中断
点亮基状态指示灯

图 8.10　主程序流程图

系统初始化部分包括硬件初始化和变量初始化，以下为系统初始化部分的程序。

```
*************************************************************************
系统初始化部分程序
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
        BaseType = UartClose;                          //变量命名
        SubStatus = Stop;
        Segment = 1;
        Addr = 0;
        Keycode = 0xFF;                                //无键按下
        for(i=0,internal_addr=0xF000;i<21;i++,internal_addr++)
            Buffer[i] = SP_ReadWord(internal_addr);    //从目录读信息
        System_Initial();                              //系统初始化(设置 I/O 和中断)
        SACM_DVR_Initial(Manual);                      //DVR 设置为手动方式
*************************************************************************
```

索引区是 SPCE061A 内部 FLASH 从 0xF000H 开始的 21 个字。0xF000H 存放的是 SegFlag，记录的是总共有多少段数据；以下每两个字记录某段的开始地址和结束地址，比如 0xF001H 和 0xF002H 存放的是第一段的开始和结束地址，0xF003H 和 0xF004H 存放的是第二段的开始和结束地址。这里的索引区只有 21 个字，所以这能保存 10 段语音数据，如果你要记录 100 段数据的话，把索引区改为 201 个字就可以了。这里用到了函数 SP_ReadWord()，它的功能是从内部 FLASH 中读出一个字的数据，System_Initial()是硬件部分的初始化函数。

以下部分是根据键值作相应设置的程序，对照图 8.10 就容易读懂了。

```
*************************************************************************
        if(BaseType == UartClose)
        {
            if(Keycode!=0xFF)                          //有键按下否？
            {
                DVR_Stop();
                if(SubStatus == Record)
                {
                    Send_QueueData_to_Sflash();
                    Modify_Index();
                }

                switch(Keycode)
                {
                    case 0:                            //播放现行
                        Addr=Buffer[Segment*2-1];
                        Play_Initial();
```

```
        SubStatus = Play;
    break;
    case 1:                                          //播放第一段
        if((Segment>1)&&(Segment<=Buffer[0]))
        {
            Segment--;
            Addr=Buffer[Segment*2-1];
        }
        else
        {
            Segment=Buffer[0];
            Addr=Buffer[Buffer[0]*2-1];
        }
        Play_Initial();
        SubStatus = Play;
    break;
    case 2:                                          //播放第二段
        if(Segment<Buffer[0])
        {
            Segment++;
            Addr=Buffer[Segment*2-1];
        }
        else
        {
            Segment=1;
            Addr=Buffer[1];
        }
        Play_Initial();
        SubStatus = Play;
    break;
    case 3:                                          //停止状态
        SubStatus = Stop;
    break;
    case 4:                                          //全部重新播放
        Addr=Buffer[Segment*2-1];
        Play_Initial();
        SubStatus = RePlay_All;
    break;
    case 5:                                          //只重新播放现行段
```

```
                    Addr=Buffer[Segment*2−1];
                    Play_Initial();
                    SubStatus = RePlay;
                break;
                case 7:                                          //录音
                    Addr = Buffer[Buffer[0]*2+1];
                    SACM_DVR_InitQueue();
                    SACM_DVR_InitEncoder(RceMonitorOff);
                    SubStatus = Record;
                break;
                case 15:                                         //开放串行通信口
                    Close_Interrupt();
                    SP_UART_Init();
                    BaseType=UartOpen;
                    SubStatus=Stop;
                break;
                default:
                break;
            }
        }
    }

    else
    {
        switch(Keycode)
        {
            case 1:                                              //装载
                SubStatus=Upload;
            break;
            case 2:                                              //下载
                SubStatus=Download;
            break;
            case 3:                                              //串行方式
                SP_SIOMassErase();
                SP_PageErase(0xF000);
                for(i=0;i<21;i++)
                {
                    Buffer[i]=0;
                    SP_WriteWord(0xF000+i,0);
```

```
                    }
                    Segment = 1;                                    //变量初始化
                    Addr = 0;
                    BaseType = UartClose;
                    SubStatus = Stop;
                break;
                case 15:                                            //关闭串行通信口
                    Open_Interrupt();
                    BaseType=UartClose;
                SubStatus = Stop;
                break;
                default:
                break;
            }
    }
*****************************************************************************
    以下部分是根据子状态作相应操作的程序。
*****************************************************************************
        switch(SubStatus)
        {
            case Record:
                if(Addr < C_SflashSize)                             //SPR4096 没满
                {
                    SACM_DVR_Encode();                              //读数、编码
                    if(SACM_DVR_TestQueue() != Empty)               //检测排队是否空
                    {
                        Ret = SACM_DVR_FetchQueue();                //读排队器
                        SP_SIOSendAWord(Addr,Ret);                  //存储到 SPR4096
                        Addr += 2;
                    }
                }
                else                                                //SPR4096 存满
                {
                    DVR_Stop();
                    Send_QueueData_to_Sflash();
                    Modify_Index();
                    SubStatus = Stop;
                }
            break;
```

```
case Play:
    if(SACM_DVR_Status()&0x01)                    //检测是否正在放音
    {
        Fill_A_Word_To_Queue();
    SACM_DVR_Decode();
}
    else
    {
        DVR_Stop();
        SubStatus = Stop;
    }
break;

case RePlay_All:
    if(SACM_DVR_Status()&0x01)                    //检测是否正在放音
    {
        Fill_A_Word_To_Queue();
    SACM_DVR_Decode();
}
    else
    {
        DVR_Stop();
        if(Segment<Buffer[0])
        {
            Segment++;
            Addr=Buffer[Segment*2−1];
        }
        else
        {
            Segment=1;
            Addr=Buffer[1];
        }
        Play_Initial();
    }
break;
case RePlay:
    if(SACM_DVR_Status()&0x01)                    //检测是否正在放音
    {
```

```
            Fill_A_Word_To_Queue();
        SACM_DVR_Decode();
    }
        else
        {
            DVR_Stop();
            Addr=Buffer[Segment*2−1];
            Play_Initial();
        }
    break;
    case Download:                                  //下载
        Addr=Buffer[Buffer[0]*2+1];
        UART_Download();
        Modify_Index();
        BaseType = UartClose;
        SubStatus = Stop;
        Open_Interrupt();
    break;
    case Upload:                                    //装载
        Addr0=Buffer[Segment*2−1];
        Addr1=Buffer[Segment*2];
        for(Addr=Addr0;Addr<=Addr1;Addr++)
        {
            Ret=SP_SIOReadAByte(Addr);
            SP_UartSentByte(Ret);
        }
        BaseType = UartClose;
        SubStatus = Stop;
        Open_Interrupt();
    break;
    case Stop:
    default:
        i=50;
        while(i−−)
        {
            if(BaseType)
                SP_Export(P_IOA_Data,0x3f00);       //串行通信口开，L6L7 亮
            else
```

```
                    SP_Export(P_IOA_Data,0xfc00);          //串行通信口关，light L0L1 亮
                }
            break;
    }
*************************************************************************
```

我们可以看到程序中多次了调用 Modify_Index()，它的功能是什么呢？我们看看它的程序就知道了。以下这段程序包括了一个读、改、写的过程。先把索引区的信息读到 Buffer 数组中保存起来，然后修改，在对索引区进行页擦除之后又把修改后的索引信息写回去。由于 SPR4096 的每个扇区为 2 KB 即 2048 B，所以程序中计算下一段开始地址时，地址增量是 2048。擦除 SPR4096 的最小单位就是 2048 B。为了保证删除其中某段语音数据的时候，不会删除到相邻的其它段，这里把地址增量设为 2048。

```
*************************************************************************
    void Modify_Index(void)
    {
        for(i=0,internal_addr=0xf000;i<21;i++,internal_addr++)
            Buffer[i]=SP_ReadWord(internal_addr);
        Buffer[0]+=1;                                   //段数
        Ret=Buffer[0]*2;
        Buffer[Ret]=Addr;                               //现行段地址结束
        Tem=0;
        for(i=1;(i<256)&&((Tem<Addr)||(Tem==Addr));i++)
            Tem=2048*i;
        Buffer[Ret+1]=Tem;                              //下一段地址开始
        SP_PageErase(0xf000);
        for(i=0,internal_addr=0xf000;i<21;i++,internal_addr++)
            SP_WriteWord(internal_addr,Buffer[i]);      //存在内部 FLASH
    }
*************************************************************************
```

以上涉及的函数 SP_ReadWord()、SP_WriteWord()SP_PageErase()、SP_SIOMassErase()、SP_PageErase() 、 SP_SIOSendAWord() 、 SP_SIOReadAByte() 、 SP_UART_Init() 、 SP_UartSentByte()和 UART_Download()的程序在后面作了介绍。

2. 硬件系统初始化程序

硬件系统初始化包括中断设置和 I/O 设置。这里分别给出了纯汇编格式和调用库函数两种实现方式。

以下是调用库函数实现系统初始化程序。

```
*************************************************************************
    void    System_Initial()
    {
```

```
        SP_Init_IOA(0xFFF0, 0xFFF0, 0xFFF0);
        SP_Init_IOB(0xFF7F, 0xFFFF, 0x0003);
        Set_SIO_Ctrl(0x00C3)
        R_InterruptStatus = C_FIQ_TMA;
        Set_INT_Ctrl(C_FIQ_TMA);
    }
*************************************************************************
```

以下是为用汇编语言实现系统初始化程序。

```
*************************************************************************
    .PUBLIC _System_Initial
    _System_Initial: .PROC
        CALL      F_User_Init_IO                    //I/O 初始化
        R1=C_FIQ_TMA                                //设置 FIQ_TMA 中断
        [_R_InterruptStatus] = R1
        [P_INT_Ctrl] = R1
        RETF
        .ENDP
————————————————————————————————————————
    F_User_Init_IO:
        R1 = 0xFFF0                                 //高 8 位接发光管，低 8 位接键盘
        [P_IOA_Attrib] = R1
        [P_IOA_Dir] = R1
        [P_IOA_Data] = R1
        R1 = 0xFF7F                                 //IOB7 为浮动输入，IOB10 为带缓冲输出
        [P_IOB_Dir] = R1
        R1 = 0xFFFF
        [P_IOB_Attrib] = R1
        R1 = 0x0003
        [P_IOB_Data] = R1
        R1 = 0x00C3                                 //设置 SIO 状态
        [P_SIO_Ctrl] = R1
        RETF
*************************************************************************
```

3. 内部 FLASH 的读写程序

32 K 字的内部 FLASH 被划分为 128 个 PAGE(每个 PAGE 存储容量为 256 字)，第一页为[0x8000～0x80FF]，最后一页为[0xFF00～0xFFFF]。全部 32 K 字闪存均可被编程写入或被擦除。

上电后，芯片处于读存储单元状态，读存储单元的操作与 SRAM 相同。函数 SP_ReadWord()实现的功能是从内部 FLASH 中读出一个字，参数为 FLASH 地址，返回值为该地址存放的数据。以下分别给出了用 C 语言实现和用汇编语言实现的程序。

SP_ReadWord ()用 C 语言实现的程序：

```
*************************************************************************
    unsigned int      SP_ReadWord (unsigned int addr)
    {
        unsigned int *ADDR,data;
        ADDR=(unsigned int*)addr;
        data = *ADDR;
        return data;
    }
*************************************************************************
```

SP_ReadWord()用汇编语言实现的程序：

```
*************************************************************************
    //-- Procedure:    _SP_ReadWord
    //-- Syntax:       SP_ReadWord(page addr)
    //-- Parameter:          R1 = page addr
    //-- Return:             R1 = data
————————————————————————————————————————————————————————
    .PUBLIC _SP_ReadWord
    _SP_ReadWord: .PROC
        PUSH BP TO [SP]
        BP = SP + 1
        R1 = [BP+3]
        CALL F_ReadWord
        POP BP FROM [SP]
        RETF
        .ENDP
————————————————————————————————————————————————————————
    .PUBLIC F_ReadWord
    F_ReadWord: .PROC
        PUSH R2,R2 TO [SP]
        R2 = [R1]
        R1 = R2
        POP R2,R2 from [SP]
        RETF
        .ENDP
*************************************************************************
```

和读 FLASH 相比，擦写就麻烦了一点。擦写的过程如图 2.9 所示。需要指出一点，图中所提到延时等待是由硬件完成的不需要软件延时。

SP_PageErase()为页擦除函数。

用 C 语言实现 SP_PageErase()的程序：

```
*************************************************************************
    void    SP_PageErase(unsigned int addr)
    {
    unsigned int *ADDR
    ADDR=(unsigned int*)addr
    *P_Flash_Ctrl = 0xAAAA
    *P_Flash_Ctrl = 0x5511
    *ADDR=1
    }

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

用汇编语言实现 SP_PageErase()的程序：

```
*************************************************************************
    //-- Procedure:    _SP_PageErase
    //-- Decription:   Erase a page in flash
    //-- Syntax:             SP_PageErase(page addr)
    //-- Parameter:    R1 = page addr
    //-- Return:       None
————————————————————————————————————————————————————————
    .PUBLIC F_PageErase

    F_PAGEERASE: .PROC

      PUSH R2 TO [SP]
      R2 = R1
      R1 = 0xAAAA                          //允许写入/擦除
      [P_Flash_Ctrl] = R1
      R1 = 0x5511                          //擦除命令
      [P_Flash_Ctrl] = R1
      [R2] = R1                            //擦除页写任意值
      POP R2 FROM[SP]
      RETF
      .ENDP
*************************************************************************
```

SP_WriteWord()的功能是往 FLASH 中写一个字的数据，该函数有两个参数，第一个是 FLASH 地址，第二个是要写入的数据。

用 C 语言实现 SP_WriteWord()的程序：

```
*************************************************************************
    void    SP_WriteWord (unsigned int addr,unsigned int data)
    {
    unsigned int *ADDR;
    ADDR=(unsigned int*)addr;
    *P_Flash_Ctrl = 0xAAAA;
    *P_Flash_Ctrl = 0x5533;
    *ADDR=data;
    }
*************************************************************************
```

用汇编语言实现 SP_WriteWord()的程序：

```
*************************************************************************
    //-- Procedure:    _SP_WriteWord
    //-- Syntax:       SP_WriteWord(page addr,data)
    //-- Parameter:    R1 = page addr，      R2 = data
    //-- Return:       None.PUBLIC _SP_WriteWord
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
    .PUBLIC _SP_WriteWord
    _SP_WriteWord: .PROC
        PUSH BP,BP TO [SP]
        BP = SP + 1
        R1 = [BP+3]
        R2 = [BP+4]
        CALL F_WriteWord
        POP BP,BP FROM [SP]
        RETF
         .ENDP
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
    .PUBLIC F_WriteWord
    F_WriteWord: .PROC
         PUSH R3,R3 TO [SP]
         R3 = R1
         R1 = C_EnableFlashAccess                    //允许写入/擦除
         [P_Flash_Ctrl] = R1
         R1 = C_ProgramFlash                         //编程指令
         [P_Flash_Ctrl] = R1
         [R3] = R2                                   //写 Flash
         POP R3,R3 FROM [SP]
```

```
        RETF
        .ENDP
*************************************************************************
```

4. **串行 FLASH 的读写程序**

从原理上来说，串行 FLASH **读写和** SPCE061A **内部的** FLASH **读写没有什么区别。下面给出相关函数，用到的函数均给出了用** C **语言实现和用汇编语言实现的程序。**

SP_SIOSendAByte()**实现的功能是送一个字节的数据到串行** FLASH**。**

用 C **语言实现** SP_SIOSendAByte()**的程序：**

```
*************************************************************************
    void    SP_SIOSendAByte (unsigned long int addr,unsigned int data)
    {
        int a
        *P_SIO_Addr_Low=addr                    //输入 SFLASH 低字节地址
        addr>>=8
        *P_SIO_Addr_Mid=addr                    //输入 SFLASH 中间字节
        addr>>=8                                //右移 8 位
        addr&=0x0007                            //输入 SFLASH 高字节地址
        *P_SIO_Addr_High=addr
        *P_SIO_Ctrl=0x00C3+C_SIOCLOCK
                        //SIO 初始化时钟频率为 CPUclk/8(24 位地址，写)
       *P_SIO_Start=1                           //允许写模式
        a=0x2FF                                 //延时
        while(a--)
        *P_SIO_Data=data                        //传数状态
        Wait_FreeFlag()                         //等待 SIO 空闲
        *P_SIO_Stop=1                           //禁写
    }
*************************************************************************
```

这里调用了一个函数 Wait_FreeFlag()**，其功能是查询** P_SIO_Start **单元的第七位是否为“**1**”，等待** SIO **到** non_busy **状态。以下是该函数的程序：**

```
*************************************************************************
    void Wait_FreeFlag(void)
    {
        unsigned int a;
        a=*P_SIO_Start;
        a&=0x0080;
        while(a)
        {
```

```
            a=*P_SIO_Start;
            a&=0x0080;
        }
    }
```

****

用汇编语言实现 SP_SIOSendAByte()的程序：

****

```
    // Function:        Send A Byte to Serial Flash
    // Syntax:          SP_SIOSendAByte(AddressLow,AddressHigh, data)
    // Used register:   R1,R2,R3
————————————————————————————————————————————————
    .PUBLIC _SP_SIOSendAByte
    _SP_SIOSendAByte: .PROC
        PUSH BP,BP TO [SP]
        BP = SP + 1
        R1 = [BP+3]
        [P_SIO_Addr_Low]=R1              //输入 SFLASH 低字节地址
        R1=R1 LSR 4                      //右移 8 位
        R1=R1 LSR 4
        [P_SIO_Addr_Mid]=R1              //输入 SFLASH 中间地址
        R1 = [BP+4]                      //端口指令
        R1=R1&0x0007                     //输入 SFLASH 高字节地址
        [P_SIO_Addr_High]=R1
        R1=0x00C3+C_SIOCLOCK             //C_SIOCLOCK=0x0010
        [P_SIO_Ctrl]=R1
        [P_SIO_Start]=R1                 //允许写方式
        R1 = [BP+5]
        [P_SIO_Data]=R1                  //传输数据
    L_WaitSIOSendReady:
        R1=[P_SIO_Start]
        TEST    R1,0x0080
        JNZ     L_WaitSIOSendReady
        [P_SIO_Stop]=R1                  //禁止写
        POP BP,BP FROM [SP]
        RETF
        .ENDP
```

****

SP_SIOReadAByte ()实现的功能是从串行 FLASH 中读出一个字节的数据。

用 C 语言实现 SP_SIOReadAByte ()的程序：

```
*****************************************************************************
unsigned int SP_SIOReadAByte (unsigned long int addr)
{
    unsigned int a,data;
    *P_SIO_Addr_Low=addr;                    //输入 SFLASH 低字节地址
    addr>>=8;                                //右移 8 位
    *P_SIO_Addr_Mid=addr;                    //输入 SFLASH 中间地址
    addr>>=8;                                //右移 8 位
    addr&=0x0007;                            //输入 SFLASH 高字节地址
    *P_SIO_Addr_High=addr;
    *P_SIO_Ctrl=0x0083+C_SIOCLOCK;
    *P_SIO_Start=1;                          //允许读方式
    a=*P_SIO_Data;                           //清 SFLASH 缓存
    Wait_FreeFlag();                         //等待 SJO 空闲
    data=*P_SIO_Data;                            //读数
    Wait_FreeFlag();                         //等待 SJO 空闲
    *P_SIO_Stop=1;                           //禁止读
    return(data);
}
*****************************************************************************
```

用汇编语言实现 SP_SIOReadAByte ()的程序：

```
*****************************************************************************
// Function: Read A Byte to Serial Flash
// Syntax: SP_SIOReadAByte(AddressLow, AddressHigh)
// Used register: R1,R2,R3
// Return register: R1
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
.PUBLIC _SP_SIOReadAByte
SP_SIOReadAByte: .PROC
        PUSH BP,BP TO [SP]
        BP = SP + 1
        R1 = [BP+3]
        [P_SIO_Addr_Low]=R1              //输入 SFLASH 低字节地址
        R1=R1 LSR 4
        R1=R1 LSR 4
        [P_SIO_Addr_Mid]=R1              //输入 SFLASH 中间地址
        R1 = [BP+4]                      //端口方向设置
```

```
        R1=R1&0x0007                    //输入 SFLASH 高字节地址
        [P_SIO_Addr_High]=R1
        R1=0x0083+C_SIOCLOCK
        [P_SIO_Ctrl]=R1
        [P_SIO_Start]=R1                //允许读
        R2=[P_SIO_Data]                 //清 SFLASH 缓存
L_WaitSIOReadReady1:
        R1=[P_SIO_Start]
        TEST    R1,0x0080
        JNZ     L_WaitSIOReadReady1
        R1=[P_SIO_Data]                 //读数据
L_WaitSIOReadReady2:                    //等待读结束
        R2=[P_SIO_Start]
        TEST    R2,0x0080
        JNZ     L_WaitSIOReadReady2
        [P_SIO_Stop]=R2                 //禁止读
        POP BP,BP FROM [SP]
        RETF
        .ENDP
*************************************************************************
```

SIOMassErase()实现的功能是对串行 FLASH 进行格式化操作。

用 C 语言实现 SP_SIOMassErase ()的程序：

```
*************************************************************************
void  SP_SIOMassErase (void)
{
long int a;
*P_SIO_Ctrl=0x00C0+C_SIOCLOCK;
*P_SIO_Addr_Low=0x0000;             //输入 SFLASH 低字节地址
*P_SIO_Addr_Mid=0x00C0;             //输入 SFLASH 中间地址
*P_SIO_Addr_High=0x00C0;            //输入 SFLASH 高字节地址
*P_SIO_Start=1;                     //允许写方式
*P_SIO_Data=0;                      //A7～A0 = 0, 传输状态
Wait_FreeFlag();                    //等待传数结束
    *P_SIO_Stop=1;                  //禁止读
    a=0x7FF;                        //延时
while(a--);
}
*************************************************************************
```

用汇编语言实现 SP_SIOMassErase ()**的程序：**

```
*****************************************************************************
    //Function : Mass Erase for S_Flash
    // Syntax: SIOMassErase()
    // Used register: R1,R2
————————————————————————————————————————————
    .PUBLIC _SP_SIOMassErase
    _SP_SIOMassErase: .PROC
            PUSH R1,R2 to [SP]
            R1=0x00C0+C_SIOCLOCK
            [P_SIO_Ctrl]=R1                 //SIO 初始化
            R2=0x0000
            [P_SIO_Addr_Low]=R2             //输入 SFLASH 低字节地址
            R2=0x00C0
            [P_SIO_Addr_Mid]=R2             //输入 SFLASH 中间地址
            R2=0x00C0
            [P_SIO_Addr_High]=R2
            [P_SIO_Start]=R1                //允许写
            R1=0
            [P_SIO_Data]=R1                 //传数
    L_WaitSIOSendReadyMass:
            R1=[P_SIO_Start]
            TEST    R1,0x0080
            JNZ     L_WaitSIOSendReadyMass
            [P_SIO_Stop]=R1                 //禁止写
            POP R1,R2 from [SP]
            RETF
            .ENDP
*****************************************************************************
```

5. UART 通信程序

UART **通信程序包括** UART **初始化、发送数据、读取数据等。**

UART **初始化的函数：**

```
*****************************************************************************
    void   SP_UART_Init (void)
    {
        int a;
        *P_UART_Command1=0x20;              //串行通信口复位
        *P_UART_Command1=0x00;
```

```
        *P_UART_BaudScalarHigh=0x00;          //设置串行通信口波特率为 115 200
        *P_UART_BaudScalarLow=0x6B;
        *P_UART_Command1=0x000C;
        *P_UART_Command2=0x00C0;
        a=*P_UART_Data;                       //清接收缓存
    }
*****************************************************************************
```

发送一个字节的函数：

```
*****************************************************************************
    void   SP_UartSentByte (unsigned int data)
    {
        int a;
        a=*P_UART_Command2;
        a=a&0x0040;
        while(!a)                             //检测发送准备 TxRDY=1
        {                                     //等待发送
            a=*P_UART_Command2;
            a=a&0x0040;
        }
        *P_UART_Data=data;                    //发送数据
    }
*****************************************************************************
```

接收一个字节的函数：

```
*****************************************************************************
    unsigned int   SP_UartReadByte (void)
    {
        unsigned int data,a;
        a=*P_UART_Command2;
        a=a&0x0080;
        while(!a)                             //检测接收准备 RxRDY = 1
        {                                     //等待接收
            a=*P_UART_Command2;
            a=a&0x0080;
        }
        data=*P_UART_Data;                    //读数据
        return(data);
    }
*****************************************************************************
```

6. 键盘扫描程序

F_Key_Scan()为 4×4 键盘扫描函数，如图 8.5 所示。需要说明一点，没有键按下的时候返回值为 0xFF。为了节省时间，F_Key_Scan()是用汇编语言编写的。鉴于大家对键盘扫描都比较熟悉，这里就不浪费篇幅了。

本节介绍了一个应用 SPCE061A 作为控制核心实现简易录音笔的实例。我们介绍了模块的硬件设计方法以及针对各个硬件进行编程的过程，将各个硬件功能组合起来形成一个可用的简易录音笔的方法，略加修改即可实现其它功能。其设计方法同样适用于一般单片机应用系统的开发。当然，在此主要目的还是介绍系统的设计方法，读者可以举一反三，设计出更多的优秀作品。

第 9 章　SPCE061A 实用技术

本章将介绍 μ'nSP™系列单片机的主要应用领域，并以 SPCE061A 单片机为例介绍其在产品开发、数据通信、语音识别、语音播报等领域里的应用。为使读者能够尽快地掌握 SPCE061A 单片机的应用技术，开发新产品，本章将详细介绍几个应用实例的原理、硬件结构、程序流程及参考程序。

9.1　应 用 领 域

μ'nSP™系列单片机具有工作速率范围宽、集成度高、性价比高、功耗低、对电源要求低、可靠性高等特点，有非常广泛的应用领域。例如，语音播报与语音识别的微控制器(SPCE 系列)、通信来电识别应用的微控制器(SPT660X 系列)以及通用型微控制器(SPMC903 系列)等。

9.1.1　数字信号处理

1. 数字滤波器(Digital Filter)

数字滤波器是一种计算处理或算法，它将输入的数字信号或序列中的干扰滤掉变换为标准数字序列输出。数字滤波器已被广泛地应用于数字语音、数字图像处理以及模式识别和频谱分析等领域。

数字信号处理器 DSP(Digital Signal Processor)是通过一系列数字来表示信号及其信息，并借助数字计算方法，变换和处理这些信号的。在 DSP 系统中，要求必须快速地完成数值的乘法运算并将乘积累加于寄存器，因而要求处理器具有较高的处理速度，以完成“乘和累加”MAC(Multiply & Accumulate)运算，μ'nSP™ 的指令系统中专门设计了“累计运算”指令，使得数字信号处理变得非常方便。显然，μ'nSP™的硬件结构与其指令系统非常适用于 DSP 方面的开发与应用。

2. 数字信号的压缩编码与解码

通常用于存储语音、图像等多媒体信息的数字信息量非常大，无论存储还是传输都很不方便，为了节省存储空间、提高传输效率，我们可以采用某种算法进行压缩编码与解码(篇幅所限，不再赘述，请参阅数字通信原理)。

3. 数字语音信号处理

数字语音信号处理是建立在 DSP 硬件基础上的，通常 DSP 按运算的复杂程度分为定点运算和浮点运算两类。定点 DSP 采用整数运算，适用于容量大、简单、低功耗的应用场

合。浮点 DSP 采用实数运算，适宜于高性能、较复杂的应用场合。μ'nSP™可用于定点 DSP 运算且其成本较低，在语音处理方面具有独到之处。

9.1.2 便携式终端

随着无线电通信及微电子技术的迅速发展，便携式终端越来越朝着实用化、个性化、时尚化的方向发展，笔记本电脑、电子词典、个人数字助理 PDA(Personal Digital Assistant)等都是这种时尚潮流产物。若以 μ'nSP™家族产品为 CPU，添加必要的外围电路，配合相应的外设可方便地设计出实用性强、可靠性高的各类 PDA 产品。图 9.1 是基于 μ'nSP™系列单片机的 PDA 硬件框图，是 PDA 的一个基本设计方案。针对不同的要求可对其外围电路作适当删减、修改，设计出具有不同功能和不同品位的 PDA 产品。

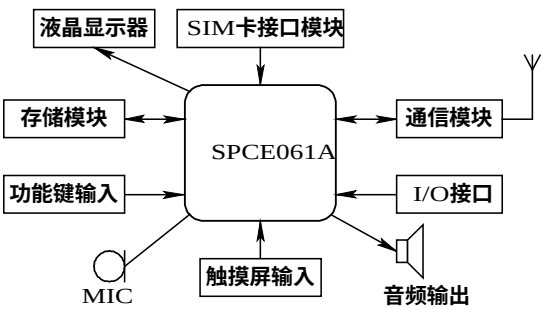


图 9.1 基于 μ'nSP™系列单片机的 PDA 硬件框图

便携式终端的基本功能如下：

- 内置微型实时操作系统 RTOS(Real-Time Operating System)。
- 语音识别输入或触摸屏输入。
- 无线移动式语音通信。
- 无线寻呼机、对讲机。
- 双机或多机通信。
- 下载升级软件。
- 移动定位导航及位置信息服务。
- 家电遥控器。
- 个人资料数据库(名片册、医疗档案及家庭理财等)。
- 电子词典、计算器、游戏机、学习机、万年历等。

9.1.3 嵌入式系统

嵌入式计算机系统 ECS(Embedded Computer System，简称嵌入式系统)是指专门用于某一应用系统或设备并隐藏于其中起控制作用的计算机系统。

与通用计算机系统相比，嵌入式系统有以下特征：专用性、可封装性、独立性、实时性及可靠性。所谓独立性，指 ECS 自成一个子系统，与目标系统的其它子系统保持相对独立。不同的应用领域对 ECS 的要求不同。

(1) 小型应用系统：一般不需大量的数据处理，要求 ECS 具有较强的实时控制功能、体积小、功耗低等。如智能化仪表、设备的自动控制、家用电器、遥控器等。通常选用 8 位机(如凌阳公司的 SPL 系列微处理器等)即可满足要求。

(2) 简单的工业控制系统：要求有相当强的实时数据处理能力和控制能力。如步进电机控制、数据采集、智能检测系统、汽车控制系统等。一般选用 8 位或 16 位单片机即可满

足要求，也可选用具有定点 DSP 运算功能的 μ'nSP™系列 16 位微处理器实现(如凌阳公司的 SPCE 系列微处理器等)。

(3) 工业控制系统：由于现场情况复杂、环境恶劣，通常采用分布式多机系统，采用模块化结构，在某些模块配置智能 I/O 处理机对现场信息进行实时测量和控制。因此，要求 ECS 不仅具有较强的数据处理能力，而且必须具有可靠性高、抗干扰能力强的硬件环境。如航空航天系统、机器人系统、生产线控制等。可根据需要选用若干个 8 位机、16 位机甚至 32 位机组合成分布式多机系统。

9.1.4　凌阳单片机应用

μ'nSP™的特点决定其能很好地胜任从简单到复杂的嵌入系统，在如下一些领域得到了广泛的应用：

(1) 工业控制。工业控制包括工厂生产线控制、锅炉控制、电力系统自动化、智能仪器仪表、机器人控制等。

(2) 汽车控制系统。汽车控制系统包括防撞系统、减震系统、静噪系统、燃油喷射系统、通信与音响、智能驾驶系统等。

(3) 消费娱乐。消费娱乐包括数字机顶盒、游戏机、智能玩具、语音复读机、电子词典、家用电器控制系统等。

(4) 通信领域。通信领域包括数字留言机、数字语音信箱、三遥系统、数字信号传输系统等。

(5) 军事领域。军事领域包括雷达与声纳信号处理、导航系统、保密通信、全球定位系统等。

9.2　应 用 举 例

单片机的应用非常广泛，为了帮助读者尽快掌握和使用 SPCE061A 单片机，下面以 SPCE061A 单片机在不同领域的应用为例，介绍其使用方法和应用技巧。

9.2.1　作息时间控制

1. 原理

在此设计了一个具有语音报时功能的作息时间控制钟。

假设某学校的作息时间如下：

08：00～08：50 第一节课	12：00～14：00 午间休息
08：50～09：00 休息	14：00～14：50 第五节课
09：00～09：50 第二节课	14：50～15：00 休息
09：50～10：10 课间操、休息	15：00～15：50 第六节课
10：10～11：00 第三节课	15：50～16：00 播放音乐
11：00～11：10 休息	16：00～16：50 第七节课
11：10～12：00 第四节课	16：50～18：00 体育活动

由于 SPCE061A 单片机具有语音播报功能和良好的时基系统，使用十分方便，所以此处选用 SPCE061A 单片机作为 CPU。利用 SPCE061A 单片机的 2 Hz 时基信号计时进行年历计算，利用语音播报功能实现报时，按照设定的时间进行相应的控制。

2. 硬件电路

硬件电路由 CPU、键盘、声音输出模块、显示模块和电源组成。SPCE061A 单片机的系统硬件框图如图 9.2 所示。

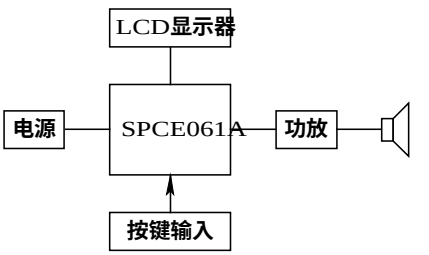


图 9.2 系统硬件框图

系统扩展了三个按键用于报时及校正时间。SPCE061A 的 DAC 为电流型输出，经负载电阻 R1、三极管 V1，放大驱动扬声器 SPEAKER 放音。SPEAKER 可选用 4 O 或 8 O 扬声器，根据具体需要播报相应的语音信息。IOA15 接一个 LED，显示作息时间到等信息，SPCE061A 的工作电压为 3.3 V，可通过两种方法获得，如图 9.3 所示。这两种方法如下：

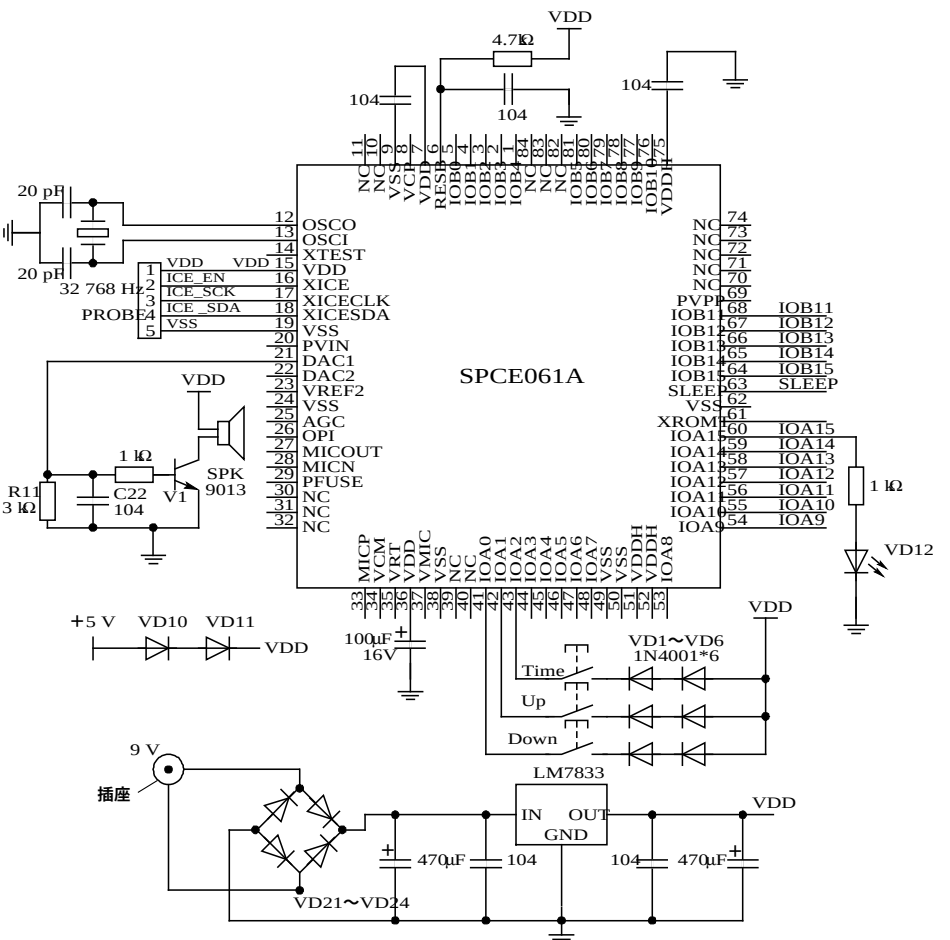


图 9.3 硬件连接图

(1) 通过两个二极管连续降压使 5 V 的电压降至 3.6 V 接近 3.3 V 供芯片使用，此法比较简单，但电压值不是很精确。

(2) 通过 LM7833 稳压块获得准确的 3.3 V 电压。

3. 程序说明

整个程序分为主程序、键盘扫描子程序、万年历计算子程序、校时子程序、语音播报子程序等几部分。

1) 主程序

程序按照模块化设计，所有功能都可通过调用子程序完成，主程序较简单，流程如图 9.4 所示。SPCE061A 具有低功耗睡眠模式，在睡眠模式下功耗很小，对于用电池供电的系统非常有利，睡眠模式可以通过按键中断唤醒。

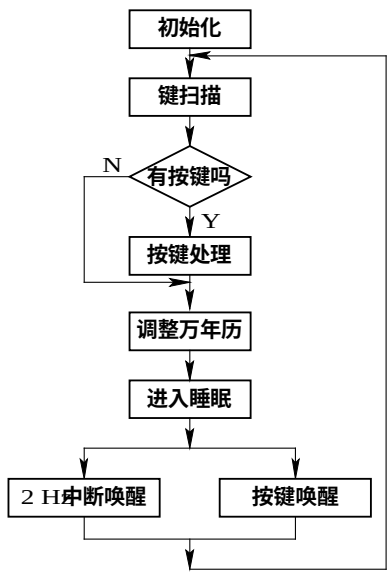


图 9.4　主程序流程图

2) 键盘扫描程序

由于机械触点有弹性，在按下或弹起按键时会出现抖动，从最初按下到接触稳定要经过数毫秒的弹跳时间，如图 9.5 所示。为了保证按键识别的准确性，必须消除抖动。消抖处理有硬件和软件两种方法：硬件消抖是利用加去抖动电路滤波避免产生抖动信号；软件消抖是利用数字滤波技术来消除抖动。我们采用软件方法，利用主程序循环扫描，主程序每循环一次扫描按键一次，当连续 N 次扫描到的键值相同时，则说明是某键按下。

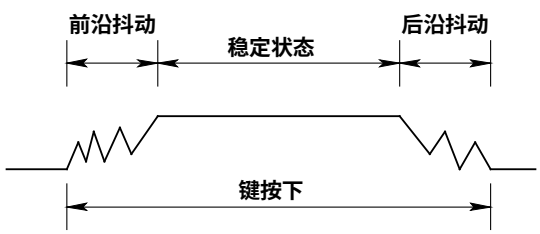


图 9.5　键按下的过程

3) 万年历计算子程序

利用 2 Hz 中断做时钟源进行计时，每两次中断秒计数器加 1 并进行年历计算，年历范围从 2001 年到 3099 年。在进行年历计算时有平闰年计算问题，闰年的条件是能被 400 整除或者能被 4 整除但不能被 100 整除。万年历调整子程序流程图如图 9.6 所示。

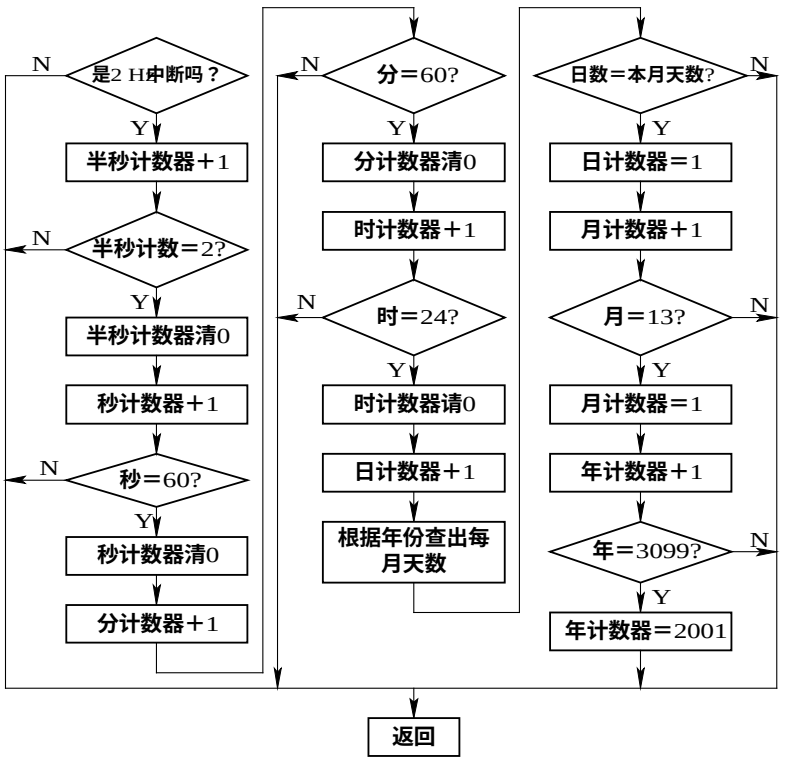


图 9.6　万年历计算子程序

4) 校时子程序

系统扩展了三个按键：TIME、UP 和 DOWN 键，常组合起来使用。TIME 键用于报时，由于时间包括年月日时分，按一次 TIME 键报年月日，再按一次报时分，一直按住 TIME 键 3 s 则进入校时状态，语音播报当前是 XXXX 年，按 UP 键年份增加，按 DOWN 键减少，再按 TIME 键来切换月日时分，调整完分后，按 TIME 键确认，语音报出年月日时分。校时程序的流程如图 9.7 所示(时间减少的流程与增加的相似，在此不再赘述)。在校正时间的

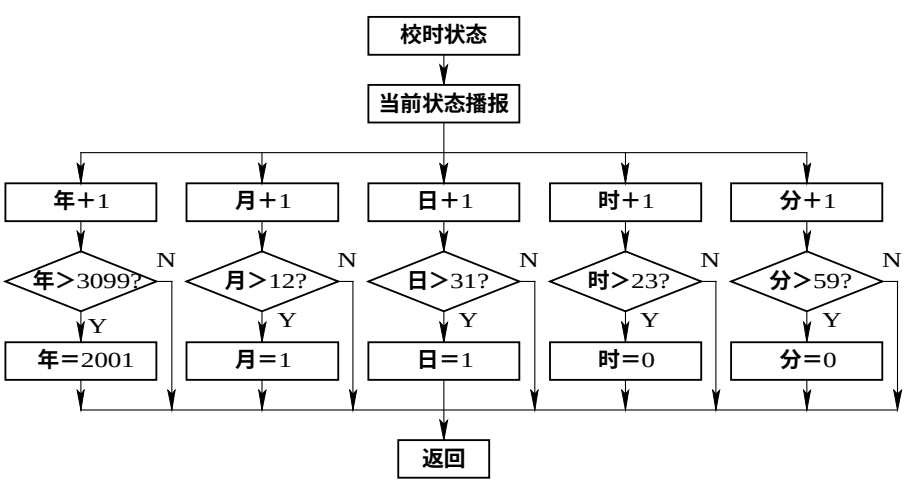


图 9.7　校时子程序

状态下，连续 2 分钟没有键按下则自动退出。

5) 语音播报子程序

报时用 SACM−A2000 播放。

4. 功能扩展

增加一个调频接收模块接受某电台的整点报时信号，利用凌阳 SPCE061A 单片机语音识别功能实现自动校时功能。

9.2.2　温度计设计

在各种控制系统中，常常要用到温度检测和控制。这里我们介绍一种采用热敏电阻作为温度传感器的测温系统，并以语音报告检测的温度，具有使用方便、扩充性强等优点，尤其适用于工业控制和温度监测领域。

1. 测温原理

采用热敏电阻作为温度传感器。热敏电阻是一种新型半导体感温元件，具有灵敏度高、体积小、寿命长、价格低等优点。热敏电阻分为正温度系数和负温度系数两种类型。负温度系数热敏电阻是当温度升高时电阻值减小，当温度降低时电阻值增大，其阻值温度特性曲线如图 9.8 所示。(热敏电阻的阻值温度特性曲线是一条指数曲线，非线性较大，实际应用中要进行线性化处理比较复杂，一般只使用线性度较好的一段，如图 9.8 所示的 AB 段。)通过测量热敏电阻的阻值就可以计算出其对应的温度值。

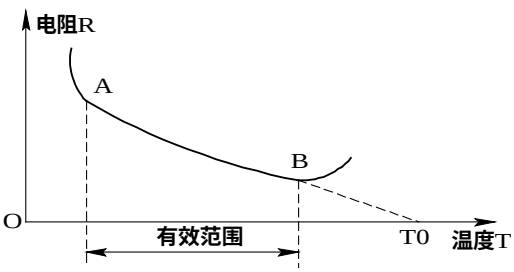


图 9.8　温度特性曲线

2. 硬件电路

用热敏电阻测温的硬件电路如图 9.9 所示。将热敏电阻 RT 与固定电阻 R 串联接 3.3 V 电源，当温度改变时，RT 阻值改变，其两端的电压也随之改变，通过测量两端的电压求得温度值：

$$T=T0-K\cdot UT$$

其中，T 为被测温度；T0 为与热敏电阻特性有关的温度参数；K 为与热敏电阻特性有关的系数；UT 为热敏电阻两端的电压。

固定电阻 R 阻值的选取：MFD-502-34 型热敏电阻线性化较好的一段是−20～80℃，为了在最高温度和最低温度时使被测信号基本接近满量程值，采取线性区域内中间某一点温度的阻值作为固定电阻的值，它们分压后 AD 的输入电压是 AD 输入电压范围的一半，在 25℃时热敏电阻的阻值为 5 kO，所以选取固定电阻 R 的值为 5 kO。

在−20℃时热敏电阻的阻值为 37.399 kO，两端电压 URT=2.9 V，接近 A/D 输入电压的上限 3.3 V；在 80℃时热敏电阻的阻值为 0.796 kO，两端电压 URT=0.45 V，接近 A/D 输入电压的下限 0 V。

按照图 9.9 硬件图，T0=76℃，K=0.1022，根据以上公式和参数，测出热敏电阻两端的电压就可以求出被测温度。

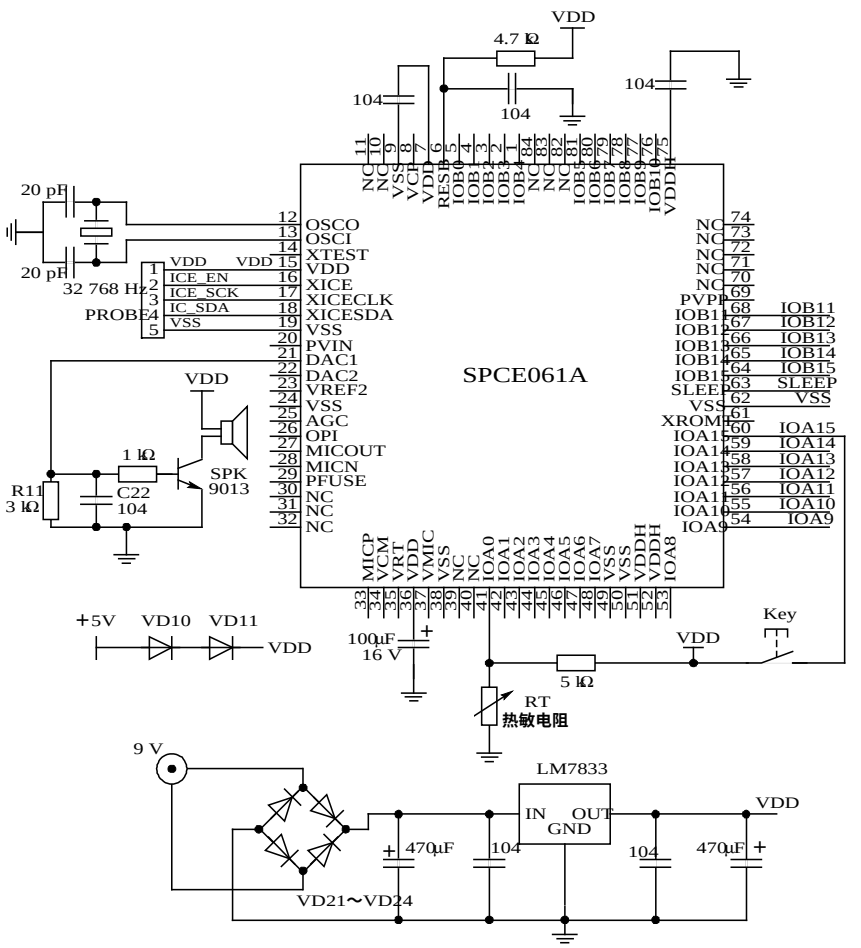


图 9.9 硬件连接图

3. 温度计算

系统扩展了一个按键接于 IOA15，当按键按下时，进行 A/D 转换初始化并进行 4 次 A/D 转换，SPCE061A 的 A/D 转换结果在高 10 位，每次将其移入低 10 位再计算 4 次平均值作为 A/D 有效结果返回。为了提高准确度，变量 TempAD 和 Temper 都采用浮点数，计算完成后用语音将温度值播报。由于在放音时播放函数会改变一些参数，为了稳定起见，在每次 A/D 转换前都做一次初始化。由于每个热敏电阻的特性不一致，与热敏电阻串联的电阻 R 不准确等原因，因此每台温度计在整个测量范围内应至少找 5 点进行校正，并适当地修改参数，以达到最佳状态。

A/D 转换程序如下：

```
.PUBLIC _ReadAD
_ReadAD: .proc
                r2=4；                          //共进行 4 次转换
                R3=0；
TestLoop:  R1=[P_ADC_MUX_Data]                  //进行一次 A/D 转换
                R1=0x8000;
TempConverLoop:
                TEST R1,[P_ADC_MUX_Ctrl]
                JZ TempConverLoop               //转换完成否
                R1=[P_ADC_MUX_Data]             //取 A/D 转换值
                R1=R1 LSR 4                     //A/D 值移到低 10 位
                R1=R1 LSR 2
                R3+=R1                          //4 次 A/D 值累加
                r2−=1
                JNZ TestLoop
                R3=R3 LSR 2                     //结果除以 4，求 4 次 A/D 值的平均值
                R1=R3                           //A/D 返回值在 R1 中
            RETF
.ENDP
```

温度计算及语音播报部分程序:

```
    if(Key==0x8000)
{
     InitAD();
     TempAD=ReadAD();                           //进行一次 A/D 转换
       Temper=76−0.1022*TempAD;                 //温度计算
       if(Temper<−20 | Temper>80)
       Temper=0XFFFF;                           //温度超出范围
       PlayVoice(Temper);                       //播报语音
}
```

9.2.3　红外遥控器

红外线遥控是目前最常用的遥控手段之一。红外线遥控装置具有体积小、功耗低、功能强、成本低等特点，因而继彩电、录像机之后，在 VCD 机、录音机、音响设备、空调机等家用电器中也纷纷采用。

1. 工作原理

1) 遥控指令编码

遥控器发送的功能指令码一般采用多位二进制串行码。下面以 29T6B−X 型彩色电视接

收机的红外遥控器为例说明其原理。

一帧数据结构为：同步脉冲、系统码、命令码、命令反码和结束位。同步脉冲作为一帧命令的起始位起到帧同步作用，系统码用于区别不同类别的电器，命令码用于完成命令功能。29T6B-X 型彩色电视的系统码为 0x08，命令码如表 9.1 所示。命令反码是将命令码按位取反，每次发送都是先发送脉宽为 4510 μs、周期为 2×4510 μs 的同步脉冲，然后连续发送两次系统码，接着发送命令码及命令反码，最后发送结束位，波形如图 9.10 所示。

表 9.1 遥控器命令码表

遥控功能	资料码	遥控功能	资料码	遥控功能	资料码
1	00	0	09	VOL+	13
2	01	-/--/---	0A	MUTE	14
3	02	POWER	OB	SLEEP	15
4	03	SYSTEM	OC	DISPLAY	16
5	04	AV	0F	SMPX	17
6	05	CH-	10	MENU	1C
7	06	CH+	11	SCAN	1E
8	07	VOL-	12	VOL M	2A
9	08	-	-	-	-

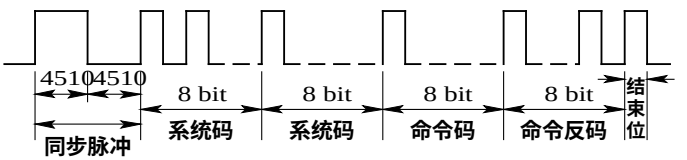


图 9.10 遥控指令编码图

2) 数据的脉冲编码

红外通信数据采用脉冲编码，即将每位数据信号用一个脉冲来表示。本例程序的红外编码采用频率调制(FSK)方式，以脉宽 561 μs 为周期，4×561 μs 代表 1，2×561 μs 代表 0。载波频率为 38 kHz，占空比为 1/3，如图 9.11 所示。这样做有两点好处：第一减少了有效发射时间，降低了平均功耗，对于采用干电池供电的发射器十分重要；第二提高了抗干扰能力。

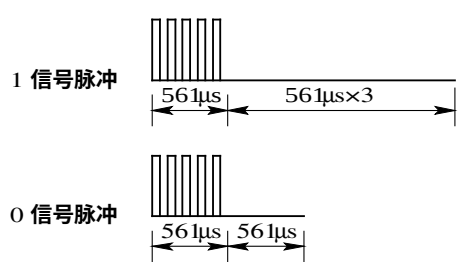


图 9.11 信号 0 和 1 的脉冲

2. 硬件电路

系统由键盘电路、红外发射电路和 CPU 组成。利用 SPCE061A 的 IOA 口扩展了 4×8 键盘矩阵，发射电路中三极管 V1 选用 9013，对信号放大，R4 为 270 Ω 电阻，R5 为 10 Ω 电阻，C5 为 22 μF，VD1 为红外发射管，

电路如图 9.12 所示。各个按键功能如表 9.2 所示。

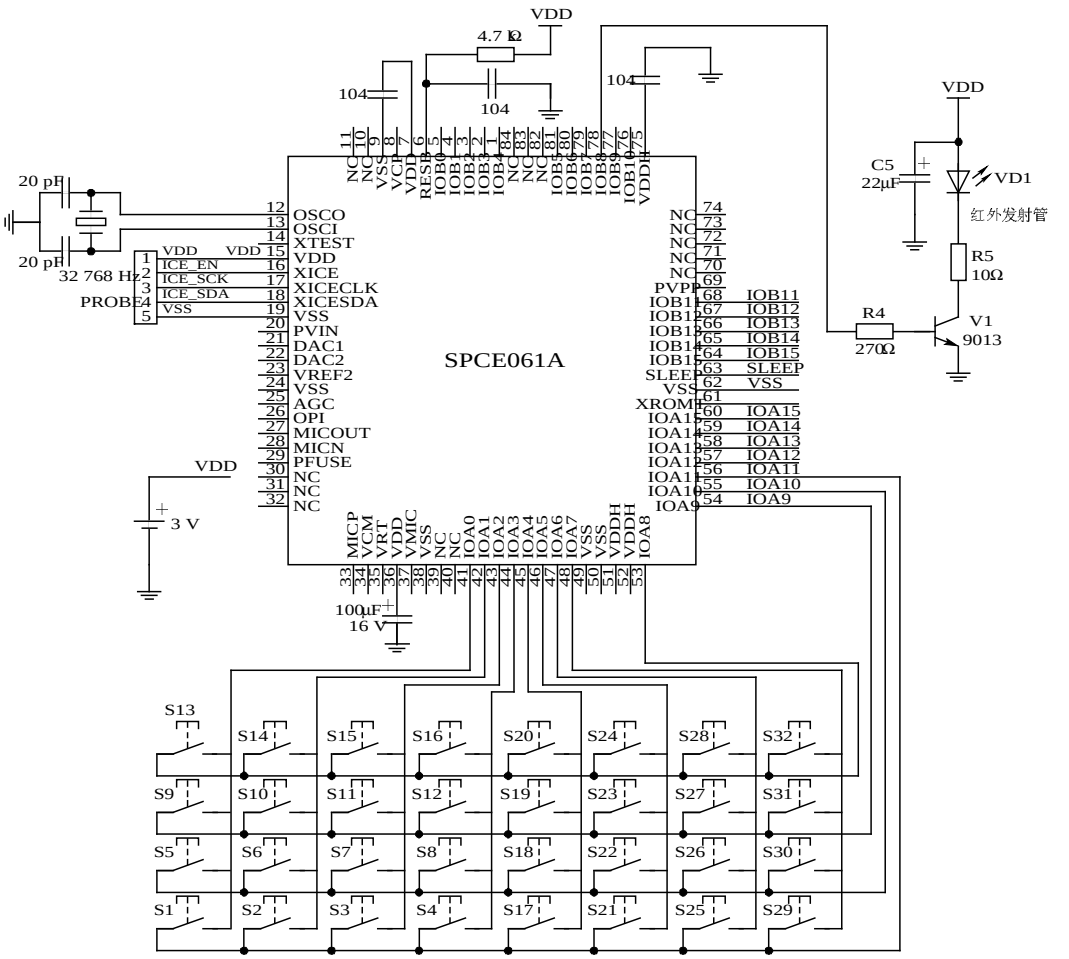


图 9.12　硬件连接图

表 9.2　按 键 功 能 表

按　键	遥控功能	按　键	遥控功能	按　键	遥控功能
S1	MUTE	S10	5	S18	0
S2	AV	S11	6	S19	SMPX
S3	SLEEP	S12	VOL M	S20	DISPLAY
S4	POWER	S13	7	S21	CH+
S5	1	S14	8	S22	MENU
S6	2	S15	9	S23	CH−
S7	3	S16	SYSTEM	S24	VOL−
S8	SCAN	S17	−/−−/−−−	S25	VOL+
S9	4				

3. 程序设计

SPCE061A 的 IOA0～IOA7 设置为在输入时具有按键唤醒功能。本例程序中将其设置为带下拉电阻的输入端口，IOA8～IOA11 设置为带数据缓存器的输出端口，键盘扫描程序比较简单，此处不作详细介绍。串行码的发送利用了 SPCE061A 的 TimerA 和 TimerB 两个定时器，IOB8 编程为第二功能时，由定时器 TimerA 控制输出占空比可调的脉宽调制信号 APWMO，产生 38 kHz 载波信号。将 APWMO 信号频率设置为 38 kHz，串行码为 1 时，打开 APWMO 输出 为 0 时，关闭 APWMO 输出，输出低电平。用 TimerB 控制脉冲宽度，发射流程如图 9.13(a)所示，同步脉冲、0 信号和 1 信号的发射流程如图 9.13(b)所示。

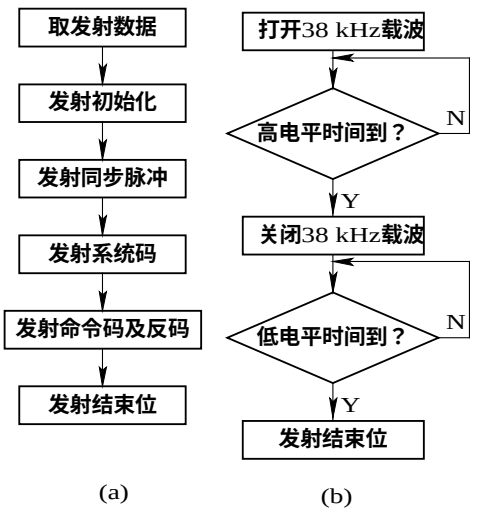


图 9.13 程序流程图

9.2.4 学习机

学习英语时，常常借助语音复读机来提高听力和检验阅读水平。这里以复读机为例介绍语音的录放技术。录音时通过 A/D 转换(包括抽样、量化和编码)将语音信号转换成数字信号，编码后存入存储器中；放音时将数据从存储器中取出并解码，然后经 D/A 转换成语音信号输出。

1. 原理

1) 录音

SPCE061A 的 A/D 转换器有 8 个通道，其中 1 个通道是 MIC-IN 输入，专门用于对语音信号进行采样。语音信号经 Mic 转换成电信号，经隔直电容输入至 SPCE061A 内部前置放大器。

SPCE061A 内部具有自动增益控制电路(AGC)，能随时跟踪前置放大器输出的音频信号电平。当输入信号增大时，AGC 电路自动减小放大器的增益；当输入信号减小时，AGC 电路自动增大放大器的增益，以便使输入 A/D 的信号保持在最佳电平。

A/D 转换器对输入的音频信号进行采样，根据抽样定理我们选用抽样频率为 8 kHz，并按照凌阳音频编码格式进行编码，每秒将占用 16 Kb 的存储器空间。为了增加存储时间，系统扩展了一块容量为 1 Mb 的 SRAM 存储器 HM628128A 用来存储语音数据。

2) 放音

将 HM628128A 中存储的语音数据顺序取出解码后以 8 kHz 的速率进行 D/A 转换输出，经电容滤波后恢复原始语音信号驱动扬声器发音。

2. 硬件电路

HM628128A 与 SPCE061A 的连接电路如图 9.14 所示，由 SPCE061A 的 I/O 口完成地址线、数据线和控制线扩展功能。系统扩展了三个按键 REC、PLAY 和 STOP。REC 用于录音，PLAY 用于放音，STOP 用于停止录音或放音。

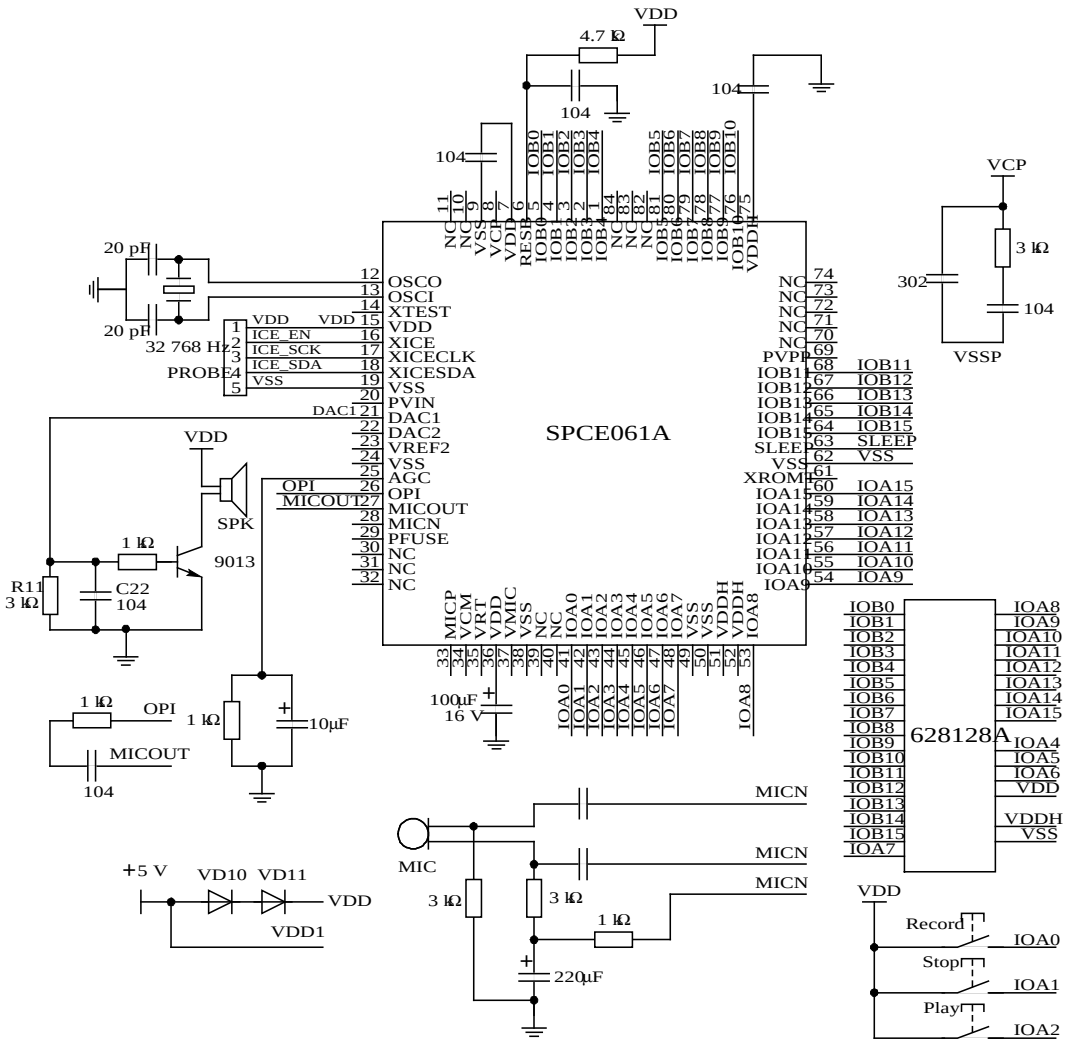


图 9.14　**硬件电路图**

3. 软件编程

录放音的编码/解码是靠调用库函数完成的，我们只要完成数据的存入和读出即可。程序主要是完成 HM628128A **的读写，对语音采样数据的编码处理是通过调用库函数完成的，写读程序都包括初始化和写读两部分，写入、读出程序较简单，只要按照** HM628128A **的写读时序即可。下面介绍初始化和写入时的参数传递过程，读出时参照写入部分。**

```
.PUBLIC _SP_InitWriteSRAM                    //写入初始化
.PUBLIC F_SP_InitWriteSRAM
_SP_InitWriteSRAM: .proc
F_SP_InitWriteSRAM:
        PUSH r1 TO [sp]；
        R1 = 0xFFF0；
```

```
                [P_IOA_Dir] = R1；                    //IOA15～A4 为输出
                [P_IOA_Attrib] = R1；                 //IOA3～A0 为输入
                [P_IOA_Data] = R1；
                R1 = 0xFFFFH;
                [P_IOB_Dir] = R1；                    //IOB15～B0 为输出
                [P_IOB_Attrib] = R1；
                [P_IOB_Data] = R1；
                POP r1 FROM [sp]；
            RETF
    .ENDP
```

初始化包括地址线、数据线和控制线的设置。IOB15～IOB0、IOA7 **接地址线，设置为输出；**IOA15～IOA8 **接数据线，写入时设置为输出；**IOA6～IOA4 **接选通信号，设置为输出。**

输入子程序:

```
.PUBLIC _SP_WriteSRAM
_SP_WriteSRAM: .PROC
                PUSH bp TO [sp]；
                bp = sp + 1；
                r1 = [bp+3]；                 //地址低位
                r2 = [bp+4]；                 //地址高位
                JZ Write_Low_Addr；           //判高位是否为 0
                R2 = 0x0080H;
 Write_Low_Addr:
                [HighAddr]=R2;
                R2 = [bp+5];                  //要输入的数据
                CALL F_SP_WriteSRAM           //输入子程序
                POP bp FROM [sp]
            RETF
    .ENDP
```

由于 HM628128A **有** 17 **根地址线，在主程序中记录地址的变量设置为长整型数据** 32 **位，在调用写入子程序时先将高** 16 **位压入堆栈再压入低** 16 **位，所以在写入子程序中取两次地址高低，并判断高位是否为** 0**。不为** 0 **则将** A16 **置** 1**，表明读/写的是高** 64 KB RAM**。读者只要对上面电路程序稍加改动，就可以实现语音复读机、跟读机和留言机等功能。**

9.2.5 多功能信号发生器

下面介绍几种基本波形的形成原理和实现方法，为满足不同的需要，所有波形的频率均为可调。

1. **原理**

三角波的上升沿和下降沿都是由 N **个** DAC **输出的小阶梯构成的，由于阶梯很小，从**

宏观上看它可以近似成三角波波形。如图 9.15 所示阶梯式的波形，阶梯之间的时间间隔 t1、t2 由定时器 TimerA 提供，通过改变 t1、t2 来改变三角波上升和下降时间 T1、T2，从而改变三角波的频率。

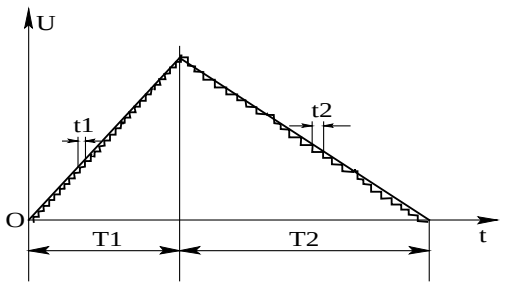


图 9.15　三角波原理图

正弦波利用 SPCE061A 的开发环境提供的数学运算函数库 math.lib，利用库中的 sinf(float<[x]>) 函数，当 x 在 0～2p 范围内变化时就产生一个完整的正弦波。以时间间隔 t 求的波形对应值 32 736×sinf(x) (注：32 736 为 SPCE061A 的 D/A 输出最大值 0xFFC0 的一半)经 SPCE061A 的 D/A 转换，在管脚 DAC1、DAC2 输出并经电容滤波后就可以得到需要的正弦波。通过改变时间间隔 t 可以改变正弦波的频率，正弦信号包括正负两个半周波形，如图 9.16 所示。由于 SPCE061A 无法输出负电平，要通过改变参考电平来获得负半周信号，所以将正弦信号电平正向偏移 1.65 V。

方波形成原理如图 9.17 所示，在 t1 期间 DAC 输出电压的峰值为 3.3 V，t2 期间 DAC 输出 0 V 电压，通过改变 t1、t2 的值来改变方波的频率占空比和频率。

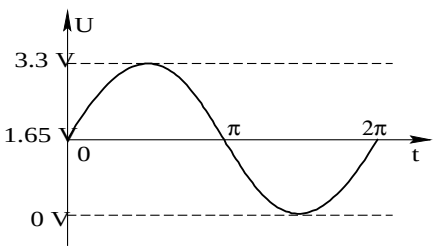


图 9.16　正弦波

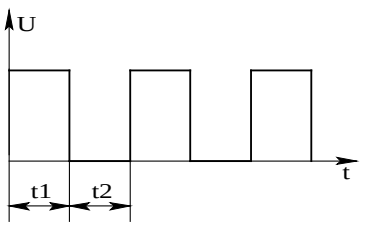


图 9.17　方波

2. 硬件电路

SPCE061A 内部具有 D/A 转换模块，所以其外围电路很简单，只要在 DAC 的输出端接一个运算放大器或负载电阻，将 DAC 输出的电流信号转换为电压信号即可，用按键方式选择输出波形，具体电路如图 9.18 所示。按 TRI 键输出三角波，按 SIN 键输出正弦波，按 SQU 键输出方波。

输出三角波时，UP 用于增加上升时间 t1 或下降时间 t2，DOWN 键用于减少 t1 或 t2，SET 用于选择上升或下降。

输出方波时，UP 用于增加高电平时间 t1 或低电平时间 t2，DOWN 键用于减少 t1 或 t2，SET 用于选择高电平或低电平。

输出正弦波时，UP 用于提高正弦波的频率，DOWN 键用于降低正弦波的频率。

在输出正弦波时，请接入滤波电容 C1，输出三角波和方波时不用 C1。

3. 程序说明

1) 三角波

SPCE061A 具有双通道 10 位电流输出型的 D/A(AUD1、AUD2)，每通道电流输出为 3 mA，在输出端接电阻 R(R 值取 600 O)，则可将输出的电流转换为电压信号。

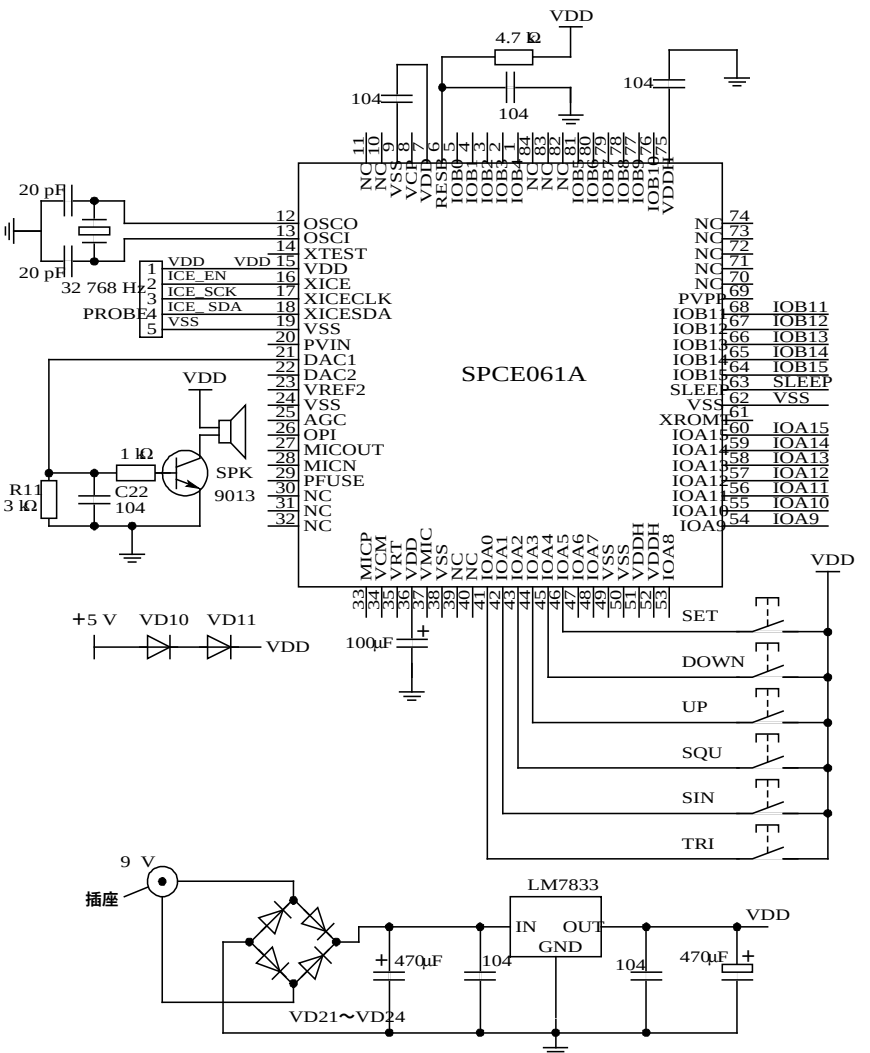


图 9.18　硬件连接图

三角波的上升和下降都由 100 **个小阶梯组成，主程序判断是上升还是下降。如果是上升，则将** TimerA **的定时值改为** t1**；如果是下降，则将** TimerA **的定时值改为** t2**，即三角波上升时间为** 100×t1**，下降时间为** 100×t2**，定时器** TimerA **的时钟源采用** 12.288 MHz**。**

定时器初值计算：

t1 **初值：**M1=65 535 − (12 288×T_High)/1000

t2 **初值：**M2=65 535 − (12 288×T_Low)/1000

由中断服务程序完成 DAC **数据输出。**

2) **方波**

方波的高电平 DAC **输出最大值** 0xFFC0H**，低电平时** DAC **输出最小值** 0**，方波高低电平的宽度由定时器** TimerB **定时值确定，方波的占空比为** D=t1/(t1+t2)**。**

3) **正弦波**

正弦波靠调用库函数 sinf(x)**产生，当弧度在** 0～2p **之间变化时，**sinf(x)**的值在**−1～1 **的**

范围内，将 sinf(x)的值乘以 32 736(0xFFC0H 的一半)即将波形放大，并将 0 点偏移到 1.65 V(即 3.3 V/2)的波形对应 AD 值计算部分程序如下：

```
mx= mz * PI/2;                    //计算弧度值
my = sinf(mx);                    //计算 sinf(mx)
vv= (int) 32 736 * my;            //AD 值
vv ^= 0x8000;                     //最高位取反
outdac(vv);                       //AD 输出
delay10ms(DelTime);               //延时 t
mz=mz+0.02;                       //增量
if(((unsigned int)mz)== 4) mz=0;  //循环
```

当弧度在 0～p 之间变化时，sin(x)×32 736 的值为正，最高位为 0，经 D/A 转换后其波形如图 9.19 中虚线 a 所示；当弧度在 p～2p 之间变化时，sin(x)×32 736 值为负(二进制补码表示最高位为 1)，经 D/A 转换之后其波形如图 9.19 中虚线 b 所示；将最高位取反后的 D/A 输出波形如图 9.19 中实线 A、B 所示。

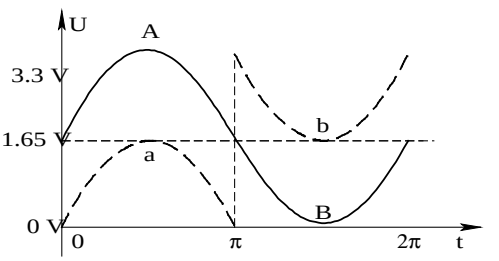


图 9.19　正弦波

9.2.6　语音识别

1. 原理

语音识别分为特定发音人识别(Speaker Dependent)和非特定发音人识别(Speaker Independent)两种方式。特定发音人识别是指语音样板由单个人训练，对训练人的语音命令识别准确率较高，而其他人的语音命令识别准确率较低或不识别；非特定发音人识别是指语音样板由不同年龄、不同性别、不同口音的人进行训练，可以识别一群人的命令。语音样板的提取非常重要，下面我们介绍采用特定发音人语音识别方式。

我们将标准模式的存储空间称之为“词库”，而把标准模式称之为“词条”或“样板”。所谓建立词库，是将待识别的命令进行频谱分析提取特征参数作为识别的“标准模式”。

识别过程(如图 6.3 所示)首先要滤除输入语音信号的噪音和进行预加重处理，提升高频分量，然后用线性预测系数等方法进行频谱分析，找出语音的特征参数作为未知模式，与预先存储的标准模式进行比较，当输入的未知模式与标准模式的特征一致时，计算机便识别出输入的语音信号并输出结果。

输入的语音与标准模式的特征完全一致固然好，但是，语音含有不确定因素，完全一致的情况几乎不存在，事实上没有人能以绝对相同的语调把一个词说两遍。因此，要预先制定好计算输入语音的特征模式与标准模式类似程度(或距离度)的算法规则，把距离最小(即最类似)的模式作为识别相应语音的方法。当然，影响识别率的因素还有很多，此处不再赘述。

本例程序采用特定发音人识别方式，将训练的标准样板存于内部 SRAM 中，每次上电复位后要进行训练，用户可以扩展一块 FLASH，将训练的标准样板存于 FLASH 中，而不用每次训练。

2. 硬件电路

硬件电路比较简单，MIC 选用驻极体电容话筒，这种话筒具有灵敏度高、无方向性、重量轻、体积小、频率响应宽、保真度好等优点，驻极体话筒所需的偏压由 SPCE061A 的 VMIC 管脚提供，如图 9.20 所示。

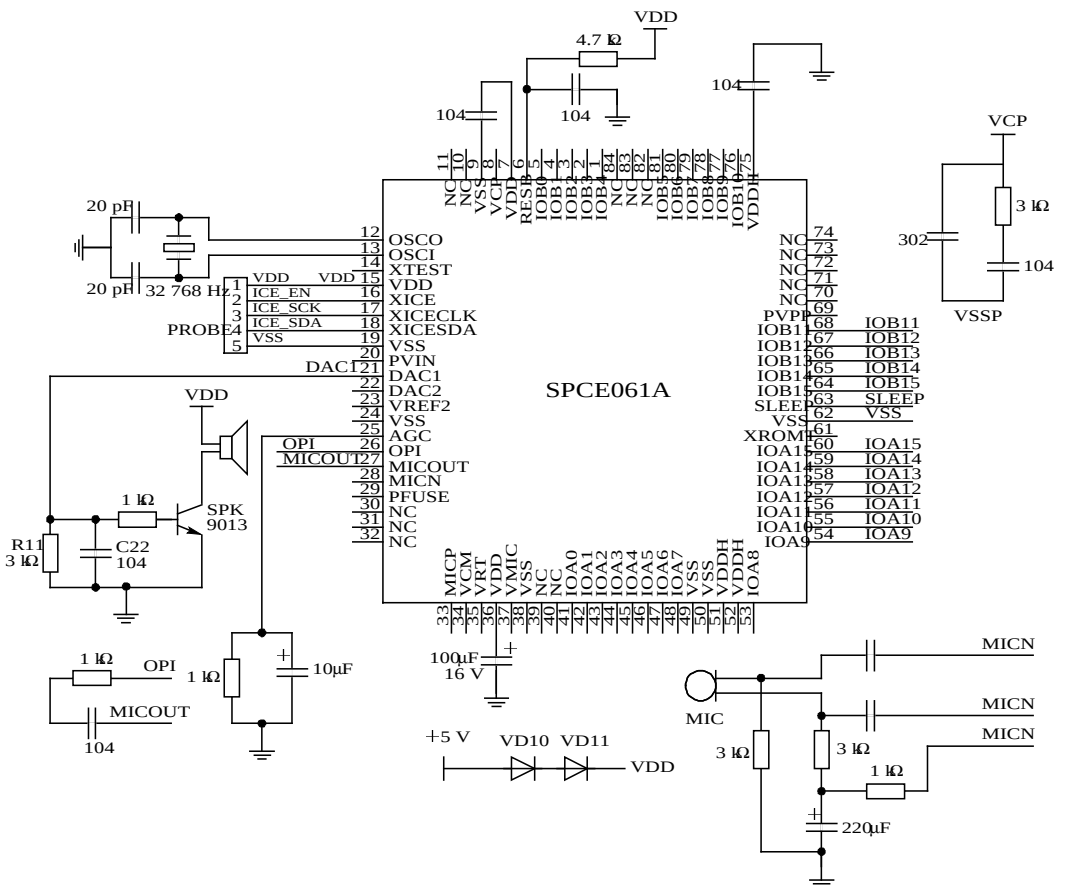


图 9.20 硬件电路图

3. 程序设计

程序包括训练样本、识别和语音提示三部分。

由于语音样本是存在内部 SRAM 中，掉电将丢失，所以每次上电复位后都必须重新训练，训练过程主要是靠调用库函数 BSR_Train 来完成。为了防止错误命令，每条语音命令训练两遍，只有两次命令相同时才成功。BSR_Train 函数有八种可能的返回值：

0——训练成功；

−1——没有检测到命令；

−2——需要再训练一次，每条命令训练两次，第一次训练成功则返回−2；

−3——环境太吵；

−4——存储器满；

−5——两次命令不一样；

-6——命令序号超出范围；

-7——命令已存在。

训练成功则继续训练下一条，否则重复训练。

语音识别程序包括识别程序和中断服务程序。识别程序完成选取词库、初始化 A/D、初始化定时器 TimerA、识别运算及识别结果处理等，流程如图 9.21 所示。

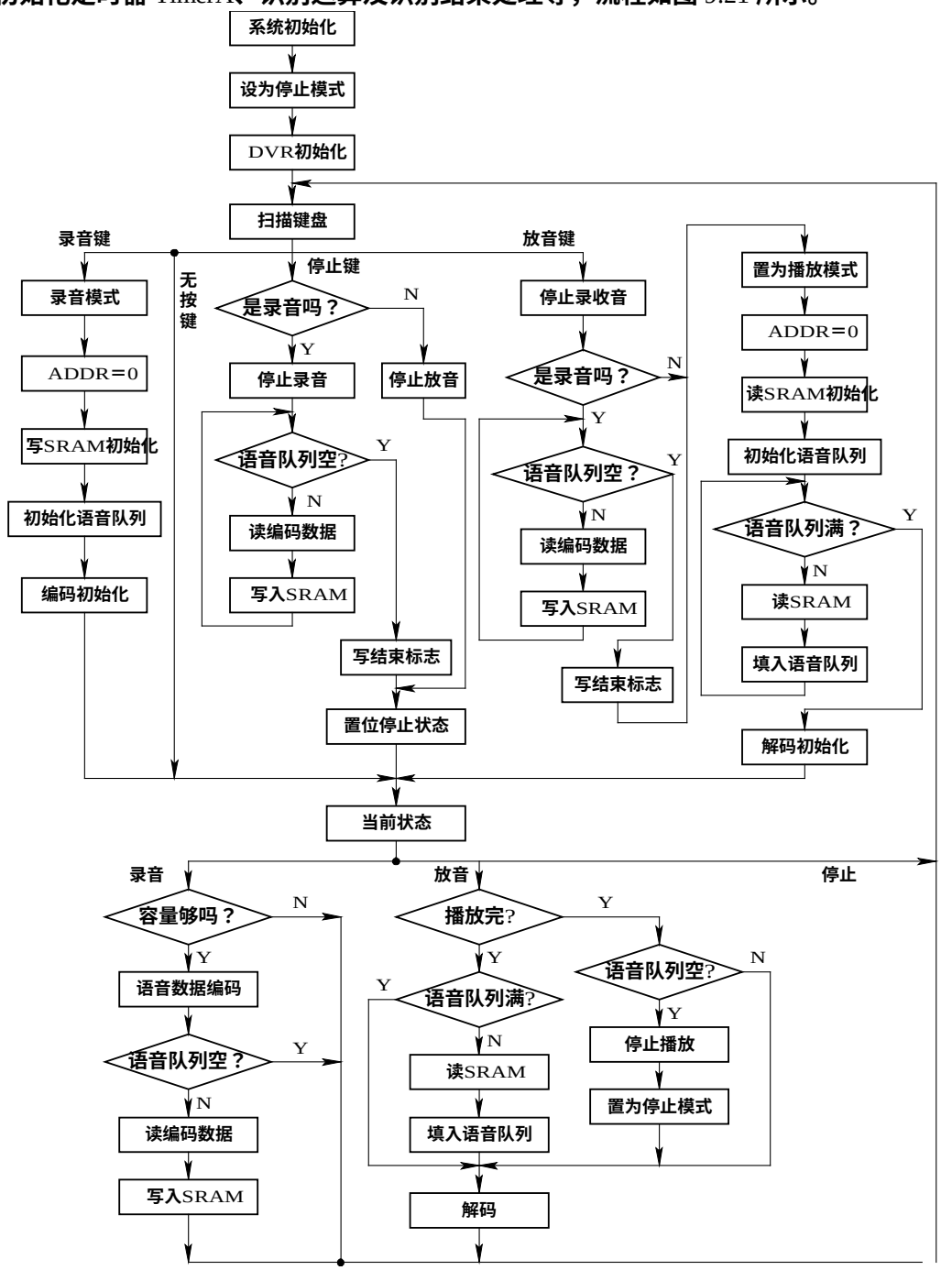


图 9.21　程序流程图

中断服务程序定时读取 A/D 转换结果并存入缓冲区，A/D 的输入为 MIC 通道的语音信号，语音识别和放音分时复用 TimerA FIQ 中断，通过标志位判断是语音识别还是放音处理。主程序流程如图 9.22 所示，中断服务程序的流程如图 9.23 所示。

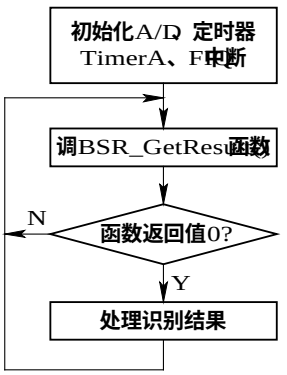


图 9.22 主程序流程图

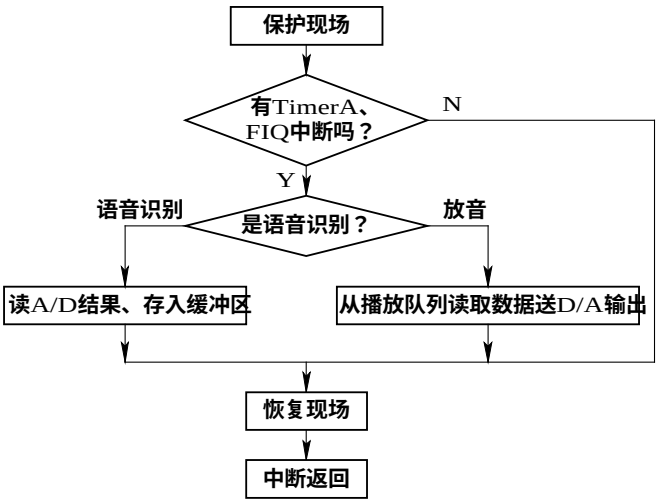


图 9.23 中断服务程序流程图

识别程序如下：

```
BSR_InitRecognizer(BSR_MIC);   //初始化识别器，包括 AGC、ADC、TimerA、FIQ
while(1)
{
   res = BSR_GetResult();      //主识别函数，识别成功则结果为语音命令的顺序号
   if(res > 0)                 //结果>0，表明识别成功


}
```

中断服务程序如下：

```
_FIQ:
            PUSH R1,R4 TO [SP]                  //寄存器入栈
            R1 = [P_INT_Ctrl]                   //读中断标志位
            R1 &= 0x2000;
            JZ FIQ_ret                          //不是 TimerA FIQ 中断
            R1 = [__gIsStopRecog]
            JNZ TimerA_is_S480
TimerA_is_Recognize:                            //TimerA FIQ 为语音识别服务
            CALL _BSR_FIQ_Routine               //语音识别服务函数
            JMP FIQ_ret
TimerA_is_S480:                                 //TimerA FIQ 为放音服务
            CALL F_FIQ_Service_SACM_S480;       //放音服务函数
```

```
FIQ_ret:    R1 = 0xa800;
            [P_INT_Clear] = R1;                 //清中断标志
            POP R1,R4 FROM [SP];                //寄存器出栈
        RETI
```

思　考　题

1. 以某中学为例，请设计一个作息时间控制器，上、下课按时打铃并以语音播报。
2. 以 SPCE061A 为 CPU 设计一个简易的语音控制密码锁。
3. 请设计时间为 10 s 的语音复读机。
4. 请设计一个具有语音播报功能的防盗报警器。

第 10 章　集成开发环境

μ'nSP™开发环境集程序的编辑、编译、链接、调试以及仿真等功能于一体，使编程、调试工作更加方便高效。此外，还具有软件仿真功能，可以在不连接仿真板的情况下模拟硬件的各项功能，来完成程序调试。IDE 的开发界面如图 10.1 所示。本章将介绍 μ'nSP™ 开发环境的菜单窗口、界面以及项目的操作等，使读者对开发环境有一个总体了解并能够正确使用开发工具。

10.1　调 试 菜 单

在桌面点击 IDE 图标，进入 IDE 开发环境。在标题栏下面，是集成开发环境的主菜单，菜单栏中的命令提供了开发调试和保存应用程序的工具。μ'nSP™ IDE 菜单栏共有七项：File(文件)、Edit(编辑)、View(视图)、Project(项目)、Build(编译)、Tools(工具)和 Help(帮助)，如图 10.1 所示。

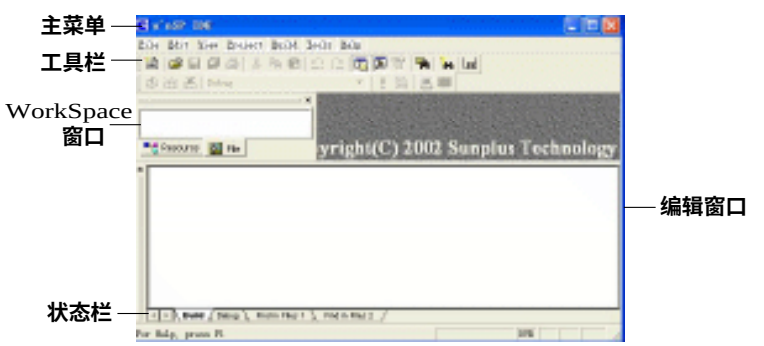


图 10.1　μ'nSP™ IDE 桌面

每个菜单项中含有若干个菜单命令，用鼠标单击某个菜单项即可打开该菜单，然后用鼠标单击菜单中的某一条就能执行相应的菜单命令。

菜单中的命令分为两种类型，一类是可以直接执行的命令，这类命令的后面没有任何信息；另一类在命令名后面带省略号，需要通过打开对话框来执行。用鼠标单击后一类命令，屏幕上将显示一个对话框，可以执行各种有关的操作。在有些命令的后面还带有其它信息，例如，打开项目 Ctrl + O(其中 Ctrl + O 叫做热键)，在菜单中，热键列在相应的菜单命令之后，与菜单命令具有相同的作用。使用热键方式，不必打开菜单即可执行相应的菜单命令。例如，按 Ctrl + O 键可以立即执行打开项目命令。

注意：只有部分菜单命令能通过热键执行。

下面介绍菜单栏各项的内容及作用。

1. File(**文件**)

File **下拉菜单界面如图** 10.2 **所示，下拉菜单的内容及功能如表** 10.1 **所示。**

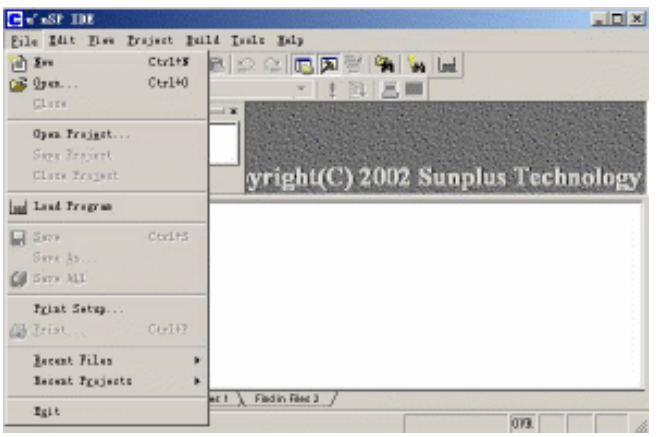


图 10.2　File **下拉菜单界面**

表 10.1　File **的下拉菜单内容及功能**

内容	作用	热键
New(**新建**)	**新建项目和各种文件**	Ctrl + N
Open(**打开**)	**打开项目或各种文件**	Ctrl + O
Close(**关闭**)	**关闭文件窗口**	
Open Project(**打开项目**)	**用来关闭当前的项目并装入新的项目，执行该命令后将打开一个对话框，可以在该对话框中输入要打开的项目名称**	
Save Project(**保存项目**)	**保存当前项目及其所有文件**	
Close Project(**关闭项目**)	**关闭当前项目**	
Load Program(**下载程序**)	**将程序下载到仿真板或本机内存中**	
Save(**保存**)	**保存当前的文件**	Ctrl + S
Save As(**另存**)	**用于改变存盘文件的名称。执行该命令后将弹出一个对话框，可以在这个对话框中输入存盘的文件名**	
Save All(**全部保存**)	**保存目前所有的文件和项目**	
Print Setup(**打印设置**)	**在执行该命令后，将显示标准的打印设置对话框，在该对话框中设置打印机页面方向、页面大小、纸张来源以及其它打印选项**	
Print(**打印**)	**把窗体及代码在由** Windows **设定的打印机打印出来**	Ctrl + P
Recent File(**近期文件**)	**打开最近使用的** 10 **个文件，主要是方便开发者在最短的时间内找到并打开所需的文件**	
Recent Project(**近期项目**)	**打开最近使用的** 10 **个项目，主要是方便开发者在最短的时间内找到并打开所需的项目**	
Exit(**退出**)	**退出开发环境**	

2. Edit(编辑)

Edit 下拉菜单界面如图 10.3 所示，下拉菜单的内容及功能如表 10.2 所示。

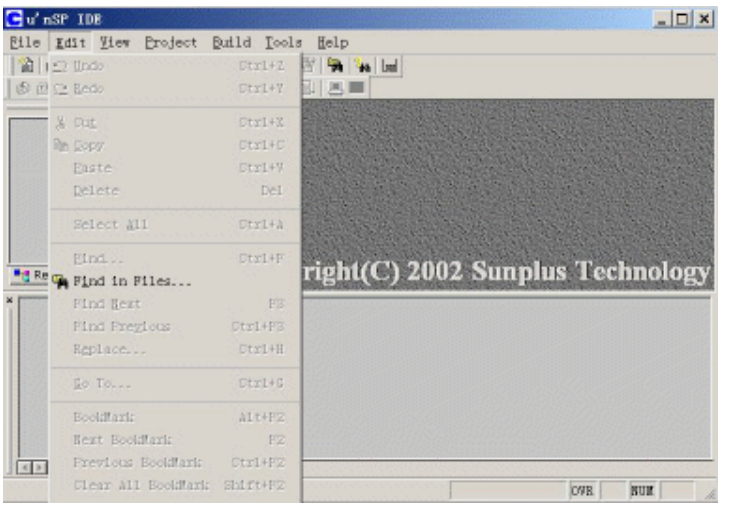


图 10.3 Edit 下拉菜单界面

表 10.2 Edit 的下拉菜单内容及功能

内容	作用	热键
Undo(撤消键入)	取消最近的多次编辑操作	Ctrl + Z
Redo(重复键入)	恢复撤消键入之前的编辑内容	Ctrl + U
Cut(剪切)	删除选中的文件内容或文件可以复制	Ctrl + X
Copy(复制)	拷贝选中的文件内容或文件	Ctrl + C
Paste(粘贴)	粘贴到指定的位置	Ctrl + V
Delete(删除)	删除选中的文件内容或文件	Del
Select All(全选)	选中所有的文件内容或文件	Ctrl + A
Find(查找)	查找文件内容或文件	Ctrl + F
Find in Files(在指定文件内查找)	在指定文件内查找文件内容或文件无快捷键	
Find Next(查找下一个)	用来查找并选择在“查找”对话框的“查找内容”框中指定的文本下一次出现位置	F3
Find Previous(查找前一个)	用来查找并选择在“查找”对话框的“查找内容”框中指定的文本上一次出现位置	Ctrl + F3
Replace(替换)	替换指定的文本，执行该命令后将显示一个对话框，在对话框的两个栏内分别输入要查找的文本和替换文本，即可一个一个地替换或一次全部替换	Ctrl + H
Go to(定位)	定位到某一行或列	Ctrl + G
Bookmark(标记)	在指定的位置设置标记	Alt + F2
NextBookMark(下一个标记)	光标指到下一个标记处	F2
Clear All BookMark (清除所有标记)	清除文件内所有标记	Shift + F2

3. View(视图)

View 下拉菜单界面如图 10.4 所示，下拉菜单的内容及功能如表 10.3 所示。

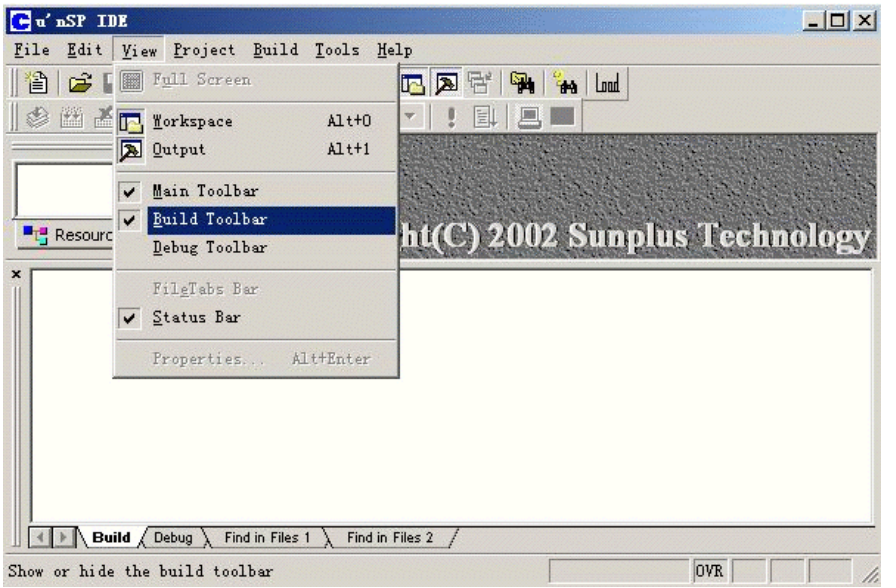


图 10.4　View 下拉菜单界面

表 10.3　View 的下拉菜单内容及功能

内容	作用	快捷键
Full Screen(全屏)	编辑窗口为全屏	
Workspace(工作区)	单击后弹出 Workspace 窗口	Alt + 0
Output(输出)	单击后弹出 Output 窗口	Alt + 1
Main Toolbar(常用工具栏)	包括新建、打开、保存、全存、打印、剪切、复制、粘贴、查找、撤消等工具	
Build Toolbar(编辑工具栏)	包括编译、编辑、停止、运行、下载、本机仿真、连接仿真板、调试	
Debug Toolbar(调试工具栏)	包括运行、下载、中断、停止调试、重新开始、单步执行等调试窗口	
FileTabs Bar(文件标签栏)	文件标签用于显示编辑窗口打开的文件名称	
Status Bar(状态栏)	提示光标所在的行列数	
Properties(属性)		

4. Project(项目)

Project 下拉菜单界面如图 10.5 所示，下拉菜单的内容及功能如表 10.4 所示。

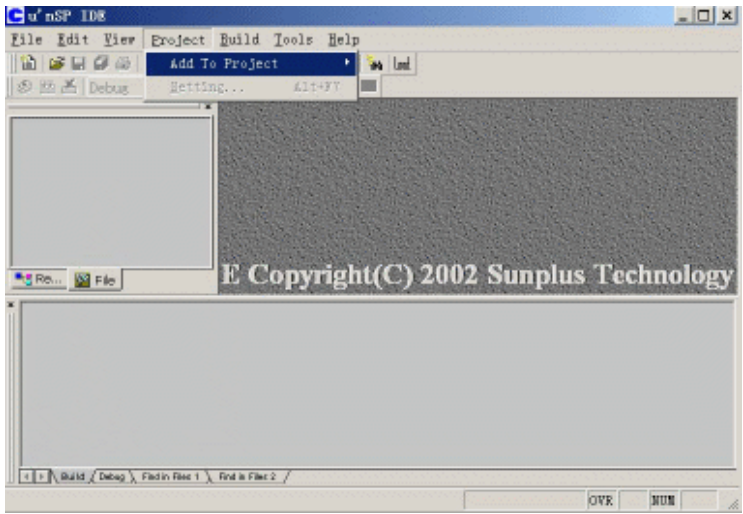


图 10.5 Project 下拉菜单界面

表 10.4 Project 的下拉菜单内容及功能

内容	作用	热键
Add to Project(加到项目)	包括向项目中加源文件和资源文件	
Setting(项目选项设置)	包括 General，Option，Link，Section，Hardware 和 Device。属性页设置后有描述	Alt + F7

5. Build(编译)

Build 下拉菜单界面如图 10.6 所示，下拉菜单的内容及功能如表 10.5 所示。

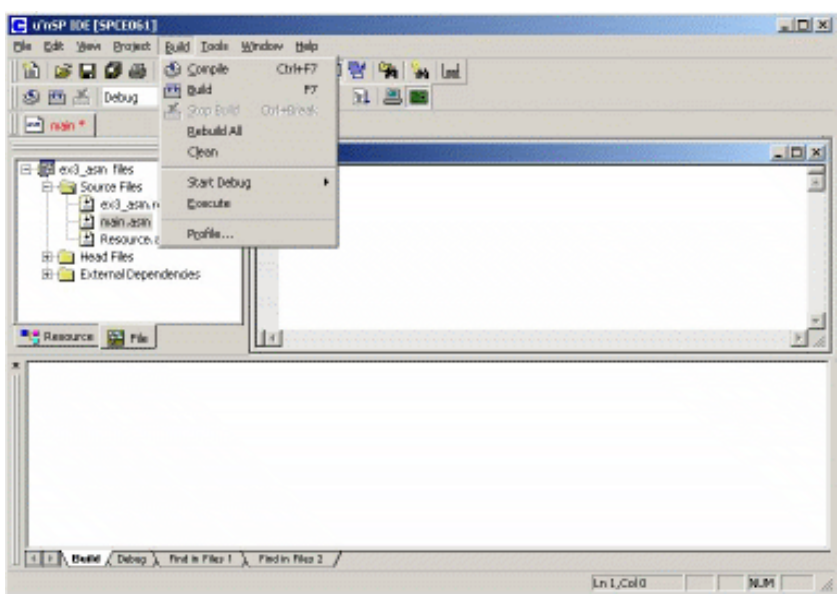


图 10.6 Build 下拉菜单界面

表 10.5　Build 的下拉菜单内容及功能

内容	作用	热键
Compile(编译)	编译目前文件	Ctrl + F7
Build(编译)	编译后连接文件	F7
Stop Build(停止编译)	停止编辑目前文件	Ctrl+ Break
Build All(编译所有文件)	编辑该项目中的所有文件	
Clean(清除)	清除刚编辑过的文件	
Start Debug(开始调试)	调试刚编辑过的文件，包括下载单步调试等	
Execute(执行)	运行文件	
Profile(分析)	详细分析软件执行效率	

6. Tools(工具)

Tools 下拉菜单界面如图 10.7 所示，下拉菜单的内容及功能如表 10.6 所示。



图 10.7　Tools 下拉菜单界面

表 10.6　Tools 的下拉菜单内容及功能

内　容	作　　用
Option(选项)	包括编辑窗口格式设置、库文件的路径设置
Lib Maker(制作库文件)	将所需的.Obj 文件转换成库文件，方便开发时使用

7. Help(帮助)

Help 下拉菜单界面如图 10.8 所示，下拉菜单的内容及功能如表 10.7 所示。



图 10.8　Help 下拉菜单界面

表 10.7 Help 的下拉菜单内容及功能

内 容	作 用
Help Topics (帮助主题)	介绍 IDE 环境
About IDE(关于 IDE)	IDE 的版本号开发公司所占空间

8. Debug(调试)

在调试模式下，菜单栏中多出一个 Debug 菜单，如图 10.9 所示，其内容及功能如表 10.8 所示。

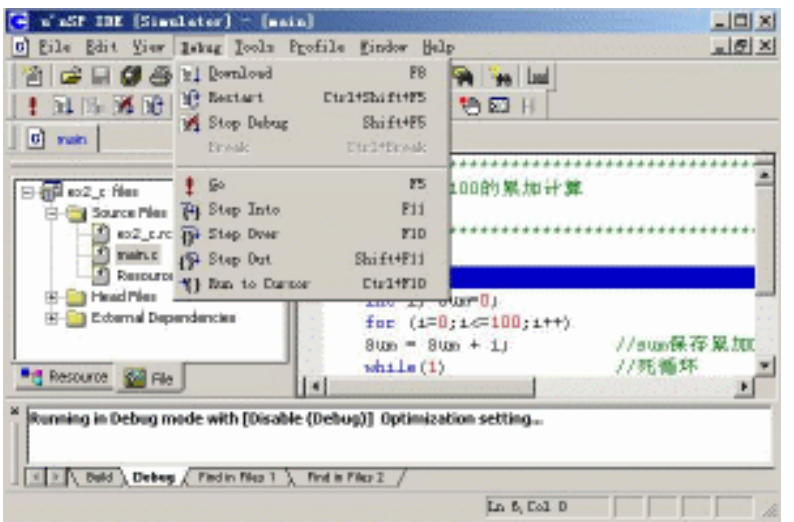


图 10.9 Dubug 下拉菜单界面

表 10.8 Debug 的下拉菜单内容及功能

内 容	作 用	热 键
Download(下载)	将程序文件编译连接生成可执行文件	F8
Restart(复位重新开始)	在调试模式下重新运行程序	Ctrl+Shift+F5
Stop Debug(停止调试)	退出调试模式	Shift+F5
Break(中断)	停止程序运行	Ctrl+ Break
Go(运行)	在调试模式下，运行程序 F5	
Step Into(单步进入)	单步运行时进入子程序	F11
Step Over(单步调过)	单步运行时不进入子程序	F10
Step Out(单步跳出)	单步运行在子程序中时跳出子程序	Shift+F11
Run to Cursor(运行到光标处)	在调试模式下程序全速运行到光标处停止	Ctrl+ F10

10.2　工　具　栏

μ'nSP™IDE 提供了三种工具栏，包括标准、编辑和调试，每种工具栏都有固定和浮动两种形式。把鼠标移到固定形式工具栏中没有图标的地方，按住鼠标左键向下拖动鼠标即可把工具栏变为浮动的，而双击浮动工具栏的标题条则可变为固定工具栏。固定形式的标准工具栏位于菜单栏的下面，它以图标的形式提供了部分常用菜单命令的功能。只要用鼠标单击代表某个命令的图标按钮，就能直接执行相应的菜单命令，工具条中有 38 个图标代表 38 种操作，如图 10.10 所示，大多数图标都有与之等价的菜单命令，图 10.11～图 10.13 是浮动形式的标准、编辑和调试工具栏。表 10.9 列出了工具栏中各图标的作用。

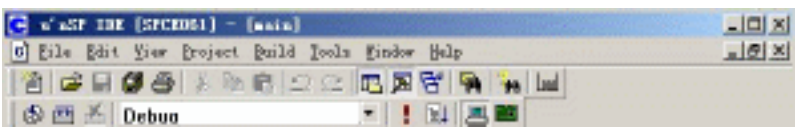


图 10.10　工具栏

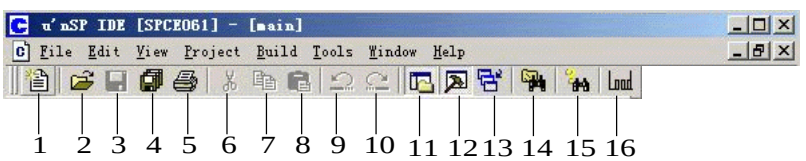


图 10.11　标准工具栏

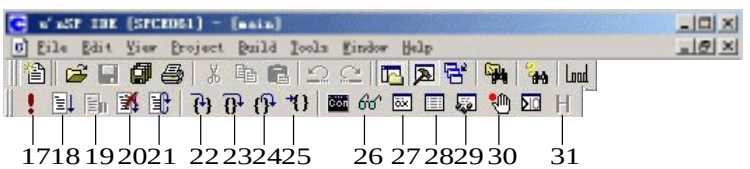


图 10.12　调试工具栏

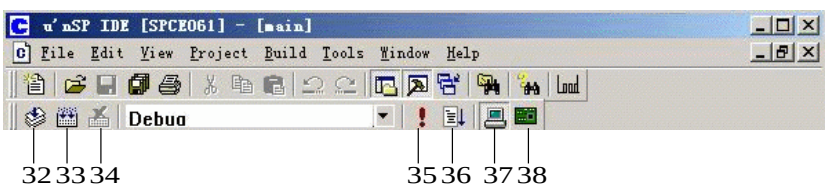


图 10.13　编辑工具栏

表 10.9 工具栏一览表

编号	名称	作用
1	新建	新建项目和文件，相当于 File 菜单中的 New 命令
2	打开	打开项目和文件，相当于 File 菜单中的 Open 命令
3	保存	保存文件，相当于 File 菜单中的 Save 命令
4	全存	保存所有文档，相当于 File 菜单中的 Save All 命令
5	打印	打印当前文件，相当于 File 菜单中的 Print 命令
6	剪切	删除并复制选中的文件内容或文件，相当于 Edit 菜单中的 Cut 命令
7	复制	复制选中的文件内容或文件，相当于 Edit 菜单中的 Copy 命令
8	粘贴	粘贴选中的文件内容或文件，相当于 Edit 菜单中的 Paste 命令
9	撤消键入	取消当前的操作
10	重复键入	对撤消的反操作
11	Workspace 窗口	打开或关闭 Workspace 窗口，相当于 View 菜单中的 Workspace 命令
12	Output 窗口	打开或关闭 Output 窗口，相当于 View 菜单中的 Output 命令
13	窗体布局	打开窗体布局窗口
14	在文件中查找	打开在文件中查找对话框，相当于 File 菜单中的 Find in File 命令
15	帮助主题	打开帮助主题窗口，相当于 Help 菜单中的 Help Topics 命令
16	打开可执行文件	打开可执行文件.s37 或.tsk
17	运行	在调试模式下运行程序，相当于 Debug 菜单中的 Go 命令
18	下载	下载可执行文件，相当于 Debug 菜单中的 Download 命令
19	中断	停止正在运行程序，相当于 Debug 菜单中的 Break 命令
20	停止调试	退出调试模式，相当于 Debug 菜单中的 Stop Debug 命令
21	复位	在调试模式下重新运行程序
22	单步	进入单步运行时进入子程序，相当于 Debug 菜单中的 Step Into 命令
23	Step Over	单步运行时不进入子程序，相当于 Debug 菜单中的 Step Over 命令
24	单步跳出	单步运行在子程序中时跳出子程序，相当于 Debug 菜单中的 Step Out 命令
25	运行到光标处	在调试模式下程序全速运行到光标处停止，相当于 Debug 菜单中的 Run to Cursor 命令
26	变量表窗口	在调试模式下打开变量表窗口，相当于 View 菜单中的 Watch 命令
27	寄存器窗口	在调试模式下打开寄存器窗口，相当于 View 菜单中的 Registers 命令
28	内存窗口	在调试模式下打开内存窗口，相当于 View 菜单中的 Memory 命令
29	反汇编窗口	在调试模式下打开反汇编窗口，相当于 View 菜单中的 Disassembly 命令
30	断点	在调试模式下打开设置断点的对话框，相当于 Edit 菜单中的 Breakpoints 命令
31	历史缓冲区	在仿真模式下打开历史缓冲区窗口
32	编译	编译文件，相当于 Build 菜单中的 Compile 命令
33	编辑	编辑文件，相当于 Build 菜单中的 Build 命令
34	停止编辑	停止编辑文件，相当于 Build 菜单中的 Stop Build 命令
35	运行	在调试模式下运行程序，相当于 Debug 菜单中的 Go 命令
36	下载	下载可执行文件，相当于 Debug 菜单中的 Download 命令
37	本机调试 (使用仿真器)	在本机上调试
38	仿真板上调试 (使用在线仿真)	结合仿真板调试

10.3　窗 口 使 用

前面介绍了标题栏、菜单栏和工具栏，它们所在的窗口称为主窗口。除主窗口外，SPCE061A IDE 编程环境中还有其它一些窗口：Workspace 窗口、Edit 窗口、Output 窗口、Debug 窗口(变量表 Watch 窗口、寄存器 Register 窗口、内存 Memory 窗口、反汇编窗口、Disassemble 窗口、历史缓冲区窗口)、其它窗口(命令窗口、转存窗口)等。

1．Workspace 窗口

在 Workspace 窗口中含有建立一个应用程序所需要的文件清单，其中包括所有与该项目相关资源文件(如语音数据等)和被编辑的程序文件。Workspace 窗口用来切换显示 File 和 Resource 两个视窗。File 视窗主要用来显示源文件组和头文件组中所包含的所有文件。Resource 视窗主要用来显示资源文件组中所包含的所有资源文件。

打开 Workspace 窗口的方法如下：

第一种方法：单击菜单栏 View→Workspace 菜单命令，即可打开或关闭 Workspace 窗口。

第二种方法：单击标准工具栏中的 Toggle Workspace 按钮，也可打开或关闭 Workspace 窗口。

图 10.14 为 Workspace 窗口界面。

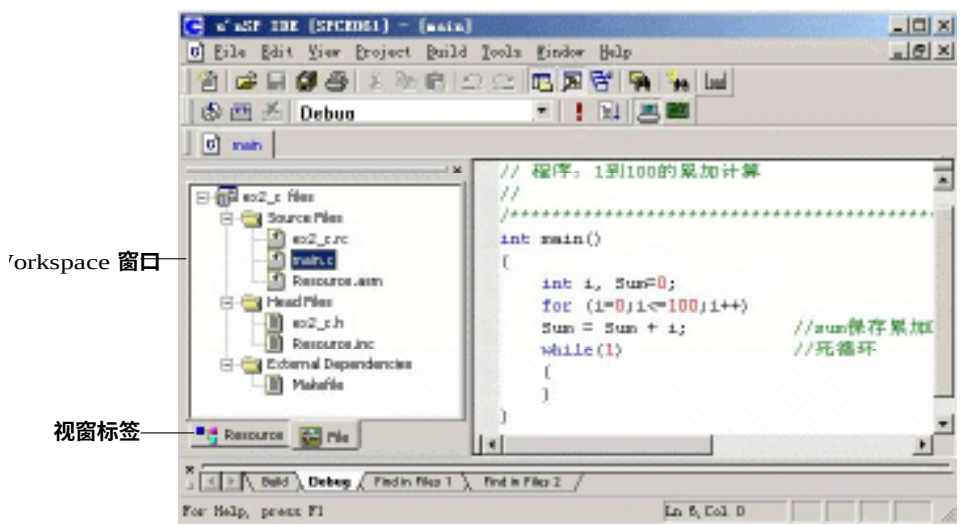


图 10.14　Workspace 窗口界面

通过对 Workspace 窗口中 File 和 Resource 标签的点击，可以切换 File 视窗和 Resource 视窗。图 10.15 是 Workspace 窗口下的 File 视窗界面，图 10.16 是 Workspace 窗口下的 Resource 视窗界面。Workspace 窗口 Resource 视窗界面中的资源文件 RES_D1_24K、RES_WW_24K、RES_TT_24K 为 A2000 格式的语音数据文件。



图 10.15 Workspace 窗口 File 视窗界面



图 10.16 Workspace 窗口 Resource 视窗界面

2. Edit 窗口

在前面介绍桌面时，已经指出 Edit 窗口的位置，Edit 窗口主要是用来键入程序文件和和其它编辑文件的显示。

打开 Edit 窗口的方法：新建任何一文件即可打开 Edit 窗口。例如，单击 File→New→Creat C File 菜单命令即可打开该 C 文件的 Edit 窗口。

Edit 窗口包括文本编辑器和二进制编辑器。

3. 文本编辑器

文本编辑器是用来编辑程序的。在项目中打开一个文件时，文件所有的内容都显示在文本编辑器中。图 10.17 就是文本编辑器的界面。

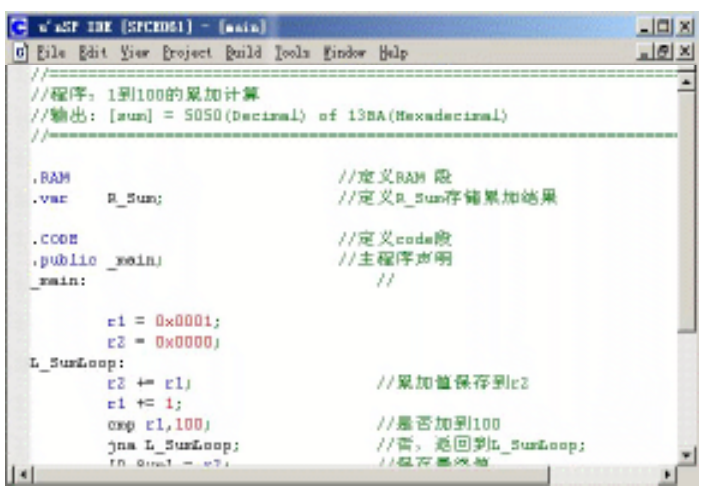


图 10.17 文本编辑器的界面

打开文本文件的方法：

(1) 单击 File→Open 菜单命令，弹出 Open 对话框，选择一文件。

(2) 单击 File→Recent Files 菜单命令，选择一近期文件立即打开。

4. 二进制编辑器

二进制编辑器用来编辑项目中十六进制或 ASCII 格式的二进制代码的资源文件。图 10.18 为二进制编辑器界面。

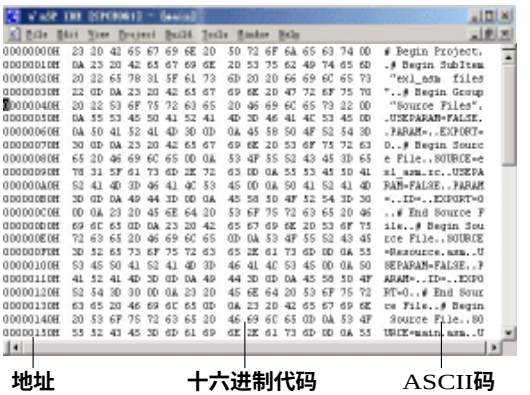


图 10.18　二进制编辑器

打开二进制文件的步骤：

(1) **单击** File→Open，**弹出** Open **对话框。**

(2) **在** Open as **文本框中选择** Binary。

(3) **选择一个二进制文件打开。**

编辑二进制编辑器的步骤如下：

(1) **单击将要修改的二进制文件内容，按数字键可以更改二进制文件内容。**

(2) **保存修改后的内容。**

(3) **在二进制编辑器中有效键为**[/]、[Page Up]、[Page Down]、[Home/End]、[Ctrl +Home]、[Ctrl + End]。

5. Output **窗口**

Output **窗口主要用来显示编辑、调试、查找的输出结果。**

打开 Output **窗口的方法：**

第一种方法：单击菜单栏 View→Output **菜单命令即可打开或关闭** Output **窗口。**

第二种方法：单击标准工具栏中的 Toggle Output **按钮，也可打开或关闭** Output **窗口。**

Oputput **窗口界面如图** 10.19 **所示。**

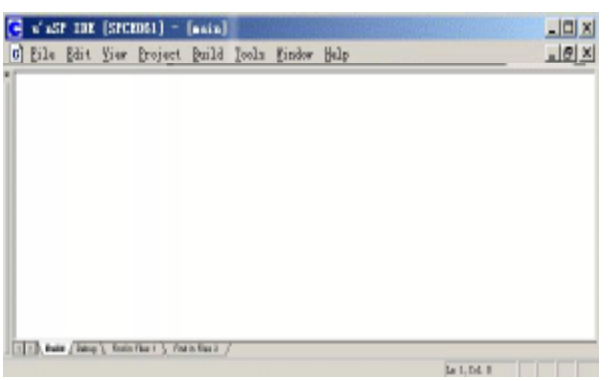


图 10.19　Output **窗口界面**

Output **窗口还可以细分为编辑输出窗口、调试输出窗口和查找输出窗口三种。这三种** Output **窗口可以通过** Output **窗口底部的标签来切换。**

6. 编译输出窗口

编译后在 Output 窗口中显示出编译连接的信息，在编译过程中出现的错误和警告信息也会被列出。当 Output 窗口无错误信息时，说明该程序已被完全成功的编译。

例如，编译 IDE 下 Example 中的 Ex2，编译后的 Output 窗口编辑信息界面如图 10.20 所示。



图 10.20 Outupt 窗口编辑信息界面

7. 调试输出窗口

在 Output 窗口中显示调试信息，通常为调试结束，调试过程采用无优化代码的方法。

例如，编辑 IDE 下 Example 中的 Ex2。编辑后的 Output 窗口调试信息界面如图 10.21 所示。

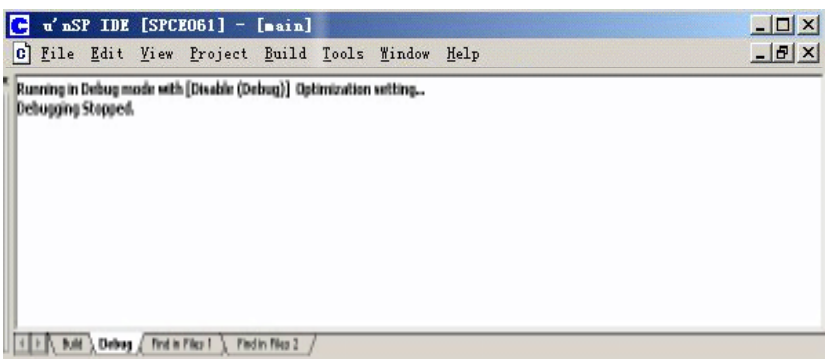


图 10.21 Output 窗口调试信息界面

8. 查找输出窗口

查找输出窗口显示在文件中查找文本的结果。

例如，在 IDE 下 Example 中的 Ex2 查找单词 code。查找后的输出窗口查找信息界面如图 10.22 所示。



图 10.22 Output 窗口查找信息界面

9. Debug 窗口

程序文件经过编译无错后，单击工具栏中的 Download 按钮即可进入调试模式。所有的调试窗口均可在调试模式下打开。

调试窗口主要显示有关的调试信息，在调试模式下，调试菜单显示在主菜单下。

调试窗口包括:

(1) 变量表 Watch 窗口；

(2) 寄存器 Register 窗口；

(3) 内存 Memory 窗口；

(4) 反汇编 Disassemble 窗口；

(5) 历史缓冲区窗口。

1) 变量表 Watch 窗口

变量表 Watch 窗口用于输入并编辑变量、显示变量内容。

打开/关闭变量表窗口的方法：

第一种方法：单击菜单栏中 View→Watch 菜单命令即可打开变量表窗口。

第二种方法：单击调试工具栏中的 Watch 按钮即可打开变量表窗口。

第三种方法：通过热键 Alt+3 即可打开变量表窗口。

变量表 Watch 窗口界面如图 10.23 所示。



图 10.23　变量表 Watch 窗口界面

使用方法：双击变量名，即出现一个文本框，在文本框中写入变量名称则相应的变量值和变量所在地址就可以显示出来，当要删除一变量时，选中该变量所在的文本行，按 Del 键即可删除，或者单击右键选中删除命令也可以删除变量。

注意：选中整行的内容方可删除变量。

2) 寄存器 Register 窗口

寄存器 Register 窗口显示当前常用寄存器和特殊寄存器的内容。

打开/关闭寄存器窗口的方法：

第一种方法：单击菜单栏中 View→Register 菜单命令即可打开寄存器窗口。

第二种方法：单击调试工具栏中的 Register 按钮即可打开寄存器窗口。

第三种方法：通过热键 Alt+4 即可打开寄存器窗口。

CPU 寄存器分为通用型寄存器和专用型寄存器。其中通用型包括 R1～R4，专用型包括 SP、BP、SR 和 PC。

寄存器 Register 窗口界面如图 10.24 所示。

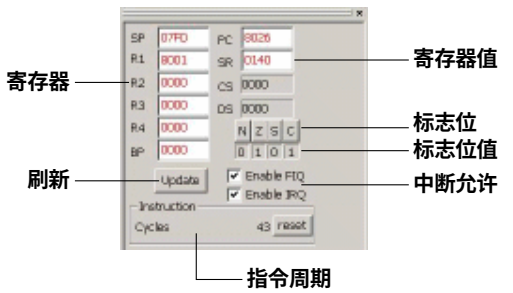


图 10.24　寄存器 Register 窗口界面

使用方法：单击寄存器的文本框即可编辑该寄存器。

注意：可以更改寄存器的值，但 CS 的值最好不要轻易更改，否则会使调试程序出错。

3) 内存 Memory 窗口

内存 Memory 窗口显示内存的内容。

打开/关闭内存窗口的方法：

第一种方法：单击菜单栏中 View→Memory 菜单命令即可打开内存窗口。

第二种方法：单击调试工具栏中的 Memory 按钮即可打开内存窗口。

第三种方法：通过热键 Alt+5 即可打开内存窗口。

内存 Memory 窗口界面如图 10.25 所示。

使用方法：在地址的文本框中直接写入要查找的地址，回车确认后内存窗口会自动转到查找的地址处。

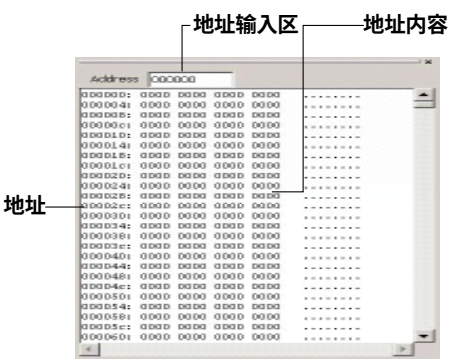


图 10.25　内存 Memory 窗口界面

4) 反汇编 Disassemble 窗口

反汇编 Disassemble 窗口显示反汇编内容。

打开/关闭反汇编窗口的方法：

第一种方法：单击菜单栏中 View→Disassemble **菜单命令即可打开反汇编窗口。**

第二种方法：单击调试工具栏中的 Disassemble **按钮即可打开反汇编窗口。**

第三种方法：通过热键 Alt+6 **即可打开反汇编窗口。**

反汇编 Disassemble **窗口界面如图** 10.26 **所示。**

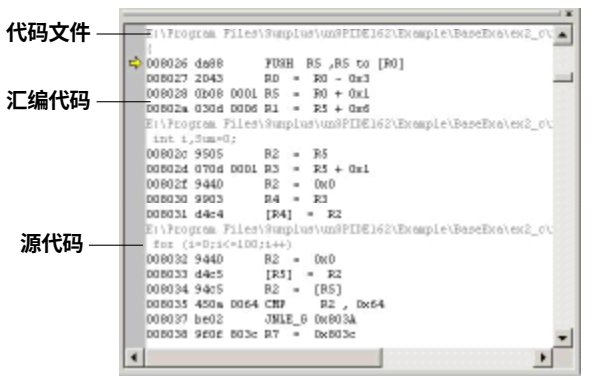


图 10.26　**反汇编窗口** Disassemble **窗口界面**

5) **历史缓冲区窗口**

在仿真模式下执行完程序后，被执行的指令状态、内存内容将被存储到历史缓冲区中。激活历史缓冲区的方法是，单击菜单栏中 Project→Setting **菜单命令，弹出** Setting **对话框，在** General **标签下单击** reset **即可激活** PC Trace Enable**。**

文件编译执行后，在调试环境下单击 H **按钮，即可打开历史缓冲区窗口，被调试的程序的汇编码显示在历史缓冲区窗口内。如果您注意观察，就会发现这时在项目文件夹中多了一个** .his **文件，即历史文件。历史缓冲区窗口界面如图** 10.27 **所示。**

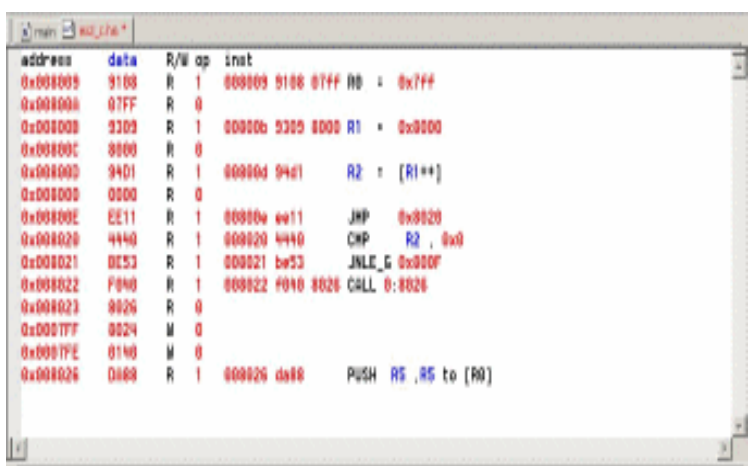


图 10.27　**历史缓冲区界面**

历史缓冲区的界面与反汇编窗口内容很相似，但反汇编界面比历史缓冲区界面多一项程序源代码，以方便开发者更清楚地查看反汇编程序，而历史缓冲区界面比反汇编窗口界面多一项读/写(R/W)**，方便开发者了解每一指令的具体操作。**

10. 其它窗口

1) 命令窗口

单击 View 菜单下的 Command 命令，打开命令窗口，在该窗口列表框下面的文本输入框中键入帮助字符 H，并确认，这时会在列表中列出 IDE 的所有命令及相应功能描述。图 10.28 所示为打开的命令窗口界面。



图 10.28 命令窗口界面

在文本框中键入 H 后，列出命令及相应功能描述界面如图 10.29 所示。

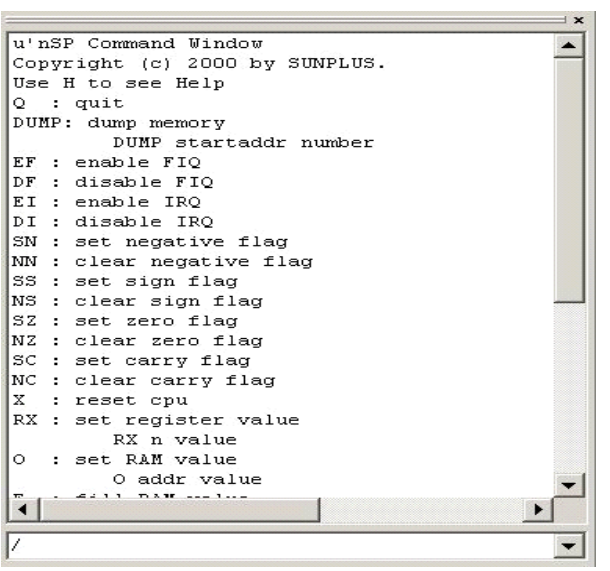


图 10.29 文本框中键入 H 后界面

IDE 的命令及其功能描述如表 10.10 所示。

表 10.10　IDE 的命令及其功能

命令	功能描述	语法格式及语例
Q	退出 μ'nSP™ IDE	
Dump	转储内存中的字数据	Dump <起始地址> <转储字数> Dump 100 100 //转储 0x100～0x1FF 中的字数据
EF	允许产生 FIQ 中断	
DF	禁止产生 FIQ 中断	
EI	允许产生 IRQ 中断	
DI	禁止产生 IRQ 中断	
SN	设置负标志	
NN	清除负标志	
SS	设置符号标志	
NS	清除符号标志	
SZ	设置零标志	
NZ	清除零标志	
SC	设置进位标志	
NC	清除进位标志	
X	复位(程序指针指向复位向量中的地址)	
RX	设定寄存器的值	Rx <寄存器号> <设定值> Rx 3 abcd //将 R3 的值设为 0xabcd
O	设定内存单元中的值	O <内存地址> <设定值> O 7016 abcd //将 0x7016 单元的值设为 0xabcd
F	设定内存区中的值	F <内存起始地址> <内存结束地址> <设定值> F 100 1FF 1234 //将 0x100～0x1FF 单元填入 0x1234
BC	清除断点	BC <断点地址> <断点标志> <断点数据> BC 8000 8082 1234 //清除当向 0x28000 单元中写入数据 0x1234 时的条件断点
BP	设置断点	BP <断点地址> <断点标志> <断点数据> BP 8000 8082 1234 //设置当向 0x28000 单元中写入数据 0x1234 时的条件断点
G	连续运行程序	
S	单步运行程序	
L	将二进制文件装入内存	L <文件名> <起始地址> <结束地址> <内存的起始地址> L te t b n 100 1FF 8000// 将 te t b n 文件中第 0x100～0x1FF 单元的//数据装入内存 0x8000 单元
RF	将内存中的数据内容转储到文件中	RF <起始地址> <内存单元个数> <文件名> RF 100 100 te t b n //将 0x100～0x1FF 单元的内容转储至 te t b n 文件中
	显示命令帮助信息	显示 μ'nSP™ IDE 的所有命令及及其内容描述

命令检索：用鼠标左键点中列表框中的某一命令，在 PC 机键盘上每敲入该命令的头一个字符时，列表框中当前命令的指向会在所有首字符与敲入字符的命令之间移动，据此功能可在列表框列出的诸多命令中迅速检索到所需的命令。

命令操作：按照列表框中列出的命令格式，在文本输入框中正确键入某命令字符并确

认，该命令便会被执行。

2) 转存窗口

在调试模式下，单击 Tools→Dump Memory，进入转存窗口。该窗口用于存储指定地址范围的内容到指定的文件中，另外它也可以将高字节和低字节分别指定的地址范围存储到两个文件中。转存窗口界面如图 10.30 所示。

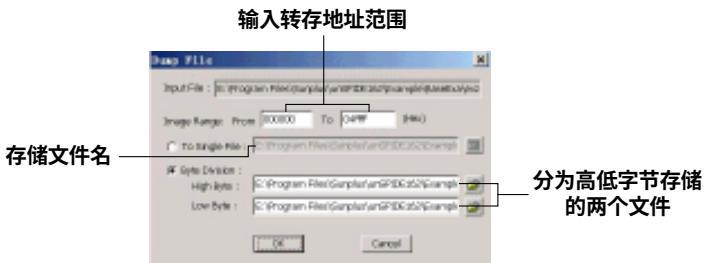


图 10.30 转存窗口界面

10.4 项 目 管 理

在 C、C++、VB 等语言中广泛使用“ Project”一词，在译成中文时有的译成“ 项目 ”，有的译成“ 工程 ”，在这里我们作为“ 项目 “ 。开发一个应用程序需要很多文件，这些文件需要规范管理，一组相关文件就构成了一个项目，项目是可以独立执行的程序单元，一个应用程序可以是一个单独的项目，在项目中可以含有不同的元组和文件。准确地说，项目是指为用户调试程序建立的一个开发环境，提供用户程序及资源文档的编辑和管理，并提供环境要素的设置途径，最后将通过用户程序及库的编制(包括编译、汇编以及链接等)提供一个良好的调试环境。

10.4.1 建立项目

新建项目的操作步骤如下：

(1) 用鼠标左键单击 File 下拉菜单 New，弹出 New 对话框，如图 10.31 所示。

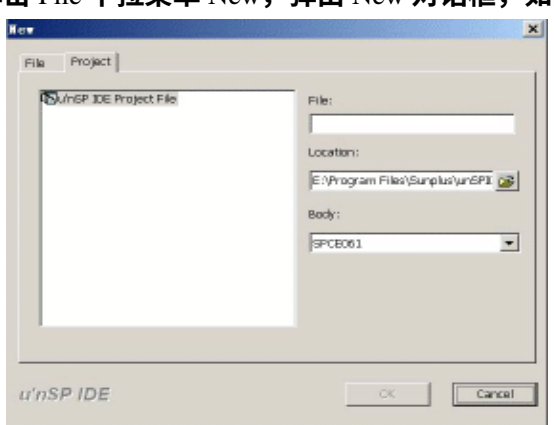


图 10.31 New 对话框

(2) 在该窗口中，选中 Project 标签并在 File 的文本框中键入项目的名称。

(3) 在 Location 下的文本框中输入项目的存取路径，或利用该文本框右端的浏览按钮制定项目的存储位置。

(4) 用鼠标左键单击 New 对话框里的 OK 按钮，则项目建立完成。

新建项目的需求：在做一个应用程序前，首先是新建项目。例如：

项目名称：Example1

项目位置：E:\ZxyProgram\ Example1

新建项目后的 Workspace 窗口结果：生成了新项目 Example1 文件，如图 10.32 所示。

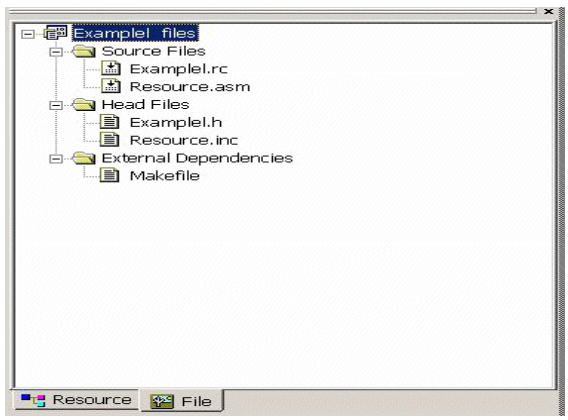


图 10.32　新建项目后的 Workspace 窗口

10.4.2　在项目中新建 C 文件(.C)

新建 C 文件的方法：在新建项目下，点击菜单 File 下拉菜单 New，弹出 New 对话框，如图 10.33 所示。点击 μ'nSP IDE C File，在 File 下的文本框内键入文件名称，单击 OK 按钮确认。

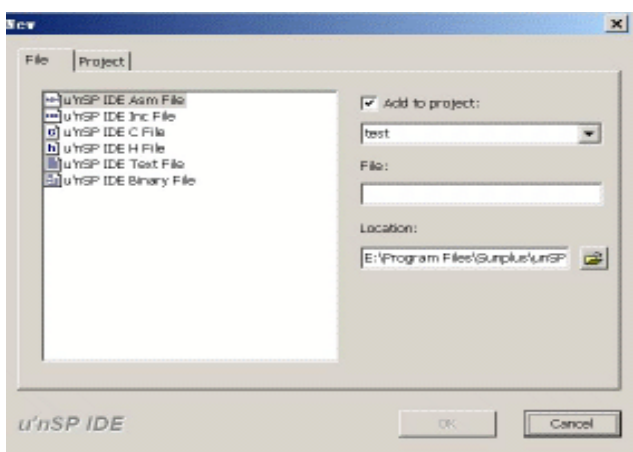


图 10.33　New 对话

新建 C 文件的需求：用 C 语言做程序时需要建立 C 文件类型。例如：

文件名称：Exa1

文件位置: E:\ZxyProgram\Example1\Exa1.c

新建 C 文件后的 Workspace **窗口结果**: Source File **下多出一个** Exa1.c **文件，如图** 10.34 **所示。**

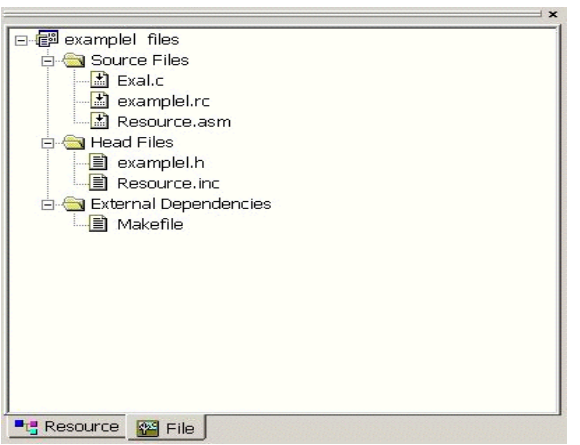


图 10.34 **新建 C 文件后的** Workspace **窗口**

10.4.3 在项目中新建汇编文件(.asm)

新建汇编文件的方法 :在新建项目下，点击菜单 File **下拉菜单** New**，弹出** New **对话框。点击** μ'nSP IDE ASM File**，在** File **下的编辑框内写入文件名称，单击** OK **按钮确认，如图** 10.33 **所示。**

新建汇编文件需求：用汇编语言做程序时需要建立汇编文件类型。例如：

文件名称：Exa1

文件位置：E:\ZxyProgram\ Example1\Exa1.ASM

新建汇编文件后 Workspace **窗口结果**: Source File **下多出一个** Exa1.asm **文件，如图** 10.35 **所示。**

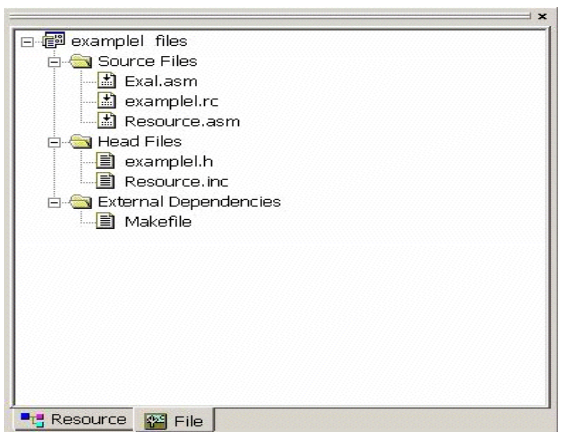


图 10.35 **新建汇编文件后的** Workspace **窗口**

10.4.4　在项目中新建头文件(.H)

新建头文件的方法：在新建项目下，点击菜单 File **下拉菜单** New**，弹出** New **对话框。点击** μ'nSP IDE H File**，在** File **下的编辑框内写入文件名称，单击** OK **按钮确认，如图** 10.33 **所示。**

新建头文件需求：多个文件共享的文件可以建成头文件。例如：

文件名称：head

文件位置：E:\ZxyProgram\ Example1

新建头文件后的 Workspace **窗口结果:** Head File **下多出一个** head.h **文件，如图** 10.36 **所示。**

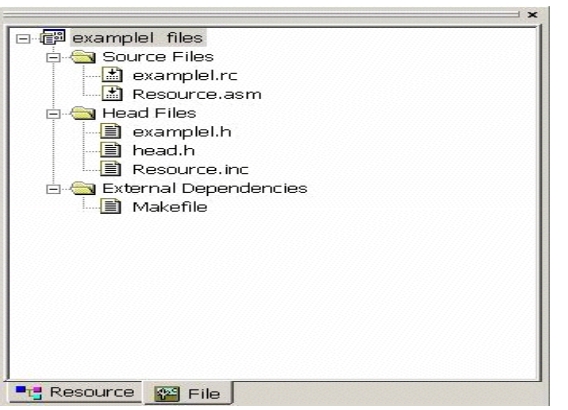


图 10.36　**新建头文件后的** Workspace **窗口**

10.4.5　在项目中新建文本文件(.txt)

新建文本文件的方法:　在新建项目下，点击菜单 File **下拉菜单** New**，弹出** New **对话框。点击** μ'nSP IDE Text File**，在** File **下的编辑框内写入文件名称，单击** OK **按钮确认，如图** 10.33 **所示。**

新建文本文件的需求：对程序文件做文档说明时，可以建文本文件类型。例如：

文件名称：text

文件位置：E:\ZxyProgram\ Example1

新建文本文件后的 Workspace **窗口结果：**External Dependencies **下多出一个** text.txt **文件，如图** 10.37 **所示。**

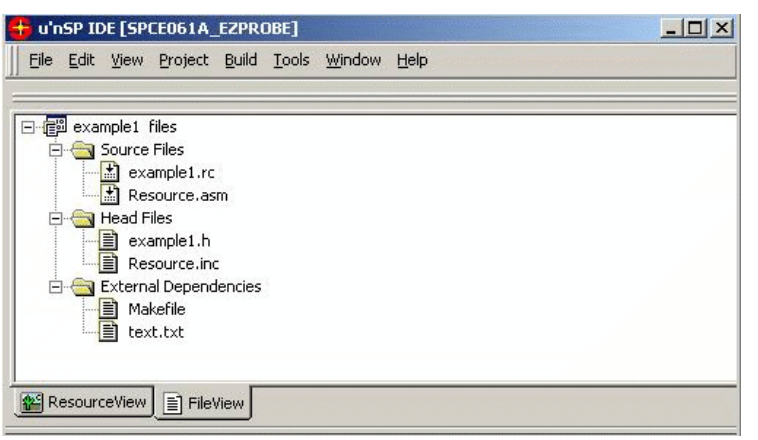


图 10.37　**新建文本文件后的** Workspace **窗口**

10.4.6 在项目中新建二进制文件

新建二进制文件的方法: **在新建项目下**, **点击菜单** File **下拉菜单** New，**弹出** New **对话框。点击** μ'nSP IDE Binary File，**在** File **下的编辑框内写入文件名称，单击** OK **按钮确认，如图** 10.33 **所示。**

新建二进制文件的需求：在做资源文件时，需建立二进制文件类型。例如：

文件名称：bin

文件位置：E:\ZxyProgram\ Example1

新建二进制文件后的 Workspace **窗口结果**: External Dependencies **下多出一个** bin.bin **文件，如图** 10.38 **所示。**

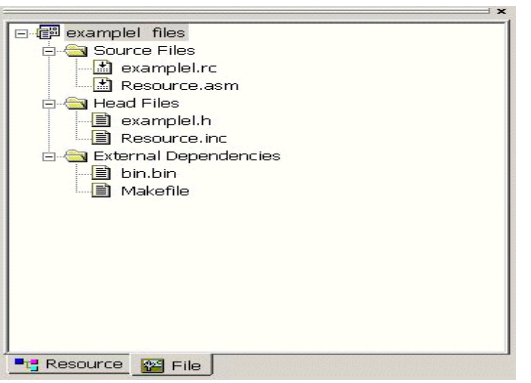


图 10.38 **新建二进制文件后的** Workspace **窗口**

10.4.7 在项目中添加/删除文件

1. **在项目中添加文件的方法**

在项目中添加文件的第一种方法：通过 Project **菜单途径用鼠标左键单击** Project **菜单里** Add to Project **选项中的** Files **或** Resource **子项，激活** Add Files **对话框。**

在项目中添加文件的第二种方法：通过 Workspace **窗口，具体步骤如下：**

(1) **在** Workspace **窗口内选中元组，单击右键弹出下拉菜单，如图** 10.39 **所示。**

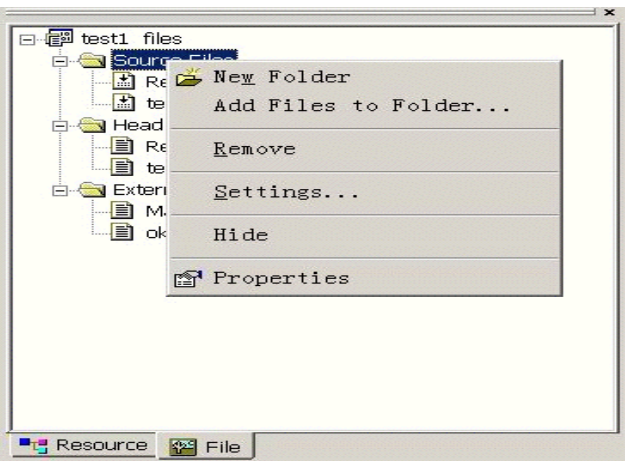


图 10.39 **添加文件下拉菜单界面**

(2) **用鼠标左键单击** Add Files To Folder… **选项可激活** Add Files **对话框，如图** 10.40 **所示。**

(3) **在文本框中键入将添加的文件，单击打开按钮即将添加的文件加到所选的元组中。**

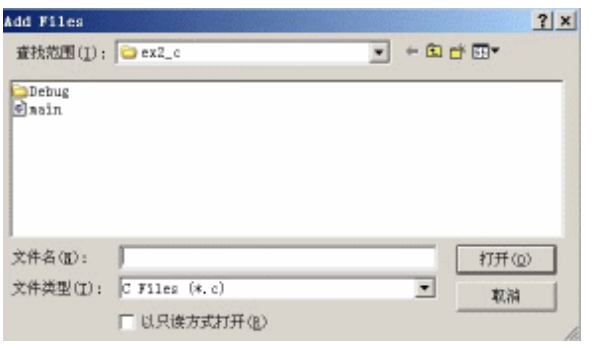


图 10.40　添加文件对话框界面

2. 删除文件步骤

删除文件的具体步骤如下：

(1) 在 File 视窗或 Resource 视窗里选中元组中的某个文件。

(2) 单击鼠标右键，在弹出的下拉菜单(如图 10.39 所示)中选中 Remove 选项，则该文件会从元组中被删除。

10.4.8　在项目中使用资源

在项目的资源元组中添加资源文件时，该资源文件的存储路径及名称会自动被记入项目中的.rc 文件中，并以 RES_*的缺省文件名格式被赋予一个新的文件名(此处“*”是指资源文件在其存储路径上的文件名)。同时，添入的资源文件还会被安排一个文件标识符 ID。

10.4.9　项目选项的设置

项目选项的设置是针对不同目标而对开发环境的各个要素进行的设置，其设置界面如图 10.41 所示。

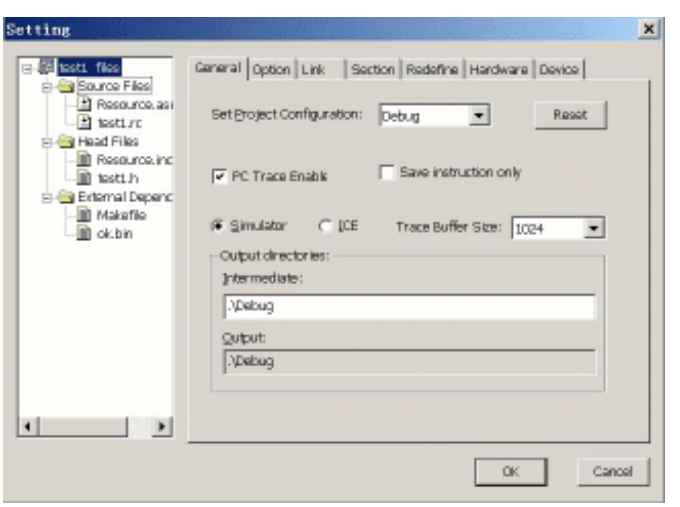


图 10.41　项目选项设置界面

根据界面中的这些标签，进入相应的属性页，进行项目的各项设置。

(1) General **属性页如图** 10.42 **所示。**

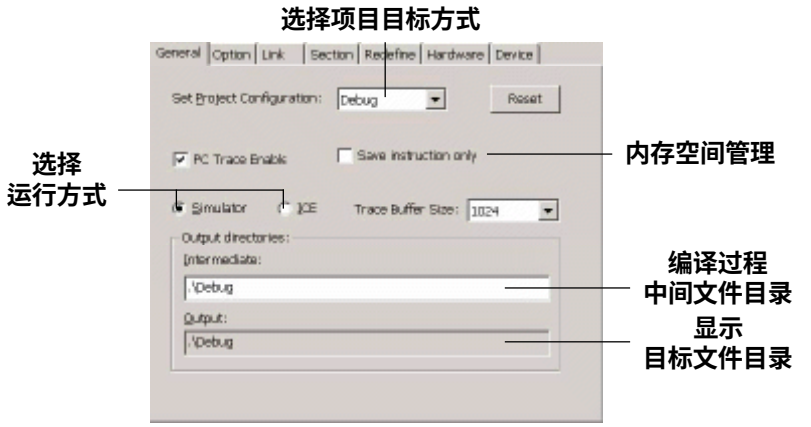


图 10.42　General **属性页**

(2) Option **属性页如图** 10.43 **所示。**

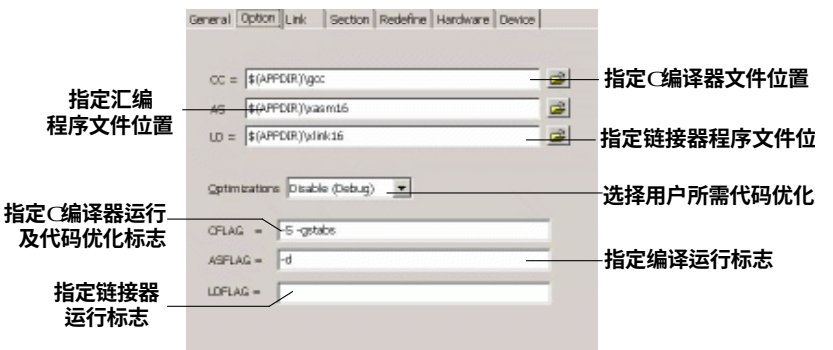


图 10.43　Option **属性页**

(3) Link **属性页如图** 10.44 **所示。**



图 10.44　Link **属性页**

(4) Section 属性页如图 10.45 所示。

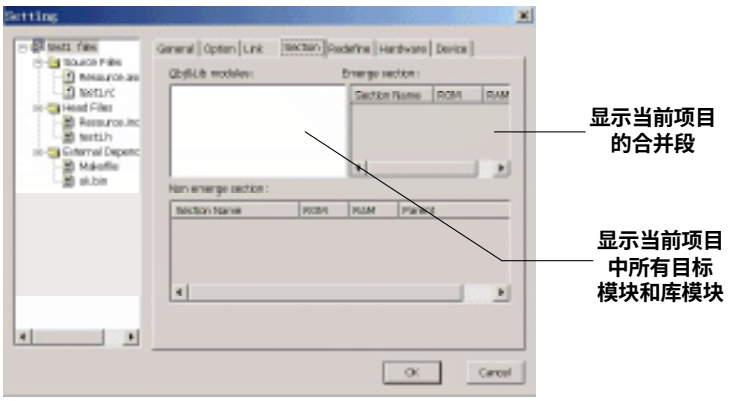


图 10.45　Section 属性页

(5) Hardware 属性页如图 10.46 所示。

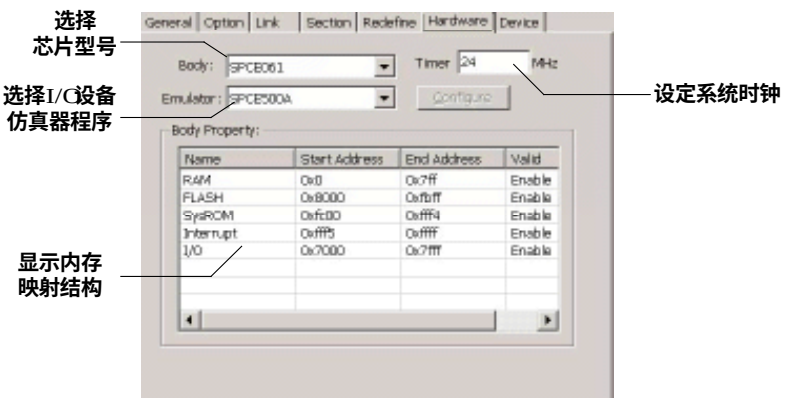


图 10.46　Hardware 属性页

(6) Device 属性页如图 10.47 所示。

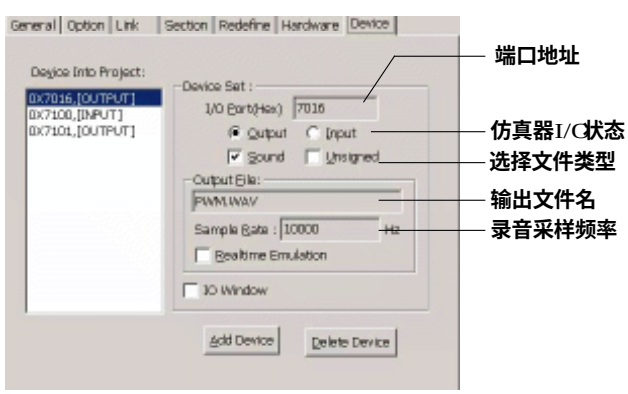


图 10.47　Device 属性页

10.4.10 项目的编译

当项目中的文件编写结束后，要对项目中的程序进行编译并将编译出来的二进制代码与库中的各个模块连接成一个完整的、地址统一的、可执行目标文件和符号表文件，供用户调试使用。在这里要使用编译器、汇编器、链接器等工具。

项目编译的基本操作如下：

(1) Compile 对编辑窗口中当前文件进行编译。

(2) Build 编制当前的文件。

(3) Rebuild All 重新编制当前项目目标，将处理当前项目中的所有文件。

(4) Stop Build 终止当前项目目标编制。

1. Compile/Build/Rebuild All/Stop Build 的方法

单击 Build 菜单，弹出下拉菜单包括 Compile/Build/Rebuild All/Stop Build 命令，或者在 Build 工具栏中也可以找到这几个工具。

2. Compile/Build/Rebuild All/Stop Build 后的结果

编制过程中的一些操作信息将显示在输出窗口的 Build 视窗中，如图 10.48 所示。



图 10.48 编辑后输出窗口的 Build 视窗

10.5 代码剖视器

μ'nSP™IDE 的代码剖视器是一个强有力的分析工具，通过应用此工具来剖析、优化程序代码。具体功能如下：

(1) 提供代码优化的准确信息，对部分程序进行诸多重要因素的剖析，包括某段程序花费了多少个指令周期的执行时间，程序中的标号流等一些有助于提高程序效率的信息；

(2) 检测并分析程序运行当中使用算法的有效性之高低；

(3) 检查用户程序的代码段是否面临处在系统测试程序区的危险。

10.5.1 激活 Profile 的方法

在非调试情况下，用鼠标左键单击 Build 菜单的 Profile 选项，激活 Profile Configure 对话框，如图 10.49 所示。

在调试情况下，直接点击菜单栏中的 Profile 菜单命令即可激活 Profile Configure 对话框，如图 10.49 所示。

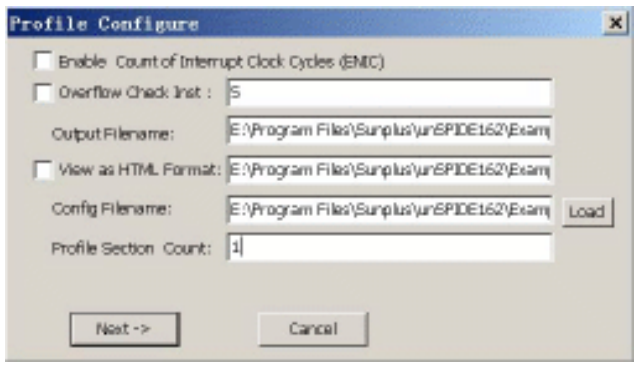


图 10.49　Profile Configure 对话框 1 界面

Profile Configure 对话框 1 中设置选项及其内容如表 10.11 所示。

表 10.11　Profile Configure 对话框 1 中设置选项及其内容

设置项	设置形式	设置内容描述
Enable Count of Interrupt Clock Cycles (ENIC)	复选框	ENIC 选项是为在 IRQ 中断服务子程序中仍可连续剖视代码所设，若要求剖视的代码段处于 IRQ 子程序中则须选中此项
Overflow Check Inst	文本输入框	设定当运算产生溢出时，多少个指令内未检查溢出标志便产生警告信息
Output Filename	文本输入框	指定容纳最终的剖视结果的文件名称
View as HTML Format	复选框及文本框	选择是否需以网页格式来查看、剖视代码结果，若需要，则应在文本框中指定网页格式的剖视结果文件名称
Config Filename	文本输入框	指定存储配置参数的文件名称，用于每次调试开始时重新装入
Profile Section Count	文本输入框	指定需要剖视程序段的段数

10.5.2　使用 Profile 的步骤

使用 Profile 的具体步骤如下：

(1) 根据对话框选项的介绍，设置对话框 1 的选项，单击 Next 按钮。如 Profile Section Count 选项设为 1。

(2) 出现对话框 2 ，如图 10.50 所示，设置 Profile 程序的停止地址，如 8df2，单击 NEXT 按钮。

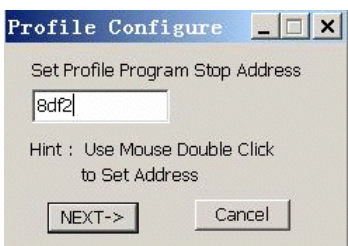


图 10.50　Profile Configure 对话框 2 界面

(3) 设置 Profile 第一部分的起始地址，如图 10.51 所示，如 8deb，单击 NEXT 按钮。

(4) 设置 Profile 第一部分的停止地址，如图 10.52 所示，如 8DEd，单击 NEXT 按钮。

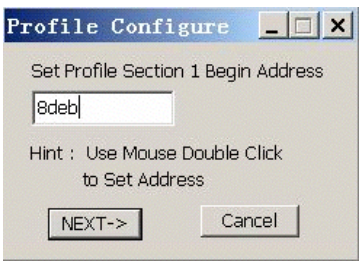


图 10.51 Profile Configure 对话框 3 界面



图 10.52 Profile Configure 对话框 4 界面

(5) 弹出对话框 5，如图 10.53 所示，单击 Profile 按钮。开始编译并弹出剖视结果信息窗口，如图 10.54 所示。

(6) 停止剖视器操作并关闭剖视结果信息窗口的方法有两种：

方法一：在 MS-DOS 窗口下按 PC 机键盘的任意键，便可关闭该窗口且结束剖视器的操作。

方法二：单击窗口右上角的关闭按钮，直接关闭剖视结果信息窗口。

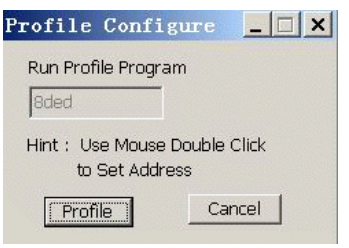


图 10.53 Profile 对话框 5 的界面

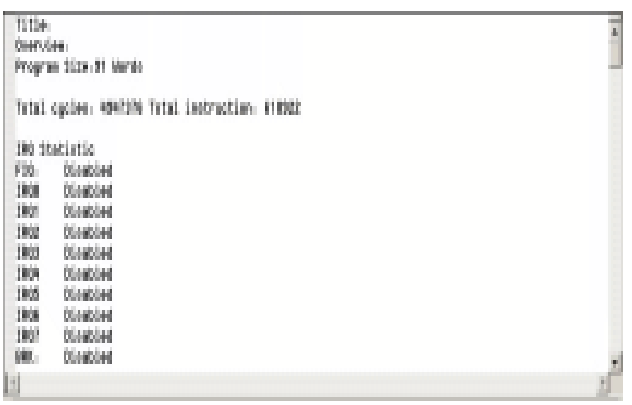


图 10.54 显示剖视结果信息的窗口

例 1 编写 1+2+ … +100 的计算结果。

```
.RAM                                    //定义 RAM 段
.VAR R_Sum;                             //定义变量 R_Sum 保存累加结果
.CODE                                   //定义 CODE 段
.PUBLIC _main;                          //MAIN 的声明
_main:
        r1 = 0x0001;
        r2 = 0x0000;
```

```
L_SumLoop:
        r2 += r1;                    //累加值放到 R2 中
        r1 += 1;                     //下一个被加数
        CMP r1,100;                  //被加数是否为 100?
        JNA L_SumLoop;               //如果 r1≤100 返回到 L_SumLoop
        [R_Sum] = r2;                //将最终累加结果保存到 R_Sum 中
L_ProgramEndLoop:                    //循环
        JMP L_ProgramEndLoop;
```

方法与步骤：

(1) **新建项目**，**项目名称为** EX1。

(2) **该项目下新建汇编文件**，**文件名称为** EX1。

(3) **在汇编文件中键入范例源代码，如图** 10.55 **所示。**

(4) **保存项目。**

(5) **编译：单击** Build→Compile **菜单命令，该程序检查是否有语法错误，如果无错继续，否则改错。**

(6) **编辑：单击** Build→Build **菜单命令，如果无错继续，否则改错。**

(7) **将程序下载到本机中调试，单击编辑工具栏中** Download **命令按钮，进入调试状态。**

(8) **在调试状态下打开寄存器变量等调试窗口，单步执行，仔细观察寄存器和变量的变化。**

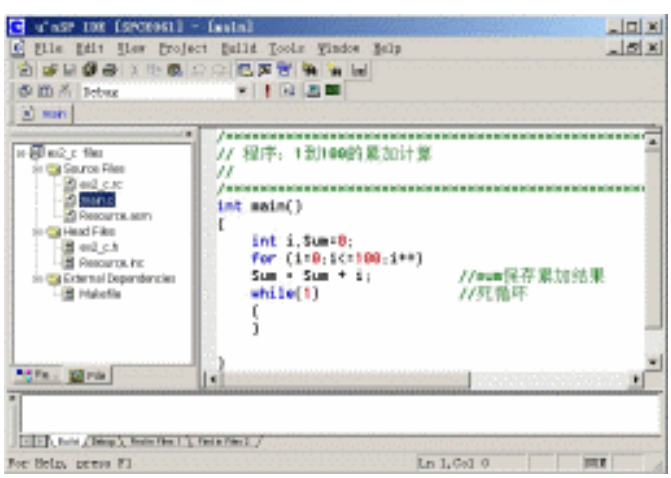


图 10.55　**汇编文件范例程序界面**

附　录

附录 A　端口功能总述

配置单元	读写属性	存储地址	配置功能说明
P_IOA_Data	读/写	7000H	写入数据到数据寄存器里,读出 I/O A 口管脚上的电平状态
P_IOA_Buffer	读/写	7001H	写入数据到数据寄存器里，读出数据寄存器里的数据
P_IOA_Dir	读/写	7002H	I/O A 口方向向量
P_IOA_Attrib	读/写	7003H	I/O A 口属性向量
P_IOA_Latch	读	7004H	进入睡眠前锁存 A 口的数据为睡眠后触键引起唤醒做准备
P_IOB_Data	读/写	7005H	写入到数据寄存器，读出 I/O B 口管脚上的电平状态
P_IOB_Buffer	读/写	7006H	写入到数据寄存器，读出数据寄存器里的数据
P_IOB_Dir	读/写	7007H	I/O B 口方向向量
P_IOB_Attrib	读/写	7008H	I/O B 口属性向量
P_FeedBack	写	7009H	通过反馈电路将外部 RC 振荡源引入
P_TimerA_Data	读/写	700AH	I/O A 口数据口
P_TimerA_Ctrl	写	700BH	I/O A 口控制单元
P_TimerB_Data	读/写	700CH	I/O B 口数据口
P_TimerB_Ctrl	写	700DH	I/O B 口控制单元
P_Timebase_Setup	写	700EH	时基发生器选频设置单元
P_Timebase_Clear	写	700FH	时基计数器复位单元
P_INT_Ctrl	读/写	7010H	中断源控制单元
P_INT_Clear	写	7011H	中断源中断请求清除单元
P_Watchdog_Clear	写	7012H	看门狗计数器复位单元
P_SystemClock	写	7013H	系统时钟选择单元
P_ADC	读/写	7014H	ADC 数据口
P_ADC_Ctrl	读/写	7015H	ADC 控制单元
P_DAC2	读/写	7016H	DAC2 数据口
P_DAC1	读/写	7017H	DAC1 数据口
P_IR_Ctrl	读/写	7018H	红外通信控制单元
P_LVD_Ctrl	读/写	7019H	低电压监测的控制单元
P_SIO_Data	读/写	701AH	串行设备接口 SIO 的数据口
P_SIO_Addr_Low	读/写	701BH	SIO 低字节地址单元
P_SIO_Addr_Mid	读/写	701CH	SIO 中字节地址单元
P_SIO_Addr_High	读/写	701DH	SIO 高字节地址单元
P_SIO_Ctrl	读/写	701EH	SIO 控制单元
P_SIO_Start	读/写	701FH	SIO 数据传输启动单元
P_SIO_Stop	写	7020H	SIO 数据传输结束单元
P_UART_Command1	写	7021H	通用异步串行 I/O 口 UART 控制单元
P_UART_Command2	读/写	7022H	UART 接收/发送功能开通/关断控制单元
P_UART_Data	读/写	7023H	UART 数据口
P_UART_BaudScalarLow	读/写	7024H	UART 波特率设定控制字低字节
P_UART_BaudScalarHigh	读/写	7025H	UART 波特率设定控制字高字节
P_DAC_Ctrl	读/写	702AH	双 DAC 音频输出和单 PWM 驱动控制单元
P_ADC_MUX_Ctrl	读/写	702BH	ADC 多通道控制
P_ADC_MUX_Data	读	702CH	读出多通道 10 位 ADC 转换数据
P_INT_Ctrl_New	读/写	702DH	激活和屏蔽中断

附录 B　SPCE061A 指令集

注意：在以下表格里，若 Rd=PC，则指令周期数为表中所列两者中较长者，此时运算后所有标志位均不受影响。

ALU-ADC

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd=X+Y+C	单字	Rd += IM6，Carry Rd = Rd + IM6，Carry	3	IM6	√	√	√	√
	双字	Rd = Rs + IM16，Carry	6/8	IM16				
	单字	Rd += [BP + IM6]，Carry Rd=Rd+[BP + IM6]，Carry	8	[BP+IM6]				
		Rd += [A6]，Carry Rd = Rd + [A6] Carry	6/8	[A6]				
	双字	Rd = Rs + [A16]，Carry	9/11	[A16]				
	单字	Rd += Rs，Carry	3/8	R				
		Rd += {D:}[Rs]，Carry Rd += {D:}[++Rs]，Carry Rd += {D:}[Rs--]，Carry Rd += {D:}[Rs++]，Carry	7/9	[R]				

ALU-ADD

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd = X+Y	单字	Rd += IM6 Rd = Rd + IM6	3	IM6	√	√	√	√
	双字	Rd = Rs + IM16	6/8	IM16				
	单字	Rd += [BP + IM6] Rd=Rd+[BP + IM6]	8	[BP+IM6]				
		Rd += [A6] Rd = Rd + [A6]	6/8	[A6]				
	双字	Rd = Rs + [A16]	9/11	[A16]				
	单字	Rd += Rs	3/8	R				
		Rd += {D:}[Rs] Rd += {D:}[++Rs] Rd += {D:}[Rs--] Rd += {D:}[Rs++]	7/9	[R]				

ALU-AND

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd = X&Y	单字	Rd &= IM6 Rd = Rd & IM6	3	IM6	√	√	–	–
	双字	Rd = Rs & IM16	6/8	IM16				
	单字	Rd &= [BP + IM6] Rd=Rd &[BP + IM6]	8	[BP+IM6]				
		Rd &= [A6] Rd = Rd & [A6]	6/8	[A6]				
	双字	Rd = Rs & [A16]	9/11	[A16]				
	单字	Rd &= Rs	3/8	R				
		Rd &= {D:}[Rs] Rd &= {D:}[++Rs] Rd &= {D:}[Rs−−] Rd &= {D:}[Rs++]	7/9	[R]				

ALU-ASR

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd#=Rs ASRnn{,Carry}	单字	Rd+=Rs ASR n{,Carry} Rd-=Rs ASRnn{,Carry} CMP Rd, Rs ASR nn	3/8	R	√	√	√	√
CMP Rd,Rs ASR nn TEST Rd,Rs ASR nn		Rd = - Rs ASR nn Rd &= Rs ASR nn Rd |= Rs ASR nn Rd ^= Rs ASR nn TEST Rd, Rs ASR nn Rd = Rs ASR nn	3/8	R	√	√	–	–

ALU-MUI

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
MR = Rd * Rs	单字	MR = Rd * Rs {, ss} MR = Rd * Rs, us	12	R	–	–	–	–

ALU-CMP

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
X−Y 只影响 N、Z、S、C 标志	单字	CMP Rd IM6	3	IM6	√	√	√	√
	双字	CMP Rd IM16	6/8	IM16				
	单字	CMP Rd [BP + IM6]	8	[BP+IM6]				
		CMP Rd [A6]	6/8	[A6]				
	双字	CMP Rs [A16]	9/11	[A16]				
	单字	CMP Rd Rs	3/8	R				
		CMP Rd {D:}[Rs] CMP Rd {D:}[++Rs] CMP Rd {D:}[Rs−−] CMP Rd {D:}[Rs++]	7/9	[R]				

ALU-LSL

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd#=Rs LSL nn{,Carry} CMP Rd, Rs LSL nn TEST Rd, Rs LSL nn	单字	Rd+=Rs LSL n{,Carry} Rd-=Rs LSL nn{,Carry} CMP Rd, Rs LSL nn	3/8	R	√	√	√	√
		Rd = - Rs LSL nn Rd &= Rs LSL nn Rd |= Rs LSL nn Rd ^= Rs LSL nn TEST Rd, Rs LSL nn Rd = Rs LSL nn	3/8	R	√	√	−	−

ALU-LSR

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd #=Rs LSR nn{,Carry} CMP Rd, Rs LSR nn TEST Rd, Rs LSR nn	单字	Rd+=Rs LSR n{,Carry} Rd-=Rs LSR nn{,Carry} CMP Rd, Rs LSR nn	3/8	R	√	√	√	√
		Rd = - Rs LSR nn Rd &= Rs LSR nn Rd |= Rs LSR nn Rd ^= Rs LSR nn TEST Rd, Rs LSR nn Rd = Rs LSR nn	3/8	R	√	√	−	−

ALU-MULS

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
MR =[Rd]*[Rs]	单字	MR = [Rd]*[Rs] {,ss} {, n} MR = [Rd]*[Rs], us {, n}	10n+8	[R]	–	–	–	–

ALU-NEG

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd=-X =(~X) + 1	单字	Rd = -IM6	3	IM6	√	√	–	–
	双字	Rd = -IM16	6/8	IM16				
	单字	Rd = -[BP + IM6]	8	[BP+IM6]				
		Rd = -[A6]	6/8	[A6]				
	双字	Rd = -[A16]	9/11	[A16]				
	单字	Rd = -Rs	3/8	R				
		Rd = -{D:}[Rs] Rd = -{D:}[++Rs] Rd = -{D:}[Rs--] Rd = -{D:}[Rs++]	7/9	[R]				

ALU-OR

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd = X | Y	单字	Rd |= IM6 Rd = Rd | IM6	3	IM6	√	√	–	–
	双字	Rd = Rs | IM16	6/8	IM16				
	单字	Rd |= [BP + IM6] Rd = Rd | [BP + IM6]	8	[BP+IM6]				
		Rd |= [A6] Rd = Rd | [A6]	6/8	[A6]				
	双字	Rd = Rs | [A16]	9/11	[A16]				
	单字	Rd |= Rs	3/8	R				
		Rd |= {D:}[Rs] Rd |= {D:}[++Rs] Rd |= {D:}[Rs--] Rd |= {D:}[Rs++]	7/9	[R]				

ALU-ROL

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd#=Rs ROL nn{,Carry} CMP Rd, Rs ROL nn TEST Rd, Rs ROL nn	单字	Rd+=Rs ROL n{,Carry} Rd-=Rs ROL nn{,Carry} CMP Rd, Rs ROL nn	3/8	R	√	√	√	√
		Rd = - Rs ROL nn Rd &= Rs ROL nn Rd |= Rs ROL nn Rd ^= Rs ROL nn TEST Rd, Rs ROL nn Rd = Rs ROL nn	3/8	R	√	√	–	–

ALU-ROR

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd#=Rs ROR nn{,Carry} CMP Rd, Rs ROR nn TEST Rd, Rs ROR nn	单字	Rd+=Rs ROR n{,Carry} Rd-=Rs ROR nn{,Carry} CMP Rd, Rs ROR nn	3/8	R	√	√	√	√
		Rd = - Rs ROR nn Rd &= Rs ROR nn Rd |= Rs ROR nn Rd ^= Rs ROR nn TEST Rd, Rs ROR nn Rd = Rs ROR nn	3/8	R	√	√	–	–

ALU-SUB

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd = X−Y	单字	Rd -= IM6 Rd = Rd - IM6	3	IM6	√	√	√	√
	双字	Rd = Rs - IM16	6/8	IM16				
	单字	Rd -= [BP + IM6] Rd=Rd - [BP + IM6]	8	[BP+IM6]				
		Rd -= [A6] Rd = Rd - [A6]	6/8	[A6]				
	双字	Rd = Rs - [A16]	9/11	[A16]				
	单字	Rd -= Rs	3/8	R				
		Rd -= {D:}[Rs] Rd -= {D:}[++Rs] Rd -= {D:}[Rs--] Rd -= {D:}[Rs++]	7/9	[R]				

ALU-SBC

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd =X−Y−C =X+ (~Y) + C	单字	Rd -= IM6,Carry Rd = Rd - IM6,Carry	3	IM6	√	√	√	√
	双字	Rd = Rs - IM16,Carry	6/8	IM16				
	单字	Rd -= [BP + IM6],Carry Rd=Rd - [BP + IM6],Carry	8	[BP+IM6]				
		Rd -= [A6],Carry Rd = Rd - [A6],Carry	6/8	[A6]				
	双字	Rd = Rs - [A16],Carry	9/11	[A16]				
	单字	Rd -= Rs,Carry	3/8	R				
		Rd -= {D:}[Rs],Carry Rd -= {D:}[++Rs],Carry Rd -= {D:}[Rs--],Carry Rd -= {D:}[Rs++],Carry	7/9	[R]				

ALU-TEST

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
X&Y 只影响 NZSC 标志	单字	TEST Rd IM6	3	IM6	√	√	–	–
	双字	TEST Rd IM16	6/8	IM16				
	单字	TEST Rd [BP + IM6]	8	[BP+IM6]				
		TEST Rd [A6]	6/8	[A6]				
	双字	TEST Rs [A16]	9/11	[A16]				
	单字	TEST Rd Rs	3/8	R				
		TEST Rd {D:}[Rs] TEST Rd {D:}[++Rs] TEST Rd {D:}[Rs--] TEST Rd {D:}[Rs++]	7/9	[R]				

ALU-XOR

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd = X ^ Y	单字	Rd^= IM6 Rd = Rd ^| IM6	3	IM6	√	√	–	–
	双字	Rd = Rs ^ IM16	6/8	IM16				
	单字	Rd ^= [BP + IM6] Rd = Rd ^ [BP + IM6]	8	[BP+IM6]				
		Rd ^= [A6] Rd = Rd ^ [A6]	6/8	[A6]				
	双字	Rd = Rs ^ [A16]	9/11	[A16]				
	单字	Rd ^= Rs	3/8	R				
		Rd ^|= {D:}[Rs] Rd ^= {D:}[++Rs] Rd ^|= {D:}[Rs--] Rd ^= {D:}[Rs++]	7/9	[R]				

Data−Transfer Instruction−LOAD

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
Rd = X	单字	Rd = IM6	3	IM6	√	√	−	−
	双字	Rd = IM16	6/8	IM16				
	单字	Rd = [BP + IM6]	8	[BP+IM6]				
		Rd = [A6]	6/8	[A6]				
	双字	Rd = [A16]	9/11	[A16]				
	单字	Rd = Rs	3/8	R				
		Rd = {D:}[Rs] Rd = {D:}[++Rs] Rd = {D:}[Rs--] Rd = {D:}[Rs++]	7/9	[R]				

Data−Transfer Instruction−POP/PUSH

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
POP Rx, Ry FROM [Rs]	单字	POP Rx,Ry FROM [Rs]	3n+4/3n+6	R	√	√	−	−
PUSH Rx, Ry TO [Rs]	单字	PUSH Rx,Ry TO[Rs]	3n+4	R	−	−	−	−

Data−Transfer Instruction−STORE

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
X = Rd	单字	[BP + Imm6] = Rd	8	[BP+IM6]	−	−	−	−
		[A6] = Rd 6	6/8	[A6]				
	双字	[A16] = Rd	9/11	[A16]				
	单字	Rd = Rs	3/8	R				
		{D:}[Rs] = Rd {D:}[++Rs] = Rd {D:}[Rs--] = Rd {D:}[Rs++] = Rd	7/9	[R]				

Miscellaneous Instructions

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
FIQ ON/ OFF	单字	FIQ ON FIQ OFF	3	−	−	−	−	−
FIR_MOV ON/ OFF	单字	FIR_MOV ON FIR_MOV OFF	3	−	−	−	−	−
IRQ ON/ OF	单字	IRQ ON IRQ OFF	3	−	−	−	−	−
INT FIQ/IRQ INT FIQ, IRQ INT OFF	单字	INT FIQ INT IRQ INT FIQ, IRQ INT OFF	3	−	−	−	−	−
NOP	单字	NOP	5	−	−	−	−	−

Transfer−Control Instructions

指令语法	指令长度	指令子集语法	指令周期	寻址方式	标志位反应			
					N	Z	S	C
BREAK	单字	BREAK	13	[A16]	-	-	-	-
CALL Label	双字	CALL Label	13	[A22]	-	-	-	-
GOTO Label	双字	GOTO Label	12	[A16]	-	-	-	-
Jcond Label Jmp Label	单字	JCC Label 清进位 C JB Label 无符号数小于 JNAE Label 无符号数不大于等于 JCS Label 进位位置 1 JNB Label 无符号数不小于 JAE Label 无符号数大于等于 JSC Label 清符号位 JGE Label 有符号数大于等于 JNL Label 有符号数不小于 JSS Label 符号位置 1 JNGE Label 有符号数不大于等于 JL Label 有符号数小于 JNE Label 不等于 JNZ Label 非零 JZ Label 零 JE Label 相等 JPL Label 为正 JMI Label 为负 JBE Label 无符号数小于等于 JNA Label 无符号数不大于 JNBE Label 无符号数不小于等于 JA Label 无符号数大于 JLE Label 有符号数小于等于 JNG Label 有符号数不大于 JNLE Label 有符号数不小于等于 JG Label 有符号数大于 JVC Label 有符号数无溢出 JVS Label 有符号数溢出 JMP Label 无条件转移	3/5	PC+IM6	−	−	−	−
RETF	单字	RETF	12	[A22]	√	√	√	√
RETI	单字	RETI	12	[A22]	√	√	√	√

附录 C　C_LIB 函数集

表 C.1　字符函数集一览

序号	调用方式	功能
1	int isalnum(int c)	若参数 c 是一个字母或一个数字，则函数返回非零值；否则返回零
2	int isalpha(int c)	若参数 c 为字母表中的字母则函数返回非零值；否则返回零
3	int isascii(int c)	若参数 c 为 ASCII 字符即 c 的低字节在 0～127(0x00～0x7F)之间，则函数返回非零值；否则返回零
4	int iscntrl(int c)	若参数 c 为控制符即 c 为删除 delete 控制符 0x7F 或一般控制符(0x00～0x1F)，则函数返回非零值；否则返回零
5	int isdigit(int c)	若参数 c 为十进制数字(0～9)，则函数返回非零值；否则返回零
6	int islower(int c)	若参数 c 为小写字母(a～z)，则函数返回非零值；否则返回零
7	int isprint(int c)	若参数 c 为可打印字符(0x20～0x7E)，则函数返回非零值；否则返回零
	int isgraph(int c)	与 isprint 函数类似，只是不包含空格符(0x20)
8	int ispunct(int c)	若参数 c 为可打印的标点符号即(isgraph(c) && !isalnum(c))为非零值，则函数返回非零值；否则返回零
9	int isspace(int c)	若参数 c 为空格水平或垂直制表符、回车符、换行符及进纸符等(0x09～0x0D、0x20)，则函数返回非零值；否则返回零
10	int isupper(int c)	若参数 c 为大写字母(A～Z)，则函数返回非零值；否则返回零
11	Intisxdigit(int <[c]>)	若参数 c 为十六进制数字，则函数返回非零值；否则返回零
12	int toascii(int c)	将整型参变量 c 转换成范围为 0～127 的 ASCII 码
13	int tolower(int c)	将整型参变量 c 转换成小写字母值 a～f
	int _tolower(int<[c]>)	与 tolower 函数功能相似，但本函数只能用于已知参数是大写字母(A～Z)的情况，否则函数的返回值不定
14	Int_toupper(int<[c]>)	将整型参变量 c 转换成大写字母值(A～Z)
		与 topper 函数功能相似，但本函数只能用于已知参数是小写字母(a～f)的情况，否则函数的返回值不定

表 C.2 字符串和内存函数集一览

序号	调用方式	功能
1	int bcmp(const char *s1, const char *s2, intn)	比较由参数 s1 和 s2 指向的两数组的前 n 个字符，若 s1 指向的数组大于等于或小于 s2 指向的数组函数，则分别返回一个大于等于或小于 0 的整数
2	void bcopy(const char *in, char *out, int n)	将参数 in 指向的存储区的 n 个字节复制到参数 out 指向的存储区
3	void bzero(char *b, int length)	将指针为参数 b，长度为参数 length 的存储区内数据初始化为零
4	void* charpack(void *dst, char *s)	将由参数 s 所指的以 16 位数表示的字符串压缩转换成由参数 dst 所指的以 8 位数表示的字符串，并返回指针 dst 值。被压缩字符串中的空字符会结束转换
5	void* charunpack(void *dst,void*s)	将由参数 s 所指的以 8 位数表示的字符串转换成由参数 dst 所指的以 16 位数表示的字符串，并返回指针 dst 值。解压缩字符串中的空字符会结束转换
6	char* index(const char *string,int c)	在参数 string 指向的字符串中包含结束符查找已转换成字符的参数 c，并返回查到 c 处的指针。若未查找到 c，则返回空指针
7	void* memccpy(void *out, const void *in, int c, int n)	将参数 in 所指内存内容拷贝到参数 out 所指内存中去，当拷贝了 n 个字节后或者第一次遇到的参数 c 被拷贝后就停止拷贝操作。若拷贝了 c，则返回指向 out 中紧跟 c 后字符的指针，否则返回空指针
8	void* memchr(const void *src, int c, int length)	在由参数 src 所指的存储区中前 length 个字节内搜索参数 c，返回一个指向在 src 中最先遇到 c 的指针；若未找到 c，则返回空指针
9	int memcmp(const void *s1, const void *s2,int n)	比较参数 s1，s2 指向的数组的前 n 个字符，若 s1 指向的数组大于等于或小于由 s2 指向的数组函数，则分别返回一个大于等于或小于零的整数；若两数组比较区域重叠，则为非法
10	void* memcpy(void *out, const void *in,int n)	从参数 in 所指的数组中拷贝 n 个字符到参数 out 所指的数组中去，并返回一个指向 out 的指针；若两数组区域重叠，则函数的行为非法
11	void* memmove(void *dst,const void *src,unsigned length)	从 src 所指的数组中拷贝 length 个字符到 dst 所指的数组中去，并返回一个指向 dst 的指针；数组区域重叠不会影响内容正确地拷贝
12	char *rindex(const char *string, int c)	在参数 string 所指的字符串中(包括结束符) 查找最后一次出现的已转换为字符型的参数 c，并返回查到 c 处的指针；如未找到 c，则返回一个空指针。本函数与函数 strrchr 功能相同
13	void setmem(char *dst, int length, char c)	将参数 dst 所指的数组中 length 个字节的块都设置为字符参数 c
14	char *stpcpy(char *dst, const char *src)	将参数 src 指向的数组中字符串拷贝到参数 dst 指向的数组中，直至遇到结束符为止，并返回指针 dst+strlen(src)之值

续表

序号	调用方式	功能
15	int strcasecmp(const char *a,const char *b)	将参数 a、b 所指的两个字符串进行比较并识别大小写，按字典顺序，若 a 排在 b 后都转换为大写，则函数返回一个正数；若 a 排在 b 前，则函数返回一个负数；若两个字符串匹配，则函数返回零
16	char *strcat(char *dst, const char *src)	把参数 src 指向的字符串(包含结束符)连到参数 dst 指向的字符串尾部，src 第一个字符覆盖 dst 的结束符，并返回合并后字符串指针
17	char * strchr(const char *string, int c)	返回参数 string 指向的字符串中包含结束符首次出现的字符参数 c 的位置，指针若未发现与 c 匹配的字符，则返回空指针
18	int strcmp(const char *a, const char *b)	对参数 a、b 所指的字符串进行比较，按字典顺序，若 a 排在 b 后，则函数返回一个正数；若 a 排在 b 前，则函数返回一个负数；若两个字符串匹配，则函数返回零
19	int strcoll(const char *stra, onst char *strb)	对参数 stra、strb 所指的字符串进行比较，根据 stra 字符串大于小于或等于 strb 字符串函数，将相应返回一个正数、负数或零
20	char *strcpy(char *dst, const char *src)	将参数 src 指向的字符串包含结束符拷贝到参数 dst 指向的数组中去，并返回 dst 的指针；若两数组区域重叠，则函数行为非法
21	int strcspn(const char *s1, const char *s2)	返回参数 s1 所指字符串的初始子串的长度，该子串中的任一字符都不包含于参数 s2 所指的字符串中(结束符除外)
22	int strlen(const char *str)	计算并返回参数 str 所指字符串的长度。方法是计算字符个数直至结束符为止(结束符不计在内)
23	char *strlwr(char *a)	把参数 a 指向的字符串中每个字符都变为小写字母，并返回指针 a
24	int strncasecmp(const char*a, const char *b, intlength)	对参数 a、b 指向的两个字符串中 length 个字符进行比较，并识别其大小写，按字典顺序，若 a 排在 b 后都转换为大写，则函数返回一个正数 若 a 排在 b 前，则函数返回一个负数；若两个字符串匹配，则函数返回零
25	char *strncat(char *dst,const char *src, int length)	将参数 src 指向的字符串(包含结束符)中前 length 个字符连接到参数 dst 指向的字符串之尾部。src 中第一个字符覆盖 dst 的结束符，函数返回 dst 指针值
26	int strncmp(const char *a, const char *b, int length)	比较参数 a、b 指向的两个字符串中前 length 个字符，按字典顺序，若 a 排在 b 后都转换为大写，则函数返回一个正数；若 a 排在 b 前，则函数返回一个负数；若两个字符串匹配，则函数返回零
27	char *strncpy(char *dst, const char *src, intlength)	将参数 src 指向的字符串(包括结束符)中前 length 个字符拷贝到参数 dst 指向的数组中去，若 src 数组中少于 length 个字符，则会在 dst 数组中添加空字符直至凑够 length 个字符函数，并返回 dst 指针值

续表

序号	调用方式	功能
28	char *strnset(char *dst, char c,unsigned n)	将参数 dst 指向的字符串中前 n 个字符设置为字符参数 c 的值
29	char *strpbrk(const char *s1, const char *s2)	在参数 s1 指向的字符串中查找与参数 s2 指向的字符串中任何一个字符相匹配的第一个字符(空字符不包含在内)，并返回其位置指针。若没有匹配字符，则返回空指针
30	char * strrchr(const char *string, int c)	在参数 string 指向的字符串中(包含结束符)查找最后一次出现字符参数 c，并返回其位置指针；若未找到 c，则返回空指针
31	char *strrev(char *s)	将参数 s 指向的字符串中所有字符顺序都颠倒过来(结束符除外)，并返回指向颠倒顺序后的字符串指针
32	char *strset(char *s, char c)	将参数 s 所指字符串中所有字符都设置成字符参数 c，并返回指针 s
33	int strspn(const char *s1, const char *s2)	在参数 s1 所指字符串中查找第一个不属于参数 s2 所指字符串中字符的位置(结束符除外)，计算并返回从起始到此位置的长度值
34	char *strstr(const char *s1, const char *s2)	在参数 s1 所指字符串中查找第一次遇到参数 s2 所指字符串(其结束符除外)，并返回其位置指针；若未找到相匹配的字符串，则返回空指针；若 s2 指向的字符串长度为零，则返回指针 s1
35	char *strtok(char *source, const char, *delimiters)	返回参数 source 所指字符串中指向下一个由参数 delimiters 指定的字符或字符串的分隔符的指针，若无分隔符，则返回一个空指针；函数实际上修改了由 source 指向的字符串。每找到一个分隔符后，一空字符就被放到分隔符处，函数用此方法连续查遍该字符串
36	char *strupr(char *a)	将参数 a 指向的字符串中所有字符都变为大写字母，并返回指针 a
37	int strxfrm(char *s1,const char *s2, int n)	对参数 s2 指向的字符串进行转换后前 n 个字符(包含结束符)置入参数 s1 指向的数组中；若 n 为 0，则 s1 为空指针，拷贝若在重叠的区域内，则会使函数结果不定。字符串转换如下：若用于两个转换字符串的函数 strcmp 与用于两个相同的原字符串的函数 strcoll 的结果相对应，函数会分别返回一个正数、零或负数
38	void swab(char *from, char *to, unsigned n)	从参数 from 指向的字符串中拷贝 n 个字符到参数 to 指向的字符串中，并交换相邻的偶、奇数字节

注：INF 值为单精度型数的下限值；NAN 值为超出参变量定义域范围之值。

表 C.3　数学函数集一览

序号	调用方式	功能
1	float acosf(float x)	返回参变量 x 的反余弦值，以弧度表示 x 的定义域为[−1，1]
2	float acoshf(float x)	返回参变量 x 的反双曲余弦值
3	float asinf(float x)	返回参变量 x 的反正弦值，以弧度表示 x 的定义域为[−1，1]
4	float asinhf(float x)	返回参变量 x 的反双曲正弦值
5	float atanf(float x)	返回参变量 x 的反正切值，以弧度表示
6	float atan2f(float y float x)	返回参变量 y/x 的反正切值
7	float atanhf(float <[x]>)	返回参变量 x 的反双曲正切值
8	float cabs(struct complex z)	返回复数参变量 z 的绝对值
9	float cbrtf(float x)	返回参变量 x 的立方根值
10	float ceilf(float x)	返回不小于参变量 x 的最小整数
11	float copysignf (float x, float y)	构造一个数，其值为参量 x 的绝对值，其符号为参量 y 的符号
12	float cosf(float x)	返回参变量 x 的余弦值，以弧度表示
13	float coshf(float x)	返回参变量 x 的双曲余弦值
14	float dremf (float x, float y)	返回参变量运算 x/y 的余数值
15	float erff(float x)	估算落在参数 x 标准平均误差范围内的概率(假设为正态分布统计值)
	float erfcf(float x)	直接计算函数 erff 的互补概率(1-erff(x))， 用此函数可以避免通过计算 1-erff(x)造成的精度损失
16	float expf(float x)	计算并返回参变量 x 的指数值，即 e^x(e 约为 2.718 28)
17	float expm1f(float x)	计算并返回($e^x - 1$)， 参变量 x 值即使很小亦能保证精度。但若使用 expf 函数计算($e^x - 1$)，则会丢失有效位
18	float fabsf(float x)	返回参变量 x 的绝对值
19	int finitef(float x)	若参变量 x 为有限值，返回非零值；否则返回零
20	float floorf(float x)	返回不大于参变量 x 的最大整数
21	float fmodf(float x, float y)	返回浮点数单精度型参变量运算 x/y 的余数
22	float frexpf(float val, int *exp)	把参量 val 分解成一个 0.5～1 范围内的尾数和一个整型指数，即 val=尾数$*2^{exp}$。其中尾数由函数返回指数存储在参量 exp 中
23	float gammaf(float x)	计算参变量 x 的 gamma 函数的自然对数
	float lgammaf(float x)	是 gammaf 函数的别名
	float gammaf_r(float x, int *signgamp)	计算参变量 x 的 gamma 函数的自然对数，并将 gamma 函数的符号存储在参量 signgamp 中
	float lgammaf_r(float x, int *signgamp)	同上
24	float hypotf(float x, float y)	返回据给定直角三角形的两个直角边算出其斜边的长度
25	int ilogbf(float val)	所有非零数都可表示为 $m*2^p$。若参变量 val 的定义域在 0～INF[1]范围内，则函数返回 p；若 val 定义为零，则返回 INT_MAX；若定义为 INF 或超出定义域，则函数返回 INT_MAX
26	float infinityf(void)	返回 INF 值

续表

序号	调用方式	功能
27	int isinff(float x)	若参变量 x 为 INF 值，返回非零值；否则返回零
28	int isnanf(float arg)	若参数变量为 NAN[2]值，返回非零值；否则返回零
29	float j0f(float x)	求解微分方程的第一类贝塞尔函数的零阶特例函数
	float j1f(float x)	求解微分方程的第一类贝塞尔函数的一阶特例函数
	float jnf(int n, float x)	求解微分方程的第一类 n 阶贝塞尔函数
	float y0f(float x)	求解微分方程的第二类贝塞尔函数的零阶特例函数
	float y1f(float x)	求解微分方程的第二类贝塞尔函数的一阶特例函数
	float ynf(int n, float x)	求解微分方程的第二类 n 阶贝塞尔函数
30	float ldexpf(float val, int exp)	计算并返回 val*2^{exp} 的值。若计算发生溢出，则返回 HUGE_VAL
31	float logf(float x)	返回参变量 x 的自然对数值，x 的定义域为(0，INF)
32	float log10f(float x)	返回参变量 x 以 10 为底的对数值，x 的定义域为(0，INF)
33	float log1pf(float x)	返回(1+x) 的自然对数值，参量 x 很小时用此函数运算精度高
34	int logbf (float val)	同 ilogbf()函数
35	int matherr(struct exception *err)	函数用于常见的数学错误处理，用类型为 exception 的参数调用,其结构为：struct exception { int type; char *name; double arg1, arg2, retval; int err;}
36	float modff(float val, float *ipart)	将参变量 val 分解成整数部分和小数部分，其中小数由函数返回，整数则存储在参变量 ipart 中
37	float nanf(void)	返回 NAN 值
38	float nextafterf(float val, float dir)	返回参量 val IEEE 格式向参量 dir 方向变化的下一个数值
39	float polyf(float x, int n,float c[ ])	返回计算 x^n 之系数为 c[0]～c[n]的运算多项式的值
40	float powf(float x, float y)	返回计算以参变量 x 为底的 y 次幂，即 x^y 之值
41	float remainderf(float x, float y)	同 dremf()函数
42	float rintf(float x)	返回将参变量 x 经四舍五入处理后的整数值
43	float scalbf(float x, float n)	返回 x*2^n 之运算值，n 为单精度型数
44	float scalbnf(float x, int n)	返回 x*2^n 之运算值，n 为整型数
45	float significandf(float x)	返回函数 scalbf(x(float) −ilogb(x))调用的结果值
46	float sinf(float x)	返回参变量 x 的正弦值，x 以弧度表示
47	float sinhf(float x)	返回参变量 x 的双曲正弦值
48	float sqrtf(float x)	返回参变量 x 的平方根值，x 的定义域为(0，INF)
49	float tanf(float x)	返回参变量 x 的正切值，x 以弧度表示
50	float tanhf(float x)	返回参变量 x 的双曲正切值，x 以弧度表示

注: [1] INF 值即为单精度型数的下限值；[2] NAN 值即为超出参变量定义域范围之值。

表 C.4　标准库函数集一览

序号	调 用 方 式	功　能
1	void abort(void)；	程序检测到一种无法处理的异常情况时终止程序的运行
2	int abs(int x);	返回计算整型参数 x 的绝对值
3	Void assert(int [expression]>);	用于在程序中嵌入调试诊断信息的宏，若程序正常运行，则表达式参数[expression] 为非零值；若程序运行出现异常，则[expression]为零值。可调用 abort()函数以终止程序的运行
4	float atoff(char *s);	返回将参数 s 所指字符串的起始部分转换的单精度数，若转换未成功(包括溢出)，则返回“ 0.0 ”。若转换值超出了其可代表数的范围，则返回“ -HUGE_VAL ”或“ HUGE_VAL ”
5	int atoi(char *s);	返回将参数 s 指的字符串转换成的整型数，转换未成则返回零
6	long atol(const char *s);	返回将参数 s 指的字符串转换成的长整型数，转换未成则返回零
7	void *bsearch(const void *key, const void *base, size_t nmemb, size_t size,int (*compar)(const void *, const void *))	在参数 base 指的排序数组中执行二元搜索，并返回指向与 key 所指关键字相匹配的第一个元素的指针。若数组未含关键字，则返回空指针。数组中元素数目由参数 nmemb 指定，且每个元素的大小(以字节表示)由参数 size 给定，数据类型 size_t 在 stdlib.h 中被定义为 unsigned int
8	void *calloc(size_t <[n]>, size_t<[s]>);	返回为具有 n 个长度为 s 的数据的数组分配内存区域第一字节的指针；若无足够的内存可分配，则返回空指针
9	div_t div(int n, int d)	将两整型数相除的商和余数返回在结构型参数 div_t 中

10	char*ecvtf(float val,int chars, int *decpt, int *sgn);
	char*fcvtf(float val,intdecimals,int*decpt, int*sgn);
	char *gcvtf(float val, int precision,char *buf);
		将单精度浮点型参数 val 转换成长度为参数 chars 的字符串，并返回指向该字符串的指针。参数 decpt 指向小数点的位置，而参数 sgn 则指向符号变量。fcvtf()函数的参数 decimals 与 chars 不同，它指定的是小数点后的数值转换成字符串的长度
		将单精度浮点型参数 val 转换成长度为参数 precision 的字符串，参数 buf 作为指向该字符串数组的指针而被返回

11	void exit(int <[code]>) ;	使得程序立即正常终止运行。状态参数<[code]>被传递到调用过程，若其为零，表明程序正常终止；若其为非零，表明存在执行错误
12	long labs(long x);	返回计算长整型参数 x 的绝对值
13	ldiv_t ldiv(long n, long d);	两长整型参数 n,d 相除,商和余数返回在结构型参数 div_t 中
14	void *malloc(size_t <[nbytes]>); void free(void *<[aptr]>);	返回申请分配大小(以字节表示)为参数<[nbytes]>的内存区域首字节的指针；若申请未成功，则返回空指针 释放由参数<[aptr]>指向的内存区域，并将它返回给堆

续表

序号	调用方式	功能
15	void qsort(void *base, size_t nmemb,size_t size, int (*compar)(const void *, const void *));	对参数<[base]>指向的数组中<[nmemb]>个元素进行分类，且每个元素的大小由参数<[size]>定义。参数 compar 用于指向一个比较函数，其中每一参数都指向<[base]>数组中某一元素；根据第一参数大于、等于或小于第二参数，比较函数分别返回一个正数、零或负数
16	int rand(void);	返回伪随机数序列中在 0～RAND_MAX(包括 RAND_MAX)之间的一个整数
17	void *realloc(void <[aptr]>, size_t <[nbytes]>);	将参数<[aptr]>指向的已分配的内存大小变成由参数<[nbytes]>确定新的大小的内存块，并返回指向新块首字节的指针；若堆中分配不出<[nbytes]>个字节，则函数返回空指针
18	void srand(unsigned int seed);	建立由 rand()函数所产生的伪随机数序列中数值的起始点。它允许多个程序用不同的伪随机数序列运行
19	char *strdup(_CONST char *str);	按参数 str 所指字符串的长度开出内存区，且将字符串内容拷贝到该存储区域并返回指向该区域首字节的指针
20	float strtodf(const char *str, char **endptr);	返回将参数 str 所指的以数值形式表示的字符串转换成一个单精度型数，参数*endptr 指向转换字符串的结束符 null；若转换未成功或转换值溢出，则函数返回零；若转换值超出其所能代表数的范围，则函数返回 HUGE_VAL
21	long strtol(const char *s, char **ptr,int base);	返回将参数 s 所指的以数值形式表示的字符串转换成的一个长整型数，数值的进制由参数 base 确定。若转换未成功，则函数返回零；若转换值上下溢出，则函数分别返回 LONG_MAX 及 LONG_MIN
22	unsigned long strtoul(const char *s, char **ptr, int base);	功能与 strtol()函数类似，不同之处在于本函数将字符串转换成一无符号长整型数
23	int system(char *s);	从一正在执行的 C 程序中执行系统的命令，参数 s 指向该命令字符串。若函数调用成功，则返回零；否则返回非零值

注：表中带阴影部分为虚函数

表 C.5 I/O 函数集一览

调用方式	功能
int printf(const char *format,...)	按参数 format 指定的格式，将其后参量表中列出的参数写到流文件[注]中

注：程序中若需调用 printf 函数，应使 μ'nSP™ IDE 运行在 Simulator 方式下并进行如下操作：选择 Project 菜单的 Setting 选项，进入 Device 属性页，在 Device Set 中会有一个缺省的 I/O 口地址 0x7016，选择 Output 单选按钮后，Sound 复选框会被激活，但不要选择 Sound；然后在 Output File 文本框中输入流文件名即可(详见第 9 章中项目的设置)。

表 C.6　针对错误号 errnum 的错误信息

errnum	错误信息	含义
E2BIG	Arg list too long	参数表太长
EACCES	Permission denied	不允许
EADDRINUSE	Address already in use	地址已被占用
EADV	Advertise error	警告错误
EAFNOSUPPORT	Address family not supported by protocol family	地址体系超出规定范围
EAGAIN	No more processes	没有更多的步骤
EALREADY	Socket already connected	接口已经连接
EBADF	Bad file number	错误的文件号
EBADMSG	Bad message	错误的信息
EBUSY	Device or resource busy	设备或资源正被使用
ECHILD	No children	无子系统
ECOMM	Communication error	通讯错误
ECONNABORTED	Software caused connection abort	软件错误引起连接失败
ECONNREFUSED	Connection refused	连接未成功
EDEADLK	Deadlock	死锁
EDESTADDRREQ	Destination address required	未给目标地址
EEXIST	File exists	文件已存在
EDOM	Math argument	计算函数参数的域错误
EFAULT	Bad address	错误的地址
EFBIG	File too large	文件太长
EHOSTDOWN	Host is down	主机故障
EHOSTUNREACH	Host is unreachable	主机功能达不到
EIDRM	Identifier removed	标识符丢失
EINPROGRESS	Connection already in progress	连接已在处理中
EINTR	Interrupted system call	中断系统调用
EINVAL	Invalid argument	非法参数
EIO	I/O error	输入/输出错误
EISCONN	Socket is already connected	插口已连接
EISDIR	Is a directory	路径错误
ELIBACC	Cannot access a needed shared library	要求共享的库不能被访问
ELIBBAD	Accessing a corrupted shared library	要访问的共享库已被破坏
ELIBEXEC	Cannot exec a shared library directly	不可直接执行一个共享库
ELIBMAX	Attempting to link in more shared libraries than system limit	链接的共享库已超出限度
ELIBSCN	<<.lib>> section in a.out corrupted	在*.out 文件中的<.lib>损坏
EMFILE	Too many open files	要打开的文件太多
EMLINK	Too many links	要链接的模块太多
EMSGSIZE	Message too long	信息太长
EMULTIHOP	Multihop attempted	非法的多重接收
ENAMETOOLONG	File or path name too long	文件或路径名太长
ENETDOWN	Network interface not configured	网络接口未配置

续表

errnum	错误信息	含义
ENETUNREACH	Network is unreachable	网络功能达不到
ENFILE	Too many open files in system	系统中打开的文件太多
ENODEV	No such device	所需要的设备不存在
ENOENT	No such file or directory	输入的文件或路径不存在
ENOEXEC	Exec format error	执行的格式错误
ENOLCK	No lock	未锁
ENO LINK	Virtual circuit is gone	实际电路不存在
ENOMEM	Not enough space	无足够的存储空间
ENOMSG	No message of desired type	并非所需类型的信息
ENONET	Machine is not on the network	所要搜索的机器未上网
ENOPKG	No package	未经压缩
ENOPROTOOPT	Protocol not available	协议不可用
ENOSPC	No space left on device	设备未留有足够的空间
ENOSR	No stream resources	非流资源
ENOSTR	Not a stream	不是一个流
ENOSYS	Function not implemented	函数未被执行
ENOTBLK	Block device required	未接通所需的模块设备
ENOTCONN	Socket is not connected	插口未连接
ENOTDIR	Not a directory	并非路径
ENOTEMPTY	Directory not empty	路径名仍在占用
ENOTSOCK	Socket operation on non-socket	插口连接未成
ENOTSUP	Not supported	不支持
ENOTTY	Not a character device	非字符设备
ENXIO	No such device or address	无此设备或地址
EPERM	Not owner	并非物主
EPIPE	Broken pipe	破坏的流通管道
EPROTO	Protocol error	协议错误
EPROTOTYPE	Protocol wrong type for socket	协议接口警告
EPROTONOSUPPORT	Unknown protocol	未知协议
ERANGE	Result too large	运算结果超出范围
EREMOTE	Resource is remote	资源位置太远
EROFS	Read-only file system	只读文件系统不可写入
ESHUTDOWN	Can't send after socket shutdown	接口关闭后不能传输信息
ESOCKTNOSUPPORT	Socket type not supported	不支持此类接口类型
ESPIPE	Illegal seek	非法搜寻
ESRCH	No such process	无此过程
ESRMNT	Srmount error	高级装配错误
ETIME	Stream ioctl timeout I/O	流控制超时
ETIMEDOUT	Connection timed out	连接时间太长
ETXTBSY	Text file busy	文本文件在处理中
EXDEV	Cross-device link	设备交叉链接

附录 D　常用伪指令表

序号	语法格式	类别	功能
1	.CODE	定义类	切换定义预定义段 CODE
2	.DATA		切换定义预定义段 DATA
3	Label:.DD[value][,value][,…]	存储类	以 32 位长整型数据的形式存储双字常量
4	Lable:.DOUBLE value[,value][,…]		以双精度浮点型、实数数据的形式存储实数常量
5	.DEFINE variable[value][,…]	定义类	为变量赋值(定义常量符号)
6	[lable:].DW number DUP(vale) [lable:].FLOAT number DUP(vale) [lable:].DD number DUP(vale) [lable:].DOUBLE number DUP(vale)	存储定义类	与 DW、FLOAT、DD、DOUBLE 存储类伪指令组合在一起，用与存放若干个具有相同数值的常量；或者申请若干个备用的整型、单精度浮点型、长整型以及双精度浮点数据单元
7	[lable:].DW[value][,value][,…]	存储类	以 16 位整型数据的形式来存储常量
8	.ELSE	条件类	引出 IF 伪指令设置条件汇编结果为假时的汇编指令
9	.END	存储类	用来结束程序文件或结束包含文件
10	.ENDIF	条件类	用来结束条件汇编组合定义
11	.ENDM	定义类	结束宏定义
12	.ENDP		结束程序定义
13	.ENDS		结束结构定义
14	Lable:.EQU value		为标号赋值
15	.EXTERNAL lable[lable][,…]		在某个文件中声明已在其文件里定义过的标号
16	Lable:FLOAT value[,lable][,…]	存储类	以单精度浮点型、实数数据单元来存储实数常量
17	.IF value	条件类	引出在条件汇编结果为真时的汇编指令
18	.IFDEF variable		引出若某变量已被赋过值时的汇编指令
19	.IFMA argument#		引出若指定的宏参量存在时的汇编指令
20	.IFNDEF variable		引出某变量还未被赋值时的汇编指令
21	.INCLUDE filename	汇　编方式类	在汇编文件中包含某个文件
22	.IRAM	定义类	切换文件与定义段 IRAM
23	.ISRAM		切换文件与定义段 ISRAM
24	.MACEXIT		从宏定义中退出
25	Lable:.MACRO arge		起始宏定义
26	.ORAM		切换定义与定义段 ORAM
27	.OSRAM	条件类	切换定义与定义段 OSRAM
28	Lable:.PROC		起始程序定义
29	.PUBLIC　lable[,lable][,…]		声明将引用在其他文件中的全局标号
30	.RAM	定义类	切换文件与定义段 RAM
31	Lable:SECTION .attribute	汇　编方式类	建立用户定义段
32	.SRAM	定义类	切换文件与定义段 SRAM
33	Lable:STRUCT		起始结构定义
34	.TEXT		切换文件与定义段 TEXT
35	.VAR variable[=value]		定义变量并为变量赋初值
36	Lable:VDEF value		为标号赋值

附录 E　SPCE061A 编译相关错误信息

汇编器 Xasm16 **在对** SPCE061A **的指令进行汇编过程中可能会出现的错误信息如下**:

A0000: Syntax error

语法错误，当某一指令或表达式不符合 Xasm16 **规定的格式时会出现此信息。**

A0001: ‘…’ already defined

某符号已被定义过，不能对其重复定义。

A0002: Bad use of local symbol …

局部符号不能被用作宏名、程序名、段名、结构名、结构变量名以及常量名。

A0003: Bad use of keyword ‘…’

指令或伪指令中的关键字不能被用作符号。

A0004: … already defined

某局部符号与前面已定义过的符号同名，符号不允许被重复定义。

A0005: Local symbol ‘…’ can not be declared as external

局部符号只能用在局部区域不能被声明为外部符号。

A0006: ‘…’ Illegal forward reference or symbol

变量在引用前须经定义，不可超前引用。

A0007: ‘…’ undefined

引用的符号未经定义。

A0008: Local symbol ‘…’ can't be declared as public

局部符号只能用在局部区域不能被声明为全局符号。

A0009: ‘…’ can't be declared as public—wrong type

某符号已被定义成常量，故其不能再被声明为全局变量符号。

A0010: ‘…’ can't be declared as public because it's defined with VAR

某符号已用伪指令 VAR **被定义为常量，故其不能再被声明为全局符号。**

A0011: ‘…’ should represent a number

宏调用中引用的参量应为常量。

A0020: File ‘…’ can not be opened

指定的文件访问不到或不能被打开，可能是文件名输入有误。

A0021: Include inside a macro

在宏定义中含有 Include **的操作。**

A0030: Bad use of macro name ‘…’

使用的宏名不符合 Xasm16 **规定的使用方式。**

A0031: Can't define a macro inside another macro

在宏定义里不允许嵌套定义另一个宏。

A0032: IFMA used outside macro

伪指令 IFMA **须与伪指令** MACRO、ENDM **一起使用，不允许将其用在宏定义外。**

A0033: Can't define a macro inside the definition of a structure

在一个结构定义里不允许定义宏。

A0034: Bad declaration of dummy parameters

宏定义中声明的伪参量不符合 Xasm16 **规定的格式。**

A0035: Bad representation of real parameters

实参的表述与声明的形参不匹配。

A0036: ENDM expected before end of file

源文件中伪指令 ENDM **与** MACRO **的数目不匹配。**

A0050: Bad use of section name '…'

使用的段名不符合 Xasm16 **规定的使用方式。**

A0051: Can't define more than '…' sections

Xasm16 **规定段数最多为** 4096**，使用时不可超过此限制。**

A0052: Instruction or data emitted with no section

所有指令和数据都必须定位在某个段内。

A0053: Symbol defined with no section

标号应定义在段内。

A0060: Number overflow

使用的数字值已超出其类型限制。

A0061: Can't add two address expressions

Xasm16 **中不允许将两个地址表达式相加。**

A0062: Can't subtract two address expressions that belong to different sections

Xasm16 **中仅允许将两个属于相同段的地址表达式相减，而属于不同段的地址表达式则被禁之。**

A0063: Can't subtract a number expression with an address expression

Xasm16 **中不允许将一个数字表达式与一个地址表达式相减。**

A0064: Operator ('!', '%', '&', '*', '-', '/', '^', '|', '~', '<<', '>>'), number expression expected

以上操作符只能用在数字表达式中。

A0065: Divided by zero

除法中除数不可为零。

A0066: Bad use of external symbol

外部变量使用有误。

A0067: Operator ('%', '&', '^', '|', '~', '<<', '>>'), illegal operand type of float

以上操作符不可直接用在单精度浮点型操作数中。

A0080: Illegal conditional assembly

条件编译的格式有误。

A0090: Missing field name in structure definition

结构定义中未定义其域结构体成员名。

A0091: Field name '…' already defined

结构定义中的某域名已被定义，不可重复定义。

A0092: '…' not a valid field name

结构定义中引用的域名未经定义。

A0093: '…' is not a structure variable

引用的符号不是一个结构变量。

A0094: Can't define a structure variable in another structure definition

Xasm16 **中不允许将某结构变量定义在另一个结构定义中。**

A0095: Bad use of structure name '…'

使用的结构名不符合 Xasm16 **规定的使用方式。**

A0096: Can't nest a structure definition inside another structure definition

Xasm16 **中不允许把一个结构嵌套定义在另一个结构定义中。**

A0097: Can't emit instruction into a structure definition

结构定义中不可传递指令。

A0098: Define a field but not yet allocate memory for it

结构定义中声明的某成员变量未指明数据类型。

A0099: ENDS expected before end of file

源文件中伪指令 MACRO **和** ENDM **的数目不均，需用一个** ENDS **伪指令来均衡。**

A0100: Field '…' of '…' can't be re-initialized

结构体中那些用逗号(，)预留存储空间的域不能被初始化。

A0101: Field ... of ... is re-initialized unsuccessfully

结构体中那些用逗号(，)预留存储空间的域不能通过初始值列表而被初始化。

A0102: Field '…' of '…' can't be re-initialized with a string

结构的某个域不可用一个字符串被重新初始化。

A0103: The string is too long, and the field '…' of '…' is re-initialized unsuccessfully

引用在初始值列表中的初始值数目与结构定义中的域(成员变量)数不符。

A0104: Can’t define or switch a section inside the definition of a structure
结构定义中不允许定义或切换定义某段。

A0105: Include inside a structure definition
结构定义里不可包括文件。

A0107: Can’t use VAR to allocate memory in a structure definition
结构定义中不可使用伪指令 VAR **来分配存储单元。**

A0110: Can’t define a string with FLOAT
伪指令 FLOAT **不可用来定义字符串。**

A0111: Left operand of DUP should not be negative
伪指令 DUP **左边的参数值表示的是存储单元重复的数目，不能是负数。**

A0112: ‘…’ operand type doesn't match the directive of storage allocation
操作数太大已超出存储类伪指令所能表达的数的范围。

A0113: DB can't be used
Xasm16 **中禁用伪指令** DB **来声明操作数类型，因为访问的数据里无** 8 **位字节型数。**

A0114: Bad use of Imm6
6 **位立即数修饰符** Imm6 **不能用来修饰某一标号或外部符号。**

A0115: Bad use of A6
6 **位地址修饰符** A6 **不能用来修饰数字表达式。**

A0116: Bad use of OFFSET
地址偏移量修饰符 OFFSET **不能用来修饰数字表达式。**

A0117: Bad use of SEG
段修饰符 SEG **不能用来修饰数字表达式。**

A0120: Can't nest a procedure definition in a structure definition
Xasm16 **规定在结构定义里不可嵌套定义一段程序。**

A0121: Expect to switch back to the same section before ENDP
某个程序应定义在相同的一个段内。

A0130: SEG can't modify number expression
修饰符 SEG **不能用来修饰数字表达式。**

A0131: OFFSET can't modify number expression
地址偏移量修饰符 OFFSET **不能用来修饰数字表达式。**

A0132: A6 can't modify number expression
6 **位地址修饰符** A6 **不能用来修饰数字表达式。**

A0140: Address expected
函数调用指令须是调用某一函数名，该函数名代表的应是函数的起始地址。

A0141: Invalid loop counter, integer 1 to 16 expected

寄存器乘法求和指令中的循环计数器的值须是 1～16 范围内的整数。

A0142: BP or R5 register expected

在存储器间接寻址(即变址寻址)指令中，基址指针须是 BP 或 R5。

A0143: Source register and destination register should be the same

源寄存器应当与目标寄存器相同。

A0144: Invalid BP offset, integer 0 to 63 expected

在存储器间接寻址(即变址寻址)指令中，有效地址偏移量应是 0～63 范围内的整数。

A0145: Can't jump to external label

Xasm16 **规定短跳转指令不可跳转到外部标号上。**

A0146: Can't jump to other section

Xasm16 **规定短跳转指令不可跳转到处于其它段的标号上。**

A0147: Jump too far

短跳转指令的跳转限制在 PC±63 范围内。

A0148: Float unexpected

浮点数不可用作指令中的操作数。

A0149: Can't use Imm6 mode

指令中 6 位立即数寻址方式使用不当。

A0150: Can't pop into R0 or SP

在 Xasm16 中不可把栈中数据弹入 R0 或 SP 寄存器中。

A0151: Shift counter should be non-negative integer 1 to 4

寄存器移位指令的移位计数器值应是 1～4 范围内的整数。

A0152: Can't use A6 mode

修饰符 A6 寻址方式下的操作数不可超出 0～63 的整数范围。

A0153: Source registers can't be same as destination register

在寄存器乘法指令(Mul)中，源寄存器不可与目标寄存器相同，且源寄存器不能是 R3 或 R4。

A0154: Address expected

Goto **指令后应为一个地址参量。**

A0155: Source registers can't be same as destination register

寄存器乘法求和指令(Muls)中的源寄存器与目标寄存器不可相同。

A0156: Register PC cannot been used in this addressing mode

当目标/源寄存器为 PC 时，会出现此错误信息。

A0157: Register SR can't been used in this addressing mode
当目标/源寄存器为 SR **时，会出现此错误信息。**

A0158: Register SP can't been used in this addressing mode
当目标/源寄存器为 SP **时，会出现此错误信息。**

A0159: Invalid bit operation offset, integer 0 to 15 expected
在位操作指令中，偏移量应当在整数 0～15 **范围内。**

A0160: Source register R4 or R3 was expected in 32-bit shift operation
在 32 **位移位操作指令中，只有** R4 **或** R3 **能被使用。**

A0161: Destination register R2 was expected in exp instruction
在扩展操作指令中，只有 R2 **可以是目标寄存器。**

A0162: Source register R4 was expected in exp instruction
在扩展操作指令中只有 R4 **可以是源寄存器。**

链接器 Xlink16 **库管理器** Xlib16 **在运行过程中可能会出现的错误信息如下：**

L0000: MFC initialization failed
Xlink16 **不能运行在当前操作平台上，本错误通常发生在** MFC **库初始化过程中。**

L0010: Cannot Open the File ...
指定的文件访问不到或不能被打开，可能是文件名输入有误。

L0011: Link File ... is Empty
链接文件中未含有任何关于重新定位的信息，可能是文件内容有误。

L0012: Cannot Open the File ..., Return from Make the Exe
当需生成可执行文件中相应的代码时，该文件不能被打开。

L0013: Cannot Open the File ..., Return from Make the Symbol File
当需生成符号表文件时，相关的可执行文件不能被打开以写入相应的代码。

L0014: File destroyed: ...
文件被毁坏。

L0020: Can't locate section automatically, please manually locate it
某段容量太大，已不能由 Xlink16 **自动定位，而只能由使用者手动定位。**

L0023: Can't locate ... section at ram address 0-63
段中变量数目太多，使其不能被定位在 0～63 **的** RAM **里。**

L0031: The ram variables are too much to be located
RAM **中的变量数目太多，以致其地址不够分配。**

L0032: UnKnown Linker Option...
输入的链接任选项有误，Xlink16 **无法识别。**

L0040: Can't find this body information in body file

在 body.dat 文件里找不到任何关于芯片型号的信息。

L0041: No interrupt vector information in body file

为使中断向量被正确定位，body.dat 文件里应含有中断向量的地址信息。

L0042: Page 0 hasn't enough space for init table

零页 ROM 中已没有足够的空间为程序代码定位。

L0043: … function hasn't been defined.

中断服务子程序尚未被定义。

L0050: No object filename

ARY **文件或其它链接文件中未输入要链接的目标文件名。**

L0051: Unwanted aqddress ...

声明的某段地址尚未引用在任何文件里。

L0052: Illegal Address ...

链接文件中输入的段地址不符合 Xlink16 规定的格式。

L0053: Syntax error

ARY **文件或其它链接文件的输入有语法错误。**

L0054: Not point out object name for defined section

为某段重新命名时，未指明段所在的目标文件名。

L0055: No redefined section name

为某段重新命名时，未指定新段名。

L0056: No address after the word "At"

为某段定位时，关键字“ At ”后应有一地址参量值。

L0057: No section name after the word "After" in link file

为某段定位时，关键字“ After ”后应有被定位段的段名。

L0058: No address after the word "LinkAt"

为某段定位时关键字“ LinkAt ”后应有一个地址参量值。

L0059: No section name after the word "Linkafter"

为某段定位时关键字“ Linkafter ”后应有被定位段的段名。

L0060: The section … has not been located

在增强数据文件链接方式下，某个未被定位的段将会被定位到 0x00 地址上。

L0061: Define two times ...

某全局符号在不同的目标文件或库文件中被重复定义了两次。

L0062: The section … has not been defined at any obj file

引用的段尚未在任何目标文件中定义。

L0063: Can't locate … section at addr ...

本地址已被其它段占用或已无合适的块单元可用来分配占用。

L0064: The addr space of section … and … is overlapped

两个段产生了地址冲突。

L0065: Text section is too big to fill in first bank

零页中已无合适的块单元供 Text **段分配占用。**

L0066: This Address … has be used by other section

本地址已被其它段占用。

L0067: No address after the character **“** with **”**

为某段定义时关键词“ with **”后应有一个地址参量值。**

L0068: Illegal alignment … the value must be positive

参量值必须为正。

L0069: The section … has an ADDR directive

因为该段已使用 ADDR **指定地址，将无法在** Link **中再指定地址。**

L0070: The section … has not been located before locate the section … after it

增强数据文件链接方式下，用来定位其它段的某段应首先被定位。

L0071: The section … has not been located before locate the section … linkafter it

增强数据文件链接方式下，用来定位其它段的某段须经定位。

L0080: The external symbol … has not a public definition

引用的某局部符号是定义在其它文件里的一个外部符号，而此符号在其被定义的文件里未被声明为全局符号。

L0100: … end address is little than start address in BODY file

Body.dat **文件中的** RAM、ROM **或** I/O **口的结束地址小于其起始地址。**

L0111: Obj type is not Sunplus. The file may be corrupted

指定的文件不是凌阳格式的目标文件或目标文件可能被毁坏。

L0112: Obj file has been destroyed. Return from make exe file

Xlink16 **检测到目标文件已被毁坏，故不能生成所需的可执行文件。**

L0113: This project need vector table/init table for IRAM/ISRAM section

因为在这个项目中有使用 IRAM/ISRAM **段，使用者是不可以用“** -novec **”参数给** Xlink16**。**

L0115: Section ... is older obj file format. Please update resource compiler

Section ...**是旧的目标文件格式，请更新资源编译器。**

L0116: Section … in … cross different bank

Section … in …**跨越了不同的页单元。**

L0120: The Library File ... is not SunplusLib. The file may be corrupted

指定的文件不是凌阳格式的库文件，或库文件可能已被毁坏。

L0500: MFC initialization failedXlib16

不能运行在当前操作平台上。当 MFC **库被初始化时，会出现此类错误。**

L0510: The second argument must be a lib file

第二个参量须是凌阳格式的库文件其文件扩展名应为 .lib**，非凌阳格式的库文件不能被使用。**

L0511: No FIND argument

命令行中的 FIND **命令无参变量。**

L0512: No ADD argument

命令行中的 ADD **命令无参变量。**

L0513: No REP argument

命令行中的 REP **命令无参变量。**

L0514: No DEL argument

命令行中的 DEL **命令无参变量。**

L0515: Can't identify the command...

命令行中的命令格式有误。

L0520: Read Lib Error

读指定的库文件有误。

L0521: Write Lib Error

写指定的库文件有误。

L0522: Can't find the module...

在库文件中未能找到指定的程序模块。

L0530: Can't open the file...

指定的库文件未能找到或打开，可能是文件名输入有误。

L0531: File destroyed:...

库文件被毁坏，或可能是文件名输入有误。

L0532: The bTag is valid. The Obj file could has been destroyed

要链接的目标文件或库文件可能已被毁坏。

L0540: The file ... is not SunplusLib

指定的文件不是凌阳格式的库文件，或库文件可能已被毁坏。

L0541: Obj Type is not Sunplus

指定的文件不是凌阳格式的目标文件，或目标文件可能已被毁坏。

参 考 文 献

1 **许海燕, 付炎. 嵌入式系统技术与应用. 北京: 机械工业出版社**, 2002

2 **房小翠, 王金凤. 单片机实用系统设计技术. 北京: 国防工业出版社**, 1999

3 **张洪润, 蓝清华. 单片机应用技术教程. 北京: 清华大学出版社**, 1999

4 **李建忠. 单片机原理及应用. 西安: 西安电子科技大学出版社**, 2002

5 **李伯成等. 微型计算机原理及应用. 西安: 西安电子科技大学出版社**, 1999

6 **薛钧义等. 凌阳** 16 **位单片机原理及应用. 北京: 北京航空航天大学出版社**, 2003

7 **张涵芳**. Intel 16 **位单片机. 北京: 北京航空航天大学出版社**, 2000

8 **张毅坤等. 单片微型计算机原理及应用. 西安：西安电子科技大学出版社**, 2001

9 **凌阳科技公司**. 16 **位单片机相关资料. 北阳电子**, 2002

10 **雷思孝, 李伯成**. SPCE061A 16 **位单片机原理及应用. 内部讲义. 西安：西安电子科技大学**, 2003

11 http//www.sunplus.com.tw

12 http//www.unsp.com.cn

13 http//www.21ic.com

14 http//www.mcu51.com